

JAVA:

устранение проблем



Чтение, отладка
и оптимизация
JVM-приложений

Лауренциу Спилкэ

В этой книге представлены практические методики исследования и улучшения незнакомого кода. Вы узнаете о том, как определять скрытые зависимости, выявлять главные причины критических сбоев и аварийных завершений приложений, а также интерпретировать неожиданные результаты. Осваивайте профилирование и отладку и начинайте исследовать, как в действительности работают Java-приложения.

Лауренциу Спилкэ — опытный разработчик Java и Spring, а также преподаватель технических дисциплин с большим стажем. Автор книг «Spring Start Here» и «Spring Security in Action».

Издание подойдет для Java-разработчиков средней квалификации.

Основные темы:

- как понять, что делает код, который вы впервые видите;
- анализ дампов кучи для обнаружения утечек памяти;
- мониторинг потребления ресурсов ЦП для оптимизации выполнения;
- использование дампов потоков для поиска и устранения взаимоблокировок;
- выявление недостатков в логике кода;
- обнаружение проблем, нерегулярно возникающих во время выполнения.

«На то, чтобы понять код, мы тратим больше времени, чем на его написание. Эта книга показывает, как рационально использовать рабочее время».

Брент Хонадел, Infor

«Наконец-то вышла книга, которая демонстрирует, как устранять проблемы и исследовать Java-приложения в производственной среде! Она определенно может спасти положение!»

Атул Шринивас Хом, SquareOne Insights

«Полное понимание внутренней работы Java-приложений».

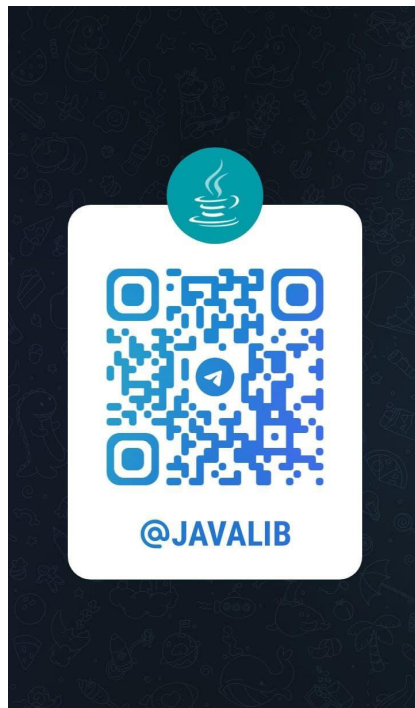
Фернандо Бернардино, Wise

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javalib>

Лауренциу Спилкэ

Java: устранение проблем

Чтение, отладка и оптимизация
JVM-приложений



Troubleshooting Java

READ, DEBUG, AND OPTIMIZE JVM
APPLICATIONS

LAURENȚIU SPILCĂ



MANNING
SHELTER ISLAND

Java: устранение проблем

Чтение, отладка и оптимизация
JVM-приложений

ЛАУРЕНЦИУ СПИЛКЭ

Москва, 2023

В этой книге рассматриваются простые и практичные методики исследования и улучшения незнакомого кода на языке программирования Java. Рассказывается о том, как эффективно использовать журналы для понимания поведения приложений; как применять методы профилирования для повышения эффективности при анализе проблем или изучении рабочих сред; как анализировать взаимодействие одних приложений с другими и осуществлять мониторинг системных событий, и обо многом другом.

Издание будет полезно начинающим и опытным разработчикам. Для чтения необходимо понимать основы языка Java.

Оглавление

Предисловие.....	10
Благодарности.....	12
Об этой книге.....	14
Об авторе	18
Об иллюстрации на обложке	18
 ЧАСТЬ I. Основы анализа кодовой базы.....	 19
Глава 1. Раскрытие секретов приложения.....	21
1.1. Как облегчить понимание работы приложения	22
1.2. Типовые сценарии для использования методик анализа.....	26
1.2.1. Выяснение причины вывода неожиданных результатов.....	27
1.2.2. Изучение конкретных технологий.....	32
1.2.3. Выяснение причин замедления	33
1.2.4. Исследование случаев аварийного завершения приложения.....	35
1.3. Зачем нужно читать эту книгу	38
1.4. Резюме	38
 Глава 2. Изучение логики приложения с помощью методик отладки	 40
2.1. Когда недостаточно просто проанализировать код	42
2.2. Анализ кода с помощью отладчика	45
2.2.1. Что такое трассировка стека выполнения, и как ее использовать.....	51
2.2.2. Перемещение по коду с использованием отладчика.....	56
2.3. Когда применения отладчика может оказаться недостаточно.....	64
2.4. Резюме	65
 Глава 3. Поиск главных причин возникновения проблемы с использованием расширенных методик отладки.....	 67
3.1. Минимизация времени анализа с помощью условных точек останова.....	68
3.2. Использование точек останова, которые не приостанавливают выполнение	73
3.3. Динамическое изменение сценария анализа	75
3.4. Повторное воспроизведение варианта анализа	79
3.5. Резюме	85

Глава 4. Удаленная отладка приложений.....	87
4.1. Что такое удаленная отладка.....	89
4.2. Анализ в удаленных рабочих средах	92
4.2.1. Сценарий.....	92
4.2.2. Выявление проблем в удаленных средах	94
4.3. Резюме	105
Глава 5. Максимальное использование журналов: инспектирование поведения приложения	107
5.1. Анализ проблем с использованием журналов	111
5.1.1. Использование журналов для идентификации исключений.....	112
5.1.2. Использование трассировок стека исключений для определения стороны, вызывающей метод.....	114
5.1.3. Измерение времени, затраченного на выполнение конкретной инструкции	116
5.1.4. Анализ проблем в многопоточных архитектурах.....	117
5.2. Реализация функций журналирования	119
5.2.1. Постоянно хранимые журналы	119
5.2.2. Определение уровней журналирования и использование рабочих сред для ведения журналов.....	121
5.2.3. Проблемы, возникающие при журналировании, и способы их устранения	129
5.3. Сравнение журналирования с удаленной отладкой.....	134
5.4. Резюме	136
ЧАСТЬ II. Глубокий анализ выполнения приложения	137
Глава 6.Выявление проблем потребления ресурсов с использованием методик профилирования	138
6.1. В каких случаях профилировщик оказывается полезным.....	139
6.1.1. Выявление аномального использования ресурсов	139
6.1.2. Как определить, какой код выполняется	141
6.1.3. Определение узких мест (замедлений) при выполнении приложения	141
6.2. Использование профилировщика.....	142
6.2.1. Установка и конфигурирование профилировщика VisualVM	142
6.2.2. Наблюдение за использованием ЦП и памяти.....	145
6.2.3. Обнаружение утечек памяти	156
6.3. Резюме	161
Глава 7. Поиск скрытых проблем с использованием методик профилирования.....	163
7.1. Выборка для наблюдения за выполняемым кодом.....	164

7.2. Профилирование с целью узнать, сколько раз выполнен метод ...	172
7.3. Использование профилировщика для идентификации SQL-запросов, выполняемых приложением	176
7.3.1. Использование профилировщика для извлечения SQL-запросов, не генерируемых фреймворком.....	176
7.3.2. Использование профилировщика для получения SQL-запросов, генерируемых фреймворком	182
7.3.3. Использование профилировщика для получения программно сгенерированных SQL-запросов	186
7.4. Резюме.....	190
Глава 8. Использование продвинутых инструментов визуализации для профилируемых данных.....	192
8.1. Выявление проблем в JDBC-соединениях	193
8.2. Изучение проектного решения кода приложения с использованием графов вызовов.....	206
8.3. Использование flame-графиков для обнаружения проблем с производительностью	209
8.4. Анализ запросов в NoSQL базы данных	212
8.5. Резюме	213
Глава 9. Анализ блокировок в многопоточных архитектурах.....	215
9.1. Мониторинг потоков с целью обнаружения блокировок.....	216
9.2. Анализ блокировок потоков.....	222
9.3. Анализ ожидающих потоков	231
9.4. Резюме	238
Глава 10. Анализ взаимоблокировок с помощью дампов потоков... 240	240
10.1. Получение дампа потоков	240
10.1.1. Получение дампа потока с использованием профилировщика	243
10.1.2. Генерация дампа потоков из командной строки	245
10.2. Чтение дампов потоков	249
10.2.1. Чтение дампов потоков в виде простого текста.....	249
10.2.2. Использование инструментальных средств для лучшего понимания дампов потоков	255
10.3. Резюме	257
Глава 11. Обнаружение проблем, связанных с использованием памяти, при выполнении приложения	260
11.1. Выборка и профилирование для выявления проблем с памятью	261
11.2. Использование дампов кучи для поиска утечек памяти.....	268

11.2.1. Получение дампа кучи	270
11.2.2. Чтение дампа кучи	275
11.2.3. Использование консоли OQL для запроса в дамп кучи	279
11.3. Резюме	286
ЧАСТЬ III. Поиск проблем в крупных системах.....	287
Глава 12. Анализ поведения приложений в крупных системах.....	288
12.1. Анализ обмена данными между сервисами.....	289
12.1.1. Использование проб HTTP-сервера для наблюдения HTTP-запросов	291
12.1.2. Использование проб HTTP-клиента для наблюдения HTTP-запросов, отправляемых приложением	293
12.1.3. Анализ событий низкого уровня в сокетах.....	295
12.2. Важность интегрированного мониторинга журналов.....	298
12.3. Использование средств развертывания для анализа	305
12.3.1. Использование инъекции критической ошибки для имитации трудновоспроизводимых проблем	307
12.3.2. Использование зеркалирования для обеспечения тестирования и выявления ошибок.....	309
12.4. Резюме	310
Приложение А. Необходимые инструментальные средства.....	312
Приложение В. Открытие проекта	313
Приложение С. Литература, рекомендуемая для дополнительного чтения.....	316
Приложение D. Понимание потоков Java.....	318
D.1. Что такое поток.....	318
D.2. Жизненный цикл потока	320
D.3. Синхронизация потоков	323
D.3.1. Синхронизированные блоки.....	323
D.3.2. Использование <code>wait()</code> , <code>notify()</code> и <code>notifyAll()</code>	326
D.3.3. Присоединение потоков.....	328
D.3.4. Блокировка потоков на определенное время	329
D.3.5. Синхронизация потоков с блокирующими объектами.....	331
D.4. Проблемы, наиболее часто возникающие в многопоточных архитектурах	332
D.4.1. Состояние гонки	332
D.4.2. Взаимоблокировки	333
D.4.3. Динамические (активные) взаимоблокировки	334
D.4.4. Голодание (зависание).....	335
D.5. Материал для дополнительного чтения	336

Приложение Е. Управление памятью в Java-приложениях	337
Е.1. Как JVM организует память приложения	338
Е.2. Стек, используемый потоками для хранения локальных данных	340
Е.3. Куча, которую приложение использует для хранения экземпляров объектов	346
Е.4. Метапространство – локация памяти для хранения типов данных	349
Предметный указатель	351

Предисловие

Чем в действительности зарабатывает на жизнь разработчик программного обеспечения? Чаще всего ответом на этот вопрос становится фраза «реализацией программного обеспечения». Но что она означает? Это только лишь написание исходного кода? Ну уж нет. Хотя код действительно является результатом работы каждого разработчика, собственно написание исходного кода занимает всего лишь малую часть его рабочего времени. А большую часть времени разработчик программного обеспечения на самом деле использует для создания проектных решений, чтения существующего кода, понимания, как он выполняется, и изучения новых методик программирования и проектирования ПО. Написание исходного кода – это результат успешного выполнения разработчиком всех перечисленных выше задач. Таким образом, программист основную часть своего времени тратит на изучение существующих решений, а не на фактическое написание кода, реализующего новые возможности.

В конечном счете чистое кодирование имеет ту же цель: научить разработчиков писать легко читаемые программные решения. Разработчики прекрасно понимают, что гораздо полезнее с самого начала написать легко читаемое решение, чем в дальнейшем тратить время, пытаясь понять его. Но нужно честно признаться в том, что не все решения являются доступными для быстрого понимания. Нам всегда будут встречаться сценарии, в которых потребуется понимание выполнения некоторой незнакомой функциональной возможности.

Действительность такова: разработчики программного обеспечения тратят огромное количество времени, изучая и анализируя, как работают приложения. Они читают и исследуют код в кодовых базах приложений, а также установленные зависимости, чтобы понять до конца, почему что-то не работает так, как предполагалось. Иногда разработчики читают код только для того, чтобы выявить или лучше понять конкретную зависимость. Во многих случаях чтения кода недостаточно, и необходимо найти другие (иногда более сложные) способы исследования того, что именно делает конкретное приложение. Чтобы понять, как рабочая среда воздействует на приложение или как конкретный экземпляр JVM выполняет Java-приложение, можно воспользоваться сочетанием средств профилирования, отладки и анализа журналов. Если вам хорошо известны все возможные варианты и методики выбора наиболее подходящих инструментов, то вы сэкономите драгоценное время. Следует помнить о том, что именно на это разработчики тратят большую часть времени. Эта часть процесса разработки может стать весьма продуктивной.

Я написал эту книгу, чтобы помочь людям оптимизировать процесс анализа трудностей и проблем при разработке программного обеспечения.

В ней вы найдете наиболее важные методики анализа с соответствующими практическими примерами. В книге рассматривается отладка, профилирование, использование журналов и эффективное объединение этих методик. На протяжении всей книги я буду давать полезные советы и описывать приемы, которые помогут вам достичь большей продуктивности и быстрее устранять проблемы (даже самые сложные). Другими словами, главная цель этой книги – сделать вас более эффективным разработчиком.

Надеюсь, книга окажется для вас чрезвычайно полезной и поможет гораздо быстрее обнаруживать главные причины анализируемых проблем.

Благодарности

Создание этой книги было бы невозможным без множества профессионалов и дружелюбных людей, которые помогали мне на протяжении всего процесса ее написания.

Я хочу сказать большое спасибо моей жене Даниэле, которая была рядом со мной, делилась важными мнениями, постоянно поддерживала и воодушевляла меня. Также хотелось бы выразить особую благодарность всем коллегам и друзьям, чьи ценные советы помогли мне с самым первым вариантом оглавления и предварительным планом.

Я хотел бы поблагодарить весь коллектив издательства Manning за их огромную помощь в создании полезного ресурса. Особая благодарность Марине Майклс (Marina Michaels), Нику Уоттсу (Nick Watts) и Жан-Франсуа Морену (Jean-François Morin) за их невероятную поддержку и профессионализм. Их советы сделали эту книгу более полезной. Также благодарю менеджера проекта Дейдру Хайам (Deirdre Hiam), редактора Мишель Митчелл (Michele Mitchell) и корректора Кэти Теннант (Katie Tennant).

Спасибо моей подруге Иоане Гёц (Ioana Göz) за создание рисунков для этой книги. Она превратила мои мысли в своеобразные комиксы, которые вы будете видеть на протяжении всей книги.

Я также хотел бы поблагодарить всех, кто рецензировал рукопись и предоставил полезные отзывы, которые помогли мне улучшить содержание этой книги. Хотелось бы особо отметить рецензентов из издательства Manning; это Алекс Гау (Alex Gout), Алекс Зурофф (Alex Zuroff), Амрах Умудлу (Amrah Umudlu), Ананд Натарайан (Anand Natarajan), Андрес Дамиан Сакко (Andres Damian Sacco), Андрий Стосик (Andriy Stosyk), Аиндья Бандопадхьяй (Anindya Bandopadhyay), Атул Шринивас Хот (Atul Shriniwas Khot), Бекки Хьюитт (Becky Huett), Бонни Малек (Bonnie Malec), Брент Хонадел (Brent Honadel), Карл Хоуп (Carl Hope), Кэтэлин Матей (Cătălin Matei), Кристофер Карделл (Christopher Kardell), Сисеро Сандона (Cicero Zandona), Козимо Дамиано Прете (Cosimo Damiano Prete), Дэниел Р. Карл (Daniel R. Carl), Дешуанг Танг (Deshuang Tang), Фернандо Бернардино (Fernando Bernardino), Габор Хайба (Gabor Hajba), Гаурав Тули (Gaurav Tuli), Джампьеро Грантелла (Giampiero Granatella), Гиорги Циклаури (Giorgi Tsiklauri), Говинда Самбамурти (Govinda Sambamurthy), Халил Каракёзе (Halil Karaköse), Уго Фигуэредо (Hugo Figueiredo), Якопо Бисчелла (Jacopo Biscella), Джеймс Р. Вудрафф (James R. Woodruff), Джейсон Ли (Jason Lee), Джавид Асгаров (Javid Asgarov), Жан-Баптист Банг Нтеме (Jean-Baptiste Bang Nteme), Йерун ван Вильденбург (Jeroen van Wilgenburg), Джоэль Каплин (Joel Caplin), Юрг Марти (Jürg Marti), Кжиштоф Камичек (Krzysztof Kamyczek), Латиф Бенззине (Latif Benzzine), Леонардо Гомес да Сильва (Leonardo Gomes da Silva),

Маной Редди (Manoj Reddy), Маркус Гезелле (Marcus Geselle), Мэтт Деймел (Matt Deimel), Мэтт Уэлки (Matt Welke), Майкл Колесидис (Michael Kolesidis), Майкл Уолл (Michael Wall), Михал Овсяк (Michal Owsiak), Оливер Кортен (Oliver Korten), Олубунми Огунсан (Olubunmi Ogunsan), Паоло Брунасти (Paolo Brunasti), Петер Сабош (Peter Szabós), Прабхути Пракаш (Prabhuti Prakash), Райеш Баламохан (Rajesh Balamohan), Райеш Моханан (Rajesh Mohanan), Равиш Шарма (Raveesh Sharma), Рубен Гонсалес-Рубио (Ruben Gonzalez-Rubio), Абуду Самаду-Сапе (Aboudou SamadouSare), Симеон Лейзерзон (Simeon Leyzerzon), Симоне Кафиеро (Simone Cafiero), Сраванти Редди (SravanthiReddy), Света Нату (Sveta Natu), Тан Ви (Tan Wee), Тануй Шрофф (Tanuj Shroff), Тревис Нелсон (Travis Nelson), Яков Боглев (Yakov Boglev) и Юрий Клайман (Yuri Klayman). Также спасибо друзьям, которые помогали мне советами: Марии Кицу (Maria Chițu), Адриану Бутуругэ (Adrian Buturugă), Мирче Вакариуку (Mircea Vacariuc), Кэтэлин Матей (Cătălin Matei).

Об этой книге

Для кого предназначена эта книга

Поскольку вы открыли эту книгу, я предполагаю, что вы разработчик, использующий один из языков JVM. Возможно, вы пишете на Java, но, вероятно, также применяете Kotlin или Scala. Независимо от используемого языка JVM вы найдете полезным содержимое этой книги. Она обучит вас важным методикам анализа, которые можно применять для выявления главных причин проблем (т. е. ошибок и дефектов), и продемонстрирует легкий способ изучения новых технологий. Как разработчик программного обеспечения, вы, вероятно, уже заметили, сколько времени тратится на понимание того, что именно делает то или иное приложение. Как и другие разработчики, вы, вероятнее всего, тратите больше времени на чтение исходного кода, отладку и/или использование функций журналирования, чем на написание кода. Так почему бы не сделать более эффективным то, чем вы занимаетесь чаще всего в течение рабочего дня?

В этой книге рассматриваются следующие темы (с соответствующими практическими примерами):

- простые и усовершенствованные методики отладки;
- эффективное использование журналов для понимания поведения приложений;
- профилирование ЦП и потребления ресурса памяти;
- профилирование для выявления действительно выполняющегося кода;
- профилирование для понимания того, как работает приложение с постоянно хранимыми данными;
- анализ взаимодействия приложений с другими приложениями;
- мониторинг системных событий.

Независимо от вашего практического опыта вы найдете эту книгу полезной при изучении новых методик анализа, а если вы уже являетесь весьма опытным разработчиком, то сможете освежить свои знания.

Предварительное условие для чтения: понимание основ языка Java. Я преднамеренно написал все примеры на языке Java (даже если они применимы для любого языка JVM), чтобы обеспечить логическую согласованность. Если вы знаете язык Java на базовом уровне (классы, методы, основные инструкции, такие как условные и циклические выражения, и объявление переменных), то вполне способны понять весь материал, представленный здесь.

Как организована эта книга: общая схема

Книга разделена на три части, содержащие в общей сложности 12 глав. Начнем с рассмотрения (в первой части) методик отладки. Мы рассмотрим и применим на практике простые и более усовершенствованные методики отладки, а также обсудим, где можно использовать их, чтобы сэкономить время в различных сценариях анализа. Я выбрал в качестве начальной темы именно отладку, потому что обычно это самый первый этап анализа того, как некоторое функциональное свойство приложения ведет себя во время разработки. Некоторые люди спрашивали, почему я не начал книгу с методики журналирования, поскольку это первоначальная методика анализа возникновения проблем в производственном контексте. Это действительно так, но разработчик должен работать с отладчиком, когда начинает реализацию функциональных средств и возможностей, поэтому я решил, что лучше расположить главы так, чтобы сначала рассматривались методики отладки.

В первой главе обсуждается важность методик анализа, рассматриваемых в книге, и приводится общий план их изучения. В главах 2, 3 и 4 все внимание сосредоточено на отладке и обучении соответствующим практическим навыкам: от добавления простой точки останова до отладки приложений в удаленных рабочих средах. В главе 5, последней в части I, рассматривается методика журналирования. Отладка и использование журналов – это самые простые (и наиболее часто используемые) методики анализа при создании приложения.

Во второй части книги рассматриваются методики профилирования. Широко распространено мнение о том, что профилирование является более усовершенствованной методикой, но в современных приложениях используется реже, чем отладка и проверка журналов. Я согласен с тем, что профилирование – более усовершенствованная методика, и показываю, как можно использовать многие методы профилирования для повышения эффективности при анализе проблем в современных JVM-приложениях или при изучении рабочих сред, считающихся весьма важными.

В главе 6, открывающей вторую часть книги, рассматриваются способы, позволяющие определить, содержит ли приложение критические ошибки в управлении ЦП и ресурсами памяти. В главе 7 эта тема раскрывается более подробно, здесь показано, как выявить ту часть приложения, в которой возникают конкретные причины таких задержек, и как увидеть, что именно выполняет приложение в конкретные интервалы времени. В главах 6 и 7 используется бесплатное инструментальное средство VisualVM. Глава 8 продолжает рассмотрение темы главы 7, представляя более развитые инструментальные средства визуализации, которые обычно доступны только в лицензионных комплектах инструментов профилирования. Для исследования подробностей, обсуждаемых в этой главе, используется JProfiler, не являющийся бесплатным программным средством.

В главах 9 и 10 основное внимание уделено более тонким методикам профилирования. Вы овладеете практическими навыками, позволяющими

сэкономить время при устранении проблем, глубоко скрытых в многопоточной архитектуре при выполнении любого приложения. Глава 11 завершает часть II, демонстрируя, как выполняется анализ управления памятью в приложении.

Книгу завершает часть III, состоящая только из одной главы 12. В ней мы перейдем за границы приложения, чтобы рассмотреть анализ проблем во внешней системе, состоящей из нескольких (возможно, многих) приложений.

Я рекомендую изучать главы именно в таком порядке, хотя в каждой главе рассматривается отдельная тема. Так что, если вас интересует какая-либо конкретная тема, то можно сразу же перейти к чтению соответствующей главы. Например, если необходима подробная информация об анализе проблем при управлении памятью, то можно начать прямо с главы 11.

Примеры исходного кода

Книга содержит множество примеров исходного кода как в форме пронумерованных листингов, так и в виде отдельной строки или нескольких строк в обычном тексте. В обоих случаях исходный код отформатирован с использованием такого шрифта постоянной ширины для отделения его от обычного текста. Иногда код также дополнительно выделяется *полужирным шрифтом*, чтобы специально выделить фрагмент кода, который изменился по сравнению с предыдущими этапами (примерами) в текущей главе, например при добавлении нового функционального средства (свойства) в существующую строку кода.

Во многих случаях изначальный исходный код был переформатирован: добавлены символы перехода на новую строку и изменено выравнивание для адаптации к доступному пространству на странице этой книги. В редких случаях, когда даже такие меры оказались недостаточными, в листинги включены маркеры продолжения строки (➡). Кроме того, комментарии часто удалялись из исходного кода, если этот код подробно описывался в тексте. Многие листинги предваряются аннотациями к коду, которые особо выделяют наиболее важные концепции.

Выполняемые фрагменты кода из версии liveBook (онлайновой) этой книги можно получить на сайте <https://livebook.manning.com/book/troubleshooting-java>. Полный код примеров из книги доступен для загрузки с сайта издательства Manning www.manning.com.

Форум обсуждения liveBook

Приобретение книги «Java: устранение проблем» подразумевает свободный (бесплатный) доступ к liveBook, платформе чтения в режиме онлайн издательства Manning. Используя эксклюзивные возможности обсуждения платформы liveBook, вы можете добавлять комментарии ко всей книге в целом или к конкретным разделам и даже абзацам. Можно с легкостью соз-

давать собственные заметки, задавать технические вопросы и отвечать на них, а также получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, перейдите по адресу <https://livebook.manning.com/book/troubleshooting-java/discussion>. О форумах издательства Manning и правилах их модерации можно узнать более подробно здесь: <https://livebook.manning.com/discussion>.

Обязательство издательства Manning перед читателями состоит в том, чтобы предоставить место, где может состояться содержательный диалог между отдельными читателями, а также между читателями и автором. Это не является обязательством какой-либо конкретной степени участия со стороны автора, чей вклад в форум остается добровольным (и неоплачиваемым). Мы предлагаем попробовать задать автору несколько сложных вопросов, чтобы он не потерял интерес к форуму. Форум и архивы предыдущих обсуждений будут доступны на веб-сайте издателя, пока книга находится в печати.

Доступ к автору в онлайн-режиме

Рекомендую поддерживать со мной связь в режиме онлайн. Вы непременно обнаружите многочисленные качественные учебные материалы, связанные с приложениями из книги «Java: устранение проблем», на моем YouTube-канале: youtube.com/c/laurentiuspilca. Вы также можете следить за моими сообщениями в Твиттере, @laurspilca.

Об авторе



Лауренциу Спилкэ (Laurențiu Spilcă) – целеустремленный руководитель разработки ПО и инструктор в компании Endava (Лондон), где он отвечает за руководство и консультирование нескольких проектов в разных странах Европы, США и Азии. Он занимается разработкой программного обеспечения с 2007 г. Лауренциу считает важным не только создавать высококачественное программ-

ное обеспечение, но и делиться знаниями и помогать другим повышать квалификацию. Это кредо стало мотивацией для разработки и преподавания курсов, связанных с технологиями Java, а также для проведения презентаций и рабочих семинаров. Лауренциу также является автором книги «Spring Security in Action» (Manning, 2020 г.), а недавно он закончил написание книги «Spring Start Here» (Manning, 2021 г.).

Об иллюстрации на обложке

На обложке книги «Java: устранение проблем» размещен рисунок «Homme de l'Istrie», или «Мужчина из Истрии», взятый из коллекции (изображений одежды) Жака Грассе из Сен-Совёр (Jacques Grasset de Saint-Sauveur), опубликованный в 1797 г. Каждая иллюстрация тщательно прорисована и раскрашена от руки.

В те времена легко было определить, где живут люди, какова их профессия или положение в обществе, просто по их одежде. Издательство Manning воздает должное изобретательности и инициативности компьютерного бизнеса, создавая обложки книг на основе богатого разнообразия региональной культуры многовековой давности, оживленные фотографиями из таких коллекций, как эта.

Часть I



Основы анализа кодовой базы

При разработке программного обеспечения работа с реальными приложениями часто подразумевает анализ выполнения их кода. Вы должны понимать поведение приложения при устранении проблем, а также при реализации новых функциональных возможностей. Для этой цели используется несколько методик: отладка, журналирование, профилирование и т. д., которые мы подробно рассмотрим в этой книге.

В части I мы начнем с методик, с которыми разработчик имеет дело в первую очередь, это отладка и работа с журналами. При разработке приложения программист должен часто применять отладку. Например, имеется небольшой фрагмент кода, и вам необходимо понять, как он работает. Вы используете отладчик для временной приостановки выполнения приложения и погружаетесь в исследование того, как это приложение обрабатывает данные. Затем, если приложение работает в некоторой среде, можно воспользоваться многочисленными журналами, которые предоставляют требуемые сведения о том, где именно могла возникнуть проблема.

В главе 1 обсуждается необходимость освоения методик анализа и приводится общий обзор тех методик, которые будут подробно описаны в остальной части книги. Затем мы рассмотрим эти методики в том порядке, в котором они используются разработчиком. В главах 2–4 подробно обсуждается отладка. В главе 5 рассматриваются самые важные подробности реализации и использования журналов при выполнении анализа.



Начало разработки приложения



Чтение кода

```
profile.ifPresentOrElse(  
    p -> {  
        healthMetric.setProfile(p);  
        healthMetricRepository.save(healthMetric);  
    },  
    () -> {  
        throw new NonExistentHealthProfileException();  
    });
```

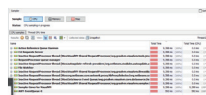


Отладка



**Анализ
журналов**

Профилирование



Глава 1

Раскрытие секретов приложения

Темы этой главы:

- определение методики анализа кода;
- какие методики анализа кода используются для понимания работы Java-приложений.

У разработчика программного обеспечения (ПО) много обязанностей, и в большинстве своем они зависят от понимания кода, с которым приходится работать. Разработчики ПО тратят много рабочего времени на анализ кода, чтобы найти способы устранения проблем, реализации новых функциональных возможностей и даже для изучения новых технологий. Но поскольку время дорого, разработчикам необходимы эффективные методики анализа для обеспечения продуктивности. Поэтому главная тема этой книги – научиться эффективно исследовать и анализировать код.

ПРИМЕЧАНИЕ. Обычно разработчики программного обеспечения тратят больше времени на понимание того, как работает программа, чем на написание кода реализации новых функциональных возможностей или на исправление ошибок.



Разработчики ПО часто используют слово «отладка» (debugging) для обозначения любой методики анализа, но это всего лишь один из разнообразных инструментов, доступных для исследования логики, реализованной в виде кода. Хотя отладка в действительности должна означать «поиск проблем и их устранение», разработчики применяют это слово, чтобы обозначить различные задачи анализа работы кода:

- изучение новой рабочей среды (framework);
- поиск главной причины возникновения проблемы;
- понимание существующей логики с целью ее расширения с помощью новых возможностей.

1.1. Как облегчить понимание работы приложения

Во-первых, важно понять, что из себя представляет анализируемый код и как разработчики создали его. В этом разделе мы рассмотрим несколько часто встречающихся сценариев, в которых можно применить методики, описанные в данной книге.

Я определяю термин «анализ кода» как процесс анализа конкретного поведения конкретного средства/свойства программного обеспечения. Возможно, вы удивитесь: «Зачем такое обобщенное определение? Какова цель анализа?» В ранней истории разработки ПО просмотр кода имел одну четко определенную цель: обнаружение и исправление программных ошибок (bugs). Именно поэтому многие разработчики продолжают использовать термин *debugging* (отладка) для всех методик анализа. Давайте более внимательно рассмотрим, как образовался термин *debug*:

de-bug = «уничтожение насекомых» (bugs), устранение ошибок.

В настоящее время во многих случаях мы продолжаем отлаживать (de-bug) приложения для поиска и исправления ошибок. Но, в отличие от раннего периода разработки ПО, современные приложения стали более сложными. В большинстве случаев разработчикам приходится анализировать работу конкретного свойства ПО, просто изучая соответствующую технологию или библиотеку. Отладка уже не означает только лишь обнаружение конкретной проблемы, в это понятие также входит правильное понимание поведения программы (см. рис. 1.1, также см. <http://mng.bz/M012>).



Рис. 1.1. Анализ кода – это не только поиск проблем в программном обеспечении. В наше время приложения стали сложными. Мы часто используем методики анализа для понимания поведения приложения или просто для изучения новых технологий

Почему необходимо анализировать код в приложениях:

- для обнаружения конкретной проблемы;
- для понимания того, как работает конкретное программное средство, чтобы получить возможность его развития/усовершенствования;
- для изучения конкретной технологии или библиотеки.

Многие разработчики анализируют код еще и ради развлечения, потому что исследование работы кода – весьма увлекательное занятие. Иногда это может привести и к разочарованию, но ничто не сравнится с чувством, воз-

никающим при обнаружении главной причины проблемы или при полном понимании, как работает программа (см. рис. 1.2).

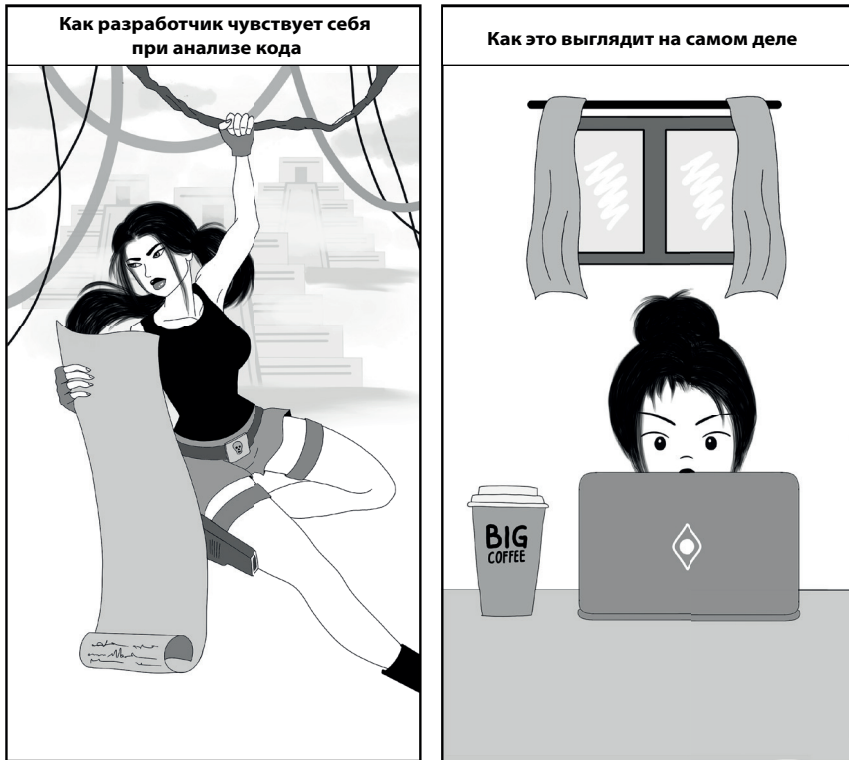


Рис. 1.2. Анализ кода не требует больших физических усилий, но отладка иногда заставляет разработчика чувствовать себя как Лара Крофт или Индиана Джонс. Многие разработчики наслаждаются этим неповторимым ощущением при решении головоломки во время устранения проблемы в программном коде

Существуют разнообразные методики, которые можно применить для анализа поведения ПО. Как мы увидим далее в этой главе, разработчики (особенно начинающие) часто неверно считают отладку равнозначной использованию отладчика. Отладчик – это программа, которую можно использовать для чтения и облегчения понимания исходного кода приложения, как правило, временно приостанавливая выполнение конкретных инструкций и начиная пошаговый проход по коду. Это общепринятый способ анализа поведения ПО (и обычно изучаемый разработчиком в первую очередь). Но это не единственная методика, которую можно использовать, и она помогает далеко не в каждом сценарии. Мы подробно рассмотрим стандартный и более продвинутый способ применения отладчика в главах 2 и 3. На рис. 1.3 представлены разнообразные методики анализа, которые вы освоите, читая эту книгу.

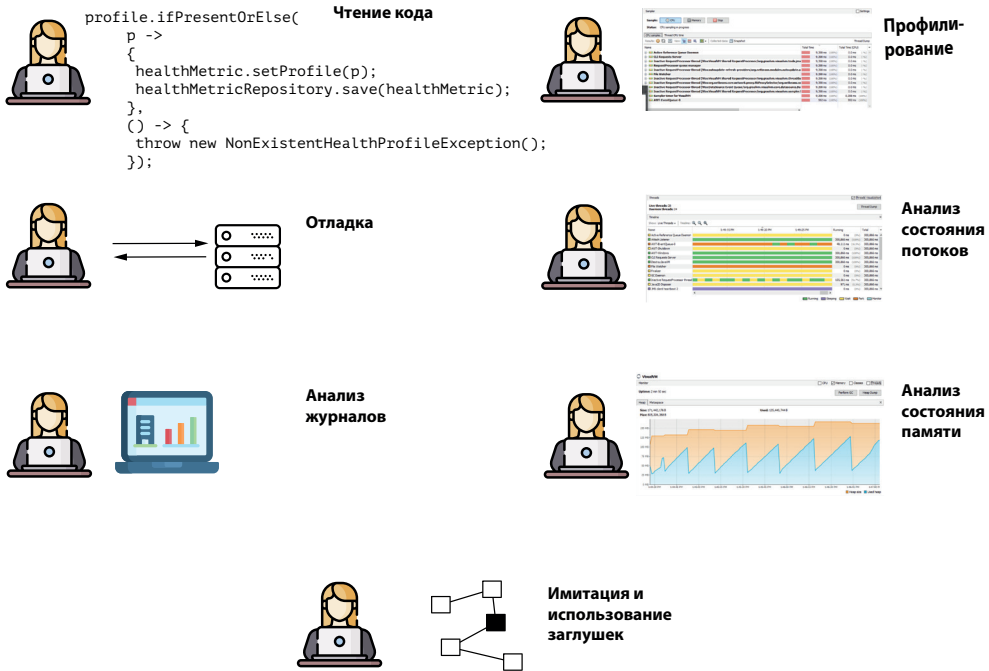


Рис. 1.3. Методики анализа кода. В зависимости от конкретного случая разработчик может выбрать одну или несколько методик, чтобы понять, как работает то или иное программное средство

Когда разработчик исправляет ошибку, он тратит большую часть времени на то, чтобы понять конкретное функциональное свойство. Вносимые изменения иногда в итоге сводят проблему к единственной строке кода – пропущенное условное выражение, отсутствующая инструкция или неправильно использованный оператор. Поэтому большую часть рабочего времени разработчика занимает не написание кода, а изучение того, как работает приложение.

В некоторых случаях простого чтения кода достаточно, чтобы понять его, но чтение кода совсем не похоже на чтение книги. При чтении кода мы не читаем удобные короткие абзацы, записанные в логическом порядке сверху вниз, а вместо этого переходим от одного метода к другому, из одного файла в другой, и иногда возникает ощущение, что мы оказались в огромном лабиринте и потерялись. (По этой теме я рекомендую превосходную книгу Фельенн Херманс (Feliene Hermans) «The Programmer’s Brain» (издательство Manning, 2021 г.).)

В большинстве случаев исходный код написан так, что читать его совсем не просто. Да, я знаю, о чем вы подумали: это необходимость. И я согласен с вами. В настоящее время мы изучаем многочисленные шаблоны и принципы проектирования исходного кода и способы устранения запутан-

ности в нем, но давайте признаемся честно: разработчики слишком часто продолжают неправильно применять эти принципы и правила. Кроме того, написанные ранее приложения обычно вообще не следуют подобным принципам просто потому, что в те далекие времена написания старых приложений такие принципы не существовали. А необходимость анализа этого кода остается.

Рассмотрим листинг 1.1. Предположим, что вы обнаружили этот фрагмент кода, пытаетесь обнаружить главную причину проблемы в приложении, над которым вы работаете. Такой код определенно требует рефакторинга. Но, прежде чем вы сможете реорганизовать его, необходимо понять, что именно он делает. Мне хорошо знакомы некоторые разработчики, которые могут прочесть такой код и сразу же понять, как он работает, но я не из их числа.

Листинг 1.1. Сложная для чтения и понимания логика, требующая использования отладчика

```
public int m(int f, int g) {
    try {
        int[] far = new int[f];
        far[g] = 1;
        return f;
    } catch(NegativeArraySizeException e) {
        f = -f;
        g = -g;
        return (-m(f, g) == -f) ? -g : -f;
    } catch(IndexOutOfBoundsException e) {
        return (m(g, 0) == 0) ? f : g;
    }
}
```

Для упрощения понимания логики в листинге 1.1 используется отладчик (debugger) – инструментальное средство, позволяющее временно приостанавливать выполнение в конкретных строках и вручную выполнять каждую инструкцию, одновременно наблюдая, как изменяются данные, т. е. постепенно проходить по каждой строке, чтобы увидеть, как она работает при конкретно заданных входных данных (более подробно мы обсудим это в главе 2). При наличии небольшого опыта и с помощью некоторых приемов (которые мы рассмотрим в главах 2 и 3), проанализировав этот код несколько раз, вы поймете, что он вычисляет максимальное значение из двух переданных ему входных данных. Код является частью проекта da-ch1-ex1, представленного в этой книге.

Некоторые сценарии не позволяют свободно перемещаться по коду или затрудняют такое перемещение. В настоящее время для большинства приложений существуют зависимости, например от библиотек или рабочих

сред. Чаще всего, даже если вы получили доступ к исходному коду (при использовании зависимости от ПО с открытым исходным кодом), продолжает существовать затруднение при проходе по исходному коду, определяющему логику рабочей среды. Иногда даже неизвестно, с чего начать. В подобных случаях необходимо применять различные методики для понимания работы приложения. Например, можно воспользоваться инструментом профилирования (описанным в главах 6–9), чтобы определить, какой код выполняется, прежде чем решить, где начать анализ.

Другие сценарии не предоставляют возможности получения в свое распоряжение выполняющегося приложения. В некоторых случаях приходится анализировать проблему, из-за которой приложение завершается аварийно. Если приложение, в котором возникли проблемы, прекратило работу, находится в производственной эксплуатации, то необходимо как можно быстрее вернуть его в рабочее состояние. Поэтому требуется собрать все подробности и использовать их для обнаружения проблемы и улучшить приложение, чтобы в будущем подобные проблемы не возникали. Такой анализ, основанный на собранных данных после аварийного завершения приложения, называется «анализом после аварии» (*postmortem investigation*). В подобных случаях можно использовать журналы, дампы памяти или дампы потоков, применяя инструментальные средства устранения проблем, которые рассматриваются в главах 11 и 12.

1.2. Типовые сценарии для использования методик анализа

Обсудим некоторые обобщенные сценарии для применения методик анализа кода. Мы должны рассмотреть несколько типичных случаев из реально эксплуатируемых приложений и проанализировать их, особенно подчёркивая важность тематики этой книги:

- чтобы понять, почему конкретный фрагмент кода или элемент ПО при выполнении выдает результат, отличающийся от ожидаемого;
- чтобы узнать, как работают технологии, используемые приложением как зависимости;
- чтобы определить причины проблем с производительностью, например замедления работы приложения;
- чтобы найти главные причины в тех случаях, когда приложение внезапно завершается.

Для каждого перечисленного выше случая вы найдете одну или несколько методик, полезных для анализа логики приложения. В дальнейшем мы более подробно рассмотрим эти методики и покажем, как их использовать, на конкретных примерах.

1.2.1. Выяснение причины вывода неожиданных результатов

Наиболее часто встречающимся сценарием, требующим анализа кода, является наличие некоторой логики, результатом выполнения которой считаются выходные данные, отличающиеся от ожидаемых. Проблема может показаться простой, но ее решение не всегда оказывается легким.

Сначала определим понятие «выходные данные» (output). В приложении этот термин может иметь несколько определений. Выходными данными может быть некоторый текст в консоли приложения или некоторые записи, измененные в базе данных. Можно рассматривать как выходные данные HTTP-запрос, передаваемый в другую систему, или некоторые данные, отправленные в HTTP-ответе на запрос клиента.

ОПРЕДЕЛЕНИЕ. Любой результат выполнения фрагмента логики, который в итоге может привести к изменению данных, обмену информацией или выполнению действия в другом компоненте или системе, является выходными данными (output).

Как проанализировать случай, когда конкретная часть приложения не выдает ожидаемый результат выполнения? Для этого мы выбираем правильную методику на основе ожидаемых выходных данных. Рассмотрим несколько примеров.

Сценарий 1: простой случай

Предположим, что приложение должно вставлять некоторые записи в базу данных. Но приложение добавляет только часть записей. То есть вы рассчитываете обнаружить в базе данных больше данных, чем в действительности предоставлено приложением.

Простейший способ анализа этой ситуации – использование отладчика для пошагового просмотра выполнения кода и понимания его работы (см. рис. 1.4). Об основных функциональных возможностях отладчика вы узнаете в главах 2 и 3. Отладчик добавляет точку останова для временной приостановки выполнения приложения в заданной по вашему выбору строке кода, после чего вы получаете возможность продолжения выполнения вручную (в пошаговом режиме). Вы выполняете инструкции кода поочередно, поэтому можете наблюдать, как изменяются значения переменной и вычисляются выражения «на лету».

Это самый простой сценарий, и, узнав, как правильно применять все действительно необходимые возможности отладчика, вы можете сразу же находить решения по устранению подобных проблем. К сожалению, другие случаи более сложны, и отладчик не всегда становится инструментом, достаточным для решения головоломки и поиска истинной причины возникновения проблемы.



СОВЕТ. Во многих случаях одной методики анализа недостаточно для понимания поведения приложения. Потребуется совместное применение различных методик для ускорения понимания более сложного поведения.

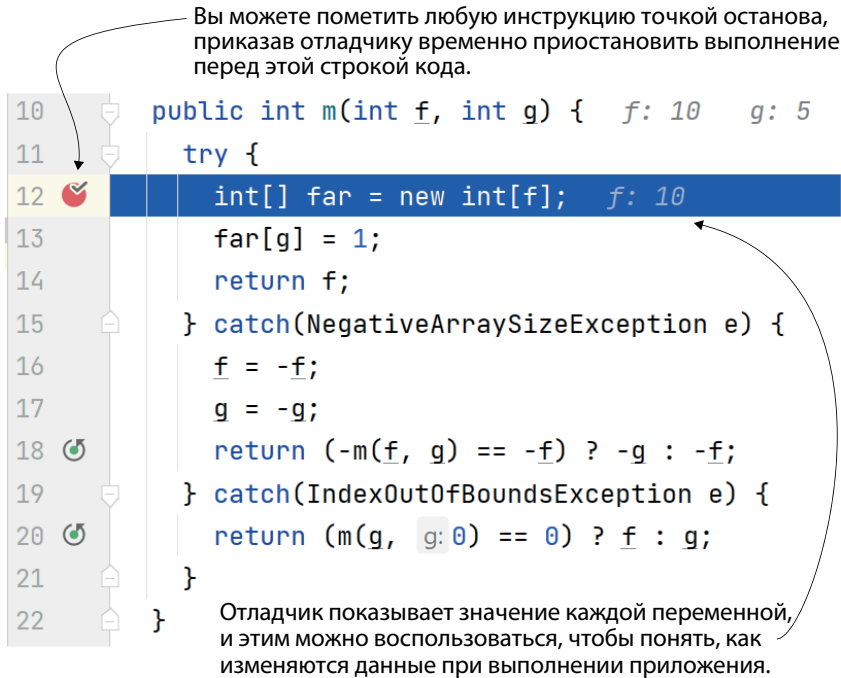


Рис. 1.4. Используя отладчик, вы можете временно приостановить выполнение перед конкретной инструкцией, а затем наблюдать, как логика приложения изменяет данные, вручную выполняя инструкции в пошаговом режиме

Сценарий 2: случай «в каком месте я должен начать отладку?»

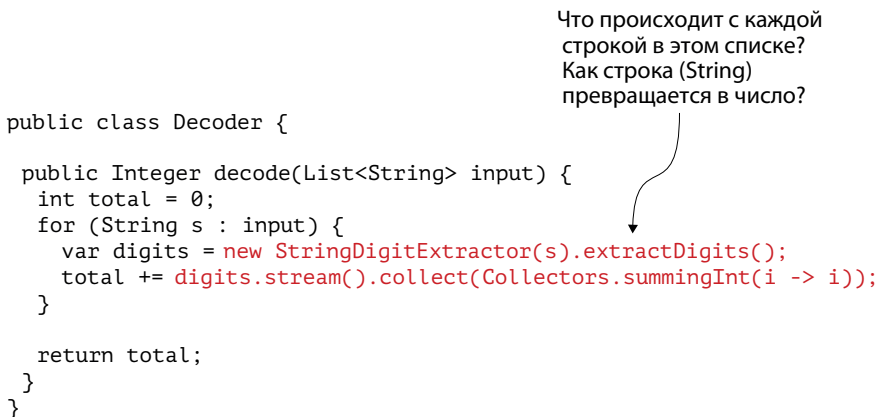
Иногда возможность использования отладчика отсутствует просто потому, что вы не знаете, что именно нужно отлаживать. Предположим, что приложение представляет собой сложный сервис с многочисленными строками исходного кода. Вы анализируете случай, в котором приложение не сохраняет ожидаемые записи в базе данных. Это определенно проблема выходных данных, но среди тысяч строк кода, определяющих приложение, вы не можете найти ту часть, которая реализует функциональность, требующую исправления.

Я вспоминаю коллегу, который анализировал именно такой случай. В стрессовой ситуации, когда невозможно было найти начальную позицию

для отладки, он воскликнул: «Хотел бы я, чтобы в отладчиках существовала функция добавления точки останова во все строки приложения, чтобы можно было увидеть, что оно в действительности использует».

Высказывание моего коллеги было забавным, но наличие такой возможности в отладчике не стало бы решением. Мы располагаем другими способами устранения подобной проблемы. Вероятнее всего, вы бы сузили возможный набор строк, в которые можно было бы добавить точку останова, используя профилировщик.

Профилировщик (profiler) – это инструментальное средство, которым можно воспользоваться, чтобы определить, какая часть кода выполняется во время работы приложения (см. рис. 1.5). Это превосходная возможность для рассматриваемого сценария, потому что она должна указать нам направление поиска того места, с которого следует начать анализ с помощью отладчика. Применение профилировщика будет рассматриваться в главах 6–9, где вы узнаете о нем гораздо больше, а не только о простой возможности наблюдать за кодом во время выполнения.



Что происходит с каждой строкой в этом списке?
Как строка (String) превращается в число?

```
public class Decoder {

    public Integer decode(List<String> input) {
        int total = 0;
        for (String s : input) {
            var digits = new StringDigitExtractor(s).extractDigits();
            total += digits.stream().collect(Collectors.summingInt(i -> i));
        }

        return total;
    }
}
```

Рис. 1.5. Определение выполняемого кода с помощью профилировщика. Если вы не знаете, где начать отладку, то профилировщик может помочь в идентификации выполняемого кода и указать направление поиска того места, где можно воспользоваться отладчиком

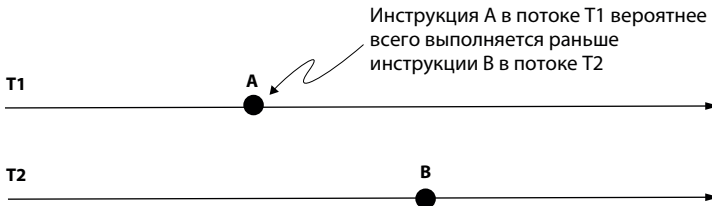
Сценарий 3: многопоточное приложение

Ситуация становится еще более сложной при работе с логикой, реализованной с использованием нескольких потоков, т. е. многопоточной архитектуры. Во многих подобных случаях применение отладчика не является приемлемым вариантом, потому что многопоточные архитектуры чувствительны к вмешательству в их работу.

Другими словами, поведение приложения изменяется, когда вы используете отладчик. Разработчики называют эту характеристику выполнением Гейзенберга (Heisenberg execution) или Гейзенбагом (Heisenbug)

(см. рис. 1.6) по имени физика XX века Вернера Гейзенберга (Werner Heisenberg), сформулировавшего принцип неопределенности, согласно которому при внешнем воздействии на частицу ее поведение изменяется так, что вы не можете точно предсказать одновременно ее скорость и положение в пространстве (<https://plato.stanford.edu/entries/qt-uncertainty/>). Многопоточная архитектура может изменять свое поведение, если вы как-либо воздействуете на нее точно так же, как если бы воздействовали на частицу в контексте квантовой механики.

Если воздействия на приложение нет



Если отладчик воздействует на приложение



Рис. 1.6. Выполнение Гейзенберга. В многопоточном приложении при воздействии отладчика на выполнение кода поведение всего приложения может измениться. Это изменение не позволит правильно проанализировать естественное поведение приложения, которое требовало исследования

При использовании многопоточной функциональности существует большое разнообразие вариантов. По моему мнению, именно это делает подобные сценарии особенно трудными для тестирования. Иногда профилировщик является правильным выбором, но даже профилировщик может воздействовать на выполнение приложения, тогда и этот инструмент неприемлем. Другой вариант: использование журналирования (это тема главы 5) в приложении. При возникновении определенных проблем можно найти способ сократить количество потоков до одного, чтобы можно было воспользоваться отладчиком для анализа.

Сценарий 4: передача некорректных вызовов в определенный сервис

Может потребоваться анализ сценария, в котором приложение некорректно взаимодействует с другим компонентом системы или с внешней системой. Предположим, что приложение отправляет HTTP-запросы другому приложению. От лиц, сопровождающих второе приложение, вы получаете сообщение о том, что HTTP-запросы имеют неправильный формат (возможно, отсутствует заголовок или тело запроса содержит неверные данные). На рис. 1.7 этот случай представлен графически.

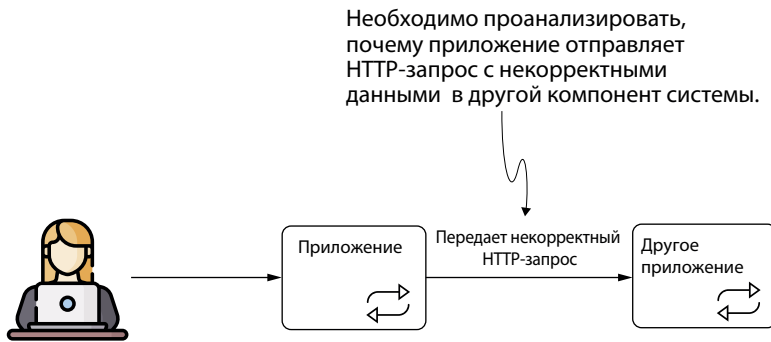
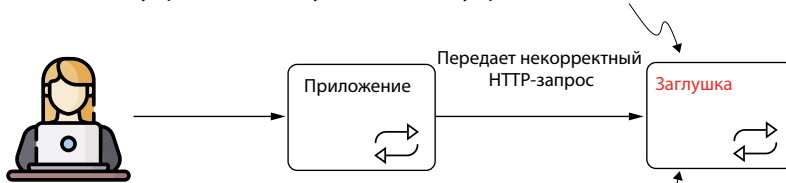


Рис. 1.7. Некорректные выходные данные могут быть ошибочными запросами, передаваемыми приложением в другой компонент системы. Вам могут предложить проанализировать это поведение и найти главную причину его возникновения

Это сценарий с некорректными выходными данными (wrong output). Как устранить эту проблему? В первую очередь необходимо определить, какая часть кода отправляет запросы. Если это уже известно, то можно воспользоваться отладчиком, чтобы проанализировать, как приложение создает запрос, и понять, что пошло не так. Если необходимо узнать, какая часть приложения отправляет запрос, то, возможно, потребуется профилировщик, о котором вы узнаете подробнее в главах 6–9. Профилировщик можно использовать, чтобы определить, какая часть кода работает в определенный интервал времени в процессе выполнения.

Есть один прием, который я всегда применяю в сложных случаях, подобных описанному выше, когда по какой-то причине невозможно сразу определить, в каком месте приложения отправляется/принимается запрос: я заменяю другое приложение (которому первое приложение отправляет некорректные запросы) заглушкой. Заглушка (stub) – это имитация приложения, которой я могу управлять, чтобы с ее помощью идентифицировать проблему. Например, чтобы определить, какая часть кода отправляет запросы, можно сделать заглушку, блокирующую запрос, тогда приложение будет бесконечно ожидать ответ. Затем я просто использую профилировщик для определения кода, остановившегося в ожидании ответа заглушки. На рис. 1.8 показано применение заглушки. Сравните с рис. 1.7, чтобы понять, как заглушка может подменять настоящее приложение.

Можно создать имитацию приложения для замены компонента, к которому обращается приложение. Это называется заглушкой. Вы управляете заглушкой, чтобы упростить анализ.



Например, можно сделать заглушку бесконечно блокирующей HTTP-запрос. В этом случае приложение будет оставаться заблокированным как раз в той инструкции, которая отправляет запрос. Можно воспользоваться профилировщиком, чтобы с легкостью найти эту инструкцию.

Рис. 1.8. Можно заменить компонент системы, к которому обращается приложение, заглушкой. Вы управляете заглушкой, чтобы быстро определить, из какого места приложения отправляется запрос. Заглушку также можно использовать для тестирования принятого решения после устранения проблемы

1.2.2. Изучение конкретных технологий

Еще один вариант использования методик анализа кода – изучение работы конкретных технологий. Некоторые разработчики шутят: шесть часов отладки могут сэкономить пять минут чтения документации. Хотя в действительности чтение документации не менее важно при изучении чего-то нового, некоторые технологии слишком сложны, чтобы изучать их, только лишь читая книги или спецификации. Я всегда советую обучающимся погружаться как можно глубже в конкретную рабочую среду или библиотеку, чтобы полностью понять ее.



СОВЕТ. При изучении любой технологии (рабочей среды или библиотеки) рекомендую выделить некоторое время на тщательный просмотр кода, который вы пишете. Всегда пытайтесь заглянуть как можно глубже и применяйте отладку кода рабочей среды.

Я начну со своего любимого фреймворка (рабочей среды) Spring Security. На первый взгляд Spring Security может показаться слишком простым. Это же просто средство реализации аутентификации и авторизации, не так ли? Именно так, но первое впечатление сохранится лишь до тех пор, пока вы не откроете для себя разнообразие способов конфигурирования этих двух функциональных возможностей в своем приложении. Объедините их неправильно и, возможно, получите проблему. Если эти средства не работают, то необходимо найти причину, и наилучшим выбором для этого является анализ кода Spring Security.

Отладка в большей степени, чем что-либо другое, помогла мне понять функциональность Spring Security. Чтобы помочь разобраться другим людям, я поделился своим опытом и знаниями в книге «Spring Security in Action» (Manning, 2020 г.). В этой книге я представил более 70 проектов не только для создания и выполнения, но также для отладки. Рекомендую выполнять отладку всех примеров из книг, которые вы читаете, чтобы изучать разнообразные технологии.

Второй пример изучения технологии в основном через отладку – Hibernate. Hibernate – это рабочая среда (фреймворк) высокого уровня, используемая для реализации в приложении возможности работать с базой данных SQL. Hibernate является одной из самых известных и часто используемых рабочих сред в мире Java, поэтому каждый разработчик на языке Java должен освоить ее.

Основы Hibernate изучаются с легкостью, это можно сделать, просто читая книги. Но в реальной практике использование Hibernate (где и как) подразумевает нечто гораздо большее, чем знание основ. Поэтому без глубокого погружения в код Hibernate я определенно не узнал бы столько об этой рабочей среде, сколько знаю сейчас.

Совет простой: изучая любую технологию (рабочую среду или библиотеку), выделяйте некоторое время на тщательный просмотр кода, который вы пишете. Всегда пытайтесь заглянуть как можно глубже и применяйте отладку кода рабочей среды. Это сделает вас разработчиком более высокой квалификации.

1.2.3. Выяснение причин замедления

Время от времени в приложениях возникают проблемы с производительностью, и, как и при любых других проблемах, требуется анализ, прежде чем вы узнаете, как их устранить. Освоение правильного применения различных методик отладки для выявления причин проблем с производительностью чрезвычайно важно.

На основе своего опыта отмечу, что в приложениях чаще всего возникают проблемы с производительностью, связанные с тем, насколько быстро приложение отвечает (реагирует). Но даже если большинство разработчиков считает замедление и снижение производительности равнозначным, это не тот случай. Проблемы замедления (ситуации, в которых приложение медленно реагирует на определенный триггер (событие)) – это лишь одна из разновидностей проблем с производительностью.

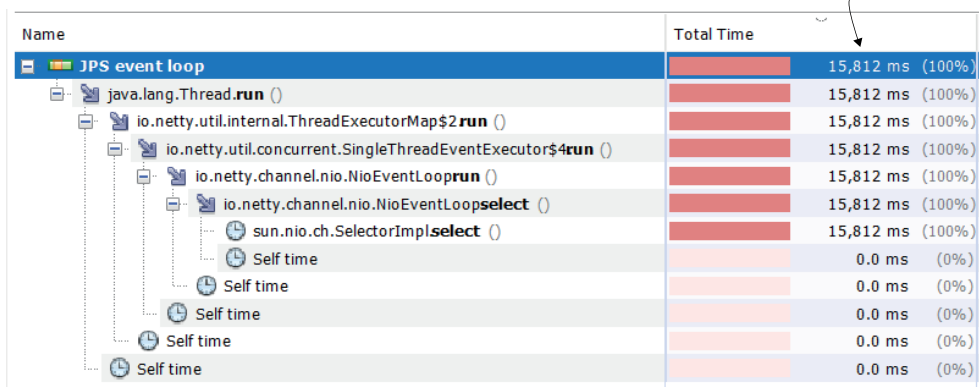
Например, однажды я отлаживал мобильное приложение, которое слишком быстро потребляло заряд батареи устройства. Это было приложение Android, использующее библиотеку, устанавливающую соединение с внешним устройством через Bluetooth. По какой-то причине эта библиотека создавала огромное количество потоков, но не закрывала их. Такие потоки, остающиеся открытыми и выполняющиеся без конкретной цели, называются потоками-зомби (zombie threads) и обычно становятся причиной

проблем с производительностью и с использованием памяти. Кроме того, их, как правило, очень трудно анализировать.

Но этот тип проблемы, при которой заряд батареи расходуется слишком быстро, также относится к проблемам производительности приложения. Приложение, использующее чрезмерно широкую сетевую полосу пропускания при передаче данных, – это еще один пример проблемы с производительностью.

Остановимся на проблемах замедления выполнения, которые возникают наиболее часто. Многие разработчики опасаются проблем замедления. Обычно не потому, что такие проблемы трудно идентифицировать, а потому что поиск решения по их устранению может оказаться весьма непростым делом. Поиск причины возникновения проблемы с производительностью обычно является легкой задачей для профилировщика, о котором вы узнаете подробнее в главах 6–9. В дополнение к определению выполняемого фрагмента кода, описанного в подразделе 1.2.1, профилировщик также показывает время, затраченное приложением на выполнение каждой инструкции (см. рис. 1.9).

Профилировщик показывает время выполнения каждой инструкции, что позволяет легко обнаружить место возникновения проблемы снижения производительности.



Name	Total Time
JPS event loop	15,812 ms (100%)
java.lang.Thread.run ()	15,812 ms (100%)
io.netty.util.internal.ThreadExecutorMap\$2.run ()	15,812 ms (100%)
io.netty.util.concurrent.SingleThreadEventExecutor\$4.run ()	15,812 ms (100%)
io.netty.channel.nio.NioEventLoop.run ()	15,812 ms (100%)
io.netty.channel.nio.NioEventLoop.select ()	15,812 ms (100%)
sun.nio.ch.SelectorImpl.select ()	15,812 ms (100%)
Self time	0.0 ms (0%)
Self time	0.0 ms (0%)
Self time	0.0 ms (0%)
Self time	0.0 ms (0%)
Self time	0.0 ms (0%)

Рис. 1.9. Анализ замедления работы приложения с помощью профилировщика.

Профилировщик показывает время работы каждой инструкции во время выполнения кода приложения. Эта функция профилировщика наилучшим образом подходит для определения главных причин возникновения проблем с производительностью

Во многих случаях причиной замедления выполнения являются вызовы методов ввода-вывода, например чтение или запись в файл или базу данных или передача данных по сети. Из-за этого разработчики часто интуитивно выполняют поиск причины возникновения проблемы. Если известно, какая функция повреждена, то можно сосредоточиться на вызовах

методов ввода-вывода, выполняемых этой функцией. Такой подход также помогает минимизировать область действия проблемы, но обычно остается необходимость в применении инструментального средства для точного определения места ее возникновения.

1.2.4. Исследование случаев аварийного завершения приложения

Иногда приложения полностью прекращают реагировать на различные события. Эти разновидности проблем обычно считаются гораздо более трудными для анализа, чем все прочие. Во многих случаях аварийное завершение приложений случается только при особенных условиях, поэтому невозможно воспроизвести те же условия в локальной среде (т. е. преднамеренно заставить проблему проявиться).

Каждый раз, когда вы анализируете проблему, следует в первую очередь попытаться воспроизвести ее в среде, в которой ее можно исследовать во всех подробностях. Такой поход придает анализу большую гибкость и помогает подтвердить правильность принятого решения. Но воспроизвести проблему удастся не всегда. Кроме того, обычно весьма непросто воспроизвести аварийное завершение приложения.

Известны два основных варианта сценариев аварийного завершения приложений:

- приложение останавливается (завершается) полностью;
- приложение продолжает работу, но не отвечает на запросы.

Если приложение полностью прекратило работу, то, как правило, потому что возникла ошибка, которую невозможно исправить. Чаще всего причиной такого поведения становится ошибка памяти. В Java-приложении ситуация, в которой основная память – куча (heap) – заполняется до отказа и приложение не может продолжить работу, представлена сообщением об ошибке `OutOfMemoryError`.

Для анализа проблем с основной памятью (кучей) мы используем дампы памяти (heap dumps), предоставляющие снимок содержимого памяти в определенный момент времени. Можно сконфигурировать процесс Java так, чтобы он автоматически генерировал такой снимок, когда возникает ошибка `OutOfMemoryError` и приложение завершается аварийно.

Дампы памяти – мощный инструмент, предоставляющий множество подробностей о том, как внутри приложения обрабатываются данные. Более подробно использование дампов памяти рассматривается в главе 11. А сейчас продемонстрируем их применение на коротком примере.

В листинге 1.2 показан небольшой фрагмент кода, который заполняет память экземплярами класса `Product`. Это приложение можно найти в проекте `da-ch1-ex2`, прилагаемом к книге. Приложение непрерывно добавляет экземпляры `Product` в список, преднамеренно создавая условие возникновения ошибки и вывода сообщения `OutOfMemoryError`.

Листинг 1.2. Пример приложения, в котором возникает ошибка `OutOfMemoryError`

```

public class Main {

    private static List<Product> products =
        new ArrayList<>();

    public static void main(String[] args) {
        while (true) {
            products.add(
                new Product(UUID.randomUUID().toString()));
        }
    }
}

```

- ❶ Объявляется список, в котором сохраняются ссылки на объекты `Product`.
- ❷ Мы непрерывно добавляем экземпляры `Product` в список до тех пор, пока память в куче не будет заполнена до предела.
- ❸ Каждый экземпляр `Product` имеет атрибут `String`. В качестве его значения используется неповторяющийся случайно выбираемый идентификатор.

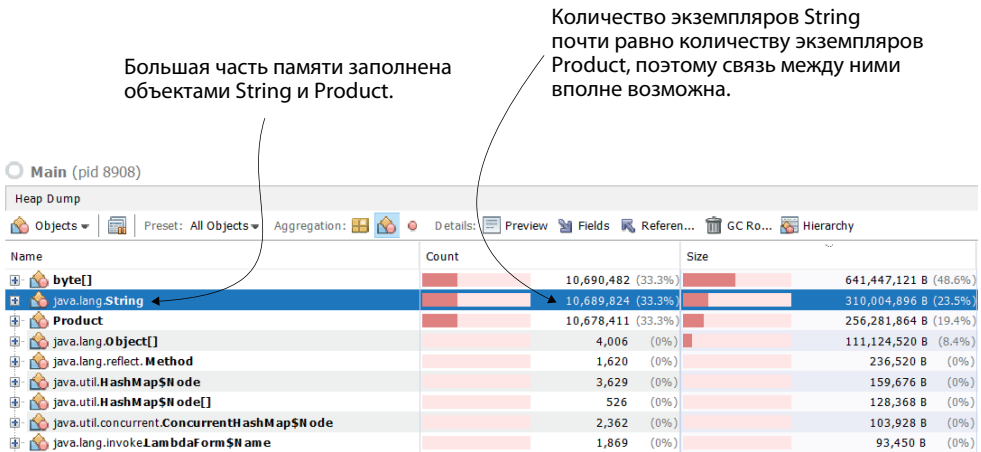


Рис. 1.10. Дамп кучи похож на карту памяти. Если вы научитесь читать его, то получите бесценные сведения о том, как в приложении происходит внутренняя обработка данных. Дамп кучи помогает анализировать проблемы использования памяти и производительности. В этом примере можно с легкостью увидеть, какой объект заполняет большую часть памяти приложения, а также связь между экземплярами `Product` и `String`

На рис. 1.10 показан дамп памяти кучи, созданный для одного выполнения этого приложения. Здесь отчетливо видно, что экземпляры `Product` и

String заполняют большую часть памяти. Дамп кучи похож на карту памяти. Он предоставляет множество подробностей, в том числе отношения между экземплярами и значениями. Например, даже если вы не видите код, то все равно можете заметить связь между Product и String, обоснованную близостью числовых значений количества их экземпляров. Не беспокойтесь, если эти характеристики выглядят слишком сложными. Все, что вам необходимо знать об использовании дампов памяти, мы подробно рассмотрим в главе 11.

Если приложение продолжает выполняться, но не отвечает на запросы, то дамп потоков (thread dump) является наилучшим инструментом для анализа случившегося. На рис. 1.11 показан пример дампа потоков и некоторые подробности, которые он предоставляет. В главе 10 мы рассмотрим генерацию и исследование дампов потоков для анализа кода.

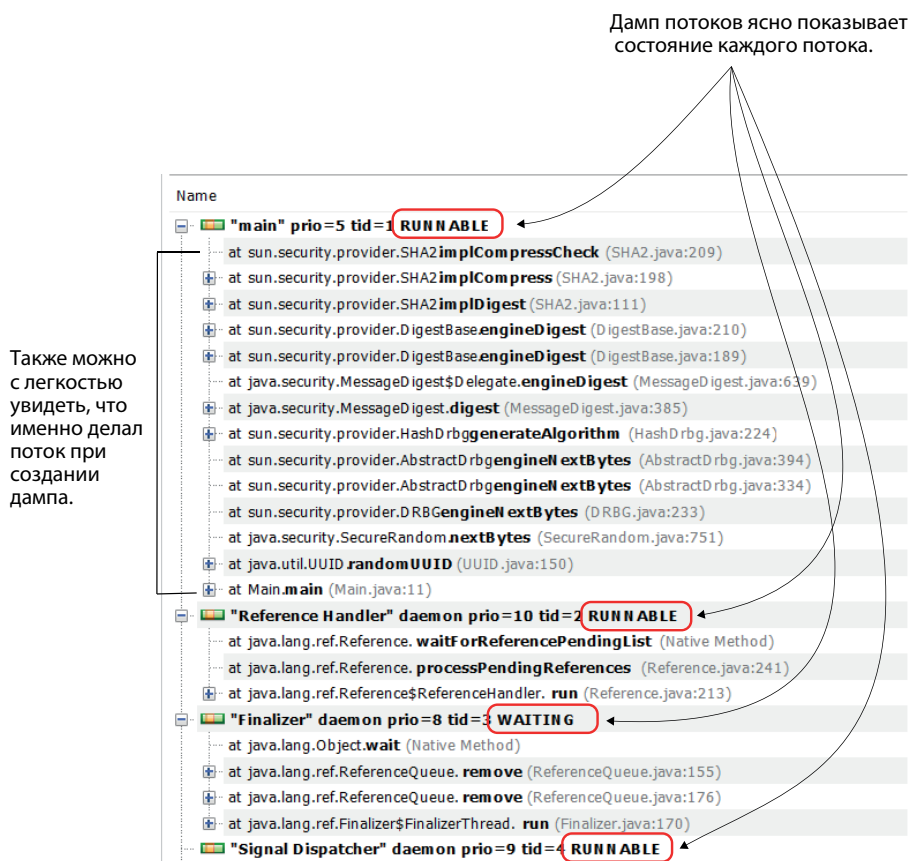


Рис. 1.11. Дамп потоков предоставляет подробную информацию о потоках, которые активны (выполняются) в момент создания дампа. Он включает состояния потоков и трассировки стека, которые сообщают вам о том, что именно выполняли потоки или что стало причиной их блокировки. Эти подробности весьма полезны для анализа причин зависания приложения или проблем с производительностью

1.3. Зачем нужно читать эту книгу

Эта книга предназначена для разработчиков на языке Java с различными уровнями практического опыта: от начинающего до эксперта. Вы изучите разнообразные методики анализа кода, наилучшие сценарии, в которых можно применять эти методики, и конкретные способы их применения, позволяющие сэкономить время на анализ и устранение проблем.

Если вы начинающий разработчик, то, вероятнее всего, многому научитесь, читая эту книгу. Некоторые разработчики полностью овладели всем этими методиками только после многих лет практики, а другим недоступен такой уровень мастерства. Если вы уже являетесь экспертом, то, возможно, обнаружите, что многие темы вам уже хорошо знакомы, но при этом остается неплохая возможность найти новые интересные подходы, с которыми вам еще не приходилось иметь дело.

После завершения изучения этой книги вы приобретете следующие практические навыки:

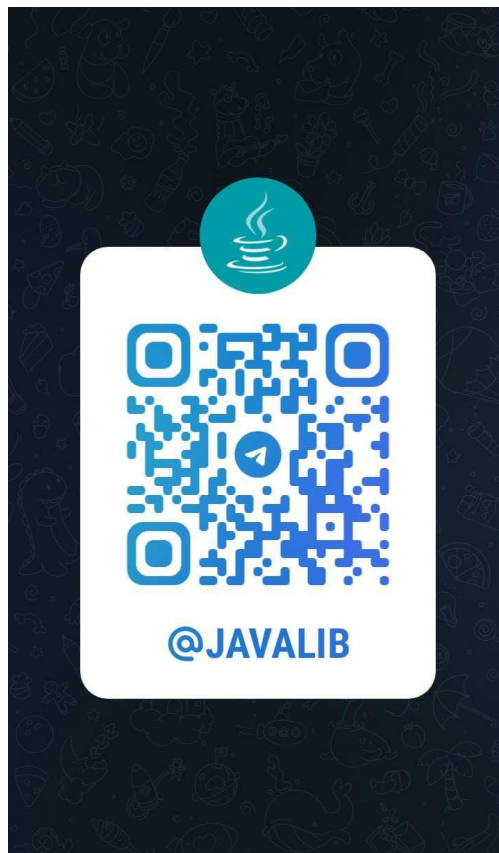
- применение различных методов использования отладчика, чтобы понять логику приложения или обнаружить проблему;
- анализ скрытой функциональности с помощью профилировщика, чтобы лучше понимать, как работает приложение или его конкретные зависимости;
- использование методик анализа кода, чтобы определить, что само приложение или одна из его зависимостей стали причиной возникновения проблемы;
- анализ данных в снимке памяти приложения для идентификации потенциальных проблем, а также чтобы узнать, как приложение обрабатывает данные;
- использование журналирования для выявления проблем в поведении приложения или потенциальных угроз для его безопасности;
- использование удаленной отладки для выявления проблем, которые невозможно воспроизвести в другой рабочей среде;
- правильный выбор методики анализа приложения, чтобы ускорить процесс анализа.

1.4. Резюме

- Для анализа поведения программного обеспечения можно использовать разнообразные методики.
- В зависимости от конкретной ситуации одна методика анализа может работать лучше другой. Необходимо знать, как выбрать правильную методику, чтобы сделать анализ более эффективным.

- В некоторых сценариях использование сочетания методик помогает быстрее обнаружить проблему. Изучение практического применения каждой методики анализа предоставляет значительное преимущество при устранении сложных проблем.
- Во многих случаях разработчики используют методики анализа не для устранения проблем, а для изучения новых технологий. При освоении сложных рабочих сред (фреймворков), таких как Spring Security или Hibernate, простого чтения книг или документации недостаточно. Превосходный способ ускорения обучения – отладка примеров использования технологии, которую вы хотите лучше понять.
- Процесс анализа упрощается, если можно воспроизвести проблему в среде, в которой есть возможность исследования этой проблемы. Воспроизведение проблемы не только помогает найти главную причину ее возникновения, но также может подтвердить правильность выбора решения по ее устранению при непосредственном практическом применении.

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javallib>



Глава 2

Изучение логики приложения с помощью методик отладки

Темы:

- когда использовать отладчик и когда избегать его применения;
- использование отладчика для анализа кода.

Не так давно, когда я брал уроки игры на фортепиано, мы вместе с учителем музыки читали нотную запись произведения, которое я хотел научиться играть. Я был невероятно впечатлен, когда учитель сыграл это произведение, впервые читая нотную запись, и подумал: «Как же это круто. Как люди учатся этому?»

Потом я вспомнил, как несколько лет назад участвовал в рабочем сеансе парного программирования с одним из новых программистов, только что принятых в компанию, в которой я тогда работал. Мы анализировали достаточно большой и сложный фрагмент кода, используя отладчик, и пришла моя очередь сесть за клавиатуру. Я начал перемещение по коду, довольно-таки быстро нажимая клавиши, позволяющие пропускать, входить и выходить из конкретных строк кода. Я полностью сосредоточился на коде, но был абсолютно спокоен и расслаблен, почти забыв о том, что рядом кто-то находится (это было невежливо с моей стороны). И услышал, как этот парень сказал: «Эй, притормози. Ты слишком быстро это делаешь. Как ты можешь вообще читать этот код?»

Я понял, что та ситуация была очень похожа на урок с учителем музыки. Как люди учатся этому? Ответ гораздо проще, чем может показаться: упорная работа и приобретение практического опыта. Хотя практический опыт незаменим и требует огромного количества времени, я поделюсь несколькими советами, которые помогут вам намного быстрее улучшать практические навыки. В этой главе рассматривается самый важный инструмент, используемый для понимания кода: отладчик.



ОПРЕДЕЛЕНИЕ. Отладчик (debugger) – это инструментальное средство, которое позволяет временно приостанавливать выполнение в конкретных строках кода и вручную выполнять каждую инструкцию, одновременно наблюдая, как изменяются данные.

Использование отладчика похоже на навигацию по картам Google (Google Maps): он помогает найти путь в сложной логике, реализованной в коде. Кроме того, это инструмент, чаще всего применяемый для изучения и понимания кода.

Отладчик обычно является первым инструментом, который разработчик осваивает, чтобы лучше понять, что именно делает код. К счастью, все интерактивные среды разработки (IDE) включают в свой инструментальный комплект отладчик, поэтому для получения его в свое распоряжение ничего делать не нужно. В этой книге используется IntelliJ IDEA Community во всех примерах, но любая другая IDE во многом на нее похожа и предоставляет все возможности, которые мы будем рассматривать (иногда их внешний вид может немного различаться). Хотя отладчик кажется инструментом, способ применения которого известен большинству разработчиков, вы можете обнаружить в этой главе и в главе 3 несколько новых методик его использования.

В разделе 2.1 мы начнем с обсуждения того, как разработчики читают код и почему во многих случаях простого чтения кода недостаточно для его понимания. Необходимо войти в отладчик или профилировщик (который рассматривается несколько позже в главах 6–9). В разделе 2.2 мы продолжим обсуждение, применяя простейшие методики использования отладчика на примере.

Если вы опытный разработчик, то, возможно, уже знаете эти методики. Но вы можете найти кое-что полезное, читая эту главу для освежения знаний, или можете сразу перейти к более продвинутым методикам применения отладчика, которые мы будем рассматривать в главе 3.



ОК! Посмотрим!



Рис. 2.1

2.1. Когда недостаточно просто проанализировать код

Начнем с обсуждения того, как читать код и почему иногда простого чтения логики недостаточно для ее понимания. В этом разделе описано, как выполняется чтение кода и чем оно отличается от чтения чего-либо другого, например прозы или стихов. Чтобы наглядно показать это различие и понять причины возникновения сложности в «расшифровке» кода, мы воспользуемся фрагментом кода, реализующим небольшую часть логики. Понимание способа, которым наш мозг интерпретирует код, поможет вам правильно оценить необходимость инструментов, подобных отладчику.

Любой эпизод анализа программы начинается с чтения ее кода. Но чтение кода отличается от чтения поэзии. При чтении стихотворения вы перемещаетесь по тексту строка за строкой в определенном линейном порядке, позволяя мозгу собирать информацию и воспринимать смысл. Если вы прочитаете то же стихотворение во второй раз, то, возможно, уловите другой смысл.

Но при чтении кода ситуация совершенно другая. Во-первых, код имеет нелинейную структуру. При чтении кода вы не просто переходите от строки к строке. Вместо этого вы прыгаете вперед и назад по инструкциям, чтобы понять, как они воздействуют на обработку данных. Чтение кода больше похоже на лабиринт, чем на прямую дорогу. И если вы не наблюдательны, то можете заблудиться и забыть, с чего начали. Во-вторых, в отличие от поэзии смысл кода всегда и для всех один и тот же. Этот смысл и есть цель анализа.

По аналогии с компасом для определения направления движения отладчик облегчает понимание того, что делает код. В качестве примера воспользуемся методом `decode(List<Integer> input)`. Этот код можно найти в проекте `da-ch2-ex1`, прилагаемом к этой книге.

Листинг 2.1. Пример отладки метода

```
public class Decoder {

    public Integer decode(List<String> input) {
        int total = 0;
        for (String s : input) {
            var digits = new StringDigitExtractor(s).extractDigits();
            total += digits.stream().collect(Collectors.summingInt(i -> i));
        }

        return total;
    }
}
```

Если вы читаете код с первой до последней строки, то должны предполагать, как работают некоторые вещи, чтобы понять этот код. Эти инструкции на самом деле выполняются так, как вы думаете? Если вы не уверены, то должны заглянуть глубже и посмотреть, что действительно делает этот код, т. е. необходимо проанализировать его внутреннюю логику. На рис. 2.2 изображены две неопределенности в рассматриваемом фрагменте кода:

- что действительно делает конструктор `StringDigitExtractor()`? Возможно, он просто создает объект, но, может быть, делает что-то еще. Может оказаться, что он каким-то образом изменяет значение переданного параметра;
- что является результатом вызова метода `extractDigits()`? Он возвращает список цифр? Изменяет ли он параметр внутри объекта, используемый при создании с помощью конструктора `StringDigitExtractor`?

```
public class Decoder {

    public Integer decode(List<String> input) {
        int total = 0;
        for (String s : input) {
            var digits = new StringDigitExtractor(s).extractDigits();
            total += digits.stream().collect(Collectors.summingInt(i -> i));
        }

        return total;
    }
}
```

Этот конструктор только создает объект или делает что-то еще?

Что в действительности делает этот метод? Использует ли он значение параметра типа String?

Рис. 2.2. При чтении кода часто необходимо понять, что происходит «за кулисами» некоторых инструкций, формирующих логику данного приложения. Имена методов не всегда достаточно информативны, поэтому на них не следует полагаться. Вместо этого требуется заглянуть глубже и узнать, что они делают

Даже в небольшом фрагменте кода вам, возможно, придется заглянуть поглубже в инструкции. Код каждой очередной изучаемой инструкции создает новый план исследования и увеличивает когнитивную сложность (см. рис. 2.3 и 2.4). Чем глубже вы погружаетесь в логику и чем больше планов исследования при этом создается, тем более сложным становится весь процесс в целом.

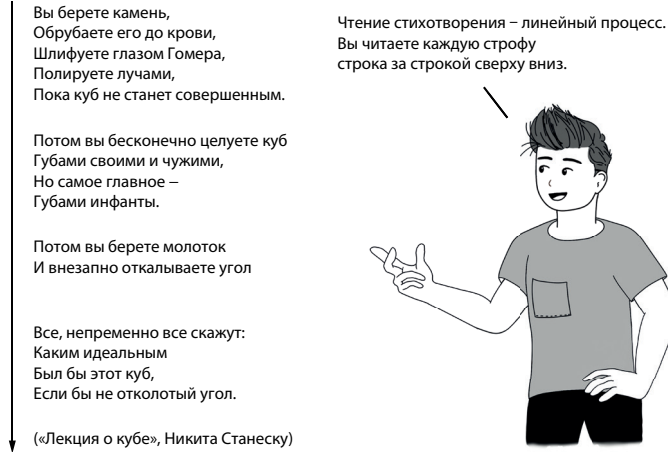


Рис. 2.3. Сравните, как вы читаете стихотворение и код. Стихотворение читается строка за строкой, но при чтении кода вы все время «прыгаете» вперед и назад

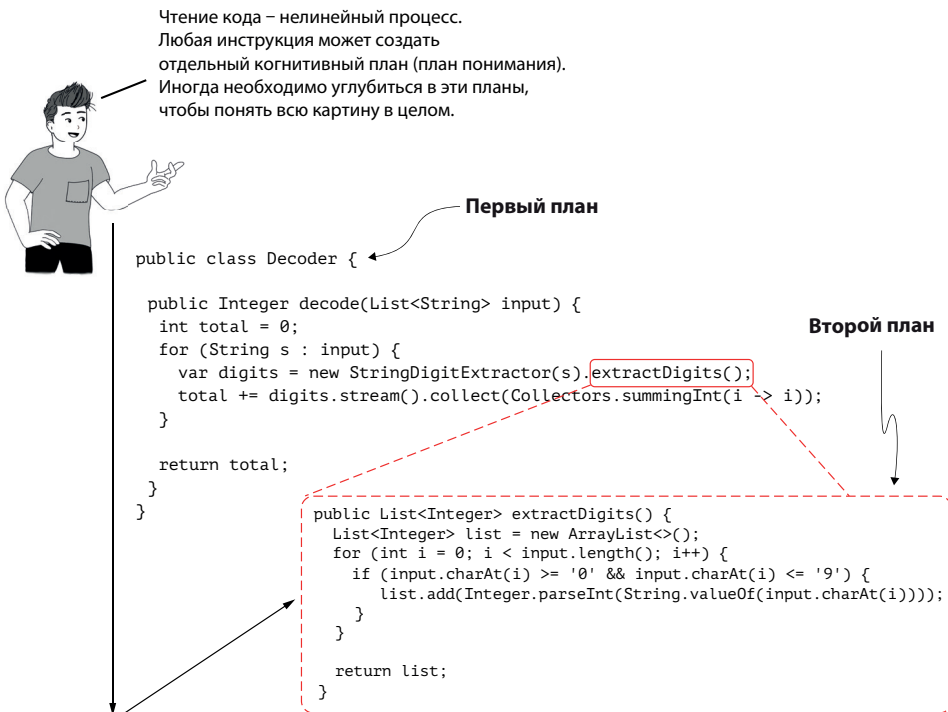


Рис. 2.4. Чтение кода отличается от чтения стихотворения – это гораздо более сложный процесс. Чтение кода можно мысленно представить как чтение в двух измерениях. Одно измерение – чтение фрагмента кода от начала до конца («сверху вниз»). Второе измерение – углубление в конкретную инструкцию, чтобы понять ее во всех подробностях. Очень трудно, просто читая исходный код, пытаться запомнить, как работают составные части каждого плана и как их совокупность формирует понимание кода

При чтении стихов маршрут всегда один. Но анализ кода создает множество маршрутов в одном и том же фрагменте логики. Чем меньше новых планов вы открываете, тем менее сложным является процесс чтения. Вы должны принимать решения о пропуске определенных инструкций, чтобы упростить общий процесс анализа, или углубляться в подробности, чтобы лучше понять каждую отдельную инструкцию, увеличивая сложность процесса.



СОВЕТ. Всегда пытайтесь сократить маршрут чтения, минимизируя количество планов, открываемых для анализа. Использование отладчика поможет вам более удобно перемещаться по коду, постоянно отслеживать текущее местонахождение и наблюдать, как приложение изменяет данные при выполнении.

2.2. Анализ кода с помощью отладчика

В этом разделе мы рассмотрим инструмент, который может помочь минимизировать когнитивные усилия при чтении кода с целью понять, как он работает, – отладчик (debugger). Все интегрированные среды разработки (IDE) предоставляют отладчик, и даже если его интерфейс может выглядеть немного по-разному в различных IDE, возможности, как правило, одинаковы. В этой книге используется IntelliJ IDEA Community, но вы можете работать в IDE, которую предпочитаете, и сравнивать ее с примерами из книги. Вы обнаружите, что они почти одинаковы.

Отладчик упрощает процесс анализа следующим образом:

- предоставляет средства временной приостановки выполнения на конкретном шаге и обеспечивает выполнение каждой инструкции вручную в нужном темпе;
- показывает текущую локацию и локацию, из которой вы пришли в нее по маршруту чтения кода, таким образом, отладчик работает как карта, которой можно пользоваться, не пытаясь запомнить все подробности;
- показывает значения, которые содержатся в переменных, упрощая анализ в аспекте наблюдения и обработки;
- позволяет пробовать различные варианты непосредственно по ходу отладки, используя средства контроля (watchers) и возможности вычисления выражений.

Снова обратимся к примеру из проекта da-ch2-ex1 и используем самые простые возможности отладчика для понимания кода.

Листинг 2.2. Фрагмент кода, который необходимо понять

```
public class Decoder {

    public Integer decode(List<String> input) {
        int total = 0;
        for (String s : input) {
            var digits = new StringDigitExtractor(s).extractDigits();
            total += digits.stream().collect(Collectors.summingInt(i -> i));
        }

        return total;
    }
}
```

Уверен, что вы удивленно спрашиваете: «Ну и как я узнаю, где именно использовать отладчик?» Это вполне обоснованный вопрос, на который я обязан ответить, прежде чем продолжить описание. Главное предварительное условие – знание того фрагмента логики, который необходимо анализировать. Как вы узнаете в этом разделе, самым первым шагом применения отладчика является выбор инструкции, на которой требуется временно приостановить выполнение.



ПРИМЕЧАНИЕ. Если вы пока еще не знаете, с какой инструкции необходимо начать анализ, то не сможете воспользоваться отладчиком.

В реальной практике вы будете встречаться с ситуациями, в которых неизвестен конкретный фрагмент логики, требующий анализа. В подобных случаях, прежде чем вы сможете воспользоваться отладчиком, необходимо применить разнообразные методики поиска той части кода, которую вы предполагаете проанализировать, используя отладчик (но эти методики мы рассмотрим в следующих главах). В этой главе и в главе 3 все внимание сосредоточено только на использовании отладчика, поэтому подразумевается, что каким-то образом нашли фрагмент кода, который необходимо понять.

Вернемся к приведенному выше примеру: с чего начать? Во-первых, необходимо прочитать код и выяснить, что в нем понятно, а что нет. После определения того места, где логика становится непонятной, можно выполнить приложение и «приказать» отладчику временно приостановить выполнение. Приостановку выполнения можно определить в тех строках кода, в которых нельзя понять, как они изменяют данные. Чтобы «указать» отладчику место приостановки выполнения приложения, мы используем точки останова (breakpoints).



ОПРЕДЕЛЕНИЕ. Точка останова (breakpoint) – это маркер, используемый для строк, в которых отладчик должен приостановить выполнение, чтобы появилась возможность анализа реализованной здесь логики. Отладчик приостановит выполнение перед строкой, помеченной точкой останова.

На рис. 2.5 я выделил цветом код, который легко понять (предполагая, что вы знаете основы языка). Здесь можно видеть, что код принимает список как входные данные, выполняет синтаксический анализ (парсинг) этого списка, обрабатывает каждый элемент списка и каким-то образом вычисляет целочисленное значение, которое метод возвращает в конце. Кроме того, обработку, реализованную в методе, легко определить и без отладчика.

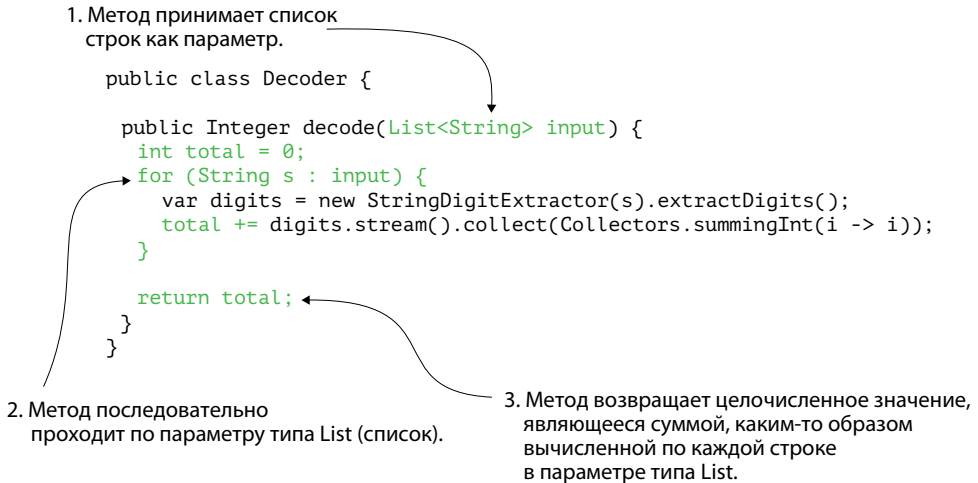


Рис. 2.5. Если вам знакомы основы языка, то вы с легкостью увидите, что этот код принимает набор (collection) как входные данные и выполняет синтаксический анализ (парсинг) этого набора данных, чтобы вычислить некоторое целочисленное значение

На рис. 2.6 я выделил цветом строки, в которых обычно возникают затруднения в понимании того, что делает метод. Эти строки более сложны для расшифровки, потому что скрывают свою реализацию логики. Вы можете распознать `digits.stream().collect(Collectors.summingInt(i -> i))`, как если бы это была часть Stream API, предоставленного в пакете JDK, начиная с версии Java 8. Но мы не можем сказать то же самое о `new StringDigitExtractor(s).extractDigits()`. Поскольку это часть анализируемого приложения, такая инструкция может делать все что угодно.

```

public class Decoder {

    public Integer decode(List<String> input) {
        int total = 0;
        for (String s : input) {
            var digits = new StringDigitExtractor(s).extractDigits();
            total += digits.stream().collect(Collectors.summingInt(i -> i));
        }

        return total;
    }
}

```

Что происходит с каждой строкой в этом списке?
Как строка (String) превращается в число?

Рис. 2.6. В этом фрагменте кода я выделил цветом строки кода, которые более трудны для понимания. При использовании отладчика добавляйте первую точку останова в первой строке, которая делает код очень трудным для понимания

Способ, который разработчик выбирает для написания исходного кода, также может добавлять дополнительную сложность. Например, начиная с версии Java 10, разработчики могут логически выводить тип локальной переменной, используя ключевое слово `var`. Логический вывод типа переменной не всегда является разумным выбором, потому что может сделать код еще более трудным для чтения (см. рис. 2.6), создавая еще один сценарий, в котором применение отладчика оказывается полезным.



СОВЕТ. При анализе кода с помощью отладчика начинайте с первой строки кода, которую вы не можете понять.

Обучая начинающих разработчиков и студентов в течение многих лет, я заметил, что во многих случаях они начинают отладку с первой строки конкретного блока кода. Разумеется, вы можете поступать так же, но более полезно сначала прочитать код без отладчика и попытаться определить, можете ли вы понять этот код. Затем начинайте отладку прямо с того места, где возникло затруднение. Такой подход позволит сэкономить время, так как, возможно, вы сразу поймете, что для понимания того, что происходит в рассматриваемом фрагменте логики, отладчик не нужен. В конце концов, даже если вы используете отладчик, вам необходимо всего лишь пройти по непонятному коду.

В некоторых сценариях точка останова добавляется в строку, потому что предназначение этой строки неочевидно. Иногда приложение генерирует исключение – его вы увидите в журналах, но не узнаете, какая из

предшествующих строк стала причиной проблемы. В этом случае можно добавить точку останова, чтобы приостановить выполнение приложения непосредственно перед генерацией исключения. Но основная идея остается той же самой: избегайте приостановки выполнения в инструкциях, которые вы понимаете. Используйте точки останова в тех строках кода, которые заслуживают особого внимания.

В рассматриваемом здесь примере мы начнем с добавления точки останова в строке 11, как показано на рис. 2.7:

```
var digits = new StringDigitExtractor(s).extractDigits();
```

1. Добавьте точку останова в строке, где отладчик должен приостановить выполнение. В этой строке должна находиться первая инструкция, заслуживающая особого внимания.
2. Запустите приложение под управлением отладчика.

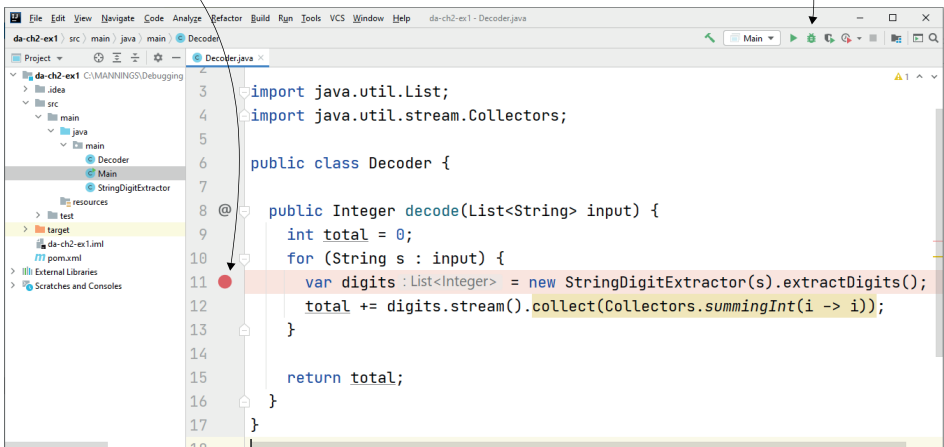


Рис. 2.7. Щелкните левой кнопкой мыши около номера строки, чтобы добавить в нее точку останова. Затем запустите приложение под управлением отладчика.

Выполнение приостановится в строке, помеченной точкой останова, и позволит вам управлять им в ручном режиме

В общем случае для добавления точки останова в строке в любой IDE необходимо щелкнуть (левой кнопкой мыши) по номеру строки или около него (но лучше воспользоваться комбинацией клавиш: в IntelliJ можно использовать **Ctrl+F8** в ОС Windows/Linux или **Command+F8** в macOS). Точка останова будет изображена в виде небольшого кружка, как показано на рис. 2.7. Убедитесь в том, что приложение работает под управлением отладчика. В IntelliJ найдите кнопку с изображением маленького жучка рядом с кнопкой запуска приложения. Также можно щелкнуть правой кнопкой мыши по файлу основного (main) класса и использовать кнопку **Debug** из контекстного меню. Когда поток выполнения дойдет до строки, помеченной точкой останова, выполнение приостановится, позволяя вам продолжить движение по коду в ручном режиме.

Поскольку комбинации клавиш могут изменяться и различаться в зависимости от используемой операционной системы (некоторые разработчики даже предпочитают настраивать их по своему усмотрению), я не уделяю много внимания их описанию. Но рекомендую обратиться к руководству по IDE и узнать, как пользоваться комбинациями клавиш для быстрого выполнения команд.

Примечание. Помните о том, что необходимо выполнять приложение, используя опцию **Debug** (Отладка) для активизации отладчика. Если вы применили опцию **Run** (Пуск), то точки останова учитываться не будут, так как IDE не связывает отладчик с текущим работающим процессом. Некоторые IDE могут запускать приложение с уже подключенным отладчиком, но если такая возможность отсутствует (как в IntelliJ или Eclipse), то выполнение приложения не будет приостановлено в определенных вами точках останова.

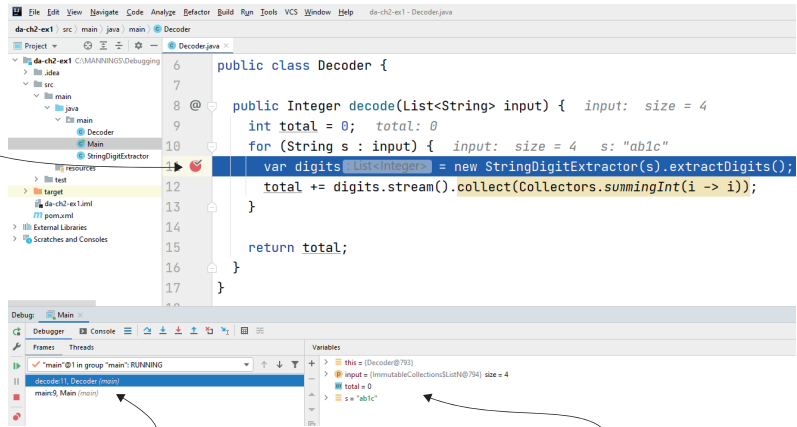
Когда отладчик приостанавливает выполнение кода на конкретной инструкции в строке, помеченной точкой останова, вы можете воспользоваться полезной информацией, предоставляемой IDE. На рис. 2.8 можно видеть, что используемая мной IDE выводит два весьма важных блока информации:

- значения всех переменных в текущей области видимости – знание всех переменных в текущей области видимости и соответствующих значений помогает понять, какие данные обрабатываются и как логика воздействует на данные. Напомню, что выполнение приостанавливается перед строкой, помеченной точкой останова, так что состояние данных остается неизменным;
- трассировка стека выполнения – показывает, как приложение выполняет строку кода, в которой отладчик приостановил поток выполнения. Каждая строка в трассировке стека – это метод, включенный в цепочку вызовов. Трассировка стека выполнения помогает визуально представить маршрут выполнения, при этом нет необходимости запоминать, как вы пришли к рассматриваемой инструкции при использовании отладчика для перемещения по коду.



СОВЕТ. Можно добавлять столько точек останова, сколько необходимо, но лучше одновременно использовать ограниченное количество и сосредоточиться на соответствующих строках кода. Я обычно использую одновременно не более трех точек останова. Я часто наблюдаю, как разработчики добавляют слишком много точек останова, забывают о них, а в результате запутываются в анализируемом коде.

Выполнение приостанавливается в строке, помеченной точкой останова.



Отладчик также показывает трассировку стека с выводом маршрута выполнения, так что вы легко можете увидеть, кто вызвал анализируемый метод.

Когда отладчик приостанавливает выполнение приложения в заданной строке, вы можете видеть значения всех переменных в текущей области видимости.

Рис. 2.8. Когда отладчик приостанавливает выполнение в заданной строке кода, вы можете увидеть все переменные в текущей области видимости и их значения. Также можно воспользоваться трассировкой стека выполнения, чтобы вспомнить, где вы находитесь при перемещении по строкам кода

В общем случае видимые значения переменных в текущей области видимости легко понять. Но в зависимости от вашего практического опыта вы можете не знать о том, что такое трассировка стека выполнения. В подразделе 2.2.1 подробно описывается трассировка стека выполнения и объясняется, почему этот инструмент весьма важен. Затем мы рассмотрим, как перемещаться по коду, используя основные операции, такие как шаг с обходом (step over), шаг с входом (step into) и шаг с выходом (step out). Если вы уже хорошо знакомы с трассировкой стека выполнения, то можете пропустить подраздел 2.2.1 и сразу перейти к подразделу 2.2.2.

2.2.1. Что такое трассировка стека выполнения, и как ее использовать

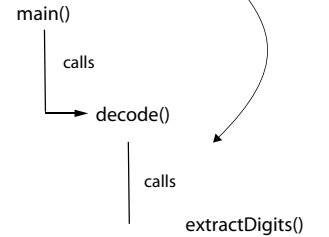
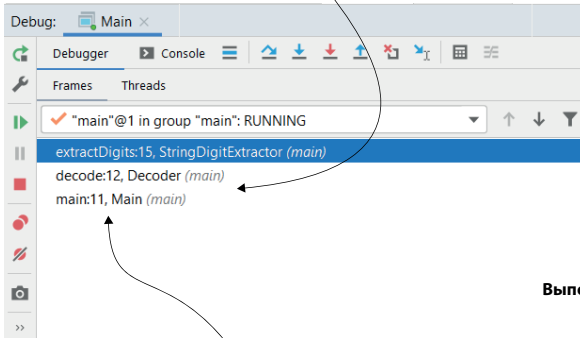
Трассировка стека выполнения – чрезвычайно полезный инструмент, применяемый для понимания кода во время его отладки. Почти как карта, трассировка стека выполнения показывает маршрут к конкретной строке кода, где отладчик приостановил выполнение, и помогает решить, куда двигаться дальше.

На рис. 2.9 показано сравнение трассировки стека выполнения и потока выполнения в формате дерева. Трассировка стека демонстрирует, как методы вызывают друг друга до точки, в которой отладчик приостановил

выполнение. В трассировке стека можно видеть имена методов и классов, а также строки, из которых выполнялись вызовы.

Мы читаем стек выполнения снизу вверх. Нижний уровень в стеке – первый уровень. На первом уровне началось выполнение. Верхний (последний) уровень – это метод, в котором в текущий момент приостановлено выполнение.

Это представление в форме дерева трассировки стека выполнения. Метод `main()` в классе `Main` вызывает метод `decode()` из класса `Decoder`. Далее метод `decode()` вызывает метод `extractDigits()` из класса `StringDigitsExtractor`. Выполнение приостанавливается в методе `extractDigits()`.



Выполнение приостановлено в методе `extractDigits()`

Трассировка стека выполнения показывает имена классов и строку в файле, где был вызван метод.

Рис. 2.9. Верхний уровень трассировки стека выполнения, где отладчик приостановил выполнение. Все прочие уровни в трассировке стека показывают, где были вызваны методы, представленные предыдущими уровнями. Самый нижний уровень трассировки стека (первый уровень) – место, где началось выполнение текущего потока

Одним из вариантов использования трассировки стека выполнения, который я часто применяю, является поиск скрытой логики в маршруте (потоке) выполнения. В большинстве случаев разработчики используют трассировку стека выполнения просто для того, чтобы понять, откуда был вызван конкретный метод. Но при этом следует также учитывать тот факт, что приложения, использующие рабочие среды (такие как Spring, Hibernate и т. п.), иногда изменяют цепочку вызовов метода.

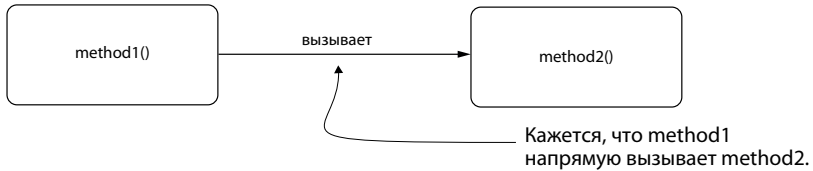
Например, приложения Spring часто используют код, разделяемый на так называемые аспекты (aspects) (в терминологии Java/Jakarta EE – перехватчики (interceptors)). Эти аспекты реализуют логику, которую рабочая среда применяет для расширения возможностей выполнения конкретных методов при определенных условиях. К сожалению, подобную логику часто трудно наблюдать, поскольку вы не можете напрямую увидеть код аспекта в цепочке вызовов при чтении кода (см. рис. 2.10). Такая характеристика чрезвычайно затрудняет анализ конкретного функционального средства.

Рассмотрим пример кода, чтобы исследовать его поведение и узнать, как трассировка стека выполнения помогает в подобных случаях. Этот пример можно найти в проекте `da-ch2-ex2`, прилагаемом к книге (в приложении В

представлены дополнительные инструкции по открытию проекта и запуску приложения). Проект представляет собой небольшое приложение Spring, которое выводит значение параметра в консоли.

В листингах 2.3, 2.4 и 2.5 показана реализация трех классов. Из листинга 2.3 понятно, что метод `main()` вызывает метод `saveProduct()` класса `ProductController` с передачей значения параметра "Beer".

Видимый поток выполнения методов



Как в действительности выполняется код

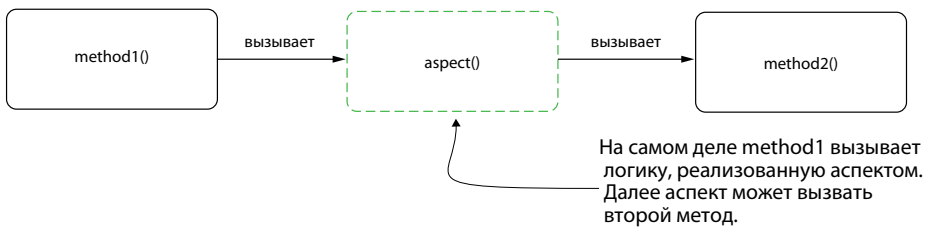


Рис. 2.10. Логика аспекта полностью отделена от кода (приложения). Поэтому при чтении кода трудно понять, что существует дополнительная логика, которая будет выполняться. Подобные случаи скрытой логики выполнения могут вносить путаницу при анализе конкретного функционального средства

Листинг 2.3. Основной класс вызывает метод `saveProduct()` класса `ProductController`

```

public class Main {

    public static void main(String[] args) {
        try (var c =
            new AnnotationConfigApplicationContext(ProjectConfig.class)) {
            c.getBean(ProductController.class).saveProduct("Beer");
        }
    }
}
  
```

❶ Метод `saveProduct()` вызывается со значением параметра "Beer".

В листинге 2.4 можно видеть, что метод `saveProduct()` класса `ProductController` просто вызывает метод `saveProduct()` класса `ProductService` с принятым значением параметра.

Листинг 2.4. ProductController, вызывающий ProductService

```

@Component
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    public void saveProduct(String name) {
        productService.saveProduct(name);    ❶
    }
}

```

❶ ProductController вызывает сервис и передает ему значение параметра.

В листинге 2.5 определен метод saveProduct() класса ProductService, который выводит значение параметра в консоли.

Листинг 2.5. ProductService, выводящий значение параметра

```

@Component
public class ProductService {

    public void saveProduct(String name) {
        System.out.println("Saving product " + name);    ❶
    }
}

```

❶ Выводит значение параметра в консоли.

На рис. 2.11 можно видеть, что поток выполнения достаточно прост:

- 1) метод main() вызывает метод saveProduct() компонента (bean) ProductController, передавая ему значение "Beer" как параметр;
- 2) затем метод saveProduct() компонента ProductController вызывает метод saveProduct() другого компонента ProductService;
- 3) компонент ProductService выводит значение параметра в консоли.

Вы можете вполне естественно предположить, что при запуске этого приложения будет выведено следующее сообщение:

```
Saving product Beer
```

Но когда вы запускаете проект, выводится другое сообщение:

```
Saving product Chocolate
```

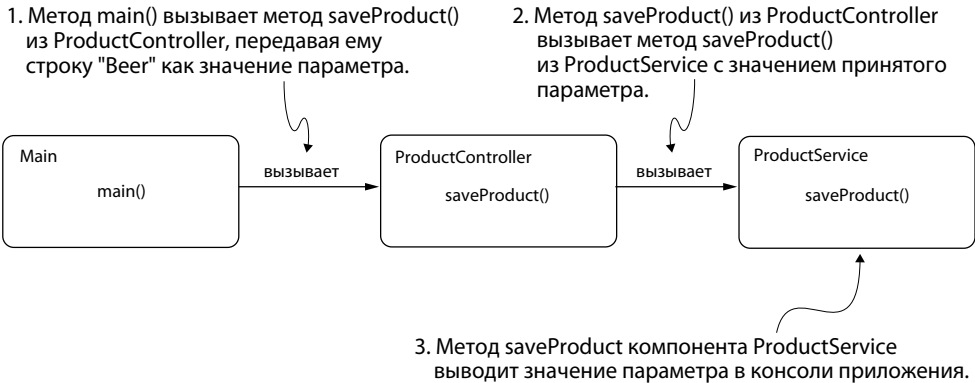



Рис. 2.11. Метод main() вызывает saveProduct компонента ProductController, передавая ему значение "Beer" как параметр. Метод saveProduct() компонента ProductController вызывает компонент ProductService, передавая ему то же значение принятого параметра. Компонент ProductService выводит значение параметра в консоли. Ожидается, что в консоли будет выведена строка "Beer"

Как такое возможно? Первое, что необходимо сделать для ответа на этот вопрос, – воспользоваться трассировкой стека выполнения, чтобы понять, кто изменил значение параметра. Добавьте точку останова в строке, которая выводит не то значение, которое вы ожидали, запустите приложение под управлением отладчика и внимательно рассмотрите трассировку стека выполнения (рис. 2.12). Вместо метода saveProduct() класса ProductService, вызванного из компонента ProductController, вы обнаружите, что аспект изменяет выполнение. Если проверить код класса аспекта, то вы, несомненно, увидите, что именно аспект несет ответственность за замену "Beer" на "Chocolate" (см. листинг 2.6).

В приведенном ниже коде (листинг 2.6) показан аспект, который изменяет выполнение, заменяя значение параметра ProductController, передаваемое в ProductService.

Листинг 2.6. Логика аспекта, которая изменяет выполнение

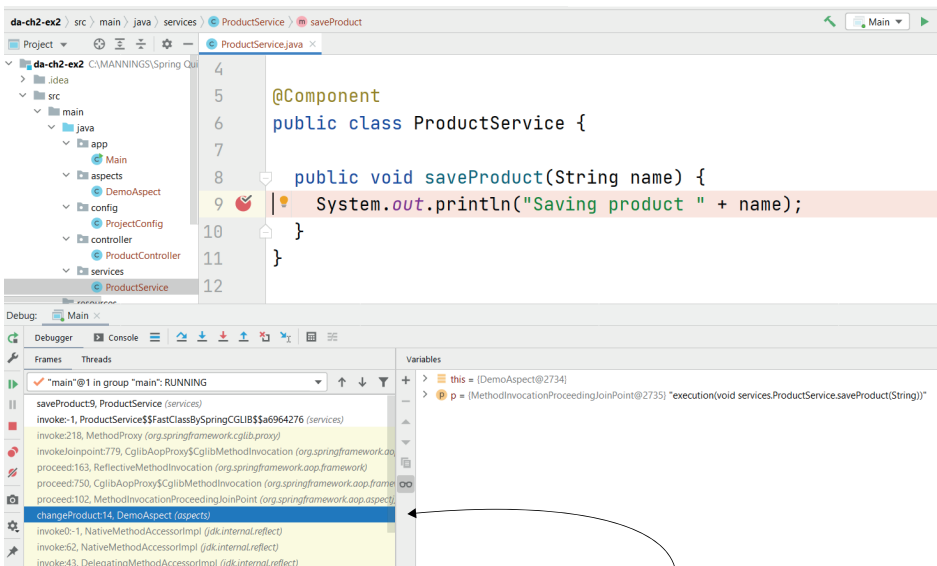
```

@Aspect
@Component
public class DemoAspect {

    @Around("execution(* services.ProductService.saveProduct(..)")
    public void changeProduct(ProceedingJoinPoint p) throws Throwable {
        p.proceed(new Object[] { "Chocolate" });
    }
}
  
```

В настоящее время аспекты представляют собой интереснейшее и весьма полезное функциональное средство в рабочих средах (фреймворках)

для приложений Java. Но если вы применяете аспекты неправильно, то они могут сделать приложение трудным для понимания и сопровождения. Разумеется, в этой книге мы рассмотрим основные методики, которые могут помочь обнаружить и понять код даже в подобных случаях. Но поверьте, если возникает необходимость применения такой методики для приложения, это означает, что сопровождение приложения становится нелегким делом. Приложение с понятным кодом (без технического долга (недоработок)) всегда является лучшим вариантом, чем приложение, в котором вы в дальнейшем вынуждены прилагать дополнительные усилия по отладке. Если вы хотите лучше понять, как работают аспекты в Spring, то я рекомендую прочесть главу 6 другой моей книги «Spring Start Here» (Manning, 2021 г.).



Трассировка стека выполнения предоставляет намного больше возможностей, чем можно было ожидать при простом чтении кода. Она явно показывает, что метод `saveProduct()` класса `ProductService` не вызывается напрямую из `ProductController`. Аспект каким-то образом выполняется между этими двумя методами.

Рис. 2.12. Трассировка стека выполнения показывает, что аспект изменил выполнение. Именно аспект становится причиной изменения значения параметра. Без использования трассировки стека было бы гораздо труднее понять, почему изменилось поведение приложения

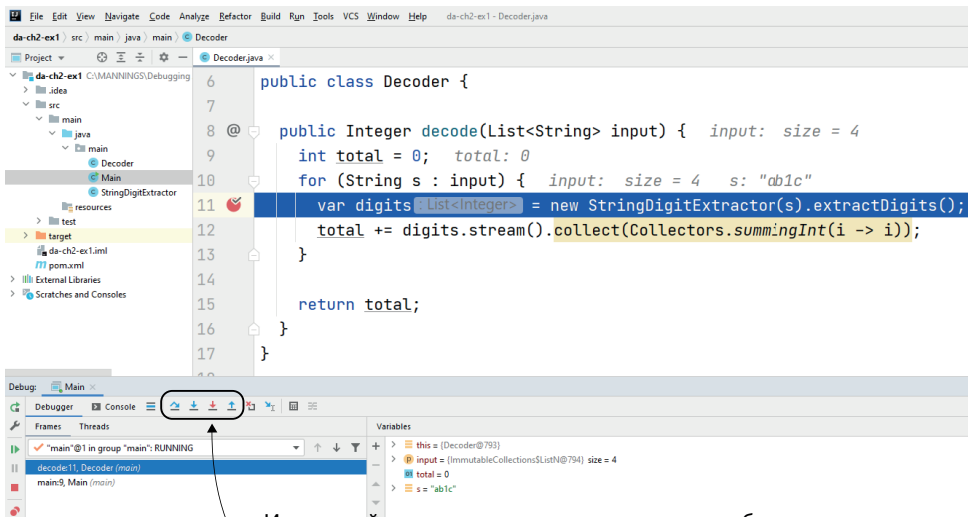
2.2.2. Перемещение по коду с использованием отладчика

В этом подразделе мы рассмотрим основные способы перемещения по коду с помощью отладчика. Вы узнаете, как применяются три основополагающие операции перемещения:

- шаг с обходом (step over) – продолжение выполнения со следующей строки кода в том же методе;
- шаг с входом (step into) – продолжение выполнения в одном из методов, вызванных в текущей строке;
- шаг с выходом (step out) – возврат выполнения в метод, который ранее вызвал метод, анализируемый в текущий момент.

Чтобы начать процесс анализа, необходимо определить первую строку кода, в которой отладчик должен приостановить выполнение. Для понимания логики требуется перемещение по строкам кода и наблюдение за тем, как изменяются данные при выполнении различных инструкций.

В каждой IDE существуют кнопки в графическом пользовательском интерфейсе (GUI) и быстрые комбинации клавиш для операций перемещения. На рис. 2.13 показано, как выглядят эти кнопки в GUI IntelliJ IDEA Community, которую я использую.



Используйте инструкции перемещения, чтобы приказать отладчику продолжить выполнение. Самые важные инструкции перемещения: шаг с обходом (step over), шаг с входом (step in) и шаг с выходом (step out).

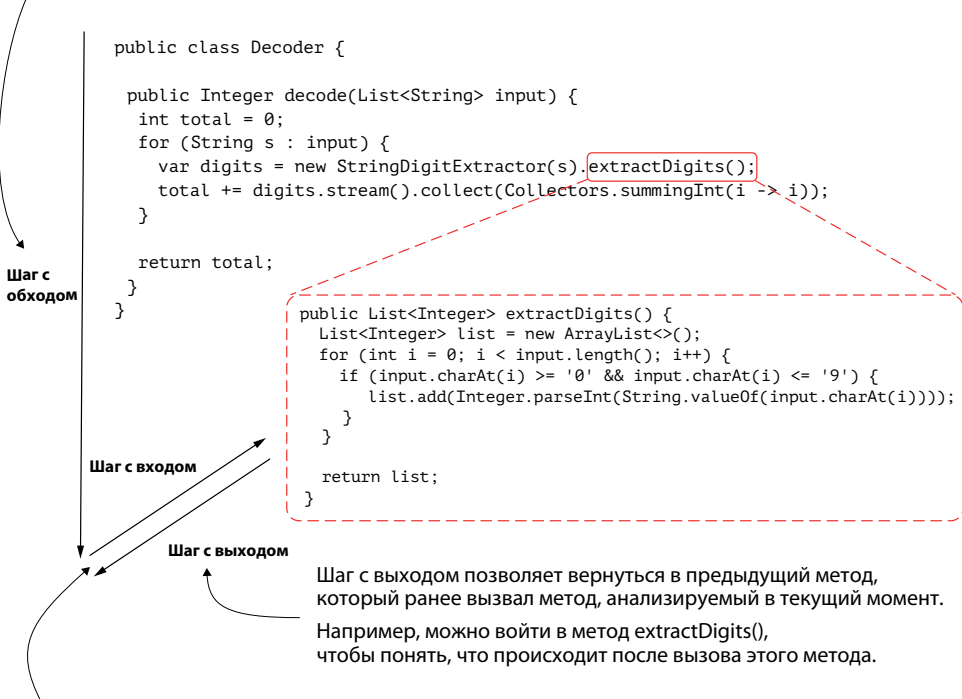
Рис. 2.13. Операции перемещения помогают управлять «шагами» по логике приложения, чтобы определить, как работает код. Для перемещения по коду можно воспользоваться кнопками в GUI IDE или быстрыми комбинациями клавиш, связанных с операциями перемещения



СОВЕТ. Даже если сначала вам кажется более простым использование кнопок в GUI IDE, я рекомендую применять вместо них быстрые комбинации клавиш. Если вы привыкнете и будете уверенно пользоваться комбинациями клавиш, то обнаружите, что это гораздо быстрее, чем манипуляции мышью.

На рис. 2.14 наглядно показаны операции перемещения. Операцию шага с обходом (step over) можно использовать для перехода в следующую строку того же (текущего) метода. Вообще говоря, это наиболее часто применяемая операция перемещения.

Шаг с обходом позволяет продолжить выполнение в текущем методе со следующей строки кода без углубления в какие-либо подробности из текущей строки.



Шаг с входом позволяет войти в инструкцию, на которой в текущий момент приостановлено выполнение.

Например, если вы вошли в метод `extractDigits()`, то можете использовать шаг с выходом для возврата в метод `decode()`, где ранее выполнялся анализ.

Рис. 2.14. Операции перемещения. Шаг с обходом позволяет перейти к следующей инструкции в текущем методе. Если необходимо начать новый план анализа и углубиться в подробности конкретной инструкции, то можно воспользоваться операцией шага с входом. А вернуться к предыдущему плану анализа можно с помощью операции шага с выходом

Иногда необходимо лучше понять, что происходит в конкретной инструкции. В рассматриваемом здесь примере может потребоваться вход в метод `extractDigits()`, чтобы полностью понять, что он делает. В таких случаях применяется операция шага с входом. Если необходимо вернуться в метод `decode()`, можно применить шаг с выходом.

Также можно визуально представить эти операции в трассировке стека выполнения, как показано на рис. 2.15.

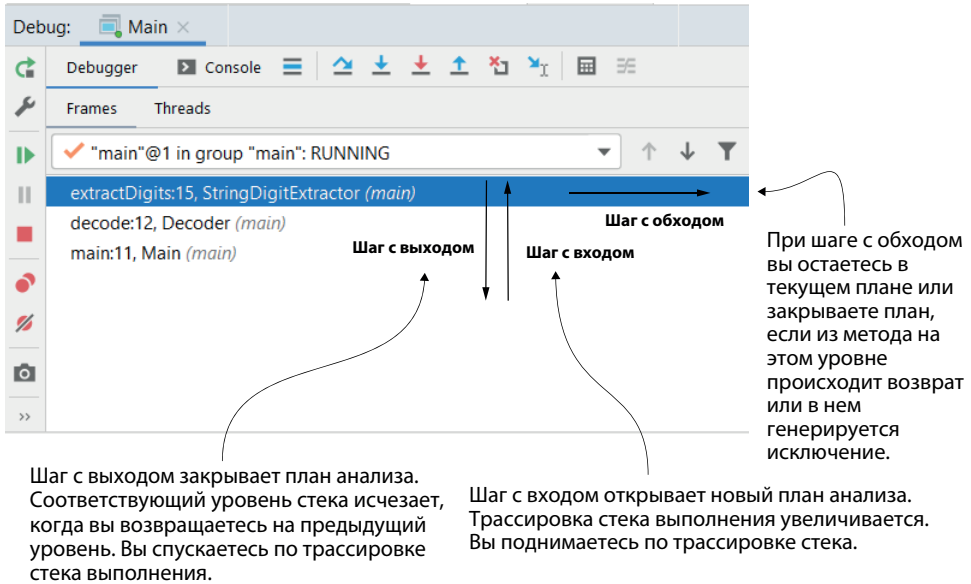


Рис. 2.15. Операция перемещения, показанная с точки зрения трассировки стека выполнения. При шаге с выходом вы спускаетесь по трассировке стека и закрываете план анализа. При шаге с входом вы открываете новый план анализа, поднимаетесь по трассировке стека, и она увеличивается. При шаге с обходом вы остаетесь в том же плане анализа. Если метод завершается (происходит возврат из него или генерируется исключение), то шаг с выходом закрывает план анализа, а вы спускаетесь по трассировке стека, как если бы вы действительно выполнили шаг с выходом

В идеальном случае вы начинаете с использования операции шага с обходом, если это возможно, пытаясь понять, как работает рассматриваемый фрагмент кода. Чем больше вы выполняете шагов с входом, тем больше планов анализа открываете, следовательно, процесс анализа усложняется (см. рис. 2.16). Во многих случаях вполне можно понять, что делает конкретная строка кода, выполнив только шаг с ее обходом и наблюдая вывод результата.

На рис. 2.17 показан результат использования операции перемещения шаг с обходом. Выполнение приостанавливается в строке 12, т. е. на одну строку ниже строки, в которой мы изначально остановили отладчик с помощью точки останова. Теперь переменная `digits` также инициализирована, и вы можете видеть ее значение.

Попробуйте продолжить выполнение несколько раз. Вы заметите, что в строке 11 для каждого строковых входных данных результатом является список, содержащий все цифры в заданной строке. Часто логика становится достаточно простой для понимания при простом анализе выходных данных после нескольких выполнений. Но что, если вы не можете понять, что именно делает строка, просто выполняя ее?



Рис. 2.16. Кинофильм «Начало» (Inception, 2010 г.) отображает идею сна во сне. Чем больше уровней погружения в сон, тем дольше вы остаетесь там. Эту идею можно сравнить с выполнением шага с входом в метод и открытием нового уровня анализа. Чем глубже вы погружаетесь в методы, тем больше времени потратите на анализ кода

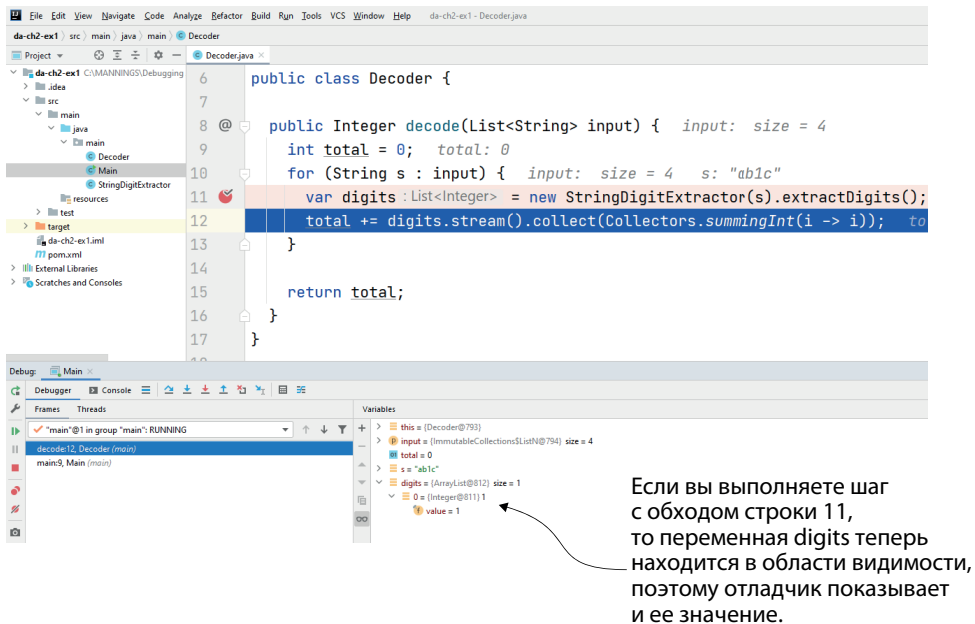


Рис. 2.17. При шаге с обходом строки выполнение продолжается в текущем методе. В нашем случае выполнение приостановлено на строке 12, и вы можете видеть значение переменной `digits`, которая была инициализирована в строке 11. Это значение можно использовать для логического вывода о том, что делает строка 11, не углубляясь в подробности

Если вы не понимаете, что происходит, необходимо исследовать такую строку более подробно. Это должно быть самым последним средством, потому что требуется открытие нового плана анализа, который усложняет весь процесс в целом. Но когда нет другого варианта выбора, вы можете выполнить шаг с входом в инструкцию, чтобы узнать более подробно, что делает этот код. На рис. 2.18 показан результат шага с входом в строку 11 класса Decoder:

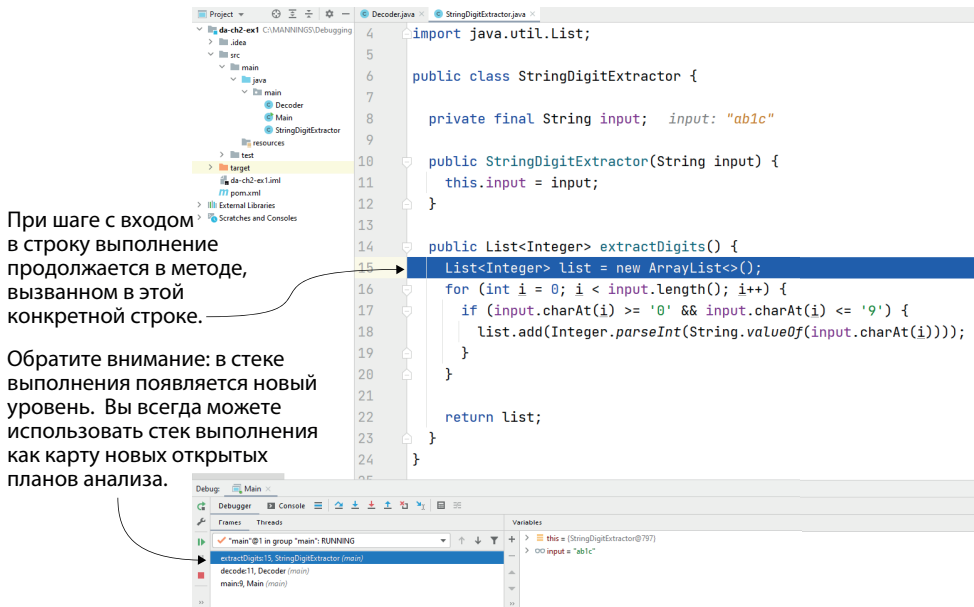


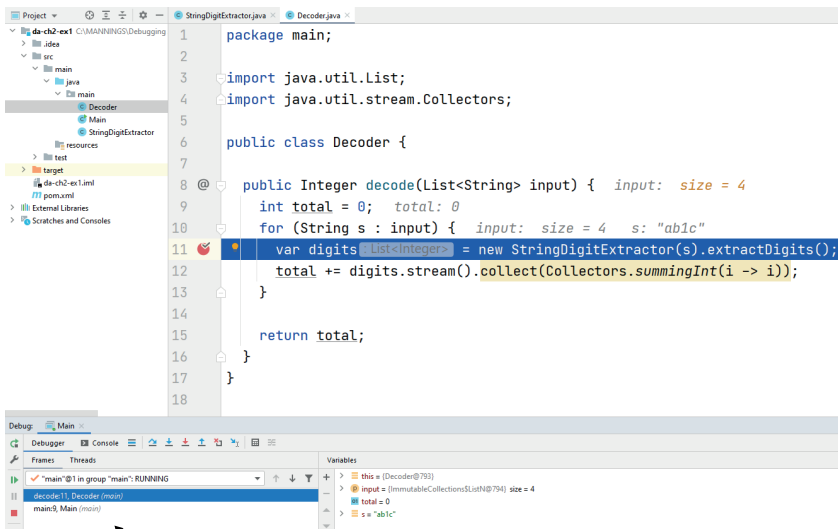
Рис. 2.18. Шаг с входом позволяет наблюдать полностью все выполнение текущей инструкции. Открывается новый план анализа, позволяющий выполнить подробный синтаксический анализ (парсинг) логики в этой конкретной инструкции. Можно воспользоваться трассировкой стека выполнения для повторного отслеживания потока выполнения

Если вы выполнили шаг с входом в инструкцию, сначала потратьте некоторое время на чтение кода, скрывающегося за этой строкой. Во многих случаях просмотра кода достаточно, чтобы сразу понять, что происходит, после чего можно вернуться туда, где вы находились перед шагом с входом. Я часто наблюдаю, как обучающиеся слишком поспешно начинают отладку метода, в который они вошли, вместо того чтобы перевести дыхание и прочитать этот фрагмент кода. Почему так важно сначала прочитать код? Потому что шаг с входом в метод открывает новый план анализа, и если вы хотите работать эффективно, то необходимо повторить все этапы анализа:

- 1) прочитать код метода и найти первую строку, которая вам непонятна;
- 2) добавить точку останова в эту строку кода и начать анализ с нее.

Если вы сначала прочитаете код, то часто выясняется, что нет необходимости в продолжении этого плана анализа. Если вы уже поняли, что происходит, то просто нужно вернуться туда, где вы находились ранее. Это можно сделать с помощью операции шага с выходом. На рис. 2.19 показано, что происходит при шаге с выходом из метода `extractDigits()`: выполнение возвращается в предыдущий план анализа в методе `decode(List<String> input)`.

При выполнении шага с выходом из метода `extractDigits()` выполнение возвращается в предыдущий план анализа.



Трассировка стека выполнения показывает, что план выполнения метода `extractDigits()` был закрыт, и выполнение возвращено в метод `decode()`.

Рис. 2.19. Операция шага с выходом позволяет закрыть текущий план анализа и вернуться в предыдущий в трассировке стека выполнения. Применение шага с выходом помогает сэкономить время, поскольку нет необходимости в выполнении шагов с обходом каждой инструкции до тех пор, пока текущий план выполнения закроется сам по себе. Шаг с выходом предлагает ускоренный способ возврата в предыдущий план выполнения, где вы проводили анализ



СОВЕТ. Операция шага с выходом может сэкономить ваше время. При входе в новый план анализа (после шага с входом в строку кода) сначала прочитайте новый фрагмент кода. Как только поймете, что делает этот код, сразу выполняйте шаг с выходом из текущего плана анализа.

Почему следующая строка выполнения не всегда является следующей строкой (кода)?

При обсуждении перемещения по коду с помощью отладчика я часто говорю о «следующей строке выполнения». Хотелось бы убедиться в том, что я четко обозначаю различие между «следующей строкой (кода)» и «следующей строкой выполнения».

Следующая строка выполнения – это строка кода, которую приложение выполнит прямо сейчас. Когда мы говорим, что отладчик приостановил выполнение в строке 12, то следующей строкой (кода) всегда является строка 13, но следующая строка выполнения может быть другой. Например, если в строке 12 не сгенерировано исключение, как показано на рис. 2.20, то следующей строкой выполнения будет 13, но при генерации исключения в строке 12 следующей строкой выполнения становится строка 18. Этот пример можно найти в проекте da-ch2-ex3.

При использовании операции шага с обходом выполнение продолжится в следующей строке выполнения.

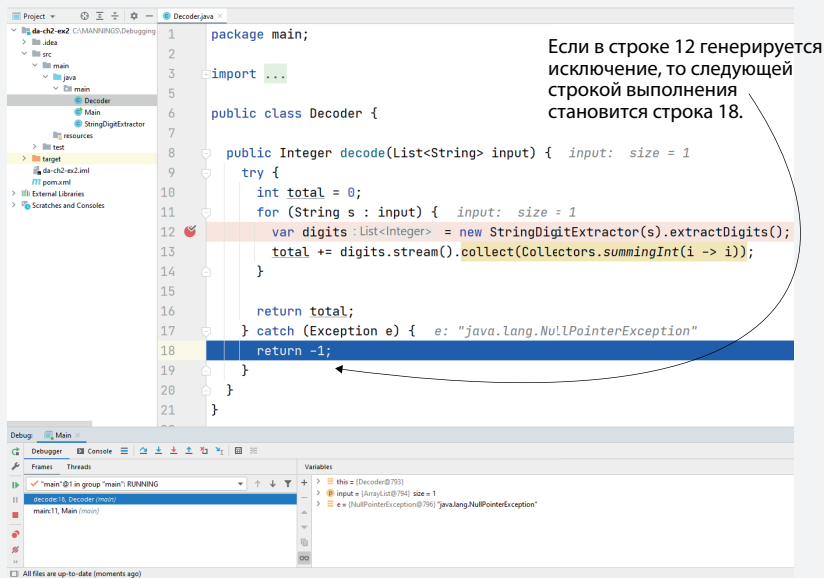


Рис. 2.20. Здесь мы выполняем шаг с обходом из строки 12, и строка 12 генерирует исключение, поэтому выполнение продолжается в строке 18, которая становится следующей строкой выполнения. Другими словами, следующая строка выполнения не всегда является следующей строкой (кода)

2.3. Когда применения отладчика может оказаться недостаточно

Отладчик является превосходным инструментом, который может помочь при анализе, обеспечивая перемещение по коду, чтобы понять, как он работает с данными. Но не весь код можно проанализировать с помощью отладчика. В этом разделе мы рассмотрим сценарии, в которых использование отладчика невозможно или недостаточно. Вы должны знать о подобных случаях, чтобы не тратить понапрасну время, пытаясь применить отладчик.

Ниже перечислены некоторые из наиболее часто встречающихся сценариев анализа, в которых использование отладчика как основного инструмента обычно не является правильным подходом:

- анализ проблем с выходными данными, если вам неизвестно, какая часть кода создает выходные данные;
- анализ проблем производительности;
- анализ критических ошибок (сбоев) при аварийном завершении приложения;
- анализ многопоточных реализаций.



СОВЕТ. Всегда следует помнить, что самым главным предварительным условием использования отладчика является знание той точки, в которой необходимо приостановить выполнение.

Прежде чем начать отладку, необходимо найти ту часть кода, которая создает неправильные выходные данные. В зависимости от приложения может оказаться более простым поиск места, где что-то происходит в реализованной логике. Если приложение содержит понятные проектные решения классов, то относительно легко обнаружить часть, отвечающую за вывод результата. Если проектные решения классов приложения недостаточно проработаны, то, вероятнее всего, более трудно будет найти требуемые места, с которых можно начать использование отладчика. В следующих главах вы изучите несколько других методик. Некоторые из них, например профилирование приложения или использование заглушек, помогут определить, где начать анализ с применением отладчика.

Проблемы производительности – это особая разновидность проблем, которые обычно невозможно анализировать с помощью отладчика. Медленная работа приложений или их полное зависание – часто встречающиеся проблемы производительности. В большинстве случаев методики профилирования и журналирования (которые рассматриваются в главах 5–9) помогут устранить проблемы в таких сценариях. В более конкретных ситуа-

циях, в которых приложение полностью блокируется, получение и анализ дампа потоков обычно является самым простым и очевидным способом анализа. Анализ дампов потоков рассматривается в главе 10.

Если в приложении возникает серьезная проблема и выполнение прерывается (аварийное завершение приложения), то невозможно использовать отладчик в коде. Отладчик позволяет наблюдать приложение в процессе выполнения. Если приложение уже не выполняется, то отладчик явно не сможет помочь. В зависимости от того, что именно произошло, возможно, потребуется аудит журналов, который будет описан в главе 5, или анализ дампов потоков или памяти, о котором вы узнаете в главах 10 и 11.

Большинство разработчиков считают многопоточные реализации наиболее трудными для анализа. Подобные реализации могут с легкостью подвергаться внешним воздействиям при вмешательстве с применением таких инструментов, как отладчик. Такое вмешательство создает эффект Гейзенберга (описанный в главе 1): при использовании отладчика поведение приложения отличается от поведения при отсутствии вмешательства извне. Однако в следующих главах вы узнаете, что иногда можно ограничить анализ одним потоком (изолировав его) и воспользоваться отладчиком. Но в большинстве случаев вам придется применять комплект методик, включающий отладку, имитацию и создание заглушек, а также профилирование, чтобы понять поведение приложения в самых сложных сценариях.

2.4. Резюме

- Каждый раз, когда вы открываете новый фрагмент логики (например, входите в новый метод, который определяет собственную логику), вы открываете новый план анализа.
- В отличие от абзаца текста чтение кода не является линейной процедурой. Каждая инструкция может создавать новый план, который необходимо проанализировать. Чем сложнее исследуемая логика, тем больше планов анализа требуется открыть. Чем больше планов вы открываете, тем сложнее становится весь процесс в целом. Один из приемов ускорения процесса анализа кода – открывать как можно меньше планов.
- Отладчик – это инструмент, позволяющий приостановить выполнение приложения в конкретной строке, чтобы можно было наблюдать выполнение в пошаговом режиме и способ обработки данных. Использование отладчика может помочь немного снизить когнитивную нагрузку при чтении кода.
- Можно использовать точки останова для пометки тех строк кода, в которых отладчик должен приостановить выполнение приложения, чтобы появилась возможность оценить значения всех переменных в текущей области видимости.

- Можно выполнить шаг с обходом строки, т. е. с переходом к следующей строке выполнения в текущем плане анализа, или шаг с входом в строку, т. е. с углублением в подробности инструкции, на которой отладчик приостановил выполнение. Необходимо минимизировать количество шагов с входом в строку и в большей степени полагаться на операцию шага с выходом. При каждом шаге с входом в строку маршрут анализа увеличивается, а процесс потребляет больше времени.
- Даже если использование мыши и GUI IDE для перемещения по коду сначала кажется более удобным, освоение применения быстрых комбинаций клавиш для операций перемещения поможет вам быстрее выполнять отладку. Я рекомендую освоить быстрые комбинации клавиш в IDE, в которой вы предпочитаете работать, и использовать их вместо управления перемещением с помощью мыши.
- После шага с входом в строку сначала прочитайте код и попытайтесь понять его. Если вы сразу можете понять, что происходит, то используйте операцию шага с выходом, чтобы вернуться в предыдущий план анализа. Если невозможно понять, что происходит, определите первую непонятную инструкцию, добавьте в нее точку останова и начинайте отладку с этого места.

Глава 3

.....

Поиск главных причин возникновения проблемы с использованием расширенных методик отладки

Темы:

- использование условных точек останова для анализа особых сценариев;
- использование точек останова для журналирования отладочных сообщений в консоли;
- изменение данных во время отладки, чтобы заставить приложение работать особым образом;
- повторное выполнение определенной части кода во время отладки.

В главе 2 мы начали рассматривать наиболее часто применяемые способы использования отладчика. При отладке конкретного фрагмента реализованной логики разработчики часто используют операции перемещения по коду: шаг с обходом, шаг с входом и шаг с выходом. Знание того, как правильно применять эти операции при анализе кода, помогает лучше понять его работу или быстрее обнаружить проблему.

Но многие разработчики не всегда знают о том, что на самом деле отладчик является более мощным инструментом. Иногда разработчики ведут нелегкую борьбу по отладке кода, используя только простейшие средства перемещения, хотя могли бы сэкономить огромное количество времени, применяя некоторые из других (менее известных) методов, предлагаемых отладчиком.

В этой главе вы узнаете, как извлечь максимум пользы, применяя возможности, предоставляемые отладчиком:

- условные точки останова;
- точки останова как события, фиксируемые в журнале;
- изменение данных, находящихся в памяти;
- отбрасывание фреймов (кадров) выполнения.

Мы рассмотрим некоторые не самые простые способы перемещения по анализируемому коду, и вы узнаете, как и когда следует применять такие способы. При рассмотрении этих методов анализа будут использоваться примеры кода, чтобы вы поняли, как можно применять их для экономии времени и когда следует избегать их.

3.1. Минимизация времени анализа с помощью условных точек останова

В этом разделе рассматривается использование условных точек останова для временной приостановки выполнения приложения в некоторой строке кода при определенных (заданных) условиях.



ОПРЕДЕЛЕНИЕ. Условная точка останова (conditional breakpoint) – это точка останова, которую вы связываете с некоторым условием, так что отладчик приостанавливает выполнение, только если это условие выполнено. Условные точки останова полезны в сценариях анализа, где вас интересует только работа части кода при конкретных значениях. Использование условных точек останова там, где это уместно, экономит рабочее время и помогает быстрее понять, как работает приложение.

Рассмотрим пример, чтобы лучше понять, как работают условные точки останова, и определить типичные случаи, в которых потребуется их применение. В листинге 3.1 представлен метод, возвращающий сумму цифр в списке значений типа String. Вероятно, вам уже знаком этот метод из главы 2. Здесь он используется для демонстрации применения условных точек останова. Затем мы сравним этот упрощенный пример с аналогичными ситуациями, которые могут встречаться в реальной практике. Пример можно найти в проекте da-ch3-ex1, прилагаемом к этой книге.

Листинг 3.1. Использование условных точек останова для анализа

```
public class Decoder {  
  
    public Integer decode(List<String> input) {
```

```

try {
    int total = 0;
    for (String s : input) {
        var digits = new StringDigitExtractor(s).extractDigits();
        var sum = digits.stream().collect(Collectors.summingInt(i -> i));
        total += sum;
    }

    return total;
} catch (Exception e) {
    return -1;
}
}
}

```

При отладке фрагмента кода часто вас интересует только то, как работает логика при определенных значениях. Допустим, вы предполагаете, что реализованная логика не работает нормально в каком-то определенном случае (например, некоторая переменная содержит конкретное значение), но необходимо подтвердить это предположение. Или вам просто нужно понять, что происходит в конкретной ситуации, чтобы получить более полную картину общей функциональности.

Предположим, что в рассматриваемом здесь случае необходимо всего лишь понять, почему переменная `sum` иногда обнуляется. Как можно обеспечить работу в таком особом случае? Можно было бы применить шаг с обходом для перемещения по коду до того момента, когда вы обнаружите, что метод возвращает ноль. Вероятно, такой подход приемлем в демонстрационном примере, подобном рассматриваемому здесь (достаточно небольшому). Но в реальной практике в аналогичной ситуации вам, вероятнее всего, пришлось бы выполнять шаг с обходом многократно до тех пор, пока не возникнет ожидаемый вариант. В действительности в реальном практическом сценарии, возможно, вам даже не будет известно, когда возникает этот особый вариант, который нужно проанализировать.

Использование условных точек останова более эффективно, чем перемещение по коду в ожидании создания условий, которые вы намерены исследовать. На рис. 3.1 показано, как применить условие к точке останова в IntelliJ IDEA. Щелкните правой кнопкой мыши по точке останова, в которой необходимо добавить условие, и введите условное выражение, применяемое к ней. Условное выражение непременно должно быть логическим выражением (т. е. выражением, при вычислении которого можно получить результат `true` (истина) или `false` (ложь)). Используя условное выражение `sum == 0` для этой точки останова, вы приказываете отладчику принять во внимание заданное условие и приостановить выполнение только в том случае, когда переменная `sum` содержит значение ноль.

В IntelliJ щелкните правой кнопкой мыши по точке останова, чтобы определить для нее условие. В рассматриваемом здесь примере отладчик останавливается в этой точке, только если значение переменной `sum` равно нулю.

Можно добавлять условие в конкретных точках останова. Отладчик рассматривает эти точки останова, только если вычисление соответствующих им условий дает результат `true` (истина).

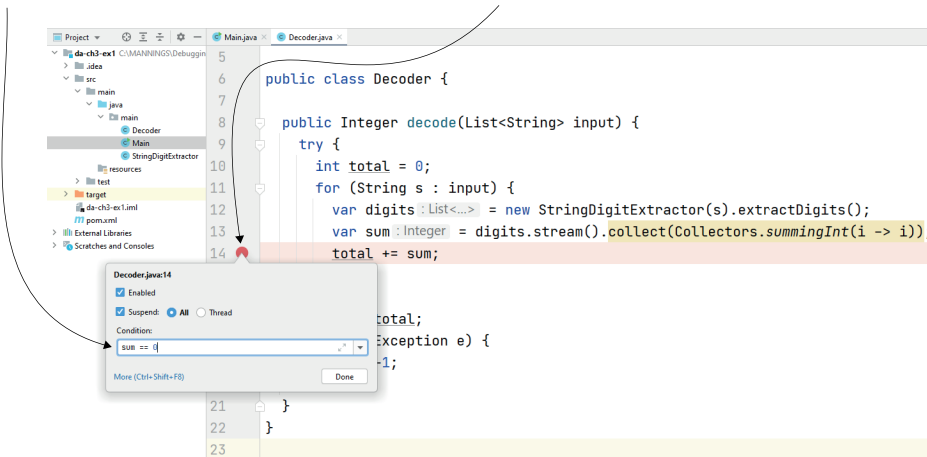


Рис. 3.1. Использование условной точки останова для приостановки выполнения только в особых случаях. Здесь необходимо приостановить выполнение в строке 14, только если значение `sum` равно нулю. Можно применить в точке останова условие, определяющее, что отладчик должен рассматривать эту точку останова, только если вычисленным состоянием является `true` (истина). Это помогает быстрее перейти к сценарию, который необходимо проанализировать

Если вы запускаете приложение под управлением отладчика, то выполнение приостанавливается, только когда в итерации цикла впервые встречается строка, не содержащая цифр, как можно видеть на рис. 3.2. Это становится причиной того, что переменная `sum` содержит значение ноль, таким образом, условное выражение в точке останова вычисляется как `true` (истина).

Условная точка останова экономит рабочее время, поскольку нет необходимости искать особый вариант, требующий анализа. Вместо этого вы позволяете приложению выполняться, а отладчик приостанавливает выполнение, когда обнаруживает соответствие определенному условию, позволяя начать анализ в этот момент. Несмотря на то что применять условные точки останова легко, кажется, что многие разработчики забывают об этой методике и тратят слишком много времени на анализ сценариев, которые можно было бы упростить с помощью условных точек останова.

Настройка условных точек останова – превосходный способ анализа кода. Но и у этого подхода есть свои недостатки. Условные точки останова могут весьма существенно повлиять на производительность при выполнении, поскольку отладчик должен постоянно перехватывать значения переменных в используемой области видимости и вычислять условия для точек останова.

Если вы запускаете приложение под управлением отладчика, то он приостанавливает выполнение на первом элементе списка параметров, не содержащем цифр (из-за которого значение переменной `sum` становится равным нулю).

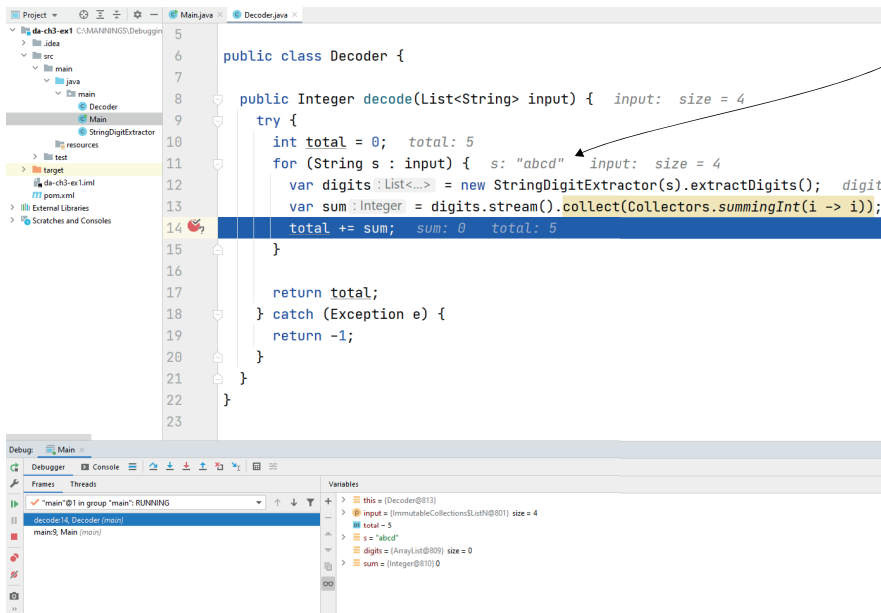


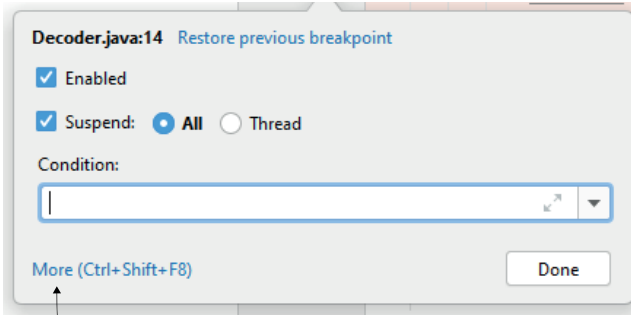
Рис. 3.2. Условная точка останова. Здесь строка 14 была выполнена несколько раз, но отладчик приостановил выполнение, только когда значение переменной `sum` стало равным нулю. Таким образом, мы пропустили все варианты, которые нас не интересуют, чтобы получить возможность начать с условий, важных для анализа



СОВЕТ. Используйте небольшое количество условных точек останова. Наиболее предпочтительно применять одновременно только одну условную точку останова, чтобы избежать весьма существенного замедления выполнения.

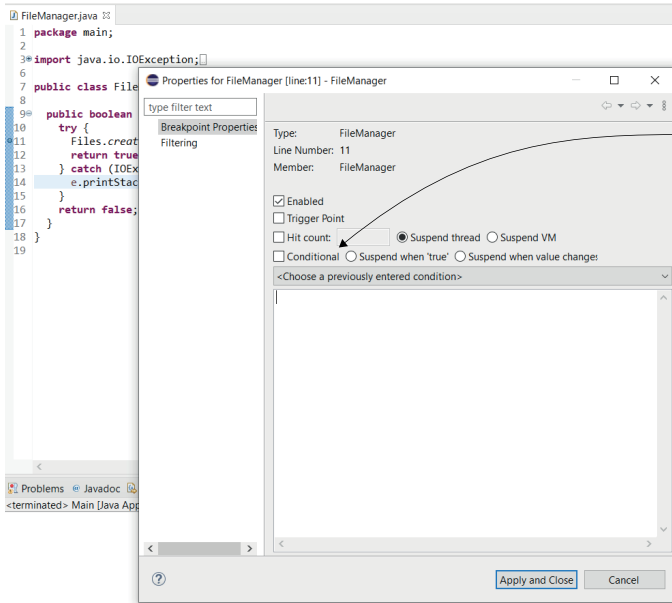
Другим способом применения условных точек останова является фиксация в журнале специфических подробностей выполнения, например разнообразных значений выражений и трассировки стека при конкретных условиях (см. рис. 3.3).

К сожалению, эта функция доступна только в некоторых IDE. Например, даже при возможности применения условных точек останова в Eclipse описанным выше способом Eclipse не позволит использовать точки останова только для фиксации в журнале подробностей выполнения (см. рис. 3.4).



Щелкните по кнопке **More** для определения расширенной конфигурации для условной точки останова.

Рис. 3.3. Для применения расширенной конфигурации точки останова в IntelliJ можно щелкнуть по кнопке **More** (Дополнительно)



В Eclipse можно определить условную точку останова. Но вы не можете записывать в журнал специфические подробности выполнения вместо приостановки выполнения потока.

Рис. 3.4. Не все IDE предоставляют одинаковые инструменты отладки.

Везде поддерживаются основные операции, но некоторые средства могут отсутствовать, например запись в журнал подробностей выполнения вместо приостановки. В Eclipse можно определить условные точки останова, но нельзя использовать функцию журналирования

У вас может возникнуть вопрос: не стоит ли использовать для этих примеров только IntelliJ IDEA? Даже если в большинстве примеров этой книги используется IntelliJ IDEA, это не означает, что данная IDE лучше других. Я работал со многими IDE для Java, такими как Eclipse, Netbeans и JDeveloper. Рекомендую не ограничиваться единственной IDE, даже если она стано-

вится весьма удобной и привычной. Экспериментируйте, пробуйте разнообразные функциональные возможности, чтобы появилась возможность выбора IDE, наиболее подходящей вам и вашей группе разработки.

3.2. Использование точек останова, которые не приостанавливают выполнение

В этом разделе мы рассмотрим применение точек останова для записи в журнал сообщений, которые в дальнейшем можно использовать для анализа кода. Я часто применяю один из способов использования точек останова для записи в журнал подробностей, которые могут помочь лучше понять, что происходит при выполнении приложения без его приостановки. В главе 5 вы узнаете, что в некоторых случаях журналирование является превосходной методикой анализа. Многие разработчики тратят время на добавление инструкций журналирования в код, хотя могли бы просто воспользоваться условной точкой останова.

На рис. 3.5 показано, как сконфигурировать условную точку останова, которая не приостанавливает выполнение. Вместо этого отладчик записывает в журнал сообщение каждый раз, когда переходит к строке, помеченной точкой останова. В рассматриваемом здесь случае отладчик фиксирует в журнале значение переменной `digits` и трассировку стека выполнения.

Можно использовать точку останова для записи в журнал конкретных подробностей без приостановки выполнения.

Здесь, если значение переменной `sum` равно нулю, в консоли выводится значение переменной `digits` и трассировка стека.

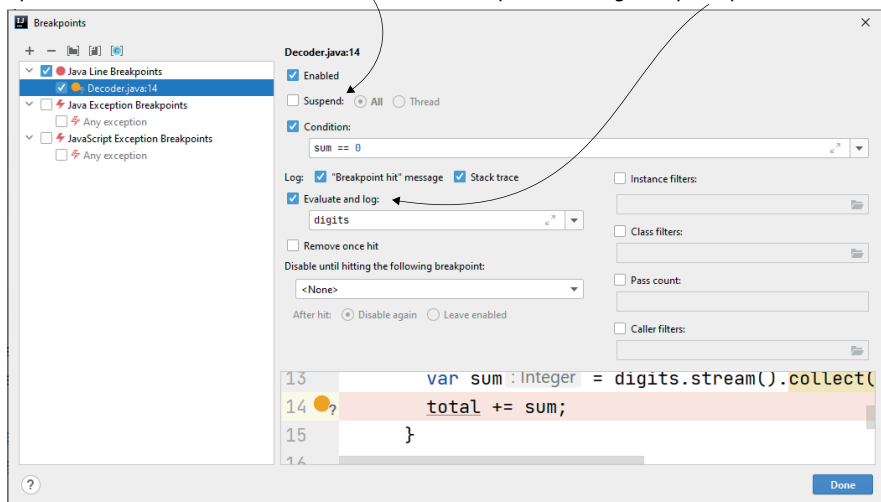


Рис. 3.5. Расширенная конфигурация условной точки останова.

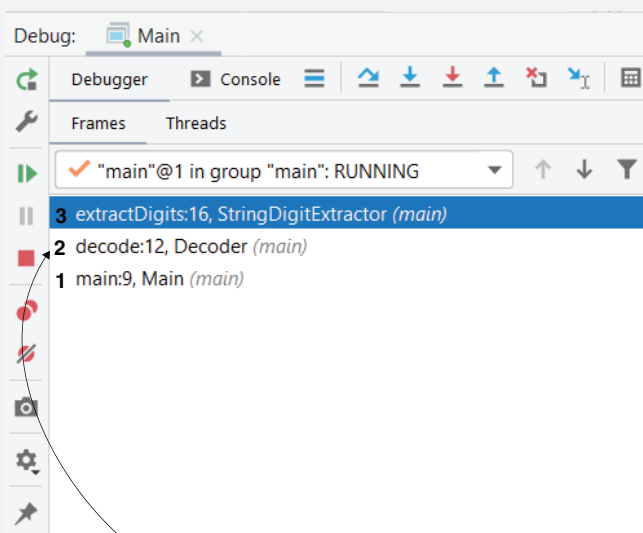
В дополнение к определению условия для этой точки останова можно приказывать отладчику не приостанавливать выполнение в этой точке. Вместо этого можно просто записать в журнал данные, необходимые для понимания работы приложения в конкретном случае

Трассировка стека выполнения: сравнение визуального и текстового представления

Обратите внимание на способ вывода в консоли трассировки стека. Вам часто будет встречаться трассировка стека выполнения в текстовом формате, а не в визуальном. Преимущество текстового представления заключается в том, что его можно сохранить в любом текстовом формате вывода, например в консоли или в файле журнала.

На рис. 3.6 показано сравнение визуального представления трассировки стека выполнения, созданного отладчиком, и соответствующего текстового представления. В обоих случаях отладчик предоставляет одинаковые важные подробности, которые могут помочь лучше понять, как была выполнена конкретная строка кода.

В рассматриваемом здесь случае трассировка стека сообщает, что выполнение началось из метода `main()` класса `Main`. Напомним, что первый уровень трассировки стека является самым нижним. В строке 9 метод `main()` вызвал метод `decode` из класса `Decoder` (уровень 2), который затем обратился к строке, помеченной точкой останова.



Breakpoint reached:
 3 at main.StringDigitExtractor.extractDigits(StringDigitExtractor.java:16)
 2 at main.Decoder.decode(Decoder.java:12)
 1 at main.Main.main(Main.java:9)

Рис. 3.6. Сравнение визуального представления трассировки стека выполнения в отладчике и соответствующего текстового представления. Трассировка стека показывает, как был вызван метод, и предоставляет достаточно подробностей, чтобы понять ход потока выполнения

На рис. 3.7 показан результат выполнения приложения с условной точкой останова, имеющей расширенную конфигурацию. Обратите внимание: отладчик вывел в консоли трассировку стека выполнения и значение переменной `digits` как пустой список `[]`. Информация такого рода может помочь в решении головоломок с кодом, анализируемым в реальных сценариях.

При наличии такой условной точки останова отладчик не останавливает выполнение. Вместо этого он выполняет журналирование значения переменной `digits` и трассировки стека выполнения, выводя их в консоли.

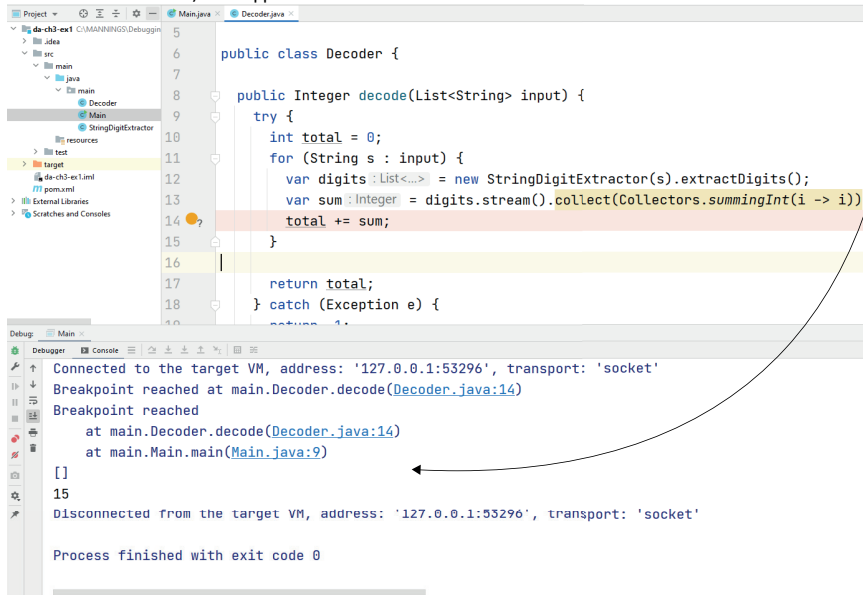


Рис. 3.7. Использование точек останова без приостановки выполнения. Вместо этого отладчик выводит сообщение, когда переходит в помеченную строку. Отладчик также выводит значение переменной `digits` и трассировку стека выполнения

3.3. Динамическое изменение сценария анализа

В этом разделе вы узнаете о другой полезной методике, упрощающей анализ кода: изменении значений переменных в текущей области видимости во время отладки. В некоторых случаях такой подход позволяет сэкономить немало времени. Начнем с рассмотрения сценариев, в которых наиболее эффективным приемом является изменение значений переменных прямо в процессе отладки. Затем я продемонстрирую использование этой методики на примере.

Ранее в этой главе рассматривались условные точки останова, которые позволяют приказать отладчику приостановить выполнение при определенных условиях (например, когда конкретная переменная имеет опреде-

ленное значение). Часто мы анализируем логику, которая выполняется за короткий интервал времени, поэтому применения условных точек останова вполне достаточно. В таких вариантах, как отладка фрагмента логики, вызываемого через конечную точку REST (особенно если вы располагаете корректными данными для воспроизведения проблемы в своей рабочей среде), вы должны просто использовать условную точку останова для приостановки выполнения в нужный момент, потому что знаете, что не придется долго ждать, когда выполняемый фрагмент будет вызван через конечную точку. Но рассмотрим следующие сценарии:

- вы анализируете проблему в процессе, который выполняется в течение длительного интервала времени. Например, это запрограммированный по расписанию процесс, для завершения которого иногда требуется больше часа. Вы предполагаете, что некоторые конкретные значения параметра являются причиной некорректных выходных данных, и хотите подтвердить свое предположение, прежде чем принять решение о том, как устранить эту проблему;
- в вашем распоряжении имеется фрагмент кода, выполняемый быстро, но вы не можете воспроизвести проблему в своей рабочей среде. Проблема возникает только в производственной рабочей среде, к которой у вас нет доступа из-за ограничений по безопасности. Вы уверены в том, что проблема возникает, когда конкретные параметры имеют определенные значения. Необходимо доказать, что ваше предположение верно.

В первом сценарии точки останова (условные или обычные) не слишком полезны. Если только вы не анализируете логику в самом начале процесса, его запуск и ожидание приостановки выполнения в строке, помеченной точкой останова, может занять слишком много времени (см. рис. 3.8).

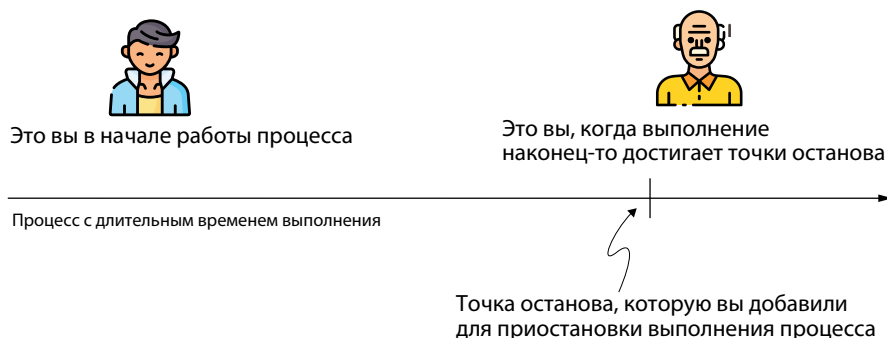
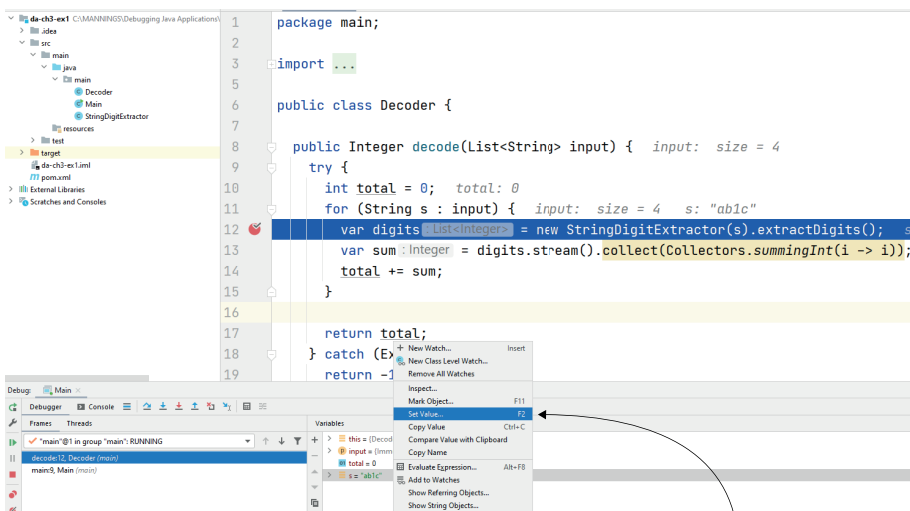


Рис. 3.8. Обычно при анализе проблем в долговременном процессе применение точек останова не является приемлемым вариантом. Переход к выполнению той части кода, которую необходимо проанализировать, может занимать много времени, а если вам приходится перезапускать такой процесс несколько раз, то вы определенно потратите слишком много времени на это

Во втором сценарии иногда возможно использование точек останова. В главе 4 мы рассмотрим удаленную отладку, и вы узнаете, как и когда удаленная отладка становится полезной методикой анализа. Но сейчас предположим, что в данном случае невозможно применить удаленную отладку (ведь мы пока еще ее не рассмотрели). Вместо этого если у вас есть предположение о том, что является причиной возникновения проблемы, и его необходимо подтвердить, но нет необходимых данных, то можно использовать динамическое изменение значений переменных.

На рис. 3.9 показано, как изменить данные в одной из переменных в текущей области видимости, когда отладчик приостанавливает выполнение. В IntelliJ IDEA нужно щелкнуть правой кнопкой мыши по переменной, значение которой вы хотите изменить. Завершается это действие в поле, где отладчик выводит значения переменных в текущей области видимости. Рассмотрим предыдущий пример da-ch3-ex1.



Когда отладчик приостанавливает выполнение в заданной строке, вы можете устанавливать значения переменных в текущей области видимости. Таким образом, вы можете создать собственный сценарий анализа с условиями, необходимыми в данном случае.

Рис. 3.9. Установка нового значения для переменной в текущей области видимости. Отладчик показывает значения переменных в текущей области видимости, когда приостанавливает выполнение в заданной строке. Вы также можете изменять значения для создания нового варианта анализа. В некоторых случаях такой подход помогает проверить предположения о том, что делает анализируемый код

После выбора переменной, которую вы хотите изменить, установите значение, как показано на рис. 3.10. Помните о том, что необходимо использовать значение, соответствующее типу переменной. Это значит, что если вы изменяете переменную типа `String`, то должны продолжать использовать значение типа `String`, нельзя использовать значение типа `long` или `Boolean`.

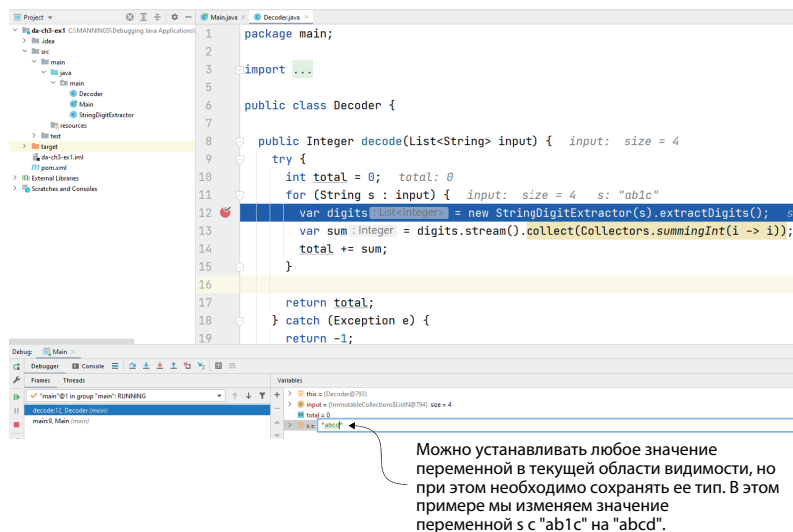


Рис. 3.10. Изменение значения переменной, чтобы наблюдать, как изменяется поведение приложения при выполнении в различных условиях

Теперь при продолжении выполнения приложение использует новое значение, как показано на рис. 3.11. Вместо вызова `extractDigits()` с значением `"abc"` приложение использует значение `"abcd"`. Возвращаемый методом список пуст, потому что строка `"abcd"` не содержит цифры.

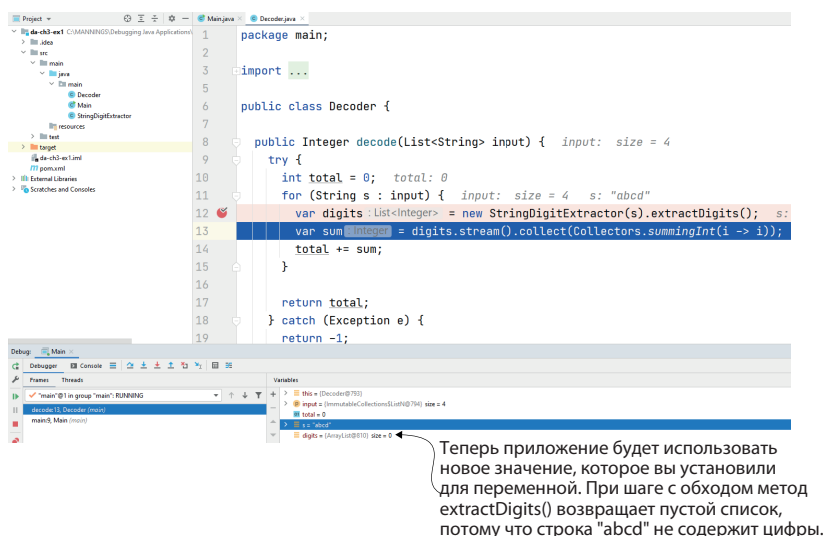


Рис. 3.11. При операции шага с обходом приложение использует новое значение, установленное для переменной `s`. Метод `extractDigits()` возвращает пустой список, потому что строка `"abcd"` не содержит цифры. Динамическое изменение значений переменных позволяет проверять различные сценарии даже при отсутствии необходимых входных данных

Сравним методику применения условных точек останова, описанную в разделе 3.1, с динамическим изменением данных. В обоих случаях необходимо сначала определить ту часть кода, которая предположительно является источником проблемы. Использовать условные точки останова можно, если:

- имеются данные, формирующие сценарий, который необходимо анализировать. В рассматриваемом здесь примере требуется значение, необходимое для определения поведения в предоставляемом списке;
- анализируемый код не слишком долго выполняется. Например, предположим, что имеется список со многими элементами, и обработка приложением каждого элемента занимает несколько секунд. В этом случае использование условной точки останова может означать, что вы затратите слишком много времени на анализ такого варианта.

Можно использовать методику изменения значений переменных, если:

- отсутствуют данные, необходимые для создания сценария, который вы намерены анализировать;
- выполнение кода отнимает слишком много времени.

Знаю, что вы сейчас думаете: а зачем вообще использовать условные точки останова? Может показаться, что следует всегда избегать применения условных точек останова, поскольку можно создать любую рабочую среду, необходимую для анализа, просто динамически изменяя значения переменных.

У обеих методик есть свои преимущества и недостатки. Изменение значений переменных может оказаться превосходным способом, если требуется изменить лишь пару значений. Но если изменения становятся более многочисленными, то сложность сценария быстро возрастает, существенно затрудняя его управление.

3.4. Повторное воспроизведение варианта анализа

Мы не можем вернуться в прошлое. Но при отладке иногда появляется возможность повторного воспроизведения процесса анализа. В этом разделе мы рассмотрим, когда и как можно «повернуть время вспять» при анализе кода с помощью отладчика. Обозначим этот подход как «отбрасывание фреймов» (dropping frames), «отбрасывание фреймов выполнения» (dropping execution frames) или «освобождение фреймов выполнения» (quitting execution frames).

Рассмотрим пример с использованием IntelliJ IDEA. Будем сравнивать эту методику с ранее описанными в предыдущих разделах этой главы, а затем также определим, когда эту методику применить невозможно.

Отбрасывание фрейма выполнения – это в действительности возврат на один уровень (ниже) в трассировке стека выполнения. Например, предположим, что выполнен шаг с входом в метод и необходимо вернуться обратно, тогда можно отбросить фрейм выполнения, чтобы вернуться туда, откуда был вызван этот метод.

Многие разработчики путают отбрасывание фрейма с шагом с выходом в основном потому, что текущий план анализа закрывается в обоих случаях, и выполнение возвращается в точку вызова метода. Но различие между этими действиями весьма существенное. При шаге с выходом из метода выполнение продолжается в текущем плане до тех пор, пока не произойдет возврат из метода или не будет сгенерировано исключение. Затем отладчик приостанавливает выполнение сразу после выхода из текущего метода.

На рис. 3.12 показано, как работает шаг с выходом на примере из проекта da-ch3-ex1. Вы находитесь в методе `extractDigits()`, который, как можно видеть в трассировке стека выполнения, был вызван из метода `decode()` класса `Decoder`. Если вы используете операцию шага с выходом, то выполнение продолжается в методе, вызвавшем `extractDigits()`, до возврата из него. Затем отладчик приостанавливает выполнение в методе `decode()`. Другими словами, шаг с возвратом похож на быструю перемотку вперед плана выполнения, чтобы закрыть его и вернуться в предыдущий.

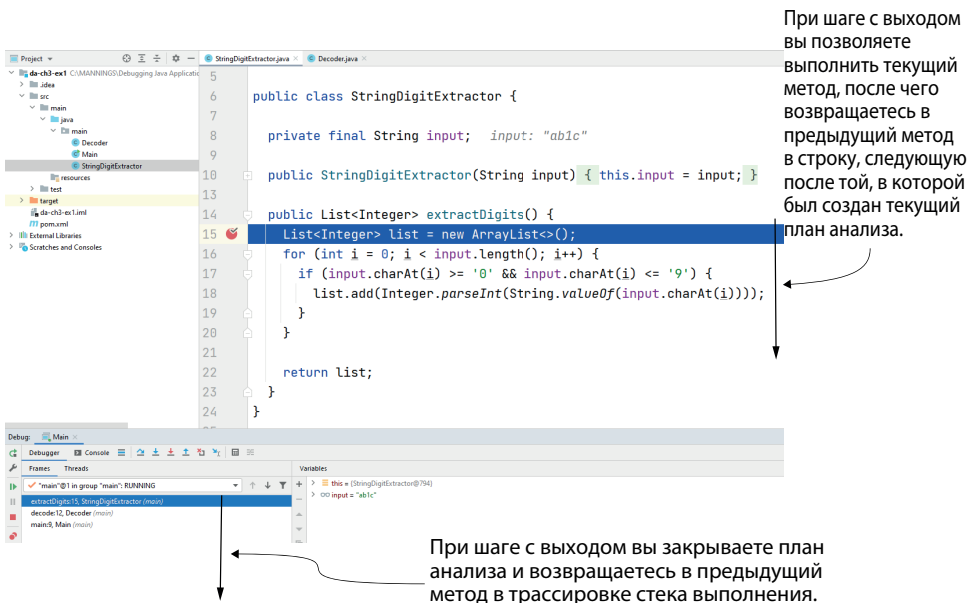


Рис. 3.12. Шаг с выходом закрывает текущий план анализа, выполняя метод, а затем приостанавливает выполнение сразу после вызова этого метода. Эта операция позволяет продолжить выполнение и вернуться на один уровень в стеке выполнения

Если вы отбрасываете фрейм выполнения, то выполнение возвращается в предыдущий план до вызова метода в отличие от шага с выходом. Таким образом, можно выполнить повторный вызов. Если шаг с выходом похож на быструю перемотку вперед, то отбрасывание фрейма выполнения (см. рис. 3.13) напоминает перемотку в обратном направлении.

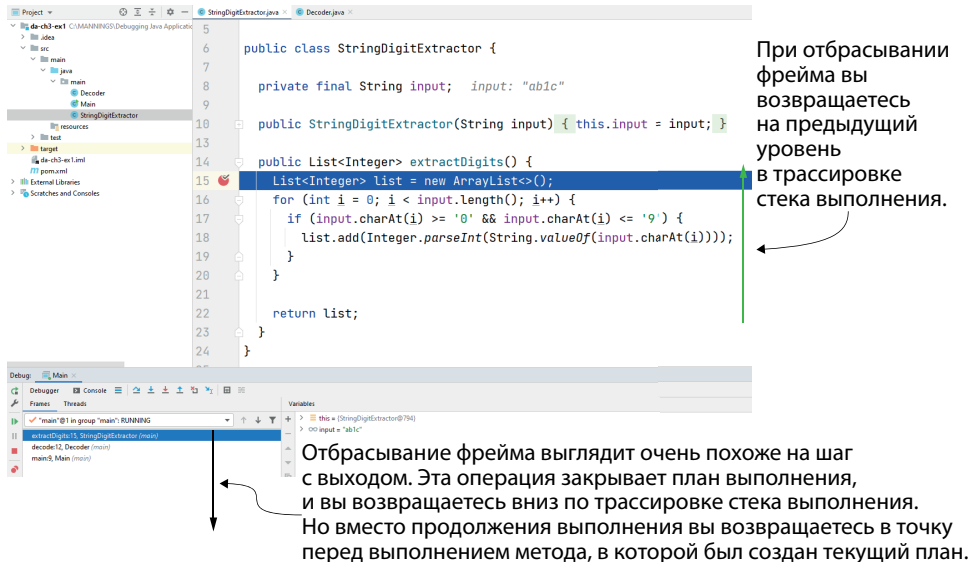
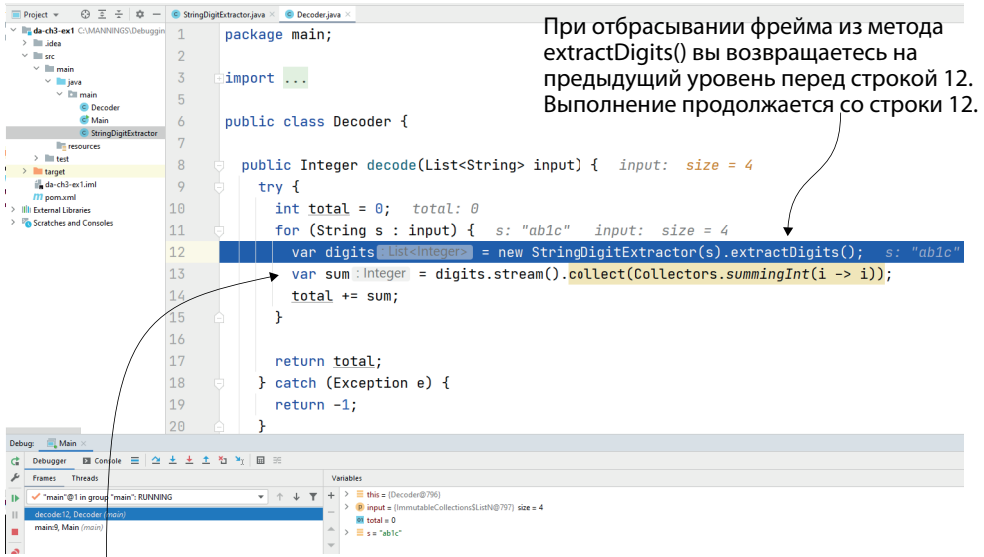


Рис. 3.13. При отбрасывании фрейма вы возвращаетесь на предыдущий уровень в трассировке стека выполнения до вызова метода. Таким образом, можно повторить выполнение метода опять, применив шаг с входом в него или шаг с обходом

На рис. 3.14 показан рассматриваемый здесь пример и сравнение шага с выходом из метода `extractDigits()` и операции отбрасывания фрейма, созданного методом `extractDigits()`. При шаге с выходом вы вернетесь в строку 12 в методе `decode()`, из которой вызывается `extractDigits()`, и следующей выполняемой отладчиком инструкцией становится строка 13. Если вы отбрасываете фрейм, то отладчик возвращается в метод `decode()`, но следующей выполняемой инструкцией останется строка 12. Фактически отладчик возвращается в строку перед выполнением метода `extractDigits()`.

На рис. 3.15 показано, как использовать функциональные возможности операции отбрасывания фрейма в IntelliJ IDEA. Чтобы отбросить текущий фрейм выполнения, щелкните правой кнопкой мыши по уровню текущего метода в трассировке стека выполнения и в контекстном меню выберите пункт **Drop Frame** (Отбросить фрейм).

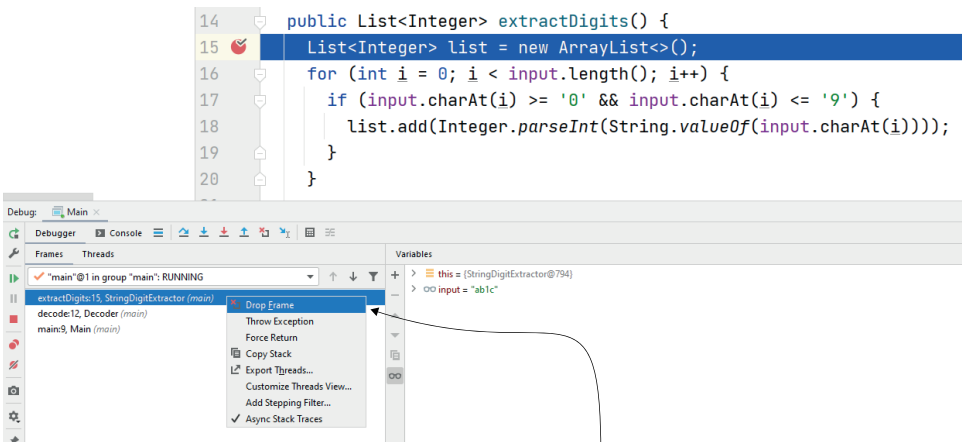


При шаге с выходом из метода `extractDigits()` вы возвращаетесь на предыдущий уровень в строку 12. Выполнение продолжается со строки 13.

Рис. 3.14. Сравнение операции отбрасывания фрейма с шагом с выходом.

При отбрасывании фрейма вы возвращаетесь в строку перед выполнением метода.

При шаге с выходом выполнение продолжается, но закрывается текущий план анализа (представленный текущим уровнем в стеке выполнения)



Чтобы отбросить фрейм выполнения и вернуться в строку перед выполнением текущего метода, щелкните правой кнопкой мыши по уровню метода в трассировке стека выполнения. Затем в контекстном меню выберите пункт **Drop Frame**.

Рис. 3.15. При использовании IntelliJ IDEA можно отбросить фрейм, щелкнув правой кнопкой мыши по уровню метода в трассировке стека выполнения, а затем выбрать в контекстном меню пункт **Drop Frame**

Почему операция отбрасывания фрейма полезна и как она помогает сэкономить время? Неважно, используете ли вы конечную точку для поиска особого варианта, который необходимо проанализировать, или сами создаете вариант, изменяя значения переменных, как описано в разделе 3.3, иногда обнаруживается, что полезно было бы повторить одно и то же выполнение несколько раз. Понять конкретный фрагмент кода не всегда просто, даже если вы используете отладчик для приостановки выполнения и продолжаете работу в пошаговом режиме. Но время от времени возврат для более внимательного наблюдения за шагами и тем, как конкретные инструкции изменяют данные, может помочь вам понять, что происходит.

Кроме того, требуется особое внимание, если вы решили повторять конкретные инструкции, отбрасывая фрейм. Этот подход может оказаться больше запутанным, чем полезным. Напомним, что если вы выполняете любую инструкцию, которая изменяет значения за пределами внутренней памяти приложения, то не можете отменить это изменение, отбрасывая фрейм. Примеры таких случаев (см. рис. 3.16):

- изменение данных в базе данных (вставка, обновление или удаление);
- изменение файловой системы (создание, удалением или изменение файлов);
- вызов другого приложения, изменяющего данные для текущего приложения;
- добавление сообщения в очередь, которую читает другое приложение, изменяющее данные для текущего приложения;
- отправка сообщения электронной почты.

Можно отбросить фрейм, результатом которого является подтверждение транзакции, изменяющей данные в базе данных, но возврат к предыдущей инструкции не отменит изменения, сделанные этой транзакцией. Если приложение вызывает конечную точку (сетевую рабочую станцию), передающую что-то в другой сервис, то итоговые изменения, сделанные при вызове этой конечной точки, невозможно отменить отбрасыванием фрейма. Если приложение отправляет сообщение электронной почты, то отбрасывание фрейма не позволит вернуть обратно это сообщение и т. д. (см. рис. 3.16).

При изменении данных за пределами приложения требуется особая осторожность и внимательность, так как иногда повторение одного и того же кода не дает одинакового результата. В качестве примера рассмотрим простой фрагмент кода (листинг 3.2 из проекта `da-ch3-ex2`). Что происходит, если вы отбрасываете фрейм после выполнения строки, которая создает файл?

```
Files.createFile(Paths.get("File " + i));
```



Рис. 3.16. При использовании операции отбрасывания фрейма может оказаться, что некоторые события невозможно отменить. Примерами являются изменение данных в базе данных, изменение данных в файловой системе, вызов другого приложения или отправка сообщения электронной почты

Созданный файл остается в файловой системе, и после второго выполнения этого кода после отбрасывания фрейма генерируется исключение (потому что такой файл уже существует). Это простой пример того, что возврат во времени при отладке не всегда оказывается полезным. Но хуже всего то, что в случаях из реальной практики эта ситуация неочевидна. Я рекомендую избегать повторения выполнения крупных фрагментов кода и, перед тем как принять решение о применении этой методики, убедиться в том, что повторяемая часть логики не выполняет внешние изменения.

Если после повторного выполнения отброшенного фрейма вы заметили различия, выглядящие необычно, вероятно, причиной этого является изменение внешних данных за пределами приложения. В крупных приложениях чаще всего не так-то просто исследовать такое поведение. Например, приложение может использовать кеш или данные журнала с доступом к определенной библиотеке для наблюдения или выполнения код, который полностью разъединяется через перехватчики (аспекты).

Листинг 3.2. Метод, который вносит изменения за пределами выполняемого приложения

```
public class FileManager {

    public boolean createFile(int i) {
        try {
```

```

        Files.createFile(Paths.get("File " + i));    ❶
        return true;
    } catch (IOException e) {
        e.printStackTrace();
    }
    return false;
}
}

```

❶ Создание нового файла в файловой системе.

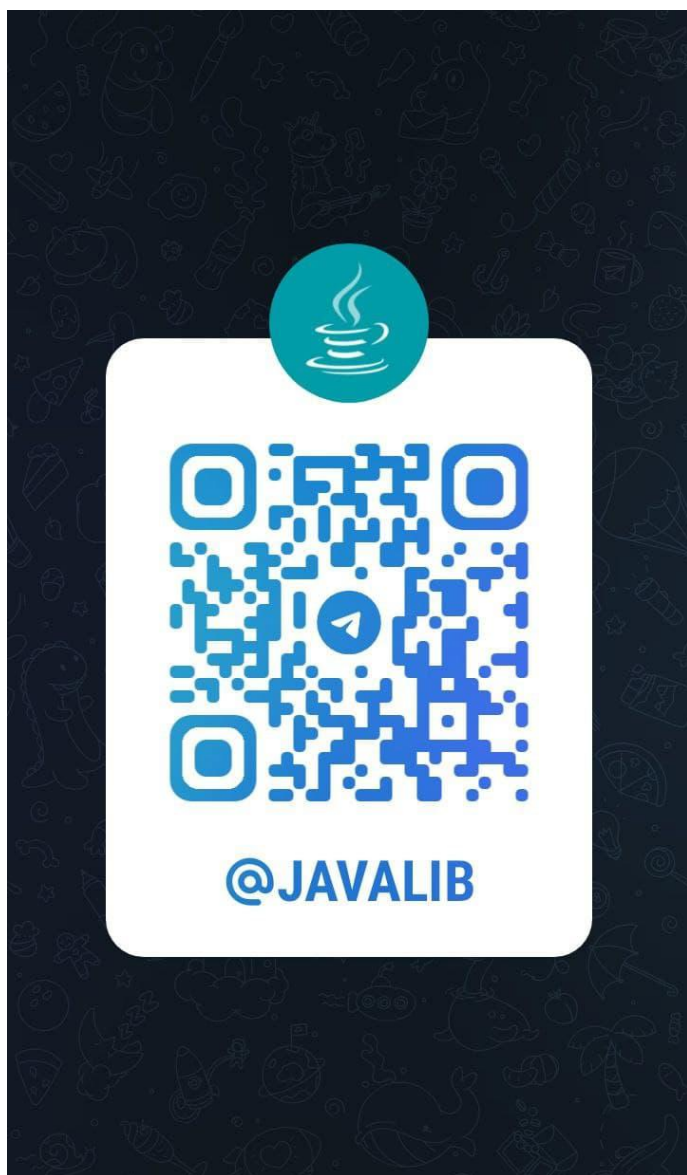
Вызов метода `Files.createFile()` создает новый файл в файловой системе. Если вы отбрасываете фрейм после выполнения этой строки, то возвращаетесь в строку перед вызовом метода `createFile()`. Но это не позволяет отменить создание файла.

3.5. Резюме

- Условная точка останова – это точка останова, связанная с логическим условием. Отладчик приостанавливает выполнение, только если заданное условие истинно, т. е. только при выполнении конкретных условий. Таким образом, вы экономите время при перемещении по коду до того момента, когда достигается требуемая точка начала анализа.
- Можно использовать точки останова для регистрации в журнале значений конкретных переменных с выводом в консоли, не приостанавливая выполнение приложения. Эта методика весьма полезна, поскольку можно добавлять сообщения в журнал, не изменяя исходный код.
- Когда отладчик приостанавливает выполнение в заданных строках кода, вы можете динамически изменять данные для создания специализированных сценариев, соответствующих требуемым вариантам анализа. Таким образом, не нужно ждать, пока выполнение достигнет условной точки останова. В некоторых случаях при отсутствии необходимой рабочей среды изменение данных при отладке позволяет сэкономить время, которое потребовалось бы на подготовку данных в текущей рабочей среде.
- Изменение значений переменных для создания специализированного сценария анализа может оказаться эффективной методикой при попытке понять только небольшой фрагмент логики долговременного процесса или при отсутствии требуемых данных в рабочей среде, где выполняется приложение. Но одновременное изменение более чем одной-двух переменных может существенно увеличить сложность и затруднить анализ.

- Можно выполнить шаг с выходом из плана анализа и вернуться в точку перед вызовом метода (из которого вы вышли). Это называется отбрасыванием фрейма, но иногда может создавать нежелательный побочный эффект. Если приложение изменило что-либо во внешней среде (например, подтвердило транзакцию и изменило некоторые записи в базе данных, изменило файл в файловой системе или выполнило REST-вызов другого приложения), то возврат к предыдущему шагу выполнения не отменит эти изменения.

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javallib>



Глава 4

Удаленная отладка приложений

Темы:

- отладка приложения, установленного в удаленной рабочей среде;
- обучение новым методикам отладки с практическим примером.

Недавно у одного из моих друзей возникла проблема: в программном обеспечении, реализацией которого он занимался, конкретная часть кода работала очень медленно. В общем случае, когда возникает такая разновидность проблем с производительностью, мы предполагаем, что причиной является интерфейс ввода-вывода (например, соединение с базой данных или файловые операции чтения/записи). Напомню, что в главе 1 было отмечено, что эти интерфейсы часто замедляют работу приложения, поэтому они становятся наиболее вероятными подозреваемыми. Но в случае с моим другом интерфейсы не являлись источником проблемы.

Причиной возникновения проблемы с производительностью стала простая генерация случайного значения (универсального уникального идентификатора [UUID], сохраняемого в базе данных). Операционная система использует аппаратуру (например, перемещения мыши, нажатия клавиш на клавиатуре и т. п.) как источники случайности, обозначаемой термином «энтропия» (entropy). Приложение применяет эту случайность для генерации случайных значений. Но при развертывании приложения в виртуальной рабочей среде, например в виртуальной машине или в контейнере (наиболее широко распространенная среда для развертывания приложений в настоящее время), операционная система имеет меньше источников для создания такой энтропии. Таким образом, иногда энтропия оказывается недостаточной для того, чтобы приложение получило возможность создания требуемых случайных значений. Такая ситуация становится при-

чиной проблем с производительностью и в некоторых случаях может оказывать отрицательное воздействие на безопасность приложения.

Этот тип проблем может оказаться действительно чрезвычайно трудным для анализа без прямого соединения с рабочей средой, в которой возникает подобная проблема. Для таких сценариев решением может стать удаленная отладка. Вы можете анализировать только некоторые определенные варианты в конкретных рабочих средах. Предположим, что ваши клиенты обнаружили некоторую проблему, но она не проявляется, когда вы выполняете то же приложение на своем компьютере. В конце концов, вы же не можете решить этот вопрос, просто сказав клиенту: «На моем компьютере все работает».

Если невозможно воспроизвести проблему на своем компьютере, то необходимо установить соединение с рабочей средой, в которой она возникает. Хотя иногда других вариантов нет, и вам приходится идти трудным путем, пытаясь устранить невоспроизводимую проблему, в иных случаях рабочая среда открыта для удаленной отладки. Удаленная отладка (remote debugging), или отладка приложения, установленного во внешней рабочей среде, – это тема текущей главы (см. рис. 4.1).

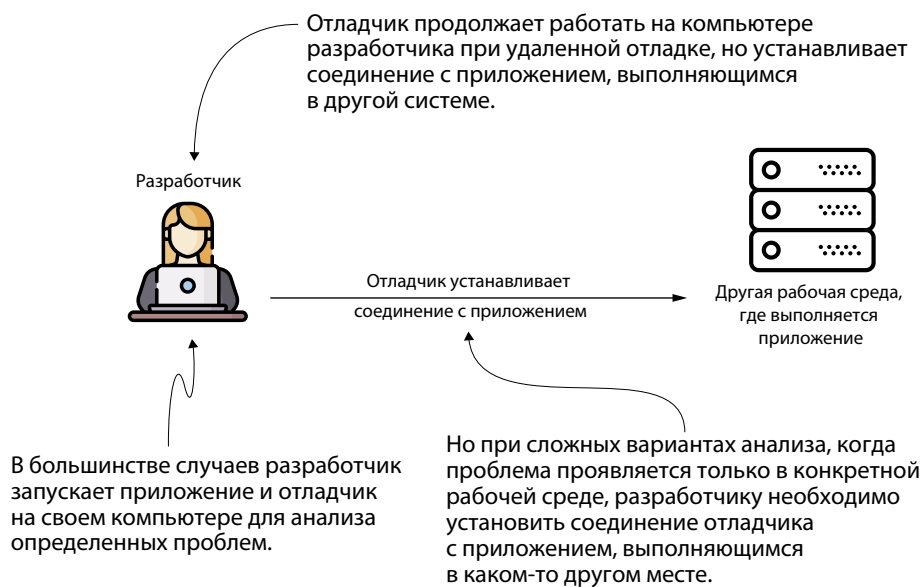


Рис. 4.1. Удаленная отладка приложения. Разработчик может запустить отладчик локально, но установить соединение с экземпляром приложения, работающим в другой рабочей среде. Эта методика позволяет разработчику анализировать проблемы, возникающие только в конкретных рабочих средах

Начнем главу с обсуждения того, что представляет собой удаленная отладка и когда появляется возможность ее применения, а также тех случаев, когда не следует использовать этот метод. Затем для практического приме-

нения этой методики рассмотрим проблему, которую необходимо проанализировать. Вы узнаете, как должно быть сконфигурировано приложение для удаленной отладки и как установить соединение и использовать отладчик для работы в удаленной рабочей среде.

4.1. Что такое удаленная отладка

В этом разделе мы рассмотрим, что такое удаленная отладка, когда ее нужно применять, а когда следует избегать ее использования. Удаленная отладка – это не что иное, как применение методик отладки, изученных в главах 2 и 3, для приложения, которое выполняется не локально в вашей системе, а работает в некоторой внешней среде. Почему необходимо применять те же методики в удаленной рабочей среде? Для ответа на этот вопрос кратко рассмотрим обычный процесс разработки программного обеспечения.

Когда разработчик реализует приложение, он пишет его не для своей локальной системы. Конечной целью создания приложения является его развертывание в производственной среде, где оно помогает пользователям решать разнообразные деловые задачи. Кроме того, при реализации программного обеспечения мы часто не развертываем приложение непосредственно в пользовательских или производственных средах. Вместо этого используются аналогичные рабочие среды для предварительного (оценочного) тестирования функциональных возможностей и точно определяется, что необходимо для реализации, прежде чем устанавливать приложения в рабочих средах, где они будут использоваться в реальном рабочем режиме с настоящими данными.

Как показано на рис. 4.2, при разработке приложения группа разработчиков использует как минимум три рабочие среды:

- *среду разработки (dev)* – рабочую среду, аналогичную той, в которой будет развертываться приложение. В основном разработчики используют эту среду для тестирования реализации новых функциональных возможностей и исправлений ошибок после разработки в локальной системе;
- *среду пользовательского теста приемки (UAT)* – после успешного тестирования в среде разработки приложение устанавливается в среде пользовательского теста приемки. Пользователи могут тестировать новые реализации и исправления ошибок и подтверждают корректную работу, прежде чем приложение передается в среду с реальными данными;
- *производственную среду (prod)* – после того как пользователи подтвердили, что новая реализация работает как ожидалось и им удобно работать с ней, приложение устанавливается в производственной среде.

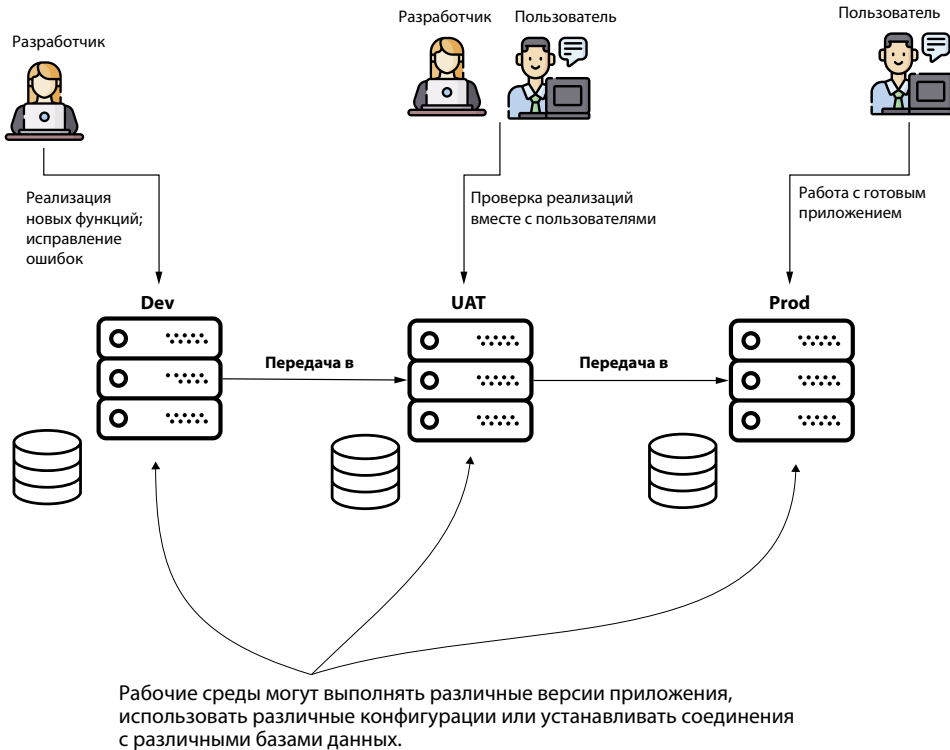


Рис. 4.2. При создании реального приложения разработчики часто используют несколько рабочих сред. Сначала они создают приложение в средах разработки (dev). Затем, когда функциональность или решение готово, они представляют приложение пользователям (или стороне, обладающей правами собственности на приложение), используя среду UAT (user acceptance test – пользовательский тест приемки). Наконец, после того как правообладатели подтвердили, что данная реализация работает, приложение устанавливается в производственной (prod) среде

Но если реализация работает на вашем локальном компьютере, но ведет себя по-другому в иной рабочей среде? Возможно, вы удивитесь тому, что приложение может работать по-разному. Даже при использовании одного и того же скомпилированного приложения можно наблюдать различия в его поведении в двух разных рабочих средах. Некоторые причины возникновения таких различий перечислены ниже:

- данные, доступные в рабочих средах приложения, отличаются. Разные среды используют различные экземпляры баз данных, разнообразные файлы конфигурации и т. д.;
- операционные системы, в которых устанавливается приложение, не одинаковы;
- оркестровка способа развертывания может быть различной. Например, в одной среде могут использоваться виртуальные машины для

развертывания, тогда как в другой организовано решение с применением контейнеров;

- настройка прав доступа может быть различной в каждой рабочей среде;
- среды могут обладать различными ресурсами (выделяемой памятью или ЦП).

Это всего лишь несколько из множества факторов, которые могут стать причиной различий в получаемых выходных данных или в поведении. В последний раз, когда я столкнулся с подобной проблемой (не так давно), приложение выводило различные данные из-за запроса, отправляемого веб-сервису приложением, применяемым в реализуемом варианте использования. Из-за проблем с безопасностью мы не могли пользоваться той же конечной точкой в среде разработки, а кроме того, не было возможности установить соединение с тем приложением в (целевой) среде, в котором возникла проблема. Такие условия серьезно затруднили анализ (честно говоря, мы даже не считали, что сама конечная точка была причиной возникновения проблемы, до тех пор, пока не начали отладку).

Удаленная отладка действительно может помочь быстрее понять поведение программного обеспечения при возникновении проблем этого типа. Но при этом всегда следует помнить о весьма важной особенности этой рекомендации: никогда не используйте удаленную отладку в производственной среде (см. рис. 4.3). Кроме того, всегда необходима полная уверенность в том, что вы понимаете основные различия между используемыми рабочими средами.

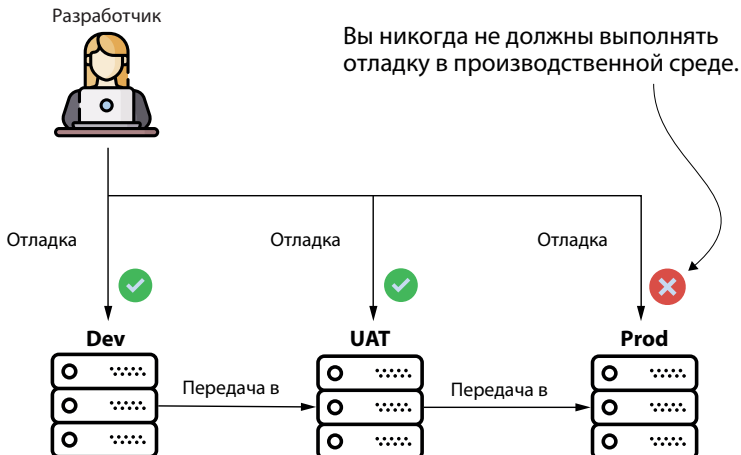


Рис. 4.3. Разработчики реализуют приложение, используя рабочие среды dev и UAT. Отладка приложений в этих средах вполне приемлема.

Но всегда следует помнить: никогда не отлаживайте приложения в производственной (prod) среде, так как это может повлиять на выполнение приложения, вмешиваться в действия пользователей и даже оставлять без защиты секретные данные, создавая угрозу для безопасности



СОВЕТ. Следует уделять особое внимание тому, как различия в рабочих средах дают вам ключи к пониманию того, что пошло не так. Можно даже сэкономить время анализа проблемы, при которой простое знание подробностей об этих различиях на основе опыта обеспечит решение такой проблемы.

В дальнейшем вы узнаете, что необходимо подключить часть программного обеспечения, которую мы называем агентом (agent), к выполнению приложения для обеспечения удаленной отладки. Некоторые последствия подключения отладочного агента (и обоснования того, почему нельзя делать это в производственной среде) описаны ниже:

- агент может замедлить выполнение приложения, и такое замедление, возможно, станет причиной проблем с производительностью;
- агент требует обмена данными с отладчиком через сеть. Чтобы обеспечить такой обмен, необходимо сделать доступными определенные порты, а из-за этого могут возникнуть проблемы с безопасностью;
- отладка определенного фрагмента кода может создавать помехи функциональности, если та же часть приложения одновременно используется где-либо еще;
- иногда отладка может заблокировать приложение на неопределенное время, и вам придется перезапускать процесс.

4.2. Анализ в удаленных рабочих средах

В этом разделе описана отладка приложения, выполняющегося в удаленной среде. Начнем с описания сценария в подразделе 4.2.1. Затем в подразделе 4.2.2, используя приложение из проекта `da-ch4-ex1`, дополняющего эту книгу, рассмотрим, как запустить приложение для удаленной отладки и как подключить отладчик к удаленно выполняющемуся приложению, применяя методики, изученные в главах 2 и 3.

4.2.1. Сценарий

Предположим, что вы работаете в группе, которая реализует и сопровождает крупное приложение со многими клиентами, использующими его для управления своими товарными запасами. Недавно ваша группа реализовала новую функцию, помогающую клиентам с легкостью управлять ценами. Группа успешно протестировала поведение в среде разработки (dev) и установила приложение в среде UAT, чтобы пользователи могли проверить корректность работы новой функции перед передачей ее в производство. Но лицо, ответственное за тестирование этой функции, сообщает вам, что веб-интерфейс, где должны выводиться новые данные, ничего не показывает.

Вы немедленно начали выяснение причин и быстро обнаружили, что проблема возникает не в клиентской части (frontend) приложения. Но поведение конечной точки в серверной (внутренней) части приложения выглядит странно. Когда конечная точка вызывается в среде UAT, возвращается код состояния HTTP-ответа 200 ОК, но приложение не возвращает данные в этом HTTP-ответе (см. рис. 4.4). Вы проверяете журналы, но в них нет ничего необычного. Поскольку проблему невозможно обнаружить в локальном режиме или в среде разработки (dev), вы решаете установить соединение своего отладчика с удаленной средой UAT, чтобы найти причину возникновения проблемы.

ПРИМЕЧАНИЕ. Даже если мы рассматриваем отладку приложения, работающего в удаленной среде, для упрощения примера используется локальная система для выполнения приложения, с которым устанавливается соединение. Поэтому на рисунках можно видеть, что я использую localhost для доступа к среде выполнения приложения. В сценарии из реальной практики приложение должно работать в другой системе, которая идентифицируется по IP-адресу или DNS-имени.

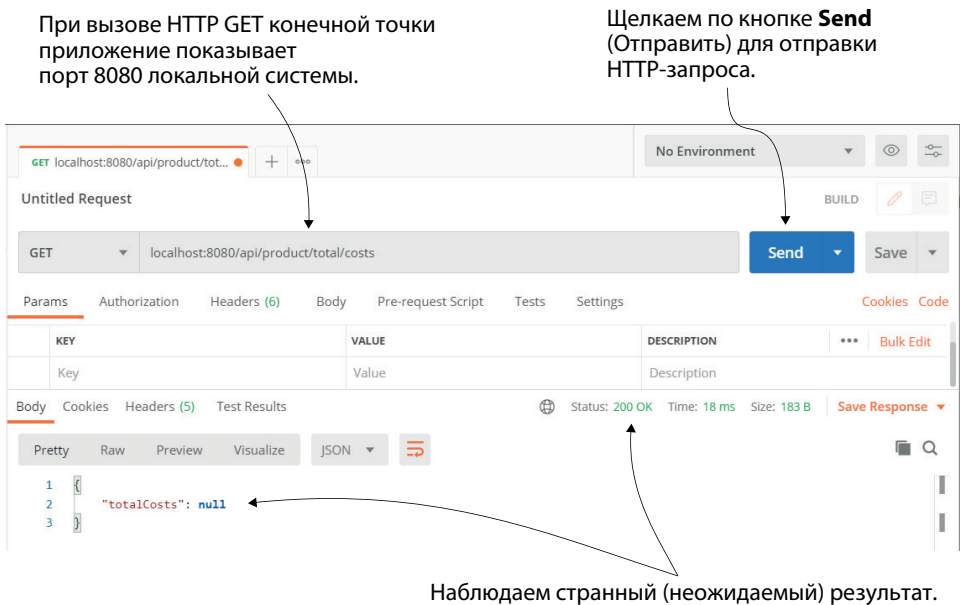


Рис. 4.4. Сценарий анализа. Конечная точка `/api/product/total/costs` должна возвращать суммарные стоимости из базы данных. Вместо этого при отправке запроса в конечную точку приложение ведет себя странно. Код состояния HTTP 200 ОК, но вместо ожидаемого списка значений суммарных стоимостей возвращается значение `null`

4.2.2. Выявление проблем в удаленных средах

В этом подразделе мы используем удаленную отладку для анализа учебного варианта, описанного в подразделе 4.2.1. Начнем с конфигурирования и запуска приложения, чтобы установить соединение с удаленным отладчиком, а затем подключить отладчик и начать анализ.

При варианте из реальной практики приложение уже выполняется, но, вероятнее всего, пока еще не сконфигурировано для обеспечения удаленной отладки. Таким образом, мы начинаем с запуска приложения, чтобы представить вам полную картину удаленной отладки и информацию о предварительных условиях этой методики.

При запуске приложения, требующего удаленной отладки, необходимо убедиться в том, что агент отладчика подключен к выполнению. Чтобы подключить агента отладчика к выполнению Java-приложения, нужно добавить параметр `-agentlib:jdwp` в командную строку `java`, как показано на рис. 4.5. Вы обязательно должны указать номер порта, к которому подключается отладчик. Фактически агент отладчика действует как сервер, слушая отладчик для установления соединения с указанным при конфигурировании портом и обеспечения выполнения отладочных операций (приостановки выполнения в точке останова, шага с обходом, шага с входом и т. д.).

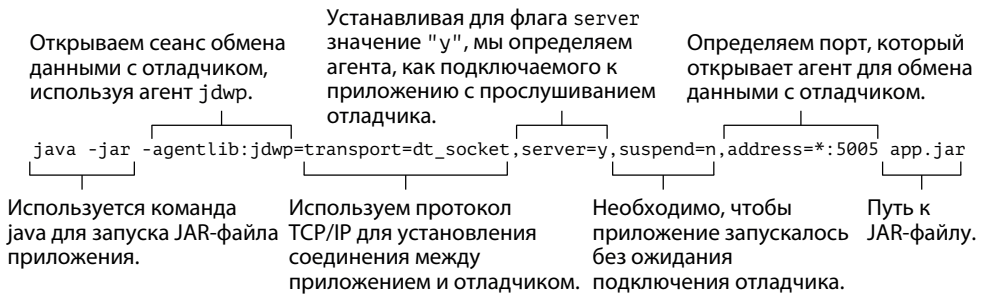


Рис. 4.5. При отладке приложения в локальном режиме IDE подключает отладчик. Но если приложение работает в удаленной среде, то вы должны сами подключить агента отладчика при запуске приложения

Вы можете скопировать (для выполнения) следующую команду:

```
java -jar -agentlib:jdwp=transport=dt_socket,  
➡ server=y,suspend=n,address=5005 app.jar
```

Обратите внимание на некоторые параметры конфигурации, определенные в этой команде:

- `transport=dt_socket` определяет конфигурацию способа, которым отладчик обменивается данными со своим агентом. Параметр конфи-

группации `dt_socket` означает, что используется протокол TCP/IP для обеспечения обмена данными через сеть. Этот способ всегда применяется для установления канала обмена данными между агентом и инструментом;

- `server=y` означает, что агент действует как сервер после подключения к выполнению приложения. Агент ожидает установления соединения с отладчиком и управляет выполнением приложения через это соединение. Вы должны использовать конфигурацию `server=n` для установления соединения с агентом отладчика без его запуска;
- `suspend=n` приказывает приложению начать выполнение без ожидания установления соединения с отладчиком. Если необходимо запретить запуск приложения до установления соединения с отладчиком, то следует использовать параметр `suspend=y`. В рассматриваемом здесь примере мы имеем дело с веб-приложением, и проблема возникает при вызове одной из его конечных точек, поэтому необходимо, чтобы приложение уже работало и могло вызвать проблемную конечную точку. Если бы мы анализировали проблему при начальной загрузке процесса сервера, то, вероятнее всего, потребовалось бы использование параметра `suspend=y`, чтобы разрешить запуск приложения только после установления соединения с отладчиком;
- `address=*:5005` приказывает агенту открыть порт 5005 в системе, т. е. порт, через который отладчик устанавливает соединение для обмена данными с агентом. Значение номера порта не должно уже использоваться в системе, а сетевая среда должна разрешить обмен данными между отладчиком и агентом (т. е. необходимо, чтобы порт был открыт в сетевой среде).

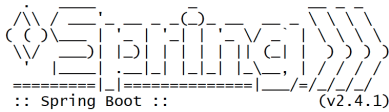
На рис. 4.6 показано приложение, запускаемое с подключенным агентом отладчика. Обратите внимание: сообщение, выведенное в консоли сразу после команды, информирует нас о том, что агент прослушивает сконфигурированный порт 5005.

После подключения агента отладчика к удаленному приложению можно установить соединение с отладчиком для начала анализа проблемы. Напомню, что сетевая среда конфигурируется так, чтобы обеспечить обмен данными между двумя приложениями (отладчик и его агент). В рассматриваемом здесь примере оба приложения запускаются на `localhost` (локальном компьютере), поэтому при демонстрации применения методики конфигурация сетевой среды не имеет особого значения.

Но в реальном сценарии вы всегда должны быть уверены в том, что обмен данными обеспечен до начала отладки. В большинстве случаев, вероятнее всего, потребуется участие специалиста из группы инфраструктуры, чтобы помочь вам открыть требуемый порт, если обмен данными не разрешен. Напомню, что обычно порты закрыты по умолчанию, и обмен данными через них запрещен из соображений безопасности.

Для запуска приложения можно использовать командную строку. При запуске приложения необходимо сделать его доступным для установления соединения с отладчиком через заданный порт.

```
$ java -jar -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=:5005 da-ch4-ex1-0.0.1-SNAPSHOT.jar
Listening for transport dt_socket at address: 5005
```



```
2021-08-21 08:59:12.122 INFO 83884 --- [main] com.example.Main : Starting Main v0.0.1-SNAPSHOT using Java 11.0.12 on EN1310832 with PID 83884 (C:\MANNINGS\debugging Java Applications\CODE\spilca3\code\da-ch4-ex1\target\da-ch4-ex1-0.0.1-SNAPSHOT.jar started by lspilca in C:\MANNINGS\Debugging Java Applications\CODE\spilca3\code\da-ch4-ex1\target)
```

Рис. 4.6. При выполнении команды для запуска приложения можно видеть, что приложение начинает выполнение. Одновременно можно заметить, что агент отладчика выводит строку, сообщающую, что он слушает отладчик для подключения через сконфигурированный порт 5005

Далее мы рассмотрим, как подключить отладчик к удаленному приложению, используя IntelliJ IDEA Community. Шаги запуска отладчика в приложении, работающем в удаленной среде, описаны ниже:

- 1) добавить новую конфигурацию выполнения;
- 2) сконфигурировать удаленный адрес (IP-адрес и номер порта) агента отладчика;
- 3) начать отладку приложения.

На рис. 4.7 показано, как открыть раздел **Edit Configurations** (Редактирование конфигураций), чтобы добавить новую конфигурацию выполнения.

На рис. 4.8 показано, как добавить новую конфигурацию выполнения.

Поскольку необходимо установить соединение с удаленным агентом отладчика, нужно добавить новую конфигурацию удаленной отладки, как показано на рис. 4.9.

Далее конфигурируется адрес агента отладчика, как показано на рис. 4.10. В рассматриваемом здесь примере приложение выполняется в той же системе, что и отладчик, поэтому мы используем localhost. В реальных условиях если приложение работает в другой системе, то потребуются указание IP-адреса этой системы. Мы используем порт 5005 для прослушивания агентом и установления соединения с отладчиком.

Чтобы добавить конфигурацию удаленной отладки в IntelliJ, сначала выберите в меню пункт **Edit Configurations**.

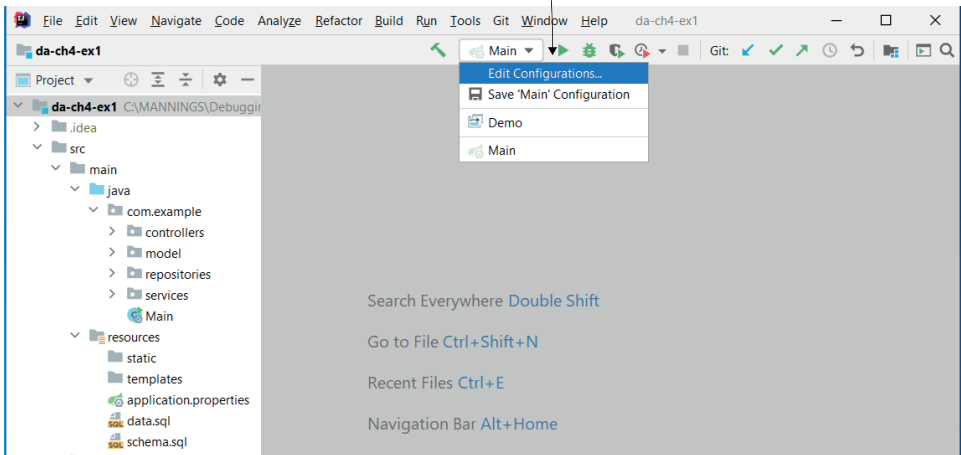


Рис. 4.7. Можно воспользоваться IDE, чтобы сконфигурировать отладчик для подключения к уже выполняющемуся приложению в конкретной удаленной среде при условии, что к этому приложению подключен агент отладчика. В IntelliJ IDEA Community необходимо создать новую конфигурацию выполнения, чтобы приказать отладчику подключиться к уже выполняющемуся приложению. Можно добавить новую конфигурацию выполнения, выбрав пункт меню **Edit Configurations**

После щелчка по пункту **Edit Configurations** IntelliJ открывает это окно. Щелкните по небольшому значку «плюс», затем выберите пункт **Add New Configuration** (Добавить новую конфигурацию).

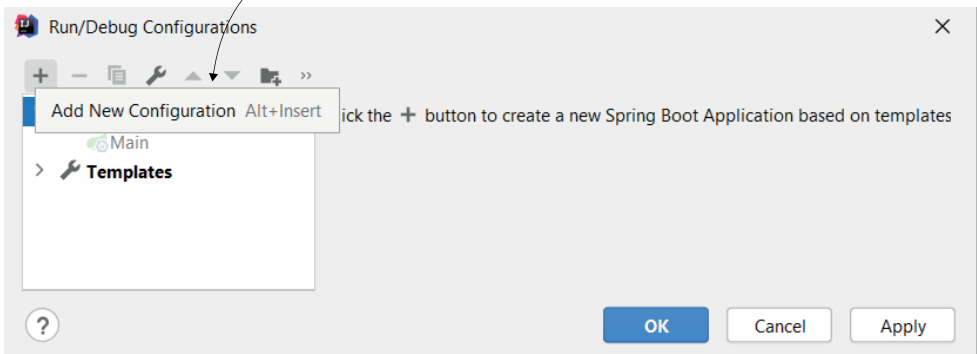


Рис. 4.8. После выбора пункта **Edit Configuration** можно добавить новую конфигурацию. Сначала щелкните по значку плюс, затем по пункту **Add New Configuration**

Из выведенного списка конфигураций выберите **Remote JVM Debug** (Удаленная отладка JVM).

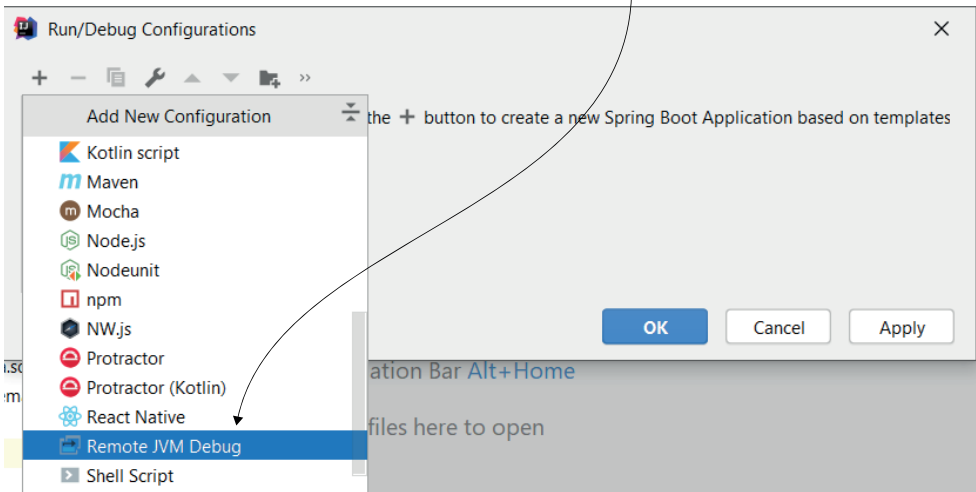


Рис. 4.9. Поскольку необходимо подключить отладчик к приложению, выполняющемуся в удаленной среде, выбираем тип конфигурации **Remote JVM Debug**

1. Выберите имя для новой конфигурации.

2. Укажите адрес системы, в которой выполняется приложение (в нашем примере localhost) и номер порта, сконфигурированный для подключения отладчика.

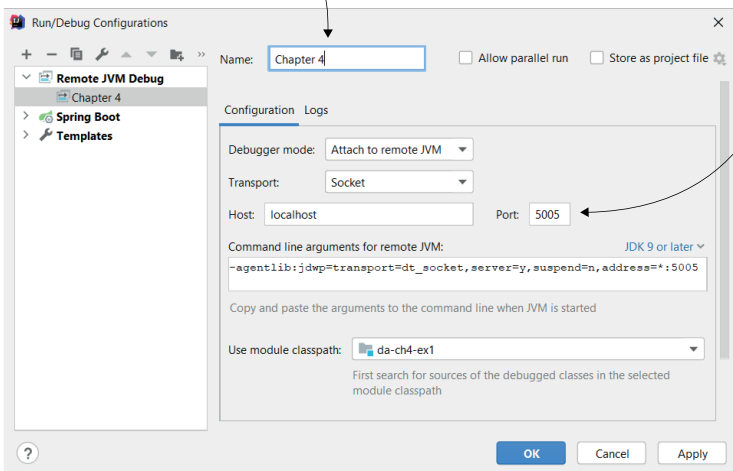


Рис. 4.10. Присвойте имя новой добавляемой конфигурации и укажите адрес удаленной среды и порт, сконфигурированный для прослушивания агентом отладчика (здесь: порт 5005 при запуске приложения)

Напомним, что мы устанавливаем соединение отладчика с его агентом, который открывает порт 5005 (см. рис. 4.11). Не следует путать порт, открытый агентом отладчика (5005), с портом, открытым веб-приложением (8080).

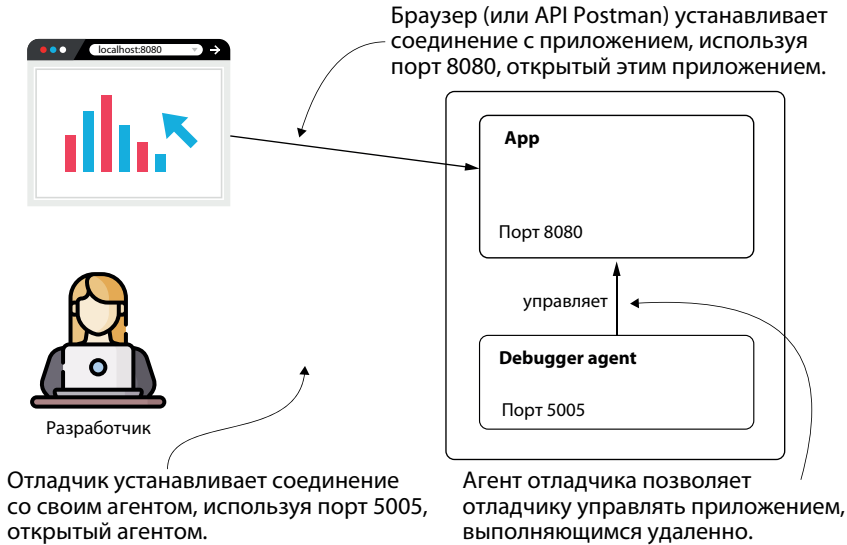


Рис. 4.11. Отладчик, работающий на компьютере разработчика, устанавливает соединение со своим агентом через порт 5005. Агент отладчика позволяет отладчику управлять приложением. Приложение также открывает порт, но этот порт предназначен для его клиентов (для веб-приложения это браузер)

После завершения процедуры конфигурирования запускаем отладчик (см. рис. 4.12). Он начинает «диалог» со своим агентом, подключенным к приложению, что позволяет вам управлять выполнением.

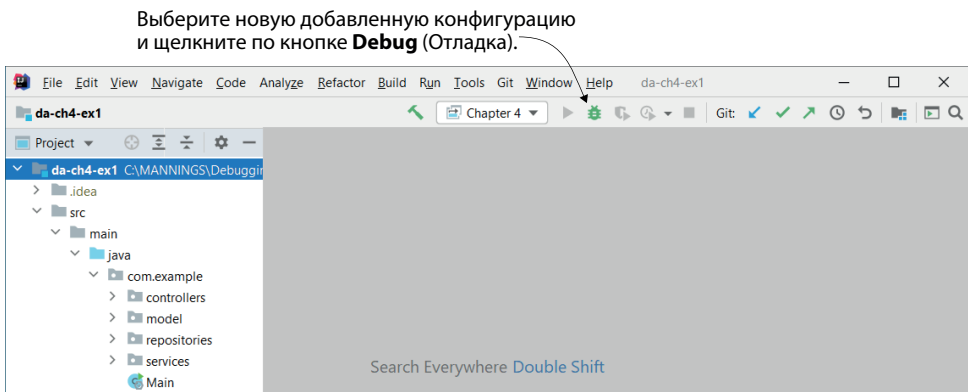


Рис. 4.12. Теперь можно запустить отладчик, используя новую добавленную конфигурацию. Щелкните по значку с небольшим жучком, чтобы запустить отладчик

Теперь можно использовать отладчик точно так же, как было описано в главах 2 и 3. Важно проследить за версией используемого кода (см. рис. 4.13). При локальной отладке приложения вам известно, что IDE компилирует приложение, а затем подключает отладчик к только что скомпилированному коду. Но если вы устанавливаете соединение с удаленным приложением, то уже не можете быть уверенными в том, что исходный код, имеющийся в вашем распоряжении, соответствует скомпилированному коду удаленного приложения, к которому вы подключили отладчик. Если группа разработки начала решать новые задачи, то код, требующий анализа, мог быть предположительно изменен, дополнен или удален в некоторых классах, входящих в состав приложения. Использование другой версии исходного кода может привести к странному и непонятному поведению отладчика. Например, отладчик может показывать, что вы перемещаетесь по пустым строкам или даже по строкам за пределами методов или классов. Трассировка стека выполнения также может стать несоответствующей ожидаемому выполнению.

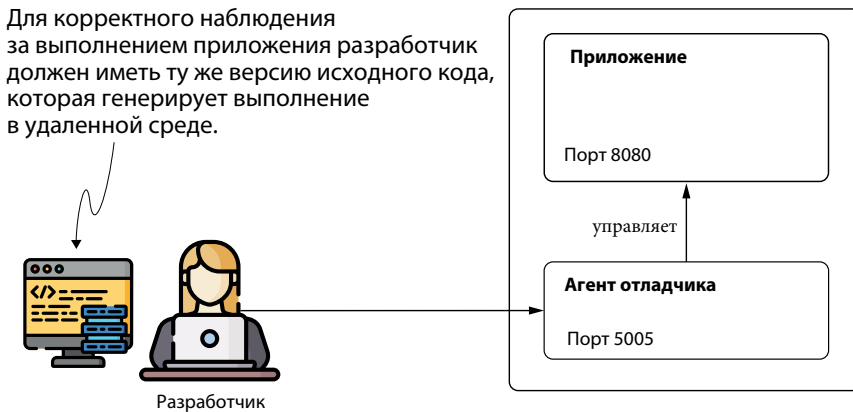


Рис. 4.13. Разработчик должен быть уверен в том, что располагает той же версией исходного кода, которая используется для генерации выполняемого кода приложения в удаленной среде. Иначе действия отладчика могут стать несоответствующими коду, анализируемому разработчиком, что больше запутывает, чем помогает разработчику понять поведение приложения

К счастью, в настоящее время мы используем программное обеспечение, позволяющее управлять версиями исходного кода, такое как Git или SVN, поэтому можем определить, какая версия исходного кода для создания приложения была развернута. Перед отладкой необходимо убедиться в том, что вы работаете с тем же исходным кодом, который был скомпилирован в приложение, требующее удаленной отладки. Воспользуйтесь своим инструментальным средством управления версиями, чтобы точно определить версию исходного кода.



Вот учебное упражнение для тебя!
Сделай небольшой перерыв в чтении
и попробуй решить конкретную проблему.

Что является причиной странного поведения
приложения, и как бы ты решил эту проблему?



Рис. 4.14.

Добавим точку останова в первой строке, вызывающей сомнения: в строке 23 в классе `ProductService`, как показано на рис. 4.15. Здесь приложение должно выбрать данные из базы данных для возврата в HTTP-ответе. В первую очередь необходимо определить, извлекаются ли данные корректно из базы, поэтому я останавливаю выполнение в этой строке и выполняю шаг с обходом, чтобы увидеть результат.

После запуска отладчика можно использовать его точно
так же, как и при анализе локального приложения.
Добавляйте точки останова и перемещайтесь по коду.



Рис. 4.15. Как и при отладке приложения в локальном режиме, можно добавлять точки останова и использовать операции перемещения по коду. Добавим новую точку останова в строке 23 в классе `ProductService`

После добавления точки останова используйте Postman (или аналогичное инструментальное средство) для отправки HTTP-запроса с неожиданным поведением (см. рис. 4.16). Postman (его можно скачать здесь: <https://www.postman.com/downloads/>) – это простой инструмент, который можно использовать для вызова конкретной конечной точки, а не так давно он стал одним из наиболее предпочитаемых разработчиками средств для этой цели. Postman предоставляет удобный графический пользовательский интерфейс (GUI), но если вы предпочитаете командную строку, то можно выбрать другой инструмент, например cURL. Чтобы упростить учебный пример, я использую Postman.

При использовании инструмента, например, Postman, запрос отправляется в конечную точку. Можно видеть, что запрос не завершился, он остается в состоянии ожидания, потому что отладчик приостановил выполнение приложения в строке, которую вы поместили точкой останова.

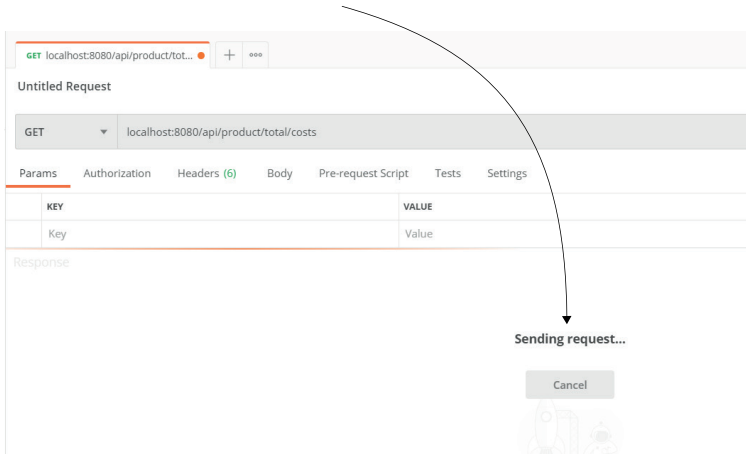


Рис. 4.16. При использовании Postman для отправки запроса ответ не приходит немедленно. Вместо этого Postman ожидает ответ без ограничения по времени, так как выполнение приложения приостановлено в строке, которую вы поместили точкой останова

Обратите внимание: Postman не показывает HTTP-ответ немедленно. Вместо этого вы видите, что запрос остается в состоянии ожидания, потому что отладчик приостановил выполнение приложения в строке, которую вы поместили точкой останова, как показано на рис. 4.17. Теперь можно начать применение операций перемещения по коду для анализа проблемы.

Применяя операцию шага с обходом, вы видите, что вместо возврата данных из базы приложение генерирует исключение (см. рис. 4.18). Теперь можно начать более подробное исследование проблемы:

- 1) разработчик, реализовавший эту функциональность, использовал простейший тип для представления столбца, который может содержать нулевые значения в базе данных. Поскольку примитивный тип в Java не является объектным типом и не может содержать значение «ноль», приложение генерирует исключение;
- 2) разработчик использовал метод `printStackTrace()` для вывода сообщения об исключении, которое бесполезно, потому что невозможно сконфигурировать вывод в различных рабочих средах без дополнительных трудозатрат. Вероятнее всего, это и является причиной того, что вы ничего не увидели в журналах при предварительном анализе (журналирование будет рассматриваться в главе 5);
- 3) проблема не проявлялась локально и в среде разработки `dev`, потому что в базе данных не было нулевых значений для этого поля (столбца).

Выполнение приостановлено в строке помеченной точкой останова. Теперь вы можете начать перемещение по коду для анализа проблемы.

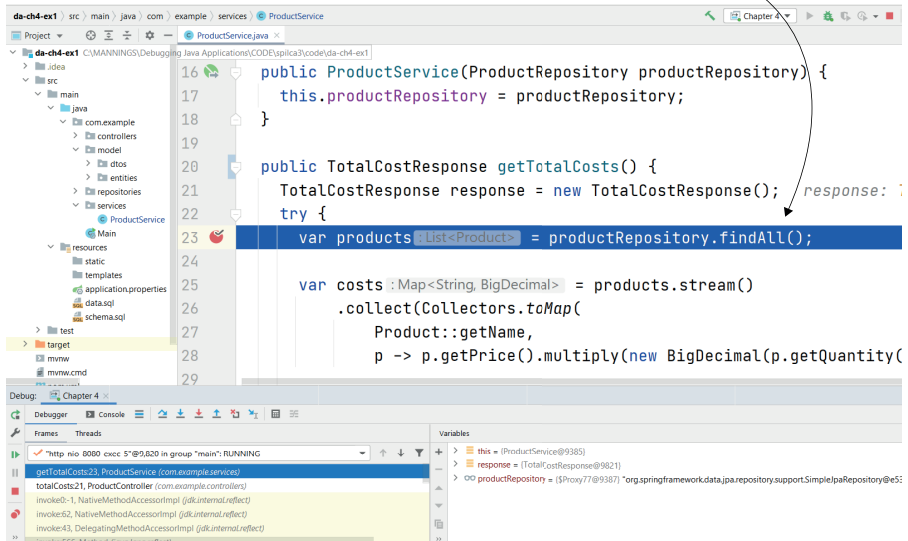


Рис. 4.17. IDE показывает, что отладчик действительно приостановил выполнение в строке, помеченной точкой останова. Таким образом, вы можете воспользоваться операциями перемещения по коду для продолжения анализа

Применяя операцию шага с обходом, можно увидеть, что код генерирует исключение. Теперь необходимо понять, почему конечная точка не возвращает ожидаемый результат.

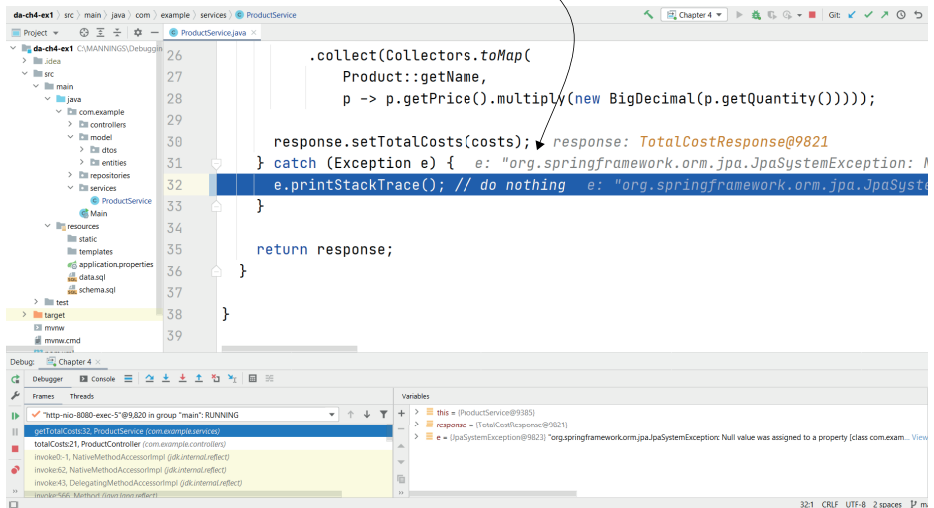


Рис. 4.18. Операция шага с обходом показывает, что приложение генерирует исключение. Теперь вам необходимо понять, в чем именно заключается проблема, и принять решение по ее устранению

Очевидно, что код требует рефакторинга, и, возможно, улучшение процесса инспекции кода должно стать темой обсуждения в группе во время очередного ретроспективного совещания. Но, как бы то ни было, вы довольны, что обнаружили причину возникновения проблемы и нашли решение по ее устранению.

Создание конфигурации удаленной отладки в IDE Eclipse

Я использую IntelliJ IDEA как основную IDE для всех примеров в книге. Но, как отмечалось в предыдущих главах, эта книга не об использовании конкретной IDE. Рассматриваемые методики можно применять с разнообразными инструментальными средствами по вашему выбору. Например, удаленную отладку можно выполнять в других IDE, например Eclipse.

На рис. 4.19 показано, как добавить новую конфигурацию отладки в IDE Eclipse.

Для добавления новой конфигурации отладки в IDE Eclipse в меню выберите **Run (Пуск) > Debug Configuration (Конфигурация отладки)**.

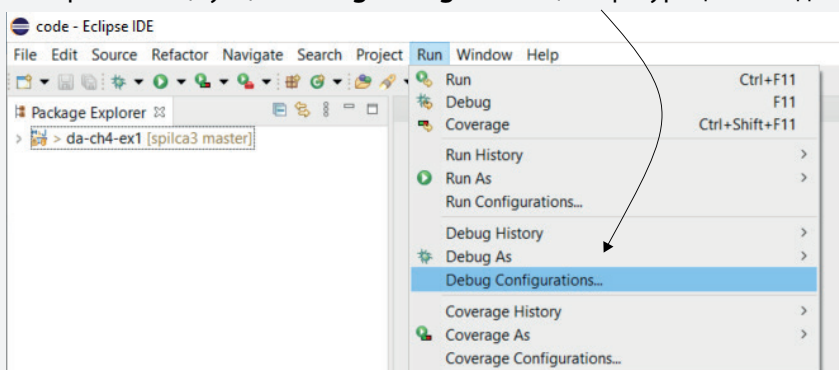
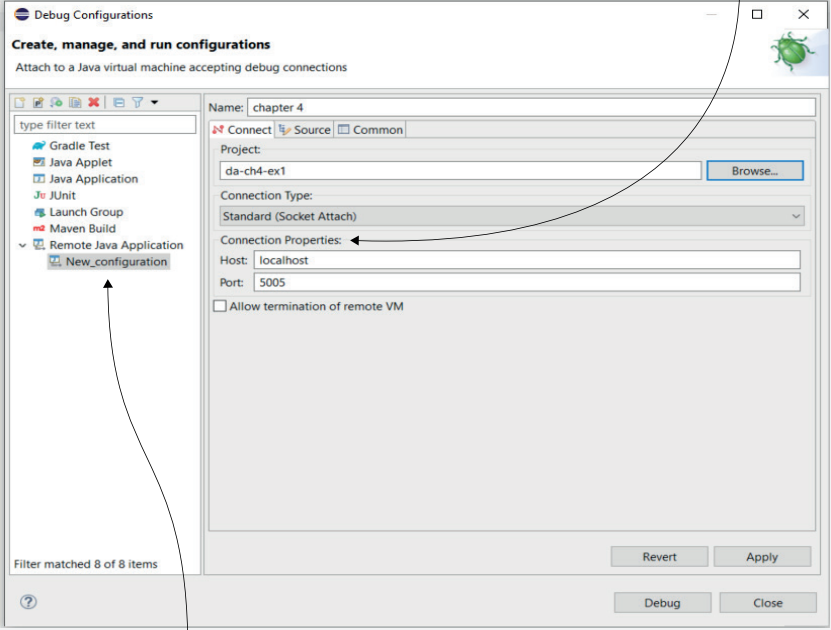


Рис. 4.19. Добавление новой конфигурации отладки в Eclipse

Чтобы добавить новую конфигурацию отладки в IDE Eclipse, в меню выберите **Run (Пуск) > Debug Configurations (Конфигурации отладки)**. Можно создать конфигурацию отладки для подключения к агенту отладчика, управляющего удаленным приложением.

Как и в IntelliJ IDEA, необходимо определить адрес агента отладчика (IP-адрес и номер порта), с которым отладчик устанавливает соединение.

Введите адрес агента отладчика (IP-адрес и номер порта) и сохраните конфигурацию.



В левой панели окна добавьте новую конфигурацию Remote Java Application.

Рис. 4.20. Добавление новой конфигурации Remote Java Application и установка адреса агента отладчика. Затем можно сохранить эту конфигурацию и использовать функцию отладки для установления соединения с удаленным приложением для его отладки

После добавления новой конфигурации запустите отладчик и добавьте точки останова для приостановки выполнения там, где необходимо начать анализ кода.

4.3. Резюме

- Иногда особенное неожиданное поведение работающего приложения возникает только в конкретных рабочих средах, в которых оно выполняется. Если это происходит, то отладка становится чрезвычайно трудной.
- Можно использовать отладчик в приложении Java, которое выполняется в удаленной рабочей среде, при следующих условиях:

- приложение должно быть запущено с подключенным агентом отладчика;
 - конфигурация сетевой среды должна обеспечивать обмен данными между отладчиком и агентом отладчика, подключенным к приложению в удаленной рабочей среде.
- Удаленная отладка позволяет применять те же методики, что и при локальной отладке, с подключением к процессу, выполняемому в удаленной рабочей среде.
- Перед отладкой приложения, работающего в удаленной рабочей среде, убедитесь в том, что отладчик использует копию того же исходного кода, который создал анализируемое приложение. Если у вас нет точно соответствующего исходного кода и изменения вносятся в части анализируемого приложения, то отладчик может вести себя странно, и удаленный анализ осложняется и даже становится практически бесполезным.

Глава 5



Максимальное использование журналов: инспектирование поведения приложения

Темы:

- эффективное использование журналов для понимания поведения приложения;
- правильная реализация функций журналирования в приложении;
- устранение проблем, создаваемых журналами.

В этой главе мы рассмотрим использование сообщений, которые приложение записывает в журнал. Идея ведения журналов впервые появилась не в сфере разработки программного обеспечения. На протяжении многих веков люди использовали журналы, которые помогали им понять события и процессы прошлого. Журналы использовались с момента изобретения письменности и продолжают использоваться по сей день. На всех кораблях есть судовые журналы. Моряки записывают принятые решения (курс, увеличение или уменьшение скорости и т. п.) и отданные или полученные приказы, а также любые события (см. рис. 5.1). Если что-то происходит с корабельным оборудованием, то записи в журнале могут помочь понять, где находится корабль и как дойти до ближайшего побережья. В аварийных случаях (крушение и т. п.) записи в журнале можно использовать при расследовании, чтобы определить, как можно было избежать несчастного случая.



Рис. 5.1. Моряки записывают события в журналы, которые можно использовать для определения курса или анализа ответной реакции экипажа на конкретное событие. Точно так же приложения сохраняют сообщения в журналах, чтобы в дальнейшем разработчики могли проанализировать потенциальную проблему или обнаружить слабые места в приложении

Если вы когда-нибудь наблюдали за шахматной партией, то знаете, что оба игрока записывают каждый ход. Зачем? Эти записи помогают им восстановить всю партию по прошествии некоторого времени. Шахматисты изучают свои ходы и ходы соперника, чтобы обнаружить потенциальные ошибки и слабости.

По тем же причинам приложения также записывают сообщения в журнал. Эти сообщения можно использовать, чтобы понять, что происходит при выполнении приложения. Читая сообщения в журнале, вы можете восстановить ход выполнения точно так же, как шахматист воспроизводит всю партию в целом. Журналами можно воспользоваться при анализе непонятного или нежелательного поведения или проблем, которые очень трудно обнаружить, например уязвимостей в системе обеспечения безопасности.

Я уверен, что вы уже знаете, как выглядят журналы. Вы видели сообщения в журнале, по крайней мере, когда выполняли приложение в IDE (см. рис. 5.2). Во всех IDE имеется консоль журнала (log console). Это одно из средств, которые все разработчики программного обеспечения осваивают в первую очередь. Но приложение выводит журнальные сообщения не только в консоли IDE. Реальные промышленные приложения сохраняют сообщения в журнале, чтобы позволить разработчикам проанализировать конкретное поведение приложения в определенный момент времени.

При выполнении приложения в локальной системе с использованием DE вы видите журнальные сообщения в консоли.

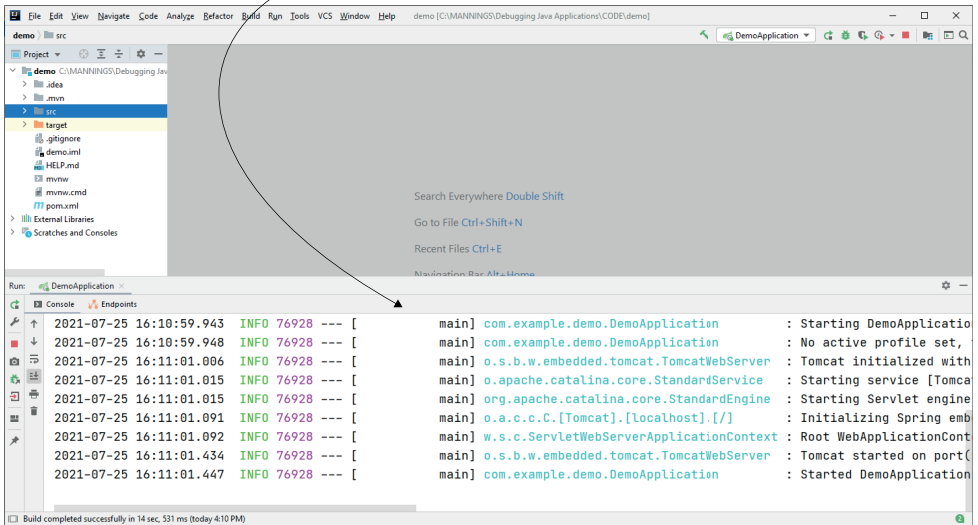


Рис. 5.2. Консоль журнала IDE. Все IDE имеют консоль журнала. Сообщения, выводимые в этой консоли, полезны при выполнении приложения в локальном режиме, но реальные промышленные приложения также сохраняют сообщения в журнале – это необходимо для того, чтобы понять, как ведет себя приложение в определенный момент времени

На рис. 5.3 показана структура журнального сообщения в стандартном формате. Журнальное сообщение – это просто строка, поэтому теоретически она может быть любым предложением. Но понятные и удобные для использования журналы должны соблюдать некоторые выработанные практические правила (о которых вы узнаете в этой главе). Например, в дополнение к описанию журнальное сообщение содержит метку времени, определяющую, когда приложение записало сообщение, характеристику степени важности и указание той части приложения, которая записала сообщение (см. рис. 5.3).

Во многих случаях журналы обеспечивают эффективный способ анализа поведения приложения. Несколько примеров:

- анализ события или хронологической последовательности событий, которые уже произошли;
- анализ проблем, возникающих в случаях, когда взаимодействие с приложением изменяет его поведение (Гейзенбаги);
- изучение поведения приложения в течение длительного интервала времени;
- выявление предупреждений о весьма важных событиях, требующих немедленного внимания.

Метка времени. Когда приложение записало это сообщение? Метка времени показывает, когда сообщение было записано в журнал, и это весьма важная подробность, позволяющая хронологически упорядочить сообщения. Поэтому метка времени должна всегда располагаться в самом начале сообщения.

Степень важности. Насколько важно это сообщение? Это чрезвычайно важное сообщение, требующее немедленного особого внимания, или сообщение о подробностях события при выполнении.

2021-07-25 16:11:01.434 INFO [s.b.w.embedded.tomcat.TomcatWebServer] :
Tomcat started on port(s): 8080 (http) with context path "

Сообщение. Что произошло? Удобное для чтения и понимания (человеком) описание события.

Локация. Где именно приложение обнаружило это событие? Обычно журнальное сообщение показывает как минимум модуль и класс, который записал это сообщение в журнал.

Рис. 5.3. Структура правильно отформатированного журнального сообщения.

В дополнение к описанию ситуации или события сообщение также должно содержать некоторые другие важные подробности: метку времени записи приложением сообщения в журнал, степень важности события и локацию, из которой было записано сообщение. Использование этих подробностей в журнальных записях облегчает анализ проблемы

Вообще говоря, мы не ограничиваемся только одной методикой при анализе поведения функций и частей конкретного приложения. В зависимости от сценария разработчик может объединить несколько методик, чтобы понять поведение в данном конкретном случае. В некоторых вариантах будет использоваться отладчик и журналы, а также другие методики (о которых вы узнаете в следующих главах), чтобы узнать, почему что-то работает (или не работает) как должно.

Я всегда рекомендую разработчикам проверять журналы перед выполнением любых других действий по анализу проблемы (см. рис. 5.4). Журналы часто позволяют сразу же обнаружить непонятное поведение, и это помогает точно определить место начала анализа. Журналы не всегда отвечают на все вопросы, но определение начального пункта чрезвычайно важно. Если журнальные сообщения указывают, с чего начать, то вы уже сэкономили огромное количество времени.

По моему мнению, журналы не просто чрезвычайно полезны, в действительности они незаменимы для любого приложения. В следующем разделе мы рассмотрим, как использовать журналы, и изучим типовые сценарии анализа, в которых журналы весьма важны. В разделе 5.2 вы узнаете, как правильно реализовать функции журналирования в приложении. Мы рассмотрим использование уровней журналирования, которые помогают более удобно фильтровать события и проблемы, связанные с журналами. В разделе 5.3 мы обсудим различия между использованием журналов и удаленной отладкой.

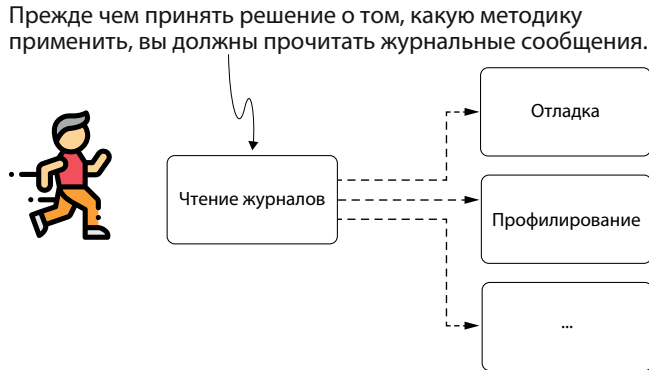


Рис. 5.4. При анализе проблемы первое, что вы должны делать всегда, – прочитать журналы приложения. Во многих случаях журнальные сообщения указывают начальный пункт или дают полезные подсказки о том, что нужно делать дальше для устранения проблемы

Кроме того, я рекомендую прочитать часть IV книги «Logging in Action» Фила Уилкинса (Phil Wilkins) (Manning, 2022 г.). В текущей главе больше внимания уделено методикам анализа с использованием журналов, тогда как в «Logging in Action» более глубоко рассматриваются технические подробности самих журналов, к тому же демонстрируется функция журналирования при использовании не только Java, но и другого языка (Python).

5.1. Анализ проблем с использованием журналов

Как и любая другая методика анализа, использование журналов имеет смысл в одних ситуациях и неприменимо в других. В этом разделе мы изучим различные сценарии, в которых использование журналов может обеспечить более легкое понимание поведения программного обеспечения. Начнем с обсуждения нескольких основных характеристик журнальных сообщений, затем подробно рассмотрим, как эти характеристики помогают разработчикам при анализе проблем, возникающих в приложениях.

Одним из самых главных преимуществ журнальных сообщений является то, что они позволяют визуальное представление выполнения конкретной части кода в определенном интервале времени. При использовании отладчика, как было описано в главах 2–4, ваше внимание в основном сосредоточено на текущем моменте времени. Вы наблюдаете за тем, как выглядят данные, когда отладчик приостанавливает выполнение в заданной строке кода. Отладчик не сообщает многие подробности о предыстории выполнения. Можно воспользоваться трассировкой стека выполнения для определения пути выполнения, но все остальное сосредоточено на текущем моменте.

В противоположность отладке при использовании журналов внимание сосредоточено на выполнении приложения в прошедшем интервале времени (см. рис. 5.5). Для журнальных сообщений характерна строгая связь со временем.

Отладка



При отладке вы сосредоточены на текущем состоянии выполнения приложения.

Прошлое

Будущее

Анализ с использованием журналов



При анализе с использованием журналов вы сосредоточены на интервале времени в прошлом.

Прошлое

Будущее

Рис. 5.5. При анализе проблемы с помощью отладчика вы сосредоточены на текущем моменте. При использовании журнальных сообщений все внимание уделяется конкретному интервалу времени в прошлом. Это различие может помочь вам принять решение о том, какую методику применить

Помните о необходимости учета часового пояса, установленного в системе, в которой выполняется приложение. Время фиксации в журнале может быть сдвинуто на несколько часов из-за различных временных зон (например, между зоной, где выполняется приложение, и зоной, в которой находится разработчик), и это может стать источником путаницы.



ПРИМЕЧАНИЕ. Всегда включайте метку времени в журнальное сообщение. Метка времени будет использоваться для упрощения определения порядка, в котором сообщения были записаны в журнал. Это дает вам точную информацию о том, когда приложение записало конкретное сообщение. Рекомендуется размещать метку времени в первой части (в самом начале) сообщения.

5.1.1. Использование журналов для идентификации исключений

Журналы помогают идентифицировать проблему после ее возникновения и проанализировать ее главную причину. Мы часто используем журналы, чтобы решить, где начать анализ. Затем продолжаем исследование проблемы, применяя другие инструменты и методики, такие как отладчик

(описанный в главах 2–4) или профилировщик (который будет рассматриваться в главах 6–9). Часто в журналах можно найти трассировки стека исключений. Приведенный ниже фрагмент демонстрирует пример трассировки стека исключений Java:

```
java.lang.NullPointerException
at java.base/java.util.concurrent.ThreadPoolExecutor
➡ runWorker(ThreadPoolExecutor.java:1128) ~[na:na]
at java.base/java.util.concurrent.ThreadPoolExecutor$Worker
➡ run(ThreadPoolExecutor.java:628) ~[na:na]
at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable
➡ run(TaskThread.java:61) ~[tomcat-embed-core-9.0.26.jar:9.0.26]
at java.base/java.lang.Thread.run(Thread.java:830) ~[na:na]
```

При изучении этой трассировки стека исключений или какого-либо аналогичного в журнале приложения вы получаете информацию о том, в чем предположительно заключается проблема с конкретной функцией. У каждого исключения есть свой особый смысл, который помогает определить, где именно в приложении возникла проблема. Например, `NullPointerException` сообщает, что каким-то образом инструкция обратилась к атрибуту или методу через переменную, которая не содержала ссылку на экземпляр объекта (см. рис. 5.6).

Если приложение генерирует `NullPointerException` в указанной строке, это означает, что выполнено обращение к переменной, не содержащей ссылку на объект. Другими словами, результатом вычисления такой переменной является ноль (`null`).

```
var invoice = getLastIssuedInvoice();

if (client.isOverdue()) {
    invoice.pay(); ←
}
}
```

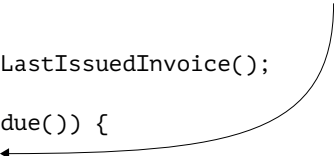
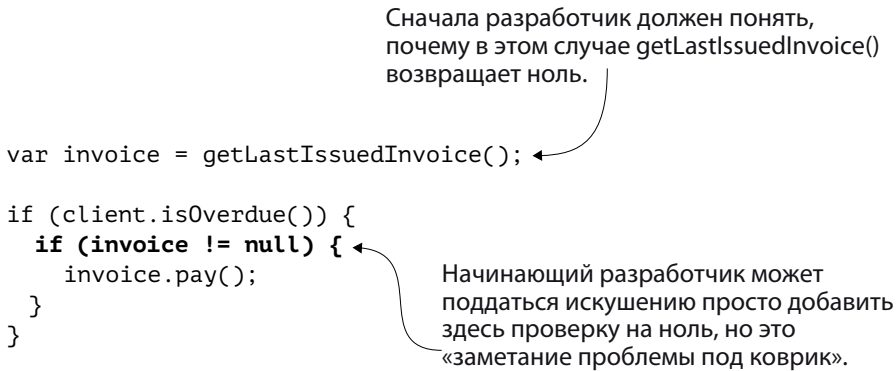


Рис. 5.6. Исключение `NullPointerException` указывает, что при выполнении приложения возникло поведение, которое было вызвано без действующего экземпляра. Но это не означает, что строка, в которой сгенерировано исключение, также является источником проблемы. Исключение могло оказаться последствием главной причины. Вы всегда должны искать главную причину, а не пытаться локально устранить проблему

Я часто замечаю, что этот принцип смущает начинающих программистов. Рассмотрим простой пример `NullPointerException`, вероятно, являющегося самым первым исключением, с которым встречается любой разработчик на Java, и одним из тех, которые проще всего понять. Но если вы обнаруживаете `NullPointerException` в журналах, то должны в первую очередь спросить себя: почему отсутствует эта ссылка? Возможно, потому, что не-

которая инструкция, выполненная приложением ранее, не работала так, как ожидалось (см. рис. 5.7).



```
var invoice = getLastIssuedInvoice();

if (client.isOverdue()) {
    if (invoice != null) {
        invoice.pay();
    }
}
```

Сначала разработчик должен понять, почему в этом случае `getLastIssuedInvoice()` возвращает ноль.

Начинающий разработчик может поддаться искушению просто добавить здесь проверку на ноль, но это «замечание проблемы под ковром».

Рис. 5.7. Попытка локального устранения проблемы во многих случаях равнозначна замечанию мусора под ковер. Если продолжает существовать главная причина, то в дальнейшем может возникнуть еще больше проблем. Помните, что исключение в журналах не всегда показывает главную причину

ПРИМЕЧАНИЕ. Всегда следует помнить о том, что локация, в которой возникло исключение, не обязательно является местонахождением главной причины проблемы. Исключение сообщает, где что-то пошло не так, но само по себе исключение может оказаться последствием проблемы, возникшей в другом месте. Исключение не всегда является самой проблемой. Не следует принимать слишком быстрое решение о локальном устранении исключения добавлением блока `try-catch-finally` или инструкции `if-else`. Сначала убедитесь в том, что вы понимаете главную причину возникновения проблемы, прежде чем искать решение по ее устранению.

5.1.2. Использование трассировок стека исключений для определения стороны, вызывающей метод

Одной из методик, которую разработчики считают бесполезной, но, на мой взгляд, обеспечивающей определенные преимущества на практике, является журналирование трассировки стека исключений для определения стороны, вызывающей конкретный метод. С самого начала моей карьеры разработчика программного обеспечения я работал со сложными кодовыми базами (как правило) крупных приложений. Одним из часто возникающих затруднений становилось выяснение того, кто вызывает конкретный метод, когда приложение выполняется в удаленной рабочей среде. Если вы просто читаете код приложения, то обнаружите сотни способов, которыми может быть вызван интересующий вас метод.

Разумеется, если вы счастливчик и обладаете необходимыми правами доступа, то можете использовать удаленную отладку, описанную в главе 4. Далее можно получить доступ к трассировке стека выполнения, предоставляемой отладчиком. Но что, если невозможно использование отладчика в удаленном режиме? В этом случае можно воспользоваться методикой журналирования.

Исключения в Java обладают свойством, на которое часто не обращают никакого внимания: они отслеживаются в трассировке стека исключений. При обсуждении исключений мы часто называем трассировку стека выполнения трассировкой стека исключений. Но в конечном счете это одно и то же. Трассировка стека исключений показывает цепочку вызовов методов, которые стали причиной конкретного исключения, и вы получаете доступ к этой информации даже без генерации этого исключения. В коде достаточно использовать исключение:

```
new Exception().printStackTrace();
```

Рассмотрим метод в листинге 5.1. Если у вас нет отладчика, то можно просто вывести трассировку стека исключений, как показано в этом примере в первой строке кода метода. Всегда следует помнить о том, что эта инструкция только выводит трассировку стека, но не генерирует исключение, поэтому не воздействует на выполняемую логику. Это пример из проекта da-ch5-ex1.

Листинг 5.1. Вывод трассировки стека исключений в журнал с использованием исключения

```
public List<Integer> extractDigits() {
    new Exception().printStackTrace();
    List<Integer> list = new ArrayList<>();
    for (int i = 0; i < input.length(); i++) {
        if (input.charAt(i) >= '0' && input.charAt(i) <= '9') {
            list.add(Integer.parseInt(String.valueOf(input.charAt(i))));
        }
    }

    return list;
}
```

❶ Выводит трассировку стека исключений.

Приведенный ниже фрагмент показывает, как приложение выводит трассировку стека исключений в консоли. В сценарии из реальной практики трассировка стека помогает сразу же определить поток выполнения, который приводит к анализируемому вызову, как было описано в главах 2 и 3. В рассматриваемом здесь примере по записям журнала можно

видеть, что метод `extractDigits()` был вызван в строке 11 класса `Decoder` из метода `decode()`:

```
java.lang.Exception at main.StringDigitExtractor
↳ extractDigits(StringDigitExtractor.java:15)
  at main.Decoder.decode(Decoder.java:11)
  at main.Main.main(Main.java:9)
```

5.1.3. Измерение времени, затраченного на выполнение конкретной инструкции

Журнальные сообщения представляют собой простой способ измерения времени, которое требуется для выполнения конкретного набора инструкций. Всегда можно зафиксировать в журнале разность между метками времени до и после конкретной строки кода. Предположим, что вы анализируете проблему производительности, при которой определенный фрагмент функциональности выполняется слишком долго. Вы предполагаете, что причиной является запрос, выполняемый приложением для извлечения данных из базы данных. При некоторых значениях параметра запрос работает медленно, снижая общую производительность приложения.

Чтобы найти параметр, создающий проблему, можно записывать запрос и время его выполнения в журнал. После определения проблемных значений параметра можно начинать поиск решения. Возможно, потребуется добавление еще одного индекса в таблицу базы данных, или необходимо переписать запрос, чтобы он работал быстрее.

В листинге 5.2 показано, как зафиксировать в журнале время, затраченное на выполнение конкретного фрагмента кода. Например, определим, сколько времени требуется приложению для выполнения операции поиска всех товаров в базе данных. Да, я знаю, что здесь нет параметров, но я упростил пример, чтобы позволить вам сосредоточиться на обсуждаемом синтаксисе. Но в реальном производственном приложении вам, вероятнее всего, придется анализировать более сложные операции.

Листинг 5.2. Запись в журнал времени выполнения конкретной строки кода

```
public TotalCostResponse getTotalCosts() {
    TotalCostResponse response = new TotalCostResponse();

    long timeBefore = System.currentTimeMillis();           ❶
    var products = productRepository.findAll();             ❷
    long spentTimeInMillis =                                ❸
        System.currentTimeMillis() - timeBefore;

    log.info("Execution time: " + spentTimeInMillis);       ❹

    var costs = products.stream().collect(
```

```

        Collectors.toMap(
            Product::getName,
            p -> p.getPrice()
                .multiply(new BigDecimal(p.getQuantity()))));

    response.setTotalCosts(costs);

    return response;
}

```

- ❶ Запись в журнал метки времени перед выполнением метода.
- ❷ Выполнение метода, для которого необходимо вычислить время выполнения.
- ❸ Вычисление затраченного времени между метками времени после и до выполнения.
- ❹ Вывод времени выполнения.

Точное измерение времени, затраченного приложением на выполнение конкретной инструкции, – простая, но эффективная методика. Но я бы использовал эту методику только временно, при анализе проблемы. Я не рекомендую сохранять такие средства журналирования в коде на длительное время, поскольку, вероятнее всего, они не потребуются в дальнейшем и делают код более трудным для чтения. После устранения проблемы, когда уже не нужно знать время выполнения конкретной строки кода, такие средства журналирования можно удалить.

5.1.4. Анализ проблем в многопоточных архитектурах

Многопоточная архитектура представляет собой тип производственного потенциала (ресурса), который использует несколько потоков для определения своей функциональности и часто является чувствительным к внешним воздействиям (см. рис. 5.8). Например, если вы используете отладчик или профилировщик (инструменты, которые воздействуют на выполнение приложения), то поведение приложения может измениться (см. рис. 5.9).

Но если вы используете журналирование, то существенно снижаете вероятность воздействия на выполнение приложения. Средства журналирования также могут иногда воздействовать на многопоточные приложения, но их воздействие недостаточно велико, чтобы изменить поток выполнения приложения. Таким образом, журналирование может стать решением по извлечению данных, необходимых для анализа.

Поскольку журнальные сообщения содержат метку времени (как было отмечено выше), можно упорядочить их, чтобы определить последовательность выполнения операций. В Java-приложении иногда полезно фиксировать в журнале имя потока, выполняющего конкретную инструкцию. Можно получить имя текущего выполняемого потока, используя следующую инструкцию:

```
String threadName = Thread.currentThread().getName();
```

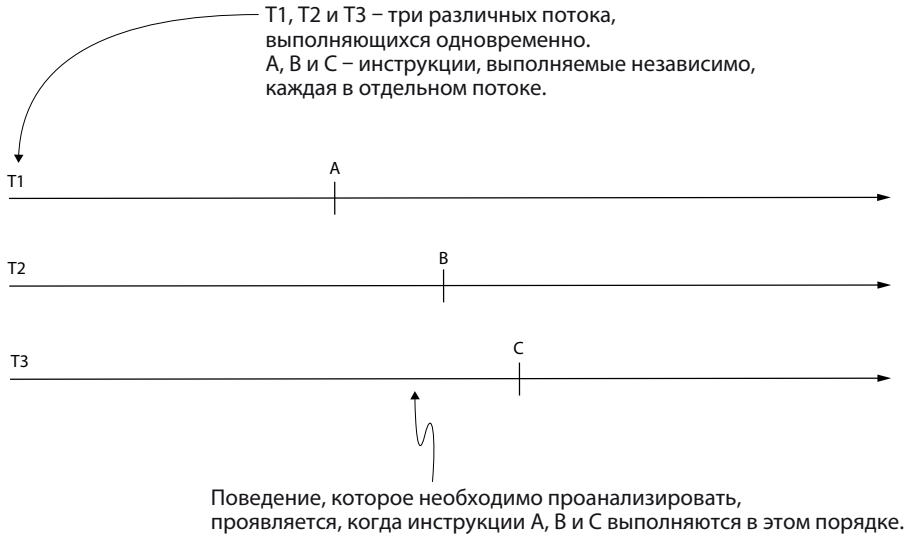


Рис. 5.8. Многопоточная архитектура. Приложение с возможностью использования нескольких потоков, одновременно выполняющихся для обработки данных, является многопоточным приложением. Без явной синхронизации инструкции, работающие в независимых потоках (A, B и C), могут выполняться в любом порядке

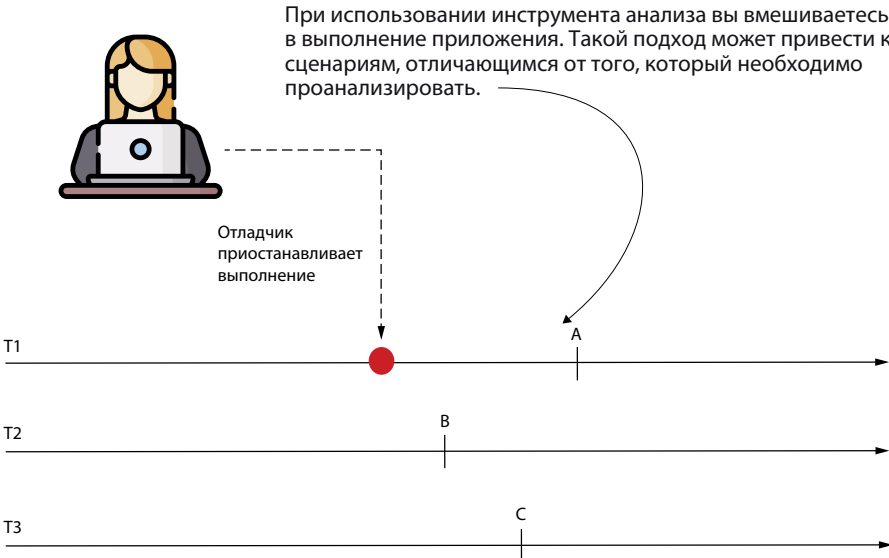


Рис. 5.9. Использование инструментального средства, например отладчика или профилировщика, воздействует на выполнение, замедляя некоторые (или все) потоки. Из-за этого выполнение часто изменяется, и некоторые инструкции могут выполняться в порядке, отличающемся от сценария, который необходимо проанализировать. В этом случае инструмент становится бесполезным, поскольку вы лишаетесь возможности исследования интересующего вас поведения

В Java-приложениях все потоки имеют имена. Их может именовать разработчик, или JVM идентифицирует потоки, используя имя по шаблону Thread-х, где х – инкрементируемое число. Например, первый созданный поток получит имя Thread-0, следующий – Thread-1 и т. д. В главе 10 вы узнаете о том, что при использовании дампов потоков именовании потоков приложения является правильным практическим подходом, который позволяет легко идентифицировать потоки при анализе.

5.2. Реализация функций журналирования

В этом разделе рассматриваются наилучшие практические методики реализации функций журналирования в приложениях. Чтобы правильно подготовить журнальные сообщения приложения для анализа и избежать проблем при выполнении приложения, необходимо обратить особое внимание на некоторые подробности реализации.

Начнем с обсуждения в подразделе 5.2.1 того, как приложения постоянно хранят журналы, особо выделяя преимущества и недостатки этих практических подходов. В подразделе 5.2.2 вы узнаете, как более эффективно использовать журнальные сообщения, классифицируя их на основе степени важности, следовательно, обеспечивая улучшение производительности приложения. В подразделе 5.2.3 рассматриваются проблемы, которые могут возникать в журнальных сообщениях, и способы их устранения.

5.2.1. Постоянно хранимые журналы

Постоянное хранение – одна из самых важных характеристик журнальных сообщений. Как уже отмечалось выше, журналирование отличается от прочих методик анализа, поскольку сосредоточено в большей степени на прошлом, нежели на настоящем. Мы читаем журнал, чтобы понять, что произошло, поэтому приложение должно сохранять записи журнала, обеспечивая возможность их чтения в будущем. Способ сохранения журнальных сообщений может влиять на удобство использования журнала и на производительность приложения. Я работал со многими приложениями и имел возможность наблюдать разнообразные способы, которыми разработчики реализуют постоянное хранение журнальных сообщений:

- сохранение журнальных записей в реляционной базе данных;
- сохранение журнальных записей в файлах;
- сохранение журнальных записей в реляционной базе данных.

Все эти способы являются подходящими вариантами в зависимости от того, что именно делает приложение. Рассмотрим некоторые из основных аспектов, которые необходимо учитывать, чтобы принять правильное решение.

Сохранение журнальных записей в нереляционной базе данных

Нереляционные (NoSQL) базы данных помогают сохранять баланс между производительностью и согласованностью. Базу данных NoSQL можно использовать для хранения журнала с поддержкой более высокой производительности, но создается вероятность того, что база данных потеряет некоторые журнальные сообщения или не сохранит сообщения в точном хронологическом порядке, в котором их записало приложение. Но, как было отмечено выше, журнальное сообщение всегда должно содержать метку времени сохранения сообщения, преимущественно в самом его начале.

Сохранение журнальных сообщений в базах данных NoSQL является общепринятой практикой. В большинстве случаев приложения используют полноценный механизм, сохраняющий журнал и предоставляющий возможности для извлечения, поиска и анализа журнальных сообщений. В настоящее время чаще всего используются такие механизмы, как стек ELK (<https://www.elastic.co/what-is/elk-stack>) и Splunk (<https://www.splunk.com/>).

Сохранение журнальных записей в файлах

В прошлом приложения сохраняли журналы в файлах. Возможно, вы и сейчас найдете более старые приложения, которые записывают журнальные сообщения напрямую в файлы, но этот подход в настоящее время менее распространен, потому что в общем случае он более медленный, и поиск данных в журнале более сложен. Я обратил ваше внимание на этот способ, потому что вы обнаружите многочисленные руководства и примеры вариантов сохранения журналов приложений в файлах, но в более современных приложениях следует избегать применения этого способа.

Сохранение журнальных записей в реляционной базе данных

Мы редко используем реляционные базы данных для хранения журнальных сообщений. Реляционная база данных главным образом обеспечивает согласованность и целостность данных, гарантируя, что журнальные сообщения не теряются. После сохранения сообщений вы можете извлекать их. Но согласованность и целостность обеспечивается за счет (снижения) производительности.

В большинстве приложений потеря одного журнального сообщения не так критична, а производительность, как правило, важнее целостности и согласованности. Но, как всегда, в реальных приложениях существуют исключения. Например, правительства во всем мире вводят правила регистрации журнальных сообщений для финансовых приложений, особенно для платежных функциональных средств. Такие средства обычно должны фиксировать определенные журнальные сообщения, потеря которых приложением недопустима. Несоблюдение этих правил может привести к санкциям и штрафам.

5.2.2. Определение уровней журналирования и использование рабочих сред для ведения журналов

В этом подразделе мы рассмотрим уровни журналирования и правильную реализацию функций журналирования в приложении с использованием соответствующих рабочих сред (фреймворков). Начнем с объяснения, почему уровни журналирования чрезвычайно важны, а затем рассмотрим их реализацию на примере.

Уровни журналирования, также называемые степенями важности (*severities*), – это способ классификации журнальных сообщений на основе их важности для анализа. При выполнении приложение обычно генерирует большое количество журнальных сообщений. Но часто вам не нужны подробности, содержащиеся в о всех журнальных сообщениях. Некоторые из них более важны для анализа, чем прочие: сообщения, представляющие критические события, всегда требуют особого внимания.

Наиболее часто используемые уровни журналирования (степени важности) описаны ниже:

- *error* (ошибка) – весьма важная (критическая) проблема. Приложение всегда должно фиксировать такие события. Обычно необработанные исключения в Java-приложениях регистрируются в журнале как ошибки;
- *warn* (предупреждение) – событие, которое предположительно является ошибкой, но приложение обрабатывает ее. Например, если соединение с внешней сторонней системой не было установлено с первого раза, но приложению удалось отправить вызов со второй попытки, то проблема должна быть зафиксирована в журнале как предупреждение;
- *info* (информация) – журнальные сообщения «общего характера». Такие сообщения представляют основные события при выполнении приложения, которые помогают понять поведение приложения в большинстве ситуаций;
- *debug* (отладка) – уточненные подробности, фиксацию которых следует разрешать только в тех случаях, когда информационных сообщений недостаточно.

Классификация журнальных сообщений на основе степени важности позволяет минимизировать количество сообщений, которые приложение сохраняет в журнале. Вы должны разрешать приложению фиксировать в журнале только наиболее важные детали и расширять уровень журналирования, только если необходимы дополнительные подробности.

Рассмотрим рис. 5.10, где представлена пирамида степеней важности журнальных сообщений:

- приложение регистрирует небольшое количество самых важных проблем, но все они имеют высокую степень важности, поэтому всегда должны записываться в журнал;
- чем ближе вы продвигаетесь к основанию пирамиды, тем больше журнальных сообщений записывает приложение, но они становятся менее важными и гораздо реже требуются при анализе.



ПРИМЕЧАНИЕ. Следует отметить, что различные библиотеки могут использовать больше уровней важности, чем перечисленные выше четыре, или определять для них другие имена. Например, в некоторых случаях приложения или рабочие среды могут использовать уровни важности *fatal* (неисправимая ошибка; более важный и опасный уровень, чем *error* (ошибка)) и *trace* (трассировка; менее важный уровень, чем *debug* (отладка)). В этой главе я обращаю ваше внимание только на наиболее часто используемые в реальных приложениях степени важности и соответствующие термины.



Рис. 5.10. Пирамида степеней важности журнальных сообщений. На вершине находятся самые важные сообщения, которые обычно требуют немедленного внимания. В основании представлены журнальные сообщения, предоставляющие подробности, которые редко бывают востребованными. При перемещении сверху вниз журнальные сообщения становятся менее важными, но их количество возрастает. Обычно сообщения отладочного уровня по умолчанию запрещены, но разработчик может разрешить их регистрацию, если для анализа требуются уточняющие подробности о выполнении приложения

В большинстве вариантов анализа вам не потребуются сообщения, классифицированные как *debug* (отладка). Кроме того, из-за слишком большого их количества выполнение анализа становится более сложным. Поэтому

сообщения уровня `debug` обычно запрещены, и разрешать их запись необходимо, только если возникла проблема, для устранения которой требуется больше подробностей.

Начиная изучать язык Java, вы узнаете, как вывести что-либо в консоли, используя `System.out` или `System.err`. Со временем вы осваиваете использование `printStackTrace()` для записи в журнал сообщения об исключении, как было показано в подразделе 5.1.2. Но эти способы работы с журналами в Java не обеспечивают достаточную гибкость при конфигурировании. Поэтому вместо применения таких способов в реальных приложениях я рекомендую пользоваться инструментальным комплектом (фреймворком) журналирования.

Реализация уровней журналирования осуществляется просто. В настоящее время экосистема Java предоставляет разнообразные варианты инструментальных комплектов, такие как `Logback`, `Log4j` и `Java Logging API`. Эти комплекты инструментов похожи друг на друга, а использовать их легко.

Рассмотрим конкретный пример реализации журналирования с использованием `Log4j` из проекта `da-ch5-ex2`. Для реализации функций журналирования с помощью `Log4j` сначала необходимо добавить зависимость `Log4j`. В нашем проекте Maven необходимо изменить файл `pom.xml`, добавив в него зависимость `Log4j`.

Листинг 5.3. Зависимости, которые необходимо добавить в файл `pom.xml`, чтобы использовать `Log4j`

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.14.1</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.14.1</version>
  </dependency>
</dependencies>
```

После включения этой зависимости в проект можно объявить экземпляр `Logger` в каждом классе, где необходимо записывать журнальные сообщения. При наличии `Log4j` самым простым способом создания экземпляра `Logger` является использование метода `LogManager.getLogger()`, как показано в листинге 5.4. Этот метод позволяет записывать в журнал сообщения с именами, совпадающими со степенью важности события, которое они представляют. Например, если нужно записать в журнал сообщение с уровнем важности `info`, то используется метод `info()`. Если требуется зафиксировать сообщение с уровнем важности `debug`, то применяется метод `debug()` и т. д.

Листинг 5.4. Запись в журнал сообщений с различными степенями (уровнями) важности

```

public class StringDigitExtractor {

    private static Logger log = LogManager.getLogger();           ❶

    private final String input;

    public StringDigitExtractor(String input) {
        this.input = input;
    }

    public List<Integer> extractDigits() {
        log.info("Extracting digits for input {}", input);         ❷
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < input.length(); i++) {
            log.debug("Parsing character {} of input {}",          ❸
                input.charAt(i), input);
            if (input.charAt(i) >= '0' && input.charAt(i) <= '9') {
                list.add(Integer.parseInt(String.valueOf(input.charAt(i))));
            }
        }

        log.info("Extract digits result for input {} is {}", input, list);
        return list;
    }
}

```

- ❶ Объявление экземпляра `Logger` в текущем классе для записи в журнал сообщений.
- ❷ Запись сообщения с уровнем важности `info` (информация).
- ❸ Запись сообщения с уровнем важности `debug` (отладка).

После того как вы решили, какие сообщения нужно регистрировать в журнале, и использовали экземпляр `Logger` для их записи, необходимо сконфигурировать `Log4j`, чтобы сообщить приложению, как и куда записывать эти сообщения. Мы будем использовать XML-файл с именем `log4j2.xml` для конфигурации `Log4j`. Этот файл обязательно должен находиться в пути классов приложения, поэтому мы добавляем его в папку ресурсов нашего проекта Maven. Необходимо определить три элемента (см. рис. 5.11):

- *logger* (диспетчер журналирования) – указывает `Log4j`, какие сообщения должны записываться и в какой аппендер;
- *appender* (аппендер) – указывает `Log4j`, куда записывать журнальные сообщения;
- *formatter* (форматтер) – указывает `Log4j`, как выводить сообщения.

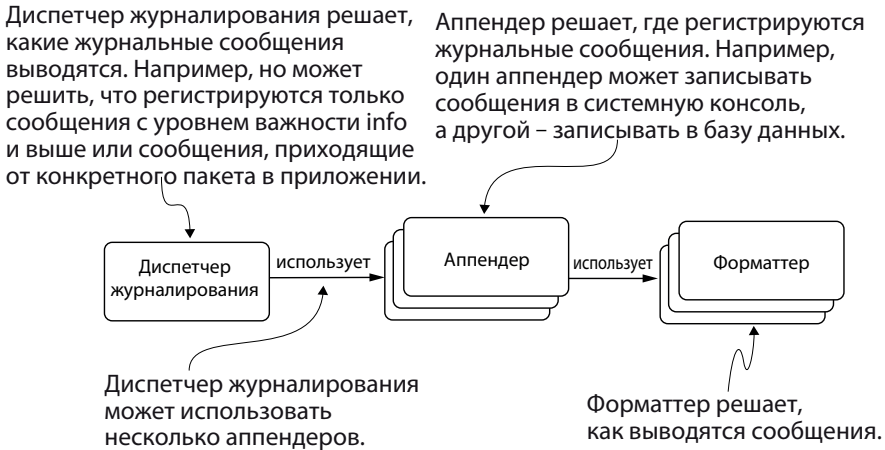


Рис. 5.11. Связи между аппендером, диспетчером журналирования и форматтером. Диспетчер журналирования использует один или несколько аппендеров и решает, что записывать (например, только журнальные сообщения, выводимые объектами в конкретном пакете). Диспетчер журналирования передает сообщения, которые должны быть записаны, в один или несколько аппендеров. Затем каждый аппендер реализует конкретный способ сохранения этих сообщений. Аппендер использует форматтеры для форматирования сообщений перед их сохранением

Диспетчер журналирования (logger) определяет, какие сообщения приложение записывает в журнал. В рассматриваемом здесь примере используется Root для записи сообщений из любой части приложения. Это уровень атрибута, имеющего значение info, означающее, что в журнал записываются только сообщения со степенью важности info и выше. Диспетчер журналирования также может решить, что регистрируются только сообщения из определенных частей приложения. Например, при использовании некоторой рабочей среды (фреймворка) сообщения, выводимые этой рабочей средой, редко привлекают ваше внимание, гораздо чаще интерес вызывают журнальные сообщения самого приложения, поэтому можно определить диспетчер журналирования, который исключает сообщения рабочей среды и выводит только сообщения от приложения. Следует помнить о необходимости записывать только важные журнальные сообщения. Иначе анализ может стать неоправданно затрудненным, поскольку придется отфильтровывать несущественные журнальные сообщения.

В реальных производственных приложениях можно определить несколько аппендеров, которые, вероятнее всего, будут сконфигурированы для сохранения сообщений в различных хранилищах, таких как база данных или файлы в конкретной файловой системе. В разделе 5.2.1 мы рассмотрели несколько возможных способов сохранения журнальных сообщений. Аппендеры – это просто реализации, отвечающие за сохранение журнальных сообщений выбранным способом.

Аппендеры также используют форматтеры, определяющие формат сообщения. В рассматриваемом здесь примере форматтер определяет, что сообщения должны содержать метку времени и уровень важности, так что приложению нужно передать только описание.

В листинге 5.5 показана конфигурация, определяющая аппендер и диспетчер журналирования. В этом примере определяется только один аппендер, указывающий Log4j, что необходимо регистрировать сообщения с записью в стандартный поток вывода системы (консоль).

Листинг 5.5. Конфигурация аппендера и диспетчера журналирования в файле *log4j2.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>                                ❶
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{yy-MM-dd HH:mm:ss.SSS} [%t]
        %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>                                    ❷
    <Root level="info">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

❶ Определение аппендера.

❷ Определение конфигурации диспетчера журналирования.

На рис. 5.12 наглядно показана связь между конфигурацией в формате XML и тремя компонентами, которые она определяет: диспетчером журналирования, аппендером и форматтером.

В приведенном ниже фрагменте показана часть журнала, выводимая во время выполнения примера. Обратите внимание: отладочные сообщения не регистрируются, поскольку их уровень важности ниже, чем *info* (строка 10 в листинге 5.5).

```
21-07-28 13:17:39.915 [main] INFO
➔ main.StringDigitExtractor
➔ Extracting digits for input ab1c
21-07-28 13:17:39.932 [main] INFO
➔ main.StringDigitExtractor
➔ Extract digits result for input ab1c is [1]
21-07-28 13:17:39.943 [main] INFO
➔ main.StringDigitExtractor
➔ Extracting digits for input a112c
```



```
21-07-28 13:17:39.944 [main] INFO
➔ main.StringDigitExtractor
➔ Extract digits result for input a112c is [1, 1, 2]
...
```

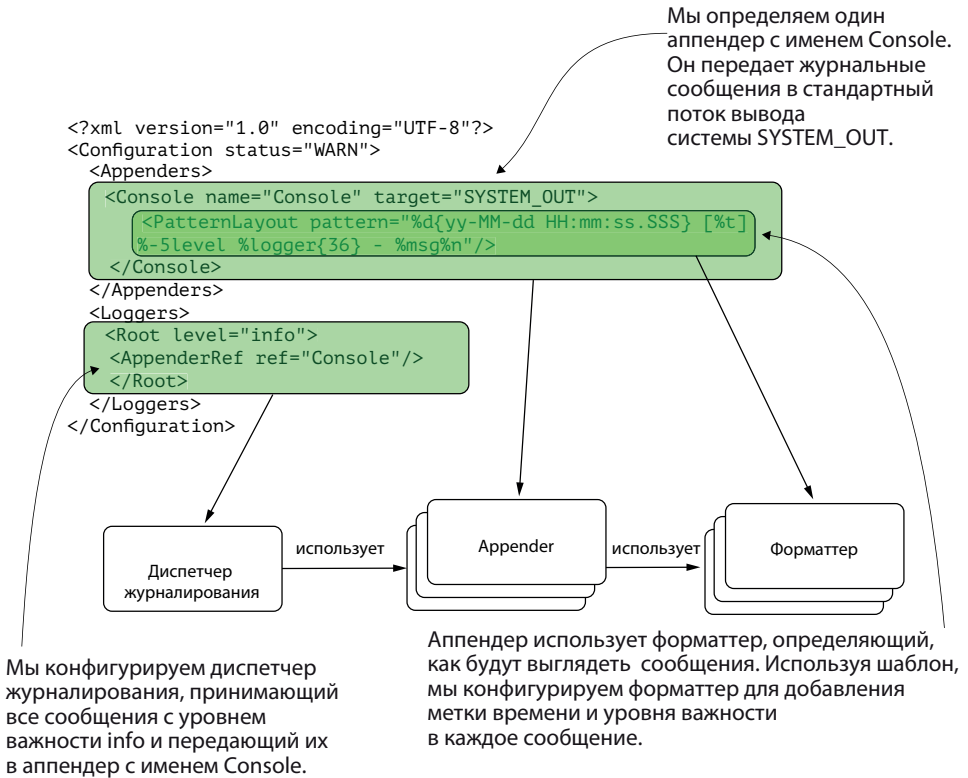


Рис. 5.12. Компоненты конфигурации. Диспетчер журналирования Root принимает все сообщения с уровнем важности info, записываемые приложением, и передает их в аппендер с именем Console. Аппендер сконфигурирован для вывода сообщений на системный терминал. Он использует форматтер для добавления к сообщению метки времени и уровня важности перед выводом

Если необходимо, чтобы приложение также записывало в журнал сообщения с уровнем важности debug (отладка), то мы должны изменить определение диспетчера журналирования.

Листинг 5.6. Использование конфигурации с другим уровнем важности

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
```

```

        <PatternLayout pattern="%d{yy-MM-dd HH:mm:ss.SSS} [%t]
            %-5level %logger{36} - %msg%n"/>
    </Console>
</Appenders>

<Loggers>
    <Root level="debug">                                ❷
        <AppenderRef ref="Console"/>
    </Root>
</Loggers>
</Configuration>

```

- ❶ Установка уровня журналирования для внутренних событий Log4j.
- ❷ Изменение уровня журналирования на debug (отладка).

В листинге 5.6 можно видеть состояние и уровень журналирования. Обычно это вызывает некоторое замешательство. Вы по большей части привыкли работать с атрибутом `level`, определяющим, какие сообщения будут записываться в журнал в соответствии с их степенью важности. Атрибут `status` в теге `<Configuration>` – это уровень важности событий Log4j, т. е. проблем, возникающих в этой библиотеке. То есть атрибут `status` – это конфигурация журналирования самой библиотеки журналирования.

Можно изменить диспетчер журналирования в листинге 5.6 так, чтобы он также записывал сообщения с приоритетом:

```

21-07-28 13:18:36.164 [main ] INFO
➡ main.StringDigitExtractor
➡ Extracting digits for input ab1c
21-07-28 13:18:36.175 [main] DEBUG
➡ main.StringDigitExtractor
➡ Parsing character a of input ab1c
21-07-28 13:18:36.176 [main] DEBUG
➡ main.StringDigitExtractor
➡ Parsing character b of input ab1c
21-07-28 13:18:36.176 [main] DEBUG
➡ main.StringDigitExtractor
➡ Parsing character 1 of input ab1c
21-07-28 13:18:36.176 [main] DEBUG
➡ main.StringDigitExtractor
➡ Parsing character c of input ab1c
21-07-28 13:18:36.177 [main] INFO
➡ main.StringDigitExtractor
➡ Extract digits result for input ab1c is [1]
21-07-28 13:18:36.181 [main] INFO
➡ main.StringDigitExtractor
➡ Extracting digits for input a112c
...

```

Библиотека журналирования обеспечивает гибкость, позволяющую записывать в журнал только то, что действительно необходимо. Запись минимального количества журнальных сообщений, необходимых для анализа конкретной проблемы, – правильная практическая методика, которая помогает быстрее понять журнальные записи и при этом сохранить производительность и удобство сопровождения приложения. Библиотека журналирования также предоставляет возможность конфигурирования журналов без необходимости перекомпиляции приложения.

5.2.3. Проблемы, возникающие при журналировании, и способы их устранения

Мы сохраняем журнальные сообщения, чтобы можно было использовать их для понимания поведения приложения в конкретный момент времени или в течение некоторого интервала времени. Журналы необходимы и чрезвычайно полезны во многих случаях, но при неправильном обращении они могут стать вредоносными. В этом подразделе рассматриваются три основные проблемы, возникающие при журналировании, и способы их устранения (см. рис. 5.13):

- проблемы безопасности и защиты информации – создаются журнальными сообщениями, публикующими секретные данные;
- проблемы производительности – создаются приложением, сохраняющим слишком многочисленные или излишне подробные журнальные сообщения;
- проблемы удобства сопровождения – создаются инструкциями журналирования, которые делают исходный код более трудным для чтения.

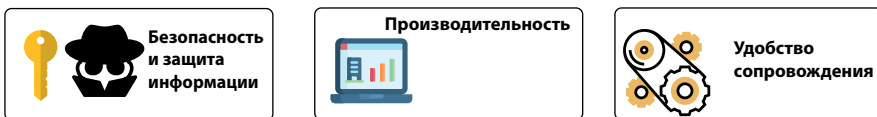


Рис. 5.13. Мелкие подробности могут создавать большие проблемы. Иногда разработчики считают функции журналирования приложения безвредными по умолчанию и не принимают во внимание проблемы, которые может создать журналирование. Но, как и все прочие функциональные компоненты программного обеспечения, журналирование имеет дело с данными, и при неправильной реализации может отрицательно повлиять на функциональность и удобство сопровождения приложения

Проблемы безопасности и защиты информации

Безопасность – одна из моих любимых тем и один из наиболее важных аспектов, которые должны учитывать разработчики при реализации приложения. Темой одной из написанных мною книг является безопасность,

и если вы занимаетесь реализацией приложений с использованием Spring Framework и хотите узнать больше об их защите, то я рекомендую прочитать ее: «Spring Security in Action» (Manning, 2020 г.).

Удивительный факт: иногда журналы могут создавать уязвимости в приложениях, и в большинстве случаев такие проблемы возникают, потому что разработчики не уделяют должного внимания подробностям, фиксируемым в журналах. Следует помнить о том, что подробности, описанные в журнальных записях, становятся видимыми для всех, кто может получить доступ к ним. Всегда необходимо думать о том, должны ли данные в журналах становиться видимыми тем, кто может получить к ним доступ (см. рис. 5.14).

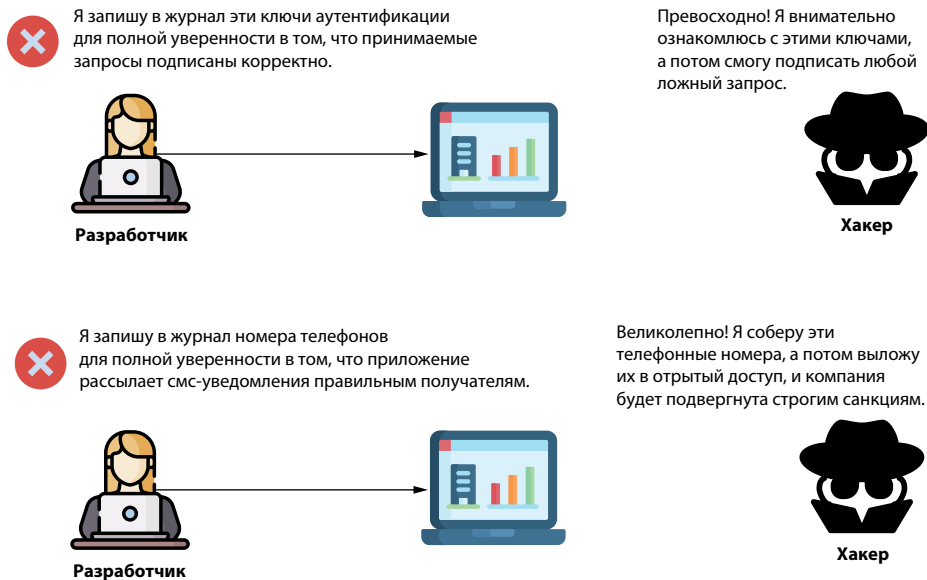


Рис. 5.14. Журнальные сообщения не должны содержать секретные или личные подробности. Ни один человек, работающий с приложением или с инфраструктурой, где разворачивается приложение, не должен получать доступ к таким данным. Запись важных подробностей в журналы может помочь злоумышленнику (хакеру) найти простые способы взлома системы или создания проблем, связанных с безопасностью

В приведенном ниже фрагменте показаны некоторые примеры журнальных сообщений, которые открывают важные подробности и создают уязвимости:

```
Successful login.
User bob logged in with password RwjBawIs66
```

```
Failed authentication.
The token is unsigned.
```

The token should have a signature with IVL4KiKMfz.

A new notification was sent to the following phone number +1233...

Что неправильно в представленных здесь журнальных записях? Два первых сообщения открывают секретные подробности. Вы никогда не должны записывать в журнал пароли или секретные ключи, используемые для подписи идентификационных элементов или любой другой передаваемой информации. Пароль должен знать только его владелец. Поэтому ни одно приложение не должно хранить пароли открытым текстом (в журнале или в базе данных). Закрытые ключи и аналогичные секретные подробности должны храниться в надежном сейфе, чтобы защитить их от похищения. Если кто-то получает значение такого ключа, то может выдать себя за приложение или пользователя.

В третьем примере журнального сообщения открывается номер телефона. Номер телефона считается личной информацией, и во всем мире особые правила ограничивают использование таких подробностей. Например, Евросоюз опубликовал правила General Data Protection Regulation (GDPR) в мае 2018 г. Приложение с пользователями в любом государстве Евросоюза должно в обязательном порядке соблюдать эти правила, чтобы избежать строгих санкций. Правила позволяют любому пользователю требовать немедленного удаления своих личных данных после использования их в приложении. Сохранение информации, подобной номерам телефонов, в журналах открывает такие секретные подробности и серьезно затрудняет их извлечение и удаление.

Проблемы производительности

Запись в журнал подразумевает передачу подробностей (обычно в форме строк) через поток ввода-вывода где-то за пределами приложения. Можно просто отправлять эту информацию в консоль приложения (на терминал) или сохранять ее в файлах или даже в базе данных, как было описано в подразделе 5.2.1. В любом случае необходимо помнить о том, что запись каждого сообщения в журнал – это также инструкция, выполнение которой требует определенного времени, и добавление слишком большого количества журнальных сообщений может существенно снизить производительность приложения.

Я вспоминаю проблему, которую моя группа анализировала несколько лет назад. Клиент из Азии сообщил о проблеме в приложении, которое мы разрабатывали для инвентаризации на промышленных предприятиях. Проблема не создавала слишком больших неудобств, но мы обнаружили, что весьма трудно найти ее главную причину, поэтому решили добавить больше журнальных сообщений. После доставки и установки патча (пакета исправлений) с небольшими изменениями система стала работать очень медленно, иногда почти переставала реагировать, и в итоге это привело к простоям в производстве. Мы вынуждены были срочно отменить вне-

сенные изменения. Вышло так, что каким-то образом мы сделали из мухи слона.

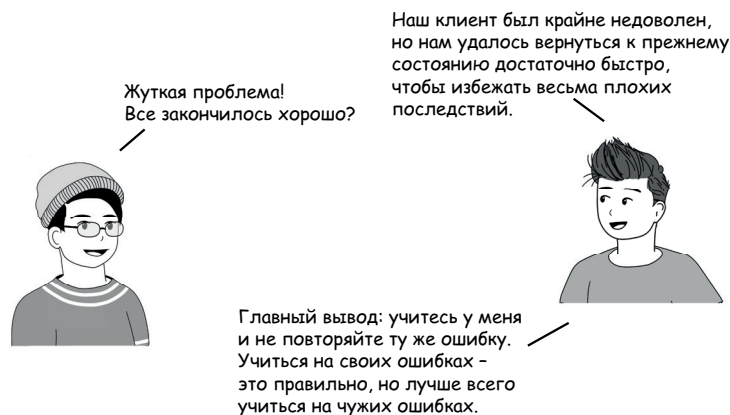


Рис. 5.15

Но как несколько простых журнальных сообщений могли стать причиной возникновения такой крупной проблемы? Журналы были сконфигурированы для отправки сообщений на отдельный сервер в сети, в которой постоянно работало приложение. На этом предприятии максимально замедлилась не только работа сети, но, кроме того, журнальное сообщение было добавлено в цикл, который выполнял итеративный проход по весьма большому количеству элементов, существенно замедляя выполнение приложения.

В итоге мы узнали некоторые вещи, которые помогли нам быть более внимательными и не повторять ту же ошибку:

- необходима полная уверенность в том, что вы понимаете, как приложение записывает сообщения в журнал. Напомню, что даже различные варианты развертывания одного и того же приложения могут иметь разные конфигурации (см. подраздел 5.2.2);
- следует избегать слишком большого количества сообщений. Не следует размещать инструкции записи сообщений в циклах, выполняющих итерации по значительному числу элементов. Кроме того, запись в журнал слишком большого количества сообщений затрудняет их чтение. Если необходима запись сообщения в большом цикле, то используйте условное выражение, чтобы сократить количество итераций, для которых сообщения записываются в журнал;
- следует убедиться в том, что приложение сохраняет конкретное журнальное сообщение только в том случае, когда оно действительно необходимо. Ограничивайте количество журнальных сообщений, выполняя сохранение с использованием уровней важности, как описано в подразделе 5.2.2;

- необходимо выполнить реализацию механизма журналирования так, чтобы можно было подключать и отключать его без перезапуска сервиса. Это позволит аккуратно управлять изменениями уровня журналирования, получать все требуемые подробности, следовательно, сделает подсистему журналирования менее подверженной внешним факторам.

Проблемы удобства сопровождения

Журнальные сообщения также могут отрицательно повлиять на удобство сопровождения приложения. Если журнальные сообщения добавляются слишком часто, то они могут сделать логику приложения более трудной для понимания. Рассмотрим эту проблему на примере: попробуйте прочитать листинги 5.7 и 5.8. Какой код легче понять?

Листинг 5.7. Метод, реализующий простой фрагмент логики

```
public List<Integer> extractDigits() {
    List<Integer> list = new ArrayList<>();
    for (int i = 0; i < input.length(); i++) {
        if (input.charAt(i) >= '0' && input.charAt(i) <= '9') {
            list.add(Integer.parseInt(String.valueOf(input.charAt(i))));
        }
    }

    return list;
}
```

Листинг 5.8. Метод, реализующий простой фрагмент логики, переполненный инструкциями записи сообщений в журнал

```
public List<Integer> extractDigits() {
    log.info("Creating a new list to store the result.");
    List<Integer> list = new ArrayList<>();
    log.info("Iterating through the input string " + input);
    for (int i = 0; i < input.length(); i++) {
        log.info("Processing character " + i + " of the string");
        if (input.charAt(i) >= '0' && input.charAt(i) <= '9') {
            log.info("Character " + i +
                " is digit. Character: " +
                input.charAt(i))
            log.info("Adding character" + input.charAt(i) + " to the list");
            list.add(Integer.parseInt(String.valueOf(input.charAt(i))));
        }
    }
}
```

```
Log.info("Returning the result " + list);  
return list;  
}
```

В обоих листингах показан один и тот же фрагмент реализации логики. Но в листинге 5.8 я добавил многочисленные инструкции записи сообщений в журнал, и из-за этого логику метода стало гораздо труднее читать.

Как избежать такого отрицательного влияния на удобство сопровождения приложения?

- Нет необходимости добавлять журнальное сообщение для каждой инструкции в коде. Следует определить те инструкции, которые предоставляют наиболее важные подробности. Помните, что вы можете в дальнейшем добавлять дополнительные журнальные сообщения, если недостаточно существующих.
- Сохраняйте размер кода методов относительно небольшим, чтобы требовалась запись в журнал только значений параметров и значения, возвращаемого методом после выполнения.
- Некоторые рабочие среды (фреймворки) позволяют выделять часть кода из метода. Например, в Spring можно использовать специализированные аспекты для регистрации в журнале результата выполнения метода (включая значения параметров и значения, возвращаемого методом после выполнения).

5.3. Сравнение журналирования с удаленной отладкой

В главе 4 мы рассматривали удаленную отладку, и вы узнали, что можно установить соединение между отладчиком и приложением, выполняющимся во внешней рабочей среде. Я привожу это сравнение, потому что обучающиеся часто спрашивают, зачем нужно использовать журналы, если можно установить соединение и напрямую отлаживать возникшую проблему. Но, как уже отмечалось ранее в этой главе, а также в предыдущих главах, эти методики отладки не исключают друг друга. Иногда одна из них подходит больше, в других случаях необходимо применять их вместе.

Давайте проанализируем, что можно и что невозможно сделать с помощью журналирования по сравнению с удаленной отладкой, чтобы точно узнать, как наиболее эффективно использовать обе эти методики. В табл. 5.1 показано подробное сравнение по отдельным пунктам журналирования и удаленной отладки.

Таблица 5.1. Сравнение журналирования и удаленной отладки

Функциональность	Журналирование	Удаленная отладка
Возможность использования для понимания поведения приложения, выполняющегося удаленно	✓	✓
Необходимость специализированных сетевых прав доступа или конфигураций	✗	✓
Постоянное хранение информации о выполнении	✓	✗
Возможность приостановки выполнения в конкретной строке кода, чтобы понять, что делает приложение	✗	✓
Возможность использования для понимания поведения приложения без воздействия на выполняемую логику	⚡	✗
Рекомендуется для применения в производственных рабочих средах	✓	✗

Можно использовать журналирование и удаленную отладку совместно, чтобы понять поведение приложения, выполняющегося в удаленном режиме. Но в обеих методиках существуют собственные затруднения. Журналирование подразумевает, что приложение записывает события и данные, требуемые для анализа. Если этого не происходит, то необходимо добавить такие инструкции и повторно развернуть приложение. Разработчики обычно называют это «добавлением экстралогов». Удаленная отладка позволяет отладчику установить соединение с приложением, выполняющимся удаленно, но при этом должны быть предоставлены специальные сетевые конфигурации и права доступа.

Самым главным различием является основной принцип (идея) каждой методики. Отладка сосредоточена на настоящем. Вы приостанавливаете выполнение и исследуете текущее состояние приложения. Журналирование в большей степени обращается к прошлому. Вы получаете набор журнальных сообщений и анализируете выполнение, согласуясь со шкалой времени. Часто отладка и журналирование используются одновременно, чтобы понять причины возникновения более сложных проблем, и на основе собственного опыта могу сказать, что иногда выбор журналирования или отладки зависит от предпочтений разработчика. Иногда я замечаю, что разработчики используют одну из методик просто потому, что она более удобна для них, чем другая.

5.4. Резюме

- Всегда проверяйте журналы приложения, прежде чем начать анализ любой проблемы. Журналы могут показать, что пошло не так, или, по крайней мере, указать начальный пункт для анализа.
- Все журнальные сообщения должны содержать метку времени. Помните, что в большинстве случаев система не гарантирует порядок записи в журналы. Метка времени помогает разместить журнальные сообщения в хронологическом порядке.
- Избегайте сохранения слишком большого количества журнальных сообщений. Не каждая подробность важна или полезна при анализе предполагаемой проблемы, и сохранение множества лишних журнальных сообщений может отрицательно влиять на производительность приложения и сделать код более трудным для чтения.
- Вы должны реализовать большой объем журналирования, только если это действительно необходимо. Выполняющееся приложение должно регистрировать в журнале только самые важные события. Если требуются дополнительные подробности, то всегда можно разрешить на короткое время записывать в журнал больше сообщений.
- Исключение, зарегистрированное в журнале, не всегда является главной причиной проблемы. Оно может являться последствием проблемы. Поэтому необходимо проанализировать, что стало причиной исключения, прежде чем предпринимать действия по локальному устранению исключения.
- Можно использовать трассировки стека исключений для определения элемента, вызвавшего конкретный исследуемый метод. В крупной, запутанной, сложной для понимания кодовой базе такой подход может оказаться чрезвычайно полезным и позволит сэкономить время.
- Никогда не записывайте важные секретные подробности (например, пароли, секретные ключи или личные данные) в журнальные сообщения. Запись в журнал паролей или секретных ключей создает уязвимости в системе безопасности, поскольку все, кто имеет доступ к журналам, могут увидеть и воспользоваться этой информацией. Запись личных данных, таких как фамилии, имена, адреса и номера телефонов, также может быть запрещена различными государственными законами и правилами.

Часть II

Глубокий анализ выполнения приложения

В части II будут рассматриваться продвинутые методики, которые можно использовать для более глубокого анализа выполнения приложения. Большинство разработчиков предпочитает отладку и журналирование (описанные в части I), но немногие знают о том, как раскрыть все «секреты» процесса выполнения, применяя методики профилирования, анализ потоков и потребления памяти. Знание этих методик чрезвычайно важно, и иногда это становится единственным способом решения сложных задач по устранению проблем.



В главе 6 вы узнаете, как проанализировать потребление ЦП и памяти. В главе 7 рассматривается использование профилировщика для определения проблем с задержками. Главы 8 и 9 дают более глубокое представление о многопоточных архитектурах и использовании в них профилировщиков. В главе 10 мы обсудим дампы потоков. Часть II завершается главой 11, из которой вы узнаете, как выявить проблемы потребления памяти с помощью дампов кучи.

Глава 6

.....

Выявление проблем потребления ресурсов с использованием методик профилирования

Темы:

- оценка потребления ресурсов;
- выявление проблем потребления ресурсов.

А для тебя, Фродо Бэггинс, я приготовила свет Эарендила, нашей самой любимой звезды. Пусть он светит тебе во тьме, когда погаснут остальные огни.

— Галадриэль («Властелин колец», Дж. Р. Р. Толкиен)

В этой главе мы начнем использовать профилировщик, а продолжим изучение его применения в главе 7. Возможно, профилировщик не обладает такой мощностью, как свет звезды Эарендил, но этот инструмент определенно освещает темные места, когда гаснут все остальные огни. Профилировщик (profiler) – это мощное инструментальное средство, которое помогало мне понять главную причину странного поведения приложения во многих сложных ситуациях. Я считаю изучение практического применения профилировщика обязательным для всех разработчиков, так как он может стать компасом, указывающим на причину возникновения проблемы, выглядящей безнадежной. Как вы узнаете из этой главы, профилировщик перехватывает выполняющиеся процессы JVM и предоставляет чрезвычайно полезные подробности:

- как приложение потребляет ресурсы, например ЦП и память;
- потоки во время выполнения и их текущее состояние;
- код во время выполнения и ресурсы, расходуемые конкретным фрагментом кода (например, продолжительность выполнения каждого метода).

В разделе 6.1 мы будем анализировать некоторые сценарии, чтобы вы могли увидеть и понять, как подробности, предоставляемые профилировщиком, могут приносить пользу и почему они так важны. В разделе 6.2 рассматривается использование профилировщика для решения проблем в сценариях из раздела 6.1. В подразделе 6.2.1 мы начнем с установки и конфигурирования профилировщика. Затем в подразделе 6.2.2 проанализируем, как приложение потребляет системные ресурсы, а в подразделе 6.2.3 узнаем, как определить, что в приложении возникают проблемы с управлением используемой памятью. Изучение практического использования профилировщика продолжится в главе 7, из которой вы узнаете, как идентифицировать выполняющийся код и обнаружить связанные с ним проблемы производительности.

В примерах этой главы я использую профилировщик VisualVM. Это бесплатная программа-профилировщик и превосходный инструмент, который я успешно применяю в течение многих лет. Скачать VisualVM можно здесь: <https://visualvm.github.io/download.html>. VisualVM не единственный инструмент профилирования для приложений Java. Существуют и другие широко известные средства профилирования: Java Mission Control (<http://mng.bz/AVQE>) и JProfiler (<http://mng.bz/Zplj>).

6.1. В каких случаях профилировщик оказывается полезным

В этом разделе мы анализируем три сценария, в которых инструмент профилирования может помочь вам:

- выявление аномального использования ресурсов;
- определение того, какая часть кода выполняется;
- выявление мест, где замедляется выполнение приложения.

6.1.1. Выявление аномального использования ресурсов

Профилировщик чаще всего используется для определения того, как приложение потребляет ЦП и память. Это помогает понять специфические проблемы, возникающие в конкретном приложении. Таким образом, это является самым первым этапом анализа подобных проблем. Наблюдение за тем, как приложение потребляет ресурсы, обычно приводит к выявлению двух категорий проблем:

- проблем, связанных с потоками, – обычно это проблемы параллельного выполнения, возникающие из-за отсутствия синхронизации или неправильной ее организации;
- утечки памяти – ситуаций, в которых в приложении возникают ошибки при удалении ненужных данных из памяти, что вызывает замедление выполнения и с большой вероятностью приводит к полному критическому отказу в работе приложения.

В реальных производственных приложениях мне встречались оба типа проблем гораздо чаще, чем хотелось бы. Проблемы при использовании ресурсов создают весьма разнообразные эффекты. В некоторых случаях они просто замедляют работу приложения. В других ситуациях они могут привести к аварийному завершению всего приложения. Моей «любимой» задачей, связанной с потоками, которую я решал с использованием профилировщика, была проблема с аккумулятором мобильного устройства. Замедление работы было не самой большой проблемой. Пользователи жаловались на то, что заряд аккумулятора их мобильного устройства неестественно быстро заканчивался, когда они использовали именно это Android-приложение. Такое поведение определенно требовало анализа. После наблюдения в течение некоторого времени за поведением приложения я обнаружил, что одна из библиотек, используемых приложением, иногда создавала потоки, которые продолжали выполняться, ничего не делали, но потребляли системные ресурсы. В мобильном приложении использование ресурса ЦП часто отражается на потреблении заряда аккумулятора.

После обнаружения предполагаемой проблемы вы можете продолжить анализ с использованием дампов потоков, о которых вы узнаете подробнее в главе 10. В общем случае главной причиной возникновения подобных проблем является дефектная синхронизация потоков.

Кроме того, я время от времени обнаруживаю утечки памяти в приложениях. В большинстве случаев конечным результатом утечек памяти становится критическая ошибка `OutOfMemoryError`, которая приводит к аварийному завершению приложения. Поэтому, когда я слышу о «падении» приложения, то обычно предполагаю, что возникла проблема с памятью.



СОВЕТ. Когда вам встречается приложение, которое аварийно завершается случайным образом, вы непременно должны рассмотреть вариант с утечкой памяти.

Часто главной причиной аномального использования ресурсов является ошибка при кодировании, которая позволяет объекту ссылаться на существующие объекты даже после того, как они становятся ненужными. Следует помнить о том, что, хотя в JVM имеется автоматический механизм удаления ненужных данных из памяти (этот механизм называется сбор-

щиком мусора (garbage collector – GC)), разработчик продолжает в полной мере отвечать за удаление всех ссылок на ненужные данные. Если реализуется код, в котором остаются ссылки на объекты, то GC не знает о том, что они больше не нужны, и не удалит их. Такая ситуация называется утечкой памяти (memory leak). В подразделе 6.2.3 вы научитесь использовать профилировщик для выявления этой проблемы, а затем в главе 11 узнаете о том, как обнаружить главную причину ее возникновения с помощью дампа динамической памяти (кучи).

6.1.2. Как определить, какой код выполняется

Как разработчик и консультант, я иногда работал с крупными, сложными и неаккуратно организованными кодовыми базами. Несколько раз я попадал в ситуации, когда был вынужден анализировать конкретное функциональное свойство приложения, – я мог воспроизвести проблему, но не понимал, какая часть кода при этом выполнялась. Несколько лет назад я изучал проблему в старом приложении, запускающем несколько процессов. Руководство компании приняло необоснованное решение, позволяющее только одному разработчику отвечать за весь код. Никто другой ничего не знал о том, что было в этом коде и как с ним работать. Когда этот разработчик уволился, не оставив после себя ни документации, ни удобной для чтения кодовой базы, меня попросили помочь в обнаружении причины возникшей проблемы. Первый взгляд на этот код слегка испугал меня: в приложении отсутствовало проектное решение на основе классов, это была комбинация Java и Scala, смешанная с некоторым кодом Java, управляемым рефлексией.

Как определить, какой код необходимо анализировать в подобном случае? К счастью, в профилировщике есть функция выборки выполняющегося кода. Инструмент перехватывает методы и наглядно показывает, что именно выполняется, предоставляя информацию, достаточную для начала анализа. После определения выполняющегося кода можно его читать и в конце концов воспользоваться отладчиком, как описано в главах 2–4.

С помощью профилировщика можно найти код, выполняющийся незаметно для внешнего наблюдателя, без предварительного просмотра кодовой базы. Эта функциональная возможность называется выборкой (sampling), и она особенно полезна, когда код настолько запутан, что невозможно даже понять, что в нем вызывается.

6.1.3. Определение узких мест (замедлений) при выполнении приложения

Иногда приходится иметь дело с проблемами производительности. В подобных случаях необходимо ответить на главный вопрос: «Что отнимает так много времени на выполнение?» На основе практического опыта разработчики всегда в первую очередь обращают внимание на части кода, связанные с вводом-выводом. Вызов веб-сервиса, установление соединения

с базой данных или сохранение данных в файле – это примеры операций ввода-вывода, которые чаще всего создают задержки в приложениях. Тем не менее обращение к подсистеме ввода-вывода не всегда является причиной возникновения замедления. Но даже в этом случае, если вам не известна кодовая база во всех подробностях (что бывает крайне редко), сохраняется затруднение в точном определении места возникновения проблемы без дополнительной помощи.

К счастью, профилировщик обладает поистине «волшебными» свойствами – предоставляет возможность перехватывать код во время выполнения и вычислять ресурсы, потребляемые каждым фрагментом кода. Эти возможности мы рассмотрим в главе 7.

6.2. Использование профилировщика

В этом разделе рассматривается, как применять профилировщик для устранения проблем, подобных описанным в разделе 6.1. Мы начнем с установки и конфигурирования профилировщика VisualVM (в подразделе 6.2.1). Затем рассмотрим функции анализа профилировщика. Для наглядной демонстрации по каждой теме я использую приложение, довольно небольшое, чтобы вы могли сосредоточиться на предоставляемом материале, но достаточно сложное, чтобы представлять интерес для обсуждения.

В подразделе 6.2.2 мы рассмотрим потребление системных ресурсов и узнаем, как определить, возникают ли в приложении проблемы, связанные с чрезмерным потреблением ресурсов. В подразделе 6.2.3 вы узнаете о типах проблем с памятью, которые могут возникать в приложении, и о способах их обнаружения.

6.2.1. Установка и конфигурирование профилировщика VisualVM

В этом подразделе вы узнаете, как установить и сконфигурировать профилировщик VisualVM. Прежде чем использовать профилировщик или любое аналогичное инструментальное средство, необходимо убедиться в том, что он правильно установлен и сконфигурирован. Затем можно воспользоваться примерами из этой книги, чтобы попробовать применить функциональные возможности, описанные в текущей главе. Если вы работаете с реальным производственным проектом, то я рекомендую использовать описанные здесь методики в реализуемом приложении.

Установка VisualVM выполняется просто. После скачивания версии, соответствующей вашей операционной системе, с официального сайта (<https://visualvm.github.io/download.html>) все, что нужно сделать, – убедиться в том, что вы правильно сконфигурировали место расположения комплекта JDK, который должен использовать VisualVM. В файле конфигурации, который находится в подкаталоге *etc/visualvm.config* в папке VisualVM, определите локацию JDK в вашей системе. Необходимо присвоить путь к JDK переменной

visualvm_jdkhome и раскомментировать строку (удалить символ # в ее начале), показанную ниже. VisualVM работает с версией Java 8 или более поздней:

```
visualvm_jdkhome="C:\Program Files\Java\openjdk-17\jdk-17"
```

После конфигурирования локации JDK можно запустить VisualVM, используя выполняемый код в подкаталоге *bin*, расположенном там, где установлено приложение. Если локация JDK сконфигурирована правильно, то приложение запустится, и вы увидите интерфейс, аналогичный изображенному на рис. 6.1.

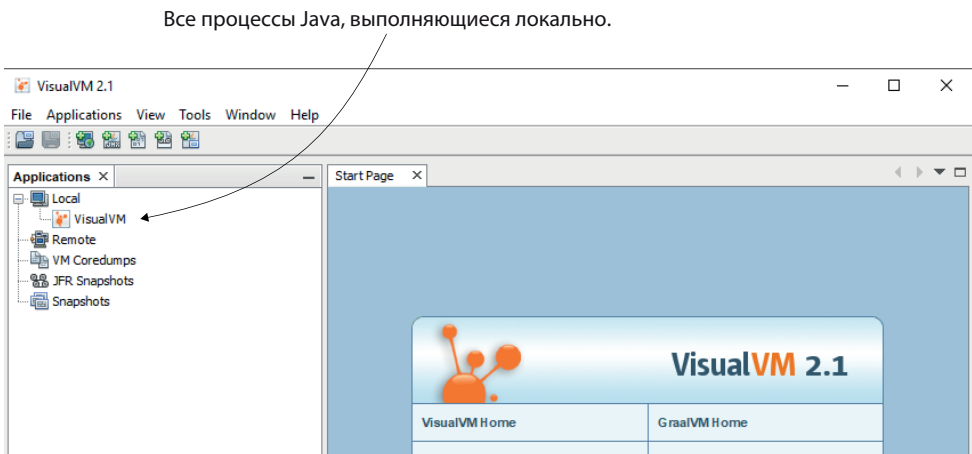


Рис. 6.1. Стартовый экран профилировщика VisualVM. После конфигурирования и запуска VisualVM вы обнаружите, что он предлагает простой и удобный GUI. Слева от панели с приветствием показаны процессы, выполняющиеся локально, которые можно анализировать с помощью этого инструмента

Запустим Java-приложение. Вы можете использовать проект *da-ch6-ex1*, прилагаемый к этой книге. Для запуска приложения используйте IDE или выполните соответствующую команду прямо в консоли. Профилирование процесса Java не зависит от способа запуска приложения.

После того как приложение начало работу, VisualVM выводит соответствующий процесс в левой панели. Обычно, если вы не присвоили явно процессу какое-либо имя, VisualVM использует имя основного класса *main*, как показано на рис. 6.2.

В общем случае запуск конкретного приложения вполне достаточно. Но в некоторых ситуациях VisualVM не знает, как установить соединение с локальным процессом из-за разнообразных проблем, как показано на рис. 6.3. В этом случае в первую очередь необходимо попытаться явно задать имя домена, используя VM-аргумент при запуске приложения, которое вы хотите профилировать:

```
-Djava.rmi.server.hostname=localhost
```

После запуска приложения вы увидите соответствующий процесс в левой панели интерфейса VisualVM. Поскольку имя для этого процесса не задано, VisualVM выводит имя класса main.

Выполните двойной щелчок по имени процесса, и VisualVM выведет для него вкладку **Details** (Подробности).

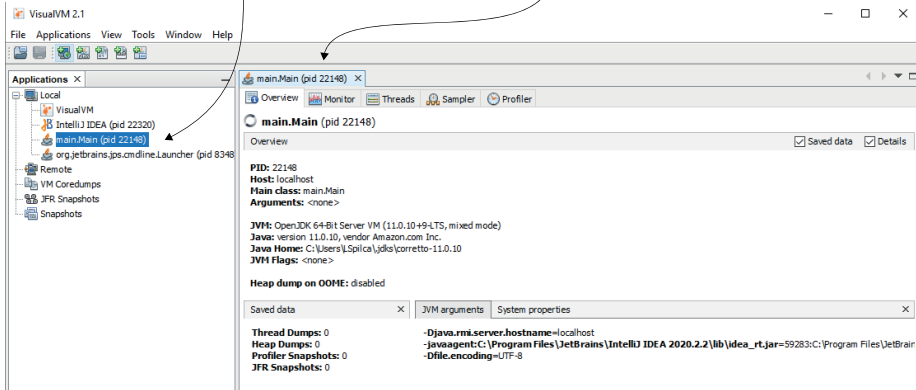


Рис. 6.2. Выполните двойной щелчок по имени процесса, чтобы начать использование VisualVM для его анализа, и появится новая вкладка.

В этой вкладке находятся все необходимые функции VisualVM, предоставляемые для исследования данного конкретного процесса

Вы узнаете о некорректной конфигурации, если отсутствует одна из вкладок (как в данном случае отсутствует вкладка **Threads**) или если VisualVM выводит сообщение об ошибке, показанное здесь: сконфигурированная версия JVM не поддерживается.

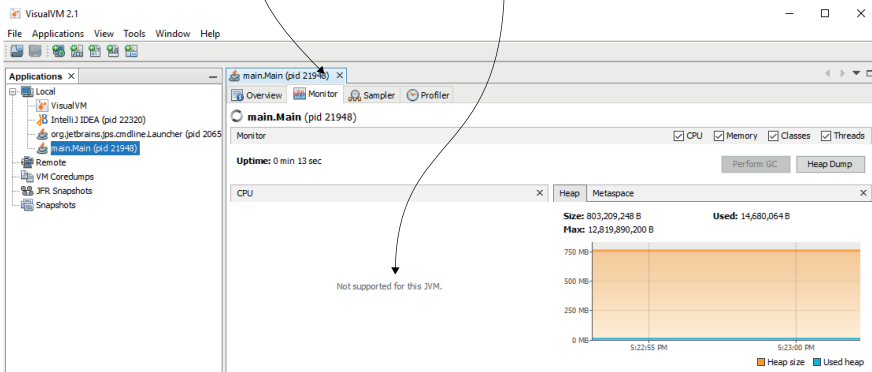


Рис. 6.3. Если профилировщик не работает надлежащим образом, то необходимо проверить правильность его конфигурации. Подобная проблема может возникать, если указанная в конфигурации версия дистрибутива JVM отсутствует в списке поддерживаемых VisualVM. Иногда профилировщик не может установить соединение с локальным процессом, требующим анализа, именно по этой причине. В таких случаях используйте другую версию дистрибутива JVM, соответствующую требованию профилировщика, или проверьте, как был запущен процесс, требующий анализа

Аналогичная проблема также может возникать из-за использования версии JVM, не поддерживаемой VisualVM. Если добавление аргумента `-Djava.rmi.server.hostname=localhost` не устранило проблему, то проверьте наличие указанной в конфигурации версии дистрибутива JVM в списке поддерживаемых профилировщиком VisualVM (список находится на странице загрузки веб-сайта <https://visualvm.github.io/download.html>).

6.2.2. Наблюдение за использованием ЦП и памяти

Одной из простейших операций, которые можно выполнить с помощью профилировщика, является наблюдение за тем, как приложение потребляет системные ресурсы. При этом можно обнаружить такие проблемы, как утечки памяти или потоки-зомби в приложении.



ОПРЕДЕЛЕНИЕ. Утечка памяти (memory leak) возникает, когда приложение не освобождает память, занятую уже ненужными данными. Со временем свободная память может закончиться. Это и есть проблема.

Как вы узнаете в этом подразделе, профилировщик можно использовать для наглядного (визуального) подтверждения того, что приложение ведет себя некорректно. Например, потоки-зомби (zombie threads) – это потоки, которые остаются непрерывно выполняемыми, потребляя ресурсы приложения. Такие проблемы легко обнаружить, используя VisualVM.

Я подготовил несколько проектов, чтобы показать, как применять профилировщик для обнаружения в приложении проблем, вызывающих аномальное потребление ресурсов. Мы будем по очереди выполнять эти приложения, используя VisualVM для наблюдения за поведением и выявления отклонений от нормы.

Начнем с приложения `da-ch6-ex1`. Принцип его работы прост: два потока постоянно добавляют значения в список, а два других потока постоянно удаляют (потребляют) эти значения из списка. Такую реализацию часто называют моделью производитель–потребитель (producer-consumer) – это многопоточный паттерн проектирования, часто встречающийся в приложениях.

Листинг 6.1. Поток-производитель, добавляющий значения в список

```
public class Producer extends Thread {

    private Logger log = Logger.getLogger(Producer.class.getName());

    @Override
    public void run() {
        Random r = new Random();
        while (true) {
            if (Main.list.size() < 100) {
```

```

        int x = r.nextInt();
        Main.list.add(x);
        log.info("Producer " + Thread.currentThread().getName() +
                " added value " + x);
    }
}
}
}

```

❶ Устанавливает максимальное число значений в списке.

❷ Добавляет случайное значение в список.

В листинге 6.2 показана реализация потока-потребителя.

Листинг 6.2. Поток-потребитель, удаляющий значения из списка

```

public class Consumer extends Thread {

    private Logger log = Logger.getLogger(Consumer.class.getName());

    @Override
    public void run() {
        while (true) {
            if (Main.list.size() > 0) {
                int x = Main.list.get(0);
                Main.list.remove(0);
                log.info("Consumer " + Thread.currentThread().getName() +
                        " removed value " + x);
            }
        }
    }
}
}

```

❶ Проверяет, содержит ли список какое-либо значение.

❷ Если список содержит значения, то удаляет первое значение из списка.

Класс Main создает и выполняет два экземпляра потока-производителя и два экземпляра потока-потребителя.

Листинг 6.3. Класс Main, создающий и запускающий потоки-производители и потоки-потребители

```

public class Main {

    public static List<Integer> list = new ArrayList<>();

    public static void main(String[] args) {

```

```

        new Producer().start();
        new Producer().start();
        new Consumer().start();
        new Consumer().start();
    }
}

```

❷

- ❶ Создает список для хранения случайных значений, генерируемых производителем.
- ❷ Начинает выполнение потоков-производителей и потоков-потребителей.

В этом приложении неправильно реализована многопоточная архитектура. Точнее говоря, несколько потоков одновременно получают доступ и изменяют список типа `ArrayList`. Поскольку тип `ArrayList` не является реализацией набора (коллекции) с параллельным режимом доступа в Java, сам по себе он не управляет собственно доступом. Несколько потоков, получивших доступ к этому набору данных, предположительно входят в состояние гонки (*race condition*). Состояние гонки возникает, когда несколько потоков соперничают за доступ к некоторому ресурсу. То есть они находятся в состоянии гонки за доступ к одному и тому же ресурсу.

В реализации проекта `da-ch6-ex1` отсутствует синхронизация потоков. После запуска приложения некоторые из потоков останавливаются через короткое время из-за исключений, вызванных состоянием гонки, тогда как другие продолжают непрерывно оставаться активными, ничего при этом не делая (потоки-зомби). Воспользуемся `VisualVM` для обнаружения всех этих проблем. Затем перейдем к проекту `da-ch6-ex2`, в котором применено исправление, синхронизирующее доступ потоков к списку. Сравним результаты, демонстрируемые `VisualVM` в первом примере, со вторым примером, чтобы понять различие между обычным и проблемным приложением.

Приложение будет выполняться быстро, а затем остановится (предположительно показывая трассировку стека исключений в консоли). Приведенный ниже фрагмент кода показывает, как выглядят журнальные сообщения, которые приложение выводит в консоли:

```

Aug 26, 2021 5:22:42 PM main.Producer run
INFO: Producer Thread-0 added value -361561777
Aug 26, 2021 5:22:42 PM main.Producer run
INFO: Producer Thread-1 added value -500676534
Aug 26, 2021 5:22:42 PM main.Producer run
INFO: Producer Thread-0 added value 112520480

```

Вам может показаться, что, поскольку это приложение содержит только три класса, нет необходимости в профилировщике, чтобы обнаружить проблему, – здесь достаточно чтения кода. Разумеется, при наличии всего лишь трех классов, возможно, вы легко идентифицируете проблему без использования отдельного инструмента. Это потому, что рассматриваемые здесь приложения являются упрощенными примерами, позволяющими

вам сосредоточиться на использовании профилировщика. Но в реальной практике приложения более сложны, а проблемы гораздо труднее обнаружить без соответствующего инструментального средства (такого как профилировщик).

Даже если кажется, что выполнение приложения приостановлено, вы можете увидеть кое-что интересное при использовании VisualVM для анализа того, что происходит внутри, незаметно для пользователя. Чтобы проанализировать такое неожиданное поведение, выполните следующие шаги.

1. Проверьте, как процесс использует ЦП.
2. Проверьте, как процесс использует память.
3. Выполните визуальный анализ выполняющихся потоков.

Рассматриваемый процесс потребляет огромный объем ресурсов ЦП, поэтому каким-то образом выглядит сохраняющим активность. Для наблюдения за потреблением этого ресурса используйте вкладку **Monitor** в окне **VisualVM** после двойного щелчка по имени процесса в левой панели. Один из виджетов на этой вкладке показывает использование ЦП (см. рис. 6.4).

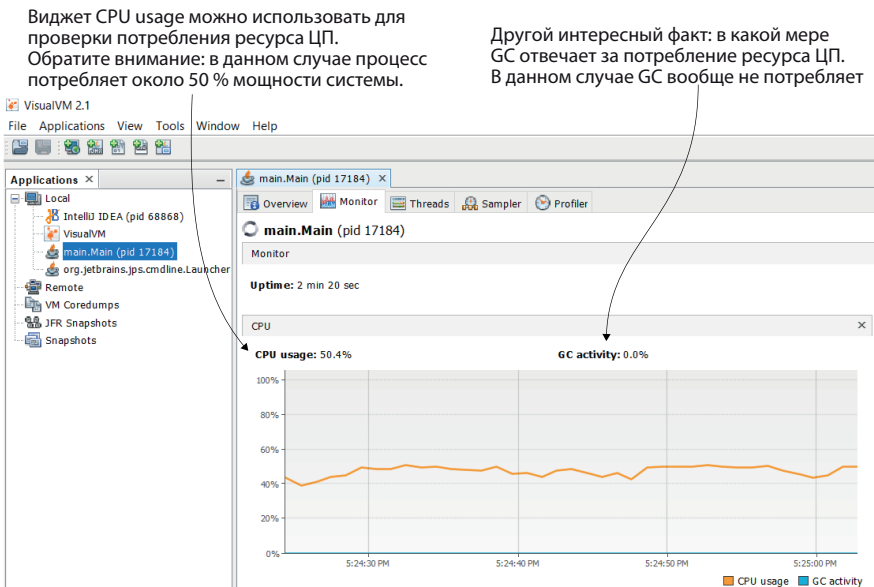


Рис. 6.4. Применение VisualVM для наблюдения за использованием ресурсов ЦП. Виджет на вкладке **Monitor** показывает степень использования ЦП процессом и какая часть этого ресурса используется сборщиком мусора (GC). Эта информация помогает понять, существуют ли проблемы в данном приложении, и представляет собой превосходное руководство для выполнения дальнейших этапов анализа.

В рассматриваемом здесь конкретном примере процесс потребляет около 50 % ЦП. Сборщик мусора не влияет на это значение. Эти данные часто являются признаками наличия потоков-зомби, которые обычно возникают из-за проблем при параллельном выполнении

Наблюдение показывает, что потоки-производители и потоки-потребители вошли в состояние непрерывного выполнения, в котором они потребляют системные ресурсы, даже если не выполняют надлежащим образом свои задачи. В этом случае такое состояние является последствием гонки (по данным), поскольку потоки пытаются получить доступ и изменить набор (коллекцию) данных, не поддерживающий параллельный режим обработки. Но мы уже знаем, что с приложением что-то не так. Необходимо наблюдать за признаками таких проблем, чтобы в других подобных ситуациях точно знать, что в приложении возникла та же проблема.

В этом виджете также можно увидеть объем ресурсов ЦП, используемых GC. Сборщик мусора GC – это механизм JVM, предназначенный для удаления из памяти данных, которые больше не нужны приложению. Использование ЦП сборщиком мусора является важной информацией, поскольку может показать, что в приложении возникли проблемы с распределением памяти. Если GC потребляет слишком большой объем ресурсов ЦП, это может указывать на то, что в приложении существует проблема утечки памяти.

В рассматриваемом здесь примере GC вообще не потребляет какие-либо ресурсы. Но это тоже плохой признак. Другими словами, приложение расходует огромный объем мощности процессора, но не обрабатывает никакие данные. Такие признаки означают наличие потоков-зомби, которые обычно являются последствием проблем при параллельном выполнении.

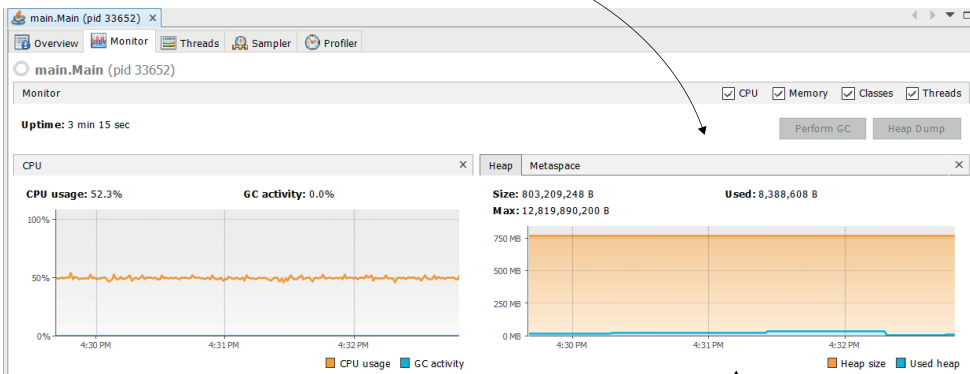
Следующий шаг – наблюдение за виджетом, в котором показано потребление памяти. Этот виджет вполне разумно размещен рядом с виджетом, демонстрирующим потребление ЦП, как показано на рис. 6.5. Более подробно мы рассмотрим этот виджет в подразделе 6.2.3, а сейчас отметим, что приложение почти не потребляет память. И такое поведение также является плохим признаком, поскольку равнозначно фразе: «Приложение ничего не делает». Используя всего лишь два этих виджета, мы можем сделать вывод: вероятнее всего, в приложении существует проблема параллельного выполнения.

Использование дампов потоков мы рассмотрим в главе 10. А сейчас сосредоточимся только на виджетах высокого уровня, предоставляемых профилировщиком, и сравним результаты наблюдения за этими виджетами, полученные при выполнении дефектного и нормального приложений.



ПРИМЕЧАНИЕ. До перехода к более подробному анализу потоков во время выполнения я предпочитаю использовать VisualVM для визуального наблюдения за выполнением потоков. В большинстве случаев это дает некоторые сведения о том, каким потокам необходимо уделить особое внимание. После получения этой информации я использую дампы потока, чтобы обнаружить проблему параллельного выполнения и понять, как ее устранить.

Справа от виджета использования ЦП находится другой виджет, показывающий потребление памяти.



Обратите внимание: даже несмотря на то, что приложение расходует 50 % ЦП, оно почти не потребляет память.

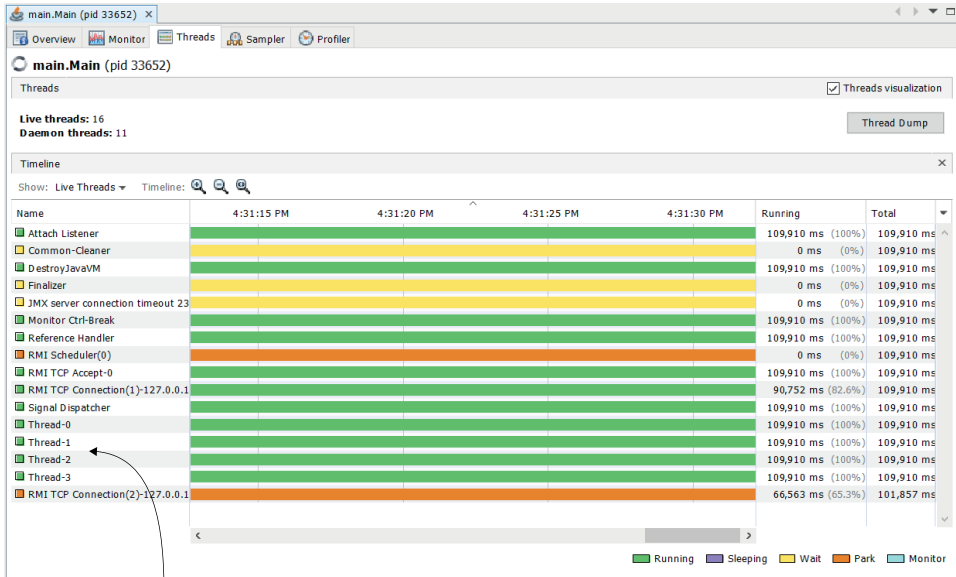
Рис. 6.5. Справа от виджета использования ЦП находится виджет использования памяти. В рассматриваемом здесь примере приложение почти не использует память. Это также является причиной нулевой активности GC. Отсутствие потребления памяти приложением означает, что оно ничего не делает

На рис. 6.6 показана вкладка **Threads** (Потоки), расположенная рядом с вкладкой **Monitor**. Вкладка **Threads** показывает визуальное представление потоков во время выполнения, а также их состояние. В рассматриваемом здесь примере все четыре потока, запущенные приложением, выполняются и находятся в активном (Running) состоянии.

Проблемы при параллельном выполнении могут приводить к различным результатам. Например, не всегда все потоки продолжают существовать. Иногда одновременный доступ может привести к генерации исключений, которые полностью прерывают работу некоторых или всех потоков. В приведенном ниже фрагменте показан пример такого исключения, которое может генерироваться во время выполнения приложения:

```
Exception in thread "Thread-1"
↳ java.lang.ArrayIndexOutOfBoundsException:
↳ Index -1 out of bounds for length 109
at java.base/java.util.ArrayList.add(ArrayList.java:487)
at java.base/java.util.ArrayList.add(ArrayList.java:499)
at main.Producer.run(Producer.java:16)
```

Если возникает такое исключение, то некоторые потоки могут быть остановлены, и на вкладке **Threads** они не отображаются. На рис. 6.7 показан случай, в котором приложение сгенерировало исключение, и в результате остался существующим только один поток.

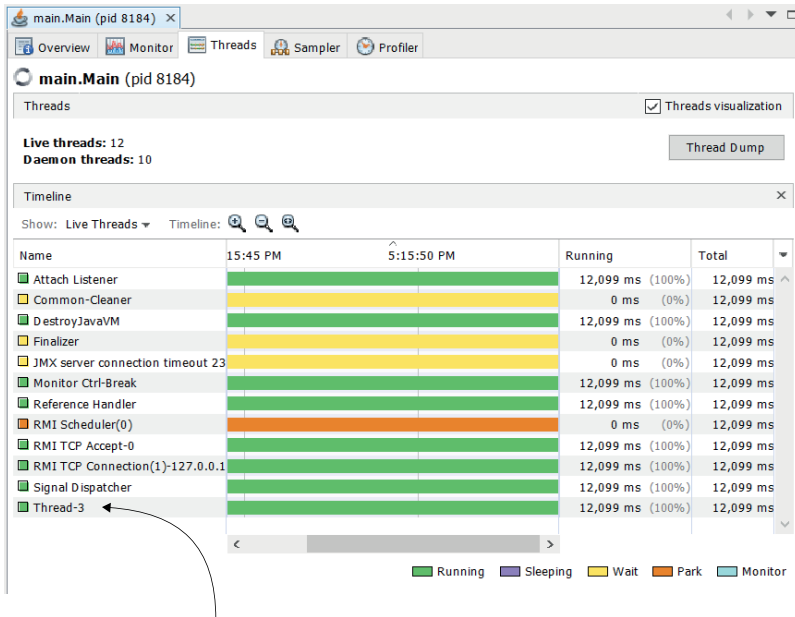


Даже если кажется, что приложение ничего не делает, созданные им четыре потока непрерывно выполняются. Такие выполняющиеся потоки, которые ничего не делают, но продолжают существовать, называются потоками-зомби. Единственное, что они делают, – потребляют ресурсы ЦП.

Рис. 6.6. На вкладке **Threads** показано визуальное представление существующих потоков и их состоянии. В виджете представлены все потоки процесса, включая созданные JVM. Это помогает с легкостью определить, каким потокам необходимо уделить особое внимание, и в итоге выполнить более глубокий анализ с использованием дампа потока

В рассматриваемом здесь примере мы сосредоточились только на обнаружении проблемы потребления ресурсов. Следующий шаг – использование дампа потока, чтобы обнаружить точную причину возникновения проблемы при параллельном выполнении. Все подробности о дампах потоков вы узнаете в главе 7, но сейчас наше внимание остается сосредоточенным на обнаружении проблем потребления ресурсов. Мы выполним те же проверки в нормальном (исправленном) приложении и сравним с дефектным. Теперь вы знаете, как сразу же отличить корректное поведение приложения от некорректного.

Пример в проекте `da-ch6-ex2` – это исправленная версия того же приложения, которое мы рассматривали выше. Я добавил некоторые блоки синхронизации, чтобы избежать одновременного доступа потоков и устранить проблемы, связанные с состоянием гонки. Я использовал экземпляр списка `list` как монитор потоков для синхронизированных блоков кода для потребителей и производителей.



В рассматриваемом здесь примере продолжает существовать только один поток, он стал потоком-зомби. В других потоках сгенерированы исключения, причиной которых является состояние гонки, и эти потоки остановлены.

Рис. 6.7. Если во время выполнения приложения возникает исключение, то некоторые потоки могут быть остановлены. Здесь показан случай, в котором одновременный доступ привел к генерации исключений в трех из четырех потоков, и в результате они были остановлены. Продолжает существовать только один поток. Следует помнить, что проблемы при параллельном выполнении в многопоточных приложениях могут приводить к различным неожиданным результатам

Листинг 6.4. Синхронизированный доступ для потребителя

```
public class Consumer extends Thread {

    private Logger log = Logger.getLogger(Consumer.class.getName());

    public Consumer(String name) {
        super(name);
    }

    @Override
    public void run() {
        while (true) {
            synchronized (Main.list) {
                if (Main.list.size() > 0) {
                    int x = Main.list.get(0);
```

```

        Main.list.remove(0);
        log.info("Consumer " +
            Thread.currentThread().getName() +
            " removed value " + x);
    }
}
}
}
}
}

```

- ❶ Синхронизирует доступ к списку, используя экземпляр `list` как монитор потоков.

В листинге 6.5 показана синхронизация, применяемая к классу `Producer`.

Листинг 6.5. Синхронизированный доступ для производителя

```

public class Producer extends Thread {

    private Logger log = Logger.getLogger(Producer.class.getName());

    public Producer(String name) {
        super(name);
    }

    @Override
    public void run() {
        Random r = new Random();
        while (true) {
            synchronized (Main.list) {
                if (Main.list.size() < 100) {
                    int x = r.nextInt();
                    Main.list.add(x);
                    log.info("Producer " +
                        Thread.currentThread().getName() +
                        " added value " + x);
                }
            }
        }
    }
}

```

- ❶ Синхронизирует доступ к списку, используя экземпляр `list` как монитор потоков.

Кроме того, я присвоил специальные имена каждому потоку. Рекомендую всегда делать это. Вы обратили внимание на имена, которые по умолчанию были присвоены JVM потокам в предыдущем примере? Обычно `Thread-0`, `Thread-1`, `Thread-2` и т. д. – это не те имена, которые позволяют

с легкостью идентифицировать конкретный поток. Я предпочитаю присваивать потокам специальные имена, когда это возможно, чтобы быстро идентифицировать их. Кроме того, такие имена начинаются с символа подчеркивания, чтобы проще было сортировать их. Сначала я определил конструктор в классах `Consumer` и `Producer` (см. листинги 6.4 и 6.5 соответственно) и использовал конструктор `super()` для присваивания имен потокам. Затем я присвоил им имена, как показано в листинге 6.6.

Листинг 6.6. Присваивание специальных имен создаваемым потокам

```
public class Main {

    public static List<Integer> list = new ArrayList<>();

    public static void main(String[] args) {
        new Producer("_Producer 1").start();
        new Producer("_Producer 2").start();
        new Consumer("_Consumer 1").start();
        new Consumer("_Consumer 2").start();
    }
}
```

Обратите внимание: после запуска приложения в консоли непрерывно выводятся журнальные сообщения. Приложение не останавливается, как это было в примере `da-ch6-ex1`. Используем `VisualVM` для наблюдения за потреблением ресурсов. В виджете использования ЦП можно видеть, что приложение потребляет меньше ресурсов ЦП, а виджет использования памяти показывает, что во время выполнения приложение работает с выделенной памятью. Мы также можем наблюдать за работой GC. Как вы узнаете немного позже в этой главе, в правой части графика потребления памяти наблюдаются впадины – это результат операций GC.

На вкладке **Threads** показано, что монитор иногда блокирует потоки, позволяя только одному потоку в определенный момент времени пройти через синхронизированный блок. Потоки не выполняются непрерывно, и поэтому приложение потребляет меньше ресурсов ЦП, как показано на рис. 6.8. На рис. 6.9 отображено визуальное представление потоков на вкладке **Threads**.

ПРИМЕЧАНИЕ. Даже если мы добавили синхронизированные блоки, некоторый код остается за пределами этих блоков. Поэтому потоки могут по-прежнему выглядеть выполняющимися в параллельном режиме (как показано на рис. 6.9).

Приложение с корректным поведением потребляет меньше ресурсов ЦП.

Приложение потребляет память, и это подтверждает, что оно выполняет какую-то работу.

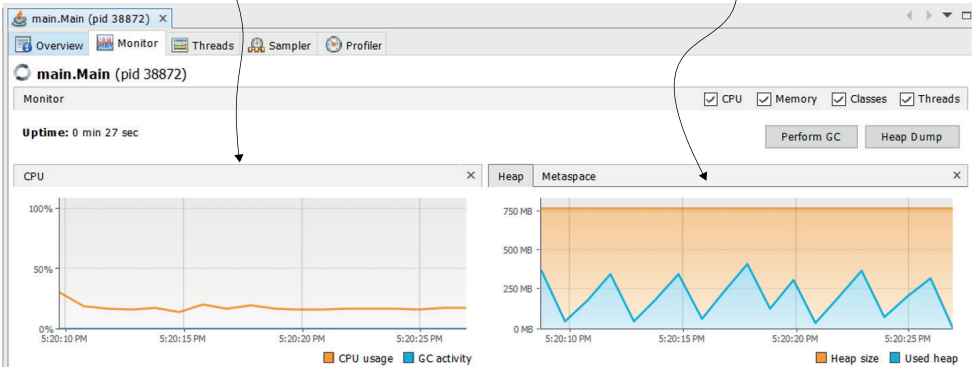


Рис. 6.8. После правильной синхронизации кода виджеты потребления ресурсов выглядят по-другому. Потребление ресурсов ЦП снизилось, а приложение использует некоторую память

Теперь потоки не выполняются непрерывно. Профилировщик показывает, когда потоки блокируются монитором, находятся в состоянии ожидания или засыпают.

Инструкции, оставшиеся за пределами синхронизированных блоков, могут заставлять потоки выполняться параллельно. Обратите внимание на те места, где два потока-производителя одновременно обозначены на диаграмме менее ярким цветом (как неактивные).

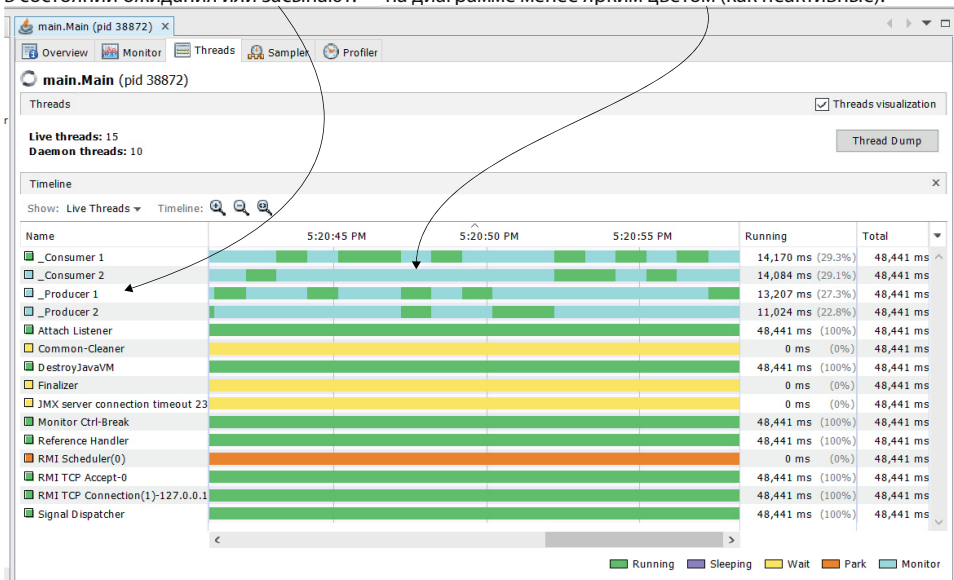
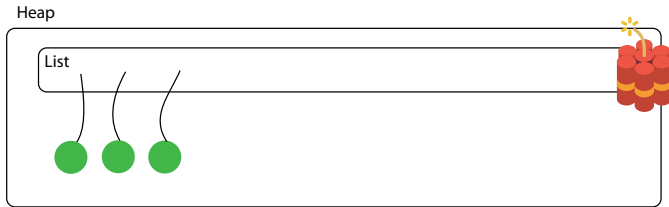


Рис. 6.9. Вкладка **Threads** помогает получить визуальное представление выполнения потоков в приложении. Поскольку имена потоков начинаются с символа подчеркивания, можно с легкостью сортировать их по имени, чтобы сгруппировать нужные. Обратите внимание: выполнение потоков время от времени прерывается монитором, который позволяет только одному потоку в определенный интервал времени проходить через синхронизированные блоки кода

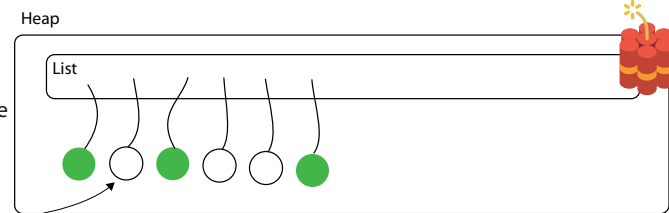
6.2.3. Обнаружение утечек памяти

В этом подразделе мы рассмотрим утечки памяти и способы, позволяющие определить их существование в приложении. Утечка памяти (memory leak) возникает, когда приложение продолжает сохранять ссылки на ненужные объекты (см. рис. 6.10). Из-за существования таких ссылок GC (сборщик мусора, механизм, отвечающий за удаление ненужных данных из памяти приложения) не может удалить эти объекты. По мере того как приложение продолжает добавлять новые данные, память заполняется. Когда в приложении нет достаточного объема памяти для добавления новых данных, оно генерирует исключение `OutOfMemoryError` и прекращает выполнение (останавливается). Мы будем использовать простое приложение, в котором генерируется исключение `OutOfMemoryError`, чтобы продемонстрировать, как обнаружить утечку памяти, применяя VisualVM.

1. Предположим, что существует приложение, создающее экземпляры объекта и сохраняющее ссылки на эти экземпляры в списке.



2. Приложение продолжает создавать новые экземпляры. Некоторые из ранее созданных экземпляров уже не нужны, но приложение не удаляет ссылки на них из списка.



3. Поскольку приложение сохраняет все ссылки, GC не может удалить ненужные объекты из памяти. Память переполняется, и в какой-то момент приложение не может разместить новые объекты. Процесс останавливается, и приложение завершается аварийно с ошибкой `OutOfMemoryError`.

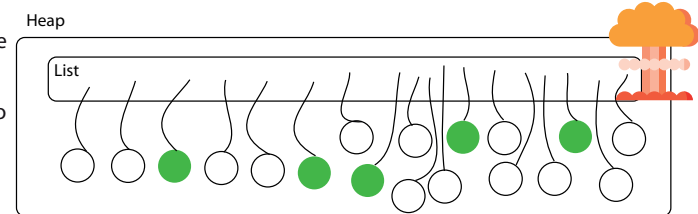


Рис. 6.10. Исключение `OutOfMemoryError` похоже на бомбу замедленного действия. Приложение не удаляет ссылки на объекты, которые уже не используются. GC не может удалить соответствующие экземпляры из памяти, потому что приложение сохраняет ссылки на них. Создается все больше объектов, и в конце концов память заполняется. В некоторый момент в куче уже нет места для размещения других объектов, и приложение «падает» с ошибкой `OutOfMemoryError`.

В примере из проекта `da-ch6-ex3` вы найдете простое приложение, сохраняющее случайные экземпляры в списке, но никогда не удаляющее ссылки на них. В листинге 6.7 приведено пример простой реализации, в которой возникает исключение `OutOfMemoryError`.

Листинг 6.7. Генерация исключения `OutOfMemoryError`

```
public class Main {

    public static List<Cat> list = new ArrayList<>();

    public static void main(String[] args) {
        while(true) {
            list.add(new Cat(new Random().nextInt(10)));    ❶
        }
    }
}
```

- ❶ Непрерывно добавляет новые экземпляры в список до тех пор, пока в JVM не закончится память.

Класс `Cat` – это простой объект Java, как показано в приведенном ниже фрагменте кода:

```
public class Cat {

    private int age;

    public Cat(int age) {
        this.age = age;
    }

    // Здесь не показаны get- и set-методы.
}
```

Выполним это приложение и понаблюдаем за потреблением ресурсов с помощью VisualVM. Нас особенно интересует виджет, показывающий потребление памяти. Если утечка памяти воздействует на приложение, то этот виджет может подтвердить, что объем используемой памяти непрерывно увеличивается. GC пытается удалить ненужные данные из памяти, но удаляет слишком мало. В конце концов память заполняется до отказа, приложение не может сохранить новые данные и генерирует исключение `OutOfMemoryError` (см. рис. 6.11).

Обратите внимание: объем используемой памяти непрерывно увеличивается. GC пытается освободить память, но не может удалить большинство экземпляров, так как приложение продолжает хранить в памяти ссылки на них.

Когда вся выделенная память заполняется, и приложение уже не может сохранить новые данные, генерируется исключение `OutOfMemoryError`.

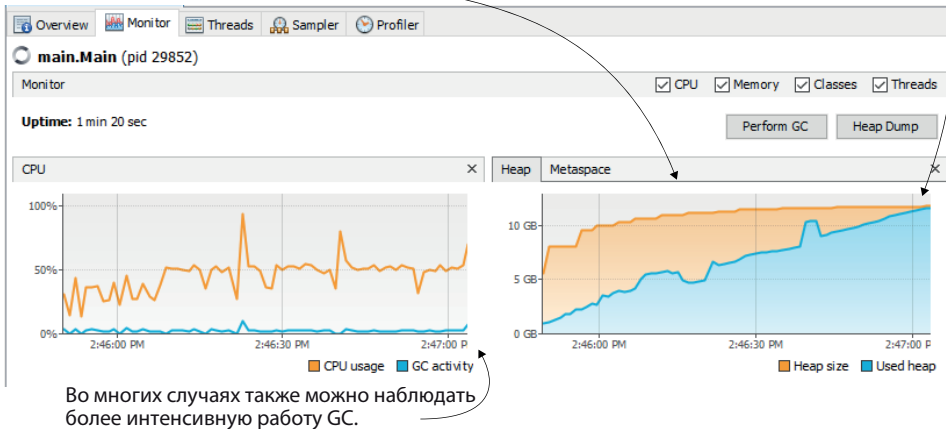


Рис. 6.11. Если утечка памяти воздействует на приложение, то объем используемой памяти постоянно увеличивается. GC пытается освободить память, но не может удалить достаточное количество данных. Объем используемой памяти возрастает до того момента, когда приложение уже не может разместить новые данные. В этой точке приложение генерирует исключение `OutOfMemoryError` и прекращает выполнение. Во многих случаях утечка памяти также приводит к более интенсивной работе GC, что можно видеть в виджете использования ресурсов ЦП

Если позволить приложению работать достаточно долго, то в итоге вы увидите трассировку стека ошибок (исключений) в консоли приложения:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.base/java.util.Arrays.copyOf(Arrays.java:3689)
    at java.base/java.util.ArrayList.grow(ArrayList.java:238)
    at java.base/java.util.ArrayList.grow(ArrayList.java:243)
    at java.base/java.util.ArrayList.add(ArrayList.java:486)
    at java.base/java.util.ArrayList.add(ArrayList.java:499)
    at main.Main.main(Main.java:13)
```

Важно помнить о том, что трассировка стека `OutOfMemoryError` не всегда показывает место, в котором возникла проблема. Поскольку приложение имеет только одну локацию динамической памяти (кучи), конкретный поток может создать проблему, а другой поток может оказаться настолько невезучим, что при попытке использования локации динамической памяти возникает такая ошибка. Единственным надежным способом определения главной причины является использование дампа динамической памяти (кучи), который будет рассматриваться в главе 11.

На рис. 6.12 сравнивается обычное поведение и поведение приложения с утечкой памяти, как показано в VisualVM. Для приложения с нормальным выполнением (без утечки памяти) следует отметить, что на графике имеются пики и впадины. Приложение выделяет память и заполняет ее данными (пики), а время от времени GC удаляет данные, ставшие ненужными (впадины). Эти «приливы и отливы» обычно являются хорошим признаком того, что на анализируемую функциональность не воздействует утечка памяти.

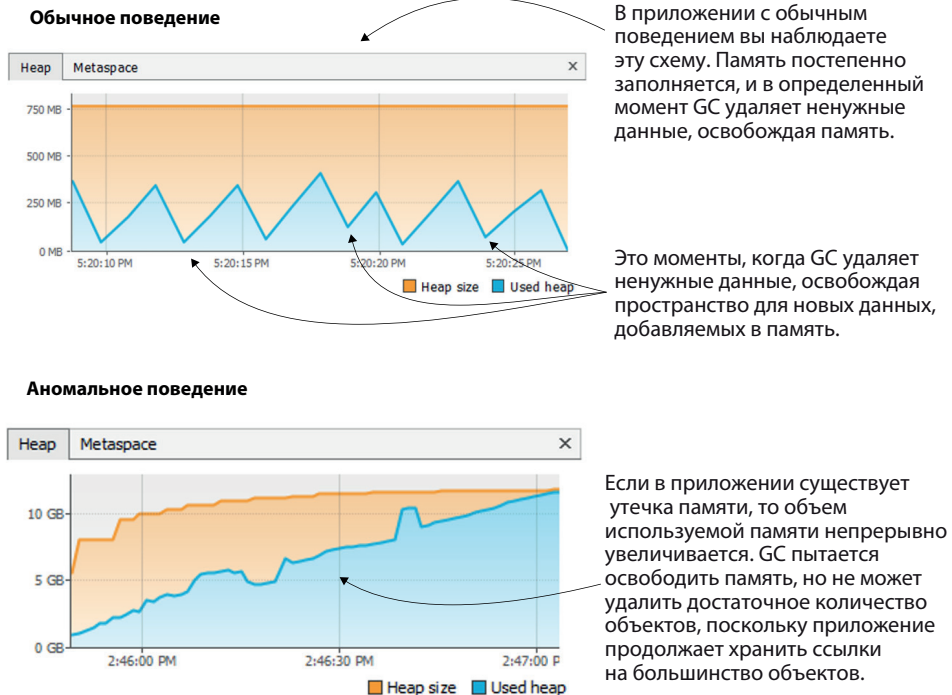


Рис. 6.12. Сравнение использования памяти в нормальном приложении и приложении с утечкой памяти. GC управляет удалением ненужных данных из памяти в нормальном приложении, и выделенное пространство памяти никогда не заполняется до отказа. Приложение с утечкой памяти не позволяет GC удалить достаточное количество данных. В некоторый момент память заполняется, и генерируется исключение `OutOfMemoryError`

Но если вы видите, что память постоянно заполняется, а GC ее не очищает, то приложение, вероятно, содержит утечку памяти. После предположения о наличии утечки памяти необходимо продолжить анализ с использованием дампа динамической памяти (кучи).

Вы можете управлять размером выделяемой динамической памяти в Java-приложении. Таким образом, можно увеличить максимальное предельное значение размера памяти, которую JVM выделяет приложению. Но увеличение памяти для приложения не является решением проблемы уте-

чек памяти. Тем не менее такой подход может стать временным решением, предоставляющим больше времени для выявления главной причины возникновения этой проблемы. Чтобы установить максимальный размер кучи для приложения, используйте ключ JVM `-Xmx`, после которого укажите необходимый выделяемый объем памяти (например, `-Xmx1G` определяет максимальный размер кучи 1 Гб). Также можно задать минимальный начальный размер кучи с помощью ключа `-Xms` (например, `-Xms500m` определяет минимальный размер кучи 500 Мб).

Кроме обычного пространства динамической памяти (кучи) каждое приложение использует метапространство (metaspace): локацию памяти, в которой JVM хранит метаданные класса, необходимые для выполнения приложения. В VisualVM можно наблюдать распределение метапространства также в виджете использования памяти. Для оценки распределения метапространства используйте вкладку **Metaspace** этого виджета, как показано на рис. 6.13.

Вкладка **Metaspace** виджета использования памяти показывает размер метапространства и степень его использования.

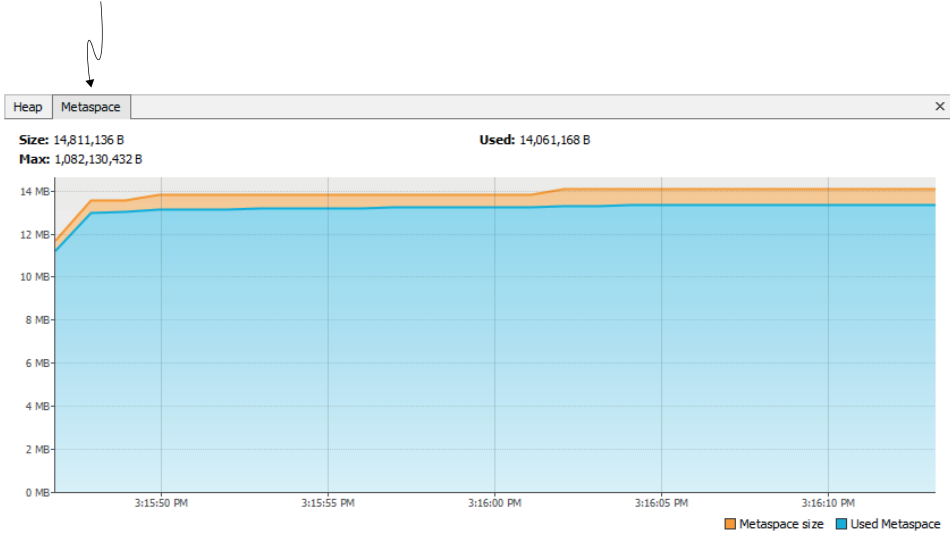


Рис. 6.13. Метапространство – это часть памяти, используемая для хранения метаданных класса. В особых случаях метапространство может быть переполнено.

Виджет использования памяти VisualVM также показывает использование метапространства

Исключение `OutOfMemoryError` в пространстве метаданных возникает не так часто, но это возможно. Недавно я имел дело с таким случаем в приложении, которое некорректно использовало рабочую среду для поддержки постоянного хранения (персистентности) данных. Обычно рабочие среды и библиотеки, использующие отражение (reflection) Java, с большой веро-

ятностью создают подобные проблемы, если используются некорректно, поскольку часто основаны на динамических прокси-объектах и косвенных вызовах.

В моем случае приложение некорректно использовало фреймворк Hibernate. Я не удивлюсь, если вы уже слышали о Hibernate, так как в настоящее время это одно из наиболее часто используемых решений для управления персистентными (постоянно хранимыми) данными в Java-приложениях. Hibernate – превосходное инструментальное средство, помогающее реализовать наиболее часто применяемые функции персистентности в приложении, исключая необходимость написания лишнего кода. Hibernate управляет контекстом экземпляров и отображает изменения в этом контексте в базу данных. Но этот инструмент не рекомендуется применять для слишком больших контекстов. Другими словами, не следует одновременно работать со слишком большим количеством записей из базы данных.

В этом приложении я обнаружил проблему при определении процесса-планировщика, загружающего множество записей из базы данных и обрабатывающего их определенным образом. Казалось, что в некоторый момент количество записей, извлеченных этим процессом для обработки, становилось настолько большим, что выполнение самой операции загрузки приводило к заполнению метапространства, т. е. проблема заключалась в некорректном использовании фреймворка, а не в его внутренней ошибке. Разработчики вынуждены были отказаться от использования Hibernate, а вместо него воспользовались другим решением более низкого уровня, подобного JDBC.

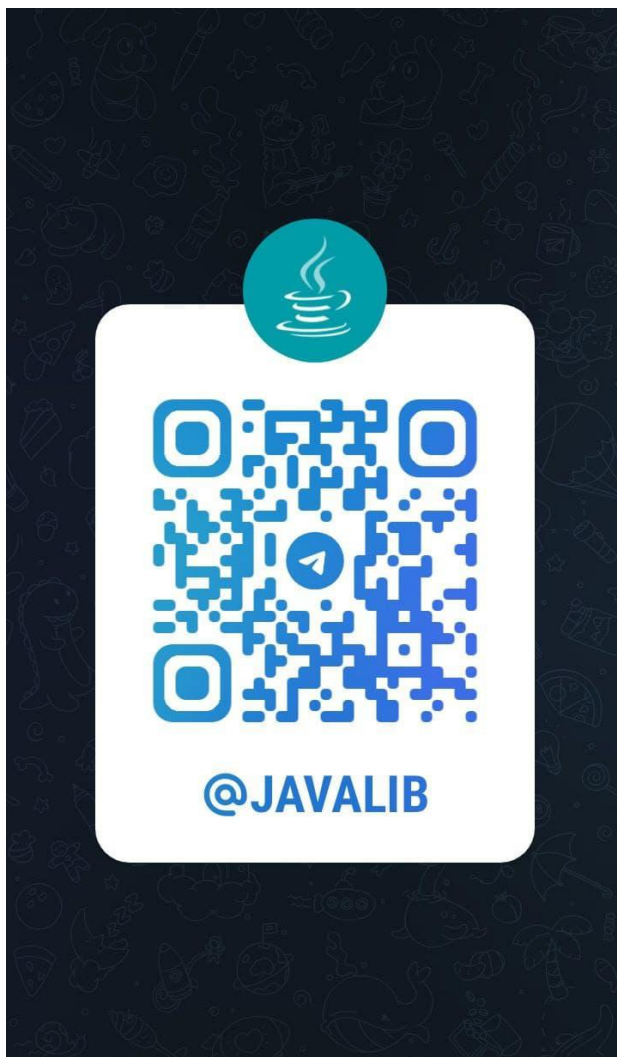
Эта проблема была чрезвычайно острой, и я должен был найти оперативное временное решение, поскольку полный рефакторинг кода занял бы слишком много времени. Как и для кучи, для метапространства также можно настроить размер. Используя ключ `-XX:MaxMetaspaceSize`, можно увеличить метапространство (например, `-XX:MaxMetaspaceSize=100M`), но при этом следует помнить, что такой способ не является настоящим решением проблемы. Постоянным решением для такого случая является рефакторинг функциональности, чтобы избежать одновременной загрузки слишком большого количества записей в память и в конце концов использовать другую методику поддержки персистентности, если это необходимо.

6.3. Резюме

- Профилировщик – это инструментальное средство, позволяющее наблюдать за выполнением приложения, чтобы определить причины возникновения конкретных проблем, которые более сложно обнаружить каким-либо другим способом. Профилировщик показывает:
 - как приложение использует системные ресурсы, такие как ЦП и память;

- какой код выполняется (сейчас) и продолжительность выполнения каждого метода;
 - стек выполнения методов в различных потоках;
 - выполняющиеся потоки и их состояние.
- Профилировщик предоставляет удобные визуальные виджеты, которые помогают быстро понять конкретные аспекты.
 - Используя профилировщик, можно наблюдать выполнение работы GC (сборщика мусора), что помогает обнаружить проблемы, например некорректное удаление из памяти приложения неиспользуемых данных (утечки памяти).

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javallib>



Глава 7

Поиск скрытых проблем с использованием методик профилирования

Темы:

- выборка выполняемого кода приложения для поиска методов, выполняемых в текущий момент;
- наблюдение за временами выполнения;
- идентификация SQL-запросов, выполняемых приложением.

В главе 6 я назвал профилировщик мощным инструментальным средством, которое может указать путь, когда погаснут все остальные огни. Но то, что мы обсуждали ранее, – это лишь небольшая часть функциональных возможностей профилировщика. Он предоставляет мощные средства анализа выполнения приложения, и изучение правильного применения этих средств может помочь во многих сценариях.

Во многих случаях я вынужден был оценивать или анализировать выполнение приложений из кодовых баз, читаемых с трудом, – старых приложений с плохо сформированными проектными решениями кода, которые некоторые компании продолжают хранить в своих «сундуках». В подобных ситуациях профилировщик становился единственным эффективным инструментом поиска того, что именно выполняется, когда активизируется конкретная функциональная возможность. Теперь вы понимаете, почему я сравнил профилировщик со светом звезды Эarendил: по словам Галадриэль, это действительно был луч света во многих темных местах, где гасли все другие огни.

В этой главе мы подробно рассмотрим три методики анализа, выполняемые посредством профилирования, которые я считаю чрезвычайно полезными:

- выборку для определения части кода приложения, выполняемой в текущий интервал времени;
- профилирование выполнения (также называемое инструментровкой (instrumentation)) для выявления некорректного поведения и оптимизации;
- профилирование приложения для идентификации SQL-запросов, используемых для обмена данными с системой управления базой данных (СУБД).

Мы продолжим обсуждение этих тем в главе 8 с привлечением более продвинутых методик визуализации выполнения приложения. При правильном применении эти методики позволят сэкономить огромное количество времени при поиске причин возникновения разнообразных проблем. К сожалению, даже несмотря на мощные возможности таких методик, многие разработчики не знакомы с ними. Некоторые программисты знают об их существовании, но склонны считать, что использовать их слишком сложно (в этой главе я покажу, что это неверное мнение). Следовательно, они пытаются использовать другие методы для решения проблем, которые можно было бы устранить гораздо более эффективно с помощью профилировщика (как показано в этой главе).

Чтобы убедиться в том, что вы правильно понимаете, как применять эти методики и какие проблемы можно анализировать, я создал четыре небольших проекта. Мы будем использовать эти проекты для практического применения изучаемых методик. В разделе 7.1 рассматривается выборка (sampling) – методика, применяемая для определения того, какой код выполняется в конкретный интервал времени. В разделе 7.2 вы узнаете, как профилировщик может предоставить больше подробностей о выполнении, чем простая выборка. В разделе 7.3 обсуждается использование профилировщика для получения подробностей об SQL-запросах, отправляемых приложением в СУБД.

7.1. Выборка для наблюдения за выполняемым кодом

Что такое выборка и как ее можно применить с пользой? Выборка (sampling) – это методика, в которой вы используете профилировщик, чтобы определить, какой код выполняет приложение в конкретный интервал времени. Выборка предоставляет немного подробностей о выполнении, но позволяет сформировать общую картину того, что происходит, снабжая вас полезной информацией, необходимой для дальнейшего анализа. Поэтому выборка всегда должна являться первым этапом профилирования приложения, и, как вы скоро увидите, во многих случаях выполнения выборки даже может оказаться достаточно для анализа. Для этого раздела я подготовил проект da-ch7-ex1. Мы будем использовать профилировщик для

выполнения выборки в этом приложении, чтобы понять, как можно воспользоваться VisualVM для обнаружения проблем, связанных с временем выполнения конкретного функционального компонента.

Проект для демонстрации выполнения выборки представляет собой весьма небольшое приложение, предъявляющее конечную точку */demo*. Если кто-то вызывает эту конечную точку с использованием cURL, Postman или аналогичного средства, то далее приложение вызывает конечную точку, предъявленную **httpbin.org**.

Для многих примеров и демонстрационных приложений я предпочитаю использовать **httpbin.org** – веб-приложение с открытым исходным кодом и инструментальное средство, написанное на Python, которое предоставляет имитационные конечные точки, используемые для тестирования различных функциональных свойств при реализации приложений.

В рассматриваемом здесь примере мы вызываем конечную точку, где **httpbin.org** отвечает с определенной задержкой. Для этого примера используем пятисекундную задержку, чтобы имитировать сценарий со временем ожидания ответа в приложении, а **httpbin.org** имитирует главную причину возникновения этой проблемы.

Кроме того, этот сценарий представлен визуально на рис. 7.1.

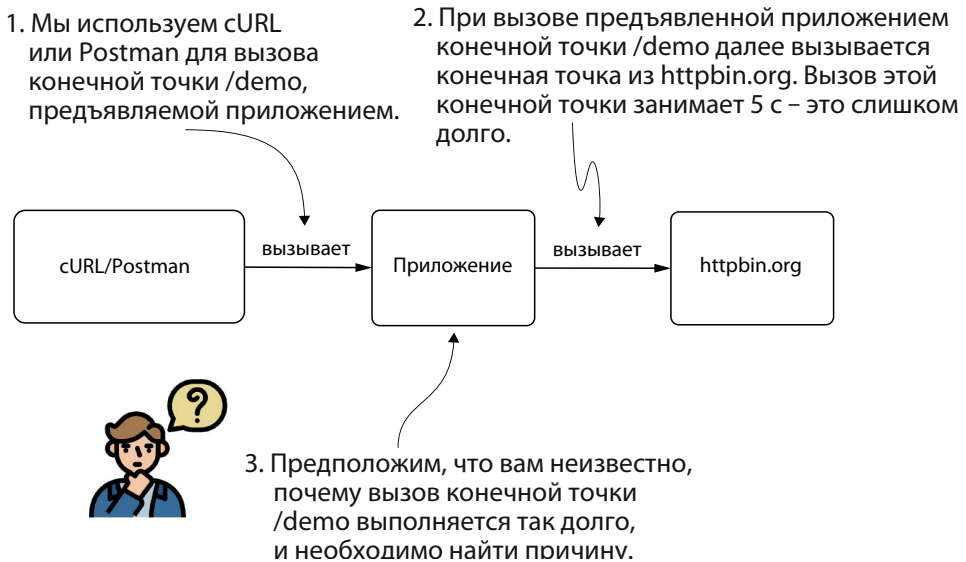


Рис. 7.1. Анализируемое приложение предъявляет конечную точку */demo*. При вызове этой конечной точки вы должны ждать ответа приложения в течение 5 с. Необходимо понять, почему конечная точка отвечает с такой длительной задержкой. Мы знаем, что приложение вызывает имитационную конечную точку из **httpbin.org**, создающую задержку, но необходимо узнать, как проанализировать этот сценарий с помощью профилировщика. Таким образом, вы научитесь применять аналогичные методики в реальных ситуациях



ПРИМЕЧАНИЕ. Используя время ожидания ответа, мы понимаем, каким образом приложение реагирует медленнее, чем ожидалось.

Методика профилирования включает два этапа:

- выборку для обнаружения выполняемого (в интересующий нас интервал времени) кода и определения места, где следует извлечь более подробную информацию (этот подход мы рассмотрим в текущем разделе);
- профилирование (также называемое инструментровкой (instrumentation)) для получения более подробной информации о выполнении конкретных частей кода.

Иногда первого этапа (выборки) достаточно для понимания проблемы, и, возможно, профилирования приложения (этап 2) не потребуется. Как вы узнаете из этой главы и из глав 8–10, профилирование может предоставить больше подробностей о выполнении, если это необходимо. Но сначала нужно узнать, какую часть кода следует профилировать, а для этого и применяется выборка.

Как возникает проблема в рассматриваемом здесь примере? Мы вызываем конечную точку */demo*, выполнение занимает 5 с (см. рис. 7.2), и мы считаем, что это слишком долго. В идеальном случае предполагается, что для выполнения требуется менее 1 с, поэтому необходимо понять, почему так долго происходит обращение к конечной точке */demo*. Что является причиной задержки? Причина находится в приложении или где-то еще?

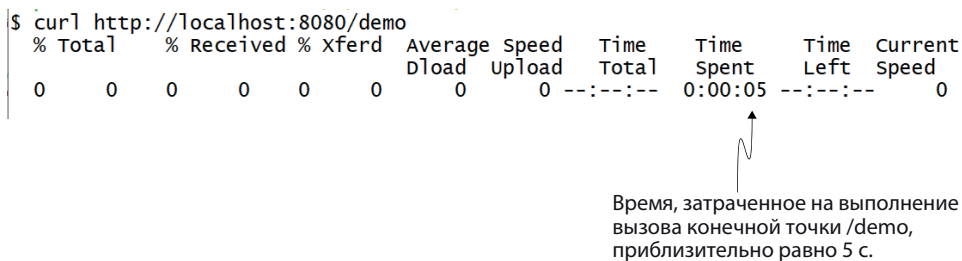


Рис. 7.2. При вызове конечной точки (здесь показано использование cURL) приложению требуется 5 с для ответа. В этом сценарии мы используем профилировщик для анализа проблемы с задержкой

При анализе проблем замедления в неизвестной кодовой базе использование профилировщика должно являться самым первым вариантом выбора. Такая проблема не всегда напрямую связана с конечной точкой. В рассматриваемом здесь примере конечная точка стала самым простым

решением. Но в любой ситуации, в которой возникает замедление, – вызов конечной точки, выполнение процесса или использование простого вызова метода при наступлении конкретного события, – профилировщик должен быть самым первым инструментом.

Сначала запустим приложение, потом VisualVM (профилировщик, который будет применяться для анализа). Не забудьте добавить VM-ключ `-Djava.rmi.server.hostname=localhost`, описанный в главе 6. Он позволяет VisualVM установить соединение с процессом. Выберите процесс из списка в левой панели, затем перейдите на вкладку **Sampler**, как показано на рис. 7.3, чтобы начать выборку выполнения.

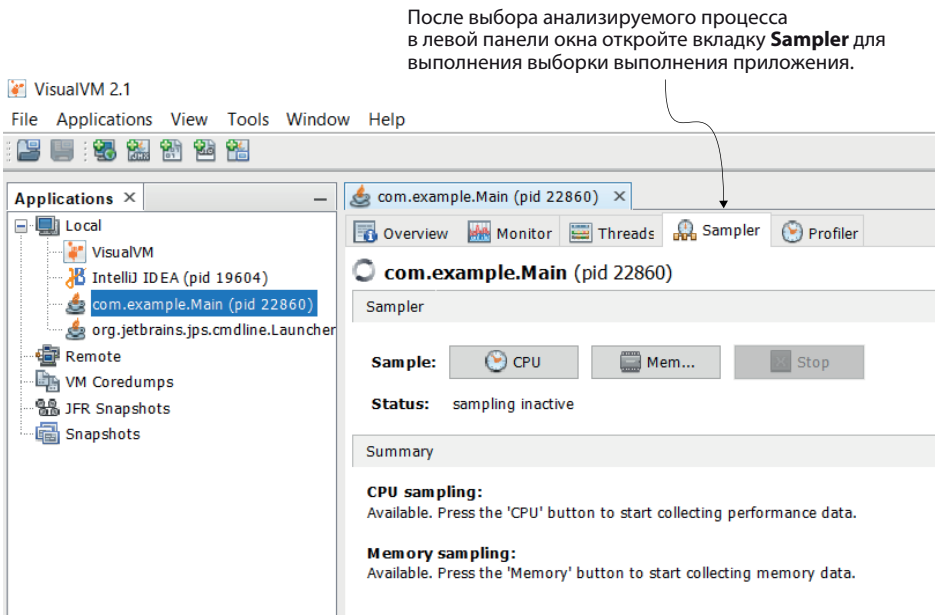


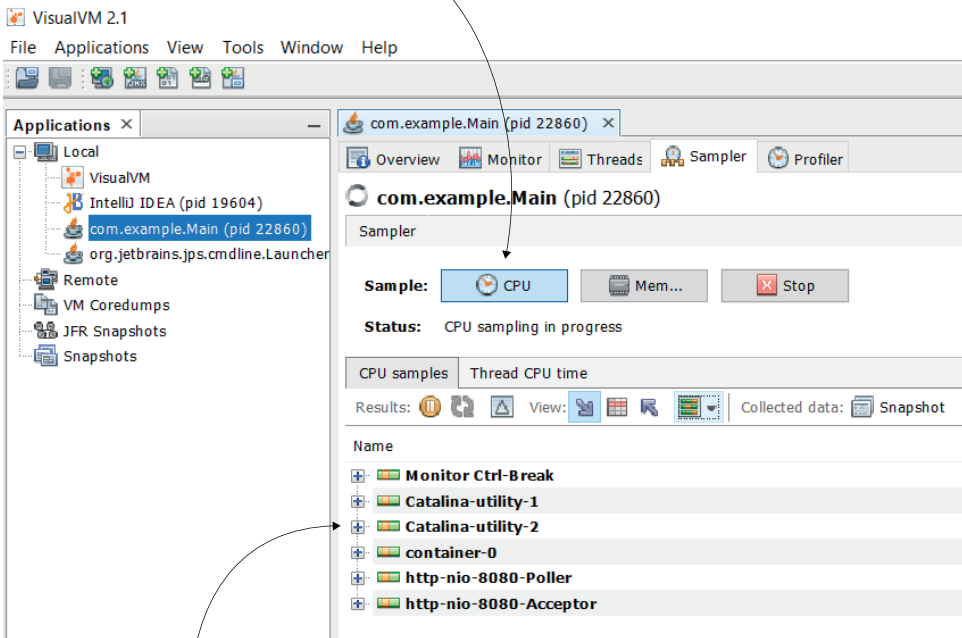
Рис. 7.3. Чтобы начать выборку выполнения, выберите процесс из списка в левой панели, затем перейдите на вкладку **Sampler**

При выполнении выборки выполнения существуют три цели:

- поиск кода, выполняемого в определенный интервал времени – выборка показывает, что именно выполняется незаметно для пользователя, и это превосходный способ найти ту часть кода приложения, которую необходимо проанализировать;
- определение потребления ЦП – мы будем использовать эту характеристику для анализа проблем с задержками и обнаружения методов, совместно использующих время выполнения;
- определение потребления памяти – это позволяет проанализировать проблемы, связанные с памятью. Более подробно выборка и профилирование памяти рассматривается в главе 11.

Щелкните по кнопке **CPU** (как показано на рис. 7.4), чтобы начать выборку данных о производительности. VisualVM выводит список всех активных потоков и трассировки их стеков. Далее профилировщик перехватывает выполнение процесса и показывает все вызванные методы и приблизительное время их выполнения. При вызове конечной точки */demo* профилировщик показывает, что происходит незаметно для пользователя, когда приложение выполняет этот функциональный компонент.

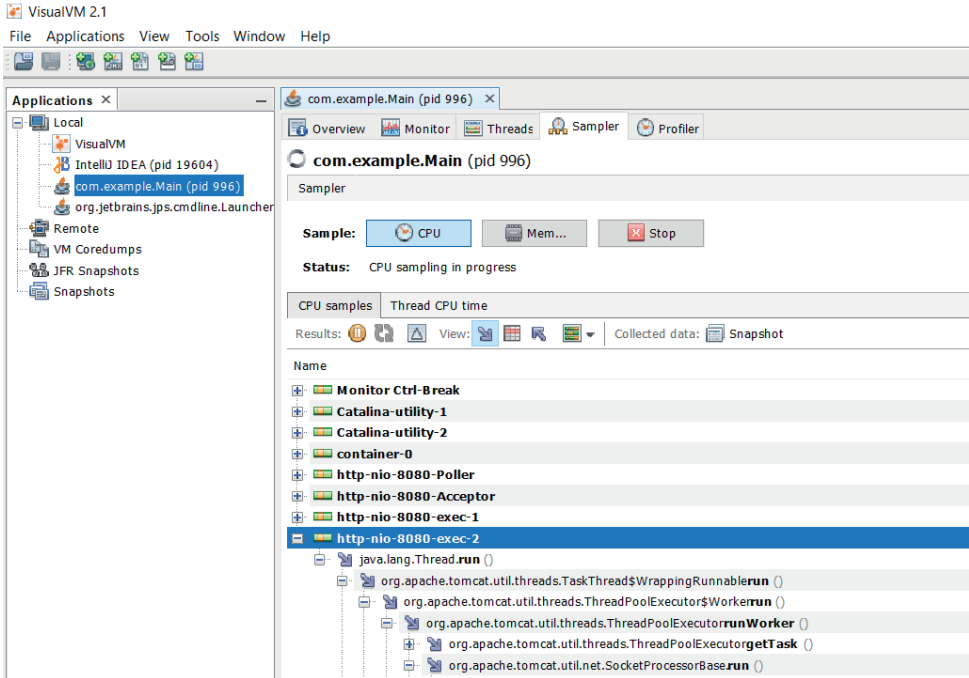
Щелкните по кнопке **CPU**, и VisualVM начнет перехват выполняющихся потоков.



Выполняющиеся потоки появляются в этом списке. Используя маленькую кнопку +, вы можете более подробно рассмотреть стек выполнения для каждого потока.

Рис. 7.4. Профилировщик показывает все активные потоки в списке. Каждый пункт можно развернуть, чтобы увидеть стек выполнения и приблизительное время выполнения. Во время работы приложения новые создаваемые потоки появляются в этом списке, и вы можете анализировать их выполнение

Теперь можно вызвать конечную точку */demo* и наблюдать, что происходит. Как показано на рис. 7.5, в списке появляется несколько новых потоков. Приложение активизирует эти потоки, когда мы вызываем конечную точку */demo*. Если открыть новые потоки, то вы должны увидеть в точности то, что приложение делает во время их выполнения.



VisualVM открывает полную трассировку стека выполняемого приложения, когда вызывается конечная точка /demo. Эту трассировку стека можно использовать, чтобы определить, какой именно код выполняет приложение и какая инструкция затрачивает на выполнение больше времени.

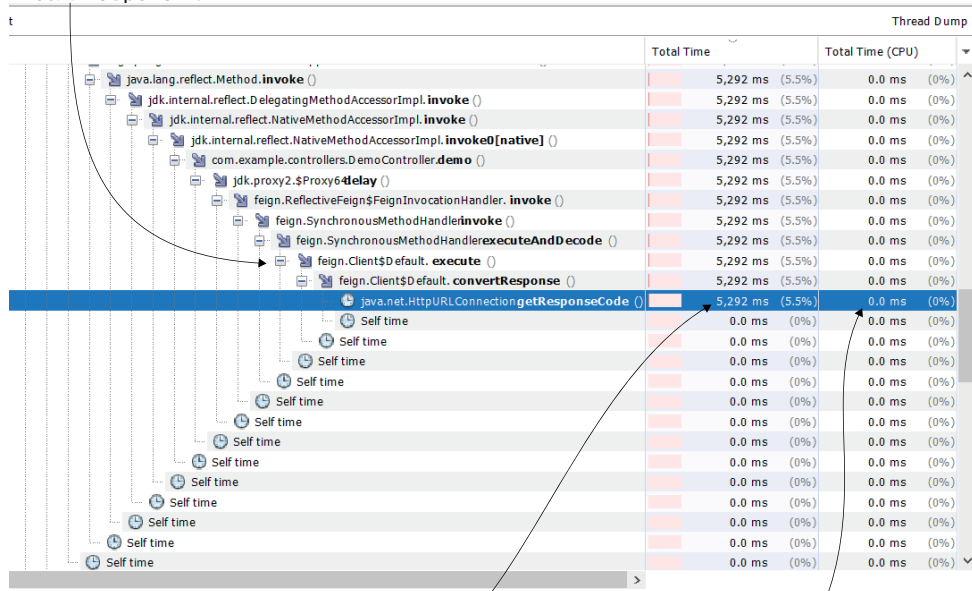
Рис. 7.5. Трассировка стека показывает, что именно выполняет приложение. Здесь можно видеть каждый метод и каждый последующий вызываемый метод. Такое визуальное представление помогает быстро найти код, на котором необходимо сосредоточить внимание при анализе конкретного функционального компонента

Прежде чем приступить к рассмотрению подробностей, таких как время выполнения, необходимо особенно подчеркнуть, насколько важен описанный здесь первый этап. При анализе кода я многократно использовал только выборку, чтобы определить, с чего начать поиск проблемы. Возможно, даже не нужен был анализ производительности или проблем с задержками, а просто требовалось найти точку начала отладки. Помните, о чем было сказано в главах 2–4: для отладки чего-либо вы должны знать, где добавить точку останова, чтобы приостановить выполнение приложения. Если нет предположений о том, где добавить точку останова, то вы не сможете начать отладку. Выборка может оказаться способом прояснения ситуации, в которой вы не можете понять, где начать отладку (особенно в случаях, подобных описанному в начале главы, когда в приложении отсутствует ясное проектное решение кода).

Рассмотрим стек выполнения, чтобы понять, что нам показывает профилировщик. Если нужно определить, какой именно код выполняется, вы

просто раскрываете трассировку стека до точки, где показаны интересные вас методы приложения. При анализе проблемы с задержкой (как в рассматриваемом здесь примере) можно развернуть трассировку стека, чтобы наблюдать максимальное время выполнения, как показано на рис. 7.6.

VisualVM открывает полную трассировку стека выполняемого приложения, когда вызывается конечная точка /demo. Эту трассировку стека можно использовать, чтобы определить, какой именно код выполняет приложение и какая инструкция затрачивает на выполнение больше времени.



Этот инструмент показывает общее время, затраченное на вызов каждого метода. Эту информацию можно использовать для определения главных причин замедления работы приложения. В данном случае метод `getResponseCode()` из класса `URLConnection` потратил все время выполнения.

Следует обратить внимание на другую весьма важную подробность: время использования ЦП равно нулю. Это означает, что приложение потратило 5 с общего времени выполнения на ожидание чего-то, а не на какую-то работу.

Рис. 7.6. Если развернуть стек выполнения, то вы увидите, какие методы выполняются и сколько времени они затрачивают на выполнение. Также можно логически вывести время их ожидания и реальной работы. Профилировщик показывает методы из кодовой базы приложения и методы, вызываемые из конкретных зависимостей (библиотек или фреймворков), используемых приложением

Я развернул стек выполнения, щелкнув по маленькой кнопке (+) в самом последнем методе. Профилировщик показывает для него время выполнения чуть больше 5 с, из чего понятно, какой метод стал причиной задержки. В данном конкретном случае очевидно, что только один метод является причиной замедления: `getResponseCode()` из класса `URLConnection`.



СОВЕТ. Следует помнить, что в сценариях из реальной практики не всегда существует только один метод, который расходует все время выполнения. Вы будете часто обнаруживать, что затраченное время разделяется между несколькими выполняемыми методами. В подобных случаях применяется правило: в первую очередь сосредоточить внимание на методе, который затрачивает на выполнение самое большое время.

Важным аспектом этого примера является то, что время использования ЦП (т. е. сколько времени метод действительно работает) равно нулю. Хотя метод затрачивает 5 с на выполнение, он не использует ресурсы ЦП, потому что ожидает HTTP-вызова для завершения и получения ответа. Отсюда можно сделать вывод: проблема возникает не в приложении, оно замедляется только потому, что ожидает ответ на свой HTTP-запрос.

Чрезвычайно полезно учитывать разность между общим временем использования ЦП и общим временем выполнения. Если метод расходует время ЦП, это означает, что он «действительно работает». Для увеличения производительности в этом случае обычно требуется корректировка алгоритма для минимизации его сложности (если это возможно). Если при выполнении затрачивается небольшое количество времени ЦП, но время выполнения длительное, то, вероятнее всего, метод чего-то ожидает: некоторое действие может требовать длительного времени, но само приложение ничего не делает. В этом случае необходимо определить, чего именно ждет приложение.

Еще один весьма важный аспект наблюдения – профилировщик перехватывает не только кодовую базу приложения. Вы также можете видеть методы зависимостей, вызываемых во время выполнения приложения. В рассматриваемом здесь примере приложение использует зависимость с именем OpenFeign для вызова конечной точки `httpbin.org`. Это можно заметить в трассировке стека пакетов, не принадлежащих к кодовой базе приложения. Эти пакеты являются частью зависимостей, используемых приложением для реализации своей функциональности. Одной из таких зависимостей может являться OpenFeign, как в рассматриваемом здесь примере.

OpenFeign – это проект из экосистемы технологий Spring, который Spring-приложения могут использовать для вызова REST конечных точек. Поскольку приведенный здесь пример является Spring-приложением, в трассировке стека вы обнаружите пакеты реализации технологий, связанных со Spring. Нет необходимости понимать, что делает каждая часть трассировки стека. Даже в сценарии из реальной практики вам будет ничего не известно об этом. Но в действительности эта книга о понимании кода, который вам пока еще неизвестен. Если вы хотите изучать Spring, то рекомендую начать со «Spring Start Here» (Manning, 2021 г.) – это другая написанная мною книга. В ней вы также найдете подробности об OpenFeign.

Почему наблюдение за методами зависимостей так важно? Потому что иногда почти невозможно понять, что выполняется из конкретной зависимости, использующей другие средства. Рассмотрим код, написанный в нашем примере приложения для вызова конечной точки `httpbin.org` (см. листинг 7.1). Вы не можете увидеть настоящую реализацию отправки HTTP-запроса. Причина в том, что, как и во многих современных фреймворках Java, эта зависимость использует динамические прокси для разделения реализации.

Листинг 7.1. Реализация HTTP-клиента, использующая OpenFeign

```
@FeignClient(name = "httpBin", url = "${httpBinUrl}")
public interface DemoProxy {

    @PostMapping("/delay/{n}")
    void delay(@PathVariable int n);
}
```

Динамические прокси (dynamic proxies) предоставляют приложению способ выбора реализации метода во время выполнения. Если функциональность приложения использует динамические прокси, то оно действительно может вызывать метод, объявленный через интерфейс, без знания о том, какая реализация будет взята для выполнения во время работы приложения (см. рис. 7.7). Это упрощает использование функциональных возможностей фреймворка, но недостаток заключается в том, что вам неизвестно, где следует анализировать проблему.



ПРИМЕЧАНИЕ. Один из моих личных способов использования выборки применяется при изучении нового фреймворка или библиотеки. Выборка помогает понять, что выполняется скрыто от пользователя в новой функциональности. Я применял такой подход при изучении Hibernate и Spring Security, обладающих сложными функциональными свойствами, и это помогло мне быстро понять, как работать с конкретными функциями и средствами.

7.2. Профилирование с целью узнать, сколько раз выполнен метод

Весьма важно узнать, какой код выполняется в определенный интервал времени, но иногда этого недостаточно. Часто необходимо больше подробностей, чтобы полностью понять конкретное поведение. Например, выборка не сообщает о количестве вызовов метода. Приложение может затрачивать всего лишь 50 мс на выполнение, но если оно вызывает метод

тысячу раз, то потребуется 50 с для выполнения при выборке. Чтобы показать, как получить подробную информацию о выполнении с помощью профилировщика и определить ситуации, в которых он является полезным, мы снова воспользуемся проектами, прилагаемыми к этой книге. Начнем с проекта da-ch7-ex1, который мы уже обсуждали в разделе 7.1, но на этот раз будем рассматривать профилирование для получения более подробной информации о выполнении.



Фреймворки, используемые приложением, могут предоставлять несколько реализаций для одной абстракции. Приложение решает, какую реализацию использовать во время выполнения. Из-за этого разделения более трудно определить, какой именно код будет выполняться, просто читая исходный код. Возможно, вам даже будет неизвестно, в какой зависимости искать подобные динамические реализации.

Рис. 7.7. Фреймворк хранит реализации для отдельных абстракций и предоставляет их динамически во время выполнения. Поскольку реализации разделены и приложение предоставляет их во время выполнения, более трудно найти нужную реализацию, просто читая код

Начнем с приложения, представленного в проекте da-ch7-ex1. При профилировании приложения ни в коем случае нельзя анализировать всю кодовую базу в целом. Вместо этого необходимо выделить только то, что действительно весьма важно для анализа. Профилирование – это операция с чрезвычайно интенсивным потреблением ресурсов, поэтому если в вашем распоряжении нет действительно мощной системы, то профилирование

всего подряд может потребовать огромного количества времени. Это еще одна причина для того, чтобы всегда начинать с выборки для определения того, что в дальнейшем необходимо профилировать.



СОВЕТ. Никогда не профилируйте всю кодовую базу приложения в целом. Всегда необходимо сначала решить на основе выборки, какую часть приложения необходимо профилировать для получения более подробной информации.

В рассматриваемом здесь примере мы оставим без внимания кодовую базу приложения (без зависимостей) и примем к рассмотрению только классы **OpenFeign** из зависимостей. Учтите, что вы не можете ссылаться на полный код приложения в реальной практике, так как, вероятнее всего, это должно приводить к значительному потреблению времени и ресурсов. Для учебного небольшого примера это не будет проблемой, но в крупных приложениях при профилировании всегда следует ограничивать объем перехватываемого кода, насколько это возможно.

На рис. 7.8 показано, как применять эти ограничения. В правой панели вкладки **Profiler** можно указать, какая часть приложения должна перехватываться. В рассматриваемом здесь примере мы используем следующие части:

- `com.example.**` – код во всех пакетах и вложенных (под)пакетах `com.example`;
- `feign.**` – код во всех пакетах и вложенных (под)пакетах `feign`.

Синтаксис, который можно использовать для выбора пакетов и классов, подлежащих профилированию, определяется всего лишь несколькими простыми правилами:

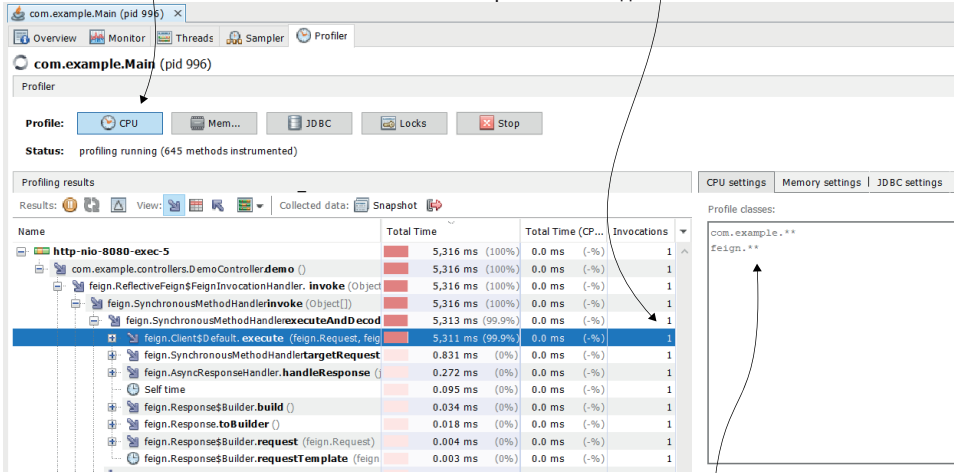
- записывайте каждое условие выбора в отдельной строке;
- используйте одну звездочку (*) для ссылки на пакет; например, можно было бы определить условие `com.example.*`, если бы было нужно профилировать все классы в пакете `com.example`;
- используйте две звездочки (**) для ссылки на пакет и все вложенные в него (под)пакеты. В рассматриваемом здесь примере при использовании `com.example.**` подразумеваются все классы в пакете `com.example`, а также в любых вложенных в него (под)пакетах;
- записывайте полное имя класса, если необходимо профилировать только этот класс; например, можно было бы использовать `com.example.controllers.DemoController` для профилирования только этого класса.

Я выбрал эти пакеты после выполнения выборки, описанной в разделе 7.1. Поскольку я заметил, что вызов метода при возникновении пробле-

мы с задержкой выполняется из классов пакета `feign`, то решил добавить этот пакет и все вложенные в него (под)пакеты в список условий для получения более подробной информации.

Щелкните по кнопке **CPU**, чтобы начать профилирование приложения.

Профилирование помогает получить больше информации о выполнении, но при этом приводит к более интенсивному потреблению ресурсов. Одна из первых дополнительных подробностей, которую вы видите здесь, – количество выполнений конкретного метода.



Всегда профилируйте небольшое количество пакетов. Перед началом профилирования определите условия выбора, чтобы сообщить профилировщику, какие классы необходимо перехватывать.

Рис. 7.8. Профилирование части приложения во время выполнения для получения подробной информации о количестве вызовов исследуемого метода. Здесь можно видеть, что метод, создающий пятисекундную задержку, вызывается только один раз, а это означает, что в данном случае количество вызовов не является причиной проблемы

В рассматриваемом конкретном случае количество вызовов не выглядит причиной возникновения проблем: метод вызывается только один раз и требует 5 с для завершения своего выполнения. Малое количество вызовов метода подразумевает, что мы не должны повторять ненужные выполнения (которые, как вы узнаете немного позже в этой главе, являются часто встречающейся проблемой во многих приложениях).

В другом сценарии, возможно, вы наблюдали, что вызов исследуемой конечной точки занимает всего лишь 1 с, но сам метод вызывается пять раз (из-за некоторого неудачного проектного решения). Затем в приложении возникает проблема, и мы должны узнать, как и где решить ее. В разделе 7.3 мы проанализируем такую проблему.

7.3. Использование профилировщика для идентификации SQL-запросов, выполняемых приложением

В этом разделе вы узнаете, как использовать профилировщик для идентификации SQL-запросов, которые приложение отправляет в СУБД. Эта тема, несомненно, является одной из наиболее предпочитаемых мною. В наши дни почти каждое приложение использует как минимум одну реляционную базу данных, и почти во всех сценариях время от времени возникают задержки, связанные с SQL-запросами. Кроме того, современные приложения применяют замысловатые способы для реализации уровня персистентности, и во многих случаях SQL-запросы, отправляемые приложением, создаются динамически фреймворком или библиотекой. Такие динамически сгенерированные запросы трудно идентифицировать, но профилировщик может совершить нечто магическое и существенно упростить анализ.

Мы будем использовать сценарий, реализованный в проекте `da-ch7-ex2`, чтобы узнать, сколько раз метод выполняется и перехватывает SQL-запросы, отправляемые приложением в реляционную базу данных. Затем будет показано, что выполняемые SQL-запросы могут быть извлечены, даже если приложение работает с фреймворком и не обрабатывает запросы напрямую. Далее мы обсудим эту тему более подробно с использованием нескольких примеров.

7.3.1. Использование профилировщика для извлечения SQL-запросов, не генерируемых фреймворком

В этом подразделе используется пример для демонстрации применения профилировщика с целью получения SQL-запросов, выполняемых приложением. Мы рассмотрим простое приложение, которое отправляет запросы в СУБД напрямую без использования фреймворка.

Начнем выполнение проекта `da-ch7-ex2` и воспользуемся вкладкой **Profiler** (Профилировщик), как было описано в разделе 7.2. Проект `da-ch7-ex2` также представляет собой небольшое приложение, которое конфигурирует прямо в памяти базу данных с двумя таблицами (`product` (товар) и `purchase` (покупка)) и заполняет их несколькими записями.

Приложение предъявляет все купленные товары, вызывая конечную точку `/products`. Под «купленными товарами» подразумеваются товары, имеющие как минимум одну запись о покупке в таблице `purchase`. Целью является анализ поведения приложения при вызове этой конечной точки без предварительного анализа кода. Таким образом, мы сможем узнать, насколько полезным может оказаться применение только профилировщика.

На рис. 7.9 показано использование вкладки **Profiler**, так как вы уже освоили выборку в разделе 7.1, но все же напомним, что в любом сценарии из реальной практики необходимо начать с выборки. Запускаем приложение

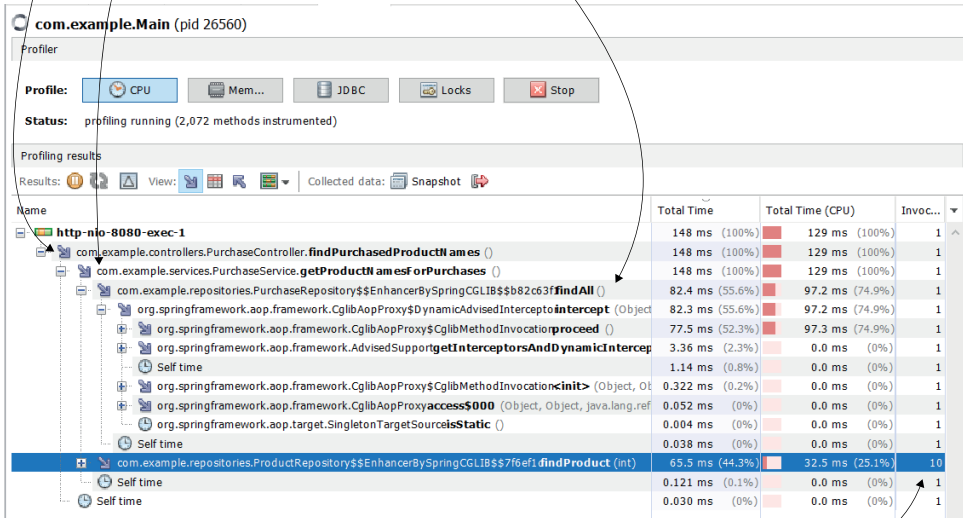
и с помощью cURL или Postman вызываем конечную точку /products. Профилировщик в точности показывает, что происходит:

- 1) был вызван метод `findPurchasedProductNames()`, принадлежащий классу `PurchaseController`;
- 2) этот метод делегировал вызов методу `getProductNamesForPurchases()` из класса `PurchaseService`;
- 3) метод `getProductNamesForPurchases()` из класса `ProductService` вызывает `findAll()` из класса `PurchaseRepository`;
- 4) метод `getProductNamesForPurchases()` из класса `ProductService` вызывает `findProduct()` из класса `ProductRepository` 10 раз.

1. Выполнение начинается с метода `findPurchasedProductNames()` в классе `PurchaseController`.

2. Вызывается метод `getProductNamesForPurchases()` из класса `PurchaseService`.

3. Метод из класса `PurchaseService` вызывает метод `findAll()` из класса `PurchaseRepository`.



4. После вызова `findAll()` в `PurchaseRepository` этот метод вызывает `findProduct()` класса `ProductRepository` 10 раз.

Рис. 7.9. При профилировании приложения можно видеть, что один из методов вызывается 10 раз. Необходимо выяснить, не является ли это проблемой проектирования. Поскольку теперь мы имеем общую картину всего алгоритма и знаем, какой код выполняется, можно начать отладку приложения, если невозможно точно определить, что происходит

Удивительно, не правда ли? Мы даже не заглядывали в код, а уже узнали так много о выполнении. Это чрезвычайно значимые подробности,

потому что теперь вы точно знаете, в какую часть кода нужно перейти и что предполагается там обнаружить. Профилировщик сообщает нам имена классов, методов и порядок их вызовов. Теперь рассмотрим код в листинге 7.2 и определим, где все это происходит. Применяя профилировщик, можно видеть, что большинство действий выполняется в методе `getProductNamesForPurchases()` класса `PurchaseService`, так что это наиболее вероятная локация, требующая анализа.

Листинг 7.2. Реализация алгоритма в классе `PurchaseService`

```
@Service
public class PurchaseService {

    private final ProductRepository productRepository;
    private final PurchaseRepository purchaseRepository;

    public PurchaseService(ProductRepository productRepository,
                           PurchaseRepository purchaseRepository) {
        this.productRepository = productRepository;
        this.purchaseRepository = purchaseRepository;
    }

    public Set<String> getProductNamesForPurchases() {
        Set<String> productNames = new HashSet<>();
        List<Purchase> purchases = purchaseRepository.findAll();           ❶
        for (Purchase p : purchases) {                                     ❷
            Product product =
                productRepository.findProduct(p.getProduct());           ❸
            productNames.add(product.getName());                           ❹
        }

        return productNames;                                             ❺
    }
}
```

- ❶ Получение всех покупок из таблицы базы данных.
- ❷ Итеративный проход по каждому товару.
- ❸ Получение подробной информации по каждому купленному товару.
- ❹ Добавление товара в набор.
- ❺ Возврат набора товаров.

Наблюдаем реализованное поведение: приложение выбирает некоторые данные и помещает их в список, затем выполняет итеративный проход по списку, чтобы получить больше данных из базы. Подобная реализация обычно свидетельствует о проблеме проектирования, потому что, как правило, можно сократить выполнение многих запросов до одного. Очевидно,

что чем меньше запросов выполняется, тем эффективнее работает приложение.

В рассматриваемом здесь примере нетрудно извлечь запросы прямо из кода. Поскольку профилировщик точно показывает, где они выполняются, а размер приложения чрезвычайно мал, поиск запросов не становится проблемой. Но реальные приложения имеют немалые размеры, и во многих случаях совсем не просто извлечь запросы непосредственно из кода. Но бояться нечего. Можно использовать профилировщик для извлечения всех SQL-запросов, которые приложение отправляет в СУБД. Это показано на рис. 7.10. Вместо кнопки **CPU** щелкните по кнопке **JDBC**, чтобы начать профилирование SQL-запросов.



Рис. 7.10. Профилировщик перехватывает SQL-запросы, отправляемые приложением в СУБД через драйвер JDBC. Это предоставляет простой способ получения запросов, их работы, наблюдения за тем, какая часть кодовой базы выполняет их, а также позволяет узнать, сколько раз выполняется запрос

Работа, скрыто выполняемая профилировщиком, очень проста: Java-приложение отправляет SQL-запросы в СУБД через драйвер JDBC. Профилировщик перехватывает драйвер и копирует запросы, прежде чем драйвер отправит их в СУБД. На рис. 7.11 показан этот подход. Результат поразительный, поскольку вы можете просто копировать и вставлять запросы в свой клиент базы данных, где можно их выполнить или анализировать соответствующий план.

Профилировщик также показывает, сколько раз был отправлен запрос. В рассматриваемом здесь примере приложение отправило первый запрос 10 раз. Такое проектное решение ошибочно, поскольку один и тот же запрос повторяется многократно, следовательно, непроизводительно рас-

ходуется время и ресурсы. Автор реализации этого кода пытался получить информацию о покупках, а затем подробности о товаре при каждой покупке. Но простой запрос с применением соединения (join) между двумя таблицами (товаров и покупок) способен решить эту задачу в один прием. К счастью, используя VisualVM, вы обнаружили причину и теперь точно знаете, что необходимо изменить для улучшения производительности приложения.

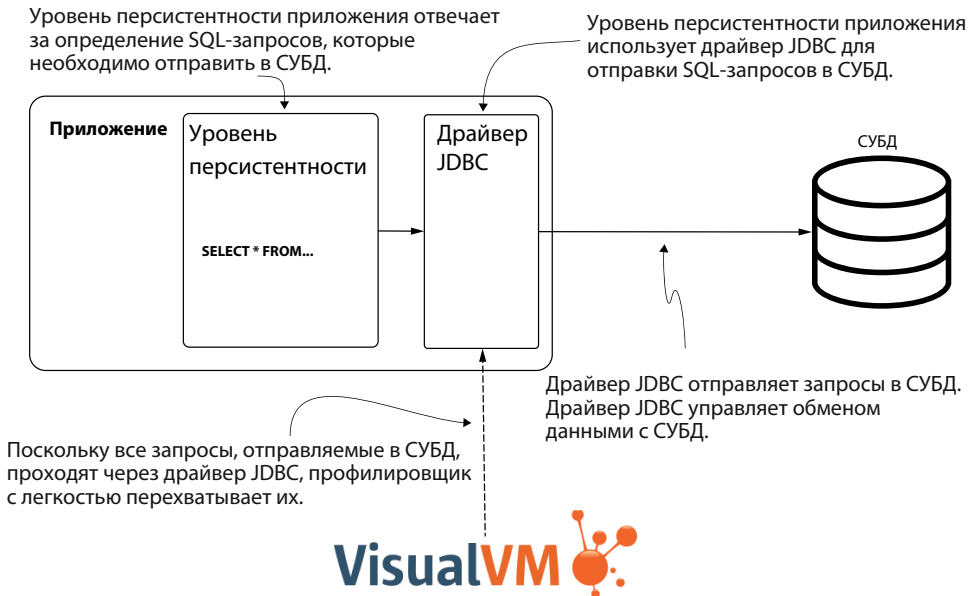


Рис. 7.11. В Java-приложении обмен данными с реляционной СУБД выполняется через драйвер JDBC. Профилировщик может перехватывать все вызовы методов, включая выполняемые драйвером JDBC, и извлекать SQL-запросы, отправляемые приложением в СУБД. Вы можете получить эти запросы и использовать их при анализе

На рис. 7.12 показано, как найти ту часть кодовой базы, которая отправляет конкретный запрос. Можно развернуть стек выполнения и с легкостью идентифицировать первый метод в кодовой базе приложения.

В листинге 7.2 показан код, вызов которого мы идентифицировали с использованием профилировщика. После определения причины и источника проблемы наступает время для чтения кода и поиска способа его оптимизации. В рассматриваемом здесь примере можно было бы объединить все запросы в один. Возможно, это выглядит как глупая ошибка, но поверьте мне, вы будете обнаруживать такие типы ошибок даже в крупных приложениях, реализованных весьма солидными организациями.

- ❶ Приложение получает список всех товаров.
- ❷ Выполняется итеративный проход по каждому товару.
- ❸ Получение подробной информации о товаре.

В примере `da-ch7-ex2` используется драйвер JDBC для отправки SQL-запросов в СУБД. Приложение содержит SQL-запросы непосредственно в коде Java (см. листинг 7.3) в их собственной форме, поэтому можно предположить, что копирование запросов прямо из кода является не таким уж сложным делом. Но в современных приложениях вам гораздо реже будут встречаться запросы в собственной форме. В настоящее время многие приложения используют фреймворки, такие как Hibernate (наиболее часто применяемая реализация Java Persistence API – JPA) или Java Object Oriented Querying (JOOQ), поэтому собственно запросы не находятся непосредственно в коде. Более подробно о JOOQ можно узнать в их репозитории GitHub: <https://github.com/jooq/jooq>.

Листинг 7.4. Репозиторий, использующий собственную форму SQL-запросов

```
@Repository
public class ProductRepository {

    private final JdbcTemplate jdbcTemplate;

    public ProductRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public Product findProduct(int id) {
        String sql = "SELECT * FROM product WHERE id = ?"; ❶
        return jdbcTemplate.queryForObject(sql, new ProductRowMapper(), id);
    }
}
```

- ❶ Собственная форма SQL-запроса, отправляемого приложением в СУБД.

7.3.2. Использование профилировщика для получения SQL-запросов, генерируемых фреймворком

Рассмотрим еще более необычную тему. Для дальнейшего подтверждения полезности профилировщика при анализе SQL-запросов ознакомимся с проектом `da-ch7-ex3`. С алгоритмической точки зрения проект делает то же самое, что и предыдущий: возвращает наименования купленных товаров. Я преднамеренно сохранил ту же логику для упрощения примера и обеспечения возможности сравнения.

В приведенном ниже фрагменте кода показано определение репозитория Spring Data JPA. Репозиторий – это простой интерфейс, и вы не види-

те никаких SQL-запросов. При использовании Spring Data JPA приложение генерирует запросы скрыто от пользователя на основе имен методов или с применением особого способа определения запросов под именем Java Persistence Query Language (JPQL), который основан на объектах приложения. В любом случае здесь не существует простого способа копирования и вставки запроса прямо из кода.

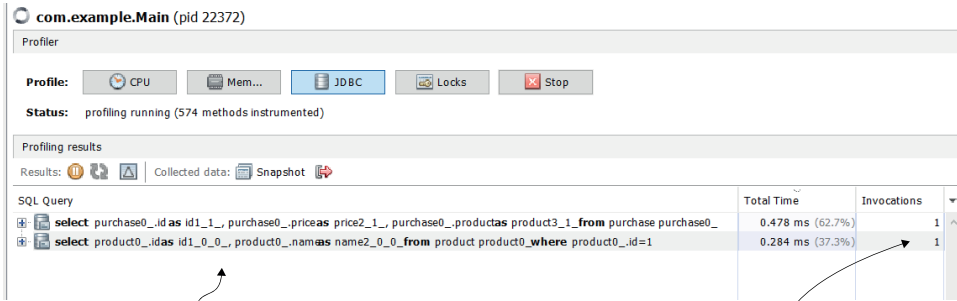
```
public interface ProductRepository
    extends JpaRepository<Product, Integer> {
}
```

Некоторые фреймворки генерируют SQL-запросы скрыто от пользователя на основе кода и конфигураций, написанных вами. В таких случаях еще более затруднительно получить выполняемые запросы. Но профилировщик может помочь в извлечении их из драйвера JDBC перед отправкой в СУБД.

Профилировщик приходит на помощь. Поскольку он перехватывает запросы до того, как приложение отправит их в СУБД, этим можно воспользоваться, чтобы точно определить, какие запросы использует приложение. Запустим приложение da-ch7-ex3 и применим VisualVM для профилирования SQL-запросов, как это было сделано для двух предыдущих проектов.

На рис. 7.13 показана информация, выводимая при профилировании вызова конечной точки /products. Приложение отправляет два запроса. Обратите внимание: псевдонимы (aliases) в каждом запросе имеют странные имена, потому что запросы сгенерированы фреймворком. Также следует отметить, что даже при той же логике в сервисе и 10-кратном вызове приложением метода репозитория второй запрос выполняется только один раз, потому что Hibernate оптимизирует выполнение там, где это возможно. Теперь можно скопировать и проанализировать этот запрос с помощью клиента SQL-разработки, если это необходимо. Во многих случаях анализ медленного запроса требует его выполнения в SQL-клиенте, чтобы выяснить, какая часть запроса создает затруднение для СУБД.

Запрос выполняется только один раз, даже несмотря на то, что соответствующий метод вызывается 10 раз. Используют ли фреймворки с поддержкой персистентности такие приемы постоянно? Хотя они являются интеллектуальными средствами, иногда то, что делается скрыто от пользователя, может увеличить сложность. Кроме того, программист, неправильно понимающий функциональность фреймворка, может написать код, создающий проблемы. Это еще одно обоснование использования профилировщика для проверки запросов, генерируемых фреймворком, и для того, чтобы убедиться в том, что приложение работает так, как ожидалось.



SQL Query	Total Time	Invocations
<code>select purchase0_id as id1_1_, purchase0_priceas price2_1_, purchase0_productas product3_1_from purchase purchase0_</code>	0.478 ms (62.7%)	1
<code>select product0_idas id1_0_0_, product0_nameas name2_0_0_from product product0_where product0_id=1</code>	0.284 ms (37.3%)	1

Профилировщик перехватил запросы при передаче в СУБД из реализации JPA. Вы можете вставить их в клиент БД, если это необходимо для дальнейшего анализа.

Обратите внимание: даже если метод вызывается 10 раз в этом примере, в СУБД запрос отправляется только один раз. Фреймворки, подобные Hibernate, оптимизируют поведение приложения.

Рис. 7.13. Даже при работе с фреймворком профилировщик способен продолжать перехваты SQL-запросов. Это существенно упрощает анализ, поскольку невозможно скопировать запрос непосредственно из кода, как это делалось при использовании JDBC и запросов в собственной форме

Чаще всего возникающие при работе с фреймворками проблемы, которые требуют анализа, перечислены ниже:

- медленные запросы становятся причиной задержек – легко обнаруживаются с помощью профилировщика при анализе времени выполнения;
- многочисленные ненужные запросы, сгенерированные фреймворком (обычно из-за того, что разработчики называют проблемой запросов $N+1$), – легко обнаруживаются с помощью профилировщика для определения количества выполнений запроса;
- длительные подтверждения транзакций из-за неудачного проектного решения приложения – легко обнаруживаются при использовании профилирования ЦП.

Если фреймворку необходимы данные из нескольких таблиц, то обычно ему известно, как сформировать единственный запрос и получить все данные в одном вызове. Но если вы неправильно используете фреймворк, то он может получить лишь часть данных при первом запросе, а затем для каждой изначально извлеченной записи выполняется отдельный запрос. Таким образом, вместо выполнения единственного запроса фреймворк будет отправлять первоначальный запрос плюс N других (по одному для каждой из N записей, извлеченных первым запросом). Это называют проблемой запросов $N+1$ ($N+1$ query problem), которая обычно создает существенную задержку из-за выполнения множества запросов вместо одного.

Кажется, что большинство разработчиков предпочитает работать с журналами или отладчиком для анализа подобных проблем. Но, исходя из

своего опыта, отмечу, что ни одно из этих средств не является наилучшим вариантом выбора для определения главной причины возникновения проблемы.

Первое затруднение при использовании журналов для этого варианта возникает из-за того, что сложно определить, какой запрос стал причиной проблемы. В сценариях из реальной практики приложение может отправлять десятки запросов, а некоторые из них по нескольку раз, и в большинстве случаев они длинные и используют большое количество параметров. С помощью профилировщика, который выводит все запросы в списке вместе с соответствующим временем выполнения и количеством выполнений, можно почти сразу обнаружить проблему. Второе затруднение состоит в том, что, даже если вы определили запрос, предположительно создающий проблему (например, при мониторинге журналов обнаружилось, что приложение затрачивает слишком длительное время на выполнение конкретного запроса), не так-то просто взять этот запрос и выполнить его. В журнале вы обнаружите, что параметры отделены от запроса.

Можно сконфигурировать приложение так, чтобы оно выводило запросы, сгенерированные Hibernate, в журналы, добавив некоторые параметры в свойства приложения в файле `da-ch7-ex3`:

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
logging.level.org.hibernate.type.descriptor.sql=trace
```

Но при этом следует учесть, что вам придется использовать различные способы конфигурирования подсистемы журналирования в зависимости от технологий, применяемых при реализации приложения. В примере, прилагаемом к книге, мы используем Spring Boot и Hibernate. В листинге 7.5 показано, как приложение выводит запрос в журнал.

Листинг 7.5. Журнальные записи о запросах, сгенерированных и передаваемых Hibernate

```
Hibernate:
  Select
    product0_.id as id1_0_0_,
    product0_.name as name2_0_0_
  from
    product product0_
  where
    product0_.id=?

2021-10-16 13:57:26.566 TRACE 9512 --- [nio-8080-exec-2]
➔ o.h.type.descriptor.sql.BasicBinder      : binding parameter [1] as
➔ [INTEGER] - [1]
2021-10-16 13:57:26.568 TRACE 9512 --- [nio-8080-exec-2]
```

```

➔ o.h.type.descriptor.sql.BasicExtractor : extracted value ([name2_0_0_] : | ❸
➔ [VARCHAR]) - [Chocolate]

```

- ❶ Запрос, сгенерированный приложением.
- ❷ Значение первого параметра.
- ❸ Значение второго параметра.

В журнале показан запрос и представлены его входные и выходные данные. Но вам необходимо связать значения параметров с самим запросом, если требуется выполнить его отдельно. А когда в журнал записываются многочисленные запросы, поиск необходимого может стать по-настоящему обескураживающим. Кроме того, журнал не показывает, какая часть приложения выполняет проблемный запрос, а это может сделать анализ еще более сложным.



ПРИМЕЧАНИЕ. Я рекомендую всегда начинать с использования профилировщика при анализе проблем с задержками. Первым шагом непременно должно быть выполнение выборки. Если предполагаются проблемы, связанные с SQL-запросами, то продолжайте профилирование для JDBC. Тогда проблемы будет проще понять, и вы сможете воспользоваться отладчиком или журналом для подтверждения своих предположений при необходимости.

7.3.3. Использование профилировщика для получения программно сгенерированных SQL-запросов

Для полноты картины рассмотрим еще один пример, демонстрирующий, как работает профилировщик, если приложение программно определяет запросы. Проанализируем проблему с производительностью, связанную с запросом, сгенерированным Hibernate (фреймворком, использованным в этом примере), в приложении, применяющем запросы с критерием поиска (*criteria queries*), – это программный способ определения уровня персистентности приложения с использованием Hibernate. При таком подходе вы никогда не пишете сам запрос ни в собственной форме, ни на языке JPQL.

Как можно видеть в листинге 7.6, где представлен класс *ProductRepository*, заново реализованный с использованием запроса с критерием поиска, этот подход требует увеличения объема кода. Обычно он считается более сложным и создает большую возможность для ошибок. Реализация в проекте *da-ch7-ex4* содержит ошибку, которая может стать причиной серьезных проблем с производительностью в реальных промышленных приложениях. Попробуем обнаружить эту проблему и определить, как профилировщик может помочь в понимании того, что здесь неправильно.

Листинг 7.6. Репозиторий, определенный с запросом с критерием поиска

```

public class ProductRepository {

    private final EntityManager entityManager;

    public ProductRepository(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    public Product findById(int id) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Product> cq = cb.createQuery(Product.class);      ❶

        Root<Product> product = cq.from(Product.class);                 ❷
        cq.select(product);                                              ❸

        Predicate idPredicate =                                         | ❹
            cb.equal(cq.from(Product.class).get("id"), id);
        cq.where(idPredicate);                                           ❺

        TypedQuery<Product> query = entityManager.createQuery(cq);      ❻
        return query.getSingleResult();
    }
}

```

- ❶ Создание нового запроса.
- ❷ Определение запроса, выбирающего товары.
- ❸ Выбор товаров.
- ❹ Определение условия, которое становится частью спецификатора where в следующей строке.
- ❺ Определение спецификатора where.
- ❻ Выполнение запроса и извлечение результата.

Мы используем профилирование JDBC для перехвата запросов, отправляемых приложением в СУБД. Здесь можно видеть, что запрос содержит перекрестное соединение (cross join) таблицы товара (product) с собой (см. рис. 7.14). Это весьма значительная проблема. С 10 записями в таблице мы не наблюдаем ничего подозрительного в данном случае. Но в реальном промышленном приложении, где таблицы содержат гораздо больше записей, такое перекрестное соединение может создавать огромные задержки и в ряде случаев даже некорректные выходные данные (дублирующиеся строки). Простой перехват этого запроса с помощью VisualVM и его чтения показывает нам возникшую проблему.

Запрос содержит бесполезное перекрестное соединение.
В реальном промышленном приложении это может
стать причиной проблем с производительностью и даже
некорректного поведения при выводе результата.

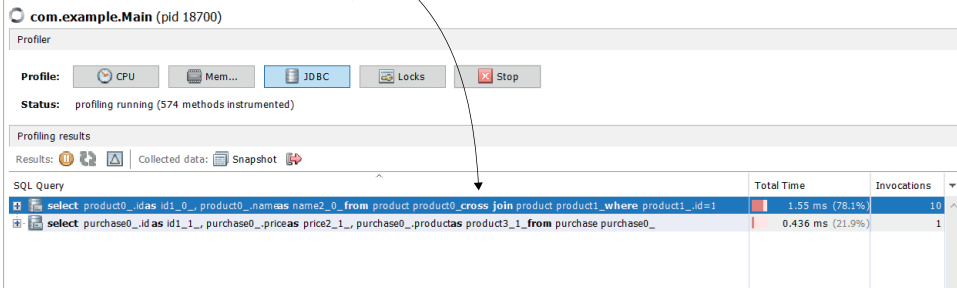


Рис. 7.14. Профилировщик может перехватить любой SQL-запрос, отправленный в СУБД через драйвер JDBC. Здесь мы обнаруживаем проблему в сгенерированном запросе – ненужное перекрестное соединение становится причиной проблем с производительностью

Следующий вопрос: «Почему приложение сгенерировало запрос именно так?» Мне нравится высказывание о реализациях JPA, подобных Hibernate: «Отличная новость – они делают генерацию запросов прозрачной и требующей минимума работы. Плохая новость – они делают генерацию запросов прозрачной, создавая в приложении большую вероятность возникновения ошибок». При работе с такими фреймворками я обычно рекомендую разработчикам профилировать запросы как часть процесса разработки, чтобы выявлять подобные проблемы заранее. Применение профилировщика в большей степени ориентировано на инспекцию (аудитинг) кода, нежели на поиск проблем, тем не менее это неплохая мера обеспечения безопасности.

В следующем примере я преднамеренно ввел эту крошечную ошибку с существенным воздействием. Я вызываю метод `from()` дважды, сообщая Hibernate о необходимости создания перекрестного соединения (см. листинг 7.7).

Листинг 7.7. Причина возникновения проблемы с перекрестным соединением

```
public class ProductRepository {

    // Здесь код не показан.

    public Product findById(int id) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Product> cq = cb.createQuery(Product.class);
```

```

Root<Product> product = cq.from(Product.class);    ❶
cq.select(product);

Predicate idPredicate = cb.equal(
    cq.from(Product.class).get("id"), id);        ❷
cq.where(idPredicate);

TypedQuery<Product> query = entityManager.createQuery(cq);
return query.getSingleResult();
}
}

```

❶ Первый вызов метода `from()` класса `CriteriaQuery`.

❷ Повторный вызов метода `from()` класса `CriteriaQuery`.

Эта проблема устраняется просто: во второй раз используется экземпляр товара вместо вызова метода `CriteriaQuery from()`, как показано в листинге 7.8.

Листинг 7.8. Устранение проблемы с перекрестным соединением

```

public class ProductRepository {

    // Omitted code

    public Product findById(int id) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Product> cq = cb.createQuery(Product.class);

        Root<Product> product = cq.from(Product.class);
        cq.select(product);

        Predicate idPredicate = cb.equal(product.get("id"), id);    ❶
        cq.where(idPredicate);

        TypedQuery<Product> query = entityManager.createQuery(cq);
        return query.getSingleResult();
    }
}

```

❶ Использование уже существующего объекта `Root`.

После внесения этого небольшого изменения сгенерированный SQL-запрос уже не содержит ненужное перекрестное соединение (см. рис. 7.15). Но приложение все еще выполняет один и тот же запрос несколько раз, что не является оптимальным решением. Алгоритм, применяемый прило-

жением, требует рефакторинга, чтобы по возможности получать данные, используя только один запрос.

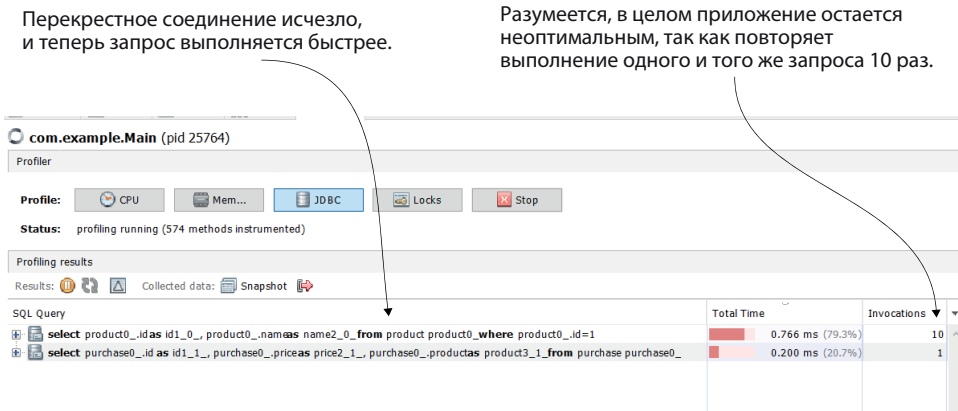


Рис. 7.15. После исключения вспомогательного вызова метода `select()` перекрестное соединение исчезает. Но весь алгоритм в целом для этого приложения необходимо пересмотреть, поскольку он продолжает выполнять один и тот же запрос несколько раз, а это не оптимальное решение

7.4. Резюме

- Профилировщик перехватывает поток выполнения приложения и предоставляет весьма важные подробности о выполняемом коде, например трассировку стека выполнения каждого потока, время, затрачиваемое на выполнение каждого метода, и количество вызовов конкретного метода.
- При анализе проблем с задержками первым этапом применения профилировщика является выборка – метод перехвата выполняющегося кода без показа многочисленных подробностей. Выборка потребляет меньше ресурсов и позволяет наблюдать общую картину выполнения в целом.
- Выборка предоставляет три существенные подробности:
 - какой именно код выполняется – при анализе проблемы иногда неизвестно, какой код выполняется в определенный интервал времени, но это можно определить с помощью выборки;
 - общее время выполнения каждого метода – эта характеристика помогает определить, какая часть кода предположительно является причиной возникновения задержки;
 - общее время использования ЦП – эта характеристика помогает определить, затрачивает ли интересующий нас код время выполнения на «реальную работу», или просто ожидает чего-то.

- Иногда выполнения выборки достаточно, чтобы понять, где возникла проблема. Но во многих случаях необходимы дополнительные подробности. Их можно получить, профилируя выполнение.
- Профилирование является процессом с высоким потреблением ресурсов. В реальном промышленном приложении почти всегда невозможно профилирование всей кодовой базы в целом. Поэтому при подробном профилировании необходимо выбрать конкретные пакеты и классы, на которых вы должны сосредоточить внимание при анализе. Части приложения, требующие особого внимания, обычно определяются во время предварительной выборки.
- Весьма важной характеристикой, получаемой при профилировании, является количество вызовов метода. При выполнении выборки вы узнаете общее время, затраченное на выполнение метода, но не указывается, сколько раз он был вызван. Эта характеристика важна для того, чтобы понять, что метод замедляет работу или используется некорректно.
- Профилировщик можно использовать для получения SQL-запросов, отправляемых приложением в СУБД. Профилировщик перехватывает любые запросы независимо от технологии, применяемой для реализации уровня персистентности приложения. Это чрезвычайно ценная информация при анализе медленных запросов в приложении, использующем фреймворки (например, Hibernate) для работы с базой данных.

Глава 8

Использование продвинутых инструментов визуализации для профилируемых данных

Темы:

- выявление проблем в соединениях с реляционными базами данных;
- использование графов вызовов методов для более быстрого понимания проектного решения приложения;
- использование flame-графиков для упрощенной визуализации выполнения приложения;
- анализ запросов, отправляемых приложением на сервер NoSQL базы данных.

В этой главе мы рассмотрим полезные методики, упрощающие работу при анализе в особых сценариях. Начнем главу с изучения методики выявления проблем в соединениях между Java-приложением и сервером реляционной базы данных. Мы уже обсуждали профилирование SQL-запросов в главе 7, но иногда проблемы возникают во время установления приложением соединения с СУБД. Такие ситуации могут привести даже к тому, что приложение вообще перестает отвечать, поэтому чрезвычайно важно найти причины возникновения подобных проблем.

В разделе 8.2 демонстрируется один из наиболее предпочитаемых мною способов понимания кода в конкретном сценарии выполнения – простая методика использования графов вызовов методов, т. е. визуальных представлений зависимостей между объектами приложения. Я считаю графы вызовов полезными, особенно при работе с запутанными кодовыми базами, с которыми раньше не имел дела. И поскольку я уверен в том, что большинству разработчиков приходится разбираться с запутанными кодовыми

базами в определенные моменты их профессиональной деятельности, знание этой методики будет весьма полезным.

В главе 7 рассматривался один из наиболее часто применяемых способов визуализации выполнения приложения – стек выполнения. Вы узнали, как сгенерировать стек выполнения во время выборки или профилирования с помощью VisualVM и как воспользоваться им для определения задержек при выполнении. В разделе 8.3 мы используем другое представление стека выполнения: flame-график. Flame-графики представляют собой способ визуализации выполнения приложения, который одновременно сосредоточен и на выполняемом коде, и на времени выполнения. Наблюдение за одними и теми же данными с дополнительного ракурса иногда может помочь более просто обнаружить то, что вы ищете. Как вы узнаете в разделе 8.3, flame-графики предлагают другую точку зрения на выполнение приложения, которая помогает идентифицировать потенциальные задержки и проблемы с производительностью. В разделе 8.4 рассматриваются методики анализа работы уровня персистентности приложения, когда не используются реляционные базы данных, а вместо них применяются различные способы обеспечения персистентности, называемые «семейством NoSQL-технологий».

Для изучения тем этой главы использования профилировщика VisualVM недостаточно. VisualVM – превосходный бесплатный инструмент, который я применяю более чем в 90 % сценариев анализа с профилировщиком, но и у него есть свои ограничения.

Для демонстрации возможностей, описываемых в этой главе, мы будем использовать JProfiler (<http://mng.bz/RvVn¹>) – профилировщик с лицензией. JProfiler предоставляет (за небольшую цену) те же возможности, что и VisualVM, но кроме них еще и функции, которых нет в VisualVM. Вы можете воспользоваться предлагаемым интервалом времени для опробования программы, чтобы профилировать примеры из этой книги, и сформировать собственное мнение о различиях между VisualVM и JProfiler.

8.1. Выявление проблем в JDBC-соединениях

В главе 7 мы обсуждали многочисленные подробности анализа проблем в SQL-запросах. А как насчет соединения, которое приложение должно установить с СУБД для отправки запросов? Небрежное отношение к управлению соединением может привести к возникновению проблем, которые мы рассмотрим в этом разделе, а также уделим особое внимание способам поиска главных причин их возникновения.

Кто-то может возразить, что приложения используют фреймворки и библиотеки, которые позаботятся об управлении соединением в большинстве случаев, следовательно, подобные проблемы возникать не

¹ Это нерабочая ссылка. Информацию о JProfiler можно найти здесь: <https://www.ej-technologies.com/products/jprofiler/overview.html>. – Прим. перев.

должны. Но опыт подсказывает мне, что такие проблемы продолжают возникать в основном из-за того, что разработчики зависимы от множества аспектов, которые должны быть автоматизированными. Иногда мы вынуждены использовать менее общепринятые и относительно низкоуровневые реализации вместо зависимостей, предлагаемых фреймворком, и именно здесь возникает большинство проблем такого типа.

Позвольте мне рассказать о проблеме, с которой я встретился недавно. В специализированном сервисе (реализованном с применением Spring) разработчикам пришлось реализовать нечасто применяемое функциональное средство: механизм отмены выполнения хранимой процедуры (процедуры, выполняемой на уровне базы данных).

Эта реализация не была сложной, но требовала прямого доступа к объекту соединения. Spring – надежный фреймворк и легко специализируется, поэтому можно без затруднений получить доступ к управляемым им соединениям, но будет ли Spring продолжать управлять этими соединениями после того, как вы получили доступ к ним? Ответ: иногда. И вот это «иногда» создает весьма интересную (но и чрезвычайно затруднительную) ситуацию.

Разработчики обнаружили, что при стандартном выполнении метода, в котором Spring управляет транзакциями, фреймворк еще и закрывает соединения в конце работы. Процедура (хранимая) отменялась с использованием пакетной (batching) методики, реализованной из Spring Batch. В таких случаях фреймворк не закрывает соединение – вы обязаны управлять им. Разработчики использовали одинаковый подход в обоих случаях, но не учли, что в одном из них соединения не были корректно закрыты, а это могло стать причиной серьезной проблемы. К счастью, эта ошибка была обнаружена вовремя и не причинила вреда.

Эта история показывает, почему методика, рассматриваемая в текущем разделе, остается столь важной. Названия используемых фреймворков значения не имеют, и вы, вероятно, никогда не узнаете обо всем, что происходит скрыто от ваших глаз, так что будьте готовы анализировать выполнение приложения любыми способами, которые всегда остаются полезными.

Мы будем использовать проект da-ch8-ex1, приложенный к этой книге. Проект определяет простое приложение с огромной проблемой: один из его методов «забыл» закрыть ранее открытые JDBC-соединения. Приложение создает JDBC-соединение для отправки SQL-запросов в СУБД. JDBC-соединения всегда должны закрываться, если они больше не нужны приложению. Все СУБД предоставляют клиентам (т. е. приложениям) возможность установления ограниченного количества соединений (обычно это небольшое число, например 100). Если приложение отрывает все эти соединения, но не закрывает их, то не сможет установить очередное соединение с сервером базы данных (см. рис. 8.1).

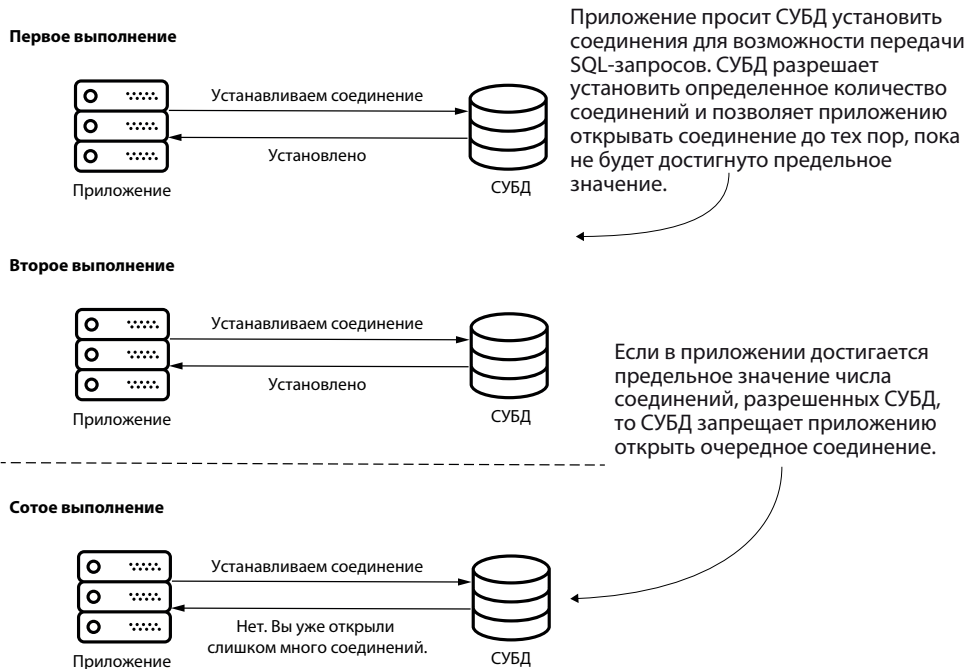


Рис. 8.1. СУБД позволяет приложению открыть конечное и обычно небольшое число соединений. Если приложение достигает предельного числа соединений, которые оно может открыть, то СУБД не позволяет открывать новые соединения. В этом случае приложение может стать неспособным выполнить свою работу

СУБД не всегда предоставляет ровно 100 соединений. Это число конфигурируется на уровне базы данных. При работе с базой данных лучше всего выяснить (обычно у администратора базы данных), какое максимально количество соединений может открыть приложение.



ПРИМЕЧАНИЕ. Для упрощения демонстрационного примера мы будем использовать уровень персистентности, который ограничивает количество соединений значением 10.

Запустим проект `da-ch8-ex1` и проанализируем поведение приложения. Проект определяет простое приложение, которое сохраняет подробности о товарах в базе данных. Приложение предъявляет конечную точку `/products`. При вызове конечной точки приложение возвращает подробности на основе данных, хранимых в базе. При первом вызове конечной точки приложение отвечает почти мгновенно. Но при отправке второго запроса в ту же конечную точку приложение после 30 с паузы отвечает сообщением об ошибке, как показано на рис. 8.2.

При первом вызове конечной точки приложение отвечает немедленно.

```
$ curl http://localhost:8080/products
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Dload  Upload   Total   Spent    Left   Speed
100    13      0    13      0      0      119      0  --:--:-- --:--:-- --:--:-- 120["Chocolate"]
```

Но при вызове той же конечной точки во второй раз приложение выдает сообщение об ошибке после 30 с паузы.

```
$ curl http://localhost:8080/products
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Dload  Upload   Total   Spent    Left   Speed
100   109      0   109      0      0       3      0  --:--:-- 0:00:30 --:--:-- 28{"timestamp":
07:55:09.272+00:00","status":500,"error":"Internal Server Error","path":"/products"}
```

Рис. 8.2. При первом вызове конечной точки `/products` приложение отвечает мгновенно, возвращая список, содержащий слово «Chocolate». Но если вы попытаетесь во второй раз вызвать ту же конечную точку, то приложение «зависает» приблизительно на 30 с, а потом выдает сообщение об ошибке

Для показанной здесь демонстрации абсолютно не важно, что именно делает это приложение, поэтому я не буду углубляться в подробности его функциональности, но давайте предположим, что ваш друг, работающий над отдельным проектом, просит вас помочь и показывает такую проблему. Он не описывает множество подробностей, относящихся к тому, как работает его приложение (в реальном промышленном приложении это может быть сложная бизнес-модель). Сможете ли вы помочь ему в такой ситуации? Начнем с анализа показанного поведения.

Необходимо найти причины проблематичного поведения. Вы получаете доступ к журналу и сразу же предполагаете, что проблема связана с JDBC-соединениями. Сообщение об исключении в приведенном ниже фрагменте говорит о том, что приложение не может установить соединение, вероятнее всего, из-за того, что СУБД не разрешает открывать новые соединения. Но предположим, что мы не можем всегда полагаться на журнальные записи. В конце концов, нельзя же знать наверняка, что другой фреймворк или библиотека, используемая приложением, не сгенерировала это очевидное сообщение об исключении:

```
java.sql.SQLTransientConnectionException:
➔ HikariPool-1 - Connection is not available,
➔ request timed out after 30014ms.
    at com.zaxxer.hikari.pool.HikariPool
        ➔ .createTimeoutException(HikariPool.java:696) at
        ➔ com.zaxxer.hikari.pool.HikariPool
        ➔ .getConnection(HikariPool.java:197)
    at [CA]com.zaxxer.hikari.pool.HikariPool
        ➔ .getConnection(HikariPool.java:162)
    at [CA]com.zaxxer.hikari.HikariDataSource
```

❶

```

➔ .getConnection(HikariDataSource.java:128)
  at [CA]com.example.repositories.PurchaseRepository
➔ .findAll(PurchaseRepository.java:31)
  at [CA]com.example.repositories.PurchaseRepository
➔ $$FastClassBySpringCGLIB$$d661c9a0.invoke(<generated>)

```

❶ Сообщение об исключении.

Как рекомендовалось в главе 7, любой анализ с профилированием должен начинаться с выборки, которая формирует общую картину выполнения и предоставляет подробности, необходимые для продолжения анализа. Если вы используете VisualVM, то результат выборки должен выглядеть приблизительно так, как показано на рис. 8.3.

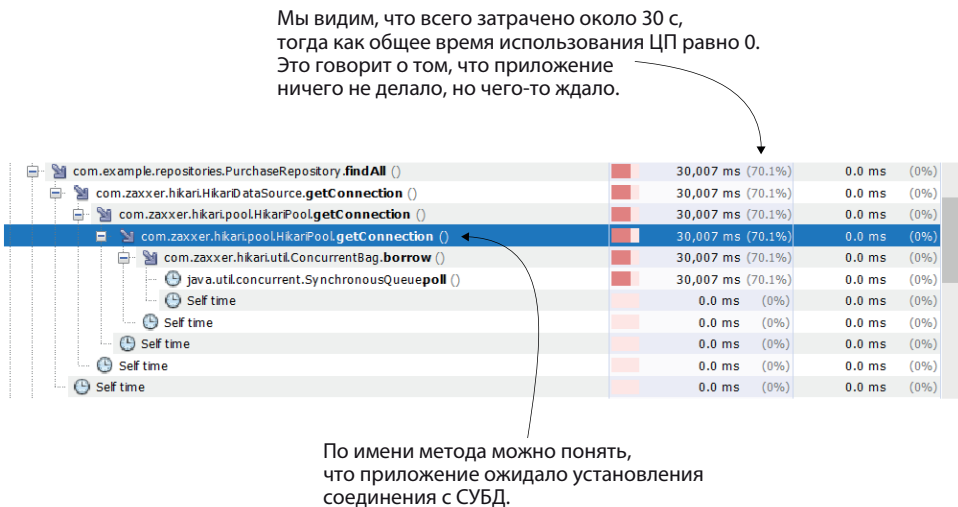


Рис. 8.3. После завершения выборки выполнения у нас появляется больше оснований для предположения о том, что пошло не так при установлении соединения с СУБД. В стеке выполнения можно видеть, что приложение ждет 30 с установления соединения

После выборки и наблюдения трассировки стека исключений в журнале мы знаем, что в приложении возникла проблема при установлении соединения с СУБД. Но что стало причиной возникновения этой проблемы? Возможен один из двух вариантов:

- обмен данными между приложением и СУБД не может выполняться из-за каких-то проблем в инфраструктуре или сетевой среде;
- отказ СУБД в установлении соединения с приложением:
 - из-за проблемы с аутентификацией;
 - из-за того, что приложение уже установило все разрешенные соединения с СУБД.

Поскольку в рассматриваемом здесь случае проблема всегда возникает во время второй передачи запроса (это явно определенный паттерн для ее воспроизведения), можно исключить проблему с обменом данными. Это явно должно происходить из-за того, что СУБД запрещает установление соединения. Но это не может быть проблемой аутентификации, потому что первый вызов отработал корректно. Вряд ли что-то изменилось в идентификационных данных, поэтому наиболее вероятной причиной является то, что приложение не всегда закрывает соединения. Теперь необходимо просто узнать, где это происходит. Напомню, что метод, в котором возникла проблема, не всегда является ее источником. Возможно, он оказался «неудачником», пытающимся установить соединение после того, как некоторый другой метод «съел» все разрешенные соединения.

Но с помощью VisualVM невозможно напрямую анализировать JDBC-соединения, поэтому мы не можем воспользоваться этим профилировщиком, чтобы определить, какое соединение остается открытым. Мы продолжим анализ, используя JProfiler. Подключение JProfiler к выполняющемуся процессу Java очень похоже на подключение VisualVM. Рассмотрим эту процедуру подробнее, шаг за шагом.

Сначала в основном окне JProfiler щелкните по кнопке **Start Center**, расположенной в верхнем левом углу, как показано на рис. 8.4.

После открытия окна JProfiler щелкните по кнопке Start Center для подключения профилировщика к процессу.

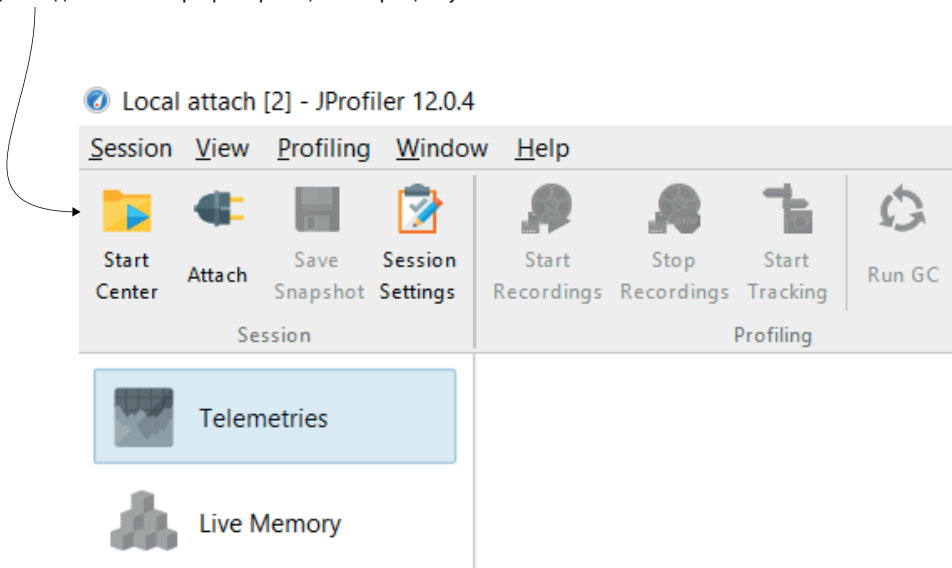
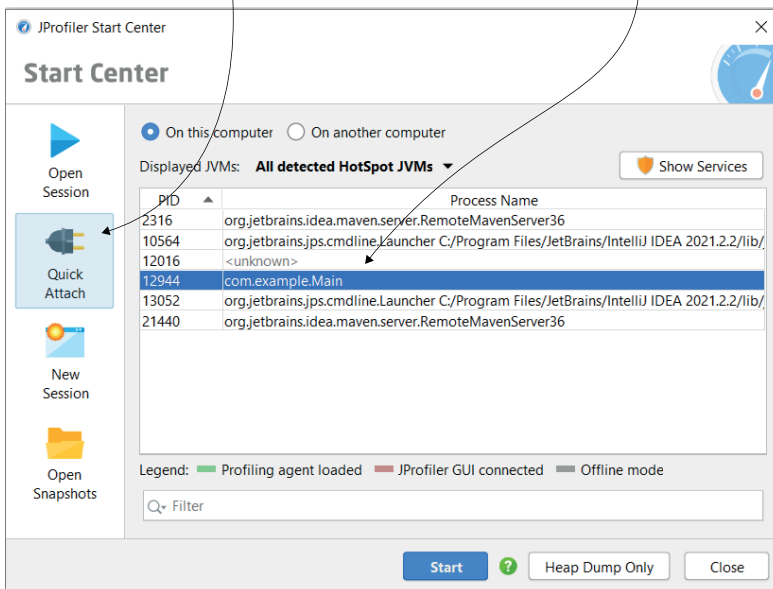


Рис. 8.4. Начните сеанс выборки или профилирования в JProfiler, щелкнув по кнопке **Start Center** в верхнем левом углу окна профилировщика

Появится всплывающее окно (см. рис. 8.5), и вы сможете выбрать функцию **Quick Attach** (Быстрое подключение) на левой панели, чтобы получить список всех процессов Java, выполняющихся локально. Выберите процесс, который необходимо профилировать, затем щелкните по кнопке **Start** (Пуск). Как и в VisualVM, вы идентифицируете процессы по имени основного класса или по идентификатору PID (process ID).

1. Щелкните по кнопке **Quick Attach** в левой панели окна **Start Center**.

2. В окне **Start Center** вы увидите список всех процессов Java, выполняющихся в локальной системе. Выберите процесс, который необходимо профилировать. Процессы можно идентифицировать, используя класс Main и основные имена пакетов.



3. После выбора процесса для профилирования щелкните по кнопке **Start**, чтобы начать сеанс профилирования.

Рис. 8.5. Во всплывающем окне щелкните по кнопке **Quick Attach**, затем в списке выберите процесс, который необходимо профилировать. После этого щелкните по кнопке **Start**, чтобы начать сеанс профилирования

JProfiler спросит, хотите ли вы использовать выборку или инструментровку (инструментовка равнозначна профилированию в терминологии VisualVM), как показано на рис. 8.6. Мы выбираем инструментовку, поскольку используем профилировщик для получения подробностей о JDBC-соединениях, следовательно, требуется более глубокий анализ выполнения.

Далее выбираем вариант **Instrumentation** (Инструментовка). Инструментовка равнозначна профилированию в VisualVM. Как было отмечено при описании работы с VisualVM, при анализе проблемы сначала используется выборка для определения области возникновения проблемы, затем – профилирование (инструментовка) для дальнейшего анализа.

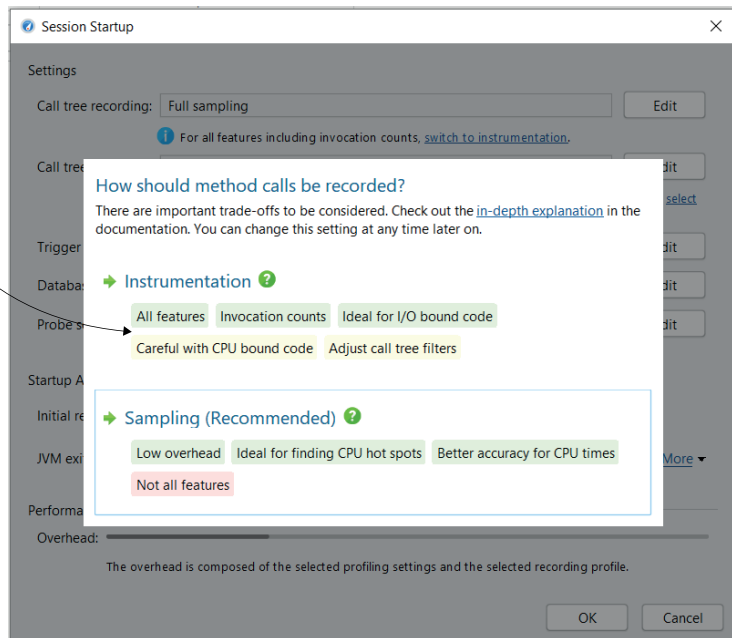


Рис. 8.6. Для более глубокого анализа выполнения необходимо выбрать вариант **Instrumentation** (Инструментовка), равнозначный профилированию в VisualVM

В меню слева под пунктом **Databases** (Базы данных) выберите пункт **JDBC**. Затем начните профилирование JDBC, как показано на рис. 8.7.

После начала профилирования нас больше всего интересуют две вкладки: **Connections** (Соединения) и **Connection Leaks** (Утечки в соединениях) (см. рис. 8.8). На этих вкладках показаны подробности о соединениях с СУБД, открытых приложением, и мы воспользуемся ими для определения главной причины возникновения проблемы.

Теперь можно воспроизвести проблему и профилировать выполнение. Отправим запрос в конечную точку `/products` и понаблюдаем, что происходит. Вкладка **Connections** сообщает, что создано много соединений, как показано на рис. 8.9. Поскольку нам неизвестно, что делает приложение, множество соединений не всегда означает возникновение проблем. Но предполагается, что приложение закрыло эти соединения, чтобы при необходимости можно было создать новые. Необходимо узнать, закрыло ли приложение эти соединения корректно.

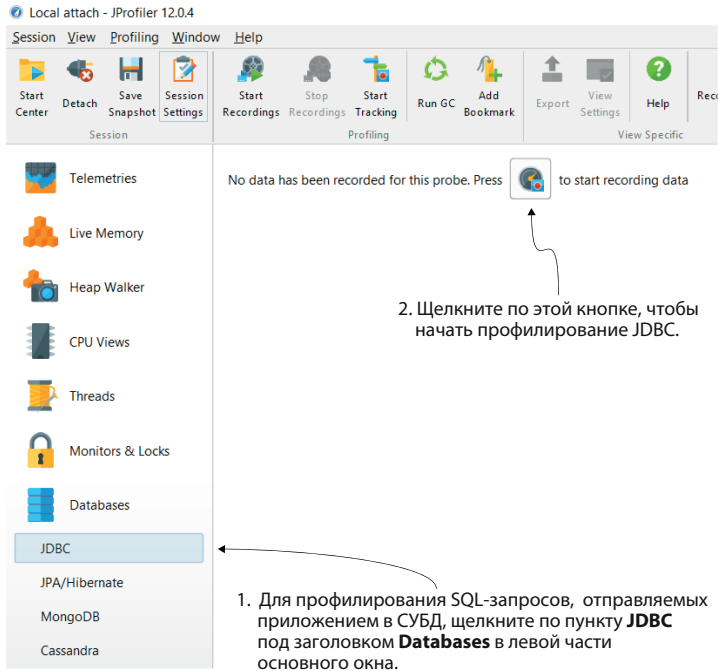


Рис. 8.7. Начать профилирование JDBC с помощью JProfiler можно, сначала выбрав пункт **JDBC** в меню слева, а затем активизировать процесс профилирования

В текущем разделе нас интересуют вкладки **Connections** и **Connection Leaks**, которые мы будем использовать для выявления проблем с соединениями, не закрытыми приложением.

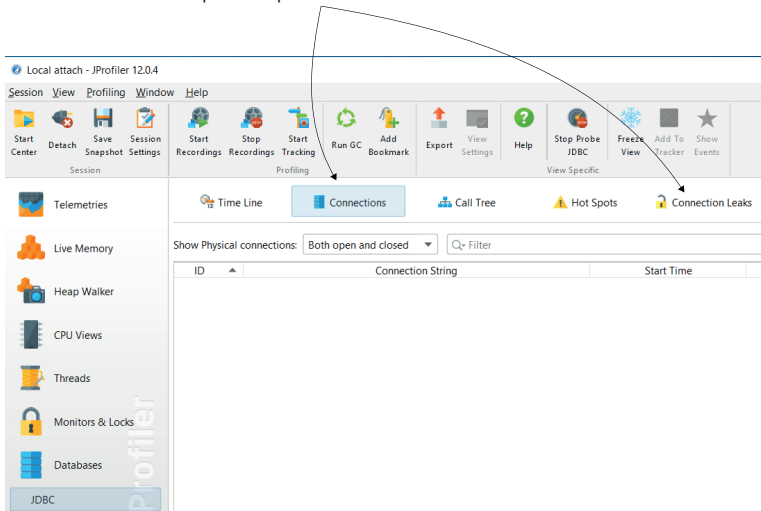


Рис. 8.8. На вкладках **Connections** и **Connection Leaks** показаны подробности о соединениях, установленных приложением, в том числе и предполагаемые проблемные соединения. Мы воспользуемся этими подробностями, чтобы понять, где именно возникла проблема в приложении

При передаче запроса в конечную точку `/products` приложение открывает большое количество соединений. Мы используем вкладку **Connections** в профилировщике, чтобы увидеть все соединения, открытые приложением.

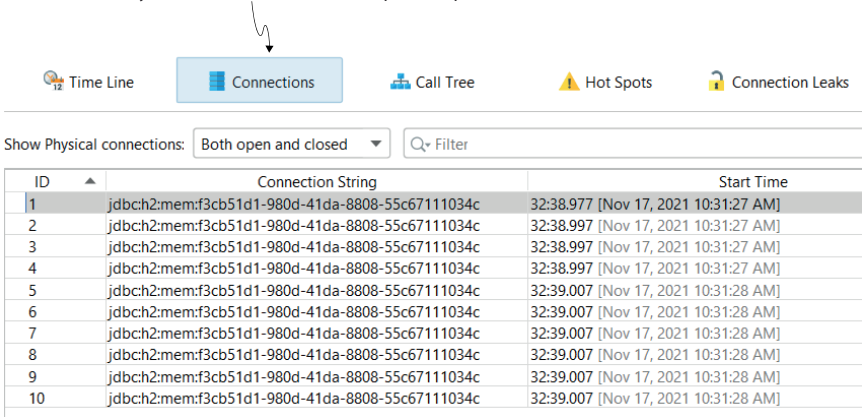


Рис. 8.9. После передачи запроса в конечную точку `/products` мы видим, что приложение создает много соединений. Нам точно неизвестно, что именно делает приложение, но это может оказаться тревожным признаком

Вкладка **Connection Leaks** подтверждает наше подозрение (см. рис. 8.10) – приложение не просто открывает множество соединений, они долго остаются открытыми после ответа конечной точки. Это очевидный признак утечки соединений. Если бы мы явно начали профилирование ЦП (скоро я продемонстрирую, как это делается), то вы увидели бы только имя потока, создавшего соединение. Иногда достаточно узнать имя потока, и в этом случае нет никакой необходимости в профилировании ЦП. Но в ситуации, описываемой здесь, нам не предоставлено достаточной информации для определения кода, создавшего соединение.

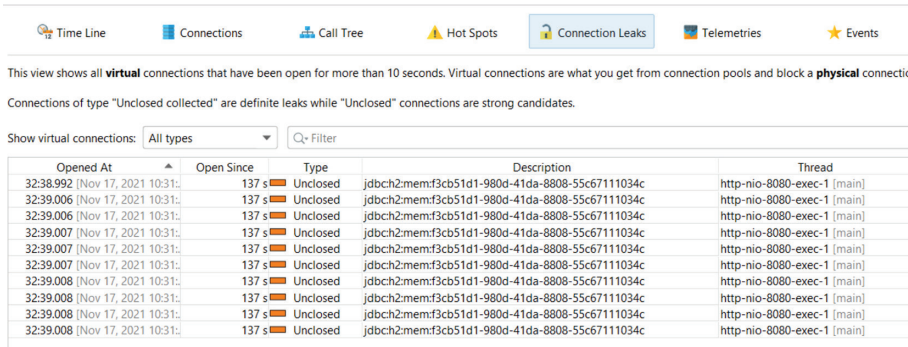


Рис. 8.10. На вкладке **Connection Leaks** показано состояние каждого соединения.

Нас интересуют соединения, которые не закрываются слишком долго или вообще никогда. Здесь соединения, открытые приложением, остаются открытыми в течение длительного времени после того, как конечная точка отправила ответ клиенту, и это убедительный признак возникновения проблемы

Но этого недостаточно, не так ли? Мы предполагали, что проблема возникла при установке приложением соединения с СУБД. Теперь необходимо использовать функцию профилирования ЦП, чтобы определить ту часть кодовой базы, которая создает соединения и забывает их закрыть.

Нам все еще нужен способ, позволяющий найти код, создающий утечки соединений. К счастью, JProfiler может помочь нам и в этом, но потребуется повторить выполнение примера после разрешения профилирования ЦП. После активизации профилирования ЦП JProfiler показывает для каждого незакрытого соединения трассировку стека вплоть до метода, создавшего это соединение.

На рис. 8.11 показано, как подключить профилирование ЦП и как найти трассировку стека для каждого незакрытого соединения.

Local attach [4] - JProfiler 12.0.4

Session View Profiling Window Help

Start Center Detach Save Session Snapshot Settings Start Recordings Stop Recordings Start Tracking Run GC

Session Profiling

Press to record CPU data

Telemetries Live Memory Heap Walker CPU Views Call Tree

Если необходимо увидеть именно ту трассировку стека, которая показывает, где было создано соединение, то сначала нужно выполнить профилирование ЦП. Для этого щелкните по пункту **Call Tree** (Дерево вызовов) в левой части окна, затем начинайте запись данных ЦП.

Трассировка стека выводится под таблицей соединений.

Time Line Connections Call Tree Hot Spots Connection Leaks JDBC connections and execution of statements

This view shows all **virtual** connections that have been open for more than 10 seconds. Virtual connections are what you get from connection pools and block a **physical** connection until they are closed.

Connections of type "Unclosed collected" are definite leaks while "Unclosed" connections are strong candidates.

Show virtual connections: All types Filter

Opened At	Open Since	Type	Description	Thread	Class Name
0:49.542 [Jan 8, 2022 4:3...	49.543 ms	Unclosed	jdbch2mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
0:49.574 [Jan 8, 2022 4:3...	49.511 ms	Unclosed	jdbch2mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
0:49.577 [Jan 8, 2022 4:3...	49.508 ms	Unclosed	jdbch2mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
0:49.578 [Jan 8, 2022 4:3...	49.507 ms	Unclosed	jdbch2mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
0:49.580 [Jan 8, 2022 4:3...	49.505 ms	Unclosed	jdbch2mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
0:49.581 [Jan 8, 2022 4:3...	49.504 ms	Unclosed	jdbch2mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
0:49.583 [Jan 8, 2022 4:3...	49.503 ms	Unclosed	jdbch2mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
0:49.584 [Jan 8, 2022 4:3...	49.501 ms	Unclosed	jdbch2mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
0:49.585 [Jan 8, 2022 4:3...	49.500 ms	Unclosed	jdbch2mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...

Stack trace:

```

com.zaxxer.hikari.HikariDataSource.getConnection()
javax.sql.DataSource.getConnection()
com.example.repositories.ProductRepository.findProduct(int)
com.example.repositories.ProductRepository$$FastClassBySpringCGIJB$$69752884.invoke(int, java.lang.Object, java.lang.Object[])
org.springframework.cglib.proxy.MethodInterceptor.intercept(java.lang.Object, java.lang.reflect.Method, java.lang.Object[])
com.example.repositories.ProductRepository$$EnhancerBySpringCGIJB$$de9c90b7.findProduct(int)
com.example.services.PurchaseService.getProductNamesForPurchases()
com.example.controllers.PurchaseController.findPurchasedProductNames()
HTTP: /products
org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run()

```

Рис. 8.11. После подключения профилирования ЦП JProfiler показывает трассировку стека, которая помогает определить, какая часть кода приложения создает утечки соединений

Теперь перейдем непосредственно к коду примера da-ch8-ex1 (см. листинг 8.1). Мы видим, что найденный с помощью трассировки стека метод действительно создает соединение, которое, очевидно, нигде не закрывается. Главная причина возникновения проблемы найдена.

Листинг 8.1. Определение главной причины возникновения проблемы

```
public Product findProduct(int id) throws SQLException {
    String sql = "SELECT * FROM product WHERE id = ?";

    Connection con = dataSource.getConnection();           ❶
    try (PreparedStatement statement = con.prepareStatement(sql)) {
        statement.setInt(1, id);
        ResultSet result = statement.executeQuery();

        if (result.next()) {
            Product p = new Product();
            p.setId(result.getInt("id"));
            p.setName(result.getString("name"));
            return p;
        }
    }

    return null;
}
```

❶ В этой строке создается соединение, которое никогда не закрывается.

Проект da-ch8-ex2 (см. листинг 8.2) содержит исправленный код. После включения операции создания соединения в блок try-with-resources приложение закроеет это соединение в конце блока try, когда оно станет уже ненужным.

Листинг 8.2. Устранение проблемы посредством обеспечения закрытия соединения

```
public Product findProduct(int id) throws SQLException {
    String sql = "SELECT * FROM product WHERE id = ?";

    try (Connection con = dataSource.getConnection();           ❶
        PreparedStatement statement = con.prepareStatement(sql)) {
        statement.setInt(1, id);
        ResultSet result = statement.executeQuery();

        if (result.next()) {
            Product p = new Product();
            p.setId(result.getInt("id"));
        }
    }
}
```

```

        p.setName(result.getString("name"));
        return p;
    }
}
return null;
}

```

- ❶ Соединение объявлено в блоке try-with-resources, закрывающем это соединение в конце блока try.

Можно еще раз профилировать это приложение после внесения корректирующего изменения. Теперь на вкладке **Connections** в JProfiler показано, что создано только одно соединение, а вкладка **Connection Leaks** пуста, и это подтверждает, что проблема действительно устранена (см. рис. 8.12). При тестировании приложения также видно, что можно отправлять несколько запросов в конечную точку /products.

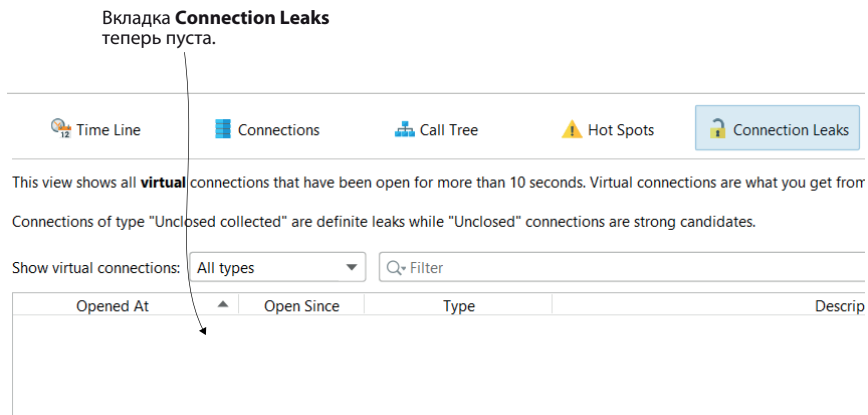
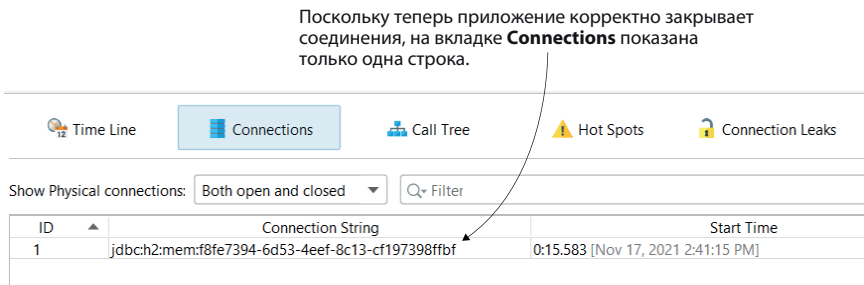


Рис. 8.12. После исправления ошибки мы используем JProfiler, чтобы подтвердить, что утечек соединений больше нет. Мы наблюдаем, как приложение одновременно открывает только одно соединение и корректно закрывает его, когда оно больше не нужно. На вкладке **Connection Leaks** отсутствуют какие-либо другие некорректные соединения

Вы хотите узнать, существует ли наилучшая практическая методика, позволяющая избежать подобных проблем в реальных сценариях? Я рекомендую всем разработчикам уделять около 10 мин после каждой реализации или исправления ошибки, чтобы протестировать функциональность, над которой они работали, с помощью профилировщика. Эта практическая методика может помочь в обнаружении проблем с задержками, созданными некорректными запросами или неправильным управлением соединениями, на ранних этапах разработки.

8.2. Изучение проектного решения кода приложения с использованием графов вызовов

В этом разделе мы рассмотрим одну из наиболее предпочитаемых мною методик изучения проектных решений классов приложения: визуальное представление выполнения в виде графа вызовов. Эта методика особенно полезна при работе с запутанным кодом, который может возникнуть при работе с новым приложением.

До сих пор мы использовали трассировки стека, чтобы изучать выполнение. Трассировки стека выполнения являются весьма важными инструментами, и мы уже видели, как много можно сделать с их помощью. Они полезны, потому что создают простое и понятное текстовое представление, которое можно зафиксировать в журнале (обычно как трассировки стека исключений). Но если оценивать трассировки стека по критерию наглядности, то они не так хороши для быстрого определения отношений между объектами и вызовами методов. Графы вызовов (методов) – это другой способ представления данных, собираемых профилировщиком, который сосредоточен в большей степени на отношениях между объектами и вызовами методов.

Для демонстрации создания графа вызовов мы будем использовать пример `da-ch8-ex2`, приложенный к книге, чтобы показать, как применяются графы вызовов для быстрого определения объектов и методов, взаимодействующих при выполнении, без анализа исходного кода. Разумеется, смысл здесь не в том, что полностью избежать чтения кода, в итоге вам все равно придется копаться в коде, но, предварительно воспользовавшись графами вызовов, вы получите более точную картину всего происходящего.

Для демонстрации мы продолжим использовать профилировщик JProfiler. Поскольку графы вызовов являются способом представления данных о профилировании ЦП, сначала необходимо выполнить этот тип профилирования. На рис. 8.13 показано, как начать профилирование ЦП, результатом которого становится трассировка стека (в JProfiler она обозначена термином «дерево вызовов» (`call tree`)). Мы будем анализировать, что происходит при вызове конечной точки `/products`, предъявляемой приложением.

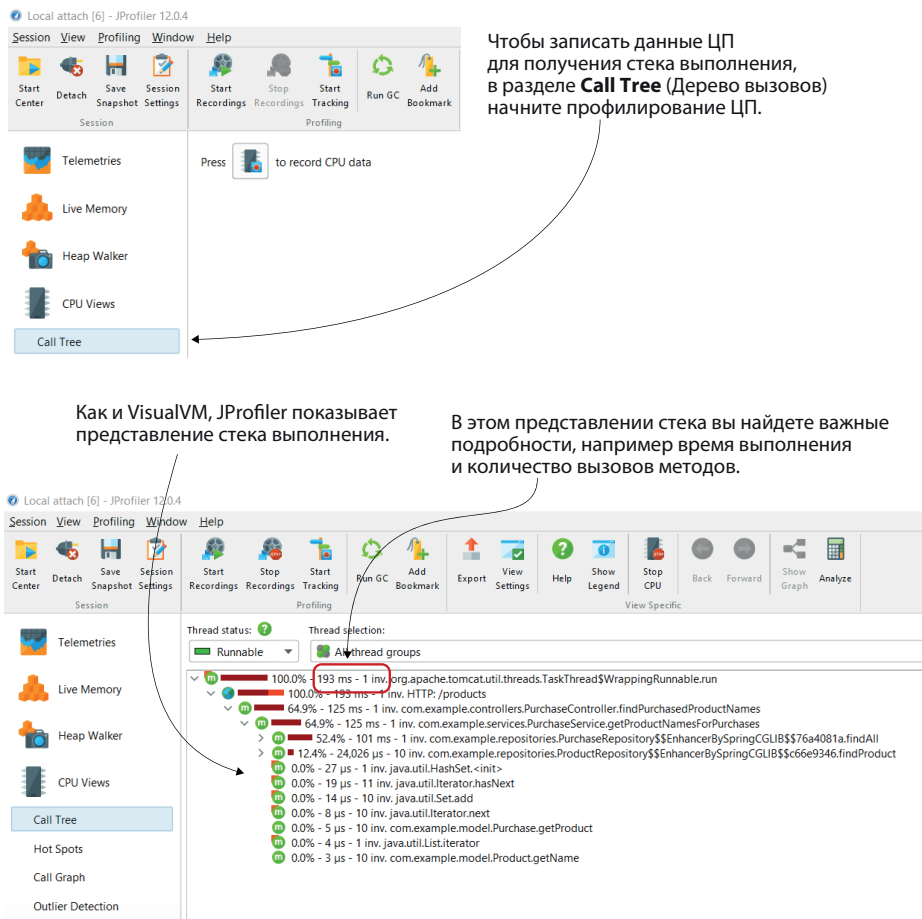


Рис. 8.13. В меню слева выберите пункт **Call Tree** (Дерево вызовов) для начала профилирования ЦП (с записью соответствующих данных). Отправьте запрос в конечную точку /products, и профилировщик сразу покажет записанные данные как трассировку стека, включая подробности о количестве вызовов и времени выполнения

Щелкните правой кнопкой мыши по строке в трассировке стека и в контекстном меню выберите пункт **Show Call Graph** (Показать граф вызовов), чтобы получить визуальное представление данных, собранных о выполнении, в виде графа вызовов (см. рис. 8.14).

JProfiler сгенерирует представление графа вызовов с фокусировкой на методе, указанном в строке, которую вы выбрали при генерации графа вызова. Изначально вам известно только, откуда этот метод вызывается и что он вызывает. Можно продолжить перемещение и проследить всю цепочку вызовов (см. рис. 8.15). Граф вызовов также предоставляет подробности о времени выполнения и количестве вызовов, но сосредоточен он главным образом на отношениях между объектами и вызовами методов.

Чтобы получить представление графа вызовов по данным трассировки стека выполнения, щелкните правой кнопкой мыши по имени любого метода и выберите пункт **Show Call Graph**.

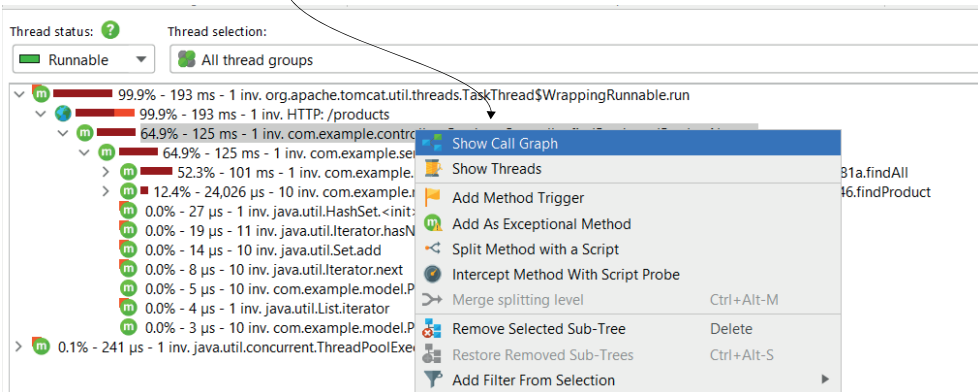


Рис. 8.14. Для создания представления графа вызовов из данных трассировки стека выполнения щелкните правой кнопкой мыши по строке в трассировке стека и выберите пункт **Show Call Graph** (Показать граф вызовов)

Метод `findPurchasedProductNames()` из класса `PurchaseController` вызывается непосредственно из потока, созданного за пределами кодовой базы приложения.

Метод `findPurchasedProductNames()` вызывает метод `getProductNamesForPurchases()` из класса `PurchaseService`.



Рис. 8.15. Граф вызовов показывает выполнение, сосредоточившись главным образом на отношениях между объектами и вызовами методов.

Можно прямолинейно перемещаться по цепочке выполнения (вызовов) методов, чтобы определить, откуда был вызван каждый метод и что он вызывает. Граф вызовов также показывает объекты и методы, являющиеся частью кодовой базы приложения, а также методы библиотек и фреймворков, используемых приложением

8.3. Использование flame-графиков для обнаружения проблем с производительностью

Другой способ визуализации профилируемого выполнения – использование flame-графика. Если графы вызовов главное внимание уделяют отношениям между объектами и вызовами методов, то flame-графики наиболее полезны при идентификации потенциальных задержек. Это просто другой способ увидеть те же подробности, которые предоставляет стек выполнения методов, но, как было отмечено во вводной части этой главы, различные представления одних и тех же данных могут помочь найти нужную информацию.

Для этой демонстрации мы продолжим рассмотрение примера da-ch8-ex2. Воспользуемся JProfiler для изменения представления стека выполнения на flame-график и обсудим преимущества нового представления.

После генерации дерева вызовов, описанного в разделах 8.1 и 8.2, можно изменить его на flame-график, используя пункт главного меню **Analyze** (Анализ), как показано на рис. 8.16.

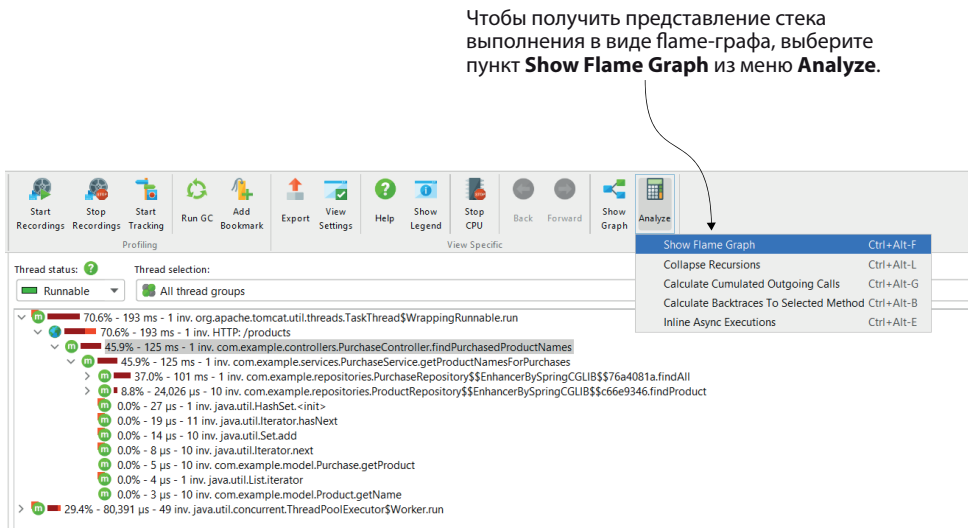


Рис. 8.16. Для замены стека выполнения (дерева вызовов) на flame-график щелкните по кнопке **Analyze** в главном меню, затем выберите пункт **Show Flame Graph** (Показать flame-график)

Flame-график – это способ представления дерева выполнения (вызовов) в виде стопки (штабеля). Такое необычное имя он получил, потому что обычно график похож на пламя (flame). Первым в этой стопке является метод, выполняемый самым первым в потоке. Затем каждый вышележащий уровень совместно используется методами, вызываемыми с более низкого уровня. На рис. 8.17 показан flame-график, созданный для дерева выполнения, изображенного на рис. 8.16.



Рис. 8.17. Flame-график – это представление трассировки выполнения в виде стопки. На каждом уровне показаны методы, вызываемые с более низкого уровня. Первый (самый нижний) уровень стопки – это начало работы потока. При таком представлении мы наблюдаем стек выполнения по вертикали, а по горизонтали на графике отображается время, затраченное на каждом уровне по сравнению с нижележащим уровнем

Любой метод может вызывать несколько других методов. На flame-графике вызванные методы будут показаны на одном уровне. В этом случае длина каждого метода соответствует затраченному времени относительно вызвавшего их метода (на нижележащем уровне). На рис. 8.17 можно видеть, что метод `findById()` из класса `ProductRepository` и метод `findAll()` из класса `PurchaseRepository` были вызваны из метода `getProductNamesForPurchases()` из класса `ProductService`.

На рис. 8.18 мы замечаем, что `getProductNamesForPurchases()` из класса `ProductService` является нижележащим уровнем для обоих методов `findById()` из класса `ProductRepository` и `findAll()` из класса `PurchaseRepository`. Кроме того, `findById()` и `findAll()` используют совместно один и тот же уровень. Но при этом следует отметить, что они имеют разную длину. Длина указана в отношении ко времени выполнения вызывающего метода, поэтому в данном случае время выполнения `findById()` меньше времени выполнения `findAll()`.

Вероятно, вы уже заметили, что в этом графике легко заблудиться. А ведь это всего лишь простой пример для учебных целей. В реальном промышленном приложении flame-график может быть гораздо более сложным. Чтобы как-то уменьшить сложность, можно использовать JProfiler для раскраски в разные цвета уровней на основе имен методов, классов или пакетов. На рис. 8.19 показано, как использовать цветовые обозначения для маркировки конкретных уровней на flame-графиках. Для добавления правил цветовых обозначений используется пункт главного меню **Colorize** (Выделение цветом). Можно добавить несколько правил цветовых обозна-

чений для определения уровней, которые должны быть выделены цветом, и предпочитаемого цвета.

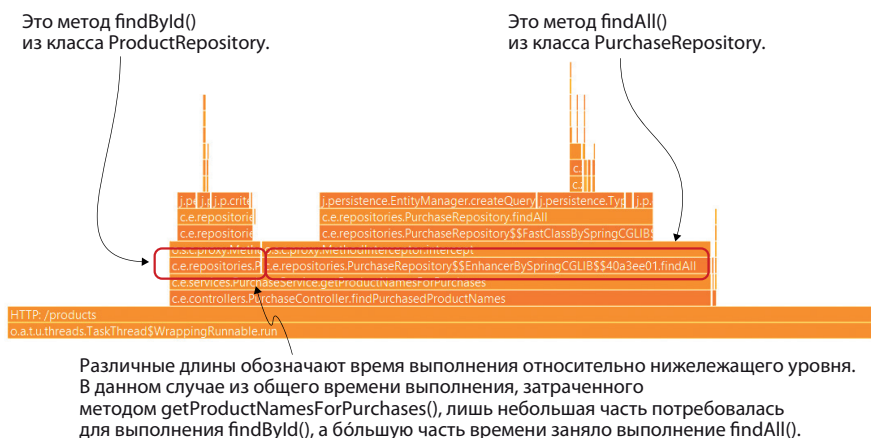


Рис. 8.18. Если несколько методов находятся на одном уровне, значит, все они вызваны методом с нижележащего уровня. Сумма длин их представления равна длине метода под ними. Длина каждого метода – это представление времени его выполнения по отношению к общему времени. В данном случае findAll() затрачивает намного больше времени на выполнение, чем findById()

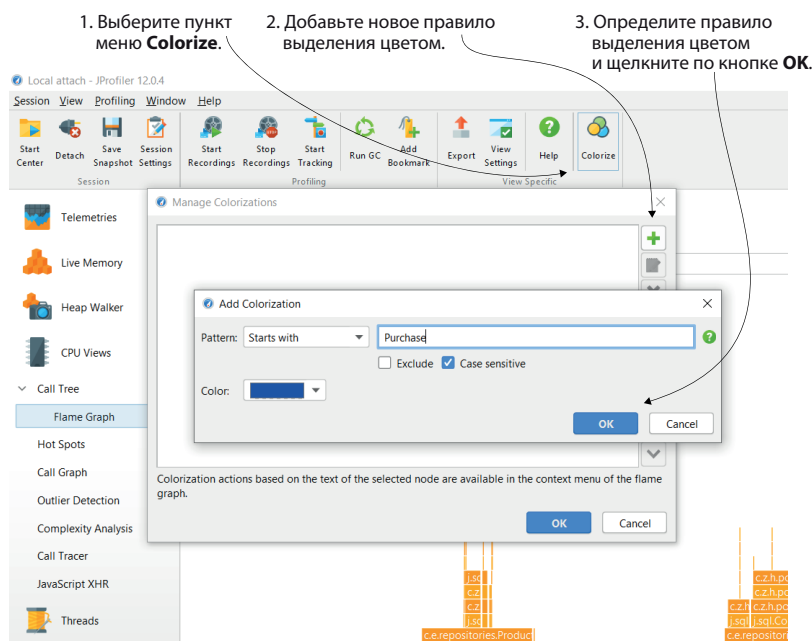


Рис. 8.19. Чтобы сделать flame-график цветным и более удобным для чтения, добавьте правила выделения цветом, используя пункт главного меню **Colorize**. Правила определяют, какие уровни на flame-графике должны быть выделены цветом и какой цвет следует использовать

После установки сервера можно начать выполнение проекта `da-ch8-ex3`. К этому процессу также подключается JProfiler. Чтобы начать мониторинг событий MongoDB, в левой панели меню под заголовком **Databases** (Базы данных) выберите раздел **MongoDB** и начните запись. Чтобы наблюдать, как JProfiler представляет события, вызовем две конечные точки, предъявленные приложением. Обе конечные точки можно вызвать, используя команды `cURL` (как показано в приведенном ниже фрагменте) или с помощью инструментального средства, например Postman:

```
curl -XPOST http://localhost:8080/product/Beer ❶
```

```
curl http://localhost:8080/product ❷
```

❶ Добавление товара с наименованием «Beer» в базу данных.

❷ Получение всех товаров из базы данных.

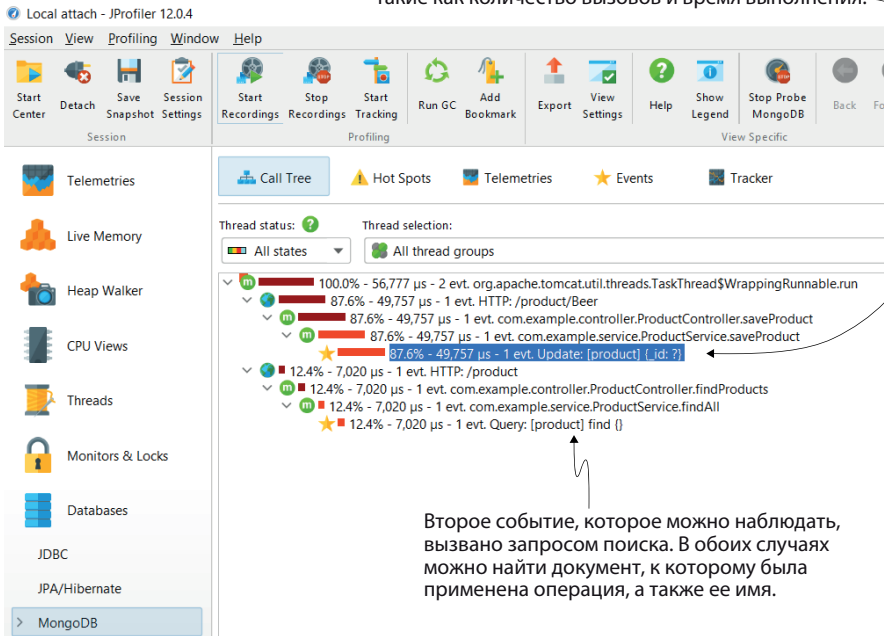
На рис. 8.21 показаны два события, перехваченные JProfiler. Профилировщик показывает трассировки стека (дерева вызовов), связанные с каждым событием. Мы получаем подробную информацию о количестве вызовов и времени выполнения.

8.5. Резюме

- Бесплатные инструментальные средства, такие как VisualVM, предоставляют множество виджетов, которые могут помочь при любом анализе. Но лицензионные инструменты, например JProfiler, могут сделать анализ еще эффективнее, обеспечивая разнообразные способы представления анализируемых данных.
- Иногда в приложениях возникают проблемы при установлении соединения с СУБД. Применяя JProfiler, можно быстрее и проще обнаружить проблему в JDBC-соединении с сервером реляционной базы данных. Можно узнать, остаются ли соединения открытыми, и определить часть кода, которая «забыла» закрыть их.
- Графы вызовов – это альтернативный способ визуализации стека выполнения, который сосредоточен главным образом на отношениях между объектами и вызовами методов. Поэтому графы вызовов являются превосходным инструментом, который можно использовать, чтобы быстрее понять проектное решение классов во время выполнения приложения.
- Flame-графики предоставляют другое визуальное представление профилируемых данных. Flame-графики можно использовать для ускоренного выявления областей кода, создающих задержки выполнения и длинные трассировки стека. Отдельные уровни flame-графика можно выделять цветом для улучшения визуального представления выполнения.

- Некоторые лицензионные инструментальные средства предоставляют расширенные функциональные возможности, такие как анализ обмена данными между приложениями и сервером базы данных NoSQL.

При активизации конечной точки, генерирующей обновление в базе данных, можно обнаружить событие, перехваченное в JProfiler. Вы получаете трассировку стека (дерево вызовов) и полезные подробности, связанные с этими операциями, такие как количество вызовов и время выполнения.



Второе событие, которое можно наблюдать, вызвано запросом поиска. В обоих случаях можно найти документ, к которому была применена операция, а также ее имя.

Рис. 8.21. JProfiler может перехватывать операции, применяемые приложением в NoSQL базе данных. В рассматриваемом здесь примере JProfiler перехватывает два события: обновление данных и чтение документа с именем «product».

Таким образом, можно выполнять мониторинг взаимодействия приложения с NoSQL базой данных и учитывать количество вызовов конкретных операций и время их выполнения. Профилировщик также показывает полную трассировку стека для конкретной операции, поэтому можно быстро найти код, являющийся причиной наступления рассматриваемого события

Глава 9

Анализ блокировок в многопоточных архитектурах

Темы:

- мониторинг потоков приложения;
- обнаружение блокировок потоков и выявление причин их возникновения;
- анализ потоков, находящихся в состоянии ожидания.

В этой главе мы рассматриваем методики анализа выполнения приложений с многопоточными архитектурами. Как правило, разработчики считают реализацию многопоточных архитектур одной из самых сложных тем в разработке приложений, добавляющей еще одну степень сложности при выполнении приложения. Методики, обсуждаемые в этой главе, предоставят возможность заглянуть внутрь выполняемых многопоточных приложений, что позволит облегчить обнаружение проблем и оптимизировать выполнение приложения.

Для полноценного изучения содержимого этой главы необходимы знания об основах механизмов многопоточности в Java, а также о состоянии потоков и синхронизации. Чтобы освежить эти темы в памяти, прочитайте приложение D. В нем не содержится всей возможной информации о потоках и режиме параллельного выполнения в Java (для этого потребуются отдельный комплект книг), но это приложение предоставляет достаточно подробностей для того, чтобы в полной мере освоить материал текущей главы.

9.1. Мониторинг потоков с целью обнаружения блокировок

В этом разделе рассматриваются блокировки потоков и способы их анализа с целью выявления возникающих проблем или возможностей оптимизации выполнения приложения. Блокировки потоков (thread locks) возникают при различных подходах к синхронизации потоков, обычно реализуемых для управления потоком событий (flow of events) в многопоточной архитектуре. Ниже приведены характерные примеры:

- потоку необходимо запретить другим потокам доступ к некоторому ресурсу в то время, когда он изменяет этот ресурс;
- поток должен ждать, когда другой поток завершится или достигнет определенной точки своего выполнения, прежде чем появится возможность продолжить работу.

Блокировки потоков необходимы, они помогают приложению управлять потоками. Но при реализации синхронизации потоков возникает чрезвычайно много потенциальных возможностей для совершения ошибок. Неправильно реализованные блокировки могут привести к полной остановке работы приложения или к проблемам производительности. Необходимо использовать профилировщики для полной уверенности в том, что конкретная реализация оптимальна, а также для того, чтобы сделать приложение более эффективным, минимизируя время блокировок.

В этом разделе мы будем использовать небольшое приложение (проект da-ch9-ex1), в котором реализована простая многопоточная архитектура. Мы применим профилировщик для анализа блокировок во время выполнения приложения. Необходимо обнаружить блокировки потоков и проанализировать их поведение:

- какой поток блокирует другой поток;
- сколько раз блокируется поток;
- время, в течение которого поток приостанавливается, а не выполняется.

Эти подробности позволяют понять, является ли выполнение приложения оптимальным и существуют ли способы улучшения его выполнения. В приложении, используемом в этом примере, реализованы два потока, работающие параллельно: производитель (producer) и потребитель (consumer). Производитель генерирует случайные значения и добавляет их в экземпляр списка, а потребитель удаляет значения из того же набора, который использует производитель (см. рис. 9.1).

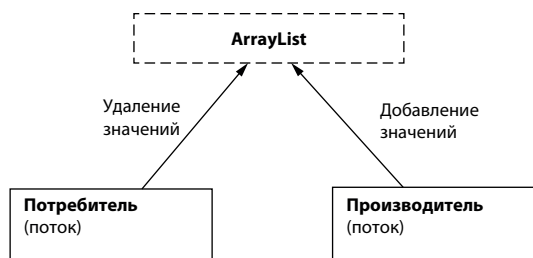


Рис. 9.1. Приложение активизирует два потока, которые мы обозначаем как «производитель» и «потребитель». Оба потока используют общий ресурс: они изменяют экземпляр списка типа `ArrayList`. Производитель генерирует случайные значения и добавляет их в список, а потребитель одновременно удаляет значения, добавленные производителем

Рассмотрим реализацию этого приложения в листингах 9.1, 9.2 и 9.3, чтобы узнать, чего можно ожидать при анализе его выполнения. В листинге 9.1 содержится класс `Main`, активизирующий два экземпляра потоков. Я ввел в приложение 10-секундный интервал ожидания перед активизацией потоков, чтобы обеспечить некоторое время для запуска профилировщика и наблюдения полного графика работы потоков. В приложении потоки получают имена `_Producer` и `_Consumer`, позволяющие с легкостью идентифицировать их при работе с профилировщиком.

Листинг 9.1. Класс `Main` приложения, активизирующий два потока

```

public class Main {

    private static Logger log = Logger.getLogger(Main.class.getName());

    public static List<Integer> list = new ArrayList<>();

    public static void main(String[] args) {
        try {
            Thread.sleep(10000);           ❶

            new Producer("_Producer").start();  ❷
            new Consumer("_Consumer").start();  ❸
        } catch (InterruptedException e) {
            log.severe(e.getMessage());
        }
    }
}
  
```

- ❶ В начале выполнения приложение ждет 10 с, чтобы позволить программисту начать профилирование.
- ❷ Активизация потока производителя.
- ❸ Активизация потока потребителя.

В листинге 9.2 показана реализация потока потребителя. Этот поток повторяет проход по блоку кода миллион раз (это достаточное число для того, чтобы приложение работало несколько секунд и позволило использовать профилировщик для сбора некоторой статистики). Во время каждой итерации поток использует статический экземпляр списка, объявленный в классе Main. Поток-потребитель проверяет, содержатся ли в списке значения, и удаляет первое значение в этом списке. Весь блок кода, реализующий логику, синхронизирован с использованием самого экземпляра списка в качестве монитора. Монитор не позволяет нескольким потокам одновременно входить в синхронизированные блоки, которые он защищает.

Листинг 9.2. Определение потока-потребителя

```
public class Consumer extends Thread {

    private Logger log = Logger.getLogger(Consumer.class.getName());

    public Consumer(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 1_000_000; i++) {
            synchronized (Main.list) {
                if (Main.list.size() > 0) {
                    int x = Main.list.get(0);
                    Main.list.remove(0);
                    log.info("Consumer " + Thread.currentThread().getName() +
                        " removed value " + x);
                }
            }
        }
    }
}
```

- ❶ Итеративный проход миллион раз по синхронизированному блоку кода потребителя.
- ❷ Синхронизация блока кода с использованием в качестве монитора статического списка, определенного в классе Main.
- ❸ Попытка потребления значения, только если список не пустой.
- ❹ Запись в журнал удаленного значения.
- ❺ Потребление первого значения в списке и его удаление.

В листинге 9.3 представлена реализация потока-производителя, весьма похожая на реализацию потребителя. Производитель также выполняет итеративный проход по блоку кода миллион раз. На каждой итерации про-

изводитель генерирует случайное значение и добавляет его в список, объявленный как статический в классе Main. Список тот же самый, из которого потребитель удаляет значения. Производитель добавляет новые значения, только если размер списка не превышает 100.

Листинг 9.3. Определение потока-производителя

```
public class Producer extends Thread {

    private Logger log = Logger.getLogger(Producer.class.getName());

    public Producer(String name) {
        super(name);
    }

    @Override
    public void run() {
        Random r = new Random();
        for (int i = 0; i < 1_000_000; i++) {      ❶
            synchronized (Main.list) {           ❷
                if (Main.list.size() < 100) {     ❸
                    int x = r.nextInt();
                    Main.list.add(x);
                    log.info("Producer " + Thread.currentThread().getName() + ❹
                        « added value « + x);      ❺
                }
            }
        }
    }
}
```

- ❶ Итеративный проход миллион раз по синхронизированному блоку кода производителя.
- ❷ Синхронизация блока кода с использованием в качестве монитора статического списка, определенного в классе Main.
- ❸ Генерация нового случайного значения и добавление его в список.
- ❹ Значение добавляется, только если в списке меньше 100 элементов.
- ❺ Запись в журнал значения, добавленного в список.

Логика производителя также синхронизирована с использованием в качестве монитора того же списка. Таким образом, только один из потоков, производитель или потребитель, может изменять этот список в любой момент времени. Монитор (экземпляр списка) позволяет одному из потоков войти в блок своей логики и заставляет другой поток ждать в начале своего блока кода до тех пор, пока первый поток не закончит выполнение синхронизированного блока (см. рис. 9.2).

Производитель
(поток)

```
@Override
public void run() {
    Random r = new Random();
    for (int i = 0; i < 1_000_000; i++) {

        synchronized (Main.list) {
            if (Main.list.size() < 100) {
                int x = r.nextInt();
                Main.list.add(x);
                log.info("Producer " +
                    Thread.currentThread().getName() +
                    " added value " + x);
            }
        }
    }
}
```

Потребитель
(поток)

```
@Override
public void run() {
    for (int i = 0; i < 1_000_000; i++) {

        synchronized (Main.list) {
            if (Main.list.size() > 0) {
                int x = Main.list.get(0);
                Main.list.remove(0);
                log.info("Consumer " +
                    Thread.currentThread().getName() +
                    " removed value " + x);
            }
        }
    }
}
```

Если производитель выполняет свой синхронизированный блок (закрашенный прямоугольник), то потребитель не может получить доступ к своему синхронизированному блоку. Потребитель ждет, когда монитор позволит ему войти в свой синхронизированный блок.

Рис. 9.2. В любой момент времени только один поток может находиться в синхронизированном блоке. Либо производитель выполняет логику, определенную в своем методе `run()`, либо потребитель выполняет свою логику

Можем ли мы обнаружить это поведение приложения и прочие подробности выполнения, используя профилировщик? В реальном промышленном приложении код может быть гораздо более сложным, поэтому понимания того, что делает приложение, при простом чтении кода в большинстве случаев может оказаться недостаточно.



ПРИМЕЧАНИЕ. Следует помнить, что проекты, используемые в этой книге, упрощены и адаптированы для обучения. Не воспринимайте их как наилучшие практические подходы для применения в реальных производственных приложениях.

Воспользуемся VisualVM, чтобы посмотреть, как все это выглядит на вкладке мониторинга потоков **Threads** (см. рис. 9.3). Обратите внимание на различное выделение цветом (закраску), поскольку большая часть кода для каждого потока синхронизирована. В основном либо производитель работает, а потребитель ожидает, либо выполняется потребитель, а производитель находится в состоянии ожидания.

Эти два потока редко могут выполнять код одновременно. Поскольку существуют инструкции, находящиеся за пределами синхронизированного блока, такой код может быть выполнен одновременно обоими потоками. Примером является цикл `for`, в обоих случаях определенный вне синхронизированного блока.

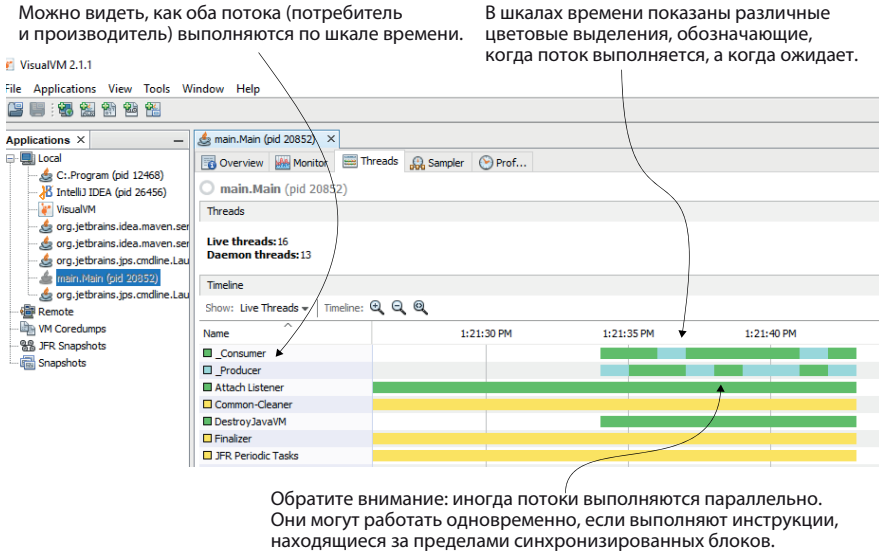


Рис. 9.3. В основном потоки последовательно блокируют друг друга и выполняют собственные синхронизированные блоки кода. Оба потока сохраняют возможность одновременного выполнения инструкций, находящихся за пределами синхронизированного блока

Поток может быть заблокирован синхронизированным блоком кода и ожидать, когда другой поток завершит свое выполнение (присоединение – joining), или он может управляться блокирующим объектом. В случаях, когда поток остановлен и не может продолжать свое выполнение, мы говорим, что поток заблокирован (locked). На рис. 9.4 можно видеть ту же информацию, представленную в JProfiler, работающем по методикам, которые мы использовали.

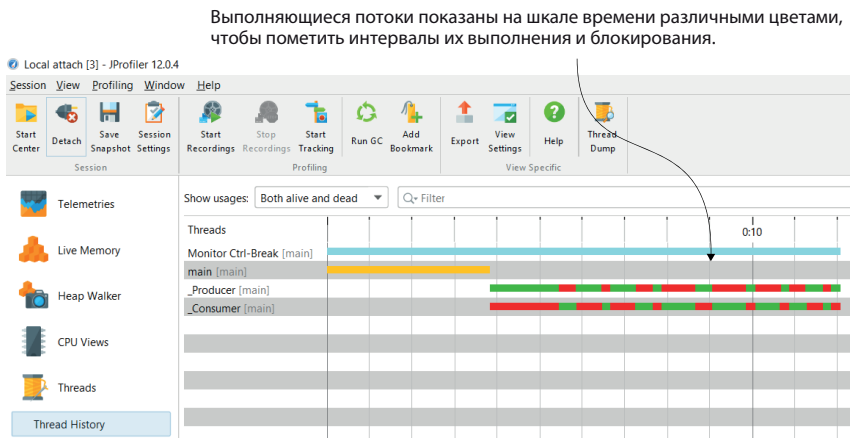


Рис. 9.4. Вместо VisualVM можно использовать другие профилировщики. Здесь можно видеть, как показаны шкалы времени выполнения потоков в JProfiler

9.2. Анализ блокировок потоков

При работе с архитектурой приложения, использующей блокировки потоков, необходима полная уверенность в том, что приложение реализовано оптимально. Для этого требуется способ идентификации блокировок, позволяющий определить количество блокировок потоков и время блокирования. Также необходимо понять, что заставляет поток входить в режим ожидания в конкретных сценариях. Можно ли каким-то образом собрать всю эту информацию? Да, профилировщик может сообщить все, что необходимо для понимания поведения потока.

Продолжим, применяя действия, подробно описанные в главе 7, для анализа методом профилирования:

- 1) использование выборки, чтобы понять на высоком уровне, что происходит во время выполнения, и определить, где нужно получить более подробную информацию;
- 2) использование профилирования (инструментовки) для выяснения подробностей о конкретном элементе, который необходимо проанализировать.

На рис. 9.5 показаны результаты выборки выполнения приложения. Наблюдая за временами выполнения, мы замечаем, что общее время выполнения больше, чем общее время использования ЦП. В главе 7 мы видели похожую ситуацию и определили: это означает, что приложение чего-то ожидает.

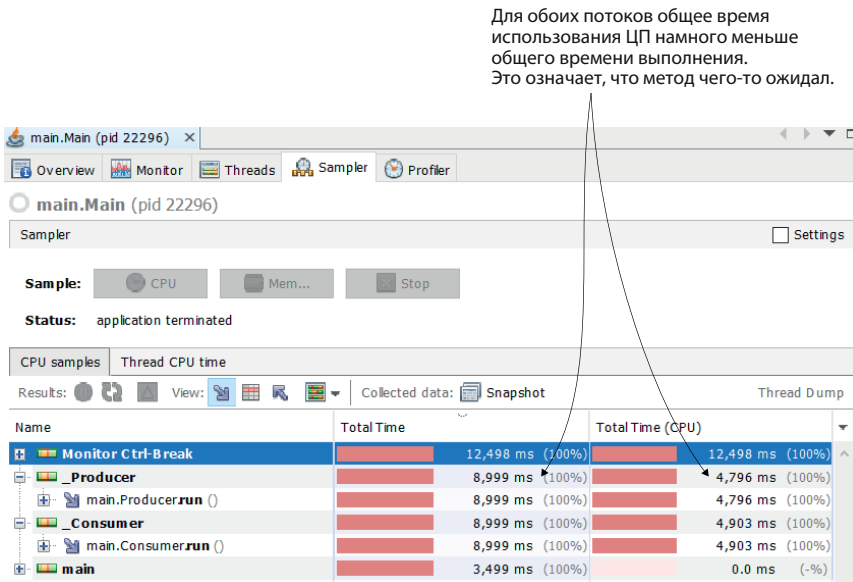


Рис. 9.5. Если время использования ЦП меньше общего времени выполнения, это означает, что приложение чего-то ожидает. Необходимо определить, чего именно ожидает приложение и можно ли оптимизировать это время ожидания

На рис. 9.6 можно видеть кое-что интересное: метод ожидает, но, как показано в данных выборки, он не ждет кого-то другого. Кажется, что он просто ждет самого себя. Строка, помеченная как «Self time», сообщает, сколько времени заняло выполнение метода. Обратите внимание: метод затратил всего лишь около 700 мс времени ЦП как self time, но гораздо больше времени – 4903 мс – как общее время собственного выполнения.

Обратите внимание: метод не ждет чего-то извне. Его время собственного выполнения весьма длительное, хотя время использования ЦП очень короткое.

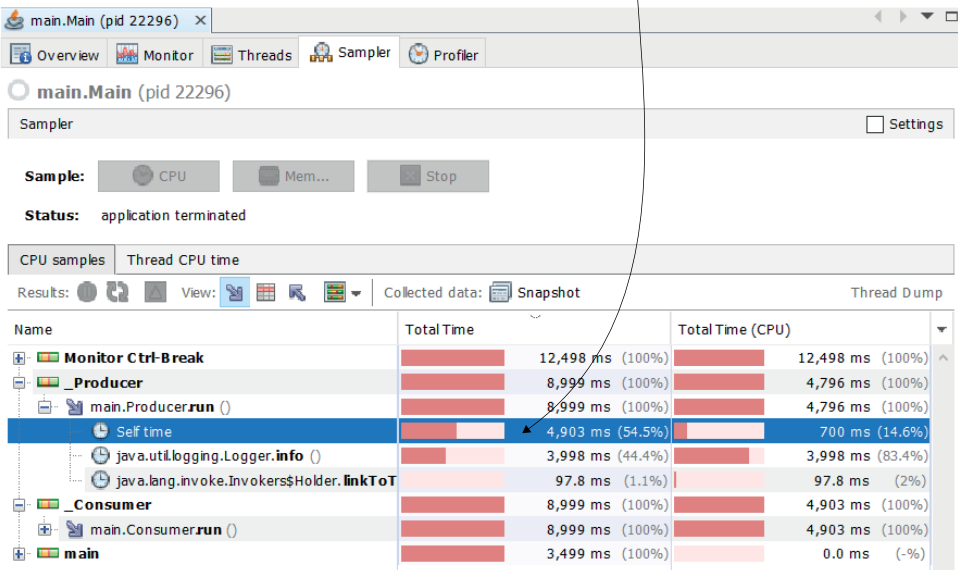


Рис. 9.6. Метод не ждет кого-то другого, он ждет самого себя. Мы видим, что это время собственного выполнения больше, чем время использования ЦП, а это обычно означает, что поток заблокирован. Этот поток мог быть заблокирован другим потоком

В главе 7 мы работали с примером, в котором приложение ожидало ответа внешнего сервиса. Приложение отправляло вызов, а затем ожидало ответа на него другого сервиса. В том случае причина ожидания приложения была вполне объяснимой, но здесь ситуация выглядит необычной. Что могло стать причиной такого поведения?

Возможно, вы удивленно спрашиваете: «Как метод может ждать самого себя? Он слишком ленив, чтобы работать?» Когда наблюдается такое поведение, при котором метод находится в состоянии ожидания, но не ждет чего-то извне, вероятнее всего, соответствующий поток был заблокирован. Чтобы получить более подробную информацию о том, что стало причиной блокировки потока, необходимо применить углубленную методику анализа с использованием профилирования выполнения.

Выборка не ответила на все эти вопросы. Можно видеть, что методы находятся в состоянии ожидания, но неизвестно, чего именно они ждут. Необходимо продолжить расследование с применением профилирования (инструментовки) для получения более подробной информации. В VisualVM мы используем вкладку **Profiler** для начала мониторинга блокировок. Чтобы начать профилирование блокировок, щелкните по кнопке **Locks**, как показано на рис. 9.7, и получите представленный здесь же результат профилирования. На рис. 9.7 кнопка **Locks** неактивна, потому что процесс уже был остановлен в конце сеанса профилирования.

Чтобы начать профилирование данных о блокировках, щелкните по кнопке **Locks**. После завершения сеанса профилирования кнопка становится неактивной (запрещенной).

Можно видеть, что потоки были многократно заблокированы. Для каждого потока показано более 3600 блокировок во время выполнения.

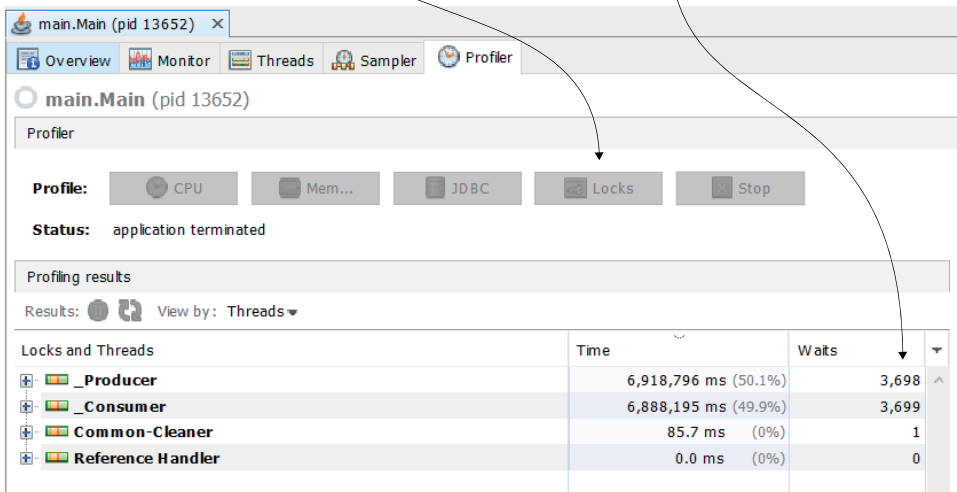
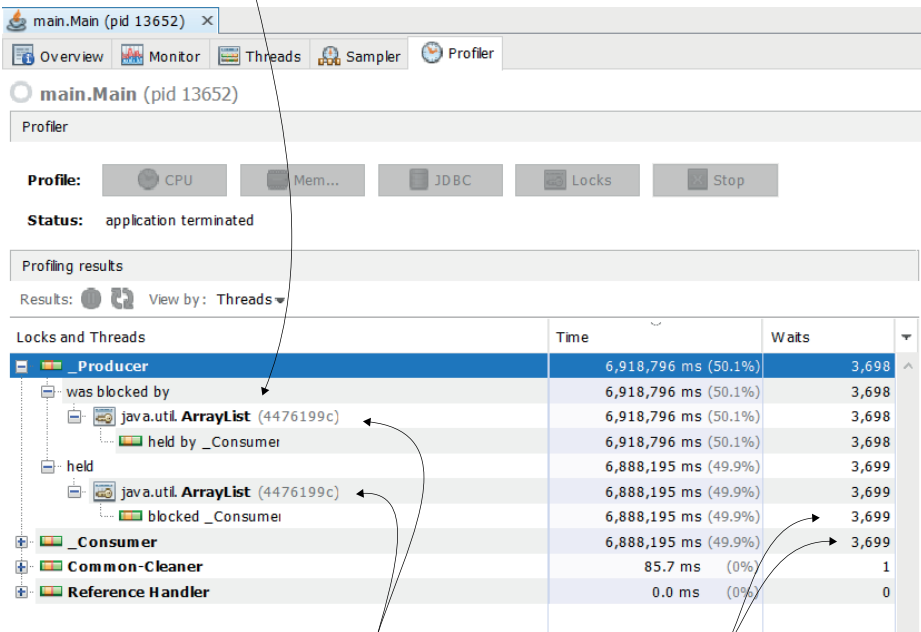


Рис. 9.7. Чтобы начать профилирование блокировок, щелкните по кнопке **Locks** на вкладке **Profiler**. После завершения сеанса профилирования мы видим более 3600 блокировок в каждом из потоков производителя и потребителя

По каждому потоку можно получить более подробную информацию, щелкнув по маленькому значку плюс (+) слева от его имени. Теперь можно узнать детали о каждом объекте-мониторе, который повлиял на выполнение потока. Профилировщик показывает подробности о потоках, которые были заблокированы другим потоком, а также о том, что именно заблокировало конкретный поток.

Все эти подробности можно наблюдать на рис. 9.8. Мы видим, что поток производителя был заблокирован экземпляром монитора типа ArrayList. Ссылка на этот объект (4476199c на рис. 9.8) помогает однозначно идентифицировать экземпляр объекта, чтобы узнать, что один и тот же монитор воздействовал на несколько потоков. Это также позволяет точно определить отношение между потоками и монитором.

Здесь мы обнаруживаем объекты (мониторы), которые стали причиной блокировки потоков, а также мониторы, выделенные для использования потоком.



В данном случае мы видим, что один и тот же объект (экземпляр типа ArrayList), который заблокировал анализируемый поток, также удерживается этим потоком.

Обратите внимание: число блокировок, выполненных производителем для потребителя, равно общему числу блокировок для потребителя, а это означает, что только производитель блокирует потребителя.

Рис. 9.8. Результаты профилирования обеспечивают правильное понимание того, что стало причиной создания блокировок и на что они воздействуют. Мы видим, что существует только один монитор, с которым работает поток производителя. Кроме того, поток производителя был заблокирован 3698 раз с использованием этого монитора. Используя тот же экземпляр монитора, производитель заблокировал потребителя почти столько же раз: 3699

Все показанное на рис. 9.8 можно прочесть следующим образом:

- поток с именем `_Producer` был заблокирован экземпляром монитора со ссылкой 4476199c – экземпляром типа `ArrayList`;
- поток с именем `_Consumer` блокировал поток `_Producer` 3698 раз при попытке получения монитора 4476199c;
- поток производителя также удерживал монитор (становился его владельцем) со ссылкой 4476199c 3699 раз, или поток `_Producer` блокировал поток `_Consumer` 3699 раз.

На рис. 9.9 развернута подробная информация о потоке потребителя. Выясняется, что все данные коррелируют. На протяжении всего времени

выполнения только один экземпляр монитора – экземпляр типа `ArrayList` – блокировал то один, то другой поток. В итоге поток потребителя был заблокирован 3699 раз, когда поток производителя выполнял блок, синхронизированный объектом `ArrayList`. Поток производителя блокировался 3698 раз, когда поток потребителя выполнял блок, синхронизированный тем же монитором `ArrayList`.

Оба потока (производитель и потребитель) получали во владение один и тот же монитор и им же блокировались. Это означает, что потоки попеременно блокируют друг друга.

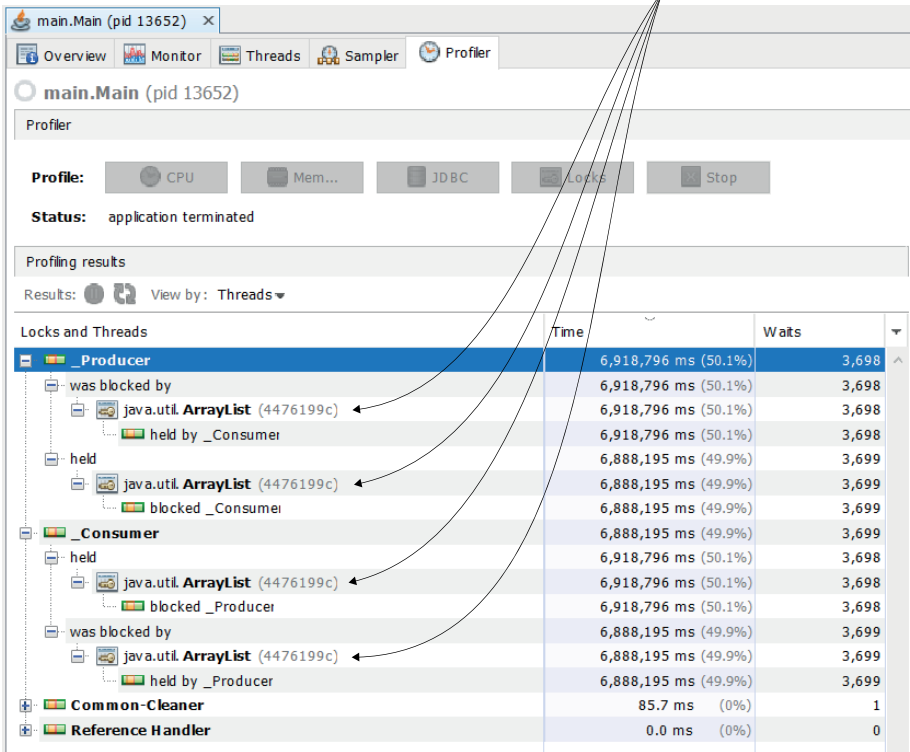


Рис. 9.9. Оба потока используют один и тот же монитор для блокировки друг друга. Пока один поток выполняет блок, синхронизированный экземпляром монитора `ArrayList`, другой ждет. Таким образом, один поток блокируется 3698 раз, другой – 3699 раз



ПРИМЕЧАНИЕ. Следует учесть, что вы не всегда будете получать одинаковые числовые характеристики при выполнении этого приложения на своем компьютере. В действительности, вероятнее всего, характеристики не будут одинаковыми, даже если вы повторите выполнение на том же компьютере. Несмотря на возможность получения отличающихся числовых значений, в целом вы можете выполнить аналогичные наблюдения.

Для демонстрации анализа этого примера я использовал VisualVM, потому что он бесплатный и мне удобно работать с ним. Но вы можете применить ту же методику и с использованием других инструментов, например JProfiler.

После подключения JProfiler к процессу (это описано в главе 8) убедитесь в том, что вы установили для параметра **JVM exit action** (Действие при выходе из JVM) значение **Keep the VM alive for profiling** (Продолжить работу VM для профилирования), как показано на рис. 9.10.

При подключении JProfiler к процессу сконфигурируйте параметр JVM exit action для продолжения работы VM, чтобы можно было наблюдать статистические данные о профилировании после завершения выполнения приложения.

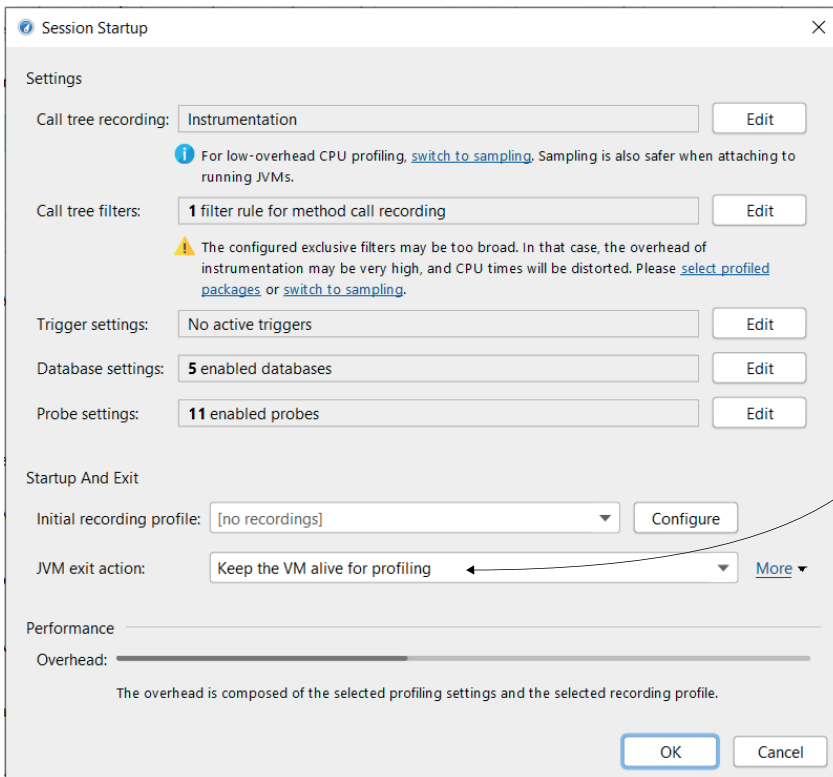


Рис. 9.10. Начиная сеанс профилирования в JProfiler, не забудьте установить для параметра **JVM exit action** значение **Keep the VM alive for profiling**, чтобы можно было наблюдать результаты профилирования после того, как приложение завершит выполнение

JProfiler предлагает несколько представлений для визуализации тех же подробностей, которые мы получили с помощью VisualVM, но результаты одинаковы. На рис. 9.11 показан вид отчета **Monitor History** (Хронология монитора) для блокировок.

Для доступа к хронологии блокировок в JProfiler выберите пункт **Monitor History** под заголовком **Monitors & Locks** на панели меню слева.

JProfiler показывает полную хронологию событий блокировок: продолжительность блокировки, используемый монитор, поток, применяющий блокировку (поток-владелец), заблокированный поток (ожидаящий поток) и точное время наступления события.

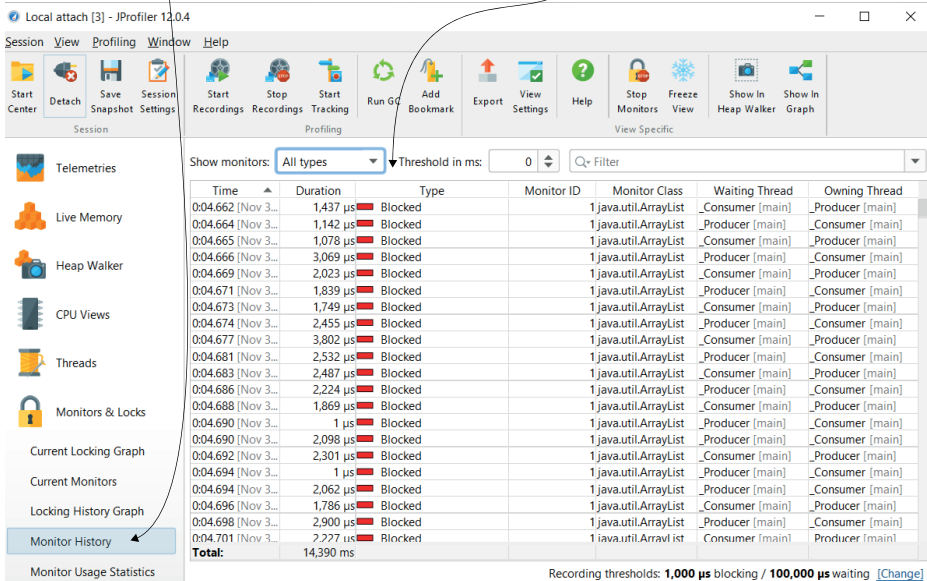


Рис. 9.11. JProfiler выводит подробную хронологию всех блокировок потоков, выполненных в приложении. Профилировщик показывает точное время наступления события, его продолжительность, монитор, который выполнил блокировку, и потоки, участвующие в этом событии

В большинстве случаев такой подробный отчет не нужен. Я предпочитаю группировать события (блокировки) по потокам или реже по монитору. В JProfiler можно группировать события так, как показано на рис. 9.12. На панели меню слева из пункта **Monitor Usage Statistics** (Статистика использования монитора) можно выбрать вариант группирования событий по участвующим в них потокам или по мониторам, создающим блокировки. В JProfiler есть даже весьма необычный вариант, позволяющий группировать блокировки по классам объектов мониторов.

Если вы группируете события блокировок по участвующим в них потокам, то получите статистические данные, очень похожие на предоставленные VisualVM. Каждый поток блокируется более 3600 раз во время выполнения приложения (см. рис. 9.13).

Является ли такое выполнение оптимальным? Чтобы ответить на этот вопрос, необходимо знать предназначение приложения. В данном случае приложение – простой демонстрационный пример, и, поскольку у него нет конкретного реального предназначения, трудно в полной мере проанализировать результаты, чтобы понять, можно ли улучшить это приложение.

В JProfiler можно воспользоваться пунктом **Monitor Usage Statistics** для получения информации о блокировках, сгруппированных по участвующим в них потокам или по мониторам, создающим блокировки.

Для получения статистических данных обо всех перехваченных событиях блокировок, сгруппированных по участвующим в них потокам, выберите пункт **Group by Threads**, затем щелкните по кнопке **OK**.

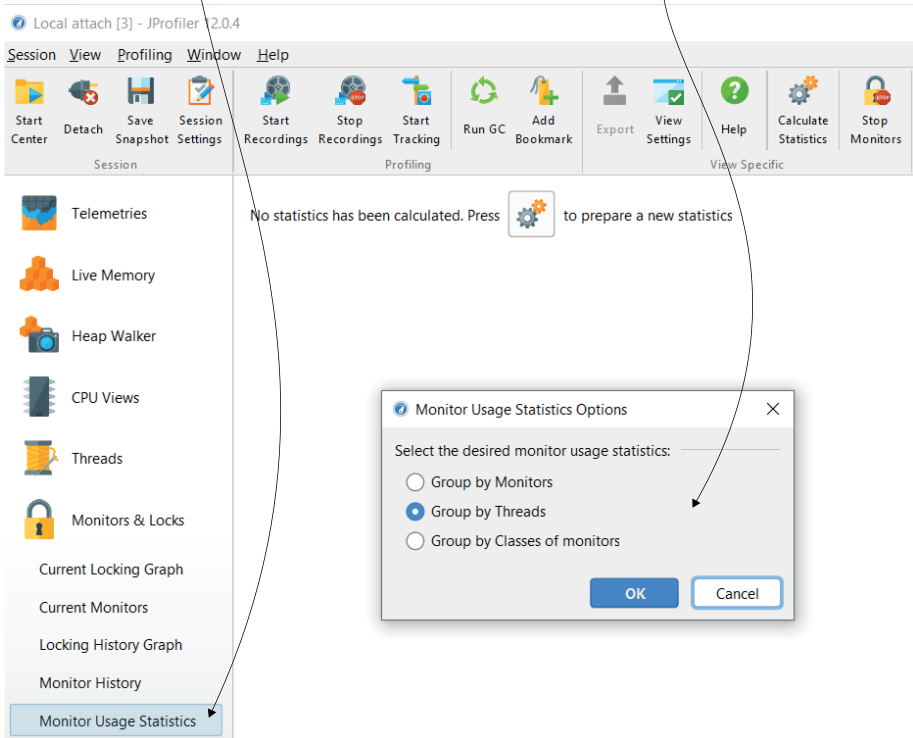


Рис. 9.12. Можно группировать события блокировок по участвующим в них потокам или по мониторам, используя пункт меню **Monitor Usage Statistics**.

Можно воспользоваться объединенным представлением, чтобы понять, на какие потоки воздействие больше и что на них воздействует, или какой монитор чаще всего заставляет потоки останавливаться

Но поскольку приложение создает два потока, которые используют общий ресурс (список), если учесть тот факт, что они не могут одновременно работать с совместно используемым ресурсом, то мы предполагаем следующее:

- общее время выполнения должно быть приблизительно суммой интервалов времени использования ЦП, т. е. интервалов реального выполнения (поскольку потоки не могут работать одновременно, они взаимно исключают друг друга);
- потоки должны иметь почти одинаковое время, выделенное для выполнения, и должны блокироваться приблизительно одинаковое ко-

личество раз. Если один из потоков преобладает, то другой в итоге может зависать, переходить в состояние «голодания» (starvation): ситуация, когда поток блокируется «несправедливым» способом и не может продолжить выполнение.

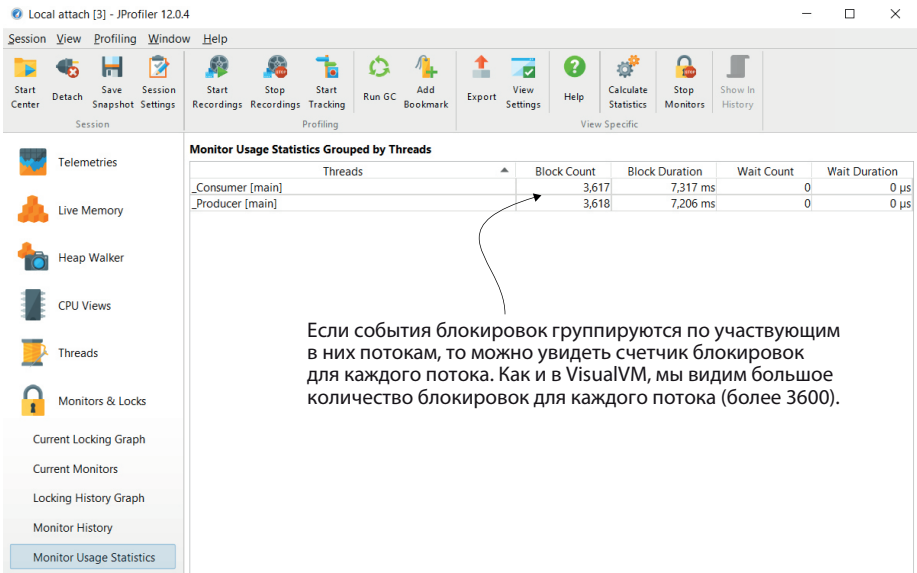


Рис. 9.13. Группирование событий блокировок по потокам создает объединенное представление, показывающее, сколько раз каждый поток был заблокирован во время выполнения

Если еще раз взглянуть на анализ потоков, то можно увидеть, что отношение к обоим потокам справедливое. Количество их блокировок почти одинаковое, они взаимно исключают друг друга, но имеют приблизительно равное время активного выполнения (использования ЦП). Это оптимальный вариант, и мы почти ничего не можем сделать для улучшения. Но следует помнить, что ситуация зависит от того, что именно делает приложение и каковы наши ожидания в отношении того, как оно должно выполняться.

Рассмотрим пример другого сценария, в котором приложение не всегда считается оптимальным. Предположим, что имеется приложение, которое действительно обрабатывает значения. Пусть производителю требуется больше времени для добавления каждого значения в список, чем потребителю для последующей обработки этого значения. В реальном промышленном приложении может возникнуть похожая ситуация: потоки не должны иметь равное «рабочее» время для выполнения своих задач.

В этом случае приложение можно улучшить:

- минимизировать количество блокировок для потребителя и ввести интервал ожидания, чтобы позволить производителю работать больше;

- определить больше потоков-производителей или позволить потребителю считывать и обрабатывать значения пакетами (сразу по нескольким значениям).

Все зависит от того, что именно делает конкретное приложение, но понимание возможностей его улучшения начинается с анализа выполнения. Поскольку никогда не существует единственной методики, которую можно применить ко всем приложениям, я всегда рекомендую разработчикам использовать профилировщик и анализировать изменения выполнения в процессе реализации многопоточного приложения.

9.3. Анализ ожидающих потоков

В этом разделе анализируются потоки, ожидающие некоторого оповещения. Ожидающие потоки отличаются от заблокированных. Монитор блокирует поток для выполнения синхронизированного блока кода. В этом случае мы не ожидаем, что монитор выполнит некоторое специализированное действие, чтобы «сообщить» заблокированному потоку о возможности продолжения его выполнения. Но монитор может заставить поток ждать в течение неопределенного интервала времени и в дальнейшем решить, когда позволить этому потоку продолжить выполнение. После того как монитор перевел поток в состояние ожидания, такой поток возобновляет выполнение только после оповещения от того же монитора. Возможность ввода потока в состояние ожидания до специального оповещения обеспечивает невероятную гибкость в управлении потоками, но также может стать источником проблем при неправильном использовании.

Чтобы получить наглядное представление о различии между заблокированными и ожидающими потоками, посмотрите на рис. 9.14. Представим себе, что синхронизированный блок кода – это ограниченная зона дорожного движения, контролируемая офицером полиции. Потоки – это автомобили. Офицер полиции разрешает только одной машине одновременно проехать через ограниченную зону (синхронизированный блок). Остановленные машины мы называем заблокированными (locked). Офицер полиции также может контролировать машины,двигающиеся внутри ограниченной зоны. Он может приказать машине, проезжающей ограниченную зону, остановиться и ждать явного приказа о продолжении движения. Такие машины мы называем ожидающими (waiting).

Воспользуемся тем же приложением, которое мы анализировали ранее в текущей главе, и рассмотрим следующий сценарий: один из разработчиков этого приложения решил улучшить архитектуру производитель-потребитель. Теперь поток-потребитель ничего не может делать, если список пуст, поэтому многократно повторяет итерации при ложном условии до тех пор, пока JVM не сделает его ожидающим, чтобы позволить потоку-производителю работать и добавлять значения в список. То же самое происходит, когда производитель добавляет 100 значений в список. Поток-производитель



повторяет итерации при ложном условии до тех пор, пока JVM не позволит потребителю удалить несколько значений из списка.

Эти потоки находятся в заблокированном состоянии. Они не могут продолжить выполнение, пока другой поток работает внутри синхронизированного блока. Мы называем эти потоки заблокированными.

Заблокированные потоки

Офицер полиции говорит машинам за пределами синхронизированного блока: «Слушайте внимательно! Вы должны ждать. В настоящее время кто-то другой выполняет синхронизированный блок».

Этот поток работает внутри синхронизированного блока. Монитор (офицер полиции) не позволяет другим потокам войти в синхронизированный блок до тех пор, пока этот поток не выйдет из него.

```
synchronized(  ) {
    _____
    _____
}

```

Ожидающие потоки

Офицер полиции говорит машине внутри синхронизированного блока: «Слушайте внимательно! Вы должны ждать до тех пор, пока я не скажу, что можно продолжать выполнение».

Этот поток работает внутри синхронизированного блока. Монитор (офицер полиции) приостанавливает его и переводит в заблокированное состояние. В соответствии с действиями монитора мы называем этот поток ожидающим.



```
synchronized(  ) {
    _____
    _____
}

```

Рис. 9.14. Сравнение заблокированных потоков с ожидающими. Заблокированный поток останавливается на входе в синхронизированный блок кода. Монитор не разрешает потоку войти в синхронизированный блок, пока другой поток активно работает внутри этого блока. Ожидающий поток – это поток, который монитор явно перевел в состояние блокировки. Монитор может сделать ожидающим любой поток, находящийся внутри управляемого им синхронизированного блока. Ожидающий поток может возобновить работу только после того, как монитор явно сообщит ему о разрешении продолжения выполнения

Можно ли что-то сделать, чтобы переводить потребителя в состояние ожидания при отсутствии потребляемых значений и возобновлять его работу, только если известно, что список содержит как минимум одно значение (см. рис. 9.15)? И наоборот, можно ли переводить производителя в состояние ожидания, когда в списке содержится слишком много значений, и позволять ему работать, только если имеет смысл добавление новых значений? Сделает ли такой подход более эффективным это приложение?



После того как производитель добавил значение в список...



Рис. 9.15. Некоторые машины являются потоками-потребителями, другие – потоками-производителями. Офицер полиции приказывает потребителю подождать, если в списке нет значений для потребления, позволяя производителям работать и добавлять значения. Когда список содержит хотя бы одно потребляемое значение, офицер приказывает ожидающему потребителю продолжить выполнение

Изменим приложение для реализации описанного выше нового поведения, но при этом продемонстрируем, что в таком сценарии приложение не становится более эффективным. Напротив, его выполнение менее оптимально.

Может показаться, что есть неплохое решение: сделать потоки ожидающими, когда они не могут работать с совместно используемым ресурсом (списком). Но после анализа можно понять, что это отрицательно влияет на производительность, а не ускоряет работу приложения.

В листинге 9.4 показана новая реализация потока-потребителя. Он ждет, если список пуст, поскольку потреблять нечего. Монитор переводит поток-потребитель в состояние ожидания и сообщит ему о возможности продолжения выполнения только после того, как производитель что-то добавит в список. Мы используем метод `wait()`, чтобы приказать потребителю

ждать, если список пуст. Аналогично, когда потребитель удаляет значения из списка, он оповещает ожидающие потоки, и если производитель находится в состоянии ожидания, то теперь он знает, что можно продолжить выполнение, потому что список уже не заполнен до отказа. Мы используем метод `notifyAll()` для оповещения ожидающих потоков. Эту реализацию можно найти в проекте `da-ch9-ex2`.



ПРИМЕЧАНИЕ. Я всегда рекомендую использовать профилировщик во время разработки для подтверждения оптимального выполнения приложения.

Листинг 9.4. Перевод потока-потребителя в состояние ожидания, если список пуст

```
public class Consumer extends Thread {

    // Здесь код не показан.

    @Override
    public void run() {
        try {
            for (int i = 0; i < 1_000_000; i++) {
                synchronized (Main.list) {
                    if (Main.list.size() > 0) {
                        int x = Main.list.get(0);
                        Main.list.remove(0);
                        log.info("Consumer " + Thread.currentThread().getName() +
                               " removed value " + x);
                        Main.list.notifyAll();           ❶
                    } else {
                        Main.list.wait();                 ❷
                    }
                }
            }
        } catch (InterruptedException e) {
            log.severe(e.getMessage());
        }
    }
}
```

❶ После потребления элемента из списка потребитель сообщает ожидающим потокам о сделанном изменении в содержимом списка.

❷ Если список пуст, то потребитель ждет до тех пор, пока не получит сообщение о том, что в список был добавлен элемент.

В листинге 9.5 показана реализация потока-производителя. Подобно потоку-потребителю, производитель ждет, если в списке слишком много значений. В конце концов потребитель оповещает производителя и позволяет ему возобновить работу после потребления значения из списка.

Листинг 9.5. Перевод потока-производителя в состояние ожидания, если список заполнен

```
public class Producer extends Thread {

    // Здесь код не показан.

    @Override
    public void run() {
        try {
            Random r = new Random();
            for (int i = 0; i < 1_000_000; i++) {
                synchronized (Main.list) {
                    if (Main.list.size() < 100) {
                        int x = r.nextInt();
                        Main.list.add(x);
                        log.info("Producer " + Thread.currentThread().getName() +
                               " added value " + x);
                        Main.list.notifyAll();           ❶
                    } else {
                        Main.list.wait();                 ❷
                    }
                }
            }
        } catch (InterruptedException e) {
            log.severe(e.getMessage());
        }
    }
}
```

- ❶ После добавления элемента в список производитель сообщает ожидающим потокам о сделанном изменении в содержимом списка.
- ❷ Если список содержит 100 элементов, то производитель ждет до тех пор, пока не получит сообщение об удалении элемента из списка.

Как вы уже знаете, анализ начинается с выборки выполнения. Мы уже видим кое-что подозрительное: выполнение выглядит гораздо более длительным (см. рис. 9.16). Если вернуться к предыдущим наблюдениям, сделанным в разделе 9.1, то можно заметить, что раньше для выполнения всего приложения в целом требовалось всего лишь около 9 с. Теперь приложение выполняется около 50 с – огромное различие.

Выполнение занимает больше времени, и возникает большое различие между общим временем выполнения и общим временем использования ЦП – признак того, что приложение слишком долго находится в состоянии ожидания.

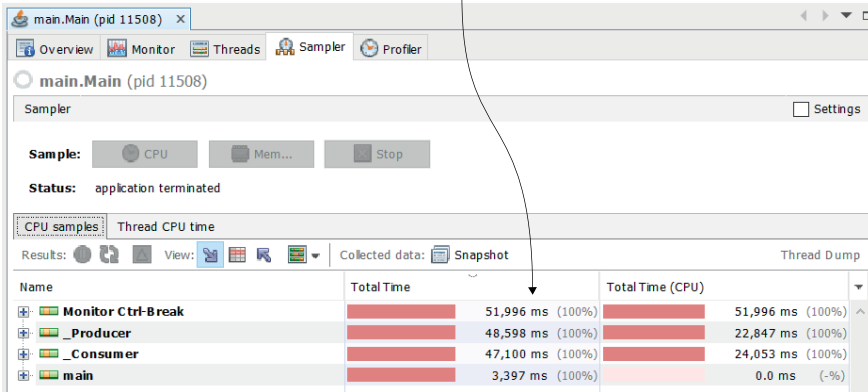


Рис. 9.16. При выборке выполнения можно видеть, что приложение выполняется медленнее, чем раньше, до ввода потоков в состояние ожидания

Подробности выборки (см. рис. 9.17) показывают, что добавленный в приложение метод `wait()` создает большую часть времени ожидания. Поток не блокируется надолго, потому что время его собственного выполнения весьма близко ко времени использования ЦП. И хотя нашей целью остается повышение общей эффективности приложения, тем не менее кажется, что мы всего лишь переместили состояние ожидания из одного места в другое, но при этом замедлили процесс выполнения приложения.

Подробности выполнения показывают, что большую часть времени ожидания создает метод `wait()`, вызываемый монитором.

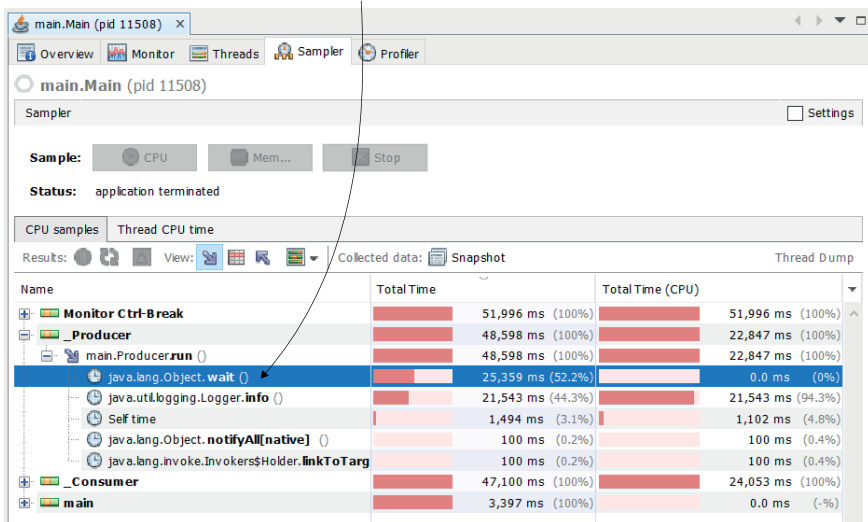


Рис. 9.17. Анализируя подробности, можно заметить, что время собственного выполнения не такое уж большое, но поток блокируется, следовательно, находится в состоянии ожидания в течение более длительного времени

Продолжим профилирование, чтобы получить более подробную информацию (см. рис. 9.18). Разумеется, результаты профилирования показывают несколько блокировок, но это не слишком помогает, так как выполнение происходит намного медленнее.

Обратите внимание: количество блокировок уменьшилось. Но при этом увеличилось общее время выполнения.

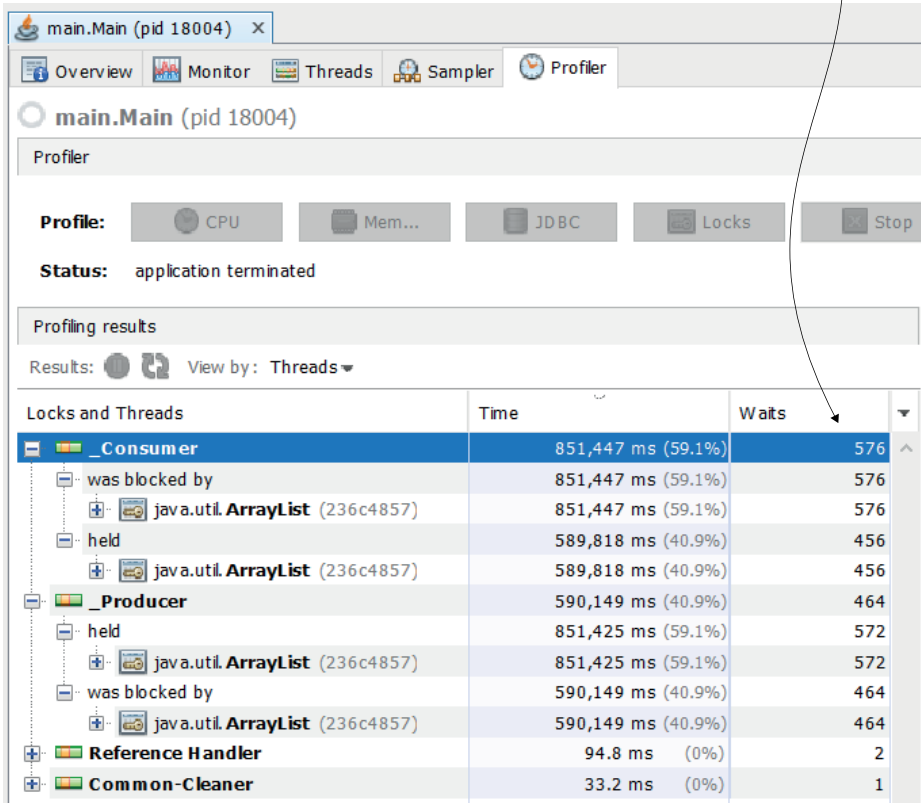


Рис. 9.18. Характеристика блокировок похожа на ранее полученные результаты, но потоки блокируются менее часто

На рис. 9.19 показаны те же подробности анализа, полученные при использовании JProfiler. В JProfiler после группировки событий блокировок по потокам мы получаем информацию о количестве блокировок и о времени ожидания. В предыдущем примере время ожидания было нулевым, но создавалось гораздо больше блокировок. Теперь мы видим меньше блокировок, но увеличилось время выполнения. Это говорит о том, что при использовании методики ожидание/оповещение JVM переключается между потоками медленнее, чем при разрешении монитору синхронизированного блока кода естественным образом блокировать и снимать блокировки потоков.

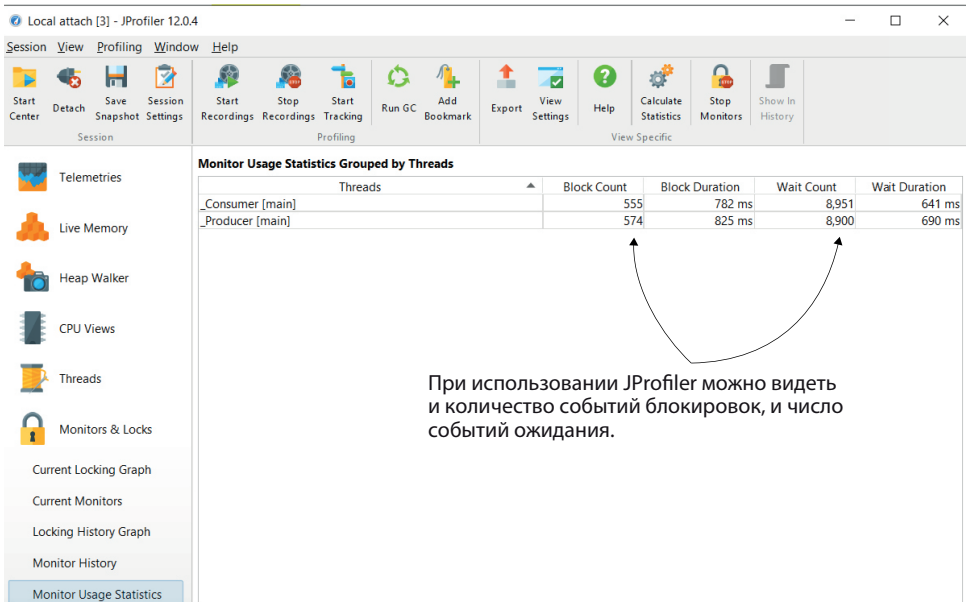


Рис. 9.19. Те же подробности, полученные с использованием JProfiler. Меньше блокировок потоков, но теперь они блокируются на гораздо более длительное время

9.4. Резюме

- Поток может быть заблокирован и приведен в состояние ожидания синхронизированным блоком кода. Блокировки возникают, если потоки синхронизированы для того, чтобы избежать одновременного изменения совместно используемых ресурсов.
- Блокировки необходимы для того, чтобы избежать состояний гонки, но иногда приложения используют неправильные методики синхронизации потоков, которые могут привести к нежелательным результатам, таким как проблемы с производительностью, или даже к полному зависанию приложения (в случае взаимоблокировок).
- Блокировки, созданные синхронизированными блоками кода, замедляют выполнение приложения, потому что заставляют потоки входить в режим ожидания вместо того, чтобы позволить им работать. Блокировки могут потребоваться в особых вариантах реализации, но лучше найти способы минимизации времени блокировок потоков приложения.
- Можно использовать профилировщик, чтобы определить, когда блокировки замедляют приложение, сколько блокировок создается во время выполнения приложения и до какой степени они ухудшают производительность.

- При использовании профилировщика всегда следует выполнять сначала выборку выполнения, чтобы узнать, влияют ли блокировки на выполнение приложения. Обычно блокировки обнаруживаются, если при выборке можно видеть, что метод ожидает сам себя.
- Если при выборке выяснилось, что блокировки, вероятнее всего, влияют на выполнение приложения, то можно продолжить анализ, используя профилирование (инструментовку) блокировок, которое показывает участвующие в блокировках потоки, количество блокировок, управляющие мониторы и отношения между заблокированными потоками и потоками, создающими блокировки. Это подробности помогут определить, является ли оптимальным выполнение приложения или можно найти способы его улучшения.
- Каждое приложение имеет собственное предназначение, поэтому не существует единой универсальной методики для анализа блокировок потоков. В общем случае необходимо минимизировать время блокировок потоков и убедиться в отсутствии несправедливого исключения потоков из выполнения (ситуации «голодания» потоков).

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javalib>

Глава 10

Анализ взаимоблокировок с помощью дампов потоков

Темы:

- получение дампов потоков с использованием профилировщика;
- получение дампов потоков с использованием командной строки;
- чтение дампов потоков для анализа проблем.

В этой главе рассматривается использование дампов потоков для анализа выполнения потока в заданный момент времени. Мы часто используем дампы потоков в ситуациях, когда приложение перестает реагировать, как в случае взаимоблокировки. Взаимоблокировка (deadlock) возникает, когда несколько потоков приостанавливают работу и ждут друг друга, чтобы выполнить заданное условие. Если гипотетический поток А ждет поток В, чтобы что-то сделать, а поток В ждет поток А, то ни один из них не может продолжить выполнение. В этом случае приложение или по крайней мере его часть зависает (freeze). Необходимо знать, как анализировать подобную проблему, чтобы найти главную причину ее возникновения и в итоге устранить ее.

Поскольку взаимоблокировка может привести к полной остановке процесса, обычно невозможно применить выборку или профилирование (инструментовку), как это было сделано в главе 9. Вместо этого можно получить статистические данные обо всех потоках и их состояниях в конкретном JVM-процессе. Такие статистические данные называются дампом потоков (thread dump).

10.1. Получение дампа потоков

В этом разделе мы рассмотрим способы получения дампа потоков. Для этого будет использоваться небольшое приложение, в котором реализована

проблема, преднамеренно создающая взаимоблокировки. Это приложение можно найти в проекте `da-ch10-ex1`. Запустим это приложение и подождем, когда оно зависнет (это должно произойти через несколько секунд), затем рассмотрим способы получения дампов потоков. После этого мы обсудим, как читать дампы потоков (в разделе 10.2).

Рассмотрим подробнее, как реализовано используемое здесь приложение и почему при его выполнении возникают взаимоблокировки. В приложении работают два потока, изменяющие два совместно используемых ресурса (два экземпляра списка). Поток с именем `producer` добавляет значения в первый или второй список во время выполнения. Другой поток с именем `consumer` удаляет значения из этих списков. Если вы изучили главу 9, то, возможно, вспомните, что мы работали с похожим приложением. Но поскольку логика приложения не существенна для исследования этого примера, в листингах она пропущена, сохранена только та часть, которая важна для демонстрации, – синхронизированные блоки.

Пример упрощен, чтобы позволить вам сосредоточиться на обсуждаемых методиках анализа. В реальных производственных приложениях, как правило, все гораздо сложнее. Кроме того, неправильно используемые синхронизированные блоки – это не единственный способ возникновения взаимоблокировок. Некорректное применение блокирующих объектов, таких как семафоры, триггеры с защелкой (D-триггеры) или барьеры, также может стать причиной возникновения подобных проблем. Но действия, которые вы будете изучать для анализа таких проблем, одинаковы.

В листингах 10.1 и 10.2 обратите внимание на то, что оба потока используют вложенные синхронизированные блоки с двумя различным мониторами: `listA` и `listB`. Проблема заключается в том, что один из потоков использует монитор `listA` для внешнего синхронизированного блока, тогда как монитор `listB` применяется для внутреннего. Другой поток использует те же мониторы в обратном порядке. Такое проектирование кода создает возможности для возникновения взаимоблокировок, как наглядно показано на рис. 10.1.

Листинг 10.1. Использование вложенных синхронизированных блоков для потока-потребителя

```
public class Consumer extends Thread {

    // Здесь код не показан.

    @Override
    public void run() {
        while (true) {
            synchronized (Main.listA) {           ❶
                // ...

                synchronized (Main.listB) {       ❷
                    // ...
                }
            }
        }
    }
}
```

```

        work();
    }
}
}

// Здесь код не показан.
}

```

- ❶ Внешний синхронизированный блок использует монитор listA.
- ❷ Внутренний синхронизированный блок использует монитор listB.

В листинге 10.1 поток-потребитель использует listA как монитор для внешнего синхронизированного блока. В листинге 10.2 поток-производитель использует тот же монитор для внутреннего блока, а монитор listB также меняется местами в этих двух потоках.

Листинг 10.2. Использование вложенных синхронизированных блоков для потока-производителя

```

public class Producer extends Thread {

    // Здесь код не показан.

    @Override
    public void run() {
        Random r = new Random();
        while (true) {
            synchronized (Main.listB) {           ❶

                synchronized (Main.listA) {       ❷
                    work(r);
                }
            }
        }
    }

    // Здесь код не показан.
}

```

- ❶ Монитор listB используется внешним синхронизированным блоком.
- ❷ Монитор listA используется внутренним синхронизированным блоком.

На рис. 10.1 показано, как эти два потока могут войти в состояние взаимоблокировки.

1. Предположим, что при выполнении оба потока вошли во внешний синхронизированный блок, но пока еще не вошли во внутренний синхронизированный блок. Стрелка показывает, где находится каждый поток во время выполнения.

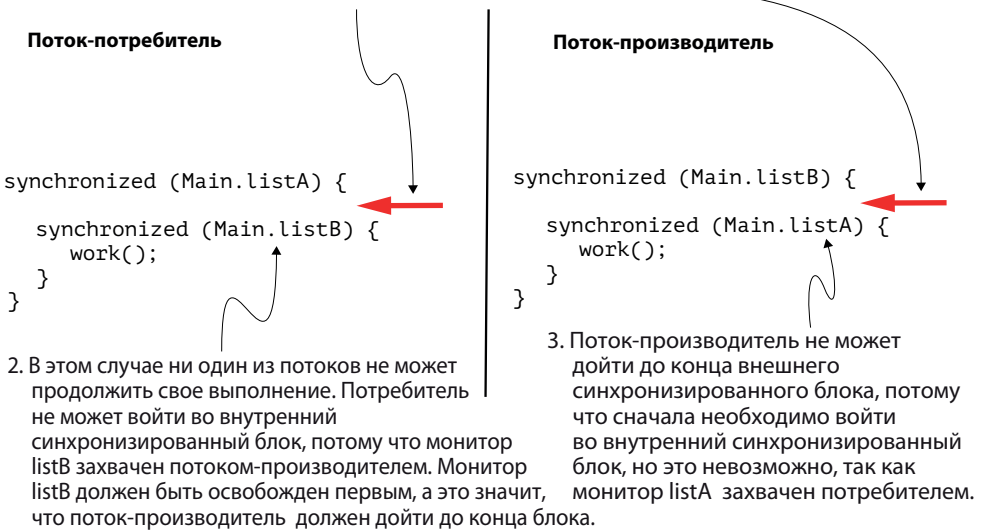


Рис. 10.1. Если оба потока входят во внешний синхронизированный блок, но пока еще не вошли во внутренний, то они остаются приостановленными и ждут друг друга. Мы говорим, что они вошли в состояние взаимоблокировки

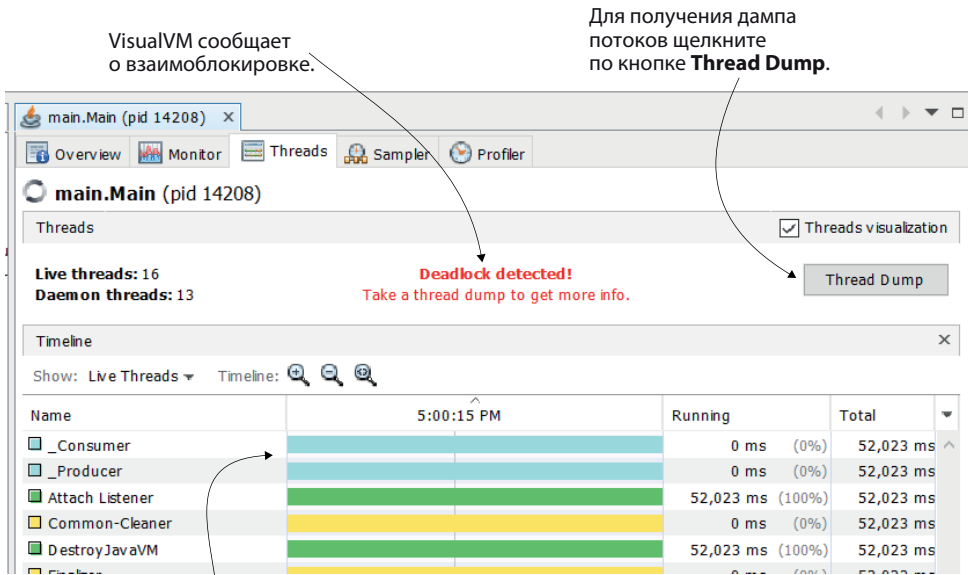
10.1.1. Получение дампа потока с использованием профилировщика

Что мы делаем, когда имеется зависшее приложение и необходимо определить главную причину возникновения проблемы? Использование профилировщика для анализа блокировок, вероятнее всего, не подходит для сценария, в котором приложение или его часть зависает полностью. Вместо анализа блокировок во время выполнения, как было показано в главе 9, мы сделаем моментальный снимок (snapshot) только состояний потоков приложения. Затем прочитаем этот моментальный снимок (т. е. дамп потоков) и узнаем, какие потоки воздействуют друг на друга и приводят к зависанию приложения.

Дамп потоков можно получить с помощью профилировщика (например, VisualVM, JProfiler) или прямым вызовом инструментального средства, предоставляемого JDK, используя командную строку. В этом подразделе мы рассмотрим, как создать дамп потоков с помощью профилировщика, а в подразделе 10.1.2 узнаем, как получить ту же информацию, используя командную строку.

Запустим приложение (проект da-ch10-ex1) и подождем несколько секунд до входа в состояние взаимоблокировки. Понятно, что приложение находится в состоянии взаимоблокировки, когда оно больше не выводит сообщения в консоли (оно зависло).

Получение дампа потоков с использованием профилировщика – это простая методика. Необходимо всего лишь один щелчок по определенной кнопке. Воспользуемся VisualVM для получения дампа потоков. На рис. 10.2 показан интерфейс VisualVM. Здесь можно видеть, что VisualVM обладает достаточным интеллектом и уже определил, что некоторые потоки текущего процесса вошли в состояние взаимоблокировки. Это показано на вкладке **Threads** (Потоки).



В таблице потоков можно видеть, что оба потока – потребитель и производитель – находятся в состоянии ожидания.

Рис. 10.2. Когда некоторые потоки приложения входят в состояние взаимоблокировки, VisualVM оповещает об этой ситуации сообщением на вкладке **Threads**. Обратите внимание: оба потока `_Consumer` и `_Producer` заблокированы на графической шкале времени. Для получения дампа потоков просто щелкните по кнопке **Thread Dump** (Дамп потоков) в правом верхнем углу вкладки

После формирования дампа потоков интерфейс выглядит так, как показано на рис. 10.3. Дамп потоков представлен как обычный текст, описывающий потоки приложения и предоставляющий о них подробную информацию (например, их состояние в жизненном цикле, кто заблокировал их и т. п.).



ПРИМЕЧАНИЕ. На первый взгляд текстовый дамп потоков на рис. 10.3 может показаться непонятным. Далее в этой главе вы научитесь читать его.

Дамп потоков показывает информацию о каждом активном потоке. Мы находим потоки производитель и потребитель в этом сгенерированном дампе потоков.

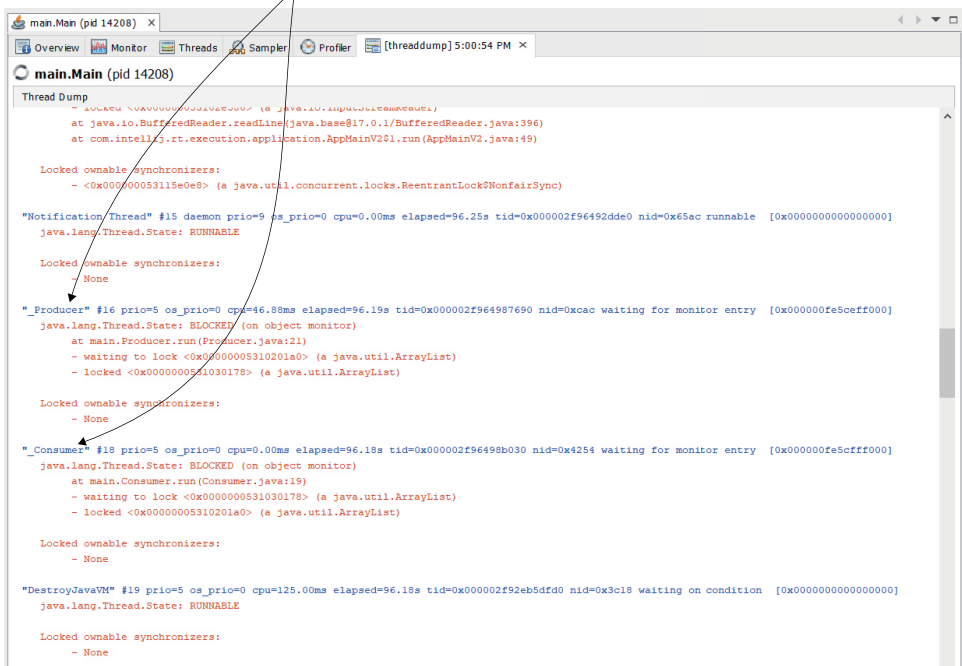


Рис. 10.3. Дамп потоков в виде обычного текста, описывающего потоки приложения. В сформированном здесь дампе потоков можно найти оба потока, находящиеся в состоянии взаимоблокировки, – `_Consumer` и `_Producer`

10.1.2. Генерация дампа потоков из командной строки

Дамп потоков также можно получить, используя командную строку. Этот способ особенно полезен, если необходимо сформировать дамп потоков из удаленной рабочей среды. Большую часть времени у вас не будет возможности удаленного профилирования приложения, установленного во внешней рабочей среде (и следует помнить о том, что удаленное профилирование и отладка не рекомендуются в производственной рабочей среде, – об этом было сказано в главе 4). Поскольку в большинстве случаев получить доступ к удаленной среде можно только с использованием командной строки, необходимо знать в том числе и о таком способе формирования дампа потоков.

К счастью, получить дамп потоков с использованием командной строки достаточно просто (см. рис. 10.4):

- 1) найти идентификатор ID процесса, для которого необходимо сформировать дамп потоков;

- 2) получить дамп потоков в виде (простых, необработанных) текстовых данных и сохранить их в файле;
- 3) загрузить сохраненный дамп потоков в профилировщик, чтобы проще было его читать.

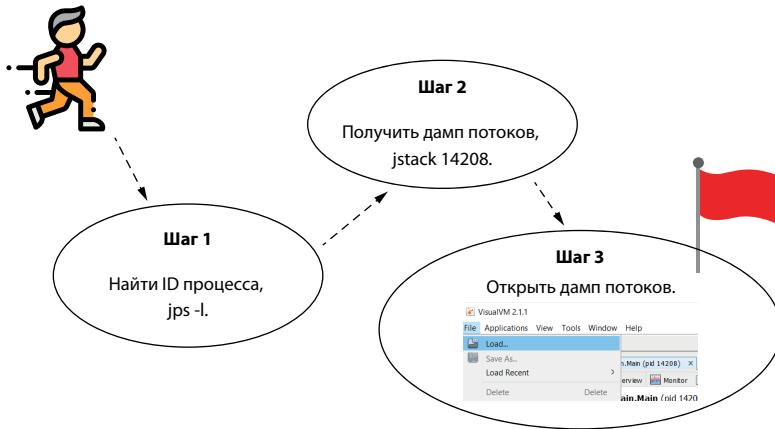


Рис. 10.4. Для получения дампа потоков с помощью командной строки необходимо выполнить три простых шага. Во-первых, найти идентификатор процесса, для которого требуется дамп потоков. Во-вторых, использовать инструментальное средство JDK, чтобы получить дамп потоков. Наконец, открыть этот дамп потоков в профилировщике, чтобы прочитать его

Шаг 1: найти идентификатор ID процесса, который необходимо проанализировать

До сих пор мы идентифицировали профилируемые процессы по имени (представленному как имя основного (main) класса). Но при получении дампа потоков с использованием командной строки необходимо идентифицировать процесс по его ID. Как узнать идентификатор процесса (PID) для выполняющегося Java-приложения? Простейший способ – использование инструмента `jps`, предоставляемого в комплекте JDK. В приведенном ниже фрагменте показана команда, которую необходимо выполнить. Ключ `-l` (буква `L` в нижнем регистре) используется для получения имен основных классов, связанных с идентификаторами процессов PID. Таким образом, можно идентифицировать процессы точно так же, как мы делали это в главах 6–9, где изучали профилирование выполнения приложения:

```
jpl -l
```

На рис. 10.5 показан результат выполнения этой команды. Числовые значения в первом столбце выходных данных – это идентификаторы процессов PID. Второй столбец связывает имя основного класса с каждым PID. Таким образом, мы узнаём PID, который будем использовать на шаге 2 для получения дампа потоков.

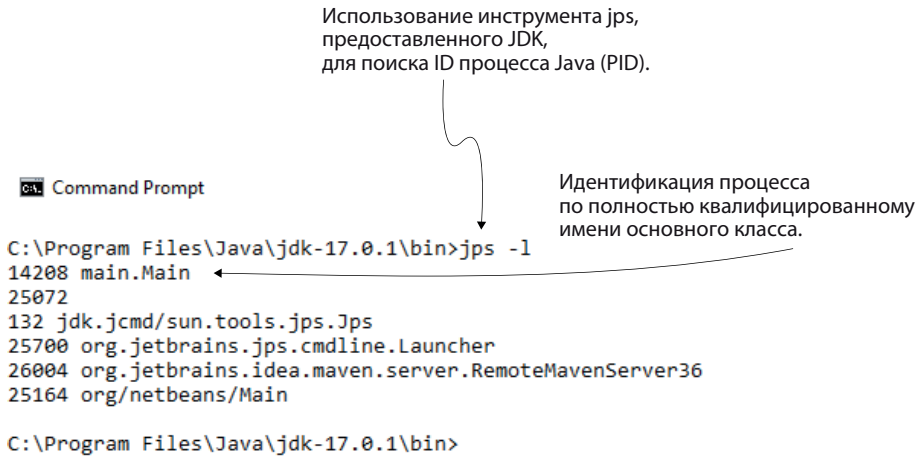


Рис. 10.5. Применяя инструмент jps, предоставленный JDK, мы получаем идентификаторы PID работающих процессов Java. Эти PID необходимы для формирования дампов потоков для конкретного процесса

Шаг 2: формирование дампа потоков

После получения возможности идентификации процесса (по его PID), для которого необходимо сформировать дамп потоков, можно воспользоваться другим инструментом, предоставляемым JDK – jstack, – для генерации дампа потоков. При использовании jstack необходимо передать ему только идентификатор процесса как параметр (вместо шаблона <<PID>> нужно подставить числовое значение PID, полученное на шаге 1):

```
jstack <<PID>>
```

Пример выполнения такой команды:

```
jstack 14208
```

На рис. 10.6 показан результат выполнения команды jstack с параметром PID, равным 14208. Дамп потоков выводится как простой текст, который можно сохранить в файле для передачи или для загрузки в какое-либо инструментальное средство для анализа.

Шаг 3: импортирование сгенерированного дампа потоков в профилировщик для упрощения чтения

Обычно вывод команды jstack, т. е. дамп потоков, сохраняют в файле. Сохранение дампа потоков в файле позволяет перемещать, хранить или импортировать его в инструментальные средства, которые помогают выполнить анализ подробностей дампа.

На рис. 10.7 показано, как можно поместить вывод команды jstack в файл в командной строке. После создания такого файла его можно загрузить в VisualVM, используя пункт меню **File** (Файл) > **Load** (Загрузить).

Использование инструмента jstack, предоставленного JDK, для получения дампа потоков. Единственный обязательный параметр – идентификатор (PID) процесса, для которого необходимо сгенерировать дамп потоков.



```

Command Prompt

C:\Program Files\Java\jdk-17.0.1\bin>jstack 14208
2021-12-01 17:10:51
Full thread dump OpenJDK 64-Bit Server VM (17.0.1+12-39 mixed mode, sharing):

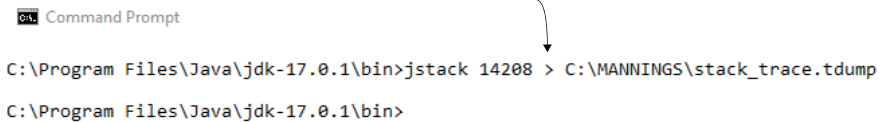
Threads class SMR info:
 java.thread_list=0x000002f96843b110, length=25, elements={
 0x000002f9646faaf0, 0x000002f9646fc110, 0x000002f96470ec20, 0x000002f9647107e0,
 0x000002f964713a40, 0x000002f964718090, 0x000002f964719cb0, 0x000002f96471a750,
 0x000002f96472e0b0, 0x000002f9647c08b0, 0x000002f964921580, 0x000002f96492dde0,
 0x000002f964987690, 0x000002f96498b030, 0x000002f92eb5dfd0, 0x000002f965cb8ce0,
 0x000002f966f07490, 0x000002f967087840, 0x000002f9666ba6a0, 0x000002f96747e060,
 0x000002f96747f870, 0x000002f96747fd40, 0x000002f964a76010, 0x000002f964a746c0,
 0x000002f967480210
 }

"Reference Handler" #2 daemon prio=10 os_prio=2 cpu=0.00ms elapsed=692.87s tid=0x000002f9646faaf0 nid=0x6888 waiting on
condition [0x000002f9646fc110]
 java.lang.Thread.State: RUNNABLE
   at java.lang.ref.Reference.waitForReferencePendingList(java.base@17.0.1/Native Method)
   at java.lang.ref.Reference.processPendingReferences(java.base@17.0.1/Reference.java:253)
   at java.lang.ref.Reference$ReferenceHandler.run(java.base@17.0.1/Reference.java:215)

"Finalizer" #3 daemon prio=8 os_prio=1 cpu=0.00ms elapsed=692.87s tid=0x000002f9646fc110 nid=0x5fa8 in Object.wait() [0
x000002f9646fc110]
 java.lang.Thread.State: WAITING (on object monitor)
   at java.lang.Object.wait(java.base@17.0.1/Native Method)
   - waiting on <0x0000000531818640> (a java.lang.ref.ReferenceQueue$Lock)
   at java.lang.ref.ReferenceQueue.remove(java.base@17.0.1/ReferenceQueue.java:155)
   - locked <0x0000000531818640> (a java.lang.ref.ReferenceQueue$Lock)
  
```

Рис. 10.6. Команда jstack со следующим за ней параметром PID сгенерирует дамп потоков для заданного процесса. Дамп потоков показан как простой текст (также называемый сырым (необработанным) дампом потоков). Можно сохранить этот текст в файле для импортирования и дальнейшего анализа

Правильный подход: размещение содержимого вывода jstack в файле, чтобы можно было хранить его, передавать и анализировать.



```

Command Prompt

C:\Program Files\Java\jdk-17.0.1\bin>jstack 14208 > C:\MANNINGS\stack_trace.tdump

C:\Program Files\Java\jdk-17.0.1\bin>
  
```

Можно открыть сохраненный дамп потоков в любом профилировщике, чтобы облегчить чтение. Например, в VisualVM можно открыть файл дампа, используя меню **File > Load**.

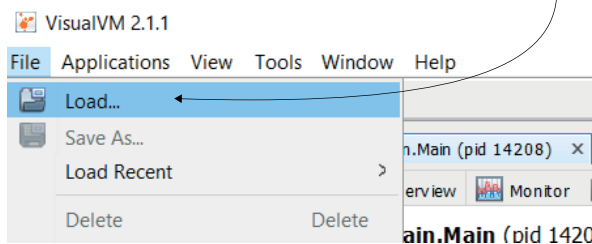


Рис. 10.7. После сохранения дампа потоков в файле его можно открыть в разнообразных инструментальных средствах для анализа. Например, чтобы открыть дамп в VisualVM, в меню выберите пункт **File > Load**

10.2. Чтение дампов потоков

В этом разделе рассматривается чтение дампов потоков. После генерации дампа потоков необходимо узнать, как его читать и как эффективно использовать для обнаружения проблем. Начнем с того, как читать дампы потоков в виде простого текста в подразделе 10.2.1, – это значит, что вы научитесь читать необработанные (сырые) данные, предоставленные jstack (см. подраздел 10.1.2). Затем в подразделе 10.2.2 мы воспользуемся инструментальным средством под названием fastThread (<https://fastthread.io/>), предоставляющим более простой способ визуализации данных в дампе потоков.

Обе методики (чтение текстовых дампов потоков и использование более продвинутой визуализации) полезны. Разумеется, мы всегда предпочитаем продвинутую визуализацию, но если ее невозможно применить, то вы должны знать, как правильно пользоваться сырыми данными.

10.2.1. Чтение дампов потоков в виде простого текста

После генерации дампа потоков вы получаете описание потоков в простом текстовом формате (т. е. необработанные (сырые) данные). Хотя существуют инструменты, которые можно использовать для быстрой визуализации этих данных (это тема подраздела 10.2.2), я всегда считал важным для разработчика понимание и представления в необработанном виде. Вы можете попасть в ситуацию, когда невозможно переместить сырой дамп потоков из рабочей среды, в которой он был сгенерирован. Например, вы установили соединение с удаленным контейнером и можете использовать только командную строку для исследования журналов и анализа того, что происходит с работающим приложением. Вы предполагаете, что проблема связана с потоками, поэтому необходимо сгенерировать дамп потоков. Если вы можете читать дамп потоков как текст, то, кроме консоли, вам ничего не нужно.

Рассмотрим листинг 10.3, в котором показан один из потоков в дампе. Здесь нет ничего лишнего, кроме аналогичных подробностей, выводимых для каждого потока в приложении, активного в момент генерации дампа. Ниже перечислены подробности, которые можно узнать о каждом потоке:

- имя потока;
- идентификатор ID потока;
- идентификатор ID внутреннего системного потока;
- приоритет потока на уровне операционной системы;
- общее время и время использования ЦП, потребленное потоком;
- описание состояния;
- имя состояния;
- трассировка стека;
- кто заблокировал этот поток;
- какими блокировками владеет этот поток.

Листинг 10.3. Внутренние подробности об одном из потоков, показанные в дампе потоков

```

"_Producer" #16 prio=5 os_prio=0 cpu=46.88ms elapsed=763.96s      ❶
↳ tid=0x000002f964987690 nid=0xcac waiting for monitor entry    ❷
↳ [0x000000fe5ceff000]
java.lang.Thread.State: BLOCKED (on object monitor)             ❸
at main.Producer.run(Unknown Source)                             ❹
- waiting to lock <0x000000052e0313f8> (a java.util.ArrayList)    ❺
- locked <0x000000052e049d38> (a java.util.ArrayList)            ❻

```

- ❶ Имя потока и подробности о потреблении ресурсов и времени выполнения.
- ❷ Идентификатор потока и описание состояния.
- ❸ Состояние потока.
- ❹ Трассировка стека потока.
- ❺ Идентификатор блокировки, остановившей текущий поток, и тип объекта монитора.
- ❻ Идентификатор блокировки, созданной текущим потоком.

Данные, выводимые в первую очередь, – имя потока, в нашем случае "_Producer". Имя потока весьма важно, так как это один из способов его идентификации в дампе в дальнейшем, если потребуется. JVM также связывает поток с его идентификатором (thread ID) (в листинге 10.3 tid=0x000002f964987690). Поскольку имя присваивает разработчик, существует небольшая вероятность того, что некоторые потоки получают одинаковое имя. Если возникает такая нежелательная ситуация, то у вас остается возможность идентификации потока в дампе по его ID (который никогда не повторяется).

В JVM-приложении поток – это обертка для системного потока, а это означает, что всегда можно идентифицировать поток операционной системы (ОС), выполняющийся незаметно для пользователя. И если вдруг вам потребуется сделать это, то обратите внимание на идентификатор системного потока native thread ID (в листинге 10.3 nid=0xcac).

После идентификации потока вы определяете интересующие вас подробности. Первые три фрагмента информации в дампе потоков – это приоритет потока, время использования ЦП и общее время выполнения. Любая ОС присваивает приоритет каждому выполняющемуся потоку. Я редко использую это значение в дампе потоков. Но если вы видите, что поток не так активен, как следовало бы по вашему мнению, и ОС присвоила ему низкий приоритет, то именно это может оказаться причиной возникновения проблемы. В такой ситуации общее время выполнения также должно намного превышать время использования ЦП. Вспомним из главы 7, что общее время выполнения соответствует времени жизни потока, а время использования ЦП показывает, насколько активно он работал.

Описание состояния является весьма полезной деталью. Оно сообщает вам на простом английском языке, что происходит с потоком. В данном

случае поток «waiting for monitor entry» (ожидает входа в монитор), т. е. блокирован на входе в синхронизированный блок. Поток может находиться в состоянии «timed waiting on a monitor» (ожидание монитора с контролем времени), а это означает, что поток «спит» (sleeping) в течение определенного интервала времени или выполняется. Имя состояния (state name) – Running, Waiting, Blocked и т. п. – связано с описанием состояния. В приложении D вы найдете подробную справочную информацию по жизненному циклу потока и его состояниям, если потребуется.

Дамп потоков предоставляет трассировку стека (stack trace) для каждого потока, где точно показано, какая часть кода потока выполнялась, когда формировался дамп. Трассировка стека полезна, поскольку она точно показывает, какой именно поток активно работал. Трассировку стека можно использовать для поиска конкретной части кода, которую необходимо отлаживать в дальнейшем, или, если поток работает слишком медленно, для точного определения того, что задерживает или блокирует этот поток.

Наконец, для потоков, которые создают блокировки или сами заблокированы, можно узнать, какими блокировками они владеют и из-за каких блокировок они находятся в состоянии ожидания. Именно эти подробности вы будете анализировать каждый раз при анализе взаимоблокировок. Кроме того, они дают подсказки по оптимизации. Например, если вы видите, что поток владеет многими блокировками, то пора задуматься на том, зачем и как можно изменить это поведение, чтобы поток не блокировал так много других выполнений.

Важно помнить об одном свойстве дампов потоков – они предоставляют почти столько же подробностей, что и обычное профилирование блокировки (описанное в главе 9). Профилирование блокировки обладает преимуществом перед дампом потоков: оно показывает выполнение в динамике. Подобно различию между картиной (или фотографией) и кинофильмом, дамп потоков – это всего лишь моментальный снимок в определенный момент времени (здесь: во время выполнения), тогда как профилирование показывает, как изменяются параметры во время выполнения. Но во многих ситуациях картины (фотографии) вполне достаточно, и ее намного проще получить.



ПРИМЕЧАНИЕ. Иногда достаточно использовать дамп потоков вместо профилировщика.

Если необходимо всего лишь узнать, какой код выполняется в определенный момент времени, вполне достаточно сгенерировать дамп потоков. Вы научились использовать выборку для этой цели, но полезно знать, что дамп потоков может сделать то же самое. Например, у вас нет доступа для выполнения удаленного профилирования приложения, но необходимо уз-

нать, какой код выполняется «за кулисами». Для этого можно сгенерировать дамп потоков.

Теперь сосредоточимся на том, как найти связь между потоками в дампе. Как можно проанализировать способ, которым потоки взаимодействуют между собой? Нас особенно интересуют потоки, блокирующие друг друга. В листинге 10.4 я добавил подробности из дампа потоков для двух потоков, о которых известно, что они находятся в состоянии взаимоблокировки. Но возникает вопрос: «Как мы узнаем, что они находятся в состоянии взаимоблокировки, если эта подробность заранее неизвестна?»

Листинг 10.4. Поиск потоков, блокирующих друг друга

```
"_Producer" #16 prio=5 os_prio=0 cpu=46.88ms
➔ elapsed=763.96s tid=0x000002f964987690
➔ nid=0xcac waiting for monitor entry [0x000000fe5ceff000]
java.lang.Thread.State: BLOCKED (on object monitor)
at main.Producer.run(Unknown Source)
- waiting to lock <0x000000052e0313f8>
➔ (a java.util.ArrayList)
- locked <0x000000052e049d38>
➔ (a java.util.ArrayList)
"_Consumer" #18 prio=5 os_prio=0 cpu=0.00ms
➔ elapsed=763.96s tid=0x000002f96498b030
➔ nid=0x4254 waiting for monitor entry [0x000000fe5cfff000]
java.lang.Thread.State: BLOCKED (on object monitor)
at main.Consumer.run(Unknown Source)
- waiting to lock <0x000000052e049d38> (a java.util.ArrayList) ❶
- locked <0x000000052e0313f8> (a java.util.ArrayList) ❷
```

❶ Поток `_Consumer` находится в состоянии ожидания из-за блокировки, созданной потоком `_Producer`.

❷ Поток `_Producer` находится в состоянии ожидания из-за блокировки, созданной потоком `_Consumer`.

Если предполагается возникновение взаимоблокировки, то при анализе вы должны сосредоточиться на причинах блокировок потоков (см. рис. 10.8):

- 1) исключить из анализа все незаблокированные потоки, чтобы сосредоточиться на потоках, которые могли привести к взаимоблокировке;
- 2) начать с первого потока-кандидата (с потока, который не был исключен на шаге 1) и искать идентификатор блокировки этого потока;
- 3) найти поток, создавший эту блокировку, и проверить, что именно блокирует найденный поток. Если в этом месте вы возвращаетесь к потоку, с которого начали поиск, то все обнаруженные (проанализированные) потоки находятся в состоянии взаимоблокировки.

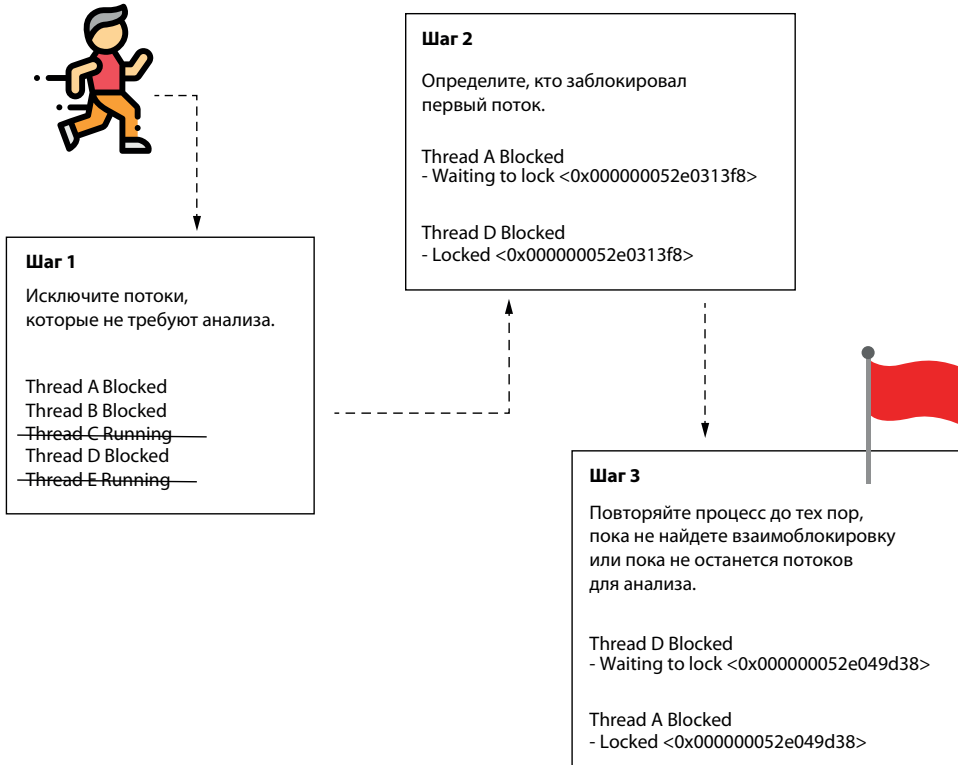


Рис. 10.8. Чтобы найти взаимоблокировку в дампе потоков, выполните три простых шага. Во-первых, исключите все незаблокированные потоки. Затем начните с одного из заблокированных потоков и определите, что стало причиной его блокировки, используя идентификатор блокировки. Продолжайте этот процесс для каждого потока. Если вы вернулись к потоку, который уже был проанализирован, это означает, что взаимоблокировка найдена

Шаг 1: исключение незаблокированных потоков

В первую очередь необходимо исключить все незаблокированные потоки, чтобы сосредоточиться только на потоках, являющихся потенциальными кандидатами на участие в анализируемой ситуации, т. е. во взаимоблокировке. В дампе потоков могут быть описаны десятки потоков. Поэтому нужно устранить шум (помехи) и сосредоточиться только на заблокированных потоках.

Шаг 2: анализ первого потока-кандидата и поиск источника его блокировки

После исключения ненужных подробностей начните с первого потока-кандидата и по идентификатору блокировки найдите причину, заставляющую ждать этот поток. Идентификатор блокировки – это значение в

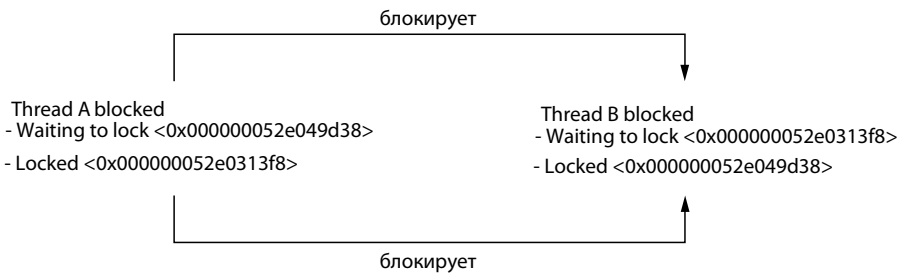
угловых скобках (в листинге 10.4 поток "_Producer" находится в состоянии ожидания из-за блокировки с идентификатором 0x000000052e0313f8).

Шаг 3: поиск источника блокировки следующего потока

Повторите процесс. Если в некоторый момент вы вернетесь к потоку, который уже был проанализирован, то вы нашли взаимоблокировку – см. листинг 10.4.

В рассматриваемом здесь примере показана простая взаимоблокировка, в которой два потока блокируют друг друга. Выполняя описанный выше трехшаговый процесс, вы обнаружите, что поток "_Producer" блокирует поток "_Consumer", и наоборот. Более сложные взаимоблокировки возникают, когда в них участвует более двух потоков. Например, поток А блокирует поток В, поток В блокирует поток С, а поток С блокирует поток А. Можно обнаружить длинную цепочку потоков, блокирующих друг друга. Чем длиннее цепочка потоков во взаимоблокировке, тем сложнее ее найти, понять и устранить. На рис. 10.9 показано различие между простой и сложной взаимоблокировкой.

Простая взаимоблокировка



Сложная взаимоблокировка (более двух потоков)

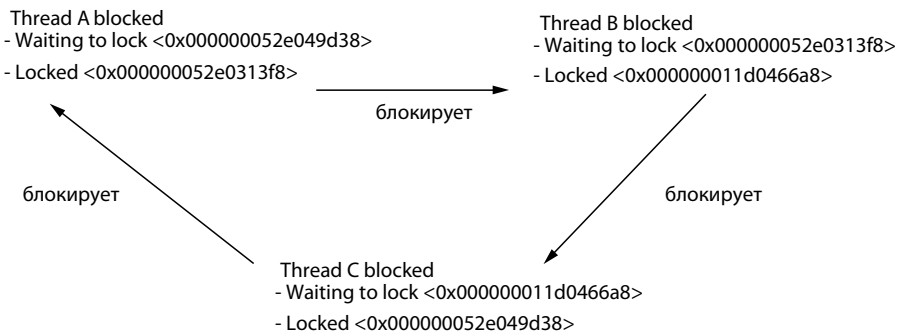


Рис. 10.9. Если только два потока блокируют друг друга, это называется простой взаимоблокировкой, но взаимоблокировка может быть создана несколькими потоками, блокирующими друг друга. Большее количество потоков означает бóльшую сложность. Следовательно, при участии более двух потоков взаимоблокировка называется сложной

Иногда сложную взаимоблокировку можно перепутать с каскадно заблокированными потоками (см. рис. 10.10). Каскадно заблокированные потоки (также называемые каскадными блокировками (cascading locks)) – это другая проблема, которую можно обнаружить с помощью дампа потоков. Для обнаружения каскадных блокировок выполняйте те же шаги, что и при анализе взаимоблокировки. Но вместо обнаружения того, что один из потоков блокируется другим в цепочке (взаимоблокировка), в каскаде блокировок вы увидите, что один из потоков ожидает некоторого внешнего события, заставляя ждать и все другие потоки.

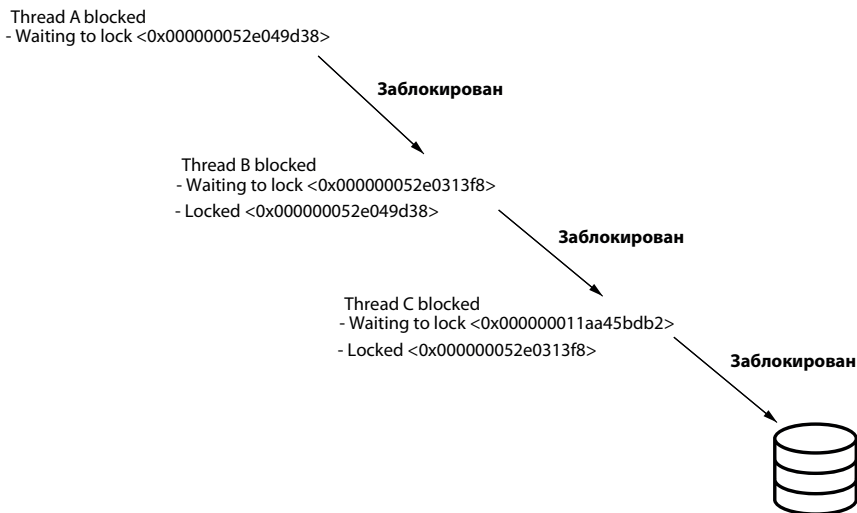


Рис. 10.10. Каскадные блокировки возникают, когда несколько потоков образуют цепочку, в которой они ждут некоторый другой поток. Самый последний поток в такой цепочке блокируется внешним событием, например процедурой чтения из источника данных или вызовом конечной точки

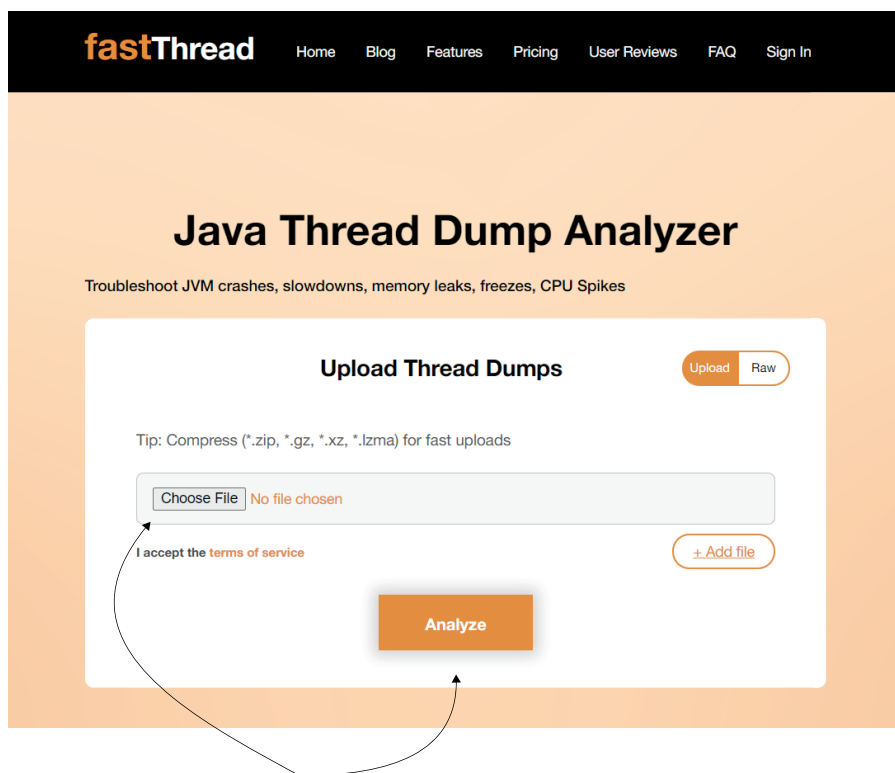
Каскадно заблокированные потоки обычно являются признаком неудачного проектного решения в многопоточной архитектуре. При проектировании приложения с несколькими потоками мы реализуем многопоточность так, чтобы позволить приложению работать с данными в параллельном режиме. Наличие потоков, ожидающих другие потоки, лишает смысла предназначение многопоточной архитектуры. Хотя иногда возникает необходимость в том, чтобы поток(и) ждал(и) некоторый другой поток, вы не должны планировать использование длинных цепочек потоков с каскадными блокировками.

10.2.2. Использование инструментальных средств для лучшего понимания дампов потоков

Чтение дампа потоков в виде необработанного простого текста полезно, но иногда может оказаться весьма затруднительным. Большинство разра-

ботчиков предпочитает более простой способ визуализации данных в дампе потоков, если это возможно. В наше время можно использовать инструментальные средства, помогающие проще и быстрее понять дамп потоков. По возможности я перемещаю дамп потоков из рабочей среды, в которой он был создан. Обычно я предпочитаю использовать fastThread (<https://fastthread.io/>) для анализа дампа, а не иметь дело с необработанными данными.

fastThread – это веб-инструмент, специально предназначенный для того, чтобы помочь при чтении дампов потоков. Предлагается бесплатная и лицензионная (оплачиваемая) версия, но для моих потребностей всегда было вполне достаточно бесплатной версии. Нужно просто загрузить файл, содержащий необработанные данные дампа потоков, и немного подождать, пока fastThread извлекает необходимые подробности, чтобы представить их в легко читаемой форме. На рис. 10.11 показана начальная страница, где вы выбираете файл, содержащий необработанные данные дампа потоков, из своей системы и выгружаете его для анализа.



1. Для анализа дампа потоков выгрузите файл с необработанными данными, затем щелкните по кнопке **Analyze**.

Рис. 10.11. Для анализа дампа потоков выгрузите файл, содержащий необработанные данные дампа, в fastThread и подождите, когда подробности будут представлены в удобной для чтения и понимания форме

Анализ fastThread показывает разнообразные подробности из дампа потоков, в том числе определение взаимоблокировки, графы зависимостей, трассировки стека, потребление ресурсов и даже flame-график (см. рис. 10.12).

После анализа дампа потоков инструмент предоставляет несколько виджетов визуализации: определение взаимоблокировок, потребление ЦП для каждого потока и даже представление процесса в виде flame-графика.

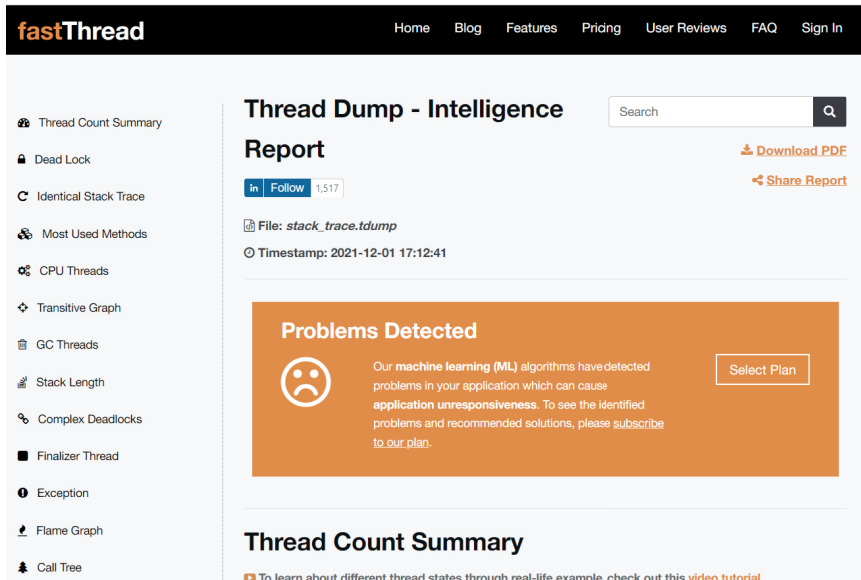


Рис. 10.12. fastThread предоставляет разнообразную подробную информацию в удобном для чтения формате. Эти подробности включают определение взаимоблокировки, графы зависимостей, потребление ресурсов и flame-график

На рис. 10.13 показано, как fastThread определил взаимоблокировку в нашем дампе потоков.

10.3. Резюме

- Если два и более потоков заблокированы при ожидании друг друга, то они находятся в состоянии взаимоблокировки. Когда приложение входит в состояние взаимоблокировки, оно обычно зависает и не может продолжить выполнение.
- Определить главную причину взаимоблокировки можно, используя дампы потоков, которые показывают состояние всех потоков приложения в момент генерации дампа потоков. Эта информация позволяет найти потоки, ожидающие друг друга.

Инструмент идентифицировал взаимоблокировку
и потоки, создавшие ее.

Dead Lock

Learn more about [Deadlock](#)

Thread **_Producer** is in deadlock with thread **_Consumer**

_Producer

PRIORITY : 5
 THREAD ID : 0X000002F964987690
 NATIVE ID : 0XCAC
 NATIVE ID (DECIMAL) : 3244
 STATE : BLOCKED

stackTrace:
 java.lang.Thread.State: BLOCKED (on object monitor)
 at main.Producer.run(Unknown Source)
 - waiting to lock <0x000000052e0313f8> (a java.util.ArrayList)
 - locked <0x000000052e049d38> (a java.util.ArrayList)

_Consumer

PRIORITY : 5
 THREAD ID : 0X000002F96498B030
 NATIVE ID : 0X4254
 NATIVE ID (DECIMAL) : 16980
 STATE : BLOCKED

stackTrace:
 java.lang.Thread.State: RI OCKFD (on object monitor)

Рис. 10.13. После анализа необработанных данных дампа потоков `fastThread` обнаружил взаимоблокировку, созданную потоками `_Consumer` и `_Producer`, и предоставил все подробности о ней

- Дамп потоков также показывает такие подробности, как потребление ресурсов и трассировки стека для каждого потока. Если эта подробная информация достаточна, то для анализа можно использовать дампы потоков вместо инструментальной профилировки. Различие между дампом потоков и профилированием можно мысленно представить как различие между картиной (фотографией) и кинофильмом. При использовании дампа потоков вы получаете только статичное изображение, поэтому отсутствует динамика выполнения, но в вашем распоряжении остается огромное количество важных и полезных подробностей.
- Дамп потоков предоставляет информацию о потоках, которые выполнялись в приложении, когда был сгенерирован дамп. Дамп потоков показывает важнейшие подробности о потоках в простом текстовом формате, включая потребление ресурсов, состояние потока в его жизненном цикле, ожидает ли поток чего-либо и какие блокировки он создал, или какие блокировки воздействуют на него.
- Дамп потоков можно сгенерировать с помощью профилировщика или воспользоваться командной строкой. Использование профили-

ровщика для получения дампа потоков – это самый простой способ, но если невозможно установить соединение профилировщика с работающим процессом (например, из-за ограничений сетевой среды), то для создания дампа можно использовать командную строку. Дамп потоков позволит проанализировать активные потоки и отношения между ними.

- Дамп потоков в простом текстовом формате (также называемый необработанным (сырым) дампом потоков) может оказаться весьма трудным для чтения. Инструментальные средства, такие как `fastThread`, помогают получить визуальное представление подробностей.

Глава 11

.....

Обнаружение проблем, связанных с использованием памяти, при выполнении приложения

Темы:

- выборка выполнения для обнаружения проблем, связанных с распределением памяти;
- профилирование части кода для идентификации главных причин возникновения проблем, связанных с распределением памяти;
- получение и чтение дампов кучи (heap).

Каждое приложение обрабатывает данные, и для этого ему необходимо где-то хранить эти данные при работе с ними. Приложение выделяет часть системной памяти для работы с данными, но память не является бесконечным ресурсом. Все приложения, работающие в системе, совместно используют конечный объем пространства памяти, предоставленный системой. Если приложение неразумно управляет выделенной ему памятью, то память может быть исчерпана, и продолжение работы приложения станет невозможным. Даже если приложение не израсходовало всю память, использование слишком большого ее объема может замедлить работу приложения, поэтому некорректное распределение памяти может стать причиной возникновения проблем с производительностью.

Приложение может работать медленнее, если оно не оптимизировало размещение данных в памяти. Если приложение требует больше памяти, чем предоставляет система, то его работа останавливается и выводится сообщение об ошибке. Таким образом, побочными эффектами плохого управления памятью является замедление выполнения и даже аварийное завершение всего приложения в целом. Весьма важно оснащать приложе-

ние такими функциональными возможностями, которые обеспечивают наилучшее использование выделенной ему памяти.



ПРИМЕЧАНИЕ. Если приложение не размещает обрабатываемые данные оптимальным образом, это может стать причиной более частых запусков сборщика мусора (GC), и потребление ЦП в таком приложении станет более интенсивным.

В управлении своими ресурсами приложение должно быть настолько эффективным, насколько это возможно. При обсуждении ресурсов приложения мы главным образом подразумеваем ЦП (средство обработки) и память. В главах 7–10 мы узнали, как анализировать проблемы, связанные с потреблением ЦП. В этой главе мы сосредоточимся на идентификации проблем, связанных с тем, как приложение размещает данные в памяти.

Глава начинается с рассмотрения выборки выполнения и профилирования для получения статистических данных об использовании памяти в разделе 11.1. Вы узнаете, как определить, существуют ли в приложении проблемы с использованием памяти, и как найти ту часть приложения, которая является источником проблем.

Затем в разделе 11.2 вы узнаете, как получить полный дамп распределенной памяти (т. е. дамп кучи (heap dump)) для анализа ее содержимого. В особых случаях, когда приложение аварийно завершается из-за некорректного управления памятью, невозможно профилировать выполнение. Но получение и анализ содержимого памяти, выделенной приложению, в момент возникновения проблемы может помочь в обнаружении ее главной причины.

Прежде чем продолжить чтение этой главы, следует запомнить несколько основных принципов, по которым Java-приложение распределяет и использует память. Если требуется напоминание об этих принципах, то приложение `E` предоставляет всю информацию, необходимую для полного понимания материала текущей главы.

11.1. Выборка и профилирование для выявления проблем с памятью

В этом разделе используется небольшое приложение, имитирующее некорректно реализованную функциональную возможность, использующую слишком большой объем выделенной памяти. Мы воспользуемся этим приложением для изучения методик анализа, применимых для идентификации проблем при распределении памяти или мест в коде, которые можно оптимизировать для более эффективного использования системной памяти.

Предположим, что имеется реальное приложение, и вы заметили, что некоторая его функция работает слишком медленно. Вы используете методи-

ки, описанные в главе 6, для анализа потребления ресурсов и обнаруживаете, что, хотя приложение «действительно работает» не слишком часто (т. е. потребляет ресурсы ЦП), тем не менее использует большой объем памяти. Если приложение использует слишком много памяти, то JVM может активизировать сборщик мусора (GC), который также будет потреблять ресурсы ЦП. Напомню, что GC – это механизм, который автоматически удаляет ненужные данные из памяти (более подробно об этом см. приложение E).

Посмотрите на рис. 11.1. При изучении методик анализа потребления ресурсов в главе 6 мы пользовались вкладкой **Monitor** в профилировщике VisualVM для наблюдения за ресурсами, потребляемыми приложением. Можно воспользоваться виджетом **Memory** (Память) на этой вкладке, чтобы увидеть, когда приложение использует чрезмерный объем памяти.

На вкладке **Monitor** вы найдете виджет, позволяющий выполнять мониторинг использования памяти приложением.

Обратите внимание на значительное увеличение объема используемой памяти при вызове конечной точки приложения. Кроме того, JVM изменяет максимальный размер кучи, как результат увеличения потребления памяти.

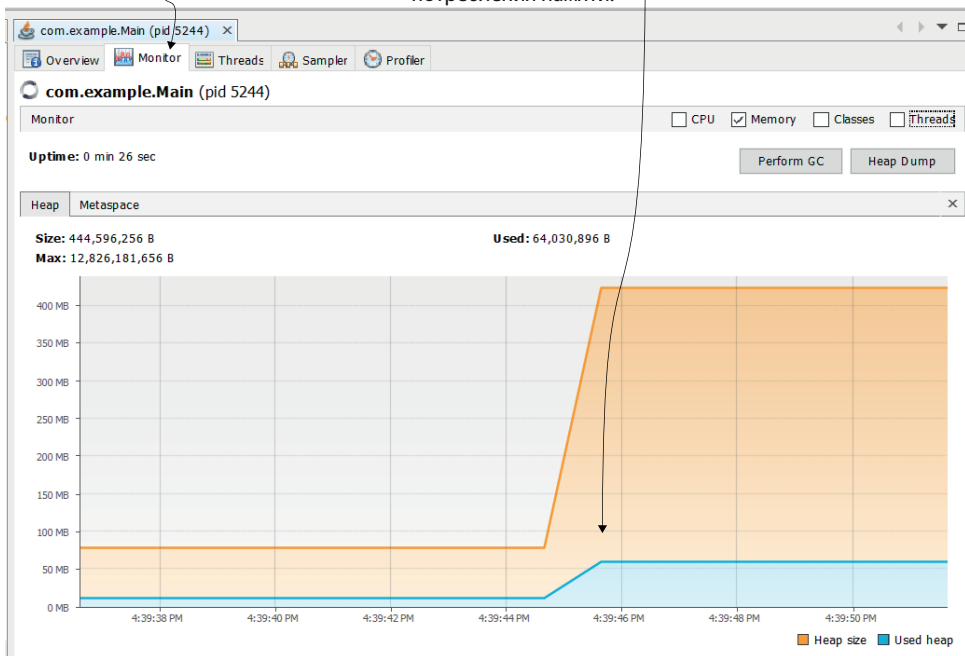


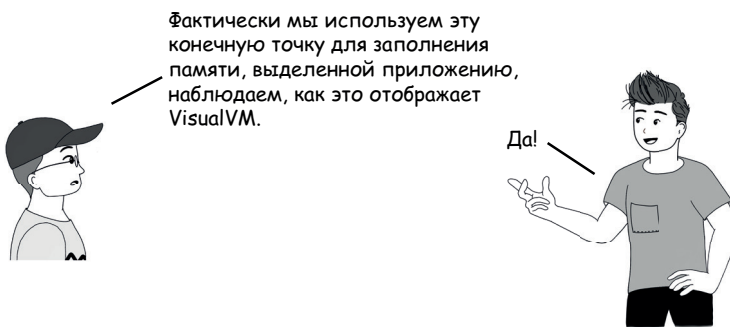
Рис. 11.1. Виджет **Memory** на вкладке **Monitor** в профилировщике VisualVM помогает определить необычное увеличение объема памяти, используемой приложением, в любой момент времени. Часто виджеты на вкладке **Monitor**, такие как **CPU** и **Memory**, подсказывают, как нужно продолжить анализ. Если мы видим, что приложение потребляет аномальный объем памяти, то, вероятнее всего, примем решение о продолжении профилирования памяти при выполнении

Приложение, используемое в этой главе, размещено в проекте da-ch11-ex1. Это небольшое веб-приложение предъявляет конечную точку. При ее

вызове мы задаем некоторое числовое значение, и конечная точка создает соответствующее количество экземпляров объекта. Изначально мы выполняем запрос на создание миллиона объектов (это достаточно большое количество для нашего эксперимента), а затем смотрим, что сообщает профилировщик о выполнении такого запроса. Выполнение конечной точки имитирует происходящее в реальной ситуации, когда конкретная функция приложения расходует огромный объем ресурсов памяти приложения (см. рис. 11.2).



Рис. 11.2. При вызове конечной точки, предъявленной предоставленным проектом da-ch11-ex1, приложение создает большое количество экземпляров, которые потребляют значительную часть памяти приложения. Мы будем анализировать этот сценарий, используя профилировщик



Чтобы начать работу с проектом, выполните следующие шаги:

- 1) запустите проект da-ch11-ex1;
- 2) запустите VisualVM;
- 3) выберите процесс, соответствующий проекту da-ch11-ex1 в VisualVM;
- 4) перейдите на вкладку **Monitor** в VisualVM;
- 5) вызовите конечную точку `/products/1000000`;

- 6) откройте для наблюдения виджет **Memory** (Память) на вкладке **Monitor** в VisualVM.

В виджете **Memory** на вкладке **Monitor** можно видеть, что приложение использует огромный объем ресурсов памяти. Внешний вид виджета показан на рис. 11.1. Что мы должны сделать, если предполагаем, что некоторая функция приложения неоптимально использует ресурсы памяти? Процесс анализа включает два основных шага:

- 1) использование выборки памяти для получения подробной информации об экземплярах объектов, хранимых приложением;
- 2) использование профилирования (инструментовки) для получения дополнительных подробностей о конкретной части выполняемого кода.

Применим методику, которую вы изучили в главах 7–9, для анализа потребления ресурсов ЦП: получение высокоуровневого визуального представления происходящего с помощью выборки. Чтобы получить выборку выполнения приложения для использования памяти, в профилировщике VisualVM откройте вкладку **Sampler**. Затем щелкните по кнопке **Memory**, чтобы начать сеанс выборки использования памяти. Вызовите конечную точку и подождите, когда завершится выполнение. В окне VisualVM будут показаны объекты, размещенные приложением.

Мы ищем то, что занимает бо́льшую часть памяти. Как правило, это один из двух вариантов:

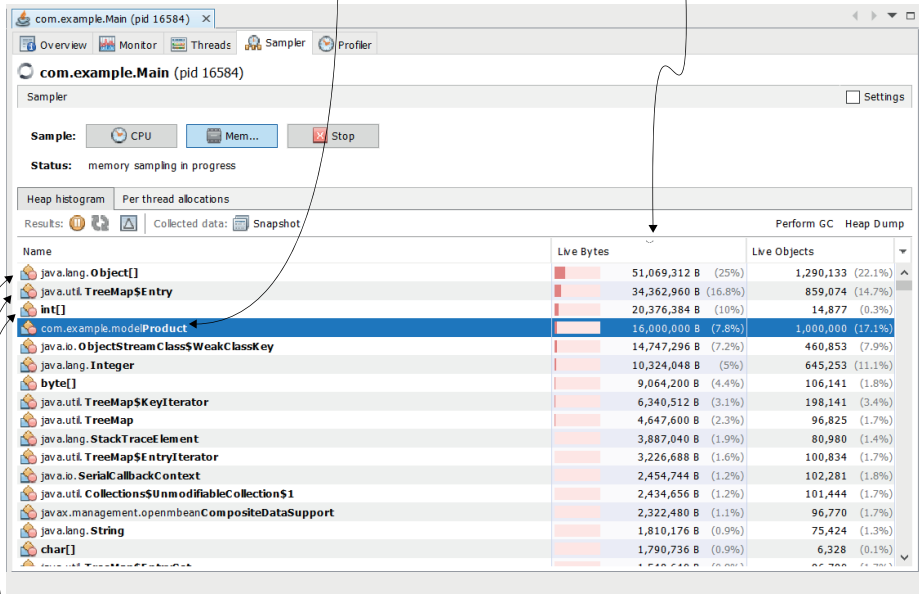
- создано много экземпляров объектов определенных типов, которые заполняют память (именно это произошло в нашем сценарии);
- экземпляров определенного типа не очень много, но каждый экземпляр очень большой.

Множество экземпляров, заполняющих выделенную память, можно понять, но как такое может произойти при небольшом количестве экземпляров? Представьте себе такой сценарий: приложение обрабатывает большие видеофайлы и, возможно, одновременно загружает два или три файла, но поскольку они весьма велики, то заполняют всю выделенную память. У разработчика есть возможность проанализировать, можно ли оптимизировать эту функциональность. Возможно, нет необходимости загружать в память приложения файлы полностью, но можно поочередно обрабатывать их фрагменты.

Начиная анализ, мы не знаем, в какой сценарий попадем. Обычно я выполняю сортировку в убывающем порядке по размеру занятой памяти, а затем по количеству экземпляров. На рис. 11.3 обратите внимание: VisualVM показывает потребление памяти и количество экземпляров для каждого типа в выборке. Эти данные необходимо отсортировать в убывающем порядке по второму и третьему столбцам таблицы.

2. Поиск первого объекта типа, принадлежащего к кодовой базе приложения или библиотеки, используемой приложением. На типы из JDK не обращаем внимания.

1. Сортировка в убывающем порядке по размеру занятой памяти.



3. Не учитываются простейшие типы, массивы простейших типов и JDK-объекты.

Рис. 11.3. Мы сортируем результаты выборки в убывающем порядке по размеру занятой памяти. Таким образом, можно увидеть, какие объекты потребляют большую часть памяти. Обычно мы вообще не обращаем внимания на простейшие типы, строки и массивы строк или на JDK-объекты. Больше всего нас интересует обнаружение объекта, напрямую связанного с кодовой базой приложения, из-за которого возникла проблема. В данном случае объект типа `Product` (являющийся частью кодовой базы приложения) занимает весьма большую часть памяти

На рис. 11.3 ясно видно, что выполнена сортировка таблицы в убывающем порядке по столбцу `Live Bytes` (т. е. по занятому пространству памяти). Затем мы ищем тип из кодовой базы приложения, который первым встречается в таблице. Не обращаем внимания на простейшие типы, строки, массивы простейших типов и массивы строк. Они обычно находятся в верхней части таблицы, поскольку создаются как побочный эффект. Но в большинстве случаев все вышеперечисленные типы не предоставляют никакой полезной информации о проблеме.

На рис. 11.3 очевидно, что причиной возникновения проблемы является тип `Product`. Он занимает весьма большую часть выделенной памяти, а в столбце `Live Objects` мы видим, что приложение создало миллион экземпляров этого типа.

Если требуется узнать общее количество экземпляров конкретного типа, созданное во время выполнения, то необходимо использовать методики профилирования (инструментовки). Мы рассмотрим их немного позже в этой главе.

Рассматриваемое здесь приложение – это всего лишь пример, но в реальном промышленном приложении простой сортировки по занятому пространству памяти может оказаться недостаточно. Требуется точно определить, является ли причиной проблемы большое количество экземпляров или огромное пространство памяти, занятое каждым экземпляром. Знаю, о чем вы думаете: разве это не ясно в данном случае? Ясно, но в реальном промышленном приложении это может оказаться не таким очевидным, поэтому я всегда рекомендую разработчикам также выполнять сортировку в убывающем порядке по количеству экземпляров для полной уверенности. На рис. 11.4 показаны данные выборки, отсортированные в убывающем порядке по количеству экземпляров каждого типа, созданных приложением. И в этом случае тип `Product` занимает первое место.

2. Поиск первого объекта типа, принадлежащего кодовой базе приложения или библиотеки, используемой приложением. Не учитываются типы из JDK.

1. Сортировка в убывающем порядке по количеству экземпляров объектов (живых объектов).

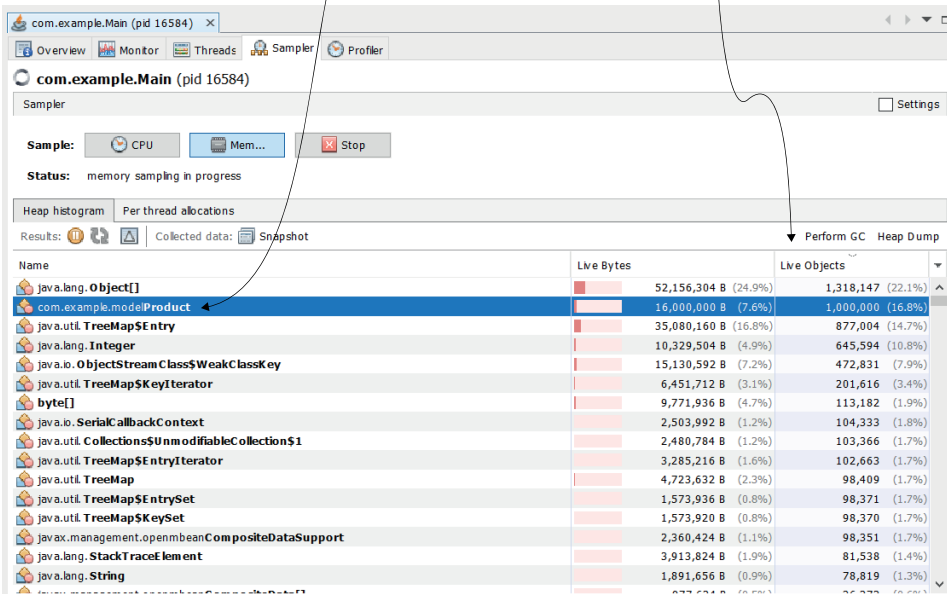


Рис. 11.4. Можно отсортировать результаты выборки по количеству экземпляров (живых объектов). Это помогает понять, не создает ли некоторое функциональное средство слишком большое количество объектов, отрицательно воздействуя на распределение памяти

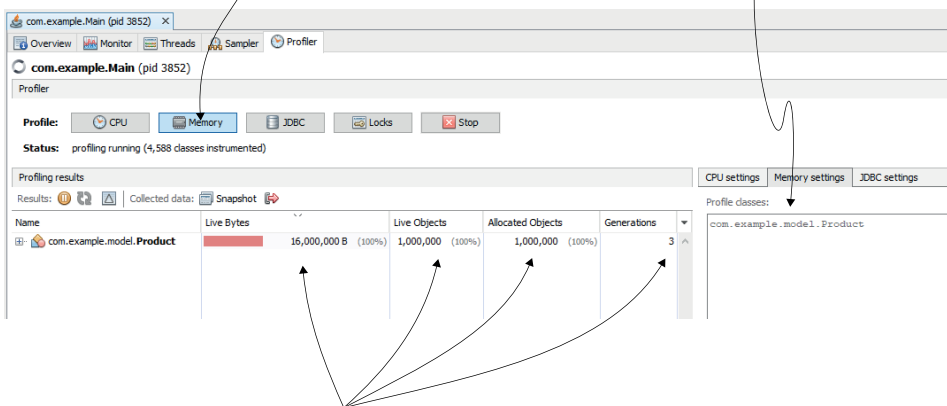


ПРИМЕЧАНИЕ. Профилировщик называет их **Live Objects** (Живые объекты), потому что выборка показывает только те экземпляры, которые продолжают существовать в памяти.

Иногда выборки достаточно для идентификации проблемы. Но что, если невозможно определить, какая часть приложения создает эти объекты? Если невозможно обнаружить источник проблемы только выборкой выполнения, то следующим этапом становится профилирование (инструментовка). Профилирование предоставляет больше подробностей, включая часть кода, создающую потенциально проблематичные экземпляры. Но при этом следует помнить простое практическое правило: при использовании профилирования сначала необходимо точно узнать, что именно нужно профилировать. Поэтому мы всегда начинаем с выборки.

Так как мы уже знаем, что проблема связана с типом `Product`, будем профилировать этот тип. Как в главах 7–9, необходимо указать, какую часть приложения вы намерены профилировать, используя (условное) выражение. На рис. 11.5 выполняется профилирование только для типа `Product`. Этот тип выбран с помощью полностью квалифицированного имени (пакета и класса) класса на панели настроек **Memory settings** в правой части окна.

2. Начать профилирование, затем вызвать конечную точку приложения.
1. Ввод выражения, определяющего, какие объекты необходимо профилировать по использованию памяти.



3. Профилировщик показывает подробности о каждом объекте, участвующем в выполнении во время сеанса профилирования. Вы найдете здесь размер памяти, выделенной для каждого объекта, количество экземпляров каждого объекта, существующих в памяти, количество объектов, удаленных сборщиком мусора и остающихся в памяти, а также количество попыток GC удалить объекты из памяти.

Рис. 11.5. Для профилирования распределения памяти сначала нужно указать, какие пакеты и/или классы вы намерены профилировать, а затем начать профилирование, щелкнув по кнопке **Memory**. Профилировщик предоставляет важные подробности об указанных типах, включая используемую память, количество экземпляров, общее количество размещенных объектов и количество генераций GC

Как и при профилировании ЦП (в главе 8), одновременно можно профилировать несколько типов и даже задавать целые пакеты. Ниже приведены примеры наиболее часто используемых выражений:

- точно определенный тип с полностью квалифицированным именем (например, `com.example.model.Product`) – поиск только для этого заданного типа;
- типы в конкретном пакете (например, `com.example.model.*`) – поиск только для типов, объявленных в пакете `com.example.model.*`, но не в его подпакетах;
- типы в конкретном пакете и в его подпакетах (например, `com.example.**`) – поиск в указанном пакете и во всех его подпакетах.



ПРИМЕЧАНИЕ. Всегда следует помнить о необходимости ограничения профилируемых типов, когда это возможно. Если известно, что `Product` создает проблему, то имеет смысл профилировать только этот тип.

В дополнение к живым объектам (т. е. к экземплярам того же типа, продолжающим существовать в памяти) вы получаете общее количество экземпляров этого типа, созданных приложением. Кроме того, вы увидите, сколько раз эти экземпляры «спасались» от GC (это называется generations (генерации)).

Все эти подробности полезны, но чаще всего еще более полезным является определение части кода, создающей проблемные объекты. Как показано на рис. 11.6, для каждого профилируемого типа профилировщик показывает, где были созданы его экземпляры. Щелкните по значку плюс (+) слева от строки в таблице. Эта функция сразу показывает главную причину возникновения проблемы.

11.2. Использование дампов кучи для поиска утечек памяти

Если приложение выполняется, то у вас есть возможность профилировать его, чтобы определить любой функциональный компонент, который можно оптимизировать. Но что, если приложение завершилось аварийно, и вы предполагаете, что это произошло из-за проблемы с распределением памяти? В большинстве случаев причинами аварийного завершения приложений являются функциональные компоненты с проблемами распределения памяти, такими как утечки памяти, – приложение не удаляет ранее созданные в памяти объекты, даже когда они становятся ненужными. Поскольку память не бесконечна, непрерывное размещение в ней объектов в некоторый момент заполнит память до отказа, что приведет к аварийному

завершению приложения. В JVM-приложении об этом свидетельствует исключение `OutOfMemoryError`, генерируемое во время выполнения.

Для каждого профилируемого типа объекта профилировщик показывает часть кода, создающего такой объект во время выполнения. Таким образом, можно найти потенциальную проблему.

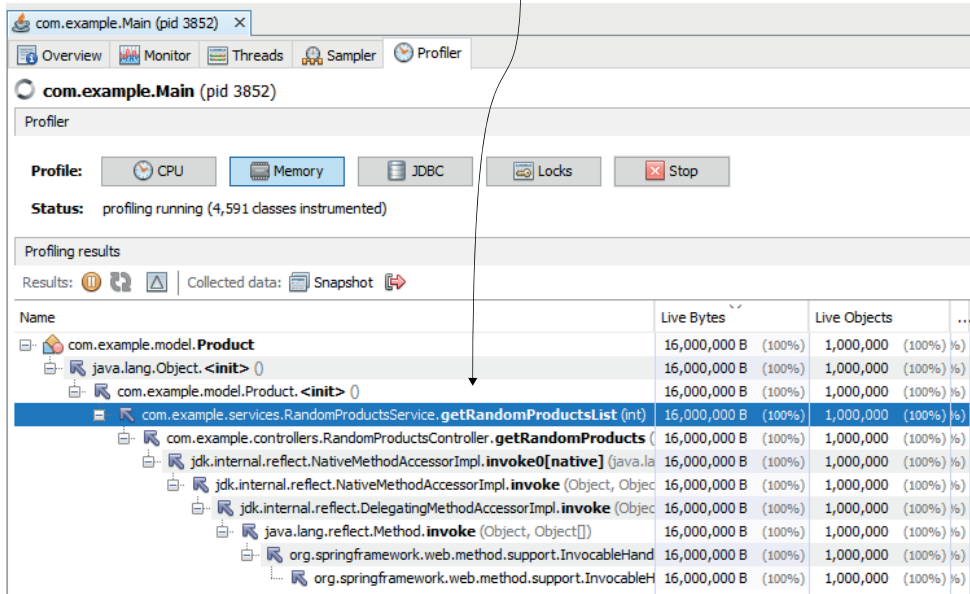


Рис. 11.6. Профилировщик показывает трассировку стека для кода, создающего экземпляры каждого профилируемого типа. Таким образом, можно с легкостью определить, какая часть приложения создала проблемные экземпляры

Если приложение не выполняется, то вы не можете подключить профилировщик для анализа выполнения. Но даже в этой ситуации существуют другие варианты анализа проблемы. Можно использовать дамп кучи (heap dump) – моментальный снимок состояния памяти кучи в момент аварийного завершения приложения. Получить дамп кучи можно в любое время, но он наиболее полезен, когда профилирование приложения невозможно по некоторой причине – возможно, из-за его аварийного завершения или просто при отсутствии прав доступа для профилирования процесса, и вам необходимо узнать, являются ли причиной аварии какие-либо проблемы с распределением памяти.

В следующем подразделе мы рассмотрим три возможных способа получения дампа кучи, а в подразделе 11.2.2 я покажу, как использовать дамп кучи для идентификации проблем с распределением памяти и главных причин их возникновения. В подразделе 11.2.3 будет описан более продвинутый метод чтения дампа кучи с использованием языка запросов под названием Object Query Language (OQL). Язык OQL похож на SQL, но вместо

запросов в базу данных вы используете OQL для запросов данных в дампе кучи.

11.2.1. Получение дампа кучи

В этом подразделе мы рассмотрим три способа генерации дампа кучи:

- конфигурирование приложения для автоматической генерации дампа кучи в заданной локации, когда приложение завершается аварийно из-за проблемы с памятью;
- использование профилировщика (например, VisualVM);
- использование инструментальных средств командной строки (например, `jcmd` или `jmap`).

Получить дамп кучи можно даже программным методом. В некоторых фреймворках существуют функциональные средства, которые могут генерировать дамп кучи, что позволяет разработчикам интегрировать в приложение средства мониторинга. Более подробную информацию по этой теме можно получить из описания специального класса `HotSpotDiagnosticMXBean` в официальной документации Java API (<https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/HotSpotDiagnosticMXBean.html>).

В проекте `da-ch11-ex1` реализована конечная точка, которую вы можете использовать для генерации дампа кучи с применением класса `HotSpotDiagnosticMXBean`. Вызов этой конечной точки при помощи `cURL` или `Postman` приведет к созданию файла дампа:

```
curl http://localhost:8080/jmx/heapDump?file=dump.hprof
```

Конфигурирование приложения для генерации дампа кучи при возникновении проблемы с памятью

Разработчики часто используют дамп кучи для анализа аварийного завершения приложения, когда предполагают, что некорректное распределение памяти является причиной возникновения проблемы. Поэтому приложения чаще всего конфигурируются для генерации дампа кучи, показывающего, как выглядела память при аварийном завершении приложения. Вы должны всегда конфигурировать приложение для генерации дампа кучи, если оно прекращает работу из-за проблемы с распределением памяти. К счастью, такое конфигурирование выполняется просто. Необходимо всего лишь добавить пару аргументов JVM при запуске приложения:

```
-XX:+HeapDumpOnOutOfMemoryError  
-XX:HeapDumpPath=heapdump.bin
```

Первый аргумент `-XX:+HeapDumpOnOutOfMemoryError` сообщает приложению о необходимости генерации дампа кучи, когда возникает ошибка `OutOfMemoryError` (куча заполнена до отказа). Вторым аргументом `-XX:HeapDumpPath=`

heapdump.bin определяет путь в файловой системе, по которому будет сохранен файл дампа. В рассматриваемом здесь случае файл, содержащий дамп кучи, будет иметь имя *heapdump.bin* и размещаться рядом с выполняемым файлом приложения, от корня classpath (так как здесь использован относительный путь). Необходимо убедиться в том, что процесс обладает правом записи (write) в этот путь, чтобы обеспечить возможность сохранения файла дампа в заданной локации.

Следующий фрагмент показывает полную команду запуска приложения:

```
java -jar -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heapdump.bin app.jar
```

Мы будем использовать демоприложение с именем da-ch11-ex2 для применения описанной выше методики. Это приложение можно найти в проектах, приложенных к книге. Приложение в листинге 11.1 постоянно добавляет экземпляры типа Product в список до тех пор, пока память не будет заполнена до отказа.

Листинг 11.1. Генерация большого количества экземпляров, которые не могут быть удалены из памяти

```
public class Main {

    private static List<Product> products = new ArrayList<>();

    public static void main(String[] args) {
        Random r = new Random();
        while (true) {
            Product p = new Product();
            p.setName("Product " + r.nextInt());
            products.add(p);
        }
    }
}
```

❶ Этот цикл выполняется бесконечно.

❷ Добавление экземпляров в список до тех пор, пока память не заполнится до отказа.

В следующем фрагменте кода показано, как выглядит простой тип Product:

```
public class Product {

    private String name;

    // Здесь не показаны get- и set-методы.
}
```

Возможно, вы удивлены, почему здесь присваивается случайное имя каждому экземпляру товара. Это потребуется в дальнейшем при обсуждении чтения дампа кучи в подразделе 11.2.2. А сейчас нас интересует только способ генерации дампа кучи, чтобы узнать, почему это приложение заполняет свою память кучи за считанные секунды.

Можно воспользоваться IDE для запуска приложения и настройки аргументов. На рис. 11.7 показано, как настроить аргументы JVM в IntelliJ. Я добавил еще один аргумент `-Xmx` для ограничения памяти кучи приложения размером всего лишь в 100 Мб. Это уменьшит файл дампа кучи и упростит пример.

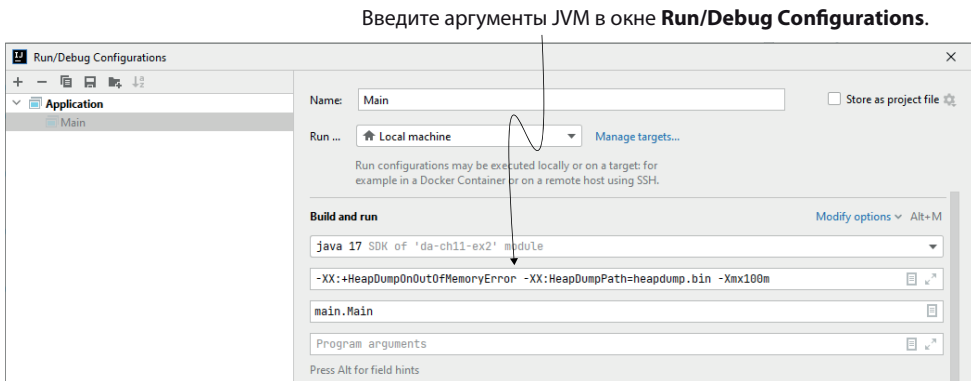


Рис. 11.7. Можно сконфигурировать аргументы JVM в IDE.

Добавьте значения в окне **Run/Debug Configurations**
(Конфигурации запуска/отладки) перед запуском приложения

После запуска приложения немного подождите, и оно завершится аварийно. При ограничении пространства кучи в 100 Мб потребуется не более нескольких секунд для заполнения памяти до отказа. В каталоге проекта содержится файл с именем *heapdump.bin*, в который включены все подробности о данных в куче в момент прекращения работы приложения. Этот файл можно открыть с помощью VisualVM, чтобы проанализировать его, как показано на рис. 11.8.

Получение дампа кучи с использованием профилировщика

Иногда необходимо получить дамп кучи для работающего процесса. В этом случае самое простое решение – применить VisualVM (или аналогичный инструмент профилирования) для генерации дампа. Получение дампа кучи с помощью VisualVM не сложнее щелчка по кнопке. Просто используйте кнопку **Heap Dump** (Дамп кучи) на вкладке **Monitor**, как показано на рис. 11.9.

Используйте кнопку **Load**, чтобы найти файл там, где он был сохранен после генерации. Затем откройте этот файл.

После открытия файла VisualVM покажет его в отдельной вкладке.

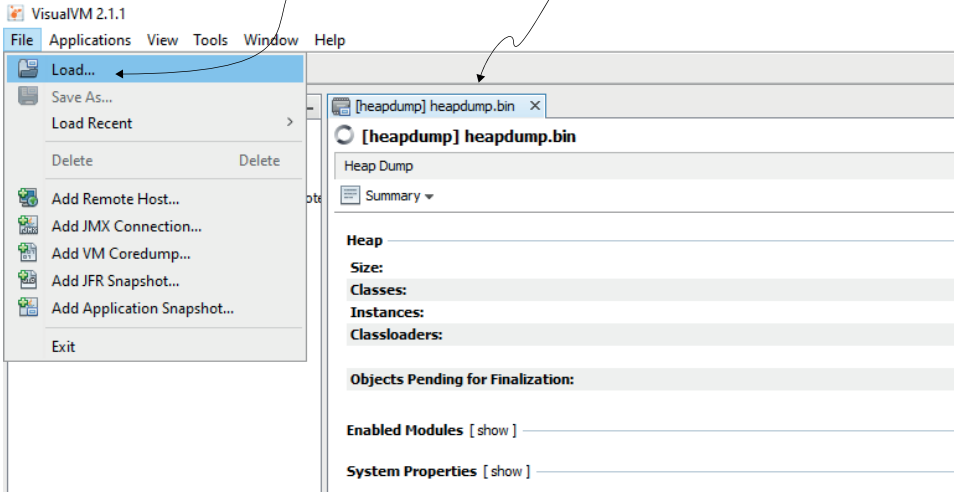


Рис. 11.8. Можно воспользоваться VisualVM, чтобы открыть файл дампа кучи для анализа. Используйте кнопку **Load** в меню, чтобы найти нужный файл. Откройте файл, и VisualVM покажет дамп кучи в отдельной вкладке

Щелкните по кнопке **Heap Dump** на вкладке **Monitor** для получения дампа кучи. VisualVM открывает дамп в отдельной вкладке, и вы получите возможность продолжить анализ или сохранить его в любом месте по своему усмотрению.

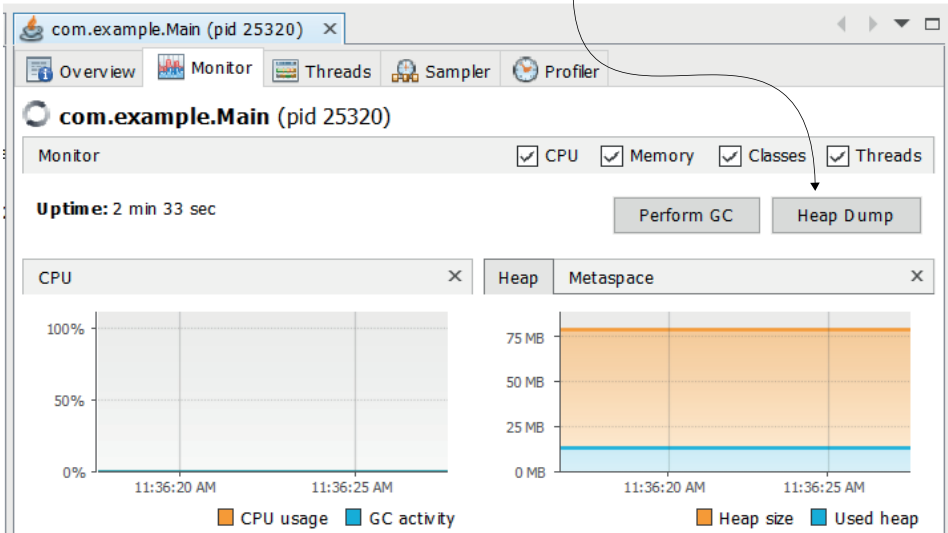


Рис. 11.9. Щелкните по кнопке **Heap Dump** на вкладке **Monitor** в VisualVM для получения дампа кучи для выбранного процесса. VisualVM откроет дамп в отдельной вкладке, и вы можете продолжить анализ или сохранить его в любом месте по своему усмотрению

Получение дампа кучи с использованием командной строки

Если необходимо получить дамп кучи для работающего процесса, но приложение развернуто в среде, к которой у вас нет доступа, позволяющего установить соединение профилировщика с приложением, не беспокойтесь – существуют и другие варианты. Можно использовать `jmap` – инструмент командной строки, предоставляемый JDK, для генерации дампа кучи.

Для получения дампа кучи с помощью `jmap` необходимо выполнить два шага:

- 1) найти идентификатор процесса (PID) выполняющегося приложения, для которого нужно получить дамп кучи;
- 2) использовать `jmap` для сохранения дампа в файле.

Чтобы найти PID работающего процесса, можно воспользоваться инструментом `jps`, как это было сделано в главе 10:

```
jps -l
25320 main.Main
132 jdk.jcmd/sun.tools.jps.Jps
25700 org.jetbrains.jps.cmdline.Launcher
```

Второй шаг – использование `jmap`. Для вызова `jmap` укажите PID и локацию, в которой будет сохранен файл дампа кучи. Кроме того, обязательно также определить, что выходными данными является двоичный (бинарный) файл, используя параметр `-dump:format=b`. На рис. 11.10 показано применение этого инструмента в командной строке.

1. Определение формата дампа: в данном случае `format=b` означает экспорт дампа в формате двоичного файла.

2. Определение пути для сохранения файла, содержащего дамп кучи.

```
C:\Program Files\Java\jdk-17.0.1\bin>jmap -dump:format=b,file=C:/DA/heapdump.bin 25320
Dumping heap to C:\DA\heapdump.bin ...
Heap dump file created [58079103 bytes in 0.259 secs]
```

3. Указание идентификатора процесса, для которого должен быть получен дамп кучи.

Рис. 11.10. Использование `jmap` в командной строке для получения дампа кучи. Необходимо задать путь для сохранения файла, содержащего дамп, и идентификатор процесса, для которого генерируется дамп.

Инструмент сохраняет дамп кучи как двоичный файл в заданной локации

Скопируйте показанную ниже строку кода, чтобы без затруднений выполнить требуемую команду:

```
jmap -dump:format=b,file=C:/DA/heapdump.bin 25320
```

Теперь можно открыть файл, сохраненный с помощью `jmap`, в VisualVM для анализа.

11.2.2. Чтение дампа кучи

В этом подразделе мы сосредоточимся на использовании дампа кучи для анализа проблем с распределением памяти. Дамп кучи похож на моментальный снимок памяти в момент генерации дампа. Он содержит все данные, которые приложение хранило в куче, а это означает, что дамп можно использовать для исследования данных и способа их структуризации. Таким образом, вы можете определить, какие объекты занимают большую часть выделенной памяти, и понять, почему приложение не смогло удалить их из памяти.



ПРИМЕЧАНИЕ. Следует помнить, что в таком «мгновенном снимке» (дампе кучи) можно увидеть все. Если незашифрованные пароли или любые секретные данные находятся в памяти, то любой человек, получивший в свое распоряжение дамп кучи, сможет увидеть эти подробности.

В отличие от дампа потоков дамп кучи невозможно анализировать как обычный текст. Вместо этого непременно требуется VisualVM (или любой инструмент профилирования). В этом подразделе мы будем использовать VisualVM для анализа дампа кучи, сгенерированного для проекта da-ch11-ex2 в подразделе 11.2.1. Вы научитесь применять эту методику для поиска главной причины возникновения ошибки `OutOfMemoryError`.

После открытия дампа кучи в VisualVM профилировщик показывает общий вид дампа (см. рис. 11.11), в котором кратко представлены основные подробности о файле дампа (например, размер файла, общее количество классов, общее количество экземпляров в дампе). Эту информацию можно использовать, чтобы убедиться в том, что вы получили корректный дамп в том случае, если его создавал кто-то другой.

Когда-то мне пришлось анализировать дампы кучи, предоставленные группой сопровождения, которой был разрешен доступ к средам, где выполнялось приложение. Но сам я не мог получить доступ к этим средам, поэтому был вынужден полагаться на других людей, передающих мне данные. Я неоднократно с удивлением обнаруживал, что получил неверный дамп кучи. Ошибку можно было определить, сравнив размер полученного дампа с известным мне максимальным значением, сконфигурированным для этого процесса, или даже просто проверив версию операционной системы или Java.

Рекомендую всегда сначала быстро проверить страницу общего обзора и убедиться в том, что вы получили корректный файл дампа. На странице обзора вы также найдете типы, занимающие слишком большой объем памяти. Обычно я не слишком полагаюсь на быстрый обзор и сразу перехожу к представлению объектов, где начинаю анализ. В большинстве случаев для меня общий обзор недостаточен, чтобы сделать какой-либо вывод.

Для реального промышленного приложения дамп кучи обычно намного больше, чем показанный в этом примере.

В обзоре приведены краткие сведения о дампе и о среде, в которой выполнялось приложение.

[heapdump] heapdump.bin

Heap Dump

Summary

Heap		Environment	
Size:	127,315,095 B	System	Windows 10 (10.0)
Classes:	975	Architecture:	amd64 64bit
Instances:	3,661,862	Java Home:	C:\Program Files\Java\jdk-17.0.1
Classloaders:	3	Java Version:	17.0.1
GC Roots:	944	Java Name:	1 (17.0.1+12-39, mixed mode, sharing)
Objects Pending for Finalization:	0	Java Vendor:	Oracle Corporation
		JVM Uptime:	n/a

Enabled Modules [show]

System Properties [show]

OutOfMemoryError Thread [view all]

This heap dump has been created automatically on an OutOfMemoryError thrown in this thread:

"main" prio=5 tid=1 RUNNABLE

Classes by Number of Instances [view all]			Classes by Size of Instances [view all]		
byte[]	1,218,526	(33.3%)	byte[]	51,203,427 B	(40.2%)
java.lang.String	1,218,429	(33.3%)	java.lang.String	36,552,870 B	(28.7%)
model.Product	1,215,488	(33.2%)	model.Product	29,171,712 B	(22.9%)
java.util.HashMap\$Node	1,194	(0%)	java.lang.Object[]	9,860,720 B	(7.7%)
java.util.concurrent.ConcurrentHashMap\$Node	1,149	(0%)	java.util.HashMap\$Node	56,824 B	(0%)

Instances by Size [view all]			Dominators by Retained Size [view all]	
java.lang.Object[]#1141 [G	9,723,920 B	(7.6%)	Retained sizes must be computed first:	
java.util.concurrent.ConcurrentHashMap\$Node	16,408 B	(0%)	<button>Compute Retained Sizes</button>	
char[]#1 [GC root - Java fr	16,408 B	(0%)		
char[]#10 : ...	16,408 B	(0%)		
char[]#11 : ...	16,408 B	(0%)		

Обзор представляет краткую информацию о типах, занимающих большую часть памяти или создавших большое количество экземпляров.

Рис. 11.11. На начальном экране после открытия дампа кучи VisualVM предоставляет общий обзор, включающий информацию о самом дампе и о системе, в которой работало приложение. Здесь также показаны типы, занимающие наибольший объем памяти

Для перехода к представлению объектов выберите пункт **Objects** (Объекты) из спускающегося меню в левом верхнем углу вкладки дампа кучи (см. рис. 11.12). Это позволит начать анализ экземпляров объектов в дампе кучи.

Чтобы перейти к представлению всех типов объектов в дампе кучи, выберите в меню пункт **Objects**.

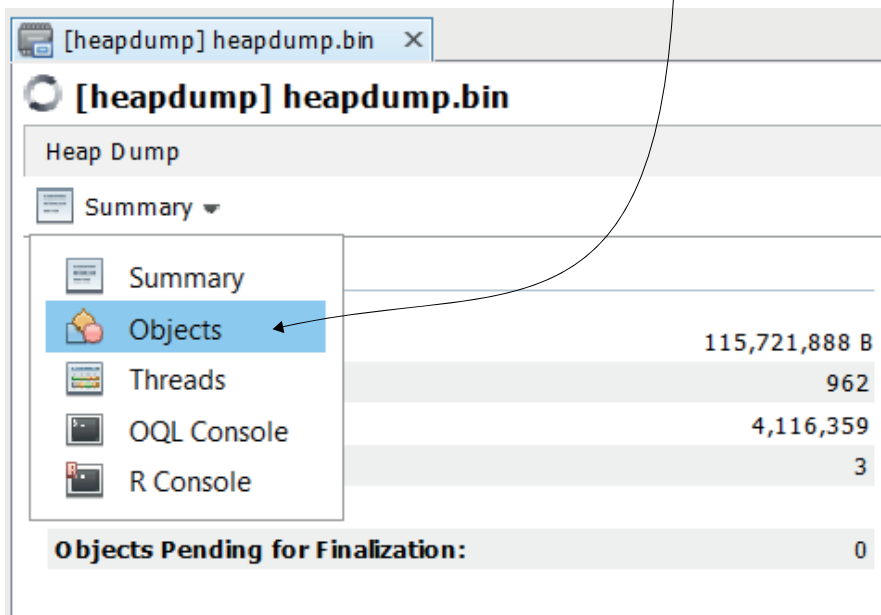


Рис. 11.12. Вы можете перейти к представлению **Objects**, что позволит упростить анализ экземпляров в дампе кучи

Как и при выборке и профилировании памяти, мы ищем типы, использующие наибольший объем памяти. Самый лучший способ – сортировка в убывающем порядке по количеству экземпляров и размеру занятой памяти и определение первых типов, являющихся частью кодовой базы приложения. Не обращайте внимание на простейшие типы, строки и массивы простейших типов и строк. Как правило, их очень много, но они не дают практически никакой информации о существующей проблеме.

На рис. 11.13 можно видеть, что после сортировки тип `Product` кажется имеющим отношение к анализируемой проблеме. Тип `Product` – первый тип, являющийся частью кодовой базы приложения, и он использует большую часть памяти. Необходимо выяснить, почему было создано так много экземпляров и почему сборщик мусора (GC) не может удалить их из памяти.

Можно щелкнуть по небольшому значку плюс (+) слева от строки, чтобы увидеть подробности обо всех экземплярах этого типа. Мы уже знаем, что создано более миллиона экземпляров `Product`, но теперь нужно определить:

- какая часть кода создает эти экземпляры;
- почему сборщик мусора (GC) не может удалить их вовремя, чтобы избежать аварийного завершения приложения.

Ищите тип объекта из кодовой базы приложения, который занимает наибольший объем памяти.

Name	Count	Size	Retained
byte[]	1,218,526 (33.3%)	51,203,427 B (40.2%)	51,199,918 B (40.2%)
java.lang.String	1,218,429 (33.3%)	36,552,870 B (28.7%)	87,705,789 B (68.9%)
modelProduct	1,215,488 (33.2%)	29,171,712 B (22.9%)	116,665,403 B (91.6%)
java.util.HashMap\$Node	1,194 (0%)	52,536 B (0%)	104,234 B (0.1%)
java.util.concurrent.ConcurrentHashMap\$Node	1,149 (0%)	50,556 B (0%)	68,717 B (0.1%)
java.lang.Object[]	1,141 (0%)	9,860,720 B (7.7%)	126,583,190 B (99.4%)
java.lang.module.ModuleDescriptor\$ExportPackage	370 (0%)	14,800 B (0%)	18,824 B (0%)
java.util.HashMap	328 (0%)	20,992 B (0%)	137,786 B (0.1%)
java.util.HashMap\$Node[]	325 (0%)	56,824 B (0%)	129,794 B (0.1%)
java.util.HashSet	264 (0%)	6,336 B (0%)	89,204 B (0.1%)
java.lang.Integer	262 (0%)	5,240 B (0%)	5,240 B (0%)

Рис. 11.13. Используйте сортировку по столбцам, чтобы определить, какой тип создал слишком большое количество экземпляров или занял огромный объем памяти. Всегда ищите первый объект из кодовой базы приложения. В рассматриваемом здесь случае первым в списке по числу экземпляров и по размеру занятой памяти является тип Product

Можно узнать, на что ссылается каждый экземпляр (через поля) и что ссылается на него. Поскольку известно, что GC не может удалить экземпляр из памяти, если существуют ссылки на него, мы будем искать такие ссылки, чтобы определить, действительно ли он остается необходимым в контексте обработки, или приложение забыло удалить ссылку на него.

На рис. 11.14 показано развернутое представление с подробной информацией об одном из экземпляров Product. Здесь можно видеть, что этот экземпляр ссылается на String (наименование товара), а ссылка на него сохранена в массиве Object, который является частью экземпляра ArrayList. Кроме того, очевидно, что экземпляр ArrayList содержит большое количество ссылок (более миллиона). Обычно это плохой признак, так как либо в приложении реализован неоптимизированный функциональный компонент, либо мы обнаружили утечку памяти.

Чтобы разобраться в этой ситуации, необходимо проанализировать код, применяя методики отладки и журналирования, рассмотренные в главах 2–5. К счастью, профилировщик точно показывает, где найти этот список в коде. В данном случае список объявлен как статическая переменная в классе Main.

Используя VisualVM, можно легко понять отношения между объектами. Объединяя эту методику с другими методиками анализа, уже освоенными при чтении книги, вы получаете в свое распоряжение все инструменты,

необходимые для устранения проблем таких типов. Сложные проблемы (и приложения), возможно, потребуют значительных усилий, но, применяя такой подход, вы сэкономите огромное количество времени.

Объект, ссылающийся на этот экземпляр Product, – ArrayList, который содержит 1 215 487 других ссылок.

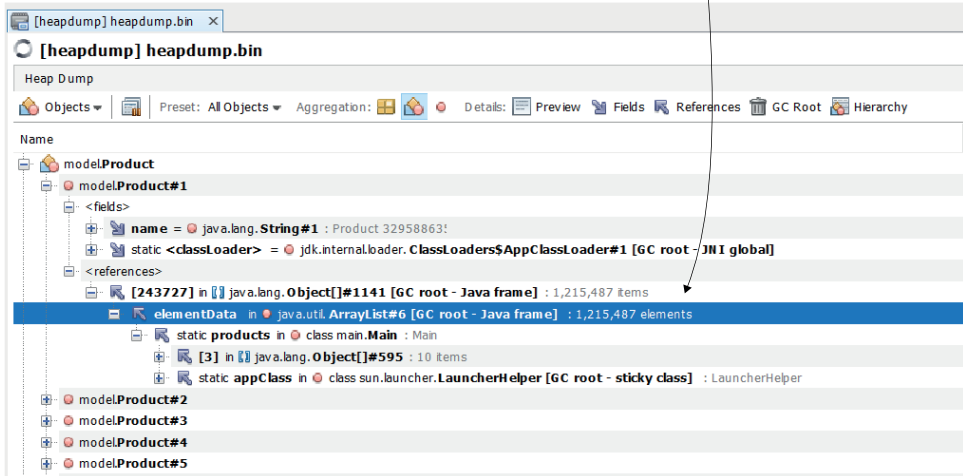


Рис. 11.14. Ссылки на экземпляр. Используя дамп кучи, для каждого экземпляра можно узнать, на какие другие экземпляры существовали ссылки в момент генерации дампа. Профилировщик также показывает, в какой части кода хранятся интересные нас ссылки. В рассматриваемом здесь случае ArrayList, содержащий более миллиона ссылок, – это статическая переменная в классе Main

11.2.3. Использование консоли OQL для запроса в дамп кучи

В этом подразделе мы рассмотрим более продвинутый способ анализа дампа кучи. Для извлечения подробностей из дампа кучи мы будем использовать язык запросов, похожий на SQL. Простых методик, описанных в подразделе 11.2.2, обычно достаточно для выявления главных причин возникновения проблем с распределением памяти. Но этого недостаточно, если необходимо сравнивать подробности двух и более дампов кучи.

Предположим, что требуется сравнить дампы кучи, созданные для двух или более версий приложения, чтобы определить, не было ли реализовано что-то ошибочное или неоптимизированное между релизами версий. Можно выполнить анализ вручную, поочередно сравнивая версии. Но я научу вас писать запросы, с легкостью выполняющиеся для каждой версии, и это позволит сэкономить рабочее время. Для этой цели превосходным инструментом является OQL. На рис. 11.15 показано, как перейти в окно консоли OQL, где можно выполнять запросы для анализа дампа кучи.

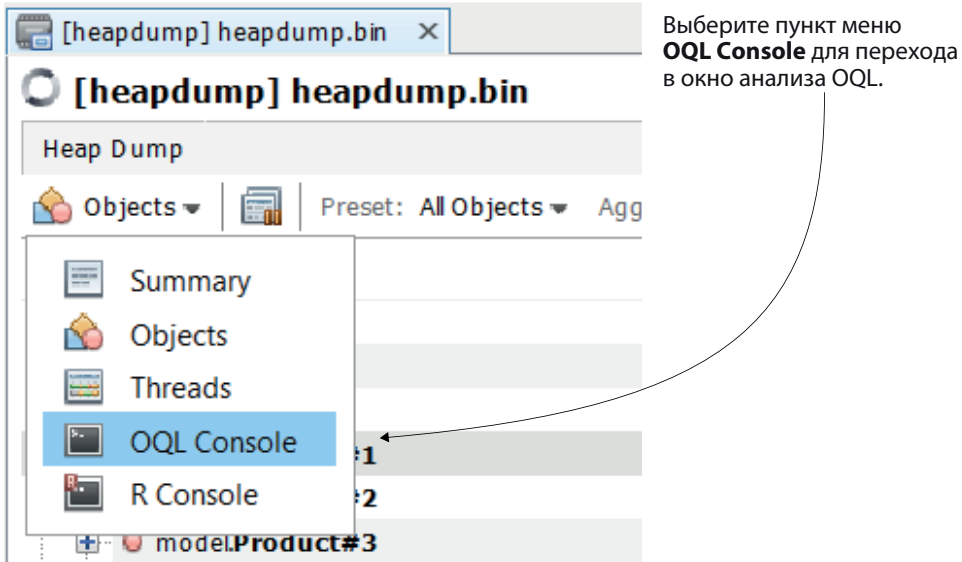


Рис. 11.15. Для перехода в окно консоли OQL в VisualVM выберите пункт **OQL Console** в спускающемся меню в верхнем левом углу вкладки дампа кучи

Мы рассмотрим несколько примеров, которые я считаю наиболее полезными, но следует помнить о том, что OQL – это более сложный язык запросов. (Более подробную информацию о функциях языка OQL можно найти здесь: https://cr.openjdk.org/~sundar/8022483/webrev.01/raw_files/new/src/share/classes/com/sun/tools/hat/resources/oqlhelp.html.)

Начнем с простого примера: выбора всех экземпляров заданного типа. Предположим, что необходимо получить все экземпляры типа `Product` из дампа кучи. При использовании SQL-запроса для получения всех записей о товаре из таблицы в реляционной базе данных мы должны написать что-то вроде:

```
select * from product
```

Для запроса всех экземпляров `Product` из дампа кучи с использованием OQL необходимо написать:

```
select p from model.Product p
```



ПРИМЕЧАНИЕ. В языке OQL ключевые слова, такие как `select`, `from` или `where`, всегда записываются в нижнем регистре. Типы всегда указываются с полностью квалифицированным именем (пакет + имя класса).

На рис. 11.16 показан результат выполнения простого запроса, который извлекает все экземпляры Product из дампа кучи.

3. После выполнения OQL-запроса результаты отображаются выше – на панели вывода.

1. Напишите OQL-запрос в текстовой панели.

2. Щелкните по кнопке **Run**.

Рис. 11.16. Выполнение OQL-запроса в VisualVM. В консоли OQL напишите OQL-запрос в текстовой панели в нижней части окна, затем щелкните по кнопке **Run** (Пуск) (зеленая стрелка слева от текстовой панели) для выполнения запроса. Результаты будут показаны на панели вывода



ПРИМЕЧАНИЕ. При изучении OQL используйте небольшие дампы кучи. Дампы кучи для реальных промышленных приложений обычно весьма велики (4 Гб и более). OQL-запросы будут работать медленно. Если вы просто изучаете язык OQL, то сгенерируйте и используйте дампы кучи небольшого размера, как рассматриваемый в этой главе.

Можно выбрать любой экземпляр, полученный по запросу, чтобы увидеть его подробности. Можно узнать, где хранятся ссылки на этот экземпляр, на что он ссылается и его значения (см. рис. 11.17).

В панели результатов выберите любую строку (представляющую экземпляр объекта), чтобы увидеть подробности об этом экземпляре.

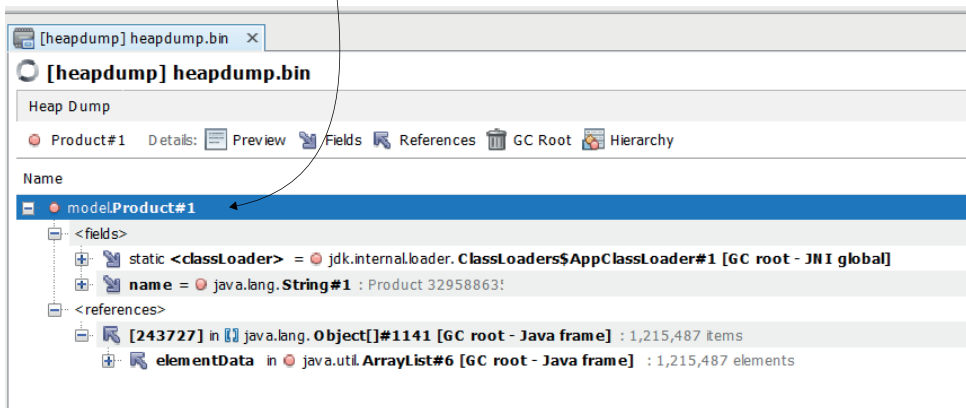


Рис. 11.17. Можно получить доступ к подробностям о любом запрошенном экземпляре (его ссылки и ссылки на него), щелкнув по его имени

Также можно выбрать значения или ссылки, исходящие из конкретных экземпляров. Например, если требуется получить все наименования товаров вместо их экземпляров, то можно написать такой запрос (см. рис. 11.18):

```
select p.name from model.Product p
```

С помощью OQL можно одновременно извлекать несколько значений. Для этого необходимо применить формат JSON, как показано в листинге 11.2.

Листинг 11.2. Использование формата JSON

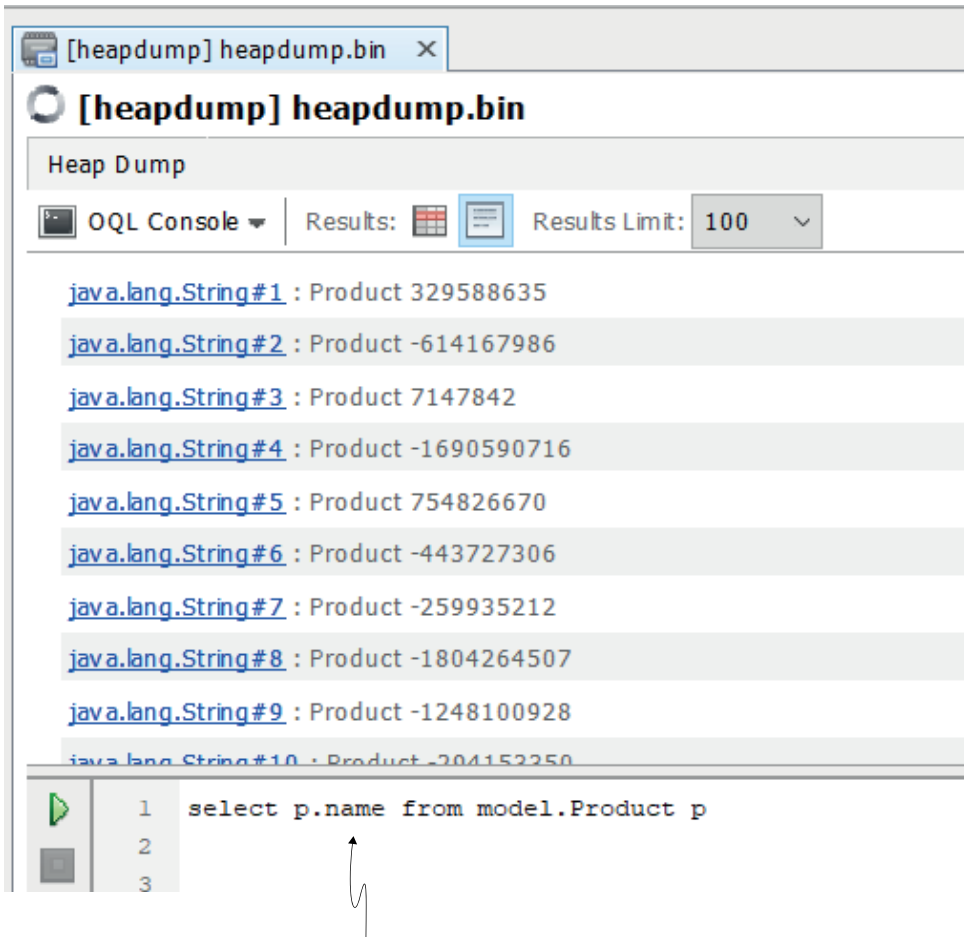
```
select

{
    name: p.name,           ❶
    name_length: p.name.value.length  ❷
}                             ❸

from model.Product p
```

- ❶ В фигурных скобках содержится JSON-представление объекта.
- ❷ Атрибут `name` принимает значение наименования товара.
- ❸ Атрибут `name_length` принимает значение количества символов в наименовании товара.

На рис. 11.19 показан результат выполнения этого запроса.



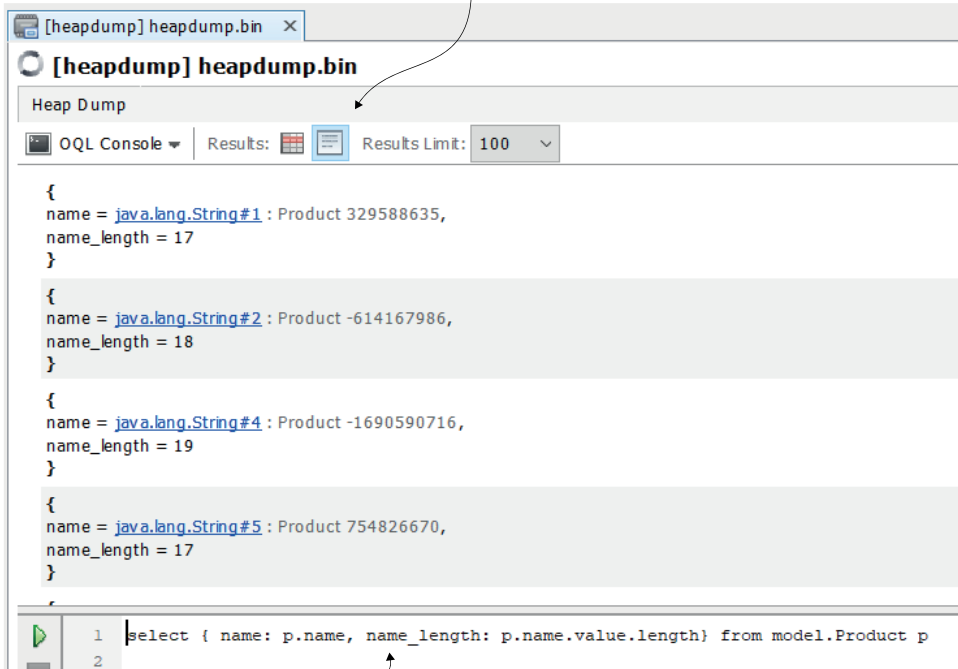
Можно выбрать любой атрибут конкретного объекта типа.

Рис. 11.18. Выбор атрибута конкретного объекта типа.
Как и в Java, можно использовать стандартный оператор «точка»
для ссылки на атрибут экземпляра

Этот запрос можно изменить, например добавить условия для одного или нескольких выбираемых значений. Допустим, что необходимо выбрать только экземпляры с именем длиннее 15 символов. Для этого можно написать следующий запрос:

```
select { name: p.name, name_length: p.name.value.length}
from model.Product p
where p.name.value.length > 15
```

Чтобы результаты отображались более понятно, используйте форматированный вывод.



Для выбора нескольких значений используйте формат JSON.

Рис. 11.19. Выбор нескольких значений. Можно использовать формат JSON для получения нескольких значений в одном запросе

Теперь перейдем к немного более сложным вещам. При анализе проблем с памятью я часто использую запрос с применением метода `referrers()`, чтобы получить объекты, ссылающиеся на экземпляры конкретного типа. Используя встроенные функции OQL, подобные `referrers()`, вы можете выполнять множество полезных операций:

- поиск или запрос экземпляров, ссылающихся на что-либо (`referees`), – может сообщить, имеются ли в приложении утечки памяти;
- поиск или запрос экземпляров, на которые имеются ссылки (`referrals`), – может сообщить, являются ли конкретные экземпляры причиной утечек памяти;
- поиск дублирующихся экземпляров – может сообщить, возможна ли оптимизация конкретных функциональных компонентов для использования меньшего объема памяти;
- поиск подклассов и суперклассов конкретных экземпляров – позволяет увидеть изнутри проектное решение класса приложения без необходимости чтения исходного кода;

- идентификация длительного времени жизни – может помочь в обнаружении утечек памяти.

Для получения всех неповторяющихся ссылок на экземпляры типа `Product` можно использовать следующий запрос:

```
select unique(referrers(p)) from model.Product p
```

На рис. 11.20 показан результат выполнения этого запроса. Здесь можно видеть, что на все экземпляры товаров ссылается один объект – список. Обычно, если на большое количество экземпляров ссылается малое число объектов, это признак утечки памяти. В данном случае список хранит ссылки на все экземпляры `Product`, предотвращая их удаление из памяти сборщиком мусора (GC).

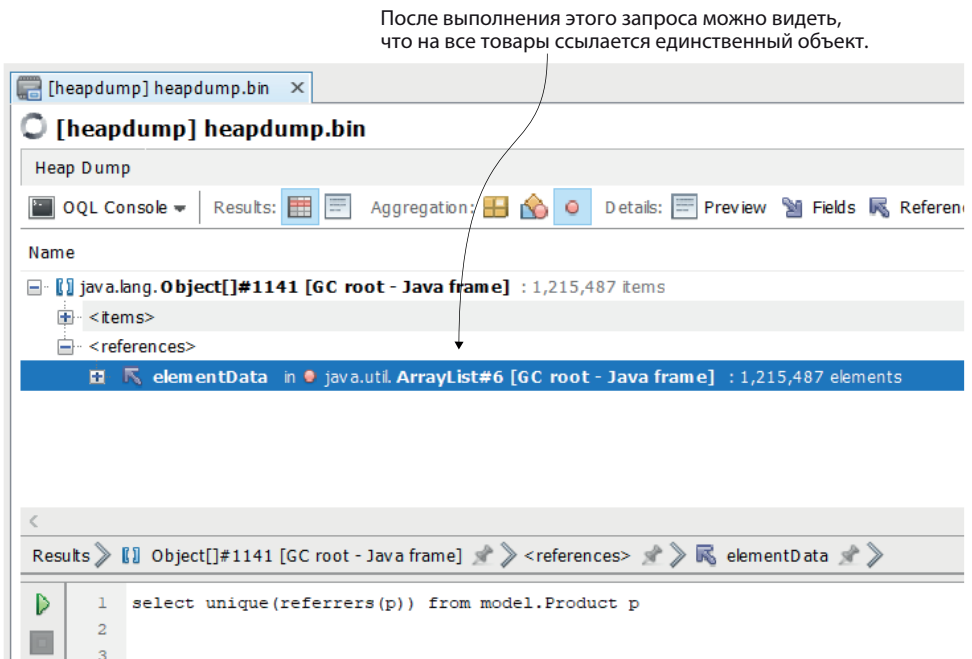


Рис. 11.20. Выбор всех неповторяющихся ссылок на экземпляры некоторого типа показывает, что существует один объект, не позволяющий GC удалить эти экземпляры из памяти. Это возможный быстрый способ обнаружить утечку памяти

Если результатом является не единственный объект, то можно подсчитать количество ссылок по экземплярам, используя приведенный ниже запрос для поиска экземпляров, которые, предположительно, имеют отношение к утечке памяти:

```
select { product: p.name, count: count(referrers(p))} from model.Product p
```

OQL-запросы предоставляют огромное количество возможностей, и, после того как вы написали запрос, его можно выполнять столько раз, сколько необходимо, и применять к различным дампам кучи.

11.3. Резюме

- В приложении с функциональными компонентами, которые не оптимизированы для корректного распределения памяти, могут возникнуть проблемы с производительностью. Оптимизация приложения для разумного (исключающего неоправданное расходование пространства памяти) размещения данных в памяти весьма важна для производительности приложения в целом.
- Профилировщик позволяет выполнить выборку и профилирование, чтобы понять, как распределяется память во время выполнения. Это может помочь при идентификации неоптимизированных частей приложения и предоставить подробную информацию о том, что можно улучшить.
- Если новые экземпляры объекта непрерывно добавляются в память во время выполнения, но приложение никогда не удаляет ссылки на новые экземпляры, то GC лишается возможности удалять такие объекты и освобождать память. Когда память заполняется до отказа, приложение не может продолжать выполнение и останавливает работу. Перед остановкой приложение генерирует исключение `OutOfMemoryError`.
- Для анализа ошибки (исключения) `OutOfMemoryError` мы используем дампы кучи. В дампе кучи собраны все данные из памяти кучи приложения, что позволяет проанализировать их, чтобы понять, из-за чего возникла проблема.
- Приложение можно запустить с двумя аргументами JVM, обеспечивающими генерацию дампа кучи и запись его в файл по заданному пути, если приложение завершается аварийно с ошибкой `OutOfMemoryError`.
- Также можно получить дамп кучи, используя профилировщик или инструмент командной строки, например `jmap`.
- Для анализа дампа кучи необходимо загрузить его в профилировщик, например `VisualVM`, который позволяет проанализировать все экземпляры из дампа и отношения между ними. Таким образом, можно определить, какая часть приложения не оптимизирована или содержит утечки памяти.
- `VisualVM` предоставляет более продвинутые способы анализа дампа кучи, например OQL-запросы. OQL – это язык запросов, похожий на SQL, используемый для извлечения данных из дампа кучи.

Часть III

Поиск проблем в крупных системах

В настоящее время большинство приложений является частью крупных систем, которые используют разнообразные архитектурные стили. В таких системах иногда очень трудно выявлять проблемы, и исследования одного процесса недостаточно. Предположим, что вы владелец автомобиля, и в вашем распоряжении имеются различные способы анализа его отдельных независимых частей, но нет способа, позволяющего определить, как эти части взаимодействуют между собой. Сможете ли вы обнаружить все возможные проблемы?

В части III мы будем рассматривать методики, применяемые для анализа систем, состоящих из нескольких (и даже многих) приложений. Мы сосредоточимся на исследовании способа, которым эти приложения «разговаривают» друг с другом, узнаем, как на них воздействует рабочая среда, в которой они развернуты, а также что необходимо учитывать при реализации таких систем.

Глава 12

Анализ поведения приложений в крупных системах

Темы:

- анализ проблем при обмене данными приложений;
- использование инструментов мониторинга журналов в вашей системе;
- использование преимуществ инструментов развертывания.

В этой главе мы перейдем границу единственного приложения и рассмотрим, как анализировать ситуации, созданные несколькими приложениями, совместно работающими в системах. В настоящее время многие системы состоят из нескольких приложений, обменивающихся данными друг с другом. Крупные бизнес-системы используют разнообразные приложения, которые часто реализованы с применением различных технологий на разных платформах. В большинстве случаев уровень развития, «зрелость» (maturity) таких приложений также варьируется от современных сервисов до старых запутанных скриптов.

Отладки, профилирования и ведения журналов здесь уже недостаточно. Иногда необходимо найти более обобщенные основные идеи и принципы. Приложение может превосходно работать в независимом режиме, но некорректно интегрироваться с другими приложениями или с рабочей средой, в которой оно развертывается.

В разделе 12.1 мы начнем изучение способов анализа обмена данными между сервисами системы. В разделе 12.2 наше внимание будет сосредоточено на важности реализации мониторинга для приложений в системе, а также на том, как использовать информацию, предоставляемую инструментами мониторинга. Глава завершается разделом 12.3, в котором рассматривается получение преимуществ от инструментов развертывания.

12.1. Анализ обмена данными между сервисами

В этом разделе мы рассмотрим анализ обмена данными между приложениями. В системе приложения «общаются» друг с другом, чтобы выполнить свои обязанности. До сих пор наше внимание было сосредоточено на анализе внутренней работы приложения и на обмене данными между приложением и базой данных. Но что, если приложения беседуют друг с другом? Существует ли способ мониторинга событий в целом во всей системе, состоящей из многих приложений (см. рис. 12.1)?

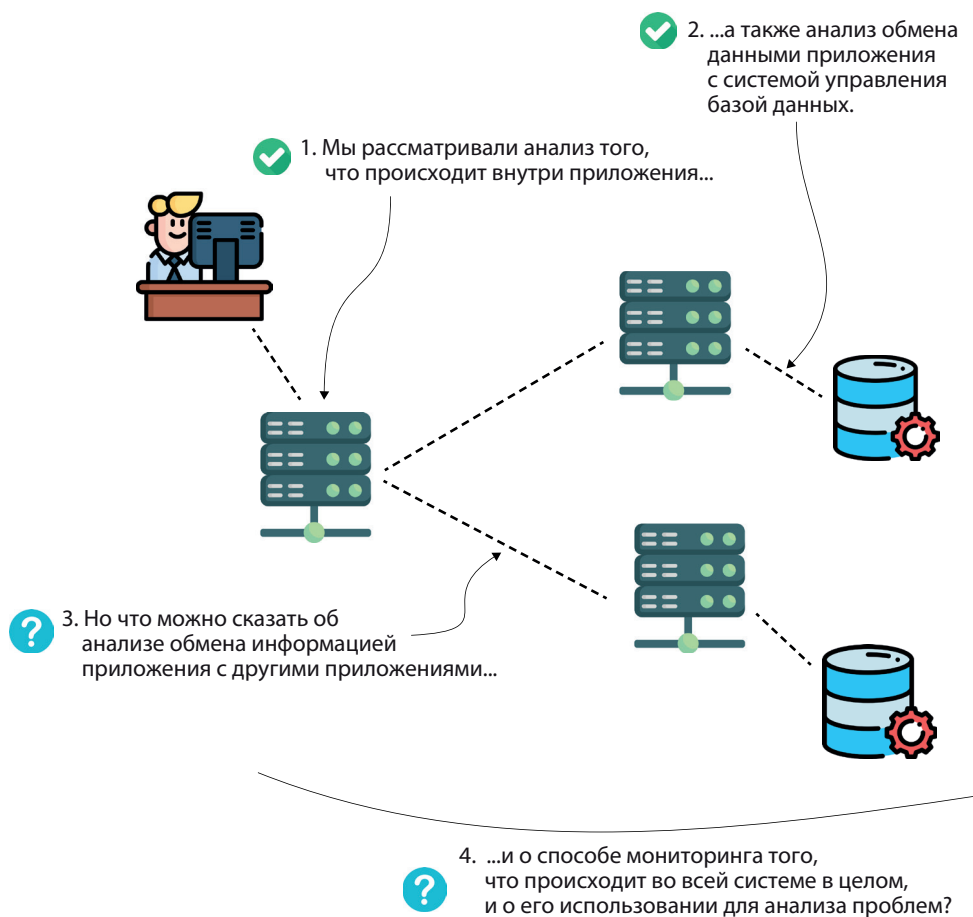


Рис. 12.1. Во многих случаях анализ выполняется в границах приложения.

Но может потребоваться выход за пределы того, что происходит внутри конкретного процесса. Общие системные проблемы или странное поведение могут быть вызваны приложениями, в которых существуют проблемы обмена данными друг с другом, и реализация инструмента мониторинга упрощает анализ именно этих типов проблем

Микросервисы

Уделим немного внимания микросервисам. Многие системы, с которыми вам придется работать, заявляют себя как микросервисы. Чаще всего это неправда – это просто сервис-ориентированные архитектуры. Микросервисы стали (не могу сказать, что до конца понимаю, по какой причине) брендом, который достаточно хорошо продается:

- вы хотите нанять сотрудников как можно быстрее? Скажите им, что они будут работать с микросервисами;
- хотите произвести впечатление на клиента во время предварительной встречи? Скажите ему, что вы занимаетесь микросервисами;
- хотите, чтобы на вашу презентацию пришло как можно больше людей? Вы уже догадались: просто добавьте «микросервисы» в заголовок.

Но микросервисы сложнее, чем их восприятие и интерпретация большинством разработчиков, насколько я вижу. Если вы хотите понять, чем в действительности являются микросервисы, то обнаружите изобилие литературы по этой теме. Можно начать с книги «*Microservices Patterns*» Криса Ричардсона (Chris Richardson) (Manning, 2018 г.), затем прочитать «*Monolith to Microservices*» Сэма Ньюмана (Sam Newman) (O'Reilly Media, 2018 г.) или «*Building Microservices: Designing Fine-Grained Systems*» (второе издание; O'Reilly Media, 2021 г.), автор также Сэм Ньюман (Sam Newman).

Неважно, являются системы микросервисами или нет, в любом случае вам потребуются знания о том, как анализировать проблемы и как быстрее понять, что делает система в конкретных сценариях. В этой главе мы будем рассматривать методики анализа, применимые к микросервисам, но не только к ним. Я предпочитаю простой термин «сервис» (service) вместо «микросервис» (microservice). Иногда я просто использую термин «приложение».

Рассмотрим методику использования инструмента профилирования для анализа проблем, относящихся к тому, как приложения «разговаривают» друг с другом. Мы будем применять JProfiler для наблюдения за обменом данными простого приложения (проект `da-ch12-ex1`) для предъявления конечной точки (`/demo`), которую вы можете вызвать. Когда вы отправляете HTTP-запрос в эту конечную точку, приложение передает запрос в конечную точку, предъявленную httpbin.org, что задерживает ответ на 5 с, а затем выдает ответ HTTP-состояния 200 OK.

Как вы узнаете из этого раздела, JProfiler предоставляет набор инструментальных средств, которые можно использовать для наблюдения за отправляемыми и получаемыми запросами приложения. Кроме того, при обмене данными можно анализировать события низкого уровня в сокетах.

Такие методики могут помочь в определении главной причины возникновения проблем при обмене данными.

В подразделе 12.1.1 мы будем использовать JProfiler для наблюдения запросов, принимаемых приложением. В подразделе 12.1.2 будет выполняться анализ подробностей запросов, отправляемых приложением, а в подразделе 12.1.3 мы сосредоточимся на анализе событий низкого уровня в сокетах при обмене данными.

12.1.1. Использование проб HTTP-сервера для наблюдения HTTP-запросов

Когда два приложения обмениваются информацией, поток данных является двунаправленным. Приложение либо отправляет, либо принимает запрос. Если приложение отправляет запрос, то мы говорим, что оно действует как клиент (client). Если приложение принимает запрос, мы называем его сервером (server). В этом подразделе сосредоточимся на HTTP-запросах, принимаемых приложением (как сервером). Здесь используется простое приложение, представленное в проекте da-ch12-ex1, чтобы понять, как применяется мониторинг подобных событий в JProfiler.

Откройте проект da-ch12-ex1 в своей IDE и запустите приложение. Используя JProfiler, установите соединение с приложением и начните запись принимаемых HTTP-запросов – перейдите на вкладку **HTTP Server > Events**, затем щелкните по значку со звездочкой для записи отдельных событий. На рис. 12.2 показано, как начать запись событий. Необходимо узнать, какие HTTP-запросы принимает приложение и какую информацию могут предоставить эти запросы.

Вызываем только конечную точку, которую предъявляет это демоприложение:

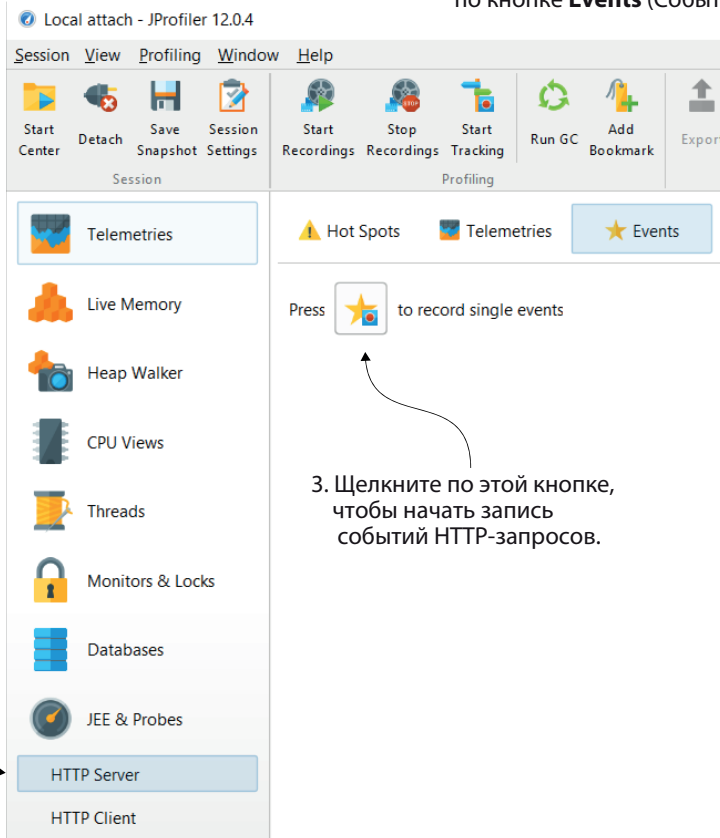
```
curl http://localhost:8080/demo
```

Как видно на рис. 12.3, JProfiler показывает запрос, принятый сервером. Сразу же можно с легкостью определить, когда завершается событие: выводится состояние «completed» (завершен). Если операция никогда не завершается, это означает, что по какой-то причине запрос не был полностью обработан или ответ не был отправлен клиенту, и выводится состояние «in progress» (в процессе обработки). Таким образом, можно определить, обрабатывается ли запрос слишком долго, или он был отложен или прерван по какой-то причине.

В таблице событий HTTP-сервера также показана продолжительность события. Если событие завершено, но заняло длительное время, то необходимо определить причину такой задержки. Возможно, это некорректный обмен данными, который вы будете наблюдать, используя события сокета, рассматриваемые в подразделе 12.1.3, или может потребоваться выборка и профилирование выполнения, описанные в главах 7–9.

1. Инструментовка запросов HTTP-сервера в секции JEE & Probes в JProfiler.

2. Для доступа к секции событий HTTP-запросов щелкните по кнопке **Events** (События).



3. Щелкните по этой кнопке, чтобы начать запись событий HTTP-запросов.

Рис. 12.2. Чтобы начать запись HTTP-запросов в JProfiler, перейдите на вкладку **HTTP Server > Events**, затем щелкните по значку со звездочкой для записи отдельных событий. После этого при приеме профилируемым приложением HTTP-запросов JProfiler будет выводить все подробности

Также важно узнать, сколько событий произошло в приложении. В некоторых случаях один запрос не создает проблемы, но я помню ситуацию, в которой на приложение воздействовал опрос (polling), выполняемый одним из клиентов (повторяющаяся отправка запросов в течение короткого времени) для одной из конечных точек. Если клиент отправляет большое количество запросов в течение короткого интервала времени и ничто не препятствует их приему приложением, то в приложении может возникнуть проблема, связанная с ответом на все запросы, и даже аварийное завершение.

Когда приложение получает HTTP-запрос, соответствующее событие появляется в таблице **Events** и вы можете наблюдать его подробности.

Состояние события показывает, остается ли событие в процессе обработки, или оно завершено.

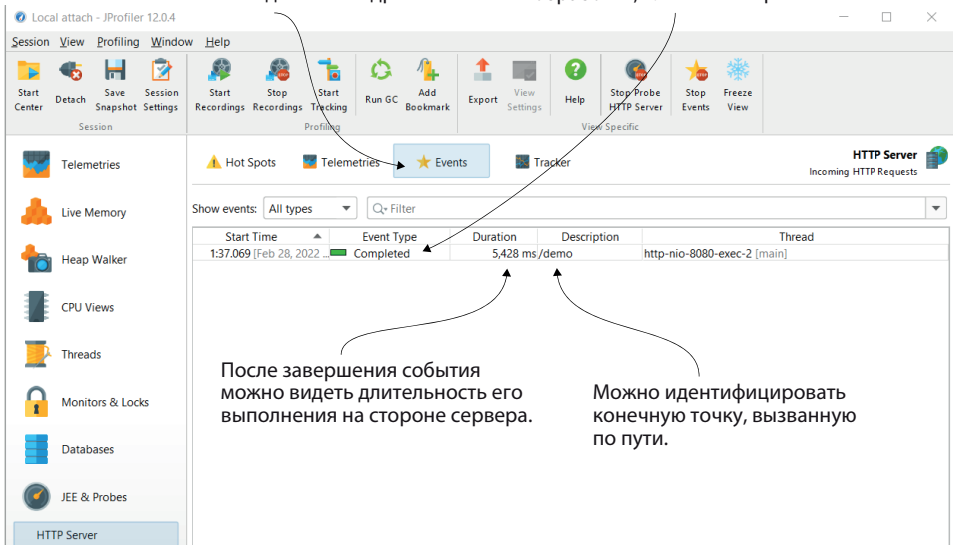


Рис. 12.3. После начала записи событий HTTP-сервера (HTTP-запросов, принимаемых приложением), на этой странице профилировщик показывает подробности обо всех принятых событиях. Можно легко узнать, завершилось ли событие и сколько времени потребовалось на его завершение

2.1.2. Использование проб HTTP-клиента для наблюдения HTTP-запросов, отправляемых приложением

Как и события HTTP-сервера (принимаемые приложением HTTP-запросы), можно профилировать события HTTP-клиента (отправляемые приложением HTTP-запросы). В этом подразделе рассматриваются HTTP-запросы, отправляемые приложением, для идентификации потенциальных проблем, причиной которых могут являться такие запросы. Для этого мы продолжим использование приложения из проекта `da-ch12-ex1`, с которым мы работали в подразделе 12.1.1. Это приложение отправляет запрос в конечную точку `httpbin.org`, когда вы вызываете конечную точку `/demo`, предоставляемую приложением. Запускаем приложение, вызываем конечную точку `/demo` и проверим, можно ли наблюдать HTTP-запросы, отправляемые приложением.

После запуска приложения начинаем запись событий HTTP-запросов в JProfiler (см. рис. 12.4) и вызываем конечную точку `/demo`:

```
curl http://localhost:8080/demo
```

Событие клиента выводится в таблице **Events**. JProfiler показывает подробности, в том числе продолжительность, вызванный URL, использованный метод HTTP и состояние, принятое как ответ.

Для каждого события, перехваченного JProfiler, HTTP-ответ в виде состояния кода указан в таблице.

Local attach - JProfiler 12.0.4

Session View Profiling Window Help

Start Center Detach Save Snapshot Session Settings Start Recordings Stop Recordings Start Tracking Run GC Add Bookmark Export View Settings Help Stop Probe HTTP Client Stop Events Freeze View

Telemetries Live Memory Heap Walker CPU Views Threads Monitors & Locks Databases JEE & Probes HTTP Server HTTP Client

Call Tree Hot Spots Telemetries **Events** Tracker

HTTP Client Outgoing HTTP Requests

Show events: All types Q+ Filter

Start Time	Event Type	Duration	Description	Thread	Status Code	Method	Throwable
57:59.807 [Fe...	Completed	5,526 ms	http://httpbin.org/delay/5	http-nio-80...	200	POST	

Total: 5,526 ms

Рис. 12.4. Можно записывать все HTTP-запросы, отправляемые приложением независимо от способа их передачи (технологии, используемой приложением), в JProfiler, который показывает подробности, такие как продолжительность, код состояния, метод HTTP и вызываемый URI, а также не было ли сгенерировано исключение, – т. е. всю полезную информацию для анализа конкретных сценариев, в которых участвуют HTTP-запросы, отправляемые приложением

Рассмотрим подробнее информацию, предоставленную профилировщиком (рис. 12.4). В первую очередь нас интересуют столбцы **Description** (Описание) и **Method** (Метод), поскольку они помогают определить, какую конечную точку вызывает приложение. После этого полную картину дополняют такие подробности, как продолжительность вызова, ответ в виде кода состояния и информация о том, было ли сгенерировано исключение.

Если вы обнаружили, что вызов выполняется в течение длительного времени (дольше, чем ожидалось), то необходимо узнать, почему. В первую очередь попытайтесь определить, является ли причиной проблемы обмен данными (через сеть) или что-то внутри приложения (например, десериализация ответа или его обработка).

В подразделе 12.1.3 вы увидите, что анализ событий низкого уровня в сокетах может подсказать, возникает ли проблема непосредственно при

обмене данными, или ее следует искать в том, что делает приложение. Если выясняется, что обмен данными не является причиной проблемы, то можно применить методики профилирования, описанные в главах 7–9, чтобы определить, что именно влияет на производительность при выполнении приложения.

Как и в случае HTTP-запросов, получаемых приложением (см. подраздел 12.1.1), важно учитывать счетчик событий (количество строк, содержащихся в таблице событий) HTTP-запросов, отправленных приложением. Не отправляет ли приложение слишком много запросов, заставляя другой сервис замедляться при ответе? В приложении, которое я реализовал некоторое время назад, обнаружилось, что оно отправляло частые запросы из-за некорректного механизма повторной передачи. Проблему было трудно обнаружить из внешнего компонента, так как запросы предназначались для извлечения некоторых данных, ничего не изменяли и не создавали некорректный вывод. В этом случае только дополнительные избыточные запросы воздействовали на производительность приложения.

12.1.3. Анализ событий низкого уровня в сокетах

В этом подразделе рассматривается анализ событий низкого уровня в сокетах, чтобы узнать, является ли причиной возникновения проблемы при обмене данными коммуникационный канал (например, сеть) или какая-то некорректная часть приложения. Для наблюдения таких событий низкого уровня можно использовать JProfiler. Для этого нужно перейти на вкладку **Sockets** (Сокеты) > **Events** (События).

Запустите приложение, активизируйте регистрацию событий в JProfiler, затем отправьте запрос в конечную точку /demo:

```
curl http://localhost:8080/demo
```

JProfiler перехватывает все события в сокетах и выводит их в таблице, как показано на рис. 12.5.

Сокет (socket) – это пункт связи с другим процессом, с которым приложение обменивается данными. При установлении соединения для обмена данными приложение выполняет следующие события в сокете (см. рис. 12.6):

- 1) открытие сокета для установления соединения (процедура рукопожатия с приложением, с которым необходимо общаться);
- 2) чтение из сокета (прием данных) или запись в сокет (отправка данных);
- 3) закрытие сокета.

Рассмотрим эти события более подробно, чтобы понять, что они могут сообщить нам о поведении приложения.

JProfiler перехватывает все события в сокетах и выводит подробности о них. Наиболее важные подробности: тип события, продолжительность и пропускная способность (скорость передачи).

Local attach - JProfiler 12.0.4

Session View Profiling Window Help

Start Center Detach Save Session Settings Start Recordings Stop Recordings Start Tracking Run GC Add Bookmark Export View Settings Help Stop Probe Sockets Stop Events Freeze View Control Object

Session Profiling

Heap Walker CPU Views Threads Monitors & Locks Databases JEE & Probes HTTP Server HTTP Client Web Services JNDI JMS RMI Class Loaders Exceptions Sockets

Time Line Sockets Call Tree Hot Spots Telemetries Events Tracker

I/O operations for sockets

Show events: All types Q Filter

Start Time	Event Type	Duration	Throughput	Socket ID	Description	Thread
63:17:20 [Feb 28, 2020]	Socket opened	0 µs	0 bytes 1	/0:0:0:0:0:0:0:0:8080	http-nio-8080-Acceptor [main]	
63:17:50 [Feb 28, 2020]	Socket opened	0 µs	0 bytes 2	/0:0:0:0:0:0:0:0:8080	http-nio-8080-Acceptor [main]	
63:17:50 [Feb 28, 2020]	Read	86 µs	0 bytes 1	/0:0:0:0:0:0:0:0:8080	http-nio-8080-exec-8 [main]	
63:17:50 [Feb 28, 2020]	Read	52 µs	1,994 bytes 2	/0:0:0:0:0:0:0:0:8080	http-nio-8080-exec-9 [main]	
63:17:50 [Feb 28, 2020]	Socket closed	0 µs	0 bytes 1	/0:0:0:0:0:0:0:0:8080	http-nio-8080-exec-8 [main]	
63:17:66 [Feb 28, 2020]	Socket opened	0 µs	0 bytes 3	httpbin.org/3.226.169.83:80	http-nio-8080-exec-9 [main]	
63:17:66 [Feb 28, 2020]	Write	154 µs	107 bytes 3	httpbin.org/3.226.169.83:80	http-nio-8080-exec-9 [main]	
63:17:66 [Feb 28, 2020]	Read	5,122 ms	535 bytes 3	httpbin.org/3.226.169.83:80	http-nio-8080-exec-9 [main]	
63:22:79 [Feb 28, 2020]	Write	644 µs	121 bytes 2	/0:0:0:0:0:0:0:0:8080	http-nio-8080-exec-9 [main]	
63:22:79 [Feb 28, 2020]	Read	55 µs	0 bytes 2	/0:0:0:0:0:0:0:0:8080	http-nio-8080-exec-9 [main]	
63:27:79 [Feb 28, 2020]	Socket closed	0 µs	0 bytes 3	httpbin.org/3.226.169.83:80	Keep-Alive-Timer [InnocuousThreadGroup]	
64:22:82 [Feb 28, 2020]	Socket closed	0 µs	0 bytes 2	/0:0:0:0:0:0:0:0:8080	http-nio-8080-exec-10 [main]	

Total: 5,123 ms 2,757 bytes

Stack trace:

Enable CPU recording to see stack traces

В столбце **Description** (Описание) показан ваш IP-адрес и порт. Эти подробности помогают определить, как было установлено соединение для обмена данными через этот socket.

Рис. 12.5. Любое сообщение, которым приложение обменивается на сетевом уровне, скрыто использует сокеты. Можно воспользоваться профилировщиком, например JProfiler, для наблюдения за всеми событиями низкого уровня в сокетах.

Для мониторинга таких событий используйте вкладку **Sockets > Events**.

Эти события помогают понять, возникают ли в приложении сетевые проблемы, или это просто некорректное управление обменом данными

Открытие сокета для установления соединения

Одной из характеристик, которая должна привлечь ваше внимание, является длительное время выполнения для события открытия сокета. Открытие сокета не должно занимать много времени. Если это происходит, то является признаком проблемы в канале обмена данными. Например, система или виртуальная машина, в которой работает приложение, возможно, неправильно сконфигурирована, или проблемы существуют в самой сетевой среде. Если событие открытия сокета отнимает много времени, то, как правило, это не проблема в коде приложения.

Запись или чтение данных через сокет

Чтение или запись данных через сокет – это и есть, собственно, процесс обмена данными. Два приложения установили соединение друг с другом и обмениваются данными. Если эта операция выполняется медленно, то, возможно, из-за большого объема передаваемых данных или из-за медленной или некорректной работы коммуникационного канала.



Рис. 12.6. Когда приложение начинает обмен данными, в первую очередь оно открывает сокет. Для обмена данными приложение может выполнять несколько событий обмена данными (события чтения и/или записи данных). Когда обмен данными завершается, приложение закрывает сокет

Объем данных, передаваемых через сокет, можно определить с помощью JProfiler (см. столбец **Throughput** (Пропускная способность) на рис. 12.5), поэтому есть возможность выяснить, является ли причиной замедления большой объем данных или что-то другое. В рассматриваемом здесь примере можно видеть, что приложение принимает весьма небольшой объем данных (всего лишь 535 байт), но вынуждено ждать более 5 с. В этом случае можно сделать вывод: проблема возникает не в нашем приложении, а в коммуникационном канале или в процессе, с которым общается приложение.

Приложение, используемое в рассматриваемом здесь примере, вызывает конечную точку на httpbin.org, которая создает 5-секундную задержку. Поэтому наш вывод, несомненно, правильный: другая коммуникационная конечная точка является причиной замедления.

Заккрытие сокета

Заккрытие сокета не является причиной замедления. Эта операция позволяет приложению освободить ресурсы, выделенные сокету. Поэ-

тому после завершения обмена данными приложение обязано закрыть сокет.

12.2. Важность интегрированного мониторинга журналов

В настоящее время многие системы применяют сервис-ориентированный подход и увеличивают количество приложений, предлагаемых в динамике по времени. Эти приложения взаимодействуют друг с другом и выполняют обмен, сохранение и обработку данных, обеспечивая бизнес-функции, необходимые пользователям. С увеличением количества и размера приложений системы становятся все более сложными для мониторинга. Уже сейчас обнаружение возникающих проблем связано с существенными затруднениями. Чтобы идентифицировать части системы, являющиеся источниками проблем, можно воспользоваться возможностями, предоставляемыми инструментальными средствами мониторинга журналов (см. рис. 12.7).



ОПРЕДЕЛЕНИЕ. Инструментальное средство мониторинга журналов (log-monitoring tool) – это программное обеспечение, которое можно интегрировать в приложение для наблюдения за исключениями, генерируемыми во всей системе в целом.

Это инструментальное средство наблюдает за выполнением всех приложений и собирает данные во всех случаях, когда приложение генерирует исключение во время выполнения. Затем оно выводит собранную информацию в удобном для чтения виде, чтобы помочь вам быстрее найти причину возникновения проблемы.

Мы будем использовать простой инструмент, который можно сконфигурировать в любой системе для сбора данных о событиях исключений и для их представления в удобной для чтения форме. Sentry (<https://sentry.io>) – это инструментальное средство мониторинга журналов, его я использовал во многих системах, с которыми работал, и оно доказало свою чрезвычайную полезность как в производственной рабочей среде, так и при разработке приложений. Существует бесплатная версия Sentry, которую можно использовать в целях обучения (для примеров в этой главе).

Создадим приложение, преднамеренно генерирующее исключение, и интегрируем в него Sentry. Это приложение размещено в проекте da-ch12-ex2.

В приведенном ниже фрагменте кода показана простая реализация такого приложения. Необходимо использовать Sentry для наблюдения за исключениями, генерируемыми этим приложением.

```
@RestController
public class DemoController {
    @GetMapping
```

```
public void throwException() {
    throw new RuntimeException("Oh No!"); ❷
}
}
```

❶ Определяет конечную точку, вызываемую с использованием HTTP GET.

❷ Генерация исключения при отправке запроса в конечную точку.

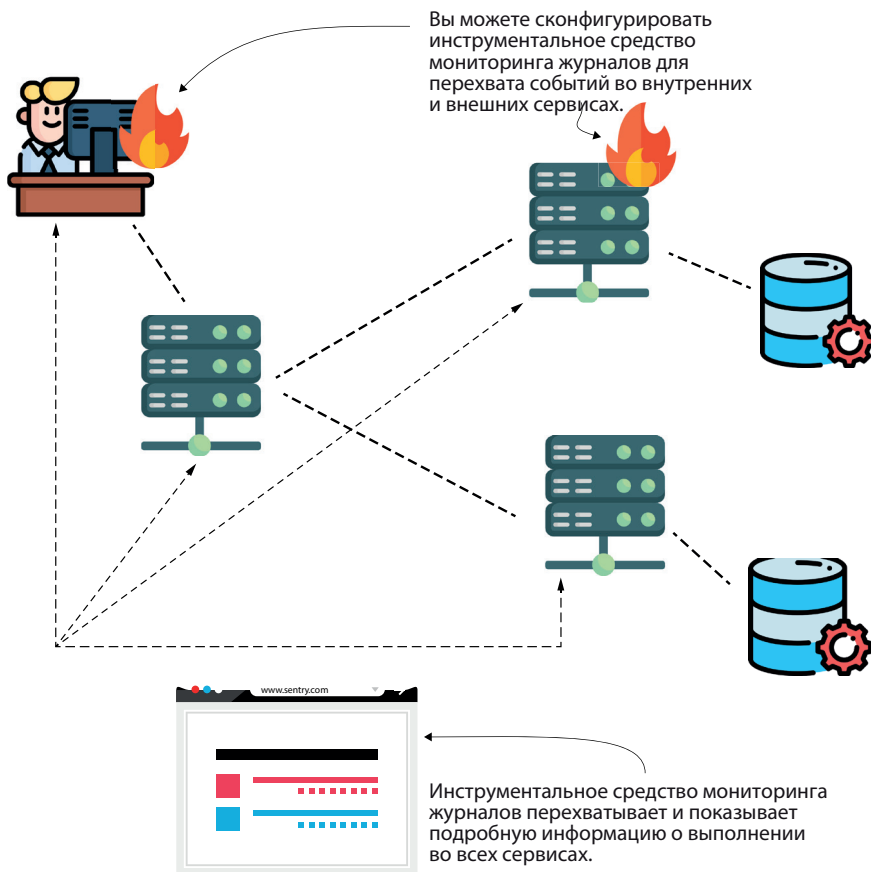


Рис. 12.7. Инструментальное средство мониторинга журналов помогает с легкостью собирать и визуализировать события во всей системе в целом. Эти подробности можно использовать для анализа проблем и специфического поведения приложения

Интеграция приложения с Sentry выполняется просто. Sentry предоставляет API, поддерживающий интеграцию с приложениями, разработанными на разнообразных платформах, с помощью всего лишь нескольких строк кода. В официальной документации приведены примеры и подробное пошаговое описание того, как выполнить интеграцию приложения с учетом использованных в нем технологий.

Простые шаги, которые нужно выполнить, описаны ниже:

- 1) создать аккаунт в Sentry;
- 2) добавить новый проект (представляющий приложение, в котором необходим мониторинг);
- 3) получить имя источника данных проекта (DSN), адрес которого предоставляет Sentry;
- 4) сконфигурировать DSN-адрес в созданном проекте.

После создания аккаунта (шаг 1) можно добавлять проекты (шаг 2). Для выполнения этих двух шагов нужно просто следовать инструкциям на сайте sentry.io. Этот процесс так же прост, как создание аккаунта на любом веб-сайте. Каждый добавляемый проект будет отображаться на панели управления, как показано на рис. 12.8.

Имя проекта. На панели управления может находиться несколько проектов. Все приложения должны быть сконфигурированы как независимые проекты в Sentry.

Гистограмма показывает активность событий в конкретном процессе. Этот небольшой столбик сообщает о генерации исключения в рассматриваемом здесь примере.

Имя команды. проекты можно связывать с командами. Если кто-то присоединяется к команде, то видит все соответствующие проекты на панели управления.

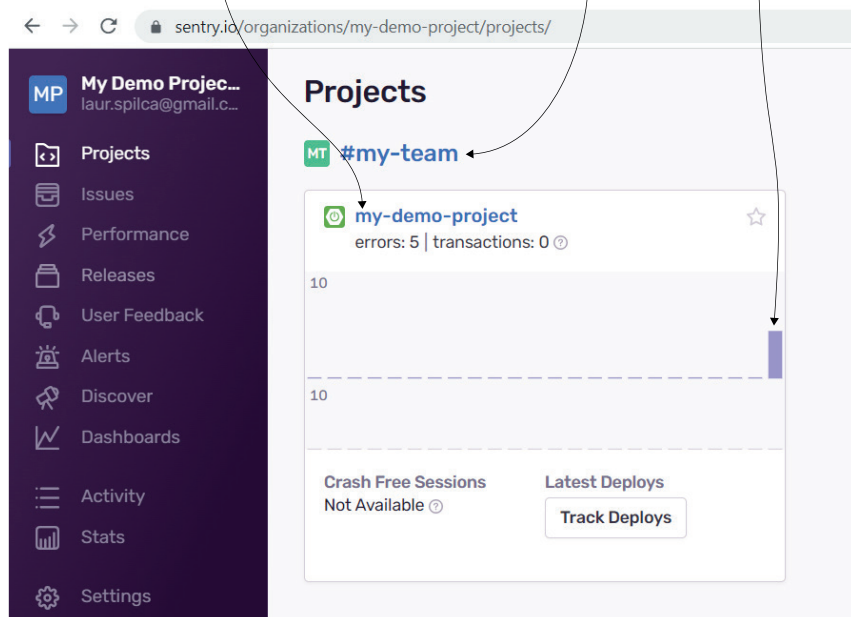


Рис. 12.8. Sentry независимо проводит мониторинг журналов каждого сервиса в вашей системе и выводит краткий обзор событий для каждого сервиса в основной панели управления. Сервисы (здесь они называются проектами) распределены по командам (teams), и Sentry можно сконфигурировать для отправки сообщений электронной почты о событиях членам каждой команды

Я создал проект `my-demo-project`. Один или несколько проектов можно добавить для команды. В данном случае Sentry создал команду `my-team` по умолчанию при добавлении первого проекта. При желании команду можно переименовать и добавить другие команды, если необходимо. Если имеется много приложений, то можно распределить их по командам. Каждый пользователь может быть членом одной или нескольких команд и получает возможность выполнять мониторинг событий в приложениях, связанных с его командой (командами). Команды в Sentry – это простой способ организации с определением того, кто за что отвечает, и предоставлением доступа для разработчиков для мониторинга конкретных сервисов.

Поскольку рассматриваемое здесь приложение пока еще не отправляло какие-либо события в Sentry, проект не показывает столбик на гистограмме (в отличие от рис. 12.8). Сначала необходимо указать приложению, куда отправлять события. Для этого требуется сконфигурировать DSN-адрес, предоставляемый Sentry, как показано на рис. 12.9. DSN-адрес вы найдете в настройках проекта в разделе **Client Keys** (Ключи клиента) (шаг 3).

В разделе **Settings > Client Keys** вы найдете DSN-адрес. Он необходим, чтобы сконфигурировать приложение для установления соединения с Sentry.

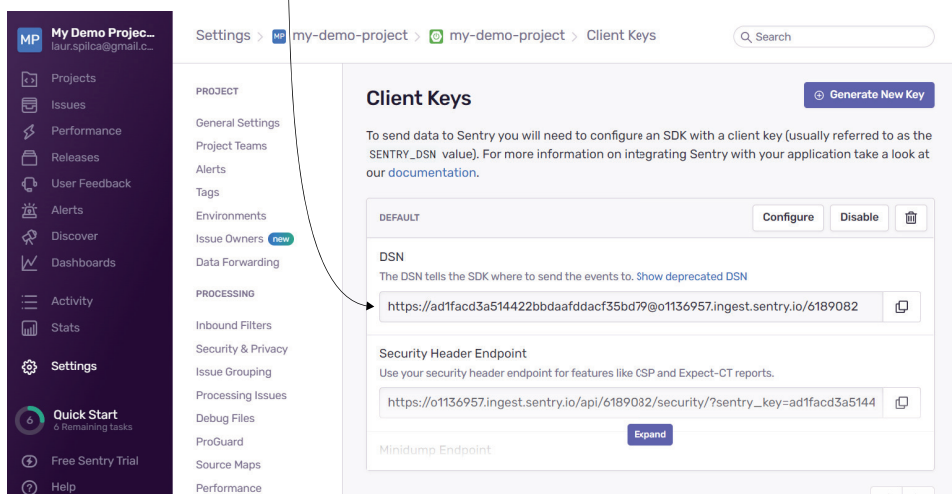


Рис. 12.9. В настройках проекта в разделе **Client Keys** вы найдете DSN-адрес, представленный в форме URL. Приложение использует этот URL для передачи событий в Sentry

В зависимости от типа приложения Sentry предоставляет различные способы настройки конфигурации (шаг 4). Подробное описание шагов настройки можно найти на официальной странице документации для каждой платформы: <https://docs.sentry.io/platforms/>.

Поскольку наш проект использует в качестве платформы Spring Boot, можно просто добавить DSN-значение в свойство `sentry.dsn` в файле `appli-`

cation.properties. Вы увидите эту конфигурацию в приведенном ниже фрагменте. Хотя это и не обязательно в Sentry, я рекомендую всегда определять имя среды, в которой работает приложение. Это позволит в дальнейшем отфильтровывать события, чтобы получать только те из них, в которых вы заинтересованы:

```
sentry.dsn=https://ad1facd3a514422bbdaafddacf...
sentry.environment=production
```

На рис. 12.10 показано, как получить подробности о событиях исключений в рассматриваемом приложении. В меню слева выберите пункт **Issues** (Проблемы) для доступа к панели, где можно просматривать все события, которые Sentry перехватил в интегрированном с ним приложении. Можно выбрать (отфильтровать) приложения, в которых необходимо наблюдать события, рабочую среду и интересующий вас интервал времени.

Можно щелкнуть по пункту **Issues**, чтобы увидеть все сгенерированные исключения и произошедшие события.

Фильтрация для получения событий по конкретным проектам и рабочим средам.

Выбор интервала времени, в течение которого необходимо наблюдать события.

Инструмент показывает краткий обзор подробностей для каждого события. Можно выбрать элемент в списке для получения о нем более подробной информации.

Количество произошедших событий одного типа.

Рис. 12.10. Sentry собирает все исключения, сгенерированные наблюдаемыми сервисами. В меню **Issues** можно просмотреть список этих событий (проблем). Можно выбрать (отфильтровать) их по времени возникновения события, рабочей среде и конкретным сервисам, в которых возникли эти события. Это упростит идентификацию проблемы

Панель **Issues** – основной начальный пункт для анализа. Если вы используете Sentry и необходимо проанализировать что-то произошедшее с сервисом в системе, в первую очередь проверьте события на панели **Issues**.

Применение Sentry для поиска событий исключений гораздо быстрее, чем поиск этих событий в журналах, описанный в главе 5.

Первое, что вы обнаружите на этой панели, – список с краткими подробностями для каждого исследуемого события. Наиболее важные подробности: тип исключения, соответствующее сообщение и количество обнаруженных исключений.

Еще одна весьма важная деталь, которую вы увидите для каждого события, – последняя метка времени обнаружения события и метка времени его первого появления. Эту информацию можно использовать, чтобы определить, является ли проблема многократно повторяющейся, если она возникала часто, или это отдельный случай. Если событие единичное, то можно узнать, что его причиной стала проблема, случайно возникшая в рабочей среде, но ошибки, создаваемые логикой приложения, возникают многократно и более часто. Как показано на рис. 12.10, все эти подробности отображены на основной панели **Issues**.

Если вы хотите получить более подробную информацию о конкретном событии, выберите требуемое событие на основной панели **Issues**. Sentry собирает следующие полезные подробности (см. рис. 12.11):

Тип исключения и соответствующее сообщение. Подробности о системе и приложении.

RuntimeException
0h No!

mechanism `HandlerExceptionResolver` handled `false`

`com.example.controllers.DemoController` in `throwException` at line 11
Called from: `jdk.internal.reflect.NativeMethodAccessorImpl` in `invoke0`

BREADCRUMBS Filter By Search breadcrumbs

TYPE	CATEGORY	DESCRIPTION	LEVEL	TIME
	http	GET /	Info	17:43:14
	exception	RuntimeException: 0h No!	Error	17:43:14

GET / localhost Formatted curl

Headers

Accept `text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9`

Accept-Encoding `gzip, deflate, br`

Accept-Language `en-US,en;q=0.9` Show More

Tags

browser	Chrome 98.0.4758	100%
browser.name	Chrome	100%
client_os	Windows 10	100%
client_os.name	Windows	100%
environment	production	100%
handled	no	100%
level	fatal	100%
mechanism	HandlerExceptionResolver	100%
runtime	Oracle Corporation 17.0.1	100%
runtime.name	Oracle Corporation	100%
server_name	host.docker.internal	100%
transaction	GET	100%
uri	http://localhost:8080/	100%

Подробности о событии, которое стало причиной исключения.

Рис. 12.11. Каждое из событий, зафиксированное Sentry, – трассировка стека, подробности о средах клиента и сервера и даже подробности о запросе (заголовки, HTTP-метод и т. п.), – предоставляет информацию, которая может помочь в определении главной причины возникновения проблемы

- трассировку стека исключений;
- подробности о рабочей среде, например операционная система, JVM времени выполнения и имя сервера;
- подробности о клиенте, если исключение было сгенерировано во время HTTP-запроса;
- информацию, передаваемую в запросе, если исключение произошло во время HTTP-запроса.

Использование Sentry для управления командой (группой разработки)

Несмотря на то что Sentry является инструментом, используемым главным образом для аудита, мониторинга и анализа проблем в приложениях, существует альтернативный вариант его применения, который я считаю достаточно полезным.

Как руководитель разработки, я одновременно являюсь руководителем группы и техническим руководителем. До пандемии COVID-19, когда мы все привыкли работать в офисе, в непосредственной близости друг от друга, мне было гораздо проще узнать о том, что один из членов группы боролся с некоторой проблемой, и наоборот: намного легче было запустить в меня бумажным шариком, чтобы привлечь внимание, когда требовалось мое участие. Но все изменилось после введения удаленной работы в режиме онлайн. Одним из факторов, создающих дополнительные задержки в работе группы, стали элементарные трудности в обмене информацией между членами группы.

Sentry можно сконфигурировать для отправки сообщений электронной почты о каждом возникающем событии, поэтому я создал конфигурацию для получения электронных писем даже о событиях, возникающих в локальных средах, чтобы видеть, с какими затруднениями сталкиваются члены группы. И поскольку я прекрасно знаю свою группу, мне сразу было понятно, когда кого-то из разработчиков остановила конкретная проблема. В некоторых случаях два и более членов группы имели дело с одинаковой проблемой, но, поскольку обмен информацией был нарушен, они нерационально тратили время на анализ.

При использовании Sentry я получил возможность действовать оперативно и помогать каждому, прежде чем он потратит слишком много времени, пытаясь проанализировать ошибку, а также возможность более эффективно планировать задачи группы. Также я мог остановить их работу, когда видел, что они действовали не по плану или одновременно занимались одним и тем же. Совсем неплохо, согласитесь?

Особенно полезной я считаю функцию Sentry, которая автоматически собирает подробную информацию о HTTP-запросе при генерации исключения в потоке, обслуживающем этот запрос. Эту информацию можно использовать для воспроизведения проблемы в среде разработки или для попытки определить, могут ли какие-либо данные, передаваемые в HTTP-запросе, являться причиной наступления события исключения. Хотя Sentry не показывает причину проблемы, он предоставляет больше фрагментов головоломки, помогая вам быстрее определить главную причину ее возникновения.



ПРИМЕЧАНИЕ. В настоящее время во многих случаях приложения обмениваются информацией друг с другом по протоколу HTTP, и высока вероятность того, что события исключений возникают как следствие этого. Sentry фиксирует подробности HTTP-запроса и связывает их с соответствующим событием.

12.3. Использование средств развертывания для анализа

Работая над многими проектами, со временем я осознал многое, в том числе тот факт, что рабочие среды, управляющие приложениями, разнообразны и постоянно развиваются. Я извлек важный урок: правильное понимание среды, в которой работает приложение, может оказаться невероятно полезным для выяснения, почему приложение ведет себя необычно. Рассмотрим один из самых современных способов развертывания сервис-ориентированных архитектур, а также почему это оказывается полезным при анализе проблем, которые могут возникать в приложениях: это архитектура service mesh.

Архитектура service mesh – это способ управления тем, как различные приложения в системе обмениваются информацией друг с другом, и это может оказаться чрезвычайно полезным с различных точек зрения, в том числе с точки зрения мониторинга и анализа приложений, когда в них возникают проблемы. Я предпочитаю использовать инструментальное средство service mesh Istio (<https://istio.io>). Для получения более подробной информации рекомендую книгу «Istio in Action» (Manning, 2022 г.), авторы Кристиан Е. Поста (Christian E. Posta) и Ринор Малоку (Rinor Maloku).

Я приведу здесь краткий обзор работы service mesh, а затем мы рассмотрим пару способов, полезных при анализе выполнения приложений:

- инъекцию критической ошибки (fault injection) – способ, которым можно принудительно привести приложение к критической ошибке при обмене данными, чтобы создать специальный сценарий, необходимый для анализа;

- зеркалирование (mirroring) – способ воспроизведения событий из приложения в производственной среде для их анализа в среде тестирования.

На рис. 12.12 показаны три сервиса, развернутые в service mesh. Каждому сервису соответствует приложение, перехватывающее данные, которыми приложение обменивается с другими приложениями.

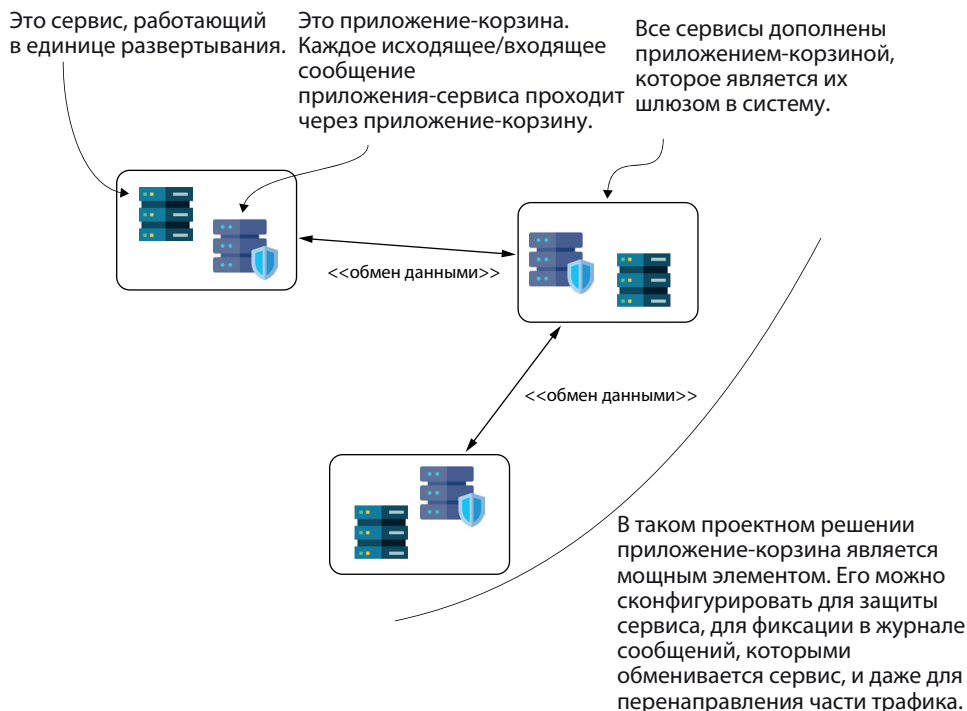


Рис. 12.12. В архитектуре развертывания service mesh обмен данными каждого приложения перехватывается специальным приложением-корзиной (side cart app) (это отдельное приложение). Поскольку приложение-корзина перехватывает передаваемые данные, его можно сконфигурировать для фиксации в журнале необходимых подробностей и даже изменить процесс обмена данными, чтобы принудительно переводить систему в сценарии, которые требуется анализировать

Поскольку приложение-корзина перехватывает коммуникационный поток между сервисом и связанными с ним другими приложениями, можно сконфигурировать приложение-корзину таким способом, что оно будет полностью прозрачным для сервиса. Мы рассмотрим этот способ более подробно в подразделах 12.3.1 и 12.3.2.

12.3.1. Использование инъекции критической ошибки для имитации трудновоспроизводимых проблем

Некоторые из самых сложных для анализа сценариев – это те, которые трудно воспроизвести в локальной среде или в среде, в которой вы имеете больше прав доступа для отладки или профилирования. Исходя из моего практического опыта, отмечу, что рабочая среда способна создавать некоторые из самых трудновоспроизводимых сценариев. События, подобные перечисленным ниже, могут вызвать ужасную головную боль и сделать анализ поведения приложения невероятно сложным:

- некоторое некорректно работающее устройство создает критические ошибки в сети;
- некоторое дополнительное программное обеспечение, работающее там, где установлено ваше приложение, делает всю рабочую среду некорректной или нестабильной.

Но вы должны запомнить кое-что важное относительно подобных проблем: ваше приложение обязано предполагать, что такие проблемы возникнут. Сеть никогда не бывает надежной на 100 %, поэтому полностью доверять рабочей среде невозможно. Если приложение аварийно завершается из-за сетевого сбоя, то оно недостаточно надежно. Не пытайтесь переложить ответственность за возникшую проблему на кого-то другого – устраняйте ее сами.

Необходимо проектировать надежные приложения и предполагать, что они знают, как действовать при возникновении экстремального события, которое не позволяет выполнять нормальный рабочий поток. Но проектирование системы таким способом – это совсем не простая работа. Как разработчик, вы обязаны предвидеть подобные ситуации, но, даже если вы приложили огромные усилия для покрытия всех базовых случаев, проблемы все равно могут возникать. Необходимо быть готовым к обнаружению источников возникновения проблем и реализации решений по их устранению.

Я многократно подчеркиваю важность этого положения на протяжении всей книги, но это необходимо повторить именно здесь: наилучший подход к анализу проблемы – обнаружение способа ее воспроизведения. Хотя какие-то проблемы, созданные рабочей средой, могут оказаться сложными для воспроизведения, некоторые сценарии можно с легкостью воспроизвести, если при развертывании используется service mesh.



ПРИМЕЧАНИЕ. Наилучший способ анализа сценария – воспроизведение поведения приложения или системы сначала в среде тестирования.

Самый простой и полезный способ – имитация сценария некорректного обмена данными. В сервис-ориентированной системе или системе с микросервисами вся система в целом основана на способе обмена данными между приложениями. Поэтому чрезвычайно важна возможность тестирования того, что происходит, когда к конкретному сервису в системе невозможен доступ. Необходимо имитировать некорректное поведение для тестирования или анализа.

Поскольку в архитектуре service mesh обмен данными между приложениями управляется приложением-корзиной, его можно сконфигурировать для нестандартных действий, чтобы имитировать некорректный обмен данными (см. рис. 12.13). Таким образом, можно проанализировать, как ведет себя система в подобном случае.

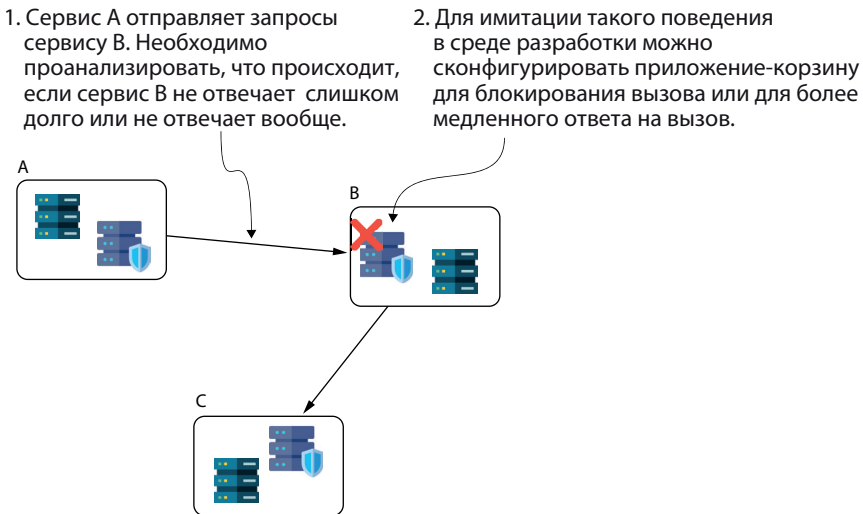


Рис. 12.13. Можно использовать приложение-корзину в service mesh для перевода системы в сценарий, которые необходимо проанализировать.

Например, нужно воспроизвести случай из реальной практики, в котором обмен данными между двумя сервисами часто прерывается.

Можно с легкостью сконфигурировать приложение-корзину для принудительного перевода выполнения в такой сценарий, что позволит вам выполнить анализ

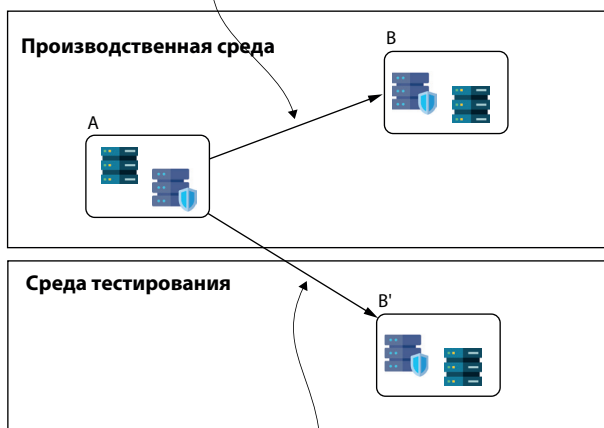


ПРИМЕЧАНИЕ. Инъекция критической ошибки означает преднамеренное нарушение работы системы в среде тестирования для воспроизведения конкретного поведения, которое трудно воспроизвести другими способами.

12.3.2. Использование зеркалирования для обеспечения тестирования и выявления ошибок

При использовании архитектуры service mesh можно применить методику воспроизведения проблемы в другой рабочей среде – зеркалирование (mirroring). Зеркалирование – это конфигурирование приложения-корзины для передачи копий запросов, отправляемых сервисом, в точную копию приложения, с которым происходит обмен данными. Такая точная копия может работать в другой среде, которую вы используете для тестирования (см. рис. 12.14). Это позволяет воспользоваться приложением, выполняющимся в среде тестирования для отладки или профилирования обмена данными между сервисами.

1. Вы анализируете проблему в производственной среде и предполагаете, что ее главную причину следует искать в процедуре обмена данными между сервисами А и В.



2. С помощью зеркалирования можно передавать копию каждого запроса в точную копию сервиса В в среде тестирования для получения полного доступа и начать отладку или журналирование без риска вмешательства в работу.

Рис. 12.14. Можно сконфигурировать приложение-корзину для зеркального отображения событий из промышленного приложения в сервисе, развернутом в среде разработки. Таким образом, можно анализировать проблему, которую трудно воспроизвести в среде разработки без вмешательства в работу производственной среды

Зеркалирование является действительно полезным инструментом анализа, но при этом следует помнить о том, что, даже если система использует service mesh для развертывания, вы не всегда будете иметь возможность применения этой методики. Во многих системах данные, используемые в производственной среде, являются секретными, и их невозможно просто скопировать в среду тестирования. Если система не позволяет копировать

данные из производственной среды в среду тестирования, то применение зеркалирования также будет запрещено.

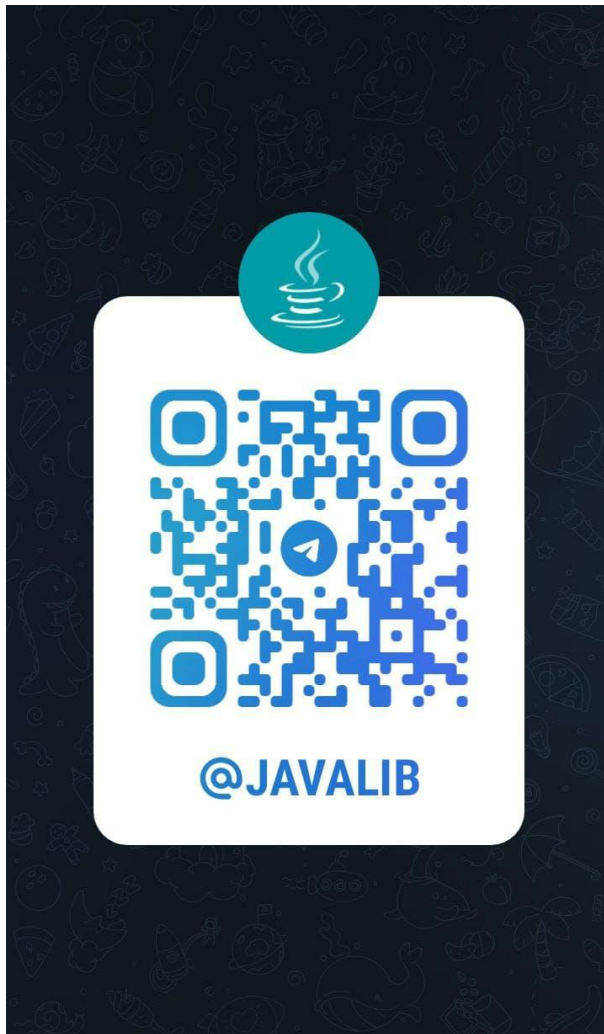
12.4. Резюме

- Современные системы часто состоят из множества сервисов, обменивающихся данными друг с другом. Некорректный обмен данными между сервисами может стать причиной возникновения таких проблем, как ухудшение производительности или даже некорректные выходные данные. Весьма важно знать, как анализировать процедуру обмена данными между сервисами, используя такие инструментальные средства, как профилирование или *service meshes*.
- Можно использовать JProfiler для перехвата HTTP-запросов, принимаемых приложением-сервером, и измерения продолжительности событий. Затем можно использовать эту информацию для выяснения, не вызывается ли конкретная конечная точка слишком часто, или требует слишком много времени для выполнения, создавая замедление работы экземпляра приложения.
- Можно использовать JProfiler для наблюдения за поведением приложения, работающего как HTTP-клиент. Можно перехватывать все запросы, отправляемые приложением, и получать подробную информацию, например продолжительность, код состояния HTTP-ответа и сгенерированные исключения. Эта информация может помочь в обнаружении возникших проблем при интеграции приложения с другими сервисами.
- JProfiler предоставляет превосходные инструменты для наблюдения за обменом данными на низком уровне в приложении, обеспечивая прямой анализ событий сокетов, что позволяет выделить проблему и определить, связано ли ее возникновение с коммуникационным каналом или с некоторой частью приложения.
- В крупных сервис-ориентированных системах использование инструмента мониторинга журналов является превосходным способом выявления проблем и ускоренного получения полной картины происходящего для обнаружения главной причины возникновения проблемы. Инструмент мониторинга журналов – это программное обеспечение, которое фиксирует события исключений в каждом приложении в системе и показывает информацию, необходимую для понимания проблемы и источников ее возникновения. Sentry – отличный инструмент, который можно использовать для мониторинга журналов системы.
- В некоторых случаях можно воспользоваться преимуществами инструментальных средств, используемых для развертывания приложений. Например, если сервис развертывается на основе архитектуры

service mesh, то можно использовать функциональные возможности этой архитектуры для воспроизведения сценариев, которые необходимо проанализировать. Можно сконфигурировать:

- инъекцию критической ошибки (fault injection) для имитации сервиса, который работает некорректно, и анализа его воздействия на другие сервисы;
- зеркалирование (mirroring) для передачи копий всех запросов, отправляемых приложением, в точную копию сервиса-приемника. Такая точная копия устанавливается в среде тестирования, где можно анализировать сценарий, используя методики отладки и профилирования без воздействия на производственную систему.

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javallib>



Приложение А

Необходимые инструментальные средства

В этом приложении вы найдете ссылки на инструкции по установке всех рекомендуемых инструментальных средств для работы с примерами, рассматриваемыми в книге.

Чтобы открыть и выполнить проекты, приложенные к этой книге, необходимо установить IDE. Я использовал IntelliJ IDEA: <https://www.jetbrains.com/idea/download/>. Также можно использовать Eclipse IDE: <https://www.eclipse.org/downloads/>. Или вы можете предпочесть Apache Netbeans: <https://netbeans.apache.org/download/index.html>.

Для выполнения проектов Java, представленных в этой книге, потребуются установить JDK версии 17 или более поздней. Рекомендую использовать дистрибутив Open JDK: <https://jdk.java.net/17/>.

Для изучения методик профилирования и чтения дампов кучи и потоков мы используем VisualVM: <https://visualvm.github.io/download.html>. Для некоторых рассматриваемых методик применения VisualVM недостаточно. Для них мы пользуемся JProfiler: <https://www.ej-technologies.com/products/jprofiler/overview.html>.

Инструмент, который поможет анализировать дампы потоков, — fastThread, мы используем его в главе 9: <https://fastthread.io/>.

На протяжении всей книги для вызова конечных точек применяется Postman, чтобы продемонстрировать методики анализа: <https://www.postman.com/downloads/>.

В главе 12 рассматривается мониторинг событий, зафиксированных в журналах, с помощью Sentry: <https://sentry.io>.

Приложение В

Открытие проекта

В этом приложении описаны пошаговые действия для открытия и запуска существующего проекта. Проекты, прилагаемые к этой книге, являются приложениями, использующими версию Java 17. Мы используем их для демонстрации применения нескольких технологий и инструментальных средств.

В первую очередь необходимо наличие установленной IDE, например IntelliJ IDEA, Eclipse или Apache Netbeans. Для примеров я использовал IntelliJ IDEA: <https://www.jetbrains.com/idea/download/>.

Для выполнения проектов, рассматриваемых в книге, необходимо установить JDK версии 17 или более поздней. Можно использовать любой дистрибутив Java. Я использую OpenJDK: <https://jdk.java.net/17/>.

На рис. В.1 показано, как открыть существующий проект в IntelliJ IDEA. Чтобы открыть нужный проект, в меню выберите пункт **File** (Файл) > **Open** (Открыть).

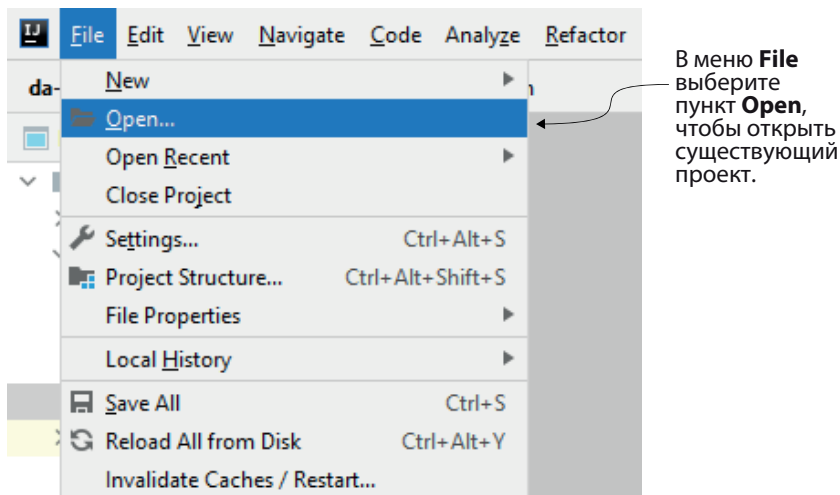


Рис. В.1. Чтобы открыть нужный проект в IntelliJ IDEA, выберите пункт **Open** в меню **File**

Щелкните по пункту **File** > **Open**, и появится всплывающее окно, показанное на рис. В.2. Выберите проект, который нужно открыть.

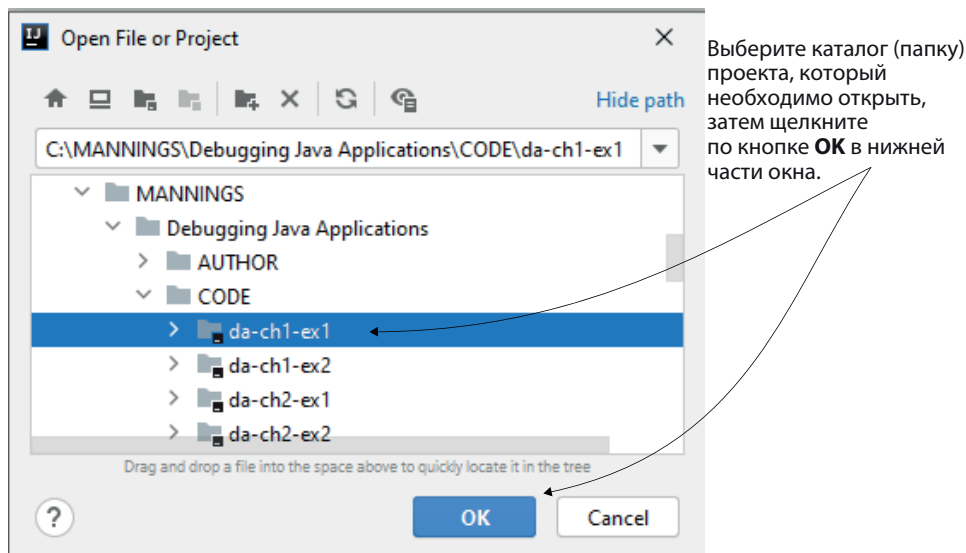


Рис. В.2. После выбора пункта **Open** в меню **File** появляется всплывающее окно. Выберите в нем проект, который необходимо открыть в файловой системе, и щелкните по кнопке **OK**

Для запуска приложения щелкните правой кнопкой мыши по классу, содержащему метод `main()`. В проектах, рассматриваемых в этой книге, метод `main()` определен в классе с именем `Main`. Щелкните правой кнопкой мыши по этому классу, как показано на рис. В.3, и в контекстном меню выберите пункт **Run** (Выполнить).

Если необходимо выполнять приложение с использованием отладчика, то после щелчка правой кнопкой мыши по классу `Main` выберите пункт **Debug** (Отладка).

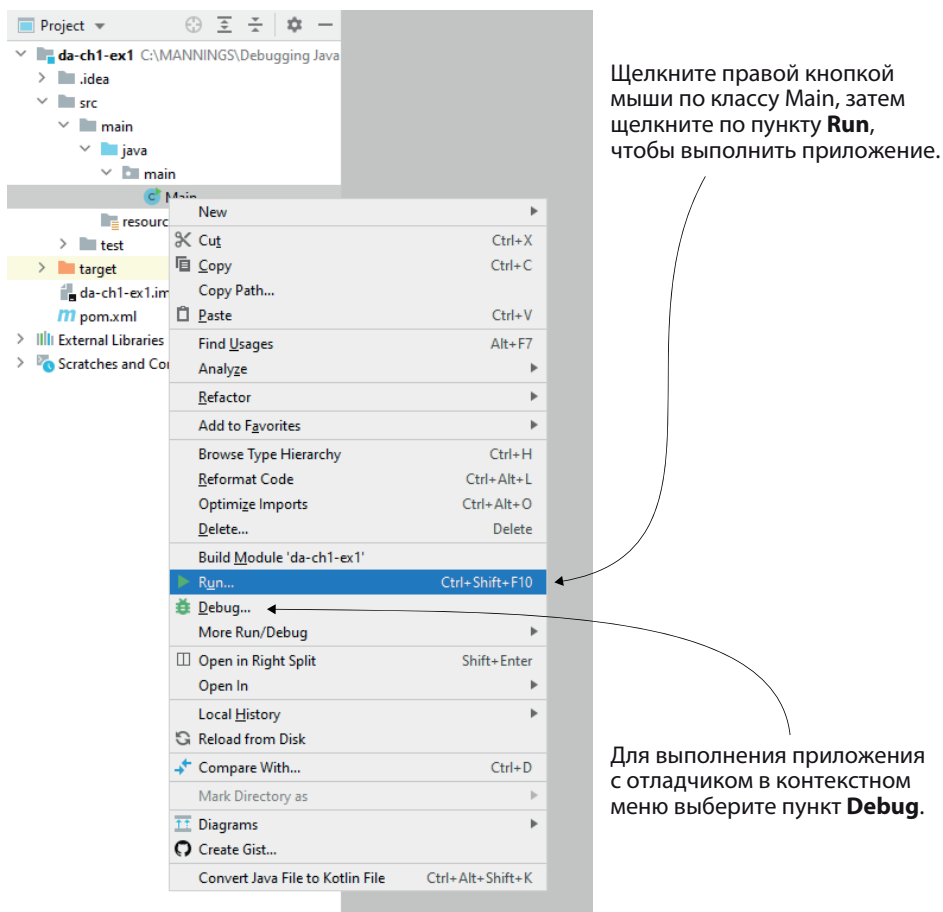


Рис. В.3. После открытия приложения его можно выполнить. Для запуска приложения щелкните правой кнопкой мыши по классу Main и в контекстном меню выберите пункт **Run**. Если необходимо выполнять приложение с отладчиком, то выберите пункт **Debug**

Приложение С

Литература, рекомендуемая для дополнительного чтения

Это приложение содержит краткие описания рекомендуемых для чтения книг, связанных с темой этой книги. Перечисленные ниже книги могут оказаться полезными и интересными для вас.

- *Фельенн Херманс* (Felicienne Hermans), «The Programmer’s Brain» (Manning, 2021 г.) – описано, как работает мозг программиста-разработчика, когда он анализирует код. Чтение кода является частью процесса понимания программного обеспечения, и иногда мы делаем это до того, как применить методики анализа. Лучшее понимание этих аспектов также помогает при анализе кода.
- *Сэм Ньюман* (Sam Newman), «Monolith to Microservices» (O’Reilly Media, 2019 г.) – я рекомендовал эту книгу в главе 12 для изучения микросервисов как стиля архитектуры. Главной темой книги является различие между монолитным подходом и микросервисами, в ней также описано, где и как использовать каждый из этих стилей архитектуры.
- *Сэм Ньюман* (Sam Newman), «Building Microservices: Designing Fine-Grained Systems, Second Edition» (O’Reilly Media, 2021 г.) – еще одна книга Сэма Ньюмана, главной темой которой является проектирование систем, состоящих из мелкоструктурных сервисов. Автор анализирует достоинства и недостатки представляемых методик, используя простые и подробные примеры.
- *Крис Ричардсон* (Chris Richardson), «Microservices Patterns» (Manning, 2018 г.) – одна из книг, которые я считаю обязательными для чтения всеми, кто работает с архитектурами микросервисов. Автор подробно на понятных примерах описывает самые важные методики, используемые в крупномасштабных микросервисах и сервис-ориентированных системах.
- *Кристиан Клаузен* (Christian Clausen), «Five Lines of Code» (Manning, 2021 г.) – обучение практическим методикам написания простого и ясного кода. Многие современные приложения не структурированы

и чрезвычайно сложны для понимания. Я проектировал многие листинги кода, которые вы найдете в примерах книги, так, чтобы они выглядели реалистично, поэтому не всегда следовал принципам ясного кодирования. Но, после того как вы поймете, как работает часть непонятного кода, необходимо провести рефакторинг, чтобы сделать его более простым для понимания. Разработчики называют этот принцип правилом бойскаута (Boy Scout rule). Во многих случаях за отладкой следует рефакторинг, чтобы сделать код более понятным в будущем.

- *Том Лонг* (Tom Long), «Good Code, Bad Code» (Manning, 2021 г.) – превосходная книга, обучающая принципам написания высококачественного кода. Я рекомендую прочесть ее еще и для повышения практических навыков рефакторинга и создания приложений, простых для понимания.
- *Томаш Лелек* (Tomasz Lelek) и *Йон Скит* (Jon Skeet), «Software Mistakes and Tradeoffs» (Manning, 2022 г.) – в этой книге рассматривается на превосходных примерах, как принимать трудные решения, находить компромиссы и оптимизировать проектные решения при разработке программного обеспечения.
- *Мартин Фаулер* (Martin Fowler) и *Кент Бек* (Kent Beck), «Refactoring: Improving the Design of Existing Code» (Addison-Wesley Professional, 2018 г.) – еще одна обязательная для чтения книга для каждого разработчика программного обеспечения, желающего улучшить свои практические навыки проектирования и создания простых и ясных приложений, которые удобно сопровождать.

Приложение D

Понимание потоков Java

В этом приложении мы рассмотрим основы применения потоков в приложениях Java. Поток (thread) – это независимый набор последовательных инструкций, выполняемых приложением. Операции в конкретном потоке выполняются одновременно (параллельно) с операциями в других потоках. В любом современном Java-приложении предполагается наличие нескольких потоков, поэтому практически невозможно обойтись без сценариев анализа, в которых необходимо более глубоко понимать, почему конкретные потоки не делают то, что должны, или просто не взаимодействуют с другими потоками. Именно поэтому на протяжении всей этой книги вы будете обнаруживать потоки во многих ее разделах (особенно в главах 7–9, но также и в первой половине книги, где рассматривается отладка). Для правильного понимания этого материала необходимо знать некоторые основы практического применения потоков. Данное приложение позволит вам изучить эти элементы, весьма важные для понимания других тем книги.

Начнем с раздела D.1, в котором я напомним общие принципы применения потоков и объясню, почему они используются в приложениях. В разделе D.2 содержится более подробная информация о том, как выполняются потоки, с описанием их жизненного цикла. Знание состояний и возможных переходов между ними в жизненном цикле потоков необходимо при анализе любой проблемы, связанной с потоками. В разделе D.3 рассматривается синхронизация потоков как способ управления их выполнением. Некорректные реализации синхронизации создают большинство проблем, которые вы должны анализировать и устранять. В разделе D.4 описаны наиболее часто возникающие проблемы, связанные с потоками.

Потоки – сложная тема, поэтому я сосредоточил внимание только на тех аспектах, которые необходимо знать для полного понимания методик, представленных в этой книге. Не могу обещать, что сделаю вас экспертом по этой теме после изучения всего лишь нескольких страниц, поэтому рекомендую воспользоваться несколькими ресурсами, перечисленными в конце текущего приложения.

D.1. Что такое поток

В этом разделе мы рассмотрим, что представляют из себя потоки и как использование нескольких потоков помогает приложению. Поток (thread) –

это независимая последовательность операций в выполняющемся процессе. Любой процесс может содержать несколько потоков, работающих одновременно, предположительно, позволяя приложению решать несколько задач в параллельном режиме. Потоки являются чрезвычайно важным компонентом реализации параллельного выполнения в любом языке.

Я предпочитаю наглядно изображать многопоточное приложение как группу последовательных шкал времени, как показано на рис. D.1. Обратите внимание: приложение начинает работу с одним потоком (основной (main) поток). Этот поток запускает другие потоки, которые активизируют прочие и т. д. Следует помнить, что каждый поток независим от других. Например, основной поток может завершить свое выполнение намного раньше, чем завершит работу все приложение. Процесс завершается, когда останавливаются все потоки.

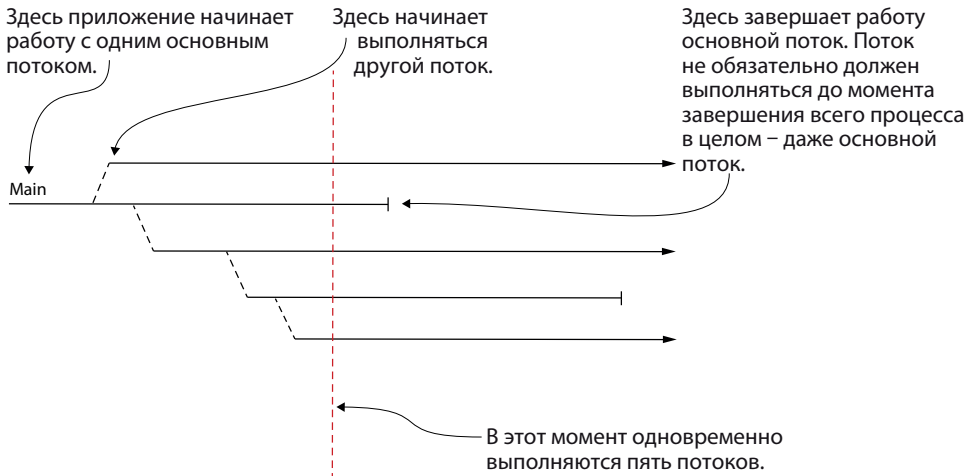
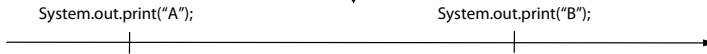


Рис. D.1. Визуальное представление многопоточного приложения как группы последовательных шкал времени. Здесь каждая стрелка представляет шкалу времени отдельного потока. Приложение начинает работу с основным потоком, который может запускать другие потоки. Некоторые потоки выполняются до завершения всего процесса в целом, другие останавливаются раньше. В любой конкретный момент времени в приложении может существовать один или несколько потоков, работающих в параллельном режиме

Инструкции в конкретном потоке всегда расположены в определенном порядке. Вы знаете, что А будет выполнена раньше В, если инструкция А находится перед инструкцией В в одном и том же потоке. Но поскольку два потока независимы друг от друга, невозможно сказать то же самое о двух инструкциях А и В, находящихся в разных потоках. В этом случае либо А может выполняться раньше В, либо наоборот (см. рис. D.2). Иногда мы говорим, что один вариант более вероятен, чем другой, но не можем знать, как один общий поток процесса (flow) будет выполняться согласованно.

Две инструкции в одном потоке всегда будут выполняться в том порядке, в котором они записаны. Здесь нам известно, что приложение всегда выводит A, затем B.



Поскольку два потока независимы друг от друга, мы не можем сказать, в каком порядке будут выполнены две инструкции в двух различных потоках. В этом случае приложение может вывести AB или BA.

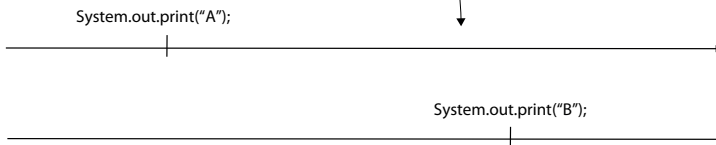


Рис. D.2. Если две инструкции находятся в одном потоке, то мы всегда знаем точный порядок их выполнения. Но поскольку два потока независимы друг от друга, если две инструкции расположены в разных потоках, то нам неизвестен порядок их выполнения. Чаще всего можно только сказать, что один сценарий более вероятен, чем другой

Во многих случаях вы увидите выполнение потока, визуальное представление инструментальными средствами как последовательные шкалы времени. На рис. D.3 показан способ, которым VisualVM (инструмент, которым мы пользовались в книге) представляет выполнение потоков в виде последовательных шкал времени.

D.2. Жизненный цикл потока

После визуализации выполнения потоков другим весьма важным аспектом понимания их работы является знание жизненного цикла потока. Во время выполнения поток проходит несколько состояний (см. рис. D.4). При использовании профилировщика (как описано в главах 6–9) или дампа потоков (см. главу 10) мы часто упоминаем состояние потока, которое важно при попытке определения его выполнения. Знание того, как поток может переходить из одного состояния в другое и как поток ведет себя в каждом состоянии, чрезвычайно важно для наблюдения и анализа поведения приложения.

Выполнение потоков показано как последовательные шкалы времени в VisualVM.

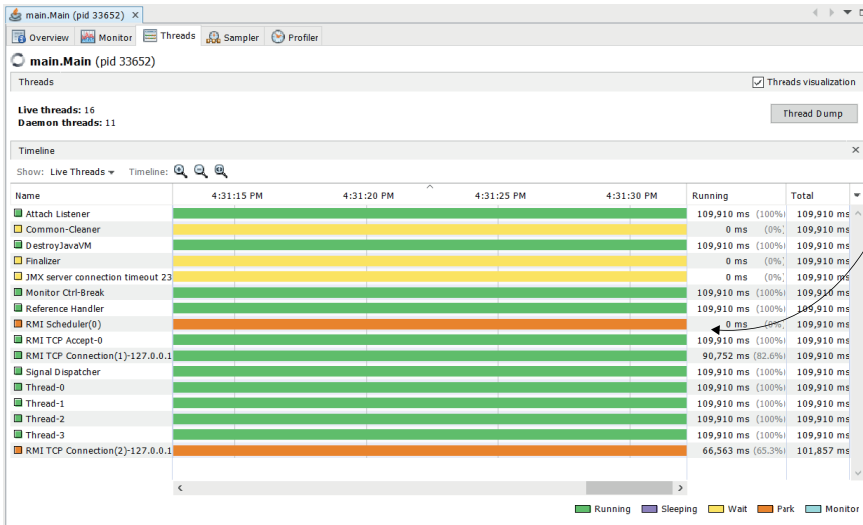


Рис. D.3. VisualVM показывает выполнение потоков как последовательные шкалы времени. Такое визуальное представление позволяет быстрее понять выполнение приложения и помогает проанализировать вероятные проблемы

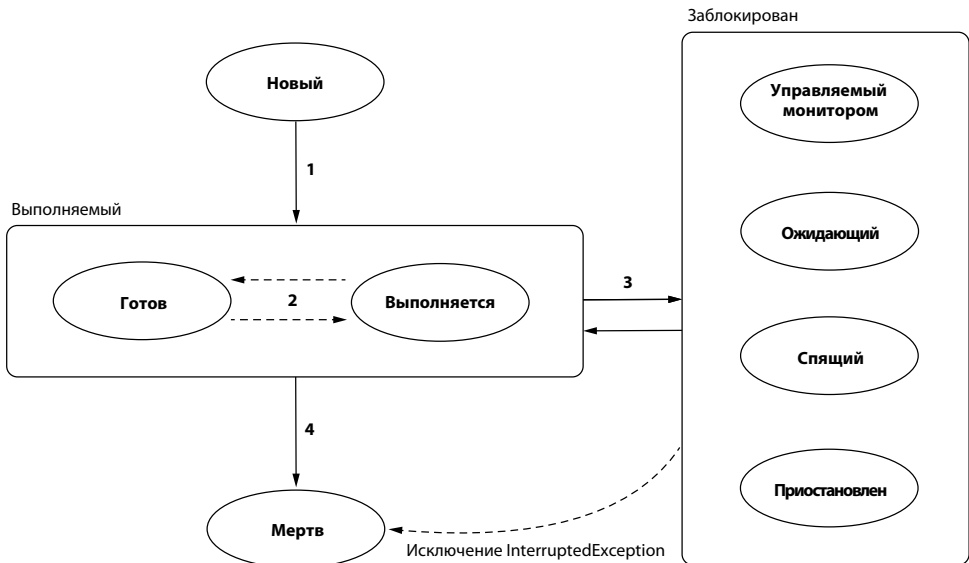


Рис. D.4. Жизненный цикл потока. На протяжении жизни поток проходит несколько состояний. Сначала поток является новым, и JVM не может выполнять инструкции, которые определены в потоке. После запуска потока он становится выполняемым, и JVM начинает управлять им. На протяжении жизни поток может быть временно заблокирован, а в конце жизненного цикла он переходит в состояние «мертв», из которого его невозможно перезапустить

На рис. D.4 визуальны представлены состояния потока, а также его возможные переходы из одного состояния в другое. Можно определить следующие основные состояния потока Java:

- *new* (новый) – поток находится в этом состоянии сразу после создания его экземпляра (перед запуском). В этом состоянии поток представляет собой простой объект Java. Приложение пока еще не может выполнять инструкции, определенные в нем;
- *runnable* (выполняемый) – поток находится в этом состоянии после вызова его метода `start()`. В этом состоянии JVM может выполнять инструкции, определенные в потоке. Когда поток находится в этом состоянии, JVM будет постоянно перемещать поток между двумя подсостояниями:
 - *ready* (готов) – поток не выполняется, но JVM в любой момент может выполнить его;
 - *running* (выполняется) – поток выполняется. В этом подсостоянии ЦП выполняет инструкции, определенные в потоке;
- *blocked* (заблокирован) – поток был запущен, но временно выведен из состояния выполнения, поэтому JVM не может выполнять его инструкции. Это состояние помогает разработчику управлять выполнением потока, позволяя временно «скрывать» его от JVM, чтобы исключить возможность его выполнения. Во время блокировки поток может находиться в одном из следующих подсостояний:
 - *monitored* (управляемый монитором) – поток приостановлен монитором синхронизированного блока (объектом, управляющим доступом к синхронизированному блоку) и ждет освобождения, чтобы выполнить этот блок;
 - *waiting* (ожидающий) – во время выполнения был вызван метод монитора `wait()`, что привело к приостановке текущего потока. Поток остается заблокированным до тех пор, пока не будет вызван метод `notify()` или `notifyAll()`, чтобы позволить JVM освободить поток для выполнения;
 - *sleeping* (спящий) – был вызван метод `sleep()` из класса `Thread`, который приостановил выполнение текущего потока на определенное время. Время определяется как параметр для метода `sleep()`. По истечении заданного времени поток вновь становится выполняемым;
 - *parked* (приостановлен) – почти то же самое, что ожидающий (*waiting*). Поток будет показан как приостановленный после того, как кто-то вызвал метод `park()`, блокирующий текущий поток до вызова метода `unpark()`;

- *dead* (мертв) – поток умирает или завершается после выполнения набора своих инструкций, или после останова по ошибке или исключению, или если его выполнения было прервано другим потоком. После смерти поток невозможно запустить снова.

На рис. D.4 также показаны возможные переходы между состояниями потока:

- поток переходит из состояния «новый» в состояние «выполняемый», когда кто-то вызывает его метод `start()`;
- находясь в состоянии «выполняемый», поток постоянно осуществляет переходы между подсостояниями «готов» и «выполняется». JVM решает, какой поток выполняется и когда;
- иногда поток блокируется. Он может быть переведен в состояние «заблокирован» несколькими способами:
 - вызов метода `sleep()` из класса `Thread` переводит поток в состояние временной блокировки;
 - кто-то вызывает метод `join()`, заставляющий текущий поток ждать другого;
 - кто-то вызывает метод `wait()` монитора, приостанавливая выполнение текущего потока до тех пор, пока не будет вызван метод `notify()` или `notifyAll()`;
 - монитор синхронизированного блока приостанавливает выполнение потока до тех пор, пока другой активный поток не завершит выполнение этого синхронизированного блока;
- поток может перейти в состояние «мертв» (завершен), когда закончит свое выполнение или если другой поток прервет его. JVM рассматривает переход из заблокированного состояния в состояние «мертв» как неприемлемый. Если заблокированный поток прерывается другим потоком, то такой переход осуществляется через исключение `InterruptedException`.

D.3. Синхронизация потоков

В этом разделе рассматриваются методики синхронизации потоков, которые разработчики используют для управления потоками в многопоточной архитектуре. Некорректная синхронизация также является главной причиной возникновения многих проблем, которые вы должны анализировать и устранять. Здесь приведен краткий обзор наиболее часто применяемых способов синхронизации потоков.

D.3.1. Синхронизированные блоки

Простейшим способом синхронизации потоков и, как правило, самой первой концепцией синхронизации потоков, осваиваемой разработчика-

ми, является использование синхронизированного блока кода. Его предназначение – разрешить только одному потоку проходить через синхронизированный код, т. е. запрещение параллельного выполнения конкретного фрагмента кода. Для этого существует два возможных варианта:

- синхронизация блока – применение модификатора `synchronized` к конкретному блоку кода;
- синхронизация метода – применение модификатора `synchronized` к методу.

Ниже показан пример синхронизированного блока кода:

```
synchronized (a) {           ❶
    // Здесь что-то выполняется.  ❷
}
```

- ❶ Объект в круглых скобках – это монитор синхронизированного блока.
- ❷ Синхронизированный блок инструкций определяется между фигурными скобками.

В следующем фрагменте кода показана синхронизация метода:

```
synchronized void m() {      ❶
    // Здесь что-то выполняется.  ❷
}
```

- ❶ Модификатор `synchronized` применяется к методу.
- ❷ Весь блок кода в методе, определенный между фигурными скобками, является синхронизированным.

Оба способа использования ключевого слова `synchronized` работают одинаково, даже если они выглядят немного по-разному. В каждом синхронизированном блоке существует два важных компонента:

- монитор – объект, управляющий выполнением синхронизированных инструкций;
- блок инструкций – реальные инструкции, которые становятся синхронизированными.

При синхронизации метода кажется, что монитор отсутствует, но для этого синтаксиса в действительности существование монитора подразумевается. Для нестатического метода экземпляр `this` будет использоваться в качестве монитора, тогда как для статического метода синхронизированный блок будет использовать экземпляр типа класса.

Монитор (который не может быть нулевым) – это объект, придающий смысл синхронизированному блоку. Этот объект решает, может ли поток войти и выполнить синхронизированные инструкции. С технической точки зрения правило простое: после входа в синхронизированный блок по-

ток получает в свое распоряжение блокировку от монитора. Никакой другой поток не будет допущен в синхронизированный блок до тех пор, пока поток, владеющий блокировкой, не освободит ее. Проще говоря, будем считать, что поток освобождает (снимает) блокировку только при выходе из синхронизированного блока. На рис. D.5 показан наглядный пример. Предположим, что два синхронизированных блока находятся в разных частях приложения, но, поскольку оба используют один и тот же монитор M1 (единственный экземпляр объекта), поток может выполняться только в одном из блоков в любое время. Ни одна из инструкций A, B или C не будет вызвана параллельно (по крайней мере, не из представленных здесь синхронизированных блоков).

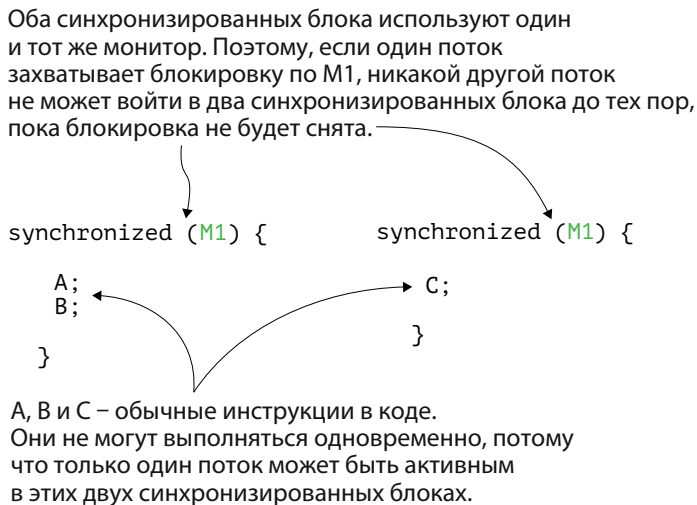


Рис. D.5. Пример использования синхронизированных блоков.

Несколько синхронизированных блоков в приложении могут использовать один и тот же экземпляр объекта как монитор. При этом все потоки координируются так, чтобы только один активный поток выполнял все такие блоки. Здесь если один поток входит в синхронизированный блок, определяющий инструкции A и B, то никакой другой поток не может войти в тот же блок или в другой, определяющий инструкцию C

Но в приложении может быть определено несколько синхронизированных блоков. Монитор связывает несколько синхронизированных блоков, но если два синхронизированных блока используют два различных монитора (см. рис. D.6), то эти блоки не связаны между собой (не синхронизированы друг с другом). На рис. D.6 первый и второй блоки синхронизированы друг с другом, поскольку используют один и тот же монитор. Но оба этих блока не синхронизированы с третьим. В результате инструкция D, определенная в третьем синхронизированном блоке, может выполняться одновременно (параллельно) с любыми инструкциями в первых двух блоках.

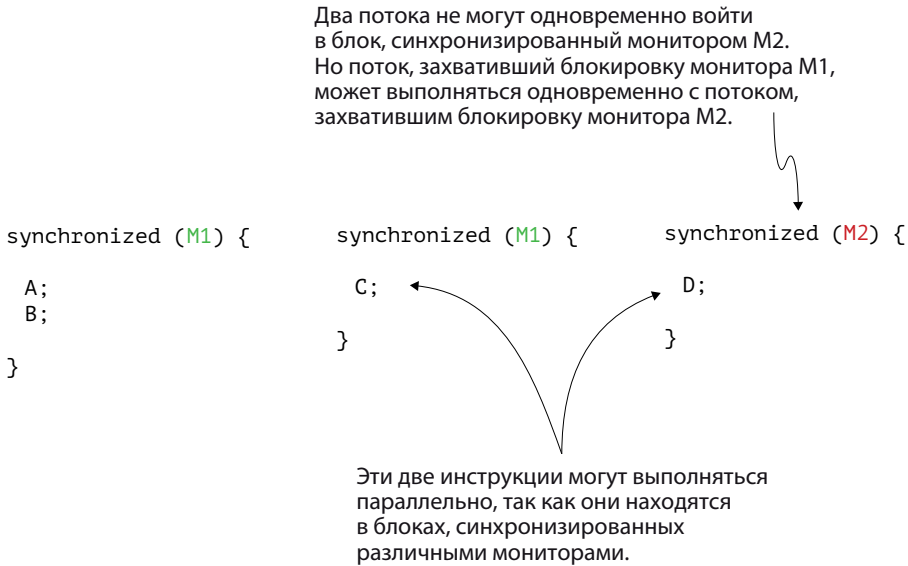


Рис. D.6. Если два синхронизированных блока не используют один и тот же экземпляр объекта как монитор, то они не синхронизированы друг с другом. В данном случае второй и третий синхронизированные блоки используют различные мониторы. Это означает, что инструкции в этих блоках могут выполняться одновременно

При анализе проблем с использованием таких инструментов, как профилировщик или дамп потоков, необходимо понимать способ, которым блокируются потоки. Эта информация помогает прояснить ситуацию, узнать, что происходит и по каким причинам конкретный поток не выполняется. На рис. D.7 можно видеть, как VisualVM (профилировщик, который мы использовали в главах 7–9) показывает, что монитор синхронизированного блока заблокировал поток.

D.3.2. Использование `wait()`, `notify()` и `notifyAll()`

Другой возможный способ блокировки потока – обращение к нему с просьбой подождать в течение неопределенного интервала времени. Используя метод `wait()` монитора синхронизированного блока, можно сообщить потоку о необходимости ждать без ограничения по времени. Затем некоторый другой поток может «сказать» ждущему, что он может продолжить работу. Это делается с помощью методов монитора `notify()` или `notifyAll()`. Эти методы часто используются для улучшения производительности приложения, предотвращая выполнение потоков в тех случаях, где оно не имеет смысла. В то же время неправильное применение этих методов может привести к взаимоблокировкам или к ситуациям, в которых потоки бесконечно находятся в состоянии ожидания, но никогда не освобождаются для продолжения выполнения.

В этом примере VisualVM показывает конкретные потоки, заблокированные монитором синхронизированного блока кода. При анализе поведения приложения знание того, что означает это состояние, помогает понять, что именно выполняется, и, возможно, обнаружить конкретные проблемы.

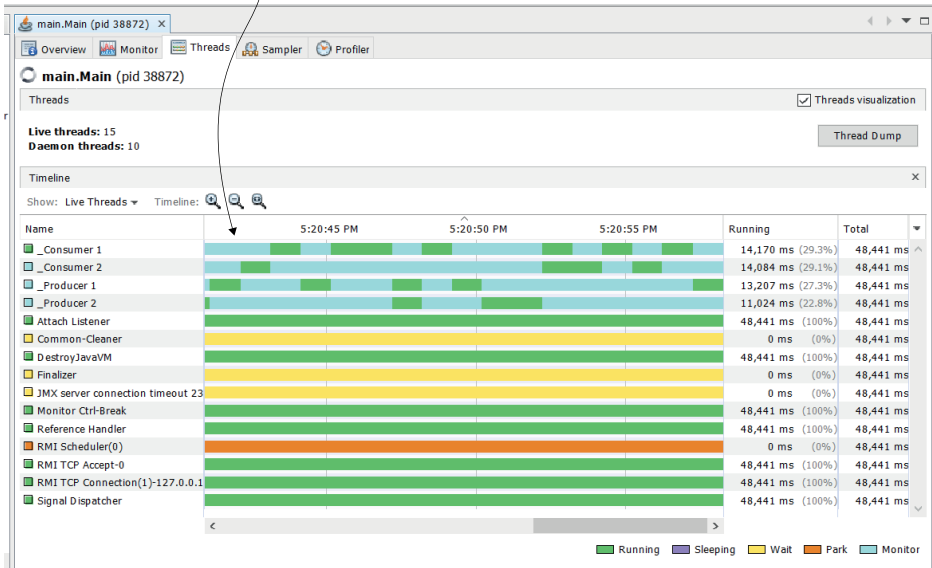


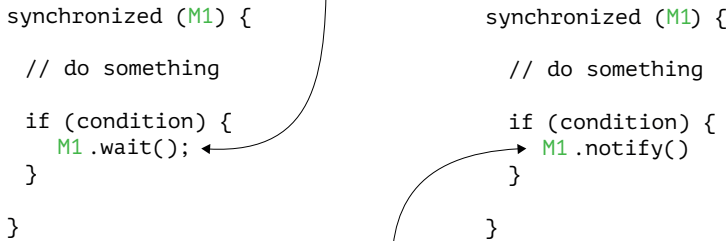
Рис. D.7. VisualVM показывает состояние потока. На вкладке **Threads** в профилировщике представлена полная информация о том, что делает каждый поток, и если поток заблокирован – что заблокировало этот поток

Следует помнить, что `wait()`, `notify()` и `notifyAll()` имеют смысл, только если они используются в синхронизированном блоке. Эти методы определяют поведение монитора синхронизированного блока, поэтому невозможно воспользоваться ими при отсутствии монитора. С помощью метода `wait()` монитор блокирует поток на неопределенное время. При блокировании потока также снимается захваченная им блокировка, так что другие потоки могут входить в блоки, синхронизированные этим монитором. При вызове метода `notify()` заблокированный поток может продолжить выполнение. На рис. D.8 показана работа методов `wait()` и `notify()`.

На рис. D.9 показан более конкретный сценарий. В главе 7 мы рассматривали пример приложения, реализующего модель производитель–потребитель, в котором несколько потоков совместно использовали ресурс. Потоки-производители добавляли значения в совместно используемый ресурс, а потоки-потребители потребляли эти значения. Но что происходит, если совместно используемый ресурс не содержит значения? В этот интервал времени выполнение потребителей не дает никакого преимущества. С технической точки зрения они могут продолжать выполнение, но у них нет значения для потребления, поэтому разрешение от JVM на продолжение их работы может привести к неоправданному потреблению системных ресурсов. Более

правильным было бы «оповещение» потребителей о необходимости ждать при отсутствии значений в совместно используемом ресурсе и продолжить выполнение только после того, как производитель добавит новое значение.

Если по некоторому условию поток должен приостановить выполнение, то вы используете метод монитора `wait()`, чтобы приказывать потоку ждать. В состоянии ожидания поток освобождает блокировку, полученную от монитора, чтобы позволить другим потокам входить в синхронизированные блоки.



```

synchronized (M1) {
    // do something
    if (condition) {
        M1.wait();
    }
}

synchronized (M1) {
    // do something
    if (condition) {
        M1.notify()
    }
}

```

Чтобы позволить ожидающему потоку продолжить выполнение, вы вызываете метод монитора `notify()` или `notifyAll()`.

Рис. D.8. В некоторых случаях выполнение потока необходимо приостановить, и он должен ждать наступления некоторого события. Чтобы заставить поток ждать, монитор синхронизированного блока может вызвать свой метод `wait()`.

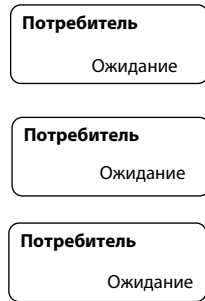
Когда появляется возможность снова продолжить выполнение потока, монитор вызывает метод `notify()` или `notifyAll()`

D.3.3. Присоединение потоков

Достаточно широко применяемой методикой синхронизации потоков является присоединение потоков (joining threads), по которой один поток ожидает, когда другой завершит свое выполнение. Отличие от модели `wait/notify` заключается в том, что поток ждет не оповещения, а просто завершения выполнения другого потока. На рис. D.10 показан сценарий, в котором эта методика синхронизации может оказаться полезной.

Предположим, что имеется некоторая реализация обработки на основе данных, извлекаемых из двух различных независимых источников. Обычно извлечение данных из первого источника занимает около 5 с, а из второго – около 8 с. Если эти операции выполняются последовательно, то время, необходимое для получения всех обрабатываемых данных, равно $5 + 8 = 13$ с. Но вам известен более эффективный подход. Поскольку источниками данных являются две независимые базы данных, можно одновременно извлекать данные из обеих БД, если использовать два потока. Но при этом нужна полная уверенность в том, что поток обработки данных ожидает завершения обоих потоков, извлекающих данные, прежде чем начать свою работу. Для этого вы должны присоединить (`join`) поток обработки к потокам, извлекающим данные (см. рис. D.10).

Три потребителя ждут, потому что список пуст, и у них нет значения для потребления. Не имеет смысла выполнять их при пустом списке, поэтому они должны ждать, чтобы экономить системные ресурсы.



Когда производитель добавляет элемент в список, он оповещает потребителей, используя метод `notify()` или `notifyAll()`. Потребители возвращаются в активное состояние, и JVM может выполнять их.

Рис. D.9. Пример использования методов `wait()` и `notify()`.

Если для потока нет значений при выполнении в текущих условиях, то можно заставить его ждать до следующего оповещения. В данном случае потребитель не должен выполняться, если для него нет значения. Можно перевести потребителей в состояние ожидания, а производитель может сообщить им о возможности продолжения только после добавления нового значения в совместно используемый ресурс

Во многих случаях присоединение потоков является необходимой методикой синхронизации. Но при некорректном использовании она также может стать источником проблем. Например, если один поток, ожидающий другого, зависает или никогда не завершается, то присоединенный поток никогда не будет выполнен.

D.3.4. Блокировка потоков на определенное время

Иногда поток должен ожидать в течение определенного интервала времени. В этом случае поток находится в состоянии «ожидания с ограничением по времени» или «спит». Перечисленные ниже операции наиболее часто применяются для перевода потока в состояние ожидания с ограничением по времени:

- `sleep()` – всегда можно использовать статический метод `sleep()` из класса `Thread`, чтобы перевести текущий поток, выполняющий код, в состояние ожидания на фиксированный интервал времени;

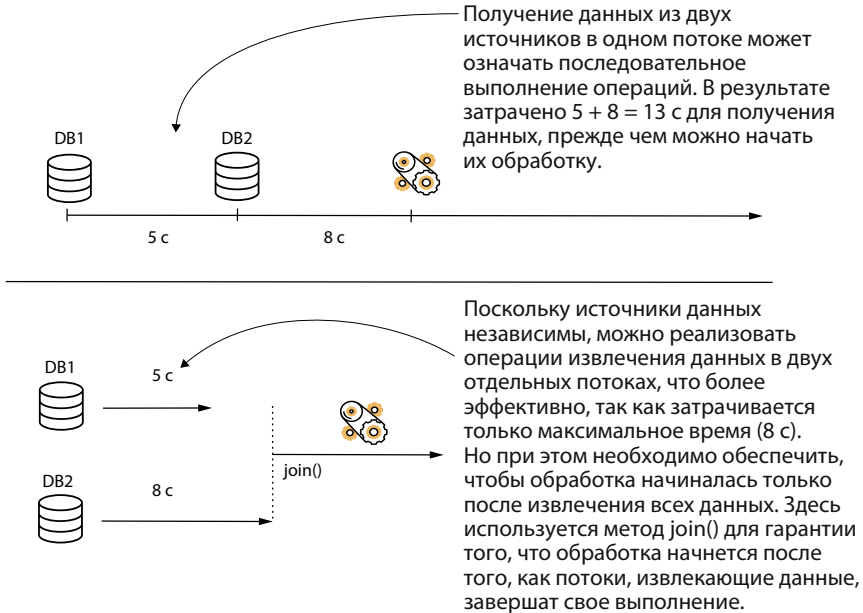


Рис. D.10. В некоторых случаях можно улучшить производительность приложения, используя несколько потоков. Но при этом необходимо приказывать некоторым потокам ждать, поскольку они зависят от результатов выполнения других потоков. Заставить потоки ждать других можно с помощью операции присоединения

- `wait(long timeout)` – метод `wait()` с параметром `timeout` можно использовать так же, как и метод `wait()` без параметров, как показано в подразделе D.3.2. Но если передается параметр, то поток будет ждать в течение заданного интервала времени, если раньше не получит оповещение;
- `join(long timeout)` – эта операция работает так же, как метод `join()`, описанный в подразделе D.3.3, но находится в состоянии ожидания в течение интервала времени, максимальное значение которого задано переданным параметром `timeout`.

Типичным антипаттерном, который я часто обнаруживаю в приложениях, является использование `sleep()` для того, чтобы перевести поток в состояние ожидания, вместо метода `wait()`, как описано в главе 4. Снова обратимся к ранее рассмотренному примеру архитектуры производитель–потребитель. Можно было бы использовать `sleep()` вместо `wait()`, но сколько должен «спать» потребитель, чтобы предоставить производителю достаточное время для выполнения и добавления значений в совместно используемый ресурс? На этот вопрос нет ответа. Например, засыпание потока на 100 мс (как показано на рис. D.11) может оказаться слишком длительным или слишком коротким. В большинстве случаев при использовании такого подхода в итоге вы не получите наилучшую производительность.

Иногда ожидание с ограничением по времени ошибочно используется вместо неограниченного ожидания. Функциональность такого подхода иногда может сработать, но обычно это реализация с ухудшенной производительностью.

```
synchronized (M1) {  
  
    // do something  
  
    if (condition) {  
        Thread.sleep(100);  
    }  
  
}
```




Рис. D.11. Методика ожидания с ограничением по времени вместо `wait()` и `notify()` обычно не является наилучшей стратегией. Если непосредственно в коде есть возможность определить, когда поток может продолжить выполнение, то используйте `wait()` и `notify()` вместо `sleep()`

D.3.5. Синхронизация потоков с блокирующими объектами

JDK предоставляет впечатляющий комплект инструментальных средств для синхронизации потоков. Некоторые из них являются хорошо известными классами, используемыми в многопоточных архитектурах:

- Semaphore – объект, применяемый для ограничения количества потоков, которые могут выполнять конкретный блок кода;
- CyclicBarrier – объект, применяемый для обеспечения гарантии того, что как минимум заданное количество потоков является активным для выполнения конкретного блока кода;
- Lock – объект, предоставляющий более широкие возможности синхронизации;
- Latch – объект, применяемый для того, чтобы перевести некоторые потоки в состояние ожидания до тех пор, пока не будет выполнена определенная логика в других потоках.

Эти объекты представляют собой реализации высокого уровня, и каждая развертывает точно определенный механизм для упрощения реализации в конкретных сценариях. В большинстве случаев эти объекты становятся источниками потенциальных проблем из-за некорректного их использования, а во многих случаях разработчики чрезмерно усложняют код такими объектами. Мой совет: применяйте самое простое решение, которое можете найти для устранения проблемы, и, прежде чем использовать любой из вышеперечисленных объектов, убедитесь в том, что вы правильно понимаете, как они работают.

D.4. Проблемы, наиболее часто возникающие в многопоточных архитектурах

При анализе многопоточных архитектур вы будете обнаруживать общие проблемы, которые являются главными причинами разнообразного неожиданного поведения (вывод, отличающийся от предполагаемого, или проблема производительности). Предварительное понимание сущности этих проблем поможет вам быстрее определить, где возникла конкретная проблема, и устранить ее. Ниже кратко описаны эти общие проблемы:

- *состояние гонки* (race conditions) – два или более потоков соперничают за изменение совместно используемого ресурса;
- *взаимоблокировки* (deadlocks) – два или более потоков останавливают свое выполнение в ожидании друг друга;
- *динамические* (активные) *взаимоблокировки* (livelocks) – два или более потоков не обнаруживают условия для остановки и непрерывно выполняются без совершения какой-либо полезной работы;
- *голодание* (зависание) (starvation) – поток постоянно заблокирован, тогда как JVM выполняет другие потоки. Такой поток никогда не выполнит определенные в нем инструкции.

D.4.1. Состояние гонки

Состояние гонки возникает, когда несколько потоков пытаются одновременно изменить совместно используемый ресурс. Когда это происходит, мы можем получить непредсказуемые результаты или неожиданные исключения. Обычно для того, чтобы избежать подобных ситуаций, применяются методики синхронизации. На рис. D.12 наглядно представлен один из таких случаев. Потоки T1 и T2 одновременно пытаются изменить значение переменной *x*. Поток T1 пытается увеличить значение *x*, а поток T2 – уменьшить его. Этот сценарий может создавать различные варианты выходных данных при повторном выполнении приложения. Возможны следующие сценарии:

- после выполнения операций значение *x* может быть равно 5 – если поток T1 изменил значение первым, а поток T2 прочитал уже измененное значение переменной *x*, или наоборот, то значение переменной останется равным 5;
- после выполнения операций значение *x* может быть равно 4 – если оба потока одновременно прочитали значение *x*, но T2 записал значение последним, то значение *x* будет равно 4 (значению, прочитанному T2, т. е. 5 минус 1);
- после выполнения операций значение *x* может быть равно 6 – если оба потока одновременно прочитали значение *x*, но T1 записал зна-

чение последним, то значение x будет равно 6 (значению, прочитанному T1, т. е. 5 плюс 1).

Подобные ситуации обычно приводят к выводу неожиданного результата. В многопоточных архитектурах, где возможно существование нескольких потоков выполнения, такие сценарии весьма трудно воспроизвести. Иногда они происходят только в конкретных средах, что затрудняет анализ.

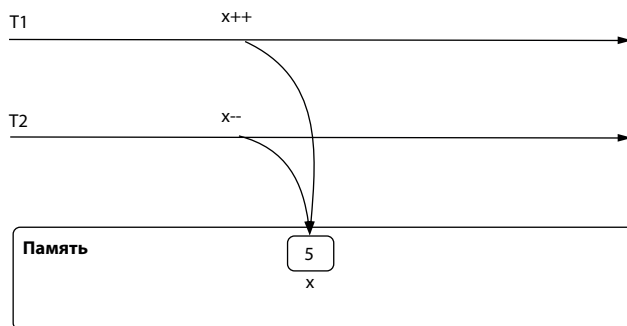


Рис. D.12. Состояние гонки. Несколько потоков одновременно пытаются изменить совместно используемый ресурс. В рассматриваемом здесь примере потоки T1 и T2 пытаются изменить значение переменной x одновременно, что может привести к выводу различных результатов

D.4.2. Взаимоблокировки

Взаимоблокировки – это ситуации, в которых два или более потоков приостанавливаются, а затем ждут чего-то друг от друга, чтобы продолжить свое выполнение (см. рис. D.13). Взаимоблокировки заставляют приложение или по крайней мере его часть остановиться (зависнуть), не позволяя выполнить некоторые функции.



Рис. D.13. Пример взаимоблокировки. В случае, когда T1 ждет T2 для продолжения выполнения, а T2 ждет T1, оба потока находятся в состоянии взаимоблокировки. Ни один из них не может продолжить выполнение, потому что они ждут друг друга

На рис. D.14 показан вариант взаимоблокировки, который может возникнуть в коде. В этом примере один поток захватывает блокировку ресурса A, а другой – блокировку ресурса B. Но каждому потоку также необходим ресурс, захваченный другим потоком, для продолжения выполнения. Поток T1 ждет, когда T2 освободит ресурс A, но в то же время поток T2 ждет освобождения ресурса B потоком T1. Ни один из потоков не может продол-

жить выполнение, потому что оба ждут друг от друга освобождения необходимых ресурсов, и в результате возникает взаимоблокировка.

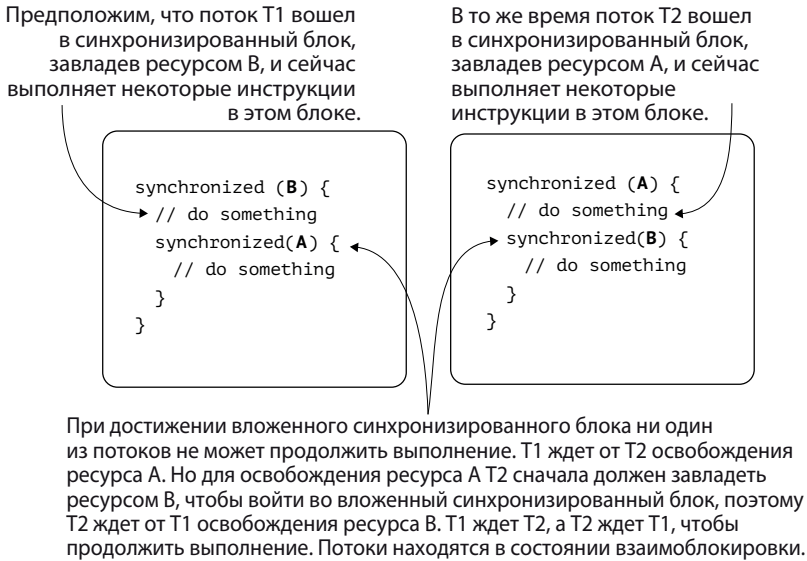


Рис. D.14. Взаимоблокировка. Поток T1 не может войти во вложенный синхронизированный блок, потому что T2 владеет блокировкой ресурса A.

Поток T1 ждет от T2 освобождения ресурса A, чтобы получить возможность продолжить свое выполнение. Но поток T2 находится в такой же ситуации: он не может продолжить выполнение, потому что T1 владеет блокировкой ресурса B. Поток T2 ждет от потока T1 освобождения ресурса B, чтобы продолжить свое выполнение. Поскольку оба ждут друг друга и ни один из них не может продолжить выполнение, они находятся в состоянии взаимоблокировки

Пример, представленный на рис. D.14, простой, но он предназначен только для учебных целей. Сценарии из реальной практики обычно гораздо сложнее проанализировать и понять, а кроме того, в них могут участвовать более двух потоков. Помните о том, что синхронизированные блоки – это не единственный путь входа в состояние взаимоблокировки. Самый лучший способ понять подобные сценарии – применение методик анализа, которые вы изучили в главах 7–9.

D.4.3. Динамические (активные) взаимоблокировки

Динамические (активные) взаимоблокировки в определенной степени являются противоположностью обычным взаимоблокировкам. Когда потоки находятся в состоянии динамической взаимоблокировки, условие всегда изменяется так, что потоки продолжают выполнение, даже если они должны остановиться по конкретно заданному условию. Потоки не могут остановиться и выполняются непрерывно, обычно потребляя системные

ресурсы без должного основания. Динамические взаимоблокировки могут стать причиной возникновения проблем с производительностью при выполнении приложения.

На рис. D15 показана динамическая (активная) взаимоблокировка на диаграмме последовательностей. Два потока T1 и T2 выполняются в цикле. Для остановки выполнения T1 делает условие истинным перед своей последней итерацией. В следующий раз, когда T1 возвращается к этому условию, ожидается, что оно истинно и поток должен остановиться. Но этого не происходит, поскольку другой поток T2 изменил условие на false (ложь). Поток T2 находится в такой же ситуации. Каждый поток изменяет условие, чтобы он мог остановиться, но в то же время каждое изменение условия заставляет другой поток продолжать выполнение.

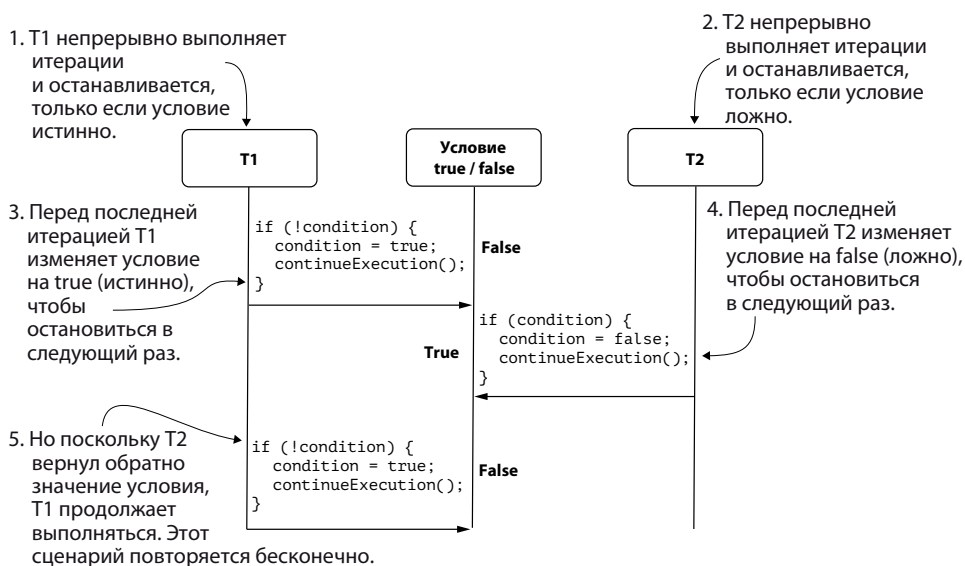


Рис. D.15. Пример динамической (активной) взаимоблокировки.

Два потока зависят от условия остановки их выполнения. Но при изменении значения условия для создания возможности остановки каждый поток заставляет другой продолжать выполнение. Потоки не могут остановиться, следовательно, нерационально расходуют системные ресурсы

Как и в примере взаимоблокировки в главе 4 (подраздел 4.4.2), следует помнить, что это упрощенный сценарий. Динамические взаимоблокировки могут создаваться более сложными сценариями в реальной практике, и в них могут участвовать более двух потоков. В главах 7–9 рассматривается несколько способов анализа подобных сценариев.

D.4.4. Голодание (зависание)

Еще одной общей проблемой, хотя и реже возникающей в современных приложениях, является голодание (зависание). Зависание возникает, ког-

да конкретный поток постоянно исключен из процесса выполнения, даже если является выполняемым (находится в состоянии `runnable`). Такой поток хочет выполнить свои инструкции, но JVM постоянно позволяет другим потокам получить доступ к системным ресурсам. Поскольку поток не может получить доступ к системным ресурсам и выполнить определенный в нем набор инструкций, мы говорим, что он голодает (завис).

В ранних версиях JVM такие ситуации возникали, когда разработчик устанавливал слишком низкий приоритет для конкретного потока. Современные версии JVM намного интеллектуальнее работают в подобных случаях, поэтому сценарии зависания менее вероятны (по крайней мере, в моей практике).

D.5. Материал для дополнительного чтения

Работа с потоками сложна, и в этом приложении рассматривались самые важные аспекты, которые помогут вам понять методики, описанные в данной книге. Но для любого разработчика Java глубокое понимание того, как работают потоки, является полезным навыком. Ниже приведен список ресурсов, которые я рекомендую для более подробного изучения потоков.

- «Oracle Certified Professional Java SE 11 Developer Complete Study Guide», Джин Боярски (Jeanne Boyarsky) и Скотт Селикофф (Scott Selikoff) (Sybex, 2020 г.). В главе 18 описаны потоки и параллельный режим выполнения, начиная с нуля, и рассматриваются все основы работы с потоками. Требуется OCP-сертификация. Изучение потоков рекомендую начать с этой книги.
- Второе издание «The Well-Grounded Java Developer», Бенджамин Эванс (Benjamin Evans), Джейсон Кларк (Jason Clark), Мартин Вербург (Martijn Verburg) (Manning, 2022 г.). Здесь рассматривается параллельный режим выполнения – от основ до тонкой настройки производительности.
- «Java Concurrency in Practice», Брайан Гётц (Brian Goetz) и др. (Addison-Wesley, 2006 г.). Это более старая книга, но ее ценность не уменьшилась. Ее обязан прочитать каждый разработчик Java, стремящийся пополнить свои знания о потоках и параллельном выполнении.

Приложение **Е**

.....

Управление памятью в Java-приложениях

В этом приложении мы рассмотрим, как виртуальная машина JVM (Java Virtual Machine) управляет памятью в Java-приложениях. Некоторые из наиболее сложных проблем, которые вам придется анализировать в Java-приложениях, связаны со способом управления памятью в этих приложениях. К счастью, мы можем использовать несколько методик для анализа подобных проблем и поиска главных причин их возникновения с минимальными затратами времени. Но чтобы извлечь пользу из этих методик, необходимо знать по меньшей мере некоторые основы работы механизма управления памятью в Java-приложениях.

Память приложения – ограниченный ресурс. Даже если современная система может предоставить большой объем памяти для использования приложением во время выполнения, все же следует внимательно относиться к тому, как приложение потребляет этот ресурс. Ни одна система не может предоставить неограниченную память как «волшебное» решение (см. рис. Е.1). Проблемы с памятью приводят к проблемам с производительностью (приложение начинает работать медленно, увеличиваются затраты на его развертывание, его запуск требует больше времени и т. д.), а иногда даже может полностью остановить весь процесс (например, при возникновении ошибки `OutOfMemoryError`).

Мы рассмотрим самые важные аспекты управления памятью. В разделе Е.1 описано, как JVM организует память для выполняющегося процесса. Вы узнаете о трех способах распределения памяти приложения – это стек, куча и метапространство. В разделе Е.2 рассматривается стек – пространство памяти, используемое потоками для хранения локально объявленных переменных и соответствующих данных. В разделе Е.3 описана куча и способ хранения приложением экземпляров объектов в памяти. В заключительном разделе Е.4 рассматривается метапространство – локация памяти, где приложение хранит метаданные о типах объектов.

Следует помнить, что управление памятью в Java-приложениях – это сложный механизм. В данном приложении я представляю только те подробности, которые необходимы для понимания методик, описанных в этой книге.

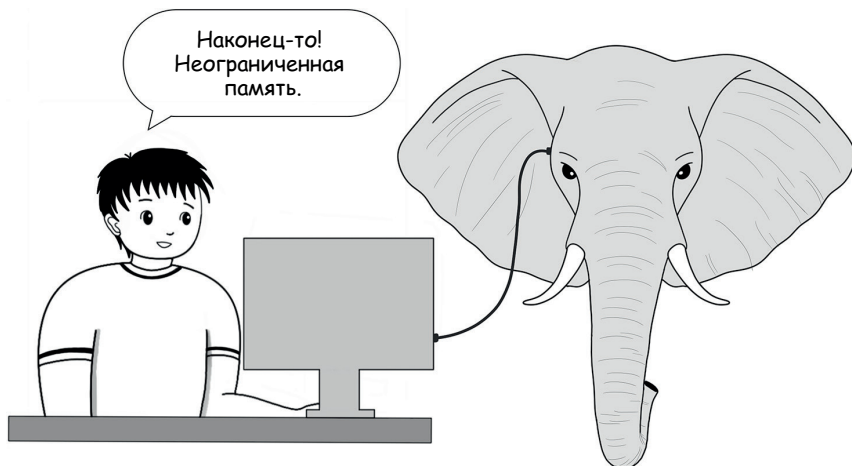


Рис. Е.1. Память приложения – ограниченный ресурс. Не существует «волшебного» решения, позволяющего предоставить неограниченную память приложению. При разработке приложения необходимо внимательно следить за потреблением памяти и избегать ее необоснованного расходования. Иногда в приложениях могут возникать проблемы с памятью. Если какой-то функциональный компонент использует слишком много памяти, это может стать причиной возникновения проблем с производительностью или даже полной аварии. Вы должны быть всегда готовы к поиску причин возникновения таких проблем и их устранению

Е.1. Как JVM организует память приложения

В этом разделе мы рассмотрим, как JVM организует данные в различных локациях памяти, каждая из которых управляется по-разному. Понимание того, как JVM управляет памятью, чрезвычайно важно для анализа проблем, связанных с памятью. Мы будем использовать некоторые визуальные представления при обсуждении основных аспектов управления памятью, и вы узнаете о распределении данных в памяти Java-приложения. Затем мы подробно рассмотрим управление памятью в каждой локации.

Пока предположим (для упрощения обсуждения), что в Java-приложении существует два способа управления данными, хранимыми во время выполнения: стек и куча. В зависимости от того, как определены эти данные, приложение размещает их либо в стеке, либо в куче. Но, прежде чем начать обсуждение локаций размещения данных, напомним о весьма важной подробности: в приложении имеется более одного потока, что позволяет обрабатывать данные параллельно. Куча – это отдельная специальная локация памяти, и все потоки в приложении используют ее. Но у каждого потока есть собственная локация памяти под названием стек. Когда разработчики только начинают изучать механизм управления памятью, из-за этого может возникать путаница. На рис. Е.2 эти подробности представлены визуально.

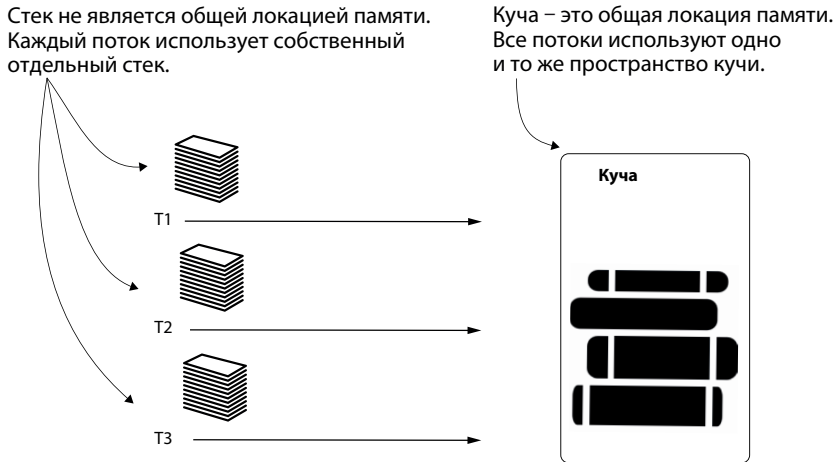


Рис. Е.2. T1, T2 и T3 – потоки в Java-приложении. Все они используют одну и ту же кучу. Куча – это локация памяти, в которой приложение хранит данные экземпляров объектов. Но каждый поток использует собственную локацию памяти – стек – для хранения локально объявленных данных

Стек – это локация памяти, принадлежащая потоку. Каждый поток владеет собственным стеком, который не используется другими потоками. В этой локации памяти поток хранит данные, локально объявленные в блоке кода и выполняемые этим потоком. Предположим, что имеется метод, представленный в показанном ниже фрагменте кода. Параметры *x*, *y* и переменная *sum*, объявленные внутри блока кода метода, являются локальными переменными. Эти значения сохраняются в стеке потока при выполнении метода:

```
public int sum(int x, int y) {           | ❶
    int sum = x + y;
    return sum;
}
```

❶ Переменные *x*, *y* и *sum* сохраняются в стеке.

Куча – это локация памяти, в которой приложение хранит данные экземпляров объектов. Предположим, что приложение объявляет класс *Cat*, как показано в следующем фрагменте кода. При каждом создании экземпляра с использованием конструктора класса `new Cat()` этот экземпляр отправляется в кучу:

```
public class Cat {
}
```

Если класс объявляет атрибуты экземпляра, то JVM также сохраняет их значения в куче. Например, если класс *Cat* выглядит так, как показано в

приведенном ниже фрагменте кода, то JVM сохранит имя и возраст каждого экземпляра в куче:

```
public class Cat {
    private String name;
    private int age;
}
```

❶ Атрибуты объекта сохраняются в куче.

На рис. Е.3 визуально представлен пример размещения данных. Обратите внимание: локально объявленные переменные и их значения (x и c) сохранены в стеке потока, тогда как экземпляр `Cat` и его данные размещены в куче приложения. Ссылка на экземпляр `Cat` будет сохранена в стеке потока в переменной c . Даже параметр метода, в котором хранится ссылка на массив типа `String`, становится частью стека.

В этом примере мы рассматриваем основной поток, который начинает свое выполнение с метода `main()`. Переменные, объявленные локально в методе `main()`, хранятся в стеке основного потока. Значения в стеке: переменная x , содержащая значение 10, и переменная c , содержащая ссылку на объект `Cat`.

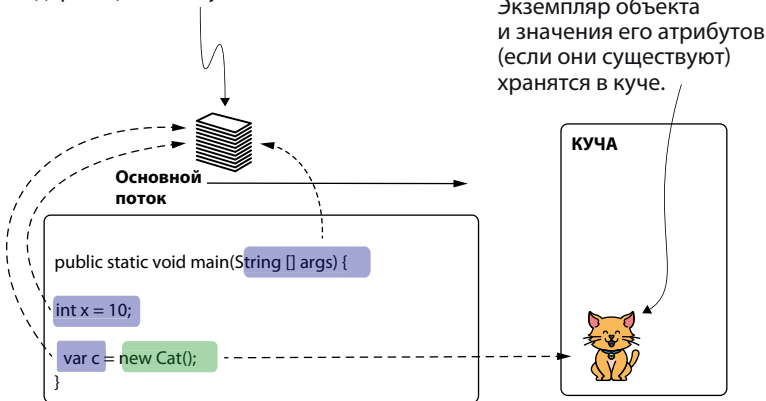


Рис. Е.3. Приложение сохраняет локально объявленные переменные в стеке потока, а данные, определяющие экземпляр объекта, – в куче. Переменная в стеке одного из потоков может ссылаться на объект в куче. В этом примере переменная x , содержащая значение 10, и переменная c , содержащая ссылку на экземпляр `Cat`, являются частью стека потока

Е.2. Стек, используемый потоками для хранения локальных данных

В этом разделе мы более подробно проанализируем работу механизма стека. Из раздела Е.2 вы узнали, что локальные переменные хранятся в стеке

и каждый поток владеет собственной локацией стека. Теперь рассмотрим, как хранятся эти значения и когда приложение удаляет их из памяти. Мы будем использовать визуальные представления для пошагового описания этого процесса, а также небольшой пример кода. После завершения изучения механизма управления памятью стека мы обсудим, что может пойти не так, создавая проблемы, связанные со стеком.

Почему эта локация памяти называется стеком? Стек потока использует принципы работы структуры данных под названием «стек». Стек (stack) – это упорядоченный набор, в котором всегда можно удалить только самый последний добавленный элемент. Обычно мы визуальным образом представляем такой набор как стопку слоев, где каждый слой располагается поверх предыдущего. Добавить новый слой можно только поверх всех существующих, а удалить можно только самый верхний слой. Такой метод добавления и удаления элементов также называется LIFO (*last in, first out* – «последним вошел, первым вышел»). На рис. Е.4 показана работа стека с последовательностью шагов добавления и удаления. Для упрощения примера значениями в стеке являются числа.

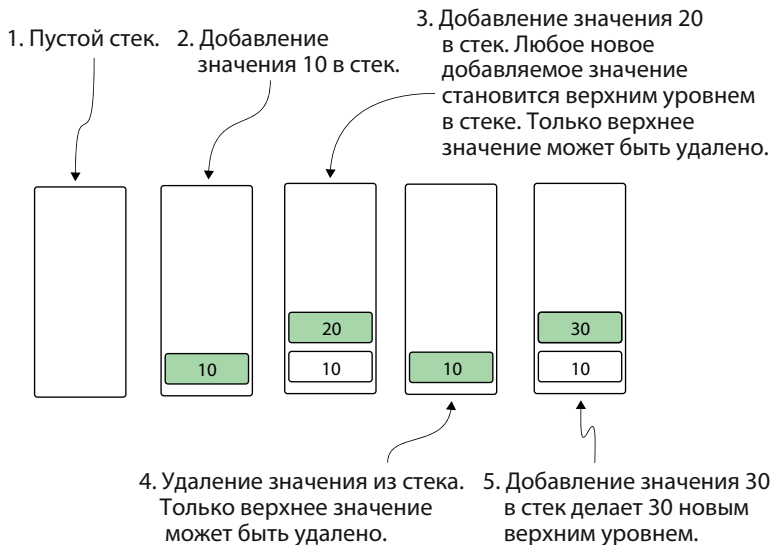


Рис. Е.4. Добавление значений в стек и удаление из стека. Стек – это упорядоченный набор, работающий по принципу LIFO. При добавлении в стек значение становится верхним уровнем – единственным, который можно удалить

Точно такое же поведение наблюдается при управлении приложением данными в стеке потока. Когда выполнение достигает начала блока кода, создается новый уровень в стеке потока. Согласно основному принципу работы стека любой новый уровень становится верхним и является первым кандидатом на удаление. На рис. Е.5, Е.6, Е.7, Е.8 мы последовательно наблюдаем выполнение простого фрагмента кода шаг за шагом, чтобы проследить, как изменяется стек потока:

```

public static void main(String [] args) {
    int x = 10;
    a();
    b();
}

public static void a() {
    int y = 20;
}

public static void b() {
    int y = 30;
}

```

Выполнение начинается с метода `main()` (см. рис. Е.5). Когда выполнение достигает начала метода `main()`, первый уровень добавляется в стек потока. Этот уровень является локацией памяти, где сохраняется каждое локальное значение, объявленное в текущем блоке кода. В данном случае в блоке кода объявляется переменная `x`, которая инициализируется значением `10`. Переменная сохраняется в только что созданном уровне стека потока. Этот уровень будет удален из стека, когда метод завершит свое выполнение.

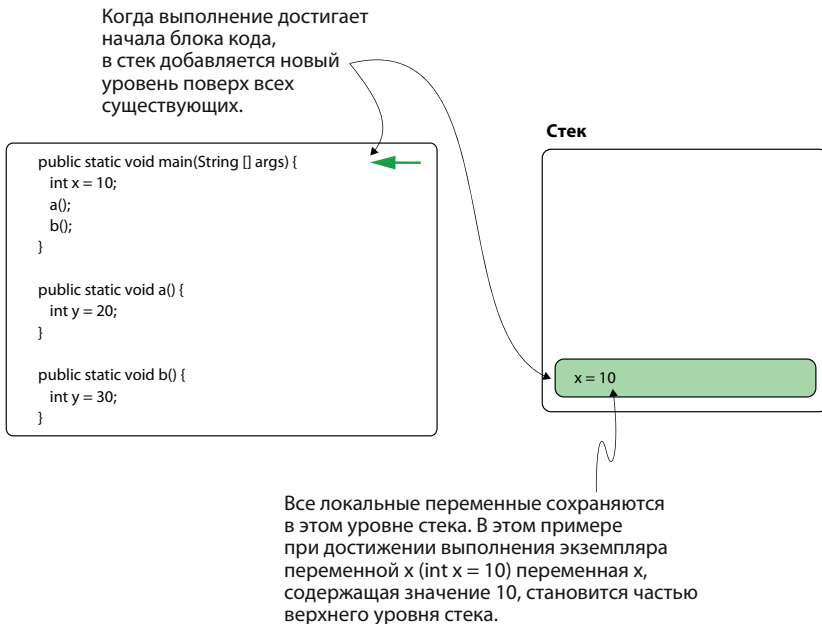


Рис. Е.5. Когда выполнение достигает начала блока кода, создается новый уровень в стеке потока. Все переменные, определяемые текущим блоком кода, сохраняются в этом новом уровне. Уровень удаляется, когда завершается блок кода. Таким образом, нам известно, что значения в этой части памяти удаляются, когда они больше не нужны

Любой блок кода может вызывать другие блоки кода. Например, в рассматриваемом здесь случае метод `main()` вызывает методы `a()` и `b()`, которые работают аналогично. Когда выполнение переходит к началу их блоков кода, в стек добавляется новый уровень. Такой новый уровень – это локация памяти, где хранятся все данные, объявленные локально. На рис. Е.6 показано, что происходит, когда выполнение переходит к методу `a()`.

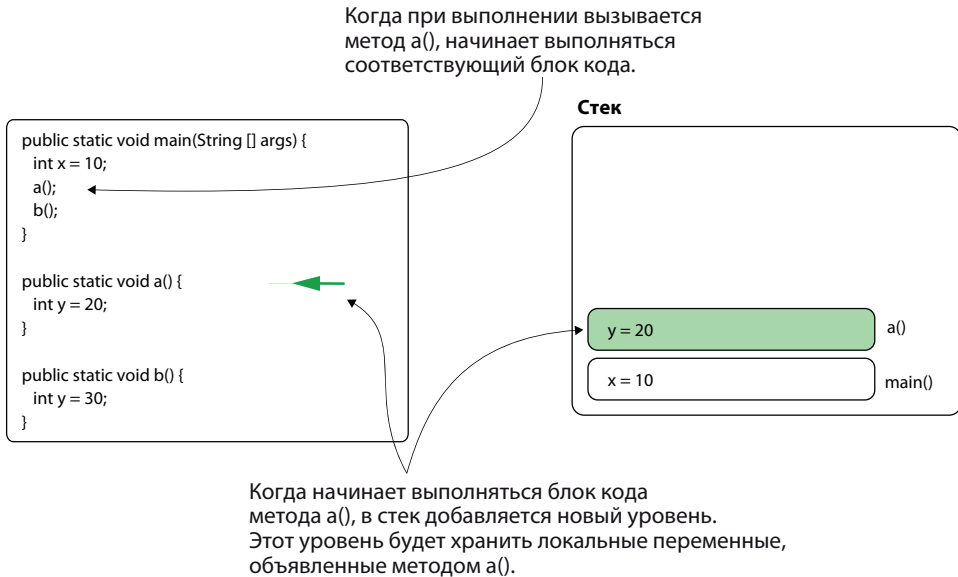
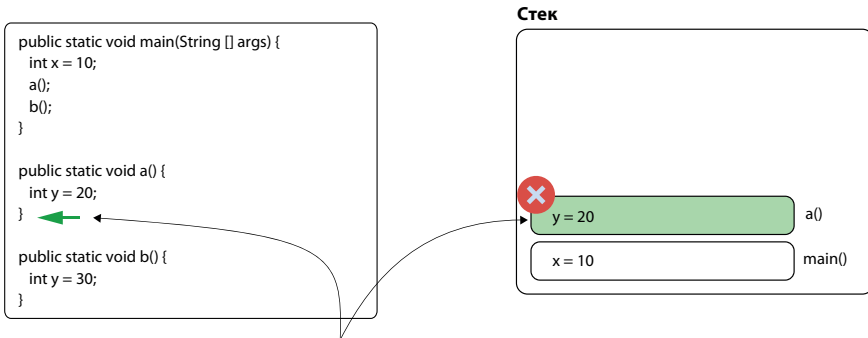


Рис. Е.6. При выполнении из блока кода может быть вызван другой блок. В данном случае метод `main()` вызывает метод `a()`. Поскольку `main()` не был завершен, его уровень остается частью стека. Метод `a()` создает собственный уровень, где будут храниться определяемые им локальные переменные

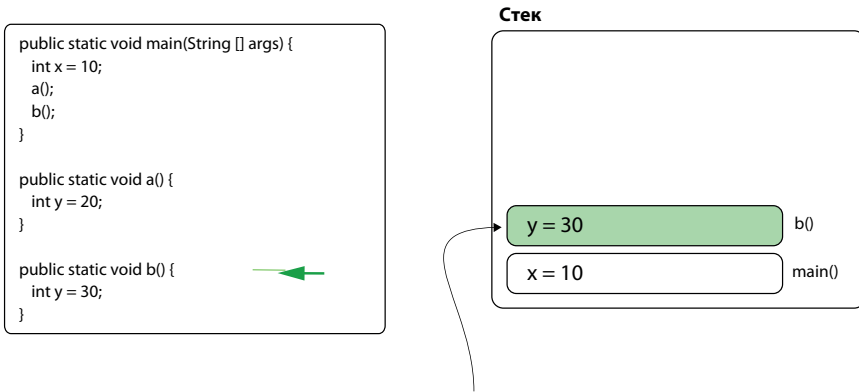
Когда метод `a()` завершает свое выполнение и происходит возврат в `main()`, уровень, созданный в стеке потока, также удаляется (см. рис. Е.7). Это означает, что ранее сохраненных данных уже нет в памяти. Таким образом, память, которая больше не нужна, освобождается, предоставляя пространство для хранения новых данных. Блок кода завершается, когда выполнение достигает последней инструкции блока, выполняется инструкция `return` или генерируется исключение. Обратите внимание: при завершении блока кода соответствующий ему уровень всегда является самым верхним в стеке, соблюдая принцип LIFO.



Когда выполнение достигает конца блока кода (или происходит возврат из метода, или генерируется исключение), соответствующий уровень и все его содержимое удаляется из стека.

Рис. Е.7. Когда выполнение достигает конца блока кода, созданный для этого блока уровень стека удаляется со всеми данными, содержащимися в нем. В данном случае при возврате из метода `a()` соответствующий ему уровень стека удаляется. Таким образом, мы можем быть уверены в том, что ненужные данные удаляются из памяти.

Метод `main()` продолжает выполнение и вызывает метод `b()`. Как и метод `a()` ранее, метод `b()` создает новый уровень в стеке для хранения объявляемых им локальных данных (см. рис. Е.8).



Когда выполнение переходит к методу `b()`, уровень стека для метода `a()` уже не существует. Метод `b()` создает собственный уровень в стеке и хранит в нем объявляемые локальные переменные. Когда метод `b()` завершает выполнение, соответствующий уровень в стеке также будет удален. То же самое происходит и с методом `main()`. В конце, когда поток завершает выполнение, стек становится пустым.

Рис. Е.8. Как и в случае с методом `a()`, при вызове метода `b()` выполнение переходит к началу соответствующего блока кода, и в стек добавляется новый уровень. Метод `b()` может использовать этот уровень для хранения локальных данных до тех пор, пока не произойдет выход из этого метода, и уровень будет удален.

Поскольку каждый поток владеет собственным стеком, исключение `StackOverflowError` воздействует только на тот поток, стек которого переполнен. Процесс может продолжать свое выполнение, и на другие потоки не будет оказано никакого воздействия. Кроме того, исключение `StackOverflowError` создает трассировку стека, которую можно использовать для определения кода, в котором возникла проблема. На рис. Е.10 показан пример такого типа трассировки стека. Вы можете воспользоваться проектом `da-app-e-ex1`, приложенным к этой книге, чтобы воспроизвести показанную здесь трассировку стека.

Exception in thread "main" java.lang.StackOverflowError

```
...
at main.Main.a(Main.java:11)
at main.Main.b(Main.java:16)
at main.Main.a(Main.java:11)
at main.Main.b(Main.java:16)
at main.Main.a(Main.java:11)
at main.Main.b(Main.java:16)
at main.Main.a(Main.java:11)
at main.Main.b(Main.java:16)
at main.Main.a(Main.java:11)
at main.Main.b(Main.java:16)
at main.Main.a(Main.java:11)
...
```

Когда вы получаете исключение стека, подобное показанному здесь, которое выглядит как собака, гонящаяся за собственным хвостом, вероятнее всего, вы имеете дело с рекурсией с некорректным условием завершения.

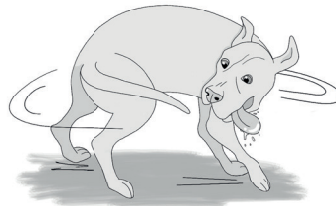


Рис. Е.10. Трассировка стека, созданная исключением `StackOverflowError`. Обычно `StackOverflowError` легко идентифицировать. Трассировка стека показывает метод, вызывающий себя многократно, или группу методов, вызывающих друг друга, как в этом примере. Вы можете перейти прямо к этим методам, чтобы определить, как они начали бесконечно вызывать друг друга

Е.3. Куча, которую приложение использует для хранения экземпляров объектов

В этом разделе рассматривается куча: локация памяти, совместно используемая всеми потоками Java-приложения. Куча хранит данные экземпляра объекта. Как вы увидите в этом разделе, в куче проблемы возникают чаще, чем в стеке. Кроме того, главные причины возникновения проблем, связанных с кучей, гораздо труднее обнаружить. Мы рассмотрим, как объекты хранятся в куче и кто может хранить ссылки на них, – это связано с пониманием, когда они могут быть удалены из памяти. Далее мы обсудим главные причины возникновения проблем, связанных с кучей. Эта информация необходима для того, чтобы понимать методики анализа, описанные в главах 7–9.

ПРИМЕЧАНИЕ. Куча имеет сложную структуру. Мы не будем рассматривать все подробности организации кучи, так как прямо сейчас они вам не нужны. Мы также не будем касаться таких подробностей, как пул строк или генерации кучи.

Первое, что необходимо запомнить о куче, – это локация памяти, совместно используемая всеми потоками (см. рис. Е.11). Это не только допускает возникновение проблем, связанных с потоками, таких как состояние гонки (описанное в приложении D), но также значительно усложняет анализ проблем с памятью. Поскольку все потоки добавляют создаваемые экземпляры объектов в одну и ту же локацию памяти, один поток может воздействовать на выполнение других. Если в одном потоке возникает утечка памяти (т. е. экземпляры добавляются в память, но никогда не удаляются), это повлияет на весь процесс в целом, потому что другие потоки также пострадают от этой утечки памяти.

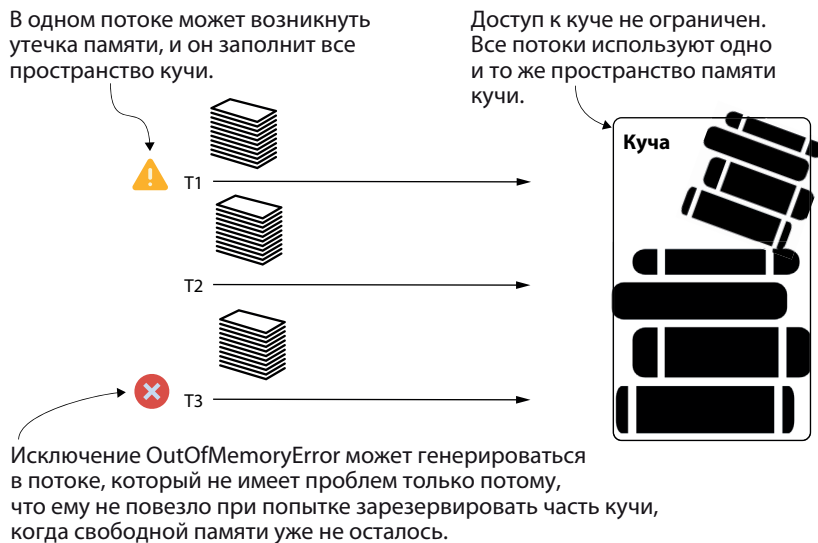


Рис. Е.11. Все потоки используют одну и ту же локацию памяти кучи.

Если в одном из потоков возникает проблема, приводящая к заполнению кучи до отказа (утечка памяти), то другой поток может сообщить о проблеме. Такой сценарий возникает достаточно часто, потому что о проблеме сообщает самый первый поток, который не может сохранить данные в куче. Поскольку любой поток может оповестить о проблеме, но не всегда проблема возникает именно в нем, проблемы, связанные с кучей, труднее устранить.

В большинстве случаев при генерации исключения `OutOfMemoryError`, как показано на рис. Е.11, о ситуации сообщает другой поток, а не тот, в котором содержится главная причина возникновения проблемы (утечки памяти). Об исключении `OutOfMemoryError` сигнализирует самый первый поток,

который пытается добавить что-то в память, но не может этого сделать по причине отсутствия свободного пространства памяти.

Сборщик мусора (garbage collector – GC) – это механизм, который освобождает память в куче, удаляя ненужные данные. GC знает, что экземпляр объекта больше не нужен, если на него нет ссылок. Таким образом, если объект уже не нужен, но приложение почему-то не удалило все ссылки на него, то GC не сможет удалить этот объект. Если приложение постоянно «забывает» удалять ссылки на вновь создаваемые объекты вплоть до заполнения памяти до отказа (что приводит к исключению `OutOfMemoryError`), мы говорим, что в приложении имеется утечка памяти.

На экземпляре объекта может существовать ссылка из другого объекта в куче (см. рис. Е.12). Общеизвестным примером утечки памяти является набор, в который непрерывно добавляются ссылки на объекты. Если эти ссылки не удаляются, то, пока этот набор находится в памяти, GC не может удалить объекты ссылок, и это становится утечкой памяти. Особое внимание необходимо уделять статическим объектам (экземплярам объектов, на которые существуют ссылки из статических переменных). Эти переменные не исчезают после того, как они были созданы, поэтому если вы явно не удаляете их, то можно предположить, что ссылка на объект из статической переменной будет существовать в течение всего жизненного цикла процесса. Если объект является набором, ссылающимся на другие объекты, которые никогда не удаляются, это может стать потенциальной утечкой памяти.

Объекты в куче могут ссылаться друг на друга. В данном случае экземпляр `Cat` не может быть удален из памяти сборщиком мусора до тех пор, пока не будет удалена ссылка на него, созданная экземпляром `Person`, или сам экземпляр `Person` не будет удален.

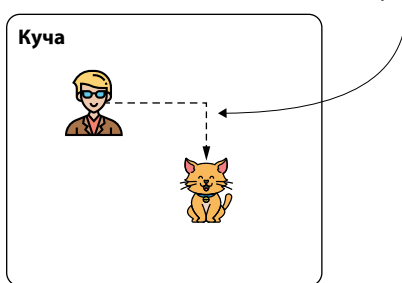


Рис. Е.12. Любой объект в куче может сохранять ссылки на другие объекты в куче. GC может удалить объект, только если на него нет ссылок

Кроме того, на любой экземпляр объекта может существовать ссылка из стека (см. рис. Е.13). Обычно ссылки из стека не приводят к утечкам памяти, поскольку (как отмечено в разделе Е.2) уровень стека автоматиче-

ски удаляется, когда выполнение достигает конца блока кода, для которого приложение создало этот уровень. Но в особых случаях в сочетании с другими проблемами ссылки из стека также могут стать причиной беспокойства. Допустим, что существует взаимоблокировка, препятствующая выполнению всего блока кода в целом. Соответствующий уровень в стеке не будет удален, и, если в нем есть ссылки на объекты, это также может стать утечкой памяти.

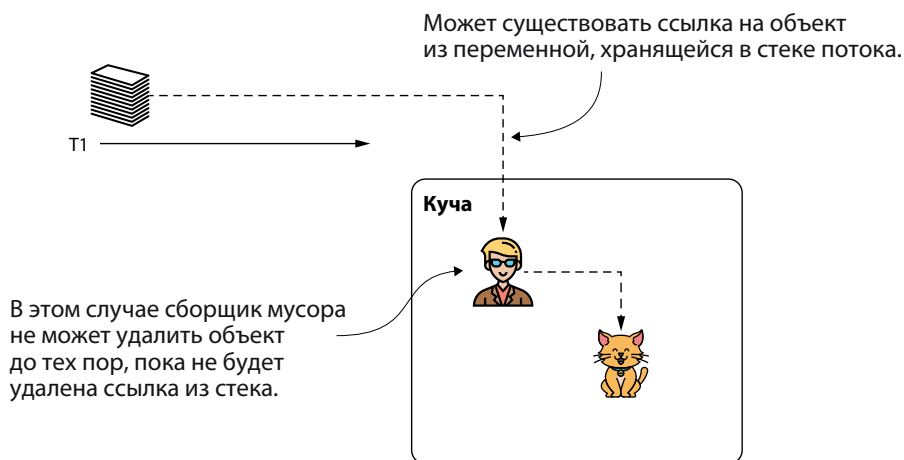
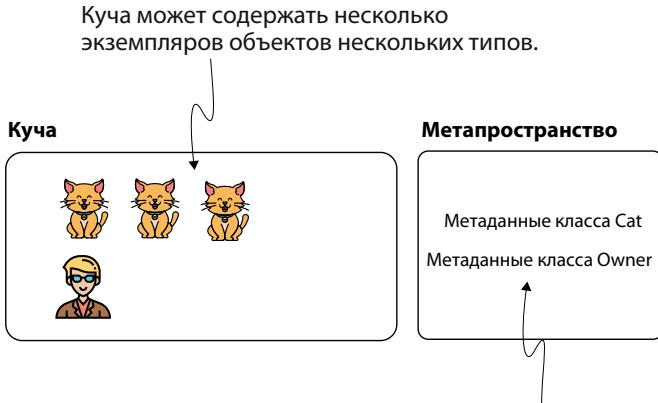


Рис. Е.13. Переменная в стеке также может ссылаться на экземпляр в куче, который невозможно удалить, пока все ссылки на него (включая ссылки из стека) не будут удалены

Е.4. Метапространство – локация памяти для хранения типов данных

Метапространство – это локация памяти, которую JVM использует для хранения типов данных, необходимых для создания экземпляров, хранящихся в куче (см. рис. Е.14). Приложению нужна эта информация для обработки экземпляров объектов в куче. Иногда при определенных условиях исключение `OutOfMemoryError` может воздействовать и на метапространство. Если метапространство заполнено до отказа и больше нет памяти для того, чтобы приложение сохранило новые типы данных, то приложение генерирует исключение `OutOfMemoryError`, сигнализирующее о том, что метапространство переполнено. По своему опыту отмечу, что такие ошибки весьма редки, тем не менее я должен предупредить вас о возможности их возникновения.



Метaproстранство содержит информацию, которая сообщает JVM, как интерпретировать и создавать экземпляры, обнаруженные в куче.

Рис. Е.14. Метaproстранство – это локация памяти, в которой приложение хранит дескрипторы типов данных. В нем содержится вспомогательная информация, используемая для определения экземпляров, хранящихся в куче

Предметный указатель

Символы

- _Consumer, поток 217, 225
- _Hlk139972743Дамп потоков_Hlk139972743
чтение
в виде простого текста 249
- _Producer, поток 217, 225
дамп потоков, пример 250
- agentlib:jdwp, параметр для подключения агента
отладчика 94
- Xms, ключ JVM, минимальный начальный размер
кучи 160
- Xmx, ключ JVM, максимальный размер кучи 160
- XX:MaxMetaspaceSize, ключ JVM, размер
метапространства 161

А

- Анализ
кода 22
после аварии 26
проблемы с производительностью 33
- Аспект (в Spring) 52
изменение выполнения кода 55

Б

- Блокировка 216.
в многопоточной архитектуре 216
каскадная 255
потока 216
анализ 222
монитор 218

В

- Взаимоблокировка 240, 333
дамп потоков 241
динамическая (активная) 334
- Выборка выполняемого кода 164
- Выходные данные 27

некорректные 31

Г

- Гейзенбаг 29
- Гейзенберга
выполнение 29
- Гейзенберг, Вернер (Heisenberg, Werner) 30
- Генерация 268
- Голодание (зависание) 335
- Граф вызовов 206

Д

- Дамп кучи
использование консоли OQL для запроса 279
конфигурирование приложения 270
поиск утечек памяти 268
получение в командной строке 274
чтение 275
VisualVM 275
- Дамп потоков 240
время использования ЦП 249
генерация
из командной строки 245, 247
с использованием профилировщика 243
- идентификатор ID внутреннего системного
потока 249
- идентификатор ID потока 249
- импортирование в профилировщик 247
- имя потока 249
- имя состояния 249
- общее время выполнения 249
- описание состояния 249
- приоритет потока 249
- сравнение с профилированием
блокировки 251
- трассировка стека 249
- чтение 249
fastThread 256
с использованием инструментальных
средств 256

Динамический прокси 172

Ж

Журнал 108

- анализ проблем в многопоточной архитектуре 117
- анализ проблемы 111
- время выполнения конкретной инструкции 116
- идентификация исключений 112
- консоль 108
- метка времени 112
- методика реализации функций журналирования в приложениях 119
 - appender (аппендер) 124
 - formatter (форматтер) 124
 - Log4j 123
 - logger (диспетчер журналирования) 124
 - постоянное хранение 119
- мониторинг 298
- проблема безопасности и защиты информации 129
- проблема производительности 129
- проблема удобства сопровождения 129
- сообщение 109
- сравнение с удаленной отладкой 134
- степень важности (severity) 121
- трассировка стека исключений 113
- уровень журналирования 121
 - debug (отладка) 121
 - error (ошибка) 121
 - fatal (неисправимая ошибка) 122
 - info (информация) 121
 - trace (трассировка) 122
 - warn (предупреждение) 121

Журнальное сообщение 109

З

Заглушка 31

Запрос с критерием поиска 186

Зеркалирование 306

И

Идентификатор процесса (PID) 246

Инструментальное средство мониторинга журналов 298

Инъекция критической ошибки 305

К

Код

чтение 42

анализ внутренней логики 43

Командная строка

генерация дампа потоков 245

поиск идентификатора ID процесса 245

Куча 346

М

Метапространство 349

Метапространство JVM 160

Микросервис 290

Многопоточная архитектура 29

синхронизация 218

Модель производитель–потребитель (producer-consumer) 145

Монитор

listA 241

listB 241

О

Обмен данными между сервисами (приложениями) 289

проба HTTP-клиента 293

проба HTTP-сервера 291

Отладка

уточнение термина 22

Отладчик 25, 41

анализ кода 45

выявление проблем с неожиданными выходными данными 27

изменение значения переменной во время отладки 75

освобождение фреймов выполнения 79

отбрасывание фреймов 79

выполнения 79, 81, 83

нерекомендуемые случаи 83

перемещение по коду 56

шаг с входом (step into) 57, 61

шаг с выходом (step out) 57, 62, 80

шаг с обходом (step over) 57, 58

повторное воспроизведение процесса анализа 79

точка останова 46

для записи в журнал сообщений 73

- условная 68
- трассировка стека выполнения 51
- удаленная отладка 89
 - агент 92
 - анализ в удаленной рабочей среде 92
 - параметр конфигурации 94
 - соответствие версий 100
- П**
- Память
 - куча 35
 - дамп 35
- Переменная
 - изменение значения в текущей области видимости во время отладки 75
- Перехватчик (в Java/Jakarta EE) 52
- Поток 318
 - взаимоблокировка 333
 - динамическая (активная) 334
 - голодание 230
 - голодание (зависание) 336
 - дамп 37
 - жизненный цикл 320
 - каскадно заблокированный 255
 - ожидающий 231
 - синхронизация 147, 151, 323
 - wait(), метод 326
 - блок кода 324
 - метода 324
 - присоединение (joining) 328
 - состояние гонки 332
- Поток-зомби 34, 145
- Приложение
 - аварийное завершение 35
 - удаленная отладка 88
- Проблема
 - производительность 33, 64
- Проблема N+1 запросов 184
- Проблема аномального потребления ресурсов 139
- Производственная среда (prod) 89
- Профилировщик 29, 138, 312
 - flame-график 209
 - JProfiler 198
 - дерево вызовов (call tree) 206
 - инструментовка 199
 - использование цвета для выделения уровней flame-графика 210
- VisualVM 139
 - наблюдение за выполнением потоков 150
 - наблюдение за потреблением ресурсов 154
 - установка и конфигурирование 142
- анализ SQL-запросов, генерируемых фреймворком 182
- анализ запросов в NoSQL базу данных 212
- выборка выполняемого кода 164, 166, 167
- выборка выполняющегося кода 141
- выбор пакетов и классов, подлежащих профилированию 174
- выявление проблем в JDBC-соединениях 193
- граф вызовов 206
- дамп потоков 243
- идентификация SQL-запросов 176
- извлечение программно определенных SQL-запросов 186
- импортирование дампа потоков 247
- инструментовка (instrumentation) 166
- количество вызовов метода 172
- методика анализа, выполняемая посредством профилирования 163
- наблюдение за использованием системных ресурсов 145
- обнаружение утечек памяти 156
- перехват SQL-запросов через драйвер JDBC 179, 187
- проблема потребления ресурсов 139
- профилирование выполняемого кода 166
- Р**
- Развертывание приложений 305
 - инъекция критической ошибки 307
- С**
- Сборщик мусора 141, 149, 262
- Сервис
 - обмен данными
 - сокет 295
- Синхронизация
 - блока 324
 - метода 324
- Синхронизированный блок кода 324
- Следующая строка выполнения 63

Сокет 295

 заккрытие 297

 открытие 296

 чтение/запись данных 296

Состояние гонки 147, 332

Среда пользовательского теста приемки (UAT) 89

Среда разработки (dev) 89

Стек 341

Т

Точка останова 47

 для записи в журнал сообщений 73

 условная 68

Трассировка стека выполнения 51

У

Удаленная отладка

 создание конфигурации в IDE Eclipse 104

Удаленная отладка приложения 89

Управление памятью

 куча 346

 метапространство 349

 стек 340

Утечка памяти 140, 141, 145

А

ArrayList, тип списка 217

 монитор 224

В

Breakpoint 47

 conditional 68

С

Cascading lock 255

Criteria query 186

CyclicBarrier, объект 331

Д

Deadlock 240

Debugger 41

Dropping execution frames 79

Dropping frames 79

Dynamic proxy 172

Ф

fastThread, веб-инструмент для чтения

 дампов потоков 256

Fault injection 305

Flame-график 209

Г

Garbage collector, GC 141

General Data Protection Regulation (GDPR),

 правила Евросоюза 131

Generation 268

Н

Heap 35

 dump 35

Heisenberg execution 29

Heisenbug 29

HotSpotDiagnosticMXBean, специальный

 класс 270

httpbin.org, конечная точка 165

И

IDE 312

Ж

Java Object Oriented Querying (JOOQ) 182

Java Persistence API, JPA 182

Java Persistence Query Language (JPQL) 183

JDBC-соединение

 ограничение количества со стороны

 СУБД 194

JDK 312

jmap, инструмент командной строки 274

join(long timeout), метод 330

join(), метод 330

join(), метод потока 323

jps, инструментальное средство JDK 246

jstack, инструментальное средство JDK 247

Л

Latch, объект 331

LIFO (last in, first out – последним вошел, первым
 вышел) 341

Lock, объект 331

Log-monitoring tool 298

M

Memory leak 141, 145

Metaspace 160

Mirroring 306

N

N+1 query problem 184

NoSQL база данных

анализ запросов 212

notifyAll(), метод 234, 326

notify(), метод 326

NullPointerException, исключение 113

O

OpenFeign, проект из экосистемы Spring 171

OQL (Object Query Language) 280

запрос 280

referrers(), метод 284

формат JSON 282

консоль 279

OutOfMemoryError, исключение 157

OutOfMemoryError, критическая ошибка 140

P

park(), метод 322

Profiler 138

Q

Quitting execution frames 79

R

Race condition 147

referrers(), встроенный метод OQL 284

Remote debugging 88

agent 92

S

Sampling 164

Semaphore, объект 331

sentry.dsn, свойство Sentry 301

Sentry, инструмент мониторинга

журналов 298

Service mesh 305

приложение-корзина

зеркалирование 309

инъекция критической ошибки 308

sleep(), метод 329

sleep(), метод класса Thread 322

SQL-запрос 176

перекрестное соединение (cross join) 187

Stack 341

start(), метод потока 322

Starvation 230

Stub 31

synchronized, ключевое слово 324

T

Thread 318

dump 37

Thread dump 240

Thread lock 216

Thread, класс 329

Thread, класс потока 322

U

unpark(), метод 322

V

VisualVM, профилировщик 262

дамп потоков 244

мониторинг потоков 220

W

wait(long timeout), метод 330

wait(), метод 233, 236, 326

wait(), метод монитора 322

Wrong output 31

Z

Zombie thread 34, 145

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javalib>

Лауренциу Спилкэ

Java: устранение проблем

Чтение, отладка и оптимизация JVM-приложений

