Бикрамадитья Сингхал
Гаутам Дамеджа
Прияну Сехар Панда

# Блокчейн

## Руководство для начинающих разработчиков

bhv®

Apress®

# Beginning Blockchain

## A Beginner's Guide to Building Blockchain Solutions

**Bikramaditya Singhal**
**Gautam Dhameja**
**Priyansu Sekhar Panda**

Apress®

# Блокчейн

## Руководство для начинающих разработчиков

**Бикрамадитья Сингхал**
**Гаутам Дамеджа**
**Приянсу Сехар Панда**

Санкт-Петербург
«БХВ-Петербург»
2020

УДК 004.75+519.83+336.7
ББК 32.973.26-018
С38

**Сингхал, Б.**

С38 Блокчейн. Руководство для начинающих разработчиков: Пер. с англ. / Б. Сингхал, Г. Дамеджа, П. С. Панда. — СПб.: БХВ-Петербург, 2020. — 288 с.: ил.

ISBN 978-5-9775-4052-0

Книга предназначена для изучения фундаментальных основ блокчейна и решения прикладных задач. С нуля изложены основы криптографии, устройство блокчейна и его основные компоненты: математика, криптография, теория игр. Изложены технические основы самых известных блокчейнов в мире — Bitcoin и Ethereum. Продемонстрировано, как можно запрограммировать блокчейн для разных вариантов использования, не ограничиваясь только криптовалютой. Рассмотрен процесс разработки кода для управления транзакциями на языках JavaScript и Solidity, показано, как самостоятельно создавать и размещать умные контракты. Продемонстрирован полный цикл разработки децентрализованного приложения (DApps).

*Для программистов, преподавателей и студентов,*
*а также специалистов отделов развития компаний и банков*

УДК 004.75+519.83+336.7
ББК 32.973.26-018

**Группа подготовки издания:**

| | |
|---|---|
| Руководитель проекта | *Евгений Рыбаков* |
| Зав. редакцией | *Екатерина Сависте* |
| Перевод с английского | *Валерия Яценкова* |
| Компьютерная верстка | *Ольги Сергиенко* |
| Оформление обложки | *Карины Соловьевой* |



Подписано в печать 05.08.19.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 23,22.
Тираж 1200 экз. Заказ № 9741.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

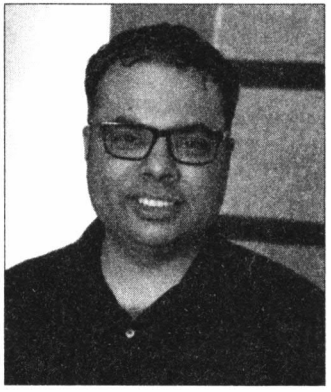
Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-1-4842-3443-3 (англ.)
ISBN 978-5-9775-4052-0 (рус.)

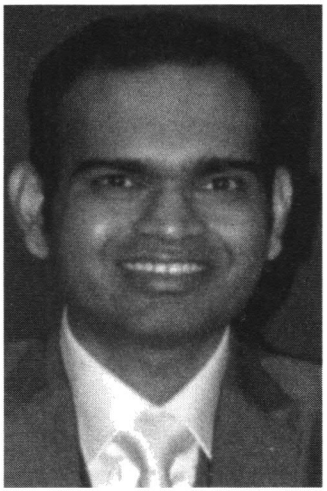
# Оглавление

# Об авторах

**Бикрамадитья Сингхал** (Bikramaditya Singhal) — эксперт по блокчейну и специалист по искусственному интеллекту с опытом работы в различных отраслях. Он обладает обширными знаниями в области криптографии, кибербезопасности и науки о данных, владеет навыками работы с блокчейнами Bitcoin, Ethereum и Hyperledger. Бикрамадитья Сингхал работал с такими компаниями, как WISeKey, Tech Mahindra, Microsoft India, Broadridge и Chelsio Communications, а также основал компанию под названием Mund Consulting, которая специализируется на анализе больших данных и искусственном интеллекте. Имеет большой опыт обучения и консультирования по технологии блокчейна, разработал множество блокчейн-решений. Активный докладчик на различных конференциях, встречах и семинарах. Является также автором книги под названием «Spark for Data Science».

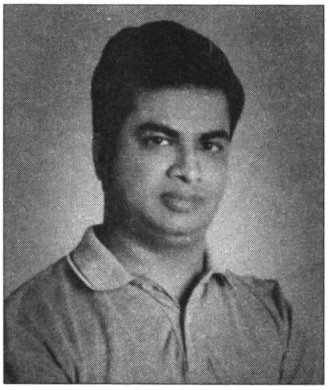


**Гаутам Дамеджа** (Gautam Dhameja) — консультант по блокчейн-приложениям из Берлина, Германия. В течение последнего десятилетия он занимался разработкой и поставкой корпоративного программного обеспечения, включая веб-приложения и мобильные приложения, облачные гипермасштабируемые решения Интернета вещей и, в последнее время, децентрализованные приложения на основе блокчейна (DApps). Он обладает глубоким пониманием децентрализованного стека, архитектуры облачных решений и объектно-ориентированного проектирования. Гаутам Дамеджа специализируется на блокчейне, облачных корпоративных решениях, Интернете вещей, распределенных системах, а также специальных и гибридных мобильных приложениях. Является активным блоггером и регулярно выступает на технических конференциях и мероприятиях.



**Приянсу Сехар Панда** (Priyansu Sekhar Panda) — инженер-исследователь компании Underwriters Laboratories, Бангалор, Индия. Он сотрудничал с другими ИТ-компаниями, такими как Broadridge, Infosys Limited и Tech Mahindra. Обладает навыками применения блокчейнов Bitcoin, Ethereum, Hyperledger, а также знаниями в области теории игр, Интернета вещей и искусственного интеллекта. Текущие исследования автора направлены на создание приложений нового поколения, использующих блокчейн, Интернет вещей и искусственный интеллект. Приянсу Сехар Панда трудится над созданием децентрализованных автономных организаций (DAO), а также изучает безопасность, масштабируемость и механизмы консенсуса блокчейнов.

# О техническом рецензенте



**Навин Манасви** (Navin K Manaswi) уже много лет разрабатывает решения и приложения с использованием передовых технологий на основе искусственного интеллекта. Работая в консалтинговых компаниях в Малайзии, Сингапуре и в проекте Dubai Smart City, он приобрел редкий навык разработки комплексных проектов в области искусственного интеллекта. Навин Манасви создал решения для видеосвязи, документооборота и корпоративных чат-роботов. В настоящее время решает проблемы делового взаимодействия в сфере здравоохранения, бизнеса, промышленного Интернета вещей и розничной торговли в инкубаторе Symphony AI, где работает архитектором систем глубокого машинного обучения. С помощью этой книги он хочет как можно шире популяризовать децентрализованные вычисления и сервисы, особенно среди разработчиков программного обеспечения, специалистов по обработке данных, инженеров баз данных, бизнес-аналитиков и руководителей компаний.

# Благодарности

Авторы хотели бы поблагодарить Никхила (Nikhil) и Дивью (Divya) за их сотрудничество и поддержку на всем протяжении работы над книгой. Большое спасибо Навину Манасви за его тщательный технический анализ этой книги. Мы также благодарим всех, кто прямо или косвенно внес в эту книгу свой вклад.

# Предисловие

«Блокчейн с нуля» — это книга для тех, кто хочет изучить технические основы блокчейна и вопросы разработки и применения прикладных приложений для работы с блокчейном. Краткое, но тщательно продуманное изложение истории платежных систем и основ криптографии поможет погрузиться в изучение технологии блокчейна, а примеры прикладного кода помогут читателю быстро приступить к разработке приложений.

*Глава 1* знакомит вас с историей платежных систем и происхождением технологии блокчейна. В *главе 2* детально рассмотрены основные компоненты блокчейна: математика, криптография, теория игр. *Глава 3* посвящена техническим основам самого известного блокчейна в мире — Bitcoin (Биткойн) и содержит примеры использования этой криптовалюты. *Глава 4* посвящена блокчейн-платформе Ethereum (Эфириум) и демонстрирует, как можно запрограммировать блокчейн для разных вариантов использования, не ограничиваясь только криптовалютой. В *главе 5* вы познакомитесь с разработкой кода для управления транзакциями на языках JavaScript и Solidity и научитесь создавать и размещать умные контракты. *Глава 6* завершает книгу и демонстрирует полный цикл разработки децентрализованного приложения (Decentralized Applications, DApps). К концу этой главы вы овладеете достаточным количеством инструментов и методов для решения различных прикладных задач с помощью технологии блокчейна. Сделайте первый шаг на пути к безграничным возможностям!

ГЛАВА 1

# Знакомство с блокчейном

Блокчейн — это очередная волна перемен, которая уже начала менять структуру деловых, социальных и политических связей, а также способы перемещения средств. С другой стороны, блокчейн — это не просто перемены, а некая сущность, которая никогда не стоит на месте. На момент подготовки этой книги более 40 ведущих финансовых учреждений и множество фирм в различных отраслях начали осваивать блокчейн — чтобы снизить транзакционные издержки, ускорить прохождение транзакций, снизить риск мошенничества и устранить посредников. Некоторые фирмы пытаются с его помощью перестроить устаревшие системы и сервисы, чтобы вывести их на следующий уровень, а также предложить новые виды услуг.

Мы будем детально исследовать блокчейн на протяжении всей книги. Если вы новичок, то можете последовательно изучать главу за главой или выбрать только те главы, которые вам нужнее. Эта глава расскажет о том, что такое блокчейн, как он развивался, где применяется и почему так важен в современном мире. Вы получите из нее общее представление о блокчейне, которое поможет вам глубже погрузиться в его изучение.

## 1.1. Происхождение блокчейна

Одним из первых переломных моментов цифровой истории стало появление в 1970-х годах протокола TCP/IP[1], на котором основан современный Интернет. До появления TCP/IP мы жили в эпоху коммутируемых каналов, которые нуждались в прямом физическом соединении между двумя устройствами.

---

[1] Transmission Control Protocol/Internet Protocol — протокол управления передачей/протокол сети Интернет (*англ.*). — *Здесь и далее примечания от редакции русского перевода.*

Когда в начале 1990-х годов Интернет явился общественности в виде «сетевой паутины» World Wide Web (WWW), ему пришлось обеспечивать связь всех со всеми. Это связано с тем, что Интернет построен поверх открытого и децентрализованного протокола TCP/IP. Когда какие-либо новые технологии, особенно революционные, попадают на рынок, они либо умирают сами по себе, либо приобретают такое влияние, что становятся общепринятой нормой. Общество приспособилось к сетевой революции и по-своему воспользовалось возможностями, которые она предлагала. В результате сеть сформировалась, пожалуй, не совсем в том виде, как это было задумано. Она могла бы стать более открытой, доступной и равноправной. Однако многие новые технологии начали накладываться на существующие структуры, и к сегодняшнему дню Интернет стал таким, каков он есть, — более централизованным. Люди склонны привыкать к ограничениям технологии. Нынче они вполне довольны, если международный перевод средств занимает несколько дней[2], или обходится слишком дорого, или недостаточно надежен.

Давайте подробнее рассмотрим банковскую систему и ее эволюцию. Начиная с первобытной меновой системы и вплоть до фиатных валют[3], между сделкой и ее подтверждением не было никакой реальной разницы, поскольку они не были двумя отдельными действиями. Например, если Алисе нужно заплатить 10 долларов Бобу, она просто передает Бобу банкноту номиналом 10 долларов. На этом сделка полностью завершена. Банку не нужно было списывать 10 долларов со счета Алисы и записывать на счет Боба или служить поручителем, чтобы Алиса не обманула Боба. Однако прямое взаимодействие с каждым человеком весьма затруднительно. Поэтому в банковской сфере появилось множество услуг, включая денежные переводы из любого уголка мира. Появление Интернета сломало последние преграды, и банковское дело стало проще, чем когда-либо. Но не только банковское дело — Интернет облегчил и другие способы перемещения ценности через сетевую паутину.

Традиционная технология позволяет кому-либо из Индии совершить денежную сделку с кем-либо в Соединенном Королевстве, но с некоторыми издержками. Для урегулирования таких транзакций требуются дни, и вдобавок они дорого обходятся. Банку всегда необходимо гарантировать доверие и обеспечить безопасность для таких сделок между двумя или более сторонами. А что, если найдется технология, которая может обеспечить доверие и безопасность без этих посредников и централизованных систем? По какой-то причине эта часть технологии — обеспечение доверия — отстала в развитии, и в результате расплодились централизованные системы, такие как банки, службы условного депонирования, клиринговые палаты, регистраторы и многие другие подобные учреждения. Блокчейн стал той недостающей частью интернет-революции, которая превращает уязвимую систему обмена ценностями в криптографически защищенную крепость.

---

[2] Даже если адресат получил средства через несколько секунд, в традиционной банковской системе взаимные расчеты банков закрываются через клиринговую палату в срок от нескольких дней до месяца.

[3] Фиатная валюта — валюта, которую правительство позиционирует как единственное законное платежное средство.

Тот, кто ныне скрывается под всемирно известным псевдонимом Сатоши Накамото, прекрасно понимал, что банковская система образца 1980-х годов отстала от технологической революции. Банки создали централизованные организации, которые хранят транзакционные записи, контролируют взаимодействие, обеспечивают доверие и безопасность и регулируют всю систему. Вся коммерция опирается на эти финансовые учреждения, которые служат доверенными посредниками при обработке платежей. Посредничество финансовых учреждений увеличивает затраты и время на прохождение транзакции, а также ограничивает размеры транзакций. Посредники необходимы для разрешения споров, но по сути это означает, что совершенно необратимая транзакция невозможна — ведь посредник может ее отменить. Это следует из ситуации, когда для совершения сделки с контрагентом требуется доверенный посредник. Разумеется, эта бюрократическая система рано или поздно должна измениться, чтобы идти в ногу с наступающей цифровой трансформацией экономики. Итак, Сатоши изобрел криптовалюту под названием биткойн, в основу которой заложен блокчейн. Биткойн — это всего лишь частный случай использования блокчейна, который учитывает внутреннюю уязвимость моделей, основанных на доверии. В этой книге мы рассмотрим фундаментальные основы как биткойна, так и блокчейна.

## 1.2. Что такое блокчейн?

Интернет радикально изменил многие аспекты жизни, общества и бизнеса. Однако в предыдущем разделе мы отмечали, что способы проведения транзакций между людьми и организациями за последние несколько десятилетий не сильно изменились. Блокчейн, как мы уже говорили, ставит все на свои места и делает систему транзакций более открытой, доступной и надежной.

Чтобы понять сущность блокчейна, вы должны посмотреть на него как с точки зрения бизнеса, так и с технической точки зрения. Давайте сначала рассмотрим блокчейн в контексте бизнес-транзакций, чтобы понять *что* он дает, а в следующих главах углубимся в техническую составляющую, чтобы понять *как* он это делает.

Итак, блокчейн — это система записей о переносе любой ценности (а не только денег!) по принципу «от равного к равному» (peer-to-peer). Это означает, что нет необходимости в посредниках, таких как банки, брокеры или другие службы депонирования, которые служат доверенной третьей стороной. Например, если Алиса заплатит Бобу 10 долларов, почему они обязательно должны проходить через банк? Взгляните на рис. 1.1.

Давайте рассмотрим другой пример. Типичная операция с акциями происходит за доли секунды, но сведение балансов через клиринговую палату длится недели. Приемлемо ли это в цифровую эпоху? Конечно же, нет! На рис. 1.2 показана текущая ситуация.

Но если кто-то хочет купить акции у компании или у человека, они могут, используя блокчейн, купить их напрямую и с немедленной регистрацией сделки, без

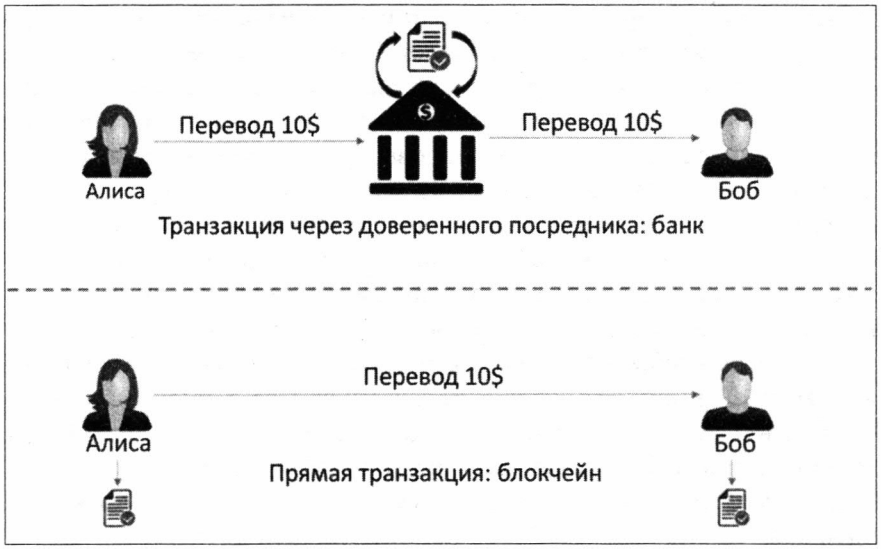



**Рис. 1.1. Транзакция через посредника и прямая транзакция**

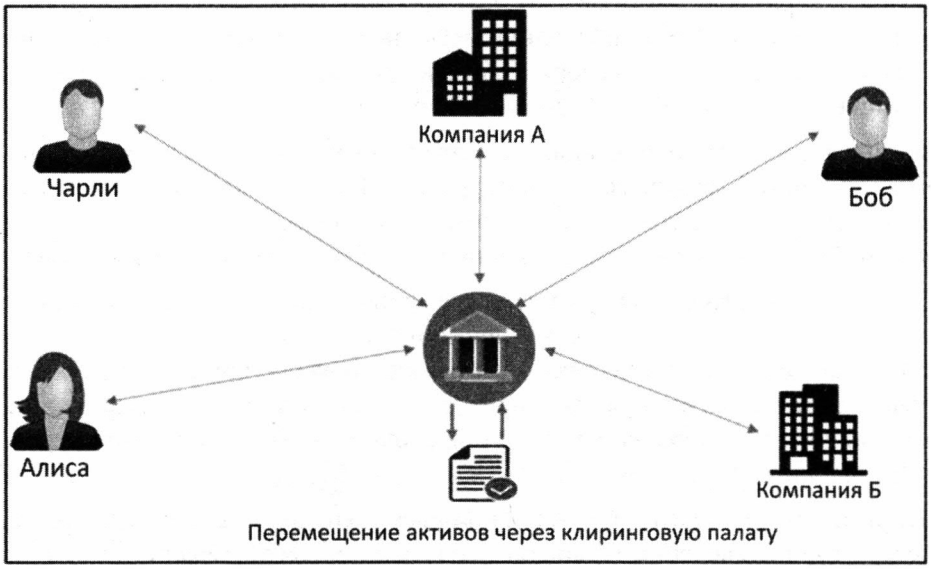



**Рис. 1.2. Биржевая торговля через клиринговую палату**

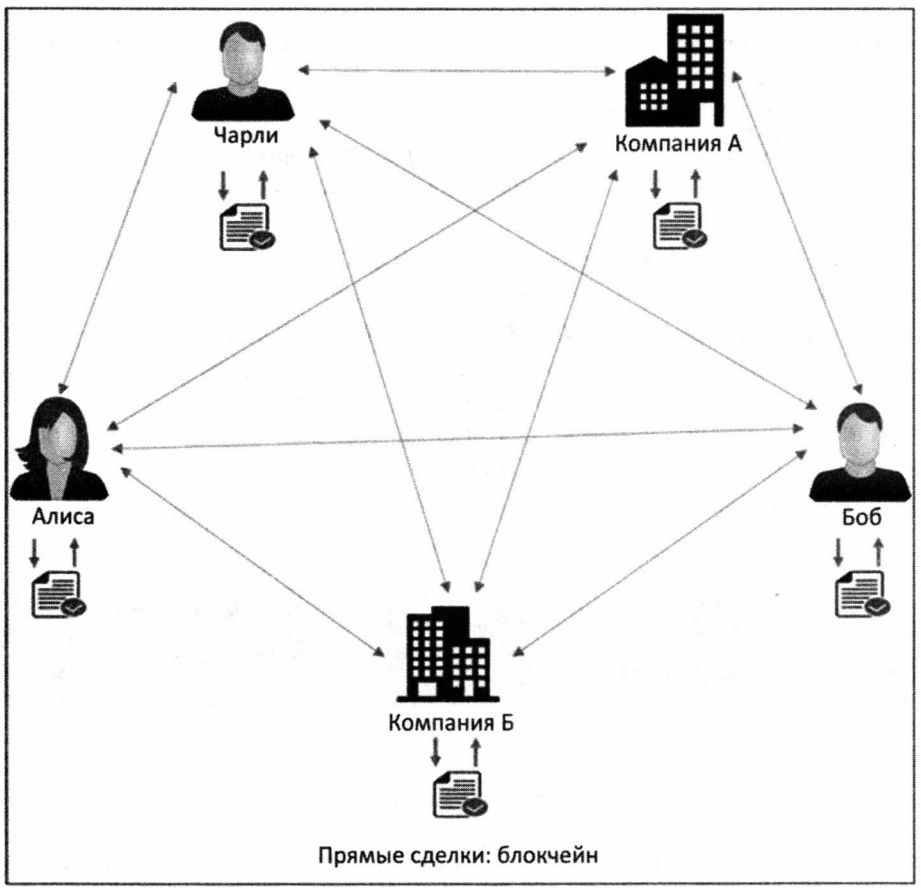



**Рис. 1.3. Сделки в одноранговой сети**

необходимости участия брокеров, клиринговых палат или других финансовых учреждений. Децентрализованное и одноранговое решение такой задачи может выглядеть, как показано на рис. 1.3.

Обратите внимание, что транзакция и подтверждение сделки не являются двумя разными сущностями в среде блокчейна! Транзакции здесь аналогичны денежным сделкам, где, к примеру, кто-то платит банкнотой номиналом 10 долларов и больше не владеет ею, а банкнота физически передается новому владельцу.

Теперь, когда вы поняли функциональную суть блокчейна на верхнем уровне, давайте взглянем на его устройство с технической точки зрения, и нам станет понятно, почему блокчейн называют именно этим словом. Сейчас мы увидим, *что* это такое, а изучение того, *как* это работает, оставим для *главы 2*.

- Блокчейн — это одноранговая система передачи ценности без участия доверенной третьей стороны.
- Это общий, децентрализованный и открытый реестр транзакций. База данных реестра реплицируется (копируется) на большое количество узлов.

- База данных реестра работает только в режиме добавления записей и не может быть изменена или исправлена. Это означает, что каждая запись является постоянной и неизменной. Любая новая запись появляется во всех копиях базы данных, размещенных на разных узлах.
- Нет необходимости, чтобы доверенные третьи стороны выступали в качестве посредников для проверки, обеспечения безопасности и подтверждения транзакций.
- Блокчейн — это еще один функциональный слой поверх Интернета, и он может сосуществовать с другими интернет-технологиями.
- Точно так же, как протокол TCP/IP был разработан для создания открытой системы обмена данными, технология блокчейна была разработана для подлинной децентрализации обмена ценностями. Руководствуясь этой идеей, создатели биткойна открыли исходный код, чтобы на него могли опираться разработчики других децентрализованных приложений.

Типичный блокчейн в общем виде выглядит так, как показано на рис. 1.4.

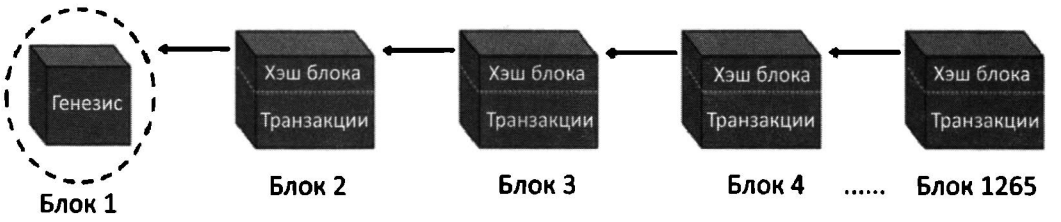



**Рис. 1.4.** Структура данных блокчейна

Каждый узел в сети имеет идентичную копию блокчейна, условно показанную на рис. 1.4, где каждый блок представляет собой совокупность транзакций и связь с предыдущим блоком — отсюда и происходит название[4] технологии. Как вы можете видеть, каждый блок состоит из двух частей: *заголовка* и *тела* блока. Заголовок ссылается на предыдущий блок в цепочке. Каждый заголовок блока содержит хэш предыдущего блока, поэтому никто не может незаметно изменить транзакцию в предыдущем блоке (подробности этой концепции мы рассмотрим в следующих главах). Тело блока содержит список проверенных транзакций, их суммы, адреса сторон и некоторые другие подробности. Таким образом, имея последний блок, можно получить последовательный доступ ко всем предыдущим блокам в цепочке блоков.

Чтобы понять, как работает эта система, рассмотрим практический пример и проследим, как происходят транзакции и обновляется реестр.

Предположим, что есть три участника: Алиса, Боб и Чарли, которые проводят некоторые денежные транзакции между собой в сети блокчейна. Давайте проследуем по пути транзакций шаг за шагом, чтобы понять, как работают открытые и децентрализованные функции блокчейна.

---

[4] Block — блок, chain — цепь (*англ.*).

## Шаг 1

Допустим, что у Алисы в кошельке было 50 долларов, что является *генезисом* (начальной точкой) всех транзакций, и каждый узел сети знает об этом (рис. 1.5).

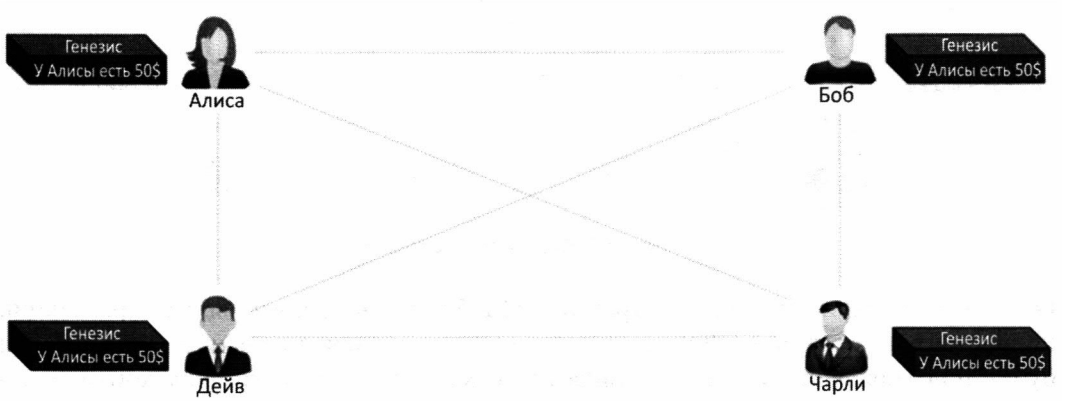



**Рис. 1.5. Генезис — начальный блок в цепочке транзакций**

## Шаг 2

Алиса совершает сделку, заплатив 20 долларов Бобу. Обратите внимание, что блокчейн обновился на каждом узле (рис. 1.6).

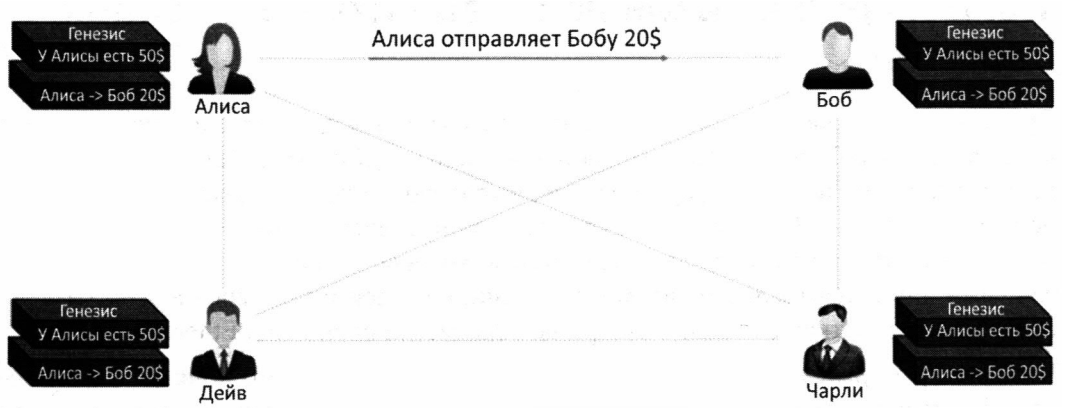



**Рис. 1.6. Первая транзакция**

## Шаг 3

Боб совершает другую сделку, заплатив 10 долларов Чарли, и блокчейн снова обновляется (рис. 1.7).


**Рис. 1.7. Вторая транзакция**

Обратите внимание, что данные транзакций в блоках неизменяемы. Все транзакции полностью необратимы. Любое изменение породит новую транзакцию, которая будет подтверждена всеми участниками. У каждого узла есть своя копия блокчейна.

Если сейчас у вас появились разные вопросы, например: «Что, если Алиса одновременно выплатит Дейву такую же сумму, как и Бобу (двойное списание), или что, если она запустит транзакцию, не имея достаточного количества средств на своем счете?», «Как обеспечивается безопасность?» — это замечательно! Мы рассмотрим эти вопросы в следующих главах.

## 1.3. Централизованные и децентрализованные системы

Причина, по которой мы обсуждаем централизацию и децентрализацию, состоит лишь в том, что блокчейн создан для децентрализации и отвергает централизованный подход. Тем не менее, термины «децентрализованный» и «централизованный» не всегда понятны. Зачастую они очень плохо определены и вводят в заблуждение. Дело в том, что почти не существует систем, которые являются чисто централизованными или децентрализованными. Большинство идей и примеров в этом разделе основано на заметках Виталика Бутерина, основателя блокчейна Ethereum.

Что такое *распределенная система*? Чтобы не примешивать это понятие к текущему обсуждению, давайте сначала разберемся с ним и отложим в сторону. Дело в том, что независимо от того, централизована или децентрализована система, ее все равно можно распределить. *Централизованная распределенная система* — это такая система, в которой есть главный узел, ответственный за дробление задач или данных и распределение нагрузки между узлами. Напротив, *децентрализованная распределенная система* — это такая система, где нет главного узла как такового, но все же вычисления могут быть распределены. Блокчейн — один из таких примеров, и позже мы рассмотрим различные графические представления блокчейна.

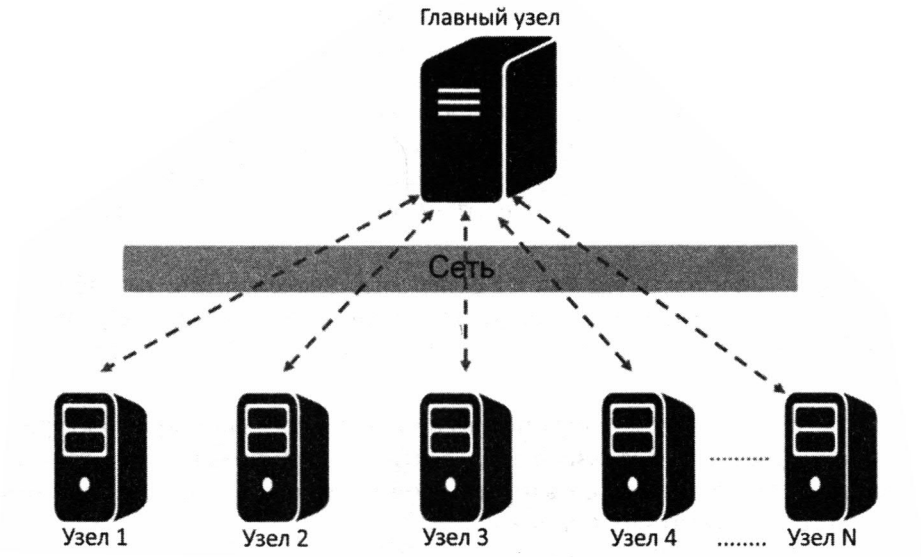



**Рис. 1.8. Пример распределенной системы с централизованным управлением**

Пример того, как может выглядеть централизованная распределенная система, показан на рис. 1.8.

Например, это представление соответствует тому, как реализована сеть распределенных вычислений Hadoop. Хотя вычисление в таких проектах происходит быстрее благодаря распределению, оно также страдает от ограничений из-за централизации.

Вернемся к обсуждению вопросов централизации и децентрализации. Крайне важно отметить, что централизация или децентрализация системы определяется не только технической архитектурой. Мы хотим подчеркнуть, что система может быть централизованной или децентрализованной с *технической* точки зрения, но *логически* или *политически* может быть устроена совершенно иначе. Давайте рассмотрим эти аспекты архитектур, чтобы иметь возможность правильно спроектировать систему, исходя из потребностей пользователей.

- **Технический аспект.** Система может быть централизованной или децентрализованной с точки зрения технической архитектуры. Здесь мы анализируем, сколько физических компьютеров (или узлов) используется для создания системы, количество отказов узлов, которые она может выдержать до того, как вся система рухнет, и т. д.
- **Политический аспект.** Здесь мы анализируем контроль, который человек, группа людей или организация имеют над системой в целом. Если все компьютеры системы контролируются узким кругом лиц, то система совершенно очевидно централизована. Однако если ни один конкретный индивид или группа не контролируют систему, и у всех пользователей есть равные на нее права, то в политическом смысле это децентрализованная система.

- **Логический аспект.** Система может быть логически централизована или децентрализована, исходя из ее устройства, — вне зависимости от того, централизована она или децентрализована технически или политически. Это можно пояснить таким примером: представьте, что вы вертикально разрезаете систему пополам, причем каждая половина имеет своих поставщиков услуг и потребителей. Если обе половины могут работать как независимые единицы, значит, они логически децентрализованы. В противном случае это логически централизованная система.

Все вышеупомянутые подходы имеют решающее значение при разработке реальной системы и определении ее как централизованной или децентрализованной. Давайте обсудим некоторые из примеров смешанного подхода, чтобы устранить любую путаницу:

- если вы рассматриваете корпорации, то они централизованы архитектурно (один головной офис), централизованы политически (управляются генеральным директором или советом директоров) и они также централизованы логически (вы не можете разрезать их пополам);
- наш язык общения децентрализован со всех точек зрения — как в архитектурном, так и в политическом плане, а также логически. Когда общаются два человека, их язык не обусловлен политически, а также логически не связан с языком общения других людей;
- системы торрентов — такие как BitTorrent — также децентрализованы со всех точек зрения. Любой узел может быть поставщиком или потребителем, поэтому, даже если вы разрезаете систему на половинки, она по-прежнему функционирует;
- с другой стороны, сеть доставки контента является децентрализованной по архитектуре, а также децентрализованной логически, но политически она централизована, поскольку принадлежит корпорациям. Примером может служить Amazon CloudFront;
- теперь рассмотрим блокчейн. Назначение блокчейна заключается в том, чтобы обеспечить децентрализацию. Действительно, он децентрализован технически. Кроме того, он децентрализован с политической точки зрения, поскольку его никто не контролирует. Однако блокчейн централизован логически, т. к. существует единственное общее согласованное состояние, и вся система ведет себя как один глобальный компьютер.

Рассмотрим теперь понятия централизованных и децентрализованных систем по отдельности и сопоставим их, чтобы убедиться, что блокчейн действительно децентрализован по своему устройству.

## 1.3.1. Централизованные системы

Как следует из названия, централизованная система имеет централизованное управление со всеми административными полномочиями. Такие системы легко разрабатывать, поддерживать, навязывать им доверие и управлять ими, но они страдают от многих неотъемлемых ограничений, а именно:

- у них есть центральная точка отказа, поэтому они менее стабильны;
- они более уязвимы для атаки и, следовательно, менее защищены;
- централизация власти может привести к неэтичным действиям;
- масштабирование их в большинстве случаев затруднено.

Типичная централизованная система может выглядеть, как показано на рис. 1.9.

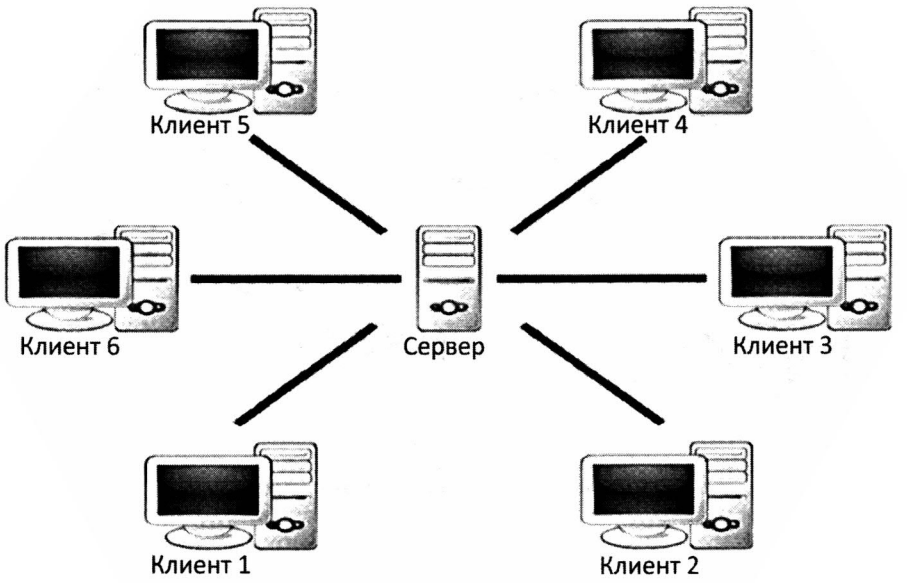



**Рис. 1.9. Централизованная система**

## 1.3.2. Децентрализованные системы

Как следует из названия, децентрализованная система не имеет центрального органа управления, и каждый ее узел обладает равными полномочиями. Такие системы сложно разрабатывать, поддерживать, управлять или навязывать им доверие. Однако они не страдают от ограничений обычных централизованных систем. Децентрализованные системы предлагают следующие преимущества:

- у них нет центральной точки отказа, поэтому они более стабильные и отказоустойчивые;
- устойчивость к атакам, т. к. они не имеют центральной точки, доступной для легкой атаки, и, следовательно, более защищены;
- представляют собой симметричную систему с равными полномочиями для всех, поэтому в ней меньше объем неэтичных операций, и она демократична по своей природе.

Типичная децентрализованная система может выглядеть, как показано на рис. 1.10.

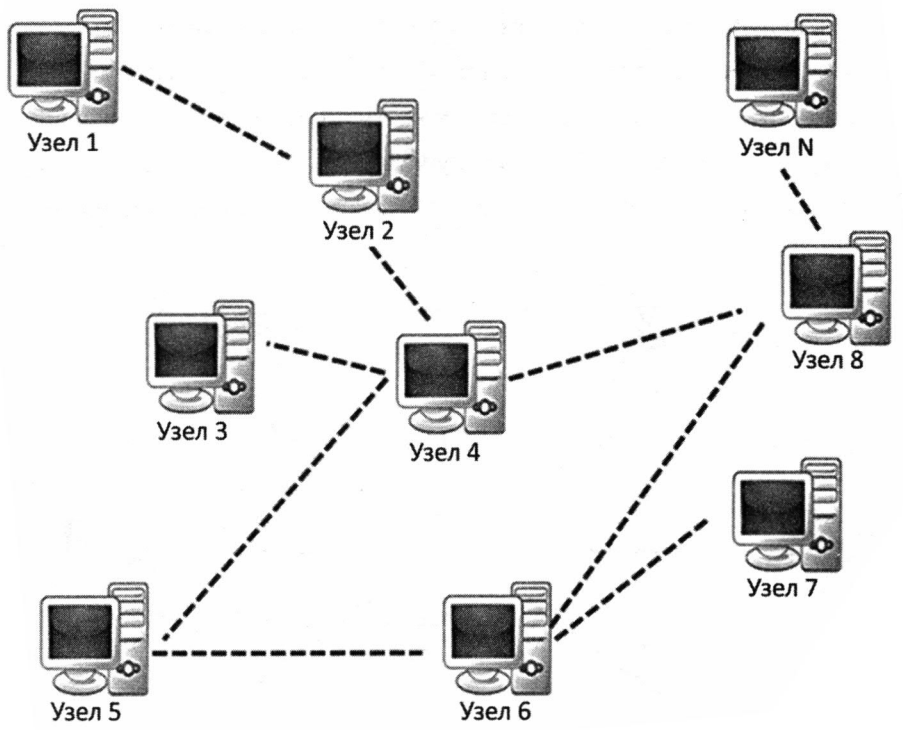



**Рис. 1.10.** Децентрализованная система

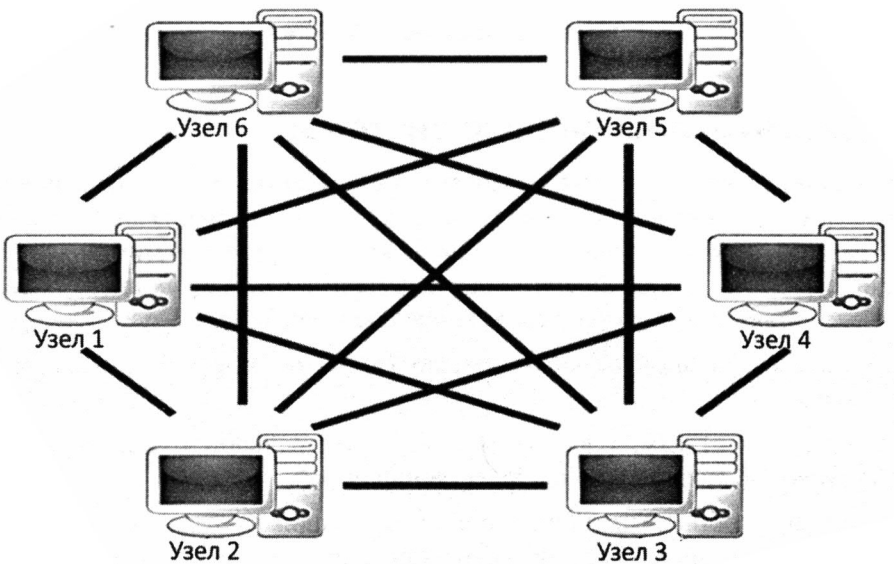



**Рис. 1.11.** Децентрализованная пиринговая система «каждый-с-каждым»

Обратите внимание, что распределенная система также может быть децентрализованной. Примером может служить блокчейн. Однако, в отличие от обычных распределенных систем, в блокчейне задача не разделяется и не делегируется узлам, т. к. в блокчейне нет руководителя, который делал бы это. Распределенные узлы не работают над частями вычислений. Напротив, заинтересованные или выбранные случайным образом узлы в одиночку выполняют всю работу. Типичная децентрализованная и распределенная система, которая, по сути, представляет собой пиринговую сеть, может выглядеть, как показано на рис. 1.11.

## 1.4. Уровни блокчейна

На момент подготовки этой книги публичные варианты блокчейна, такие как Ethereum, находятся в процессе развития, и создание сложных приложений поверх этих блокчейнов может быть не очень хорошей идеей. Имейте в виду, что блокчейн — это не просто технология, а сочетание принципов бизнеса, экономики, теории игр, криптографии и инженерных наук. Большинство прикладных приложений весьма сложны по своей природе, поэтому бывает целесообразно разрабатывать блокчейн-решения с нуля.

Цель этого раздела заключается только в том, чтобы взглянуть на различные уровни блокчейна с высоты птичьего полета, а углубляться в фундаментальные основы мы будем в следующих главах. Для начала давайте вспомним общее понятие стека протокола TCP/IP. Многоуровневый подход в стеке TCP/IP фактически является стандартом для построения открытой системы. Наличие уровней абстракции не только дает возможность лучше понять стек, но также помогает создавать продукты, совместимые со стеком. Кроме того, наличие отделенных друг от друга уровней делает систему более надежной и простой в обслуживании. Любое изменение любого из уровней не влияет на другие уровни. С другой стороны, пример TCP/IP не является полной аналогией блокчейна. TCP/IP — это протокол связи, которым пользуется каждое интернет-приложение, включая блокчейн.

Вернемся к блокчейну. Пока не существует согласованных глобальных стандартов, которые бы четко разделяли компоненты блокчейна на отдельные уровни. Нам хотелось бы иметь общепризнанную многоуровневую гетерогенную архитектуру, но пока это еще впереди. Итак, мы попытаемся дать свою формулировку уровней блокчейна, чтобы лучше понять технологию и увидеть связи между сотнями рыночных вариантов блокчейна и криптовалют. Посмотрите на обобщенное многоуровневое представление блокчейна (рис. 1.12).

Вы можете спросить, почему выделено именно пять уровней и почему не больше или меньше? Очевидно, что не должно быть слишком много или слишком мало уровней — нужен компромисс между сложностью, надежностью, адаптивностью и т. д. И снова наша цель состоит не в том, чтобы стандартизировать технологию блокчейна, а в том, чтобы добиться лучшего понимания. Не забывайте, что все эти уровни присущи всем узлам сети.


**Рис. 1.12.** Различные уровни блокчейна

В *главе 6* этой книги мы будем создавать децентрализованное приложение с нуля и узнаем, как блокчейн функционирует на всех уровнях при выполнении реальной работы.

## 1.4.1. Прикладной уровень (application layer)

Благодаря свойствам блокчейна — таким как неизменность данных, прозрачность для участников, устойчивость к атакам и пр., может быть создано несколько типов приложений. Некоторые приложения просто находятся на прикладном уровне (уровень конечного пользователя) и обслуживают любое применение блокчейна, а остальные приложения встроены в прикладной уровень и переплетаются с другими уровнями блокчейна. По этой причине *прикладной уровень* следует рассматривать как часть блокчейна.

Это уровень, на котором вы пишете код нужных функций и создаете из него приложение для конечных пользователей. Обычно он включает в себя традиционный технологический цикл разработки программного обеспечения — такой как программные конструкции на стороне клиента, сценарии, API, платформы разработки и т. п. Приложения, которые рассматривают сеть блокчейна как распределенную среду выполнения, могут быть размещены на веб-серверах, а для этого потребуется разработка веб-приложений, программирование на стороне сервера и API-интерфейсы. В идеале, хорошие приложения блокчейна не используют модель «клиент-сервер», и не существует централизованных серверов, к которым обращаются клиенты. Именно так работает Bitcoin[5].

Возможно, вы что-то слышали о *сетях вне блокчейна* (off-chain network). Идея состоит в том, чтобы создавать приложения, которые не будут задействовать блок-

[5] Следует различать приложение Bitcoin и криптовалюту биткойн. Здесь авторы имеют в виду децентрализованное приложение Bitcoin.

чейн для всех операций подряд, а станут использовать его с умом. Другими словами, эта концепция заключается в том, чтобы обеспечить выполнение тяжелой работы на локальном уровне или организовать громоздкие хранилища данных вне блокчейна, чтобы базовая цепочка блоков была легкой и эффективной, а сетевой трафик не был слишком большим.

## 1.4.2. Уровень выполнения (execution layer)

*Уровень выполнения* — это уровень, на котором происходит выполнение инструкций, сформированных приложениями, на всех узлах сети блокчейна. Инструкции могут быть простыми командами или набором инструкций в форме *умного контракта*. В любом случае необходимо где-то выполнить программу или сценарий, чтобы обеспечить правильное прохождение транзакции. Все узлы в сети блокчейна должны выполнять программы/сценарии независимо друг от друга. Детерминированное выполнение программ/сценариев на одном и том же наборе входных данных и условий всегда дает одинаковый выходной результат на всех узлах, что помогает избежать несоответствий.

В случае с Bitcoin — это простые сценарии, которые не являются полными по Тьюрингу и допускают выполнение лишь небольшого набора инструкций. Системы Ethereum и Hyperledger, с другой стороны, допускают сложные сценарии. Код Ethereum — его умные контракты, написанные на языке Solidity, — компилируются в байт-код или машинный код, который выполняется на собственной *виртуальной машине Ethereum*. Hyperledger реализует более простой подход к своим умным контрактам. Он поддерживает запуск скомпилированных машинных кодов внутри Docker-образа и поддерживает несколько языков высокого уровня, таких как Java и Go.

## 1.4.3. Семантический уровень (semantic layer)

*Семантический уровень* — это логический уровень, потому что в транзакциях и блоках есть упорядоченность. Транзакция, действительная или недействительная, имеет набор инструкций, которые проходят через уровень выполнения, но проверяются на семантическом уровне. Если это Bitcoin, то на семантическом уровне проверяется, совершает ли кто-либо законную транзакцию, не является ли она атакой двойного расхода (например, у кого-то есть только десять долларов и он пытается заплатить эту сумму одновременно нескольким людям), авторизован ли он на совершение этой транзакции и т. п. В следующих главах вы узнаете, что биткойны фактически существуют в виде транзакций, представляющих состояние системы. Чтобы иметь возможность потратить биткойны, вы должны опираться на одну или несколько предыдущих транзакций, потому что в системе Bitcoin нет понятия личного счета. Это означает, что когда кто-то совершает транзакцию, он использует одну из предыдущих транзакций, по которой получил как минимум ту сумму, которую тратит сейчас. Новая транзакция должна быть проверена всеми узлами путем обхода предыдущих транзакций, чтобы определить, является ли она допус-

тимой транзакцией. В системе Ethereum, напротив, имеется система балансовых счетов. Это означает, что после сделки будет обновлен счет того, кто отправил транзакцию, и счет того, кто ее получил.

На этом уровне могут быть определены правила системы, такие как модели данных и структуры. Могут возникать ситуации, которые немного сложнее по сравнению с простыми транзакциями. Сложные наборы команд часто кодируются в виде умных контрактов. *Умный контракт* — это особый тип учетной записи с исполняемым кодом и частными состояниями. Состояние системы обновляется, если при получении транзакции сработал код умного контракта. Блок обычно содержит несколько транзакций и несколько смарт-контрактов. Структуры данных — такие как дерево Меркла — формируются на этом уровне через корень Меркла в заголовке блока и поддерживают связь между заголовками блоков и набором транзакций в блоке (обычно это пары «ключ-значение», хранящиеся на диске). Также на этом логическом уровне могут быть определены модели данных, режимы хранения диск/память и пр.

Кроме того, семантический уровень определяет, как блоки связаны друг с другом. Каждый блок в блокчейне содержит хэш предыдущего блока, вплоть до генезиса. Хотя конечное состояние блокчейна достигается за счет вкладов всех уровней, на семантическом уровне необходимо определить связь блоков друг с другом. В зависимости от варианта использования, вы можете закодировать в этом слое дополнительные функции.

## 1.4.4. Уровень распространения (propagation layer)

Предыдущие уровни были скорее индивидуальным явлением — у них невелика координация действий с другими узлами в системе. *Уровень распространения* — это уровень одноранговой связи, который позволяет узлам обнаруживать друг друга, а также общаться и синхронизироваться друг с другом относительно текущего состояния сети. Когда сделка совершена, она распространяется (или, как иногда говорят, *транслируется*) по всей сети. Точно так же, когда узел предлагает новый блок, он немедленно распространяется по всей сети, чтобы другие узлы могли использовать его в качестве последнего блока цепочки. Таким образом, на этом уровне определяется распространение транзакций/блоков в сети, и это распространение обеспечивает устойчивость всей сети. По своей конструкции большинство блокчейнов спроектировано таким образом, что они незамедлительно направляют транзакцию/блок сразу ко всем подключенным узлам, как только становится известно о новой транзакции или блоке.

В асинхронной сети Интернет часто возникают проблемы с задержкой распространения транзакций или блоков. Иногда распространение происходит в течение нескольких секунд, а иногда занимает больше времени, — в зависимости от пропускной способности узлов, пропускной способности сети и некоторых других факторов.

### 1.4.5. Уровень консенсуса (consensus layer)

*Уровень консенсуса* обычно является главным для большинства блокчейновых систем. Основное назначение этого уровня — добиться от всех узлов согласия с одним определенным состоянием реестра. В зависимости от варианта использования блокчейна, могут применяться разные способы достижения консенсуса между узлами. Именно на этом уровне формируется безопасность и надежность блокчейна. В сети Bitcoin или Ethereum консенсус достигается с помощью методов поощрения, называемых *майнингом*. Для того чтобы открытый блокчейн не нуждался в центральном руководящем узле, должны существовать какие-то механизмы стимулирования, которые не только помогают поддерживать сеть, но и обеспечивают согласие между узлами — консенсус. Bitcoin и Ethereum используют механизм консенсуса Proof of Work (PoW, доказательство работы) для случайного выбора узла, который может найти и предложить сети новый блок. Как только новый блок найден и распространен на все узлы, они проверяют, является ли этот блок допустимым блоком со всеми законными транзакциями, и правильно ли решена головоломка PoW. Затем узлы добавляют этот блок в свою собственную копию блокчейна. Существует множество вариантов протокола консенсуса — таких как: Proof of Stake (PoS, доказательство владения), PoS с разделением полномочий (delegated Pos, dPoS), прикладная византийская парадигма отказоустойчивости (Practical Byzantine Fault Tolerance, PBFT) и другие, которые мы подробно рассмотрим в следующих главах.

## 1.5. Почему блокчейн так важен?

Мы рассмотрели особенности устройства централизованных и децентрализованных систем и получили некоторое представление о технических преимуществах децентрализованных систем по сравнению с централизованными системами. Мы также узнали о различных уровнях блокчейна. Блокчейн, представляющий собой децентрализованную одноранговую систему, имеет свои преимущества и недостатки. Имейте в виду, что блокчейн — это не волшебная палочка, которая может решить все проблемы в мире, но есть конкретные случаи, когда он помогает прямо сейчас. Бывают также ситуации, когда добавление блокчейна в существующее решение делает его более надежным, прозрачным и защищенным. Тем не менее, внедрение блокчейна может привести к катастрофе, если не будет реализовано правильно! Давайте теперь проанализируем блокчейн с точки зрения функциональности.

### 1.5.1. Ограничения централизованных систем

Если вы окинете взглядом положение дел в области программного обеспечения, то увидите, что многие программные решения имеют централизованный характер. Причина не в том, что их легко разрабатывать и поддерживать, а в том, что мы привыкли к такому устройству, чтобы иметь возможность доверять системе. Нам всегда нужна надежная третья сторона, которая может удостоверить, что нас не

обманывают, и мы не станем жертвами мошенничества. Мало кто захочет иметь дело с теми, кого не знал раньше.

Давайте приведем пример из повседневной жизни. Сегодня, когда мы заказываем что-то из Amazon, мы чувствуем себя в безопасности и уверены в доставке товара. Производитель товара — это одна сторона сделки, а покупатель — другая сторона. Тогда какую роль здесь играет Amazon? Он действует в качестве доверенного посредника, а также упрощает транзакции. Покупатель доверяет продавцу, хотя доверительные отношения фактически навязаны посредником. Блокчейн говорит нам, что в современную цифровую эпоху на самом деле не нужна третья сторона, которая навязывает доверие, и технология уже достаточно развита, чтобы обойтись без посредника. В блокчейне доверие является неотъемлемой частью системы по умолчанию, о чем мы поговорим подробнее в следующих главах.

Бегло перечислим некоторые недостатки обычной централизованной системы:

- проблема доверия;
- проблема безопасности;
- проблема конфиденциальности — персональные данные постоянно продают;
- стоимость и временной фактор транзакций.

Можно назвать и некоторые преимущества децентрализованных систем:

- устранение посредников;
- более простая и достоверная проверка транзакций;
- повышенная безопасность с меньшими затратами;
- большая прозрачность;
- отсутствие уязвимого центра и неизменность.

## 1.5.2. Долго ли ждать блокчейн?

Блокчейн явился миру в 2009 году вместе с цифровой криптовалютой биткойн через простой список рассылки. Вскоре после запуска биткойна люди смогли оценить истинный потенциал блокчейна, далеко выходящий за рамки криптовалюты. Некоторые компании предложили новые варианты применения блокчейна, такие как Ethereum, Hyperledger и тому подобные. Microsoft и IBM предложили SaaS (Software as a Service, программное обеспечение как услуга) на своих облачных платформах Azure и Bluemix соответственно. Были созданы различные стартапы, а многие признанные компании выступили с инициативами внедрения блокчейна для решения некоторых бизнес-задач, которые раньше было нелегко решить.

Сейчас уже поздно говорить, что блокчейн обладает огромным потенциалом, чтобы так или иначе встряхнуть практически все отрасли, — революция уже идет. Она сильно повлияла на рынок финансовых услуг. Трудно назвать глобальный банк или финансовую организацию, которые не исследуют блокчейн. Помимо финансового рынка, энергичные исследования уже ведутся в таких областях, как медиа и развле-

чения, торговля энергоресурсами, рынки прогнозирования, розничные сети, системы поощрения лояльности, страхование, логистика и цепочки поставок, медицинские документы, а также органы государственного управления.

На момент подготовки этой книги текущая ситуация такова, что многие стартапы и компании хорошо представляют, как система, основанная на блокчейне, может решить некоторые болезненные проблемы. Тем не менее, разработка прикладного применения блокчейна является сложной задачей. Есть несколько действительно хороших идей для продуктов или решений на основе блокчейна, но их одинаково трудно как разработать, так и внедрить. Есть несколько вариантов использования, которые могут быть построены только на открытом (общедоступном) блокчейне. Разработка самодостаточного блокчейна с надлежащей экосистемой майнинга является весьма сложной задачей, и когда дело доходит до существующих открытых блокчейнов для создания приложений без криптовалюты, тут-то и оказывается, что нет ничего, кроме блокчейна Ethereum. Вопрос о том, должно ли приложение блокчейна быть только «оберткой» и использовать базовые слои готового блокчейна такими, как они есть, или приложение должно быть построено с нуля, включая инфраструктуру блокчейна, — это вопрос, на который трудно ответить однозначно. Есть и чисто технические проблемы. Технология блокчейна все еще развивается, и для массового внедрения может потребоваться еще несколько лет. На сегодняшний день существует несколько подходов к решению проблемы масштабируемости блокчейна. В нашей книге мы постараемся сформировать ясное понимание всех этих перспектив. А пока давайте рассмотрим некоторые конкретные варианты использования блокчейна.

## 1.6. Практическое применение блокчейна

В этом разделе мы рассмотрим лишь некоторые примеры, которые можно встретить в таких отраслях, как финансы, страхование, банковское дело, здравоохранение, правительство, логистика, IoT (Internet of Things, Интернет вещей), а также средства массовой информации и развлечения. Однако возможности блокчейна безграничны! Истинная *экономика совместного пользования*, которую было трудно построить в централизованных системах, становится возможной с использованием технологии блокчейна (например, пиринговые версии Uber и AirBNB). Также можно позволить гражданам владеть и управлять своей цифровой идентичностью (Self-Sovereign Digital Identity) и на основе технологии блокчейна монетизировать использование персональных данных.

- В блокчейне можно зарегистрировать любой тип имущества или активов, будь то физические или цифровые активы — такие как ноутбуки, мобильные телефоны, бриллианты, автомобили, недвижимость, электронная регистрация, цифровые файлы и т. д. Это решает проблемы передачи активов от одного человека к другому, ведения журнала транзакций и проверки действительности документа или права собственности. Кроме того, могут быть разработаны услуги цифрового нотариуса, подтверждение существования, индивидуальные схемы страхования и многие другие варианты использования.

- Существует множество вариантов финансового применения блокчейна — таких как трансграничные платежи, торговля акциями, система лояльности и вознаграждений, банковская система «Знай своего клиента» (Know Your Customer, KYC) и др. А первичный выпуск монет (Initial Coin Offering, ICO) — на момент подотовки этой книги — является сейчас одним из самых популярных видов использования блокчейна. На сегодняшний день ICO — это лучший способ краудсорсинга с использованием криптовалюты в качестве цифровых активов. Монету в ICO можно рассматривать как цифровую акцию предприятия, которую очень легко покупать и продавать.
- Блокчейн может быть задействован для того, чтобы позволить «Мудрости толпы»[6] взять на себя инициативу и переформатировать бизнес, экономику и прочие национальные институты, используя коллективную мудрость! Финансовые и экономические прогнозы, основанные на мудрости толпы, децентрализованных рынках прогнозирования, децентрализованном голосовании, а также на торговле акциями, могут быть реализованы при помощи блокчейна.
- Процесс распределения лицензионных платежей всегда был запутанным. Интернет-сервисы потоковой трансляции музыки облегчили выход авторов на рынок, но усложнили определение роялти. Эта проблема может быть в значительной степени решена с помощью блокчейна путем ведения общедоступного реестра информации о правах собственности на музыку, а также авторизованного распространения медиаконтента.
- Мы вошли в эру Интернета вещей (Internet of Things, IoT) с миллиардами IoT-устройств по всему миру. Наличие несогласованных производителей, моделей и протоколов связи затрудняет создание централизованной системы управления устройствами и общей платформы обмена данными. Это также область, где блокчейн можно использовать для построения децентрализованной одноранговой системы, чтобы устройства IoT могли обмениваться данными друг с другом. ADEPT (Autonomous Decentralized Peer-To-Peer Telemetry, Автономная децентрализованная одноранговая телеметрия) — это совместная инициатива IBM и Samsung, которые разработали платформу, использующую элементы базовой структуры биткойна для построения распределенной сети устройств — децентрализованного Интернета вещей. ADEPT задействует три протокола: BitTorrent — для обмена файлами, Ethereum — для умных контрактов и TeleHash — для обмена сообщениями между равноправными узлами на платформе. Еще одной такой инициативой является фонд IOTA.
- В государственном секторе блокчейн также набрал обороты. Существуют области деятельности, для которых необходима техническая децентрализация, но политически они должна регулироваться государством: регистрация земли,

---

[6] «Мудрость толпы» (The Wisdom of Crowds) — статистическое и социологическое явление, открытое Френсисом Гальтоном в 1906 году, согласно которому совокупная оценка большого количества случайных людей оказывается точнее, чем оценка небольшого количества экспертов, — за счет взаимной компенсации отклонений на достаточно большом количестве выборок.

регистрация собственности и права на управление транспортными средствами, электронное голосование и др. Цепочки поставок — это еще одна область, в которой есть несколько примеров использования блокчейна. Цепочки поставок постоянно оказывались предметом споров по всему миру, т. к. всегда было трудно поддерживать прозрачность в этих системах.

## 1.7. Заключение

В этой главе мы рассмотрели эволюцию блокчейна, его историю, преимущества его устройства и почему это так важно в некоторых случаях. Давайте подведем итог рассуждениям о технологической революции, которая меняет правила игры.

В 1990-х годах массовое внедрение Интернета изменило способы ведения бизнеса и убрало препятствия для создания и распространения информации. Это, в свою очередь, проложило путь на новые рынки с большими перспективами. Точно так же блокчейн призван вывести Интернет на совершенно новый уровень, устранив препятствия в трех ключевых областях: управление, доверие и ценность.

- **Управление**: блокчейн позволил распределить управление, сделав систему децентрализованной.
- **Доверие**: блокчейн представляет собой неизменяемый, защищенный от несанкционированного доступа реестр. Это дает единый, общий источник истины для всех узлов, делая систему надежной без посредника. Для совершения сделок с каким-либо неизвестным лицом или организацией больше не требуется доверие, ибо оно заложено в саму сущность системы.
- **Ценность**: блокчейн позволяет обмениваться ценностями в любой форме. Можно выпускать и перемещать активы без центральных органов или посредников.

В *главе 2* мы глубоко окунемся в теоретические основы блокчейна.

## 1.8. Рекомендуемые источники

- Значение децентрализации: Buterin, Vitalik, «The Meaning of Decentralization[7]», Medium, **https://medium.com/@VitalikButerin/the-meaning-of-decentralization-a0c92b76a274**, February 6, 2017.
- Технология блокчейна:
  - Crosby, Michael; Nachiappan; Pattanayak, Pradhan; Verma, Sanjeev; Kalyanaraman, Vignesh, «BlockChain Technology: Beyond Bitcoin» Berkeley, CA: Sutardja Center for Entrepreneurship & Technology, University of California.
  - **http://scet.berkeley.edu/wp-content/uploads/BlockchainPaper.pdf**, October 16, 2015.

---

[7] Русский перевод опубликован по адресу: **https://bits.media/vitalik-buterin-o-znachenii-detsentralizatsii/**.

- Torpey, Kyle, «Eric Lombrozo: Bitcoin Needs Protocol Layers Similar to the Internet», CoinJournal.
- **https://coinjournal.net/eric-lombrozo-bitcoin-needs-protocol-layers-similar-to-the-internet/**, January 28, 2016.

♦ Blockbench: платформа для анализа частных блокчейнов: Dinh, Tien Tuan Anh; Wang, Ji; Chen, Gang; Liu, Rui; Ooi, Beng Chin; Tan, Kian-Lee, «Blockbench: A Framework for Analyzing Private blockchains», **https://arxiv.org/pdf/1703.04057.pdf**, March 12, 2017.

ГЛАВА 2

# Как работает блокчейн?

Мы стоим на пороге новой цифровой революции. Блокчейн, наверное, самое большое изобретение со времен самого Интернета! Это наиболее перспективная технология для следующего поколения систем, основанных на сетевых коммуникациях, и ей уделяется большое внимание во многих секторах промышленности, а также в научных кругах. Сегодня многие организации уже поняли, что им нужно быть готовыми к приходу блокчейна, чтобы сохранить свои позиции на рынке. Мы уже рассмотрели несколько вариантов использования блокчейна в *главе 1*, но возможности блокчейна безграничны. Хотя блокчейн не является универсальным средством для решения всех задач бизнеса, он начал оказывать влияние на большинство бизнес-функций и применяемые в них технологии.

Чтобы научиться решать реальные проблемы бизнеса с помощью блокчейна, нам необходимо глубокое понимание того, *что* такое блокчейн и *как* он работает. Для этого необходимо изучить блокчейн с разных точек зрения, таких как бизнес, технические и юридические аспекты. Эта глава — попытка понять основы блокчейна и получить полное представление о том, как он работает.

## 2.1. Фундаментальные основы блокчейна

Блокчейн — это не просто компьютерная технология, он прочно связан с деловыми функциями в повседневной жизни. В виде криптовалюты он также переплетается с экономическими принципами. В этом разделе мы сосредоточимся в основном на его технических аспектах. С технической точки зрения блокчейн представляет собой блестящее объединение концепций криптографии, теории игр и информатики (рис. 2.1).

Давайте посмотрим, какую роль эти компоненты играют в устройстве блокчейна на общем уровне, а затем углубимся в основы. Но прежде вспомним, как работали

привычные централизованные системы. Традиционный подход состоял в том, что существует централизованная система, которая поддерживает только одну историю транзакций/изменений. Она должна осуществлять параллельный контроль над всей базой данных и обеспечивать доверие системе через посредников. В чем же проблема с такой стабильной системой? В том, что приходится доверять централизованной системе, честная она или нет! Кроме того, неизбежны затраты на посредников, а время прохождения транзакции может быть большим по очевидным причинам. Теперь задумайтесь о централизации власти — полный контроль над всей системой позволяет централизованным органам правления делать практически все, что они хотят.

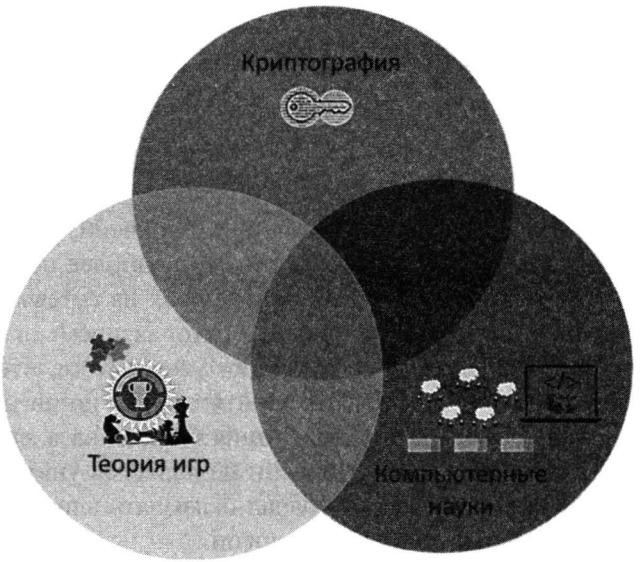



**Рис. 2.1.** Ядро технологии блокчейна

Теперь давайте обсудим, как блокчейн решает проблему наличия централизованных посредников с помощью криптографии, теории игр и информатики. Все транзакции блокчейна криптографически защищены независимо от их назначения. Используя криптографию, можно гарантировать, что лишь подлинный пользователь инициирует транзакцию, и никто не может выступать от его имени. Это означает, что с помощью криптографии можно гарантировать, что Алиса никоим образом не сможет совершить транзакцию от имени Боба, подделав его подпись.

Но что, если узел или пользователь пытается запустить *атаку с двойным расходом* (например, у кого-то есть только десять долларов, и он пытается одновременно заплатить эту сумму нескольким людям)? Обратите пристальное внимание — несмотря на отсутствие достаточных средств, двойные расходы являются криптографически допустимыми. Алгоритму шифрования абсолютно безразлично, сколько у вас денег.

> Единственный способ предотвратить двойные расходы — сделать так, чтобы каждый узел знал обо всех транзакциях.

Однако это приводит к другой интересной проблеме. Поскольку каждый узел должен поддерживать базу данных транзакций, как они могут согласовать общее состояние базы данных? Опять же, как система может оставаться невосприимчивой к ситуациям, когда один или несколько вычислительных узлов намеренно пытаются подорвать систему и внедрить фальшивое состояние базы данных? Большинство таких ситуаций относится к *проблеме византийских генералов* (она рассмотрена далее). В общем-то, проблема византийских генералов приобрела еще большую популярность из-за блокчейна, но фактически она известна целую вечность. Если вы посмотрите на реализацию центров обработки данных или распределенных баз данных, станет ясно, что проблема византийских генералов является очевидной и распространенной проблемой, с которой они сталкиваются при обеспечении отказоустойчивости. На самом деле и подобные проблемы, и их решения пришли к нам из теории игр, которая предоставляет принципиально иной подход к определению поведения системы. Методы теории игр, пожалуй, самые жесткие и циничные. Обычно они никогда не учитывают, является ли узел честным, злонамеренным, этичным или имеет какие-либо другие подобные характеристики, и считают, что участники действуют исключительно в соответствии с выгодами, которые они получают, и не подвержены предрассудкам морали. Единственная цель теории игр в блокчейне — обеспечить стабильность системы (т. е. равновесие по Нэшу) с консенсусом среди участников.

Существуют различные виды прикладных задач и ситуаций с различной степенью сложности. Соответственно, базовые протоколы криптографии и консенсуса теории игр могут быть разными в зависимости от прикладного применения. Тем не менее, общий принцип ведения согласованного реестра или базы данных проверенных транзакций всегда один и тот же. Хотя понятия криптографии и теории игр существуют уже весьма давно, именно в области компьютерных наук эти фрагменты соединяются между собой посредством организованных структур данных и технологий одноранговой сети. Очевидно, что для реализации любых логических или математических концепций в цифровом мире необходима «умная разработка программного обеспечения», охватывающая смежные области знаний. Именно благодаря такому подходу разработчики блокчейна включают в приложения понятия криптографии и теории игр, позволяя децентрализовать и распределить вычисления между узлами.

## 2.2. Криптография

Криптография является наиболее важной составляющей блокчейна. Это самостоятельная область исследований, основанная на передовых математических методах, которые весьма сложны для понимания. В этом разделе мы попытаемся сформировать у вас глубокое понимание некоторых криптографических концепций, потому что разные проблемы могут потребовать разных криптографических решений, — один метод никогда не подходит всем. Вы можете пропустить некоторые детали или возвращаться к ним по мере необходимости, но это самый важный компонент

обеспечения безопасности в системе. Из-за недостаточной криптозащищенности взломано много кошельков и обменных бирж.

Криптография существует уже более двух тысяч лет. Это наука о конфиденциальности с применением методов шифрования. Однако конфиденциальность — не единственная цель. В следующем списке приведены и другие способы использования криптографии, которые мы рассмотрим позже:

- *конфиденциальность* — только предполагаемый или авторизованный получатель может понять сообщение. Это также называют *приватностью* или *секретностью*;
- *целостность данных* — данные не могут быть подделаны/изменены злоумышленником или за счет случайных ошибок. Хотя целостность данных не может предотвратить изменение данных, она предоставляет средства проверки того, были ли данные изменены;
- *аутентификация* — подлинность отправителя гарантированно проверяется получателем;
- *безотзывность* — отправитель после отправки сообщения не может позднее отрицать, что отправил сообщение. Это означает, что отправитель (человек или система) не может отказаться от ответственности за предыдущие обязательства или действия.

Любая информация в виде текстового сообщения, числовых данных или компьютерной программы может называться *открытым текстом*. Идея криптографии состоит в том, чтобы зашифровать открытый текст, используя алгоритм шифрования и ключ, которые создают *закрытый текст*. Затем закрытый текст может быть передан предполагаемому получателю, который расшифровывает его, используя алгоритм дешифрования и ключ для получения открытого текста.

Давайте рассмотрим пример. Алиса хочет отправить сообщение (m) Бобу. Если она просто отправит сообщение как есть, любой противник — скажем, Ева — может легко перехватить сообщение, и конфиденциальность будет нарушена. Итак, Алиса хочет зашифровать сообщение, используя алгоритм шифрования (E) и секретный ключ (K), чтобы создать зашифрованное сообщение (шифротекст). Злоумышленник должен знать как алгоритм (E), так и ключ (K), чтобы перехватить сообщение. Чем сильнее алгоритм и ключ, тем сложнее атаковать противника. Обратите внимание, что всегда желательно проектировать блокчейн-системы, которые, по крайней мере, *доказуемо безопасны*. Это означает, что система должна гарантированно противостоять определенным типам возможных атак противников.

Общие принципы работы криптографии схематически изображены на рис. 2.2.

В целом, существуют два типа криптографии: криптография с *симметричным ключом* и криптография с *асимметричным (открытым) ключом*. Давайте рассмотрим их по отдельности в следующих разделах.

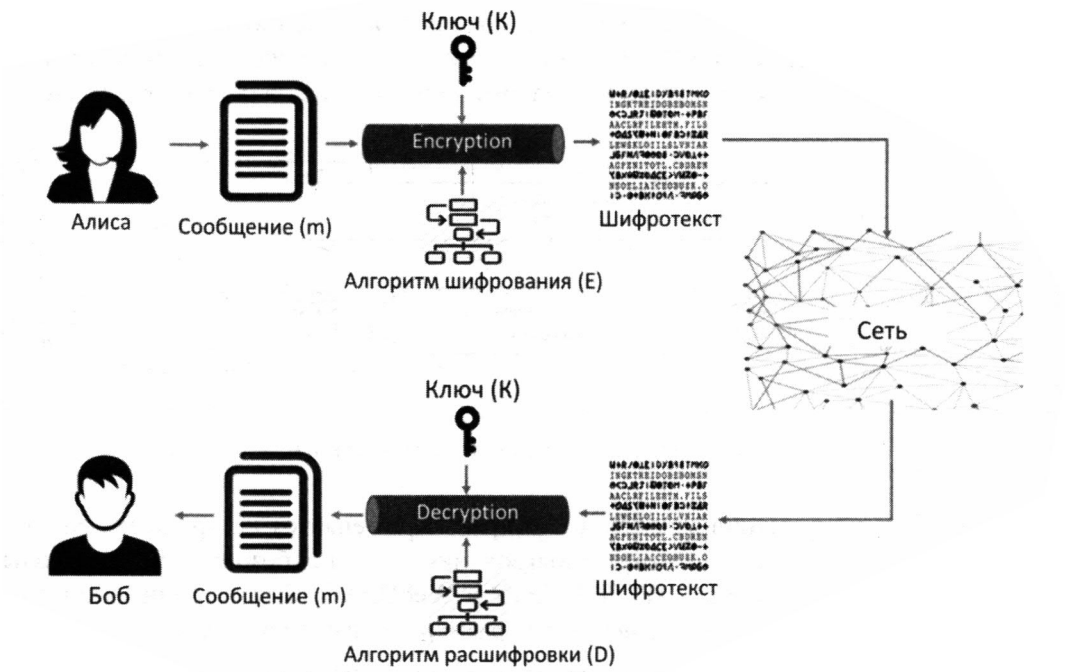



**Рис. 2.2. Общие принципы работы криптографии**

## 2.2.1. Криптография с симметричным ключом

В предыдущем разделе мы рассмотрели, как Алиса может зашифровать сообщение и отправить закрытый текст Бобу. Затем Боб может расшифровать закрытый текст, чтобы получить исходное сообщение. Если один и тот же ключ используется для шифрования и дешифрования, этот метод называется *криптографией с симметричным ключом*. Это означает, что и Алиса, и Боб должны заранее согласовать ключ (***K***), так называемый *общий секрет*, прежде чем они обменяются зашифрованным текстом. Итак, процесс выглядит следующим образом (рис. 2.3):

- Алиса — отправитель:
  - шифрует текстовое сообщение ***m***, используя алгоритм шифрования ***E*** и ключ ***K***, чтобы подготовить закрытый текст ***c***;
  - ***c*** = ***E*** (***K***, ***m***);
  - отправляет закрытый текст ***c*** Бобу.
- Боб — получатель:
  - расшифровывает закрытый текст ***c***, используя алгоритм дешифрования ***D*** и тот же ключ (***K***), чтобы получить открытый текст ***m***;
  - ***m*** = ***D*** (***K***, ***c***).

Вы обратили внимание, что отправитель и получатель использовали один и тот же ключ (*K*)? Как они договариваются об одном и том же ключе и делятся им друг с другом? Очевидно, что им нужен безопасный канал связи для обмена ключами.

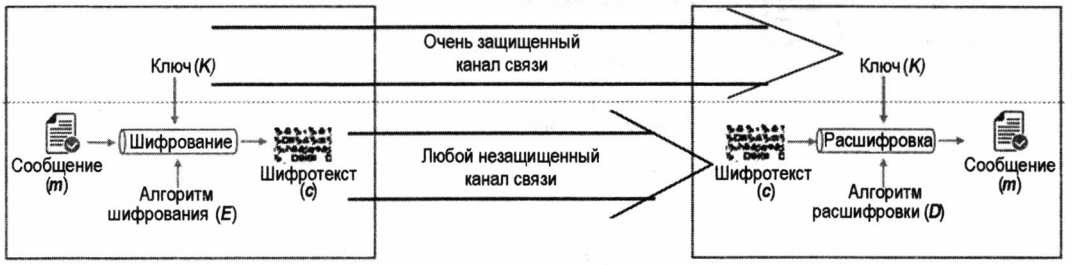



**Рис. 2.3.** Криптография с симметричным ключом

Криптография с симметричным ключом широко применяется на практике. Наиболее распространенными областями использования являются безопасные протоколы передачи файлов — такие как HTTPS, SFTP и WebDAVS. Симметричные криптосистемы обычно быстрее и выгоднее, когда размер данных очень велик.

Отметим, что криптография с симметричным ключом существует в двух вариантах: *потоковые шифры* и *блочные шифры*. Мы обсудим это в следующих разделах, но перед этим рассмотрим *принцип Керкгоффса*[1] и *функцию XOR*, чтобы понять, как работают криптосистемы.

## Принцип Керкгоффса и функция XOR

Принцип Керкгоффса гласит, что *криптосистема должна быть защищена, даже если о ней известно вообще все, кроме ключа*. Кроме того, общее предположение заключается в том, что *канал передачи сообщений никогда не является безопасным*, и сообщения могут быть легко перехвачены во время передачи. Это означает, что даже если алгоритм шифрования (*E*) и алгоритм дешифрования (*D*) являются общедоступными, и существует вероятность того, что сообщение может быть перехвачено во время передачи, сообщение все еще защищено из-за наличия общего секрета. Следовательно, в симметричной криптосистеме ключи должны храниться в строжайшем секрете.

Функция XOR является основным строительным блоком для многих алгоритмов шифрования и дешифрования. Давайте подробнее рассмотрим эту функцию, чтобы понять, как она помогает криптографии. Функция XOR, иначе известная как «Исключающее ИЛИ» и обозначаемая символом ⊕, может быть представлена следующей таблицей истинности (табл. 2.1).

---

[1] Огюст Керкгоффс (Auguste Kerckhoffs, 1835–1903) — великий нидерландский криптограф, лингвист, историк и математик.

***Таблица 2.1.*** *Таблица истинности функции XOR*

| A | B | A ⊕ B |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Функция XOR имеет следующие свойства, которые важны для понимания математики, лежащей в основе криптографии:

- ассоциативность: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$;
- коммутативность: $A \oplus B = B \oplus A$;
- отрицание: $A \oplus 1 = \bar{A}$;
- идентичность: $A \oplus A = 0$.

Теперь можно вычислить закрытый текст $c$ с использованием открытого текста $m$ и ключа $K$, а затем расшифровать закрытый текст $c$ с помощью того же ключа $K$, чтобы получить открытый текст $m$.

> Одна и та же функция XOR используется как для шифрования, так и для дешифрования:
>
> $$m \oplus K = c$$
> $$c \oplus K = m$$

Приведенный пример дан в самой простейшей форме, чтобы получить представление о шифровании и дешифровании. Заметьте, что очень легко получить исходное текстовое сообщение, просто применив XOR к зашифрованному тексту с ключом, который является общим секретом и известен только доверенным сторонам. Каждый может знать, что алгоритмом шифрования и дешифрования является функция XOR, но никто посторонний не должен знать ключ.

## Потоковое и блочное шифрование

Алгоритмы потокового шифрования и блочного шифрования различаются по способу кодирования и декодирования сообщений.

*Потоковые шифры* преобразуют один символ открытого текста в один символ шифрованного текста. Это означает, что шифрование выполняется по одному биту или байту открытого текста за один шаг. В побитовом сценарии шифрования для шифрования каждого бита открытого текста генерируется и используется другой ключ. Таким образом, потоковый шифр использует бесконечный поток псевдослучайных битов в качестве ключа и выполняет операцию XOR с входными битами открытого текста для генерации закрытого текста. Чтобы такая система оставалась безопасной, генератор псевдослучайных ключей должен быть безопасным и не-

предсказуемым. Потоковые шифры — это аппроксимация абсолютно стойкого шифра, так называемого *одноразового блокнота*[2], о котором мы поговорим чуть позже.

Как генерируется псевдослучайный ключевой поток? Обычно он последовательно генерируется из случайного начального значения с использованием цифровых сдвиговых регистров. Потоковые шифры довольно просты и быстро работают. Можно заранее сгенерировать псевдослучайные биты, а затем очень быстро выполнять расшифровку, но в большинстве случаев требуется синхронность действий.

Очевидно, что генератор псевдослучайных чисел, который генерирует ключевой поток, является центральным элементом, от которого зависит уровень безопасности. В этом заключается главный недостаток алгоритма. В прошлом генератор псевдослучайных чисел подвергался множественным атакам, что приводило к устареванию потоковых шифров. Наиболее широко используемым потоковым шифром является RC4 (Rivest Cipher 4) для различных протоколов, таких как SSL, TLS, Wi-Fi WEP/WPA и др. Обнаружено, что в RC4 существуют уязвимости, поэтому Mozilla и Microsoft рекомендовали по возможности не использовать этот шифр.

Другим недостатком является то, что каждому биту исходного текста соответствует бит зашифрованного текста, следующий в том же порядке. Это проблема так называемой *низкой диффузии*. Шифр был бы более защищенным, если бы информация об одном входном бите после шифрования перемешивалась с информацией о других битах выходного зашифрованного текста, как в случае с блочными шифрами. Примерами потоковых шифров являются одноразовый блокнот, RC4, FISH, SNOW, SEAL, A5/1 и другие шифры.

*Блочный шифр*, в свою очередь, основан на идее разделения открытого текста на относительно большие блоки фиксированной длины и дальнейшего кодирования каждого из блоков по отдельности с использованием одного и того же ключа. Это *детерминированный алгоритм с постоянным преобразованием* с использованием симметричного ключа. Это означает, что когда вы шифруете один и тот же блок открытого текста тем же ключом, вы получаете один и тот же результат.

Стандартные размеры каждого исходного блока составляют 64 бита, 128 битов и 256 битов, называемых *длиной блока*. Результирующие блоки зашифрованного текста также имеют одинаковую длину. Мы берем, скажем, $r$-битный ключ $k$ для шифрования каждого блока длины $n$ и ограничиваем перестановки ключа $k$ очень маленьким подмножеством $2^r$. Это означает, что понятие «идеальный шифр» здесь не применимо. Тем не менее, случайный выбор секретного $r$-битного ключа важен в том смысле, что большая случайность подразумевает большую секретность.

---

[2] Одноразовый блокнот (шифр Вернама) — шифр, в котором ключ имеет длину, равную длине самого сообщения. Если каждый бит ключа выбран абсолютно случайно, то вскрыть шифр невозможно.

Чтобы зашифровать или расшифровать сообщение в криптографии с блочным шифром, мы должны использовать режим работы, который определяет, каким образом повторно применять операцию шифрования одиночного блока для кодирования объемов данных, больших, чем блок. Так вот, работа заключается не только в том, чтобы зашифровать блоки данных фиксированного размера, — задача намного серьезнее. Мы узнали, что блочный шифр является детерминированным алгоритмом. Это означает, что блоки с одинаковыми данными при шифровании с использованием одного и того же ключа будут создавать один и тот же зашифрованный текст, — это весьма опасно и на больших массивах данных открывает злоумышленнику слишком широкий простор для криптоанализа. Идея заключается в том, чтобы определенным образом смешать блоки открытого текста с только что созданными блоками зашифрованного текста, чтобы соответствующие блоки зашифрованного текста были разными. Пока не очень понятно, но станет понятнее, когда мы перейдем к алгоритмам DES и AES в следующих разделах.

Обратите внимание, что различные режимы работы приводят к достижению разного уровня безопасности базового блочного шифра. Хотя мы не будем вдаваться в подробности режимов работы, вот вам несколько названий для справки: *режим электронной кодовой книги* (Electronic CodeBook, ECB), *режим сцепления блоков* (Cipher Block Chaining, CBC), *режим обратной связи по шифротексту* (Cipher Feedback, CFB), *режим обратной связи вывода* (Output Feedback, OFB) и *режим счетчика* (CTR).

Блочные шифры работают немного медленнее по сравнению с потоковыми шифрами. В отличие от потоковых шифров, где влияние ошибок намного меньше, здесь ошибка в одном бите может повредить весь блок. Зато блочные шифры обладают преимуществом высокой диффузии, а это означает, что каждый входной бит открытого текста распространяется по нескольким символам шифрованного текста. Примерами блочных шифров являются DES, 3DES, AES и др.

**ПРИМЕЧАНИЕ.** Детерминированный алгоритм — это алгоритм, который при заданном входном сигнале всегда будет давать один и тот же результат.

## Одноразовый блокнот

Это симметричный потоковый шифр, где открытый текст, ключ и зашифрованный текст являются битовыми строками. Кроме того, он полностью основан на предположении об «истинно случайном» ключе (а не псевдослучайном), с помощью которого он может достичь «идеальной секретности». Кроме того, согласно задумке, ключ может использоваться только один раз. Проблема заключается в том, что ключ должен быть как минимум таким же длинным, как открытый текст. Это означает, что если вы шифруете файл размером 1 Гбайт, ключ также будет 1 Гбайт! Во многих реальных ситуациях это неприемлемо.

Чтобы узнать, например, как при помощи функции XOR с ключом генерируется зашифрованный текст, вы можете вернуться к таблице истинности XOR в преды-

дущем разделе (см. табл. 2.1). Обратите внимание, что открытый текст, ключ и зашифрованный текст имеют длину 18 битов.

Получатель после получения зашифрованного текста может снова применить функцию XOR с ключом и получить открытый текст. Если вы попробуете проделать это самостоятельно с табл. 2.2, то получите исходный текст из шифротекста.

***Таблица 2.2.*** *Пример шифрования с использованием функции XOR*

| **Исходный текст** | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Ключ** | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| **Шифротекст** | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |

Основная проблема с одноразовым блокнотом — скорее практическая, чем теоретическая. Как отправитель и получатель согласовывают между собой секретный ключ? Если отправитель и получатель уже имеют защищенный канал для передачи ключа, зачем им вообще нужен ключ? Если у них нет защищенного канала (именно поэтому мы используем криптографию), то как они могут безопасно передать ключ? Эта проблема называется *проблемой раздачи ключей.*

Решение состоит в том, чтобы имитировать одноразовый блокнот, используя генератор псевдослучайных чисел (PseudoRandom Number Generator, PRNG). Это детерминированный алгоритм, который использует начальное значение для генерации последовательности случайных чисел, которые не являются действительно случайными, что само по себе представляет проблему. Для работы этой системы отправитель и получатель должны знать одинаковое начальное значение генератора. Раздача небольшого начального значения намного проще по сравнению с раздачей полного ключа, но очень важно обеспечить безопасность и этой раздачи, — ведь ключ может быть рассекречен любым, кто знает алгоритм генерации и начальное значение.

## Стандарт шифрования данных DES

Стандарт шифрования данных (Data Encryption Standard, DES) — это название метода симметричного блочного шифрования. Он использует 64-битный размер блока с 64-битным ключом для шифрования и дешифрования. Из 64-битного ключа 8 битов зарезервированы для контроля четности, а оставшиеся 56 битов — это длина ключа. Было доказано, что DES уязвим для атаки перебором и может быть взломан менее чем за день. Принимая во внимание закон Мура, в будущем он может быть взломан намного быстрее, поэтому его использование давно устарело из-за недостаточной длины ключа. В свое время DES был очень популярен и использовался в банковских приложениях, банкоматах и других коммерческих областях, и в большей степени в аппаратных реализациях, чем в программном обеспечении. В этом разделе мы дадим общее описание алгоритма шифрования DES.

В симметричной криптографии большое количество блочных шифров используют схему, известную как «шифр Фейстеля», или «сеть Фейстеля». Шифр Фейстеля состоит из нескольких раундов обработки открытого текста с ключом, и каждый раунд состоит из шага замены с последующим шагом перестановки. Чем больше количество раундов, тем выше безопасность, но медленнее шифрование и дешифрование. DES основан на шифре Фейстеля с 16-ю раундами. Общая последовательность шагов в алгоритме DES показана на рис. 2.4.

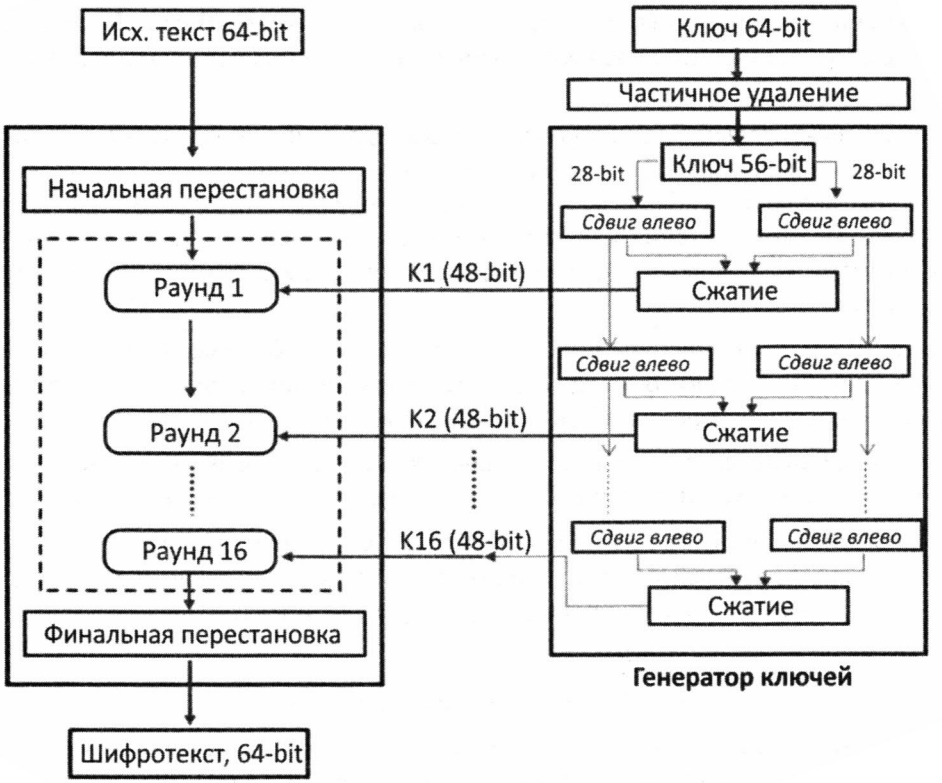



**Рис. 2.4.** Алгоритм DES

Давайте сначала поговорим о генераторе ключей, а затем перейдем к процессу шифрования:

- как упоминалось ранее, ключ имеет длину 64 бита. Поскольку 8 битов используются в качестве битов четности (точнее, биты 8, 16, 24, 32, 40, 48, 56 и 64), только оставшиеся 56 битов служат для шифрования и дешифрования;
- после удаления контроля четности 56-битный ключ делится на два блока по 28 битов. Затем они постепенно сдвигаются влево в каждом раунде. Мы знаем, что DES использует 16 раундов сети Фейстеля. Обратите внимание, что каждый раунд занимает место сдвинутого влево битового блока предыдущего раунда, а затем снова сдвигается влево на один бит в текущем раунде;

- оба сдвинутых влево 28-битовых блока затем объединяются с помощью механизма сжатия, который выводит 48-битовый подключ, используемый для шифрования. Аналогично, в каждом раунде два 28-битовых блока из предыдущего раунда снова сдвигаются влево на один бит, а затем объединяются и сжимаются до 48-битового ключа. Этот ключ затем направляется в функцию шифрования того же раунда.

Давайте теперь посмотрим, как DES использует раунды сети Фейстеля для шифрования:

- сначала входящий открытый текст делится на 64-битовые блоки. Если количество битов в сообщении не делится поровну на 64, то последний блок дополняется до 64 битов;
- каждый 64-битовый блок входных данных проходит раунд начальной перестановки (Initial Permutation, IP). Это простая пермутация, т. е. перестановка всех 64-х битов входного блока в определенном порядке. Этот раунд не имеет криптографического значения как такового, и его цель состоит в том, чтобы упростить загрузку открытого или зашифрованного текста в чипы DES в байтовом формате;
- после IP-раунда 64-битовый блок делится на два 32-битовых блока, левый блок ($L$) и правый блок ($R$). В каждом раунде блоки представлены как $L_i$ и $R_i$, где индекс «$i$» обозначает раунд. Итак, итоги IP-раунда обозначаются как $L_0$ и $R_0$;
- теперь начинаются собственно раунды Фейстеля. Первый раунд принимает $L_0$ и $R_0$ в качестве входных данных и выполняет следующие шаги (рис. 2.5):
  - правый входной 32-битный блок ($R$) проходит как есть на *левую* сторону, а левый 32-битный блок ($L$) участвует в операции с ключом $K$ этого раунда и 32-битным блоком на правой стороне ($R$), как показано здесь:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, Ki), \text{ где } i \text{ — число раундов}$$

  - Функция $f()$ называется *функцией шифрования* и является основной частью каждого раунда. Она состоит из нескольких последовательных этапов:
    - на первом этапе 32-битовый блок ($R$) дополняется до 48 битов и подвергается перестановке;
    - на втором этапе этот 48-битовый блок складывается по XOR с 48-битовым подключом $K_i$, полученным от генератора ключей того же раунда;
    - на третьем этапе 48-битный результат операции XOR подается в S-блок замещения, чтобы уменьшить разрядность до 32 битов. Операция замещения в этом S-блоке является единственной нелинейной операцией в DES и вносит существенный вклад в безопасность этого алгоритма;
    - на четвертом этапе 32-битовый выходной сигнал S-блока подается на P-блок перестановки, выход которого фактически является окончательным выходом функции шифрования $f()$;

- наконец, выход функции $f()$ складывается по XOR с 32-битовым L-блоком, который введен в этот раунд. Результат сложения становится итоговым выводом $R_i$ этого раунда.

- Упомянутый ранее раунд Фейстеля повторяется 16 раз, где результат одного раунда является входными данными для следующего раунда.
- После того как все 16 раундов закончились, результаты 16-го раунда снова меняются местами так, что левый блок становится правым блоком, и наоборот.
- Затем два блока объединяются в 64-битовый блок и проходят через операцию завершающей перестановки, которая является обратной по отношению к начальной перестановке и выводит 64-битовый зашифрованный текст.

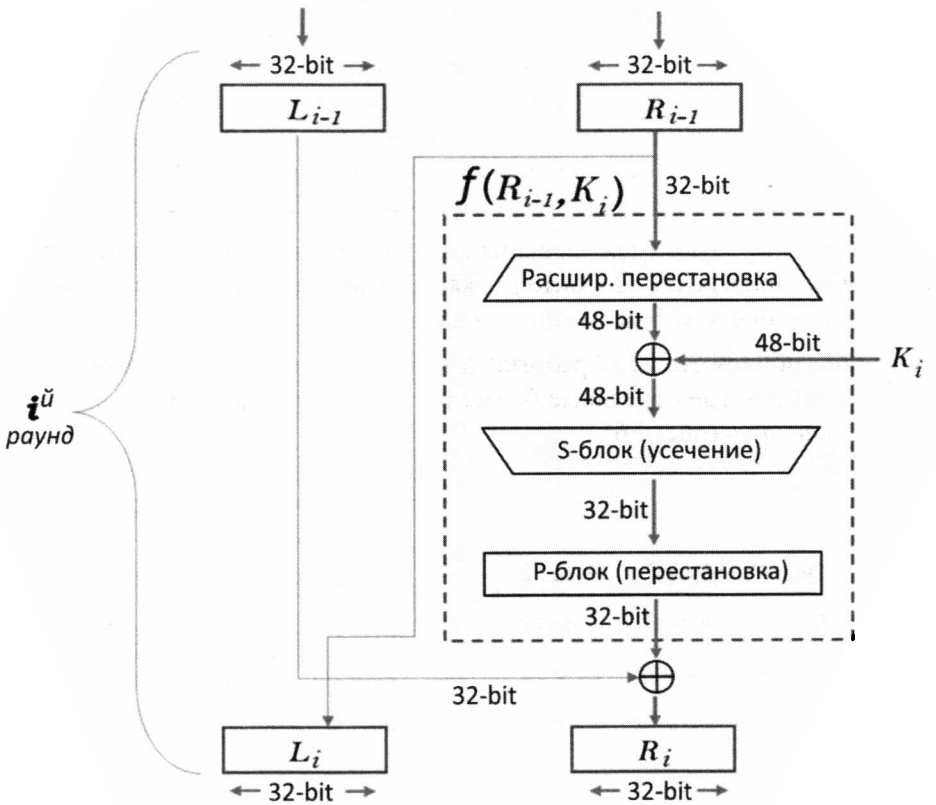



**Рис. 2.5.** Операции раунда DES

Итак, мы разобрались, как работает шифрование по алгоритму DES. Расшифровка работает аналогичным образом в обратном порядке. Мы не будем вдаваться в эти детали, но у вас есть возможность изучить расшифровку самостоятельно.

Как мы уже говорили, у стандарта DES есть серьезные недостатки. Короткий 56-битовый ключ подвержен атаке методом прямого подбора, а S-блоки, используемые для подстановки в каждом раунде, подвержены атаке криптоанализом из-за

некоторых присущих им недостатков. По этим причинам на смену DES пришел расширенный стандарт шифрования (Advanced Encryption Standard, AES). Теперь во многих приложениях используют AES вместо DES.

## Расширенный стандарт шифрования AES

Как и DES, алгоритм AES также является симметричным блочным шифром, но не основан на сети Фейстеля в ее исходном виде. AES использует подстановочно-перестановочную сеть в более общем смысле. Это не только повышает безопасность, но и увеличивает скорость. В соответствии со стандартами AES размер блока всегда равен 128 битам, и можно выбирать один из трех ключей: 128 битов, 192 бита и 256 битов. В зависимости от длины ключа используются обозначения AES-128, AES-192 и AES-256.

В AES количество раундов шифрования зависит от длины ключа. Для AES-128 выполняется десять раундов, для AES-192 — 12 раундов, а для AES-256 — 14 раундов. В этом разделе наше обсуждение ограничено длиной ключа 128 (т. е. AES-128), т. к. процесс почти не отличается от других вариантов AES. Единственное, что меняется — это *развертка ключа*, о который мы скажем чуть позже.

В отличие от DES, раунды шифрования AES являются итеративными и оперируют целым блоком данных по 128 битов в каждом раунде. Кроме того, в отличие от DES, дешифрование в AES не очень похоже на процесс шифрования.

Чтобы лучше понять этапы обработки в каждом раунде, рассмотрим 128-битный блок как 16 байтов, где отдельные байты расположены в матрице 4×4, называемой *массивом состояний* (рис. 2.6).

| | | | |
|---|---|---|---|
| Byte 0 | Byte 4 | Byte 8 | Byte 12 |
| Byte 1 | Byte 5 | Byte 9 | Byte 13 |
| Byte 2 | Byte 6 | Byte 10 | Byte 14 |
| Byte 3 | Byte 7 | Byte 11 | Byte 15 |

**Рис. 2.6.** Матрица 4×4: массив состояний

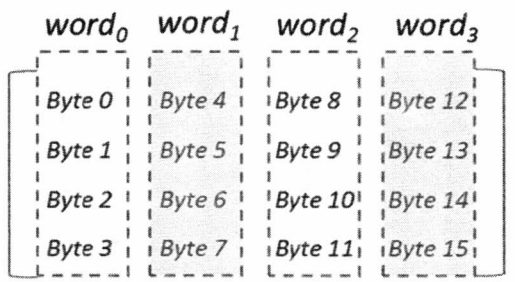



**Рис. 2.7.** Слова состояний

Обратите внимание: каждый раунд получает массив входных состояний и создает массив выходных состояний.

AES также использует специальный термин «слово» (word), который необходимо определить, прежде чем мы пойдем дальше. В то время как байт состоит из восьми битов, *слово состоит из четырех байтов*, т. е. 32-х битов. Четыре байта в каждом столбце массива состояний образуют 32-битные слова и могут называться *словами состояния*. Массив состояний при этом может выглядеть, как показано на рис. 2.7.

Также каждый байт может быть представлен двумя шестнадцатеричными числами. Например, если байт равен {00111010}, он может быть представлен как «3A» в шестнадцатеричной записи. Символ «3» представляет четыре левых бита «0011», а «A» — четыре правых бита «1010».

Теперь, если говорить о раунде в целом, обработка в каждом раунде происходит на уровне байтов и заключается в подстановке на уровне байтов, за которой следует перестановка на уровне слов, — следовательно, это сеть подстановки-перестановки. Мы подробнее расскажем об этом, когда будем обсуждать различные операции в каждом раунде. Общая схема процесса шифрования и дешифрования AES приведена на рис. 2.8.

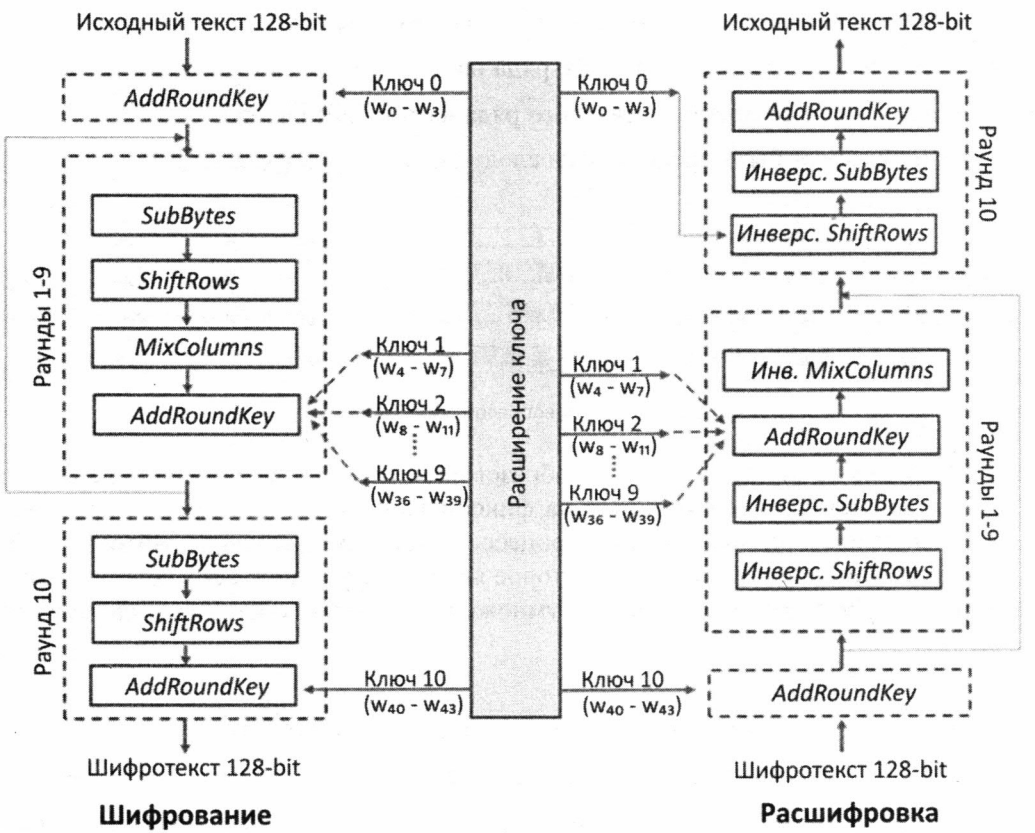



**Рис. 2.8.** Алгоритм AES

Если вы внимательно рассмотрите рис. 2.8, то заметите, что процесс дешифрования — это не просто противоположность шифрованию. Операции в раундах выполняются в другом порядке! Все операции раунда: *SubBytes*, *ShiftRows*, *MixColumns*, *AddRoundKey* — являются обратимыми. Также обратите внимание, что раунды носят итеративный характер. В раунде с 1 по 9 выполняются все четыре операции, а в последнем раунде исключается только операция *MixColumns*. Давайте теперь разберемся в назначении каждой операции, которая происходит в раунде.

- *SubBytes* — это операция замещения. Здесь каждый байт представлен в виде двух шестнадцатеричных цифр. В качестве примера возьмем байт {00111010}, представленный двумя шестнадцатеричными цифрами, скажем, {3A}. Чтобы найти значения подстановки, обратимся к таблице соответствия S-блока (таблица 16×16), чтобы найти соответствующее значение для строки {3} и столбца {A}. Таким образом, для {3A} соответствующее замещающее значение будет, скажем, {80}. Этот шаг обеспечивает нелинейность в шифре.
- *ShiftRows* — это шаг преобразования, основанный на матричном представлении массива состояний. Он состоит из следующих сменных операций:
  - первый ряд не смещаем, он остается как есть;
  - циклическое смещение второго ряда на один байт влево;
  - циклическое смещение третьего ряда на два байта влево;
  - циклическое смещение четвертого ряда на три байта влево.

  Операция может быть представлена следующей схемой (рис. 2.9).

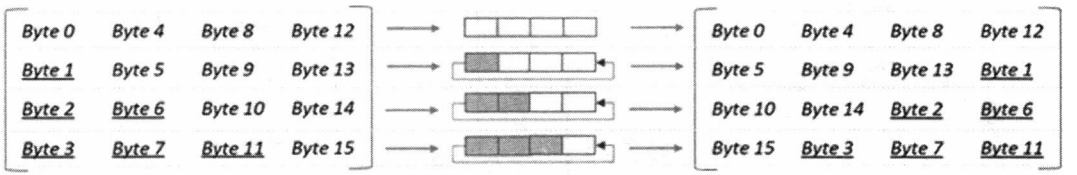



**Рис. 2.9.** Шаг преобразования *ShiftRows*

- *MixColumns* — это также шаг преобразования, при котором все четыре столбца матрицы состояния умножаются на фиксированный полином ($C_x$) и преобразуются в новые столбцы. В этом процессе с каждым исходном байтом столбца сопоставляется новое значение, которое является функцией всех четырех байтов в столбце. Это достигается путем умножения матрицы состояния, как показано на рис. 2.10.

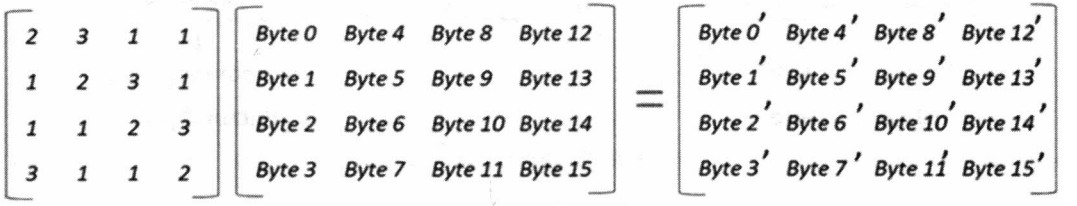



**Рис. 2.10.** Шаг преобразования *MixColumns*

Умножение матриц выполняется как обычно, но результаты логического умножения AND складываются по XOR. Давайте рассмотрим пример, чтобы понять, как это делается. ***Байт 0'*** рассчитывается так:

$$\textbf{\textit{Byte 0}}' = (2 \bullet \textbf{\textit{Byte 0}}) \oplus (3 \bullet \textbf{\textit{Byte 1}}) \oplus \textbf{\textit{Byte 3}} \oplus \textbf{\textit{Byte 4}}$$

Важно отметить, что шаг *MixColumns* вместе с шагом *ShiftRows* обеспечивают необходимый уровень диффузии шифра (информация из одного байта распределяется по нескольким байтам).

- *AddRoundKey* — это тоже шаг преобразования, при котором 128-битный ключ раунда побитово складывается по XOR со 128-ю битами состояния. Операция выполняется по столбцам, т. е. четыре байта столбца (слово состояния) складываются со словом ключа раунда. Аналогично тому, как раньше мы представили 128-битовый текстовый блок, 128-битовый ключ также должен быть представлен матрицей 4×4 (рис. 2.11).

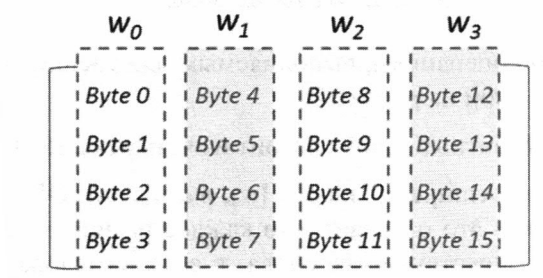



**Рис. 2.11. Представление 128-битового ключа матрицей 4×4**

Эта операция влияет на каждый бит состояния. Теперь вспомните, что есть десять раундов, и каждый раунд имеет свой собственный ключ. Поскольку перед началом раундов выполняется шаг *AddRoundKey*, фактически существует одиннадцать (10+1) операций *AddRoundKey*. В одном раунде все 128 битов подключа, т. е. все четыре слова подключа, складываются по XOR с 128-битовым блоком входных данных. Итак, для одиннадцати операций *AddRoundKey* нам нужно 44 ключевых слова: от $w_0$ до $w_{43}$. Вот почему 128-битный ключ должен пройти через операцию *расширения ключа*, к которой мы скоро приступим.

Обратите внимание, что слова-ключи $[w_0, w_1, w_2, w_3]$ складываются по XOR с начальным блоком ввода перед началом раундов обработки. Оставшиеся 40 слов-ключей: от $w_4$ до $w_{43}$ — используются по четыре слова за раз в каждом из десяти раундов.

## Расширение ключа AES

Алгоритм расширения ключа AES получает на вход 128-битовый ключ шифрования (ключ из четырех слов) и формирует из него набор из 44-х ключевых слов. Идея состоит в том, чтобы спроектировать эту систему таким образом, чтобы изменение единственного бита ключа значительно влияло на все ключи раундов.

Операция расширения ключа разработана таким образом, что каждая ключевая группа из четырех слов создает последующую группу из четырех слов по базису четырех слов. Это замысловатое предложение легко объяснить с помощью графического представления (рис. 2.12).

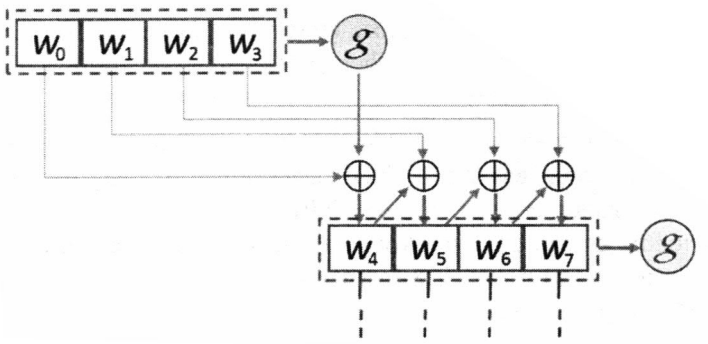



**Рис. 2.12.** Расширение ключа AES

Мы бегло пройдемся по операциям, выполняемым для расширения ключа, обращаясь к схеме, представленной на рис. 2.12.

- Первоначальный 128-битный ключ состоит из четырех слов $[w_0, w_1, w_2, w_3]$.
- Теперь взгляните на расширение ключа $[w_4, w_5, w_6, w_7]$. Обратите внимание, что $w_5$ зависит от $w_4$ и $w_1$. Это означает, что каждое расширенное слово зависит от непосредственно предшествующего слова, т. е. $w_{i-1}$, и слова, которое находится на четыре позиции назад, т. е. $w_{i-4}$. Убедитесь, что сказанное справедливо и для $w_6$. Как видите, здесь выполняется простая операция XOR.
- А как насчет $w_4$? Или любой другой позиции, кратной четырем, — например, $w_8$ или $w_{12}$? Для этих слов используется более сложная функция, обозначенная как $g$. Это функция, выполняемая за три шага. На первом шаге входной блок из четырех слов циклически сдвигается влево на один байт. Например, $[w_0, w_1, w_2, w_3]$ превращается в $[w_1, w_2, w_3, w_0]$. На втором шаге полученное слово из четырех байтов (например, $[w_1, w_2, w_3, w_0]$) берется в качестве входного значения, и к каждому байту применяется подстановка байтов с использованием S-блока. Затем, на третьем шаге, результат второго шага складывают по XOR с *раунд-константой*, обозначаемой как **Rcon[]**. Раунд-константа — это слово, в котором самые правые три байта всегда равны нулю, — например, $[x, 0, 0, 0]$. Назначение **Rcon[]** — просто выполнить XOR с крайним левым байтом ключевого слова после шага 2. Обратите внимание, что **Rcon[]** отличается для каждого раунда. Этим способом генерируется выходное значение комплексной функции $g$, которое затем складывается по XOR с $w_{i-4}$, чтобы получить $w_i$, где $i$ кратно 4. Так происходит расширение ключа в AES.

Массив выходных состояний последнего раунда реорганизуется обратно, образуя 128-битный блок зашифрованного текста. Процесс дешифрования происходит в обратном порядке, который изображен на рис. 2.8 справа. Мы хотели дать вам представление о том, как этот алгоритм работает на высоком уровне, и ограничимся только описанием процесса шифрования.

Алгоритм AES стандартизирован NIST (National Institute of Standards and Technology, Национальный институт стандартов и технологий). Алгоритму присуще огра-

ничение по времени обработки. Предположим, что вы отправляете всего лишь файл размером в 1 мегабайт (8388608 битов), зашифрованный с помощью AES. При использовании 128-битового алгоритма AES необходимо обработать 8388608/128 = = 65 536 блоков данных, с каждым из которых надо проделать процедуру из десяти раундов (см. рис. 2.8)! Эффективность AES может быть увеличена за счет параллельной обработки, но этот стандарт все равно не очень подходит, когда вы работаете с большими данными.

### Проблемы криптографии с симметричным ключом

В криптографии с симметричным ключом есть некоторые принципиальные ограничения. Вот некоторые из них:

- ключ должен быть передан отправителю и получателю до начала сеанса связи. Для этого требуется защищенный механизм создания и доставки ключа;
- отправитель и получатель должны доверять друг другу, т. к. они используют один и тот же симметричный ключ. Если злоумышленник взломал получателя, или получатель сознательно поделился ключом с кем-то еще, вся система будет скомпрометирована;
- большая сеть, скажем, из $n$ узлов, требует управления $n(n-1)/2$ парами ключей;
- рекомендуется постоянно менять ключ для каждого сеанса связи;
- часто для эффективного управления ключами необходима доверенная третья сторона, что само по себе является большой проблемой.

## 2.2.2. Криптографические хэш-функции

*Хэш-функции* — это математические функции, которые относятся к наиболее важным криптографическим примитивам и являются неотъемлемой частью структуры данных блокчейна. Они широко применяются во многих криптографических протоколах и приложениях информационной безопасности, таких как цифровые подписи и коды аутентификации сообщений (Message Authentication Code, MAC). Поскольку хэш-функция используется в криптографии с асимметричным ключом, мы обсудим ее здесь до того, как приступим к изучению асимметричной криптографии. Обратите внимание, что объяснения, приведенные в этом разделе, могут не соответствовать академическим учебникам, и немного смещены в сторону экосистемы блокчейна.

Криптографические хэш-функции — это особый класс хэш-функций, которые пригодны для криптографии, и мы ограничимся только этим применением. Таким образом, криптографическая хэш-функция — это односторонняя функция, которая преобразует входные данные произвольной длины и создает выходные данные фиксированной длины. Вывод обычно называется *хэш-значением*, *хэш-суммой* или *дайджестом* сообщения. Функцию в общем виде можно представить, как показано на рис. 2.13.

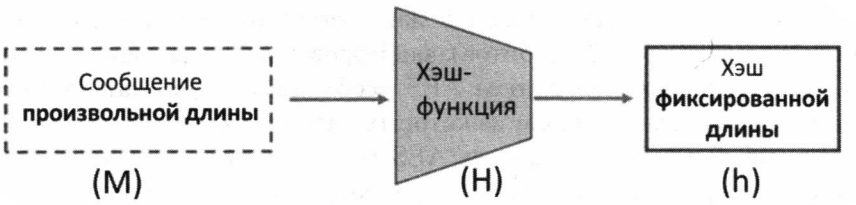



**Рис. 2.13. Хэш-функция в общем виде**

Чтобы хэш-функции служили задуманным целям и были пригодными для использования, они должны иметь следующие основные свойства:

- входными данными могут быть любые строки любого размера, но выходные данные имеют фиксированную длину — скажем, 256 или 512 битов;
- хэш-значение должно быть рационально вычисляемым для любого данного сообщения;
- функция является детерминированной в том смысле, что одни и те же исходные данные, поданные на вход одной и той же хэш-функции, каждый раз дают одно и то же хэш-значение;
- нереализуемо (хотя и *не невозможно*!) восстановить исходное сообщение из его хэш-значения любым способом за исключением перебора всех возможных вариантов сообщения;
- любое малейшее изменение в сообщении должно сильно влиять на выходной хэш, чтобы никто не мог соотнести новое значение хэша после внесенного изменения со старым.

Помимо упомянутых основных свойств, они также должны соответствовать следующим свойствам безопасности, которые следует рассматривать как криптографический протокол.

- **Стойкость к коллизиям.** Это означает, что невозможно найти два разных входа — скажем, $X$ и $Y$, которые хэшируют одно и то же значение:

$$\left.\begin{array}{l} X - H \rightarrow H(X) \\ Y - H \rightarrow H(Y) \end{array}\right\} H(X) \neq H(Y),$$

где $X \neq Y$.

- Это делает хэш-функцию $H(\,)$ стойкой к коллизиям, потому что никто не может найти такие $X$ и $Y$, что $H(X) = H(Y)$. Обратите внимание, что эта хэш-функция является функцией сжатия, т. к. она сжимает некоторый вход в выход фиксированного размера, который короче, чем вход. Входное пространство слишком велико (нечто произвольного размера) по сравнению с фиксированным пространством выхода. Если выходные данные представляют собой 256-битное хэш-значение, то выходное пространство может иметь максимум $2^{256}$ значений, и не более того. Отсюда следует, что теоретически коллизия возможна. Однако найти эту коллизию крайне сложно. В соответствии с теорией «парадокса дня рожде-

ния», мы можем сделать вывод, что можно найти коллизию, используя квадратный корень из выходного пространства. Таким образом, если перебрать $2^{128} + 1$ вариантов входных значений, то с вероятностью, близкой к 100%, мы обнаружим коллизию, но это невероятно большое число, которое делает поиск коллизии практически невозможным.

- Это свойство может быть полезным в большинстве онлайн-хранилищ, облачных файловых хранилищ, хранилищ больших двоичных объектов, магазинов приложений и тому подобных применениях. Свойство устойчивости к коллизиям широко используется для проверки целостности файлов. Например, владелец вычисляет хэш файла и загружает его в облачное хранилище. Позже, когда он скачивает файл обратно, он может снова вычислить хэш и сверить со старым значением, которое у него есть. Таким образом, если файл был изменен злоумышленником или поврежден из-за ошибок передачи, это сразу станет заметно по расхождению хэшей. Из-за свойства устойчивости к коллизии никто не может придумать другой файл или модифицировать исходный файл, который хэшируется с тем же значением, что и исходный файл.
- **Стойкость к восстановлению прообраза.** Это свойство означает, что невозможно вычислительными методами восстановить прообраз входных данных, т. е. невозможно найти вход $X$ по выходу $H(X)$. Поэтому данное свойство также можно назвать *скрывающим* свойством. Обратите пристальное внимание — здесь есть один тонкий нюанс. Когда $X$ может принимать произвольное и ничем не обусловленное значение, стойкость к восстановлению прообраза легко реализуема. Однако, если существует только ограниченное число значений, которые может принимать $X$, и это известно злоумышленнику, он может легко перебрать все возможные значения $X$ и найти, какое из них соответствует хэшированному результату.

  Рассмотрим пример. Научная лаборатория решила вычислить хэш исходных параметров для успешного результата эксперимента, чтобы любой злоумышленник, получивший доступ к базе данных экспериментов, не мог увидеть параметры, поскольку то, что хранится в системе, является выходным хэшем. Предположим, что может быть только три возможных набора параметров эксперимента, таких как OP111, OP112 и OP113, из которых успешен, скажем, только эксперимент с набором параметров OP112. Таким образом, лаборатория решает хэшировать данные, вычислить $H(OP112)$ и сохранить хэшированные значения в базе данных. Хотя злоумышленник не может восстановить OP112 из $H(OP112)$, он может просто перебрать и хэшировать все возможные результаты эксперимента — т. е. $H(OP111)$, $H(OP112)$ и $H(OP113)$, и увидеть, что только $H(OP112)$ соответствует хэшу, который хранится в базе данных. Такая система, безусловно, уязвима! Это означает, что когда входные данные для хэш-функции поступают из ограниченного пространства, они плохо защищены. Тем не менее, существует следующее решение этой проблемы:

  давайте возьмем входные данные $X$, которые распределены по ограниченному входному пространству (как и результаты эксперимента с несколькими

возможными параметрами, который мы только что обсудили). Если мы можем объединить входные данные с другим случайным входом — скажем, с $r$, который получается из распределения вероятностей с высокой минимальной энтропией, то будет трудно подобрать $X$ для $H(r||X)$. Здесь высокая минимальная энтропия означает, что значения $r$ распределены по входному пространству очень равномерно, и нет особого значения (или набора значений), которое имеет преимущество. Предположим, что число $r$ было выбрано из 256-битового распределения. Для злоумышленника, чтобы получить точное значение $r$, которое использовалось вместе с входными данными, существует вероятность успеха, равная $1/2^{256}$, которую практически невозможно реализовать. Единственный способ — перебрать все возможные значения этого распределения один за другим, что опять-таки практически невозможно. Значение $r$ также часто упоминается как **nonce**[3]. В криптографии **nonce** — это случайное число, которое может использоваться только один раз.

Давайте теперь обсудим, где это свойство стойкости к восстановлению прообраза может пригодиться. Это очень полезно при регистрации обязательства. Поясним на примере. Предположим, что вы участвовали в каком-то событии со ставками или в азартной игре. Скажем, вы должны подтвердить свой выбор, а также объявить его. Однако никто не должен заранее знать, на что вы делаете ставку, а вы сами не можете позже отрицать, на что ставили. Тогда вам следует использовать свойство устойчивости к восстановлению прообраза. Вы вычисляете хэш вашей ставки и объявляете его публично. Никто не может инвертировать хэш-функцию и выяснить, на что вы делаете ставку. Кроме того, позже вы не сможете сказать, что ваша ставка была другой, потому что если вы хэшируете другую ставку, ее хэш не будет соответствовать тому, что вы объявили публично. Как мы только что пояснили, для таких систем желательно использовать число **nonce**, чтобы устранить возможность раскрытия вашей ставки путем перебора ограниченного числа вариантов.

- **Стойкость к подбору второго прообраза.** Это свойство немного отличается от стойкости к коллизиям и означает, что при заданном входе $X$ и его хэше $H(X)$ невозможно найти такой вход $Y$, когда $H(X) = H(Y)$. В отличие от стойкости к коллизиям, где говорится о произвольной паре входных значений, здесь речь идет о входе $X$, который является определенным. Если хэш-функция устойчива к коллизиям, то она также защищена и от подбора второго прообраза.

Среди упомянутых свойств есть еще одно производное свойство, которое весьма полезно для биткойна. Давайте вглянем на это свойство с технической точки зрения и поймем, как биткойн использует его для майнинга, о чем говорится в *главе 3*. Название этого свойства — *дружественная головоломка* (puzzle friendliness). Это название подразумевает, что для поиска решения вычислительной задачи нет предопределенного пути (ярлыка), и единственный способ добраться до решения состоит в том, чтобы перебрать все возможные варианты во входном пространстве.

---

[3] Сокращение от английского **n**umber used **on**ly onc**e** — число, которое применяется однократно.

Давайте вспомним пример с числом **nonce**: $H(r||X) = Z$, где $r$ выбрано из распределения с высокой минимальной энтропией, $X$ — наше входное значение, объединенное с $r$, а $Z$ — выходное значение хэш-функции. Головоломка означает, что злоумышленнику чрезвычайно сложно найти путем перебора такое значение **nonce**, которое в сочетании с поддельными данными $Y$ тоже хэшируется в $Z$, т. е. $H(r'||Y) = Z$, где число $r'$ является частью ввода и выбрано таким же случайным способом, что и $r$. Отсюда следует, что когда часть входных данных существенно рандомизирована, практически невозможно взломать хэш-функцию с помощью быстрого решения. Единственный способ — проверить все возможные случайные значения.

В предыдущем примере, если $Z$ является n-битовым выходом, тогда он принимает только одно значение из $2^n$ возможных значений. Обратите внимание, что часть входных данных — скажем, $r$, относится к распределению с высокой минимальной энтропией, которое должно быть добавлено к вашим исходным данным $X$. Теперь перейдем к интересной части устройства головоломки. Допустим, $Z$ является n-битовым выходом и представляет собой *набор* из $2^n$ возможных значений, а не просто одно значение. Вас попросят найти такое значение $r$, что при добавлении к исходным данным $X$ выходной хэш $Z$ попадает в этот выходной набор из $2^n$ значений, — в этом случае головоломка считается решенной. Идея состоит в том, чтобы перебирать все возможные значения $r$ до тех пор, пока хэш не попадет в заданный диапазон $Z$. Обратите внимание, что размер $Z$ ограничивает пространство вывода меньшим набором из $2^n$ возможных значений. Чем меньше выходное пространство, тем сложнее проблема. Очевидно, что если диапазон выходных значений широкий, то в него легче попасть, а если диапазон весьма узкий, то угадать значение в нем очень сложно. В этом прелесть наличия во входных данных хэш-функции числа $r$, называемого **nonce**. Какое бы случайное значение $r$ вы ни выбирали, оно будет объединено с $X$ и будет проходить через одну и ту же хэш-функцию снова и снова, пока вы не обнаружите правильное значение $r$, которое удовлетворяет требуемому диапазону для $Z$. Нет никакого иного метода решения головоломки, кроме перебора всех возможных значений $r$!

Обратите внимание, что для n-битового выходного значения хэш-функции требуется среднее вычислительное усилие $2^n$, чтобы преодолеть защиту от восстановления первого и второго прообраза, а стойкость к коллизии составляет $2^n/2$.

Мы обсудили различные фундаментальные и защитные особенности хэш-функций. В следующих разделах мы рассмотрим некоторые важные хэш-функции и углубимся в них по мере необходимости.

## Обзор различных хэш-функций

Одной из самых старых хэш-функций является хэш-функция MD4. Она принадлежит к семейству дайджестов сообщений (MD, Message Digest). Другими членами семейства MD являются MD5 и MD6, а также вариации MD4, такие как RIPEMD. Семейство алгоритмов MD создает 128-битовый дайджест сообщения, используя блоки длиной 512 битов. Дайджесты MD широко применялись в качестве контрольных сумм для проверки целостности данных. Многие файловые серверы или

репозитории приложений сообщали предварительно вычисленную контрольную сумму MD5, по которой пользователи могли проверить подлинность загружаемого файла. Однако в семействе MD было обнаружено множество уязвимостей, и оно устарело.

Другим семейством хэш-функций является семейство SHA (Secure Hash Algorithm, алгоритм защищенного хэширования). Семейство состоит из четырех основных алгоритмов, таких как SHA-0, SHA-1, SHA-2 и SHA-3. Первый алгоритм этого семейства сначала был назван SHA, но затем появились новые версии с исправлениями и обновлениями безопасности, поэтому его переименовали в SHA-0. Потом было обнаружено, что он имеет серьезную уязвимость, и использование этого алгоритма прекратили.

Позже, в качестве замены для SHA-0, был предложен SHA-1. Его разработало Агентство национальной безопасности (АНБ) для использования в алгоритме цифровой подписи (DSA). У SHA-1 имелся дополнительный вычислительный шаг, который решал проблему уязвимости. И SHA-0, и SHA-1 представляли собой 160-битовые хэш-функции, которые использовали 512-битовые размеры блоков. SHA-1 довольно часто задействуется во многих инструментах безопасности и интернет-протоколах, таких как SSL, SSH, TSL и др. Он также применялся в системах контроля версий — таких как Mercurial, Git — для проверки согласованности версий, а не для целей безопасности. Позже, примерно в 2005 году, в нем были обнаружены криптографически слабые места, и после 2010 года SHA-1 объявили устаревшим.

А алгоритмы SHA-2 и SHA-3 мы подробно рассмотрим в следующих разделах.

## SHA-2

Этот алгоритм принадлежит к семейству SHA, но на самом деле тоже представляет собой семейство хэш-функций из множества вариантов SHA — таких как SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 и SHA-512/256. SHA-256 и SHA-512 являются базовыми хэш-функциями, а другие варианты представляют собой производные от них. Семейство хэш-функций SHA-2 широко используется в таких приложениях, как SSL, SSH, TSL, PGP, MIME и им подобных.

SHA-224 — это усеченная версия SHA-256 с другим начальным значением или *вектором инициализации* (IV, Initialization Vector). Варианты SHA с разной степенью усечения могут давать на выходе хэши одинаковой длины, поэтому в разных вариантах SHA применяются разные векторы инициализации, чтобы иметь возможность их различать.

Теперь вернемся к вычислению SHA-224. Это двухэтапный процесс. Сначала вычисляется хэш SHA-256, но с другим вектором инициализации, не таким, который используется в SHA-256 по умолчанию. Затем полученное 256-битовое значение хэша усекается до 224-битового. Обычно оставляют 224 бита слева, но окончательный выбор остается за вами.

SHA-384 является усеченной версией SHA-512, точно так же, как SHA-224 является усеченной версией SHA-256. Вам интересно, почему существует понятие *усечение*?

Отметим, кстати, что усечение не ограничивается теми версиями, которые мы только что упомянули, и могут быть другие варианты. Основными причинами усечения могут быть следующие:

- некоторым приложениям нужен дайджест сообщения определенной длины, отличной от стандартной;
- независимо от используемого нами варианта SHA-2, мы можем выбрать уровень усечения в зависимости от того, какое свойство безопасности мы хотим поддерживать. Например, учитывая сегодняшнее состояние вычислительной мощности, когда необходима устойчивость к коллизии, мы должны сохранить хэш с длиной не менее 160 битов, а когда требуется только устойчивость к восстановлению прообраза, мы должны сохранить как минимум 80 битов. Защита от коллизии ухудшается с усечением, и его следует выбирать так, чтобы было невозможно найти коллизию вычислительными методами за разумное время;
- усечение также помогает поддерживать обратную совместимость со старыми приложениями.

Эффективность этих алгоритмов связана с тем, что алгоритм SHA-256 основан на 32-битовом слове, а SHA-512 — на 64-битовом. Таким образом, на оборудовании с 64-битовой архитектурой функции SHA-512 и все ее усеченные варианты могут быть вычислены быстрее (и с более высоким уровнем безопасности) по сравнению с SHA-1 или другими вариантами SHA-256.

Таблица 2.3 взята из статьи NIST и кратко представляет различные свойства алгоритмов SHA-1 и SHA-2.

***Таблица 2.3.*** *Свойства алгоритмов хэш-функций SHA-1 и SHA-2*

| Алгоритм | Размер сообщения (битов) | Размер блока (битов) | Размер слова (битов) | Размер дайджеста (битов) |
|---|---|---|---|---|
| SHA-1 | $<2^{64}$ | 512 | 32 | 160 |
| SHA-224 | $<2^{64}$ | 512 | 32 | 224 |
| SHA-256 | $<2^{64}$ | 512 | 32 | 256 |
| SHA-384 | $<2^{128}$ | 1024 | 64 | 384 |
| SHA-512 | $<2^{128}$ | 1024 | 64 | 512 |
| SHA-512/224 | $<2^{128}$ | 1024 | 64 | 224 |
| SHA-512/256 | $<2^{128}$ | 1024 | 64 | 256 |

Как правило, желательно не обрезать хэш, когда в этом нет необходимости. Некоторые хэш-функции допускают усечение, а некоторые — нет, и это также зависит от того, в каком контексте вы их используете.

## SHA-256 и SHA-512

Как уже упоминалось, SHA-256 относится к семейству хэш-функций SHA-2, и именно этот алгоритм используется в протоколе Bitcoin. Как следует из названия, он выдает 256-битовое хэш-значение. Таким образом, в соответствии с парадоксом дня рождения, он обеспечивает безопасность с уровнем сложности $2^{128}$ битов.

Напомним, что хэш-функции принимают входные данные произвольной длины и выводят значение фиксированного размера. Входные данные произвольной длины не подаются на функцию сжатия в исходном виде, а предварительно разбиваются на блоки фиксированной длины. Это означает, что нам нужен итерационный метод, который будет разбивать входное сообщение на блоки заданной длины и последовательно их перебирать, формируя в итоге выходной хэш фиксированной длины. Существуют различные способы построения хэш-функций — такие как *конструкция Меркла — Дамгарда, древовидная конструкция* и *конструкция криптографической губки*. Доказано, что если базовая функция сжатия устойчива к коллизиям, то общая хэш-функция любой конструкции также должна быть устойчивой к коллизиям.

Метод построения хэша, который использует SHA-256, — это конструкция Меркла — Дамгарда, схематически изображенная на рис. 2.14.

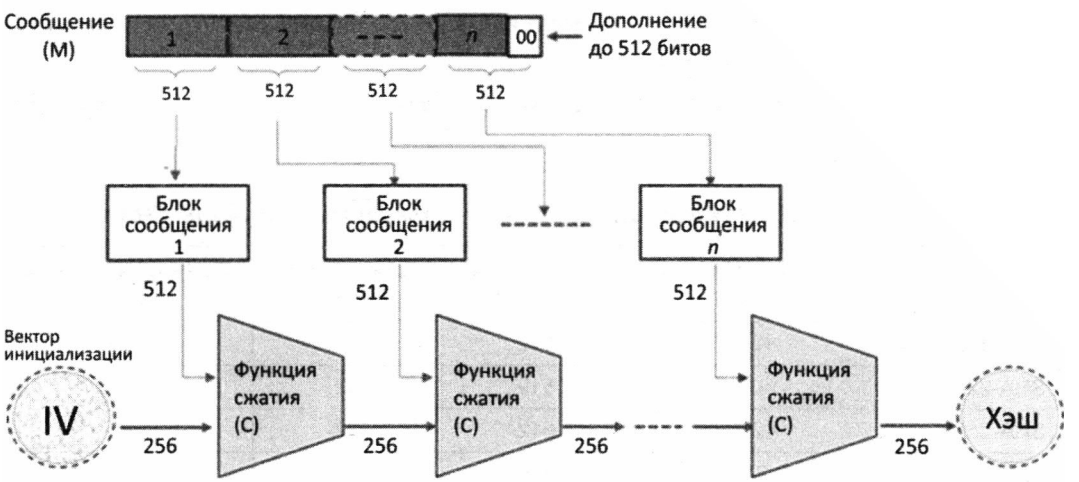



**Рис. 2.14. Конструкция Меркла — Дамгарда для алгоритма SHA-256**

Для вычисления окончательного значения хэша выполняются следующие шаги в соответствии со схемой, представленной рис. 2.14:

- как можно видеть, сообщение сначала делится на 512-битовые блоки. Когда сообщение не является кратным 512 битам (обычно так и бывает), в последнем блоке не хватает битов, поэтому он дополняется до 512 битов;
- 512-битовые блоки дополнительно делятся на 16 блоков 32-битных слов ($16 \times 32 = 512$);

- каждый блок проходит через 64 цикла раунд-функций, в которых каждое 32-битное слово проходит через серию операций. Раунд-функции представляют собой сочетание общих логических операций, таких как XOR, AND, OR, NOT, битовый сдвиг влево/вправо и некоторых других. Здесь мы не будем вдаваться в эти детали.

Подобно SHA-256, алгоритм SHA-512 также использует конструкцию Меркла — Дамгарда. Основное различие состоит в том, что в SHA-512 применяется 80 циклов раунд-функций, а длина слова составляет 64 бита. Размер блока в SHA-512 составляет 1024 бита. Этот блок дополнительно делится на 16 блоков 64-битовых слов. Длина выходного дайджеста сообщения составляет 512 битов, т. е. восемь блоков 64-битовых слов.

Пока SHA-512 набирал обороты и начинал использоваться во многих приложениях, некоторые дальновидные люди начали переход на SHA-3. Алгоритм SHA-3 — это просто другой подход к SHA-256 или SHA-512, но зато его можно настраивать. Мы расскажем некоторые подробности о SHA-3 в следующих разделах.

## RIPEMD

Хэш-функция RIPEMD (RACE Integrity Primitives Evaluation Message Assessment) является вариантом хэш-функции MD4 с почти такими же конструктивными решениями. Мы кратко обсудим эту функцию, поскольку она используется в протоколе Bitcoin.

Оригинальная функция RIPEMD была 128-битовой, позже был разработан вариант RIPEMD-160. Существуют также 128-, 256- и 320-битовые версии этого алгоритма, называемые RIPEMD-128, RIPEMD-256 и RIPEMD-320 соответственно, но мы ограничим наше обсуждение наиболее популярным и широко используемым RIPEMD-160.

Итак, RIPEMD-160 — это криптографическая хэш-функция, в которой сжатие основано на конструкции Меркла — Дамгарда. Входное сообщение делится на 512-битовые блоки. Если длина входного сообщения не кратна 512, то последний блок дополняется до 512 битов. Выходной 160-битовый хэш обычно представляют в виде 40-значного шестнадцатеричного числа.

Функция сжатия состоит из 80 этапов, состоящих из двух параллельных веток по пять раундов из 16 шагов в каждом ($5 \times 16 = 80$). Функция сжатия работает с шестнадцатью 32-битовыми словами (512-битовыми блоками).

**ПРИМЕЧАНИЕ.** Для генерации адресов Bitcoin использует хэши SHA-256 и RIPEMD-160. При этом RIPEMD-160 задействуется для дальнейшего сокращения значения хэш-значения SHA-256 до 160 битов.

## SHA-3

В 2015 году NIST утвердил алгоритм Keccak (произносится как «кет-чак») в качестве стандарта следующего поколения SHA-3. Заметим, что цель нового стандар-

та — не заменить стандарт SHA-2, а дополнить его и сосуществовать с ним, хотя в некоторых ситуациях действительно имеет смысл выбрать SHA-3 вместо SHA-2.

Поскольку и SHA-1, и SHA-2 были основаны на конструкции Меркла — Дамгарда, в новом стандарте требовался другой подход к хэш-функции. Поэтому одним из критериев, установленных NIST, стал отказ от использования конструкции Меркла — Дамгарда. Новый алгоритм следовало избавить от недостатков и ограничений конструкции Меркла — Дамгарда, таких как мультиколлизия. Алгоритм Keccak, который стал стандартом SHA-3, использовал другой метод, называемый *конструкцией криптографической губки*.

Для обеспечения обратной совместимости требовалось, чтобы SHA-3 мог выдавать выходные данные переменной длины, такой как 224, 256, 384 и 512 битов, а также любой другой произвольной длины. Таким образом, внутри SHA-3 образовалось семейство криптографических хэш-функций, таких как SHA3-224, SHA3-256, SHA3-384, SHA3-512, и двух функций расширяемого вывода (eXtendable-Output Functions, XOF), которые называются SHAKE128 и SHAKE256. Кроме того, SHA-3 должен был иметь настраиваемый параметр (вместимость), чтобы обеспечить компромисс между безопасностью и производительностью. Поскольку SHAKE128 и SHAKE256 являются расширяемыми функциями XOF, их вывод может быть расширен до любой желаемой длины, отсюда и название.

На рис. 2.15 показана структура SHA-3 (алгоритм Keccak) в общем виде.

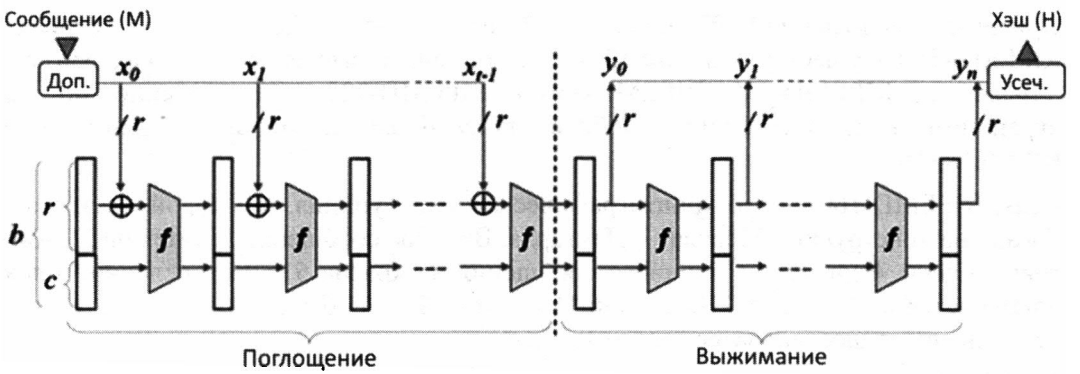



**Рис. 2.15. Конструкция криптографической губки в SHA-3**

Последовательность шагов для SHA-3:

- как можно видеть на рис. 2.15, сообщение сначала делится на блоки ($x_i$) размером $r$ битов. Если входные данные не кратны $r$ битам, то требуется дополнение. Если вас интересует вопрос размерности $r$, не волнуйтесь, мы скоро до него доберемся. Теперь давайте сосредоточимся на том, как происходит это дополнение. Для блока сообщений $x_i$, который не кратен $r$ и содержит некоторое сообщение **M**, заполнение происходит так:

$$x_i = \mathrm{M} \,||\, \mathrm{P}\, 1\{0\}^*1,$$

где P — это заранее заданная битовая строка, за которой следует последовательность вида 1{0}*1, что означает начальные и конечные единицы и некоторое количество нулей между ними (этих нулевых битов может и не быть), что, собственно, и делает $x_i$ кратным $r$. В табл. 2.4 приведены различные значения P;

***Таблица 2.4.** Дополнение в различных вариантах SHA-3*

| Режим | Длина выхода | P | 1{0}*1 |
|---|---|---|---|
| SHA3-224 | 224 | 11001 | 1{0}*1 |
| SHA3-256 | 256 | 11101 | 1{0}*1 |
| SHA3-384 | 384 | 11001 | 1{0}*1 |
| SHA3-512 | 512 | 11101 | 1{0}*1 |
| Переменная длина (XOF) | Любая | 1111 | 1{0}*1 |

- как можно видеть на рис. 2.15, в конструкции криптографической губки SHA-3 есть две основные фазы: первая — это фаза «поглощения» для входа, а вторая — фаза «выжимания» для выхода. На этапе поглощения блоки сообщений ($x_i$) проходят через различные операции алгоритма, а на этапе выжимания вычисляется выходное сообщение настраиваемой длины. Обратите внимание, что для обеих этих фаз используется одна и та же функция, называемая «Keccak-*f*»;
- для вычислений SHA3-224, SHA3-256, SHA3-384, SHA3-512, которые фактически являются заменой SHA-2, используются только начальные биты первого выходного блока $y_0$ с требуемым уровнем усечения;
- SHA-3 спроектирован таким образом, чтобы его можно было настраивать с учетом уровня безопасности, входных и выходных размеров с помощью параметров настройки;
- как можно видеть на рис. 2.15, $b$ представляет ширину слова состояния, и необходимо, чтобы соблюдалось условие $r + c = b$. Кроме того, $b$ зависит от степенного показателя $\ell$, такого, что $b = 25 \times 2^{\ell}$;
- поскольку $\ell$ может принимать значения от 0 до 6, $b$ может иметь ширину {25, 50, 100, 200, 400, 800 и 1600}. Желательно не использовать наименьшие два значения $b$ на практике, т. к. они предназначены только для исследования и выполнения криптоанализа алгоритма;
- в уравнении $r + c = b$ параметр $r$ — это значение, которое мы использовали для предварительной обработки сообщения и деления на блоки длиной $r$. Оно называется *битрейтом*[4] (bit rate). В свою очередь, параметр $c$ называется *емкостью*, которая должна вычисляться и удовлетворять условию:

$$r + c = b \in \{25, 50, 100, 200, 400, 800, 1600\}$$

[4] В блочной криптографии *битрейтом* называют количество битов в блоке. Не путайте это понятие с битрейтом цифровой аудио- или видеозаписи — скоростью передачи сигнала, измеряемой в битах в секунду.

Таким образом, $r$ и $c$ используются в качестве параметров настройки для компромисса между безопасностью и производительностью;

- для SHA-3 значение показателя степени $\ell$ фиксированное и равно шести, поэтому значение $b$ равно 1600 битам. Для этого заданного $b = 1600$ допустимы два значения битрейта: $r = 1344$ и $r = 1088$. Это приводит к двум различным значениям $c$. Таким образом, для $r = 1344$ $c = 256$ и для $r = 1088$ $c = 512$;
- давайте теперь посмотрим на ядро этого алгоритма, т. е. Keccak-*f*, который также называется «Перестановка Keccak-*f*». В каждом Keccak-*f* есть $n$ раундов, где $n$ вычисляется как: $n = 12 + 2\ell$. Поскольку для SHA-3 значение $\ell = 6$, в каждом Keccak-*f* будет 24 раунда. Каждый раунд принимает $b$ битов ($b = r + c$) и выдает то же количество битов $b$ на выходе;
- в каждом раунде вход $b$ называется *состоянием*. Этот массив состояний $b$ может быть представлен в виде трехмерного (3D) массива $b = (5 \times 5 \times w)$, где размер слова $w = 2\ell$. Итак, $w = 64$ бита, что означает $5 \times 5 = 25$ слов по 64 бита каждое. Напомним, что SH = 6 для SHA-3, поэтому $b = 5 \times 5 \times 64 = 1600$. Трехмерный массив может быть представлен, как показано на рис. 2.16;

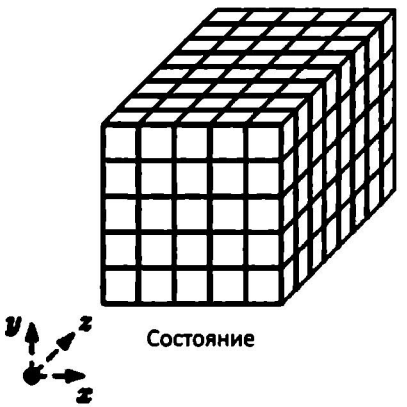



**Рис. 2.16.** Представление трехмерного массива состояния SHA-3

- каждый раунд состоит из пяти последовательных шагов, и массив состояний обрабатывается на каждом из этих шагов (рис. 2.17);

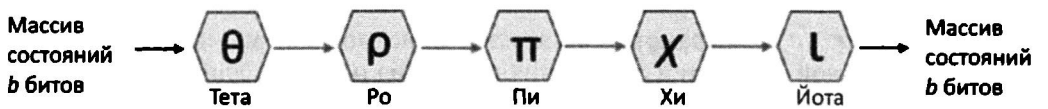



**Рис. 2.17.** Пять шагов каждого раунда SHA-3

- не вдаваясь в подробности каждого из пяти шагов, давайте бегло рассмотрим, что они делают на высоком уровне:
  - шаг *Тета* (θ) — выполняет операцию XOR, чтобы обеспечить минимальную диффузию;

- шаг *Ро* (ρ) — выполняет побитовый циклический сдвиг каждого из 25 слов;
- шаг *Пи* (π) — выполняет перестановку каждого из 25 слов;
- шаг *Хи* (χ) — на этом этапе биты слова заменяются путем объединения битов с битами двух следующих слов ряда;
- шаг *Йота* (ι) — сложение по XOR раунд-константы и слова состояния, чтобы нарушить симметрию;

♦ последний раунд Keccak-*f* выдает выходное сообщение $y_0$, которого достаточно для режима замены SHA-2, т. е. выходное сообщение с 224, 256, 384 и 512 битами. Обратите внимание, что младшие значащие биты $y_0$ используются для формирования вывода желаемой длины. В случае вывода переменной длины, наряду с $y_0$, также могут использоваться другие выходные биты: $y_1$, $y_2$, $y_3$ и т. д.

Когда дело доходит до прикладной реализации SHA-3, обнаруживается, что его производительность хороша в программной реализации (хотя и не так хороша, как в SHA-2) и превосходна в аппаратной реализации (лучше, чем в SHA-2).

## Применение хэш-функций

Криптографические хэш-функции широко применяются в различных ситуациях. Приведем несколько примеров:

♦ хэш-функции используются при проверке целостности и подлинности информации.

♦ хэш-функции также можно использовать для индексации данных в хэш-таблицах. Это может ускорить процесс поиска. Если вместо целых данных мы ищем хэши (предполагая, что хэш-значение намного короче по сравнению с целыми данными), то очевидно, что такой поиск должен быть быстрее;

♦ хэш-функции можно использовать для безопасной аутентификации пользователей без локального хранения паролей. Будет совершенно разумно не хранить пароли на сервере в открытом виде. Если злоумышленник взломает сервер, он не сможет восстановить пароли из сохраненных хэшей. Каждый раз, когда пользователь пытается войти в систему, хэш введенного пароля вычисляется заново и сопоставляется с сохраненным хэшем. Это безопасно, не так ли?

♦ поскольку хэш-функции являются односторонними функциями, их можно использовать для реализации генератора псевдослучайных чисел;

♦ Bitcoin использует хэш-функции в качестве алгоритма доказательства работы (PoW). Мы подробно расскажем об этом, когда дойдем до главы о биткойнах;

♦ Bitcoin также использует хэш-функции для генерации адресов в целях повышения безопасности и конфиденциальности;

♦ двумя наиболее важными приложениями являются цифровые подписи и коды аутентификации сообщений на основе хэшей (HMAC).

Понимая работу и свойства хэш-функций, можно придумать и другие способы применения хэш-функций.

**ПРИМЕЧАНИЕ.** Инженерная рабочая группа по Интернету (IETF) в 1999 году приняла версию 3.0 протокола SSL (SSLv3), затем переименовала его в протокол Transport Layer Security (TLS) версии 1.0 (TLSv1) и определила его в RFC 2246. Протокол TLSv1 обеспечивает обратную совместимость с SSLv3.

## Примеры кода хэш-функций

В листинге 2.1 приведены примеры кода различных хэш-функций[5]. Этот раздел предназначен для того, чтобы дать вам общее представление о прикладном применении хэш-функций в программах. Примеры кода написаны на Python, но будут выглядеть очень похоже и на других языках. Вам просто нужно найти подходящие библиотечные функции.

**Листинг 2.1. Примеры практического использования различных хэш-функций**

```
# -*- coding: utf-8 -*-
import hashlib
# hashlib module is a popular module to do hashing in python

#Конструкторы md5(), sha1(), sha224(), sha256(), sha384() and sha512() представленных в hashlib
md=hashlib.md5()
md.update("The quick brown fox jumps over the lazy dog")
print md.digest()
print "Digest Size:", md.digest_size, "\n", "Block Size: ", md.block_size

# Сравнение дайджестов SHA224, SHA256,SHA384,SHA512
print "Digest SHA224", hashlib.sha224("The quick brown fox jumps over the lazy dog").hexdigest()
print "Digest SHA256", hashlib.sha256("The quick brown fox jumps over the lazy dog").hexdigest()
print "Digest SHA384", hashlib.sha384("The quick brown fox jumps over the lazy dog").hexdigest()
print "Digest SHA512", hashlib.sha512("The quick brown fox jumps over the lazy dog").hexdigest()
# Все дайджесты уникальные

# Пример 160-битового хэша RIPEMD160
h = hashlib.new('ripemd160')
h.update("The quick brown fox jumps over the lazy dog")
h.hexdigest()

#Алгоритм генерации ключей
#Исходный алгоритм хэширования уязвим к атаке перебором
#Алгоритм генерации ключей доработан для повышения защищенности
import hashlib, binascii
algorithm='sha256'
password='HomeWifi'
```

[5] Это не законченные программы, а фрагменты полезного кода (сниппеты), которые вы можете использовать в своих программах.

```
salt='salt' # salt это случайные данные, которые добавляются в одностороннюю функцию
nu_rounds=1000
key_length=64 #dklen это длина генерируемого ключа
dk = hashlib.pbkdf2_hmac(algorithm,password, salt, nu_rounds, dklen=key_length)
print 'derived key: ',dk
print 'derived key in hexadeximal :', binascii.hexlify(dk)

#Пример хэширования
import hashlib

input = "Sample Input Text"
for i in xrange(20):
        # добавляем итератор к исходному тексту
        input_text = input + str(i)
        # печатаем входное сообщение и его хэш
        print input_text, ':', hashlib.sha256(input_text).
        hexdigest()
```

## 2.2.3. MAC и HMAC

HMAC — это разновидность MAC (Message Authentication Code, код аутентификации сообщения). Как следует из названия, назначением MAC является обеспечение аутентификации сообщений с использованием симметричного ключа и целостности сообщений с использованием хэш-функций. Итак, отправитель передает MAC вместе с сообщением для получателя, чтобы тот смог проверить сообщение и доверять ему. Получатель уже имеет ключ K (поскольку используется криптография с симметричным ключом, отправитель и получатель уже согласовали его). Получатель просто использует ключ K, чтобы вычислить MAC сообщения и сравнить его с MAC, который был отправлен вместе с сообщением.

В простейшем виде, MAC = H (ключ || сообщение). По сути, HMAC — это способ превращения хэш-функций в MAC. В HMAC хэш-функции могут применяться несколько раз вместе с ключом и его производными ключами. HMAC широко используются в системах на основе RFID, TLS и других. В протоколе SSL/TLS HMAC служит для того, чтобы клиент и сервер могли удостовериться, что данные не были изменены во время передачи. Давайте рассмотрим несколько важных и широко применяемых стратегий MAC:

- *MAC-then-Encrypt* — вычисление MAC для открытого текста, добавление его к данным и последующее шифрование всего вместе. Эта схема не обеспечивает целостность зашифрованного текста. На принимающей стороне сначала выполняется дешифрование сообщения, а затем проверяется его целостность. Однако такой подход обеспечивает целостность открытого текста. TLS использует эту схему MAC для обеспечения безопасности сеанса связи «клиент-сервер»;
- *Encrypt-and-MAC* — вычисление MAC для открытого текста, шифрование, а затем добавление MAC в конце зашифрованного сообщения. В этой схеме тоже

можно гарантировать целостность открытого текста, но не зашифрованного сообщения, что оставляет возможность для некоторых атак. Но, в отличие от предыдущей схемы, этот вариант более устойчив к атакам. Такую схему MAC использует протокол SSH (Secure Shell);

- *Encrypt-then-MAC* — для этого метода необходимо сначала зашифровать открытый текст, а затем вычислить MAC для зашифрованного текста. Этот MAC зашифрованного текста затем добавляется к самому зашифрованному тексту. Такая схема обеспечивает целостность зашифрованного текста, поэтому можно сначала проверить целостность и, если она соблюдена, затем расшифровать сообщение. Схема легко отфильтровывает недействительные шифротексты, что во многих случаях делает ее более эффективной. Кроме того, поскольку MAC вычислен для зашифрованного сообщения, он никоим образом не раскрывает информацию о незашифрованном тексте. Это, пожалуй, лучшая из трех и наиболее широко применяемая схема. Используется в протоколе IPsec.

## 2.2.4. Криптография с асимметричным ключом

Криптография с асимметричным ключом, также известная как *криптография с открытым ключом*, является революционной концепцией, представленной Диффи и Хеллманом[6]. С помощью этой технологии они решили проблему распространения ключей в симметричной криптографической системе, введя цифровые подписи. Учтите, что криптография с асимметричным ключом не устраняет необходимость в криптографии с симметричным ключом. Они обычно дополняют друг друга — преимущества одной могут компенсировать недостатки другой.

Давайте рассмотрим практический сценарий, чтобы понять, как работает криптография с открытым ключом. Предположим, что Алиса хочет отправить конфиденциальное сообщение Бобу, чтобы никто, кроме Боба, не мог понять смысл сообщения. Тогда для этого потребуются следующие шаги (рис. 2.18):

- Алиса — отправитель:
  - шифрует текстовое сообщение $m$, используя алгоритм шифрования $E$ и *открытый ключ Боба* $Puk_B$, чтобы подготовить зашифрованный текст $c$;
  - $c = E(Puk_B, m)$;
  - отправляет зашифрованный текст $c$ Бобу.
- Боб — получатель:
  - расшифровывает зашифрованный текст $c$, используя алгоритм дешифрования $D$ и *свой закрытый* ключ $Prk_B$, чтобы получить исходный открытый текст $m$;
  - $m = D(Prk_B, c)$.

---

[6] Уитфилд Диффи, Мартин Хеллман — сотрудники Стэнфордского университета, в 1976 году представили реализацию своей идеи асимметричной криптосистемы. В 2016 году удостоены премии Тьюринга — аналога Нобелевской премии для программистов.

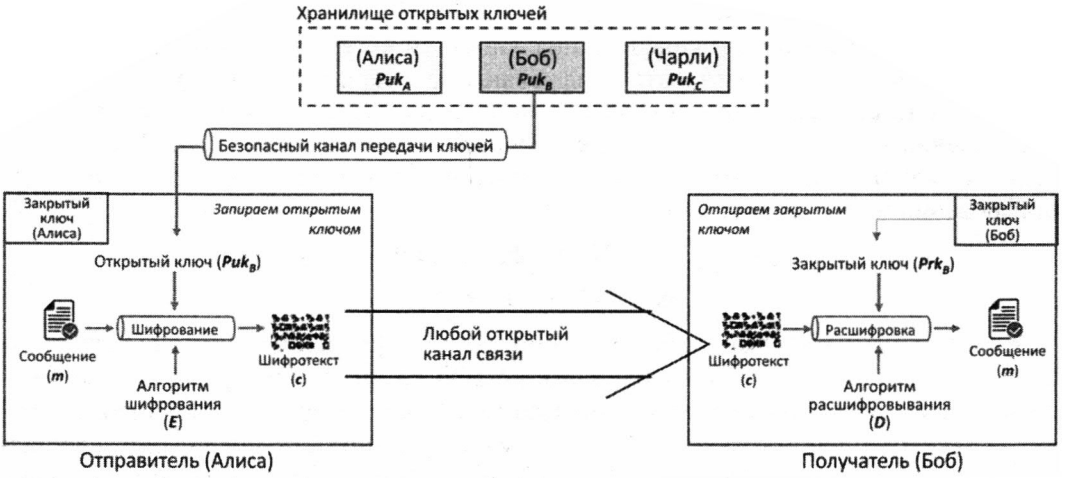



**Рис. 2.18. Шифрование с открытым ключом**

Открытый ключ должен храниться в общедоступном хранилище и быть доступен всем, а закрытый ключ является тщательно охраняемым секретом.

Криптография с открытым ключом также обеспечивает способ аутентификации. Получатель Боб может проверить подлинность источника сообщения ***m***. Давайте посмотрим, как работает аутентификация источника. Для этого обратимся к схеме, представленной на рис. 2.19.

В данном случае сообщение было зашифровано с использованием *закрытого ключа Алисы*, поэтому можно убедиться, что оно пришло именно от Алисы[7]. По сути,

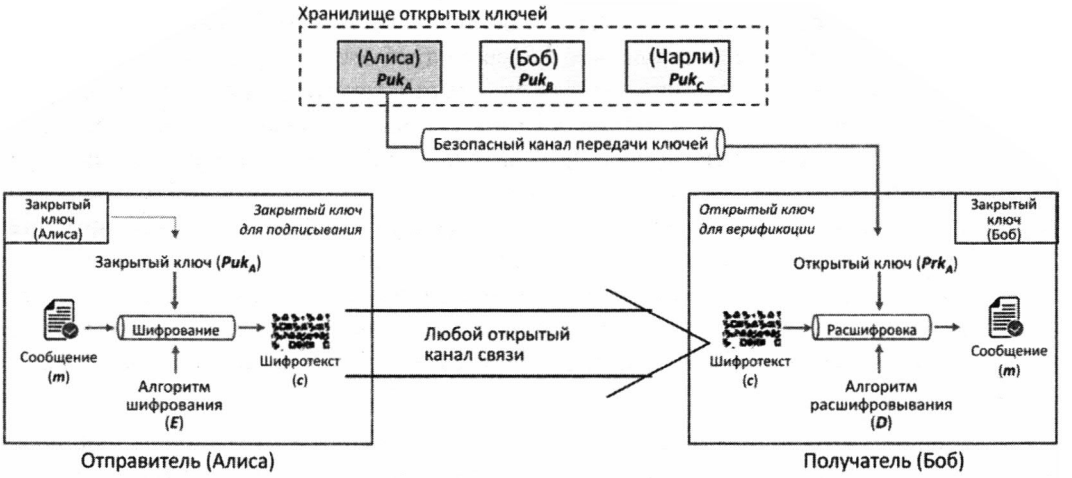



**Рис. 2.19. Асимметричная криптография для аутентификации**

[7] Доказательством подлинности отправителя служит тот факт, что Боб может расшифровать сообщение только открытым ключом Алисы, ничей другой ключ не подойдет.

все сообщение является цифровой подписью Алисы. Впрочем, на практике нам одновременно требуются и конфиденциальность, и аутентификация. Для этого необходимо *дважды* использовать шифрование с открытым ключом. Сообщение должно быть сначала зашифровано с помощью закрытого ключа отправителя для обеспечения цифровой подписи. Затем сообщение должно быть зашифровано открытым ключом получателя для обеспечения конфиденциальности. Операции шифрования и расшифровки можно записать так:

$$c = E[Puk_B, E(Prk_A, m)]$$

$$m = D[Puk_A, D(Prk_B, c)]$$

Как видите, расшифровка происходит в обратном порядке. В целом, криптография с открытым ключом используется здесь четыре раза: дважды для шифрования и дважды для расшифровки. Иногда отправитель может подписать сообщение, применяя закрытый ключ только к небольшому блоку данных, полученных из сообщения, которое нужно отправить, а не ко всему сообщению. В реальном мире такие магазины приложений, как Google Play или Apple App Store, требуют, чтобы приложения были подписаны цифровой подписью, прежде чем они будут опубликованы.

Итак, мы рассмотрели общие принципы использования двух ключей в асимметричной криптографии, которые можно обобщить следующим образом:

- открытые ключи известны и доступны каждому. Их можно использовать для шифрования сообщения или для проверки подписей;
- закрытые ключи являются исключительно секретными и принадлежат отдельным лицам. Они используются для расшифровки сообщения или для создания подписей.

В криптографии с открытым ключом нет проблем с распространением ключей, т. к. обмена общим согласованным ключом более не требуется. Однако при таком подходе существует серьезная проблема. Как можно гарантировать, что открытый ключ, который использован для шифрования сообщения, действительно является открытым ключом предполагаемого получателя, а не злоумышленника или перехватчика? Чтобы решить эту проблему, введено понятие доверенной третьей стороны, называемой *инфраструктурой открытых ключей* (Public Key Infrastructure, PKI). Органы PKI гарантируют подлинность открытых ключей за счет аттестации или нотариального удостоверения личности пользователя. Принцип работы PKI заключается в том, что они предоставляют проверенные открытые ключи, встраивая их в сертификат безопасности, подписанный цифровой подписью заверяющего органа[8].

Схему шифрования с открытым ключом также можно назвать *односторонней функцией* или *функцией секретного люка* (trapdoor function). Это связано с тем, что

---

[8] Примером удостоверения личности в России является регистрация на сайте Госуслуг. Для завершения регистрации пользователь должен явиться с паспортом в заверяющий орган (районная администрация, почтовое отделение, МФЦ) и подтвердить свою личность.

зашифровать исходный текст с использованием открытого ключа ***Puk*** легко, а вот обратная процедура практически невозможна. Никто на самом деле не может восстановить исходный открытый текст из зашифрованного текста, не зная закрытого ключа ***Prk***, который является тщательно охраняемой секретной информацией владельца. Кроме того, хотя ключи математически связаны, но вычислительными методами невозможно найти один ключ из другого.

Мы обсудили важные аспекты криптографии с открытым ключом — такие как ввод в действие ключей, аутентификация и неотрекаемость[9] с помощью цифровых подписей, а также конфиденциальность с помощью шифрования. Однако не все алгоритмы шифрования с открытым ключом могут обеспечить указанные характеристики. Кроме того, алгоритмы отличаются с точки зрения основной вычислительной задачи и классифицируются в соответствии с ней. Некоторые алгоритмы, такие как RSA, основаны на схеме целочисленной факторизации, поскольку трудно вычислять большие числа. Другие алгоритмы, такие как обмен ключами Диффи — Хеллмана (DH) и DSA, основаны на задачах дискретного логарифма в конечных полях. Обобщенной версией задач дискретного логарифма являются схемы открытых ключей на эллиптических кривых (Elliptic Curve, EC). Примером такой схемы может служить алгоритм цифровой подписи на эллиптических кривых (Elliptic Curves Digital Signature Algorithm, ECDSA). Мы рассмотрим большинство из этих алгоритмов в следующем разделе.

## RSA

Алгоритм RSA, названный в честь его разработчиков Рона Ривеста, Ади Шамира и Леонарда Адлемана, возможно, является одним из наиболее широко используемых криптографических алгоритмов. Он основан на практической сложности факторинга очень больших чисел. В RSA открытый текст и зашифрованный текст являются целыми числами от $\mathbf{0}$ до $\mathbf{n} - \mathbf{1}$ для некоторого $\mathbf{n}$.

Мы рассмотрим схему RSA с двух сторон. Во-первых, это генерация пар ключей, а во-вторых, процесс шифрования и дешифрования. Поскольку механизм генерации ключей RSA базируется на модульной арифметике, давайте с ней бегло ознакомимся.

### Модульная арифметика

Пусть $\mathbf{m}$ — положительное целое число, называемое *модулем*. Два целых числа: $\mathbf{a}$ и $\mathbf{b}$ являются конгруэнтными (сравнимыми, равноостаточными) по модулю $\mathbf{m}$, если:

$$\mathbf{a} \equiv \mathbf{b} \pmod{\mathbf{m}},$$

что означает $\mathbf{a} - \mathbf{b} = \mathbf{m} \times \mathbf{k}$ для некоторого целого числа $\mathbf{k}$.

**Пример**: если $\mathbf{a} \equiv 16 \pmod{10}$, то $\mathbf{a}$ может иметь следующие значения:

$$\mathbf{a} = \ldots, -24, -14, -4, 6, 16, 26, 36, 46.$$

---

[9] Невозможность отказа от авторства сообщения или от факта участия в информационном обмене.

Любое из этих чисел после вычитания 16 делится нацело на 10. Например:

$$-24 - 16 = -40, \text{ т. е. } -24-16 = -10 \times 4.$$

Обратите внимание, что в случае $\mathbf{a} \equiv 36 \pmod{10}$ справедливы те же значения **a**.

Согласно теореме о делении с остатком, существует только одно-единственное решение **a**, которое удовлетворяет условию: $0 \leq \mathbf{a} < \mathbf{m}$. В примере $\mathbf{a} \equiv 16 \pmod{10}$ только значение 6 удовлетворяет условию $0 \leq 6 < 10$. Это свойство будет использоваться в процессе шифрования/дешифрования по алгоритму RSA.

Давайте теперь разберемся, что такое обратное по модулю число. Если число **b** является обратным числу **a** по модулю **m**, то это может быть представлено как:

$$\mathbf{a} \times \mathbf{b} \equiv 1 \pmod{\mathbf{m}},$$

откуда следует, что $\mathbf{a} \times \mathbf{b} - 1 = \mathbf{m} \times \mathbf{k}$ для некоторого целого числа **k**.

**Пример**: для числа 3 есть обратное число 7 по модулю 10, т. к.

$3 \times 7 = 1 \pmod{10} \Rightarrow 3 \times 7 - 1 = 10 \times 2$, делится на 10 без остатка ($\mathbf{m} = 10$, $\mathbf{k} = 2$).

### Генерация пар ключей

Как уже обсуждалось, для участия в асимметричном криптографическом обмене обе стороны должны иметь свою пару ключей. В схеме RSA открытый ключ состоит из (**e**, **n**), где **n** называется *модулем*, а **e** называется *открытой экспонентой*. Точно так же закрытый ключ состоит из (**d**, **n**), где **n** — тот же самый модуль, а **d** — *закрытая экспонента*.

Давайте на примере посмотрим, как генерируются эти ключи:

- создайте пару из двух больших простых чисел **p** и **q**. Давайте возьмем два небольших простых числа в качестве примера для простоты понимания. Итак, пусть два простых числа будут: $\mathbf{p} = 7$ и $\mathbf{q} = 17$;
- рассчитайте модуль RSA (**n**) как $\mathbf{n} = \mathbf{p} \times \mathbf{q}$. Это **n** должно быть большим числом, обычно не менее 512 битов. В нашем примере модуль (**n**) $= \mathbf{p} \times \mathbf{q} = 119$;
- найдите открытую экспоненту **e**, такую, что $1 < \mathbf{e} < (\mathbf{p} - 1) \times (\mathbf{q} - 1)$, и не должно быть общего множителя для **e** и $(\mathbf{p} - 1) \times (\mathbf{q} - 1)$, кроме 1. Это означает, что **e** и $(\mathbf{p} - 1) \times (\mathbf{q} - 1)$ взаимно просты. Может существовать несколько значений **e**, которые удовлетворяют этому условию, но выбрать надо какое-то одно;
- в нашем примере $(\mathbf{p} - 1) \times (\mathbf{q} - 1) = 6 \times 16 = 96$. Таким образом, **e** может быть простым и меньше 96. Давайте примем **e** равным 5;
- теперь пара чисел (**e**, **n**) образует открытый ключ и должна быть обнародована. Итак, в нашем примере открытым ключом являются числа (5, 119);
- рассчитайте закрытую экспоненту **d**, используя **p**, **q** и **e**, считая, что число **d** является обратным к **e** по модулю $(\mathbf{p} - 1) \times (\mathbf{q} - 1)$. Это означает, что **d** при умножении на **e** равно 1 по модулю $(\mathbf{p} - 1) \times (\mathbf{q} - 1)$ и $\mathbf{d} < (\mathbf{p} - 1) \times (\mathbf{q} - 1)$. Это может быть представлено как:

$$\mathbf{e} \times \mathbf{d} = 1 \bmod (\mathbf{p} - 1) \times (\mathbf{q} - 1)$$

- обратите внимание, что эта мультипликативная инверсия является связующим звеном между закрытым ключом и открытым ключом. Хотя ключи не являются производными друг от друга, между ними есть математическая связь;
- в нашем примере мы должны найти **d** такой, чтобы выполнялось приведенное равенство. Это означает, что $5 \times \mathbf{d} = 1 \bmod 96$, а также $\mathbf{d} < 96$;
- рассматривая множество значений **d** (их можно рассчитать с использованием расширенной версии алгоритма Евклида), мы видим, что **d** = 77 удовлетворяет нашему условию. Проверим это:

  $77 \times 5 = 385$ и $385 - 1 = 384$ делится на 96, потому что $4 \times 96 + 1 = 385$

- итак, в нашем примере закрытый ключ будет представлен числами (77, 119).

Теперь у вас есть пара ключей!

### Шифрование/расшифровка с использованием пар ключей

Когда сгенерированы ключи, последующее шифрование и дешифрование не составляет особого труда. Математика процесса выглядит следующим образом:

- шифрование открытого сообщения **m** для получения зашифрованного сообщения:

  $\mathbf{c} = \mathbf{m}^{\mathbf{e}} \pmod{\mathbf{n}}$, имея открытый ключ (**e**, **n**) и открытый текст **m**

- расшифровка зашифрованного текстового сообщения **c** для получения открытого текстового сообщения **m**:

  $\mathbf{m} = \mathbf{c}^{\mathbf{d}} \pmod{\mathbf{n}}$, имея закрытый ключ (**d**, **n**) и зашифрованный текст **c**

Заметим, что схема RSA является блочным шифром, в котором входное сообщение делится на небольшие блоки, пригодные для обработки по алгоритму RSA. Кроме того, открытый текст и зашифрованный текст являются целыми числами от 0 до $\mathbf{n} - 1$ для некоторого целого числа **n**, которое известно как отправителю, так и получателю. Это означает, что входной открытый текст представлен как целое число, и когда он проходит через RSA и становится зашифрованным, он снова является целым числом, но уже другим — мы ведь его зашифровали. Теперь, используя те же пары ключей из предыдущего примера, давайте пошагово разберем, как RSA работает на практике:

- отправитель хочет отправить текстовое сообщение получателю, открытый ключ которого известен: пара чисел (**e**, **n**);
- отправитель разбивает текстовое сообщение на блоки, которые могут быть представлены в виде последовательности чисел, меньшей **n**;
- зашифрованный эквивалент открытого текста можно найти с помощью равенства:

  $\mathbf{c} = \mathbf{m}^{\mathbf{e}} \pmod{\mathbf{n}}$

  Если открытый текст (**m**) равен 19, а открытый ключ — (5, 119), где $\mathbf{e} = 5$ и $\mathbf{n} = 119$, то зашифрованный текст **c** будет:

  $19^5 \pmod{119} = 2476099 \pmod{119} = 66$

Здесь 66 является остатком, а 20807 — частным, которое мы не используем[10]. Итак, **c** = 66;

- когда зашифрованный текст **c** = 66 доставлен получателю, его необходимо расшифровать, чтобы получить открытый текст, используя $\mathbf{m} = \mathbf{c}^{\mathbf{d}} \pmod{\mathbf{n}}$;
- получатель уже имеет закрытый ключ (**d**, **n**) с **d** = 77 и **n** = 119 и получил зашифрованный текст **c** = 66. Таким образом, получатель может легко получить открытый текст, используя эти значения[11] как $\mathbf{m} = 66^{77} \pmod{119} = 19$;
- для модульных арифметических вычислений существует множество онлайн-калькуляторов, с которыми вы можете поиграть, например, здесь: **http://comnuan.com/cmnn02/cmnn02008/**.

Итак, мы рассмотрели вычисления, на которых основан алгоритм RSA. Теперь мы знаем, что **n** (должно быть очень большим числом) является общедоступным. Но, хотя оно общедоступно, факторинг этого большого числа для получения простых чисел **p** и **q** чрезвычайно труден. Схема RSA основана на практической трудности факторизации больших чисел. Если **p** и **q** недостаточно велики, или открытый ключ **e** мал, то устойчивость RSA снижается. В настоящее время ключи RSA обычно имеют длину от 1024 до 2048 битов. Разумеется, вычислительные издержки алгоритма RSA увеличиваются с размером ключей.

В ситуациях, когда объем данных очень велик, рекомендуется использовать метод симметричного шифрования данных, но для передачи ключей задействовать методы асимметричного шифрования, такие как RSA.

Мы рассмотрели один из практических аспектов RSA: шифрование и дешифрование. Однако его также можно использовать для аутентификации с помощью цифровой подписи. Отправитель может взять хэш данных, подписать его своим закрытым ключом и отправить зашифрованный хэш вместе с данными. Получатель может проверить хэш с помощью открытого ключа отправителя и убедиться, что данные отправил именно ожидаемый отправитель, а не кто-то другой. Таким образом, в дополнение к безопасной передаче ключа, метод шифрования с открытым ключом RSA также предлагает аутентификацию с использованием цифровой подписи. Отметим, что в таких ситуациях может использоваться другой алгоритм, называемый алгоритмом цифровой подписи (Digital Signature Algorithm, DSA). Об этом мы расскажем в следующем разделе.

RSA широко используется в веб-браузерах, электронной почте, VPN и спутниковом телевидении. Кроме того, многие коммерческие приложения или программы в магазинах приложений также имеют цифровую подпись с использованием RSA. Протокол SSH тоже использует криптографию с открытым ключом — когда вы подключаетесь к SSH-серверу, он передает вам открытый ключ, который можно ис-

---

[10] Иными словами, вычисляем $19^5 = 2476099$ и берем остаток от деления на 119.

[11] Строго говоря, незачем вычислять огромное число $66^{77}$. При каждом умножении достаточно вычислять не полное произведение, а только остаток от деления на 119.

пользовать для шифрования данных, отправляемых на этот сервер. Затем сервер может расшифровать данные, используя свой закрытый ключ.

## Алгоритм цифровой подписи DSA

DSA был разработан NSA как часть стандарта цифровой подписи (Digital Signature Standart, DSS) и стандартизирован NIST. Основное назначение DSA заключается в цифровой подписи сообщений, а не в шифровании. Иначе говоря, RSA предназначен как для управления ключами, так и для аутентификации, а DSA — только для аутентификации. Кроме того, в отличие от RSA, который основан на факторизации большого числа, DSA основан на дискретных логарифмах. Принцип использования DSA показан на рис. 2.20.

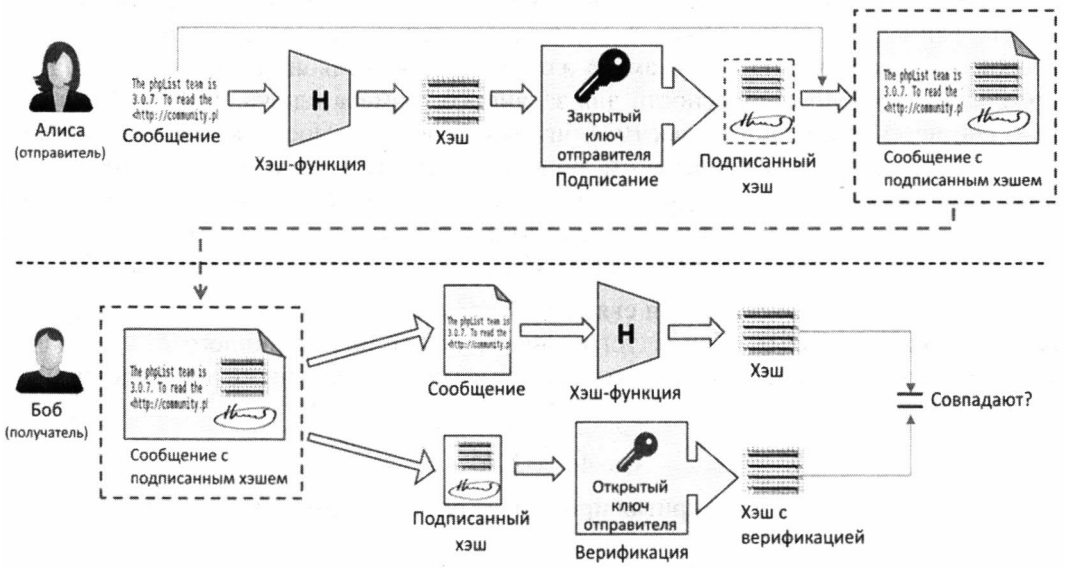



**Рис. 2.20.** Алгоритм цифровой подписи DSA

Как вы можете здесь видеть, сообщение сначала хэшируется, а затем подписывается, потому что так оно будет более защищено по сравнению с подписанием и последующим хэшированием. В идеале вам следует проверить подлинность сообщения перед выполнением любой другой операции. Итак, после того, как хэш подписан, он присоединяется к сообщению и отправляется получателю. Затем получатель может проверить подлинность и расшифровать хэш. Также он хэширует расшифрованное сообщение, чтобы снова получить хэш и проверить, совпадают ли эти два хэша. Таким образом, DSA предоставляет следующие свойства безопасности:

- **подлинность отправителя** — подписано закрытым ключом и проверено открытым ключом;
- **целостность данных** — хэши не будут совпадать, если данные изменены;
- **неотрекаемость** — поскольку отправитель подписал хэш, он не может позже отрицать, что отправил именно это сообщение. Неотрекаемость — это свойство,

наиболее востребованное в ситуациях, когда есть вероятность спора по поводу обмена данными. Например, как только заказ размещен в электронном виде, покупатель не может отклонить заказ на покупку, если в такой ситуации предусмотрен запрет отказа.

Типичная схема DSA состоит из трех алгоритмов: (1) генерация ключа, (2) генерация подписи и (3) проверка подписи.

## Криптография на эллиптических кривых

Криптография на эллиптических кривых (ECC) фактически возникла из криптографии Диффи — Хеллмана. Она была разработана как альтернативный механизм реализации криптографии с открытым ключом. На самом деле она относится к набору криптографических протоколов и основана на проблеме дискретного логарифма, как в DSA. Однако считается, что дискретная логарифмическая задача еще сложнее применительно к точкам на эллиптической кривой. Таким образом, ECC предлагает лучшую безопасность для заданного размера ключа. 160-битный ключ ECC считается защищенным как 1024-битный ключ RSA. Поскольку ключи меньшего размера в ECC могут обеспечить большую безопасность и производительность по сравнению с другими алгоритмами открытого ключа, ECC широко используется в небольших встроенных устройствах, датчиках и других устройствах Интернета вещей. Для ECC доступны чрезвычайно эффективные аппаратные реализации.

ECC основана на математически связанном наборе чисел на эллиптической кривой над конечными полями. Кроме того, она не имеет ничего общего с эллипсами! Эллиптическая кривая удовлетворяет следующему математическому уравнению[12]:

$$y^2 = x^3 + ax + b,$$

$$\text{где } 4a^3 + 27b^2 \neq 0.$$

При разных значениях $a$ и $b$ кривая принимает разные формы (рис. 2.21).

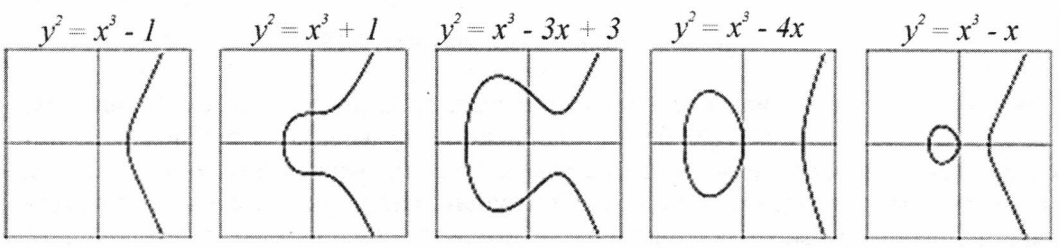



**Рис. 2.21.** Формы эллиптических кривых при разных значениях *a* и *b*

Есть несколько важных характеристик эллиптических кривых, которые используются в криптографии:

- они горизонтально симметричны — т. е. то, что находится ниже оси $X$, является зеркальным отображением того, что находится выше оси $X$. Таким образом,

[12] Это уравнение называется *обычной формулировкой Вейерштрасса* для эллиптических кривых.

любая точка на кривой при отражении по оси *X* по-прежнему остается на кривой;

- любая не вертикальная линия может пересекать кривую не более, чем в трех местах;
- если вы выберете две точки ***P*** и ***Q*** на эллиптической кривой и проведете линию через них, линия пересечет кривую еще в одном месте (рис. 2.22). Давайте назовем эту точку ***-R***. Если вы проведете вертикальную линию через ***-R***, она пересечет кривую, скажем, в точке ***R***, которая является *отражением* точки ***-R***. То есть, третье свойство подразумевает, что ***P*** + ***Q*** = ***R***. Это называется *сложением точек* и означает, что добавление двух точек на эллиптической кривой приведет вас к третьей точке кривой;

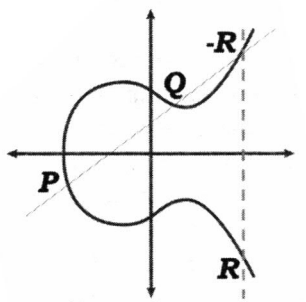



**Рис. 2.22.** Схема сложения точек

- таким образом, вы можете применить сложение точек к любым двум точкам на кривой. В предыдущем пункте мы выполнили сложение точек ***P*** и ***Q*** (т. е. ***P*** + ***Q***) и нашли ***-R***, а затем в конечном итоге пришли к ***R***. Как только мы нашли ***R***, мы можем провести линию от ***P*** к ***R*** и увидеть, что линия снова пересекает график в третьей точке. Затем мы можем взять эту точку и двигаться вдоль вертикальной линии, пока она снова не пересечет график. Новая точка пересечения будет результатом сложения точек ***P*** и ***R***. Этот процесс с фиксированной точкой ***P*** и получающейся точкой может продолжаться столько, сколько мы хотим, и мы будем продолжать получать на кривой все новые точки;
- что, если вместо двух точек ***P*** и ***Q*** мы применим операцию к одной и той же точке ***P***, т. е. выполним ***P*** + ***P*** (так называемое *удвоение точки*)? Очевидно, что через ***P*** можно провести бесконечное число линий, поэтому мы будем рассматривать только касательную линию. Касательная линия пересечет кривую еще в одной точке (***-2P***), а вертикальная линия оттуда снова пересечет кривую и укажет нам конечное значение (***2P***). Эта ситуация представлена на рис. 2.23;
- очевидно, что мы можем применить удвоение начальной точки **n** раз, и это будет каждый раз приводить нас к новой точке на кривой. Когда мы впервые применили удвоение к точке ***P***, это привело нас к результирующей точке ***2P***, как вы только что видели на графике (см. рис. 2.23). Теперь, если то же самое повторяется **n** раз, мы достигнем на кривой точки, показанной на рис. 2.24;

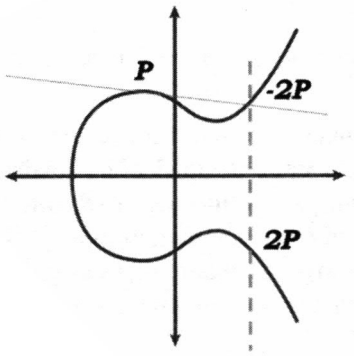



Рис. 2.23. Схема удвоения точек

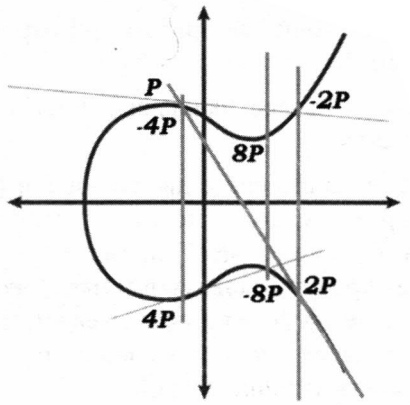



Рис. 2.24. Схема многократного удвоения точек

- в упомянутом ранее сценарии, когда известны только начальная и конечная точки, ни у кого нет простого способа определить, что для достижения конечной результирующей точки удвоение точки применялось **n** раз, — за исключением перебора всех возможных **n** одного за другим. Это проблема дискретного логарифма для ECC, где говорится, что для заданных точек **G** и **Q**, где **Q** кратно **G**, необходимо найти **d** такое, при котором **Q** = **dG**, что формирует одностороннюю функцию без короткого пути к решению. Здесь **Q** — открытый ключ, а **d** — закрытый ключ. Можете ли вы восстановить закрытый ключ **d** из открытого ключа **Q**? В этом и заключается задача дискретного логарифма (*задача дискретного логарифмирования*[13]) эллиптической кривой, которую очень трудно решить с вычислительной точки зрения;
- в дополнение к этому, кривая должна быть определена над *конечным полем*, а не уводить нас в бесконечность! Это означает, что значение **max** на оси *X* должно быть ограничено некоторым значением, поэтому мы просто обращаем значения при достижении максимума. Это граничное значение в криптосистеме ECC тоже обозначается буквой **P** (но это не та буква ***P***, которая на графиках) и называется *значением по модулю*, и оно также определяет размер ключа, т. е. задает конечное поле. Во многих реализациях ECC в качестве **P** выбирают простое число;

**ПРИМЕЧАНИЕ ПЕРЕВОДЧИКА.** В других публикациях по криптографии значение модуля обозначают маленькой буквой **p** — во избежание путаницы с обозначением точек на кривой. Трудно сказать, почему авторы книги решили в данном случае использовать в тексте заглавную букву **P**, совпадающую с обозначением опорной точки на графике кривой.

- чем больше **P**, тем больше пригодных к использованию значений на кривой и лучше безопасность;

[13] Слово *логарифмирование* употребляется в этой области знаний вместо слова *деление* исключительно для единства терминов с другими криптографическими системами, где вместо умножения применяется возведение в степень.

- мы видели, что сложение точек и удвоение точек составляют основу для поиска значений, которые используются для шифрования и дешифрования.

Таким образом, для определения ECC необходимо установить следующие *параметры области определения*:

- уравнение кривой: $y^2 = x^3 + ax + b$, где $4a^3 + 27b^2 \neq 0$;
- **P** — простое число (модуль), которое определяет конечное поле, над которым будет определяться кривая;
- **a** и **b** — коэффициенты, которые определяют эллиптическую кривую;
- **G** — *базовая точка* или *точка генерации* на кривой. Это точка, где начинаются все точечные операции, и она определяет циклическую подгруппу;
- **n** — количество точечных операций на кривой, пока результирующая линия не станет вертикальной. Таким образом, это *порядок* группы **G**, т. е. наименьшее положительное число, такое, что $\mathbf{nG} = \infty$. Обычно это простое число;
- **h** — это число называется *кофактором* и равно общему количеству точек на кривой, деленному на **n**. Это целочисленное значение, обычно близкое к 1.

Заметим, что ECC — это отличный метод для генерации ключей, но для цифровых подписей и обмена ключами он используется вместе с другими методами. Например, алгоритм Elliptic Curve Diffie-Hellman (ECDH) весьма широко применяется для обмена ключами, а ECDSA — для цифровых подписей.

## Алгоритм ECDSA

ECDSA (Elliptic Curve Digital Signature Algorithm, алгоритм цифровой подписи на эллиптических кривых) — это тип DSA, который использует ECC для генерации ключей. Как следует из названия, его назначением является цифровая подпись, а не шифрование. ECDSA может быть лучшей заменой RSA с точки зрения меньшего размера ключа, лучшей безопасности и более высокой производительности. Это один из самых важных криптографических компонентов, используемых в протоколе Bitcoin.

Мы уже рассмотрели, как для установления доверия между отправителем и получателем используются цифровые подписи. Поскольку подлинность отправителя и целостность сообщения могут быть проверены с помощью цифровых подписей, две неизвестные стороны могут заключать сделки друг с другом. Обратите внимание, что отправитель и получатель должны согласовать ключевые параметры, прежде чем устанавливать связь.

Алгоритм ECDSA состоит из трех этапов: генерация ключей, генерация подписи и проверка подписи.

### Генерация ключей

Поскольку параметры области определения (**P**, **a**, **b**, **G**, **n**, **h**) заданы заранее, кривая и базовая точка известны обеим сторонам. Также им известно простое число **P**, которое определяет конечное поле (обычно **P** равно 160 битам и может быть даже

больше). Итак, отправитель, скажем, Алиса делает для генерации ключей следующее:

- выбирает случайное целое число **d** в интервале [1, **n** – 1];
- вычисляет **Q** = **dG**;
- объявляет **Q** своим открытым ключом и оставляет **d** закрытым ключом.

### Генерация подписи

После того как ключи сгенерированы, отправитель Алиса воспользуется закрытым ключом **d**, чтобы подписать сообщение **m**. То есть для генерации подписи она выполняет следующие шаги:

- выбирает случайное число **k** в интервале [1, **n** – 1];
- вычисляет **kG** и находит новые координаты ($x_1$, $y_1$) и $r = x_1 \bmod n$.

  Если **r** = 0, тогда начинает все сначала;
- вычисляет **e** = SHA–1 (**m**);
- вычисляет $s = k^{-1} (e + dr) \bmod n$.

  Если **s** = 0, то начинает все сначала с первого шага;
- подпись Алисы для сообщения **m** теперь будет (**r**, **s**).

### Проверка подписи

Допустим, Боб является получателем и имеет доступ к параметрам области определения и открытому ключу **Q** отправителя Алисы. В качестве меры безопасности Боб должен сначала проверить, что все данные, которыми он обладает, а именно: параметры области определения, подпись и открытый ключ Алисы **Q** — являются действительными. Чтобы проверить подпись Алисы в сообщении **m**, Боб выполняет следующие операции в указанном порядке:

- убеждается, что **r** и **s** являются целыми числами в интервале [1, **n** – 1];
- вычисляет **e** = SHA–1(**m**);
- вычисляет $w = s^{-1} \bmod n$;
- вычисляет $u_1 = ew \bmod n$ и $u_2 = rw \bmod n$;
- вычисляет $X = u_1G + u_2G$, где **X** представляет координаты — скажем ($x_2$, $y_2$);
- вычисляет $v = x_1 \bmod n$;
- если **r** = **v** принимает подпись, а иначе отклоняет ее.

* * *

В этом разделе мы рассмотрели математическую базу алгоритма ECDSA. Вспомните, что при генерации ключа и подписи мы использовали случайное число. Чрезвычайно важно убедиться, что генерируемые случайные числа действительно криптографически случайны. Во многих случаях используется 160-битный ECDSA, поскольку он должен соответствовать хэш-функции SHA-1.

Кроме многих других применений, ECDSA используется в *цифровых сертификатах*. В простейшем виде цифровой сертификат — это открытый ключ, связанный с идентификатором устройства и датой истечения срока действия сертификата. Таким образом, сертификаты позволяют нам проверять и подтверждать, кому принадлежит открытый ключ, и убеждаться, что устройство является законным членом рассматриваемой сети. Эти сертификаты очень важны для предотвращения «атаки подменой участника» в протоколах ввода ключей в действие. Многие сертификаты TLS основаны на паре ключей ECDSA, и использование алгоритма продолжает расширяться.

## Примеры кода для криптографии с открытым ключом

В листинге 2.2 приведены примеры кода различных алгоритмов с открытым ключом[14]. Этот раздел предназначен для того, чтобы дать вам общее представление о том, как программно реализовать различные алгоритмы. Примеры кода написаны на Python, но будут выглядеть очень похоже и на других языках. Вам просто нужно найти подходящие библиотечные функции.

**Листинг 2.2. Примеры кода для криптографии с открытым ключом**

```
# -*- coding: utf-8 -*-
import Crypto
from Crypto.PublicKey import RSA
from Crypto import Random
from hashlib import sha256

# Функция генерации ключей заданной длины (1024)
def generate_key(KEY_LENGTH=1024):
    random_value= Random.new().read
    keyPair=RSA.generate(KEY_LENGTH,random_value)
    return keyPair

# Генерация ключей для Алисы и Боба
bobKey=generate_key()
aliceKey=generate_key()

# Вывод на печать открытых ключей для Алисы и Боба
# Эти ключи должны быть обнародованы
alicePK=aliceKey.publickey()
bobPK=bobKey.publickey()

print "Alice's Public Key:", alicePK
print "Bob's Public Key:", bobPK
```

[14] Обратите внимание — это не законченные программы, а фрагменты полезного кода (сниппеты), которые вы можете использовать в своих программах.

```
# Алиса хочет отправить сообщение Бобу
# Давайте создадим тестовое сообщение для Алисы
secret_message="Alice's secret message to Bob"
print "Message", secret_message

# Функция генерации подписи
def generate_signature(key,message):
    message_hash=sha256(message).digest()
    signature=key.sign(message_hash,'')
    return signature

# Сгенерируем подпись для секретного сообщения
alice_sign=generate_signature(aliceKey,secret_message)

# Перед отправкой сообщения в сеть зашифруем его
# с использованием открытого ключа Боба
encrypted_for_bob = bobPK.encrypt(secret_message, 32)

# Боб расшифровывает секретное сообщение
# при помощи своего закрытого ключа
decrypted_message   = bobKey.decrypt(encrypted_for_bob)
print "Decrypted message:", decrypted_message

# Боб использует эту функцию для проверки подписи Алисы
# при помощи ее открытого ключа
def verify_signature(message,PublicKey,signature):
    message_hash=sha256(message).digest()
    verify = PublicKey.verify(message_hash,signature)
    return verify

# Боб проводит верификацию, используя расшифрованное
# сообщение и открытый ключ Алисы
print "Is alice's signature for decrypted message valid?",
verify_signature(decrypted_message,alicePK, alice_sign)

# ********************************************
# *            The ECDSA Algorithm           *
# ********************************************
import ecdsa

# SECP256k1 это эллиптическая кривая биткойна
signingKey = ecdsa.SigningKey.generate(curve=ecdsa.SECP256k1)
# Получаем ключ верификации
verifyingKey = signingKey.get_verifying_key()

# Генерируем подпись сообщения
signature = signingKey.sign(b"signed message")
```

```
# Проверяем достоверность подписи
verifyingKey.verify(signature, b"signed message")
# True => Подпись достоверна

# Проверяем достоверность подписи
assert verifyingKey.verify(signature, b"message")
# Ошибка. Подпись недостоверна
```

## 2.2.5. Обмен ключами по Диффи — Хеллману

В предыдущих разделах мы рассматривали криптографию с симметричным ключом. Вспомните, что обмен секретным ключом между отправителем и получателем является очень сложной задачей. Как правило, канал связи всегда небезопасен. Некто Ева всегда может попытаться перехватить сообщение Алисы во время его передачи, используя различные виды атак.

Технология Диффи — Хеллмана (DH) была разработана как раз для безопасного обмена криптографическими ключами. Очевидно, вам интересно, как может быть безопасен обмен ключами, если сам канал связи небезопасен? Что ж, позже в этом разделе вы увидите, что технология DH на самом деле не пересылает секретный ключ между двумя сторонами, — скорее, речь идет о совместном создании ключа. В конце концов, нам важен лишь результат — чтобы отправитель и получатель имели один и тот же ключ. Однако имейте в виду, что это не криптография с асимметричным ключом, т. к. *во время обмена ключами не происходит шифрование/дешифрование*, на чем фактически основана криптография с асимметричным ключом. Причина, по которой мы начали рассматривать эту технологию только сейчас, состоит в том, что здесь пригодится вся та математика, которую мы изучили в предыдущем разделе.

Прежде чем углубляться в математическое объяснение, сначала попробуем понять концепцию в самом общем виде. Посмотрите на рис. 2.25, где при помощи разных цветов представлено простое объяснение алгоритма DH.

Обратите внимание, что на первом этапе между двумя сторонами был указан только желтый цвет, где может быть любой другой цвет или случайное число. Затем обе стороны добавляют к общему цвету свой секрет и создают смесь. Эта смесь снова передается через тот же незащищенный канал. Затем соответствующие стороны снова добавляют к смеси свой секрет и формируют свой окончательный *общий секрет*. Теперь у Алисы и Боба есть общий секретный ключ, который сформирован путем обмена через открытый канал. В этом примере с цветами следует отметить, что общие секреты — это сочетание одинаковых наборов цветов. Давайте теперь рассмотрим реальные математические операции, которые применяются для генерации ключей:

- Алиса и Боб договариваются о **P** = 23 и **G** = 9;
- Алиса выбирает закрытый ключ **a** = 4, вычисляет $9^4 \bmod 23 = 6$ и отправляет результат Бобу;

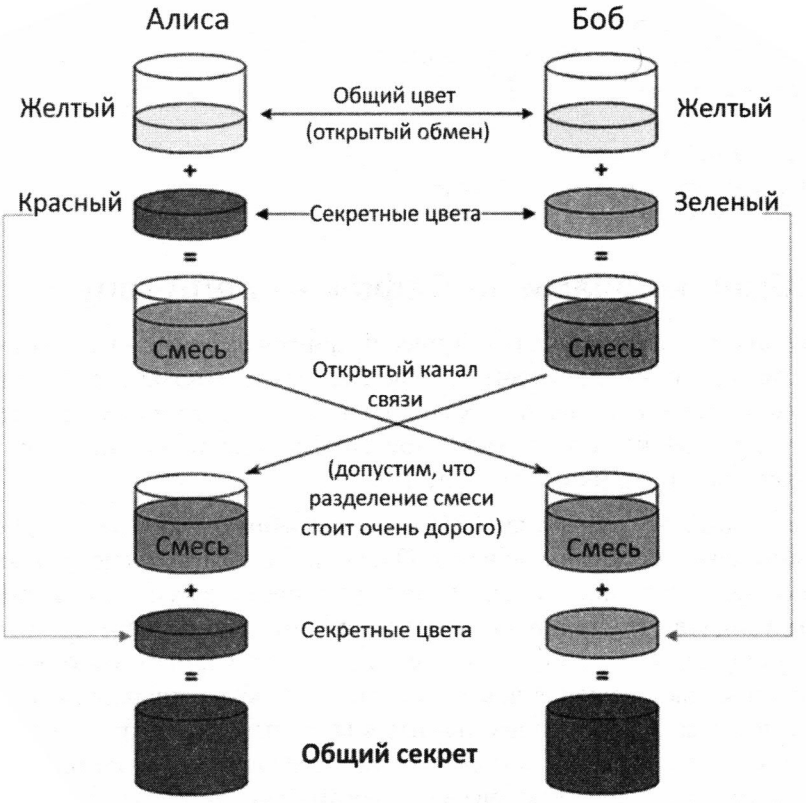



**Рис. 2.25. Обмен ключами по Диффи — Хеллману**

- Боб выбирает закрытый ключ **b** = 3, вычисляет $9^3$ mod 23 = 16 и отправляет результат Алисе;
- Алиса вычисляет $16^4$ mod 23 = 9;
- Боб вычисляет $6^3$ mod 23 = 9.

Если вы выполните эти шаги, то увидите, что и Алиса, и Боб могут создавать на своей стороне один и тот же секретный ключ, который можно использовать для шифрования/дешифрования. В этом примере для простоты понимания мы использовали небольшие числа, но на практике применяются большие простые числа. Чтобы лучше это понять, давайте обратимся к следующему фрагменту кода (на языке C. — *Ред.*) и посмотрим, как алгоритм DH может быть реализован простым способом (листинг 2.3).

**Листинг 2.3. Реализация алгоритма Диффи — Хеллмана**

```
/* Программа для вычисления ключей для двух сторон
с использованием алгоритма обмена Диффи - Хеллмана */

// Функция возвращает значение a ^ b mod P
long long int power(long long int a, long long int b, long long int P)
```

```
{
    if (b == 1)
        return a;

    else
        return (((long long int)pow(a, b)) % P);
}

// Программа вычисления ключа по Диффи – Хеллману
int main()
{
    long long int P, G, x, a, y, b, ka, kb;

    // Обе стороны согласны использовать открытые ключи G и P
    P = 23; // Берем простое число P
    printf("The value of P : %lld\n", P);

    G = 9; // Первообразный корень для P это G
    printf("The value of G : %lld\n\n", G);

    // Алиса выбирает свой закрытый ключ
    a = 4; // a - это закрытый ключ
    printf("The private key a for Alice : %lld\n", a);
    x = power(G, a, P); // получаем обменный ключ

    // Боб выбирает свой закрытый ключ
    b = 3; // b - это закрытый ключ
    printf("The private key b for Bob : %lld\n\n", b);
    y = power(G, b, P); // получаем обменный ключ

    // Генерируем закрытые ключи после обмена
    // ранее сгенерированными обменными ключами
    ka = power(y, a, P); // Секретный ключ для Алисы
    kb = power(x, b, P); // Секретный ключ для Боба

    printf("Secret key for the Alice is : %lld\n", ka);
    printf("Secret Key for the Bob is : %lld\n", kb);

    return 0;
}
```

**ПРИМЕЧАНИЕ.** Хотя традиционно используется проблема дискретного логарифма ($x^y \bmod p$), основной процесс может быть модифицирован для использования криптографии на эллиптических кривых.

## 2.2.6. Открытый или закрытый ключ?

Мы рассмотрели различные типы алгоритмов — как с симметричным, так и асимметричным ключом. Очевидно, что назначение и особенности применения этих алгоритмов различаются. Давайте проведем небольшой сравнительный анализ, чтобы использовать правильный алгоритм в нужном месте.

- Криптография с симметричным ключом также называется криптографией с закрытым ключом. Криптография с асимметричным ключом также называется криптографией с открытым ключом.
- Обмен ключами или их распространение в криптографии с симметричным ключом — большая головная боль, в отличие от криптографии с асимметричным ключом.
- Асимметричное шифрование требует значительных вычислительных ресурсов, поскольку длина ключей обычно велика. Следовательно, процесс шифрования и дешифрования происходит медленнее. Наоборот, симметричное шифрование работает быстрее.
- Криптография с симметричным ключом подходит для длинных сообщений, потому что скорость шифрования/дешифрования высока. Криптография с асимметричным ключом подходит для коротких сообщений, а скорость шифрования/дешифрования ниже.
- В криптографии с симметричным ключом при шифровании происходит перестановка или замена символов исходного сообщения. В криптографии с асимметричным ключом открытый текст и зашифрованный текст рассматриваются как целые числа.
- Во многих ситуациях, когда симметричный ключ используется для шифрования и дешифрования сообщений, метод асимметричного ключа применяется для совместного использования и согласования ключа, используемого при шифровании.
- Криптография с асимметричным ключом находит наиболее активное применение в ненадежных средах, когда участвующие стороны не имеют каких-либо предварительных отношений. Поскольку неизвестные стороны не обладают возможностью устанавливать общие секретные ключи, обмен секретными данными обеспечивается с помощью криптографии с открытым ключом.
- Симметричные криптографические методы не обеспечивают способ цифровой подписи, который возможен только с помощью асимметричной криптографии.
- Другим любопытным моментом является количество ключей, необходимых для группы узлов, когда каждому узлу нужна связь с остальными узлами. Как вы думаете, сколько ключей потребуется, скажем, для 100 участников, при шифровании с симметричным ключом? Эта проблема поиска количества необходимых ключей может рассматриваться как проблема полного графа с порядком 100. Подобно тому, как каждой вершине графа требуется 99 связанных ребер, чтобы

соединиться со всеми остальными точками, каждому участнику потребуется 99 ключей для установления защищенных соединений со всеми другими узлами.

Таким образом, в целом, нам понадобится $100 \times (100 - 1) / 2 = 4950$ ключей. В общем виде для $n$ участников количество ключей можно вычислить так:

$$n \times (n - 1) / 2$$

- С увеличением количества участников это становится кошмаром! Однако в случае криптографии с асимметричным ключом каждому участнику просто потребуются два ключа (один закрытый и один открытый). Для сети из 100 участников общее количество необходимых ключей составило бы только 200. В табл. 2.5 приведены некоторые примеры, чтобы вы могли представить, как растет количество ключей при увеличении количества участников.

***Таблица 2.5.*** *Сравнение основных требований для алгоритмов симметричной и асимметричной криптографии*

| Количество участников | Количество симметричных ключей | Количество асимметричных ключей |
|---|---|---|
| 2 | 1 | 4 |
| 4 | 6 | 8 |
| 10 | 45 | 20 |
| 50 | 1225 | 100 |
| 100 | 4950 | 200 |
| 1000 | 499 500 | 2000 |

## 2.3. Теория игр

Теория игр, безусловно, является достаточно давно известной концепцией, и на ее основе решаются сложные задачи во многих реальных ситуациях. Причина, по которой мы рассматриваем эту тему, заключается в том, что теория игр используется в протоколе Bitcoin и многих других реализациях блокчейна. Она была официально представлена Джоном фон Нейманом при исследовании социально-экономических проблем. Позже теория игр была популяризована Джоном Форбсом Нэшем-младшим благодаря его теории «равновесия Нэша», которую мы рассмотрим в ближайшее время. Но давайте сначала определим, что такое теория игр.

*Теория игр* — это теория таких игр, в которые играют не только дети. Чаще всего это ситуации, когда две или более стороны вовлечены в стратегическое поведение. Примеры: турнир по крикету — игра, две конфликтующие стороны в суде с адвокатами и присяжными — игра, брат и сестра, борющиеся за мороженое, — игра, политические выборы — игра, сигнал светофора — тоже игра. Другой пример: скажем, вы подали заявку на должность разработчика блокчейн-приложений, и ва-

шу кандидатуру выбрали и предложили работу с определенной зарплатой, но вы отклонили предложение, полагая, что спрос и предложение огромны, и есть вероятность, что наниматель пересмотрит предложение и увеличит зарплату. Вы, должно быть, думаете сейчас, что это не игра? Так вот, в реальных ситуациях почти все вокруг нас — игра. Таким образом, *игра* может быть определена как ситуация, подразумевающая *коррелированный рациональный выбор*. Это означает, что перспективы, доступные для любого игрока, зависят не только от его собственного выбора, но также и от выбора, который делают другие игроки в данной ситуации. Другими словами, если на вашу судьбу влияют действия других людей, значит, вы в игре. Так что же такое теория игр как наука?

Теория игр — это изучение стратегий, используемых в сложных играх. Это искусство сделать лучший ход или выбрать лучшую стратегию достижения цели в той или иной ситуации. Для этого нужно понять стратегию противника, а также то, что, по мнению противника, будет вашим ходом. Вот вам простой пример: есть два родных брата: старший и младший. Допустим, в холодильнике есть два типа мороженого: одно — с апельсиновым вкусом, а другое — с ароматом манго. Старший хочет съесть апельсиновое мороженое, но знает, что если он выберет его, то младший будет плакать и требовать именно апельсиновое мороженое. Поэтому он выбирает мороженое со вкусом манго, и, как ожидалось, младший хочет того же самого. Теперь старший делает вид, что принес в жертву мороженое со вкусом манго, отдает его младшему и сам ест апельсиновое мороженое (как и хотел). Посмотрите на ситуацию: это беспроигрышный вариант для обеих сторон, т. к. именно апельсиновое мороженое было целью старшего брата. Если бы старший брат захотел выбрать простую стратегию, он мог бы просто подраться с малышом и получить желаемое. Во втором случае старший должен определить, куда и насколько сильно ударить младшего, чтобы тот не пострадал, но отказался от мороженого со вкусом апельсина. Это и есть задача теории игр: какова ваша цель и каким должен быть ваш лучший ход?

Еще один пример, на этот раз, скорее, из области бизнеса. Представьте, что вы продавец, поставляющий овощи в город. Допустим, есть три способа добраться до города, из которых один является обычным маршрутом в том смысле, что все идут этим путем, может быть, потому, что он короче и лучше. Однажды вы видите, что привычная дорога перекрыта из-за ремонтных работ, и вы никоим образом не можете проехать по этому пути. Теперь у вас осталось два других пути. Один из них — короткий, но узковатый. Другой — более длинный, но достаточно широкий. Здесь вы должны разработать стратегию выбора пути, по которому вам нужно ехать. Ситуация может оказаться такой, что на дорогах будет интенсивное движение, и многие люди попытаются проехать по кратчайшему маршруту. Это может привести к сильному затору на узком участке и вызвать огромную задержку. Итак, вы решили выбрать более длинный путь, чтобы гарантированно добраться до города вовремя, но за счет нескольких дополнительных долларов, потраченных на топливо. Вы уверены, что можете легко компенсировать затраты, если прибудете вовремя и продадите свои овощи раньше других по хорошей цене. Это тоже теория

игр: как лучше всего достичь поставленной цели, которая обычно заключается в поиске оптимального решения.

Во многих ситуациях роль, которую вы играете, и ваша цель чрезвычайно важны для формирования стратегии. Например, если вы являетесь организатором спортивного мероприятия, а не участником соревнования, вы должны сформулировать стратегию, в которой ваша цель может состоять в том, чтобы участники играли по правилам и следовали протоколу. Вам безразлично, кто победит в конце, — вы просто организатор. С другой стороны, участник соревнования будет формировать стратегию выигрышных ходов, принимая во внимание сильные и слабые стороны оппонента, а также правила, установленные организатором, потому что за нарушение правил может быть наложен штраф. Теперь давайте глубже рассмотрим ситуацию, когда вы играете роль организатора. Вам следует хорошо подумать о том, может ли сложиться такая ситуация, когда участник нарушает правила и теряет лишь одно очко, но наносит такой ущерб противнику, что тот больше не может соревноваться. Таким образом, вы должны предположить, что могут иметь в виду участники, и соответственно установить свои правила.

Попробуем еще раз определить теорию игр, основываясь на том, что мы узнали из предыдущих примеров. Это метод моделирования реальных жизненных ситуаций в форме игры и анализа того, какой должна быть наилучшая стратегия или действия человека или организации в той или иной ситуации для достижения желаемого результата. Концепции из теории игр широко используются практически во всех аспектах жизни, таких как политика, социальные сети, городское планирование, торги, ставки, маркетинг, распределенное хранение, распределенные вычисления, цепочки поставок и финансы, — и это лишь краткий перечень. Используя теоретические концепции игры, можно проектировать системы, в которых участники играют по правилам, даже если не принимают их эмоциональные или моральные аспекты. Если вы хотите выйти за рамки простого изучения идей и разработать продукт или решение для производства, вам следует воспринимать теорию игр, как один из ключевых элементов системы. Она поможет вам принять надежные решения и протестировать их с различными интересными сценариями. Впрочем, многие люди уже мыслят понятиями теории игр, не зная, что это теория игр. Однако, если вы осознанно вооружитесь множеством инструментов и техник из теории игр, это определенно пойдет вам на пользу.

## 2.3.1. Равновесие по Нэшу

В предыдущем разделе мы рассмотрели различные примеры игр. Существует много способов классификации игр, таких как кооперативные/некооперативные игры, симметричные/асимметричные игры, игры с нулевой суммой/ненулевой суммой, одновременные/последовательные игры и т. д. Давайте сосредоточимся на кооперативных и некооперативных играх, потому что это связано с *равновесием по Нэшу*.

Как следует из названия, игроки сотрудничают (кооперируют) друг с другом и могут работать вместе, чтобы сформировать альянс в кооперативных играх. Кроме

того, для обеспечения кооперативного поведения игроков может быть применена некоторая внешняя сила. С другой стороны, в некооперативных играх игроки соревнуются как личности, не имеющие возможности сформировать альянс. Участники просто заботятся о своих интересах. Кроме того, нет никакой внешней силы, способной навязать игрокам совместное поведение.

Равновесие по Нэшу гласит, что в любых некооперативных играх, в которых игроки знают стратегии друг друга, существует, по крайней мере, одно равновесие, при котором все игроки используют свои лучшие стратегии для получения максимальной выгоды, и ни одна из сторон не выиграет от изменения своей стратегии. Если вы знаете стратегии других игроков, и у вас есть своя собственная стратегия, но при этом вы не можете увеличить выгоду, изменив собственную стратегию, то это состояние равновесия Нэша. Таким образом, каждая стратегия в равновесии Нэша является лучшим ответом на все другие стратегии в этом равновесии.

Обратите внимание, что в теории игр игрок может вырабатывать стратегию, чтобы извлечь выгоду для себя, но не может победить противника, стараясь причинить ему максимальный ущерб. Кроме того, любая игра, в которую играют повторно, может в конечном итоге попасть в равновесие по Нэшу[15].

В следующем разделе мы рассмотрим *дилемму заключенного*, чтобы получить практическое представление о равновесии по Нэшу.

## 2.3.2. Дилемма заключенного

Многие игры в реальной жизни также могут быть играми с ненулевой суммой. Дилемма заключенного является одним из таких примеров, который можно в целом классифицировать как симметричную игру. Если вы меняете местами личности игроков (например, если играют два игрока «А» и «В», то «А» становится «В», а «В» становится «А»), однако не меняете стратегии, то выгоды остаются прежними. Значит, это симметричная игра.

Давайте начнем непосредственно с примера. Предположим, что есть два парня — Боб и Чарли — которых полицейские поймали за независимую друг от друга продажу наркотиков в разных местах. Они содержатся для допроса в двух разных камерах. Затем им сказали, что за это преступление они будут приговорены к тюремному заключению на два года. Но теперь полицейские почему-то подозревают, что эти два парня также могли быть причастны к ограблению, которое произошло на прошлой неделе. Но если они и не совершили ограбление, то все равно получат по два года лишения свободы. При этом полицейские должны выработать стратегию, чтобы добраться до правды. Что же они делают?

Полицейские идут к Бобу и дают ему выбор, хороший выбор, который выглядит следующим образом: если Боб признается в своем преступлении, а Чарли нет, то

---

[15] Многие игры завершаются до того, как найдено равновесие по Нэшу. Повторение игры ускоряет выработку оптимальной стратегии всеми игроками.

его наказание снизится с двух до одного года, а Чарли получит пять лет. Однако, если Боб отрицает вину, а Чарли признается, тогда Боб получает пять лет, а Чарли — только один год. Кроме того, если оба признаются, то оба получают по три года лишения свободы. Точно такой же выбор предоставляется и Чарли. Как вы думаете, что они будут делать? Эта ситуация и называется *дилеммой заключенного*.

Боб и Чарли находятся в двух разных камерах. Они не могут поговорить друг с другом и договориться о стратегии, когда они оба отрицают участие в ограблении и получают по два года тюрьмы (только за торговлю наркотиками), что представляется глобальным оптимумом в этой ситуации. Но даже если бы они могли разговаривать друг с другом, они могут не доверять друг другу.

Что сейчас пришло в голову Бобу? У него есть два варианта: признаться или отрицать участие в ограблении. Он знает, что Чарли выберет то, что лучше для него, и он сам ничем не отличается от Чарли. Если он отрицает вину, а Чарли признается, то у Боба проблема с пятью годами тюрьмы, а Чарли получит всего один год. Разумеется, Боб не хочет попадать в эту ситуацию.

Если Боб признается, то у Чарли есть два варианта: признаться или отрицать. Теперь Боб думает, что если он признается, то, что бы ни делал Чарли, ему не дадут больше трех лет. Давайте рассмотрим сценарии для Боба:

- Боб признается, а Чарли отрицает: Боб получает один год, Чарли — пять лет (это лучший случай для Боба, учитывая, что он признается);
- Боб признается, и Чарли тоже признается: и Боб, и Чарли получают по три года (в худшем случае, учитывая, что Боб признается).

Эта ситуация называется *равновесием по Нэшу*, когда каждая сторона сделала лучший ход, учитывая выбор другой стороны. Это явно не глобальный оптимум, но оптимальный выбор для отдельных личностей. Если вы посмотрите на эту ситуацию со стороны, то скажете, что оба должны отрицать ограбление и получить по два года. Но когда вы являетесь участником игры, равновесие по Нэшу — это то, во что вы, в конечном итоге, попадете. Обратите внимание, что это наиболее стабильное состояние, когда изменение решения не приносит вам никакой пользы. Его можно наглядно представить в виде матрицы выигрышей (рис. 2.26).


**Рис. 2.26. Матрица выигрышей для дилеммы заключенного**

## 2.3.3. Проблема византийских генералов

В предыдущих разделах мы рассмотрели различные примеры игр и изучили несколько концепций теории игр. Теперь мы обсудим конкретную проблему, которая с давних времен и до сих пор широко используется для решения многих задач информатики.

*Проблема византийских генералов* возникла в ситуации, с которой столкнулась византийская армия во время нападения на город. В целом проблема простая, но с ней очень трудно справиться. Короче говоря, ситуация состояла в том, что несколько армейских группировок, которыми командовали отдельные генералы, окружили город, чтобы завоевать его. Единственный шанс на победу — это когда все генералы атакуют город вместе. Однако проблема заключается в том, как достичь консенсуса. Это подразумевает, что либо все генералы должны атаковать, либо все они должны отступить. Если некоторые из них атакуют, а некоторые отступают, то, скорее всего, они проиграют битву. Давайте рассмотрим пример с числами, чтобы лучше понять ситуацию.

Предположим, что к городу подошли пять группировок византийской армии. Они нападут на город, если хотя бы три из пяти генералов захотят атаковать, а иначе должны отступить. Если среди генералов есть предатель, то он может проголосовать за атаку с генералами, желающими атаковать, и проголосовать за отступление с генералами, желающими отойти. Он способен так поступить, потому что армия рассредоточена по группировкам, что затрудняет их централизованную координацию. Это может привести к тому, что только два генерала нападают на город, и их численно превосходят и побеждают. В такой ситуации могут возникнуть и более сложные проблемы:

- что если есть более одного предателя?
- как будет осуществляться координация сообщений между генералами?
- что если посланник пойман/убит/подкуплен командиром городского гарнизона?
- что если генерал-предатель подделывает сообщение и обманывает других генералов?
- как определить, какие из генералов честны, а какие — предатели?

Как видите, для скоординированного нападения на город необходимо решить множество задач. Проблему можно наглядно представить, как показано на рис. 2.27.

В реальной жизни существует много сценариев, которые аналогичны проблеме византийских генералов. То, как группа людей достигает консенсуса по вопросам голосования, как поддерживают согласованное состояние распределенной или децентрализованной базы данных или как поддерживают согласованное состояние копий блокчейнов на узлах в сети — вот несколько примеров, похожих на проблему византийских генералов. Обратите внимание, однако, что решения этих проблем в разных ситуациях могут быть совершенно разными. Далее в этой книге мы рассмотрим, как Bitcoin решает проблему византийских генералов.

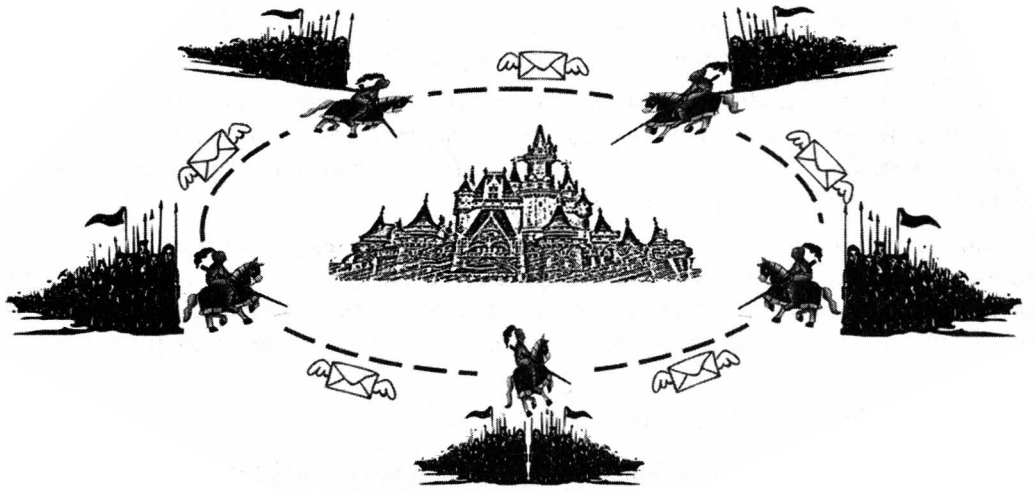


**Рис. 2.27. Византийская армия окружает город**

## 2.3.4. Игры с нулевой суммой

Суть игры с нулевой суммой в теории игр довольно проста. В таких играх выигрыш одного игрока эквивалентен проигрышу другого игрока. Один игрок выигрывает ровно столько же, сколько проигрывает противник, что означает, что выбор игроков не может ни увеличить, ни уменьшить доступные в такой ситуации ресурсы.

Покер, шахматы, Го — вот несколько примеров игр с нулевой суммой. Если обобщить еще шире, то игры, в которых выигрывает только один человек, а противник проигрывает, такие как теннис, бадминтон и т. п., также являются играми с нулевой суммой. Многие финансовые инструменты, такие как свопы, форварды и опционы, также могут быть описаны как игры с нулевой суммой.

Во многих ситуациях в реальной жизни трудно количественно оценить выгоды и потери. Поэтому игры с нулевой суммой встречаются реже, чем игры с ненулевой суммой. Большинство финансовых транзакций или сделок, а также фондовый рынок — игры с ненулевой суммой. Страхование, однако, является областью, где игра с нулевой суммой выполняет важную роль. Просто задумайтесь, как работает страховая схема. Мы выплачиваем страховые взносы страховым компаниям, чтобы защитить себя от некоторых сложных ситуаций, таких как несчастные случаи, госпитализация, смерть и т. п. Помня, что мы застрахованы, мы живем спокойной мирной жизнью и получаем справедливую выплату от страховой компании, когда попадаем в трудную ситуацию. Это финансовая поддержка, которая помогает нам выжить. Далеко не все из тех, кто платит взносы, встречаются с несчастным случаем и попадают в больницу. Зато те, кто действительно получили компенсацию, сравнивают ее со взносами, которые они уплатили страховой компании, и видят, что размер компенсации намного больше суммы взноса. В то же время в страховой компании приходы и расходы весьма точно сбалансированы, даже с учетом операционных расходов компании. Впрочем, страховая компания может инвестировать

взносы, которые мы платим, и получать от этого некоторую прибыль. Тем не менее, в целом страховой бизнес — это игра с нулевой суммой.

Вот вам еще один пример: если есть одна открытая позиция по найму на работу, для которой проводится собеседование, тогда кандидат, который соответствует требованиям, фактически получает работу за счет дисквалификации других кандидатов. Это также игра с нулевой суммой.

Вы можете спросить, есть ли смысл изучать игры с нулевой суммой? Даже поверхностное понимание того, что такое игра с нулевой суммой, помогает разработать оптимальную стратегию в любой сложной ситуации. Мы можем проанализировать, получится ли извлечь выгоду из той ситуации, в которой происходят транзакции.

### 2.3.5. Зачем изучать теорию игр?

Теория игр — это революционное междисциплинарное направление, объединяющее психологию, экономику, математику, философию и обширную смесь прочих научных областей.

Мы говорим, что теория игр связана с реальными проблемами. Однако проблемы безграничны. Значит, теоретические концепции игр также безграничны? Конечно! Мы опираемся на теорию игр каждый день, сознательно или неосознанно, потому что всегда включаем свой мозг, чтобы предпринять лучшие стратегические действия с учетом ситуации. Не так ли? Если это так, зачем изучать теорию игр?

В теории игр есть множество примеров, которые помогают нам научиться думать по-иному. Существует несколько проработаных теорий — таких как равновесие по Нэшу — которые имеют отношение ко многим реальным ситуациям. В настоящей жизни участники или игроки часто сталкиваются с матрицей решений, аналогичной дилемме заключенного. Таким образом, изучение этих концепций не только помогает нам сформулировать проблемы более математически, но и позволяет нам сделать лучший ход. Это дает нам возможность определить аспекты, которые каждый участник должен рассмотреть, прежде чем выбирать стратегическое решение в любом конкретном взаимодействии. Чтобы иметь возможность предпринимать лучшие действия, мы должны сначала определить тип игры: кто является игроками, каковы их цели или задачи, какими могут быть их ответные действия и т. д. Большая часть решений в реальной жизни основана на изучении поведения разных сторон, а теория игр обеспечивает основу для рационального принятия решений.

Проблема византийских генералов, которую мы изучали в предыдущем разделе, широко используется в приложениях распределенного хранения и в центрах обработки данных для обеспечения согласованности между вычислительными узлами.

## 2.4. Информатика

Как мы уже говорили, технология блокчейна — это умный инжиниринг с использованием концепций информатики, криптографии, теории игр и многих других областей знаний. В этом разделе мы изучим некоторые важные компоненты информатики, которые используются в блокчейне.

## 2.4.1. Хэш-указатель

Как мы уже знаем, блокчейн на самом деле является блочно-цепочной структурой данных — в том смысле, что это цепочка блоков, соединенных вместе. Когда мы говорим «блок», это может означать как одну транзакцию, так и несколько транзакций, объединенных вместе. Мы начнем наше обсуждение с хэш-указателей, которые являются основным строительным элементом структуры данных блокчейна.

*Хэш-указатель* — это криптографический хэш, указывающий на блок данных (рис. 2.28). В отличие от связанных списков, которые указывают на следующий блок, чтобы вы могли получить к нему доступ, хэш-указатели указывают на *предыдущий* блок данных и предоставляют способ убедиться, что данные не были подделаны.

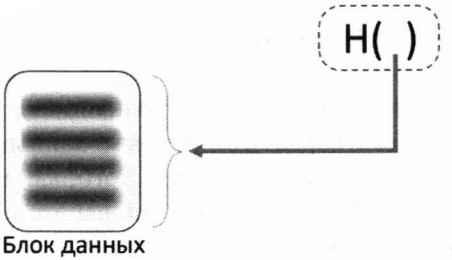



**Рис. 2.28.** Хэш-указатель для блока транзакций

Назначение хэш-указателя состоит в создании устойчивого к взлому блокчейна, который можно рассматривать как единый источник истины. Как блокчейн достигает этой цели? Вся суть в том, что хэш предыдущего блока сохраняется в заголовке текущего блока, а хэш текущего блока вместе с его заголовком будет храниться в заголовке следующего блока. В итоге и получается блокчейн (рис. 2.29).

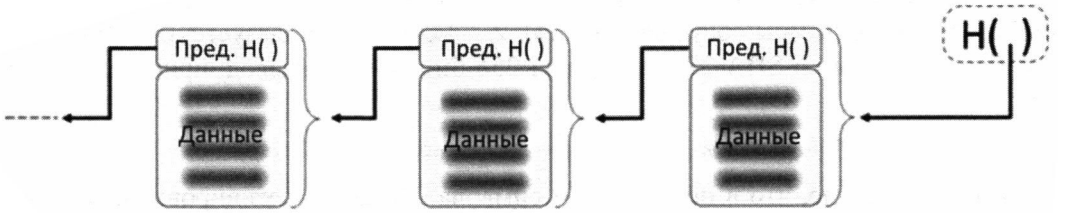



**Рис. 2.29.** Блоки в блокчейне связаны через хэш-указатели

Как мы можем видеть, каждый блок ссылается на предшествующий блок, именуемый *родительским блоком*. Каждый новый блок, который добавляется в цепочку, становится родительским блоком для следующего блока. По цепочке ссылок можно пройти весь путь до первого блока, который называется *блоком генезиса*. В такой схеме, где блоки связаны с помощью хэшей, практически невозможно незаметно изменить данные в любом блоке. Мы уже рассмотрели свойства хэш-функций, поэтому понимаем, что если данные будут изменены, хэш-коды не будут совпадать.

А что, если кто-то вместе с данными поменяет хэш? Давайте внимательно посмотрим на рис. 2.30, чтобы понять, почему невозможно каким-либо образом подменить данные, — любая попытка изменить содержимое заголовка или блока нарушает всю цепочку. Предположим, что вы изменили данные в блоке под номером 1234. Если вы это сделаете, хэш, который хранится в заголовке блока 1235, не будет совпадать с новым хэшем блока 1234.

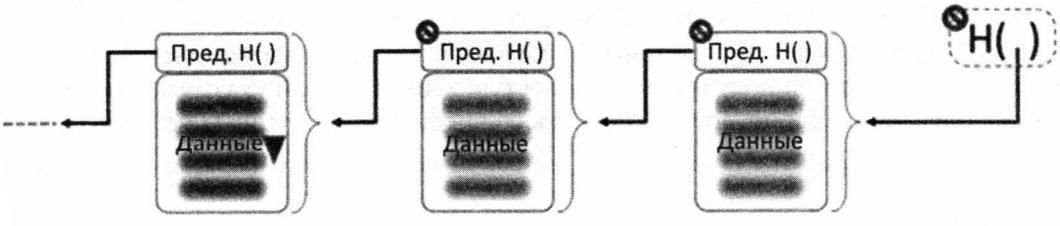



**Рис. 2.30. Механизм защиты от изменений блока**

Что случится, если вы также измените хэш, хранящийся в заголовке блока 1235, чтобы он полностью соответствовал измененным данным блока 1234? Другими словами, вы хэшируете блок данных 1234 после того, как изменили его, и подставляете новый хэш в блок 1235. Но после того, как вы это сделаете, хэш блока 1235 тоже изменится (потому что блок 1235 — это данные и заголовок вместе), и он перестанет совпадать с хэшем, хранящимся в заголовке блока 1236. Вам придется продолжать выполнять подмену вплоть до самого последнего блока или самого последнего хэша. Поскольку каждый узел или многие узлы сети уже имеют копию блокчейна вместе с самым последним хэшем, никоим образом невозможно одновременно взломать множество узлов и изменить все хэши за раз. Это и делает структуру данных блокчейна защищенной от взлома.

Очевидно, что каждый блок может быть уникально идентифицирован по его хэшу. Чтобы вычислить этот хэш, вы можете использовать семейство хэш-функций SHA-2 или SHA-3, которые мы обсуждали в *разд. 2.2.* Если вы используете для хэширования блоков алгоритм SHA-256, он выдаст 256-битный хэш-вывод, примерно такой, как:

```
000000000000000a73b6a2af7bad40ec3fc2a83dafd76ef15f3d1b71a7132765
```

Обратите внимание, что в нем всего 64 символа. Поскольку хэшированный вывод представлен шестнадцатеричными символами, а каждый шестнадцатеричный символ может быть представлен четырьмя битами, длина выходных данных составляет:

$$64 \times 4 = 256 \text{ битов.}$$

Обычно на практике 256-битный хэшированный вывод представляют именно в виде 64-х шестнадцатеричных символов.

При разработке прикладного блокчейн-решения вы должны решить, какова будет структура блока, т. е. размер блока, размеры разделов данных и заголовка, количе-

ство транзакций в блоке и т. д. Для существующих блокчейнов, таких как Bitcoin, Ethereum или Hyperledger, структура уже определена их разработчиками, и вы должны ясно понимать, что можно строить на этих платформах. Мы подробнее рассмотрим блокчейны Bitcoin и Ethereum далее в этой книге.

## 2.4.2. Дерево Меркла

Дерево Меркла — это двоичное дерево криптографических хэш-указателей, конечные узлы которого — это хэши транзакций, а внутренние вершины представляют собой результаты сложения значений связанных вершин, — т. е. это бинарное дерево хэша. Оно названо так в честь своего изобретателя Ральфа Меркла[16]. Это еще одна полезная структура данных, используемая в реализациях блокчейна, таких как Bitcoin. Деревья Меркла создаются путем хэширования парных данных (обычно транзакций на уровне «листьев»), а затем повторного хэширования парных хэшей следующего уровня и так вплоть до корневого узла, называемого *корнем Меркла*. Как и любое другое логическое дерево, оно строится снизу вверх. В блокчейне Bitcoin листья дерева (конечные узлы) всегда являются транзакциями одного блока в блокчейне. Вскоре мы обсудим преимущества использования деревьев Меркла, чтобы вы сами могли решить, должны ли быть листья отдельными транзакциями или группой транзакций в блоках. Типичное дерево Меркла может быть представлено, как показано на рис. 2.31.

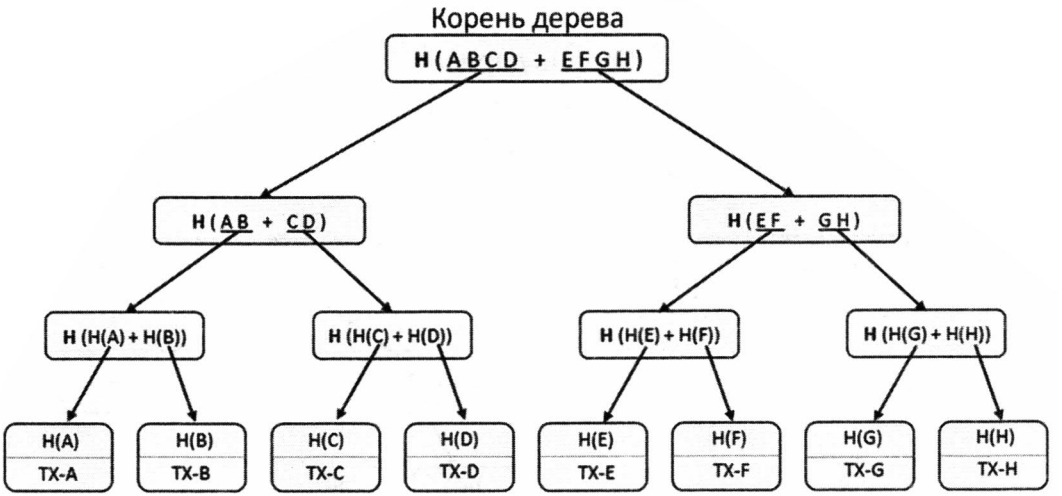



**Рис. 2.31. Структура дерева Меркла**

Подобно структуре данных указателя хэша, дерево Меркла также защищено от взлома. При наличии поддельной транзакции хэш на любом уровне дерева не будет совпадать с хэшем, хранящимся на одном уровне вверх по иерархии, а также

[16] Merkle — читается как «Меркл», соответственно, дерево Меркла.

вплоть до корневого узла. Злоумышленнику действительно трудно изменить все хэши во всем дереве. Это также обеспечивает целостность порядка транзакций. Если вы измените только порядок транзакций (даже не меняя сами транзакции), то также изменятся и хэши в дереве до самого корня Меркла.

Здесь есть одна тонкость. Дерево Меркла — это бинарное дерево, и на уровне листьев должно быть четное количество элементов. Что делать, если имеется нечетное количество транзакций? Хорошим решением является дублирование хэша последней транзакции. Это не создаст проблемы, такие как двойные расходы или повторные транзакции, поскольку мы дублируем именно хэш, а не саму транзакцию. Таким образом, мы можем сбалансировать дерево.

Допустим, нам надо найти транзакцию по ее хэшу или проверить, произошла ли транзакция в прошлом. Как мы проверим наличие такой транзакции в блокчейне? Самый простой способ — двигаться по цепочке в сторону генезиса до тех пор, пока вы не встретите блок, содержащий транзакцию. Очевидно, что в огромном блокчейне с миллионами транзакций такой поиск перебором требует много времени и ресурсов. Это тот случай, когда дерево Меркла оказывается чрезвычайно полезным.

Деревья Меркла обеспечивают очень эффективный способ проверки, принадлежит ли конкретная транзакция определенному блоку. Если на дереве Меркла есть $n$ транзакций (элементов-листьев), то эта проверка требует только $\log_2(n)$ вычислений (рис. 2.32).

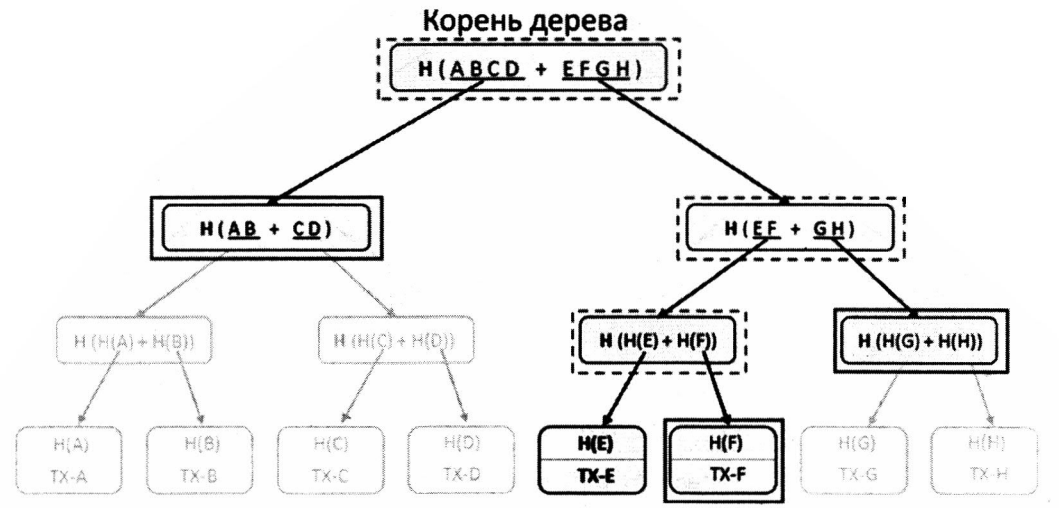



**Рис. 2.32.** Верификация в дереве Меркла

Чтобы проверить, принадлежит ли транзакция или какой-либо другой конечный элемент (лист) к дереву Меркла, нам не нужны все элементы и полное дерево. Скорее, необходимо его подмножество, как мы можем видеть на схеме, приведенной на рис. 2.32. Мы начинаем вычисления с транзакции, чтобы проверить ее вместе с ее

парным элементом (это ведь бинарное дерево, так что обязательно будет парный элемент), вычисляем хэш этой пары и смотрим, соответствует ли он их родительскому хэшу. Затем берем этот родительский хэш и его напарника по уровню, соединяем их вместе, чтобы получить и проверить родительский хэш более высокого уровня. Прохождение этого процесса вплоть до верхнего корневого хэша — это самый быстрый способ проверки транзакции: только $\log_2(n)$ раз для $n$ элементов. На рис. 2.32 необходимо знать только содержимое сплошных прямоугольников, а содержимое пунктирных прямоугольников может быть вычислено, исходя из данных в сплошных прямоугольниках. Поскольку в данном блоке восемь транзакций ($n = 8$), для проверки потребуются только три вычисления ($\log_2(8) = 3$).

Хорошо, а как насчет гибрида структуры данных блокчейна и дерева Меркла? Представьте себе ситуацию: блокчейн, где в каждом блоке много транзакций. Поскольку это блокчейн, хэш предыдущего блока уже известен. Теперь можно ускорить проверку всех транзакций в блоке, проходя до корня дерева Меркла. Если нам нужно проверить транзакцию, которая, как утверждается, была получена, скажем, из блока 22 456, мы можем взять все транзакции этого блока, проверить дерево Меркла, составленное из этих транзакций, и быстро подтвердить, является ли эта транзакция действительной. Мы уже убедились, что проверка транзакции с помощью дерева Меркла весьма проста и быстра. Хотя блоки в блокчейне устойчивы к несанкционированному изменению и не предоставляют даже малейших возможностей для изменения *содержимого* блока, дерево Меркла обеспечивает сохранение *порядка транзакций* внутри блока.

В типичной сети блокчейна может возникнуть много ситуаций, когда некий узел (для простоты предположим, что это любой узел, который не имеет полных данных блокчейна, т. е. *легкий узел*) должен проверить, имела ли место определенная транзакция в прошлом. Здесь на самом деле нужно проверить две вещи: транзакцию, как часть блока, и блок, как часть блокчейна. Для этого узел не должен загружать все транзакции блока, он может просто запросить у сети информацию, относящуюся к хэшу блока и хэшу транзакции. Узлы в сети, имеющие соответствующую информацию, могут в ответ сообщить путь Меркла для этой транзакции. Впрочем, вы можете спросить: как доверять данным, которыми поделился с вами неизвестный узел в сети? Вы уже знаете, что хэш-функции являются односторонними. Таким образом, злонамеренный узел не может подделать транзакцию, которая бы соответствовала заданному хэш-значению.

Использование деревьев Меркла не ограничивается только блокчейнами. Они широко используются во многих других приложениях — таких как BitTorrent, Cassandra — база данных NoSQL, Apache Wave и пр.

## 2.4.3. Сниппеты кода для дерева Меркла

В листинге 2.4 приведены сниппеты кода для дерева Меркла. Этот раздел предназначен для того, чтобы дать вам общее представление о том, как кодировать дерево Меркла на самом базовом уровне. Сниппеты (примеры кода) написаны на Python,

**но будут выглядеть очень похоже и на других языках. Вам просто нужно найти библиотечные функции для соответствующего языка.**

**Листинг 2.4. Сниппеты для реализации дерева Меркла**

```
# -*- coding: utf-8 -*-

from hashlib import sha256

class MerkleTree(object):
        def __init__(self):
                pass

        def chunks(self,transaction,n):
        #Эта функция генерирует количество n транзакций за раз
        for i in range (0, len(transaction),number):
                yield transaction[i:i+2]

        def merkle_tree(self,transactions):
        #Здесь мы находим хэш дерева Меркла для всех
        #транзакций, переданных в эту функцию.
        #Задача решается применением рекурсии.
        #Берем список транзакций, объединяем их по две,
        #вычисляем хэш группы, храним этот хэш.
        #Затем снова группируем попарно, опять вычисляем
        #хэш пары и так до тех пор, пока не останется один хэш.
        sub_tree=[]
        for i in chunks(transactions,2):
                if len(i)==2:
                        hash = sha256(str(i[0]+i[1])).hexdigest()
                else:
                        hash = sha256(str(i[0]+i[0])).hexdigest()
                sub_tree.append(hash)
        #Если поддерево содержит только один хэш,
        #значит мы достигли корня дерева.
        #В противном случае рекурсивно вызываем функцию.
        if len(sub_tree) == 1:
                return sub_tree[0]
        else:
                return self.merkle_tree(sub_tree)

if __name__=='__main__':
        mk=MerkleTree()
        merkle_hash= mk.merkle_tree(["TX1","TX2","TX3","TX4","TX5","TX6"])
        print merkle_hash
```

## 2.5. Обобщаем знания

Добираясь до этого раздела, мы рассмотрели все необходимые компоненты блокчейна. Изучив концепции криптографии, теории игр и информатики, мы можем теперь выработать представление о том, как работает блокчейн. Хотя эти концепции существуют уже давно, до сих пор никто не мог представить, как можно использовать старый фундамент для создания настолько революционной технологии, как блокчейн. Давайте кратко повторим основные идеи, которые мы рассмотрели, и закрепим полученные знания:

- криптографические функции хэширования являются односторонними и не могут быть инвертированы. Они являются детерминированными и производят одинаковый результат для определенных входных данных. Малейшие изменения входных данных приведут к совершенно другому выходу при повторном хэшировании;
- при использовании криптографии с открытым ключом возможна генерация цифровой подписи. Это помогает проверить подлинность подписавшегося лица или организации. Учитывая, что закрытый ключ хранится в тайне, подделать подпись другой персоны не представляется возможным. Кроме того, если кто-то подписал какой-либо документ или транзакцию, он не может позже отрицать, что сделал это;
- используя принципы теории игры и лучшие стратегии, можно разработать надежные системы, которые способны выдержать большинство проблемных ситуаций. Системы, которые могут столкнуться с проблемой византийских генералов, должны быть разработаны с учетом этой проблемы. Наш подход к проектированию любой системы должен быть таким, чтобы участникам было выгодно играть по правилам, чтобы получить максимальную отдачу. Отклонение от протокола не должно приносить им пользу;
- структура данных блокчейна с применением криптографических хэшей обеспечивает устойчивую к взлому цепочку блоков. Использование деревьев Меркла делает проверку транзакций проще и быстрее.

Замечательно, вы усвоили теоретические основы. Давайте теперь подумаем о практической реализации блокчейна. Какие проблемы вы должны решить, чтобы децентрализованная сеть блокчейна работала должным образом? На самом деле, таких проблем много. Некоторые из них будут общими для большинства случаев использования блокчейна, а некоторые уникальны для отдельных приложений. Давайте обсудим, по крайней мере, некоторые из сценариев, которые необходимо иметь в виду:

- кто будет вести распределенный реестр транзакций? Должны ли все участники (узлы сети) поддерживать и хранить реестр, или это будут делать только некоторые? Как насчет вычислительных узлов, которые недостаточно мощны для обработки транзакций или не имеют достаточно места для хранения всей истории транзакций?

- как поддерживать единое согласованное состояние распределенного реестра? Задержки сети, потери пакетов, преднамеренные попытки взлома и прочие технические проблемы неизбежны. Как система выдержит все это?
- кто будет проверять или аннулировать транзакции? Будут ли транзакции проверяться только некоторыми авторизованными узлами, или все узлы вместе должны достигнуть консенсуса? Что, если некоторые узлы недоступны в данный момент?
- что делать, если некоторые вычислительные узлы намеренно хотят подорвать систему или пытаются необоснованно отклонить некоторые транзакции?
- как вы собираетесь обновлять систему, если нет центрального органа, который бы взял на себя ответственность за управление обновлением? Что делать, если в децентрализованной сети несколько вычислительных узлов обновляются сами, а остальные не могут или, хуже того, не хотят обновляться?

На самом деле существует гораздо больше проблем, которые необходимо решать, кроме только что упомянутых. Пока мы оставим вас наедине с этими мыслями, но большинство из вопросов будут разъяснены к концу этой главы.

Давайте начнем с некоторых фундаментальных свойств системы блокчейна, которые могут потребоваться для разработки любого децентрализованного решения.

## 2.5.1. Свойства блокчейн-решений

До сих пор мы изучали только технические аспекты устройства блокчейна, чтобы понять, как он работает. В этом разделе мы обсудим некоторые из важных свойств системы, основанной на блокчейне. Потребитель ожидает от разработчика реализацию этих свойств.

### Неизменность

Это наиболее желаемое свойство для поддержания *атомарности транзакций* блокчейна. Как только транзакция записана в реестр, она не может быть изменена. Если транзакции передаются в сеть, то почти у каждого узла есть их копия. Со временем, когда в цепочку добавляется все больше и больше блоков, неизменность увеличивается, и через некоторое время цепочка становится полностью неизменной. Изменить данные такого количества блоков подряд практически невозможно, потому что они криптографически защищены. Таким образом, любая транзакция, которая регистрируется в реестре, навсегда остается в системе в неизменяемом состоянии.

### Стойкость к подделке

Децентрализованное решение, при котором транзакции являются публичными, подвержено различным видам атак. Попытки подделки являются наиболее очевидными из всех — особенно, когда вы совершаете сделку с чем-то ценным. Для обеспечения устойчивости системы к подделке могут использоваться цифровые подпи-

си и криптографический хэш. Мы уже знаем, что вычислительными методами невозможно подделать чужую подпись. Если вы совершаете транзакцию и подписываете ее хэш, никто не сможет изменить транзакцию позже и сказать, что вы подписали другую транзакцию. Кроме того, позже вы не сможете утверждать, что никогда не совершали транзакцию, потому что вы подписали ее.

## Демократичность

Любая одноранговая децентрализованная система должна быть демократичной по своему замыслу (но это может быть не в полной мере применимо к *частному блокчейну*, который мы обсудим позже). В системе не должно быть какой-либо сущности, более влиятельной, чем другие. Каждый участник должен иметь равные права в любой ситуации, и решения принимаются, когда большинство достигает консенсуса.

## Устойчивость к двойным расходам

Атаки с двойным расходом довольно распространены как в денежных, так и в неденежных операциях. В мире криптовалюты попытка двойного расходования — это попытка потратить одну и ту же сумму на несколько человек. Например, на вашем счете есть 100 долларов, и вы платите по 90 долларов двум или более сторонам. Ситуация немного отличается, когда дело доходит до криптовалюты — такой как биткойн — где нет понятия конечного баланса. Вход в транзакцию — это результат предыдущей транзакции, в которой вы получили хотя бы ту сумму, которую хотите потратить в текущей транзакции. Предположим, что Боб некоторое время назад получил 10 долларов от Алисы (входящая транзакция). Сегодня Боб хочет заплатить Чарли 8 долларов, и транзакция, в которой он получил 10 долларов от Алисы, станет входящим остатком для сделки с Чарли. Таким образом, Боб не может использовать один и тот же вход (Алиса заплатила ему 10 долларов) несколько раз, чтобы заплатить другим людям и удвоить расходы. Другим примером двойного расхода может стать ситуация, когда кто-то владеет землей и продает один и тот же участок земли двум людям.

В централизованной системе довольно легко предотвратить двойные расходы, потому что центральный орган знает обо всех транзакциях. Реализация блокчейна также должна быть невосприимчивой к таким атакам. Хотя криптография обеспечивает подлинность транзакции, она не может предотвратить двойные расходы, потому что и обычная транзакция, и транзакция с двойным расходом являются криптографически подлинными. Таким образом, единственный способ предотвратить двойные расходы — это знать обо всех транзакциях. Если нам известны все транзакции, которые произошли в прошлом, мы можем выяснить, является ли текущая транзакция попыткой удвоить расходы. Следовательно, узлам, которые будут проверять транзакции, обязательно должны быть доступны все данные блокчейна, начиная с блока генезиса.

### Согласованное состояние реестра

Свойства, которые мы только что обсудили, гарантируют, что реестр в определенных ситуациях не будет противоречить сам себе. Представьте ситуацию, когда некоторые узлы намеренно хотят, чтобы транзакция не проходила и была отклонена. Или если по какой-то причине некоторые узлы не синхронизированы с реестром и, следовательно, не знают о нескольких транзакциях, которые проведены, пока они находились в автономном режиме, то для них транзакция может выглядеть как мошенническая. Следовательно, обеспечение консенсуса среди участников это то, с чем нужно обращаться очень осторожно. Вспомните проблему византийских генералов. Правильный вид консенсуса, подходящий для конкретной ситуации, играет важнейшую роль для обеспечения стабильности децентрализованной системы. Мы изучим различные механизмы консенсуса позже в этой книге.

### Жизнестойкость

Сеть должна быть достаточно устойчивой, чтобы выдерживать временные сбои узлов, иногда недоступность некоторых вычислительных узлов, задержку в сети и потери пакетов, атаки с преднамеренной перегрузкой и тому подобные затруднения.

### Проверяемость

Блокчейн — это цепочка блоков, которые связаны друг с другом через хэши. Поскольку блоки транзакций связаны вплоть до блока генезиса, возможность аудита уже предусмотрена системой, но мы должны гарантировать работоспособность блокчейна любой ценой. Кроме того, если кто-то хочет проверить, имела ли место транзакция в прошлом, то такая проверка должна быть максимально быстрой[17].

## 2.5.2. Транзакции и блокчейн

Когда мы говорим «блокчейн», обычно мы имеем в виду блокчейн транзакций, верно? Таким образом, все начинается с транзакции, а затем транзакция проходит через ряд шагов и в конечном итоге оказывается в блокчейне. Блокчейн является реализацией одноранговой сети, и если вы имеете дело с системой, в которой каждую секунду происходит много транзакций, вряд ли вы захотите переполнить сеть, транслируя все транзакции подряд. Очевидно, что когда физическое или юридическое лицо совершает транзакцию, ему нужно сообщить ее всей сети. Как только это произойдет, транзакция должна быть проверена несколькими узлами. После проверки необходимо снова передать широковещательную рассылку на всю сеть, чтобы транзакция была включена в цепочку блоков. Теперь задумайтесь, зачем вы транслируете в сеть цепочку транзакций вместо блоков? Пожалуй, это имеет смысл, если в вашей экономической деятельности не используется много транзакций. Однако если каждую секунду происходит огромное количество сделок, то хэ-

[17] От скорости проверки зависит скорость прохождения следующей транзакции.

ширование на уровне отдельных транзакций, отслеживание и передача их в сеть могут привести к нестабильной работе системы. Трансляция отдельных транзакций может стать дорогостоящим делом! Но вы можете сгруппировать определенное количество транзакций в блок и транслировать этот блок. Еще одна веская причина для формирования цепочки блоков вместо цепочки транзакций — предотвращение атаки Сибиллы[18] (Sybil Attack). В *главе 3* вы узнаете более подробно, как используется алгоритм PoW-майнинга и случайным образом выбирается один узел, который может предложить сети блок. Если бы не было механизма выбора узла, люди могли бы создавать множественные реплики своего собственного узла, чтобы подорвать систему.

В наиболее упрощенной форме транзакции проходят следующие шаги, чтобы попасть в блокчейн:

- каждая новая транзакция транслируется в сеть, чтобы все вычислительные узлы знали об этом факте в тот момент, когда она имела место (чтобы обеспечить устойчивость системы к двойным расходам);
- транзакции могут быть подтверждены или отклонены узлами путем проверки подлинности;
- узлы могут затем группировать несколько транзакций в блоки для последующей обработки и включения в цепочку блоков;
- далее возникает сложная ситуация. Кто должен предложить блок транзакций, который узлы сгруппировали каждый сам по себе? Вообще говоря, генерация новых блоков должна быть управляемой, но не централизованно, и механизм управления должен быть таким, чтобы каждому узлу был присвоен равный приоритет. Согласие всех узлов по поводу блока называется *консенсусом*, но в зависимости от вашего варианта использования блокчейна существуют разные алгоритмы для достижения одной и той же цели. Мы обсудим различные механизмы консенсуса в следующем разделе;
- хотя не существует понятия всеобщего времени системы из-за задержек в сети, потерь пакетов и географического местоположения, наша система вполне работоспособна, потому что блоки добавляются строго по порядку, один за другим. Таким образом, мы можем считать, что блоки имеют метки времени в порядке их поступления и добавления в цепочку блоков;
- как только узлы в сети единодушно или большинством голосов принимают блок (приходят к консенсусу), этот блок добавляется в цепочку блоков, и к нему присоединяется хэш блока, созданного непосредственно перед ним. В результате блокчейн удлиняется на один блок.

Мы уже обсуждали структуру данных блокчейна и деревья Меркла, поэтому теперь мы понимаем их ценность. Вспомните, что когда узел хочет проверить транзакцию,

---

[18] Вид атаки в одноранговой сети, в результате которой жертва подключается только к узлам, контролируемым злоумышленником. Названа по имени героини книги «Сивилла», страдавшей раздвоением личности.

он может сделать это более эффективно с помощью пути Меркла. Другим узлам сети незачем транслировать полный блок данных, чтобы обосновать принадлежность транзакции к блоку. С технической точки зрения в таких сценариях широко используются эффективные в плане использования памяти и удобные для вычислений структуры данных — такие как фильтры Блума для проверки принадлежности элемента к множеству.

Также обратите внимание, что для того чтобы узел мог проверять любую транзакцию, он в идеале должен иметь все данные блокчейна (транзакции вместе с их метаданными) в виде локальной копии. При разработке блокчейна вы должны выбрать эффективный механизм хранения данных.

## 2.5.3. Механизмы распределенного консенсуса

Если узлы знают всю историю транзакций, имея локальную копию полных данных блокчейна для предотвращения двойных расходов, и они могут проверить подлинность транзакции с помощью цифровых подписей, какой тогда смысл в консенсусе? Представьте себе наличие одного или нескольких вредоносных узлов. Разве они не могут сказать, что недействительная транзакция является действительной или наоборот? Вспомните проблему византийских генералов, которая чаще всего возникает во многих децентрализованных системах. Чтобы преодолеть такие проблемы, нам нужен надлежащий механизм консенсуса.

Пока что в нашем обсуждении не ясно, кто предлагает новый блок. Очевидно, что не все узлы одновременно должны предлагать блок остальным узлам, потому что возникнет беспорядок, и можно забыть о согласованном состоянии реестра. С другой стороны, если речь идет о простых транзакциях без группировки их в блоки, то можно утверждать, что если каждая транзакция будет транслироваться по всей сети, и каждый узел в сети будет голосовать за отдельные транзакции, это существенно усложнит систему и приведет к снижению производительности.

Таким образом, группировка транзакций в блоки важна по соображениям производительности, а для каждого блока требуется консенсус. Лучшая стратегия решения проблемы состоит в том, что только один узел должен предлагать блок за раз, а остальные узлы должны проверять транзакции в блоке и добавлять его в свою копию блокчейна, если транзакции допустимы. Мы знаем, что у каждого узла есть собственная копия реестра и нет централизованного источника для синхронизации. Таким образом, если какой-либо один узел предлагает блок, и остальные узлы согласны с ним, то все эти узлы добавляют новый блок в свои копии цепочки блоков. В такой схеме желательно, чтобы интервал между появлениями блоков составлял не менее нескольких минут, и не должно быть случаев, когда несколько блоков поступают одновременно. А теперь вопрос: кто может стать тем счастливым узлом, который предлагает блок? Это самая сложная часть системы, успешная реализация которой приводит нас к надлежащему консенсусу. Мы обсудим этот аспект в рамках различных механизмов консенсуса.

Все механизмы консенсуса на самом деле происходят из теории игр. Ваша система должна быть спроектирована таким образом, чтобы узлы получали наибольшую

выгоду, если они играют по правилам. Одним из аспектов обеспечения правильного поведения узлов является вознаграждение за честное поведение и наказание за мошеннические действия. Однако здесь есть одна загвоздка. В открытом блокчейне, таком как Bitcoin, можно иметь много разных публичных идентификаторов (адресов), и они совершенно анонимны. Очень трудно наказать злоумышленников, потому что они могут избежать наказания, создавая для себя новые цифровые личности. С другой стороны, в такой ситуации прекрасно работает вознаграждение, потому что даже если у кого-то есть несколько личностей, он может с радостью собирать заслуженные награды, выданные этим личностям. Таким образом, решение зависит от вашего бизнес-кейса: если идентификационные данные являются анонимными, то наказание мошенника может не сработать. Зато хорошо сработает вознаграждение, если у вас есть возможность его предоставить. Наверняка при разработке своей системы вы захотите обдумать этот вопрос вознаграждения/наказания, несмотря на наличие отличного механизма выбора узла, который будет предлагать следующий блок. Дело в том, что вы никогда не узнаете заранее, является ли выбранный узел вредоносным или честным. А пока запомните термин *майнинг*[19], который мы будем использовать весьма часто. Он означает создание новых блоков.

Цель консенсуса также состоит в том, чтобы гарантировать, что сеть достаточно устойчива для отражения различных атак. Независимо от того, какой алгоритм консенсуса выбран исходя из назначения блокчейна, он должен формировать матрицу решений, устойчивую к проблеме византийских генералов. Давайте теперь изучим некоторые механизмы достижения консенсуса в системе блокчейна, которые мы можем использовать в различных ситуациях.

## Доказательство работы (PoW)

Механизм консенсуса на основе *доказательства работы* (Proof of Work, PoW) существует уже давно. Однако то, как он наряду с другими концепциями использовался в блокчейне Bitcoin, сделало его еще более популярным. Мы обсудим этот механизм консенсуса на его базовом уровне и посмотрим, как он реализован в Bitcoin, в *главе 3*.

Идея алгоритма PoW заключается в том, что для блока транзакций выполняется определенная работа, прежде чем он будет предложен всей сети. Результат PoW представляет собой фрагмент данных, который трудно «добыть» с точки зрения вычислений и времени, но который легко проверить. Одним из старых способов использования PoW было предотвращение спама в электронной почте. Если до того, как можно будет отправить электронное письмо, необходимо выполнить определенную работу, то для рассылки обширного спама потребуется много вычислений. Это может помочь предотвратить спам по электронной почте. Аналогично, если перед созданием блока необходимо выполнить некоторую сложную работу, требующую большого объема вычислений, это полезно по двум соображениям: во-

[19] Mining — добыча полезных ископаемых (*англ.*).

первых, это определенно займет некоторое время, а во-вторых, если узел попытается внедрить мошенническую транзакцию в блок, то отклонение этого блока остальными узлами слишком дорого обойдется для мошенника[20]. Ведь вычисления, выполненные мошенником для получения PoW, окажутся напрасными.

Просто подумайте о том, что было бы, если бы мошенник мог предложить блок без особых усилий, а не выполнять тяжелую работу. Если бы это не требовало затрат, то предложение узла с мошеннической транзакцией и получение отказа не было бы большой проблемой. Люди могли бы просто продолжать предлагать такие блоки в надежде, что какой-нибудь из них когда-нибудь сможет пройти проверку и добраться до блокчейна. Напротив, выполнение некоторой сложной работы по формированию блока препятствует тому, чтобы узел вбрасывал мошеннические транзакции легким способом.

Кроме того, сложность работы должна регулироваться таким образом, чтобы можно было контролировать скорость генерации блоков. Вы, должно быть, задумались: если речь идет о какой-то работе, требующей некоторых вычислений и времени, какой должна быть эта работа? Это очень простая, но в то же время трудная работа. Рассмотрим пример такой работы. Представьте себе задачу, когда вам нужно найти такое число, результат хэширования которого будет начинаться с буквы «а». Как бы вы это сделали? Мы знакомы с хэш-функциями и знаем, что у них нет ярлыков (заранее известных путей достижения результата). Так что вам придется просто угадывать число. Возможно, вы возьмете некое начальное число, будете увеличивать его на единицу и каждый раз хэшировать, чтобы проверить, начинается ли хэш с буквы «а». Если необходимо повысить уровень сложности, можно сказать, что хэш должен начинаться с трех последовательных «а». Очевидно, что найти решение для чего-то вроде «ахххххххх» легче по сравнению с «аааххххх», потому что второй вариант более ограничен.

В только что приведенном примере, если несколько разных узлов работают над решением такой вычислительной задачи, вы никогда не знаете, какой узел решит ее первым. Это можно использовать для выбора случайного узла (на этот раз он действительно случайный, потому что выбор не основан на детерминированном алгоритме), который первым решает головоломку и предлагает блок. Чрезвычайно важно отметить, что в случае публичных блокчейнов узлы, которые вкладывают свои вычислительные ресурсы, должны быть щедро вознаграждены за честное поведение, иначе будет трудно поддерживать такую систему в рабочем состоянии.

## Доказательство владения долей (PoS)

Алгоритм доказательства *владения долей* (Proof of Stake, PoS) является еще одним алгоритмом, который весьма популярен при достижении распределенного консенсуса. Однако хитрость в том, что речь идет не о майнинге, а о проверке блоков транзакций. Вознаграждение за майнинг в виде генерации новых монет отсутству-

[20] Дорого в буквальном смысле, если помнить о стоимости электроэнергии и оборудования для майнинга.

ет, майнеры получают только комиссионные за транзакции (точнее, не майнеры, а *валидаторы*, но мы будем называть их майнерами для простоты понимания).

В системах с алгоритмом PoS майнеры должны привязывать свою долю (вносить в залог сумму криптовалюты, как доказательство владения долей), чтобы иметь возможность участвовать в проверке транзакций. Вероятность того, что майнер создаст блок, пропорциональна его доле: чем больше сумма в залоге, тем больше у него шансов проверить новый блок транзакций. Майнеру нужно только доказать, что он владеет определенным процентом всех монет, доступных в определенное время в той или иной валютной системе. Например, если майнеру принадлежит 2% всего эфира (ETH) в сети Ethereum, он сможет добывать в Ethereum 2% всех транзакций. От используемого вами алгоритма PoS зависит, кто из майнеров-дольщиков будет создавать новый блок транзакции. На самом деле существуют разные варианты алгоритма PoS — такие как наивный PoS, делегированный PoS, PoS на основе цепочки, PoS в BFT-стиле и Casper PoS. Делегированный PoS (Delegated PoS, DPOS) используется в Bitshares, а Casper PoS разрабатывается для использования в Ethereum.

Поскольку создатель блока в системе PoS является детерминированным (в зависимости от стоимости доли), он работает намного быстрее по сравнению с системами PoW. Кроме того, поскольку нет никаких вознаграждений за блок, а только комиссионные за транзакции, весь объем цифровой валюты должен быть создан при запуске системы, и общая сумма валюты фиксирована на всем протяжении работы.

Системы PoS могут обеспечить лучшую защиту от злонамеренных атак, поскольку выполнение успешной атаки просто дестабилизирует блокчейн и ставит под угрозу все деньги, включая долю мошенника. Кроме того, поскольку алгоритм PoS не требует сжигания большого количества электроэнергии и использования мощных процессоров, он получает приоритет над системами PoW везде, где это применимо.

## Алгоритм PBFT

Протокол консенсуса, *устойчивый к византийской ошибке* (Practical Byzantine Fault Tolerance, PBFT), — это еще один из многих алгоритмов достижения консенсуса, который следует иметь в виду при разработке прикладного блокчейна. Например, блокчейны Hyperledger, Stellar и Ripple используют консенсус PBFT.

В алгоритме PBFT, подобно алгоритму PoS, не используется вознаграждение за майнинг. Однако технические особенности их реализации различны. Обсуждение деталей внутреннего устройства PBFT выходит за рамки этой книги, но на высоком уровне запросы передаются всем участвующим узлам, которые имеют свои собственные реплики блокчейна (внутренние состояния). Когда узлы получают запрос, они выполняют вычисления на основе своих внутренних состояний. Результаты вычислений затем передаются всем остальным узлам системы. Таким образом, каждый узел знает результаты вычислений других узлов. Сравнивая свои собственные результаты вычислений с результатами, полученными от других узлов, они принимают решение и фиксируют окончательное значение, которое снова рассыла-

ется между узлами. В этот момент каждый узел знает об окончательном решении всех остальных узлов. Затем на основе большинства достигается окончательный консенсус. Этот процесс схематически изображен на рис. 2.33.

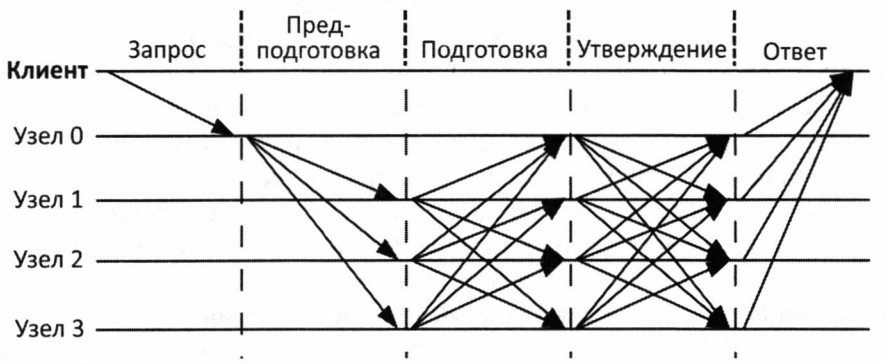



**Рис. 2.33. Механизм достижения консенсуса в PBFT**

PBFT может оказаться эффективнее по сравнению с другими алгоритмами консенсуса, однако анонимность в системе может быть нарушена из-за особенностей алгоритма. Это один из наиболее широко используемых алгоритмов для достижения консенсуса даже в неблокчейновых средах.

## 2.6. Применение блокчейна

Несмотря на то, что в этой главе мы рассмотрели основные аспекты внутреннего устройства блокчейна, важно также узнать, как он используется при создании прикладных решений. Существуют решения, которые используют блокчейн как базу данных для веб-сервера, и есть системы, которые полностью децентрализованы, т. е. не имеют центрального сервера. Например, Bitcoin — это блокчейн-система, в которой невозможно отправить запрос на сервер. Каждая транзакция транслируется на всю сеть. Тем не менее возможна ситуация, когда веб-приложение размещено на централизованном веб-сервере, и при необходимости вносит обновления в блокчейн Bitcoin. Посмотрите на рис. 2.34, где узел Bitcoin передает транзакции узлам, которые доступны в данный момент времени.

С точки зрения программного обеспечения каждый узел самодостаточен и поддерживает свою собственную копию базы данных блокчейна. Блокчейн Bitcoin можно считать эталонным образцом применения блокчейнов без централизованных серверов. Большинство из них являются полностью децентрализованными приложениями и подпадают под определение *публичный блокчейн*. Обычно для таких общедоступных блокчейнов еще не очень распространено использование ресурсов от поставщиков облачных услуг, таких как Microsoft Azure, IBM Bluemix и др. Однако для большинства закрытых (частных) блокчейнов поставщики облачных услуг становятся все более популярными. В качестве примера рассмотрим ситуацию, когда существует одно или несколько веб-приложений для разных отделов предприятия

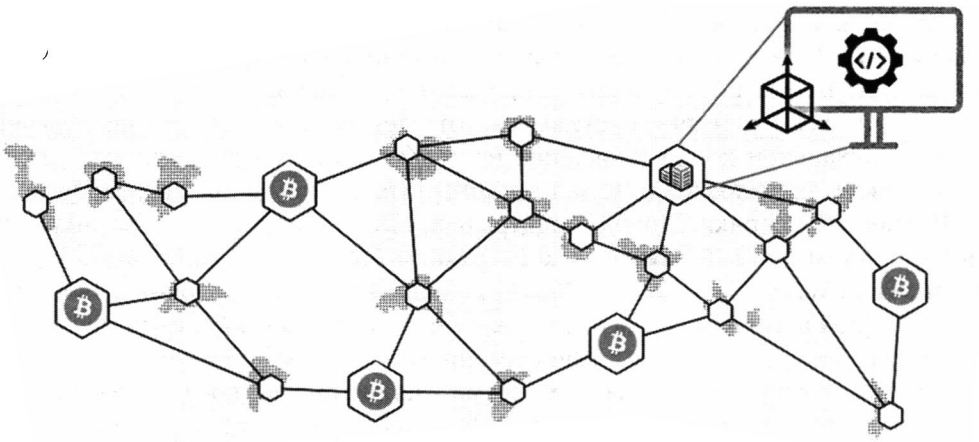


**Рис. 2.34.** Узлы блокчейна в сети Bitcoin

или действующих лиц, у всех из которых есть свои собственные блокчейны, но при этом блокчейны синхронизированы друг с другом. В таких условиях, хотя достигнута техническая децентрализация, с политической точки зрения система все еще может быть централизованной. Однако даже если контроль или управление навязаны извне, система все еще способна поддерживать прозрачность и доверие благодаря доступности единого источника правды. На рис. 2.35 в общем виде изображено большинство реализаций блокчейна, где цепочки блоков поддерживаются и размещаются неким провайдером облачного сервиса «блокчейн как услуга» (Blockchain as a Service, BaaS).

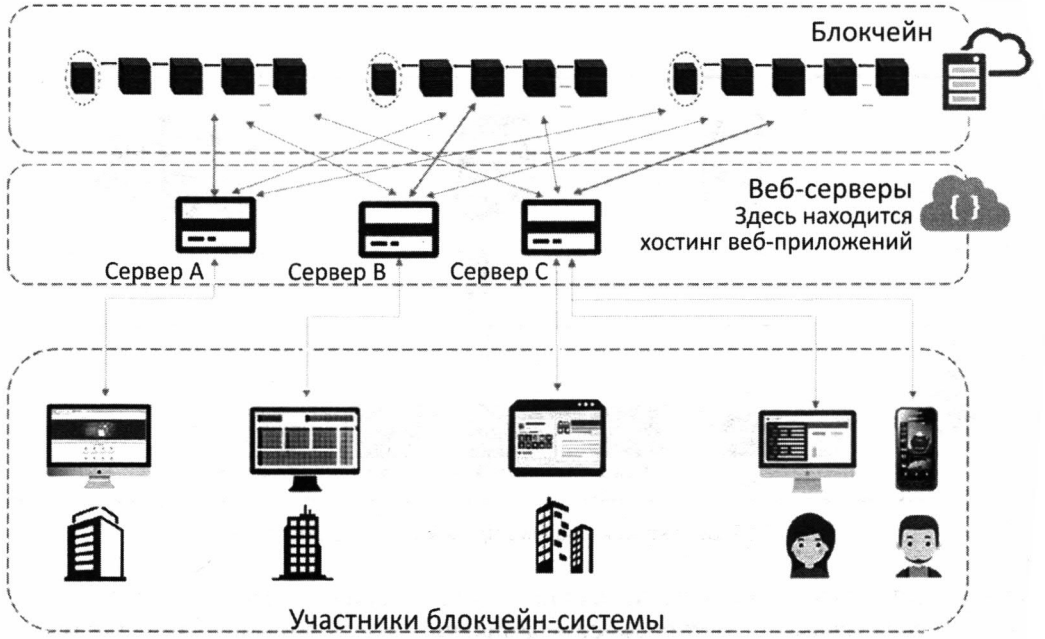



**Рис. 2.35.** Облачная блокчейн-система

Не обязательно, чтобы у всех участников системы было свое собственное веб-приложение. При наличии надлежащего механизма контроля доступа одно веб-приложение может обрабатывать запросы от нескольких разных участников. Хорошо, когда все участники системы имеют свои собственные копии блокчейнов. Наличие локальной копии блокчейна не только помогает поддерживать прозрачность в системе, но также помогает организовать анализ данных в реальном времени. Различные цепочки блоков, поддерживаемые и генерируемые разными участниками системы, согласованы по конструкции благодаря алгоритмам консенсуса, таким как PoW, PoS и тому подобным. Большинство частных блокчейнов предпочитают любой алгоритм согласования, кроме PoW, потому что стараются сократить потребление электроэнергии и вычислительную мощность, насколько это возможно. Механизм консенсуса типа PoS весьма распространен, когда речь идет о частных или корпоративных блокчейнах. Блокчейн революционно меняет многие аспекты бизнеса, и для облачного провайдера нет лучшего способа обеспечить прозрачность тарифов, чем использование бизнес-модели «плати по мере пользования», которая все больше набирает обороты. Облачные сервисы помогают компаниям пройти через цифровую трансформацию с минимальными первоначальными инвестициями.

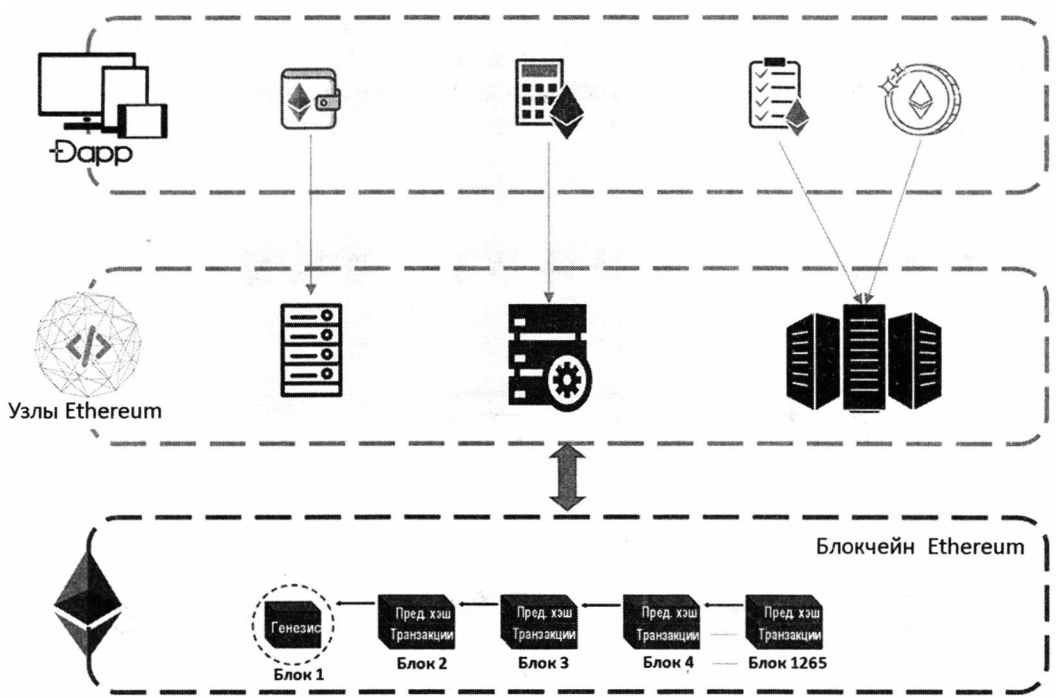



**Рис. 2.36.** Децентрализованные приложения в сети Ethereum

Существуют также децентрализованные приложения (Distributed Applications, DApps), создаваемые в сетях блокчейна Ethereum. Эти приложения могут быть разрешены в частном блокчейне Ethereum, но в то же время могут быть запрещены

в общедоступной сети Ethereum. Кроме того, эти приложения могут быть настроены для разных вариантов использования в одной общедоступной сети Ethereum. Позже в этой книге мы рассмотрим детали, относящиеся к Ethereum, а пока просто взгляните на рис. 2.36, чтобы представить, как могут выглядеть эти приложения.

Как уже говорилось в предыдущих разделах, разработка приложений блокчейна ограничена только вашим воображением. Можно разработать стандартное приложение на «чистом» блокчейне. Также создаются приложения, которые рассматривают блокчейн как серверную часть, или создаются гибридные решения, которые заменяют устаревшие приложения и используют блокчейн только для некоторых конкретных целей. С самого начала эпохи блокчейна и до сих пор одной из самых больших проблем является масштабируемость блокчейна. Хотя масштабируемость до сих пор остается предметом исследований, давайте изучим некоторые методы масштабирования блокчейна.

## 2.7. Масштабирование блокчейна

Мы рассмотрели блокчейн с исторической точки зрения и обсудили то, почему он оказался одной из самых революционных технологий сегодняшнего дня. Исследуя технические аспекты, мы узнали о проблемах масштабируемости, свойственных большинству разновидностей блокчейна. Блокчейны трудно масштабировать в силу особенностей устройства, и эта проблема остается предметом исследований в академических кругах и для некоторых инновационных корпораций. Если вы посмотрите на использование блокчейна Bitcoin в реальной жизни, то обнаружите, что он не используется для замены фиатных валют именно из-за проблем с масштабируемостью. Вы не можете оплатить кофе криптовалютой и подождать час, пока сделка не завершится. Поэтому биткойны используются главным образом в качестве разновидности активов для инвесторов. Сеть блокчейна Bitcoin на сегодняшний день не способна обслуживать столько транзакций, сколько Visa или MasterCard.

Вспомните протоколы консенсуса, которые мы изучали до сих пор, — такие как PoW для Bitcoin или Ethereum, или PoS и другие BFT-консенсусы некоторых других разновидностей блокчейнов, таких как Multichain, Hyperledger, Ripple или Tendermint. Основная цель этих алгоритмов консенсуса — византийская отказоустойчивость. Конструктивно каждый узел (по крайней мере, полные узлы) в сети блокчейна поддерживает в актуальном состоянии собственную копию всей цепочки блоков, проверяет все транзакции и блоки, обслуживает запросы от других узлов в сети и т. д. Все это делается ради достижения децентрализации, которая становится узким местом масштабируемости. Смотрите, какая ирония: в централизованной системе для масштабирования мы просто добавляем больше серверов, но то же самое неприменимо в децентрализованной системе, потому что с ростом количества узлов задержка только увеличивается! Хотя с увеличением числа узлов обычно возрастает уровень децентрализации, количество транзакций в сети также увеличивается, что приводит к повышенным требованиям к вычислительным ресурсам и

ресурсам хранения. Имейте в виду, что эта ситуация в большей степени относится к публичным блокчейнам и в меньшей степени присуща частным блокчейнам. Частные блокчейны легче масштабируются по сравнению с общедоступными, поскольку управляющие узлы могут оптимально распределять нагрузку по узлам с высокой вычислительной мощностью и большей пропускной способностью. Кроме того, некоторые вычислительные задачи можно вынести наружу и вычислять вне блокчейна, а это помогает системе лучше масштабироваться.

В этой главе мы изучим некоторые общие методы масштабирования, а методы масштабирования, специфичные для Bitcoin и Ethereum, обсудим в соответствующих главах. Пожалуйста, имейте в виду, что нельзя наобум брать все методы масштабирования и применять их к любым случаям использования блокчейна. Лучший способ применения — это тщательно изучить все технические приемы масштабирования и использовать наилучший из возможных методов в конкретной ситуации.

## 2.7.1. Вычисления вне блокчейна

Вычисления вне блокчейна являются одним из наиболее перспективных методов масштабирования блокчейн-решений. Идея состоит в том, чтобы ограничить использование блокчейна и выполнять тяжелую работу за его пределами, а сохранять в блокчейне только результаты. Заметим, что не существует стандартного определения того, как должны выполняться вычисления вне блокчейна. Реализация сильно зависит от задачи и людей, которые пытаются ее решить. Кроме того, разные варианты блокчейна могут потребовать разных подходов для вычислений вне цепочки. В общем виде это можно представить, как еще один слой поверх блокчейна, который выполняет тяжелую, требующую большого объема вычислений работу и разумно использует блокчейн. Очевидно, что вы не сможете сохранить все характеристики блокчейна, выполняя вычисления вне цепочки, но верно и то, что вам не обязательно нужен блокчейн для всех видов вычислений, и вы можете использовать его только в определенных критических точках вашей системы.

Вычисления вне блокчейна могут быть реализованы в *боковой цепочке* (сайдчейн, sidechain), могут быть распределены среди случайной группы узлов или могут быть централизованы. Боковые цепочки не зависят от основного блокчейна. Это не только помогает хорошо масштабировать блокчейн, но также изолирует повреждения в боковой цепочке и защищает от них основной блокчейн. Одним из таких примеров боковой цепочки является проект Lighting Network для Bitcoin, который должен помочь более быстро выполнять транзакции с минимальной комиссией — это полезно для микроплатежей. Другим примером боковой цепочки для Bitcoin является Zerocash, основной целью которого является не масштабируемость, а конфиденциальность. Если вы используете Zerocash для транзакций в биткойнах, вы не можете быть отслежены, и ваша конфиденциальность под защитой. Мы в этой книге ограничимся обсуждением общих методов масштабируемости и не будем подробно останавливаться на обсуждении масштабирования Bitcoin.

Очевидный вопрос, который может сейчас возникнуть, заключается в том, как люди будут проверять подлинность транзакций, если они отправлены вне цепочки?

Для создания действительной транзакции вам не нужен блокчейн как таковой. В *разд. 2.2* этой главы мы узнали о криптографии с асимметричным ключом, которая используется в блокчейне. Чтобы совершить транзакцию, вы должны быть владельцем личного ключа и подписать транзакцию. Как только транзакция создана, все преимущества блокчейна начинаются с момента попадания транзакции в блокчейн. С блокчейном Bitcoin невозможны двойные расходы, но есть и другие преимущества. Пока же мы хотим лишь подчеркнуть, что вы можете создать транзакцию, только если у вас есть закрытый ключ для вашей учетной записи.

Блокчейны типа Bitcoin являются блокчейнами без сохранения состояния в том смысле, что они не поддерживают балансовое состояние счета. В блокчейне Bitcoin все представлено в форме транзакции, и нет понятия *конечный баланс счета* как такового. Чтобы иметь возможность совершить транзакцию, вы должны использовать информацию о предыдущих транзакциях. Напротив, блокчейн Ethereum сохраняет состояние счета. Блоки в блокчейне Ethereum содержат информацию о состоянии всего блока, где баланс счета также является составной частью. Когда информацию о состоянии поддерживает каждый узел в сети, она занимает значительный объем. Это справедливо и для других блокчейнов, которые сохраняют состояние.

Давайте приведем пример для более ясного понимания. Алиса и Боб — две стороны, проводящие несколько транзакций между собой. Допустим, у них обычно 50 денежных транзакций в месяц. В блокчейне с отслеживанием состояния все эти отдельные транзакции будут содержать информацию о текущем состоянии, и она будет поддерживаться всеми узлами, что представляет собой большой объем избыточной информации. Для решения этой проблемы введена концепция *каналов состояния* (state channels). Идея состоит в том, чтобы обновлять блокчейн в соответствии с неким конечным результатом, скажем, в конце месяца или при достижении определенного порога транзакций, а не с каждой транзакцией.

Каналы состояния по существу являются двусторонним каналом связи между пользователями, объектами или услугами. Обмен данными абсолютно безопасен благодаря использованию криптографических методов. Чтобы понять, как это работает, взгляните на рис. 2.37.

Обратите внимание, что каналы состояния для вычислений вне блокчейна в основном являются частными и ограничены группой участников. Первым шагом вычислений является *блокировка состояния* блокчейна для всех участников. Это может быть схема MultiSig или блокировка на основе смарт-контракта. После блокировки участники совершают друг с другом транзакции, которые криптографически защищены. Все транзакции криптографически подписаны, что делает их проверяемыми, причем эти транзакции не сразу передаются в блокчейн. Как мы уже говорили, эти каналы состояния могут иметь определенный срок службы или могут быть привязаны к количеству транзакций, объему транзакций или любой другой количественной мере. Таким образом, в блокчейне фиксируется только окончательный результат транзакций, и это является последним шагом вычислений, который называется *раскрытием состояния*.


**Рис. 2.37.** Каналы состояния для вычислений вне блокчейна

Каналы состояния могут быть по-разному реализованы в различных приложениях, а их реализация фактически оставлена на усмотрение разработчиков. Это, безусловно, открывает нам обширные перспективы и делает каналы состояния одним из самых важных компонентов для широкого применения приложений блокчейна. Например, сеть Lightning была разработана для автономных вычислений и ускорения транзакций Bitcoin. Аналогично, для блокчейна Ethereum была разработана сеть Raiden Network. Существует много других подобных разработок, позволяющих сделать микроплатежи в сетях с блокчейном более быстрыми и удобными.

## 2.7.2. Шардинг

Шардинг (sharding), или сегментирование, — это одна из технологий масштабирования, которая применялась веками и стала более востребованной в эпоху баз данных. Люди по-разному используют эту технологию для решения различных проблем масштабируемости. Прежде чем мы поймем, как можно использовать сегментирование для масштабирования блокчейна, давайте сначала разберемся, что это такое.

Чтение/запись на диск всегда было узким местом при работе с огромными наборами данных. Когда данные распределены по нескольким дискам, чтение/запись может выполняться параллельно, и задержка значительно уменьшается. Эта техника называется *шардингом базы данных*.

Обратите внимание, как выполняется разбиение для распределения таблицы базы данных объемом 300 Гб на три сегмента (шарда) по 100 Гб каждый, которые хранятся на отдельных экземплярах сервера (рис. 2.38). Такая концепция применима и для блокчейна, где общее состояние блокчейна делится на разные сегменты, которые содержат свои собственные подсостояния. Впрочем, это далеко не так просто, как разделить базу данных с помощью горизонтального рассечения.

Итак, как же шардинг работает в контексте блокчейна? Идея состоит в том, что узлам не нужно загружать и хранить копию всей цепочки блоков. Вместо этого они будут загружать и сохранять только нужные им части (шарды). Тем самым они об-

рабатывают только те транзакции, которые имеют отношение к хранимым данным, и появляется возможность параллельного выполнения транзакций. Таким образом, когда происходит транзакция, она направляется только на определенные узлы в зависимости от того, какие шарды они обслуживают. Если взглянуть на это с другой точки зрения, все узлы не обязаны выполнять все виды вычислений и проверок для каждой транзакции. Если для обработки каких-либо конкретных транзакций требуется более одного шарда, необходимо определить протокол и механизм обмена данными между шардами. Помните, что разные блокчейны могут иметь разные варианты шардинга.

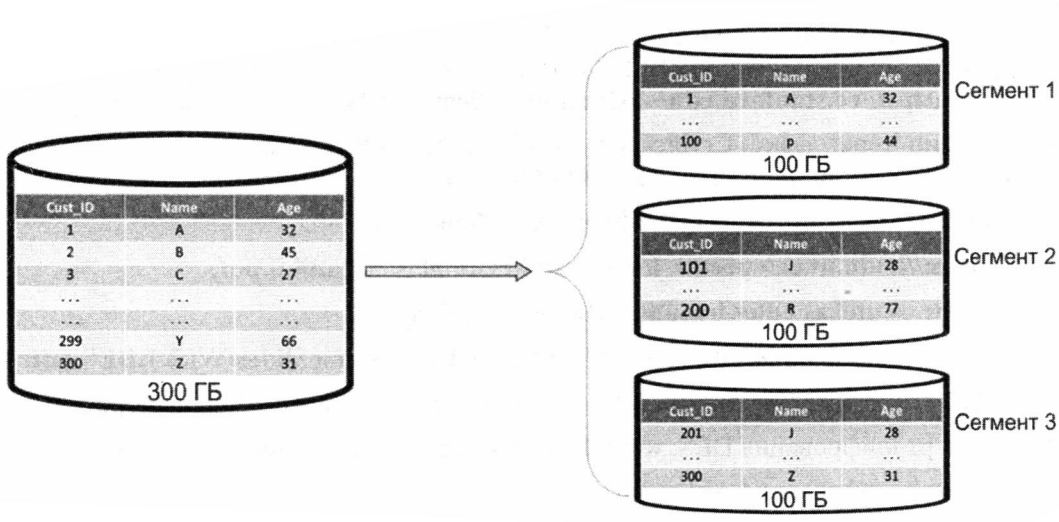



**Рис. 2.38. Пример шардинга базы данных**

Одним из примеров может служить ситуация, когда в шардах находится несколько уникальных учетных записей. Другими словами, каждая уникальная учетная запись находится в каком-то одном сегменте (более применимо для блокчейнов в стиле Ethereum с сохранением состояния), и для учетных записей внутри одного шарда очень легко вести обмен транзакциями между собой. Очевидно, что для работы шардинга требуется еще один координирующий уровень, находящийся над уровнем сегментов, поскольку узлы могут хранить только часть информации.

## 2.8. Заключение

В этой главе мы глубоко погрузились в фундаментальные основы криптографии, теории игр и компьютерных технологий. Изученные концепции помогут вам разработать собственное решение для блокчейна с учетом ваших особых потребностей. Блокчейн определенно не волшебная палочка для решения любых проблем. Однако там, где действительно нужен блокчейн, в разных ситуациях потребуются различные варианты механизма цепочки блоков.

Мы изучили различные криптографические методы для обеспечения безопасности транзакций и осознали всю полезность хэш-функций. Мы узнали, как теория игр

может быть использована для разработки надежных системных решений. Мы также изучили некоторые основные принципы информатики, такие как структура данных блокчейна и деревья Меркла. Некоторые из концепций были дополнены сниппетами кода, чтобы помочь вам быстрее начать работать с блокчейном.

В следующей главе мы узнаем больше о блокчейне Bitcoin и о том, как именно он работает.

## 2.9. Рекомендуемые источники

- Новые направления в криптографии: Diffie, Whitfield; Hellman, Martin E., «New Directions in Cryptography» IEEE Transactions on Information Theory, Vol IT-22, No 6, **https://ee.stanford.edu/~hellman/publications/24.pdf**, November, 1976.
- Принцип Керкгоффса: Crypto-IT Blog, «Kerckhoff's Principle» **www.crypto-it.net/eng/theory/kerckhoffs.html**.
- Блочный шифр, потоковый шифр и шифр Фейстеля:
  - **http://kodu.ut.ee/~peeter_l/teaching/kryptoi05s/streamkil.pdf**;
  - **www.cs.utexas.edu/~byoung/cs361/lecture45.pdf**;
  - **www.cs.man.ac.uk/~banach/COMP61411.Info/CourseSlides/Wk2.1.DES.pdf**;
  - **https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture3.pdf**.
- Стандарт шифрования DES: **www.facweb.iitkgp.ernet.in/~sourav/DES.pdf**.
- Стандарт шифрования AES:
  - **www.facweb.iitkgp.ernet.in/~sourav/AES.pdf**;
  - National Institute of Standards and Technology (NIST), «Announcing the Advanced Encryption Standard (AES)», Federal Information Processing Standards Publication 197, **http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf**, November 26, 2001.
- Стандарт SHS: National Institute of Standards and Technology (NIST), «Announcing the Advanced Encryption Standard (AES)», Federal Information Processing Standards Publication 197, **http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf**, November 26, 2001.
- Стандарт SHA-3:
  - NIST, «Announcing DraftFederl Information Processing Standard (FIPS) 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, and Draft Revision of the Applicability Clause of FIPS 180-4, Secure Hash Standard, and Request for Comments», **https://csrc.nist.gov/News/2014/Draft-FIPS-202, -SHA-3–Standard-and-Request-for-Com**, May 28, 2014.
  - Paar, Christof, Pelzl, Jan, «SHA-3 and the Hash Function Keccak», Understanding Cryptography—A Textbook for Students and Practitioners, (Springer, 2010), **https://pdfs.semanticscholar.org/8450/06456ff132a406444fa85aa7b5636266a8d0.pdf**.

- Алгоритм RSA:
  - Kaliski, Burt, «The Mathematics of the RSA Public-Key Cyptosystem», RSA Laboratories, **www.mathaware.org/mam/06/Kaliski.pdf**.
  - Milanov, Evgeny, «The RSA Algorithm», **https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf**, June 3, 2009.
- Теория игр: Pinkasovitch, Arthur, «Why Is Game Theory Useful in Business?», Investopedia, **www.investopedia.com/ask/answers/09/game-theory-business.asp**, December 19, 2017.
- Алгоритм PoS:
  - Buterin, Vitalik, «A Proof of Stake Design Philosophy», Medium, **https://medium.com/@VitalikButerin/a-proof-of-stake-design-philosophy-506585978d51**, December 30, 2016.
  - Ray, James, «Proof of Stake FAQ», Ethereum Wiki, **https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ**.
- Внедрение блокчейн-инноваций: Back, Adam, Corallo, Matt, Dash Jr, Luke, et al., «Enabling blockchain Innovations with Pegged Sidechains», **https://blockstream.com/sidechains.pdf**.

ГЛАВА 3

# Как работает Bitcoin?

Блокчейн вошел в моду благодаря технологии Bitcoin. Блокчейн, как мы знаем, это подарок изобретателя Bitcoin Сатоши Накамото всему миру. Доподлинно неизвестно, кто такой Сатоши Накамото, — это имя используется неизвестным лицом или группой лиц, которые создали Bitcoin. Мы предлагаем вам понять и оценить по достоинству замечательную новую технологию, не разыскивая изобретателя. Изучение технических основ Bitcoin поможет вам лучше понять другие варианты блокчейна, которые представлены на рынке.

Поскольку многолетнее применение технологии Bitcoin свидетельствует о надежности технологии блокчейна в целом, люди поверили в блокчейн и начинают искать другие способы его применения. В предыдущей главе мы уже познакомились с тем, как работает блокчейн в общем виде, но изучение устройства Bitcoin даст вам понимание блокчейна с прикладной точки зрения. Возможно, вы захотите изучить Bitcoin как пример использования криптовалюты, основанной на технологии блокчейн. Эта глава даст вам представление о том, как на основе блокчейна могут быть реализованы нужные именно вам модификации технологии Bitcoin.

В этой главе мы подробно рассмотрим Bitcoin, при этом многие основы блокчейна будут повторно разъяснены с прикладной точки зрения. Если вы уже знакомы с основами Bitcoin, вы можете пропустить эту главу. В противном случае мы советуем вам внимательно изучить все разделы по порядку. В этой главе объясняется, что такое Bitcoin, как он устроен, а также анализируются некоторые сильные и слабые его стороны.

## 3.1. История денег

Вы никогда не задумывались, что такое деньги и почему они вообще существуют? Деньги — это, прежде всего, средство обмена произвольными ценностями. Мы бегло окинем взглядом историю денег, чтобы понять, как деньги превратились в то,

чем они являются для нас сегодня, и как Bitcoin продвигает их на следующий уровень развития.

Не у всех есть все, что им нужно. В старые добрые времена, когда не было понятия о валюте или деньгах, люди старались обменять то, что у них было в избытке, на то, что им было нужно от кого-то другого. Это были дни *бартерной системы*. Обмен пшеницы на ботинки или апельсинов на лимоны был обычным делом. Это конечно хорошо, но что, если кому-то, имеющему пшеницу, нужно лекарство, которого нет у ближайшего партнера по обмену? Например, у Алисы есть пшеница и ей нужны лекарства, а Бобу нужна пшеница, но у него есть апельсины. В этой ситуации простой бартер не работает. Поэтому Алиса и Боб должны найти третьего человека, Чарли, у которого есть лекарство, и при этом ему нужны апельсины. Наглядное представление этого сценария обмена изображено на рис. 3.1.

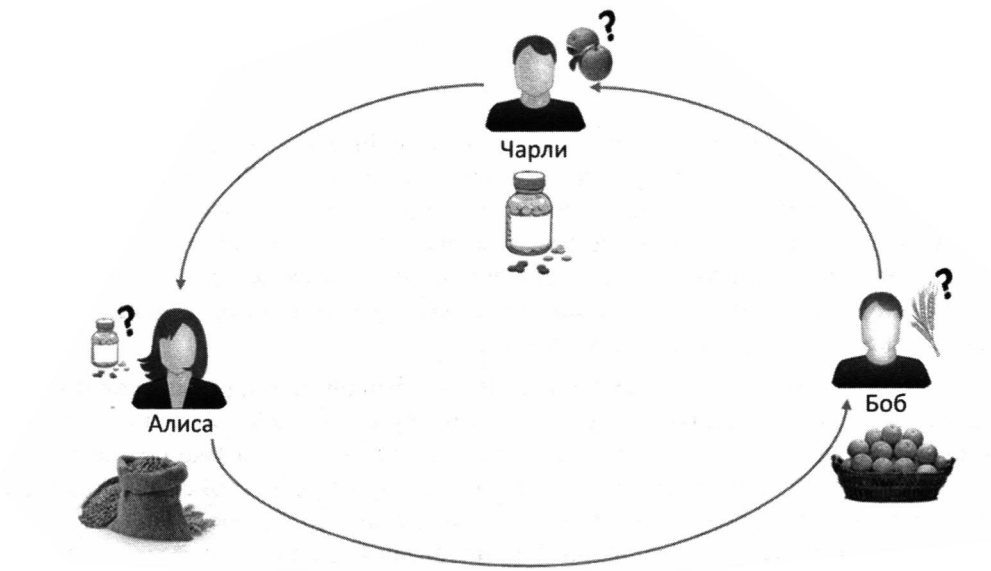



**Рис. 3.1. Примитивная бартерная система**

Но на самом деле очень трудно найти такого человека, как Чарли, который бы идеально вписался в схему обмена. Поэтому люди начали думать о коммодитизированной системе[1] обмена ценностями. В те времена было несколько товаров, которые нужны всем, — например, молоко, соль, семена, овцы и т. д. Эта система неплохо работала! Но через какое-то время люди поняли, что хранить такие товары весьма неудобно и сложно.

В конце концов, люди придумали более удобные расчетные инструменты, такие как кусочки металла. Редкие металлы ценились больше, чем обычные. Золото и серебро возглавляли список металлов, поскольку они не подвержены коррозии. Затем

[1] Система, основанная на товарах массового спроса (commodities), которые нужны абсолютному большинству участников рынка.

страны начали чеканить свою собственную валюту (металлические монеты разного веса) с официальной печатью. Металлические слитки и монеты прекрасно подходят для взаиморасчетов, т. к. их можно легко хранить и переносить. Но есть и обратная сторона удобства — монеты уязвимы для кражи. В этой ситуации на помощь пришли храмы, потому что люди доверяли церкви и твердо верили, что никто не будет воровать из храма. Священники давали квитанцию лицу, вносящему металлические деньги, в которой указывалось количество полученного золота/серебра. Квитанция служила обещанием подтвердить депозит по запросу третьей стороны или вернуть взнос владельцу квитанции по первому требованию. Человек, имеющий квитанцию, может пустить эту бумагу в оборот на рынке, чтобы получить то, что он хотел. Это был зародыш современной банковской системы. Квитанция успешно работала вместо металлических денег, поскольку храмы, хранившие фиатную валюту, играли роль централизованных банков, которым доверяли люди. Взгляните на рис. 3.2, чтобы понять, как в те годы выглядели взаиморасчеты.


**Рис. 3.2. Начало эры банков**

В только что описанной системе бумажная валюта всегда поддерживалась каким-то драгоценным металлом — таким как золото или серебро. Эта система действовала даже после того, как на место храмов пришли правительства и банки. Какая бы валюта ни была в те дни, она была подкреплена золотом или серебром.

Постепенно бумажная валюта получила статус законного платежного средства, которое больше не поддерживалось золотом или серебром. Этот статус основывался исключительно на доверии — в том смысле, что у людей не было выбора, кроме как доверять правительству. Иными словами, бумажные деньги не имеют собственно ценности, их просто поддерживает правительство. Сегодня все деньги, кото-

рые мы используем во взаиморасчетах, — это бумажные валюты. Таким образом, ценность денег сегодня зависит от стабильности и эффективности правительств, в юрисдикции которых валюта выпускается и используется. По сути, многие банки не располагают ничем более ценным, чем записи о движении бумажных валют. Такое состояние банковской системы сохранялось продолжительное время, а цифровой мир только начинал формироваться.

В 1990-е годы Интернет быстро набрал обороты, а банковские системы начали оцифровываться. При пользовании бумажными деньгами все еще сохраняется некоторый уровень дискомфорта — купюры быстро изнашиваются и уязвимы для кражи ничуть не меньше металлических монет. Это побудило банки перейти к безналичным расчетам, когда бумажные купюры не нужно даже печатать. Деньги стали всего лишь числами в компьютерных системах банков. Сегодня, если каждый владелец счета пойдет в свой любимый банк и потребует выдать купюрами все деньги, которые держит на своих счетах, у банков начнутся большие проблемы! Общее количество «физических» денег в обращении у людей крайне незначительно по сравнению с количеством цифровых денег в мировом обороте.

## 3.2. Появление биткойна

В *главе 1* мы рассмотрели технологические аспекты интернет-революции, а в предыдущем разделе этой главы говорили об основных этапах эволюции денег. Теперь мы должны рассмотреть их в совокупности, чтобы понять точку зрения Сатоши Накамото, стоящего за разработкой биткойна — самой популярной криптовалюты. В этом и в других разделах этой главы мы попытаемся более подробно остановиться на идеях Сатоши, изложенных в статье, которую он написал о биткойне.

Мы узнали о том, какую роль играли в денежных системах храмы, а затем правительства и банки. Даже сегодня ситуация не изменилась. Единственная ключевая вещь, которая делает традиционные платежные системы стабильными, — это доверие. Люди сначала доверяли храмам, а потом — правительствам и банкам. Сегодня вся коммерция в Интернете опирается на централизованные, доверенные третьи стороны, предназначенные для обработки платежей. Хотя Интернет был разработан, чтобы воплотить всеобщее равенство, люди продолжают строить в нем централизованные системы. Что ж, справедливости ради отметим, что еще в начале 2000-х годов было весьма сложно построить одноранговую систему, учитывая незрелые технологии того времени. Высокая стоимость транзакций, время, необходимое для совершения транзакций, и другие проблемы, связанные с централизацией, оставались нерешенными. Сейчас мы не говорим о физических валютах, т. к. транзакции означают сведение взаимного баланса между банками.

Может ли сегодня существовать цифровая валюта, обеспеченная вычислительной мощью, так же, как в свое время золото обеспечивало бумажные деньги? Благодаря биткойнам Сатоши ответ — «да». Биткойны предназначены для обеспечения электронных платежей между двумя сторонами на основе криптографического доказа-

тельства, а не на основе доверия к посреднику в виде третьей стороны сделки. Сегодня это стало возможно благодаря новым технологическим достижениям. В этой главе мы увидим, как в 2008 году Сатоши Накамото объединил основы криптографии, теории игр и информатики в системе обмена ценностями под названием Bitcoin. Начиная с фактического запуска в 2009 году и до сегодняшнего дня, система достаточно стабильна и устойчива, чтобы выдержать различные виды кибератак. Платежная система выдержала испытание временем, а биткойн утвердился в качестве мировой криптовалюты.

## 3.2.1. Что такое биткойн?

Блокчейн служит основой для криптовалюты — цифровых денег. Подобно тому, как мы можем осуществлять операции с физической валютой без банков или других централизованных организаций, биткойн предназначен для упрощения одноранговых денежных транзакций без доверенных посредников. Биткойн — это децентрализованная глобальная криптовалюта, которая не ограничивается какой-либо страной. Расчетная система биткойна децентрализована во всех аспектах — технических, логических и политических. Как только проверены новые транзакции, в оплату за работу проверяющих выпускаются новые биткойны. Всего в системе может быть выпущен 21 миллион биткойнов, и выпуск этой суммы будет завершен приблизительно в 2140 году. Любой, кто обладает хорошими вычислительными возможностями, может участвовать в майнинге и генерировать новые биткойны. После того, как все биткойны будут сгенерированы, выпуск новых криптомонет прекратится, и будут использоваться только те, которые находятся в обращении. Обратите внимание, что биткойны не имеют фиксированных номиналов, таких как национальные банкноты или металлические монеты. В соответствии с устройством системы, биткойны могут иметь любое значение с точностью до восьми знаков после запятой. Наименьшее значение в биткойнах составляет 0,00000001 BTC, которое называется *1 сатоши*.

Майнеры обрабатывают транзакции для генерации новых монет, а также получают комиссию, которую готов заплатить человек, желающий совершить транзакцию. В будущем, когда общее количество монет достигнет 21-го миллиона, майнеры будут проверять транзакции исключительно за комиссионные с транзакций. Но сегодня, если кто-то попытается совершить корректную транзакцию без уплаты комиссии, она все равно рано или поздно будет обработана, потому что майнер больше заинтересован в вознаграждении за майнинг, который позволяет ему генерировать новые монеты.

Вам интересно, что определяет ценность биткойна? Когда валюта была обеспечена золотом, на основе золотых стандартов было легко оценить ее стоимость. Когда мы говорим, что биткойн обеспечен вычислительными мощностями, которые люди используют для майнинга, этого недостаточно, чтобы понять, откуда возникает его ценность. Вот вам немного экономики, необходимой для понимания ценности биткойна.

Когда бумажная валюта была запущена впервые, она была обеспечена золотом. Поскольку люди верили в золото, они также верили в валюту. Через некоторое время валюта больше не была обеспечена золотом и полностью зависела от правительств. Люди продолжали верить в ценность валюты, потому что они сами участвовали в формировании собственного правительства. Поскольку правительства гарантируют платежеспособность валют, а люди доверяют правительствам, валюты приобретают свою ценность. В международном контексте стоимость валюты конкретных стран зависит от различных факторов, и наиболее важным из них является соотношение спроса и предложения. Отметим, что некоторые страны, которые выпустили в обращение большое количество денег, обанкротились — их экономика рухнула! Должен существовать точно выверенный баланс, но чтобы разобраться в монетарном вопросе, нужно глубоко изучить экономику, а это выходит за рамки нашей книги. Итак, давайте сейчас вернемся к биткойну.

Когда биткойн был впервые выпущен в оборот, у него не было официальной цены или общественной ценности, в которую люди могли бы поверить. Если бы в то время биткойн продавали за несколько долларов США, я бы ни за что его не купил! Постепенно, с началом биржевой торговли, у биткойна появилась своя цена, но один биткойн тогда не стоил даже одного доллара США. Поскольку биткойны генерируются конкурентным и децентрализованным процессом, называемым *майнингом*, и генерируются они с фиксированной скоростью с ограниченной суммой в 21 миллион биткойнов, с точки зрения экономики это делает биткойн дефицитным ресурсом. Свяжите технический контекст с биржевой игрой спроса и предложения, и вы поймете, почему ценность биткойна начала расти. Постепенно, когда все больше людей по всей планете стало верить в биткойн, его цена взлетела с нескольких центов до тысяч долларов. Востребованность биткойнов среди пользователей, продавцов, стартапов и крупных предприятий непрерывно растет, потому что биткойны используются ими в качестве платежного средства. Таким образом, на стоимость биткойна сильно влияют доверие, востребованность и соотношение спроса и предложения, а его цена определяется рынком.

Другой вопрос заключается в том, почему на момент подготовки книги стоимость биткойна так нестабильна. Одной из очевидных причин является колебание спроса и предложения. Мы узнали, что в обращении может быть только ограниченное количество биткойнов, а именно 21 миллион, и скорость их получения со временем уменьшается. Из-за такого устройства системы всегда существует разрыв между спросом и предложением, что приводит к подобной волатильности. Другая причина в том, что биткойны никогда не торгуются в одном месте. Есть много независимых бирж в разных местах по всему миру, и все эти биржи имеют свои собственные биржевые цены. Индексы котировок, которые вы видите, собирают биржевые цены биткойнов с нескольких бирж, а затем усредняют их. Опять же, поскольку все эти индексы не собирают данные из одного и того же набора источников, даже они не совпадают. Точно так же на волатильность цены биткойна влияет *фактор ликвидности*, который подразумевает количество биткойнов, проходящих через весь рынок в любой момент времени. На данный момент это однозначно высокорисковый актив, но со временем он может стабилизироваться. Давайте взглянем на следую-

щий список факторов, которые могут повлиять на спрос и предложение биткойна, а следовательно, и на его цену:

- уверенность людей в биткойнах или, наоборот, боязливая неуверенность;
- освещение в прессе хороших или плохих новостей о биткойнах;
- некоторые люди держат биткойны и не позволяют им проходить через рынок, а некоторые продолжают покупать и продавать, чтобы минимизировать риск. Вот почему уровень ликвидности биткойна продолжает меняться;
- принятие биткойнов гигантами электронной коммерции;
- запрет биткойнов в определенных странах.

Если вы хотите спросить, существует ли вероятность того, что биткойн потерпит крах, то мы ответим «да». Есть много примеров стран, чьи валютные системы потерпели крах. Ну да, у них были политические и экономические причины, чтобы они рухнули, — например, гиперинфляция, которая неприменима к биткойну, потому что нельзя генерировать столько биткойнов, сколько вздумается, и общее количество биткойнов ограничено. Тем не менее существует вероятность технического или криптографического сбоя биткойнов. Хотя биткойн выдержал испытание временем с момента своего основания в 2008 году, абсолютную надежность биткойна гарантировать невозможно!

## 3.2.2. Работа с биткойнами

Для того чтобы начать работу с биткойнами, не требуется никакой технической подготовки. Вам просто нужно скачать биткойн-кошелек и приступить к работе с ним. Когда вы загружаете и устанавливаете кошелек на свой ноутбук или смартфон, он генерирует ваш первый биткойн-адрес (открытый ключ). Но вы не только можете сгенерировать больше одного адреса — вы *должны* это сделать. Рекомендуется использовать адреса в сети Bitcoin только один раз. Повторное использование адресов является нежелательной практикой, хотя и работает. Повторное использование может нанести ущерб приватности и конфиденциальности. Например, если вы повторно используете один и тот же адрес (открытый ключ) и подписываете транзакции одним и тем же закрытым ключом, получатель может легко и надежно установить, что повторно используемый биткойн-адрес принадлежит вам. Если один и тот же адрес используется в нескольких транзакциях, все они могут быть отслежены, и будет намного проще определить, кто вы. Помните, что Bitcoin не является полностью анонимным, — скорее его можно назвать псевдонимом, и есть способы отследить происхождение транзакции, которое приведет к владельцу.

Вы должны сообщить свой биткойн-адрес лицу, желающему перевести вам биткойны. Это вполне безопасно, потому что открытый ключ в любом случае является публично доступным. Мы знаем, что в системе Bitcoin нет понятия о конечном сальдо, и все записи представлены в виде транзакций. Биткойн-кошельки могут легко рассчитать свой доступный остаток, поскольку у них есть закрытые ключи для соответствующих открытых ключей, по которым принимаются транзакции.

Существует множество биткойн-кошельков от разных разработчиков. Вам доступны мобильные кошельки, кошельки для настольных компьютеров, веб-кошельки на основе браузера, аппаратные кошельки и их комбинации с различными уровнями безопасности. Вы должны быть предельно осторожны относительно безопасности кошелька при работе с биткойнами. Платежи в биткойнах необратимы!

Вам интересно, насколько надежно защищены эти кошельки? Разные типы кошельков имеют разные уровни безопасности, и техническое решение зависит от того, как вы хотите использовать кошелек. Многие сервисы онлайн-кошельков пострадали от уязвимостей в системе безопасности. Всегда полезно включать двухфакторную аутентификацию, когда это возможно. Если вы являетесь постоянным пользователем системы Bitcoin, наверное, было бы неплохо держать небольшие суммы в разных кошельках и хранить их отдельно в безопасной среде. Наиболее высокий уровень безопасности обеспечивает автономный или «холодный» кошелек, который не подключен к сети. Кроме того, должны быть надежные механизмы резервного копирования для вашего кошелька на случай, если вы потеряете свой компьютер или смартфон. Помните, что если вы утратите свой закрытый ключ, то потеряете все деньги, связанные с ним!

Если вы еще не подключились к Bitcoin в качестве майнера с полным узлом, то вы можете просто быть пользователем или трейдером биткойнов. Вам определенно понадобится биржа валют, где вы сможете купить биткойны за доллары США или другую валюту. Будьте благоразумны и пользуйтесь услугами законных и безопасных обменных бирж. Известно много примеров, когда пользователи пострадали от собственной легкомысленности или уязвимостей системы безопасности при обмене.

## 3.3. Блокчейн Bitcoin

В предыдущей главе мы уже рассмотрели структуру данных блокчейна, а также обсудили основные составные части блокчейна, такие как хэширование и асимметричная криптография. В этом разделе мы изучим специфику блокчейна Bitcoin.

Структура данных блокчейна Bitcoin похожа на структуру данных любого другого блокчейна. Клиент Bitcoin Core использует базу данных LevelDB от Google для хранения данных внутри цепочки блоков. Каждый блок идентифицируется своим хэшем (Bitcoin использует алгоритм хэширования SHA-256). Каждый блок в своем разделе заголовка содержит хэш предыдущего блока. Помните, что этот хэш охватывает не только предыдущий заголовок, но и весь предыдущий блок, и так продолжается вплоть до генезисного блока. Генезисный блок — это начало любого блокчейна. Как правило, блокчейн в формате Bitcoin выглядит так, как показано на рис. 3.3.

Как видите, в блоке есть сегмент заголовка, который содержит информацию заголовка блока, и есть тело блока, где хранятся транзакции этого блока. Заголовок каждого блока содержит хэш предыдущего блока, взятого целиком с заголовком. Таким образом, невозможно с легкостью изменить содержимое произвольного бло-

ка — придется соответствующим образом изменить информацию всех последующих блоков. Допустим, кто-то пытается изменить предыдущую транзакцию, которая была сохранена, скажем, в блоке с номером 441. После изменения транзакции новый хэш этого блока не будет совпадать со старым хэшем, который находится в заголовке блока с номером 442, поэтому вам придется изменить также и блок с номером 442. Но теперь хэш блока 442 перестал совпадать с хэшем, который хранится в заголовке блока 443. Следовательно, вам придется исправить все блоки, вплоть до текущего блока. Это почти невозможно, поскольку у каждого узла есть своя собственная копия блокчейна, а взлом всех узлов (или, по крайней мере, 51% из них) практически невозможен.

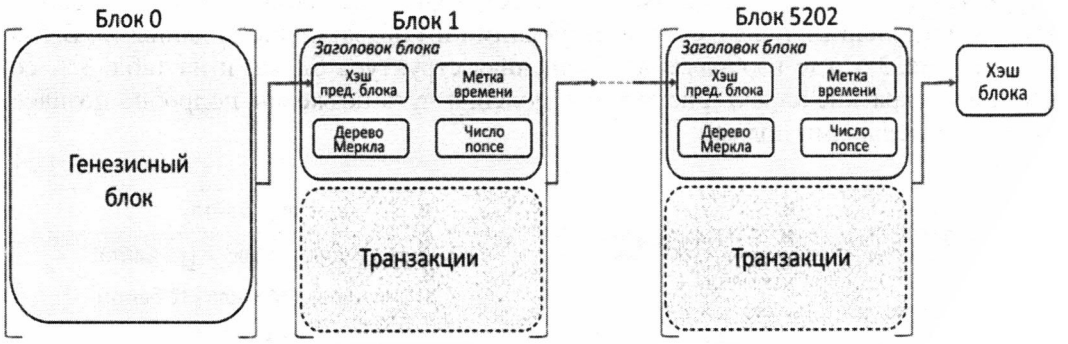



**Рис. 3.3. Блокчейн в формате Bitcoin**

В блокчейне есть только один верный обратный путь к блоку генезиса — когда вы движетесь назад от текущего блока. Однако если вы движетесь вперед, т. е. начинаете с блока генезиса, то вы можете столкнуться с развилками. Когда майнеры одновременно предлагают два блока, и оба они действительны, только один из них станет частью истинной цепочки, а другой окажется *осиротевшим блоком* (orphaned block). Каждый узел строится на основании самой длинной цепочки, поэтому он будет принимать к синхронизации самую длинную цепочку из тех, что «услышал» в сети. Подобный сценарий изображен на рис. 3.4.

Обратите внимание: на рис. 3.4 видно, что на уровне третьего звена цепочки два блока претендуют на то, чтобы стать блоком номер три, но только один из них

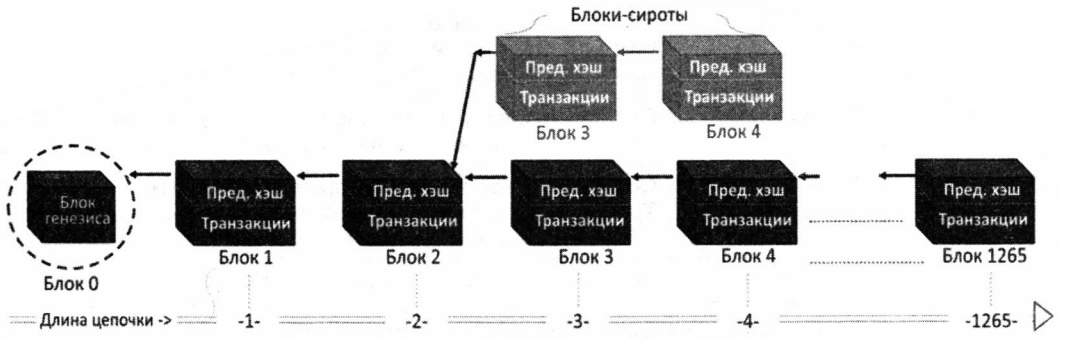



**Рис. 3.4. Осиротевшие блоки в блокчейне**

может попасть в конечный блокчейн, а другой останется сиротой. Очевидно, что в определенных ситуациях на узле может храниться версия блокчейна, где к цепочке прикреплены блоки, которые отвергнуты остальной сетью. При следующей синхронизации версия цепочки будет обновлена, а эти блоки станут сиротами. Поэтому не следует для идентификации блока использовать его порядковый номер в цепочке — если блок сгенерирован относительно недавно, он может стать сиротой. Используйте для однозначной идентификации блока его хэш.

## 3.3.1. Структура блока

Структура блока в блокчейне Bitcoin является фиксированной для всех блоков и имеет определенные поля с соответствующими им обязательными данными. Взгляните на рис. 3.5, где изображена обобщенная структура блока, и на табл. 3.1, содержащую краткие характеристики его полей, а чуть позже мы подробно познакомимся с отдельными полями.

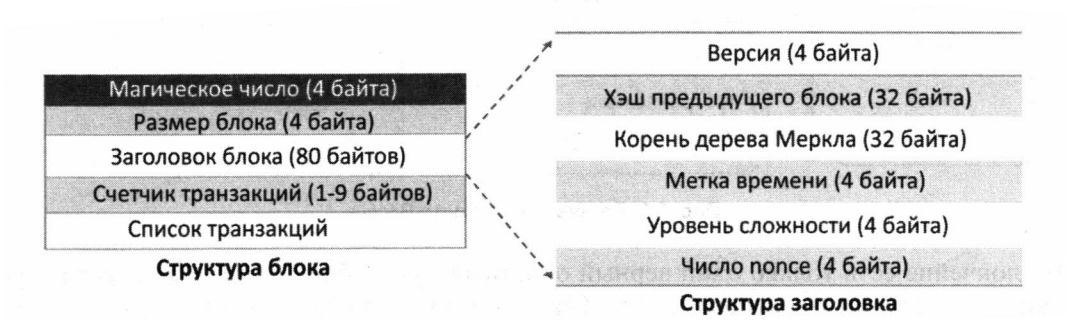



**Рис. 3.5.** Структура блока в блокчейне Bitcoin

***Таблица 3.1.*** *Структура блока*

| Поле | Размер | Описание |
|---|---|---|
| Магическое число | 4 байта | Имеет фиксированное значение `0xD9B4BEF9`, которое означает, что блок принадлежит к блокчейну Bitcoin, а также расположен в основной (рабочей, а не отладочной) цепочке |
| Размер блока | 4 байта | Обозначает размер блока. Изначально блок Bitcoin имел размер 1 Мбайт, но в новой версии под названием Bitcoin Cash блоки имеют размер 2 Мбайт |
| Заголовок блока | 80 байтов | Составное поле, которое содержит много информации: хэш предыдущего блока, число nonce и другие данные |
| Счетчик транзакций | 1–9 байтов (переменная длина) | Содержит общее количество транзакций в блоке. Транзакции имеют разный размер, количество транзакций в блоке тоже бывает разным |
| Список транзакций | Переменное количество, но постоянный размер | Перечень транзакций, входящих в данный блок. В зависимости от размера блока (1 или 2 Мбайт), это поле всегда заполняет оставшееся пространство блока |

Теперь давайте более детально рассмотрим поле **Заголовок блока** и изучим различные элементы, из которых состоит заголовок (табл. 3.2).

***Таблица 3.2.** Компоненты заголовка блока*

| Поле | Размер | Описание |
|---|---|---|
| Версия | 4 байта | Версия протокола Bitcoin. В идеале каждый узел сети Bitcoin должен использовать одинаковую версию протокола |
| Хэш предыдущего блока | 32 байта | Содержит хэш заголовка предыдущего блока цепочки. Когда все поля заголовка предыдущего блока хэшированы по алгоритму SHA-256, на выходе мы получаем 256-битовый результат, т. е. 32 байта |
| Корень дерева Меркла | 32 байта | Все транзакции блока хэшируются по принципу дерева Меркла, а корень Меркла — это самый верхний хэш в дереве. Если одна из транзакций блока модифицирована, то вычисленное значение корня Меркла не будет совпадать с записанным в заголовке. Этот подход гарантирует, что хэширования заголовков блоков достаточно, чтобы построить защищенный блокчейн. Кроме того, дерево Меркла позволяет определить, является ли транзакция частью блока всего за $\log_2(n)$ вычислений, что весьма быстро (см. *разд. 2.4.2*) |
| Метка времени | 4 байта | В сети Bitcoin не предусмотрено понятие глобального времени. Это поле содержит приблизительное время создания блока, записанное в формате UNIX |
| Уровень сложности | 4 байта | Уровень сложности в алгоритме консенсуса с доказательством работы (PoW) на момент майнинга этого блока |
| Число nonce | 4 байта | Случайное число, которое найдено в процессе майнинга и удовлетворяет условию головоломки PoW |

Для большинства полей вполне достаточно пояснений, приведенных в табл. 3.1 и 3.2, а далее мы рассмотрим несколько полей, для которых нужны более подробные пояснения.

## Дерево Меркла

Мы рассмотрели понятие деревьев Меркла в предыдущей главе. В этом разделе мы просто обсудим, как Bitcoin использует деревья Меркла. Каждый блок в цепочке Bitcoin содержит хэш всех транзакций блока, а корень Меркла этих транзакций включен в заголовок блока. На самом деле, когда мы говорим, что каждый заголовок блока содержит хэш всего предыдущего блока, смысл заключается в том, что он просто содержит хэш заголовка предыдущего блока. Тем не менее этого достаточно, потому что заголовок уже содержит корень Меркла. Если транзакция в блоке изменяется, корень Меркла больше не будет совпадать с заголовком, и такая конструкция обеспечивает целостность цепочки блоков.

Дерево Меркла — это древовидная структура хэшей транзакций. «Листовые узлы» в дереве Меркла фактически представляют собой хэши транзакций, тогда как корнем дерева является корень Меркла.

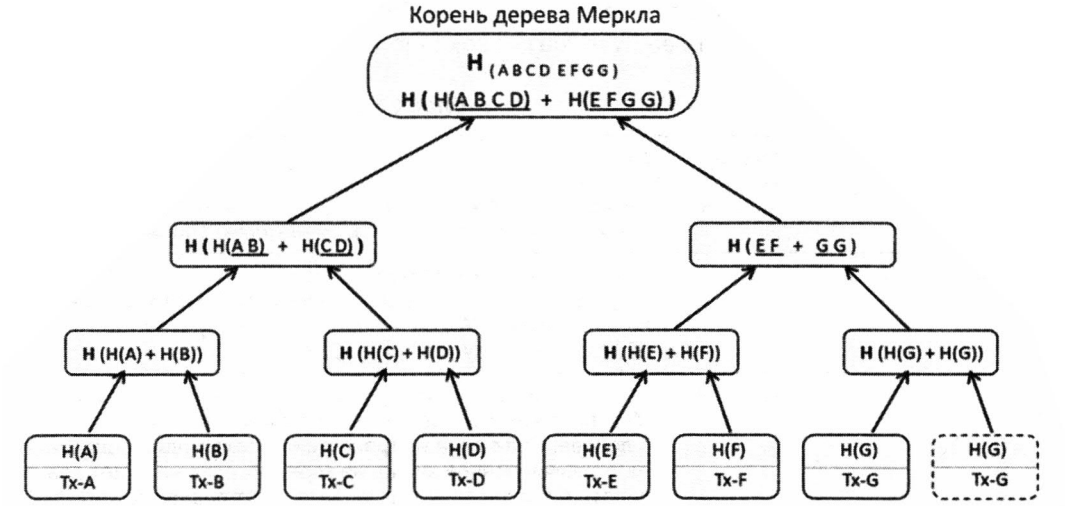



**Рис. 3.6. Представление дерева Меркла**

Посмотрите на рис. 3.6. Хэши семи транзакций: A, B, C, D, E, F и G образуют листья дерева. Общее количество конечных узлов в двоичном дереве должно быть четным, поэтому последний лист дублируется. Хэш каждой транзакции в 32 байта (т. е. 256 битов) рассчитывается путем двукратного применения SHA-256 к транзакциям. Аналогично, хэш двух транзакций объединяется, а затем дважды хэшируется с помощью SHA-256, чтобы получить родительский хэш из 32 байтов.

Чтобы проверить, была ли транзакция частью какого-либо блока, достаточно знать только путь Меркла к транзакции, поэтому описанный механизм весьма эффективен. Таким образом, блокчейн фактически может быть представлен, как показано на рис. 3.7.

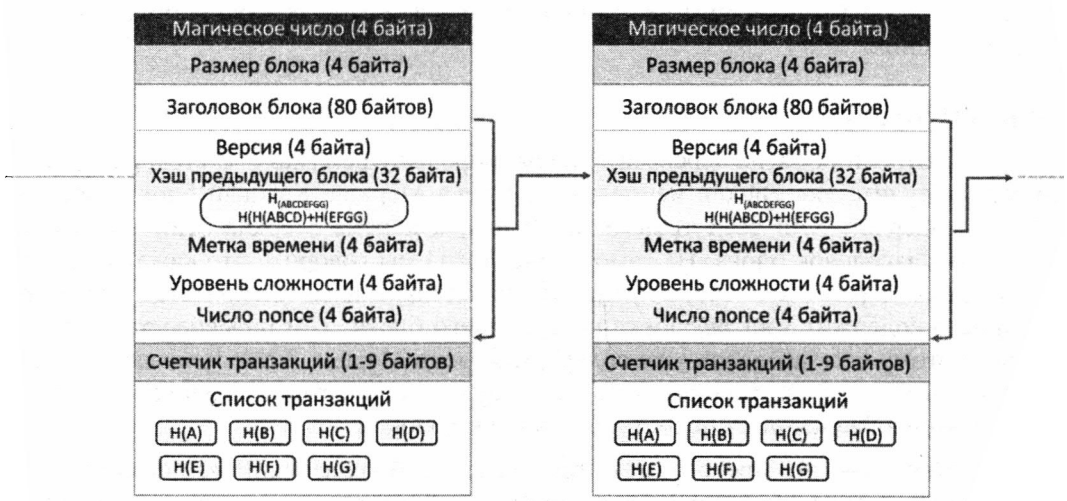



**Рис. 3.7. Фактическая структура блокчейна**

## Уровень сложности

Уровень сложности — это ключевой параметр, от которого зависит работа алгоритма PoW в Bitcoin. Идея состоит в том, что после заполнения блока корректными транзакциями хэш заголовка этого блока должен быть рассчитан так, чтобы он был меньше значения уровня сложности в том же заголовке. Одноразовое число nonce в заголовке начинается с нуля. Майнер должен продолжать увеличивать значение nonce и заново хэшировать заголовок, пока значение хэша не станет меньше заданного значения сложности.

От четырех байтов (32 бита) сложности в заголовке зависит, каким будет целевое значение сложности (256 битов) для этого блока. В процессе майнинга нужно найти такое число nonce, чтобы хэш всего заголовка был меньше заданного целевого значения. Помните, что чем ниже целевое значение, тем сложнее будет подобрать хэш заголовка, который будет меньше целевого значения. Поскольку Bitcoin использует SHA-256, каждый раз, когда вы хэшируете заголовок блока, на выходе получается непредсказуемое число в диапазоне от 0 до $2^{256}$. Если при выбранном вами числе nonce значение хэша заголовка оказывается меньше целевого значения, блок будет принят всей сетью, иначе вам придется пробовать другие числа nonce до тех пор, пока не будет выполнено условие головоломки. Однако на данный момент все еще не ясно, как из четырех байтов сложности в заголовке вычисляется целевое значение сложности.

Целевое значение может быть получено из четырех байтов сложности в заголовке при помощи определенной формулы, которая известна всем узлам, потому что поставляется в пакете установки программного обеспечения. Формула для расчета целевого значения в сети Bitcoin выглядит следующим образом:

$$\texttt{target} = \texttt{coefficient} \times 2^{(8 \times (\texttt{exponent} - 3))}$$

Обратите внимание, что в этой формуле есть переменные `coefficient` и `exponent`, которые формируются из байтов сложности заголовка. Давайте рассмотрим пример.

Если четыре байта сложности в поле заголовка в шестнадцатеричной форме имеют значение `0x1B0404CB`, то первые две шестнадцатеричные цифры образуют степенной член формулы, который равен `0x1B`, а оставшиеся образуют коэффициент, который в данном случае равен `0x0404CB`. Подставим эти числа в формулу расчета целевого значения `target`:

$$\texttt{target} = \texttt{0x0404CB} \times 2^{(\texttt{0x08} \times (\texttt{0x1B} - \texttt{0x03}))}$$

$$\texttt{target} = \texttt{0x0000000000404CB000000000000000000000000000000000000000000000000}$$

Bitcoin спроектирован таким образом, что каждые 2016 блоков должны генерироваться в течение двух недель, и не трудно подсчитать, что на каждый блок приходится около 10 минут. В асинхронной сети с механизмом PoW трудно сделать так, чтобы генерация каждого блока занимала ровно десять минут. На самом деле, это среднее время для блока, и есть вероятность, что блок будет сгенерирован в течение минуты, или, наоборот, генерация займет 15 минут. Таким образом, изменяемая сложность предназначена для увеличения или уменьшения *средней* продолжитель-

ности генерации блока в зависимости от того, потребовалось ли меньше или больше двух недель, чтобы найти 2016 блоков. Время, необходимое для генерации 2016 блоков, можно найти, используя значение метки времени в заголовке каждого блока. Если, допустим, на 2016 блоков потребовалось количество времени `T`, которое никогда не составляет ровно две недели, цель сложности в каждом блоке умножается на `(T / 2 недели)`. Таким образом, результат `[target × (T / 2 недели)]` увеличится, если `T` был меньше двух недель, и уменьшится в противном случае.

Теперь вы знаете, что уровень сложности можно регулировать: головоломка может быть сложнее или проще в зависимости от ситуации. У вас может возникнуть вопрос: кто регулирует сложность, когда система децентрализована? Главное правило, которое вы должны всегда помнить, заключается в том, что все, что происходит в децентрализованном проекте, должно происходить индивидуально на каждом узле. После каждых 2016 блоков все узлы по отдельности вычисляют новое целевое значение сложности, и у всех получается один и тот же результат, потому что для него определена общая формула. Еще раз запишем эту формулу в общем виде:

```
NewTarget= OldTarget × (T / 2 недели)
⇨ NewTarget= OldTarget × (Время для генерации 2016 блоков, сек / 1209600 сек)
```

**ПРИМЕЧАНИЕ.** Такие параметры, как 2016 блоков и `TargetTimespan`, равный двум неделям (1 209 600 секунд), определены в файле *chainparams.cpp*, как показано здесь:

```
consensus.nPowTargetTimespan = 14 * 24 * 60 * 60; // two weeks
consensus.nPowTargetSpacing = 10 * 60;
consensus.nMinerConfirmationWindow = 2016; //
nPowTargetTimespan / nPowTargetSpacing
```

Обратите внимание, что в формуле мы делим фактическое время на продолжительность двух недель (`T / 2 недели`), а не наоборот (`2 недели / T`). Идея состоит в том, чтобы уменьшить целевое значение, когда требуется увеличить сложность, чтобы генерация занимала больше времени.

> Чем меньше целевой хэш, тем сложнее найти хэш, который меньше этого целевого хэша.

Например, если для добычи 2016 блоков потребовалось десять дней, то вычисление (`T / 2 недели`) дает дробь меньше единицы, которая при умножении на `OldTarget` дает новое значение `NewTarget`, которое меньше предыдущего. Это затрудняет поиск хэша, и для генерации нового блока потребуется больше времени. В результате интервал времени между блоками в среднем составляет десять минут. Представьте, что уровень сложности оставили неизменным. Как вы думаете, какая возникнет проблема? Вспомните, что вычислительная мощность оборудования увеличивается со временем, поскольку для майнинга блоков используются все более мощные компьютеры. Ситуация, когда 10 или 100 или даже 1000 блоков предлагаются одновременно, вредна для нормальной работы сети. Таким образом, идея заключается в том, что даже когда все больше и больше мощных вычислительных узлов включаются в сеть Bitcoin, среднее время, необходимое для генерации блока, все равно должно

составлять десять минут за счет регулирования целевого уровня сложности. Кроме того, шансы майнера на успех зависят от того, сколько хэш-мощности он имеет по сравнению с глобальной хэш-мощностью всех действующих майнеров.

Вы спросите, почему именно десять минут? Почему не 12 минут? Или почему не шесть минут? Для согласования всех узлов децентрализованной асинхронной системы необходим определенный промежуток времени. Если бы не было этого промежутка, на узлы поступало бы слишком много блоков, разделенных мизерным интервалом, и не было бы никакой выгоды от использования *цепочки блоков* по сравнению с *цепочкой транзакций*. Каждая транзакция является трансляцией в сеть, и каждый новый блок также является трансляцией в сеть. Кроме того, упорядоченность, которой обладает система из цепочки блоков, совершенно невозможна в цепочке транзакций. Концепция блоков позволяет включать в блоки взаимно несвязанные транзакции от любого отправителя к любому получателю, что нелегко реализовать в цепочке транзакций. Передача одного проверенного блока, в который упаковано несколько транзакций, технически более эффективна по сравнению с широковещательной передачей отдельных проверенных транзакций. Теперь вернемся к обсуждению десяти минут. Действительно, этот интервал вполне может быть немного меньше или немного больше, но между двумя последовательными блоками обязательно должен быть некоторый разрыв. Представьте, что вы майнер, и майните блок 4567, но другому майнеру повезло, и он чуть раньше предложил блок 4567, который вы только что получили при решении криптографической головоломки. Теперь вам нужно проверить его блок, и, если он действителен, добавить его в свою локальную копию цепочки блоков и немедленно начать добычу блока 4568. Вам вовсе не хочется, чтобы кто-то еще предложил блок 4568, когда вы только что закончили проверку блока 4567, который получили чуть позже по сравнению с другими майнерами из-за задержки в сети. Теперь повторим вопрос: так почему же 10 минут — лучший вариант? Что ж, это сложно объяснить одним словом, но именно десятиминутный промежуток решает множество проблем из-за ограничений асинхронной сети, задержек, потерь пакетов, пропускной способности системы и многого другого. Иногда удается оптимизировать систему и уменьшить интервал, скажем, до пяти минут или около того, что вы можете видеть на примере многих новых криптовалют и других случаев использования блокчейна.

## 3.3.2. Блок генезиса

Самый первый блок цепочки называется *генезисом*, или *генезисным блоком*. Помните, что генезисный блок должен быть статично закодирован в приложениях блокчейна, как и в случае с Bitcoin. Вы можете рассматривать его как специальный блок, потому что он не содержит ссылки на предыдущие блоки. Генезис Bitcoin был создан при запуске в 2009 году. Если вы откроете ядро Bitcoin, в частности файл chainparams.cpp, вы увидите, что в нем статично закодирован генезисный блок. Используя командную строку Bitcoin Core, вы можете получить ту же информацию, введя хэш блока:

```
$ bitcoin-cli getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

Ответ на запрос выглядит следующим образом:

```
{
    "hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
    "confirmations" : 308321,
    "size" : 285,
    "height" : 0,
    "version" : 1,
    "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
    "tx" : ["4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"],
    "time" : 1231006505,
    "nonce" : 2083236893,
    "bits" : "1d00ffff",
    "difficulty" : 1.00000000,
    "nextblockhash" : "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"
}
```

Если вы конвертируете метку времени UNIX (поле `"time"`), показанную в предыдущем выводе, вы найдете информацию о дате и времени: суббота, 3 января 2009 г., 18:15:05. Вы также можете получить ту же информацию с сайта **https://blockchain.info**. Просто перейдите на этот сайт, вставьте хэш генезисного блока в поле поиска на главной панели и нажмите клавишу <Enter>. Вот что вы увидите на сайте (табл. 3.3 и 3.4).

***Таблица 3.3.** Информация о транзакциях генезисного блока*

| | |
|---|---|
| Количество транзакций | 1 |
| Всего выходов | 50 BTC |
| Предполагаемый объем транзакций | 0 BTC |
| Комиссия за транзакцию | 0 BTC |
| Высота | 0 (Главная цепочка) |
| Временная отметка | 2009-01-03 18:15:05 |
| Время получения | 2009-01-03 18:15:05 |
| Передано по | Unknown |
| Сложность | 1 |
| Биты | 486604799 |
| Размер | 0.285 kB |
| Вес | 0.896 kWU |
| Nonce (случайно перебираемое число) | 2083236893 |
| Награда за блок | 50 BTC |

*Таблица 3.4. Информация о хэшах*

| | |
|---|---|
| Хэш | 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f |
| Предыдущий блок | 0000000000000000000000000000000000000000000000000000000000000000 |
| Следующий(е) блок(и) | 00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048 |
| Корень Меркла | 4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b |

В этом нулевом блоке есть только одна транзакция, которая представляет собой *исходную монетарную транзакцию* (coinbase transaction). Майнерские транзакции являются именно такими монетарными транзакциями. Для таких транзакций нет входных данных, и они могут генерировать только новые биткойны. Если вас интересуют транзакции, связанные с этим блоком, вот как это будет выглядеть (рис. 3.8).

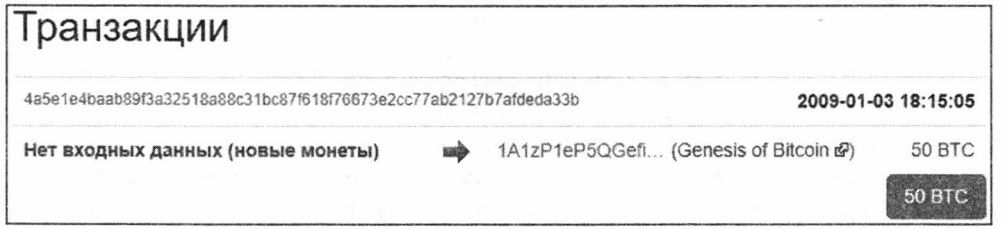



**Рис. 3.8. Исходная монетарная транзакция в генезисном блоке**

# 3.4. Сеть Bitcoin

Как мы уже говорили, сеть Bitcoin — это одноранговая сеть. В такой сети нет централизованного сервера, и каждый узел имеет одинаковые права. В подобной системе нет иерархии и разделения на роли «главный-подчиненный». Поскольку Bitcoin работает в Интернете, он использует стандартный стек протокола TCP/IP (рис. 3.9).

Схема на рис. 3.9 показывает, как различные узлы Bitcoin взаимодействуют через общий стек Интернета. Сеть Bitcoin весьма динамична в том смысле, что узлы могут присоединяться и выходить из сети по желанию, а система продолжает бесперебойно работать. Кроме того, несмотря на то, что она асинхронна по своей природе, с сетевыми задержками и потерями пакетов, система очень надежна, потому что так устроен протокол Bitcoin.

Сеть Bitcoin представляет собой децентрализованную сеть без центральной точки отказа, а также без центрального органа управления. Как при таком устройстве сети оценить, насколько велика сеть Bitcoin? Надежного способа оценки не существует, поскольку узлы могут присоединяться и отключаться в любое время. Тем не менее

предпринимаются попытки исследовать сеть Bitcoin, и некоторые исследователи утверждают, что в среднем почти 10 000 узлов подключены к сети все время[2], а в целом могут существовать миллионы узлов.

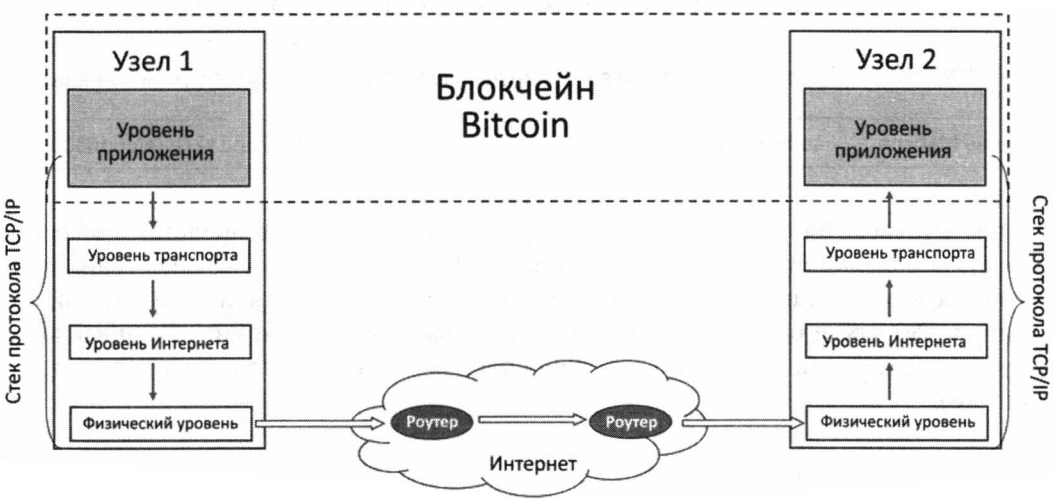



**Рис. 3.9. Сеть блокчейна Bitcoin в Интернете**

Каждый узел в сети Bitcoin одинаков с точки зрения полномочий, а сеть имеет плоскую структуру, но узлы могут быть полными узлами или легкими узлами. *Полные узлы* могут выполнять практически все допустимые действия в системе Bitcoin, такие как майнинг и трансляция транзакций, и могут предоставлять услуги кошелька. Полные узлы также предоставляют функцию маршрутизации для подключения к сети Bitcoin и технических нужд. Чтобы стать полным узлом, вы должны загрузить всю базу данных блокчейна, которая содержит все транзакции, имевшие место до сих пор. Кроме того, узел должен оставаться постоянно подключенным к сети Bitcoin и прослушивать все происходящие транзакции. Важно, чтобы у вас было надежное сетевое соединение, хорошее дисковое хранилище (не менее 200 Гбайт) и не менее 2 Гбайт ОЗУ. Эти требования могут возрасти со временем.

С другой стороны, *легкие узлы* не могут добывать новые блоки, но могут проверять транзакции с помощью *упрощенной проверки платежей* (Simplified Payment Verification, SPV). Их также называют *тонкими клиентами*, или *легкими клиентами*. Подавляющее большинство узлов в сети Bitcoin являются операторами SPV. Они также могут участвовать в формировании майнинг-пула, где собираются много узлов, пытающихся добывать новые блоки вместе. Легкие узлы могут помогать проверять транзакции для полных узлов. Хорошим примером SPV является пользовательский кошелек (клиент). Если у вас есть кошелек, и кто-то отправляет вам деньги, вы можете выступать в качестве узла в сети Bitcoin и скачать себе соответ-

[2] Статистику и текущее количество узлов можно посмотреть по адресу: **https://bitnodes.earn.com/dashboard/?days=365**.

ствующие транзакции, имеющие отношение к отправителю, чтобы проверить, действительно ли владелец, отправляющий вам биткойны, владел ими.

Важно отметить, что SPV не так защищен, как полный проверяющий узел, потому что он обычно содержит только заголовки блоков, а не целые блоки. В результате SPV не могут проверять сторонние транзакции, поскольку они не имеют в наличии транзакции определенного блока, а также потому, что у них нет всех *неизрасходованных входящих остатков транзакций* (Unspent Transaction Outputs, UTXO), за исключением их собственных транзакций[3].

## 3.4.1. Регистрация нового узла в сети

Теперь задумайтесь, как будет подключаться к сети Bitcoin новый узел? Это ведь не ваша домашняя интрасеть с сетью 192.168.1.*xxx*, в которой вы можете осуществлять широковещательную передачу по IP 192.168.1.255 так, что любой компьютер, являющийся частью сети 192.168.1.*xxx*, получит широковещательное сообщение. Разумеется, сетевые коммутаторы приспособлены для раздачи таких широковещательных пакетов. Однако помните, что речь идет о глобальном Интернете, в котором располагается Bitcoin. Если вы работаете с узлом в Лондоне, есть вероятность, что другие узлы расположены в Германии, России, Ирландии, США и Индии, и все они связаны через Интернет и обладают каким-то общедоступным IP-адресом.

Вопрос заключается в следующем: когда новый узел присоединяется к сети, как он находит IP-адреса остальных равноправных узлов? Традиционные интернет-приложения используют для этой цели центральный сервер, но в данном случае такого сервера нет. Вы ведь помните, что блокчейн децентрализован? При первом запуске программы Bitcoin Core или BitcoinJ не имеют IP-адреса какого-либо полного узла, зато они оснащены несколькими методами для поиска подключений. Одним из таких методов являются начальные DNS или *DNS-семена* (DNS seeds). Несколько DNS-семян жестко закодированы в программу. Кроме того, в системе DNS поддерживается несколько имен хостов, которые преобразуются в список IP-адресов, на которых работают биткойн-узлы. DNS-семена поддерживаются членами сообщества Bitcoin. Некоторые члены сообщества предоставляют статические начальные значения DNS, вводя IP-адреса и номера портов вручную. Кроме того, некоторые участники сообщества предоставляют динамические начальные DNS-серверы, которые могут автоматически получать IP-адреса активных узлов Bitcoin, которые работают на портах Bitcoin по умолчанию (8333 — для основной сети `mainnet` и 18333 — для тестовой сети `testnet`). Если вы выполните команды `NSLOOKUP` для DNS-семян, вы получите набор IP-адресов, на которых работают биткойн-узлы.

Программы-клиенты Bitcoin Core и BitcoinJ также поддерживают жестко запрограммированный список IP-адресов, которые указывают на некоторые (более одного!) стабильные биткойн-узлы. Такие узлы можно назвать *узлами начальной загрузки*, конечные точки которых уже доступны вместе с исходным кодом. Каждый раз,

---

[3] Про механизм UTXO можно прочитать подробнее по адресу: **https://probtc.info/materialy/39054/**.

когда кто-то загружает двоичные файлы программы, вместе с файлами загружается свежий список активных узлов. Как только установлено соединение хотя бы с несколькими биткойн-узлами, от них очень легко получить список других биткойн-узлов, активных на данный момент. Графическое представление того, как новый узел становится частью сети, можно увидеть на рис. 3.10–3.14.

- **Шаг 1**: представьте, что в какой-то момент времени в сети Bitcoin было шесть активных узлов (рис. 3.10).

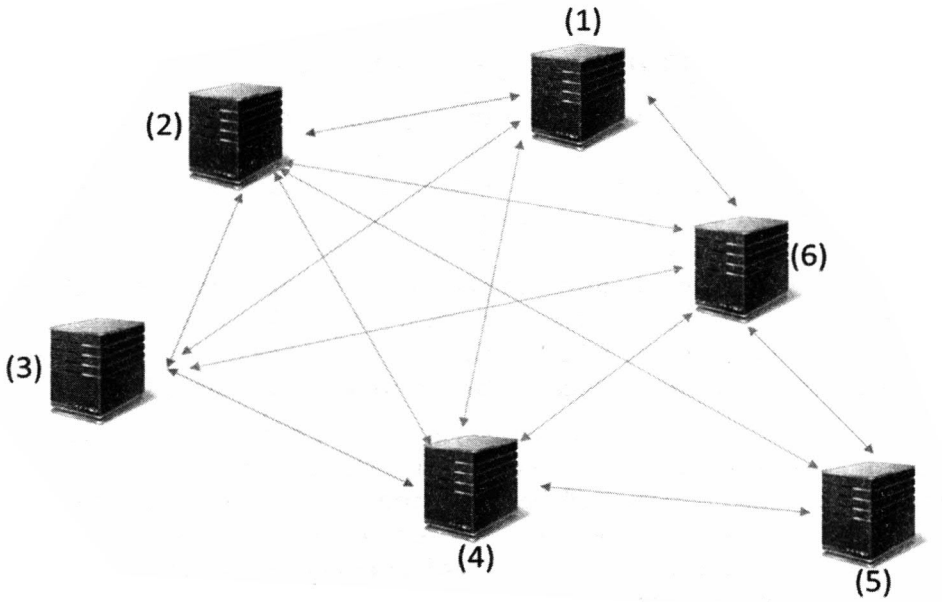



**Рис. 3.10.** Сеть Bitcoin в общем виде

- **Шаг 2**: допустим, только что появился седьмой узел, который пытается присоединиться к существующей сети Bitcoin, но еще не имеет какого-либо соединения (рис. 3.11).
- **Шаг 3**: седьмой узел будет пытаться охватить как можно больше узлов, используя начальные DNS-семена или свой список стабильных узлов Bitcoin (рис. 3.12).

  В последовательности рисунков мы пропустили этап так называемого *разрешения DNS*. Это примерно то же самое, что происходит, когда вы просматриваете любой веб-сайт. Вы вводите буквенное имя сайта, а разрешением DNS называется получение цифрового IP-адреса сайта, который затем используется в качестве адреса веб-сервера для отправки пакетов TCP. Разрешение DNS выполняют в сети специальные DNS-серверы. Чтобы подключиться к новому узлу, наш узел устанавливает TCP-соединение через порт 8333 (порт 8333 принят в сети Bitcoin, но в других версиях сети может отличаться). Затем выполняется «рукопожатие» двух узлов — обмен информацией, такой как номер версии, время, IP-адрес, вы-

сота блокчейна и т. д. Вот действующий биткойн-код для сообщения «Версия» ("version"), определенный в net.cpp:

```
PushMessage( "version", PROTOCOL_VERSION, nLocalServices, nTime, addrYou, addrMe,
nLocalHostNonce, FormatSubVersion(CLIENT_NAME, CLIENT_VERSION, std::vector<string>()),
nBestHeight, true );
```

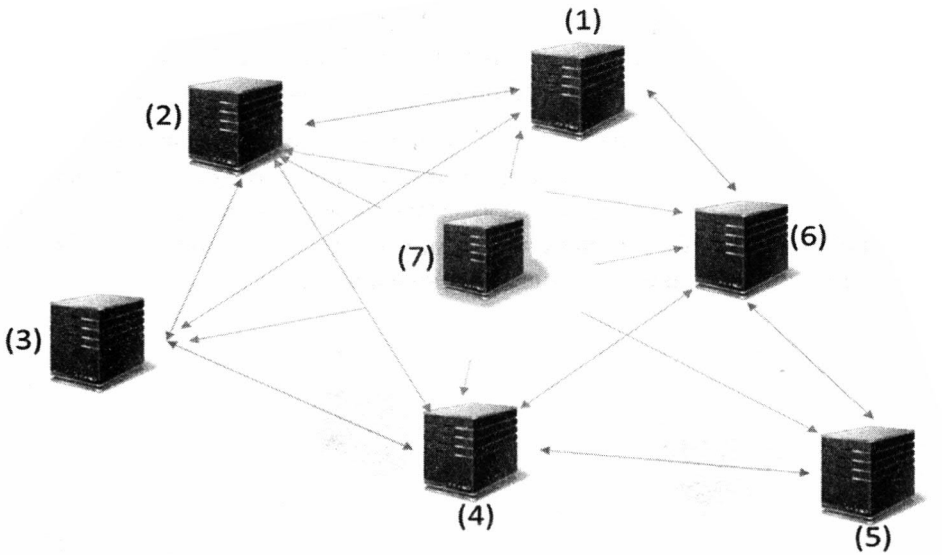



**Рис. 3.11.** Новый узел пытается подключиться к сети

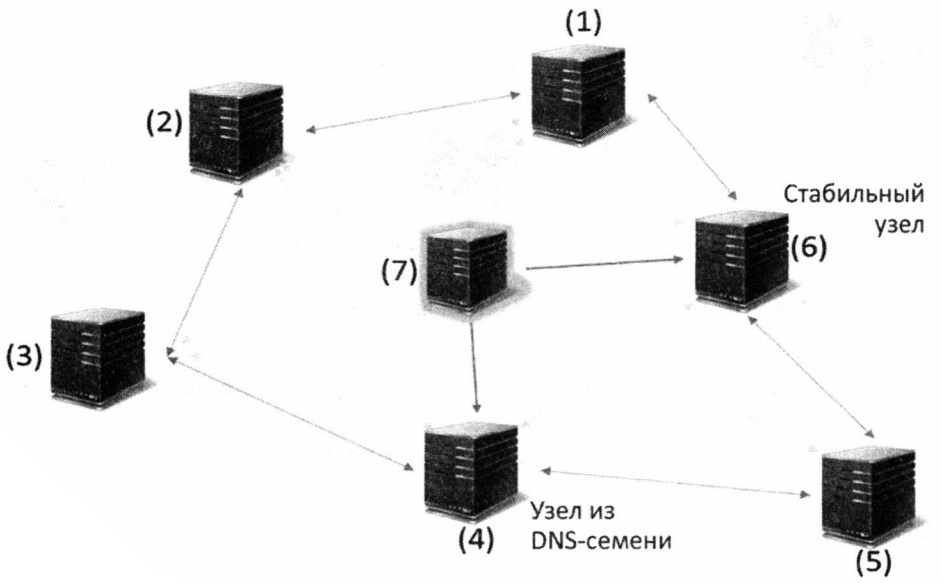



**Рис. 3.12. Новый узел контактирует с другими узлами**

Посредством этого обмена версиями проверяется совместимость между двумя узлами в качестве первого шага к дальнейшему взаимодействию.

- **Шаг 4**: на четвертом шаге запрошенные узлы отвечают списком IP-адресов и номерами портов других активных узлов Bitcoin, о которых они знают. Обратите внимание, что активные узлы не обязательно знают о каждом узле Bitcoin в сети в любое время, поэтому желательно собрать списки с нескольких узлов. Номер порта важен, потому что как только TCP-пакеты достигают узла назначения, именно номер порта используется операционной системой для направления сообщения правильному приложению/процессу, запущенному в системе.

  Пожалуйста, посмотрите на рис. 3.13. Обратите внимание, что только одного стабильного узла может быть достаточно для начальной загрузки списка узлов Bitcoin[4]. Узел должен в процессе работы продолжать обнаруживать новые узлы и подключаться к ним. Это связано с тем, что узлы приходят и уходят в любое время, и ни одно соединение не является надежным.

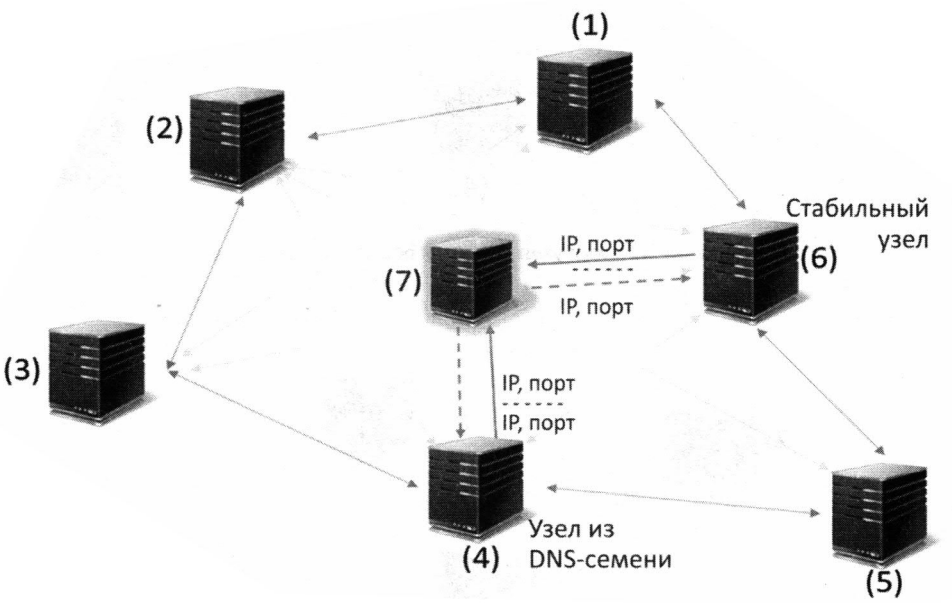



**Рис. 3.13.** Узлы сети Bitcoin отвечают на сетевой запрос нового узла

- **Шаг 5**: на пятом этапе новый узел устанавливает соединение со всеми доступными биткойн-узлами в зависимости от списка, который он получил от узлов, с которыми связался на предыдущем шаге. Это состояние показано на рис. 3.14.

---

[4] Технически достаточно, но очень опасно, если этим первым и единственным узлом окажется злоумышленник, который подсунет список фальшивых узлов. Это называется *атака начальной загрузки*, или *бутстрап-атака*.

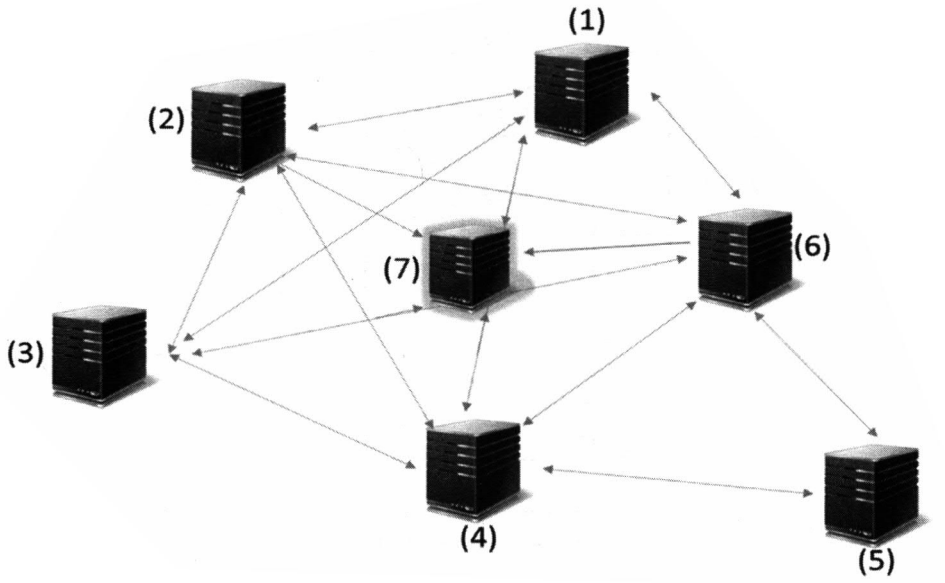



**Рис. 3.14.** Новый узел становится частью сети Bitcoin

## 3.4.2. Bitcoin-транзакции

Транзакции являются основными строительными блоками системы Bitcoin. Транзакции делятся в основном на две наиболее обширных категории:

- *исходные монетарные транзакции* — каждый блок в блокчейне Bitcoin содержит одну монетарную транзакцию, включенную самими майнерами для майнинга новых монет. У майнеров нет контроля над тем, сколько монет они могут добыть в каждом блоке, потому что это контролируется самой сетью. Монетарные транзакции начались с 50 BTC при запуске системы и продолжают уменьшаться вдвое, пока общая сумма монет не достигнет 21 миллиона биткойнов;
- *обычные транзакции* — обычные транзакции в целом очень похожи на перевод денег, когда кто-то пытается перевести некоторую сумму денег, которой он владеет, другому лицу. Как правило, в Bitcoin все активы представлены в виде транзакций. Чтобы потратить определенную сумму, отправитель должен использовать предыдущие транзакции, в которых он получил достаточную сумму, — это и есть обычные транзакции в биткойнах. Основное внимание в этой главе будет уделено таким транзакциям.

Каждый владелец биткойнов может передать монеты кому-то еще, подписав цифровой подписью хэш предыдущей транзакции, в которой он получил монеты, вместе с открытым ключом получателя. Получатель уже имеет открытый ключ плательщика и может проверить транзакцию. Следующая схема (рис. 3.15) взята из «Белой книги» Сатоши Накамото, наглядно демонстрирующей, как работает механизм Bitcoin.

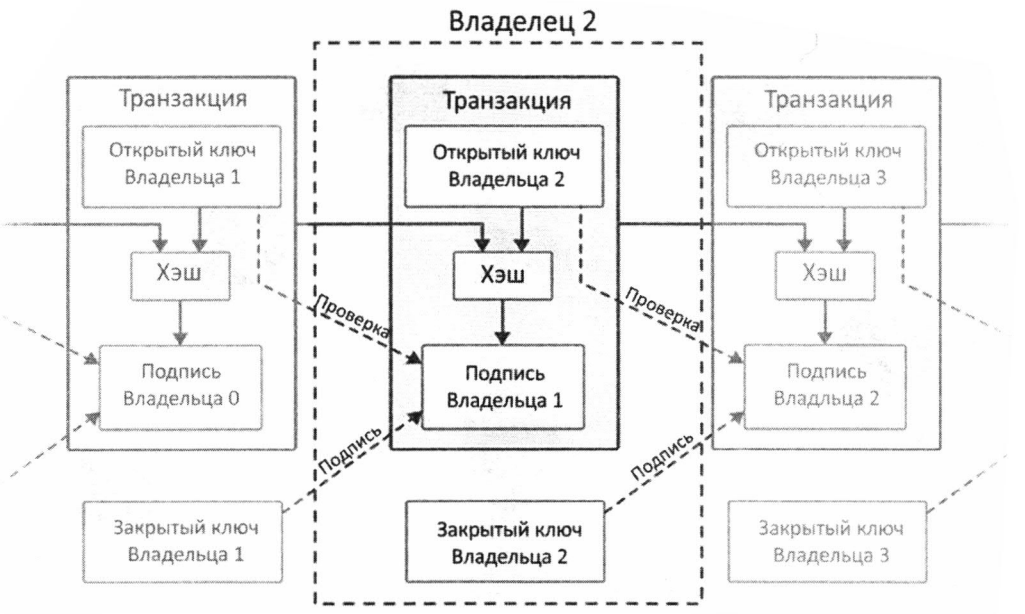



**Рис. 3.15. Транзакции в сети Bitcoin**

Обратите внимание только на выделенный пунктиром фрагмент **Владелец 2** на рис. 3.15. Поскольку Владелец-1 инициирует эту транзакцию, он использует свой закрытый ключ для подписания хэша двух элементов: один — предыдущая транзакция, в которой он сам получил сумму, а второй — открытый ключ Владельца-2. Эта подпись может быть легко проверена с использованием открытого ключа Владельца-1, чтобы убедиться, что она является законной транзакцией. Точно так же, когда Владелец-2 инициирует транзакцию в пользу Владельца-3, он использует свой закрытый ключ для подписи хэша предыдущей транзакции (той, которую он получил от Владельца-1) вместе с открытым ключом Владельца-3. Такая транзакция может быть проверена и подтверждена *любым* участником сети. Очевидно, что поскольку каждая транзакция широковещательная, большинство узлов будут иметь всю историю транзакций и могут предотвратить попытки двойных расходов.

В сети Bitcoin, как уже отмечалось, не существует принципа конечного сальдо, и общая сумма, которой вы владеете, представляет собой сумму всех входящих транзакций на ваши публичные адреса. Вы можете создать столько публичных адресов, сколько захотите. Если у вас есть десять адресов, то, независимо от того, какие транзакции были сделаны по этим адресам, вы можете потратить эти монеты, используя свой закрытый ключ. Если вам, допустим, необходимо потратить пять биткойнов, у вас есть выбор:

- использовать одну из предыдущих транзакций, где вы получили пять или более биткойнов. Перечислите пять биткойнов получателю, уплатите некоторую сумму майнеру в качестве комиссии за транзакцию, а остаток оставьте себе (рис. 3.16);

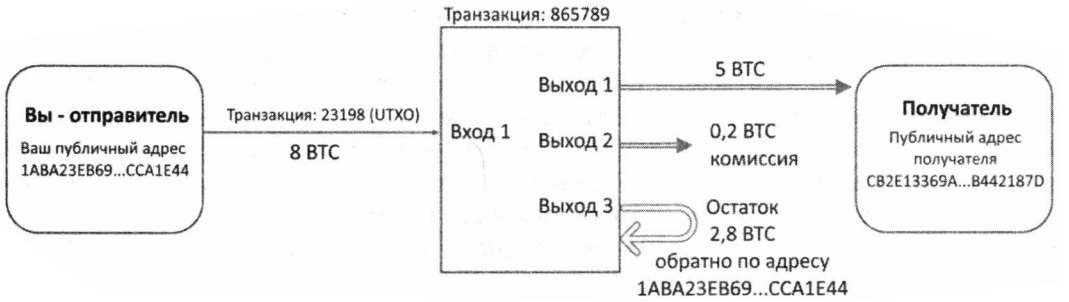



**Рис. 3.16.** Транзакция биткойна с одиночной транзакцией на входе

- использовать несколько предыдущих транзакций, которые вы получили, и сумма которых может превышать пять биткойнов. Перечислите пять биткойнов получателю, уплатите некоторую сумму майнеру в качестве комиссии за транзакцию, а остаток оставьте себе (рис. 3.17).

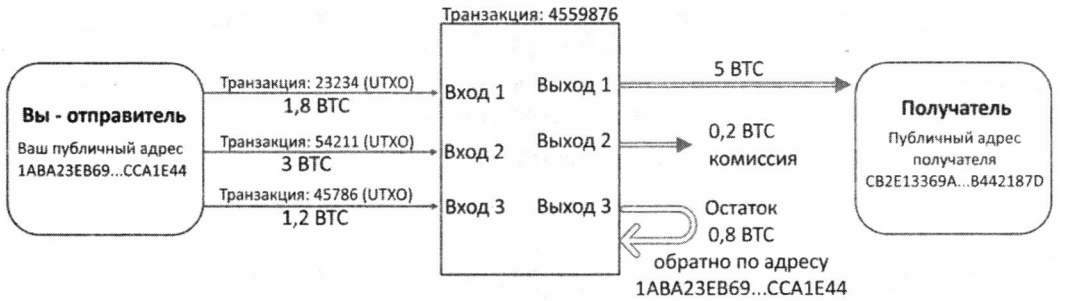



**Рис. 3.17.** Транзакция биткойна с несколькими транзакциями на входе

Как вы можете видеть, каждая транзакция принимает в качестве входных данных предыдущие транзакции. В системе Bitcoin не хранится состояние счета, в котором говорится, что у вас восемь BTC, и вы можете потратить все, что меньше этой суммы, а если вы потратите пять BTC, остаток будет три BTC. В Bitcoin любое действие является транзакцией, в которой есть входы и выходы. Если выход предыдущей транзакции еще не полностью потрачен, то остаток превращается в неизрасходованный входящий остаток для следующих транзакций (UTXO).

Мы знаем, что каждая транзакция в сети транслируется на всю сеть. Независимо от того, обслуживает ли кто-то узел или нет, он все равно может совершить транзакцию, и эта транзакция публикуется на всех доступных биткойн-узлах. Затем узлы-получатели транслируют транзакции на другие узлы, и в большинстве случаев транзакции растекаются по всем узлам сети. Этот механизм иногда называют *протоколом Gossip* (сплетня), и он играет важную роль в предотвращении атак двойного расхода. Вспомните, как в *главе 2* мы говорили, что единственный способ предотвратить двойные расходы — это когда каждый узел знает обо всех транзакциях.

Каждый узел хранит список всех транзакций, которые он «услышал» от других узлов, и транслирует в сеть только новые транзакции, которые еще не были частью

списка. Узлы держат транзакцию в списке до тех пор, пока она не попадет в блок и не станет подтвержденной частью блокчейна. Дело в том, что даже если блок имеет все действительные транзакции и предложен в качестве действительного блока, он все равно может стать осиротевшим, если не является частью самой длинной цепочки. Как только подтверждено, что блок теперь является частью самой длинной цепочки, транзакции, которые находятся в этом блоке, удаляются из списка ожидающих транзакций. Каждый полный узел в сети Bitcoin должен хранить весь список неизрасходованных входящих транзакций (UTXO), даже если они исчисляются миллионами. Если транзакция находится в списке UTXO, то она наверняка не является попыткой двойного расходования. Узел транслирует такие транзакции после подтверждения того, что транзакции не являются атакой с двойным расходом, а также после проверки транзакций с других точек зрения. Если вы хотите спросить, каким образом удается быстро искать миллионы UTXO для проверки двойных расходов, то вы на правильном пути. Поскольку выходы транзакций упорядочены по их хэшам, поиск элемента в упорядоченном хэш-списке выполняется весьма быстро.

Давайте теперь задумаемся и углубимся в сценарий попытки двойных расходов. Вполне возможно, что Алиса (**A**) пытается отправить Бобу (**B**) и Чарли (**C**) одну и ту же транзакцию (напомним, что вход в транзакцию — это выход предыдущей транзакции, и в технологии Bitcoin нет концепции закрытия баланса).

Здесь мы наблюдаем следующий сценарий (рис. 3.18):

- отправитель **A** пытается провести одну и ту же транзакцию для получателей **B** и **C**;
- узел (2) получил транзакцию **A** Tx(1234) → **B**, а узел (3) получил транзакцию **A** Tx(1234) → **C**;
- для узла (2) и узла (3) их соответствующие полученные транзакции *в данный момент* являются законными транзакциями;

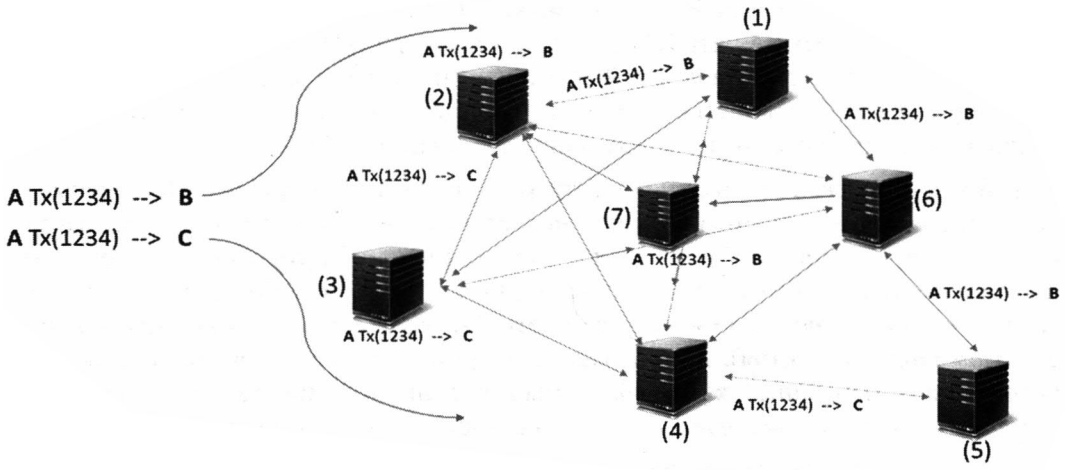



**Рис. 3.18.** Сценарий двойного расхода в сети Bitcoin

- когда узел (3) пытается передать транзакцию **A** Tx(1234) → **C** на узел (2) (ведь каждый узел передает друг другу новые транзакции!), то узел (2) откажется от этой транзакции, поскольку у него уже есть транзакция **A** Tx(1234) → **B** с той же входной транзакцией Tx(1234);
- аналогичные ситуации возникают и с другими узлами, и к моменту обмена данными они могут иметь у себя транзакцию **A** Tx (1234) → **B** или **A** Tx (1234) → **C**, в зависимости от того, что пришло быстрее, но никак не обе транзакции вместе;
- во время майнинга, что бы там узлу ни предлагали, он включит в блок ту транзакцию, которую имеет. Эта транзакция станет частью блокчейна, а остальные узлы, которые содержат другую транзакцию, просто отбросят повторную транзакцию Tx(1234), потому что она перестанет быть неизрасходованной входящей транзакцией. Таким образом, в блокчейн в любом случае попадет только одна из транзакций, и двойное расходование не случится.

## 3.4.3. Консенсус и майнинг блоков

В предыдущем разделе мы рассмотрели *гранулярные* (несвязанные) транзакции. Сейчас мы узнаем, как связывают транзакции при формировании блока, и как достигается консенсус между узлами, благодаря чему вся сеть принимает этот блок. Заметим, что *майнингом блоков* называется именно успешное создание нового блока в цепочке блоков. В Bitcoin это алгоритм распределенного консенсуса PoW, который помогает добывать новые блоки и поддерживает децентрализацию. Достижение распределенного консенсуса в такой сети — очень сложная задача. Хотя такие распределенные системы, как Facebook, Google, Amazon и многие другие, существуют уже десятки лет и владеют миллионами серверов, которым требуется согласованность данных, термин *консенсус* стал очень популярен благодаря сети Bitcoin. В этой главе мы разберемся с основными понятиями консенсуса и майнинга.

Прежде всего запомните, что в сети Bitcoin абсолютно все представлено как транзакции. Если вы хотите выполнить транзакцию, вы должны использовать одну или несколько предыдущих транзакций в качестве входных данных и выполнить новую транзакцию, которая даст выходные данные. Мы уже знаем, что нужно подписать транзакцию, используя свой закрытый ключ, чтобы дать гарантию, что транзакция выполняется нужным человеком. Несмотря на такую криптографическую безопасность, не может ли этот человек повторно подписать сделку, которую он уже провел? Например, Алиса получила десять биткойнов в транзакции с номером транзакции 1234. Она вполне может провести ту же транзакцию 1234 и отправить эти десять биткойнов Бобу и Чарли. Так как она подписала транзакцию своим личным ключом, это будет подлинная транзакция. Что, по вашему мнению, может предотвратить двойные расходы Алисы? Учтите, что в механизме Bitcoin нет способа, которым вы могли бы помешать ей *попытаться* совершить двойные траты, но система разработана таким образом, что попытка не будет успешной. Единственный

способ блокировать такие попытки — это всем участникам знать обо всех проводимых транзакциях. Вот почему все транзакции в биткойнах транслируются по всей сети. Как только транзакция израсходована, она больше не является частью списка UTXO, и в системе генерируется новый номер транзакции, которая станет частью UTXO. А дальше этот входящий остаток сможет потратить только получатель. Это способ, которым узлы могут проверять транзакции.

Итак, единственный способ предотвратить атаку с двойным расходом — это знать все транзакции. Когда вы будете в курсе всех транзакций, вы будете знать о расходах и входящих остатках. Когда майнер предлагает новый блок, необходимо, чтобы все транзакции в блоке были действительными. Означает ли это, что узел, предлагающий новый блок, не сможет включить в него недопустимую транзакцию? Да, не сможет. Он, безусловно, может попытаться ввести в блок мошенническую транзакцию, но остальные узлы отклонят ее. В итоге получится, что все затраты на майнинг блока, включая ресурсы компьютера и электричество, окажутся напрасными! Таким образом, благодаря консенсусу PoW, узел никогда не захочет предлагать недопустимый блок и оплачивать напрасные расходы из своего кармана. Несмотря на отсутствие понятия глобального времени, транзакции объединяются по порядку, чтобы сформировать блок, который становится частью блокчейна. Подчеркнем, что порядок, в котором проводились транзакции, сохраняется в блокчейне. Таким образом, консенсус происходит на уровне блоков, но распространяется до уровня гранулярных транзакций.

Исходя из того, о чем мы говорили до сих пор, теперь мы знаем, что каждый узел в сети Bitcoin имеет свою собственную копию блокчейна, и как такового «глобального блокчейна» не существует — все-таки это децентрализованная сеть! Также верно, что каждый узел поддерживает список UTXO и, когда ему предоставляется шанс (узел выбирается случайным образом — мы увидим, как) предложить блок, узел включает в блок как можно больше транзакций, вплоть до лимита в 1 или 2 Мбайт. Если блок успешно проходит проверку и добирается до блокчейна, ожидающие транзакции удаляются из списка UTXO. Заметим, что узлы в определенные моменты времени могут иметь разные списки ожидающих транзакций, поскольку существует вероятность того, что некоторые транзакции не будут услышаны некоторыми узлами.

Пришло время разобраться, как на самом деле работает алгоритм PoW. Мы знаем о наличии поля сложности в заголовке каждого блока. Каждый майнинговый узел пытается разгадать криптографическую загадку, ожидая, что ему повезет, и он сможет сгенерировать блок. Причина, по которой они так отчаянно пытаются предложить блок, заключается в том, что они получают ощутимую выгоду, когда предлагаемый блок становится частью блокчейна. В каждой транзакции, которую совершают пользователи, они могут предложить некоторую комиссию за транзакцию для майнеров. Мы знаем, что все узлы поддерживают список ожидающих транзакций, которые еще не являются частью блокчейна, и когда они получают возможность предложить блок, они принимают столько транзакций, сколько могут, и формируют блок. Очевидно, что они предпочтут те транзакции, которые дадут им наи-

большую прибыль, и постараются не брать транзакции с минимальными или нулевыми комиссионными отчислениями. Транзакциям с низкой комиссией может потребоваться некоторое время, чтобы попасть в блоки, а у транзакций без комиссии шансы еще ниже. Помимо комиссии за транзакцию, узлы, предлагающие новый блок, получают биткойны. С каждым успешным блоком генерируются новые биткойны, и майнер, предложивший блок, забирает их себе полностью. Это единственный способ создания новых монет в системе Bitcoin, который называется *вознаграждение за блок* (block reward). Майнер включает в предложенный блок специальную транзакцию, называемую *создание монеты*, где адрес получателя — собственный адрес майнера. Когда Bitcoin был впервые запущен, награда за блок составляла 50 биткойнов (BTC). По замыслу, в общей сложности могут быть выпущены только 21 000 000 монет, поэтому награда за блок уменьшается вдвое через каждые 210 000 блоков. Вознаграждение началось с 50 BTC, потом стало 25 BTC, затем 12,5 BTC, и так будет продолжаться до тех пор, пока в какой-то момент времени вознаграждение не достигнет нуля. Далее приведен фрагмент кода из Bitcoin Core (main.cpp), который демонстрирует этот процесс многократного деления пополам:

```
int64_t GetBlockValue(int nHeight, int64_t nFees)
{
    int64_t nSubsidy = 50 * COIN;
    int halvings = nHeight / Params().SubsidyHalvingInterval();

    // Если дальше некуда уменьшать награду за блок,
    // то возвращаем сумму комиссии без награды за блок
    if (halvings >= 64)
        return nFees;

    // Награда делится пополам каждые 210 000 блоков,
    // что случается  приблизительно каждые 4.
    nSubsidy >>= halvings;

    return nSubsidy + nFees;
}
```

Следующий фрагмент кода демонстрирует этот механизм более наглядно[5]:

```
//Делим пополам вознаграждение за блок, остаток 50%
BlockReward = BlockReward >> 1;

//Дважды делим пополам вознаграждение за блок, остаток 25%
BlockReward = BlockReward >> 2;

//Трижды делим пополам вознаграждение за блок, остаток 12.5%
BlockReward = BlockReward >> 3;
```

[5] Напомним, что деление числа пополам равносильно побитовому сдвигу этого числа на один разряд вправо.

Несмотря на то, что вознаграждение выглядят заманчивым, не так-то просто поймать удачу за хвост и стать тем узлом, который может предложить блок. Если вам не повезет, вся ваша работа окажется напрасной — очень жаль! Что же делают узлы, чтобы получить вознаграждение? Решают криптографическую головоломку. Каждый майнинговый узел постоянно работает над предложением блока, но только один из них может преуспеть в данный момент времени. Предположим, что блок уже предложен, и теперь все майнинговые узлы работают над предложением нового блока. Давайте разберем процесс майнинга по шагам:

- **шаг 1**: майнеры используют программное обеспечение для отслеживания транзакций, удаления из списка ожидания тех транзакций, которые уже перешли в блокчейн, отклонения мошеннических транзакций и решения криптографической головоломки, чтобы предложить новый блок и передать его всей сети. Лучшее программное обеспечение с точки зрения авторов книги — это официальная версия Bitcoin Core, но существует много других вариантов. Если вы перейдете по ссылке: **https://bitcoin.org/en/download**, то обнаружите, что официальная версия Bitcoin Core поддерживается в Windows, Linux, macOS, Ubuntu и ARM Linux. Итак, майнеры набирают транзакции (скорее всего те, которые приносят майнеру наибольшую прибыль) до лимита блока в 1 Мбайт (2 Мбайт для Bitcoin Cash), а также хэшируют эти транзакции и генерируют корень Меркла, который станет частью заголовка нового блока. Корень Меркла криптографически однозначно представляет все транзакции блока;
- **шаг 2**: майнеры формируют заголовок блока. На этом этапе известны все данные заголовка, кроме одноразового числа nonce. Задача майнеров состоит в том, чтобы подобрать такое число nonce, при котором двойной хэш заголовка блока меньше, чем текущее целевое значение сложности. Они продолжают изменять одноразовое число до тех пор, пока хэш не будет удовлетворять условию, и нет быстрого способа найти одноразовое число — нужно последовательно пробовать все возможные варианты. В *разд. 3.3.1* мы уже показали, как вычислить целевое значение сложности, используя четыре байта данных, представленные в заголовке, и узнали, как сложность меняется каждые две недели. В общем виде условие головоломки можно записать следующим образом:

  ```
  H[H(Версия | Предыдущий хэш блока | Корень Меркла | Штамп времени | Значение сложности |
  Одноразовое число)] < [Значение сложности]
  ```

- **шаг 3**: майнер продолжает изменять одноразовое число nonce на шаге 2, увеличивая его на единицу, пока оно не удовлетворит условию, — это рекурсивный процесс. Целевое значение сложности для каждого узла одинаковое, и все они пытаются решить одну и ту же задачу, но вспомните, что у них могут быть разные наборы транзакций, и, следовательно, корень Меркла для них будет разным. Поскольку каждый узел пытается расширить самую длинную цепочку блоков, хэш предыдущего блока у всех узлов будет одинаковым.

  Таким образом, чтобы можно было предложить блок всей сети, двойной хэш SHA-256 заголовка блока должен быть меньше целевого значения сложности. Следующий пример иллюстрирует найденное решение головоломки:

```
Target    : 0000000000000074cd0000000000000000000000000000000000000000000000
Hash      : 000000000000074cc4471deff052ced7f07347e4eda86c845a2fcf0553ed7f0
```

Обратите внимание, что значение хэш-функции `Hash` и целевое значение `Target` имеют одинаковое количество ведущих нулей (т. е. 14), однако `74cc` меньше, чем `74cd`, значит, головоломка решена, и теперь этот блок может быть предложен сети для включения в блокчейн. Часто можно видеть, как этот пример упрощают, используя приблизительные значения как цели, так и хэша, и с учетом только ведущих нулей. Разумеется, если хэш имеет больше начальных нулей, чем цель, то он заведомо удовлетворяет условию головоломки. Напомним еще раз: чем больше нулей в цели, тем сложнее найти хэш, который меньше этого значения.

Давайте сопоставим эти знания с реальной реализацией сети Bitcoin. Мы знаем, что среднее время создания блока установлено равным десяти минутам — оно закодировано в двоичных файлах Bitcoin из расчета 2016 блоков за две недели и не изменяется до тех пор, пока не произойдет хардфорк. Вы можете просмотреть блоки и хэши, которые удовлетворяют целевому уровню сложности, на вебсайте **https://blockchain.info** и убедиться, что хэши для разных блоков будут иметь разное количество ведущих нулей. В первые дни работы сети Bitcoin число ведущих нулей составляло около девяти или десяти, а сегодня оно увеличилось до 18–20 нулей. Начальных нулей может стать еще больше по мере того, как более мощные узлы, способные повысить скорость хэширования системы, присоединяются к сети;

- **шаг 4**: как только майнер находит действительный блок, он немедленно публикует блок во всей сети. Каждый узел, который получает этот блок, тут же проверяет, действительно ли майнер, предложивший блок, решил головоломку майнинга. Для проверки достаточно выполнить всего лишь одну операцию:

```
H [H (Версия | Хэш предыдущего блока | Корень Меркла | Штамп времени | Значение сложности |
Найденное одноразовое число)] < [Значение сложности]
```

Обратите внимание, что проверяющие узлы просто используют заголовок блока, который содержит одноразовый номер, найденный предложившим майнером, чтобы удостовериться, что хэш меньше целевого значения и корректен. Если условие головоломки выполнено, то узлы проверяют отдельные транзакции, предложенные в блоке с его корнем Меркла в заголовке блока. Если все транзакции действительны, узлы добавляют этот блок в локальную копию своего блокчейна. В этом новом блоке есть монетарная транзакция, которая генерирует новые монеты в качестве награды для майнера, предложившего действительный блок.

Еще раз подчеркнем, что майнинг блоков не является простой задачей благодаря алгоритму доказательства работы. Если бы это была легкая работа, многие узлы просто продолжали бы пытаться выполнить задачу и наполнили бы сеть поддельными блоками. Оцените по достоинству, как изящно Bitcoin предотвращает подобные ситуации в игровой форме! Для майнеров всегда выгодно играть по правилам, и они не получают никаких дополнительных преимуществ от нарушения правил.

На рассмотренных этапах мы изучили процедуру майнинга, которая реализована в сети Bitcoin. Одной из лучших идей в этом проекте является случайный выбор майнингового узла, который может предложить блок. Никто не знает, кому посчастливится найти правильное одноразовое число nonce и предложить блок, — это чисто случайное явление. Мы уже знаем, что генерировать истинное случайное число весьма сложно, и это является наиболее уязвимым направлением атаки для большинства криптографических реализаций. При таком устройстве системы, как Bitcoin, случайный выбор узла, предлагающего блок, является *действительно* случайным.

Следующая замечательная вещь в майнинге биткойнов — награда за блок. Это монеты, которые майнер получает за создание блока, включенные в этот же блок в виде монетарной транзакции. Майнеры также получают комиссионные за транзакции, которые они включили в блок. Итак, вознаграждение майнера представляет собой комбинацию вознаграждения за блок и комиссии за транзакцию:

```
Вознаграждение за майнинг = Вознаграждение за блок + Сумма комиссионных за все транзакции в блоке
```

Мы знаем, что майнинг — это единственный способ эмиссии новых биткойнов, но является ли это целью майнинга? Нет! Цель майнинга — добыча новых блоков, генерация новых биткойнов, а также сбор платы за транзакции — стимулирование майнеров, чтобы все больше людей интересовались майнингом. Действительно, зачем вам добывать блоки, если вы не зарабатываете хорошие деньги? Опять мы возвращаемся к теории игр. Правильный механизм стимулирования является ключом к тому, чтобы сделать систему децентрализованной и самодостаточной. Обратите внимание, что в системе Bitcoin нет способа штрафовать узлы, которые не играют честно, есть только поощрение честного поведения. Участники сети Bitcoin, будь то простые люди, которые используют Bitcoin для платежей или майнеры, — все они идентифицируются с помощью своих открытых ключей. Для них можно сгенерировать как можно больше пар ключей, и это делает Bitcoin псевдоанонимной системой. Узел не может быть однозначно идентифицирован с помощью его открытого ключа, который он использовал в транзакции с монетами, т. к. в следующий момент он может создать новую пару ключей и объявить свой новый сетевой адрес. Таким образом, надлежащая мотивация — лучший способ обеспечить честную игру участников системы. Вот она, теория игр во всей красе!

А теперь задумайтесь над новым вопросом. Допустим, узел проверил новый блок, убедился, что одноразовое число, транзакции и прочие атрибуты являются корректными, и включил его в свою локальную копию блокчейна. Означает ли это, что все транзакции, которые были в блоке, с этого момента исполнены и подтверждены? Нет, не совсем так! Существует вероятность того, что два блока попали в сеть одновременно, и хотя некоторые узлы начали добавлять в блокчейн один из них, существует вероятность того, что большинство узлов использовали другой блок. В конце концов, действующей веткой блокчейна является самая длинная цепочка. Это ситуация, когда блок, который является абсолютно допустимым, со всеми законными транзакциями и надлежащим значением nonce, которые удовлетворяли задаче майнинга, все еще может быть отвергнут сетью Bitcoin. Такие блоки, кото-

рые не становятся частью окончательного блокчейна, называются, как мы уже знаем, *осиротевшими блоками*. Теперь нам ясно, что существует определенная возможность того, что один или несколько блоков могут стать осиротевшими в любое время. Поэтому лучше подождать, пока в цепочку не будет добавлено еще несколько блоков. Если после некоторого блока добавляется еще несколько блоков, транзакции этого блока получают подтверждения по числу добавленных блоков. Другими словами, когда транзакция получает несколько подтверждений, можно с уверенностью полагать, что она является частью окончательной цепочки и не станет осиротевшей. Хотя не существует жесткого правила, которое определяет, сколько подтверждений нужно получить, прежде чем принимать транзакцию, общепринятой практикой является шесть подтверждений. Даже с четырьмя подтверждениями можно с уверенностью предположить, что транзакция была подтверждена, но шесть — еще лучше, потому что при большем количестве подтверждений шансы на то, что блок потеряется, экспоненциально уменьшаются.

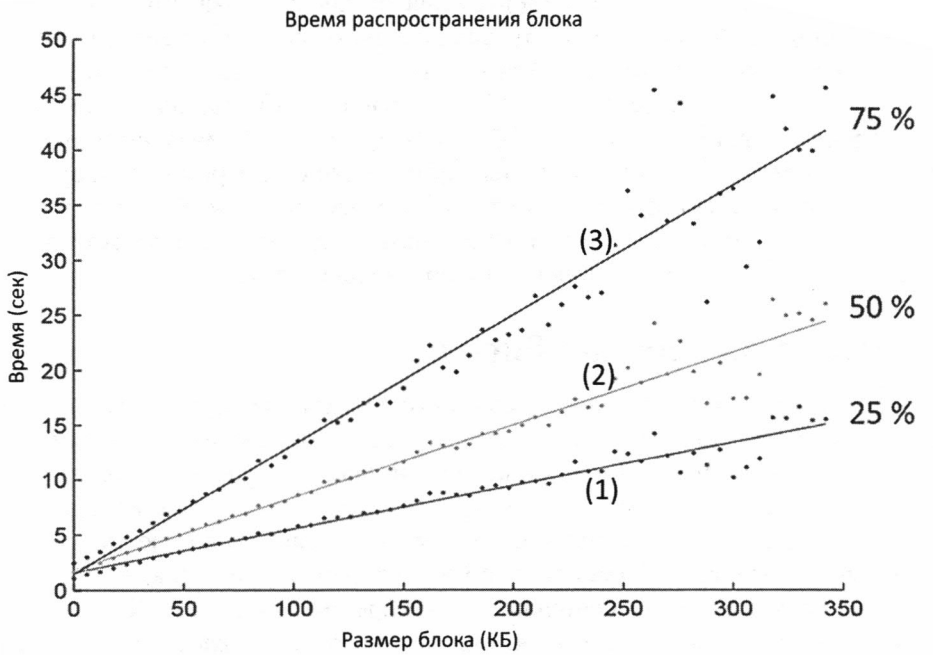
## 3.4.4. Распространение блока

Bitcoin использует майнинг с доказательством работы для случайного выбора узла, который может предложить действительный блок. Когда майнер находит действительный блок, он передает этот блок всей сети. Распространение блока в сети происходит так же, как и распространение транзакции. Каждый узел, который получает новый блок, далее транслирует его другим узлам, так что в конечном итоге блок достигает все узлы в сети. Отметим, что узел не транслирует блок, если он не является частью самой длинной цепочки с точки зрения этого узла. Как только узлы получают новый предложенный блок, они не только проверяют заголовок и значение хэш-функции, но также проверяют каждую транзакцию, которая была включена в этот блок. Из предыдущего материала вы узнали, что для узла в сети Bitcoin проверка блока немного сложнее по сравнению с проверкой транзакций. Подобно транзакциям, существует вероятность, что два действительных блока предлагаются одновременно. В таком сценарии узел сохранит оба блока, но примет тот, который относится к самой длинной цепочке.

Следует понимать, что всегда существует задержка распространения блока по всей сети и достижения им каждого узла. Соотношение между размером блока и затраченным временем линейно пропорционально в том смысле, что каждый килобайт, добавленный к размеру блока, пропорционально увеличивает задержку. Очевидно, что задержка распространения блоков влияет на скорость роста блокчейна. Декер и Ваттенхофер провели исследование скорости распространения блоков. Взгляните на рис. 3.19, где показано соотношение между размером блока и временем, которое потребовалось для достижения 25% (1), 50% (2) и 75% (3) отслеживаемых узлов.

Основной причиной таких сетевых задержек является пропускная способность сети, и она никогда не бывает одинаковой во всех регионах земного шара. Кроме того, мы знаем, что широковещательные пакеты блоков проходят через множество сетевых переходов Интернета, чтобы наконец достичь всех узлов. Типичный биткойн-блок имеет размер 1 Мбайт, а новый вариант Bitcoin после хардфорка (Bitcoin

Cash) имеет размер блока 2 Мбайт. Нетрудно представить себе ограничения, связанные с задержкой. Согласно исследованию сети, существует более миллиона узлов Bitcoin, которые периодически подключаются к сети Bitcoin в течение месяца, и есть тысячи полных узлов, которые почти всегда подключены к сети.




Источник: Yonatan Sompolinsky, Aviv Zohar - «Как ускорить обработку транзакций»

**Рис. 3.19. Зависимость времени распространения блока в зависимости от его размера**

## 3.5. Промежуточные итоги главы

Если мы просто запишем ключевые события в том порядке, в котором они происходят, то список будет выглядеть так:

- все новые транзакции транслируются на все узлы сети Bitcoin;
- каждый майнинговый узел, который получает новые транзакции, собирает их в блок;
- каждый майнер работает над поиском решения криптографической головоломки для своего блока, чтобы иметь возможность предложить его сети;
- когда майнеру везет, и он находит правильное одноразовое число для головоломки, он передает блок всем узлам;
- узлы принимают предложенный блок, только если одноразовое число и все транзакции в нем корректны и еще не проведены;

- узлы сети Bitcoin выражают свое согласие с блоком, и майнеры начинают работать над созданием следующего блока в цепочке, используя хэш утвержденного блока в качестве предыдущего хэша для нового блока, который они будут добывать.

## 3.6. Скрипты Bitcoin

В предыдущих разделах мы говорили о выполнении транзакций на общем уровне. Теперь мы приступим к изучению реальных программных конструкций, которые обеспечивают выполнение транзакций. Все входные и выходные данные биткойн-транзакций встроены в скрипты (сценарии). Скрипты Bitcoin основаны на стеке, как мы вскоре увидим, и выполняются слева направо. Помните, что скрипты Bitcoin не являются полными по Тьюрингу, поэтому вы не можете делать все то, что возможно с помощью других полных по Тьюрингу языков, — таких как C, C++ или Java и им подобных. В Bitcoin нет концепции циклов, поэтому время выполнения скриптов не является переменным и пропорционально количеству инструкций. Это означает, что скрипты выполняются за ограниченное время, и у них нет возможности застрять в цикле. Кроме того, что наиболее важно, скрипты в любом случае завершаются. Где и когда работают скрипты? Всякий раз, когда выполняются транзакции — через приложение кошелька или любую другую программу — скрипты встраиваются в транзакции, и майнеры должны запускать эти скрипты во время майнинга. Назначение скриптов Bitcoin состоит в том, чтобы позволить сетевым узлам убедиться, что имеющиеся средства востребованы и потрачены только теми правомочными сторонами, которые действительно владеют ими.

### 3.6.1. Еще раз про транзакции в сети Bitcoin

Транзакция в сети Bitcoin — это передача ценности, которая распространяется по всей сети и завершается в некотором блоке в блокчейне. Как правило, пользователю кажется, что биткойны переводятся с одного счета или кошелька на другой, но в действительности происходит перемещение ценности с выхода одной транзакции на вход другой транзакции. Прежде чем углубляться в детали, отметим, что адреса в сети Bitcoin — это результат двойного хэширования открытого ключа участников. Открытый ключ сначала хэшируется с использованием SHA-256, а затем с помощью алгоритмов хэширования RIPEMD-160, которые дают на выходе 160-битные адреса Bitcoin. Мы уже рассмотрели эти методы хэширования в *главе 2*. Давайте более детально изучим механизм транзакций. Посмотрите на ветвление транзакций (рис. 3.20), которое обычно происходит в сети Bitcoin.

Выходные данные предыдущих транзакций становятся входными данными для новых транзакций, и этот процесс продолжается бесконечно. Если вы получили 100 тыс. сатоши (100k) из некоторой предыдущей сделки, эта сумма станет источником для новых транзакций. Обратите внимание, что в транзакции TX0 вы перевели получателям 40 тыс. (40k) и 50 тыс. (50k) сатоши, а оставшаяся сумма (10 тыс.) стала вознаграждением майнера. *По умолчанию оставшаяся сумма вы-*

*плачивается майнеру*, поэтому нужно быть осторожным, чтобы не попадать в такие ситуации против своей воли. В этой же ситуации из оставшейся суммы в 10 тыс. вы можете перевести 9 тыс. на свой адрес и оставить 1 тысячу в качестве платы за майнинг. Если сумма не израсходована — в том смысле, что выход последней транзакции не используется в качестве входных данных для новой транзакции, она превращается в UTXO — остаток, который можно потратить позже. Очевидно, что предшествующие данному UTXO остатки уже были потрачены ранее. Таким образом, сумма всех остатков, объединенных для всех имеющихся у вас адресов (открытых ключей), является содержимым вашего кошелька.

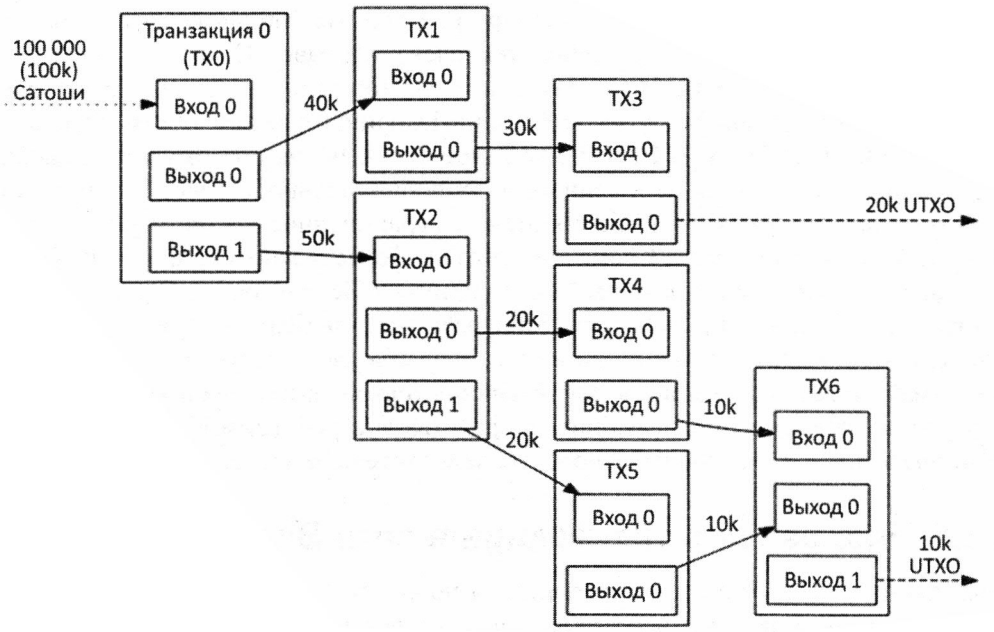



**Рис. 3.20. Типовая структура биткойн-транзакции**

На минутку остановитесь и подумайте, как это можно запрограммировать. Вспомните, что и входы, и выходы транзакций снабжены соответствующими скриптами. Только с помощью скриптов можно гарантировать, что вы являетесь авторизованным пользователем для совершения транзакции, и у вас есть необходимая сумма, которую вы получили от предыдущей транзакции. Это означает, что и входы, и выходы одинаково важны. Вот как выглядит содержимое транзакции:

```
Выход транзакции = количество биткойнов для перевода + выходной скрипт
Вход транзакции = ссылка на предыдущий выход транзакции + скрипт входа
```

На какой скрипт нужно обратить внимание в первую очередь — на входной или выходной? На самом деле это проблема курицы и яйца. Но все-таки сначала мы возьмем выходной скрипт предыдущей транзакции, потому что он используется входным скриптом текущей транзакции. Постарайтесь правильно понять логическую связь между скриптами: при выполнении транзакции выходной скрипт теку-

щей транзакции существует только для того, чтобы запустить *будущую* транзакцию, которая может использовать его как вход, но для текущей транзакции нужен выходной скрипт *предыдущей* транзакции, который позволяет вам получить и потратить входящий остаток. Вот почему выходные скрипты имеют открытый ключ получателя и передаваемое значение (количество биткойнов). Когда выходные скрипты используются в качестве входных данных, их основное назначение заключается в проверке подписи с открытым ключом. Выходные скрипты обозначаются термином `ScriptPubKey`. Если этот выход не израсходован в следующей транзакции, он превращается в UTXO.

Входной скрипт текущей транзакции по сути является механизмом использования предыдущей транзакции, остаток которой вы пытаетесь потратить. Соответственно, он должен содержать ссылку на предыдущую транзакцию. Хэш предыдущей транзакции и индекс `{hash, index}` однозначно указывают на источник суммы, которую вы сейчас тратите. Если вы утверждаете, что являетесь получателем предыдущей транзакции, вы должны предоставить свою подпись, чтобы подтвердить, что вы являетесь законным владельцем открытого ключа, с которым была совершена транзакция. Это позволит вам использовать сумму, поступившую в предыдущей транзакции. Кроме того, вы должны предоставить свой открытый ключ, который хэшировался в составе адреса назначения в предыдущей транзакции. Входные скрипты также известны как `ScriptSigs`. Конечная цель скрипта — поместить подписи и ключи в стек.

Типичная биткойн-транзакция имеет следующие поля (табл. 3.5).

***Таблица 3.5.** Поля транзакции Bitcoin*

| Поле | Размер | Описание |
|---|---|---|
| Номер версии | 4 байта | В данный момент 1. Указывает узлам и майнерам сети Bitcoin, какой набор правил использовать для проверки данной транзакции |
| Число входов | 1–9 байтов | Положительное целое число. Указывает общее количество входов |
| Список входов | переменная длина | Список всех входов транзакции |
| Число выходов | 1–9 байтов | Положительное целое число. Указывает общее количество выходов |
| Список выходов | переменная длина | Список всех выходов транзакции |
| Время удержания | 4 байта | В настоящее время не используется. Указывает на то, должна ли транзакция быть включена в блок немедленно после проверки майнером, или должна быть задержана на некоторое время перед добавлением в блок |

Давайте теперь рассмотрим другое представление той же структуры транзакций, которую мы обсуждали в предыдущем разделе. Вам следует более детально разобраться в составных частях транзакции.

Взгляните на рис. 3.21. Как здесь можно видеть, все элементы данных, такие как подписи или открытые ключи, встроены в скрипты и являются частью транзакции. Инструкции скрипта помещаются в стек и последовательно выполняются. Вскоре мы рассмотрим этот вопрос более подробно.

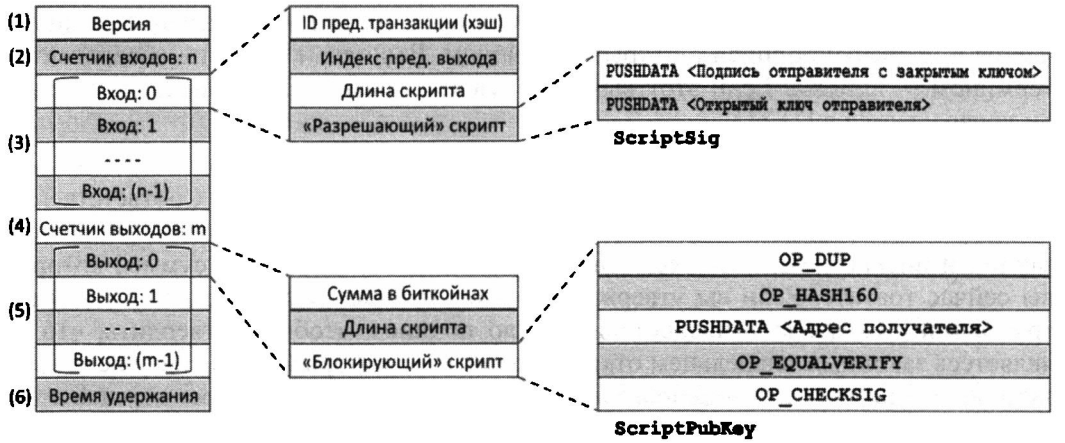



**Рис. 3.21. Компоненты транзакции Bitcoin**

Когда узлы Bitcoin получают широковещательные транзакции, они проверяют эти транзакции по отдельности, комбинируя выходной скрипт предыдущей транзакции со входным скриптом текущей транзакции и выполняя следующие шаги:

- находим предыдущую транзакцию, выходные данные которой используются в качестве входных данных для этой транзакции. Поле **ID предыдущей транзакции** содержит хэш предыдущей транзакции;
- в выходе предыдущей транзакции находим точный индекс, по которому была направлена сумма. В транзакции может быть несколько получателей, поэтому индекс используется для идентификации инициатора текущей транзакции, адрес которой был использован в качестве получателя для предыдущей транзакции;
- используем выходной скрипт предыдущей транзакции, т. е. скрипт разблокировки `ScriptSig`. Обратите внимание, что на рис. 3.21 перед скриптом есть поле, в котором указывается длина этого скрипта;
- соединяем этот выходной скрипт с входным скриптом, просто присоединяя его к скрипту проверки, а затем выполняем этот скрипт проверки (помните, что мы используем стековый язык скриптов);
- значение суммы фактически присутствует в выходном скрипте `ScriptPubKey`. Это скрипт блокировки, который блокирует выход транзакции и гарантирует, что только законный владелец адреса Bitcoin, которому адресована эта транзакция, позже сможет претендовать на входящий остаток. Обратите внимание, что перед ним также есть поле длины скрипта. В текущей транзакции этот выходной скрипт предназначен только для информации, но сыграет свою роль в будущем, когда получатель попытается потратить остаток;

- таким образом, выходной скрипт является и скриптом проверки, который проверяет подписи и решает, имеет ли текущий вход транзакции право расходовать предыдущий UTXO. Если скрипт проверки выполняется успешно, подтверждается, что транзакция действительна и фактически проведена.

Давайте рассмотрим это объяснение на примере. Предположим, что Алиса отправляет Бобу 5 BTC. Это означает, что Алиса получила, как минимум, 5 BTC в одной из предыдущих транзакций, которая была заблокирована с помощью `ScriptPubKey`. Алиса может заявить, что является законным получателем этой транзакции, разблокировав ее с помощью `ScriptSig`, а затем потратить ее на Боба. Теперь, если Боб попытается потратить 3 BTC на Чарли и 2 BTC на себя, это будет выглядеть, как показано на рис. 3.22.

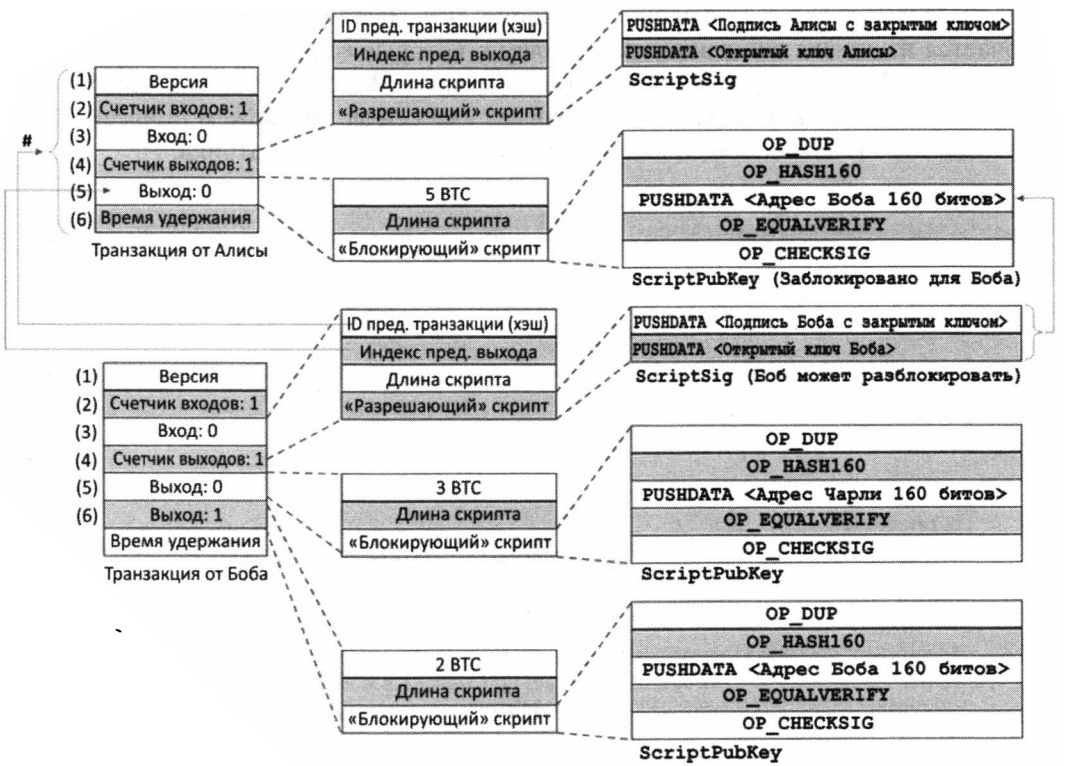



**Рис. 3.22.** Практический пример скрипта Bitcoin

Когда сеть Bitcoin, точнее майнеры, получают транзакцию от Алисы, они убеждаются, что это действительная транзакция, и включают ее в свои блоки (кто-то один предложит свой блок раньше других). Когда это происходит, выходные данные этой транзакции становятся частью UTXO для Боба, который впоследствии может их потратить. А дальше в нашем примере Боб тратит входящий остаток на Чарли. Боб использовал предыдущую транзакцию, разблокировав ее своей подписью и открытым ключом, чтобы доказать, что он является владельцем адреса Bitcoin, на

который отправила деньги Алиса. Обратите внимание, что в транзакции Боба есть два выхода: Боб получил пять биткойнов от Алисы и платит три биткойна Чарли. Поэтому Боб должен перевести остаток самому себе, чтобы создать UTXO и потратить два биткойна в будущем. В транзакции Боба три биткойна для Чарли заблокированы при помощи скрипта блокировки, чтобы только Чарли мог их потратить.

Наверное, вы сейчас задумались о том, как объединяются и выполняются скрипты? Вспомните, что скрипт разблокировки текущей транзакции запускается вместе со скриптом блокировки предыдущей транзакции. Как уже говорилось, запуск скриптов является задачей майнера, а не программного обеспечения кошелька. В предыдущем примере, когда Боб совершает транзакции, майнеры выполняют скрипт разблокирования `ScriptSig` из транзакции Боба, а затем сразу же запускают скрипт блокировки `ScriptPubKey` из транзакции Алисы. Если комбинированный скрипт успешно выполняется в стеке и возвращает значение «ИСТИНА», то транзакция Боба исключается из списка ожидания всеми узлами, которые ее проверяли. В следующем разделе мы более подробно рассмотрим скрипты Bitcoin и то, как виртуальная машина выполняет команды в стеке. Однако уже сейчас мы можем привести фрагмент кода, представляющий транзакцию с точки зрения разработчика:

```
{
  "hash": "a320ba8bbe163f26cafb2092306c153f87c1c2609b25db0c13664ae1afca78ce",
  "ver": 1,
  "vin_sz": 1,
  "vout_sz": 1,
  "lock_time": 0,
  "size": 51,

  "in":[
      {
         "prev_out":{
            "hash":"83cd5e9b704c0a4cb6066e3a1642b483adc8f73a76791c82a73dfa381281d32f",
            "n":0
         },
         "scriptSig":"63883d3d2dea35029d17d25b8a926675def0045c397d3df55b0ae145ef80db7849599b930220
          ab13bd2dda2ca0a67e2c5cd28030bb9b7b3dcacf176652dac82fe9d5873f3409661281d32f6d35b46906cd56
          2bf8b48f4f938c077bcb29d46b0560fa5c61813d3d2d"
      }
   ],

  "out":[
      {
         "value":"0.08",
         "scriptPubKey":"OP_DUP OP_HASH160 b3a2c0d84ec82cff932b5c3231567a0d48ab4c78
OP_EQUALVERIFY OP_CHECKSIG"
      }
   ]
}
```

Обратите внимание, что транзакции Bitcoin не зашифрованы, поэтому можно находить и просматривать любую транзакцию, когда-либо включенную в блок.

## 3.6.2. Скрипты

Скрипт — это список инструкций в каждой транзакции, которые описывают, как следующий человек может получить доступ к полученным биткойнам и потратить их. Bitcoin использует основанный на стеке язык сценариев, не полный по Тьюрингу, где инструкции обрабатываются слева направо.

В предыдущем разделе мы рассмотрели скрипты входа и выхода. Теперь мы знаем, что скрипт входа `ScriptSig` является скриптом разблокировки и состоит из двух компонентов: открытого ключа и подписи. Открытый ключ используется потому, что он хэшируется с адресом Bitcoin, на который была отправлена предыдущая транзакция. Цифровая подпись ECDSA подтверждает владение открытым ключом, а значит, и адресом Bitcoin, который будет иметь возможность тратить полученные деньги. Точно так же выходной скрипт `ScriptPubKey` в предыдущей транзакции должен заблокировать транзакцию для законного владельца адреса Bitcoin. Эти два скрипта: `ScriptSig` текущей транзакции и `ScriptPubKey` предыдущей транзакции, выполняются вместе. Посмотрите, как выглядит комбинированный скрипт (рис. 3.23).

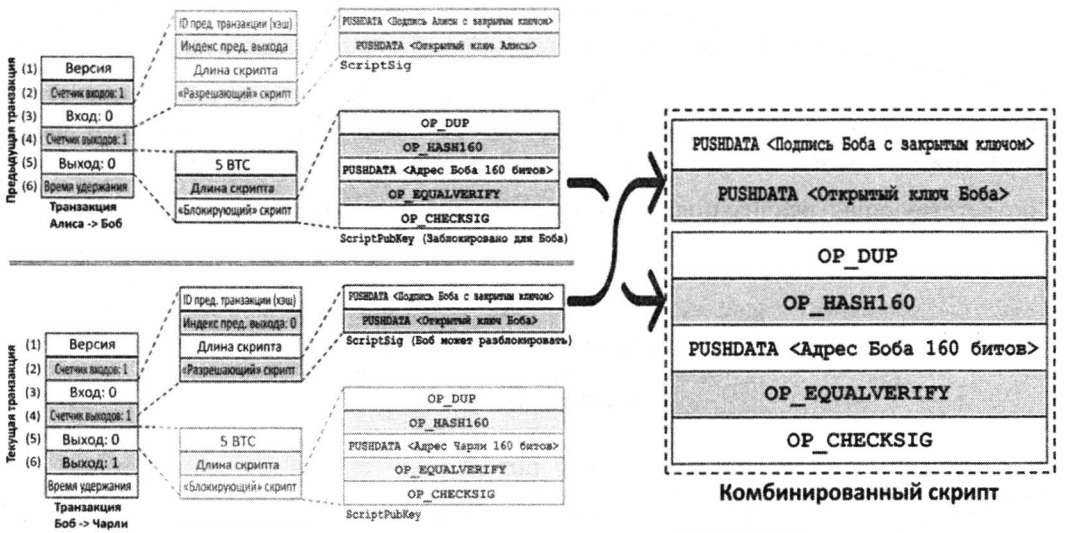



**Рис. 3.23. Формирование комбинированного скрипта проверки**

Следует заметить, что скрипт Bitcoin либо успешно выполняется, либо не работает. Когда транзакции действительны, скрипты успешно выполняются без каких-либо ошибок. Язык скриптов Bitcoin является очень упрощенной версией языков программирования и весьма мал — всего 256 инструкций. Притом из этих 256 инструкций 15 — отключены, а 75 — зарезервированы для последующего использования. Основные инструкции содержат математические и логические (`if/then`) опера-

торы, отчеты об ошибках и просто операторы возврата. Помимо них, есть некоторые дополнительные криптографические инструкции, такие как хэширование, проверка подписи и другие. Мы не станем приводить все доступные инструкции и сосредоточимся только на тех, которые будем использовать в этой главе. Вот некоторые из них:

- `OP_DUP` — просто дублирует верхний элемент в стеке;
- `OP_HASH160` — двукратное хэширование, сначала с SHA-256, а затем с RIPEMD-160;
- `OP_EQUALVERIFY` — возвращает `TRUE`, если входы совпадают, в противном случае возвращает `FALSE` и отмечает транзакцию как недействительную;
- `OP_CHECKSIG` — проверяет, является ли входная подпись действительной подписью, используя открытый входной ключ для хэша текущей транзакции.

Чтобы запустить скрипт, нам просто нужно поместить инструкции в стек, а затем выполнить их по порядку. Помимо памяти, которую занимает стек, дополнительная память не требуется, и это делает скрипты Bitcoin очень эффективными. Как вы уже видели, в скрипте есть два вида инструкций: инструкции данных и коды операций. Приведенный только что список — это коды операций, а комбинированный скрипт проверки, который мы видели ранее, содержит оба вида инструкций. Инструкции данных предназначены просто для помещения данных в стек, а назначение кодов операций состоит в том, чтобы выполнить некоторые операции над данными в стеке и выйти, если это необходимо.

Давайте обсудим, как транзакция Боба будет выполняться с точки зрения стека. Вспомните комбинированный скрипт, в котором Боб пытается отправить Чарли транзакцию на основании ранее полученной транзакции от Алисы (рис. 3.24).

Соответствующая реализация на основе стека будет выглядеть, как показано на рис. 3.25.

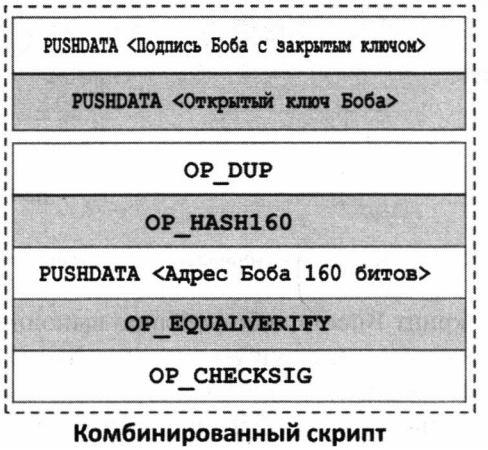



**Рис. 3.24. Комбинация скриптов `ScriptPubKey` и `CheckSig`**

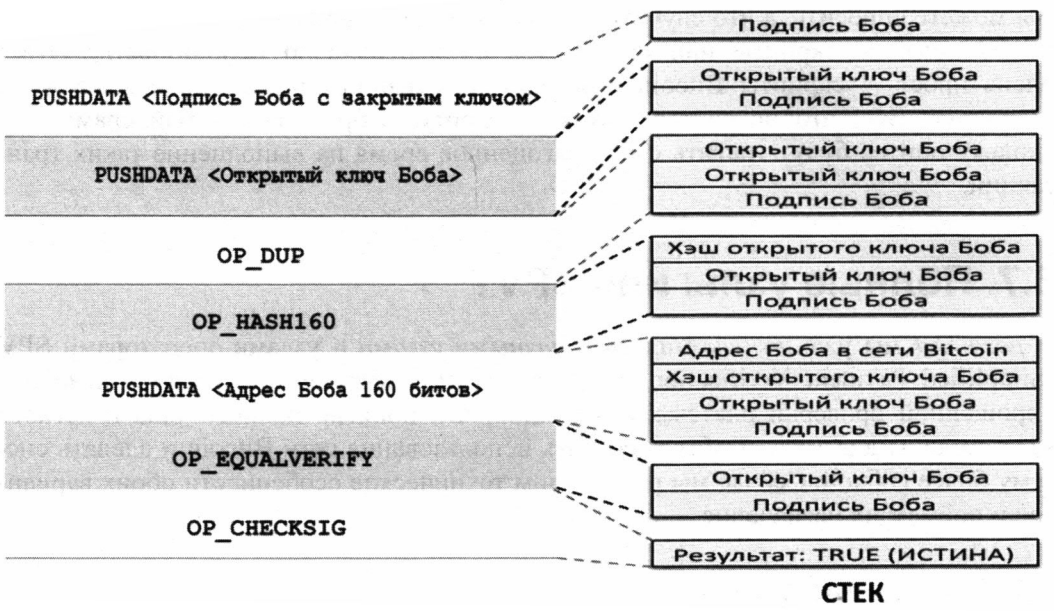



**Рис. 3.25.** Пример реализации скрипта Bitcoin на основе стека

Эта реализация, основанная на стеке, не требует пояснений, но все же мы кратко покажем, что здесь происходит:

- сначала выполняется инструкция данных, которая помещает подпись Боба в стек;
- следующая инструкция помещает в стек открытый ключ Боба;
- затем следует код операции `OP_DUP`. Он дублирует первый[6] элемент в стеке, поэтому открытый ключ Боба дублируется и становится третьим элементом в стеке;
- далее выполняется код операции `OP_HASH160`, который дважды хэширует открытый ключ Боба: один раз по алгоритму SHA-256, а затем по алгоритму RIPEMD-160. Полученное 160-битное число заменяет открытый ключ Боба и становится вершиной стека;
- затем выполняется инструкция данных, которая помещает в стек биткойн-адрес Боба (160 битов);
- далее операция `OP_EQUALVERIFY` проверяет два верхних элемента в стеке и, если они совпадают, выводит их оба, в противном случае выдается ошибка, и скрипт завершается;
- затем выполняется операция `OP_CHECKSIG`, которая проверяет открытый ключ на соответствие подписи для проверки подлинности владельца. Эта операция также использует два входных значения и может выводить их из стека.

[6] Отсчет элементов стека начинается с нуля.

Вы можете спросить, а что случится, если кто-то попробует внедрить в транзакции мошеннические скрипты или попытается их неправильно использовать? Ответ очень прост — скрипты Bitcoin строго стандартизированы, и майнеры следуют стандарту. Все, что не соответствует стандарту, отбрасывается майнерами, поскольку они не будут тратить свое драгоценное время на выполнение таких транзакций.

# 3.7. Полные узлы или SPV?

В *разд. 3.4* мы уже познакомились с *полными узлами* и узлами-операторами SPV (Simplified Payment Verification) или, как их еще называют, *легкими узлами* для упрощенной проверки платежей. Очевидно, что понятия полного узла и легкого узла введены для того, чтобы упростить использование сети Bitcoin и сделать систему более гибкой. Сейчас мы рассмотрим технические особенности обоих вариантов и поймем их назначение.

## 3.7.1. Полные узлы

Полные узлы являются наиболее важным компонентом сети Bitcoin. Именно они поддерживают работу сети Bitcoin из разных уголков Земного шара. Как мы уже говорили, полный узел загружает весь блокчейн со всеми транзакциями, начиная с блока генезиса до последнего известного блока. Последний блок определяет длину[7] блокчейна.

Полные узлы чрезвычайно безопасны, потому что они хранят всю цепочку блоков. Чтобы злоумышленник смог обмануть узел, необходимо представить альтернативную цепочку блоков, что практически невозможно. Истинная цепочка является самой накопительно длинной (кумулятивной) цепочкой в алгоритме PoW, и с вычислительной точки зрения совершенно нереально предложить новый мошеннический блок. Если в блоке хоть одна транзакция недействительна, затраты на майнинг, выполняемый злоумышленником, будет напрасным, потому что другие узлы не примут этот блок. Мошеннический блок очень скоро станет сиротой — блоком, который отвергли остальные узлы. Полные узлы не полагаются на сеть для подтверждения транзакции, потому что они самодостаточны. Они просто заинтересованы в скорейшем получении новых блоков, предлагаемых другими узлами, чтобы обновить свою локальную копию после проверки блоков. Итак, каждый полный узел должен обрабатывать все транзакции, он должен хранить всю базу данных, каждую транзакцию, которая в настоящее время передана в сеть для обработки, знать каждую транзакцию, которая когда-либо проводилась, и список UTXO, участвовать в обслуживании всей сети Bitcoin, а также обслуживать легкие SPV-узлы, отвечая на их запросы.

---

[7] На русском языке нам привычнее говорить о длине цепочки, но в англоязычной литературе исторически сформировался термин *высота блокчейна* (blockchain height).

Заметим, что существует много разновидностей программного обеспечения Bitcoin для полных узлов, которые существенно различаются по архитектуре и написаны на разных языках программирования. Однако наиболее распространенным является программное обеспечение Bitcoin Core, которое использует более трех четвертей узлов сети.

## 3.7.2. Упрощенная проверка транзакций

В системе Bitcoin есть замечательная концепция легких SPV-узлов, которые можно задействовать для проверки транзакций без использования полных узлов. Принцип работы SPV-узлов заключается в том, что они загружают только заголовки всех блоков во время начальной синхронизации с сетью Bitcoin. Заголовки блоков Bitcoin занимают 80 байтов каждый, а общий объем всех заголовков составляет всего несколько мегабайт.

Легкие SPV-узлы предоставляют механизм проверки того, что конкретная транзакция была в блоке, не требуя полных данных блокчейна. Каждый заголовок блока содержит корень Меркла, который является хэшем блока. Мы знаем, что хэш каждой транзакции связан с хэшем блока через дерево Меркла, о котором мы говорили в *главе 2*. Все транзакции в блоке образуют листья дерева Меркла, а хэш блока представляет собой корень Меркла. Прелесть дерева Меркла в том, что для доказательства того, что транзакция действительно является частью блока, нужна только небольшая часть блока. Для подтверждения транзакции SPV-узел делает две вещи. Во-первых, узел проверяет доказательство дерева Меркла для транзакции, чтобы убедиться, что она является частью блока. Во-вторых, узел проверяет, является ли этот блок частью действующей ветви блокчейна, — после него должно быть создано как минимум еще шесть блоков, чтобы гарантировать, что он является частью самой длинной цепочки. Процесс проверки схематически изображен на рис. 3.26.

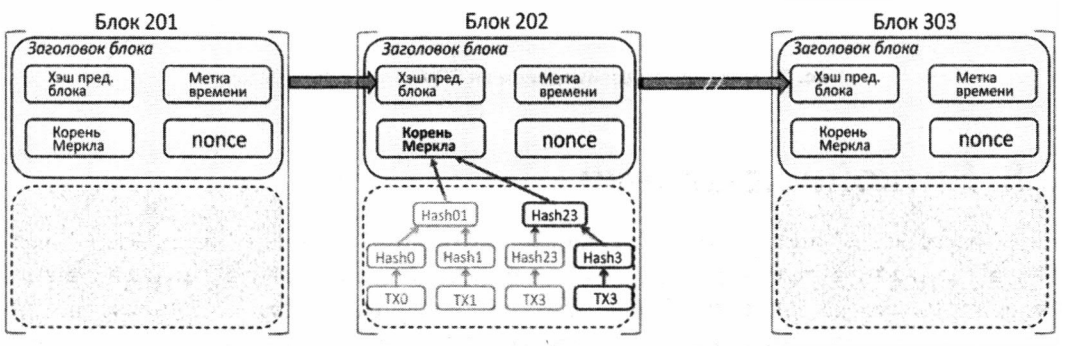



**Рис. 3.26. Корень Меркла в заголовке блока и упрощенная проверка**

Давайте в общих чертах рассмотрим алгоритм, по которому работают SPV-узлы при проверке транзакций:

- SPV-узел устанавливает фильтры Блума для узлов, к которым он подключен. Цель фильтров Блума состоит в том, чтобы пропускать только те транзакции,

которые действительно нужны, а не весь блок, но не раскрывать при этом адреса или ключи, которыми интересуется SPV-узел. В идеале необходимо установить обмен с множеством узлов, потому что единственный узел может попытаться обмануть или отказать в обслуживании;

- в ответ на запрос SPV-узла другие узлы отправляют обратно соответствующие транзакции в сообщении `merkleblock`, которое содержит корень Меркла и путь Меркла к интересующей транзакции. Сообщение `merkleblock` занимает всего несколько килобайт и довольно эффективно с точки зрения нагрузки на сеть;
- теперь SPV-узел может легко проверить, действительно ли транзакция принадлежит определенному блоку в блокчейне;
- после подтверждения транзакции следующим шагом является проверка того, является ли данный блок частью самой длинной ветви блокчейна.

На рис. 3.27 показано, как SPV-узел общается со своими коллегами.

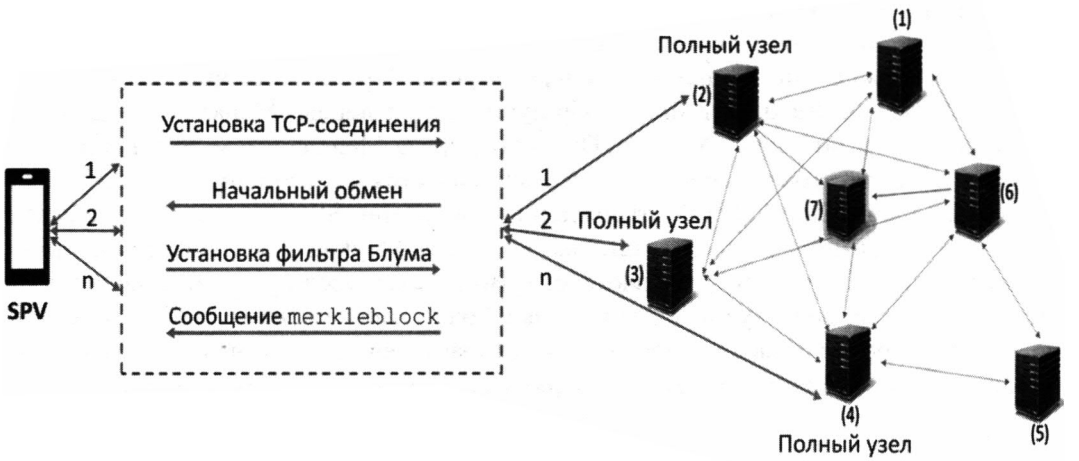



**Рис. 3.27. Механизм взаимодействия SPV-узлов в сети Bitcoin**

## 3.8. Биткойн-кошельки

Биткойн-кошелек очень похож на кошелек, который вы используете в своей повседневной жизни, — в том смысле, что вы имеете к нему доступ и можете тратить содержимое, когда захотите. Однако биткойн-кошельки — это особенный цифровой феномен. Вспомните пример из предыдущего раздела, когда Алиса заплатила Бобу некоторую сумму. Как она смогла это сделать, если у Боба нет счета в привычном понимании? В системе Bitcoin счета, или кошельки представлены в виде адреса. Боб должен сначала сгенерировать пару ключей (закрытый и открытый ключ). Bitcoin использует алгоритм ECDSA с кривой secp256k1 (не пугайтесь, это просто обозначение кривой, рекомендованной стандартом). Сначала генерируется случайная битовая строка, которая служит закрытым ключом. Затем закрытый

ключ детерминированно преобразуется в открытый ключ. Как мы узнали ранее в *главе 2*, закрытые и открытые ключи математически связаны, и открытый ключ может быть сгенерирован из закрытого ключа в любое время с одинаковым результатом (детерминированный результат). Таким образом, на самом деле нет необходимости сохранять открытые ключи как таковые. В программной реализации генератора случайных чисел невозможна истинно случайная последовательность, поэтому многие серверы или приложения используют аппаратные модули безопасности (Hardware Security Module, HSM) для генерации истинно случайных чисел, а также для защиты закрытых ключей. В отличие от открытых ключей, закрытые ключи, безусловно, требуют хранения с максимальной аккуратностью. Если вы их потеряете, вы не сможете сгенерировать новую подпись, которая оправдывала бы владение открытым ключом или биткойн-адресом, который получил входящие транзакции. Для генерации биткойн-адреса открытые ключи хэшируются дважды, сначала по алгоритму SHA-256, а затем по алгоритму RIPEMD-160. Это также детерминированные операции, поэтому при наличии открытого ключа создание адреса Bitcoin — это всего лишь вопрос вычисления пары хэшей.

Обратите внимание, что хотя адрес Bitcoin получается из открытого ключа, на самом деле он не раскрывает открытый ключ. Это связано с тем, что открытые ключи подвергаются двойному хэшированию, и восстановить открытый ключ по адресу Bitcoin совершенно невозможно. Однако обладатель открытого ключа легко докажет владение адресом. Техника хэширования укорачивает и запутывает открытый ключ. Это не только облегчает ручной ввод, но также обеспечивает защиту от непредвиденных уязвимостей, которые могли бы позволить восстановить закрытый ключ из открытого ключа. Это, возможно, самая безопасная реализация системы с двумя ключами! Как показано на рис. 3.28, открытый ключ предоставляется только тогда, когда получатель запрашивает выходные данные транзакции, а не тогда, когда транзакция была передана ему.

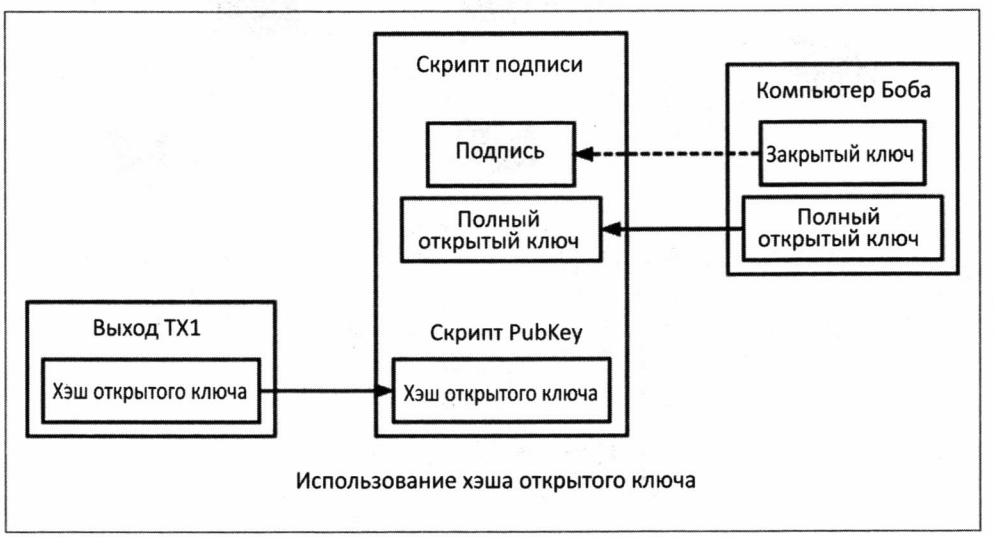



**Рис. 3.28. Предоставление открытого ключа для запроса входящей транзакции**

Биткойн-кошельки — это не что иное, как SPV-узлы, и их обслуживают полные узлы, предоставляющие данные о блоках и транзакциях в ответ на запрос. Мы познакомились с функционированием SPV-узла в *разд. 3.7.2*, а здесь рассмотрим некоторые операции, связанные с кошельком. Теперь мы понимаем, что для отправки или получения транзакции нет необходимости запускать полный узел. Все, что вам нужно, — это кошелек, с помощью которого вы сможете хранить вашу пару закрытых/открытых ключей, чтобы иметь возможность совершать и принимать транзакции. Вы уже знаете, как происходит проверка существующих транзакций. Давайте теперь посмотрим, как инициировать транзакцию с помощью кошелька.

Разумеется, было бы хорошо развернуть свой собственный полный узел и подключить к нему кошелек, потому что это наиболее безопасный способ работы с биткойнами. Однако это не исключительное требование, и вы все равно можете работать с биткойнами, не владея полным узлом. В чем же преимущество владения полным узлом? Допустим, у вас запущен только облегченный SPV-узел. Когда вы хотите получить список UTXO и запрашиваете информацию у полных узлов, вы должны указать свой публичный адрес. Следовательно, все полные узлы, получившие запрос, узнают ваш публичный адрес, а это является утратой конфиденциальности! Все, что нужно сделать кошельку, — это получить список UTXO, чтобы он мог провести транзакцию, подписав ее своим закрытым ключом, и опубликовать эту транзакцию в сети Bitcoin. Это можно сделать, создав собственное программное обеспечение кошелька или используя сторонний сервис кошелька. Однако будьте

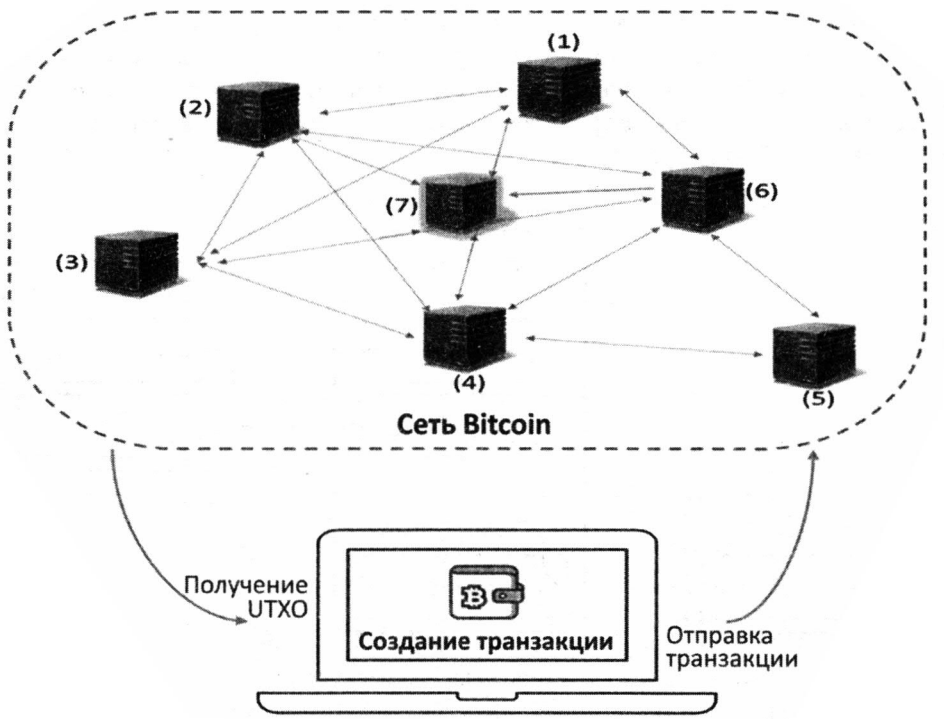



**Рис. 3.29. Приложение кошелька, взаимодействующее с сетью Bitcoin**

осторожны с поставщиками сервисов кошелька, потому что вы позволяете им контролировать ваш закрытый ключ! Независимо от того, намеренно ли они похищают ваши биткойны или были взломаны, как это часто случалось, вы рискуете потерять свои биткойны навсегда. В конце концов, все поставщики сервисов кошелька централизованы, хотя и подключаются к децентрализованной сети Bitcoin. Типичный процесс создания транзакции Bitcoin через программное обеспечение кошелька показан на рис. 3.29.

Примером реализации SPV-узла, который может служить кошельком, является BitcoinJ. На самом деле BitcoinJ — это библиотека для работы с протоколом Bitcoin, ведения кошелька и инициирования/проверки транзакций. Она не требует развертывания полного узла, такого как узел Bitcoin Core, и может функционировать как легкий узел. Хотя библиотека реализована на Java, ее можно использовать с любым JVM-совместимым языком — таким как JavaScript и Python.

## 3.9. Заключение

В этой главе мы увидели, как объединение различных концепций блокчейна позволило создать эффективную и безопасную систему криптовалютных транзакций. Мы рассмотрели эволюцию и особенности устройства сети Bitcoin. Мы узнали о нюансах функционирования Bitcoin, транзакциях, блоках, блокчейне, консенсусе и о том, как все это собрано в единую систему. Затем мы разобрались, как приложение кошелька взаимодействует с блокчейном Bitcoin.

В 1990-х годах массовое внедрение Интернета изменило способы ведения бизнеса и устранило препятствия для создания и распространения информации. Точно так же блокчейн обещает вывести Интернет на совершенно новый уровень. Bitcoin — это всего лишь одно из возможных приложений блокчейна. На самом деле варианты применения блокчейна и его возможности безграничны. В следующей главе мы узнаем о том, как работает блокчейн Ethereum и как он стал стандартом де-факто для различных децентрализованных приложений в публичной сети.

## 3.10. Рекомендуемые источники

- Bitcoin: пиринговая электронная платежная система: Nakamoto, Satoshi, «Bitcoin: A Peer-to-Peer Electronic Cash System», **https://bitcoin.org/bitcoin.pdf**.
- Все о сети Bitcoin и транзакциях: Bitcoin wiki, **https://en.bitcoin.it/**.
- Технология блокчейна: Crosby, Michael, Nachiappan; Pattanayak, Pradhan, Verma, Sanjeev, Kalyanaraman, Vignesh, «BlockChain Technology: Beyond Bitcoin», Sutardja Center for Entrepreneurship & Technology, University of California, Berkeley, **http://scet.berkeley.edu/wp-content/uploads/BlockchainPaper.pdf**, October 16, 2015.
- Ускорение обработки транзакций в сети Bitcoin: Sompolinsky, Yonatan, Zohar, Aviv, «Secure High-Rate Transaction Processing inBitcoin», Hebrew University of Jerusalem, Israel, School of Engineering and Computer Science, **https://eprint.iacr.org/2013/881.pdf**.

ГЛАВА 4

# Как работает Ethereum?

Эра блокчейн-приложений только начинается. Вы наверняка слышали про блокчейн Ethereum, который фактически стал общепринятой платформой для создания децентрализованных приложений. В предыдущих главах мы уже узнали, что блокчейн применяется не только для криптовалюты, а возможности приложений ограничены лишь вашим воображением! Ethereum завоевал популярность во многих секторах бизнеса и прекрасно работает как в публичных, так и в частных приложениях. Нам следует внимательно изучить платформу Ethereum, чтобы понимать, как создаются и работают децентрализованные приложения. Сегодня благодаря Ethereum можно создавать блокчейн-приложения с минимальными навыками в области криптографии, теории игр, математики и программирования.

В *главе 3* мы определили, что протокол Bitcoin неразрывно связан с понятием криптовалюты. Теперь вы знаете, что Bitcoin — это не приложение, работающее поверх блокчейна, а специально разработанный блокчейн. В этой главе мы расскажем, как Ethereum формирует абстрактный базовый уровень, позволяющий запускать различные приложения на платформе одного и того же блокчейна.

## 4.1. От Bitcoin до Ethereum

Вспомним, что технология блокчейна пришла вместе с биткойнами еще в 2009 году. После того как биткойн выдержал испытание временем, люди поверили в потенциал блокчейна. Использование блокчейна вышло за пределы банковского и финансового сектора и охватило другие отрасли, такие как логистика, розничная торговля, электронная коммерция, здравоохранение, энергетика и правительственный сектор. Для этих целей разрабатывают специальные варианты блокчейна, которые решают конкретные задачи. С другой стороны, существуют открытые блокчейн-платформы, такие как Ethereum, которые позволяют создавать совершенно

разные децентрализованные приложения на основе одного общедоступного блокчейна.

Блокчейн Bitcoin предназначался для децентрализованных одноранговых транзакций криптовалюты. Однако вскоре стало очевидно, что блокчейн можно использовать для перемещения и отслеживания любых ценностей, а не только криптовалюты. В качестве примера можно назвать *доказательство существования* — это один из случаев альтернативного использования блокчейна, когда хэш документа внедряют в блок Bitcoin, чтобы впоследствии кто угодно мог проверить, существовал ли такой документ в определенный момент времени. Виталик Бутерин разработал платформу блокчейна Ethereum, которая в равной мере позволяет работать не только с деньгами, но и с акциями, земельными участками, цифровым контентом, транспортными средствами и многими другими объектами, которые имеют собственную ценность.

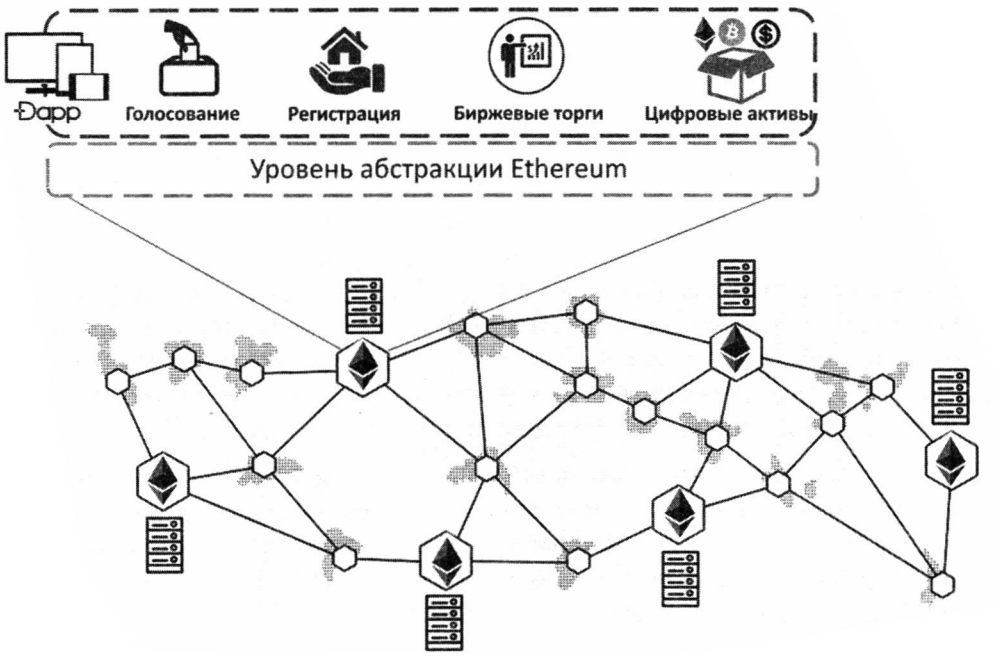



**Рис. 4.1.** Несколько децентрализованных приложений на одной платформе Ethereum

Как и Bitcoin, Ethereum является открытым блокчейном, однако с другим философским подходом к устройству сети. Инновационная сущность этого подхода заключается в создании *уровня абстракции*, чтобы транзакции из разных приложений обобщались в программный код, который может выполняться на всех узлах Ethereum (рис. 4.1). Впрочем, даже в Ethereum майнеры генерируют *эфир* (ether) — свободно обращаемую криптовалюту, благодаря которой общедоступная сеть является самодостаточной. Любое приложение, работающее в Ethereum, должно платить комиссионные за транзакции, которые в конечном итоге получают майнеры за работу узлов и поддержку всей сети.

### 4.1.1. Ethereum как блокчейн нового поколения

По мере роста популярности протокола Bitcoin разработчики стали создавать различные децентрализованные приложения с совершенно новыми блокчейнами или пытались модифицировать Bitcoin Core для увеличения набора функций. В любом случае это было сложно, а также отнимало много времени. На помощь разработчикам пришла альтернативная платформа Ethereum. Идея заключается в том, чтобы разрабатывать различные блокчейн-приложения на единой платформе, а не создавать с нуля блокчейны для каждого приложения в отдельности. Ethereum позволяет быстро создавать децентрализованные приложения, которые могут взаимодействовать между собой, и обеспечивает достаточную безопасность. Как упоминалось в предыдущем разделе, Ethereum реализует эти возможности за счет добавления абстрактного базового уровня. В отличие от Bitcoin, платформа Ethereum поддерживает язык, полный по Тьюрингу, так что любой желающий может писать умные контракты, которые способны делать практически все, что угодно, с точки зрения программирования. Кроме того, Ethereum отслеживает текущее состояние счетов, чем сильно отличается от Bitcoin, где все основано на транзакциях и нет внутренней постоянной памяти для скриптов. С помощью абстрактного базового уровня основные сложности реализации скрыты от разработчиков, но выгода не только в этом — разработчики обладают гибкостью при разработке собственных функций перехода состояния для прямой передачи ценности и информации, а также собственных форматов транзакций.

Основным техническим нововведением Ethereum является *виртуальная машина Ethereum* (Ethereum Virtual Machine, EVM). Поддержка Тьюринг-полных языков позволяет разработчикам легко создавать приложения блокчейна. Так же, как для выполнения Java-кода требуется виртуальная машина Java (Java Virtual Machine, JVM), для выполнения умных контрактов требуется EVM. Сейчас мы просто кратко заметим, что *умные контракты* — это написанные на полном по Тьюрингу языке скрипты Ethereum, которые автоматически выполняются в случае возникновения заранее определенного события. Уже знакомые нам `ScriptSig` и `ScriptPubKey` в Bitcoin — это, так сказать, базовые версии умных контрактов. В предыдущей главе мы узнали, что в Bitcoin набор инструкций скриптов очень ограничен. В Ethereum, однако, можно написать практически любую программу, которая будет выполняться виртуальной машиной на каждом узле сети блокчейна Ethereum. Децентрализованные приложения для платформы Ethereum принято обозначать сокращением *DApp* (Decentralized Applications). В глобальной компьютерной системе Ethereum без центрального сервера приложения DApp работают без простоев, мошенничества или навязанных кем-либо правил. Одноранговую систему электронных денег — такую как Bitcoin — можно легко построить на платформе Ethereum в виде приложения DApp. Точно так же любой другой актив, обладающий внутренней ценностью, — например, земля, автомобили, дома, голоса избирателей — может легко передаваться в виде токенов через соответствующие приложения Ethereum.

В отличие от традиционной разработки и развертывания программного обеспечения, приложения DApp не нужно размещать на внутреннем сервере. Код приложе-

ния внедряется в качестве полезной нагрузки в транзакции, которые затем отправляются на майнинговые узлы в сети Ethereum. Такие транзакции будут обрабатываться экосистемой майнинга, потому что майнеры получают вознаграждение в виде оплаты за *газ* — комиссии, которая заложена в транзакцию. Как и в сети Bitcoin, эти транзакции транслируются другим майнерам в сети. После нахождения консенсуса транзакция в конечном итоге попадает в блок и становится вечной частью блокчейна. Разработчики могут свободно кодировать любое приложение и развертывать его в сети Ethereum. Сеть сама выполняет код, а также проверяет транзакции и выдает выходные данные. Но если бы этот механизм работал без затрат, сеть мгновенно рухнула бы под валом мусорных транзакций. С каждой транзакцией блокчейна связаны расходы на газ, поэтому написание мусорного кода и его размещение в сети Ethereum становится очень затратным делом.

### 4.1.2. Философия блокчейна Ethereum

Блокчейн Ethereum позаимствовал многие концепции из Bitcoin Core, поскольку он выдержал испытание временем, но разработан на основе другой философии. Разработка Ethereum осуществлялась по следующим принципам:

- **простота** — блокчейн Ethereum спроектирован максимально простым, чтобы его было легко изучить и использовать как платформу для децентрализованных приложений. Сложности в реализации сведены к соблюдению минимальных требований на уровне консенсуса. Компиляция кода на языке высокого уровня или сериализация аргументов не являются проблемой для разработчиков приложений Ethereum;
- **свобода разработки** — платформа Ethereum предназначена для поощрения любой децентрализации на основе блокчейна, не ограничивает и не поддерживает какие-либо конкретные варианты использования. Эта свобода безгранична — вплоть до того, что разработчик может закодировать бесконечный цикл в умном контракте и запустить его. Очевидно, что цикл будет работать до тех пор, пока владелец контракта платит комиссию за транзакцию (так называемый *газ*), и цикл в конце концов завершится, когда газ на счете закончится;
- **отсутствие понятия о наборе функций** — чтобы сделать систему максимально абстрактной, Ethereum не имеет встроенных функций для разработчиков. Вместо этого Ethereum обеспечивает поддержку Тьюринг-полного языка и позволяет пользователям разрабатывать свои собственные функции так, как они хотят. В Ethereum можно запрограммировать все, что угодно, начиная с базовых функций и заканчивая полноценными сценариями обработки транзакций.

## 4.2. Введение в блокчейн Ethereum

Мы обсудили назначение блокчейна Ethereum и его внутреннюю философию. Чтобы понять устройство блокчейна нового поколения и строить на нем децентрализованные приложения, следует подробно рассмотреть основные компоненты Ethereum.

## 4.2.1. Структура данных блокчейна Ethereum

Структура данных блокчейна Ethereum очень похожа на структуру данных блокчейна Bitcoin, за исключением того, что в заголовке блока Ethereum содержится гораздо больше информации. В этом разделе мы сосредоточимся на структуре данных и заголовке, а так называемые *состояния* Ethereum обсудим в следующих разделах. В заголовке блока Bitcoin размещен только один корень Меркла для всех транзакций в блоке. В блок Ethereum добавлены еще два корня Меркла — итого в его заголовке присутствуют три корня Меркла:

- *stateRoot* — помогает хранить глобальное состояние;
- *TransactionsRoot* — предназначен для отслеживания и обеспечения целостности всех транзакций в блоке аналогично корню Меркла в Bitcoin;
- *receiptsRoot* — это корневой хэш дерева квитанций, соответствующих транзакциям в блоке.

Мы еще вернемся к этим корням Меркла при изучении назначения полей заголовка. А сейчас взгляните на рис. 4.2. Каждый блок обычно содержит заголовок блока, список транзакций, список блоков-«дядюшек» (оммеров) и необязательные дополнительные данные. Имейте в виду, что названия полей и порядок их перечисления могут различаться в разных книгах и статьях. Мы рекомендуем вам тщательно разобраться с назначением этих полей, чтобы различия в терминах не вводили вас в замешательство.

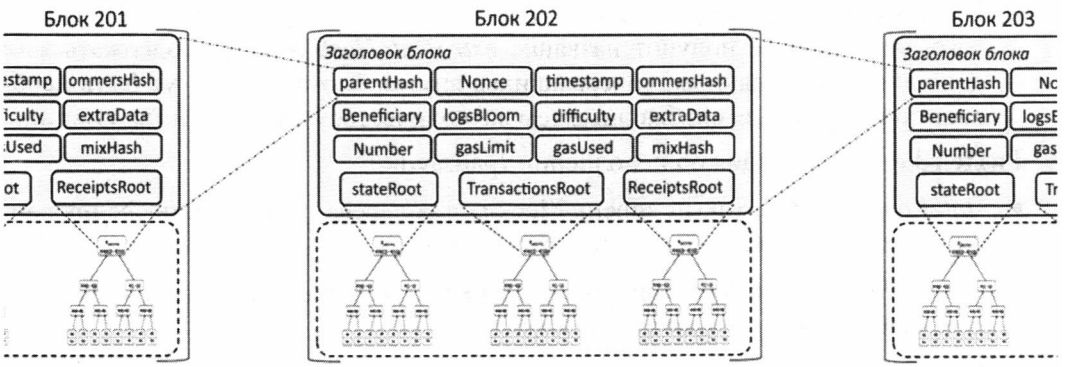



**Рис. 4.2.** Структура блока в блокчейне Ethereum

- **Раздел 1** — метаданные блока:
  - *parentHash* — 256-битовый хэш-код заголовка родительского блока, как в Bitcoin;
  - *timestamp* — метка времени UNIX для текущего блока;
  - *Number* — номер текущего блока;
  - *Beneficiary* — 160-битовый адрес майнера, на который перечисляется вознаграждение за майнинг этого блока.

- **Раздел 2** — ссылки на данные:
  - *TransactionsRoot* — 256-битовый корень Меркла для дерева транзакций, размещенных в этом блоке;
  - *ommersHash* — этот хэш также известен как *uncleHash*. Это хэш списка оммеров данного блока;

---

**ПРИМЕЧАНИЕ НАУЧНОГО РЕДАКТОРА ПЕРЕВОДА.** *Оммер* (ommer) — это блок, чей родительский блок тождественен родительскому блоку родительского блока текущего блока. Поэтому такой блок иногда называют *дядей* текущего блока (uncle block). Дело в том, что в Ethereum интервал между блоками значительно меньше (~15 секунд), чем в Bitcoin (~10 минут). Это ускоряет прохождение транзакций, однако майнеры одновременно находят много конкурирующих блоков. Если майнеры не будут получать вознаграждение за такие блоки, то они потеряют интерес к майнингу, и сеть умрет. Оммеры нужны, чтобы майнеры могли получать вознаграждение за правильные, но отброшенные блоки, не попавшие в цепочку. Оммеры должны быть сформированы не более чем в шестом поколении от текущего блока. За подтверждение оммеров майнеры получают меньшее вознаграждение, чем за обычный блок. Тем не менее это достаточный стимул для майнинга. Забавно, что ommer — это немецкое слово, которое означает округлую плоскую фишку (бабку) для игры в «камушки». По сути, майнер получает «бабки» за выбитые из цепочки бабки.

---

  - *extraData* — произвольный байтовый массив, содержащий дополнительные данные, относящиеся к текущему блоку. Размер этих данных ограничен 32 байтами (256 битов). На момент подготовки книги существует вероятность того, что это поле получит название *extraDataHash* и будет содержать хэш дополнительных данных, содержащихся внутри блока. Это могут быть необработанные данные, снабженные тем же количеством газа, что и транзакции.
- **Раздел 3** — информация об исполнении транзакции:
  - *stateRoot* — 256-битовый корень Меркла для дерева состояний после проверки и выполнения всех транзакций этого блока;
  - *receiptsRoot* — 256-битовый корень Меркла для дерева квитанций от получателей каждой транзакции в этом блоке;
  - *logBloom* — накопительный фильтр Блума, т. е. логическое «ИЛИ» фильтров Блума для каждой транзакции в блоке;
  - *gasUsed* — общее количество газа, использованного на выполнение транзакций в этом блоке;
  - *gasLimit* — максимальное количество газа, которое может использовать этот блок (динамическое значение в зависимости от активности в сети).
- **Раздел 4** — информация о подсистеме консенсуса:
  - *difficulty* — уровень сложности для этого блока, рассчитанный на основе сложности и метки времени предыдущего блока;

- *mixHash* — 256-битовый хэш, который в сочетании с *nonce* служит подтверждением того, что для данного блока было выполнено достаточное количество вычислительных операций;
- *nonce* — 64-битовый хэш, который комбинируется с *mixHash* и может использоваться как проверка PoW.

## 4.2.2. Счета Ethereum

Счета Ethereum, в отличие от Bitcoin, не являются отображением неизрасходованных транзакций (UTXO). В *главах 2* и *3* мы узнали, что биткойны на самом деле хранятся в форме транзакций, которые имеют владельца и сумму. Владелец может потратить входящую транзакцию, если у него есть действительный закрытый ключ для транзакции, которую он пытается потратить. Bitcoin, следовательно, является системой *перехода состояний*, где *состояние* относится к совокупности всех UTXO системы. Каждый раз, когда добывается блок, происходит изменение состояния, потому что каждый блок содержит набор транзакций, где каждая транзакция тратит входящие UTXO и производит новые UTXO. Обратите внимание, что текущее состояние не закодировано внутри блоков в явном виде. Таким образом, в системе Bitcoin отсутствует понятие баланса счета. Ethereum, наоборот, представляет собой систему *хранения состояний*, и его базовой единицей является *счет* (учетная запись, адрес). С каждым счетом связано его состояние, а также 20-байтовый (160-битовый) адрес. Цель блокчейна в Ethereum — хранить изменения состояния счетов. Существует два типа счетов Ethereum:

- *внешний счет*, или *счет внешнего владельца* (Externally Owned Account, EOA). Эти счета также называются *простыми счетами*. Обычно они принадлежат пользователям или устройствам, которые управляют счетами с помощью закрытых ключей. EOA могут отправлять транзакции на другие EOA или счета контрактов, подписывая их закрытым ключом. Назначение транзакции между двумя EOA обычно заключается в передаче некой формы стоимости. С другой стороны, когда EOA совершает транзакцию на счет контракта, назначение транзакции состоит в том, чтобы активировать код контракта;
- *счет контракта* — такие счета управляются только кодом контракта. Этот код, привязанный к счету, называется *умным контрактом*, или *смарт-контрактом* (smart contract). Обычно контракты активируются, когда получают транзакцию от EOA или другого контракта. Несмотря на то, что контракты способны выполнять сложные процедуры с помощью кода, который они содержат, они не могут инициировать новые транзакции самостоятельно и всегда зависят от EOA. Все, что они могут сделать, — это ответить на входящие транзакции согласно логике своего программного кода.

Взгляните на следующие три сценария (рис. 4.3–4.5), чтобы лучше понять связь между EOA и счетами контрактов.

Чтобы правильно понять суть изображенных на рис. 4.3–4.5 операций, имейте в виду, что счета контрактов являются *внутренними*, и связь между ними также

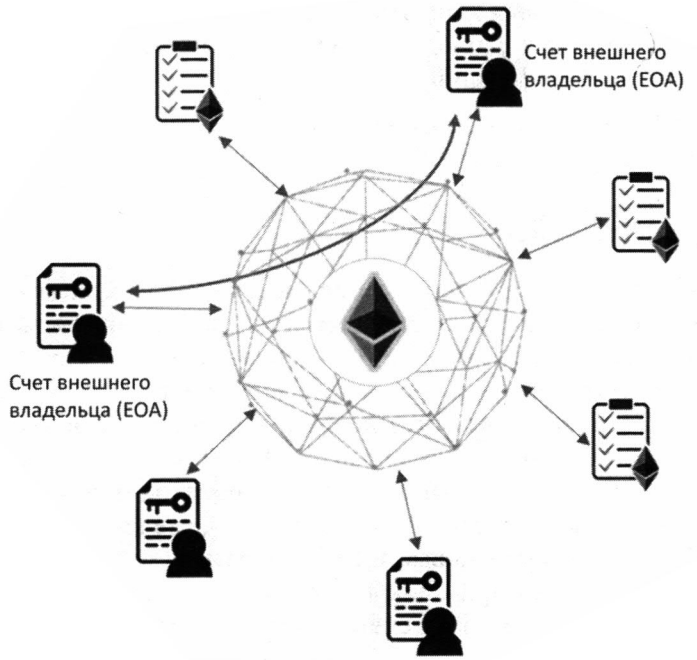



**Рис. 4.3.** Транзакция между внешними счетами: EOA — EOA

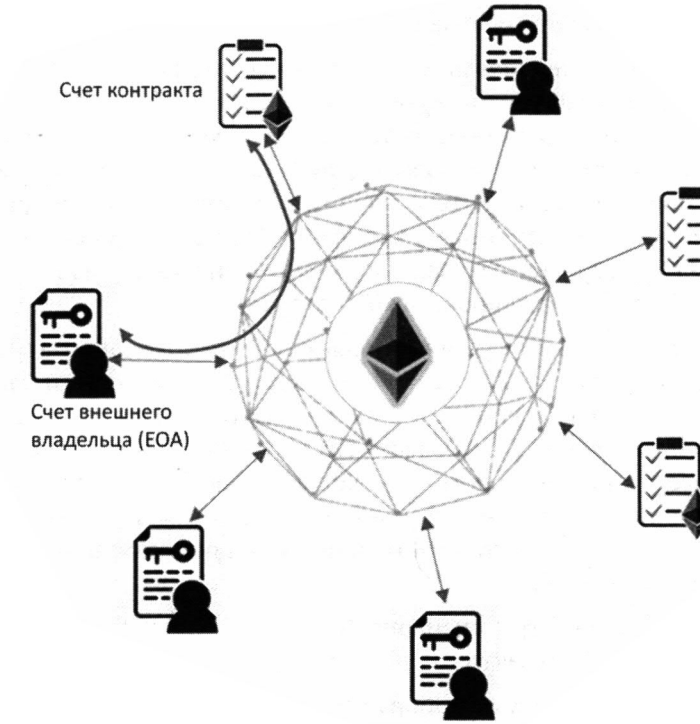



**Рис. 4.4.** Транзакция: EOA — счет контракта

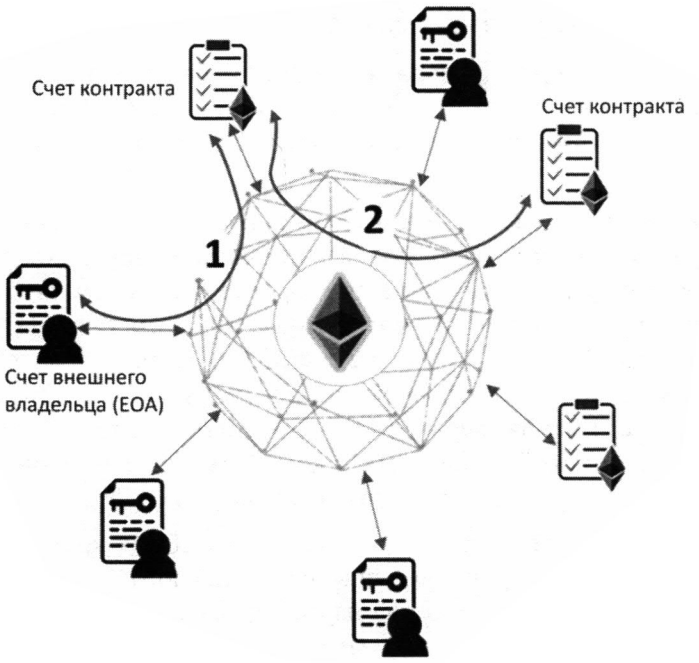



**Рис. 4.5.** Транзакция: EOA — счет контракта — счет контракта

не выходит за пределы среды выполнения контрактов, в отличие от внешних счетов EOA, которые инициируют транзакции и вводят их в блокчейн извне.

## Преимущества концепции UTXO

Одна из ключевых концепций Bitcoin состоит в том, чтобы поддерживать максимальную анонимность. Когда мы сравниваем Bitcoin с Ethereum, то находим следующие важные преимущества UTXO:

- **лучшая конфиденциальность** — в сети Bitcoin рекомендуется использовать новый адрес при получении транзакций, что укрепляет анонимность. Даже с помощью сложных статистических методов или машинного обучения трудно сопоставить разные адреса и их владельца (хотя и не совсем невозможно);
- **потенциально лучшая масштабируемость** — рассуждения о масштабировании обычно очень субъективны и зависят от контекста использования блокчейна, прикладных задач и прочих факторов. Мы сейчас хотим просто отметить присущий UTXO потенциал масштабирования. В системе на основе UTXO можно выполнять транзакции параллельно. Кроме того, когда владелец кошелька или другие узлы, хранящие сведения о владении активами, теряют эти данные, это касается только владельца и только в пределах суммы неподтвержденных UTXO. Напротив, если потеряны данные дерева Меркла для некоторого счета Ethereum, то становится невозможной любая операция с этим счетом, даже зачисление средств.

## Преимущества концепции счетов

Хотя Ethereum в некотором роде является расширением Bitcoin, она представляет собой совершенно новую систему со своим собственным набором плюсов и минусов. Давайте рассмотрим преимущества счетов Ethereum по сравнению с системой UTXO Bitcoin:

- **значительная экономия хранилища** — допустим, в сети Bitcoin вам надо отправить транзакцию на 5 BTC, но у вас нет единственной входящей транзакции на достаточную сумму. Вам придется собрать вместе несколько входящих транзакций, чтобы такую сумму набрать. Кроме того, эти транзакции могли поступать на принадлежащие вам разные адреса! В то же время в системе счетов Ethereum достаточно единственной ссылки на счет. Хотя Ethereum использует улучшенную реализацию дерева Меркла — Merkle-Patricia Tree (MPT) — которая занимает немного больше места, в конечном итоге вы экономите значительное количество места на сложных транзакциях;
- **простота программирования** — используя UTXO и скрипты, которые не являются полными по Тьюрингу, трудно создавать сложные системы. UTXO могут быть либо израсходованы, либо не израсходованы — для них нет промежуточного состояния. Это ограничение затрудняет кодирование сложных процедур. Даже если расширить набор команд, программирование для UTXO все равно будет сложнее по сравнению с использованием простых балансовых счетов. Назначение Ethereum состоит в том, чтобы выйти за рамки криптовалюты и выполнять произвольные децентрализованные приложения для обмена любыми ценностями, — следовательно, невозможно обойтись без счетов, обладающих конечным состоянием;
- **простые ссылки для клиентов** — в отличие от клиентских приложений Bitcoin, клиентские приложения Ethereum могут легко и быстро получить доступ к истории счета, просто сканируя дерево состояний в определенном направлении. В модели UTXO обычно присутствует несколько ссылок на несколько транзакций, связанных с какой-либо конкретной рассматриваемой транзакцией.

## Состояние счета

Мы уже говорили, что каждый счет имеет текущее состояние, а также рассмотрели два варианта счетов в системе Ethereum: внутренние счета контрактов и счета внешних владельцев (EOA, или внешние счета). Вне зависимости от типа счета, его состояние можно отследить через поле *stateRoot* в заголовке блока (рис. 4.6).

Как вы можете видеть на рис. 4.6, независимо от разновидности счета, он имеет следующие четыре компонента:

- *Balance* — это общий баланс эфира (ETH) на счете. Точнее, сумма денежных единиц Wei, принадлежащих данному адресу (1 ETH = $10^{18}$ Wei);
- *CodeHash* — это хэш кода контракта. В каждом счете умного контракта есть код, который выполняется на виртуальной машине Ethereum. Хэш этого кода

хранится в поле *CodeHash*. Однако для счетов EOA код отсутствует, поэтому поле *CodeHash* содержит хэш пустой строки;

- *StorageRoot* — это 256-битовый корневой хэш дерева Меркла, в котором закодировано содержимое хранилища счета. Наличие корневого хэша этого дерева в поле *StorageRoot* помогает отслеживать содержимое счета, а также обеспечивает его целостность;
- *Nonce* — это счетчик, который обеспечивает обработку каждой транзакции только один раз. Для счетов EOA это число представляет количество исходящих транзакций с адреса счета. Для счетов умных контрактов *Nonce* представляет количество контрактов, созданных этим счетом.

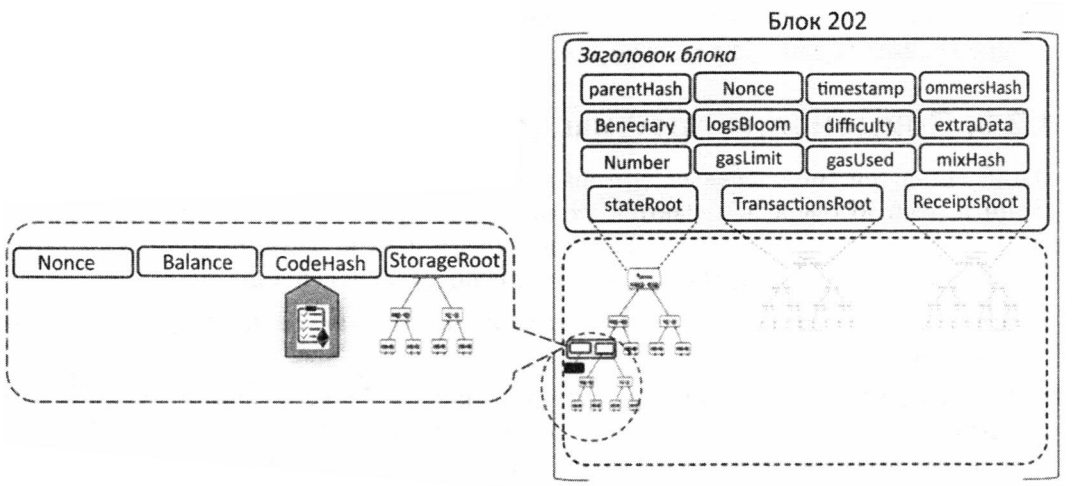



**Рис. 4.6. Детальное представление состояния счета**

Таким образом, именно состояние *префиксного дерева* (trie-дерева)[1] отвечает за отслеживание состояния блокчейна Ethereum. Однако хитрость заключается в том, что состояние не сохраняется напрямую в каждом блоке, а хранится в форме данных состояния, кодируемых в дереве Меркла с помощью *рекурсивного префикса длины* (Recursive Length Prefix, RLP) на каждом узле Ethereum. Таким образом, чтобы отслеживать глобальное состояние системы, блокчейн Ethereum добавляет *корни состояний* в каждый блок, где хранится корневой хэш дерева хэшей (корень Меркла), отражающий состояние системы на момент создания блока.

Согласно «Желтой книге»[2] Ethereum, *глобальное состояние* является отображением между адресами (160-битовыми идентификаторами) и состояниями счетов. Следо-

[1] Trie-дерево (дерево лучевого поиска) — модифицированное дерево Меркла для системы состояния счетов Ethereum. Более подробно на русском языке о trie-деревьях можно почитать в статье по адресу: **https://habr.com/ru/post/111874/**.

[2] «Желтая книга» содержит детальное описание системы и криптовалюты Ethereum. Документ написал и опубликовал в 2013 году Гэвин Вуд, который участвовал в создании Ethereum вместе с Виталиком Бутериным.

вательно, глобальное состояние содержит информацию обо всех счетах в блокчейне, но не хранится в каждом блоке. Каждый блок влияет только на часть состояния. В некотором смысле, глобальное состояние формируется при обработке каждого блока, начиная с блока генезиса. Впрочем, некоторые узлы Ethereum могут хранить всю историю состояний счетов, сохраняя все исторические транзакции, т. е. переходы состояний и их выходные данные. Это позволяет клиентам запрашивать состояние блокчейна в любое время, даже для исторических транзакций, без необходимости вычислять историю с самого начала. Получение информации о состоянии счетов аналогично агрегатному запросу SQL, где данные легкодоступны, и требуется только агрегация. Таким образом, старые данные о состоянии могут быть легко отброшены (это называется *обрезкой*), потому что они могут быть вычислены обратно при необходимости. Что ж, так и было задумано — данные о состоянии являются неявными данными, т. е. информация о состоянии должна рассчитываться, а не храниться в полном виде на каждом узле.

## 4.2.3. Применение префиксного trie-дерева

Мы изучили три типа деревьев, которые имеют свои корни в заголовке блока. Эти корни в первую очередь являются указателями на соответствующие префиксные деревья. Хотя мы кратко упоминали эти деревья и их корни в *разд. 4.2.1*, давайте вернемся собственно к деревьям и добавим некоторые пояснения:

- *State trie* — представляет полное состояние (глобальное состояние) после доступа к блоку;
- *Transaction trie* — представляет все транзакции в блоке, снабженном индексом (вида `key:0` для первой транзакции, которую нужно выполнить, `key:1` для второй транзакции и т. д.). Вспомните основы деревьев Меркла, которые мы рассмотрели ранее (см. *разд. 2.4.2* и *3.3.1*), и попытайтесь найти взаимосвязь с ранее изученным материалом;
- *Receipt trie* — представляет квитанции, соответствующие каждой транзакции. Квитанция для транзакции является структурой данных в кодировке RLP:

  ```
  [medstate, gas_used, logbloom, logs]
  ```

  Давайте теперь более детально разберем структуру данных *Receipt trie*, поскольку мы еще не рассмотрели основы этого вопроса. Назначение полей в структуре данных *Receipt trie* выглядит следующим образом:

  - *medstate* — это корень дерева *State trie* после обработки транзакции. Успешная транзакция обновляет состояние Ethereum;
  - *gas_used* — это общее количество газа, использованного для обработки транзакции;
  - *logs* — это список элементов вида:

    ```
    [address, [topic1, topic2...], data]
    ```

    Эти элементы списка создаются кодами операций `LOG0`, `LOG1`... во время выполнения транзакции. Поле `address` — это адрес контракта, который создал жур-

нал, поля `topic` — до четырех 32-байтовых значений, а поле `data` — байтовый массив произвольного размера;

- *Logbloom* — это фильтр Блума, состоящий из значений `address` и `topic` всех журналов транзакции. Он отличается от того, что присутствует в заголовке блока.

## 4.2.4. Дерево Меркла — Патриции

В Ethereum счета сопоставляются с их соответствующими состояниями. Карта соответствий между всеми счетами Ethereum, включая EOA и умные контракты, и их состояниями в совокупности называется *глобальным состоянием*. Для хранения карты соответствий в Ethereum применяется *префиксное дерево Меркла — Патриции* (Merkle-Patricia trie, MPT). По сути, дерево Меркла —Патриции представляет собой комбинацию элементов дерева Меркла и дерева Патриции.

В *главе 3* мы говорили, что деревья Меркла — это бинарные хэш-деревья, где конечные узлы (листья) содержат хэш блоков данных, а каждый нелистовой узел содержит хэши своих дочерних узлов. Если мы выстроили такую структуру хэшей, то можем легко проверить, является ли определенная транзакция частью блока. Деревья Меркла облегчают эффективную и безопасную проверку содержимого в децентрализованных системах. Вместо загрузки каждой транзакции и каждого блока легкие узлы могут загружать только цепочки заголовков блоков, т. е. 80-байтовые порции данных для каждого блока, которые содержат всего пять компонентов: хэш предыдущего заголовка блока, метку времени, уровень сложности, найденное значение nonce и корневой хэш дерева Меркла, отражающий все транзакции для этого блока. Это весьма полезная информация, но, помимо проверки доказательства наличия транзакции в блоке, вы ничего больше не можете сделать. Существенное ограничение заключается в том, что невозможно, опираясь на ветвь дерева Меркла, доказать информацию о текущем состоянии (например, общее количество цифровых активов, регистрацию имен, статус финансовых контрактов). Даже для того чтобы просто узнать, сколько у вас биткойнов, требуется выполнить много запросов и проверок.

Дерево Патриции, с другой стороны, является разновидностью *дерева остатков* (Radix tree), или радиксного дерева[3]. Название PATRICIA расшифровывается как Practical Algorithm to Retrieve Information Coded In Alphanumeric — практический алгоритм получения информации, закодированной в буквенно-цифровой форме. Дерево Патриции облегчает эффективные операции вставки/удаления. Поиск значения ключа в дереве Патриции очень эффективен. Ключи всегда кодируются в виде пути. Итак, ключ — это путь от корня до конечного узла, где хранится искомое значение. Ключами обычно являются строки, которые ведут нас вниз по пути,

---

[3] Пояснения и примеры к понятию *дерева остатков* на русском языке можно найти, например, по адресу: **http://korzh.net/2010-11-derevo-ostatkov-ili-radix-tree.html**.

где каждый символ строки указывает, какой дочерний узел следует далее, чтобы в итоге достичь конечного узла и найти значение, сохраненное в нем.

Таким образом, деревья Меркла — Патриции предоставляют собой криптографически аутентифицированную структуру данных, используемую для хранения всех связок вида `{ключ, значение}` в блокчейне Ethereum. Они полностью детерминированы, а это означает, что деревья Патриции с одинаковыми связками `{ключ, значение}` гарантированно будут одинаковыми вплоть до последнего байта. Операции вставки, поиска и удаления достаточно эффективны при сложности $O(\log_2(n))$[4]. В дереве Меркла — Патриции, как и в обычном дереве Меркла, хэш узла используется как указатель на узел, а пары «ключ-значение» создаются следующим образом:

```
Ключ == SHA-3(RLP(значение))
```

Дерево Меркла — Патриции наследует от метода Меркла защищенную и детерминированную древовидную структуру, а метод Патриции обеспечивает эффективную функцию поиска информации. Как вы уже могли догадаться, корневой узел в дереве Меркла — Патриции становится криптографическим отпечатком всей структуры данных. Когда транзакции передаются по каналам связи в одноранговой сети Ethereum, они собираются каждым узлом майнинга, который их получил. Затем узлы формируют дерево (точнее, префиксное trie-дерево) и вычисляют корневой хэш для включения в заголовок блока. Транзакции отправляются другим узлам или клиентам после их сериализации в списки. Принимающие стороны должны десериализовать их обратно, чтобы сформировать у себя дерево транзакций для проверки на соответствие корневому хэшу. Также заметим, что в Ethereum деревья Меркла — Патриции немного модифицированы для лучшего соответствия требованиям Ethereum. Вместо двоичного представления используется шестнадцатеричная строка — $X$ символов из 16-значного «алфавита». Следовательно, узлы дерева могут иметь по 16 дочерних узлов (алфавит из 16 символов), а максимальная глубина дерева равна $X$. Просто, чтобы вы знали, — шестнадцатеричный символ строки, образующей путь, часто называют словом *nibble* (полубайт, тетрада).

Основная идея использования дерева Меркла — Патриции в Ethereum заключается в том, что для пересчета корневого хэша после одной операции требуется изменять минимальное количество узлов. Таким образом, объем и сложность структуры данных остаются минимальными.

## 4.2.5. RLP-кодирование

Как вы заметили, мы в предыдущих разделах несколько раз упоминали RLP-кодирование. Сейчас мы расскажем вам, о чем идет речь. Это метод сериализации, используемый в Ethereum для блоков, транзакций и сообщений протокола проводной[5] связи при отправке данных по сети, а также для данных состояния счета при

---

[4] В данном случае запись $O(\log_2(n))$ обозначает не функцию O от аргумента в скобках, а сокращение от «Operations» — подразумевается количество выполняемых в скобках операций.

[5] Очевидно, под проводной связью авторы подразумевают любой физический канал передачи данных, включая Wi-Fi и мобильный Интернет.

сохранении состояния в дереве Патриции. Короче говоря, когда сложные структуры данных необходимо сохранить или передать, а затем восстановить на принимающей стороне для обработки, хорошей практикой является сериализация объектов. RLP в этом смысле похож на форматы JSON и XML, но считается, что RLP более минималистичен, экономичен, прост в реализации и гарантирует абсолютную целостность данных. Вот почему RLP был выбран в качестве основного метода сериализации для Ethereum. Его единственное назначение — хранить вложенные массивы необработанных байтов. Он даже не пытается определять какие-либо конкретные типы данных — такие как логические значения, числа с плавающей запятой, целые числа и т. д., и предназначен только для хранения структур данных в виде вложенных массивов. Карты соответствия «ключ-значение» не поддерживаются RLP в явном виде. Следовательно, рекомендуется представлять такие карты в виде `[[k1,v1], [k2,v2],...]`, где `k1`, `k2...` расположены в лексикографическом порядке (отсортированы с использованием стандартного способа сортировки строк). В качестве альтернативы можно использовать кодирование дерева Патриции[6] более высокого уровня, которое имеет собственную схему кодирования RLP.

Пожалуйста, не забывайте, что RLP используется только для кодирования структуры данных и совершенно не знает о типе кодируемого объекта. Хотя это помогает уменьшить размер массива необработанных байтов, получатель должен знать тип объекта, который он пытается декодировать.

## 4.2.6. Транзакция Ethereum и структура сообщения

В предыдущем материале мы рассмотрели структуру блока и различные поля в заголовке блока. Чтобы транзакция была обработана майнерами или узлами Ethereum, она должна иметь стандартизированную структуру. Типичная транзакция Ethereum (например, отправленная методом `sendRawTransaction()`, который мы рассмотрим позже) состоит из следующих полей:

- *nonce* — это целое число, просто счетчик, равный количеству транзакций, отправленных с адреса отправителя, т. е. порядковый номер транзакции;
- *gasPrice* — цена, которую вы готовы заплатить за единицу газа, выраженная в Wei;
- *gasLimit* — максимальное количество газа, которое должно использоваться при выполнении этой транзакции, что также ограничивает максимальное количество вычислительных шагов, которые разрешено использовать при выполнении транзакции;
- *To* — 160-битовый адрес получателя или адрес контракта. Для транзакции, которая используется для создания контракта (это означает, что адрес контракта еще не существует), поле остается пустым;
- *Value* — общее количество эфира (в единицах Wei), который должен быть передан получателю отправителем транзакции;

---

[6] См. **https://github.com/ethereum/wiki/wiki/Patricia-Tree**.

- *V*, *r*, *s* — значения, соответствующие подписи ECDSA транзакции. Они также представляют отправителя этой транзакции;
- *init* — это необязательное поле, которое используется только с транзакциями создания контрактов. Поле *init* может содержать байтовый массив неограниченного размера, в котором содержится исполняемый байт-код для процедуры инициализации.

  Содержимое *init* используется только один раз для инициализации нового счета умного контракта и после этого удаляется. Операция инициализации возвращает основную часть кода контракта после сопоставления его со счетом контракта. Имейте в виду, что это соответствие устанавливается навсегда и никогда не меняется;
- *Data* — необязательное поле, которое может содержать сообщение для отправки в контракт или в простой внешний счет. Оно не имеет специальной функции как таковой по умолчанию, но у EVM есть код операции, с помощью которой контракт может получить доступ к этому полю данных, выполнить необходимые вычисления и поместить их в хранилище.

Обратите внимание, что упомянутые поля поставляются в строго указанном порядке и все они кодируются по RLP, за исключением имен полей. Таким образом, транзакция Ethereum фактически означает подписанный пакет данных в этих полях. Поля *gasPrice* и *gasLimit* важны для предотвращения атаки типа «отказ в обслуживании». Чтобы предотвратить случайные или преднамеренные попытки организации бесконечных циклов или других вычислительных потерь в коде, каждая транзакция должна установить ограничение на количество вычислительных шагов, которые она может использовать.

Транзакции Ethereum на самом деле являются функциями перехода состояний, потому что успешная транзакция меняет состояние системы. Кроме того, результат этих транзакций может быть сохранен, как мы уже упоминали ранее в *разд. «Состояние счета»*.

Сообщения Ethereum, с другой стороны, похожи на транзакции, но инициируются только счетами контрактов, а не счетами внешних владельцев. Кроме того, сообщения пересылаются только между счетами контрактов, поэтому их также называют *внутренними транзакциями*. Таким образом, контракты имеют возможность отправлять сообщения другим контрактам.

Как правило, сообщение генерируется, когда контракт, выполняя свой код, сталкивается с кодами операций `CALL` или `DELEGATECALL`. Таким образом, сообщения похожи на вызовы функций в среде выполнения Ethereum. Также важно отметить, что сообщения всегда являются необработанными и никогда не сериализуются и не десериализуются. Сообщение содержит следующие поля:

- *Sender* — отправитель сообщения как необязательная опция;
- *Recipient* — адрес контракта получателя;
- *Value* — сумма перевода в Wei на адрес контракта, вместе с сообщением;

- *Data* — необязательное поле, которое может содержать входные данные для контракта получателя, предоставленные отправителем;
- *gasLimit* — максимальное количество газа, которое можно потратить на выполнение кода, запускаемого сообщением. Иногда его называют *startGas*.

Итак, мы посмотрели транзакции и сообщения. Транзакция Ethereum может быть направлена от EOA до EOA или от EOA до счета контракта. Существует также ситуация, когда EOA запускает транзакцию для создания счета контракта (вспомните поле *init*, которое мы только что рассмотрели). А теперь подумайте, что такое транзакция? Да, это мост между внешним миром и блокчейном Ethereum, но что еще? Если вы рассмотрите транзакцию поближе, то увидите, что транзакция — это подписанная внешним пользователем инструкция, которая затем сериализуется и передается в блокчейн (рис. 4.7).

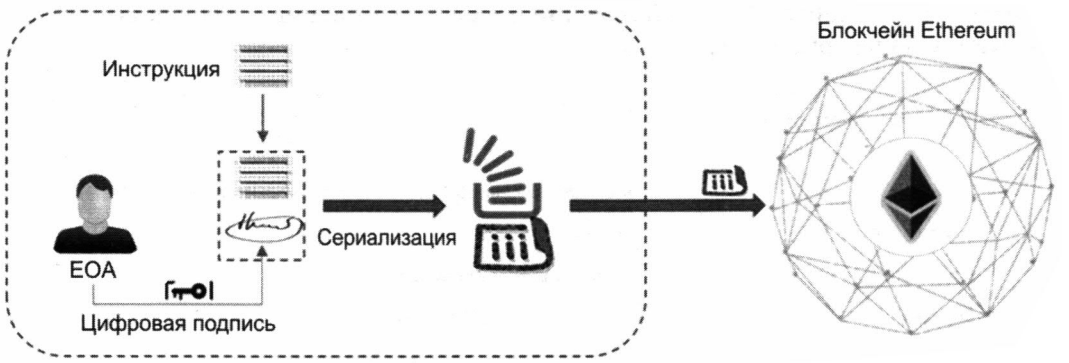



**Рис. 4.7. Пошаговая схема инициализации транзакции**

Что происходит после того, как транзакция отправлена в блокчейн? Если транзакция признана действительной, она начинает выполняться на каждом узле Ethereum. В то время как эта транзакция находится в процессе выполнения, Ethereum следит за промежуточным состоянием транзакции и отслеживает ход выполнения. Это связано с тем, что если транзакция не завершается из-за нехватки газа, то все выполнение вплоть до текущего момента должно быть отменено. Кроме того, собранная во время выполнения информация требуется сразу после завершения транзакции. Итак, промежуточное состояние транзакции содержит следующие данные:

- *Self-destruct set* — набор счетов (если есть), которые будут аннулированы после завершения транзакции;
- *Log series* — архивированные и индексируемые «контрольные точки» выполнения кода EVM для отслеживания вызовов контракта;
- *Refund balance* — это сумма, подлежащая возврату на счет отправителя после выполнения транзакции. Хранение данных в Ethereum весьма затратно, поэтому в Ethereum есть инструкция `SSTORE`, которая используется в качестве счетчика возврата. Счетчик возврата начинается с нуля (состояние возврата отсутствует) и увеличивается каждый раз, когда транзакция или контракт удаляет что-либо из хранилища. Обратите внимание, что эта сумма возврата отличается от суммы

неиспользованного газа транзакции и возвращается отправителю в дополнение к неиспользованному газу.

В более ранних версиях Ethereum весь газ, отведенный для выполнения транзакции или контракта, считался потраченным независимо от успеха выполнения. Это не всегда имеет смысл. Вам вряд ли понравится, если выполнение остановится из-за какой-либо проблемы с авторизацией/разрешением или другой технической проблемы, а оставшийся газ все равно будет израсходован. В обновлении Bizantium добавлено понятие «возвратного кода» для обработки исключений. В случае прекращения действия контракта возвратный код может использоваться для отмены изменений состояния, возврата причины сбоя и зачисления оставшегося газа отправителю. После успешного выполнения транзакций или контрактов происходит переход состояния, который мы подробно рассмотрим в следующем разделе.

Точно так же, как на сайте **blockchain.info** вы могли увидеть живую транзакцию в биткойнах, на сайте **https://etherscan.io** по хэшу транзакции Ethereum вы найдете следующую информацию (рис. 4.8).

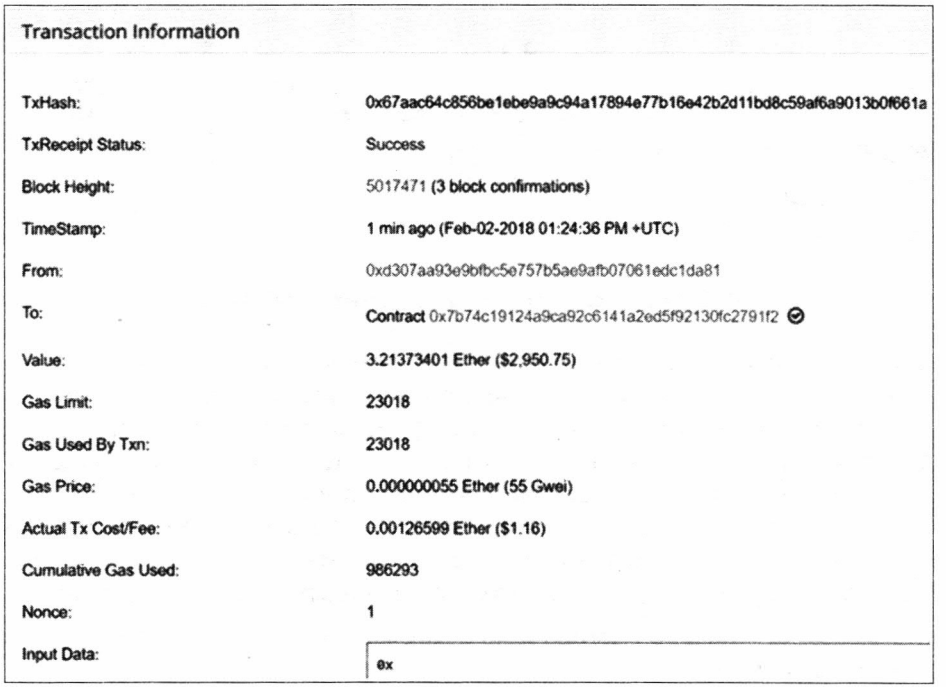



**Рис. 4.8. Информация по хэшу транзакции Ethereum**

## 4.2.7. Функция перехода состояния Ethereum

В предыдущем разделе мы говорили о транзакциях и сообщениях Ethereum. Теперь мы знаем, что когда транзакция проведена успешно, происходит переход состояния блокчейна. Функция перехода состояния Ethereum имеет общий вид (рис. 4.9):

```
APPLY(S,Tx) -> S'\\ где S - старое состояние, а S' - новое состояние
```

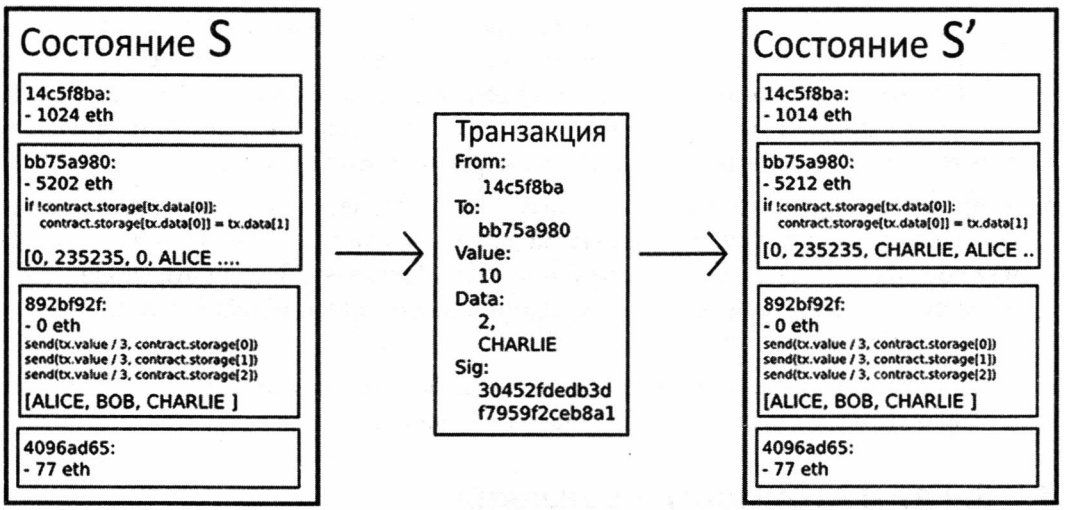



**Рис. 4.9.** Функция перехода состояния Ethereum

Таким образом, функция перехода состояния, когда `Tx` применяется к состоянию `S`, чтобы получить новое состояние `S'`, может быть определена следующим образом:

- проверяем транзакцию, чтобы увидеть, правильно ли она сформирована:
  - имеет ли правильное количество значений;
  - действительна ли подпись;
  - совпадает ли одноразовое число (счетчик) транзакции с одноразовым числом в данных счета отправителя;
  - если какой-либо из предыдущих пунктов недействителен, возвращаем ошибку;
- рассчитываем комиссию и остаток на счете:
  - вычисляем комиссию за транзакцию: `gasLimit * gasPrice`;
  - определяем адрес отправителя по его подписи;
  - вычитаем комиссию из баланса счета отправителя и увеличиваем одноразовое число (счетчик транзакций) отправителя;
  - если на балансе недостаточно средств, возвращаем ошибку;
- инициализируем `GAS=gasLimit` и вычитаем определенное количество газа на байт исполняемого кода, чтобы оплатить транзакцию;
- переводим предмет транзакции (может быть что угодно) со счета отправителя на счет получателя. Помните, что транзакция может быть совершена для чего угодно, имеющего внутреннюю ценность, — например, для земли, транспортного средства, токенов других криптовалют и т. п., но цена газа номинируется в эфире (точнее, в Wei), чтобы майнеры могли принять транзакцию и получить оплату за работу. Если получающая учетная запись еще не существует, создайте ее.

Если счет получателя принадлежит контракту, а не внешнему пользователю, запустите код контракта либо до завершения, либо до тех пор, пока не закончится газ. Обратите внимание, что код контракта исполняется на EVM каждого узла как часть процесса проверки блока, так что блок, а следовательно, и выходной результат контракта, становится частью основной цепочки блоков;

- если транзакция не удалась из-за того, что у отправителя не было достаточно денег на счете, или при выполнении кода не хватило газа, отменяем все изменения состояния (благодаря реализации дерева Меркла — Патриции), за исключением оплаты комиссии за уже проделанные вычисления, и добавляем комиссию на счет майнера;
- если транзакция прошла успешно, возвращаем оставшийся газ отправителю и отправляем стоимость потребленного газа майнеру.

## 4.2.8. Газ и стоимость транзакции

Транзакции в системе Ethereum работают на *газе* — фундаментальной единице стоимости вычислений. Каждая транзакция, будь то EOA или контракт, должна иметь установленные значения `gasLimit` и `gasPrice` для расчета комиссии. Эта комиссия выплачивается майнерам для компенсации их вложений в ресурсы и за работу, которую они выполняют. Очевидно, что это создает у майнеров мотивацию включить транзакцию в блок и получить вознаграждение.

Обычно один вычислительный шаг стоит ровно один газ, но некоторые операции, требующие большого объема вычислений или хранения, стоят дороже. На каждый байт данных транзакции требуется около пяти единиц газа. Приведем несколько примеров: для сложения двух чисел (код операции EVM `ADD`) требуется около трех единиц газа, умножение двух чисел (код операции EVM `MUL`) требует примерно пять единиц газа, вычисление хэша по SHA-3 требует около 30 единиц газа (как вы понимаете, по причине большой вычислительной нагрузки). Стоимость хранения рассчитывается аналогичным образом, но по уважительным причинам весьма дорого стоит. И в самом деле: транзакция может включать в себя неограниченное количество данных. При этом хранение стоит 68 единиц газа за каждый байт ненулевых данных транзакции. Для хранения 256-битового слова в контракте требуется приблизительно 20 000 единиц газа. В «Желтой книге» Ethereum вы можете найти большой перечень кодов операций и соответствующих цен.

Затем стоимость операции в единицах газа умножается на текущую стоимость газа `gasPrice`. В отличие от Bitcoin, расчет стоимости транзакции в системе Ethereum является более сложным. Он учитывает затраты на пропускную способность, хранение и вычисления. Наличие такого механизма расчета оплаты защищает сеть Ethereum от злоумышленника, который может захотеть внедрить бесконечный цикл вычислений (приводящий к атакам типа «отказ в обслуживании») или потреблять все больше и больше места, сохраняя бессмысленные данные.

С одной стороны, инициатор транзакции сам указывает цену, которую согласен уплатить за единицу газа. Стоимость транзакции в эфирах вычисляется как произ-

ведение необходимого количества газа на его стоимость, указанную отправителем. С другой стороны, у майнеров есть своя стратегия расчета минимальной приемлемой цены за газ. Майнер соглашается взять в обработку только те транзакции, в которых его устраивает предложенная цена за газ. Итак, каким же образом вычисляют общую стоимость транзакции? Не приблизительную, а фактическую стоимость? Общая «эфирная» стоимость транзакции `TotalCost` основана на двух факторах: `gasUsed` и `gasPrice`:

```
TotalCost = gasUsed × gasPrice
```

Множитель `gasUsed` — это общее количество газа, потребленное при исключении кодов операций EVM для инструкций, а множитель `gasPrice` — это цена газа в эфирах, которая указана пользователем.

Если общее количество газа, необходимого для вычислений (включая транзакцию, сообщение и любые вложенные сообщения, которые могут быть вызваны), меньше или равно `gasLimit`, тогда транзакция обрабатывается майнером. Однако если затраты газа превышают `gasLimit`, то все изменения отменяются (хотя это действительная транзакция), за исключением того, что майнер все равно может получить вознаграждение. Но что же происходит с избытком газа? Весь неиспользованный газ (точнее, его стоимость в криптовалюте) после выполнения транзакции возмещается отправителю. Отправителям не нужно беспокоиться о перерасходе средств, поскольку с них взимают плату только за потребленный газ. Это означает, что удобно и безопасно отправлять транзакции с лимитом газа, значительно превышающим расчетный объем, — для гарантированного прохождения транзакций. Также рекомендуется не предлагать майнерам слишком высокую цену за газ и использовать среднюю цену с сайта **https://ethgasstation.info/**.

Давайте шаг за шагом пройдем весь процесс создания транзакции в сети Ethereum:

- каждая транзакция должна установить `gasLimit` — предельное количество газа, которое она готова потратить (`gasLimit` также иногда называют `startGas`), и цену, которую она готова платить за единицу газа (`gasPrice`). В начале обработки транзакции максимальная стоимость `gasLimit × gasPrice` списывается со счета отправителя транзакции. Помните, что на самом деле это не окончательная стоимость транзакции, а ее *предполагаемая* стоимость с небольшим запасом. После завершения транзакции определяется ее *фактическая* стоимость `gasUsed × gasPrice`, а излишек возвращается на счет отправителя. Предварительная стоимость `gasLimit × gasPrice` заранее вычитается со счета отправителя в качестве залога, поскольку существует вероятность того, что отправитель обанкротится, пока инициированная им транзакция находится на обработке;
- все операции во время выполнения транзакции, включая чтение и запись в базу данных, сообщения и каждый вычислительный шаг, предпринимаемый EVM, — такой как сложение, вычитание, хэширование и т. п., потребляют определенное и заранее объявленное количество газа;
- обычная транзакция — это та, которая успешно выполняется без превышения указанного значения `gasLimit`. После таких транзакций может оставаться некото-

рое количество газа — скажем, `gas_rem`. После успешного выполнения транзакции отправитель транзакции получает остаток `gas_rem * gasPrice`, а майнер блока — вознаграждение `(gasLimit - gas_rem) * gasPrice`;

- если для успешного завершения транзакции не хватает газа, все операции отменяются, но транзакция, тем не менее, остается действительной. В таких ситуациях единственным результатом транзакции является то, что вся сумма `gasLimit * gasPrice` передается майнеру в оплату за фактически выполненные вычисления;
- в случае внутренних транзакций контрактов, когда контракт отправляет сообщение другому контракту для дальнейшего выполнения, он также имеет возможность установить `gasLimit`. Эта опция специально предназначена для *подчиненного выполнения*, возникающего при получении сообщения, поскольку существует вероятность, что вызываемый контракт имеет бесконечный цикл. Если во время выполнения подчиненного контракта не хватает газа, то это выполнение отменяется, что защищает от случайных бесконечных циклов или преднамеренных попыток DoS-атак. В любом случае газ расходуется, и его стоимость зачисляется майнеру. Обратите внимание, что когда вычисления инициированы родительским контрактом, в сообщении контракта только инструкции расходуют газ, а данные не стоят ничего. Дело в том, что данные из родительского контракта не нужно копировать, — на них можно просто ссылаться через указатель.

Первый выпуск Ethereum (Frontier) имел цену по умолчанию на газ $0{,}05 \cdot 10^{12}$ Wei. Во втором выпуске Ethereum (Homestead) цена на газ по умолчанию была снижена до $0{,}02 \cdot 10^{12}$ Wei. (Напомним, что Wei — это наименьшая денежная единица в платежной системе Ethereum. 1 Ether = $1 \cdot 10^{18}$ Wei.)

Наверное, вам интересно, почему для оценки вычислительных расходов используют отдельную единицу измерения — газ, а не номинируют вычисления сразу в единицах эфира? Это сделано преднамеренно, потому что единицы газа обозначают количество вычислительных единиц (например, количество операций сложения или умножения), и объем вычислений никогда не меняется для одной и той же операции, в то время как цена криптовалюты обычно колеблется под влиянием рыночной ситуации. Кроме того, пользователи могут участвовать в своего рода аукционе и независимо друг от друга предлагать майнерам большую или меньшую цену на газ. От этого может зависеть скорость прохождения транзакции — майнерам выгоднее обработать транзакции с более высокой предложенной ценой.

Мы уже знаем, что каждый узел Ethereum, участвующий в сети, запускает виртуальную машину EVM как часть протокола проверки блока. Это означает, что все узлы выполняют один и тот же набор транзакций и контрактов — избыточные параллельные вычисления, необходимые для достижения консенсуса. Поскольку такая избыточность делает вычисления дорогостоящими, появляется стимул не использовать блокчейн для вычислений, которые могут быть выполнены вне цепочки (теория игр в действии!).

Как правило, 21 000 единиц газа взимается за любую транзакцию в качестве минимальной платы для покрытия стоимости расчетов эллиптической кривой при вы-

числении адреса отправителя из подписи, а также за дисковое пространство для хранения транзакции. Существуют различные способы оценки потребности в газе для транзакций и умных контрактов. Например `valuGas` — это функция библиотеки web3 для оценки потребности в газе заданной функции. Кроме того, для оценки общей стоимости в клиентском приложении Geth имеется «предсказатель» цены газа (gas price oracle), а `web3.eth.getGasPrice` — это встроенная функция web3 для поиска приблизительной цены на газ. В листинге 4.1 приведен фрагмент кода, который можно использовать при работе с фреймворком Truffle:

**Листинг 4.1. Пример расчета ожидаемой стоимости транзакции**

```
var MyContract = artifacts.require("./MyTest.sol");

// getGasPrice возвращает цену газа в Wei
MyContract.web3.eth.getGasPrice(function(error, result){
    var gasPrice = Number(result);
    console.log("Current gasPrice is " + gasPrice + " wei");

    // Получаем ссылку на объект контракта
    MyContract.deployed().then(function(instance) {

        // Запрашиваем оценку газа из функции giveAwayDividend()
        return instance.giveAwayDividend.estimateGas(1);

    }).then(function(result) {
        var gas = Number(result);

        console.log("Total gas estimation = " + gas + " units");
        console.log("Total Transaction Cost estimation in Wei = " + (gas * gasPrice) + " wei");
        console.log("Total Transaction Cost estimation in Ether = " +
                                MyContract.web3.fromWei((gas * gasPrice), 'ether') + " Ether");
    });
});
```

При написании умных контрактов на языке Solidity многие предпочитают использовать *постоянные функции* для вычисления определенных значений вне цепочки или просто делать RPC-запрос к своей локальной цепочке блоков. Поскольку такие постоянные функции не изменяют состояние блокчейна, они в некотором смысле бесплатны, т. к. не потребляют газ. Если такие функции используются внутри транзакции, то весьма вероятно, что возникнут расходы на газ.

Давайте теперь поговорим о лимите газа для блока. Вспомните, что в системе Bitcoin установлено ограничение размера блока 1 Мбайт, а у Bitcoin Cash — 2 Мбайт. Майнеры собирают столько транзакций, сколько может поместиться в этих блоках. Ethereum, однако, применяет совершенно другой способ ограничения — размер блока ограничен лимитом газа блока. Различные транзакции имеют

разный лимит газа. Когда майнер собирает транзакции Ethereum в блок, сумма лимитов газа для всех транзакций не должна превышать лимит газа для блока.

Разные майнеры могут формировать разные наборы транзакций для включения в блок. Лимит газа для блока рассчитывается динамически. Протокол Ethereum позволяет майнеру корректировать лимит газа блока с коэффициентом $^{1}/_{1024}$ (0,0976%) в большую или меньшую сторону. Майнеры в сети Ethereum используют программу майнинга, например, майнинговый клиент Ethminer, который подключается к клиентскому узлу geth или Parity. И у geth, и у Parity есть опции, которые майнеры могут изменить.

## 4.3. Умные контракты Ethereum

В отличие от системы Bitcoin, которая предназначена для криптовалютных платежей, Ethereum может намного больше благодаря *умным контрактам*. Вы уже получили некоторое представление об умных контрактах, когда в предыдущих разделах мы говорили о счетах контрактов. Мы рассмотрим прикладные аспекты разработки умных контрактов в следующих главах, а сейчас поговорим о том, что представляет собой такой контракт.

Давайте начнем с того, почему умный контракт, или смарт-контракт, так называется? На самом деле в умном контракте изначально нет ничего «умного». Он будет умен ровно настолько, насколько умно вы сами запрограммируете его логику. В этом и заключается красота и привлекательность платформы Ethereum, которая позволяет вам реализовать свои идеи. Давайте кратко сформулируем главные свойства умных контрактов Ethereum:

- умные контракты находятся внутри блокчейна Ethereum;
- у них есть свой счет, следовательно, есть адрес и баланс;
- они способны отправлять сообщения и получать транзакции;
- они активируются при получении транзакции, а также могут быть деактивированы;
- к ним, так же как и к транзакциям, применимо понятие платы за исполнение кода и платы за хранение.

Весь код в Ethereum, включая умные контракты, компилируется в низкоуровневый стековый байт-код виртуальной машины Ethereum (EVM). Популярными языками высокого уровня для написания умных контрактов являются Solidity, Serpent и LLL, а их соответствующие компиляторы преобразуют код высокого уровня в байт-код EVM. Мы знаем, что контракты могут быть добавлены в блокчейн любым внешним агентом EOA. Поскольку вычисления и хранение в Ethereum обходятся очень дорого, желательно, чтобы логика контракта была написана настолько простым и оптимизированным образом, насколько это возможно. Когда умный контракт развернут в сети блокчейна Ethereum, любой пользователь сети может вызвать функции контракта. Функции обычно имеют встроенные опции безопасности,

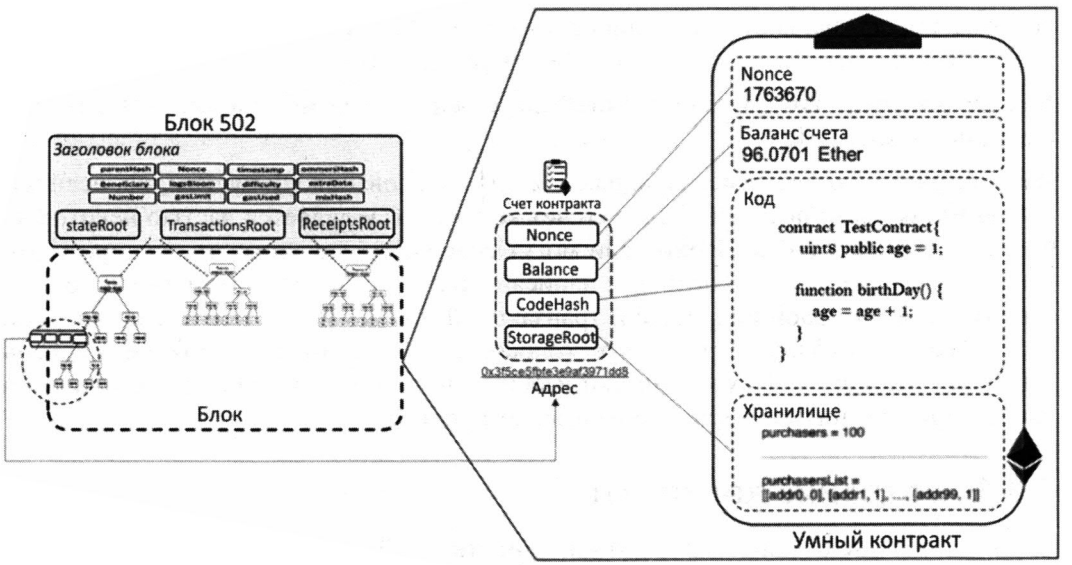



**Рис. 4.10.** Умный контракт в составе блока Ethereum

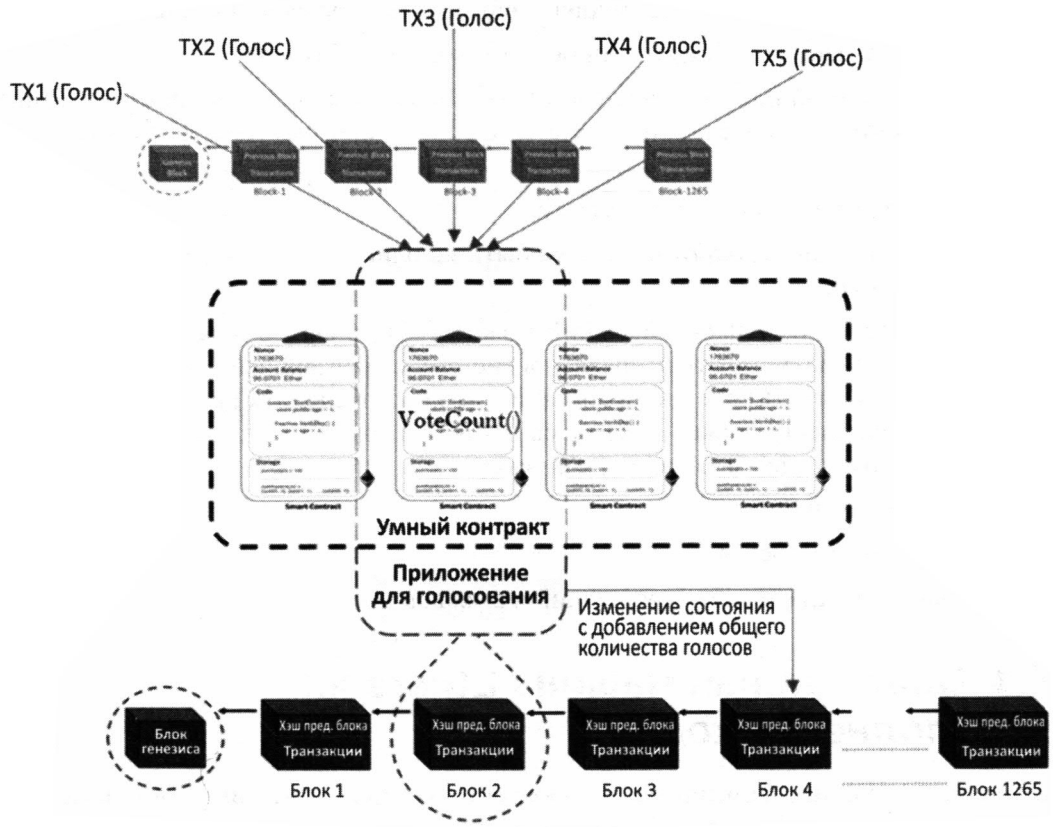



**Рис. 4.11.** Взаимодействие приложения с логикой умного контракта

которые предотвращают несанкционированный доступ, — тем не менее попытки вызова могут быть сделаны, хотя и не будут успешными.

Умный контракт внутри блока Ethereum можно схематически представить, как показано на рис. 4.10.

Давайте рассмотрим пример приложения для голосования. Допустим, написан умный контракт, который имеет адрес (счет контракта) и является частью некоторого блока в цепочке блоков. Избиратели могут совершать сделки по этому адресу (отдавать свои голоса). Код контракта написан так, что он будет увеличивать количество голосов с каждой полученной транзакцией и через некоторое время завершит свою работу, публикуя результат голосования (изменяет состояние системы Ethereum). На рис. 4.11 схематически изображено взаимодействие приложений для голосования с умным контрактом на высоком уровне.

### 4.3.1. Создание контракта

Вспомните о специальной транзакции, единственной целью которой является создание контракта. Это немного иной тип транзакции по сравнению с другими транзакциями Ethereum. Прежде чем транзакция создания контракта будет запущена для создания счета, она должна инициализировать четыре свойства счета:

- счетчик транзакций *nonce* должен быть изначально обнулен;
- в поле *Account Balance* (баланс счета) должно быть записано количество эфира, переданного отправителем, и та же сумма должна быть вычтена из баланса счета отправителя;
- хэш *StorageRoot* должен быть пустым;
- хэш кода контракта *codeHash* должен быть заполнен 256-битовым хэшем Keccak пустой строки.

После подготовки начальных значений можно создать счет с помощью кода инициализации. Код может быть отправлен в систему с транзакцией, которая выполняет реальную работу. В коде инициализации можно определить целый набор действий, а его выполнение может вызвать события, которые не являются внутренними по отношению к среде выполнения, например:

- изменение хранилища счета;
- создание других счетов;
- отправка дополнительных сообщений и вызовов.

## 4.4. Виртуальная машина Ethereum и выполнение кода

Ethereum — это программируемый блокчейн, который позволяет пользователям создавать собственные процедуры произвольной сложности с помощью Тьюринг-полных языков. Виртуальная машина EVM — это исполнительный механизм

Ethereum, который служит средой выполнения для умных контрактов. В этом заключается основная инновация Ethereum, которая делает его уникальным по сравнению с другими системами блокчейна. EVM также играет важную роль в выполнении транзакций, изменении состояния Ethereum и достижении консенсуса. При разработке EVM подразумевались следующие цели:

- **простота** — идея заключалась в том, чтобы сделать EVM как можно более простой на основе кодов языка низкого уровня. Вот почему количество низкоуровневых кодов операций и типов данных сведено к минимуму, но до такой степени, что с их помощью все еще можно составлять логические конструкции произвольной сложности. Всего предусмотрено 160 инструкций, из которых 65 логически различаются;
- **абсолютный детерминизм** — выполнение инструкций с одним и тем же набором входных данных должно давать одинаковый набор выходных данных. Это помогает поддерживать целостность EVM без какой-либо неопределенности. Детерминизм вместе с понятием *вычислительный шаг* помогает приближенно оценить расход газа перед выполнением кода;
- **оптимизация пространства** — в децентрализованных системах экономия места в хранилище является самой большой проблемой. Вот почему сборка EVM остается максимально компактной;
- **оптимизация для специальных операций** — машина EVM оптимизирована для выполнения некоторых специальных операций, таких как конкретные типы арифметических операций в криптографии (модульная арифметика), чтения блоков или данных транзакций, взаимодействия с «состояниями» и тому подобных. Еще один пример: 256-битовое слово (32 байта) для хранения криптографических хэшей, где EVM работает с 256-битовым целым числом;
- **простая безопасность** — в определенном смысле цена на газ помогает гарантировать, что EVM не подвержена взлому. Если бы газ не имел ценности, злоумышленники могли бы атаковать систему всеми возможными способами. В то время как почти каждая операция на EVM требует затрат на газ, легко найти хорошую модель стоимости газа для добропорядочных пользователей[7].

Мы уже говорили, что каждый узел в сети Ethereum запускает локальную копию EVM, выполняет все транзакции и умные контракты и сохраняет у себя конечное состояние. Именно EVM записывает умные контракты и данные в блокчейн и выполняет коды операций транзакции и код контракта. То есть EVM служит средой выполнения (Runtime Environment, RTE) для умных контрактов Ethereum и обеспечивает безопасное выполнение кода. Очевидно, что умные контракты или транзак-

---

[7] Как показала практика, механизм формирования рыночной цены на газ в Ethereum далек от идеального. Богатые пользователи склонны предлагать высокую цену за газ, чтобы привлечь майнеров и обеспечить быстрое прохождение своих транзакций. Это ведет к тому, что остальным пользователям приходится тоже поднимать цену за газ или рисковать тем, что их транзакции надолго «зависнут» в очереди. Многие пользователи платформы Ethereum считают, что рекомендованная цена за газ, которая по умолчанию выставляется в кошельках на основании статистики сети, необоснованно завышена.

ции выполняются в EVM после проверки соответствующих цифровых подписей. Таким образом, состояние Ethereum может измениться только после успешного выполнения инструкций в среде EVM.

Если вы не подключите EVM к остальной сети, она будет исправно работать на изолированном узле. Запуск EVM в безопасной «песочнице» может использоваться для тестирования умных контрактов. Это облегчает разработку качественных, надежных и готовых к работе контрактов.

Чтобы разбираться, как умные контракты работают в среде EVM, вы должны понимать, как данные организуются, хранятся и обрабатываются на языке Solidity, Serpent и даже на тех языках, которые могут появиться в будущем. Возможно, вы захотите использовать EVM как движок базы данных. Хотя мы не будем углубляться в основы программирования на языке Solidity, в этом разделе мы покажем, как язык высокого уровня взаимодействует с EVM (рис. 4.12).

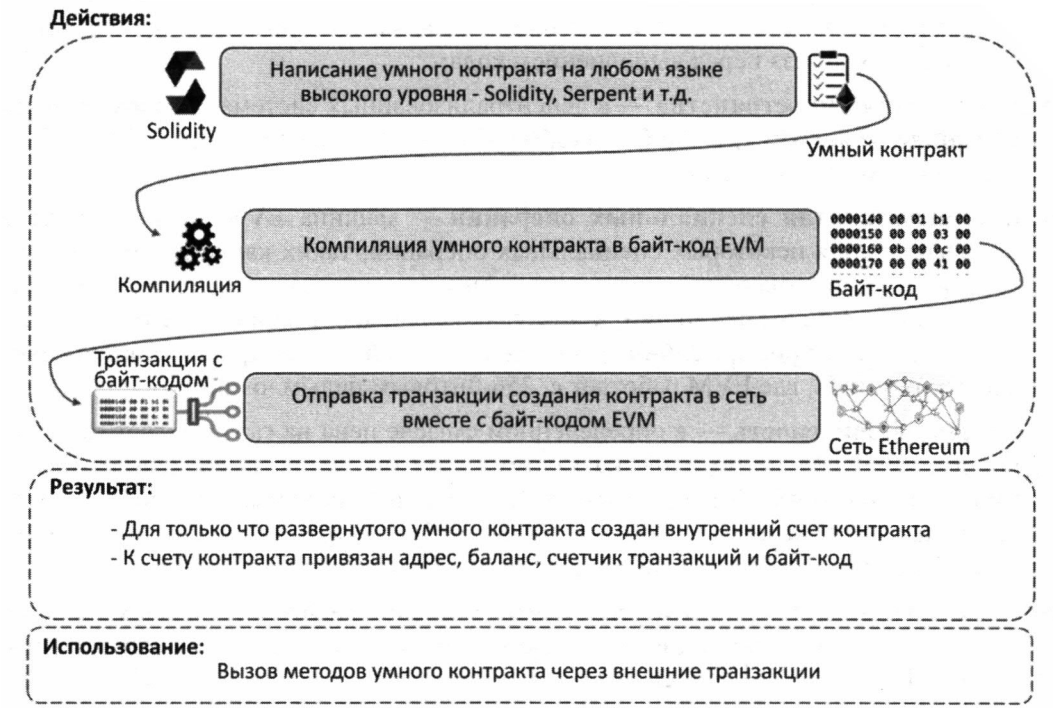



**Рис. 4.12. Создание, развертывание и использование умного контракта**

Давайте теперь разберемся с управлением памятью с помощью EVM. Взгляните на следующие три стратегии, которых придерживается EVM:

- **постоянное хранилище:**
  - хранилище представляет собой набор 256-битовых пар «ключ-значение». Это означает, что ключи и значения представляют собой 256-битовые слова (32 байта);

- в рамках контракта невозможно использовать перечислитель хранилища[8];
- в любой момент времени состояние контракта может быть определено с помощью переменных уровня контракта, называемых *переменными состояния*, которые всегда находятся в хранилище и не могут быть обновлены во время выполнения. Это означает, что структура хранилища задается только один раз при создании договора и не может быть изменена. Однако содержимое переменных можно изменить с помощью вызовов `sendTransaction`;
- чтение и обновление хранилища — затратные операции;
- контракты не могут читать, записывать или обновлять любое другое хранилище, которое им не принадлежит;
- `SSTORE` и `SLOAD` — часто используемые инструкции. Пример: инструкция `SSTORE` извлекает два верхних элемента из стека, рассматривает первый элемент как индекс и вставляет второй элемент в хранилище контракта в соответствии с индексом;

♦ **энергозависимая память:**

- энергозависимая память аналогична ОЗУ в обычной компьютерной системе, необходима для выполнения любого кода или приложения и используется для хранения временных значений;
- контракт может использовать любой объем памяти во время выполнения, оплачивая его, и это пространство памяти очищается после завершения выполнения. Выходные данные во время выполнения можно перенести в постоянное хранилище и повторно использовать при будущих вызовах контракта;
- память, в отличие от хранилища, является непрерывным байтовым массивом. Он размещен в 256-битовых (32 байта) блоках;
- память начинается с нулевого объема (пустого места) и занимает место блоками по 32 байта;
- если не указано ключевое слово `memory`, то языки контрактов, такие как Solidity, будут объявлять переменные для сохранения в хранилище;
- память не может быть использована на глобальном уровне умного контракта, она доступна только в области определения методов (функций) контракта;
- аргументы функции почти всегда находятся в памяти;
- `MSTORE` и `MLOAD` — часто используемые инструкции для сохранения в память и чтения из памяти;

---

[8] Это значит, что по соображениям безопасности вы можете извлечь или сохранить только конкретное значение по известному вам ключу, но нет функции, которая позволяет перебрать подряд все значения и составить список ссылок на них.

- **стек:**
  - машина EVM основана на стеке типа LIFO (Last-In, First-Out, вошел последним — вышел первым), где стек используется для выполнения вычислений;
  - элементы стека также являются 256-битовыми словами, используемыми для имитации 256-битовых псевдорегистров. Они применяются для хранения локальных переменных типа «значение» и для передачи параметров в инструкции или функции, операций с памятью и других алгоритмических операций;
  - стек позволяет хранить максимум 1024 элемента и практически бесплатен для использования;
  - большинство операций стека ограничено вершиной стека. Выполнение очень похоже на то, как выполняется скрипт Bitcoin (см. *разд. 3.6.2*).

Когда EVM работает, и вместе с транзакцией поступает исполняемый байт-код, его полное вычислительное состояние может быть определено с помощью следующего кортежа:

```
block_state, transaction, message, code, memory, stack, pc, gas]
```

Теперь вы должны быть способны самостоятельно разобраться со всеми этими полями. Они содержат три типа памяти, которые мы обсуждали (поле `block_state` представляет глобальное состояние и предназначено для хранилища). Поле `PC` играет роль указателя на выполняемую команду из стека.

В Ethereum *двоичный интерфейс приложения* (Application Binary Interface, ABI) — это абстракция, которая не является частью основного протокола Ethereum, но используется для доступа к байт-коду в умном контракте. Хотя любой разработчик может определить собственный ABI для своих контрактов и взаимодействовать с ним, чтобы получить желаемый результат, проще использовать Solidity. Назначение ABI заключается в стандартизации взаимодействия:

- в ABI описано, как и какие функции следует вызывать внутри контрактов;
- двоичный формат, в котором информация должна передаваться в функции контракта в качестве входных данных;
- двоичный формат, в котором вы ожидаете результат выполнения функции после вызова этой функции.

Благодаря спецификациям ABI, две программы, написанные на двух разных языках, могут легко (хотя и не обязательно) взаимодействовать друг с другом.

## 4.5. Экосистема Ethereum

Мы изучили основные компоненты платформы Ethereum и постарались понять, как она работает. Есть некоторые присущие Ethereum ограничения, например:

- низкое быстродействие EVM — не рекомендуется использовать ее для больших вычислений;

- вычисления и хранение данных в блокчейне обходятся дорого. Желательно использовать вычисления вне цепочки и хранить данные в децентрализованных файловых системах IPFS/Swarm;
- проблема масштабирования системы по-прежнему актуальна. Существуют различные методы решения этой проблемы, но они, как правило, подходят только для конкретного бизнес-кейса, с которым вы имеете дело;
- скорее всего, будут преобладать частные (закрытые) блокчейны.

Теперь давайте бегло окинем взглядом экосистему, в которой обитает Ethereum.

## 4.5.1. Swarm

Это не только платформа распределенного хранения статических файлов в режиме одноранговой пиринговой сети, но и служба распространения файлов. Swarm (рой — *англ.*) обеспечивает адекватную децентрализацию и избыточное хранение данных блокчейна Ethereum, кодов приложений DApp и тому подобной информации. В отличие от сети WWW, загрузка файлов в Swarm не сосредоточена на одном веб-сервере. Swarm имеет нулевое время простоя, устойчива к DDOS-атакам и отказоустойчива.

## 4.5.2. Whisper

Протокол связи Whisper (шепот — *англ.*) позволяет децентрализованным приложениям (DApp) связываться друг с другом. Он предоставляет распределенные функции обмена сообщениями. Whisper поддерживает одноадресную, многоадресную и широковещательную рассылку сообщений.

## 4.5.3. Децентрализованное приложение (DApp)

DApp обычно состоит из двух частей кода: внешней (front-end, интерфейсная или клиентская часть) и внутренней (back-end, серверная часть). Внутренний код находится в умных контрактах и выполняется в сети блокчейна. Внешний пользовательский интерфейс может быть реализован на любом языке, например на HTML+JavaScript, при условии, что он может выполнять обращения к своей серверной части. Кроме того, интерфейс вместо централизованного веб-сервера может быть размещен в децентрализованном хранилище, таком как Swarm или IPFS. Компоненты пользовательского интерфейса будут кэшироваться в неком децентрализованном BitTorrent-подобном облаке и по мере необходимости загружаться браузером приложений. Как и в любом магазине приложений, в браузере можно просматривать распределенный каталог DApps. Конечный пользователь сможет установить любое подходящее приложение DApp в своем браузере.

## 4.5.4. Компоненты разработки

Существует очень много инструментов, которые используются для разработки децентрализованных приложений Ethereum и взаимодействия с ними. Далее упомяну-

ты несколько наиболее востребованных компонентов, но есть еще множество таких, которые вы можете найти и исследовать самостоятельно. Пока мы просто назовем их, а детальным изучением займемся в других главах:

- **Web3.js** — это очень важный элемент разработки DApps, представляет собой JavaScript-библиотеку, позволяющую использовать API Ethereum;
- **Truffle** — это среда разработки и фреймворк (набор строительных блоков для создания, компиляции, развертывания и тестирования приложений блокчейна);
- **Mist** — в предыдущих главах мы узнали, что для взаимодействия с блокчейном Bitcoin требуется кошелек, то же самое относится и к Ethereum. Кошелек Mist — это интерфейсное приложение, которое можно задействовать для подключения к блокчейну Ethereum. Используя кошелек Mist, можно создавать учетные записи, разрабатывать и развертывать контракты, передавать криптовалюту между счетами и просматривать детали транзакций.

  С технической точки зрения Mist зависит от клиента Geth (GoEthereum), с которым взаимодействует при выполнении операций.

## 4.6. Заключение

В этой главе мы изучили основные элементы платформы Ethereum и поняли принципы проектирования. Мы определили концептуальные различия между блокчейном Ethereum и блокчейном Bitcoin и поняли, как блокчейн Ethereum облегчает разработку и запуск различных приложений на одной платформе. Мы ближе познакомились с умными контрактами, которые децентрализованно выполняются на виртуальной машине.

В *главе 5* мы рассмотрим общие аспекты разработки блокчейн-приложений, а затем — в *главе 6* — перейдем к углубленному изучению разработки приложений для платформы Ethereum.

## 4.7. Рекомендуемые источники

- «Белая книга» Ethereum: **https://github.com/ethereum/wiki/wiki/White-Paper**.
- «Желтая книга» Ethereum: **https://ethereum.github.io/yellowpaper/paper.pdf**.
- Как работает Ethereum: **https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway–22d1df506369**.
- Дерево Патриции: **https://dl.acm.org/citation.cfm?id=321481**.
- Применение дерева Меркла в Ethereum: **https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/**.
- Газ и транзакции в Ethereum: **http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html**.
- Обзорное описание устройства Ethereum: **https://github.com/comaeio/porosity/wiki/Ethereum-Internals**.

ГЛАВА 5

# Разработка блокчейн-приложений

В предыдущих главах мы подробно рассмотрели, что такое блокчейн, и как работают блокчейны Bitcoin и Ethereum. Мы также изучили различные криптографические и математические алгоритмы и теоремы, которые использованы при создании технологии блокчейна.

В этой главе мы сначала разберемся, чем приложения блокчейна отличаются от обычных приложений, а затем будем учиться создавать приложения, основанные на блокчейне. Мы также рассмотрим настройку инфраструктуры, необходимой для разработки децентрализованных приложений.

## 5.1. Децентрализованные приложения

Популярность технологии блокчейна в основном обусловлена тем фактом, что она потенциально может решить различные реальные проблемы, поскольку обеспечивает бóльшую прозрачность и защиту от несанкционированного доступа по сравнению с обычными технологиями. Стартапы и активные члены блокчейн-сообщества разработали много разных вариантов использования блокчейна. Так вот, чтобы реализовать эти варианты, мы и создаем приложения, которые работают поверх блокчейна. Мы уже говорили, что приложения, которые взаимодействуют с блокчейном, называются *децентрализованными приложениями* или, сокращенно, просто DApps.

Чтобы лучше понять суть децентрализованных приложений, давайте еще раз вспомним, что такое блокчейн. Блокчейн, или распределенный реестр, — это, по сути, особый вид базы данных, в которой данные не хранятся на централизованном сервере, а копируются на все действующие узлы в сети. Кроме того, данные в цепочках блоков криптографически подписаны, что подтверждает аутентичность объекта, записавшего эти данные в цепочку блоков. Для внесения данных в блок-

чейн и извлечения их из блокчейна мы создаем специальные приложения. Такие приложения называются децентрализованными, поскольку опираются не на централизованную базу данных, а на децентрализованное хранилище. Для этих приложений нет единой точки отказа или контроля.

Давайте в качестве примера рассмотрим сценарий цепочки, в котором несколько поставщиков и логистических партнеров участвуют в процессе поставок промышленных товаров. Вот что мы должны сделать для использования блокчейна в цепочке поставок:

- настроить узлы блокчейна у каждого из поставщиков и логистических партнеров, чтобы они могли участвовать в процессе формирования общих данных;
- разработать интерфейс, чтобы все партнеры и пользователи системы могли хранить, извлекать, проверять и оценивать данные в блокчейне. Этот интерфейс будет использоваться производителями — для ввода информации о произведенных товарах, логистическими партнерами — для ввода информации о перевозке товара, сотрудниками склада — чтобы проверить, совпадает ли количество произведенных и доставленных товаров, и т. д. Таким инструментом для работы с блокчейном и может быть децентрализованное приложение.

Другим классическим примером децентрализованного приложения является основанная на блокчейне система голосования. Используя блокчейн для голосования, мы сможем сделать весь процесс намного более прозрачным и надежным, потому что каждый голос будет криптографически подписан. Для этого нужно создать приложение, которое получает список кандидатов для голосования и предоставляет простой интерфейс для подачи и регистрации голосов.

## 5.2. Создание блокчейн-приложений

Прежде чем мы перейдем к коду, давайте сначала разберемся с некоторыми основными понятиями, касающимися создания приложений для блокчейна. При разработке обычных программ мы привыкли к таким понятиям, как объекты, классы, функции и т. п. Однако, когда речь идет о блокчейн-приложениях, нам приходится оперировать еще и такими понятиями, как транзакции, счета и адреса, токены и кошельки, входы, выходы и балансы.

Прежде всего, при разработке приложения, основанного на блокчейне, нам нужно определить, как данные приложения будут сопоставляться с моделью данных блокчейна. Например, при разработке DApp на блокчейне Ethereum нам необходимо понять, как *состояние* приложения может быть представлено в терминах структуры данных Solidity, и как *поведение* приложения может быть выражено в терминах смарт-контрактов Ethereum. Поскольку мы знаем, что все данные в блокчейне криптографически подписаны закрытыми ключами пользователей, нам необходимо заранее решить, какие объекты в нашем приложении будут иметь идентификаторы или адреса, представленные в блокчейне. В обычных приложениях такой вопрос обычно не возникает, потому что данные не всегда подписаны. Для применения

блокчейна нам нужно определить, кто будет ставить цифровую подпись, и какие данные он подпишет. Например, в DApp для голосования каждый избиратель криптографически подписывает свой голос — это понятно и логично. Однако представьте себе сценарий, в котором нам нужно перенести существующее обычное приложение распределенной системы, хранящее данные в нескольких таблицах и базах данных SQL, в DApp на основе блокчейна Ethereum. В этом случае нам нужно определить, какие объекты и в какой таблице будут иметь свои идентификаторы (т. е. будут непосредственно привязаны к личностям с правом подписи. — *Ред.*), а какие объекты будут присоединены к другим идентификаторам.

## 5.2.1. Программирование приложений Bitcoin и Ethereum

В следующих нескольких разделах мы рассмотрим программирование приложений Bitcoin и Ethereum с использованием простых фрагментов кода для отправки транзакций. Цель этих упражнений — познакомиться с API двух самых популярных блокчейнов и общими практиками программирования для них. Для простоты мы воспользуемся общедоступными тестовыми сетями этих блокчейнов и напишем код на JavaScript. Причиной выбора JavaScript является то, что на момент подготовки этой книги уже существовали стабильные библиотеки JavaScript, доступные для обоих блокчейнов, поэтому вам будет легче понять сходства и различия в подходах, которые мы используем при написании кода. Фрагменты кода подробно объясняются после каждого логического шага и будут понятны, даже если читатель мало знаком с программированием на JavaScript.

На рис. 5.1 показано, как децентрализованное приложение взаимодействует с блокчейном.

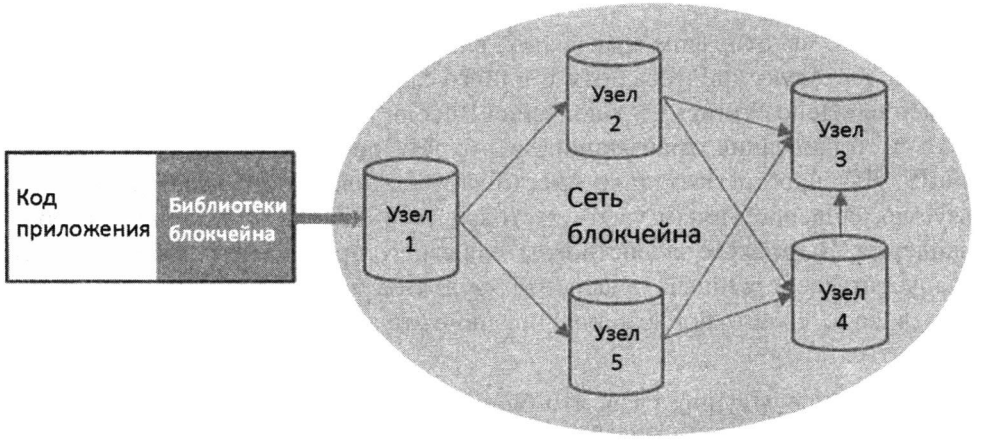



**Рис. 5.1. Взаимодействие приложения с блокчейном**

### 5.2.2. Библиотеки и инструменты

Функционирование блокчейна основано на различных криптографических алгоритмах и математических вычислениях. Прежде чем отправлять нашу транзакцию из приложения в блокчейн, мы должны ее подготовить. Подготовка транзакции включает в себя определение учетных записей и адресов, добавление необходимых параметров и значений к объектам транзакции, подписание с использованием закрытого ключа и некоторые другие действия. При разработке приложений лучше использовать стабильные и проверенные библиотеки для подготовки транзакций, а не писать код с нуля. Некоторые из стабильных библиотек как для Bitcoin, так и для Ethereum доступны с открытым исходным кодом, который можно использовать для подготовки и подписания транзакций, а также для отправки их в сеть блокчейна. В наших упражнениях с кодом мы воспользуемся библиотекой BitcoinJS — для взаимодействия с блокчейном Bitcoin и библиотекой web3.js — для взаимодействия с блокчейном Ethereum. Обе эти библиотеки написаны на языке JavaScript, доступны в виде пакетов node.js и могут быть загружены и интегрированы с помощью команд менеджера пакетов npm.

**ВАЖНОЕ ПРИМЕЧАНИЕ.** Примеры кода в этой главе подразумевают использование среды выполнения node.js. Это сделано для того, чтобы гарантировать, что код, который мы пишем как часть упражнения, имеет контейнер, в котором он может запускаться и взаимодействовать с другими предварительно упакованными библиотеками (модулями узлов). Приступая к рассмотрению приведенных здесь примеров, весьма полезно уже иметь базовые знания о разработке приложений в среде node.js. Так что мы рекомендуем начинающим разработчикам предварительно ознакомиться с учебными пособиями по работе с node.js и npm.

## 5.3. Взаимодействие с блокчейном Bitcoin

В этом разделе мы отправим транзакцию в общедоступную тестовую сеть Bitcoin и переведем тестовую криптовалюту с одного адреса на другой. Считайте, что это приложение «Hello World» для блокчейна Bitcoin. Как упоминалось ранее, для подготовки и подписания транзакций мы будем ориентироваться на библиотеку BitcoinJS. Для простоты отладки вместо запуска локального узла Bitcoin мы воспользуемся общедоступным узлом тестовой сети Bitcoin, запущенным сторонним провайдером. Вы можете задействовать для своего приложения любого провайдера или запустить собственный локальный узел. Все, что вам нужно сделать, — это указать в коде своего приложения предпочтительный узел, к которому следует подключиться.

Вспомните из предыдущих глав, что блокчейн Bitcoin предназначен прежде всего для обеспечения одноранговых платежей. Транзакция Bitcoin — это, в основном, просто перевод криптовалюты с одного адреса на другой. Давайте сделаем это программно.

На рис. 5.2 показано, как код приложения взаимодействует с блокчейном Bitcoin (это упрощенная схема, на которой не показана подробно архитектура службы обозревателя блоков Block Explorer).

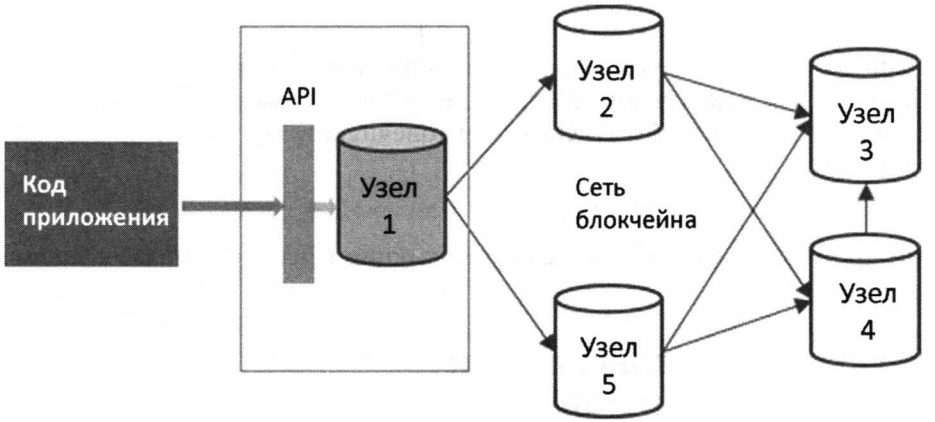



**Рис. 5.2. Взаимодействие приложения с блокчейном Bitcoin через API Block Explorer**

Следующие подзаголовки этого раздела являются шагами, которые необходимо выполнить в указанном порядке для отправки транзакции в тестовую сеть Bitcoin с использованием JavaScript.

## 5.3.1. Установка и инициализация библиотеки BitcoinJS в приложении node.js

Прежде чем мы вызовем зависящий от библиотеки BitcoinJS код для транзакций Bitcoin, мы установим и инициализируем эту библиотеку.

**ПРИМЕЧАНИЕ ПЕРЕВОДЧИКА.** Для быстрого ознакомления с установкой node.js и командами менеджера пакетов npm прочтите, например, следующие статьи на русском языке: **http://prgssr.ru/development/vvedenie-v-paketnyj-menedzher-npm-dlya-nachinayushih.html** и **https://inter-net.pro/soft/komandy-npm**.

Инициализируйте среду выполнения node.js с помощью команды: `npm init`. Создайте точку входа для нашего приложения — файл index.js и пользовательский модуль для вызова функций библиотеки BitcoinJS — файл btc.js. Импортируйте btc.js в index.js. Теперь мы готовы выполнить следующие шаги.

Сначала установим модуль узла BitcoinJS:

```
npm install --save bitcoinjs-lib
```

Затем в нашем модуле Bitcoin btc.js подключим библиотеку BitcoinJS, используя ключевое слово `require`:

```
var btc = require('bitcoinjs-lib');
```

Теперь мы можем использовать эту переменную `btc` для вызова библиотечных функций библиотеки BitcoinJS. Кроме того, как часть процесса инициализации, мы инициализируем еще пару переменных:

- целевую сеть для подключения — мы используем тестовую сеть Bitcoin:

```
var network = btc.networks.testnet;
```

- конечную точку API публичного узла — для получения и публикации транзакций. Здесь мы используем API Block Explorer тестовой сети Bitcoin (обратите внимание, что вы можете заменить эту конечную точку API на другую по вашему выбору):

```
var blockExplorerTestnetApiEndpoint = 'https://testnet.blockexplorer.com/api/';
```

На этом этапе все готово для создания транзакции Bitcoin с использованием среды node.js.

## 5.3.2. Создание пары ключей для отправителя и получателя

Первое, что нам понадобится, — это пары ключей для отправителя и получателя. Они будут идентифицировать пользователей в блокчейне. Итак, давайте сначала создадим две пары ключей — для Алисы и для Боба:

```
var getKeys = function () {
    var aliceKeys = btc.ECPair.makeRandom({
        network: network
    });
    var bobKeys = btc.ECPair.makeRandom({
        network: network
    });
    var alicePublic = aliceKeys.getAddress();
    var alicePrivate = aliceKeys.toWIF();
    var bobPublic = bobKeys.getAddress();
    var bobPrivate = bobKeys.toWIF();
    console.log(alicePublic, alicePrivate, bobPublic, bobPrivate);
};
var getKeys = function () {
    var aliceKeys = btc.ECPair.makeRandom({
        network: network
    });
    var bobKeys = btc.ECPair.makeRandom({
        network: network
    });
    var alicePublic = aliceKeys.getAddress();
    var alicePrivate = aliceKeys.toWIF();
    var bobPublic = bobKeys.getAddress();
    var bobPrivate = bobKeys.toWIF();
    console.log(alicePublic, alicePrivate, bobPublic, bobPrivate);
};
```

В этом фрагменте кода мы использовали класс `ECPair` библиотеки BitcoinJS и вызвали метод `makeRandom` для создания случайных пар ключей для тестовой сети — обратите внимание на параметр, переданный для типа сети.

Теперь, когда мы создали две пары ключей, давайте применим их для отправки биткойнов от одного пользователя другому. Алиса и Боб — это любимые персонажи во всех книгах по криптографии. Однако каждый раз, когда мы видим пример криптографического обмена, обычно Алиса что-то шифрует, подписывает и отправляет Бобу. Мы считаем, что у Боба накопился большой долг перед Алисой, поэтому мы поможем Бобу погасить часть этого долга. В нашем примере именно Боб отправляет тестовые биткойны Алисе.

## 5.3.3. Получение тестовых биткойнов

Итак, мы решили, что Боб будет выступать в качестве отправителя. Прежде чем он отправит биткойны Алисе, они у него должны быть (он должен владеть ими). Поскольку мы знаем, что эта демонстрационная транзакция работает в тестовой сети Bitcoin, реальных денег не потребуется, но нам все равно нужно, чтобы в кошельке Боба имелись тестовые биткойны. В Интернете существует много веб-сайтов, на которых размещена простая веб-форма, позволяющая получать адреса тестовых сетей Bitcoin, а затем отправлять им тестовые биткойны. Эти сервисы называются «Bitcoin testnet faucet[1]», и поиском по этому названию вы получите очень много результатов. Мы не приводим готовый список и не рекомендуем какой-либо конкретный сервис, поскольку они, как правило, не являются постоянными. Как только провайдер тестовой сети исчерпает свои тестовые монеты или не хочет более размещать услугу, он закрывает ее. Но при этом постоянно появляются новые сервисы. Список некоторых из этих сервисов также доступен на странице: **https://en.bitcoin.it/wiki/Testnet**.

Еще один способ получить биткойны для тестовой сети — разместить локальный биткойн-узел, подключенный к тестовой сети, и добыть биткойны самостоятельно. Майнинг блоков в тестовой сети Bitcoin не так сложен, как в основной сети. Этот метод хорошо подходит для следующего уровня, когда вы создаете прикладное приложение Bitcoin, и вам нужно часто его тестировать. Вместо того, чтобы просить у кого-то тестовые монеты каждый раз, когда вы хотите протестировать свое приложение, вы можете просто добыть их самостоятельно.

Но для нашего несложного примера мы все же получим несколько готовых биткойнов из тестовой сети. В приведенном ранее фрагменте кода значение в переменной `bobPublic` является адресом Bitcoin. Когда мы запустили этот фрагмент, он сгенерировал последовательность:

```
msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2
```

в качестве адреса Боба. Это также открытый ключ Боба, кодированный по Base58. Мы введем это значение в одну из веб-форм Bitcoin testnet faucet, а взамен получим

[1] Faucet — сливной краник в какой-либо емкости, например в пивной бочке. — *англ.*

идентификатор транзакции. Если мы затем введем этот идентификатор транзакции в любом из сервисов просмотра тестовых сетей Bitcoin, то увидим, что какой-то другой адрес отправил несколько тестовых биткойнов на адрес Боба, который мы указали в форме.

## 5.3.4. Получение неизрасходованных остатков

Теперь, когда мы знаем, что в кошельке Боба есть несколько тестовых биткойнов, мы можем их потратить и передать Алисе через биткойн-транзакцию. Давайте вспомним, что биткойн-транзакции оперируют входящими и исходящими суммами. Вы можете потратить неизрасходованные остатки, добавив их в качестве входной суммы в следующие транзакции. Для этого сначала нужно запросить в сети информацию о неизрасходованных входящих остатках отправителя. Сейчас мы сделаем это для адреса Боба, используя API Block Explorer. Чтобы получить неизрасходованные остатки, мы отправим HTTP-запрос в конечную точку UTXO с адресом Боба `msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2`:

```
var getOutputs = function () {
    var url = blockExplorerTestnetApiEndpoint + 'addr/' + msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2 +
'/utxo';
    return new Promise(function (resolve, reject) {
        request.get(url, function (err, res, body) {
            if (err) {
                reject(err);
            }
            resolve(body);
        });
    });
};
```

В этом фрагменте кода мы использовали модуль запроса node.js для отправки HTTP-запросов с помощью приложения node.js. Не стесняйтесь использовать вашу любимую библиотеку или модуль `http`. Этот фрагмент представляет собой функцию JavaScript. Функция возвращает обещание (`Promise`), которое разрешается в теле ответа метода API. Вот как выглядит этот ответ:

```
[
    {
        address: 'msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2',
        txid: 'db2e5966c5139c6e937203d567403867643482bbd9a6624752bbc583ca259958',
        vout: 0,
        scriptPubKey: '76a914806094191cbd4fcd8b4169a70588adc51dc02d6888ac',
        amount: 0.99992,
        satoshis: 99992000,
        height: 1258815,
        confirmations: 1011
    },
```

```
    {
      address: 'msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2',
        txid: '5b88d5fc4675bb86b0a3a7fc5a36df9c425c3880a7453e3afeb4934e6d1d928e',
        vout: 1,
        scriptPubKey: '76a914806094191cbd4fcd8b4169a70588adc51dc02d6888ac',
        amount: 0.99998,
        satoshis: 99998000,
        height: 1258814,
        confirmations: 1012
    }
]
```

Тело ответа, возвращаемое при вызове функции, представляет собой массив JSON с двумя объектами. Каждый из этих объектов показывает неизрасходованный остаток для Боба. Каждый вывод имеет поле `txid`, которое является идентификатором транзакции. В поле указан идентификатор, сумма выхода и значение `vout`, которое означает последовательность или порядковый номер выхода в этой транзакции. В объектах JSON есть и другая информация, но она не будет использоваться в процессе подготовки транзакции.

Первый объект в массиве JSON говорит нам, что адрес тестовой сети Bitcoin

`msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2`

содержит неизрасходованные сатоши в количестве `99992000`, поступившие из транзакции:

`db2e5966c5139c6e937203d567403867643482bbd9a6624752bbc583ca259958`

с индексом `0`.

Точно так же второй объект массива JSON представляет `99998000` неизрасходованных сатоши, поступивших из транзакции:

`5b88d5fc4675bb86b0a3a7fc5a36df9c425c3880a7453e3afeb4934e6d1d928e`

с индексом `1`.

Не забывайте, что `msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2` — это адрес Боба в тестовой сети Bitcoin, который мы создали на втором шаге. Теперь мы знаем, что у Боба есть определенное количество монет, которые он может потратить на новую транзакцию.

## 5.3.5. Подготовка биткойн-транзакции

Следующим шагом является подготовка транзакции в биткойнах, в которой Боб может отправить тестовые монеты Алисе. Подготовка транзакции в основном состоит из определения входов, выходов и суммы.

Как мы знаем из предыдущего шага, у Боба есть два неизрасходованных остатка для определенного адреса в тестовой сети. Остатки представлены нам в виде элементов массива JSON. Давайте потратим первый элемент массива. Добавим его как вход в нашу транзакцию:

```
var utxo = JSON.parse(body.toString());
var transaction = new btc.TransactionBuilder(network);
transaction.addInput(utxo[0].txid, utxo[0].vout);
```

В этом фрагменте кода сначала мы проанализировали ответ, полученный от предыдущего вызова API, чтобы получить неизрасходованные входящие остатки Боба.

Затем создали объект построителя транзакций для тестовой сети Bitcoin, используя библиотеку BitcoinJS.

В последней строке мы определили вход транзакции. Обратите внимание, что этот вход ссылается на элемент с индексом `0` массива `utxo`, который мы получили в вызове API на предыдущем шаге. В качестве входных параметров мы передали в метод `transaction.addInput` идентификатор транзакции `txid` и индекс остатка `vout`.

По сути, мы здесь показываем, что хотим потратить, и откуда мы это получили.

Далее добавляем выходные данные транзакции. Здесь мы говорим, как мы хотим потратить то, что добавили на вход. Допустим, Боб хочет послать 99990000 сатоши Алисе. В следующей строке мы добавили вывод транзакции, вызвав метод `addOutput` для объекта построителя транзакций, и передали целевой адрес и сумму:

```
transaction.addOutput(alicePublic, 99990000);
```

Обратите внимание, что в качестве первого параметра функции мы указали адрес тестовой сети Bitcoin Алисы.

Хотя в этом примере транзакции мы использовали только один вход и один выход, транзакция может иметь несколько входов и выходов. Важно отметить, что общая сумма на входах не должна быть меньше общей суммы на выходах. В большинстве случаев сумма входов немного больше, чем сумма выходов, а разница между суммами предлагается майнеру в качестве вознаграждения, чтобы он был заинтересован скорее добавить транзакцию в блок.

В этой транзакции у нас есть 2000 сатоши в качестве комиссии за транзакцию, которая представляет собой разницу между суммой входа (99992000) и суммой выхода (99990000). Обратите внимание, что нам не нужно создавать какие-либо выходные данные для комиссии за транзакцию. Разница между суммой входа и выхода автоматически принимается майнером в качестве такой комиссии.

Также обратите внимание, что мы не можем частично потратить неизрасходованные остатки. Если с неизрасходованным остатком связано $X$ биткойнов, то мы должны потратить все $X$ биткойнов при добавлении этого остатка в качестве входных данных транзакции. Таким образом, если Боб не хочет передать Алисе все 99990000 сатоши, то мы должны *в явном виде вернуть остаток* Бобу! Для этого нужно добавить еще один вывод в транзакцию с суммой, равной разнице между общим неизрасходованным остатком, и суммой, которую Боб хочет отправить Алисе. Только второй вывод транзакции Боб должен адресовать самому себе.

## 5.3.6. Подписание входных данных транзакции

Теперь, когда мы определили в транзакции входные и выходные данные, нам нужно подписать входные данные, используя ключи Боба. Следующая строка кода вызывает функцию `sign` конструктора транзакций для криптографического подписания транзакции с использованием закрытого ключа Боба, но в качестве входного параметра конструктор транзакций принимает объект пары ключей:

```
transaction.sign(0, bobKeys);
```

Обратите внимание, что в качестве параметров функция `transaction.sign` принимает индекс входа и полную пару ключей. В этой транзакции, поскольку у нас есть только один вход, индекс равен `0`.

На данном этапе наша транзакция подготовлена и подписана.

## 5.3.7. Создание HEX-кода транзакции

Теперь мы создадим шестнадцатеричную строку из объекта транзакции:

```
var transactionHex = transaction.build().toHex();
```

Результатом этой операции является строка, которая представляет нашу подготовленную транзакцию. Такой шаг необходим, потому что API отправки транзакции принимает необработанную транзакцию в виде строки.

## 5.3.8. Трансляция транзакции в сеть

Наконец, мы берем значение шестнадцатеричной строки, сгенерированное на последнем шаге, и отправляем его на публичный узел `testnet` с помощью API:

```
var txPushUrl = blockExplorerTestnetApiEndpoint + 'tx/send';
request.post({
    url: txPushUrl,
        json: {
            rawtx: transactionHex
        }
    }, function (err, res, body) {
        if (err) console.log(err);

        console.log(res);
        console.log(body);
    });
```

Если транзакция принята публичным узлом, в ответ на этот вызов API мы получим идентификатор транзакции:

```
{
    txid: "db2e5966c5139c6e937203d567403867643482bbd9a6624752bbc583ca259958"
}
```

Теперь, когда у нас есть идентификатор для нашей транзакции, мы можем найти его при помощи любого обозревателя блоков тестовой сети, чтобы узнать, обработан ли блок, содержащий транзакцию, и сколько подтверждений у него есть.

Далее представлен полный код функции для отправки тестовой транзакции Bitcoin с использованием JavaScript. Входными параметрами функции являются пары ключей тестовой сети Bitcoin, которые мы создали на первом шаге:

```
var createTransaction = function (aliceKeys, bobKeys) {
    getOutputs(bobKeys.getAddress()).then(function (res) {
        var utxo = JSON.parse(res.toString());
        var transaction = new btc.TransactionBuilder(network);
        transaction.addInput(utxo[0].txid, utxo[0].vout);
        transaction.addOutput(alicekeys.getAddress(), 99990000);
        transaction.sign(0, bobKeys);
        var transactionHex = transaction.build().toHex();
        var txPushUrl = blockExplorerTestnetApiEndpoint + 'tx/send';
        request.post({
            url: txPushUrl,
            json: {
                rawtx: transactionHex
            }
        }, function (err, res, body) {
            if (err) console.log(err);

            console.log(res);
            console.log(body);
        });
    });
};
```

* * *

В этом разделе мы узнали, как программно отправить транзакцию в тестовую сеть Bitcoin. Аналогично мы можем отправлять транзакции в основную сеть Bitcoin, указывая основную сеть в качестве цели в функциях библиотеки и в конечных точках API. Мы также использовали API запросов для получения неизрасходованных остатков адреса Bitcoin. Эти функции можно задействовать при создании простого приложения кошелька для получения информации об остатках и управления адресами и транзакциями Bitcoin.

## 5.4. Программное взаимодействие с Ethereum — отправка транзакций

С точки зрения разработки приложений блокчейн Ethereum может предложить гораздо больше по сравнению с Bitcoin. Ключевой особенностью блокчейна Ethereum является возможность выполнения умных контрактов, позволяющая разработчикам

создавать децентрализованные приложения. В этом разделе мы узнаем, как программно взаимодействовать с блокчейном Ethereum с помощью JavaScript. Мы рассмотрим основные аспекты программирования приложений Ethereum — от простых транзакций до создания и вызова умных контрактов.

Как и в случае взаимодействия с блокчейном Bitcoin в предыдущем разделе, мы воспользуемся библиотекой JavaScript и тестовой сетью для взаимодействия с Ethereum, а также библиотекой web3 JavaScript для Ethereum. Эта библиотека охватывает множество API-интерфейсов Ethereum JSON RPC и предоставляет простые в применении функции для создания децентрализованных приложений Ethereum на основе JavaScript. В процессе подготовки этой книги мы использовали новую версию библиотеки web3 JavaScript, совместимую с версией 1.0.0-beta.28 этой библиотеки.

Для тестирования мы воспользуемся тестовой сетью Ropsten блокчейна Ethereum.

Для простоты мы снова обратимся к общедоступному тестовому узлу, чтобы нам не приходилось, выполняя наши фрагменты кода, запускать локальный узел. Однако все фрагменты кода в равной мере должны работать и с локальным узлом. Мы используем API Ethereum сервиса Infura. Это сервис, который предоставляет открытые узлы Ethereum, чтобы разработчики могли легко тестировать свои приложения. Перед тем как задействовать API Infura, необходимо пройти несложную бесплатную регистрацию, поэтому перейдите по адресу: **https://infura.io** и зарегистрируйтесь. После регистрации вы получите ключ API. С помощью этого ключа мы теперь сможем обращаться к API Infura.

На рис. 5.3 показано, как код приложения взаимодействует с блокчейном Ethereum (это всего лишь схематичный набросок, на котором не показана внутренняя архитектура сервиса Infura).

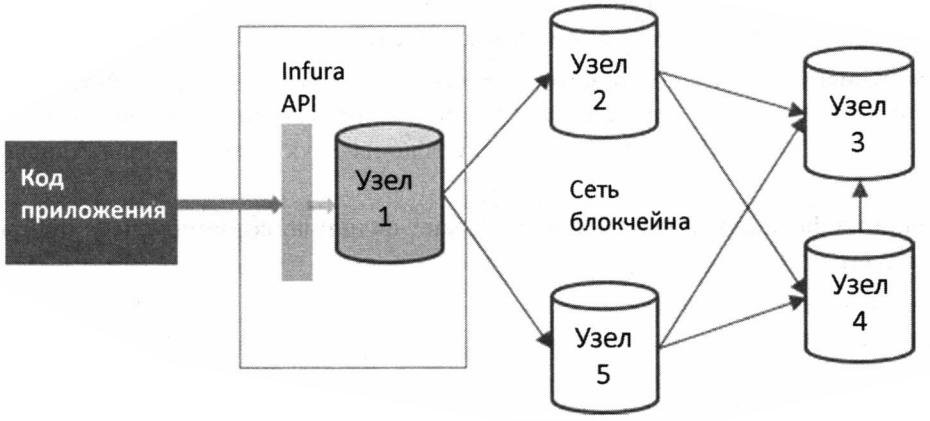



**Рис. 5.3.** Приложение взаимодействует с блокчейном Ethereum через API Infura

Далее мы опишем шаги, которые необходимо выполнить в указанном порядке для отправки транзакции в тестовую сеть Ethereum Ropsten с использованием JavaScript.

## 5.4.1. Настройка библиотеки и подключения

Сначала установим библиотеку web3 в нашу среду выполнения node.js. Обратите внимание на конкретную версию библиотеки, упомянутую в команде установки. Это связано с тем, что в версии 1.0.0 библиотеки доступно еще несколько API и функций, которые уменьшают зависимость от других внешних пакетов:

```
npm install web3@1.0.0-beta.28[2]
```

Затем мы подключаем библиотеку в нашем модуле nodejs Ethereum, используя ключевое слово `require`:

```
var Web3 = require('web3');
```

Теперь у нас есть ссылка на библиотеку web3, но нам нужно создать экземпляр HTTP-провайдера, прежде чем мы сможем ее использовать. Следующая строка кода создает новый экземпляр Web3 и определяет узел тестовой сети Ethereum Ropsten, размещенный в Infura, в качестве провайдера для этого экземпляра:

```
var web3 = new Web3(new Web3.providers.HttpProvider('https://ropsten.infura.io/<ваш ключ Infura
                                                    API, полученный при регистрации>'));
```

## 5.4.2. Настройка счетов Ethereum

Теперь, когда наши инструменты готовы к работе, давайте отправим транзакцию в блокчейн Ethereum. В этой транзакции мы переведем немного эфира с одного счета на другой. Напомним, что Ethereum не использует модель UTXO, а опирается на традиционную модель счетов и балансов.

По сути, блокчейн Ethereum управляет состоянием и активами с точки зрения счетов и балансов, как это делают банки. Здесь нет входов и выходов транзакций. Вы можете просто отправить эфир с одного счета на другой, и Ethereum позаботится об обновлении состояний этих счетов на всех узлах.

Чтобы отправить транзакцию, которая переводит эфир со счета на счет, нам сначала потребуется создать пару счетов Ethereum (или *учетных записей*, что в данном случае одно и то же). Давайте начнем с создания двух счетов — для Алисы и для Боба.

Следующий фрагмент кода дважды вызывает функцию создания счета библиотеки web3 и создает два счета:

```
var createAccounts = function () {
    var aliceKeys = web3.eth.accounts.create();
    console.log(aliceKeys);
    var bobKeys = web3.eth.accounts.create();
    console.log(bobKeys);
};
```

[2] На момент подготовки перевода была доступна версия 1.0.0-beta.36.

А вот вывод, который мы получаем в окне консоли после запуска предыдущего фрагмента:

```
{
    address: '0xAff9d328E8181aE831Bc426347949EB7946A88DA',
    privateKey: '0x9fb71152b32cb90982f95e2b1bf2a5b6b2a53855eacf59d132a2b7f043cfddf5',
    signTransaction: [Function: signTransaction],
    sign: [Function: sign],
    encrypt: [Function: encrypt]
}
{
    address: '0x22013fff98c2909bbFCcdABb411D3715fDB341eA',
    privateKey: '0xc6676b7262dab1a3a28a781c77110b63ab8cd5eae2a5a828ba3b1ad28e9f5a9b',
    signTransaction: [Function: signTransaction],
    sign: [Function: sign],
    encrypt: [Function: encrypt]
}
```

Как вы можете видеть, наряду с адресами и закрытыми ключами, вывод каждого вызова функции создания счета также включает в себя несколько функций. Сейчас мы сосредоточимся на адресе и закрытом ключе возвращаемых объектов. Адрес — это хэш открытого ключа по алгоритму Keccak-256. Комбинация адреса и закрытого ключа представляет собой *учетную запись* (адрес, счет) в блокчейне Ethereum. Вы можете отправить эфир на чей-либо счет, также можете потратить его, используя закрытый ключ соответствующего счета.

## 5.4.3. Получение тестового эфира на счет отправителя

Чтобы создать транзакцию Ethereum, которая передает эфир со счета на счет, нам понадобится эфир на одном из счетов. Как вы помните из *разд. 5.3.3* про Bitcoin, мы воспользовались специальным сервисом источника монет, чтобы получить несколько тестовых биткойнов на сгенерированный нами адрес. Теперь мы сделаем то же самое для Ethereum. Мы ориентируемся на тестовую сеть Ropsten для Ethereum, поэтому будем искать в Интернете источник монет Ropsten. В нашем примере мы отправили адрес Алисы, сгенерированный в предыдущем фрагменте кода, в тестовый источник Ropsten и получили на этот адрес три эфира.

После получения денег на адрес Алисы, давайте проверим баланс этого адреса, чтобы убедиться, действительно ли там есть эфир. Хотя мы можем проверить баланс этого адреса с помощью любого из обозревателей Ethereum, сейчас мы сделаем это с помощью кода. Следующий фрагмент кода вызывает функцию `getBalance`, передавая адрес Алисы в качестве входного параметра:

```
var getBalance = function () {
web3.eth.getBalance('0xAff9d328E8181aE831Bc426347949EB7946A88DA').then(console.log);
};
```

В ответ мы получаем остаток на счете Алисы. С виду это огромное число, но на самом деле это значение баланса в Wei — самой маленькой денежной единице

эфира. Один эфир равен $10^{18}$ Wei. Итак, следующее значение равно трем эфирам, которые мы получили из тестового сетевого источника:

```
3000000000000000000
```

## 5.4.4. Подготовка транзакции Ethereum

Теперь, когда на счете у Алисы есть тестовые деньги, давайте создадим транзакцию Ethereum, чтобы отправить часть этих денег Бобу. Напомним, что в данном случае нет входов, выходов и запросов UTXO, потому что Ethereum использует систему счетов и балансов. Таким образом, все, что нам нужно сделать в транзакции, — это указать адрес `from` (адрес отправителя), адрес `to` (адрес получателя), количество отправляемого эфира и некоторые другие атрибуты.

Также вспомним, что в случае транзакции Bitcoin нам не нужно было указывать комиссию за транзакцию, однако в случае транзакции Ethereum нам необходимо заполнить два связанных поля. Первое — лимит газа (`gas`), а второе — цена газа (`gasPrice`). В *главе 4* мы говорили, что *газ* — это единица платы за транзакции, которую мы должны заплатить сети Ethereum, чтобы подтвердить наши транзакции и добавить их в блоки. Цена газа — это количество эфира (в GWei), которое мы согласны заплатить за единицу газа. Максимальная плата, которую мы разрешаем взять за транзакцию, является произведением количества газа и цены на газ.

Итак, для нашей учебной транзакции мы определяем объект JSON с несколькими полями. Поле `from` содержит адрес Алисы, поле `to` — адрес Боба, а `value` равно одному эфиру в номинале Wei. Цена на газ, которую мы установили, составляет 20 GWei, а максимальное количество газа, которое мы готовы заплатить за эту сделку, составляет 42 000.

Обратите внимание, что мы оставили поле `data` пустым (мы вернемся к этому позже, в *разд. 5.5.3*):

```
{
    from: "0xAff9d328E8181aE831Bc426347949EB7946A88DA",
    gasPrice: "20000000000",
    gas: "42000",
    to: '0x22013fff98c2909bbFCcdABb411D3715fDB341eA',
    value: "1000000000000000000",
    data: ""
}
```

## 5.4.5. Подписание транзакции

Итак, мы создали объект транзакции с необходимыми полями и значениями, и теперь нам нужно подписать его, используя закрытый ключ учетной записи отправителя. В нашем случае отправителем является Алиса, поэтому мы воспользуемся закрытым ключом Алисы для подписания транзакции. Это послужит криптографическим доказательством того, что именно Алиса тратит эфир со своего счета:

```
var signTransaction = function () {
    var tx = {
        from: "0xAff9d328E8181aE831Bc426347949EB7946A88DA",
        gasPrice: "20000000000",
        gas: "42000",
        to: '0x22013fff98c2909bbFCcdABb411D3715fDB341eA',
        value: "1000000000000000000",
        data: ""
    };

    web3.eth.accounts.signTransaction(tx,
                            '0x9fb71152b32cb90982f95e2b1bf2a5b6b2a53855eacf59d132a2b7f043cfddf5')
    .then(function(signedTx){
        console.log(signedTx.rawTransaction);
    });
};
```

В этом фрагменте кода вызывается функция `signTransaction` с объектом транзакции, который мы создали на предыдущем шаге, и с закрытым ключом Алисы, который мы получили при создании учетной записи Алисы. Далее приведен вывод, который мы получаем при запуске фрагмента кода:

```
{
    messageHash: '0x91b345a38dc728dc06a43c49b92a6ac1e0e6d614c432a6dd37d809290a25aa6b',
    v: '0x2a',
    r: '0x14c20901a060834972a539d7b8ad1f23161c2144a2b66fbf567e37e963d64537',
    s: '0x3d2a0a818633a11832a5c48708a198af909eaf4884a7856c9ac9ed216d9b029c',
    rawTransaction: '0xf86c018504a817c80082a4109422013fff98c2909bbfccdabb411d3715fdb341
    ea880de0b6b3a7640000802aa014c20901a060834972a539d7b8ad1f23161c2144a2b66fbf567e37e963d64537a03d
    2a0a818633a11832a5c48708a198af909eaf4884a7856c9ac9ed216d9b029c'
}
```

В выводе функции `signTransaction` мы получаем объект JSON с несколькими свойствами. Важным значением для нас является `rawTransaction`. Это шестнадцатеричное представление подписанной транзакции. Вспомните, как мы создали шестнадцатеричную строку транзакции Bitcoin в *разд. 5.3.7*.

## 5.4.6. Отправка транзакции в сеть Ethereum

Нам осталось сделать последний шаг — просто отправить подписанную необработанную транзакцию общедоступному узлу тестовой сети Ethereum, который мы указали в качестве поставщика для нашего объекта web3.

Следующий код вызывает функцию `sendSignedTransaction` для отправки необработанной транзакции в тестовую сеть Ethereum. Входной параметр — это значение строки `rawTransaction`, которое мы получили на предыдущем шаге в функции подписания транзакции:

```
web3.eth.sendSignedTransaction(signedTx.rawTransaction).then(console.log);
```

Обратите внимание на использование `then` во фрагменте кода. Это важно, поскольку транзакция Ethereum проходит через несколько состояний после отправки, и библиотека web3 поддерживает различные *уровни завершенности* при работе с транзакциями. В этой функции происходит вызов функции отправки транзакции в сеть, а оператор `then` выполняется, когда создана квитанция транзакции, и транзакция завершена.

Через несколько секунд, когда обещание JavaScript успешно разрешилось, мы получаем следующий вывод в консоль:

```
{
 blockHash: '0x26f1e1374d11d4524f692cdf1ce3aa6e085dcc181084642293429eda3954d30e',
 blockNumber: 2514764,
 contractAddress: null,
 cumulativeGasUsed: 125030,
 from: '0xaff9d328e8181ae831bc426347949eb7946a88da',
 gasUsed: 21000,
 logs: [],
 logsBloom: '0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000',
 status: '0x1',
 to: '0x22013fff98c2909bbfccdabb411d3715fdb341ea',
 transactionHash: '0xd3f45394ac038c44c4fe6e0cdb7021fdbd672eb1abaa93eb6a1828df5edb6253',
 transactionIndex: 3
}
```

Как видите, вывод содержит много полезной информации. Наиболее важной частью является поле `transactionHash`, которое содержит хэш транзакции и служит идентификатором транзакции в сети. Также мы получаем `blockHash` — идентификатор блока, в который была включена наша транзакция. Наряду с этим, мы также получаем информацию о том, сколько газа было использовано для обработки транзакции. Если затраченный газ меньше максимального газа, указанного нами при создании транзакции, остаток газа отправляется обратно на счет отправителя.

В этом разделе мы отправили простую транзакцию в блокчейн Ethereum с помощью JavaScript. Но это только начало разработки приложений Ethereum. В следующем разделе мы расскажем, как программно создавать и вызывать умные контракты.

## 5.5. Создание умного контракта Ethereum

Здесь мы продолжим наше упражнение по программированию для платформы Ethereum и создадим простой умный контракт, используя прежнюю библиотеку JavaScript web3 и API сервиса Infura.

Поскольку ни одно руководство для начинающих программистов не обходится без программы «Hello World», умный контракт, который мы собираемся создать, будет при вызове возвращать строку «Hello World».

Создание умного контракта представляет собой отправку в блокчейн Ethereum специальной транзакции. В таких транзакциях не упоминается адрес получателя, а владельцем умного контракта является отправитель (точнее, его адрес), указанный в транзакции.

## 5.5.1. Подготовка

Приступая к новому упражнению, мы предполагаем, что библиотека JavaScript web3 установлена в среде выполнения node.js, и вы зарегистрировались в сервисе Infura, как было рекомендовано в *разд. 5.4*.

Далее описана последовательность действий по созданию умного контракта Ethereum с использованием JavaScript.

## 5.5.2. Программируем умный контракт

Напомним, что умные контракты Ethereum чаще всего пишут на языке программирования Solidity. Хотя библиотека JavaScript web3 поможет нам развернуть контракт в сети Ethereum, нам все равно придется написать и скомпилировать наш умный контракт на языке Solidity, прежде чем мы отправим его в сеть с помощью web3. Итак, давайте сначала создадим учебный контракт.

У разработчиков есть широкий выбор инструментов для программирования на Solidity. Большинство основных IDE и редакторов кода имеют плагины Solidity для редактирования и написания умных контрактов. Существует также веб-редактор Solidity под названием Remix. Его можно бесплатно использовать по адресу: **https://remix.ethereum.org/**. Remix предоставляет очень простой интерфейс для написания и компиляции умных контрактов в вашем браузере. В этом упражнении мы воспользуемся Remix для кодирования и тестирования нашего умного контракта, а затем отправим контракт в сеть Ethereum с помощью библиотеки JavaScript web3 и службы Infura API.

Следующий фрагмент кода написан на языке Solidity и представляет собой простой умный контракт, который возвращает строку «Hello World» из своей функции `Hello`. Он также имеет конструктор, который устанавливает значение возвращаемого сообщения:

```
pragma solidity ^0.4.0;
contract HelloWorld {
    string message;
    function HelloWorld(){
        message = "Hello World!";
    }
```

```
    function Hello() constant returns (string) {
        return message;
    }
}
```

Откройте редактор Remix, создайте новый файл HelloWorld.sol и вставьте этот код в окно редактора. На вкладке **Compile** (Компиляция) выберите версию компилятора **0.4.0** и нажмите кнопку **Start to compile** (Начать компиляцию). Для просмотра выходных данных нажмите кнопку **Details** (Подробности). Обратите внимание, что по умолчанию редактор Remix предназначается для компиляции умных контрактов под виртуальную машину JavaScript и использует тестовую учетную запись с некоторым количеством эфира на балансе для целей тестирования. Когда мы нажимаем кнопку **Start to compile**, в среде виртуальной машины JavaScript создается контракт для выбранной учетной записи. На рис. 5.4 показано, как выглядит наш пример умного контракта в редакторе Remix, а на рис. 5.5 — как выглядят выходные данные после окончания компиляции.

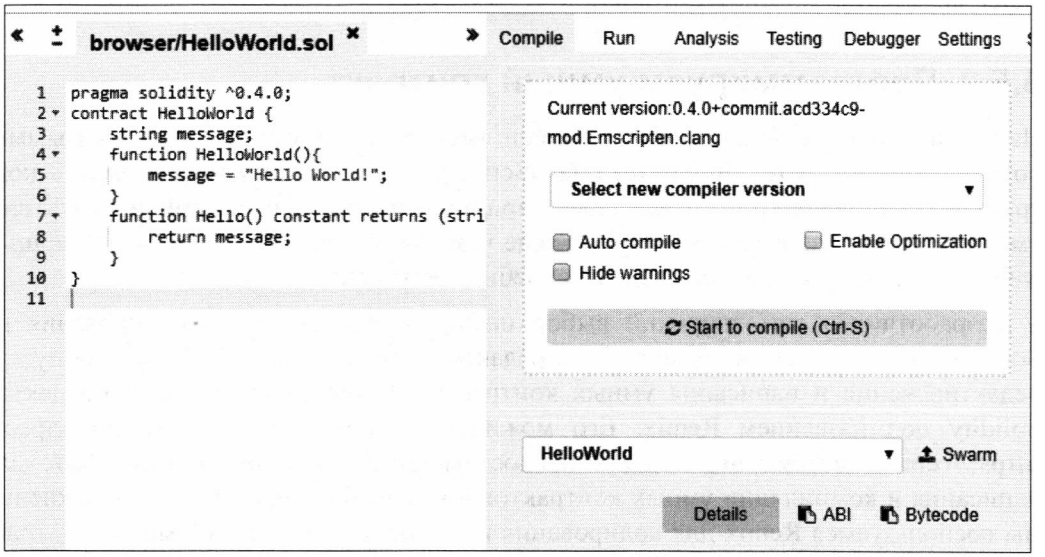


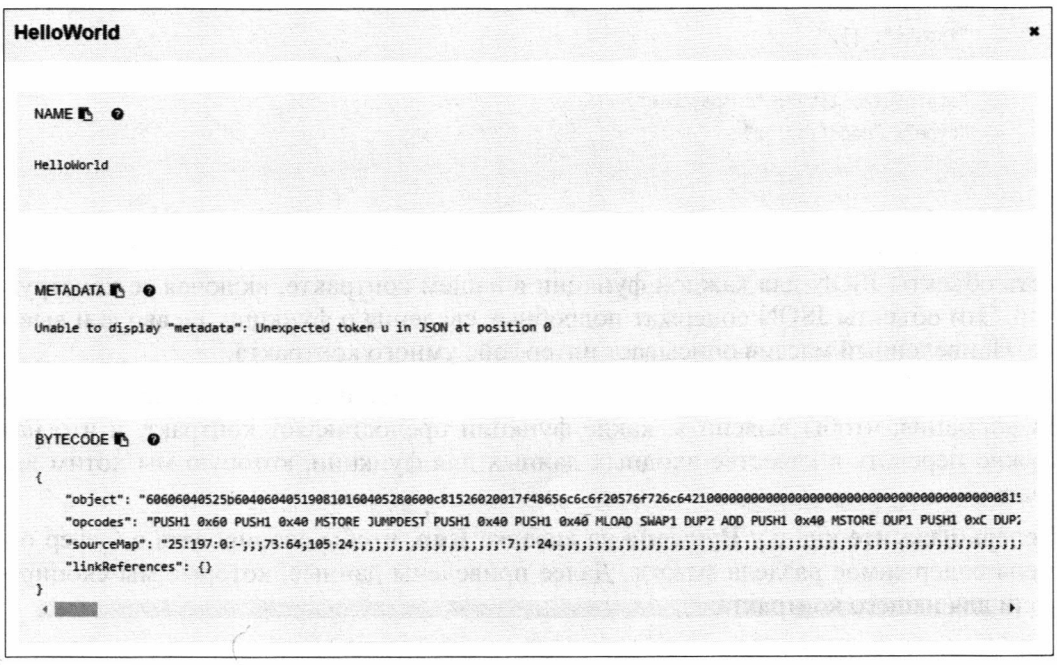

**Рис. 5.4. Редактирование умного контракта в IDE Remix**

Далее приведен полный вывод, сгенерированный операцией создания контракта, и он показывает нам, что контракт успешно создан, поскольку у него есть адрес контракта. Значение поля `from` — это адрес учетной записи, который использовался для создания контракта. Также мы можем видеть хэш транзакции создания контракта:

```
status      0x1 Transaction mined and execution succeed
contractAddress     0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
from    0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to      HelloWorld.(constructor)
gas     3000000 gas
```

```
transaction cost   205547 gas
execution cost     109539 gas
hash   0x9f3c21c21f263084b9f031966858a5d8e0648ed19c77d4d2291875b01d89a141
input  0x6060604052341561000f57600080fd5b6040805190810160405280600c81526020017f48656c6c6f20576f726
c64210000000000000000000000000000000000000000815250600090805190602001906100 5a929190610060565b50610
105565b828054600181600116156101000203166002900490600052602060002090601f016020900481019282601f10610
0a157805160ff1916838001178555610 0cf565b82800160010185558215610 0cf579182015b828111156100ce578251825
5916020019190600101906100b3565b5b5090506100dc91906100e0565b5090565b61010291905b808211156100fe57600
0816000905550600101610 0e6565b5090565b90565b6101bc806101146000396000f300606060405260043610610041576
000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff168063bcdfe0d514610
046575b600080fd5b341561005157600080fd5b6100596100d4565b6040518080602001828103825283818151815260200
191508051906020019080838360005b8381101561009957808201518184015260208101905061007e565b5050505090509
0810190601f1680156100c65780820380516001836020036101000a031916815260200191505b509250505060405180910
390f35b6100dc61017c565b600080546001816001161561010002031660029004806 01f0160208091040260200160405190
8101604052809291908181526020018280546001816001161561010002031660029004801561017257806 01f106101475
76101008083540402835291602001916101 72565b820191906000526020600020905b815481529060010190602001808 31
16101555782900360 1f168201915b505050505090509056 5b6020604051908101604052806000815250905600a165627a7
a72305820d6796e48540eced3646ea52c632364666e6409447945106631 7789a712aef4da0029
 decoded input  {}
 decoded output        -
 logs   []
 value  0 wei
```

**Итак, у нас есть готовый умный контракт «Hello World», и теперь следующим шагом является его программное развертывание в блокчейне Ethereum.**



```
HelloWorld
```

```
Unable to display "metadata": Unexpected token u in JSON at position 0
```

```
{
    "object": "60606040525b6040604051908101604052806 00c81526020017f48656c6c6f20576f726c642100000000000000000000000000000000000000008 1
    "opcodes": "PUSH1 0x60 PUSH1 0x40 MSTORE JUMPDEST PUSH1 0x40 PUSH1 0x40 MLOAD SWAP1 DUP2 ADD PUSH1 0x40 MSTORE DUP1 PUSH1 0xC DUP
    "sourceMap": "25:197:0:-;;;73:64;105:24;;;;;;;;;;;;;;;;;;;;;;;;:7;;:24;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    "linkReferences": {}
}
```

Рис. 5.5. Выходные данные после компиляции умного контракта в IDE Remix

## 5.5.3. Получение сведений о контракте

Давайте сначала получим от Remix некоторые сведения о нашем умном контракте, которые понадобятся для развертывания контракта в сети Ethereum с использованием библиотеки web3. Нам нужны разделы: ABI и BYTECODE. Нажмите кнопку **ABI** на вкладке **Run**, чтобы скопировать содержимое раздела ABI в буфер обмена. Далее приводится значение данных ABI для нашего умного контракта:

```
[
    {
        "constant": true,
        "inputs": [],
        "name": "Hello",
        "outputs": [
            {
                "name": "",
                "type": "string"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    }
]
```

Это массив JSON, и если мы внимательно посмотрим на него, то увидим, что в нем есть объекты JSON для каждой функции в нашем контракте, включая ее конструктор. Эти объекты JSON содержат подробные сведения о функции, ее вводе и выводе. Приведенный массив описывает интерфейс умного контракта.

Когда мы вызываем умный контракт после его развертывания в сети, нам нужна эта информация, чтобы выяснить, какие функции предоставляет контракт, и что нам нужно передать в качестве входных данных для функции, которую мы хотим вызвать.

Теперь нажмите кнопку **Bytecode** на вкладке **Run**, чтобы скопировать в буфер обмена содержимое раздела BYTECODE. Далее приведены данные, которые мы скопировали для нашего контракта:

```
{
    "linkReferences": {},
    "object": "60606040523415610000f57600080fd5b6040805190810160405280600c81526020017f48656c6c6f205
76f726c6421000000000000000000000000000000000000000000000000815250600090805190602001906100a5929190610060565
```

```
b50610105565b828054600181600116156101000203166002900490600052602060002090601f016020900481019282601
f106100a157805160ff1916838001178555556100cf565b828001600101855582156100cf579182015b828111156100ce578
251825591602001919060010190610b3565b5b5090506100dc91906100e0565b5090565b61010291905b808211156100f
e576000816000905550600101610e6565b5090565b90565b6101bc806101146000396000f300606060405260043610610
041576000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff168063bcdfe0d
514610046575b600080fd5b34156100515760008 0fd5b6100596100d4565b6040518080602001828103825283818151815
2602001915080519060200190808383600005b83811015610099578082015181840152602081019050610 07e565b5050505
09050908101906 01f1680156100c6578082038051600183602003610100 0a031916815260200191505b509250505060405
180910390f35b6100dc61017c565b6000805460018160011615610100020316600290048060 1f0160208091040260200 16
04051908101604052809291908181526020018280546001816001161561010002031660029004801561017257806 01f106
1014757610100808354040283529160200191610172565b820191906000526020600020905b815481529060010190602 00
18083116101555782900360 1f168201915b505050505090509056 5b6020604051908101604052806000815250905600a16
5627a7a72305820877a5da4f7e05c4ad9b45dd10fb6c133a523541ed06db6dd31d59b35d51768a30029",

"opcodes": "PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH2 0xF JUMPI PUSH1 0x0 DUP1 REVERT
JUMPDEST PUSH1 0x40 DUP1 MLOAD SWAP1 DUP2 ADD PUSH1 0x40 MSTORE DUP1 PUSH1 0xC DUP2 MSTORE PUSH1
0x20 ADD PUSH32 0x48656C6C6F20576F726C64210000000000000000000000000000000000000000 DUP2 MSTORE POP
PUSH1 0x0 SWAP1 DUP1 MLOAD SWAP1 PUSH1 0x20 ADD SWAP1 PUSH2 0x5A SWAP3 SWAP2 SWAP1 PUSH2 0x60 JUMP
JUMPDEST POP PUSH2 0x105 JUMP JUMPDEST DUP3 DUP1 SLOAD PUSH1 0x1 DUP2 PUSH1 0x1 AND ISZERO PUSH2
0x100 MUL SUB AND PUSH1 0x2 SWAP1 DIV SWAP1 PUSH1 0x0 MSTORE PUSH1 0x20 PUSH1 0x0 KECCAK256 SWAP1
PUSH1 0x1F ADD PUSH1 0x20 SWAP1 DIV DUP2 ADD SWAP3 DUP3 PUSH1 0x1F LT PUSH2 0xA1 JUMPI DUP1 MLOAD
PUSH1 0xFF NOT AND DUP4 DUP1 ADD OR DUP6 SSTORE PUSH2 0xCF JUMP JUMPDEST DUP3 DUP1 ADD PUSH1 0x1
ADD DUP6 SSTORE DUP3 ISZERO PUSH2 0xCF JUMPI SWAP2 DUP3 ADD JUMPDEST DUP3 DUP2 GT ISZERO PUSH2
0xCE JUMPI DUP3 MLOAD DUP3 SSTORE SWAP2 PUSH1 0x20 ADD SWAP2 SWAP1 PUSH1 0x1 ADD SWAP1 PUSH2 0xB3
JUMP JUMPDEST JUMPDEST POP SWAP1 POP PUSH2 0xDC SWAP2 SWAP1 PUSH2 0xE0 JUMP JUMPDEST POP SWAP1
JUMP JUMPDEST PUSH2 0x102 SWAP2 SWAP1 JUMPDEST DUP1 DUP3 GT ISZERO PUSH2 0xFE JUMPI PUSH1 0x0 DUP2
PUSH1 0x0 SWAP1 SSTORE POP PUSH1 0x1 ADD PUSH2 0xE6 JUMP JUMPDEST POP SWAP1 JUMP JUMPDEST SWAP1
JUMP JUMPDEST PUSH2 0x1BC DUP1 PUSH2 0x114 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1 0x60
PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH2 0x41 JUMPI PUSH1 0x0 CALLDATALOAD PUSH29
0x100000000000000000000000000000000000000000000000000000000 SWAP1 DIV PUSH4 0xFFFFFFFF AND DUP1
PUSH4 0xBCDFE0D5 EQ PUSH2 0x46 JUMPI JUMPDEST PUSH1 0x0 DUP1 REVERT JUMPDEST CALLVALUE ISZERO
PUSH2 0x51 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH2 0x59 PUSH2 0xD4 JUMP JUMPDEST PUSH1 0x40
MLOAD DUP1 DUP1 PUSH1 0x20 ADD DUP3 DUP2 SUB DUP3 MSTORE DUP4 DUP2 DUP2 MLOAD DUP2 MSTORE PUSH1
0x20 ADD SWAP2 POP DUP1 MLOAD SWAP1 PUSH1 0x20 ADD SWAP1 DUP1 DUP4 DUP4 PUSH1 0x0 JUMPDEST DUP4
DUP2 LT ISZERO PUSH2 0x99 JUMPI DUP1 DUP3 ADD MLOAD DUP2 DUP5 ADD MSTORE PUSH1 0x20 DUP2 ADD SWAP1
POP PUSH2 0x7E JUMP JUMPDEST POP POP POP POP SWAP1 POP SWAP1 DUP2 ADD SWAP1 PUSH1 0x1F AND DUP1
ISZERO PUSH2 0xC6 JUMPI DUP1 DUP3 SUB DUP1 MLOAD PUSH1 0x1 DUP4 PUSH1 0x20 SUB PUSH2 0x100 EXP SUB
NOT AND DUP2 MSTORE PUSH1 0x20 ADD SWAP2 POP JUMPDEST POP SWAP3 POP POP POP PUSH1 0x40 MLOAD DUP1
SWAP2 SUB SWAP1 RETURN JUMPDEST PUSH2 0xDC PUSH2 0x17C JUMP JUMPDEST PUSH1 0x0 DUP1 SLOAD PUSH1
0x1 DUP2 PUSH1 0x1 AND ISZERO PUSH2 0x100 MUL SUB AND PUSH1 0x2 SWAP1 DIV DUP1 PUSH1 0x1F ADD
PUSH1 0x20 DUP1 SWAP2 DIV MUL PUSH1 0x20 ADD PUSH1 0x40 MLOAD SWAP1 DUP2 ADD PUSH1 0x40 MSTORE
DUP1 SWAP3 SWAP2 SWAP1 DUP2 DUP2 MSTORE PUSH1 0x20 ADD DUP3 DUP1 SLOAD PUSH1 0x1 DUP2 PUSH1 0x1
AND ISZERO PUSH2 0x100 MUL SUB AND PUSH1 0x2 SWAP1 DIV DUP1 ISZERO PUSH2 0x172 JUMPI DUP1 PUSH1
0x1F LT PUSH2 0x147 JUMPI PUSH2 0x100 DUP1 DUP4 SLOAD DIV MUL DUP4 MSTORE SWAP2 PUSH1 0x20 ADD
SWAP2 PUSH2 0x172 JUMP JUMPDEST DUP3 ADD SWAP2 SWAP1 PUSH1 0x0 MSTORE PUSH1 0x20 PUSH1 0x0
KECCAK256 SWAP1 JUMPDEST DUP2 SLOAD DUP2 MSTORE SWAP1 PUSH1 0x1 ADD SWAP1 PUSH1 0x20 ADD DUP1 DUP4
GT PUSH2 0x155 JUMPI DUP3 SWAP1 SUB PUSH1 0x1F AND DUP3 ADD SWAP2 JUMPDEST POP POP POP POP POP
SWAP1 POP SWAP1 JUMP JUMPDEST PUSH1 0x20 PUSH1 0x40 MLOAD SWAP1 DUP2 ADD PUSH1 0x40 MSTORE DUP1
PUSH1 0x0 DUP2 MSTORE POP SWAP1 JUMP STOP LOG1 PUSH6 0x627A7A723058 KECCAK256 DUP8 PUSH27
0x5DA4F7E05C4AD9B45DD10FB6C133A523541ED06DB6DD31D59B35D5 OR PUSH9 0xA30029000000000000 ",
    "sourceMap": "24:199:0:-;;;75:62;;;;;;;;106:24;;;;;;;;;;;;;;;;;:7;:24;;;;;;;;;;;;;:::i;:::-
;;24:199;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;:::i;:::-;;;:::o;:::-;;;;;;;;;;;;;;;;;;;;;;;;;;;:::o;:::-;;;;;;;"
}
```

Как мы видим, данные в разделе BYTECODE тоже являются объектами JSON. Это, в основном, выходные данные компилятора. Remix скомпилировал наш умный контракт, и в результате мы получили байт-код. Теперь внимательно изучите этот массив JSON и посмотрите на поле object и его значение. Это шестнадцатеричная строка, которая содержит байт-код для нашего умного контракта, и мы будем отправлять его в транзакции создания контракта в поле data — помните то поле данных, которое мы оставили пустым в предыдущей транзакции Ethereum между Алисой и Бобом?

Теперь у нас есть все данные нашего умного контракта, и мы готовы отправить его в сеть Ethereum.

## 5.5.4. Развертывание контракта в сети Ethereum

Итак, у нас есть умный контракт и его данные, и нам нужно подготовить транзакцию, которая развернет контракт в блокчейне Ethereum. Транзакция размещения контракта будет очень похожа на денежную транзакцию, которую мы подготовили в предыдущем разделе, но она содержит еще несколько свойств, необходимых для создания контрактов.

Во-первых, нам нужно создать объект класса web3.eth.Contract, который может представлять наш контракт. Следующий фрагмент кода создает экземпляр для указанного класса с массивом JSON в качестве входного параметра. Это тот самый массив JSON, который мы скопировали из раздела ABI в редакторе Remix на предыдущем этапе:

```
var helloworldContract = new web3.eth.Contract([{
        "constant": true,
        "inputs": [],
        "name": "Hello",
        "outputs": [{
            "name": "",
            "type": "string"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }, {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    }]);
```

Теперь нам нужно отправить этот контракт в сеть Ethereum, используя метод Contract.deploy библиотеки web3. Следующий фрагмент кода показывает, как это сделать:

```
helloworldContract
.deploy({
        data:
'0x6060604052341561000f57600080fd5b604080519081016040528060c81526020017f48656c6c6f20576f726c64210
0000000000000000000000000000000000000000000815250600090805190602001906100 5a9291906100605650610105565
b828054600181600116156101000203166002900490600052602060002090601f016020900481019282601f106100a1578
05160ff1916838001178555610cf565b828001600101855582156100cf579182015b828111156100ce578251825591602
0019190600101906100b3565b5b5090506100dc91906100e0565b5090565b61010291905b808211156100fe57600081600
0905550600101610e6565b5090565b90565b6101bc806101146000396000f300606060405260043610610041576000357
c0100000000000000000000000000000000000000000000000000000000900463ffffffff168063bcdfe0d514610046575
b600080fd5b34156100515760008 0fd5b6100596100d4565b604051808060200182810382528381815181526020019150
8
0519060200190808383600 05b838110156100995780820151818401526020810190506100 7e565b5050505090509081019
0601f1680156100c6578082038051600183602003610100 0a031916815260200191505b509250505060405180910390f35
b6100dc61017c565b60008054600181600116156101000203166002900480601f016020809104026020016040519081016
0405280929190818152602001828054600181600116156101000203166002900480156101725780601f106101475761010
0808354040283529160200191610172565b820191906000526020600020905b815481529060010190602001808311610 15
557829003601f168201915b505050505090509056 5b602060405190810160405280600081525090560 0a165627a7a72305
820877a5da4f7e05c4ad9b45dd10fb6c133a523541ed06db6dd31d59b35d51768a30029'
    })
    .send({
        from: '0xAff9d328E8181aE831Bc426347949EB7946A88DA',
        gas: 4700000,
        gasPrice: '20000000000000'
    },
    function(error, transactionHash){
        console.log(error);
        console.log(transactionHash);
    })
    .then(function(contract){
        console.log(contract);
    });
```

Обратите внимание, что значение поля `data` в параметре функции `deploy` совпадает со значением, которое мы получили в поле `BYTECODE` на предыдущем шаге. Перед этим значением добавляются символы `0x`. Таким образом, данные, передаваемые в функции развертывания, представляют собой `0x` + байт-код контракта.

В функцию отправки после развертывания мы добавили адрес отправителя `from`, который будет являться владельцем контракта, а также информацию о плате за транзакцию: лимит газа `gas` и цену на газ `gasPrice`. Наконец, когда вызов функции завершен, возвращается объект контракта. Этот объект контракта содержит адрес контракта, который можно использовать для вызова функций контракта (member functions, иногда их называют *методами контракта*).

Другой способ отправки контракта в сеть — упаковать контракт внутрь транзакции и отправить его напрямую. Следующий фрагмент кода создает объект транзакции с байт-кодом контракта в качестве содержимого поля данных, подписывает его с помощью закрытого ключа отправителя и затем отправляет его в блокчейн Ethereum.

**Обратите внимание, что мы не назначили адрес получателя в этом объекте транзакции, т. к. адрес контракта неизвестен до его развертывания:**

```
var tx = {
        from: "0x22013fff98c2909bbFCcdABb411D3715fDB341eA",
        gasPrice: "20000000000",
        gas: "4900000",
        data:
"0x6060604052341561000f57600080fd5b604080519081016040528060c81526020017f48656c6c6f20576f726c64210
0000000000000000000000000000000000000000081525060009080519060200190610005a929190610060565b50610105565
b828054600181600116156101000203166002900490600052602060002090601f016020900481019282601f106100a1578
05160ff1916838001178555610cf565b8280016001018555582156100cf579182015b828111156100ce578251825591602
0019190600101906100b3565b5b5090506100dc91906100e0565b5090565b61010291905b808211156100fe57600081600
0905550600101610100e6565b5090565b90565b6101bc806101146000396000f300606060405260043610610041576000357
c0100000000000000000000000000000000000000000000000000000000900463ffffffff168063bcdfe0d514610046575
b600080fd5b34156100515760008fd5b6100596100d4565b604051808060200182810382528381815181526020019150
8051906020019080838360005b838110156100995780820151818401526020810190506100 7e565b505050509050908101 9
0601f1680156100c6578082038051600183602003610100a031916815260200191505b509250505060405180910390f35
b6100dc61017c565b60008054600181600116156101000203166002900480601f016020809104026020016040519081016
0405280929190818152602001828054600181600116156101000203166002900480156101725780601f106101475761010
0808354040283529160200191610172565b820191906000526020600020905b8154815290600101906020018083116101 5
557829003601f168201915b505050505090509056 5b602060405190810160405280600081525090560 0a165627a7a72305
820877a5da4f7e05c4ad9b45dd10fb6c133a523541ed06db6dd31d59b35d51768a30029"
    };

    web3.eth.accounts.signTransaction(tx,
'0xc6676b7262dab1a3a28a781c77110b63ab8cd5eae2a5a828ba3b1ad28e9f5a9b')
    .then(function (signedTx) {
        web3.eth.sendSignedTransaction(signedTx.rawTransaction)
        .then(console.log);
    });
```

**Когда мы выполняем этот фрагмент кода, то получаем следующий вывод — квитанцию нашей транзакции:**

```
{
    blockHash: '0xaba93b4561fc35e062a1ad72460e0b677603331bbee3379ce6c74fa5cf505d82',
    blockNumber: 2539889,
    contractAddress: '0xd5a2d13723A34522EF79bE0f1E7806E86a4578E9',
    cumulativeGasUsed: 205547,
    from: '0x22013fff98c2909bbfccdabb411d3715fdb341ea',
    gasUsed: 205547,
    logs: [],
    logsBloom:
'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000',
    status: '0x1',
    to: null,
```

```
    transactionHash: '0xc333cbc5fc93b52871689aab22c48b910cb192b4875bea69212363030d36565a',
    transactionIndex: 0
}
```

Обратите внимание на свойства квитанции. У нее есть значение, присвоенное свойству `contractAddress`, а значение свойства `to` равно `null`. Это означает, что была выполнена транзакция создания контракта, которая успешно обработана в сети, и контракт, созданный в рамках этой транзакции, развернут по адресу:

```
0xd5a2d13723A34522EF79bE0f1E7806E86a4578E9
```

Итак, мы успешно создали и развернули умный контракт Ethereum.

# 5.6. Вызов функций умного контракта

Теперь, когда мы развернули наш умный контракт в сети Ethereum, мы можем вызывать его функции. Далее приведены шаги программного вызова функций умного контракта Ethereum.

## 5.6.1. Получение ссылки на смарт-контракт

Чтобы вызвать функцию умного контракта, сначала нам нужно создать экземпляр класса `web3.eth.Contract` с ABI и адресом нашего развернутого контракта. Следующий фрагмент кода показывает, как это сделать:

```
var helloworldContract = new web3.eth.Contract([{
        "constant": true,
        "inputs": [],
        "name": "Hello",
        "outputs": [{
            "name": "",
            "type": "string"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }, {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    }], '0xd5a2d13723A34522EF79bE0f1E7806E86a4578E9');
```

В этом фрагменте кода мы создали экземпляр класса `web3.eth.Contract`, передав ABI контракта, который мы создали в предыдущем разделе, а также передали адрес контракта, который мы получили после развертывания контракта.

Этот объект теперь можно использовать для вызова функций нашего контракта.

## 5.6.2. Вызываем функцию умного контракта

Напомним, что в нашем контракте есть только одна публичная функция. Это метод, который называется `Hello` и возвращает строку `Hello World` после выполнения.

Чтобы выполнить этот метод, мы будем вызывать его с помощью класса `contract.methods` библиотеки web3. Следующий фрагмент кода демонстрирует вызов метода `Hello`:

```
helloworldContract.methods.Hello().send({
        from: '0xF68b93AE6120aF1e2311b30055976d62D7dBf531'
    }).then(console.log);
```

В этом фрагменте кода мы добавили значение к полю `from` в функции `send`, и этот адрес будет использоваться для отправки транзакции, которая, в свою очередь, будет запускать функцию `Hello` в нашем умном контракте.

Полный код для вызова умного контракта приведен в следующем фрагменте:

```
var callContract = function () {
    var helloworldContract = new web3.eth.Contract([{
        "constant": true,
        "inputs": [],
        "name": "Hello",
        "outputs": [{
            "name": "",
            "type": "string"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }, {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    }], '0xd5a2d13723A34522EF79bE0f1E7806E86a4578E9');

    helloworldContract.methods.Hello().send({
        from: '0xF68b93AE6120aF1e2311b30055976d62D7dBf531'
    }).then(console.log);
};
```

Другим способом выполнения функции контракта будет отправка необработанной подписанной транзакции на адрес контракта. Это похоже на то, как в предыдущих разделах мы отправили необработанные транзакции Ethereum для пересылки эфира и создания контракта. В этом случае все, что нам нужно сделать, — это подставить адрес контракта в поле `to` объекта транзакции и закодированное значение ABI вызова функции — в поле данных.

Следующий фрагмент кода сначала создает объект контракта, а затем получает закодированное значение ABI вызываемой функции умного контракта. Потом он

создает транзакцию на основе этих значений, подписывает и отправляет ее в сеть. Обратите внимание, что мы использовали функцию `encodeABI` в функции контракта, чтобы получить актуальное значение данных (payload value) для транзакции. Это входные данные для умного контракта:

```
var callContract = function () {
    var helloworldContract = new web3.eth.Contract([{
        "constant": true,
        "inputs": [],
        "name": "Hello",
        "outputs": [{
            "name": "",
            "type": "string"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }, {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    }], '0xd5a2d13723A34522EF79bE0f1E7806E86a4578E9');

    var payload = helloworldContract.methods.Hello().encodeABI();

    var tx = {
        from: "0xF68b93AE6120aF1e2311b30055976d62D7dBf531",
        gasPrice: "20000000000",
        gas: "4700000",
        data: payload
    };

    web3.eth.accounts.signTransaction(tx,
'0xc6676b7262dab1a3a28a781c77110b63ab8cd5eae2a5a828ba3b1ad28e9f5a9b')
        .then(function (signedTx) {
            web3.eth.sendSignedTransaction(signedTx.rawTransaction)
            .then(console.log);
    });
};
```

**ВАЖНОЕ ПРИМЕЧАНИЕ.** При использовании публичного узла для создания и выполнения умных контрактов Ethereum мы должны использовать подход с отправкой необработанных транзакций, поскольку субмодуль `web3.eth.Contract` библиотеки web3 задействует либо незаблокированную учетную запись, либо учетную запись по умолчанию, связанную с узлом провайдера Ethereum, но этот субмодуль на момент подготовки книги не поддерживается публичными узлами.

## 5.7. Блокчейн с новой точки зрения

В предыдущих разделах мы программно отправляли транзакции в блокчейны Bitcoin и Ethereum, используя JavaScript. Глядя на процесс создания транзакций, мы можем с новых позиций обсудить общие понятия блокчейна.

- **Транзакции.** Теперь мы можем сказать, что транзакции блокчейна — это операции, инициированные владельцем учетной записи, которые, в случае успешного завершения, обновляют состояние блокчейна. Например, после выполнения наших транзакций между Алисой и Бобом владение определенным количеством биткойнов и эфира перешло от Алисы к Бобу и наоборот, и это изменение владельца было зарегистрировано в блокчейне, что привело его к новому состоянию. В случае Ethereum транзакции могут создавать и выполнять контракты, и эти транзакции также обновляют состояние блокчейна. Мы создали транзакцию, которая развернула умный контракт в блокчейне Ethereum. Состояние блокчейна было обновлено, потому что теперь у нас есть новый адрес (счет), созданный в блокчейне.
- **Входы, выходы, счета и балансы.** Мы рассмотрели, как Bitcoin и Ethereum отличаются друг от друга с точки зрения управления состоянием. В то время как Bitcoin использует модель UTXO, Ethereum применяет модель счетов и балансов. Однако общая идея заключается в том, что блокчейны регистрируют владение активами, а транзакции используются для смены владельца этих активов.
- **Плата за транзакцию.** За каждую транзакцию, которую мы проводим в публичных блокчейнах, мы должны платить майнерам комиссию. В Bitcoin комиссия рассчитывается автоматически, в то время как в Ethereum мы должны указать максимальную сумму, которую готовы платить, исходя из цены на газ и лимита газа.
- **Подписание.** В обоих случаях после создания объекта транзакции с необходимыми значениями мы подписали его с помощью открытого ключа отправителя. Криптографическая подпись — это способ доказать право собственности на активы. Если подпись неверна, транзакция становится недействительной.
- **Трансляция транзакций.** После создания и подписания транзакций мы отправляем их на узлы блокчейна. Хотя мы отправляли наши учебные транзакции на общедоступные узлы тестовой сети Bitcoin и Ethereum, на практике мы можем отправлять транзакции нескольким узлам одновременно, если не доверяем обработку наших транзакций единственному узлу. Это называется *трансляцией транзакций в сеть*.

Подводя итог, можно сказать, что если мы собираемся обновить состояние блокчейна, мы отправляем подписанные транзакции, и чтобы эти транзакции были подтверждены, нам нужно заплатить определенную плату майнерам.

# 5.8. Публичные и частные блокчейны

По типу доступа блокчейны могут быть классифицированы как публичные (открытые) и частные (закрытые). Основное различие между ними заключается в контроле доступа. Публичные или открытые блокчейны не ограничивают добавление новых узлов в сеть, и любой желающий может создать новый узел. Частные блокчейны имеют ограниченное количество узлов, и не каждый может присоединиться к сети. Примерами публичных блокчейнов являются основные сети Bitcoin и Ethereum. Примером частного блокчейна может быть корпоративная сеть из нескольких узлов Ethereum, связанных друг с другом, но не связанных с основной сетью. Эти узлы и будут вместе называться *частным блокчейном*.

Частные блокчейны, как правило, применяются для обмена данными как внутри обособленного предприятия, так и между партнерами и/или подразделениями.

Когда мы разрабатываем приложения для блокчейнов, тип доступа к блокчейну имеет значение, потому что правила взаимодействия с блокчейном могут различаться. Каждый частный блокчейн может иметь собственный набор правил. Например, могут различаться токены (единицы ценности), цена на газ, плата за транзакцию, конечные точки и прочие условия. Эти различия будут влиять и на наши приложения.

В наших примерах кода мы в первую очередь ориентировались на общедоступные тестовые сети Bitcoin и Ethereum. Хотя основные идеи взаимодействия с закрытыми реализациями этих блокчейнов остаются неизменными, различия будут заключаться в точной подгонке кода под правила частных сетей.

# 5.9. Архитектура децентрализованных приложений

Децентрализованные приложения предназначены для непосредственного взаимодействия с узлами блокчейна без участия каких-либо центральных органов. Однако если мы внедряем технологии блокчейна на существующих предприятиях с их иерархической системной интеграцией и ограниченными функциональными возможностями, то зачастую приходится делать выбор между полной и частичной децентрализацией приложений.

## 5.9.1. Публичные и локальные узлы

Блокчейн с технической точки зрения — это децентрализованная сеть узлов. Все узлы имеют одинаковую копию данных, и они всегда согласовывают состояние данных между собой. Когда мы разрабатываем приложение для блокчейна, мы можем заставить наше приложение взаимодействовать с любым из узлов целевой сети. Обычно ориентируются на один из двух основных подходов:

- *локальный узел* — децентрализованное приложение и узел, с которым оно соединяется, работают на одном локальном компьютере. Это означает, что пользо-

ватели нашего приложения должны запустить локальный узел блокчейна на своем компьютере и приказать приложению подключиться к нему. Эта модель является полностью децентрализованной моделью запуска приложения. Примером такой модели является основанный на Ethereum браузер Mist, который использует локальный узел geth.

Этот подход схематически изображен на рис. 5.6.

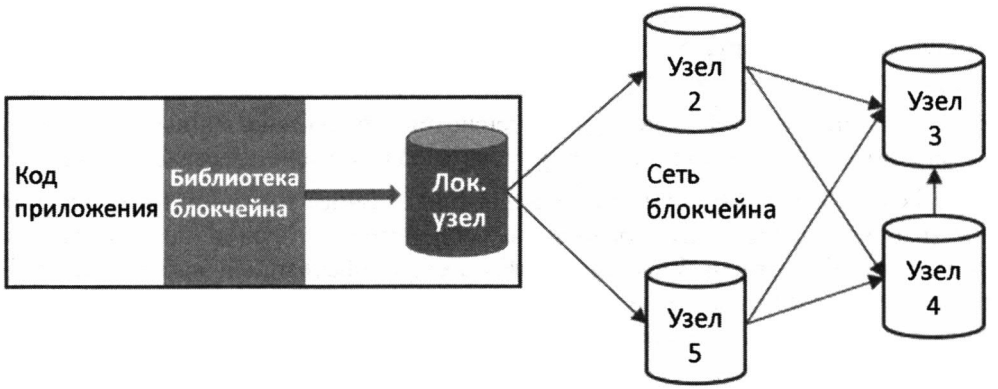



**Рис. 5.6. Приложение соединяется с локальным узлом**

- *публичный узел* — приложение общается с общедоступным узлом, размещенным третьей стороной. При этом нашим пользователям не нужно запускать локальный узел на своем компьютере. У такого подхода есть преимущества и недостатки. В то время как пользователям не нужно платить за оборудование и энергию для запуска локального узла, им приходится доверять третьей стороне для передачи своих транзакций в блокчейн. Такую модель использует браузерный плагин MetaMask, который соединяется с публичными узлами Ethereum.

Этот подход схематически изображен на рис. 5.7.

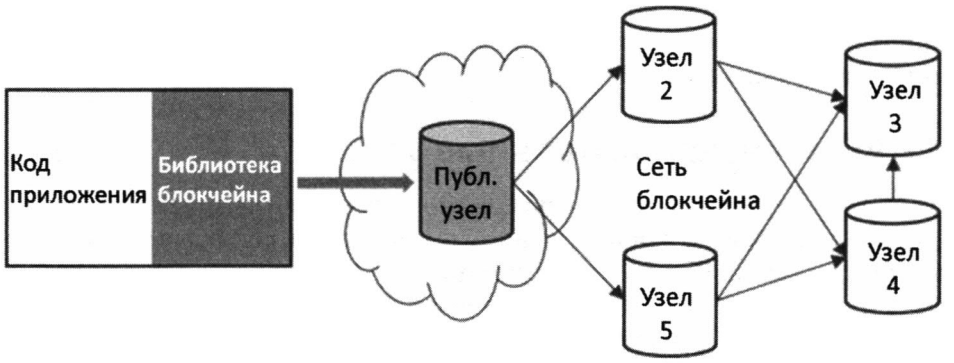



**Рис. 5.7. Приложение соединяется с публичным узлом**

### 5.9.2. Децентрализованные приложения и серверы

Помимо ранее упомянутых сценариев, могут быть и другие варианты — в зависимости от конкретных случаев использования и требований заказчика. Существует множество сценариев, когда между приложением и блокчейном необходим промежуточный сервер. Например, когда вам нужно хранить кэш состояния блокчейна для более быстрых запросов, когда приложение должно отправлять уведомления (электронные письма, push-сообщения, SMS и т. п.) пользователям на основе событий обновления состояния блокчейна или когда задействованы несколько реестров, и вам нужно запустить внутреннюю логику для преобразования данных между реестрами. Представьте себе сложную инфраструктуру, которая используется некоторыми крупными биржами криптовалют, где, среди прочего, мы получаем все услуги — такие как двухфакторная аутентификация, уведомления и шлюзы оплаты, и ни одна из этих служб недоступна напрямую ни в одной из цепочек блоков. В более широком смысле, блокчейны просто обеспечивают защиту от мошенничества на уровне данных.

## 5.10. Заключение

В этой главе мы говорили о разработке децентрализованных приложений и проделали несколько упражнений по программному взаимодействию с блокчейнами Bitcoin и Ethereum. Мы также рассмотрели модели архитектуры приложений и их различия в зависимости от вариантов подключения.

В следующей главе мы создадим частную сеть Ethereum, а затем разработаем полноценное приложение, которое будет взаимодействовать с этой частной сетью и использовать умные контракты для реализации прикладной логики.

## 5.11. Рекомендуемые источники

- Документация web3.js: **http://web3js.readthedocs.io/en/1.0/index.html**.
- Документация Solidity: **https://solidity.readthedocs.org/**.
- Репозиторий исходного кода BitcoinJS: **https://github.com/bitcoinjs/bitcoinjs-lib**.
- Документация Infura: **https://infura.io/docs**.
- Документация API обозревателя блоков: **https://blockexplorer.com/api-ref**.
- Разработка архитектуры приложений Ethereum: **https://blog.zeppelin.solutions/designing-the-architecture-for-your-ethereum-application-9cec086f8317**.

ГЛАВА 6

# Разработка приложений Ethereum

В предыдущей главе мы узнали, как взаимодействовать с блокчейнами Bitcoin и Ethereum при помощи JavaScript. Мы также кратко обсудили, как создавать и развертывать умные контракты Ethereum. В этой главе мы выходим на новый уровень разработки приложений и разберемся, как разрабатывать и развертывать децентрализованные приложения на основе платформы Ethereum. Мы настроим частную сеть Ethereum, а затем используем эту сеть в качестве рабочего блокчейна для нашего приложения. Функциональная часть приложения (серверная часть) будет реализована в виде умного контракта Ethereum, а для доступа к функциям контракта мы воспользуемся подключенным к частной сети Ethereum веб-приложением (клиентская часть). Таким образом, мы намерены охватить все аспекты разработки приложений Ethereum: от настройки узлов и сетей до использования умного контракта и выполнения функций контракта через клиентские приложения.

## 6.1. Децентрализованное приложение

Прежде чем мы начнем разработку приложения, нам нужно придумать пример использования. Нам также необходимо определить перечень компонентов, которые станут частью нашего приложения. Итак, давайте с этого и начнем.

Допустим, нам необходимо разработать приложение для проведения опросов, которое позволяет избирателям отвечать на вопрос, опубликованный в открытом доступе. Механизм опроса с использованием централизованной системы не отличается большой надежностью, поскольку в нем имеется центральная точка подмены или повреждения данных. Итак, цель нашего приложения — создать децентрализованный опрос, где каждый избиратель контролирует свой голос, и каждый голос обрабатывается на всех узлах в блокчейне, поэтому нет возможности подделать результаты опроса. Хотя эта задача легко решается с помощью публичного блокчейна Ethereum, чтобы сделать наше упражнение интересным, мы настроим частную сеть

Ethereum и развернем в ней наше приложение. Звучит увлекательно? Приступаем к работе!

Первым шагом станет запуск частной сети Ethereum. Затем мы создадим умный контракт, который будет развернут в этой частной сети. Чтобы взаимодействовать с умным контрактом, мы создадим интерфейсное веб-приложение с использованием библиотеки web3. Вот и все.

В соответствии с только что описанным планом, наше упражнение по разработке децентрализованного приложения будет включать следующие шаги:

1. Настройку частной сети Ethereum.
2. Создание умного контракта для реализации функций опроса.
3. Развертывание умного контракта в частной сети.
4. Создание интерфейсного веб-приложения для взаимодействия с умным контрактом.

В следующих разделах мы подробно рассмотрим каждый из этих шагов.

---

**ПРИМЕЧАНИЕ.** Как уже упоминалось, для разработки приложений мы можем использовать общедоступную сеть Ethereum. Чтобы ускорить разработку приложения, мы также можем использовать несколько инструментов, таких как браузерный плагин MetaMask и фреймворк Truffle. Эти инструменты помогают нам лучше управлять кодом и развертыванием контрактов. Читателю предлагается изучить эти и другие инструменты, чтобы попытаться найти наилучшую комбинацию для создания удобной и продуктивной среды разработки децентрализованных приложений. Наш же рассказ в первую очередь направлен на то, чтобы показать читателю, что скрывается за кулисами при создании приложений Ethereum, поэтому все инструменты, обеспечивающие разработку верхнего уровня (поверх процесса разработки непосредственно приложения), остаются за рамками книги.

---

## 6.2. Настройка частной сети Ethereum

Чтобы создать частную сеть Ethereum, нам понадобится один из множества доступных клиентов Ethereum. Проще говоря, клиент Ethereum — это приложение, которое реализует протокол блокчейна Ethereum. Одним из самых популярных клиентов является GoEthereum, также известный как geth. Им мы и воспользуемся для настройки нашей частной сети. Выполнять это упражнение мы будем на виртуальной машине под управлением Ubuntu Linux версии 16.04. Работа под ОС Windows не отличается практически ничем, разве что в консольных командах нам не придется использовать команду `sudo`.

---

**ПРИМЕЧАНИЕ.** Узел geth последних версий работает только на компьютерах с ОЗУ не менее 4 Гбайт. В противном случае при попытке запустить узел вы будете получать сообщение о нехватке памяти.

---

## 6.2.1. Установка клиента GoEthereum

Скачайте установщик geth из официального источника: **https://geth.ethereum.org/downloads/**. На странице загрузки официального сайта geth доступны пакеты установки для всех основных платформ: Windows, macOS, Linux (на момент подготовки русского перевода была доступна версия 1.8.21).

Загрузите установочный пакет для вашей платформы и установите geth на свой локальный компьютер. Вы также можете установить geth на удаленном (облачном) сервере или на виртуальной машине.

После успешной установки geth на своем локальном компьютере вы можете проверить установку, выполнив следующую команду в терминале (командной строке):

```
geth version
```

В зависимости от операционной системы и установленной версии geth эта команда должна выдать вывод, подобный следующему:

```
Geth
Version: 1.7.3-stable
Git Commit: 4bb3c89d44e372e6a9ab85a8be0c9345265c763a
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.9
Operating System: linux
GOPATH=
GOROOT=/usr/lib/go-1.9
```

## 6.2.2. Создание каталога данных geth

По умолчанию после установки у geth будет свой рабочий каталог, но мы создадим собственный каталог с простым и понятным названием `mygeth`:

```
mkdir mygeth
```

## 6.2.3. Создание учетной записи geth

Первое, что нам нужно, — это учетная запись Ethereum, которая может содержать криптовалюту эфир. Эта учетная запись понадобится нам для создания наших умных контрактов и транзакций позднее, в процессе разработки DApp. Мы можем создать новую учетную запись, используя следующую команду:

```
sudo geth account new --datadir <путь к каталогу данных, который мы создали на предыдущем шаге>
sudo geth account new --datadir /mygeth
```

**ПРИМЕЧАНИЕ.** Мы используем команду `sudo`, чтобы избежать проблем с разрешениями ОС Linux. При работе под управлением ОС Windows `sudo` не требуется.

После запуска команды создания учетной записи в командной строке у вас будет запрошен пароль для нее. Придумайте, введите и подтвердите пароль, после чего ваша учетная запись geth будет создана (обязательно запомните введенный вами пароль. Позже он понадобится для разблокировки учетной записи при подписании транзакций). Адрес созданной учетной записи будет показан вам на экране.

Наш адрес сгенерированной учетной записи:

```
baf735f889d603f0ec6b1030c91d9033e60525c3.
```

У вас же будет сгенерирован другой адрес учетной записи. Скопируйте и сохраните его, он понадобится вам позже.

На снимке с экрана (рис. 6.1) показан этот процесс.

```
author@testserver:~$ sudo geth account new --datadir /mygeth
Your new account is locked with a password. Please give a password. Do not forget this p
assword.
Passphrase:
Repeat passphrase:
Address: {baf735f889d603f0ec6b1030c91d9033e60525c3}
```

**Рис. 6.1. Настройка учетной записи Ethereum в geth**

Обратите внимание, что мы передали имя каталога данных в качестве параметра команды создания учетной записи. Это сделано для того, чтобы убедиться, что файл, содержащий данные учетной записи, создан внутри нашего каталога данных, и обеспечить прямой доступ к учетной записи из контекста этого каталога. Если мы не передадим каталог данных в команду создания учетной записи, он автоматически примет местоположение каталога данных geth по умолчанию (которое может различаться в зависимости от платформы).

## 6.2.4. Создание файла конфигурации genesis.json

После установки geth и создания новой учетной записи следующим шагом является настройка начального состояния для нашей частной сети. Как мы видели в предыдущих главах, блокчейны имеют блок генезиса, который служит начальной точкой блокчейна, и все транзакции и блоки проверяются по блоку генезиса. В нашей частной сети будет пользовательский блок генезиса и, следовательно, пользовательская конфигурация блокчейна. Эта конфигурация определяет ключевые значения для блокчейна, такие как уровень сложности, лимит газа для блоков и некоторые другие.

Файл конфигурации Ethereum имеет формат JSON. Каждый из ключей этого объекта является значением конфигурации, которая управляет сетью:

```
{
    "config": {
        "chainId": 3792,
        "homesteadBlock": 0,
```

```
        "eip155Block": 0,
        "eip158Block": 0
    },
    "difficulty": "2000",
    "gasLimit": "2100000",
    "alloc": {
        "baf735f889d603f0ec6b1030c91d9033e60525c3": { "balance": "9000000000000000000" }
    }
}
```

Объект JSON начинается с раздела `config`, содержащего ряд вложенных параметров. Важным параметром, который следует здесь отметить, является `chain`. Он представляет собой идентификатор блокчейна и помогает предотвратить атаку повторением[1]. Для нашей частной цепочки мы выбрали случайный идентификатор `3792`. Вы можете выбрать любое число, отличное от номеров, используемых основной сетью (1) и тестовыми сетями (2, 3 и 4).

Следующий важный параметр — `difficulty`. Это значение сложности, которое определяет, насколько непросто будет добывать новый блок. Сложность намного выше в основной сети Ethereum, но для частных сетей мы можем выбрать относительно небольшое значение.

Далее следует параметр `gasLimit`. Это общий лимит газа для блока, а не транзакции. Более высокое значение обычно означает больше транзакций в каждом блоке.

Конфигурацию завершает раздел `alloc`. Используя эту конфигурацию, мы можем добавить в счета Ethereum начальную сумму валюты в Wei. Как видите, мы профинансировали ту же учетную запись Ethereum, которую создали на последнем шаге, и добавили в нее стартовую сумму 9 эфиров.

## 6.2.5. Запуск первого узла частной сети

Чтобы запустить первый узел частной сети, сначала скопируем содержимое JSON из предыдущего шага и сохраним его как файл с именем genesis.json (не забудьте подставить в параметр `alloc` адрес своей учетной записи). Для удобства сохраните этот файл в том же каталоге, который мы используем в качестве каталога данных для geth.

Теперь нам нужно инициализировать geth с помощью genesis.json. Эта инициализация необходима для установки пользовательской конфигурации генезиса для нашей частной сети.

Перейдем в каталог, где мы сохранили файл genesis.json:

```
cd mygeth
```

[1] Разновидность сетевой атаки, когда злоумышленник копирует и повторно отправляет в сеть достоверные данные, либо преднамеренно задерживает их доставку и отправляет позже.

Следующая команда инициализирует geth с пользовательской конфигурацией, которую мы определили в файле genesis.json:

```
sudo geth --datadir "/mygeth" init genesis.json
```

geth подтвердит настройку пользовательской конфигурации genesis выводом, показанном на следующем снимке экрана (рис. 6.2).

```
author@testserver:~/mygeth$ sudo geth --datadir "/mygeth" init genesis.json
INFO [02-11|17:45:32] Allocated cache and file handles         database=/mygeth/geth/chaindata cache=16 handles=16
INFO [02-11|17:45:32] Writing custom genesis block
INFO [02-11|17:45:32] Successfully wrote genesis state         database=chaindata              hash=294e0a..1f4aa7
INFO [02-11|17:45:32] Allocated cache and file handles         database=/mygeth/geth/lightchaindata cache=16 handles=16
INFO [02-11|17:45:32] Writing custom genesis block
INFO [02-11|17:45:32] Successfully wrote genesis state         database=lightchaindata         hash=294e0a..1f4aa7
```

**Рис. 6.2.** Инициализация узла geth с конфигурацией согласно файлу genesis.json

Далее нам нужно запустить geth, используя следующую команду и параметры (это одна строка команды!):

```
sudo geth --datadir "/mygeth" --networkid 8956 --ipcdisable --port 30307 --rpc --rpcapi
"eth,web3,personal,net,miner,admin,debug" --rpcport 8507 --mine --minerthreads=1 --
etherbase=0xbaf735f889d603f0ec6b1030c91d9033e60525c3
```

Вновь напомним, что в параметр `etherbase` нужно подставить адрес вашей учетной записи, предварительно добавив в начало строки символы `0x`.

Давайте поясним назначение каждого параметра команды geth:

- `datadir` — расположение каталога данных, как в предыдущих шагах;
- `networkid` — это идентификатор сети, который отличает нашу частную цепочку блоков от других блокчейнов Ethereum. Параметр по назначению похож на `chainId`, который мы определили в файле genesis.json, но обеспечивает еще один уровень разделения между сетями. Как можно видеть, мы использовали для этого значения другой пользовательский номер;
- `ipcdisable` — с помощью этого параметра мы отключили межпроцессный коммуникационный порт для geth, чтобы при запуске нескольких экземпляров geth (узлов) на одном и том же локальном компьютере не возникало никаких конфликтов;
- `port` — мы выбрали нестандартное значение номера порта для взаимодействия с geth;
- `rpc`, `rpcapi`, `rpcport` — эти три параметра определяют конфигурацию для API RPC, предоставляемого geth. Мы хотим, чтобы API-интерфейсы geth `eth`, `web3`, `personal`, `net`, `miner`, `admin`, `debug` были доступны через RPC, и запускаем RPC на пользовательском порту `8507`;
- `mine`, `minerthreads`, `etherbase` — с помощью этих трех параметров мы даем команду geth запустить этот узел в качестве майнера, ограничить потоки процесса майнера только одним (чтобы мы не потребляли много ресурсов ЦП) и отправить воз-

награждение за майнинг на учетную запись Ethereum, которую мы создали на первом этапе.

Вот и все настройки, которые нам нужны сейчас для запуска нашего первого узла geth для частной сети.

Когда мы запустим эту команду со всеми параметрами, geth выдаст следующий вывод (как на снимке с экрана, показанном на рис. 6.3).

```
author@testserver:~/mygeth$ sudo geth --datadir "/mygeth" --networkid 8956 --ipcdisable --port 30307 --rpc --rpcapi "eth,web3,personal,net,miner,a
dmin,debug" --rpcport 8507 --mine --minerthreads=1 --etherbase=0xbaf735f889d603f0ec6b1030c91d9033e60525c3
INFO [02-11|18:00:55] Starting peer-to-peer node               instance=Geth/v1.7.3-stable-4bb3c89d/linux-amd64/go1.9
INFO [02-11|18:00:55] Allocated cache and file handles         database=/mygeth/geth/chaindata cache=128 handles=1024
WARN [02-11|18:00:55] Upgrading database to use lookup entries
INFO [02-11|18:00:55] Initialised chain configuration          config="{ChainID: 3792 Homestead: 0 DAO: <nil> DAOSupport: false EIP150: <nil> EIP1
55: 0 EIP158: 0 Byzantium: <nil> Engine: unknown}"
INFO [02-11|18:00:55] Database deduplication successful        deduped=0
INFO [02-11|18:00:55] Disk storage enabled for ethash caches   dir=/mygeth/geth/ethash count=3
INFO [02-11|18:00:55] Disk storage enabled for ethash DAGs     dir=/home/author/.ethash count=2
INFO [02-11|18:00:55] Initialising Ethereum protocol           versions="[63 62]" network=8956
INFO [02-11|18:00:55] Loaded most recent local header          number=0 hash=294e0a…1f4aa7 td=2000
INFO [02-11|18:00:55] Loaded most recent local full block      number=0 hash=294e0a…1f4aa7 td=2000
INFO [02-11|18:00:55] Loaded most recent local fast block      number=0 hash=294e0a…1f4aa7 td=2000
INFO [02-11|18:00:55] Regenerated local transaction journal    transactions=0 accounts=0
INFO [02-11|18:00:55] Starting P2P networking
INFO [02-11|18:00:57] UDP listener up                          self=enode://e03b50e9b1b2579904f2bbdff7dd0826bd4e4eb2e225c1d1cb1a765195474d7418f3e8
fbfeefd55bd85722973d17626f0e53208c62e38d1099bb583e702b3b48@[::]:30307
INFO [02-11|18:00:57] RLPx listener up                         self=enode://e03b50e9b1b2579904f2bbdff7dd0826bd4e4eb2e225c1d1cb1a765195474d7418f3e8
fbfeefd55bd85722973d17626f0e53208c62e38d1099bb583e702b3b48@[::]:30307
INFO [02-11|18:00:57] HTTP endpoint opened: http://127.0.0.1:8507
INFO [02-11|18:00:57] Transaction pool price threshold updated price=18000000000
INFO [02-11|18:00:57] Starting mining operation
INFO [02-11|18:00:57] Commit new mining work                   number=1 txs=0 uncles=0 elapsed=142µs
INFO [02-11|18:00:59] Generating DAG in progress               epoch=0 percentage=0 elapsed=793.271ms
INFO [02-11|18:01:00] Generating DAG in progress               epoch=0 percentage=1 elapsed=1.592s
```

**Рис. 6.3.** Запуск через geth первого узла

Обратите внимание на строку журнала прослушивания UDP в выводе терминала:

```
INFO [02-11|18:00:57] UDP listener up
self=enode://e03b50e9b1b2579904f2bbdff7dd0826bd4e4eb2e225c1d1cb1a765195474d7418f3e8fbfeefd55bd8572
2973d17626f0e53208c62e38d1099bb583e702b3b48@[::]:30307
```

Эта строка содержит адрес узла, который мы только что запустили:

```
enode://e03b50e9b1b2579904f2bbdff7dd0826bd4e4eb2e225c1d1cb1a765195474d7418f3e8fbfeefd55bd85722973d
17626f0e53208c62e38d1099bb583e702b3b48@[::]:30307
```

Скопируйте этот адрес и сохраните его в надежном месте. Он пригодится нам для подключения к этому узлу других узлов.

Обратите внимание на символы `[::]` перед номером порта, который мы определили в команде. Замените этот фрагмент адреса на локальный IP-адрес хоста, если будете запускать другой узел на той же машине, или замените его на внешний IP-адрес машины, на которой запущен узел. Поскольку мы собираемся запустить другой сетевой узел на том же компьютере (для целей разработки), мы подставим вместо фрагмента `[::]` IP-адрес локальной машины (localhost: `127.0.0.1`). Тогда адрес первого узла будет выглядеть следующим образом:

```
enode://e03b50e9b1b2579904f2bbdff7dd0826bd4e4eb2e225c1d1cb1a765195474d7418f3e8fbfeefd55bd85722973d
17626f0e53208c62e38d1099bb583e702b3b48@127.0.0.1:30307
```

## 6.2.6. Запуск второго узла частной сети

Не бывает сети, состоящей из одного узла. В сети должно быть как минимум два узла, не так ли? Тогда давайте запустим еще один экземпляр geth на той же машине. Новый узел будет взаимодействовать с узлом, который мы запустили раньше, и оба этих узла вместе сформируют нашу частную сеть Ethereum.

Для запуска другого узла нам, прежде всего, нужен еще один каталог, который можно задать как каталог данных второго узла. Давайте создадим его:

```
mkdir mygeth2
```

Теперь мы инициализируем этот узел с той же конфигурацией genesis.json, которую создали для первого узла. Скопируйте файл genesis.json и сохраните его в новом каталоге, который мы только что создали. Перейдите в новый каталог:

```
cd mygeth2
```

Теперь инициализируйте конфигурацию генезиса для второго узла:

```
sudo geth --datadir "/mygeth2" init genesis.json
```

И мы получим в окне терминала вывод (рис. 6.4), аналогичный полученному ранее для первого узла (см. рис. 6.2).

```
author@testserver:~/mygeth2$ sudo geth --datadir "/mygeth2" init genesis.json
WARN [02-11|18:19:15] No etherbase set and no accounts found as default
INFO [02-11|18:19:15] Allocated cache and file handles         database=/mygeth2/geth/chaindata cache=16 handles=16
INFO [02-11|18:19:15] Writing custom genesis block
INFO [02-11|18:19:15] Successfully wrote genesis state         database=chaindata                hash=294e0a…1f4aa7
INFO [02-11|18:19:15] Allocated cache and file handles         database=/mygeth2/geth/lightchaindata cache=16 handles=16
INFO [02-11|18:19:15] Writing custom genesis block
INFO [02-11|18:19:15] Successfully wrote genesis state         database=lightchaindata           hash=294e0a…1f4aa7
```

**Рис. 6.4.** Инициализация второго узла geth с конфигурацией согласно файлу genesis.json

Теперь наш второй узел также инициализируется с помощью начальной конфигурации. Давайте запустим новый узел.

Для запуска второго узла мы передадим в команду geth несколько параметров. Этот второй узел не будет работать как майнер, поэтому мы пропустим последние три параметра из команды, которой запускали первый узел. Кроме того, мы хотим открыть консоль geth во время работы этого узла, поэтому добавим в команду параметр `console`. В результате команда для запуска второго узла будет выглядеть так:

```
sudo geth --datadir "/mygeth2" --networkid 8956 --ipcdisable --port 30308 --rpc --rpcapi
"eth,web3,personal,net,miner,admin,debug" --rpcport 8508 console
```

Как видите, для второго узла были изменены каталог данных и порты. Мы также добавили в команду флаг консоли, чтобы запустить консоль geth для этого узла.

Когда мы введем в окне терминала эту команду, второй узел также стартует, и мы увидим в терминале следующий вывод (рис. 6.5).

```
author@testserver:~/mygeth2$ sudo geth --datadir "/mygeth2" --networkid 8956 --ipcdisable --port 30308 --rpc --rpcapi "eth,web3,p
ersonal,net,miner,admin,debug" --rpcport 8508 console
WARN [02-11|18:20:27] No etherbase set and no accounts found as default
INFO [02-11|18:20:27] Starting peer-to-peer node               instance=Geth/v1.7.3-stable-4bb3c89d/linux-amd64/go1.9
INFO [02-11|18:20:27] Allocated cache and file handles         database=/mygeth2/geth/chaindata cache=128 handles=1024
WARN [02-11|18:20:27] Upgrading database to use lookup entries
INFO [02-11|18:20:27] Initialised chain configuration          config="{ChainID: 3792 Homestead: 0 DAO: <nil> DAOSupport: false E
IP150: <nil> EIP155: 0 EIP158: 0 Byzantium: <nil> Engine: unknown}"
INFO [02-11|18:20:27] Disk storage enabled for ethash caches   dir=/mygeth2/geth/ethash count=3
INFO [02-11|18:20:27] Disk storage enabled for ethash DAGs     dir=/home/author/.ethash count=2
INFO [02-11|18:20:27] Initialising Ethereum protocol           versions="[63 62]" network=8956
INFO [02-11|18:20:27] Database deduplication successful        deduped=0
INFO [02-11|18:20:27] Loaded most recent local header          number=0 hash=294e0a…1f4aa7 td=2000
INFO [02-11|18:20:27] Loaded most recent local full block      number=0 hash=294e0a…1f4aa7 td=2000
INFO [02-11|18:20:27] Loaded most recent local fast block      number=0 hash=294e0a…1f4aa7 td=2000
INFO [02-11|18:20:27] Regenerated local transaction journal    transactions=0 accounts=0
INFO [02-11|18:20:27] Starting P2P networking
INFO [02-11|18:20:29] UDP listener up                          self=enode://e7ba0ca680ab92e0d665a95edca226dc824230cab7b0bdc403524
32522075d555b38afa9033deaa367377f5b5c53ae0ada377c9795f2840b73ffa1c579433e060[::]:30308
INFO [02-11|18:20:29] RLPx listener up                         self=enode://e7ba0ca680ab92e0d665a95edca226dc824230cab7b0bdc403524
32522075d555b38afa9033deaa367377f5b5c53ae0ada377c9795f2840b73ffa1c579433e060[::]:30308
INFO [02-11|18:20:29] HTTP endpoint opened: http://127.0.0.1:8508
Welcome to the Geth JavaScript console!

instance: Geth/v1.7.3-stable-4bb3c89d/linux-amd64/go1.9
 modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

>
```

**Рис. 6.5.** Запуск второго узла частной сети

В настоящее время оба наших узла geth работают, но не знают друг о друге. Если мы запустим команду `admin.peers` в консоли geth второго узла, то получим пустой массив (рис. 6.6).

```
> admin.peers
[]
>
```

**Рис. 6.6.** Консоль geth: список подключенных узлов пока пуст

Это означает, что узлы не связаны друг с другом. Давайте соединим их. Для этого мы отправим команду:

```
admin.addPeer(...)
```

из консоли geth второго узла, указав адрес первого узла в качестве параметра. Помните, вы сохранили адрес первого узла после его запуска? Итак, запустим в консоли geth второго узла следующую команду:

```
admin.addPeer("enode://e03b50e9b1b2579904f2bbdff7dd0826bd4e4eb2e225c1d1cb1a765195474d7418f3e8fbfee
fd55bd85722973d17626f0e53208c62e38d1099bb583e702b3b48@127.0.0.1:30307")
```

Разумеется, вы должны подставить в команду свой адрес первого узла.

И как только мы запустим эту команду на втором узле, она возвратит `true`. Также через несколько секунд начнется синхронизация с первым узлом. На следующем снимке экрана (рис. 6.7) показан вывод консоли второго узла.

Теперь оба наших узла подключены и сформировали частную сеть Ethereum. Для дальнейшей проверки мы снова запустим команду `admin.peers` на втором узле,

```
> admin.addPeer("enode://e03b50e9b1b2579904f2bbdff7dd0826bd4e4eb2e225c1d1cb1a765195474d741
8f3e8fbfeefd55bd85722973d17626f0e53208c62e38d1099bb583e702b3b480127.0.0.1:30307")
true
> INFO [02-11|18:34:46] Block synchronisation started
INFO [02-11|18:34:46] Imported new state entries                count=1 elapsed=31µs proces
sed=1 pending=0 retry=0 duplicate=0 unexpected=0
INFO [02-11|18:34:47] Imported new block headers                count=192 elapsed=1.329s nu
mber=192 hash=0939a1…01bc65 ignored=0
INFO [02-11|18:34:47] Imported new block receipts               count=192 elapsed=3.419ms b
ytes=768 number=192 hash=0939a1…01bc65 ignored=0
INFO [02-11|18:34:47] Imported new block headers                count=192 elapsed=306.568ms
 number=384 hash=d4236a…0a25ef ignored=0
```

**Рис. 6.7.** Консоль geth: добавление узла

```
> admin.peers
[{
    caps: ["eth/63"],
    id: "e03b50e9b1b2579904f2bbdff7dd0826bd4e4eb2e225c1d1cb1a765195474d7418f3e8fbfeefd55bd
85722973d17626f0e53208c62e38d1099bb583e702b3b48",
    name: "Geth/v1.7.3-stable-4bb3c89d/linux-amd64/go1.9",
    network: {
      localAddress: "127.0.0.1:53790",
      remoteAddress: "127.0.0.1:30307"
    },
    protocols: {
      eth: {
        difficulty: 120652076,
        head: "0x511dfe69b3e05360ba5deb073e6c34601d6e7266298ccf4c4407aa6683db9285",
        version: 63
      }
    }
}]
```

**Рис. 6.8.** Консоль geth: повторная проверка подключенных узлов сети

и на этот раз мы увидим массив JSON с объектом, показывающим первый узел в качестве равноправного узла (рис. 6.8).

На следующем снимке экрана (рис. 6.9) приведены окна терминалов обоих настроенных нами узлов. Слева показан первый узел, который является майнерским узлом, и, как мы видим, он постоянно добывает новые блоки. Второй узел показан справа, и можно видеть, что он синхронизируется с первым узлом. Из-за большого количества информации детали на этой иллюстрации слишком мелкие, и она приводится лишь с тем, чтобы показать вам, как терминалы получают и показывают журналы с обоих узлов Ethereum.

Итак, оба узла подключены к одноранговой сети, и у нас есть работающий частный блокчейн Ethereum с двумя узлами. У нас также есть учетная запись Ethereum, которая настроена как майнер и предварительно снабжена некоторым количеством эфира. Теперь мы можем создать еще больше учетных записей и передавать между ними транзакции в частном блокчейне.

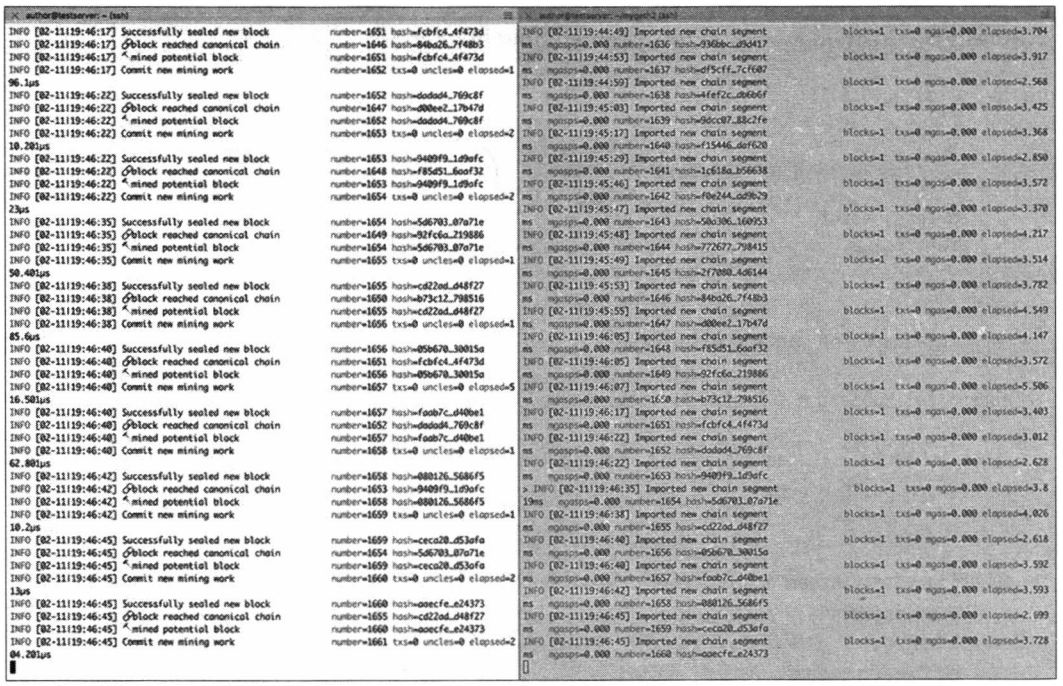


**Рис. 6.9.** Журналы geth для обоих узлов

* * *

В этом разделе мы узнали, как настроить частную сеть Ethereum с двумя узлами. На самом деле в сеть может быть включено любое количество узлов — нам просто нужно выполнять для каждого нового узла один и тот же процесс. В случае удаленных узлов мы должны правильно указать IP-адреса удаленных машин, а также убедиться, что на них открыты необходимые порты, — если на них запущен брандмауэр, блокирующий внешний трафик к машинам.

# 6.3. Создание умного контракта

Теперь, когда у нас настроена и работает частная сеть Ethereum, мы можем перейти к этапу создания умного контракта. Затем мы развернем этот контракт в нашей частной сети. Для создания и развертывания умного контракта мы выполним те же шаги, что и в предыдущей главе.

Итак, запустим онлайновую среду разработки Remix и напишем наш умный контракт на языке Solidity.

Следующий фрагмент кода Solidity (листинг 6.1) представляет собой умный контракт, который выполняет функции опроса.

**Листинг 6.1. Умный контракт для проведения опросов**

```
pragma solidity ^0.4.19;

contract Poll {
    event Voted(
        address _voter,
        uint _value
    );

    mapping(address => uint) public votes;

    string pollSubject = " Должно ли государство освободить кофе от налогов? 1 - ДА, 2 - НЕТ.";

    function getPoll() constant public returns (string) {
        return pollSubject;
    }

    function vote(uint selection) public {
        Voted(msg.sender, selection);

        require (votes[msg.sender] == 0);
        require (selection > 0 && selection < 3);
        votes[msg.sender] = selection;
    }
}
```

Теперь проанализируем исходный код контракта, чтобы понять, как он работает. Как мы видим, контракт называется `Poll`.

Следующие строки кода объявляют событие умного контракта, которое принимает два параметра: один относится к типу адреса Ethereum, а другой — к целому числу без знака. Мы создали событие, чтобы иметь возможность определить, кто и за что проголосовал в опросе (мы вернемся к этому событию позже):

```
event Voted(
    address _voter,
    uint _value
);
```

Далее следует строка кода:

```
mapping(address => uint) public votes;
```

Она сопоставляет адреса Ethereum с целыми числами без знака. Это хранилище данных, в котором мы будем хранить адреса участников опроса и их выбор при голосовании.

В следующем фрагменте кода содержится строка с темой опроса и объявлена функция, которая возвращает значение этой строки, чтобы участники могли понять, за что голосуют:

```
string pollSubject = "Должно ли государство освободить кофе от налогов? 1 - ДА, 2 - НЕТ.";
function getPoll() constant public returns (string) {
    return pollSubject;
}
```

И наконец, у нас есть функция, которая реализует собственно процесс опроса:

```
function vote(uint selection) public {
    Voted(msg.sender, selection);

    require (votes[msg.sender] == 0);
    require (selection > 0 && selection < 3);
    votes[msg.sender] = selection;
}
```

Внимательно изучите каждую строку предыдущего фрагмента:

- во-первых, как только мы входим в эту функцию, мы инициируем созданное нами событие `Voted` с адресом отправителя (избирателя) и значением, которое он выбрал;
- далее, мы ограничиваем один голос на каждого участника, проверяя, равно ли значение голоса нулю для соответствующего адреса. Оператор `require` используется для проверки условий на основе пользовательских данных;
- затем при помощи оператора `require` мы ограничиваем значение выбора вариантами: 1 — «да» и 2 — «нет». Тему голосования мы передали избирателю ранее, в строке `pollSubject`.

Снимок экрана на рис. 6.10 показывает наш умный контракт в редакторе Remix.

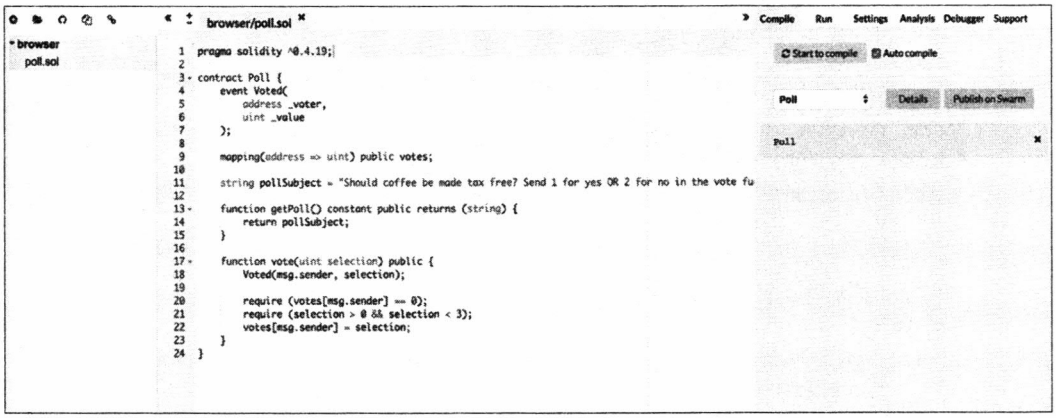



**Рис. 6.10. Текст умного контракта в редакторе Remix**

Скомпилируйте код контракта, используя Remix, а затем скопируйте ABI и байт-код контракта, чтобы развернуть его в нашей частной сети, — именно так, как мы делали это в предыдущей главе:

**ABI контракта:**

```
[
    {
        "constant": true,
        "inputs": [
            {
                "name": "",
                "type": "address"
            }
        ],
        "name": "votes",
        "outputs": [
            {
                "name": "",
                "type": "uint256"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [],
        "name": "getPoll",
        "outputs": [
            {
                "name": "",
                "type": "string"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": false,
                "name": "_voter",
                "type": "address"
            },
            {
                "indexed": false,
                "name": "_value",
```

```
            "type": "uint256"
        }
    ],
    "name": "Voted",
    "type": "event"
},
{
    "constant": false,
    "inputs": [
        {
            "name": "selection",
            "type": "uint256"
        }
    ],
    "name": "vote",
    "outputs": [],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
}
]
```

### И байт-код контракта:

```
{
    "linkReferences": {},
    "object": "6060604052608060405190810160405280605081526020017f53686f756c6420636f666665652062652
06d6164652074617820667265653f2081526020017f53656e64203120666f7220796573204f52203220666f72206e6f206
96e2074688152602001 7f6520766f74652066756e6374696f6e2e000000000000000000000000000000008152506001908
0519060200190 61009c9291906100ad565b5034156100a857600080fd5b610152565b828054600181600116156101000200
3166002900490600052602060002090601f016020900481019282601f106100ee57805160ff191683800117855561011c5
65b8280016001018555821561011c579182015b828111156101 1b57825182559160200191906001019061010056 5b5b509
0506101299190610 12d565b5090565b61014f91905b8082111561014b576000816000905550600101610133565b5090565
b90565b6103738061016160003960 00f300606060405260043610610057576000357c0100000000000000000000000000
0000000000000000000000000000000900463ffffffff1680630121b93f1461005c57806303c322781461007f578063d8bff
5a51461010d575b600080fd5b34156100675760008 0fd5b61007d600480803590602001909190505061015a565b005b341
561008a57600080fd5b610092610273565b604051808060200182810382528381815181526020019150805190602001908
0838360005b838110156100d257808201518184015260208101905061 00b7565b50505050905090810190601f168015610
0ff5780820380516001836020036101000a031916815260200191505b509250505060405180910390f35b3415610118576
00080fd5b610144600480803573ffffffffffffffffffffffffffffffffffffffff169060200190919050506103 1b565b6
040518082815260200191505060405180910390f35b7f4d99b957a2bc29a30ebd96a7be8e68fe50a3c701db28a91436490
b7d53870ca4338260405180837 3ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffff
ffffffffffff16815260200182815260200192505050604051809103 90a160008060003373ffffffffffffffffffffffff
ffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002054141 51
561021257600080fd5b60008111801561022575060038 1105b151561022d57600080fd5b806000803373ffffffffffff
ffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160 00
0208190555050565b61027b610333565b60018054600181600116156101000203166002900480601f01602080910402602
0016040519081016040528092919081815260200182805460018160011615610100020316600290048015610311578060 1
f106102e65761010080835404028352916020019161031 1565b820191906000526020600020905b8154815290600101906
020018083116102f457829003601f168201915b505050505090509056 5b600602052806000526040600020600091509 05
```

05481565b60206040519081016040528060008152509056 00a165627a7a72305820ec7d3e1dae8412ec85045a8eafc248e37ae506802cc008ead300df1ac81aab490029",

"opcodes": "PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x80 PUSH1 0x40 MLOAD SWAP1 DUP2 ADD PUSH1 0x40 MSTORE DUP1 PUSH1 0x50 DUP2 MSTORE PUSH1 0x20 ADD PUSH32 0x53686F756C6420636F6666656520626520 6D6164652074617820667265653F20 DUP2 MSTORE PUSH1 0x20 ADD PUSH32 0x53656E64203120666F7220796573204F52203220666F72206E6F20696E207468 DUP2 MSTORE PUSH1 0x20 ADD PUSH32 0x6520766F74652066756E6374696F6E2E0000000000000000000000000000000000 DUP2 MSTORE POP PUSH1 0x1 SWAP1 DUP1 MLOAD SWAP1 PUSH1 0x20 ADD SWAP1 PUSH2 0x9C SWAP3 SWAP2 SWAP1 PUSH2 0xAD JUMP JUMPDEST POP CALLVALUE ISZERO PUSH2 0xA8 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH2 0x152 JUMP JUMPDEST DUP3 DUP1 SLOAD PUSH1 0x1 DUP2 PUSH1 0x1 AND ISZERO PUSH2 0x100 MUL SUB AND PUSH1 0x2 SWAP1 DIV SWAP1 PUSH1 0x0 MSTORE PUSH1 0x20 PUSH1 0x0 KECCAK256 SWAP1 PUSH1 0x1F ADD PUSH1 0x20 SWAP1 DIV DUP2 ADD SWAP3 DUP3 PUSH1 0x1F LT PUSH2 0xEE JUMPI DUP1 MLOAD PUSH1 0xFF NOT AND DUP4 DUP1 ADD OR DUP6 SSTORE PUSH2 0x11C JUMP JUMPDEST DUP3 DUP1 ADD PUSH1 0x1 ADD DUP6 SSTORE DUP3 ISZERO PUSH2 0x11C JUMPI SWAP2 DUP3 ADD JUMPDEST DUP3 DUP2 GT ISZERO PUSH2 0x11B JUMPI DUP3 MLOAD DUP3 SSTORE SWAP2 PUSH1 0x20 ADD SWAP2 SWAP1 PUSH1 0x1 ADD SWAP1 PUSH2 0x100 JUMP JUMPDEST JUMPDEST POP SWAP1 POP PUSH2 0x129 SWAP2 SWAP1 PUSH2 0x12D JUMP JUMPDEST POP SWAP1 JUMP JUMPDEST PUSH2 0x14F SWAP2 SWAP1 JUMPDEST DUP1 DUP3 GT ISZERO PUSH2 0x14B JUMPI PUSH1 0x0 DUP2 PUSH1 0x0 SWAP1 SSTORE POP PUSH1 0x1 ADD PUSH2 0x133 JUMP JUMPDEST POP SWAP1 JUMP JUMPDEST SWAP1 JUMP JUMPDEST PUSH2 0x373 DUP1 PUSH2 0x161 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH2 0x57 JUMPI PUSH1 0x0 CALLDATALOAD PUSH29 0x100000000000000000000000000000000000000000000000000000000 SWAP1 DIV PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x121B93F EQ PUSH2 0x5C JUMPI DUP1 PUSH4 0x3C32278 EQ PUSH2 0x7F JUMPI DUP1 PUSH4 0xD8BFF5A5 EQ PUSH2 0x10D JUMPI JUMPDEST PUSH1 0x0 DUP1 REVERT JUMPDEST CALLVALUE ISZERO PUSH2 0x67 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH2 0x7D PUSH1 0x4 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20 ADD SWAP1 SWAP2 SWAP1 POP POP PUSH2 0x15A JUMP JUMPDEST STOP JUMPDEST CALLVALUE ISZERO PUSH2 0x8A JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH2 0x92 PUSH2 0x273 JUMP JUMPDEST PUSH1 0x40 MLOAD DUP1 DUP1 PUSH1 0x20 ADD DUP3 DUP2 SUB DUP3 MSTORE DUP4 DUP2 DUP2 MLOAD DUP2 MSTORE PUSH1 0x20 ADD SWAP2 POP DUP1 MLOAD SWAP1 PUSH1 0x20 ADD SWAP1 DUP1 DUP4 DUP4 PUSH1 0x0 JUMPDEST DUP4 DUP2 LT ISZERO PUSH2 0xD2 JUMPI DUP1 DUP3 ADD MLOAD DUP2 DUP5 ADD MSTORE PUSH1 0x20 DUP2 ADD SWAP1 POP PUSH2 0xB7 JUMP JUMPDEST POP POP POP POP SWAP1 POP SWAP1 DUP2 ADD SWAP1 PUSH1 0x1F AND DUP1 ISZERO PUSH2 0xFF JUMPI DUP1 DUP3 SUB DUP1 MLOAD PUSH1 0x1 DUP4 PUSH1 0x20 SUB PUSH2 0x100 EXP SUB NOT AND DUP2 MSTORE PUSH1 0x20 ADD SWAP2 POP JUMPDEST POP SWAP3 POP POP POP PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 RETURN JUMPDEST CALLVALUE ISZERO PUSH2 0x118 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH2 0x144 PUSH1 0x4 DUP1 DUP1 CALLDATALOAD PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND SWAP1 PUSH1 0x20 ADD SWAP1 SWAP2 SWAP1 POP POP PUSH2 0x31B JUMP JUMPDEST PUSH1 0x40 MLOAD DUP1 DUP3 DUP2 MSTORE PUSH1 0x20 ADD SWAP2 POP POP PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 RETURN JUMPDEST PUSH32 0x4D99B957A2BC29A30EBD96A7BE8E68FE50A3C701DB28A91436490B7D53870CA4 CALLER DUP3 PUSH1 0x40 MLOAD DUP1 DUP4 PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND DUP2 MSTORE PUSH1 0x20 ADD DUP3 DUP2 MSTORE PUSH1 0x20 ADD SWAP3 POP POP POP PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 LOG1 PUSH1 0x0 DUP1 PUSH1 0x0 CALLER PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND DUP2 MSTORE PUSH1 0x20 ADD SWAP1 DUP2 MSTORE PUSH1 0x20 ADD PUSH1 0x0 KECCAK256 SLOAD EQ ISZERO ISZERO PUSH2 0x212 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH1 0x0 DUP2 GT DUP1 ISZERO PUSH2 0x222 JUMPI POP PUSH1 0x3 DUP2 LT JUMPDEST ISZERO ISZERO PUSH2 0x22D JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST DUP1 PUSH1 0x0 DUP1 CALLER PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND DUP2 MSTORE PUSH1 0x20 ADD SWAP1 DUP2 MSTORE PUSH1 0x20 ADD PUSH1 0x0 KECCAK256 DUP2 SWAP1 SSTORE POP POP JUMP JUMPDEST PUSH2 0x27B PUSH2 0x333 JUMP JUMPDEST PUSH1 0x1 DUP1 SLOAD PUSH1 0x1 DUP2 PUSH1 0x1 AND ISZERO PUSH2 0x100 MUL SUB AND PUSH1 0x2 SWAP1 DIV DUP1 PUSH1 0x1F ADD PUSH1 0x20 DUP1 SWAP2 DIV MUL PUSH1 0x20 ADD PUSH1 0x40 MLOAD SWAP1 DUP2 ADD PUSH1 0x40 MSTORE DUP1 SWAP3 SWAP2 SWAP1 DUP2 DUP2 MSTORE PUSH1 0x20 ADD DUP3 DUP1 SLOAD PUSH1 0x1 DUP2 PUSH1 0x1 AND ISZERO PUSH2 0x100 MUL SUB AND PUSH1 0x2 SWAP1 DIV DUP1 ISZERO PUSH2 0x311 JUMPI DUP1 PUSH1 0x1F LT PUSH2 0x2E6 JUMPI PUSH2 0x100 DUP1 DUP4 SLOAD DIV MUL DUP4 MSTORE SWAP2 PUSH1 0x20 ADD SWAP2 PUSH2 0x311 JUMP JUMPDEST DUP3 ADD SWAP2 SWAP1 PUSH1 0x0 MSTORE PUSH1 0x20 PUSH1 0x0 KECCAK256

```
SWAP1 JUMPDEST DUP2 SLOAD DUP2 MSTORE SWAP1 PUSH1 0x1 ADD SWAP1 PUSH1 0x20 ADD DUP1 DUP4 GT PUSH2
0x2F4 JUMPI DUP3 SWAP1 SUB PUSH1 0x1F AND DUP3 ADD SWAP2 JUMPDEST POP POP POP POP POP SWAP1 POP
SWAP1 JUMP JUMPDEST PUSH1 0x0 PUSH1 0x20 MSTORE DUP1 PUSH1 0x0 MSTORE PUSH1 0x40 PUSH1 0x0
KECCAK256 PUSH1 0x0 SWAP2 POP SWAP1 POP SLOAD DUP2 JUMP JUMPDEST PUSH1 0x20 PUSH1 0x40 MLOAD SWAP1
DUP2 ADD PUSH1 0x40 MSTORE DUP1 PUSH1 0x0 DUP2 MSTORE POP SWAP1 JUMP STOP LOG1 PUSH6
0x627A7A723058 KECCAK256 0xec PUSH30
0x3E1DAE8412EC85045A8EAFC248E37AE506802CC008EAD300DF1AC81AAB49 STOP 0x29 ",
    "sourceMap": "26:576:0:-;;;167:103;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;:::i;:::-
;;26:576;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;:::i;:::-;;;:::o;:::-;;;;;;;;;;;;;;;;;;;;;;;;;;;;:::o;:::-;;;;;;;"
}
```

Теперь наш умный контракт готов к развертыванию в сети Ethereum.

---

**ВАЖНОЕ ПРИМЕЧАНИЕ.** Созданный контракт, разумеется, нуждается в доработке, чтобы сделать его более безопасным и экономичным. Приведенный здесь код Solidity лишь демонстрирует принцип создания контрактов, и его не следует брать за основу при разработке практических приложений. Подробное обсуждение программирования на Solidity выходит за рамки нашей книги. Мы рекомендуем обратиться к официальной документации по Solidity и к специализированным руководствам.

---

## 6.4. Развертывание умного контракта

В этом разделе мы рассказываем о развертывании умного контракта в частной сети Ethereum, которую мы создали ранее. Процесс развертывания контракта такой же, как и в предыдущей главе. Мы используем уже знакомую вам библиотеку web3.js для программирования на платформе Ethereum с использованием JavaScript. Разница лишь в том, что на этот раз мы размещаем контракт в частной сети, а не в публичной. Мы рекомендуем читателям предварительно внимательно изучить *главу 5*, если они этого еще не сделали.

### 6.4.1. Настройка библиотеки web3 и подключения

Прежде всего, мы установим библиотеку web3 в среде node.js. Среда выполнения node.js будет использоваться для развертывания умного контракта. Именно так мы и сделали в предыдущей главе:

```
npm install web3@1.0.0-beta.28
```

После установки мы сначала инициализируем библиотеку и создаем экземпляр web3:

```
var Web3 = require ('web3');
var web3 = new Web3 (new Web3.providers.HttpProvider ('http://127.0.0.1:8507'));
```

Заметим, что на этот раз наш HTTP-провайдер для экземпляра web3 изменился на локальную конечную точку (`127.0.0.1:8507`) вместо общедоступной конечной точки Infura, которую мы использовали в предыдущей главе. Дело в том, что мы сейчас подключаемся к нашей локальной частной сети. Также обратите внимание, мы ис-

пользуем порт 8507, который указали в параметре --rpcport при настройке первого узла нашей частной сети. Это означает, что мы подключаемся к первому узлу сети.

## 6.4.2. Развертывание контракта в частной сети

Теперь, когда у нас есть умный контракт и его данные, мы подготовим объект контракта web3, а затем развернем этот контракт в блокчейне Ethereum, вызвав метод deploy для объекта контракта.

Нам нужно создать объект класса web3.eth.Contract, который может представлять наш контракт. Следующий фрагмент кода создает экземпляр контракта, используя ABI нашего контракта в качестве входных данных для конструктора:

```
var pollingContract = new web3.eth.Contract([
    {
        "constant": true,
        "inputs": [
            {
                "name": "",
                "type": "address"
            }
        ],
        "name": "votes",
        "outputs": [
            {
                "name": "",
                "type": "uint256"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [],
        "name": "getPoll",
        "outputs": [
            {
                "name": "",
                "type": "string"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
```

```
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": false,
                "name": "_voter",
                "type": "address"
            },
            {
                "indexed": false,
                "name": "_value",
                "type": "uint256"
            }
        ],
        "name": "Voted",
        "type": "event"
    },
    {
        "constant": false,
        "inputs": [
            {
                "name": "selection",
                "type": "uint256"
            }
        ],
        "name": "vote",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    }
]);
```

**Теперь нам нужно развернуть этот контракт в сети Ethereum, используя метод** `deploy` **библиотеки web3. Следующий фрагмент кода показывает, как это сделать. В этом фрагменте мы вставили байт-код контракта в поле** `data` **объекта, переданного методу развертывания:**

```
pollingContract
    .deploy({
        data: '0x6060604052608060405190810160405280605081526020017f53686f756c6420636f6666656520626
5206d6164652074617820667265653f2081526020017f53656e64203120666f7220796573204f52203220666f72206e6f2
0696e2074688152602001 7f6520766f74652066756e6374696f6e2e000000000000000000000000000000081525060019
080519060200190610 09c9291906100ad565b5034156100a857600080fd5b610152565b828054600181600116156101000
203166002900490600052602060002090601f016020900481019282601f106100ee57805160ff1916838001178555610 11
c565b828001600101855582156101 1c579182015b8281111561011b57825182559160200191906001019061010056 5b5b5
090506101299190610 12d565b5090565b61014f91905b8082111561014b576000816000905550600101610133565b50905
65b90565b6103738061016160003960 00f300606060405260043610610057576000357c0100000000000000000000000000
```

```
000000000000000000000000000000900463ffffffff1680630121b93f1461005c57806303c322781461007f578063d8b
ff5a51461010d575b600080fd5b34156100675760080fd5b61007d60048080359060200190919050505061015a565b005b3
41561008a57600080fd5b610092610273565b604051808060200182810382528381815181526020019150805190602001
0808383600005b838110156100d257808201518184015260208101905061006b7565b505050509050908101906601f1680156
100ff57808203805160018360200361010000a031916815260200191505b509250505060405180910390f35b34156101185
7600080fd5b610144600480803573ffffffffffffffffffffffffffffffffffffffff169060200190919050505061031b565
b604051808281526020019150506040518091039035b7f4d99b957a2bc29a30ebd96a7be8e68fe50a3c701db28a914364
90b7d53870ca43382604051808373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffff
ffffffffffffff168152602001828152602001925050506040518091039a160008060003373ffffffffffffffffffffff
ffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152602001600020541415
515610212576000080fd5b60008111801561022257506003811105b151561022d57600080fd5b806000803373ffffffffff
ffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152602001600020016
000208190555050565b61027b610333565b60018054600181600116156101000203166002900480601f016020809104026
020016040519081016040528092919081815260200182805460018160011615610100020316600290048015610311578060
01f106102e65761010080835404028352916020019161031156 5b820191906000526020600020905b81548152906001019
060200180831161 02f457829003601f168201915b505050505090509056 5b600602052806000526040600020600091509
0505481565b602060405190810160405280600081525090560 0a165627a7a72305820ec7d3e1dae8412ec85045a8eafc24
8e37ae506802cc008ead300df1ac81aab490029'
    })
    .send({
        from: '0xbaf735f889d603f0ec6b1030c91d9033e60525c3',
        gas: 4700000,
        gasPrice: '20000000000000'
    },
    function(error, transactionHash){
        console.log(error);
        console.log(transactionHash);
    })
    .then(function(contract){
        console.log(contract);
    });
```

**Обратите внимание, что мы также подставили адрес учетной записи, созданной во время настройки сети, в поле `from` функции отправки. Поскольку эта учетная запись была предварительно снабжена девятью эфирами и добавлена в качестве учетной записи `etherbase` для выплаты вознаграждений за майнинг, у нее достаточно денег для развертывания контракта.**

**Готовая функция для развертывания контракта выглядит следующим образом:**

```
var deployContract = function () {
    var pollingContract = new web3.eth.Contract([{
            "constant": true,
            "inputs": [{
                "name": "",
                "type": "address"
            }],
            "name": "votes",
            "outputs": [{
                "name": "",
```

```
            "type": "uint256"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [],
        "name": "getPoll",
        "outputs": [{
            "name": "",
            "type": "string"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "anonymous": false,
        "inputs": [{
                "indexed": false,
                "name": "_voter",
                "type": "address"
            },
            {
                "indexed": false,
                "name": "_value",
                "type": "uint256"
            }
        ],
        "name": "Voted",
        "type": "event"
    },
    {
        "constant": false,
        "inputs": [{
            "name": "selection",
            "type": "uint256"
        }],
        "name": "vote",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    }
]);
```

```
    pollingContract
        .deploy({
            data: '0x6060604052608060405190810160405280605081526020017f53686f756c6420636f666665652
06265206d6164652074617820667265653f2081526020017f53656e64203120666f7220796573204f52203220666f72206
e6f20696e2074688152602001 7f6520766f74652066756e6374696f6e2e000000000000000000000000000000008152506
0019080519060200190 61009c9291906100ad565b5034156100a857600080fd5b610152565b828054600181600116156 10
1000203166002900490600052602060002090601f016020900481019282601f106100ee57805160ff19168380011785556
1011c565b828001600101855582156101 1c579182015b8281111561011b57825182559160200191906001019061010056 5
b5b5090506101299190610 12d565b5090565b61014f91905b8082111561014b576000816000905550600101610133565b5
090565b90565b6103738061016160003960 00f300606060405260043610610057576000357c0100000000000000000000000
0000000000000000000000000000000000000900463ffffffff1680630121b93f1461005c57806303c322781461007f57806
3d8bff5a51461010d575b600080fd5b34156100675760008 0fd5b61007d600480803590602001909190505061015a565b0
05b341561008a57600080fd5b610092610273565b604051808060200182810382528381815181526020019150805190602
0019080838360005b838110156100d2578082015181840152602081019050 6100b7565b505050509050908101906 01f168
0156100ff57808203805160018360200361010 00a031916815260200191505b509250505060405180910390f35b3415610
11857600080fd5b6101446004808035 73ffffffffffffffffffffffffffffffffffffffff169060200190919050505061031
b565b604051808281526020019150506040518091039 0f35b7f4d99b957a2bc29a30ebd96a7be8e68fe50a3c701db28a91
436490b7d53870ca43382604051808373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffff
ffffffffffffffffffff1681526020018281526020019250505060405180910390a160008060003373ffffffffffffffff
ffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152602001600020 5
414151561021257600080fd5b600081118015610222575060038 1105b151561022d57600080fd5b806000803373ffffffff
ffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152602
0016000208190555050565b61027b610333565b60018054600181600116156101000203166002900480601f01602080910
4026020016040519081016040528092919081815260020018280546001816001161561010002031660029004801561031 5
780601f106102e6576101008083540402835291602001916103115 65b8201919060005260206000020905b8154815290600
1019060200180831161 02f457829003601f168201915b505050505050905090565b6000602052806000526040600020600 9
15090505481565b602060405190810160405280600081525090560 0a165627a7a72305820ec7d3e1dae8412ec85045a8ea
fc248e37ae506802cc008ead300df1ac81aab490029'
        })
        .send({
                from: '0xbaf735f889d603f0ec6b1030c91d9033e60525c3',
                gas: 4700000,
                gasPrice: '20000000000000'
            },
            function (error, transactionHash) {
                console.log(error);
                console.log(transactionHash);
            })
        .then(function (contract) {
            console.log(contract);
        });
};
```

**После выполнения этой функции в среде node.js мы получим следующий вывод:**

```
Contract {
  currentProvider: [Getter/Setter],
  _requestManager:
   RequestManager {
     provider: null,
```

```
    providers:
     { WebsocketProvider: [Function: WebsocketProvider],
       HttpProvider: [Function: HttpProvider],
       IpcProvider: [Function: IpcProvider] },
    subscriptions: {} },
givenProvider: null,
providers:
 { WebsocketProvider: [Function: WebsocketProvider],
   HttpProvider: [Function: HttpProvider],
   IpcProvider: [Function: IpcProvider] },
_provider: null,
setProvider: [Function],
BatchRequest: [Function: bound Batch],
extend:
 { [Function: ex]
   formatters:
    { inputDefaultBlockNumberFormatter: [Function: inputDefaultBlockNumberFormatter],
      inputBlockNumberFormatter: [Function: inputBlockNumberFormatter],
      inputCallFormatter: [Function: inputCallFormatter],
      inputTransactionFormatter: [Function: inputTransactionFormatter],
      inputAddressFormatter: [Function: inputAddressFormatter],
      inputPostFormatter: [Function: inputPostFormatter],
      inputLogFormatter: [Function: inputLogFormatter],
      inputSignFormatter: [Function: inputSignFormatter],
      outputBigNumberFormatter: [Function: outputBigNumberFormatter],
      outputTransactionFormatter: [Function: outputTransactionFormatter],
      outputTransactionReceiptFormatter: [Function: outputTransactionReceiptFormatter],
      outputBlockFormatter: [Function: outputBlockFormatter],
      outputLogFormatter: [Function: outputLogFormatter],
      outputPostFormatter: [Function: outputPostFormatter],
      outputSyncingFormatter: [Function: outputSyncingFormatter] },
   utils:
    { _fireError: [Function: _fireError],
      _jsonInterfaceMethodToString: [Function: _jsonInterfaceMethodToString],
      randomHex: [Function: randomHex],
      _: [Function],
      BN: [Function],
      isBN: [Function: isBN],
      isBigNumber: [Function: isBigNumber],
      isHex: [Function: isHex],
      isHexStrict: [Function: isHexStrict],
      sha3: [Function],
      keccak256: [Function],
      soliditySha3: [Function: soliditySha3],
      isAddress: [Function: isAddress],
      checkAddressChecksum: [Function: checkAddressChecksum],
      toChecksumAddress: [Function: toChecksumAddress],
```

```
      toHex: [Function: toHex],
      toBN: [Function: toBN],
      bytesToHex: [Function: bytesToHex],
      hexToBytes: [Function: hexToBytes],
      hexToNumberString: [Function: hexToNumberString],
      hexToNumber: [Function: hexToNumber],
      toDecimal: [Function: hexToNumber],
      numberToHex: [Function: numberToHex],
      fromDecimal: [Function: numberToHex],
      hexToUtf8: [Function: hexToUtf8],
      hexToString: [Function: hexToUtf8],
      toUtf8: [Function: hexToUtf8],
      utf8ToHex: [Function: utf8ToHex],
      stringToHex: [Function: utf8ToHex],
      fromUtf8: [Function: utf8ToHex],
      hexToAscii: [Function: hexToAscii],
      toAscii: [Function: hexToAscii],
      asciiToHex: [Function: asciiToHex],
      fromAscii: [Function: asciiToHex],
      unitMap: [Object],
      toWei: [Function: toWei],
      fromWei: [Function: fromWei],
      padLeft: [Function: leftPad],
      leftPad: [Function: leftPad],
      padRight: [Function: rightPad],
      rightPad: [Function: rightPad],
      toTwosComplement: [Function: toTwosComplement] },
   Method: [Function: Method] },
clearSubscriptions: [Function],
options:
 { address: [Getter/Setter],
   jsonInterface: [Getter/Setter],
   data: undefined,
   from: undefined,
   gasPrice: undefined,
   gas: undefined },
defaultAccount: [Getter/Setter],
defaultBlock: [Getter/Setter],
methods:
 { votes: [Function: bound _createTxObject],
   '0xd8bff5a5': [Function: bound _createTxObject],
   'votes(address)': [Function: bound _createTxObject],
   getPoll: [Function: bound _createTxObject],
   '0x03c32278': [Function: bound _createTxObject],
   'getPoll()': [Function: bound _createTxObject],
   vote: [Function: bound _createTxObject],
```

```
    '0x0121b93f': [Function: bound _createTxObject],
    'vote(uint256)': [Function: bound _createTxObject] },
  events:
   { Voted: [Function: bound ],
    '0x4d99b957a2bc29a30ebd96a7be8e68fe50a3c701db28a91436490b7d53870ca4': [Function: bound ],
    'Voted(address,uint256)': [Function: bound ],
    allEvents: [Function: bound ] },
  _address: '0x59E7161646C3436DFdF5eBE617B4A172974B481e',
  _jsonInterface:
   [ { constant: true,
       inputs: [Array],
       name: 'votes',
       outputs: [Array],
       payable: false,
       stateMutability: 'view',
       type: 'function',
       signature: '0xd8bff5a5' },
     { constant: true,
       inputs: [],
       name: 'getPoll',
       outputs: [Array],
       payable: false,
       stateMutability: 'view',
       type: 'function',
       signature: '0x03c32278' },
     { anonymous: false,
       inputs: [Array],
       name: 'Voted',
       type: 'event',
       signature: '0x4d99b957a2bc29a30ebd96a7be8e68fe50a3c701db28a91436490b7d53870ca4' },
     { constant: false,
       inputs: [Array],
       name: 'vote',
       outputs: [],
       payable: false,
       stateMutability: 'nonpayable',
       type: 'function',
       signature: '0x0121b93f' } ] }
```

**Выходные данные содержат различные свойства контракта, который мы развернули в нашей частной сети. Наиболее важным свойством является адрес, по которому развернут контракт:** `0x59E7161646C3436DFdF5eBE617B4A172974B481e`.

**ABI и адрес контракта могут использоваться для вызова функции контракта. В следующем разделе мы создадим простое веб-приложение, которое вызывает функцию голосования умного контракта и демонстрирует, как опрос может быть выполнен через клиентский интерфейс.**

## 6.5. Клиентское веб-приложение

Как и в предыдущей главе, мы воспользуемся библиотекой web3 для вызова функции умного контракта. Но в прошлый раз мы сделали это, используя среду node.js, а не браузер. Теперь мы рассмотрим другой вариант и запустим вызов функции умного контракта в браузерном приложении.

Самым простым веб-приложением, которое мы можем задействовать в качестве клиентского интерфейса, является отдельная веб-страница с несколькими текстовыми и кнопочными элементами управления. Для формирования веб-страницы мы можем взять приведенный в листинге 6.2 код HTML-файла, а затем отобразить его в браузере с локального сервера. Заметим, что именно загрузка веб-страницы с локального сервера, а не открывание файла напрямую в браузере имеет значение для правильной загрузки скриптов с точки зрения системы безопасности браузера.

**Листинг 6.2. Файл веб-страницы клиентской части приложения для голосования**

```
<html>
<head>
    <meta charset="UTF-8">
    <title>Изучаем умные контракты</title>
    <script src="<адрес кода библиотеки web3 на сервере или локальной машине>"></script>
</head>

<body>
    <div>
        <p>
            <strong>Изучаем умные контракты</strong>
        </p>
        <p>Добро пожаловать в приложение для голосования!</p>
        <p> </p>
        <p>Узнать тему опроса: 
            <button onclick="getPoll()">Получить вопрос</button>
        </p>
        <p>
            <div id="pollSubject"></div>
        </p>
        <p>Ваш выбор: Да: <input type="radio" name="voted" id="yes">
                  Нет: <input type="radio" name="voted" id="no">
        </p>
        <p>Проголосовать: 
            <button onclick="submitVote()">Отправить голос</button>
        </p>
        </p>
    </div>
    <script>
```

```
if (typeof web3 !== 'undefined') {
    web3 = new Web3(web3.currentProvider);
} else {
    web3 = new Web3(new Web3.providers.HttpProvider('http://127.0.0.1:8507'));
}

function getPoll() {
    var pollingContract = new web3.eth.Contract([{
            "constant": true,
            "inputs": [{
                "name": "",
                "type": "address"
            }],
            "name": "votes",
            "outputs": [{
                "name": "",
                "type": "uint256"
            }],
            "payable": false,
            "stateMutability": "view",
            "type": "function"
        },
        {
            "constant": true,
            "inputs": [],
            "name": "getPoll",
            "outputs": [{
                "name": "",
                "type": "string"
            }],
            "payable": false,
            "stateMutability": "view",
            "type": "function"
        },
        {
            "anonymous": false,
            "inputs": [{
                    "indexed": false,
                    "name": "_voter",
                    "type": "address"
                },
                {
                    "indexed": false,
                    "name": "_value",
                    "type": "uint256"
                }
            ],
```

```
            "name": "Voted",
            "type": "event"
        },
        {
            "constant": false,
            "inputs": [{
                "name": "selection",
                "type": "uint256"
            }],
            "name": "vote",
            "outputs": [],
            "payable": false,
            "stateMutability": "nonpayable",
            "type": "function"
        }
    ], '0x59E7161646C3436DFdF5eEE617B4A172974B481e');

    pollingContract.methods.getPoll().call().then(function (value) {
        document.getElementById('pollSubject').textContent = value;
    });
};

function submitVote() {
    var value = 0
    var yes = document.getElementById('yes').checked;
    var no = document.getElementById('no').checked;

    if (yes) {
        value = 1
    } else if (no) {
        value = 2
    } else {
        return;
    }

    var pollingContract = new web3.eth.Contract([{
            "constant": true,
            "inputs": [{
                "name": "",
                "type": "address"
            }],
            "name": "votes",
            "outputs": [{
                "name": "",
                "type": "uint256"
            }],
```

```
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [],
        "name": "getPoll",
        "outputs": [{
            "name": "",
            "type": "string"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "anonymous": false,
        "inputs": [{
                "indexed": false,
                "name": "_voter",
                "type": "address"
            },
            {
                "indexed": false,
                "name": "_value",
                "type": "uint256"
            }
        ],
        "name": "Voted",
        "type": "event"
    },
    {
        "constant": false,
        "inputs": [{
            "name": "selection",
            "type": "uint256"
        }],
        "name": "vote",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    }
], '0x59E7161646C3436D8FdF5eEE617B4A172974B481e');
```

```
            pollingContract.methods.vote(value).send({
                from: '0xbaf735f889d603f0ec6b1030c91d9033e60525c3'
            }).then(function (result) {
                console.log(result);
            });
        };
    </script>
</body>
</html>
```

Давайте теперь проанализируем составные части этого HTML-файла:

- в секции `head` (заголовок страницы) мы загрузили скрипт библиотеки web3 из внешнего источника — внешнего сервера или сервера на локальной машине. Эта ссылка работает точно так же, как если бы мы ссылались на любую другую библиотеку JavaScript на страницах нашего веб-сайта (например, JQuery и тому подобные библиотеки);
- в секции `body` (тело страницы) у нас есть элементы управления для отображения темы опроса и переключатели, а также кнопка отправки результата голосования. Внешний вид веб-страницы показан на рис. 6.11.

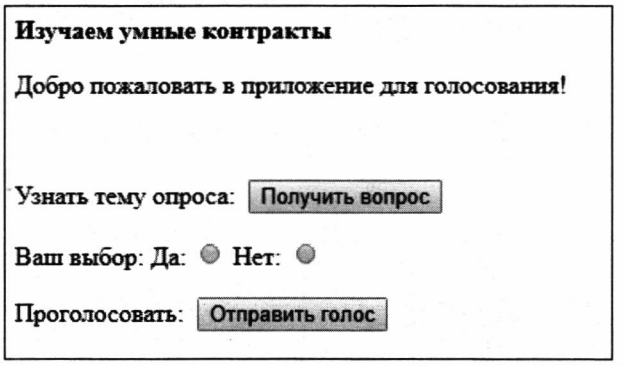



**Рис. 6.11. Веб-приложение для опроса**

Для нас наиболее важен скрипт внутри тела страницы. Именно здесь происходит взаимодействие с умным контрактом. Давайте рассмотрим этот скрипт подробнее:

```
<script>
        if (typeof web3 !== 'undefined') {
            web3 = new Web3(web3.currentProvider);
        } else {
            web3 = new Web3(new Web3.providers.HttpProvider('http://127.0.0.1:8507'));
        }

        function getPoll() {
            var pollingContract = new web3.eth.Contract([{
                    "constant": true,
```

```
    "inputs": [{
        "name": "",
        "type": "address"
    }],
    "name": "votes",
    "outputs": [{
        "name": "",
        "type": "uint256"
    }],
    "payable": false,
    "stateMutability": "view",
    "type": "function"
},
{
    "constant": true,
    "inputs": [],
    "name": "getPoll",
    "outputs": [{
        "name": "",
        "type": "string"
    }],
    "payable": false,
    "stateMutability": "view",
    "type": "function"
},
{
    "anonymous": false,
    "inputs": [{
            "indexed": false,
            "name": "_voter",
            "type": "address"
        },
        {
            "indexed": false,
            "name": "_value",
            "type": "uint256"
        }
    ],
    "name": "Voted",
    "type": "event"
},
{
    "constant": false,
    "inputs": [{
        "name": "selection",
        "type": "uint256"
    }],
```

```
            "name": "vote",
            "outputs": [],
            "payable": false,
            "stateMutability": "nonpayable",
            "type": "function"
        }
    ], '0x59E7161646C3436DFdF5eEE617B4A172974B481e');

    pollingContract.methods.getPoll().call().then(function (value) {
        document.getElementById('pollSubject').textContent = value;
    });
};

function submitVote() {
    var value = 0
    var yes = document.getElementById('yes').checked;
    var no = document.getElementById('no').checked;

    if (yes) {
        value = 1
    } else if (no) {
        value = 2
    } else {
        return;
    }

    var pollingContract = new web3.eth.Contract([{
            "constant": true,
            "inputs": [{
                "name": "",
                "type": "address"
            }],
            "name": "votes",
            "outputs": [{
                "name": "",
                "type": "uint256"
            }],
            "payable": false,
            "stateMutability": "view",
            "type": "function"
        },
        {
            "constant": true,
            "inputs": [],
            "name": "getPoll",
            "outputs": [{
                "name": "",
```

```
                    "type": "string"
                }],
                "payable": false,
                "stateMutability": "view",
                "type": "function"
            },
            {
                "anonymous": false,
                "inputs": [{
                        "indexed": false,
                        "name": "_voter",
                        "type": "address"
                    },
                    {
                        "indexed": false,
                        "name": "_value",
                        "type": "uint256"
                    }
                ],
                "name": "Voted",
                "type": "event"
            },
            {
                "constant": false,
                "inputs": [{
                    "name": "selection",
                    "type": "uint256"
                }],
                "name": "vote",
                "outputs": [],
                "payable": false,
                "stateMutability": "nonpayable",
                "type": "function"
            }
        ], '0x59E7161646C3436DFdF5eBE617B4A172974B481e');

        pollingContract.methods.vote(value).send({
            from: '0xbaf735f889d603f0ec6b1030c91d9033e60525c3'
        }).then(function (result) {
            console.log(result);
        });
    };
</script>
```

В этом скрипте сначала мы определяем объект web3 в качестве HTTP-провайдера локального узла Ethereum (если он еще не определен).

Далее следуют две функции JavaScript: одна — для получения значения строки `pollSubject` из умного контракта, а другая — для вызова функции `vote` умного контракта.

Вызов функций умного контракта происходит точно так же, как мы это делали в предыдущей главе, используя субмодуль `web3.eth.Contract` библиотеки web3.

Обратите внимание, что в первой функции (`getPoll`) мы обращаемся к методу `call` экземпляра умного контракта, а во второй функции (`submitVote`) — к методу `send` экземпляра умного контракта. Это основная разница между двумя функциями.

Используя вызов функции контракта `getPoll`, мы получаем возвращаемое значение без отправки какой-либо транзакции в сеть. Затем мы показываем это значение в пользовательском интерфейсе, назначая его как текст элемента интерфейса (элемент с индексом `pollSubject` в теле веб-страницы).

Затем, используя метод `send`, отправляем транзакцию для выполнения функции голосования в сети, поэтому мы также должны указать учетную запись, которая будет использоваться для вызова функции умного контракта.

Далее приведен ответ функции `submitVote`, который является квитанцией транзакции:

```
{
        blockHash: '0x04a02dd56c037569eb6abe25e003a65d3366407134c90a056f64b62c2d23eb84',
        blockNumber: 4257,
        contractAddress: null,
        cumulativeGasUsed: 43463,
        from: '0xbaf735f889d603f0ec6b1030c91d9033e60525c3',
        gasUsed: 43463,
        logsBloom: '0x00000000000000000000000000000080000000000000040000000000000000000020000000000
        00000000000000000000000000000000000000000000000000000000000000000000000000000002000000000
        02000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        00000000000000000000000000000000000000000000000000000000000000000000000000000020000
        00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        00000000000000000000000000000000000000000000000000000000000000000000000000000000',
        root: '0x58bc4ee0a3025ca3f303df9bb243d052a12302651963730c52c88aafe92ebeee',
        to: '0x59e7161646c3436dfdf5ebe617b4a172974b481e',
        transactionHash: '0x434aa9c0037af3367a0d3d92985781c50774241ace1d382a8723985efcea73b3',
        transactionIndex: 0,
        events: {
            Voted: {
                address: '0x59E7161646C3436DFdF5eBE617B4A172974B481e',
                blockNumber: 4257,
                transactionHash: '0x434aa9c0037af3367a0d3d92985781c50774241ace1d382a8723985
                                                                                efcea73b3',
                transactionIndex: 0,
                blockHash: '0x04a02dd56c037569eb6abe25e003a65d3366407134c90a056f64b62c2d23eb84',
                logIndex: 0,
                removed: false,
```

```
            id: 'log_980a1744',
            returnValues: [Result],
            event: 'Voted',
            signature: '0x4d99b957a2bc29a30ebd96a7be8e68fe50a3c701db28a91436490b7d53870ca4',
            raw: [Object]
        }
    }
}
```

Если мы внимательно рассмотрим эту квитанцию, то увидим, что в ней есть раздел событий, и он показывает запуск события `Voted`, которое мы создали в нашем умном контракте:

```
events: {
        Voted: {
            address: '0x59E7161646C3436DFdF5eBE617B4A172974B481e',
            blockNumber: 4257,
            transactionHash: '0x434aa9c0037af3367a0d3d92985781c50774241ace1d382a8723985
                                                                                efcea73b3',
            transactionIndex: 0,
            blockHash: '0x04a02dd56c037569eb6abe25e003a65d3366407134c90a056f64b62c2d23eb84',
            logIndex: 0,
            removed: false,
            id: 'log_980a1744',
            returnValues: [Result],
            event: 'Voted',
            signature: '0x4d99b957a2bc29a30ebd96a7be8e68fe50a3c701db28a91436490b7d53870ca4',
            raw: [Object]
        }
    }
```

* * *

Итак, мы подошли к концу нашего упражнения по созданию децентрализованного приложения. В предыдущих разделах этой главы мы разработали комплексное децентрализованное приложение на блокчейне Ethereum, а также развернули частную сеть Ethereum.

Приложение можно использовать и в публичной сети Ethereum. Избиратель должен запустить собственный узел, и затем он может голосовать, используя свою учетную запись Ethereum в публичной (основной) сети.

Реализацию умного контракта можно улучшить с помощью различных проверок и правил.

Это упражнение по программированию дает нам лишь общее видение того, как принято подходить к разработке децентрализованных приложений. Упражнение следует рассматривать как отправную точку для разработки приложений Ethereum, и читателю предлагается самостоятельно изучить более сложные примеры программирования и сценарии поведения приложений.

## 6.6. Заключение

В этой главе мы выполнили упражнение по разработке децентрализованного приложения на основе блокчейна Ethereum. Мы также узнали, как настроить частную сеть Ethereum и как взаимодействовать с ней через клиентскую часть приложения.

## 6.7. Рекомендуемые источники

- Документация web3.js: **http://web3js.readthedocs.io/en/1.0/index.html**.
- Документация Solidity: **https://solidity.readthedocs.org/**.
- Учебник «Ethereum Private Networking»: **https://github.com/ethereumproject/go-ethereum/wiki/Private-Networking-Tutorial**.

ПРИЛОЖЕНИЕ

# Описание электронного архива

К английскому изданию книги прилагаются только файлы JavaScript для *глав 5* и *6*. Свежую версию этих файлов можно скачать по адресу: **https://github.com/Apress/beginning-blockchain**.

Для читателей русского издания мы подготовили дополнительные файлы, которые содержат примеры кода из остальных глав книги, а также снабдили переводом комментарии к исходному коду скриптов для *глав 5* и *6*.

В табл. П.1 приведен полный перечень файлов, которые содержатся в электронном архиве, сопровождающем русское издание. Этот архив можно скачать с FTP-сервера издательства «БХВ-Петербург» по интернет-адресу: **ftp://ftp.bhv.ru/9785977540520.zip**. Ссылка на него доступна и со страницы книги на сайте **http://www.bhv.ru/**.

***Таблица П.1.** Перечень файлов, содержащихся в электронном архиве*

| Файл | Описание | Язык |
|---|---|---|
| genesis.json | Файл конфигурации Ethereum для *разд. 6.2.4* | JSON |
| Listing2-1.py | Примеры использования различных хэш-функций | Python |
| Listing2-2.py | Примеры кода для криптографии с открытым ключом | Python |
| Listing2-3.c | Реализация алгоритма Диффи — Хеллмана | C |
| Listing2-4.py | Примеры кода для работы с деревом Меркла | Python |
| Listing4-1.sol | Пример расчета ожидаемой стоимости транзакции | Solidity |
| Listing6-1.sol | Пример умного контракта для проведения опросов | Solidity |
| Listing6-2.html | Код клиентской веб-страницы приложения | HTML |
| ch5 | Каталог с примерами кода для *главы 5* | |
| ch5/bitcoin.js | Полный код примера отправки транзакции Bitcoin | JavaScript |
| ch5/ethereum.js | Полный код примера отправки транзакции Ethereum | JavaScript |
| ch6 | Каталог с примерами кода для *главы 6* | |
| ch6/ethereum.js | Полный код примера размещения контракта и вызова функций | JavaScript |

# Предметный указатель

## У



## Ф



## Х



## Ц



## Ч



## Ш



## Э

www.bhv.ru

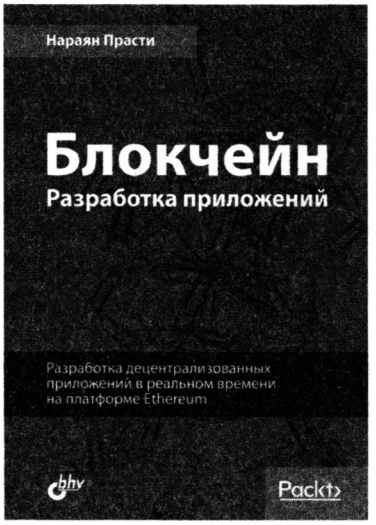
**Отдел оптовых поставок**

E-mail: opt@bhv.spb.su

# Прасти Н.

# Блокчейн. Разработка приложений

## Разработка децентрализованных приложений в реальном времени на платформе Ethereum




Блокчейн — это децентрализованный регистр, который хранит постоянно пополняемый набор записей, защищенных от подделки и правки. Каждый пользователь может подключаться к сети, отправлять в сеть новые транзакции, проверять транзакции и создавать новые блоки без каких-либо разрешений.

Эта книга расскажет о том, как устроен блокчейн, как он обеспечивает целостность данных и как создавать действующие блокчейн-проекты, используя платформу Ethereum. Вы познакомитесь с полезными прикладными приложениями, научитесь писать смарт-контракты, которые работают строго заданным образом без малейшей возможности мошенничества, цензуры и вмешательства третьей стороны, узнаете, как создавать комплексные приложения для блокчейна.

В книге даны основные понятия в области криптографии и криптовалют, безопасности в сети Ethereum, смарт-контрактов, языка Solidity и многого другого. Также вы узнаете о веб-сокетах и освоите интерфейсы различных сервисов Ethereum.

Блокчейн является ключевой инновацией биткойна и обеспечивает ведение открытого регистра транзакций.

**Главные темы книги:**

- обзор основ технологии блокчейна;
- прикладная реализация технологии блокчейна и ее возможностей;
- создание децентрализованных приложений на основе Solidity и Web3.js;
- инструмент разработки Geth и его команды;
- основы криптографии;
- разработка сервиса кошелька для платформы Ethereum;
- понятие корпоративного блокчейна.

**Нараян Прасти** (Narayan Prusty), разработчик приложений полного цикла на технологиях блокчейн и JavaScript и децентрализованных приложений на основе Ethereum, Bitcoin, Hyper-ledger, IPFS, Ripple и других протоколов. Автор книг по ECMAScript 6 и JavaScript.

Нараян стремится делать многие вещи проще, быстрее и дешевле при помощи технологии блокчейн. Он также ищет возможности предотвратить коррупцию, мошенничество и добивается открытости во всем мире при помощи блокчейн-приложений.

www.bhv.ru

**Грас Дж.**

# Data Science. Наука о данных с нуля

**Отдел оптовых поставок**

E-mail: opt@bhv.spb.su

«Джоэл проведет для вас экскурсию по науке о данных. В результате вы перейдете от простого любопытства к глубокому пониманию насущных алгоритмов, которые должен знать любой аналитик данных.»

*Роит Шивапрасад*

*Специалист компании Amazon в области Data Science с 2014 г.*


Книга позволяет освоить науку о данных, начав «с чистого листа».

Она написана так, что способствует погружению в Data Science аналитика, фактически не обладающего глубокими знаниями в этой прикладной дисциплине.

При этом вы убедитесь, что описанные в книге программные библиотеки, платформы, модули и пакеты инструментов, предназначенные для работы в области науки о данных, великолепно справляются с задачами анализа данных.

А если у вас есть способности к математике и навыки программирования, то Джоэл Грас поможет вам почувствовать себя комфортно с математическим и статистическим аппаратом, лежащим в основе науки о данных, а также с приемами алгоритмизации, которые потребуются для работы в этой области.

В сегодняшнем хаотическом потоке данных скрыты ответы на многие волнующие человека вопросы. Книга познакомит с методологией, которая позволит правильно сформулировать эти вопросы и найти на них ответы.

**Вместе с Джоэлом Грас и его книгой**

- пройдите интенсивный курс языка Python;
- изучите элементы линейной алгебры, математической статистики, теории вероятностей и их применение в науке о данных;
- займитесь сбором, очисткой, нормализацией и управлением данными;
- окунитесь в основы машинного обучения;
- познакомьтесь с различными математическими моделями и их реализацией по методу *k*-ближайших соседей, наивной байесовской классификации, линейной и логистической регрессии, а также моделями на основе деревьев принятия решений, нейронных сетей и кластеризации;
- освойте работу с рекомендательными системами, приемы обработки естественного языка, методы анализа социальных сетей, технологии MapReduce и баз данных.

**Джоэл Грас** работает инженером-программистом в компании Google. До этого занимался аналитической работой в нескольких стартапах. Активно участвует в неформальных мероприятиях специалистов в области науки о данных. Всегда доступен в Tweeter по хештегу **@joelgrus**.

Для разработчиков от разработчиков

# Блокчейн

## Руководство для начинающих разработчиков

Изучите устройство блокчейна и основы криптографии с нуля. Научитесь разрабатывать децентрализованные приложения на основе простых примеров кода.

Эта книга предназначена для изучения фундаментальных основ блокчейна с технической точки зрения. Исследуя особенности устройства и работы разных типов блокчейна, вы научитесь находить наилучшие решения практических задач. В книге подробно раскрыты технические аспекты технологии блокчейна, криптографии, криптовалют и механизма распределенного консенсуса. Вы узнаете, как объединить эти компоненты в одну систему для создания прикладных приложений в эпоху блокчейна.

В книге рассказано:

- Как работают криптовалюты
- Как устроен блокчейн
- Как криптография делает блокчейн безопасным
- Какие типы блокчейна существуют и для чего они подходят
- Как устроены Bitcoin и Ethereum
- Как запрограммировать блокчейн для разных вариантов использования
- Что может блокчейн сегодня и чего нам ждать от него в будущем

bhv®

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-4052-0

Apress®