Waves Book

Введение

Репозиторий, который у вас сейчас перед глазами, является попыткой помочь разработчикам, которые хотят делать децентрализованные приложения на базе блокчейна Waves, лучше понимать протокол и знать его особенности.

Данная книга может вам помочь максимально быстро сделать свое первое приложение на блокчейне Waves не допуская никаких фатальных ошибок. В этой книге акцент делается на то, что действительно важно для создания продуктов на базе Waves.

Эта книга акцентирована на примеры, разбор реальных кейсов, конкретные рецепты, которые вы можете применять в своих приложениях. Но, как и во многих аспектах жизни, переход к реальным задачам требует фундаментальных знаний о протоколе. Поэтому в первых трех главах рассказывается об особенностях работы блокчейна Waves, чем он отличается от других блокчейнов, наряду с разбором что и как может влиять на архитектуру вашего приложения и какие могут быть ограничения.

Книга сфокусирована на особенностях работы Waves, но не на базовых понятиях вроде "что такое блок?" и "как работает алгоритм консенсуса", так как материалов о том, как работает блокчейн, огромное количество, в то время как найти особенности отдельных протоколов, чтобы делать на них бизнес, достаточно проблематично.

Важно понимать, что книга не является "полным справочником" по Waves, так как делает акцент на прикладные аспекты протокола. Она не пытается конкурировать с документацией протокола, которая уже есть достаточно давно и покрывает многие аспекты работы с Waves.

1.1 История создания Waves

Что такое Waves?

Waves это Proof-of-Stake permissionless блокчейн платформа общего назначения, создаваемая с 2016 года и призванная помочь найти массовое применение технологии блокчейн. Блокчейн платформа Waves является одной из наиболее зрелых (как в виду возраста, так и в виду большого числа проектов) и легких для начинающих разработчиков, которые хотят использовать преимущества блокчейна с максимальной пользой. В рамках этой книги мы разберем основные технические особенности, поговорим о преимуществах и разберем много реального кода, но прежде чем заняться этим, немного поговорим про историю Waves, чтобы лучше понимать истоки тех или иных особенностей.

Начало проекта

Waves как блокчейн начался в 2016 году, когда основатель платформы Александр Иванов инициировал сбор средств в рамках ICO. Платформа с самого начала позиционировала себя как блокчейн общего назначения, без фокуса на отдельных специфических сферах применения. Основной задачей, которую собиралась решать (и во многом успешно решала) платформа является пропускная способность. На

начало 2016 года существовало очень мало блокчейнов, которые могли обрабатывать сотни транзакций в секунду. Фактически, на рынке полноценно работали только Bitcoin (и его форки вроде Litecoin) с 7 транзакциями в секунду и Ethereum с 15 транзакций/сек. Так что проблема пропускной способности блокчейна была крайне актуальной.

Основные вехи развития проекта

Успешный сбор средств на ICO был только началом проекта, дальше предстояло реализовать все обещанное максимально хорошо. Изначально была выбрана технологическая база фреймворка Scorex. От самого фреймворка в кодовой базе проекта сейчас почти ничего не осталось, но Scorex написан на языке Scala, что и определило надолго технологический стэк разработки протокола. До недавнего времени реализация ноды (то есть протокола) на Scala была единственной, только относительно недавно появилась так же реализация на Go. Стоит заметить, что на момент написания этих строк версия на Go по возможностям отставала от версии на Scala, об этом мы поговорим в следующих разделах. Запуск основной сети (в дальнейшем будем называть mainnet) Waves произошел в декабре 2016 года. Проект с самого начала имел особенности: легкий выпуск своих токенов/ассетов (с помощью отправки одной транзакции) и PoS с моделью лизинга (стейкинга). Данные особенности мы разберем в дальнейших разделах.

Другими примечательными вехами в истории протокола можно назвать следующие даты:

- Выпуск DEX (гибридной биржи) в 2017 году. Логичным продолжением нативной поддержки выпуска токенов с помощью транзакции (не контракта как в Ethereum), явился выпуск матчера, который обеспечивал работу децентрализованной биржи.
- В том же году был реализован протокол Waves NG, который позволил достичь хорошей пропускной способности. Waves NG был реализован на основе идей, изложенных в статье. Данные предложения были нацелены на реализацию в Bitcoin, но никогда не были там реализованы, зато были реализованы в Waves.
- Летом 2018 года был выпущен язык для смарт-контрактов Ride для базовой логики аккаунтов, а в января 2019 года появилась возможность писать контракты для токенов
- Летом 2019 года до мейннета добрался Ride для полноценных децентрализованных приложений (Ride4dApps)
- Осенью 2019 года появилась первая в своем роде (среди основных блокчейнов проектов) монетарная политика

Влияение истории Waves на протокол

История создания Waves достаточно сильно повлияла на технические детали проекта. Например, фреймворк Scorex корнями уходит в проект Nxt, другой Proof-of-Stake блокчейн с нативными токенами. Обе особенности были унаследованы Waves именно из Nxt. Легкость создания токенов и скорость работы протокола в дальнейшем сделали проект второй по популярности платформой для выпуска токенов и запуска ICO (сразу после Ethereum). Большое количество проектов на Waves использовали блокчейн именно для работы с токенами (выпуск, небольшая стоимость транзакций).

1.2 Подходы к разработке протокола Waves

Разработчики протокола Waves всегда руководствовались некоторыми базовыми принципами, которые сильно влияют на дальнейшее развитие протокола. Понимание данных принципов и мотивации за ними поможет легче следить за дальнейшим развитием проекта, поэтому перечислю данные особенности.

Блокчейн для людей

Мантрой Waves долгое время было "Blockchain for the people". Она полностью отражала и отражает то, что делает команда. Главное, чего хочет достичь платформа - **популяризировать технологию блокчейн для масс**. В данный момент блокчейн является технологией для очень небольшой группы людей, которые понимают, что это за технология и как ее правильно использовать. Waves хочет изменить такое положение вещей и сделать так, чтобы технология приносила максимальную пользу всем.

Многие люди думают, что технология блокчейн крайне сложная и наукоемкая (во многом так и есть), Waves же пытается скрывать всю сложность за неким слоем абсракции. Блокчейн - это не самая удобная база данных, у которой есть несколько важных свойств: децентрализация, неизменяемость и открытость. Данные особенности не являются ценностями сами по себе, а только в случае правильного применения разработчиками конкретных приложений. Цель Waves состоит в предоставлении таких инструментов разработчикам, которые позволят им быстрее,проще, без излишнего погружения в сложные технические давать ценность конечным пользователям.

Можно сказать, что принцип ориентации на реальное применение разбивается на несколько шагов:

1. Платформа предоставляет разработчикам приложений легкий инструмент для использования особенностей блокчейна
2. Разработчики приложений делают продукты, которые решают проблемы пользователей и правильно используют блокчейн
3. Пользователи получают получают преимущества блокчейна. При этом не обязательно, чтобы пользователи знали что-то про блокчейн. Главное, чтобы решалась их проблема.

Ориентация на практическую применимость

Всегда во время разработки нового функционала или продукта во главе угла ставится **практическая применимость** и какое количество людей **потенциально** смогут решить свои проблемы с помощью этого. Разработчики протокола пытаются не делать "космические корабли", решать "сферические проблемы в вакууме" или заниматься оверинжинирингом, выбирая применимость здесь и сейчас. Лучше ведь иметь работающее сейчас, чем идеальное через 10 лет? Конечно, данный принцип не должен вступать в противоречие с безопасностью сети.

Открытость разработки

Протокол Waves является полностью открытым и процесс разработки максимально децентрализован. Все исходные коды есть на github. Кроме непосредственно исходного кода там же обсуждаются пути развития протокола, различные проблемы и варианты их решения. Обновления протокола, связанные с изменением консенсуса всегда проходят процедуру обсуждения с помощью Waves Enhancement Proposals. Но обсуждения это только первый этап, ведь все обновления консенсуса должны еще проходить через процедуру активации с голосованием, о чем мы поговорим с следующем разделе. Теперь вы знаете, что делать, если захотите что-то изменить в протоколе.

1.3 Отличительные особенности блокчейна Waves

Если у вас уже есть опыт работы с другими блокчейнами, вам может быть интересно, чем же отличается Waves от условного Ethereum и почему он другой. Давайте быстро пройдемся по отличиям, которые будут детально рассматриваться в следующих разделах.

Работа с токенами/ассетами

Одной из особенностей работы с Waves с первого дня была простота выпуска токенов. Для этого достаточно отправить транзакцию или заполнить форму из 5 полей) в любом UI клиенте. Выпущенный токен автоматически становится доступен для переводов, торговки на DEX, использования в dApp и сжигания.

В отличие от Ethereum, в Waves токены не являются смарт-контрактами, а являются "гражданами первого сорта", то есть являются отдельной полноценной сущностью. У этого есть как преимущества, так и недостатки, о которых мы поговорим в разделе 4 "Токены".

Транзакции

Другая отличительная особенность Waves заключается в наличии большого количества типов транзакций. Например, в Ethereum есть смарт-контракты, которые могут являться чем угодно, в зависимости от их реализации. ERC-20, описывающий токен, это просто описание интерфейса смарт контракта, какие методы он должен иметь. В Waves это не так, т.к. подразумеватся, что лучше иметь легковесные/узкие специфичные вещи, чем абстрактные вещи "обо всем и ни о чем". Специфичность примитивов упрощает во многих местах разработку, но это иногда является менее гибким решением.

Ниже представлен список актуальных транзакций на момент написания этих строк:

Fair PoS

В Waves используется алгоритм Proof-of-Stake для определения права на генерацию блока. Блоки генерируются в среднем каждую минуту, а вероятность генерации блока нодой зависит от 3 параметров:

- Генерирующего баланса ноды, то есть баланса самой ноды + количества токенов, которые сдали ей в лизинг.
- Текущего времени и рандома (великий рандом есть всегда в жизни!)
- Генерящего баланса сети, ведь не все токены в сети участвуют в генерации блоков, они могут быть в ордерах на биржах или лежать на холодных кошельках.

Чтобы начать генерировать блоки достаточно иметь 1000 Waves генерирующего баланса (свои + полученные в лизинг). Зачем вам генерировать блоки? За каждый блок нода получает на свой баланс комиссии из транзакций в этом блоке и вознаграждение "из воздуха". Оба этих момента не такие простые, так что рассмотрим их чуть позже.

Очень частый вопрос, который встречается - сколько блоков я буду генерировать в месяц с балансом N. Точное числе предсказать невозможно, так как зависит от случайностей и изменений в сети, но примерно предсказать можн. Чтобы сделать это надо знать текущие параметры сети:

1. *Сколько токенов Waves участвуют в генерации блоков?* Это так называемый генерирующий баланс сети в целом. Для легкости рассчета скажем, что 50 млн токенов участвуют в генерации блоков.
2. *Какой баланс нашей ноды?* То есть сколько у нас своих токенов на аккаунте ноды и сколько нам сдали в лизинг. В нашем случае возьмем генерирующий баланс равный 10 000 Waves.
3. Среднее время блока составляет 1 минуту, то есть в сети генерируется примерно 1440 блоков в день или 43200 блоков в месяц.

Для вычисления примерного количества блоков, которые мы сгенерируем, делим количество блоков за период на генерирующий баланс сети и умножаем на наш баланс:

$ForgedBlocks = BlocksCountInPeriod / NetworkGenBalance * NodeGenBalance$

Сделав нехитрые вычисления получаем:

```
43200 / 50000000 * 10000 = 8.64
```

То есть, в среднем нода будет генерировать 8-9 блоков в месяц, если будет работать стабильно и параметры сети не изменятся, но, такого, конечно, не бывает, ведь в сети постоянно делается большое количество транзакций.

Leasing

В Waves есть механизм стейкинга, который называется leasing. Любой владелец токена Waves может отправить токены в лизинг любой ноде Waves, чтобы та производила блоки "от имени этих токенов". Лизингодатель передает право на генерацию блоков от имени его токенов лизингополучателю (владельцу ноды). Обычно это делается, когда пользователь не хочет или не может заниматься разворачиваем своей ноды и ее поддержкой. Обычно владельцы лизнговых пулов выплачивают бОльшую часть заработанного за счет лизинга средств лизингодателям. Отправить в лизинг средства можно моментально, но в генерируюющем балансе ноды они начнут учитываться только через 1000 блоков.

Забрать токены из лизинга можно моментально.

Community driven monetary policy

Первое время в Waves была ограниченная эмиссия в 100 млн токенов, которые были выпущены сразу на момент запуска мейннета, но с осени 2019 года в комьюнити решили, что для дальнейшего роста экосистемы лучше будет при наличии эмиссии токенов. То есть в каждом новом блоке появляются новые токены Waves. Какое именно количество токенов определяется сообществом, которое голосует за размер вознаграждения каждые 100 тысяч блоков. На момент написания этих строк вознаграждение за блок составляло 6 Waves. При этом гарантируется, что каждый период голосования размер вознаграждения не может изменяться больше, чем на 0.5 Waves.

Sponsorship

Функция спонсирования токена является способом снижения входных барьеров для пользователей. Суть в том, что аккаунт, выпустивший токен, может спонсировать транзакции с этим токеном. Представим, есть токен A, его выпустил Cooper и включил спонсирование. Например, Alice заработала 100 токенов A и хочет 10 из них отпраивть Bob. Мы то с вами знаем, что для каждой транзакции в

блокчейне необхоодимо платить комиссию, в сети Waves майнеры принимают только Waves в виде комиссии, а у Alice нет Waves. Придется идти покупать Waves как-то?

Нет. Спонсорство токена позволяет владельцу токена сказать, что он готов взять на себя комиссии за операции с этим токеном (токеном A в нашем случае). Владельцы токена A при отправке транзакции будут платить этот же токен в виде комиссии. В нашем примере, Alice сможет указать в своей транзакции, что получатель Bob, количество для отправления - 10 токенов A, комиссия - 5 токенов A. В итоге с ее аккаунта спишется 15 токенов, 10 получит Bob, 5 получит Cooper, как выпустивший и спонсирующий токен, а майнер получит Waves с аккаунта Cooper'a.

Почему Alice заплатит 5 токенов и сколько получит майнер? Поговорим об этом в следующих главах. Главное, что сейчас необходимо запомнить - **в Waves существуют способы сделать транзакции не имея токенов Waves у себя на балансе.**

Ride и смарт контракты

Waves является блокчейном общего назначения, не специализирующемся на чем-то одном, поэтому появление смарт-контрактов стало логичным продолжением развития платформы. Про смарт-контракты в Waves мы поговорим в разделе 6 "Ride". Сейчас стоит отметить, что контракты пишутся на языке Ride, который был придуман специально для смарт-контрактов и является не Тьюринг-полным. В языке нет циклов, но зато нет газа, не бывает failed транзакций в блокчейн, стоимость транзакции всегда известна заранее. Заинтриговал? Это ведь только начало, мы поговорим про модель исполнения Ride и синтаксис языка позже.

Waves NG

Чуть раньше я уже затрагивал тему, связанную с Waves NG и упоминал, что она позволяет транзакциям быстрее попадать в блоки и работать блокчейну так быстро, что платформа в состоянии обрабатывать сотни транзакций в секунду в основной сети. А там, на минуточку, больше 400 нод, распределенных по всему миру, на совершенно разном железе и пропускной способностью.


Waves nodes map

DEX

Легкий и быстрый выпуск токенов наряду с "классовым равенством" (помните, что токены являются гражданами первого сорта?) позволяет сделать торговлю токенами простой. Нода Waves (речь про версию на Scala) поддерживает возможность создания расширений, одним из таких расширений является матчер. Матчер принимает ордера на покупку и продажу токенов и хранит их (централизованно). Например, Alice хочет продать токен wBTC и купить Waves, а Bob наоборот. Они формируют ордера (криптографически подписанные примитивы) и отправляют их в матчер, который определяет, что эти ордера выставлены в одной паре и их можно сматчить по определенной цене. В результате матчер формирует Exchange транзакцию, которая содержит 2 ордера (один от Alice, другой от Bob) и отправляет в блокчейн. При этом, матчер забирает себе комиссию, нода, которая смайнит блок с Exchange транзаций, тоже получает комиссию.


How does DEX work

2.1 Нода Waves и как она работает, ее конфигурация

Непосредственное техническое знакомство с платформой Waves я бы рекомендовал начать с установки и настройки ноды. Не обязательно для основной сети, можно для testnet (полная копия по техническим возможностям) или stagenet (экспериментальная сеть). Почему я рекомендую начать с установки ноды? Во-первых, я сам свое знакомство с блокчейном Waves начинал именно с этого, а во-вторых, установка и настройка заставляют разобраться в том, какие есть настройки у ноды, какие параметры есть у сети.

Из чего состоит нода Waves

Как и почти во всех блокчейнах, нода - программный продукт, который отвечает за прием транзакций, создание новых блоков, синхронизацию данных между разными узлами и достижение консенсуса между ними. Каждый участник сети запускает свою копию ноды и синхронизируется с остальными. Про правила консенсуса мы поговорим чуть позже, сейчас давайте разберемся, что из себя представляет нода с точки зрения ПО.

По большому счету, нода это исполняемый файл (`jar` файл для Scala версии и бинарный для Go), который в момент запуска читает конфигурационный файл, чтобы на основе этих параметров начать общаться с другими узлами в сети по протоколу поверх TCP. Получаемые и генерируемые данные нода складывает в LevelDB (key-value хранилище). По большому то счету все, но дьявол кроется в деталях. По мере углубления в особенности работы вы поймете, что это далеко не так просто, как может показаться. А пока, давайте поговорим о том, с чего нода начинает свою работу в момент запуска - конфигурационного файла.

Установка ноды

В данной книге мы не будем разбирать процесс установки ноды, так как это много раз описано уже в разных источниках (документация, видео на youtube, посты на форуме). Нам никакой книги не хватит, если мы захотим окунуться в эту тему, потому что существует много способов запуска ноды:

1. Запуск `jar` файла
2. Запуск Docker контейнера
3. Установка и запуск из `.deb` файла
4. Установка из `apt-` репозитория

Лично я предпочитаю запуск Docker контейнера, так как это упрощает и поддержку, и настройку, и конфигурирование. Другой причиной моей любви к Docker может быть то, что я делал Docker образ, размещенный в Docker Hub и отлично знаю как там что работает. Хотя вряд ли, это просто удобнее! 😃

Конфигурация ноды

Конфигурационный файл ноды Waves описан в формате HOCON (это как JSON, только с комментариями, возможностью композиции нескольких файлов и, что не менее важно, с меньшим количеством кавычек). Полный файл с конфигурацией выглядит громоздко, но я все-таки приведу его здесь, версия на момент написания этих строк (файл постоянно меняется, но актульную версию можно найти в репозитории Waves на Github).

Файл содержит большое количество комментариев, поясняющих каждый параметр, поэтому подробно разбирать все параметры мы не будем. Поговорим только об основных моментах. В конфигурации содержатся следующие разделы:

- waves
- kamon
- metrics
- akka

Последние 3 раздела являются полностью служебными, которые отвечают за параметры логгирования, отправку метрик и фреймворка akka. Нас интересует только первый раздел, который касается непосредственно протокола и содержит на первом уровне следующие подразделы:

- **db** - параметры для работы с LevelDB и настройки какие данные сохранять. Например, результаты некоторых транзакций можно сжимать (экономить до 10 ГБ), но это вредит удобству работу с API. Поэтому будьте осторожны с тем, что включать, а что выключать и экономить место на диске.
- **network** - параметры общения с другими нодами в сети. В этом разделе много важных параметров, которые мы разберем чуть ниже.
- **wallet** - параметры файла для сохранения ключей. Каждая нода, которая хочет генерировать блоки (чтобы получать за них вознаграждение), должна подписывать свои блоки. Для этого у ноды должен быть доступ к ключу аккаунта. В этой секции задается приватный ключ (а если быть точнее, то seed фраза в кодировке base58), пароль для того, чтобы эту фразу зашифровать, и путь, по которому хранить файл с этим зашифрованным ключом.
- **blockchain** - параметры блокчейна, в котором будет работать нода. В данном разделе есть настройка `type`, которая позволяет задать одну из заранее определенных типов блокчейнов (stagenet, testnet или mainnet). При указании значения `custom` можно менять все параметры блокчейна, в том числе первоначальное количество токенов, их распределение, байт сети (уникальный идентификатор каждой сети), поддерживаемые фичи и т.д.
- **miner** - параметры генерации новых блоков. По умолчанию генерация включена (надо понимать, что это будет работать только при наличии генерирующего баланса больше 1000 Waves на аккаунте), но ее можно отключить с помощью параметра `enable`. Может пригодиться, если, например, нужна нода, которая будет только валидировать блоки. Другой полезный параметр - `quorum`, который определяет сколько соединений с другими нодами необходимо, чтобы нода пыталась генерировать блоки. Если задать данный параметр, равный 0, то можно запустить блокчейн, состоящий из 1 узла. Зачем вам такой блокчейн, если этот один узел может переписывать всю историю и делать все, что захочет? Для тестирования! Идеальная песочница для игр с блокчейном.
- **rest-api** - в ноде есть встроенный REST API, который по умолчанию отключен, но после включения (за это отвечает параметр `enable`) позволяет делать HTTP запросы для получения данных из блокчейна или записи в нее новых транзакций. В данном API есть как открытые методы, так и те, которые требуют API ключ, который задается в этой же секции настроек. Параметр `port` задает на каком порту будет слушать нода входящие HTTP запросы, на этом же порту будет доступен Swagger UI с описанием всех методов API. `api-key-hash` позволяет указать хэш от API-ключа, с которым будут доступны приватные методы. То есть в конфигурационном файле мы указываем не сам ключ, а хэш от него. А какой хэш надо взять? SHA-1? SHA-512? Или, прости господи, MD5?

- **synchronization** - параметры синхронизации в сети, в том числе максимальная длина форка и параметр `max-rollback` задает сколько блоков может быть откачено). Фактически можно

сказать, что время финализации транзакции, после которого точно можно быть уверенным, что транзакция не пропадет из сети, составляет 100 минут (среднее время блока составляет 1 минуту).

- **utx** задает параметры пула неподтвержденных транзакций. Каждая нода настраивает эти параметры в зависимости от объема оперативной памяти, мощности и количества CPU. Параметр `max-size`, задающий максимальное количество транзакций в списке ожидания составляет 100 000 по умолчанию, а `max-bytes-size` имеет значение `52428800`. При достижении любого из этих лимитов, нода перестанет принимать транзакции в свой список ожидания. Про UTX мы поговорим отдельно в разделе транзакций.
- **features** - каждый новый функционал и изменения консенсуса (именно правил консенсуса, не изменения API или внутренностей ноды!) должны проходить через процедуру голосования. Принцип работы голосования мы разберем позже в этом разделе. Сейчас просто скажем, что каждая нода голосует используя массив `supported` в этой части конфигурации. Так же нода может автоматически выключиться, если вся сеть приняла какое-то обновление, которое не поддерживается этой версией, используя флаг `auto-shutdown-on-unsupported-feature`.
- **rewards** позволяет установить размер вознаграждения для майнера блока. Как и в случае с обновлениями проотокола проводится голосование, но голосование за размер вознаграждения работает по другому принципу.
- **extensions** описывает какие расширения вместе с этой нодой необходимо запускать.

В разделе `waves` на первом уровне так же лежат некоторые параметры:

- `directory` - путь к директории, в которой нода будет сохранять все файлы, которые относятся к ней, к том числе файлы LevelDB с данными
- `ntp-server` - сервер синхронизации времени
- `extensions-shutdown-timeout` - это время, которое дается расширенниям, подключенным к ноде, корректно завершиться при выключении самой ноды

В следующим разделах мы будем подробнее разбирать какие параметры влияют на поведение ноды и каким образом.

Особенности конфигурации

Вы уже увидели, что конфигурация ноды осуществляется с помощью HOCON файлов, которые поддерживают возможность композиции. Другими словами, в файле конфигурации можно использовать инструкции `include filename.conf`, который может загружать разные разделы конфигурации из другого файла. Некоторые разделы могут повторяться в разных файлах, поэтому там так же есть механизм разрешения конфликтов (чем ниже подключен файл, тем больший приоритет он имеет). Если у вас есть опыт работы с CSS, то принцип такой же. В некоторых местах документации Waves приводится нотация вроде `waves.network.port`, как нетрудно догадаться, это обозначает параметр в конфигурации вместе с путем.

Безопасность

При конфигурировании ноды имеет значение уделить особое внимание тем параметрам, которые напрямую влияют на безопасность - разделам `wallet` и `rest-api`. Хорошей практикой считается указать в конфигурационном файле base58 представление seed фразы и пароль, запустить ноду, а затем удалить из файла (не перезапуская ноду). Таким образом, запущенная нода будет знать приватный ключ и пароль (это есть в оперативной памяти), но в файле конфигурации ничего не осталось. Если

вдруг кто-то получит доступ к вашей конфигурации или даже `wallet` файлу, он не сможет расшифровать ключ.

API ключ ноды не менее важен, потому что он позволяет отправить с ноды транзакции, подписанные ключами, хранящимися в ноде. В отличие от данных аккаунта, в конфигурации хранится только хэш, поэтому удалять оттуда после запуска смысла нет, но есть смысл использовать максимально сложный ключ и никак не тот, который стоит по умолчанию (в старых версиях был ключ по-умолчанию, в последних такого уже нет).

Оптимальные настройки блокчейна

Очень часто задаваемый вопрос - какие же настройки оптимальные? В первую очередь зависит от того, о какой сети мы говорим - `stagenet`, `testnet`, `mainnet`, `custom`? Например, для `custom` не существует оптимальных, существуют требования к сети. Но надо понимать, что нода Waves не всемогущая, эмпирически показано, что при времени блока меньше 12 секунд (`waves.blockchain.genesis.average-block-delay`), времени микроблока меньше 3 секунд (`waves.miner.micro-block-interval`) и относительно большом количестве узлов в сети (10+), сеть может быстро форкаться.

Важным параметром, который надо настраивать с учетом особенностей окружения, является максимальное количество транзакций в UTX. 100 000 транзаций является оптимальным для ноды, удовлетворяющей минимальным системным требованиям.

Я описал выше только самые основные параметры, многие другие мы будем рассматривать в следующих разделах, по мере погружения в протокол и его особенности. Сейчас же приведу полный файл конфигурации с комментариями:

```
# Waves node settings in HOCON
# HOCON specification:
https://github.com/lightbend/config/blob/master/HOCON.md
waves {
  # Node base directory
  directory = ""

  db {
    directory = ${waves.directory}"/data"
    store-transactions-by-address = true
    store-invoke-script-results = true
    # Limits the size of caches which are used during block validation.
Lower values slightly decrease memory consummption,
    # while higher values might increase node performance. Setting ghis
value to 0 disables caching alltogether.
    max-cache-size = 100000

    max-rollback-depth = 2000
    remember-blocks = 3h
  }

  # NTP server
  ntp-server = "pool.ntp.org"
```

```
  # P2P Network settings
  network {
    # Peers and blacklist storage file
    file = ${waves.directory}"/peers.dat"

    # String with IP address and port to send as external address during
handshake. Could be set automatically if UPnP
    # is enabled.
    #
    # If `declared-address` is set, which is the common scenario for nodes
running in the cloud, the node will just
    # listen to incoming connections on `bind-address:port` and broadcast
its `declared-address` to its peers. UPnP
    # is supposed to be disabled in this scenario.
    #
    # If declared address is not set and UPnP is not enabled, the node
will not listen to incoming connections at all.
    #
    # If declared address is not set and UPnP is enabled, the node will
attempt to connect to an IGD, retrieve its
    # external IP address and configure the gateway to allow traffic
through. If the node succeeds, the IGD's external
    # IP address becomes the node's declared address.
    #
    # In some cases, you may both set `decalred-address` and enable UPnP
(e.g. when IGD can't reliably determine its
    # external IP address). In such cases the node will attempt to
configure an IGD to pass traffic from external port
    # to `bind-address:port`. Please note, however, that this setup is not
recommended.
    # declared-address = "1.2.3.4:6863"

    # Network address
    bind-address = "0.0.0.0"

    # Port number
    port = 6863

    # Node name to send during handshake. Comment this string out to set
random node name.
    # node-name = "default-node-name"

    # Node nonce to send during handshake. Should be different if few
nodes runs on the same external IP address. Comment this out to set random
nonce.
    # nonce = 0

    # List of IP addresses of well known nodes.
    known-peers = ["52.30.47.67:6863", "52.28.66.217:6863",
"52.77.111.219:6863", "52.51.92.182:6863"]

    # How long the information about peer stays in database after the last
communication with it
```

```
    peers-data-residence-time = 1d

    # How long peer stays in blacklist after getting in it
    black-list-residence-time = 15m

    # Breaks a connection if there is no message from the peer during this
timeout
    break-idle-connections-timeout = 5m

    # How many network inbound network connections can be made
    max-inbound-connections = 30

    # Number of outbound network connections
    max-outbound-connections = 30

    # Number of connections from single host
    max-single-host-connections = 3

    # Timeout on network communication with other peers
    connection-timeout = 30s

    # Size of circular buffer to store unverified (not properly
handshaked) peers
    max-unverified-peers = 100

    # If yes the node requests peers and sends known peers
    enable-peers-exchange = yes

    # If yes the node can blacklist others
    enable-blacklisting = yes

    # How often connected peers list should be broadcasted
    peers-broadcast-interval = 2m

    # When accepting connection from remote peer, this node will wait for
handshake for no longer than this value. If
    # remote peer fails to send handshake within this interval, it gets
blacklisted. Likewise, when connecting to a
    # remote peer, this node will wait for handshake response for no
longer than this value. If remote peer does not
    # respond in a timely manner, it gets blacklisted.
    handshake-timeout = 30s

    suspension-residence-time = 1m

    # When a new treansaction comes from the network, we cache it and
doesn't push this transaction again when it comes
    # from another peer.
    # This setting setups a timeout to remove an expired transaction in
the elimination cache.
    received-txs-cache-timeout = 3m

    upnp {
      # Enable UPnP tunnel creation only if you router/gateway supports
```

```
it. Useful if your node is runnin in home
      # network. Completely useless if you node is in cloud.
      enable = no

      # UPnP timeouts
      gateway-timeout = 7s
      discover-timeout = 3s
    }

    # Logs incoming and outgoing messages
    traffic-logger {
      # Codes of transmitted messages to ignore. See
MessageSpec.messageCode
      ignore-tx-messages = [23, 25] # BlockMessageSpec,
TransactionMessageSpec

      # Codes of received messages to ignore. See MessageSpec.messageCode
      ignore-rx-messages = [25] # TransactionMessageSpec
    }
  }

  # Wallet settings
  wallet {
    # Path to wallet file
    file = ${waves.directory}"/wallet/wallet.dat"

    # Password to protect wallet file
    # password = "some string as password"

    # The base seed, not an account one!
    # By default, the node will attempt to generate a new seed. To use a
specific seed, uncomment the following line and
    # specify your base58-encoded seed.
    # seed = "BASE58SEED"
  }

  # Blockchain settings
  blockchain {
    # Blockchain type. Could be TESTNET | MAINNET | CUSTOM. Default value
is TESTNET.
    type = TESTNET

    # 'custom' section present only if CUSTOM blockchain type is set. It's
impossible to overwrite predefined 'testnet' and 'mainnet' configurations.
    #    custom {
    #      # Address feature character. Used to prevent mixing up
addresses from different networks.
    #      address-scheme-character = "C"
    #
    #      # Timestamps/heights of activation/deactivation of different
functions.
    #      functionality {
    #
    #        # Blocks period for feature checking and activation
```

```
    #        feature-check-blocks-period = 10000
    #
    #        # Blocks required to accept feature
    #        blocks-for-feature-activation = 9000
    #
    #        reset-effective-balances-at-height = 0
    #        generation-balance-depth-from-50-to-1000-after-height = 0
    #        block-version-3-after-height = 0
    #        max-transaction-time-back-offset = 120m
    #        max-transaction-time-forward-offset = 90m
    #        pre-activated-features {
    #          1 = 100
    #          2 = 200
    #        }
    #        lease-expiration = 1000000
    #      }
    #      # Block rewards settings
    #      rewards {
    #        term = 100000
    #        initial = 600000000
    #        min-increment = 50000000
    #        voting-interval = 10000
    #      }
    #      # List of genesis transactions
    #      genesis {
    #        # Timestamp of genesis block and transactions in it
    #        timestamp = 1460678400000
    #
    #        # Genesis block signature
    #        signature = "BASE58BLOCKSIGNATURE"
    #
    #        # Initial balance in smallest units
    #        initial-balance = 100000000000000
    #
    #        # Initial base target
    #        initial-base-target =153722867
    #
    #        # Average delay between blocks
    #        average-block-delay = 60s
    #
    #        # List of genesis transactions
    #        transactions = [
    #          {recipient = "BASE58ADDRESS1", amount = 50000000000000},
    #          {recipient = "BASE58ADDRESS2", amount = 50000000000000}
    #        ]
    #      }
    #    }
    }

    # New blocks generator settings
    miner {
      # Enable/disable block generation
      enable = yes
```

```
    # Required number of connections (both incoming and outgoing) to
attempt block generation. Setting this value to 0
    # enables "off-line generation".
    quorum = 1

    # Enable block generation only in the last block is not older the
given period of time
    interval-after-last-block-then-generation-is-allowed = 1d

    # Mining attempts delay, if no quorum available
    no-quorum-mining-delay = 5s

    # Interval between microblocks
    micro-block-interval = 5s

    # Max amount of transactions in key block
    max-transactions-in-key-block = 0

    # Max amount of transactions in micro block
    max-transactions-in-micro-block = 255

    # Miner references the best microblock which is at least this age
    min-micro-block-age = 4s

    # Minimal block generation offset
    minimal-block-generation-offset = 0

    # Max packUnconfirmed time
    max-pack-time = ${waves.miner.micro-block-interval}
  }

  # Node's REST API settings
  rest-api {
    # Enable/disable REST API
    enable = yes

    # Network address to bind to
    bind-address = "127.0.0.1"

    # Port to listen to REST API requests
    port = 6869

    # Hash of API key string
    api-key-hash = ""

    # Enable/disable CORS support
    cors = yes

    # Enable/disable X-API-Key from different host
    api-key-different-host = no

    # Max number of transactions
    # returned by /transactions/address/{address}/limit/{limit}
    transactions-by-address-limit = 1000
```

```
    distribution-address-limit = 1000
  }

  # Nodes synchronization settings
  synchronization {

    # How many blocks could be rolled back if fork is detected. If fork is
longer than this rollback is impossible.
    max-rollback = 100

    # Max length of requested extension signatures
    max-chain-length = 101

    # Timeout to receive all requested blocks
    synchronization-timeout = 60s

    # Time to live for broadcasted score
    score-ttl = 90s

    # Max baseTarget value. Stop node when baseTraget greater than this
param. No limit if it is not defined.
    # max-base-target = 200

    # Settings for invalid blocks cache
    invalid-blocks-storage {
      # Maximum elements in cache
      max-size = 30000

      # Time to store invalid blocks and blacklist their owners in advance
      timeout = 5m
    }

    # History replier caching settings
    history-replier {
      # Max microblocks to cache
      max-micro-block-cache-size = 50

      # Max blocks to cache
      max-block-cache-size = 20
    }

    # Utx synchronizer caching settings
    utx-synchronizer {
      # Max microblocks to cache
      network-tx-cache-size = 1000000

      # Max scheduler threads
      max-threads = 8

      # Max pending queue size
      max-queue-size = 5000

      # Send transaction to peers on broadcast request even if it's
already in utx-pool
```

```
      allow-tx-rebroadcasting = yes
    }

    # MicroBlock synchronizer settings
    micro-block-synchronizer {
      # How much time to wait before a new request of a microblock will be
done
      wait-response-timeout = 2s

      # How much time to remember processed microblock signatures
      processed-micro-blocks-cache-timeout = 3m

      # How much time to remember microblocks and their nodes to prevent
same processing
      inv-cache-timeout = 45s
    }
  }

  # Unconfirmed transactions pool settings
  utx {
    # Pool size
    max-size = 100000
    # Pool size in bytes
    max-bytes-size = 52428800 // 50 MB
    # Pool size for scripted transactions
    max-scripted-size = 5000
    # Blacklist transactions from these addresses (Base58 strings)
    blacklist-sender-addresses = []
    # Allow transfer transactions from the blacklisted addresses to these
recipients (Base58 strings)
    allow-blacklisted-transfer-to = []
    # Allow transactions from smart accounts
    allow-transactions-from-smart-accounts = true
    # Allow skipping checks with highest fee
    allow-skip-checks = true
  }

  features {
    auto-shutdown-on-unsupported-feature = yes
    supported = []
  }

  rewards {
    # desired = 0
  }

  extensions = [
    # com.wavesplatform.matcher.Matcher
    # com.wavesplatform.api.grpc.GRPCServerExtension
  ]

  # How much time to wait for extensions' shutdown
  extensions-shutdown-timeout = 5 minutes
}
```

```
# Performance metrics
kamon {
  # Set to "yes", if you want to report metrics
  enable = no

  # A node identification
  environment {
    service = "waves-node"

    # An unique id of your node to distinguish it from others
    # host = ""
  }

  metric {
    # An interval within metrics are aggregated. After it, them will be
sent to the server
    tick-interval = 10 seconds

    instrument-factory.default-settings.histogram {
      lowest-discernible-value = 100000 # 100 microseconds
      highest-trackable-value = 2000000000000 # 200 seconds
      significant-value-digits = 0
    }
  }

  # Reporter settings
  influxdb {
    hostname = "127.0.0.1"
    port = 8086
    database = "mydb"

    # authentication {
    #   user = ""
    #   password = ""
    # }
  }
}

# Non-aggregated data (information about blocks, transactions, ...)
metrics {
  enable = no
  node-id = -1 # ${kamon.environment.host}

  influx-db {
    uri = "http://"${kamon.influxdb.hostname}":"${kamon.influxdb.port}
    db = ${kamon.influxdb.database}

    # username = ${kamon.influxdb.authentication.user}
    # password = ${kamon.influxdb.authentication.password}

    batch-actions = 100
    batch-flash-duration = 5s
  }
```

```
}

# WARNING: No user-configurable settings below this line.

akka {
  loglevel = "INFO"
  loggers = ["akka.event.slf4j.Slf4jLogger"]
  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"
  log-dead-letters-during-shutdown = false

  http.server {
    max-connections = 128
    request-timeout = 20s
    parsing {
      max-method-length = 64
      max-content-length = 1m
    }
  }

  io.tcp {
    direct-buffer-size = 1536 KiB
    trace-logging = off
  }
}

include "deprecated-settings.conf"
```

Процесс майнинга (Waves NG)

Процесс майнинга является ключевым для ноды, в конце концов ее основная задача в том, чтобы производить блоки с транзакциями. Чтобы это эффективно делать, нода также должна получать информацию о блоках от других нод и отправлять им свои блоки. Давайте рассмотрим упрощенную модель майнинга в Waves. Более подробная информация о процессе майнинга, включая формулы, есть в статье Fair Proof of Stake.

Proof of Stake

В основе майнинга лежит алгоритм Proof-of-Stake, который подразумевает, что вероятность сгенерировать блок каким-либо аккаунтом прямо пропорциональна балансу этого аккаунта. Давайте рассмотрим простейший случай: допустим, у нас есть аккаунт с балансом 10 млн Waves (из 100 млн выпущенных в момент создания). Вероятность смайнить блок будет 10%, иными словами мы будем генерировать 144 блока в сутки (1440 всего блоков в сети за сутки).

Теперь немного усложним. Хоть и выпущено всего 100 миллионов токенов, не все из них участвуют в майнинге (например, токены могут быть на бирже, а не на аккаунте ноды). Если в майнинге участвует 50 миллионов, то нода с балансом в 10 млн уже будет генерировать 288 блоков в сутки. Но на самом деле количество токенов, которые участвуют в майнинге, постоянно меняется, поэтому прямо предсказать, сколько будет смайнено блоков, не получится.

Вопрос, который возник у самых любопытных - **в каком порядке ноды будут генерировать блоки?*. Для ответа на этот вопрос потребуется углубиться в особенности реализации PoS в Waves, поэтому пристегнитесь и взбодритесь.

Можно сказать, что для ответа на вопрос "Кто будет следующим генератороом блока?" ноды используют информацию о балансах, времени между блоками и генератор псевдо-случайных чисел. Начнем с последнего, использовать `urandom` в данном случае не получится, так как он недетерминированный, и каждая нода получит свой результат. Поэтому ноды "договариваются" о рандоме. Каждый блок в цепочке содержит наряду с транзакциями, адресом ноды, сгенерировавшей блок, версией и времеменем, поле, называемое `generation-signature`. Взгляните, как выглядит блок номер 1908853 в мейннете в JSON представлении (без транзакций):


```
{
  "blocksize": 22520,
  "reward": 600000000,
  "signature":
"2kCWg8HMhLPXGDi94Y6dm9NRx4aXjXpVmYAE4y4KaPzgt1Z5EX9mevfWoiBLLr1cc1TZhTSqp
ozUJJZ3BpA5j3oc",
  "generator": "3PEFQiFMLm1gTVjPdfCErG8mTHRcH2ATaWa",
  "version": 4,
  "reference":
"3Jcr6m6SM3hZ1bu6xXBmAVhA2VEUHMvE6omhEiRFn3VhEuDkgb6sgeJUC1VNRB3vTSwPb5qh5
76a8DwGt3Ts72Tx",
  "features": [],
  "totalFee": 28800000,
  "nxt-consensus": {
    "base-target": 74,
    "generation-signature": "6cVJBZsjzuSqp7LPD3ZSw5V1BZ25hZQHioh9gHjWPKNq"
  },
  "desiredReward": 600000000,
  "transactionCount": 70,
  "timestamp": 1580458301503,
  "height": 1908853
}
```


Обратите внимание: для удобства структуры данных в этой книге представлены в формате JSON, но сами ноды работают с блоками, транзакциями, подписями и т.д. в бинарном формате. Для этого есть описания бинарных структур данных в документации, а с недавнего времени бинарный формат данных представляет из себя Protobuf.

Generation signature является SHA256 хэшом от `generation-signature` предыдущего блока и публичного ключа генератора блока. Первые 8 байт хэша `generting-signature` конвертируются в число и используется как некий рандом. Значение `base-target` отвечает за среднее время между блоками и пересчитывается во время генерации каждого блока. Если бы в сети постоянно были все ноды со всем стейком сейти, готорые сгенерировать блок, то `base-target` не был бы нужен, но коль это не так, нужен синтетический параметр, который меняется в зависимости от текущего времени между блоками и автоматически выравнивать среднее время между блоками в 60 секунд.

Итак, у нас есть параметры `hit`, который является псевдо-случайным числом, баланс каждого аккаунта и значение `base-target`, но что делать со всем этим ноде? Каждая нода, в момент получения нового блока по сети, запускает функцию проверки, когда будет ее очередь генерировать блок.

```
\delta t = f(hit, balance, baseTarget)
```

В результате выполнения этой функции, нода получает число секунд до момента, когда наступит ее время генерировать блок. Фактически, после этого нода устанавливает таймер, при наступлении которого начнет генерировать блок. Если она получит следующий блок до наступления таймера, то операция будет выполнена заново и таймер будет переставлен на новое значение \delta t.

Валидация блоков происходит таким же образом, за одним исключением, что в формулу подставляется баланс не этой ноды, а сгенерировавшей блок.

Waves NG

Если вы вообще что-то знаете про Waves, то могли слышать про Waves NG, который делает блокчейн Waves быстрым и отзывчивым. Waves-NG получил свое названия от статьи Bitcoin-NG: A Scalable Blockchain Protocol, которая была опубликована в 2016 году и предлагала способ масштабирования сети Bitcoin за счет изменнеия протокола генерации блоков. NG в названии расшифровывается как Next Generation, и действительно предложение помогло бы сети Bitcoin выйти на новый уровень по пропускной способности, но эта инициатива так никогда и не была реализована в Bitcoin. Зато была воплощена в протоколе Waves в конце 2017 года. Waves NG влияет на то, как генерируются блоки и ноды общаются друг с другом.

В момент наступления своего времени майнинга, нода генерирует так называемый ключевой блок (key block), становясь лидером. Ключевой блок не содержит транзакций, он является только началом блока, который будет меняться. Дальее лидер получает право генерировать так называемые микроблоки, которые добавляют новые транзакции в конец блока и меняют его сигнатуру. Например, лидер генерирует ключевой блок со следующими параметрами:


```
{
  "blocksize": 39804,
  "reward": 600000000,
  "signature":
"4oBqMB7szmsbSYYguiaAXSE7ZLy13e4x97EKMmA4gs6puRqPKzCVJkuC6Py9eTpiovhcLAYuU
SsnEYAi4i73tvoA",
  "generator": "3P2HNUd5VUPLMQkJmctTPEeeHumiPN2GkTb",
  "version": 4,
  "reference":
"4KEFeMDQgPdntzqmSNZ92NBSMcNft1o4EyQexNLXEdN3976XbdYwDgqaucd9gu2PJWt9tpt1w
uvRcTMiiDtkZaX7",
  "features": [],
  "totalFee": 0,
  "nxt-consensus": {
    "base-target": 66,
    "generation-signature": "HpFc5qqVftyjKbqhADkQGWBg38CVR9Bz29c7uDZKKvYV"
  },
```

```
  "desiredReward": 600000000,
  "transactionCount": 0,
  "timestamp": 1580472824775,
  "height": 1909100
}
```

В блоке нет транзакций, что видно из значения `transactionCount`, но основные параметры вроде подписи и ссылки на предыдущий блок (поле `reference`) уже есть. Создатель этого блока сможет через несколько секунд сгенерировать микроблок со всеми транзакциями, которые появились в сети за эти секунды, и разослать остальным нодам. При этом в блоке поменяются некоторые поля:

```
{
  // неизмененные параметры опущены
  "blocksize": 51385,
  "signature":
"4xMaGjQxMX2Zd4jMUUUs5cmemkVwT8Jc5sqx6wzMUokVqWg5jvWSDF6SBF1P7x4UNQjYsgsCs
4csa2qtRmG8j3g4",
  "totalFee": 65400000,
  "transactionCount": 167,
  "tranasctions": [{...}, {...}, ..., {...}]
}
```

В блок добавились 167 транзакций, которые увеличили размер блока, так же поменялась подпись блока и комиссия, которую заработает лидер.

Несколько важных моментов, которые важно понимать:

- Микроблок содержит только транзакции и подпись лидера, параметры консенсуса не дублируются
- Время генерации микроблоков зависит от настроек майнера (поле `waves.miner.micro-block-interval` в конфигурации задает значение для каждой ноды). По умолчанию лидер будет генерировать микроблоки каждые 5 секунд.
- При каждом новом микроблоке меняются данные последнего блока, поэтому последний блок называют "жидким" (liquid) блоком
- Ключевой блок и все микроблоки, которые к нему относятся, объединяются в один блок так, что в блокчейне не остается никаких данных о микроблоках. Можно сказать, что они используются только для передачи информации о транзакциях между нодами.

Лидер блока будет генерировать микроблоки и менять жидкий блок до тех пор, пока не будет сгенерирован другой ключевой блок в сети (то есть у какой-то другой ноды сработает таймер начала майнинга) или достигнуты лимиты блока на размер (1 МБ).

Что дает Waves NG?

Благодаря Waves NG сокращается время попадания транзакции в блок. То есть можно в своем приложении обеспечивать гораздо лучший пользовательский опыт. Пользователь может получать обратную связь по своей транзакции за ~5 секунд, если нет большой очереди за попадание в блок.

Только надо понимать, что попадание в блок не является гарантией финализации и блок может быть отменен (до 100 блоков в глубину, но на практике 2-3 блока в крайне редких случаях).

Waves NG делает нагрузку на сеть более равномерной. В случае отсутствия Waves NG блоки генерировались бы раз в минуту (сразу 1 МБ данных) и отправлялись бы по сети целиком. То есть можно представить ситуации, когда 50 секунд ноды (кроме майнера) ничего не делают и ждут, а потом принимают блок и валидируют его на протяжении 10 секунд. С Waves NG эта нагрузка более размазана по времени, ноды получают каждые 5 секунд новую порцию данных и валидируют их. Это в целом повышает пропускную способность.

Waves NG однако может себя иногда вести не очень удобно. Как вы помните, каждый блок содержит в себе поле `reference`, которое является ссылкой на поле `signature` предыдущего блока. `reference` фиксируется в момент генерации ключевого блока, и может случиться такое, что новый майнер поставит в своем ключевом блоке ссылку не на последнее состояние жидкого блока. Иными словами, если новый майнер блока `N` не успел получить и применить последний микроблок блока `N − 1` от предыдущего майнера, то он сошлется на "старую" версию блока `N − 1`, транзакции из последнего микроблока будут удалены из блока `N − 1` для всей сети.

Но не пугайтесь, это приведет только к тому, **что исключенные транзакции попадут в блок `N`**, вместо блока `N − 1`, в котором мы уже могли успеть увидеть эти транзакции в своем клиентском коде.

Waves NG так же влиет на распределение комиссий в блоке. Майнер получает 60% от комиссий из предыдущего блока и 40% из своего блока. Сделано это для того, чтобы исключить возможную "грязную игру" узлов, когда они будут специально ссылаться на самую первую версию предыдущего блока, чтобы забрать все транзакции оттуда и положить в свой блок, а соответственно получить и комиссии.

Получаемая комиссия может быть потрачена майнером в этом же блоке. Он может добавить в блок транзакцию, за которую получит комиссию в 0.1 Waves и следующей же транзакцией положить в блок, переводящую эти 0.1 Waves с его аккаунта.

Обновления протокола и другие голосования

Как вы могли понять из предыдущего раздела, протокол работы блокчейна, в особенности Waves NG, совсем нетривиальная штука. Но как и любой протокол, он может и должен меняться со временем, чтобы становиться лучше. Но тут не все так просто. Команда разработки Waves не может просто так выпустить обновления и приказать всем обновиться или сказать, что те, кто не обновится, перестанут работать - это противоречит принципам децентрализации. Многие блокчейны идут по пути жестких "форков", просто выпускается новый функционал с новой версией ноды, дальше кому надо - устанавливает и начинает майнить с поддержкой новой фичи. Кто согласен - переключается на новую цепочку, кто нет - продолжает майнить старую. Этот путь далеко не лучший и может вести к бесчисленному количеству форков, поэтому добавление нового функционала или изменение правил консенуса в Waves возможно только через процедуру предложения нового функционала и голосование.

Процедуру изменения параметров консенсуса с помощью голосований часто называют гаверненсом (governance). Мы не будем использовать это слово, потому что governance в Waves сейчас ограничен двумя типами голосований, в то время как во многих других блокчейнах возможных изменений гораздо

больше (баны аккаунтов, жесткие изменения балансов аккаунтов и другие зверства, плохо уживающиеся с принципами децентрализации).

Процедура предложения нового функционала

В экосистеме Waves есть неписанное правило (писанного быть не может, децентрализация ведь), что все новые функции и изменения консенсуса должны проходить через процедуру обсуждения. Все предложения по изменения являются так называемыми Waves Enhancement Proposal или WEP. У каждого WEP есть порядковый номер, четка определенная структура и вопросы, на которые это предложение должно отвечать. Форма WEP была предложена на форуме Waves в специальном разделе, но в данный момент основные обсуждения и предложения на github.

Итак, каждое предложение формулируется в виде WEP, далее это предложение обсуждается всеми заинтересованными сторонами, вносятся корректировки, уточняются формулировки и т.д. В конечно итоге, любой человек может реализовать предложенный WEP в коде ноды (репозиторий ведь открытый) и отправить Pull Request на добавление изменений в основную ветку разработки. Команда Waves отвечает за качество кода в репозитории, поэтому при отсутствии проблем с качеством, код будет добавлен в основную ветку и попадет в сборку следующего релиза, которые так же публикуются на github. Но это не говорит о том, что новый код начнет работать, потому что каждый новый функционал (далее фича, от слова "feature") должен быть одобрен не только разработчиками, но и сообществом, для этого запускается процедура голосования.

Голосования за правила консенсуса

Как только владелец ноды устанавливает новую версию, у него появлется возможность голосовать за то, чтобы новая фича была активирована. У каждой фичи есть порядкой номер, по которому идет голосование и идентификация, обычно номера новых фич перечислены в описаниях к релизу на Github, но так же можно посмотреть в API ноды. Владелец ноды может добавить номер поддерживамой фичи в свою конфигурацию в массив `waves.features.supported`. После этого (а точнее после перезапуска ноды) в каждый генерируемый блок от этой ноды начинает добавляться номер поддерживаемой фичи. То есть, в бинарном представлении блока (в котором и идет работа), появляется новое значение с номером фичи.

Для того, чтобы фича была активирована, необходимо, чтобы ее поддержка была не менее 80% - не менее 80% блоков за период голосования должны в себе содержать информацию о поддержке фичи. Периоды голосования начинаются на каждом кратном десяти-тысячном блое (блок номер 100 000, 110 000, 120 000 и т.д.). Проще говоря, из блоков с номерами 100000-109999 не менее 80% содержать информацию о поддержке новой фичи.

Фактически гаверненс в Waves устроен очень близко к тому, как работает система выборов президента США. Жители страны напрямую не голосуют за президента, они выбирают представителей от каждого штата (количество представителей разнится от штата к штату. Представители уже непосредственно голосуют за президента США.

В Waves владельцы токенов могут напрямую голосовать за активацию фичи, однако, в большинстве случаев они не имеют своих нод и сдают свои токены в лизинг, передавая, таким образом, свое право голоса, владельцу лизингового пула. Важно понимать, что они в любой момент могут отменить лизинг, если владелец пула голосует не так, как они хотели бы, что снизит количество блоков, которые сгенерирует данный пул. Самые большие пулы в Waves часто делают голосования среди лизеров с

помощью децентрализованных приложений (мы рассмотрим пример такого голосования в разделе 7), так что и представительская ***демократия***/децентрализация могут быть прозрачными.

Если фича была поддержана более чем 80% стейка, то она будет активирована через 10000 блоков с момента завершения периода голосования. Например, если голосование была во время блоков 10000-19999, то фича станет активной на высоте 30000. Данная задержка позволяет еще раз проверить, что все нормально и новая фича не вызовет форки.

Если посмотреть в код ноды или в API (`/activation/status`), то можно заметить, что у каждой фичи есть следующие возможные статусы:

- **VOTING** - идет голосование по фиче
- **APPROVED** - фича одобрена, но пока не активирована (из-за задержки в 10 000 блоков)
- **ACTIVATED** - фича активирована, все ноды на этой цепочке должны поддерживать данную фичу

Как видите, у фичи нет статуса `REJECTED`, то есть голосование за фичу может идти бесконечно.

Голосование за вознаграждение за блок

В Waves есть еще один вид голосования нодами, который не встречается практически нигде - голосование за вознаграждение за блок. В 2016 году блокчейн Waves запускался с ограниченной эмиссией токенов - 100 миллионов, коорые уже были созданы на момент запуска сети. Но осенью 2019 года комьюнити осознало, что все-таки модель с эмиссией является более живой, поэтому было предложение обновление - WEP-7, которое прошло через процедуру голосования и на блоке N фича была активирована. Теперь за каждый сгенерированный блок, майнер получает не только комиссии (и то 40% от своего блока и 60% от предыдущего, вы ведь помните про Waves NG?), но и получает Waves, "генерируемые из воздуха". Примерно каждую минуту общее количество Waves в природе увеличивается на определенное значение. На какое именно значение - предмет голосования нод.

Каждые 100 тысяч блоков (примерно 2 с половиной месяца) начинается голосование за величину вознаграждения. В момент активации фичи N было установлено значение в 6 Waves. Каждые 100 000 блоков это значение может меняться не больше, чем на 0.5 Waves, и то при условии поддержки более чем 50% майнеров.

Голосование за вознаграждение за блок осуществляется немного по другому принципу, не как в случае с фичами. Владельцы нод могут в своем конфигурационном файле в параметре `waves.reward.desired` установить значение вознаграждения, которое хотели бы увидеть в долгосрочной перспективе. По истечении периода голосования, подсчитывается сколько блоков содержат желаемое вознаграждение больше, чем текущее, и если больше 50%, то вознаграждение увеличивается на 0.5 Waves на следующие 110 000 блоков (наступление нового периода голосования + сам период голосования).

Некоторые участники сообщество спрашивали, почему в `waves.reward.desired` просто не указывать `+` или `-`, раз все равно вознаграждение не будет изменено больше, чем на 0.5 Waves. Указание желаемого вознаграждения в долгосрочной перспективе избавляет от необходимости частого изменения конфигурации. Вы можете поставить значение `10` и не лезть каждый период голосования в конфигурацию, так как вы будете голосовать за увеличение до тех пор, пока вознаграждение не достигнет 10 Waves за блок. А как только достигнет (если достигнет), нода перестанет голосовать за увеличение вознаграждения. Так просто.

Ключи в блокчейне Waves

Первое, с чем сталкивается человек, когда начинает пользоваться блокчейном - работа с ключами. В отличие от классических веб приложений, где у нас есть логин и пароль, в блокчейнах есть только ключи, которые позволяют идентифицировать пользователя и валидность его действий.

У каждого аккаунта есть публичный ключ и соответствующий ему приватный. Публичный ключ является фактически идентификатором аккаунта, в то время как приватный позволяет сформировать подпись. В Waves используется подписи с использованием кривой Curve25519-Ed25519 с ключами X25519 (что иногда является проблемой, потому что поддержка ключей X25519 есть далеко не во всех библиотеках).

Публичный и приватный ключи представляют из себя 32 байтовые значения, которые соответствуют другу по определенным правилам (подробнее можете найти в описании EdDSA). Важно понимать несколько вещей, которые отличаются по сравнению с другими блокченами:

- не любые 32 байта могут быть приватным ключом
- приватный ключ не содержит в себе публичный ключ (например, в Ethereum приватный ключ содержит приватный, поэтому имеет размер в 64 байта, в Waves публичный ключ вычисляется каждый раз для приватного ключа)
- подпись с помощью EdDSA является недетерменированной, то есть одни и те же данные можно подписать одним и тем же ключом и получать разные подписи, так как используются и случайные значения

Путешествия ключа

Большинство пользователей все-таки сталкивается с ключами не в виде массива байт, а в виде сид-фразы, часто так же называемой мнемонической. Любая комбинация байт может быть сидом, но в клиентах Waves обычно используется 15 английских слов. На основе сид фразы вычисляется приватный ключ следующим образом:

1. строка переводится в массив байт
2. вычисляется хэш `blake2b256` для данного массива байт
3. вычисляется хэш `keccak256` для результата предыдущего шага
4. вычисляется приватный ключ на основе предыдущего шага, пример функции для этого шага представлен ниже

```
func GenerateSecretKey(hash []byte) SecretKey {
    var sk SecretKey
    copy(sk[:], hash[:SecretKeySize])
    sk[0] &= 248
    sk[31] &= 127
    sk[31] |= 64
    return sk
}
```

Иными словами, а точнее кодом: `privateKey = GenerateSecretKey(keccak256(blake2b256(accountSeedBytes)))`

Публичный и приватный ключи обычно представляют в виде `base58` строк вроде `3kMEhU5z3v8bmer1ERFUUhW58Dtuhyo9hE5vrhjqAWYT`.

При отправке транзакций (например, отправке токенов) пользователь имеет дело с адресом, а не публичным ключом получателя. Но адрес генерируется из публичного ключа получателя с некоторыми дополнительными параметрами: версия спецификации адреса, байт сети и чек-сумма. В данный момент в сети Waves есть только одна версия адресов, поэтому первым байтом в этой последоствальности является `1`, второй байт - уникальный идентификатор сети, который позволяет отличать адреса в разных сетях (mainnet, testnet, stagenet). Байты сети для перечисленных выше сетей `W`, `T`, `S` соответственно. Благодаря байту сети невозможно ошибиться и отправить токены на адрес, которого не может существовать в сети, в которой отправляется транзакция. После первых двух служебных байт идут 20 байт, полученных в результате функций хэширования blake2b256 и keccak256 над публичным ключом. Эта операция `keccak256(blake2b256(publicKey))` возвращает 32 байта, но последние 12 байт отбрасываются. Последние 4 байта в адресе являются чек-суммой, которая считается как `keccak256(blake2b256(data))`, где `data` это первые 3 параметра (версия, байт сети и 20 байт хэша публичного ключа). Полученная последовательность байт переводится в base58 представление, чтобы получилось похожее на это: `3PPbMwqLtwBGcJrTA5whqJfY95GqnNnFMDX`.

Опытные разработчики на Waves пользуются особенностями формирования адресов, чтобы по одному виду определять к какой сети относится адрес. Благодаря тому, что первые 2 байта в адресе похожи для всех адресов в одной сети, можно примерно понимать к какой сети относится адрес. Если адрес выглядит как `3P...`, то адрес с большой долей вероятности относится к mainnet, а если адрес начинается с `3M...` или `3N`, то перед вами скорее всего адрес из testnet или stagenet.

Работа с ключами

Если по какой-то причине, приложение необходимо генерировать ключи для пользователя, то можно воспользоваться библиотеками для разных языков программирования. Например, в библиотеке `waves-transactions` для JavaScript/TypeScript сгенерировать seed фразу можно с помощью следующего кода:

```
import {seedUtils} from '@waves/waves-transactions'

const seedPhrase = seedUtils.generateNewSeed(24);

console.log(seedPhrase);

// infant history cram push sight outer off light desert slow tape correct
chuckle chat mechanic jacket camp guide need scale twelve else hard cement
```

В консоль выведется строка из 24 слов, которые являются seed фразой нового аккаунта. Эти слова являются случайным подмножеством из словаря, который есть в коде библиотеки @waves/ts-lib-crypto и в котором содержится 2048 слов.

В данном примере я сгенерировал 24 слова, но по умолчанию во многих приложениях Waves генерируется набор из 15 слов. Почему именно 15 и увеличивается ли безопасность, если сгенерировать больше слов?

15 слов из 2048 в любом порядке достаточно, для того, чтобы вероятность генерации двух одинаковых seed фраз была пренебрежительно мала. В то же время, 24 слова еще уменьшают такую вероятность, почему бы не использовать большие значения? Ответ прост - чем больше слов мы используем, тем больше надо записывать и/или запоминать пользователю и тем сложнее ему будет. Смысл использования seed фразы (а не приватного ключа) именно в упрощении опыта пользователя, а с 24 словами мы заметно ухудшаем user experience.

Имея seed фразу можно получить приватный ключ, публичный и адрес. Я снова покажу как это сделать на JS, но вы же помните, что есть библиотеки и для других языков?

```
import {seedUtils} from '@waves/waves-transactions';
import {
  address,
  privateKey,
  publicKey
} from '@waves/ts-lib-crypto'


const seedPhrase = seedUtils.generateNewSeed(24);

console.log(privateKey(seedPhrase));   //
3kMEhU5z3v8bmer1ERFUUhW58Dtuhyo9hE5vrhjqAWYT
console.log(publicKey(seedPhrase));    //
HBqhfdFASRQ5eBBpu2y6c6KKi1az6bMx8v1JxX4iW1Q8
console.log(address(seedPhrase, 'W')); //
3PPbMwqLtwBGcJrTA5whqJfY95GqnNnFMDX
```

Обратите внимание, что в функции `privateKey` и `publicKey` мы передаем только сид фразу, в то время как в `address` передаем еще один параметр `chainId` (он же байт сети). Как вы помните из объяснения выше, адрес в себе содержит такой дополнительный параметр.

Как аккаунт появляется в блокчейне

Мы разобрали как работают ключи, как связаны сид фраза, приватный и публичный ключ, а также как к ним относится адрес, но я не упомянул один очень важный момент, о котором забывают некоторые начинающие разработчики. До момента совершения какого-либо действия с аккаунтом (отправка с него или на него транзакции), блокчейн ничего не знает об этом аккаунте. Если вы сгенерировали аккаунт (у себя локально или в любом клиенте), но в блокчейне не блоы транзакций, связанных с этим аккаунтом (входящих или исходящих), вы не сможете найти никакую информацию о вашем аккаунте в эксплорере или с помощью API. Это отличается от поведения в централизованных системах и API, поэтому может быть не так интуитивно понятным и простым, но об этом важно помнить.

UTX

UTX - список транзакций, которые находятся в ожидании попадания в блок. То есть кто-то их отправил и нода приняла транзакцию, но транзакция в блок не попала. В Waves есть определенные особенности, связанные с тем как такие транзакции обрабатываются. Как транзакция может попасть в UTX? Существует всего 2 способа для этого:

- Кто-то отправит транзакцию на эту ноду (с помощью REST API)
- Нода получит транзакция по бинарному протоколу от другой ноды в сети

В конечном итоге можно сказать, что транзакции в сеть приходят через API, но не обязательно, чтобы это был API данной конкретной ноды.

Особенности обработки UTX

Транзакции в листе ожидания это ни разу не весело, они ведь хотят в блок! Как же определить какая транзакция должна первой попасть в блок. Можно было бы сделать простую очередь и руководствоваться принципом "Кто первый встал, того и тапки", но такой подход не является оптимальным для майнеров в сети. Им гораздо выгоднее класть в блок транзакции с большей комиссией. Но и тут не все так просто, вы ведь помните, что в Waves существует разные виды транзакций. У каждого вида своя минимальная комиссия, заданная в консенсусе, поэтому обработка в зависимости от размера комиссии тоже не приведете к ожидаемому результату. Например, отправка InvokeScript транзакций с минимальной комиссией в 0.005 Waves будет всегда попадать в блок раньше, чем транзакция Transfer с комиссией в 0.001 Waves. Что же делать?

Первое, что можно придумать, это сортировать транзакции в зависимости от стоимости на байт транзакции. Нода тратит ресурсы на валидацию подписи для транзакции, и чем больше транзакция по размеру, тем больше ресурсов на это потратится. Например, Data транзакция размером в 140 kb будет валидироваться в несколько десятков раз дольше, чем Transfer транзакция размером меньше килобайта. Давайте поговорим на примерах. Скажем, у нас есть 2 транзакции:

- Data транзакция размером в 100 kb и с комиссией в 0.01 Waves
- Transfer транзакция размером в 1kb и с комиссией в 0.001 Waves

Какая транзакция будет первой в очереди? Та, которая была получена первой, потому что в пересчете на 1 байт транзакции комиссия у этих 2 транзакций равная.

Но на этом не все, у транзакций в UTX есть еще один важный параметр - сложность исполнения скрипта.

Введение

RIDE - это компилируемый статически типизированный язык программирования на основе ленивых функциональных выражений, предназначенный для построения децентрализованных приложений без ошибок для разработчиков.

RIDE не является полным Тьюрингом. В нем нет никаких циклов, рекурсий и есть много ограничений по дизайну - это помогает сохранить его простым и понятным.

Несмотря на то, что он прост, он дает много возможностей разработчику.

Он похож на Scala, а также F#. Ride - это очень простой язык. Ознакомление с этой документацией займет у вас около часа, и к концу статьи, вы изучите почти весь язык.

Hello world

```
func say() = {
  "Hello world!"
}
```

Функции объявляются с помощью `func`. Функции возвращают типы, но их не нужно объявлять, за вас это сделает компилятор. В примере выше, функция `say` вернет строку `"Hello World!"`. В языке нет `return`, потому что RIDE основан на выражениях (все является выражением), а последний оператор является результатом функции.

Блокчейн

RIDE предназначен для использования внутри блокчейна, и нет никакого способа получить доступ к файловой системе или написать что-либо в консоли.

Функции RIDE могут считывать данные из блокчейна и возвращать транзакции в результате, которые будут записаны в блокчейн.

Комментарии

```
# Это комментарий

# В языке нельзя создавать многострочные комментарии

"Hello world!" # Можно писать комментарии и в таких местах
```

Директивы

Каждый скрипт на Ride должен начинаться с директив для компилятора. Существует 3 возможных типа директив с различными возможными значениями.

```
{-# STDLIB_VERSION 3 #-}
{-# CONTENT_TYPE DAPP #-}
{-# SCRIPT_TYPE ACCOUNT #-}
```

`STDLIB_VERSION` задает версию стандартной библиотеки. Последняя версия. выпущенная в продакшн - 3.

`CONTENT_TYPE` задает тип файла, над которым вы работаете. На данный момент существуют типы `DAPP` и `EXPRESSION`. Тип `DAPP` позволяет объявлять функции и завершать выполнение скрипта некоторыми

транзакциями (изменениями в блокчейне) и использовать аннотации, тогда как `EXPRESSION` позволяет завершать выполнение скрипта логическим выражением (true/false).

`SCRIPT_TYPE` задает тип сущности, к которой мы хотим добавить скрипт и изменить поведение по умолчанию. Скрипты на Ride можно прикрепить к `ACCOUNT` или `ASSET`.

```
{-# STDLIB_VERSION 3 #-}
{-# CONTENT_TYPE DAPP #-}
{-# SCRIPT_TYPE ASSET #-} # тип dApp нельзя использовать для ASSET
```

В примере выше не все комбинации директив являются правильными. Пример выше не будет работать, ибо тип контента `DAPP` допустим только для аккаунтов, в то время как тип `EXPRESSION` доступен для активов (ассетов) и аккаунтов.

Функции

```
func greet(name: String) = {
  "Hello, " + name
}

func add(a: Int, b: Int) = {
  a + b
}
```

Тип должен следовать за именем аргумента.

Как и во многих других языках функции не могут быть перегружены. Это помогает сохранить код простым, читаемым и ремонтопригодным.

Функции можно использовать только после их объявления.

Переменные

```
let a = "Bob"
let b = 1
```

Переменные объявляются и инициализируются с помощью ключевого слова `let`. Это единственный способ объявить переменные в RIDE. Все переменные в RIDE являются неизменными. Это означает, что вы не можете изменить значение переменной после объявления.

Тип переменной определеляется исходя из значения в правой части.

RIDE позволяет определять переменные внутри любой функции или в глобальной области видимости.

```
a = "Alice"
```

Приведенный выше код не будет компилироваться, потому что переменная `a` не определена. Все переменные в Ride нужно объявлять

```
func lazyIsGood() = {
  let a = "Bob"
  true
}
```

Функция выше будет скомпилирована и вернет `true` в качестве результата, но переменная `a` не будет инициализирована, потомучто RIDE ленивый, это означает, что все неиспользуемые переменные не вычисляются.

```
func callable() = {
  42
}

func caller() = {
  let a = callable()
  true
}
```

Функция `callable` также не будет вызвана, поскольку переменная `a` не используется.

В отличие от большинства языков, переиспользование переменных не допускается. Объявление переменной с именем, которое уже используется в родительской области приведет к ошибке компиляции.

Базовые типы

Основные базовые типы показаны ниже:

```
Boolean    #   true
String     #   "Hey"
Int        #   1610
ByteVector #   base58'...', base64'...', base16'...',
fromBase58String("...") etc.
```

Строки

```
let name = "Bob"
name + " is cool!" # строки будут соеденены, ибо используется знак +

name.indexOf("o")  # 1
```

В RIDE строка - это массив байтов, доступный только для чтения. Строковые данные кодируются с помощью UTF-8.

Для обозначения строк можно использовать только двойные кавычки. Строки неизменяемы, как и все другие типы. Это означает, что функция поиска подстроки в строке очень эффективна: копирование не выполняется, дополнительные выделения не требуются.

Все операторы в RIDE должны иметь значения одного и того же типа с обеих сторон. Этот код не будет компилироваться, потому что `age` - `int`:

```
let age = 21
"Bob is " + age # won't compile
```

Чтобы заставить код работать, мы должны преобразовать `age` в `string`:

```
let age = 21
"Alice is " + age.toString() # вот так работает!
```

Специальные типы

```
List    # [16, 10, "hello"]
Nothing #
Unit    # unit
```

RIDE имеет несколько типов, которые "**выглядят** как ~~утки~~ в Scala, плавают как ~~утки~~ в Scala и крякают как ~~утки~~ в Scala".

В RIDE нет типа `null`, как во многих других языкаъ. Обычно, встроенные функции возвращают тип `Unit` вместо `null`.

```
"String".indexOf("substring") == unit # true
```

Списки

```
let list = [16, 10, 1997, "birthday"]       # коллекция может содержать
различные типы данных
let second = list[1]                        # 10 – второе значение из
списка
```

Для правильной работы со списками в RIDE, у них всегда должен быть известен размер, потому что нет циклов и рекурсий.

`List` не имеет полей, но в стандартной библиотеке есть функции, которые позволяют работать с ними проще.

```
let list = [16, 10, 1997, "birthday"]

let last = list.getElement(list.size() - 1) # "birthday", postfix call of
size() function

let lastAgain = getElement(collection, size(collection) - 1) # то же, что
и выше
```

Функция `.size()` возвращает длину списка. Обратите внимание, что это значение только для чтения, и оно не может быть изменено пользователем.

```
let initList = [16, 10]            # init value
let newList = cons(1997, initList) # добляет новый элемент в список -
[1997, 16, 10]
```

Вы можете добавить новый элемент к существующему списку с помощью функции `cons`. Нет никакого способа объединить два списка или добавить несколько значений в список.

Union-типы

```
let valueFromBlockchain = getString("3PHHD7dsVqBFnZfUuDPLwbayJiQudQJ9Ngf",
"someKey") # Union(String | Unit)
```

Типы Union - это очень удобный способ работы с абстракциями, `Union(String | Unit)` показывает, что значение является пересечением этих типов.

Пример ниже:

```
type Human : { firstName: String, lastName: String, age: Int}
type Cat : {name: String, age: Int }
```

`Unioin(Human | Cat)` является объектом с одним полем `age`, но мы можем использовать сопоставление шаблонов:

```
Human | Cat => { age: Int }
```

Сопоставление шаблонов предназначено для получения определенного поля объекта:

```
let t = ... # Cat | Human
  t.age                    # OK
  t.name                    # Compiler error
  let name = match t {            # OK
    case h: Human => h.firstName
    case c: Cat   => c.name
  }
```

`getString` возвращает `Union(String | Unit)` потому что при чтении данных из блокчейна (состояние ключа-значения) некоторые пары ключ-значение могут не существовать.

```
let valueFromBlockchain = getString("3PHHD7dsVqBFnZfUuDPLwbayJiQudQJ9Ngf",
"someKey")
let realStringValue = valueFromBlockchain.extract()

# or
let realStringValue2 = getStringValue(this, "someKey")
```

Чтобы получить реальный тип и значение от использования Union, используйте функцию `extract`, которая прервет сценарий в случае значения `Unit`. Другой вариант заключается в использовании специализированных функций, таких как `getStringValue`, `getIntegerValue` и др.

If

```
let amount = 1610
if (amount > 42) then "I claim that amount is bigger than 42"
else if (amount > 100500) then "Too big!"
else "I claim something else"
```

`if` операторы довольно просты и похожи на большинство других языков, за исключением двух отличий: `if` может использоваться как выражение (результат присваивается переменной) и ветвь `else` всегда требуется.

```
let a = 16
let result = if (a > 0) then a / 10 else 0 #
```

Сопоставление с образцом

```
let readOrInit = match getInteger(this, "someKey") {
    case a:Int => a
```

```
    case _ => 0
}
```

Сопоставление с образцом - это механизм проверки значения по образцу. RIDE позволяет использовать сопоставление с образцом только для предопределенных типов.

// : fix the definition below

Сопоставление с образцом в RIDE выглядит так же, как в Scala, но единственным вариантом использования сейчас является получение реального типа `Union` типизированной переменной. Сопоставление с образцом может быть полезно в случаях очень сложных типов, таких как `Union(Order | ReissueTransaction | BurnTransaction | MassTransferTransaction | ExchangeTransaction | TransferTransaction | SetAssetScriptTransaction | InvokeScriptTransaction | IssueTransaction | LeaseTransaction | LeaseCancelTransaction | CreateAliasTransaction | SetScriptTransaction | SponsorFeeTransaction | DataTransaction)`.

```
let amount = match tx { # tx – текущий объект исходящей транзакции в
глобальной области видимости
  case t: TransferTransaction => t.amount
  case m: MassTransferTransaction => m.totalAmount
  case _ => 0
}
```

Приведенный выше код показывает пример использования сопоставления с образцом.В блокчейне Waves существуют различные типы транзакций, и в зависимости от типа реальное количество передаваемых токенов может храниться в разных полях. Если транзакция типа `TransferTransaction` или `MassTransferTransaction` мы возьмем правильное поле, во всех остальных случаях мы получим 0.

Чистые функции

#**LET THE HOLY WAR BEGIN**

Функции RIDE являются чистыми по умолчанию, что означает, что их возвращаемые значения определяются только их аргументами, и их оценка не имеет побочных эффектов.

Это достигается отсутствием глобальных переменных и тем, что все аргументы функции по умолчанию остаются неизменными.

Однако RIDE не является чистым функциональным языком, поскольку существует функция `throw ()`, которая завершает выполнение скрипта в любой момент.

```
let a = getInteger(this, "key").extract()
throw("I will terminate it!")
if a < 0 then
        "a is negative"
```

```
else
        "a is positive or 0"
```

В приведенном выше примере скрипт завершится на строке 2 с сообщением `I will terminate it!` и никогда не достигнет выражения `if`.

Аннотации / модификаторы доступа

Функции могут быть определены только в скрипте с помощью `{-# CONTENT_TYPE DAPP #-}`. Функции могут быть без аннотаций, либо с аннотациями `@Callable` или `@Verifier`.

```
func getPayment(i: Invocation) = {
  let pmt = extract(i.payment)
  if (isDefined(pmt.assetId)) then
    throw("This function accetps waves tokens only")
  else
        pmt.amount
}

@Callable(i)
func pay() = {
  let amount = getPayment(i)
  WriteSet([DataEntry(i.caller.bytes, amount)])
}
```

Функции с аннотацией `@Callable` могут быть вызваны извне блокчейна. Для вызова вызываемых функций необходимо отправить `InvokeScriptTransaction`

Аннотации могут привязывать некоторые значения к функции. В приведенном выше примере переменная `i` была привязана к функции `pay` и хранила всю информацию о факте вызова (открытый ключ, адрес, платеж, прикрепленный к транзакции, комиссию, transactionId и т.д.).

Functions without annotations are not available from the outside. You can call them only inside other functions.

```
@Verifier(tx)
func verifier() = {
  match tx {
    case m: TransferTransaction => tx.amount <= 100 # можно отправить не
больше 100 токенов
    case _ => false
  }
}
```

Функции с аннотацией `@Verifier` устанавливают правила для исходящих транзакций децентрализованного приложения (dApp). Функции верификатора нельзя вызывать извне, но они выполняются каждый раз, когда предпринимается попытка отправить транзакцию из dApp.

Функции верификатора должны всегда возвращать значение `Boolean` в качестве результата, в зависимости от этого транзакция попадет в блокчейн или нет.

Функция верификатора использует переменную `tx`, которая является объектом со всеми полями текущей исходящей транзакции.

В одном скрипте может быть определена только одна функция верификатора.

```
@Callable(i)
func callMeMaybe() = {
  let randomValue = getRandomValue()
  WriteSet([DataEntry("key", randomValue)])
}

func getRandomValue() = {
  16101997 # достаточно рандомное число
}
```

Это не будет компилироваться, потому что функции **без** аннотаций должны быть определены **перед** функциями с аннотациями.

Предопределенные структуры данных

RIDE имеет много предопределенных специфических структур данных для Waves Blockchain, таких как: `Address`, `Alias`, `DataEntry`, `ScriptResult`, `Invocation`, `ScriptTransfer`, `TransferSet`, `WriteSet`, `AssetInfo`, `BlockInfo`.

```
let keyValuePair = DataEntry("someKey", "someStringValue")
```

`DataEntry` это структура данных, которая описывает пару ключ-значение, как в хранилище учетных записей.

```
let transferSet =
TransferSet([ScriptTransfer("3P23fi1qfVw6RVDn4CH2a5nNouEtWNQ4THs", amount,
unit)])
```

Все структуры данных могут быть использованы для проверки типов, сопоставления с образцами и их конструкторов.

Незультат выполнения

```
@Verifier(tx)
func verifier() = {
  "Returning some string"
}
```

Expression-скрипты (с директивой `{-# CONTENT_TYPE EXPRESSION #-}`) наряду с функциями `@Verifier` должны всегда возвращать логическое выражение. В зависимости от этого значения транзакция будет принята (в случае `true`) или отклонена (в случае `false`) блокчейном.

```
@Callable(i)
func giveAway(age: Int) = {
  ScriptResult(
    WriteSet([DataEntry("age", age)]),
    TransferSet([ScriptTransfer(i.caller, age, unit)])
  )
}
```

Каждый, кто вызовет функцию `giveAway` получит столько waves, сколько ему лет, и dApp будет хранить информацию о факте передачи в своем "состоянии".

`@Callable` функции могут заканчиваться двумя типами изменений блокчейна - изменениями состояния и передачей токенов.

Список `DataEntry` в `WriteSet` устанавливает или обновляет пары ключ-значение в хранилище учетной записи, а список `ScriptTransfer` в `TransferSet` перемещает токены из учетной записи dApp в другие учетные записи.

```
@Callable(i)
func callMePlease(age: Int) = {
  TransferSet([ScriptTransfer(i.caller, age, unit)])
}
```

`@Callable` функции могут возвращать одну из следующих структур: `ScriptResult`, `WriteSet`, `TransferSet`.

`WriteSet` может содержать до 100 `DataEntry`, `TransferSet` может содержать до 10 `ScriptTransfer`.

Исключения

```
throw("Here is exception text")
```

`throw` функция немедленно завершит выполнение скрипта с предоставленным текстом. Нет никаких способов поймать брошенные исключения.

Основная идея `throw` заключается в том, чтобы остановить выполнение и отправить пользователю информативную обратную связь.

```
let a = 12
if (a != 100) then throw ("a is not 100, actual value is " + a.toString())
else throw("A is 100")
```

`throw` функция может быть использована для отладки кода при разработке dApps, потому что нет отладчиков для RIDE пока что.

Контекст выполнения

```
{-# STDLIB_VERSION 3 #-}
{-# CONTENT_TYPE EXPRESSION #-}
{-# SCRIPT_TYPE ACCOUNT #-}

let a = this # Адрес текущего аккаунта
a == Address(base58'3P9DEDP5VbyXQyKtXDUt2crRPn5B7gs6ujc') # true если
скрипт запущен на аккаунте с определенным адресом
```

RIDE скрипты в блокчейне waves могут быть привязаны к счетам и активам (`{-# SCRIPT_TYPE ACCOUNT #-}`) и в зависимости от `SCRIPT_TYPE` ключевое слово `this` может ссылаться на различные сущности. Для типа скрипта `ACCOUNT`, `this` это тип `Address`

Для типа скрипта `ASSET`, `this` это тип `AssetInfo`

```
{-# STDLIB_VERSION 3 #-}
{-# CONTENT_TYPE EXPRESSION #-}
{-# SCRIPT_TYPE ACCOUNT #-}

let a = this # Адрес текущего аккантуа
a == Address(base58'3P9DEDP5VbyXQyKtXDUt2crRPn5B7gs6ujc') # true если
скрипт запущен на аккаунте с определенным адресом
```

How to: бесплатные (для пользователя) децентрализованные приложения

Среди разработчиков децентрализованных приложений есть несколько тем, обсуждение которых приводит к явно выраженной боли на лицах. Такими темами являются:

1. **Работа с ключами.** Просить у пользователя ключи нельзя, но он должен как-то подписывать транзакции.
2. **Необходимость платить комиссию в токенах за каждую транзакцию**. Как объяснить пользователям, что каждая транзакция требует комиссии в токене платформы, и что не менее важно - откуда они возьмут комиссии для своих первых транзакций?

Обозначенные проблемы приводят к очень высокой стоимости привлечения одного пользователя. Например, один из популярных dApp в экосистеме Waves имел стоимость привлечения клиента около $80 (!), при LTV меньше $10. Конверсию портили именно барьеры с расширением и комиссиями.

Первая проблема часто решается с помощью браузерных расширений вроде Metamask и Waves Keeper, но это решение не дружественное для пользователей и требует большого количества усилий, поэтому в экосистеме Waves появился Signer. Он не требует предоставлять ключи dApp, и в то же время не заставляет устанавливать браузерные расширения. В статье @Vladimir Zhuravlev рассказывается об этом и как интегрировать Waves Signer в свое приложение.

А что же по поводу второй проблемы? Многие создатели dApp просто не заботятся этим вопросом, пользователи должны откуда-то взять токены для комиссий. Другие требуют привязывать банковские карты во время регистрации, что очень сильно снижает мотивацию.

Сейчас я расскажу как решить проблему с комиссиями. Как сделать такой dApp, который не требует наличия нативного токена у пользователя. Это позволяет делать триальные периоды. В Waves существует 2 способа сделать это. Разберем оба варианта.

Спонсирование транзакций

Если у вас есть свой токен, который нужен пользователям вашего dApp, то вы можете использовать механизм спонсирования транзакций. Пользователи будут платить комиссию в вашем токене, но так как майнеры всегда получают комиссию только в Waves, то фактически Waves будут списываться с аккаунта, выпустившего токен. Давайте еще раз по шагам, так как это важно понимать:

- Пользователь платит комиссию за транзакцию в вашем токене
- Вы получаете эти токены
- С вашего аккаунта списываются WAVES в необходимом количестве и уходят майнерам

Вопрос, который должен был сразу возникнуть - сколько токенов заплатит пользователь и сколько токенов спишется с аккаунта спонсора?

Ответ: владелец может сам установить соотношение. В момент начала спонсирования создатель токена задает сколько его токенов соответствуют минимальной комиссии (0.001 Waves или 100 000 в минимальной фракции). Давайте перейдем к примерам и коду, чтобы было понятнее.

Для включения спонсирования, необходимо отправить транзакцию типа `Sponsorship`. С помощью пользовательского интерфейса можно сделать в Waves.Exchange, а с помощью waves-transactions можно выполнить следующий код:

```
const { sponsorship } = require('@waves/waves-transactions')

const seed = 'example seed phrase'

const params = {
  assetId: 'A',
  minSponsoredAssetFee: 100
}

const signedSponsorshipTx = sponsorship(params, seed)
```

Код выше сформирует (но не отправит в блокчейн) транзакцию:


```
{
  "id": "A",
  "type": 14,
  "version": 1,
  "senderPublicKey": "3SU7zKraQF8tQAF8Ho75MSVCBfirgaQviFXnseEw4PYg",
  "minSponsoredAssetFee": 100,
  "assetId": "4uK8i4ThRGbehENwa6MxyLtxAjAo1Rj9fduborGExarC",
  "fee": 100000000,
  "timestamp": 1575034734209,
  "proofs": [

"42vz3SxqxzSzNC7AdVY34fM7QvQLyJfYFv8EJmCgooAZ9Y69YDNDptMZcupYFdN7h3C1dz2z6
keKT9znbVBrikyG"
  ]
}
```


Самым главным параметром в транзакции является `minSponsoredAssetFee`, который задает соответствие 100 токенов `A` равны 0.001 Waves. Таким образом, чтобы отправить `Transfer` пользователь должен будет в качестве комиссии приложить 100 токенов `A`.

Важно понимать некоторые ограничения, связанные со спонсированием. Использовать спонсированные токены как комиссию можно только для транзакций типов `Transfer` и `Invoke`. Спонсировать токен может только аккаунт, выпустивший этот токен. То есть, вы не сможете спонсировать токены, выпущенные не вами.

Безопасность

Прежде чем включать спонсирование, надо понимать несколько важных моментов.

1. Пользователь может использовать спонсируемые токены для операций не только с этим токеном. Например, аккаунт с токенами `A` на балансе может отправлять токены `B`, а как комиссию приложить токены `A`.
2. Пользователь может платить не минимальную комиссию за транзакцию. Например, если у пользователя есть 100 000 ваших токенов, а вы поставили параметр `minSponsoredAssetFee` равным 100, то пользователь сможет все свои 100 000 токенов указать в качестве комиссии. Вы получите 100 000 токенов `A`, а майнер получит 1000 Waves с вашего аккаунта (100 000 / 100 = 1000), если они есть на вашем аккаунте.

Функция спонсирования есть в Waves долгое время и отлично работает, но есть WEP-2 Customizable Sponsorship, в котором высказывались идеи по его улучшению. Если вам есть что добавить - присоединяйтесь к обсуждению.

Оплата комиссии за счет dApp

Дисклеймер: данная возможность является НЕ документированной и может перестать работать в будущем.

Второй вариант спонсирования подходит только для одного типа транзакций - `InvokeScript` и представляет собой достаточно простой механизм. Преимуществом по сравнению со спонсированием является отсутствие необходимости использовать свой токен.

Любой пользователь может вызвать dApp имея 0 Waves на своем аккаунте, но указав в качестве комиссии именно Waves. Скрипт начнет выполняться и, если в результата выполнения скрипта на аккаунт пользователя перечисляются Waves, то он получит их и уже оттуда спишется комиссия. Давайте на примере, так всегда ведь проще.

Например, есть dApp, который на основе адреса вызывающего отправляет или нет ему 1 Waves. **Данный пример совершенно не безопасен и приводится только для объяснения принципа работа. Ни в коем случае не используйте такую логику в своих децентрализованных приложениях.**

```
@Callable(invoke)
func callMeBaby() = {

    let callerAddress = invoke.caller.bytes.toBase58String()

    if (takeRight(invoke.caller.bytes, 4) == base16'ABCD') then
        TransferSet([
            ScriptTransfer(invoke.caller, 100000000, unit)
        ])
    else
    throw("You didn't win")
}
```

Если вызывать эту функцию с аккаунта, на котором 0 Waves баланса, указав `fee` в 0.005 Waves, скрипт все равно начнет выполняться. Если пользователь имеет право на получение 1 Waves, то транзакция будет считаться валидной и попадет в блокчейн. Пользователь СНАЧАЛА получит 1 Waves в результате вызова, затем с него будет списано 0.005 комиссии. По итогу пользователь получит на своем аккаунте 0.995 Waves.

Внимательные читатели уже догадались, что скрипт выше является небезопасным, потому что можно вызывать его бесконечное количество раз - если аккаунт не выиграл, то транзакция не попадет в блокчейн, если аккаунт выиграл, то получит 0.995 Waves. Беспроигрышная лотерея получилась.

Правильный рецепт заключается в том, чтобы всегда проверять дополнительные условия имеет ли пользователь права вызывать скрипт, например, иметь свой белый список.

```
@Callable(invoke)
func callMeBaby(uuid: String) = {

    if (isInWhiteList(invoke.caller) && invoke.fee == 500000) then {

        let id = extract(invoke.transactionId)
        let callerAddress = toBase58String(invoke.caller.bytes)
```

```
        ScriptResult(
            WriteSet([
                DataEntry(toBase58String(id), callerAddress + uuid)
            ]),
            TransferSet([
                ScriptTransfer(addressFromStringValue(callerAddress),
invoke.fee, unit)
            ])
        )
    }
    else
        throw()
}
```

Хочется дополнительно отметить, что если в будущем модель исполнения контрактов в Waves изменится и транзакции с ошибками (`throw()`) тоже будут записываться в блокчейн и снимать комиссию, то использование этого функционала станет невозможным.

Обсуждение того, как работает описанный выше функционал и о возможных путях развития вы можете найти в этом issue на гитхаб.

Лучшие практики

Оба решения рекомендуется использовать только для триальных режимов продуктов и проверять все граничные условия, потому что неправильное использовать может привести к потере средств.