Blockchain - это технология, на базе которой построен Bitcoin. И если пару лет назад вся слава доставлась криптовалюте, то сегодня все чаще можно слышать смелые фразы вроде: "Forget Bitcoin, Long Live Blockchain". Активно развиваются платформы вроде Ethereum, IPFS или Overstock, которые рассматривают блокчейн не как инструмент для создания еще одной платежной системы, а как совершенно обособленную технологию, сравнимую по своей инновационности разве что с Интернетом.

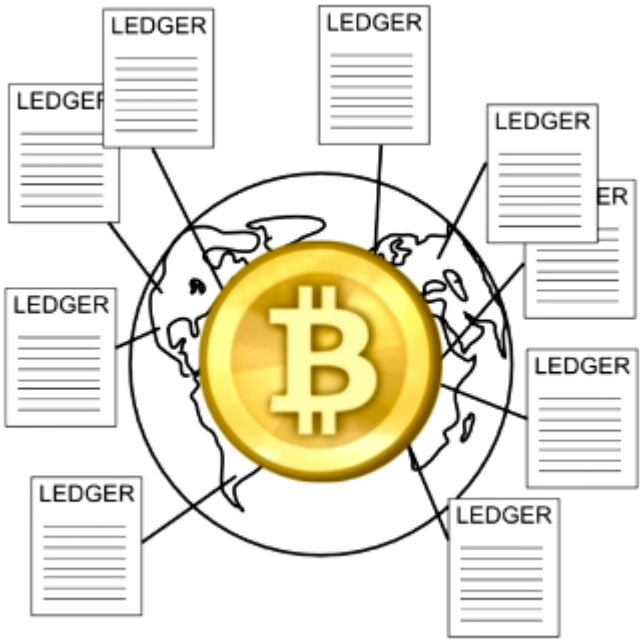
В этой главе я расскажу вам, что из себя представляет блокчейн Bitcoin. Даже по сравнению с Ethereum, это жуткий анахронизм, но понимание принципов его работы вам сильно поможет, если вы решите разобраться с более сложными проектами.

Table of content

1. Blockchain for dummies
2. Structure
3. Merkle tree
4. Timestamp
5. Raw block
6. Links

Blockchain for dummies

Сам по себе блокчейн - это крайне простая штука. Его проще всего проиллюстрировать на примере книги бухгалтерского учета, в которую записываются все транзакции в сети Bitcoin. Более того, такая книга присутствует у каждого участника сети, а значит любой, при желании, может проверить, была та или иная транзакция в реальности или нет.



И если блокчейн целиком - это книга, то отдельные блоки можно представлять как страницы, на которых "записываются" транзакции. Кажый блок "ссылается" на предыдущий и так до самого первого блока (*genesis block*). Именно это и создает такую интересную особенность блокчейна, как неизменяемость. Нельзя взять и изменить блок #123 так, чтобы этого никто не заметил. Потому что блокчейн устроен таким образом, что это повлечет изменение блока #124, потом #125 и так далее, до самого верха.

Structure

Привычным движением руки открываем спецификацию протокола и смотрим на структуру блока.

Field Size	Description	Data type	Comments
4	version	int32_t	Block version information (note, this is signed)
32	prev_block	char[32]	The hash value of the previous block this particular block references
32	merkle_root	char[32]	The reference to a Merkle tree collection which is a hash of all transactions related to this block
4	timestamp	uint32_t	A Unix timestamp recording when this block was created (Currently limited to dates before the year 2106!)
4	bits	uint32_t	The calculated difficulty target being used for this block
4	nonce	uint32_t	The nonce used to generate this block… to allow variations of the header and compute different hashes
?	txn_count	var_int	Number of transaction entries
?	txns	tx[]	Block transactions, in format of "tx" command

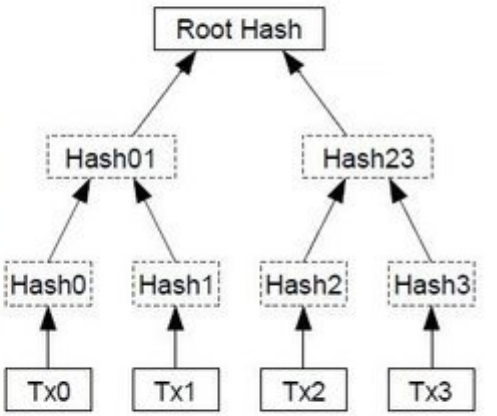
- *version* - версия блока
- *prev_block* - хэш предыдущего блока (*parent block*)
- *merkle_root* - если упрощенно, то это хэш всех транзакций в блоке
- timestamp - дата и время создания блока
- *bits*, *nonce* - про эти параметры я подробно расскажу в главе Bitcoin in a nutshell - Mining
- *txn_count*, *txns* - число транзакций в блоке и их список

Первые шесть параметров (все кроме *txn_count* и *txns*) образуют заголовок блока (*header*). Именно хэш заголовка называют хэшем блока, то есть сами транзакции непосредственного участия в хэшировании не принимают.

Вместо этого они заносятся в особую структуру - *дерево Меркла*, про которую я расскажу ниже.

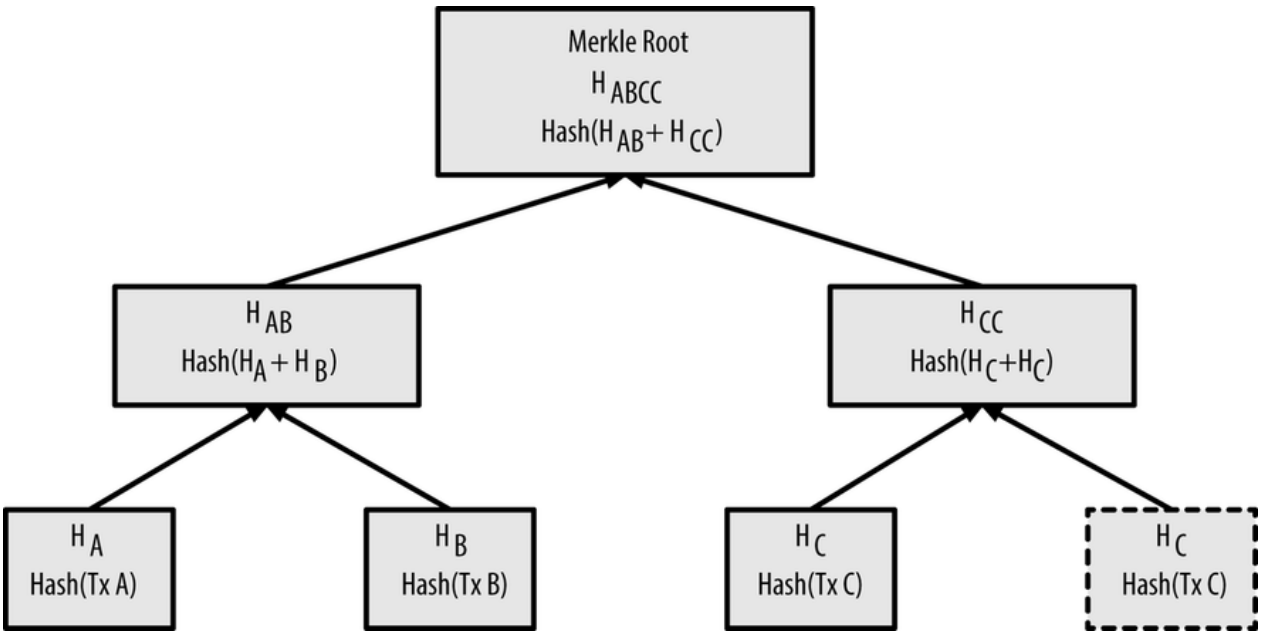
Merkle tree

Technical side



Дерево Меркла - это структура данных, также известная как бинарное дерево хэшей. В случае Bitcoin оно строится следующим образом:

1. Сначала считаются хэши всех транзакций в блоке `hash_A = SHA256(SHA256(A))`
2. Потом считаются хэши от суммы хэшей транзакций `hash_AB = SHA256(SHA256(hash_A + hash_B))`
3. Точно также считаем хэши от суммы получившихся хэшей `hash_ABCD = SHA256(SHA256(hash_AB + hash_CD))` и далее по рекурсии. Лирическое отступление - так как дерево бинарное, то на кажом шаге должно быть четное число элементов. Поэтому если, например, у нас только три транзакции, то последняя транзакция просто дублируется:



4. Процесс продолжается до тех пор, пока не получится один единственный хэш - он и называется *merkle_root* (третье поле в *header* блока)

Ниже приведена реализация дерева Меркла, можете проверить ее на каком-нибудь блоке.

```
import hashlib

# Hash pairs of items recursively until a single value is obtained
def merkle(hashList):
    if len(hashList) == 1:
        return hashList[0]
    newHashList = []
    # Process pairs. For odd length, the last is skipped
    for i in range(0, len(hashList)-1, 2):
        newHashList.append(hash2(hashList[i], hashList[i+1]))
    if len(hashList) % 2 == 1: # odd, hash last item twice
        newHashList.append(hash2(hashList[-1], hashList[-1]))
    return merkle(newHashList)

def hash2(a, b):
    # Reverse inputs before and after hashing
    # due to big-endian / little-endian nonsense
    a1 = a.decode('hex')[::-1]
    b1 = b.decode('hex')[::-1]
    h = hashlib.sha256(hashlib.sha256(a1 + b1).digest()).digest()
    return h[::-1].encode('hex')
```

Immutability

Теперь о том, зачем это нужно в Bitcoin. Я думаю, вы понимаете, что если изменить хотя бы одну транзакцию, то *merkle_root* также изменится. Поэтому такая структура данных позволяет обеспечить "неподделываемость" транзакций в блоке. То есть не может произойти следующей ситуации:

> Кто-то из майнеров нашел новый блок и начал раскидывать его по сети. В это время злоумышленник перехватывает блок и, например, удаляет из блока какую-нибудь транзакцию, после чего распространяет уже измененный блок.

Для проверки достаточно посчитать *merkle_root* самостоятельно и сравнить его с тем, что записан в *header* блока.

SPV

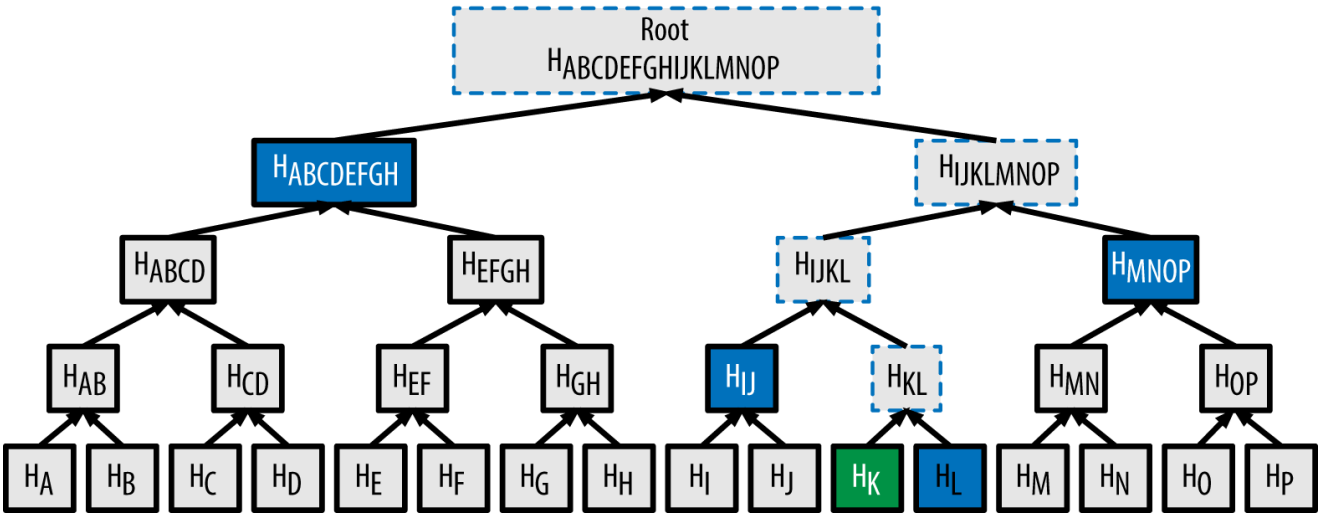
Но здесь можно резонно возразить, что, во-первых, такие сложности совершенно ни к чему. Достаточно просто посчитать хэш от суммы всех транзакций в блоке `txns_hash = SHA256(SHA256(sum(txns)))` - он точно также изменится после любых манипуляций с транзакциями. А, во-вторых, что мешает злоумышленнику подменить *merkle_root* в блоке? На второй вопрос отвечу сразу: на самом деле в блоке вообще нельзя ничего изменить, потому что блок тут же станет невалидным (это вы поймете после прочтения следующей главы Bitcoin in a nutshell - Mining).

А дерево Меркла нужно на самом деле для того, чтобы иметь возможность создавать *SPV* nodes (Simplified Payment Verification). Такие ноды синхронизируют только заголовки блоков, без самих транзакций. В результате блокчейн занимает на порядок меньше места (для красоты возьмем высоту в 500.000 блоков, размер header фиксирован - 80 байт):

> 500.000 * 80 / 1024 / 1024 ≈ 40 Мб

Такой блокчейн уже можно без проблем уместить на телефоне, планшете или каком-нибудь IoT. Что в перспективе должно дать большую децентрализацию, безопасность сети и так далее.

Суть *упрощенной верификации платежей* в следующем: пусть у вас есть *SPV* нода. У меня же есть весь блокчейн целиком и мне нужно вас убедить, что какая-нибудь транзакция действительно была (на картинке это транзакция *K*). В этом случае, мне достаточно всего лишь предоставить вам несколько хэшей: `H_L, H_IJ, H_MNOP, H_ABCDEFGH`, они еще называются *authentication path*.



После чего вы сначала считаете `H_K = SHA256(SHA256(K))`, потом `H_KL = SHA256(SHA256(H_K + H_L))` и так до самого верха. Если в итоге вы находите у себя блок с таким же *merkle_root*, то факт существования транзакции считается подтвержденным.

BTW Ральф Меркл даже запатентовал свою структуру данных, о чем свидетельствует патент US4309569 A.

Timestamp

Еще один интересный вопрос. Представим, что где-то в сети появился появился новый блок и ноды начинают передавать его друг-другу. Каждая нода должна убедиться в том, что блок корректен. Для этого она:

- проверяет синтаксис и структуру блока
- проверяет на валидность каждую транзакцию в блоке
- хэширует транзакции и сравнивает *merkle root*
- проверяет несколько критериев, связанных с майнингом, и так далее

Но как можно проверить timestamp? Понятно, что время на разных компьютерах может различаться, так что даже если у нового блока timestamp отличается от вашего текущего времени на час вперед, это еще не значит, что блок "неправильный", может у майнера просто часы спешат.

Поэтому для проверки timestamp на валидность было придумано два критерия. Во-первых, он должен быть больше, чем среднее арифметическое timestamp-ов предыдущих 11 блоков. Это делается для того, чтобы не получилось так, что блок #123 вышел 12 марта 2011 года, а #124 - 13 февраля 1984. Но в тоже время допускается некоторая погрешность.

Во-вторых, timestamp должен быть меньше чем *network adjusted time*. То есть нода, при получении нового блока, интересуется текущим временем у своих "соседей" по сети, считает среднее арифметическое и если block timestamp меньше получившегося значения + 2 часа, то все в порядке.

BTW как вы видите, timestamp нового блока может оказаться даже меньше, чем timestamp более раннего блока. Это не такая уж и редкость, например #145045, #145046 и #145047.

```
145044: 2011-09-12 15:46:39
145045: 2011-09-12 16:05:07
145046: 2011-09-12 16:00:05 // ~5 minutes before prior block
145047: 2011-09-12 15:53:36 // ~7 & ~12 minutes before 2 prior blocks
145048: 2011-09-12 16:04:06 // after 2 prior blocks but still before 145045
```

Raw block

Если у вас до сих остались какие-то вопросы по структуре блока, то предлагаю вам посмотреть на них в "сыром" виде. Самый очевидный способ это сделать - запустить на пару часов `bitcoind --daemon` , а потом исследовать уже скачанные блоки. Но, во-первых, не у всех есть время / желание синхронизировать блокчейн. Во-вторых, в Bitcoin блоки хранятся в крайне специфической базе данных LevelDB, еще и довольно странным образом. А так как книга расчитана не только на опытных разработчиков, то я пойду уже проверенным путем и снова использую протокол в его первозданном виде.

Для получения блока отправим сообщение getdata, в котором укажем `type : MSG_BLOCK` и `hash : 000000000003ba27aa200b1cecaad478d2b00432346c3f1f3986da1afd33e506` - это хэш блока #100.000. Весь код целиком можете посмотреть здесь.

```
def getdataMessage():
    block_hash = '000000000003ba27aa200b1cecaad478d2b00432346c3f1f3986da1afd33e506'

    count = struct.pack("<B", 1)
    inventory = struct.pack("<L", 2) # type : MSG_BLOCK
    inventory += block_hash.decode('hex')[::-1]

    return count + inventory
```

```
  16 8.116111  192.168.0.107  70.68.73.137   Bit…  115 getdata
  17 8.313963  70.68.73.137   192.168.0.107  Bit…  10… block
```

```
Bitcoin protocol
  Packet magic: 0xf9beb4d9
  Command name: block
  Payload Length: 957
  Payload checksum: 0x56324b0b
▾ Block message
    Block version: 1
    Previous block: 50120119172a610421a6c3011dd330d9df07b63616c2cc1f...
    Merkle root: 6657a9252aacd5c0b2940996ecff952228c3067cc38d4885...
    Block timestamp: Dec 29, 2010 14:57:43.000000000 Russia TZ 2 Standard Time
    Bits: 0x1b04864c
    Nonce: 0x10572b0f
    Number of transactions: 4
  ▾ Tx message [    1 ]
      Transaction version: 1
      Input Count: 1
    › Transaction input
      Output Count: 1
    › Transaction output
      Block lock time or block ID: 0
```

Links

- Mastering Bitcoin - The Blockchain
- Documentation of the physical Bitcoin blockchain
- What are the keys used in the blockchain levelDB
- bitcoin-core/leveldb/doc/table_format.txt
- Деревья Меркла в Эфириуме