区块与区块链

#「笔耕不辍」-生命不止，写作不息#

目录

一.区块头与区块体

区块是区块链的核心单元。区块链由区块互相连接而成。

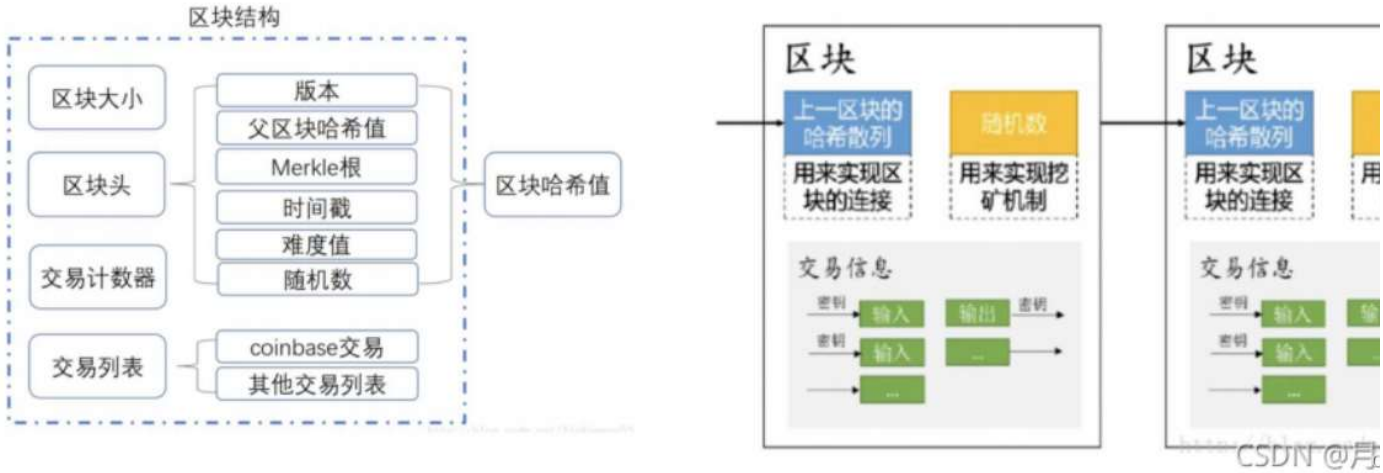

图1：来自网络

区块

区块由区块头和区块体两部分组成。其中区块的大小被限制在1M以内(为了防止资源浪费和DOS攻击)，区块头的大小被固定为80个字节。但目前随
展，交易数量持续增加，1M的大小能存储的交易数量有限，导致大量的交易积压。因此目前正在考虑扩容方案。

创世区块

比特币里的第一个区块创建于2009年，被称为创世区块。它是区块链里所有区块的共同祖先，这意味着你从任意区块，循链向后回溯，最终都会到这
为创世区块被编入到比特币客户端软件里，所以每一个节点都始于至少包含一个区块的区块链，这能确保创世区块不会被改变。每一个节点都"知道"
希值、结构、被创建的时间和里面的一个交易。因此，每一个节点都把该区块作为区块链的首区块，从而创建了一个安全的、可信的区块链的根。

区块头

区块头由三组区块元数据组成。第一组：引用父区块哈希值的数据，这组元数据用于该区块与区块链中前一区块相连接。第二组：难度、时间戳、no
竞争相关。第三组：merkle树根，一种用来有效地总结区块中所有交易的数据结构。一共80个字节。

区块哈希值可以唯一、明确地标识一个区块，任何节点通过简单地对区块头进行哈希计算都可以独立地获取该区块哈希值。

对于区块的哈希值，请注意：区块哈希值实际上并不包含在区块的数据结构里，不管是该区块在网络传输时，或者是它作为区块链的一部分被存储在
久性设备上时。相反，区块哈希值是当该区块从网络被接受时由每个节点计算出来的。区块的哈希值可能会作为区块元数据的一部分被存储在一个独
里，以便于索引和更快地从磁盘检索区块

字段	大小	描述
version	4字节	版本号，用于跟踪软件/协议的更新
prevBlockHash	32字节	上一个区块的Hash地址
merkleRoot	32字节	该区块中交易的merkle树根的哈希值
time	4字节	该区块的创建时间戳
difficultyTarget	4字节	该区块链工作量证明难度目标
nonce	4字节	用于证明工作量的计算参数

区块体

区块体中记录了该区块存储的交易数量以及交易数据。

平均每个交易至少是250个字节，平均每个区块至少包含超过500个交易信息。因此，一个包含所有交易的完整区块比区块头大1000倍不止。

字段	大小	描述
numTransactionsBytes	1字节	交易数量占用的字节数
numTransactions	0-8个字节	区块内存储的交易数量
transactions	不确定	区块内存的多个交易数据

字段	大小	描述
区块大小	4字节	用字节表示的该字段之后的区块大小
区块头	80字节	组成区块头的几个字段
1-9（可变整数）	交易计数器	交易的数量
交易	可变的	记录在区块里的交易数量

为了节约区块的存储空间，区块内的交易数量字段采用了压缩存储。在读取交易数量之前，会先读取numTransactionsBytes字段值。
- 如果该值小于253，则用直接将该值作为交易数量
- 如果该值等于253，则读取之后的两个字节作为交易数量
- 如果该值等于254，则读取之后的4个字节作为交易数量
- 否则，读取之后的8个字节作为交易数量

例子

比特币

图2：来自网络

区块链如何连接成区块链?

让我们假设：当前区块链有277314个区块，最后一个区块为第277314个区块，这个区块的区块头哈希值为：
00000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249，然后从网络上接收到一个新的区块如下：

```
{
    "size":43560,
    "version":2,
    "previousblockhash":"00000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1
    "merkleroot":"5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d"
    "time":1388185038,
    "difficulty":1180923195.25802612,
    "nonce":4215469401,
    "tx":["257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",
#[...many more transactions omitted...]
        "05cfd38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"
        ]
}
```

图3：来自网络

对于这一新的区块，接收者会在“父区块哈希值”字段里找出包含它的父区块的哈希值。这是接收者已知的哈希值，也就是第277314块区块的哈希值，是这个链条里的最后一个区块的子区块，因此现有的区块链得以扩展。接收者将新的区块添加至链条的尾端，使得区块链变长到一个新的高度27731

图4：来自网络

区块的核心代码

1. 区块中的核心常量定义

```
/** How many bytes are required to represent a block header WITHOUT the trailing 00 length byte. */
    //区块头的大小，当前为80个字节
    public static final int HEADER_SIZE = 80;

    static final long ALLOWED_TIME_DRIFT = 2 * 60 * 60; // Same value as Bitcoin Core.

    /**
     * A constant shared by the entire network: how large in bytes a block is allowed to be. One day we may have to
     * upgrade everyone to change this, so Bitcoin can continue to grow. For now it exists as an anti-DoS measure to
     * avoid somebody creating a titanically huge but valid block and forcing everyone to download/store it forever.
     */
    //全网共享的常量，用于表示区块的最大字节数。随着比特币会持续的发展，日后升级网络时可能会变更该数字。
    //目前该值作为解决拒接攻击的一种措施，避免有人创建巨量的区块，造成整个网络的资源浪费。
    public static final int MAX_BLOCK_SIZE = 1 * 1000 * 1000;
    /**
     * A "sigop" is a signature verification operation. Because they're expensive we also impose a separate limit on
     * the number in a block to prevent somebody mining a huge block that has way more sigops than normal, so is very
     * expensive/slow to verify.
     */
    //sigop是签名校验操作，因此这个操作需要大量的资源，因此需要限制大小，防止资源浪费或降低网络性能
    public static final int MAX_BL
```

```
    /** A value for difficultyTarget (nBits) that allows half of all possible hash solutions. Used in unit testing. *
//工作量的难度目标25     public static final long EASIEST_DIFFICULTY_TARGET = 0x207fFFFFL;
```

2. 区块中的核心变量定义

```
private long version;               //区块链的版本号
    private Sha256Hash prevBlockHash;   //前一个区块的hash地址
    private Sha256Hash merkleRoot;      //交易标识的merkle根
    private long time;                  //区块创建时间戳
    private long difficultyTarget; // "nBits"   //区块工作难度目标
    private long nonce;                 //用于证明区块工作量的参数

    // TODO: Get rid of all the direct accesses to this field. It's a long-since unnecessary holdover from the Dalvik
    /** If null, it means this object holds only the headers. */
    //区块中存储的交易数据
    @Nullable List<Transaction> transactions;

    /** Stores the hash of the block. If null, getHash() will recalculate it. */
    //当前区块的hash地址
    private Sha256Hash hash;
```

3. 解析区块二进制数据

```
//从原始字节数据中构造区块对象
    @Override
    protected void parse() throws ProtocolException {
        // header
        cursor = offset;
        version = readUint32();     //读取4个字节的版本号
        prevBlockHash = readHash(); //读取前一个区块的hash地址
        merkleRoot = readHash();    //读取merkle交易树的根值
        time = readUint32();        //读取区块的创建时间戳
        difficultyTarget = readUint32();    //读取区块的难度目标
        nonce = readUint32();               //读取区块用于计算难度的随机数
        //通过区块头计算当前区块的hash地址
        hash = Sha256Hash.wrapReversed(Sha256Hash.hashTwice(payload, offset, cursor - offset));
        headerBytesValid = serializer.isParseRetainMode();  //是否缓存区块的hash地址

        // transactions
        //解析区块内的交易数据
        parseTransactions(offset + HEADER_SIZE);

        //计算区块的字节数
        length = cursor - offset;
    }
```

4. 解析区块内的交易数据

```
//解析区块内的交易数据
    protected void parseTransactions(final int transactionsOffset) throws ProtocolException {
        cursor = transactionsOffset;            //设置读取数据的起始偏移地址

        optimalEncodingMessageSize = HEADER_SIZE;   //初始化编码后的区块大小
        if (payload.length == cursor) {
            // This message is just a header, it has no transactions.
            transactionBytesValid = false;
            return;
        }

        int numTransactions = (int) readVarInt();   //获取区块内的交易数据量

        //累加编码后的区块大小，不同的整数经过编码后，占用的存储空间不一样，因此需要通过VarInt进行计算
        optimalEncodingMessageSize += VarInt.sizeOf(numTransactions);

        transactions = new ArrayLis
```

```
        //逐一构造区块内的交易数据
            //构造区块内的交易数据
            Transaction tx = new Transaction(params, payload, cursor, this, serializer, UNKNOWN_LENGTH);
            // Label the transaction as coming from the P2P network, so code that cares where we first saw it knows.
            tx.getConfidence().setSource(TransactionConfidence.Source.NETWORK);
            transactions.add(tx);
            cursor += tx.getMessageSize();
            optimalEncodingMessageSize += tx.getOptimalEncodingMessageSize();
        }
        transactionBytesValid = serializer.isParseRetainMode();
    }
```

```
        for (int i = 0; i < numTransactions; i++) {
```

二.区块链数据结构

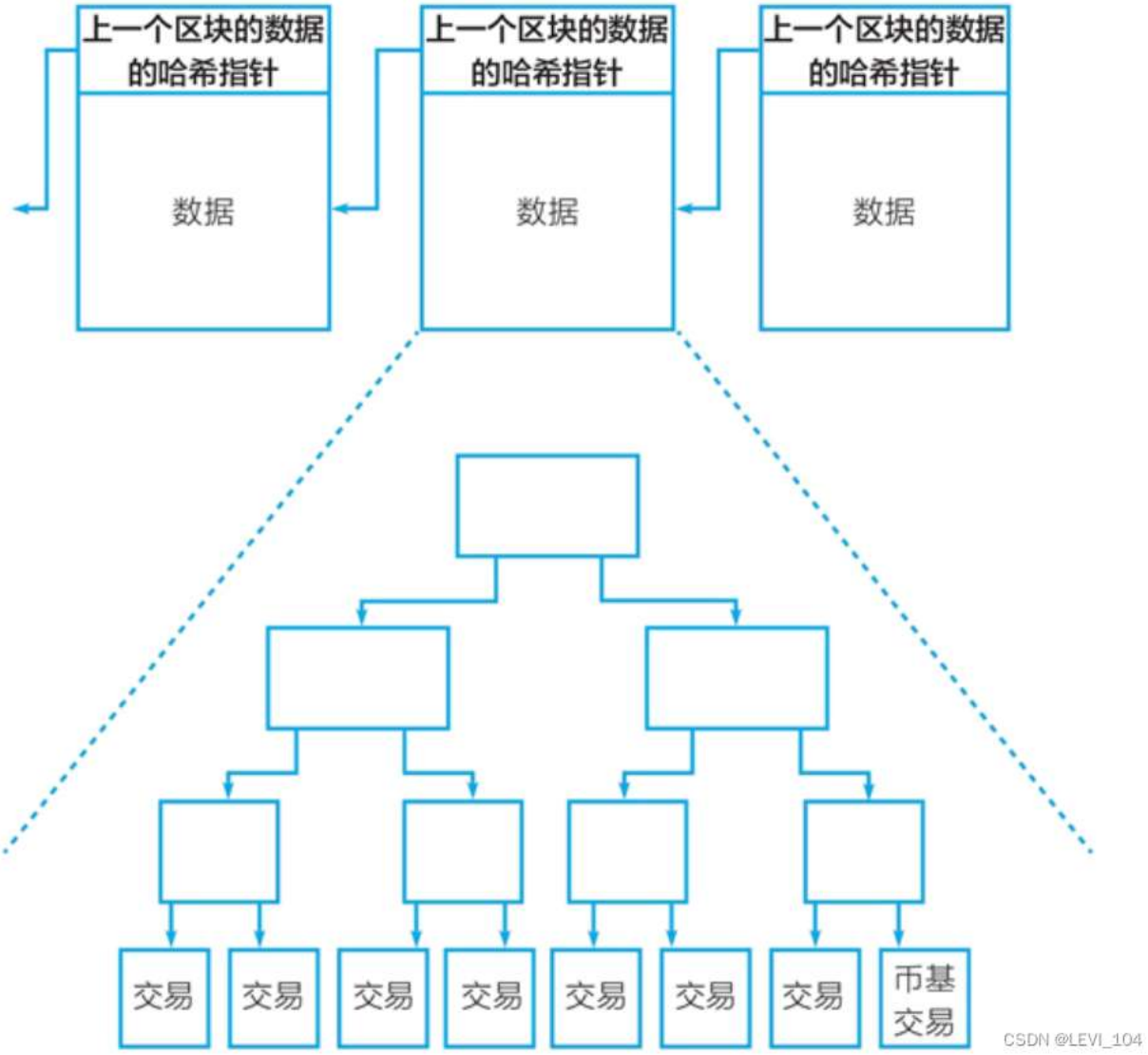

图0：来自网络

存证

存证，就是保存证据，保存在哪里？保存在区块链。

第一种上链方式

数据如何保存在区块链，目前主要的方式是“哈希上链”，我们也可以称之为**间接存证技术**。简要说这种方式的原理是这样的：

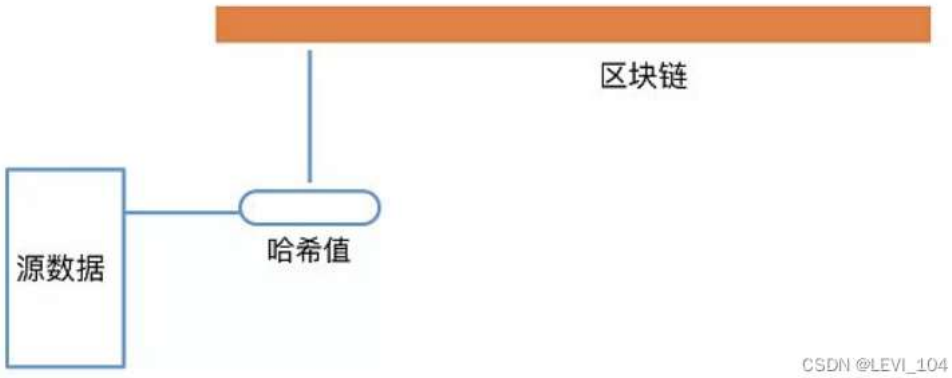

图1：来自区块链网

如图所示，并不是把真实的数据写入区块链，而是将源数据做摘要，讲哈希值写入了区块链

哈希值是可以根据需要定义固定长度，比如64位，256位，160位等。为什么不直接将源数据上链，而将哈希值上链，是因为两个主要原因：源数据可
如是一个监控视频，G级别；区块链的区块大小一般都还在1-10M这种级别。

简言之，源数据是存在另一个地方的，但是可以通过哈希算法的计算结果，来验证源数据的哈希值和保存在区块链上的哈希值一致，从而说明源数据
比如有人盗版了你的小视频，至少有两种方式保存证据，一种是去公证处做证据保存。一种是你可以录下来，将录像文件的哈希值上链，在互联网法
据。

但这种方式有一个问题，就是源数据是可能丢失的。只有一个哈希值存在链上，证明不了什么。目前技术上还不太可能通过一个哈希值还原出源数据
布式存储技术，以一种类区块链的方式帮助存储大文件。

第二种上链方式

还源数据直接上链：这种方式比较适用于文件内容比较小，多数为一些文字内容等。目前还是有一些技术门槛。比如说车辆闯红灯这件事，摄像头拍
就可以上链。这类的应用场景会比较多。

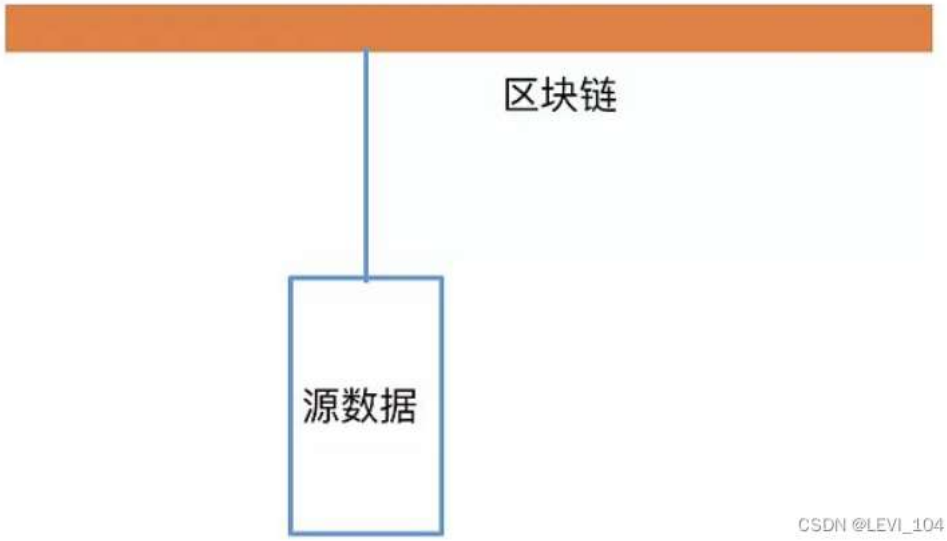

图2：来自区块链网

这类上链存证技术的优点是，链上数据很难篡改，公信力比较高。缺点也很明显，数据容量问题，效率问题，存储成本----尤其未来存储成本的不可预
区块链的区块文件大小，一般都是200G左右规模，T级区块链正在出现。但是这个数据级别，在今天的大数据时代，根本就是太小了。

这就引出了一个问题，在这样的一个数据结构中，究竟什么样的数据应该在区块链存证？有待我们去探究

Hash指针

hash指针保存了结构体的指针和hash值，能够用于找到结构体的位置，并且验证结构体是否被篡改。区块链与普通链表的区别在于，将普通指针替换
针。区块链中第一个区块被称为创世纪块，最后一个区块被成为most recent block。每个区块都有一个hash指针。hash指针的hash值使用前面区块的
括hash指针）取hash值。只要保存最后一个区块的hash值，就可以确定区块链中是否被篡改。

比特币区块链的数据结构中包括两种哈希指针，它们均是不可篡改特性的数据结构基础。一个是形成“区块+链”（block+chain）的链状数据结构，另一
针形成的梅克尔树（见图 0）。链状数据结构使得对某一区块内的数据的修改很容易被发现；梅克尔树的结构起类似作用，使得对其中的任何交易数
易被发现。

默克尔树

看我的以前的文章：Merkle树_LEVI_104的博客-CSDN博客

数据区块：区块头&区块体

见本文中第一部分

三.默克尔树

看我的以前的文章：Merkle树_LEVI_104的博客-CSDN博客