


区块链技术 - 比特币系统（C++实现）

 还有好几点能优化：

1. 内存空间的管理，可以使用智能指针
2. 钱包支付逻辑优化，比如HD分层钱包
3. 复杂支付情况下，交易生成和区块生成，并验证全局的mempool和UTXO集合是否正常，符合逻辑

实验1：Merkle Tree

关键知识点

- 交易并不存储在默克尔树中，而是将数据哈希化，然后将哈希值存储至相应的叶子节点。
- 在比特币网络中，默克尔树被用来归纳一个区块中的所有交易，生成整个交易集合的数字指纹，且提供了一种验证某交易是否存在于某个区块的高效方法。生成一棵完整的默克尔树需要递归地对每一对节点进行哈希，直到只剩一个哈希值，它就是根root，或者默克尔树根Merkle root。在比特币的默克尔树中使用的加密散列算法SHA256被应用了两次，因此也被称为双哈希double-SHA256。

设计与实现

- 默克尔树的数据结构设计

C++

```
1  // 这里只展示基本成员变量和函数，具体实现参考附件 blockchain目录 -> MerkleTree类的实现
2  class MerkleTree {
3  public:
4      std::string hash;           // 节点的hash值
5      MerkleTree* leftChild;      // 左子树
6      MerkleTree* rightChild;     // 右子树
7  private:
8      // 利用队列这一数据结构，来实现MerkleTree的构造
9      static MerkleTree* MakeMerkleTree(std::queue<std::string> q);
10 };
```

- MakeMerkleTree的实现思路正如上面“关键知识点”介绍的那样，递归地对节点进行哈希，但是其中一个细节在《精通比特币》里没有讲清楚，就是当交易数不是偶数个数时具体是如何进行补齐的呢？

- 可以参考这篇文章：[比特币中MerkleTree默克尔树的构造 - 深蓝 - 博客园](#)
 - 特别地，当区块里只有唯一一笔coinbase交易时，coinbase的hash就是默克尔根的hash值
- 至此，可以给出方法的核心实现：
1. 初始输入为一个字符串队列（利用队列先进先出的特性，可以很好地还原层层向上递归哈希的实现）
 2. 遍历该字符串队列，依次把哈希值构造成一个个叶子节点，再依次放入队列
 3. 计数当前的队列中元素个数，如果为奇数，进行标记，用于当需要哈希该层最后一个元素时，对最后的元素进行复制
 4. 对该层元素进行循环操作，每次拿出队列的前两个元素，进行两次SHA256，然后将得到的哈希值构造成节点再放入队列
 5. 当队列元素个数只有一个时，结束循环，最终的元素就是MerkleTree Root

验证正确性

使用比特币区块链上的真实数据对算法进行验证，使用了如下两个测试用例：

- [Bitcoin - 有6个交易的区块](#)
- [Bitcoin - 有9个交易的区块](#)

对应测试代码见附件test目录 -> `TestMerkleTreeSixTx`、`TestMerkleTreeNineTx` 两个函数

测试结果如下图所示：

```
默克尔树构造验证（6个交易构造测试）：开始！
目标值：A5C67F0B44C1A342DEC9135A566FD93F0C728452D1DBF850F4F9A47AF82AED77
实际值：A5C67F0B44C1A342DEC9135A566FD93F0C728452D1DBF850F4F9A47AF82AED77
默克尔树构造验证（6个交易构造测试）：成功！
-----
默克尔树构造验证（9个交易构造测试）：开始！
目标值：2FDA58E5959B0EE53C5253DA9B9F3C0C739422AE04946966991CF55895287552
实际值：2FDA58E5959B0EE53C5253DA9B9F3C0C739422AE04946966991CF55895287552
默克尔树构造验证（9个交易构造测试）：成功！
-----
```

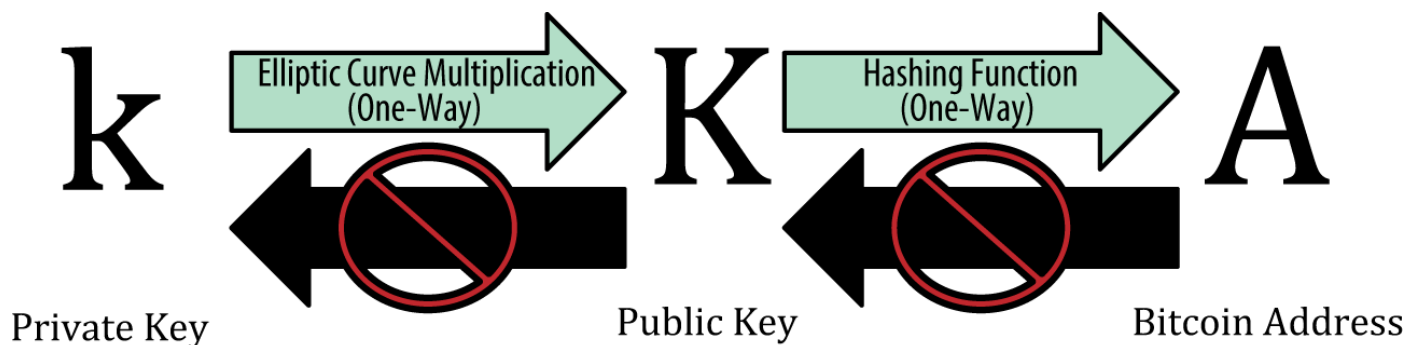
！ 在实现和验证时，要特别注意网页上展示的哈希值或者地址相关的16进制码，都是以大端形式展示的，而在程序执行时，要按照小端进行哈希/加密等操作，所以验证时，需要先将交易哈希转换字节序；

因此我在程序中，都会以小端模式保存数据，直到需要输出在控制台上时才会进行转换

实验2：比特币账户

关键知识点

- 准确的说，比特币里并没有账户这一概念，通常我们是使用钱包来进行交易，而一个钱包实际上就是管理一系列密钥对，每一对密钥对应着一个比特币地址，可以接收和花费比特币，这也正好对应比特币中交易采用UTXO的方式
- 钱包中，私钥、公钥和地址的关系可以参考这篇文章：[比特币私钥，公钥和地址的关系](#)
- 私钥实际上就是一个随机生成的256位二进制数（32字节大小），公钥是通过椭圆曲线上的计算得到的，而地址是通过对公钥执行SHA256和RIPEMD160哈希得到的160位二进制数（20字节大小）。私钥到公钥到地址是可以连续推导出来的，但是反过来不行，正如下图描述



设计与实现

- 将实验中账户的概念，对应到程序中WalletKeys类

```

1  // 具体实现参考附件 blockchain目录 -> WalletKeys类的实现
2  class WalletKeys {
3      ...
4  public:
5      // 获取字节形式的密钥
6      std::string GetRawPrivateKey();
7      // 获取Hex编码形式的密钥
8      std::string GetHexPrivateKey();
9      // 获取WIF格式的私钥
10     std::string GetWIFPrivateKey();
11     // 获取WIF-Compressed 格式的私钥
12     std::string GetWIFCompressedPrivateKey();
13
14     // 获取字节形式公钥（64字节）
15     std::string GetRawPublicKey();
16     // 获取Hex编码格式公钥（带上前缀04）
17     std::string GetHexPublicKey();
18     // 获取压缩格式公钥（前缀02或03）
19     std::string GetCompressedHexPublicKey();
20
21     // 获取使用非压缩公钥产生的比特币地址（字节形式）
22     std::string GetRawAddress();
23     // 获取使用非压缩公钥产生的比特币地址（Base58check 编码）
24     std::string GetBase58CheckAddress();
25     // 获取使用压缩公钥产生的比特币地址（字节形式）
26     std::string GetCompressedRawAddress();
27     // 获取使用压缩公钥产生的比特币地址（Base58check 编码）
28     std::string GetCompressedBase58CheckAddress();
29
30 };

```

接下来对具体实现进行说明：

- 首先是私钥，参考《精通比特币里》的说明，私钥的格式有4四种，如下表所示（Prefix指的是使用Base58check编码后的前缀）

	A	B	C
1	Type	Prefix	Description
2	Raw	None	32个字节
3	Hex	None	64位的Hex编码
4	WIF	5	使用Base58check编码: 需要加上0x80的版本前缀和32位的checksum再进行Base58编码
5	WIF-compressed	K or L	和WIF类似，但是除了需要加上0x80的版本前缀，还要加上0x01的压缩后缀（表示对应产生压缩公钥），然后再得到checksum，最后进行Base58编码

- 然后是公钥，大致有以下3种格式

	A	B	C
1	Type	Bytes	Description
2	Raw	64字节	公钥就是椭圆曲线上点的X和Y坐标的拼接，各占32字节（更准确地说，实际使用中还有一个前缀，指示是否为压缩，不过在程序里我定义的Raw类型是不带前缀的）
3	Hex	65字节	未压缩的公钥，需要加上0x04的前缀，指示该公钥未压缩
4	Compressed Hex	33字节	压缩格式的公钥，根据椭圆曲线的特性，只要知道X坐标就能计算出Y坐标，所以有了这种压缩格式。如果Y坐标为偶数，增加0x02前缀，如果Y坐标为奇数，增加0x03前缀。最后就是前缀 + X坐标

- 最后是地址，对应同一个密钥对，使用未压缩公钥和压缩公钥会产生两个不同的比特币地址，但实际上对应的是同一个私钥。然后地址的生成方式如下所述：

1. 公钥（压缩或未压缩）先后经过SHA256、RIPEMD160 两次哈希
2. 将得到的Hash值加上前缀0x00 进行Base58check 编码
3. 最后得到的比特币地址的前缀是1

验证正确性

验证算法的准确性，对比《精通比特币》中这一小结的输出：[用Python实现密钥和比特币地址](#)

使用上述指定的私钥

3ABA4162C7251C891207B747840551A71939B0DE081F85C4E44CF7C13E41DAA6，生成公私钥，然后比对程序输出和实际输出，对应测试代码见附件test目录 -> TestWalletKeys 个函数
其中比特币地址的测试结果如下图所示：

```
-----
Test Bitcoin Address (b58check) is:
1thMirt546nngXqyPEz532S8fLwbozud8
Real Bitcoin Address (b58check) is:
1thMirt546nngXqyPEz532S8fLwbozud8
Bitcoin Address (b58check) 测试: 通过!
-----

Test Compressed Bitcoin Address (b58check) is:
14cxpo3MBCYYWCgF74SWTdcmxipnGUsPw3
Real Compressed Bitcoin Address (b58check) is:
14cxpo3MBCYYWCgF74SWTdcmxipnGUsPw3
Compressed Bitcoin Address (b58check) 测试: 通过!
-----
```

实验3：签名与验证

消息签名与验证的实现主要就是使用CryptoPP第三方库提供的函数来实现，因为每次生成的签名都不是不同的，因此也没有办法找到一个参照值进行验证，算法的准确性取决于库的实现，因此这里不做过多说明。下图是测试结果展示，对应测试代码见附件test目录 -> `TestSignAndVerify` 个函数

```
-----
签名与验证测试: 开始!
message :blockchain-ss-2021
signature (Hex) :05A8CFD479A79B2668BEFB8AA9D82CA0313EA4CEC50F50291E6D0A2848A6F4D5C8
Verify result :true
签名与验证测试: 通过!
-----
```

实验4：交易

关键知识点

- 交易包含的数据内容，主要有4个，可以参考 [交易数据内容](#)
 - 版本号
 - 锁定时间 (locktime)：在一定时间内交易不生效
 - 交易的输入：指定花费哪些UTXO，其结构字段如下
 - 一个交易ID，引用包含将要消费的UTXO的交易
 - 一个输出索引 (vout)，用于标识来自该交易的哪个UTXO被引用（第一个为零）
 - 一个 scriptSig（解锁脚本），满足UTXO的消费条件，解锁用于支出
 - 一个序列号 (sequence)，跟locktime的效果类似
 - 交易的输出：指定输出到哪些比特币地址上，其字段结构如下

- 一定量的比特币 (value), 面值为“聪” (satoshis) , 是最小的比特币单位;
- 确定花费输出所需条件的加密难题 (cryptographic puzzle) , 这个加密难题也被称为锁定脚本(locking script), 见证脚本(witness script), 或脚本公钥 (scriptPubKey)。
- 应注意的是TXID, 严格来说并不是交易数据的一部分, 可参考 <https://wiki.bitcoinsv.io/index.php/TXID>
- **交易哈希生成**, 也就是交易的TXID, 简单来说, 就是对签名后的Raw Transaction进行序列化, 然后对序列化后的字节进行两次SHA256哈希得到 TXID
- **交易的序列化**, 之前说道TXID生成, 依赖于交易的序列化, 交易的序列化格式如下图所示

version		01 00 00 00
input count		01
input	previous output hash (reversed)	48 4d 40 d4 5b 9e a0 d6 52 fc a8 25 8a b7 ca a4 25 41 eb 52 97 58 57 f9 6f b5 0c d7 32 c8 b4 81
	previous output index	00 00 00 00
	script length	8a
	scriptSig	47 30 44 02 20 2c b2 65 bf 10 70 7b f4 93 46 c3 51 5d d3 d1 6f c4 54 61 8c 58 ec 0a 0f f4 48 a6 76 c5 4f f7 13 02 20 6c 66 24 d7 62 a1 fc ef 46 18 28 4e ad 8f 08 67 8a c0 5b 13 c8 42 35 f1 65 4e 6a d1 68 23 3e 82 01 41 04 14 e3 01 b2 32 8f 17 44 2c 0b 83 10 d7 87 bf 3d 8a 40 4c fb d0 70 4f 13 5b 6a d4 b2 d3 ee 75 13 10 f9 81 92 6e 53 a6 e8 c3 9b d7 d3 fe fd 57 6c 54 3c ce 49 3c ba c0 63 88 f2 65 1d 1a ac bf cd
	sequence	ff ff ff ff
output count		01
output	value	62 64 01 00 00 00 00 00
	script length	19
	scriptPubKey	76 a9 14 c8 e9 09 96 c7 c6 08 0e e0 62 84 60 0c 68 4e d9 04 d1 4c 5c 88 ac
block lock time		00 00 00 00

重点关注交易输入和输出的序列化, 具体实现过程可以参考: [Exploring Bitcoin: signing the P2PKH input](#)

- 首先来看交易输出的序列化, 其序列化格式如下表所示

	A	B	C
1	Size	Field	Description
2	8 bytes (little-endian)	Amount	Bitcoin value in satoshis
3	1–9 bytes (VarInt)	Locking-Script Size	Locking-Script length in bytes, to follow
4	Variable	Locking-Script	A script defining the conditions needed to spend

拿一个实际例子进行说明, 对应的锁定脚本如下:

Shell	
1	OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG

该锁定脚本对应的序列化值为：**1976a914f86f0bc0a2232970ccdf4569815db500f126836188ac**

- 第一个字节 19：其实不应该算在锁定脚本里，这是用来指示锁定脚本的字节长度
 - 真正的脚本编码内容为：**76a914f86f0bc0a2232970ccdf4569815db500f126836188ac**
- 第二个字节 76：对应OP_DUP
- 第三个字节 a9：对应OP_HASH160
- 第四个字节 14：官网里没有明确支出，但我发现这是个定值，应该就是用来表示公钥地址的字节长度，刚好是20字节（RIPEMD160的输出长度）
- 接下来20个字节：对应 公钥哈希也就是比特币地址
- 倒数第二个字节 88：对应OP_EQUALVERIFY
- 倒数第一个字节 ac：对应OP_CHECKSIG

脚本操作符具体的编码数值可以参考：[脚本操作符对应编码](#)

交易输入的序列化，详细格式如下表所示

	A	B	C
1	Size	Field	Description
2	32 bytes	Transaction Hash	Pointer to the transaction containing the UTXO to be spent
3	4 bytes	Output Index	The index number of the UTXO to be spent; first one is
4	1-9 bytes (Var	Unlocking-Script Size	Unlocking-Script length in bytes, to follow
5	Variable	Unlocking-Script	A script that fulfills the conditions of the UTXO locking script
6	4 bytes	Sequence Number	Used for locktime or disabled (0xFFFFFFFF)

- 其中再重点来看如何生成解锁脚本，也就是如何对输入进行签名，参考：[比特币中对交易进行签名的详细过程 - 深蓝 - 博客园](#)

当采用ALL SIGHASH类型时， 对一个input输入进行签名的过程如下

1. 查找该Input交易对应的UTXO
2. 获得该UTXO对应的锁定脚本
3. 复制该交易对象（指的是现在构造的这个交易），并在复制的交易副本中将该Input的解锁脚本字段的值设置为对应output的锁定脚本（也就是第2步中得到的锁定脚本）
4. 清除其他Input的解锁脚本字段

- 对这个改造后的交易对象进行序列化，然后对序列化后的字节计算Hash（使用两次sha256）
- 使用该Input对应的私钥对Hash进行签名，得到R值和S值（对应于ECDSA中的签名）。
- 签名用DER编码后，后缀一个字节表示SIGHASH类型（如0x01表示ALL）

设计与实现

- 交易相关的类，主要设计了 `Transaction`、`TxIn`、`TxOut` 四个类，数据结构如下所示

C++

```
1 // 只展示数据字段，具体方法可以参考代码blockchain/transaction目录
2 class Transaction {
3 public:
4     unsigned int version;
5     std::vector<TxIn> vin;
6     std::vector<TxOut> vout;
7     unsigned int lockTime;
8 };
9
10
11 class TxIn {
12 public:
13     unsigned int sequence;
14     std::string prevOutHash; // 对应UTXO所在交易的hash, hex格式, 注意在这里
                             // 还是以小端形式保存, 只有输出展示时需要转换字节序
15     unsigned int prevOutIndex; // 对应UTXO所在交易 vout 的下标
16     std::string scriptSig; // 解锁脚本
17 };
18
19
20 class TxOut {
21 public:
22     // value 使用8个字节来存储
23     unsigned long long value;
24     // 锁定脚本
25     std::string scriptPubKey;
26 };
```

- 此外为了更加真实地模拟交易的生成，设计了Wallet钱包相关类（`WalletKeys`、`SimpleWallet`）和 `UTXO` 类，以及全局数据的管理类 `DataBase`
 - 简单来说 `WalletKeys` 就是钱包密钥，使用ECDSA椭圆曲线一类加密算法，生成密钥对，可以进行签名和验证；同时提供计算比特币地址（压缩和未压缩格式）等功能
 - `SimpleWallet` 就是对密钥进行管理，提供更高级的功能，主要是提供比特币地址、查看余额、以及转账来构造交易（本实验中就是依靠钱包的转账交易P2PKH，来生成交易）

- `UTXO` 主要用来帮助实现交易的构造，因为在模拟生成交易过程中，交易的输入在签名时，需要找到对应的UTXO获取锁定脚本
- `DataBase` 类主要是模拟节点的内存和数据库内容，包括维护历史区块数据，历史交易数据，内存池中待打包的交易，UTXO集合（更多说明可以参考「代码附件说明」）
- 考虑到篇幅问题，这里不再一一展示代码

接下来，重点说明程序中是如何生成交易的，对应 `SimpleWallet` -> `SendMoney` 函数方法

1. 根据函数的输入，计算此次转账的总费用，检查钱包中维护的UTXO集合，判断余额是否充足
2. 构造交易的输出TxOut，主要根据函数输入中的数值金额以及转账地址构造生成
3. 构造交易的输入TxIn，主要使用钱包维护的WalletKeys（相当于一个密钥对）集合，以及这些密钥对所对应的UTXO。支付策略采用简单的方式，即依次变量密钥集合，使用对应的UTXO进行支付，然后利用对应的私钥进行签名（签名的方法正如之前描述的那样）
4. 更新本地维护的UTXO，理论上待交易上链确认后，同步更新全局的UTXO
5. 填充交易的其他字段，交易生成完毕

验证正确性

- 主要设计了四个测试用例
 - `TestTransactionSerializeAndHash`：测试交易的序列化方法和交易的哈希生成，主要通过将交易数据赋值实际的交易数据，然后进行比对
 - `TestGenerateTransaction`：主要测试单个交易的生成，重点比对锁定脚本和解锁脚本，测试生成的也是真实的交易数据
 - `TestGenerateComplexTransaction`：测试生成一个复杂交易，即多个输入，多个输出；因为找不到给定私钥和签名的真实交易数据，所以只能验证格式上的确没有问题
 - `TestGenerate1000Tx`：使用钱包的转账功能，测试生成1000个交易

测试1: `TestTransactionSerializeAndHash` 交易的序列化方法和交易的哈希生成测试结果：

```
测试交易的序列化 与 交易TxID 哈希生成：开始！
Test Tx Hex:
0100000001A8FF0305E43645CF7BCFB6AE8DF6E121CB9FC43A55206AE22823B987AA28EDB800000006B48304
Real Tx Hex:
0100000001A8FF0305E43645CF7BCFB6AE8DF6E121CB9FC43A55206AE22823B987AA28EDB800000006B48304
交易序列化测试：成功！
-----
Test TXID:
D8C5C42CBD1DF7E48ACAB76FE05F2C9E612A20996FD37F4FFD4DC251385B6BA3
Real TXID:
D8C5C42CBD1DF7E48ACAB76FE05F2C9E612A20996FD37F4FFD4DC251385B6BA3
交易TXID测试：成功！
-----
测试交易的序列化 与 交易TxID 哈希生成：结束！
-----
```

测试2: `TestGenerateTransaction` 测试简单的单个交易生成，比对签名脚本，交易序列化和交易哈希

```
交易生成测试 (锁定脚本、交易Hash) : 开始!
Test sigScript:
483045022100e15a8ead9013d1de55e71f195c9dc613483f07c8a0692a2144ffa9050643687
Real sigScript:
483045022100e15a8ead9013d1de55e71f195c9dc613483f07c8a0692a2144ffa9050643687
锁定脚本测试: 成功!

-----

Test Transaction Hex:
0100000001449D45BBBF7FC93BBE649BB7B6106B248A15DA5DBD6FDC9BDFC7EFEDE83235E0
Real Transaction Hex:
0100000001449D45BBBF7FC93BBE649BB7B6106B248A15DA5DBD6FDC9BDFC7EFEDE83235E0
交易序列化 (Hex) 测试: 成功!

-----

Test Transaction Hash:
4484ec8b4801ada92fc4d9a90bb7d9336d02058e9547d027fa0a5fc9d2c9cc77
Real Transaction Hash:
4484ec8b4801ada92fc4d9a90bb7d9336d02058e9547d027fa0a5fc9d2c9cc77
交易Hash值 (Hex) 测试: 成功!

-----
```

测试3: `TestGenerateComplexTransaction` 测试复杂交易，多输入多输出，测试结果展示了生成交易的JSON格式

```
生成交易JSON格式:
{
  "inputs": [
    {
      "prevOutHash": "1dda832890f85288fec616ef1f4113c0c86b7bf36b560ea244fd8a6ed12ada52",
      "prevOutIndex": 1,
      "scriptSig": "493046022100BB730A2F1B3EC99A276355D2FD647997BD41E093547BD529616A213B2FDF8D69022100C6",
      "sequence": 4294967295
    },
    {
      "prevOutHash": "24f284aed2b9dbc19f0d435b1fe1ee3b3ddc763f28ca28bad798d22b6bea0c66",
      "prevOutIndex": 1,
      "scriptSig": "48304502201AFD2E6B1181BC6CC340F230FF41D1FD14FCD19C505E8E069CFBC439E86C838D022100BF3E",
      "sequence": 4294967295
    }
  ],
  "lockTime": 0,
  "outputs": [
    {
      "scriptPubKey": "76A914B5407CEC767317D41442AAB35BAD2712626E17CA88AC",
      "value": 29910240
    },
    {
      "scriptPubKey": "76A914BE09ABCFDA1F2C26899F062979AB0708731235A88AC",
      "value": 12000000
    }
  ],
  "version": 1
}
```

实验5：区块

关键知识点

- 区块是一种容器数据结构，用于聚合要上传到公共分类账簿（区块链）的交易。它由一个包含元数据的区块头和紧跟其后的占据区块最大空间的一长串交易列表组成。区块的序列化结构如下表所示：

	A	B	C
1	Size	Field	Description
2	4 bytes	Block Size	The size of the block, in bytes, following this f
3	80 bytes	Block Header	Several fields form the block header
4	1–9 bytes (VarInt)	Transaction Counter	How many transactions follow
5	Variable	Transactions	The transactions recorded in this block

- 交易列表相关的交易再上一节以及介绍过，这里重点介绍下区块头。区块头由三组区块元数据组成。首先，有一个对前一个区块哈希的引用即区块哈希值，它将此区块连接到区块链中的前一个区块。第二组元数据，即难度difficulty、时间戳timestamp和随机数nonce，与挖矿竞争相关。第三组元数据是默克尔树根，一种用来有效地汇总区块中所有交易的数据结构。

- 重点再介绍下区块哈希的生成，其实也就是挖矿的过程
 - 一个区块头包含的字段信息如下图所示，所有size加在一起正好是80字节，具体的对区块的Hash方式如下：

Field	Purpose	Updated when...	Size (Bytes)
Version	Block version number	You upgrade the software and it specifies a new version	4
hashPrevBlock	256-bit hash of the previous block header	A new block comes in	32
hashMerkleRoot	256-bit hash based on all of the transactions in the block	A transaction is accepted	32
Time	Current block timestamp as seconds since 1970–01–01T00:00 UTC	Every few seconds	4
Bits	Current target in compact format	The difficulty is adjusted	4
Nonce	32-bit number (starts at 0)	A hash is tried (increments)	4

- 对version、prevhash、merklehash、time、bits、nonce都进行Hex编码（一个字节分高四位、低四位编码成两个十六进制；对于数字而言，其实就相当于从二进制转换成十六进制字符串）
 - 这里要注意的是，需要按照小端模式进行Hex编码，即高字节位在高地址，低字节位在低地址；而比特币网站上通常展示的Block Hash或者是MerkleHash 使用大端模式来展示的（符合我们通常的认知）
- 然后将上述字段按顺序拼接在一起，对结果连续做两次SHA256 hash
- 然后需要再转换下字节序，把计算值变成大端显示，并与目标值进行比较，如果小于目标值，表示此次查找的nonce有效
 - 关于目标值的计算， 目标值 = 最大目标值 / 难度值；最大目标值为定值

0x00000000FF

F

，难度值会进行调整，作为比特币出块速度的调整参数（增大难度值，目标值减小，随机找到nonce的概率区间就越小，即难度越大）

- 区块头里字段Bits，用来计算目标值，Bits共4字节，用Hex编码表示就有8位，前2位代表指数，后6位表示系数，计算公式为

$$\text{目标值 (target)} = \text{系数} * 2^{(8 * (\text{指数} - 3))}$$

- 比如Bits为0x171320bc，那么指数为0x17，系数为0x1320bc，计算得到

```
0x1320bc000000000000000000000000000000000000000000000000000000000000
```

设计与实现

- 区块的字段设计参考实际区块数据：[实际区块字段参考](#)，设计的 Block 数据结构如下所示

C++

```

1  class Block {
2  public:
3      // header    区块头
4      unsigned int version;
5      std::string hashPrevBlock;
6      std::string hashMerkleRoot;
7      unsigned int time;
8      unsigned int bits;
9      unsigned int nonce;
10
11     // 网络传输时需要对交易进行序列化
12     std::vector<Transaction*> vtx;
13     // 由交易列表生成的MerkleTree
14     MerkleTree* merkleRoot;
15     // 区块Hash
16     std::string blockHash;
17     // 获取区块Hash (小端)
18     std::string GetBlockHash();
19     // 转换成JSON格式输出
20     std::string MarshalJson();
21 }

```

- 对于区块的生成，设计了 Miner 矿工类，Mine 方法用来打包交易生成区块，FindNonce 实际上就是模拟挖矿过程，找到符合条件的nonce值生成区块哈希

C++

```
1 // 矿工：模拟挖矿，产生区块
2 class Miner {
3 private:
4     // 根据区块头中的 Bits 计算目标值
5     std::string GetTarget(const std::string& bits);
6 public:
7     // 模拟挖矿，打包交易，生成区块
8     Block* Mine();
9     // 模拟挖矿过程，返回符合条件的nonce值
10    unsigned int FindNonce(const Block& block);
11    // 验证一个区块的哈希是否满足条件
12    bool proveBlock(const Block& block);
13 };
```

- 简单地描述下 `Mine` 函数的实现过程：
1. 从内存交易池中取出100个交易
 2. 对这100交易生成相应的TXID，利用实验1中的方法，生成一个MerkleTree，并为构造的区块赋值MerkleTree Root的哈希值
 3. 取出当前区块链上的最后一个区块，计算它的哈希值作为构造区块的prevBlockHash
 4. 填充区块的version, time, bits
 5. 执行FindNonce函数，找打符合条件的nonce值，生成区块哈希
 6. 整理内存中的交易池数据，更新UTXO集合列表

验证正确性

- 设计了两个测试用例
 - `TestMiner`：测试矿工挖矿的准确性，利用真实数据进行比对，参考：https://en.bitcoin.it/wiki/Block_hashing_algorithm
 - `Miner10Block`：测试生成10个区块，输出对应的目标值和BlockHash，以及区块的JSON格式，简略地判断区块是否正常生成

测试1: TestMiner 结果如下所示

测试矿工挖矿算法正确性：开始！

Target:

00000000000044B9F200

Block hash:

000000000000000001E8D6829A8A21ADC5D38D0A473B144B6765798E61F98BD1D

挖矿正确性验证测试结果: true

Miner10Block

挖出第1个区块

```
0000FFFF0000000000000000000000000000000000000000000000000000000000
```

000078CD957342E6777F58217BF3DC56F7B3E64F995DE877B02E631EB2CF2C5A

```
"bits": 520159231,
"hashMerkleRoot": "DD251E5ACECF59A59053D4F5D17C5A416E4A5388DD1AA13239C61F2DAA299E81",
"hashPrevBlock": "0000000000000000000000000000000000000000000000000000000000000000",
"nonce": 36411,
"time": 1635647715,
"txIds": [
    "99C6CF25C537CDCF6B5C977B24583B195FEDF37BD9E785A890FFBEABF32B6468",
    "FB9D95EA14323C0DB3BA745A87BC0FB93A5BA804A75352BC1AA787E188CBB8EC",
    "566952E082EFE36F5ABAD3A517BEDB03197C895357D50F47233638E2038AFC40",
    "455017EB8ACDC1E84816F9C65C31D01F49D99FCD511B03FC45AB02A8059F2B40",
    "C2AD496755729EA1847BB4037668DBE975DDFA188DE0CF3C5C8D59D6ACFBBE65",
    "2D4D50AAC0292FC6F4B35B45E8F86B88E664BDE8C2A039C9B6FEF9B1CC1CC380",
```

代码附件说明



1. 代码中使用的地址都采用小端模式进行表示，只有在测试输出展示时转换字节序
2. 交易实验中，比特币地址采用的是压缩地址

目录结构说明

- 根目录
 - main.cpp：程序入口，执行一系列测试函数
 - json.hh：第三方JSON库
 - util目录：主要实现util工具类，提供密码学中加密、哈希、编码等一系列操作
 - test目录：主要实现test类，设计了一系列测试函数
 - blockchain目录
 - block目录：存放Block类和Miner类，区块相关
 - transaction目录：存放Transaction类、TxIn类、TxOut类、UTXO类，交易相关
 - wallet目录：存放Wallet类、WalletKeys类、SimpleWallet类，钱包管理相关
 - crypto目录：CryptoPP第三方库

参考资料附录

比特币实现逻辑参考资料:

精通比特币中文电子书  《精通比特币》

[Signature verification in python using compressed public key](#)

[Exploring Bitcoin: signing the P2PKH input](#)

[比特币私钥，公钥和地址的关系](#)

[在线搜索交易记录、区块信息（各种数字货币）](#)