

比特币挖矿

挖矿命令	2
块结构	4
创建块	5
奖励比特币	6
SHA256 算法加密	8
检验块	9
工作量证明	10
计算算力	12
比特币网络	12

挖矿命令

比特币客户端内嵌了挖矿模块，可以使用相关的命令开始挖矿、获取挖矿参数。

有2个方式可以挖矿。

1、-gen

在配置文件(默认文件名bitcoin.conf)中添加此项，为1时开始挖矿，为0时停止挖矿。

客户端程序启动过程中，初始化(AppInit2)时获取此命令参数，进行挖矿。

2、setgenerate命令

命令格式是：setgenerate generate (genproclimit)。

generate为true时开始挖矿，false时停止挖矿，这个参数是必须有的。

genproclimit表示挖矿CPU个数，这个参数是可选的，默认是-1，无限制，所有的CPU都运行挖矿。为0时停止挖矿。

此命令最终会修改参数映射数组mapArgs中“-gen”项，从而决定后续是否进行挖矿。

如果是regtest模式，此参数表示需要挖出的块个数，直到挖出指定的块数量时退出。

获取挖矿命令是：getgenerate。

返回boolean值，true表示正在挖矿，false表示停止挖矿，默认是false。

生产比特币

生产比特币是在GenerateBitcoins函数。

在GenerateBitcoins函数中，创建、停止挖矿线程，挖矿线程主函数是BitcoinMiner。

创建指定数量的线程，但如果指定的线程数量为0，则停止挖矿。如果指定的线程数量小于0，则如果是测试网络，则只创建1个线程，否则创建的线程个数为CPU个数。

挖矿线程保存在线程组中(boost::thread_group类型)。

如果正在挖矿，即挖矿线程正在运行，则停止挖矿，即中断所有的挖矿线程，重新创建挖矿线程。

挖矿时，创建线程数组，创建所有的挖矿线程，每个线程绑定挖矿BitcoinMiner函数，绑定钱包。

挖矿线程

线程主函数为BitcoinMiner。

挖矿线程的优先级为THREAD_PRIORITY_LOWEST。

线程重命名为"bitcoin-miner"。

每个线程先创建一个基于钱包的KEY(CReserveKey类型)，然后开始循环挖矿。

如果是回归测试模式，则立刻开始挖矿；否则，每隔1秒检验一次网络，直到网络节点连接上才进行挖矿，如果网络节点没有连接，则只能浪费时间挖过期的块。

基于KEY创建新的块(CreateNewBlockWithKey)，保存在块模版中，增加线程自己的计数(IncrementExtraNonce)。

从堆栈中申请3块空间作为哈希缓冲区，16字节对齐，然后格式化哈希缓存区(FormatHashBuffers)。

缓冲区类型	大小(字节)
midstate	32+16
data	128+16
hash1	64+16

遍历缓冲区，用SHA256算法加密缓存区(ScanHash_CryptoPP)，如果找到一些块，则检验块是否有效(CheckWork)，检验之时，线程的优先级必须为THREAD_PRIORITY_NORMAL。统计算力。更新块的时间。

如果是回归测试，生成1个块后就停止挖矿。

如果挖矿过程中网络连接失败，则停止挖矿。

如果是测试网络，修改块的时间来修改需要的POW。

挖矿过程中会打印日志。

块结构

每个节点都会收集新的交易信息添加到块中，并且把交易信息保存到块的哈希树中，通过nonce值扫描块，使块的哈希值满足POW的要求。然后把块广播给每个节点，添加到块链中。块的第一个交易信息是创建了块的创建者拥有的新币的交易信息。

块结构主要涉及到CBlockHeader、CBlock、CBlockTemplate。

类CBlockHeader类包含了块头的基本信息，如：版本、时间戳等。

```
class CBlockHeader
{
public:
    static const int CURRENT_VERSION=2; // 当前版本，默认是2
    int nVersion;                        // 版本
    uint256 hashPrevBlock;               // 上一个块哈希值
    uint256 hashMerkleRoot;              // Merkle哈希树根节点
    unsigned int nTime;                   // 创建块时的时间
    unsigned int nBits;                   // 下一个工作需要的POW
    unsigned int nNonce;                  // Nonce值
};

    nVersion默认是2。
    nNonce用于扫描块，使块的hash满足POW的要求。
```

类CBlock继承了CBlockHeader类，增加了交易信息相关的数据。

```
class CBlock : public CBlockHeader
{
public:
    // network and disk
    std::vector<CTransaction> vtx;        // 交易信息数组

    // memory only
    mutable std::vector<uint256> vMerkleTree; // Merkle 树
};

    vtx包含了交易信息，用于网络传播、硬盘存储。
    vMerkleTree仅仅用于内存。
```

结构CBlockTemplate定义了块模版，增加了交易费、签名。

```
struct CBlockTemplate
{
    CBlock block;                        // 块
```

```

std::vector<int64_t> vTxFees;      // 交易手续费
std::vector<int64_t> vTxSigOps;    // 签名
};

```

创建块

如果是挖矿创建新块，则先需要从CReserveKey中获取公钥(CPubKey)，然后计算出公钥脚本(CScript)，再创建新块。

创建新块的函数是CreateNewBlock函数。

新建块模版CBlockTemplate实例。

新建空的coinbase交易信息作为块模版的交易信息的末尾项。把新建的交易信息添加到新建的块中，数组vTxFees、vTxSigOps添加-1值。如果指定了公钥脚本，则保存到交易信息的vout[0].scriptPubKey中。

计算创建的块的最大值，块最大值在1000与(MAX_BLOCK_SIZE(1000000)-1000)之间，如果指定了参数"-blockmaxsize"，则块最大值设置为指定值，否则设置为默认值DEFAULT_BLOCK_MAX_SIZE(750000)。

计算创建的块的最小值，块最小值在0与块最大值之间，如果指定了参数"-blockminsize"，则块最小值设置为指定值，否则设置为默认值0。

计算块优先级值，块优先级值决定了块中包含的高优先级交易信息的数量，不管支付的交易费是多少。块优先级值的最大值为块最大值，如果指定了参数"-blockprioritysize"，则设置为指定值，否则设置为默认值DEFAULT_BLOCK_PRIORITY_SIZE(50000)。

遍历内存池交易信息，组建优先级数组，不处理内存池中的CoinBase类型的交易信息、最后的交易信息。计算优先级、每千字节的手续费，添加到优先级数组中。(COrphan)

优先级计算公式：

$$\text{Priority} = \text{sum}(\text{valuein} * \text{age}) / \text{modified_txsize}$$

每千字节的手续费计算公式：

$$\text{dFeePerKb} = \text{double}(\text{nTotalIn-tx.GetValueOut()}) / (\text{double}(\text{nTxSize}) / 1000.0);$$

遍历优先级数组，把交易信息添加到新建的块中。

添加时注意以下几点：

- 1、不添加最高优先级的交易信息。
- 2、添加交易信息时，块大小总和不能超过块的最大值，块中的签名的总和不能超过最大值MAX_BLOCK_SIGOPS(20000)。

3、当块大小总和低于块最小值时，可以添加免费的交易信息，当块大小总和超过块最小值时，不能添加免费的交易信息了，免费的交易信息是指交易手续费低于最小值($dFeePerKb < CTransaction::nMinTxFee$)。

4、只添加在UTXO集中的交易信息(view.HaveInputs)。

5、不添加交易信息的接收信息无效的信息(CheckInputs)。

然后提交在UTXO集中的交易信息的结果(UpdateCoins)。把交易信息添加到CBlock的vtx中，计算交易费，添加到vTxFees中，计算签名，添加到vTxSigOps。如果porphan中含有相同的hash项，则从porphan中移除，添加到vecPriority中。

计算此块得到的比特币数量、手续费。

初始化块的成员信息。设置hashPrevBlock为上一个块的哈希索引。更新块时间nTime。设置nBits为下一个需要的POW。设置nNonce为0。设置vTxSigOps[0]为块的签名数量。

提交UTXO集上的指定索引的块的结果(ConnectBlock)。

奖励比特币

挖矿创建新区块后(CreateNewBlock)，会获取比特币奖励。

开始奖励50个比特币，每创建210000个区块后，奖励数量减半。创建210000个区块大概需要4年，从而确保到2140年时比特币总量约为2100万个(20999999.9769个)，比特币的奖励从0.00000001BTC降为0。

计算比特币奖励是在GetBlockValue函数中。

计算公式为：

$nValue = (50 * COIN) >> (nHeight / Params().SubsidyHalvingInterval());$

COIN的值为100000000，即1BTC=100000000聪。

nHeight是上一个块的nHeight+1。

nSubsidyHalvingInterval是奖励减半的间隔。

类型	奖励减半间隔
正常	210000
回归测试	150

块的比特币奖励保存在块的block->vtx[0].vout[0].nValue中，以聪为单位，此值为奖励的比特币数量与手续费的和。

格式化哈希缓冲区

格式化块的数据，分解成3份，FormatHashBuffers函数实现。
首先定义了一个临时结构用于格式化，其定义如下：

```
struct
{
    struct unnamed2
    {
        int nVersion;                // 与CBlock的定义相同
        uint256 hashPrevBlock;
        uint256 hashMerkleRoot;
        unsigned int nTime;
        unsigned int nBits;
        unsigned int nNonce;
    }
    block;
    unsigned char pchPadding0[64];    // block的补齐位
    uint256 hash1;
    unsigned char pchPadding1[64];    // hash1的补齐位
}
```

先用块CBlock中的相关数据初始化tmp结构，再格式化tmp中的block、hash1，最后把tmp中的数据按字节交换。

把tmp中的block用pSHA256InitState定义的状态以SHA256算法加密，作为pmidstate的数据。

tmp.block中的前128字节数据作为pdata。

tmp.hash1中的前64字节作为phash1的数据。

调用FormatHashBlocks函数格式化tmp中的block、hash1，先根据len计算blocks数($1 + ((len + 8) / 64)$)，把pbuffer + len为开始的地址，长度为($64 * blocks - len$)的数据置零，把pbuffer[len]设置为0x80。然后计算pend地址($pdata + 64 * blocks$)，把pend[-4]~pend[-1]的数据区设置为($len * 8$)。

SHA256算法加密

扫描nonces寻找有0位的哈希值(ScanHash_CryptoPP)。

SHA256加密算法用的是openssl库中的sha.h头文件中的sha接口函数。

所有的输入缓冲区都是16字节对齐的。此加密主要操作大编码数据。调用者进行字节交换。

循环加密，加密函数是SHA256Transform，加密分2步：

1、把pmdstate中的数据作为开始状态，把pdata中的数据HASH到已经格式化的phash1中。

2、用pSHA256InitState中定义的状态把phash1中的数据HASH到phash中。

加密过程中，如果phash中的偏移为14的WORD类型的值为0，则加密成功，找到了0位的哈希。

pdata中的偏移为12的数据作为临时计数nNonce，最多循环0xFFFF次。当循环0xFFFF次后，线程设置断点。当循环0xFFFF次后，表示加密失败，重新创建块，nNonce置零。

检验块

挖矿创建的块需要校验有效性(CheckWork)。

首先满足2个条件再进行后续校验。

1、块的哈希值不大于块nBits的压缩值(CBigNum().SetCompact)。

2、块的上一个块的哈希值(hashPrevBlock)等于活跃块链的顶端块(chainActive.Tip())的哈希值。

后续的校验与网络协议节点发送过来的块(“block”消息)的校验方式相同(ProcessBlock函数)。

校验分以下几步：

1、检验此块是否在块索引映射mapBlockIndex、独立块映射mapOrphanBlocks中，如果已经存在，则此块无效。

2、检验块内容(CheckBlock)，这是基本的检查。检验块的大小、POW(CheckProofOfWork)、时间戳、交易信息(CheckTransaction)、签名的有效性。块的第一个交易信息必须是coinbase，必须重置。检验多重支付的id，可以尽早捕获潜在的DoS攻击。构建merkle树，检验merkle树根节点的有效性。

3、如果已经检验过，则进行扩展检查，目的是阻止假冒的块填充到内存中。

满足以下条件的块是假冒的块。

a、块的时间戳早于上一次检验的时间戳，则是假冒的块。

b、块需要的POW小于上一次检验的最小POW，则是假冒的块。

如果块有上一个块，但块索引映射(mapBlockIndex)中没有，则需要构建独立的块(COrphanBlock)，把新块添加到mapOrphanBlocks、mapOrphanBlocksByPrev中。

检验通过后，把此块保存到磁盘(AcceptBlock)。

写入磁盘时，再次检查，满足以下几点才能写入磁盘：

1、块索引映射(mapBlockIndex)中不含有此块的哈希值。

2、块的POW、时间戳、交易信息有效。

3、与检验点匹配。

4、版本检查，当网络更新至95%时(测试网络更新至75%时)不再写入版本号为1的块。coinbase中，最后1000个块中至少含有750个版本号为2的块，测试网络最后100个块中至少含有51个块。

检查通过后，寻找文件中空余位置，把此块写入文件，且添加到块索引中。最后添加到节点的块清单中。

重复处理mapOrphanBlocksByPrev中依赖此块的独立块。

工作量证明

工作量证明 (Proof Of Work, 简写POW)要求计算机用SHA256算法转换数据, 数据区中偏移为12的4个字节作为随机数, 直到运算出的哈希缓冲区中偏移为28的2个字节的数据值为0, 运算过程一般需要10分钟。在比特币的挖矿、确认交易中应用了工作量证明机制。

挖矿时创建新区块平均需要10分钟($\text{int64_t } nTargetSpacing = 10 * 60$), 2周($\text{int64_t } nTargetTimespan = 14 * 24 * 60 * 60$)内创建2016个块($\text{int64_t } nInterval = nTargetTimespan / nTargetSpacing$), 创建2016个块后根据这2016个块的难度计算新创建的块的难度, 保证新块的生成时间是10分钟。

块的难度保存在块头CBlockHeader的nBits中, 长度是4个字节, 设置为CBigNum的压缩值GetCompact()。

块的最大难度是0.最小难度保存在系统定义的变量中。

类型	变量	难度值
正常	类CMainParams的 bnProofOfWorkLimit	CBigNum(~uint256(0) >> 32)
回归测试	类CRegTestParams的 bnProofOfWorkLimit	CBigNum(~uint256(0) >> 1)

新创建的块的哈希值不大于块的难度时(nBits的256位无符号整数)才是有效块(CheckProofOfWork函数)。

创世纪块(Genesis block)的难度为参数定义的最小难度。

每创建2016个块后计算新的难度, 此后的2016个块使用新的难度。

在GetNextWorkRequired函数中计算新的难度, 保存在新块的nBits中。计算步骤:

- 1、找到前2016个块的第一个块。
- 2、计算生成这2016个块花费的时间, 即最后一个块的时间与第一个块的时间差。时间差在3.5天~56天之间。
- 3、计算前2016个块的难度总和, 即单个块的难度*时间。
- 4、计算新的难度, 即难度总和 / 14天的秒数, 得到每秒的难度值。
- 5、新的难度不能大于参数定义的最小难度。

在测试网络中, 生成这2016个块时的难度按新的规则计算:

- 1、生成新块的时间比上一个块的时间晚了28天, 则新难度定为参数定义的最小难度值。
- 2、在上一个块的后28天之内生成了新块, 则新难度为在此2016个块组中不是参数定义的最小难度的最近的块的难度 (the last non-special-min-difficulty-rules-block)。

比特币系统提供了一个函数计算一段时间内的最小难度，用来计算某个检验点之后一段时间的最小难度。(ComputeMinWork函数)，计算规则如下：

1、在测试网络中，如果时间区间大于28天，则最小难度为参数定义的最小难度。

2、时间以56天为单位递减，难度4倍递增，循环元算，计算出有效的难度值。

3、新的难度不能大于参数定义的最小难度。

挖矿时创建新块后，需要校验POW是否有效(ProcessBlock函数)，先检验难度是否在有效难度范围内(CheckProofOfWork函数)。如果存在检验点，再检验是否在检验点的最小工作难度内(ComputeMinWork函数计算检验点、新块之间的最小难度)。

把块写入磁盘时要进行精确校验POW(AcceptBlock函数)，计算新块的难度(GetNextWorkRequired函数)，检验是否与块中的nBits相等。

在测试网络中，更新时间时，重新计算块的难度(UpdateTime函数)。

从磁盘中读取块时也要校验POW是否在有效范围内(ReadBlockFromDisk函数)。

计算算力

算力是指每秒创建的块数量，以khash/s为单位，double类型。

每4秒计算一次算力，统计每4秒内创建的块数量。时间间隔以毫秒为单位。

计算公式为：

算力 = 块数量 * 1000.0 / ((当前时间 - 开始时间) * 1000.0)

每隔30分钟把算力写入日志。

比特币网络

比特币系统实例定义了3种网络类型，分别是：

```
enum Network {  
    MAIN,  
    TESTNET,  
    REGTEST,
```

```
    MAX_NETWORK_TYPES
```

```
};
```

MAIN类型用于用户交易商品、服务。

TESTNET类型是公共测试网络，重置时间。

REGTEST类型，全名是regression test，回归测试，仅仅用于个人网络。