比特币源码研读系列5 挖矿（1）

之前我们花了好几篇的篇幅来讲解比特币是如何创建tx的，tx是整个比特币系统最重要的单元，单纯产生一个tx还不够，前面我们也说到过，只有当tx被放入到block中并且在区块链中深度超过6时，该tx才能得到公认。所以tx如何被放入block的过程就显得特别重要，因此今天我们来讲讲比特币系统是如何生产block的，在代码中的方法就是generateBlocks方法。这个过程俗称**挖矿**。

我们先来简单回忆下挖矿的过程：

如何产生一个有效的Block？当任一用户需要新产生一个Block，它将接收到的transactions打包，然后为尝试为这些transactions创建一个有效的Block。这个过程稍微有些复杂，为了方便理解我们同样采用函数的形式来表示:

**Do repeatedly** output = f (transactions, previous Block ID， an additional random guess)，**until** output < target value，

**Then** Block ID = the output.

当得到有效的Block ID也就意味着生成了一个有效的Block。

从这个过程可以看出，target value是挖矿的关键因素，这个值越大，挖矿的难度越低。下面我们来看看代码里是如何实现的？

我们先来看看block的数据结构。

|  |
| --- |
| **class CBlock** |
| int32\_t nVersion; |
| uint256 hashPrevBlock; |
| uint256 hashMerkleRoot; |
| uint32\_t nTime; |
| uint32\_t nBits; |
| uint32\_t nNonce;  vector<CTransactionRef> vtx |

在这些字段中，nVersion是版本号，hashPrevBlock是前一个block的hash值，hashMerkleRoot是该block包括的tx的merkle root，nTime是其生成时间，nBits就是上面提到的**target value**，nNonce则是上述公式中的an additional random guess，vtx则是该block中的全部tx。

本篇我们就来介绍下挖矿的难度指标nBits是如何得到和调整的？

**GetNextWorkRequired方法就是用来计算本次block生成的难度。返回的32bit数字即为挖矿的难度值nBits，也即target value.**

*unsigned int GetNextWorkRequired(const CBlockIndex\* pindexLast, const CBlockHeader \*pblock, const Consensus::Params& params)*

*{*

*assert(pindexLast != nullptr);*

*unsigned int nProofOfWorkLimit = UintToArith256(params.powLimit).GetCompact();****//powLimit代表nBits的上限。因此nProofOfWorkLimit就是最低难度***

*// Only change once per difficulty adjustment interval*

***//跟前一篇提到的一致，2016个block为一个周期。假设当前nHeight=2016\*m+n,当0<=n<2015时，都会return pindexLast->nBits，表示沿用上一个block的难度。说明在一个周期中的所有block难度都一样。***

*if ((pindexLast->nHeight+1) % params.DifficultyAdjustmentInterval() != 0)*

*{*

*if (params.fPowAllowMinDifficultyBlocks)****//只有测试时fPowAllowMinDifficultyBlocks是false，表示允许最小难度。***

*{*

*// Special difficulty rule for testnet:*

*// If the new block's timestamp is more than 2\* 10 minutes*

*// then allow mining of a min-difficulty block.*

*if (pblock->GetBlockTime() > pindexLast->GetBlockTime() + params.nPowTargetSpacing\*2)*

*return nProofOfWorkLimit;*

*else*

*{*

*// Return the last non-special-min-difficulty-rules-block*

*const CBlockIndex\* pindex = pindexLast;*

*while (pindex->pprev && pindex->nHeight % params.DifficultyAdjustmentInterval() != 0 && pindex->nBits == nProofOfWorkLimit)*

*pindex = pindex->pprev;*

*return pindex->nBits;*

*}*

*}*

*return pindexLast->nBits;*

*}*

***//计算每个周期开始时那个block的难度***

*// Go back by what we want to be 14 days worth of blocks*

*int nHeightFirst = pindexLast->nHeight - (params.DifficultyAdjustmentInterval()-1);*

*assert(nHeightFirst >= 0);*

*const CBlockIndex\* pindexFirst = pindexLast->GetAncestor(nHeightFirst);*

*assert(pindexFirst);*

*return CalculateNextWorkRequired(pindexLast, pindexFirst->GetBlockTime(), params);****//下面我们具体来看看该方法。***

*}*

**// CalculateNextWorkRequired方法计算每个周期起始block的难度。**

*unsigned int CalculateNextWorkRequired(const CBlockIndex\* pindexLast, int64\_t nFirstBlockTime, const Consensus::Params& params)*

*{*

*if (params.fPowNoRetargeting)****//如果不调整难度…***

*return pindexLast->nBits;*

***//******nActualTimespan为上一个周期总共消耗的时间。nPowTargetTimespan = 14 \* 24 \* 60 \* 60second，为两个星期的时间。如果上个周期花费时间过少(小于3.5天)则nActualTimespan调整为3.5天；如果花费时间过长(超过了8个星期)，则nActualTimespan调整为8星期。***

*// Limit adjustment step*

*int64\_t nActualTimespan = pindexLast->GetBlockTime() - nFirstBlockTime;*

*if (nActualTimespan < params.nPowTargetTimespan/4)*

*nActualTimespan = params.nPowTargetTimespan/4;*

*if (nActualTimespan > params.nPowTargetTimespan\*4)*

*nActualTimespan = params.nPowTargetTimespan\*4;*

*// Retarget*

*const arith\_uint256 bnPowLimit = UintToArith256(params.powLimit);****//nBits上限***

*arith\_uint256 bnNew;*

*bnNew.SetCompact(pindexLast->nBits);****//将nBits从compact格式恢复成256 bit表示。下面我们会简单介绍下compact数据格式。***

***//调整nBits。至于这么调整会带来什么效果，目前我也未仔细研究。***

*bnNew \*= nActualTimespan;*

*bnNew /= params.nPowTargetTimespan;*

*if (bnNew > bnPowLimit)*

*bnNew = bnPowLimit;*

*return bnNew.GetCompact();****//转换成compact格式***

*}*

在本文的最后我们再介绍下Compact数据格式：

Compact格式是数据的一种紧凑表示，仅需4字节。比如原数据是256 bit的一个数字，用compact表示只需要32 bit。我们看看它是如何做的？

Compact表示规则：

1. 如果第一个字节大于127，则在最开始添加个字节，内容为0
2. Compact格式中第一个字节内容为：步骤1中得到的新数字用256进制表示占用的字节数。
3. Compact格式接下来的2-4字节内容为：步骤2中的256进制表示的结果中前三个字节内容，如果少于3字节则补0.

下面我们以两个例子来说明下：

1. 将1000转换成compact格式

将1000用16进制表示则1000=0x00 00 03 e8，用256(2^8)进制表示则为1000=(0x03)\*256+(0xe8)\*1，由于用256进制表示会占用2字节(0x03e8)，因此compact格式第一个字节内容为：02。第2-4字节内容为：03 e8 00.

因此最后1000的compact格式为：0x0203e800

1. 将2^(256-32)-1用compact格式表示

2^(256-32)-1就是上面提到的最大的powLimit值，

2^(256-32)-1 = 0x ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff （共56个f）

根据规则1由于ff>127，则前面需要补充一个字节，内容为00得到

0x 00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff （共56个f+2个0）

用256进制表示为：

0x 00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff（54个f）\*256 + 0xff \*1

由于256进制表示占用的字节数：1 + 54/2 + 1共需要29个字节，因此compact第一个字节内容为29=0x1d，第2-4字节内容为256进制表示的前三个字节内容：0x 00 ff ff，因此最后2^(256-32)-1的compact表示为：0x 1d 00 ff ff。

好了，本篇我们就介绍到这里。