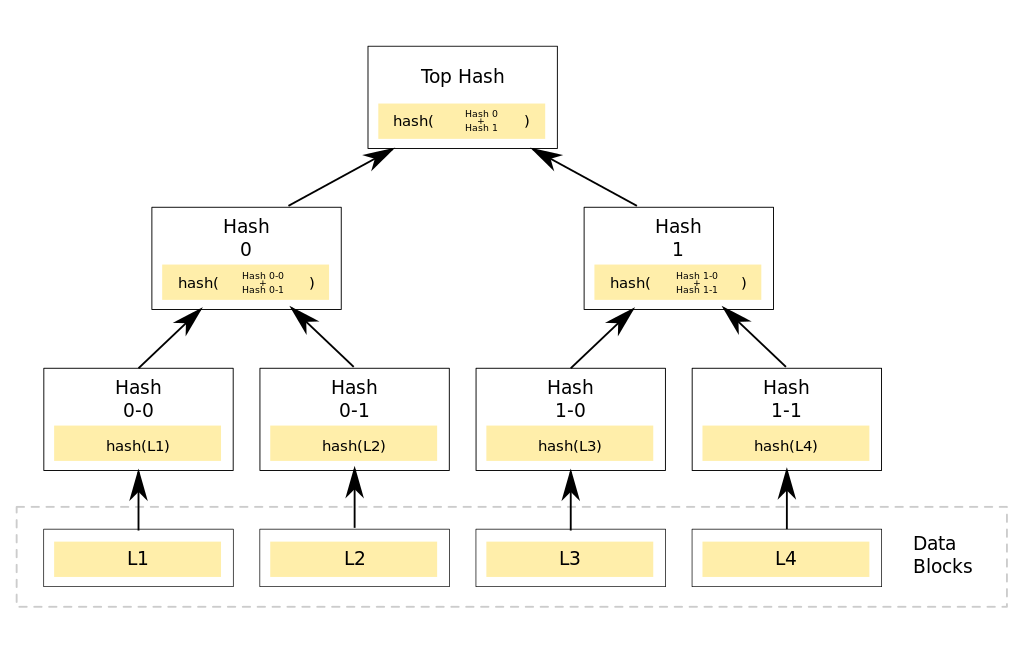
比特币源码研读系列5 merkle树

上一篇我们介绍了挖矿的一个关键的难度系数target value的产生原理，今天我们来看看block中hashMerkleRoot这个值是如何产生的。简单说hashMerkleRoot是block中所有tx组成的merkle tree的树根(tree root)。

**Merkle Tree介绍**

Merkle tree是一种典型的二叉树，merkle root是merkle tree的树根，在下图中就是最上面的那个Top Hash。



其中最下面的L1, L2, L3, L4, …等整个这一层就是该block中所有的tx。以上图为例，该block中只有4个tx，要计算hashMerkleRoot：

1. 首先H1 = Hash (L1, L2)
2. 然后H2 = Hash(L3, L4)
3. 最后hashMerkleRoot = Hash (H1, H2)
4. 在计算过程中，如果遇到奇数个点时，最后一个节点与自身进行hash计算。

引入merkle tree主要是用于简单支付验证SPV (Simple Payment Verification)，快速查询某个tx是否在该block中。简单的说，现在有一个Tx1和一个Block1，正常情况我需要对Block1中的每一个tx进行比较，最坏的情况一遍查找下来没找到，假设Block1中有N个tx，则最坏情况需要查找N次。如果使用merkle tree需要多少次呢？需要LogN次，如果N=1024，普通的查找可能需要1024次，而SPV只需要Log(1024)=10次。

**SPV验证过程**

以下图为例，编号Hk的tx如何证明自己是在该merkle tree中呢？



Hk只用提供一个如上图中所示的4个蓝色节点值，也称merkle tree path：

hashMerkleRoot = H(a-p)

H1 = H(a-h)

H2 = H(ij)

H3 = H(mnop)

H4 = H(l)

那么互联网上其他节点只需要做如下操作就能验证Hk是否在该block中：

1. 计算tmp1 = Hash (H4, Hk)
2. 计算tmp2 = Hash(tmp1, H2)
3. 计算 tmp3 = Hash(tmp2, H3)
4. 计算tmp4 = Hash(tmp3, H1)
5. 最后比较tmp4是否与hashMerkleRoot相等，如果相等则说明Hk在该block中。

**hashMerkleRoot的代码实现**

ComputeMerkleRoot方法用于计算block的hashMerkleRoot值，入参mutated开始为false，如果在生成hashMerkleRoot的过程出现某个tx与某subtree的hash值一样，则mutated=true，表示出现了重复子树。

*uint256 ComputeMerkleRoot(const std::vector<uint256>& leaves, bool\* mutated) {*

*uint256 hash;*

*MerkleComputation(leaves, &hash, mutated, -1, nullptr);*

*return hash;*

*}*

*static void MerkleComputation(const std::vector<uint256>& leaves, uint256\* proot, bool\* pmutated, uint32\_t branchpos, std::vector<uint256>\* pbranch) {*

***//leaves数组是block中的全部tx***

*if (pbranch) pbranch->clear();*

*if (leaves.size() == 0) {*

*if (pmutated) \*pmutated = false;*

*if (proot) \*proot = uint256();*

*return;*

*}*

*bool mutated = false;*

*// count is the number of leaves processed so far.*

*uint32\_t count = 0;*

*// inner is an array of eagerly computed subtree hashes, indexed by tree*

*// level (0 being the leaves).*

*// For example, when count is 25 (11001 in binary), inner[4] is the hash of*

*// the first 16 leaves, inner[3] of the next 8 leaves, and inner[0] equal to*

*// the last leaf. The other inner entries are undefined.*

*uint256 inner[32];*

*// Which position in inner is a hash that depends on the matching leaf.*

*int matchlevel = -1;*

*// First process all leaves into 'inner' values.*

***我们来说下Inner数组所表达的意思：我们假设block中有25个tx。25用二进制表示为*** ***11001，则Inner[4]等于第1-16个tx的merkle tree root，Inner[3]等于第17-24个tx的merkle tree root，Inner[0]等于第25个tx的hash值。***

***我们来演示下这个计算过程：假设block中有7个tx，也就是leaves.size=7***

***Count=0,得到inner[0]=leaves[0]***

***Count=1,得到inner[1] = hash(inner[0],leaves[1]) = hash(leaves[0],leaves[1])***

***Count=2,得到inner[0]=leaves[2]***

***Count=3,得到inner[2]=hash( hash( leaves[3],inner[0] ), inner[1] )***

***Count=4,得到inner[0]=leaves[4]***

***Count=5,得到inner[1]=hash( inner[0],leaves[5] )***

***Count=6,得到inner[0] = leaves[6]***

***最后inner[0]等于第7个tx的hash值，inner[1]等于第5-6两个tx的子树根，inner[2]等于第1-4个tx的子树根。***

***//计算Inner数组开始***

*while (count < leaves.size()) {*

*uint256 h = leaves[count];*

*bool matchh = count == branchpos;*

*count++;*

*int level;*

*// For each of the lower bits in count that are 0, do 1 step. Each*

*// corresponds to an inner value that existed before processing the*

*// current leaf, and each needs a hash to combine it.*

*for (level = 0; !(count & (((uint32\_t)1) << level)); level++) {*

*if (pbranch) {*

*if (matchh) {*

*pbranch->push\_back(inner[level]);*

*} else if (matchlevel == level) {*

*pbranch->push\_back(h);*

*matchh = true;*

*}*

*}*

*mutated |= (inner[level] == h);****//存在某个子树的tree root与某个tx的hash值相同，则mutated=true***

*CHash256().Write(inner[level].begin(), 32).Write(h.begin(), 32).Finalize(h.begin());****//计算Hash(inner[level],h)并保存到h中***

*}*

*// Store the resulting hash at inner position level.*

*inner[level] = h;*

*if (matchh) {*

*matchlevel = level;*

*}*

*}*

***//计算Inner数组结束。***

*// Do a final 'sweep' over the rightmost branch of the tree to process*

*// odd levels, and reduce everything to a single top value.*

*// Level is the level (counted from the bottom) up to which we've sweeped.*

*int level = 0;*

*// As long as bit number level in count is zero, skip it. It means there*

*// is nothing left at this level.*

***//inner数组已经计算完毕，这里我们来计算下merkle root，也就是block的hashMerkleRoot值。***

***我们也来演示下下面这个过程：假设count=7=(111)，block中有7个tx***

***Count=(111), level=0,计算h=hash(inner[0],inner[0])，h为第7个tx与自身的hash值。***

***Count=(1000),level=1,计算h=hash( inner[1], h)，h为第4-7个tx的子树根***

***Count=(1000),level=2,计算h=hash( inner[2], h )，h为第1-7个tx的树根***

*while (!(count & (((uint32\_t)1) << level))) {****//level移动到count第一个非0位时循环终止，如count=111,则level=0循环终止***

*level++;*

*}*

*uint256 h = inner[level];*

*bool matchh = matchlevel == level;*

*while (count != (((uint32\_t)1) << level)) {*

*// If we reach this point, h is an inner value that is not the top.*

*// We combine it with itself (Bitcoin's special rule for odd levels in*

*// the tree) to produce a higher level one.*

*if (pbranch && matchh) {*

*pbranch->push\_back(h);*

*}*

*CHash256().Write(h.begin(), 32).Write(h.begin(), 32).Finalize(h.begin());*

*// Increment count to the value it would have if two entries at this*

*// level had existed.*

*count += (((uint32\_t)1) << level);*

*level++;*

*// And propagate the result upwards accordingly.*

*while (!(count & (((uint32\_t)1) << level))) {*

*if (pbranch) {*

*if (matchh) {*

*pbranch->push\_back(inner[level]);*

*} else if (matchlevel == level) {*

*pbranch->push\_back(h);*

*matchh = true;*

*}*

*}*

*CHash256().Write(inner[level].begin(), 32).Write(h.begin(), 32).Finalize(h.begin());*

*level++;*

*}*

*}*

***//到此block的hashMerkleRoot值已计算完毕。***

*// Return result.*

*if (pmutated) \*pmutated = mutated;*

*if (proot) \*proot = h;****//返回block的hashMerkleRoot值***

*}*

到此，我们便大致了解到block中的hashMerkleRoot值的计算过程，好了，本篇到此为止。