1. 整理C++文件-函数
2. 整理函数关系
3. 整体头文件-类和函数
4. 整体类关系
5. 分析交易，共识，加密

# 1 C++ file

## main.cpp

### CTxIn-入交易

#### bool CTxIn::IsMine() const

矿工处理

#### int64 CTxIn::GetDebit() const

获取发送方（借方）信息

### CWalletTx-钱包交易

#### int64 CWalletTx::GetTxTime() const

获取交易时间

#### void CWalletTx::AddSupportingTransactions(CTxDB& txdb)

增加可支持的交易

### CTransaction-交易

#### bool CTransaction::AcceptTransaction(CTxDB& txdb, bool fCheckInputs, bool\* pfMissingInputs)

接受交易

#### bool CTransaction::AddToMemoryPool()

增加到内存池

### CMerkleTx-可信数

Merkle 可信树是为了**解决多重一次签名中的认证问题**而产生的，Merkle 可信树结构具有**一次签名大量认证**的优点，在认证方面具有显著的优势。如今，Merkle 可信树的树形结构已经被广泛应用到了信息安全的各个领域，比如**证书撤销、源组播认证、群密钥协商**等等。并且基于 Merkle 可信树的数字签名方案在安全性上**仅仅依赖于哈希函数**的安全性，且不需要太多的理论假设，这使得基于 Merkle 可信树的**数字签名更加安全、实用**。

#### int CMerkleTx::SetMerkleBranch(const CBlock\* pblock)

设置可信树分支

### bool AddKey(const CKey& key)

增加key

### vector<unsigned char> GenerateNewKey()

生成key

vector是一个类模板不是数据类型，但可以定义多种数据类型，vector <类型> 指定其保存元素的类型，如vector <int>是数据类型。类似于一个数组。

### bool AddToWallet(const CWalletTx& wtxIn)

增加到wallet钱包

### bool AddToWalletIfMine(const CTransaction& tx, const CBlock\* pblock)

增加到矿工的钱包中，调用AddToWallet

### bool EraseFromWallet(uint256 hash)

从钱包中清除

### void AddOrphanTx(const CDataStream& vMsg)

增加一个orphan（独立的）交易

### void EraseOrphanTx(uint256 hash)

清除Orphan（独立的）交易

### bool ProcessBlock(CNode\* pfrom, CBlock\* pblock)

处理区块

### bool ScanMessageStart(Stream& s)

从头部开始浏览信息

template<typename Stream>

bool ScanMessageStart(Stream& s) {}

使用方法详见附录1

### string GetAppDir()

获得app的目录信息

### FILE\* OpenBlockFile(unsigned int nFile, unsigned int nBlockPos, const char\* pszMode)

打开区块文件

### FILE\* AppendBlockFile(unsigned int& nFileRet)

增加区块文件

### bool LoadBlockIndex(bool fAllowNew)

加载区块索引

### void PrintBlockTree()

打印区块数

### bool AlreadyHave(CTxDB& txdb, const CInv& inv)

### bool ProcessMessages(CNode\* pfrom)

处理所有消息

### bool ProcessMessage(CNode\* pfrom, string strCommand, CDataStream& vRecv)

处理消息，被处理所有消息调用

### bool SendMessages(CNode\* pto)

发送所有消息

### int FormatHashBlocks(void\* pbuffer, unsigned int len)

格式化hash块

### void BlockSHA256(const void\* pin, unsigned int nBlocks, void\* pout)

块加密

### bool BitcoinMiner()

比特币矿工（工人）

### int64 GetBalance()

获得balance

### bool SelectCoins(int64 nTargetValue, set<CWalletTx\*>& setCoinsRet)

选择币

### bool CreateTransaction(CScript scriptPubKey, int64 nValue, CWalletTx& wtxNew, int64& nFeeRequiredRet)

创建交易

### bool CommitTransactionSpent(const CWalletTx& wtxNew)

提交交易

### bool SendMoney(CScript scriptPubKey, int64 nValue, CWalletTx& wtxNew)

发送币

# 2 C++ header file

# 附录

## 附录1

template<typename T>

这个是C++中的模板..template<typename T> 这个是定义模板的固定格式,

比如你想求2个int float 或double型变量的值,只需要定义这么一个函数就可以了,

假如不用模板的话,你就必须针对每种类型都定义一个sum函数..

int sum(int, int);

float sum(float, float);

double sum(double, double);

如果定义模块，则需要以下定义

template<typename T>

T sum(T a, T b);

深层次解释：

1.因为T是一个**模版实例化时才知道的类型**，所以编译器更对T不知所云，为了通知**编译器T是一个合法的类型，使用typename语句可以避免编译器报错**。

2.template < typename var\_name >表示var\_name是一个类型， 在模版实例化时可以替换任意类型，不仅包括内置类型（int等），也包括自定义类型class。

换句话说，在template<typename Y>和template<class Y>中， typename和class的意义完全一样。

3.建议在这种语句中尽可能采用typename，以避免错觉（因为只能替换class,不能只换int),

这也是C++新标准引进typename关键词的一个初衷

例子：

1. ///</summary>
2. ///<remarks>1:template的使用是为了简化不同类型的函数和类的重复定义.
3. ///<remarks>2:char类型变量c,d输入的都是字母，不是数字，如输入32则c=3,d=2.
4. #include "stdafx.h"
5. #include <iostream>
6. #include<vector>
7. **using** **namespace** std;
8. **template** <**typename** T>
9. T mmax(T a,T b)
10. {
11. **return** a>b?a:b;
12. }
13. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
14. {
15. cout<<"Please enter the value of a and b:"<<endl;
16. **int** a,b;
17. cin>>a>>b;
18. cout<<mmax(a,b)<<endl;
19. cout<<"Please enter the value of c and d:"<<endl;
20. **char** c,d;
21. cin>>c>>d;
22. cout<<mmax(c,d)<<endl;
23. cout<<"Please enter the value of f and g:"<<endl;
24. **double** f,g;
25. cin>>f>>g;
26. cout<<mmax(f,g)<<endl;
27. **while**(1);
28. **return** 0;
29. }