比特币交易

比特币的发送、接收都是通过比特币交易来实现的，发送时构造比特币交易，接收到比特币交易后处理。

比特币交易主要包含3个方面的数据：

1、比特币的来源，挖矿或者比特币交易。

2、接收方的地址、数额。

3、签名。

比特币交易定义了6种类型，支持各种交易。

当节点接收到比特币交易后，需要先校验交易的有效性，校验成功后再处理。

一、结构

系统定义了一些比特币交易相关的结构，用于交易、撤销等。

1、输出点

系统定义了交易的输出点，包含交易输出的哈希值、交易输出数组索引。

```
class COutPoint
{
public:
    uint256 hash;           // 交易哈希值
    unsigned int n;         // 交易输出数组索引
};
```

2、输入点

系统定义了输入点，包含交易、交易输入数组索引。

```
class CInPoint
{
public:
    const CTransaction* ptx;            // 交易
    unsigned int n;                     // 交易输入数组索引
};
```

此结构主要用于交易的内存池中。

3、输入交易

系统定义了交易输入，包含了上一个交易输出点的位置、公钥的签名。

```
class CTxIn
```

```
{
public:
    COutPoint prevout;                  // 上一个交易输出点的位置
    CScript scriptSig;                  // 公钥的签名
    unsigned int nSequence;             // 序列
};
```

此结构主要是为了说明用于交易的比特币的来源。

4、输出交易

系统定义了交易输出，包含了交易的比特币数量、公钥、以及一些标志。

```
class CTxOut
{
public:
    int64_t nValue;                 // 比特币数量
    CScript scriptPubKey;           // 公钥脚本
};
```

此结构主要标明交易时发送的比特币数量、接收方的地址、标志。

5、交易信息

系统定义了交易信息，在网络中广播。

交易信息可以包含多个交易输入、交易输出。

```
class CTransaction
{
public:
    static int64_t nMinTxFee;                   // 最小的交易手续费
    static int64_t nMinRelayTxFee;              // 最小的传播交易手续费
    static const int CURRENT_VERSION=1;         // 当前版本
    int nVersion;                               // 版本号
    std::vector<CTxIn> vin;                     // 交易输入数组
    std::vector<CTxOut> vout;                   // 交易输出数组
    unsigned int nLockTime;                     // 锁定时间
};
```

6、交易输出压缩器

系统定义了交易输出压缩器，提供一个更紧凑的序列化的数据，用于后面的币（Coin）。

```
class CTxOutCompressor
```

```
{
private:
    CTxOut &txout;
};
```

7、交易输入的撤销信息

系统定义了交易输入的撤销信息，包含了花费之前的交易输出数据，如果交易输出是最后一本未花费的，则元数据还包含是否是coinbase、高度、交易版本号。

```
class CTxInUndo
{
public:
    CTxOut txout;           // the txout data before being spent
    bool fCoinBase;         // if the outpoint was the last unspent: whether it belonged
                               to a coinbase
    unsigned int nHeight;  // if the outpoint was the last unspent: its height
    int nVersion;           // if the outpoint was the last unspent: its version
};
```

8、交易的撤销信息

系统定义了交易的撤销信息。

```
class CTxUndo
{
public:
    // undo information for all txins
    std::vector<CTxInUndo> vprevout;
};
```

二、类型

系统定义了6种交易类型，类型包含在输出交易的scriptPubKey中，函数Solver用于解析scriptPubKey，获取交易类型等。

交易类型大体分2种：标准的、非标准的。除了非标准的，剩下的5种都是标准的。

序号	交易类型	意义
0	TX_NONSTANDARD	非标准的交易
1	TX_PUBKEY	公钥
2	TX_PUBKEYHASH	公钥哈希

序号	交易类型	意义
3	TX_SCRIPTHASH	脚本哈希
4	TX_MULTISIG	多重签名
5	TX_NULL_DATA	空数据

三、构造交易

钱包类CWallet的CreateTransaction函数创建交易。

主要分3步：

1、填充交易的输出交易（vout）

创建交易时，指定输出交易的信息，主要是输出的脚本（地址构造成CScript）、输出的币数量（nValue）。

参考类CWallet的SendMoneyToDestination函数。

2、填充交易的输入交易（vin）

先从钱包的交易信息中选择合适的比特币（SelectCoins函数），填充到交易的输入交易中。

3、签名（CTxIn.scriptSig）

对输入交易的scriptSig签名（SignSignature函数）。

由新的交易信息、私钥计算哈希值（SignatureHash函数），填充到输入交易的scriptSig中（Solver函数）。

构造交易完毕，再提交交易，发送出去。

四、发送交易

当构造完交易，则提交交易（钱包类CWallet的CommitTransaction函数），发送出去。

提交交易时，先把交易添加到钱包中，然后标记旧的比特币为已经花费，再添加到交易内存池中，最后把交易传播下去。

传播交易时（RelayTransaction函数），由交易信息的哈希值构造CInv，类型为MSG_TX，添加到每个节点的发送清单（vInventoryToSend）中，发送消息（SendMessages）时把节点中的发送清单中的交易信息以”inv”命令发送出去。

五、接收交易

当节点接收到交易命令（”tx”）后，把交易信息添加到交易内存池中，且传播下去。

六、交易有效性

接收到交易信息后，需要校验交易的有效性，有效的交易才处理。

满足以下几个条件的交易才是有效的：

1、交易的输入交易数组(vin)不能为空。

2、交易的输出交易数组(vou)不能为空。

3、交易的大小不能超过区块最大值(MAX_BLOCK_SIZE=1000000)。

4、交易的单个输出交易的值(txout.nValue)必须在0~2100万之间，且所有输出交易的值总和也在0~2100万之间。

5、不能是双重支付，即交易输入的上一次交易输出点不能重复(txin.prevout)。

6、如果交易是挖矿产生的(CoinBase)，则交易输入数组中的索引为0的项的签名的大小(tx.vin[0].scriptSig.size())必须在0~100之间。如果交易是普通交易，则交易输入的上一次输出点不能为空。

密钥

比特币系统的密钥体系采用了椭圆曲线数字签名算法（ECDSA），生成私钥、公钥，以及签名。

ECDSA算法采用了openssl库的ecdsa.h中的实现。

系统定义了EC_KEY_regenerate_key函数，成对生成私钥（EC_KEY_set_private_key函数）、公钥（EC_KEY_set_public_key函数）。

一、密钥

系统定义了类CECKey，封装了openssl库的EC_KEY，包含了公钥、私钥的生成、获取，以及签名、校验。

类的定义如下：

```
class CECKey {
private:
    EC_KEY *pkey;
};
```

SetSecretBytes函数重新生成公钥、私钥，调用EC_KEY_regenerate_key函数实现。

GetPrivKey函数获取私钥，调用i2d_ECPrivateKey函数实现。

GetPubKey函数获取公钥，调用i2o_ECPublicKey函数实现。

Sign函数打签名，调用ECDSA_do_sign、i2d_ECDSA_SIG函数实现。

Verify函数校验签名，调用ECDSA_verify函数实现。

SignCompact函数压缩签名。

系统定义了类CKey，封装了密钥，主要提供给其他模块调用。

类CKey的定义如下：

```
class CKey {
private:
    // Whether this private key is valid. We check for correctness when modifying the key
    // data, so fValid should always correspond to the actual state.
    bool fValid;

    // Whether the public key corresponding to this private key is (to be) compressed.
    bool fCompressed;

    // The actual byte data
```

```
    unsigned char vch[32];
};
```

GetPrivKey函数获取私钥。

GetPubKey函数获取公钥。

Sign函数打签名。

SignCompact函数打压缩的签名。

二、公钥

系统定义了类CPubKey，封装了公钥。

类CPubKey的定义如下：

```
class CPubKey {
private:
    // Just store the serialized data.
    // Its length can very cheaply be computed from the first byte.
    unsigned char vch[65];
};
```

GetID函数获取公钥的ID，公钥字符串经过Hash160算法计算后的数值。Hash160算法是先做SHA256运算，再经过RIPEMD160算法计算。

GetHash函数获取公钥的256位哈希值，公钥字符串经过Hash算法计算后的数值。Hash算法是做2次SHA256运算。

Verify函数校验DER签名。

VerifyCompact函数校验压缩的签名。

三、加密的公钥、私钥

系统定义了可加密的公钥、私钥。加密、解密算法可以看Encode、Decode函数。

系统定义了类CExtKey封装了可加密的密钥。

类CExtPubKey的定义如下：

```
struct CExtKey {
    unsigned char nDepth;
    unsigned char vchFingerprint[4];
    unsigned int nChild;
    unsigned char vchChainCode[32];
    CKey key;
};
```

Encode函数加密密钥。

Decode函数解密密钥。

系统定义了类CExtPubKey封装了可加密的公钥。

类CExtPubKey的定义如下：

```
struct CExtPubKey {
    unsigned char nDepth;
    unsigned char vchFingerprint[4];
    unsigned int nChild;
    unsigned char vchChainCode[32];
    CPubKey pubkey;
};
```

Encode函数加密公钥。

Decode函数解密公钥。

四、签名

比特币交易时需要打签名，保证安全性。

1、签名

交易的签名保存在交易的输入（CTxIn）的scriptSig中，由交易的输出（CTxOut）的公钥（scriptPubKey）经过运算得出来。

SignSignature函数是做签名的，签名之前先组合好交易输出的公钥，然后做签名。

签名时先调用SignatureHash函数计算公钥的哈希值，然后再对哈希值做签名。

SignatureHash函数计算哈希值时，先把相关的数据构成CTransactionSignatureSerializer类型的数据，再把此数据构成CHashWriter类型的数据，然后获取哈希值。计算哈希时，先经过SHA256_Final运算，再经过SHA256运算。

签名时调用Solver函数根据交易的类型做签名，Solver函数调用Sign1函数，Sign1函数调用密钥CKey的Sign函数，Sign函数调用CECKey的Sign函数，Sign函数调用ECDSA的函数计算签名。

交易类型做签名分以下几种：

交易类型	签名
TX_NONSTANDARD	不做签名
TX_PUBKEY	公钥经过Hash160算法运算后再签名
TX_PUBKEYHASH	公钥转换成uint160数后再做签名

交易类型	签名
TX_SCRIPTHASH	公钥转换成uint160数，获取其对应的脚本。
TX_MULTISIG	多重签名，对交易的每个输出的公钥经过Hash160算法运算后做签名。
TX_NULL_DATA	不做签名

2、验证签名

CheckSig函数验证签名。先调用SignatureHash函数计算公钥的哈希值，再调用公钥CPubKey的Verify函数验证签名。

Verify函数调用CECKey的Verify函数校验，Verify函数再调用ECDSA_verify函数验证签名。

地址

系统定义了类CBitcoinAddress表示地址。

类CBitcoinAddress的定义如下：

```
class CBitcoinAddress : public CBase58Data
{
};
```

类CBitcoinAddress可以从交易目的地址（CTxDestination）、字符串等多种方式构造地址。

比特币地址由公钥经过运算得到的。

比特币地址的结构如下：

前缀（1个字节）	公钥ID（20个字节）	公钥ID的校验和的前4个字节（4个字节）

计算步骤如下：

1、获取公钥对应的前缀。

2、计算公钥的ID，先做SHA256运算，再经过RIPEMD160算法计算。

3、计算公钥ID的校验和，做2次SHA256运算。EncodeBase58Check函数计算公钥ID的校验和。

4、组合成比特币地址。

公钥前缀对应关系如下：

地址类型	主网络前缀	测试网络前缀
PUBKEY_ADDRESS	list_of(0)	list_of(111)
SCRIPT_ADDRESS	list_of(5)	list_of(196)
SECRET_KEY	list_of(128)	list_of(239)
EXT_PUBLIC_KEY	list_of(0x04)(0x88)(0xB2)(0x1E)	list_of(0x04)(0x35)(0x87)(0xCF)
EXT_SECRET_KEY	list_of(0x04)(0x88)(0xAD)(0xE4)	list_of(0x04)(0x35)(0x83)(0x94)

疑问：前缀0怎么变成1了呢?

脚本

比特币的交易采用了脚本系统，可以支持多种类型的交易，脚本采用了类Forth语言，是基于堆栈的，从左至右处理，是非图灵完整，没有循环指令。

脚本保存在交易CTransaction的交易输出CTxOut的scriptPubKey中。

一、脚本

脚本（CScript）是无符号字符数组（std::vector<unsigned char>），最大520字节（MAX_SCRIPT_ELEMENT_SIZE）。

类CScript的定义如下：

```
class CScript : public std::vector<unsigned char>
{
}
```

二、操作码

脚本操作码分9种，对应不同的操作，其中OP_INVALIDOPCODE（0xff）是无效操作码。

有些操作码已经被禁用，如果脚本中包含已禁用的操作码，则交易必须停止，且执行失败。

1、常数

常数是把数据压入堆栈。

操作码	数值	描述
OP_0、OP_FALSE	0x00	一个空的数组字节压入堆栈
OP_PUSHDATA1	0x4c	后面1个字节包含压入堆栈的字节长度
OP_PUSHDATA2	0x4d	后面2个字节包含压入堆栈的字节长度
OP_PUSHDATA4	0x4e	后面4个字节包含压入堆栈的字节长度
OP_1NEGATE	0x4f	数字-1压入堆栈
OP_RESERVED	0x50	交易无效，除非发生在未执行的OP_IF分支。
OP_1、OP_TRUE	0x51	数字1压入堆栈
OP_2～OP_16	0x52～0x60	单词名称中的数字压入堆栈

2、流程控制

与流程控制相关的操作码。

操作码	数值	描述
OP_NOP	0x61	空指令

操作码	数值	描述
OP_VER	0x62	交易无效，除非发生在未执行的OP_IF分支。
OP_IF	0x63	如果栈顶元素非0，则语句执行，移除栈顶元素
OP_NOTIF	0x64	如果栈顶元素是0，则语句执行，移除栈顶元素
OP_VERIF	0x65	交易无效，即使发生在未执行的OP_IF分支。
OP_VERNOTIF	0x66	交易无效，即使发生在未执行的OP_IF分支。
OP_ELSE	0x67	如果OP_IF、OP_NOTIF、OP_ELSE没有执行，则后面的语句块执行；如果OP_IF、OP_NOTIF、OP_ELSE执行了，则后面的语句块不执行。
OP_ENDIF	0x68	if/else块的结束
OP_VERIFY	0x69	如果栈顶元素是非真，则标示交易是无效的。如果是真，则移除；否则不移除。
OP_RETURN	0x6a	标示交易是无效的

3、堆栈操作

与堆栈操作相关的操作码。

操作码	数值	描述
OP_TOALTSTACK	0x6b	把主栈栈顶元素移除，压入辅栈。
OP_FROMALTSTACK	0x6c	把辅栈栈顶元素移除，压入主栈。
OP_2DROP	0x6d	移除栈顶的2个元素
OP_2DUP	0x6e	复制栈顶的2个元素
OP_3DUP	0x6f	复制栈顶的3个元素
OP_2OVER	0x70	复制次栈顶的2个元素到栈顶
OP_2ROT	0x71	把堆栈第5、6个元素移动到栈顶
OP_2SWAP	0x72	交换栈顶的2对元素
OP_IFDUP	0x73	如果栈顶元素非0，则复制一份。
OP_DEPTH	0x74	把堆栈的大小压入栈顶
OP_DROP	0x75	移除栈顶元素
OP_DUP	0x76	复制栈顶元素
OP_NIP	0x77	移除次栈顶元素
OP_OVER	0x78	复制次栈顶元素到栈顶
OP_PICK	0x79	把堆栈第n个元素复制到栈顶
OP_ROLL	0x7a	把堆栈第n个元素移动到栈顶
OP_ROT	0x7b	栈顶的3个元素左移

操作码	数值	描述
OP_SWAP	0x7c	交换栈顶的2个元素
OP_TUCK	0x7d	复制栈顶元素到次栈元素之前

4、字符串处理

与字符串操作相关的操作码。

操作码	数值	描述
OP_CAT	0x7e	把2个字符串连接起来，已禁用。
OP_SUBSTR	0x7f	返回字符串的一部分，已禁用。
OP_LEFT	0x80	保留字符串中指定指针左边的部分，已禁用。
OP_RIGHT	0x81	保留字符串中指定指针右边的部分，已禁用。
OP_SIZE	0x82	把栈顶字符串的长度压入堆栈

5、逻辑处理

这些操作码是输入的逻辑运算。

操作码	数值	描述
OP_INVERT	0x83	输入的所有位取反，已禁用。
OP_AND	0x84	输入的所有位进行逻辑与运算，已禁用。
OP_OR	0x85	输入的所有位进行逻辑或运算，已禁用。
OP_XOR	0x86	输入的所有位进行逻辑异或运算，已禁用。
OP_EQUAL	0x87	如果输入相等，则返回1；否则返回0.
OP_EQUALVERIFY	0x88	与OP_EQUAL相同，接着运行OP_VERIFY。
OP_RESERVED1	0x89	交易无效，除非发生在未执行的OP_IF分支。
OP_RESERVED2	0x8a	交易无效，除非发生在未执行的OP_IF分支。

6、数值运算

算法的输入的长度是受限的，是有符号的32位整数，但是输出是可能溢出的。

如果这些命令中的输入值长度大于4字节，则脚本必须停止，且执行失败。

操作码	数值	描述
OP_1ADD	0x8b	输入加1
OP_1SUB	0x8c	输入减1
OP_2MUL	0x8d	输入乘以2，已禁用。

操作码	数值	描述
OP_2DIV	0x8e	输入除以2，已禁用。
OP_NEGATE	0x8f	输入的符号取反。
OP_ABS	0x90	输入取正
OP_NOT	0x91	如果输入是0、1，则取反；否则输出为0。
OP_0NOTEQUAL	0x92	如果输入是0，则返回0；如果输入是1，则返回1。
OP_ADD	0x93	a加上b
OP_SUB	0x94	a减去b
OP_MUL	0x95	a乘以b，已禁用。
OP_DIV	0x96	a除以b，已禁用。
OP_MOD	0x97	返回a除以b的余数，已禁用。
OP_LSHIFT	0x98	a左移b位，保留符号位，已禁用。
OP_RSHIFT	0x99	a右移b位，保留符号位，已禁用。
OP_BOOLAND	0x9a	如果a、b都是非0，则输出1；否则输出0.
OP_BOOLOR	0x9b	如果a或者b非0，则输出1；否则输出0.
OP_NUMEQUAL	0x9c	如果数值相等，则返回1；否则返回0.
OP_NUMEQUALVERIFY	0x9d	与OP_NUMEQUAL相同，之后执行OP_VERIFY。
OP_NUMNOTEQUAL	0x9e	如果数值不相等，则返回1；否则返回0.
OP_LESSTHAN	0x9f	如果a小于b，则返回1；否则返回0.
OP_GREATERTHAN	0xa0	如果a大于b，则返回1；否则返回0.
OP_LESSTHANOREQUAL	0xa1	如果a小于等于b，则返回1；否则返回0.
OP_GREATERTHANOREQUAL	0xa2	如果a大于等于b，则返回1；否则返回0.
OP_MIN	0xa3	返回a、b的较小值。
OP_MAX	0xa4	返回a、b的较大值。
OP_WITHIN	0xa5	如果x在指定范围内，则返回1；否则返回0.

7、加密

运算过程中使用的加密算法。

操作码	数值	描述
OP_RIPEMD160	0xa6	输入用RIPEMD-160哈希
OP_SHA1	0xa7	输入用SHA-1哈希
OP_SHA256	0xa8	输入用SHA-256哈希
OP_HASH160	0xa9	输入哈希2次，先用SHA-256，再用RIPEMD-160。

操作码	数值	描述
OP_HASH256	0xaa	输入用SHA-256哈希2次。
OP_CODESEPARATOR	0xab	所有的签名检查仅仅匹配最近执行了OP_CODESEPARATOR的签名。
OP_CHECKSIG	0xac	整个交易的输出、输入、脚本（从最近执行OP_CODESEPARATOR的到结尾）都要哈希，OP_CHECKSIG使用的签名必须是hash、公钥的有效签名，如果有效，则返回1；否则返回0。
OP_CHECKSIGVERIFY	0xad	与OP_CHECKSIG相同，之后执行OP_VERIFY。
OP_CHECKMULTISIG	0xae	每一个签名、公钥对都要执行OP_CHECKSIG。如果列表中的公钥币签名多，则某些公钥、签名对可以失败。所有的签名都需要匹配一个公钥。如果所有的签名都是有效的，则返回1；否则返回0.有个bug，从堆栈中移除一个额外的未使用的值。
OP_CHECKMULTISIGVERIFY	0xaf	与OP_CHECKMULTISIG相同，之后执行OP_VERIFY。

8、扩展

未分配的操作码是无效的，使用未分配的操作码的交易是无效的。

操作码	数值	描述
OP_NOP1~OP_NOP10	0xb0~0xb9	单词被忽视

9、模版匹配参数

这些操作码在内部使用，用于辅助交易匹配，在实际的脚本中是无效的。

操作码	数值	描述
OP_SMALLDATA	0xf9	很小的压栈数据，小于MAX_OP_RETURN_RELAY（40）
OP_SMALLINTEGER	0xfa	单字节的小整数，压入vSolutions。
OP_PUBKEYS	0xfb	多个OP_PUBKEY
OP_PUBKEYHASH	0xfd	兼容与OP_HASH160哈希的公钥
OP_PUBKEY	0xfe	兼容与OP_CHECKSIG的公钥

三、构造脚本

创建交易时，设置交易的脚本，再打签名。

设置交易的脚本分2种：设置目的地址、设置多个签名。

1、设置目的地址

设置目的地址函数是SetDestination函数，此函数可以设置3种类型的脚本：空地址（CNoDestination）、公钥ID、脚本ID。

公钥ID（CKeyID）表示TX_PUBKEYHASH地址。

脚本ID（CScriptID）是TX_SCRIPTHASH地址。

SetDestination函数通过类CScriptVisitor构造脚本的。

类CScriptVisitor的定义如下：

```
class CScriptVisitor : public boost::static_visitor<bool>
{
private:
    CScript *script;
};
```

空地址脚本仅仅是把脚本清空。

公钥ID类型的脚本的结构是：

OP_DUP	OP_HASH160	KeyID	OP_EQUALVERIFY	OP_CHECKSIG

脚本ID类型的脚本的结构是：

OP_HASH160	ScriptID	OP_EQUAL

2、设置多个签名

设置多个签名是通过SetMultisig函数设置的。签名、公钥数量最多是17个。

多签名的脚本结构：

签名数量的编码	多个PubKey	公钥数量的编码	OP_CHECKMULTISIG

编码是通过EncodeOP_N函数转换的。

如果数量是0，转换为OP_0；否则转换为(OP_1+n-1)。

解码是通过DecodeOP_N函数实现的。

如果编码是0，则解码为0；否则解码为（(int)opcode - (int)(OP_1 - 1)）。

四、解析执行脚本

解析并执行脚本的函数是EvalScript函数。

遍历整个脚本，解析操作码、数据，数据用数组存储，各种操作码（流程控制、运算）用c语言实现，加密调用加密函数实现，校验签名调用签名校验的函数（CheckSig函数），执行完毕后把结果放到堆栈中。

解析过程中注意以下几点：

1、脚本总大小最大是10000字节。

2、脚本元素最大位520字节。

3、多重签名校验时公钥最多是201个。

4、如果脚本中含有已禁用的操作码，则解析失败。

五、校验脚本

校验脚本是VerifyScript函数。

先校验交易输入的签名，再校验交易输出的公钥，如果交易是spend-to-script-hash类型，且指定了SCRIPT_VERIFY_P2SH标志，则校验子脚本，子脚本保存在校验输入签名时返回的主栈的末尾。

堆栈栈顶的数据是校验的结果，需要转换成bool类型。

六、数值转换

脚本、堆栈采用了字符串类型，数值采用了CBigNum类型，解析过程中涉及到类型转换。

1、CBigNum转换成字符串

类CBigNum的接口getvch()把数值转换成字符串。

2、字符串转换成CBigNum

函数CastToBigNum把字符串转换成CBigNum。

3、字符串转换成bool

函数CastToBool把字符串转换成bool类型。

七、压缩版脚本

标准的脚本包含了很多数据，系统定义了压缩版脚本，用于压缩版交易输出、币等。

系统可以压缩KeyID、ScriptID、PubKey类型的脚本，KeyID、ScriptID类型的脚本压缩到21字节，PubKey类型的脚本压缩到33字节。

压缩版脚本包含类型、公钥，不包含脚本的操作码。

压缩版脚本的构成如下：

类型	脚本

压缩版脚本的类型对应关系如下：

类型	编号
KeyID	0x0
ScriptID	0x1
PubKey	0x2 0x3 0x4 0x5

压缩脚本是Compress函数，压缩时先判断脚本类型，然后按照压缩版脚本的结构组合。

解压脚本是Decompress函数，解压时先判断脚本类型，然后按照标准版的脚本的结构组合。

各个类型的脚本判断条件如下：

类型	条件
KeyID	script.size() == 25 && script[0] == OP_DUP && script[1] == OP_HASH160 && script[2] == 20 && script[23] == OP_EQUALVERIFY && script[24] == OP_CHECKSIG
ScriptID	script.size() == 23 && script[0] == OP_HASH160 && script[1] == 20 && script[22] == OP_EQUAL
PubKey（2种方式）	script.size() == 35 && script[0] == 33 && script[34] == OP_CHECKSIG && (script[1] == 0x02 ll script[1] == 0x03)
	script.size() == 67 && script[0] == 65 && script[66] == OP_CHECKSIG && script[1] == 0x04

交易内存池

系统定义了交易内存池，存放交易信息，存放的交易是依据当前最优区块中的有效交易（valid-according-to-the-current-best-chain transactions），且可能出现在下一个区块中。

节点创建的交易、网络接收的交易会添加到内存池中，但并不是所有的交易都会添加到内存池中，如果新的交易的输入已存在内存池中的某个交易中，则丢弃。

节点之间可以发送”mempool”命令获取交易内存池中的交易信息。

一、内存池

交易内存池中由交易内存池项构成。

1、交易内存池项

系统定义了交易内存池项（CTxMemPoolEntry）描述每个交易信息。

类CTxMemPoolEntry的定义如下：

```
class CTxMemPoolEntry
{
private:
    CTransaction tx;          // Transaction
    int64_t nFee;             // Cached to avoid expensive parent-transaction lookups
    size_t nTxSize;           // ... and avoid recomputing tx size
    int64_t nTime;            // Local time when entering the mempool
    double dPriority;         // Priority when entering the mempool
    unsigned int nHeight;  // Chain height when entering the mempool
};
```

成员nHeight表示交易在交易链中的高度，空交易信息的高度默认是MEMPOOL_HEIGHT（0x7FFFFFFF）。

交易的优先级为插入到内存池时的优先级与优先级增量的和，优先级增量的计算公式为：

```
double deltaPriority = ((double)(currentHeight-nHeight)*(tx.GetValueOut()+nFee))/nTxSize;
```

2、交易内存池

系统定义了交易内存池（CTxMemPool）存放交易信息。

类CTxMemPool的定义如下：

```
class CTxMemPool
{
```

```
private:
    bool fSanityCheck;                  // Normally false, true if -checkmempool or -regtest
    unsigned int nTransactionsUpdated;

public:
    mutable CCriticalSection cs;
    std::map<uint256, CTxMemPoolEntry> mapTx;
    std::map<COutPoint, CInPoint> mapNextTx;
};
```

内存池中包含了2个MAP。mapTx是交易项、哈希的MAP，保存交易项。mapNextTx是输入、输出的MAP，保存输入、输出的对应关系。添加、移除交易时2个MAP都要操作。

内存池中统计交易更新次数（nTransactionsUpdated），添加、移除交易时，更新次数。

添加交易内存池项时，可以指定是否进行正常检查（fSanityCheck）。默认是不检查，但可以通过参数”-checkmempool”修改，如果没有指定参数”-checkmempool”，则根据是否是回归测试模式决定，如果是回归测试模式，则进行正常检查；否则不进行检查。把网络接收的交易添加到内存池时，不进行正常检查。

交易项的检查主要是检查内存池的mapTx、mapNextTx与指定的CCoinsViewCache这3者中的交易、输入、输出、币数量之间是否对应。详细过程见类CTxMemPool的check函数。

二、网络接收交易信息

节点接收到”tx”命令时，把交易添加到内存池中（AcceptToMemoryPool）。

先对交易进行检查，然后构造交易内存池项，再添加到交易内存池中，最后同步钱包。

满足以下几点才能添加到内存池中：

1. 交易是有效的（CheckTransaction函数）。
2. 交易不能是Coinbase类型。
3. 如果网络是MAIN类型，则交易必须是标准交易；如果是测试网络、回归测试，则交易可以不是标准的。
4. 交易不在交易内存池中。
5. 交易的输入不在交易内存池中。
6. 交易不在币视图内存池中。
7. 交易的输入的输出须在币视图内存池中。
8. 交易的输入是有效的。
9. 如果网络是MAIN类型，则交易的输入须是标准的。
10. 如果限制免费交易，则交易费不能低于最小交易费（GetMinFee函数获取最小交易费）。

11. 当限制免费交易，交易费低于最小传播交易费时，必须限制交易的流量，即免费交易的大小不能超过免费传播最大值，默认是15*10*1000字节，可以通过参数”-limitfreerelay"修改。
12. 如果限制交易费最大值，则交易费必须小于最小传播交易费的10000倍（CTransaction::nMinRelayTxFee * 10000）。
13. 交易的输入不能有二次花费（CheckInputs）。

添加到内存池中不在进行交易的正常检查，直接添加。

初始化交易项时，交易费的计算公式是：

```
int64_t nFees = view.GetValueIn(tx)-tx.GetValueOut();
```

优先级的计算公式是：

```
double dPriority = view.GetPriority(tx, chainActive.Height());
```

高度是chainActive.Height()。

三、获取节点交易内存池数据

节点发送”mempool”命令获取交易内存池数据，接收方发送”inv"命令返回数据。

返回的数据是类CInv的数组，类CInv包含了类型、交易项的哈希值，类型默认是MSG_TX。每次最多发送50000个。

节点接收到”mempool”命令后，先获取交易内存池中的所有的交易项的哈希值（queryHashes函数），然后构造类CInv对象，最后发送出去。发送之前要在内存池中查找此哈希值是否存在，因为其他线程可能移除此哈希项，依然存在的项才发送。

交易精简版

系统定义了交易的精简版以及相应的视图。交易的精简版主要用于钱包发送币时选择发送哪些币。

一、结构

系统定义了5个结构用于交易精简版。

系统定义的币视图、币视图备份、币视图缓存、币视图数据库在代码中没有用到，虽然定义了相应的变量，但只是删除了，基本没用到。

1、币

交易精简版仅仅包含元数据、未花费的交易输出。

类CCoins的定义如下：

```
class CCoins
{
public:
    // whether transaction is a coinbase
    bool fCoinBase;

    // unspent transaction outputs; spent outputs are .IsNull(); spent outputs at the end of the array are dropped
    std::vector<CTxOut> vout;

    // at which height this transaction was included in the active block chain
    int nHeight;

    // version of the CTransaction; accesses to this value should probably check for nHeight as well,
    // as new tx version will probably only be introduced at certain heights
    int nVersion;
};
```

当花掉比特币时，检验币的有效性，然后把币的输出信息置空，移除位于输出信息末尾的输出，且构造交易输入回滚信息。

检验币的有效性时，检验以下2点：

1. 输出的索引值小于输出的大小。
2. 交易输出数组中输出索引值的项不为空。

2、币视图

系统定义了交易输出数据集的视图。类CCoinsView。

CCoinsView的函数多数是返回false，或许是代码没写完吧。

3、币视图备份

系统定义了币视图的备份视图。

类CCoinsViewBacked的定义如下：

```
class CCoinsViewBacked : public CCoinsView
{
protected:
   CCoinsView *base;
};
```

币视图备份的函数都是调用币视图的对应函数。

4、币视图缓存

系统定义了币视图的内存缓冲。

类CCoinsViewCache的定义如下：

```
class CCoinsViewCache : public CCoinsViewBacked
{
protected:
   uint256 hashBlock;                                  // 最佳区块哈希值
   std::map<uint256,CCoins> cacheCoins;                // 币映射
};
```

在初始化区块链时，创建币视图缓存。

币视图缓存中保存了最佳区块的哈希值。设置时直接保存区块的哈希值（SetBestBlock）。获取时，先获取hashBlock值，如果hashBlock为0，则从币视图中获取最佳区块（GetBestBlock）。

币视图缓存中保存了币映射。设置时以数组方式修改数组项（SetCoins）。获取时，先从币映射中获取，如果没有，则从币视图中获取（GetCoins），也可以保存到币映射中（FetchCoins）。

刷新币视图缓存时（Flush），强制修改币映射，且设置最佳区块，最后把币视图清空。

系统提供了接口获取交易的优先级（GetPriority）。

5、币视图内存池

系统定义了币视图内存池，用于把内存池中的交易引入到币视图中。

类CCoinsViewMemPool的定义如下：

```
class CCoinsViewMemPool : public CCoinsViewBacked
{
protected:
    CTxMemPool &mempool;            // 交易内存池
};
```

此类提供了2个接口。

1. GetCoins函数，获取指定交易id的币。
2. HaveCoins函数，判断指定的交易id是否在交易内存池、币视图中。

二、币视图数据库

系统定义了币视图数据库，把币视图中的币、最佳区块等数据保存到数据库中。数据库采用了google开发的开源的LevelDB数据库。

LevelDB数据库的官网是https://code.google.com/p/leveldb。

系统定义了3个类用于币视图数据库。

1、LevelDB包装器

系统定义了类CLevelDBWrapper，包装了LevelDB数据库的操作。

类CLevelDBWrapper的定义如下：

```
class CLevelDBWrapper
{
private:
    // custom environment this database is using (may be NULL in case of default environment)
    leveldb::Env *penv;

    // database options used
    leveldb::Options options;

    // options used when reading from the database
    leveldb::ReadOptions readoptions;

    // options used when iterating over values of the database
    leveldb::ReadOptions iteroptions;

    // options used when writing to the database
    leveldb::WriteOptions writeoptions;
```

```
    // options used when sync writing to the database
    leveldb::WriteOptions syncoptions;

    // the database itself
    leveldb::DB *pdb;
};
```

类CLevelDBWrapper中包含了读（Read）、写（Write）、擦除（Erase）、是否存在（Exists）、同步（Sync）等操作，这里不再一一分析。

2、LevelDB批量器

系统定义了类CLevelDBBatch，包装了LevelDB的WriteBatch，提供了批量写（Write）、批量擦除（Erase）操作，这里不再一一分析。

类CLevelDBBatch的定义如下：

```
class CLevelDBBatch
{
    friend class CLevelDBWrapper;

private:
    leveldb::WriteBatch batch;
};
```

3、币视图数据库

币视图数据库用于保存币视图的相关数据。

类CCoinsViewDB的定义如下；

```
class CCoinsViewDB : public CCoinsView
{
protected:
    CLevelDBWrapper db;
};
```

数据库文件默认保存在比特币的数据目录（GetDataDir()）下的"chainstate"文件夹中。

数据库缓存默认大小是100M，最小是4M，32位系统最大是1G，64位系统最大是4G，可以通过参数”-dbcache"设置数据库大小。数据库缓存的1/8为区块数据库大小。剩余的7/8中的1/2位币视图数据库大小。

在初始化区块链时，创建币视图数据库。

币视图数据库保存了币、最佳区块的数据。

最佳区块在数据库中以'B'为标识。设置最佳区块时（SetBestBlock），在数据库中保存区块哈希值。读取时从数据库中读取'B'标识的值（GetBestBlock）。

币在数据库中以 'c'为标识。设置币时（SetCoins），在数据库中保存标识、交易ID、币。读取时，读取数据库中带标识、交易ID相同的币值（GetCoins）。

币视图数据库可以批量写入币映射、最佳区块（BatchWrite）。

币视图数据库定义了币状态，定义如下：

```
struct CCoinsStats
{
    int nHeight;                    //  高度
    uint256 hashBlock;              //  最佳区块
    uint64_t nTransactions;         //  交易次数
    uint64_t nTransactionOutputs;   //  交易中输出总量
    uint64_t nSerializedSize;       //  币序列化总大小
    uint256 hashSerialized;         //  CHashWriter的哈希值
    int64_t nTotalAmount;           //  输出币的总量
};
```

获取币状态时（GetStats），读取数据库中所有的币数据，统计相关的数据。

4、数据键类型

数据库中的数据都是以键值对形式存储，每种键对应不同类型的数据。

键	类型
'b'	区块
'B'	最佳区块链
'c'	币
'f'	区块文件信息
'F'	标记
'l'（i大写）	最佳无效工作
'l'（L小写）	上一个区块文件
'R'	重建索引
't'	交易

钱包

钱包主要用于交易比特币，存储交易信息。存储采用伯克利数据库。

一、钱包数据库

钱包数据库分2层，底层定义了类CDBEnv、类CDB，包装了Berkeley DB的相关操作，在此之上定义了类CWalletDB，支持钱包、交易、账户等操作。

1、Berkeley DB封装

类CDBEnv封装了Berkeley DB的DbEnv，提供了数据库文件的打开（Open）、关闭（Close）、刷新（Flush）、移除（RemoveDb）、校验（Verify）等相关操作。

类CDBEnv的定义如下：

```
class CDBEnv
{
private:
    bool fDbEnvInit;
    bool fMockDb;
    boost::filesystem::path path;

    void EnvShutdown();

public:
    mutable CCriticalSection cs_db;
    DbEnv dbenv;
    std::map<std::string, int> mapFileUseCount;
    std::map<std::string, Db*> mapDb;
};
```

类CDB封装了Berkeley DB的Db，提供了Berkeley DB的访问，包含读（Read）、写（Write）、擦除（Erase）、判断是否存在（Exists）、版本读写（ReadVersion、WriteVersion）、交易开始（TxnBegin）、交易提交（TxnCommit）、交易取消（TxnAbort）等相关操作。

类CDB的定义如下：

```
class CDB
{
protected:
```

```
    Db* pdb;
    std::string strFile;
    DbTxn *activeTxn;
    bool fReadOnly;
};
```

2、钱包数据库

类CWalletDB提供了钱包数据库的访问，支持对名称、目的、交易、key、脚本、最佳区块、key池、设置、版本、账户、目标数据等相关的读写、擦除、加载、恢复操作。

类CWalletDB的定义如下：

```
class CWalletDB : public CDB
{
public:
    CWalletDB(std::string strFilename, const char* pszMode="r+") :
CDB(strFilename.c_str(), pszMode)
    {
    }
private:
    CWalletDB(const CWalletDB&);
    void operator=(const CWalletDB&);
};
```

钱包中的数据都是以key-value方式存储的，key类型定义如下：

类型	key
账户	"acc"
账户项	"acentry"
最佳区块	"bestblock"
加密密钥	"ckey"
脚本	"cscript"
默认密钥	"defaultkey"
密钥	"key"
密钥信息	"keymeta"
最小版本号	"minversion"
主密钥	"mkey"
名称	"name"

类型	key
下一个order位置	"orderposnext"
密钥池	"pool"
目的	"purpose"
设置	"setting"
交易	"tx"
钱包密钥	"wkey"

在应用程序初始化钱包时，加载钱包（LoadWallet函数），先判断版本号，版本号小于等于CLIENT_VERSION，然后从当前光标位置开始读取钱包中的key、value是否匹配，最后更新版本号。

钱包备份（BackupWallet函数），复制钱包文件到指定目录。只有空闲的钱包数据库文件才能复制，复制之前，需要刷新日志数据到文件中。

钱包恢复（Recover函数），把旧的钱包的Key、Value数据写入新的钱包文件。先把现有的钱包文件改名，名称格式为：wallet.时间.bak，然后获取旧的钱包文件中的Key、Value信息（Salvage函数），创建新的钱包文件，把数据写入新的钱包文件。

读取键值时（ReadKeyValue函数），根据key的类型，把数据添加到钱包相应的数据中。

交易重新排序（ReorderTransactions函数），把钱包中的交易、账号项添加到multimap中，按时间排序，逐次写入钱包数据库中。交易的时间是nTimeReceived，帐号项的时间是nTime。

获取账号时（ListAccountCreditDebit函数），读取数据库中的类型为"acentry"的数据。

刷钱包数据库线程（ThreadFlushWalletDB函数），应用程序初始化时，最后创建此线程，此线程只能创建一次，线程名为"bitcoin-wallet"。默认是刷钱包，但可以通过参数"-flushwallet"指定是否刷钱包数据库。此线程每隔500毫秒循环一次。当钱包数据有更新，且距离上一次更新时间超过2，且钱包数据库文件引用次数为0时，关闭数据库文件从而刷新数据，

二、钱包

1、钱包交易

类CMerkleTx包含了链接到区块链的merkle分支。

类CMerkleTx的定义如下：

```
class CMerkleTx : public CTransaction
{
```

```
public:
    uint256 hashBlock;
    std::vector<uint256> vMerkleBranch;
    int nIndex;

    // memory only
    mutable bool fMerkleVerified;
};
```

Merkle交易的深度计算公式：

```
Depth = chainActive.Height() - pindex->nHeight + 1;
```

Merkle交易的成熟度最小是0，计算公式：

```
Maturity = (COINBASE_MATURITY+1) – depth;
```

类CMerkleTx封装了AcceptToMemoryPool函数，当提交钱包交易时，调用CMerkleTx的AcceptToMemoryPool函数添加到内存池中，默认需要检查手续费，手续费不可低于最小值。

当添加交易到钱包时，需要设置钱包交易的vMerkleBranch，vMerkleBranch为区块的vMerkleBranch。

钱包交易类CWalletTx继承了类CMerkleTx，包含了交易附加的很多只有所有者在意的信息，主要包含未记录的需要链接区块链的交易。

类CWalletTx的定义如下：

```
class CWalletTx : public CMerkleTx
{
private:
    const CWallet* pwallet;

public:
    std::vector<CMerkleTx> vtxPrev;
    mapValue_t mapValue;
    std::vector<std::pair<std::string, std::string> > vOrderForm;
    unsigned int fTimeReceivedIsTxTime;
    unsigned int nTimeReceived;  // time received by this node
    unsigned int nTimeSmart;
    char fFromMe;
    std::string strFromAccount;
    std::vector<char> vfSpent;    // which outputs are already spent
    int64_t nOrderPos;            // position in ordered transaction list
```

```
    // memory only
    mutable bool fDebitCached;
    mutable bool fCreditCached;
    mutable bool fImmatureCreditCached;
    mutable bool fAvailableCreditCached;
    mutable bool fChangeCached;
    mutable int64_t nDebitCached;
    mutable int64_t nCreditCached;
    mutable int64_t nImmatureCreditCached;
    mutable int64_t nAvailableCreditCached;
    mutable int64_t nChangeCached;
};
```

钱包交易中包含了指向钱包的指针（CWallet* pwallet），绑定钱包或者初始化钱包交易时设置此地址。

钱包交易中包含了用于内存的一些交易数据，借款数量（nDebitCached）、贷款数量（nCreditCached）、不成熟的贷款数量（nImmatureCreditCached）、可用的贷款数量（nAvailableCreditCached）、变更数量（nChangeCached），这些数据是直接从钱包中获取的，其中nAvailableCreditCached是钱包交易中的未花费的输出的贷款数量的总和，初始化钱包交易时这些数据设置为0，这些数据仅仅需要获取一次，但绑定钱包时这些数据需要重新获取。

如果钱包交易的借款数量大于0，则此交易信息是来源于自己（IsFromMe函数）。

当获取钱包的余额时，需要判断交易是否是可信的（IsTrusted函数），判断方法如下：

1. 如果交易不是最后的交易，则是不可信的。
2. 交易在主链中的深度大于等于1时，是可信的；小于0时，不可信。
3. 交易不是来源于自己，则是不可信的。
4. 交易的所有输入是来源于自己，且在内存池中，才是可信的，否则不可信。

钱包交易可以直接调用钱包数据库的写操作写入硬盘（WriteToDisk函数）。

钱包时间优先取值nTimeSmart，如果nTimeSmart为0，则取值交易的接收时间（nTimeReceived）。

传播钱包交易时（RelayWalletTransaction函数），交易不能是CoinBase类型的，且交易在主链的深度为0。

获取钱包交易中的冲突数据时（GetConflicts函数），直接从钱包中获取，且移除自己这一项。

获取钱包交易中的请求数量时（GetRequestCount），分2种情况：

1. 钱包交易是CoinBase类型，则从钱包中获取hashBlock的请求数量，如果不存在，则返回-1。

2. 钱包交易不是CoinBase类型，则从钱包中获取GetHash()的请求数量，如果数量为0，则从钱包中获取hashBlock的请求数量，如果不存在，则返回1。

获取账户总额时（GetAccountAmounts函数），需要计算发送数量、接收数量、手续费。

手续费的计算公式是：nFee = GetDebit()-GetValueOut()。

当借款数量（nDebit）大于0时，统计交易输出项的值（txout.nValue）总和就是发送总额。

统计交易输出中属于自己的项的值（txout.nValue）总和就是接收总额，

2、密钥商店

密钥分2种，未加密的、加密的，类CBasicKeyStore保存了未加密的密钥，类CCryptoKeyStore中保存了加密的密钥，类CCryptoKeyStore中的密钥源于类CBasicKeyStore。

类CBasicKeyStore的定义如下：

```
class CBasicKeyStore : public CKeyStore
{
protected:
    KeyMap mapKeys;
    ScriptMap mapScripts;
};
```

公钥保存在mapKeys中，添加公钥就是保存在mapKeys中（AddKeyPubKey函数），获取公钥就是获取mapKeys中的数据（GetKeys函数），可以获取指定公钥ID的公钥（GetKey函数）。

脚本保存在mapScripts中，添加脚本就是添加到mapScripts中（AddCScript函数），获取脚本时（GetCScript函数），从mapScripts中获取指定脚本ID的脚本。

类CCryptoKeyStore的定义如下：

```
class CCryptoKeyStore : public CBasicKeyStore
{
private:
    CryptedKeyMap mapCryptedKeys;

    CKeyingMaterial vMasterKey;

    // if fUseCrypto is true, mapKeys must be empty
    // if fUseCrypto is false, vMasterKey must be empty
```

```
    bool fUseCrypto;
};
```

成员fUseCrypto标识是否加密，初始化时是不加密的，但可以设置加密（SetCrypted函数）。

添加加密密钥时（AddCryptedKey函数），添加到mapCryptedKeys中。

添加公钥时（AddKeyPubKey函数），未加密时添加到类CBasicKeyStore的mapKeys中；加密时，先把公钥加密（EncryptSecret函数），再添加到mapCryptedKeys中。

加密密钥时（EncryptKeys函数），加密类CBasicKeyStore的mapKeys（EncryptSecret函数），然后添加到mapCryptedKeys中。

获取密钥时（GetKeys），如果未加密，则从类CBasicKeyStore的mapKeys中获取；如果已加密，则从mapCryptedKeys中获取。可以获取指定密钥ID的加密密钥（GetPubKey函数）。也可以获取指定密钥ID的解密（DecryptSecret函数）后的密钥（GetKey函数）。

可以对其进行加锁、解锁。加锁就是设置成加密模式，把vMasterKey清空。解锁时设置加密模式，且把mapCryptedKeys中的Key解密，设置vMasterKey。

3、钱包类

钱包类封装了钱包相关的各种操作，主要包含公钥、版本号、地址簿、交易、余额、加密钱包、加载钱包等。

类CWallet的定义如下：

```
class CWallet : public CCryptoKeyStore, public CWalletInterface
{
private:
    CWalletDB *pwalletdbEncryption;

    // the current wallet version: clients below this version are not able to load the wallet
    int nWalletVersion;

    // the maximum wallet format version: memory-only variable that specifies to what version this wallet may be upgraded
    int nWalletMaxVersion;

    int64_t nNextResend;
    int64_t nLastResend;
```

```
public:
    /// Main wallet lock.
    /// This lock protects all the fields added by CWallet
    ///   except for:
    ///      fFileBacked (immutable after instantiation)
    ///      strWalletFile (immutable after instantiation)
    mutable CCriticalSection cs_wallet;

    bool fFileBacked;
    std::string strWalletFile;

    std::set<int64_t> setKeyPool;
    std::map<CKeyID, CKeyMetadata> mapKeyMetadata;

    typedef std::map<unsigned int, CMasterKey> MasterKeyMap;
    MasterKeyMap mapMasterKeys;
    unsigned int nMasterKeyMaxID;

    std::map<uint256, CWalletTx> mapWallet;
    int64_t nOrderPosNext;
    std::map<uint256, int> mapRequestCount;

    std::map<CTxDestination, CAddressBookData> mapAddressBook;

    CPubKey vchDefaultKey;

    std::set<COutPoint> setLockedCoins;

    int64_t nTimeFirstKey;

    typedef std::pair<CWalletTx*, CAccountingEntry*> TxPair;
    typedef std::multimap<int64_t, TxPair > TxItems;
};
```

1、钱包版本

钱包中包含2个版本号，nWalletVersion是当前钱包版本号，低于此版本号的客户端不能够加载此钱包。nWalletMaxVersion是最大的钱包版本号，仅仅保存在内存中，指定了钱包可以升级到的最大版本号。

获取版本时（GetVersion函数），获取的是钱包当前版本号（nWalletVersion）。

设置最大版本时（nWalletMaxVersion函数），保存到nWalletMaxVersion中，但新版本号必须大于钱包的当前版本号。

设置最小版本号（SetMinVersion函数），最小版本号必须大于当前版本号，当前版本号修改为最小版本号，如果最小版本号大于大版本号，则修改最大版本号，最后修改钱包数据库中的最小版本号。

加载最小版本号（LoadMinVersion函数），设置nWalletVersion为指定的版本，如果指定的版本大于nWalletMaxVersion，则设置nWalletMaxVersion为指定的版本。

2、地址簿

钱包中的地址簿保存了地址相关的信息（std::map<CTxDestination, CAddressBookData> mapAddressBook），包含了交易地址、地址附加信息。

交易地址的定义如下：

```
typedef boost::variant<CNoDestination, CKeyID, CScriptID> CTxDestination;
```

地址附加信息类CAddressBookData的定义如下：

```
class CAddressBookData
{
public:
    std::string name;
    std::string purpose;

    CAddressBookData()
    {
        purpose = "unknown";
    }

    typedef std::map<std::string, std::string> StringMap;
    StringMap destdata;
};
```

地址相关的信息不仅仅保存在钱包的地址簿中，且保存在钱包数据库中。

地址簿中的地址是可以修改的，修改类型分3种：

序号	类型	描述
0	CT_NEW,	添加地址
1	CT_UPDATED	更新地址
2	CT_DELETED	删除地址

对于地址的各种操作会发送通知信号。

设置钱包（SetAddressBook函数），先从地址簿中查询地址，如果已经存在，则更新地址信息，如果不存在则添加。然后通知地址簿修改，最后把地址的purpose、name写入到钱包数据库中。

删除地址簿中的地址（DelAddressBook函数），先从地址簿中移除地址信息，然后通知地址簿删除，最后从钱包数据库中移除地址的purpose、name。如果钱包文件已经备份，则从钱包数据库中移除地址目的数据（"destdata"）。

地址簿的附加信息含有目的数据（destdata），目的数据在钱包数据库中也保存了一份。添加目的数据时（AddDestData函数），先插入到地址簿中，再写入钱包数据库中。擦除时（EraseDestData函数），从地址簿中移除，再从钱包数据库中擦除。可以从地址簿中获取指定交易地址的目的数据（GetDestData函数）。加载目的数据（LoadDestData函数），把指定的目标数据插入到地址簿中。（感觉逻辑不对）

3、公钥

钱包中定义了公钥池（setKeyPool）、公钥元数据（mapKeyMetadata）。公钥池保存公钥。公钥元数据保存了公钥的相关数据。

创建新的公钥池（NewKeyPool函数）时，先清除钱包数据库中的公钥，然后把钱包中的公钥池（setKeyPool）清空。再创建新的公钥，写入钱包数据库，且插入到钱包中的公钥池（setKeyPool）中。

公钥池的公钥数量默认是100个，可以通过设置参数”-keypool"来指定。

调用GetKeyPoolSize函数可以获取公钥池的大小。

应用程序初始化时，从公钥池中获取公钥，设置为钱包的默认公钥（SetDefaultKey函数），默认公钥保存到vchDefaultKey中，且写入到钱包数据库中。

从公钥池中获取公钥（GetKeyFromPool函数）时，先从公钥池中获取保留的公钥，如果失败，则生成新的公钥；如果成功，则从公钥池中移除指定索引的公钥。

生成新的公钥（GenerateNewKey函数）时，首先检验钱包版本是否支持压缩公钥，0.6.0版本以上的钱包支持压缩公钥，如果支持压缩公钥，则设置钱包最小版本为0.6.0。再生成随机种子、公钥，把公钥添加到钱包中。然后获取当前时间，生成元数据，保存到钱包的mapKeyMetadata中。最后添加到公钥商店中。

钱包的公钥商店保存在父类CCryptoKeyStore的mapCryptedKeys中。

添加公钥到商店时（AddKeyPubKey函数），先把公钥添加到钱包的公钥商店中，如果公钥商店没有加密，则把公钥写入钱包数据库。

加载公钥（LoadKey函数）时，添加公钥到钱包的公钥商店中，但不保存到磁盘中。

加载公钥元数据（LoadKeyMetadata函数）时，设置mapKeyMetadata中以指定公钥id为索引的项的值为指定的元数据，如果元数据的创建时间（nCreateTime）小于nTimeFirstKey，则更新nTimeFirstKey为元数据的创建时间。

添加保留的公钥（AddReserveKey函数）时，获取公钥池的顶部位置，写入钱包数据库，插入到钱包中的公钥池（setKeyPool）中。

在公钥池中保留公钥（ReserveKeyFromKeyPool函数）时，生成公钥，然后在钱包不加锁的情况下上升到公钥池的顶部，再移除公钥池开始的公钥，最后读取公钥池开始的公钥，检验公钥池中是否含有公钥。

移除公钥（KeepKey函数）时，如果指定了文件备份，则从钱包数据库中移除指定了索引的公钥池。

返回公钥池（ReturnKey函数）时，把指定的索引插入到公钥池中。

获取所有的保留公钥（GetAllReserveKeys函数）时，遍历公钥池，读取钱包数据库的公钥池，检验是否含有公钥池的id的公钥，然后插入到setAddress中。

上升到公钥池顶部（TopUpKeyPool函数）时，首先获取指定的需要上升到公钥池的位置，如果为0，则是公钥池的顶部。然后从公钥池未填充位置开始，生成新的公钥，写入钱包数据库，插入到公钥池。公钥池不加锁时才能上升。

获取最早的公钥池的时间（GetOldestKeyPoolTime函数）时，先从公钥池中获取保留的公钥，如果失败，则返回当前时间；如果成功，则返回公钥池的时间。

获取公钥创建时间（GetKeyBirthTimes函数）时，先获取公钥元数据（mapKeyMetadata）中的创建时间不为0的公钥id、创建时间，然后获取所有的公钥，再从交易、区块中统计受到影响的公钥，最后把受到影响的公钥的时间减少2个小时。

加载加密公钥（LoadCryptedKey函数）时，把公钥添加到加密公钥商店中。

添加加密公钥（AddCryptedKey函数）时，先把公钥添加到加密公钥商店中，再保存到钱包数据库中。

4、交易

给指定地址发送货币（SendMoneyToDestination函数）时，先检验参数有效性，然后分解地址，再把货币发送出去。

检验2点：

1. 货币发送数量必须大于0。
2. 货币发送数量与交易费的和不能大于钱包余额。

发送货币（SendMoney函数），先按指定的发送参数构造成交易，然后提交交易。发送时钱包必须处于未加锁状态。

创建交易（CreateTransaction函数）时，构造发送货币请求后再创建交易。首先统计发送货币的总额，总额不能小于0，且每一笔发送请求的数额也不能小于0。然后对新的交易绑定钱包。再把发送交易中的输出添加到新的交易中。然后从钱包中选择货币。计算找零，如果找零大于0，且大于最小传播交易费，则从私钥池中获取保留的私钥，构造新的输出，添加到交易中；如果小于最小传播交易费，则不构造新的交易输出。把选出的货币的输入添加到新的交易中。再给新的交易打签名。然后计

算交易的序列化大小，交易大小必须小于交易最大值（100000字节）。最后检验是否含有足够的交易费，如果不够，则重新构造交易。

从钱包中选择币（SelectCoins函数）时，首先从钱包的币中选出能花费的币，如果上层想控制币时，则返回所有的币，否则从币中按从小到大的顺序选出合适的币。

选择能花费的币（AvailableCoins函数）时，遍历钱包中的交易信息，获取能花费的交易的币。能花费的交易须满足以下几点：

1. 交易是最后一笔交易。
2. 交易是可信的。
3. 交易不是CoinBase类型的。
4. 币在主链中的深度不能小于0。

按最小配置选择币（SelectCoinsMinConf函数）时，从币中选择一些币，满足以下2点：

1. 如果币是自己的，交易的深度最小是1。
2. 如果币是接收的，交易的深度最小是6.

如果其中某个币的值满足要求，则直接获取此币；否则添加到币数组中。且记录最小大于目标值的币。

如果比数组中的币总额满足要求，则返回此币数组中的币。如果不够，则返回币数组中的币以及最小大于目标值的币。

如果币数组中的币总额大于目标值，则先把币数组中的币排序，然后选择最优方案，如果选择成功，则获取最优方案中的币；否则获取最优方案中的币已经最小大于目标值的币。

选择币的最优子集（ApproximateBestSubset函数）时，从币数组中随机选择一些币，总额满足目标值，且总额是多次选择后最小的，最多选择1000次。随机的目的是保护隐私，阻止币退化。

提交交易（CommitTransaction函数）时，先把钱包数据库中的公钥池的公钥移除，重新生成新的公钥，再把交易添加到钱包中（AddToWallet函数），然后通知货币花费，再把钱包中的以交易哈希值为索引的请求数量（mapRequestCount）置为0，把交易添加到内存池中，最后把交易传播下去。

添加交易到钱包（AddToWallet函数）时，先获取交易的哈希值，把交易添加到钱包的mapWallet中，且添加到已花费中，如果是新的交易，则初始化交易相关的数据。如果是更新交易，则合并交易。再把交易写入磁盘。然后把交易标记为脏的。再通知UI有新交易生成或者更新交易。最后执行钱包交易的扩展脚本，扩展脚本默认为空，可以通过参数"-walletnotify"设置。

添加到已花费中（AddToSpends函数）时，根据指定的交易ID获取钱包中的交易，遍历交易的输入，把输入的上一个输出添加到钱包的已花费交易中（mapTxSpends），最后同步钱包的元数据。

同步元数据（SyncMetaData函数），把相同区域内的钱包交易的元数据设置最老的交易的元数据。首先遍历区域内的钱包交易，找到nOrderPos最小的交易。然后把区域内的所有交易的相关数据设置为nOrderPos最小的交易的数值，不修改区域

内与nOrderPos最小交易相同的交易。主要修改交易的mapValue、vOrderForm、nTimeSmart、fFromMe、strFromAccount。

判断交易是否已花费（IsSpent函数）时，先获取制定哈希值的输出，再获取已花费交易的区域，如果区域中的交易ID存在钱包的交易中，且交易在主链的深度大于0，则此交易是已花费的。

标识交易为脏（MarkDirty函数）时，把钱包中的所有交易设置为脏的。

排序交易项（OrderedTxItems函数）时，先把钱包中的所有交易添加到排序的multimap中，然后获取钱包数据库中的账户，把账户也添加到排序的multimap中。

重新发送钱包交易（ResendWalletTransactions函数）时，把钱包中的交易传报下去，但需要满足以下几点：

1. 每隔30分钟重新发送一次。
2. 上一次重新发送的时间小于最优区块的接收时间。
3. 钱包交易的接收时间距离最优区块的接收时间超过5分钟。

传播发送钱包交易（RelayWalletTransaction函数）时，传播的钱包交易不能是CoinBase类型，且交易在主链的深度必须为0。

同步交易（SyncTransaction函数）时，把交易添加到钱包中，且把钱包交易中以交易所有的输入的输出为索引的项标识为脏。

添加交易到钱包中（AddToWalletIfInvolvingMe函数）时，先检验交易是否在钱包中，如果存在，则只能更新交易。再把交易构造成钱包交易，添加到钱包中。如果指定了区块，则设置钱包交易的Merkle分支。

擦除交易（EraseFromWallet函数）时，先把交易从钱包交易中移除，然后删除钱包数据库中的交易。

应用程序初始化时，移除指定路径的钱包中的交易（ZapWalletTx函数），如果需要重写，则重写钱包文件，重写时跳过”\x04pool"字符串。

应用程序初始化时，扫描钱包交易（ScanForWalletTransactions函数），从指定的区块索引开始读取区块，把区块中的交易添加到钱包中，如果已经存在，则更新交易。

应用程序初始化时，重新接受钱包交易（ReacceptWalletTransactions函数），把钱包交易中的交易添加到内存池中，添加的交易不能是CoinBase类型，且在主链中的深度小于0。

当获取钱包交易中与指定交易含有相同输出的交易（GetConflicts函数）时，先按指定交易ID获取钱包中的交易（mapWallet），然后获取也在已花费交易（mapTxSpends）中的交易的输出。

递增下一个操作位置（IncOrderPosNext函数）时，钱包中的nOrderPosNext保存了下一个读写位置，初始化钱包时nOrderPosNext设置为0。nOrderPosNext递增后，把此位置写入钱包数据库。

更新钱包花费（WalletUpdateSpent函数），当签名校验成功后，证明输出已经花费了，更新钱包的花费状态。遍历交易的输入，找到输入的输出的钱包交易，如果交易是没花费的，则设置为已花费，把交易写入磁盘，通知交易改变。

5、加密钱包

钱包加密（EncryptWallet函数）时，先生成随机种子、私钥，设置私钥的短语，加密私钥，把加密后的私钥保存到mapMasterKeys中，加密私钥，重写钱包数据库。如果钱包文件备份，则把私钥写入文件中。再设置钱包最小版本为FEATURE_WALLETCRYPT，钱包从FEATURE_WALLETCRYPT版本开始支持加密。最后发送钱包加密的通知。

钱包只能加密一次，如果已经加密，则不能再加密。

修改钱包短语（ChangeWalletPassphrase函数）时，先按旧的短语解密，再按新的短语加密，最后把新的主要公钥保存到钱包数据库中。

加密钱包时，需要解锁钱包（Unlock函数），先设置钱包短语，再解密公钥，最后解锁。

6、余额

计算余额（GetBalance函数）时，钱包余额是钱包交易中的所有的可信交易的有效的借款的总和。

计算未确认余额（GetUnconfirmedBalance函数）时，钱包未确认余额是钱包交易中的某些交易的有效借款的总和。

交易是以下2种：

1. 交易不是最终交易。
2. 交易不可信，且交易在主链的深度为0。

计算不成熟的余额（GetImmatureBalance函数）时，钱包不成熟余额是钱包交易中的所有的交易的不成熟的借款的总和。

获取交易输入的借款（GetDebit函数）时，从钱包交易中寻找交易输入的输出，如果交易输出是自己的，则获取交易输出的值。

检验交易输入是否时自己的（IsMine函数）时，从钱包交易中寻找交易输入的输出，如果交易输出是自己的，则是自己的。

获取交易输出的贷款（GetCredit函数）时，先检验交易输出的值是否在合理的范围，如果交易输出是自己的，则返回输出的数额；否则返回0。

检验交易输出是否时自己的（IsMine函数）时，直接调用IsMine函数校验。

获取交易输出的改变（GetChange函数）时，先检验交易输出是否自己的，如果交易输出发生了改变，则返回输出的数额；否则返回0。

检验交易输出是否发生改变（IsChange函数）时，扩展交易输出的地址，如果不在地址簿中，则发生改变。

获取交易的贷款（GetDebit函数）时，统计交易的所有输入的贷款数额，且检验每一项的贷款是否在允许范围之内。

获取交易的借款（GetCredit函数）时，统计交易的所有输出的借款数额，且检验每一项的借款是否在允许范围之内。

获取交易的改变（GetChange函数）时，统计交易的所有输出的改变数额，且检验每一项的改变是否在允许范围之内。

获取地址余额（GetAddressBalances函数）时，统计钱包中每个地址的余额。遍历钱包交易，统计满足某些条件的交易的输出的数额。

条件如下：

1、交易必须是最后的。

2、交易需要经过确认的。

3、交易不能是CoinBase类型。

4、交易在主链的深度小于某个值，如果交易是自己发出的，此值是0；否则是1。

5、交易的输出是自己的。

检验交易是否是自己的（IsMine函数）时，校验交易的输出是否是自己的。

校验交易是否是自己发出的（IsFromMe函数）时，获取交易的贷款，如果大于0，则是自己发出的。

7、交易输出的加锁、解锁

钱包含有交易输出的集合（std::set<COutPoint> setLockedCoins），包含了需要加锁的交易输出。

锁币时（LockCoin函数），把指定的交易输出添加到setLockedCoins中。

解锁时（UnlockCoin函数），从setLockedCoins中移除指定的交易输出。

解锁所有的币时（UnlockAllCoins函数），清空setLockedCoins。

检验币是否加锁时（IsLockedCoin函数），判断setLockedCoins中是否含有指定的输出。

获取所有的加锁的币时（ListLockedCoins函数），获取setLockedCoins中的所有的交易输出。

8、脚本

钱包的脚本保存在父类CCryptoKeyStore的父类CBasicKeyStore的mapScripts中。钱包提供了2个脚本接口。

添加脚本（AddCScript函数），先把脚本添加到mapScripts中，再保存到钱包数据库中。

加载脚本（LoadCScript函数），只把脚本添加到mapScripts中。

9、其他

加载钱包（LoadWallet函数）时，加载指定路径的钱包文件，路径保存在钱包的strWalletFile中。加载后，如果需要重写，则重写钱包文件，重写时跳过"\x04pool"字符串。最后检验默认公钥的有效性。

设置最佳区块（SetBestChain函数），把最佳区块保存到钱包数据库中。

获取账户地址（GetAccountAddresses函数），遍历钱包的地址簿，寻找指定账户名称的地址。

获取地址组（GetAddressGroupings函数），统计钱包交易中的输入、输出的扩展地址，输入必须是自己的，输出必须是有改变的。

区块树数据库

一、区块树数据库

系统定义了区块树数据库，包含了区块、交易索引等信息。

类CBlockTreeDB的定义如下：

```
class CBlockTreeDB : public CLevelDBWrapper
{
public:
    CBlockTreeDB(size_t nCacheSize, bool fMemory = false, bool fWipe = false);
private:
    CBlockTreeDB(const CBlockTreeDB&);
};
```

数据库文件默认保存在比特币的数据目录（GetDataDir()）下的”blocks\index"文件夹中。

数据库缓存默认大小是100M，最小是4M，32位系统最大是1G，64位系统最大是4G，可以通过参数”-dbcache”设置数据库大小。区块树数据库大小为数据库缓存的1/8。

在在初始化区块链时，创建区块树数据库。

在数据库中写入区块索引时以’b'为标识。

写入最优有效工作时，以’l’为标识。

读取、写入区块文件信息时，以’f’为标识。

读取、写入上一个区块文件时，以’l’为标识。

读取、写入重建索引时，以’R’为标识。

读取、写入交易索引时，以’t'为标识。

读取、写入标记时，以’F'为标识。

系统定义了区块索引映射（map<uint256, CBlockIndex*> mapBlockIndex），在初始化区块链时，从数据库中加载区块索引（LoadBlockIndexGuts）到区块索引映射中。

从数据库中读取标识为’b’的区块索引数据，创建区块索引，进行POW工作证明检查，最后插入到区块索引映射中。