设计说明书

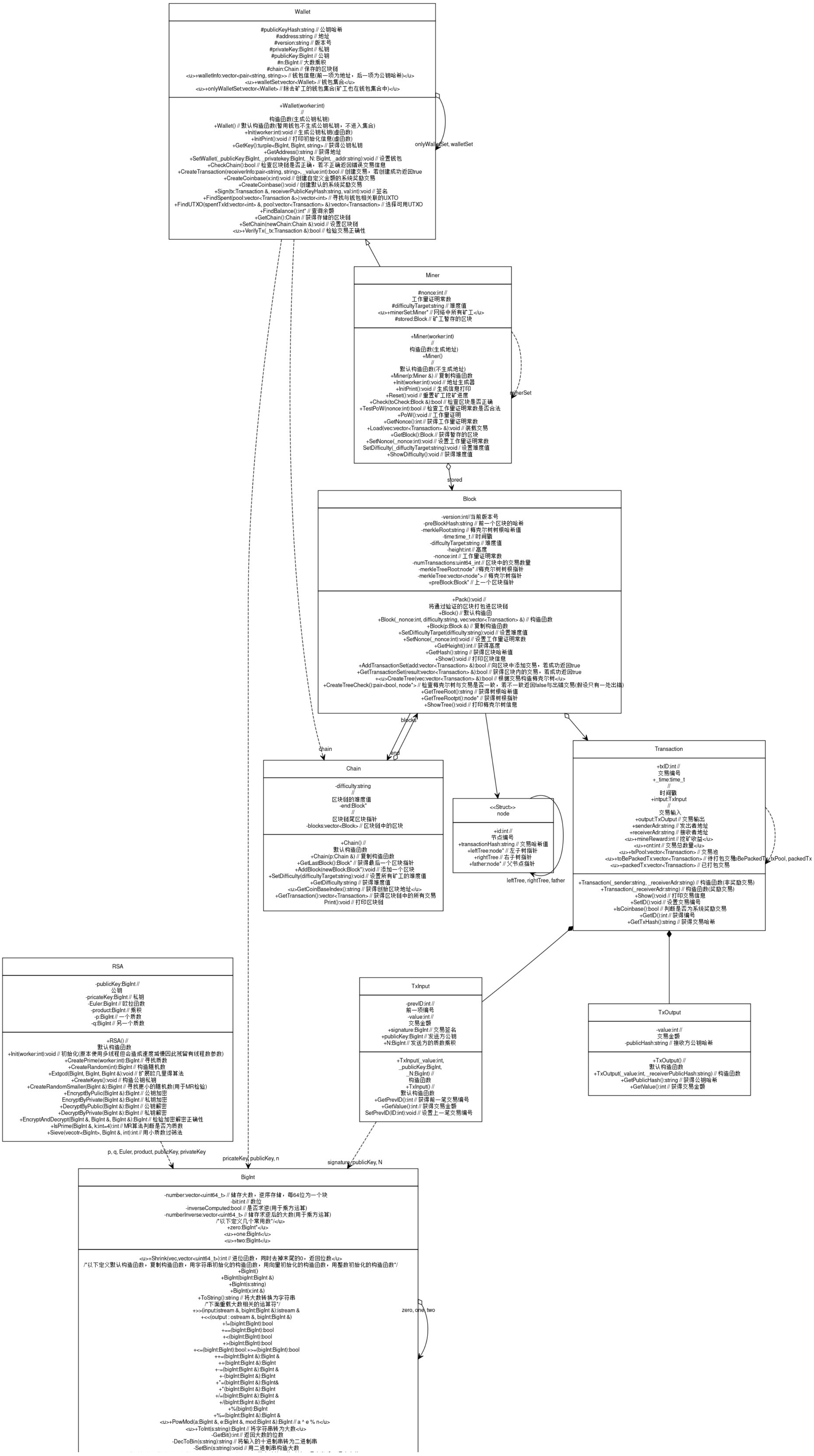
类设计

类功能说明

1. BigInt类：大数类，实现任意大正整数的运算
2. RSA类：实现加密解密功能
3. （哈希函数类提供公共接口）
4. Wallet类：钱包类，实现交易等功能
5. Miner类：矿工类，继承钱包，实现挖矿等功能
6. Block类：区块类，记录区块头区块体信息
7. Chain类：区块链类，链表储存区块
8. Transaction类：交易类，记录交易信息
9. txInput类：记录交易输入信息
10. txOutput类：记录交易输出信息

类图

- svg/jpg图请参考[这里]



Wallet
#publicKeyHash:string // 公钥哈希
#address:string // 地址
#version:string // 版本号
#privateKey:BigInt // 私钥
#publicKey:BigInt // 公钥
#n:BigInt // 大数乘积
#chain:Chain // 保存的区块链
<u>+walletInfo:vector<pair<string, string>> // 钱包信息(前一项为地址，后一项为公钥哈希)</u>
<u>+walletSet:vector<Wallet> // 钱包集合</u>
<u>+onlyWalletSet:vector<Wallet> // 除去矿工的钱包集合(矿工也在钱包集合中)</u>
+Wallet(worker:int)
//
构造函数(生成公钥私钥)
+Wallet() // 默认构造函数(暂用钱包不生成公钥私钥，不进入集合)
+Init(worker:int):void // 生成公钥私钥(虚函数)
+InitPrint():void // 打印初始化信息(虚函数)
+GetKey():turple<BigInt, BigInt, string> // 获得公钥私钥
+GetAddress():string // 获得地址
+SetWallet(_publicKey:BigInt, _privatekey:BigInt, _N: BigInt, _addr:string):void // 设置钱包
+CheckChain():bool // 检查区块链是否正确，若不正确返回错误交易信息
+CreateTransaction(receiverInfo:pair<string, string>, _value:int):bool // 创建交易，若创建成功返回true
+CreateCoinbase(x:int):void // 创建自定义金额的系统奖励交易
+CreateCoinbase():void / 创建默认的系统奖励交易
+Sign(tx:Transaction &, receiverPublicKeyHash:string, val:int):void // 签名
+FindSpent(pool:vector<Transaction &>):vector<int> // 寻找与钱包相关联的UXTO
+FindUTXO(spentTxId:vector<int> &, pool:vector<Transaction> &):vector<Transaction> // 选择可用UTXO
+FindBalance():int* // 查询余额
+GetChain():Chain // 获得存储的区块链
+SetChain(newChain:Chain &):void // 设置区块链
<u>+VerifyTx(_tx:Transaction &):bool // 检验交易正确性
onlyWalletSet, walletSet
Miner
#nonce:int //
工作量证明常数
#difficultyTarget:string // 难度值
<u>+minerSet:Miner* // 网络中所有矿工</u>
#stored:Block // 矿工暂存的区块
+Miner(worker:int)
//
构造函数(生成地址)
+Miner()
//
默认构造函数(不生成地址)
+Miner(p:Miner &) // 复制构造函数
+Init(worker:int):void // 地址生成器
+InitPrint():void // 生成信息打印
+Reset():void // 重置矿工挖矿进度
+Check(toCheck:Block &):bool // 检查区块是否正确
+TestPoW(nonce:int):bool // 检查工作量证明常数是否合法
+PoW():void // 工作量证明
+GetNonce():int // 获得工作量证明常数
+Load(vec:vector<Transaction> &):void // 装载交易
+GetBlock():Block // 获得暂存的区块
+SetNonce(_nonce:int):void // 设置工作量证明常数
SetDifficulty(_diffucltyTarget:string):void / 设置难度值
+ShowDifficulty():void // 获得难度值
minerSet
stored
Block
-version:int//当前版本号
-preBlockHash:string // 前一个区块的哈希
-merkleRoot:string // 梅克尔树树根哈希值
-time:time_t // 时间戳
-diffcultyTarget:string // 难度值
-height:int // 高度
-nonce:int // 工作量证明常数
-numTransactions:uint64_int // 区块中的交易数量
-merkleTreeRoot:node* //梅克尔树树根指针
-merkleTree:vector<node*> // 梅克尔树指针
+preBlock:Block* // 上一个区块指针
+Pack():void //
将通过验证的区块打包进区块链
+Block() // 默认构造函
+Block(_nonce:int, difficulty:string, vec:vector<Transaction> &) // 构造函数
+Block(p:Block &) // 复制构造函数
+SetDifficultyTarget(difficulty:string):void // 设置难度值
+SetNonce(_nonce:int):void // 设置工作量证明常数
+GetHeight():int // 获得高度
+GetHash():string // 获得区块哈希值
+Show():void // 打印区块信息
+AddTransactionSet(add:vector<Transaction> &):bool // 向区块中添加交易，若成功返回true
+GetTransactionSet(result:vector<Transaction> &):bool // 获得区块内的交易，若成功返回true
+<u>CreateTree(vec:vector<Transaction> &):bool // 根据交易构造梅克尔树</u>
+CreateTreeCheck():pair<bool, node*> // 检查梅克尔树与交易是否一致，若不一致返回false与出错交易(假设只有一处出错)
+GetTreeRoot():string // 获得树根哈希值
+GetTreeRootpt():node* // 获得树根指针
+ShowTree():void // 打印梅克尔树信息
blocks
chain
end
Chain
-difficulty:string
//
区块链的难度值
-end:Block*
//
区块链尾区块指针
-blocks:vector<Block> // 区块链中的区块
+Chain() //
默认构造函数
+Chain(p:Chain &) // 复制构造函数
+GetLastBlock():Block* // 获得最后一个区块指针
+AddBlock(newBlock:Block*):void // 添加一个区块
+SetDifficulty(difficultyTarget:string):void // 设置所有矿工的难度值
+GetDifficulty:string // 获得难度值
<u>GetCoinBaseIndex():string // 获得创始区块地址</u>
+GetTransaction():vector<Transaction> // 获得区块链中的所有交易
Print():void // 打印区块链
<<Struct>>
node
+id:int //
节点编号
+transactionHash:string // 交易哈希值
+leftTree:node* // 左子树指针
+rightTree // 右子树指针
+father:node* // 父节点指针
leftTree, rightTree, father
Transaction
+txID:int //
交易编号
+_time:time_t
//
时间戳
+intput:TxInput
//
交易输入
+output:TxOutput // 交易输出
+senderAdr:string // 发出者地址
+receiverAdr:string // 接收者地址
<u>+mineReward:int // 挖矿收益</u>
<u>+cnt:int // 交易总数量</u>
<u>+txPool:vector<Transaction> // 交易池
<u>+toBePackedTx:vector<Transaction> // 待打包交易
<u>+packedTx:vector<Transaction> // 已打包交易
toBePackedTx, txPool, packedTx
+Transaction(_sender:string, _receiverAdr:string) // 构造函数(非奖励交易)
+Transaction(_receiverAdr:string) // 构造函数(奖励交易)
+Show():void // 打印交易信息
+SetID():void // 设置交易编号
+IsCoinbase():bool // 判断是否为系统奖励交易
+GetID():int // 获得编号
+GetTxHash():string // 获得交易哈希
RSA
-publicKey:BigInt //
公钥
-pricateKey:BigInt // 私钥
-Euler:BigInt // 欧拉函数
-product:BigInt // 乘积
-p:BigInt // 一个质数
-q:BigInt // 另一个质数
+RSA() //
默认构造函数
+Init(worker:int):void // 初始化(原本使用多线程但会造成速度减慢因此残留有线程数参数)
+CreatePrime(worker:int):BigInt // 寻找质数
+CreateRandom(int):BigInt // 构造随机数
+Extgcd(BigInt, BigInt, BigInt &):void // 扩展欧几里得算法
+CreateKeys():void // 构造公钥私钥
+CreateRandomSmaller(BigInt &):BigInt // 寻找更小的随机数(用于MR检验)
+EncryptByPulic(BigInt &):BigInt // 公钥加密
EncryptByPrivate(BigInt &):BigInt // 私钥加密
+DecryptByPublic(BigInt &):BigInt // 公钥解密
+DecryptByPrivate(BigInt &):BigInt // 私钥解密
+EncryptAndDecrypt(BigInt &, BigInt &, BigInt &):BigInt // 检验加密解密正确性
+IsPrime(BigInt &, k:int=4):int // MR算法判断是否为质数
+Sieve(vecotr<BigInt>, BigInt &, int):int // 用小质数过筛法
TxInput
-prevID:int //
前一项编号
-value:int //
交易金额
+signature:BigInt // 交易签名
+publicKey:BigInt // 发送方公钥
+N:BigInt // 发送方的质数乘积
+TxInput(_value:int,
_publicKey:BigInt,
_N:BigInt) //
构造函数
+TxInput() //
默认构造函数
+GetPrevID():int // 获得前一笔交易编号
+GetValue():int // 获得交易金额
SetPrevID(ID:int):void // 设置上一笔交易编号
TxOutput
-value:int //
交易金额
-publicHash:string // 接收方公钥哈希
+TxOutput() //
默认构造函数
+TxOutput(_value:int, _receiverPublicHash:string) // 构造函数
+GetPublicHash():string // 获得公钥哈希
+GetValue():int // 获得交易金额
p, q, Euler, product, publicKey, privateKey
pricateKey, publicKey, n
signature, publicKey, N
BigInt
-number:vector<uint64_t> // 储存大数，逆序存储，每64位为一个块
-bit:int // 数位
-inverseComputed:bool // 是否求逆(用于乘方运算)
-numberInverse:vector<uint64_t> // 储存求逆后的大数(用于乘方运算)
/*以下定义几个常用数*/</u>
+zero:BigInt*</u>
<u>+one:BigInt</u>
<u>+two:BigInt</u>
<u>+Shrink(vec,vector<uint64_t>):int // 进位函数，同时去掉末尾的0，返回位数</u>
/*以下定义默认构造函数，复制构造函数，用字符串初始化的构造函数，用向量初始化的构造函数，用整数初始化的构造函数*/
+BigInt()
+BigInt(bigInt:BigInt &)
+BigInt(s:string)
+BigInt(x:int &)
+ToString():string // 将大数转换为字符串
/*下面重载大数相关的运算符*/
+>>(input:istream &, bigInt:BigInt &):istream &
+<<(output : ostream &, bigInt:BigInt &)
+!=(bigInt:BigInt):bool
+==(bigInt:BigInt):bool
+<(bigInt:BigInt):bool
+>(bigInt:BigInt):bool
+<=(bigInt:BigInt):bool:+>=(bigInt:BigInt):bool
++=(bigInt:BigInt &):BigInt &
++(bigInt:BigInt &):BigInt
+-=(bigInt:BigInt &):BigInt &
+-(bigInt:BigInt &):BigInt
+*=(bigInt:BigInt &):BigInt&
+*(bigInt:BigInt &):BigInt
+/=(bigInt:BigInt &):BigInt &
+/(bigInt:BigInt &):BigInt
+%(bigInt):BigInt
+%=(bigInt:BigInt &):BigInt &
<u>+PowMod(a:BigInt &, e:BigInt &, mod:BigInt &):BigInt // a ^ e % n</u>
<u>+ToInt(s:string):BigInt // 将字符串转为大数</u>
-GetBit():int // 返回大数的位数
-DecToBin(s:string):string // 将输入的十进制串转为二进制串
-SetBin(s:string):void // 用二进制串构造大数
zero, one, two

技术难点 & 算法设计

加密算法

RSA

大数类

- （参考项目 https://github.com/thuliangjz/rsa-DIY）

技术难点

1. 如何存储长整数
2. 如何加速大整数乘方运算

算法设计

1. 2^64 进制存储数据，减少位数
2. 加减乘法汇编加速，一次可以处理多位数字与进位溢出问题
   - 理论上处理器支持 128 位寄存器，可以进一步扩大进制
   - 加法每次即每次从寄存器 `EFLAGS` 中读取上一次相加所得进位，用 `adc` 指令将数和上一次进位 `CF` 相加，保存进位增加索引进入下一轮相加，核心代码如下：

```
sahf;
movq (%%r8, %%rcx, 8), %%rax;
movq (%%r9, %%rcx, 8), %%rbx;
adc %%rax, %%rbx;
lahf;
movq %%rbx, (%%r10, %%rcx, 8);
```

- 减法即将 `adc` 改为 `sub` 即可
- 乘法模拟竖式，先将一个乘数中的某一个整型乘以另一个大整数，移位后加到最终结果上。第一步代码如下：

```
mov %%r8, %%rax;     //小乘数保存在r8中
mov (%%rsi, %%rcx, 8), %%rbx;    //取出大乘数
mul %%rbx;
mov %%rax, %%rbx;
ASM_RESTORE
adc %%r14, %%rbx;    //本轮结果即本次乘法+上次乘法溢出+上次加法溢出
ASM_LOAD
mov %%rdx, %%r14;    //mul指令溢出保存在r14中
mov %%rbx, (%%rdi, %%rcx, 8);
```

- 除法为长除法。这里实现的实际上是基于二进制的整形除法，每次确定商的一个比特位。长除法中的乘法全部被移位所替代，因为每次除数需要乘的都是 2 的幂次，这在一定程度上加快了除法的速度

3. 快速幂借助 Newton-Raphson 算法，用乘法和加减法代替除法。用定点数存储，以调用之前的整数运算实现小数运算
   - 精度问题在原项目中有详细分析，摘抄如下
     - 关于定点数的精度，对于计算 $a^b \pmod n$，在使用快速幂算法进行迭代计算的过程中，每一步实际上都是对于某个$x \le n$计算$x^2 \pmod n$，所以实际上被除数不大于n^2。如果n有p位，则n^2不超过$2p$位，从而对于这样的x^2，逆的精度只要精确到小数点后$2p$位即误差不超过2^{-2p}即可保证与被除数相乘与商的误差不超过 1。
     - 注意到如果将$\lfloor \frac{1}{n} \times x \rfloor$作为商，即使误差本身不超过 1，向下取整之后可能会达到 1（向下取整在这里实现为移位运算），从而按上述方法计算的结果可能需要减去 n 才能得到真正比 n 小的余数。

验证质数

技术难点

1. 如何快速检测大数是否为质数

算法设计

1. 采用方案：Miller-Rabin 检测（伪代码）

   Miller-Rabin 素性检验是一个基于概率的检验方案，与费马素性检验相比条件更强，更不易出现类似 Carmichael 数一类的漏网之鱼；而 AKS 素性检验可以确定地判断给定数字是否为素数，但是速度相较 Miller-Rabin 更慢。

   该素性检验的主要思想为：

   待检验的数为 n （n为奇数）， 将 $n-1$ 分解为 $2^r \cdot d$, 其中 d 为奇数。若 $\exists\ a \in [2, n-2],\ a \in \mathrm{N}\ s.t. \forall 0 \le s \le r-1,\ a^d \not\equiv 1 (mod\ n)$ 且 $a^{2^s d} \not\equiv n-1 (mod\ n)$,则$n$为合数。通过$k$轮检验($i.e.\ \ k$ 次随机选取a)。以下为该检验方法的伪代码：

   write $n-1$ as $2^r \cdot d$ by factoring powers of 2 from $n-1$

   Loop: repeat k times:

     pick a random integer $a \in [2, n-2]$

     $x \leftarrow\ a^d\ mod\ n$

     if $x = 1$ then

       continue Loop

     repeat r times:

       if $x = n-1$ then

         continue Loop

       $x \leftarrow\ x^2\ mod\ n$

     return composite

   return probably prime

参考密码学资料以及 Java 的 RSA 库，为了平衡效率和正确率，1024 位 RSA 将选取$k = 4$.

RSA 实现

- RSA 生成公私钥的步骤如下
  1. 任选两个 512 位素数p, q, 计算其积为N
  2. 计算欧拉数$\varphi = (p-1) \cdot (q-1)$
  3. 任选 e 满足 $1 < e < \varphi$ 且 $gcd(e, \varphi) = 1$, (e, N)即为公钥。实际应用中 e 大多选择 3 或 65537

4. 任选 d 满足 $1 < d < \varphi$ 且 $e \cdot d \equiv 1(mod\ \varphi)$, (d, N)即为私钥。

在步骤 a. 中，为了加快素数生成速度，先随机生成一个奇数 n, 取以 n 为首项的 1000 个连续奇数，用小于 10000 的素数筛一遍，剩下的数用 Miller-Rabin 素性检验进行检测，如果不存在素数，则重复上述过程。

在步骤 d. 中，采用扩展欧几里得算法计算 $d\ mod\ N$ 的逆元。

- RSA 加解密
  - 公钥加密，私钥解密

    $密文 = 明文^e\ mod\ N$

    $明文 = 密文^d\ mod\ N$
  - 私钥加密，公钥解密（用于交易中的数字签名及验证，详细介绍见后文）

    $密文 = 明文^d\ mod\ N$

    $明文 = 密文^e\ mod\ N$

p2p 网络

技术难点

1. 如何单机模拟网络操作

算法设计

1. 手动添加节点并模拟复制区块链等过程（通过多线程加速多个节点生成速度）
2. 通过复制的方式模拟广播过程
3. 多线程模拟多个矿工挖矿过程
4. 设计输出日志以记录各种信息

交互

技术难点

1. 如何实现与用户的高效交互

算法设计

1. 模仿gdb设计命令行交互
2. 提供help指令降低用户学习成本
3. 通过最短编辑距离实现错误输入智能纠正
4. 逐个字符输出，借助cool-retro-term,仿佛重回IBM 3270, Apple 2的年代（