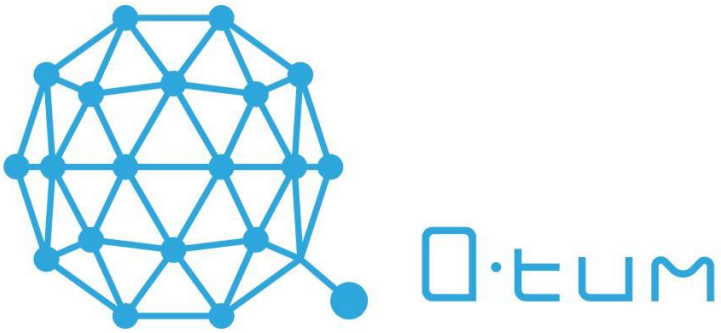
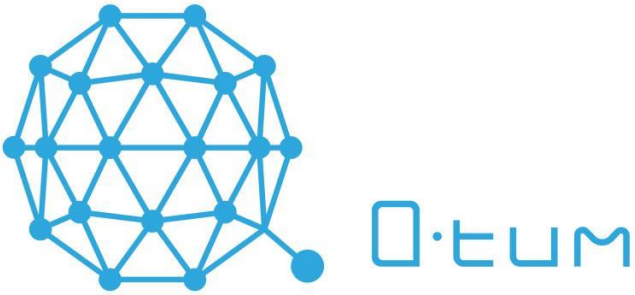
Qtum 部分设计和实现过程

（完整设计和实现文档预计在 6 月份发布）2017/5/29



说明：本文档只是部分 JIRA task 的汇总和摘要，只包含了 Qtum 的一部分设计过程，并且随着 Qtum 项目开发的快速进展，有一些内容会和最终的测试网络不一致，本文档的目的只在于帮助大家了解 Qtum 的开发过程，从中可以了解开发团队在过去一年中做的部分工作。本文档只包含了 Qtum Core 的一些简单实现工作，并不包含 Qtum 设计过程，钱包、浏览器、移动端设计、测试、研究等工作，以上提到的其他工作，Qtum 开发团队也基本上已经完成。测试网络预计 6 月份发布。 （备注：目前 Qtum 团队合计 25 人，核心开发者 20 人，如果有兴趣加入 Qtum，请发简历到 job@qtum.org ）

Qtum 整个项目的实现过程大概花费了 1.5 年的时间，原型想法开始于 2016 年 3 月份，在 2016 年 3 月，领导技术团队搭建基于以太坊的企业级应用的时候，发现了很多局限性，并且结合自己读博士期间的研究和过去 5 年在区块链行业经历的一切和思考，以及对区块链行业未来的判断，（优酷视频：2 个小时解读 量子链的过去 现在 和未来），当时考虑的是如何为现有的生态增加价值，并找到 Qtum 的独特定位，能够在中国搭建一个优质的区块链技术社区。



比特币带来的技术革命的开端，以太坊把智能合约从概念变成了现实，如何站在现有的生态上走出一条激励相容的道路是一个挑战，答案也只有一个：那就是创新。

Qtum 在比特币 UTXO 模型的基础上，做了账户的抽象，从而可以支持多种虚拟机（EVM 是其中一个，Lua 和 Bloq-ora 也在实现过程中），并且解耦了普通交易和合约交易，底层的共识机制采用拓展性更好的 POS 机制，并且创造了分布式自治协议（DGP）， 使基于机器节点共识的决策共识可以民主化和分布式。

我们相信 Qtum 的实现为现有的区块链生态带来了价值，Qtum AAL 打通了 UTXO 模型和虚拟机之前的鸿沟，Qtum DGP 使区块链的分布式决策和升级成为可能，即将公布的 Qtum MPOS 极大的提高了传统 POS 机制的安全性。更多虚拟机的支持和闪电网络以及侧链的开发，以及后期无限扩展的共识机制等也在我们的开发日程中，Qtum 会有快速的技术迭代。

另外随着 Qtum 测试网络的发布，我们相信我们也培养了中国区块链领域最好的一支技术团队，他们之前大多就职于百度 腾讯 阿里，毕业于北大 北邮 中科院，现在他们精通比特币和精通以太坊，并且全程参与了 Qtum 的设计和开发，这也是 Qtum 项目的另外一个目的吧，希望在中国，有更多的人参与到区块链的技术社区，推动技术的发展。

如果说我们这一代人还有什么优势的话，那就是这一代年轻人洞悉中国移动互联网带来的翻天覆地的变化，我们知道如何把区块链的易用性做到最好，这一切也将助力 Qtum 在中国和世界的发展。

未来已来，只是还没有开始流行。

QTUMCORE-1

Qtum基本品牌策略

进行基础性的比特币代码重命名工作，将其纳入Qtum. 不必太泛，但需要包含以下内容：

1. 使用~/ .qtum而不是~/ .bitcoin作为默认的数据目录
2. 构建系统输出qtumd而非bitcoind / cli等

QTUMCORE-2

创造Qtum基础网络

更改参数，区分Qtum网络与比特币网络

1. 更改创世种子
2. 更改P2P / RPC端口
3. 更改P2P网络字节
4. 添加必要的任何修改，挖矿并同步新的主网络
5. 删除比特币检查点和节点列表
6. 确保主网络、测试网络和regtest网络正常运行， 主网络和测试网络与比特币网络明确区分且不兼容

移除只影响比特币的不相干测试，修改网络参数相关的测试。

具体参数：

块时间: 128 秒
块大小: 2Mb
P2P 端口: 3888 (随机选择)
RPC 端口: 3889
测试网络 P2P 端口: 13888
测试网络 RPC 端口: 13889
Qtum pubkeyhash 地址: 58 (Q)
Qtum p2sh（Pay-to-Script-Hash） 地址: 50 (M)
Qtum 测试网络 p2pkh（Pay-to-Public-Key-Hash）: 120 (q)
Qtum 测试网络 p2sh: 110 (m)

QTUMCORE-3

为合约执行添加EVM和OP_CREATE

集成EVM并确保基本合约可以执行

添加新的opccode OP_CREATE (之前是OP_EXEC), 包含以下四个参数，顺序如下：

1. VM版本 (目前1代表EVM)
2. Gas 费用(尚未使用，任意值)
3. Gas限制(尚未使用, 假定为很大的值)
4. 字节码

现在，OP_CREATE 在区块链上的交易的脚本格式是固定且强制的（即只允许“标准格式”在区块链上运行）。

当遇到OP_CREATE时, 执行 EVM 并同步交易数据到数据库（triedb）。

注意:

确保遵循外部代码策略(优先提交vanilla未修改的代码，并根据需求进行修改)。

确保EVM测试通过 (也可以安装持续集成来完成)。

新合约的QTUM地址
对QTUM地址是否可以在合约创建之后生成进行测试. 进行测试前需确认以下事项：

- в –regtest 节点启动
- 通过比特币测试

测试步骤:

1. 在bitcoinlib-js内创建TX_DEPLOYMENT交易, 运行decoderawtransaction代码
2. 在bitcoinlib-js内创建 TX_ASSIGN_SC交易, 运行decoderawtransaction代码.
3. 在bitcoinlib-js内创建TX_DEPLOYMENT交易,运行sendrawtransaction代码
3. 在bitcoinlib-js内创建TX_ASSIGN_SC交易, 运行 sendrawtransaction代码.

预期结果:

1. 交易被标记为已部署.
2. 交易被标记为已委派.
3. 合约创建成功(在控制台上显示地址及余额).
4. 合约状态改变(在控制台上显示地址及余额).

QTUMCORE-4

添加 PoS 区块验证和挖矿

完成这个任务后， 量子链应能够验证基本的PoS v3 区块,钱包应能够挖掘些类似于PoW的区块。目前并没有只接受PoS区块的网络规则，因此量子链将会接收PoW区块和PoS区块。两个区块也将使用和比特币相同的默认标准难度。

需确认事项:

权益调节器可以修改每个区块
币龄不会改变原有PoS值
无去中心化的检查点 (设计中)

确保以下陈述正确:

1.不强制把tx费用作为共识规则
2.静态PoS奖励 (先设置为1个token, 后续再做更改)
3.权益调节器v2 –其目的在于防止 txout (coin) 持有者更改txout在交易确认时产生的权益证明.区块验证改变后,现在你只需要之前的区块，如此一来，其他的所有区块都可以被清空，空间就节省了下来（你需要最后的500个区块赋予权益），还为未来大的数据释放省下了容量。
4.与BIP66的兼容性
5.只有最后500个区块需要通过权益证明
6.没有币龄,只是一个币确认请求(500个区块刚好足够)

QTUMCORE-7

添加"createcontract" RPC 调用

添加新的RPC调用 "createcontract"。这个RPC调用是用来部署新的智能合约到量子链。

调用方式:
createcontract gas-price gas-limit bytecode [sender-address] [txfee] [broadcast]

参数sender-address 为可选项. 如果地址没有指定，将从钱包中随机挑选一个地址 。如果sender-address地址没有可花费的输出, 将显示错误，交易也无法创建。

txfee也非必选项，如果没有指定txfee，应该使用自动交易费(比如： sendtoaddress使用自动交易费)

broadcase应该设置为true。如果设置为false，交易创建并签名后在以十六进制打印到显示器上而非广播到网络中。

如果 sender-address 有一个交易输出, 但不足够支付gas费用和tx费用, 那么该钱包拥有的任何UTXO将被用于支付这些费用. (不是所有的资金都要来源于sender-address，但sender-address必须为vin[0])

执行程序后, 如果广播正确, 将打印 txid和新的合约地址。

QTUMCORE-10

为合约添加调用其他已部署合约的能力。

合约应能够使用新的opcode OP_CALL调用其他合约。

参数顺序如下:
version (32 bit integer)
gas price (64 bit integer)
gas limit (64 bit integer)
contract address (160 bits)
data (任意长度)

OP_CALL 现在应该总是返回false。

OP_CALL 只在输出时执行合约； 与OP_CREATE相似,使用特定规则在输出时运行脚本(而非比特币中消费时才运行脚本一样。). 合约执行只在交易脚本是标准格式且没有其他opcodes时才会运行。 如果OP_CALL使用了非法的合约地址，任何合约执行都不会发生。区块链中的交易仍然有效。如果OP_CALL发送的是资金，那么这笔资金（减去Gas费用）将会产生一个退款交易来退回到vin[0]的vout脚本。
暴露给EVM的“sender”应该是被vin[0]花费的pubkeyhash。如果vin[0]花费的vout[0]不是第一个pubkeyhash，那么“sender”就是0。
使用OP_CALL的vout可以把资金发送给合约。这个资金将会被AAL处理，并且暴露给EVM。
在一个交易中可以多次调用OP_CALL，然而在合约执行之前，需要检查每个OP_CALL的Gas费用和Gas限制，确保能够提供足够的交易费来支付gas费用。
此外, 当合约被执行时还要确认以下事项，如进入内存池的时间等。

QTUMCORE-13

添加"sendtocontract" RPC调用

添加一个RPC方法，能够发送数据和资金给已经部署在区块链上合约。

格式如下：

sendtocontract contract-address data (value gaslimit gasprice sender broadcast)

该操作将使用OP_CALL来创建一个合约交易。

参数sender-address 为可选项. 如果地址没有指定，将从钱包中随机挑选一个地址 。如果sender-address地址没有可花费的输出, 将显示错误，交易也无法创建。

broadcase应该设置为true。如果设置为false，交易创建并签名后在以十六进制打印到显示器上而非广播到网络中。

如果 sender-address 有一个交易输出, 但不足够支付gas费用，tx费用和value, 那么该钱包拥有的任何UTXO 将被用于支付剩余费用. (不是所有的资金都要来源于sender-address，但sender-address必须为vin[0])

执行结束后, 如果broadcast是true, 将打印txid和新的交易地址。

QTUMCORE-14

为链下计算添加 "callcontract" RPC调用

应该有一个RPC方法用来执行链下合约，即不需要和区块链网络进行交互，也不需要Gas费用和其他费用。

callcontract contract-address data (sender)

合约可以在本地执行， 但如果合约函数返回了数据，那么返回值将由RPC返回或者打印 . 参数 sender 是可选项，且不要求拥有vout (任何有效地址皆可)

QTUMCORE-16

改变主网络使其只允许在特定的区块高度之后进行PoS

QTUMCORE-18

将 EVM "LOG" opcode的输出值输入到一个可访问的日志文件中

日志文件应该接收来自EVM LOG opcodes 的所有数据，以便用于测试和debug。下面是为此而写的ITDs

该日志文件的功能通过添加选项`-record-log-opcodes`给qtumd来使用。

ITD: https://github.com/qtumproject/qtum itds/blob/master/evm/log%20opcode%20logging.md

QTUMCORE-22

安装持续集成软件

安装Jenkins服务器完成以下事项:

1. 自动编译qtum代码库的每个分支，并报告编译错误。
2. 自动运行 Qtum (Bitcoin) 的测试代码，并报错失败的测试。
3. 销毁已经构建的虚拟机，以确保我们不会使用更多的虚拟机时间。

QTUMCORE-23

更改所有输出字符串，Bitcoin -> Qtum, BTC -> QTUM

用 qtum 和 QTUM替换所有bitcoin和BTC (只涉及输出字符链，而不是实际的类或变量名也不是代码注释)

QTUMCORE-26

将脚本限制增加到合适的值

在比特币中存在着许多使用智能合约的限制，我们将解除这些限制。

- 最大化数据发送大小: 1Mb
- 最大化数据栈大小: 1Mb
- 其他执行合约脚本的限制

QTUMCORE-27

PoS的共识机制/链参数

在网络中，共识机制提供影响新区块创造的参数。

添加PoS限制用以确定PoS初始难度

对减半间隔现在为四年，一共有7个减半间隔。

主网的最后一个PoW区块需要被定义，难度设置为5000，PoW奖励设置为20000个币，PoS奖励为4个币

QTUMCORE-28

更新PoS区块和区块头项参数

区块头项参数的基本原则是保证其尽量小。

PoS正常运行需要四项参数:

- 区块签名，即整个区块的签名，由该区块创造者来完成。
- 区块类型 (PoW或POS)
- 前驱stake位置，用来确认区块的有效性, 是PoS的必要参数
- staking时间, 即staking交易的创建时间, 是PoS的必要参数

区块签名必须为一个参数.该参数还需出现在区块头中，因为隔离见证，所以这个参数将被发送到区块头和交易中。

其他从PoS交易中提取出来的参数，不管是否包含以上3个在头项中的参数都可被讨论. 因为区块头总是先于区块被下载下来，如果不包含上述参数将不能在下载整个区块前进行PoS检查。黑币(Blackcoin)是在检查之前将整块下载下来并将上述参数填入区块中。

QTUMCORE-29

PoW/PoS内核更新

PoW和PoS内核源代码保存在以下文件中：

- pos.h
- pos.cpp
- pow.h
- pow.cpp

这些文件的位置由master-pos决定, 在黑币中这些文件位于 kernel.h 及 kernel.cpp.

我们需要确定PoW和PoS内核代码是否保持原有代码结构还是使用不同组织结构。

权益调节器（stake modifier）的版本将在第二版中使用。

区块难度、masks、modifiers, time-stamps等都在此使用，因此对 PoS 来说都是很重要的部分.

QTUMCORE-30

Quantum的矿工

矿工指自动推送区块到网络的工具。
我们需要用他们来避免使用原生代码推送新区块和建立新网络
因为使用了比特币的矿工，PoW挖矿有很多方式。

- 内部矿工
- CPU 矿工
- 显卡矿工

PoW挖矿仅用于最初的区块来确立权益持有者。因此，矿工对PoW来说永远是充足的。

在对最初的 PoS区块进行挖矿后，才开始需要 PoS 矿工.
PoS矿工同样也在黑币中使用.

Quantum程序启动时需要参数来激活与解除激活矿工

参数 "gen" 用来启动 PoW "内部矿工"，参数"staking"激活staking矿工。激活Quantum矿工又与此不同。

QTUMCORE-31

更新量子币的最大数量

文件amount.h包含最大数量币的代码:

static const CAmount MAX_MONEY = 21000000 * COIN;

PoS 本身没有对减半间隔，所以总值将会无穷增长。

MAX_MONEY 应该更新为最大值。

QTUMCORE-32

在交易中增加对coinbase和coinstake的检查

检查交易类型。

Coinbase检查是否有同样的输入币在之前的输出中已经产生过. 这些币将成为挖矿（ PoW区块）的回报 。

Coinstake检查是否输入币在之前的输出中已经产生过。第一次交易输出是空值，第二次、第三次则是回报。第一次交易输出为空是为避免将其误当做普通交易。假定输入较小，那么输出币将被存储在第二次输出中, 此种情况外，输入与输出币都将被分散到第二次与第三次输出之中 。

我们可以添加额外的检查对普通交易进行定义，可称为IsNormal (或者叫其他更合适的名称)，是用来定义普通交易，而不像PoW和PoS这种生成的交易。

同时还要保证没有OP_CREATE和OP_CALL交易被使用。

QTUMCORE-33

为PoS交易添加时间戳

时间戳指交易创造的时间点，应用于PoS中以将交易时间大于区块时间的交易排除掉。.
时间较长的交易将会在时间值少于区块时间值时并入链中

另外，还可以利用该参数决定区块权益时间(PoS 交易时间).

QTUMCORE-34

PoS区块签名

新 PoS区块的创建者需要对区块进行签名，以便其他矿工确认该区块是否创建自矿工以及检查它的有效性。

根据PoS v3, 第一笔交易为空，第二笔交易是在区块内的PoS交易

区块签名在PoW 中为空- 但第一笔交易不能为空
.

所有PoS区块版本应升级到2

QTUMCORE-35

创建新的PoS区块

PoS区块第一笔交易为空.
PoS 区块第二笔交易包含 PoS交易

假设产生币的交易输出值是完成的，则币就是可用的

创建权益所需的币列表,可以使用UTXO模型创建权益交易

权益币用于创造PoS交易. 这是一种输出大于输入的特殊交易类型。意味着假设输入币为1000, 输出币加上回报将达到1001币. 普通交易的输出值一般小于或等于输入值，但PoS交易并不符合这个规则。

PoS里一般认为有持有超过200个币的人即视为大宗权益持有者, 那么输出将包含两种，例如：输入为1000, 输出为500.5 和 500.5. 这个参数可在后续中进行修正（如有必要）

创造PoS 交易后, 区块将包含签名后的交易。

QTUMCORE-36

接受PoS区块

接收区块头之后要进行检查。

接受整个区块之前，需要对其检查。(当这个区块是由旷工创建或者来自其他链).

当运行PoS 区块时我们要处理重复的哈希及孤立区块。

QTUMCORE-37

PoS的币序列化和交易撤销

交易被接受之后，币就被存储起来(区块被推送到主链) ，防止发生二次消费。

为了给正在交易的币提供标记，存储币的公式需根据币权益交易进行更新。直到他们确认完成之后方可进行消费。

如果其他人创造了一个含有多工作量的区块，那么链的顶端将会随之改变。这种情况下，就需要取消之前区块的交易。在此过程中还需要更新包括币权益在内的交易公式。

QTUMCORE-38

PoS信息 RPC（远程方法调用）更新

需要提供PoS相关参数信息，包括：难度、区块内挖矿类型（PoW/PoS）以及PoS交易的回报。

QTUMCORE-39

在钱包里向用户显示权益信息

更新钱包以获得权益所需信息并在GUI标签中显示

更新钱包正确显示可用于消费的币而不是权益运行中分配的币。

QTUMCORE-40

区块验证更新

钱包/后台进程开启后，将通过四个级别的验证程序来验证区块是否有错误

默认级别是3, 但代码如需被认定为有效需要通过全部四层

通过创建区块索引从区块数据中心读取PoS参数时，需要更新区块验证

QTUMCORE-41

测试网络启动之前的TODO列表

下列是测试网络开启前需要更改的事项。
这些值将被保持在合适的数值上:

1- 恢复区块成熟值到500
2- 在链开启后设定BIPs的 nStartTime 和 nTimeout 到实际值, 或者按默认启动

3-测试网络发布前还原 DEFAULT_MAX_TIP_AGE 为 24 * 60 * 60

QTUMCORE-42

为区块每个地址和时间点添加索引

为区块每个地址和时间点添加索引是非常有用的. 基于：
https://github.com/bitpay/bitcoin

将来可以用作深度浏览节点部署

这个任务可以做到相对独立，因为它不会影响到任何其他正在打开状态的任务。

QTUMCORE-43

添加储备余额

储备余额是指被保留并不用于产生收益的币。由参数nReserveBalance定义储备余额。可以在Blackcoin搜索到ReserveBalance.

以下功能包含储备余额并需要更新:

- -可以在应用程序初始化或者在配置文件时设置reservebalance参数
- 余额储备的 RPC调用
- 通过储备余额减少权益和权益可用币
- 用标签将储备余额显示在optionsdialog.ui上并更新模型（Qt钱包）

- QTUMCORE-44

在CBlockHeader 中添加hashStateRoot

CblockHeader中hashStateRoot是用来处理以下操作:

§ 回滚数据库状态(DisconnectBlock, incorrect block等)
§ 利用一些数据初始化运行（目前为空）

§ QTUMCORE-45

§

EVM 环境初始化(EnvInfo)

§ EVM 环境需要从Solidity中提取区块信息。以便合约能够利用这些信息。

§

QTUMCORE-46

创建 QtumTxConverter and ByteCodeExec测试

创建QtumTxConverter and ByteCodeExec的单元测试

QTUMCORE-47

为合约添加AAL账户抽象层

账户抽象层是可以让EVM合约可以运行在UTXO模型上的一种途径。该任务需要覆盖近乎整个模块。

通过OP_CALL.资金被发送给合约。当合约收到或发送资金时，将产生“压缩交易”. 该交易将会

消费任何需要改变余额的合约，输出值将会成为合约新的余额。

压缩交易的创建使合约不会拥有超过1个UTXO.这显著简化了coin 收取, 也防止了许多攻击向量填满区块。

每个区块可以拥有一个以上的压缩交易。假定一个区块中单一合约产生多个余额，压缩交易将会花费此前同个区块中压缩交易的余额。这有点浪费，但是将极大简化了逻辑，并允许在不许重写任何先前交易情况下轻松添加更多合约执行交易

完整行为案例 的 ITD 如下: https://github.com/qtumproject/qtum-itds/blob/master/aal/condensing-transaction.md

（压缩交易行为

在初始模型中，我们有个未解决的DoS问题，就是 攻击者可以向合约发送许多小的输出值，然后一次花费这些值，从而超出区块大小的限制。

解决这个问题的办法就是我们提到的“压缩交易”。基本上说，当合约执行完成后，只有一个UTXO 拥有所有的tokens。

环境

需要包含安装过程, 也可在主网络和注册网络中进行

测试步骤

1. 部 署 发送方合约
2. 每个合约使用 setSenders来设置，使他们之间相互识别

3. 挖掘一个区块 (区块 B1)
4. 发送8 个币 到 Sender1, method share(Transaction T1)
5. 挖掘一个区块(区块B2)
6. 发送 2 个币到Sender1, method keep (Transaction T2)

7. 发送 2个 币 到 Sender1, method sendAll (Transaction T3)
8. 挖掘一个区块 (区块 B3)
9. 发送2个 币到Sender2, method share. 发送低值 gas limit造成其用尽(Transaction T4)
10. 调用 Sender2, method withdrawAll (Transaction T5, sender address is A1)
11. 挖掘一个区块 (区块 B4)

预计结果

步骤3中, 3个合约的余额都为0。 而且除创建的合约外没有 UTXOs 与之相关联

区块B2

步骤5中, 区块B2 应包含 4 个交易

1. Coinbase
2. Stake
3. T1
4. Condensing TX (Transaction C1)

T1 是未经修改.

C1 应包含1个输入:

- Spend contract send (OP_CALL) in T1 using OP_TXHASH

以及 3 个输出:

- C1.V0: version 0 OP_CALL to Sender1 of value 5 (8 initially, -4 for send to Sender2, then +1 for callback from Sender3)
- C1.V1: version 0 OP_CALL to Sender2 of value 2.5 (4 from Sender1, -2 for send to Sender3, then +0.5 for callback from Sender3)
- C1.V2: version 0 OP_CALL to Sender3 of value 0.5 (2 from Sender2, -1 for send to Sender1, -0.5 for send to Sender2)

如果在区块之后检查这些合约的余额，应该是与C1的输出值完全匹配的。而且，合约执行并完成 condensing tx后是绝对不能拥有超过1个UTXO 的。

这甚至可能成为抗击潜在攻击向量的网络规则。

现在我们将数据发送给 T2 ，T3 并挖掘区块 C.

区块 B3

区块 B3 需包含以下六项交易:

1. Coinbase
2. Stake
3. T2
4. Condensing TX C2
5. T3
6. Condensing TX C3

C2输入:

- T2 contract spend vout spent with OP_TXHASH
- C1.V0 --Sender1's balance

C2 输出:

- C2.V0: version 0 OP_CALL to Sender1 of value 7 (5 from previous balance, +2 from T2)

C3 输入:

- T3 contract spend vout spent with OP_TXHASH
- C2.V0 -- Sender1's balance
- C1.V1 -- Sender2's balance

C3 输出:

- C3.V0: version 0 OP_CALL to Sender2 of value 11.5 (2.5 from previous balance, +9 from Sender1's balance plus T3's contract coins)

需注意的是，在这个点上Sender1 不拥有自身的 UTXOs,因为其余额为0.

区块 B4

现在区块 B4 已经被挖掘, 它包含以下交易:

- Coinbase
- Stake
- T4
- Refund Transaction R1
- T5
- Condensing Transaction C4

R1 输入:

- T4 contract spend vout spent with OP_TXHASH

R1: 输出:

- R1.V0: pubkeyhash script (back to the first vin of T4) of value 2

C4 输入:

- C3.V0 -- Sender2's balance
- C1.V2 -- Sender3's balance (no coins sent with T5, so no input for it. We ignore 0-value outputs)

C4 输出:

- C4.V0: pubkeyhash script for address A1, of value 12 (11.5 from Sender2 balance, +0.5 from withdrawal of Sender3's balance)

测试最后阶段，最终结果是这3 个合约都没有余额和UTXOs.

合约代码

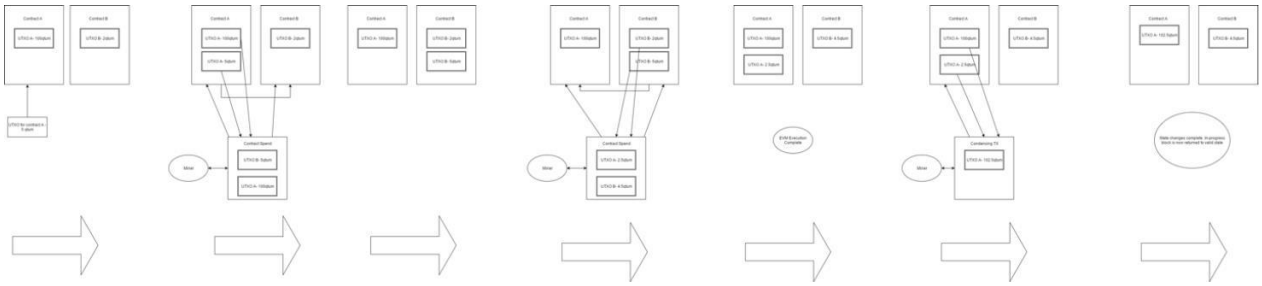
```
pragma solidity ^0.4.0;
contract Sender1 {
    Sender2 sender2;
    Sender3 sender3;
    function Sender1() {
    }
    function setSenders(address senderx, address sendery) public{
```

```
        sender2=Sender2(senderx);
        sender3=Sender3(sendery);
    }
    function share() public payable{
        if(msg.sender != address(sender3)){
            sender2.share.value(msg.value/2);
        }
    }
    function sendAll() public payable{
        sender2.keep.value(msg.value + this.balance);
    }
    function keep() public payable{
    }
    function() payable { } //always payable
}
contract Sender2{
    Sender1 sender1;
    Sender3 sender3;
    function Sender2() {
    }
    function setSenders(address senderx, address sendery) public{
        sender1=Sender1(senderx);
        sender3=Sender3(sendery);
    }
    function share() public payable{
        sender3.share.value(msg.value/2);
    }
    function keep() public payable{
    }
    function withdrawAll() public{
        sender3.withdraw();
        msg.sender.send(this.balance);
    }
    function() payable { } //always payable
}

contract Sender3 {
    Sender1 sender1;
    Sender2 sender2;
    function Sender3() {
    }
    function setSenders(address senderx, address sendery) public{
        sender1=Sender1(senderx);
        sender2=Sender2(sendery);
```

```
    }
    function share() public payable{
        sender1.share.value(msg.value/2);
        sender2.keep.value(msg.value/4);
    }
    function withdraw() public{
        msg.sender.send(this.balance);
    }
    function() payable { } //always payable
}
```

)



QTUMCORE-48

PoC后的改进

在初始PoC 重新实现后开放讨论：

1.移除tx中的时间戳
2. 对合约地址进行Base58编码
3. 保持 PoS 余额不可消费
4. 讨论区块成熟度
5. 合约管理接口

QTUMCORE-49

PoC后研究Research 攻击向量

攻击向量:

1. 添加最小gas fee: 我们需要添加最小gas fee避免部署 0代码合约和不支付gas情况

或许是这样,但我想合约费用应该足够用以处理这些。
2. 需要明确如何处理wei vs satoshi

这步比较困难.

3. 可能存在利用EXEC操作来进行垃圾信息攻击，因为EXEC操作并不被认定为dust (使用不支付gas VM 版本0)

我认为txfee应该考虑到这点, 因为版本0是一个潜在的非标准交易(仍需要考量)

4. 重复块垃圾信息 – 区块内可能存在内核哈希值相同内容不同的垃圾信息. 最糟糕的情况如果包含许多高价交易待验证，那最后一笔交易将会无效. 这应该会触发DoS条件禁止节点,但价格多高才会触发节点来处理还有待评估。

5.全面考虑segwit与PoSv3合并的意义，以及SPV轻型钱包的安全性。

§ QTUMCORE-51

§

为OP_CREATE 和 OP_CALL规定version字段

为保证协议的可扩展性，我们需要设置一些规则用以后续改变OP_CREATE和OP_CALL版本参数从而升级或添加新的虚拟机

我们需要一个确定的虚拟机版本格式而不是目前的"升级时增加版本号"。 这能使我们更轻松地规划升级和软分叉。

建议字段：

1. 虚拟机格式(可从0拓展): 2位
2. Root 虚拟机 – 实际使用的虚拟机, 例如 EVM, Lua, JVM等: 6位
3. 虚拟机版本 –Root 虚拟机所使用的版本(为了向后兼容性升级root虚拟机) – 8位
4. 标记选项 – 虚拟机执行和AAL的标记: 16位

总共: 32 bits (4字节). 因为这个数据将存在于每个 EXEC 交易之中，所以大小很关键。

标记选项用来控制合约的创建(仅适用于 OP_CREATE):

- 0 (保留) 固定gas计划 – 如果为true, 该合约不允许选择不同的gas计划。结合gas计划使用OP_CALL而非合约创建过程中形成的进程表将导致异常和ga耗尽而退款情况。

- 1 (保留) 启用合约管理接口 (仅限备用,稍后实现)。允许合约自行控制他们允许的虚拟机版本，以及允许在合约生命周期内改变值。
- 2 (保留) 拒绝版本0资金 – 如果为true, 该合约将不能接收除AAL所需以外的版本0 OP_CALL的钱
- 字节3-15 可用于将来的扩展

控制合约请求标记选项: (仅适用于OP_CALL)

- (暂无)

控制合约请求和创建的标记选项:

- (暂无)

这些标记将在后续的任务中实现

需注意，现在版本字段必须为4字节推送。

一个标准的EVM现在使用版本号 (十六进制) "01 00 00 00"

共识行为:

虚拟机格式必须为0

Root 虚拟机可以为任何值. 1 代表 EVM, 0 代表no-exec. 所有其他值都将造成no-exec
(允许, 但不得执行, 便于后续软分叉)
VM版本可以为任何值 (软分叉兼容性). 如果新版本大于0 (0为初始版本), 那将执行版本0并忽略该值。
标记选项可以为任何值(软分叉兼容性)。（ 忽略非活动标记字段 ）

标准内存池行为:

VM Format 为 0
Root VM 为 0 或者 1
VM Version 为 0
标记选项 – 可设置所有有效字段，未分配字段设为0。

EVM默认设置:

VM Format: 0
Root VM: 1
VM Version: 0
Flags: 0

QTUMCORE-52

合约管理接口

(注意, 虽然这是个很好的功能，但并不是主网络的目标之一)

需要能在合约内部管理合约的生命周期. 过程中需要对相关的变量和配置值进行更改：

- 允许的 gas 进程表
- 允许的 VM 版本 (如果将来的VM版本违反了这个合约将禁止其使用，同时不允许过去的和不建议使用的VM 版本与现在的合约产生交互。)

- 创造标记 (OP_CREATE 中的版本标记)

所有这些变量都必须能在合约本身内部通过分散代码进行控制。例如，在DAO中，参与者可以在合约内进行表决，而后合约启动代码改变这些参数。此外，一个合约还应能够检测自身整个执行过程即便是在他最初创建的时候。

我建议将整个接口开发成一个预编译的合约。如果一个合约想与其交互，它会像其他合约一样，使用Solidity ABI进行调用。

合约建议使用的ABI

- `bytes[2048] GasSchedule(int n)`
- `int GasScheduleCount()`
- `int AddGasSchedule(bytes[2048]`
- `bytes[32] AllowedVMVersions()`
- `void SetAllowedVMVersions(bytes[32])`

备选实施方案:
为了验证合约允许自身被以特定方式进行调用，我们还需要一个特定Solidity函数

```
pragma solidity ^0.4.0;
contract BlockHashTest {
function BlockHashTest() {
}

function ValidateGasSchedule(bytes32 addr) public returns (bool) {
if(addr=="123454")
{ return true; //allow contract to run }
```

return false; //do not allow contract to run
}
function ValidateVMVersion(byte version) public returns (bool){
if(version >= 2 && version < 10)
{ return true; //allow to run on versions 2-9. Say for example 1 had a vulnerability in it, and 10 broke the contract }

return false;
}
}
```

如此一来，合约就能管理自身的状态。基本工作方式为当你使用OP_CALL调用一个合约时将首先执行这两个函数(执行也将包含在gas花费中). 如果任何一个函数返回false，将马上触发gas耗尽的异常并且取消执行。

管理"ValidateVMVersion"回调有点复杂，因为我们要决定使用哪个VM版本。挑选错误将导致函数本身不能正常运行。

## QTUMCORE-53

### 为“version 0 sends”给合约添加opt-out标记

有些合约想要排除在以太网中没有而在Qtum中有的一些功能，这样一来更多的以太网合约能够进入 Qtum 还无需担心Qtum区块链的新功能的影响。

应该在版本字段中添加两个标记选项，且仅对OP_CREATE 创建合约有效:

2. (1st bit) Disallow "version 0" OP_CALLs to this contract outside of the AAL. (DisallowVersion0)

如若启用, 使用"root VM 0" (导致不执行)的OP_CALL 将被禁止向合约发送. 如果试图使用"version 0" OP_CALL 发送货币，将造成out of gas exception 资金也会被退回.账户抽象层内部的 Version 0支付不受该标记的影响。

除了这些新的共识规则，还要有一些标准的内存池检查：
```

1. 如果OP_CALL tx使用了不同的 gas 进程表而非交易创建所产生的进程表, 将会设起禁止活动gas标记， 而后该交易江将被mempool视为非标准交易而遭到拒绝。

(版本 0支付在内存池中也不应被视为标准交易)

QTUMCORE-54

给ALL合约添加测试

QTUMCORE-55

分布式自治协议(DGP) 和 动态Gas.计划

DGP 是量子链的一个新概念，它允许我们改变各种网络参数，而不用重新发布钱包。DGP 配置的安全性是由一份实现了多签名检查的智能合约保障。

DGP 的第一个重要概念是允许修改 Gas 计划，而无须下载更新钱包，或者创建网络分叉。
DGP 旨在控制很多事情，但首先只能用于动态 Gas 计划

每个DGP功能包含2个主要合约:

1. DGP框架合约
2. 控制共识数据的DGP功能合约
合约代码地址: https://github.com/qtumproject/qtum-dgp/blob/master/dgp-template.sol.js

DGP框架合约实现了以下几点:

1. 提案和投票：每个参数变更（包括内部密钥管理）需要被提议，然后投票。如果投票满足选定

的条件，提议被接受并且执行操作。因为使用`msg.sender`来计算投票，所以一个公钥哈希地址或者一个合约地址都算可以当作一个参与者。

2. 密钥管理：应该可以添加和删除参与者，以及更改参数，例如接受一个提议或者添加一个密钥需要接受多少密钥。

3. 发送正确格式的数据给DGP功能合约

4. 允许自己被禁止，以便不使用硬分叉来修复。

5. 一次只能允许一个提议，一个提议只能由一个参与者发起。每个提议应包含不超过5000个区块的限制。每个提议到期后或者投票完成后可以拒绝和同意。

6. 应该有可以删除提议的管理员列表，这些管理员还应该是添加提议的唯一认可组织。

同时，DGP功能合约非常简单，只做2件事：：

1. 只能从合适的 DGP 功能合约接收消息或者数据

2. 使用`SSTORE`作为标准化方式存储共识数据，以便区块链可以对 RLP 中的数据进行检索和解析，而无需执行 EVM。

每个DGP功能数据记录都有区块高度，注意，旧的共识数据记录不可以被替换或者修改。
这是因为我们以后需要引用这些历史数据。
虽然改变历史数据不会影响同步区块链，但历史数据应该被保存，以防万一。

例如，通过动态 Gas 计划数据，来添加一个附加功能，以便一份合约调用可以选择使用创建时就已经激活的 Gas 计划。
为了在没有独立数据库的情况下也可以使用，历史数据必需保存在 RLP 中。这也允许 DGP 修改在出现问题时修改这个功能，而且这些修改也是可以追溯的。

动态Gas计划数据记录包含以下数据：

1. tierStepGas0
2. tierStepGas1
3. tierStepGas2
4. tierStepGas3
5. tierStepGas4
6. tierStepGas5
7. tierStepGas6
8. tierStepGas7
9. expGas
10. expByteGas

11. sha3Gas
12. sha3WordGas
13. sloadGas
14. sstoreSetGas
15. sstoreResetGas
16. sstoreRefundGas
17. jumpdestGas
18. logGas
19. logDataGas
20. logTopicGas
21. createGas
22. callGas
23. callStipend
24. callValueTransferGas
25. callNewAccountGas
26. suicideRefundGas
27. memoryGas
28. quadCoeffDiv
29. createDataGas
30. txGas
31. txCreateGas
32. txDataZeroGas
33. txDataNonZeroGas
34. copyGas
35. extcodesizeGas
36. extcodecopyGas
37. balanceGas
38. suicideGas
39. maxCodeSize (not gas really, but related so it can stay here)

对于每个 DGP 功能，都应该有一个对应的硬编码，以便在没有 DGP 部署和功能的情况下，块链是正常的。 因此，对于动态 Gas 计划，应该有一个默认的硬编码过的 Gas 计划，在 DGP 合约产生一个 Gas 计划前使用。

激活注意事项：

尽管没有按照共识强制执行，但在激活前，少于 500 个块的无 DGP 参数的修改不应该被发布。这样做可能导致不必要的网络分叉和孤立块。DGP 参与者有责任确保没有提议被通过，因为这些提议可能在 500 个块之后活跃。

动态 Gas 计划一些限制

在 EVM 中，我们应该给每个 Gas 计划项确定合理的最小值。一些操作允许免费，但另一些操作码，比如签名和散列不允许免费。

DGP 功能点

1. getGasSchedule
2. [reserved for getReward]
3. [reserved for getMinGasPrice]

未来可扩展性

可能出现的一个重大需求是使用更复杂的授权模型，例如社区投票，两个独立的多重签名配置或其他的在 DGP 框架合约中实现的其它模型。
可以通过添加一个新的“参与者密钥”来完全改变授权模型，然后删除所有其它密钥，并把所需的最大签名减小到 1。
在这种情况下，这个“参与者密钥”将是一个实现了所有授权逻辑的智能合约。
在这种情况下，授权合约也需要处理提议/投票系统，然后在一次执行中提议和同意对实际 DGP 框架合约修改。

DGP参数修改步骤

1. 提议经过社区讨论后产生。若非紧急问题，启动日期应不早于 20000 个区块。如果是紧急问题，不应早于 1000 个区块（500 个区块用于投票，500 个区块用于激活）。

2. 提议是被发送到 DGP 框架合约

3. 对提议进行投票

4. 如果投票通过，DGP 框架发送 DGP 功能合约给提议数据。

5. 新的 DGP 参数更改在提议激活块内有效

实际上必须实现什么？

为了支持多个 DGP 合约，所以我们应该实现一个支持它们的框架。
每个 DGP 合约会被部署到区块链，合约地址将被硬编码到区块链中。
每个 DGP 合约有一个制定的地址，该地址包含了所有 DGP 共识数据。
这些数据应该直接从 RLP 数据库读取，而不需要执行 EVM。
每个 DGP 记录将包含 2 个项目：

1. 区块高度
2. 在指定的区块高度之后使用新的参数。

这些记录将作为数组顺序存储在内存中。还有一些数据，其中包括数据长度，这些数据会被标准化。

DGP 框架可以发现存储在 RLP 的参数，使用这些参数也很容易确定当前块的 Gas 计划

注意事项

1. 应该可以有多个 DPG 参数记录
2. 缓存参数，但当一个消息发送到 DGP 合约的同时，应该使缓存失效。
3. 如果 DGP 记录有可追溯的区块高度，应立即生效且不影响先前的区块。
4. 务必注意无效数据，我们应该优雅的忽略无效的 DGP 记录。
5. 对于 Gas 计划，如果一个 Gas 计划的记录表示使用少于默认模版的参数，那么该值应该可以被忽略并可以设置为模版参数的值。
6. 旧的 DGP 数据不可以被覆盖，只能追加。因此，当 DGP 记录激活，没有数据被修改或者删除，只有数据被添加。

QTUMCORE-56

为qtum支付创建receive_server

创建新的receive_server用以从外部程序支持qtum pay

服务器可以根据外部app的请求为交易创建地址，也会将所有资金转入注册地址。

QTUMCORE-57

开发qtum pay的移动端sdk

开发移动端 sdk (安卓和iOS) 支持通过量子链进行app内购买

移动端sdk需要做以下步骤:

1. 向receive_server 发送带参数的请求生成新地址。
2. 向receive_server 发送请求来检查所需的交易状态
3. 在app重装或更换设备时为用户恢复app内的购买记录

QTUMCORE-58

添加额外的AAL和压缩交易验证规则

我们实现了AAL，但是它的验证规则并不安全，我们在收到一个区块之后还要执行以下操作：

为了实现此操作，应该逐个删除先前的OP_TXHASH，并且验证所有VM的未来交易类型

首先确保 stateRoot 哈希和 utxoRoot 哈希参与区块哈希的计算

1. 使用 checkblock 函数来验证区块，并且在验证交易脚本和合约执行之前执行验证其它步骤
2. 创建一个完整的新区块，并且拷贝原始区块的所有头信息到新区块。
3. 从原始区块添加 coinbase 交易和 stake 交易到新区块
4. 逐一把旧区块的交易添加到新区块，并且执行合约。
5. 所有预期的 OP_TXHASH 交易应该按照创建顺序被添加到新区块，如果在原始区块中出现其它非预期的 OP_TXHASH 交易，新区块应该拒绝这些交易。
6. 处理每一个交易，直到没有剩余交易。
7. 创建新的 merklehash，并赋值到区块头。
8. 计算 stateRoot 哈希
9. 比较新区块的哈希值和旧区块的哈希值
10. 如果新区块和旧区块的哈希值相等则接收该区块，否则拒绝。

伪代码：

```
block = receiveBlock();
CheckBlock(block)
testBlock = new Block();
testBlock.header=block.header
testBlock.MerkelRoot=0; //make 0 because no transactions in block yet
foreach(tx in block){
    if(tx.hasExec()){
        testblock.Add(tx) //add contract tx to block
        result = tx.exec();
        if(result.hasOpHashTx()){
            foreach(opHashTx in result){
                testblock.Add(opHashTx); //add resulting condensing tx from
contract execution (I think there can be more than 1 if a single tx has
multiple EVM execs?)
            }
        }
    }else if(tx.isContractSpend()){
        //don't add spends
    }else{
        testblock.Add(tx) //add any other tx type, non-standard, pubkeyhash,
etc
    }
}
testBlock.calculateMerkelRoot();
testBlock.calculateStateRoots();
assert(testBlock.hash() == block.hash())
```

QTUMCORE-59

压缩交易的正确性验证

矿工将压缩交易添加入区块后，vouts替代的风险亦然存在

结果是:
用户的余额可能会丢失
dbUTXO 和 dbState 中的数据会被破坏.

环境

调整从压缩交易将所有余额发送给矿工的方式

测试步骤
Create contract:
contract Temp {
function () payable {}
}
Mining created block.
Adding balances to created contract.
Mining created block.

预期结果：
Block not accepted.

QTUMCORE-60

Gas 计划实现细节

下面的工作需要在子模块中完成，且尽量简洁，还要尽可能少的修改主代码，
例如使用 EVM 内部的函数和类，并将定义放在一个新文件。这样可以保证 EVM 代码简洁以备将来更新。

1-创建一个新的量子链创世文件：libethashseal/genesis/qtumMainNetwork.cpp
这个文件基于 "libethashseal/genesis/mainNetwork.cpp" 并做出一下修改：

首先，我们修改一些常量名：

static dev::h256 const c_genesisStateRootQtumMainNetwork(""); // stateRootHash should be placed here after its computation
static std::string const c_genesisInfoQtumMainNetwork = std::string() +

然后是参数部分：

```
"homsteadForkBlock": "0x0",
                    "daoHardforkBlock": "0xffffffffffffffff",
                    "EIP150ForkBlock": "0x0",
                    "EIP158ForkBlock": "0x0",
                    "registrar": ""
```

还有 genesis 部分：:

```
"extraData":
"0x5174756d4d61696e4e6574c9987fd35877cdbbbb84ffeb5315ab1f86c21398c0",
```

和账户部分：

保留前4个账户，移除其他的帐户。

然后添加了5个帐户：dgp-template.sol.js的所有实例
(https://github.com/qtumproject/qtum-dgp)


```
"0000000000000000000000000000000000000080": {
"code": "0x6060604052361561011057600357c010000000....",
"storage":
"{"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b":
{"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"}}"
},
"0000000000000000000000000000000000000081": {
"code": "0x6060604052361561011057600357c010000000....",
"storage":
"{"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b":
{"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"}}"
},
"0000000000000000000000000000000000000082": {
"code": "0x6060604052361561011057600357c010000000....",
"storage":
"{"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b":
{"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"}}"
},
"0000000000000000000000000000000000000083": {
"code": "0x6060604052361561011057600357c010000000....",
"storage":
"{"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b":
{"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"}}"
},
"0000000000000000000000000000000000000084": {
"code": "0x6060604052361561011057600357c010000000....",
"storage":
"{"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b":
{"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"}}"
}
```


这些帐户将用于 Gas 计划，最低 Gas 费用和区块大小等。这个任务只关注 Gas 计划。

代码应该包含已部署的 DGP 合约代码（可以调用 getaccountinfo 获取，不要忘记加上 0x 前缀）

存储格式和可能的代码应该略有不同，有关格式的详细信息，请参阅
libethereum/Account.cpp 的 75 行到 106 行。

这里的目标是将两个合约作为 genesis state 的一部分，并可以通过 RPC 调用

不要忘记更新常量用新的 stateRootHash 来更新常量
c_genesisStateRootQtumMainNetwork，还有核心创世 stateRootHash 和测试哈希。

2. 相关修改：

在/libethashseal/GenesisInfo.h中，

从区块0到ENUM(枚举类型)网络中添加并激活 qtumMainNetwork = 9, ///QTUM Homestead + EIP150 + EIP158规则。

在文件: /libethashseal/GenesisInfo.cpp
中

添加
#include "genesis/qtumMainNetwork.cpp"

add
```
case Network::qtumMainNetwork: return c_genesisInfoQtumMainNetwork;
```

to std::string const& dev::eth::genesisInfo(Network _n) cases

and
```
case Network::qtumMainNetwork: return
c_genesisStateRootQtumMainNetwork;
```

to h256 const& dev::eth::genesisStateRoot(Network _n) cases

3. 在validation.cpp文件中，用dev::eth::Network::qtumMainNetwork替换
dev::eth::Network::HomesteadTest，这样我们可以在量子链甚至任何需要的地方使用新的genesis。

4. 继续之前，确保编译子模块和内核代码，并且通过测试。确保子模块和内核代码的合约可以用地址来调用。

5. 最有趣的部分是我们需要实现 DGP 核心功能：从合约读取参数 Params。

获取参数的方法是调用方法 globalState->storage(addrAccount)从地址
0000000000000000000000000000000000000080 的存储中读取。

首先，我们从地址 0000000000000000000000000000000000000080 获取存储空间，重新构造 paramsHistory 数组（可以参考 DGP 合约，使用已存在的函数来解析存储内容或者创建一个 storage 来进行重新构造）
然后，我们创建一个参数为 blockNumber 的函数，该函数功能如下：

1. 这个函数解析数组paramsHistory，返回Gas计划合约的当前地址。

实现这个函数调用的一个比较好的位置是文件 libethcore/SealEngine.cpp 的 63 行之前，在函数 EVMSchedule const& SealEngineBase::evmSchedule(EnvInfo const& _envInfo) const 中。
调用函数 qtumDGP->getGasSchedule(u256 blockHeight)（我们把这个函数放到文件 qtumDGP.h 和 qtumDGP.cpp 中，以便管理所有后续加入的 DGP 函数）

§ 如果地址返回值不是null(即!=0000000000000000000000000000000000000000)，这个函数将调用globalState->storage(scheduleAddress)或者其他方法获取返回地址的storage，然后通过解析storage获取每个操作的Gas值，最后返回解析好的数据。

§ 如果地址返回值是null，使用在内核代码中硬编码的schedule,这个函数返回false，并且程序继续执行。

首先测试会返回空地址，然后需要部署 dgp-template.sol.js 合约，给地址 0000000000000000000000000000000000000080 添加管理员和新的合约地址，如果这个函数再次调用，应该返回新的合约地址。

通过部署两个 DGP 合约并把它们添加到不同的区块来检查它们，并且分析 storge 的变化来了解 storage 是如何被解析。

主 DGP 存储样例：

```
"storage": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563": {

"0000000000000000000000000000000000000000000000000000000000000000":
"0000000000000000000000000000000000000000000000000000000000000005"
    },

"45991fce5fc3033a0031207ac65e4c413069a0cd6e6bc7664254f8c4341cfe66": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e56a":
"000000000000000000000009c63b222d9db1de070e45802d7fd636cb5da922a"
    },
```

```
"510e4e770828ddbf7f7b00ab00a9f6adaf81c0dc9cc85f1f8249c256942d61d9": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563":
"00000000000000000000000000000000000000000000000000000000000000ef"
    },

"51bdce570797d7347ee2c632abdbe21de6c1cddf1026b9348df268225cf35eed": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e56b":
"00000000000000000000000000000000000000000000000000000000000000f3"
    },

"5306a7ea1091503364459f70885dc372117f70834621ea9300aa244571124d0a": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e568":
"0000000000000000000000008c63b222d9db1de070e45802d7fd636cb5da922a"
    },

"63d75db57ae45c3799740c3cd8dcee96a498324843d79ae390adc81d74b52f13": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e565":
"00000000000000000000000000000000000000000000000000000000000000f0"
    },

"68ebfc8da80bd809b12832608f406ef96007b3a567d97edcfc62f0f6f6a6d8fa": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e566":
"0000000000000000000000007c63b222d9db1de070e45802d7fd636cb5da922a"
    },

"6c13d8c1c5df666ea9ca2a428504a3776c8ca01021c3a1524ca7d765f600979a": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e564":
"0000000000000000000000006c63b222d9db1de070e45802d7fd636cb5da922a"
    },

"7b5fcc8f73196524ea5f04c38888c2f09c6cbef411cb31e259d35b56e3d0047b": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e56c":
"0000000000000000000000001063b222d9db1de070e45802d7fd636cb5da922a"
    },

"8a35acfbc15ff81a39ae7d344fd709f28e8600b4aa8c65c6b64bfe7fe36bd19b": {
```

```
"0000000000000000000000000000000000000000000000000000000000000004":
"0000000000000000000000000000000000000000000000000000000000000001"
    },

"9c418048a637d1641c6d732dd38174732bbf7b47a1cf6d5f65895384518b07d9": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e567":
"00000000000000000000000000000000000000000000000000000000000000f1"
    },

"aaab8540682e3a537d17674663ea013e92c83fdd69958f314b4521edb3b76f1a": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e569":
"00000000000000000000000000000000000000000000000000000000000000f2"
    },

"b10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6": {

"0000000000000000000000000000000000000000000000000000000000000001":
"0000000000000000000000000000000000000000000000000000000000000001"
    },

"b5d9d894133a730aa651ef62d26b0ffa846233c74177a591a4a896adfda97d22": {

"b10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6":
"000000000000000000000008c53de5f85032a9cba46ccbdc2b7c2598ec58f42"
    },

"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b": {

"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"
    }
  },
```


你可以看到，paramsHistory数组元素如下：


```
   "63d75db57ae45c3799740c3cd8dcee96a498324843d79ae390adc81d74b52f13": {
      "290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e565":
"00000000000000000000000000000000000000000000000000000000000000f0"
    },
    "68ebfc8da80bd809b12832608f406ef96007b3a567d97edcfc62f0f6f6a6d8fa": {
      "290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e566":
"000000000000000000000007c63b222d9db1de070e45802d7fd636cb5da922a"
```

上面展示了 paramsHistory 数组元素，元素是一个结构体：

```
struct paramsInstance{
        uint blockHeight;
        address paramsAddress;
}
```

我们可以看到数组的所有元素，元素的键值相差 1。

样例 schedule state 更简单：

```
  "storage": {
    "290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563": {
      "0000000000000000000000000000000000000000000000000000000000000000":
"0000000a0000000a0000000a0000000a0000000a0000000a0000000a0000000a"
    },
    "405787fa12a823e0f2b7631cc41b3ba8828b3321ca811111fa75cd3aa3bb5ace": {
      "0000000000000000000000000000000000000000000000000000000000000002":
"00002328000008fc0000002800007d000000017700000008000017700000001"
    },
    "8a35acfbc15ff81a39ae7d344fd709f28e8600b4aa8c65c6b64bfe7fe36bd19b": {
      "0000000000000000000000000000000000000000000000000000000000000004":
"00000000ffffffff000000000000001400000014000000140000000300000044"
    },
    "b10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6": {
      "0000000000000000000000000000000000000000000000000000000000000001":
"00003a980000138800004e200000003200000006000001e0000000a0000000a"
    },
    "c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b": {
      "0000000000000000000000000000000000000000000000000000000000000003":
"000000040000cf0800005208000000c800000200000000300005dc0000061a8"
    }
  },
```

我们需要对 0～4 的键值排序，然后从右开始解析 Gps 值（4 字节来表示），所有上面的第一个 Gas 值是 0000000a, 第八个也是 0000000a。

同时欢迎讨论实现方法，也许你的想法更好。

QTUMCORE-61

实现 MPoS 奖励系统

MPoS 是量子链设计的全新的奖励系统，该系统使得对量子链的 DoS 攻击更昂贵。

目标

1. 为了防止恶意矿工攻击量子链网络，需要昂贵的Gas费用来验证区块，然后再通过挖矿返还给矿工

2. 尽量使得DoS变得困难和昂贵

步骤

1. 当一个股东挖到一个区块，他只收到一部分 PoS 奖励和 Gas 费，剩余的奖励和 Gas 费与其他 9 个人分享。
2. 当一个股东挖到一个区块时，他的权益脚本就会被注册用以接收后续第 10 个区块的奖励。
3. 每一个区块都会有 10 个奖励获得者（一个区块创建者和 9 个共同权益者）
4. 获得之后的 9 个区块奖励之后，权益所有者的脚本将被移除，并会用另一个脚本替换。

添加并替换它

5. 如果一个权益脚本在10个区块之间再挖到超过1个区块，他就可以获得2倍的利息。然而，最早的权益脚本实例超过510个区块,将不再计算利息，奖励也会降低到正常值。相同的权益脚本不应该合并到一个UTXO，但奖励可以重复。

6. 为了防止恶意或者粗心的权益者以免费或者低价的Gas费用来执行（挖矿）合约，将会添加“最低Gas价格”。这个价格将根据情况进行提高或者降低。

Example

1. X挖到第1000个块
2. X获得10%的挖矿奖励和额外的Gas费用，但权益交易包括至少9个其他共同权益者的输出。
3. X 挖到另一个高度为1004块
4. X又获得10%的挖矿奖励和额外的Gas费用，但权益交易包括至少9个其他共同权益者的输出。
5. X没有挖到任何区块
6. 在区块1500，1501，1502和1503处，X和区块的创造者以及其他8个权益共同者获得每个区块的收入(10%奖励和Gas费用)。
7. 在1504到1509之间的区块，X和区块的创造者以及其他7个权益共同者获得每个区块的2倍的收入(10%奖励和Gas费用)。

8． 在1510到1513之间的区块，X和区块创造者以及其他8个权益共同者获得每个区块的1个UTXO。
9. X的MPoS结束，它不再获得任何奖励。

共识规则

- 发给MPoS权益者的奖励必须来自权益区块
- 权益交易必须包括 10 个输出（1 个来自创造者，其他 9 个来自 MPoS 权益者）
- 权益交易的前 10 个输出是共识的关键，必须是按照正确的顺序输出，并且创造者是第一个输出。
- 权益交易可以包含其他输出，比如把一个大的 UTXO 分成多个 UTXO。第 10 个输出之后，只要输出值不会导致交易超过输入值+区块奖励+费用，输出就不应该有其他共识规则。
- 最开始的 500 个区块没有使用 MPoS 机制，所以区块创建者获取所有奖励。

请注意，投标交易的第0个输出应该是空的（因此它被检测为投注交易）。 这个故事打破了Blackcoin的一些“mutlisig staking”（误导性名称）功能。 我们可以在以后的故事中解决这个问题

s

QTUMCORE-62

从交易中移除时间戳(timstamps)

由于PoSv3的实现对交易时间戳（timestamps）没有强烈的目的或者用例，而且会更复杂，比如微支付通道。

应该删除该字段，并且使用在PoS内核哈希中的区块时间戳，而不是使用事务时间戳（有一个共识规则强制他们是相等的，所以这里没有逻辑上的改变）。 如果在内核哈希中使用的coinstake之外存在任何安全问题，则应引起讨论。 （据我所知，没有）

QTUMCORE-63

正确描述Qt交易清单中的合同交易

如果您在现在这个节点部署或发送合约，那么该交易将没有具体描述或标记。 此外，应该有一个迹象表明，AAL退款中的“采矿交易”是退款，而不是钱包自动采矿。

QTUMCORE-71

为QtumDGP创建测试

Qtum 测试网络预计 6 月份发布，感谢您的支持！

如果有兴趣加入 Qtum, 请发简历到 job@qtum.org