Qtum 区块链指南 0.1

目录

1. Qtum 区块链简介 4
1.1 区块链基本概念 4
1.2 Qtum 区块头 6
1.3 Qtum 区块 6
1.4 MPoS 共识算法 7
1.5 AAL 和智能合约 8
1.6 分布式自治协议 10
2. 交易 11
2.1 UTXO 账户模型 11
2.2 合约账户模型 11
2.3 交易类型 12
2.3.1 普通交易 12
2.3.2 合约交易 17
3. Qtum 全节点和钱包 20
3.1 Ubuntu16.04 上安装 Qtum 20
3.1.1 安装依赖 20
3.1.2 拉取 Qtum 源代码 20
3.1.3 编译程序 20
3.2 Mac OS 上安装 Qtum 21
3.2.1 环境准备 21
3.2.2 安装 Homebrew 21
3.2.3 安装命令行工具 21
3.2.4 安装依赖 21
3.2.5 拉取 Qtum 代码 21
3.2.6 编译代码 21
3.3 Qtum 钱包 21
3.3.1 qtumd 自带钱包 22
3.3.2 PC 钱包 22

3.3.2.1 往特定的地址发送 Qtum 23
3.3.2.2 接收 Qtum 24
4. 智能合约 25
4.1 智能合约编写和编译 25
4.2 部署智能合约 26
4.3 和智能合约交互 27
4.3.1 ABI 数据生成 27
4.3.2 使用 sendtocontract 和智能合约交互 28
4.3.3 使用 callcontract 查看智能合约执行结果 29
4.4 在 PC 钱包上使用智能合约 30
4.4.1 部署智能合约 31
4.4.2 和智能合约交互 31
4.4.3 查看智能合约执行结果 32

Qtum 区块链，又称为量子链，是一个基于未花费交易输出(Unspent Transaction Output，UTXO)和权益证明(Proof of Stake，PoS)的智能合约平台，融合了比特币和以太坊生态系统各自的优点 。智能合约可以应用在诸多行业，如金融科技，物联网和身份认证等。智能合约的核心技术是分布式账本，也就是我们常常提到的区块链。Qtum 区块链实现了完整的智能合约功能，通过账户抽象层(Account Abstraction Layer，AAL)技术把 UTXO 模型转换成可供以太坊虚拟机(Ethereum Virtual Machine，EVM)执行智能合约的账户模型，合约开发者不需关心对合约操作相关的 UTXO 转换细节，即可使用 EVM 的特性进行开发而且兼容现有以太坊的智能合约。

Qtum 量子链采用了互惠权益证明(Mutualized Proof Of Stake, MPoS)共识机制，使得在智能合约下实现更安全的 PoS 共识。另外，独创性的提出和实现了分布式自治协议(Decentralized Governance Protocol, DGP)，DGP 是通过内嵌到创世区块的智能合约来治理区块链网络的参数，去中心化的网络自治机制使得区块链网络在一定程度上实现自动升级和快速迭代而无需进行软件升级。

本指南介绍 Qtum 区块链基本概念和使用的技术，介绍了 Qtum 的交易类型和智能合约的使用，对 Qtum 区块链的使用可以有一个基本的了解。

1. Qtum 区块链简介

1.1 区块链基本概念

区块链是一种去中心化、分布式的账本，账本以冗余的方式存储在所有参与节点。区块链主要由以下底层技术组成：密码学、共识算法、点对点网络。就数据组织方式来看区块链是由一系列的区块链接形成的一种数据结构。最底层的第一个区块是创世区块，随后每个新区块都被放置在前一个区块之上，这些区块有序地链接在一起形成链。区块之间是通过哈希指针进行连接的，哈希指针表示从原区块指向目标区块并且在目标区块存储有原区块的哈希值。区块由区块头和区块记录组成，区块头用于描述本区块的组成信息，区块头最重要的用途是用作记录前一区块头哈希值和区块记录的根哈希值。

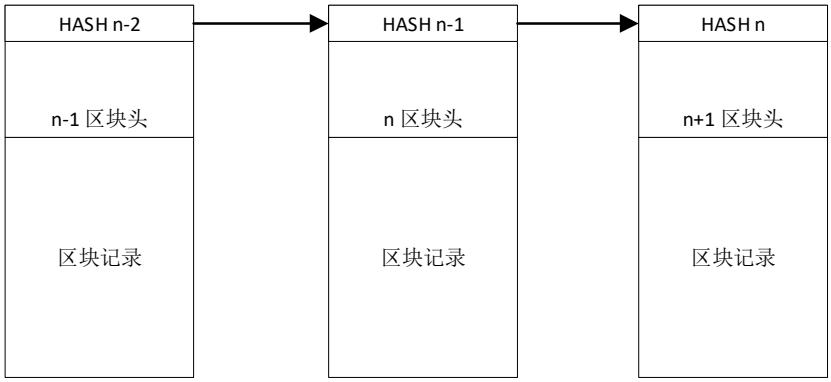

Figure 1 简化版区块链示意图

一个或多个新的交易收集到一起成为了一个区块的交易数据，这些数据作为区块记录。为了对数据完整性进行保护，每个交易的副本都会进行 SHA256 哈希，然后这些哈希作为 Merkle 树的节点，对 Merkle 树节点进行从下到上的哈希运算，最终在 Merkle 根节点得到一个根哈希值。该哈希写入到区块头，而区块头也进行哈希运算得到一个哈希值，该哈希值写入下一个区块头部，因此通过写入区块头的根哈希值就可以验证区块记录。

由于区块头里的哈希值与前一个区块相关，如果前一个区块内容发生变化，其哈希值也会发生变化，因此会改变当前区块的哈希值，这个改变会传导到下一个区块，并再传导到下下一个区块，以此类推，直到最后一个区块。因此当改变某个区块记录时，会破坏该区块之后的链接关系，除非对该区块之后所有区块进行相应的更改，即重新计算符合共识的哈希值。共识算法是用来确保重新计算哈希值在一个合适的难度，这样如果想改变区块内容，必须在计算能力或权益大小上超过共识难度，多个区块共识难度的不断积累，使得在概率上保证了最长链的安全性，这是就是工作量证明(Proof of Work，PoW)和权益证明(PoS)共识算法的作用原理。比特币区块链通过工作量证明，而 Qtum 区块链采用的是权益证明机制改进算法：互惠权益证明(MPoS)。

点对点网络是实现 Qtum 节点之间进行通信的方法，交易广播、区块同步与广播等都通过点对点网络来实现。通过点对点网络，每个节点共享同一份的账本，并对账本进行验证，共识算法和共识规则是进行验证的标准，由于共识的一致，和密码学结合确保了作恶节点无法篡改和作假账本上的数据。可以看到，密码学、共识算法和点对点网络的结合确保了区块链在去中心化的条件下能实现可信的账本。Qtum 在此基础上增加了可以运行智能合约虚拟机，从而实现了一个高度智能化的价值传输网络。

1.2 Qtum 区块头

Qtum 量子链的区块头描述了区块的状态记录，用于管理记录区块交易结果相关的信息和共识算法所需的数据。以下是 Qtum 区块头数据结构的定义：

nVersion	版本信息
hashPrevBlocK	前一区块头哈希值
hashMerkleRoot	Merkle树根哈希值
nTime	区块产生时间
nBits	区块难度
nNonce	计算哈希值的变量
hashStateRoot	EVM状态根哈希值
hashUTXORoot	UTXO根哈希值
prevoutStake	前一区块Stake的输入
vchBlockSig	PoS使用的区块签名

Figure 2 Qtum 区块头数据结构

其中 nVersion 版本信息描述了当前 Qtum 区块链软件的版本，hashPrevBlock 记录前一区块头哈希值，hashMerkleRoot 记录本区块交易 ID 生成的 Merkle 树根节点哈希值，nTime 记录区块产生的时间，nBit 记录当前计算区块头哈希值的难度，nNonce 与比特币兼容，但 PoS 阶段未使用，hashStateRoot 记录本区块 EVM 执行完成后的根状态哈希值，hashUTXORoot 记录本区块 EVM 执行完后产生的 UTXO 根状态哈希值，prevoutStake 为前一区块用于 PoS 过程的交易输出点，vchBlockSig 是 PoS 时使用的区块签名。

1.3 Qtum 区块

Qtum 交易的基本单位是未花费的一个交易输出，简称 UTXO。UTXO 是不能再分割，并记录在区块链中。一个交易的输入是一个或多个 UTXO，生成交易时对这些 UTXO 进行签名用于解锁输入，签名标志着对某个地址上 Qtum 的使用许可。这些签名后的交易广播到 Qtum 网络中。网络的每一个全节点都可以接收、验证、和在网络中转发这些交易。Qtum 区块链每隔一段时间就有挖矿节点产生新的区块，把验证通过的交易记录到区块。

Merkle 树是一种哈希二叉树，用于快速归纳和校验大规模数据完整性。这种二叉树的节点包含哈希值。在 Qtum 区块链中，Merkle 树用来组织每个区块中的交易标识(txid)。txid 是对交易进行两次哈希算法生成。Merkle 树是自底向上构建的，通过对相邻叶子节点的 txid 配对然后做哈希运算，生成为父节点，对所有叶子节点重复这个过程，并对生成的父节点进行类似的操作，只到剩下顶部的一个节点，最终构建了 merkle 树。如下图所示:

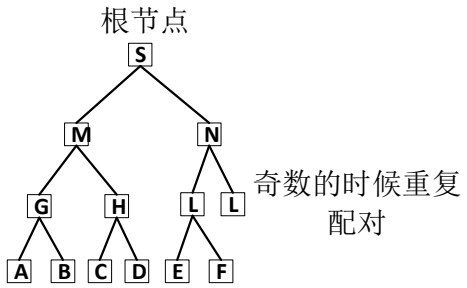

Figure 3 Merkle 树

在简化支付验证（SPV）提案中指出，Merkle 树允许客户端通过一个完整节点从一个区块头中获取其 Merkle 根节点和中间哈希值列表来验证一个交易被包含在这个区块中。这个完整节点并不需要是可信的，因为伪造区块头十分困难而中间哈希值是不可能被伪造的，否则验证就会失败。

Qtum 的 PoS 区块至少包含 2 笔交易。这些交易的第一笔是一个特殊的 coinbase 交易，该交易没有 Qtum 输出。第二笔交易是 coin stake 交易，它包含了这个区块所有交易费和区块奖励。coin stake 交易在之后的 500 个区块内之后才能花费，在 Qt 钱包上显示 staking 的 Qtum 就是参 stake 之后不够 500 个区块锁定的。

1.4 MPoS 共识算法

Qtum 基本的共识算法基于 PoSv3 共识机制。关于共识机制孰优孰劣的讨论一直在进行中，最常被讨论的共识机制有：工作量证明机制 PoW 、权益共识机制 PoS 、动态权益共识机制和 HyperLedger 的拜占庭容错机制 。共识机制的目的是利用分布式算法达成数据的一致性。正如 Fischer Lynch & Paterson 定理指出，除非所有节点达成 100%一致，否则无法达成共识。

在比特币网络中，矿工通过工作量证明机制哈希碰撞参与交易验证。当矿工计算出满足一定条件的哈希值时，矿工可以向全网宣布新区块的诞生：

$$Hash(BlockHeader) \leq M/D$$

M 是矿工数量，D 是挖矿难度，Hash()代表 SHA256 哈希运算，输出值范围为[0，M]。比特币的 SHA-256 哈希运算满足快速验证（方便网络中每个节点对运算结果进行验证）、可调整的难度值（可以根据全网算力进行调整）和相对公平性（也就是说，每个矿工解决哈希计算的概率与其算力成正比）。

在 PoSv3 交易中，新区块的产生必须满足下列条件：

ProofHash < coins × target

在 ProofHash 中，StakeModifier 利用未花费输出和当前时间进行运算。由于 Qtum 实现了智能合约，交易处理需要花更多的时间去和智能合约交互和执行，因此需要更动态的费用机制。这其中有影响 Qtum 的几个安全隐患。一个比较大的隐患是，攻击者可以通过支付昂贵费用执行恶意程序，但由于这些费用会归于区块生产者，最终攻击者付出的费用会变得很小。基于现有 PoS 系统的加密货币由于不支持图灵完备的智能合约 VM，不会受到这类攻击的影响。Qtum 是第一个基于 PoS 和以太坊智合约的区块链，实现了规范在区块链上允许的计算用途和能力的 gas 机制。

Qtum 进一步对 PoSv3 做了一个修改，用于部分解决通过垃圾交易攻击而产生的无用 satke 问题。采用 PoSv3 一致的共识机制，但改变了区块奖励和交易费用的支付体系。与区块生产者会立即收到区块奖励和交易费用相比，新的方法是奖励和交易费用在网络中多个挖矿者之间互相共享。具体说来，区块生产者在生成区块的时候会收到总费用的 1/10，而经过预定的区块确认时间之后，可以接收到另外 9 个块的费用的 1/10。

这个新的方法看起来改动很微小，但它解决了影响 Qtum 的几个安全隐患。一个比较大的隐患是，攻击者可以通过买入足够的币从而成为有可能的区块生产者，通过某些 EVM 操作码使网络变得容易遭受 DoS 攻击。这只需要很小的代价，甚至恶意交易需要消耗交易费用和 gas，攻击者可以通过支付昂贵费用执行恶意程序，但由于这些费用会归于区块生产者，最终攻击者付出的费用会变得很小。使用 MPoS,就不会出现这种情况，产生区块是只能收到 1/10 的 gas 费用，除非他能再挖出连续的 9 个区块，否则其余 9/10 的 gas 费用会因分给网络上其他的挖矿者而丢失。因此会导致这种类型的垃圾交易攻击会变得很昂贵。

当一个区块产生时，区块生产者的奖励交易必须包含至少 10 个输出。第一个输出属于本区块生产者，可以接收自己用于 stake 的币和 1/10 的交易费用，其他九个输出属于 500 个区块之前的区块生产者。当一个区块产生时，增加一个奖励费用的接收者，同时减去另外一个接收者，因此总接收者数量一直有 10 个。区块生产者另外 9/10 的奖励将在 500 个区块之后，即(+501，+502，...，+509)每个区块奖励的 1/10。

1.5 AAL 和智能合约

智能合约（Smart Contract），以编程方式定义的一系列承诺，包括合约参与方可以在上面执行这些承诺的协议。通过区块链可以实现这种智能合约的编程，并在区块链上执行，因此一旦设立指定后，能够无需中介的参与自动执行。

Qtum 实现了满足以太坊虚拟机（EVM）规范的智能合约功能，在 Qtum 上可以部署和调用智能合约。以太坊虚拟机使用了 256 比特长度的机器码，是一种基于堆栈的虚拟机，用于执行以太坊智能合约。由于 EVM 是针对以太坊体系设计的，因此使用了以太坊账户模型（Account Model）进行价值传输。而 Qtum 量子链的设计是基于比特币 UTXO 模型，所以在量子链模型设计中加入了账户抽象层（Account Abstraction Layer），用于将 UTXO 模型转换成可供 EVM 执行的账户模型。账户抽象层可以隐藏某些特定功能部署细节，并为增强互操作性和平台独立性建立关注划分。

Qtum 量子链中的所有交易使用比特币脚本语言，并在其基础上进行了拓展，加入了三个全新的操作符。

- OP_CREATE – 用于执行 EVM 智能合约的创建，把通过交易传输的字节代码存放到合约 RLP 数据库，并生成一个合约账号；
- OP_CALL – 用于传递调用智能合约所需要的相关数据（即 EVM 中的 CALLERDATA）和地址信息，并执行合约中的代码内容。该操作符还可为智能合约发送资金。
- OP_SPEND – 将当前合约的 ID 哈希值作为输入的交易 HASH，或发送到合约的 UTXO 的交易 HASH，然后使用 OP_SPEND 作为花费指令构建交易脚本。

在比特币系统中，只有当解锁脚本（ScriptSig）与锁定脚本（ScriptPubKey）验证通过后才能花费相对应的交易输出。举例来说，锁定脚本通常会把一个交易输出锁定到一个比特币地址上（公钥的哈希值），只有当解锁脚本和锁定脚本的设定条件相符时，执行组合脚本才会显示结果为真（系统返回值为 1），这样相对应的交易输出才会被花费。

而在 Qtum 系统中，我们更强调智能合约执行的及时性。因此我们在锁定脚本中加入了 OP_CREATE 和 OP_CALL 操作符。当 Qtum 系统检测到该操作符时，全网节点就会执行该交易。这样一来，比特币脚本扮演的角色，更多的是将相关数据传送至 EVM，而不仅仅作为一种编码语言。与以太坊执行智能合约一样，由 OP_CREATE 和 OP_CALL 操作符触发的合约，EVM 会在各自的状态数据库中更改其状态。

考虑到量子链智能合约的易用性，需要对触发智能合约的数据以及数据来源的公钥哈希值进行验证。为了防止量子链上 UTXO 所占比例过大，将 OP_CREATE 和 OP_CALL 的交易输出也设计成可花费的，OP_CALL 的输出可以为其他合约或公钥哈希地址发送资金。

对于智能合约的开发者来说，EVM 的账户模型相对简单。它支持合约余额的查询，还可为其他合约发送资金等操作。对于在 Qtum 上创建的智能合约，系统会生成一个交易哈希值用于合约的调用。新合约的初始余额为 0（目前不支持非 0 初始余额的合约）。为了满足合约发送资金的需求，Qtum 使用 OP_CALL 操作符来创建交易输出。

目前 Qtum 支持使用 Solidity 语言进行编写的智能合约，并且使用比如 Remix Solidity IDE 进行开发和编译智能合约。部署智能合约或和合约进行交互可以通过 Qtum 提供的 RPC 调用或 PC 钱包进行。

1.6 分布式自治协议

Qtum 区块链的分布式自治协议(Decentralized Governance Protocol，DGP)是通过智能合约实现的，可管理的每个区块链网络功能都由独立的 DGP 智能合约(采用 DGP-template 描述)控制，这意味着每个功能有独立的治理、授权机制以及内置限制条件。每个 DGP 都有一个非常简单的核心治理模式：即通过具有管理或治理席位的地址来对提案进行管理，包括对投票席位本身和普通提案的管理。现在支持 0，1，2 三种类型的提案，分别对应：管理或治理席位增加、管理或治理席位删除、普通参数提修改。所有提案的设置只有具备管理席位的发送地址才有权限设置，具有管理或投票席位的发送地址才能进行投票。

对 2 类型提案，即普通类型提案的投票，至少支持以下项目:

- 每个 Qtum 虚拟机操作码对应的 Gas 价格
- 区块创建者可接受的交易对应最低 Gas 价格
- 区块大小
- 区块 gas 限制

具体的网络参数更改流程为：

设计 DGP 参数更改提案。提案中需要明确新的参数值、实施变更的起始区块以及投票过程所对应的区块数量。

- 向社区公布更改提案，并收集相关反馈；
- 根据社区反馈，对更改提案进行相应调整；
- 将最终版提案发送至 DGP 智能合约；
- 投票立即启动；
- 具有管理或治理席位的代理者可以发送一笔交易给 DGP 智能合约，对该提案进行投票；
- 在投票过程中，若提案未获得足够投票，则该提案被否决，不执行任何修改；
- 若提案得到足够同意票，则 DGP 分布式自治合约将提案中的相关数据存储在持久 RLP 特殊存储空间内；

提案投票成功后会更新到相应的变量保存，新参数会在 500 个块之后生效，以避免出现不必要的孤儿块或分叉，钱包和区块链上的所有节点可以定时检查 RLP 储存从而判断是否有新的更改发生。对于 0，1 类型的提案，即管理或治理席位增加和删除，其投票流程和普通提案类似，也是通过具有管理或治理席位的发送者投票来决定，不过当投票通过后席位的变化可以立即生效。

2. 交易

Qtum 区块链支持比特币类型交易，即 UTXO 类型的交易，目前支持比特币定义的五种 UTXO 类型的交易。UTXO 交易的基本单位是未经使用的一个交易输出，当接收比特币时，金额是在输出的 UTXO 里的，并且通过脚本来锁定，要花费这个 UTXO 需要提供解锁脚本。对于智能合约功能，当需要进行部署合约或调用合约相关功能时，是通过合约交易来触发的，合约交易的形式很像 UTXO，是通过增加了操作码进行扩展的。

2.1 UTXO 账户模型

在未花费交易输出（UTXO）模型中，交易使用未花费的比特币作为输入，此时输入的 UTXO 就会作废，而输出是另一个 UTXO, Qtum 数量上变化的结果会在新的 UTXO 记录。一定数量的 Qtum 在不同私钥持有人之间进行转移，新的未花费交易输出在交易中花费，并记录在区块上。在交易中，UTXO 可用交易接收方公钥地址生成的秘钥进行解锁。Qtum 处理交易时利用脚本语言只能进行有限的操作，并以堆栈的形式进行数据处理 ，并遵循“后进先出”（FIFO）原则。

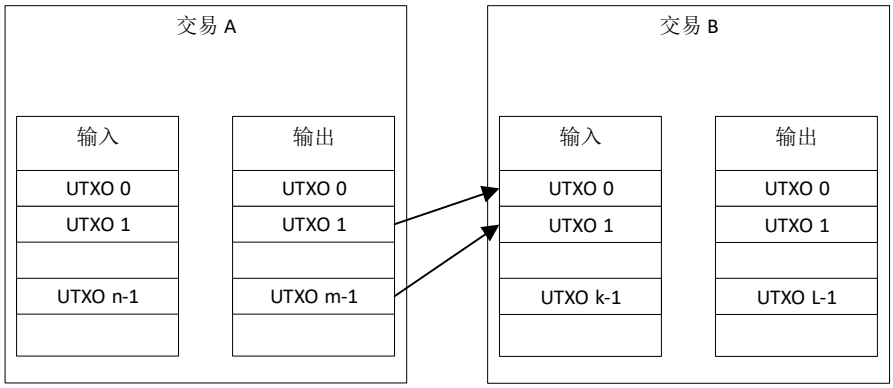

Figure 4 基于 UTXO 模型的交易

UTXO 模型有诸多优势： 任何人可以通过比特币公共账本对每一笔交易历史进行查询，UTXO 有良好的可拓展性，能够同时处理多个地址发起的交易请求。此外，UTXO 模型也提供了隐私保护，用户可以使用变更地址作为 UTXO 输出。

每一笔 UTXO 交易都会记录在区块账本上，只要知道了私钥就可以扫描账本获得所有属于该私钥用户的 UTXO。用户的所有余额，是所有属于用户私钥的 UTXO 金额的总和，钱包通过扫描整个区块链收集到用户所有的 UTXO 来计算该用户的余额。

2.2 合约账户模型

比特币脚本语言并不是图灵完备的，无法实现循环功能。这极大地制约了交易执行量和交易复杂度。因此 Qtum 量子链在 UTXO 模型的基础上加入抽象账户层设计，在基于 UTXO 模型的交易上实现智能合约平台。

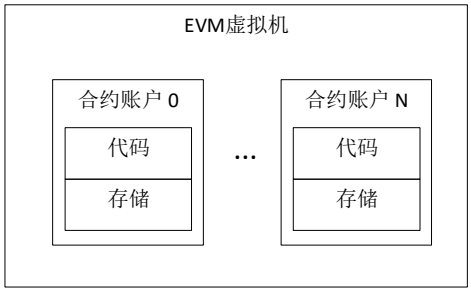

Figure 5 Qtum 合约账户

Qtum 量子链智能合约功能使用了以太坊虚拟机(EVM)，EVM 使用了与 UTXO 不同的账户模型，EVM 使用基于账户（Account）的模型。具体来说，以太坊通过账户状态的改变进行价值和信息的交换与传输，并通过长度为 20 字节的随机数作为指针以确保交易处理的唯一性。用于供内部使用的费用是以 Qtum 计价，通过合约交易中的参数 GasPrice 和 GasLimit 来指定。对于以太坊 EVM 合约代码是可选的，但对 Qtum 的合约账号来说是强制的，即账号的生成必定伴随有合约代码，生成账号时存储默认为空。

以太坊账户有两种类型，一种由外部私钥控制的外部账户，另一种由合约代码控制的合约账户。外部账户用于信息传输的创建、交易签名。合约账户用于收到内部存储读写操作信息后创建合约或发送其他信息。以太坊中的账户余额管理与日常生活中的银行账户管理相类似。每一个新产生的区块都有可能影响其他账户的全局状态。每个账户都有各自的余额、存储和代码空间用于调用其他账户或地址。Qtum 智能合约账户和以太坊一致，通过长度为 20 字节的随机数作为地址，但没有外部私钥控制的外部账户，UTXO 账户和以太坊的外部账户类似，但通过脚本语言，其功能比以太坊强大。

2.3 交易类型

2.3.1 普通交易

Qtum 区块链的普通交易和比特币的交易是兼容的，目前支持比特币定义的五种 UTXO 类型的交易，分别为：P2PKH（Pay to Public Key Hash）、P2PK （Pay to Public Key）、多重签名（少于 15 个私钥签名）、P2SH （Pay to Script Hash）和 OP_RETURN。利用这五种交易标准，客户端可以满足复杂的支付逻辑。以下举例简单说明各种交易类型的用法：

- P2PKH 交易

使用 P2PKH 交易方式，我们假设用户向虚拟 Qtum 地址 Bread Address 支付了 0.01 QTUM 购买面包。该交易的输出为：

OP_DUP OP_HASH160 <Bread Public Key Hash> OP_EQUAL OP_CHECKSIG

OP_DUP 复制堆栈顶层数据；OP_HASH160 返回公钥哈希值并存入栈顶。除了公钥哈希值，还需要数字签名和数字秘钥才能拥有 QTUM 所有权。若栈顶数据一致，则 OP_EQUAL 返回真值（1），否则返回非真值（0）。OP_CHECKSIG 生成公钥和签名，并校验交易哈希值。若一致，则返回真值。

锁定脚本相对应的解锁脚本为：

<Bread Signature> <Bread Public Key>

将上述两个脚本相结合：

<Bread Signature> <Bread Public Key> OP_DUP OP_HASH160 <Bread Public Key Hash> OP_EQUAL OP_CHECKSIG

只有当解锁脚本和锁定脚本满足预先设定的条件时，执行结合脚本的输出为真。上述例子中，当 Bread Signature 签名与 Bread Address 私钥相匹配，则返回真值。

- P2PK 交易

P2PK 交易的锁定脚本为：

<Bread Public Key> OP_CHECKSIG

对应的解锁脚本为：

<Bread Signature>

将上述两个脚本相结合：

<Bread Signature> <Bread Public Key> OP_CHECKSIG

这样只要 Bread Signature 使用 Bread Public Key 对应的私钥签名，执行结合脚本的输出为真。

- 多重签名交易

一个 M-N 类型的多重签名交易的锁定脚本为：

M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG

锁定脚本需要至少 M 个签名才能解锁。

一个设置了 K(K>=M)个签名的解锁基本为：

OP_0 <Signature 1> <Signature 2>…<Signature K>

将上述两个脚本相结合：

OP_0 <Signature 1> <Signature 2>…<Signature K> M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG

通过执行这个组合脚本，只要签名成功的个数达到 M 就会输出真。

- P2SH 交易

P2SH 交易的锁定脚本为：

OP_HASH160 <redeem scriptHash> OP_EQUAL

其中 redeem scriptHash 是解锁脚本的哈希值。

解锁脚本的形式为：

<Signature> {redeem script}

P2SH 执行等效于以下的两个验证：

1) {redeem script} OP_HASH160 <redeem scriptHash> OP_EQUAL
2) <Signature> {redeem script}

首先 1)用于验证{ redeem script }为是否真，2)再验证 redeem script 的执行结果为真，只有两个验证都为真时，整个赎回脚本才输出真。

- OP_RETURN

OP_RETURN 允许在交易输出上增加 40 字节的非交易数据，一般这个数据是一个哈希值，用于记录证明。OP_RETURN 输出的资金不能被使用，所以设置为一个金额为 0 的输出，任何输出不为 0 的 QTUM 都会消失而不可用。OP_RETURN 交易格式很简单，如下所示：

OP_RETURN <data>

下面使用 P2PKH 交易方式，使用 sendtoaddress 发送一笔交易到一个公钥哈希地址，详细的交易信息如下：

{

"txid": "41af815df6af399bec82b4e7b25587463aec6c2c6a11815ddd62aaa314b9aa0c",

"hash": "41af815df6af399bec82b4e7b25587463aec6c2c6a11815ddd62aaa314b9aa0c",

```
 "size": 225,
 "vsize": 225,
 "version": 2,
 "locktime": 720,
 "vin": [
  {
    "txid": "2dac9840b183661e5aed458133daa466765131bed7b2cd4e625ef9d477c81b71",
    "vout": 0,
    "scriptSig": {
      "asm":
"3044022073caa7d8fe0ba4a7561a6165369c32d6de48d594f0eab169aee82c915205ecc6022050
c9ee0dcc7e8a967ea64e8c40df271c479d8bed446e10d0b4da9e05352248bc[ALL]
02646b87a66a70311619fe6e1cc0300a6940fd5bc191c16f1d6d4d1cea9fbe8e74",
      "hex":
"473044022073caa7d8fe0ba4a7561a6165369c32d6de48d594f0eab169aee82c915205ecc60220
50c9ee0dcc7e8a967ea64e8c40df271c479d8bed446e10d0b4da9e05352248bc012102646b87a6
6a70311619fe6e1cc0300a6940fd5bc191c16f1d6d4d1cea9fbe8e74"
    },
    "sequence": 4294967294
  }
 ],
 "vout": [
  {
    "value": 12.50000000,
    "n": 0,
    "scriptPubKey": {
      "asm": "OP_DUP OP_HASH160 52f60b28f652ff43e07ddae02e6bb037f6937ef8
OP_EQUALVERIFY OP_CHECKSIG",
      "hex": "76a91452f60b28f652ff43e07ddae02e6bb037f6937ef888ac",
      "reqSigs": 1,
      "type": "pubkeyhash",
```

```
      "addresses": [
        "qR83KAp8u2nCxSpwTx39MSTrFedMerJydA"
      ]
    }
  },
  {
    "value": 1987.49909600,
    "n": 1,
    "scriptPubKey": {
      "asm": "OP_DUP OP_HASH160 0bb60c781c8f31be21e3901b63cb7f25dd3aeeb0
OP_EQUALVERIFY OP_CHECKSIG",
      "hex": "76a9140bb60c781c8f31be21e3901b63cb7f25dd3aeeb088ac",
      "reqSigs": 1,
      "type": "pubkeyhash",
      "addresses": [
        "qJdJg7BaPV89Em8xQZyMdHni4Ss3VG5fTo"
      ]
    }
  }
 ]
}
```


其中输入 UTXO 的 hex 格式的解锁脚本为：

使用私钥进行签名的结果：
473044022073caa7d8fe0ba4a7561a6165369c32d6de48d594f0eab169aee82c915205ecc6022050c9ee0dcc7e8a967ea64e8c40df271c479d8bed446e10d0b4da9e05352248bc0121

公钥：

02646b87a66a70311619fe6e1cc0300a6940fd5bc191c16f1d6d4d1cea9fbe8e74

组合在一起就是"scriptSig"里面的内容了。

输出的 UTXO 0 锁定脚本为：

OP_DUP OP_HASH160 52f60b28f652ff43e07ddae02e6bb037f6937ef8 OP_EQUALVERIFY OP_CHECKSIG

2.3.2 合约交易

Qtum 在设计上以比特币 UTXO 为基础账户模型实现了支持 EVM 规范的智能合约，这是通过新增 UTXO 操作码和账户抽象层(AAL)来完成的。AAL 对 UTXO 账户和 EVM 合约账户之间进行了适配，这样通过 AAL 可以使用 UTXO 交易输出实现在链上创建智能合约，发送交易到合约账户用于触发合约的执行，完成执行后 AAL 最终对执行结果进行处理并适配至 UTXO。由于采用了 AAL，合约开发者不需关心对合约操作相关的 UTXO 转换细节，即可使用 EVM 的特性进行开发而且兼容现有以太坊的智能合约。

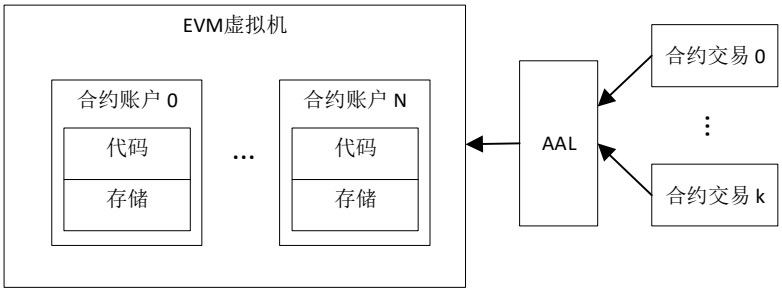

Figure 6 Qtum 合约交易处理

Qtum 针对 UTXO 交易脚本新增了三个操作码 OP_CREATE，OP_CALL 和 OP_SPEND，目的是用于为 UTXO 和 EVM 账户模型之间的转换提供操作支持。这个三个操作码分别有以下作用：OP_CREATE 用于智能合约的创建；OP_CALL 用于合约的执行；OP_SPEND 用于合约余额的花费。

产生或验证新的区块时，除了对 UTXO 交易进行常规的参数合法性、共识规则、DDOS 攻击检查等之外，还需要使用操作码检查函数判断交易输出是否包含 OP_CREATE 或 OP_CALL，分别对应着 EVM 需要执行合约创建或合约调用。在合约创建和执行前，需要进行 UTXO 交易到 EVM 模型交易的转换，之后使用构建的 EVM 执行环境和引擎，完成合约的执行。

目前合约交易相关的 qtum-cli 命令或 RPC 有：

- 合约部署： createcontract，该命令将智能合约部署到量子链上，格式如下

  createcontract "bytecode" (gaslimit gasprice "senderaddress" broadcast)

  bytecode(必选)： 智能合约编译生成的字节码

gaslimit(可选)： 最大 gas 数量，默认是 2500000

gasprice(可选)：gas 价格，以 QTUM 为单位，默认为 0.0000004

senderaddress(可选)：指定发送者的地址，默认为当前账号地址

broadcast(可选)：是否广播，默认为 true

比如，

qtum-cli createcontract "60606040525b33600060006101000a81548173ffffffffffffffffffffffffffffffffffffffff

ffffffffffff02191690836c0100000000000000000000000090810204021790555061037860016000508190555505b600c80605b6000396000f360606040526008565b600256"

或

qtum-cli createcontract "60606040525b33600060006101000a81548173ffffffffffffffffffffffffffffffffffffffff

ffffffffffff02191690836c0100000000000000000000000090810204021790555061037860016000508190555505b600c80605b6000396000f360606040526008565b600256" 3000000 0.0000005

都可以完成部署同一个智能合约。

- 合约交易：sendtocontract，该命令将发送数据到智能合约，即调用智能合约相关函数。

  sendtocontract "contractaddress" "data" (amount gaslimit gasprice senderaddress)

  contractaddress (必选)： 智能合约地址，大小 20 字节

  data(必选)： 智能合约调用函数及参数

  amount：发送到智能合约的 QTUM 数量，可选择为 0

  gaslimit(可选)： 最大 gas 数量，默认是 2500000

  gasprice(可选)：gas 价格，以 QTUM 为单位，默认为 0.0000004

  senderaddress(可选)：指定发送者的地址，默认为当前账号地址

  broadcast(可选)：是否广播，默认为 true

比如，

qtum-cli sendtocontract "c6ca2697719d00446d4ea51f6fac8fd1e9310214" "54f6127f"

或

qtum-cli sendtocontract "c6ca2697719d00446d4ea51f6fac8fd1e9310214" "54f6127f" 12.0015 6000000 0.0000005

或

qtum-cli sendtocontract "c6ca2697719d00446d4ea51f6fac8fd1e9310214" "54f6127f" 12.0015
6000000 0.0000005 qYcH9wHcd86LaDAAxquKPyfpmSraxAbBop

都可以完成发送交易到智能合约，调用智能合约提供的功能。当需要配置某个参数时，该参数前面的参数也要配置，比如需要配置 senderaddress 时，gaslimit 和 gasprice 也要相应的配置。

3. Qtum 全节点和钱包

Qtum 主网发布后在链接 https://github.com/qtumproject/qtum/releases 上提供已经编译好的软件下载，支持 Windows 、Linux 和 OSX 平台，支持 X86 和 ARM 架构。也可以自己下载源代码进行编译生成。Qtum core 可执行软件 qtumd 和 qtum-qt 都具有全节点功能，同时也具有钱包功能。

节点数据和钱包文件 wallet.dat 默认保存在电脑的特定目录下，不同操作系统对应的目录如下：

- Linux： ~/.qtum
- Mac OSX： ~/Library/Application Support/Qtum
- Windows: %APPDATA%\Qtum

下面介绍在 Linux 上和 Mac OS 上从源代码编译 Qtum Core 的方法。

3.1 Ubuntu16.04 上安装 Qtum

3.1.1 安装依赖

```
sudo apt-get install build-essential libtool autotools-dev automake pkg-config libssl-dev libevent-dev bsdmainutils git cmake libboost-all-dev
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:bitcoin/bitcoin
sudo apt-get update
sudo apt-get install libdb4.8-dev libdb4.8++-dev
sudo apt-get install libqt5gui5 libqt5core5a libqt5dbus5 qttools5-dev qttools5-dev-tools libprotobuf-dev protobuf-compiler
```

3.1.2 拉取 Qtum 源代码

```
git clone --recursive https://github.com/qtumproject/qtum.git
cd qtum
git submodule update --init --recursive
```

3.1.3 编译程序

cd 到 qtum 目录下，执行

```
./autogen.sh
./configure
make install
```

此时 src 文件夹会生成 qtumd、qtum-cli、qtum-tx，和 src/qt 文件夹下的 qtum-qt

3.2 Mac OS 上安装 Qtum

3.2.1 环境准备

以下需要安装的工具或者库，如果已经安装过，可以忽略对应步骤，直接进入下一步。

3.2.2 安装 Homebrew

/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

3.2.3 安装命令行工具

xcode-select --install

3.2.4 安装依赖

brew install cmake automake berkeley-db4 libtool boost --c++11 miniupnpc openssl pkg-config protobuf qt libevent

这一步安装的包比较多，如果电脑里从未安装过以上的包的话，需要耐心的等待一段时间。

3.2.5 拉取 Qtum 代码

首先找个路径来存放 qtum 的代码，没有的话新建个路径，比如代码放在路径 ~/workspace/qtum

```
cd ~/workspace/ qtum
git clone --recursive https://github.com/qtumproject/qtum.git
cd qtum
git submodule update --init –recursive
```

3.2.6 编译代码

```
./autogen.sh
./configure
make install
```

此时 src 文件夹会生成 qtumd、qtum-cli、qtum-tx，和 src/qt 文件夹下的 qtum-qt

3.3 Qtum 钱包

QTUM 的所有权是私钥、地址和签名来确立的。私钥是 QTUM 的所有权的证明，所有权认证基于密码学证明的安全模型。私钥的泄漏也就会导致得到私钥的人可以转移掉私钥控制的 QTUM，因此要保护好自己的 QTUM 私钥。地址是由私钥导出公钥，并对公钥进行哈希和编码后得到的，用于接收 QTUM。签名用私钥对交易进行签名并形成解锁脚本，从

而证明某个 UTXO 的所有权。钱包，是管理生成、存储私钥，并实现接收和发送 QTUM 功能的工具，钱包一般把私钥保存在一个文件或简单的数据库中，因此钱包文件的保存很重要，在没备份私钥的情况下丢失钱包文件也就丢失了该私钥拥有的 QTUM。下面介绍两种 QTUM 钱包的使用。

3.3.1 qtumd 自带钱包

qtumd 自带钱包数据保存在.qtum 目录下的 wallet.dat 文件，所以要妥善保管 wallect.dat 文件，需要经常进行备份否则一旦丢失或损坏里面的 QTUM 将永久丢失。qtum-cli 是和 qtumd 交互的命令行工具，通过 qtum-cli + 命令 + 参数方式使用。

钱包相关的常用命令有：

getaccount "address"　　// 获取地址对应的账户名

getaccountaddress "account"　// 获取某个账户名下关联的一个地址

getaddressesbyaccount "account"　// 获取某个账户名下关联的所有地址

getbalance ("account")　// 获取账户可花费的 QTUM 数量

getnewaddress ("account") //生成指定账户名的一个新地址

dumpprivkey "address"　　//导出私钥

importprivkey　"mykey" // 导入私钥

listaccounts //列出所有在用账户名

sendtoaddress "address" amount //往某个地址发送一定数量的 QTUM

3.3.2 PC 钱包

qtum-qt 钱包支持完整功能的全节点，支持钱包应用。PC 钱包数据保存在.qtum 目录下的 wallet.dat 文件，和 qtumd 共享这个文件，要妥善保管 wallect.dat 文件，经常进行备份。运行 qtum-qt 即可以打开软件。打开钱包时进入到 Overview 界面，如下所示：

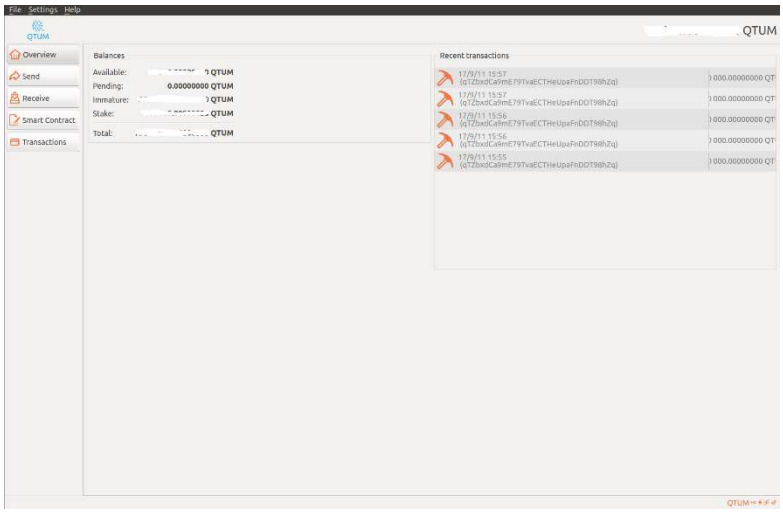

Figure 7 PC 钱包界面

3.3.2.1 往特定的地址发送 Qtum

点击钱包左边的 Send 按钮，进入发送 Qtum 的操作界面，可以往特定的地址发送 Qtum。使用流程如下：

1. 在 Pay to 输入框输入接收 Qtum 的地址
2. Amount 输入框输入要发送的 Qtum 数量
3. 点击 Choose 可以选择交易费用
4. 点击底下的 Send 按钮完成交易

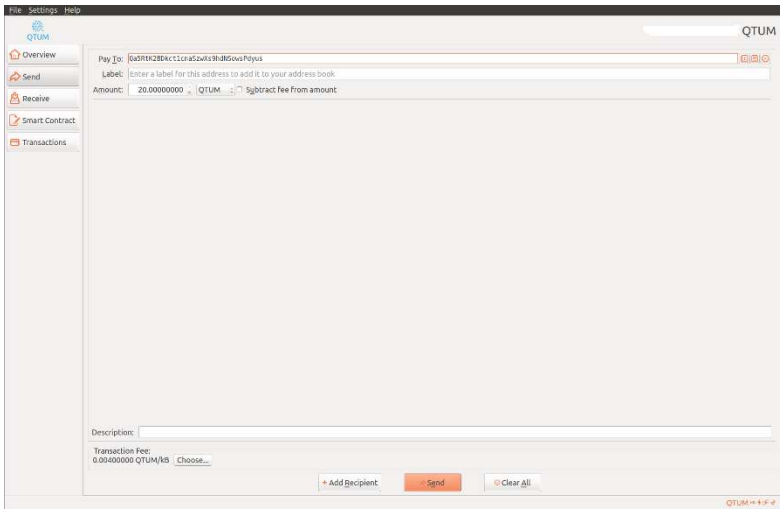

Figure 8 发送 Qtum

Figure 9 交易费用选择

3.3.2.2 接收 Qtum

点击钱包左边的 Receive 按钮，进入接收 Qtum 的操作界面，可以管理用于接收 Qtum 地址。使用流程如下：

1. 点击 Request payment，出现一个对话框，点击 close 即可
2. 点击 Requested payment history 栏目可以显示刚才生成的地址信息
3. 点击 Remove 可以去除生成的地址信息

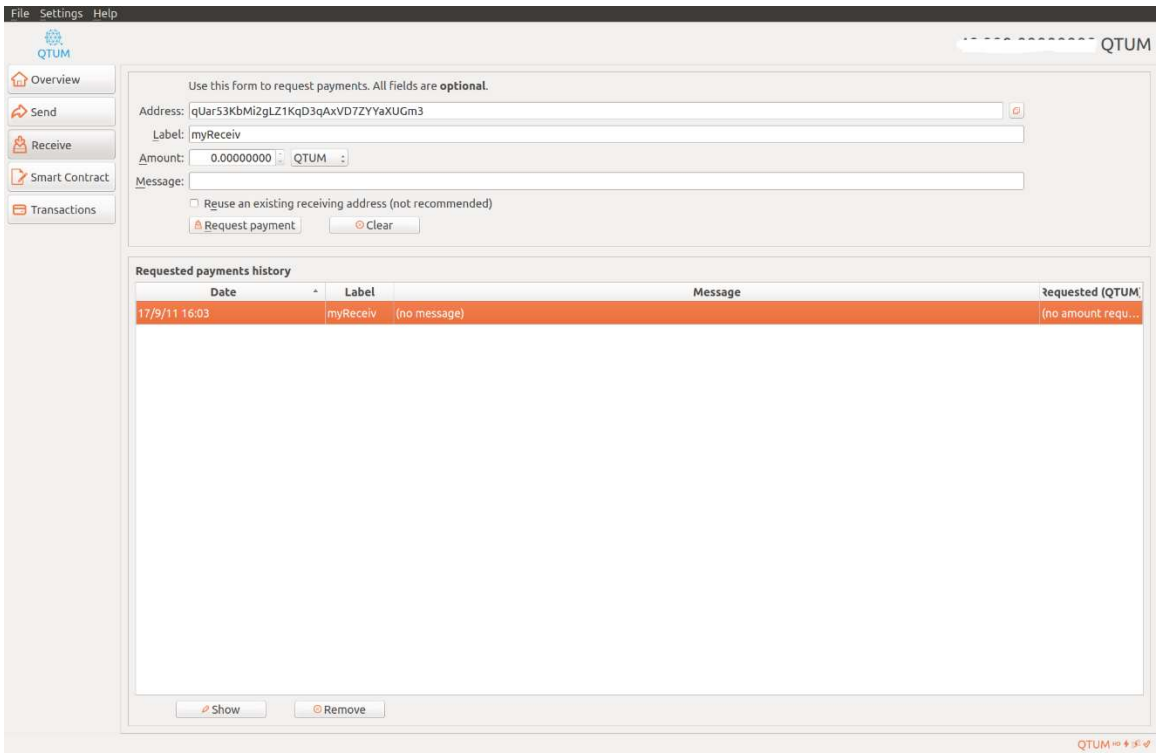

Figure 10 管理接收 Qtum 地址

4. 智能合约

智能合约（Smart Contract），是密码学家 Nick Szabo 在 1994 年首次提出以数字形式定义的一系列承诺（promises），包括合约参与方可以在上面执行这些承诺的协议。智能合约通过区块链来执行，因此一旦设立指定后，能够无需中介的参与自动执行。

Qtum 支持使用 Solidity 语言进行智能合约的编写。Solidity 写好智能合约的智能合约，可以使用 solc 来编译，也可以直接使用基于浏览器的编译器，比如 Remix Solidity IDE。下面通过一个简单的例子来说明开发、部署智能合约的过程。

4.1 智能合约编写和编译

Qtum 支持与 EVM 兼容的智能合约，编写智能合约最常用的语言是 Solidity。编写完智能合约之后可以采用 Remix Solidity IDE 进行编译，该工具一个基于浏览器的编译器，其链接如下：

https://ethereum.github.io/browser-solidity/#version=soljson-v0.4.11+commit.68ef5810.js

这里通过一个简单的智能合约实例，说明合约怎样完成编译，部署以及和合约交互。测试的智能合约代码如下：

```
pragma solidity ^0.4.11;
contract Test {
        uint256 public totalSupply = 0;
        mapping (address => uint256) public balanceOf;
        function Test() { }
        modifier validAmount(uint256 _amount) {
              require(_amount > 0);
                                    _;
         }
       function contribute()  public  payable  validAmount(msg.value)
      returns (uint256 amount) {
          uint256 tokenAmount = msg.value/100000000;
          totalSupply += tokenAmount;
```

```
            balanceOf[msg.sender] += tokenAmount;
            return tokenAmount;
        }
    }
```

把以上代码拷贝到 Remix Solidity IDE 左边的空白处，点击右边 Contract details 就可以看到编译生成的字节码(bytecode)。这是智能合约编译后的可执行二进制代码，下面的部署就是用这个生成的字节码。部署智能合约相当于是安装一个软件到操作系统，因此需要把智能合约部署到量子链后，才能使用该智能合约的服务。

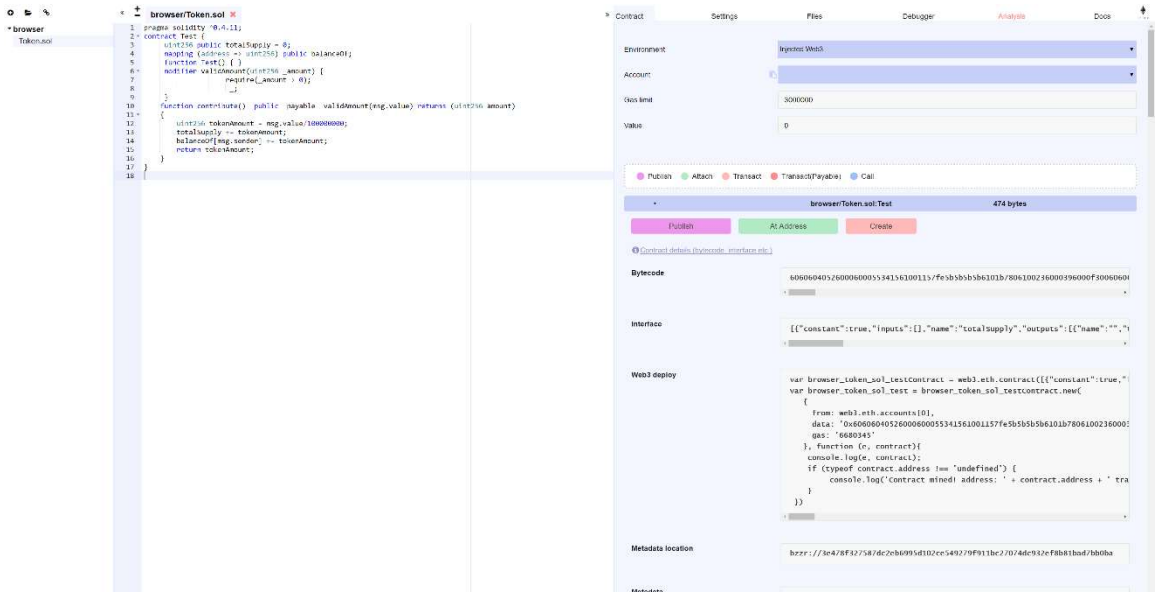

Figure 11 使用 Remix Solidity IDE 编译智能合约

4.2 部署智能合约

部署智能合约需要访问 Qtum 区块链的节点，通过 Qtum 节点提供的 cli 命令行或 RPC 调用可以进行智能合约相关的操作。

首先要运行 Qtum 全节点软件 qtumd，然后使用 qtum-cli 调用 createcontract 可以把智能合约部署到 Qtum 量子链上。

```
qtum-cli  createcontract
60606040526000600055341561001157fe5b5b5b5b6101b78061002360003960000f30060606040
526000357c0100000000000000000000000000000000000000000000000000000000900463ffffff
ff16806318160ddd1461005157806370a082311461007757806d7bb99ba146100c1575bfe5b34
```

1561005957fe5b6100616100df565b6040518082815260200191505060405180910390f35b3415
61007f57fe5b6100ab600480803573ffffffffffffffffffffffffffffffffffffffff16906020019091905050610100
e5565b604051808281526020019150506040518091039f35b6100c96100fd565b604051808281
526020019150506040518091039f35b60005481565b60016020528060005260406000206000 9
15090505481565b600060003460008111151561011257600006000fd5b6305f5e10034811515610
12157fe5b04915081600060008282540192505081905550816001600033 73ffffffffffffffffffffffffffffffffffffffff
ffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152602001600020600082825401
92505081905550819250 5b5b5050905600a165627a7a723058203e478f327587dc2eb6995d102c
e549279f911bc27074dc932ef8b81bad7bb0ba0029 3000000 0.0000005

3000000 0.0000005 是 gaslimit 和 gasprice 参数，参考第 2.3.1 节知道者两个参数可以不设置或可以改为其他的值。命令执行后或产生如下输出：

{

"txid": "52b3eef0fe0032f685974256972b340655501fc87adec4cffe2b96cc46446001",

"sender": "qYcH9wHcd86LaDAAxquKPyfpmSraxAbBop",

"hash160": "a5161fee8ff92457c0829889854888b45255f961",

"address": "2d4b6564a012ae1383854ac48574c2248660d897"

}

这个输出是一个交易，所有合约相关的功能都是通过发送交易触发的，该交易在 Qtum 量子链上部署了地址为 2d4b6564a012ae1383854ac48574c2248660d897 的智能合约。智能合约的需要等待区块对交易确认后，才在 Qtum 链上部署完成。

4.3 和智能合约交互

和智能合约的交互也需要访问 Qtum 区块链的节点，通过 Qtum 节点提供的 cli 命令行或 RPC 调用可以和智能合约进行交互，使用智能合约提供的服务。使用智能合约比较简单，首先是生成需要交互的函数或公共变量的 ABI(Application Binary Interface)数据，之后使用 sendtocontract 发送合约交易，完成和 Qtum 链上智能合约交互。使用 callcontract 可以在本地调用智能合约相关函数或公共变量，但其执行只在本地执行，不会影响到链上的智能合约状态。

4.3.1 ABI 数据生成

ABI 是和智能合约交互的接口，为了使用上一节部署的合约，需要智能合约相关函数和参数生成的数据(ABI 数据)，可以使用 ethabi 工具把函数和参数转换成以上的”data”数据。

首先创建一个空文件 test.json，把 Remix Solidity IDE 右边的 Interface 的内容放到文件 test.json 上。比如上面的合约 ABI 如下：

[{"constant":true,"inputs":[],"name":"totalSupply","outputs":[{"name":"","type":"uint256"}],"payable":false,"type":"function"},{"constant":true,"inputs":[{"name":"","type":"address"}],"name":"balanceOf","outputs":[{"name":"","type":"uint256"}],"payable":false,"type":"function"},{"constant":false,"inputs":[],"name":"contribute","outputs":[{"name":"amount","type":"uint256"}],"payable":true,"type":"function"},{"inputs":[],"payable":false,"type":"constructor"}]

- 生成与合约 contribute()函数交互的数据

contribute()函数没有参数，因此运行如下命令：

```
ethabi encode function ~/test.json contribute
```

得到输出数据：

```
d7bb99ba
```

- 生成与合约 balanceOf(Address)函数交互的数据

balanceOf(Address)需要传递 Address 参数，使用选项-p 指定，Address 是一个 20 字节的 16 进制参数，使用 ethabi 运行：

```
ethabi encode function ./test.json balanceOf -p a5161fee8ff92457c0829889854888b45255f961
```

得到输出：

```
70a08231000000000000000000000000a5161fee8ff92457c0829889854888b45255f961
```

- 生成与合约公共变量 totalSupply 交互的数据

totalSupply 是一个带有 public 属性的变量，因此虽然 totalSupply 不是一个函数，但也可以和它进行交互，同样使用 ethabi 生成交互所需的数据：

```
ethabi encode function ./test.json totalSupply
```

得到输出：

```
18160ddd
```

4.3.2 使用 sendtocontract 和智能合约交互

如果要调用链上的智能合约，把执行结果保存在链上，需要发起一个合约交易，这是通过 sendtocontract 来完成的。使用 sendtocontract 发起一个合约交易的格式如下：

sendtocontract "contractaddress" "data" (amount gaslimit gasprice senderaddress)

其中“data”为上一节产生的 ABI 数据，因此要调用合约的 contribute 函数，采用如下的命令：

sendtocontract 2d4b6564a012ae1383854ac48574c2248660d897 d7bb99ba 2 1000000 0.0000004 qYcH9wHcd86LaDAAxquKPyfpmSraxAbBop

d7bb99ba 即为数据，2 是这笔交易发生到智能合约的 QTUM 数量，1000000 是 gaslimit，0.0000004 是 gasprice，qYcH9wHcd86LaDAAxquKPyfpmSraxAbBop 是 senderaddress。此处为了指定 senderaddress，需要把前面的 amount、gaslimit 、gasprice 参数也指定，执行后命令会返回：

{

"txid": "1e9742000cc5d261f18b3ecc41ffd44b37f965567127b1779fb7d48b12bdfcda",

"sender": "qYcH9wHcd86LaDAAxquKPyfpmSraxAbBop",

"hash160": "a5161fee8ff92457c0829889854888b45255f961"

}

合约交易的执行需要新区块打包和执行，因此查看智能合约的结果需要等待至少 1 个区块的确认。

4.3.3 使用 callcontract 查看智能合约执行结果

虽然使用 callcontract 可以在本地调用智能合约相关函数或公共变量，但其执行只在本地执行，不会保存结果和影响到链上的智能合约状态。因此该功能一般用于查看由 sendtocontract 发起的合约交易引起的智能合约状态改变结果。其格式如下：

callcontract "address" "data" (senderaddress)

比如要查看 a5161fee8ff92457c0829889854888b45255f961 的 balance 使用：

callcontract 2d4b6564a012ae1383854ac48574c2248660d897 70a082310000000000000000000000000a5161fee8ff92457c0829889854888b45255f961

命令返回：

{

  "address": "2d4b6564a012ae1383854ac48574c2248660d897",
  "executionResult": {
    "gasUsed": 23183,
    "excepted": "None",
    "newAddress": "2d4b6564a012ae1383854ac48574c2248660d897",
    "output": "0000000000000000000000000000000000000000000000000000000000000002",
    "codeDeposit": 0,
    "gasRefunded": 0,
    "depositSize": 0,
    "gasForDeposit": 0
  },
  "transactionReceipt": {
    "stateRoot": "cb3e98d8bfcde37e37df441915a3d6f50712a7e368f6e798cc804aa574fd6c27",
    "gasUsed": 23183,
    "bloom": "00",
    "log": [
    ]
  }
}

4.4 在 PC 钱包上使用智能合约

在 PC 钱包(qtum-qt 软件)上可以通过 GUI 实现和 qtum-cli 同样的智能合约的部署、交互等功能。

4.4.1 部署智能合约

部署合约都是要和 Qtum 链上进行交互的，因此连接到一个全节点，由于 PC 钱包本身是一个全节点，因此不需要另外运行 qtumd 之类的节点。在 PC 钱包左边点击 Smart Contract 按钮出现智能合约功能界面，点击 Create 后，把 4.1 节编译的字节码拷贝在 Bytecode 输入框。对应 qtum-cli createcontract 命令参数，界面下面的 GasLimit、GasPrice、和 Sender Address 可以设置。最后点击 Create Contract 按钮，就会生成一个合约交易，把合约部署到 Qtum 链上，弹出的对话框显示了发送者、合约地址等相关信息。

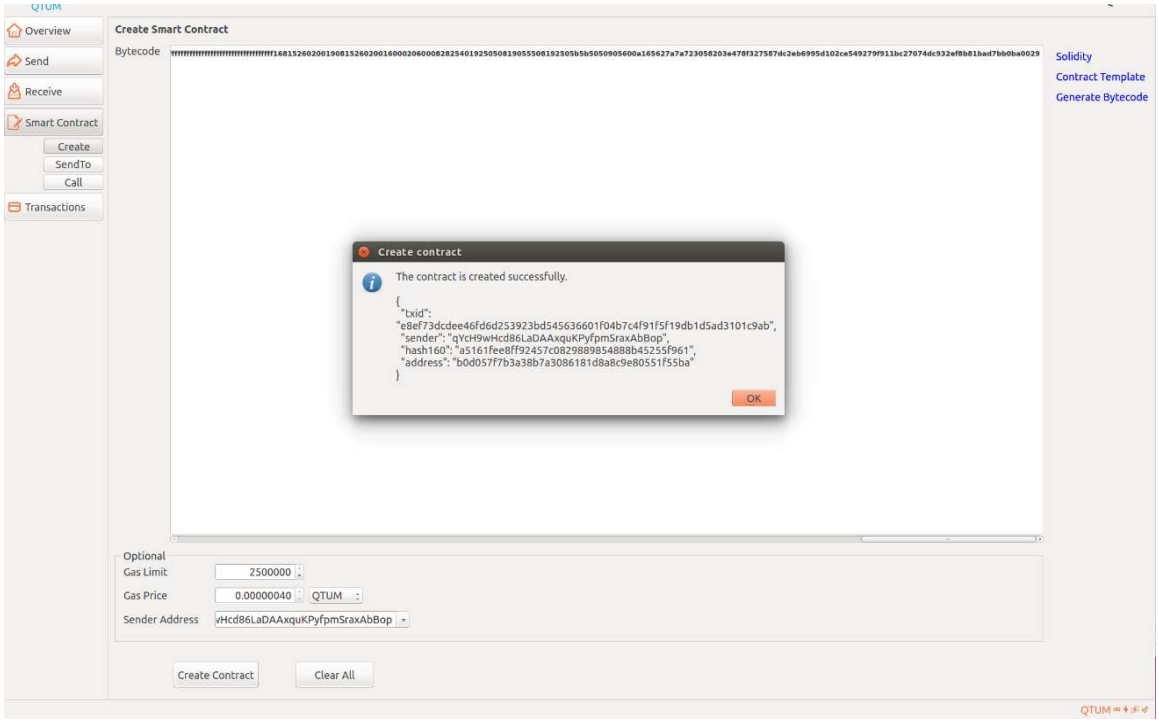

Figure 12 PC 钱包部署智能合约

4.4.2 和智能合约交互

和智能合约的交互，同样要连接 Qtum 节点，通过 PC 钱包就可以了。生成需要交互的函数或公共变量的 ABI 数据可以参考 4.3.1 节。点击 Sendto 切换到合约交易生成界面，把 Contract Address、Data Hex 填好，GasLimit、GasPrice、和 Sender Address 可以根据需要设置。点击 Send To Contract 按钮即生成了合约交易， 完成和 Qtum 链上智能合约交互。

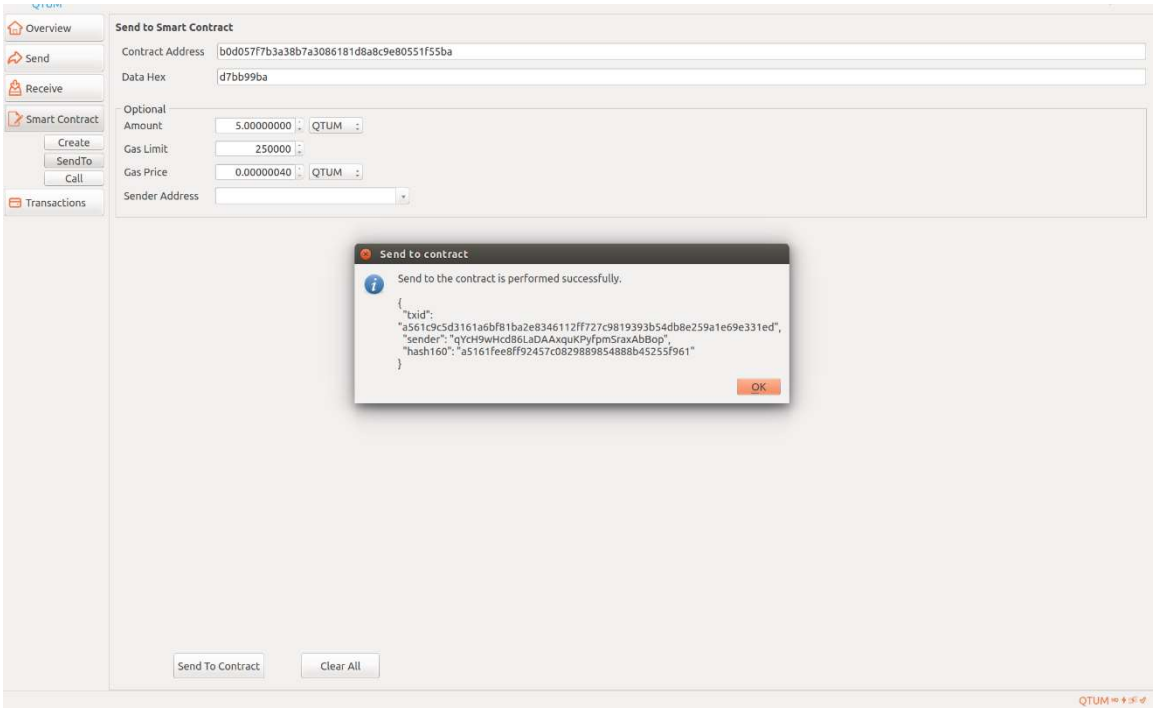

Figure 13 和智能合约交互

4.4.3 查看智能合约执行结果

和 callcontract 一样可以通过 qt 钱包在本地调用智能合约相关函数或公共变量，其执行不会影响到链上的智能合约状态，一般用来查看智能合约在 Qtum 链上执行某个函数之后的结果。点击 Call 按钮切换到本地执行合约界面，把 Contract Address、Data Hex 填好，Sender Address 可选设置，最后点击 Call Contract 完成智能合约相关功能调用，合约执行的结果会在弹出的对话框中显示出来。

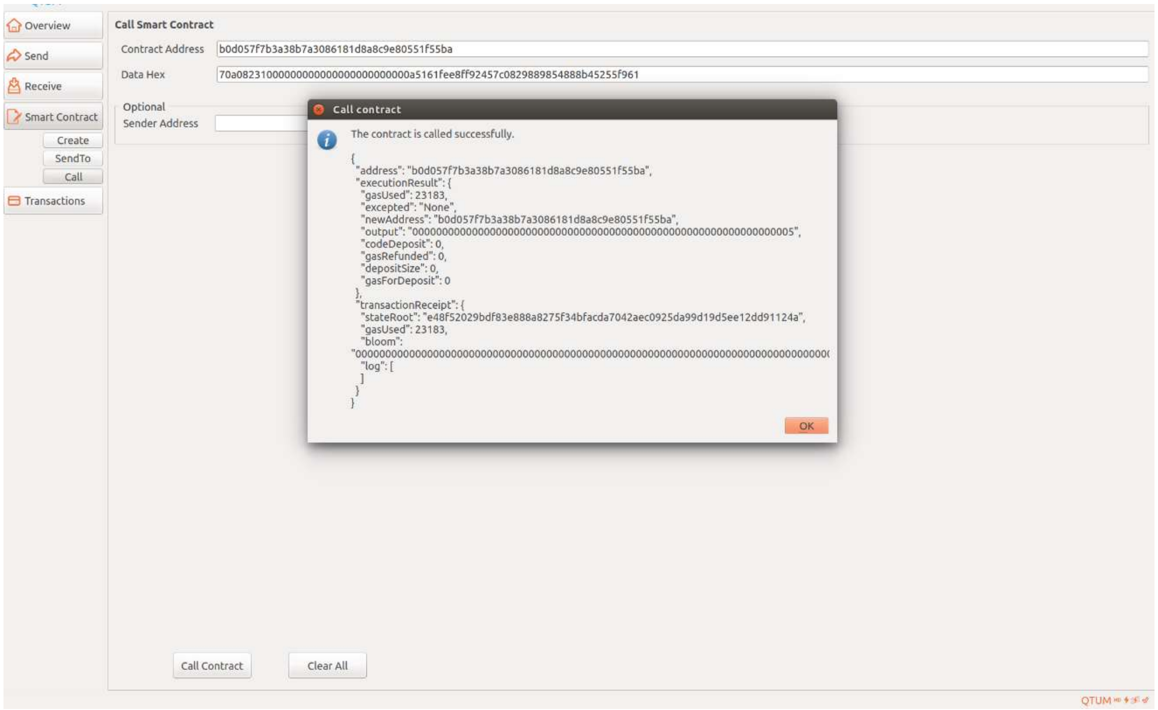

Figure 14 本地调用智能合约