PalletOne 技术黄皮书

V1.0 beta

2018年5月

目录

摘要

PalletOne 旨在建立区块链世界的 IP 协议，通过统一抽象各区块链的接口，提供多语言的智能合约运行环境，将底层链和智能合约完全解耦， 实现跨链的图灵完备的智能合约通用区块链平台，PalletOne 的总体结构如图 1.1 所示。

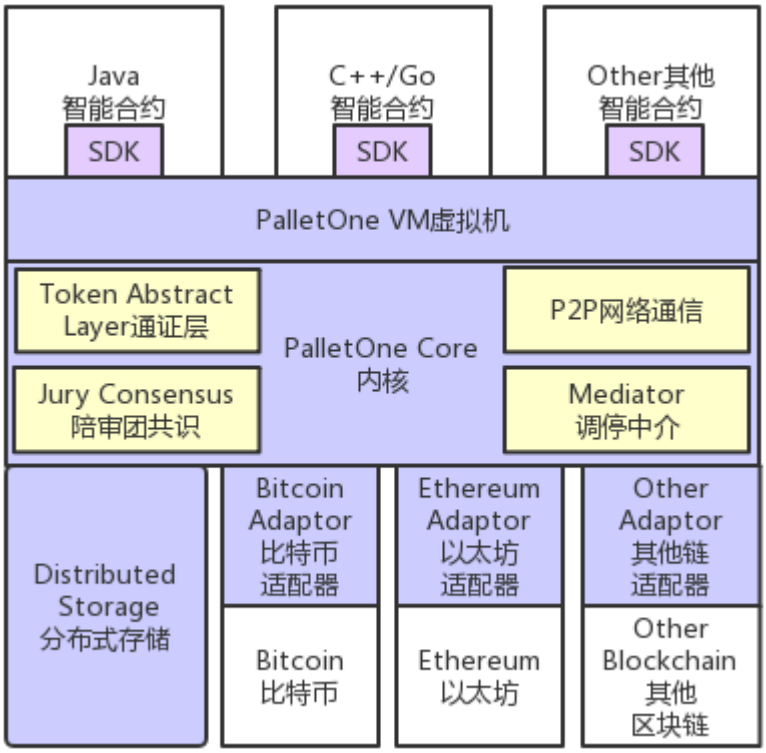

图 1.1 PalletOne 的整体结构图

PalletOne 具有多链、多语言、高并发、通证抽象和完整生态等特点。

1. 多链

PalletOne 作为一个跨链平台，并不绑定具体的区块链，而是抽象出区块链的通用接口，只要针对不同的链建立对应的适配器，即可实现多链的跨链价值交换。

2. 多语言

PalletOne 支持图灵完备的智能合约，但是并不像其他的智能合约区块链一样发明一套独立的编程语言，而是采用世界上最流行的编程语言（比如 Java，C/C++,JS 等）作为智能合约的开发语言，PalletOne 虚拟机为不同的语言提供了一个主机安全的沙盒环境执行合约，降低了智能合约开发人员的学习曲线和开发难度，使得开发人员可以使用自己最熟悉的语言进行智能合约的开发。

3. 高并发

PalletOne 独创的陪审团共识和 DAG 分布式存储方式，有别于传统的区块链的串行生成区块的模式，从计算和数据存储上都实现了并行，从而实现了高并发的支持。

4. 通证抽象

PalletOne 在内核中直接集成了通证抽象层，使得通证的发行、流转变得异常高效和容易，而不必像以太坊那样，需要用户编写自己的 ERC20 或者 ERC721 的智能合约代码。在 PalletOne 上用户只需输入几个参数，无需编写合约就可以发行自己的通证，避免了因合约出现漏洞造成黑客的攻击。

5. 稳健生态

PalletOne 良好的通证经济设计为其稳健的生态的形成奠定了基础。同时 PalletOne 应用商店采用了类似于苹果 AppStore 的模式，让开发者、消费者、矿工都在 PalletOne 平台上受益。

PalletOne 整个网络如图 1.2 所示。

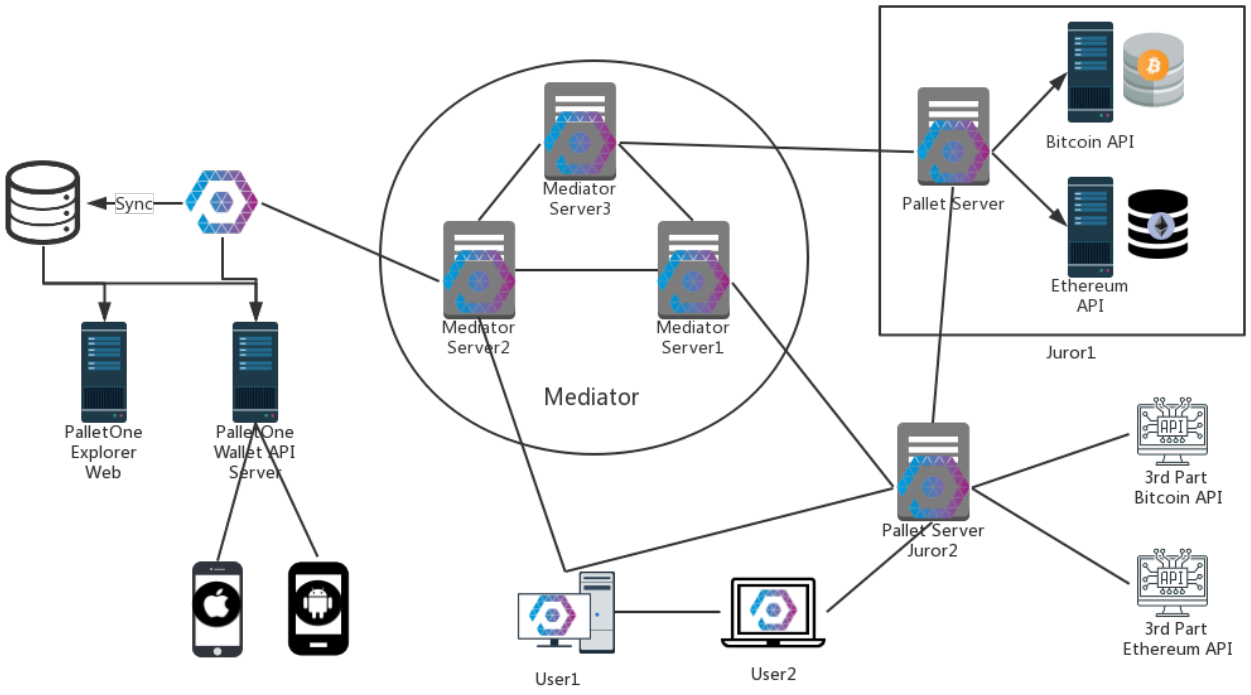

图 1.2 PalletOne 整个网络

图 1.2 所示网络主要分为以下几部分：

1. 矿工节点（Mediator）

Mediator 是整个 PalletOne 网络的核心，由 21 个独立的节点组成。Mediator 通过 DPOS 算法由用户选举产生，负责通证的流转、合约的部署、陪审团的创建，DAG 单元的见证等。Mediator 之间两两连接，实现快速的共识通讯。

2. 矿工节点（Jury）

Jury 是智能合约的执行节点，Jury 的数量没有限制，一个节点可以同时参与多个陪审团，对合约的执行进行共识。Jury 节点由于需要跨链，所以需要对其他的链进行查询和操作，目前有两种模式：

- 陪审员自建其他链的全节点服务器，比如下图中的 Juror1，PalletOne 在需要进行跨链操作时，直接连接自建的 API 服务。这种模式操作的速度更快，而且更安全，但是部署成本高。
- 调用第三方的链操作 API，比如比特币的 blockchain.info。这种模式成本低，但是可能网络延迟大，对外部的依赖容易被攻击。

3. PalletOne 浏览器（Explorer）

PalletOne 浏览器提供了对所有历史账本和当前网络状态的查询，用户通过 PalletOne 浏览器可以查看合约、通证、网络等信息以及各种统计分析和报表。

4. 钱包全节点

类似于比特币的 Bitcoin Core，用户可以建立自己的全节点，将账本数据同步到本地，然后进行查询和交易。

5. 手机钱包

类似于 Imtoken 的钱包，PalletOne 将提供 Android 和 iOS 的手机钱包，用户只需要通过手机 APP 即可完成通证管理和合约管理。另外，为了方便跨链操作，PalletOne 的手机钱包还会支持比特币、以太币等资产的管理。

PalletOne 整体系统如图 1.3 所示。

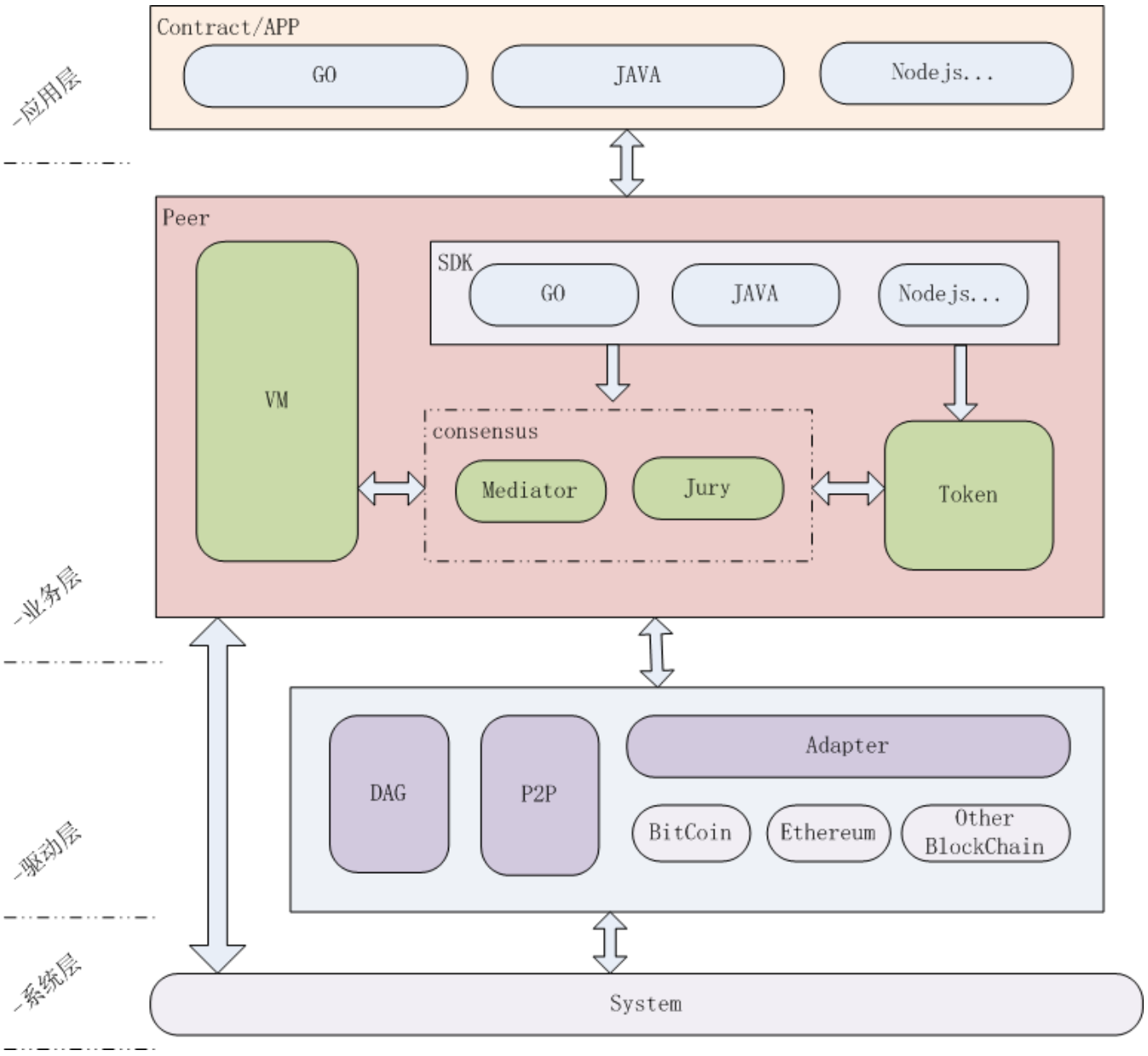

图 1.3 PalletOne 系统框图

从图 1.3 可以看到 PalletOne 系统分为四层：

1. 系统层：由具体系统平台来提供基础服务。

2. 驱动层：主要包括 DAG 存储、P2P 网络及对不同区块链平台功能适配模块组成，用以满足对业务层的支持。

3. 业务层：主要包括 PalletOne Mediator 和 Jury 共识机制模块、支持智能合约虚拟机以及通证经济模块，另外针对不同编程语言的应用的开发，提供了对应的 SDK 模块。

4. 应用层：包括基于对应 SDK 的 APP 及智能合约的上层实现，满足不同具体业务的功能开发。

PalletOne 内核

PalletOne 内核是 PalleOne 的核心构成部分，是 PalletOne 各个模块之间通信的桥梁。PalletOne 内核主要由 5 个部分组成，分别是：调停中介模块、陪审团共识模块、通证抽象层、与其他模块交互接口以及 P2P 网络模块。

PalletOne 内核从用户友好性、易用性方面考虑，封装了许多上层用户使用的场景，比如通证定义的抽象、通用合约的抽象。同时，为了实现用户层与分布式存储层、底层区块链适配层的解耦，PalletOne 内核封装了各个模块的交换接口。陪审团共识模块和调停中介模块一起维护 Pal letOne 的安全和问责机制，保障 PalletOne 的正常运行。

调停中介 Mediator

调停中介（Mediator）负责 PalletOne 网络的整体安全性。Mediator 的角色和传统区块链有些相似，都是信任机器，因此，Mediator 需要保证所有的决定都是正确的。Mediator 使用代理权益证明（Delegated Proof of Stake，DPoS）来选举产生，而在 Mediator 内部， 多个节点之间采用 BFT 共识。为了防止 Mediator 成为 PalletOne 的瓶颈，大部分工作只需要陪审团完成而不需要调用 Mediator。以下是 Mediator 的主要工作：

1. 持有陪审员的保证金；
2. 部署智能合约模板；
3. 随机选择陪审员组建陪审团；
4. 为多签钱包提供签名；
5. 分布式存储中 DAG 单元的见证。

保证金管理

保证金管理

Mediator 及 Jury 候选人池的保证金通过 PalletOne 保证金合约执行运作。保证金合约不同于普通的智能合约，该合约固化在 PalletOne 内核中，由 Mediator 执行，具有保证金交付、保证金退还、保证金没收等功能。

为了避免 PalletOne 的 Mediator 及 Jury 作恶，有意愿成为 Mediator 及 Jury 的用户必须先交付一定数量的 PalletOne Token 作为运作保证金，以示严肃对待担任 Mediator 及 Jury 角色，从而进入 Mediator 及 Jury 的候选人池。

保证金交付

在 PalletOne 中，每一个用户都可以依照自己的意愿交付一定数量的 PalletOne Token 进入 Mediator 及 Jury 的候选人池。

```
//handle witness pay
void deposit_witness_pay(const witness_object& wit, token_type amount)
```

保证金退还

在退出 Mediator 及 Jury 候选人池的时候，由 PalletOne 智能合约模块执行运作将保证金退还当初登记的帐户地址上。

```
//handle cashback rewards
void deposit_cashback(const account_object& acct, token_type amount,
bool require_vesting = true);
```

保证金没收

在担任 Mediator 或 Jury 期间如发现作恶，将会由 PalletOne 智能合约模块执行运作将当初交付的保证金没收。

```
void forfeiture_deposit(const witness_object& wit, token_type amount)
```

在 PalletOne 网络上线时，可由基金会建立全部 Mediator 节点作为启动节点，然后其他用户和组织便可通过缴纳保证金竞选投票的方式替代掉启动节点，由社区来运行正式的节点。

智能合约模板的部署

智能合约的执行是由陪审团来完成，但是在智能合约创建之前，开发人员需要将合约模板部署到 PalletOne 网络中，这个工作由 Mediator 负责合约模板的验证和确认。

关于智能合约模板部署的详细过程，可参考后续章节“智能合约的生命周期”内容。

随机选择陪审员创建陪审团

用户在发起合约创建时，由 Mediator 根据合约模板的参数随机选择陪审员组成陪审团，然后将合约的实例化和合约的执行交给陪审团来完成。大致过程如下：

1) 用户发送创建合约请求消息至 Mediator P2P 网络中；

2) 在某个时间分片内，Mediator 超级节点对消息进行验证；

3) 当前 Mediator 轮值节点根据陪审员随机选择算法，选出陪审员列表，将合约执行结果及签名发送到超级节点的 P2P 网络中；

4) 所有 Mediator 节点对选出的陪审员进行验证，并把数据单元存储至 DAG 中。关于陪审员的随机选择过程，请参考“陪审员随机选择算法”章节。

为多签钱包提供签名

由于 Mediator 需要更多的保证金，具有很高的在线率，而且总体来说比较稳定，所以在跨链多签钱包操作时，可以提供稳定的签名。

以比特币的多签钱包为例，如果生成一个 7/12 的多签钱包，AB 两个用户各持有 1 个密钥，而剩下的 10 个密钥可以由得票最高的 10 个 Mediator 持有。当陪审团做出了需要多签的共识后，Mediator 直接相信陪审团的共识结果，在需要签名的地方提供自己的签名，Mediator 并不需要自己执行合约内容。

DAG 单元的见证

在 PalletOne 中，Mediator 需要同时负责 DAG 单元的见证，产生的见证单元分为交易确认单元、合约创建认证单元、合约调用认证单元。Mediat or 见证 DAG 单元的流程如图 2.1 所示。

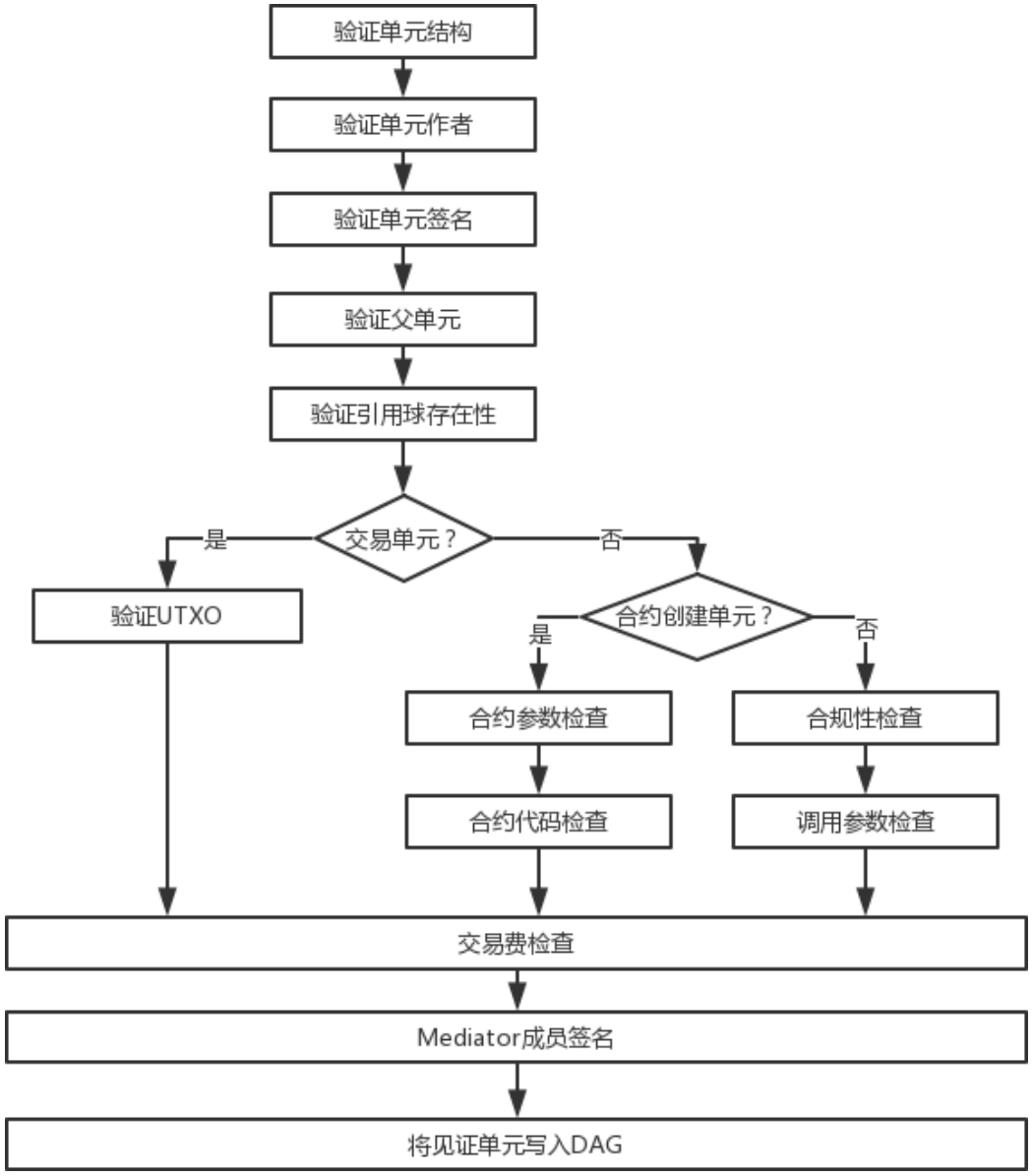

图 2.1 Mediator 见证 DAG 单元的流程

陪审团 Jury

陪审团（Jury）是维护 PalletOne 安全性和完整性的基本单位。更具体的说，陪审团被委任运行智能合约和管理多重签名账户。为了实现安全和去中心化的设计，陪审团被设计为由许多参与者组成，这些参与者被称为陪审员。每位陪审员支付保证金以保证安全。陪审团采用 VRF+BFT 共识的算法来实现共识。关于陪审团的随机生成和内部的共识算法，将会在“Jury 共识算法”章节进行详细介绍。

陪审团锁定与非锁定模式

在陪审团的组建方式上，有两种组建模式：

- 陪审团锁定模式

- 陪审团非锁定模式

下面分别介绍两种模式的实现方法。

陪审团锁定模式

陪审团在合约创建时同时创建，并与合约绑定，陪审团的成员是固定不变的，也就是说，之后该合约的所有调用都是由这些陪审团成员来执行，不允许陪审员的替换。在这种模式下要求大部分陪审员必须长期在线，只允许小部分的陪审员离线。比如在 4 个陪审员组成的陪审团中，只允许 1 个陪审员离线。陪审团锁定模式适用于合约生命周期较短的合约，对于要执行几个月甚至更久的合约，谁也不能保证陪审团能够永远长期在线担任陪审员角色。

陪审团非锁定模式

陪审团非锁定模式是指合约创建时选定了一个大于陪审员实际需要的陪审员池，而接下来的每次合约调用，都是从该合约对应的陪审员池中， 根据一定的随机选择算法，选择出要求数量的陪审团，由这个陪审团执行合约。非锁定模式适用于不需要多签的，长期运行的合约。

以示例”BTC 和 ETH 购买游戏币“合约为例，这个合约不需要多签，而且是长期运行的，所以适合陪审团非锁定模式。用户在创建购买游戏币合约时，假设合约需要 10 个陪审员进行共识，Mediator 会从所有候选陪审员列表中选出 50 个愿意执行该合约的陪审员组成陪审员池，然后从池中随机选出 10 个陪审员，组成该合约本次执行的陪审团，每次合约调用时，Mediator 都会重新从陪审员池中随机选出 10 个陪审员执行合约。

陪审员池的节点越多，一次性选举的陪审员越多，合约执行的共识越慢，但同时也越安全，受到攻击的可能性越小。智能合约开发者可以根据具体的业务场景，在共识的效率和安全之间选择一个平衡点。

通证抽象层

在 PalletOne 中，通证的发行类似于彩色币的原理，发行的通证与 PalletOne Token 在同一个层次，用户可以像操作 PalletOne Token 一样操作其他通证。在分布式数据存储中，我们可以看到，通证数据与合约的状态数据是隔离开的，而不是像以太坊那样将通证数据作为状态数据来进行处理。

通证类型

PalletOne 在初期将内置以下的通证抽象模型：

1. 同质化通证

类似于以太坊中的 ERC20 发行的通证，用户只需要在发行通证时指定通证的总额、精度、通证名称、缩写等信息即可。PalletOne 一次性将通证创建并发行出来，接下来不可增发。

2. 非同质化通证

人们通常说的通证是同质化的，也就是说你拥有的 1 个 Token 和我拥有的 1 个 Token 没有任何区别。然而现实世界中存在着大量非同质化的 Token，比如将艺术品（比如字画）Token 化后，每一个 Token 都代表着独一无二的艺术品，该 Token 不可合并，不可分割。在以太坊中 ERC721 定义了这种非同质化通证。PalletOne 原生支持非同质化通证。非同质化通证不是一次性创建的，而是由指定地址创建，该地址可以是用户地址，也可以是合约地址。如果创建地址是用户地址，意味着只有这个用户才能创建非同质化通证；如果创建地址是合约地址，意味着只有智能合约中的逻辑才能创建该通证。

非同质化通证包含以下信息：

通证名称，通证唯一编号，通证附加属性，创建人地址，创建时间

以上信息在创建后便不可更改。

除了以上提到的两种常用通证模型外，以后我们还会考虑引入更多的通证模型，以适应更多的应用场景，比如：

3. 半同质化通证

半同质化通证意味着通证内存在区别，但是每一种区别开的通证又不一定是唯一的，比如将集邮市场通证化后，每一枚邮票用一个 Token 表示，同一种的邮票又不止一张。同一种邮票是可以合并和分割的，但是不同种邮票之间不能合并。与同质化通证一样，半同质化通证需要指定创建地址，而不是一次性创建完毕。

半同质化通证包含以下信息：

通证名称，通证唯一编号，通证数量，通证附加属性，创建人地址，创建时间

4. 挖矿通证

类似于比特币的经济模型，用户在发行该通证时并不完全预挖或者不预挖，通证会随着时间和出块的高度而慢慢发行。

5. 固定面额通证

类似于现实生活中的纸币，用户可以定义 1,2,5,10,20,50,100 等面额的通证，使用该通证时不可分割。

通证的操作

1. 通证的初始化

PalletOne 为通证的初始化提供了同质化通证和非同质化通证的模板，用户通过模板，填写参数，便可快速的创建自己的通证。用户在通证初始化时需要提供表 2.1 所示的参数。

表 2.1 通证初始化参数

参数	描述
initTotalSupply	通证在初始化时的发行总额
decimals	精度，可分割到小数点后几位
name	通证名称,该通证的全名，500个字符内
symbol	通证简写符号，5个字符内
tokenType	通证类型：同质化/非同质化
issueAddress	哪个地址有权利发行该通证，可以是个人账户地址或者是合约地址，如果为空则不允许再发行
extProperties	扩展属性，用户自定义的一些属性，JSON格式

2. 通证的发行

在定义通证时，用户可以指定通证的发行地址，该地址可以是一个个人钱包地址，一个多签钱包地址，或者一个合约地址。一旦指定地址后， 只有该地址才能发行该通证。如果是全预挖的同质化通证，不允许在通证创建后继续发行，则没有发行地址。发行完成后的所有通证都在发行地址名下。

```
function issueToken(string _uname,json _properties,uint256 _value)
returns (bool success);
```

发行通证时如果是非同质化通证，可以指定新发行的通证的唯一名称，JSON 格式的属性，数量是 1。而对于同质化通证，一般是初始化后就不再发行；如果允许发行，不需要指定唯一名称和 JSON 属性，只需要指定发行数量即可。

3. 通证的转移

PalletOne 中对通证的操作采用了与比特币类似的操作模式，在底层使用 UTXO 模型记录用户余额，支持多个输入和多个输出，在操作上支持类似比特币的 P2PKH 和 P2SH，也就是单签转账和多签转账。

```
function transfer(UTXO[] _from, (address _to, uint256 _value)[]) returns
(bool success);
```

授权指定地址特定的转账额度。被授权的地址可以多次调用 transfer 函数代替源地址转账，总值不超过 _value。

4. 通证的授权

PalletOne 中的通证授权类似于 ERC20/721 中的 approve 方法，也就是允许一个地址从授权方提取指定数额的通证。

```
function approve(address _spender, uint256 _value) returns (bool
success);
```

由于 PalletOne 采用了 UTXO 模型，所以 approve 的本质是将一定数量的通证转移到 1/2 多签的地址，授权人和被授权人都可以使用该多签地址的通证。

5. 通证的销毁

PalletOne 中定义了一个特殊的地址，被称为销毁地址，如果用户要销毁某通证，只需要把该通证转移到销毁地址即可。销毁后的通证不可恢复，该通证的发行总量也相应减少。

```
function destroy(uint256 _value) returns (bool success);
```

6. 通证发行权转移

用户拥有一个通证的发行权后，可以将发行权转移给其他用户或者合约，转移后该用户就失去发行权。

```
function transferIssue(string _address)  returns (bool success);
```

与各模块的交互

与智能合约虚拟机的交互

在 PalletOne 内核中，我们定义了以下虚拟机操作的接口，虚拟机层只需要实现该接口即可，对内核来说无需关心具体使用哪种方式实现的虚拟机，从而实现了与 Docker 的松耦合，方便以后可能支持更多的虚拟机实现。

1. 部署虚拟机

```
string Deploy(string templateId, string config,byte[] program)
```

2. 启动虚拟机

```
ContractResult Start(string vmId, string instanceId, string[]
parameters)

struct ContractResult
{
 TokenPayment[] TokenPaymentSet,
 KeyValue[] ReadSet,
 KeyValue[] WriteSet,
 ChainTrans[] InterchainSet
}
```

- TokenPaymentSet 与 PalletOne 上通证相关的转移操作。ReadSet 该合约读取了哪些数据。
- WriteSet 该合约写了哪些数据。
- InterchainSet 如果是跨链的合约，那么对其他链发起了什么交易。

3. 调用虚拟机中的函数

```
ContractResult Invoke(string vmInstanceId,string function,string[]
parameters)
```

4. 停止虚拟机

```
Stop(string instanceId)
```

5. 销毁虚拟机

```
Destroy(string vmId)
```

内核为虚拟机运行提供了跨链访问接口，这些接口的调用最终会传递到适配器，由适配器执行。

与分布式存储的交互

在 PalletOne 内核中，我们定义了对分布式存储操作的抽象接口，内核只通过接口操作分布式存储中的数据，而不关心具体的实现。DAG 是 Pall etOne 默认的高性能分布式存储方案，未来可能支持更多的存放方式比

如 IPFS 或者其他分布式数据库。在 PalletOne 内核看来，分布式存储是一个读写交易的数据库，支持常用的交易数据操作和合约状态以及通证 UTXO 的查询。

1. 建立分布式存储的连接

```
OpenConnection()
```

2. 根据交易 Id 读取存储的数据

```
Transaction GetTxData(string txId)
```

3. 查询合约状态数据

```
KeyValue[] QueryStateData(string contractId)
```

4. 查询合约历史数据

```
StateHistory[] QueryStateHistory(string contractId)
```

5. 查询通证 UTXO 数据

```
Utxo[] QueryUtxoData(string address)
```

6. 查询通证地址历史交易记录

```
Transaction[] QueryAddressHistory(string address)
```

7. 写入交易数据

```
UnitId WriteTxData(string txId,Transaction trans)
```

```
struct Transaction
{
 string TransactionId
 Datetime CreateTime
 TokenPayment[] TokenPaymentSet
 KeyValue[] ReadSet
 KeyValue[] WriteSet
 ChainTrans[] InterchainSet
 PubkeySign[] AuthorSign
}
```

```
bool CheckTxConfirm(string txId)
```

8. 检查交易数据是否已经被确认

该函数用于确认某一笔交易是否已经被确认，被确认意味着不再会被更改。

9. 断开与分布式存储的连接

```
CloseConnection()
```

与区块链适配器的交互

在 PalletOne 内核中，我们针对其它区块链的特性，设计了 3 类操作接口，通用数字货币适配器接口、基于 UTXO 模型的适配器接口和智能合约适配器接口。智能合约通过 SDK 提供的这些接口，实现了跨链的操作。

通用数字货币适配器接口

1. 钱包

- 生成公私钥对
- 生成钱包地址
- 根据私钥生成对应的公钥
- 验证钱包地址的合法性

2. 查询

- 根据钱包地址查询账户余额
- 根据钱包地址查询交易历史

- 根据交易 ID 查询交易对象
- 根据交易 ID 查询交易确认状态

3. 交易

- 生成一个交易
- 签名一个交易
- 发送一个交易到网络

UTXO 适配器接口

数字货币适配器在通用抽象适配器接口的基础上，提出以下 UTXO 专用的接口函数：

- 根据钱包地址查询其拥有的 UTXO
- 发起一个花费指定 UTXO 的交易

智能合约适配器接口

智能合约适配器在通用抽象适配器接口的基础上，提出了合约相关的接口函数：

- 部署合约
- 初始化合约
- 调用合约
- 销毁合约

P2P 网络

在 PalletOne 中存在矿工节点和用户节点两种类型，矿工节点指缴纳保证金后，加入了候选陪审员列表的节点，这种节点又分为 Mediator 节点和 Jury 节点。矿工节点都是全节点，保留了全部的账本数据。矿工节点需要运行合约，进行共识等操作。矿工节点必须保证长时间在线和良好的运算环境（CPU、内存、硬盘等）以及网络环境（高带宽、低延迟）。用户节点是浏览和发起交易的节点，用户节点不需要保证是全账本，允许更长时间的延迟。PalletOne 中矿工节点和用户节点的 P2P 网络结构如图 2.2 所示。

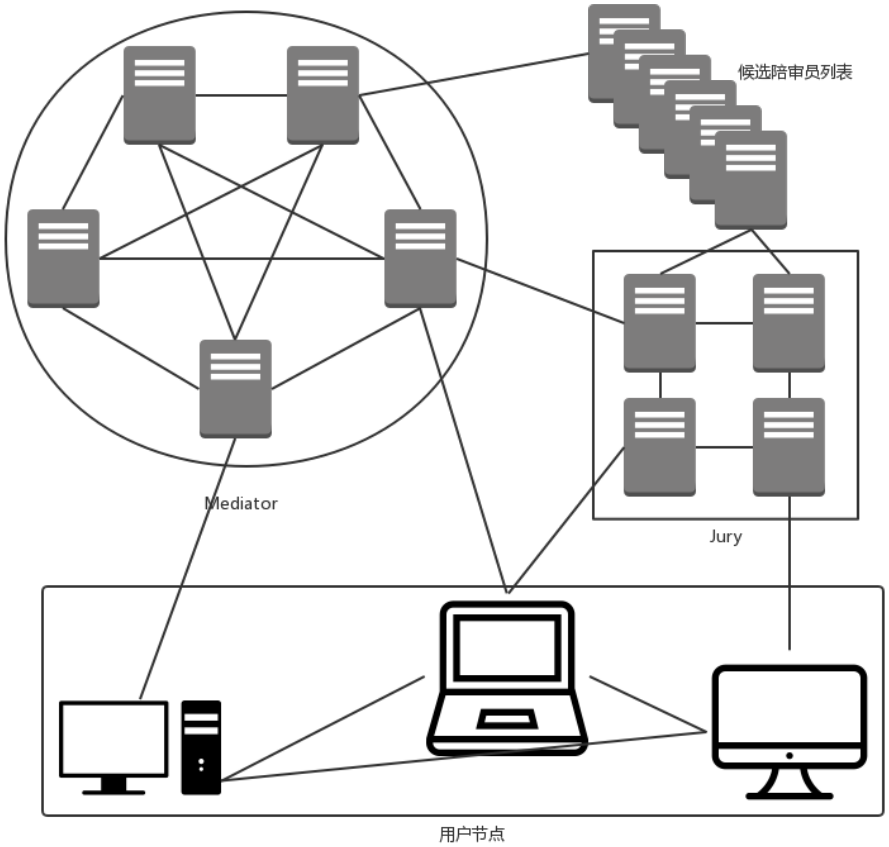

图 2.2 PalletOne 矿工节点和用户节点的 P2P 网络结构

我们假定矿工节点长时间在线，而用户节点会随时上线和离线，但是不会出现大量用户节点同时上线和同时离线的情况发生。

每个节点存储的数据，在理想状态下是一致的，矿工节点要求较低的延迟，用户节点不要求实时性，但是在足够长的时间后也能同步到所有的数据。

PalletOne 中的 Mediator 节点，会相互之间建立 P2P 连接，保证了 Mediator 之间数据延迟更低。Jury 只是临时组件的团体，所以并不要求 Jury 内部每个陪审员之间两两连接。

PalletOne 网络中的消息类型

在 PalletOne 中存在三种类型的网络消息：

1. Request 也就是用户发起的各种请求。如果是用户节点收到，则在先验证 Request 的有效性，有效则转发。如果是矿工节点，则在验证数据有效性后检查该请求是否与自己相关，如果相关则处理请求，不相关则直接转发。
2. Unit Data 也就是 DAG 中的 Unit 数据，各个节点在收到 Unit 数据后会验证 Unit 的有效性，有效则写入本地，并广播给其他节点。
3. Network Data 包括注册、注销、心跳等网络检测相关数据。

共识算法

Mediator 共识机制

DPoS 共识算法

DPoS 算法使用见证人（witness）机制解决中心化问题。总共有 N 个（PalletOne 中默认是 21 个）见证人（PalletOne 中也叫做 Mediator 超级节点）轮流生产区块（PalletOne 中 Mediator 的功能请参考“调停中介 Mediator”章节），而这些见证人由整个区块链网络的主体投票产生。

DPoS 算法的优势

由于使用了去中心化的投票机制，DPoS 相比其他的系统更加民主化。DPoS 并没有完全去除对于信任的要求，而是要确保那些代表整个网络的受信任的签名区块的证人节点要正确无误且没有偏见。另外，每个被签名的块必须有一个先前区块被可信任节点签名的证明。DPoS 中交易不需要等待一定数量的非可信节点验证之后才能确认。

通过减少对确认数量的要求，DPoS 算法大大提高了交易的速度。通过信任少量的诚信节点，DPoS 剔除区块签名过程中不必要的步骤。DPoS 的区块可以比 PoW 或者 PoS 容纳更多的交易数量，从而使加密数字货币的交易速度接近像 Visa 和 Mastercard 这样的中心化清算系统。

DPoS 系统仍然存在中心化，但是这种中心化是受到控制的，因为每个客户端都有能力决定信任哪些节点。基于 DPoS 的区块链网络保留了一些中心化系统的大量优势，同时又能保证一定程度的去中心化。DPoS 通过公平选举使得任何节点都有可能成为大多数节点委托的代表。

DPoS 背后的理性逻辑

- 股东有权通过投票决定记账人
- 最大化股东获得的分红
- 最大限度地减少保证网络安全所支付的金额
- 最大限度地提高网络的性能
- 最大限度地降低运行网络的成本（带宽，CPU 等）

权益所有者拥有控制权

DPoS 的根本特性是权益所有者保留了控制权，从而使系统去中心化。就像投票机制也有缺陷一样，DPoS 是管理公司共有产权的唯一可行方式。幸运的是，如果你不喜欢运营公司的人，你可以通过卖出权益离场。而这种反馈机制可以使权益所有者在投票时比普通公民更加理性。权益所有者通过投票决定区块的签名验证者，得票总数最高的前 N 个人都可以参与到董事会。所有的代表构成一个“董事会”，轮流签署区块。如果一个董事错过了签署区块的机会，客户会自动把投票给予其他人。最终，这些错过签署机会的董事会被取消资格，其他人就可以加入董事会。董事会成员会收到少量代币作为奖励，用来激励在线时间和参与竞选。每一个董事必须要将单个区块平均奖励的 100 倍作为保证金，从而确保其至少 99%的在线时间。

扩展性

假设每笔交易的确认成本和手续费都是固定的，那么实现去中心化的数量也是有限制的。假设验证成本与手续费相等，则整个网络是完全中心化的，并且只能支持一个验证节点。假设手续费是验证成本的 100 倍，则网络可以支持 100 个验证节点。

见证人的作用

1. 见证人被授予生产和广播区块的权利。
2. 生成见证单元的过程包括收集 P2P 网络中的交易并使用见证人的私钥进行签名。
3. 在前一个见证单元（维护周期 maintenance interval）的结尾随机分配见证人在下一轮（round）的轮次。

DPoS 对于攻击的抑制

1. 如果某个见证人不生产见证单元，那么他可能被解职并在未来失去可预期的稳定收入。
2. 不诚实的见证人只有在明确有其他利益诉求时才会选择放弃见证单元生成。
3. 见证人无法签署无效的交易，因为交易需要所有见证人都确认。

Mediator 超级节点

要参与 PalletOne 共识，首先要支付保证金成为超级节点候选人。

Mediator 超级节点的选择

PalletOne Token 持有者可以从候选陪审员中选出任意数量的 Mediator 见证人节点。对于每个 PalletOne Token 账户，每个 Token 可以投一票给某见证人节点或多个见证节点，这个过程被称为 approval voting。

投票数最多的前 N 个（默认 21 个）见证人节点即为 Mediator 的超级节点。未成功成为超级节点的账户可以退还保证金，退出超级节点候选人列表，也可以保持保证金，一边参与陪审团的合约执行，一边等待随时替补其他超级节点落选或退出。

交易费

在合约部署时，超级节点验证或执行合约，并产生出数据单元，会支付给超级节点一定的交易费，交易费比例由 PalletOne Token 持有人投票来制定。如果一个见证人节点未能产生见证单元，将不会收到交易费，并可能后续会被 PalletOne Token 持有者投票出局。

超级节点的更新

超级节点的在线情况记录每天会进行更新，同时，投票也会每天进行清点，超级节点将进行更新。超级节点轮流“记账”，每个节点在 N 秒（默认为 5 秒，可社区投票修改）的时间分片内对未见证的单元进行验证并产生见证单元。当所有超级节点完成一次轮询后，将进行一次节点更新。如果一个超级节点在某个时间分片内未产生数据单元，该时间分片将被自动跳过，并由下一个节点来“记账”。

对于超级节点在线率低于某个水平时

任何人都可以通过查看超级节点的在线参与率，监控全网的健康状况。PalletOne 对于超级节点参与率的标准要求是 99%。在任何的时间点，超级节点的参与率如果低于某个水平，PalletOne 的用户可以要求该节点退出超级节点列表。

BFT 共识算法

在传统 DPoS 共识机制中，我们让每个见证人在出块时向全网广播这个区块，但即使其他见证人收到了目前的新区块，也无法对新区块进行确认，需要等待轮到自己出块时，才能通过生产区块来确认之前的区块。所以一笔交易需要等待很长的时间才能被确认。

为了改进传统 DPoS 算法确认时间长的确定，我们在 Mediator 引入了 BFT 共识机制。在新的机制下，每个见证人出块时先在 Mediator 内部广播，其他见证人收到新区块后，立即对此区块进行验证，并将验证签名完成的区块立即返回出块见证人，不需等待其他见证人自己出块时再确认。从当前的出块见证人看来，他生产了一个区块，并内部广播，然后陆续收到了其他见证人对此区块的确认，在收到 2/3 见证人确认的瞬间再进行全网广播，见证单元以及其中的交易单元就不可逆了。交易确认时间大大缩短。这种机制在 EOS 中也被采用。

Jury 共识算法

陪审员随机选择算法

可验证随机函数 verifiable random function 简称 VRF。

为什么 PalletOne Jury 依赖于随机性?

在区块链中，只有通过使用密码学生成随机数的方法才能在一个抗攻击的网络中管理大量的客户端（矿工），从而实现一个虚拟的超级计算机。当然，中本聪也是设计让矿工去计算随机难题的方法产生随机性。PalletOne Jury 需要一个更强的、更不易被操纵，而且更高效的随机数生成算法。只有这样才能保证陪审团的随机不可预测性，从而降低黑客攻击与矿工作恶的可能。

VRF 的意义很好理解，用以完成 PalletOne Jury 的随机选择。为此，VRF 的返回值应尽力难以预测。PalletOne 是先将前一个随机数（最初的随机数是协议给定的）和某种变量（比如交易的哈希值）进行组合，用某种私钥对之进行签名（或者先签名再组合），最后哈希一下得出最新的随机数。这样产生的随机数其他人很容易验证是否满足算法，可验证性（Verifiable）就得到了保证；而哈希返回值又是随机分布的，随机性（Random）也因此得到保证。

在此过程中，为降低操纵结果的可能性，有两个注意事项：

1. 签名算法应当具有唯一性，也就是用同一把私钥对同样的信息进行签名，只有一个合法签名可以通过验证，普通的非对称加解密算法一般不具备这个属性，如 SM2。如果用的签名算法没有这种 uniqueness 属性，那在生成新随机数的时候就存在通过反复多次尝试签名以挑出最有利者的余地，会降低安全性。

2. 避免在生成新随机数时将当前块的数据作为随机性来源之一，比如引用本块交易列表的 Merkle root 值等，因为这样做会给出块人尝试变更打包交易顺序、尝试打包不同交易以产生最有利的新随机数的余地。

在设计和检视新的共识算法时，以上两个注意事项是要特别留意的。我们采用的算法称为 PalletOne 陪审团共识，这个算法基于密码学，可以在一个参与者足够多的网络中生成几乎不可攻破、完全不可操纵、不可预测的随机数。基于 PalletOne 陪审团共识，PalletOne Jury 网络中的参与者可以生成一个可验证的随机函数 VRF，基于 VRF 随机函数 PalletOne Jury 网络完成对自己组织和运行管理。

密码学相关概念介绍

1. 伪随机数生成器 PRG，可以将种子数 ξ 变为序列值 PRG（ξ，i）i=0,1,2...。

2. 伪随机数排序：PRG（ξ，i）作为高纳德置乱算法的输入，产生一个随机的排序 PermU（ξ）。

阈值签名

阈值签名是一种范式，密钥可以在不同的服务器上被分为 N 份，可以增加系统的可用性和抗攻击性。在（t,n）阈值加密系统中，私钥的操作需要 N 个服务器中的 t+1 个进行配合。（这个机制可以被看做是 Shamir 私钥共享的扩展）。这个函数（签名确认密钥）的公钥部分相比于普通的格式，并没有发生改变。

阈值签名原生用于分布式协议中。阈值同态加密方案被用在投票系统中和多方计算协议中。阈值签名加强了高度敏感的私钥安全，就像认证主体一样。他们也可以作为分布式存储系统的工具。

Threshold 签名方案 ——BLS 签名

BLS 签名是由 Boneh，Lynn 和 Shacham 在 2003 年提出。

● BLS 函数

假设我们已经产生了一个私钥/公钥对（sk，pk），BLS 提供了下列的功能：

1） sign（m，sk），用私钥 sk 对消息进行签名，并且返回签名 σ。

2） Verify（m，pk，σ），利用公钥对信息的签名进行验证，返回真或者假。

● Threshold BLS

Threshold 版本的 BLS 指代 TBLS. TBLS 提供了另外的一个函数：

Recover 函数，从签名分片中恢复出组签名 σ。

PalleOne 中 VRF 的实现

在 PalletOne 中，VRF 的实现采用了阈值签名（t-of -n），在该阈值签名方案使用一个 DKG(分布式密钥生成器, Decentralize Key Generator)（不需要依赖可信第三方），可以使得若干个参与方协同产生密钥（例如组内公钥和各自的私钥分片）。DKG 协议不单单是一个私钥分享协议。在私钥分享协议中，私钥分片可以被用来恢复组内私钥，但是仅仅能用一次，当每个参与方都得到组内私钥后，私钥分片就不能再使用。但是 DKG 中，私钥分片可以无限次使用，不需要对组内私钥进行恢复的操作。

在阈值签名方案中，n 个参与节点组合起来建立一个公钥（组内公钥），每个参与方保留一个私钥分片（组内私钥的分片）。上述步骤完成后， 只需要 n 个参与方节点中的 t 个就可以创建一个签名，并且通过组内公钥就可以对签名进行验证。

PalleOne 中阈值签名方案使具备两个特征：

（1） 非交互性

一个阈值签名方案如果在组签名的创建过程中对于每个参与方都只进行了一次单向通信，那么该阈值签名方案就叫做非交互性的。在非交互性方案中，每一个参与方用自己的私钥创建了签名的一部分，并且将该部分签名发送给第三方，一旦第三方收集到 t 个合法的签名部分，就可以恢复出组签名。

（2） 唯一性

签名方案的唯一性，是指对于每一个信息和每一个公钥，只有一个签名是合法的。但在阈值签名方案中，就有一个额外的要求，就是无论哪个参与方签名，最终产生的组签名必须是相同的。

PalletOne 中陪审团共识包括两部分：

（1） 只执行一次的建立过程（利用 DKG）

（2） 无限次重复的签名，并产生输出。

建立过程：组内运行 DKG 并建立组内公钥以及密钥分片，该步骤是在区块链系统初始化过程时完成的。DKG 完成后，产生一个公共验证矢量和组内每个节点都持有的私钥分片。公共验证矢量可以用来恢复每个私钥分片对应的公钥，每个私钥所对应的公钥分片对各个节点进行公布。各个节点产生的签名分片，组内公钥可以对 Recover 函数的结果进行验证。

签名过程：第 r 轮时每个节点产生一个签名 σ=Sign（r||ξr-1，sk），其中 ξr-1 是 r-1 轮产生的随机值，并把该签名进行广播。其他任何的节点接收到该消息后，可以依据公钥对该签名进行验证， 如果验证是正确的，将对该签名进行存储和广播。如果一个节点接收到 t 个签名，就可以运行 Recover 函数计算出组签名，那么第 r 轮的随机值 ξr 就可以通过对该组签名进行哈希得出。

PalletOne 中分组的初始化与更新

1. 分组初始化

系统初始化时，Mediator 负责对全网节点进行随机分组：

1) 假设分组的数量为 n 个，陪审团就根据随机种子值 ξ 来产生：

```
Group（ξ，j）：= PermU （PRG（ξ，i））（{1.....n}）
```

2) 在系统最开始时，我们将该值设为 m，则有：

```
Gj：=Group（ξ，j）j=1...m
```

其中 ξ 可设定为“PalletOne”的哈希值。

3) 分组产生后，将分组信息广播给全网节点进行缓存，其中每个分组中各个节点的标识为 Nodeid 为节点名称通过公钥签名并进行哈希。

2. 分组的更新

每隔 t 秒（假设 t=3600），Mediator 将对全网节点进行重新分组。

```
Gj=Group（ξ，j）j=1...m
```

其中 ξ 可设定为 Mediator 当前“值班”的超级节点最后一次广播的数据单元签名的哈希值。

3. Jury 的随机选择

当 Mediator 第 r 次接收到某个合约请求时，首先将合约请求消息进行广播，并将为该合约随机选择陪审员组成陪审团，陪审团的选取方法如下：

1) 假设选取的陪审团为 Gr，我们设定：

```
Gr=Gjj=r mod m
```

其中 ξr 为该合约 ID 的哈希值。

2) 将该信息（信息中包含合约 ID、合约 ID 哈希值、以及选中的陪审团）放在数据单元中进行广播。

4. 陪审员节点的发布方法

全网节点接收到数据单元后，可以通过私钥验证自己是否被选中为陪审员节点。同时所有节点也可以通过合约 ID 哈希值并结合缓存中的陪审团预分组来验证选择的陪审团是否正确。

5. 节点排名

基于 ξr，PalletOne 可以对陪审团内的每个节点进行排名；这个排名可以定义为 PermU（ξr），并由 Rank 值最高的节点对陪审团达成共识的数据单元进行发布。

6. 阈值接力

从可扩展性考虑，上述随机数的产生必须是在固定节点数量（21 个 Mediator 节点）的组内产生，而非在全网内产生。否则的话，信息复杂度就会随着节点数量的增长而无限增长。

对于非锁定陪审团模式，就是全网节点中随机选取的 n 个节点。这种随机选取节点组成陪审团，建立陪审团后进行 threshold 操作，选出当前的陪审团，并由当前的陪审团 relay 到下一个陪审团的方法就叫 threshold relay 阈值接力。

陪审团内共识

在通过 VRF 算法随机选出陪审员组成陪审团后，陪审团内部会执行以下步骤完成对合约执行的共识：

1. 所有陪审员节点根据合约调用请求各自执行合约，并将合约结果哈希签名打包广播到网络。
2. 根据一定算法规则在当前陪审团中选定出一个陪审员作为 Leader。
3. Leader 负责收集所有陪审员节点执行的结果哈希，如果满足合约要求的阈值，则直接将结果哈希打包成合约调用单元写入 DAG 数据库， 并广播到网络。

4. 如果选举出的 Leader 在规定时间内未打包广播，则重新选举新的 Leader 进行打包广播。

合约的执行

在陪审团组建后，陪审团就可以根据合约调用请求启动 PalletOne 虚拟机，运行智能合约。智能合约的内部不应该存在随机数、时间变量等， 那么无论在哪个陪审员节点上执行，对于相同的输入，必然产生相同的输出。陪审员将合约执行的输出结果，连同输入参数计算哈希，再用自己的私钥对哈希进行签名，然后将合约输入输出哈希、签名、同执行时间和其他相关信息发送到 P2P 网络。

陪审团 Leader 的选定

当每个陪审员收到了超过 2/3 的节点数的相同执行结果哈希和各自的签名后，会根据自己的签名哈希是否所有哈希中最小的从判断而自己是否作为本次执行合约的 Leader。

合约调用单元生成

Leader 通过比对各个签名后得知自己是 Leader，于是打包合约的输入输出、陪审员签名等，再加上引用的父单元、时间戳、签名等生成一个合约执行单元。

超时重新选举 Leader

如果所有陪审员在等待 N 秒后，仍然没有收到 Leader 打包的合约执行单元，则认为 Leader 节点故障，改为签名哈希第二小的陪审员节点担任 Leader 节点打包合约执行单元。如果等待 N 秒仍然没有收到单元，则重复执行选举 Leader 和打包的工作。

陪审团内部采用了 BFT 的共识模式，由合约执行结果的签名哈希来选定 Leader，Leader 最终享有本次执行的手续费。如果陪审员的计算机配置较差，导致执行合约缓慢，无法及时得到合约的执行结果，那么就可能错过担任 Leader 打包单元和收取手续费的机会，所以陪审员也有动力去配置更好的计算机资源来增加获得交易手续费的机会。

分布式账本存储

PalletOne的分布式存储结构如图4.1所示。

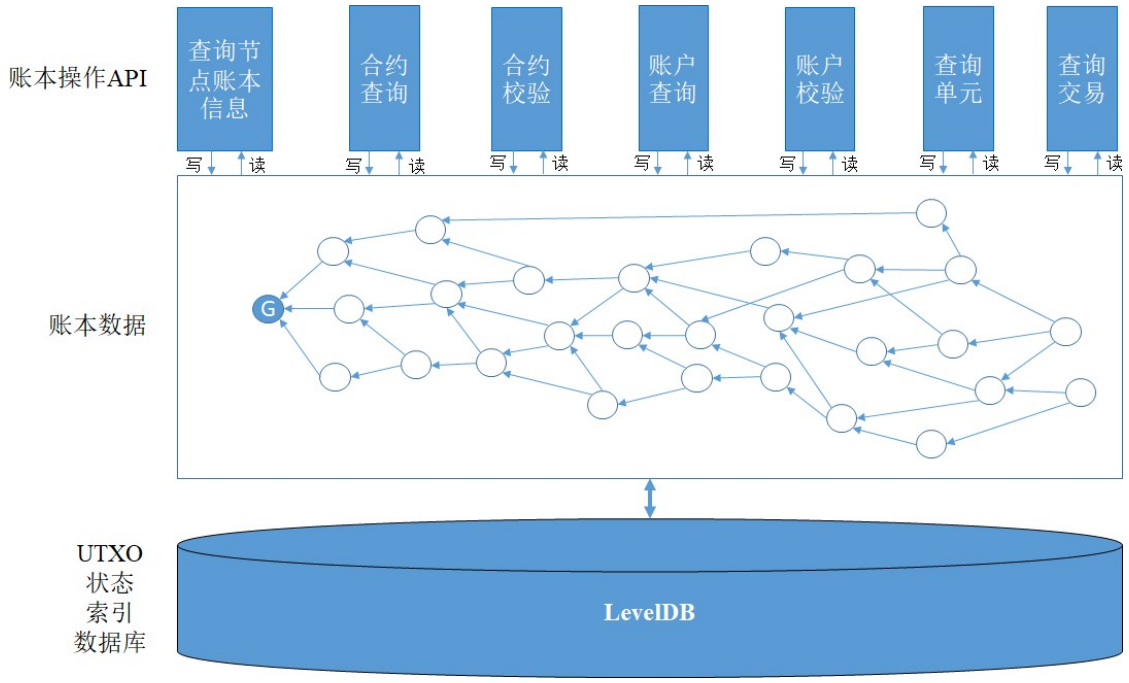

图 4.1 PalletOne 分布式存储结构

有向无环图

有向无环图（Directed Acyclic Graph）简称 DAG。在 DAG 中没有区块概念，所有数据并不打包成区块再用区块链接区块，而是每个用户都可以提交一个数据单元，这个数据单元里可以有很多东西，比如交易、消息等。数据单元间通过引用关系链接起来，如图 4.2 所示。DAG 的特点是把数据单元的写入操作异步化，大量节点可以自主地把交易数据写入 DAG。PalletOne 使用 DAG 的目的是为了解决传统区块链中只有一条主链，无法并行执行的问题，同时节省了打包交易出块的时间。

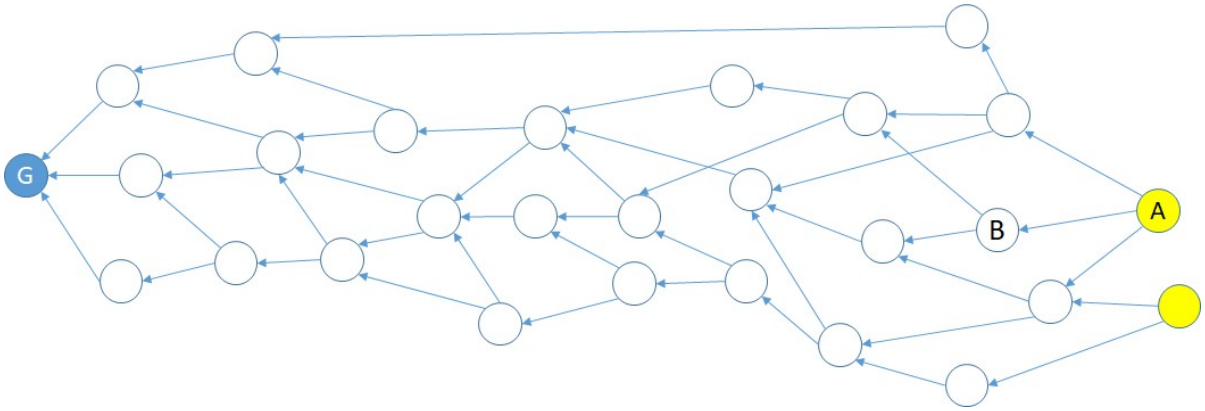

图 4.2 PalletOne 中 DAG 结构

传统的区块链里，在生成区块之前，需要给所有的交易做一个交易池，矿工从交易池中挑选要打包的交易，然后根据共识算法打包为区块，最后放到链中。在交易被打包到区块并广播到全网之前，交易是属于未确认状态，这阻塞了其他的交易写入，从而使得传统区块链交易确认时间非常长。

DAG 并行处理的特点使其有潜力为各行业的经济活动和价值交易带来全新的技术革新与升级，在弥补不同交易主体之间信任鸿沟的同时，通过时间戳、不可逆性、可追溯性、并行处理等特点，降低交易成本，实现具有更强表达力的智能合约，更快的交易确认，更广泛的应用场景，更强的安全性和隐私保护。

单元结构

在图 4.2“PalletOne 中 DAG 结构”中，PalletOne DAG 的存储单元结构为 Unit（单元）。在 PalletOne 中，存储单元包括以下几种单元：

（1）创世单元，也称为 G 单元，由创世交易所构成的单元，不具有任何父单元，是所有后续单元通过引用关系可以到达的终点。

（2）父单元和子单元，假设存在单元 A 和单元 B，从单元 A 出发可直接到达单元 B，即单元 A 到单元 B 的路径长度为 1，则单元 B 称为单元 A 的父单元，单元 A 称为单元 B 的子单元。

（3）顶端单元，不具有任何子单元的单元，也可称为无子单元或未经验证的单元，如上图中标记为黄颜色的单元。

单元结构由以下几部分组成：

（1）单元数据：数据以 Message 的形式构成；

（2）地址签名：输入所需的相应地址签名；

（3）父单元：当前单元的父单元列表。

其中 messages 字段中 app 的分类包括以下几种：

（1）payment，PalletOne 通证交易；

（2）contract_template，智能合约模板部署；

（3）contract_deploy，智能合约创建（实例化）；

（4）contract_invoke，智能合约的调用；

（5）verify，交易/合约部署/合约创建/合约调用的见证

（6）……

下面将对上述的五种 messages 的分别进行说明

payment：通证交易

下面以 PalletOne 通证转账交易单元结构为例说明存储单元的 JSON 格式：

```
{
    version: '1.0',
    messages: [{
          app: 'payment',
          payload_location: 'inline',
          payload_hash: 'the hash text of payload field',
          payload: {
                inputs: [{
                      unit: 'create token unit hash and this unit must have
became a stable packet',
                      message_index: 0,
                      output_index: 1
                }],
                outputs: [{
                      address: 'transfer to address',
                      amount: 208,
                      asseId: 'an immutable identifier of an asset that is
created at issuance',
                      uniqueId: 'every token has its unique id'
                },
                ......]
          }
    }],
    authors: [{
          address: 'create or sign author addresses for this unit',
          pubkey: 'the authors's public key if needed ',
          authentifiers: {
                r: 'the authors 's ECC signature'
          }
    }],
    parent_units: ['referenced unit hash text', 'referenced unit hash
text', 'referenced unit hash text'],
    last_packet: 'the hash of last packet',
    last_packet_unit: 'the unit hash of last packet',
}
```

其中：

（1） version（版本）是协议版本号。将根据这个版本的协议解释该单元；

（2） messages（消息）是包含实际数据的一个或多个消息的数组：

- app 代表信息类型，比如说“payment”代表支付，“text”代表任意文本信息等。
- payload_location 说明在哪里找到消息的有效载荷。如果有效载荷包括在消息中，则可以是"inline”，如果在互联网地址可用有效载荷，则是“uri”，如果未发布有效载荷，则是“non e”，被私有地存储和/或共享，并且 payload_hash 用于证明它存在于特定的时间；
- payload_hash 是 base64 编码中的有效载荷散列。

payload 就是真实的荷载量（上例中是“inline”）。有效载荷结构是特定的应用程序。支付说明描述如下：

- inputs 输入代表了一串由支付（指令）消耗的输入货币。输入货币所有者必须在单元的签名者（authors）之中；
  - unit 是指生产货币的哈希单元。该单元必须计入在最后一个 packet 的单元才可消费；
  - message_index 是 UTXO 消息的索引；
  - output_index 是指 UTXO 输出的索引；
- outputs 是一组输出，表明谁收到钱；
  - address（地址）是指收件人的地址，可以是外部账户地址，也可以是合约地址
  - amount 是他收到的金额；
  - assetId 是指发行 token 标识时生成的不可变标识符；
  - uniqueId 是每类 token 的唯一标识符

（3） authors 是指创建和签署本单元的作者数组。所有输入货币必须属于作者；

- address 是作者 Token 对应的账户地址；
- pubkey 签署该单元作者的公钥，该字段是可以选字段。在需要的时候可以添加，比如认证单元等。
- authentifiers 是作者的签名。最常见的是 ECDSA（椭圆曲线数字签名算法）签名；

（4） parent_units 是父母单元的散列数组。它必须按照字母顺序排列；

（5） last_packet 和 last_packet_unit 分别是最后一个 packet 及其单元的哈希值。

单元结构中的所有哈希值都采用 base64 编码。

contract_template：智能合约模板部署

```
unit: {
    …
    messages: [{
          app: "contract_template",
          payload_location: "inline",
          payload_hash: "hash of payload field",
          payload: {
                config: "configure xml file of contract",
                bytecode: "contract     bytecode",
          }
    },
    {
          app: 'payment',
          payload_location: 'inline',
          payload_hash: 'hash of payload field',
          payload: {
                inputs: [{
                      unit: 'create token unit hash and this unit must have
became a stable packet',
                      message_index: 0,
                      output_index: 1
                }],
                outputs: [{
                      address: 'the address of contract issuer, the
difference between inputs and outputs is the transaction fee',
                      amount: 208,
                      asseId: 'an immutable identifier of an asset that is
created at issuance',
                      uniqueId: 'every token has its unique id'
                },
                ]
          }
    },
    ],
    authors: [{
          address: 'create or sign author addresses for this unit',
          pubkey: 'the authors's public key if needed ',
          authentifiers: {
                r: 'the authors 's ECC signature'
          }
    }],
    …
}
```

上述单元中，类型为“contract_template”的 messages 字段中 payload 字段下各子字段的含义如下：

（1） config：合约模板配置 xml 文件的序列化值，在 config 中需要指出合约的语言、合约的初始化参数、合约函数详细说明列表等。

（2） bytecode：合约编译文件的打包文件序列化值。因为合约最终不只一个文件，还可能有许多引用文件或者库，所以以 zip 打包的形式传输。

上述单元中，类型为“**payment**”的 messages 字段表示的是合约部署的手续费。outputs 指向的是合约部署作者自己的地址，inputs 和 outpus 之间的差价即为支付给陪审团的交易费。

authors 字段的含义和“payment”中字段的含义相同。

contract_deploy：智能合约的创建（实例化）

```
unit: {
    …
    messages: [{
          app: "contract_deploy",
          payload_location: "inline",
          payload_hash: "hash of payload field",
          payload: {
                template_id: "contract template id",
                config: "configure xml file of contract instance
parameters",
          }
    },
    {
          app: 'payment',
          payload_location: 'inline',
          payload_hash: 'hash of payload field',
          payload: {
                inputs: [{
                      unit: 'create token unit hash and this unit must have
became a stable packet',
                      message_index: 0,
                      output_index: 1
                }],
                outputs: [{
                      address: 'the address of contract issuer, the
difference between inputs and outputs is the transaction fee',
                      amount: 208,
                      asseId: 'an immutable identifier of an asset that is
created at issuance',
                      uniqueId: 'every token has its unique id'
                },
                ]
          }
    },
    ],
    authors: [{
          address: 'create or sign author addresses for this unit',
          pubkey: 'the authors's public key if needed',
          authentifiers: {
                r: 'the authors 's ECC signature'
          }
    }],
    …
}
```

上述单元中，类型为“**contract_deploy**”的 messages 字段中 payload 字段下各子字段的含义如下：

（1） template_id：合约模板 ID。

（2） config：合约配置 xml 文件的序列化值，在 config 中需要指出合约的初始化参数的实际初始值。

上述单元中，类型为“**payment**”的 messages 字段表示的是合约部署的手续费。outputs 指向的是合约部署作者自己的地址，inputs 和 outpus 之间的差价即为支付给陪审团的交易费。

authors 字段的含义和“payment”中字段的含义相同。

contract_invoke：智能合约的调用

```
unit: {
    …
    messages: [{
          app: "contract_invoke",
          payload_location: "inline",
          payload_hash: "hash of payload",
          payload: {
                contract_id:  “the hash id of contract”
                function: "serialized value of invoked function with call
parameters"
          }
    },

    {
          app: 'payment',
          payload_location: 'inline',
          payload_hash: 'hash of payload field',
          payload: {
                inputs: [{
                      unit: 'create token unit hash and this unit must have
became a stable packet',
                      message_index: 0,
                      output_index: 1
                }],
                outputs: [{
                      address: 'the address of contract issuer, the
difference between inputs and outputs is the transaction fee',
                      amount: 208,
                      asseId: 'an immutable identifier of an asset that is
created at issuance',
                      uniqueId: 'every token has its unique id'
                },
                ]
          }
    },

    ],
    …
}
```

上述单元中，**contract_invoke** 类型 messages 中 playload 各字段的说明如下：

（1） contract_id：要调用的合约 ID；

（2） function：带调用参数的函数序列化值。

上述单元中，**payment** 类型 messages 的含义与合约部署一样。

verify：交易/合约部署/合约创建/合约调用的见证

```
{
    version: '1.0',
    messages: [{
        app: 'verify',
        payload_location: 'inline',
        payload_hash: 'the hash text of payload field',
        payload: {
            inputs: [{
                unit: 'create token unit hash and this unit must have
became a stable packet',
                message_index: 0,
                output_index: 1
            }],
            outputs: [{
                address: 'transfer to address',
                amount: 208,
                asseId: 'an immutable identifier of an asset that is
created at issuance',
                uniqueId: 'every token has its unique id'
            },
            ......]
        }
    }],
    authors: [{
        address: 'create or sign author addresses for this unit, the
first author could be the unit author',
        pubkey: 'the authors's public key if needed ',
        authentifiers: {
          r: 'the authors 's ECC signature'
        }
       }],
    parent_units: ['referenced unit hash text', 'referenced unit hash
text', 'referenced unit hash text'],
    last_packet: 'the hash of last packet',
    last_packet_unit: 'the unit hash of last packet',
}
```

上述单元中，各字段的含义与 payment 类型的含义基本一致。authors 中将包含见证单元的所有见证人的签名信息。

状态存储

账户状态

PalletOne 在底层采用了 UTXO 的模型，有别于以太坊的账户模型，但在 PalletOne 中账户也存在外部账户和合约账户，外部账户又分为单签账户和多签账户。无论是外部账户还是合约账户，都存在以下状态：

- codeHash：此账户中拥有的合约代码的 Hash。
- storageRoot：此账户中存储的状态数据的 Hash Root。

智能合约存证

在 DAG 中，合约状态都是实时并行写入到 DAG 中，因此合约的状态信息直接存储在 DAG 的单元结构中。在外部账户和合约账户中，仍然保留了 cod eHash 和 storageRoot 两个状态是为了方便账户的查询和存证。

通证定义

在 PalletOne 中，Token 定义都是通过通证抽象层来定义。Token 不在合约状态中存储，而是在独立的 Token 区域存储。

对于通证抽象层创建的通证，他们固定只有类似于 ERC20 和 ERC721 定义的几种外部接口，这类通证是直接存储在 UTXO 数据库中。然后通过 asset Id 和 uniqueId 进行区分。

如果我们需要定义高阶的通证操作，则需要编写自定义的合约来完成。合约可以关联自定义的通证，由合约拥有该通证的控制权。

数据库设计

在 PalletOne 中，我们将设计四种数据库，如图 4.3 所示。这些数据库都是用 key-value 格式的 LevelDB 进行存储。四种数据库分别是：

- DAG 原始数据库——所有的数据都以 DAG 的结构存储在该数据库中。
- 索引数据库—— 一些合约状态历史以及其他需要在智能合约中检索的历史数据都存储在该数据库并建立索引。
- UTXO 数据库——所有通证的 UTXO 数据。
- 合约状态数据库——所有合约的最新状态数据。

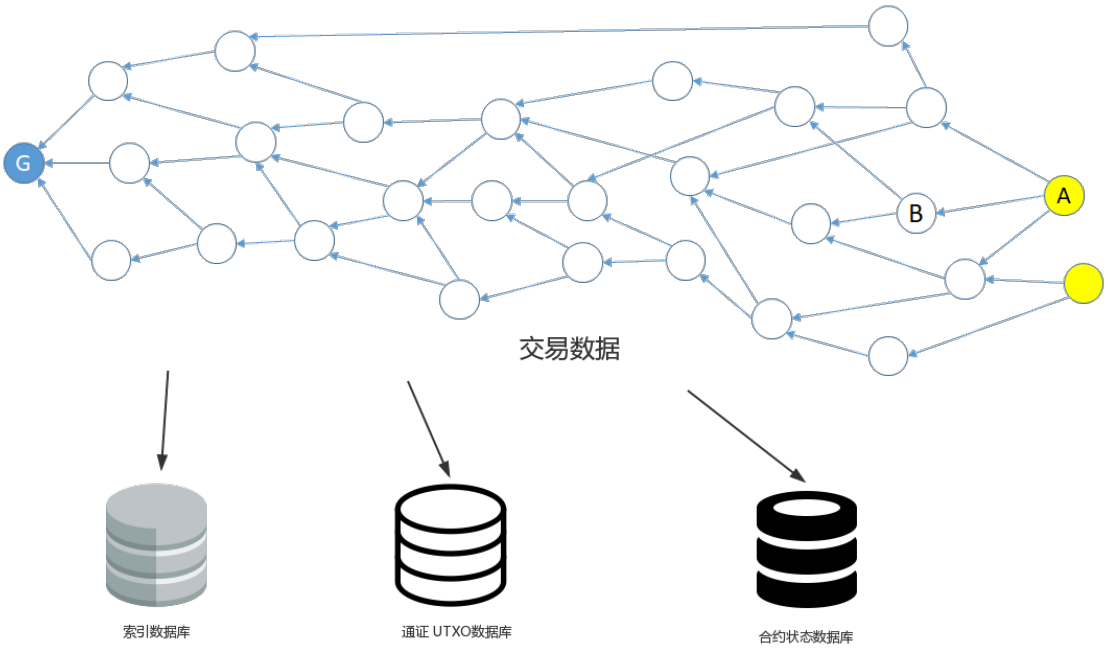

图 4.3 PalletOne 数据库

在 PalletOne 中数据库底层使用使用 LevelDB 进行存储。LevelDB 是一个 key-value 数据库，所有数据都是以键-值对的形式存储。key 一般与存储类型和 hash 相关，value 一般是要存储的数据结构。

UTXO 存储

参考比特币的 UTXO 存储格式，在 PalletOne 中 UTXO 的存储格式如下：

- Key

‘c’（或者 0x63）	64 字节 ballHash

- value

the non-spent CTxOut	nHeight

其中：

the non-spent CTxOut 的结构如下：

amount	scriptType	address	assetId	uniqueId

其中：

（1） amount: output 的数量

（2） scriptType：脚本类型，00-P2PKH；01-P2SH 3. address：output 地址

（3） assetId：通证唯一标识符

（4） uniqueId：某个通证的类型标识符

合约状态存储

- Key：’a’+合约账户地址
- value

codeHash	storageRoot

DAG 原始数据存储

1. Unit 数据存储
   - key：'u' + Unit hash
   - value：rlpEncode(unitdata)

2. Packet 数据存储
   - key：'b'+Unit hash
   - value: rlpEncode(packet data)

3. 交易存储
   - key: 't'+payload hash
   - value: rlpEncode(payload data)

DAG 索引数据存储

1. 交易-单元索引
   - key: ’tu‘+payload hash
   - value: unit hash

2. 合约-交易索引
   - key: 'ct'+contract id
   - value: payload hash

DAG 网络

记账权选择

在 PalletOne 中，有以下几种情况需要记录 Unit：

1. 普通交易单元

普通交易单元的记账权属于交易发起者。

2. 合约模板部署单元

合约模板单元的记账权属于在某个时间片工作的 Mediator 成员。

3. 合约创建（实例化）单元

合约创建（实例化）的记账权属于在某个时间片工作的 Mediator 成员。

4. 合约调用单元

合约调用单元的记账权属于陪审团 Leader。需要注意的是，在合约创建和合约调用的时候，Leader 拥有记账权，但是 Leader 需要获得所有陪审团成员的多数（>1/2 或者全部）签名后才能进行记账。

5. 交易确认单元

交易确认单元的记账权由当前活跃的 Mediator 成员来进行记账。但是交易的确认是由 Mediator 中所有成员完成签名后才能进行记账。

6. 合约模板部署认证单元

合约模板认证单元的记账权同样由当前活跃的 Mediator 成员来进行记账，Mediator 对合约创建的认证工作主要有：合约参数检查、合约代码检查、交易费的检查等。同样，合约模板部署认证需要 Mediator 中大多数成员完成签名后才能进行记账。

7. 合约创建认证单元

合约模板认证单元的记账权同样由当前活跃的 Mediator 成员来进行记账，Mediator 对合约创建的认证工作主要有：合约参数检查、合约代码检查、初始化参数检查、交易费的检查等。同样，合约创建认证需要 Mediator 中大多数成员完成签名后才能进行记账。

8. 合约调用认证单元

合约调用认证单元的记账权同样由当前活跃的 Mediator 成员来进行记账，Mediator 对合约调用的认证工作有：调用合规性检查、方法参数检查等。同样，合约创建认证需要 Mediator 中大多数成员完成签名后才能进行记账。

父单元选择

父单元的选择分为两种情况，一种情况是普通交易单元、合约模板部署单元、合约创建单元、合约调用单元的父单元选择；一种情况是认证单元的父单元选择。

第一种情况父单元的选择原则是最近原则，即对所有子单元根据单元哈希进行排序，然后尽可能选择多的较小的单元为父单元。该原则的目的是为了将 DAG 进行收敛，最后形成一条链。

第二种情况父单元的选择原则是直接引用被认证的交易单元。

packet 的生成

在单元稳定后，基于单元结构，我们生成一个新的结构，如下所示：

```
packet: {
        unit: "hash of unit",
        parent_packets: [array of hashes of packets based on parent units],
        is_nonserial: true, // this field included only if the unit is
nonserial
        skiplist_packets: [array of earlier packets used to build
skiplist]
}
```

引入 packet 概念的目的是为了帮助轻节点进行验证。其中 skiplist_packets 的目的是加快轻节点认证。只有那些直接在主链（MC）上的 packet 有一个跳表，其 MC 指数可以被 10 整除，跳表列出最接近之前 MC 的 packet，其索引在末端具有相同或较少数量的零点。例如，MCI（MC 指数） 190 处的 packet 具有参照 MCI 180 处的跳表。在 MCI 3000 处的 packet 具有参照 MCI2990，2900 和 2000 处的跳表。

其中：

- 单元是否稳定是在确定主链的过程中确定的，其中创世单元是默认的稳定点。
- MCI 表示主链索引值。创世单元的 MCI=0，创世单元的子单元 MCI=1。对不在主链上的单元 MCI 值为首先包括（直接或者间接）该单元的 M C 索引。

最后一个 packet

为了保护 packet（最重要的是，非序列标识）不被修改，我们要求每个新单元包含作者所知的最后一个 packet 的哈希值（这是从最后一个稳定单元建立的 packet，并且它位于 MC 上）。这样，最后一个 packet 将受到作者签名的保护。之后，认证单元将（直接或间接）计入新单元本身。

主链

主链的确认过程分为三步：

1. 最优父单元选择

a) 见证级别最高的父单元为最优父单元；

b) 如果见证级别相同，则单元级别最低的作为最优父单元；

c) 如果两者都相同，则选择单元哈希值（base64 编码）更小的作为最优父单元。

2. 候选主链生成

从任何一个顶端单元出发到达创世单元的最优路径称为候选主链。而候选主链通过最优父单元选择来确定。不同的候选主链会在某个单元位置交叉（最差的情况是在创世单元交叉），该交叉点称为稳定点。

3. 稳定主链生成

对于所有候选主链，从稳定点到创世单元的路径完全相同，该路径称为稳定主链（Stable Mainchain），如图 4.4 所示。稳定主链是一条确定的路径，从候选路径变为稳定主链是一个从不确定逐渐变成确定的过程

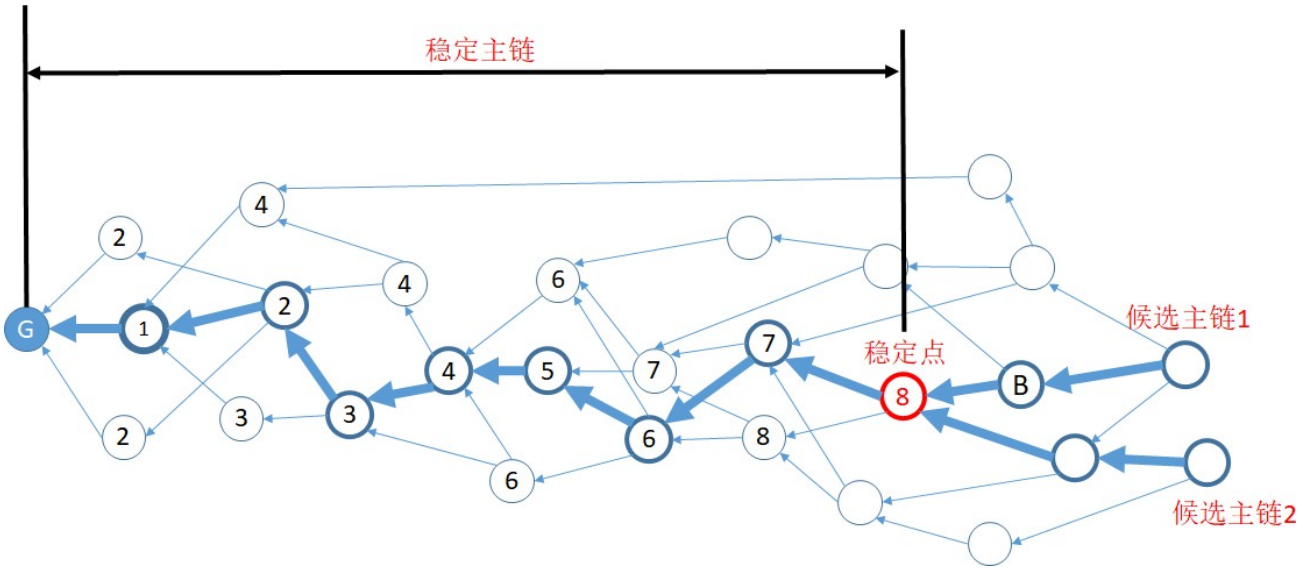

图 4.4 PalletOne 主链

其中：

- 单元级别：由当前单元出发至创世单元的最长路径长度定义为单元级别（unit level）。
- 见证级别：从当前单元开始沿主链回溯，并对路径中不同见证人（Mediator）进行计数（相同见证人只计数 1 次），当遇到的见证人数足够多时（由于 Mediator 是在不停的轮换，Mediator 的成员数量是 21，所以此处可以设置为 11 个见证人）停止回溯；然后计算停止位置的单元级别，将其称作当前单元的见证级别（witnessed level）。
- 当前主链：由于在某一时刻，从不同顶端单元出发具有不同的候选主链，我们将从见证级别最高的顶端节点出发的候选主链称为当前主链（Curre nt Mainchain）。

双花

在同一个地址发出新单元时，要求相同地址发布的所有单元应当直接或间接包含该地址之前所有的单元，即相同地址的所有单元连通（保证有序或连续）。所以双花的判断可以从以下两种情况来进行：

1. 单元连续

在这种情况下，在 DAG 中出现较早的交易为有效交易。

2. 单元不连续

这种情况下类似于影子链的情况，可以根据主链序号来解决，主链序号较小的交易为有效交易。因为影子链上单元的见证级别都比较低，通过最优父单元的确认，则影子链不可能在稳定主链上。

防篡改

由于 DAG 存储结构的特殊性，任何一个单元在成为父单元后，如果需要修改该单元的数据，则单元哈希相应改变，则引用该单元的所有子单元都需要进行修改。而 DAG 父单元选择策略是具有随机性，所以需要多人协同作恶才能修改 DAG 结构。同时，即使作恶者立即发布多个单元引用被修改单元，但是不同单元是被不同的 Mediator 成员认证，而 Mediator 成员的作恶成本比较高，因为一旦发现他们作恶不但会没收保证金，同时他们将没有资格再担当 Mediator 赚取交易费。

UTXO 与账户的同步

对比量子链

在量子链中，底层架构是比特币的架构，为了支持 EVM，在比特币的脚本语言中增加了 3 种操作：

- OP_EXEC：通常用来部署新合约。
- OP_EXEC_ASSIGN：通常用来进行 Token 转账给合。
- OP_TXHASH：合约向其他地址转账。

其中 OP_EXEC_ASSIGN 和 OP_TXHASH 用于表示 vint 的花费方式，OP_EXEC 用于将 vout 中的数据传递给 EVM，由 EVM 执行相关代码，同时更新 Contract outputs。所以，在量子链中设计了合约抽象层（Account Abstract Layer，AAL）作为比特币 core 与 EVM 的通信层。

PalletOne 与量子链有如下的不同：

1. 我们仅支持比特币标准的 P2PKH 和 P2SH 两种脚本，并且合约的执行与脚本执行是独立分开的。

2. 由于我们使用“单元”存储结构，在合约部署和合约调用的两种单元结构中，Token 的转账、交易费支付等操作都是通过 payment 字段单独进行表示。

3. 不存在单独的 contract outputs 列表，与合约相关的 Token 也都存储在 UTXO 中。

4. PalletOne 的 UTXO 中增加了 assetId、uniqueId，从而扩大了 Token 的表示范围。

getBalance(address)

getBalance(address)：PalletOne Core 调用接口，用于查询某个账户余额信息。底层操作是查询、求和 UTXO 数据库中某个地址的信息。

Inputs Token 挑选算法

通过分析比特币源码（bitcoin-master/wallet/wallet.cpp）中 Inputs Token 挑选接口 bool CWallet::SelectCoins(.....)和 void CWallet::AvailableCoins(.....)可以认为，比特币的 Inputs Token 挑选策略是 First-In-First-Out 的规则（量子链、Byteball 也使用该规则）。

但是最好的 Inputs Token 挑选算法是在最快时间内找到满足要求如下标准的 UTXO：

1. 尽量不找零；

2. 优先花费小额 Token。

使用这两个标准的目的是为了维持 UTXO 数据库大小的稳定，从而减少 UTXO 数据库操作带来的时间消耗。所以在 PalletOne 中也基本遵守该规则，则 Token 挑选流程大致如下：

1. 将该用户所有的 UTXOs 进行升幂排序。

2. 在序列中查找与目标 inputs（设为 Target）相等的 UTXO；如果找到就返回，如果没有找到，进行步骤 3。

3. 求和最小额的 n 个 UTXOs，当 sum(n of smallest UTXOs)>=Target 时停止。如果 sum(n of smallest UTXOs)==Target，返回。如果 sum(n of smallest UTXOs)>Target 则进行步骤 4。

4. 该步骤可以分为如下几个小步骤：

   a) 去掉 sum 组合中最小的 UTXO，重新计算 sum(n of smallest UTXOs)。

   b) 如果 sum<=Target 时返回，如果 sum>Target 则重复步骤 a。

   c) 当循环次数超过 maxRound 时执行步骤 5。

5. 使用步骤 4 中 maxRound 对应的组合作为 inputs。

交易流程

普通交易流程

普通交易流程图如图 4.5 所示。

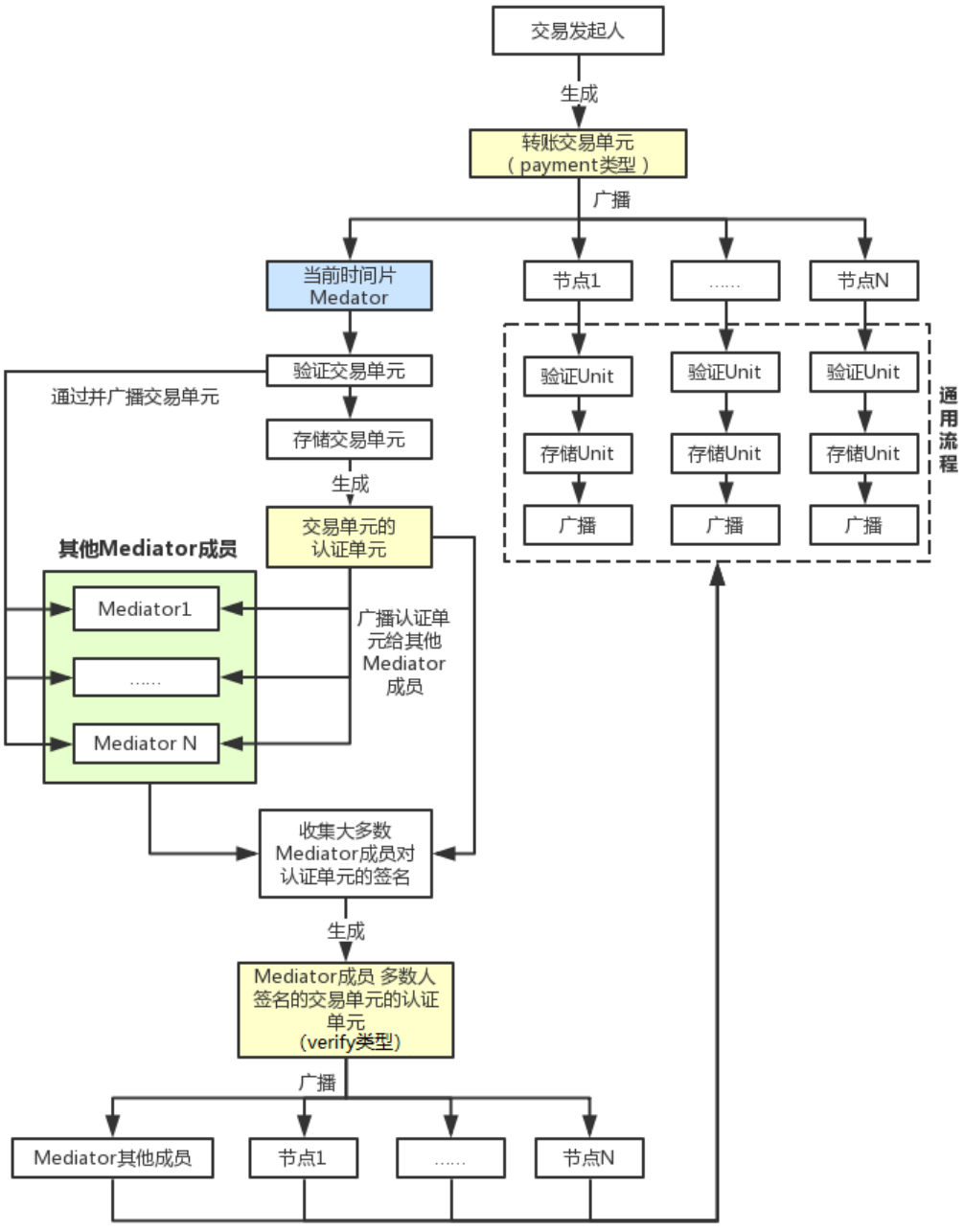

图 4.5 普通交易流程图

合约创建流程

合约创建流程图如图 4.6 所示。

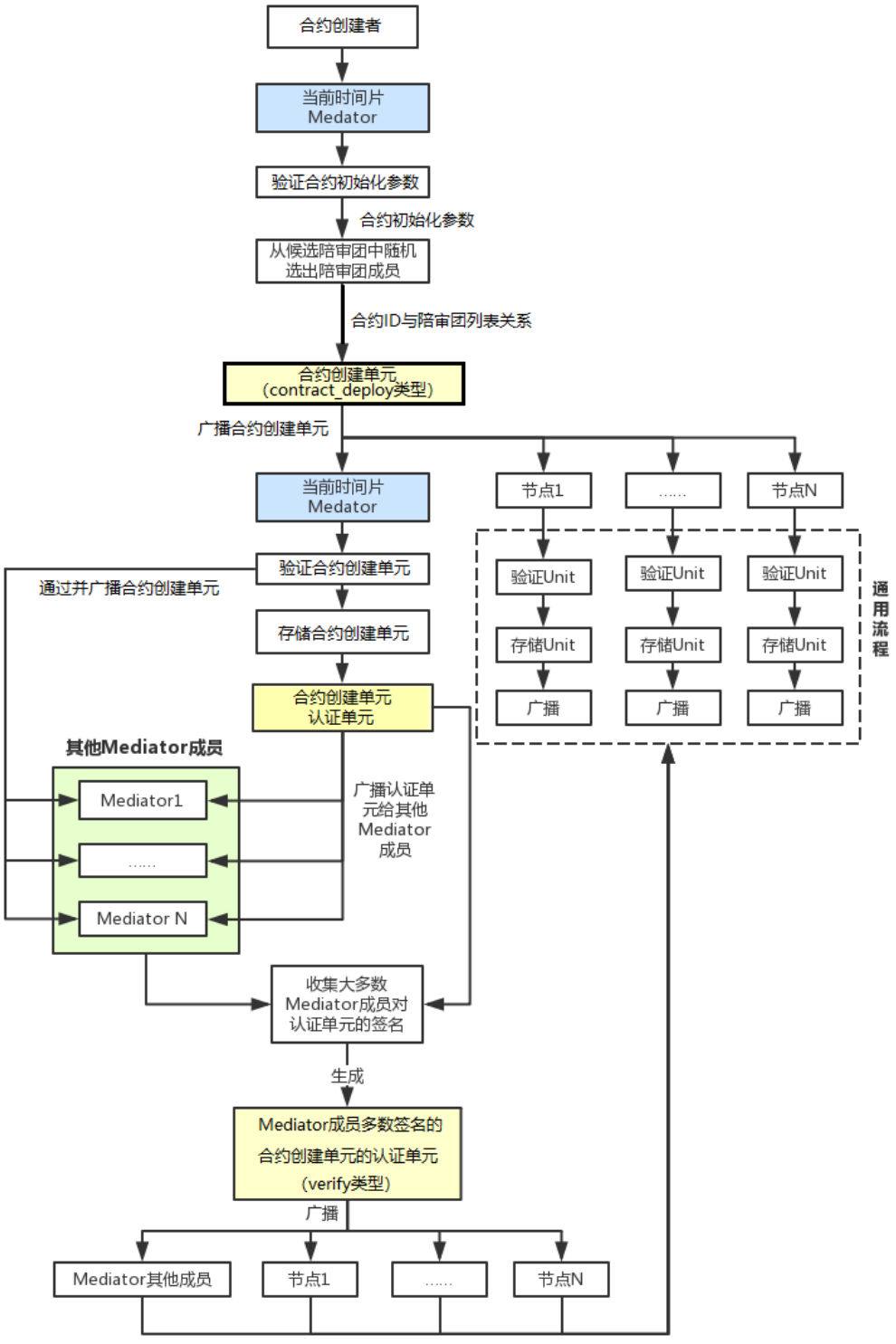

图 4.6 合约创建流程

合约调用流程

合约调用流程如图 4.7 所示。

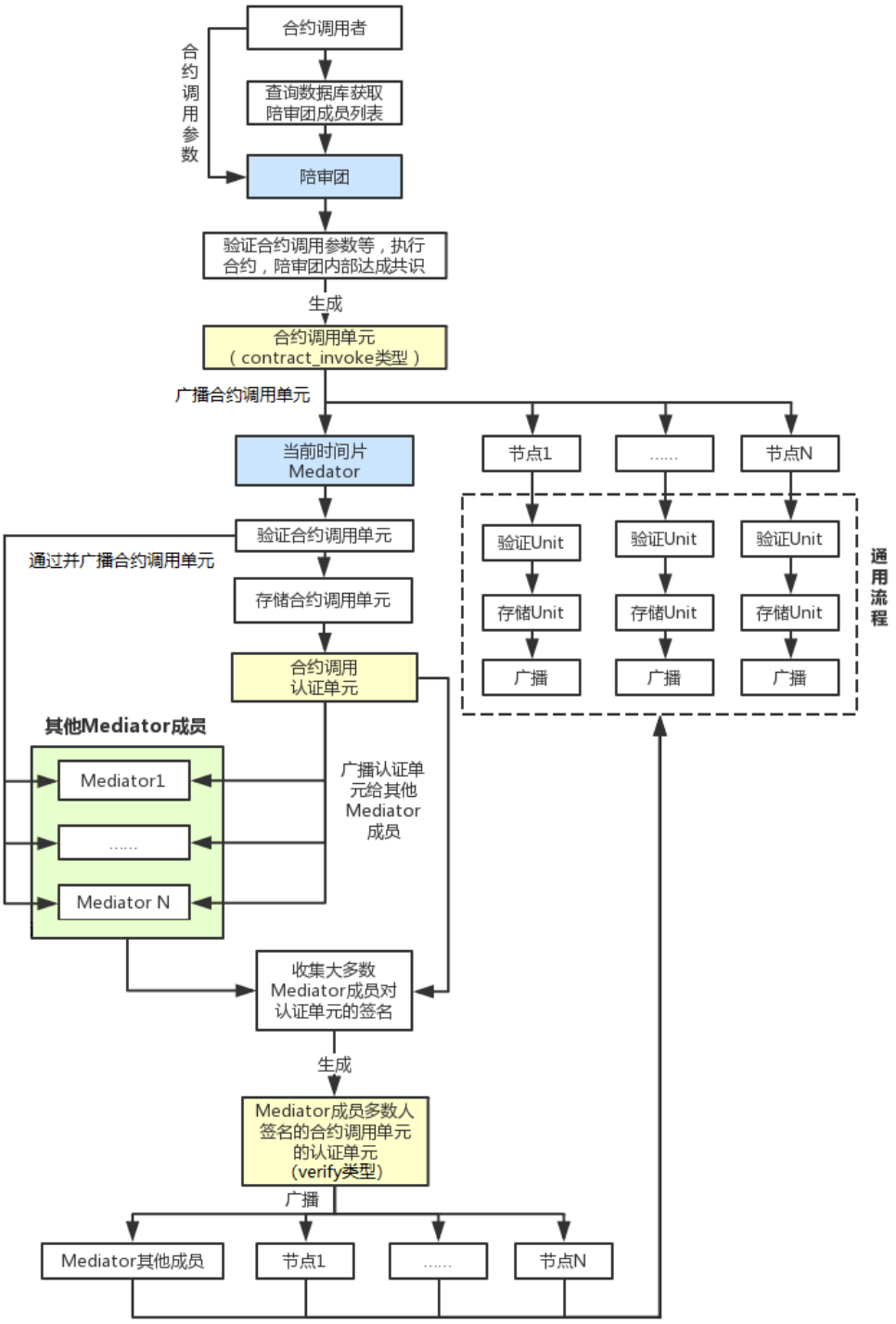

图 4.7 合约调用流程

合约终止流程

合约终止流程如图 4.8 所示。

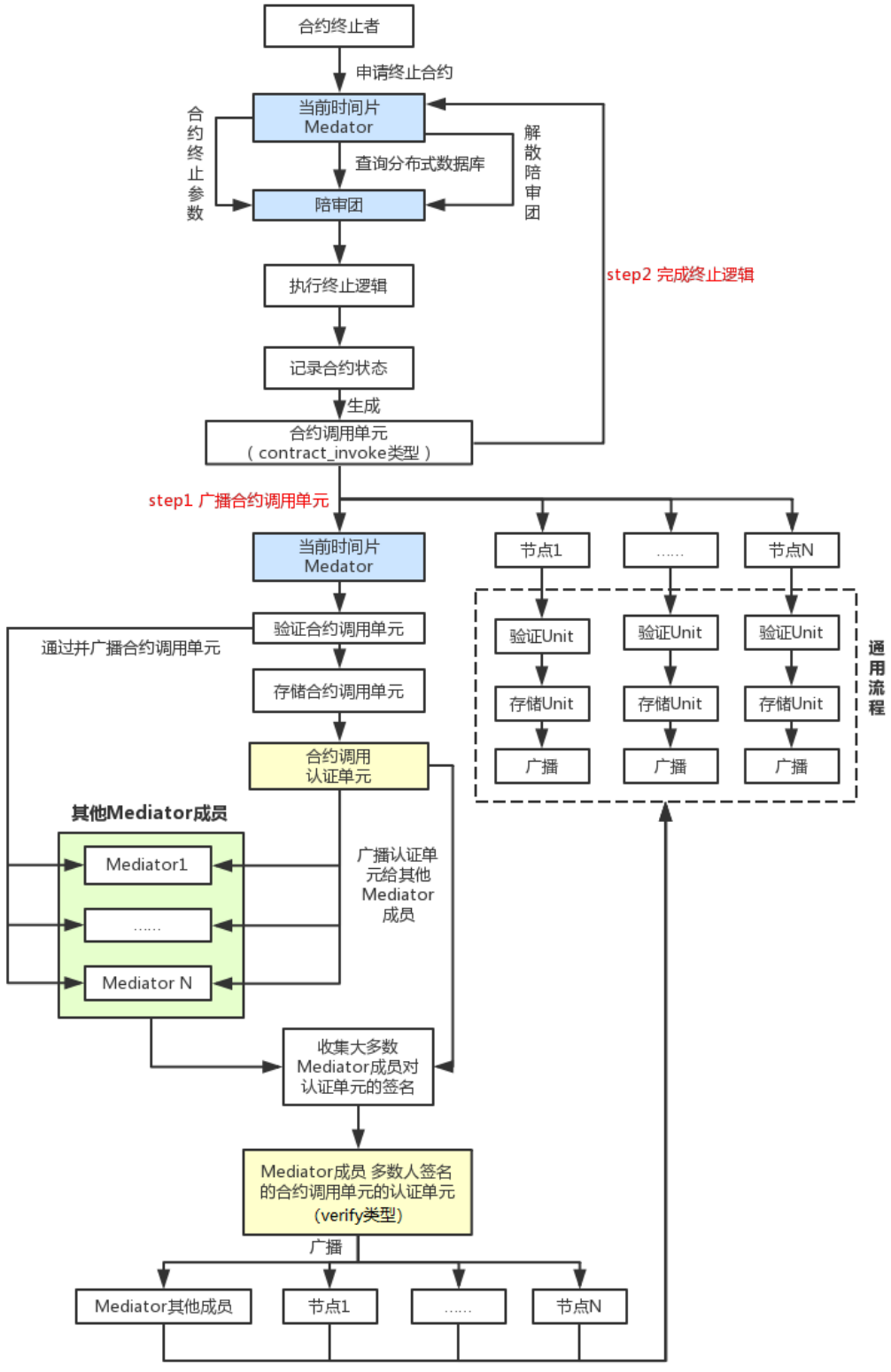

图 4.8 合约终止流程

跨链交易流程

跨链交易的示例说明见“1.BTC 和 ETH 互换(Jury 背书)”一节。跨链交易流程分为如下几步：

1. 由交易双方部署跨链合约交易合约，该流程是合约部署流程。

其中，在生成陪审团后，陪审团需要生成对应的多签账户，并将账号信息返回给交易双方。

2. 交易双方调用合约转账接口分别向多签账户中转账对应金额的货币，该流程是合约调用流程。

3. 交易双方调用合约申请收币接口发起收币申请，该流程是合约调用流程。

在该步骤中，陪审团需要去验证第二步是否已经完成。如果完成，陪审团执行第四步。

4. 陪审团调用转账接口将合约账户中的货币转到对应的交易方账户中，该流程是合约调用流程。

5. 交易方发起终止交易请求，该步骤流程是合约终止流程。

如果交易方没有发起终止交易，则在达到超时时间后，由陪审团发起终止交易请求。上述过程如图 4.9 所示。途中设计到的合约部署、合约调用、合约终止流程与“合约创建流程”、“合约调用流程”中所示相同。

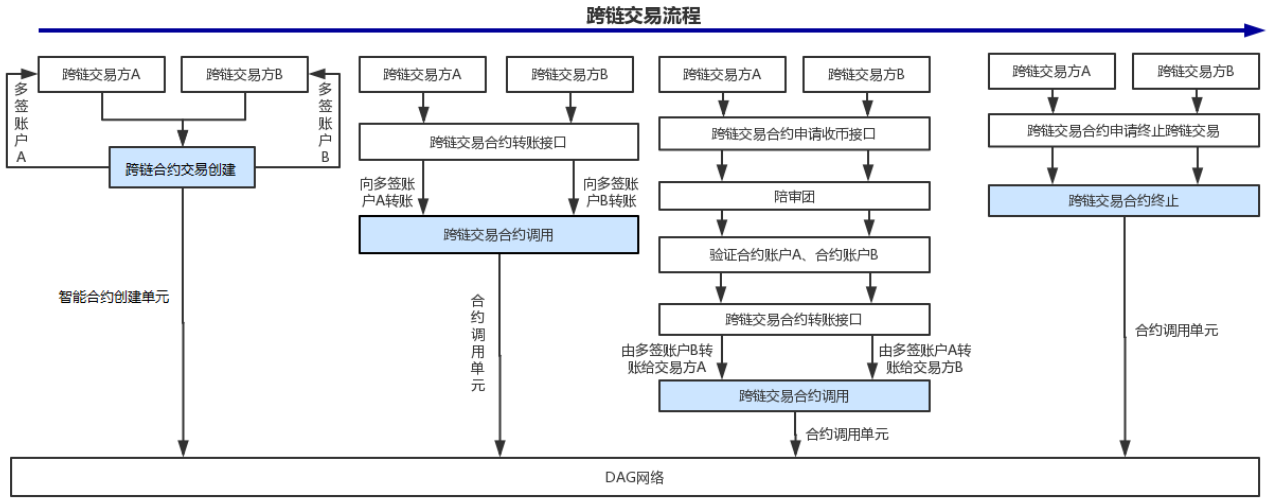

图 4.9 跨链交易流程

智能合约

PalletOne 智能合约概述

智能合约（Smart Contract）这个概念最早由 Nick Szabo 于 1996 年提出，该概念在区块链中被广泛使用，并诞生了以太坊、量子链、EOS 等众多支持图灵完备的智能合约的区块链。概括下来，智能合约具有以下特性：

（1） 智能合约必须是一种合约，是平等的当事人之间执行约定内容的协议。智能合约与传统合约不同之处在于，智能合约是数字形式的，是由计算机读取和执行。

（2） 智能合约是部分或者完全自我执行或者自我强制的。

（3） 智能合约需要安全的运行环境。合约运行环境必须安全可靠，执行结果能在多方达成一致。

（4） 只有智能合约才能修改账本数据。

出于智能合约的特点，合约安全以及在多语言，多平台智能合约上的考虑，PalletOne 默认采用 Docker 容器化技术实现智能合约。在业界，Docker 作为智能合约的虚拟机已经在 HyperLedger Fabric 中应用，被大量的企业和项目所论证。

Docker 容器化——支持多语言的智能合约

系统为不同的编程语言提供了不同的运行时 Docker 镜像，另外也提供了对应语言的 SDK 帮助合约开发人员能够快速便捷的开发出智能合约。合约开发人员基于 SDK 在本地编译好合约程序后，只需要将编译后的文件放到 PalletOne 发布，发布后的合约程序被称为合约模板。用户基于模板创建合约时，合约执行节点（陪审员）会根据合约的语言选择对应的运行时镜像，构建新的合约模板镜像，并最终实例化合约容器，该过程如图 5.1 所示。

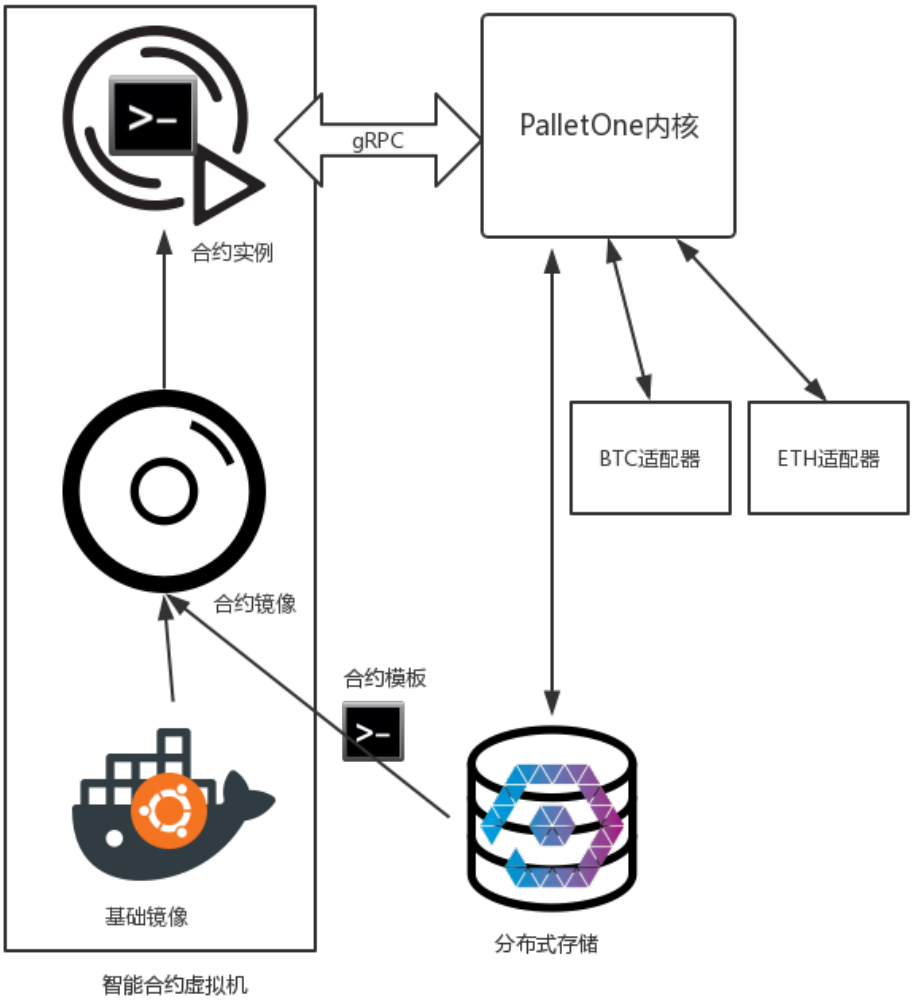

图 5.1 基于 Docker 的合约创建流程

合约的安全性

合约程序在 Docker 提供的沙盒环境中运行，该环境隔绝了主机的环境和网络访问，并只提供了受限的 CPU、内存和硬盘等资源，避免了恶意合约对主机的攻击的可能性。

为了避免合约中出现大量运算或者死循环对主机进行攻击，以太坊使用 Gas 机制避免的这种攻击，而 PalletOne 也使用了类似的惩罚机制。只是不同于以太坊的是 PalletOne 不会详细计算每个指令要消耗的 Gas，因为这将严重影响合约的性能，而且每个指令的定价也是存在争议的事情。在 PalletOne 中，用户在运行合约时需要指定整个合约生命周期所需要消耗的内存量、执行超时时间和愿意支付的手续费 （PalletOne Token 数）。如果合约在规定的时间内未执行完成，陪审员会终止合约的执行，并收取用户支付的 Token。

具体合约系统组成架构如图 5.2 所示。其中 contract manger 负责合约的统一管理。

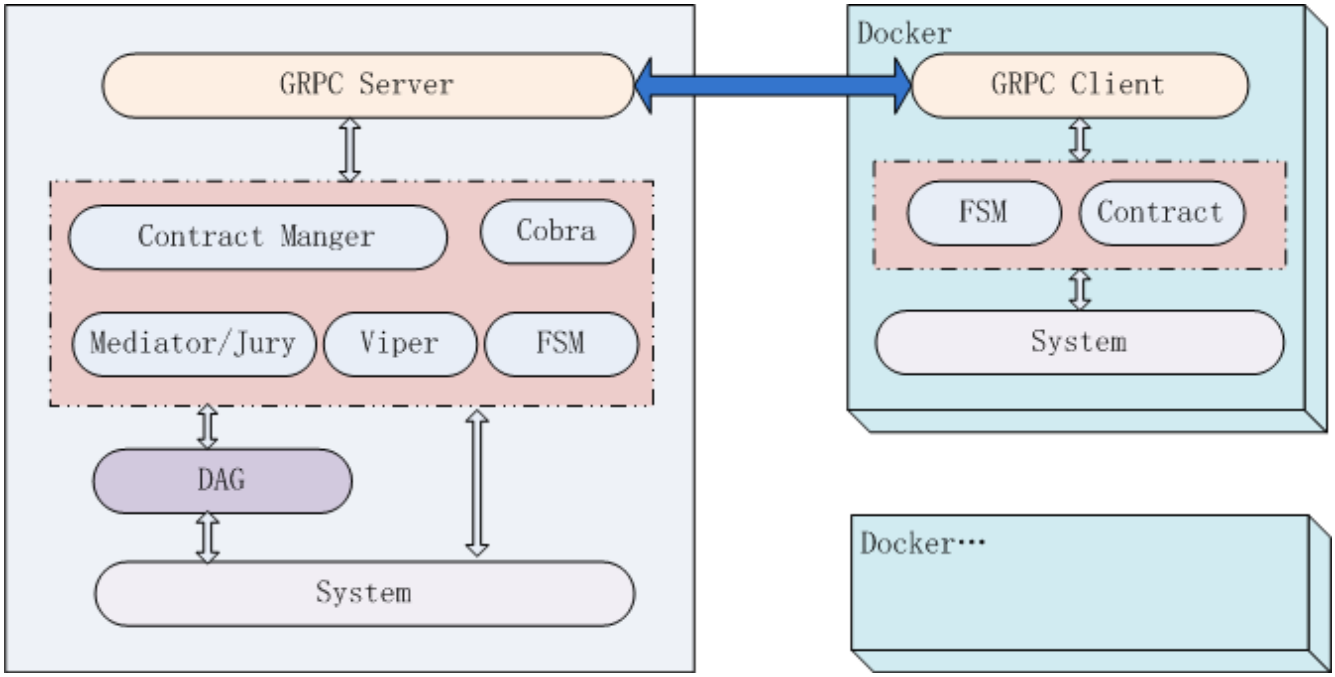

图 5.2 合约系统组成架构

智能合约的生命周期

模板的开发

合约模板就是基于 PalletOne 合约 SDK，符合 PalletOne 合约规范的可执行程序。智能合约开发人员使用自己熟悉的开发语言（PalletOne 前期会提供 C++/Go，Java 的语言支持，后期会加入 NodeJS、C# 等更多语言的支持），在本地编写和编译好合约模板程序，另外出于开发调试的方便，后期 PalletOne 会提供单机 PalletOne 网络的模拟运行。

模板的部署

智能合约开发人员不需要将源代码公布到 PalletOne 网络，只需要将编译后的程序部署到 PalletOne 网络。部署步骤如下：

1. 合约开发人员使用自己的私钥对模板程序进行签名。

2. 合约开发人员将模板程序、模板配置文件（包括接口说明、使用费、签名等），以及部署模板要支付的 PalletOne Token 组合成模板部署请求（TDR: Template Deploy Request）发到 PalletOne 网络。

3. PalletOne 中的调停中介(Mediator)收到模板部署请求 TDR 后会校验模板程序是否与模板接口说明匹配，以及是否符合 PalletOne 模板的策略。

4. PalletOne 调停中介检查合格后，会生成模板 ID，签名并将模板程序及配置记入分布式存储中。

合约部署时序图如图 5.3 所示。

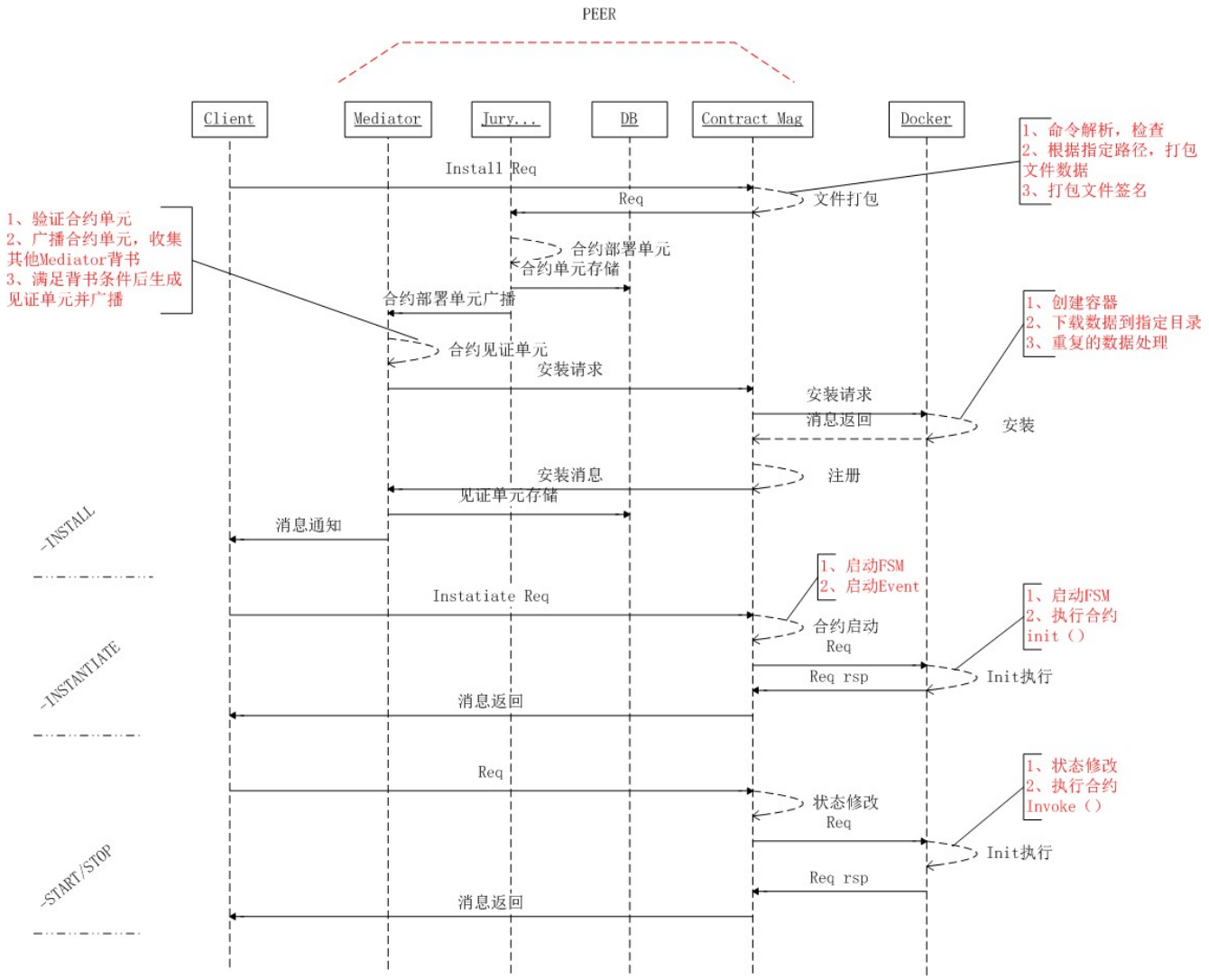

图 5.3 合约部署时序图

合约的创建/初始化

智能合约模板部署到 PalletOne 网络中后，所有用户就可以使用该智能合约了。在 PalletOne 中合约模板就像是 App Store 中的 APP，开发者可以发布收费的合约模板，也可以发布免费的合约模板。对于收费的合约模板，使用者在初始化（实例化）合约时，需要支付一定的 Token 给开发者，该 PalletOne Token 数额由开发者自定。

用户在创建合约时，通过以下步骤完成：

1. 用户在客户端发起合约创建请求（CCR: Contract Creation Request）到 PalletOne 网络，在该请求中包含了：合约模板 ID，初始化参数（具体参数根据不同的合约模板而不同），交易费 PalletOne Token，模板使用费 PalleOne Token，签名等，以及 CCR 的哈希值，也就是合约 ID。

2. PalletOne 中的调停中介在收到合约初始化请求 CCR 之后，根据模板 ID 查询分布式存储中该合约模板相关的配置参数，并根据配置参数决定陪审团的生成模式。

a) 如果是锁定陪审团模式，那么就选定指定人数的陪审员节点组成陪审团，将合约 ID 和陪审员关联起来，并记录到分布式存储中。

b) 如果是非锁定陪审团模式，则记录比指定人数更多的一个陪审员列表，并记录到分布式存储中。

3. Mediator 将 CCR 转发给陪审团。

4. 陪审团的每个陪审员根据收到的 CCR，查询分布式存储中的合约程序和配置，根据合约模板 ID 查找对应的镜像和容器实例，如果没有找到则执行步骤 5 和 6。

5. 陪审员根据合约模板程序语言找到对应的 Docker 基础镜像，结合从分布式存储中拿到的合约程序，创建新的合约镜像。

6. 陪审员根据创建的新合约镜像，创建 Docker 容器实例。

7. 陪审员执行合约中的初始化函数，并将初始化函数返回的结果写入分布式存储中。

合约的调用

用户根据合约 ID 查询分布式存储，可以看到合约是否初始化完毕，是有哪些陪审员分配到该合约等信息。用户根据合约模板的配置说明文件，可以看到合约的所有调用接口。用户调用合约时，需要创建一个合约调用请求（CIR: Contract Invoke Request），该请求包含以下信息：合约 ID，被调用的函数名，参数列表，交易费、签名等。

对于陪审团锁定的合约，用户可以查询分布式存储获得合约对应的陪审员列表，然后将 CIR 发送给陪审团所有陪审员。陪审员在获得 CIR 之后会相互通讯，随机选举出主陪审员（Leader），所有陪审员各自检查合约调用的前置条件（合约模板中定义了每个合约方法的前置条件），并反馈给主陪审员，最后由主陪审员根据陪审员的反馈和合约执行的策略，决定是否真正执行合约调用。上述合约调用时序图如图 5.4 所示。

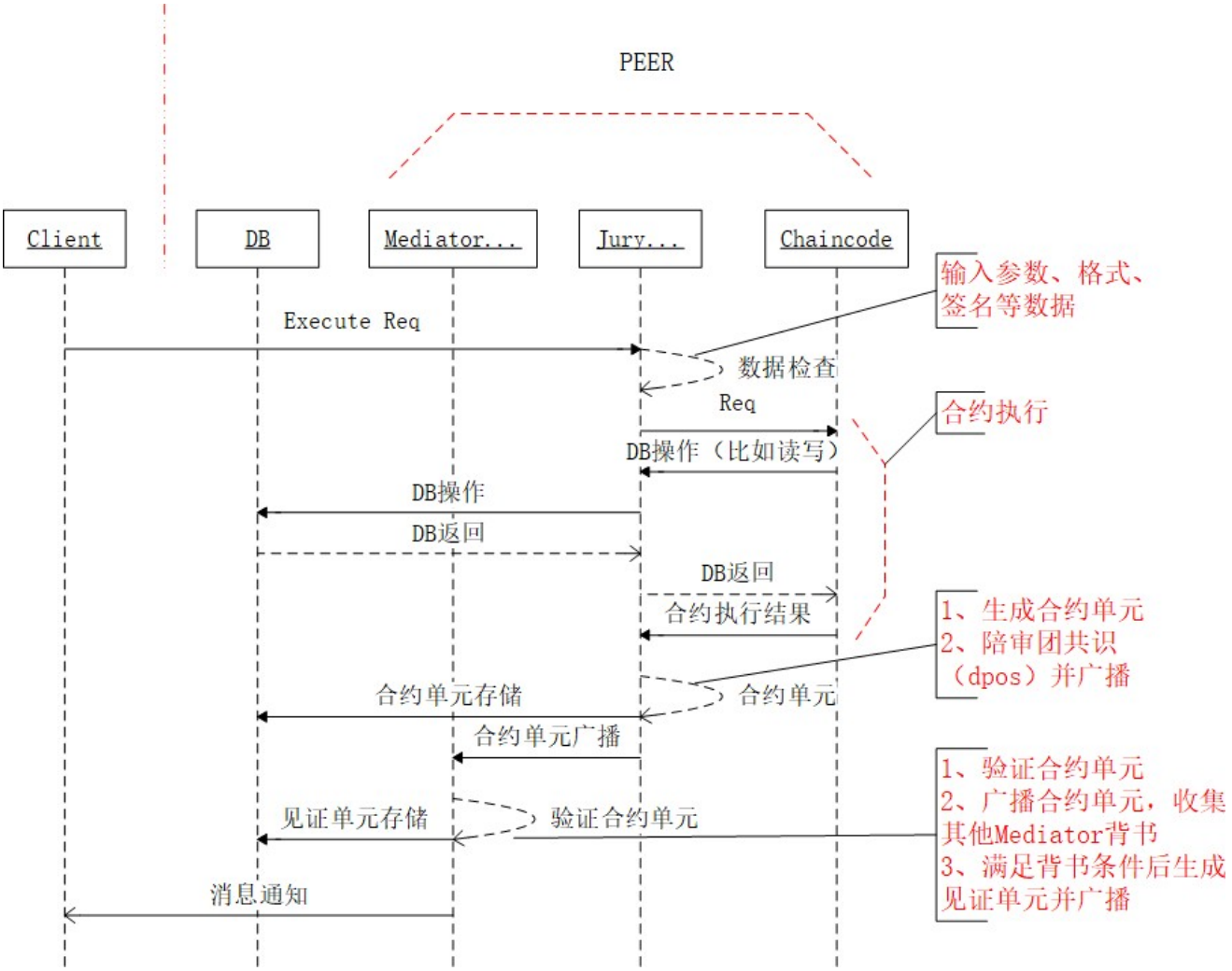

图 5.4 合约调用时序图

合约的升级

以太坊的智能合约在发布后便不再允许升级，使得 DApp 即使有 Bug 也无法修复。PalletOne 引入合约的升级机制，使得合约在必要时可由合约模板创建发起升级请求，然后由合约使用者投票进行升级。

合约的升级过程如下：

1. 合约模板开发人员对合约代码进行修改，编译，然后发起合约模板升级请求（TUR: Template Upgrade Request）到 PalletOne 网络。该请求包含：原合约模板 ID，新合约的程序，新合约接口说明，升级说明，合约实例升级策略（强制升级、选择升级、不升级），升级费用 PalletOne Token，签名。

2. Mediator 收到 TUR 后校验模板程序是否与模板接口说明匹配，以及是否符合 PalletOne 模板的策略。

3. PalletOne 调停中介检查合格后，会生成模板 ID，签名并将模板程序及配置记入分布式存储中。

4. 正在使用该模板老版本的用户在后续使用合约的过程中会收到新版本提示，如果是强制升级，则不允许用户继续使用旧版本，如果是选择升级，用户可以投票选择是否升级。

5. 陪审团收到用户的升级确认或者投票后，会从分布式存储中检索 TUR，升级本地的合约模板镜像和合约容器。

合约的终止

在合约执行完毕后，或者合约双方因为某些原因，达成一致，希望终止合约，则可以发起终止申请。合约终止的过程如下：

1. 向 PalletOne 发起合约终止申请（CTR: Contract Termination Request）。CTR 包含以下信息：合约 ID，合约终止参数，签名。

2. 陪审团收到 CTR 之后，执行合约终止检查，如果满足终止条件，则主陪审员将合约状态修改为终止，并记录到分布式存储中。

智能合约与 PalletOne 内核的交互流程

智能合约在创建后，陪审员在主机上会创建对应的 Docker 镜像和容器，合约程序在容器中执行。合约在执行时通过 gRPC 与陪审员主机的 PalletOne 内核通讯，由 PalletOne 内核再与其他模块和节点通讯。每一个合约的实例就会对应一个 gRPC 连接，PalletOne 内核在调用合约容器之前会构造合约的上下文，然后才发起 gRPC 连接，而合约在执行完毕后返回给 PalletOne 内核之后，PalletOne 内核负责对返回的结果进行签名，然后转发签名后的结果给其他陪审员。

在 PalletOne 提供给智能合约开发人员的 SDK 中，提供了大量需要与 PalletOne 内核进行互操作的接口，包括以下函数：

获得调用的参数

```
GetArgs() [][]byte
```

以 byte 数组的数组的形式获得传入的参数列表 GetStringArgs() []string 以字符串数组的形式获得传入的参数列表。

```
GetFunctionAndParameters() (string, []string)
```

将字符串数组的参数分为两部分，数组第一个字是 Function，剩下的都是 Parameter。

```
GetArgsSlice() ([]byte, error)
```

以 byte 切片的形式获得参数列表。

对合约状态数据的操作

```
GetState()
```

```
PutState()
```

```
DelState()
```

对 PalletOne Token 的操作。

```
CreateAddress()
```

```
GetBalance()
```

```
GetTxHistory()
```

对外部链的操作

```
GetOutChainBalance(string chainId, string address)
```

```
CreateOutChainTx(string chainId, Transaction tx)
```

```
InvokeOutChainFunc(string chainId, string function, string[] args)
```

以上函数都是合约通过 gRPC 调用 PalletOne 内核，然后由内核根据具体函数觉得是对分布式存储的读写还是对区块链适配器的调用。

智能合约的状态

智能合约本身是不会存储任何数据的，业务逻辑处理过程是通过建立好的 gRPC 连接实现和 PalletOne 内核节点的交互。交互过程是通过有限状态机（FSM）来实现的。在 PalletOne 内核和智能合约端都通过有限状态定

义了各自生命周期内所处的所有状态，以及如何在各种状态下响应各种事件和转移到其他状态。智能合约的状态转换如图 5.5 所示。

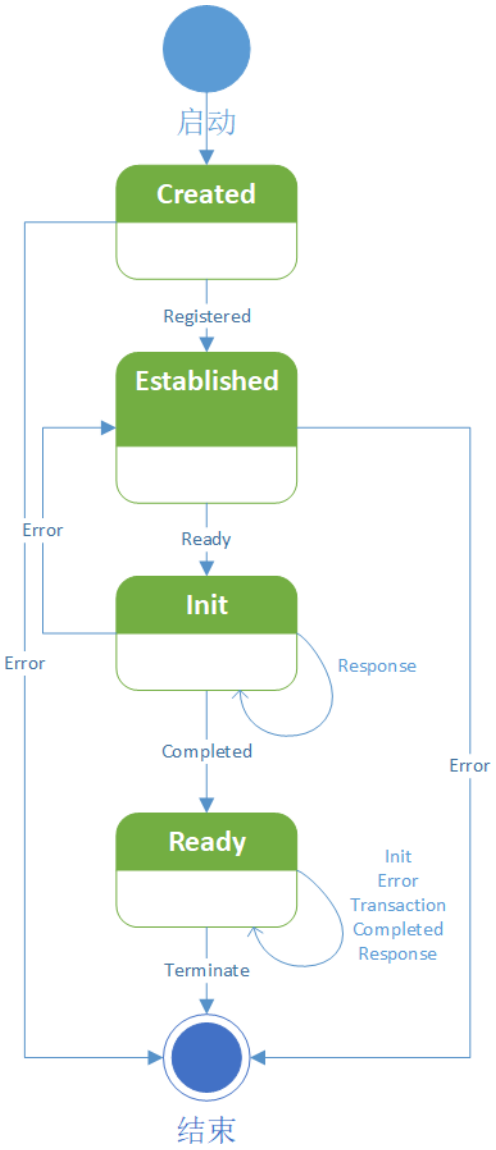

图 5.5 智能合约的状态转换

而 PalletOne 内核的状态转换如图 5.6 所示。

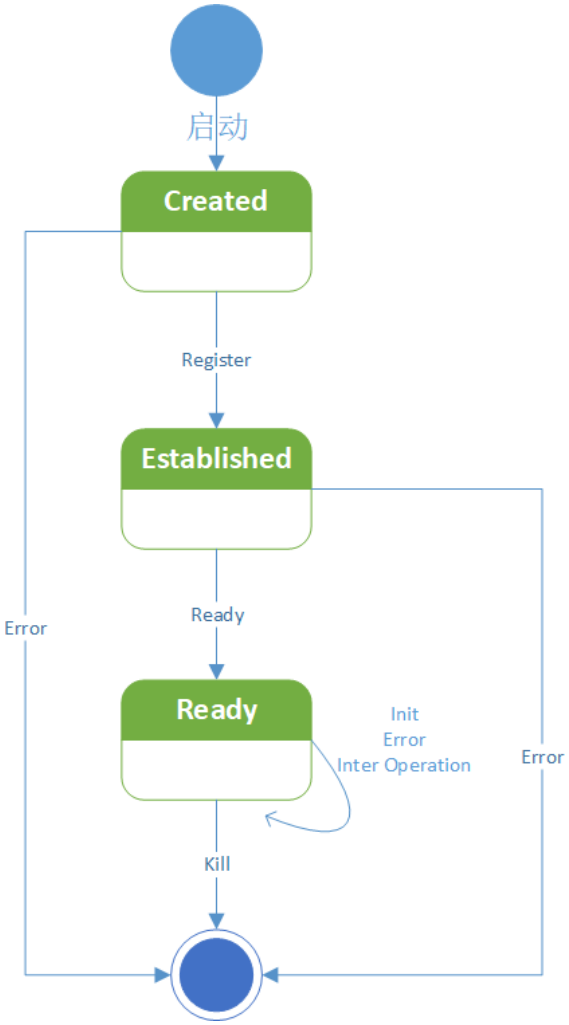

图 5.6 PalletOne 内核的状态转换

适配层

适配器 (Adapter) 是作为 PalletOne 与不同的链交互时的沟通媒介，在不同的链会需要通过不同的操作逻辑实现，所以需要针对不同的链做对应的接口实现供 PalletOne 合约调用。

在对数种主流链比较分析之后，PalletOne 将这些操作定义为下面四个阶段 (stage)，如图 6.1 所示。

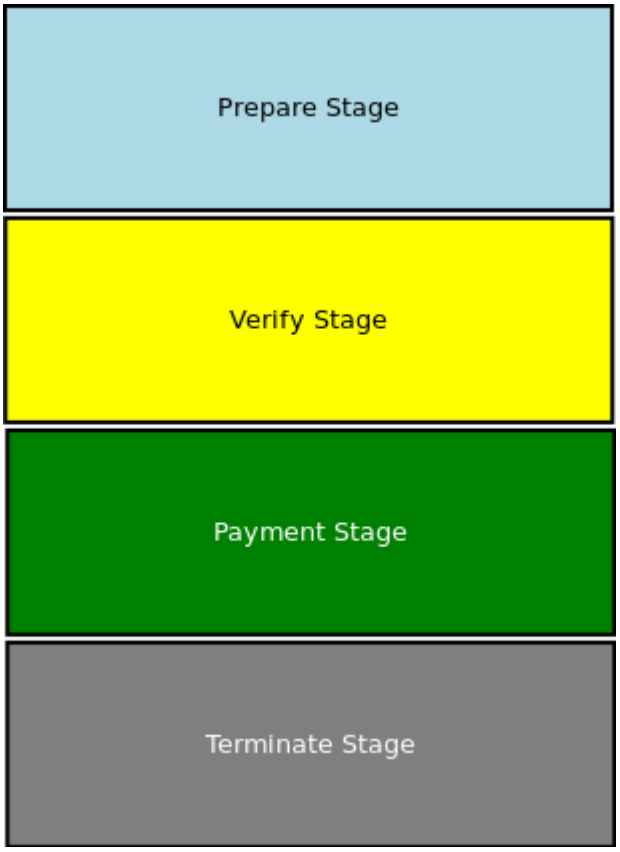

图6.1 PalletOne适配层分段图

1. Prepare Stage：这个阶段用于创建对应的准备工作。以比特币为例这个阶段是创建多签地址；而以太坊为例则是部署合约。

2. Verify Stage：这个阶段是混合阶段，需要 PalletOne 合约与适配器互操作双方的验证机制。以比特币为例，这个阶段是做多重签名；以太坊为例是为合约验证函数调用及公证人验证/投票阶段。

3. Payment Stage：这个阶段是双方对应链的输出阶段，经过 Verify 完成后，请求支付方调用合约时去检查合约逻辑后依照合约结果去输出完成支付请求。

4. Terminate Stage: 这个阶段表示该合约已经完成或是双方无法达成协议后失效时调用的，以以太坊为例此为调用终止合约的动作。

比特币适配器

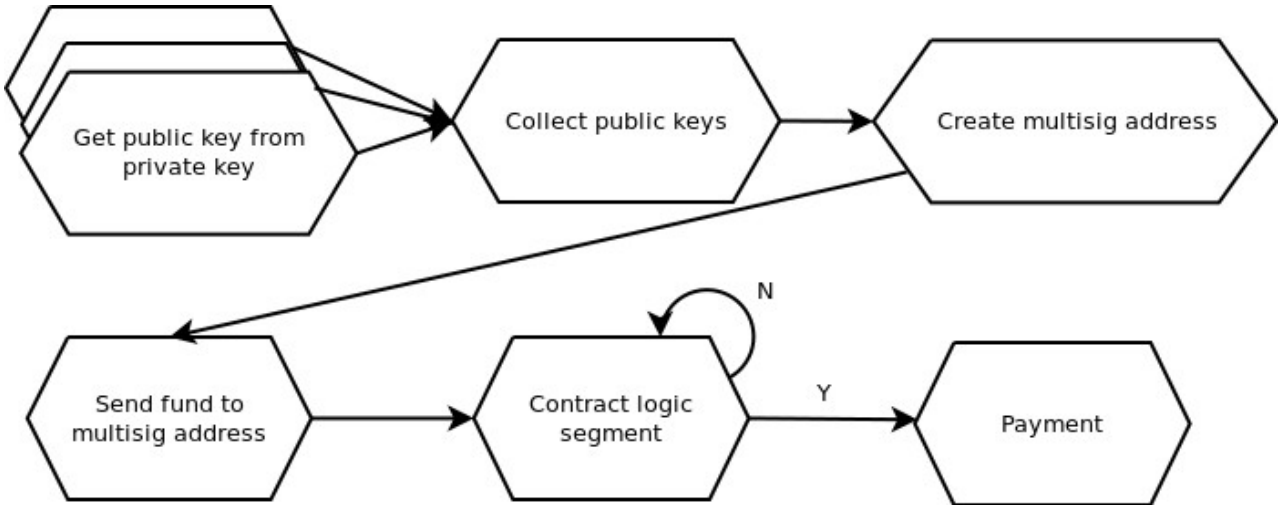

图6.2 比特币适配器

在 PalletOne 中，比特币适配器的工作流程如图 6.2 所示，大致包括以下三个步骤：

1) 发起比特币多签地址创建方收集多方的公钥之后产生一个多签地址，然后将资金放入此多签地址之中；

2) 通过 PalletOne 合约的运行结果决定每一个握有私钥的陪审员是否要执行签名动作，合约逻辑可以是一次性支付或是多次性支付；

3) 如果 PalletOne 合约逻辑判定需要支付,将会把签名交易返回给主陪审员, 广播此交易信息之后即完成支付。

比特币适配器接口

1. GetPublicKey [get public key from private key]：

- 使用私钥运算取得公钥

```
input:
{
  "params":
    { "private_key":
"cUKo4q8MSsDGcTgvmrb8T3D2WDVYQ1DqA3DicnNQc5u9EyDVJ4rA",
    "address": "mrMUE2a5oFNzontQcSDBzGL4ZXdrTg1Sxh"
  }
}
output:
{
  {
    "data":
      { "public_key"
      :
"0351b9d44456a74c8912d3324e75e0600417a519db73112e6c52d21e542e624267",
      "address": "mrMUE2a5oFNzontQcSDBzGL4ZXdrTg1Sxh"
    }
  }
}
```

2. CreateMultiSigAddress [create mutiple signature address]

- 使用 Juror 的公钥产生一次性多签地址
- 产生 n-m 的多签地址

```
input:
{
  "params": {
    "public_keys":
["04a882d414e478039cd5b52a92ffb13dd5e6bd4515497439dffd691a0f12af9575fa349b5
694ed3155b136f09e63975a1700c9f4d4df849323dac06cf3bd6458cd","046ce31db9bdd54
3e72fe3039a1f1c047dab87037c36a669ff90e28da1848f640de68c2fe913d363a51154a0c6
2d7adea1b822d05035077418267b1a1379790187","0411ffd36c70776538d079fbae117dc3
8effafb33304af83ce4894589747aee1ef992f63280567f52f5ba870678b4ab4ff6c8ea600b
d217870a8b4f1f09f3a8e83"],
    "n": 3,
    "m": 2
  }
}
output:
{
  "data": {
    "multisig_addr": {
      "p2sh_address": "347N1Thc213QqfYCz3PZkjoJpNv5b14kBd",
      "redeem_script":
"524104a882d414e478039cd5b52a92ffb13dd5e6bd4515497439dffd691a0f12af9575fa34
9b5694ed3155b136f09e63975a1700c9f4d4df849323dac06cf3bd6458cd41046ce31db9bdd
543e72fe3039a1f1c047dab87037c36a669ff90e28da1848f640de68c2fe913d363a51154a0
c62d7adea1b822d05035077418267b1a1379790187410411ffd36c70776538d079fbae117dc
38effafb33304af83ce4894589747aee1ef992f63280567f52f5ba870678b4ab4ff6c8ea600
bd217870a8b4f1f09f3a8e8353ae"
    }
  }
}
```

3. GetUnspendUTXO [取得多签地址的 Utxo 列表]

- minconf (numeric, optional, default=1), The minimum confirmations to filter
- maxconf (numeric, optional, default=9999999), The maximum confirmations to filter

```
input:
{
  "params": {
    "addresses": ["mkMQUVCtDmc8WVFURQ5ANuy9cRg7vJRQ4d",
"msJ2ktSgDaqs2jnytW1XUcR9WXiNK2pH1M"],
    "minconf": 0,
    "maxconf": 99999,
    "maximumCount": 2
  }
}
output:
{
  "data": {
    "utxo": [
      {
          "txid":
"129a5e3a8f746d332177e6559958251c304832b749df147117f232c1828a11f2",
          "vout": 1,
          "address": "mkMQUVCtDmc8WVFURQ5ANuy9cRg7vJRQ4d",
          "scriptPubKey":
"76a914350a52baeffa008dcd910af6e56e7b36e2d7d86f88ac",
          "amount": 49.89996200,
          "confirmations": 17,
          "spendable": true,
          "solvable": true,
          "safe": true
      },
      {
          "txid":
"6bf1b5672b633123ebbcd562f2e80ae90a8e3c71d3b3e35cdd84d5c4059101fb",
          "vout": 0,
          "address": "msJ2ktSgDaqs2jnytW1XUcR9WXiNK2pH1M",
          "scriptPubKey":
"21026be0e41fa7b42fa3b41c8f15e2f8fa2e2aedde35a6abf4f01d47f30be0b703ceac",
          "amount": 50.00000000,
          "confirmations": 106,
          "spendable": true,
          "solvable": true,
          "safe": true
      }
    ]
  }
}
```

4. RawTransactionGen [产生交易的 raw transaction]

```
input
{
  "params": {
    "inputs": [
      {
        "txid":"id",
        "vout":n,
        "sequence":n,
      }
    ],
    "outputs": [
      {
        "address": 0.001,
        "data": "hex"
      }
    ],
    "locktime": 0,
    "replaceable": false
  }
}
output
{
  "data": {
    "rawtx":
"0200000001fb019105c4d584dd5ce3b3d3713c8e0ae90ae8f262d5bceb2331632b67b5f
16b0000000000ffffffff01000e27070000000017a914800bf99cf4d78e58e86513baa0c
4c79f7a4d70208700000000"
  }
}
```

5. DecodeRawTransaction [显示详细交易信息]

```
input
{
  "params": {
    "rawtx":
"0200000001fb019105c4d584dd5ce3b3d3713c8e0ae90ae8f262d5bceb2331632b67b5f16b
0000000000ffffffff01000e27070000000017a914800bf99cf4d78e58e86513baa0c4c79f7
a4d70208700000000"
  }
}
output
{
  "data": {
    "txid":
"44a081ba701f17eee823f1d1a5a27312dd692e347ad21a56ed80e45434ed9154",
    "hash":
"44a081ba701f17eee823f1d1a5a27312dd692e347ad21a56ed80e45434ed9154",
    "version": 2,
    "size": 83,
    "vsize": 83,
    "locktime": 0,
    "vin": [
      {
        "txid":
"6bf1b5672b633123ebbcd562f2e80ae90a8e3c71d3b3e35cdd84d5c4059101fb",
        "vout": 0,
        "scriptSig": {
          "asm": "",
          "hex": ""
        },
        "sequence": 4294967295
      }
    ],
    "vout": [
      {
        "value": 1.20000000,
        "n": 0,
        "scriptPubKey": {
          "asm": "OP_HASH160 800bf99cf4d78e58e86513baa0c4c79f7a4d7020
OP_EQUAL",
          "hex": "a914800bf99cf4d78e58e86513baa0c4c79f7a4d702087",
          "reqSigs": 1,
          "type": "scripthash",
          "addresses": [
            "2N4vGpsxXQvuKoExz8XVxYwvu5kX5CPk6zP"
          ]
        }
      }
    ]
  }
}
```

6. SignTransaction [签署交易]

```
input
{
  "params":
    { "hexstring
    ":
"0200000001fb019105c4d584dd5ce3b3d3713c8e0ae90ae8f262d5bceb2331632b67b5f
16b0000000000ffffffff01000e27070000000017a914800bf99cf4d78e58e86513baa0c
4c79f7a4d70208700000000",
    "privkeys": ["cT82eQM6ZLuRY8CS2rKZ9XhjvQ8fhCQbV6oTcai7E7WFYbx98amX"]
  }
}
output
{
  "data": {
    "hex":
"0200000001fb019105c4d584dd5ce3b3d3713c8e0ae90ae8f262d5bceb2331632b67b5f
16b00000000474630430220347 11d03a650dc1639736a28475799273039038a24aa9e16c
1bb060f555e2562021f31685371525d155e17a9fe9a008f59c377839d6a100bb876e2ada
d49df347201ffffffff01000e27070000000017a914800bf99cf4d78e58e86513baa0c4c
79f7a4d70208700000000",
    "complete": true
  }
}
```

7. GetBalance [取得多签地址的可用余额]

- minconf (numeric, optional, default=1) The minimum confirmations to filter

```
input
{
  "params": {
    "address": "2N4vGpsxXQvuKoExz8XVxYwvu5kX5CPk6zP",
    "minconf": 3
  }
}
output
{
  "data": {
    "value": 1.2
  }
}
```

8. GetTransactions [取得多签地址相关交易]

- skipping the first [from] transactions for account

```
input
{
  "params": {
    "account": "2N4vGpsxXQvuKoExz8XVxYwvu5kX5CPk6zP",
    "count": 10,
    "skip": 0
  }
}
```

9. ImportMultisig [将多签地址加入钱包观察地址]

- 扫描区块建立相关该地址资讯的 index

```
input
{
  "params": {
    "account": "2N89aUC4JUjXyapvPvc1ivJVcfXpqBR4P2g",
    "rescan": true
  }
}
output
{
  "data":
    { "success": rue
  }
}
```

以太坊适配器

以太坊本身是支持图灵完备的智能合约，此处只是提供一个范例，实现一个最简单的以太坊合约接口，不同类型的以太坊合约需要做对应的外部接口实现以供 PalletOne 合约调用。

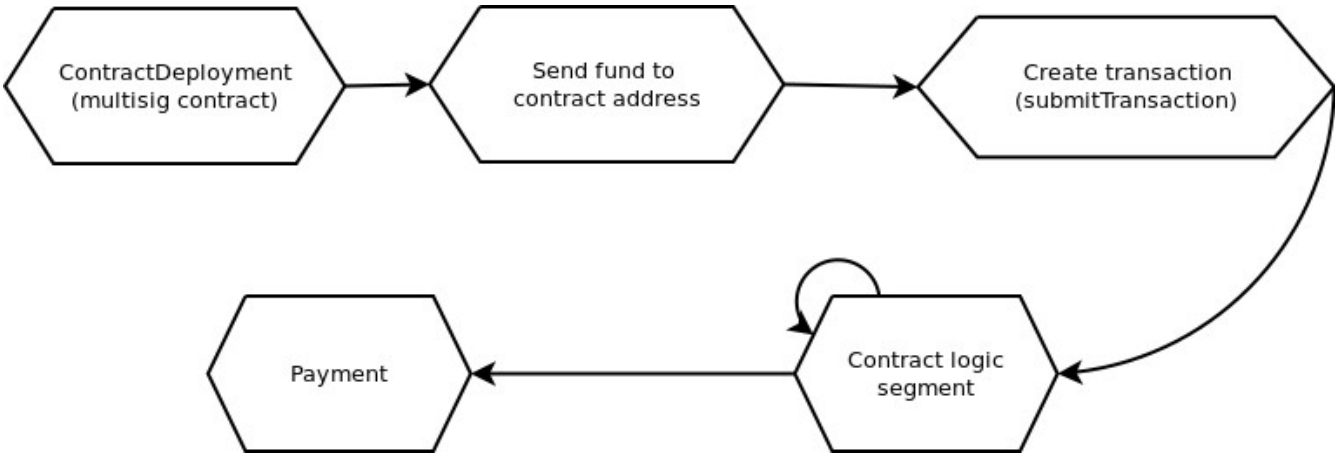

图 6.3 以太坊适配器

PalletOne 以太坊适配器工作如图 6.3 所示，大致可以分为以下三个步骤：

1. PalletOne 调用以太坊接口部署多方签名的 Solidity 智能合约, 部署合约时指定以太坊陪审员列表以及签名策略(交易有效签名数);

2. 将资金放入部署完成的以太坊智能合约的地址后，可以创建合约内的支付合约交易，陪审团会通过 PalletOne 合约逻辑验证此请求支付交易的合理性，如果 PalletOne 合约执行后判定此交易有效，将会调用执行以太坊合约的 confirmTransaction 函数;

3. 对该交易执行签名指令，当签名数满足签名策略时，可以执行 executeTransaction 合约指令去完成以太坊上的链上支付请求。

以太坊适配器接口

1. GetBalance [取得帐户余额]

```
input
{
  "params": {
    "account": "0xf947D41eA54953c212ecCaa4dF5c015821Ba8731"
  }
}
output
{
  "data":
    { "balance":
    1.0042
  }
}
```

2. GetEthereumTransactionByHash [使用哈希取得交易信息]

```
input
{
  "params": {
    "hash":
"0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238"
  }
}
output
{
  "data": {
    "id":1,
    "jsonrpc":"2.0",
    "result": {

"hash":"0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d105
5b",
      "nonce":"0x",
      "blockHash":
"0xbeab0aa2411b7ab17f30a99d3cb9c6ef2fc5426d6ad6fd9e2a26a6aed1d1055b",
      "blockNumber": "0x15df",
      "transactionIndex":  "0x1",
      "from":"0x407d73d8a49eeb85d32cf465507dd71d507100c1",
      "to":"0x85h43d8a49eeb85d32cf465507dd71d507100c1",
      "value":"0x7f110",
      "gas": "0x7f110",
      "gasPrice":"0x09184e72a000",

"input":"0x603880600c6000396000f300603880600c6000396000f3603880600c60003
96000f360",
    }
  }
}
```

3. ContractDeployment [部属合约模板]

- address：为多签需要审核人的地址
- num_of_confirm: 设置需要几个人签署才可以执行交易

```
input
{
    "params": {
        "address": ["0x627306090abaB3A6e1400e9345bC60c78a8BEf57",
"0xf17f52151EbEF6C7334FAD080c5704D77216b732"],
        "num_of_confirm": 2
    }
}
output
{
    "data": {
        "contract_address": "0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f"
    }
}
```

4. submitTransaction

```
input
{
  "params": {
    "address": "0xf17f52151EbEF6C7334FAD080c5704D77216b732",
    "value": 1000,
    "bytes": "data"
  }
}
output
{
  "data": {
    "status": "ok"
  }
}
```

5. getTra·nsactionIds [取得等待确认签名的交易 id 列表]

```
input
{
  "params": {
    "from": 0,
    "to": 10,
    "pending": true,
    "executed": false
  }
}
output
{
  "data": {
    "ids": [1,2,3]
  }
}
```

6. getTransactionById [透过 id 取得交易内容]

```
input
{
  "params": {
    "id": 1
  }
}
output
{
  "data": {
    "destination": "0xf17f52151EbEF6C7334FAD080c5704D77216b732",
    "value": 1000,
    "bytes": "data",
    "bool": false
  }
}
```

7. getConfirmations [取得某交易确认签名的交易]

```
input
{
  "params": {
    "id": 1
  }
}
output
{
  "data": {
    "address": ["0xf17f52151EbEF6C7334FAD080c5704D77216b732"]
  }
}
```

8. confirmTransaction [对某笔交易进行签名]

```
input
{
  "params": {
    "id": 1
  }
}
output
{
  "data": {
    "status": "ok"
  }
}
```

9. executeTransaction [执行交易]

- (任何人都可以触发执行), 确认签名足够时交易将会被执行

```
input
{
  "params": {
    "id": 1
  }
}
output
{
  "data": {
    "status": "ok"
  }
}
```

多签合约模板

以太坊不像比特币一样原生支持多签，而是通过智能合约来实现。以下是以太坊中进行多签的一个开源合约源码，PalletOne 会基于该模板进一步的优化，以满足 PalletOne 在以太坊上的多签需求。

https://github.com/gnosis/MultiSigWallet

示例

BTC 和 ETH 互换(Jury 背书)

陪审团锁定模式

Alice 想用 1BTC 换 16 个 ETH，Bob 正好也想用 16 个 ETH 换 1 个 BTC，但是他们在网络上，并不相信对方，所以他们打算在 PalletOne 上通过 BTC 和 ETH 互换合约进行币种的交换。

具体操作过程如下：

1. Alice 在 PalletOne 的合约市场找到“BTC 和 ETH 互换”合约，填写好要兑换的 BTC 数 1，ETH 数 16，违约押金数 10，合约超时时间 24 小时， 并签名创建该合约。并将该合约发送给 Bob。

2. Bob 收到合约后，检查合约无误，也在合约上签名。合约正式激活，并请求双方各提供 4 个公钥和合约押金。

3. Alice 在钱包中生成 4 个 BTC 的公私钥对和 4 个 ETH 的公私钥对，并将公钥和 10 PalletOne Token 发送给合约。

4. Bob 也是同样的操作，在钱包中生成 4 个 BTC 的公私钥对和 4 个 ETH 的公私钥对，并将公钥和 10 PalletOne Token 发送给合约。

5. 合约在收到 Alice 和 Bob 发来的公钥后，会连同合约对于 4 个陪审员提供的公钥，生成 1 个 BTC 7/12 多签地址和 1 个 ETH7/12 多签地址，并将这两个地址发送给 Alice 和 Bob。

以上是合约准备和签收阶段，下面是合约的执行过程，分两种情形：

- 互换合约正常完成
- 一方履约另一方违约

下面分别介绍这两种情况下合约的处理逻辑：

互换合约正常完成

在双方都产生了公钥并支付合约押金后，Alice 和 Bob 会收到 1 个 BTC 地址和 1 个 ETH 地址。接下来双方和系统进行了如下操作：

1 Alice 将 1BTC 转账到合约生成的多签地址中。

2 Bob 将 16ETH 转账到合约生成的多钱地址中。

3 Bob 检查多签地址发现 Alice 已经转了 1BTC，所以他发起收 BTC 申请，从多签地址转账到 Bob 钱包的 Transaction，并用自己的 4 个 BTC 私钥签名。

4 陪审团在收到 Bob 的收 BTC 申请后，检查 ETH 多签地址，发现 Bob 已经转了 16ETH，Alice 也转了 1BTC 到多签地址，满足互换条件。

5 每个陪审员在检查满足互换条件后，用自己的私钥在 Bob 的收 BTC Transaction 上签名。

6 当陪审团中的 3 个陪审员都签名后，满足了 7/12 的多签条件，多签地址的 1BTC 会转账到 Bob 的钱包。

7 Alice 在得知 Bob 已经转了 16 个 ETH 到多签地址后，也进行同样的操作，将多签地址中的 16ETH 转移到自己的钱包。

8 Alice 在收到 16 个 ETH 后，同时也看到 Bob 已经收了 1BTC，她可以向陪审团发起合约终止申请。

9 陪审团检查两个多签地址，发现币已互换完成，将双方的合约押金各 10 PalletOne Token 转回各自的钱包，并将该合约状态改为终止，合约执行完毕。

上述操作步骤如图 7.1 所示。

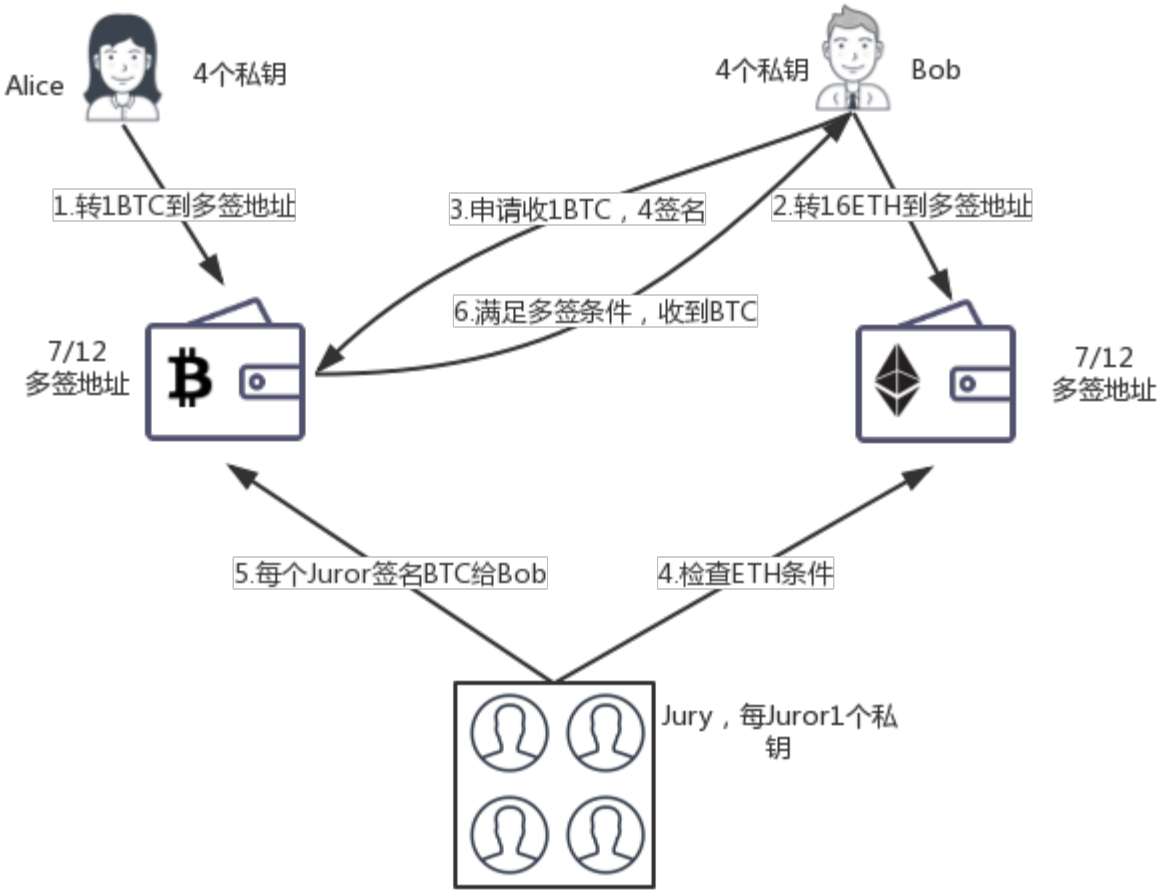

图 7.1 互换合约正常操作流程

一方履约另一方违约

现实世界总是充满各种不确定性，比如币价的变动，导致某一方觉得之前签订的合约导致自己亏损，就会主观上不愿继续执行互换合约，而另一方已经转币到多签地址，合约的不继续执行会导致履约方的损失，所以在该互换合约中必须有押金保证履约方的损失得到补偿。具体操作过程如下：

1. Alice 将 1BTC 转账到合约生成的多签地址中。

2. Bob 反悔了，不愿意将 16 个 ETH 转入多签地址。

3. 过了 24 小时了，Bob 仍然没有转 ETH，于是 Alice 向陪审团申请 Bob 违约，并要求退回 1BTC，并用自己的 4 个私钥签名。

4. 陪审团检查 BTC 和 ETH 多签地址，发现 Alice 已经转账，而 Bob 没有转账，同时时间已经超过了合约规定的超时时间，认定 Bob 违约。

5. 陪审团在退还 Alice 1BTC 的交易上签名，当 3 个及以上的陪审员签名后，满足 7/12 的多签条件，1BTC 会转移到 Alice 的钱包地址。

6. 陪审团将 Alice 和 Bob 的合约押金合计 20 PalletOne Token 转移到 Alice 的地址，作为 Bob 违约给 Alice 的补偿。

7. 陪审团将合约状态修改为终止，合约执行完毕。

BTC 和 ETH 互换(Mediator 背书)

前面提到的锁定陪审团模式，由于 BTC 的多签限制，陪审员数量最多只能到 5 个，而且用户 Alice 和 Bob 持有多个私钥进行签名会很麻烦，于是我们提出了另一种互换模式：Mediator 背书模式。Mediator 背书模式的优点是，Mediator 节点会有比 Juror 节点更高的在线率，可以保证多签钱包的持续存在，这种模式更适合持续时间很长的跨链需要多签的合约。

仍然以同样的示例为背景：Alice 想用 1BTC 换 16 个 ETH，Bob 正好也想用 16 个 ETH 换 1 个 BTC，但是他们在网络上，并不相信对方，所以他们打算在 PalletOne 上通过 BTC 和 ETH 互换合约进行币种的交换。

具体操作过程如下：

1. Alice 在 PalletOne 的合约市场找到“BTC 和 ETH 互换”合约，填写好要兑换的 BTC 数 1，ETH 数 16，违约押金数 10，合约超时时间 24 小时， 并签名创建该合约。并将该合约发送给 Bob。

2. Bob 收到合约后，检查合约无误，也在合约上签名。合约正式激活，并请求双方各提供 4 个公钥和合约押金。

3. Alice 在钱包中生成 1 个 BTC 的公私钥对和 1 个 ETH 的公私钥对，并将公钥和 10 PalletOne Token 发送给合约。

4. Bob 也是同样的操作，在钱包中生成 1 个 BTC 的公私钥对和 1 个 ETH 的公私钥对，并将公钥和 10 PalletOne Token 发送给合约。

5. 合约在收到 Alice 和 Bob 发来的公钥后，查询当前 Mediator 投票前 10 的节点，找到对应的公钥（共 10 个），生成 1 个 BTC7/12 多签地址和 1 个 ETH 7/12 多签地址，并将这两个地址发送给 Alice 和 Bob。

以上是合约准备和签收阶段，下面是合约的执行过程，分两种情形：

- 互换合约正常完成
- 一方履约另一方违约

下面分别介绍这两种情况下合约的处理逻辑：

互换合约正常完成

在双方都产生了公钥并支付合约押金后，Alice 和 Bob 会收到 1 个 BTC 地址和 1 个 ETH 地址。接下来双方和系统进行了如下操作：

1. Alice 将 1BTC 转账到合约生成的多签地址中。

2. Bob 将 16ETH 转账到合约生成的多钱地址中。

3. Bob 检查多签地址发现 Alice 已经转了 1BTC，所以他发起收 BTC 申请，从多签地址转账到 Bob 钱包的 Transaction，并用自己的 1 个 BTC 私钥签名。

4. 陪审团在收到 Bob 的收 BTC 申请后，检查 ETH 多签地址，发现 Bob 已经转了 16ETH，Alice 也转了 1BTC 到多签地址，满足互换条件，对合约的结果进行陪审团内部共识。

5. 当陪审团中大于 2/3 的陪审员都签名后，满足了合约的结果签名阈值，陪审团 Leader 会将执行结果和 BTC Transaction 发送给对应的 Mediator 节点。

6. Mediator 收到陪审团 Leader 发来的请求后，会验证请求的签名和 BTC 多签 Redeem Script，验证成功则签名该 BTC Transaction 并返回给陪审团的 Leader。

7. 当陪审团 Leader 收到 6 个及以上的 Mediator 签名后，便可将多签地址的 1BTC 会转账到 Bob 的交易发布到 BTC 网络。

8. BTC 网络收到满足多签要求的交易，打包交易，Bob 的钱包收到 1BTC。

9. Alice 在得知 Bob 已经转了 16 个 ETH 到多签地址后，也进行同样的操作，将多签地址中的 16ETH 转移到自己的钱包。

10. Alice 在收到 16 个 ETH 后，同时也看到 Bob 已经收了 1BTC，她可以向陪审团发起合约终止申请。

11. 陪审团检查两个多签地址，发现币已互换完成，将双方的合约押金各 10 PalletOne Token 转回各自的钱包，并将该合约状态改为终止，合约执行完毕。

上述操作过程如图 7.2 所示。

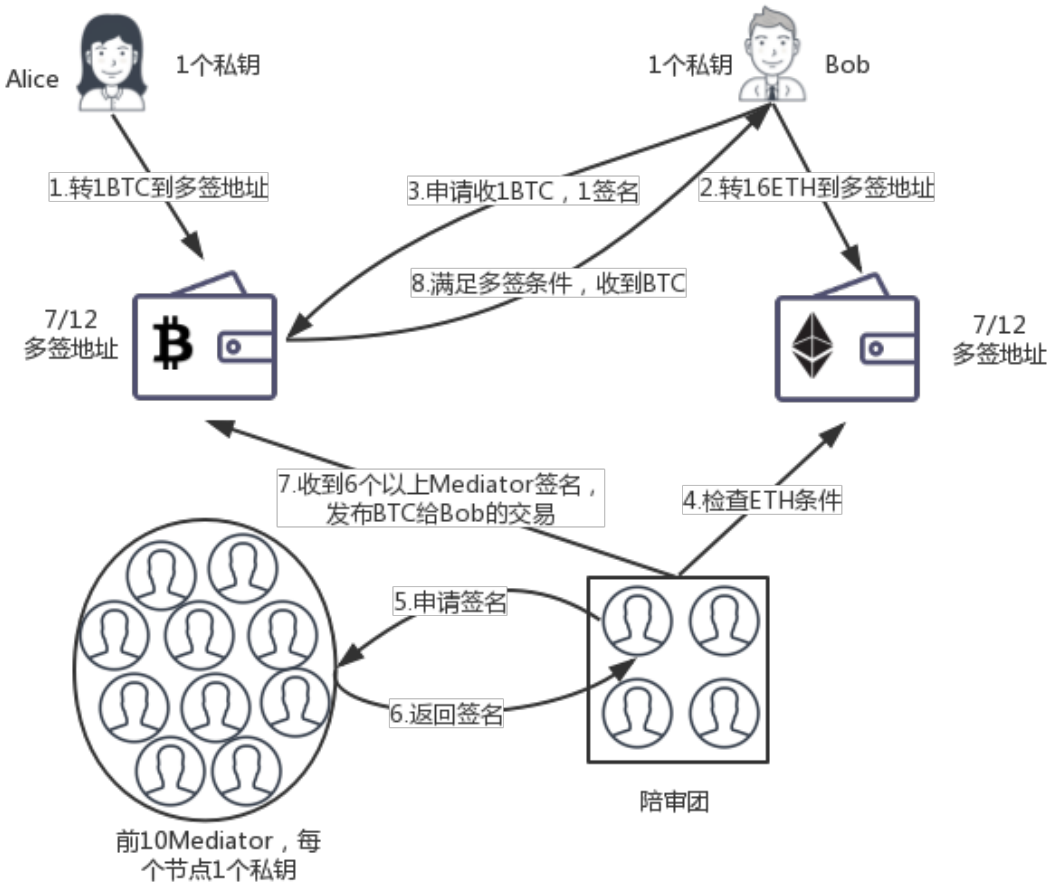

图 7.2 互换合约正常操作流程

一方履约另一方违约

现实世界总是充满各种不确定性，比如币价的变动，导致某一方觉得之前签订的合约导致自己亏损，就会主观上不愿继续执行互换合约，而另一方已经转币到多签地址，合约的不继续执行会导致履约方的损失，所以在该互换合约中必须有押金保证履约方的损失得到补偿。具体操作过程如下：

1. Alice 将 1BTC 转账到合约生成的多签地址中。

2. Bob 反悔了，不愿意将 16 个 ETH 转入多签地址。

3. 过了 24 小时了，Bob 仍然没有转 ETH，于是 Alice 向陪审团申请 Bob 违约，并要求退回 1BTC，并用自己的 1 个私钥签名。

4. 陪审团检查 BTC 和 ETH 多签地址，发现 Alice 已经转账，而 Bob 没有转账，同时时间已经超过了合约规定的超时时间，认定 Bob 违约。

5. 陪审团对该合约的执行超过 2/3 签名后，Leader 将退回 1BTC 的交易发给对应的 Mediator 节点。6. Mediator 验证发来的请求后，在退还 Alice 1BTC 的交易上签名，并返回给 Leader。

6. 当 6 个及以上的陪审员签名后，满足 7/12 的多签条件，1BTC 会转移到 Alice 的钱包地址。

7. 陪审团将 Alice 和 Bob 的合约押金合计 20 PalletOne Token 转移到 Alice 的地址，作为 Bob 违约给 Alice 的补偿 9. 陪审团将合约状态修改为终止，合约执行完毕。

BTC 和 ETH 购买游戏币

某游戏工作室在 PalletOne 上发行了游戏币，用户使用游戏币可以在游戏中购买特殊装备，加速等增值服务。该游戏币的定价是 1BTC=1600 GToken (Game Token)，1ETH=100 GToken，用户可以选择使用 BTC 或者 ETH 进行充值。

1. 游戏工作室在 PalletOne 上基于同质化通证模板，一次性发行 10 亿个 GToken 用于代表游戏中的游戏币。

2. 游戏工作室在 PalletOne 上创建 BTC 和 ETH 购买 Token 的合约，指定 BTC 购买地址，汇率是 1:1600，ETH 购买地址，汇率是 1:100，流转的 To ken 是第一步创建的 GToken。

3. 将这 10 亿 GToken 转移到合约地址，以后由合约负责分配该 Token。

以上是工作室的准备步骤，接下来是用户购买游戏币的过程，如图 7.3 所示：

1. Alice 拥有 BTC，想购买 1600 个游戏币，于是她在 PalletOne 中发起了一个购买游戏币的申请，填写要购买的游戏币数 1600，付款方式是 BTC，自己的 BTC 钱包地址。

2. 合约将根据游戏币的定价，生成订单，告之 Alice 订单 ID，需要转 1BTC，以及充值的 BTC 地址。

3. Alice 用自己的 BTC 钱包转 1BTC 到充值地址，如果是支持转账附言的钱包，请填写订单号。

4. Alice 转 1BTC 后向合约发起收 1600 个游戏币的申请，并填写自己的订单 ID 和转 1BTC 的交易 ID。

5. 合约的陪审员在收到 Alice 的申请后，会检查 BTC 交易记录，和付游戏币记录，确保收到 BTC 而且还没有付游戏币。

6. 合约检查订单的 BTC 钱包地址与交易的输入是否匹配，确保 Alice 不是冒领别人充值记录。

7. 合约将 1600 个 GToken 转移到 Alice 的钱包中。

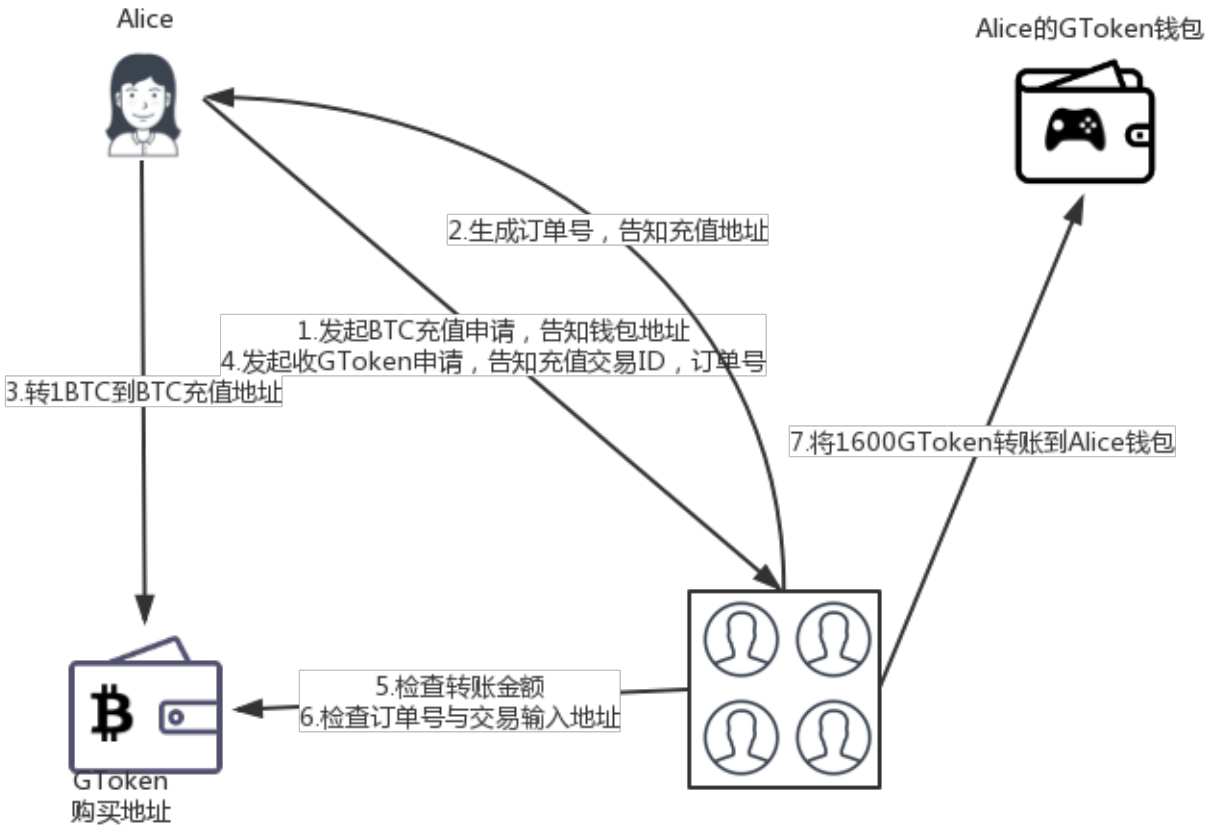

图 7.3 用户购买游戏币过程

Bob 拥有 ETH，他想购买 GToken 时也是采用类似的操作流程，只是合约会去检查的是 ETH 的购买地址余额和交易记录。

以上合约由于不涉及到多签地址，所以不需要锁定陪审团，也就是说，每次交易的执行，并不是由固定的陪审团成员执行的。

PalletOne 通证互换

这是一款基于 PalletOne 发行的 Token 的互换合约，与 BTC&ETH 互换合约不同的是，这里要互换的是 PalletOne 的 Token（简写作 PToken）和用户在 PalletOne 上发行的自己的 Token（假设用户 Token 的名字叫 UToken）。

Alice 拥有 PToken，希望用 10 个 PToken 和 Bob 手里的 100 个 UToken 互换，他们进行了如下的操作：

1. Alice 在 PalletOne 的合约市场找到“PalletOne Token 互换”合约，填写好要兑换的通证是 PToken，数量 10，另一方是 UToken，数量 100，违约押金数 1 PToken，合约超时时间 24 小时，并签名创建该合约。并将该合约发送给 Bob。

2. Bob 收到合约后，检查合约无误，也在合约上签名。合约正式激活，并请求双方缴纳合约押金。

3. Alice 在钱包中将 1 PToken 发送给合约作为押金。

4. Bob 也是同样的操作，在钱包中将 1 PToken 发送给合约。

以上是合约准备和签收阶段，下面是合约的执行过程，分两种情形：

- 互换合约正常完成
- 一方履约另一方违约

下面分别介绍这两种情况下合约的处理逻辑：

互换合约正常完成

在双方都支付合约押金后，Alice 和 Bob 便可进行互换交易。Alice 向这个合约地址转账 10PToken。Bob 看到合约地址已经收到 10PToken 后，直接将 100 个 UToken 转账给 Alice 的钱包。

在确认转账 100 个 UToken 成功后，Bob 向合约申请收款，并附上转 100UToken 的 TxId，并用自己的私钥签名。

陪审员根据 TxId 检查 PalletOne 的网络，发现 Bob 确实转了 100UToken 给 Alice，满足转账条件，所以也对 Bob 的收款交易签名。上述操作步骤如图 7.4 所示。

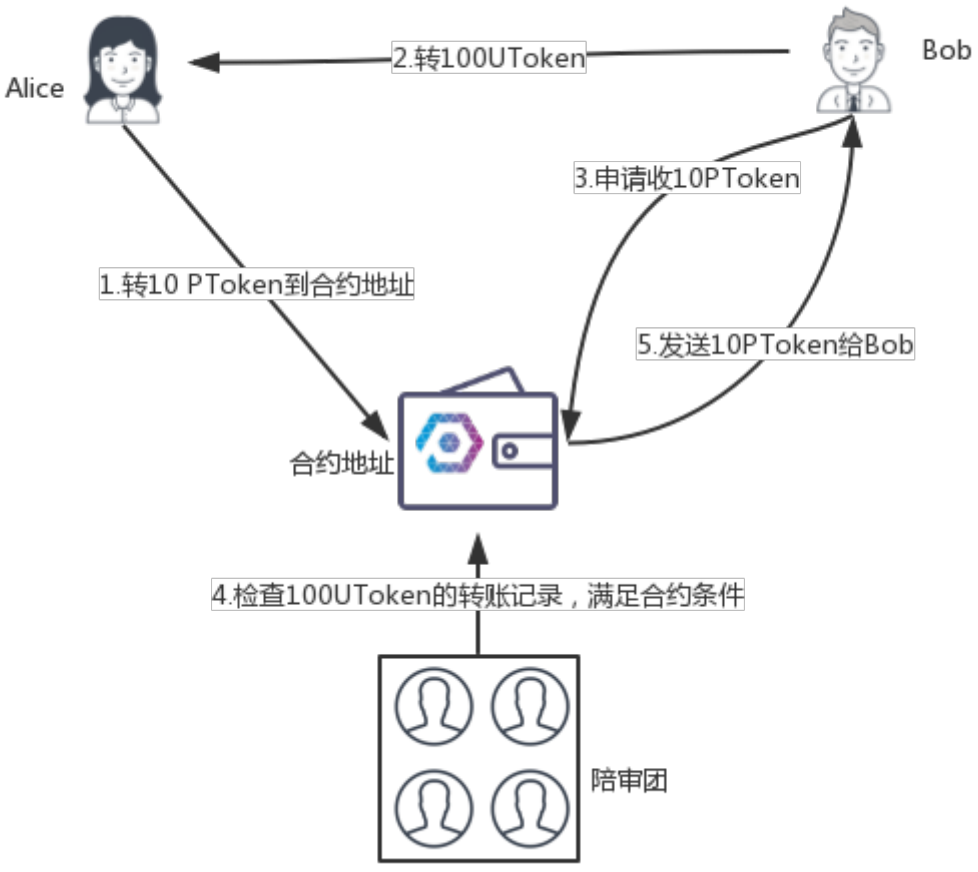

图 7.4 互换合约正常完成操作流程

一方履约另一方违约

现实世界总是充满各种不确定性，比如币价的变动，导致某一方觉得之前签订的合约导致自己亏损，就会主观上不愿继续执行互换合约，而另一方已经转币到多签地址，合约的不继续执行会导致履约方的损失，所以在该互换合约中必须有押金保证履约方的损失得到补偿。具体操作过程如下：

1. Alice 将 10PToken 转账到合约生成的多签地址中。

2. Bob 反悔了，不愿意将 100 个 UToken 转入给 Alice。

3. 过了 24 小时了，Bob 仍然没有转 UToken，于是 Alice 向陪审团申请 Bob 违约，并要求退回 10PToken，并用自己的私钥签名。

4. 陪审团检查合约的余额，发现 Alice 已经转账，而 Bob 没有转账，同时时间已经超过了合约规定的超时时间，认定 Bob 违约。

5. 陪审团在退还 Alice 10PToken 的交易上签名，当 3 个及以上的陪审员签名后，10PToken 转入 Alice 的钱包。

6. 陪审团将 Alice 和 Bob 的合约押金合计 2 PToken 转移到 Alice 的地址，作为 Bob 违约给 Alice 的补偿

7. 陪审团将合约状态修改为终止，合约执行完毕。

去中心化交易所

在去中心化交易所示例中，存在 PalletOne 陪审团、交易所服务器（负责交易网站的运营，KYC、AML，充币、提币、撮合等的见证）、交易双方 Alice 和 Bob 这 4 个角色。交易所服务器虽然是中心化的，但是最终交易的达成，以及交易数据的记录都是去中心化的，避免了传统中心化交易所的各种不透明操作以及交易所账户被盗的情况发生。

以 XToken（PalletOne 发行的某种通证）和 BTC 的币币交易为例。

陪审团持有 6 个公私钥，交易所服务器持有 6 个公私钥，加上用户的 1 个公私钥，组成一个 10/13 的多签地址，由于每个用户提供的公钥不同，所以每个用户的充币地址也是唯一而且一一对应的。用户在该去中心化交易所中可以进行以下操作：

1. 充币

1) Alice 想用自己手里的比特币在去中心化交易所中换成 XToken，于是 Alice 将自己的比特币钱包公钥发给合约，由合约生成一个多签地址。

2) Alice 用比特币钱包向多签地址转入 1BTC。

3) 交易所服务器在扫描到该钱包充值成功后，会向合约发起发放 BTC-Token（这是一种在 PalletOne 上发行的 Token）的调用，由合约向 Alice 的 Pa lletOne 地址发送 1 个 BTC-Token。该 Token 是与充值的币种及面值一一对应的。

2. 挂出买单

1) Alice 想以 1BTC 换取 1000XToken，于是挂出买 XToken 的订单，1BTC-Token 进入合约的账户中。

2) 交易所服务器也会记录该买单，并在网站进行相关展示。

3. 挂出卖单

1) Bob 也想用自己手里的 1000XToken 卖掉，换成 BTC，于是挂出 1000XToken 的卖单，同时将 1000XToken 转入合约的账户中。

2) 交易所服务器提供了撮合服务，发现 Alice 和 Bob 的买卖单匹配，于是向合约发起匹配调用。

3) 合约检查匹配双方的条件，发现条件满足，于是将 1000XToken 转入 Alice 的地址，将 1BTC-Token 转入 Bob 的地址。

4. 撤销买卖单

1) 如果 Alice 挂出卖单后一直没有成交，那么她可以向合约发起撤单申请。

2) 合约检查该笔订单的状态，如果满足撤单条件，则将该订单状态修改为撤销。撤销后的订单不可能被撮合匹配。

5. 提币

1) Bob 在卖出了 XToken 后获得了 1BTC-Token，他向合约提出了提币申请，并提供了自己的比特币地址和 1BTC-Token。

2) 合约销毁 1 个 BTC-Token，将多签 BTC 地址中的 1BTC 转入 Bob 的比特币地址。

结论

PalletOne 既是一个跨链协议，更是一个高性能的“超级公链”。PalletOne 以数字货币抽象、合约抽象和 UTXO 抽象等接口的形式，将所有区块链底层封装到适配器中，对上提供统一的接口。而 PalletOne 虚拟机为 Java、C++等常用编程语言提供了安全稳定的智能合约运行环境，开发人员不用关心区块链底层的细节，就可以使用自己常用的开发语言编写跨链的区块链应用，使得一个应用可以同时在多个链上运行。同时 PalletOne 独创的陪审团机制以及 DAG 数据存储+DPOS 的 Mediator 见证人机制使得合约执行和数据存储都并行处理，从而实现了一个高性能的“超级公链”。

与其他的跨链项目相比，PalletOne 具有更高的性能和通用性。通过 PalletOne 多利益关联方可以互利共赢，共建一个稳健的生态系统。

随着越来越多的公链上线，不同的应用只能运行在不同的链上运行，信息和价值的割裂给区块链应用的普及带来了严重的不便，跨链的价值转移成为了区块链大规模应用的刚需。但是目前跨链技术并没有完全成熟，比如 Cosmos、Polkadot，ArcBlock、万维链等跨链产品都是因为跨链而诞生并各有特点。PalletOne 与众不同之处在于，PalletOne 不仅仅可以跨所有的链，而且在性能考量、通证设计、开发者友好性上都做了更多的优化设计，更容易建立良好的区块链应用生态。

PalletOne 本身只是一个区块链跨链平台，基于该平台可以很方便的建立一个去中心化交易所应用，与现在的 0x、EtherDelta、Kyber.Network 等常见去中心交易所相比，PalletOne 上的去中心化交易所除了同样具有交易透明、资产安全等特点外，还可以做到更多的币种支持、更快的处理速度，树立下一代去中心化交易所的标杆。

PalletOne 的上述优势将随着越来越多的参与方的加入而更突出，该生态系统也将变得更稳健。

词汇表

词汇	解释
BLS	一种阈值签名算法
DAG	有向无环图，分布式账本技术的一种，实现了数据的并行写入。IOTA、ByteBall、TrustNote 等区块链项目都应用了该技术。
DKG	分布式密钥生成
Jury	陪审团，分配到具体合约实例的局部共识团体，由多个陪审员组成。整个 PalletOne 中存在很多个陪审团。
Juror	陪审员，合约的实际执行者，一个陪审员可以同时参与多个陪审团。
Mediator	调停中介，DPOS 产生，由 21 个超级节点组成。
Packet	包，DAG 中的一种稳定的数据结构，在包中的单元都是稳定不会因为后续单元的插入而修改。
Unit	DAG 中的基本数据单元，一个 Unit 包含一笔交易，可以引用多个父 Unit，也可以被多个子 Unit 所引用。
UTXO	未花费的交易输出，比特币中提出的一种记账模型
VRF	Verifiable Random Function，可验证的随机函数