DEC

生态协同 开启颠覆式未来

# DEC chain 项目白皮书

# 去中心化生态协同系统

## Decentralized Ecosystem Cooperation—DEC

**DEC 系统基于去中心化任务协作平台（DTCP），并且可以在这个平台上衍生出丰富的生态。 例如，Digital Asset Project Incubator –DAPI 数字资产孵化器。**

## 项目背景

在传统的协作系统中，需要一个第三方组织作为信息中介，数据保存于中心化的服务器设备中。从可用性角度上看，传统系统存在着中心故障无法使用的问题。从安全行角度上看，传统系统存在数据被非法篡改，无法保证信息的真实性的问题。

区块链技术也被称为分布式账本技术，是一种分布式互联网数据库技术，其特点是去中心化、公开透明、不可篡改、可信任。区块链中存储的每一条数据，将会广播存储至区块链中各个设备节点中，因此，区块链的每个设备节点中存储全量的、一致的数据。

智能合约是一种旨在以信息化方式传播、验证或执行合同的计算机协议。智能合约对接收到的信息进行回应，它可以接收和储存价值，也可以向外发送信息和价值。区块链技术为智能合约提供可信的执行环境。区块链和智能合约的结合，实现了可以在没有第三方的情况下进行可信交易，并且这些交易可追踪且不可逆转。

我们设计了一种基于区块链的分布式生态协同系统，它具有去中心化的特点。该系统中设计发明的协同合约引擎，从安全性、可用性、便捷性等方面对传统系统进行改进，有效解决了传统协作系统中的问题。

# 去中心化任务协作平台

Decentralized Task Collaboration Platform--DTCP

基于区块链的分布式任务协作系统的整体架构承载了八个角色分工 :分布式自治组织、管理员、组织成员，投票人、任务发布人、任务执行人、仲裁陪审团的律师和法官。系统的主要功能是实现在分布式环境下，跨地域跨时间地对任务进行劳务分工协作，包括从自治性组织管理、子基金设立，到每个任务的创建、执行、审查、结算和评价的整个业务流程，以及区块链数据存储方法和协同合约的编写和执行方法，还有分布式协同写作超文本的实现方法。

## ➢ 分布式协同系统的功能、流程和规则

基于区块链的分布式协同系统的系统架构如图 1 所示：

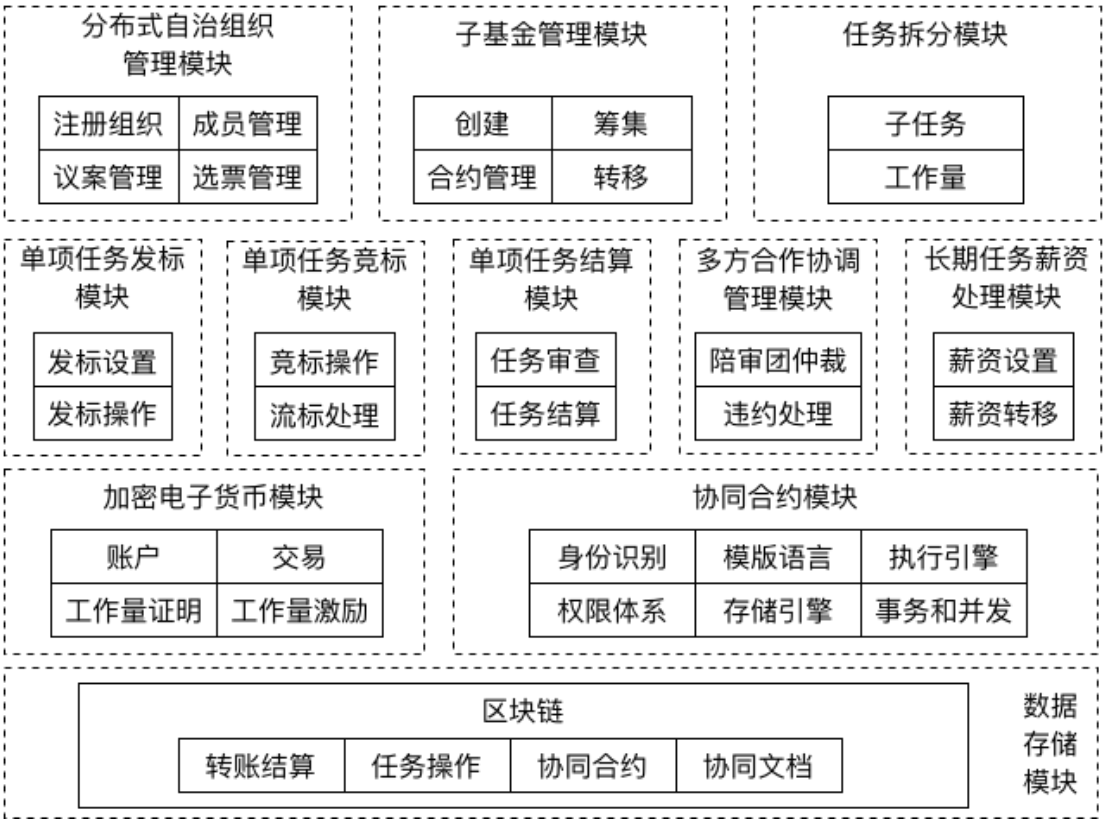
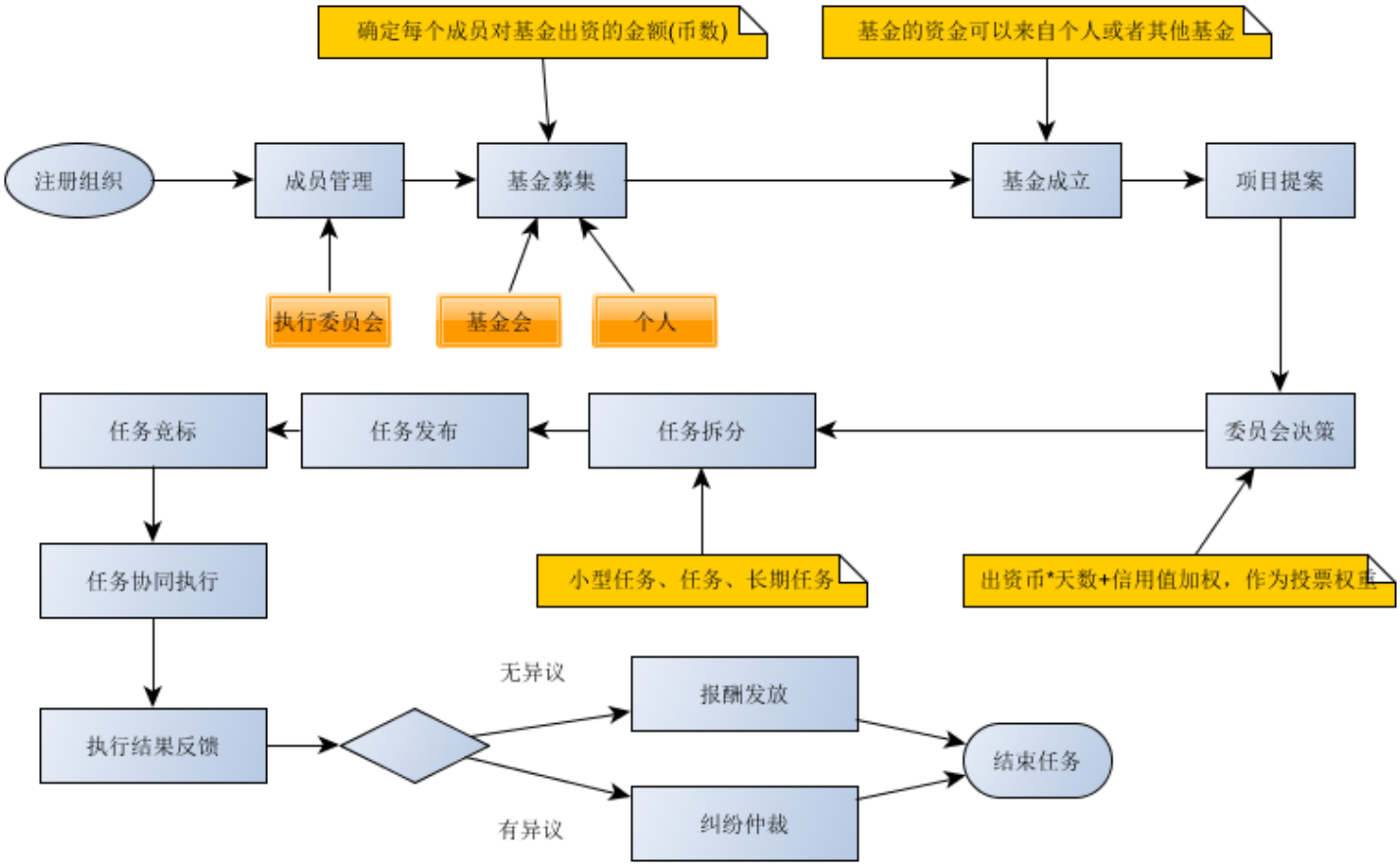



图 1




基于区块链的分布式协作系统，包括如下模块：

1、**分布式自治组织管理模块**，提供了一种对各种提议作出决策的在线民主治理机制，以便于可以综合处理各种可能的管理问题。

这个管理模块是去中心化的，通过民主决议形式投票来决策。

每个分布式自治组织对应现实中的团体，可以是松散型组织，如社区、社团、协会等，也可以是法人，如公司、企业等。

分布式自治组织申请注册的该组织的系统管理员，管理员有权限开通该组织的任务发布成员和投票成员。任务发布成员可以发布待处理的任务，并参与任务处理结果的审查和评价。

投票成员可以新建待表决的议案，并参与该组织内议案的投票。新建议案需要支付一定的加密电子货币作为保证金，参与投票的成员也需要支付一定的加密电子货币作为保证金。投票成员的选票权重和他支付的保证金是正比关系。保证金激励尽量作出好的选票，不好的选票会使已支付的保证金价值降低，好的选票

可以额外获取加密电子货币。投票过程采用匿名计票，系统计算投票结果，但不展示每个投票成员的选票内容。议案的投票结束后，投票结果自动触发协同合约执行，同时系统将自动返还成员所支付的加密电子货币。

2、**子基金管理模块**，提供了一种基于协同合约的，为众多的分散的小委托人进行资产管理的机制。

每个自治组织可以创建多个基金，每个基金对应一个或多个协同合约。协同合约的源代码经过审查后，在系统中进行部署。

自治组织的成员可以通过向协同合约地址发送加密电子货币的方式来参与基金的筹集。自治组织成员都可以花费一定的加密电子货币来发起任务，并使用一部分子基金中的加密电子货币用于该任务，如果任务进行投票后被批准，这些加密电子货币会发送到另外一个表示项目任务的协同合约中，当任务完成并审查后，协同合约中的加密电子货币可以转移到任务执行人的账户中,使子基金用于促进任务的持续进行。

3、**任务拆分模块**，提供了任务内容的提交和拆分功能。

每个自组织成员可以在系统中提交一个任务的描述、规格、验收标准和加密电子货币量的信息。

一个任务必须从属于一个子基金。

任务可以拆分为一个或多个子任务，指定每个子任务占任务总工作量的比例。

任务提交后，自动创建该任务对应的协同合约账户，并从子基金转入加密电子货币，按照子任务的工作量比例分配每个子任务的加密电子货币数量。自治组织成员对该项目进行立项投票，如果投票后被拒绝，自动撤销该项目的合约账户，并返还加密电子货币给子基金。如果投票通过，任务可以进行发标操作。

4、**单项任务发标模块**，可以对子任务进行集中或单个分别进行发标操作。

发标操作，需指定招标有效期，任务完成可获得的加密电子货币数量，任务完成日期，任务违约需补偿的加密电子货币数量。

同时，需设定任务流标时，是否自动调高任务的加密电子货币数量，并设定调整幅度。

5、**单项任务竞标模块**，任务执行人对单项任务进行反向竞标。

在招标有效期内，协同合约自动执行，由竞价低的执行人中标。如果在有效期内无执行人参与竞标，协同合约按照发标时的设定，判定是否自动调高加密电子货币，以吸引执行人竞标。

竞标后，协同合约自动确定该任务的约定完成日期，任务违约时需要付出的补偿。

任务执行人需按照补偿量来发送等量的加密电子货币到合约账户，来作为任务执行的押金。如果任务完成并审查通过后，该押金退回任务执行人账户。如果任务执行违约，该押金作为补偿转移到子基金。

6、**单项任务结算模块**，在任务完成日期之前，执行人提交结果，由任务发布人审查结果。

如果任务发布人确认结果通过，则自动触发协同合约，向任务执行人的账户转移加密电子货币。任务发布人可以提交对执行人的评价，执行人所有的评价可公开查询展示。

7、**多方合作协调管理模块**，处理任务发布人对执行人的结果审查不通过的情况，如下图 2。

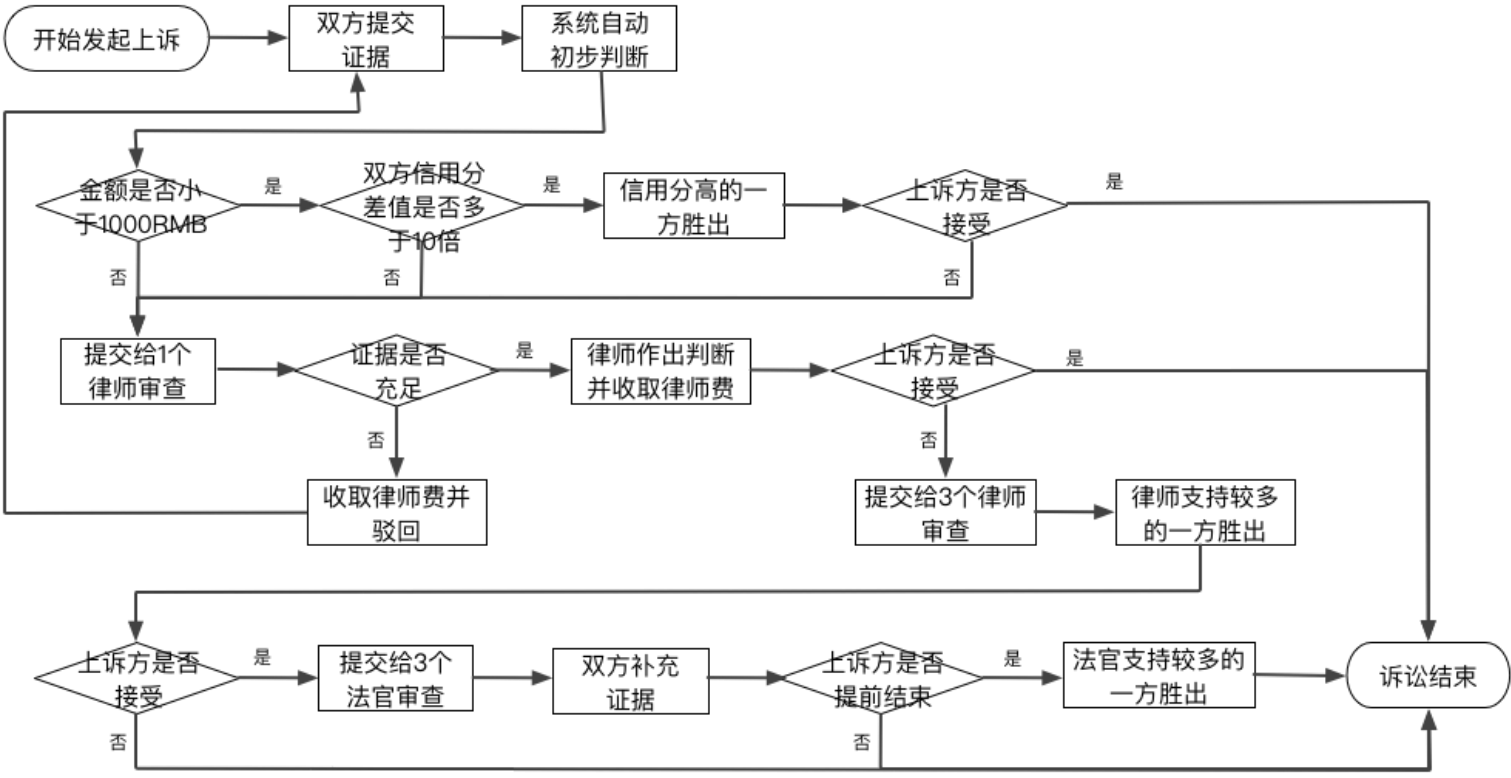


图 2

对于双方发生合同纠纷的情况，分两种情况。

◎合同金额小于 1000 元等值 RMB 的情况下，并且双方信用分差 10 倍以上，则系统自动支持信用高的一方。

◎合同金额大于 1000 元等值 RMB 的情况下，系统引入两个陪审团的角色：律师和法官。

系统首先给出自动仲裁（支持信用分高的一方），如果另一方反对系统仲裁结果，则合同总金额的 8%（或者任务接受人的信用分）作为律师费，支付给担任律师角色的参与方。律师负责审核阅读双方提交的证据，并给出初步的意见。

如果律师支持上诉一方，认为证据充分，则把上诉和证据提交给法官团；如果律师认为上诉证据不足，则直接收取上诉人 4%的合同总金额或者 1 个信用点作为律师费。律师在对于案件进行证据收集、分析的整个过程，公开在区块链上，所以实际上无法在没有履行好自己义务的情况下就直接收取律师费，并且拒绝提交案件。进一步的，上诉人如果对于 A 律师的处理结果不满意，可以继续以 3 个信用分作为质押，选择 3 个律师对于 A 律师的履职进行复核。

这里分几种子情况:

（a）三个律师都认为 A 律师的履职过程没问题，则 3 个律师直接每人分到 1 个信用点 ，来自上诉人。

（b）三个律师 2:1 支持 A 律师 ，则支持 A 律师的 2 个人分别获得 1 个信用点，剩下一个点退回上诉人。

（c）三个律师 2：1 或者 3：0 支持上诉人， 认为 A 律师的处理错误，则支持上诉人的 2 个律师从 A 律师获得 2 个信用点，上诉人获得之前上诉的案件的仲裁支持，获取之前应得的全部利益，以及之前被 A 律师扣除的费用返还。

如果上诉人依然不接受复核结果，可以直接以 6 个信用点作为质押，直接提交上诉到法官团（由 3 个高信用点的志愿者组成，每个参与人的角色固定，信用点超过 100 的律师可以转为法官），由法官团中的一个代表，再次对合同纠纷进行预审，并且要求补充证据；如果上诉方最终认为自己证据不足，最终获胜机会不大，则可以以一个信用点的代价，结束上诉。

否则，经过补充证据，3 个法官对案件进行终审。上诉方获胜则获取合同的应得收益，并且败诉一方扣除 6 个信用点平分给 3 个法官。如果有一个法官和其他两个法官作出相反的判断，则这个法官给另外两个法官每人 1 个信用点。

在整个生态中，信用点是由任务清算的手续费转化而来的，并且在律师、法

官中零手续费的传输。

在三个律师和三个法官进行仲裁的过程中，我们需要一个加密算法，使得只有三个人都提交结果之后，才能揭晓投票结果，这个要求可以使用 Secret_sharing 加密算法来完成。仲裁开始之前，由系统生成一个 RSA 公私钥对 pk，公钥分发给 3 个投票人，私钥 sk 分成三份 s1，s2，s3 ，分别给 3 方，每人一个，用参与人的公钥 pkp 加密后发送，参与人可以用自己的 psk 私钥解密这个三分之一的私钥。提交仲裁结果的时候，仲裁人用公钥 pk 加密投票结果，然后发送到区块链，只有当第三个参与人提交结果的时候，凑齐 3 个秘钥 s1，s2，s3，可以获得 RSA 私钥 sk，从而解密投票结果。由于 RSA 公钥对同一段数据加密，结果是不同的，所以避免了预先知道投票结果的可能。

8、**长期任务薪资处理模块**，是针对执行时间较长的任务，按照自然月份向任务执行人转移电子货币的操作。

任务执行人提交阶段性结果；发布人对阶段性结果进行审查，如果审查通过则自动执行协同合约，向执行人转移加密电子货币。如果审查不通过则进入多方合作协调管理模块。

9、**加密电子货币模块**，包括了账户、交易、区块的创建和打包、工作量证明机制。

账户采用 UXTO 模式，只记录交易本身不记录交易结果，可以追溯货币的交易历史。在基于 PoW 的工作量证明机制下，获得记账权的节点产生新区块并打包交易。

10、**数据存储模块**，包含区块链和文件存储两部分。

区块链中记录的信息有：加密电子货币的交易，任务的发标、竞标、结算、评价。区块链的数据结构由数据块链和数据区块组成。

每个数据区块记录了：神奇数、区块大小、数据区块头部信息、记录计数、交易和任务详情。其中的数据区块头部结构中记录了：版本号、前一个区块的记录、Merkle 树的根值、时间戳、目标特征值、随机数。其中的交易和任务详情中记录了加密电子货币结算，也包含任务操作的记录：任务发标、竞标、结算、仲裁的记录信息。

文件存储部分，提供了协同合约、任务发标和竞标结算的内容和文件的存储

空间。文件存储部分由存储节点、存储集群、存储联盟构成。存储节点是运行数据库服务器和相关软件的机器或一组紧密链接的机器。一组存储节点可以相互连接以形成一个存储集群。存储群集中的每个存储节点都运行相同的数据库软件。一个存储集群是一个逻辑数据集合的存储单位。一个存储集群有部分机器来完成集群监控等任务。每个存储集群由一个人或组织管理控制。一个联盟是一个拥有一个或多个集群的组织，并且由该组织运营。所有联盟的存储集群组合在一起，组成了整个系统的文件存储空间。存储部分按照数据分片规范，保存在不同的存储集群中。数据操作中使用事务机制保证数据完整性。在任务竞拍、任务仲裁、加密电子货币结算中使用事务，事务可以在跨存储集群和存储联盟中使用。

## DEC带来的生产关系变革

The revolution of production relations via DEC

**参与**

—用户自愿和独立的参与到一个松散的任务中

**协作**

—用户进行协同工作，为目标项目贡献价值

**合作**

—用户公平便捷的获得约定收益报酬

**分布式的**

—通过在更大网络上进行增殖，开始在网络上进行传播

**去中心化的**

—稳定可靠、不会宕机，低成本

**自治的**

—自治代理、智能程序、不断进步的人工智能和算法，将提供可自我维持的运作

**免费、稳定的管理系统**

—自带免安装、免费、拿来即用的新一代稳定管理系统。

自上而下的传统组织


One legal entity
Employment contracts

去中心化自治组织

No centralized entity
No employment contracts

## 应用场景
Applications

传统的在线任务发布系统中， 较少有软件开发类的任务， 这类任务复杂度高、协同困难、评估困难， 软件研发工作一般通过jira这类安装在企业内部的专用软件来进行协调管理，而jira并没有价值传输的功能，同时也不可能让不认识的多方协同完成同一个任务。

DEC系统中通过智能合约提供了jira的软件研发管理功能，同时提供了基于代币的任务悬赏，可以作为去中心化自治组织(DAO)的高效的生产力工具。

由于DEC系统的免费、可靠、安全、易用， 将在区块链经济时代中的同类产品中获得较大的竞争优势，在这个新的市场中占据领先地位。

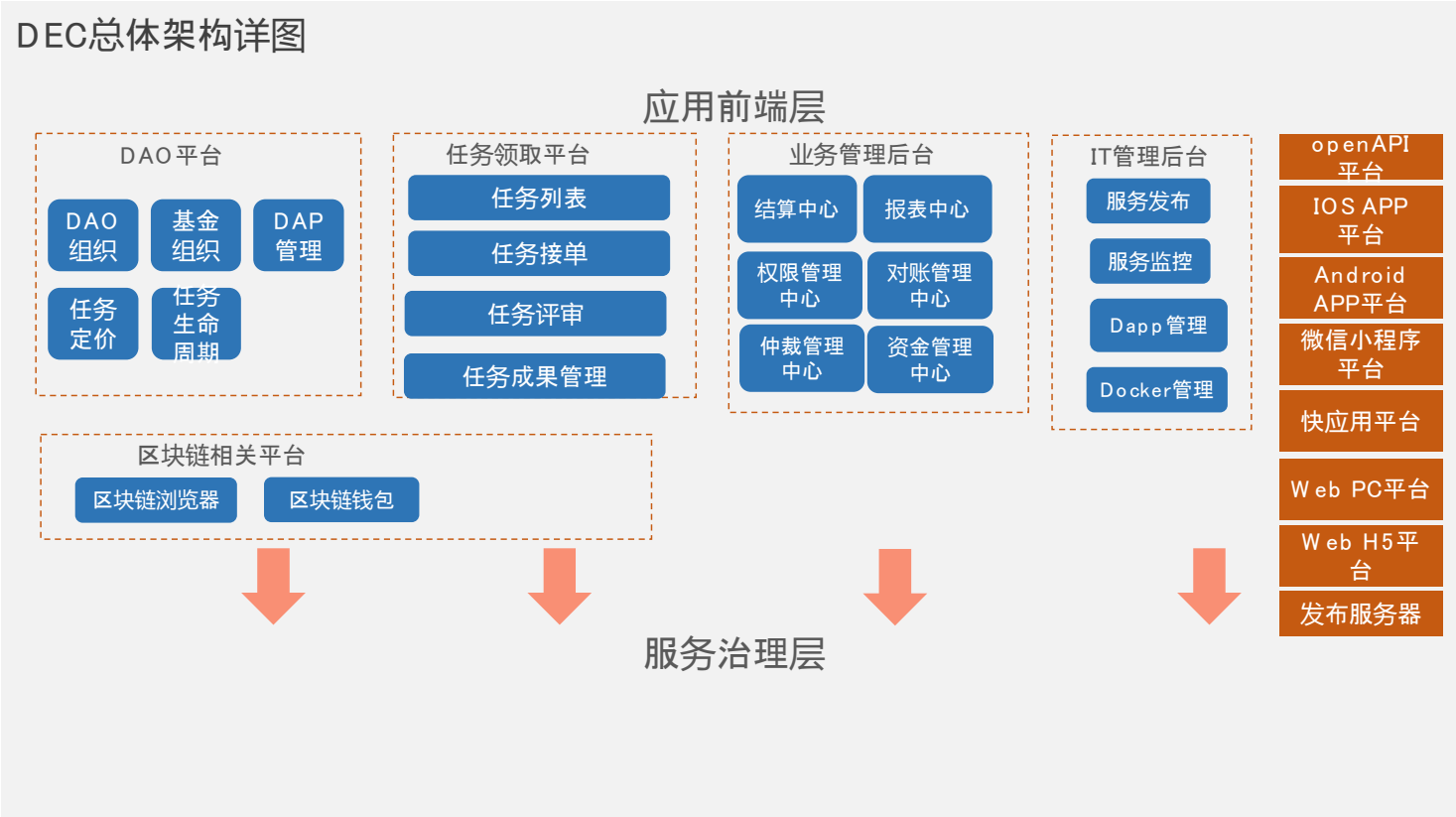

注意：

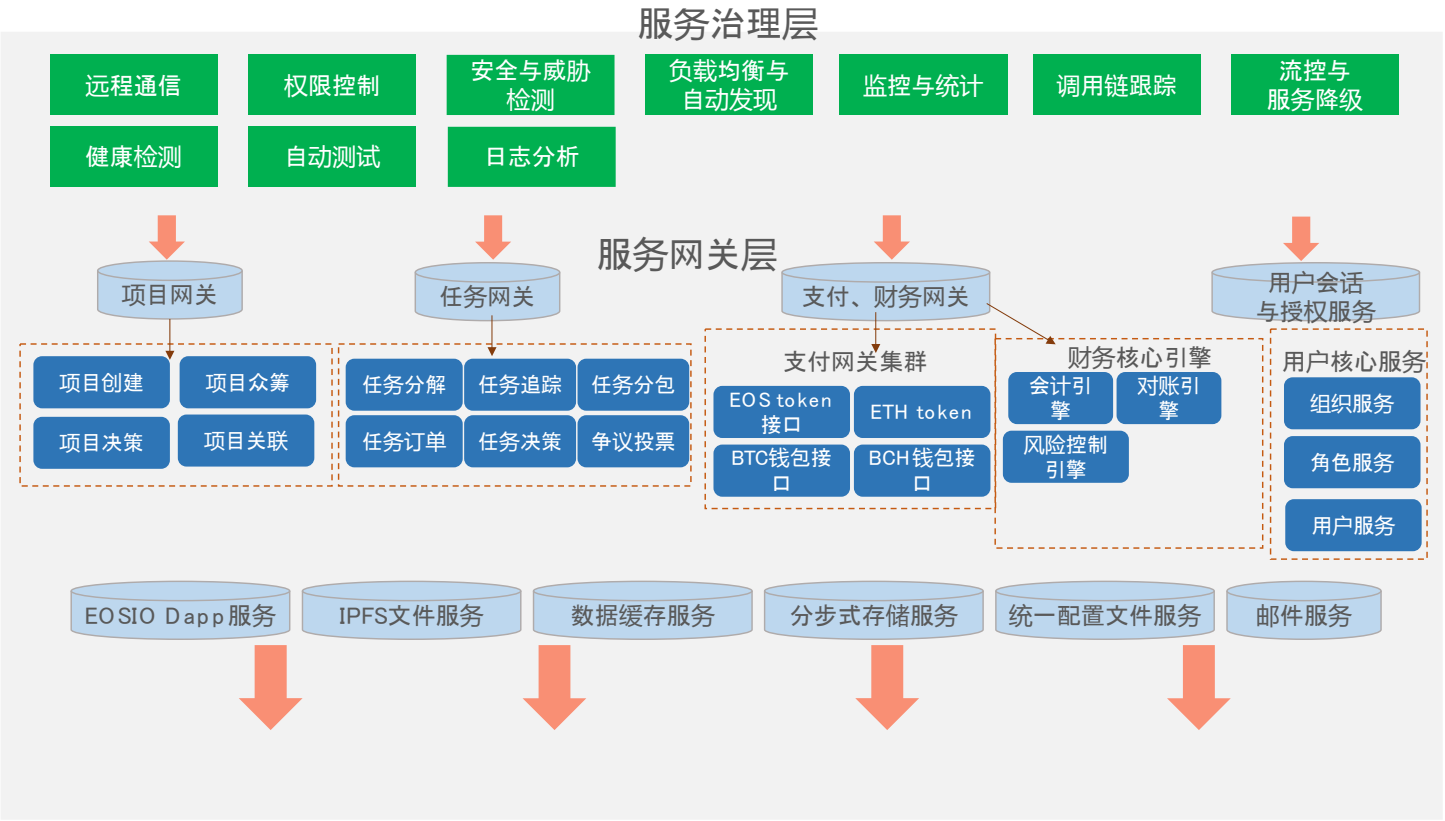
DEC应用场景可以从区块链开发延伸到 ICO孵化流程一体化以及P2P借贷等领域.....

DEC拥有非常好的延展性

## 技术架构

- 基于 EOS 平台的架构




服务治理层
远程通信
权限控制
安全与威胁检测
负载均衡与自动发现
监控与统计
调用链跟踪
流控与服务降级
健康检测
自动测试
日志分析
服务网关层
项目网关
任务网关
支付、财务网关
用户会话与授权服务
项目创建
项目众筹
项目决策
项目关联
任务分解
任务追踪
任务分包
任务订单
任务决策
争议投票
支付网关集群
EOS token接口
ETH token
BTC钱包接口
BCH钱包接口
财务核心引擎
会计引擎
对账引擎
风险控制引擎
用户核心服务
组织服务
角色服务
用户服务
EOSIO Dapp服务
IPFS文件服务
数据缓存服务
分步式存储服务
统一配置文件服务
邮件服务


数据层
数据中心
EOSIO-database
EOSIO-wallet
IPFS
Spark
ELK
MySQL集群
Redis集群
etcd集群
Kafka集群

## ➢ 基于 Decchain 公链的架构

**协同合约引擎**/协同合约由一下几个逻辑部分组成：

a) 预置于 Decchain 公链中的，支撑各项功能模块的，使用类 lisp 语言编写的，可编译执行的一系列代码文件，

b) Decchain 公链内置的链下存储引擎

c) 顺序写入的事件存储区。

d) 合约可视化引擎，提供浏览器可识别显示的 html 内容

e) 合约工作流编辑器，用于可视化编辑协同合约工作流。使用协同合约编程语言可以生成链上交易，可以修改绑定到合约自身链下存储。协同合约语言可以生成事件，并顺序存储于绑定到合约的事件存储区。Decchain 公链不是一个通用平台，协同合约的升级必须通过软分叉形式，在保持兼容性的前提下，逐步升级全网节点。

f) 文档协同模块 bloki

## ➢ Decchain 公链的分组共识机制

首先根据 Elastico [文献 1] 中的节点分组共识方法，我们能够获得 tnsBlock.1 这个包含交易列表的对象，tnsBlock 中记录了这个共识小组（commitee）所达成的 transaction 列表的共识，当 committee 的共识达成之后，将结果发送到 directory committee，等待打包。

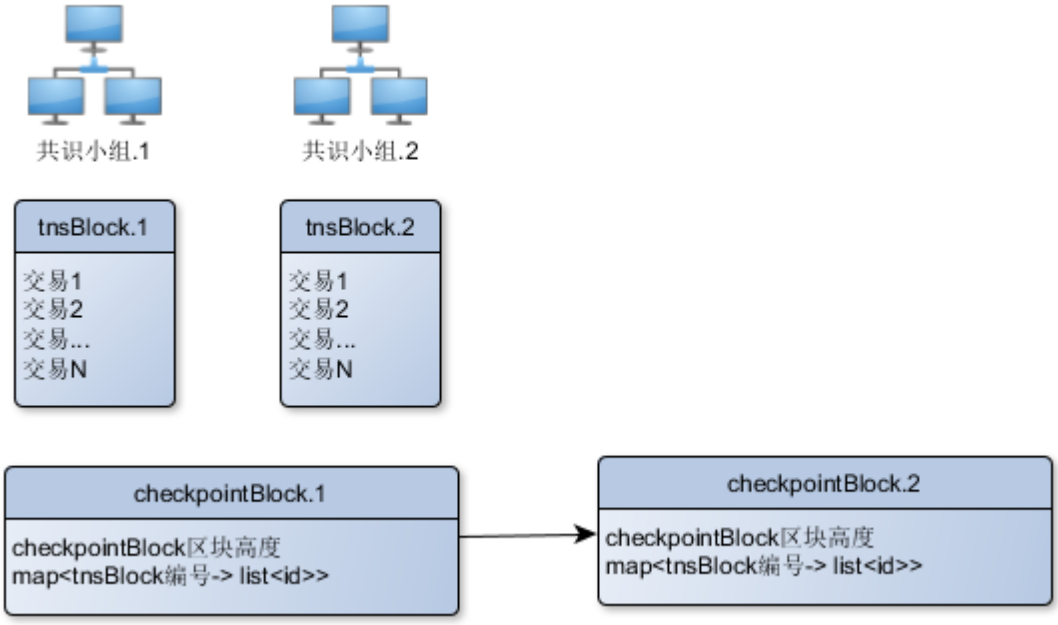



图 3

tnsBlockId 的生成方式：

tnsBlockId = epochId + commiteeId + 这个 epoch 中的这个 committee 中的第几个 tnsBlock

当本轮共识，各个共识小组都达成共识之后，开始打包 checkpointBlock。


图 4

对于全网的每个节点，在共识过程中，顺便构建以上 id 和 ip 地址的映射关键，并且维护更新：当一个 epoch 开始的时候，某个 id 没有节点声明自己是这个 id，则这个节点的 ip 地址从对于的 list 中删除。

节点下线后，补数据的过程：

对于另一个并行的数据完整性维护线程，如果发现某个 id 对应的节点下线了，则需要根据 checkpointBlock 所记录的，所有和 id（记为 idx）共同参与过共

识的其他 id，找到对应的 tnsBlock，选中 1 个在本轮分组的时候，和 idx 分在一组的节点，如果这个节点上没有对应 tnsBlock 的信息（大概率事件，以为分组是随机的），则向这个节点写入因为 idx 标识的节点下线所丢失的数据。如果这个节点上已经有这个 tnsBlock 的数据，则在这个 committee 里继续找，如果全部都有（极小概率事件），则跳过，本轮不补数据。

```
(def TnsBlockInfo  {
      :tnsBlockId
      :tnsBlockHash
})

(def CommiteeInfo  {
      :blocks  ;; commiteeId => TnsBlockInfo  共识小组 id 对应的区块信息
      :ids ;;commiteeId => list<id>  共识小组 id 与小组成员节点的标识符的列表的映射关系
})

(def SimpleCommiteeInfo  {
      commiteeId => TnsBlockInfo  ;;共识小组 id 对应的区块信息
      commiteeId => list<id>  ;;共识小组 id 与小组成员节点的标识符的列表的映射关系
})
```

一个 checkpointBlock 中包含：

```
List<CommiteeInfo>  cis  ;; CommiteeInfo 的列表
Int  checkpointBlockHeight  ;; 当前的 checkpointBlock 高度
```

当 checkpointBlock 打包完成，通过 directory committee 把打包完成消息同步到全网，并且带有各个 tnsBlock hash 值，用于给每个节点进行校验。

对于每个节点，做两件事：

1.在本地根据 checkpointBlockHeight 把 checkpointBlock 区块写入本地的 checkpointBlockStore，

2.根据当前的 checkpointBlockHeight，把当前所在的 commiteeId 共识出来的 tnsBlock 列表进行校验，如果 tnsBlock 列表中的每一个，他的 id、hash 值和

directory committee 发过来的一致，则用 checkpointBlockHeight +"&"+ tnsBlockId 作为 key 值写入 tnsBlockStore。

于是，经过以上过程，对于全网的每个节点，一方面保存了所有的 checkpointBlock，构成一个完整的区块链（但是这个链上只保存的每个 tnsblock 的 id 和 hash）。

在做全数据的完整性校验的时候，我们依次取出每个 checkpointBlock，根据区块链的规则，校验 checkpointBlock 的完整性，然后从 checkpointBlock 中依次取出：

checkpointBlock 和 CommiteeInfo 的列表，对于 CommiteeInfo 中的每个 TnsBlockInfo，根据全局的 idip 映射表，依次的，用 CommiteeInfo 中的 id 对于的 ip，用 tnsBlockId 作为参数，取出 tnsBlock，得到完整的全局数据。我们用伪代码来表示这个过程。

完整数据校验过程：

```
for(checkpointBlock 区块链 中的每个 checkpointBlock， 记为 cb){
  list<tnsBlock>  tnsBlocks; // 待收集的 tnsBlock 列表
  List<CommiteeInfo> cis = cb.cis
  for（cis 中的每一个 CommiteeInfo ， 记为 ci）{
    block = ci.block //tnsBlock 列表
    ids =ci.ids //共识后 tnsBlock 区块写入的 id 列表
    GET_BLOCK:for（ids 中的每个 id ， 记为 id）{
      根据全局的 idIp 映射表， 取出 ip
      从 ip 对应的节点， 根据 block.blockId，取出完整的 tnsBlock
      if(节点上存在这个 blockId 的数据){
        tnsBlocks.append(tnsBlock)
        break GET_BLOCK;
      }else{处理下一个 id}
        由于数据会被积极补全，所以这里大概率能找到存在的数据。
        如果找不到， 则到冷备节点上去找，确保能恢复数据
      }
    }
```

```
  }
  tnsBlock 收集完毕
}
```

[文献 1] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P.Saxena, “A Secure Sharding Protocol for Open Blockchains,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS ’16), pp. 17–30: https://www.comp.nus.edu.sg/~loiluu/papers/elastico.pdf.

## ➢ 协同合约的模版语言

协同合约的定义，包含以下要素：

a) 参与人、权利、义务、投票权
b) 资金转移的触发条件描述
c) 合约执行的工作流描述
d) 合同纠纷情况下，有没有特殊定义的仲裁流程

模版语言语法关键字解释：

a) def：在当前命名空间值来定义变量；
b) fn：在当前命名空间值来定义函数；
c) defmacro：宏定义，包含了名称、参数、代码块，在代码编译期间调用；
d) macroexpand：宏扩展，把宏定义的名称替换为代码块，在代码编译时调用执行。

自治组织合约：

```
(def Organisation  {
        :name   ;;组织名称
        :detail "storj://dec/xxx.md"   ;;组织描述
        :administrators   ;;管理员
        :members   ;;成员
        :modifyMembers (fn [args]   ;;成员管理
```

```
                ((if ( = args "create") (macroexpand '(createMember)))
                 (if ( = args "add") (macroexpand '(addMember)))
                 (if ( = args "delete") (macroexpand '(deleteMember)))
                 (if ( = args "assign") (macroexpand '(assignRole)))))
})
```

子基金合约：

```
(def Fund  {
        :name   ;;基金名称
        :detail "storj://dec/xxx.md"   ;;基金描述
        :organisation   ;;基金所属的组织
        :balance   ;;基金余额
        :votingRights   ;;投票权分配
        :invest (fn [args]   ;;注资入基金
                ((def ret (macroexpand '(transferBalance fundAddr)))
                 (if (== ret "success")
                    (macroexpand '(assignRights memberAddr voting)))
                 ))
})
```

任务合约：

```
(def Task  {
        :name   ;;任务名称
        :detail "storj://dec/xxx.md"   ;;任务描述
        :organisation   ;;任务所属的组织
        :fund   ;;任务所属的基金
        :balance   ;;任务金额
        :issue (fn [arguments]    ;;发标
                ((macroexpand '(transferBalance taskAddr))
                 (macroexpand '(setAttr task overdue date))
                 (macroexpand '(setAttr task finish rewardsRule))
                 (macroexpand '(setAttr task fail punishRule))
```

```
                (macroexpand '(setAttr task fail re-issue))
                (macroexpand '(notify bidderAddr msg))))
    :bid (fn [arguments]     ;;竞标
                ((macroexpand '(offerPrice task))
                 (macroexpand '(offerPrice task))
                 (def allBids (set '(findAllBids task)))
                 (def lowestBid)
                 (doseq [value allBids]
                   (if (< item lowestBid)
                     ((ref-set lowestBid value))))
                 (macroexpand '(assignRights bidderAddr excution))))
    :abort (fn [arguments]     ;;流标
                ((def msg (macroexpand '(msgRecieved address)))
                 (if (= msg "abort")
                   (if (re-issue)
                     (macroexpand '(transferBalance taskAddr))
                     (macroexpand '(exit))))
                 ))
    :checkup (fn [arguments]     ;;审查任务结果
                ((macroexpand '(submit task result))
                 (def judge (macroexpand '(check task result)))
                 (if (= judge "pass")
                   ((macroexpand '(transferBalance bidderAddr)))
                   ((macroexpand '(startArbitrating task))))
                 ))
    :arbitrate (fn [arguments]     ;;仲裁
                ((def price (macroexpand '(getAttr task balance)))
                 (if (< price 1000RMB)
                   ((def result (macroexpand '(judge task)))
                    (if ((macroexpand '(accept task result)))
                      (macroexpand '(exit))
                    )))
```

```
                (macroexpand '(bringIntoCourt lawer))
                (def result (macroexpand '(judge task)))
                (if ((macroexpand '(accept task result)))
                    (macroexpand '(exit)))
                (macroexpand '(bringIntoCourt lawer1 lawer2 lawer3))
                (def result (macroexpand '(judge task)))
                (if ((macroexpand '(accept task result)))
                    (macroexpand '(exit)))
                (macroexpand '(bringIntoCourt arbiter))
                (macroexpand '(judge task))))
})
```

协同合约 API：

```
;;任务处理相关的 API
(defmacro createMember [address] ()) ;;新建成员
(defmacro addMember [address] ()) ;;自治组织添加成员
(defmacro deleteMember [address] ()) ;;自治组织删除成员
(defmacro assignRole [address] ()) ;;自治组织对成员进行权限管理
(defmacro transferBalance [address] ()) ;;转账处理
(defmacro assignRights [address right] ()) ;;分配投票/执行权限
(defmacro setAttr [target attr value] ()) ;;设置属性值
(defmacro getAttr [target attr] ()) ;;设置属性值
(defmacro notify [address msg] ()) ;;通知消息
(defmacro offerPrice [task] ()) ;;竞标人出价投标
(defmacro msgSend [address msg] ()) ;;消息发送
(defmacro msgRecieved [address] ()) ;;消息接收
(defmacro submit [task result] ()) ;;提交任务结果
(defmacro check [task result] ()) ;;审查任务结果
(defmacro startArbitrating [task] ()) ;;发起仲裁
(defmacro bringIntoCourt [...] ()) ;;引入律师和法官
(defmacro judge [task] ()) ;;律师和法官的判决
(defmacro accept [task result] ()) ;;接受任务判决
```

```
(defmacro reject [task result] ()) ;;拒绝任务判决


;;区块和交易的 API
(defmacro blockBlockhash [uint blockNumber] ()) ;;返回给定区块号的哈希值，只支持最近 256 个区块，且不包含当前区块。
(defmacro blockCoinbase [address] ()) ;;当前块矿工的地址。
(defmacro blockDifficulty [uint] ()) ;;当前块的难度。
 (defmacro blockNumber [uint] ()) ;;当前区块的块号。
(defmacro blockTimestamp [uint] ()) ;;  当前块的 Unix 时间戳（从 1970/1/1 00:00:00 UTC 开始所经过的秒数）
 (defmacro txOrigin [address] ()) ;;交易的发送者（全调用链）
```

上面代码中，关键字 defmacro 和 macroexpand 分别是宏定义和宏展开。宏定义看起来很像函数定义。它们都是有一个名字定义，一个参数列表定义和一个代码块定义。调用宏时，在编译期间，宏的代码块将在调用处展开；如果参数是表达式，表达式并不执行，参数变量被表达式代替，在运行期间执行。

例如下面的宏定义：

```
(defmacro add [x y] (x + y))
```

有如下代码行调用宏：

```
(macroexpand '(add 10 20))
```

则编译源代码时，宏被展开如下：

```
(10 + 20)
```

## ➢ 协同合约的存储引擎

Decchain 公链上的协同合约的持久化，引入了开源数据库 RocksDB 来存储。RocksDB 使用一套日志结构的数据库引擎，Key 和 value 是任意大小的字节流。RocksDB 支持 ColumnFamily 的概念。每个 ColumnFamily 的 meltable 与 sstable 都是分开的，所以每一个 ColumnFamily 都可以单独配置，所有 ColumnFamily 共用同一个 WA 文件，可以保证跨 ColumnFamily 写入时的原子性。

协同合约以 Record 为逻辑数据单元进行存储。Record 是 json 格式的结构化文本块，存储在 RocksDB 中。当对 Record 数据变更时，产生一条 RecordLog 记录。RecordLog 是 json 格式并有序列号的数据块，记录了合约执行中的数据变化

历史，保证了数据的可溯性。RecordLog 存储在 binlog 格式的文件中。

binlog 可以分段存储，每段称为一个 block。不同 block 可以存在不同节点 node 中，节点也可以不存储 binlog 只保存最终 state（合约的 state，也就是合约的存储空间，一个合约的存储空间对应一个 rocksdb）。

RecordLog 的种类：

a) ROWS_RECORD

WRITE_ROWS_RECORD，UPDATE_ROWS_RECORD，DELETE_ROWS_RECORD，分别对应 insert，update 和 delete 操作。

对于 insert 操作，WRITE_ROWS_RECORD 包含了要插入的数据。

对于 update 操作，UPDATE_ROWS_RECORD 不仅包含了修改后的数据，还包含了修改前的值。

对于 delete 操作，仅仅需要指定删除的主键（在没有主键的情况下，会给定所有列）。

b) XID_RECORD

在事务提交时，不管是 STATEMENT 还是 ROW 格式的 binlog，都会在末尾添加一个 XID_RECORD 事件代表事务的结束。该事件记录了该事务的 ID，在 CCS 引擎进行崩溃恢复时，根据事务在 binlog 中的提交情况来决定是否提交存储引擎中状态为 prepared 的事务。

c) ROTATE_RECORD

当 binlog 文件的大小达到 max_binlog_size 的值或者执行 flush logs 命令时，binlog 会发生切换，这个时候会在当前的 binlog 日志添加一个 ROTATE_RECORD 事件，用于指定下一个日志的名称和位置。

按照列式数据库的设计范式，RocksDB 中的一个 ColumnFamily 可以看做行式数据库中个一个 Table，每条记录 Record 都是一个版本号的 json 数据，版本号也就是 transactionID。我们在下文用 cf 来表示 ColumnFamily，于是 WRITE_ROWS_RECORD 和对应到 rocksdb 上的操作是 db.put(cf,key,value)；其中，key 是主键，value 是这条数据的内容，为了简化处理，这里的 value 是一个 json 字符串。

例如：

UPDATE_ROWS_RECORD 对应到对应到 rocksd 上的操作是 db.put(cf, key,

value)；其中，key 是主键，value 是这条数据的内容。为了简化处理，这里的 value 是一个 json 字符串。

DELETE_ROWS_RECORD 对应到对应到 rocksdb 上的操作是 db.delete(cf,key);其中，key 是主键。

协同合约的 Record 中可以声明索引字段，这样可使查询时更快。声明索引之后，Record 在创建和修改之后，也同时更新索引数据。索引就是带索引的值和主键的 key-value 对。binlogblock 共识完成之后，修改状态，修改索引。

binlogblock 进行数据分片后，不同节点的数据同步，是以 RecordLog 为最小数据单元的增量同步。协同合约执行过程中，生成 RecordLog 格式的数据日志，线性增加到 binlog 文件中。

RecordLog 同步功能包含了 4 个组件：Connector 组件、RecordParser 组件、RelayStore 组件、RecordStore 组件。整个过程分为以下几步：

1. Connector 组件建立网络连接，获取待同步的 RecordLog 序号，由 cursor 来记录并更新序号；

2.发送传输指令，按照 cursor 标记来传输相关 RecordLog；

3.RecordParser 组件接收到 RecordLog 后解析，并传输给 RelayStore 组件进行中继保存；

4.RecordStore 组件持久化保存 RecordLog 到 binlog 文件中，并交由 RocksDB 重放 RecordLog 中的数据指令，完成数据同步。

过程如图 5：

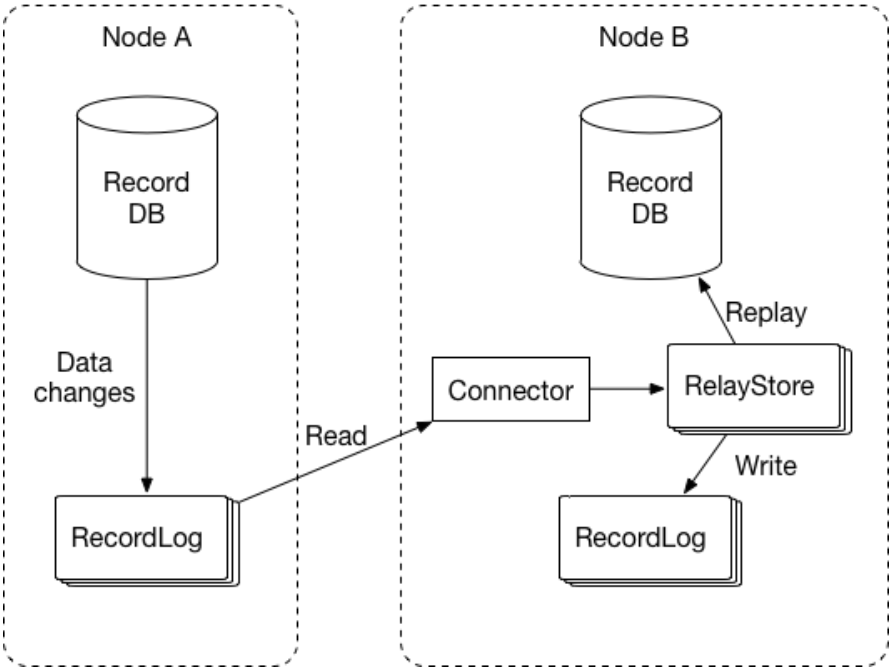



图 5

## ➢ 协同合约的执行引擎

Decchain 公链上的协同合约的执行，由合同参与者的操作和相关事件来触发（比如发标、竞拍任务、任务执行完成等）。协同合约在执行过程中日志记录，保存在由 RecordLog 结构组成的日志文件 binlog 中，最终执行结果保存在 RocksDB 中。当调用协同合约执行时，生成一个新事务并创建一个 transactionId 来标识。一个事务可以包含一个或多个 RecordLog。

协同合约的调用接口，有两种方式：只读调用、写数据调用。只读调用按照类型返回读取的数据结果，写数据调用返回交易号。

我们用接任务这个场景来作为例子：

*insert: { tbl: taskMiner , value: { taskID: xxx, minerId: yyy, price: 123}*

这次调用会返回一个交易号，客户端可以用这个交易号（也可以与 transactionID 一起）进行查询。

当协同合约执行时，如果待处理数据在不同 block 中，则需要一个 warmup 过程，该过程把不同 block 的数据拉取到本执行节点中。例如，a 合约会调用 b、c 合约，则 warmup 调用使得 a 合约的存储节点把 b、c 的合约 state 拉取到本节

点，然后通过事件通知链外 warmup 已经完成。

协同合约执行中的 gas 消耗，以执行过程中产生 RecordLog 对应的 binlog 中的大小来计算。产生的 RecordLog 越多，gas 的消耗量则越多。

协同合约在执行过程中有事件机制，以事件形式通知监听者状态发生变化。事件监听者采用 poll 方式，通过 API 来拉取新事件。该事件机制可与链外系统写作，使链上系统和链外系统同步工作。

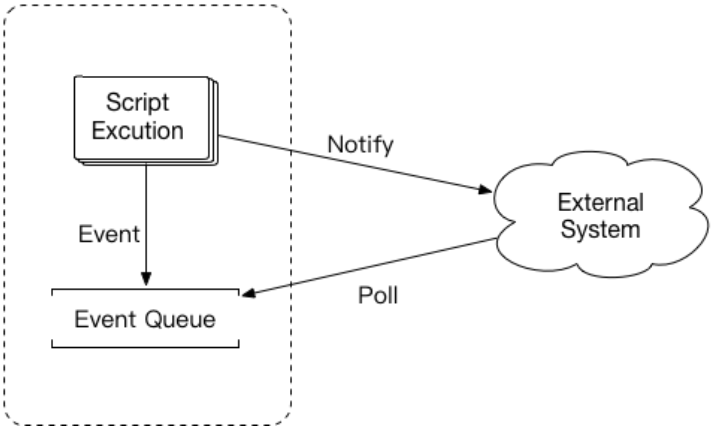



图 6

## ➢ 协同合约的事务引擎

Decchain 公链上，对于每个并发执行的协同合约，根据 RecordLog 的 transactionId 来判断是否数据冲突。对于一个 binlogblock 里更改的每条 record，可以允许不同的记录带有不同的 transactionId，但是不允许同一个记录里出现两个不同的 transactionid，因为这以为着有两个并行提交对同一条记录的修改。打包交易的验证者，验证多个事务是否可以同时提交。在 binlog 打包的时候，在区块头里写入本次包含的 transactionid，用于方便查询某个事务是否提交成功。

如果发生合约执行跨越不同节点 node 时，则两边都要记，如果发生 A->B，A->C 这样的调用，那么网络把 ABC 三个合约调度到一个 node 存储。合约开始执行之前，就把需要调用的各个合约的 state 拉取到一起，然后在本地高速执行。

## ➢ 协同合约的并发执行引擎

在 Decchain 公链并发执行协同合约的过程中，对于一个 binlogblock 里更改的每条 record，可以允许不同的 record 带有不同的 transactionId，但是不允许同一个 record 里出现两个不同的 transactionid，这意味着有两个并行的执行过程提交了对同一条记录的修改。打包交易的验证者负责验证多个事务是否可以同时提交。

假设当前已经提交的 log 的序号是 SEQ1，user1 和 user2 同时执行 contract1，并且被打包到同一个 block 中：

user1 执行 contract1 执行（记为 exe1）后的 log 序号是 SEQ2，也就是说，这次执行输出了从 SEQ1+1 到 SEQ2 这些 RecordLog。

user2 执行 contract1 执行（记为 exe2）后的 log 序号是 SEQ3，也就是说，这次执行输出了从 SEQ1+1 到 SEQ3 这些 RecordLog。

当 SEQ2 和 SEQ3 中的内容没有冲突时，则数据修改成功并提交修改。当 SEQ2 和 SEQ3 中修改了相同的数据，则根据 RecordLog 的时间戳先后顺序判断，先提交者成功，后提交者失败。

这种方式属于乐观锁，并发时可以先都执行，在最后在验证是否有修改冲突，如果有冲突则只允许一个事务提交成功，其它事务执行失败并回滚。

我们可以把 binlog 也做分段 block 打包，加上 hash 的校验。于是，事务的提交绑定到 block 的高度，block 提交等于 transaction 提交。在 binlog 打包的时候，在区块头里写入本次包含的 transactionid，方便后面，其他人查询某个事务是否提交成功。

协同合约执行时，限制单次事务的最长执行时间，比如出块间隔 5s，事务最长时间 3s。当执行超时时，该事务执行失败并回滚。

## ➢ 文档协同

文档协同写作（BLOKI），实现了在不可变的区块链上实现可编辑的协作写作

超文本的方法，它包含了 7 部分：身份识别、名字空间、目录结构、权限体系、历史记录保存、反 spam 机制、标题检索。

a) 身份识别

在 BLOKI 模块中，首先定义 DID 这个概念，也就是基于协同合约的身份识别功能。

我们首先定义一个协同合约，起名为 DidContract。这个协同合约中，存储着可以使用这个身份标识的 address。

我们先简单描述一下 DidContract：

```
(def DidContract {
    :ownerList
    :check (fn [address]
                ((if ( call.address not in ownerList) (return '(false)))
                 (return '(true)))
    :addOwner (fn [address]
                ((if (check(address)) (ownerList.add(address))))
    :removeOwner (fn [address]
                ((if (check(address)) (ownerList.remove(address))))
})
```

假设 A 用户希望开始使用 bloki 模块，则他首先需要部署自己的 DidContract，比如这个 DidContract 的 address=didAddress，并且把自己的多个<address, pk, sk>对中的 address 写入 DidContract 合约中的 ownerList 字段，设这几个 address 分别为 address1，address2。

也就是调用：

didAddress.addOwner(address1);

didAddress.addOwner(address2);

之后， 用户就可以用 address1，address2 这两个地址对于的秘钥对，以 didAddress 的身份进行操作了。

b) ID 注册

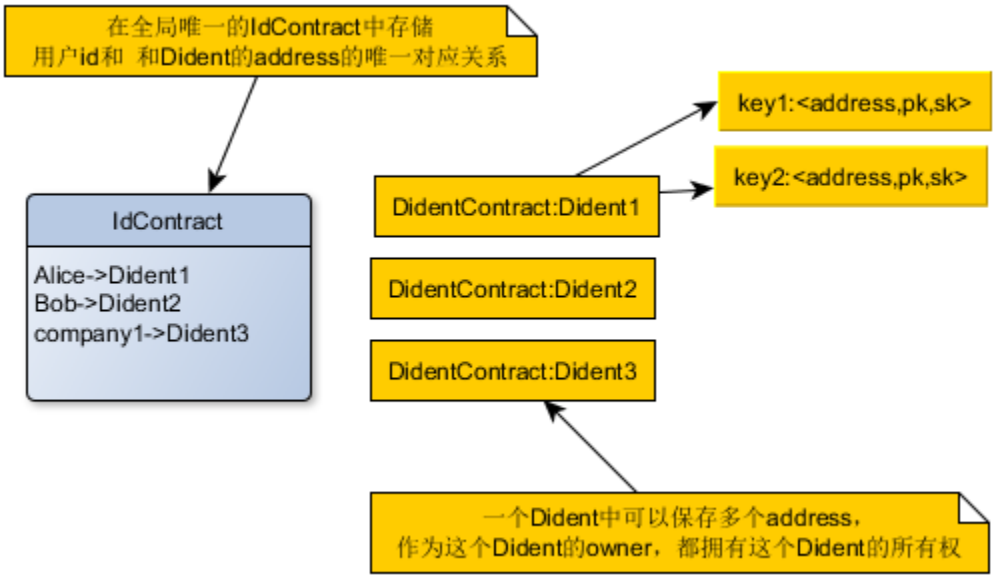
用户可以在 DEC 系统中对自己所拥有的 Dident 注册一个 id 名，比如 abc，这个对应关系注册在系统全局唯一的 IdContract 中。

通过 IdContract 合约的方法：

function registerNewId( str id， address addr，str type)

在这个全局唯一 IdContract 合约中写入数据。如果某个 id 已经被一个 Dident 合约的 address 绑定，则其他 address 不能再次绑定。

Type 有两种：person 和 org， 个人和组织机构。




c) 企业组织结构

进一步的，对于一个企业（组织机构），可以定义一个名为 OrgStructContract 的协同合约，通过这个协同合约来定义企业内的部门结构。

在某个 OrgStructContract 的存储区，这个合约的 owner（私钥持有者）根据 OrgStructContract 合约的标准接口，写入企业的组织结构。

```
(def OrgStructContract {
    :addDepartment (fn [address] ())   ;;添加部门
    :addSubDepartment (fn [department subDepartment] ())   ;;添加次级部门
    :addMember (fn [department memberId] ())   ;;在部门里添加成员
    :listSubDepartment (fn [department] ())   ;;列出次级部门列表，传空串代表一级部门
    :listMember (fn [department] ())   ;;列出部门中的成员
```

```
    :isMember (fn [department memberId] ())   ;;是否是部门中的成员
})
```

按照这个标准合约模板，企业（组织机构）可以建立部门结构和成员。

d) 文档名字空间

用户创建新文档的时候，指定文档路径的时候，只能以自己的 id 或者所述的组织机构的路径开头，不能指定为其他人或者组织机构的名字空间。

比如，用户 alice，属于机构 mycompany，创建新文章的时候，只能指定

docPath: alice/somedir/somesubdir/somedocname

或者

docPath: alice@mycompany/somedir/somesubdir/somedocname

这个限制确保用户只能在他拥有权限的名字空间下发表文档。

e) 权限体系

```
(def BlokiContract  {
    :readPermissionMap   ;;权限映射表，保存文档路径和读取权限设置的对应关系
    :updatePermissionMap   ;;权限映射表，保存文档路径和更新权限设置的对应关系
    :docMetaMap   ;;文档路径和元信息的对应表
})
```

权限体系是 bloki 模块的重要逻辑组成部分。

分为几个层面来设计这个机制：

1. 一个 bloki 文档可以指定归属的协同合约，也就是，对于所对应的协同合约的参与人，都可以读这个 bloki 文档。具体来说，一个阅读者请求阅读这个 bloki 文档的内容的时候，系统通过 readPermissionMap 来判断权限

添加 bloki 时候的交易报文 AddBlokiDocRequest：

```
transaction: {
    type: bloki
    action: create
    refContract: someoneContractId
    id: alice
    org: myorg
```

```
    link: storj://someaddress
    docPath: myid@myorg/somedir/somesubdir/somedocname
    version: 0
    didContractId: xxx
    title: xxx   ;;文档标题
    readPrice: 0.00x   ;;付费阅读价格
    pubkey: xxx
    sign: xxx
}
```

Decchain 公链收到这个报文之后，首先校验签名，以及 pkey 对应的 address 是否是 didContractId 这个合约中的 owner，以及 didContractId 对应的 id 之间的对应关系，再验证 alice 是否是 myorg 的员工，全部验证通过之后，才可以把这个 bloki 发布到链上。

2. 进一步的，如果额外设置读权限字段，比如

{ *allowedDepartment*: dep1, dep2 }

则只有属于 dep1，dep2 部门的员工才可以读取这个文档。

Bloki 文档发布成功之后，BlokiContract 合约的存储区写入权限设置：

```
permission.put (
    myid@myorg/somedir/somesubdir/somedocname => {
    allowedDepartment: dep1, dep2
    })
```

也可以进一步设置更复杂的规则，比如

allowedId: 设置允许访问的 id

allowPublicAccess: 允许企业外的 id 访问

当符合访问权限的用户发出读请求的时候，伪代码如下

```
(def BlockContract  {
        :requestDoc (fn [id address sign pk]
                        ((check(id, address))   ;;校验 id 和 address 的对应关系（address 隶属的 dident
是否申请到了这个 id)
                        ((check(sign))   ;;校验报文签名
                        (def text (encrypt(AESKey)) ;;使用 pk 对文档的 AES 密钥做加密
                         return text ;;返回加密后的 AES 密钥给调用者
})
```

调用者用自己的 sk 解密出 AES 秘钥，获得文档的原文。

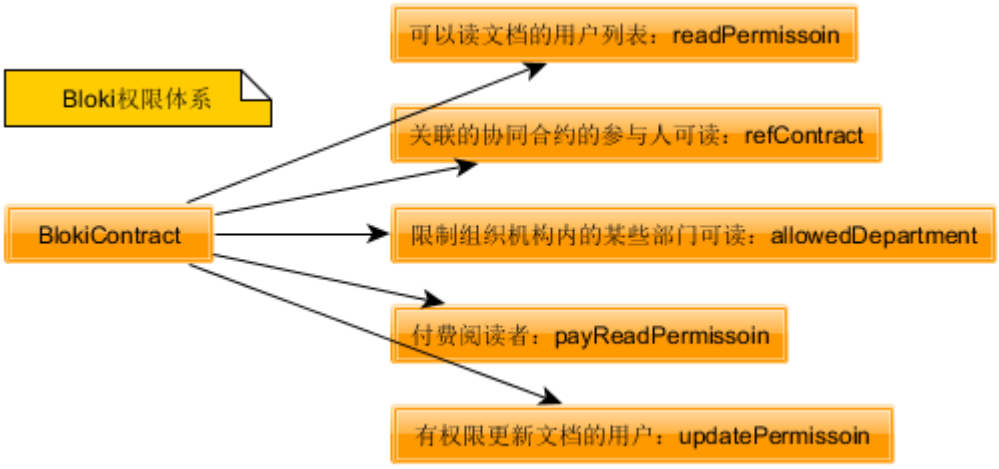
只要调用者不泄露自己的私钥，文档的 AES 秘钥不会在这个过程中公开。

3. 付费阅读者，付费后，在权限表中对应的路径上增加一个条目

payReadPermission.put (

myid@myorg/somedir/somesubdir/somedocname => {allowedId: newReaderId}

)




f)  文档保存

文档的内容需要先加密保存到去中心化存储上， 然后再把元信息更新到 BlokiContract 合约的存储中。

BlokiContract 合约以 AddBlokiDocRequest 类型的报文作为参数，调用 addDoc 方法后，在 docMeta 中增加一个 keyvalue 对：

docMeta.put ({

myid@myorg/somedir/somesubdir/somedocname =>

{refContract: someoneContractId

id：alice

org: myorg

link: storj://someaddress

version:0

didContractId：xxx

address：xxx

hash：xxx

})

更新的时候，调用 updateDoc 方法，更新其中的 version 字段和 hash 字段，删除文档的报文：DelBlokiDocRequest：

transaction:{

type: bloki

action:delete

id：alice

org: myorg

link: storj://someaddress

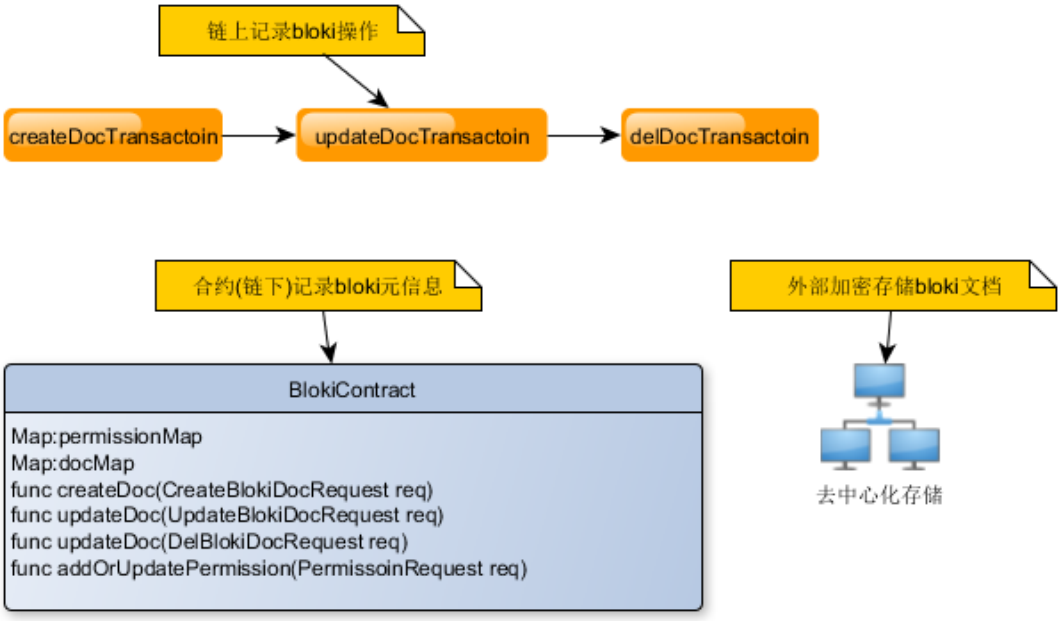
docPath: myid@myorg/somedir/somesubdir/somedocname

didContractId：xxx

pubkey：xxx

sign：xxx

}




g) 历史记录保存

Bloki 文档保存上链的时候，DEC 系统将编辑结果按照“全路径+版本号”，加密保存在去中心化储存上，并且使用 diff 命令比较“编辑前”和“编辑后”两个版本的差异，把 patch 内容加密保存在“全路径+版本号.patch”这个路径下。 经过这样的处理， DEC 系统既保存了当前的快照内容， 也保存了历史修改记录。

h) 反 spam 机制与经济机制

发表一个 bloki 文档的费用由三部分组成：链上交易打包费，链下合约存储费、去中心化存储费。 其中，链上交易打包费和链下合约存储费一次性收取，去中心化存储费由 DEC 系统向文档拥有者按存储时间收费。

以组织机构成员（member of org）的身份发表 bloki 文档的时候，费用由这个 org 承担（org 的 didentAddress 上的 coin 余额）；以 person 身份发布文档的时候，费用由个人账户（person 的 didentAddress 上的 coin 余额）承担。

优秀的 bloki 文档可以通过读者的“赞同”获得信用分。读者可以直接对文档“捐助”，以 coin 形式直转账入文档发表者的 didentAddress 账户余额。

不在文档发表者设置的可读权限内的用户，如果这个文档设置了 readPrice，则读者可以根据文档设置的 readPrice，提出付费阅读申请。

## 术语解释

**自治性组织**：是指一定范围内的自治体全体成员在自由、平等的基础上依法对自治体公共事务实行自我管理的不具有强制性的组织形态。

**加密电子货币**：是一种表示现金的加密序列数，它可以用来表示现实中各种金额的币值。

**共识机制**：是区块链系统中实现不同节点之间建立信任、获取权益的数学算法。

**协同合约**：根据预先定义的电子合同模板，约定参与方同意的权利、义务、执行流程，这些承诺定义了合约的本质、目的和执行方式；并且以数字形式将合约写入计算机可读的代码中，由一台计算机或者计算机网络执行的。当一个预先编好的条件被触发时，协同合约推进合同参与各方共同推进流程、执行合同条款。

# Digital Asset Project Incubator

# DAPI 数字资产孵化器

## 名词解释

DAO ：分布式自治组织。

DET：Decentralized Ecosystem Token，DEC 系统内部流通的通证。

Fund：基金， 在 DAPI 中， 有 DET 币基金和贡献点基金两种基金。

**贡献点**： 每个项目设立之后，项目基金拥有 1 亿贡献点， 用于奖励给项目贡献者。

**投票权重**：出资币数* ln(1+天数)+信用值加权。 备注：天数上加一个自然对数平滑。

**发起提案**：每个提案的发起， 需要消耗 1DET 币（支付给 DAPI 基金会），

通过则返还 0.9DET，驳回则返还 0.5。

**提案通过条件**： 投同意的总权重/总权重 > 0.6。

**提案驳回条件**：投反对的的总权重/总权重 >= 0.4。

如果同意总权重>0.5 但是 <=0.6，则提案状态变更为： 非绝对多数通过。默认提案有效时间 3 天， 到达 50%延长 3 天， 到 55%延长 3 天。 到最长时间依然未到绝对多数，提案取消，返回 0.6DET。

## DET 初始分配方案

DET 作为 DEC 系统内通用的代币，天生具有很高的经济价值，并随着系统的发展壮大而不断增值。

目前 DET 币使用 ERC20 标准发行，合约地址是 0xF72DA6E99b864e26e3a386F2Cc6022882eCB1125（https://etherscan.io/token/0xF72DA6E99b864e26e3a386F2Cc6022882eCB1125）。

初始的 DET 分配方案如下：

- 总发行量 10 亿
- 基金会拥有 50%，即 5 亿
- 私募拥有 20%，即 2 亿
- 机构投资人拥有 20%，即 2 亿
- 团队拥有 10%，即 1 亿

## 创立 DAO 和 DAP

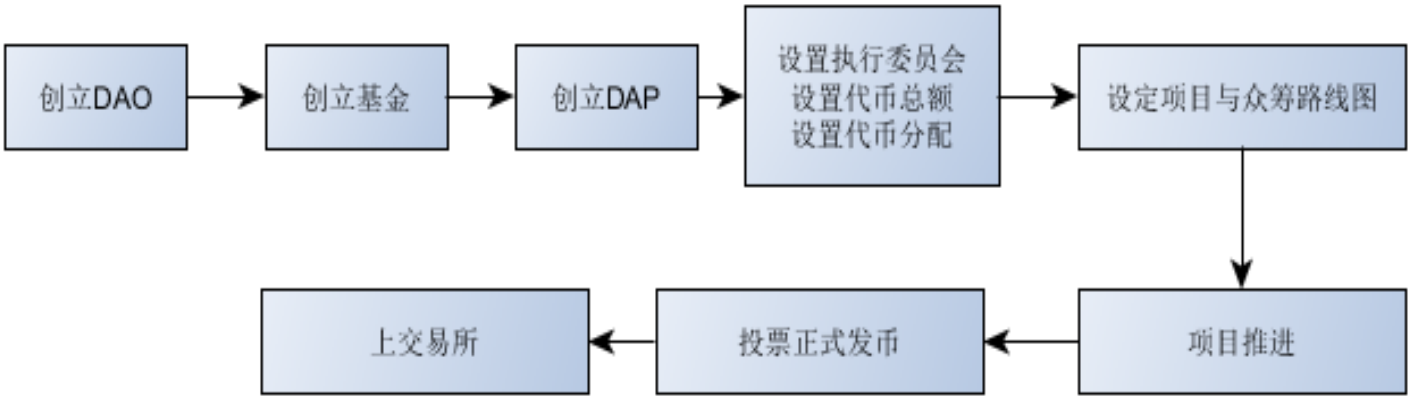
1. 创立一个 DAO 需要 1 千个 DET 币，支付给 DEPI 基金会，作为平台服务费。
2. 基金：一个 DAO 可以拥有多个基金，用于不同目的。默认生成一个 DET 币基金和一个贡献点基金。额外创立一个基金需要 1 千个 DET 币，作为 DAPI 的平台服务费。
3. 创立一个 DAP 需要 1 万个 DET 币，一个 DAO 可以拥有多个 DAP，这 1 万个 DET 币中一半自动转为 DAPI 母基金对于 DAP 项目的基石投资份额，另一半作为平台服务费进入 DAPI 基金会。

   **DAP 创建人支付的这 1 万 DET 币参与投票权的计算。**也就是说，DAP 项目的决策投票权的权重计算中， 项目创始人的币的数量是 1 万 DET。

项目的主要参数：

项目名称，项目简介，项目目标，

发币总数（不可修改）， TPC（token per contribute point）每个贡献点兑换多少个币（不可修改）




## DAP 的创立

1. DAP 项目创始人绑定基金，众筹金额进入所绑定的基金。
2. 设置数字资产的创始团队代币分配比例，比如 团队 10%，于是总代币金额-总贡献点 1 亿*TPC=众筹投资人总共的代币。
   比如，总发行数 10 亿 token，创始团队 10%，TPC=2，留给众筹投资人的总代币数
   =10 亿-1 亿-2 亿=7 亿个。
3. 设置项目 roadmap，每期众筹 DET 数量，以及兑换比例，比如
   18 年 9 月：众筹 10 万 DET，1DET 兑换 1000 xyzToken，共 1 亿 xyzToken
   18 年 10 月：众筹 20 万 DET，1DET 兑换 500 xyzToken，共 1 亿 xyzToken

   每个 milestone 由众筹投资人投票确认达成后，才可以发起下一阶段众筹。每个 milestone 完成后， 可以由决策委员会发起正式发币提案，选择在哪个平台上正式发币， 比如以太坊、EOS、NEO、QTUM。如果在这个阶段，剩余的 xyzToken 尚未兑换完毕，则按照已经兑换完毕的总金额发币。

比如 :创始团队 1 亿 token，众筹投资人 4 亿 token，贡献点基金 2 亿 token，则总共发行 7 亿 token， 而不是最初设定的 10 亿。

## DAP 的运作

### ➢ DAP 决策委员会

DAP 决策委员会由创始人、联合创始人、监督委员会和观察员组成。

项目创始人与联合创始人构成“创始团队”，共同分享 DAP 项目设立时分给团队的代币份额。每个联合创始人增加到 “创始团队”中时，以及代币分配比例，需要“创始团队”按照当前的代币份额（不考虑时间因素），投票，大于等于 60%，绝对多数情况下提议通过。

这套规则主要用于代币分配。

DAP 决策投票权： DAP 项目孵化过程中， 会需要做以下几类重要决策

1. milestone 达成
2. milestone 延期
3. 贡献点发放
4. 发币
5. 上交易所
6. 项目提前终止，退币

**这几类提案只能由决策委员会成员提出。由全部 DAP 项目投资人共同投票决定。**

### ➢ 监督委员会

监督委员会的成员是：众筹投资人、投资基金。

在投资人提交投资的时候，如果投资金额超过目前目标募集金额的 1%，可以附加一个加入决策委员会的要求。如果创始团队投票接受这笔投资，则必须接受这个投资人为监督委员会成员。

如果没有投资人的投资金额超过 1%，则投资金额排名前 100 的投资人自动获得加入监督委员会的资格，投资人也可以选择放弃加入监督委员会。

数字资产投行团队可以申请获得观察员席位，可以了解项目重要决策过程。

**创始团队+监督委员会=DAP 决策委员会。**

基金管理人由监督委员会成员担任，基金的支出需要监督委员会中的至少 2 个人加上签名后才可以生效。

## ➢ 决策类型

- milestone 达成

如果 milestone 提前达成，则后续 milestone 时间自动提前。

- milestone 延期

DAP 决策委员会成员可以提案 milestone 延期，但是每次延后不能超过 1 个月，并且延后之后，最后发币之后属于创始团队决策委员会的代币份额按照最初金额降低 2%。一个 milestone 延期之后，后续的 milestone 时间顺延。

## ➢ 贡献点发放

决策委员会可以提案分发贡献点，根据项目贡献人的实际贡献。投票通过之后自动发放。

## ➢ 发币

在基础链，比如以太坊、EOS、NEO、QTUM 上发行代币

## ➢ 上交易所

数字资产的交易所发行，由 DAP 决策委员会和数字资产投行团队协商具体发行事宜。

## ➢ 项目提前终止，退币

如果项目出现问题，决策委员会可以发起关闭项目，剩余 DET 币按出资比例退币给出资人。

## 总结

DAPI 平台通过系统化的管理整个数字资产的关键决策--创建、众筹、决策、发币、上交易所、代币分配，可以大大提升整个生态中的效率。
与此同时，DET 代币作为生态中唯一的核心代币，其所代表的价值将会不断提升。

## 核心团队

创始人兼 CEO——Robin Wong

- 互金独角兽公司技术总监，技术研究院院长
- 搜索引擎与广告技术专家
- 金融系统技术专家
- 区块链早期研究者，所管理的技术团队提交了多项区块链核心专利
- 经验丰富的证券交易者，主管开发股票智能投顾、基金智能投顾系统
- 人工智能研究者，主管开发智能会话机器人
- 现代密码学研究者

联合创始人兼 CTO——Alex Xing

- 国内某一线互联网公司工作经历
- 10 年互联网从业
- 经历多语言编程专家
- 擅长高并发系统的整体构架设计
- 擅长解决复杂业务的系统规划
- 连续创业者，精锐新成 CTO
- 涉足电商、母婴和金融等创业领域
- 管理多个区块链技术投资社群

联合创始人兼 COO——Jason Shen

- 8 年成功的创业经历
- 创立的电商公司最高年营收上亿
- 管理 100 人以上的运营团队经验
- 曾为多个国际一线品牌提供线上运营服务

- 丰富的新媒体运营管理经验
- 丰富的 KOL、内容、网红等市场推广资源

联合创始人兼 CSO——Sun Moon

- 高级信息安全专家
- 具有代码审计、漏洞挖掘、工具研发、形式化验证等多项核心技术经验
- 曾为多个大中型金融企业提交高质量漏洞并进行应急响应处理工作
- 国内最早一批区块链及智能合约生态安全推动研究人员
- 多个交易所安全及智能合约代码审计实战经验

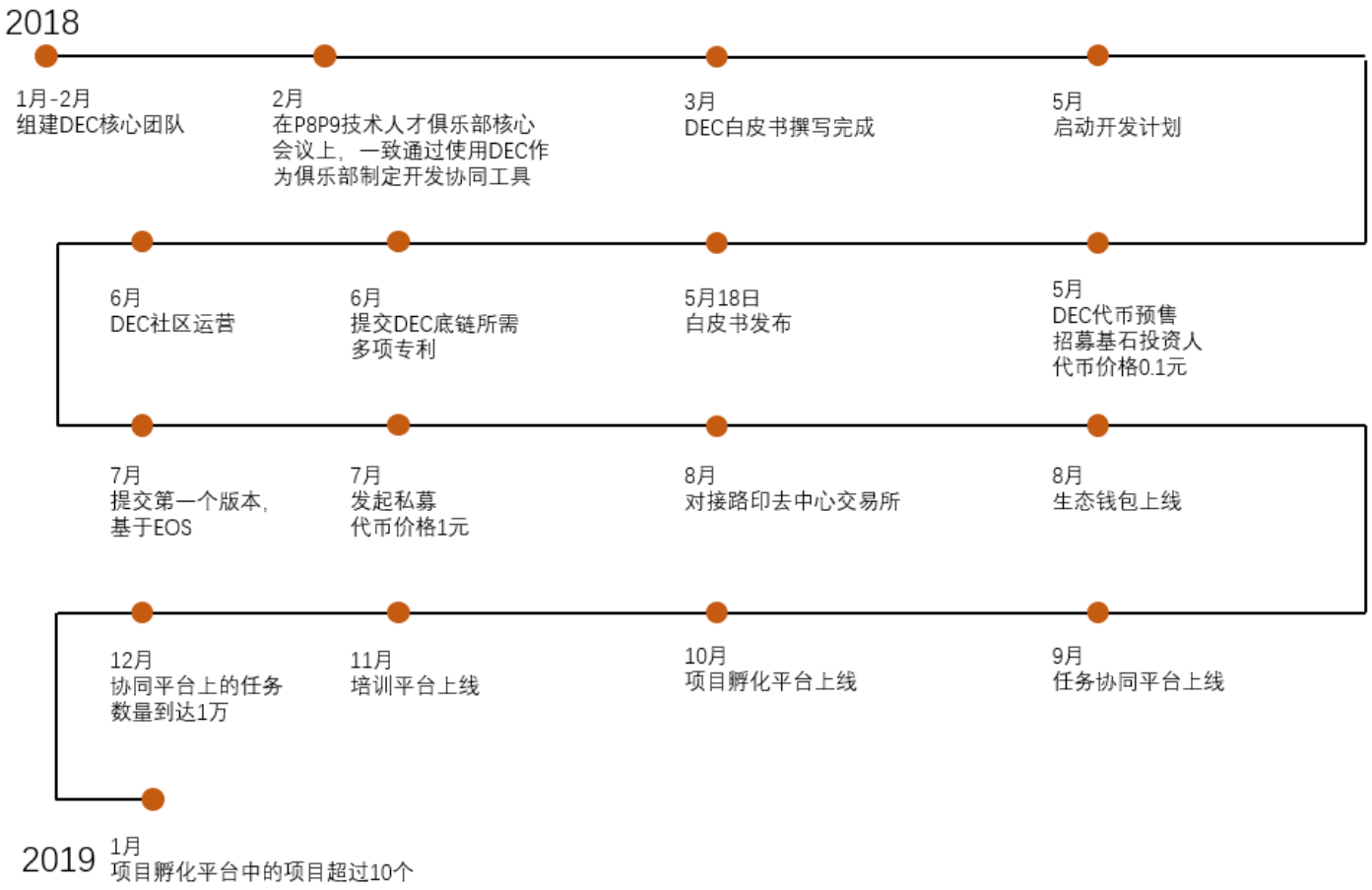
联合创始人兼 CMO——Timon Sun

- 连续创业者，14 年创业，做社交产品，负责运营
- 15 年成立外包公司做外包业务

两万三微信好友，皆为互联网从业者，渠道资源丰富且可落地

- 团队开发项目丰富，经历了七十多个各行业早期和中期的项目

## 项目路线图


2018
1月-2月
组建DEC核心团队
2月
在P8P9技术人才俱乐部核心会议上，一致通过使用DEC作为俱乐部制定开发协同工具
3月
DEC白皮书撰写完成
5月
启动开发计划
6月
DEC社区运营
6月
提交DEC底链所需多项专利
5月18日
白皮书发布
5月
DEC代币预售
招募基石投资人
代币价格0.1元
7月
提交第一个版本，基于EOS
7月
发起私募
代币价格1元
8月
对接路印去中心交易所
8月
生态钱包上线
12月
协同平台上的任务数量到达1万
11月
培训平台上线
10月
项目孵化平台上线
9月
任务协同平台上线
2019
1月
项目孵化平台中的项目超过10个