一种去中心化的、可扩展的区块链查询解决方案

APIS 基金会

2020 年 10 月 5 日

摘要

我们所提出的应用程序接口（API），是去中心化索引和查询协议运行的中间件协议，它可以确保完全去中心化金融以及去中心化网络体系结构成为主流。目前的数据索引和查询方案要么是中心化的体系结构（依赖随时面临众多性能和监管风险的某一方），要么就是逻辑上的去中心化，因此我们需要深入了解公共区块链协议层，同时掩盖全球大部分开发者与公共区块链基础设施之间的交互。从体系结构来说，API 是去中心化的，即端点数据的提供者可以是任何加入 API 网络的行为者，但它在逻辑上却是中心化的，即客户端开发者将可以参考常见且经常使用的 API 格式结构，即 RESTful 和 GraphQL 端点，从而查询公共区块链所产生的数据集。

目录表

1 背景......4

1.1 背景......4

1.2 引言......5

2 体系结构......6

2.1 信息传播协议......7

2.2 信息格式概述......8

2.3 API 核心合约......9

2.4 管理合约（GC）......10

2.5 争议解决合约（DRC）工厂......11

2.6 Optimistic Rollup 合约......14

3 应用范围......16

3.1 IDs......17

3.2 THE APIS 代币......17

3.3 代币发行，社区所有权......18

4 讨论......19

5 附录：REST 与 GraphQL 的对比分析 19
5.1 REST 的历史和分析 19
5.2 REST APIs 运用: 22
5.3 REST 的优缺点: 24
5.4 GraphQL 的历史和分析 .... 错误！未定义书签。
5.5 GraphQL 作为“获取树（Fetching Tree）”的作用 27
5.6 GraphQL 请求的剖析 .... 错误！未定义书签。
5.7 关于 GraphQL 解析器的注意事项: .... 错误！未定义书签。
5.8 直接对比 .... 错误！未定义书签。
5.9 GraphQL 的不足之处 .... 错误！未定义书签。
6 参考书目 32

1　背景

1.1　背景

公共区块链体系结构确保了互联网的下一次迭代：数字应用的去中心化。去中心化的数字应用，目前可分为后续的两类：去中心化金融以及去中心化网络。

去中心化金融目前的应用率明显高于去中心化网络，造成这一现象的原因是多方面的，不过最重要的是形成当代用户体验所需的计算开销间的差异：与依赖“新闻管理（news feeds）”、“匹配算法”和“产品推荐”的网络应用准功能所需的算法相比，货币或类货币工具的计算主要依赖于浮点运算，所需计算开销较少。因此，去中心化金融如今可以实现与国际支付网络（比如 SWIFT[1]）相同的吞吐量，并有望在不久的将来，实现与现代国内支付网络（比如 VISA）相同的私有扩展性。去中心化金融会继续将除支付之外的其他金融应用纳入其中，即投资、交易、借贷和储蓄。随着去中心化金融规模的不断扩大，这 4 种情况已经开始出现有意义的增长趋势。

去中心化网络离主流应用的距离更远：我们近似认为，与对应的去中心化金融相比，目前所流行的去中心化网络协议，在私密性和性能上落后 3 到 5 年。去中心化金融可利用零知识证明来实现私密性的扩展，而且零知识证明的改进速度是成指数倍增的[2]，但去中心化网络所依赖的技术却还处于进一步的初级阶段，即安全多方计算（sMPC）和全同态加密（FHE）[3]。如果这些加密技术继续以零知识证明在过去 5 年中的方式来进行扩展，那么去中心化网络将能够达到与去中心化金融一样的应用效果。

但是，只要去中心化的金融和网络产品存在任何程度的中心化，那么整个系统就会面临风险。API 法则指出：“一个产品只有在它最集中的部分才是分散的。”我们认为 API 是为了实现去中心化金融和网络这一愿景，没有任何平台风险会屈从于地方法规，尤其是那些目前控制着全球政权的政体，所以这些政体有动力去减少干扰。去中心化金融和去中心化网络栈中最为集中的组件是数据索引和查询层，这是一个关键任务组件，如果没有这个组件，那么去中心化应用将无法实现主流应用。因此，

API 可以让去中心化金融和网络产品渗透到全世界，同时确保完全去中心化网络体系结构的安全属性

1.2 引言

当前的公共区块链系统在推出时，就最大限度地实现了中心化，但很快，这一系统就围绕解决用户痛点的元素重新实现了中心化。最突出的中心化成功实例即中心化交易所，尽管去中心化交易所的指数化应用认为，公共区块链系统的这种中心化特征可能很快就会变成去中心化[4] 。

因此，我们转向现在将成为公共区块链设计空间中最大、最关键任务的中心化特征：数据库层，本文也称之为查询层。对于所有的去中心化应用来说，查询层都是至关重要的任务型应用。

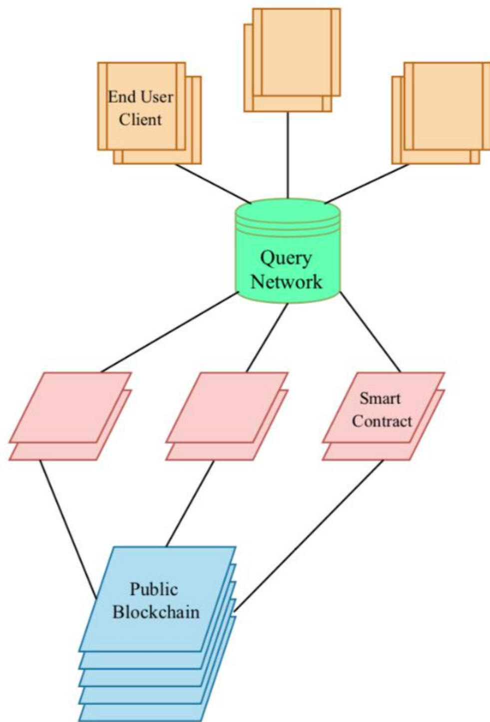

图 1.2.1: 当前分散式应用体系结构。

终端用户客户端应用程序通常通过以下过程与公共区块链进行交互：用户的私钥[5]通过客户端的轻客户端[6]来签署一项交易，随后由一个全节点（其上线时间由应用程序本身或第三方服务提供商承诺）传播到其他全节点，包括公共区块链的验证者，然后验证者计算该交易的结果，并将该交易及其结果打包到其随后的区块中，如果支付给验证者的费用激励他们选择该交易而不是其他交易。关于发送交易的过程，我们在此将其称为写入区块链。写入区块链这一过程中最主要的障碍是支付给网络的交易费用的优化[7]，这是因为经常使用的区块链存在交易纳入的可变费用，这主要表现为交易失败[8]或交易费用过高[9]，这两种情况都将造成公共区块链所特有的糟糕的用户体验。

在所有主流使用的公共区块链虚拟机器中，写入区块链这一过程会产生事件和日志[10]，这些事件和日志是作为交易以及与智能合约交互的结果而存在的；事件和日志的主要作用是为去中心化的应用用户界面生成返回值。但是，就智能合约所发出的事件和日志及其结果状态的管理而言，这是一个时间密集且计算量大的过程，绝大多数开发者不具备执行资源。智能合约和去中心化应用开发者能够为其合约的安全执行进行最合适的优化，并为这些产品找到其与市场之间的契合点。专业化指数和查询服务的市场以指数级的速度进行增长[11]，但市场领导者是中心化的服务提供商，其运营商和股东受到所在国家的监管，同时也有动力形成垄断，以获取超额费用和利润，最终将成本转嫁给广大用户和开发者。依赖于中心化服务商的去中心化应用，将会面临键人和平台风险，从而使其整个用户体验的运营也面临风险；只需要一次糟糕的用户体验，用户就再也不信任这一产品[12]。考虑到从区块链上读取，API 为索引和查询服务的去中心化提供了一个协议，其中包括 Gas 优化指数（gas optimization indices），它可以帮助开发者和用户尽可能高效地写入区块链，从而保证去中心化网络、其构建者以及用户的完全去中心化体系结构。

2 体系结构

通过利用发布-订阅信息传播协议[13]，API 实现其去中心化的查询和索引体系结构，并在此基础上叠加了一个由以太坊（Ethereum）公共区块链所启用的激励驱动

去中心化组织，以及一个自定义的第二层区块链，其特征表现为 optimistic rollup [14]。

2.1 信息传播协议

API 查询网络由两个参与者组成，他们都是开放的，且任何一方都可以进行配置：即 API 网关和 API 节点。分散式应用及其开发者所请求的查询由 API 网关进行响应，同时 API 节点对与一个或一组智能合约相关的数据库进行索引和管理，这样 API 节点可以以平均 400ms 的响应时间（$\epsilon(t) < 400ms$）来响应 API 网关的请求，其条件是来自 API 网关的请求端点已经由 API 节点来进行维护。

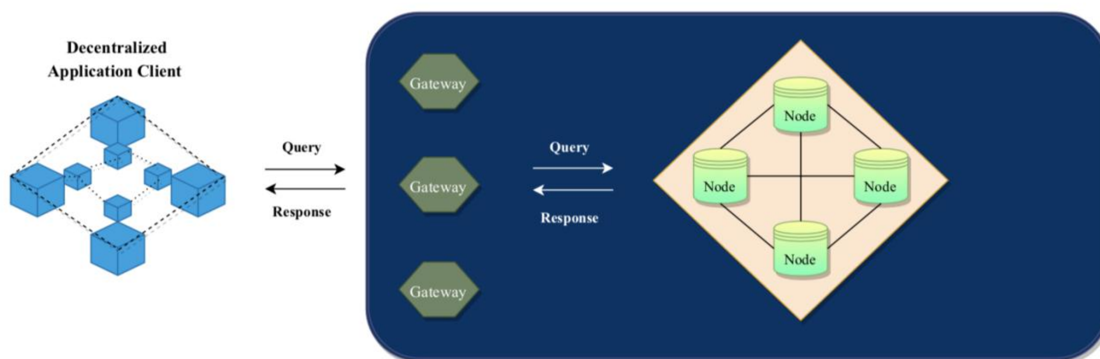

图 2.1.1: API 网络信息传播协议。

网关和节点之间的通信是通过无经纪人的 libp2p 发布-订阅信息协议来进行维持的，其中节点订阅来自网关的信息，它们可以直接从网关接收信息，也可以从其他同样订阅了该网关信息传播的 API 节点来接收信息，尽管对于任何一个设计良好的加密经济系统来说，关于节点的假设是，对手方节点在任何时候都是敌对的。

网关维护订阅节点的一个注册表，它们根据端点<ID>向其请求端点，这是因为节点只会索引一组选定的 ID。ID 是网络所支持的智能合约集的代表性散列；只有 ID 的散列才得以维持在节点和网关状态尝试的链上，在“2.6 Optimistic Rollup 合约”中，我们将进一步阐述。尽管我们建议采用先入先出的方式，但事实上，是由每个网关来决定如何过滤每个终端响应的节点信息，如果后续信息与第一次发送的信息

有所不同，那么我们可以为网关提供产生质疑的证据，在“2.5 争议解决合约工厂”中，我们将进一步阐述。

2.2 信息格式概述

查询由客户端应用程序直接发送到网关，并由网关以行业标准的 GraphQL 和 REST 格式返回，从而允许全球开发者库采用 API，以及扩大公共区块链的开发者市场。GraphQL 和 RESTful 在管理和返回查询的方式上有所区别：GraphQL 为请求数据提供了一个单一的端点，而 REST 则提供了多个端点，这使得 REST 的效率较低，但也更具自定义性。由于 REST 的先发优势以及随之而来的众所周知的整合，REST 比 GraphQL 取得了更多的主流应用范围；但是，GraphQL 可以根据查询请求的具体内容来自定义其查询响应，从而确保不会出现数据的过度获取，而且应用软件只需要一种单边的通信模式即可收到所需数据。我们相信 GraphQL 最终会超越 REST，但为了鼓励尽可能多的工程师使用，在可预见的未来，API 网关将支持这两种 API 格式。

为了保持 API 节点和网关之间的最佳通信，API 节点在启动时只支持 RESTful 请求，这是因为通过 API RG 转换器，API 网关可以将 RESTful 请求打包成 GraphQL 格式，而 API RG 转换器是一种拥有专利但开源的技术，建立在过去对话方案中所实现的著名的拼接[15]和前缀[16]技术之上。因此，网关允许 API 网络支持客户端的两种查询类型，同时减轻运行某一 API 节点所需的操作强度。API RG 转换器可以在 100ms 以内（ϵ < 100ms）将 RESTful 端点转化为 GraphQL 端点（反之亦然），每次成功的时间约为 95ms。

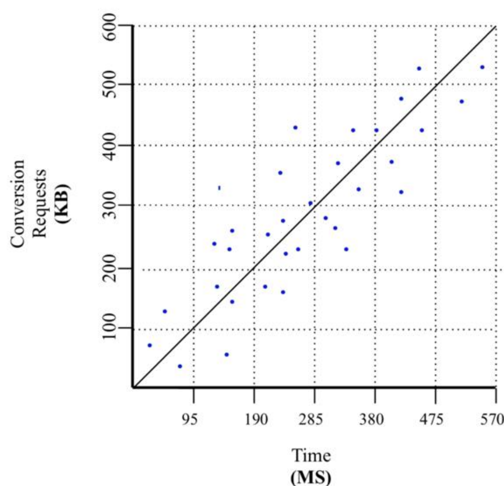

图 2.2.1: 在内部实际测试中的 API RG 转换时间。

95ms，如果加上信息通信所估计的往返 400ms，我们可认为查询的返回时间在 500ms 以内（$\epsilon < 500ms$）。随着网络规模的扩大，网络性能也会随之扩大，这是因为在尽可能快的条件下，网关和节点之间存在关于返回查询的竞争，比如为了赢得第三方客户端开发者业务。与中心化聚合器相比，这是协议的长期经济效益，由于垄断的市场行为，中心化聚合器可以承受较慢的创新速度。不过，在过渡期内，对于去中心化金融和网络用户体验来说，我们认为 500ms 以下的返回时间已经足够了。

2.3 API 核心合约

在合理的经济激励下，API 网关和节点可起一定作用，在一个假设为永久对抗的环境中征集服务费。API 网络的状态由部署在以太坊（Ethereum）公共区块链上的 API 智能合约以及第二层 rollup 体系结构来进行维护。

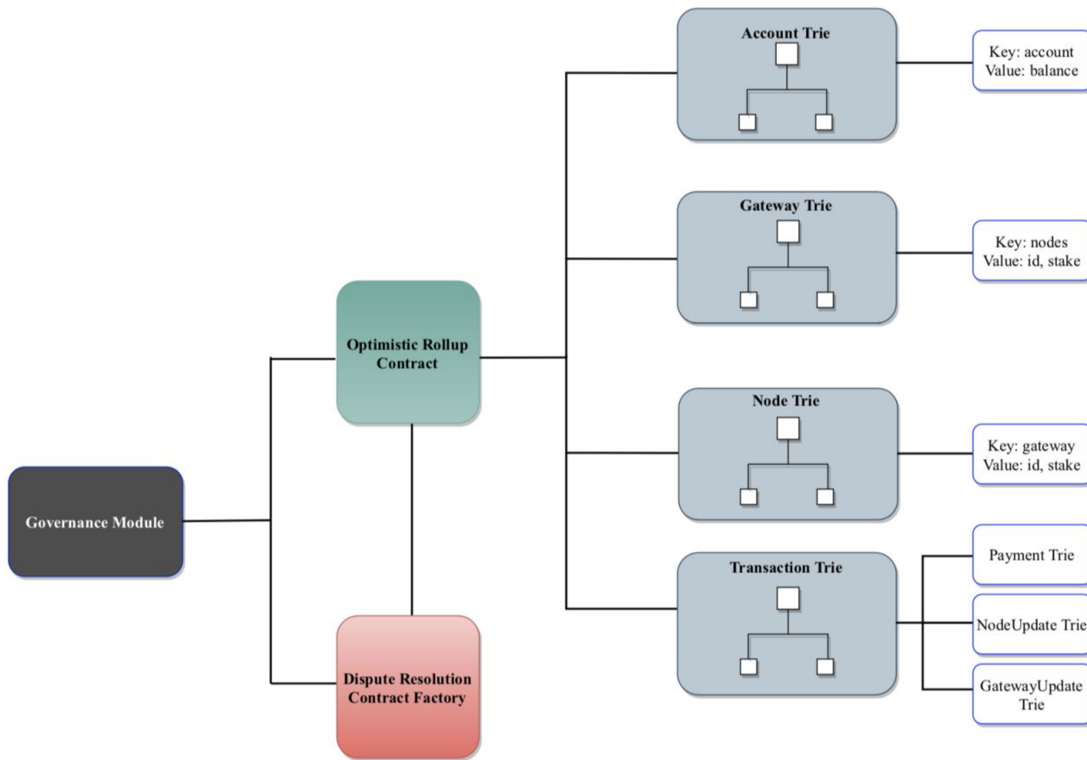

图 2.3.1：API 网络以太坊（Ethereum）一层和二层体系结构。

在以太坊（Ethereum）公共区块链上所部署的 3 个主要合约是管理合约、争议解决工厂合约以及 optimistic rollup 合约。管理合约可以对体系结构的其他部分进行配置和升级；争议解决合约工厂可以确保 API 网关和节点有动力只完成准确响应；optimistic rollup 合约可以让体系结构的规模大大超过当前以太坊（Ethereum）公共区块链的吞吐量。在整个设计中，对于 THE APIS 代币强调了超流动性这一概念，如果达到这一程度，那么代币持有者可以同时参与管理、争议解决和 optimistic rollup 验证，最大限度地提高 THE APIS 代币持有者的潜在创收能力。

2.4 管理合约（GC）

API 由去中心化的 THE APIS 代币持有者群体来进行升级和配置，从而推动管理合约（GC）成为该设计中最重要的对象。GC 是开源复合管理合约所自定义的分叉，

人们已证明该合约是以太坊（Ethereum）生态系统中最全面、最完善的管理模块，这一点从著名的第二层协议（比如 yearn.finance[17]）的重新部署中就可以看出。GC 的目的是允许 THE APIS 代币持有者对 API 争议解决工厂合约和 API Optimistic Rollup 合约进行更新，以及在 API 协议中实现更多的智能合约。THE APIS 代币持有者可以直接进行投票，也可以将其投票权委托给可信赖的交易方；在达到法定人数并最终完成投票后，所有投票还要求代币在 GC 中锁定 1.504 天（10000 个以太坊（Ethereum）主网区块），从而抑制恶意方购买 THE APIS 代币，通过对 API 网络产生负面影响的 API 改进提案（APiiPs）。批准一个 APiiP 所需的法定人数将是网络的 25%，尽管我们希望未来的 APiiP 将投票提高法定人数要求，这是因为通过 API 流动性挖掘计划，项目变得越来越分散，这一点我们将在“3.3 社区所有权”中进一步说明。

2.5 争议解决合约（DRC）工厂

API 争议解决合约（DRC）工厂最初是作为管理合约中的一个函数而设计的，但后来人们将其分离出来，以确保完整的模块化，允许在能够部署一个主观性较低的解决机制时（随着零知识证明规模的不断扩大，应该会出现这种情况），对合约进行升级。DRC 工厂的目的是允许 API 持有者专门对发生在链外的事件进行投票，即一个节点或网关是否向另一个网关或终端用户客户端提供了一个欺诈或误报的端点。这一函数不能包含在管理合约或 Optimistic Rollup 合约中，这是因为它需要在与该系统其他部分隔离时最为安全的自定义结构和函数。但是，我们可以将管理合约中的 THE APIS 代币，同时用于任何 DRC 工厂子合约中来进行投票，尽管这一操作需要在以太坊（Ethereum）主网上签署额外交易才能实现。

DRC 工厂利用自定义的、股权驱动的投票函数，即<challenge>函数和<defend>函数，THE APIS 代币持有者可通过这些函数以 THE APIS 代币的押注方式发出投票信号，结合简单的操作码（即 CREATE2），从而允许任何人部署争议解决（DR）子合约，其地址和状态以键值映射的方式存储在 DRC 工厂中；DR 子合约在 DRC 工厂合约中可为下列 2 种状态中的 1 种：<真>或<假>。考虑到正当程序，所有 DR 子合约都默认为<假>。

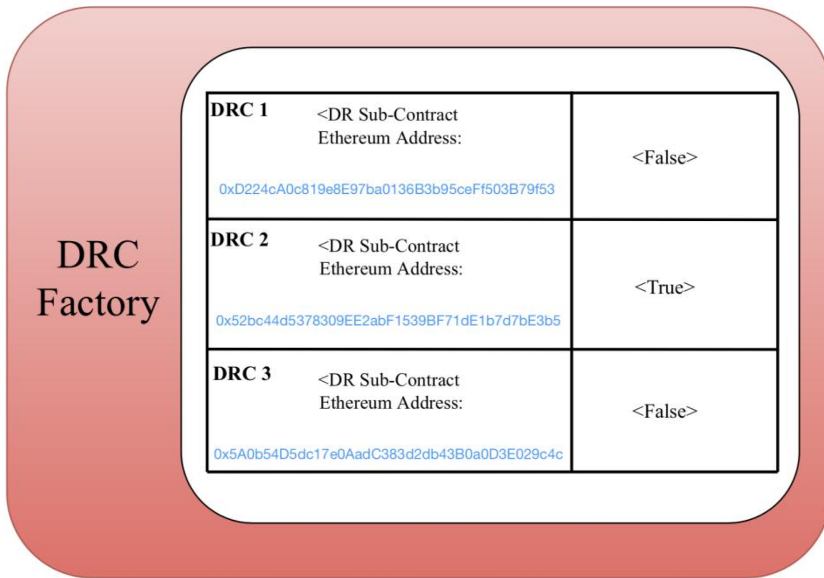

图 2.5.1: DRC 工厂简化体系结构。

一个 DRC 子合约拥有 7.52 天（50000 个以太坊（Ethereum）主网区块）的时间来解决自身问题，之后，如果问题仍未解决，那么整个系统将分成两个独立的域，其设计与 Augur 的滞留状态解决方案非常相似[18]。这时，THE APIS 代币持有者必须根据其认为合适的争议解决结果，决定承认网络的哪个分叉为规范式。

DR 子合约的唯一目的是解决来自 Optimistic Rollup 合约（ORC）的交易是否为<真>或<假>。为了执行知悉投票，API 持有者必须知道从网关发送到用户或从节点发送到网关的端点。因此，Optimistic Rollup 合约中的每一个支付交易都存在这样的结构：〈客户端账户地址、客户端账户签名、服务器账户地址、服务器账户签名、交易金额、端点散列值、随机数（Nonce）〉，其中客户端为支付方（终端用户客户端或网关），服务器为收款方（网关或节点）。

```
{
        struct    PaymentTransaction    {
    address client;
    signature                    client;
    address server;
```

```
            signature server;
            uint                                amount;
            string[]                            endpoint;
            uint                                nonce;
                                                       }
        }
    }
```

图 2.5.2：自定义支付交易结构

端点散列提供了所交付端点的规范真实性，这样 DR 子合约的投票者只需将自己所请求端点的端点散列结果与被质疑交易中的端点散列进行核对即可。如果投票者的端点散列与交易的端点散列相匹配，那么投票者将投票认为该交易为<真>，因此 DR 子合同为<假>；如果投票者的端点散列与交易的端点散列不匹配，那么投票者将投票认为该交易为<假>，因此 DR 子合约为<真>。

为了生成端点散列，投票者必须能够访问发送给请求者的端点。为了实现这一操作，所有 API 节点和网关都会将请求和传输的端点存储 7.52 天（50000 个以太坊（Ethereum）主网区块），之后它们可以从存储中修剪完整的端点数据。完整的端点数据（端点散列代表该数据）包括：<ID>，一个可调用 API 数据集的散列代表值；<数据类型>，有 GraphQL 或 REST 这两种可能的返回方式；<信息输入>，是 API 调用中所请求的具体变量；<信息输出>，是对所请求变量的具体响应，<信息输入>和<信息输出>都用等大小的数组进行存储，这样就可以进行反向查找。由于区块链无法客观验证一个<信息输出>是否是对<信息输入>的正确响应，因此，争议解决是一个主观的过程，有明确的链上参照物作为规范。如果 DR 子合约为<真>，那么已经支付 API 请求费用的服务器方的 API 股权将得以削减，其比例为：

$$k * \sum_{i=1}^{n} x\left(\frac{z}{y}\right),$$

其中，x 为股权资金，y 为其股权的锁定时间，z 为该股权存入后所产生的违法行为次数（开始为 1，每出现一次虚假违法行为则增加 1 次，在 ORC 的节点和网关状态尝试中均可识别），k=0.326。在人们证实某一节点或网关存在欺诈的情况下，付款无法返还，而是按照被欺诈的客户以及投票正确（与其投票的规模成比例 ）

的当事人群体的 10-90 这一比例，来分配所削减的股权。尽管我们相信使用率最高的节点和网关的股权数量会大大高于 THE APIS，因为股权越多的 THE APIS 就越值得信任，获得的使用率也就越高，但是，它仍然要求至少需要 10000 个 THE APIS 代币的股权才能作为节点或网关。

如果质疑本身具有欺诈性的，而不是 API 请求，那么质疑者的股权将被削减：为了减少垃圾信息，这需要 500 个 THE APIS 代币的股权才能提出质疑。虽然这可能会给那些希望提出质疑的人带来冲突，但由于其交易所的流动性，获得 500 个 THE APIS 代币是可行的，而且正确质疑的好处将是超过 10 倍以上的 gas 成本。如果质疑者错了，那么其股权将按照 10/90 的比例分配给进行正确 API 请求的一方以及投票正确的一方，所获得股权与其投票规模成比例。

要想提出质疑，必须满足的条件是在最后一次质疑后的 18.1 小时内（5000 个以太坊（Ethereum）主网区块），一组当事人至少押注对方当前押注资金的 1.5 倍。如果 DR 子合约在 7.52 天后（5000 个以太坊（Ethereum）主网区块）仍未解决，那么该系统将产生分叉，由个人用户决定支持哪个分叉。分叉的目的是鼓励人们更快解决，因为市场很可能会以至少 90-10 的比例汇聚到其中一个分叉上，这也是大多数公共区块链协议分叉的共同点。

2.6 Optimistic Rollup 合约

ORC 拥有节点状态、网关状态、交易状态和账户状态。尽管最近基于 UTXO 的 optimistic rollups 很流行，但是，在所有基于账户的 optimistic rollups 中，账户状态出现频率最高，而对主链和 rollup 来说，交易执行的现状仍将是账户状态[19]。所有状态在以太坊（Ethereum）主链上都以其 MPT 树（Patricia Merkle Trie）的散列来表示[20]。节点和网关树的深度为 14，允许2^{14}（$1.6*10^4$）个节点和网关，未来可以通过 APiiP 增加深度大小。帐户状态深度为 22，允许$2^{22}(4.2*10^6)$个帐户（用户、网关和节点之和），将来可以通过 APiiP 增加深度。交易状态树深度为 40，允许$2^{40}(1.1*10^{12})$笔交易，已处理且在 DRC 工厂或 ORC 中均未受到质疑的交易在 50000 块（7.52 天）内进行修剪并从交易树中进行移除。

如果有所需要，那么从树形结构（Trie）中修剪出来的交易可以通过节点和网关维持更长时间，尽管我们没有理由通过 API 协议的激励措施来持有它们。

交易树有三个子的树形结构，其中之前引用的只有支付树。除了支付树，还实现了节点更新树和网关更新树，从而允许对节点状态和网关状态树进行自定义更新。节点和网关交易都包含相同的结构，其主要目的是允许节点和网关增加或减少其 THE APIS 利害关系，这些节点和网关必须已经通过以太坊（Ethereum）主网的标准智能合约存款交易存入 ORC 中，方才有效：/ <节点/网关地址、节点/网关签名、节点/网关押金（允许为负整数，并检查节点/网关状态树中所列出的账户押金大于或等于提现金额）、ID 散列（该节点或网关当前支持的所有 ID 散列）、随机数（Nonce）>。

```
{
    struct NodeStateTransaction {
        address node;
        signature node;
        int change;
        string identity;
        uint nonce;
        }
    }
}
```

图 2.6.1: 自定义节点状态交易域。

ORC 还包含了乐观主义欺诈证明（Optimism Fraud Proof）的标准公共函数，不同于所提出的质疑本身，它可以在不需要交互的情况下，对欺诈状态更新提出质疑和解决[21]。需要强调的是，DRC 和 ORC 欺诈证明是不同的过程（由于对争议项目的真实性存在主观性，因此 DCR 具有高度的交互性，而 ORC 由于使用以太坊（Ethereum）主链作为数据存储层，因此具有高度的客观性。）ORC 欺诈证明依赖于数据的可用性，即所有状态更新在以太坊（Ethereum）主网上都有相应的交易随时可查，主要以调用数据的形式存储[22]。为了保证数据的可用性，同时保持可扩

展性，在以太坊（Ethereum）主链上，我们将包含在交易树中的交易信息的修剪版本存储为调用数据：<TrieID（对三个交易树之一的引用），交易散列（交易的散列代表值）>，其中 TrieID 为 4 个字节，交易散列为 32 个字节，这样每项交易只使用 36 个字节的调用数据。一个标准的以太坊（Ethereum）交易是 247 字节[23]，由于使用了以太坊（Ethereum）状态树，所以需要 21000gas；API 的 optimistic rollup 体系结构已经有效地将交易以 125:1 的比例进行了分批处理，因此，与以太坊（Ethereum）主网目前每秒 15 项交易的吞吐量相比，该体系结构每秒可以处理 1875 个交易，也就是每秒可以调用 1875 个 API 数据。

如果发现 ORC 状态更新存在欺诈行为，那么验证者的保税 THE APIS 代币会有所削减，并将其分配给发现并报告欺诈区块过渡的一方。任何一方都可以向区块链提交 API ORC 区块，只要他们在 ORC 合约中绑定了 10 万个 THE APIS 保税。和 DRC 工厂一样，按照该系统设计要求的超流动性，在 API GC 中下注的代币也可以作为验证者在 ORC 中下注的代币。

如果我们的 optimistic rollup 设计无法满足需求，那么我们将过渡到零知识 rollup 设计，利用修改后的 zkSync 或 Validium，这取决于用户需求的扩展要求[24]。

3 应用范围

API 网络的应用范围主要是 2 个方面：创造更好的终端用户体验以及创造更好的开发者体验。对于终端用户来说，API 允许开发者无缝集成功能，同时保证 100%的正常运行时间，抵抗审查，以及近乎零的平台风险。虽然去中心化网络的正常运行时间和抵抗审查功能已经在公共区块链社区彻底讨论过，但我们认为关于平台风险，仍然没有进行充分讨论。就去中心化应用而言，利用 API 网络的开发者永远不会被垄断、追求利润的公司收取额外费用，这是因为协议创造的是市场，而不是垄断[25]。Metamask 最近在开源许可方面的改变，或许可以进一步证实平台风险这一观点，让开发者质疑未来 Metamask 的集成是否需要收费[26]。通过聚合 API 的中心化竞争者，API 已经可以支持以太坊（Ethereum）的所有查询和索引，由于 IPFS 和 Filecoin 越来越受欢迎，此后不久 API 也将支持 IPFS 和 Filecoin。此外，如

果 API 网关或节点创建了该区块链的 ID，那么任何额外区块链都可以添加到 API 索引和查询协议中。

3.1 IDs

以太坊（Ethereum）和 IPFS 都包含了可索引和查询的兆字节数据，且其数据集每天都在增长。因此，API 引入了 ID 这一概念，这样，API 网关和节点只能索引 API 客户端的相关数据集。ID 是对开发者希望索引的特定以太坊（Ethereum）智能合约或 IPFS 散列的标准化引用。我们可刺激 API 网关和节点索引开发者和用户认为相关的所有 ID；因此，API 已经支持所有容量显著的去中心化金融 ID，这样，我们就可以查询任意自动化市场制造商、去中心化交易所、链上贷款人以及链上聚合器的数据集了。

ID 将通过 API 管理模块上的链外提案来进行请求和添加，该模块与链上 API 管理合约分开。一旦对 ID 提出请求，那么网关和节点就会在 ORC 中进行网关或节点交易，通过将 ID 添加到其注册表中，来表示其支持。该交易的作用是证明他们对 ID 的支持，这使得网关和节点可以相互搜索，提供更完整的数据集，而不需要事先了解对方。这使得网络变得更加分散，因为节点和网关可以根据正常运行时间，对其连接采用标准的内部私有信誉系统，而最初的通信是由希望将更多的 ID 纳入其服务产品的愿望所驱动的。此外，该协议后来还引用了争议解决合约工厂中的 ID，这样，API staker 可以更加深入地了解争议事项

3.2 THE APIS 代币

如前所述，我们将 THE APIS 代币设计成超流体，同时利用于链上管理、争议解决方案以及 optimistic rollup 验证。因此，那些拥有或赚取 THE APIS 代币的人对整个 API 协议保持着话语权，为其行为赚取费用。由于如今大多数代币都未开采，因此，THE APIS 将通过一个由使用量而非流动性本身所衍生的流动性挖掘计划，将其分配给用户。API 用户往往是拥有技术能力的开发者，他们可以对 APiiP 进行投票、解决争议，并部署自己的 API 节点或网关；由于研究透彻的无利害关系问题集，因此，所有可能损害网络的 API 行为者都必须了解 THE APIS 利害关系[27]。

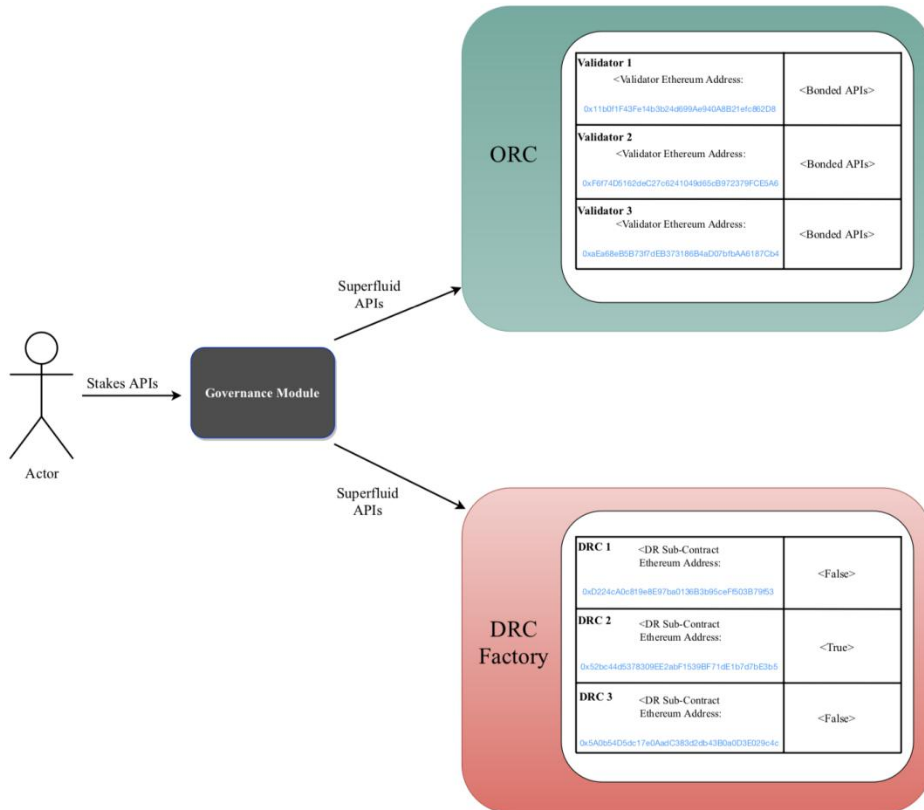

图 3.2.1: 超流体 THE APIS 构成。

3.3 代币发行，社区所有权

在 ORC 状态树中，流动性挖掘账户将进行公开注明。该账户将每周向这些地址发送一次 THE APIS 代币。尽管理事会最终将取消，但目前来说，该账户仍然通过一个九人多签来控制，并由 API 理事会进行监督，之后将由 API 管理合约取而代之，这样 THE APIS 代币持有者可以每周投票来决定是否增加供应。如果他们认为不再需要的话，那么 API 持有人就可以防止未来的通货膨胀，这是因为大多数项目已经实施[28]，且人们已证明其不如市场意识的流动性挖掘，例如 yearn.finance[29]。

如前所述，API 流动性挖掘是通过使用，而非 API 平台本身的流动性来实现的。虽然许多协议选择激励其代币交易和出借，但 API 网络最适用于激励 API 查询和 API

调用，奖励那些将协议用于其预期目的的人，为网络提供的服务支付和接收费用。根据下列函数，第三方客户端开发者和 API 节点与网关将每周获得一次支付：

$$T * \sum_{i=1}^{n} (\sqrt{F}/_{TF})$$

其中，F 为每个第三方开发者向每个节点或网关所支付的费用，TF 为网络上所支付的费用总额，T 为该周网络分配给该程序的代币数量。取所支付费用的平方根之和既为了抑制清洗交易，同时鼓励第三方开发者使用各种网关和节点，激励网络最大限度的去中心化。T 目前的计划是每周 4.7mn THE APIS，一旦管理模块通过提案，即将理事会的权力从理事会中移除，并通过 APiiP 将权力交给 API 社区，那么我们将根据理事会和整个 API 网络的情况进行改变。

4 讨论

在本文中，我们提出了一个可扩展的、去中心化的协议，可将其用于索引和查询公共区块链固有的数据集。该协议基于 REST 和 GraphQL 信息格式和优化等知识，从而确保开发者可以从以太坊（Ethereum）、IPFS 以及所有其他相关区块链中获取数据，并具有正常运行时间保证、超强的市场定价能力，以及主流开发者已经具有丰富经验的格式集成。该协议由两部分组成：网关和节点，尽管一个网关可以是一个节点，反之亦然。双重基础架构的目的是让开发者尽快选择接收端点，同时确保端点的提供者都能得到激励，以符合 API 网络的最佳利益。API 网络是由社区来进行管理的，管理主要集中在“争议解决合约”（该合约受到 Augur 项目方式的主观 oracle 设计的影响）以及“Optimistic Rollup 合约”（该合约主要受到乐观主义（Optimism）项目的影响）。最后，为了鼓励最大限度的社区所有权以及相应的去中心化，API 用户将获得与其使用量成正比的 THE APIS 代币，该系统的设计是为了缓解清洗交易，并鼓励节点和网关的进一步去中心化。本文剩余内容将专门对 REST 和 GraphQL API 进行分析和概述，即每个标准是如何运行的以及其中的权衡方式。

5 附录：REST 和 GraphQL 的对比分析

5.1 REST 的历史与分析

表述性状态传递（REST）[30]是一种软件架构风格。它定义了很多关于信息如何在未连接的、不同的 web 应用之间进行信息的及时传递。REST 应用程序接口（API）目前占据了全球应用程序接口接口中 71%的市场份额[31]，开发人员以给消费者提供很多世界上最流行的 web 服务为基础作为立足点。

任何为 API 交互部署的 REST 架构的 web 应用程序必须遵守五大指导性约束条件。这些约束条件限制了服务器处理和响应客户请求的方式。但是，这样做可以在效率、可扩展性以及简化性方面为网站提供大量的理想属性。约束条件如下：

- **客户端-服务器架构：**其基本原理是在查询数据与数据提供者的用户界面之间存在明显的分离。举一个具体的例子，Facebook 的 API 不应该直接反射自己的前端接口。因此，查询 Facebook API 的第三方开发者不应该强迫与其交互。如 Facebook 的“个人资料页面”端点是一个“用户”的信息，第三方开发人员可以在他们的 web 应用程序的独特用户界面中利用这些信息。

- **无状态：**在请求之间，不应将已经查询过的数据存储在服务器上，这样的话，对数据供应者来说，为任何额外的请求提供服务时的边际成本为零。因此，客户端开发人员的每个请求必须包含服务器为该请求提供服务所需的所有信息。

- **可缓存性：**客户端能够缓存响应。缓存意味着“存储以备后用”，这表示如果客户端已经请求过的信息，客户端可以避免向服务器请求该信息。例如，如果我通过 Facebook 的 API 请求访问某个人的 Facebook 个人资料页面，那么我可以缓存这些信息，这样的话我下次需要的时候就不需要再次发送请求了。

- **分层架构：**客户端通常不能判断客户端是直接与提供商的端点服务器[32]连接还是存在于端点服务器和客户端之间。例如如果服务器端 web 应用程序使用负载均衡，这种技术允许开发者跨多个服务器分发请求，这样的话，就不会有单个服务器被淹没的问题。而且用户可以从该 web 应用请求数据 该

web 应用应该继续与提供响应的服务器架构进行完全的性能通信，从而不需要客户端或服务器体系结构编写任何额外的代码。

- **统一接口：**根据第三方开发者的经验，统一接口是 REST 最重要的组成部分，而且区别于竞争架构风格。统一接口将 web 应用程序的接口与安装启用分离开来，这意味着不管 web 应用程序用来做什么，开发人员都可以用相同的方法与其 API 实现交互。因此，即使他们检索的信息是完全不同的，开发人员还是可以用同样的方法与提供的 REST 的 API 实现交互。REST 实现了这种非常理想的延展性有以下四个限制条件：

  - **资源标识：**REST 不同于之前的 API。因为它是围绕资源而不是方法或者过程。这些资源可以有效地采用任何形式（无论是静态图片还是实时股票价格的转播），但他们通常代表来自业务领域的主体（客户、订单、价格、产品等）。不管资源提供了什么数据，它们必须通过统一资源标识（URI）来体现独特的辨识度。尤其是即使资源本身发生了改变，资源的标识符也必须是持久的。例如，如果 Spotify 音乐播放器更新了给定歌曲的专辑封面，开发人员仍然可以通过 API 在同一个查询中找到，而且对于 API 的开发人员来说，没有任何改变。

  - **通过表示来操作资源：**客户端不能通过服务器端的 API 直接访问服务器端的数据库。如果客户端希望通过服务器端的 API 操作信息，例如更改用户的配置文件图片，那么客户端必须获取该资源的副本，对其进行操作，将更新后的内容发送给服务器，并要求服务器更新其深层次的资源。那么服务器端可以验证这个更改是不是恰当的，从而保护公司的深层数据存储不受恶意编辑。

  - **自我描述信息：**自我描述信息约束要求一个信息（无论是请求还是响应）可以包含足够多的信息，让接收方能够独立理解它。我们将在之后描述 REST API 的实际运用时更详细地介绍这部分内容。

- **超媒体作为应用程序状态的引擎：**任何 RESTful 应用程序都应该可以通过链接达到完全导航。可导航的链接允许端点用户客户端从单一的“基础”URI（如 facebook.com）导航到网站的任何部分，而且，对于请求 API 的开发人员来说，这意味着开发人员不需要查看指定的 API 文档来确定如何获得所需的资源。

- **按需编码：**服务器可以在运行时通过发送可以执行的代码（像 JavaApplets 或 JavaScript）给客户端来扩展客户端功能。此属性与 REST 的可缓存性类似，突出了架构的高性能。

5.2 REST APIs 运用:

如果客户端开发人员希望与 API 交互，开发人员需要放置一个包含一定数量信息的请求：

- **端点：**端点是开发人员发出请求的 URL，它将定义开发人员可以接收的信息类型。端点的结构如下：
  - 端点总是以根源开头，它是开发者与之交互的高级 URL 的典型代表。例如，如果开发人员正在与 Facebook 的 API （Facebook.com，而非 Facebook 子公司的 API）交互，请求总是以 https://graph.facebook.com 开头。
  - 从那里，客户端开发人员必须指定检索所需信息的路径。打个比方，可以将端点的这个子部分看作自动应答机，开发人员通过菜单选项来数字进行导航以到达预期的目的地。例如，如果一个开发人员想要查看某个人所有的 Facebook 好友，路径应该是/user-id/friend。需要注意的是，如果开发者使用 Spotify 音乐播放器的 API 来获取相同的信息，而不是通过链接，其实这看起来是完全一样的。要通过 Spotify 音乐播放器来查看用户在 Facebook 上的好友，开发人员仍

然需要点击用户的个人资料(用户 id)，然后再次点击来查看用户的好友。

- **方法：**该方法定义了客户端开发人员发送给服务器的请求类型，通常是五种类型之一，并允许所有基本 CRUD（增删改查）操作：

  - GET：开发者从服务器上获取资源，如：获取用户的 Facebook 好友。

  - POST：开发人员在服务器上创建一个新资源，如：分享一张照片到用户的 Facebook 个人简介。

  - PUT：开发者覆盖服务器上的一个资源，如：在 Facebook 上更换用户的头像。

  - PATCH：开发人员更新服务器中现有资源的一部分，如：只改变用户个人简介页面中的个人描述。放置和修补非常相似，选择使用哪个是特定的情况，并且这是一个经常争论的话题[33]。

  - DELETE：开发者从服务器上删除一个资源，如：删除一个 Facebook 帖子。

- **报头：**报头向客户端提供关于客户端应用程序如何处理信息的信息，换句话说，允许个人访问的类型。例如，只有用户自己才能通过 Facebook 的 API 更新自己的 Facebook 个人简介，因此客户端应用程序的用户必须验证一个访问许可，该许可证明服务器已经允许他们更改该用户的简介；该访问许可也包含在报头里，以便客户端应用程序可以向其端点用户提供所需的操作。可以在这里[34]找到一个例子列表。

- **数据（或主体）：**数据（或主体）提供客户端希望还原给服务器的信息，并且仅用于执行创建、更新或删除操作。例如，如果用户希望在利用

Facebook API 的客户端应用程序上更改其 Facebook 个人资料图片时，这个消息返回到 Facebook 服务器时必须包含把替代的图片上传。

为了把这些内容结合起来，我们构造了一个查询和响应示例，如下所示，演示了一个简单而精确的与 REST API 的交互：

- **请求：api.example.com/GET/users/10**
- **响应：**

```
{
    "user" : {
          "id"  : 10,
          "name"  :  "Satoshi Nakamoto",
          "nickname"  :  "Seb",
          "height"  : 180,
          "Image_url"  :  "/images/10.jpg"
    }

}
```

图 5.2.1：REST API 响应。

5.3 REST 的优缺点：

既然我们已经阐释了什么是 REST API，以及开发人员如何与 REST API 交互，那么讨论使 REST 如此成功的特征以及威胁到其可持续性的缺点就显得很重要了。

- **优势:**
  - **跨语言和跨框架的灵活性：** REST API 如此成功的一个显著的原因在于它可以跨语言和跨框架操作。无论开发人员如何构建他们的应用程序，REST API 都可为其操作。

- **高可解释性：**事实上，REST API 是基于 URL 的，每个 URL 都提供了一定数量的信息，这使得阐释 REST API 显得非常简单。所有的第三方开发人员需要的只是一个 URL，了解他们将在那里找到什么信息，以及了解如何与那些信息进行交互。

- **服务器端逻辑：**对于一个公司而言，建立一个 REST API，处理它的 URL 驱动性质是极其简单的工作。服务器端开发人员只需定义一个 URL，他们希望在其中得到一些现有的信息，编写如何处理每种类型的请求的逻辑，然后创建端点。然而，随着公司 API 使用规模的扩大，探索结构端点及其提供信息的最佳方式会增加其复杂性，这也是该架构的一个弱点，我们来看下一节。

- 弱势:
  - **多次往返（延时）：**客户经常需要执行多次往返服务器，以获取所有客户需要的信息。每个端点指定固定数量的信息，并且在许多情况下，这些信息只是客户端填充客户端页面所需信息的子集。正如上面的图 5.2.1 所示，如果客户端希望在客户端应用程序中显示他们的图像以及昵称，该怎么办？客户端发送一个获取用户信息的 GET 请求，从而从响应中获得用户图像的位置，然后发送另一个获取图像 URL 的 GET 请求。每次客户端执行一个额外的请求时，就需要再一次往返于服务器，这个过程需要额外的时间，因此也削弱了用户体验。

  - **冗余：**当客户通过 REST API 提出请求时，即使开发人员不需要所有的信息，开发人员将取回所有存储在那里的信息。举个例子，如果一个客户端只对 Facebook 上用户的姓名和年龄感兴趣，那么当客户端查询 Facebook API 时，客户端将得到这个信息，但是也会收到所有与该用户端点信息相关的信息，如用户的个人资料图片，城市住宅，电子邮件地址等。这将导致传递过多的信息，从而导致客户端需要筛选响应的那一部分来增加提供端点用户体验的时间。尽管开发人员已经建立了许多方法来解决 REST 过度的问题，但过度仍然会给客户端

开发人员带来巨大的开销，因为客户端开发人员需要为某些参数设置大量的极端情况。

- **多个端点的安全性：**假定每个端点是静态的，并且只包含一小组信息，端点的规模几乎与服务器希望暴露的信息量成线性关系。因此，流行的 web 应用程序倾向于构建大量的端点。这对于数据安全来说是一场噩梦，因为公司对其服务器端有数百个不同的访问点，而恶意参与者将在每个访问点上搜索漏洞。

- **文档：**REST API 最大的缺点之一是它们是无模式的，这意味着客户端开发人员不清楚在查询 API 时会返回什么样的数据结构。也无法从端点本身了解客户端将接收什么，因此后端服务器开发人员必须维护稳健的文档，以便让用户了解需要接收什么。这可能非常困难，尤其是当服务器端开发人员试图将其 API 归档时，还可能导致新开发人员的接手时间变慢。这虽然是一个可以解决的问题，但使用 REST 时这还是一个令人头痛的问题。

- **更新：**在实践中，前端服务器端团队接触后端服务器团队他的数据需求时，大量的端点被设计在运行中，即在他们之间建立一个端点。这在软件开发过程中产生了大量的开销。前端的每次涉及到显示数据更改的设计迭代，都需要一个后端团队直接参与，这会使 API 变得脆弱并且容易出错。经常更改的 API 很难维护，而且客户端也很难获得所需的数据。当从某些 API 响应中删除字段而且客户端没有意识到它时（客户端只是没有收到警报，并且仍然在旧的 API 版本上运行)，由于丢失数据，那么客户端的程序崩溃的可能性很大。

因此，尽管 REST API 具有显著的优点，并且是至今世界上最通用的 API 类型，但是大规模地应用它还是存在很大的缺点的。这些问题在 10 年 20 年前并不常见，这是由于 API 不那么普遍，而且传输的数据量也很小。但是 REST 的问题对于现代开发者来说变得越来越严重了。

GraphQL 是一个由 Facebook 开发的的开源项目，它用来处理很多这样的问题，同时仍然保持使 REST API 变得如此流行的优势。再进一步研究之后，我们希望能让您确信 GraphQL 实际上是 API 开发的未来。

5.4 GraphQL 的历史和分析

2010 年初，Facebook 开始尝试在手机移动端研发自己的 APP,[35]，Facebook 网络体验功能一直以来都很强大,基于此.他们决定在移动电话上建立 Web 视图,，而不是为其专门开发相应的原生体验,但这样的开发给用户带来了很大的困扰,因为移动电话无论从硬件还是软件均无法满足这种大型 Web 组件所需的带宽和网络需求。

其主要问题在于 Facebook 系统中的 NewsFeed 功能如何实现，因为用户在浏览网页时不仅仅是需要检索某一个贴子或文章.而且还需要实时更新的动态消息.如评论和赞,此外所有这些贴子在 Facebook 系统中都是嵌套和关联的.由于系统本身存在往返和冗长响应的弱点,导致这些实时动态的复杂性 Facebook 无法处理。为了解决此问题,在 2015 年 Facebook 移动团队的四名工程师：NickSchrock.Alex Langenfield、Dan Schafer 和 Lee Byron 联手开发了 GraphQL

GraphQL 是一种开源的数据库查询和操作语言，用户可以通过它检索所需要的信息，　一个 GraphQL 查询是一个发往服务端的字符串。该查询在服务端被解释和执行后返回 JSON 数据给客户端。GraphQL 查询可以直接映射返回的数据，客户端开发人员收到其应用程序所需要的数据，从而能很简单的写出查询所需要的语言。

在 GraphQL 中，服务端负责返回目标数据给客户端，而在 BEST 中，客户端要进行多次访问才能从信息中挑选出所需要的信息，这种方式无形中加重了服务器的负担，客户端的一个请求需要服务器提供多种信息资源才能应对，GraphQL 是强类型的[36]，API 用户会很清楚什么样的数据是可用的，以及它存在的格式，这使得客户端构建查询的方法更便捷，开发人员在后端维护 API 也更容易。

5.5 GraphQL 和 Fetching Tree 相比较

GraphQL 这个名称字面上理解主要用途是优化数据库的 API 结构.但事实并非如此。GraphQL 允许服务器端开发人员将服务器提供的资源和流程建模为领域特定语言。因此，服务器端开发人员可以创建一种专门映射到其数据库结构的查询语言，客户端开发人员必须遵守其命名约定和结构规范查询，这样有利于客户端开发人员执行更高效的查询。

理解 GraphQL 查询原理的最好方法是看其客户端的应用，一般情况下，客户端应用程序是由离散页面组成设计的，这些页面包含一些微小的数据，然后执行级联通过提取所需的数据，为终端用户提供独特的用户体验。例如，让我们假设一下，在客户端 web 应用程序上，配置文件页面如果有用户导入查询信息，web 应用程序

可以把这些查询信息传到服务器的多个端口，从而获取用户所需的信息。开发GraphQL 的基本思路是，在大多数情况下，数据是随机抓取的，这些随机的数据会形成一个树状结构.在一个特定的页面来，这个树的结构是固定的， 来自早期响应的数据包含后续请求的键(如前面提供的 REST 示例中的我的配置文件映像的地址)，并且这些请求之间是直接链接在一起的。因此,如果客户端能把这些所有不同的点归结到一个地方,再将他们的编码作为一个大型抓取树,然后向服务器发送获取树,那么客户端就可以接收用户所有请求中的数据,这样就大大消除了多个经常需要往返的 REST api,由于节省了带宽和延迟服务器响应速度得到了很大的提高。现在我们已经将查询抽象的理解为一个抓取树，下面就让我们研究一下抓取树在 GraphQL 中是如何被定义的。

5.6 GraphQL 请求解析

GraphQL 请求是由客户端至少一个 API 发起的，然后是一系列的后续操作。这些后续操作也即查询，它们只是检索数据，不会改变服务器的任何部分，所有的这些操作都以字段形式存在，这些字段主要分为两种类型：

- 标量类型:标量是服务器最终传给客户端的各个数据片段。可以理解为这些是树状结构中的树叶，存储在这些叶子中的数据是杂乱无序的。
- 标量类型:标量是服务器最终传给客户端的各个数据片段。可以理解为这些是树状结构中的树叶，存储在这些叶子中的数据是杂乱无序的。

给定图形的 GraphQL API 的整个模型可以定义为图形模式，与 REST 比较而言.它是强类型的。每个模式都拥有一个固定的 route 查询类型，它的字段即是 API 的入口点，参见如下示例：

GraphQL 查询中首先是根查询，根查询的对象至少有一个字段，这些字段可以用来引导后续的相关查询。必须指出的是，请求树中的任何字段都可以接受参数，因此可以对请求进行更深层次的参数化。示例如下:

在这里服务器通过检索用户 ID,返回用户的相关信息如名称、昵称、头像等，通过这些可以一次性执行请求指令，而不是像 REST 需要执行两次请求指令，GraphQL 之所以能够实现这种提高的效率，是由于解析器的缘故，这一点是值得提出的。

5.7 GraphQL 解析器注意事项：

在 GraphQL 中，只有一种信息传播方法:通过一系列解析器传播上下文。虽然这听起来很复杂，但它仅仅意味着当客户端检索到上面图 5.6.2 中的’imag 对象.它只解析我所请求的上下文.即与 id 为“10”的用户相关联的图像。每个字段都有一个相对应的解析器。对于标量字段，解析器负责返回客户端看到的实际数据。对于对象字段，解析器返回一个隐藏的数据块，该数据块被转发到对象中包含

的字段的解析器中;这些解析器获取其父对象中隐藏的数据、全局上下文和任何参数，根据这些数据值生成客户端所需要的信息返回到客户端。

现在我们做一个 GraphQ 查询的展示示例，需要注意的是，响应将体现该查询的结构情况，如下图所示

5.8 直接对比

现在，我们对 REST GraphQL 有了更详细的了解，那么我们来比较一下这两种架构的优劣，为什么我们认为在今天众多的应用中，GraphQL 比 REST 更好，以及为什么它会在将来完全占据主导地位。

我们可以将公司的 API 建模为一台自动售货机。传统的模式为，客户按下自动售货机上的一个按钮，只能得到一件物品。那么如查客户需要售货机里的所有物品，那客户就必须多次的按按钮才能解决。这个过程很烦琐耗时。解决这个问题方法有两种，一种方法是可以设计一个特殊用途的按钮，客户只需要按一次按钮就可以获得多个物品。另一种方法是在客户端安装一个专用按钮,客户可以从自动售货机获取大量的信息但却不能控制这些信息，,而这些过多的信息在某些情况下会大增加带宽和延迟。而 GraphQL 解决了上述自动售货机另一种方法带来的不足:自动售货机允许客户端精确地按下一个智能按钮，一次性提供客户端需要的一切。

GraphQL 通过一个智能端点来解决这些问题，这个端点是智能的而不是愚钝的，关键的好处在于，智能端点能够将复杂的查询形成数据输出给客户端正确的需求。GraphQL 层从本质上讲像一个桥梁联接在客户端和数据源两端。接收客户端需求并根据客户端的需求准确获取必需的数据。GraphQL 查询方法从根本上解决了大量大规模应用程序开发问题.

可以想象，GraphQL 解决了前面提到的 REST 的大部分主要弱点，如下：

- 多次往返（延迟）：现在客户端可以将复杂的查询推到服务器，服务器从这些查询中准确检索出客户端所需的所有信息并一次性返回，再不需要多次往返返回信息了。
- 冗长：使用 GraphQL，客户端只返回客户端请求的信息，不再返回不相关的信息。
- 多个端点的安全性：使用 GraphQL，服务器只需要保护一个端点，不需要像之前得保护多个端点，使后端安全保护工作更简单。
- 文档：因为 GraphQL 是强类型的，自动创建文档很容易;故而它的很多服务都可以为后端开发人员完成这项工作。另外还有一些很强的仪表板，后端开发人员可以通过它来试验不同的查询，还可以查看到后端服务器如何响应这些查询。

- 更新：通过这种方式，GraphQL 完美实现了向后兼容，它可以很容易添加其他字段到服务器中，即在保留旧产品功能的同时添加新的功能，而不影响现有客户端的使用。而且也没有必要再增加新的版本，从而大大简化了版本的控制。

如上所述，这些都是 GraphQL 最有价值的方面——它不但保留了 REST 优秀的功能，而且还改进了它这么多的弱点。这是一次突破，是一次对一个广泛应用系统的完美补充，这就是它得到如此迅速普及的原因。

从现有的 REST API 切换到 GraphQL API 是很容易的，而且用于维护 GraphQL 系统工具越来越多且成本很低，后续切换会更简单容易。

尽管如此，GraphQL 还远远不是一个完美的系统。实际它与 Kubernetes[37]很类似，它之所以比 REST 有这么多的优点，归根结底还在于它比 REST 的系统更复杂.

5.9 GraphQL 的不足之处

GraphQL 通过优化适应了现在应用构建的变化趋势 现在世界的发展超势为微服务，这将意味着大型应用程序会被拆解为多个独立的组件，这些组件通过 api[38]进行信息传递 。正如我们所说过的，REST 的规模越来越大，这是 microservices 的一大缺点。由于这种解耦体系结构的特殊性，服务器后端需要多次的 API 调用，数据模式也需进行多次的更新，这些都是 REST 需要考虑和解决的问题。

因此，重视 GraphQL 在服务微服务架构时所面临的问题是很必要的，对于一些琐碎的用例，它还存在过度设计的可能。

- 架构设计
  - 微服务体系架构与所有多服务体系架构一样，在服务端有多个后台应用在运行，且有多个服务在协同工作。为了实现 GraphQL 的强大功能，后端开发人员需要将所有这些服务集中在一个统一的模式下，这样做的后果是资源重复，且资源之间存在相互冲突的可能。
  - 另一种解决方案来自于 GraphQL 的官方推荐，即采用绑定的形式，后端开发人员将每个微服务中不同的 GraphQL 架构拼接到一起，在过去这些操作都需要开发人员手动完成，无形中使工作更加复杂，但是最近随着 ApollFederation 的推出，这种操作的复杂性大大降低。因此，团队研究方向已转为在保留 GraphQL 方法的情况下，从多个服务获取信息的解决方案
- 认证和授权
  - 如上所见，GraphQL 提供的是多层服务。顶层是 GraphQL 服务器本身，负责从客户端获取请求并处理响应。顶层下是中间层，负责处理从

GraphQL 服务器到各种微服务的数据信息以及数据层，微服务从该数据层获取所需数据再返回到 GraphQL 服务器。

  - 因此，对于授权开发人员来说，他可以首先权衡体系架构方式，然后决定用户权限，看用户在 GraphQL 服务器级别或微服务的哪种层级使用何种资源，其实这个问题在处理起来其复杂程度远远超于预期。让我们假设一下，如果所有的操作都授权在服务器层级上进行，这就意味着其它的服务端不要有任何漏洞，但显然这是不可能的，那如果把授权建立在其它的服务层级上决定，那由于服务端众多，每个服务逻辑各有不同，需要为每一个请求配备相应的端口，这就从很大程度上增加了复杂性，最好的办法是采用一种混合方法，在微服务端只对一些必要的服务进行二次检验，但实施起来也很不容易。

- 路由设计
  - 当请求传入系统时，GraphQL 服务器负责将该请求路由到它可以获取正确信息的端口。如果服务器只拥有一个后端服务，那这是很容易做到的，因为它可以访问所有的信息，但是如果情况不是这样(大型微服务部署的典型情况)，那么有效地路由请求就成了一个待解决的问题。
  - 在这里，有一种解决方案那就是 GraphQL 模式拼接，它的方法是通过各种更新的解析器来进行，它通过解析器将各种请求片段路由到相应的微服务，在微服务将各种服务模式拼接在一起。在这种情况下，如果各模式类型间不存在冲突（数据模型更新）则不需要手动干预，GraphQL 本身有一些很好的特性可以帮助拼接完成，比如开发人员可以定义执行请求所需的预期时间，这样服务器就可以据此有效地划分优先级，但是此方法最大的缺点是需要大量的手工工作来处理请求路由，这是每个 GraphQL 系统中普遍存在的一个问题，处理大量的并发需求到服务，而且还能保证可靠和高效，实在不是一件简单的事情。
- 错误处理
  - 在典型的 REST 中，服务器对请求给出的响应是反馈请求的结果，如果请求成功，客户端会收到 200 的代码，如果系统未找到资源，客户端会收到 404 的代码，而在 GraphQL 中，情况并非如此，不论请求成功与否，客户端都会收到 200 的代码，一些错误的信息也包含在返回对象中一并返回。这就意味着开发人员必须为这些错误信息编制特定的代码，这种工作很乏味很繁琐，整个系统也会更复杂。

因此，尽管 GraphQL 存在诸多的优点，但是开发人员在生产实现时仍然会面临很多重大的问题，然而，我们认为，任何新技术的研发在投放市场时都会经历各种各样的磨合.这个过程是必需的，而且 GraphQL 的设计也似乎没有哪个方面会影响它的成功应用，因为它架构的复杂性它能够处理更复杂的用例，而且就像 Kubernetes 一样，要在微服务体系架构上广泛使用对用户有益的 GraogQL 版本，可能还会需要很长的时间，但一旦实现，它将会成为市场主导。我们相信，随着世界云和微服务架构的发展，GraphQL 能够很好的利用自身架构，在 API 生态系统中占据主导地位，然而，需要特别说明的是，REST 仍然是主要的解决方案。

6 参考书目

[1] Wanseob-Lim. "Zkopru (Zk Optimistic Rollup) for Private Transactions." *Ethereum Research*, 21 July 2020, https://ethresear.ch/t/zkopru-zk-optimistic-rollup-for-private-transactions/7717

[2] Bakst, Andrew. "Bizantine's Law - Andrew Bakst." *Medium*, 18 July 2020, medium.com/@apbakst/bizantines-law-c9bc93529e89.

[3] "Secure Multi-Party Computation: Theory, Practice and Applications." *ScienceDirect*, 1 Feb. 2019, www.sciencedirect.com/science/article/pii/S0020025518308338. & "What Is Fully Homomorphic Encryption." *Inpher*, https://www.inpher.io/technology/what-is-fully-homomorphic-encryption.

[4] Voell, Zack. "Decentralized Exchange Volumes Up 70% in June, Pass $1.5B." *CoinDesk*, 1 July 2020, https://www.coindesk.com/decentralized-exchange-volumes-up-70-in-june-pass-1-5-billion.

[5] Wikipedia contributors. "Elliptic Curve Digital Signature Algorithm." *Wikipedia*, 17 Aug. 2020, https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.

[6] Yang, Junha. "Blockchain Light Client - CodeChain." *Medium*, 3 Feb. 2020, https://medium.com/codechain/blockchain-light-client-1171dfa1269a.

[7] Bengtsson, Ivar and Fichter, Michael. "Modeling and Optimizing Transaction Fees in a proof-of-stake cryptocurrency." 2018, http://kth.divaportal.org/smash/get/diva2:1218593/FULLTEXT01.pdf.

[8] Blocknative. "Evidence of Mempool Manipulation on Black Thursday: Hammerbots, Mempool Compression, and Spontaneous Stuck Transactions." *BlockNative*, https://blog.blocknative.com/blog/mempool-forensics.

[9] Young, Joseph. "Why A Mysterious Ethereum User Paid $2.6 Million To Send $130 Of Crypto." *Forbes*, 10 June 2020, https://www.forbes.com/sites/youngjoseph/2020/06/10/why-a-mysterious-crypto-user-paid-26-million-to-send-merely-130-in-ethereum/#485be9b9588a

[10] Pourmajidi, William and Miranskyy, Andriy. "Logchain: Blockchain-assisted Log Storage." 22 May, 2020, https://arxiv.org/pdf/1805.08868.pdf

[11] "Global Cloud Database and DBaaS Market (2020 to 2025) - Increase in the Growth of NoSQL Database Provides Opportunities - ResearchAndMarkets.Com." *Business Wire*, 17 Mar. 2020, www.businesswire.com/news/home/20200317005638/en/Global-Cloud-Database-DBaaS-Market-2020-2025.

[12] "Usability: A Part of the User Experience." *The Interaction Design Foundation*, 28 July 2020, https://www.interaction-design.org/literature/article/usability-a-part-of-the-user-experience.

[13] Zhao, Yuanyuan, Sturman, Daniel, and Bhola Sumeer. "Subscription Propagation in Highly-Available Publish/Subscribe Middleware." https://www.researchgate.net/profile/Yuanyuan_Zhao17/publication/221461384_Subscription_Propagation_in_HighlyAvailable_PublishSubscribe_Middleware/links/5640600208ae34e98c4e7d50/Subscription-Propagation-in-Highly-Available-Publish-Subscribe-Middleware.pdf

[14] "Optimistic Rollups." https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/optimistic_rollups/

[15] Giroux, Marc-AndrÉ "On GraphQL Schema Stitching & API Gateways - Marc-André Giroux." *Medium*, 24 Oct. 2018, https://medium.com/@__xuorig__/on-graphql-schema-stitching-api-gateways-5dcb579fa90f.

[16] "Web Cryptography API." *Wikipedia*, 7 Aug. 2020, https://en.wikipedia.org/wiki/Web_Cryptography_API

[17] Compound-Finance. "Compound-Finance/Compound-Protocol." *GitHub*, https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/GovernorAlpha.sol.

[18] Peterson, Jack, Krug, Joseph, Zoltu, Micah, Williams, Austin, and Alexander, Stephanie "Augur: A Decentralized Oracle and Prediction Market Platform (v2.0)." Whitepaper, https://augur.net/whitepaper.pdf

[19] “Fuel Labs.” *Medium*, https://medium.com/@fuellabs.

[20] “Patricia-Tree.” *Ethereum Wiki*, https://eth.wiki/en/fundamentals/patricia-tree.

[21] Ethereum-Optimism. “Ethereum-Optimism/Optimism-Monorepo.” *GitHub*, https://github.com/ethereum-optimism/optimism-monorepo.

[22] “When Should I Use Calldata and When Should I Use Memory?” *Ethereum Stack Exchange*, 30 Aug. 2019, https://ethereum.stackexchange.com/questions/74442/when-should-i-use-calldata-and-when-should-i-use-memory.

[23] Wood, Gavin “Ethereum: A Secure Decentralised Generalised Transaction Ledger Petersburg Version” 2020-06-08, https://ethereum.github.io/yellowpaper/paper.pdf

[24] **zkSync**: Gluchowski, Alex. “Introducing ZkSync: The Missing Link to Mass Adoption of Ethereum.” *Medium*, 18 June 2020, https://medium.com/matter-labs/introducing-zk-sync-the-missing-link-to-mass-adoption-of-ethereum-14c9cea83f58.
Validium: Kirejczyk, Market, Szlachciak, Piotr, Jelski, Krzysztof, Maretskyi, Dmytro, Wiech, Arleta, Charczuk, Joanna, and Kirejczyk, Natalia “Zero-Knowledge Blockchain Scalability.” Summer, 2020, https://ethworks.io/assets/download/zero-knowledge-blockchain-scaling-ethworks.pdf.

[25] Alex, Chris. “Balancer Thesis.” *Placeholder*, 20 July 2020, https://www.placeholder.vc/blog.

[26] The Block. “Popular Ethereum Wallet MetaMask Adopts New Software License as Firm Eyes Commercial Opportunities.” *The Block*, 21 Aug. 2020, https://www.theblockcrypto.com/linked/75751/metamask-new-software-license-commercial.

[27] Martinez, Julian. “Understanding Proof of Stake: The Nothing at Stake Theory.” *Medium*, 22 Jan. 2020, https://medium.com/coinmonks/understanding-proof-of-stake-the-nothing-at-stake-theory-1f0d71bc027.

[28] “Synthetix | Decentralised Synthetic Assets.” *Synthetix*, https://www.synthetix.io/. Accessed 22 Aug. 2020.

[29] Substreight. “YIP 30: YFI Inflation Schedule.” *Yearn.Finance*, 2 Aug. 2020, https://gov.yearn.finance/t/yip-30-yfi-inflation-schedule/1439.

[30] Fielding, Roy “Architectural Styles and the Design of Network-based Software Architectures.” UCI, 2000, https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[31] Elements, Cloud. “The State of API Integration 2019 | Cloud Elements.” *Cloud Elements*, https://offers.cloud-elements.com/the-state-of-api-integration-2019#:%7E:text=The%20State%20of%20API%20Integration%202019%20Report&text=Our%202019%20State%20of%20API,industry%20is%20heading%20in%202019.%20Accessed%2022%20Aug.%202020.

[32] “Front End and Back End.” *Wikipedia*, 22 June 2020, https://en.wikipedia.org/wiki/Front_end_and_back_end.

[33] rapidAPI Staff “What’s the Difference between PUT vs PATCH?” *rapidAPI, 17 August 2020,* https://rapidapi.com/blog/put-vs-patch/

[34] “HTTP Headers.” *MDN Web Docs*, 27 Apr. 2020,
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.

[35] “TechCrunch Is Now a Part of Verizon Media.” *TechCrunch*, 29 Jan. 2014,
https://techcrunch.com/2014/01/29/one-app-at-a-time/.

[36] “Strongly Typed.” *Techopedia.Com*,
https://www.techopedia.com/definition/24434/strongly-typed.

[37] “Production-Grade Container Orchestration.” *Kubernetes*,
https://kubernetes.io/. Accessed

[38] “Microservices.” *Martinfowler.Com*,
https://martinfowler.com/articles/microservices.html.

[39] “Apollo Federation.” *Apollo Blog*,
https://www.apollographql.com/blog/apollo-federation-f260cf525d21/.