IPFS技术详解2

1.IPFS核心技术

如同区块链作为集大成者，汇聚了分布式存储、密码学、点对点通信等技术一样，IPFS也是对DHT、Kademila、Merkle DAG、BitTorrent、Git、SFS等技术的整合和系列应用。

- IPFS身份技术——Kademila
- IPFS路由技术——DHT
- IPFS关键数据结构——Merkle DAG
- IPFS通信技术——BitSwap
- IPFS文件版本控制——Git
- IPFS文件系统——SFS

2.DHT和Kademila

DHT和Kademila是实现了快速的基于内容的寻址——快速定位数据。

2.1 DHT

分布式哈希表（distributed hash table，缩写DHT）是分布式计算系统中的一类，可由键值来唯一标示的信息按照某种约定/协议被分散地存储在多个节点上，这样也可以有效地避免“中央集权式”的服务器的单一故障而带来的整个网络瘫痪。实现DHT的技术/算法有很多种，常用的有：Chord, Pastry, Kademlia等。

DHT分布式哈希表

- 哈希表被分割成不连续的块，每个节点被分配给一个属于自己的哈希块（称为区间表），并成为这个哈希块的管理者。
- 通过哈希函数，一个对象的名字或关键词被映射为128位或160位的散列值。

DHT网络的基本思想

- 每一份资源都由一组关键字进行标识。
- 系统对其中的每一个关键字进行Hash，根据Hash的结果决定此关键字对应的那条信息（即资源索引中的一项）由哪个用户负责储存。由此，生成哈希表。
- 用户搜索的时候，用同样的算法计算每个关键字的Hash，然后从哈希表获得该关键字对应的信息存储位置（具体查看关键字定位），并迅速定位资源。

DHT关键字定位

- DHT通过分布式散列函数，将输入的关键字唯一映射到某个节点上，然后通过某些路由算法同该节点建立连接。
- 每个节点并不需要保存整个系统的节点视图信息，只在节点中存储其邻近的几个后继节点信息。当一个节点收到一个查询操作时，如果它发现所查询的标识不在自己关联的区间内，那么该节点将会把该查询发送给其存储节点信息表中它认为最靠近目标的邻居。
- 每次转发都能更进一步地接近数据源。因此较少的路由信息就可以有效地实现到达目标节点。

DHT特性

- 离散性：构成系统的节点并没有任何中央式的协调机制。
- 伸缩性：即使有成千上万个节点，系统仍然应该十分有效率。
- 容错性：即使节点不断地加入、离开或是停止工作，系统仍然必须达到一定的可靠度。

2.2 Kademlia

Kademlia是一种通过 DHT 的协议算法，它是由Petar和David在2002年为P2P网络而设计的。Kademlia规定了网络的结构，也规定了通过节点查询进行信息交换的方式。

Kademlia提供：

- 通过大量网络进行高效查询：查询平均联系人O(log2N)节点。（例如，20跳10万个节点的网络）
- 低协调开销：优化数量的控制消息发送到其他节点。
- 抵抗各种攻击，喜欢长寿节点。
- 在对等应用中广泛使用，包括Gnutella和BitTorrent，形成了超过2000万个节点的网络。

Kademlia网络节点之间使用UDP进行通讯。参与通讯的所有节点形成一张虚拟网（或者叫做覆盖网）。这些节点通过一组数字（或称为节点ID）来进行身份标识。节点ID不仅可以用来做身份标识，还可以用来进行值定位（值通常是文件的散列或者关键词）。

当我们在网络中搜索某些值（即通常搜索存储文件散列或关键词的节点）的时候，Kademlia算法需要知道与这些值相关的键，然后逐步在网络中开始搜索。每一步都会找到一些节点，这些节点的ID与键更为接近，如果有节点直接返回搜索的值或者再也无法找到与键更为接近的节点ID的时候搜索便会停止。

这种搜索值的方法是非常高效的：与其他的分布式哈希表的实现类似，在一个包含n个节点的系统的值的搜索中，Kademlia仅访问O(log(n))个节点。

Kademlia简称为Kad,它使用了一个精妙的算法，来计算节点之间的"距离" (这里的距离不是地理空间的距离，而是路由的跳数)，这个算法就是XOR操作(异或)，因为这个操作和距离的计算类似：

- (A ⊕ B) == (B ⊕ A): XOR 符合“交换律”，具备对称性。A和B的距离从哪一个节点计算都是相同的。
- (A ⊕ A) == 0: 反身性，自己和自己的距离为零。

- (A ⊕ B) > 0: 两个不同的 key 之间的距离必大于零。
- (A ⊕ B) + (B ⊕ C) >= (A ⊕ C): 三角不等式, A经过B到C的距离总是大于A直接到C的距离。

Kad使用160位的哈希算法（比如 SHA1），完整的 key 用二进制表示有160位，这样可以容纳2160个节点，可以说是不计其数了。

Kad把 key 映射到一个二叉树，每一个 key 都是这个二叉树的叶子。

Kademlia协议： Kademlia协议共有四种消息。

- PING消息: 用来测试节点是否仍然在线。
- STORE消息: 在某个节点中存储一个键值对。
- FIND_NODE消息: 消息请求的接收者将返回自己桶中离请求键值最近的K个节点。
- FIND*VALUE消息: 与FIND*NODE一样，不过当请求的接收者存有请求者所请求的键的时候，它将返回相应键的值。

每一个RPC消息中都包含一个发起者加入的随机值，这一点确保响应消息在收到的时候能够与前面发送的请求消息匹配。

3.Merkle DAG

Merkle DAG实现了版本控制的对象存储——数据存储。

3.1 Merkle Tree

Merkle Tree，通常也被称作Hash Tree，顾名思义，就是存储hash值的一棵树。Merkle树的叶子是数据块(例如，文件或者文件的集合)的hash值。非叶节点是其对应子节点串联字符串的hash。

- 哈希算法（Hash）又称哈希函数、散列技术、摘要。
- 哈希算法是可以将任意长度的数据映射成固定长度的数据的函数，对一份数据进行哈希运算的结果一般都是唯一的
- 只要原文发生改变，哈希运算的结果就会完全不一样。
- 哈希逆运算在数学上是不可能的。
- 主要用于数字签名、错误检验等场景。
- 哈希碰撞：对不同的数据进行哈希运算产生相同的结果。概率极小
- 常见的hash算法有MD5以及SHA系列。

Merkle Tree的特点：

- MT是一种树，大多数是二叉树，它具有树结构所有的特点；
- MT的叶子节点的值可以是小数据块的hash或小数据块；
- 非叶子节点的值都是其子节点值的hash结果。

3.2 Merkle DAG

Merkle DAG的全称是 Merkle directed acyclic graph（默克有向无环图），它是在Merkle tree基础上构建的。Merkle DAG跟Merkle tree很相似，区别在于：Merkle DAG不需要进行树的平衡操作，非叶子节点允许包含数据等。

IPFS的Merkle DAG是由Git改造而来的，定义了一组对象：

- Blob-一个大小可变的数据块（等于或小于256KB）。Blob对象代表一个文件且包含一个可寻址的数据单元，用来存储用户的数据。Blob没有links。
- List-块或者其他链表的集合。List对象代表着由几个blob对象连接而成的大文件或者重复数据删除文件。List中包含有序的blob序列或者其他list 对象。
- Tree-块、链表或者而其他树的集合。Tree代表一个目录，一个名字到哈希值的映射。哈希值则代表了bolbs、lists和其他trees。
- Commit-任何对象在版本历史记录中的一个快照。他同样连接着父提交。

Merkle DAG拥有如下的功能：

- 内容寻址：使用多重哈希来唯一识别一个数据块的内容
- 防篡改：可以方便的检查哈希值来确认数据是否被篡改
- 去重：由于内容相同的数据块哈希是相同的，可以很容去掉重复的数据，节省存储空间

4.BitSwap

BitSwap实现了点对点的数据交换——数据传输。

点对点的数据传输：

- 数据不是存储在中心化的服务器，而是存储在各个节点上。
- 节点是对等的，节点即是客户端，又充当服务器的角色。

4.1 Bittorrent

特点：

- 一种点对点传输的网络协议。
- 对于一个文件，下载的用户数越大，下载速度就越快。
- 部分的网络拥堵或服务器宕机并不会对整个传输链路造成太大的影响。
- 常见的BT软件有：迅雷、QQ旋风、比特精灵等。 存在问题：

- 节点只索取不贡献，导致数据交换的效率极速下降，甚至点对点网络瘫痪。

- 容易遭受女巫攻击，即恶意机器不断请求，占用本机上传带宽。

4.2 BitSwap

IPFS在BitTorrent的基础上实现了p2p数据交换协议：BitSwap协议。 IPFS每一个节点都维护了两个列表：

- 已有的数据块（have_list）
- 想要的数据块（want_list）

当两个节点建立连接后，他们会根据hava*list和want*list互通有无。跟BitTorrent不一样的是：BitSwap获取数据块的时候不限于从同一个torrent里面。也就是说BitSwap可以从不属于本文件的其他文件获取数据块(只要数据块的哈希值一样，数据内容必然是一样的，同样的数据块只会保留一份)。从全局考虑，这使得BitSwap的效率相比于BitTorrent更高。 对于p2p网络，有一个很重要的问题是：如何激励大家分享自己的数据？IPFS改进了Bittorrent，实现了自己的特殊的数据分享策略BitSwap。BitSwap由信用体系、实现策略、账单组成。

信用体系：

- 建立一个指标：信用值，节点间通过信用值来决定是否给对方传输数据。
- 发送数据给其他节点增加信用值，从其他节点接受数据降低信用值（线性的关系）。
- 如果一个节点只收不发，那么信用值会降到很低，以至于被其他节点忽略。
- 乐于分享的节点，容易得到其他节点的分享，从而创建高效的网络

账单：

- 用来记录节点与其他节点通信数据的收发情况，每个节点都有自己的账单
- 记录的数据最后用来计算该节点的信用值
- 数据交换之前，两个节点互换账单，先确定账单是否匹配（防篡改），再计算信用值（确认是否可信）
- 账单的存在基本杜绝了女巫攻击

实现策略：

- 信用值用r来表示：

$$\mathrm{r}=\frac{bytes\_send}{byte\_recv+1}$$

- 根据信用值计算给该节点的发送数据的概率：

$$p(\mathrm{send}\,|\,\mathrm{r})=1-\frac{1}{1+e^{6-3r}}$$

这是一个sigmod函数，如果节点的信用值下降到一个临界值，那么发送数据的概率就会急剧下降。

5.Git

Git(版本化)实现了文件的历史版本控制器，并且让多节点使用保存不同版本的文件。

版本控制系统提供了对随时间变化的文件进行建模的设施，并有效地分发不同的版本。流行版本控制系统Git提供了强大的Merkle DAG对象模型，以分布式友好的方式捕获对文件系统树的更改。

- 不可更改的对象表示文件（blob），目录（树）和更改（提交）。
- 通过加密hash对象的内容，让对象可寻址。
- 链接到其他对象是嵌入的，形成一个Merkle DAG。这提供了很多有用的完整和work-flow属性。
- 很多版本元数据（分支，标示等等）都只是指针引用，因此创建和更新的代价都小。
- 版本改变只是更新引用或者添加对象。
- 分布式版本改变对其他用户而言只是转移对象和更新远程引用。

6.SFS

SFS(自我认证认文件系统)实现了

- 分布式信任链。
- 平等共享的全局命名空间。

SFS引入了一种自我建构技术—注册文件： 寻址远程文件系统使用以下格式：

`/sfs/<Location>:<HostID>` `Location:代表的是服务网络地方`
`HostID = hash(public_key || Location)`

因此SFS文件系统的名字认证了它的服务，用户可以通过服务提供的公钥来验证，协商一个共享的私钥，保证所有的通信。所有的SFS实例都共享了一个全局的命名空间，这个命名空间的名称分配是加密的，不被任何中心化的body控制。

参考：

```
1. [https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf](IPFS白皮书)

2. [http://colobu.com/2018/03/26/distributed-hash-table/](DHT分布式哈希表)
```

推荐：

```
1.[https://protocol.ai/projects/](协议实验室主页)
2.[https://gguoss.github.io/2017/05/28/ipfs/](IPFS(中文白皮书))
```