IPFS - 内容寻址的版本化点对点文件系统(草稿3)翻译

IPFS - Content Addressed, Versioned, P2P File System (draft 3)

摘要

星际文件系统是一个点对点分布式文件系统, 旨在使用相同的文件系统连接所有的计算设备。在某种情形下，IPFS很类似Web，不过IPFS可以被看作一个独立的使用Git仓库来交换对象的BitTorrent集群。换句话说，IPFS提供了一个高吞吐量的以内容可寻址的块存储模型，具有内容寻址的超链接。这便形成了广义的Merkle DAG数据结构，这种数据结构可以构建成一个文件版本系统，区块链，甚至一个永久性Web。IPFS使用分布式哈希表，激励化的块交易模型和自我认证的命名空间。IPFS没有单独的故障节点，节点之间不需要完全信任基础。

1.介绍

关于构建全球分布式文件系统，已经有许多尝试。一些系统已经取得了重大的成功， 而一些却完全失败了。在学术尝试中， AFS[6] 就是成功的例子，并且得到广泛的应用。 然而，其他的[7, ?] 却没有成功。学术界之外，最成功的系统是面向音视频媒体的点对点文件共享系统。 最值得注意的是， Napster, KaZaA 和BitTorrent[2] 部署的文件分发系统可以支持1亿用户同时在线。即使今天， BitTorrent 也维持着每天千万节点的活跃数[16]。 相比学术文件系统，这些应用获得了更多的用户和文件分发，然而这些应用并没有被设计为基础设施。虽然这些应用已经取得了成功的回报，但目前还没有出现为全球提供低延迟和分散式分发的广义的文件系统。 也许是因为HTTP这样“足够好”的系统已经存在。到目前为止，HTTP作为最成功的“分布式文件系统”的协议已经大量部署，再与浏览器相结合，具有巨大的技术和社会影响力。它已经成为互联网传输文件的事实标准。然而，他没有采用最近15年的发明的数十种先进的文件分发技术。 从一方面讲， 由于向后兼容的限制和当前模式的强烈投入， 进化目前的Web基础架构几乎不可能。但从另一个角度看，自从出现了HTTP，新的协议已经出现并被广泛使用。目前的难题缺是升级设计：不会降低用户体验情况下引入新功能，以增强目前的HTTP Web。 长期使用HTTP的业界已经消失了，因为移动小文件是非常便宜，即使是具有大量流量的小型组织。但是，随着新的挑战，我们正在进入数据分发的新时代：

- （a）托管和分发PB级数据集
- （b）计算跨组织的大数据
- （c）按需量或实时媒体流量大规模高清晰度定义
- （d）大规模数据集的版本化和链接
- （e）防止重要文件的意外消失等。

许多这些可以归结为“大量数据，无处不在”。由于关键功能和带宽问题，我们已经放弃了不同数据的HTTP分发协议。下一步是使它们成为一部分的Web本身。

2.背景

本节回顾了IPFS所采用的点对点系统的重要技术特性。

2.1 分布式哈希表

分布式哈希表(DHTs)广泛的使用于定位和维持点对点系统的元数据。例如BitTorrent的MainlineDHT技术跟踪torrent集群的对等节点。

2.1.1 Kademlia DHT

Kademlia[10] 是一个广泛流行的DHT，提供了一下特点:

1. 大规模网络下高效查询：查询平均访问O(log2N)节点。(例如，1000万节点的网络需要20跳查询)
2. 低协调开销：优化控制消息发送到其他节点的数量。
3. 使用长时间在线节点来抵抗各种攻击。
4. 广泛使用于点对点应用中，包括Gnutella和BitTorrent，形成了超过2000万个节点的网络[16]。

2.1.2 Coral DSHT

虽然一些对等文件系统直接在DHT中存储数据块，“不应该存储在节点的数据存储在节点会浪费存储和带宽”[5]。Coral DSHT通过以下三种方式扩展了Kademlia：

1. Kademlia将值存储在距离其key最接近(使用XOR-distance)的节点中。这不考虑应用数据的局部性，忽略“远处”可能已经拥有此数据的节点，并强制“最近”节点存储它，无论它们是否需要。这将浪费了大量的存储和带宽。相反, Coral 存储了地址， 该地址的对等节点可以提供相应的数据块。
2. Coral将DHT API的get_value(key)换成了get_any_values(key)（DSHT中的“sloppy”）。这将依旧工作，是因为Coral用户只需要一个（工作）的对等节点，而不是完整的列表。作为回报，Coral可以仅将子集分配到“最近”的节点，避免热点（当密钥变得流行时，重载所有最近的节点）。
3. 另外，Coral根据区域和大小组织了一个称为clusters的独立DSHT层次结构。这使得节点首先查询其区域中的对等体，“查找附近的数据而不查询远程节点”[5]并大大减少查找的延迟。

2.1.3 S/Kademlia DHT

S/Kademlia[1]使用以下方式扩展了Kademlia来防止恶意的攻击:

1. S/Kad 提供了方案来保证NodeId的生成，防止Sybill攻击。它需要节点产生PKI公私钥对，从中获得他们的Identity，并彼此间签名。一个方案使用POW工作量证明，使得生成Sybills攻击的成本高昂。
2. S/Kad 节点在不相交的路径上查找， 即使网络中存在大量的不诚实节点，也能确保诚实节点可以互相链接。即使网络中存在一半的不诚实节点，S/Kad 也能达到85%的成功率。

2.2块交换 - BitTorrent

BitTorrent[3] 是一个非常成功的点对点共享文件系统，它可以在存在不信任的对等节点（群集）的协作网络中分发各自的文件数据片。从BitTorrent和它的生态系统的关键特征中 IPFS得到启示如下：

1. BitTorrent的数据交换协议使用了一种bit-for-tat的激励策略， 可以奖励对其他节点做出贡献的节点，惩罚只吸取对方资源的节点
2. BitTorrent对等节点跟踪文件的可用性，优先发送稀有片段。这减轻了seeds节点的负担， 让non-seeds节点有能力互相交易。
3. 对于一些剥削带宽共享策略， BitTorrent的标准tit-for-tat策略是非常脆弱的。 然而，PropShare[8]是一种不同的对等节点带宽分配策略， 可以更好的抵制剥削战略， 提高群集的表现。

2.3版本控制系统 - Git

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。

1. 不可更改的对象表示文件（blob），目录（tree）和更改（commit）。
2. 对内容加密哈希，让对象可寻址。
3. links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。
4. 很多版本元数据（分支，标示等）都只是指针引用，因此创建和更新的代价都小。
5. 版本改变只是更新引用或者添加对象。
6. 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。

2.4自验证文件系统 - SFS

SFS[12,11]提出了两个引人注目的实现（a）分布式信任链，和（b）平等共享的全局命名空间。SFS引入了一种建构自我认证文件系统的技术：使用以下格式`/sfs/<Location>:<HostID>`寻址远程文件系统，Location是服务器网络地址，并且:`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。

3.IPFS设计

IPFS是一个对等系统，没有节点拥有特权。IPFS节点在本地存储IPFS对象。节点之间彼此连接并且传送对象。IPFS协议根据不同的功能划分成以下子协议栈:

1. 身份 - 管理节点的创建和认证。
2. 网络 - 通过可配置化的不同的底层网络协议，管理节点之间连接。
3. 路由 - 维护定位特定对等节点和对象的信息表。应对本地和远端查询。默认是DHT，可以替换。
4. 交换 - 一个新奇的块交换协议(BitSwap)，用来管理块分发效率, 模拟市场, 轻微刺激数据复制。交易策略可以替换。
5. 对象 - 一个对应链接的以内容寻址的不可变的Merkle DAG对象。可以表示任意数据结构，例如，文件层级结构和通信系统。
6. 文件 - 受Git启发的文件版本层级系统
7. 命名 - 自我认证的可变命名系统

这些子系统不是独立的；它们集成在一起，特性互相使用。然而分开描述他们依然是有用的，构建自底而上的协议栈。 符号：以下数据结构和方法都是Go语言语法。

3.1 身份

节点被NodeId标识，是通过S/Kademlia's的静态加密难题[1]创建的公钥的加密散列。节点存储他们的公私钥(密码加密)。用户可以在每次启动时都实例一个新的节点标识，不过会丢失之前拥有的网络利益。节点激励措施保持不变。

```
type NodeId Multihash
type Multihash []byte
//self-describing cryptographic hash digest
type PublicKey []byte
type PrivateKey []byte
//self-describing keys

type Node struct {
    NodeId NodeID
    PubKey PublickKey
    PriKey PrivateKey
}
```

基于S/Kademlia创建IPFS身份标识:

```
difficulty = <integer parameter>
n = Node{}
do{
    n.PubKey, n.PrivaKey = PKI.genKeyPair()
    n.NodeId = hash(n.PubKey)
    p = count_preceding_zero_bits(hash(n.NodeId)
} while (p < difficulty)
```

第一次建立连接时，对等节点之间交换公钥，校验: `hash(other.PublicKey) equals other.NodeId`。如果不相等，连接中断。

加密函数须知

相对于将系统锁定在一些特殊的方法集，IPFS更倾向自我描述值。哈希摘要值以multihash形式存储，其头部包括使用的哈希函数说明和以字节为单位的摘要长度。例如：

```
<function code><digest length><digest bytes>
```

这允许系统来(a)选择最佳函数(安全性和速度性能) (b)随着方法选项的更新而进化，自我描述值允许使用不同参数选项兼容。

3.2 网络

IPFS节点通过潜在的广义网络和IPFS网络中的上百个节点定期通信。IPFS网络栈特性:

- 传输层: IPFS可以使用任何传输层协议，最好符合WebRTC DataChannels?或者uTP(LEDBAT [14])。
- 可靠性: IPFS所依赖的底层网络不能保证的可靠性的话，便使用uTP (LEDBAT [14])或者SCTP [15]来保证可靠性。
- 连通性: IPFS同样使用ICE NAT遍历技术[13]。
- 完整性: 提供使用哈希校验和来检验消息的完整性
- 确定性: 提供使用发送者公钥的HMAC来核查消息的真实性

3.2.1 对等网络地址须知

IPFS可以使用任意网络连接。他不依赖IP并且不假设已经获取IP。这使得IPFS可以在覆盖网络中使用。IPFS以multiaddr字节字符串形式保存地址，以供给底层网络使用。multiaddr提供了一种地址和协议的描述方式，并且提供了封装格式。例如：

```
# an SCTP/IPv4 connection
/ip4/10.20.30.40/sctp/1234/
# an SCTP/IPv4 connection proxied over TCP/IPv4
/ip4/5.6.7.8/tcp/5678/ip4/1.2.3.4/sctp/1234/
```

3.3 路由

IPFS节点需要一个路由系统来(a)寻找别的节点的网络地址，(b)寻找节点的服务对象。IPFS通过使用一个基于S/Kademlia和Coral的DSHT来解决上述问题，具体的特性详见2.1。对象的大小和IPFS使用的模式方面，类似于Coral[5]和Mainline[16]，因此IPFS的DHT根据对象存储时的大小对其进行区分。小的值(小于等于1KB)直接存储在DHT中。对于大的值，DHT中存储的是具体存储不同区块的对等节点的NodeIds的引用。 DSHT的接口如下：

```
type IPFSRouting interface {
    FindPeer(node NodeId)
    // gets a particular peer's network address
    SetValue(key []bytes, value []bytes)
    // stores a small metadata value in DHT
    GetValue(key []bytes)
    // retrieves small metadata value from DHT
    ProvideValue(key Multihash)
    // announces this node can serve a large value
    FindValuePeers(key Multihash, min int)
    // gets a number of peers serving a large value
```

注意: 不用的路由系统中需要不同的路由实例(广泛网络中使用DHT, 局域网络中使用静态HT)。因此IPFS路由系统可以根据用户需求来替换。只要接口保持匹配，系统便能正常运行。

3.4 块交换 - BitSwap协议

受到BitTorrent 的启发，IPFS 中的BitSwap协议通过对等节点间交换数据块来分发数据的。像BitTorrent一样， 对等节点在换取想要的块数据是以交换自己所拥有的块数据为前提的。和BitTorrent协议不同的是， BitSwap不局限于一个torrent文件中的数据块。BitSwap 协议中存在一个永久的市场。 这个市场包括各个节点可以获取的所有块数据，而不管在意这些块是哪些文件中的一部分。这些快数据可能来自文件系统中完全不相关的文件。 这个市场是由所有的节点组成的。 虽然易货系统的概念意味着可以创建虚拟货币，但这将需要一个全局分类账本来跟踪货币的所有权和转移。这可以实施为BitSwap策略，并将在未来的论文中探讨。 在基本情况下，BitSwap节点必须以块形式彼此提供直接的值。只有当跨节点的块的分布是互补的时候，各取所需的时候，这才会工作的很好。 通常情况并非如此，在某些情况下，节点必须为自己的块而工作。 在节点没有其对等节点所需的（或根本没有的）情况下，它会更低的优先级去寻找对等节点想要的块。这会激励节点去缓存和传播稀有片段， 即使节点对这些片段不感兴趣。

3.4.1 BitSwap信用

这个协议必须带有激励机制， 去激励节点去seed 其他节点所需要的块，而它们本身是不需要这些块的。 因此，BitSwap的节点很积极去给对端节点发送块，期待获得报酬。但必须防止水蛭攻击（空负载节点从不共享块），一个简单的类似信用的系统解决了这些问题：

1. 对等节点间会追踪他们的平衡（通过字节认证的方式）。
2. 随着债务增加而概率降低，对等者概率的向债务人发送块。

如果节点决定不发送到对等节点，节点随后忽略对等体的ignore_cooldown超时。 这样可以防止发送者尝试多次发送（洪水攻击） （BitSwap默认是10秒）。

3.4.2 BitSwap策略

BitSwap 对等节点采用很多不同的策略，这些策略对整个数据块的交换执行力产生了不同的巨大影响。在BitTorrent 中， 标准策略是特定的（tit-for-tat），从BitTyrant [8]（尽可能分享）到BitThief [8]（利用一个漏洞，从不共享），到PropShare [8]（按比例分享），其他不同的策略也已经被实施。BitSwap 对等节点可以类似地实现一系列的策略（良好和恶意）。对于功能的选择，应该明确：

1. 为整个交易和节点最大化交易成绩。
2. 防止空负载节点利用和损害交易。
3. 高效抵制未知策略
4. 对可信任的对等节点更宽容。

探索这些策略的空白是未来的事情。在实践中使用的一个选择性功能是sigmoid，根据负债比例进行缩放： 让负债比例在一个节点和它对等节点之间： `r = bytes_sent / bytes_recv + 1` 根据r，发送到负债节点的概率为： `P(send | r ) = 1 − ( 1/ ( 1 + exp(6 − 3r) ) )` 从图片1中看到，当节点负债比例超过节点已建立信贷的两倍，发送到负债节点的概率就会急速下降。 负债比是信任的衡量标准：对于之前成功的互换过很多数据的节点会宽容债务，而对不信任不了解的节点会严格很多。这个(a)给与那些创造很多节点的攻击者（sybill 攻击）一个障碍。(b)保护了之前成功交易节点之间的关系，即使这个节点暂时无法提供数据。(c)最终阻塞那些关系已经恶化的节点之间的通信，直到他们被再次证明。

3.4.3 BitSwap账本

BitSwap节点保存了一个记录与所有其他节点之间交易的账本。这个可以让节点追踪历史记录以及避免被篡改。当激活了一个链接，BitSwap节点就会互换它们账本信息。如果这些账本信息并不完全相同，分类账本将会重新初始化， 那些应计信贷和债务会丢失。 恶意节点会有意去失去“这些”账本， 从而期望清除自己的债务。节点是不太可能在失去了应计信托的情况下还能累积足够的债务去授权认证。伙伴节点可以自由的将其视为不当行为， 拒绝交易。

```
type Ledger struct {
    owner NodeId
    partner NodeId
    bytes_sent int
    bytes_recv int
    timestamp Timestamp
}
```

节点可以自由的保留分布式账本历史，这不需要正确的操作，因为只有当前的分类账本条目是有用的。节点也可以根据需要自由收集分布式帐本，从不太有用的分布式帐开始：老（其他对等节点可能不存在）和小。

3.4.4 BitSwap设计详述

BitSwap 节点有以下简单的协议。

```
// Additional state kept
type BitSwap struct {
    ledgers map[NodeId]Ledger
    // Ledgers known to this node, inc inactive
    active map[NodeId]Peer
    // currently open connections to other nodes
    need_list []Multihash
    // checksums of blocks this node needs
    have_list []Multihash
    // checksums of blocks this node has
}
type Peer struct {
    nodeid NodeId
    ledger Ledger
    // Ledger between the node and this peer
    last_seen Timestamp
    // timestamp of last received message
    want_list []Multihash
    // checksums of all blocks wanted by peer
    // includes blocks wanted by peer's peers
```

```
}
// Protocol interface:
interface Peer {
    open (nodeid :NodeId, ledger :Ledger);
    send_want_list (want_list :WantList);
    send_block (block :Block) -> (complete :Bool);
    close (final :Bool);
}
```

对等连接的生命周期草图：

1. Open: 对等节点间发送ledgers 直到他们同意。
2. Sending: 对等节点间交换want_lists 和blocks。
3. Close: 对等节点断开链接。
4. Ignored: （特殊）对等被忽略（等待时间的超时）如果节点采用防止发送策略。

Peer.open(NodeId, Ledger).

当发生链接的时候，节点会初始化链接的账本，要么保存一个份链接过去的账本，要么创建一个新的被清零的账本。然后，发送一个携带账本的open信息给对等节点。 接收到一个open信息之后，对等节点可以选择是否接受此链接。如果，根据接收者的账本，发送者是一个不可信的代理（传输低于零或者有很大的未偿还的债务），接收者可能会选择忽略这个请求。忽略请求是ignore_cooldown超时来概率性实现的，为了让错误能够有时间改正和攻击者被挫败。 如果链接成功，接收者用本地账本来初始化一个Peer对象以及设置last_seen时间戳。然后，它会将接受到的账本与自己的账本进行比较。如果两个账本完全一样，那么这个链接就被Open，如果账本并不完全一致，那么此节点会创建一个新的被清零的账本并且会发送此账本。

Peer.send_want_list(WantList)

当链接已经Open的时候，节点会广发它们的want_list给所有已经链接的对等节点。这个是在(a)open链接后(b)随机间歇超时后(c)want_list改变后(d)接收到一个新的块之后完成的。 当接收到一个want_list之后，节点会存储它。然后，会检查自己是否拥有任何它想要的块。如果有，会根据上面提到的BitSwap策略来将want_list所需要的块发送出去。

Peer.send_block(Block)

发送一个块是直接了当的。节点只是传输数据块。当接收到了所有数据的时候，接收者会计算多重hash校验和来验证它是否是自己所需数据，然后发送确认信息。 在完成一个正确的块传输之后，接受者会将此块从need_list一到have_list,最后接收者和发送者都会更新它们的账本来反映出传输的额外数据字节数。 如果一个传输验证失败了，发送者要么会出故障要么会攻击接收者，接收者可以选择拒绝后面的交易。注意，BitSwap是期望能够在一个可靠的传输通道上进行操作的，所以传输错误（可能会引起一个对诚实发送者错误的惩罚）是期望在数据发送给BitSwap之前能够被捕捉到。

Peer.close(Bool)

传给close最后的一个参数，代表close链接是否是发送者的意愿。如果参数值为false,接收者可能会立即重新open链接，这避免链过早的close链接。 一个对等节点close链接发生在下面两种情况下： silence_wait超时已经过期，并且没有接收到来自于对等节点的任何信息（BitSwap默认使用30秒），节点会发送Peer.close(false)。 在节点退出和BitSwap关闭的时候，节点会发送Peer.close(true). 接收到close消息之后，接收者和发送者会断开链接，清除所有被存储的状态。账本可能会被保存下来为了以后的便利，当然，只有在被认为账本以后会有用时才会被保存下来。

注意: 非open信息在一个不活跃的连接上应该是被忽略的。在发送send_block信息时，接收者应该检查这个块，看它是否是自己所需的，并且是否是正确的，如果是，就使用此块。总之，所有不规则的信息都会让接收者触发一个close(false)信息并且强制性的重初始化此链接。

3.5 Merkle DAG对象

DHT和BitSwap允许IPFS形成一个大规模的对等网络系统来快速并且健壮的存储和发布块文件。在这些之上, IPFS构建了一个Merkle DAG: 一个有向无环图，其中对象之间的链接是嵌入在源中的目标的加密散列。这是Git数据结构的泛化应用。MerKle DAGs为IFPS提供了许多有用的特性，包括：

1. 内容寻址: 所有的内容都通过它的multihash校验和唯一标识，包括links。
2. 防止篡改: 所有的内容通过它的校验和验证。如果数据被篡改或者损坏，IPFS会探测出。
3. 重复删除: 所有的对象内容完全相同便会只存储一份。对于索引对象尤其有用。例如git的tree和commit对象，以及数据的共同部分。

IFPS对象的格式如下：

```
type IPFSLink struct {
    Name string
    // name or alias of this link
    Hash Multihash
    // cryptographic hash of target
    Size int
    // total size of target
}
type IPFSObject struct {
    links []IPFSLink
    // array of links
    data []byte
    // opaque content data
}
```

IPFS的Merkle DAG是存储数据的一种非常灵活的方式。唯一的要求是对象的引用是(a)内容可寻址, (b)以之前描述的格式编码。IPFS给予应用对于data字段的完全控制权。应用可以定制任何数据格式，即使IPFS不能够理解。独立的内部对象link表允许IPFS:

- 列出一个对象列表中所有的对象引用，比如:

  ```
  > ipfs ls /XLZ1625Jjn7SubMDgEyeaynFuR84ginqvzb
  XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x 189458 less
  XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5 19441 script
  XLF4hwVHsVuZ78FZK6fozf8Jj9WEURMbCX4 5286 template
  <object multihash> <object size> <link name>
  ```

- 解决字符串路径查询，比如`foo/bar/baz`。给定一个对象,IPFS将第一个路径部分映射成对象link表里面的一个hash，获取第二部分对象。然后重复至下一个路径部分。因此任何数据格式的字符串路径都可以通过Merkle DAG定位位置。
- 解析递归引用的所有对象:

```
> ipfs refs --recursive \
/XLZ1625Jjn7SubMDgEyeaynFuR84ginqvzb
XLLxhdgJcXzLbtsLRL1twCHA2NrURp4H38s
XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x
XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5
XLWVQDqxo9Km9zLyquoC9gAP8CL1gWnHZ7z
...
```

一个原始data字段和通用link结构是构建IPFS顶层中的任意数据结构的必要组件。 通过观察如下数据结构: (a) 键值对存储, (b)传统关系型数据, (c)关联数据三层存储, (d) 关联文档发布系统, (e)通信平台, (f)加密货币区块, 可以容易的得知Git的对象模型是如何使用DAG。这些都可以通过IPFS的Merkle DAG建模，使得一些更复杂的系统使用IPFS系统作为传输层协议。

3.5.1路径

通过字符串路径API可以遍历IPFS对象。路径的工作方式与传统的UNIX文件系统和Web一致。Merkle DAG links使得访问很容易。IPFS完整路径格式如下:

```
# format
/ipfs/<hash-of-object>/<name-path-to-object>
# example
/ipfs/XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x/foo.txt
```

/ipfs前缀允许挂载到现在系统中的一个无冲的标准挂载点上(挂载点的名称是可以配置的)。路径第二个部分是对象的hash。通常情况下，没有一个全局跟路径。一个根对象需要来处理分布式系统中的百万个对象的一致性，这是一个不可能的任务。因此，我们模拟通过内容地址来模拟根地址。所有的对象总是可以通过他们的hash访问到。意味着给定地址`<foo>/bar/baz`的三个对象, 最后一个对象可以通过以下方式获取:

```
/ipfs/<hash-of-foo>/bar/baz
/ipfs/<hash-of-bar>/baz
/ipfs/<hash-of-baz>
```

3.5.2本地对象

IPFS客户端需要local storage, 一个外部系统用来存储并且检索本地原生数据，以便管理IPFS对象。存数的类型取决于节点的使用方式。大多数情况下，仅仅是硬盘空间的分配(或者通过文件系统管理，或者通过类似leveldb的键值存储，或者直接通过IPFS客户端)。其他情况下，比如非持久的缓存场景中，存储需要RAM的分配。 IPFS中所有的块都在节点的local storage。当用户获取对象时，对象被查到到然后下载最后存储在本地，至少是临时存储。这为一些可配置时间量提供了快速查询。

3.5.3对象固定

节点可以通过固定对象的方式来确保特定对象的生存。这样确保了此对象存储在节点的local storage中。固定操作可以使递归的，继而将所有关联的后代对象固定。所有的对象都在本地存储。在持久化文件中很有用处，包括引用。同样使得IPFS成为一个连接是永久的Web，并且对象可以确保其他被指定对象的生存。

3.5.4对象发布

IPFS是全球分发的。它被设计为允许用户成千上万的文件共同存在。通过内容哈希寻址，DHT可以在一种公平安全, 完全分布式的方式发布对象。任何人都可以通过简单的向DHT中添加对象的键来发布对象, 并且以对象是对等的方式添加, 别人通过这个对象的路径来访问。对象本质上是不可改变的, 就像Git。新的版本的hash是不同的, 意味着是新的对象。跟踪版本是额外版本对象的一个工作。

3.5.5对象级别加密

IPFS拥有处理对象级别加密的操作，一个加密或者签名的对象是被包裹在一个特殊的frame中，这个frame允许加密或者验证原生数据

```
type EncryptedObject struct {
    Object []bytes
    // raw object data encrypted
    Tag []bytes
    // optional tag for encryption groups
}
type SignedObject struct {
    Object []bytes
    // raw object data signed
    Signature []bytes
    // hmac signature
    PublicKey []multihash
    // multihash identifying key
}
```

加密操作改变了对象的hash，被定义为一个新的对象。IPFS自动验证签名并且通过用户声明钥匙链解密数据。加密数据的links同样也被保护, 没有解密钥匙便不能遍历。父对象和子对象使用不同的加密密钥是可行。这种方式保证分享对象links的安全。

3.6 文件

IPFS同样定义了一系列的对象来为Merkle DAG顶层的版本文件系统建立模型。对象模型类似于Git:

1. block: 可变的数据块
2. list: 一个blocks或者别的list的集合
3. tree: 一个blocks或者lists或者别的tree的集合
4. commit: 版本历史中一个tree的快照

我希望完全使用Git对象格式，但就需要分开来引入分布式系统中有用的特性。他们是, (a) 快速大小查询(总子节已经被添加到对象中), (b) 大文件去重(添加一个list对象), (c)将嵌入commits到trees中。总之，IPFS文件对象很接近Git，两者有可能进行交流。而且，一些Git对象可以在没有丢失信息(比如UNIX文件权限等)的情况下转换被引入。注意: 文件对象使用JSON格式。虽然ipfs包括了JSON的导入导出，实际上这些结构是使用protobufs的二进制编码。

3.6.1文件对象: blob

blob对象包含了一个可寻址的数据单元, 并且标识一个文件。IPFS Blocks类似Git的blobs或者文件系统的数据块。他们存储着用户的数据。注意, IPFS文件可以使用lists或者blobs来表示。blobs没有links。

```
{
"data": "some data here",
// blobs have no links
}
```

3.6.2文件对象: list

list对象是一个通过一些IPFS blobs聚合一起形成的大的或者去重的文件的表示。lists包含一个关于blob或者list的顺序序列。IPFS list的作用类似文件系统中的间接blocks。由于lists可以包含其他的lists, 拓扑包含带链接的lists和平衡数是有可能的。有向图中相同的节点在不同的位置使得文件内部去重。由于强制哈希寻址，环是不可能的。

```
{
"data": ["blob", "list", "blob"],
    // lists have an array of object types as data
"links": [
    { "hash": "XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x","size": 189458 },
    { "hash": "XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5", "size": 19441 },
    { "hash": "XLWVQDqxo9Km9zLyquoC9gAP8CL1gWnHZ7z", "size": 5286 }
    // lists have no names in links
    ]
}
```

3.6.3文件对象: tree

IPFS中的tree对象类似Git中的。表示一个目录, 一个名称和哈希的映射。这些哈希表示着blobs, lists, 别的tree或者commits。注意传统的路径名称已经被Merkle DAG实现。

```
{
"data": ["blob", "list", "blob"],
    // trees have an array of object types as data
"links": [
  { "hash": "XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x",
    "name": "less", "size": 189458 },
  { "hash": "XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5",
    "name": "script", "size": 19441 },
  { "hash": "XLWVQDqxo9Km9zLyquoC9gAP8CL1gWnHZ7z",
    "name": "template", "size": 5286 }
    // trees do have names
  ]
}
```

3.6.4文件对象: commit

commit对象表示着一个对象在版本历史中的一个快照。和Git中的类似，不过可以引用任何类型。可以链接作者的对象。

```
{
"data": {
    "type": "tree",
    "date": "2014-09-20 12:44:06Z",
    "message": "This is a commit message."
```

```
},
"links": [
    { "hash": "XLa1qMBKiSEEDhojb9FFZ4tEvLf7FEQdhdU",
      "name": "parent", "size": 25309 },
    { "hash": "XLGw74KAy9junbh28x7ccWov9inu1Vo7pnX",
      "name": "object", "size": 5198 },
    { "hash": "XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm",
      "name": "author", "size": 109 }
  ]
}
```

3.6.5版本控制

3.6.6文件系统路径

通过Merkle DAG小节, IPFS对象通过字符串路径API遍历。IPFS文件对象被设计成更容易挂载到UNIX文件系统上。为了表示trees为目录，他们限制trees没有数据。并且commit可以被表示成目录或者完全隐藏在文件系统。

3.6.7文件分割成Lists和Blob

对于大文件的版本和分布式化的一个主要挑战就是找到合理的方式来切分它成blocks。与其假设不同类型的文件有正确的切分法，不如说IPFS提供了如下选择:

- (a) 使用LBFS[?]中的Rabin Fingerprints[?]方法来选择合适的block边界。
- (b) 使用rsync[?] rolling-checksum算法, 来检测版本之间的块变动。
- (c) 允许用户为制定的文件来设定块分割算法

3.6.8路径查询性能

基于路径访问需要遍历对象图。检索每一个对象需要查询DHT中对应的密钥, 连接到对应的对等节点, 检索出它的blocks。这是很大的开销，尤其是在查询路径中存在许多子路径构成。通过以下方式来缓和:

- tree caching: 由于所有的对象都是哈希可寻址的, 他们可以被无限的缓存。此外, trees的大小更小一些，IPFS更倾向在blobs之上缓存它们。
- flattened trees: 对于给定的tree, 一个特殊的flattened tree可以被构建成一个链表, 所有对象从这个tree中访问得到。flattened tree中的名称可以分别通过原始tree以分隔符连接获得。

例如, ttt111的flattened tree如下:

```
{
"data":
    ["tree", "blob", "tree", "list", "blob" "blob"],
"links": [
    { "hash": "<ttt222-hash>", "size": 1234
    "name": "ttt222-name" },
    { "hash": "<bbb111-hash>", "size": 123,
    "name": "ttt222-name/bbb111-name" },
    { "hash": "<ttt333-hash>", "size": 3456,
    "name": "ttt333-name" },
    { "hash": "<lll111-hash>", "size": 587,
    "name": "ttt333-name/lll111-name"},
    { "hash": "<bbb222-hash>", "size": 22,
    "name": "ttt333-name/lll111-name/bbb222-name" },
```

```
    { "hash": "<bbb222-hash>", "size": 22
    "name": "bbb222-name" }
] }
```

3.7 IPNS: 命名空间和可变状态

目前为止, IPFS协议栈通过构建一个内容可寻址的DAG对象集形成了一个点对点块交换系统。它用来发布和检索不可变对象。它可以追踪这些对象的版本变化。然而, 一个重要的部分缺失了: 可变的命名。没有此部分, 所有以IPFS links通信的新内容都会有所偏差。我们需要在一个相同的路径下检索可变的状态。 如果可变的数据到最后是必须的, 我们必须明确为什么我们努力构建了一套不可变的Merkle DAG。考虑到IPFS的特性: 对象可以(a)通过哈希检索 (b)完整性检查 (c)关联到其他对象 (d)无限的缓存。某种意义上: 对象是永久的
这个也是高性能分布式系统的重要特性, 数据通过网络links来移动是昂贵的。对象内容可寻址构建了一个具有以下特点的web:

- (a) 显著的带宽优化
- (b) 非信任的内容服务
- (c) 永久的链接
- (d) 任何一个对象和它的引用的完整永久性备份

基于不可变的内容可寻址的对象和命名的Merkle DAG和指向Merkle DAG的可变的指针, 实例化了一个呈现在许多成功的分布式系统中的二分法。他们包括Git版本控制系统, 拥有不可变的对象和可变的引用。Plan9[?]拥有可变的Fossil[?]和不可变的Venti[?]文件系统。LBFS[?]同样拥有可变的indices和不可变的chunks。

3.7.1自我认证命名

通过SFS[12, 11]的命名方案，我们得到一个方式，在整个加密制定的命名空间中构建自我认证的命名，并且是可变的。IPFS的方案如下:

1. 回想IPFS中: NodeId = hash(node.PubKey)
2. 我们给每一个用户分配了一个可变的命名空间: `/ipns/<NodeId>`
3. 一个用户可以通过附上它的私钥签名发布一个对象到这个路径上, 比如:`/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/`
4. 当别的用户检索这个对象时, 他们可以检查这个签名是否匹配公钥和NodeId。这将验证用户发布对象的真实性，获取可变状态的查询。

注意细节:

- ipns(InterPlanetary Name Space)独立的前缀是为了建立一个简单的识别区分来区分可变和不可变的路径, 为了程序也为了人们阅读。
- 由于这不是一个内容可寻址的对象，发布它需要依赖IPFS可变状态分发系统, 路由系统。过程是(1)按照常规不可变IPFS对象发布对象, (2) 在路由系统中发布它的哈希作为一个meta元数据值:

  ```
  routing.setValue(NodeId, <ns-object-hash>)
  ```

- 被发布的对象的任何links被视为命名空间的子名称:

  ```
  /ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/
  /ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/docs
  /ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/docs/ipfs
  ```

- 被推荐发布一个commit对象, 或者版本历史系统的别的对象, 以至于客户端可以找到旧的名称。这样便留给用户一个选择。

注意当用户发布这个对象时, 它不能用相同的方式发布。

3.7.2人类可读命名

IPNS确实是一种方式来分发和再分发命名, 不过它还是暴露的长长的hash值作为命名, 对于用户不是很友好,很难记忆。IPFS通过一下技术增强用户友好的IPNS。

Peer Links

收到SFS的启发, 用户可以直接关联别人的对象到自己的对象中(命名空间,home)。这样同样增加了网络信任。

```
# Alice links to bob Bob
ipfs link /<alice-pk-hash>/friends/bob /<bob-pk-hash>
# Eve links to Alice
ipfs link /<eve-pk-hash/friends/alice /<alice-pk-hash>
# Eve also has access to Bob
/<eve-pk-hash/friends/alice/friends/bob
# access Verisign certified domains
/<verisign-pk-hash>/foo.com
```

DNS TXT IPNS 记录

如果/ipns/是一个有效的域名名称, IPFS可以在自己的DNS TXT记录中查找ipns键。IPFS翻译那个值或者作为一个对象hash或者另一个IPNS路径:

```
# this DNS TXT record
ipfs.benet.ai. TXT "ipfs=XLF2ipQ4jD3U ..."
# behaves as symlink
ln -s /ipns/XLF2ipQ4jD3U /ipns/fs.benet.ai
```

Proquint Pronounceable Identifiers

有方案将二进制转换成可发音的单词。IPNS支持Proquint[?]。如下:

```
# this proquint phrase
/ipns/dahih-dolij-sozuk-vosah-luvar-fuluh
# will resolve to corresponding
/ipns/KhAwNprxYVxKqpDZ
```

Name Shortening Services

涌现出提供缩短名称的服务, 向用户提供他们的命名空间。就像我们现在看到的DNS和Web的URLs:

```
# User can get a link from
/ipns/shorten.er/foobar
# To her own namespace
/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm
```

3.8 IPFS使用场景

IPFS设计为可以使用多种不同的方法来使用的，下面就是一些我将会继续追求的使用方式：

1. 作为一个挂载的全局文件系统，挂载在/ipfs和/ipns下
2. 作为一个挂载的个人同步文件夹，自动的进行版本管理，发布，以及备份任何的写入
3. 作为一个加密的文件或者数据共享系统
4. 作为所有软件的版本包管理者
5. 作为虚拟机器的根文件系统
6. 作为VM的启动文件系统 (在管理程序下)
7. 作为一个数据库：应用可以直接将数据写入Merkle DAG数据模型中，获取所有的版本，缓冲，以及IPFS提供的分配
8. 作为一个linked（和加密的）通信平台
9. 作为一个为大文件的完整性检查CDN（不使用SSL的情况下）
10. 作为一个加密的CDN
11. 在网页上，作为一个web CDN
12. 作为一个links永远存在新的永恒的Web

IPFS实现的目标：

- (a)一个IPFS库可以导出到你自己应用中使用
- (b)命令行工具可以直接操作对象
- (c)使用FUSE[?]或者内核的模型挂载文件系统

4. 未来

5. 感谢