**IPFS - Content Addressed, Versioned, P2P File System**

**IPFS-内容寻址，版本控制的P2P文件系统**

# ABSTRACT

The InterPlanetary File System (IPFS) is a peer-to-peer distributedle system that seeks to connect all computing devices with the same system of fles. In some ways, IPFS is similar to the Web, but IPFS could be seen as a single BitTorrent swarm, exchanging objects within one Git repository. In other words, IPFS provides a high throughput content-addressed block storage model, with content-addressed hyper links. This forms a generalized Merkle

DAG, a data structure upon which one can build versioned file systems, blockchains, and even a Permanent Web. IPFS

combines a distributed hashtable, an incentivized block exchange, and a self-certifying namespace. IPFS has no single

point of failure, and nodes do not need to trust each other.

**摘要：**星际文件系统（IPFS）是一个对等的分布式系统，旨在用相同的文件系统连接所有的计算机设备。在某些方面，IPFS和Web类似，但是IPFS能够被看做一个单一的BitTorrent群，在一个Git 仓库中交换对象。换句话说，IPFS用内容寻址超链接提供一个高吞吐量的，内容寻址的块存储模型。它形成一个整体的Merkle DAG，这是一个数据结构，据此可建立版本控制的文件系统，区块链，甚至是永久Web。IPFS结合分布式哈希表，激励块交换，和自我证明的命名空间。IPFS没有单点故障，节点不需要互相信任。

# 1. INTRODUCTION

There have been many attempts at constructing a global distributed file system. Some systems have seen signifi

cant success, and others failed completely. Among the academic attempts, AFS [6] has succeeded widely and is still

in use today. Others [7, ?] have not attained the same success. Outside of academia, the most successful systems

have been peer-to-peer file-sharing applications primarily geared toward large media (audio and video). Most no-

tably, Napster, KaZaA, and BitTorrent [2] deployed large file distribution systems supporting over 100 million simul-

taneous users. Even today, BitTorrent maintains a massive deployment where tens of millions of nodes churn daily [16].These applications saw greater numbers of users and files distributed than their academic file system counterparts. However, the applications were not designed as infrastructure to be built upon. While there have been successful repurposings, no general file-system has emerged that others global,low-latency, and decentralized distribution.

在构建一个全球分布式文件系统方面，已经有许多尝试。有些系统已取得显著成功，其他系统完全失败。在学术界的尝试中，AFS [ 6 ]已经取得了广泛的成功，至今仍在继续使用。其的系统没有取得同样的成功。在学术界之外，最成功的系统主要是面向大型媒体（音频和视频）的对等文件共享应用程序。最值得注意的是，Napster，Kazaa和BitTorrent部署，[ 2 ]大文件分发系统支持超过1亿个并发用户。即使在今天，BitTorrent仍然保持着大规模的部署，每天有数以千万计的节点在流失。这些应用程序比他们的学术文件系统对应的用户和文件数量更多。然而，这些应用程序并没有被设计成基础设施。虽然有过成功的再利用，但没有出现能提供全球的，低延迟的，去中心化的通用的文件系统。

Perhaps this is because a “good enough" system for most use cases already exists: HTTP. By far, HTTP is the most successful “distributed system of files" ever deployed. Coupled with the browser, HTTP has had enormous technical and social impact. It has become the de facto way to transmit files across the internet. Yet, it fails to take advantage of dozens of brilliant file distribution techniques invented in the last fifteen years. From one prespective, evolving Web infrastructure is near-impossible, given the number of backwards compatibility constraints and the number of strong parties invested in the current model. But from another perspective, new protocols have emerged and gained wide use since the emergence of HTTP. What is lacking is upgrading design: enhancing the current HTTP web, and introducing new functionality without degrading user experience.

这可能是因为对大多数用例来说，一个“足够好”系统已经存在：HTTP。到目前为止，HTTP是部署最成功的“分布式文件系统”。与浏览器结合，HTTP产生了巨大的技术和社会影响。它已经成为在因特网上传输文件的实际的方法。然而，它未能充分利用过去十五年发明的几十种出色的文件分发技术。从一个角度来看，考虑到向后兼容性约束的数量和在己经投入到当前模型中的政党数量, 不断发展的网络基础设施，几乎是不可能的。但是从另一个角度来看，自从HTTP出现以来，新的协议就应运而生并得到了广泛的应用。缺乏的是升级设计：增强当前的HTTP Web，以及引入新功能而不降低用户体验。

Industry has gotten away with using HTTP this long because moving small files around is relatively cheap, even for

small organizations with lots of traffic. But we are entering a new era of data distribution with new challenges: (a)

hosting and distributing petabyte datasets, (b) computing on large data across organizations, (c) high-volume high-

denition on-demand or real-time media streams, (d) versioning and linking of massive datasets, (e) preventing ac-

cidental disappearance of important files, and more. Many of these can be boiled down to “lots of data, accessible ev-

erywhere." Pressed by critical features and bandwidth concerns, we have already given up HTTP for different data

distribution protocols. The next step is making them part of the Web itself.

长期以来，业界一直在使用HTTP，因为移动小文件是相对便宜的，即使对于拥有大量流量的小组织来说也是如此。但我们正在进入一个数据分发的新时代，面临的挑战有：(a)托管和分发PB级数据集,(b)跨组织的大数据计算,(c)大容量高清晰度点播或实时媒体流,(d)海量数据集的版本控制与链接,(e)防止重要文件的意外丢失，等等。这其中的许多问题可以归结为“大量数据，随处可访问”。由于关键特性和带宽问题的影响紧缺，对于不同的数据分发协议来说，我们已经放弃使用HTTP。下一步是使它们成为Web本身的一部分。

Orthogonal to efficient data distribution, version control systems have managed to develop important data collaboration workflows. Git, the distributed source code version control system, developed many useful ways to model and

implement distributed data operations. The Git toolchain offers others versatile versioning functionality that large file distribution systems severely lack. New solutions inspired by Git are emerging, such as Camlistore [?], a personal file storage system, and Dat [?] a data collaboration toolchain and dataset package manager. Git has already influenced distributed file system design [9], as its content addressed Merkle DAG data model enables powerful file distribution

strategies. What remains to be explored is how this data structure can influence the design of high-throughput oriented file systems, and how it might upgrade the Web itself.

正交高效的数据分发，版本控制系统已经开发出了重要数据协作工作流。Git, 分布式源代码版本控制系统，开发了许多有用的建模和实现分布式数据操作的方法。Git工具提供了其他大型分布式文件系统严重缺乏的丰富的版本控制功能。以Git为灵感的新的解决方案层出不穷，如camlistore，是一个个人文件存储系统；Dat，是一个数据协作工具链和数据包管理工具。Git已经影响了分布式文件系统的设计，其内容寻址Merkle DAG数据模型驱动了强大的文件分配策略。有待探讨的是这种数据结构如何影响面向高吞吐量的文件系统的设计，以及它如何可能升级Web本身。

This paper introduces IPFS, a novel peer-to-peer version controlled file system seeking to reconcile these issues. IPFS

synthesizes learnings from many past successful systems.Careful interface-focused integration yields a system greater

than the sum of its parts. The central IPFS principle is modeling all data as part of the same Merkle DAG

本文介始了IPFS，一个新的对等版本控制的文件系统，旨在调和这些问题。IPFS从过去的许多成功系统中综合学习。以接口为中心的精心集成产生一个大于其各部分之和的系统。IPFS中心的原则是把所有的数据作为相同的Merkle DAG的一部分建模。

# 2. BACKGROUND

This section reviews important properties of successful peer-to-peer systems, which IPFS combines.

本节介绍IPFS所结合的成功的P2P系统的重要特性。

## 2.1 Distributed Hash Tables

Distributed Hash Tables (DHTs) are widely used to coordinate and maintain metadata about peer-to-peer systems.

For example, the BitTorrent Mainline DHT tracks sets of peers part of a torrent swarm.

分布式哈希表（DHT）广泛用于协调和保持P2P系统的元数据。例如，BitTorrent Mainline DHT跟踪一个torrent群的一部分peers集合。

### 2.1.1 Kademlia DHT

Kademlia [10] is a popular DHT that provides:

1. Efficient lookup through massive networks: queries on average contact dlog2(n)e nodes. (e.g. 20 hops for a

network of 10; 000; 000 nodes).

2. Low coordination overhead: it optimizes the number of control messages it sends to other nodes.

3. Resistance to various attacks by preferring long-lived nodes.

4. Wide usage in peer-to-peer applications, including Gnutella and BitTorrent, forming networks of over 20

million nodes [16].

Kademlia是一个受欢迎的分布式哈希表，它提供了：

1. 通过大规模网络的有效查找：查询平均连接log2N节点.(例如：一个10000000节点的网络有20跳)
2. 低协调开销：它优化发送到其他节点的控制消息的数量。
3. 通过偏爱长寿命节点抵抗各种攻击。
4. 在对等网络的广泛应用，包括Gnutella和BitTorrent，形成超过2000万节点的网络。

### 2.1.2 Coral DSHT

While some peer-to-peer file systems store data blocks directly in DHTs, this “wastes storage and bandwidth, as data

must be stored at nodes where it is not needed" [5]. The Coral DSHT extends Kademlia in three particularly important ways:

1. Kademlia stores values in nodes whose ids are“nearest"(using XOR-distance) to the key. This does not take

into account application data locality, ignores “far"nodes that may already have the data, and force“\nearest" nodes to store it, whether they need it or not.This wastes significant storage and bandwith. Instead,Coral stores addresses to peers who can provide the data blocks.

2. Coral relaxes the DHT API from get\_value(key) to get\_any\_values(key) (the \sloppy" in DSHT). This

still works since Coral users only need a single (working) peer, not the complete list. In return, Coral can

distribute only subsets of the values to the \nearest"nodes, avoiding hot-spots (overloading all the nearest

nodes when a key becomes popular).

3. Additionally, Coral organizes a hierarchy of separate DSHTs called clusters depending on region and size.

This enables nodes to query peers in their region

一些P2P文件系统直接把数据块存储在DHTs中，这种作法“浪费存储和带宽，因为数据

必须被存储在不需要的节点上”[5]。Kademlia在三个特别重要的方面扩展了DSHT：

1. Kademlia在id离key“最近”的节点上存储值(异或距离)。这并不考虑应用程序数据局部性，忽略己经有数据的“远”的节点，只关注最近的节点存储而不管节点是否需要。这浪费了有用的存储和带宽。相反，Coral存储地址到可以提供数据块的peers中。
2. Coral把DHT API 从 get\_value(key) 放宽到 get\_any\_values(key) （DSHT中的”sloppy”）.这同样是有用的，因为Coral用户只需要一个单一(工作中)的peer,而不需要整个列表。反过来，Coral只能将值的子集分配到“最近的”节点,避免热点(当一个密钥变得流行时，重载所有最近的节点)。
3. 此外，Coral组织一个独立的层次结构，DSHT称为基于区域和大小的集群，这使得节点能够查询其区域中的peers。

### 2.1.3 S/Kademlia DHT

S/Kademlia [1] extends Kademlia to protect against malicious attacks in two particularly important ways:

1. S/Kademlia provides schemes to secure NodeId generation, and prevent Sybill attacks. It requires nodes to

create a PKI key pair, derive their identity from it,and sign their messages to each other. One scheme

includes a proof-of-work crypto puzzle to make generating Sybills expensive.

2. S/Kademlia nodes lookup values over disjoint paths,in order to ensure honest nodes can connect to each

other in the presence of a large fraction of adversaries in the network. S/Kademlia achieves a success rate of

0.85 even with an adversarial fraction as large as half of the nodes.

S / Kademlia [ 1 ]用两个尤为重要手段对Kademlia进行扩展以对抗恶意攻击：

1. S/Kademlia提供一个方案来保护NodeId的生成，预防Sybill攻击。它要求节点创建一个PKI密钥对，从中提取它们的身份，并将它们的消息相互签名，一个方案包括工作证明密码拼图使生成sybills昂贵。

2. S/Kademlia节点通过不相交的路径查找值，以确保在网络中的存在很大一部分对手的时候，诚实的节点也可以互相连接。当对手占所有节点一半的时候，S/Kademlia甚至能实现0.85的成功率。

## 2.2 Block Exchanges - BitTorrent

BitTorrent [3] is a widely successful peer-to-peer filesharing system, which succeeds in coordinating networks of untrusting peers (swarms) to cooperate in distributing pieces of files to each other. Key features from BitTorrent and its

ecosystem that inform IPFS design include:

1. BitTorrent's data exchange protocol uses a quasi tit-for-tat strategy that rewards nodes who contribute to

each other, and punishes nodes who only leech others' resources.

2. BitTorrent peers track the availability of file pieces,prioritizing sending rarest pieces first. This takes load

off seeds, making non-seed peers capable of trading with each other.

3. BitTorrent's standard tit-for-tat is vulnerable to some exploitative bandwidth sharing strategies. PropShare [8]

is a different peer bandwidth allocation strategy that better resists exploitative strategies, and improves the

performance of swarms.

2.2 块交换-BitTorrent

BitTorrent的[ 3 ]是一种被广泛利用的成功的P2P文件共享系统，它成功的在不信任节点网络中协调彼此的文件分发。被引入IPFS的BitTorrent及其生态系统的关键特征包括：

1. BitTorrent的数据交换协议使用了一种准针锋相对的策略，奖励互相贡献的节点，惩罚只挖掘他人资源的节点。
2. BitTorrent对等点跟踪文件片段的可用性，优先发送最稀有的片断。这就需要把种子卸下来，使没有种子的对等点能够相互交易。
3. BitTorrent的标准针锋相对是容易受到一些剥削性带宽共享策略的攻击。propshare [ 8 ]是一个不同的对等带宽分配策略，能更好地抵抗剥削性策略，提高群的性能。

## 2.3 Version Control Systems - Git

Version Control Systems provide facilities to model files changing over time and distribute different versions efciently.

The popular version control system Git provides a powerful Merkle DAG object model that captures changes to a file system tree in a distributed-friendly way.

1. Immutable objects represent Files (blob), Directories(tree), and Changes (commit).

2. Objects are content-addressed, by the cryptographic hash of their contents.

3. Links to other objects are embedded, forming a Merkle DAG. This provides many useful integrity and workflow properties.

4. Most versioning metadata (branches, tags, etc.) are simply pointer references, and thus inexpensive to cre-

ate and update.

5. Version changes only update references or add objects.

6. Distributing version changes to other users is simply transferring objects and updating remote references.

2.3 版本控制系统-Git

版本控制系统提供了模型文件随时间变化的工具，并有效地分发不同版本.流行的版本控制系统Git提供了一个强大的Merkle DAG对象模型，它以分布式友好方式捕获对文件系统树的更改。

1. 不可变对象代表文件（数据集）、目录（树）和更改（提交）
2. 对象通过内容的加密散列进行内容寻址
3. 其他对象嵌入链接，形成一个Merkle DAG。这提供了许多有用的完整性和工作流属性。
4. 新版本的元数据（标签，分支，等）是用指针引用到，因此可以廉价的创建和更新。
5. 版本更改只更新引用或添加对象
6. 将版本更改分发给其他用户仅仅是传递对象和更新远程引用。

## 2.4 Self-Certified Filesystems - SFS

SFS [12, 11] proposed compelling implementations of both(a) distributed trust chains, and (b) egalitarian shared global

namespaces. SFS introduced a technique for building Self Certified Filesystems: addressing remote file systems using

the following scheme

*/sfs/<Location>:<HostID>*

where Location is the server network address, and:

*HostID = hash(public\_key || Location)*

Thus the name of an SFS file system certifies its server.The user can verify the public key offered by the server,

negotiate a shared secret, and secure all traffic. All SFS instances share a global namespace where name allocation

is cryptographic, not gated by any centralized body.

2.4 自认证的文件系统-SFS

SFS提供了（a）发布式信任链和（b）平等共享全局命名空间 两者的强制实现。SFS介绍了构建自证明的文件系统的技术。使用以下方案寻址远程文件系统：

*/sfs/<Location>:<HostID>*

*Location*是服务器网络地址，而且：

*HostID = hash(public\_key || Location)*

因此一个SFS文件系统名称认证了它自己的服务器，用户可以验证服务器提供的公钥，协商共享机密，并确保所有的通信。所有SFS实例共享一个全局名称空间，其中名称分配是加密的，而不是由任何集中式机构控制的

# 3.IPFS DESIGN

**IPFS 设计**

IPFS is a distributed file system which synthesizes successful ideas from previous peer-to-peer sytems, including

DHTs, BitTorrent, Git, and SFS. The contribution of IPFS is simplifying, evolving, and connecting proven techniques

into a single cohesive system, greater than the sum of its parts. IPFS presents a new platform for writing and de-

ploying applications, and a new system for distributing and versioning large data. IPFS could even evolve the web itself.

IPS是一个分布式文件系统，它成功的综合了以前的p2p系统的思想，包括DHTs,BitTorrent,Git 和SFS.

IPFS的贡献是简化,进化并且连接己经经过验证的技术到一个单一的紧密结合的系统中，达到整体大于各部分之和的效果。它提供了一个新的平台来编写和部署应用程序，以及一个分配和版本化大数据的新系统。它甚至可以进化Web本身。

IPFS is peer-to-peer; no nodes are privileged. IPFS nodes store IPFS objects in local storage. Nodes connect to each

other and transfer objects. These objects represent files and other data structures. The IPFS Protocol is divided into a

stack of sub-protocols responsible for dierent functionality:

IPFS是对等的；没有节点享有特权。IPFS节点在本地存储IPFS对象。节点相互连接并传输对象。这些对象表示文件和其他数据结构。IPFS协议分为负责不同功能的一堆子协议。

1. Identities - manage node identity generation and verification. Described in Section 3.1.

2. Network - manages connections to other peers, uses various underlying network protocols. Configurable.

Described in Section 3.2.

3. Routing - maintains information to locate specific peers and objects. Responds to both local and re-

mote queries. Defaults to a DHT, but is swappable.Described in Section 3.3.

4. Exchange - a novel block exchange protocol (BitSwap)that governs efficient block distribution. Modelled as

a market, weakly incentivizes data replication. Trade Strategies swappable. Described in Section 3.4.

5. Objects - a Merkle DAG of content-addressed immutable objects with links. Used to represent arbitrary datastructures, e.g. file hierarchies and communication systems. Described in Section 3.5.

6. Files - versioned file system hierarchy inspired by Git.

Described in Section 3.6.

1. Naming - A self-certifying mutable name system. Described in Section 3.7.
2. 身份-管理节点身份生成和验证，在3.1节描述。
3. 网络-管理与其他节点的连接，使用各种底层网络协议。可配置。在3.2节描述。
4. 路由-维护信息以定位特定的对等点和对象.响应本地和远程查询。默认为DHT，但是可替换。在3.3节中描述。
5. 交换-一个新的块交换协议（BitSwap）,管理有效的块分配。在3.4节中描述。
6. 对象-一个有链接的内容寻址的不可改变的对象的Merkle DAG。用于表示任意数据结构。例如：文件层次结构和通信系统。在3.5节描述。
7. 文件-版本控制的文件系统，由Git启发分层。在3.6节中描述。
8. 命名-一个自认证的可变名称系统。第3.7节中描述。

These subsystems are not independent; they are integrated and leverage blended properties. However, it is useful to describe them separately, building the protocol stack from the bottom up.

Notation: data structures and functions below are specified in Go syntax.

这些子系统不是独立的,它们是综合的，并利用混合属性。然而，分开描述是有用的，从底层构建协议栈。

标记：下面的数据结构和函数在GO语法中指定。

## 3.1 Identities

Nodes are identified by a NodeId, the cryptographic hash3 of a public-key, created with S/Kademlia's static crypto puz-

zle [1]. Nodes store their public and private keys (encrypted with a passphrase). Users are free to instatiate a “new" node identity on every launch, though that loses accrued network benefits. Nodes are incentivized to remain the same.

节点由NodeId鉴定，NodeId是公钥的加密散列，由S/Kademia的静态密码之迷创建。节点存储它们的公钥和私钥（用密码加密）. 虽然失去了应有的网络效益，但每一次发布时用户可以自由地实例化一个“新的”节点身份。节点激励保持不变。

*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 PublicKey*

*PriKey PrivateKey*

*}*

*S/Kademlia based IPFS identity generation:*

*基于S/Kademlia的IPFS身份生成:*

*difficulty = <integer parameter>*

*n = Node{}*

*do {*

*n.PubKey, n.PrivKey = PKI.genKeyPair()*

*n.NodeId = hash(n.PubKey)*

*p = count\_preceding\_zero\_bits(hash(n.NodeId))*

*} while (p < difficulty)*

Upon first connecting, peers exchange public keys, and check: hash(other.PublicKey) equals other.NodeId. If

not, the connection is terminated.

在第一次连接时，节点交换公钥，并检查：hash（other.PublicKey）=other.nodeid。如果不相等，则连接终止。

**Note on Cryptographic Functions.**

加密函数的注意事项：

Rather than locking the system to a particular set of function choices, IPFS favors self-describing values. Hash digest values are stored in multihash format, which includes a short header specifying the hash function used, and the

digest length in bytes. Example:

*<function code><digest length><digest bytes>*

This allows the system to (a) choose the best function for the use case (e.g. stronger security vs faster performance),

and (b) evolve as function choices change. Self-describing values allow using different parameter choices compatibly.

IPFS更喜欢自描述的值，而不是锁定系统到一组特定的函数选择中。Hash摘要的值被存储在一个multihash格式中，它包括一个指明所使用的hash函数的短头部，以及摘要长度。例如：

*<function code><digest length><digest bytes>*

它允许系统(a)为用例选择最佳函数（例如：更强的安全性VS更快的性能），（b）随着功能选择的变化而演化.

自描述的值允许使用合适的不同的参数选择

## 3.2 Network

IPFS nodes communicate regualarly with hundreds of other nodes in the network, potentially across the wide internet.

The IPFS network stack features:

* Transport: IPFS can use any transport protocol,and is best suited for WebRTC DataChannels [?] (for browser connectivity) or uTP(LEDBAT [14]).
* Reliability: IPFS can provide reliability if underlying networks do not provide it, using UTP (LEDBAT [14])

or SCTP [15].

* Connectivity: IPFS also uses the ICE NAT traversal techniques [13].
* Integrity: optionally checks integrity of messages using a hash checksum.
* Authenticity: optionally checks authenticity of messages using HMAC with sender's public key.

IPFS 节点节点定期在网络上的其他节点通信，跨越广域网。

IPFS网络的特点：

* 传输：IPFS可以使用任何传输协议，它最适合WebRTC DataChannels
* 可靠性：如果底层网络不提供可靠性，IPFS能提供。如当使用UTP或SCTP时。
* 连通性：IPFS也使用ICE NAT 穿越技术
* 完整性：可选择使用hash校验和检查消息的完整性。
* 真实性：可以选择使用HMAC与发送方的公钥检查消息的真实性。

### 3.2.1 Note on Peer Addressing

3.2.1关于对等地址的注释

IPFS can use any network; it does not rely on or assume access to IP. This allows IPFS to be used in overlay networks.

IPFS stores addresses as multiaddr formatted byte strings for the underlying network to use. multiaddr provides a way

to express addresses and their protocols, including support for encapsulation. For example:

*# 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/*

IPFS可以使用任何网络；它不依赖或假定访问IP. 这允许IPFS用于覆盖网络。对于使用底层网络，IPFS存储地址作为格式化的多地址字符串。多地址提供了一种方法来表示地址和协议。包括对封装的支持。例如：

*#一个SCTP/IPv4连接*

*/ip4/10.20.30.40/sctp/1234/*

*#一个基于TCP/IP代理的SCTP/IPv4连接*

*/ip4/5.6.7.8/tcp/5678/ip4/1.2.3.4/sctp/1234/*

## 3.3 Routing

IPFS nodes require a routing system that can find (a)other peers' network addresses and (b) peers who can serve

particular objects. IPFS achieves this using a DSHT based on S/Kademlia and Coral, using the properties discussed in

2.1. The size of objects and use patterns of IPFS are similar to Coral [5] and Mainline [16], so the IPFS DHT makes a

distinction for values stored based on their size. Small values (equal to or less than 1KB) are stored directly on the DHT. For values larger, the DHT stores references, which are the NodeIds of peers who can serve the block.

The interface of this DSHT is the following:

IPFS节点需要一个路由系统，该路由系统可以找到(a)其他对等点的网络地址 (b)能为特定对象服务的对等点。

IPFS使用基于S/Kademlia 和Coral的DSHT实现上述功能，使用的是2.1节所讨论的属性。对象的大小和IPFS的用模式与Coral和Mainline类似,因此IPFS DHT根据存储的大小对存储的值进行区分。小值（等于或小于1KB）直接存储在DHT。对于较大的值，DHT存储引用，引用是能服务块的对等点的NodeId.

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

}

Note: different use cases will call for substantially different routing systems (e.g. DHT in wide network, static HT

in local network). Thus the IPFS routing system can be swapped for one that fits users' needs. As long as the interface above is met, the rest of the system will continue to function.

注释：不同的用例将调用实质上不同的路由系统。(例如：广域网中的DHT，局域网中的静态HT).因此，IPFS路由系统可以换成符合用户需求的路由系统。只要满足上面的接口，系统的其余部分将继续运行。

## 3.4 Block Exchange - BitSwap Protocol

In IPFS, data distribution happens by exchanging blocks with peers using a BitTorrent inspired protocol: BitSwap.

Like BitTorrent, BitSwap peers are looking to acquire a set of blocks (want\_list), and have another set of blocks to of-

fer in exchange (have\_list). Unlike BitTorrent, BitSwap is not limited to the blocks in one torrent. BitSwap operates as a persistent marketplace where node can acquire the blocks they need, regardless of what files those blocks are

part of. The blocks could come from completely unrelated files in the filesystem. Nodes come together to barter in the

marketplace.

3.4 块交换-BitSwap 协议

在IPFS中,对等点交换块通过使用由BitTorrent启发的协议:BitSwap来实现数据分配。和BitToreent类似的是，BitSwap对等点希望获得一组块（want\_list）,并且拥有另外一组块提供交换(have\_list)。与BitTorrent不同的是：BitSwap不把块限定在一个torrent. BitSwap作为一个持续的市场运作，无论这些块属于哪个文件，节点都可以获得他们所需要的块。这些块可能来自文件系统中完全无关的文件。节点聚集在一起进行市场交易。

While the notion of a barter system implies a virtual currency could be created, this would require a global ledger to

track ownership and transfer of the currency. This can be implemented as a BitSwap Strategy, and will be explored in

a future paper.

虽然易货制度的概念意味着可以创建虚拟货币,但这就需要一个全球分类账来追踪货币的所有权和转移。这可以作为一个bitswap战略来实现，并将在未来的论文中探索

In the base case, BitSwap nodes have to provide direct value to each other in the form of blocks. This works fine when the distribution of blocks across nodes is complementary, meaning they have what the other wants. Often, this

will not be the case. In some cases, nodes must work for their blocks. In the case that a node has nothing that its

peers want (or nothing at all), it seeks the pieces its peers want, with lower priority than what the node wants itself.

This incentivizes nodes to cache and disseminate rare pieces,even if they are not interested in them directly.

在基本案例中，BitSwap节点必须以块的形式向对方提供直接的值。当跨节点的块分配是互补的时，这种方法很好，意思是节点正好有其他想要的。通常情况下，并非如此。在某些情况下，节点必须为它们的块工作。在一个节点没有它的对等节点想要的块的情况下（或该节点根本没有块），它将以比本身节点需求更低的优先级去寻找它的对等节点想要的部分。这鼓励节点缓存和传播稀有部分，即使节点本身对这些稀有部分不感兴趣。

### 3.4.1 BitSwap Credit

The protocol must also incentivize nodes to seed when they do not need anything in particular, as they might have

the blocks others want. Thus, BitSwap nodes send blocks to their peers optimistically, expecting the debt to be repaid.

But leeches (free-loading nodes that never share) must be protected against. A simple credit-like system solves the

problem:

当节点不需要块的时候，该协议还必须激励节点做种子，因为他们可能拥有其他节点想要的块。

因此，BitSwap节点能乐观地发送块给对等点，并预计需要偿还的债务。但水蛭（免费加载节点不共享）必须防止。一个简单的信用状系统解决了这个问题：

1. Peers track their balance (in bytes verified) with other nodes.

2. Peers send blocks to debtor peers probabilistically, according to a function that falls as debt increases.

Note that if a node decides not to send to a peer, the node subsequently ignores the peer for an ignore\_cooldown timeout. This prevents senders from trying to game the probability by just causing more dice-rolls. (Default BitSwap is

10 seconds).

1. 对等节点跟踪与其他节点的平衡（以字节验证）。
2. 对等节点概率性地发送块给债务人节点，根据随债务增加而递减的函数。

注意，如果一个节点决定不发送给对等节点，这个节点在一个ignore\_cooldown 超时随后忽略了对等节点.这可以防止发送者试图通过掷更多的骰子来进行游戏。（默认bitswap 是10秒）

### 3.4.2 BitSwap Strategy

The differing strategies that BitSwap peers might employ have wildly different effects on the performance of the exchange as a whole. In BitTorrent, while a standard strategy is specified (tit-for-tat), a variety of others have been

implemented, ranging from BitTyrant [8] (sharing the least possible), to BitThief [8] (exploiting a vulnerability and never share), to PropShare [8] (sharing proportionally). A range of strategies (good and malicious) could similarly be implemented by BitSwap peers. The choice of function, then,should aim to:

1. maximize the trade performance for the node, and the whole exchange

2. prevent freeloaders from exploiting and degrading the exchange

3. be effective with and resistant to other, unknown strategies

4. be lenient to trusted peers

3.4.2 BitSwap 策略

BitSwap对等点采用不同的策略可能会对整体交换的性能产生很大的影响。在BitTorent,，当一个标准被指定（tit-for-tat），各种其他的东西也随之实现，从BitTyrant [ 8 ]（尽可能少共享），到bitthief [ 8 ]（利用一个漏洞并且不共享），以propshare [ 8 ]（按比例分配）。一系列的策略（好的和恶意的）同样可以被BitSwap对等点实现。函数选择的目标是：

1. 最大化节点以及整个交换的交易表现
2. 防止贪图便宜的节点利用和降低交换
3. 有效抵抗其他未知的策略
4. 对可信节点宽容

The exploration of the space of such strategies is future work. One choice of function that works in practice is a

sigmoid, scaled by a debt retio:

这样的策略空间的探索是未来的工作。在实际可使用的一个可选的函数是sigmoid.

Let the *debt* ratio *r* between a node and its peer be:

一个节点和它的对等节点的负债率r:

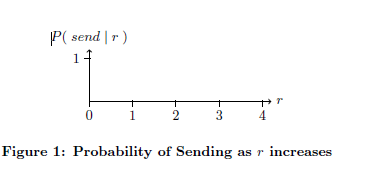


给定r, 发送给一个债务人的概率为：



As you can see in Figure 1, this function drops off quickly as the nodes' debt ratio surpasses twice the established credit.

如图1所示，当节点的债务比率超过既定信用额的两倍时，这个函数就会迅速下降。



The debt ratio is a measure of trust: lenient to debts between nodes that have previously exchanged lots of data successfully, and merciless to unknown, untrusted nodes. This (a)provides resistance to attackers who would create lots of new nodes (sybill attacks), (b) protects previously successful

trade relationships, even if one of the nodes is temporarily unable to provide value, and (c) eventually chokes relationships that have deteriorated until they improve.

债务比率是一种信任度量：对以前成功交换大量数据的节点之间的债务进行宽松，对未知的、不可信的节点严格。这样可以（a）提供抵抗会创造很多新的节点的攻击（Sybill攻击），（b）保护以前成功的交易关系，即使其中一个节点暂时无法提供价值。(c)最终阻塞己经恶化的节点，直到节点改善才恢复。

### 3.4.3 BitSwap Ledger

3.4.3 BitSwap分类账

BitSwap nodes keep ledgers accounting the transfers with other nodes. This allows nodes to keep track of history and

avoid tampering. When activating a connection, BitSwap nodes exchange their ledger information. If it does not

match exactly, the ledger is reinitialized from scratch, losing the accrued credit or debt. It is possible for malicious nodes to purposefully \lose" the Ledger, hoping to erase debts. It is unlikely that nodes will have accrued enough debt to warrant also losing the accrued trust; however the partner node is free to count it as misconduct, and refuse to trade.

BitSwap节点用分类账记述与其他节点的传输。这允许节点跟踪，避免篡改历史。当激活一个连接时，BitSwap节点交换他们的分类账信息,如果它不完全匹配，则分类账从零开始重新初始化，失去积累的信用或债务。可能恶意节点有可能故意“丢失”分类账，希望抹去债务。节点累加足够多的债务来保证失去累加的信任是不太可能的；然而伙伴节点可以自由地认为它处理不当，拒绝交易。

*type Ledger struct {*

*owner NodeId*

*partner NodeId*

*bytes\_sent int*

*bytes\_recv int*

*timestamp Timestamp*

*}*

Nodes are free to keep the ledger history, though it is not necessary for correct operation. Only the current ledger

entries are useful. Nodes are also free to garbage collect ledgers as necessary, starting with the less useful ledgers:

the old (peers may not exist anymore) and small.

节点可以自由保存分类帐记录，但不需要正确操作。只有当前的分类账入口是有用的,必要的时候节点可以从无用的分类账开始，自由地对分类账进行垃圾回收：老的（对等节点可以不存在了）和小的。

### 3.4.4 BitSwap Specification

3.4.4 BitSwap说明书

BitSwap nodes follow a simple protocol.

BitSwap节点遵循下面简单的协议

*// 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);*

*}*

Sketch of the lifetime of a peer connection:

1. Open: peers send ledgers until they agree.

2. Sending: peers exchange want\_lists and blocks.

3. Close: peers deactivate a connection.

4. Ignored: (special) a peer is ignored (for the duration of a timeout) if a node's strategy avoids sending

一个对等节点生命期的草图：

1. 打开：对等节点之间发送分类账直到它他们同意
2. 发送：对等节点之间交换want\_list和块
3. 关闭：对等点关闭连接
4. 忽略：（特殊情况）如果节点的策略避免发送，则忽略对等节点（在超时期间）。

**Peer.open(NodeId, Ledger).**

When connecting, a node initializes a connection with a Ledger, either stored from a connection in the past or a new one zeroed out. Then, sends an Open message with the Ledger to the peer.

Upon receiving an Open message, a peer chooses whether to activate the connection. If - acording to the receiver's

Ledger -the sender is not a trusted agent (transmission below zero, or large outstanding debt) the receiver may opt

to ignore the request. This should be done probabilistically with an ignore\_cooldown timeout, as to allow errors to be

corrected and attackers to be thwarted.

If activating the connection, the receiver initializes a Peer object with the local version of the Ledger and sets the

last\_seen timestamp. Then, it compares the received Ledger with its own. If they match exactly, the connections have

opened. If they do not match, the peer creates a new zeroed out Ledger and sends it.

连接的时候，节点用分类账初始化连接,分类账要么是从过去的连接中存储的要么是一个新清零后的。然后和分类账一起发送一个开放的消息给对等节点。

当接收到一个开放的消息的时候，对等节点选择是否激活连接。如果-根据接收方的分类账-发送方不是一个可信任的代理(传输在零以下，或者有大量未偿债务)接收方可以选择忽略这个请求。由于要允许使错误得到纠正，攻击者受挫,因此这应该在一个ignore\_cooldown 超时内概率性地完成。

如果激活连接，接收者用本地版本的分类账初始化一个对等节点对象，并且设置last\_seen 时间戳。

**Peer.send\_want\_list(WantList).**

While the connection is open, nodes advertise their want\_list to all connected peers. This is done (a) upon opening the connection, (b) after a randomized periodic timeout, (c) after a change in the want\_list and (d) after receiving a new

block.

Upon receiving a want\_list, a node stores it. Then, it checks whether it has any of the wanted blocks. If so, it sends them according to the BitSwap Strategy above.

当连接打开，节点把他们的want\_list通知给所有己连接上的对等节点。这在(a)打开连接的基础上(b)随机周期超时之后（c）want\_list改变之后（d）接收到一个新块之后完成。

当一个节点接收到一个want\_list之后，将之存储。然后，节点检查它是否拥有任何想要的块。如果有，它会根据以上的BitSwap策略发送这些块

**Peer.send\_block(Block).**

Sending a block is straightforward. The node simply transmits the block of data. Upon receiving all the data, the re-

ceiver computes the Multihash checksum to verify it matches the expected one, and returns confirmation.

Upon finalizing the correct transmission of a block, the receiver moves the block from need\_list to have\_list, and

both the receiver and sender update their ledgers to reflect the additional bytes transmitted.

If a transmission verification fails, the sender is either malfunctioning or attacking the receiver. The receiver is free to refuse further trades. Note that BitSwap expects to operate on a reliable transmission channel, so transmission errors

{ which could lead to incorrect penalization of an honest sender { are expected to be caught before the data is given

to BitSwap.

发送一个块是直截了当的，节点简单地发送数据块。在接收到的所有数据之后，接收方计算校验和来验证接收到的数据与是所预期的,并返回确认。

在完成了一个块的正确传输之后，接收方把块从need\_list移动到have\_list，接收方和发送方同时更新他们的分类账以反映额外传输的字节数。

如果传输验证失败，发送方要么出现故障，要么正在攻击接收方。接收方可以自由拒绝继续交易。

注意，bitswap期望在一个可靠的传输通道操作，因此-可能导致一个诚实的发送方被错误惩罚的传输错误,希望在数据被交给BitSwap之前被发现。

**Peer.close(Bool).**

The final parameter to close signals whether the intention to tear down the connection is the sender's or not. If

false, the receiver may opt to re-open the connection immediatelty. This avoids premature closes.

A peer connection should be closed under two conditions:

* a silence\_wait timeout has expired without receiving any messages from the peer (default BitSwap uses 30

seconds). The node issues Peer.close(false).

* the node is exiting and BitSwap is being shut down.In this case, the node issues Peer.close(true).

After a close message, both receiver and sender tear down the connection, clearing any state stored. The Ledger may be stored for the future, if it is useful to do so.

Notes.

* Non-open messages on an inactive connection should be ignored. In case of a send\_block message, the receiver may check the block to see if it is needed and correct, and if so, use it. Regardless, all such out-of-order messages trigger a close(false) message from the receiver to force re-initialization of the connection.

是否想要关闭连接的最终参数是发送方的，或者不是。如果不是，接收方可能选择马上重新打开连接。这避免了过早关闭。

一个对等连接应该在以下两种条件下关闭：

* Silence\_wait 超时己经过期但还没有从对等节点接收到任何消息（默认的BitSwap使用30s）。节点宣布Peer.close(false)
* 节点退出并且BitSwap被关闭。在这种情况下，节点宣布Peer.close(false)

在关闭消息之后，接收者和发送者拆毁连接，清除所有存储的状态。分类账如果有必要可能存储下来。

注意：

* 应忽略非活动连接上的非公开消息。Send\_block消息的情况下，接收者可能检查块是否是必要的和正确，如果有必要，则使用它。无论如何，所有这样的无序消息都会从接收方触发关闭（false）消息，迫使连接重新初始化。

## 3.5 Object Merkle DAG

The DHT and BitSwap allow IPFS to form a massive peer-to-peer system for storing and distributing blocks quickly and robustly. On top of these, IPFS builds a Merkle DAG, a directed acyclic graph where links between objects are cryptographic hashes of the targets embedded in the sources.This is a generalization of the Git data structure. Merkle DAGs provide IPFS many useful properties, including:

1. Content Addressing: all content is uniquely identified by its multihash checksum, including links.

2. Tamper resistance: all content is veri\_ed with its checksum. If data is tampered with or corrupted, IPFS detects it.

3. Deduplication: all objects that hold the exact same content are equal, and only stored once. This is particularly useful with index objects, such as git trees and commits, or common portions of data.

The IPFS Object format is:

DHT和BitSwap 允许IPFS快速和有力地形成一个庞大的P2P系统储存和分配块。除这些之外，IPFS创建了Merkle DAG,这是一个有向非循环图，对象之间的连接是嵌入在源中的目标的加密哈希。这是Git数据结构的归纳。Merkle DAG提供IPFS许多有用的属性，包括：

1. 内容寻址：所有的内容都是由其multihash校验唯一标识，包括链接。
2. 防篡改：所有内容都用校验和验证。如果数据被篡改或损坏，IPFS都会检测到。
3. 重复数据删除：持有相同内容的所有对象都是平等的，只存储一次。这对于索引对象（例如Git树和提交）或数据的公共部分特别有用。

IPFS对象的格式是：

*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 {*

*inks []IPFSLink*

*// array of links*

*data []byte*

*// opaque content data*

*}*

The IPFS Merkle DAG is an extremely flexible way to store data. The only requirements are that object references be (a) content addressed, and (b) encoded in the format above. IPFS grants applications complete control over the data field; applications can use any custom data format they chose, which IPFS may not understand. The separate in object link table allows IPFS to:

IPFS Merkle DAG是存储数据的一个非常灵活的方式。唯一的要求是对象引用是（a）内容寻址，（b）以上述格式编码。规授予数据域上的应用完全控制，应用程序可以使用他们选择的任何自定义数据格式，IPFS可能不理解。对象

* List all object references in an object. For example:

列出一个对象中的所有引用：

> ipfs ls /XLZ1625Jjn7SubMDgEyeaynFuR84ginqvzbXLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x 189458 lessXLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5 19441 script XLF4hwVHsVuZ78FZK6fozf8Jj9WEURMbCX4 5286 template

<object multihash> <object size> <link name>

* Resolve string path lookups, such as foo/bar/baz. Given an object, IPFS resolves the first path component to a hash in the object's link table, fetches that second object, and repeats with the next component. Thus, string paths can walk the Merkle DAG no matter what the data formats are.
* 解析字符串路径查找，像foo/bar/bax.给定一个对象，IPFS把路径第一部分分解为对象的链表中的哈希，然后获取第二个对象，重复上述操作。因此，不管数据格式是什么，字符串路径都能遍历Merkle DAG.
* Resolve all objects referenced recursively:
* 解析递归引用的所有对象

> ipfs refs --recursive \

/XLZ1625Jjn7SubMDgEyeaynFuR84ginqvzb

XLLxhdgJcXzLbtsLRL1twCHA2NrURp4H38s

XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x

XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5

XLWVQDqxo9Km9zLyquoC9gAP8CL1gWnHZ7z

...

A raw data field and a common link structure are the necessary components for constructing arbitrary data structures on top of IPFS. While it is easy to see how the Git object model fits on top of this DAG, consider these other potential data structures: (a) key-value stores (b) traditional relational databases (c) Linked Data triple stores (d) linked document publishing systems (e) linked communications platforms (f) cryptocurrency blockchains. These can all be modeled on top of the IPFS Merkle DAG, which allows any of these systems to use IPFS as a transport protocol for more complex applications.

原始数据字段和一个普通的链结构是在IPFS上构建任意数据结构的必要组成部分. 虽然很容易看出Git对象模型如何适合于DAG，但考虑到其他潜在的这些数据结构：（a）键值存储（b）传统的关系型数据库(c) 链接数据三元组（d）链接文档发布系统（E）链接的通信平台（F）加密货币区块链。这些都可以在IPFS Merkle DAG上建模，它允许任何这些系统使用IPFS作为更复杂的应用程序的传输协议。

## 3.5.1 Paths

IPFS objects can be traversed with a string path API.Paths work as they do in traditional UNIX filesystems and the Web. The Merkle DAG links make traversing it easy Note that full paths in IPFS are of the form:

IPFS 可以通过字符串路径API遍历。路径像传统的UNIX文件系统和网络那样工作。Merkle DAG使得遍历容易。注意IPFS中的全路径是以下形式：

# format

/ipfs/<hash-of-object>/<name-path-to-object>

# example

/ipfs/XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x/foo.txt

The /ipfs prefix allows mounting into existing systems at a standard mount point without conflict (mount point names are of course configurable). The second path component (first within IPFS) is the hash of an object. This is always the case, as there is no global root. A root object would have the impossible task of handling consistency of millions of objects in a distributed (and possibly disconnected) environment. Instead, we simulate the root with content addressing. All objects are always accessible via their hash. Note this means that given three objects in path <foo>/bar/baz, the last object is accessible by all:

/ipfs前缀允许挂载到现有系统中的一个没有冲突的标准挂载点（挂载点名称当然是可配置的）。路径的第二个组成部分（IPFS中的第一部分）是一个对象的哈希。情况总是如此，因为没有全球根源。根对象在分布式环境（可能断开的环境）中处理数百万对象的一致性，这是不可能完成的任务。相反，我们用内容寻址模拟根目录。所有对象都可以通过他们的哈希访问。注意这意味着给定的路径<foo>/bar/baz对象，最后的对象可以通过以下方式访问：

```

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

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

/ipfs/<hash-of-baz>

```

### 3.5.2 Local Objects

本地对象

IPFS clients require some local storage, an external system on which to store and retrieve local raw data for the objects IPFS manages. The type of storage depends on the node's use case. In most cases, this is simply a portion of disk space(either managed by the native filesystem, by a key-value store such as leveldb [4], or directly by the IPFS client). In others, for example non-persistent caches, this storage is just a portion of RAM.

IPFS客户端需要一些本地存储、在外部系统上存储和检索由IPFS管理的本地的原数据对象。存储的类型取决于节点的使用情况。在大数情况下，只是磁盘空间的一部分（无论是由本地文件系统，还是由一个键值存储诸如LevelDB [ 4 ]，或直接由IPFS客户端管理）。在其他情况下，例如非持久缓存，这个存储只是RAM的一部分。

Ultimately, all blocks available in IPFS are in some node's local storage. When users request objects, they are found, downloaded, and stored locally, at least temporarily. This provides fast lookup for some configurable amount of time there after.

最终，IPFS所有可用的块都位于某些节点的本地存储中。当用户请求对象时，它们被发现、下载和至少暂时存储在本地。这提供了对某些可配置时间的快速查找。

### 3.5.3 Object Pinning

对象钉固

Nodes who wish to ensure the survival of particular objects can do so by pinning the objects. This ensures the objects are kept in the node's local storage. Pinning can be done recursively, to pin down all linked descendent objects as well. All objects pointed to are then stored locally. This is particularly useful to persist files, including references.This also makes IPFS a Web where links are permanent,and Objects can ensure the survival of others they point to.

希望确保特定对象生存的节点可以通过钉固对象来实现。这样可以确保对象保存在节点的本地存储中。钉固可以递归的实现，也可向下钉固住所有相连的子孙对象。指向的所有对象会被存储在本地。这对于保存文件（包括引用）特别有用。这也使IPFS成为链接永久的一个网络，对象可以确保他们指向的其他对象的保持存活。

### 3.5.4 Publishing Objects

发布对象

IPFS is globally distributed. It is designed to allow the files of millions of users to coexist together. The DHT, with content-hash addressing, allows publishing objects in a fair, secure, and entirely distributed way. Anyone can publish an object by simply adding its key to the DHT, adding themselves as a peer, and giving other users the object's path.Note that Objects are essentially immutable, just like in Git. New versions hash differently, and thus are new objects. Tracking versions is the job of additional versioning objects.

IPFS是全球分布的。它的目的是允许数百万用户的文件共存。使用内容哈希寻址的DHT允许以公平、安全和完全分布式的方式发布对象。任何人都可以通过简单地将它的密钥添加到DHT，将自己添加为一个对等体，并给其他用户对象的路径来发布对象。注意，就像在Git中一样，对象本质上是不可变的。新版本哈希不同，因此是新对象。跟踪版本是另外的版本控制对象的工作。

### 3.5.5 Object-level Cryptography

IPFS is equipped to handle object-level cryptographic operations. An encrypted or signed object is wrapped in a special frame that allows encryption or verification of the raw bytes.

IPFS 设置成能处理对象级的加密操作。一个加密的或签名过的对象被包装在一个特殊的框架中，允许对原始字节加密或验证。

```

type EncryptedObject struct {

Object []bytes

// raw object data encrypted

// 原对象加密后的数据

Tag []bytes

// optional tag for encryption groups

// 加密组的可选标签

}

type SignedObject struct {

Object []bytes

// raw object data signed

// 原对象签名后的数据

Signature []bytes

// hmac signature

// 签名

PublicKey []multihash

// multihash identifying key

// 验证密钥

}

```

Cryptographic operations change the object's hash, defining a different object. IPFS automatically verifies signatures, and can decrypt data with user-specied keychains.Links of encrypted objects are protected as well, making traversal impossible without a decryption key. It is possible to have a parent object encrypted under one key, and a child under another or not at all. This secures links to shared objects.

加密操作改变对象的哈希，定义了一个不同的对象。IPFS自动地验证签名，并能使用用户指定的钥匙链解密数据。加密后的对象的链接也被保护，没有解密密钥是不可能解析的。用一个密钥加密父对象子，对象用另外一个密钥加密或者根本不加密子对象，这是可能的。这确保链接用共享的对象。

## 3.6 Files

3.6 文件

IPFS also defines a set of objects for modeling a versioned filesystem on top of the Merkle DAG. This object model is similar to Git's:

1. block: a variable-size block of data.

2. list: a collection of blocks or other lists.

3. tree: a collection of blocks, lists, or other trees.

4. commit: a snapshot in the version history of a tree.

IPFS也对Merkle DAG上的版本化的文件系统建模定义了一组对象。该对象模型与Git类似：

1. 块：可变大小的数据块。

2. 列表：块的集合或是其他的列表的集合。

3. 树：块的集合，列表的集合，或其他树的集合。

4. 提交：树的版本历史的快照 。

I hoped to use the Git object formats exactly, but had to depart to introduce certain features useful in a distributed filesystem, namely (a) fast size lookups (aggregate byte sizes have been added to objects), (b) large file deduplication

(adding a list object), and (c) embedding of commits into trees. However, IPFS File objects are close enough to Git that conversion between the two is possible. Also, a set of Git objects can be introduced to convert without losing any information (unix file permissions, etc).

虽然我们希望完全使用Git形式的对象，但是必要分开介绍在分布式文件系统中有用的特定特征。(a)快速大小查找（己经添加到对象中的字节大小总计）（b）大文件去重（添加一个列表对象）（c）提交到树中的嵌入。

IPFS对象和Git对象十分接近，以至于他们两者之间可以相互转换。一组Git对象在不丢失任可信息的情况下可以转化为IPFS对象(unix文件允许)。

Notation: File object formats below use JSON. Note that this structure is actually binary encoded using protobufs,though ipfs includes import/export to JSON.

注意：下面文件对象的格式使用JSON。注意，虽然IPFS包括import/export到JSON，该结构实际上使用的是protobufs进行二进制编码。

### 3.6.1 File Object: blob

3.6.1 文件对象：blob

>The blob object contains an addressable unit of data, and represents a file. IPFS Blocks are like Git blobs or filesystem

data blocks. They store the users' data. Note that IPFS files can be represented by both lists and blobs. Blobs have no links.

blob对象包含一个可地址化的数据单元，代表一个文件。IPFS块像Git blob或文件系统数据块。他们都存储用户的数据。注意IPFS文件可以表示为列表或blob。blob没有链接。

```

{

"data": "some data here",

// blobs have no links

}

```

### 3.6.2 File Object: list

The list object represents a large or deduplicated file made up of several IPFS blobs concatenated together. Lists contain an ordered sequence of blob or list objects. In a sense, the IPFS list functions like a filesystem file with indirect blocks. Since lists can contain other lists, topologies including linked lists and balanced trees are possible.Directed graphs where the same node appears in multiple places allow in-file deduplication. Of course, cycles are not possible, as enforced by hash addressing.

列表对象代表一个大的或去重的由几个IPFS blob串联在一起的文件。列表包含一个有序的blob或列表对象序列。在某种意义上，IPFS列表的功能就像一个间接块的文件系统的文件。由于列表可以包含其他列表，所以包含链表和平衡树的拓扑是可能的。相同的节点出现在多个地方的有向图中允许在文件中去重。当然，哈希寻址强制循环是不可能的。

```

{

"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 File Object: tree

The tree object in IPFS is similar to Git's: it represents a directory, a map of names to hashes. The hashes reference

blobs, lists, other trees, or commits. Note that traditional path naming is already implemented by the Merkle DAG.

IPFS中的树对象与Git类似:它代表一个目录，一个名字到哈希的映射。哈希引用blob,list,其他的树，或者commit。值得注意的是，传统的路径名称已经由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 File Object: commit

The commit object in IPFS represents a snapshot in the version history of any object. It is similar to Git's, but can reference any type of object. It also links to author objects.

IPFS中的一个提交对象代表任意对象版本历史的快照 。和Git类似，但是能引用任意类型的对象。它也能连接到另外的一个对象。

```

{

"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 }

]

}

```

```

> ipfs file-cat <ccc111-hash> --json

{

"data": {

"type": "tree",

"date": "2014-09-20 12:44:06Z",

"message": "This is a commit message."

},

"links": [

{ "hash": "<ccc000-hash>",

"name": "parent", "size": 25309 },

{ "hash": "<ttt111-hash>",

"name": "object", "size": 5198 },

{ "hash": "<aaa111-hash>",

"name": "author", "size": 109 }

]

}

> ipfs file-cat <ttt111-hash> --json

{

"data": ["tree", "tree", "blob"],

"links": [

{ "hash": "<ttt222-hash>",

"name": "ttt222-name", "size": 1234 },

{ "hash": "<ttt333-hash>",

"name": "ttt333-name", "size": 3456 },

{ "hash": "<bbb222-hash>",

"name": "bbb222-name", "size": 22 }

]

}

> ipfs file-cat <bbb222-hash> --json

{

"data": "blob222 data",

"links": []

}

```

### 3.6.5 Version control

3.6.5 版本控制

>The commit object represents a particular snapshot in the version history of an object. Comparing the objects (and children) of two different commits reveals the differences between two versions of the filesystem. As long as a single commit and all the children objects it references are accessible, all receding versions are retrievable and the full history of the filesystem changes can be accessed. This falls out of the Merkle DAG object model.

提交对象表示对象版本历史中的特定快照。比较两个不同提交的对象（和孩子）揭示了两个版本的文件系统之间的差异。只要单个提交和它引用的所有的孩子对象是可访问的，所有的滚动版本都可获取，文件系统改变的整个历史都能访问到。这由于Merkle DAG对象模型 。

>The full power of the Git version control tools is available to IPFS users. The object model is compatible, though not the same. It is possible to (a) build a version of the Git tools modified to use the IPFS object graph, (b) build a mounted FUSE filesystem that mounts an IPFS tree as a Git repo,translating Git filesystem read/writes to the IPFS formats.

在Git版本控制工具的所有功能都可供IPFS用户使用。对象模型是兼容的，但不是相同的。（a）构建一个修改的Git工具版本，以使用IPFS对象图.（b）构建一个挂载的FUSE文件系统，该文件系统挂载IPFS树作为一个Git repo,把Git文件系统的读/写翻译为IPFS形式，都是可能了。

### 3.6.6 Filesystem Paths

3.6.6 文件系统路径

>As we saw in the Merkle DAG section, IPFS objects can be traversed with a string path API. The IPFS File Objects are designed to make mounting IPFS onto a UNIX filesystem simpler. They restrict trees to have no data, in orde to represent them as directories. And commits can either be represented as directories or hidden from the filesystem entirely.

正如我们在Merkle DAG部分所看到了，IPFS对象可以通过字符口中 路径API遍历。IPFS文件对象被设计在将其更简单地挂载IPFS到UNIX文件系统上。他们限制树没有数据，为了将它们表示为目录。提交可以表示为目录，也可以完全隐藏在文件系统中。

### 3.6.7 Splitting Files into Lists and Blob

3.6.7 将文件分割成列表和blob

>One of the main challenges with versioning and distributing large files is finding the right way to split them into independent blocks. Rather than assume it can make the right decision for every type of file, IPFS offers the following alternatives:

版本控制和分发大文件的主要挑战之一是找到正确的方法将它们分成独立的块。与其假设它能对每一种类型的文件做出正确的决定，IPFS提供以下选择：

(a) Use Rabin Fingerprints [?] as in LBFS [?] to pick suitable block boundaries.

(b) Use the rsync [?] rolling-checksum algorithm, to detect blocks that have changed between versions.

(c) Allow users to specify block-splitting functions highly tuned for specific files.

(a)在LBFS中用Rabin Fingerprints挑选合适的块边界。

(b)使用远程滚动校验算法，检测两个版本之间有改变的块。

(c)允许用户对特点的文件指定高度调谐的块分割函数。

### 3.6.8 Path Lookup Performance

3.6.8 路径查找性能

>Path-based access traverses the object graph. Retrieving each object requires looking up its key in the DHT, connecting to peers, and retrieving its blocks. This is considerable overhead, particularly when looking up paths with many components. This is mitigated by:

基于路径的访问遍历对象图。检索每个对象需要在DHT中查找它的密钥。连接到对等点并检索它的块。这是相当大的开销，尤其是在查找具有多个部分的路径时。这是通过以下减轻的：

+ tree caching: since all objects are hash-addressed,they can be cached indenitely. Additionally, trees tend to be small in size so IPFS prioritizes caching them over blobs.

+ flattened trees: for any given tree, a special flattened tree can be constructed to list all objects reachable from the tree. Names in the flattened tree would really be paths parting from the original tree, with slashes.

>For example, flattened tree for ttt111 above:

+ 树缓存：因为所有对象都是哈希寻址的，它们肯定能被缓存。另外，树偏重于小规模以便IPFS在blob上能优点缓存它们。

+ 扁平的树：对于任意给定的树，可以构建一个特定的扁平树来列出从树出来所有可以到达的对象。扁平树中的名称可能真的是从源始的树中分离的路径，包括斜杠。

例如，对于上面的ttt111的扁平树为：

```

{

"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: Naming and Mutable State

3.7 IPNS:命名和可变状态

>So far, the IPFS stack forms a peer-to-peer block exchange constructing a content-addressed DAG of objects. It serves to publish and retrieve immutable objects. It can even track the version history of these objects. However, there is a critical component missing: mutable naming. Without it,all communication of new content must happen off-band,sending IPFS links. What is required is some way to retrieve mutable state at the same path.

到目前为止，IPFS堆栈形成了一个对等块交换，构造对象的内容寻址DAG。它用于发布和检索不可变对象。它甚至可以跟踪这些对象的版本历史。但是，缺少一个关键组件：可变的命名。没有它，所有新内容的通信都必须解绑，发送IPFS链接。所需的是在同一路径上检索可变状态的某种方法。

>It is worth stating why -if mutable data is necessary in the end -we worked hard to build up an immutable Merkle DAG. Consider the properties of IPFS that fall out of the Merkle DAG: objects can be (a) retrieved via their hash, (b) integrity checked, (c) linked to others, and (d) cached indefinitely. In a sense:

值得说明的是为什么如果在端的可变数据是必要的，我们努力建立一个不变的Merkle DAG。考虑到从Merkle DAG分离出来的IPFS的性能：对象可以（a）通过它们的哈希被检索 （b）完整性检查（c）连接到其他对象(d)缓存。在某种意义上：

```

Objects are permanent

对象是永久性的

```

>These are the critical properties of a high-performance distributed system, where data is expensive to move across network links. Object content addressing constructs a web with(a) significant bandwidth optimizations, (b) untrusted content serving, (c) permanent links, and (d) the ability to make full permanent backups of any object and its references.

这些都是高性能分布式系统的关键特性，在跨网络链路移动时，数据很昂贵。对象内容寻址构造具有（a）显著带宽优化的（b）不受信任的内容服务(c) 永久链接和（d）对任何对象及其引用进行完全永久备份的能力的Web.

>The Merkle DAG, immutable content-addressed objects,and Naming, mutable pointers to the MerkleDAG, instantiate a dichotomy present in many successful distributed systems. These include the Git Version Control System, with its immutable objects and mutable references; and Plan9 [?],the distributed successor to UNIX, with its mutable Fossil[?] and immutable Venti [?] filesystems. LBFS [?] also uses mutable indices and immutable chunks.

Merkle DAG，不可改变的内容寻址和命名的对象，可变指向Merkle DAG，许多成功的分布式系统中存在的二分法的实例。这些包括Git版本控制系统。使用不可变对象和可变引用。Plan9是UNIX的分布式继任者，有不可变的Fossil和可变的Venti文件系统。LBFS也使用可变索引和不可变块。

### 3.7.1 Self-Certified Names

3.7.1 自认证的名字

>Using the naming scheme from SFS [12, 11] gives us a way to construct self-certified names, in a cryptographically assigned global namespace, that are mutable. The IPFS scheme is as follows.

使用SFS的命名方案提供了一种构造自认证名称的方法,在一个加密的指定全局命名空间，这是可变的。IPFS方案如下：

1. Recall that in IPFS（在IPFS中召回）

```

NodeId = hash(node.PubKey)

```

2. We assign every user a mutable namespace at:(我们给每个用户分配一个可变名称空间：)

```

/ipns/<NodeId>

```

3. A user can publish an Object to this path Signed by her private key, say at:(用户可以将对象发布到由其私钥签名的路径上，例如)

```

/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/

```

4. When other users retrieve the object, they can check the signature matches the public key and NodeId. This verifies the authenticity of the Object published by the user, achieving mutable state retrival.

当其他用户检索对象的时候，他们可以检查匹配的公共密钥和节点ID的签名，这验证了由用户发布的对象的真实性，实现可变状态检索。

>Note the following details:

（注意下面的细节）

+ The ipns (InterPlanetary Name Space) separate prefix is to establish an easily recognizable distinction between mutable and immutable paths, for both programs and human readers.

+ Because this is not a content-addressed object, publishing it relies on the only mutable state distribution system in IPFS, the Routing system. The process is (1) publish the object as a regular immutable IPFS object, (2) publish its hash on the Routing system as a metadata value:

+ IPNS的独立前缀是为程序和人类读者建立可变的和不可变路径之间容易识别的区别。

+ 因为这不是一个内容寻址对象，发布它依靠IPFS中的唯一的可变状态分布系统，路由系统。过程是（1）作为一个合格的不可变的IPFS对象发布（2）在路由系统为作为一个元数据值发布它的哈希。

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

>Any links in the Object published act as sub-names in the namespace:

发布的对象中的任何链接都充当名称空间中的子名称：

```

/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/

/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/docs

/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/docs/ipfs

```

>it is advised to publish a commit object, or some other object with a version history, so that clients may be

able to find old names. This is left as a user option, as it is not always desired.

建议发布一个提交对象或具有版本历史的其他对象，以便客户机能够找到旧名称。这是作为用户选项留下的，因为它并不总是需要的。

>Note that when users publish this Object, it cannot be published in the same way

注意，当用户发布此对象时，它不能以相同的方式发布。

### 3.7.2 Human Friendly Names

3.7.2 对人类友好的名称

>While IPNS is indeed a way of assigning and reassigning names, it is not very user friendly, as it exposes long

hash values as names, which are notoriously hard to remember. These work for URLs, but not for many kinds of offline transmission. Thus, IPFS increases the user-friendliness of IPNS with the following techniques.

虽然这的确是分配和再分配的名字的方式，但它不是非常用户友好的，因为它以暴露的长哈希值作为名字，这这是出了名的难记。这些对于URL是有用的，但不适用于多种脱机传输。因此，IPFS使用以下技术增加了用户的友好性：

**Peer Links.**

对等链接

>As encouraged by SFS, users can link other users' Objects directly into their own Objects (namespace, home, etc).

This has the benefit of also creating a web of trust (and supports the old Certificate Authority model):

在SFS的鼓励下，用户可以直接将其他用户的对象链接到自己的对象（名称空间、家等）中。这也有利于建立信任网（支持旧的证书权威模型）。

```

# 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 Records.**

>If /ipns/<domain> is a valid domain name, IPFS looks up key ipns in its DNS TXT records. IPFS interprets the value

as either an object hash or another IPNS path:

如果/ipns/<domain>是一个有效的域名，IPFS在它的DNS TXT记录中查找密码ipns. IPFS 作为一个对象哈希或作为另外一个IPNS路径解释值。

```

# this DNS TXT record

ipfs.benet.ai. TXT "ipfs=XLF2ipQ4jD3U ...

# behaves as symlink

ln -s /ipns/XLF2ipQ4jD3U /ipns/fs.benet.ai

```

**Proquint Pronounceable Identifiers.**

Proquint可断言的标识符

>There have always been schemes to encode binary into pronounceable words. IPNS supports Proquint [?]. Thus:

总是有把二进制编码为可断言的单词的体系。IPNS支持Proquint，因此：

```

# this proquint phrase

/ipns/dahih-dolij-sozuk-vosah-luvar-fuluh

# will resolve to corresponding

/ipns/KhAwNprxYVxKqpDZ

```

**Name Shortening Services**.

名称缩短服务。

>Services are bound to spring up that will provide name shortening as a service, offering up their namespaces to users.

This is similar to what we see today with DNS and Web URLs:

服务绑定到spring ，提供名称缩短的服务，把他们的名称空间提供给用户。和我们今天所看到的DNS和 Web URLs类似。

```

# User can get a link from

/ipns/shorten.er/foobar

# To her own namespace

/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm

```

## 3.8 Using IPFS

3.8 使用IPFS

>IPFS is designed to be used in a number of different ways.Here are just some of the usecases I will be pursuing:

1. As a mounted global filesystem, under /ipfs and /ipns.

2. As a mounted personal sync folder that automatically versions, publishes, and backs up any writes.

3. As an encrypted file or data sharing system.

4. As a versioned package manager for all software.

5. As the root filesystem of a Virtual Machine.

6. As the boot filesystem of a VM (under a hypervisor).

7. As a database: applications can write directly to the Merkle DAG data model and get all the versioning,caching, and distribution IPFS provides.

8. As a linked (and encrypted) communications platform.

9. As an integrity checked CDN for large files (without SSL).

10. As an encrypted CDN.

11. On webpages, as a web CDN.

12. As a new Permanent Web where links do not die.

IPFS被设计成多种不同的方式使用，下面只是一些用例:

1. 作为一个挂载的全球文件系统，在/ipfs和/ipns之下

2. 作为一个挂载的个人同步文件夹，它可以自动地编辑、发布和备份任何写操作。

3. 作为加密文件或数据共享系统。

4. 作为所有软件的版本包管理器。

5. 作为虚拟机的根文件系统。

6. 作为虚拟机的启动文件系统（在虚拟机管理系统下）

7. 作为一个数据库:应用程序可以直接写进Merkle DAG数据模型并且得到的所有版本，缓存，并分配IPFS提供。

8. 作为链接（加密）的通信平台

9. 作为大文件的完整性检查的CDN（无SSL）。

10. 作为加密的CDN

11. 在网页上，作为一个web CDN

12. 作为一个新的永久网络，链接不会死亡。

>The IPFS implementations target:

IPFS实现的目标：

(a) an IPFS library to import in your own applications.

(b) commandline tools to manipulate objects directly.

(c) mounted file systems, using FUSE [?] or as kernel modules.

（a）一个导入你自己的应用程序的IPFS库。

（b）直接操作对象的命令行工具。

（c）挂载的文件系统，使用FUSE或者作为内核模块。

# 4. THE FUTURE

4.未来

> ideas behind IPFS are the product of decades of successful distributed systems research in academia and open source. IPFS synthesizes many of the best ideas from the most successful systems to date. Aside from BitSwap, which is a novel protocol, the main contribution of IPFS is this coupling of systems and synthesis of designs.

IPFS背后的想法是几十年来成功的分布式系统研究在学术界和开源的成果。它把迄今为止最成功的系统中的许多最好的想法综合起来。除了bitswap，这是一个新的协议，IPFS主要贡献是耦合这些系统以及合成设计。

>IPFS is an ambitious vision of new decentralized Internet infrastructure, upon which many different kinds of applications can be built. At the bare minimum, it can be used as a global, mounted, versioned filesystem and namespace, or as the next generation file sharing system. At its best, it could push the web to new horizons, where publishing valuable information does not impose hosting it on the publisher but upon those interested, where users can trust the content they receive without trusting the peers they receive it from, and where old but important files do not go missing. IPFS looks forward to bringing us toward the Permanent Web.

IPFS是一个雄心勃勃的新的分散互联网基础设施的构想，在此基础上可以建立许多不同类型的应用程序。在最低限度，它可以作为一个全球性的，挂载的版本文件系统的命名空间，或作为下一代文件共享系统。在其最好的情况下，它能推动web到一生片新天地，在那里发布有价值的信息并需要出版商，而是发布给那些感兴趣的人，用户可以信任他们收到的内容而不用信任从哪里收到消息，老的但重要文件不丢失。IPFS期待带我们走向永久的Web。

# 5. ACKNOWLEDGMENTS

5.感谢

>IPFS is the synthesis of many great ideas and systems. It would be impossible to dare such ambitious goals without standing on the shoulders of such giants. Personal thanks to David Dalrymple, Joe Zimmerman, and Ali Yahya for long discussions on many of these ideas, in particular: exposing the general Merkle DAG (David, Joe), rolling hash blocking (David), and s/kademlia sybill protection (David, Ali). And special thanks to David Mazieres, for his ever brilliant ideas.

IPFS是许多伟大的思想和系统的综合。如果不站在巨人的肩膀上实现如此远大的目标是不可能的。谢谢David Dalrymple，Joe Zimmerman，和Ali Yahya对这些想法进行长时间的讨论，特别是： 曝光大致的Merkle DAG（David, Joe），滚动哈希块（David)），和s/kademlia sybill保护（David，Ali）。特别感谢David Mazieres，他辉煌的想法。

翻译仅供帮助理解原文使用，有错误请指正

translated by zhangyanni

2017.7.25