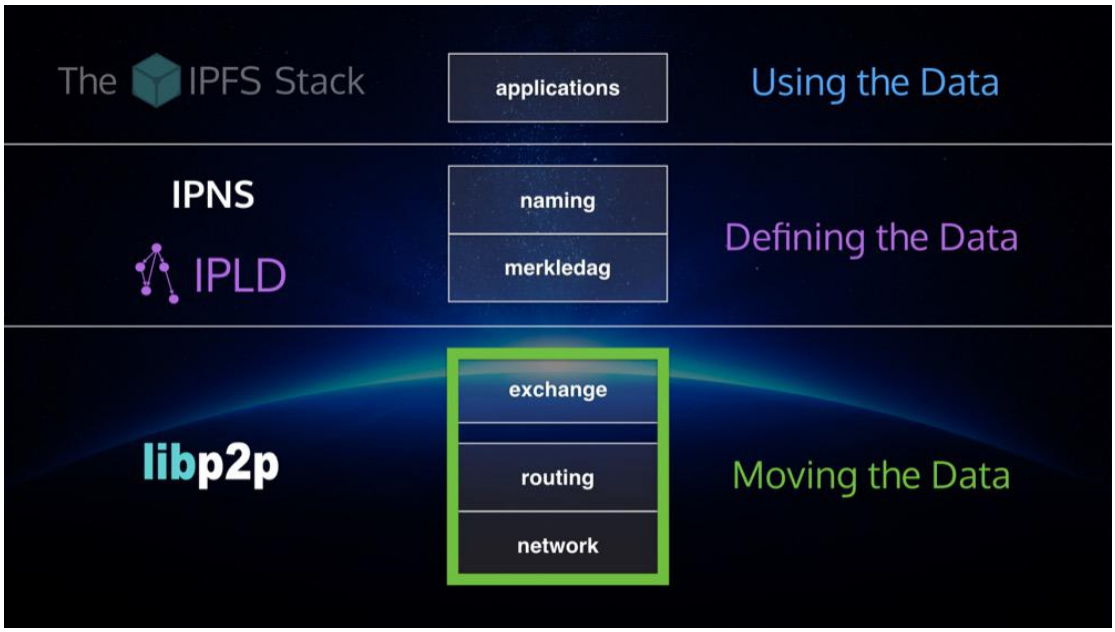
IPFS 中 IPLD 数据模型及存储

- IPFS 是基于内容可寻址的。IPFS 上的数据是使用 CID 来进行识别。
- 这些 CID 对于它引用的数据是具有惟一性的。
- IPFS 使用哈希函数作为其防篡改的属性，这使得 IPFS 成为了一个自认证的文件系统。
- IPFS 使用 Multihash，它允许对相同的数据使用不同版本的 CID(但是这并不意味 CID 不是唯一的)。如果我们使用相同的哈希函数，那么我们将会得到相同的 CID。
- IPFS 使用 IPLD(InterPlanetary Linked Data)来管理和链接所有的数据块。
- IPLD 使用 Merkle DAG(又称有向无环图)数据结构来链接数据块。
- IPLD 还向 IPFS 添加了重复数据删除特性。
- IPFS 使用 IPNS 将 CID 链接到固定的 IPNS 链接上，这种技术类似于今天的集中式 Internet 的 DNS。
- IPFS 使用 Libp2p 在 IPFS 网络上做数据通信并发现其他节点(计算机和智能手机)，这样可以显著的提高上网速度。



我们先看一下 IPFS 的系统架构图，分为 5 层：

第一层为 naming，基于 PKI 的一个命名空间，域名访问；

第二层为 merkledag，IPLD 定义的数据结构；

第三层为 exchange，节点之间 block data 的交换协议；

第四层为 routing，主要实现节点寻址和对象寻址；

第五层为 network，封装了 P2P 通讯的连接和传输部分。

站在数据的角度来看，又可以分为 2 个大的模块：

IPLD：主要用来定义数据， 给数据建模；

libp2p：解决的是数据如何传输的问题。

IPLD 是什么？

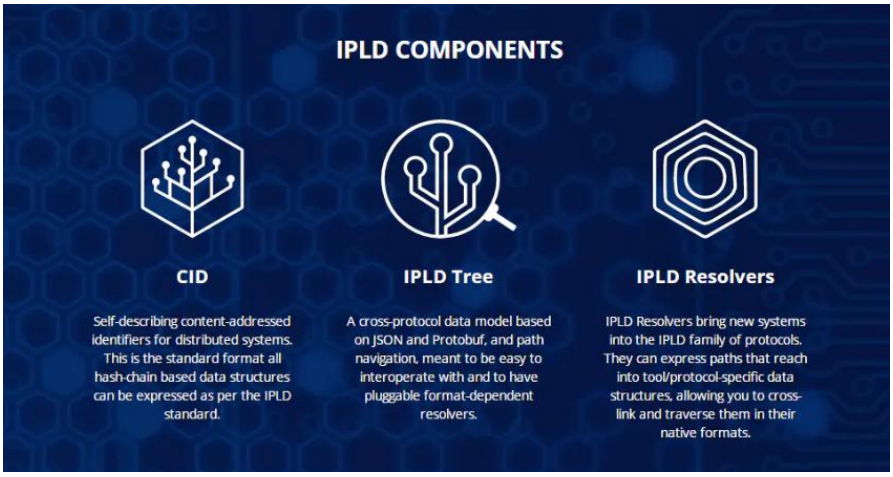
IPLD 是内容可寻址数据模型。它允许我们将所有散列连接的数据结构视为统一信息空间的子集，将所有将数据与散列连接的数据模型统一为 IPLD 实例。为不同领域之间的数据可互操作。

IPLD 组件包含：

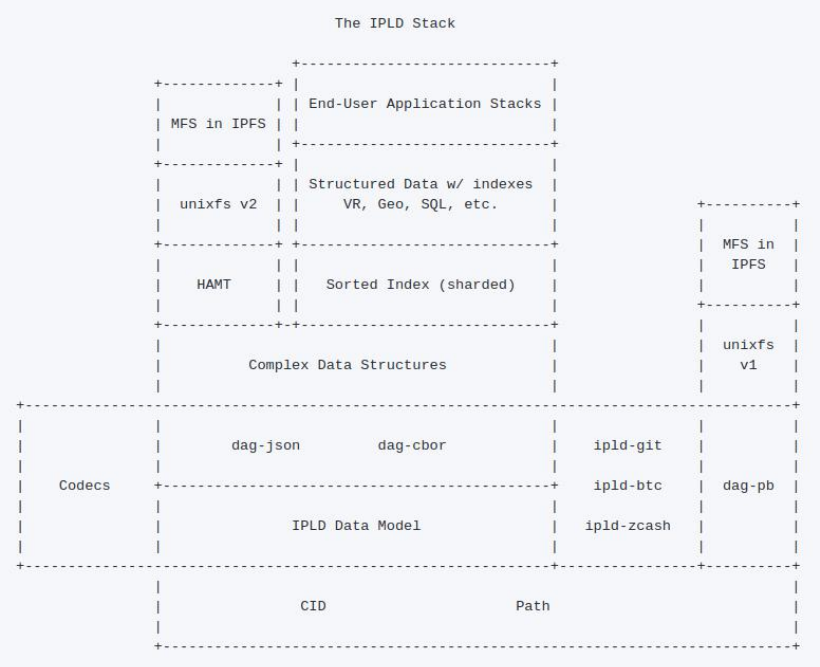
◆ CID：分布式系统的自描述内容寻址标识符。

◆ IPLD tree：基于 JSON、Protobuf 和路径导航的跨协议的数据模型，旨在易于与可插入格式相关的解析器进行互操作。

◆ IPLD Resolvers：IPLD 解析器可以将新系统引入到 IPLD 协议中。



IPLD 堆栈：该堆栈的目标是启用分散的数据结构，这又将可以应用更多分散的程序。



IPLD 的特点：

1. 规范数据模型

一个独立的描述模型，它可以标识任何基于哈希的数据结构，并确保相同的逻辑对象总是映射到完全相同的序列。

2. 协议独立解决方案

IPLD 将孤立的系统集成在一起(如连接比特币、以太坊和 git)，使与现有协议的集成变得简单

3. 可升级

有了 Multiformats 的支持，IPLD 将很容易升级，并且内容会根据你喜欢的协议而进行增加。

4. 跨格式操作

用各种可序列化的格式(如 JSON、CBOR、YAML、XML 等)表示 IPLD 对象，使 IPLD 可以轻松地与任何框架一起进行使用。

5. 向下兼容性

非侵入性的解析器使 IPLD 更加易于集成到现有的工作中。

6. 所有协议的命名空间

IPLD 允许你无缝地跨协议探索数据，通过公共的命名将基于哈希的数据结构绑定在一起

（一）Multiformat 基础组件介绍：

IPLD 旨在设计为一种通用性的数据结构，为了给不同的数据建模，我们需要一种通用的数据格式，通过它可以最大程度地兼容不同的数据，IPFS 中定义了一个抽象的集合，multiformat, 包含 multihash、multiaddr、multibase、multicodec、multistream 几个部分，同时也是

为了便于升级。

1、multihash

自识别 hash, 由 3 个部分组成，分别是：hash 函数编码、hash 值的长度和 hash 内容， 下面是个简单的例子：

这种设计的最大好处是非常方便升级，一旦有一天我们使用的 hash 函数不再安全了， 或者发现了更好的 hash 函数，我们可以很方便的升级系统。

2、multiaddr

自描述地址格式，可以描述各种不同的地址

```
/ip4/127.0.0.1/udp/9090/quic
/ip6/::1/tcp/3217
/ip4/127.0.0.1/tcp/90/http/baz.jpg
/dns4/foo.com/tcp/80/http/bar/baz.jpg
/dns6/foo.com/tcp/443/https
```

Human-readable multiaddr:

/ip4/127.0.0.1/udp/1234

Machine-readable multiaddr:

0x4 0x7f 0x0 0x0 0x1 0x91 0x2 0x4 0xd2

3、multibase

multibase 代表的是一种编码格式，方便把 CID 编码成不同的格式，

比如这里定义了 base2、base8，也有我们熟悉的 base58btc 和 base64 编码。

The current multibase table is here:

encoding	codes	name
identity	0x00	8-bit binary (encoder and decoder keeps data unmodified)
base1	1	unary tends to be 11111
base2	0	binary has 1 and 0
base8	7	highest char in octal
base10	9	highest char in decimal
base16	F, f	highest char in hex
base32	B, b	rfc4648 - no padding - highest letter
base32pad	C, c	rfc4648 - with padding
base32hex	V, v	rfc4648 - no padding - highest char
base32hexpad	T, t	rfc4648 - with padding
base32z	h	z-base-32 - used by Tahoe-LAFS - highest letter
base58flickr	Z	highest char
base58btc	z	highest char
base64	m	rfc4648 - no padding
base64pad	M	rfc4648 - with padding - MIME encoding
base64url	u	rfc4648 - no padding
base64urlpad	U	rfc4648 - with padding

```
4D756C7469626173652069732061765736F6D6521205C6F2F  # base16 (hex)
JV2WY5DJMJQXGZJANFZSAYLXMVZW63LFEEQFY3ZP            # base32
YAjKoNbau5KiqmHPmSxYCvn66dA1vLmwbt                   # base58
TXVsdGliYXNlIGlzIGF3ZXNvbWUhIFxvLw==                 # base64
```

利用 multibase 编码之后

```
F4D756C7469626173652069732061765736F6D6521205C6F2F # base16 F
BJV2WY5DJMJQXGZJANFZSAYLXMVZW63LFEEQFY3ZP           # base32 B
zYAjKoNbau5KiqmHPmSxYCvn66dA1vLmwbt                  # base58 z
MTXVsdGliYXNlIGlzIGF3ZXNvbWUhIFxvLw==               # base64 M
```

4、multicodec

mulcodec 代表的是自描述的编解码表，用 1 到 4 个字节定义了数据内容的格式，比如 0x50 表示 protobuf 等等。类似于在一组数据前面加入一个说明字符的前缀。即用较短前缀能够说明数据类型。也可以用在 KV 存储中，作为 key 的前缀。

```
<multicodec><encoded-data>
<mc><data>
```

下面是基于一个 javascript 的例子；先 new 一个 buffer 对象， 里面是 json 对象， 然后给它加一个前缀 protobuf, 这样这个 multicodec 就构造好了，可以通过网络传输。在解析时可以先取 codec 前缀，然后移除前缀，得到具体的数据内容

```
const buf = new Buffer(JSON.stringify({hello: 'world'}));
console.log(buf)
// <Buffer 7b 22 68 65 6c 6c 6f 22 3a 22 77 6f 72 6c 64 22 7d>

const prefixedBuf = multistream.addPrefix('protobuf', buf);
console.log(prefixedBuf);
// <Buffer 50 7b 22 68 65 6c 6c 6f 22 3a 22 77 6f 72 6c 64 22 7d>
console.log(prefixedBuf.toString('hex'));
// 507b2268656c6c6f223a22776f726c64227d
const codec = multistream.getCodec(prefixedBuf);
console.log(codec);
// protobuf
console.log(multistream.rmPrefix(prefixedBuf).toString());
// {"hello":"world"}
```

5、multistream-select

multistream-select 协议实现了一个简单的流路由器功能。

6、multigram

自描述分组网络协议（未完成）

7、multikey

自描述加密秘钥（未完成）

（二）IPLD 组成：

IPFS 使用 IPLD（IPLD 使用有向非循环图）来管理所有块并将其链接到基本 CID：

IPLD（对象）由 2 个组成部分组成：

```
type Node struct {
    Links []Link
    Data    string
}
```

data——大小<256 kB 的非结构化二进制数据块

Links——链接结构数组。这些是指向其他 IPFS 对象的链接

每个 IPLD 链接 Link 有 3 个部分：

```
type Link struct {
    Name, Hash string
    Size       uint64
}
```

Name——链接的名称

Hash——链接的 IPFS 对象的哈希值

Size——链接的 IPFS 对象的总大小，包括其子链接内容

如果文件大于 256 kB，则它们被分解为更小的部分，因此所有部分都等于或小于 256 kb，每个 Node 下面最多 link 174 个分支节点。

进行文件切割如将 chain33-cli 文件（53M）传入 IPFS 中时候生成的 root CID 为 QmdiwrDRzPZsFYRKYCCL7LVsgquPLezx4ULM1pSnXhiHtz：

```
ipfs object get QmdiwrDRzPZsFYRKYCCL7LVsgquPLezx4ULM1pSnXhiHtz
{
    "Links":[
    {
        "Name":"",
        "Hash":"QmUUdvoxa6RSw6m9Hj7ax1YdtAExAZv9Z2Rw2MXFoWRMXK",
        "Size":45623854
    },
    {
        "Name":"",
        "Hash":"QmXGP31aL3nP9tHpZiVP6v7bjKCHV8UgYH5FS37jmrXnD1",
        "Size":10388064
    }
    ]
    "Data": " "
}
```

即两个先有两个分支

CID QmdiwrDRzPZsFYRKYCCL7LVsgquPLezx4ULM1pSnXhiHtz
SIZE 53 MB
LINKS 2
DATA

▶ *Object {type: "file", data: undefined, blockSizes: Array[2]}*

	PATH	CID
0	Links/0	QmUUdvoxa6RSw6m9Hj7ax1YdtAExAZv9Z2Rw2MXFoWRMXK
1	Links/1	QmXGP31aL3nP9tHpZiVP6v7bjKCHV8UgYH5FS37jmrXnD1

Links0 下面有 174 个 links

CID QmUUdvoxa6RSw6m9Hj7ax1YdtAExAZv9Z2Rw2MXFoWRMXK
SIZE 44 MB
LINKS 174
DATA

▶ *Object {type: "file", data: undefined, blockSizes: Array[174]}*

	PATH	CID
0	Links/0	QmQPpSKhdswLn3x3f4VM5xzWW844Sq399soNAdrMjPEFK8
1	Links/1	QmaqL5HuikdRHatM1nKBRN5TDBFhvqrTrDFmm5zuLn16Jn
2	Links/2	QmQatPWbtTfPDnAHvXsvdbx1PHr6CEEudS54BGSPv8o76f
3	Links/3	QmWQ5krAcFnLVvztycrLRnRZh3xZcct1yBtqXvD8ryQvxV
4	Links/4	QmaYbMRvK7fWYn8AojUHCryohX988ppbgBQjVZ2ux4kjxL
5	Links/5	QmSjNEXm9XzdATWegm11Rg41NRXL6pLSyVfGtGHyaPoVU2
6	Links/6	QmTM7NxA6M213qiXRYy8bb94eeTDNF9YkSSE7zo5TVcxU1
7	Links/7	QmbqvZ8kt4PG2YJcPM47BecYAzjfuSMmLAR9kaWBQxD7wy
8	Links/8	QmfTFCWBdURGNpno7pkuXa5s8i2rqyAz8WH5PCrvvfDP5f
9	Links/9	QmTg7ddBw7Bm3GZw7ewG4qzanwnetM6LUk27VKfvsLE3oS
10	Links/10	QmZFa4iX94uRNdVz25tz2xKBGnAjWNftkFG5jYuQRq7Dkn
11	Links/11	QmeCYGALwbGrQsoajieo89qJ4pYtGH4D2TXus5tdi6y5Cw

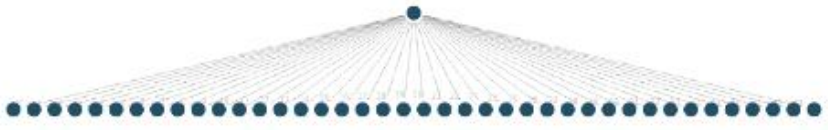
Links1 下面有 40 个 links

Protobuf UnixFS 在IPFS网关上查看

CID QmXGP31aL3nP9tHpZiVP6v7bjKCHV8UgYH5FS37jmrXnD1
SIZE 10 MB
LINKS 40
DATA

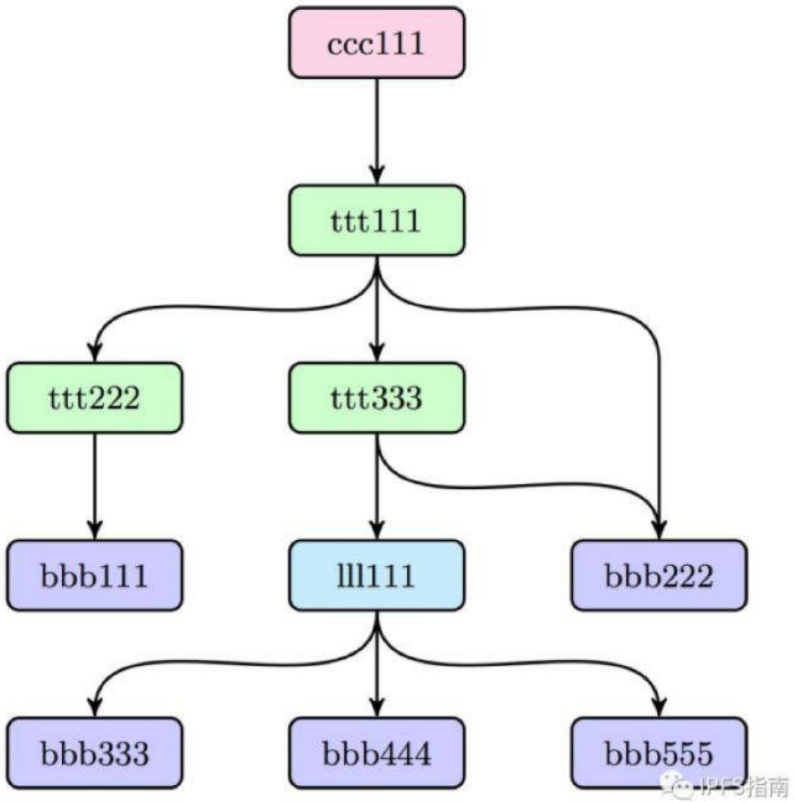
▸ Object {type: "file", data: undefined, blockSizes: Array[40]}

	PATH	CID
0	Links/0	Qmf1oafmeiAj5jvsgNA8QCThDo6mHEd6CDx2WCmimTAzkQ
1	Links/1	QmThd3kYxWJcdG3jaM1C97Bkt4esTkUU7rxoxdB1HQoeJ7
2	Links/2	QmRXGAwam2Gvt5BgJUz2aQAJ84gqcUhz3Ua2nFAZCa1uD5
3	Links/3	QmWuKFLBzQzi6C6qsbrCWVQLpY23JK9vxMKmc4BRxDweWe
4	Links/4	QmXVwDXodyy55W6N5UjabqXvXJaVp2dhz3qCsa3NJ1T19E
5	Links/5	QmebRhrfay5ivTLimdznvy9HPmE3S6S67jZ86xu52pa6e8
6	Links/6	QmNeJM7isYmTDxAsF4XxBZQ6kZ9VunAuFY5XkC5mEuKLeU
7	Links/7	QmdZZB3eqzxYtkBizk5t7fJmycC2JRuxj67P6cvAhbA2Qt
8	Links/8	QmcdLSBX6prigCo5kQ6eUY9S2x2EFmR34zePuuotDshQuB
9	Links/9	QmPVW6U52q6HpDWNSKsEAEgRmqigZJz5FPUutZ8hYkd7CZ
10	Links/10	QmYo8GTPksjahqaFgibD7q6nyP5EgdesGM5HC1KkRaNQTR
11	Links/11	QmTYHsGT7cCTQnK31w7WJ9fcrhxzEPautNXk3FWnAqr9Sv



（三）Merkledag

文件被切分成了多个 block，通过 merkledag 链接 block 数据。下面是 merkledag 的一个示意图：



简单来说，就是 2 种数据结构 merkle 和 DAG（有向无环图）的结合，通过这种逻辑结构，可以满足：

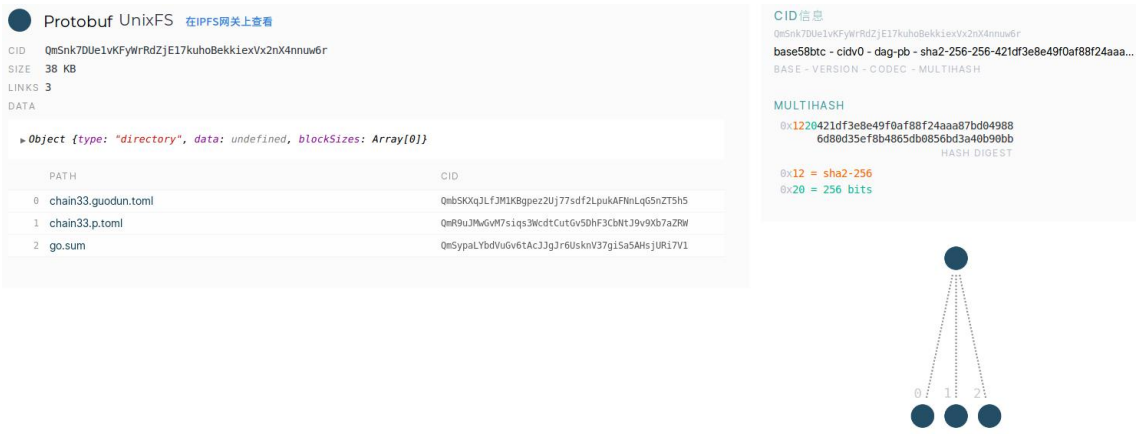
内容寻址:使用 hash ID 来唯一识别一个数据块的内容

防篡改:可以方便的检查哈希值来确认数据是否被篡改

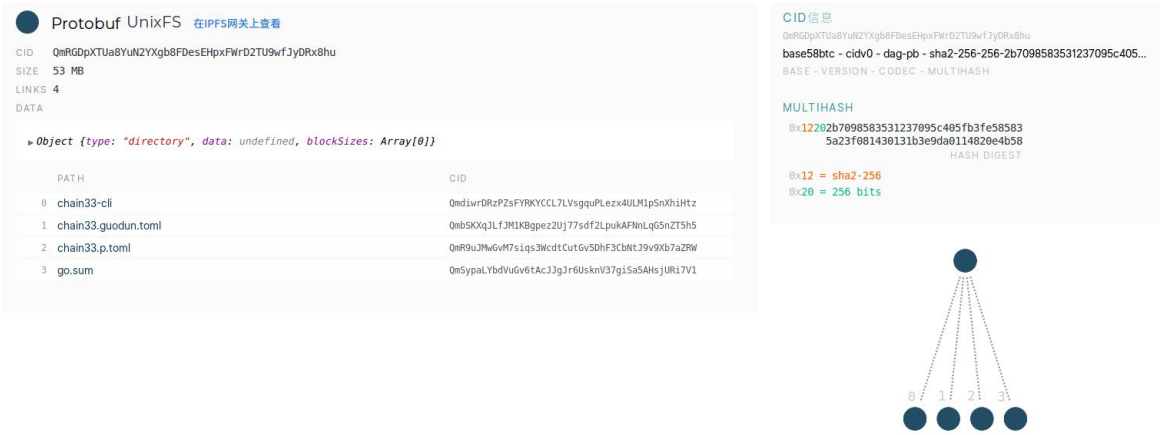
去重:由于内容相同的数据块哈希是相同的，可以很容去掉重复的数据，节省存储空间

Merkledag 可以类似与状态数据库一样带有版本，即不同的 roothash 下面是不一样的如：

如在 IPFS 系统某个文件夹下面存放三个文件 chain33.guodun.toml、chain33.p.toml、go.sum 则就会形成该文件夹的 rootCID 为: QmSnk7DUe1vKFyWrRdZjE17kuhoBekkiexVx2nX4nnuw6r merkledag 图右下图所示



在该 IPFS 文件夹中在加入 chain33-cli，则 rootCID 变为：QmRGDpXTUa8YuN2YXgb8FDesEHpxFWrD2TU9wfJyDRx8hu，其余链接文件 CID 不变，merkledag 图右下图所示



（四）CID(Content ID)

CID 是 IPFS 分布式文件系统中标准的文件寻址格式，它集合了内容寻址、加密散列算法和自我描述的格式，是 IPLD 内部核心的识别符。目前有 2 个版本，CIDv0 和 CIDv1。

CIDv0 格式: <multibase><version><multicodec><multihash>

multibase 即为 base58btc

multicodec 即为 protobuf-mdag

version 即为 CIDv0

multihash 即为 sha256

举例：

QmSnuWmxptJZdLJpKRarxBMS2Ju2oANVrgbr2xWbie9b2D

base58btc - cidv0 - dag-pb - sha2-256-256-422896a1ce82a7b1cc0ba27c7d8de2886c7df95588473d5e88a28a9fcfa0e43e

multihash 0x1220422896a1ce82a7b1cc0ba27c7d8de2886c7df95588473d5e88a28a9fcfa0e43e

Hash digest

0x12 = sha2-256

0x20 = 256 bits

CIDv1 格式: <multibase-prefix><cid-version><multicodec><multihash>

举例：

```
# example CID

zb2rhe5P4gXftAwvA4eXQ5HJwsER2owDyS9sKaQRRVQPn93bA

# corresponding human readable CID

base58btc - cidv1 - raw -
sha2-256-256-6e6ff7950a36187a801613426e858dce686cd7d7e3c0fc42ee0330072d245c95
```

（五）ETH 数据基于 IPLD 定义

当前以太坊中区块以及 MPT 树节点可以加入该中格式存储于 IPFS 中，定义结构如下：

```
// EthBlock (eth-block, codec 0x90), represents an ethereum block header
type EthBlock struct {
	*types.Header
	cid	*cid.Cid
	rawdata []byte
}
type EthTx struct {
	*types.Transaction
	cid	*cid.Cid
	rawdata []byte
}
// TrieNode is the general abstraction for
//ethereum IPLD trie nodes.
type TrieNode struct {
	// leaf, extension or branch
	nodeKind string
	// If leaf or extension: [0] is key, [1] is val.
	// If branch: [0] - [16] are children.
	elements []interface{}
	// IPLD block information
	cid	*cid.Cid
	rawdata []byte
}
```

上述定义需要实现 IPFS 中的 Node 接口，即可以利用 ipld 相对应的库生成相应的存储 CID 对区块相应内容存储到 IPFS

```
type Node interface {
	blocks.Block
	Resolver
	...
```

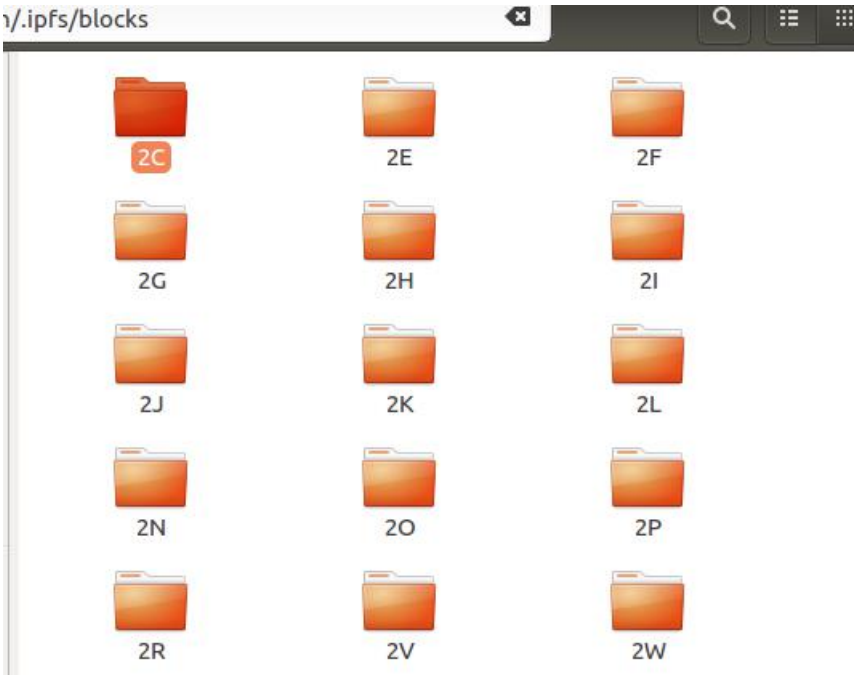
}
解析出的格式如下：


```
{
    "bloom": "0x000000000.....0000000",
    "coinbase": "0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01",
    "difficulty": 11966502474733,
    "extra": "0xd783010400844765746887676f312e352e31856c696e7578",
    "gaslimit": 3141592,
    "gasused": 21000,
    "mixdigest":
"0x2565992ba4dbd7ab3bb08d1da34051ae1d90c79bc637a21aa2f51f6380bf5f6a",
    "nonce": "0xf7a14147c2320b2d",
    "number": 997522,
    "parent": {
        "/": "z43AaGF24mjRxbn7A13gec2PjF5XZ1WXXCyhKCyxzYVBcxp3JuG"
    },
    "receipts": {
        "/": "z44vkPhjt2DpRokuesTzi6BKDriQKFEwe4Pvm6HLAK3YWiHDzrR"
    },
    "root": {
        "/": "z45oqTRunK259j6Te1e3FsB27RJfDJop4XgbAbY39rwLmfoVWX4"
    },
    "time": 1455362245,
    "tx": {
        "/": "z443fKyLvyDQBBQRGMNnPb8oPhPerbdwUX2QsQCUKqte1hy4kwD"
    },
    "uncles": {
        "/": "z43c7o73GVAMgEbpaNnaruD3ZbF4T2bqHZgFfyWqCejibzvJk41"
    }
}
```

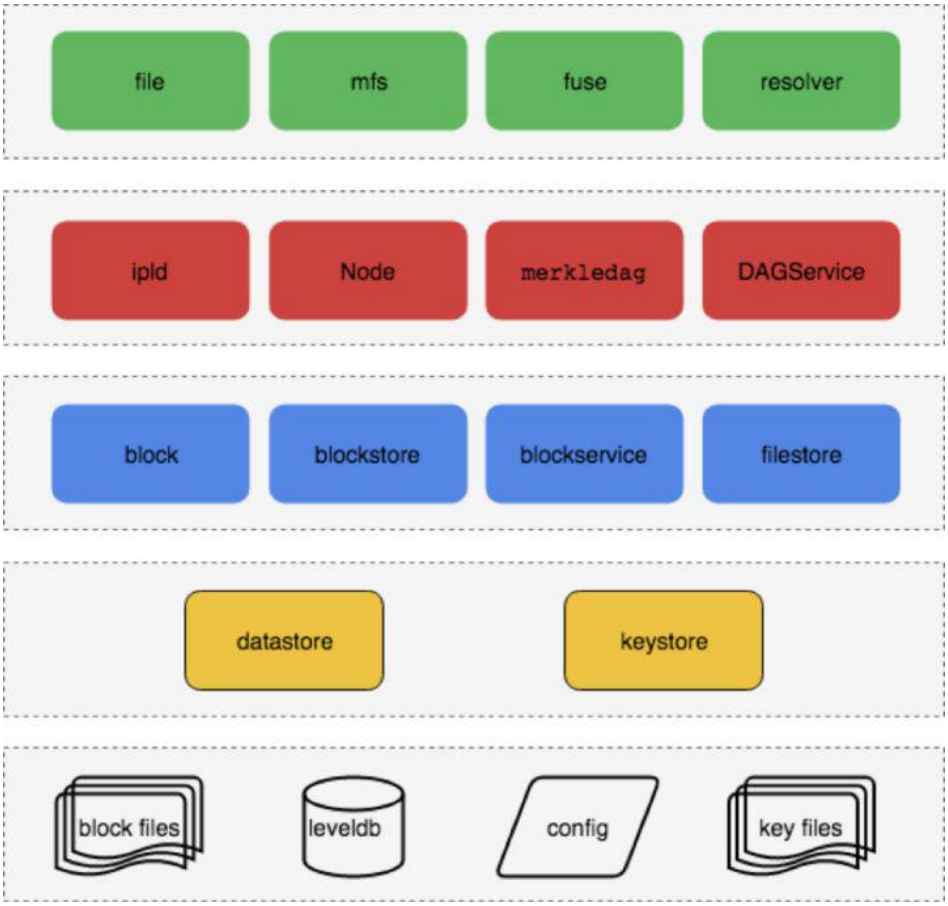

（六）IPFS 本地存储目录

```
.ipfs/
├── api            <--- running daemon api addr
├── blocks/        <--- objects stored directly on disk
│   └── aa         <--- prefix namespacing like git
│       └── aa     <--- N tiers
├── config         <--- config file (json or toml)
├── hooks/         <--- hook scripts
├── keys/          <--- cryptographic keys
│   ├── id.pri     <--- identity private key
│   └── id.pub     <--- identity public key
├── datastore/     <--- datastore
├── logs/          <--- 1 or more files (log rotate)
│   └── events.log <--- can be tailed
├── repo.lock      <--- mutex for repo
└── version        <--- version file
```

blocks：实际存储被拆分的文件数据。



datastore：默认以 leveldb 作为存储后端，其中的 key 为 CID，value 为 IPFS Node，其中 data 字段数据通过 blocks 文件夹下数据进行映射。存储逻部分辑图，我们可以看到上面是一些高级的接口，比如 file, mfs 等。 下面是数据结构的持久化部分，节点之间交换的内容是以 block 为基础的。



（七）相关软件包以及调用流程

ipfs/go-ipld-format

IPLD Node 格式接口，利用 IPLD 存储需要实现内部接口，上面 ETH 存储中都需要实现这些接口。

ipfs/go-cid

CID 的定义以及实现

ipfs/go-unixfs

在 ipld merkledag 之上实现类 unix 的文件系统实用程序，其中涉及到两种文件 DAG 文件分割方案 balanced 分割和 trickle 分割;从本地上传一个文件到 IPFS 首先需要多文件进行分割然后生成 IPFS Node。

ipfs/go-mfs

实现内存模型 IPFS 文件系统。

ipfs/go-blockservice

以 block 为单位为本地和远程存储后端提供了存储接口

ipfs/go-ipfs-blockstore

底层数据存储接口，即需要有 leveldb 或者是内存型数据结构实现这些接口。

以 Add 命令为例，需要上传一个文件到 IPFS 系统中：

（1）core/commands/add.go 通过调用 api.Unixfs().Add(req.Context, addit.Node(), opts...)函数进行上传；

（2）进入 core/coreapi/unixfs.go 下的 Add()函数进行 DAG 运算以及数据库中的存储以及向对端节点进行上传数据，所有的这些数据都会切分成每个 block，然后根据 DAG 组成 Node 进行处理。即通过 NewDAGService 进行处理。

（3）从（2）分割的数据通过 ipfs/go-mfs/ops.go 中的 PutNode()函数将文件实际数据以及关系数据进行保存，实际数据保存到文件系统中，关系保存到 leveldb 中（通过如下两个函数 d.dagService.Add()与 d.addUnixFSChild()）。

（4）进行底层存储 blockService 主要利用中间层的 blockstore 抽象接口调用 ipfs/go-ipfs-blockstore/blockstore.go 中的 PutMany 保存到 leveldb 中（注：blockstore 抽象出一个存储层的接口（具体实现对象可以是 leveldb,或者内存等），blockstore 以 CID 为 key 以其数据为 value 存入 datastore）

（八）IPLD 中 dag 操作

ipfs 中添加了一些关于 dag 命令用于创建 dag 节点:

```
lyh@lyh-virtual-machine:~$ ipfs dag
USAGE
  ipfs dag - Interact with ipld dag objects.

  ipfs dag

  'ipfs dag' is used for creating and manipulating dag objects.

  This subcommand is currently an experimental feature, but it is i
  to deprecate and replace the existing 'ipfs object' command movin


SUBCOMMANDS
  ipfs dag get <ref>             - Get a dag node from ipfs.
  ipfs dag put <object data>... - Add a dag node to ipfs.
  ipfs dag resolve <ref>         - Resolve ipld block

  For more information about each command, use:
  'ipfs dag <subcmd> --help'

lyh@lyh-virtual-machine:~$
```

下图为 js-ipfs 中的用例

```
'use strict'

const createNode = require('./create-node')

async function main () {
  const ipfs = await createNode()

  console.log('\nStart of the example:')

  const myData = {
    name: 'David',
    likes: ['js-ipfs', 'icecream', 'steak']
  }

  const cid = await ipfs.dag.put(myData, { format: 'dag-cbor', hashAlg: 'sha2-256' })
  console.log(cid.toString())
  // should print:
  //   bafyreigsccjrxlioppkkzv27se4gxh2aygbxfnsobkaxxqiuni544uk66a
}

main()
```

参考

[IPFS SPECS]