IPFS IN TYPESCRIPT

Helia

去中心化
模块化
高性能

为分布式网络
打造的全新实现

目录

01 js-IPFS 的演进

了解 Helia 的起源、与先前 IPFS 实现的关系，以及创建全新模块化方法的战略决策

02 设计理念

模块化架构、嵌入式节点方法，以及让 Helia 区别于基于守护进程解决方案的开发者体验原则

03 互操作性与性能

通过全面测试套件实现的兼容性保证，以及显著的性能改进基准测试

04 架构与功能

路由机制、传输协议、识别方法，以及全面的 libp2p 集成

01 | 第一章

js-IPFS 的演进

了解 Helia 的起源及其与先前 IPFS 实现的关系

历史背景

Helia 与 js-IPFS 的关系是什么?

历史背景

IPFS 是一套用于数据寻址和传输的规范和工具。历史上，它已在 Go (Kubo) 和 JavaScript (js-IPFS) 中实现。

随着 Filecoin 的开发，Protocol Labs 发现当实现名称不包含"IPFS"时，"一个协议，多种实现"的进展更为顺利，以避免对某个项目的无意识偏见。

重命名策略

2021 年底，实施了从实现名称中移除"IPFS"的计划：

go-ipfs → Kubo （照常继续）

js-ipfs → Helia （全新设计）

JavaScript 的机遇

当 Kubo 保持不变时，js-IPFS 提供了一个机会，可以**重新构想 JavaScript 中 IPFS 实现的样子**，并应用多年的开发经验。

- ✓ 更模块化、更轻量级的设计
- ✓ 专注于最重要的 JS 用例
- ✓ 为下一代 dApp 重新设计 API

共享组件

Helia 与 js-IPFS 共享核心组件，同时引入重新设计的 API：

- libp2p
- bitswap
- 重新设计的 API

js-IPFS **正在被弃用**，转而采用 Helia。阅读更多关于 Helia 设计的思考过程，并查看示例以开始移植您的应用程序。

对比分析

为何选择 Helia 而非 js-IPFS?

全新设计

Helia 代表了对 JavaScript 中全功能 IPFS 实现的全新审视，不受历史约束。

- ✓ 从头开始重新设计
- ✓ 现代 JavaScript 模式
- ✓ 针对当前生态系统优化

js-IPFS 遗留问题

从一开始，js-IPFS 就被设计为 go-IPFS（现为 Kubo）的 API 兼容克隆。

- ✗ 难以跟上 Kubo 的步伐
- ✗ 并非所有功能都适用于 JS
- ✗ 厨房水槽式方法

模块化自由

不受 Kubo API 的约束，Helia 可以自由创新，赋能 JavaScript 用例。

- ✓ 无需遵循厨房水槽式方法
- ✓ 只导入你需要的功能
- ✓ 与 Kubo 网络兼容

js-IPFS 已弃用

js-IPFS 将不再接收更新，仅提供紧急安全修复。所有开发重点已转移到 Helia。

→ 提供迁移指南以实现无缝过渡

02 | 第二章

设计理念

Helia 的模块化架构和
开发者体验原则

架构对比

为何选择 Helia 而非 kubo-rpc-client?

Helia 和 kubo-rpc-client 是两种截然不同的架构。 它们之间的选择会影响性能、部署复杂性、监控能力和架构灵活性。

kubo-rpc-client

基于守护进程的架构

- ⚠ 需要 Kubo 守护进程
  独立的 Go 运行时进程
- ⚠ 监控不透明
  无法使用熟悉的 JS 工具
- ⚠ HTTP RPC API 安全性
  内置保护极少
- ⚠ 延迟增加
  网络通信开销

Helia

嵌入式节点架构

- ✓ 嵌入式 IPFS 节点
  直接集成，最低延迟
- ✓ JS 监控与检查
  使用现有技能
- ✓ 减少活动部件
  简化的部署流程
- ✓ 浏览器兼容性
  每个用户一个完整的 IPFS 节点

关键洞察

运行 Helia 允许开发者使用他们的 JavaScript 专业知识来调优和监控他们的节点，使用与检查应用程序相同的技能和工具。无需在语言或调试工具之间切换上下文。

设计原则

模块化架构：按需选择

Helia 采用模块化设计理念，与先前实现的单体方法形成鲜明对比。这种模块化使开发者能够构建更精简、更专注的应用程序。

单体方法

js-IPFS / Kubo 模型

所有功能捆绑在一起，无论是否需要。应用程序承受着未使用功能的重量。

- ✕ 捆绑包体积大
- ✕ 不需要的依赖项
- ✕ 定制受限
- ✕ 资源占用高

模块化方法

Helia 模型

只导入应用程序需要的功能。构建针对您的用例定制的 IPFS 解决方案。

- ✓ 最小捆绑包体积
- ✓ 选择性依赖项
- ✓ 高度可定制
- ✓ 优化资源使用

Helia 核心包

- @helia/helia （核心）
- @helia/ipns （命名）
- @helia/unixfs （文件）
- @helia/remote （固定）

对应用程序大小的影响

最小 Helia 设置	~500KB
典型 js-IPFS 捆绑包	~15MB+

基本使用情况下小 30 倍

03

第三章

互操作性与性能

兼容性保证和
性能基准测试

互操作性

Helia 如何保证兼容性?

互操作性测试套件

每个 Helia 组件都有一个互操作性测试套件，用于测试与其他 IPFS 实现的兼容性。这些测试在每个 PR 期间作为 CI 的一部分运行，并在发布前运行。

- ✓ 每个 PR 自动进行 CI 测试
- ✓ 发布前兼容性验证
- ✓ 持续扩展测试套件

开放测试矩阵

最初测试与 Kubo 的兼容性，但向其他 IPFS 实现开放：

如果您维护另一个 IPFS 实现：

提交 issue 或 PR 以加入测试矩阵

组件互操作性套件

Helia 为增强模块化维护单独的互操作性测试套件：

helia 互操作性
核心 Helia 功能测试

@helia/ipns 互操作性
IPNS 命名系统兼容性

@helia/unixfs 互操作性
UnixFS 文件操作

注意：Helia 不依赖于 https://github.com/ipfs/interop。该仓库由 Kubo 用于与其他实现的互操作性测试。

基准测试

性能：基准测试结果

在已进行基准测试的领域，Helia 表现非常优异。 本仓库中的基准测试套件正在持续扩展，以覆盖大多数功能领域。

传输速度

通过优化的协议和减少的开销，数据传输性能显著提升。

~40% 更快

相比 js-IPFS 基准测试

内存使用

通过更优的架构实现更高效的内存分配和减少的垃圾回收压力。

~60% 更少

内存消耗

启动时间

通过延迟加载和优化的依赖关系图，初始化和节点启动更快。

~3 倍更快

冷启动初始化

垃圾回收：引用计数

Helia 使用引用计数进行垃圾回收，这已被证明比 js-IPFS 或 Kubo 采用的方法更具可扩展性。

- 数据规模线性性能扩展
- 常数时间 GC 操作

基准测试资源

- 仓库中的基准测试套件
- 数据传输基准测试问题
- 带图表的 GC 性能分析
- 2023 年 IPFS 大会 Helia 性能演讲

04

第四章

架构与功能

技术实现细节
和支持的协议

路由机制

内容和节点路由机制

Helia 支持所有现有的 libp2p 内容和节点路由机制。 这种全面的支持确保 Helia 节点可以使用多种策略在 IPFS 网络中发现内容和节点。

KAD-DHT

分布式哈希表 实现，用于跨 IPFS 网络的去中心化节点和内容路由。

- ✓ 分布式路由
- ✓ Coral DHT 优化
- ✓ 女巫攻击抵抗

@libp2p/kad-dht

IPNI 内容路由

IPFS 网络索引器 集成，用于通过索引器加速内容发现和路由。

- ✓ 快速内容发现
- ✓ 索引器集成
- ✓ 可扩展路由

@libp2p/ipni-content-routing

Reframe 路由

Reframe 协议 支持，用于委托路由，具有可定制的路由策略和提供者。

- ✓ 委托路由
- ✓ 自定义路由提供者
- ✓ 协议灵活性

@libp2p/reframe-content-routing

全面的路由策略

Helia 的路由架构结合了多种机制以实现最大的兼容性和性能。节点可以查询 DHT 进行去中心化路由，使用 IPNI 进行快速索引内容，或通过 Reframe 委托给特定路由提供者。这种多管齐下的方法确保了整个 IPFS 生态系统中的内容和节点可发现性。

传输协议

数据传输：连接选项

Helia 支持所有现有的 libp2p 传输（参见 libp2p 传输实现），为不同环境和用例提供全面的连接选项。

WebTransport

现代浏览器传输，使用 HTTP/3 实现低延迟、多路复用连接。绕过传统 WebSocket 限制。

- ✓ 基于 HTTP/3
- ✓ 浏览器原生
- ✓ 低延迟

@libp2p/webtransport

WebRTC

浏览器点对点连接，具有 NAT 穿透功能。支持直接的浏览器间通信。

- ✓ P2P 浏览器连接
- ✓ NAT 穿透
- ✓ STUN/TURN 支持

@libp2p/webrtc

WebSockets

服务器端连接，用于 Node.js。通过标准 Web 基础设施实现双向通信。

- ✓ Node.js 环境
- ✓ 服务器应用程序
- ✓ 标准 Web 端口

@libp2p/websockets

TCP

传统网络，用于 Node.js。原始 TCP 连接，实现最大性能和灵活性。

- ✓ 原始 TCP 性能
- ✓ Node.js 服务器应用
- ✓ 自定义协议

@libp2p/tcp

传输灵活性

Helia 可以同时使用多种传输方式。浏览器可能同时使用 WebTransport 和 WebRTC，而服务器使用 TCP 和 WebSockets。这确保了**最大的可达性**跨越网络环境。

识别协议

Helia 如何"识别"自身

代理版本格式

除非手动指定，Helia 在 libp2p 识别 期间使用以下字符串：

```
helia/x.x.x libp2p/x.x.x UserAgent=$USER_AGENT
```

自定义代理版本

开发者可以通过配置 libp2p 实例来覆盖默认代理字符串：

```
{ agentVersion: "my-app/v1.0.0" }
```

$USER_AGENT 示例

Node.js 环境：

v18.16.0

浏览器环境：

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36

实现参考

参见 Helia 源代码中的 addHeliaToAgentVersion 函数了解实现细节：

packages/helia/src/index.ts#L144

未来是去中心化的

JavaScript 中 IPFS 的未来

Helia 代表了 JavaScript 生态系统中 IPFS 的新篇章——
模块化、高性能，为去中心化未来而构建。

从今天开始使用 Helia 构建。


ipfs/helia


docs.helia.io


示例