# Libp2p 简介

欢迎来阅读libp2p相关文档，不论你是刚开始学习如何用libp2p来搭建P2P系统， 还是想潜心研究P2P相关概念和解决方案，亦或者是查阅相关的详细的参考信息，这里都是个很好的开始。

# 综述

libp2p是什么？（ [What is libp2p?](https://docs.libp2p.io/introduction/what-is-libp2p/)） 这个章节介绍了libp2p的基本信息和目前已公布的问题的概要信息

# 教程

如果你想进一步研究libp2p，你可以查阅我们的教程（ [tutorials](https://docs.libp2p.io/tutorials/)）， 它可以成为你学习libp2p的指南。

# 实例

如果你是想知道libp2p是什么， 或者只是想看它如何使用，可以查阅实例（[examples](https://docs.libp2p.io/examples/)）。每一个libp2p的实现都是一组可工作的实例项目， 用于展现关键概念和用例。

# 参考

## 规范和规划

虽然libp2p有多种不同的实现，但是它基本上是一组用于对等点标识、发现、路由、传输等的协议。

详细信息可以阅读规范说明章节（ [specifications section](https://docs.libp2p.io/reference/specs/)）

## 实现

Libp2p的核心是一组规范集合（ [specifications](https://docs.libp2p.io/reference/specs/)）， 这是规范组合在一起从抽象概念上定义了什么是libp2p 和怎么做才是libp2p的正确实现。现今libp2p有着几种不同编程语言的实现，并且每个都有着不同的完成度。由Go和JavaScript编写的完成度最高，另外用rust语言编写的也正在快速成熟。

除了上面已经提到的，libp2p社区也正在积极的用python来实现libp2p和用Kotlin实现JVM。请通过查看项目主页来获得每个项目实现的状态和完成状态。

# 社区

在 [https://discuss.ipfs.io](https://discuss.ipfs.io/)网站上，你可以联系在libp2p社区使用libp2p编写工具和应用程序的成员，可以向他们提相关问题，讨论新的想法或就某些问题获得支持等，并且你也可以到IRC（[hop on IRC](https://docs.libp2p.io/community/irc/)）进行快速交谈。

在社区章节可以看到更多的链接，通过这些链接可以查看更多的关于会议，事件，apps等的更多信息。

关于在这社区中对libp2p和其他软件项目做出贡献的信息也都可以在这查看到。

# 加入我们

Libp2p是一个社区开源项目。虽然协议实验室（ [Protocol Labs](https://protocol.ai/)）能够赞助一些与其相关的工作，但是大部分工作，包括设计，编码，都是由像你这样的社区成员和志愿者来完成的。如果你有兴趣来帮助完善libp2p，可以先从阅读贡献指南（[contributing](https://docs.libp2p.io/contributing/) ）开始。

如果你想编写新的代码，请先阅读贡献指南（[contribution guidelines](https://github.com/libp2p/community/blob/master/CONTRIBUTE.md)）和代码风格说明书。

# 关联项目

Libp2p在开始的时候是IPFS项目的一部分， 也是IPFS项目的基本组件。因此，libp2p能够与IPFS家族的项目的抽象概念和工具完美的结合。你可以去查看他们各自的网站来获得更多信息。

[IPFS](https://libp2p.io/) 是星际文件系统， 是使用libp2p作为它的网络层。

[Multiformats](https://multiformats.io/)是各种自描述数据格式系统

[IPLD](https://ipld.io/) 是一系列用于描述地址-内容链接的工具，比如IPFS 文件， Git commits, or Ethereum blocks.

[The Permissive License Stack](https://protocol.ai/blog/announcing-the-permissive-license-stack) 是一个为软件开发提供特许策略的工具，这个非常适合开源项目。

# Llbp2p是什么？

这是个很好的问题。用一句话来概况就是libp2p是一个模块化的协议系统，它的规范和程序库可以用来开发p2p网络应用程序。

# 对等节点基础

对于我们对libp2p在上面的概要描述有很多内容需要进一步解释，我们先从最后提到的一点开始，“p2p网络应用程序”。你现在正在这里阅读这篇文章，很可能你正在进行p2p系统的开发，而且正需要寻求帮助，亦或者这是你第一次来探索p2p网络的世界。不论是那种情况，我们都应该先花一分钟时间来定义一下我们的术语，我们有一些术语你可以前往[shared vocabulary](https://docs.libp2p.io/reference/glossary/)进行查看。

P2p网络是一种所有参与者能够或多或少的在平等地位的基础上进行直接通讯的网络。这并不是意味着所有节点都是相同的，他们在整个网络中可能有着不同的角色。可是，p2p网络定义的一个重要特征是没有servers 和clients，severs 和 clinets 是[client / server model](https://docs.libp2p.io/reference/glossary/" \l "client-server)的特征。

因为p2p网络的定义非常的广泛，现在许多已经建立的系统都属于p2p的范畴。最突出的例子就是文件共享系统bittorrent， 还有更多的最近激增的区款链网络系统。

# Libp2p能解决什么问题？

虽然P2p网络比client/server 模型有很多优点，但是它仍有很多挑战需要我们认真思考和在实践中进行克服。我们在建立IPFS的时候，我们是通过模块化的解决方案来处理这些挑战的，这样说为啥现在有了libp2p。现在libp2p已经从IPFS独立出来了， 它不再需要或依赖IPFS，现今有很多项目都是用libp2p作为他们的网络传输层。我们可以一起利用我们现在积累的经验解决这些基础性问题，这对整个开发者生态系统和整个世界上的用户都有益处。这里我将会简单罗列出libp2p所宣称的主要问题领域（2019年之前）。这些范围一直在增长，所以如果随着时间的推移他们有所变化，你也不要感到吃惊。我们将尽最大努力保持这些内容是最新的，但是如果你发现有些内容缺失或者有其他想法可以改善这些文档，请联系我们，让我们知道（[reach out to let us know](https://github.com/libp2p/docs/issues)）。

## 传输

libp2p的基础是传输层，它是负责与其他节点进行传送和接收数据。现今有很多方法通过网络进行数据传输，甚至包括现在正在开发和设计的方式。Libp2p提供了简单的接口用于适应现存的或未来的协议，这使得libp2p应用程序可以运行在许多不同的网络环境中。

## 身份

在整个世界有数十亿的设备进行着网络连接，知道你正在跟谁说话是进行安全可靠通讯的关键。Libp2p使用公钥密码学（[public key cryptography](https://en.wikipedia.org/wiki/Public-key_cryptography)）作为节点身份认证的基础，公钥密码学可以提供两个基本特征。第一，它给全球每一个节点指定一个唯一的名字，也就是节点ID。第二，节点ID允许任何人通过检索公钥进行节点身份认证，这使得节点之间能够安全的通信。

## 安全

最基本的需求是我们能够在节点之间安全的发送和接收数据，那就意味着我们能够相信我们正在通信的节点的身份，并且没有第三方能够读到我们谈话信息和修改他们。

Libp2p支持传输层模块提供的通过升级通讯连接使其变成安全加密通道连接。这个过程非常灵活，还可以支持多种加密通讯方法。当前默认的是 [secio](https://docs.libp2p.io/reference/glossary/" \l "secio)，对 [TLS 1.3](https://www.ietf.org/blog/tls13/) 的支持正在开发。

## 节点路由

当你想发送消息到其他节点，你需要两个关键信息：他们的节点ID和一种在网络上定位他们的方法以至于能够与他们建立连接。

我们的很多例子只有我们想要连接节点的ID，我们还需要一个用来发现他们的网络地址的方法。节点路由是利用其它节点的信息来发现节点地址的过程。

在一个节点路由系统中，一个节点如果知道我们所需节点的地址，它会发送给我们，否则它会把我们的查询信息发送给它认为最有可能知道答案的其它节点。随着我们联系越来越多的节点，我们不仅增加了我们查找所需节点的机会，而且更完善了我们的路由表视图，这使得我们能够回答来自其它节点的查询。

目前libp2p中稳定的对等节点路由实现使用一个分布式哈希表，使用Kademlia路由算法迭代路由请求，使其逐渐接近所期望的对等节点ID。

## 内容发现

在一些系统中，相比于知道正在跟谁通讯，我们更关心他们能够向我们提供什么。例如，我们想获得一些特定数据，但是我们并不关心我们是从哪里获得的因为我们能够验证这些数据的完整性。

Libp2p基于这个目的提供了内容路由接口（ [content routing interface](https://github.com/libp2p/interface-content-routing)），现在稳定版本的实现使用的也是与节点路由相似的基于 [Kademlia](https://en.wikipedia.org/wiki/Kademlia) 的DHT算法。

## 消息、发布与订阅

发送消息到其他节点是P2P系统的核心， 发布和订阅是一种向特定人群发送他们所感兴趣的消息的一种非常有用的模式。

Libp2p定义了发布与订阅接口，用于向所有订阅了特定主题的节点发送相应的消息。当前稳定的接口实现有两种：floodsub使用起来非常简单，但是效率不高，它使用的是network flooding策略。 [gossipsub](https://github.com/libp2p/specs/tree/master/pubsub/gossipsub)定义了一个可扩展的gossip协议。[episub](https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/episub.md)目前也正在积极的开发中，它是对gossipsub进行了扩展，这是为単源组播和少数固定源向大量客户端广播进行优化的。

# 教程

## 入门指南

该教程讲述了设置开发环境，熟悉libp2p的一些基本知识，实现一个超级简单的可以发送和接收ping消息的节点。

# 入门指南

## 从go-libp2p开始

这里是从一系列关于libp2p的go实现教程开始，*[go-libp2p](https://github.com/libp2p/go-libp2p)*

我们会讲述go的安装，go模块的设置，启动libp2p节点，并在它们之间发送消息。

## 从js-libp2p开始

这里是从一系列关于libp2p的javascirpt实现教程开始，*[js-libp2p](https://github.com/libp2p/js-libp2p)*

我们将带你完成一个具有基本功能的全功能libp2p节点的设置。最后我们还会在两个节点之间发送Ping消息。

# 从go-libp2p开始

这里是从一系列关于libp2p的go实现教程开始，*[go-libp2p](https://github.com/libp2p/go-libp2p)*

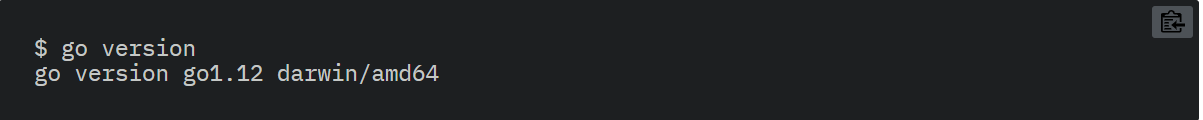
我们会讲述go的安装，go模块的设置，启动libp2p节点，并在它们之间发送消息。

## 安装go

go-libp2p推荐使用包含 [modules feature](https://github.com/golang/go/wiki/Modules)的go版本，也就意味着你必须使用1.11或以上版本。

你可以按照 [official installation instructions](https://golang.org/doc/install)安装go的最新版本。

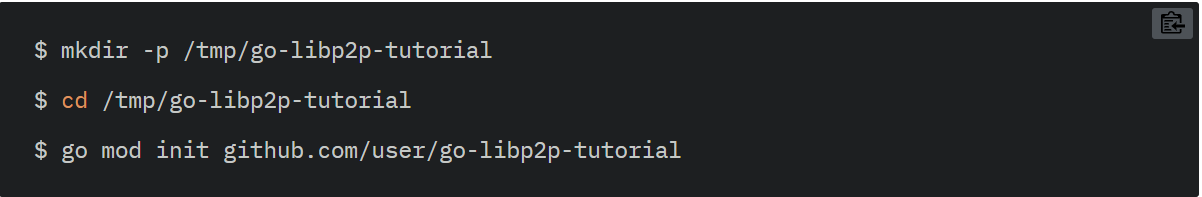
安装成功后，你应该能够运行go version 并且能够看到版本>= 1.11，例如:



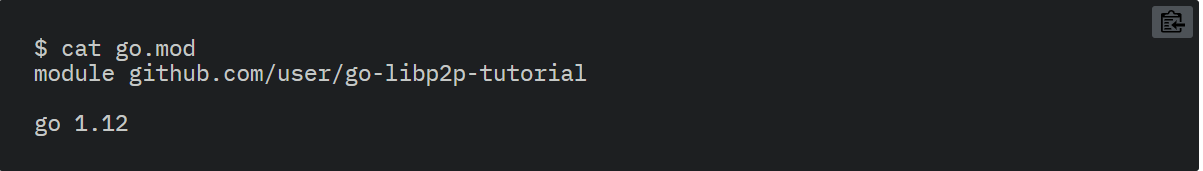
## 创建go模块

我们将要创建一个可以在命令中运行的go模块。

我们首先创建一个目录，然后再用go mod 初始化它成为一个go 模块。我们将在/tmp目录中创建这个新目录， 你可以在文件系统的任何目录创建它都是可以的（但是我们建议你不要在GOPATH目录创建这个新目录）。我们使用模块名github.com/user/go-libp2p-tutorial来初始化它。但是你也可能想用你自己的代码库相应的名字去初始化，以方便你想发布你自己的代码版本。



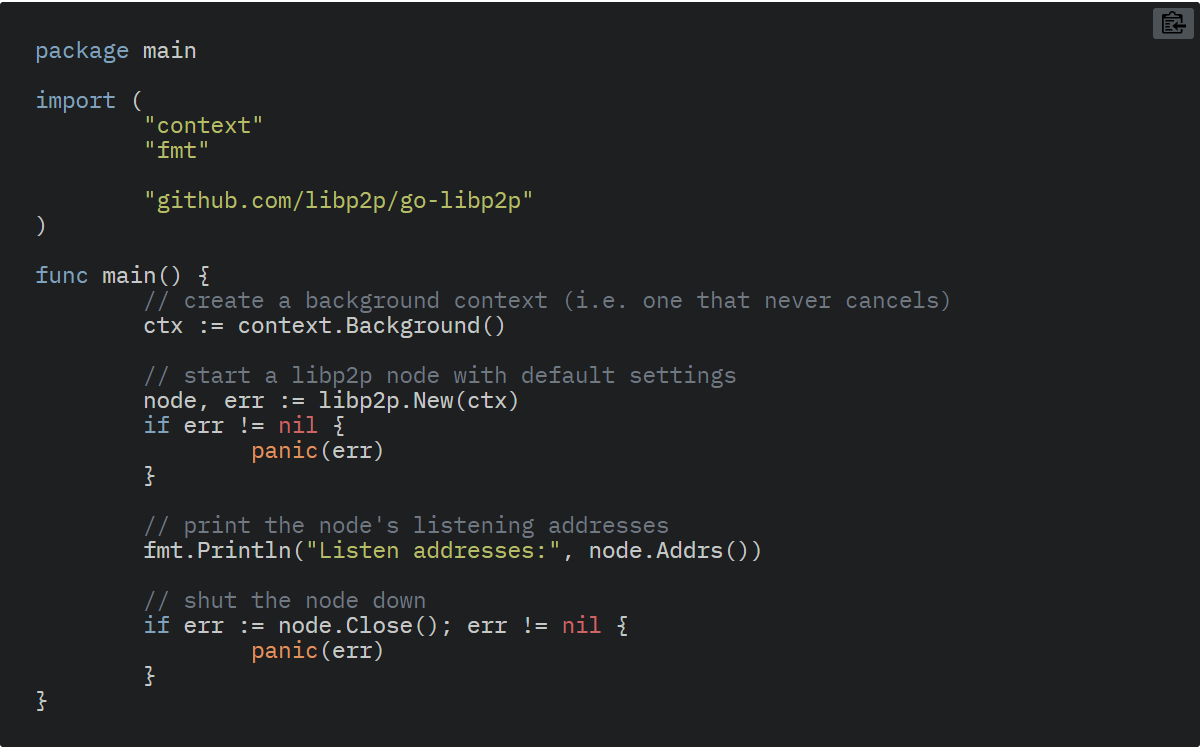
在当前目录，你应该有一个叫做go.mod的文件，这个文件的内容包括你初始化模块的名字和你当前使用的go 的版本。



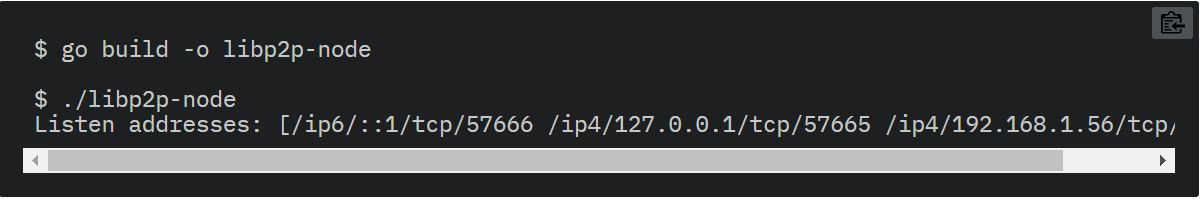
## 启动libp2p节点

我们将添加一些代码到我们的模块，并启动一个libp2p节点。

我们首先添加一个叫main.go的文件， 使用默认设置来启动一个libp2p节点，功能是打印这个节点的监听地址，然后关闭它。



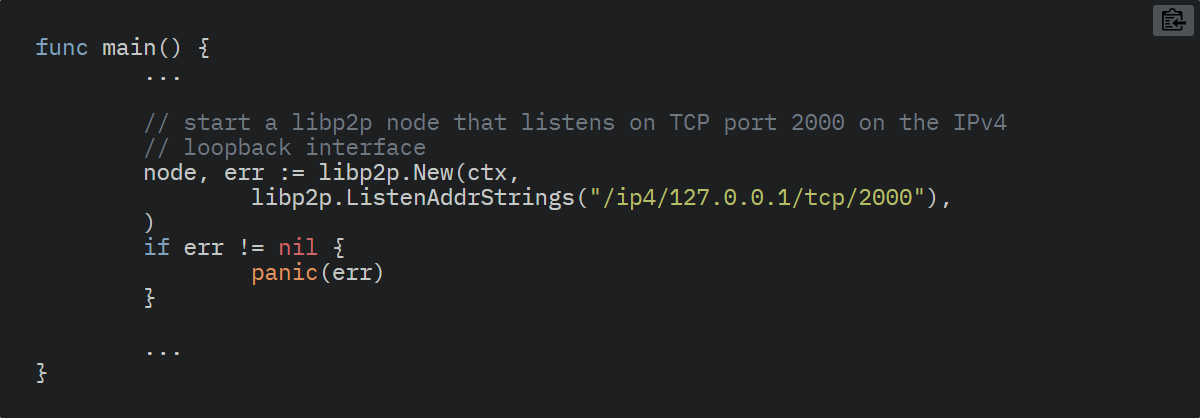
我们现在可以使用go build编译这段代码，然后在命令行运行：



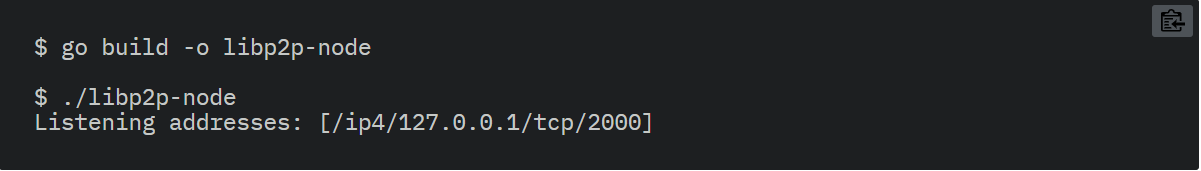
这个监听地址被使用multiaddr 格式化了，go-libp2p在默认情况下会监听所有可用的IPv4和IPv6网络。

## 配置节点

通过向 libp2p.New传递参数可以修改节点的默认设置。让我们使用libp2p.ListenAddrStrings来配置节点IPv4回环地址的监听端口为2000。



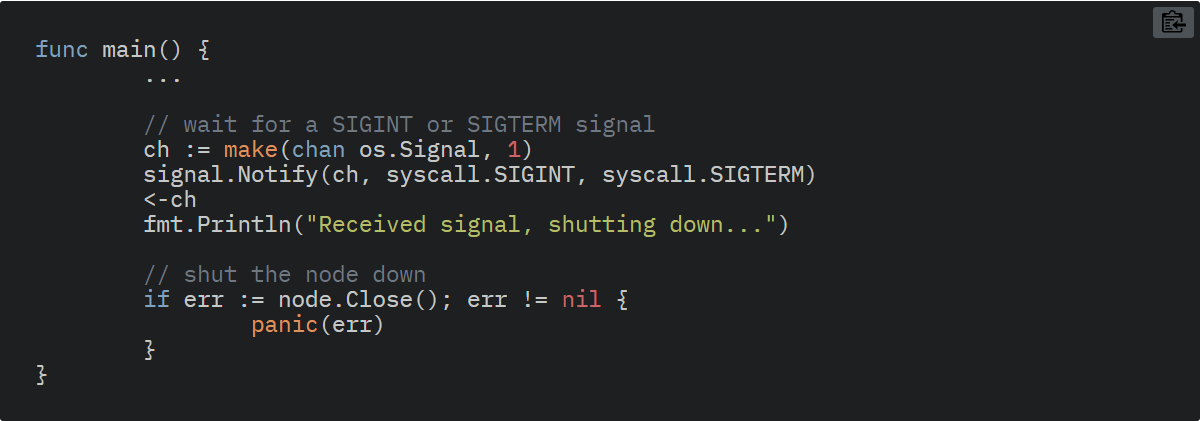
现在重新编译并运行可执行程序，可以看到现在打印的是我们配置的地址：



Libp2p.New 可以接收各种大量的不同方面的节点配置参数。详细请看[options.go](https://github.com/libp2p/go-libp2p/blob/master/options.go) 。

## 信号等待

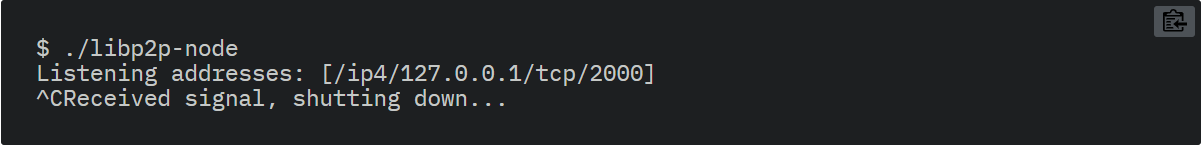
一个立即对出的节点没有什么用处。让我们在main函数的结尾添加一个等待操作系统信号的代码块，用于阻塞程序防止其在我们关闭节点前退出：



我们也需要更新引用的程序包，需要增加我们现在使用的os, os/signal 和 syscall。



再次运行这段代码，它会一直运行直到收到SIGINT或SIGTERM信号为止。



## Ping协议

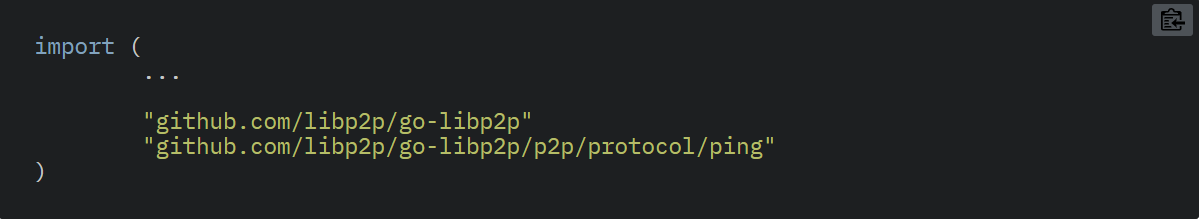
现在我们有能力配置和启动libp2p节点，可以开始讲它们之间的通讯了。

### 设置流处理器

以go-libp2p实现的节点，默认是运行它自己内部的ping协议，但是现在让我们禁止使用它的内部ping协议，然后通过手工注册字节流句柄的方式设置它，用于方便展示运行协议的过程。

libp2p.New返回的对象实现了[Host interface](https://godoc.org/github.com/libp2p/go-libp2p-host" \l "Host)，我们将使用SetStreamHandler方法为我们的ping协议设置句柄。

首先，让我们在我们的程序中引用github.com/libp2p/go-libp2p/p2p/protocol/ping程序包：



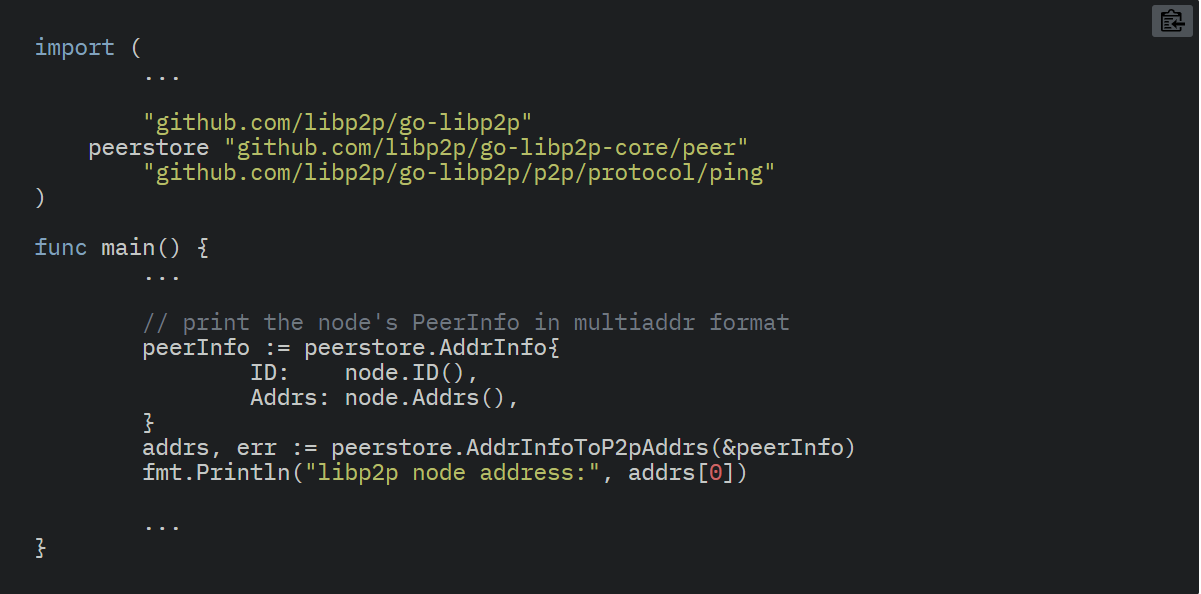
现在我们将向libp2p.New传递一个参数，用于禁止内建的ping协议， 让后通过使用ping 包里的PingService类型来手工设置流句柄（注意我们通过配置让节点监听随机的本地TCP端口，而不是硬编码一个端口，这就意味着我们可以在一台机器上运行多个节点，且这些节点不会试着去监听相同的端口）：



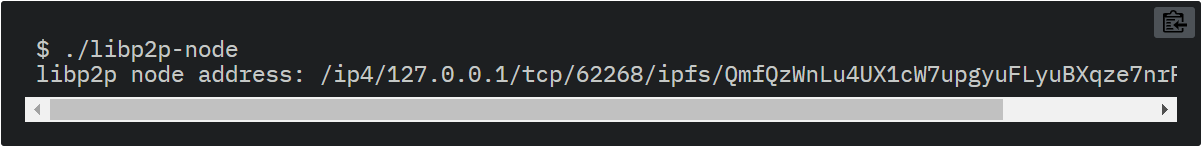
### 连接对等节点

配置好ping协议，我们需要一个方法来实现节点之间的互联，然后发送ping消息。

我们首先扩展一下我们启动节点后打印的日志，让它能够打印节点PeerId的值，然后我们要使其他节点连接到本节点。我们需要引入github.com/libp2p/go-libp2p-core/peer程序包，然后使用它来代替“Listen addresses”日志信息，用于打印地址和PeerId：



现在运行这个节点，然后打印出这个节点地址信息，可以使用这些信息连接到它。



节点需要接收一个命令行参数，这个参数是我们需要向其他节点发送ping消息的节点的地址，我们可以运行一个等待信号的监听节点，或者运行一个在关闭之前向其他节点发送多次ping消息的节点（我们使用github.com/multiformats/go-multiaddr程序包解析来自命令行参数的节点地址）：

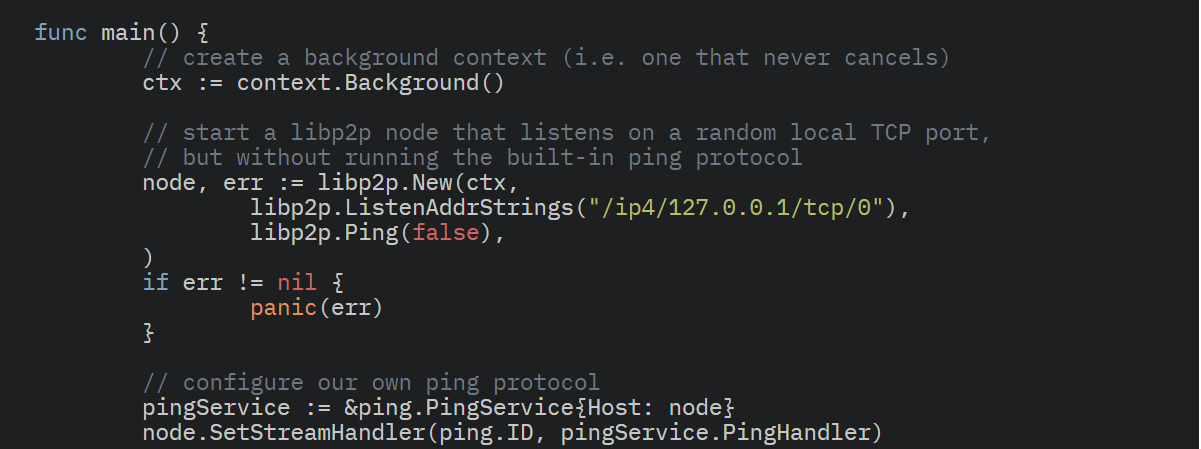


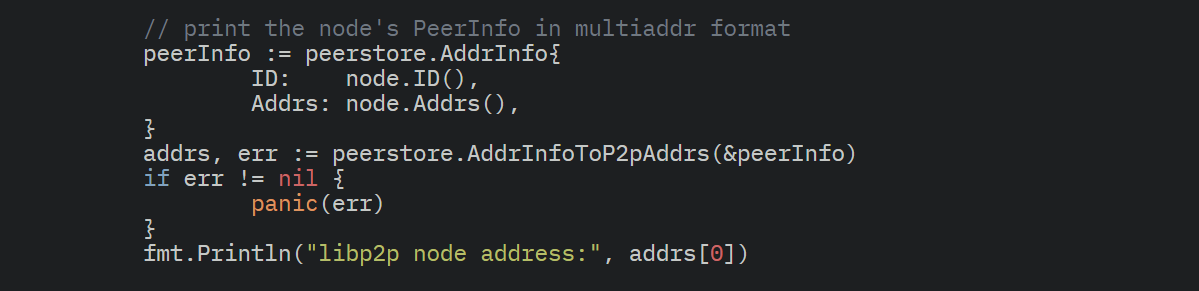
### 发送ping、pong消息

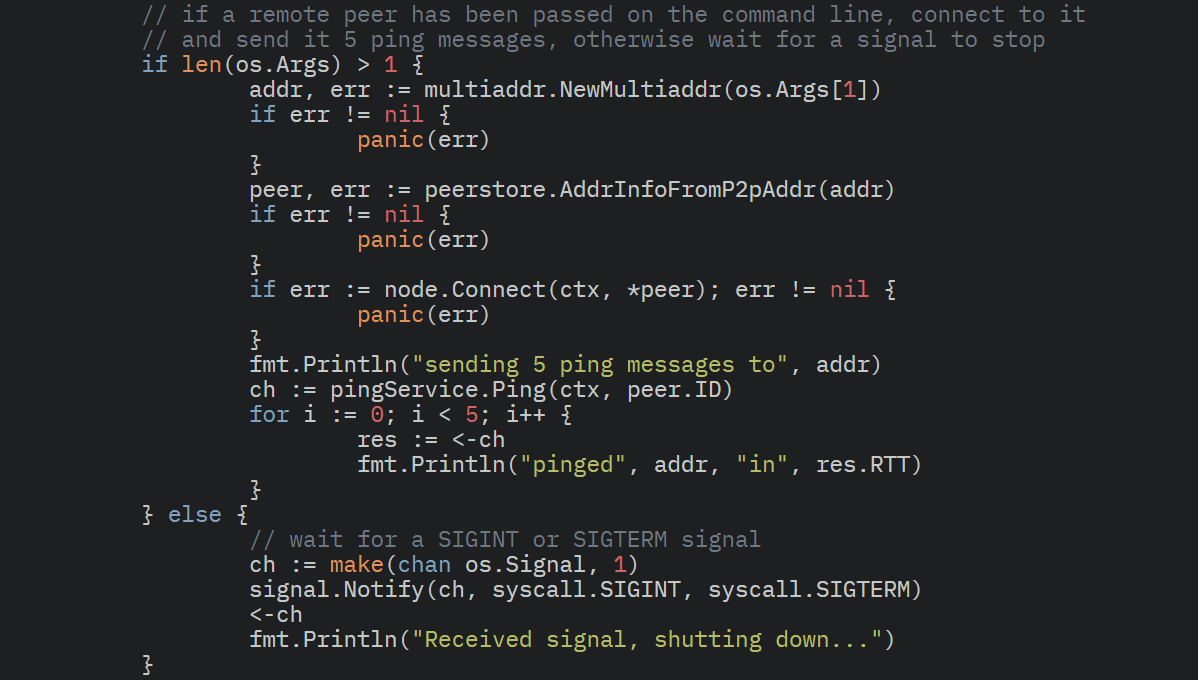
我们终于可以运行两个节点了，为它们运行一个协议，使一个节点连接另一个节点。

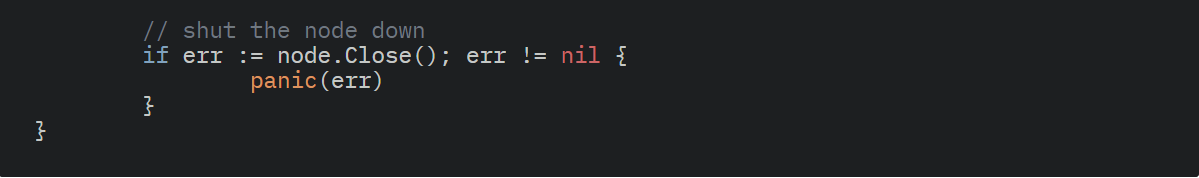
简要概括，这里是我们写的一个完整的程序：



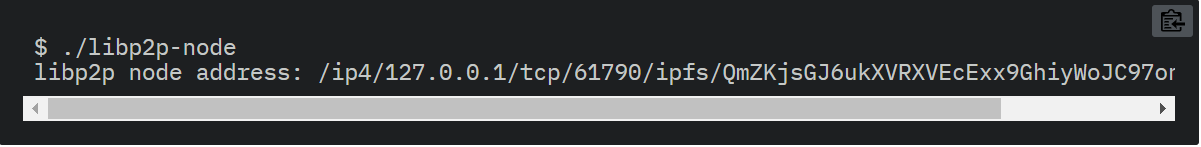




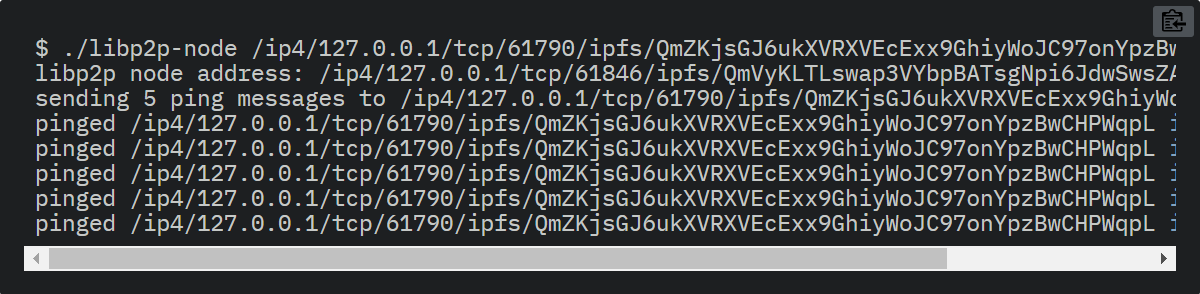




我们在一个终端窗口启动一个监听节点（i.e. 不要传递任何命令行参数）：



在另一个终端， 让我们运行第二个节点，但是要转入第一个节点的地址，我们应该能够看到一些ping消息的响应日志：



成功了！我们使用go-libp2p的两个节点现在可以通讯了！确实，它们现在只能说“ping”， 但是现在只是开始！

# 从JS-LIBP2P开始

这里是从一系列关于libp2p的javascirpt实现教程开始，*[js-libp2p](https://github.com/libp2p/js-libp2p)*

我们将带你完成一个具有基本功能的全功能libp2p节点的设置。最后我们还会在两个节点之间发送Ping消息。

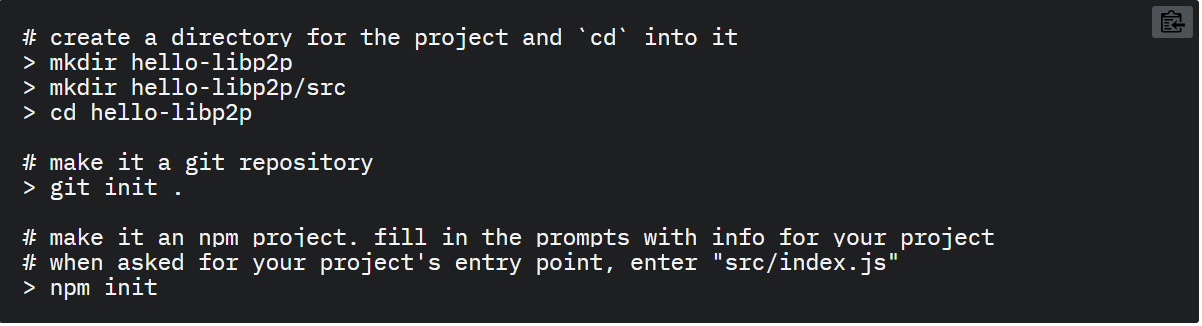
## 安装node.js

想要进行js-libp2p的开发需要[node.js](https://nodejs.org/)。如果你还没有准备node.js，不管你使用什么包管理工具，或者官方安装工具（ [using the official installer](https://nodejs.org/en/download/)），首先需要安装它。

我们推荐你使用最新版本的node.js，当然最近的几个版本也都是可以的。如果你想知道可用的最低版本是什么，你通常可以在 [js-libp2p project page](https://github.com/libp2p/js-libp2p)找到当前需要的版本。

## 创建一个空项目

我们需要一个目录来放置我们的工作，因此，打开终端在你想在的位置创建一个目录，并把它设置成一个npm项目。



注解：在整个教程中，我们使用>字符来表示shell终端提示符。当你输入命令时，不要输入 >，否则你会碰到一些奇怪错误。

## 配置libp2p

Libp2p是一个模块框架， 它允许javascript开发人员使用不同的运行时环境或者通过定制包含不同的模块来选择需要的功能特征。

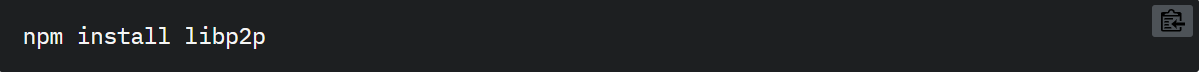
因为每个应用程序都是不同的，我们建议你只配置包含你需要的libp2p节点功能模块。如果你需要多个javacript运行时环境和功能特征，你甚至可以做多个不同配置。例如，IPFS项目有两个libp2p配置，一个是为node.js（ [one for node.js](https://github.com/ipfs/js-ipfs/blob/master/packages/ipfs-core/src/runtime/libp2p-nodejs.js)），一个是为浏览器（[one for the browser](https://github.com/ipfs/js-ipfs/blob/master/packages/ipfs-core/src/runtime/libp2p-browser.js)）。

注意：

在一个产品应用中，为libp2p节点创建一个分离的npm模块可能是有道理的，它给你提供了一个为所有javascirpt项目来管理libp2p所有依赖的地方。这个实例中，你不应该在你的程序中直接依赖libp2p，而是应该采用依赖libp2p配置模块的方式，进而达到使用你需要的任何libp2p功能模块。

如果你对libp2p是新手的话，我们推荐你采用分步骤逐步配置你的节点，参照这个方式更容易排除配置问题。在这个教程中，我们只会采用这种方式。如果你对libp2p已经很有经验了，你可能想跳过这些，直接阅读[Configuration readme](https://github.com/libp2p/js-libp2p/blob/master/doc/CONFIGURATION.md)。

按照最开始的步骤，你应该先安装libp2p模块。



## 基本设置

现在我们已经安装好了libp2p，我们现在设置能是节点运行的最小配置。Lib2p仅需要的模块是**Transport** and **Crypto。**但是我们仍推荐你在基本的配置上加上**Stream Multiplexer** ， 我们会在随后解释。首先让我以配置Transport开始。

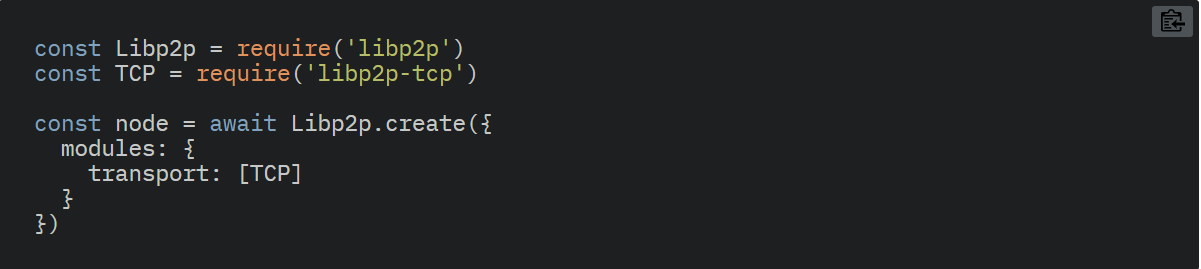
### Transports

整个网络中，libp2p使用Transport来建立对等节点直接的网络连接。你可以配置任意数量的Transports，但是最开始你只需要一个。

你应该根据你应用程序的运行时环境来选择Transports；Node.js 或者 浏览器。你可以在[configuration readme](https://github.com/libp2p/js-libp2p/blob/master/doc/CONFIGURATION.md" \l "transport)里面看到可用Transports的列表。根据这个指南，我们先安装libp2p-tcp。



现在我们已经安装好了这个模块， 让我们同过配置libp2p来使用它。我们将使用[Libp2p.create](https://github.com/libp2p/js-libp2p/blob/master/doc/API.md" \l "create)方法，这个函数是使用一个配置对象作为它唯一的参数。我们可以将Transport作为参数传递给[Libp2p.create](https://github.com/libp2p/js-libp2p/blob/master/doc/API.md" \l "create)，使其加入到modules.transport数组：

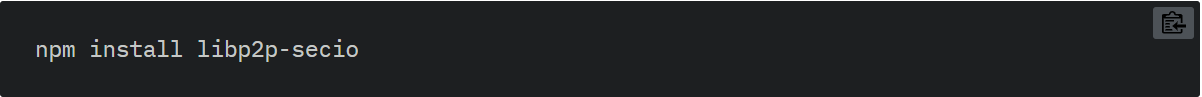


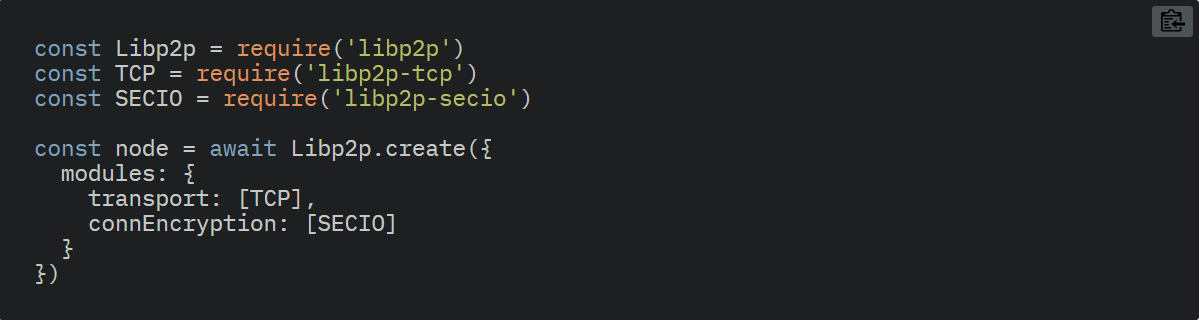
你可以向 modules.transport加入尽可能多的transpprts，以至于能够建立更可能多的节点。

### 连接加密

每一个连接都必须加密，以确保每一人的通讯是安全的。因此，连接加密是libp2p需要的模块。

为libp2p开发的加密模块数量还在持续增多。已经发布的可以在 [Connection Encryption section of the configuration readme](https://github.com/libp2p/js-libp2p/blob/master/doc/CONFIGURATION.md" \l "connection-encryption)查看。现在，我们将为我们的节点配置使用libp2p-secio模块。



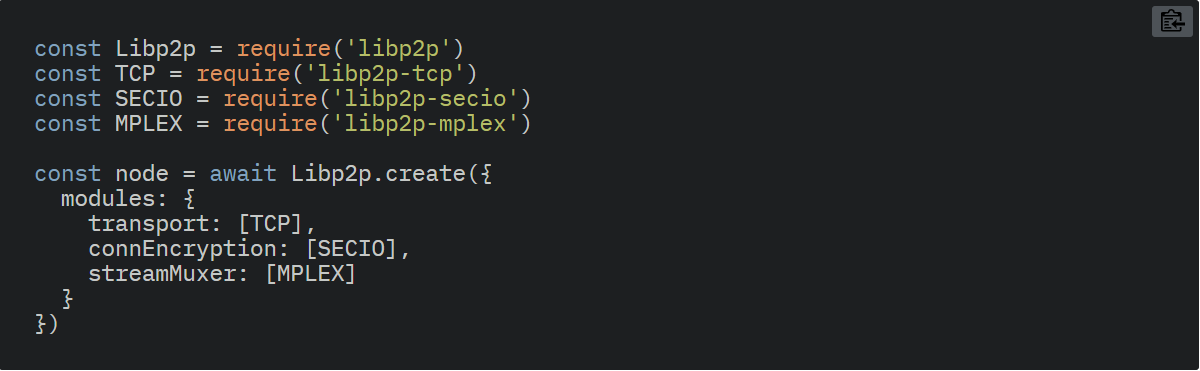


### Multiplexing

当然，multiplexers 严格意义上说并不是必须的，它们被强烈推荐是因为它们能够改善libp2p运行的各种链接协议的有效性和效率。

查看[available stream multiplexing](https://github.com/libp2p/js-libp2p/blob/master/doc/CONFIGURATION.md" \l "stream-multiplexing)模块，js-libp2p当前只支持libp2p-mplex， 因此我们将在这里使用它。在下面的例子中，你可以安装libp2p-mplex并将其加入到libp2p节点。

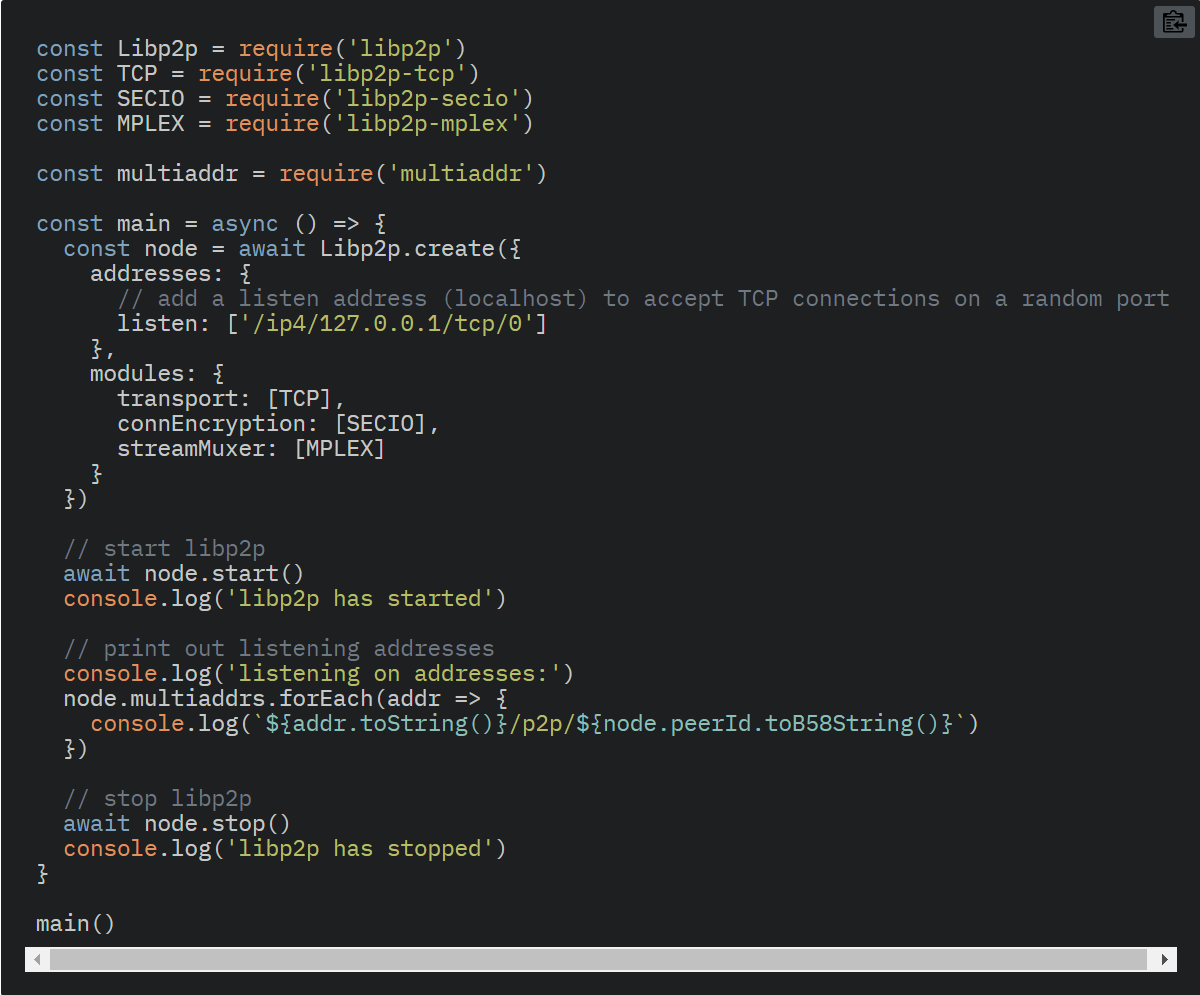




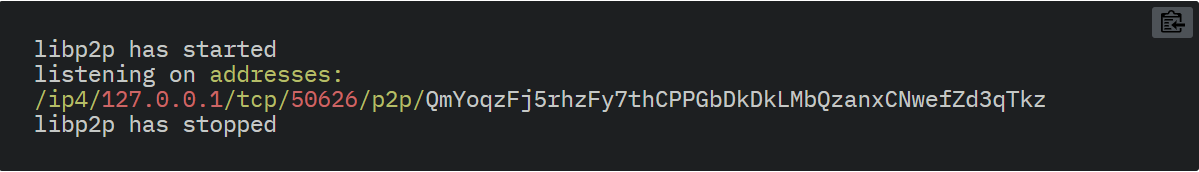
## 运行libp2p

现在你已经配置好了 **Transport**, **Crypto** and **Stream Multiplexer**模块，可以现在启动libp2p节点了。我们可以使用[libp2p.start()](https://github.com/libp2p/js-libp2p/blob/master/doc/API.md" \l "start) 和[libp2p.stop()](https://github.com/libp2p/js-libp2p/blob/master/doc/API.md" \l "stop) 方法来启动和停止libp2p节点。

一个libp2p节点需要给transport设置一个监听地址，使其他节点能够连接到它。通常，我们会安装multiaddr模块，用于创建一个TCP [multiaddress](https://docs.libp2p.io/reference/glossary/" \l "multiaddr)，并且将其添加到节点。



试着利用node src/index.js运行这些代码， 你应该可以看到一下内容：

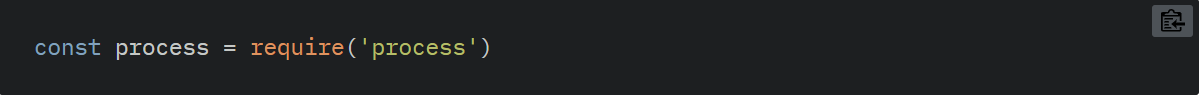


## 发送ping、pong消息

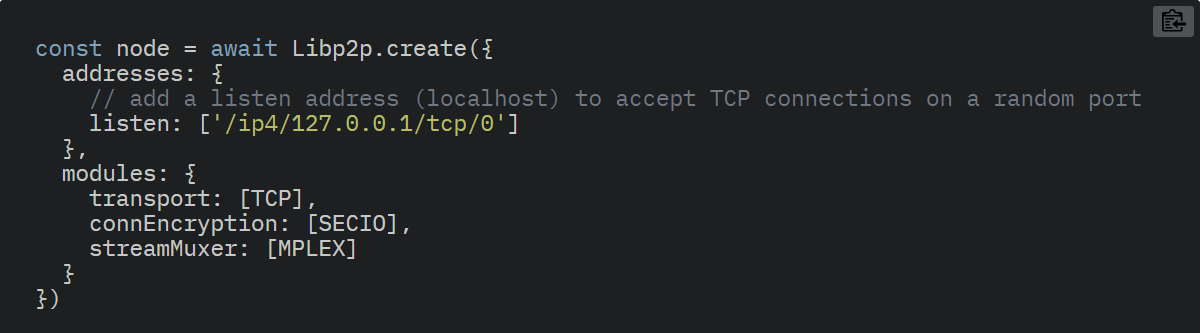
我们现在已经恰当的建好了transport, multiplexing, 和security 模块， 我们现在可以开始通讯了。

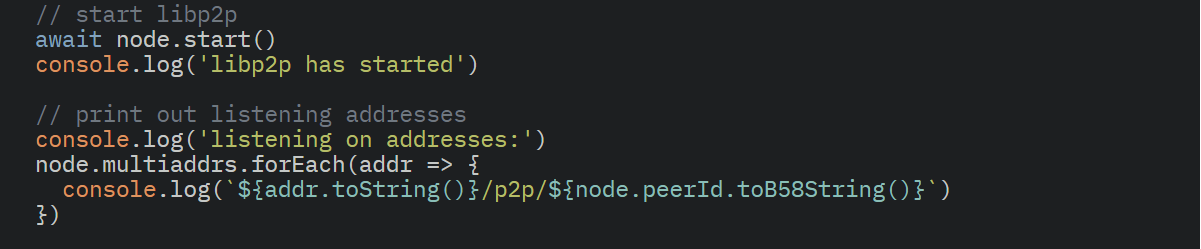
我们可以使用 [libp2p.ping()](https://github.com/libp2p/js-libp2p/blob/master/doc/API.md" \l "ping)向其他节点拨号并发送ping消息。收到ping消息的其他节点会回应pong消息，这样我们就知道它还是一个活动节点。我们也可以使用这个方式来测量节点之间的消息延迟情况。

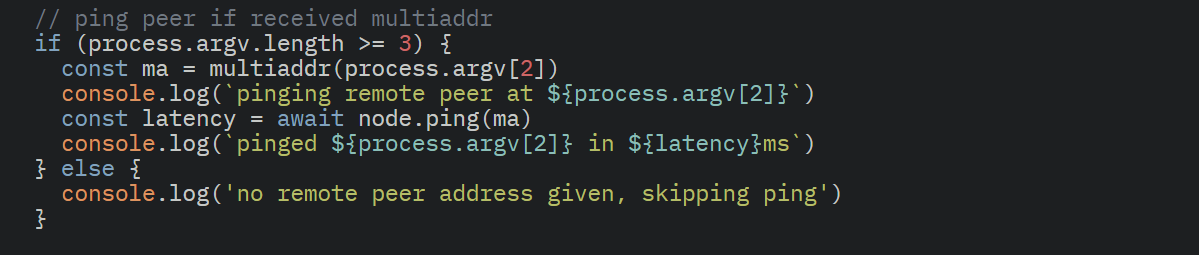
我们也可以通过命令行参数设置我们的应用程序，使其接收一个多地址节点，并向它发生ping消息。为了达到这个目的，我们还需要加入两样东西。首先，我们需要process模块，通过该模块我们能够获得命令行参数：

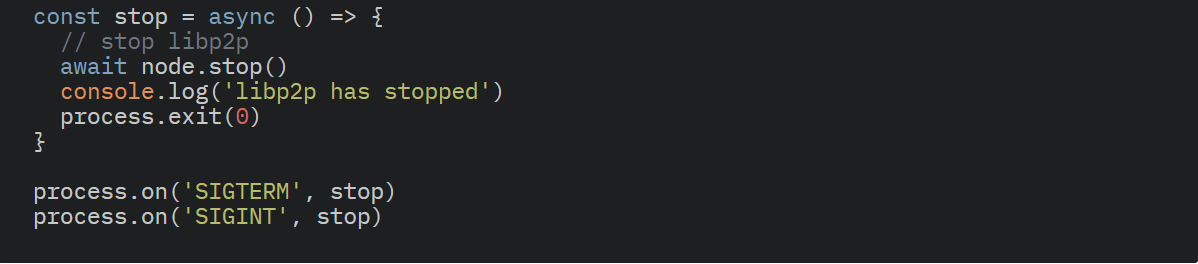


然后我们需要解析这个来自命令行的多地址参数并试着向其发送ping消息。

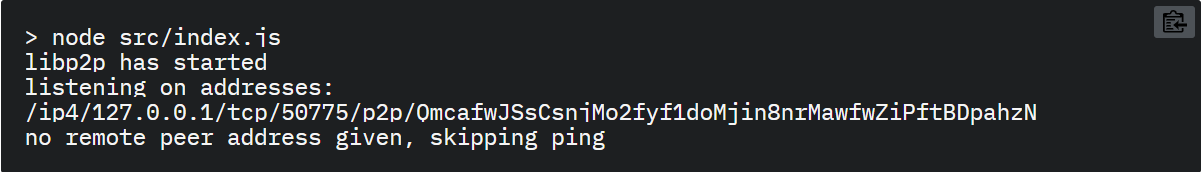




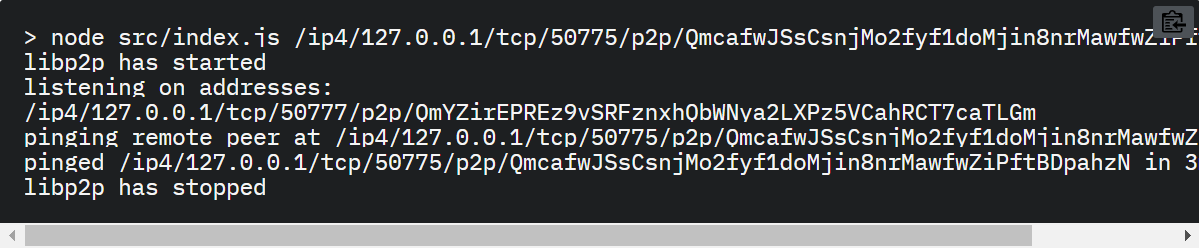




现在我们可以启动一个没有参数的节点：



复制上面从 /ip4/...开始的地址，用其做为参数在另一个终端再启动一个节点：



成功了！我们的两个节点在多路复用和安全的通道上进行了通讯。当然，它们只能发送ping消息，但是这只是开始！

## 下一节是什么？

完成这个教程之后，你应该进一步阅读[js-libp2p getting started](https://github.com/libp2p/js-libp2p/blob/master/doc/GETTING_STARTED.md)文档，从这个文档中你既可以了解到如何对libp2p进行基本的配置，也可以学到如何对libp2p进行定制化配置。

你也可以研究 [js-libp2p repo](https://github.com/libp2p/js-libp2p/tree/master/examples)库里的大量代码示例，从这些代码中你可以学到在不同的场景和运行时环境下如何使用 js-libp2p。

# 示例

在这里你可以看到很多用于展现libp2p的关键特性而编写的示例。

Go示例： [go-libp2p-examples repo](https://github.com/libp2p/go-libp2p-examples).

Javascript示例： [/examples directory of the js-libp2p repo](https://github.com/libp2p/js-libp2p/tree/master/examples).

Rust示例： [/examples directory of the rust-libp2p repo](https://github.com/libp2p/rust-libp2p/tree/master/examples).

# 概念

Libp2p涵盖了很多领域，涉及了很多不熟悉的术语和概念。这个章节将讲述libp2p涉及的一些基本概念。

* [Transport](https://docs.libp2p.io/concepts/transport/)
* [NAT Traversal](https://docs.libp2p.io/concepts/nat/)
* [Secure Communication](https://docs.libp2p.io/concepts/secure-comms/)
* [Circuit Relay](https://docs.libp2p.io/concepts/circuit-relay/)
* [Protocols](https://docs.libp2p.io/concepts/protocols/)
* [Peer Identity](https://docs.libp2p.io/concepts/peer-id/)
* [Content Routing](https://docs.libp2p.io/concepts/content-routing/)
* [Peer Routing](https://docs.libp2p.io/concepts/peer-routing/)
* [Addressing](https://docs.libp2p.io/concepts/addressing/)
* [Security Considerations](https://docs.libp2p.io/concepts/security-considerations/)
* [Publish/Subscribe](https://docs.libp2p.io/concepts/publish-subscribe/)
* [Stream Multiplexing](https://docs.libp2p.io/concepts/stream-multiplexing/)

注意：

这个章节不能涵盖完整的内容，大家可以阅读相关主题的论文。

# 传输层

当你用你自己的计算机连接到网络上另一台计算，你有会有很好的使用TCP/IP发送字节数据的机会，因为这个过程非常成功的融合了互联网协议和传输控制协议。互联网协议（IP）负责寻址和传输数据包，传输控制协议（TCP）是用于确保被发送的数据能够完全并且以正确的顺序被接收方接收。

因为TCP/IP是无处不在的并且得到了很好的支持，因此它通常是网络应用程序的默认选择。在一些场景中，TCP增加了很多开销，因此一些应用程序可能使用 [UDP](https://en.wikipedia.org/wiki/User_Datagram_Protocol)协议，这是一个不保证顺序和可靠性的更简单的协议。

虽然TCP和UDP（还有IP）是现今最常用的协议，但是这并不意味着他们是唯一的选择。在底层（发送原始以太网数据包或者蓝牙框架）和上层（例如QUIC,它是比UDP更上层的协议）都存在一些其他协议可以选择。

在libp2p中，我们调用这些基本协议，使得数据在 **transports**层流转，libp2p的其中一个重要需求就是使传输层协议不可知。这就意味着决定使用什么样的传输层协议完全依赖开发者，事实上，一个应用程序可以同时支持许多不同的传输层协议。

## 监听和拨号

传输层有两个核心操作，监听和拨号。

不论传输层实现提供的是什么样的设施，监听意味着你都可以接收来自其它对等节点的链接。例如，Unix操作系统平台上的TCP协议是用 bind 和 listen系统调用来实现向应用程序的TCP端口发送系统流量数据的。

Dialing是向外发送连接请求到监听节点的过程。与监听一样，实现决定了它的特定过程，但是在libp2p实现中，每个传输协议都会共享使用相同的编程接口。

## 寻址

在你可以拨号一个对等节点和建立链接之前，你需要先知道如何才能联系到它们。因为每种传输协议很可能都有它自己的地址框架，libp2p使用一个惯例上称作多地址或者multiaddr的方法来编码编码许多不同的寻址框架。

文档[addressing doc](https://docs.libp2p.io/concepts/addressing/) 讲述了更详细的信息， 但是整体上了解多地址是如何工作的对理解拨号和监听接口是非常有帮助的。

这里有一个关于TCP/IP传输协议的多地址的例子：



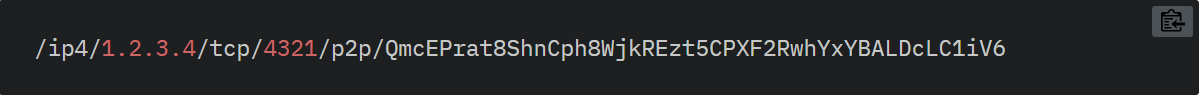
这个和我们熟悉的 7.7.7.7:6542格式是等价的，但是它还有其他好处，比如它显示的描述了这个协议，从它的多地址格式上，我们可以了解到它的地址 7.7.7.7是属于IPv4协议，而且6543是输入TCP协议。

想要了解更为复杂的示例，可以看[Addressing](https://docs.libp2p.io/concepts/addressing/)。

拨号和监听都可以处理多地址形式。当监听时，你给传输协议指定你要监听的地址，当拨号时，你提供要被拨号的地址。

当要拨号远程对等节点时，这个多地址应该包含你将要试图建立连接的节点的 [PeerId](https://docs.libp2p.io/concepts/peer-id/)。这可以让libp2p建立一个安全的信道，并阻止假冒。

一个包含PeerId的多地址例子：



/p2p/QmcEPrat8ShnCph8WjkREzt5CPXF2RwhYxYBALDcLC1iV6唯一的标识了一个远程对等节点， 它是有该节点的公钥哈希得来的。想要了解更多，请看 [Peer Identity](https://docs.libp2p.io/concepts/peer-id/)。

小贴士：

当节点路由可以使用时，你可以直接使用它们的PeerId进行拨号，而不需要提前知道它们的传输层地址。

## 支持多种传输协议

Libp2p应用程序通常需要同时支持多种传输协议。例如，你可能需要你的服务即对使用TCP协议的长期运行的应用可用，也可以接收来自运行在网络浏览器上的节点发送的链接。

Libp2p中用于管理传输协议的组件叫做[switch](https://docs.libp2p.io/reference/glossary/" \l "switch)，它也会与[protocol negotiation](https://docs.libp2p.io/concepts/protocol-negotiation), [stream multiplexing](https://docs.libp2p.io/concepts/stream-multiplexing), [establishing secure communication](https://docs.libp2p.io/concepts/secure-comms/) 等一些链接升级的其他形式的协议进行协作。

Switch为拨号和监听提供单一实体点，并且使你的应用程序代码不用担心特定的传输协议或者其他被底层使用到的连接技术栈。

# NAT穿透

因特网是有无数网络组成的，它们被最基础的[transport protocols](https://docs.libp2p.io/concepts/transport/)协议绑定在一起共用地址空间。

当数据在网络直接流转，一个叫做网络地址转换的过程经常出现。网络地址转换将一个地址从一个地址空间映射到另一个地址空间。

NAT允许很多设备共用一个公共地址，这也是IPv4协议到现在还能正常工作的基础，否则根据它32位的地址空间是不足以满足现在的网络设备需求的。

例如，当我连接到我们家的wifi，我的计算机获得了一个10.0.1.15IPv4地址。它属于为私有网络保留的内部网络地址。当我向外部公共网络地址建立一个连接，路由器会使用它自己的公共网络地址代替我的内部网络地址。当数据从其他地方传入时，路由器会将地址转换回内部地址。

然而NAT对外部连接是透明的，监听连入的连接需要一些配置。路由器在单一的公共IP地址上进行监听，但是内部网络中任何数量的计算机都能处理请求。为了服务这些请求，你的路由器必须为特定的计算机配置特定的发送通道，通常是通过一个或多个公共IP的TCP或UDP端口映射到一个内部网络地址。

虽然通常可能要手工配置路由器，但是不是每个想运行p2p应用节点或其他网络服务的人有能力这么做。

我们想让libp2p应用运行在任何地方，不只是数据中心或其他有固定公共网络IP地址的计算机上。为了实现这个目标，在现今的libp2p中已有几种主流的网络穿透方法。

## 自动路由配置

许多路由器都为端口转发提供自动路由配置，最常用的是 [UPnP](https://en.wikipedia.org/wiki/Universal_Plug_and_Play) 和 [nat-pmp.](https://en.wikipedia.org/wiki/NAT_Port_Mapping_Protocol)

如果你的路由器支持这两种协议中的一种，libp2p就会尝试自动配置端口映射，使得它能够监听连入请求。如果被网络和libp2p实现支持，这通常是最简单的选项。

Libp2p实现支持多种自动NAT配置。详细请看[current implementation status](https://libp2p.io/implementations/" \l "nat-traversal)。

## Hole-punching (STUN)

当一个内部网络的计算机拨出并与有公共地址的计算机建立连接，路由器会为这个连接映射一个公共端口到内部网络IP地址。在一些情况下，你的路由器会从这个公共端口接收接入的连接请求并将该链接路由到相应的内部IP地址。

在使用支持IP的传输时，libp2p会尝试利用这个行为，使用一个名为SO\_REUSEPORT的套接字选项，通过使用相同的端口进行拨号和监听。

如果我们的节点拥有良好的网络环境，它们是可以建立一个外部连接并获得一个公共网络可以联系的监听端口，但是它们可能永远也不知道这个事情。不幸运的是，现在没有一个可行的方法为发出拨号的程序提供发现哪个端口被分配给了自己这个连接。

但是，一个外部节点可以告诉我们它们观察到了什么地址。我们能够获得这个地址，并将其在我们的网络中（ [peer routing network](https://docs.libp2p.io/concepts/peer-routing/)）广播给其它节点，使它们知道从哪里发现我们。

将它们观察到的地址通知给其它节点的这个基本前提是 [STUN](https://en.wikipedia.org/wiki/STUN)（是NAT的会话穿透实用工具）的基础，STUN是描述用于发现公开可达IP地址和端口组合的客户端/服务器协议。

Libp2p其中的一个核心协议是认证协议[identify protocol](https://github.com/libp2p/specs/pull/97)，它允许一个节点向另一个节点询问认证信息。当节点发送公钥和其他的一些有用信息，这个节点就会被其他节点进行身份认证，其中包括这个节点所观察到的地址集合。

外部节点发现机制与STUN扮演相同角色，但是不需要STUN 服务器集合。

身份认证协议允许一些对等节点在NAT之间进行通讯，否则它们无法穿过NAT。

## AutoNAT

虽然上面描述的身份认证协议可以让对等节点相互之间通知它们观察到的网络地址，但是并不是所有的网络都允许通过它们拨号出去的端口接收连入连接。

再次，其他节点可以帮助我们观察我们自己的情况，这个过程是通过其他节点尝试拨号我们观察到的地址实现。如果这个过程是成功的，我们也可以依靠其他可以拨号我们的节点并且广播我们的监听地址。

Libp2p的一个叫做AutoNAT 的协议可以让节点从通过提供AutoNAT服务的节点发送回拨请求。

## Circuit Relay (TURN)

在一些情况下，对等节点不能够以公开的方式遍历它们的NAT。

Libp2p提供了[Circuit Relay protocol](https://docs.libp2p.io/concepts/circuit-relay/)协议，使得对等节点能够通过中间帮助节点进行间接通讯。

这里提供了与在其他系统中使用的 [TURN protocol](https://tools.ietf.org/html/rfc5766)协议相似的功能。

# 网络中继

网络中继是一种传输协议（[transport protocol](https://docs.libp2p.io/concepts/transport/)），它通过第三方中继对等节点实现两个对等节点之间进行通讯。

在很多场景中，对等节点不能以穿透它们的NAT网络（[traverse their NAT](https://docs.libp2p.io/concepts/nat/) ）来实现它们可以在公共网络上被访问， 或者它们没有共享它们的传输协议（[transport protocols](https://docs.libp2p.io/concepts/transport/) ），使得它们不能被直接通讯。

为了能够使p2p架构面临的连接障碍问题比如NAT得到处理，libp2p定义了一个叫做p2p-circuit的协议（[defines a protocol called p2p-circuit](https://github.com/libp2p/specs/tree/master/relay)）。当一个对等节点不能够在公共地址上进行监听时，它可以拨号中继对等节点，这个中继对等节点可以保持长时间的连接活跃状态。其他对等节点可以通过拨号p2p-circuit地址，间接的连接到它们的目的地址。

这个网络中继协议是收到了 [TURN](https://tools.ietf.org/html/rfc5766)的启发，TURN是交互式连接建立（ [Interactive Connectivity Establishment](https://tools.ietf.org/html/rfc8445)）的NAT穿透技术集合的一部分。

注意：

中继连接是端对端加密的，那也就意味着作为中继的对等节点不能够读取或篡改经过它的数据。

不透明是中继协议的一个重要方面。换句话说，源端节点和目的端节点都能知道它们之间的消息是被中继传递的。这个非常有用，因为目的端节点能够知道建立连接使用的中继地址并且使用该地址构建返回路径连接，用于发送消息到源端地址。这也不是匿名的，所有参与方都是使用它们的节点ID来识别它们，包括中继节点在内。

## 中继节点地址

中继是使用一个包含了对等节点ID的多地址来识别的，这个对等节点的消息会被中继传输。

让我们举个例子，我有一个节点ID是QmAlice的对等节点，我想把我的地址给我的朋友QmBob，但是我在一个NAT网络中，别人不能直接和我建立连接。

我能够构建的最基本的p2p-circuit地址是：

/p2p-circuit/p2p/QmAlice

上面的这个地址非常有趣，因为这个地址不包括任何我们想构建的连接到QmAlice的地址和任何进行传输消息的中继节点地址。没有这些地址信息，一个对等节点唯一拨号我的机会是发现一个中继节点并且这个节点是和我保持连接的。

一个比较好的节点地址类似/p2p/QmRelay/p2p-circuit/p2p/QmAlice。这个包含了指定的中继节点ID，QmRelay。如果一个对等节点知道如何与QmRelay建立连接，它们就能够连接到我。

一个好的节点地址也为中继节点包含传输地址。让我们举个例子，我已经与指定的节点ID是QmRelay的中继节点建立了一个连接。它们通过身份协议告诉我，它们是地址为7.7.7.7，端口为55555监听上的连接。我可以依托于这个特定中继节点传输构建一个可以联系到我的路径地址：  
/ip4/7.7.7.7/tcp/55555/p2p/QmRelay/p2p-circuit/p2p/QmAlice

这个地址中/p2p-circuit之前的所有内容都是中继节点的地址，它包括了传输地址和它们的对等节点ID QmRelay。 /p2p-circuit/之后的内容是我的节点ID，QmAlice。

通过把这个详细的中继地址给我的朋友QmBob，他们就能够快速的与我建立一个中继的连接，从而不用到处去询问到达QmAlice的中继路径。

当广播你自己的地址，提供一个包含了中继节点传输地址的中继地址是最好的。如果这个中继地址包含了许多传输地址，你可以为他们每个地址包含一个p2p-circuit。

## 自动中继

中继协议只有当节点能够发现愿意成为中继节点并且能够与双方节点建立连接的节点时才有效。

虽然一个可能的简单方案是在你的应用程序中硬编码添加一个已知的中继节点列表，但是这添加了一个在你的应用架构中你可能想避免的中心集合点。如果这些引导节点变的不可用，它们这种引导节点列表也可能是潜在的失败节点。

自动中继是一个优点（当前已经在go-libp2p中实现），它使得对等节点能够利用libp2p内容路由接口（ [content routing](https://docs.libp2p.io/concepts/content-routing/) ）尝试发现中继节点。

当自动中继可用时，对等节点会尝试发现一个或更多公共中继节点并打开中继连接。如果成功，这个节点就会利用libp2p的节点路由（ [peer routing](https://docs.libp2p.io/concepts/peer-routing/)）系统进行广播中继地址。

警告

自动中继还正在积极开发中并且应该考虑到实验性质。当前没有保护措施用于避免恶意的或篡改用于广播中继服务和拒绝提供服务的中继节点。

#### 自动中继如何工作

自动中继服务负责：

1. 发现周围的中继节点
2. 在它们之间建立长连接
3. 为我们的节点广播有中继能力的节点，通过代理路由使我们自己可路由。

当 [AutoNAT service](https://docs.libp2p.io/concepts/nat/" \l "autonat) 服务检测到我们在NAT网络中并且阻止连接我们的连入连接，自动中继进入激活状态，执行以下流程：

1. 我们通过运行DHT模块搜索/libp2p/relay命名空间来定位候选中继节点。
2. 我们随机的选择三个结果节点，并与它们建立长连接（/libp2p/circuit/relay/0.1.0协议）。后续将支持延迟启发式选择功能。
3. 我们可以利用/ip4/1.2.3.4/tcp/4001/p2p/QmRelay/p2p-circuit的格式形式来增强本地地址列表与需要的新的可中继的多地址之间的联系， 1.2.3.4是中继节点的公共IP地址，4001是libp2p端口，QmRelay是中继节点的节点ID。这个多地址的元素可以根据实际传输的情况进行改变。
4. 我们通过使用IdentifyPush协议向我们已经连接的对等节点广播我们新的可以作为中继节点的节点地址。

这最后一步是关键，因为它能够是我们连接的对等节点知道我们已知的可作为中继节点的地址更新，反过来，当其他节点向我们查询的时候，返回这些新学到的节点。

# 协议

当你写网络应用程序时，你到处可以看到很多协议，在libp2p中，它们的数量尤其多。

这篇文章所涉及到的协议主要是用建立libp2p，使用核心的libp2p抽象，例如传输、对等节点标识、寻址等等。

在整个文章中，我们把用于构建libp2p的协议都叫做 libp2p protocol，但是有时你也可能看到它们没称作 “wire protocols” 或 “application protocols”协议。

这些协议用于定义你的应用程序并提供它的核心功能。

这篇文章会讲到libp2p协议的一些关键特征（ [defining features of a libp2p protocol](https://docs.libp2p.io/concepts/protocols/" \l "what-is-a-libp2p-protocol)），会给出这个协议谈判过程的概要信息（[protocol negotiation process](https://docs.libp2p.io/concepts/protocols/" \l "protocol-negotiation)），也会给出包含在libp2p并提供关键功能的核心libp2p协议（ [core libp2p protocols](https://docs.libp2p.io/concepts/protocols/" \l "core-libp2p-protocols)）的概要信息。

## 什么是libp2p协议

Libp2p包含以下关键特征：

### 身份标识协议

Libp2p协议有唯一的字符串身份标识符，当连接是第一次建立时，协议谈判过程将使用它。

按照惯例，身份标识协议定义的标识符结构有些类似路径地址，并且以版本号结束。



改变你协议的格式或语义应该产生一个新的版本号。阅读[protocol negotiation section](https://docs.libp2p.io/concepts/protocols/" \l "protocol-neotiation)，你可以了解到更多关于版本号是如何在拨号和监听过程中起作用的。

虽然libp2p在技术上是可以接受任何有效的协议标识符字符串，但是这个推荐的包含版本号的结构即可以做到对开发者友好，又容易进行版本匹配（ [easier matching by version](https://docs.libp2p.io/concepts/protocols/" \l "match-using-semver)）。

### 函数句柄

为了接受连接，libp2p应用程序会使用它们的协议标识符在 [switch](https://docs.libp2p.io/reference/glossary/" \l "switch)或更高层的接口（比如 [go’s Host interface](https://github.com/libp2p/go-libp2p-host/blob/master/host.go)）中注册函数句柄。

当一个输入流被标记了协议标识符，就会调用相应的函数句柄。当你将函数句柄指向了一个匹配函数（[match function](https://docs.libp2p.io/concepts/protocols/" \l "using-a-match-function)），你可以选择是否接受协议标识符的非精确字符串匹配，例如，匹配主要语义的版本（ [semantic major versions](https://docs.libp2p.io/concepts/protocols/" \l "match-using-semver)）。

### 二进制流

由libp2p传输的多媒体数据是一个包含以下特性的双向的二进制流：

双向，可靠的二进制数据传输

任何时间，每一边都可以在这个流上进行读写

读的数据顺序与它们被写入的顺序一致

可以半关闭，比如关闭写不关闭读或不关闭写只关闭读

支持反压

读者方不会被数据洪流攻击

在这些场景中，libp2p也能够确保这个流是安全且有效率的，这些对协议句柄是透明的，它们从这些流中进行的读写是非加密的二进制数据。

这二进制数据的格式和被发送数据的结构以及什么时候和被谁发送都是由协议决定的。下面概述了在libp2p内部协议中使用的一些常规模式（[common patterns](https://docs.libp2p.io/concepts/protocols/" \l "common-patterns)）。

## 谈判协议

当拨号并初始化建立一个新的数据流，libp2p就会发送你想使用协议的协议标识符。另一端的对等节点会以注册的协议句柄来检查连入的协议标识符。

如果这个监听节点不支持请求的协议，它会终止这个流，发起拨号的对等节点可以尝试发起不同的协议，或者是刚才发起的协议的旧版本协议。

如果这个协议是支持的，监听节点就会回应这个协议标识符，使其做为一个允许在这个流上发送符合该协议结构的信息的信号。

这个在流或链接上达成使用什么协议的过程叫做谈判协议。

### 匹配协议标识符和版本号

当你注册一个协议句柄时，你有两个函数可以使用。

第一个函数接收两个参数：一个协议标识符，一个是函数句柄。如果一个输入流要求发送一个精确匹配的协议标识符，这个句柄函数就会被调用，并接受一个新的流作为参数。

#### 使用匹配函数

第二个注册协议的函数有三个参数：协议标识符，匹配函数，函数句柄。

当一个流请求来自于协议标识符不是精确匹配的节点时，这个协议标识符会被发送给所有的已注册的匹配函数。如果它们返回true，则相关的处理函数会被调用。

这使得你可以灵活的处理你的模糊匹配，为你的应用程序定义合理的协议匹配规则。

#### 语义化匹配

如果你想同时支持多个版本，你可能会想使用语义版本化。

在go-libp2p中，一个叫做[MultistreamSemverMatcher](https://github.com/libp2p/go-libp2p-core/blob/master/helpers/match.go)的帮助函数可以当做协议匹配函数来用，它能够检测请求是否满足已注册的协议版本。

在js-libp2p的 [js-multistream-select](https://github.com/multiformats/js-multistream-select/)部分中提供了相似的匹配函数（ [similar match function](https://github.com/multiformats/js-multistream-select/blob/master/src/listener/match-semver.js)）。

### 拨号一个特定协议

当拨号一个远程节点建立一个新的流时，初始化的节点会发送一个它们想使用的协议标识符。远程节点会使用上面讲述到的匹配逻辑来接受或拒绝这个协议。如果这个协议被拒绝，拨号节点还可以再次尝试。

当拨号的时候，你可以选择提供一个协议标识符列表来替代单一的协议标识符。当你提供多个协议标识符是，它们会被连续的尝试匹配，如果远程节点支持多个协议，那么第一个被匹配的协议将会被使用。如果你支持多种协议时，这个方式会非常有用，这是因为如果远程节点还不支持最新版本的协议，你可以提议最新最近版本和以前的多个旧版本协议。

## Libp2p核心协议

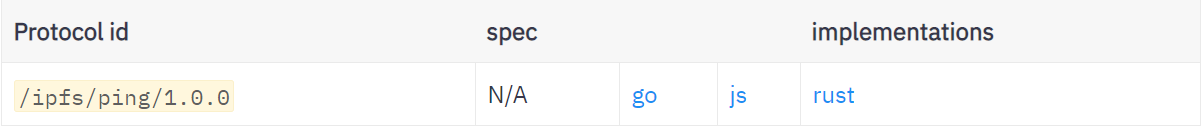
除了你在开发应用程序时自己写的一些协议外，libp2p自己定义了一些被它的核心特征功能所使用的基础协议。

### 通用模式

下面讲到的所有协议都是使用 [protocol buffers](https://developers.google.com/protocol-buffers/)定义消息框架。

消息的交换都是使用一个简单的管理方法，在二进制数据的前面添加一个表示这个数据长度的整数，这个长度按[protobuf varint](https://developers.google.com/protocol-buffers/docs/encoding" \l "varints)的方式编码。

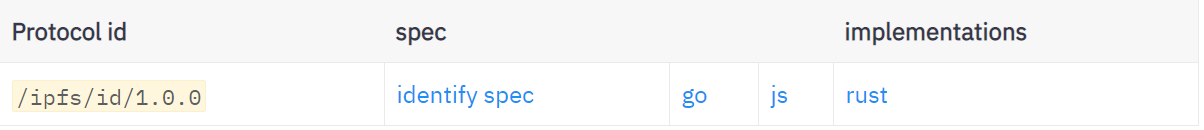
#### Ping



Ping协议是一种简单的活性检测，节点可以使用它快速的知道其他节点是否在线。

协议信息交换完成后，拨号节点发送32字节的随机二进制数据，监听节点接收并返回这些数据，拨号节点会验证这些响应数据，并检测请求与回应直接的延迟。

#### Identify



Identify协议允许节点之间相互交换信息，尤其是它们的公钥和网络地址信息。

最基本认证协议的作用是利用上表中的认证协议ID建立一个新的数据流。

当远程节点打开这个新的数据流时，它们会写入一个[protobuf message](https://github.com/libp2p/go-libp2p/blob/master/p2p/protocol/identify/pb/identify.proto)认证消息，这个消息包含了它们的一些关键信息，例如用于衍生出它们的节点ID的公钥。

更重要的，认证信息包含observedAddr域，这个域包含了节点在传入的请求中获得的多地址信息。这些帮助节点获得它们的NAT状态，它可以让他们看到其他节点观察到的他们的公共地址，并将其与自己对网络的视图进行比较。

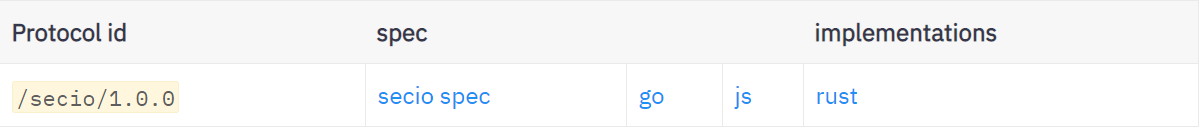
#### identify/push



与identify有点轻微变化， identify/push协议发送与Identify 相同的消息，但是它是主动发起的，而不是相应一个请求。

如果一个节点开始在一个新的地址进行监听，这个非常有用，建立一个新的中继表，或者利用标准的identify 协议从其他节点学习建立一个它自己的公共地址表。一旦建立和学习新的地址，节点可以向它知道的所有节点推送这个信息。这可以保证说有节点的路由表都是最新的并且增加其他节点发现这个新地址的可能性。

#### secio



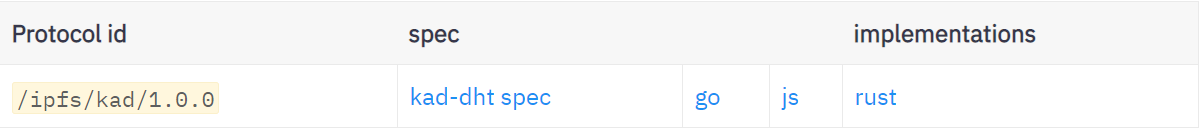
secio （secio是安全输入输出的缩写）是一个类似 TLS 1.2用于加密通讯的协议，但是不需要授权证书。因为每一个libp2p节点都有一个 [PeerId](https://docs.libp2p.io/concepts/peer-id)，PeerId是衍生于它们的公钥，一个节点的身份可以通过使用公钥来验证签名信息的方式来验证，而并不需要授权证书。

阅读[Secure Communication article](https://docs.libp2p.io/concepts/secure-comms/) 来获得更多详细信息

注意

虽然secio 是现今libp2p默认的加密协议，但是在libp2p中集成TLS 1.3的工作正在进行，并且希望一旦完成，以后成为默认协议。想要了解设计概要 可以阅读[the libp2p TLS 1.3 spec](https://github.com/libp2p/specs/tree/master/tls)。

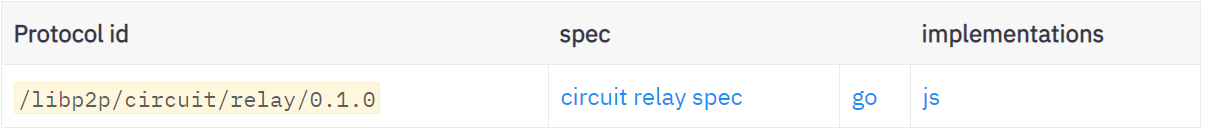
#### kad-dht



kad-dht是基于[Kademlia](https://en.wikipedia.org/wiki/Kademlia) 路由算法的分布式哈希表（ [Distributed Hash Table](https://en.wikipedia.org/wiki/Distributed_hash_table)），并进行了一些修改。

Libp2p使用DHT作为节点路由（ [peer routing](https://docs.libp2p.io/concepts/peer-routing/)）和内容路由（ [content routing](https://docs.libp2p.io/concepts/content-routing/)）功能的基础。

#### Circuit Relay



根据[Circuit Relay article](https://docs.libp2p.io/concepts/circuit-relay/)所描述的，当两个节点不能直接建立连接时，libp2p提供了通过中继节点进行打洞通讯的协议。阅读这篇文章以获得更多中继节点作用的信息，包括关于中继节点应该注意的事项和当面临棘手的NAT网络时如何自动建立中继节点连接。

# 节点标识

一个节点标识（通常被写成PeerId）是在整个p2p网络中唯一标识一个特定节点。

PeerId作为每个对等节点的唯一标识符，是对等节点与它们公钥之间的可验证链接。

## 什么是PeerId

每一个libp2p节点拥有一个私钥，它被秘密保存。每一个私钥都有一个相应的公钥，公钥用于共享给其他人。

公钥和私钥合在一起使用，可以为节点相互之间建立一个安全的信道（[secure communication](https://docs.libp2p.io/concepts/secure-comms/)）。

概念上说，PeerId是节点公钥的一个密码哈希值（ [cryptographic hash](https://en.wikipedia.org/wiki/Cryptographic_hash_function)）。这个哈希值备用来验证是否与被用来建立安全信道的公钥的所有者的PeerId一致。

文章 [PeerId spec](https://github.com/libp2p/specs/pull/100) 详细讲述了libp2p里公钥的字节格式，并且如何对这个公钥哈希产生有效的PeerId。

PeerIds 是用多哈希（[multihash](https://docs.libp2p.io/reference/glossary/" \l "multihash) ）格式编码的，在哈希值前面加入一个小的信息头，用于指示出产生该哈希使用的哈希算法。

## 如何用字符串展示Peer Ids

PeerIds 是一个多哈希值，并且用紧凑的二进制格式来表示。

多哈希值使用 [base 58](https://en.wikipedia.org/wiki/Base58)编码非常常见，与bitcoin使用相同的字母表（ [the same alphabet used by bitcoin](https://en.bitcoinwiki.org/wiki/Base58" \l "Alphabet_Base58)）。

下面是一个PeerId的例子，就是采用base58编码的多哈希值：

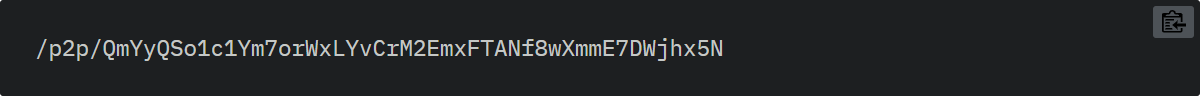
QmYyQSo1c1Ym7orWxLYvCrM2EmxFTANf8wXmmE7DWjhx5N

虽然多哈希值可以使用其他多种文本格式表示（比如16进制， base64编码等）， 但是PeerIds 通常采用base58进行编码，并且在编码字符串时并不包含前缀。

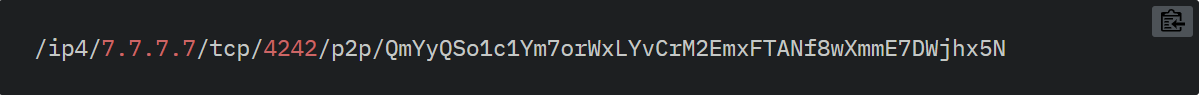
## 多地址格式中的PeerIds

PeerId 可以作为一个参数，使用/p2p地址的方式将其编码到多地址中去。

如果我的Peer Id 是 QmYyQSo1c1Ym7orWxLYvCrM2EmxFTANf8wXmmE7DWjhx5N，我的libp2p中的多地址应该是：



与其他多地址一样，一个/p2p地址可以被包含在多地址中，从而形成了一个新地址。例如，我们可以将上面的地址与传输地址/ip4/7.7.7.7/tcp/4242合在一起产生一个非常有用的地址：



如果使用 TCP/IP协议，这个地址信息为拨号特定节点提供了足够的信息。很明显，如果节点是拥有这个地址和端口的节点，它们很快能够建立连接，因为该节点不用去获得密钥对来产生地址中内嵌的PeerId。

关于更多libp2p中地址的信息，可以也读**[Addressing](https://docs.libp2p.io/concepts/addressing/)**

**注意**

Libp2p地址的多地址协议原本是以/ipfs开始的，后面会重命名为/p2p。

这两种形式是等价的，并且地址的二进制格式是一致的。哪种格式会被使用，这依赖于使用的是哪个地址代码库的版本。

## 节点信息

与节点认证相关的并且在libp2p中通用的其他数据结构是PeerInfo。

PeerInfo包含了本节点监听的多地址集合，并且这个多地址集合含有本节点的PeerId。

Libp2p应用通常拥有一个节点库或者节点书，在这个节点书中维护了一个PeerInfo的对象集合，这个集合中的数据全部来源于本节点检测到的其他节点。

这个节点库的作用就想一个电话簿，用于与其他节点拨号建立连接使用。如果一个节点在这个节点库中，我们可能不需要使用节点路由功能（ [peer routing](https://docs.libp2p.io/concepts/peer-routing/)）去发现它的地址。

如果过你能帮助改善这个教程，请参照 [refer to this issue](https://github.com/libp2p/docs/issues/12) 向我们提出建议。

# 寻址

灵活的网络需要灵活网络寻址系统。因为libp2p被设计成在多种网络中工作，所以我们需要可以在许多不同的寻址框架中工作的统一的方法。

multiaddress （通常被缩写为multiaddr）是统一的将多层次的寻址信息编码为单一的将来可证明的路径地址结构的方法。它定义了人类可读且机器优化的将通用传输协议与其他上层协议融合在一起的编码方法，它可以将多层的寻址融合在一起使用。

例如：/ip4/127.0.0.1/udp/1234编码了两个协议及其基本寻址信息。/ip4/127.0.0.1告诉我们，我们想要使用IPv4协议，并且使用 127.0.0.1这个回环地址，/udp/1234告诉我们想往1234这个端口发送UDP数据包。

进一步我们可以更多有趣的东西。例如，这个多地址/p2p/QmYyQSo1c1Ym7orWxLYvCrM2EmxFTANf8wXmmE7DWjhx5N唯一的标识了我本地的IPFS节点，它使用的是libp2p已注册的协议ID/p2p/和衍生于IPFS节点公钥的多地址哈希。

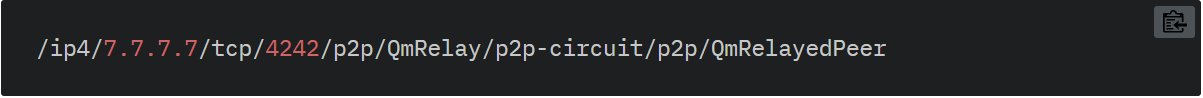
更多的节点认证和相关的公钥加密文档请阅读[Peer Identity](https://docs.libp2p.io/concepts/addressing/peer-identity/)

举例来说，按照上面说的我们有个节点ID QmYyQSo1c1Ym7orWxLYvCrM2EmxFTANf8wXmmE7DWjhx5N，我们的共有IP地址是7.7.7.7。我启动的我libp2p应用程序，在多TCP端口4242建立监听。

现在我可以用/ip4/7.7.7.7/tcp/4242/p2p/QmYyQSo1c1Ym7orWxLYvCrM2EmxFTANf8wXmmE7DWjhx5N地址向我的所有朋友发送多份备份文件了。经过融合我的本地多地址（IP和端口）和身份多地址（我的libp2p 节点ID），产生了一个新的包含每个关键信息的多地址。

现在不仅我的朋友知道在哪找到我，任何拥有这个地址的节点都可以验证这个是真正的我，或者至少它们知道这个地址的节点控制着我节点ID的相关私钥。它们也知道（这是/p2p/协议ID的优点）我很可能支持libp2p的通用交互方式，例如连接的建立和通过谈判告知我支持什么的通讯协议。这很不错。

这可以扩展为考虑多个寻址和抽象层。例如，中继节点地址结合传输地址和节点身份信息而产生一个新的地址，这个新地址被称作relay circuit。



## 更多信息

更多详细信息请阅读 [multiaddr spec](https://github.com/multiformats/multiaddr)，这里有许多实现的连接。

# 安全注意事项

Libp2p使得在两个节点之间建立加密授权的信道非常简单，但是想要建立一个健壮的p2p系统，还是有很多其他的安全性问题需要考虑。

这里描写的许多问题并没有什么完美的解决方案，现存的解决方案和减缓策略可能与其他领域进行了权衡和妥协。作为通用目的性框架，libp2p尝试为开发者提供一些工具来解决这些问题，而不是用一些任意的方法来处理安全问题，因为基于libp2p的应用程序不一定都会支持它们。

另一个需要考虑的方面是，一种特定类型的攻击在理论上是可行的，但这并不意味着它是实际的、明智的、有价值的或有效的。为了评估一个理论攻击向量的实际可利用性，考虑攻击者为了合理地成功攻击而必须花费的资源的数量、类和成本。

## 身份和信任

每个libp2p节点都是通过它的节点ID来唯一识别，这个节点ID衍生于私钥。我们可以使用节点ID和相应的秘钥来验证远程节点，这样我们就可以确定我们正在和正确的节点通讯，而不是一个冒名顶替者。

但是，验证只是处理安全问题的一个方面。许多系统也要求授权，或者可以决定谁可以做什么。

Libp2p不会提供开箱即可用的授权框架，这是因为在p2p系统中，这种需求是非常广泛而且是不同的。例如，一些网络可能完全不需要授权，只需要简单的接收来自其它节点的请求，然而其他的一些网络可能需要根据不同层级上的角色，可请求的资源和服务等显示的授予合适的权限。

为了在libp2p上设计一个授权系统，你可以参考在节点ID上进行的验证方法，在节点ID与权限之间建立关系，就像传统的授权框架一样，让节点ID当做用户名，私钥当做密码。你的协议处理句柄函数可以拒绝不可信的节点请求。

当然，也可以在libp2p上建立不依赖于节点ID的其他授权系统。例如，你可以通过使用用户名密码的方式让libp2p节点对所有人类操作者都可用。这可以通过设计一个协议来完成，这个协议接收用户名和密码，然后返回一个相应来确定授权是否有效。那些可以暴露敏感资源的协议需要这个相应来决定这些资源是否可以被访问。

那些被设计成完全非中心化的系统通常是开放的，允许所有的节点参与其核心功能。然而，这样的系统可以在维护某种信誉的系统中获益，可以识别缺陷或恶意的参与者，并且阻止或忽略他们。例如，每一个节点都可以依据设计的协议，根据远程节点可用程度和行为正确度，给远程节点打分，依据这个分数来考虑是否处理来自远程节点的请求。

一个完全去中心化的信誉管理系统的所有节点是通过相互合作来评估每个节点，这超出了libp2p的范围。然而，许多libp2p的核心开发者和社区成员非常乐意在这个领域进行研究和开发，我们非常乐意看到你在libp2p论坛上提出自己的想法。

## 具有滥用潜力的合作系统

Libp2p中内建的一些非常有用的协议是通过合作利用网络中的其他节点完成对每个人都有利的任务。例如，存储在Kad-DHT的数据会被复制到与数据关联键最近的多个节点中，不管这些节点是否对这些数据有兴趣。

相互合作系统天生易于被作恶节点滥用，虽然我们研究了很多方法用于限制这些攻击的影响，但是现今在libp2p中仍然存在这些问题。

### Kad-DHT

Kad-DHT是分布式哈希表（[distributed hash table](https://docs.libp2p.io/concepts/reference/glossary/" \l "dht)），它为所有的参与者提供了共享的键值对存储系统。除了键值对查询，DHT还是libp2p节点路由（ [peer routing](https://docs.libp2p.io/concepts/peer-routing/)）和内容路由（ [content routing](https://docs.libp2p.io/concepts/content-routing/) ）的默认实现，因此在其他节点发现和网络服务中扮演者重要角色。

#### Sybil Attacks

DHTs和p2p系统通常容易受到一类攻击叫做女巫攻击（ [Sybil attacks](https://en.wikipedia.org/wiki/Sybil_attack)），这个攻击是一个操作者产生大量的不同的标识符的节点，并将这些节点写入到DHT表中，使自己在整个网络中占有更高的比重或话语权。

一个DHT查询在完成之前可能需要路由多个节点，因此每一次都有机会通过返回不正确的数据或根本不返回数据来修改查询返回的结果。通过控制大量的节点（与网络规模成正比），坏的参与者就能增加在查询过程中碰到这些节点的可能性。为了得到特定键值的目标，他们可以依据DHT的距离算法生成更多与目标键值更近的ID，从而改变查询路径，进而改善查询到目标的机会。

应用程序可以通过对存储在DHT中数据进行签名来防止别人对数据进行修改，或者可以使用内容寻址，内容寻址是使用存储值的加密哈希值作为键值的，就像 [IPFS](https://ipfs.io/)里使用的方式。这些策略可以让你检测到数据是否被篡改，但是不能阻止发起时就已经被篡改的情况，也不能阻止作恶节点假装这些数据不存在从而使得完全忽略这些数据的情况。

与女巫攻击非常相似，Eclipse 攻击也是通过控制大量的节点，但是有着轻微不同的目的。不同于在网络中修改数据，Eclipse 攻击的目标是使特定节点在网络中看到的网络视图是失真的，通常是用来阻止他们连接合法的节点。这种攻击的执行资源相当密集，需要大量的恶意节点才能完全有效。

Eclipse和Sybil的攻击很难防御，因为这种攻击可能生成了无限量的有效的节点ID。许多实践上用于阻止女巫攻击的方法是使生成节点ID在某种程度上变的昂贵，例如，通过与现实成本相关的工作量证明或者通过“铸造”并从一个中央可信机构签名id等。虽然这些减缓攻击的的方法超出了libp2p的范围，但是在应用层级可以使用他们，从而使得女巫攻击变的非常困难或极其昂贵。

我们当前正计划实现并行查询多个不连贯的查询路径策略（路径中不共享任何相同的中间节点），做这项工作是因为受到了S/Kademlia论文和paper-s-kademlia的启发。这个将极大的增加查询到诚实节点的几率，即使一些节点返回不诚实的路由信息。

### 发布、订阅

Libp2p的发布订阅协议（[publish/subscribe protocol](https://docs.libp2p.io/concepts/publish-subscribe/)）可以让节点向其他节点广播确定主题的消息。

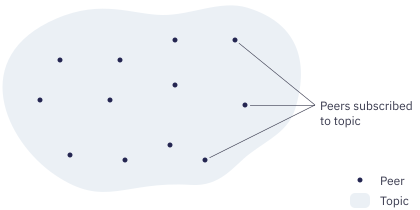
默认情况下，gossipsub 的实现要求用作者的私钥签名发送的消息，并且在接收消息或再次广播消息时需要一个有效的签名。这些阻止消息在传播过程中被修改，并且允许接受者对发送者进行验证。

然而，作为一个合作型协议，节点可能干扰消息按照消息路由算法在网络中传播。

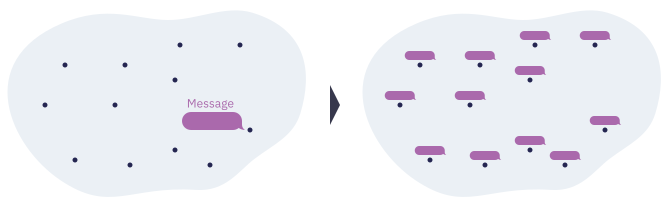
我们现在正在进行积极的研究一种方法用于缓解恶意节点在gossipsub路由算法上的影响，尤其关注于防止Sybil攻击。我们希望设计一个更健壮和防攻击的发布订阅协议。但是不可能阻止所有种类的来源于确定作恶节点的可能攻击。

# 发布和订阅

发布和订阅系统是指节点聚合它们感兴趣的主题信息。节点对某个主题感兴趣被称作订阅那个主题。



节点可以发送这个主题信息。信息会被发送到所有订阅了这个主题的节点。



使用发布和订阅的示例：

聊天室：每一个聊天室就是一个发布订阅主题，在这个聊天室中，一个用户发送消息，聊天室中的其他所有用户接收消息。

文件共享：每个发布订阅主题代表一个文件可以被下载。上传者和下载者广播关于这个主题他们都是有哪些文件并且包括这些文件的下载地址。这些下载过程可能发生在发布订阅系统之外。

## 设计目标

在一个P2P发布订阅系统中，所有的节点都将利用整个网络参与发送消息。为了应对不同的需求，P2P发布订阅系统存在几种不同的设计。希望包含的特征有：

可靠性：所有的消息都能被发送到所有订阅这个主题的节点。

速度：消息能够很快被送达。

有效率：在复制发送消息时，不会造成网络洪流。

伸缩性：节点可以自由的加入和离开，不会造成系统瘫痪。这没有失败的中心节点。

可扩展：主题可以有大量的订阅者，可以处理大吞吐量的消息。

简约性：这个系统易于理解和实现。每一个节点只需要记录很少的状态。  
libp2p当前使用的设计叫做**gossipsub**。它被叫这个名字是因为，节点之间会相互广播它们看到的消息，并利用这些消息来维护发送消息的网络。

## 发现

一个节点能够订阅某个主题之前，它首先要发现其他节点并且与其他节点建立连接。发布订阅系统自己并没有发现其他节点的功能。它需要一个应用程序代替它发现其他节点，这个过程叫做外围节点发现。

发现节点的潜在方法：

分布式哈希表

本地网络广播

与存在的节点交换节点列表

集中跟踪器和集合点

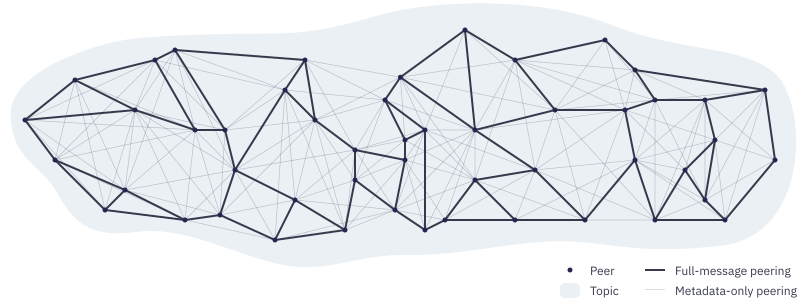
引导节点列表

例如：BitTorrent 应用程序在下载文件的过程中，上面提到的大部分方面都已经被用到了。通过重用在BitTorrent应用程序正常运行时发现的对等点，这个应用程序可以建立一个健壮的发布订阅网络。

被发现的节点会被询问是否支持发布订阅协议，如果支持，就会被加入到发布订阅网络中。

## 节点类型

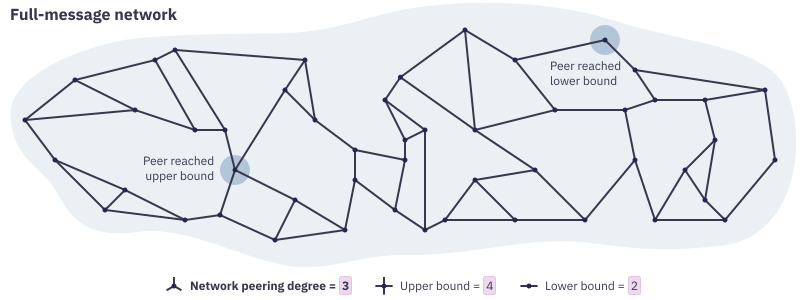
在gossipsub中，节点之间是通过 **full-message** 或者 **metadata-only**的形式进行相互连接的。这整个网络的结构是有这两种网络构建的。



### Full-message

Full-message互联网络是用于传输完整的消息内容。这个网络是一个稀疏的连接网络，每个节点与少数的节点相互连接。在[gossipsub specification](https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md)中描述的稀疏网络被称作mesh ，这个稀疏网络中的节点被称作mesh 成员。

限制Full-message互联的成员数量是非常有用的，这是因为可以控制整个网络的流量。每个节点只向其他少数几个节点发送消息，而不是像所有节点发送。每一个节点都有它们想连接节点的目标数量。在这个例子中，每一个节点的理想连接数是3个其他节点，但正常会是2-4个连接。



注意：

在这整个篇幅中，用紫色高亮显示的连接数可以被开发者进行配置的。

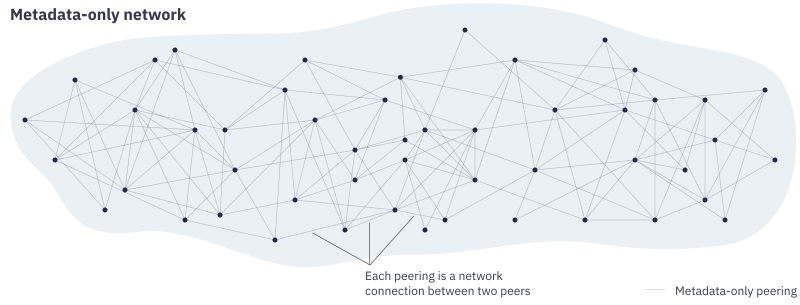
互联的度（也被称作网络度）是与网络的速度、可靠性、伸缩性和效率相互制约的。更高的互联度可以使消息传播的更快速，能够更好的到达所有订阅节点并且能够减少由于节点离开造成的网络瘫痪。然而，一个很高度的网络互联也可以造成额外冗余的消息在网络中传播，这会增加参与者对网络带宽的需求。

在libp2p默认的实现中，理想的网络互联度是6，当然，在4-12的范围中都是可以接受的。

### Metadata-only

除了full-message互联的稀疏连接网络，还有metadata-only互联的浓密的连接网络。它是由非full-message互联网络中的节点相互连接组成的所有网络构成的。

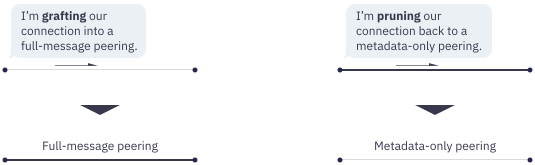
metadata-only互联的网络只是用来共享一些可用的消息，这些消息是用来帮助维护full-message互联网络的。



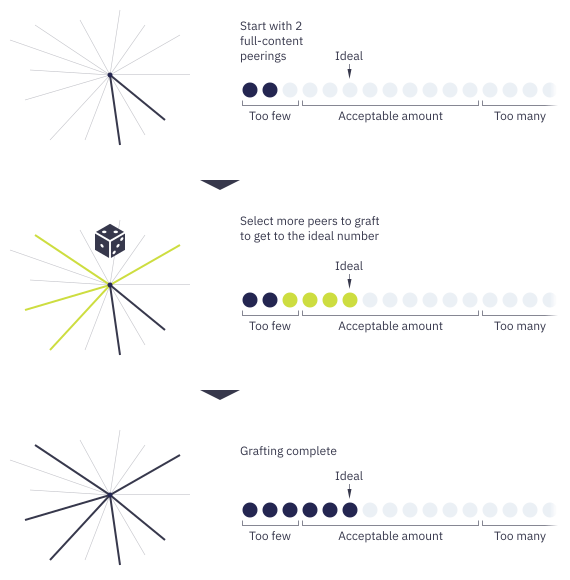
## 嫁接和修剪

互联是双向的，这就意味着任何两个相连接的节点，它们的连接有可能是full-message，也有可能是metadata-only。

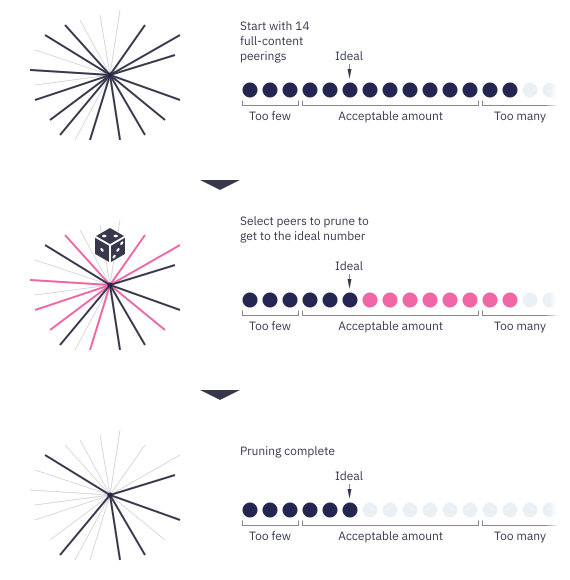
每个节点可以用通过通知其他节点的方式改变连接类型。**Grafting** 是将metadata-only连接转换为full-message连接的过程。**Pruning** 是相反的过程，将full-message连接转化为metadata-only连接。



当一个节点拥有太少的full-message连接，它就会随机的将一些metadata-only连接转化为full-message连接：



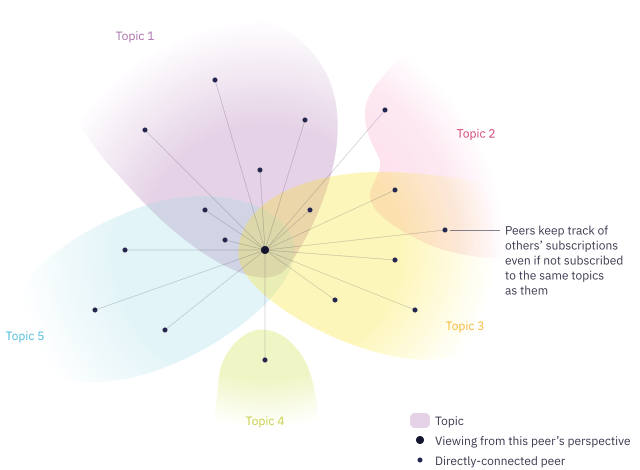
相反地，当一个节点拥有太多的full-message连接，它就会随机将一些full-message连接转化为metadata-only连接：



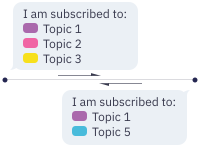
在libp2p实现中，每一个节点每一秒都会做一系列检查。这些检测被称作心跳。Grafting 和pruning就发生在这个时候。

## 订阅和取消订阅

节点会持续跟踪它们直接连接的节点都是订阅了哪些主题。利用这些消息，每个节点都可以建立一个它们周围主题的关系图和每一个主题都是有哪些节点订阅。



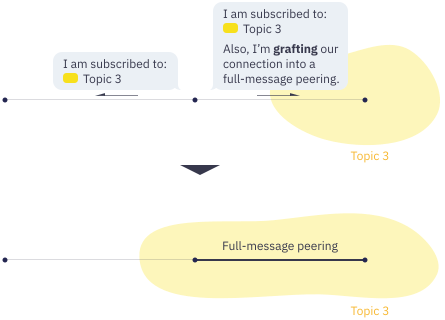
同过发送订阅或取消订阅消息来跟踪订阅情况。当在两个节点之间建立一个新连接，它们就会首先相互发送它们订阅的主题列表。



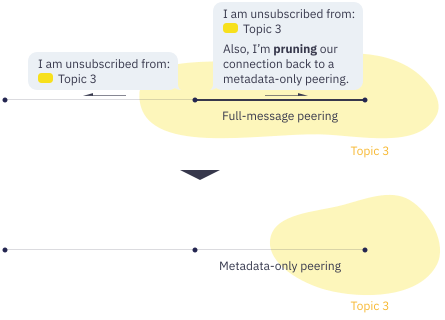
然后随着时间的推移，无论什么时候其中一个节点发生订阅或取消订阅一个主题，它都会向它连接的所有节点发送订阅或取消订阅的消息，不管这些节点是否订阅了这个主题。



订阅与取消订阅消息是与graft 和prune 消息同时发送的。当一个节点订阅一个主题，它会为这个主题挑选一些节点的连接，是这些连接变成 full-message连接，修改连接类型的graft 消息会与订阅消息同时发送。

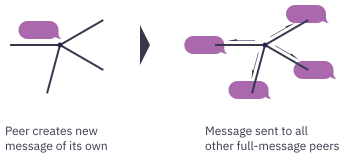


当一个节点取消订阅一个主题，当它发送取消订阅消息的时候，同时也通知相关联的full-message连接节点，它们的连接已经被转化为metadata-only连接。



## 发送消息

当一个节点发布一条消息，它会将复制的信息发送到所有与它是full-message连接的节点。



类似的，当一个节点收到了来自其他节点的信息，它首先会存储这条消息，然后复制这条消息并向所有与它是full-message连接的节点转发这条消息。



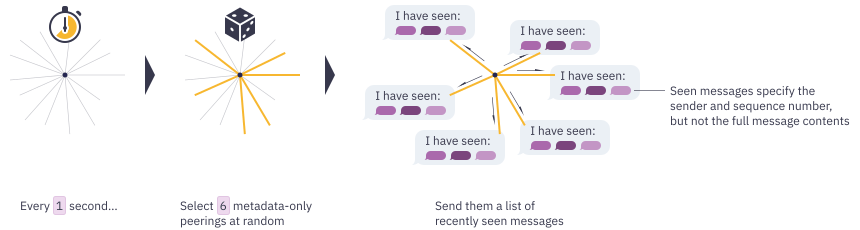
在 [gossipsub specification](https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md" \l "controlling-the-flood)文章提到，对等节点也被称为路由器，这是因为它们在网络中有维护路由信息的功能。

节点维护一个列表，用于保存它们最近收到的消息。这就可以让节点只在第一次收到该信息的时候执行相关操作，而忽略它们再次收到的已经收到过的消息。

节点也有可能选择验证它们收到的消息内容。什么样的消息有效或者无效，这依赖于应用程序本身。例如，一个聊天程序可能强制发送的消息少于100字节。如果这个应用程序告诉libp2p这个消息是无效的，这个消息就会被删除，并且这个消息也不会在网络中复制传播。

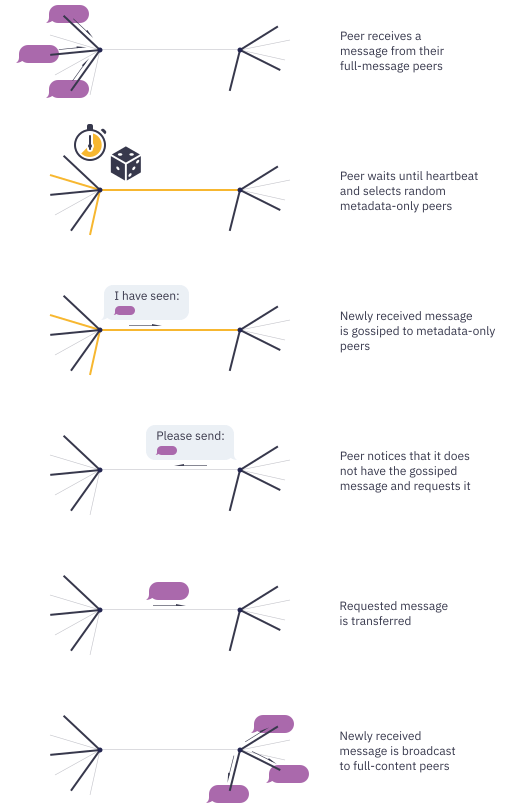
## Gossip

节点会传播它们最新收到的消息。每一秒节点都会随机选择6个metadata-only 连接的节点，然后向这6个节点发送它最近收到的消息列表。



传播给了节点通知其它节点自己在full-message网络上缺失某些信息的机会。如果节点通知说自己多次缺失某条消息，它就可以与拥有这条消息的节点建立一个新的 full-message连接。

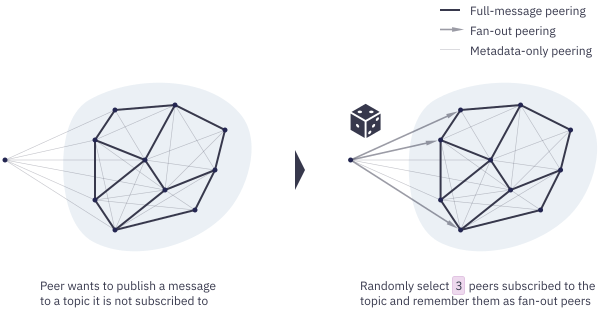
这有一个例子展示了一个特定的消息是如何在metadata-only网络中被请求的。



在 [gossipsub specification](https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md" \l "control-messages)文章中，传播最近收到消息被称作IHAVE消息，请求一个特定消息被称作IWANT消息。

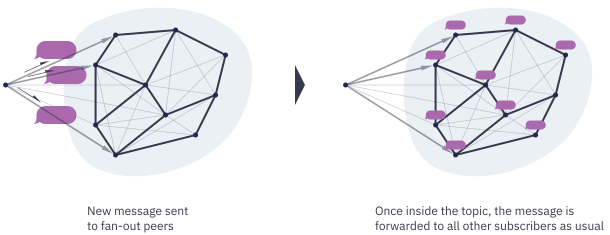
## Fan-out

节点可以向它们没有订阅的主题发布消息，这里有一些特殊规则用于确保这消息能够被可靠的发送。节点第一次想往它没有订阅的主题发布消息，它会随机挑选6个订阅了该主题的节点（最少3个节点），并把它们记录为该主题的**fan-out**节点。



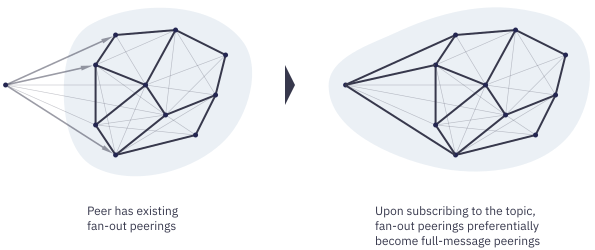
不像其他的节点互联，fan-out连接是非双向的。它们总是从非订阅某主题的节点指向已订阅该主题的节点。订阅相关主题的节点不会被告知它们已经被选中，它们仍然像看待其他metadata-only连接一样看待此连接。

发送者每次想发送一条消息，它们把消息发送到fan-out的节点，收到这这条消息后，fan-out的节点会将此消息在相关主题的网络中传播：

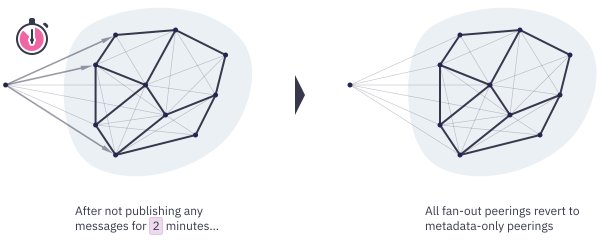


如果发送者已经发送了消息，但是它们注意到它们的fan-out节点在上次发送后离开了，它们会再次随机的挑选另外的fan-out节点，使其数目达到6个上线。

当一个节点订阅了主题的时候，如果它已经有了相关的fan-out节点连接，它首先会将它们的连接转化为full-message连接：



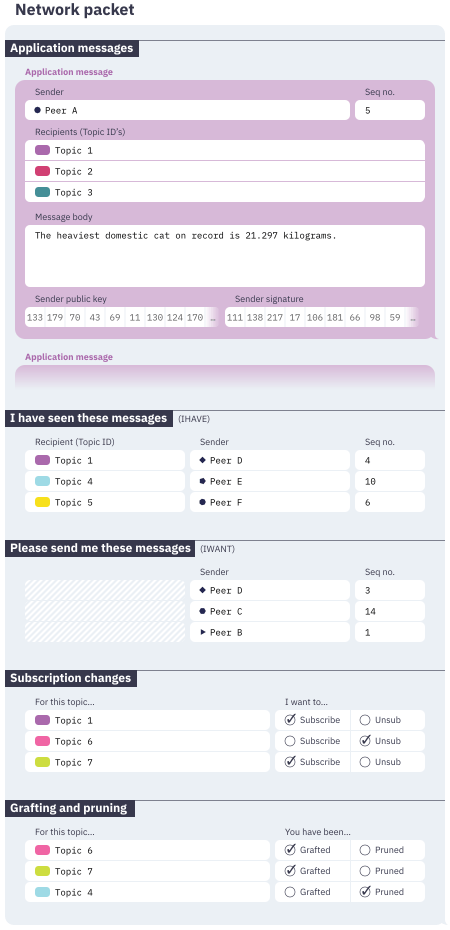
如果2分钟后没有向相关主题发送消息，所有的fan-out节点都会被忘记：



## 网络数据包

节点之间实际相互发送的数据包包含了这个指南所讲到的各种数据类型（应用数据，have/want消息，订阅和取消订阅消息，graft/prune消息）。这个结构允许这些不同的需求在一个网络数据包中进行分批和发送。

这里有一个整体的网络数据包结构的图形展示：



阅读说明书（[specification](https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md" \l "protobuf)）来了解用于编码网络数据包的 [Protocol Buffers](https://developers.google.com/protocol-buffers)框架。

## 状态

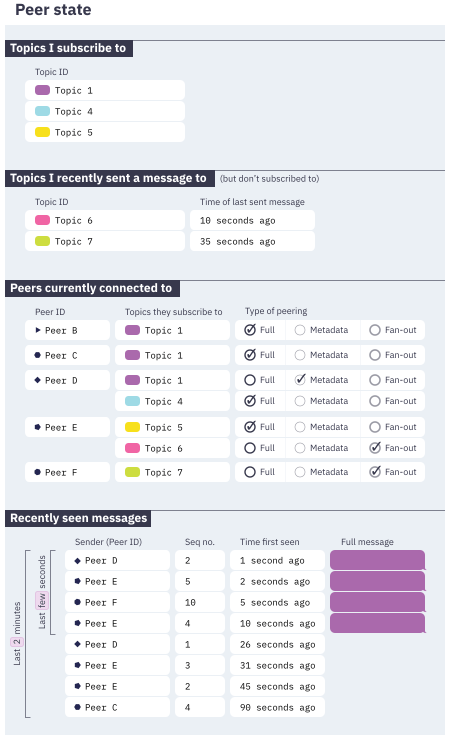
为了参与订阅和取消订阅的网络，这里总结了节点必须记住的状态。

订阅情况：订阅的主题列表

Fan-out主题：这些是最近没有被订阅的主题的消息。为每一个主题，最新一次发送到相应主题的消息的时间会被记录。

节点当前的连接列表：每一个节点连接，它们的状态包括它们订阅的主题，每一个主题的互联是 full-content, metadata-only或 fan-out。

最新收到的消息：这是最新消息的缓存。这是用来检测和忽略重复传输的消息。每个消息的状态包括谁发送的和发送的序列号，这足以唯一识别任何消息。对于最新消息，完整的消息内容会被保存并可以将这些消息发送给任何需要它们的节点。



## 更多信息

更多详细信息和影响gossipsub设计的其他发布订阅设计的讨论，请阅读[gossipsub specification](https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md)。

更多实现信息，请看 [go-libp2p-pubsub](https://github.com/libp2p/go-libp2p-pubsub)源代码中的 [gossipsub.go](https://github.com/libp2p/go-libp2p-pubsub/blob/master/gossipsub.go)文件，它是libp2p中gossipsub 的权威实现。

# 多路复用流

多路复用流（通常被缩写为“stream muxing”）允许多个相互独立的逻辑流共享一个底层的传输介质。

Libp2p应用程序通常可能在节点之间打开多个独立的通讯流，并且可能与一个远程节点同时打开几个并发流。多路复用流允许我们摊销在我们的交互生命周期中与其他节点建立新传输连接的开销。我们也只需要处理一次NAT穿透（[NAT traversal](https://docs.libp2p.io/concepts/nat/)）问题，使得我们可以打开与我们需求一样多的数据传输流，这是因为它们共享相同的底层传输连接。

多路复用并不只是存在libp2p中。大部分通讯网络都涉及到一些种类的多路复用，这是因为传输媒介通常是稀缺的，需要许多参与者共享。例如，TCP/IP技术栈会在低层网络上复用许多TCP流，它们是通过端口号来区分数据流的。Libp2p流复用器依赖于下层的传输技术栈，许多数据流都是依靠单一的TCP端口或其他的原始的传输连接。

Libp2p为流复用器的多个可用实现提供统一的接口。应用程序能够支持多个复用器，当远程节点不支持首选的复用器时，它允许你退回到广泛支持的多路复用器。

## Libp2p 技术栈中适合使用它的地方

Libp2p的多路复用发生在应用层，那也就意味着它不会被操作系统的网络技术栈所提供。然而，写libp2p应用程序的开发者很少需要直接与流多路复用器直接交互，除了在初始化配置时，控制哪些模块启用。

### Switch / swarm

Libp2p在一个叫做switch（有时候也别称作swarm，这依赖于具体实现）组件中维护了一些状态，这些状态包括节点信息和存在的连接。Switch提供了拨号和监听接口，这个接口抽象了连接所使用的哪个流复用器的细节。

当配置libp2p的时候，应用程序将启用流多路复用模块，switch将会在拨号和监听连接时使用这些模块。如果远程节点支持任何相同的流多路复用实现，switch就会在建立连接的时候选择并使用它。如果你拨号一个switch已经打开连接的节点，这个新的流就会自动地成为现存连接上的多路复用流。

达成使用哪个流复用器的协议发生在连接建立过程的早期。节点使用 [protocol negotiation](https://docs.libp2p.io/concepts/protocols/" \l "protocol-negotiation)来达成统一支持的多路复用器，这个过程会将原始的传输连接转变为可以打开新流的多连接。

## 接口和实现

### 接口

流多路复用接口（[stream multiplexing interface](https://github.com/libp2p/interface-stream-muxer)）定义了一个流多路复用模块是如何在一个连接上使用和一个多路复用连接都是支持哪些操作。

### 实现

在libp2p中有多个流多路复用模块可用。但是请注意并不是所有的流多路复用器都被每一种语言实现的libp2p版本所支持。

#### mplex

mplex 是为libp2p开发的协议。 [spec](https://github.com/libp2p/specs/tree/master/mplex)为多路复用定义了简单的协议，它得到了多个语言版本的libp2p的支持。

* Go: [go-mplex](https://github.com/libp2p/go-mplex)
* Javascript: [js-mplex](https://github.com/libp2p/js-libp2p-mplex)
* Rust: [rust-libp2p mplex module](https://github.com/libp2p/rust-libp2p/tree/master/muxers/mplex)

#### yamux

[yamux](https://github.com/hashicorp/yamux)是被[Hashicorp](https://www.hashicorp.com/)设计的一款多路复用协议。

yamux 提供了比mplex更为复杂的流控制功能，并且可以在单一的连接上扩展成千上万个多路复用流。

Yamux当前支持GO和Rust:

* Go: [go-smux-yamux](https://github.com/whyrusleeping/go-smux-yamux)
* Rust: [rust-libp2p yamux module](https://github.com/libp2p/rust-libp2p/tree/master/muxers/yamux).

#### quic

[QUIC](https://en.wikipedia.org/wiki/QUIC)是一个传输协议，它本身包含流多路复用器。当libp2p使用quic 传输协议时，会自动使用quic自带的多路复用器。

Quic当前只支持GO，[go-libp2p-quic-transport](https://github.com/libp2p/go-libp2p-quic-transport)。

#### spdy

[SPDY](https://en.wikipedia.org/wiki/SPDY) 是谷歌开发的协议，是HTTP/2的前身。SPDY 实现了一个流多路复用器，并且被libp2p的一些实现所支持。

* Go: [go-smux-spdystream](https://github.com/whyrusleeping/go-smux-spdystream)
* Javascript: [js-libp2p-spdy](https://github.com/libp2p/js-libp2p-spdy)

#### muxado

[muxado](https://github.com/inconshreveable/muxado)是一个GO实现的流多路复用库，它被go-libp2p所支持， [go-smux-muxado](https://github.com/whyrusleeping/go-smux-muxado)。