前言

这是一份介绍计算机网络中一系列常见概念的读物，注意，这份文档里面并不是我的另外那本《Beej的网络编程指南--C语言版》[flbg[*Beej的 的网络编程指南*|bgnet]]。这一系列文章旨在让读者对各种繁杂的网络术语有所了解，当然，为了加深理解，文章中会有一些使用python进行的编程练习和例子。

那么可能有人会问了，那这是 *Beej的的网络编程指南--Python版* 吗？嗯，是也不是。因为那本C语言的网络指南更多的是关于Unix的底层的网络API是怎么运作的，但是这本书，更多的关注点是在计算机网络概念本身，只不过用python作为一个载体。

如果这依然让你感到困惑，也许你可以直接跳到下面的 *观众* 部分试试找找答案。

受众

您是否对网络完全陌生，并且对ISO-OSI、TCP/IP、端口、以太网、局域网等所有这些术语感到困惑?也许您想用Python编写一些具有网络功能的代码?那么恭喜啦，本书是你的不二之选！

但是还是要事先提醒下:本指南假定您已经掌握了一些Python编程知识。

官方网址

以下是本指南的官方网址 [fl[https://beej.us/guide/bgnet0/|https://beej.us/guide/bgnet0/]].

邮件规则

如果你有问题，欢迎给我发邮件，我很乐意通过邮件收到您的问题。不过因为我平时确实很忙，有些问题我可能没法想出一个合适的答案。所以在这种情况下，我通常会删除这条邮件。这不是针对个人的，我只是永远没有时间给你所需要的合适的答案。

一般来说，问题的描述越复杂，我回答的可能性就越小。如果您可以在发送问题之前精简你的问题，并如能附上包含任何相关信息(如平台、编译器、报错信息以及您认为可能帮助我定位问题的任何其他信息)，那么将更有助于我回答您的问题。

如果你没有得到回复，那请尝试了解更深入一些，看看能不能寻找到答案。如果仍然难以捉摸，那就把你寻求答案过程中了解到的资料一并发给我，希望我踩能帮你的步伐回答出你的问题。

既然我已经赘述了很多关于写信和不写信给我的信息，我只是想让你知道，我_非常感激_这些年来对本指南所得到的所有赞扬。这真的让我深受鼓舞，特别是听到它对于您有所帮助!`:-)`,谢谢!

备份

欢迎备份本指南，无论是公开的还是您私享。如果您做了一个公开的备份并且想让我给您在主页上加一个链接，请给我写份邮件说明`beej@beej.us`。

给翻译者的话

如果您想讲本文翻译成其它语言，请给我写份邮件`beej@beej.us`，我会在首页给您的翻译加上一个入口。随信请加上您的名字和联系信息。

注意，在您的翻译里，请加入下面的版权声明和发布声明章节。(这一部分不翻译，因为要作为版权说明完整的放在翻译里面)

Copyright and Distribution

Beej's Guide to Networking Concepts is Copyright © 2023 Brian "Beej Jorgensen" Hall.

With specific exceptions for source code and translations, below, this work is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License. To view a copy of this license, visit `https://creativecommons.org/licenses/by-nc-nd/3.0/` or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

One specific exception to the "No Derivative Works" portion of the license is as follows: this guide may be freely translated into any language, provided the translation is accurate, and the guide is reprinted in its entirety. The same license restrictions apply to the translation as to the original guide. The translation may also include the name and contact information for the translator.

The programming source code presented in this document is hereby granted to the public domain, and is completely free of any license restriction.

Educators are freely encouraged to recommend or supply copies of this guide to their students.

Contact `beej@beej.us` for more information.

题献

在写本指南的过程中，我觉得最难的有如下几点：

- 理解资料中的每分细节这样我才能准确的解释给读者
- 找到深入浅出解释的方法，这是一个需要不断迭代和自我反馈的过程
- 要求自己把自己当作为所谓的权威，而实际上我只是一个试图理解这一切的普通学习者，正如你们一样
- 世间精彩纷呈，我得要求自己专注于此

在完成本指南过程中，我获得的帮助颇多，再次我想对于这些帮助一一表示感谢。

- 每一个在互联网上决定以某种形式分享他们的知识的人。免费分享有教育意义的信息使互联网成为一个伟大的地方。
- 每一个对于这篇指南给予指正和提出PR指正本文中手误的人。

再次表示深深的感谢！❤

网络概述

对于计算机网络，最重要的思想就是，我们将在Internet或LAN上的两台(或多台)计算机之间交换任意字节的数据。

为了达到此目的，我们需要：

- 甄别源和目标计算机的方案。
- 维护数据完整性的方案。
- 在数十亿的计算机中将数据从一台计算机路由到另一台计算机的方案。

所以，我们需要联合硬件和软件共同协作来到上述目的。让我们从两种最基本的网络交流方式开始旅程。

电路交换网络

不必惊慌！目前我们的英特网已经不使用这种网络了（至少就我目前所知的），所以你大可放心，在本章你所读到的知识，你基本都不会再次用上。

对于这种类型的网络，想象一个老派的电话接线员(比如一个人工接线员)。在你还不能直接拨号之前，打一个电话会是这样的:

- 拿起听筒，转动曲柄，产生一个电子信号，在电话线的另一端是在某个地方的电话局，这时他们的电话铃声开始响起。
- 电话那端的接线员听到铃声后会拿起电话，并且和你说，“你好，我是接线员。”
- 这时，你会用你的声音告诉他你想要打的电话号码是什么。
- 接下来，接线员会将跳线插入他们面前的配电盘，在你的电话线路和你想打电话的人的电话线路之间物理地完成电路连接。至此，你的电话就和对方的电话直接联通了。

这种技术的可扩展性很低，假设你必须打长途电话，那就得支付额外的费用，因为在这种情况下，由于线路长度是有限的，所以接线员之前需要互相联系才能完成这种长期距离的电路连接。

随着时代的发展，人们发现可以用电磁继电器和交换机来取代人工操作员，我们可以通过发送用心编码过的信号来控制这些电路信号。这些信号可以是电脉冲(由旧的旋转拨号电话发送)，也可以是仍然能够被识别的“触摸”音。

不过，我们仍然还有这些问题需要解决:

- 每次通话都需要专用的电路。
- 即使通话者，静静地坐在那里，什么也不说，电话电路仍然会被占用着，其他人完全不能使用。
- 一条线路不能在同时被多人使用。
- 线路数量有物理上限，所能拨出电话数量也是有限的。

正因如此，因特网采用了完全不同的尝试。

分组报文交换网络

在 *分组交换* 网络中，欲发送的数据会被分成一个个不同大小的数据包。比如，如果你想发送83,234 字节的数据，那么基本上会分成50或者60个数据包。接下来，在带宽允许的情况下，上述的每个数据包会通过线路单独发送。遐想一下，来自北美各地的计算机的小数据包到达大西洋边缘的路由器，然后通过数千英里长的

海底电缆将它们逐个发送到欧洲。一旦数据包到达目标计算机，对端计算机就会从一个个数据包中重建出原始数据。

这个过程很像写一封普通的信件，写完后把它投递在邮局。邮局将这封邮件和一堆其他邮件一起被送到一辆卡车上，但很明显，这车邮件不会和你的被送到同一个最终地址。而邮局会根据合适的物流路线和递送规则将信件投递到最终的目的地。也许你的信和一堆其他的信一起上了飞往国家另一边的飞机，并且当飞机到达时，这些信件可能会分开，一些向北，一些向南。很明显，邮局不可能仅仅一整架飞机去运送一封邮件。而这里的邮件就很像网络中的数据包，他们被从一个地方转发到另一个地方。

与此类似，互联网上的数据包也是从一台计算机转发到另一台计算机，这之间的线路被计算机们共享，直到它们最终到达它们要去的地方。这些设计为计算机网络带来了一些巨大的好处:

- 在每一对通信计算机之间不需要都有一对一的专用电路(而且在当今的计算机规模中，这已经是物理上不可能的事情了)。
- 多台计算机都可以使用同一条线路在“同一”时间发送数据。(这些数据包实际上是一次发送一个，但它们是交错的，所以看起来是同时发送的。)
- 物理电线可以被充分利用，如果有计算机想要使用，线路上就不会有死寂。在一条线路上，一台计算机的沉默就是另一台计算机传输信息的机会。

客户端/服务器架构

对于这个概念，你一定从使用网站中了解过，通常人们会说网站服务器。

服务 是指一种程序，这种程序 *监听* 可能会接入的连接，*接受* 它们发来的请求，并且回复一些消息，这个接收回复消息的程序就叫做 *客户端*。当然，取决于服务器的业务逻辑，服务器和客户端之间的实际的交互细节比上述的要复杂得多。

本质上，客户端和服务器都是一种网络程序，它们之间的最大区别是，服务器程序是被动等待客户端主动调用来连接的程序。通常，一个服务器可以被很多客户端同时连接。一个服务器集群中可能存在许多服务器，以服务许多、许多、许多客户端。你可以想象有多少客户端连接着谷歌的服务器。

操作系统，计算机网络与套接字

最本质的网络是一种计算机硬件系统，而通过操作系可以控制对硬件的访问。因此，如果你想编写使用网络的软件，必须借助操作系统的帮助。

一直以来，这个操作都是通过在Unix上首创的一个叫做 *套接字* 的API来实现的。Unix套接字API是一个通用的概念，在它繁多的功能中，其中之一就是可以为用户提供在Internet中读写数据的方法。随着时间的发展，在其它的语言和操作系统中也添加了类似的网络通信功能的API，并且赋予了它们不同的名字。但是作为对于初创者的致敬，即使和原创已经不尽相同，但是这其中大部分API还是使用”套接字“这个名字。

如果你想体验最原始纯正的套接字API，你可以使用Unix里面的C语言的socket API。

协议

你知道客户端和服务器之间具体的对话内容是什么样的么？它详细规定了什么字节何时从谁发送到谁。用户不能将过时的数据发送到网站服务器——这种情况下，服务器用一种确定的方法丢弃旧数据。就像你不能写一封信，然后把它用铝箔纸包起来，并且不写地址，但是你还指望邮局把它送到你的收件人那里。这肯定做不到，因为这违反了邮局的 *规定*。

在生活中，信息发送方和信息接收方必须使用相同的协议才能进行正确的通信。比如：

> “感谢您致电披萨餐厅。我能帮你吗?”
>
> “我想要一个加薯条的披萨”
>
> 打电话定披萨的人违反了披萨店的订餐协议。

在计算机网络中，有许许多多的协议，在本文后续中，我们会详细介绍其中的一部分。这些协议都是为了解决用户不同的问题。如果你想要在你自己编写的定制程序间传输数据，你也需要定义你自己的通信协议。

这里有一些你可能听过的协议名称：

- TCP - 用来可靠的传输数据。
- UDP - 用来快速的传输数据，但是不保证可靠性。
- IP - 用来规定如何在网络的计算机之间路由数据包。
- HTTP - 用来访问网站或者接受其他网页请求。
- Ethernet - 用来定义如何在LAN链路上传输数据。

下面我们还会看到，这些协议都工作在计算机网络软件的不同层级中。

网络层次和抽象

下面简要介绍一下数据在网络上传输时会发生什么，在接下来的内容中会围绕着这个点更详细的阐述。

1. 如果一个用户的程序说：“我想个给那边的网站服务器发送'GET / HTTP/1.1'“（在网络中，服务器是以 *IP地址* + *端口* 来标示的）。
2. 操作系统收到用户程序发送出来的数据并且将它们放在一个 _消息头_中（简单的说，就是会在用户数据之前添加一些额外数据）并且提供一些错误检测（还可能有排序）的信息。通信协议会具体定义消息头的结构，比如，TCP或者UDP。
3. 操作系统会接受上一步的所有内容，并且再次把这个整体放入一个带有路由新的的消息头中。IP协议定义了如何构造这一步的数据头。
4. 接下来，操作系统将上一步所有数据交给网络接口（一般称之为 *网卡*, 计算机中负责网络的硬件设备）。
5. 同样，网卡会把上一步产生的整个数据包放到另一个消息头中，一些协议比如Ethernet协议定义了这一层消息头格式，使得数据可以在LAN上被发送。
6. 最后，网卡会将上述完整而又多重封装的数据发送出去，既可以是通过线路，也可以是通过空气（在WiFi的帮助下）。

当接收计算机收到上述数据包时，会进行相反的过程。它的网卡剥离Ethernet消息头，这样IP信息就会暴露给操作系统，操作系统确保IP地址是正确的，找出哪个程序在该端口上侦听，并将完全解析的数据发送给正确的程序。

完成封装这所有这些不同层次的行为称为 *协议栈* (这里“栈”一词与数据结构中的“堆栈”里的“栈”意思完全不同。)。

在这个过程中，由于不同层次负责完全不同的工作，所以使得整个过程都运作的很好。举个例子，一层负责数据完整性，一层负责将数据包在网络中进行路由，一层负责数据本身能在两个应用程序中被传输。每一层并不关心在它之下的层级对于数据正在做些什么。

最后，有一个很重要的概念：当然使用WiFi发送数据时，WiFi的硬件并不关心发送的数据是什么，它是因特网数据与否，数据完整性如何保证（或者不用不用保证）。WiFi只关心将一大块数据通过空气传给另外一个计算机。当数据到达另一方计算机，该计算机会剥离Ethernet相关的信息从而可以更加深入的查看数据包，再决定如何进行下一步处理。

由于这种设计中，上一层并不用关心什么数据被封装在了下一层，所以你可以将下面层次中用到的协议随意改变并且保持这一层仍然有效运作。所以当你在协议最高层编写程序时（这也是程序员经常做的），你根本不用关心下面的层级中发生了什么。通常，他们都是其它人需要解决的问题。

举一个具体的例子，你可以通过HTTP/TCP/IP/Ethernet访问到一个网页，或是通过TFTP/UDP/IP/Ethernet将文件传送给另外一个计算机。在这个过程中都有IP和Ethernet协议，因为他们并不关心他们上层的数据是怎么被发送的。

在上面的叙述中，我们省去了很多细节，不过在宏观上，整个过程和上面描述的是一致的。

有线和无线

当我们谈论局域网时，网络编程在这两种情况下：

- 局域网中的计算机通过有线的Ethernet网线连接[1]。
- 局域网中的计算机通过同一WiFi接入点连接。

它们在底层都是用Ethernet协议进行的通信。所以，当我们说计算机位于同一局域网，意思是它们要么通过网线连接在一起或者连接的同一个WiFi接入点。

[1] “以太网电缆”这个说法其实有点不准确，因为这里一般指的是电线本身，而以太网是一种协议，它有效地定义了通过这些电缆的电流模式。但这里我想表述的意思是“通常与以太网一起使用的电缆”。

思考题

- 因特网用的是哪种交换网络？
- 客户端程序和服务器程序的关系是什么？
- 在编写的网络程序中，操作系统发挥了一个什么角色？
- 协议是什么意思？
- 协议栈和网络封装的意义是什么？
- 无线网络和有线网络的关键区别是什么？

Sockets（套接字）API

在Unix中，套接字API进程提供了一种通用的相互通信的方式。它支持很多种通信方式，其中一种是通过因特网进行通信。而这正是我们现在感兴趣的一个知识点。

在C语言和Unix中，套接字API是用户态库调用和系统调用(直接调用操作系统的函数)的混合体。 在Python中，至少在类Unix系统中，套接字API库是调用的是C语言的套接字API。在其它语言中，它调用的是操作系统暴露给它的网络通信接口。

接下来，我们将动手写一些可以通过Internet通信的程序！

客户端连接过程

在使用套接字的过程中，让用户最疑惑不解的部分是当连接到另外一台电脑前，需要进行好几步才能完成连接。而这整个过程并不是显而易见的。

这个主要的过程是:

1. **向操作系统申请一个套接字**。在C语言中，套接字是一个用来记录网络连接的文件描述符（一个整数）。在Python中，将会用一个对象来表示一个套接字。在不同的语言中，可能会由不用的概念来表示套接字。但这里最重要的是，程序端需要有一个标示物来指向接下来的数据传输。请注意，这个步骤完成时，并没有连接到一个具体的东西。

2. **通过查询DNS** 将一个人类可读的字符串（比如`example.com`）转化为一个IP地址（比如198.51.100.12）。DNS是一个分布式的数据库，用户可以通过对它进行查询来得到一个IP地址。我们需要知道IP地址，这样我们才能知道该链接到哪个机器。

    - Python提示: 当你在Python中使用机器名调用`socket.connect()`，这个函数会自动调用`socket.getaddrinfo()`做一次DNS查询。
    - C语言提示: 在C语言中，可以使用`getaddrinfo()` 进行DNS查询。

3. **连接套接字** 到对应的IP的地址和一个指定的端口。 一个端口就好比是一扇你可以通过的门，它们用序号0到65535表示。比如，一个熟知的端口号是80，这是互联网上使用HTTP协议（未加密）进行通信的端口。在远端，必须要有一个正在监听的端口，不然连接将会失败。

4. **发送和接收数据**。这是我们期待已久的部分，计算机网络中，数据就是一串字节。

5. 关闭连接。当数据传输完毕，链接到远端的套接字会被关闭因为我们已经没有什么可以传输了。远端亦可按照自己的意愿关闭套接字。

服务器端监听过程

编写服务器端程序有些许不同。

1. **向操作系统申请一个套接字**。这点和客户端无异。

2. **将套接字绑定到一个端口**。在服务端，你需要暴露出一个其它服务器可以连接上的端口，比如，“我要在80端口上开始监听啦！” 注意:非根账号或者管理员身份运行的程序是不能绑定到1024以下的端口的，这些端口是计算机的保留端口。应该给服务程序选择一个大的、不知名的端口号，比如1.5万到3

万之间的端口号。如果程序尝试绑定到一个服务器正在使用的端口，那么将会得到“地址已在使用”错误。 计算机之间的端口号是独立的，不同的计算机之间使用一样端口号是没有问题的，但是同一台计算机的上的两个程序是不能使用同一个端口号的。 有趣的事实：客户端程序在通信时也是会绑定到一个端口上的，如果你没有显示的制定它们，它们会自动绑定到一个没有被使用过的端口上——通常这正是我们想要的。

3. **监听将要到来的连接**。程序需要让操作系统知道什么时候在我们选定的端口上会有要到来的连接。

4. **接受到来的连接**。如果没有任何连接，服务器会阻塞（表现为sleep）等待新的连接。当新的连接到来时，它会被唤醒。当连接被接受时，会返回一个新的套接字，在第一步里面负责监听的套接字会依然保持监听状态，操作系统会为每一个连接都特意的创造一个新的套接字。正因为这样，服务器才能同时处理很多连接。有时，服务器会创建一个新的线程或者进程去处理每一个连接上来的客户端，不过并没有规则说服务器端一定要这样做。

5. **发送和接收数据**。通常情况下，服务器会从客户端接收请求，然后服务器会发回对该请求的响应。

6. **继续监听和接受新的连接**。服务器一般都是永不推出的进程并且在其生命周期内尽可能的处理更多的请求。

思考题

- `bind()`函数在服务器端扮演的角色是什么？
- 客户端程序调用 `bind()`函数会发生什么？（可能需要在因特网上搜索下这个问题的答案）
- 思考一下为什么 `accept()` 返回一个新的套接字，而不是仅仅重用`listen()`返回的套接字。
- 如果服务器在一次`accept()`之后没有继续循环等待新的连接会怎么样？这时如果有个新的连接会发生什么？
- 如果一个计算机通过端口号3490使用TCP协议，其它计算机可以使用同一端口号么？
- 思考一下为什么存在端口号这个概念，它能提供IP地址所不能提供的哪些功能？

资源

- Python 套接字文档
- [flbg[*Beej's Guide to Network Programming*|bgnet]]--可选，C语言版

分层网络模型

在正式介绍之前，首先需要阐述几个概念:

- **IP 地址** -- 传统上，在因特网上使用一个4字节数唯一的标示你的计算机。数字之前用点号隔开，比如:`198.51.100.99`。这些被称之为IP协议版本4（"IPv4"）地址。通常，在没有任何其他版本标识符的情况下，使用“v4”作为默认版本。

- **端口** -- 程序通过编号为0-65535端口进行通信，并且它们与TCP或UDP协议相关联。由于多个程序可以在同一个IP地址上运行，端口提供了一种唯一识别网络上这些程序的方法。举个例子，一个常见的应用就是网站服务一般都是在80端口上监听连接。对于服务端程序，发布端口号对于客户端非常重要，因为客户端程序需要知道通过什么端口才能连接上服务端程序。而客户端一般都是让操作系统自动选择一个还未使用的端口号，因为没有程序需要知道怎么能主动连上客户端程序。在一个URL中，端口号一般被置于冒号后面。比如我们想在端口`3490`连接`example.com`，那么URL写做：`http://example.com:3490/foo.html`。在计算机中，小于1024的端口号只有根账户/管理员账户才能有权限绑定（但是可以随意连接）。

- **TCP** -- 传输控制协议，负责提供可靠，有序的数据传输。从更高的层次来看，使得分组交换网络看起来更像电路交换网络。TCP使用端口号来标示发送和接收数据。TCP协议于1974年被发明出来并且现在被广泛使用于计算机网络。在套接字API中，TCP套接字被称之为 *流套接字*。

- **UDP** -- TCP的兄弟协议，不过更加轻量级。该协议不会保证数据一定会被送达，或者按需到达以及到达的数据不重复。如果数据到达，那么数据本身是没有错误的，这就是接收端能获得所有信息。在套接字API中，UDP套接字被称之为 *数据报套接字*。

- **IPv6 地址** -- 四个字节已经不足以标示唯一一个地址了，所以IP协议版本6将标示地址的大小扩展到16个字节。IPv6地址看起来是这样`::1`或者`2001:db8::8a2e:370:7334`，甚至更长。

- **NAT** -- 网络地址转换。这是一种方法，可以让用户拥有私有的子网地址，这些子网地址可以不是全局唯一的，而该方法可以提供一种转换的方式将全局唯一的地址转换成内部地址，这样可以让数据路由到合适的地址。私有地址常常以`192.168.x.x`或者`10.x.x.x`开头。

- **路由器** -- 一种在报文分组网络中转发数据报的特殊的网络设备。它检查目的IP地址，以确定哪条路由可以使数据包更接近的发送到目标地址。

- **IP** -- 互联网协议。它负责通过IP地址识别计算机，并使用这些地址通过各种路由器将数据发送到接收方。

- **LAN** -- 局域网。这是一种所有计算机都直连的网络，中间不经过路由器。

- **网络接口** -- 计算机上的物理网络设备。一台计算机上可能有很多网络接口。你的电脑上大概率有两个: 一个有线网络接口和一个无线网络接口。一个路由器可能拥有很多网络接口，如此才能把数据报发送到不同的目的地地址。但是你的家用路由器可能只有两个接口: 一个用来接入局域网，其它用来接出到因特网上的其它部分。每个接口都有一个独立的IP和MAC地址。操作系统会命名本地计算机上的接口，它们可能叫`wlan0`或者`eth2`，抑或是其它名称。这些起名方式取决于硬件和操作系统。

- **报头** -- 一些加在特定协议之前的数据。报头包含适合该协议的信息，譬如，TCP报头将包括一些错误检测和纠正信息以及源和目的端口号，IP报头包括源IP地址和目的IP地址，以太网报头包括源MAC地址和目标MAC地址。HTTP响应报头中包括数据长度、修改日期以及请求是否成功等信息。 在数据前加

一个报头就好比在传统邮件外面包一个信封，或者在信封外面再继续加上信封。 当数据在网络中传输时，额外的报头会不断的被添加和移除。通常，最顶端（最外端）的报头不断的被移除和添加是一个很普遍的操作，就像一个堆栈（不过一些软件和硬件会去取更深的报头信息）。

网络适配器 -- 通常也被称为“网卡”，你计算机上负责处理网络事宜的硬件。

MAC 地址 -- 以太网接口都有MAC地址，形式为`aa:bb:cc:dd:ee:ff`，是由一些随机的一字节的数字组成。MAC地址一般都是6字节，并且在局域网中必须是唯一的。当一个网卡被组装好，就会赋予它一个MAC地址，这个MAC地址通常情况下会伴随其一生。

分层网络模型

当你在互联网上发送数据时，数据会被 *封装* 在不同的协议层。概念上的分层网络模型的各层对应于各种类型的协议。这些协议负责不同的指责，比如，描述数据，保证数据完整性，路由数据，本地传达等等。所以这是一个先有鸡还是先有蛋的问题，因为我们不能只讨论其中一些而不讨论另一些。在深入了解这些协议之前，最好是先了解下各个层次的报头。

分组协议数据实例

让我来看看当发送一个HTTP请求时发生了什么。

1. 浏览器构造了类似如下的一个HTTP请求：

   ```
   GET / HTTP/1.1
   Host: example.com
   Connection: close
   ```

   而这就是浏览器关心的所有内容，它不关心IP协议是如何路由数据或者TCP协议是如何保证数据完整性的，更不会关心以太网上的内容。 它只会说“把这些数据发送到指定计算机的80端口”。

2. 然后，操作系统开始接管，并且说:“收到，你要求我通过面向流的套接字发送此数据，我将使用TCP协议来完成此操作，并确保所有数据完整有序地到达。” 所以，操作系统拿到上一步的HTTP数据数据之后，将其封装到一个TCP报头中，在该报头中会包含一个端口号。

3. 接着，操作系统说:“你想把它发送到IP地址是198.51.100.2的远端计算机，所以我们将使用IP协议来做到这一点。” 为了达到这一点，操作系统将上一步所产生的所有TCP-HTTP数据都放到一个IP报头中。所以，现在操作系统将要发送的数据是: IP-TCP-HTTP.

4. 之后，操作系统会看一眼它的路由表并且判断下一步将数据发送到哪里，也许很便利，终点就是局域网上的一个网站服务器。但是更多的情况下，终点会是其它的某 个地方，所以，通常情况下，数据会被发送到你房子里的一个为了将数据转发到更大因特网的一个路由器上。无论哪种情况，操作系统要么会把数据发送到局域网的一个服务器上，或者是一个依然在局域网内的出口路由器上。所以，言而总之，数据报会被发送到局域网上的一台计算机中。 局域网中的每台计算机都有一个以太网地址（也就是所谓的 *MAC 地址* -- 代表着“介质访问控制”），所以，发送端的操作系统会查找下一跳目的IP地址对应的MAC地址，它可能是一个本地的网站服务器，或者是一个出口路由器（这个过程通过查询一个叫做 *ARP 缓存* 的表，我们将在后面的章节详细介绍）。 到这里，操作系统会将上面三步所组成的整

个IP-TCP-HTTP数据报包装到一个以太网报头中，所以这时候整个发送的内容变成了Ethernet-IP-TCP-HTTP。对于网页的请求依然在其中，只不过被深深的藏在了各层的协议中！

5. 最终，所有的数据都会从线路中被发送出去（即使是WiFi，我们依然用“线路”）。

   对应远端MAC地址的计算机，在线上仔细的聆听对应的以太网数据包，然后将其读入（以太网数据包被称之为 *以太网帧*）。

   接着，该端操作系统会剥离以太网数据头，暴露出在它之下的IP数据头，用以查看目的地IP地址。

6. 如果正在被检查的计算机是一台拥有IP数据包所携带的IP地址的话，对应的操作系统会剥离IP数据头并进行更深层次的查看（如果该服务器不拥有对应的IP地址， 那么出现了一些问题，该数据包将会被丢弃）。

7. 接下来，操作系统会查看TCP数据头，并且进行所有TCP所需的计算以保证接收端的数据没有损坏。如果数据损坏，该服务器会回复“TCP魔法咒语”，告诉对端：“喂，我需要你重新发送下刚才的数据！”。 注意，无论是网页浏览器或者服务器本身都完全不知道上述的TCP转换，这一切都是发生在幕后。就它所能看到的，数据只是奇迹般地完整有序地到达。 核心原因就是，这些部件都位于计算机网络的更高层次中，它们从来不用担心路由过程或者任何相关的事宜。低层次的程序会完全的负责这一切。

8. 如果在TCP这一层一切都完好，这一层协议的报头就会被剥离，这时操作系统将会看到剩下的HTTP数据。它将唤醒一直在等待的相应的进程（网站服务器程序），并且将HTTP数据喂给它。

但是，如果目标以太网地址是一个中间的路由器会发生什么呢?

1. 路由器会如同之前描述的过程一样，剥离掉以太网帧。

2. 接着，路由器开始查看目的地IP地址，路由器会查询其路由表，然后决定使用哪个接口发送数据包。

3. 随后，路由器根据上一步得到接口信息，将整个的IP层的数据包裹在新的以太网帧中，发送给网络中的下一跳路由器。（这里以太网只是一种底层协议，还有其它很多比如光纤中使用的通信协议。这就是分层的美妙之处--用户可以在传输的中途切换协议，上面的HTTP数据完全不知道发生了任何这样的事情。）

因特网分层模型

让我们从一个简单的将这种传输从上到下划分为不同的层次的模型开始。(请注意，下面列出的协议列表远非详尽无遗。)。

层级	职责	协议举例
应用层	结构化的应用层数据	HTTP, FTP, TFTP, Telnet, SSH, SMTP, POP, IMAP
传输层	数据完整性，数据分割和组装	TCP, UDP
网络层	路由	IP, IPv6, ICMP
链路层	物理设备，线路上的电信号	Ethernet, PPP, token ring

下面，你将会看到在这个模型中，不同的协议在各个层级上是怎么履行各自职责的。另一种思考方式是，所有实现HTTP、FTP或SMTP的程序都可以使用TCP或UDP传输数据。(通常，所有用户编写的套接字程序和应用都会工作在应用层。)。所有使用TCP或UDP传输的数据都可以使用IP或IPv6进行路由。

所有使用IP或IPv6进行路由进行路由的数据包都可以通过以太网或者PPP等协议在网线上传输。一个数据包在真正的在网线上传输之前，下一层协议都会在上一层协议数据之前加入对应的数据包。这种模型足够支撑数据在因特网上传输。有一句名言:尽量简单，但是不能过度简单。

不过网络世界上不止一种因特网，所以有一些其它人定义出来的更加通用的网络模型: OSI 网络模型。

ISO OSI网络分层模型

如果你正在考某种证书或者远远不想只成为一个普通程序员，那么了解这一部分知识益处良多。

因特网分层模型是ISO定义的OSI网络模型的一种特殊而又具体的实现（ISO OSI这里形成了回文，一个神奇点）。ISO OSI模型全称为国际标准化组织定义的开放系统互连模型。我知道“ISO”不是“国际标准化组织”的直接英文缩写，但我没有足够的全球政治影响力来改变这一点。

回到现实中来，OSI模型和因特网模型很相似，只不过粒度更细。因特网模型可以一对多的映射到OSI模型，如下所示：

ISO OSI模型层级	因特网模型层级
应用层	应用层
表示层	应用层
会话层	应用层
传输层	传输层
网络层	网络层
数据链路层	链路层
物理层	链路层

如果我们看一下OSI模型，我们可以看到存在于各个层级中的不同协议，类似于我们在上面看到的因特网模型。

ISO OSI 层级	职责	示例协议
应用层	结构化的应用化数据	HTTP, FTP, TFTP, Telnet, SMTP, POP, IMAP
表示层	编码传输，加密，压缩	MIME, SSL/TLS, XDR
会话层	挂起，终止，重启两台计算机之前的会话	Sockets, TCP
传输层	数据完整性，数据分割和组装	TCP, UDP
网络层	路由	IP IPv6, ICMP
数据链路层	将数据装入帧中	Ethernet, PPP, SLIP
物理层	物理器件，线路上的信号	Ethernet physical layer, DSL, ISDN

后续的章节中，我们将专注于因特网模型，因为它对于你所想要的百分之九十九的网络编程都是适用的。不过，如果你想面试一个网络编程相关的岗位，那么OSI模型也是你所需要知晓的。

思考题

- 当路由器看到一个IP地址时，它是怎么知道如何转发至下一跳的?
- 如果每台电脑都只能有一个不重复的IP地址，4字节的IPv4地址可以表示多少台计算机?
- 如果每台电脑都只能有一个不重复的IP地址，16字节的IPv6地址可以表示多少台计算机?
- 给统计爱好者的附加题:赢得超级乐透头奖的几率约为3亿比1,那么随机挑选我预先选择的16字节(128位)数字的几率是多少?
- 想一想为什么在分层模型中，IP协议是在TCP协议之上，也就是为什么TCP的报头在IP报头的前面，而不是反过来?
- UDP是不可靠协议而TCP是可靠协议，那么有什么场景下会使用UDP协议?

实战项目: HTTP客户端和服务器端

我们将编写一个可以从web服务器下载文件的套接字程序!这将是我们的“网络客户端”。如果我们的代码正确的话，该客户端可以和任何web服务器通信。不仅如此，我们同样将编写一个简单的web服务器!这个程序将能够处理来自我们编写的web客户端的请求… 或任何其他web客户端，如Chrome或Firefox!

这些程序都会使用一种你已经听说过的协议进行通信:HTTP，即超文本传输协议。正因为它们使用HTTP，而同样，像Chrome这样的浏览器也使用HTTP，所以它们之间可以通过该协议进行通信!

限制

为了更好的从底层理解套接字API，该项目**不会**使用下面这些辅助函数:

- `socket.create_connection()` 函数
- `socket.create_server()` 函数
- 任何在`urllib`中的模块

当你完成这个项目时，你应该对于这些辅助函数如何工作的有了更深入的了解。

Python字符编码

Python套接字接收和发送的都是字节，不是Python中的字符串。当需要发送字符串或者想按照字符串模式打印字节时，用户需要将其进行转换。

字节序列取决于字符串所使用的 *字符编码*。字符编码决定了字节对应于哪一个编码，一些你可能听过的编码方式比如ASCII或者UTF-8，除此之外还有浩如烟海的编码方式。一般情况下，网页的默认编码为“ISO-8859-1”。

编码方式很重要，因为你必须把你的Python字符串编码成一个字节序列，而每当你这样做的时候，你需要指定编码方式。(默认为UTF-8。)

下面的示例表示如何把一个Python字符串转换为ISO-8859-1编码的序列:

```
s = "Hello, world!"          # String
b = s.encode("ISO-8859-1")   # Sequence of bytes
```

这串编码序列将会通过套接字进行传输。下面的示例表示了如何将从套接字接收到的字节转换为ISO-8859-1字符串:

```
s = b.decode("ISO-8859-1")
```

然后，你就可以将其按照人类可读的模式打印出来了。显然，如果数据并没有以ISO-8859-1进行编码，在字符串中，你会看到奇怪的字符或者之前得到一个错误。 编码ASCII、UTF-8和ISO-8859-1对于基础的拉丁字母，数字或者标点都是相同的，所以在这些编码下，所得到的字符串应该都是相同的，除非你开始处理一些奇怪的Unicode字符。

如果你是用C语言编程，大部分情况下你无需担心编码方式而只是直接打印出这些字节流。有时候你可能会得到一些垃圾值，不过在大多数情况下，都是可用的。

HTTP 小结

HTTP是基于 *请求* 和 *回复* 的概念来进行操作的。客户端发起一个网页的请求，然后服务器端会将请求发回。

一个简单的HTTP请求如下:

```
GET / HTTP/1.1
Host: example.com
Connection: close
```

上面展示了一个请求 *报头* 长什么样，在第一行中包含了请求方法，路径以及协议，接着是任意的头部字段。在报头的最后，有一行空行。这个请求表示 “从服务器example.com获取根网页，我将在收到你的响应后立即关闭连接。”

请求头的每一行以回车/换行符(CRLF)结束，无论是Python还是C语言，都可以用以下方式表示CRLF:

```
"\r\n"
```

如果你是在请求一个特定的文件，文件名会位于请求的第一行，看起来如下:

```
GET /path/to/file.html HTTP/1.1
```

(如果这个报头中还有一个负载内容，它会被加在在空白行后面，同时带有一个 `Content-Length` 字段，并以字节为单位给出负载的长度。在这个项目中，我们并不会涉及这个内容。)

对应的从服务器返回的HTTP回复如下所示:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 6
Connection: close

Hello!
```

这个回复表示: “你的请求已经成功接收，这里是给你的带有6个字节明文的应答。同时，我将会在将此消息发送给你之后关闭连接，这个应答里的负载是'Hello!'。”注意，这里的 `Content-Length` 字段值是6，正好对应负载 `Hello!` 的长度，6个字节。

其它常用的负载字段是 `Content-Type`， 通常如果负载是HTML数据的话，该字段会被填充为 `text/html`。

客户端

客户端程序文件名为`webclient.py`，你可以在编写服务器端程序之前先编写客户端程序，因为客户端程序可以对着真实的现有web服务器上进行测试。在正式测试之前，不需要同时编写客户机和服务器。

该客户端程序的目标是可以通过命令行运行，运行方法如下所示:

```
$ python webclient.py example.com
```

得到输出应该如下所示:

```
HTTP/1.1 200 OK
Age: 586480
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Thu, 22 Sep 2022 22:20:41 GMT
Etag: "3147526947+ident"
Expires: Thu, 29 Sep 2022 22:20:41 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: ECS (sec/96EE)
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 1256
Connection: close

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    ...
```

(为了方便展示，这里的输出被截断了，在你的测试中，应该会有网站的整个HTML内容。)

这里需要注意的是，输出的第一部分是包含来自服务器的所有字段的HTTP响应，然后有一个空行，空行后面的所有内容都是响应负载内容。同时: 在运行客户端程序时，你需要指定一个端口号来连接服务器端，当你没有指定时，会用端口号80作为默认值。所以，你同样可以使用不同的端口号连接跑在非80端口的网站服务。方法如下所示:

```
$ python webclient.py example.com 8088
```

这时程序会尝试连接8088端口上的服务端程序。

在客户端Python程序文件的最上端，需要导入socket包，这样你才能使用下面的这些函数。

```
import socket
```

下面是客户端程序里主要使用的函数以及其作用:

- 使用 `socket.socket()` 创建一个新的套接字，不需要给它传递任何参数--默认参数对于此项目足够使用了。
- 使用 `s.connect()` 将新创建的套接和目的地连接起来，在这里，你可以不用显示的去处理DNS相关的步骤，因为`.connect()`函数已经帮你做好了这一步。此函数使用一个元组作为参数，分别表示服务器名和端口号。

  ```
  ("example.com", 80)
  ```

- 接下来就是，构造并且发送该HTTP请求，你可以使用上面介绍过的那个简单的HTTP请求。**别忘了在消息头之后的空行， 也别忘了所有行都是使用`"\r\n"`结尾。** 我推荐使用 `s.sendall()` 方法来发生构造好的请求，你也可以使用 `.send()`，不过该函数可能只会发送出去一部分函数。（使用C语言的程序员可以在我的另一本书中找到`sendall()`的实现方法。）
- 使用`s.recv()`来接收网站服务器的响应，服务器通常会返回一些字节作为响应消息。如果服务器端是一个大型的网站服务器，通常客户端需要在循环中多次调用该函数来得到完整的信息。当服务器关闭连接并且没有更多的数据要读取时，它将返回一个零元素的字节数组，例如:

  ```
  d = s.recv(4096)  # Receive up to 4096 bytes
  if len(d) == 0:
      # all done!
  ```

- 当一切结束时，调用 `s.close()` 来关闭该套接字。你可以使用下面的命令来测试你的客户端和一些网站的交互:

  ```
  $ python webclient.py example.com
  $ python webclient.py google.com
  $ python webclient.py oregonstate.edu
  ```

服务端

服务端程序文件被命名为`webserver.py`。

用户可以使用以下命令从命令行启动服务端程序, 如下所示， 服务器程序会被启动并且会在28333监听连接。

```
$ python webserver.py
```

同样，用户可以显示的指定服务器端程序监听的端口号，方法如下所示:

```
$ python webserver.py 12399
```

服务器端程序会一直运行着，以便于接收各种接进来的请求。（永远代表着只有当你按下ctrl+c的时候才会停止。）

我们这个项目中的服务器端程序，无论请求是什么，都只会简单的回复同一条消息，让它发回简单的服务器响应，如上所示。所以这不是一个全功能的网站服务器，不过它可以作为复杂程序的起点。

下面是该程序的一些Python技术细节:

- 像客户端程序一样，先向操作系统申请一个套接字。
- 当调用`socket()`函数之后，你需要添加看起来很疯狂的一行:

  ```
  s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
  ```

  变量`s`是你从`socket()`得到的套接字描述符，这将防止在某些情况下`bind()` 上出现"Address already in use"错误，这个错误通常会让人感到困扰，通常发生在服务器崩溃之后。后面的章节中我们会介绍为什么会出现这个错误。

  如果你对添加这行代码感到厌倦，然后得到了这个错误，只需要等待几分钟，这样操作系统会断开的一些现有的连接。

- 调用`s.bind()`将套接字绑定在一个端口上，该函数需要携带一个参数，一个你想要绑定的地址和端口信息的元组。可以将地址置空表示绑定本地地址，比如，“绑 定在任一本地地址的28333端口”可以传递如下参数:

  ```
  ('', 28333)
  ```

- 调用`s.listen()`将套接字激活并开始监听。
- 调用`s.accept()`接收新的连接，注意，这个函数将会返回一个元组。元组的第一个元素是一个代表新连接的新套接字。（旧的套接字依然会保持监听状态直到用户在结束处理该次请求之后再次调用`s.accept()`。）

  ```
  new_conn = s.accept()
  new_socket = new_conn[0]  # This is what we'll recv/send on
  ```

- 接收从客户端来的请求，类似于在客户端程序里面一样，需要在一个循环中调用`new_socket.recv()`，当服务器端接收到一个空行（例如，`"\r\n\r\n"`）表 示服务器读取了足够的消息并且可以退出了。(在本项目中，我们不会处理负载，关于这个正确的思路是查看请求中的`Content-Length`字段的数值，然后将报头加上对应的数值，这对于你可以作为一个附加题。) **注意**: 这次，你不能通过判断`recv()`是否返回一个空字符串来判断是否终止循环，空字符串只会发生在客户

端已经关闭了连接，但是客户端端没有完全成功关闭并且依然在等待回复。所以正确的做法应该是一直调用`recv()`直到看到消息头最后的空行。

- 服务器端发送回复，以上面的描述，此项目中的服务器只会发送“一个简单的回复”。
- 使用`new_socket.close()`关闭用于接收的套接字。
- 继续使用`s.accept()`接收下一次请求。

现在，你可以在一个窗口中运行网站服务器程序而在另一个窗口中运行客户端，用来检查是否它们会互相连接！一旦服务器端程序可以和`webclient.py`通信，下面尝试使用网页浏览器程序来打开它！ 首先，在一个未被使用的端口上运行改程序（请选择一个随机的大一点的数字）：

```
$ python webserver.py 20123
```

在浏览器中打开`http://localhost:20123/` 查看这个网页（`localhost` 是“该计算机名”。），如果你看到了相应的内容，那么恭喜你，服务器端程序已经成功了！

尝试去打印出`.accept()`函数收到的内容，看看输出是什么?

你有没有注意到，如果你使用网络浏览器连接到你的服务器，浏览器实际上会建立两个连接?如有兴趣可以深入研究一下，看看你是否能找出原因!

提示和帮助

地址已经被使用

如果你的服务器程序崩溃了，而你试图重新启动它，这时你会得到一个“地址已经在使用”的错误，这说明着系统还没有完成端口销毁工作(在这种情况下，“地址”在这里指的是端口。)。如想消除这个错误，要么使用另一个端口启动服务器程序，要么等待一两分钟，这时会出现超时，操作系统会进行清理。

接收到部分数据

即使你告诉`recv()`函数你想要接收到4096字节的数据，该次调用也无法保证你能全部收齐指定多数据。有可能服务器发送少了，也有可能是在传输中，数据发生了分割，只有部分数据被接收到。

调用`recv()`函数却不能得到完整的数据，使得在处理HTTP请求或者回复中显得很棘手。更糟糕的是，数据在传输中可能会在标题末尾的空行分隔符的中间被分割!

不要对于一次`recv()`调用就能接收到所有数据抱有幻想。永远在一个循环中调用该函数，将收到的数据不断附加到一个缓冲中，直到你收到了你需要的数据。

当你尝试对一个已经关闭的连接调用`recv()`函数时，它会返回一个空字符串（Python中）或者`0`（C语言中），这就是如何判断连接已关闭的方法。

HTTP 状态码 301，302

当你的客户端收到服务器端发来的`301`或者`302`的状态码时，大概率还会带有`Moved Permanently`或者`Moved Temporarily`的消息，这说明你正在请求的URL中的服务器资源已经被移动到另一个URL。

如果你查看一下的消息头，你会发现一个`Location:`字段:

例如，当你运行`webclient.py google.com`时，你会收到如下消息:

```
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Wed, 28 Sep 2022 20:41:09 GMT
Expires: Fri, 28 Oct 2022 20:41:09 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Connection: close

<HTML><HEAD><meta http-equiv="content-type"
content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
Connection closed by foreign host.
```

注意，消息系的第一行告诉我们所请求的资源已经被挪走，第二行的`Location:`字段告诉我们该资源被移动到了哪里。

当一个网页浏览器看到`301`消息码时，它会自动切换到另外的URL，用户无需担心处理该过程。

你可以试试在浏览器中输入`google.com`然后你会发现过一会儿之后浏览器里的地址自动切换成了`www.google.com`。

HTTP状态码400，501（以及任何500状态码）

如果客户端从服务器获得`400`或任何`500`的响应代码,很有可能你发出了一个错误的请求，简而言之，该请求所携带的数据在某种程度上是错误的。确保消息头的每个字段都以`\r\n`结尾，并且消息头以空行结束(即:`\r\n\r\n`是消息头的最后4个字节)。

HTTP 状态码404，资源未找到！

请确保在你的请求中含有`Host:`字段并且正确的设置成了和你传入命令行中一样的内容，如果设置错误，将会得到404错误。

扩展练习

下面是一些可以让你加强理解的练习，请务必尝试一把！

- 修改服务器端代码，打印出连接上来的客户端IP地址和端口号。提示:可以查询下Python中`accept()`函数的返回值。

- 修改客户端程序，使其可以发送负载数据，你需要根据负载内容正确设置`Content-Type`和`Content-Length`。
- 修改服务器代码，从请求中解析和打印“请求方法”，大部分会是`GET`，不过也有可能会出现`POST`或者`DELETE`。
- 修改服务器端代码，解析并打印客户端发送的负载信息。

因特网协议（IP）

该协议负责在因特网上路由数据包，类似于邮局负责在邮件网络中路由传递信件。与邮局系统一样，Internet上的数据必须标有源地址和目的地址，即 *IP地址*。

IP地址是一个字节序列，它唯一地标识因特网上的每台计算机。

术语

- **主机** - “计算机”的另一个称呼。

两个主要的版本

IP协议两个主要版本: 版本4和版本6。IP协议版本4指的是 "IPv4" 或者简称 "IP"，IP协议版本6通常会显示的标记为“IPv6”。

你可以从如下的两个例子中瞥见IP地址的模式:

- 版本4的IP地址形如:192.168.1.3
- 版本6的IP地址形如:fe80::c2b6:f9ff:fe7e:4b4 主要区别在于组成地址空间的字节数。IPv4为每个地址4字节，IPv6为每个地址16字节。

子网

每个IP地址都可以被分割成两个部分，第一部分的用来标示独立的网络，第二部分用来标示该网络中独立的主机（例如，计算机）。这些单独的网络被称为 *子网*，它们可以支持的主机数量取决于在地址中为识别该子网上的主机保留了多少位。

举一个人造的但是非因特网中真实使用的例子，我们用8位表示一个“地址”，假设其中前六位是网络编号，而后两位是主机编号。按照这个逻辑，一个地址可能看起来如下所示:

```
00010111
```

该地址被分割称为两个部分（因为我们定义了前6位是网络编号）:

```
Network  Host
-------  ----
000101   11
```

所以该地址表示网络5（二进制`101`），主机3（二进制`11`），一个地址中网络号永远都在主机号的前面。

注意，如果只用2位表示机器，那么在这个子网中一共最多只能有4个主机编号，分别是0，1，2，3（或者用二进制表示`00`, `01`, `10`, 和 `11`）。如果按照IP协议的定义，那么真正能用的只有两个主机编号，因为全0的主机编号和全1的主机编号是保留编号。

下一章，我们将会查看两个具体的IPv4和IPv6的例子，这一章最重要的知识就是每一个地址都会分成网络和主机两个部分，网络部分在主机部分之前。

IP层额外协议

在IP协议的同一层，还有一些协同IP协议工作的其它协议。

- **ICMP**: 互联网控制消息协议, 这是一种用于在IP节点之间交流IP控制元数据的协议。
- **IPSec**: 网际安全协议, 负责加密，认证等功能，一般和VPN(虚拟专用网)一起使用。

用户通常会的使用`ping`这个程序利用ICMP协议，该命令使用ICMP协议的“回声请求”和“回声应答”消息。而`traceroute`程序则利用了ICMP的“超时”消息来找到数据包是如何被路由的。

私有网络

私有网络是指那些藏在路由器之后的，并不拥有全球唯一IP地址的机器（但是，它们在各自的局域网中有自己独立的地址。）。该技术是通过一种叫做NAT(网络地址转换协议)来实现的，我们将会在后面的章节详细介绍。目前，让我们先假设我们接触到的所有地址都是全球唯一的。

静态地址，动态地址以及DHCP协议

如果你有一个需要被访问的网站，或者一个你不时要去通过SSH连接的服务器，你需要一个 *静态IP*，这表示你需要一个被分配的全局唯一的IP地址并且它不会发生改变。这个很像门牌号，如果你需要别人能找到你的家，你的门牌号是不能随意更改的。

不过由于IPv4地址的数量是有限的，静态IP是很昂贵的。通常，网络服务提供者会有一系列的带有子网的IP地址并且会 *动态* 分配给需要的用户。这意味着，当你重启你的调制解调器的时候，你的所有终端可能会被分配到完全不同的IP地址。（除非你花费了价钱去买了一个静态IP。）

事实上，当你的笔记本电脑连上WiFi时，基本上你获得的都是一个动态IP地址，你的计算机会连上局域网，并且广播一个数据包说:"你好，我在这里！有没有人可以告诉我我的IP地址是什么？最好是能带有更多的一些元信息。"这是一个合理的过程，因为大部分情况下，不会有程序会尝试连接你笔记本电脑上的服务器程序，通常，都是笔记本电脑去连接其它服务器程序。

那么，这其中详细的原理是什么？这就要说到DHCP协议了，通常在局域网上有一个专用的服务器，上面会运行一个符合DHCP（*动态主机配置协议*）协议的程序。DHCP服务器会负责记录在局域网中，哪些IP地址已经被分配并且正在被使用，哪些还未被使用，处于空闲状态。它会找到一个未被使用的IP地址，并且使用DHCP回复消息告诉你的笔记本电脑新的IP地址是什么，并且还会附带局域网上的一些其它你的电脑接入需要的信息。（比如，局域网掩码，等等。）

如果你家中有WiFi，那么大概率你已经有了一个DHCP服务器，大部分从你的网络提供商买来的路由器都已经设置好了DHCP，这使得你的笔记本电脑可以从你的局域网中获取到相应的IP地址。

思考题

- IPv6地址的总数量是IPv4地址的多少倍?
- 应用层程序通常会有自己的加密算法的实现（比如，ssh或者带有HTTPs的网页浏览器）。思考下，使用网络层的IPSec加密而不用在应用层加密的好处和坏处都有什么?

- 如果一个子网保留5位作为主机标示符，那么该子网中可以支持多少主机？别忘了全0地址和全1地址都是保留地址。
- 拥有一个静态IP地址的好处是什么？它和DNS的关系又是什么？

因特网协议版本4（IPv4）

这是第一个被广泛使用的因特网协议版本，并且至今也在被广泛使用着。

IP 地址

一个IPv4地址用“点和数字”表示，比如:

```
198.51.100.125
```

IPv4地址永远是四个数字，每个数字用一个字节表示，所以每个数字可以表示的范围是0到255（二进制 `00000000` 到 `11111111`）。这也表示每个IPv4地址都是4个字节（32位）。

子网

整个IP地址都含有 *子网*，IP地址的第一部分表示的就是我们正在讨论的子网编号，剩下的表示该子网中计算机的编号，而由多少位来构成“IP的第一部分”是可变的。当你设置公开的IP地址时，你将需要给向你付费的来获得链接的任何人划分并提供子网地址，你能提供的子网地址越多，你需要付出的费用也越多。

所以，当你说:"我需要180个静态IP地址。"时，你的网络提供商可能会想，好的，这意味着你需要180个IP地址和2个保留的IP地址（全0和分配地址中最大数字），一共182个地址。根据这个数字，我需要用8位来表示数字0-255，这是能提供182个IP地址需要的最小的位数。基于此，它们会提供给你一个子网，其中用24位网络位和8位主机位。

这个可以用如下的方式表示:

```
你的子网位于198.51.100.0，其中前24位表示网络位，后8位表示主机位。
```

不过，这些都太繁琐了，所以一般情况下，我们使用斜杠来更简洁的表示:

```
198.51.100.0/24
```

这种表示法告诉我们IP地址中的24位表示网络编号。（所以`32-24=8`位表示主机编号。）不过这到底代表什么呢？让我们把这个描述简单的用图表示下:

```
24 位网络码
-----------
198.51.100.0
           -
         8 位主机码
```

或者将这些数字全部转换为二进制:

```
        24 位网络码          | 8 位主机码
-------------------------------+---------
11000110 . 00110011 . 01100100 . 00000000
     198         51         100          0
```

上述的所有位于我们伪造的子网中的IP都以`198.51.100.x`开头，最后一位就是我们正讨论的主机号。下面是我们这个子网中的一些合理的主机号:

```
198.51.100.2
198.51.100.3
198.51.100.4
198.51.100.30
198.51.100.212
```

不过有两个地址具有特殊的意义（请看下表）:

```
198.51.100.0     保留
198.51.100.255   广播地址（详情请看下面）
```

除了这两个IP地址之外，我们可以使用其它任何合法的ip地址。现在，我特意选择了一个表示子网的字节在合法范围边界的例子，因为这样更容易看出整个最后一个字节是否还是合法主机号。但实际上，我们对于这没有任何限制，我们可以轻易的举出一个例子。

```
198.51.100.96/28
```

在这个例子中:

```
            28 位网络位          | 4 位主机位
-------------------------------------+----
11000110 . 00110011 . 01100100 . 0110 0000
     198         51         100          96
```

我们只能用最后4位不同的数字标示该子网中的主机，其中`0000`和`1111`分别是保留地址和广播地址，这样在此划分中，我们拥有14个可用的主机编号。

比如，我们可以以主机号2（二进制`0010`）来填充最后4位:

```
            28 位网络位          | 4 位主机位
-------------------------------------+----
```

```
11000110 . 00110011 . 01100100 . 0110 0000
     198         51         100          96
```

这样，我们的得到的IP地址是 `198.51.100.98`。所有在`198.51.100.96`到`198.51.100.111`之间的IP地址都是该子网中合法的IP地址(虽然该范围内的第一个地址是保留地址，最后一个地址是广播地址)。

最后，如果你拥有一个自己的子网网段，你完全可以将其继续划分为更小的子网——只需要使用更多的位数来作为网络划分位。ISPs（因特网服务提供商，就像有线电视或者电话公司）就是一直在做这件事情。它们被赋予了一个大的子网网段，比如说，12位的网络位（这样就有20位的主机位，可以容纳将近100万的主机），然后一些客户想要从它们这里获取到子网段，ISP根据需求，决定将接下来的9位（打个比方）作为更小的子网网段，将其售卖给这些客户，这样每个客户可以自由使用剩下的11位作为自己的主机位（最大可以支持2048个主机）。

```
  ISP 网络      |   子网段    |    主机段
   (12 bits)   |  (9 bits)  |  (11 bits)
---------------+------------+--------------
11010110 . 1100 0101 . 11011 001 . 00101101   [example IP]
```

但是,该划分的过程远远不止只到这里，也许在这些你给他售卖了11位子网的客户想要更细的去划分它——他们可以增加更多的网络位来继续定义自己的子网。当然，每次增加更多网络位，将会获得更少的可分配的主机位，当然，这也是你根据实际情况需要做出的妥协。

子网掩码

另一种表示子网的方法是使用 *子网掩码*，它是一个数字，当与任何IP地址按位进行和运算时，你会得到你的子网号。那么这到底是什么意思，为什么会这样呢?

子网掩码也是用点-数形式来进行表示的，并且看起来很像一个将子网位全部置`1`的IP地址。举个例子，如果我们有一个子网段`198.51.100.0/24`，这意味着我们的子网段是这样的:

```
       24 网络位                | 8 主机位
-------------------------------+---------
11000110 . 00110011 . 01100100 . 00000000
     198         51         100          0
```

将所有网络位置`1`，我们可以得到如此一个模式:

```
       24 网络位                | 8 主机位
-------------------------------+---------
11111111 . 11111111 . 11111111 . 00000000
     255        255        255          0
```

所以，`198.51.100.0/24`的子网掩码是`255.255.255.0`，这时 *所有* `/24`的子网掩码。以此类推，将所有`/16`子网段前16位置1，可以得到对应的子网掩码`255.255.0.0`。

为什么会发明出这样的表示方法？因为路由器可以快速的通过简单的和运算来判断出一个指定IP地址的子网地址。所以子网掩码是`255.255.255.0`的IP地址`198.51.100.67`，其子网段是`198.51.100.0`。

```
        24 网络位                | 8 主机位
  -------------------------------+---------
  11000110 . 00110011 . 01100100 . 01000011    198. 51.100.67
& 11111111 . 11111111 . 11111111 . 00000000  & 255.255.255. 0
-----------------------------------------  ----------------
  11000110 . 00110011 . 01100100 . 00000000    198. 51.100. 0
```

已被淘汰的规定子网段

(此信息仅供历史参考。)

在为网络保留任意数量的位数之前，子网被划分为3个主要类别:

- **A类** - 子网掩码 `255.0.0.0` (或者说 `/8`), 支持 16,777,214 个主机
- **B类** - 子网掩码 `255.255.0.0` (或者说 `/16`), 支持 65,534 个主机
- **C类** - 子网掩码 `255.255.255.0` (或者说 `/24`) 支持 254 个主机

这种划分定义会导致了子网分布的不均匀，一些大公司有1600万台主机(虽然他们并不需要这么多)，但是没有子网类可以支持合理数量的计算机，比如1000台。所以，这才是为什么后面切换至更加灵活的“任意的子网位”尝试。

特殊的地址

有一些非常值得关注的特殊地址:

- **127.0.0.1** - 这个地址代表你正在操作的计算机，通常，它会被映射到主机名`localhost`。
- **0.0.0.0** - 保留地址，任何一个子网上主机号为0的地址都是保留地址。
- **255.255.255.255** - 广播地址，用来表示一段子网中的所有主机，虽然看起来这将广播到整个互联网，但是路由器不会真的转发数据包到这个地址。你可以通过将所有位置为1来将消息广播到本地子网段，比如，子网段`198.51.100.0/24`的广播地址是`198.51.100.255`。

特殊子网

你还可能会遇到一些有特殊意义的子网段:

- **10.0.0.0/8** - 私有网段 (*非常常见*)
- **127.0.0.0/8** - 通过 *环回* 设备的本地计算机
- **172.16.0.0/12** - 私有网段
- **192.0.0.0/24** - 私有网段
- **192.0.2.0/24** - 供说明使用，可参见
- **192.168.0.0/16** - 私有网段 (*非常常见*)
- **198.18.0.0/15** - 私有网段
- **198.51.100.0/24** - 供说明使用，可参见
- **203.0.113.0/24** - 供说明使用，可参见
- **233.252.0.0/24** - 供说明使用，可参见

在你家的网络设备中，通常你会发现你的IP地址位于上述的一个“私有”地址范围，大概率是`192.168.x.x`。你可以使用上述供说明使用的网段来作为文档中使用示例IP地址（非真实IP）。

思考题

- 为什么`192.168.262.12` 是一个非法的网络地址?
- 思考下子网概念作为划分全局地址空间的方式的一些优点。
- 当前你的电脑的IPv4地址是什么，子网掩码又是什么？（可能你需要学习下在你的操作系统中如何查看这俩信息）
- 如果你看到10.37.129.212/17，这其中有多少位是主机位?

因特网协议版本6（IPv6）

这是一个新时代的方案!因为能用32位表示的地址太少了(不包括保留的地址，只有4,294,967,296个)，互联网规则的决定者们觉得需要一个新的定址方案，一个使用更多位数，可以涵盖一切意图和需求并且能一直使用的方案。

随着网络的发展，我们已经遇到一个问题: IP地址不够用了。溯洄到1970年代，一个拥有数十亿计算机的世界是超出想象的，但是，时至今日，世界上的计算机数量已经远远超过这个维度。所以，互联网规则的制定者们决定将IP地址从32位增加到128位，这样将会有 79,228,162,514,264,337,593,543,950,336的地址空间，这可以足够被使用一个相当相当长的时间。

> 虽然在这个地址空间中也有很多保留地址，但是这个位数，仍然有**很多**地址空间，无论你用什么角度去查看。

而这是IPv4和IPv6的一大不同点。

出于演示目的，我们将继续使用IPv4，因为它依然是最常见的IP地址，并且更易于编写。但需要知道的是，总有一天IPv6将成为唯一的地址表示方法。

简介

基于IPv6有那么多的地址空间，光用点和十进制数字是不够的，所以它们想到了一种新的表示IPv6地址的方法:冒号加上十六进制数字。由于每一个每一个十六进制数字有16位，所以我们需要用8个这样的数字来表示128位。比如说:

和IPv4一样，使用斜线来表示子网，比如下面这个地址表示64位网络位(用`/64`来表示)和64位主机位(因为`128-64=64`)

拥有64位来表示主机号！这意味着在该子网段，可以拥有18,446,744,073,709,551,616 台主机！正如所示，用IPv6可以表示很多地址！

> 当我们讨论特定主机的标准IPv6地址时，`/64`是一种强力推荐的规则来说明你的子网有多大，甚至一些协议依赖于此。 但当我们讨论子网时，你可能会看到较小的数字表示较大的地址空间，但是最终都会期望于将该空间地址继续划分进入`/64`子网段以用来装填单独的主机。

很明显，编写所有这些十六进制数字可能会显得有点笨拙，特别是如果其中有大量的零，所以有一些简写规则。

1. 在一个16位数段中，可以移除开头的位为0的值。
2. 在应用规则1之后，如果出现多个连续的零，可以用两个连续的冒号替换。

让我们来看一个实例，假设我们有如下一个地址:

```
2001:0db8:6ffa:0000:0000:00ab:98bf:070a
```

首先，让我们运用规则一来移除先头的0:

```
2001:db8:6ffa:0:0:ab:98bf:70a
```

现在，我们看到了中间有两段是连续的`0`，根据规则二我们可以直接将其替换为两个冒号:

```
2001:db8:6ffa::ab:98bf:70a
```

按照这种规则，我们得到了一个更加简洁的表示。

链路本地地址

[这是那种“最好能了解下”的支持，但简单翻译就是“IPv6会自动给所有接口分配一个IP地址”。]

无论是IPv4还是IPv6，都会有一些保留地址给特定的局域网。这些地址通常不会被IPv4使用，但是在IPv6中，它们是有用处的，这些地址全部位于子网段`fe80::/10`。

> 将它展开，就是这样:
>
> ```
> fe80:0000:0000:0000:0000:0000:0000:0000
> ```

这里前10位是网络部分，在IPv6链路本地地址中，接下来的54位被保留(`0`)，然后剩下64位用来标识主机。当一个IPv6接口被激活时，它会根据以太网地址和其它相关的信息自动计算它的链路本地地址。

链路本地地址在局域网中是唯一的，但是不一定是全局唯一的，路由器将不会将任何链路本地地址的数据包转发出局域网，这样可以避免出现重复IP的问题。 例如，如果DHCP服务器分发一个IP，网络接口可能稍后获得不同的IP，在这种情况下，它将有两个IP。

特殊的IPv6地址以及子网段

如同IPv4一样，有一些地址拥有特殊的含义。

- **`::1`** - 本地地址，当前计算机，IPv6版本的`127.0.0.1`
- **`2001:db8::/32`** - 用于在文档中作为示例地址段
- **`fe80::/10`** - 链路本地地址段

这些IPv6的范围拥有特殊的意义，不过这些都是常见的地址范围。

IPv6 和 DNS

同样，在IPv6中，DNS将IPv6地址转化为人类可读的名称，你可以使用`dig`命令来查找`AAAA`记录（被DNS称之为IPv6地址记录）。

```
$ dig example.com AAAA
```

```
; <<>> DiG 9.10.6 <<>> example.com AAAA
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 13491
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
;; QUESTION SECTION:
;example.com.           IN  AAAA

;; ANSWER SECTION:
example.com.        81016   IN  AAAA    2606:2800:220:1:248:1893:25c8:1946

;; Query time: 14 msec
;; SERVER: 1.1.1.1#53(1.1.1.1)
;; WHEN: Wed Sep 28 16:05:16 PDT 2022
;; MSG SIZE  rcvd: 68
```

可以在上面`ANSWER SECTION`看到`example.com`的IPv6地址。

IPv6 与 URL

由于URL使用`:`来分割端口号，这造成了它和IPv6中的`:`的意思会发生冲突。假设你的服务跑在端口33490上，你可以在你的网页浏览器中输入IPv6地址，但将其放入中括号中。比如，可以用如下字符串去连接一个跑在本地`::1`地址上的该服务:

```
http://[::1]:33490/
```

思考题

- IPv6相对于IPv4的好处是什么?
- 如何将`2001:0db8:004a:0000:0000:00ab:ab4d:000a`写的更加简洁?

项目：一个更好的的网站服务器

现在，我们可以升级我们的网站服务器，使其可以传输真实的文件！当一个网页客户端（在本项目中，我们将会使用网页浏览器）请求一个特定的文件时，该服务端程序会将文件返回给客户端。在实现的过程中，你会学会相当多有趣的细节。

限制

为了能够更好的理解和学习底层套接字API的知识，在本项目中将**不会**使用以下辅助函数:

- `socket.create_connection()` 函数
- `socket.create_server()` 函数
- 任何`urllib`中的模块

当完成此项目时，对于这些辅助函数如何实现将会有更清晰的理解。

实现过程

如果你在浏览器中键入如下URL(将端口号替换你正在运行的服务程序):

```
http://localhost:33490/file1.txt
```

客户端程序（浏览器）会给你的服务端程序发送一个如下的请求:

```
GET /file1.txt HTTP/1.1
Host: localhost
Connection: close
```

请注意，在这个`GET`请求中，第一行里面的文件名！你的服务器程序将会:

1. 解析这个请求头，以拿到这个文件名。
2. 出于安全考虑，将路径剥离。
3. 从该文件中读取数据。
4. 判断文件中的数据类型，是HTML还是文本。
5. 在负载中使用文件数据建立HTTP应答数据包。
6. 将HTTP应答发送回客户端。

该应答如下所示:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 357
Connection: close
```

```
<!DOCTYPE html>

<html>
<head>
...
```

[在该例子中，HTML文件其余部分已经被截断省去。]

此时，你的网页浏览器应该可以显出这个文件。注意，在应答消息头中，有一些字段需要被计算和判断:`Content-Type`将会设置成保存的文件类型，`Content-Length`将会被设置成数据的长度。在这个项目中，在本项目中，我们会实现至少显示两种文件:HTML和文本文件。

解析请求头

在解析数据之前，你的服务器端程序需要读取完整的请求头，所以这里你需要有一些设计。比如使用一个变量中保存所有从`recv()`接收到的数据，并使用string的 `.find()`函数对其进行搜索，以找到标记头结束的`"\r\n\r\n"`。 此时，你可以以`"\r\n"`为参数对消息头中的数据使用`.split()`，这样可以得到独立的每一行。

第一行是`GET`，对于这一行，你可以使用`.split()`函数来将其分割成三部分:请求方法(`GET`)，请求路径（比如`/file1.txt`），以及协议（`HTTP/1.1`）。别忘了对请求的第一行使用`.decode("ISO-8859-1")`，这样你就可以得到一个供处理的字符串，目前，我们确实只需要这个路径。

从路径中分离出文件名

安全隐患 如果我们不删除其中的路径，恶意攻击者可能会使用它来访问您系统上的任意文件，比如他们可能会建立一个URL试图读取`/etc/password`。真实的web服务器一般检查路径是否被限制在某个目录层次结构中，但在本项目中我们采用简单的方法，去掉所有的路径信息，只提供web服务器运行所在目录中的文件。

由于路径将是用斜杠(`/`)分隔的目录名组成，因此对于此最简单的做法是在路径和文件名上使用`.split('/')`，然后取最后一个元素:

```
fullpath = "/foo/bar/baz.txt"

file = fullpath.spl
os.path.split("/foo/bar/baz.txt")
```

该函数将会返回一个含有两个元素的元组，其中第二部分是我们需要的文件名:

```
('/foo/bar', 'baz.txt')
```

使用该文件名去获取你想要取得的文件。

MIME 以及取得`Content-Type`内容

在HTTP中，负载可以是任何东西--任何字节的集合，那么，网页浏览器是怎么知道怎么去展示它呢？ 答案就藏在字节头里面的`Content-Type`里面，在MIME对于数据的类型有详细定义。这些信息对于客户端足够知道怎样去显示它。下面是一些MIME类型的实例:

MIME 类型名	描述
`text/plain`	明文文件
`text/html`	HTML 文件
`application/pdf`	PDF 文件
`image/jpeg`	JPEG 图像
`image/gif`	GIF 图片
`application/octet-stream`	未分类字节流

除此之外，还有很多MIME类型对应很多不同类型的数据，你只需要将正确的HTTP应答放在`Content-Type`头字段。

```
Content-Type: application/pdf
```

但是你怎么知道一个文件含有的是什么数据呢？比较通用的方法是查看文件的的后缀名，就是文件名点号之后的所有部分。幸运的是，函数`os.path.splitext()`可以方便的让我们从文件名中解析出最后一部分。

```
os.path.splitext('keyboardcat.gif')
```

返回一个元组包含:

```
('keyboardcat', '.gif')
```

对于下面的后缀部分，你可以直接做如下映射:

后缀	MIME 类型
`.txt`	`text/plain`
`.html`	`text/html`

所以如果一个文件拥有`.txt`后缀，确保发送的回复中是:

```
Content-Type: text/plain
```

如果你想万无一失，在消息头中加入`charset`来指定字符编码:

```
Content-Type: text/plain; charset=iso-8859-1
```

不过这不是必须的，因为一般来说浏览器默认会设置上面的字符编码方式。

读取文件， `Content-Length`和处理未发现

下面是一段读取整个文件并且检查错误的示例代码:

```
try:
    with open(filename, "rb") as fp:
        data = fp.read()   # Read entire file
        return data

except:
    # File not found or other error
    # TODO send a 404
```

从`.read()`返回的数据将会成为负载，并且可以使用`len()`来计算字节的长度，字节的长度将会包含在`Content-Length`头部并且被送回，比如(包含你文件中的字节数):

```
Content-Length: 357
```

> 你可能对于`open()`函数中的`"rb"`参数感到好奇，它表示使用二进制模式读取文件。 在Python中，一个以二进制模式打开读取的文件将会返回一个可以通过套接字直接发送二进制字节字符串。

如果遇到了`404 Not Found`怎么办？它很常见，你可能在日常的网络访问中不时会看到它。它表示你正在请求的文件或者其他资源并不存在，在我们的例子中，我们将会检测到一些这样文件打开时的错误(位于上面的`except`块中)，此时，它将会返回`404`。`404`是一种HTTP应答值，除此之外还有:

```
HTTP/1.1 200 OK
```

我们的应答将会以此开始

```
HTTP/1.1 404 Not Found
```

所以当你尝试去打开一个文件但是失败的时候，你应该返回如下(一字不差的)信息并且关闭连接:

```
HTTP/1.1 404 Not Found
Content-Type: text/plain
Content-Length: 13
Connection: close
```

```
404 not found
```

(在这种情况下，内容长度和有效负载都可以硬编码,当然需要对原始字节调用`.encode()`函数。)

附加题

如果你有时间，可以试试下面额外的挑战来更好地理解本章，加油!

- 增加对其他文件类型的MIME支持，使得程序可以支持JPEG图像和其他文件。
- 增加显示文件夹列表的功能，如果一个用户不在URL中指定一个文件名，显示一个指向每个文件名的文件列表。 一些提示: `os.listdir` 和 `os.path.join()`
- 相较于简单粗暴的放弃整个路径，支持请求服务器上根目录下的指定子路径。 **安全风险!** 请确保程序不会通过路径中一连串的`..`来损坏根目录！

  通常，你可能会通过配置变量将服务器根目录指定为绝对路径。但如果你上的时我的课，在我来批改你的作业的时候就惨了。因此，如果是这种情况，请为你的服务器根目录使用相对路径，并使用`os.path.abspath()`函数。

  ```
  server_root = os.path.abspath('.')        # This...
  server_root = os.path.abspath('./root')   # or something like this
  ```

  这将会把`server_root`设置成一个运行的服务程序的完整路径，比如，在我的机器上，我可能会得到:

  ```
  /home/beej/src/webserver                  # This...
  /home/beej/src/webserver/root             # or something like this
  ```

  然后，用户尝试去`GET`一些路径，你可以通过将其增加到服务程序运行路径之后来得到文件的全路径。

  ```
  file_path = os.path.sep.join(server_root, get_path)
  ```

  所以，如果客户端尝试去`GET /foo/bar/index.html`，将会得到的`file_path`为:

  ```
  /home/beej/src/webserver/foo/bar/index.html
  ```

  现在又到了安全问题！ 你需要确保`file_path`是位于服务器的根目录，不然，坏人可能会这么做:

  ```
  GET /../../../../../etc/passwd HTTP/1.1
  ```

如果它们真的如此，服务器端将无脑的请求如下文件:

```
/home/beej/src/webserver/../../../../../etc/passwd
```

这将获取到位于`/etc/passwd`中我的的密码，我可以不想发送这样的事情。所以我需要保证无论路径如何，最终都不会超出`server_root`层级，那么，怎么做到这一点呢？我们可以再次使用`abspath()`。

如果我通过`abspath()`疯狂对着以上的路径使用`..`，对于我它只会返回`/etc/passwd`，它将解析路径中类似`..`并且返回“真正的“地址。但是，我知道在这个例子中，我的路径根目录时`/home/beej/src/webserver`，所以，我可以将上面的相对路径赋予其后，并且返回404表示其并不存在。

```
# Convert to absolute path
file_path = os.path.abspath(file_path)

# See if the user is trying to break out of the server root
if not file_path.startswith(server_root):
    send_404()
```

示例文件

以下是一些可以直接复制拷贝到文件中作为返回测试文件的例子:

`file1.txt`

```
This is a sample text file that has all kinds of words in it that
seemingly go on for a long time but really don't say much at all.

And as that weren't enough, here is a second paragraph that continues
the tradition. And then goes for a further sentence, besides.
```

`file2.html`

```
<!DOCTYPE html>

<html>
<head>
<title>Test HTML File</title>
</head>

<body>
<h1>Test HTML</h1>

<p>This is my test file that has <i>some</i> HTML in in that the browser
```

```
should render as HTML.

<p>If you're seeing HTML tags that look like this <tt>&lt;p&gt;</tt>,
you're sending it out as the wrong MIME type! It should be
<tt>text/html</tt>!

<hr>
</body>
```

可以通过下面这些URL可以取到上述文件(使用正确的端口号):

```
http://localhost:33490/file1.txt
http://localhost:33490/file2.html
```

整数和字节序

我们已经学习了一些通过网络传输文本的方法，但现在想做点别的:我们想传输二进制整数数据。当然，我们*可以* 直接将这些数字转换为字符串，但是有点多此一举浪费资源了。二进制的表示方法更加紧凑并且可以节省带宽。但是计算机网络只能发送和接收字节数据！我们怎么才能将任意的数字转换为字节数据呢？这就是这一节想要阐述的。我们想要完成:

- 将整数转换为字节序列
- 将字节序列转换回整数

在本节中，将会学习到:

- 数字是怎样表示为字节序列
- 这些字节序列的顺序是什么样子的
- 在Python中怎样把一个数字转化为字节序列
- 在Python中怎么把字节序列转化为数字

需要注意的部分:

- 任意整数都能被表示为一个字节序列。
- 在网络上发送之前，我们将会把整数转换为字节序列。
- 在从网络中收到之后，我们会将字节序列再次转化为整数。
- 两种不同表示字节序列的方法，*大端序* 和 *小端序*。
- Python提供了内置的函数来将整数转化为字节序列并且将其转换回来。

字节表示法

在本节中，我们将深入研究如何用字节序列表示整数。

十进制字节表示

让我们看看整数是如何被表示为字节序列的，这些字节序列是我们将会通过网络发送给其他系统的整数数值。一个字节（在本节中，我们将一个字节等于8位）可以编码为`00000000` to `11111111`二进制数字，转化为十进制的话，表示从0到255的数字。

所以，如果当你想存储大于255的数字会发生什么呢？比如说256，在上述的规则中，你需要第二个字节去存储额外的数值。推而广之，越多的字节可以表示出越大的数。一个字节可以表示0到255，两个字节可以表示范围增大到0到65535。换个角度想，65536是16位比特能表示的数字的个数。

> 本节将只会讨论 **非负整数**。浮点数使用的是不同的方法。 负数使用和正数相似的方法来表示，不过为了简单起见，在本节中我们将暂时先不考虑它们。

让我们来看看用16位数字从253数到259的过程，因为259大于一个字节能表示的最大数，我们将用两个字节（一个字节可以表示0到255）来表示，下面是示例，相对应的十进制数在右边:

```
0 253    表示 253
0 254    表示 254
0 255    表示 255
```

```
  1   0   表示 256
  1   1   表示 257
  1   2   表示 258
  1   3   表示 259
```

注意，靠近右边的字节像里程表一样从255翻转"到0，就像在看一个以256为基数的编号系统。我们可以通过将第一个字节乘以256，然后加上第二个字节的值来得到相应的十进制数字:

```
1  * 256 +   3 = 259
```

或者在本例中，两个分别为17和178的字节组合，表示值1920:

```
17 * 256 + 178 = 1920
```

无论是17或者178都不大于255，所以它们都可以用一个字节表示。综上所述，所有的整数都可以完美的用一串字节表示，你可能需要在序列中使用更多的字节来表示更大的数字。

二进制表示法

二进制、十六进制和十进制都是表示数字的不同"语言"。因此，如果将上一节所有的十进制数字都替换为二进制，其中的数值和意思完全不会变化。为了验证这一点，让我们用上一节的例子来演示一下，请记住:这些数字完全相等——我们仅仅是把十进制转化为二进制，其他的所有概念都是完全相同的。

```
00000000 11111101    表示  11111101 (253 十进制)
00000000 11111110    表示  11111110 (254 十进制)
00000000 11111111    表示  11111101 (255 十进制)
00000001 00000000    表示 100000000 (256 十进制)
00000001 00000001    表示 100000001 (257 十进制)
00000001 00000010    表示 100000010 (258 十进制)
00000001 00000011    表示 100000011 (259 十进制)
```

但是让我们稍安勿躁，观察一下上面的模式，如果你把这两个字节放在一起，你会得到和二进制表示完全相同的数字!(忽略前置的0。)实际上，我们所做的就是把一个数字的二进制表示分成8位的块, 我们可以使用任意数，比如1,256,616,290,962，并将其转换为二进制:

```
1001001001001010000101010111010101001001 0
```

然后按照8位为一组，将其均匀的分隔开:

```
1 00100100 10010100 00101010 11101010 10010010
```

犹豫我们需要将其装入一个个比特中，我们需要在1之前补上0，最终如下所示:

```
00000001 00100100 10010100 00101010 11101010 10010010
```

现在你将得到最终的字节表示的数字结果1,256,616,290,962。

十六进制表示法

再次强调，使用什么进制并不是最重要的——它们只是表示数值的不同“语言”。程序员喜欢十六进制，因为它与字节非常的匹配（任何一个字节都是2个十六进制数字）。让我们再次以之前的例子作为说明，这次使用十六进制:

```
00 fd    表示 00fd (253 十进制)
00 fe    表示 00fe (254 十进制)
00 ff    表示 00ff (255 十进制)
01 00    表示 0100 (256 十进制)
01 01    表示 0101 (257 十进制)
01 02    表示 0102 (258 十进制)
01 03    表示 0103 (259 十进制)
```

再仔细观察一遍!十六进制表示的数字与刚刚挤在一起的两个字节相同!真的是非常的方便!

字节序

准备好继续学习新知识了吗?在上一节中，我们刚刚了解到一个数字可以被表示位，比如(十六进制):

```
45f2
```

也就是两个字节:

```
45 f2
```

但是你知道吗？在一些系统之，会将其表示为`0x45f2`:

```
f2 45
```

顺序完全颠倒了过来！这就好比我说“我想要123片吐司”，而实际上我想要321片!世界上对于这种颠倒放置字节的过程有一个专门的名称，我们称之为“小端序”。它表示的意思是数字的“小端”（书写序在后面的字节）位于整个字节序列的前端。

而相对的，更加正常化，更加便于理解的书写方法（就像我们最开始那样，数字`0x45f2`就是代表顺序`45 f2`）被称之为 *大端序*。 逻辑上位于前面的字节（也被称之为 *最高有效字节* ）就放置在前面。

坏消息是事实上所有的Intel CPU都是小端序机器，好消息是Mac M1都是大端序机器。 不过最棒的还是 **所有在网络上传输的数字都是大端的**，使用的是一眼能理解的方式:

> 但我说“全部”的时候，我一般指的的是“一定的数量的”[^—Monty Python]。但是假使 两端都同意使用小端序来传输数据，并没有严格的规定不能这样做。这在发送端和接收端 都是小端序架构的机器中是很合理的-为什么去浪费时间将字节序列翻转发送之后再翻转 解析呢？不过绝大部分的协议都是大端序的。

在本章的关于网络概念的上下文中，大端序被称之为 *网络字节序* 。

Python和字节序

那么如何在Python中将数字转换成字节序列呢?幸运的是，Python中有一个内置的函数`.to_bytes()`可以完成这个工作，同时，还有一个反过来的从字节序列组成数字的函数`.from_bytes()`。这两个函数甚至可以运行用户来指定需要使用字节序！因为本文中都是通过网络传输数据，所以对于这两个函数我们永远使用的是`"big"`序列。

将数字转换为字节

下面我们通过一个具体的例子来展示如何将数字3490转换为一个两字节的大端序的字节数组。请注意，我们需要给`.to_bytes()`函数传入两个参数:需要转换的数字和字符串`"big"` 来表示我们需要大端序字节序列，或者`"little"` 来表示需要的是小端序序列。

> 新版本的Python会将默认值设置为`"big"`，不过在旧版本中，用户仍然需要显示的来指定这个参数。

如果我们将其打印出来，我们会看到这些字节的具体值:

```
for b in bytes:
    print(b)
```

```
13
162
```

这些是组成数字3490的大端字节序列值，我们可以简单的验证下，`13 * 256 + 162 == 3490`。

如果你尝试把数字70,000装入两个字节中，你将会得到`OverflowError`，两个字节不足以能够存储大于65535的数字——这里需要再增加一个字节。让我们来再看几个十六进制的例子:

```
n = 0xABCD
bytes = n.to_bytes(2, "big")

for b in bytes:
    print(f"{b:02X}")  # Print in hex
```

将结果输出:

```
AB
CD
```

你会发现，结果中的数位和变量`n`中存储的原始值一摸一样！

将字节转换回数字

让我们来完成一个完整的转换周期，之前我们是把一个十六进制数字转化为字节序例，并且将其打印出来看看最终的结果，下面，让我们把字节字符串转换回数字并且打印出来，以验证得到的结果和最开始的数字是一样的。

```
n = 0x0102
bytes = n.to_bytes(2, "big")

print(bytes)
```

可以看到其输出为:

```
b'\x01\x02'
```

最前面的`b`表示这时一个字节字符串(以便于区分于普通的字符串)，而`\x`是对于2位十六进制数字的转义符。由于我们的原始数字是`0x0102`，所以在字节字符串中的两个字节分别是`\x01` 和 `\x02`。现在，让我们将字符串转换转换回来并且以十六进制打印:

```
v = int.from_bytes(bytes, "big")

print(f"{v:04x}")
```

我们可以看到结果是:

```
0102
```

和原始的数值完全相同！

思考题

- 仅仅使用`.to_bytes()`和`.from_bytes()` 函数如何交换一个2字节数字的字节顺序？（也就是翻转字节。）思考下如何在不用循环和其他方法的情况下完成该操作（提示:`"big"` 和 `"little"`!）
- 用你自己的语言描述下大端字节序和小端字节序的差别。
- 什么是网络字节序?

- 为什么不直接传送整个数字而是要将其分解成为字节?
- 小端序是一种和人认知相反的表示方法，那么为什么会存在这种字节序？你可以搜集一些资料来了解下该问题的答案。

数据包解析

在前面我们已经遇到了一些在服务器上接收结构化数据的问题，比如，当你调用 `recv(4096)`，但是你实际上你只接收到了20个字节。或者同样调用`recv(4096)`得到的数据却大于4096个字节，而导致你需要再次调用该函数。

实际上还有更加严重的问题，如果一个服务器给你发送了好几段的数据，你可能只接收到第一或者下一段的一部分，你将会收到一个完整的数据包加上一个部分完整的数据包，这种情况下，你如何能够重构原始数据包呢?

打个比方，如果我需要你从我给你的一大块文本中分离出单独的句子，但你一次只能得到20个字符。你调用 `recv(20)` 看到如下的字符串:

```
This is a test of th
```

这不是原始完整的句子，此时你还不能将其打印出来，因此，你需要再次调用`recv(20)`:

```
This is a test of the emergency broadcas
```

仍然不是一个句子，让我们再次调用:

```
This is a test of the emergency broadcast system. This is on
```

看！结果中出现了一个句号，所以我们得到了一个完整的句子。我们以句号为分割符将其打印出来。但是，同时我们也收到了下一句的部分信息！那么，我们怎么才能优雅的处理整个过程呢?

你知道怎么能让这事简单点吗?

你知道怎么能让这事简单点吗?如果我们把它抽象出来，然后我们可以使用如下的逻辑:

```
while the connection isn't closed:
    sentence = get_next_packet()
    print(sentence)
```

有没有根据这个逻辑发现处理思路呢？一旦我们的代码的逻辑可以从数据流中提取出下一个完整的数据包，我们就可以使用它。如果程序的逻辑足够的完备，那么事实上它可以从数据流中提取出不同类型的数据包,比如:

```
packet = get_next_packet()
```

```
if packet.type == PLAYER_POSITION:
    set_player_position(packet.player_index, packet.player_position)

elif packet.type == PRIVATE_CHAT:
    display_chat(packet.player_from, packet.message, private=True)
```

这使得无论收到的数据包是不是完整的字节序列都变得容易很多，当然，处理的过程还有很多细节，让我们来思考下如何完成整个过程。

将流处理成数据包

使得整个处理逻辑能工作的一个最大的秘密就是:申请一个大的全局缓冲区。

> 一个缓冲区是一片大量字节存储区域的另一种叫法。 在Python中，它可以是一个字节字符串，也就是可以方便的从`recv()`取得的返回值。

这个缓冲区将保存到目前为止你收到的字节，你需要检查缓冲区来确认它是否保存了一个完整的数据包。如果在缓冲区中已经有了一个完整的数据包，程序需要将其返回（以字节字符串或者其他处理过的形式）。并且，非常重要的是，需要在缓冲区中删除该数据包。如若不然，你需要再次调用`recv()`函数直到缓冲区中拥有一个完整的数据包。

```
packet_buffer = b''

def get_next_packet(s):
    global packet_buffer

    # Now we can use the global version in here
```

不然，Python会在函数中创建一个新的本地变量而掩盖了全局的那个变量。

再来看看之前的“句子”示例

让我们再来研究下本章开头的句子的示例，我们将会调用`get_sentence()`函数，它的作用是收到所有数据并且查看其中是否有句号，一开始我们拥有的是:

```
```

什么也没有，因为我们还没收到数据，理所当然，其中也不会有句号，所以我们也不会得到一个完整的句子。接下来，我们需要调用`recv(20)`来获取更多的字节:

```
This is a test of th
```

依然还没以欧句号，再次调用`recv(20)`:

```
This is a test of the emergency broadcas
```

还是没有句号，再次调用`recv(20)`:

```
This is a test of the emergency broadcast system. This is on
```

阿，终于有了一个句号！所以我们需要做以下两件事情:

1. 将句子提取出来，这样我们可以返回该结果，并且
2. 将提取出来的句子从缓冲区中删除。

在上述两步完成之后，第一句应该已经从缓冲区中消失，并且:

```
This is on
```

程序将会返回第一句话"This is a test of the emergency broadcast system."，而我们调用的`get_sentence()`函数可以打印出该句话。

接下来，让我们再次调用 `get_sentence()` ,在`get_sentence()`中，我们会再一次的去查看缓冲区（注意，这里的缓冲区是全局的，所以在其中仍然会有自从上次调用者之后剩下的数据。）

```
This is on
```

目前，这个剩余的数据中并没有句号，所以我们将会再次调用 `recv(20)` ,不过这次，我们只能取回10个字节。

```
This is only a test.
```

而这已经是一个完整的句子了，我们将其从缓冲区中取出，导致缓冲区清空，程序返回至调用方等待输出。

如果一次性收到了多句话会怎样?

如果当我调用 `recv(20)`时得到了如下的结果:

```
Part 1. Part 2. Part
```

你看，这仍然能工作，`get_sentence()` 函数可以看到数据中的第一个句号，继而从缓冲区中剥离第一句话，从而我们得到:

```
Part 2. Part
```

此时会返回 `Part 1.`。

下一次继续调用 `get_sentence()`时，和之前一样，它做的第一件事是去查看缓冲区中是否还有一个完整的句子，在发现确实仍然还有之后，将其剥离出来。

```
Part
```

继而返回`Part 2.`。当下次再次调用`get_sentence()`时，函数发现在缓冲区中没有了句号，所以，目前已经没有完整的句子了，函数会继续调用 `recv(20)`以取得更多的数据。

```
Part 3. Part 4. Part 5.
```

现在，重新组成了完整的句子，按照之前的逻辑，将其前面的数据从缓冲区剥离:

```
Part 4. Part 5.
```

继续返回`Part 3`返回给调用者，继而复始。

进一步的优化

总体来说，你可以将此抽象为一个充满数据的管道，当管道中充满了数据包时，程序会从缓冲区的前端提取出这些数据并且返回。但是如果没有足够的数据的话，管道会接收更多的数据置于缓冲区的后部，并且保持监测是否已经收到了一个完整的数据包，如下时这个过程的伪代码:

```
global buffer = b''    # 空字节字符串

function get_packet():
    while True:
        if 缓冲区已经收到一个完整数据包
           提取数据包数据
           从缓冲区前部提取出数据
           返回该数据包数据

        接收更多的数据

        if 收到的数据是0字节
           返回连接已关闭的标示

        将接收到的数据置于缓冲区后部
```

在Python中，你可以分割缓冲区以从前端删除数据包数据，举例来说，如果你知道一包数据是12字节，你可以用以下代码将其删除:

思考题

- 从编程的角度思考下从数据包中抽取出数据流的好处是什么。

项目：原子时间

在本项目中，你将要去访问获取NIST(美国国家标准与技术研究所)的原子钟，从他们的时钟上得到自1900年1月1日以来的秒数(这将是很多秒。)，并且将其输出。

于此同时，该项目还会输出你计算机上的本地系统时间，如果你计算机上的时钟是准确的话，这两个数字的输出应该非常接近:

```
NIST time  : 3874089043
System time: 3874089043
```

本项目中，我们只是在写一个客户端,服务端程序已经存在并且时刻在运行。

法律声明

NIST的原子钟服务免费面向公众开放。但是一般来说，服务的开发者不希望未经授权的用户随意访问其服务器，有时，这可能是一个触犯法律的捷径。但是在本项目中，原子钟服务是开放并且欢迎公众使用的。

可用性声明

查询NIST服务器的次数不能超过每4秒一次，如果超出这个频率，它们会开始禁止程序进行查询。如果你想要周期化的运行该程序，请确保两次访问之间大于4秒。你可以在命令行中使用`sleep`来保证这一点。

```
sleep 4; python3 timeclient.py
```

纪元

在计算机用语中，“纪元”指的是从计算机视角的“时间的开始”。许多库都以“自epoch以来的秒数”来测量时间，意思是自时间开始以来的时间。我们所说的时间之初是什么意思?这要分情况来描述，在Unix世界中，时间开始的定义是非常具体的1970年1月1日00:00 UTC(又名格林威治标准时间)。

其他的epochs，时间的开始可能是其他的时间。比如，在时间协议中，时间的开始被定义为1900年1月1号的0点（UTC时间），而这是Unix所规定的开始时间的70年前。而这些意味着，我们需要做一些转换，幸运的是，我们将会直接给出返回正确数值的代码，而你完全不需要担心这些。

关于零的可怕警告

我也不知道什么原因，有时候NIST服务器会返回四个字节的全0字节，而有时，它将会发送0字节然后关闭连接。如果这件事发生了，你大概率会看到在NIST时间里出现了`0`。这时，只要尝试再次运行你的客户端，看看你是否在一次或两次尝试后获得良好的结果，请记住每4秒查询一次的限制。对于`time.nist.gov`，背后有一些轮换的IP，并且很可能在某一天，一两个服务器不能正常工作。不过如果，如果服务器一直返回0，那么大概率是有其他的错误了。

整体方案

对于这个项目，下面是整个的程序逻辑:

1. 在端口`37`上连接`time.nist.gov`（时间协议）。
2. 接收数据（客户端不需要发送任何内容。），客户端应该能收到4字节的回复。
3. 这4个字节是一个4字节大端序数字，使用`.from_bytes()`将这4字节解析成一个数值。
4. 将这个从时间服务器返回的数值打印出来，这个数字是从1900年1月1日0点0分经过的秒数。
5. 获取系统本地时间，将其转换为以从1900年1月1日0点0分经过的秒数进行输出。

如果你的电脑时钟是准确的，这两个时间应该大致(或完全)一致。这个数字应该比38.7亿多一点，而且它应该每秒增加一次。

1. 连接服务器

时间协议通常与UDP和TCP一起工作。对于这个项目，就像我们之前其他项目一样，必须使用TCP套接字。所以创建一个套接字，并且通过时间协议在`37`端口连接 `time.nist.gov`。

2. 接收数据

技术上，你应该使用一个循环来完成此步骤，不过，一般来说，如此小量的数据一般不会被分割为很多数据包进行传输。无论你要求一次收到多少数据，最多你都只会收到4个字节的数据，当接收完数据之后，你可以关闭该套接字。

3. 解析数据

接收到的数据是一个被编码在大端4字节中的数字。调用之前我们提到的`.from_bytes()`函数以从`recv()`字节流中解析出该数值。 The data is an integer encoded as 4 bytes, big-endian.

4. 打印NIST时间

得到的时间应该是这样的格式:

```
NIST time  : 3874089043
```

5. 打印系统时间

下面是一段Python代码，它用来获取并打印自1990年1月1日0时0分系统时钟，你可以简单的将其粘贴进你的代码来直接获取系统时间。将系统时间以以下的形式直接输出在NIST时间之后:

```
System time: 3874089043
```

这里是相应的代码:

```
import time

def system_seconds_since_1900():
    """
    The time server returns the number of seconds since 1900, but Unix
```

```
    systems return the number of seconds since 1970. This function
    computes the number of seconds since 1900 on the system.
    """

    # Number of seconds between 1900-01-01 and 1970-01-01
    seconds_delta = 2208988800

    seconds_since_unix_epoch = int(time.time())
    seconds_since_1900_epoch = seconds_since_unix_epoch + seconds_delta

    return seconds_since_1900_epoch
```

假设NIST返回的数字不是0:

- 如果这个数字与NIST的数字相差不到10秒，大功告成。
- 如果相差在86,400秒内，也没有关系，我很想知道，因为这可能是上面代码中的一个bug。
- 如果相差在1百万秒内，那么我很想你能告知我。
- 如果差距大于上面的数字，那么大概率是你的代码中有bug。你是否使用了"大端"字节序？你时候接收了完整的4个字节？

项目: 单词服务器

本项目主要是介绍如何读取数据包。你需要从服务器(提供的)接收编码数据流，然后编写代码来确定何时收到了完整的数据包并打印出该数据。

综述

准备工作 下载这些文件:

- [fls[`wordserver.py`|word/wordserver.py]]: 一个可以随时运行分发随机单词列表的服务器。
- [fls[`wordclient.py`|word/wordclient.py]]: 客户端程序框架。

限制条件！请不要更改下面任何已经存在的代码！ 只需要搜索`TODO`并填入代码，你可以按照你的需要增加变量和函数。

要求！代码必须能够处理传给`recv()`的从`1`到`4096`之间任何正值！你可以使用`1`, `5`, 和 `4096`来测试代码，确保它们都能正确工作。

要求！代码必须能够处理长度1到长度65535的单词。 该服务端程序不要求处理特别长的单词，不过你可以修改代码来进行测试。你可以使用如下方法来建立一个固定长度字母的Python字符串:

```
long_str = "a" * 256   # Make a string of 256 "a"s
```

高级技巧！在开始之前，阅读并理解所有现有的客户端代码。这将是你免于陷入各种困境。请注意，代码的核心部分甚至不关心字节和流——它只关心整个数据包。简洁易懂，是不是?

你需要完成两个函数:

- `get_next_word_packet()`: 从数据流中返回下一个完整的单词数据包，注意，这里需要返回一个 *完整的数据包*, 包括数据头和数据本身。
- `extract_word()`: 从一个完整的单词数据包中提取并返回一个完整的单词。

除此之外，该程序还做什么？请继续往下看！

什么是“单词数据包？”

当你连接到服务器时，它将会给你发送一个数据流。该流由一个随机表示单词的长度的数(1到10)的前缀(以字节为单位)加上单词组成。每一个单词都是UTF-8编码。 单词长度被编码为一个大端字节序的2字节的数字。举例来说，长度为5的的单词"hello"被编码为如下字节流(十六进制表示):

```
length 5
  |
+-+-+
|   | h  e  l  l  o
00 05 68 65 6C 6C 6F
```

和字母对应的数字都是字母的UTF-8编码。

> 有趣的事实: 对于英语字母表中的字母和数字，UTF-8, ASCII, 和ISO-8859-1编码都是一样的。

单词"hi"和"encyclopedia"会被编码成两单词的数据包，其在数据流中进行传输，如下所示:

```
length 2   length 12
  |           |
+-+-+       +-+-+
|   | h  i  |   | e  n  c  y  c  l  o  p  e  d  i  a
00 02 68 69 00 0C 65 6E 63 79 63 6C 6F 70 65 64 69 61
```

实现: `get_next_word_packet()`

本函数以已经连接的套接字作为参数，将返回下一个单词数据包（单词长度加上单词本身，和所收到的一样）作为一个字节字符串。其从字节流中解析数据包的过程主要和解析数据包的框架大体一致。

举例来说，如果收到的单词是上面例子中的"hi"和"encyclopedia"，当我们接收到开头的5字节之后，字节缓冲区将会包含:

```
      h  i
00 02 68 69 00
```

我们看到开头的单词是2字节，并且我们已经获取了这2字节。接着，我们将会解析并且返回第一个单词("hi")和它的长度（字节`00 02`），最终返回的字节字符串为:

```
      h  i
00 02 68 69
```

我们还会将这些字节从数据包缓冲区中剥离出来，这样就只剩下最后的0了。

```
00
```

在此刻，缓冲区中已经没有完整的单词了，随后对该函数的调用将为下一个数据块触发`recv(5)`,我们将会得到:

```
      e  n  c  y
00 0C 65 6E 63 79
```

如此反复。

实现: `extract_word()`

本函数接收一个完整的单词数据包作为输入，例如:

```
      h  i
00 02 68 69
```

而返回该单词的字符串，`"hi"`，此过程获取单词长度并且返回正确的单词字节。

不过请注意:获取的单词都是UTF-8编码的！所以你需要调用`.decode()`来将其转换为字符串（默认编码是UTF-8，所以你不需要向`.decode()`传递任何函数。）

传输控制协议(TCP)

如果有人开挖掘机挖断了光纤，网络上的数据包可能会丢失。（通过将船锚拖入海底网络电缆，整个国家都离线了!）软件错误、电脑死机和路由器故障都可能导致问题。但是用户不需要考虑这个，用户只需要一个部件来处理所有这些，然后让用户知道何时数据是完整的、完备的和有序的。

TCP就是这个部件，它会关心数据包丢失的问题而对用户完全透明，当它确信自己接收到了完整的数据包时，它会将数据包返回给我们。

在本节中，将会学习到:

- TCP的整体目标
- 在网络占中，它位于哪一层
- 回顾TCP中端口的概念
- TCP如何建立，使用和关闭连接
- 数据完整性机制
    - 维护数据包顺序
    - 监测错误
- 流量控制 -- 接收端如何避免不知所措
- 拥塞控制 -- 发送端如何防止网络过载

TCP是一个非常复杂的主题，我们在这里只略读其中的重点。如果你想更深入的学习，应该阅读的书籍是 *TCP/IP 图解卷一* (W. Richard Stevens)。

TCP的目标

- 提供一个可靠的通信机制
- 在数据包交换网络中模拟出类似电路连接的效果
- 流量控制
- 拥塞控制
- 支持带外数据

网络协议栈中的位置

先来回忆下网络栈中的层级:

层级	职责	协议举例
应用层	结构化的应用层数据	HTTP, FTP, TFTP, Telnet, SSH, SMTP, POP, IMAP
传输层	数据完整性，数据分割和组装	TCP, UDP
网络层	路由	IP, IPv6, ICMP
链路层	物理设备，线路上的电信号	Ethernet, PPP, token ring

你可以看到，TCP位于传输层，它底下的IP层负责路由，而它上面的应用层享受所有TCP所带来的好处。这就是为什么当我们编写HTTP客户端和服务器程序时，我们完全不需要担心数据完整性。我们会让TCP全权处理这些细节！

TCP端口

回忆下当我们使用TCP的时候，我们需要指定一个连接的端口号，即使是本端的客户端，操作系统也会自动赋予一个本地的端口号（如果我们没有主动调用`bind()`函数）。IP协议使用IP地址来表示主机。

但是，一旦连接到该主机上，操作系统就会使用端口号将数据传递给正确的进程。打个比方来说，IP地址就像街道地址，端口号就像街道地址上面的公寓门牌号。

TCP综述

一个TCP在连接中主要会做三件事情:

1. 建立连接
2. 传输数据
3. 关闭连接

这上面的每一项都包括在客户端和服务器程序间传输特殊的非用户数据数据包，我们将会详细学习这些特殊类型的数据包，比如SYN, SYN-ACK, ACK, 和 FIN.

建立连接

这就是非常著名的“三次握手”，由于任何数据包都可能在传输过程中丢，TCP在连接建立传输数据的连接成功之前会使用额外的步骤保证两端都建立上了连接。

1. 客户端发送一个SYN（同步）数据包给服务器。
2. 服务回复一个SYN-ACK（同步认证）数据包给客户端。
3. 客户端回复一个ACK（认证）数据包给服务器。

如果这三步中，如果在一定时间内没有收到回复，该数据包会重新发送。

发送数据

TCP接收数据流并将其分割成数据块，每一块数据都拥有一个和其关联的TCP报头，接着整个数据包会通过IP层将其发送出去。这个数据报头加上数据块被称之为TCP *片段* 。（我们将交替使用“TCP数据包”和“TCP段”，但段更正确。）

当TCP发送一个段时，它期望该数据的接收方响应一个确认消息，称为ACK。如果TCP没有收到这个ACK，它会认为发生了某些错误并且需要重传该段。每个段消息都会被编号所以即使它们没有按照顺序到达，TCP仍然可以将它们按照正确的顺序放回。

关闭连接

当某一边想要关闭连接时，它将会发送FIN（结束[*sic*]）数据包，远端此时一般会回复一个ACK和一个自己的FIN数据包，接着，本地端会以另外一个ACK来彻底关闭连接。在一些操作系统中，如果一个主机在还有未读数据的时候关闭了连接，它会发送会一个RST（重置）来表示这个情况。在这种情况下，套接字程序一般会打印“Connection reset by peer”。

数据完整性

在网络传输中，有很多可能会导致错误的事情，数据可能乱序到达，可能损坏，可能重复，甚至根本到达不了。TCP拥有处理所有这些意外事件的机制。

数据包顺序

发送端会在每一段中放置一个永远递增的序列号，比如“这一分段是3490号，这一分段是3491号”。接收端会回复一个带有发送端序列号的ACK数据包以表达“我收到了分段3490，我收到了分段3491。” 这样，如果有两段乱序到达，TCP仍然可以将它们按照顺序组装回来。

> 类比时刻！如果你有一叠文件，把它们扔向空中, 你如何能知道他们的原始顺序？如果，你正确的 用数字标示了每一页，你只需要将它们进行排序。 这就是TCP协议栈序列号的作用。

如果有重复的分段到达对端，接收端TCP知道该序号的分段已经被收到过了，所以可以安全的丢弃该分段。如果一个分段丢失了，TCP可以要求进行重传，它通过不停的给发送端发送上一个序号的分段的ACK消息来达到这一点，发送端将会再次重传下一个序号的数据。

或者，如果发送方在一段时间内没有收到特定段的ACK，它可能会认为该段可能已经丢失，发送端会重新传输该段。超时次数越多，这种重传的成功性变得越来越低;每次超时发生时，服务器端都会将超时加倍。

最后一点，在连接开始的三次握手中，序列号被初始化为随机值。

错误检测

在发送端发送出一个分段之前，一个基于该分段的 _校验和_将会同时被计算，当接收端收到这个分段时，它会基于收到的分段数据计算自己这一边的校验和。如果两边的校验和匹配，说明收到的数据是没有错误的。如果两次交验和不相等，数据将会被丢弃并且发送端会超时并且重传。

校验和是一个将所有TCP报头和有效负载数据以及所涉及的IP地址作为参数，计算出来的一个16位数字，在下面的项目中，将会有对校验和的详细介绍。

流量控制

流量控制 是一种让两个互相通信的设备告知对方数据需要发送更慢的机制，你不能直接向一个只能处理100Mbs设备倾倒1000Mbs(每秒百位数)数据，该设备需要告知发送端减缓速度。

> 类比时间：当你在电话里告诉对方的时候，你会这样说: "你说的太快了，我没听懂，请说慢一点!"

最简单的方法(这不是TCP所做的)是发送方发送数据，然后等待接收方发回带有该序列号的ACK数据包，接着发送端再发送一个数据包。按照这个思路，如果需要发送方放慢速度，接收方可以延迟ACK。

但这是一个缓慢的来回，网络通常足够可靠，发送方可以在不等待响应的情况下发送多个段。然而，如果我们这样做，我们冒着发送者发送数据的速度比接收者处理数据的速度快的风险!

在TCP中，采用一种叫做 *滑动窗口* 的机制来解决这个问题，该机制会使用接收方的ACK包的TCP报头的“窗口”字段。在该字段中，数据接收方将会指定还可以接受多少数据（以字节为单位），随后的它收到的ACK信息将会携带新的窗口信息。使用这种机制，接收端可以让发送端知晓，“一旦你发送了X字节，你必须等待一个ACK，告诉你还可以发送多少字节“。值得一提的是，这里指的是字节数目而不是段数目，发送方可以自由地发送多个段而不接收ACK，只要总字节数不超过接收方所告知的窗口大小。

拥塞控制

流量控制是在两台计算机之间运行的一种机制，但如果放眼整个互联网，你会发现一个更大的问题。如果路由器不堪重负，它可能会开始丢弃数据包，导致发送者开始重传，这对缓解问题没有任何作用。它甚至没有在流量控制的雷达上，因为数据包没有到达接收端。

在1986年，在NSFNET（商业互联网前身）上被不知道什么时候该停止重传的而一直坚持不懈发消息的发信人压垮了，吞吐量将至正常水平的0.1%，这不是一个很妙的事情。为了解决这个问题，TCP实现了一系列的机制来评估并且消除网络拥塞，请注意，这些都是在接收方发布的流量控制窗口之外的。发送方必须遵守流量控制限制**同时**计算的网络拥塞限制，以较低者为准。在网络上的未确认段数量不能超过此限制。如果超过了这个限制，它必须停止发送并等待一些ACK重新进入。

换个角度思考这个这个问题，当发送方发出一个新的TCP段时，它会增加网络拥塞。而当它重新收到一个ACK时，这表明该网段已从网络中删除，拥塞情况开始缓解。为了缓解在NFSNET遇到的该问题，加入了两个算法: 慢启动和拥塞避免。

请注意: 下面的介绍是这两个算法的一个简化版本，详细了解这俩之间复杂的相互作用，以及更多关于避免拥堵的方法，请查看*TCP Congestion Control* (RFC5681)。

慢启动

当连接第一次建立时，发送方是无法知道网络的拥塞程度。第一阶段的工作就是进行粗略的猜测。随意一开始发送端会谨慎的启动，假设在网络上有非常高级别的拥塞情况。（假设已经存在很高的拥塞水平，那么大量地向网络中填充数据将会雪上加霜。）

首先，它定义自己有一个初始 *拥塞窗口*，即允许有多少未被确认的字节(和段，但现在我们只考虑字节)未完成。当ACK收到后，拥塞窗口的大小随着已确认字节数的增加而增加。简单地描述，一个分段发送出去并收到确认之后，接下来可以开始尝试发送两个分段。如果这两个被也成功的收到了确认回复，那么可以一次性发送四个可分段。

所以，该机制会使用一个非常小的未被确认的分段数作为开始，一旦持续成功接收到回复消息，发送窗口会快速增长。最终如果线路中ACK丢失了，这时慢启动决定开始降速。它会将拥塞窗口大小减半，然后TCP切换到拥塞避免算法。

拥塞避免

该算法和慢启动类似，不过它会使用更多的控制步骤，不会是指数增长之类的方法。

每当一个拥塞窗口值的数据成功传输时，它会通过向窗口添加一个分段的字节来增加一点强度，该机制允许它在网络上拥有另一个未连接的段。如果回复正常，它允许该机制再次运行一次，如此反复。

因此，该机制主要是保持在不拥塞网络的情况下尝试发送最大限制的数据，但它的推进速度很慢。我们称之为 *加性增长*，它通常是线性的(相比之下，慢启动通常是指数的)。不过，这个机制触及了带宽上限，不得不重新传输数据包。此时，拥塞窗口被设置为较小的大小，算法退回到慢启动状态。

思考题

- 请列举几个依赖TCP实现数据完整性的协议。
- 为什么要用三次握手来建立连接?为什么不直接开始传输呢?
- 校验和事怎么保护数据完整性的?
- 流量控制和拥塞控制的不同点是什么?
- 思考下在慢启动和拥塞避免之间切换的原因。它们在拥塞检测的不同阶段有什么优势?
- 流量控制的目的是什么?

用户数据报协议（UDP）

如果你是一个喜欢保持事物整洁并且乐观的人，UDP非常适合你。它是一个互联网上近乎极致的轻量级数据传输协议。发送端发射UDP数据包并且祈祷它们回达到对方，也许它们会，也许有人用挖土机挖穿了光缆，或者有宇宙射线干扰，又或者路由器过于拥塞或愤怒，直接将其丢弃，或者各种随机原因。它生活在互联网传输数据的边界!几乎所有令人愉快和可靠的TCP保证都消失了!

UDP目标

- 提供一种将无差错的将数据从一台计算机发送到另一台计算机的方法。

这就是UDP所有内容。

下面这些都**不是**UDP的目标:

- 保证数据是有序的
- 保证数据不丢失
- 保证数据没有重复

如果需要上面的功能，TCP将会是更好的选择，UDP不提供任何对于数据丢失或者数据乱序的保障，它唯一的保障是 *如果* 数据抵达，那么该数据是准确的。UDP协议可以提供的是低开销和快速的响应时间，它没有如TCP一般的任何数据包装配或者流量控制或者数据包响应。因此，它的数据报头更加的小。

网络栈中的位置

回一下之前说的网络栈:

层级	职责	协议举例
应用层	结构化的应用层数据	HTTP, FTP, TFTP, Telnet, SSH, SMTP, POP, IMAP
传输层	数据完整性，数据分割和组装	TCP, UDP
网络层	路由	IP, IPv6, ICMP
链路层	物理设备，线路上的电信号	Ethernet, PPP, token ring

正如我们看到的，UDP位于传输层，在其之下的IP层负责路由，在其之上的应用层享受所有UDP应该提供的好处，虽然不是很多。

UDP 端口

类似于TCP， UDP使用端口。实际上，您可以让一个TCP程序与另一个UDP程序使用相同的端口号。 IP协议使用IP地址来标示主机。但是当我们缩小到主机范围，端口号是操作系统用来将数据发送到正确进程的媒介。举例来说，IP地址就像是街道号，端口号就像是该街道地址上的公寓门牌号。

UDP 总览

UDP 是 *无连接状态* 的，前面介绍过，TCP基于数据包转发网络来保证两个计算机之间建立的连接是可靠的。UDP并不保证如此。客户端向IP地址和端口发送UDP数据报，IP协议负责将其路由到指定地址，并且接收端

计算机将数据报发送到绑定到规定端口的程序上。

在这里，没有任何连接，只有单独的基于单个包的机制。当一个数据包到达时，接收端可以分辨出该数据包是从哪个IP和端口而来，基于此，接收端可以发送一个回复。

数据完整性

在网络传输中，有很多可能会导致错误的事情，数据可能乱序到达，可能损坏，可能重复，甚至根本到达不了。 UDP几乎没有任何方法可以处理这些意外事件，实际上，它唯一做的就是错误检测。

错误检测

在发送一个数据报之前，会根据数据包计算出一个 *校验和*。 校验和的作用和TCP完全一致，除了在这里使用的是UDP的包头，相对于TCP包头，UDP包头十分的简单:

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

当接收方收到这个数据包的时候，它会根据该数据包计算出校验和，如果两方校验和一致，那么该数据被认为是无差错的。如果他们不一致，那么直接丢弃该数据。

这就是整个过程，接收端甚至不知道有一些数据是发送给它的，数据包只是消失在网络中。校验和是一个将所有UDP包头和有效负载数据以及所涉及的IP地址作为参数，计算出来的一个16位数字。

这和TCP中校验和工作方式一样。（Jon Postel为TCP和UDP编写了第一个RFC，所以它们使用相同的算法并不奇怪。）在本周的项目中，将会详细介绍如何计算校验和，只需要将TCP包头替换成UDP包头即可。

不分片最大负载

虽然这在后面才会详细介绍，但是底层协议可以决定是否将UDP数据包分割位更小的数据包。甚至也许在互联网的某一部分，只能处理发送一定大小的数据包通过。我们把这种将UDP数据包分割为多个IP数据包的过程叫做“分片”。

一个线路中能通过的数据包的最大大小叫做MTU（最大传输单元），在因特网（IPv4）上能传输的最小可能的MTU大小是576字节，而最大的IP头大小是60字节，UDP头是8字节。所以能保证最大不会被分片的负载大小是576-60-8 = 508字节[如果数据是通过一个VPN发送的，能发送的最大负载大小要小于这个数，在这里，我们先不考虑这个情况。]。由于有时IP头的大小小于60字节，所以很多发送源会说发送上限是512字节。分片有什么坏处么？有些路由器会丢弃分片的UDP数据，所以保持发送的数据小于最小MTU常常是一个好主意。

用在何处?

如果UDP可能在任意时候丢失数据包，那么为什么我们会使用它？因为它对性能提升是显著的，所以这是一个权衡。

下面是一些UDP会被用到的场景:

1. 如果你对是否丢失数据包并不在意，比如，你正在传送的是声音，视频或者甚至是游戏帧信息，丢失一些数据包是不会影响整体的，最多只是流将暂时中断，然后在下一个数据包到达时继续。这是最常见的使用模式。多玩家高帧率游戏一般使用UDP来做逐帧更新，同时也在低带宽时使用TCP来发送玩家聊天信息或者用户装备变化。

2. 如果传输过程不能忍受数据包丢失，还是可以基于UDP来实现另一种协议，TFTP（简单文件传输协议）这样一种协议。该协议会在每个数据包中放置一个序列号，并且只会在对方回复一个TFTP ACK数据包之后再发送下一个数据包。由于需要等待到ACK才能发送下一个数据包，所以该协议速度并不快，不过这是一个非常容易实现的协议。这是一种比较少见到的协议，TFTP一般用在没有操作系统的无盘计算机上。他们在启动时通过网络传输操作系统，并且利用一个预置的网络协议栈来使其可以被实现。而实现Ethernet/IP/UDP协议栈要远比实现Ethernet/IP/TCP简单。

3. 如果，你希望在没有多个TCP连接的情况下对不同的数据“流”进行多路复用，这时就可以用标识符标记每个UDP数据包，以便它们在到达时都进入正确的位置。

4. UDP使得有机会进行提前处理，你甚至又可能在数据包3还未到达的情况下预先处理数据包4。

5. 其他合适的情况。

UDP （数据报）套接字

对于UDP套接字，它和TCP套接字有如下很多不同点:

- 由于没有“连接”，调用者将不需要调用`listen()`, `connect()`, `accept()`, `send()`，或者`recv()`。
- 调用 `sendto()`来发送UDP数据。
- 调用 `recvfrom()`来接收UDP数据。

服务端程序

通用的服务端程序时创建一个新的类型为`SOCK_DGRAM`的套接字，也就是数据报/UDP套接字。（在TCP中，使用的是默认的`SOCK_STREAM` 类型套接字。）接着服务器端调用 `bind()` 将其绑定到一个端口上，该端口是客户端将要发送数据包的地方。在此之后，服务器端可以循环接收数据或者发送数据。

当服务器端接收到数据时，`recvfrom()` 将返回发送数据的主机和端口。这可以用于回复，以将数据发送回发送方。下面是服务器端程序的例子:

```
# UDP Server

import sys
import socket

# Parse command line
try:
    port = int(sys.argv[1])
```

```
except:
    print("usage: udpserver.py port", file=sys.stderr)
    sys.exit(1)

# Make new UDP (datagram) socket
s = socket.socket(type=socket.SOCK_DGRAM)

# Bind to a port
s.bind(("", port))

# Loop receiving data
while True:
    # Get data
    data, sender = s.recvfrom(4096)
    print(f"Got data from {sender[0]}:{sender[1]}: \"{data.decode()}\"")

    # Send a reply back to the original sender
    s.sendto(f"Got your {len(data)} byte(s) of data!".encode(), sender)
```

客户端程序

客户端程序几乎和服务器端程序相同，唯一的不同是客户端不需要使用`bind()`来绑定到特定的端口，它会让操作系统在第一次调用`sendto()`自动选择要绑定的端口。请记住: UDP是不可靠的，所以很有可能发送的数据最终不会到达，如果没有到达对端，再次尝试即可。（不过，如果是在本机运行的话，几乎最终一定会达到。）

下面是可以和上面服务器程序通信的客户端程序:

```
# UDP Client

import socket
import sys

# Parse command line
try:
    server = sys.argv[1]
    port = int(sys.argv[2])
    message = sys.argv[3]
except:
    print("usage: udpclient.py server port message", file=sys.stderr)
    sys.exit(1)

# Make new UDP (datagram) socket
s = socket.socket(type=socket.SOCK_DGRAM)

# Send data to the server
print("Sending message...")
s.sendto(message.encode(), (server, port))

# Wait for a reply
data, sender = s.recvfrom(4096)
```

```
print(f"Got reply: \"{data.decode()}\"")

s.close()
```

思考题

- 除了保证数据能够传输到对端，还有什么TCP可以提供但是UDP做到的么?
- 为什么一般都推荐保持发送比较小的UDP数据包?
- 为什么UDP的包头小于TCP的包头?
- 函数`sendto()`需要指定目的地IP地址和端口，为什么TCP的`send()`函数不需要指定这些参数?
- 为什么UDP相对TCP来说没有那么可靠，人们仍然会使用它呢?

项目: 验证TCP数据包

在本项目中，你将会编写一些代码来检验TCP数据包，保证其在传输的过程中并没有损坏。

输入: 一对文件序列:

- 一个以点数模式保存了源和目的地IPv4地址的文件
- 一个同时含有TCP包头和负载的原始TCP数据包 你可以[fls[在此文件夹中下载到这些数据 | tcpcksum/tcp_data.zip]]。

输出:

- 对于每一个文件对，如果TCP校验和相同，那么就打印`PASS`，如若不然，打印 `FAIL`。

本项目中有很多知识点，所以强烈建议你自己动手编写程序和和进行测试。

在你开始你预想的方法之前，你应该100%理解此规范!在世界编码之前先弄清楚相关概念!

本项目最难的部分是理解其中的知识点！在你完全理解之前，你的代码将不会工作！

标准答案只有37行代码！（不包括空格和注释。）这不意味着这是一个需要被超越的数字，但这表明你需要付出多少努力来理解里面的知识点，而不是编写代码!

禁止使用的函数

在本项目中，你不能使用任何`socket`库中的函数。

如何编码

虽然你可以以任何你喜欢的方式来做这个，不过我推荐你按照如下的顺序来实现，每一步的细节将会在后面的章节里详细说明。

1. 读取`tcp_addrs_0.txt`文件。
2. 将里面的内容分为两个部分，源地址和目的地地址。
3. 编写一个将点-数格式的IP地址转换为字节字符串的函数。
4. 读取`tcp_data_0.dat` 文件。
5. 编写一个从`tcp_addrs_0.txt`所含的IP地址生成IP伪包头的字节序列和从`tcp_data_0.dat`生产TCP数据长度的函数。
6. 构建一个新版本的校验和为0的TCP数据。
7. 拼接伪包头和上面校验和为0的TCP数据。
8. 从`tcp_data_0.dat`提取原始的校验和。
9. 对比两个校验和的值，如果相同，说明程序正确。
10. 修改你的函数，使其能够处理所有10个文件。**头5个文件应当有可以匹配上的校验和，而后5个文件的校验和匹配不上！**也就是说，后五个文件模拟的是在传输中数据损坏。

校验和

TCP中校验和是一个16位的表示数据包所有字节“总和”的数值。（再补充一点，不是说它真的等于所有字节值的和！！！后面会进行详细介绍。）TCP包头本身包含了一个从发送端计算出的校验和。

接收端主机（也就是你现在假装的这一部分）独立从接收到的数据中计算校验和，并且保证它和发送端发送过来的校验和完全一致。如果是一致的，说明该数据包是好的，如若不然，那说明已经有错误了，要么是源IP地址值错了，要么是目的地IP地址值错了，抑或是TCP包头损坏了，或者是数据本身已经发生错误了，还有种情况，就是校验和自己已经损坏了。不管是哪种情况，如果校验和不一致，操作系统中的TCP软件都会请求一次重发。

在本项目中，你的工作是基于给出的数据计算校验和，以及保证它匹配（或者不匹配）输入数据中已经存在的校验和的值。

输入文件详情

请在下载 [fls[this ZIP|tcpcksum/tcp_data.zip]] 输入文件。

输入文件一共有10批，其中5批含有正确的校验和，第二批的五个是损坏的校验和。以防你没有理解到重点，前面这句话是你需要在该项目中100%取得的结果。这批文件名规则如下:

```
tcp_addrs_0.txt
tcp_addrs_0.dat

tcp addrs_1.txt
tcp addrs_1.dat
```

一直到序号9，每一对文件都是相互相关的。

`.txt` 文件

你可以使用编辑器看下 `tcp_addrs_n.txt` 文件，其中包含了一对随机的IP地址，如下所示:

```
192.0.2.207 192.0.2.244
```

这些表示该TCP数据包的 *源IP地址* 和 *目的地IP地址* 。为什么在计算TCP校验和的时候我们需要IP信息呢？继续往下看。

`.dat`文件

这是一个二进制文件，包含原始TCP头，紧随其后的是负载数据。如果你在编辑器里打开他，他看起来就像是一堆垃圾值，如果你的电脑上装有hexdump，你可以看到其中具体的字节信息，比如，下面是来自`hexdump`的输出:

```
hexdump -C tcp_data_0.dat
```

```
00000000  3f d7 c9 c5 ed d8 23 52  6a 15 32 96 50 d9 78 d8
|?.....#Rj.2.P.x.|
00000010  67 be ba aa 2a 63 25 2d  7c 4f 2a 39 52 69 4b 75  |g...*c%-
```

```
|O*9RiKu|
00000020  42 39 53                                          |B9S|
00000023
```

但在本项目中，文件中你真正需要关心的只有:

- 数据的长度（以字节表示）
- 存储在偏移量16-17的十六进制大端表示的校验和

在后面我们会继续详细介绍!

请注意：这些文件中都是一个“非正式正确”的TCP包头，虽然所有的字段都在，但是大部分数值（特别是那些标志以及可选字段）可能都是没有意义的。

到底如何计算TCP校验和?

具体过程并不是那么简单。

这里的TCP校验和主要用来检验这些事情:

- TCP包头
- TCP负载
- 源IP地址和目的地IP地址（防止在TCP流中包含有错误路由的信息）

最后一部分相当有意思了，因为IP地址信息完全不在TCP包头或者数据中，那么我们怎么将其囊括在校验和中呢?

TCP校验和是一个按照如下方法计算的两字节数字，对于一个特定的TCP包头数据，负载，以及源和目的地IP地址数据:

- 构建一个代表IP伪包头（后面会详细介绍）的字节序列
- 将当前TCP包头的校验和初始化为0
- 将IP伪数据头和TCP包头以及负载连结起来
- 基于上面的连接数据计算校验和

这就是计算TCP校验和的全过程，不过具体过程还有很多细节。

IP伪包头

因为我们想要确保数据中的IP地址是正确的，所以我们需要在校验和中包含IP报头。只不过我们没有包含真正的IP报头。我们只需要一个假的，它看起来是这样的(下图是直接从TCP RFC中窃取的):

```
+--------+--------+--------+--------+
|           Source Address          |
+--------+--------+--------+--------+
|         Destination Address       |
+--------+--------+--------+--------+
```

```
|  Zero  |  PTCL  |    TCP Length   |
+--------+--------+--------+--------+
```

别让上面的网格形式混淆了你：实际上的IP伪包头市一个字节的字符串，上面的表示形式只是为了更容易人类阅读。

上面图表中的每个`+`都代表一个字节分割符。因此，源地址是4字节。（嘿！因为IPv4地址长度本来就是4字节！），**你可以在`tcp_addrs_n.txt`文件中看到这个信息**。目的地地址也是4个字节， **你也可以在`tcp_addrs_n.txt`文件中看到这个信息**。

数据0是一个字节，对应的16进制字节形式值是`0x00`。PTCL是协议名，这一个字段永远都会被设置为值`0x06`。（IP有一些神奇的数字来表示它上面的更高级别的协议。TCP的编号是6。这就是它的由来。）

TCP长度就是数据总长度，以大端字节序表示的TCP数据包和数据总大小。**这其实就是你需要从文件`tcp_data_n.dat`读取数据的长度。**

因此，在计算TCP校验和之前，你必须首先创建一个IP伪包头!

伪包头实例

如果目的地地址IP是 `255.0.255.1`而目的地地址是`127.255.0.1`，相关的TCP数据长度是3490 (十六进制0x0da2)，对应的伪包头会如下面所示:

`Z` 是数字0的部分，而`P`是协议（永远是6）。看到字节是如何与输入对齐的了吗?（255是十六进制0xff, 127是十六进制0x7f, 等等。）

获取字节形式的IP地址

如果你已经注意到的话，文件`tcp_addrs_n.txt`中的IP地址是点数模式的，不是二进制模式的。**你将需要编写一个函数来将点数字符串转换为了4字节的序列。**

算法步骤:

- 将点数模式转换为一个四个整数的数组
- 调用`.to_bytes()`将上面的每个数字转换为字节
- 将上面的结果全部组装起来成为一个单独字节字符串

该函数会采用一个点-数形式的IPv4地址，并且返回一个四字节的字节数组作为结果。下面是一个示例测试：

- 输入: `"1.2.3.4"`
- 输出: `b'\x01\x02\x03\x04'`

然后，只需要对输入文件里面的每个IP地址都调用该函数，最后将他们拼接到伪包头后面。

获取TCP数据长度

这一步很简单: 只要你读取了`tcp_data_n.dat` 文件里面的任何一个，你就能得到数据的长度。

```
with open("tcp_data_0.dat", "rb") as fp:
    tcp_data = fp.read()
    tcp_length = len(tcp_data)  # <-- right here
```

请注意，这里在读取的二进制数据时候一定要使用`"rb"`模式！这就是这里`b`的涵义！如果你不按这个来做的话，可能会导致一切都不成功!

TCP 包头校验和

当计算校验和时，我们需要一个将校验和字段设置为0的TCP包头。同时还需要从我们收到的TCP报头中提取现有的校验和，以便我们可以将其与我们计算的校验和进行比较!这个图式很庞大，但好在我们实际上只关心其中的一部分。所以暂时可以跳过这一段，先查看下一段。

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

（再次说明，就像之前的IP伪包头一样，实际上的TCP包头是被编码过的数据流，而上面的网格模式只是便于人类可读。这个图表是一共是32个小格，数字都标示在上方。）

找一下上面的的TCP包头哪里含有`Checksum`？这是本项目中最关心的部分，它是一个位于TCP包头中，偏移量为16字节和17字节两字节的数字（大端序）。所以在`tcp_data_n.dat`文件中，该字段也是位于TCP包头偏移量16-17字节处（然后是TCP有效载荷）。

你需要该文件中的校验和数值，方法是首先使用切片获取这两个字节，然后使用`.from bytes()`将其转换为数字。这是你需要最终进行对比的原始校验和!

同时你也需要生成校验和置为0的一份TCP包头和数据，你可以使用如下的方法得到该值:

```
tcp_zero_cksum = tcp_data[:16] + b'\x00\x00' + tcp_data[18:]
```

看到这里我们如何构建一个新版本的TCP数据了吗?方法就是将现有校验和前后的所有数据切片，并在中间放置两个零字节。

计算校验和详细步骤

终于，我们要开始学习计算校验和的具体方法了，目前我们已经完成了如下步骤:

- 构建IP伪包头
- 从已有的TCP包头中提取校验和
- 构建一个校验和为0的TCP包头

下面我们将要学习详细的计算了，这是定义中描述的:

> 校验和字段是一个用反码相加所有包头中16位的数，然后再对结果取反的一个操纵。

这下好了，这个概念使我们理解更加混乱了，这到底是什么鬼东西?

反码是一种二进制表示正整数和负整数的方法，幸运的是，我们不需要知道关于它的具体细节。但是，有一件我们需要注意的是，所有我们讨论的都是“16位的字”。。。这又是什么意思呢?

这意味着我们不再把所有这些数据看作是一堆字节，而是把它看作是一个16位整体的值。所以，如果你看到如下这些字节：

```
01 02 03 04 05 06
```

我们需要把它看作3个16位的数字：

```
0102 0304 0506
```

接下来，我们要把他们以反码的加法加起来。嗯。。。但是如果这里有奇数个字节怎么办?

> 如果一个段包含奇数个需要被计算校验和的头部和八元组数据，校验后，最后一个八位元组在 右侧填充零，最后一个填充了正确零数量的八元组组成了一个可被计算校验和的16位字。

所以，我们需要看一下用来填充`tcp_length`字段的`tcp_data_0.dat`文件的长度，如果它是一个奇数，那么就在整个数据的后面加上零。

方便的是，我们已经有了一个可以使用的TCP数据副本:我们将校验和归零后生成的版本。因为我们无论如何都要迭代访问这个版本的数据，所以不妨将0字节附加到它后面:

```
if len(tcp_zero_cksum) % 2 == 1:
    tcp_zero_cksum += b'\x00'
```

而这可以保证它的长度是一个偶数。我们可以像下面这样提取所有的16位值。记住，要校验和的数据包括伪报头和TCP数据(校验和字段设置为零):

```
data = pseudoheader + tcp_zero_cksum

offset = 0   # byte offset into data
```

```
while offset < len(data):
    # Slice 2 bytes out and get their value:

    word = int.from_bytes(data[offset:offset + 2], "big")

    offset += 2   # Go to the next 2-byte value
```

太棒了！这个循环遍历整个数据块中的所有的字。但这能给我们带来什么呢?校验和是多少?让我们使用上面的循环框架并且把校验和的代码放在里面。

让我们再回到补码的部分，和一些16位相关的特性，在Python中，这些都有些棘手因为Python用的都是任意精度整数。但是，下面是我们想怎样来计算它，在下面的例子中，`tcp_data`是使用0来补齐成偶数长度来计算校验和的TCP数据。

```
# Pseudocode

function checksum(pseudo_header, tcp_data)
    data = pseudo_header + tcp_data

    total = 0

    for each 16-bit word of data:  # See code above

        total += word
        total = (total & 0xffff) + (total >> 16)  # carry around

    return (~total) & 0xffff  # one's complement
```

上面例子中“可以学到的小知识”就是反码的计算，其中到处存在的`&0xffff`是强制Python会给我们一个16位的整数。

还记得说明中是怎么说的么?

> 校验和字段是一个用反码相加所有包头中16位的数，然后再对结果取反的一个操纵。

上面的循环告诉了我们什么叫做“反码的加法”，结尾的`~total`给我们演示了是那么叫做对结果取反。

终极对比

现在，你已经计算了TCP数据的校验和并且同时也从TCP数据中提取出了已存在的校验和，下面可以进行对比了。如果他们是相等的，说明数据是一致而且是正确的，如果它们不想等，那么说明数据损坏了。在示例数据中，开始的五个文件是一致的，后面的五个是损坏的。

输出

你程序的输出应该展示出哪些TCP数据对比是通过的，哪些是失败的，也就是说，下面是应该是你的输出的样子。

```
PASS
PASS
PASS
PASS
PASS
FAIL
FAIL
FAIL
FAIL
FAIL
```

胜利

世上无坦途，夸赞一下得到正确结果的自己吧！

IP子网和子网掩码

[*本章中所有IP地址都会使用IPv4，而不是IPv6。它们当中的逻辑是基本一样的，只是使用IPv4会更加利于学习基本原理.*]

如果你需要复习位操作，请参考附录: 位操作。

在本章中将会介绍:

- 地址的表示方法
- 如何将点-数模式转换成一个数字
- 如何将一个数字转换为点-数模式
- 子网与进阶版主机知识
- 子网掩码进阶知识
- 从斜线表示法中寻找子网掩码
- 获取一个IP地址的子网，并且构建对应的子网掩码

地址的表示

回忆下前面说的，IPv4地址通常是以点-数表示法来表示的，举例来说`198.51.100.10`。这其中的每一个数字都是一个字节，其取值范围是十进制的0-255（也就是十六进制的00-FF，二进制的00000000-11111111）。所以，实际上，点-数表示法只是为了人可以更加方便的阅读。

> 高级技巧: 不同进制比如十六进制，二进制，和十进制只是数字的不同表示方法，这就像在各个 语言中，数字的表示都是不同的。
>
> 当使用一个变量表示数值是，最好是将其视为没有任何进制的纯数字。只有当你将其写下（无论是 代码或者打印出来）的时候，进制才会有意义。
>
> 比如，Python以十进制作为默认的方式打印所有的数，同时，它也提供了很多方法来将其输出转化 为其他进制的模式。

让我们来看看地址`198.51.100.10`的十六进制表示: `c6.33.64.0a`。

让我们把这些字节合并在一起组成一个单独十六进制的数:`c633640a`。继续将这个数`c633640a`转换为十进制，我们可以得到: `3325256714`。所以换句话来说，下面这些全部都表示一个相同的IP地址:

```
198.51.100.10
c6.33.64.0a
c633640a
3325256714
```

清楚了吗？但是为什么要做这样的转换呢？

这是因为我们将要对IP地址做一些数学运算，现在，我们 *可以* 对每个字节做一次运算并且也会得到正确的结果。但事实证明，如果我们把所有这些字节组合成一个数字，我们可以一次对所有字节进行计算，这变得更容易。下面会继续详细说明过程！

从点-数表示形式转换

本节的目标是将点-数形式的字符串`198.51.100.10`转换为对应的数值，比如十进制`3325256714`。我们可以依然使用前一节的字符串操作方法，不过这次让我们做的更加 *位运算* 感强一点。

首先，让我们从IP地址中提取出单独的数字（Python可以使用`.split()`函数来实现）。字符串:

```
"198.51.100.10"
```

可以转换为一串字符串数组:

```
["198", "51", "100", "10"]
```

现在，让我们将开始处理这里面的每个数字。（在Python中，可以在一个循环中使用 `int()`函数，或者`map()`方法，或者使用列表表达式。）

```
[198, 51, 100, 10]
```

下面，我将要用十六进制表示这些数字因为这样会使后面的步骤更加清晰，但是，清记住，这些只是被存储为一个数值，如果你不主动要求Python以十六进制打印它，你是看不到这个十六进制的。

```
[0xc6, 0x33, 0x64, 0x0a]
```

为了构建我们自己的数字，我们需要依赖一些位操作，位运算的OR操作和位运算的shift操作。为了方便示例，让我们将其中的数学过程硬编码:

```
(0xc6 << 24) | (0x33 << 16) | (0x64 << 8) | 0x0a
```

运行上述代码，Python会返回十进制数字`3325256714`，将其转化为十六进制，将会得到`0xc633640a`。

你可以用上面的方程将任意4字节的集合打包成为一个数字。

> 还有一个更加简洁的方法是使用循环依次处理每个字节。 试试看你能不能给自己增加点挑战！给你的助教私信你的答案，看看自己能不能完全正确！

转换成点-数形式

如果，你现在有是上一节的数值 `3325256714`，然后想把它转换为对应的IP地址，该怎么做呢？我们也可以使用一些移位的方法来完成这个转换！不过，我们需要做一些与操作并且利用掩码来得到每个需要的部分。

让我们先将其转换为十六进制，因为每一个字节都是2个十六进制数字，这使得我们能看的更清晰点:`0xc633640a`。下面，我么只要将这个数字分别向右移0个比特，8个比特，16个比特以及24个比特:

```
0xc633640a >> 24 == 0x000000c6
0xc633640a >> 16 == 0x0000c633
0xc633640a >> 8  == 0x00c63364
0xc633640a >> 0  == 0xc633640a
```

如果你仅看最右边的两个比特，你会看到它们就是原始数字中的字节值:

```
0xc633640a >> 24 == 0x000000 c6
0xc633640a >> 16 == 0x0000c6 33
0xc633640a >> 8  == 0x00c633 64
0xc633640a >> 0  == 0xc63364 0a
```

所以我们学到了一些东西，除了直接查看右移8位，例如，我们得到这个:

```
0xc633640a >> 8  == 0x00c63364
```

所以，我只需要最右边的字节 `0x64` ，而不是其中的`0xc633`部分。我怎么才能把这其中的高位变成0并且去掉，而只留下 `0x64`? 我们可以使用 *掩码与* 运算！位与运算符可以像筛选器一样工作，让数字的一部分通过，并将其他部分归零。如果我们用字节`0xff`对该数字进行位与运算，那么相对的所有8位都设置为`1`，而除这8位之外的前面的所有位都隐含值`0`。

```
  0x00c63364
& 0x000000ff
------------
  0x00000064
```

看！`0x64`就是我们想要从IP地址里得到的字节！可以仔细看下在掩码中有二进制`1`的地方(除了这里用十六进制表示)，它让值“显示通过”，而任何有零的地方都被清除了。按照这个方法，我们可以从中提取出我们需要的数值:

```
(0xc633640a >> 24) & 0xff == 0x000000c6 == 0xc6
(0xc633640a >> 16) & 0xff == 0x00000033 == 0x33
(0xc633640a >> 8)  & 0xff == 0x00000064 == 0x64
(0xc633640a >> 0)  & 0xff == 0x0000000a == 0x0a
```

这些就是IP地址各个部分的单个字节值。为了得到最终的点-数字符串，你可以使用Python中的字符串f转换法，或者调用`.join()`函数。

子网和进阶版主机知识

让我们先回忆一下，IP地址被分成两个部分:子网号和主机号。IP地址左边的一些位是网络号，右边剩下的位是主机号。让我们再来看一个例子，这个IP地址左边的24位(3字节)是子网号，右边的8位(1字节)是主机号。

```
 Subnet    | Host
           |
198.51.100.10
```

所以这代表着在子网`198.51.100.0`中的`10`号主机。（在这里，我们将主机位替换为`0`的地址作为子网编号。）但我刚才单方面地说过，该IP地址有24个网络位。所以这样表示不是很简洁，于是诞生了斜杠表示法。

```
198.51.100.0/24     前24位被用来标示子网
```

或者，你可以将该方法直接用于IP地址上:

```
198.51.100.10/24    在子网198.51.100.0中的10号主机
```

让我们用同样的方法来表示一个16位的子网:

```
10.121.2.17/16     在子网10.121.0.0中的编号为2.17主机
```

明白了么？ Get it?

在上面的例子中，我们使用的数字都是8的倍数，所以子网划分会在字节边界上对齐，但是这并不代表着不可以使用非8的整数倍的数字来表示子网范围:

```
10.121.2.68/28    在子网10.121.2.64中的4号主机
```

如果你没搞明白上面的4和64是怎么来的，你可以把上面的字节都用二进制写出来！

子网掩码进阶

什么是 *子网掩码*?这是一串由`0`和`1`组成的序列，前面的一串`1`表示IP地址的哪一部分是网络部分， 它后面跟着一串二进制的`0`，表示哪一部分开始是标示主机的部分。这个可以用来标示一个IP地址属于哪个子网，或者说IP地址中哪些部分表示主机编码。

让我们试着用二进制来演示一下，我们将使用如下的IP地址作为例子:

```
198.51.100.10/24    子网198.51.100.0中的10号主机
```

首先，让我们将它转换为二进制。（这里有一个提示，子网掩码是按位进行与运算的掩码!）

```
11000110.00110011.01100100.00001010   198.51.100.10
```

现在，基于示例IP地址，让我们写下24个连续的`1`（因为这里有一个表示子网范围的`/24`）后面跟着8个连续的`0`（因为IP地址一共只有32位）。

```
11111111.11111111.11111111.00000000   255.255.255.0  子网掩码!

11000110.00110011.01100100.00001010   198.51.100.10
```

这就是和`/24`子网相关联的子网掩码！`255.255.255.0`!

从斜杠表示法中求出子网掩码

如果我告诉你，现在有一个`/24`的子网，你怎样求出子网掩码是`255.255.255.0`？或者我告诉你`/28`的子网掩码是`255.255.255.240`，但是这是怎么得到的呢？

对于一个`/24`的子网，我们需要24个连续的`1`，后面跟着8个连续的`0`。 For a subnet `/24`, we need a run of 24 `1`s, followed by 8 `0`s.

附录: 位运算中介绍了好几种生产连续`1`的二进制数和移位的计算方法。一旦你有了那个大的二进制数，就需要使用我们在上面介绍的方法将它转换回点-数表示法。记住，子网可以使用任意位划分，`/17`也是一个合法的子网划分，它不需要必须是8的倍数！

在IP地址中提取子网

如果你有一个特定的以斜杠表示法表示的IP地址，比如：

```
198.51.100.10/24   子网198.51.100.0中的10号主机
```

你怎么才能从其中只得到子网(198.51.100.0)和只得到主机号呢？你可以使用按位与运算！

首先，我们计算出子网 `/24`的子网掩码是 `255.255.255.0`。接着，让我们将它们都翻译成二进制并且将它们相加:

```
  11111111.11111111.11111111.00000000   255.255.255.0  子网掩码
& 11000110.00110011.01100100.00001010   198.51.100.10  IP地址
--------------------------------------
  11000110.00110011.01100100.00000000   198.51.100.0   子网号!
```

我们也可以对这个整体进行一次性计算，而不是一次只计算一个字节……我们只需要将这些字节合成一个单一的值，就像我们在之前的章节中所做的那样:

```
  11111111111111111111111100000000   255.255.255.0  子网掩码
& 11000110001100110110010000001010   198.51.100.10  IP地址
---------------------------------------
  11000110001100110110010000000000   198.51.100.0   网络号
```

与操作可以对于这个整体一次性的进行计算得到结果！接着，我们可以像前面的章节一样，将其转换为点-数表示法。现在，如果你有IP地址和子网掩码，并且想从IP地址中获得主机位，而不是网络位。你知道怎么做吗?(提示:位运算中的非运算符!)

路由器无时无刻不在进行这种计算--它们提取到IP地址并且需要它所处的子网是否符合任何一条它能路由到的地址。所以它会用掩码提取出IP地址的子网号，并且将其和自己知道的所有能路由的规则对比，接着匹配最佳规则并且将其转发。

思考题

- 将IP地址`10.100.30.90`转换位32位（4字节）十六进制表示是怎样的？用十进制表示呢？用二进制表示呢？
- 32位数字`0xc0a88225`对应的点-数表示法的IP地址是什么？
- 十进制数字`180229186`对应的点-数表示法的IP地址是什么？
- 如果想从数字`0x12ff5678`提取第二个字节，需要的按位操作是什么？
- 子网掩码`255.255.0.0`的斜线表示法的结果是什么？
- `192.168.1.12/24`的子网掩码是什么？
- 将斜杠表示法的子网掩码转换为二进制值所需的位操作是什么？
- 给定一个IP地址值(一个32位数字)和一个子网掩码值(一个32位数字)，执行哪些位操作才能从IP地址中获得子网号？

192.168.1.0是网络号，但不是子网掩码。子网掩码是由斜杠符号得到的点和数字的值。在本例中，它是`/24`，转化为相应的子网掩码是`255.255.255.0`。

IP 路由

本章所有的内容都和如何在IP层路由有关，该任务由一种被称之为 *路由器* 的特殊计算机来完成，该设备知道怎么能正确的将流量转发到与其相连的各个线路。我们将看到，连接到互联网的每台计算机都可以被视作位一台路由器，但它们都依赖于其他路由器将数据包送到目的地。

我们将要讨论如下的两个内容:

- **路由协议簇**: 路由器是如何学习整个网络的“地图”的。
- **路由算法**: 路由器在知道了整个网络地图之后，如何正确的将数据包转换到正确的目的地。

让我们开始学习吧!

路由协议簇

路由协议的作用简单来说就是:为路由器提供足够的信息来做出如何路由的决定。换句话说就是，当路由器在一个接口上接收到数据包时，它是如何决定将其转发到哪个接口?它又是如何知道哪条路由通向数据包的最终目的地?

内部网关协议簇

当谈到在因特网上路由数据包时，我们需要在更高的维度上来观察这一过程。让我们把互联网看作是一堆松散连接的分块。声明一下，“分块”并不是一个官方的行术语。官方术语是自治系统(autonomous system, AS)，维基百科将其定义为:

> [...] 在一个或多个网络运营商的控制下，代表一个单一的管理实体或域的连接的互联网协议(IP) 路由前缀的集合，它向互联网提供一个通用的和明确定义的路由策略。

不过，现在我们就把它们简单的视为一个“分块”。

例如，OSU有一个簇状网络。它有自己的内部路由器和子网，这些路由器和子网“在互联网上”。一家[FAANG公司](https://en.wikipedia.org/wiki/Big Tech#FAANG)可能会有另一块自己的簇状网络。在这些分块的因特网集群中，*内部网关协议* 用于路由。这是一种路由算法，针对具有少量子网的较小网络进行了优化。

下面是一些内部网关协议的例子

缩写	名称	说明
OSPF	开放式最短路径优先	最常使用, IP层
IS-IS	中间系统路由交换协议	链路层
EIGRP	增强内部网关路由选择协议	思科拥有一些专利权
RIP	路由信息协议	旧协议, 很少被使用

当然，大家最终需要能够在这些互联网“分块”之间进行通信——毕竟整个互联网是连接的。如果你不能在俄勒冈州使用谷歌的服务器，那将是一件令人沮丧的事情。但很明显，谷歌的服务器没有俄勒冈州立大学的网络地图... ... 那么他们是怎么知道如何进行流量的路由呢?

外部网关协议簇

让每个路由器都有一个完整的互联网地图是不切实际的。对于上一节所假设的“分块”模式下获取全局地图是可行的，但是如果想获得全局的地图肯定是不现实的。所以，我们在这里需要处理的是如何在这些“分块”间通信，而 *外部网关协议* 就是用于在分块间进行通信。

还记得”分块“的官方术语么？自治系统，互联网上的每个自治系统都被分配了一个 *自治系统号(ASN)*，*边界网关协议(BGP)* 使用ASN号来帮助确定数据包的路由路线。

缩写	名称	说明
BGP	边界网关协议	广泛被使用
EGP	外部网关协议	已经被淘汰
Abbreviation	Name	Notes

BGP有两种模式: 内部BGP和外部BGP。内部模式像内部网关协议一样运作，而外部模式是作为外部网关协议来运行。这个[fl[视频|https://www.youtube.com/watch?v=A1KXPpqINZ4]]对此有完美的说明，我强烈建议你花两分钟时间看一下这个视频，这样可以融会贯通。

路由表

路由表是一种定义了如何转发数据包使得它们可以一路到达远方目的地的配置表。或者说，如果路由器位于目的地局域网中的话，路由表会指导路由器从物理层上（比如说，以太网）将流量从本机发出。上面介绍过的路由协议的一个目标就是帮助因特网上的路由器建立起它们的路由表。

所有连接到互联网的计算机——路由器和其他设备——都有路由表。即使在你的笔记本电脑上，操作系统也必须知道如何处理不同的目标地址。如果目的地是localhost怎么办?如果是同一子网上的另一台计算机又该怎么办?如果是非上面描述的两种情况又该怎么处理?

用一个典型的笔记本电脑作为例子，操作系统会将发送给localhost的流量保持在 *环回地址设备* 上并且保证一般情况下，这些流量不会被发送到外部的网络中。从程序员的视角来看起来仍然在做网路相关的操作，只不过操作系统知道将到`127.0.0.1`的流量都维持在内部进行路由。

如果你ping到和你在同一局域网的另一台电脑呢?在这种情况下，操作系统首先会检查自己的路由表，确定它在同一个局域网上，并通过以太网将其发送到目的地。

但是，如果你ping到与你完全不同的局域网中的另一台计算机呢?当这种情况发生时，操作系统不能只是发送一个以太网帧并让它到达目的地。因为目的地不在同一个以太网网络上!因此，你的计算机转而将数据包转发到其默认网关，可以看作是最后的路由器。如果在你的计算机中，没有寻找到目标子网的路由表条目，那么操作系统会将其发送到默认网关，后者将其转发到更大的因特网上。

让我用我自己的Linux电脑上的路由表做一个例子，它已经被分配了IP地址`192.168.1.230`。

	源地址	目标地址	网关	设备
1	`127.0.0.1`	`127.0.0.1`		`lo`
2	`127.0.0.1`	`127.0.0.0/8`		`lo`
3	`127.0.0.1`	`127.255.255.255`		`lo`

	源地址	目标地址	网关	设备
4	192.168.1.230	192.168.1.230		wlan0
5	192.168.1.230	192.168.1.0/24		wlan0
6	192.168.1.230	192.168.1.255		wlan0
7	192.168.1.230	default	192.168.1.1	wlan0

首先，让我们来看下从localhost到localhost(127.0.0.1)的路由。在这种情况下，操作系统查找匹配的路由，并在相应的接口上发送数据。此时，匹配到的 *环回地址* 接口是 (lo)，一个操作系统"伪造"的网络接口。（主要是为了性能的原因。）

但是，如果我们从127.0.0.1发送数据到子网127.0.0.0/8中的任何地址会怎样呢？这种情况下仍然还是使用lo接口。并且，如果我们要发送数据到127.255.255.255（位于子网 127.0.0.0/8上）这样的广播地址也是一样的过程。其他条目就更有趣了。请记住，你查看的路由表机器已经被分配了IP地址192.168.1.230。

所以当我们查看上面的第四条时，其实我们正在了解当我从我自己的机器上发送数据给自己的入口。这就像localhost，除了我故意使用连接到我的无线局域网设备的IP wlan0。操作系统将使用这个条目，但它足够聪明，不会麻烦地通过网络发送——毕竟，这已经就是目的地了。

接下来的一条，是标示从该机器上发送数据给任何位于192.168.1.0/24 子网中的主机的路由接口。打个比方来说，从我的机器192.168.1.230发送数据给，比如说 192.168.1.22，或者任意位于该子网主机。接下来的第六条标示我们有一条位于局域网中的广播地址的路由，它也会通过WiFi设备接口发送出去。

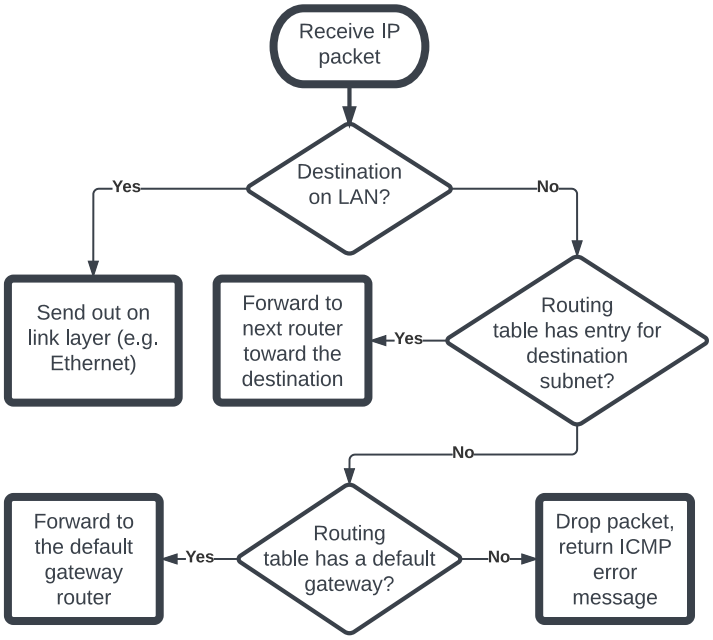
但是，如果上面所有的条目都匹配失败会怎样？比如如果我想从192.168.1.230发送到203.0.113.37？这个地址不在我路由表中目的地地址的IP或子网中。

这就是第七条的作用: *默认路由* 。这条是如果不知道如何路由数据，最终路由器发送数据包的地方。在你的家中，路由表中的这条记录一般是由你的网络提供商来填写。或者如果你愿意的话，你也可以自己购买并安装。

所以当从我家里的电脑ping example.com（当我写下这些文字的时候，是93.184.216.34）时，这些数据包就会从我的默认路由发出，因为其对应的子网并没有出现在我的路由表中。(它们也不会出现在我的默认网关的路由表中。所以,同样的道理，默认网关的路由器也会把它们从它的默认路由转发出去。)

路由算法

现在让我们假设路由协议已经完成了它们的工作，并且所有的路由器都拥有它们需要的信息，也就是知道如何一路将数据包转发到指定的目的地。数据包到达时，路由器会按照特定步骤来决定下一步做什么。（在接下来的讨论中，我们将忽略环回设备，并假设所有流量都在实际网络上。）



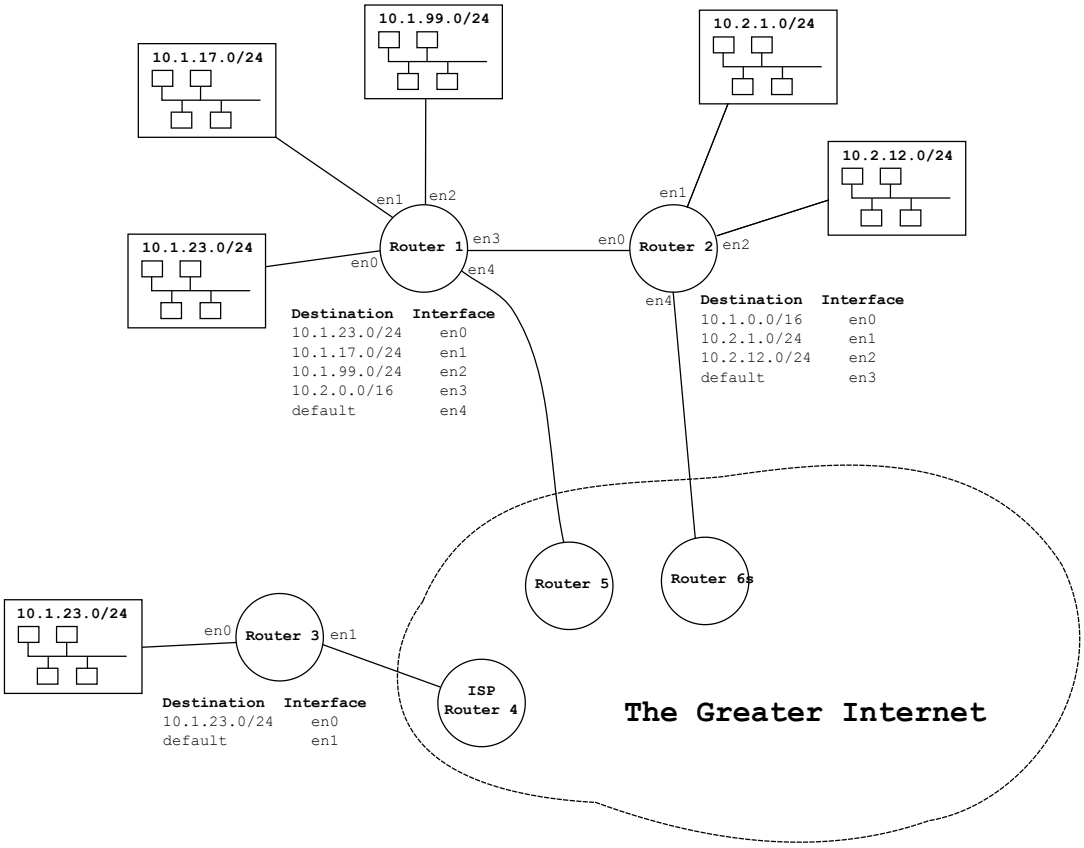
- 如果目的IP直接位于连接到路由器的本地网络中
  - 直接将数据包发送到该链接上（物理层，比如说，以太网）
- 否则，如果在路由表中拥有目的地IP网络的表项
  - 将其转发到通往目的地地址的下一个路由器上
- 否则，如果存在一个默认路由:
  - 将数据包发送给默认路由器
- 如果上述条件皆不满足:
  - 丢掉数据包
  - 发送一个ICMP错误消息“目的地不可达”给发送方。

如果有多条匹配的路由，则选择子网掩码最长的路由。如果掩码长度一样，则选择度量值最少的那个。如果仍然是平局，可以使用ECMP（等代价多路径）路由--将数据包沿两条路由一起发送。

路由实例

让我们来看看一个实际的例子 [flx[你可以下载此图表的PDF文件，这样可以获得更清晰的信息|ip-routing-demo.svg]]。在这张图表中，还有很多缺少的部分。但是值得关注的是，每个路由器接口都附带一个IP地址。

另外一个需要注意的点，比如说，1号路由直接链接到3个子网，并且每个都有对应的一个IP地址。



跟踪这些数据包的路由:

源	目的地
10.1.23.12	10.2.1.16
10.1.99.2	10.1.99.6
192.168.2.30	8.8.8.8
10.2.12.37	192.168.2.12
10.1.17.22	10.1.17.23
10.2.12.2	10.1.23.12

路由循环以及存活时间

根据上面的知识，显而易见，我们可以建立起一个循环，让数据包一直在这个循环中传播。为了解决这个问题，IP协议在它的头部有一个字段: *存活时间* (TTL)。 这是一个从255开始的一个字节表示的计数器，数据包每经历一个路由，其数字都会减一。

当这个计数器减到0的时候，这个数据包会被丢弃并且会给发送端返回一个ICMP"超时"错误消息。所以一个数据包最大只能经过255跳，哪怕在一个无休无止的循环中。

`traceroute`这个程序的工作原理是向TTL为1的目的地发送数据包，并查看谁发送回ICMP消息。然后再发送另一个TTL为2的报文，看看是谁响应了。它不断增加后续数据包的TTL，直到最终目的地响应。

广播地址

在IPv4中，有一种特殊的地址叫做 *广播地址*。这是一个将数据包发送给局域网中所有机器的地址。广播地址是一个在子网中将所有主机位都设置为`1`的地址。

举例来说:

子网	子网掩码	广播地址
`10.20.30.0/24`	`255.255.255.0`	`10.20.30.255`
`10.20.0.0/16`	`255.255.0.0`	`10.20.255.255`
`10.0.0.0/8`	`255.0.0.0`	`10.255.255.255`

在网络中，还有一个 *唯一* 的广播地址: `255.255.255.255`。它会将数据发送给局域网上的每一个人。这些广播包不会通过任何路由器，甚至`255.255.255.255`也没有。路由器不会把它们转发到任何地方。不然的话，整个网络世界将会变成一个非常嘈杂的地方。

该地址的一个重要作用是当你第一次打开你的笔记本电脑链接WiFi。笔记本电脑并不知道自己的IP，子网，默认网关，甚至不知道找谁能问到这些信息。所以它会用DHCP数据包广播一个数据包到`255.255.255.255`来询问这些信息。一个DHCP服务器会进行监听并且会回复相应的消息。

既然你们都在疑惑:IPv6没有广播地址;它被一种叫做IPv6多播的东西所消除。同样的道理，只是更加专注于一件事情。

思考题

- 内部网关协议和外部网关协议的不同点是什么?
- 一般来说，路由协议的目标是什么?
- 使用内部网关协议的地方的例子是什么?外部网关协议呢?
- 路由器用它的路由表来决定什么?
- 如果目的IP地址不在其本地子网之一上，IP路由器下一步会做什么?
- 一个程序为什么要向广播地址发送一些内容?

实战项目: 计算和发现子网

为了准备我们的后续项目，即寻找跨网络的路由，我们需要先做一些工作来弄清楚IP地址、子网掩码和子网是如何协同工作的。

在本项目中，我们将会对前面几章学习到的内容进行一些练习，我们将会：

- 编写函数将点-数字形式的IP地址转换为单个32位值，然后再转换回来。
- 编写一个函数，将斜杠形式表示的子网掩码转换为表示该掩码的单个32位值。
- 编写一个函数来查看两个IP地址是否在同一子网上。

限制

本项目中 **不能** 使用：

- 任何`socket`模块中的函数。
- 任何`struct`模块中的函数。
- 任何`netaddr`模块中的函数。
- 函数`.to_bytes()` 和 `.from_bytes()`。

目的是为了你自己动手写这些具体的位操作函数。

最终目标

[fls[请获取这个ZIP文件中的框架代码和其他相关文件|netfuncs/netfuncs.zip]]。你需要填写其中空缺的部分。

实现`netfuncs.py`中的下列函数:

- `ipv4_to_value(ipv4_addr)`
- `value_to_ipv4(addr)`
- `get_subnet_mask_value(slash)`
- `ips_same_subnet(ip1, ip2, slash)`
- `get_network(ip_value, netmask)`
- `find_router_for_ip(routers, ip)`

在文件中每个的文档字符串中都有函数作用的描述，这里要特别注意的是所示示例中的输入和输出 *类型*。请注意，这里面没有函数需要多于5到15行，如果你在实现的过程中，函数的大小大于这个值，大概率你没有走在正确的道路上。

测试

我鼓励你一次 *实现一个函数*，并在开始实现下一个函数之前，通过使用自己的示例数据来对其进行测试。你可以把将验证调用加到函数中，这样可以帮你验证你正在做的是正确的。还可以注释中的输入和输出作为测试条件。

在`netfuncs.py` 中有叫做`my_tests()`的函数，如果你取消它的注释，那么它将作为默认的主函数入口执行。如果你取消注释掉`my_tests()`，你可以按照下面的方法运行程序:

```
python netfuncs.py
```

并且查看从该函数的输出。请确保在你提交程序之前注释掉了`my_tests()`，如同下一节将要说明的一样。

运行程序

你将会像如下方式一样运行该程序:

```
python netfuncs.py example1.json
```

它从`example1.json`阅读JSON数据并且在从执行你的函数中的很多部分。输出包含在`example1_output.txt`文件中，如果所有事情都正确的话，那么输出会如下所示:

```
Routers:
     10.34.166.1: netmask 255.255.255.0: network 10.34.166.0
     10.34.194.1: netmask 255.255.255.0: network 10.34.194.0
     10.34.209.1: netmask 255.255.255.0: network 10.34.209.0
     10.34.250.1: netmask 255.255.255.0: network 10.34.250.0
      10.34.46.1: netmask 255.255.255.0: network 10.34.46.0
      10.34.52.1: netmask 255.255.255.0: network 10.34.52.0
      10.34.53.1: netmask 255.255.255.0: network 10.34.53.0
      10.34.79.1: netmask 255.255.255.0: network 10.34.79.0
      10.34.91.1: netmask 255.255.255.0: network 10.34.91.0
      10.34.98.1: netmask 255.255.255.0: network 10.34.98.0

IP Pairs:
   10.34.194.188    10.34.91.252: different subnets
   10.34.209.189    10.34.91.120: different subnets
   10.34.209.229    10.34.166.26: different subnets
   10.34.250.213    10.34.91.184: different subnets
   10.34.250.228    10.34.52.119: different subnets
   10.34.250.234     10.34.46.73: different subnets
     10.34.46.25   10.34.166.228: different subnets
    10.34.52.118     10.34.91.55: different subnets
    10.34.52.158     10.34.166.1: different subnets
    10.34.52.187    10.34.52.244: same subnet
     10.34.52.23    10.34.46.130: different subnets
     10.34.52.60    10.34.46.125: different subnets
    10.34.79.218     10.34.79.58: same subnet
     10.34.79.81    10.34.46.142: different subnets
     10.34.79.99    10.34.46.205: different subnets
    10.34.91.205    10.34.53.190: different subnets
     10.34.91.68    10.34.79.122: different subnets
     10.34.91.97    10.34.46.255: different subnets
    10.34.98.184     10.34.209.6: different subnets
     10.34.98.33   10.34.166.170: different subnets

Routers and corresponding IPs:
```

```
     10.34.166.1: ['10.34.166.1', '10.34.166.170', '10.34.166.228',
'10.34.166.26']
     10.34.194.1: ['10.34.194.188']
     10.34.209.1: ['10.34.209.189', '10.34.209.229', '10.34.209.6']
     10.34.250.1: ['10.34.250.213', '10.34.250.228', '10.34.250.234']
      10.34.46.1: ['10.34.46.125', '10.34.46.130', '10.34.46.142',
'10.34.46.205', '10.34.46.25', '10.34.46.255', '10.34.46.73']
      10.34.52.1: ['10.34.52.118', '10.34.52.119', '10.34.52.158',
'10.34.52.187', '10.34.52.23', '10.34.52.244', '10.34.52.60']
      10.34.53.1: ['10.34.53.190']
      10.34.79.1: ['10.34.79.122', '10.34.79.218', '10.34.79.58',
'10.34.79.81', '10.34.79.99']
      10.34.91.1: ['10.34.91.120', '10.34.91.184', '10.34.91.205',
'10.34.91.252', '10.34.91.55', '10.34.91.68', '10.34.91.97']
      10.34.98.1: ['10.34.98.184', '10.34.98.33']
```

如果你得到了不同的输出，请尝试查看代码，看看哪些函数使用了错误的输出。然后在`my_tests()`函数中对它们进行更详细的测试。

链接层和以太网

我们将深入到互联网的核心:链路层。

层级	职责	协议举例
应用层	结构化的应用层数据	HTTP, FTP, TFTP, Telnet, SSH, SMTP, POP, IMAP
传输层	数据完整性，数据分割和组装	TCP, UDP
网络层	路由	IP, IPv6, ICMP
链路层	物理设备，线路上的电信号	Ethernet, PPP, token ring

链路层是将网络的比特转换为电信号的一层。正如我们马上将要看到的，这是以太网存在的层级。

八位位组一览

之前我们一直使用“字节”表示8位二进制数字，现在是时候更加规范一些了。从历史上看，一个字节不一定是8位二进制。(众所周知，C编程语言并没有明确规定一个字节到底有多少位。)没有什么能阻止电脑设计者去创造东西。只不过巧合的是，基本上每台现代计算机都用8位来表示一个字节。

为了更精确，人们有时用 *八位位组* 这个词来表示8位二进制。它的表示严格的8位二进制。当一个人平时说出（或者写下）“字节”，他们大概率还是指的是8位二进制序列。当一个人说“八位位组”时，那么它表示的是严格的8位二进制，不含任何其他的意思。

帧与包

当我们讲到链路层时，我们需要习惯更多的术语。通过以太网发送的数据是一个数据包，但在数据包中是一个 *帧*。它就像一个包的更下一层概念。在会话中，人们交替使用以太网的“帧”和“包”。但是，正如我们后面将要看到的，在完整的ISO OSI分层网络模型中这俩存在着区别。在简化的因特网分层网络模型中，这个两个概念并没有进行区分，而这导致了术语混淆。

本章稍后将详细介绍这个迷人的细节。

MAC 地址

每个网络接口设备都有一个MAC(媒体访问控制)地址。这是LAN(局域网)上唯一的地址，用于发送和接收数据。

以太网MAC地址是由6个十六进制字节(12个十六进制数字)组成，中间用冒号(或连字符或句号)分隔。例如，这些都是你可能看到的MAC地址的表示方式:

```
ac:d1:b8:df:20:85
ac-d1-b8-df-20-85
acd1.b8df.2085
```

MAC地址在局域网内必须不能重复。这些数字是由制造商出厂时就分配好的，通常最终用户不会修改。(只有当你碰巧不走运，买了两个恰好分配了相同MAC地址的网卡时，你才会想要改变它们。)

以太网MAC地址的前三个字节称为 *OUI* (组织唯一标识符)，它被分配给各个制造商。这使得每个制造商有三个字节来唯一地表示他们制造的网卡。(这是16,777,216种可能的唯一组合。如果制造商用完了，他们总是可以获得另一个OUI——市面上也有1600万个OUI !)

> 有趣的互联网流言: 曾经有一家制造商生产一种叫做NE2000的仿冒网卡，该网卡本身已被称为 “便宜货”网卡。仿冒品制造商采取的捷径是在他们生产的每张卡上刻录相同的MAC地址。而当一 家公司购买了大量的电脑，却发现一次只有一台电脑可以工作时，事情就败露了。当然，在一个 家庭局域网中，一般只会有一张这样的网卡，这就不会造成问题--这就是这个制造商赚钱的漏洞。 雪上加霜的是，仿冒卡上的MAC地址是无法更改的。所以那家买了网卡的公司只能强制性的报废 这些网卡。
>
> 不过,可以留下一台。

我们达成了一致（通常情况下）

你将直接接触到的许多现代链路层协议都是基于这样的想法运行的，即它们都将都广播到共享媒介中，同时侦听对于它们的流量。这就像在一个人声鼎沸的房间里，有人喊你的名字——你得到了给你的数据，其他人都会忽略它。

在现实生活中的同样的方法也以太网等协议。当你在以太网网络上传输数据包时，同一以太网局域网上的其他人都可以看到该流量。除非该流量是专门发给他们的，不然他们的网卡会忽略不属于自己的流量。

> 注释：在这一段中有一些功能非常简单的设备。
>
> 一个是在现代的有线以太网中，一个被称为 *交换机* 设备，它通常会阻止防止数据包发送到 不应该被发送的节点。因此，网络并不像之前使用的拥挤的房间这种类比所暗示的那样嘈杂。在 交换机出现之前，我们使用的设备被称为 集线器, 它并不具有区分不同目的地的能力。它们 会将所有的以太网数据包广播到所有的地址中。

然而，并非每个链路层协议都以这种方式工作。但是所有链轮层协议的共同目标是在线路级别发送和接收数据。

多路访问方法

准备好听更多的背景故事了吗?如果同一个以太网上的每个人都在同一个房间里对其他人大喊大叫，那么在网线上是如何运作的呢?多个实体如何以不冲突的方式访问到共享的媒介?

> 当我们说“媒介”的时候，我们指的是网线(当你把电脑插入到网络中)，或者电波（当你使用WiFi时）。
> When we're talking "medium" here, we mean wires (if you've plugged your computer into the network) or radio (if you're on WiFi).

链路层协议用来允许多个实体访问共享媒介的方法称为 *多址访问方法* ，或 *通道访问方法* 。

而这有很多方法可以做到这一点,在同一个媒介上:

- 你可以在不同的频率上发送数据包。
- 你可以在不同的时间片上发送数据包，就像分时复用。
- 你可以使用扩频或跳频。
- 你可以把网络分成不同的“小组”。

- 你可以再加一条线，让流量同时在两个方向流动。

让我们再次以以太网作为一个例子，以太网使用的是面所提的类似“分时复用”模式。但如何去实现，保留了众多的选择。

有线以太网使用一种叫做CSMA/CD(带碰撞检测的载波感知多路访问)的东西。说出名词比理解起来简单多了。

该方法以以下的模式工作:

1. 以太网卡在房间里安静得待命——当没有其他网卡正在传输时。(这是CSMA/CD的“CSMA”部分。)
2. 以太网卡开始发送数据。
3. 在发送数据的同时，它也在监听网络。
4. 如果它收到了发出去的同一份，一切正常。 如果它没有收到它发送的相同的东西，这意味着另一个网络设备也在同一时间开始传输。这是一个碰撞检测，CSMA/CD的“CD”部分。
5. 为了解决这个问题，网卡会发送一个叫做“拥塞信号”的特殊信号来提醒其他网卡，网络上已经发生数据包碰撞，此时应该停止传输。然后网卡等待一个短而随机的时间，然后返回步骤1进行重试传输。

WiFi（无线）网络使用一些相似的技术，除了它叫CSMA/CA（具有碰撞避免的载波感知多址接入）。同样，说出名词比理解起来简单多了。

该方法以以下的模式工作:

1. 以太网卡在房间里安静得待命——当没有其他网卡正在传输时。(这是CSMA/CA的“CSMA”部分。)
2. 如果信道中开始有信号，那么网卡等待一个短而随机的时间，随后重新回到步骤1进行尝试。

这里省略了一些细节，但这些细节就是要点，在后面会详细说明。

以太网

还记得在分层网络模型中，每一层是如何将前一层的数据 *封装* 到自己的报头中的吗?

例如，HTTP数据(应用层)被封装在TCP(传输层)报头中。然后所有这些都被包裹在IP(网络层)报头中。接着**所有**这些又都被一起封装在以太网(链路层)帧中。并且，每一层的协议都拥有它自己的报头结构来完成自己这部分的工作。以太网这一层没有任何的区别，它也会封装它之上的层级的数据。

现在，我想对这些术语进行一些仔细阐述。传输的整个数据块是以太网 *数据包*。但在它里面是以太网 *帧*。正如我们稍后将看到的，这些对应于ISO OSI分层网络模型的两层(在因特网分层网络模型中已被浓缩为单个“链路层”)。

虽然我在这里写了数据包“内部”的帧，但请注意它们都是作为单个比特流传输的。

- **数据包:**
  - 7 个八元组: 前缀（十六进制 : `AA AA AA AA AA AA AA`）
  - 1 个八元组: 帧分割符开始（十六进制: `AB`）
  - **数据帧**
    - 6个八元组: 目的地MAC地址
    - 6个八元组: 源MAC地址

        - 4个八元组: 为区分虚拟局域网而设立的"Dot1q" 标签
        - 2个八元组: 负载长度/以太网类型
        - 46-1500个八元组: 负载
        - 4个八元组: CRC-32 校验和
    - 数据帧结束标示，载波信号丢失
    - 包间的间隙，对应于传输12八个元组所需的时间。

负载长度/以太网类型字段通常是被用来表示要发送的负载的真实长度，但是也可以填充一些其他的值来表示另一种有效载荷结构。可以传输的最大有效载荷是1500个八元组。这就是网络的最大传输单元MTU (Maximum Transmission Unit)。更大的数据必须被分割到这个大小。

> 以太网硬件可以使用这个1500的数字来区分到底这个报头字段是负载长度还是以太网类型。 如果数字小于等1500，那么它就是一个长度，不然，它就是一个以太网类型。 Ethernet hardware can use this 1500 number to differentiate the Payload Length/EtherType header field. If it's 1500 octets or fewer, it's a length. Otherwise it's an EtherType value.

在帧之后，有一个帧结束标记。这可以通过网线上载波信号的丢失来表示，或者通过某些版本的以太网中的某些显式传输标示来表示。最后，以太网数据包之间有一个时间间隔，对应于传输12八元组所需的时间。

因特网的两层?

如果你还记得的话，我们的简化层模型实际上是完整的ISO OSI 7层模型的压缩版本:

ISO OSI 层级	职责	示例协议
应用层	结构化的应用化数据	HTTP, FTP, TFTP, Telnet, SMTP, POP, IMAP
表示层	编码传输，加密，压缩	MIME, SSL/TLS, XDR
会话层	挂起，终止，重启两台计算机之前的会话	Sockets, TCP
传输层	数据完整性，数据分割和组装	TCP, UDP
网络层	路由	IP IPv6, ICMP
数据链路层	将数据装入帧中	Ethernet, PPP, SLIP
物理层	物理器件，线路上的信号	Ethernet physical layer, DSL, ISDN

当我们谈及因特网模型的"链路层"时，其实是这里的"数据链路层" *和* "物理层"的结合。因特网的数据链路层就是传输 *数据帧* 的层级， 它是（上面提到的）物理层中整个 *数据包* 的一个子集。

思考这个问题的另一种方式是，数据链路层主要需要解决处理逻辑实体，比如谁拥有什么MAC地址，有效负载校验和是什么。而物理层的全部任务就是弄清楚要发送的信号模式所对应的数据包和帧的开始和结束，何时降低载波信号，以及传输之间的延迟时间。

思考题

- 你电脑的MAC地址是什么？可以尝试在互联网上搜索下如何将它找出来。
- 以太网中 *帧* 与 *包* 的区别是什么?它们在ISO OSI网络栈中的什么位置?
- 字节和八元组之间的区别是什么?

- CSMA/CD和CSMA/CA之间的主要区别是什么？

ARP: 地址解析协议

作为一台网络中联网的计算机，我们有一个实际的问题。我们想通过局域网将数据从一台计算机发送到另外一个相同的子网。

这是我们需要按顺序构造因特网数据帧的过程:

- 我们将要发送的数据和它的长度
- 我们的源MAC地址
- 对方目的地的MAC地址

而我们目前知道的却只有:

- 我们将要发送的数据和它的长度
- 我们的源MAC地址
- 我们的源IP地址
- 对方的IP地址

这里我们缺少了什么？即使我们知道了对方电脑的IP地址， *我们也不知道其MAC地址*。 那么我们怎么样才能在没有MAC地址的情况下建立一个以太网帧？这肯定是不可以的，我们需要用某种方式拿到它。

> 再一次，在本节中，我们在谈论在局域网中发送数据，也就是本地以太网。而不是通过IP链接的 因特网。最重要的是，局域网中的两台电脑是通过相同的物理层网络链接的。 Again, for this section we're talking about sending on the LAN, the local Ethernet. Not over the Internet with IP. This is between two computers on the same Physical Layer network.

本节的所有内容都是关于ARP,地址解析协议。这是个如何一台计算怎么将其他计算机的IP地址映射到它的MAC地址的方法。

以太网广播帧

在开始之前，我们需要一些背景知识。回想一下前面介绍过的，网络硬件监听专门针对它的以太网帧，这样其他MAC地址目的的以太网帧将被忽略。

> 旁注: 这些非本人MAC地址的将被忽略，除非将网卡被设置为*混杂模式*, 该模式会接收局域网中 **所有** 流量并且将其转发给操作系统。

但是这也有一种例外: *广播数据帧*。这是一个目标MAC地址为`ff:ff:ff:ff:ff:ff`的帧。局域网中的所有设备都将接收到该帧。

我们将在ARP中使用它。

ARP--地址解析协议

在最开始的情况下，我们有目的地的IP地址，但没有它的MAC地址。而我们需要它的MAC地址。所以，ARP协议会按以下的步骤，让我们获得对方的MAC地址：

1. 源地址计算机会向网络中广播一个特殊的含有目的地IP地址的以太网帧，被称为 *ARP 请求*。（还记得前面一节中的以太网类型字段么？ARP数据包使用0x0806来区分普通以太网数据包和ARP请求包。）

2. 所有在局域网中的计算机都会收到这个ARP请求并且对其进行校验。不过，只有拥有目的地IP地址的计算才会继续处理该ARP请求，其他计算机将会丢弃该数据包。

3. 含有指定IP地址的目的地计算机会构建一个 *ARP应答*。这个数据帧中含有目的地计算机的MAC地址。

4. 目的地计算机地址会将ARP应答发回源计算机地址。

5. 源计算机地址接收到ARP应答后，从该包中就能知道目的地电脑的MAC地址了。

一切要素都齐了!既然我们知道了MAC地址，我们就可以不受约束地发送了。

ARP 缓存

每次都通过发送一些东西向一个计算机询问其MAC地址会使得流程十分繁杂，所有我们将结果 *缓存* 一会儿。然后，当我们想要发送到局域网上的特定IP时，我们可以先查看 *ARP缓存* 并查看IP/以太网对是否已经存在。如果存在，就不需要发送ARP请求——我们可以立即传输数据。

整个缓存会在一段时间之后失效并且会被清除，在规定上，没有一个标准的超时时间，不过我所了解的，从60秒到4个小时的超时时间都有。

如果给定IP地址的MAC地址发生变化，则缓存条目可能会失效。发生这种情况的最常见的情景就是，如果有人关闭了他们的笔记本电脑并离开了网络(带走了他们的MAC地址)，然后另一个拥有不同笔记本电脑(和不同MAC地址)的人出现并分配了相同的IP地址。如果发生这种情况，具有陈旧条目的发送端计算机就会把该IP的帧发送到错误的(旧的)MAC地址。

ARP 结构

ARP数据属于以太网帧的负载的一部分，它具有固定的长度。像前面提到的那样，它将会设置以太网类型/数据包字段为0x0806。在下面的有效载荷结构中，当说“硬件”时，它指的是链路层(例如本例中的以太网)，当说“协议”时，它指的是网络层(例如本例中的IP协议)。它在字段中填入这些对应的名称，因为没有硬性的要求ARP一定要使用以太网或IP——它也可以与其他协议一起工作。

负载字段，是一个具有固定长度的28位八元组,其中具体包括:

- 2位八元组: 硬件类型(以太网是`0x0001`)
- 2位八元组: 协议类型（IPv4是`0x8000`）
- 1位八元组: 以八元组表示的硬件地址长度（以太网是`0x06`）
- 1位八元组: 以八元组表示的协议地址长度（IPv4是`0x04`）
- 2位八元组: 操作类型(`0x01`是请求, `0x02`代表回复)
- 6位八元组: 发送端硬件地址（发送端MAC地址）
- 4位八元组: 发送端协议地址（发送端IP地址）
- 6位八元组: 接收端硬件地址（接收端MAC地址）
- 4位八元组: 接收端协议地址（接收端IP地址）

ARP 请求/回复

这里有点令人困惑的是，因为“发送方”字段总是传输数据的计算机设置的，而不是从谁是请求者的角度去设置的。所以我们将会声明1号计算是ARP的请求方，而2号计算机将会对其进行应答。

如果1号计算机发送一个ARP请求（”如果这是的你IP，那么你的MAC地址是什么?"）,那么它设置以下字段(除了上面提到的模版中的其他ARP请求字段之外):

- **发送端硬件地址**: 1号计算机的MAC地址
- **发送端的协议地址**: 1号计算机的IP地址
- **目的端的硬件地址**: 未使用
- **目的端的协议地址**: 1号计算机的所要询问的目的地IP地址

在2号计算机做出的ARP应答中("我拥有该IP地址，这是我的MAC地址"),它将会设置一下的字段:

- **发送端硬件地址**: 2号计算机的MAC地址
- **发送端的协议地址**: 2号计算机的IP地址
- **目的端的硬件地址**: 1号计算机的MAC地址
- **目的端的协议地址**: 1号计算机的IP地址

当1号计算机接收到声称是它要寻找的计算机的ARP回复时，它将会查看发送者字段并且从中提取出对应的MAC地址和IP地址。在这之后，1号计算机就可以通过以太网向2号计算机发送数据了，因为对于它，2号计算机现在是一个已知的MAC地址了。这就是如何在通过一个特定IP地址来查询到对应MAC地址的方法。

ARP的其他功能

ARP 公告

刚刚连上网的计算机主动宣布其ARP信息并不罕见，因为这可以让其他人有机会将数据添加到他们的缓存中，并且覆盖这些缓存中可能过时的数据以进行更新。

ARP 探测

一台计算机可以发出一个特殊构造的ARP请求，这个请求基本上是问:“还有其他人在使用这个IP地址吗?”通常发送端使用自己的IP地址进行请求，如果它得到响应，那么它就知道网络中有IP冲突。

IPv6 和 ARP

IPv6 拥有它自己的ARP版本，叫做NDP (邻居发现协议)。眼尖的人可能已经注意到，ARP只支持最多4字节的协议地址(比如说，IP地址)，而IPv6地址是16字节。

NDP解决了这个问题，并定义了许多可以用来代替ARP的协议ICMPv6(因特网消息控制协议(IPv6))消息。

思考题

- 请描述下ARP协议是为了解决什么问题。
- 为什么在ARP缓存中的条目需要有超时时间?
- 为什么IPv6不能使用ARP?

项目6: 用Dijkstra算法进行路由

通过内部网关协议，网络中的路由器可以彼此共享信息，这使得每个路由器可以了解它们自己所在网络。这样，每个路由器都可以在给定IP地址的情况下做出自主的路由决定。无论它要求去哪里，路由器总能把数据包转发到正确的方向。像开放最短路径优先(OSPF)这样的内部网关协议的实现使用Dijkstra算法沿着加权图寻找最短路径。

在这个项目中，我们将会模拟这种路由过程。我们将实现Dijkstra算法，打印出从一个IP到另一个IP的最短路径，显示中间所有路由器的IP。我们不会使用真正现实中的网络。相反，你的程序将读入一个包含网络描述的JSON文件，然后从中计算路由。

图知识复习

图由 *顶点* 和 *边* 组成。有时顶点被写作为“vertexes”或“verts”或“nodes（节点）”。边是从一个节点到另一个节点的连接。图上的任何节点都可以连接到任意数量的其他节点，这个数字甚至可以是零。一个节点可以连接到每一个其他节点。它甚至可以与自己连接。一个边还可以含有 *权重*， 通常这是指当你遍历图中的一条路径时所需要的代价。

例如，想象一张显示城市和城市间高速公路的地图。其中每条公路都标有它的长度。在这个例子中，城市是顶点，高速公路是边，高速公路的长度就是边的权重。

当你在这个图上寻找路线时，目的都是能够花最小的代价到达目的地。在我们的地图上，我们的目标是选择一条距离最短的路，从我们的起始城市出发，穿过中间的城市，到我们的目的地城市。

Dijkstra算法简介

Edsger Dijkstra (*DIKE-struh*)是一位著名的计算机科学家，他提出了很多东西，但其中有一个非常有影响力，以至于人们用他的名字来命名:Dijkstra的算法。

> 高级技巧: 记住如何拼写"Dijkstra"的小技巧就是记住这其中"ijk"是按字母顺序出现的。

如果希望在未加权的图中找到节点之间的最短路径，只需执行宽度优先遍历，直到找到所要查找的目的地。这里的距离我们只能用“跳数”来衡量。但是如果你给节点之间的边添加权重，BFT就不能帮助你解决这个问题。也许有些路径是非常理想的，而另一些则是非常不理想的。

而Dijkstra的算法可以完成这个目标，这是一个变化的BFT。算法的要点是:从起点向外探索，只追求到目前为止总长度最短的路径。而每条路径的总权重是其所有边的权重之和。

在一个良好联通的图中，从起点到终点会有很多可能的路径。但是由于我们目前只需要找到最短路径，如果我们已经发现一条比100万英里短的路径，我们永远不可能再去找到一条让我们偏离100万英里的路径了。

Dijkstra算法实现

Dijkstra算法基于图构建了一个树形结构，当你开始寻找从任何节点回到起点的最短路径时，该节点将其路径中的先前节点记录为其 *父节点*。如果稍后找到另一个更短的路径，则将父节点更换为新的更短路径的节点。[flw[Wikipedia 上有一个清晰图表来演示整个过程|Dijkstra's_算法]].

那么，到底该算法具体是如何工作的呢？Dijkstra算法本身只构建了表示回到起点的最短路径的树结构，稍后可以通过最短路径树找到一条最终的最短路径。

- 从一个起点开始计算图上所有最短路径的Dijkstra算法具体方法如下:
    - 初始化:
        - 创建一个空的叫`to visit` 集合，这是我们需要访问的所有节点的全集。
        - 创建一个叫`distance`字典。对于任何给定的节点(作为键)，它将保存从该节点到起始节点的距离。
        - 创建一个叫`parent` 字典。对于任何给定节点(作为键)，它保存(沿着最短路径)指向起始节点的节点的键集合。
        - 对于每一个节点:
            - 将它的`parent`值设置为 `None`。
            - 将它的`distance`值设置为无穷。（Python有一个默认表示无穷的值`math.inf`，不过你也可以使用一个超级大的数，比如，四十亿。）
            - 将节点加入`to_visit`集合。
        - 将开始节点的distance值设置为`0`。
    - 运行：
        - 当 `to_visit` 不为空时:
            - 找到`to_visit`里`distance`最小的节点，将其称为“当前节点”。
            - 从`to_visit`集合中移除当前节点。
            - 对于每个仍然还在`to_visit`集合中当前节点的邻居:
                - 计算每个从开始节点到这些邻居节点到距离，这个距离等于到当前节点的距离加上当前节点到邻居节点的边权重。
                - 如果当前计算出来的新距离比邻居节点当前存储的`distance`值更小时：
                - 将邻居节点的`distance`值设置为当前计算的新距离。
                - 当邻居节点的`parent`值设置为当前节点。
                - [这个过程被称之为“松弛”。节点距离从无穷大开始，然后“松弛”到它们的最短距离。]

Wikipedia提供了上述过程的伪代码，如果你觉得伪代码对于你更容易理解的话,你可以阅读下面的伪代码:

```
function Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
```

```

    return dist[], prev[]
```

在这个阶段，我们已经构造了由所有`parent`指针组成的树。为了找到从一个点回到起点(树的根)的最短路径，你只需要沿着那个点回到树的`parent`节点的路线。

- 获取从起始点到目的最短路径的方法:
  - 将当前节点设置为**目标**节点。
  - 将`path`设置为一个空数组。
  - 当前节点不是起始节点时:
    - 将当前节点加入`path`。
    - 当前节点等于当前节点的`parent`节点
  - 将起始节点加入路线

当然，这将以相反的顺序构建路径。但是这是必要的过程，因为这样父指针都指向作为根节点的起始节点。所以要么只要在最后将其反转，要么将目的地节点作为起始节点运行Dijkstra算法。

获得最小距离

算法的一部分是找到仍然在待访问集合中距离值最小的节点。在本项目中，你可以使用一个复杂度为`O(n)`的线形搜索来找到当前节点的最短路径值。但是，在实际例子中，如果使用线形搜索，复杂度将会太高--那将是和节点个数相关的`O(n²)`复杂度。所以在实现时会使用小顶堆 这样的数据结构，它最快将会提供接近`O(log n)`复杂度，这使得我们在处理众多节点时，也拥有`O(log n)`复杂度。

如果你想挑战自己一下，可以尝试使用小顶堆。

我们的这个项目中都有什么?

[*本项目中的所有IP地址都是IPv4地址.*]

[fls[在这里下载本项目框架代码的压缩包|dijkstra/dijkstra.zip]].

这里面又很多通用的内容，那么，其中有哪些适合本项目相关的呢?

函数，输入和输出

你需要实现这个函数:

```
def dijkstras_shortest_path(routers, src_ip, dest_ip):
```

该函数的输入是:

- `routers`: 一个代表图的字典
- `src_ip`: 一个点-数字符串，代表源IP地址
- `dest_ip`: 一个点-数字符串，代表目的地IP地址

该函数的输出是:

- 一个字符串数组，其中是所有路由途中的路由器IP地址

- **注意: 如果源IP地址和目的地IP地址位于相同的子网中，返回一个空的数组。** 在这种情况下，将会产生路由路线。

函数的运行代码已经包含在上面的框架代码中，它将输出到控制台，如下所示，显示源、目标和中间的所有路由器:

```
10.34.46.25 -> 10.34.166.228    ['10.34.46.1', '10.34.98.1',
'10.34.166.1']
```

图的表示方式

`routers`变量中保存的图字典看起来如下面片段所示:

```
{
    "10.34.98.1": {
        "connections": {
            "10.34.166.1": {
                "netmask": "/24",
                "interface": "en0",
                "ad": 70
            },
            "10.34.194.1": {
                "netmask": "/24",
                "interface": "en1",
                "ad": 93
            },
            "10.34.46.1": {
                "netmask": "/24",
                "interface": "en2",
                "ad": 64
            }
        },
        "netmask": "/24",
        "if_count": 3,
        "if_prefix": "en"
    },

    # ... etc. ...
```

最外层的键值（比如说`"10.34.98.1"`）是路由器的IP，他们是图的顶点。对于每一个顶点，都有一个`"connections"`的列表，也就是这其中图的边。

在每个链接中，还拥有一个字段`"ad"`，表示边的权重。

> "AD" 是 *管理距离* 的缩写，它表示手动或自动(或两者混合)设置的权重，用于定义特定路由的成本。默认值是110，越高的数值表示成本越高。
>
> 这个度量包含关于路由的许多想法，包括它提供了多少带宽、它有多拥塞、管理员希望使用(或不使用)多少，等等。

`"netmask"`字段中填写的是路由器IP的子网掩码，并且该字段也存在于所有链接的路由上。`"interface"`字段表示路由器上的哪个网络设备连接到哪个相邻的路由器，但在本项目中没有使用。

`"if_count"`和`"if_prefix"`也并没有在这个项目中被使用。

输入文件和示例输出

示例代码框架包括一个实例输入文件 (`example1.json`)和一个相当于这文件期待的输出 (`example1_output.json`)。

提示

- 本项目高度依赖于你在上一个项目中编写的网络功能!
- 在制定计划之前先完全理解这个项目的描述!
- 在编写任何代码之前，尽可能多地想出一个计划!

项目： 用WireShark查看ARP数据包

[fl[WireShark|https://www.wireshark.org/]]是一款常用的查看实时网络流量的工具，我们将利用它来看看是否可以捕获一些ARP请求和应答。

Wireshark是一个超级棒的的 *嗅探* 网络数据包的工具，它为用户提供了一种跟踪数据包在局域网中传输的方法。我们将在WireShark中设置一个过滤器，以便我们可以只查看来自特定机器的ARP数据包，这样大大减少了大海捞针的成本。

需要创建什么

一个含有以下4样东西的文档:

1. 你当前连接的MAC地址。
2. 你当前连接的IP地址。
3. 一个用户可读的WireShark ARP请求数据包。
4. 一个用户可读的WireShark ARP请求回复包。

下面是具体细节！

详细步骤

下面是我们将要做的具体事宜:

1. **查找你的以太网(MAC)地址** 你的计算机可能有很多以太网接口（比如说，一个是WiFi的接口，而另一个是有线接口--电脑机箱旁边的以太网插孔）

   因为现时大概率你正在使用你的无线，所以我们先来查找无线连接接口的MAC地址。（你可能需要上网搜索下怎么样完成这个目标。）

   对于这一步和接下来的步骤二，在Unix类的系统中这些信息都可以用如下命令获得:

   ```
   ifconfig
   ```

   如果你使用的是Windows系统，你可以使用如下命令获得:

   ```
   ipconfig
   ```

2. **查找你的IP地址**

   再次，我们想要知道你当前活动链接的接口的IP地址，大概率也是你的WiFi设备。

3. **打开WireShark**

在初始启动时，设置WireShark来查看你的活跃的以太网设备。在Linux上，这可能是 `wlan0`。在Mac上，它可能是`en0`。在Windows上，可能只是`Wi-Fi`。

4. **查询你子网中更多的IP地址** 对于本节来说，远端IP上是否有一个真的计算机并不重要，但如果存在对应IP地址的计算机那更好。通过观察一段时间的Wireshark日志，可以查看到你的局域网中还有哪些正在活跃的IP地址。

   > 你的IP地址与上子网掩码就是你的子网号，你可以尝试着将主机号部分替换成各种数字。 还可以使用你的默认网关（去网络上搜索下怎么查看默认网关地址）作为你的测试对象。

   在命令行中，`ping`你局域网上的另一个IP地址:

   ```
   ping [IP address]
   ```

   (可以按下 `CONTROL-C` 来结束你的ping。)

   在第一个ping中，你有没有看到经过Wireshark的任何ARP数据包？如果没有，请尝试子网中的其他IP地址，就像上面提及的那样。

   不管你发送了多少ping，你应该只看到一个ARP回复。（如果没有任何应答，你将看到每个ping都有一个请求!）这是因为当第一次获得应答之后，你的电脑会缓存ARP应答，而后面再也不需要发送类似请求了。

   在一到五分钟之后，你的电脑中的ARP缓存项将会过期，这时如果你继续ping那个IP，你将会看到另外一个ARP交换数据包。

5. **记录下请求和回复** 在该时间轴中，ARP请求过程将看起来像以下的片段(显然真实情况具有不同的IP地址):

   ```
   ARP 60 Who has 192.168.1.230? Tell 192.168.1.1
   ```

   如果一切顺利，你将会看到一个类似下面的回复:

   ```
   ARP 42 192.168.1.230 is at ac:d1:b8:df:20:85
   ```

   [如果你没有看到任何内容，试着修改你的过滤器，只输入“arp”。观察一会儿，看看你是否看到请求/回复数据包。]

   单击请求并查看左下方面板中的详细信息。展开“Address Resolution Protocol (request)”面板。随意选择一行，并且右击，选择"Copy->All Visible Items"。下面是示例请求（由于一行长度有限，进行了截断）：

   ```
   Frame 221567: 42 bytes on wire (336 bits), 42 bytes captured  [...]
   Ethernet II, Src: HonHaiPr_df:20:85 (ac:d1:b8:df:20:85), Dst: [...]
   Address Resolution Protocol (request)
   ```

```
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: HonHaiPr_df:20:85 (ac:d1:b8:df:20:85)
    Sender IP address: 192.168.1.230
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.1.148
```

点击时间线中的回复数据包，使用上面一样的方法复制回复信息。下面是回复消息的示例（由于一行长度有限，进行了截断）：

```
Frame 221572: 42 bytes on wire (336 bits), 42 bytes captured  [...]
Ethernet II, Src: Apple_63:3c:ef (8c:85:90:63:3c:ef), Dst:    [...]
Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: Apple_63:3c:ef (8c:85:90:63:3c:ef)
    Sender IP address: 192.168.1.148
    Target MAC address: HonHaiPr_df:20:85 (ac:d1:b8:df:20:85)
    Target IP address: 192.168.1.230
```

网络硬件

在这一节，我们将了解一些不同的网络硬件模块。他们中大多数已经安装与你家和你的电脑上了。

术语和模块

- **网络拓扑**: 描述一个网络的结构，包括设备和节点之间是怎么连接的，数据怎么从一部分网络流向另一部分。
- **Mbs**: 每秒多少M比特 (区别于“MBs”，每秒多少M字节)。
- **Gbs**: 每秒多少G比特，1Gbs = 1000 Mbs （区别于“GBs”，每秒多少G字节）。
- **带宽**: 以每秒多少比特来作为单位，某种类型的电缆在一定时间内可以传输多少数据，比如说，500 Mbs。
- **ISP**: 因特网服务提供商，你付钱，从而给你提供因特网连接的公司。（或者说是提供该服务的公司，即使你没有给他付钱！）
- **双绞线**: 通常人们认为的“以太网线缆”，他是一条两端都有插口的线缆。在其内部，成对的电线绞在一起以减少干扰。这些电缆规定了最大“运行”长度，这决定了在遇到瓶颈之前你可以将电缆串到多远的地方，通常是50-100米，具体取决于电缆和传输的速度。
    - **10baseT**: 10 Mbs 因特网双绞线
    - **100baseTX**: 100 Mbs 因特网双绞线，用于 *快速以太网*。
    - **1000baseT**: 1 Gbs 因特网双绞线，用于 *千兆以太网*。
    - **10GbaseT**: 10 Gbs 因特网双绞线，用于 *万兆以太网*。
    - **5类线**: 通常简称为 *cat-5*， 一种按照5类线规范建造的双绞线，非常适合于快速以太网。
    - **6类线**: 通常简称为 *cat-6*，一种按照6类线规范建造的双绞线，非常适合用于千兆网。
- **网口**: 不要与TCP或UDP端口号混淆，它们是完全不同的，在这里指的是设备上可以插入网线的物理插口。
- **以太网线缆**: 口语称呼，意思是插入以太网设备的双绞线。
- **交叉电缆**: 在电缆的一端交换发送和接收引脚的电缆。这通常用于将两个设备直接连接到彼此，而区别于大多数情况下使用交换机或集线器作为中介。如果你直接用电线将两台电脑直接连接起来，那么大概率是一根交叉电缆。和 *直连电缆* 相反。
- **自动侦测**: 一种可以分辨是直连电缆还是交叉电缆的网口，并且在需要的时候翻传输和接收信号。
- **细电缆**: 已经被淘汰的以太网上曾经使用的同轴电缆。
- **LAN**: 局域网。对于以太网，这将是你家中的网络。可以把它看作一个单个IP子网。
- **WAN**: 广域网。不是局域网的网络。比如公司校园里的大学局域网的集合。
- **动态 IP**: 这是你当设置通过DHCP自动获取IP地址，比如说，在这种情况下你的IP地址随着时间可能会改变。

- **静态 IP**: 这一般是当你对着一个特定的设备硬编码了IP地址，除非你输入了一个新的IP地址，不然该IP地址一直不会改变。

- **网络接口控制器（NIC），网络接口卡，网路适配器，局域网适配器，网卡**:一堆使得计算机接入网络的硬件设备的不同名称。这些硬件卡要么上面有以太网网口，要么是纯无线的。甚至，在现在他可能都不是一个真正的硬件卡--也许它和主板上内置的一堆其他I/O设备都是同一个芯片。

- **网络设备 (OS)**: 操作系统中的一种软件设备结构，通常映射到网卡。有一些网络设备，比如 *环回设备* 实际上并不使用硬件上面的任何物理部件，并且只在操作系统内部“传输”数据。在类Unix系统中，这些设备被命名为 `eth0`, `en1`, `wlan2`等等。环回地址通常被称之为 `lo`。

- **MAC 地址**: 媒体访问控制地址,一个计算机的唯一的链接层地址。在以太网中，MAC地址有6字节，通常它们是以冒号隔开的十六进制的字节: `12:34:56:78:9A:BC`。这些地址必须在局域网上是唯一的，这样才能在以太网中正常工作。

- **集线器**: 一种可以通过以太网电缆连接多台计算机的设备。一般该设备在其前面拥有4，8，或者更多的端口。所有这些插入这些端口的设备一旦连接，实际上都在同一条“电缆”上，也就是说，插入集线器的所有设备都可以看到该以太网上传输的**任何**数据包。你通常不会再看到这些设备，因为交换机将会可以更好地扮演相同的角色。

- **交换机**: 一个拥有大脑的集线器。它知道端口另一端的MAC地址，所以它不会向 *每个设备* 重新传输以太网数据包。它只是把数据包通过正确的线路送到正确的目的地。这非常有利于避免网路过载。

- **路由器**: 一种有多个接口的网络层的设备，它可以选择正确的接口发送流量，使其到达最终目的地。路由器中含有路由表，这些路由表可以让路由器判断出如何将指定的数据包发给正确的IP地址。

- **默认网关**: 一种路由器，它处理到所有其他目的地的流量，如果到达目的地的路由是未知的。计算机的路由表会将其发送给默认网关。在家庭局域网中，这是一个ISP会提供给你的“路由地址”。想象一个小岛上有一个小镇。这座岛通过一座桥与大陆相连。如果有人想知道在城里去哪里，你在城里给他们指路。对于**其他目的地**，他们会先开车过桥。在这个类比中，这里的桥梁就是流量的默认网关。

- **广播**: 将流量发送给局域网中的所有人。这可以通过在链路层发送给MAC地址为`ff:ff:ff:ff:ff`以太网帧来实现，或者在网络层发送IP数据包给所有网络位置`1`的地址。比如，如果子网是`192.168.0.0/16`，那么广播地址将是`192.168.255.255`。同样你也可以发送给`255.255.255.255`来达到同样的目的。IP路由器并不会转发IP广播数据包，所以广播数据包总是会严格的被限制在局域网内。

- **Wi-Fi**: *无线保真度*（另一个非技术性的带有双关意思的营销商标是Hi-Fi）的缩写，这是你的无线局域网连接。从链路层的角度来看。与使用以太网电缆非常相似，不同的是它使用无线电波而不是铜缆上的电。

- **防火墙**: 在LAN与ISP连接处或附近的计算机或设备，用于过滤流量，避免不需要的流量在LAN上传输，目的是希望能把坏人挡在外面。

- **NAT**: 网络地址转换。在NAT设备后面可以有一个私有IP子网，该子网对因特网的其他部分不可见。NAT设备将路由器内部的IP地址和端口转换为外部的IP地址。这就是为什么如果你去谷歌问“我的ip是什么”，你会得到一个不同的数字，而不是你在电脑上看到的数字。NAT位于内部LAN IP地址和外部公开可见的IP地址之间中间，对其进行转换。后面我们将详细讨论这种机制。

- **私有网络IPv4地址**: 对于没有连接到Internet的局域网或NAT后的局域网，有三个常用的子网:'`10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`。这些是保留给个人使用,没有公共站点会使用

它们，所以你可以把它们放在你的局域网中，而不用担心冲突。

- **WiFi调制解调器/WiFi路由器**: 指的是当你与ISP签约使用服务时，你得到的消费级设备和经常做各种各样的事情。
- **机架安装**: 如果一个设备没有一个漂亮的塑料外壳，或者不是为了直接插入电脑，它可能是机架安装。这些大型的非消费类设备，如路由器、交换机或磁盘组，被堆放在“机架”中。
- **上传**: 将一个文件从你的本地设备传输到一个远端设备。
- **下载**: 将一个文件从远端设备传输到你的本地设备。
- **对称网络**: 对于传输速度来说，意味着一个连接在两个方向上提供相同的速度。“1Gbs 对称网络”意味着上传和下载速度都是1Gbs。
- **非对称型网络**: 对于传输速度来说，意味着一个连接在两个方向上提供不同的速度。通常会用 "600 Mbs 下行, 20 Mbs 上行"来表示。比如，为了表示下载和上传速度，通常口头上称之为 "600 乘 20"。因为一般情况下是用户主要都是进行下载，很少上传，所以提供服务的公司在下载端分配了更多的总带宽。
- **线缆** (从线缆公司的角度):一些有线电视公司提供通过同轴电缆将你家里的设备接入互联网的操作，现时，就算速度高达1gb的网络也并非罕见。通常情况下，一组邻居共享带宽，所以你的网速会在晚上下降，因为你周围的人都在看电影。并且大多数公司提供的宽带都是都是非对称的。
- **DSL**: 数字用户线路。一些电话公司提供通过电话线将你家里的设备接入互联网的服务。它比电缆慢，大约100兆，但带宽不与邻居共享。大多数DSL产品也都是非对称的。
- **光纤**: *光导纤维* 的简称，这种技术使用的是光通过玻璃“线”，而不是通过铜线的电力。这使得其非常快，许多提供相对便宜的光纤的isp都有提供1Gbs对称传输速度的能力。
- **Modem**: *调制解调器* 的简称，其作用是将一种形式的信号转换为另一种形式的信号。历史上，它意味着将通过固定电话传输的声音转化为数据。在现代用法中，它意味着将以太网局域网上的信号转换为ISP需要的任何形式，例如电缆或DSL。
- **网桥**: 在链路层连接两个网络的设备，使得它们像单个网络一样工作。有些网桥盲目地转发所有的流量，而有些网桥更聪明。电缆调制解调器就是桥接器的一种，尽管它们通常与路由器和交换机内置在同一个盒子里。
- **刺穿式搭接器**: 在遥远的过去，当你想把电脑连接到一根粗网线上时，你会使用[这些名字很好听的设备](https://en.wikipedia.org/wiki/Vampire tap)之一来完成它。把它放在这里只是为了增加一点趣味性。

思考题

- 集线器和交换机的区别是什么?
- 路由器的作用是什么?
- 为什么只有一条网络连接的路由器没有任何意义?
- 当前你的笔记本电脑连接了哪些设备？你是否使用了一个物理线缆来进行连接?

- 这道思考题不需要写任何东西，除非你确实想回答点什么，但想想如果使用的是300 bps调制解调器，生活是什么样子的。我的第一个调制解调器是VICMODEM,它比现代的线缆连接慢两百万倍。不过这是40年前了，你可以畅想下到2062年，网络速度能快成什么样。

项目: 数据包跟踪器： 连接两台电脑

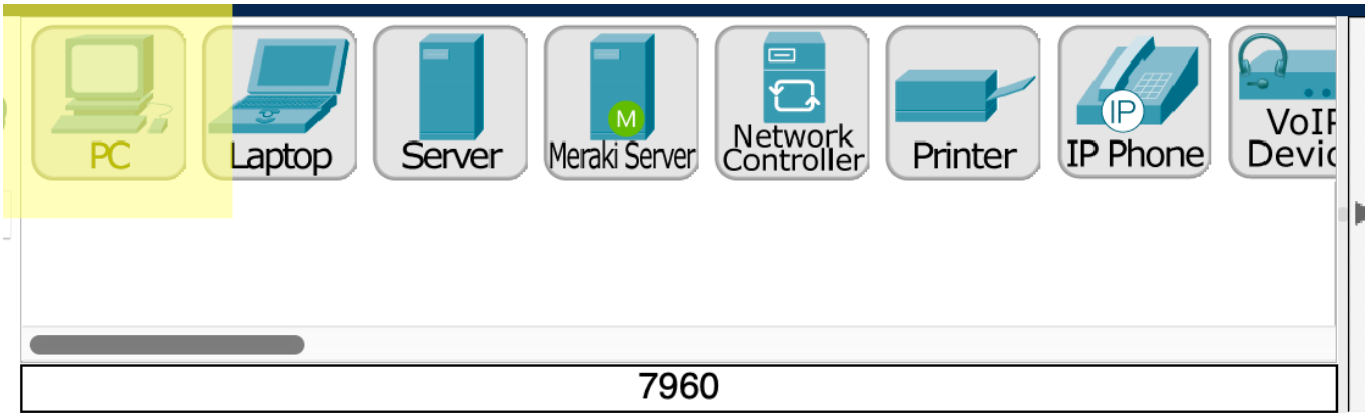
在本项目中，我们准备使用Cisco's [fl[Packet Tracer|https://www.netacad.com/courses/packet-tracer]]工具来搭建一个简单的网络，这个工具是免费的！如果你没有安装它，可以从这里Appendix: Packet Tracer查看到安装步骤。

如果在任何时候你犯了一个错误，需要删除，只需要从第二个图标行选择“Delete”工具即可。

向局域网中添加计算机

在左下方选择"End Devices"。

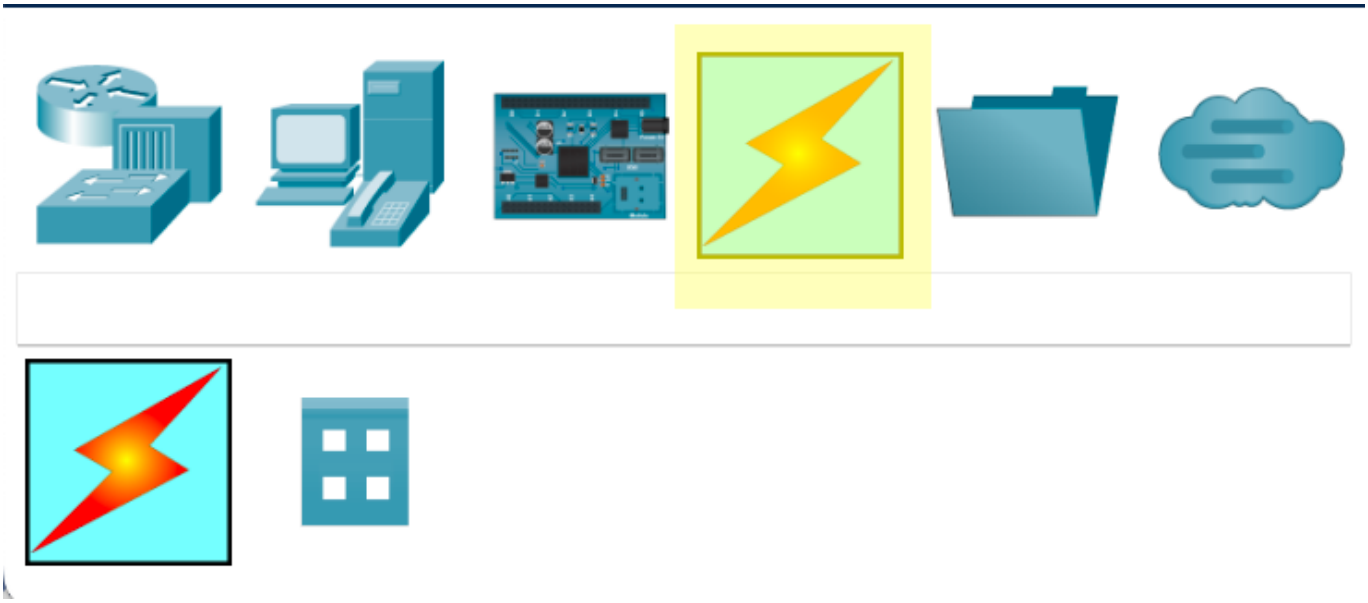
然后将两个“PC”拖到中下面板的工作区域。



将PC机用网线直连

现在，在你的工作区有两台计算机了，让我们将他们直连起来。

点击左下角的"Connections"按钮。

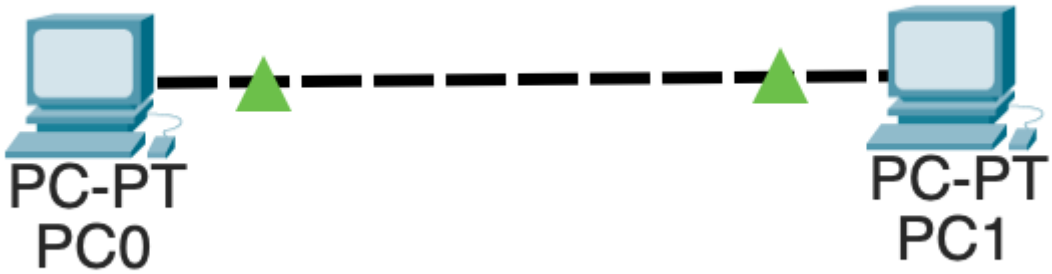


点击中间下面的"Copper Cross-Over"图标（该图标会在点击后变成"anti" 符号。）

点击其中的一个计算机，选择"FastEthernet0"。

点击另一个计算机，同样选择"FastEthernet0"。

此时，你应该可以看到类似下面的内容:



设置网络IP

这两台电脑都没有IP地址。我们将用我们选择的静态ip设置它们。

1. 点击PC0。
2. 点击"Config"面板。
3. 在侧面点击"FastEthernet0"。

4. 在"IPv4 Address"一栏,输入`192.168.0.2`。
5. 在"Subnet Mask"一栏，输入 `255.255.255.0`。

关闭配置窗口。

以相同的步骤配置PC1，除了在IP地址输入`192.168.0.3`，在结束时，关闭配置窗口。

在网络中ping联通性

让我们从PC0 ping PC1，以确保我们的网络是正确连接的。

1. 点击PC0，选择"Desktop"标签页。
2. 点击"Command Prompt"。
3. 在命令行窗口，输入`ping 192.168.0.3`。

> 当你开始ping时，第一个ping有可能会因为开始ARP而导致超时，在复杂的网络中，多次的ping可能会在成功之前超时。

当ping成功时，你应该会看到如下消息:

```
C:\>ping 192.168.0.3

Pinging 192.168.0.3 with 32 bytes of data:

Reply from 192.168.0.3: bytes=32 time<1ms TTL=128
Reply from 192.168.0.3: bytes=32 time<1ms TTL=128
Reply from 192.168.0.3: bytes=32 time<1ms TTL=128
Reply from 192.168.0.3: bytes=32 time<1ms TTL=128

Ping statistics for 192.168.0.3:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

如果你看到多于两次的`Request timed out.`，那么配置一定有错误了。

```
C:\>ping 192.168.0.3

Pinging 192.168.0.3 with 32 bytes of data:

Reply from 192.168.0.3: bytes=32 time<1ms TTL=128
Reply from 192.168.0.3: bytes=32 time<1ms TTL=128
Reply from 192.168.0.3: bytes=32 time<1ms TTL=128
Reply from 192.168.0.3: bytes=32 time<1ms TTL=128

Ping statistics for 192.168.0.3:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

保存该项目

请确保你保存了项目，如果你这样做了，你将会得到一个后缀名为`.pkt`的文件。

项目: 数据包追踪器: 交换机的使用

将两台电脑直连并不多见，在绝大数情况下，他们都是通过一个 *交换机* 连接起来。 让我们在实验中完成这个设置。

添加计算机

在左下方选择"End Devices"，并在工作区拖进三台PC机。按顺序点击每个PC机，转向他们的"Config"标签页点击`FastEthernet0`并给他们赋予IP地址:

- PC0: `192.168.0.2`
- PC1: `192.168.0.3`
- PC2: `192.168.0.4`

他们应该全部使用掩码`255.255.255.0`。

添加一个交换机

在左下角点击"Network Devices"，左下方的图标将会变化。 选择"Switches", 最下面一行的左边第二个，中间的面板将会改变。

将一个`2960`交换机拖到工作区。

如果你单击交换机并查看“Physical”标签页，你将看到交换机是一个带有许多以太网端口的设备。(这是一款相当高端的交换机，而家用交换机通常有4或8个端口。家里的WiFi路由器背面可能有4个这样的端口——这是一个内置的交换机!)

我们可以把PC连接到这些端口上，这样它们就可以互相通信了。

连接起来

没有PC机会直接互相连接，它们都直连交换机，在这里，我们不会使用交叉电缆，交换机知道它在干什么。

> 请注意，当你第一次连入局域网中时，你可能不会看到连接上两个绿色向上的箭头。其中一个或 两个可能是橙色圆圈，表示链接正在出现。你可以点击左下角的`>>`快速前进按钮，及时跳转， 直到你看到两个绿色向上的箭头。 Note that when you first wire up the LAN, you might not have two green up arrows shown on the connection. One or both of them might be orange circles indicating the link is in the process of coming up. You can hit the `>>` fast-forward button in the lower left to jump ahead in time until you get two green up arrows.

选择左下角的"Connections"选择按钮。选择"Copper Straight-Through"线缆。（这时图标将会变换称为"anti"图标。）

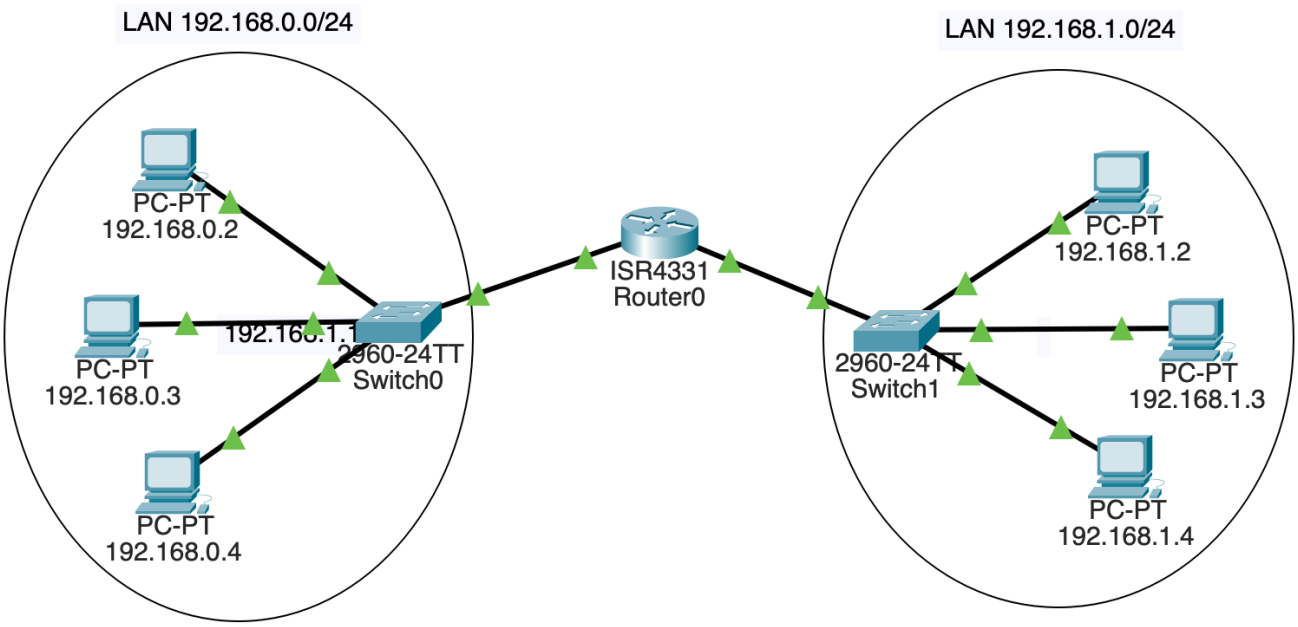
点击PC0， 选择`FastEthernet0`。接着点击交换机，然后选择任何`FastEthernet0`端口。对另外的两台PC机重复同样的操作。

Ping测试！

单击其中一台pc，转到“Desktop”标签页并运行命令提示符。确保你可以ping通另外两台电脑。

项目: 数据包追踪器：使用一个路由器

我们将在数据包跟踪器项目中的两个独立网络之间添加一个路由器。这次需要完成的目标大约如下所示:



搭建局域网

首先，我们需要建立两个独立的局域网。像以前一样，每个人有三台pc和一个2960交换机。在这里左边的子网将会是`192.168.0.0/24`。右边的子网将会是`192.168.1.0/24`。给左边的PC机的`FastEthernet0`接口上都设置下如下IP:

- `192.168.0.2`
- `192.168.0.3`
- `192.168.0.4`

给右边的PC机的`FastEthernet0`接口上都设置下如下IP:

- `192.168.1.2`
- `192.168.1.3`
- `192.168.1.4`

所有的这些PC机都有相同的子网掩码 `255.255.255.0`。将左边的PC机连接到其中一个交换机上的任何没有使用的`FastEthernet`端口上，将右边的PC机以同样的方法连接到另外一个交换机上。此时，你将拥有两个独立的局域网。

如果不是所有的链接都是绿色的，点击快进按钮。如果它们一直是红色，那就有问题了——你得再次检查你的设置。为了安全起见，请确保从`192.168.0.2` 可以ping通局域网上的其他两台机器。并确保从`192.168.1.3`也可以ping通LAN上的其他两台机器。

添加一个路由器

路由器的作用是在两个网络之中路由流量。

点击左下的"Network Devices"按钮，然后点击最底部的 "Routers"，接着将一个`4331`路由器拖到两个路由器之间。路由器将会同时连接到两个交换机上，路由器的每一“边”都有一个不同的IP地址。

我们将会在这里做一些连接线，不过 *连接还不会起来,* 下面我们将会处理此事。

对于位于`192.168.0.0/24`子网的交换机，使用一条铜直通线（Copper Straight-Through）将交换机的`GigabitEthernet0/1`端口连接到路由器的`GigabitEthernet0/0/0`端口。

对于位于`192.168.1.0/24`子网的交换机，选择同样类型的连接线将交换机的`GigabitEthernet0/1`端口连接到路由器的`GigabitEthernet0/0/1`端口。

注意，这是一个路由器其他交换机连接的不同的端口!两个交换机都插入路由器的不同端口。

我们现在有了硬件，但是我们还没有在路由器上分配任何IP地址。我们来完成该操作。

在路由器上的"Config"配置界面，选择`GigabitEthernet0/0/0`接口并且给它赋予一个IP地址`192.168.0.1`--现在它是子网`192.168.0.0/24`一部分。当你完成这一步之后，点击"On"来打开该端口上的电源。这一步会打开连接并显示绿色箭头。

接着，找到`GigabitEthernet0/0/1`接口，并且给它赋予一个IP地址 `192.168.1.1`--现在它是子网`192.168.1.0/24`的一部分。你完成这一步之后，点击"On"来打开该端口上的电源。

让我们尝试去ping另一个路由器，找一台位于`192.168.0.0/24`子网上的任一PC并且ping`192.168.0.1`（路由器），此时应该会收到回复。接着找一台位于`192.168.1.0/24`子网上的任一PC并且ping`192.168.1.1`，此时应该会收到回复。

现在是终极测试: 打开PC机`192.168.0.2`的终端，并且尝试去ping位于另一个子网上的`192.168.1.2`。呀，这没有连通，那么这是为什么呢?

添加一个默认路由

所有的电脑现在需要知道以下两件事:

- 局域网的子网号是什么，以便它知道在局域网上路由本地流量。
- 一个 *默认网关* 地址，位于局域网上的计算机需要知道该地址，这样才能把流量发送到局域网之外的地址。

我们已经添加了子网，但没有添加默认网关。

对于所有位于`192.168.0.0/24`子网上的计算机，点击"Config",选择侧边栏上的"Global"，在"Default Gateway"中输入`192.168.0.1`（我们的路由器！）

对所有位于另一个子网中的所有PC机做同样的设置，除了在这个子网中输入`192.168.1.1` 作为默认网关地址。

试一试!

现在你应该可以从任一PC ping到另一个PC！如果不能，请尝试只在一个局域网上ping以确保它能工作。然后ping一下局域网的路由器，来检测是否能正常工作。

项目： 数据包追踪器： 多个路由器

在上一个数据包追踪的项目中，我们在两个子网中间设置了一个路由器。在本项目中，我们将在两个子网中间设置多个路由器。这使问题变得复杂，因为每个路由器的路由表需要编辑，以便它知道如何转发数据包。

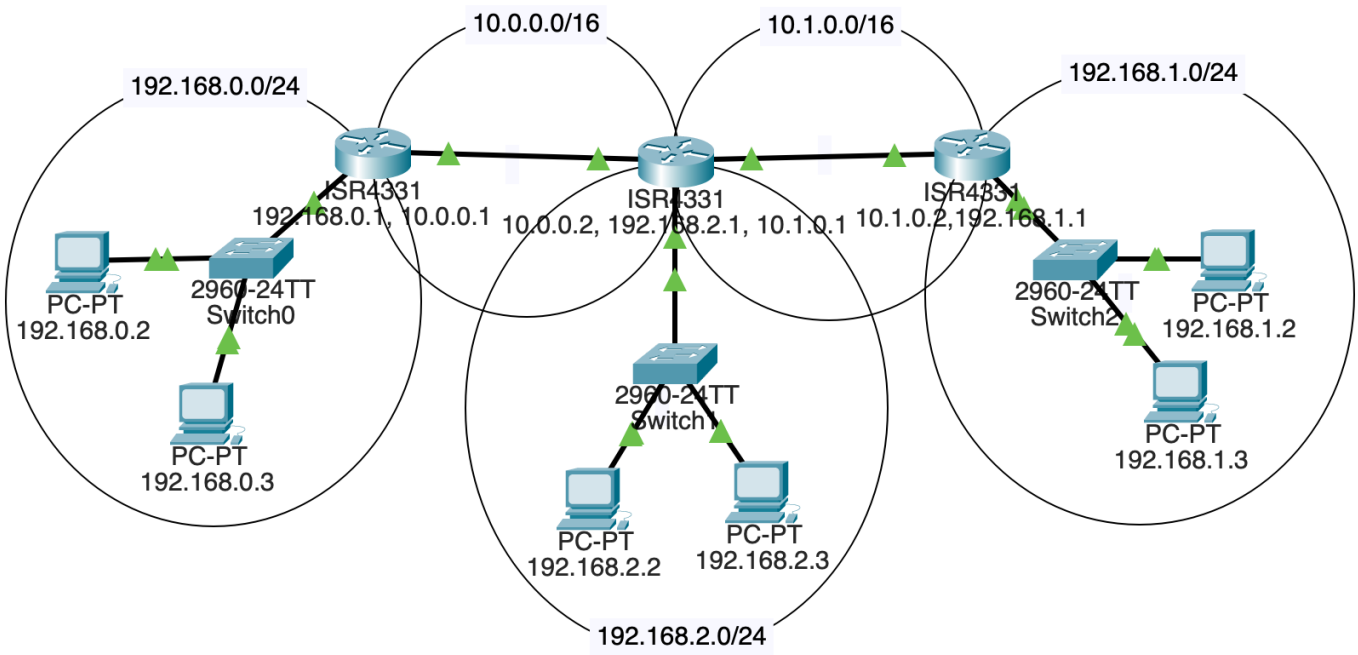
在一个庞大的局域网中，网关协议可以用来快速分发路由器所需的信息。但在我们的例子中，为了保持概念上的简单，我们只需要手动配置三个路由器中的路由表。

我们正在构建的

五个子网！

- 其中的三个是局域网:
  - `192.168.0.0/24`
  - `192.168.1.0/24`
  - `192.168.2.0/24`
- 剩下的两个位于两个路由器:
  - `10.0.0.0/16`
  - `10.1.0.0/16`

这里有一个示例图:



拖出组件

目前我们将需要:

- 每个局域网中有两台主机，所以一共有6台主机
- 3个2960交换机
- 3个4331路由器

每一个局域网都连接到一个路由器。其中两个路由器同时也连到另一个路由器。

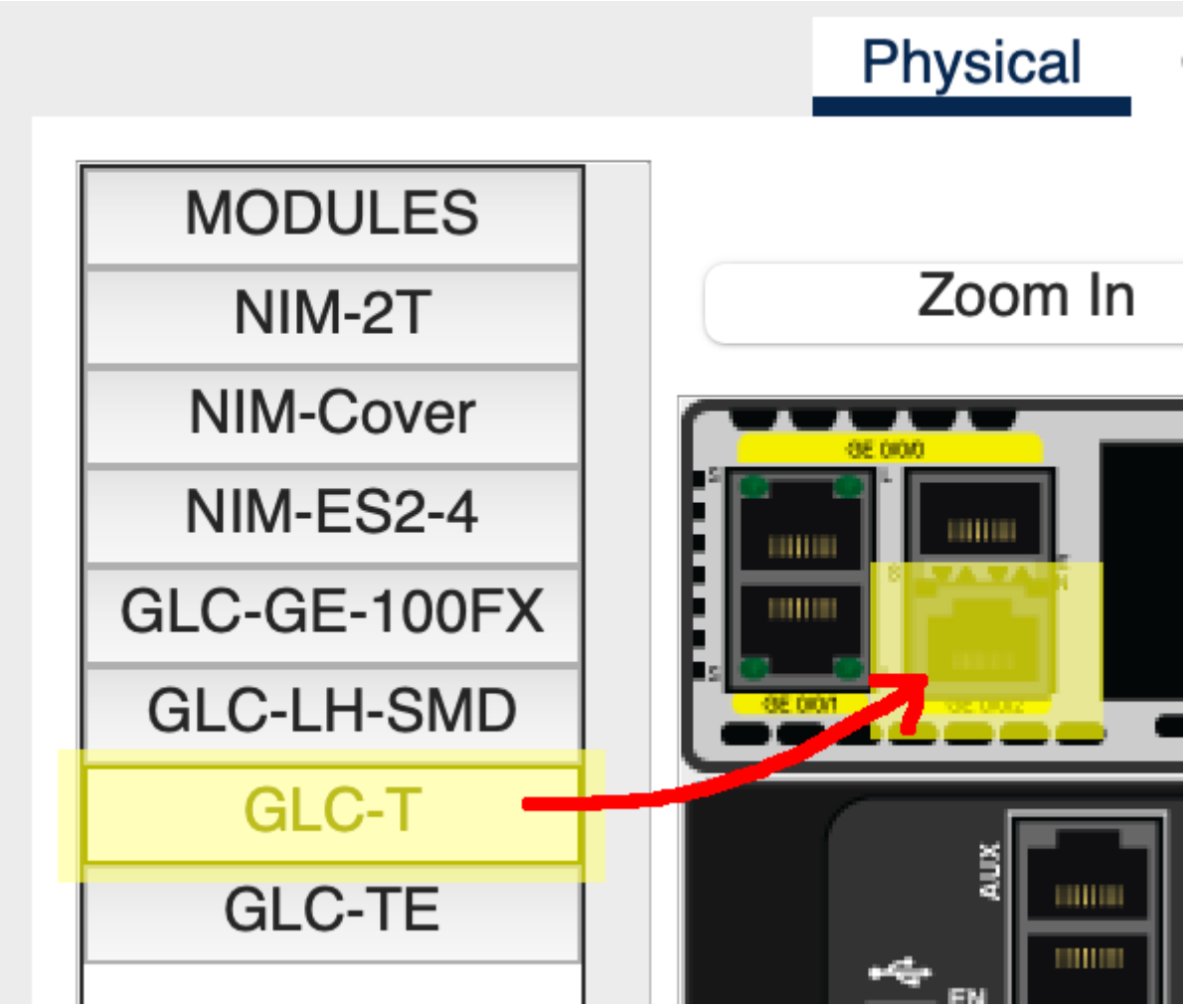
不过，**这使得设置有点有趣的是**，中间的路由器和另外两个路由器**以及**另一个局域网相连！如图所示，在组件之间放置直通铜线连接。

设置中间的路由器

中间那个路由器需要连接另外两个路由器和一个局域网的?默认情况下，它没有足够的端口。我们需要给他增加一个端口。幸运的是，这是一个虚拟的的模拟，所以你可以模拟虚拟支付的新组件，而不必打开你真实的钱包。

选择那个路由器并且选择"Physical"标签页。点击"Zoom In"放大，这样可以看的更清楚些。电源开关在右侧。滚动到那里，点击它。(因为在断电之前不能添加组件。)

其中两个以太网连接器位于左上方。就在它们的右边，还有两个我们可以插入组件的端口。从左边的侧边栏，拖拽出一个"GLC-T"到这些端口上，如下所示:



最后，重新打开路由器的电源。

设置三个子网

我们使用如下子网:

- `192.168.0.0/24`
- `192.168.1.0/24`

- `192.168.2.0/24`

按照惯例，路由器通常都是子网中的`.1`地址。例如:`192.168.2.1`。但是这不是必需的。在子网中给2台pc机和1台路由器分配IP地址，然后把它们都连接到一个交换机上。

确保路由器上正确的以太网端口在配置中被设置为“on”!完整性检查:一个子网上的所有计算机都应该能够相互ping通 **并且可以ping通**它们的路由器。

给所有主机都设置好默认网关

请记住，当从一个主机发送流量时，它们要么知道该发送到自己的子网上(因为目的地在同一个子网上)，要么不知道目的地IP在哪里。如果它们识别到IP不在同一子网上，它们会将流量发送到 *默认网关*，也就是那个一定知道如何处理目的地的路由器。

依次点击每台电脑。在侧边栏"Global/Settings"中的"Config"下，设置该局域网"Default Gateway"路由器静态IP。举例来说，如果我是`192.168.1.2`的PC机，而位于该子网的路由器地址是`192.168.1.1`，我将设置该PC机的默认网关是 `192.168.1.1`。

实际上，我会把局域网上所有pc的默认网关设置为该值。接下来，对另外两个子网执行相同的操作。

设置路由器子网

为了路由器能够正确的工作，我们需要在左边和中间路由器之间设置一个子网，在中间和右边之间设置另一个子网。

我们使用如下子网:

- `10.0.0.0/16`
- `10.1.0.0/16`

这意味着左右路由器将有两个IP地址，因为它们连接到两个子网。但是中间路由器将有三个IP地址，因为它连接到三个子网!(即其中连接到一个局域网，而其他两个连接到另外两个路由器。)

如果你还没有这样设置，请使用铜线直通连接连接子网。

设置路由表

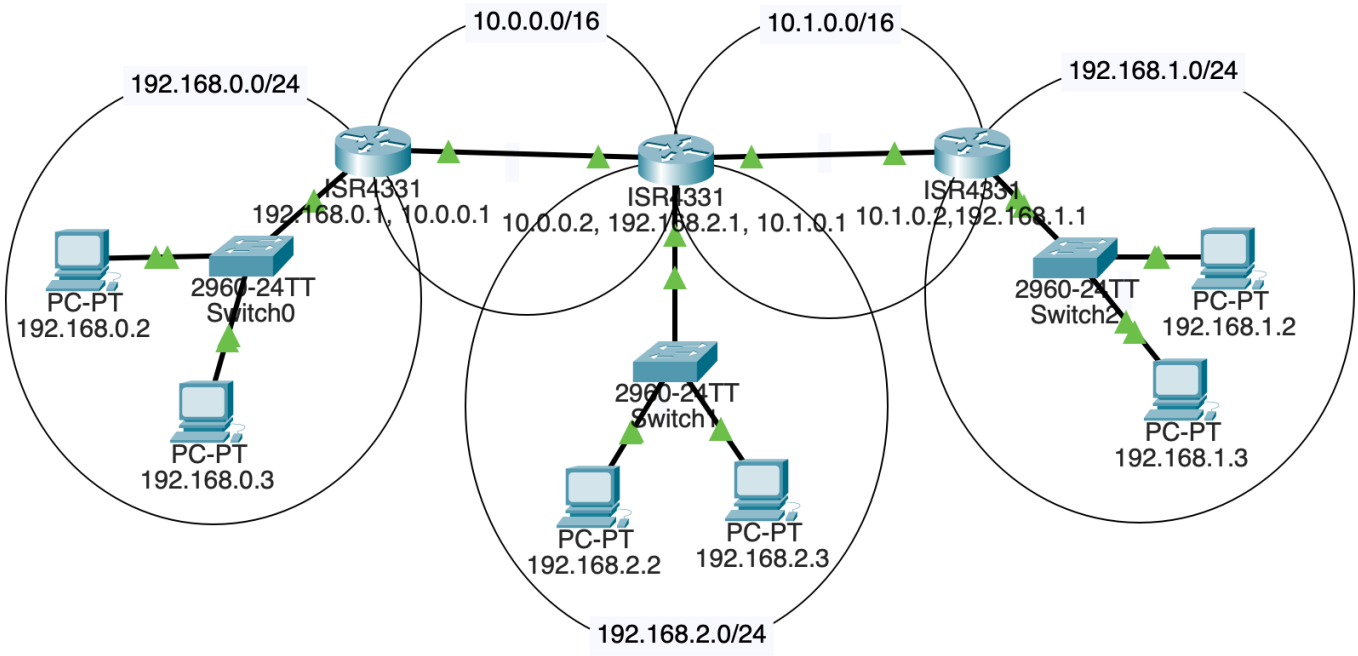
我们快要完成了，不过如果你使用`192.168.0.2`地址去ping`192.168.1.2`，将无法了产生连通的流量。这是因为位于子网`192.168.0.0/24`的路由器很笨不知道从哪里可以发送出要到达`192.168.1.2`的数据包。

我们需要将这个部分补上。

我们将手动为每个路由器添加“静态路由”，这样它们就知道要将东西发送到哪里。如之前章节介绍的，在实际的局域网中，更常见的是使用网关协议来自动设置这些路由表。

但这是我们这里是学习目的的实验，如果使用网关协议自动添加有什么好玩的?(实际上，这一节将是一个非常有用的练习，但是该知识也是一个 *非常有用* 的知识点。)

让我们再次来查看下网络图表:



如果一个数据包是需要从左边的192.168.0.3发送到192.168.1.2（右边），它怎样才能到达那里？我们可以看到它必须通过所有三个路由器。但是当它到达第一个地址192.168.0.1(LAN的路由器)时，该路由器将会把它发送到哪里?

好吧，从那里，数据包将从 10.0.0.0/16 子网出发到路由器10.0.0.2。

因此，我们必须为最左边的路由器添加一条路由，说:“嘿，如果你得到了子网192.168.0.0/24的任何信息，请将其转发到 10.0.0.2，因为它是途中的下一跳地址。”

我们点击最左边的路由器，然后进入“Config”，接着进入左侧栏的“Routing/Static”。将目的地对应“Network”和“Mask”字段填入对应信息，“Next Hop”是我们应该转发到流量的路由器。

在图中，我们像这样将一个路由器连接上最左侧的路由器(回想一下' \24网络的掩码是 255.255.255.0):

```
Network:  192.168.1.0
Mask:     255.255.255.0
Next Hop: 10.0.0.2
```

我们已经快要完成了！但是，可悲且重要的是，目前中间路由器也不知道将192.168.1.0/24的流量发送到哪里。因此，我们必须添加一条路由到中间路由器，使其能够将数据包发送到下一跳，但这次位于它的10.1.0.0/16接口之外:

```
Network:  192.168.1.0
Mask:     255.255.255.0
Next Hop: 10.1.0.2
```

现在，大功告成！IP为10.1.0.2 的路由器有一个连接到192.168.1.0/24的接口。我们的原始数据包将到达192.168.1.2，并且它们还位于在同一子网上!这时路由器就知道它可以通过该接口发送相应的流量。

当然，这并不是我们全部都需要做的。

接下来，为每台路由器添加上所有非直连子网的路由表项。而每个路由器应该有两个静态路由表项，以便涵盖所有入站和出站流量。

试一试！

如果所有配置正确，你应该能够从任何PC ping通任何PC!路由器可以将流量正确的转发到其他局域网!

Select函数

在这一章中，我们来看看 `select()` 函数。这是一个轮训系统所有套接字并让你知道哪些套接字向你发送了数据的函数。简单说就是准备调用`recv()`的对象。

这个功能使我们能够同时在大量套接字上等待数据。

我们要解决的问题

让我们假设有三个客户端需要连上我们的服务器，它想要`recv()`任何从客户端发过来的消息。但是，作为服务器端程序，它没有任何办法知道哪个客户端接下来将会发送什么数据。

此外，当服务器端对一个还没有数据到达的socket调用`recv()`时，`recv()`调用会产生 *阻塞*， 此时其他程序就不能运行了。

> *阻塞* 的意思是进程将会停止执行并且在触发条件满足之前一直保持睡眠状态。在上面描述的 调用`recv()`的例子里，进程会一直保持睡眠状态知道收到一些数据。 To *block* means that the process will stop executing here and goes to sleep until some condition is met. In the case of `recv()`, the process goes to sleep until there's some data to receive.

因此，在这种状况下，我们会遇到如下的问题:

```
data1 = s1.recv(4096)
data2 = s2.recv(4096)
data3 = s3.recv(4096)
```

假设`s1`上没有准备好的数据，那么进程将阻塞在那里，此时在 `s2`'或`s3`上调用的`recv()`将会一直等待，即使在这些套接字上有要接收的数据，函数也不会被调用。

所以，这里需要一个方法同时监控`s1`, `s2`, 和 `s3`，以判断它们之中哪个已经有数据准备好被接收了，接着只在这些套接字上调用`recv()`。

这就是`select()`函数所做的，在一系列套接字上调用`select()`会产生阻塞，直到这其中的一个或者多个套接字进入了就绪读取的状态。然后它返回给你哪些套接字准备好了，你可以在那些上调用`recv()`。

使用 `select()`

首先，你需要导入 `select`模块。

```
import select
```

如果你准备测试一系列的已经有数据准备`recv()`并且已连接的套接字，你可以将它们放在一个`set()`中，然后传递给`select()`函数。该函数会产生阻塞知道有数据可以被读取。

此集合可以作为已连接套接字的标准列表，你需要把它们都记录在某个地方，集合就是一个不错的选择。当获得新的连接时，将它们添加到集合中，当连接挂起时，将它们从集合中删除。通过这种方式，它总是保存

所有当前连接的套接字。

下面是一个示例，`select()`函数有三个参数，并且返回三个值，我们现在只看其中的第一个，忽略其他的。

```
read_set = {s1, s2, s3}

ready_to_read, _, _ = select.select(read_set, {}, {})
```

此时，我们可以通过准备好的套接字来接收数据。

```
for s in ready_to_read:
    data = s.recv(4096)
```

使用`select()`处理监听套接字

如果你认真思考，你可能有以下问题:如果服务器被阻塞在`select()`调用，并且在等待传入的数据，为什么它还能调用`accept()`来接受传入的连接?这些传入的连接不需要等待吗?此外，既然如果`accept()`被阻塞了，那么我们将如何回到`select()`，如果我们是在哪里被阻塞的?

幸运的是，`select()`函数给我们提供了如下的选项: *你同样可以添加监听套接字到初始集合中!* 当监听套接字显示为“ready-to-read”时，这意味着有一个新的连接传入到`accept()`。

主要算法

让我们把所有的这些都放在一起，我们可以使用如下伪代码循环来完成`select()`功能:

```
将监听套接字添加到集合

main loop:

    调用select()函数并且获得可以进行读取的套接字

    for 所有已准备就绪读取的套接字:
        if 套接字是一个监听套接字:
           调用accept()建立新连接
           将新的套接字加入我们的集合!

        else 如果套接字是一个普通的套接字:
            从该套接字recv() 数据

            if 收到了0个字节
               客户端已经挂掉
               从集合中删除该套接字
```

`select()`函数中其他参数有什么用?

`select()` 函数包含三个参数。（虽然对于这个项目，我们只需要使用第一个，所以这一节纯粹是额外知识。）

它们分别是:

- 你想要监控哪些套接字为准备就绪可读状态
- 你想要监控哪些套接字为准备就绪可写状态
- 你想要监控哪些套接字为发生异常

下面的返回值也相应的映射到这些不同的值。

```
read, write, exc = select.select(read_set, write_set, exc_set)
```

不过，对于本项目，我们仅仅只使用第一个返回值并且忽略其他的。

```
read, _, _ = select.select(read_set, {}, {})
```

超时

我撒了个小谎，其实还有第四个可选参数，`timeout`。它是等待事件发生的秒数，以浮点数表示，如果在该时间范围内没有发生任何事情，`select()`函数直接返回，并且所有返回的套接字都不会显示为就绪状态。如果你只是想轮询所有套接字，你可以设置超时为`0` 。

思考题

- 为什么我们不能简单的对所有已链接的套接字直接调用`recv()`? `select()`给我们带来了什么好处?
- 当`select()`函数显示"已就绪有数据可读"时，此时对于一个监听套接字和非监听套接字的差别是什么?
- 为什么我们也要监听套接字加到集合中? 为什么我们不直接先调用`accept()`，再调用`select()`?

在`send()`函数中使用`select()`

如果你的计算机尝试发送的速度太快，那么对于`send()`的调用同样可能会发生阻塞。也就是说，操作系统会使得其在处理待发送的积压数据时处于休眠状态。

但是，假设这里你真的不想有任何阻塞调用而只想保持不停的处理发送请求。你可以对`select()`函数的第二个参数是传入一个含有`send()` 函数使用套接字描述符的集合，这时，就不会发生阻塞操作。

这里，其工作原理和上面“套接字已经准备好读取”集合差不多。

项目： 使用Select函数

在本项目中，我们将要使用`select()`来编写一个同时处理多个连接的服务器端程序。我们已经提供了客户端程序，你只需要完成服务器端代码即可。

演示代码

你可以从[fls[this ZIP file|select/select.zip]]获取所有输入文件。`select_client.py` 已经全部完成了，你需要做的是在`select_server.py`填入对应的代码。

需要添加的功能

你的服务器端程序需要有如下的功能:

- 当一个客户端连接时，服务器端程序按以下格式打印出客户端连接信息:

  ```
  ('127.0.0.1', 61457): connected
  ```

- 当一个客户端断开连接时，服务器端程序用以下形式打印出连接信息:

  ```
  ('127.0.0.1', 61457): disconnected
  ```

  提示: 即使已经断开连接，你依然可以在套接字上使用`.getpeername()`方法来获取到远端的ip地址。它将返回一个含有`("host", port)`的元组，就像你传递给`connect()`函数的参数一样。

- 当一个客户端发送数据时，同样，服务器端需要打印出收到的原始字节字符串的数据长度:

  ```
  ('127.0.0.1', 61457) 22 bytes: b'test1: xajrxttphhlwmjf'
  ```

示例运行

运行如下服务器端程序:

```
python select_server.py 3490
```

运行客户端程序:

```
python select_client.py alice localhost 3490
python select_client.py bob localhost 3490
python select_client.py chris localhost 3490
```

客户端的第一个参数可以是任何字符串——服务器端程序会将它与数据一起打印出来，以帮助你识别出它来自哪个客户端程序。

示例输出: Example output:

```
waiting for connections
('127.0.0.1', 61457): connected
('127.0.0.1', 61457) 22 bytes: b'test1: xajrxttphhlwmjf'
('127.0.0.1', 61457) 22 bytes: b'test1: geqtgopbayogenz'
('127.0.0.1', 61457) 23 bytes: b'test1: jquijcatyhvfpydn'
('127.0.0.1', 61457) 23 bytes: b'test1: qbavdzfihualuxzu'
('127.0.0.1', 61457) 24 bytes: b'test1: dyqmzawthxjpkgpcg'
('127.0.0.1', 61457) 23 bytes: b'test1: mhxebjpmsmjsycmj'
('127.0.0.1', 61458): connected
('127.0.0.1', 61458) 23 bytes: b'test2: bejnrwxftgzcgdyg'
('127.0.0.1', 61457) 24 bytes: b'test1: ptcavvhroihmgdfyw'
('127.0.0.1', 61458) 24 bytes: b'test2: qrumcrmqxauwtcuaj'
('127.0.0.1', 61457) 26 bytes: b'test1: tzoitpusjaxljkfxfvw'
('127.0.0.1', 61457) 17 bytes: b'test1: mtcwokwquc'
('127.0.0.1', 61458) 18 bytes: b'test2: whvqnzgtaem'
('127.0.0.1', 61457): disconnected
('127.0.0.1', 61458) 21 bytes: b'test2: raqlvexhimxfgl'
('127.0.0.1', 61458): disconnected
```

域名系统（DNS）

我们已经了解到IP协议负责在互联网上路由流量。还学到它可以对IP地址进行处理，通常我们以点和数字格式展示IPv4地址，例如`10.1.2.3`。不过作为人类用户，我们很少使用IP地址，当你使用网页浏览器时，基本上你不会将一个IP地址直接输入到地址栏。

即使在我们的项目中，我们都倾向于输入`localhost`，而不是直接输入`127.0.0.1`来表示本地地址。在计算机网络中，一般将我们使用的`www.example.com`等名称转换为计算机使用的IP地址的过程称之为 *域名解析*，它由称之为 *域名系统*(DNS)的分布式服务器组来完成。

常见用法

从用户的角度来看，我们为设备配置一个“名称服务器”，这些设备可以通过和它连接，将名称转换为IP地址。(这可能是用DHCP配置的，稍后会详细介绍。)当你尝试去连接`example.com`时，你的计算机会首先连接名称服务器来询问IP地址。

如果该名称服务器知道上述问题的答案是什么的时候，它将直接提供结果，如果它不知道，那么一系列齿轮就要开始转动了。让我们深入了解下整个过程。

域名和IP地址

如果你从未注册过一个域名（比如说 `example.com`或者`oregonstate.edu`或者`google.com`或者`army.mil`），那么整个过程会很像是:

1. 联系 *域名注册商* (比如，一些拥有售卖全部域名权利的公司)。
2. 选择一个还没有人使用的域名。
3. 每年向他们付费来使用这些域名。
4. ...
5. 从中盈利！

但是这样做与IP地址的概念就完全脱节了，事实上，没有IP地址的域名也可以存在——它们只是不能被使用。一旦你有了你的域名，你可以联系一个托管公司，他们会为你提供一个供你使用的服务器上的IP地址。

现在你有两样东西了:域名和IP地址。不过，你仍然需要将它们关联起来，这样用户才能通过你的域名地址查询到你的IP地址。为此，你需要将包含相关信息的数据库记录添加到一个叫做DNS地方:*域名服务器*。

（域名）名称服务器

通常我们简称它为 *name servers*, 这些服务器包含它们有 *权限* 的所在域的IP记录。也就是说，没有一个域名服务器含有全世界的记录，每个服务器只服务一个特定的域或子域。

> *子域* 是一个由域拥有者创建的域。举例来说，`example.com`的拥有者可以创建子域`sub1.example.com`和`sub2.example.com`。在这些例子中，这些都不是真实的主机, 但是它们都可以被分配给真实的机器，举例来说，`host1.sub1.examople.com`, `host2.sub1.example.com`,`somecompy.sub2.example.com`。
>
> 域所有者可以根据需要创建任意多的子域。他们只需要确保有一个域名服务器来处理它们即可。

一个被授权的名称服务器可以被特定域中任何主机查询。主机号通常是域名的第一个“部分”，但是其实它不是必须的。

举例来说，`www.example.com`，这是一个位于域`example.com`的名叫`www`的主机。一个单独的命名服务器需要被很多域授权。

但是，即使名称服务器不知道它被要求提供的域名的IP地址，它也可以向其他名称服务器进行查询。从用户的角度来看，这个过程是无感的。

所以,非常简单。如果不知道被查询的域名，我只要去联系属于那个域的名称服务器以获得答案。

根域名服务器

然而，我们有个问题，如果我都不知道域名的名称服务器是什么，我如何连接到域名服务器?

为了解决这个问题，互联网上有一系列被称为 *根域名服务器* 一路上指引我们。当我们不知道IP地址的时候，我们可以从它们这里开始，它们要么直接告诉我们IP地址，要么告诉我们应该继续询问哪个服务器。稍后会详细介绍这个过程。

在互联网上，预先配置了13个计算机作为根域名服务器的IP地址。这些ip很少改变，并且只需要其中一个就可以工作。负责DNS的计算机经常检索该列表以保持最新的信息。

根域名服务器为自己依次取名`a` 到 `m`:

```
a.root-servers.net
b.root-servers.net
c.root-servers.net
...
k.root-servers.net
l.root-servers.net
m.root-servers.net
```

过程示例

让我们从查询一台域名为`www.example.com`的计算机开始。我们需要知道它的IP地址，但是我们不知道哪个域名服务器负责`example.com`域名，但是我们知道根域名服务器的列表。

1. 首先计算机会随机选择一个根服务器，比如说`c.root-servers.net`，我们将会联系这个服务器并且说，“你好，我们正在寻找`www.example.com`，你能帮帮我们吗？”但是，根域名服务器并不知道这个问题的答案，所以它说，“我不知道答案，但是我可以告诉你，你想查询任何关于`.com`域名的信息，你可以去询问下面这些域名服务器中的任何一个。”随后，它会给出一个知道`.com` 域名的服务器列表:

2. 此时，我们只要选择其中一个管理`.com`域名的服务器，然后和它说：“你好，`h.gtld-servers.net`，我们正在寻找 `www.example.com`。你能给予帮助么？“随后，它回答：”我不认识这个名字，不过我知道`example.com`的域名服务器，你可以询问下面任何一个”。同样，它返回了一系例知道`example.com`域的域名服务器名单：

3. 和上面过程一样，我们随意选择名单中的一个服务器，对它说：“你好，`a.iana-servers.net`，我们正在寻找`www.example.com`。你能提供帮助么？”，然后该域名服务器回答：“是的，我可以！我熟悉这个名字！它的IP地址是`93.184.216.34`!"

所以对于任意查询，计算机都会从根服务器开始，随后它会指引计算机去发现更多的信息。（除非信息已经被缓存到某个地方，但稍后会详细介绍。）

区

在互联网中，域名系统被划分为多个逻辑管理单位，称之为 *区*。区是在特定域名服务器权限下的域的集合。不过上面的叙述过于简化了，同一区中可以有一个或多个域，而且可能有很多域名服务器在同一区工作。

可以将这些区看做是一种对于所有它负责的域名以及子域名的管理单位的集合。举例来说，对于根区，我们可以看到一系列它负责查找的域名服务器，同样对于`.com`区，也有一系列它授权管理的域名服务器。

解析库

当你在你编写的代码中使用域名时，它将会调用一个库来做DNS查询。可能你已经注意到了在Python中的如下代码：

```
s.connect(("example.com", 80))
```

此时，你无需担心任何有关DNS的信息，在这个调用的背后，Python会自动帮你从DNS中查询到相应域名的信息。在C语言中，有一个叫`getaddrinfo()`承担者同样的功能。

简单来说就是，在各种语言中，都有一个库来完成此项功能，而作为程序编写者自己，无需做任何改动。

而操作系统还有一条记录，其中包含用于查找的默认名称服务器。(大部分时候是手工配置，但更常见的是通过DHCP进行配置。)因此，当你查找请求时，你的计算机首先会转向这个服务器。

不过等一下--上述的过程如何能与根服务器层次结构对应起来呢?答案是:缓存系统!

缓存服务器

想象一下在全局范围内发生的所有DNS查找操作，如果每个请求都必须访问根服务器，这不仅会花费很长时间来处理重复的请求，而且根服务器也会受到严重的影响。为了避免这种情况，所有的DNS解析库以及DNS服务器都会 *缓存* 它们的结果。

> 作为一个最上层的服务器，根服务器每天要处理上万亿的请求。 As it is, the root servers handle literally trillions of requests per day.

通过这种方式，我们可以避免重复请求使根服务器而导致过载。

所以，我们得修改一下已经学习过的知识点。

1. 首先计算机向我们的解析器库查询IP地址。如果它已经缓存了，则直接返回。
2. 如果缓存中没有条目，去域名服务器查找对应的IP地址，如果域名服务器中有缓存，则直接返回对应的IP地址。
3. 如果在域名服务器内仍然没有缓存 **并且** 该域名服务器有上层的域名服务器，它会向更上层的域名服务器寻找答案。

4. 如果更上层的域名服务器依然没有缓存的结果，并且也没有更上层的域名服务器了，该域名服务器会向根服务器发起查询请求并进行之前我们阐述的过程。

如果每次都能命中缓存，那么这对于分担根域名服务器的压力大有帮助。很多你家里的WiFi路由器也会缓存域名服务器信息。所以当DHCP开始配置你的计算机时，你的计算机会使用路由器作为局域网内计算机的DNS服务器。这将为DNS查找提供快速响应，因为你的计算机到路由器的ping时间非常短。

存活时间

由于域名或主机的IP地址可能会改变，我们必须有一种方法来使缓存条目过期。

这是通过DNS里面的一个被称之为 *存活时间（TTL）* 的字段来实现的。 这是服务器应该保持缓存的秒数，通常它被设置为86400秒（1天），不过这个数字都可以根据区管理员认为IP地址多久应该变化的结果而改变。当一个缓存条目过期时，如果有人请求，名称服务器将不得不再次请求来自上游或根服务器的数据。

记录类型

到目前为止，我们已经讨论了使用DNS将主机或域名映射到相应的IP地址，这些是DNS服务器上的域存储记录类型之一。

一些常见的记录类型有:

- `A`:IPv4地址记录。这就是我们一直在阐述的类型。用来回答问题“此主机或域的IPv4地址是什么?”
- `AAAA`:IPv6地址记录。用来回答问题“此主机或域的IPv6地址是什么?”
- `NS`: 特定域的域名服务器记录。用来回答问题，“这个主机或域的名称服务器该回答什么?”
- `MX`: 邮件交换记录。用来回答问题“哪些计算机负责处理这个域中的邮件?”
- `TXT`: 文本记录。保存自由格式的文本信息。有时用于反垃圾邮件目的和证明域的所有权。
- `CNAME`: 规范名称记录。把它当作一个别名。用来生成声明，“域名xyz.example.com是abc.example.com的别名。”
- `SOA`: 权限开始记录。这包含有关域的信息，包括其主域名服务器和联系信息。

除此之外，还有很多DNS记录类型.

动态DNS

通常一般Internet的用户家中设备是没有静态IP地址(即专用的或不变的)的。如果他们重启调制解调器，他们的ISP可能会给他们一个不同的IP地址。而这将引起DNS的骚动，因为这种行为会导致指向其公共ip的任何DNS记录都将过期。

动态DNS（DDNS）立志于解决这个问题。简而言之，有两种机制:

1. 一种是客户机告诉DDNS服务器其IP地址.
2. 一种是DDNS服务器会对那条记录设置一个非常小的TTL。

虽然DNS定义了一种发送更新记录的方式，但另一种常见的方式是让局域网上的计算机定期(例如每10分钟)通过采取身份验证的HTTP请求来联系DDNS提供商。当DDNS服务器将看到发送来的IP地址，使用它来更新

记录。

反向DNS

如果你有一个点和数字的IP地址，但是需要查找的是该IP的主机名，这时该怎么办?你可以进行 *反向DNS* 查找。请注意，并非所有IP地址都有这样的记录，并且反向DNS查询通常会出现空结果。

思考题

- 现在你电脑的域名服务器是什么?你可以在网上搜索如何根据你的操作系统查找到该信息。
- 根域名服务器是不是可以知道世界上的每一个IP地址?
- 动态DNS的作用是什么?
- TTL的作用是什么?

网络地址转换（NAT）

在本节中，我们将学习 *网络地址转换*，或者称之为NAT。这是一种在路由器上提供的服务，它将“私有”的局域网地址和世界其他地方隐藏起来。

这里路由器充当中间人，将内部IP地址转换为单个外部IP地址。而在这个服务中保存了这些连接的映射表，因此当数据包到达任何一个接口时，路由器可以适当地重写它们。

我们说NAT路由器后面的局域网是一个“私有网络”，而路由器为该局域网提供的外部IP地址是“公共IP”。

传统邮局类比

想象下，一个匿名的信件系统是如何工作的。你写一封信寄给收信人，上面写着你的回信地址。

你把信交给匿名者，而不是直接寄出去。匿名者将你的信从信封中取出，并将其放入另一个信封中，该信封的收信人相同，但将寄信人的地址替换为匿名者的地址。它还记录了这个寄件人给这个收件人写了一封信。

当收件人收到信时，它所看到的全是信上面的匿名人的地址。然后，收件人回复信件，并将回复寄回给匿名人士，同时收件人将自己列为回信地址。

匿名者收到邮件并注明是收件人发来的。接着，它在其记录中查找，以确定最初向收件人寄出信件的人。匿名者将回复从信封中取出，并将其放入新信封中，目的地址为原始发件人，返回地址保留为原始收件人。

最后，原始的寄信人收到了回复。

在这个过程中，原始的收件人永远不会知道原始发送人的地址，他们互相之间只能知道匿名者的地址。

为什么要这么做?

NAT在网络中无处不在，在每一个IPv4局域网上几乎无处不在，不过，为什么要有这样一个机制?

使用这种服务，主要有两个原因:

1. 你不想给外部暴露出自己的网络细节。
2. 你需要更多的IP地址，因为别人可分配给你的IP地址已经不够，或者你没有资金去购买更多的IP地址了。

隐藏你的网络

通过使用运行NAT的网关将随机的、未经请求的数据包发送到专用网络上不是一件容易的事情。

> 你可以通过一种叫做 *源地址路由* 的技术，不过它通常不会被ISP启用。

你必须在数据包上包含私有IP地址，**并且**还能将该数据包发送到路由器。除非你的设备和路由器在同一个局域网，不然没有办法做到这一点，而这这种情况几乎不可能。因为上面的第二点，我们需要了解一种被称为 *地址空间耗尽* 的现象。

IPv4地址耗尽

世界上IPv4地址的数量是有限的。因此，获得大量的IPv4地址需要花费很多钱。如果先只获得少数地址(例如 `/20` 或`/24`或 `/4`子网)，然后通过NAT再转换出后面大量私有网络花费要小的多。通过这种方式，你可以拥有许多IP，但它们都以来自单个公共IP的形式呈现给外部的互联网世界。

这就是NAT技术解决的最实际的问题。曾经有一段时间，当我们真的要用完IPv4地址时，NAT技术拯救了我们。

私有网络

有些子网是保留作为私有网络使用的，路由器不会直接转发这些地址。如果大互联网上的路由器看到来自这些子网之一的IP，将会直接丢弃它。

对于IPv4，这些私有网络地址有如下几个子网段:

- `10.0.0.0/8`
- `172.16.0.0/12`
- `192.168.0.0/16`

最后一种在家庭局域网中非常常见。同样，因特网上的路由器会直接丢弃带有上述地址的数据包。数据从这些ip进入互联网的唯一途径是通过NAT。

它如何运作

在后面的演示中，我们将使用`IP:port`表示法。冒号后的数字为端口号。此外，我们将使用术语“本地计算机”来指代局域网上的计算机，而使用“远程计算机”或“远程服务器”来指代它所连接的远程计算机。

最后，本地局域网中的路由器将被称为“路由器”或“NAT路由器”。让我们开始吧！

在我的局域网上，我想从我的私有网段`192.168.1.2:1234`发送数据到公网地址 `203.0.113.24:80`--这里是一个`80`端口，意味着这是一个HTTP服务器。

我的计算机做第一件事就是检查目的地IP地址是否位于我的局域网上。。。而我的局域网是位于`192.168.0.0/16`子网中，很明显，目的地地址不在此子网中。于是，我的电脑会把数据包发送给默认网关，那里有我的带有NAT功能的路由器。

在前面的类比示例中，路由器将扮演“匿名者”中间人的角色。回想一下，路由器上有两个接口——一个面向内部的`192.168.0.0/16`私有局域网，另一个面向更大的互联网，拥有一个公共的外部IP。让我们使用`192.168.1.1`作为路由器上的私有IP地址，`198.51.100.99`作为公网IP。

所以一个局域网看起来会像如下这样:

```
+----------+-------------+
|  Local   |             |
| Computer | 192.168.1.2 |>-------+
+----------+-------------+        |
                                   |
                                   ^
                            +---------------+
                            |  192.168.1.1  |  Internal IP
                            +---------------+
                            |    Router     |  NAT-enabled
```

```
                              +---------------+
                              | 198.51.100.99 |  External IP
                              +---------------+
                                      v
                                      |
                                      |
                           {The Greater Internet}
```

路由器现在必须为更大的互联网“重新打包”数据。这意味着它需要将数据包中的源地址改写为来自路由器的公网IP和路由器公网接口上一些未被使用的端口。(端口不需要与数据包上最初的端口相同。路由器在发送新数据包时随机分配一个未使用的端口。)

所以路由器会记录如下所有信息:

计算机	IP	端口	协议
本地私有地址	192.168.1.2	1234	TCP
远端公网地址	203.0.113.24	80	TCP
路由器公网地址	198.51.100.99	5678	TCP

(注意: 除了`80`端口，因为我们在这个例子中是连接到HTTP，但是所有的端口都是随机被选择的。)

所以当原始数据包是如下所示时:

```
192.168.1.2:1234 --> 203.0.113.24:80

     Local                Remote
    Computer            Computer
```

NAT路由器保持目的地地址不变，但使用路由器作为源地址。

```
198.51.100.99:5678 --> 203.0.113.24:80

    NAT router             Remote
                          Computer
```

从目的地的角度来看，它完全不知道该数据包是不是来自路由器本身，所以目的地会回复一些HTTP数据，将其发送会相应的路由器:

```
203.0.113.24:80 --> 198.51.100.99:5678

    Remote               NAT router
   Computer
```

接着，路由器会查看它的记录，上面写着，“稍等--如果我在端口5678收到数据，这意味着我需要将其转换回局域网上的私有IP地址！”因此，它会转换消息，使其不再是路由器的地址，而是发送到之前记录的私有源IP:

```
203.0.113.24:80 --> 192.168.1.2:1234

    Remote                Local
   Computer             Computer
```

然后该数据就会发送会局域网，收到了!局域网计算机认为它在与远程服务器通信，而远程服务器认为它在与路由器通信!这就是NAT的作用!

在最后一步中可能会有点混乱，数据包来自NAT路由器，但是它的实际地址却像是来自远程的一个计算机。这在局域网上是可行的，因为NAT路由器发送的IP数据包带有局域网上本地计算机的以太网地址。或者，换句话说，NAT路由器可以使用链路层寻址将数据包发送到本地计算机，而数据包来自的IP地址不必与NAT路由器的因特网IP相匹配。

NAT和IPv6

因为IPv6拥有 *很多* 的地址，所以我们在IPv6中还需要NAT么？技术上的答案是“不需要”。IPv6出现的部分原因是为了摆脱这种讨厌的NAT中间人功能。也就是说，在IPv6子网中，有一个为私有网络保留的网段:

```
fd00::/8
```

在这种子网中，地址的“host”部分有一些额外的结构，但如果你愿意，你可以在维基百科上读到。像IPv4私有子网一样，这个子网的IP地址会被大互联网上的路由器丢弃。

NAT对IPv6不起作用，但有另一种方法可以用网络前缀转换进行转换。但是，人们不能轻易地将未经请求的数据包发送到你的局域网，这个想法又如何呢?好吧，其实你只需要正确地配置你的防火墙来防止人们进入。但那是另一种方法了。

端口转发

如果在NAT路由器后面运行一个服务器程序，它如何向外部世界提供相应的服务呢？毕竟，没有公网上的设备可以通过内网地址联系到他--他们只能知道路由器的公网IP地址。但是你可以配置NAT路由器来做一些叫做 *端口转发* 的事情。

例如，你可以告诉它，发送到其公共IP端口80的流量应该转发到端口80上的某个私有IP。然后路由器会按照这个规则转发流量，而原始发送方并不知道它的流量最终到达一个私有的IP。在这里，不需要一定要使用相同的端口。

例如，SSH使用22号端口，这在私有网络上的计算机上是没问题的。但是如果你从公网端口22转发，你会发现很多恶意行为者不断试图通过它登录。(是的，即使是在家里的电脑上也会发生。)因此，更常见的做法是使用其他一些不常见的端口作为公共SSH端口，然后将其转发到局域网上的22号端口上。

思考题

- 查看下你的计算机的内网IP地址。（你可能需要搜索下资料如何搜到它。）然后打开google.com，并且输入“我的ip地址是什么？”，你看到的数字将大不相同，为什么？
- NAT是为了解决什么问题？

动态主机配置协议(DHCP)

当你第一次在咖啡店打开笔记本电脑时，它没有IP地址。它甚至不知道它 *应该* 有什么IP地址。或者它的域名服务器是什么。或者它的子网掩码是多少。当然，你可以手工配置它！只要输入收银员递给你的咖啡上的数字就行了!

这肯定不会发生，没人会在意。他们会用一个重复的地址。事情就行不通了。大部分很可能会狂饮咖啡，然后再也不会光顾同一家店。如果有一种方法可以自动配置刚接入网络的计算机，那就更好了，不是吗?

这就是 *动态主机配置协议* (DHCP) 的作用。

具体过程

概述:

```
Client --> [DHCPDISCOVER packet] --> Server

Client <-- [DHCPOFFER packet] <-- Server

Client --> [DHCPREQUEST packet] --> Server

Client <-- [DHCPACK packet] <-- Server
```

其中的细节是:

当你的笔记本电脑第一次尝试连接到网络时，它通过UDP发送一个`DHCPDISCOVER`包到广播地址(`255.255.255.255`)的`67`号端口，也就是DHCP服务器端口。

回想一下，广播地址只在局域网内传播——默认网关并不会转发它。

在局域网上，有另外的一个机器作为DHCP服务器，在这个服务器上有一个进程一直等待在`67`号端口上。

DHCP服务器进程看到DISCOVER消息后，会决定如何处理它。

典型的用法是客户机需要一个IP地址，我们称之为从DHCP服务器租用的IP地址。DHCP服务会记录池中哪些IP已经分配，哪些是空闲的。

作为对`DHCPDISCOVER`报文的响应，DHCP服务器在端口`68`上向客户端发送`DHCPOFFER`响应消息。 In response to the `DHCPDISCOVER` packet, the DHCP server sends a `DHCPOFFER` response back to the client on port `68`.

offer消息中包含一个IP地址和很多潜在的需要的信息，包括但是不限于:

- 子网掩码
- 默认网关地址
- 松弛时间
- DNS服务器地址

客户端可以选择接受或忽略该offer消息。(可能有多个DHCP服务器发出offer消息，但客户端可能只接受其中一个。)

如果offer消息被接收了，客户端会发送回一个`DHCPREQUEST`消息到服务器端来通知对方，自己想要那个特定的IP地址。

最后，如果一切都成功的话，服务器会回复一个确认数据包，`DHCPACK`。

此时，客户端拥有参与到网络交流所需的一切信息。

思考题

- 与手动配置LAN上的设备相比，使用DHCP之类的机制有什么优势?
- DHCP客户端会从DHCP服务器接收到哪些类型的信息?

项目: 深度探索DNS

`dig` 是一个非常有用的命令行工具，它用于从默认名称服务器或任何名称服务器获取DNS信息。

安装

在Mac系统中，运行如下命令安装:

```
brew install bind
```

在WSL中，运行如下命令安装:

```
sudo apt install dnsutils
```

试一试

输入:

```
dig example.com
```

来看看它的返回都有些什么。

```
; <<>> DiG 9.10.6 <<>> example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60465
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
;; QUESTION SECTION:
;example.com.                   IN      A

;; ANSWER SECTION:
example.com.            79753   IN      A       93.184.216.34

;; Query time: 23 msec
;; SERVER: 1.1.1.1#53(1.1.1.1)
;; WHEN: Fri Nov 18 18:37:13 PST 2022
;; MSG SIZE  rcvd: 56
```

这真是有很多内容，不过现在让我们先只关注以下的一些行:

```
;; ANSWER SECTION:
example.com.            79753   IN      A       93.184.216.34
```

这是`example.com`的IP地址!注意到`A`了吗?这意味着这是一个IP地址记录。你还可以获得其他记录类型。如果你想查询`oregonstate.edu`使用邮件交换服务器呢?你可以把它放在命令行中:

```
dig mx oregonstate.edu
```

然后我们可以得到:

```
;; ANSWER SECTION:
oregonstate.edu. 600 IN MX 5 oregonstate-edu.mail.protection.outlook.com.
```

或者我们想要得到`example.com`的域名服务器，我们可以运行:

```
dig ns example.com
```

超市时间（TTL）

如果你用`dig`去查询一个`A`记录，你可以看到如下一行数字:

```
;; ANSWER SECTION:
example.com.            78236   IN      A       93.184.216.34
```

在这个例子中，是`78236`。这是缓存中条目的TTL。这告诉你，你使用的域名服务器已经缓存了该IP地址，并且该缓存条目直到再过78，236秒才会过期(作为参考，一天有86400秒。)

主域名服务器

如果你得到的是一个缓存在你的域名服务器上的条目，你会在`dig`命令的输出中看到`AUTHORITY: 0`:

```
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
                                          ^^^^^^^^^^^^
```

如果你看到`dig`命令的输出中的`AUTHORITY: 1`（或者其他正数），说明该条目直接来源于负责该域的域名服务器。

获取根域名服务器

只要输入`dig`并按下回车键，你就会看到所有的根DNS服务器信息，其中可能包括它们的IPv4地址(记录类型为`A`)和IPv6地址(记录类型为`AAAA`)。

挖掘一个指定的域名服务器

如果你知道要查询的名称服务器名称，则可以使用“`@`符号直接指定该服务器。两个最流行的免费域名服务器是`1.1.1.1` 和 `8.8.8.8`.

让我们向他们中的一个询问`example.com`的IP地址。

```
dig @8.8.8.8 example.com
```

然后，我们得到了期许的答案。（尽管TTL可能是不同的——毕竟，这些是不同的服务器，它们的缓存条目具有不同的时限。）

挖掘一个根域名服务器

互联网上有很多根域名服务器，让我们尝试对着`example.com`用dig命令去查询其中的一个:

```
dig @l.root-servers.net example.com
```

我们可以得到一些有意思的结果:

```
com.                    172800  IN      NS      a.gtld-servers.net.
com.                    172800  IN      NS      b.gtld-servers.net.
com.                    172800  IN      NS      c.gtld-servers.net.
com.                    172800  IN      NS      d.gtld-servers.net.
```

这些都不是 `example.com`。。。并且你看--他们都是 `NS`记录，意味着都是域名服务器。这就是根域名服务器要告诉我的: "我不知道谁是`example.com`，不过我知道有一些域名服务器知道 `.com`是什么。“

所以我们可以从中选择一个，并且继续`dig`:

```
dig @c.gtld-servers.net example.com
```

然后我们会得到:

```
example.com.            172800  IN      NS      a.iana-servers.net.
example.com.            172800  IN      NS      b.iana-servers.net.
```

同样的情况，会出现更多的`NS`域名服务器，这是`c.gtld-servers.net`域名服务器告诉我们的，我不知道`example.com`的IP地址，不过这里有一些可能知道的域名服务器！所以我们可以继续尝试:

```
dig @a.iana-servers.net example.com
```

终于，我们在 `A` 记录中获得了相应的IP！

```
example.com.            86400   IN      A       93.184.216.34
```

你也可以在命令行上使用`+trace`来从头到尾观察整个查询过程:

```
dig +trace example.com
```

我们还可以尝试的

试着去寻找以下几个问题的答案:

- `microsoft.com`的IP地址是什么?
- `google.com`的邮件服务器是什么?
- `duckduckgo.com`的域名服务器是什么?
- 按照上一节中从根名称服务器挖掘的过程，从根名称服务器开始，向下挖掘到`www.yahoo.com`(**不是** `yahoo.com`)整个过程。请注意，这整个过程应该在以 `CNAME`记录结束！你需要不断重复从根服务器开始的`CNAME` 记录。在你的文档中记录下使用`dig`命令获取到IP地址的整个过程。每个`dig`命令都应该以 `@`指明从根域名服务器开始的不同的域名服务器。

端口扫描

合法性的说明: 对一台不属于自己的电脑进行端口扫描是否是合法行为是未知的，所以我们将在`localhost`上进行所有的端口扫描。**未经许可，请勿对计算机进行端口扫描!**

端口扫描是一种确定计算机上哪些端口已准备好接受连接的方法。换句话说，可以知道哪些端口有服务程序在监听。端口扫描通常是攻击系统的先锋方法，在尝试通过连接服务器来查看是否有任何漏洞之前，我们需要通过端口扫描知道服务器上有哪些正在运行的程序。

端口扫描的机制

一个端口扫描程序会尝试去寻找指定IP上的正在监听的端口范围。（有时候，该范围是全部的端口号。）

对于TCP，端口扫描程序通常会尝试在候选端口上使用`connect()`调用来建立连接，如果该方法成功，那么我们将会得到一个开放的端口，此时端口扫描程序可以输出该信息。由于没有任何东西通过连接发送，该端口会立即关闭。

> 端口扫描程序的作用是扫描寻找开放的端口，而不是发送或者接收其他数据。

调用`connect()`会促使TCP连接完成三次握手，这并不是一定需要的，因为端口扫描器程序只要能从服务器得到任何回复(即三次握手的第二步)，那么我们就知道该端口是打开的。如果完成握手将导致远程操作系统唤醒该端口上的服务器，接着会进行我们知道无论如何都要发生的连接的关闭。

一些TCP端口扫描器程序将构建一个TCP SYN包来启动握手并将其发送到对端端口，如果它们得到一个SYN-ACK应答，它们就知道端口是打开的。但是，接下来，它们不是用一个ACK完成握手，而是发送一个RST(重置)，导致远程操作系统完全终止新创建的连接——并使得它不会唤醒该端口上的服务器程序。

> 你可以使用原始套接字 来编写定制化的TCP数据包。通常你需要管理员权限来使用它们。

UDP上的端口扫描

由于没有与UDP的连接，所以基于UDP的端口扫描一般不太精确。

有一种方法是向一个端口发送一个UDP数据包，看看是否收到一个ICMP“目的地不可达”消息。如果是，则对应的端口未打开。

如果你没有收到回复，*有可能* 是端口没有打开，也有可能是UDP包被过滤掉了，没有ICMP响应。

另一种方法是将UDP数据包发送到可能有服务器程序的已知端口。如果从服务器得到响应，就知道端口是打开的。如果没有，它是关闭的，或者你的流量被过滤掉了。

思考题

- 为什么你不能对着因特网上任一计算机进行端口扫描?
- 端口扫描程序的目的是什么?

防火墙

每一部糟糕的黑客电影中最受欢迎的一个部分就是 *防火墙* 。从那些糟糕的电影中可以清楚地了解到，它与安全有关，但确切的作用确是模糊的，很明显，一个坏人成功的通过防火墙是一件坏事。

但是让我们回到现实中来: 防火墙是一台计算机(通常是路由器)，它的主要目的是限制一个接口和另一个接口之间的某些特定类型的流量。因此，相较于普通路由器从每个端口转发各种类型的每个数据包，它可能会决定丢弃一些数据包。这样做的结果是，如果有人试图连接到防火墙远端的端口，防火墙可能会阻止该连接，但这个最终取决于防火墙的配置方式。

在本章中，我们主要是从概念上介绍防火墙，而不是详细讨论实际配置。防火墙有很多种实现方式，它们都倾向于使用不同的配置方法。

防火墙操作

如果你站在一个到达路由器的数据包角度来思考，路由器有能力检查它想要的所有数据包。它可以查看源ip和目标ip，或者端口，甚至是应用层数据。

这样它才有机会决定是否继续转发基于任何数据的数据包。

例如，可以将防火墙配置为允许任何端口80 (HTTP)流量从防火墙外部进入防火墙内部，但阻止所有其他目的端口上的传入的流量。或者它还可以被配置为只允许某些IP地址连接。

因此，当防火墙在收到数据包时，会首先查看其配置的规则，再决定是否丢弃数据包(如果它位于被过滤掉的规则中)，抑或是正常转发该数据包。

当它决定丢弃数据包时，有很多选择供其实现，它可以只是静默地丢弃它，也可以用ICMP发送“目的地不可达”的应答消息。

这个基本的过滤机制提供了一些控制，但仍然有一些你可能想做但是它无法管理的事情。

例如，如果防火墙后面的某个人通过因特网从一个随机端口建立了一个TCP连接，而我们希望能够从该随机端口将流量返回到发送主机。如果只使用纯过滤，我们只能是允许所有这些端口上的所有流量。

但是通过 *状态过滤* ，防火墙可以跟踪从防火墙后面发起的TCP连接的状态。防火墙看到TCP三次握手，就知道存在一个有效的连接。然后，它可以安全地允许流量返回到内部计算机的随机端口号。

有状态过滤也可以用于UDP流量，即使它是无连接的。对于这种流量，防火墙看到UDP数据包发出，它将允许传入的UDP数据包到该端口。(这里会先允许数据通过一段时间，由于它是无连接的，因此无法判断服务器和客户端何时结束,所以防火墙在设置超时UDP规则后会自动终止流量通过。)

> 为了防止防火墙在任何规则上超时，程序可以发送一种被称之为 *keepalive* 的消息。TCP 实际上为此有一个特定的消息类型，操作系统将定期发送空的`ACK`报文，以防止任何人超时。 (如果你使用套接字编程，一般来说，你需要设置 `SO_KEEPALIVE`选项。)
>
> keepalive也可以通过使用UDP的程序实现。可以将它配置为每隔一段时间向接收方(接收方将 以某种ACK进行回复)发送一个自定义的keepalive包。
>
> Keepalive只会在长时间没有网络流量的程序中会出现问题。

最后，过滤也可以在应用层完成。例如，如果防火墙对数据包进行足够深入的挖掘，它可能会发现它是HTTP或FTP数据。有了这些知识，它可以允许或拒绝所有HTTP流量，即使它到达的是一个非标准端口。

防火墙和NAT

在家庭网络中，防火墙同时充当路由器和交换机并进行NAT转换的功能是很常见的，在这里，这些都是一体的。

但是没有什么能阻止我们在网络上拥有一台单独的计算机做防火墙，它的作用仅仅是判断流量是否允许通过。并且也并没有规定防火墙的一边必须是私有网络，另一边必须是公网。双方都可以是私有网络或公网，或者一个私有和一个公网。

防火墙的核心作用并不是分隔公网和私有网络——这是NAT的作用——而是控制哪些流量可以通过防火墙。

本地防火墙

你也可以直接在你的计算机上设置防火墙(而通常这是一个常见的行为)。通常设置所有的连接都可以从本地计算机发出，但限制从其他计算机进入的连接。

在MacOS和Windows中，防火墙可以简单地打开，并且它会根据一些常见规则阻止流量。如果你需要某些特定的端口不被阻塞，则必须手动添加这些端口。

Linux通过一种叫做iptables的机制支持防火墙。它不是最直接的配置方案，但它确实足够强大，可以从头构建NAT路由器/防火墙。

思考题

- 防火墙和路由器的不同点和联系是什么?
- 防火墙和NAT的不同点和联系是什么?
- 海军罪案调查处中最有趣或最痛苦的事情是什么?

信任用户数据

> "你永远不知道谁是值得相信的。。。“
>
> --Gregor Ivanovich, *Sneakers*

当你的服务器从因特网上的某人那里收到数据时，要首先天然的相信他们发送的数据不是善意的。

因特网上有很多不怀好意的人，你必须采取措施防止它们发送的数据使你的服务器进程崩溃，或者更糟的是，让它们访问你的服务器机器数据本身。在本章中，我们将对用户试图发送恶意数据可能产生的一些问题进行一些概念上的学习。

总体来说，有两个大的原则:

- *像恶棍一样思考*。在什么情况下会有人向你发送恶意代码让你的程序崩溃或者产生意想不到的结果?
- 不要相信从远端来的 **任何东西**。 不要相信它会是一个合理的长度。不要相信它会包含合法的数据。

缓冲区溢出/超限

这主要会影响一些内存不安全的语言比如C或者C++。

基本思路是:

1. 你的程序申请了一个固定长度的内存以供程序使用。
2. 你的程序通过网络连接将一些数据读入该内存区域。
3. 攻击者发送的数据超过了该内存区域的容量，并且数据经过精心构造以包含了有效负载。
4. 你的程序单纯地写入攻击者的数据，填充自己的内存区域，然后溢出到内存中的任何地方。
5. 根据攻击者代码的编写方式，攻击者可以覆盖堆栈上的返回地址值，从而导致函数返回到攻击者的攻击代码并运行它。攻击代码可能从系统安装病毒或更改系统以允许远程访问。

现代操作系统试图通过使内存的堆栈和堆区域不可执行，以及内存的代码区域不可写来减轻这种影响。作为一名C语言编开发人员，在代码中你需要正确执行边界检查，并且永远不会写入非允许的内存。

注入攻击

在这些攻击中，你使用用户提供给你的一些带有数据的命令。然后直接运行这个命令。一些恶意用户会向你提供引导运行另一个命令的数据。

系统命令

在许多语言中，都有一个允许你通过shell运行命令的功能。例如，在Python中，你可以运行`ls`命令来获得如下的目录列表:

```
import os
```

```
os.system("ls")
```

假设你编写了一个接用户发送的数据的服务器程序。用户按照自己想要的选择的功能，将会选择发送`1`, `2`或`3`作为数据。你在你的服务器上运行类似下面的代码:

```
os.system("mycommand " + user_input)
```

所以，如果用户发送`2`，它将按预期运行`mycommand 2`并且将输出返回给用户。

为了安全起见，`mycommand`程序会验证唯一允许的输入是`1`, `2` 或 `3` ，如果对方发送了其他内容，则返回错误。

那么我们安全了吗？并没有，你猜怎么着？用户可以传入类似如下输入:

```
1; cat /etc/passwd
```

这将会导致服务器端执行如下内容:

```
mycommand 1; cat /etc/passwd
```

分号是bash中的命令分隔符。这将导致`mycommand`程序执行，然后会继续执行显示Unix密码文件内容的命令。为了安全起见，输入中的所有特殊字符都需要被剥离或进行转义，这样shell就不会解释它们。

SQL 命令

还有一种类似的攻击称为SQL注入，在这种攻击中，没有任何防范的SQL查询检查可以允许恶意用户执行任意SQL查询。假设你在Python中构建一个这样的SQL查询，我们从网络上通过`username`获取信息:

```
q = f"SELECT * FROM users WHERE name = '{username}'
```

所以，当我输入`Alice` 作为用户名时，我们可以有如下合法的查询语句:

```
SELECT * FROM users WHERE name = 'Alice'
```

接着，我会运行它，没有任何问题。不过，让我们 *像恶棍一样思考下*。如果我们输入这些会怎样:

```
Alice' or 1=1 --
```

现在我们会得到:

```
SELECT * FROM users WHERE name = 'Alice' or 1=1 -- '
```

`--` 是SQL中的注释符号，现在我们已经组合了一个查询，它会显示所有用户信息，而不仅仅是Alice的信息。

不仅如此，一些数据库的实现还可能支持`;`命令分隔符。如果是这样，攻击者可以这样做:

```
Alice'; SELECT * FROM passwords --
```

现在我们得到了如下的命令:

```
SELECT * FROM users WHERE name = 'Alice'; SELECT * FROM passwords -- '
```

现在我们从输出中得到了`passwords`表所有信息。

要避免这种陷阱，需要使用 *参数化查询生成器* 。它是SQL库中的一部分，它可以使得任何用户输入的查询都足够安全。 永远不要尝试自己构建SQL字符串。

跨站脚本

这是一个会在HTML/JS中发生的情况。

假设你有一个接受用户评论的表单，然后服务器将该评论附加到web页面。所以当用户输入:

```
This site has significant problems. I feel uncomfortable using it.

Love, FriendlyTroll
```

不过它在最后附加了如下一段代码:

```
LOL
<script>alert("Pwnd!")</script>
```

现在，每个人在查看评论时都会看到这个警告框!

这是一个无伤大雅的例子,这里的JavaScript代码还可以是其他任何东西，并可以在远程站点上执行(这意味着它可以作为该域执行API调用和获取)。它还可以重写页面，以便登录/密码输入将该信息提交给攻击者的网站。

> "It would be bad."
>
> Egon Spengler, *Ghostbusters*

大多数面向html的库都提供了一个函数来对字符串进行转义，这样浏览器就直接显示它们而不是解释它们(例如，将所有的`<`替换为`>`等等)。

在浏览器上显示任何用户生成的数据之前，一定要先通过这样的函数来运行它。

思考题

- 总体来说，无条件信任用户输入的问题是什么?
- 为什么缓冲区溢出在Python、Go或Rust等语言中不像在C语言中那样严重?

项目：端口扫描

在本项目中，我们准备做一个端口扫描的项目！

注意:我们只会在`localhost`上运行这个程序，以避免任何法律上的麻烦

为了使该项目能够运行，我们需要安装`nmap`。

MacOS:

```
brew install nmap
```

Windows WSL:

```
sudo apt-get update
sudo apt-get install nmap
```

1. **扫描所有常用端口** 该命令会扫描1000个最常用的端口:

   ```
   nmap localhost
   ```

   该程序，输出会是什么?

2. **扫描所有端口** 该命令会从`0`开始扫描所有端口:

   ```
   nmap -p0- localhost
   ```

   它的输出会是什么?

3. **运行一个服务器和端口扫描** 在某个端口上运行前面学习的任何TCP服务器程序，并再次运行前面的端口扫描程序。

   - 注意查看你的服务器端口是否在输出中！
   - 你的服务器是否因“连接重置”错误而崩溃?如果是，为什么?如果没看到，那么即使你没有从服务器上看到它，也思考下为什么会发生这种情况。(请参阅[端口扫描章节](# Port - Scanning) !)

项目: 多客户端单服务器聊天程序

现在，让我们把之前学习的所有东西都用到一起，完成这个终极项目！

我们将实现一个多用户聊天服务器和与之配套的聊天客户端程序。

在本项目中，聊天服务器程序应该允许任意数量的来自客户机的连接，每个客户端都可以看到其他客户端在说什么。不仅如此，当用户加入或离开聊天时，还会有对应的消息。

下面是一个示例截图。输入提示符后室用户(在本例中为“pat”)他将要说的话，在其之上的区域是所有之前的聊天记录。

```
*** pat has joined the chat
pat: Hello?
*** leslie has joined the chat
leslie: hi everyone!
pat: hows it going
*** chris has joined the chat
chris: OK, now we can start!
pat: lol
leslie: Why are you always last?
*** chris has left the chat

pat> oh no!! :)
```

总体架构

我们的程序中只有一个服务器，它可以同时处理许多客户端连接。

服务器端程序

服务器端程序将使用 `select()`调用来处理多个连接，以查看哪些连接已准备好，可以从中读取信息。

侦听套接字本身也将包含在此集合中，当它显示“ready-to-read”时，这意味着有一个新的连接可以被 `accept()`。而如果任何其他已经accept的套接字显示“ready-to-read”，这意味着客户端已经发送了一些需要被处理的数据。

当服务器程序从一个客户机获得聊天数据时，它会将该聊天消息重新广播给每个连接的客户端程序。

> 注意：当我们在这里使用“广播”术语的时候，指的是我们大众意义上的把一个消息告知很多 人。而**不是**指的是IP或者因特网的广播地址。在本项目中，我们不会用到这个概念。

当一个新客户端连接或者断开时，服务器端程序也会广播该消息给所有客户端。因为很多客户端会发送消息给服务器端，服务器需要为 *每一个客户端* 维护一个数据包缓存。

> 你可以使用Python中的`dict`，用客户端的套接字本身作为键，缓冲区作为值，这样就得到了套接字到缓冲区的映射。

服务器端程序通过在命令行指定一个端口来运行，在本项目中，这是一个强制要求的选项，在这里没有默认端口。

```
python chat_server.py 3490
```

客户端程序

客户端启动时，用户在命令行中指定自己的昵称(AKA“nick”)以及想要连接的服务器信息。

客户端发送的第一个数据包是一个包含昵称的“hello”数据包。(这就是服务器如何将连接与nick关联起来，并将连接事件重新广播给其他客户端的方法。)

在此之后，用户在客户端程序中键入的每一行都将作为聊天数据发送给服务器。客户端程序从服务器获得的每个聊天数据包(或连接或断开连接数据包)都会显示在其屏幕输出上。

客户端程序有一个 **文本用户界面**（TUI）用来保持输出界面整洁。由于输出是在屏幕的不同部分异步发生的，而不是在输入部分，因此我们需要执行一些终端显示技巧来防止它们相互覆盖。该TUI代码将在下一节中提供并描述。

由于在用户输入内容时可能会有数据到达，因此我们需要一种方法来处理这种情况。基于此，客户端程序将是**多线程**的，会有两个运行的线程。

- 主线程主要完成以下工作:
  - 读取键盘输入
  - 将聊天消息从客户端发送给服务器
- 接收线程要要做的是:
  - 从服务器端接收数据包
  - 在屏幕上显示结果

由于这些线程之间没有共享数据的需求，因此不需要同步(互斥锁等)。

> 是的，他们会共享套接字，不过操作系统会保证多个线程在同一时间使用同一个套接字不会产生 资源竞争。它是一个 *线程安全* 的资源。

如果你需要学习更多关于Python多线程的知识，请参见Appendix:Threading。客户端程序将通过在命令行上指定用户的昵称、服务器端地址和服务器端口来启动。这些都是必需的参数，在客户端程序中没有对应的默认值。

```
python chat_client.py chris localhost 3490
```

客户端I/O

客户端屏幕有两个主要区域：

- 输入区，屏幕最下方的一两行。
- 输出区，屏幕上方的其他部分。

(下面的客户端TUI部分详细介绍了如何实现这些I/O的具体逻辑。)

屏幕底部的客户端输入行是用户的昵称，后面跟着 `>` 和一个空格。在此之后输入内容:

```
alice> this is some sample input
```

输出区有两种主要消息:

- **聊天消息**: 显示说话人的昵称，后面跟着":"和一个空格，然后是用户的具体聊天内容。 The output area of the screen has two main types of messages:

  ```
  pat: hows it going
  ```

- **信息消息**: 这些消息用来显示用户何时加入或离开聊天，或需要打印的任何其他系统信息。它们首先是 `***` 和空格，然后是具体的消息。加入和离开的消息模版如下:

  ```
  *** leslie has joined the chat
  *** chris has left the chat
  ```

用户特殊输入

如果用户输入以`/`开头，则表示该消息具有特殊含义，应进一步解析以确定具体的内容。目前，唯一的特殊输入被定义为`/q`:

- `/q`: 当用户输入这个消息时，客户端会退出。在这种情况下，没有任何内容会被发送到服务器。

客户端TUI

[fls[可以从这里下载聊天UI的代码和演示|chat/chatui.zip]]。

在文件 `chatui.py`中，有四个你需要的函数，你可以通过下面的代码导入它们:

这些函数的功能是:

- **init_windows()**: 在实现其他面向UI的I/O代码之前需要先调用这个，它还应该在启动接收线程之前调用，因为该线程执行I/O操作。

- **end_windows()**: 当你的程序结束之后，可以调用此函数清除一切。

- **read_command()**: 本函数将在屏幕底部打印提示并接受用户输入。它返回用户按下`ENTER`键后输入的行。 举个例子:

  ```
  s = read_command("Enter something> ")
  ```

  该函数负责正确显示元素在屏幕上的位置。

- **print_message()**: 将消息打印到屏幕的输出部分。此函数会处理滚动并确保输出不会干扰到从`read_command()`读取的输入。无需在输出中添加换行符，它会自动的被添加上。

已知问题: 在Mac上，如果有消息通过`print_message()`打印出来之后，接下来再按下删除键会显示一个`^R`并且会向下滚动一行。暂时还不知道是什么原因。

`chatui`的Curses变种

如果`chatui` 库使用的不那么顺手，你可以尝试另一个增强版本`chatuicurses`。它具有完全相同的功能，并且以完全相同的方式使用。在你使用这个版本的之前，你需要安装unicurses库:

```
python3 -m pip install uni-curses
```

在安装以后，你可以import `chatuicurses`而不用import`chatui`。

Mac版本已知问题: 在我的Mac机器上会抱怨说curses库没有安装，而实际上它已经安装了。但是这在Linux或Windows没有影响。

一个限制 使用该库会导致`CRTL-C` 不能退出应用程序。因此，你可能需要按下 `CRTL-C`并且后跟`RETURN`才能真正退出。在Windows上，你可以尝试`CTRL-BREAK`。

数据包结构

客户端程序和服务器程序将通过TCP(流)套接字，使用自定义的数据包结构进行通信。

简而言之，数据包是有一个表示有效负载长度的16位大端数字和字符串表示的有效负载组成，其中使用UTF-8编码的json格式数据。

> 你可以通过在字符串上调用 `.encode()`函数将JSON字符串编码为UTF-8字节。`.encode()` 函数使用一个参数来指定编码，在默认情况下，它是`"UTF-8"`。

因此，在查看数据流时，首先要做的是确保缓冲区中至少有两个字节，这样可以知道JSON数据的准确长度。然后，查看缓冲区中是否有相应长度(2字节头加上2)的数据。

JSON 负载

如果你对JSON不是很熟悉，可以查看Appendix: JSON节。

每个数据包都是由一个两字节的表示长度的负载，后跟具体负载组成。具体的负载是以UTF-8编码的的json字符串。每个有效负载都是一个Object，并且在其中有一个名为`"type"`的字段，表示有效负载的类型。其余字段根据类型而相应改变。

在下面的示例中，字符串中的方括号用于表示需要放入相关信息的位置。括号**不**包含在数据包中。

负载“Hello”

当客户端第一次连接上服务器时，它会发送一个带有用户昵称的“hello”数据包。它的目的是使得服务器可以将来自nick的客户端和自己本地维护的连接相关联。此数据包须在其他数据包之前发送。从客户端发送给服务器端的消息形式为:

```
{
    "type": "hello"
    "nick": "[user nickname]"
}
```

复杂"Chat"

该数据包是具体的聊天消息。它有两种形式，取决于聊天消息是来自客户端(即用户想要发送消息)还是服务器端(即服务器正在广播其他人的消息)。

从客户端发送给服务器端的消息为：

```
{
    "type": "chat"
    "message": "[message]"
}
```

从服务器端发送给客户端的消息为：

```
{
    "type": "chat"
    "nick": "[sender nickname]"
    "message": "[message]"
}
```

客户端不需要将发送方的用户名nick与数据包一起发送，因为服务器已经可以从之前发送的`hello`数据包获得对应的连接。

负载"Join"

当有客户端加入此聊天时，服务器端程序会发送此负载给所有客户端。

```
{
    "type": "join"
    "nick": "[joiner's nickname]"
}
```

负载"Leave"

当有客户端离开此聊天时，服务器端程序会发送此负载给所有客户端。

```
{
    "type": "leave"
```

```
    "nick": "[leaver's nickname]"
}
```

扩展

下面的内容不会被记入得分，不过，如果你想进行根更深的探索，这里有一些方向。**警告！**请确保你提交的任何内容首先得包含了上面的要求。这些扩展模块只是一个严格的超集，或者你可以fork一个新项目来保存它们。至少，我建议从你的工作版本进行分支，这样就不会意外地搞砸了！

- 添加直接消息功能——如果用户“pat”发送:

  ```
  /message chris how's it going?
  ```

  then "chris" will see:

  ```
  pat -> chris: how's it going?
  ```

  (如果用户不存在怎么办?也许你需要定义一个从服务器返回的对应的错误包消息！)

- 添加一个方法来列出聊天中所有人的名字。

- 添加表情——如果用户“pat”发送:

  ```
  /me goes out to buy some snacky cakes
  ```

  everyone else sees:

  ```
  [pat goes out to buy some snacky cakes]
  ```

  (此功能正确的方法是添加一个新的数据包类型!)

- 添加聊天室功能 -- 当每个人第一次加入时，可以有一个默认的聊天室，也可以添加聊天室，加入或离开聊天室，以及列出可用的聊天室。

- 将所有这些都转换成为一个MUD。这些足够使你忙起来了！

一些建议

这里有一些可能会有所帮助的建议。

- 让服务器程序使用它传递给`select()`的已连接套接字集作为已连接的每个客户端的标准列表。

  如果一个客户端断开，将其从该集合中删除。

如果一个客户端连接上服务器端，将其加入到该列表。

这个集合将始终反映在这个时间连接的所有人。

- 在服务器上为每个连接设置一个缓冲区。还记得我们在早期的项目中，我们学习了如何使用缓冲区来积累数据，直到得到一个完整的数据包吗? *每个连接* 都需要一个这样的缓冲区。在这个项目中，程序会有很多连接。
- 还记得我们编写代码从字节串缓冲区提取数据包的项目吗?可以复用其中的策略。
- 你需要用到`.to_bytes()`和`.from_bytes()`来获取和设置数据包长度。
- 从前面的项目中尽可能多的复用相关知识和模块。

附录：位运算{#appendix-bitwise}

在本节中，我们将会复习 *位运算操作*。

编程语言中的位运算符是指对数字的位进行运算的操作。这些运算符将数字当做是一个个用二进制，即使它不是。但其实，它们可以在Python代码中处理任何基数的数字，但对我们人类来说，二进制是最直观的。

> 高级技巧: 记住，不同的数字进制，比如十六进制，二进制和十进制，只是表示一个值的不同方式。这就像是在不同的语言中同一个数字有不同的表示方式一样。
>
> 当一个值存储在变量中时，最好的做法是把它看作是纯数字形式的存在——根本没有基数。只有当你把 它写下来(用代码或打印出来)的时候，基数才会起作用。
>
> 比如，Python 默认会以基数10为基础打印每一个数字。它有各种方法来覆盖它并以另一种基数输出。Protip: remember that different number bases like hex, binary, and decimal are just different ways of writing a value down. Kind of like different languages for representing the same numeric value.

我们将要学习:

- Python中不同进制
- 在Python中以不同的基打印结果
- 位与运算
- 位或运算
- 位非运算
- 移位位运算
- 将指定数字位置`1`

Python中不同进制

`110101`值是什么?

你的大脑可能会认为我在问:“这个二进制数用十进制表示是多少?”但你错了!我并没有指定这个数字的基数，所以你不知道它是二进制、十进制还是十六进制!

现在我声明一下，这个值是十进制，所以是110,1001。Python和大多数其他语言都假定代码中的数字是十进制的，除非用户自行指定。如果你想要一个不同的基数，你必须在数字前加上前缀来表示使用的是什么基数。

在Python中支持的基数有:

基数	基数名称	前缀
2	二进制	`0b`
8	八进制	`0o`
10	十进制	None
16	十六进制	`0x`

下面，让我们来书写一些不同基数的数字:

```
  110101 decimal!
0b110101 binary!
0x110101 hex!
```

让我们看一下十进制数 `3490`。我可以把它转换成十六进制，得到`0xda2`。请记住，下面的这两个数字是相等的:

```
>>> 3490 == 0xda2
True
```

这就是计算机中不同“语言”表示同一个数字。

输出和转换

请记住，变量中不存在“以十六进制存储”或“以十进制存储”的值的概念。一个变量只保存一个数值，我们不应该认为它有基数。只有在编写代码或输出时，它才会获得一个基数值，而代码编写者需要指定该基数值。(尽管Python在所有情况下都默认使用十进制。)

你可以使用`hex()`函数将值转换为十六进制字符串。所有“转换”的值最终都是字符串，也不会转换成其他类型了。

```
>>> print(hex(3490))
0xda2
```

同样，你可以使用`bin()` 函数将一个数值转换为二进制字符串。

```
>>> print(bin(3490))
0b110110100010
```

当然，你也可以使用python中的f-strings来完成这个工作。

```
>>> print(f"3490 is {3490:x} in hex and {3490:b} in binary")
3490 is da2 in hex and 110110100010 in binary
```

f-string有一个很好的特性，它可以被填充到一个带有零的字段宽度。假设你想要一个8位十六进制数字表示，你可以这样做:

```
>>> print(f"3490 is {3490:08x} in hex")
3490 is 00000da2 in hex
```

位与运算

位与运算，使用AND操作将两个数字连接在一起。如果两个输入位都为`1`，则与运算的结果为`1`。否则就是`0`。

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

我们通过一个例子来看下具体的两个数的位与运算。和许多其他语言中使用与符号操作符(`&`)一样，与运算在python中依然以这个符号表示。

```
  0     0     1     1
& 0   & 1   & 0   & 1
---   ---   ---   ---
  0     0     0     1
```

只有当两个输入位都为`1`时，结果才为`1`。

下面的这个大数通过位与运算计算每一位的结果，可以查看下面数字中的每一列计算的结果。(为了完整显示，十进制结果显示在右侧，但这是从二进制表示派生的。因为看两个十进制数并确定它们的位与运算结果并不直观。)

```
  0100011111000101010      146986
& 1001111001001111000    & 324216
----------------------   --------
  0000011001000101000       12840
```

可以仔细查看只有上面的运算中，只有两个列中的两个位都是 `1` 时，输出位的结果才是 `1` 。

位或运算

位或运算，使用OR操作将两个数字连接在一起。如果两个输入位中有一位或者两位都是`1`，则与运算的结果为`1`。否则就是`0`。

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

我们通过一个例子来看下具体的两个数的位或运算。和许多其他语言中使用与符号操作符(`|`)一样，或运算在python中依然以这个符号表示。

```
  0     0     1     1
| 0   | 1   | 0   | 1
---   ---   ---   ---
  0     1     1     1
```

可以看到，结果为`1`的位，输入位至少有一个是`1`。下面的这个大数通过位或运算计算每一位的结果，可以查看下面数字中的每一列计算的结果。(为了完整显示，十进制结果显示在右侧，但这是从二进制表示派生的。因为看两个十进制数并确定它们的位或运算结果并不直观。)

```
  0100011111000101010      146986
| 1001111001001111000    | 324216
----------------------   --------
  1101111111001111010      458362
```

可以仔细查看只有上面的运算中，当至少一个列是 `1` 时，输出位的结果才是 `1` 。

位非运算

位否运算会把所有的 `1` *翻转* 成 `0`和所有的`0` *翻转* 成`1`。 这是一个一元操作符--它只在一个数字上起作用。

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

我们通过一个例子来看下具体的一个数的位非运算。和许多其他语言中使用与符号操作符(`~`)一样，非运算在python中依然以这个符号表示。

```
~ 0   ~ 1
---   ---
  1     0
```

你可以看到，运算的结果仅仅是简单的反转了位的值。

下面的这个大数通过位非运算计算每一位的结果，可以查看下面数字中的每一列计算的结果。(为了完整显示，十进制结果显示在右侧，但这是从二进制表示派生的。因为看两个十进制数并确定它们的位非运算结果并不直观。)

```
~ 0100011111000101010    ~ 146986
----------------------   --------
  1011100000111010101      377301
```

可以看到，输出中的每一位1都是0翻转得到的。

Python提示:由于Python支持任意精度算术，直接进行非运算经常会得到负数。如果你需要保证它是正的，你需要使用位与运算，将数字最终所需位长度的全1的数进行与运算。例如，要获得' 37反转的字节，你可以执行以下任何操作:

```
(~37) & 255
(~37) & 0xff
(~37) & 0b11111111
```

(因为，当然，这些都是不同的基数表示下相同的数!)

移位运算

这是一个有趣的运算，利用此运算规则，你可以将一个数来回移动一定的位数。下面是如何将一个数的所有位左移2位的例子:

```
000111000111  left shifted by 2 is:
011100011100
```

或者我们也可以向右移:

```
000111000111  right shifted by 2 is:
000001110001
```

左边或者右边新产生的位置为0，而所有被移出的位会永远的被消除。

> 在这里，我对很多语言如何右移负数描述的不够准确。如果你将一个负数右移，新出现的位根据 语言的不同，可能是1 而不是0。Python里面的整数运算是任意精度的，所以实际上在 Python中，任何负数的左边都有无限个1 位，这让可能会让你有时候对结果感到奇怪。
>
> 所以，在目前了解原理的阶段，只考虑正数。

<<在大多数语言中是左移，而 >>在大多数语言中表示右移（Ruby除外！）。让我们来看一个具体的例子:

```
>>> v = 0b00000101
>>> print(f"{v << 2:08b}")
00010100
```

这里有一个二进制表示的数，我们用`v << 2`将该数左移了2个字节。你可以看到`101`已经向左移动了2位!

设置任意长度1位

如果我们想要得到一个连续位数为1的数，这是我们可以做的一个小技巧。举例来说，如果我们想要一个12位都为1的数，也就是:

```
0b111111111111
```

我应该怎么做？ 我们可以用一些技巧，在这里让我们从技巧1开始。如果您想将第n位设置为1(其中最右边的位是位0)，则可以将2取该次幂，这样你就能得到它。

让我们将第5位设置为1，我们可以首先计算`2**5`，得到结果`32`，而`32`用二进制表示就是`100000`，我们得到了我们想要的。

另外一种方法是将`1`左移5位，也就是`1 << 5`得到 二进制`100000`，也就是十进制的`32`。也可以工作。

但是怎么从这里得到我们需要的连续的`1`呢?

看看这个，32减1等于多少?31. 这不是一个脑筋急转弯问题。让我们看看它们的二进制:

```
32   100000
31   011111
```

嘿!这里有一连串的`1`!不仅如此，它还是5个连续的`1`，就像我们想要的那样!(这类似于十进制中的减法。10000 - 1等于9999。在二进制中，我们得到的结果都是`1`，而不是`9`。)

```
run_of_ones = (1 << count) - 1
```

下面是如何得到连续12个`1`:

```
>>> bin((1 << 12) - 1)
'0b111111111111'
```

如果你喜欢这些 *琐碎位运算的技巧*，你可能会喜欢这本书[Hacker's Delight](https://en.wikipedia.org/wiki/Hacker%27s Delight)。第2章涵盖了许多这些技术，这些技术以前都是免费发布的，你可能可以找到一个PDF文件。

思考题

- `2342 & 2332`结果是什么?
- `0b110101 | 112`结果是什么?

- `~0b101010010101`用二进制表示是什么？（在Python中，会得到一个负数，不过你的可以将其和`0b111111111111`进行与运算转化成为一个正值。别忘了Python在打印输出时会自动消除前置0！）
- `16 << 1` 结果是什么？
- `64 << 1` 结果是什么？
- `4200 << 1` 结果是什么？有找到什么规律么？
- `16 >> 2` 结果是什么？
- `0b11100111 << 3` 结果是什么？
- `(1 << 8) - 1`结果是什么？
- `0x01020304 & ((1 << 16) - 1)` 结果是什么？

附录: 安装Packet Tracer {#appendix-packettracer}

本周，我们准备使用模拟器来构造一些小型网络。这个模拟器被称为 *Packet Tracer*， 它是由Cisco提供的免费软件(只要你报名参加一个免费的课程，虽然它很有用，但你不必参加). 本项目主要目的是说明如何设置一个Cisco账户，下载Packet Tracer, 并且运行它。

下载Packet Tracer

你可以通过以下步骤直接下载它(如果你想通过Cisco在线教程来下载它，请看下面):

1. 进入Packet Tracer 下载页。
2. 点击"Log in"。
3. 创建一个账号。
4. 根据你的操作系统，点击下载链接。
5. 安装软件。

如果你想通过免费课程接口下载它，那么上面的内容就不起作用了，可以尝试如下步骤:

1. 进入Packet Tracer 下载页。
2. 点击"Getting Started with Cisco Packet Tracer"课程。
3. 建立 "Get Started"。
4. 创建一个账号。
5. 当账号创建好了之后，你应该可以根据课程链接来下载软件（或者，如果你已经登录的话，你可以尝试以下链接。

运行Packet Tracer

当成功安装之后，直接运行它!

当运行Packet Tracer之后，你应该会看到一堆圆形的物体，在它们的下面有数字。

- 将其中任意3个拖动到主工作区。(任何类型的三件物体都可以。)
- 然后点击“Place Note”图标(上面第二行，左边第五行，看起来像一个剪贴板)。然后单击工作区并输入你的命名。完成后按`ESC`键。

这个工具还有很多功能，但至少你已经知道了如何使用这个工具。

附录： 多线程 {#appendix-threading}

多线程 是一个进程能够同时运行多个 *线程* 的方法。也就是说，它可以同时运行很多个函数。

如果你的不同的函数有的需要等待某事发生，同时有的不需要等待，只需同时运行，那么这个技术就十分有用。对于一个实用的多用户聊天客户端，因为它必须同时做两件事，而这两件事互相之前会有阻塞:

- 一个是等待用户输入自己的聊天消息。
- 一个是等待服务器发送来更多消息。

如果不使用多线程，在等待用户输入时就无法从服务器接收消息，反之亦然。

> 旁注: `select()` 函数实际上也可以将常规文件描述符添加到要侦听的集合中。所以技术上 它 *可以* 同时监听套接字上的数据包 **和** 键盘来的消息。但是在Windows上不行。并且 从总体上来说，如果使用这种方法，客户端的难度要小于多线程。

让我们看下在Python中如何实现这个功能。

概念

首先，有几个术语我们应该先搞清楚。

- **线程**: “运行线程”的表示形式，即在当前特定时刻正在执行的程序的一部分。如果你想让程序的多个部分同时执行，你可以分别把它们放在单独的线程中。

- **主线程**: 这是默认情况下运行的线程，我们只是从来没给它起过名字。但是你运行的代码都是默认在主线程中运行的。

- **生成线程**: “衍生/生成”一个新线程指的是运行一个特定的函数来运行线程，当程序这样做时，线程函数将与主线程 *同时执行* 。他们会同时立即执行！

- **Join**: 一个线程可以通过调用`join()`函数来等待另一个线程退出。这在技术上，是将另一个线程和调用线程发生链接。通常主线程会调用 `join()`函数来等待它创建的线程回到自己的执行代码。

- **目标线程**: 目标线程指的是一个将要执行的运行线程的函数。当函数返回时，线程即退出。

有一个值得注意的是任何 *全局变量都会被所有线程共享*! 这意味着一个线程可以设置全局对象，而其他线程将会看到这些更改。如果共享数据是只读的，则不必担心这个问题，但如果共享数据是可写的，这就成为了一个必须要考虑的问题。

我们将在这里讨论并发和同步，因此对于这个项目，我们将不会使用任何全局共享对象。记住那句古老的谚语:

> Shared Nothing Is Happy Everybody

好吧，这并不是一句古老的谚语。而是我刚编的。现在我又读了一遍，听起来有点自私。

回到主题:局部对象不能在线程之间共享。这意味着线程可以维护自己的局部变量和参数值。每个线程都可以各自维持自己的变量，而这些更改对其他线程是不可见的。

另外，如果多个线程同时运行，它们的执行顺序是不可预测的。而当线程之间存在某种时间或数据依赖时，这就会成为一个问题，那时，再来思考这个问题。但是，我们需要意识到执行的顺序是不可预测的，只是在本项目中，我们暂时不考虑这个。

背景知识介绍的差不多了，下面可以正式开始了。

Python中的多线程

让我们写一个创建三个线程的程序。

每个线程将运行一个名为 `runner()`的函数(你可以随意调用该函数)。这个函数有两个参数:一个`name`和一个`count`。它循环并输出`name`变量`count`次。线程将在`runner()`函数返回时退出。你可以通过调用`threading.Thread()`构造函数来创建新线程。

同时，你可以通过调用它的 `.start()`方法来让线程运行。通过调用`.join()` 方法，可以等待线程退出。下面，让我们来具体看一看!

```
import threading
import time

def runner(name, count):
    """ Thread running function. """

    for i in range(count):
        print(f"Running: {name} {i}")
        time.sleep(0.2)  # seconds

# Launch this many threads
THREAD_COUNT = 3

# We need to keep track of them so that we can join() them later. We'll
# put all the thread references into this array
threads = []

# Launch all threads!!
for i in range(THREAD_COUNT):

    # Give them a name
    name = f"Thread{i}"

    # Set up the thread object. We're going to run the function called
    # "runner" and pass it two arguments: the thread's name and count:
    t = threading.Thread(target=runner, args=(name, i+3))

    # The thread won't start executing until we call `start()`:
    t.start()

    # Keep track of this thread so we can join() it later.
    threads.append(t)

# Join all the threads back up to this, the main thread. The main thread
# will block on the join() call until the thread is complete. If the
```

```
# thread is already complete, the join() returns immediately.

for t in threads:
    t.join()
```

下面是程序的输出: And here's the output:

```
Running: Thread0 0
Running: Thread1 0
Running: Thread2 0
Running: Thread1 1
Running: Thread0 1
Running: Thread2 1
Running: Thread1 2
Running: Thread0 2
Running: Thread2 2
Running: Thread1 3
Running: Thread2 3
Running: Thread2 4
```

可以看到，这些线程都是同时执行！

注意，线程的执行顺序并不一致。每一次运行顺序可能都不会一样。这对于这个程序来说没有问题，因为线程之间没有任何依赖。

守护线程

Python定义了两种不同类型的线程:

- 普通线程
- 守护线程

它的思想是守护线程将永远运行并且永远不会从其函数返回。与非守护线程不同，一旦所有的非守护线程死亡，这些线程将被Python程序自动终止。

一些关于`CTRL-C`的小知识

如果你使用`CTRL-C` 来终止主线程，并且没有其他非守护线程正在执行，所有的守护线程也会被终止。但是，但是如果你有一些非守护线程，你需要一直按`CTRL-C`，直到所有的线程被终止而看到提示符。

在最后一个项目中，我们将永远运行一个线程来监听来自服务器的传入消息。所以这应该是一个守护线程。

你可以通过如下代码来创建一个守护线程:

```
t = threading.Thread(target=runner, daemon=True)
```

这样，至少`CTRL-C`会让你方便的退出客户端。

思考题

- 描述下线程可以解决哪些问题。
- 在Python中，守护线程和非守护线程的区别是什么?
- 在Python中创建主线程需要做些什么?

多线程项目

如果你想展示一下你的编程水平，这里有一个可以参考的小项目。

我们将要完成什么

在这个项目中客户会提供一些值域范围，要求是位于所有值域范围内的数的总和。 比如，有如下这些值域:

```
[
    [1,5],
    [20,22]
]
```

我们想要得到的是:

- 首先执行`1+2+3+4+5`得到`15`。
- 接着执行`20+21+22`得到`63`。
- 最后将`15+63` 得到`78`，这就是最终我们需要的结果。

需求方想要打印出各个值域的所有值的总和。对于上面的例子，他们希望输出:

```
[15, 63]
78
```

总体架构

程序必须使用线程来解决这个问题，因为对于这个问题需求发起方非常喜欢并行程序。

你应该写一个函数把一段数字相加。然后，你将为每对值域生成一个线程，并让该线程在相应的范围上工作。如果有10个值域，则将有10个线程，每个值域用一个线程处理。

主线程将会是:

- 首先为结果分配一个数组，这个数组长度应该与值域对的数量相同(与线程的数量相同)。每个线程都有自己的空间来存储数组中的结果。

  ```
  result = [0] * n   # Create an array of `n` zeros
  ```

- 在一个循环中，运行所有线程。传入线程的参数是:

    - 线程ID是`0..(n-1)`，其中`n` 是线程数量。这也是线程用于在结果数组中存储其结果的索引值。
    - 值域的开始值。
    - 值域的结束值。
    - 存放每个线程计算结果的数组。
    - 主线程应该保存数组中从`threading.Thread()` 返回的所有线程对象，因为后续步骤还需要它们。
- 在之后的另一个循环中，对所有线程调用 `.join()` 。这可以使得主线程等待所有子线程完成。
- 打印结果，当所有的线程都`join()`之后，结果数组里将会有所有的值的和。

有用的函数

- `threading.Thread()`: 创建一个线程。
- `range(a, b)`: 生成`[a, b)` 之内的所有数的迭代器。
- `sum()`: 计算可迭代对象的和。
- `enumerate()`: 在可迭代对象上生成索引和值。

运行一个线程函数需要知道的所有事情

所有线程的结果都将写入共享数组。这个数组可以提前设置为所有元素都为零。每个线程应该都对应一个元素，以便每个线程可以正确的填入对应的结果。

要做到这一点，你必须将线程的索引号传递给它的运行函数，以便它知道将结果放入共享数组中的哪个元素!

运行示例

示例输入（你可以直接在你的程序中直接硬编码这些）：

```
ranges = [
    [10, 20],
    [1, 5],
    [70, 80],
    [27, 92],
    [0, 16]
]
```

对应的正确输出是:

```
[165, 15, 825, 3927, 136]
5068
```

扩展题

- 如果在一个`for`循环中使用`sum()`函数，那么你的时间复杂度是多少?
- 从1到n的整数和的计算方程为`n*(n+1)//2`，你能用它得到更好的时间复杂度吗?可以好多少?

附录:JSON {#appendix-json}

在该最终项目中，我们需要完成编码和解码JSON数据。如果你对该格式不是很熟悉，可以从Wikipedia上查看它的说明。在本节中，我们将会学习，什么叫JSON数据的编码和解码。

JSON格式和原始格式

下面是一个JSON对象的例子:

```
{
    "name": "Ada Lovelace",
    "country": "England",
    "years": [ 1815, 1852 ]
}
```

在Python中，你可以按照下面的方式构造一个`dict`对象:

```
d = {
    "name": "Ada Lovelace",
    "country": "England",
    "years": [ 1815, 1852 ]
}
```

但是，这里的关键不同处在于: *所有的JSON数据都是字符串*。 JSON是所需要数据的字符串表示形式。

相互转换

如果你有一个JSON字符串，你可以使用Python中的 `json.loads()` 函数将其转换原始数据模式。

```
import json

data = json.loads('{ "name": "Ada" }')

print(data["name"])   # Prints Ada
```

同样，如果你一个Python数据，你可以通过调用`json.dumps()`将其转换为一个JSON字符串。

```
import json

data = { "name": "Ada" }

json_data = json.dumps(data)
```

```
print(json_data)  # Prints {"name": "Ada"}
```

优雅输出

如果你有一个完整的对象，`json.dumps()` 会将所有内容放在一行中输出。

比如下面这段代码:

```
d = {
    "name": "Ada Lovelace",
    "country": "England",
    "years": [ 1815, 1852 ]
}

json.dumps(d)
```

将会输出:

```
'{"name": "Ada Lovelace", "country": "England", "years": [1815, 1852]}'
```

你可以通过向`json.dumps()`传入 `indent`参数来使其变得整齐，给它赋予一个锁进的级别。

```
json.dumps(d, indent=4)
```

这个将会输出:

```
{
    "name": "Ada Lovelace",
    "country": "England",
    "years": [
        1815,
        1852
    ]
}
```

清楚多了。

双引号是十分重要的

JSON需要将字符串和键名都放在双引号中，单引号是无效的，没有引号 *更是* 非法的。

思考题

- JSON对象和Python字典对象的区别是什么？
- 阅读下Wikipedia文章，哪些类型可以被JSON表示？