目录

Python 中的 Socket 编程

说明

本书翻译自 realpython 网站上的文章教程 Socket Programming in Python (Guide)，由于原文很长，所以整理成了 Gitbook 方便阅读。你可以去 首页 下载 PDF/Mobi/ePub 格式文件或者 在线阅读

原作者

> Nathan Jennings 是 Real Python 教程团队的一员，他在很早之前就使用 C 语言开始了自己的编程生涯，但是最终发现了 Python，从 Web 应用和网络数据收集到网络安全，他喜欢任何 Pythonic 的东西 —— realpython

译者注

译者 是一名前端工程师，平常会写很多的 JavaScript。但是当我使用 JavaScript 很长一段时间后，会对一些 语言无关 的编程概念感兴趣，比如：网络 /socket 编程、异步 / 并发、线 / 进程通信等。然而恰好这些内容在 JavasScript 领域很少见

因为一直从事 Web 开发，所以我认为理解了网络通信及其 socket 编程就理解了 Web 开发的某些本质。过程中我发现 Python 社区有很多我喜欢的内容，并且很多都是高质量的公开发布且开源的

最近我发现了这篇文章，系统地从底层网络通信讲到了应用层协议及其 C/S 架构的应用程序，由浅入深。虽然代码、API 使用了 Python 语言，但是底层原理相通。非常值得一读，推荐给大家

另外，由于本人水平所限，翻译的内容难免出现偏差，如果你在阅读的过程中发现问题，请毫不犹豫的提醒我或者开新 PR。或者有什么不理解的地方也可以开 issue 讨论，当然 star 也是欢迎的

授权

开始吧»

开始

网络中的 Socket 和 Socket API 是用来跨网络传送消息的，它提供了 进程间通信 (IPC) 的一种形式。网络可以是逻辑的、本地的电脑网络，或者是可以物理连接到外网的网络，并且可以连接到其它网络。英特网就是一个明显的例子，就是那个你通过 ISP 连接到的网络

本篇教程有三个不同的迭代阶段，来展示如何使用 Python 构建一个 Socket 服务器和客户端

1. 我们将以一个简单的 Socket 服务器和客户端程序来开始本教程
2. 当你看完 API 了解例子是怎么运行起来以后，我们将会看到一个具有同时处理多个连接能力的例子的改进版
3. 最后，我们将会开发出一个更加完善且具有完整的自定义头信息和内容的 Socket 应用

教程结束后，你将学会如何使用 Python 中的 socket 模块 来写一个自己的客户端 / 服务器应用。以及向你展示如何在你的应用中使用自定义类在不同的端之间发送消息和数据

所有的例子程序都使用 Python 3.6 编写，你可以在 Github 上找到 源代码

网络和 Socket 是个很大的话题。网上已经有了关于它们的字面解释，如果你还不是很了解 Socket 和网络。当你你读到那些解释的时候会感到不知所措，这是非常正常的。因为我也是这样过来的

尽管如此也不要气馁。 我已经为你写了这个教程。 就像学习 Python 一样，我们可以一次学习一点。用你的浏览器保存本页面到书签，以便你学习下一部分时能找到

让我们开始吧！

背景知识

Socket 有一段很长的历史，最初是在 1971 年被用于 ARPANET，随后就成了 1983 年发布的 Berkeley Software Distribution (BSD) 操作系统的 API，并且被命名为 Berkeley socket

当互联网在 20 世纪 90 年代随万维网兴起时，网络编程也火了起来。Web 服务和浏览器并不是唯一使用新的连接网络和 Socket 的应用程序。各种类型不同规模的客户端 / 服务器应用都广泛地使用着它们

时至今日，尽管 Socket API 使用的底层协议已经进化了很多年，也出现了许多新的协议，但是底层的 API 仍然保持不变

Socket 应用最常见的类型就是 客户端 / 服务器 应用，服务器用来等待客户端的链接。我们教程中涉及到的就是这类应用。更明确地说，我们将看到用于 Internet Socket 的 Socket API，有时称为 Berkeley 或 BSD Socket。当然也有 Unix domain sockets —— 一种用于 同一主机进程间的通信

Socket API 概览

Python 的 socket 模块提供了使用 Berkeley sockets API 的接口。这将会在我们这个教程里使用和讨论到

主要的用到的 Socket API 函数和方法有下面这些：

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `connect()`
- `connect_ex()`
- `send()`
- `recv()`
- `close()`

Python 提供了和 C 语言一致且方便的 API。我们将在下面一节中用到它们

作为标准库的一部分，Python 也有一些类可以让我们方便的调用这些底层 Socket 函数。尽管这个教程中并没有涉及这部分内容，你也可以通过 socketserver 模块 中找到文档。当然还有很多实现了高层网络协议（比如：HTTP, SMTP）的的模块。做为了解，可以查看这个链接中的内容 Internet Protocols and Support

TCP Sockets

就如你马上要看到的，我们将使用 `socket.socket()` 创建一个类型为 `socket.SOCK_STREAM` 的 socket 对象，默认将使用 Transmission Control Protocol(TCP) 协议，这基本上就是你想使用的默认值

为什么应该使用 TCP 协议？

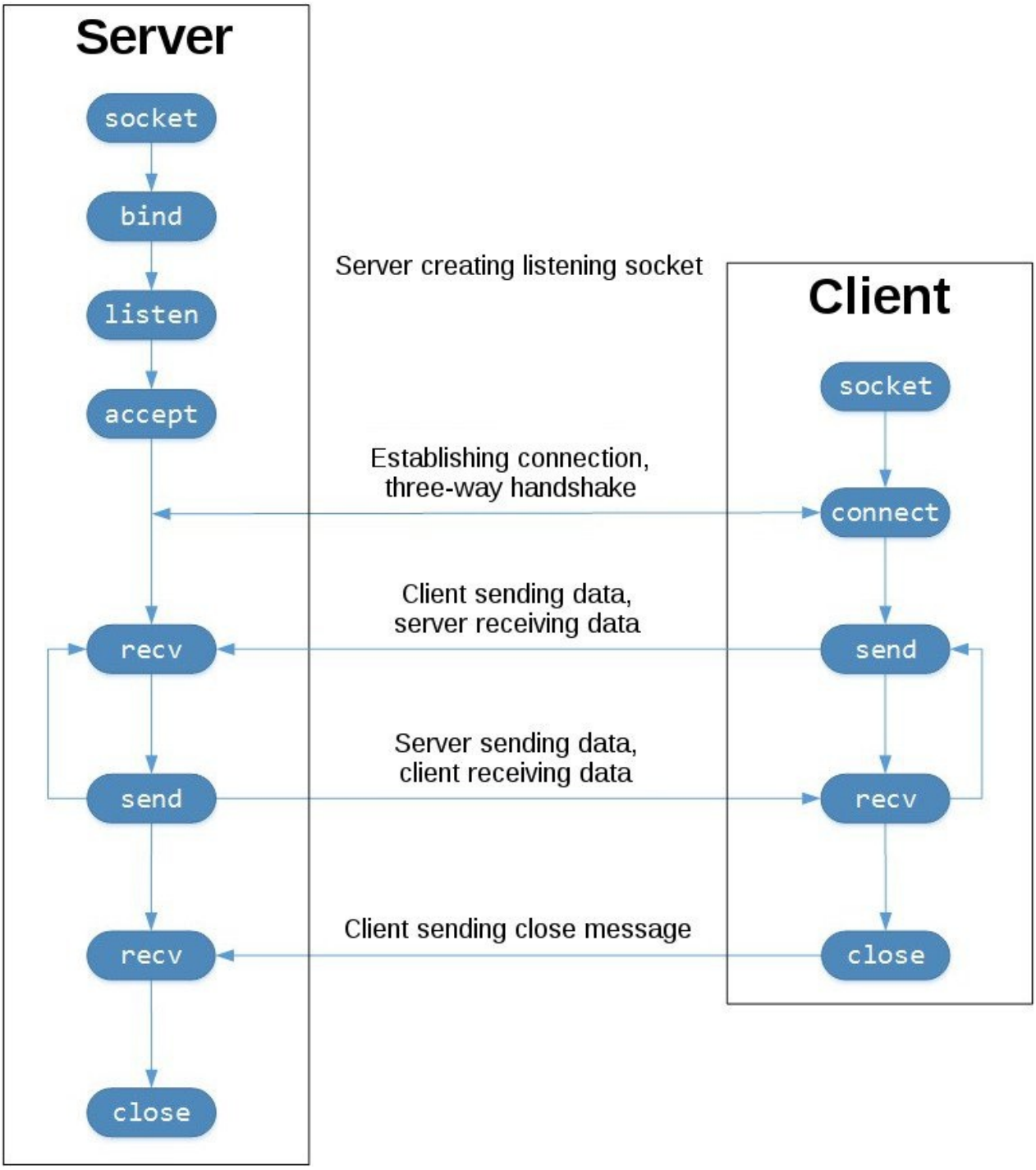
- 可靠的：网络传输中丢失的数据包会被检测到并重新发送
- 有序传送：数据按发送者写入的顺序被读取

相反，使用 `socket.SOCK_DGRAM` 创建的 用户数据报协议 (UDP) Socket 是 不可靠 的，而且数据的读取写发送可以是 无序的

为什么这个很重要？网络总是会尽最大的努力去传输完整数据（往往不尽人意）。没法保证你的数据一定被送到目的地或者一定能接收到别人发送给你的数据

网络设备（比如：路由器、交换机）都有带宽限制，或者系统本身的极限。它们也有 CPU、内存、总线和接口包缓冲区，就像我们的客户端和服务器。TCP 消除了你对于丢包、乱序以及其它网络通信中通常出现的问题的顾虑

下面的示意图中，我们将看到 Socket API 的调用顺序和 TCP 的数据流：



左边表示服务器，右边则是客户端

左上方开始，注意服务器创建「监听」Socket 的 API 调用：

- `socket()`
- `bind()`
- `listen()`
- `accept()`

「监听」Socket 做的事情就像它的名字一样。它会监听客户端的连接，当一个客户端连接进来的时候，服务器将调用 `accept()` 来「接受」或者「完成」此连接

客户端调用 `connect()` 方法来建立与服务器的链接，并开始三次握手。握手很重要是因为它保证了网络的通信的双方可以到达，也就是说客户端可以正常连接到服务器，反之亦然

上图中间部分往返部分表示客户端和服务器的数据交换过程，调用了 `send()` 和 `recv()` 方法

下面部分，客户端和服务器调用 `close()` 方法来关闭各自的 socket

客户端 / 服务器打印程序

你现在已经了解了基本的 socket API 以及客户端和服务器是如何通信的，让我们来创建一个客户端和服务器。我们将会以一个简单的实现开始。服务器将打印客户端发送回来的内容

打印程序的服务端

下面就是服务器代码，`echo-server.py`：

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1'  # 标准的回环地址 (localhost)
PORT = 65432        # 监听的端口 （非系统级的端口：大于 1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

> 注意：上面的代码你可能还没法完全理解，但是不用担心。这几行代码做了很多事情，这只是一个起点，帮你看见这个简单的服务器是如何运行的教程后面有引用部分，里面有很多额外的引用资源链接，这个教程中我将把链接放在那儿

让我们一起来看一下 API 调用以及发生了什么

`socket.socket()` 创建了一个 socket 对象，并且支持 上下文管理器，你可以使用 with 语句，这样你就不用再手动调用 `s.close()` 来关闭 socket 了

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    pass  # Use the socket object without calling s.close().
```

调用 `socket()` 时传入的 socket 地址族参数 `socket.AF_INET` 表示因特网 IPv4 地址族，`SOCK_STREAM` 表示使用 TCP 的 socket 类型，协议将被用来在网络中传输消息

`bind()` 用来关联 socket 到指定的网络接口（IP 地址）和端口号：

```
HOST = '127.0.0.1'
PORT = 65432

# ...

s.bind((HOST, PORT))
```

`bind()` 方法的入参取决于 socket 的地址族，在这个例子中我们使用了 `socket.AF_INET` (IPv4)，它将返回两个元素的元组：(host, port)

host 可以是主机名称、IP 地址、空字符串，如果使用 IP 地址，host 就应该是 IPv4 格式的字符串， `127.0.0.1` 是标准的 IPv4 回环地址，只有主机上的进程可以连接到服务器，如果你传了空字符串，服务器将接受本机所有可用的 IPv4 地址

端口号应该是 1-65535 之间的整数（0 是保留的），这个整数就是用来接受客户端链接的 TCP 端口号，如果端口号小于 1024，有的操作系统会要求管理员权限

使用 `bind()` 传参为主机名称的时候需要注意：

> 如果你在 host 部分 主机名称 作为 IPv4/v6 socket 的地址，程序可能会产生非确定性的行为，因为 Python 会使用 DNS 解析后的 第一个 地址，根据 DNS 解析的结果或者 host 配置 socket 地址将会以不同方式解析为实际的 IPv4/v6 地址。如果想得到确定的结果传入的 host 参数建议使用数字格式的地址 引用

我稍后将在 使用主机名 部分讨论这个问题，但是现在也值得一提。目前来说你只需要知道当使用主机名时，你将会因为 DNS 解析的原因得到不同的结果

可能是任何地址。比如第一次运行程序时是 10.1.2.3，第二次是 192.168.0.1，第三次是 172.16.7.8 等等

继续看上面的服务器代码示例， `listen()` 方法调用使服务器可以接受连接请求，这使它成为一个「监听中」的 socket

```
s.listen()
conn, addr = s.accept()
```

`listen()` 方法有一个 `backlog` 参数。它指定在拒绝新的连接之前系统将允许使用的 未接受的连接 数量。从 Python 3.5 开始，这是可选参数。如果不指定，Python 将取一个默认值

如果你的服务器需要同时接收很多连接请求，增加 backlog 参数的值可以加大等待链接请求队列的长度，最大长度取决于操作系统。比如在 Linux 下，参考 /proc/sys/net/core/somaxconn

`accept()` 方法阻塞并等待传入连接。当一个客户端连接时，它将返回一个新的 socket 对象，对象中有表示当前连接的 conn 和一个由主机、端口号组成的 IPv4/v6 连接的元组，更多关于元组值的内容可以查看 socket 地址族 一节中的详情

这里必须要明白我们通过调用 `accept()` 方法拥有了一个新的 socket 对象。这非常重要，因为你将用这个 socket 对象和客户端进行通信。和监听一个 socket 不同的是后者只用来授受新的连接请求

```
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data:
            break
        conn.sendall(data)
```

从 `accept()` 获取客户端 socket 连接对象 conn 后，使用一个无限 while 循环来阻塞调用 `conn.recv()`，无论客户端传过来什么数据都会使用 `conn.sendall()` 打印出来

如果 `conn.recv()` 方法返回一个空 byte 对象（ `b''` ），然后客户端关闭连接，循环结束，with 语句和 conn 一起使用时，通信结束的时候会自动关闭 socket 链接

打印程序的客户端

现在我们来看下客户端的程序，`echo-client.py` ：

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1'  # 服务器的主机名或者 IP 地址
PORT = 65432        # 服务器使用的端口

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)

print('Received', repr(data))
```

与服务器程序相比，客户端程序简单很多。它创建了一个 socket 对象，连接到服务器并且调用 `s.sendall()` 方法发送消息，然后再调用 `s.recv()` 方法读取服务器返回的内容并打印出来

运行打印程序的客户端和服务端

让我们运行打印程序的客户端和服务端，观察他们的表现，看看发生了什么事情

> 如果你在运行示例代码时遇到了问题，可以阅读 如何使用 Python 开发命令行命令，如果你使用的是 windows 操作系统，请查看 Python Windows FAQ

打开命令行程序，进入你的代码所在的目录，运行打印程序的服务端：

```
$ ./echo-server.py
```

你的命令行将被挂起，因为程序有一个阻塞调用

```
conn, addr = s.accept()
```

它将等待客户端的连接，现在再打开一个命令行窗口运行打印程序的客户端：

```
$ ./echo-client.py
Received b'Hello, world'
```

在服务端的窗口你将看见：

```
$ ./echo-server.py
Connected by ('127.0.0.1', 64623)
```

上面的输出中，服务端打印出了 `s.accept()` 返回的 addr 元组，这就是客户端的 IP 地址和 TCP 端口号。示例中的端口号是 64623 这很可能是和你机器上运行的结果不同

查看 socket 状态

想查找你主机上 socket 的当前状态，可以使用 `netstat` 命令。这个命令在 macOS, Window, Linux 系统上默认可用

下面这个就是启动服务后 netstat 命令的输出结果：

```
$ netstat -an
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  127.0.0.1.65432        *.*                    LISTEN
```

注意本地地址是 127.0.0.1.65432，如果 `echo-server.py` 文件中 `HOST` 设置成空字符串 `''` 的话，netstat 命令将显示如下：

```
$ netstat -an
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  *.65432                *.*                    LISTEN
```

本地地址是 `*.65432`，这表示所有主机支持的 IP 地址族都可以接受传入连接，在我们的例子里面调用 `socket()` 时传入的参数 `socket.AF_INET` 表示使用了 IPv4 的 TCP socket，你可以在输出结果中的 `Proto` 列中看到 (tcp4)

上面的输出是我截取的只显示了咱们的打印程序服务端进程，你可能会看到更多输出，具体取决于你运行的系统。需要注意的是 Proto, Local Address 和 state 列。分别表示 TCP socket 类型、本地地址端口、当前状态

另外一个查看这些信息的方法是使用 `lsof` 命令，这个命令在 macOS 上是默认安装的，Linux 上需要你手动安装

```
$ lsof -i -n
COMMAND     PID   USER   FD   TYPE   DEVICE SIZE/OFF NODE NAME
Python    67982 nathan    3u  IPv4 0xecf272      0t0  TCP *:65432 (LISTEN)
```

isof 命令使用 `-i` 参数可以查看打开的 socket 连接的 COMMAND, PID(process id) 和 USER(user id)，上面的输出就是打印程序服务端

`netstat` 和 `isof` 命令有许多可用的参数，这取决于你使用的操作系统。可以使用 man page 来查看他们的使用文档，这些文档绝对值得花一点时间去了解，你将受益匪浅，macOS 和 Linux 中使用命令 `man netstat` 或者 `man lsof` 命令，windows 下使用 `netstat /?` 来查看帮助文档

一个通常会犯的错误是在没有监听 socket 端口的情况下尝试连接：

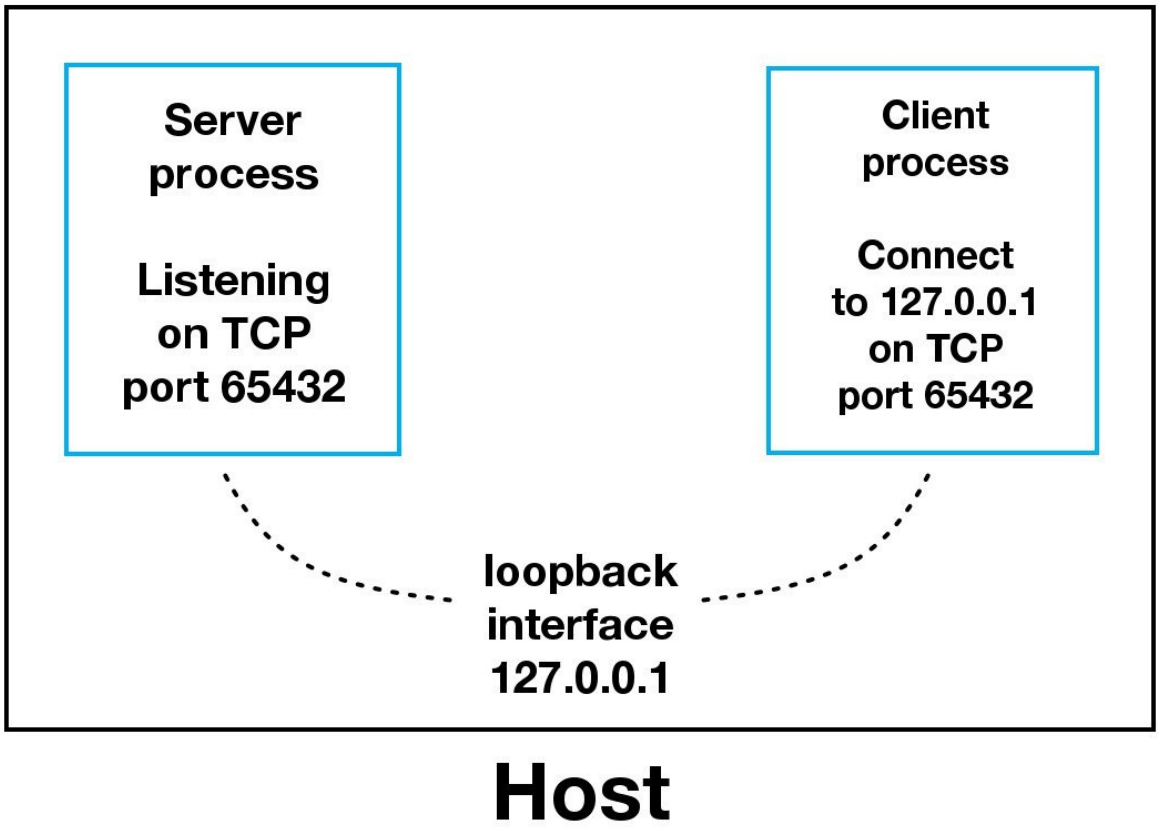
```
$ ./echo-client.py
Traceback (most recent call last):
  File "./echo-client.py", line 9, in <module>
    s.connect((HOST, PORT))
ConnectionRefusedError: [Errno 61] Connection refused
```

也可能是端口号出错、服务端没启动或者有防火墙阻止了连接，这些原因可能很难记住，或许你也会碰到 `Connection timed out` 的错误，记得给你的防火墙添加允许我们使用的端口规则

引用部分有一些常见的 错误信息

通信流程的分解

让我们再仔细的观察下客户端是如何与服务端进行通信的：

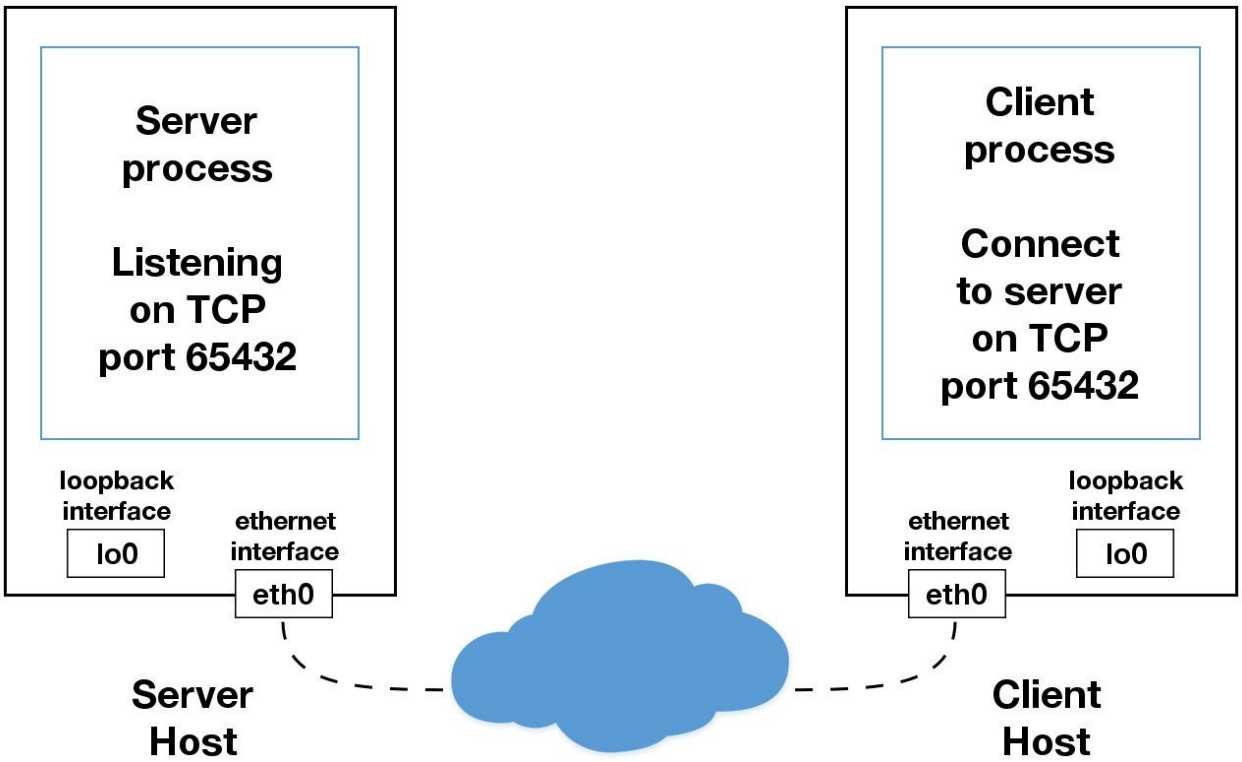


当使用回环地址时，数据将不会接触到外部网络，上图中，回环地址包含在了 host 里面。这就是回环地址的本质，连接数据传输是从本地到主机，这就是为什么你会听到有回环地址或者 `127.0.0.1` 、 `::1` 的 IP 地址和表示本地主机

应用程序使用回环地址来与主机上的其它进程通信，这使得它与外部网络安全隔离。由于它是内部的，只能从主机内访问，所以它不会被暴露出去

如果你的应用程序服务器使用自己的专用数据库（非公用的），则可以配置服务器仅监听回环地址，这样的话网络上的其它主机就无法连接到你的数据库

如果你的应用程序中使用的 IP 地址不是 127.0.0.1 或者 ::1，那就可能会绑定到连接到外部网络的以太网上。这就是你通往 localhost 王国之外的其他主机的大门



这里需要小心，并且可能让你感到难受甚至怀疑全世界。在你探索 localhost 的安全限制之前，确认读过 使用主机名 一节。 一个安全注意事项是 不要使用主机名，要使用 **IP** 地址

处理多个连接

打印程序的服务端肯定有它自己的一些局限。这个程序只能服务于一个客户端然后结束。打印程序的客户端也有它自己的局限，但是还有一个问题，如果客户端调用了下面的方法 `s.recv()` 方法将返回 `b'Hello, world'` 中的一个字节 `b'H'`

```
data = s.recv(1024)
```

1024 是缓冲区数据大小限制最大值参数 `bufsize`，并不是说 `recv()` 方法只返回 1024 个字节的内容

`send()` 方法也是这个原理，它返回发送内容的字节数，结果可能小于传入的发送内容，你得处理这处情况，按需多次调用 `send()` 方法来发送完整的数据

> 应用程序负责检查是否已发送所有数据；如果仅传输了一些数据，则应用程序需要尝试传递剩余数据 引用

我们可以使用 `sendall()` 方法来回避这个过程

> 和 send() 方法不一样的是，`sendall()` 方法会一直发送字节，只到所有的数据传输完成或者中途出现错误。成功的话会返回 None 引用

到目前为止，我们有两个问题：

- 如何同时处理多个连接请求
- 我们需要一直调用 `send()` 或者 `recv()` 直到所有数据传输完成

应该怎么做呢，有很多方式可以实现并发。最近，有一个非常流程的库叫做 Asynchronous I/O 可以实现，asyncio 库在 Python 3.4 后默认添加到了标准库里面。传统的方法是使用线程

并发的问题是很难做到正确，有许多细微之处需要考虑和防范。可能其中一个细节的问题都会导致整个程序崩溃

我说这些并不是想吓跑你或者让你远离学习和使用并发编程。如果你想让程序支持大规模使用，使用多处理器、多核是很有必要的。然而在这个教程中我们将使用比线程更传统的方法使得逻辑更容易推理。我们将使用一个非常古老的系统调用：`select()`

`select()` 允许你检查多个 socket 的 I/O 完成情况，所以你可以使用它来检测哪个 socket I/O 是就绪状态从而执行读取或写入操作，但是这是 Python，总会有更多其它的选择，我们将使用标准库中的 selectors 模块，所以我们使用了最有效的实现，不用在意你使用的操作系统：

> 这个模块提供了高层且高效的 I/O 多路复用，基于原始的 `select` 模块构建，推荐用户使用这个模块，除非他们需要精确到操作系统层面的使用控制 引用

尽管如此，使用 `select()` 也无法并发执行。这取决于您的工作负载，这种实现仍然会很快。这也取决于你的应用程序对连接所做的具体事情或者它需要支持的客户端数量

asyncio 使用单线程来处理多任务，使用事件循环来管理任务。通过使用 `select()`，我们可以创建自己的事件循环，更简单且同步化。当使用多线程时，即使要处理并发的情况，我们也不得不面临使用 CPython 或者 PyPy 中的「全局解析器锁 GIL」，这有效地限制了我们可以并行完成的工作量

说这些是为了解析为什么使用 `select()` 可能是个更好的选择，不要觉得你必须使用 asyncio、线程或最新的异步库。通常，在网络应用程序中，你的应用程序就是 I/O 绑定：它可以在本地网络上，网络另一端的端，磁盘上等待

如果你从客户端收到启动 CPU 绑定工作的请求，查看 concurrent.futures 模块，它包含一个 ProcessPoolExecutor 类，用来异步执行进程池中的调用

如果你使用多进程，你的 Python 代码将被操作系统并行地在不同处理器或者核心上调度运行，并且没有全局解析器锁。你可以通过 Python 大会上的演讲 John Reese - Thinking Outside the GIL with AsyncIO and Multiprocessing - PyCon 2018 来了解更多的想法

在下一节中，我们将介绍解决这些问题的服务器和客户端的示例。他们使用 `select()` 来同时处理多连接请求，按需多次调用 `send()` 和 `recv()`

多连接的客户端 / 服务器程序

下面两节中，我们将使用 selectors 模块中的 selector 对象来创建一个可以同时处理多个请求的客户端和服务端

多连接的服务端

首先，我们来看眼多连接服务端程序的代码，`multiconn-server.py`。这是开始建立监听 socket 部分

```
import selectors
sel = selectors.DefaultSelector()
# ...
lsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
lsock.bind((host, port))
lsock.listen()
print('listening on', (host, port))
lsock.setblocking(False)
sel.register(lsock, selectors.EVENT_READ, data=None)
```

这个程序和之前打印程序服务端最大的不同是使用了 `lsock.setblocking(False)` 配置 socket 为非阻塞模式，这个 socket 的调用将不在是阻塞的。当它和 `sel.select()` 一起使用的时候（下面会提到），我们就可以等待 socket 就绪事件，然后执行读写操作

`sel.register()` 使用 `sel.select()` 为你感兴趣的事件注册 socket 监控，对于监听 socket，我们希望使用 `selectors.EVENT_READ` 读取到事件

`data` 用来存储任何你 socket 中想存的数据，当 `select()` 返回的时候它也会返回。我们将使用 `data` 来跟踪 socket 上发送或者接收的东西

下面就是事件循环：

```
import selectors
sel = selectors.DefaultSelector()

# ...

while True:
    events = sel.select(timeout=None)
    for key, mask in events:
        if key.data is None:
            accept_wrapper(key.fileobj)
        else:
            service_connection(key, mask)
```

`sel.select(timeout=None)` 调用会阻塞直到 socket I/O 就绪。它返回一个 (key, events) 元组，每个 socket 都有一个。key 就是一个包含 `fileobj` 属性的具名元组。`key.fileobj` 是一个 socket 对象，`mask` 表示一个操作就绪的事件掩码

如果 `key.data` 为空，我们就可以知道它来自于监听 socket，我们需要调用 `accept()` 方法来授受连接请求。我们将使用一个 `accept()` 包装函数来获取新的 socket 对象并注册到 `selector` 上，我们马上就会看到

如果 `key.data` 不为空，我们就可以知道它是一个被接受的客户端 socket，我们需要为它服务，接着 `service_connection()` 会传入 `key` 和 `mask` 参数并调用，这包含了所有我们需要在 socket 上操作的东西

让我们一起来看看 `accept_wrapper()` 方法做了什么：

```
def accept_wrapper(sock):
    conn, addr = sock.accept()  # Should be ready to read
    print('accepted connection from', addr)
    conn.setblocking(False)
    data = types.SimpleNamespace(addr=addr, inb=b'', outb=b'')
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    sel.register(conn, events, data=data)
```

由于监听 socket 被注册到了 `selectors.EVENT_READ` 上，它现在就能被读取，我们调用 `sock.accept()` 后立即再立即调 `conn.setblocking(False)` 来让 socket 进入非阻塞模式

请记住，这是这个版本服务器程序的主要目标，因为我们不希望它被阻塞。如果被阻塞，那么整个服务器在返回前都处于挂起状态。这意味着其它 socket 处于等待状态，这是一种 非常严重的 谁都不想见到的服务被挂起的状态

接着我们使用了 `types.SimpleNamespace` 类创建了一个对象用来保存我们想要的 socket 和数据，由于我们得知道客户端连接什么时候可以写入或者读取，下面两个事件都会被用到：

```
events = selectors.EVENT_READ | selectors.EVENT_WRITE
```

事件掩码、socket 和数据对象都会被传入 `sel.register()`

现在让我们来看下，当客户端 socket 就绪的时候连接请求是如何使用 `service_connection()` 来处理的

```
def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to read
        if recv_data:
            data.outb += recv_data
        else:
            print('closing connection to', data.addr)
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if data.outb:
            print('echoing', repr(data.outb), 'to', data.addr)
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]
```

这就是多连接服务端的核心部分，`key` 就是从调用 `select()` 方法返回的一个具名元组，它包含了 socket 对象「fileobj」和数据对象。`mask` 包含了就绪的事件

如果 socket 就绪而且可以被读取，`mask & selectors.EVENT_READ` 就为真，`sock.recv()` 会被调用。所有读取到的数据都会被追加到 `data.outb` 里面。随后被发送出去

注意 `else:` 语句，如果没有收到任何数据：

```
if recv_data:
    data.outb += recv_data
else:
    print('closing connection to', data.addr)
    sel.unregister(sock)
    sock.close()
```

这表示客户端关闭了它的 socket 连接，这时服务端也应该关闭自己的连接。不过别忘了先调用 `sel.unregister()` 来撤销 `select()` 的监控

当 socket 就绪而且可以被读取的时候，对于正常的 socket 应该一直是这种状态，任何接收并被 `data.outb` 存储的数据都将使用 `sock.send()` 方法打印出来。发送出去的字节随后就会被从缓冲中删除

```
data.outb = data.outb[sent:]
```

多连接的客户端

现在让我们一起来看看多连接的客户端程序，`multiconn-client.py`，它和服务端很相似，不一样的是它没有监听连接请求，它以调用 `start_connections()` 开始初始化连接：

```
messages = [b'Message 1 from client.', b'Message 2 from client.']


def start_connections(host, port, num_conns):
    server_addr = (host, port)
    for i in range(0, num_conns):
        connid = i + 1
        print('starting connection', connid, 'to', server_addr)
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.setblocking(False)
        sock.connect_ex(server_addr)
        events = selectors.EVENT_READ | selectors.EVENT_WRITE
        data = types.SimpleNamespace(connid=connid,
                                     msg_total=sum(len(m) for m in messages),
                                     recv_total=0,
                                     messages=list(messages),
                                     outb=b'')
        sel.register(sock, events, data=data)
```

`num_conns` 参数是从命令行读取的，表示为服务器建立多少个链接。就像服务端程序一样，每个 socket 都设置成了非阻塞模式

由于 `connect()` 方法会立即触发一个 `BlockingIOError` 异常，所以我们使用 `connect_ex()` 方法取代它。`connect_ex()` 会返回一个错误指示 `errno.EINPROGRESS`，不像 `connect()` 方法直接在进程中返回异常。一旦连接结束，socket 就可以进行读写并且通过 `select()` 方法返回

socket 建立完成后，我们将使用 `types.SimpleNamespace` 类创建想会传送的数据。由于每个连接请求都会调用 `socket.send()`，发送到服务端的消息得使用 `list(messages)` 方法转换成列表结构。所有你想了解的东西，包括客户端将要发送的、已发送的、已接收的消息以及消息的总字节数都存储在 `data` 对象中

让我们再来看看 `service_connection()`。基本上和服务端一样：

```
def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to read
        if recv_data:
            print('received', repr(recv_data), 'from connection', data.connid)
            data.recv_total += len(recv_data)
        if not recv_data or data.recv_total == data.msg_total:
            print('closing connection', data.connid)
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if not data.outb and data.messages:
            data.outb = data.messages.pop(0)
        if data.outb:
            print('sending', repr(data.outb), 'to connection', data.connid)
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]
```

有一个不同的地方，客户端会跟踪从服务器接收的字节数，根据结果来决定是否关闭 socket 连接，服务端检测到客户端关闭则会同样的关闭服务端的连接

运行多连接的客户端和服务端程序

现在让我们把 `multiconn-server.py` 和 `multiconn-client.py` 两个程序跑起来。他们都使用了命令行参数，如果不指定参数可以看到参数调用的方法：

服务端程序，传入主机和端口号

```
$ ./multiconn-server.py
usage: ./multiconn-server.py <host> <port>
```

客户端程序，传入启动服务端程序时同样的主机和端口号以及连接数量

```
$ ./multiconn-client.py
usage: ./multiconn-client.py <host> <port> <num_connections>
```

下面就是服务端程序运行起来在 65432 端口上监听回环地址的输出：

```
$ ./multiconn-server.py 127.0.0.1 65432
listening on ('127.0.0.1', 65432)
accepted connection from ('127.0.0.1', 61354)
accepted connection from ('127.0.0.1', 61355)
echoing b'Message 1 from client.Message 2 from client.' to ('127.0.0.1', 61354)
echoing b'Message 1 from client.Message 2 from client.' to ('127.0.0.1', 61355)
closing connection to ('127.0.0.1', 61354)
closing connection to ('127.0.0.1', 61355)
```

下面是客户端，它创建了两个连接请求到上面的服务端：

```
$ ./multiconn-client.py 127.0.0.1 65432 2
starting connection 1 to ('127.0.0.1', 65432)
starting connection 2 to ('127.0.0.1', 65432)
sending b'Message 1 from client.' to connection 1
sending b'Message 2 from client.' to connection 1
sending b'Message 1 from client.' to connection 2
sending b'Message 2 from client.' to connection 2
received b'Message 1 from client.Message 2 from client.' from connection 1
closing connection 1
received b'Message 1 from client.Message 2 from client.' from connection 2
closing connection 2
```

客户端 / 服务器应用程序

多连接的客户端和服务端程序版本与最早的原始版本相比肯定有了很大的改善，但是让我们再进一步地解决上面「多连接」版本中的不足，然后完成最终版的实现：客户端 / 服务器应用程序

我们希望有个客户端和服务端在不影响其它连接的情况下做好错误处理，显然，如果没有发生异常，我们的客户端和服务端不能崩溃的一团糟。这也是到现在为止我们还没讨论的东西，我故意没有引入错误处理机制因为这样可以使之前的程序容易理解

现在你对基本的 API，非阻塞 socket、select() 等概念已经有所了解了。我们可以继续添加一些错误处理同时讨论下「房间里面的大象」的问题，我把一些东西隐藏在了幕后。你应该还记得，我在介绍中讨论到的自定义类

首先，让我们先解决错误：

> 所有的错误都会触发异常，像无效参数类型和内存不足的常见异常可以被抛出；从 Python 3.3 开始，与 socket 或地址语义相关的错误会引发 OSError 或其子类之一的异常
> 引用

我们需要捕获 `OSError` 异常。另外一个我没提及的的问题是延迟，你将在文档的很多地方看见关于延迟的讨论，延迟会发生而且属于「正常」错误。主机或者路由器重启、交换机端口出错、电缆出问题或者被拔出，你应该在你的代码中处理好各种各样的错误

刚才说的「房间里面的大象」问题是怎么回事呢。就像 `socket.SOCK_STREAM` 这个参数的字面意思一样，当使用 TCP 连接时，你会从一个连续的字节流读取的数据，好比从磁盘上读取数据，不同的是你是从网络读取字节流

然而，和使用 `f.seek()` 读文件不同，换句话说，没法定位 socket 的数据流的位置，如果可以像文件一样定位数据流的位置（使用下标），那你就可以随意的读取你想要的数据

当字节流入你的 socket 时，会需要有不同的网络缓冲区，如果想读取他们就必须先保存到其它地方，使用 `recv()` 方法持续的从 socket 上读取可用的字节流

相当于你从 socket 中读取的是一块一块的数据，你必须使用 `recv()` 方法不断的从缓冲区中读取数据，直到你的应用确定读取到了足够的数据

什么时候算「足够」这取决于你的定义，就 TCP socket 而言，它只通过网络发送或接收原始字节。它并不了解这些原始字节的含义

这可以让我们定义一个应用层协议，什么是应用层协议？简单来说，你的应用会发送或者接收消息，这些消息其实就是你的应用程序的协议

换句话说，这些消息的长度、格式可以定义应用程序的语义和行为，这和我们之前说的从 socket 中读取字节部分内容相关，当你使用 `recv()` 来读取字节的时候，你需要知道读的字节数，并且决定什么时候算读取完成

这些都是怎么完成的呢？一个方法是只读取固定长度的消息，如果它们的长度总是一样的话，这样做很容易。当你收到固定长度字节消息的时候，就能确定它是个完整的消息

然而，如果你使用定长模式来发送比较短的消息会比较低效，因为你还得处理填充剩余的部分，此外，你还得处理数据不适合放在一个定长消息里面的情况

在这个教程里面，我们将使用一个通用的方案，很多协议都会用到它，包括 HTTP。我们将在每条消息前面追加一个头信息，头信息中包括消息的长度和其它我们需要的字段。这样做的话我们只需要追踪头信息，当我们读到头信息时，就可以查到消息的长度并且读出所有字节然后消费它

我们将通过使用一个自定义类来实现接收文本 / 二进制数据。你可以在此基础上做出改进或者通过继承这个类来扩展你的应用程序。重要的是你将看到一个例子实现它的过程

我将会提到一些关于 socket 和字节相关的东西，就像之前讨论过的。当你通过 socket 来发送或者接收数据时，其实你发送或者接收到的是原始字节

如果你收到数据并且想让它在一个多字节解释的上下文中使用，比如说 4-byte 的整形，你需要考虑它可能是一种不是你机器 CPU 本机的格式。客户端或者服务器的另外一头可能是另外一种使用了不同的字节序列的 CPU，这样的话，你就得把它们转换成你主机的本地字节序列来使用

上面所说的字节顺序就是 CPU 的 字节序，在引用部分的字节序 一节可以查看更多。我们将会利用 Unicode 字符集的优点来规避这个问题，并使用 UTF-8 的方式编码，由于 UTF-8 使用了 8 字节 编码方式，所以就不会有字节序列的问题

你可以查看 Python 关于编码与 Unicode 的 文档，注意我们只会编码消息的头部。我们将使用严格的类型，发送的消息编码格式会在头信息中定义。这将让我们可以传输我们觉得有用的任意类型 / 格式数据

你可以通过调用 `sys.byteorder` 来决定你的机器的字节序列，比如在我的英特尔笔记本上，运行下面的代码就可以：

```
$ python3 -c 'import sys; print(repr(sys.byteorder))'
'little'
```

如果我把这段代码跑在可以模拟大字节序 CPU「PowerPC」的虚拟机上的话，应该是下面的结果：

```
$ python3 -c 'import sys; print(repr(sys.byteorder))'
'big'
```

在我们的例子程序中，应用层的协议定义了使用 UTF-8 方式编码的 Unicode 字符。对于真正传输消息来说，如果需要的话你还是得手动交换字节序列

这取决于你的应用，是否需要它来处理不同终端间的多字节二进制数据，你可以通过添加额外的头信息来让你的客户端或者服务端支持二进制，像 HTTP 一样，把头信息做为参数传进去

不用担心自己还没搞懂上面的东西，下面一节我们看到是如果实现的

应用的协议头

让我们来定义一个完整的协议头：

- 可变长度的文本
- 基于 UTF-8 编码的 Unicode 字符集
- 使用 JSON 序列化的一个 Python 字典

其中必须具有的头应该有以下几个：

名称	描述
byteorder	机器的字节序列（uses sys.byteorder），应用程序可能用不上
content-length	内容的字节长度
content-type	内容的类型，比如 text/json 或者 binary/my-binary-type
content-encoding	内容的编码类型，比如 utf-8 编码的 Unicode 文本，二进制数据

这些头信息告诉接收者消息数据，这样的话你就可以通过提供给接收者足够的信息让他接收到数据的时候正确的解码的方式向它发送任何数据，由于头信息是字典格式，你可以随意向头信息中添加键值对

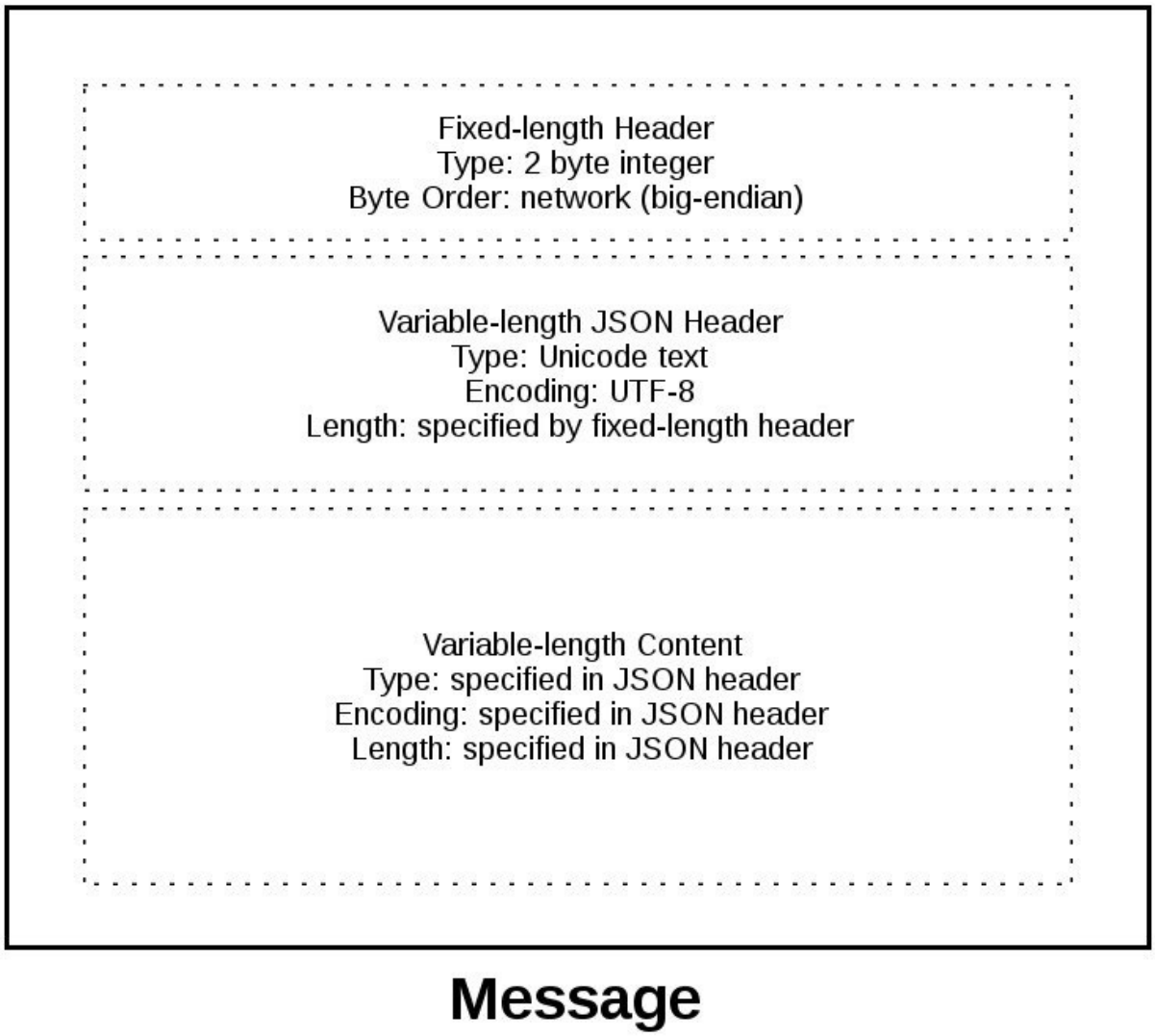
发送应用程序消息

不过还有一个问题，由于我们使用了变长的头信息，虽然方便扩展但是当你使用 `recv()` 方法读取消息的时候怎么知道头信息的长度呢

我们前面讲到过使用 `recv()` 接收数据和如何确定是否接收完成，我说过定长的头可能会很低效，的确如此。但是我们将使用一个比较小的 2 字节定长的头信息前缀来表示头信息的长度

你可以认为这是一种混合的发送消息的实现方法，我们通过发送头信息长度来引导接收者，方便他们解析消息体

为了给你更好地解释消息格式，让我们来看看消息的全貌：



消息以 2 字节的固定长度的头开始，这两个字节是整型的网络字节序列，表示下面的变长 JSON 头信息的长度，当我们从 `recv()` 方法读取到 2 个字节时就知道它表示的是头信息长度的整形数字，然后在解码 JSON 头之前读取这么多的字节

JSON 头包含了头信息的字典。其中一个就是 `content-length`，这表示消息内容的数量（不是 JSON 头），当我们使用 `recv()` 方法读取到了 `content-length` 个字节的数据时，就表示接收完成并且读取到了完整的消息

应用程序消息类

最后让我们来看下成果，我们使用了一个消息类。来看看它是如何在 socket 发生读写事件时与 `select()` 配合使用的

对于这个示例应用程序而言，我必须想出客户端和服务器将使用什么类型的消息，从这一点来讲这远远超过了最早时候我们写的那个玩具一样的打印程序

为了保证程序简单而且仍然能够演示出它是如何在一个真正的程序中工作的，我创建了一个应用程序协议用来实现基本的搜索功能。客户端发送一个搜索请求，服务器做一次匹配的查找，如果客户端的请求没法被识别成搜索请求，服务器就会假定这个是二进制请求，对应的返回二进制响应

跟着下面一节，运行示例、用代码做实验后你将会知道他是如何工作的，然后你就可以以这个消息类为起点把他修改成适合自己使用的

就像我们之前讨论的，你将在下面看到，处理 socket 时需要保存状态。通过使用类，我们可以将所有的状态、数据和代码打包到一个地方。当连接开始或者接受的时候消息类就会为每个 socket 创建一个实例

类中的很多包装方法、工具方法在客户端和服务端上都是差不多的。它们以下划线开头，就像 `Message._json_encode()` 一样，这些方法通过类使用起来很简单。这使得它们在其它方法中调用时更短，而且符合 DRY 原则

消息类的服务端程序本质上和客户端一样。不同的是客户端初始化连接并发送请求消息，随后要处理服务端返回的内容。而服务端则是等待连接请求，处理客户端的请求消息，随后发送响应消息

看起来就像这样：

步骤	端	动作 / 消息内容
1	客户端	发送带有请求内容的消息
2	服务端	接收并处理请求消息
3	服务端	发送有响应内容的消息
4	客户端	接收并处理响应消息

下面是代码的结构：

应用程序	文件	代码
服务端	app-server.py	服务端主程序
服务端	libserver.py	服务端消息类
客户端	app-client.py	客户端主程序
客户端	libclient.py	客户端消息类

消息入口点

我想通过首先提到它的设计方面来讨论 Message 类的工作方式，不过这对我来说并不是立马就能解释清楚的，只有在重构它至少五次之后我才能达到它目前的状态。为什么呢？因为要管理状态

当消息对象创建的时候，它就被一个使用 `selector.register()` 事件监控起来的 socket 关联起来了

```
message = libserver.Message(sel, conn, addr)
sel.register(conn, selectors.EVENT_READ, data=message)
```

> 注意，这一节中的一些代码来自服务端主程序与消息类，但是这部分内容的讨论在客户端也是一样的，我将在他们之间存在不同点的时候来解释客户端的版本

当 socket 上的事件就绪的时候，它就会被 `selector.select()` 方法返回。对过 key 对象的 data 属性获取到 message 的引用，然后在消息用调用一个方法：

```
while True:
    events = sel.select(timeout=None)
    for key, mask in events:
        # ...
        message = key.data
        message.process_events(mask)
```

观察上面的事件循环，可以看见 `sel.select()` 位于「司机位置」，它是阻塞的，在循环的上面等待。当 socket 上的读写事件就绪时，它就会为其服务。这表示间接的它也要负责调用 `process_events()` 方法。这就是我说 `process_events()` 方法是入口的原因

让我们来看下 `process_events()` 方法做了什么

```
def process_events(self, mask):
    if mask & selectors.EVENT_READ:
        self.read()
    if mask & selectors.EVENT_WRITE:
        self.write()
```

这样做很好，因为 `process_events()` 方法很简洁，它只可以做两件事情：调用 `read()` 和 `write()` 方法

这又把我们带回了状态管理的问题。在几次重构后，我决定如果别的方法依赖于状态变量里面的某个确定值，那么它们就只应该从 `read()` 和 `write()` 方法中调用，这将使处理 socket 事件的逻辑尽量的简单

可能说起来很简单，但是经历了前面几次类的迭代：混合了一些方法，检查当前状态、依赖于其它值、在 `read()` 或者 `write()` 方法外面调用处理数据的方法，最后这证明了这样管理起来很复杂

当然，你肯定需要把类按你自己的需求修改使它能够符合你的预期，但是我建议你尽可能把状态检查、依赖状态的调用的逻辑放在 `read()` 和 `write()` 方法里面

让我们来看看 `read()` 方法，这是服务端版本，但是客户端也是一样的。不同之处在于方法名称，一个（客户端）是 `process_response()` 另一个（服务端）是 `process_request()`

```
def read(self):
    self._read()

    if self._jsonheader_len is None:
        self.process_protoheader()

    if self._jsonheader_len is not None:
        if self.jsonheader is None:
            self.process_jsonheader()

    if self.jsonheader:
        if self.request is None:
            self.process_request()
```

`_read()` 方法首先被调用，然后调用 `socket.recv()` 从 socket 读取数据并存入到接收缓冲区

记住，当调用 `socket.recv()` 方法时，组成消息的所有数据并没有一次性全部到达。`socket.recv()` 方法可能需要调用很多次，这就是为什么在调用相关方法处理数据前每次都要检查状态

当一个方法开始处理消息时，首先要检查的就是接受缓冲区保存了足够的多读取的数据，如果确定，它们将继续处理各自的数据，然后把数据存到其它流程可能会用到的变量上，并且清空自己的缓冲区。由于一个消息有三个组件，所以会有三个状态检查和处理方法的调用：

Message Component	Method	Output
Fixed-length header	process_protoheader()	self._jsonheader_len
JSON header	process_jsonheader()	self.jsonheader
Content	process_request()	self.request

接下来，让我们一起看看 `write()` 方法，这是服务端的版本：

```
def write(self):
    if self.request:
        if not self.response_created:
            self.create_response()

    self._write()
```

`write()` 方法会首先检测是否有请求，如果有而且响应还没被创建的话 `create_response()` 方法就会被调用，它会设置状态变量 `response_created`，然后为发送缓冲区写入响应

如果发送缓冲区有数据，`write()` 方法会调用 `socket.send()` 方法

记住，当 `socket.send()` 被调用时，所有发送缓冲区的数据可能还没进入到发送队列，socket 的网络缓冲区可能满了，`socket.send()` 可能需要重新调用，这就是为什么需要检查状态的原因，`create_response()` 应该只被调用一次，但是 `_write()` 方法需要调用多次

客户端的 `write()` 版大体与服务端一致：

```
def write(self):
    if not self._request_queued:
        self.queue_request()

    self._write()

    if self._request_queued:
        if not self._send_buffer:
            # Set selector to listen for read events, we're done writing.
            self._set_selector_events_mask('r')
```

因为客户端首先初始化了一个连接请求到服务端，状态变量 `_request_queued` 被检查。如果请求还没加入到队列，就调用 `queue_request()` 方法创建一个请求写入到发送缓冲区中，同时也会使用变量 `_request_queued` 记录状态值防止多次调用

就像服务端一样，如果发送缓冲区有数据 `_write()` 方法会调用 `socket.send()` 方法

需要注意客户端版本的 `write()` 方法与服务端不同之处在于最后的请求是否加入到队列中的检查，这个我们将在客户端主程序中详细解释，原因是要告诉 `selector.select()` 停止监控 socket 的写入事件而且我们只对读取事件感兴趣，没有办法通知套接字是可写的

我将在这一节中留下一个悬念，这一节的主要目的是解释 `selector.select()` 方法是如何通过 `process_events()` 方法调用消息类以及它是如何工作的

这一点很重要，因为 `process_events()` 方法在连接的生命周期中将被调用很多次，因此，要确保那些只能被调用一次的方法正常工作，这些方法中要么需要检查自己的状态变量，要么需要检查调用者的方法中的状态变量

服务端主程序

在服务端主程序 `app-server.py` 中，主机、端口参数是通过命令行传递给程序的：

```
$ ./app-server.py
usage: ./app-server.py <host> <port>
```

例如需求监听本地回环地址上面的 65432 端口，需要执行：

```
$ ./app-server.py 127.0.0.1 65432
listening on ('127.0.0.1', 65432)
```

`<host>` 参数为空的话就可以监听主机上的所有 IP 地址

创建完 socket 后，一个传入参数 `socket.SO_REUSEADDR` 的方法 `to socket.setsockopt()` 将被调用

```
# Avoid bind() exception: OSError: [Errno 48] Address already in use
lsock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

设置这个参数是为了避免 `端口被占用` 的错误发生，如果当前程序使用的端口和之前的程序使用的一样，你就会发现连接处于 `TIME_WAIT` 状态

比如说，如果服务器主动关闭连接，服务器会保持为大概两分钟的 `TIME_WAIT` 状态，具体时长取决于你的操作系统。如果你想在两分钟内再开启一个服务，你将得到一个 `OSError` 表示 `端口被战胜` ，这样做是为了确保一些在途的数据包正确的被处理

事件循环会捕捉所有错误，以保证服务器正常运行：

```
while True:
    events = sel.select(timeout=None)
    for key, mask in events:
        if key.data is None:
            accept_wrapper(key.fileobj)
        else:
            message = key.data
            try:
                message.process_events(mask)
            except Exception:
                print('main: error: exception for',
                      f'{message.addr}:\n{traceback.format_exc()}')
                message.close()
```

当服务器接受到一个客户端连接时，消息对象就会被创建：

```
def accept_wrapper(sock):
    conn, addr = sock.accept()  # Should be ready to read
    print('accepted connection from', addr)
    conn.setblocking(False)
    message = libserver.Message(sel, conn, addr)
    sel.register(conn, selectors.EVENT_READ, data=message)
```

消息对象会通过 `sel.register()` 方法关联到 socket 上，而且它初始化就被设置成了只监控读事件。当请求被读取时，我们将通过监听到的写事件修改它

在服务器端采用这种方法的一个优点是，大多数情况下，当 socket 正常并且没有网络问题时，它始终是可写的

如果我们告诉 `sel.register()` 方法监控 `EVENT_WRITE` 写入事件，事件循环将会立即唤醒并通知我们这种情况，然而此时 socket 并不用唤醒调用 `send()` 方法。由于请求还没被处理，所以不需要发回响应。这将消耗并浪费宝贵的 CPU 周期

服务端消息类

在消息切入点一节中，当通过 `process_events()` 知道 socket 事件就绪时我们可以看到消息对象是如何发出动作的。现在让我们来看看当数据在 socket 上被读取是会发生些什么，以及为服务器就绪的消息的组件片段发生了什么

`libserver.py` 文件中的服务端消息类，可以在 Github 上找到 源代码

这些方法按照消息处理顺序出现在类中

当服务器读取到至少两个字节时，定长头的逻辑就可以开始了

```
def process_protoheader(self):
    hdrlen = 2
    if len(self._recv_buffer) >= hdrlen:
        self._jsonheader_len = struct.unpack('>H',
                                             self._recv_buffer[:hdrlen])[0]
        self._recv_buffer = self._recv_buffer[hdrlen:]
```

网络字节序列中的定长整型两字节包含了 JSON 头的长度，`struct.unpack()` 方法用来读取并解码，然后保存在 `self._jsonheader_len` 中，当这部分消息被处理完成后，就要调用 `process_protoheader()` 方法来删除接收缓冲区中处理过的消息

就像上面的定长头的逻辑一样，当接收缓冲区有足够的 JSON 头数据时，它也需要被处理：

```
def process_jsonheader(self):
    hdrlen = self._jsonheader_len
    if len(self._recv_buffer) >= hdrlen:
        self.jsonheader = self._json_decode(self._recv_buffer[:hdrlen],
                                            'utf-8')
        self._recv_buffer = self._recv_buffer[hdrlen:]
        for reqhdr in ('byteorder', 'content-length', 'content-type',
                       'content-encoding'):
            if reqhdr not in self.jsonheader:
                raise ValueError(f'Missing required header "{reqhdr}".')
```

`self._json_decode()` 方法用来解码并反序列化 JSON 头成一个字典。由于我们定义的 JSON 头是 utf-8 格式的，所以解码方法调用时我们写死了这个参数，结果将被存放在 `self.jsonheader` 中，`process_jsonheader` 方法做完他应该做的事情后，同样需要删除接收缓冲区中处理过的消息

接下来就是真正的消息内容，当接收缓冲区有 JSON 头中定义的 `content-length` 值的数量个字节时，请求就应该被处理了：

```
def process_request(self):
    content_len = self.jsonheader['content-length']
    if not len(self._recv_buffer) >= content_len:
        return
    data = self._recv_buffer[:content_len]
    self._recv_buffer = self._recv_buffer[content_len:]
    if self.jsonheader['content-type'] == 'text/json':
        encoding = self.jsonheader['content-encoding']
        self.request = self._json_decode(data, encoding)
        print('received request', repr(self.request), 'from', self.addr)
    else:
        # Binary or unknown content-type
        self.request = data
        print(f'received {self.jsonheader["content-type"]} request from',
              self.addr)
    # Set selector to listen for write events, we're done reading.
    self._set_selector_events_mask('w')
```

把消息保存到 `data` 变量中后，`process_request()` 又会删除接收缓冲区中处理过的数据。接着，如果 content type 是 JSON 的话，它将解码并反序列化数据。否则（在我们的例子中）数据将被视 做二进制数据并打印出来

最后 `process_request()` 方法会修改 selector 为只监控写入事件。在服务端的程序 `app-server.py` 中，socket 初始化被设置成仅监控读事件。现在请求已经被全部处理完了，我们对读取事件就不感兴趣了

现在就可以创建一个响应写入到 socket 中。当 socket 可写时 `create_response()` 将被从 `write()` 方法中调用：

```
def create_response(self):
    if self.jsonheader['content-type'] == 'text/json':
        response = self._create_response_json_content()
    else:
        # Binary or unknown content-type
        response = self._create_response_binary_content()
    message = self._create_message(**response)
    self.response_created = True
    self._send_buffer += message
```

响应会根据不同的 content type 的不同而调用不同的方法创建。在这个例子中，当 `action == 'search'` 的时候会执行一个简单的字典查找。你可以在这个地方添加你自己的处理方法并调用

一个不好处理的问题是响应写入完成时如何关闭连接，我会在 `_write()` 方法中调用 `close()`

```
def _write(self):
    if self._send_buffer:
        print('sending', repr(self._send_buffer), 'to', self.addr)
        try:
            # Should be ready to write
            sent = self.sock.send(self._send_buffer)
        except BlockingIOError:
            # Resource temporarily unavailable (errno EWOULDBLOCK)
            pass
        else:
            self._send_buffer = self._send_buffer[sent:]
            # Close when the buffer is drained. The response has been sent.
            if sent and not self._send_buffer:
                self.close()
```

虽然 `close()` 方法的调用有点隐蔽，但是我认为这是一种权衡。因为消息类一个连接只处理一条消息。写入响应后，服务器无需执行任何操作。它的任务就完成了

客户端主程序

客户端主程序 `app-client.py` 中，参数从命令行中读取，用来创建请求并连接到服务端

```
$ ./app-client.py
usage: ./app-client.py <host> <port> <action> <value>
```

来个示例演示一下：

```
$ ./app-client.py 127.0.0.1 65432 search needle
```

当从命令行参数创建完一个字典来表示请求后，主机、端口、请求字典一起被传给 `start_connection()`

```
def start_connection(host, port, request):
    addr = (host, port)
    print('starting connection to', addr)
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setblocking(False)
    sock.connect_ex(addr)
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    message = libclient.Message(sel, sock, addr, request)
    sel.register(sock, events, data=message)
```

对服务器的 socket 连接被创建，消息对象被传入请求字典并创建

和服务端一样，消息对象在 `sel.register()` 方法中被关联到 socket 上。然而，客户端不同的是，socket 初始化的时候会监控读写事件，一旦请求被写入，我们将会修改为只监控读取事件

这种实现和服务端一样有好处：不浪费 CPU 生命周期。请求发送完成后，我们就不关注写入事件了，所以不用保持状态等待处理

客户端消息类

在 消息入口点 一节中，我们看到过，当 socket 使用准备就绪时，消息对象是如何调用具体动作的。现在让我们来看看 socket 上的数据是如何被读写的，以及消息准备好被加工的时候发生了什么

客户端消息类在 `libclient.py` 文件中，可以在 Github 上找到 源代码

这些方法按照消息处理顺序出现在类中

客户端的第一个任务就是让请求入队列：

```
def queue_request(self):
    content = self.request['content']
    content_type = self.request['type']
    content_encoding = self.request['encoding']
    if content_type == 'text/json':
        req = {
            'content_bytes': self._json_encode(content, content_encoding),
            'content_type': content_type,
            'content_encoding': content_encoding
        }
    else:
        req = {
            'content_bytes': content,
            'content_type': content_type,
            'content_encoding': content_encoding
        }
    message = self._create_message(**req)
    self._send_buffer += message
    self._request_queued = True
```

用来创建请求的字典，取决于客户端程序 `app-client.py` 中传入的命令行参数，当消息对象创建的时候，请求字典被当做参数传入

请求消息被创建并追加到发送缓冲区中，消息将被 `_write()` 方法发送，状态参数 `self._request_queued` 被设置，这使 `queue_request()` 方法不会被重复调用

请求发送完成后，客户端就等待服务器的响应

客户端读取和处理消息的方法和服务端一致，由于响应数据是从 socket 上读取的，所以处理 header 的方法会被调用：`process_protoheader()` 和 `process_jsonheader()`

最终处理方法名字的不同在于处理一个响应，而不是创建：`process_response()`，`_process_response_json_content()` 和 `_process_response_binary_content()`

最后，但肯定不是最不重要的 —— 最终的 `process_response()` 调用：

```
def process_response(self):
    # ...
    # Close when response has been processed
    self.close()
```

消息类的包装

我将通过提及一些方法的重要注意点来结束消息类的讨论

主程序中任意的类触发异常都由 `except` 字句来处理：

```
try:
    message.process_events(mask)
except Exception:
    print('main: error: exception for',
          f'{message.addr}:\n{traceback.format_exc()}')
    message.close()
```

注意最后一行的方法 `message.close()`

这一行很重要的原因有很多，不仅仅是保证 socket 被关闭，而且通过调用 `message.close()` 方法删除使用 `select()` 监控的 socket，这是类中的一段非常简洁的代码，它能减小复杂度。如果一个异常发生或者我们自己主动抛出，我们很清楚 `close()` 方法将处理善后

`Message._read()` 和 `Message._write()` 方法都包含一些有趣的东西：

```
def _read(self):
    try:
        # Should be ready to read
        data = self.sock.recv(4096)
    except BlockingIOError:
        # Resource temporarily unavailable (errno EWOULDBLOCK)
        pass
    else:
        if data:
            self._recv_buffer += data
        else:
            raise RuntimeError('Peer closed.')
```

注意 except 行：`except BlockingIOError` ：

`_write()` 方法也有，这几行很重要是因为它们捕获临时错误并通过使用 `pass` 跳过。临时错误是 socket 阻塞的时候发生的，比如等待网络响应或者连接的其它端

通过使用 `pass` 跳过异常，`select()` 方法将再次调用，我们将有机会重新读写数据

运行客户端 / 服务器应用程序

经过所有这些艰苦的工作后，让我们把程序运行起来并找到一些乐趣！

在这个救命中，我们将传一个空的字符串做为 `host` 参数的值，用来监听服务器端的所有 IP 地址。这样的话我就可以从其它网络上的虚拟机运行客户端程序，我将模拟一个 PowerPC 的机器

首先，把服务端程序运行进来：

```
$ ./app-server.py '' 65432
listening on ('', 65432)
```

现在让我们运行客户端，传入搜索内容，看看是否能看他（墨菲斯 - 黑客帝国中的角色）：

```
$ ./app-client.py 10.0.1.1 65432 search morpheus
starting connection to ('10.0.1.1', 65432)
sending b'\x00d{"byteorder": "big", "content-type": "text/json", "content-encoding": "
utf-8", "content-length": 41}{"action": "search", "value": "morpheus"}' to ('10.0.1.1'
, 65432)
received response {'result': 'Follow the white rabbit. ' } from ('10.0.1.1', 65432)
got result: Follow the white rabbit.
closing connection to ('10.0.1.1', 65432)
```

我的命令行 shell 使用了 utf-8 编码，所以上面的输出可以是 emojis

再试试看能不能搜索到小狗：

```
$ ./app-client.py 10.0.1.1 65432 search
starting connection to ('10.0.1.1', 65432)
sending b'\x00d{"byteorder": "big", "content-type": "text/json", "content-encoding": "
utf-8", "content-length": 37}{"action": "search", "value": "\xf0\x9f\x90\xb6"}' to ('1
0.0.1.1', 65432)
received response {'result': ' Playing ball! '  } from ('10.0.1.1', 65432)
got result:  Playing ball!
closing connection to ('10.0.1.1', 65432)
```

注意请求发送行的 byte string，很容易看出来你发送的小狗 emoji 表情被打印成了十六进制的字符串 `\xf0\x9f\x90\xb6`，我可以使用 emoji 表情来搜索是因为我的命令行支持 utf-8 格式的编码

这个示例中我们发送给网络原始的 bytes，这些 bytes 需要被接受者正确的解释。这就是为什么之前需要给消息附加头信息并且包含编码类型字段的原因

下面这个是服务器对应上面两个客户端连接的输出：

```
accepted connection from ('10.0.2.2', 55340)
received request {'action': 'search', 'value': 'morpheus'} from ('10.0.2.2', 55340)
sending b'\x00g{"byteorder": "little", "content-type": "text/json", "content-encoding"
: "utf-8", "content-length": 43}{"result": "Follow the white rabbit. \xf0\x9f\x90\xb0"
}' to ('10.0.2.2', 55340)
closing connection to ('10.0.2.2', 55340)

accepted connection from ('10.0.2.2', 55338)
received request {'action': 'search', 'value': '' } from ('10.0.2.2', 55338)
sending b'\x00g{"byteorder": "little", "content-type": "text/json", "content-encoding"
: "utf-8", "content-length": 37}{"result": "\xf0\x9f\x90\xbe Playing ball! \xf0\x9f\x8
f\x90"}' to ('10.0.2.2', 55338)
closing connection to ('10.0.2.2', 55338)
```

注意发送行中写到客户端的 bytes，这就是服务端的响应消息

如果 action 参数不是搜索，你也可以试试给服务器发送二进制请求

```
$ ./app-client.py 10.0.1.1 65432 binary 😀
starting connection to ('10.0.1.1', 65432)
sending b'\x00|{"byteorder": "big", "content-type": "binary/custom-client-binary-type"
, "content-encoding": "binary", "content-length": 10}binary\xf0\x9f\x98\x83' to ('10.0
.1.1', 65432)
received binary/custom-server-binary-type response from ('10.0.1.1', 65432)
got response: b'First 10 bytes of request: binary\xf0\x9f\x98\x83'
closing connection to ('10.0.1.1', 65432)
```

由于请求的 content-type 不是 `text/json`，服务器会把内容当成二进制类型并且不会解码 JSON，它只会打印 content-type 和返回的前 10 个 bytes 给客户端

```
$ ./app-server.py '' 65432
listening on ('', 65432)
accepted connection from ('10.0.2.2', 55320)
received binary/custom-client-binary-type request from ('10.0.2.2', 55320)
sending b'\x00\x7f{"byteorder": "little", "content-type": "binary/custom-server-binary
-type", "content-encoding": "binary", "content-length": 37}First 10 bytes of request:
binary\xf0\x9f\x98\x83' to ('10.0.2.2', 55320)
closing connection to ('10.0.2.2', 55320)
```

故障排查

某些东西运行不了是很常见的，你可能不知道应该怎么做，不用担心，所有人都会遇到这种问题，希望你借助本教程、调试器和万能的搜索引擎解决问题并且继续下去

如果还是解决不了，你的第一站应该是 python 的 socket 模块文档，确保你读过文档中每个我们使用到的方法、函数。同样的可以从引用一节中找到一些办法，尤其是错误一节中的内容

有的时候问题并不是由你的源代码引起的，源代码可能是正确的。有可能是不同的主机、客户端和服务器。也可能是网络原因，比如路由器、防火墙或者是其它网络设备扮演了中间人的角色

对于这些类型的问题，额外的一些工具是必要的。下面这些工具或者集可能会帮到你或者至少提供一些线索

pin

ping 命令通过发送一个 ICMP 报文来检测主机是否连接到了网络，它直接与操作系统上的 TCP/IP 协议栈通信，所以它在主机上是独立于任何应用程序运行的

下面是一段在 macOS 上执行 ping 命令的结果

```
$ ping -c 3 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.058 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.165 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.164 ms

--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.058/0.129/0.165/0.050 ms
```

注意后面的统计输出，这对你排查间歇性的连接问题很有帮助。比如说，是否有数据包丢失？网络延迟怎么样（查看消息的往返时间）

如果你与主机之间有防火墙的话，ping 发送的请求可能会被阻止。防火墙管理员定义了一些规则强制阻止一些请求，主要的原因就是他们不想自己的主机是可以被发现的。如果你的机器也出现这种情况的话，请确保在规则中添加了允许 ICMP 包的发送

ICMP 是 ping 命令使用的协议，但它也是 TCP 和其他底层用于传递错误消息的协议，如果你遇到奇怪的行为或缓慢的连接，可能就是这个原因

ICMP 消息通过类型和代号来定义。下面有一些重要的信息可以参考：

ICMP 类型	ICMP 代码	说明
8	0	打印请求
0	0	打印回复
3	0	目标网络不可达
3	1	目标主机不可达
3	2	目标协议不可达
3	3	目标端口不可达
3	4	需要分片，但是 DF(Don't fragmentation) 标识已被设置
11	0	网络存在环路

查看 Path MTU Discovery 更多关于分片和 ICMP 消息的内容，里面遇到的问题就是我前面提及的一些奇怪行为

netstat

在 查看 socket 状态 一节中我们已经知道如何使用 netstat 来查看 socket 及其状态的信息。这个命令在 macOS, Linux, Windows 上都可以使用

在之前的示例中我并没有提及 `Recv-Q` 和 `Send-Q` 列。这些列表示发送或者接收队列中网络缓冲区数据的字节数，但是由于某些原因这些字节还没被远程或者本地应用读写

换句话说，这些网络中的字节还在操作系统的队列中。一个原因可能是应用程序受 CPU 限制或者无法调用 `socket.recv()` 、 `socket.send()` 方法处理，或者因为其它一些网络原因导致的，比如说网络的拥堵、失败、硬件及电缆的问题

为了复现这个问题，看看到底在错误发生前我应该发送多少数据。我写了一个测试客户端可以连接到测试服务器，并且重复的调用 `socket.send()` 方法。测试服务端永远不调用 `socket.recv()` 或者 `socket.send()` 方法来处理客户端发送的数据，它只接受连接请求。这会导致服务器上的网络缓冲区被填满，最终会在客户端上报错

首先运行服务端：

```
$ ./app-server-test.py 127.0.0.1 65432 listening on ('127.0.0.1', 65432)
```

然后运行客户端，看看发生了什么：

```
$ ./app-client-test.py 127.0.0.1 65432 binary test
error: socket.send() blocking io exception for ('127.0.0.1', 65432):
BlockingIOError(35, 'Resource temporarily unavailable')
```

下面是用 `netstat` 命令在错误发生时执行的结果：

```
$ netstat -an | grep 65432
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4  408300      0  127.0.0.1.65432        127.0.0.1.53225        ESTABLISHED
tcp4       0 269868  127.0.0.1.53225        127.0.0.1.65432        ESTABLISHED
tcp4       0      0  127.0.0.1.65432        *.*                    LISTEN
```

第一行就表示服务端（本地端口是 65432）

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4  408300      0  127.0.0.1.65432        127.0.0.1.53225        ESTABLISHED
```

注意 Recv-Q: 408300

第二行表示客户端（远程端口是 65432）

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0 269868  127.0.0.1.53225        127.0.0.1.65432        ESTABLISHED
```

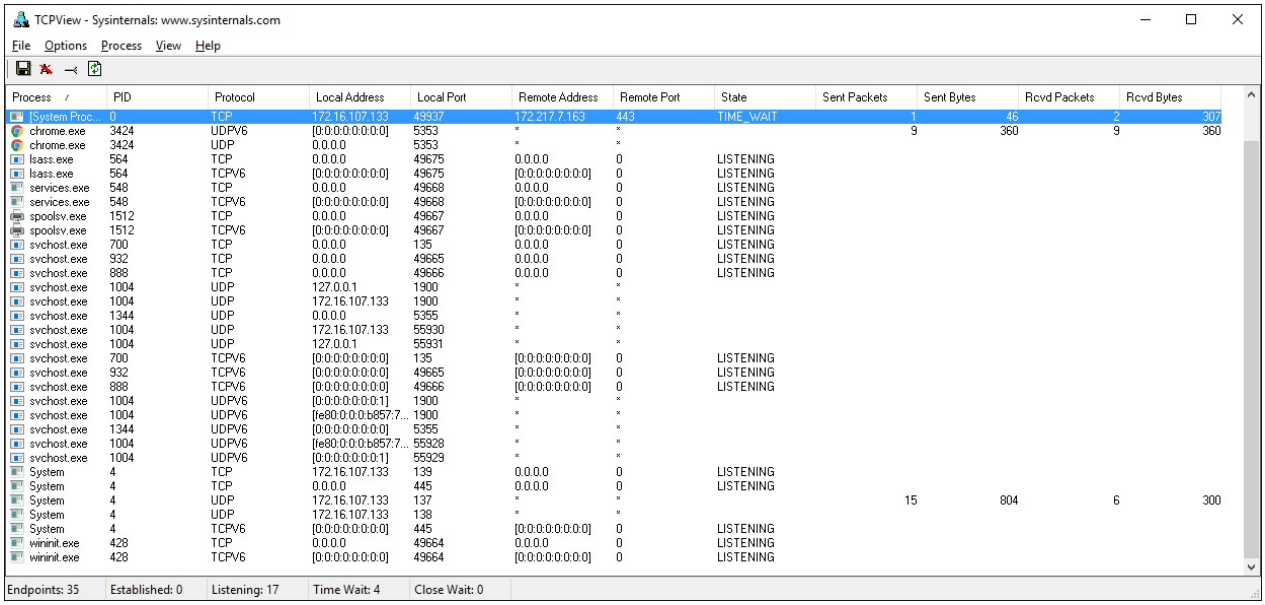
注意 Send-Q: 269868

显然，客户端试着写入字节，但是服务端并没有读取他们。这导致服务端网络缓冲队列中应该保存的数据被积压在接收端，客户端的网络缓冲队列积压到发送端

windows

如果你使用的是 windows 电脑，有一个工具套件绝对值得安装 Windows Sysinternals

里面有个工具叫 `TCPView.exe`，它是 windows 下的一个可视化的 netstat 工具。除了地址、端口号和 socket 状态之外，它还会显示发送和接收的数据包以及字节数。就像 Unix 工具集 `lsof` 命令一样，你也可以看见进程名和 ID，可以在菜单中查看更多选项



Wireshark

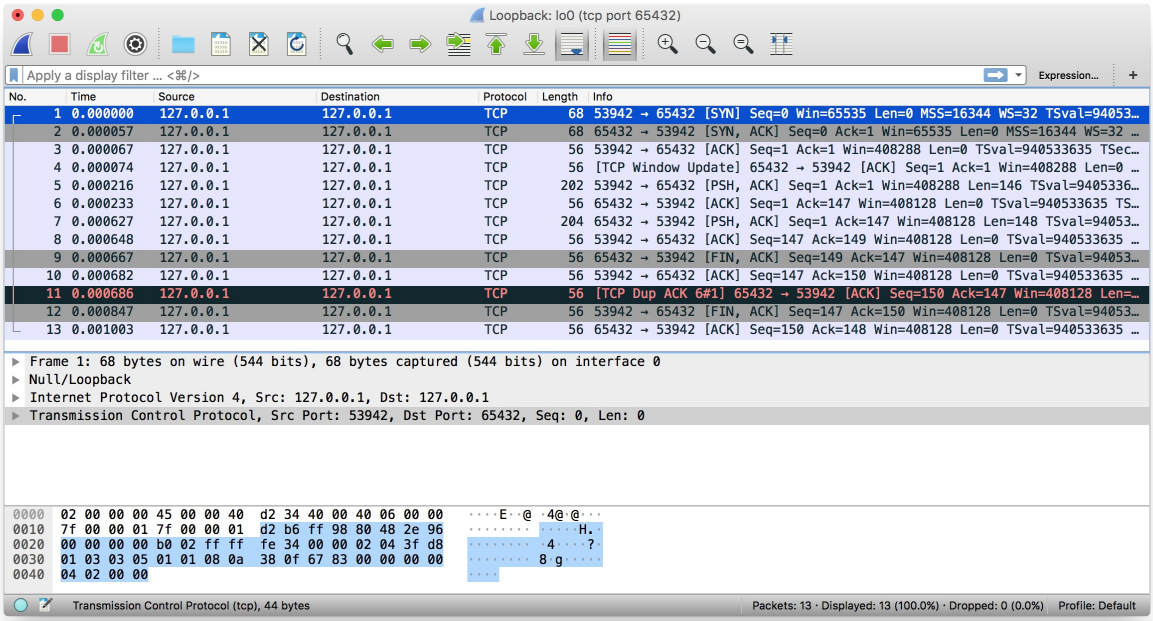
有时候你可能想查看网络底层发生了什么，忽略应用程序的输出或者外部库调用，想看看网络层面到底收发了什么内容，就像调试器一样，当你需要看清这些的时候，没有别的办法

Wireshark 是一款可以运行在 macOS, Linux, Windows 以及其它系统上的网络协议分析、流量捕获工具，GUI 版本的程序叫做 `wireshark`，命令 行的程序叫做 `tshark`

流量捕获是一个非常好用的方法，它可以让你看到网络上应用程序的行为，收集到关于收发消息多少、频率等信息，你也可以看到客户端或者服务端如何关闭 / 取消连接，或者停止响应，当你需要排除故障的时候这些信息非常的有用

网上还有很多关于 `wireshark` 和 `TShark` 的基础使用教程

这有一个使用 wireshark 捕获本地网络数据的例子：



还有一个和上面一样的使用 tshark 命令输出的结果：

```
$ tshark -i lo0 'tcp port 65432'
Capturing on 'Loopback'
    1   0.000000    127.0.0.1 → 127.0.0.1    TCP 68 53942 → 65432 [SYN] Seq=0 Win=6553
5 Len=0 MSS=16344 WS=32 TSval=940533635 TSecr=0 SACK_PERM=1
    2   0.000057    127.0.0.1 → 127.0.0.1    TCP 68 65432 → 53942 [SYN, ACK] Seq=0 Ack
=1 Win=65535 Len=0 MSS=16344 WS=32 TSval=940533635 TSecr=940533635 SACK_PERM=1
    3   0.000068    127.0.0.1 → 127.0.0.1    TCP 56 53942 → 65432 [ACK] Seq=1 Ack=1 Win
=408288 Len=0 TSval=940533635 TSecr=940533635
    4   0.000075    127.0.0.1 → 127.0.0.1    TCP 56 [TCP Window Update] 65432 → 53942
[ACK] Seq=1 Ack=1 Win=408288 Len=0 TSval=940533635 TSecr=940533635
    5   0.000216    127.0.0.1 → 127.0.0.1    TCP 202 53942 → 65432 [PSH, ACK] Seq=1 Ac
k=1 Win=408288 Len=146 TSval=940533635 TSecr=940533635
    6   0.000234    127.0.0.1 → 127.0.0.1    TCP 56 65432 → 53942 [ACK] Seq=1 Ack=147 W
in=408128 Len=0 TSval=940533635 TSecr=940533635
    7   0.000627    127.0.0.1 → 127.0.0.1    TCP 204 65432 → 53942 [PSH, ACK] Seq=1 Ac
k=147 Win=408128 Len=148 TSval=940533635 TSecr=940533635
    8   0.000649    127.0.0.1 → 127.0.0.1    TCP 56 53942 → 65432 [ACK] Seq=147 Ack=14
9 Win=408128 Len=0 TSval=940533635 TSecr=940533635
    9   0.000668    127.0.0.1 → 127.0.0.1    TCP 56 65432 → 53942 [FIN, ACK] Seq=149 A
ck=147 Win=408128 Len=0 TSval=940533635 TSecr=940533635
   10   0.000682    127.0.0.1 → 127.0.0.1    TCP 56 53942 → 65432 [ACK] Seq=147 Ack=15
0 Win=408128 Len=0 TSval=940533635 TSecr=940533635
   11   0.000687    127.0.0.1 → 127.0.0.1    TCP 56 [TCP Dup ACK 6#1] 65432 → 53942 [A
CK] Seq=150 Ack=147 Win=408128 Len=0 TSval=940533635 TSecr=940533635
   12   0.000848    127.0.0.1 → 127.0.0.1    TCP 56 53942 → 65432 [FIN, ACK] Seq=147 A
ck=150 Win=408128 Len=0 TSval=940533635 TSecr=940533635
   13   0.001004    127.0.0.1 → 127.0.0.1    TCP 56 65432 → 53942 [ACK] Seq=150 Ack=14
8 Win=408128 Len=0 TSval=940533635 TSecr=940533635
^C13 packets captured
```

引用

这一节主要用来引用一些额外的信息和外部资源链接

Python 文档

- Python's socket module
- Python's Socket Programming HOWTO

错误信息

下面这段话来自 python 的 socket 模块文档：

> 所有的错误都会触发异常，像无效参数类型和内存不足的常见异常可以被抛出；从 Python 3.3 开始，与 socket 或地址语义相关的错误会引发 OSError 或其子类之一的异常
> 引用

异常	`errno` 常量	说明
BlockingIOError	EWOULDBLOCK	资源暂不可用，比如在非阻塞模式下调用 `send()` 方法，对方太繁忙面没有读取，发送队列满了，或者网络有问题
OSError	EADDRINUSE	端口被战用，确保没有其它的进程与当前的程序运行在同一地址 / 端口上，你的服务器设置了 `SO_REUSEADDR` 参数
ConnectionResetError	ECONNRESET	连接被重置，远端的进程崩溃，或者 socket 意外关闭，或是有防火墙或链路上的设配有问题
TimeoutError	ETIMEDOUT	操作超时，对方没有响应
ConnectionRefusedError	ECONNREFUSED	连接被拒绝，没有程序监听指定的端口

socket 地址族

`socket.AF_INET` 和 `socket.AF_INET6` 是 `socket.socket()` 方法调用的第一个参数，表示地址协议族，API 使用了一个期望传入指定格式参数的地址，这取决于是 `AF_INET` 还是 `AF_INET6`

地址族	协议	地址元组	说明
socket.AF_INET	IPv4	(host, port)	host 参数是个如 `www.example.com` 的主机名称，或者如 `10.1.2.3` 的 IPv4 地址
socket.AF_INET6	IPv6	(host, port, flowinfo, scopeid)	主机名同上，IPv6 地址如： `fe80::6203:7ab:fe88:9c23` ，flowinfo 和 scopeid 分别表示 C 语言结构体 `sockaddr_in6` 中的 `sin6_flowinfo` 和 `sin6_scope_id` 成员

注意下面这段 python socket 模块中关于 host 值和地址元组文档

> 对于 IPv4 地址，使用主机地址的方式有两种： `''` 空字符串表示 `INADDR_ANY` ，字符 `'<broadcast>'` 表示 `INADDR_BROADCAST` ，这个行为和 IPv6 不兼容，因此如果你的程序中使用的是 IPv6 就应该避免这种做法。源文档

我在本教程中使用了 IPv4 地址，但是如果你的机器支持，也可以试试 IPv6 地址。 `socket.getaddrinfo()` 方法会返回五个元组的序列，这包括所有创建 socket 连接的必要参数， `socket.getaddrinfo()` 方法理解并处理传入的 IPv6 地址和主机名

下面的例子中程序将返回一个通过 TCP 连接到 `example.org` 80 端口上的地址信息：

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('93.184.216.34', 80))]
```

如果 IPv6 可用的话结果可能有所不同，上面返回的值可以被用于 `socket.socket()` 和 `socket.connect()` 方法调用的参数，在 python socket 模块文档中的 示例 一节中有客户端和服务端程序

使用主机名

这一节主要适用于使用 `bind()` 和 `connect()` 或 `connect_ex()` 方法时如何使用主机名，然而当你使用回环地址做为主机名时，它总是会解析到你期望的地址。这刚好与客户端使用主机名的场景相反，它需要 DNS 解析的过程，比如 `www.example.com`

下面一段来自 python socket 模块文档

> 如果你主机名称做为 IPv4/v6 socket 地址的 host 部分，程序可能会出现非预期的结果，由于 python 使用了 DNS 查找过程中的第一个结果，socket 地址会被解析成与真正的 IPv4/v6 地址不同的其它地址，这取决于 DNS 解析和你的 host 文件配置。如果想得到确定的结果，请使用数字格式的地址做为 host 参数的值 源文档

通常回环地址 `localhost` 会被解析到 `127.0.0.1` 或 `::1` 上，你的系统可能就是这么设置的，也可能不是。这取决于你系统配置，与所有 IT 相关的事情一样，总会有例外的情况，没办法完全保证 localhost 被解析到了回环地址上

比如在 Linux 上，查看 `man nsswitch.conf` 的结果，域名切换配置文件，还有另外一个 macOS 和 Linux 通用的配置文件地址是：`/etc/hosts`，在 windows 上则是 `C:\Windows\System32\drivers\etc\hosts`，hosts 文件包含了一个文本格式的静态域名地址映射表，总之 DNS 也是一个难题

有趣的是，在撰写这篇文章的时候（2018 年 6 月），有一个关于 让 localhost 成为真正的 localhost 的 RFC 草案，讨论就是围绕着 localhost 使用的情况开展的

最重要的一点是你要理解当你在应用程序中使用主机名时，返回的地址可能是任何东西，如果你有一个安全性敏感的应用程序，不要使用主机名。取决于你的应用程序和环境，这可能会困扰到你

注意：安全方面的考虑和最佳实践总是好的，即使你的程序不是安全敏感型的应用。如果你的应用程序访问了网络，那它就应该是安全的稳定的。这表示至少要做到以下几点：

- 经常会有系统软件升级和安全补丁，包括 python，你是否使用了第三方的库？如果是的话，确保他们能正常工作并且更新到了新版本
- 尽量使用专用防火墙或基于主机的防火墙来限制与受信任系统的连接
- DNS 服务是如何配置的？你是否信任配置内容及其配置者
- 在调用处理其他代码之前，请确保尽可能地对请求数据进行了清理和验证，还要为此添加测试用例，并且经常运行

无论是否使用主机名称，你的应用程序都需要支持安全连接（加密授权），你可能会用到 TLS，这是一个超越了本教程的范围的话题。可以从 python 的 SSL 模块文档了解如何开始使用它，这个协议和你的浏览器使用的安全协议是一样的

考虑到接口、IP 地址、域名解析这些「变量」，你应该怎么应对？如果你还没有网络应用程序审查流程，可以使用以下建议：

应用程序	使用	建议
服务端	回环地址	使用 IP 地址 127.0.0.1 或 ::1
服务端	以太网地址	使用 IP 地址，比如：10.1.2.3，使用空字符串表示本机所有 IP 地址
客户端	回环地址	使用 IP 地址 127.0.0.1 或 ::1
客户端	以太网地址	使用统一的不依赖域名解析的 IP 地址，特殊情况下才会使用主机地址，查看上面的安全提示

对于客户端或者服务端来说，如果你需要授权连接到主机，请查看如何使用 TLS

阻塞调用

如果一个 socket 函数或者方法使你的程序挂起，那么这个就是个阻塞调用，比如 accept(), connect(), send(), 和 recv() 都是 阻塞 的，它们不会立即返回，阻塞调用在返回前必须等待系统调用 (I/O) 完成。所以调用者 —— 你，会被阻止直到系统调用结束或者超过延迟时间或者有错误发生

阻塞的 socket 调用可以设置成非阻塞的模式，这样他们就可以立即返回。如果你想做到这一点，就得重构并重新设计你的应用程序

由于调用直接返回了，但是数据确没就绪，被调用者处于等待网络响应的状态，没法完成它的工作，这种情况下，当前 socket 的状态码 errno 应该是 `socket.EWOULDBLOCK` 。`setblocking()` 方法是支持非阻塞模式的

默认情况下，socket 会以阻塞模式创建，查看 socket 延迟的注意事项 中三种模式的解释

关闭连接

有趣的是 TCP 连接一端打开，另一端关闭的状态是完全合法的，这被称做 TCP「半连接」，是否需要这种保持状态是由应用程序决定的，通常来说不需要。这种状态下，关闭方将不能发送任何数据，它只能接收数据

我不是在提倡你采用这种方法，但是作为一个例子，HTTP 使用了一个名为「Connection」的头来标准化规定应用程序是否关闭或者保持连接状态，更多内容请查看 RFC 7230 中 6.3 节，HTTP 协议 (HTTP/1.1): 消息语法与路由

当你在设计应用程序及其应用层协议的时候，最好先了解一下如何关闭连接，有时这很简单而且很明显，或者采取一些可以实现的原型，这取决于你的应用程序以及消息循环如何被处理成期望的数据，只要确保 socket 在完成工作后总是能正确关闭

字节序

查看维基百科 字节序 中关于不同的 CPU 是如何在内存中存储字节序列的，处理单个字节时没有任何问题，但是当把多个字节处理成单个值（四字节整型）时，如果和你通信的另一端使用了不同的字节序时字节顺序需要被反转

字节顺序对于字符文本来说也很重要，字符文本通过表示为多字节的序列，就像 Unicode 一样。除非你只使用 `true` 和 ASCII 字符来控制客户端和服务端的实现，否则使用 utf-8 格式或者支持字节序标识 (BOM) 的 Unicode 字符集会比较合适

在应用层协议中明确的规定使用编码格式是很重要的，你可以规定所有的文本都使用 utf-8 或者用「content-encoding」头指定编码格式，这将使你的程序不需要检测编码方式，当然也应该尽量避免这么做

当数据被调用存储到了文件或者数据库中而且又没有数据的元信息的时候，问题就很麻烦了，当数据被传到其它端，它将试着检测数据的编码方式。有关讨论，请参阅 Wikipedia 的 Unicode 文章，它引用了 RFC 3629:UTF-8, a transformation format of ISO 10646

> 然而 UTF-8 的标准 RFC 3629 中推荐禁止在 UTF-8 协议中使用标记字节序 (BOM)，但是讨论了无法实现的情况，最大的问题在于如何使用一种模式在不依赖 BOM 的情况下区分 UTF-8 和其它编码方式

避开这些问题的方法就是总是存储数据使用的编码方式，换句话说，如果不只用 utf-8 格式的编码或者其它的带有 BOM 的编码就要尝试以某种方式将编码方式存储为元数据，然后你就可以在数据上附加编码的头信息，告诉接收者编码方式

TCP/IP 使用的字节顺序是 big-endian，被称做网络序。网络序被用来表示底层协议栈中的整型数字，好比 IP 地址和端口号，python 的 socket 模块有几个函数可以把这种整型数字从网络字节序转换成主机字节序

函数	说明
socket.ntohl(x)	把 32 位的正整型数字从网络字节序转换成主机字节序，在网络字节序和主机字节序相同的机器上这是个空操作，否则将是一个 4 字节的交换操作
socket.ntohs(x)	把 16 位的正整型数字从网络字节序转换成主机字节序，在网络字节序和主机字节序相同的机器上这是个空操作，否则将是一个 2 字节的交换操作
socket.htonl(x)	把 32 位的正整型数字从主机字节序转换成网络字节序，在网络字节序和主机字节序相同的机器上这是个空操作，否则将是一个 4 字节的交换操作
socket.htons(x)	把 16 位的正整型数字从主机字节序转换成网络字节序，在网络字节序和主机字节序相同的机器上这是个空操作，否则将是一个 2 字节的交换操作

你也可以使用 struct 模块打包或者解包二进制数据（使用格式化字符串）：

```
import struct
network_byteorder_int = struct.pack('>H', 256)
python_int = struct.unpack('>H', network_byteorder_int)[0]
```

结语

我们在本教程中介绍了很多内容，网络和 socket 是很大的一个主题，如果你对它们都比较陌生，不要被这些规则和大写字母术语吓到

为了理解所有的东西如何工作的，有很多部分需要了解。但是，就像 python 一样，当你花时间去了解每个独立的部分时它才开始变得有意义

我们看过了 python socket 模块中底层的一些 API，并了解了如何使用它们创建客户端服务器应用程序。我们也创建了一个自定义类来做为应用层的协议，并用它在不同的端点之间交换数据，你可以使用这个类并在些基础上快速且简单地构建出一个你自己的 socket 应用程序

你可以在 Github 上找到 源代码

恭喜你坚持到最后！你现在就可以在程序中很好地使用 socket 了

我希望这个教程能为你开始 socket 编程旅途中提供一些信息、示例、或者灵感