套接字编程指南

发布 *3.9.0a3*

Guido van Rossum
and the Python development team

二月 25, 2020

Python Software Foundation
Email: docs@python.org

Contents

1 套接字 **2**
1.1 历史 2

2 创建套接字 **2**
2.1 进程间通信 3

3 使用一个套接字 **3**
3.1 二进制数据 5

4 断开连接 **5**
4.1 套接字何时销毁 5

5 非阻塞的套接字 **5**

作者 Gordon McMillan

摘要

套接字几乎无处不在，但是它却是被误解最严重的技术之一。这是一篇简单的套接字概述。并不是一篇真正的教程——你需要做更多的事情才能让它工作起来。其中也并没有涵盖细节（细节会有很多），但是我希望它能提供足够的背景知识，让你像模像样的开始使用套接字

1 套接字

我将只讨论关于 INET（比如：IPv4 地址族）的套接字，但是它将覆盖几乎 99% 的套接字使用场景。并且我将仅讨论 STREAM（比如：TCP）类型的套接字 - 除非你真的知道你在做什么（那么这篇 HOWTO 可能并不适合你），使用 STREAM 类型的套接字将会得到比其它类型更好的表现与性能。我将尝试揭开套接字的神秘面纱，也会讲到一些阻塞与非阻塞套接字的使用。但是我将以阻塞套接字为起点开始讨论。只有你了解它是如何工作的以后才能处理非阻塞套接字。

理解这些东西的难点之一在于「套接字」可以表示很多微妙差异的东西，这取决于上下文。所以首先，让我们先分清楚「客户端」套接字和「服务端」套接字之间的不同，客户端套接字表示对话的一端，服务端套接字更像是总机接线员。客户端程序只能（比如：你的浏览器）使用「客户端」套接字；网络服务器则可以使用「服务端」套接字和「客户端」套接字来会话

1.1 历史

目前为止，在各种形式的 IPC (进程间通信) 中，套接字是最流行的。在任何指定的平台上，可能会有其它更快的 IPC 形式，但是就跨平台通信来说，套接字大概是唯一的玩法

套接字做为 BSD Unix 操作系统的一部分在伯克利诞生，像野火一样在因特网传播。有一个很好的原因——套接字与 INET 的结合使得与世界各地的任意机器间通信变得令人难以置信的简单（至少对比与其他方案来说）

2 创建套接字

简略地说，当你点击带你来到这个页面的链接时，你的浏览器就已经做了下面这几件事情:

```
# create an INET, STREAMing socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# now connect to the web server on port 80 - the normal http port
s.connect(("www.python.org", 80))
```

当连接完成，套接字可以用来发送请求来接收页面上显示的文字。同样是这个套接字也会用来读取响应，最后再被销毁。是的，被销毁了。客户端套接字通常用来做一次交换（或者说一小组序列的交换）。

网络服务器发生了什么这个问题就有点复杂了。首页，服务器创建一个「服务端套接字」:

```
# create an INET, STREAMing socket
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# bind the socket to a public host, and a well-known port
serversocket.bind((socket.gethostname(), 80))
# become a server socket
serversocket.listen(5)
```

有几件事需要注意：我们使用了 `socket.gethostname()`，所以套接字将外网可见。如果我们使用的是 `s.bind(('localhost', 80))` 或者 `s.bind(('127.0.0.1', 80))`，也会得到一个「服务端」套接字，但是后者只在同一机器上可见。`s.bind(('', 80))` 则指定套接字可以被机器上的任何地址碰巧连接

第二个需要注点是：低端口号通常被一些「常用的」服务（HTTP, SNMP 等）所保留。如果你想把程序跑起来，最好使用一个高位端口号（通常是 4 位的数字）。

最后，`listen` 方法的参数会告诉套接字库，我们希望在队列中累积多达 5 个（通常的最大值）连接请求后再拒绝外部连接。如果所有其他代码都准确无误，这个队列长度应该是足够的。

现在我们已经有一个「服务端」套接字，监听了 80 端口，我们可以进入网络服务器的主循环了:

```
while True:
    # accept connections from outside
    (clientsocket, address) = serversocket.accept()
    # now do something with the clientsocket
    # in this case, we'll pretend this is a threaded server
    ct = client_thread(clientsocket)
    ct.run()
```

事际上，通常有 3 种方法可以让这个循环工作起来 - 调度一个线程来处理 客户端套接字，或者把这个应用改成使用非阻塞模式套接字，亦或是使用 `select` 库来实现「服务端」套接字与任意活动 客户端套接字之间的多路复用。稍后会详细介绍。现在最重要的是理解：这就是一个 服务端套接字做的 所有事情。它并没有发送任何数据。也没有接收任何数据。它只创建「客户端」套接字。每个 客户端套接字都是为了响应某些其它客户端套接字 `connect()` 到我们绑定的主机。一旦创建 客户端套接字完成，就会返回并监听更多的连接请求。现个客户端可以随意通信 - 它们使用了一些动态分配的端口，会话结束时端口才会被回收

2.1 进程间通信

如果你需要在同一台机器上进行两个进程间的快速 IPC 通信，你应该了解管道或者共享内存。如果你决定使用 AF_INET 类型的套接字，绑定「服务端」套接字到 `'localhost'` 。在大多数平台，这将会使用一个许多网络层间的通用快捷方式（本地回环地址）并且速度会快很多

参见:

`multiprocessing` 模块使跨平台 IPC 通信成为一个高层的 API

3 使用一个套接字

首先需要注意，浏览器的「客户端」套接字和网络服务器的「客户端」套接字是极为相似的。即这种会话是「点对点」的。或者也可以说 你作为设计师需要自行决定会话的规则和礼节。通常情况下，连接套接字通过发送一个请求或者信号来开始一次会话。但这属于设计决定，并不是套接字规则。

现在有两组用于通信的动词。你可以使用 `send` 和 `recv` ，或者你可以把客户端套接字改成文件类型的形式来使用 `read` 和 `write` 方法。后者是 Java 语言中表示套接字的方法，我将不会在这儿讨论这个，但是要提醒你需要调用套接字的 `flush` 方法。这些是“缓冲”的文件，一个经常出现的错误是 `write` 一些东西，然后就直接开始 `read` 一个响应。如果不调用 `flush` ，你可能会一直等待这个响应，因为请求可能还在你的输出缓冲中。

现在我来到了套接字的两个主要的绊脚石 - `send` 和 `recv` 操作网络缓冲区。它们并不一定可以处理所有你想要（期望）的字节，因为它们主要关注点是处理网络缓冲。通常，它们在关联的网络缓冲区 `send` 或者清空 `recv` 时返回。然后告诉你处理了多少个字节。你的责任是一直调用它们直到你所有的消息处理完成。

当 `recv` 方法返回 0 字节时，就表示另一端已经关闭（或者它所在的进程关闭）了连接。你再也不能从这个连接上获取到任何数据了。你可以成功的发送数据；我将在后面讨论这一点。

像 HTTP 这样的协议只使用一个套接字进行一次传输。客户端发送一个请求，然后读取响应。就这么简单。套接字会被销毁。表示客户端可以通过接收 0 字节序列表示检测到响应的结束。

但是如果你打算在随后来的传输中复用套接字的话，你需要明白 套接字里面是不存在*:abbr:'EOT (传输结束)'* 的。重复一下：套接字 `send` 或者 `recv` 完 0 字节后返回，连接会中断。如果连接没有被断开，你可能会永远处于等待 `recv` 的状态，因为（就目前来说）套接字 不会告诉你不用再读取了。现在如果你细心一点，你可能会意识到套接字基本事实：消息必须要么具有固定长度，要么可以界定，要么指定了长度（比较好的做法），要么以关闭连接为结束。选择完全由你而定（这比让别人定更合理）。

假定你不希望结束连接，那么最简单的解决方案就是使用定长消息:

```
class MySocket:
    """demonstration class only
      - coded for clarity, not efficiency
    """

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
                            socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
        self.sock.connect((host, port))

    def mysend(self, msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError("socket connection broken")
            totalsent = totalsent + sent

    def myreceive(self):
        chunks = []
        bytes_recd = 0
        while bytes_recd < MSGLEN:
            chunk = self.sock.recv(min(MSGLEN - bytes_recd, 2048))
            if chunk == b'':
                raise RuntimeError("socket connection broken")
            chunks.append(chunk)
            bytes_recd = bytes_recd + len(chunk)
        return b''.join(chunks)
```

发送分部代码几乎可用于任何消息传递方案——在 Python 中你发送字符串，可以使用 `len()` 方法来确定它的长度（即使它嵌入了 `\0` 字符），主要是接收代码变得更复杂。（在 C 语言中，并没有更糟糕，除非消息嵌入了 `\0` 字符而且你又无法使用 `strlen`）

最简单的改进是让消息的第一个字符表示消息类型，由类型决定长度。现在你需要两次 `recv`- 第一次取（至少）第一个字符来知晓长度，第二次在循环中获取剩余所有的消息。如果你决定到分界线，你将收到一些任意大小的块，（4096 或者 8192 通常是比较合适的网络缓冲区大小），扫描你接收到的分界符

一个需要意识到的复杂情况是：如果你的会话协议允许多个消息被发送回来（没有响应），调用 `recv` 传入任意大小的块，你可能会因为读到后续接收的消息而停止读取。你需要将它放在一边并保存，直到它需要为止。

以其长度（例如，作为 5 个数字字符）作为消息前缀时会变得更复杂，因为（信不信由你）你可能无法在一个 `recv` 中获得所有 5 个字符。在一般使用时，你会侥幸避免该状况；但是在高网络负载中，除非你使用两个 `recv` 循环，否则你的代码将很快中断——第一个用于确定长度，第二个用于获取消息的数据部分。这很讨厌。当你发现 `send` 并不总是设法在支持搞定一切时，你也会有这种感觉。尽管已经阅读过这篇文章，但最终还是会有所了解!

限于篇幅，建立你的角色，（保持与我的竞争位置），这些改进将留给读者做为练习。现在让我们继续。

3.1 二进制数据

通过套接字传送二进制数据是可行的。主要问题在于并非所有机器都用同样的二进制数据格式。比如 Motorola 芯片用两个十六进制字节 00 01 来表示一个 16 位整数值 1。而 Intel 和 DEC 则会做字节反转——即用 01 00 来表示 1。套接字库要求转换 16 位和 32 位整数——`ntohl, htonl, ntohs, htons` 其中的「n」表示 *network*，「h」表示 *host*，「s」表示 *short*，「l」表示 *long*。在网络序列就是主机序列时它们什么都不做，但是如果机器是字节反转的则会适当地交换字节序。

在现今的 32 位机器中，二进制数据的 ascii 表示往往比二进制表示要小。这是因为在非常多的时候所有 long 的值均为 0 或者 1。字符串形式的"0" 为两个字节，而二进制形式则为四个。当然这不适用于固定长度的信息。自行决定，请自行决定。

4 断开连接

严格地讲，你应该在 `close` 它之前将套接字 `shutdown` 。`shutdown` 是发送给套接字另一端的一种建议。调用时参数不同意义也不一样，它可能意味着「我不会再发送了，但我仍然会监听」，或者「我没有监听了，真棒!」。然而，大多数套接字库或者程序员都习惯了忽略使用这种礼节，因为通常情况下 `close` 与 `shutdown(); close()` 是一样的。所以在大多数情况下，不需要显式的 `shutdown` 。

高效使用 `shutdown` 的一种方法是在类似 HTTP 的交换中。客户端发送请求，然后执行 `shutdown(1)` 。这告诉服务器“此客户端已完成发送，但仍可以接收”。服务器可以通过接收 0 字节来检测“EOF”。它可以假设它有完整的请求。服务器发送回复。如果 `send` 成功完成，那么客户端仍在接收。

Python 进一步自动关闭，并说当一个套接字被垃圾收集时，如果需要它会自动执行 `close` 。但依靠这个机制是一个非常坏的习惯。如果你的套接字在没有 `close` 的情况下就消失了，那么另一端的套接字可能会无限期地挂起，以为你只是慢了一步。完成后 请 `close` 你的套接字。

4.1 套接字何时销毁

使用阻塞套接字最糟糕的事情可能就是当另一边下线时（没有 `close` ）会发生什么。你的套接字可能会挂起。TCP 是一种可靠的协议，它会在放弃连接之前等待很长时间。如果你正在使用线程，那么整个线程基本上已经死了。你无能为力。只要你没有做一些愚蠢的事情，比如在进行阻塞读取时持有一个锁，那么线程并没有真正消耗掉资源。不要尝试杀死线程——线程比进程更有效的部分原因是它们避免了与自动回收资源相关的开销。换句话说，如果你设法杀死线程，你的整个进程很可能被搞坏。

5 非阻塞的套接字

如果你已理解上述内容，那么你已经了解了使用套接字的机制所需了解的大部分内容。你仍将以相同的方式使用相同的函数调用。只是，如果你做得对，你的应用程序几乎是由内到外的。

In Python, you use `socket.setblocking(False)` to make it non-blocking. In C, it's more complex, (for one thing, you'll need to choose between the BSD flavor `O_NONBLOCK` and the almost indistinguishable Posix flavor `O_NDELAY`, which is completely different from `TCP_NODELAY`), but it's the exact same idea. You do this after creating the socket, but before using it. (Actually, if you're nuts, you can switch back and forth.)

主要的机制差异是 `send` 、`recv` 、`connect` 和 `accept` 可以在没有做任何事情的情况下返回。你（当然）有很多选择。你可以检查返回代码和错误代码，通常会让自己发疯。如果你不相信我，请尝试一下。你的应用程序将变得越来越大、越来越 Bug 、吸干 CPU。因此，让我们跳过脑死亡的解决方案并做正确的事。

使用 `select` 库

在 C 中，编码 `select` 相当复杂。在 Python 中，它是很简单，但它与 C 版本足够接近，如果你在 Python 中理解 `select` ，那么在 C 中你会几乎不会遇到麻烦:

```
ready_to_read, ready_to_write, in_error = \
               select.select(
                  potential_readers,
                  potential_writers,
                  potential_errs,
                  timeout)
```

你传递给 select 三个列表：第一个包含你可能想要尝试读取的所有套接字；第二个是你可能想要尝试写入的所有套接字，以及要检查错误的最后一个（通常为空）。你应该注意，套接字可以进入多个列表。select 调用是阻塞的，但你可以给它一个超时。这通常是一件明智的事情——给它一个很长的超时（比如一分钟），除非你有充分的理由不这样做。

作为返回，你将获得三个列表。它们包含实际可读、可写和有错误的套接字。这些列表中的每一个都是你传入的相应列表的子集（可能为空）。

如果一个套接字在输出可读列表中，那么你可以像我们一样接近这个业务，那个套接字上的 recv 将返回 一些内容。可写列表的也相同，你将能够发送 一些内容。也许不是你想要的全部，但 有些东西比没有东西更好。（实际上，任何合理健康的套接字都将以可写方式返回——它只是意味着出站网络缓冲区空间可用。）

如果你有一个“服务器”套接字，请将其放在 potential_readers 列表中。如果它出现在可读列表中，那么你的 accept （几乎肯定）会起作用。如果你已经创建了一个新的套接字 connect 其他人，请将它放在 potential_writers 列表中。如果它出现在可写列表中，那么它有可能已连接。

实际上，即使使用阻塞套接字，select 也很方便。这是确定是否阻塞的一种方法——当缓冲区中存在某些内容时，套接字返回为可读。然而，这仍然无助于确定另一端是否完成或者只是忙于其他事情的问题。

可移植性警告：在 Unix 上，select 适用于套接字和文件。不要在 Windows 上尝试。在 Windows 上，select 仅适用于套接字。另请注意，在 C 中，许多更高级的套接字选项在 Windows 上的执行方式不同。事实上，在 Windows 上我通常在使用我的套接字使用线程（非常非常好）。