Socket 程式設計指南

發🄵 *3.11.11*

Guido van Rossum and the Python development team

12 月 07, 2024

Python Software Foundation
Email: docs@python.org

Contents

作者
Gordon McMillan

摘要

Sockets 在各處都被廣泛使用，但🄵是一項被誤解最嚴重的技術之一。這是一篇對 sockets 的概論介紹。這🄵不是一個完整的教學指南 - 你還需要做許多准備才能讓 sockets 正常運作。這篇文章也🄵有包含細節（其中有非常多的細節），但我希望這篇文章能🄵讓你有足🄵的背景知識，以便開始正確的使用 sockets 程式設計。

1 套接字

我只會討論關於 INET（例如：IPv4）的 sockets，但它們涵蓋了幾乎 99% 的 sockets 使用場景。而我也將僅討論關於 STREAM（比如：TCP）類型的 sockets - 除非你真的知道你在做什[illegible]（在這種情[illegible]下，這份指南可能不適合你），使用 STREAM 類型的 socket 會獲得比其他 sockets 類型更好的表現和性能。我將會嘗試解釋 socket 是什[illegible]，以及如何使用阻塞 (blocking) 和非阻塞 (non-blocking) sockets 的一些建議。但首先我會先談論阻塞 sockets。在處理非阻塞 sockets 之前，你需要了解它們的工作原理。

要理解這些東西的困難點之一在於"scoket" 可以代表多種具有些微差[illegible]的東西，這主要取[illegible]於上下文。所以首先，讓我們先區分「用[illegible]端 (client)」socket 和「伺服器端 (server)」socket 的差[illegible]，「用[illegible]端」socket 表示通訊的一端，「伺服器端」socket 更像是一個電話總機接[illegible]員。用[illegible]端應用程式（例如：你的[illegible]覽器）只能使用「用[illegible]端」socket; 它所連接的網路伺服器則同時使用「伺服器端」socket 和「用[illegible]端」socket 來進行通訊。

1.1 歷史

在各種形式的 IPC (Inter Process Communication) 中，sockets 是最受歡迎的。在任何特定的平台上，可能會存在其他更快速的 IPC 形式，但對於跨平台通訊來[illegible]，sockets 是唯一的選擇。

Sockets 作[illegible] Unix 的 BSD 分支的一部分在 Berkeley 被發明出來。它們隨著網際網路的普及而迅速蔓延開來。這是有很好的理由—sockets 和 INET 的結合讓世界各地任何的機器之間的通訊變得非常簡單（至少與其它方案相比是如此）。

2 建立一個 Socket

大致上來[illegible]，當你點擊了帶你來到這個頁面的連結時，你的[illegible]覽器做了以下的操作：

```
# 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))
```

當 `connect` 完成時，這個 socket `s` 可以用來發送請求來取得頁面的文本。同一個 socket 也會讀取回傳值，然後再被銷[illegible]。是的，會被銷[illegible]。用[illegible]端 socket 通常只用來做一次交[illegible]（或是一小組連續交[illegible]）。

網路伺服器 (web server) 的運作就稍微[illegible]雜一點。首先，網路伺服器會建立一個「伺服器端 socket」：

```
# 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)
```

有幾件事需要注意：我們使用了 `socket.gethostname()`，這樣 socket 才能對外部網路可見。如果我們使用了 `s.bind(('localhost', 80))` 或 `s.bind(('127.0.0.1', 80))`，我們會得到一個「伺服器端」socket，但是只能在同一台機器[illegible]可見。`s.bind(('', 80))` 指定 socket 可以透過機器的任何地址存取。

第二個要注意的是：數字小的連接埠 (port) 通常保留給「廣[illegible]人知的」服務（HTTP、SNMP 等）。如果你只是想執行程式，可以使用一個數字較大的連接埠（4 位數字）。

最後，`listen` 引數告訴 socket 函式庫 (library)，我們希望在[illegible]列 (queue) 中累積達 5 個（正常的最大值）連[illegible]請求後再拒[illegible]外部連[illegible]。如果其余的程式碼編寫正確，這應該足[illegible]了。

現在我們有一個監聽 80 連接埠的「伺服器端」socket 了，我們可以進入網路伺服器的主[illegible]圈了：

```
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()
```

事實上，有三種方法可以讓這個[F]圈運作 - 分配一個執行緒 (thread) 來處理 `clientsocket` 、建立一個新行程 (process) 來處理 `clientsocket`，或者將這個程式重新改寫成使用非阻塞 socket，[F]使用 `select` 在我們的「伺服器端」socket 和任何有效的 `clientsocket` 之間進行多工處理。稍後將會更詳細的介紹。現在最重要的是理解：這就是「伺服器端」socket 做的所有事情。它不會發送任何資料、也不接收任何資料，它只會建立「伺服器端」socket。每個 `clientsocket` 都是[F]了回應某些其他 `connect()` 到我們綁定的主機上的「用[F]端」socket。一但 `clientsocket` 建立完成，就會繼續監聽更多的連[F]請求。兩個「用[F]端」可以隨意的通訊 - 它們使用的是一些動態分配的連接埠，會在通訊結束的時候被回收[F]重新利用。

2.1 IPC

如果你需要在一台機器上的兩個行程間進行快速的行程間通訊 (IPC)，你應該考慮使用管道 (pipes) 或共享記憶體 (shared memory)。如果你確定要使用 AF_INET sockets，請將「伺服器端」socket 綁定到 `'localhost'`。在大多數平台上，這樣將會繞過幾個網路程式碼層，[F]且速度會更快一些。

也參考:

`multiprocessing` 將跨平台的行程間通訊整合到更高層的 API 中。

3 使用一個 Socket

首先需要注意，網頁[F]覽器的「用[F]端」socket 和網路伺服器的「用[F]端」socket 是非常類似的。也就是[F]，這是一個「點對點 (peer to peer)」的通訊方式，或者也可以[F]作[F]設計者，你必須[F]定通訊的規則。通常情[F]下，`connect` 的 socket 會通過發送一個請求或者信號來開始一次通訊。但這屬於設計[F]策，而不是 socket 的規則。

現在有兩組可供通訊使用的動詞。你可以使用 `send` 和 `recv`，或者可以將用[F]端 socket 轉[F]成類似檔案的形式，[F]使用 `read` 和 `write`。後者是 Java 中呈現 socket 的方式。我不打算在這[F]討論它，只是提醒你需要在 socket 上使用 `flush`。這些是緩衝的「檔案」，一個常見的錯誤是使用 `write` 寫入某些[F]容，然後直接 `read` 回覆。如果不使用 `flush`，你可能會一直等待這個回覆，因[F]請求可能還在你的輸出緩衝中。

現在我們來到 sockets 的主要障礙 - `send` 和 `recv` 操作的是網路緩衝區。他們不一定會處理你提供給它們的所有位元組（或者是你期望它處理的位元組），因[F]它們主要的重點是處理網路緩衝區。一般來[F]，它們會在關聯的網路衝區已滿 (`send`) 或已清空 (`recv`) 時回傳，然後告訴你它們處理了多少位元組。你的責任是一直呼叫它們直到你所有的訊息處理完成。

當 `recv` 回傳「零位元組 (0 bytes)」時，就表示另一端已經關閉（或著正在關閉）連[F]。你再也不能從這個連[F]上取得任何資料了。你可能還是可以成功發送資料；我稍後會對此進行更詳細的解釋。

像 HTTP 這樣的協議只使用一個 socket 進行一次傳輸，用[F]端發送一個請求，然後讀取一個回覆。就這樣，然後這個 socket 就會被銷[F]。這表示者用[F]端可以通過接收「零位元組」來檢測回覆的結束。

但是如果你打算在之後的傳輸中重新利用 socket 的話，你需要明白 *socket* 中是不存在 EOT （傳輸結束）。重申一次：如果一個 socket 的 `send` 或 `recv` 處理了「零位元組」後回傳，表示連[F]已經斷開。如果連[F][F]有斷開，你可能會永遠處於等待 `recv` 的狀態，因[F]（就目前來[F]）socket 不會告訴你[F]有更多資料可以讀取了。現在，如果你稍微思考一下，你就會意識到 socket 的一個基本事實：訊息要[F]是一個固定的長度（不好的做法），要[F]是可以被分隔的（普通的做法），要[F]是指定其長度（更好地做法），要[F]通過關閉連[F]來結束。完全由你來[F]定要使用哪種方式（但有些方法比其他方法來的更好）。

假設你不想結束連🅵，最簡單的方式就是使用固定長度的訊息：

```
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)
```

發送部分的程式碼幾乎可用於任何訊息的傳送方式 - 在 Python 中你發送一個字串，可以用 `len()` 來確認他的長度（即使字串包含了 `\0` 字元）。在這🅵，主要是接收的程式碼變得更🅵雜一些。（在 C 語言中，情🅵🅵有變得更糟，只是如果訊息中包含了 `\0` 字元，你就不能使用 `strlen` 函式。）

最簡單的改進方法是將訊息的第一個字元表示訊息的類型，🅵根據訊息的類型來🅵定訊息的長度。現在你需要使用兩次 `recv` - 第一次用於接收（至少）第一個字元來得知長度，第二次用於在🅵圈中接收剩下的訊息。如果你🅵定使用分隔符號的方式，你將會以某個任意的區塊大小進行接收（4096 或 8192 通常是網路緩衝區大小的良好選擇），🅵在收到的🅵容中掃描分隔符號。

需要注意的一個🅵雜情🅵是，如果你的通訊協議允許連續發送多個訊息（🅵有任何回應），🅵且你傳遞給 `recv` 函式一個任意的區塊大小，最後有可能讀取到下一條訊息的開頭。你需要將其放在一旁🅵保留下來，直到需要使用的時候。

使用長度作🅵訊息的前綴（例如，使用 5 個數字字元表示）會變得更🅵雜，因🅵（信不信由你）你可能無法在一次 `recv` 中獲得所有 5 個字元。在一般使用下，可能不會有這個狀🅵，但在高負載的網路下，除非使用兩個 `recv`（第一個用於確定長度，第二個用於取得訊息的資料部分），否則你的程式碼很快就會出現錯誤。這令人非常頭痛。同樣的情🅵也會讓你發現 `send` 🅵不總能在一次傳輸中完全清除所有🅵容。🅵管已經🅵讀了這篇文章，但最終還是無法解🅵!

🅵了節省篇幅、培養你的技能（🅵保持我的競🅵優勢），這些改進方法留給讀者自行練習。現在讓我們開始進行清理工作。

3.1 二进制数据

通过套接字发送二进制数据是完全可能的。主要问题是，并非所有机器都使用相同的二进制数据格式。例如，网络字节顺序 是大端序的，最大的字节在前，所以一个值为 `1` 的 16 位整数将是两个十六进制字节 `00 01` 。然而，大多数常见的处理器（x86 / AMD64 ， ARM，RISC-V）是小端序的，最小的字节在前 -- 同样的 `1` 将是 `01 00` 。

Socket 库有转换 16 位和 32 位整数的调用 - `ntohl, htonl, ntohs, htons` ，其中"n" 表示 网络，"h" 表示 主机，"s" 表示 *short* ，"l" 表示 *long* 。当网络顺序与主机顺序相同时，这些调用不做任何事情，但当机器的字节序相反时，这些调用会适当地交换字节。

在现今的 64 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大部分整数的值均为 0 或者 1。字符串形式的 `"0"` 为两个字节，而一个完整的 64 位整数将是八个。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。

在 Python 中是使用 `socket.setblocking(False)` 来设置非阻塞。在 C 中的做法更为复杂（例如，你需要在 BSD 风格的 `O_NONBLOCK` 和几乎无区别的 POSIX 风格的 `O_NDELAY` 之间作出选择，这与 `TCP_NODELAY` 完全不一样），但其思路实际上是相同的。你要在创建套接字之后但在使用它之前执行此操作。（实际上，如果你是疯子的话也可以反复进行切换。）

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）。