Socket 程式設計指南

發🄵 *3.11.8*

Guido van Rossum and the Python development team

4 月 02, 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 Sockets

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[F]，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[F]，主要是接收的程式碼變得更[F]雜一些。（在 C 語言中，情[F][F]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，[F]根據訊息的類型來[F]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[F]圈中接收剩下的訊息。如果你[F]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），[F]在收到的[F]容中掃描分隔符號。

需要注意的一個[F]雜情[F]是，如果你的通訊協議允許連續發送多個訊息（[F]有任何回應），[F]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[F]保留下來，直到需要使用的時候。

使用長度作[F]訊息的前綴（例如，使用 5 個數字字元表示）會變得更[F]雜，因[F]（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[F]，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[F]也會讓你發現 `send` [F]不總能在一次傳輸中完全清除所有[F]容。[F]管已經[F]讀了這篇文章，但最終還是無法解[F]!

[F]了節省篇幅、培養你的技能（[F]保持我的競[F]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 Binary Data

It is perfectly possible to send binary data over a socket. The major problem is that not all machines use the same formats for binary data. For example, network byte order is big-endian, with the most significant byte first, so a 16 bit integer with the value `1` would be the two hex bytes `00 01`. However, most common processors (x86/AMD64, ARM, RISC-V), are little-endian, with the least significant byte first - that same `1` would be `01 00`.

Socket libraries have calls for converting 16 and 32 bit integers - `ntohl, htonl, ntohs, htons` where "n" means *network* and "h" means *host*, "s" means *short* and "l" means *long*. Where network order is host order, these do nothing, but where the machine is byte-reversed, these swap the bytes around appropriately.

In these days of 64-bit machines, the ASCII representation of binary data is frequently smaller than the binary representation. That's because a surprising amount of the time, most integers have the value 0, or maybe 1. The string `"0"` would be two bytes, while a full 64-bit integer would be 8. Of course, this doesn't fit well with fixed-length messages. Decisions, decisions.

4 Disconnecting

Strictly speaking, you're supposed to use `shutdown` on a socket before you `close` it. The `shutdown` is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean "I'm not going to send anymore, but I'll still listen", or "I'm not listening, good riddance!". Most socket libraries, however, are so used to programmers neglecting to use this piece of etiquette that normally a `close` is the same as `shutdown(); close()`. So in most situations, an explicit `shutdown` is not needed.

One way to use `shutdown` effectively is in an HTTP-like exchange. The client sends a request and then does a `shutdown(1)`. This tells the server "This client is done sending, but can still receive." The server can detect "EOF" by a receive of 0 bytes. It can assume it has the complete request. The server sends a reply. If the `send` completes successfully then, indeed, the client was still receiving.

Python takes the automatic shutdown a step further, and says that when a socket is garbage collected, it will automatically do a `close` if it's needed. But relying on this is a very bad habit. If your socket just disappears without doing a `close`, the socket at the other end may hang indefinitely, thinking you're just being slow. *Please* `close` your sockets when you're done.

4.1 When Sockets Die

Probably the worst thing about using blocking sockets is what happens when the other side comes down hard (without doing a `close`). Your socket is likely to hang. TCP is a reliable protocol, and it will wait a long, long time before giving up on a connection. If you're using threads, the entire thread is essentially dead. There's not much you can do about it. As long as you aren't doing something dumb, like holding a lock while doing a blocking read, the thread isn't really consuming much in the way of resources. Do *not* try to kill the thread - part of the reason that threads are more efficient than processes is that they avoid the overhead associated with the automatic recycling of resources. In other words, if you do manage to kill the thread, your whole process is likely to be screwed up.

5 Non-blocking Sockets

If you've understood the preceding, you already know most of what you need to know about the mechanics of using sockets. You'll still use the same calls, in much the same ways. It's just that, if you do it right, your app will be almost inside-out.

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.)

The major mechanical difference is that `send`, `recv`, `connect` and `accept` can return without having done anything. You have (of course) a number of choices. You can check return code and error codes and generally drive

yourself crazy. If you don't believe me, try it sometime. Your app will grow large, buggy and suck CPU. So let's skip the brain-dead solutions and do it right.

Use `select`.

In C, coding `select` is fairly complex. In Python, it's a piece of cake, but it's close enough to the C version that if you understand `select` in Python, you'll have little trouble with it in C:

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

You pass `select` three lists: the first contains all sockets that you might want to try reading; the second all the sockets you might want to try writing to, and the last (normally left empty) those that you want to check for errors. You should note that a socket can go into more than one list. The `select` call is blocking, but you can give it a timeout. This is generally a sensible thing to do - give it a nice long timeout (say a minute) unless you have good reason to do otherwise.

In return, you will get three lists. They contain the sockets that are actually readable, writable and in error. Each of these lists is a subset (possibly empty) of the corresponding list you passed in.

If a socket is in the output readable list, you can be as-close-to-certain-as-we-ever-get-in-this-business that a `recv` on that socket will return *something*. Same idea for the writable list. You'll be able to send *something*. Maybe not all you want to, but *something* is better than nothing. (Actually, any reasonably healthy socket will return as writable - it just means outbound network buffer space is available.)

If you have a "server" socket, put it in the potential_readers list. If it comes out in the readable list, your `accept` will (almost certainly) work. If you have created a new socket to `connect` to someone else, put it in the potential_writers list. If it shows up in the writable list, you have a decent chance that it has connected.

Actually, `select` can be handy even with blocking sockets. It's one way of determining whether you will block - the socket returns as readable when there's something in the buffers. However, this still doesn't help with the problem of determining whether the other end is done, or just busy with something else.

Portability alert: On Unix, `select` works both with the sockets and files. Don't try this on Windows. On Windows, `select` works with sockets only. Also note that in C, many of the more advanced socket options are done differently on Windows. In fact, on Windows I usually use threads (which work very, very well) with my sockets.