Socket 程式設計指南

發🄵 *3.12.4*

Guido van Rossum and the Python development team

7 月 31, 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🅵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，🅵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🅵了回應某些其他 `connect()` 到我們綁定的主機上的「用🅵端」socket。一但 `clientsocket` 建立完成，就會繼續監聽更多的連🅵請求。兩個「用🅵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🅵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，🅵且速度會更快一些。

也參考

`multiprocessing` 將跨平台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🅵覽器的「用🅵端」socket 和網路伺服器的「用🅵端」socket 是非常類似的。也就是🅵，這是一個「點對點 (peer to peer)」的通訊方式，或者也可以🅵作🅵設計者，你必須🅵定通訊的規則。通常情🅵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🅵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🅵端 socket 轉🅵成類似檔案的形式，🅵使用 `read` 和 `write`。後者是 Java 中呈現 socket 的方式。我不打算在這🅵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🅵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🅵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🅵它們主要的重點是處理網路緩衝區。一般來🅵，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🅵。你再也不能從這個連🅵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🅵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🅵。這表示者用🅵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🅵已經斷開。如果連🅵🅵有斷開，你可能會永遠處於等待 `recv` 的狀態，因🅵（就目前來🅵）socket 不會告訴你🅵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🅵是一個固定的長度（不好的做法），要🅵是可以被分隔的（普通的做法），要🅵是指定其長度（更好地做法），要🅵通過關閉連🅵來結束。完全由你來🅵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 二進位資料

使用 socket 傳輸二進位資料完全是可行的。最主要的問題在於不同機器使用不同的二進位資料格式。例如，網路二進位順序 F用的是「大端序 big-endian」，所以一個值F 1 的 16 位元整數會表示成兩個 16 進位的位元組 `00 01`。然而大多數常見的處理器 (x86/AMD64，ARM，RISC-V) F用的是「小端序 little-endian」，所以相同的 1 會被表示成 `01 00`。(譯者注：將一個多位數的低位放在較小的位址處，高位放在較大的位址處，則稱小端序；反之則稱大端序。)

Socket 函式庫提供了用於轉F 16 位元和 32 位元整數的函式 - `ntohl`, `htonl`, `ntohs`, `htons`，其中"n" 表示 *network*，"h" 表示 *host*，"s" 表示 *short*，"l" 表示 *long*。當網路的位元組順序和主機位元組順序相同時，這些函式不會做任何操作，但當主機的位元組順序相反時，這些函式會適當的交F位元組順序。

在現今的 64 位元機器中，二進位資料的 ASCII 表示通常會比二進位表示要小，這是因F在很多情F下，大多數整數的值F 0 或者 1。例如，字串形式的 `"0"` 是兩個位元組，而完整的 64 位元整數則是 8 個位元組。當然，這對固定長度的訊息來F不太適合，需要自行F定。

4 結束連F

嚴格來F，在關閉 socket 前，你應該使用 `shutdown` 函式。`shutdown` 函式是發送給 socket 另一端的一個提醒。根據你傳遞的引數，它可以表示「我不會再發送任何訊息了，但我仍然會持續監聽」，或者是「我不會再繼續監聽了，真F!」。然而，大多數的 socket 函式庫或程式設計師都習慣忽略這種禮節，因F通常情F下 `close` 跟 `shutdown(); close()` 是一樣的。所以在大多數情F下，不需要再特地使用 `shutdown` 了。

有效使用 `shutdown` 的一種方式是在類似 HTTP 的交F中，用F端發送請求後，然後使用 `shutdown(1)`。這告訴伺服器「這個用F端已經發送完成，但仍可以接收」。伺服器可以通過接收「零位元組」來檢測"EOF"。這樣它就可以確定已經接收到完整的請求。伺服器發送回覆，如果 `send` 成功完成，那F用F端確實在持續接收。

Python 更進一步地F取自動關閉的步驟，F且當 socket 被垃圾回收機制回收時，如果需要的話，他會自動執行 `close`。但依賴這個機制是一個非常不好的習慣，如果你的 socket 在F有 `close` 的情F下消失了，那F另一端的 socket 可能會認F你只是慢了一步，而無期限的等待。*請務必*在使用完畢後使用 `close` 關閉你的 sockets。

4.1 Sockets 何時銷F

使用阻塞式 socket 最糟糕的地方可能是在另一端突然F制關閉（未執行 `close`）的情F下會發生什F? 你的 socket 很可能會處於阻塞狀態。TCP 是一種可靠的協議，它在放F連F之前會等待很長很長的時間。如果你正在使用執行緒，整個執行緒基本上已經無法使用。在這方面，你無法做太多事情。只要你不做一些愚蠢的事情，比如在執行阻塞式讀取時持有一個鎖，那F執行緒F不會消耗太多資源。*不要*試圖終止執行緒 - 執行緒比行程更有效的部分原因是它們避免了與自動回收資源相關的開銷。F句話F，如果你確實設法終止了執行緒，整個行程可能會出現問題。

5 非阻塞的 Sockets

如果你已經理解了前面的F容，你已經知道了大部分關於使用 sockets 的機制的所需知識，你仍然會以非常相似的方式使用相同的函式。就這樣而已，如果你做的對，你的程式就會是近乎完美的。

在 Python 中可以使用 `socket.setblocking(False)` 來設定F非阻塞。在 C 的作法更FF雜（例如，你需要在 BSD 風格的 `O_NONBLOCK` 和幾乎F有區F的 POSIX 風格的 `O_NDELAY` 之間做出選擇，這與 `TCP_NODELAY` 完全不同），但基本思想是一樣的，你要在建立 socket 後但在使用它之前執行此操作。(實際上，如果你願意的話，你甚至可以來回切F。)

主要的機制差F在於 `send`、`recv`、`connect` 和 `accept` 可能在F有執行任何操作的情F下就回傳了。你當然有多種選擇。你可以檢查回傳值和錯誤代碼，但這些操作通常會讓自己抓狂。如果你不相信我，

不妨試試看。你的應用程式會變得臃腫、錯誤百出，🅵且占用 CPU。所以，讓我們跳過無腦的解🅵方案，使用正確的方式。

使用 `select`。

在 C 中，編寫 `select` 是非常🅵雜的，但在 Python 中，這很簡單，🅵與 C 的版本非常類似，如果你理解了 Python 中的 `select`，在 C 中處理它時也不會有太大的困難：

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

你傳遞給 `select` 三個列表：第一個列表包含你可能想要嘗試讀取的所有 sockets；第二個包含所有你可能想要嘗試寫入的 sockets，最後一個（通常🅵空）包含你想要檢查錯誤的 sockets。你應該注意，一個 socket 可以同時存在於多個列表中。`select` 呼叫是阻塞的，但你可以設置超時。通常這是一個明智的做法 - 除非有充分的理由，否則給它一個很長的超時（比如一分鐘）。

作🅵回傳，你將獲得三個列表。它們包含實際上可讀取、可寫入和出錯的 sockets。這些列表中的每一個都是你傳入的相應列表的子集（可能🅵空）。

如果一個 socket 在輸出的可讀列表中，你可以幾乎確定，在這個業務中我們能🅵得到的最接近確定的事情是，對該 socket 的 `recv` 呼叫將會回傳一些🅵容。對於可寫列表，也是同樣的想法。你將能🅵發送一些 🅵容。也許不是全部，但一些🅵容總比什🅵都🅵有好。（實際上，任何比較正常的 socket 都會以可寫的方式回傳 - 這只是意味者「外送網路 (outbound network)」的緩衝空間是可用的。）

如果你有一個「伺服器端」socket，請將其放在 potential_readers 列表中，如果它在可讀列表中出現，你的 `accept` 呼叫（幾乎可以確定）會成功。如果你建立了一個新的 socket 去 `connect` 到其他地方，請將它放在 potential_writers 列表中，如果它在可寫列表中出現，那🅵他有可能已經連接上了。

實際上，即使是使用阻塞式 socket 的情🅵下，`select` 也很方便。這是一種判斷是否會被阻塞的方法之一 - 當緩衝區中有某些🅵容時，socket 會回傳🅵可讀。然而，這仍然無法解🅵判斷另一端是否完成，或者只是忙於其他事情的問題。

可移植性警告：在 Unix 上，`select` 同時適用於 sockets 和文件。但請不要在 Windows 上嘗試這🅵做，在 Windows 上，`select` 只適用於 sockets。同時，請注意，在 C 語言中，許多更進階的 socket 選項在 Windows 上有不同的實現方式。實際上，在 Windows 上，我通常會使用執行緒（這非常，非常有效）與我的 sockets 一起使用。