ソケットプログラミング HOWTO

リリース *2.7.17*

Guido van Rossum
and the Python development team

12 月 24, 2019

Python Software Foundation
Email: docs@python.org

目 次

著者 Gordon McMillan

概要

ソケットはそこかしこで使われているが、最大級に誤解されている技術でもある。この文書はソケットの全体像を俯瞰しており、チュートリアルとしてはあまり役に立たない。実際に動くモノを完成させるには、他にもやらなければいけないことがあるからだ。この文書はソケットの微妙なところ (たくさんある) まではカバーしていないが、恥ずかしくない使い方ができるようになる程度の情報は得られるはずだ。

第1章　ソケット

INET ソケットのことしか語らないつもりだが、利用率でいうとソケットの 99% 以上はこれだ。さらに中でも STREAM ソケットに話題を絞ろうと思う - 自分が何をしているのか分かっているのでない限り (分かってるならこの HOWTO なんて要らないだろ!)、 STREAM ソケットが一番分かりやすく、一番性能が出るのだ。そうやって謎に包まれたソケットの正体を明らかにしてゆくと共に、ブロッキングおよびノンブロッキングなソケットの扱いに関するいくつかのヒントを提示しよう。だが、まずはブロッキングソケットから始めることにする。ノンブロッキングを扱うより先に、ブロッキングの仕組みを知っておかなくてはならないのだ。

話を理解しにくくしている要因として、「ソケット」という言葉が文脈によって微妙に違うものを指すことが挙げられる。そこでまず、「クライアント」ソケット - 対話の両端 - と「サーバ」ソケット - 電話交換手みたいなもの - の区別を付けておこう。クライアント側アプリケーション (たとえばブラウザ) は「クライアント」ソケットだけを使うが、話し相手のウェブサーバは「サーバ」ソケットと「クライアント」ソケットの両方を使う。

1.1 歴史

各種 IPC (INTER PROCESS COMMUNICATION (End of Transfer) (プロセス間通信) の中でも、ソケットは群を抜いて人気がある。どのプラットフォームにも、ソケットより速い IPC はあるだろう。だが、プラットフォームをまたぐ通信はソケットの独擅場だ。

ソケットは BSD Unix の一部としてバークレイで発明され、インターネットの普及と共に野火のごとく広まった。それももっともなことで、ソケットと INET のコンビによって世界中どんなマシンとも、信じられないほど簡単 (少なくとも他のスキームと比べて) に通信できるようになったのだ。

第2章　ソケットの作成

あなたがリンクをクリックしてこのページに来たとき、ブラウザは大雑把に言って次のようなことをしたのである:

```
#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.mcmillan-inc.com", 80))
```

この connect が完了すると、ソケット s を使ってこのページ文章への要求を送ることができるようになる。その同じソケットが返答を読み、そして破壊される。そう、破壊される。クライアントソケットは通常、一回 (か少数の) やり取りで使い捨てになるのだ。

ウェブサーバで起こる事柄はもう少し複雑だ。まず「サーバソケット」を作る:

```
#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)
```

ここで注意すべき点がいくつかある: 今回はソケットが外界に見えるよう、 socket.gethostname() を使った。 s.bind(('localhost', 80)) や s.bind(('127.0.0.1', 80)) でも「サーバ」ソケットにはなるが、それだと同じマシン内にしか見えないものになってしまう。 s.bind(('', 80)) はこのマシンが持っている全てのアドレスで接続可能になるようにという指定になる。

ふたつめ: 小さな番号のポートは大抵、「ウェルノウン (有名)」なサービス (HTTP, SNMP 等々) のために取ってある。お遊びで使うのなら適当に大きな数 (4 桁) を使おう。

最後に: listen の引数はソケットライブラリに、接続要求を 5 個 (通常の最大値) まで順番待ちさせるように命じている。これ以降の外部接続は拒否するのだが、コードが適切に書かれていれば、それで十分すぎるほどだ。

よし、「サーバーソケット」ができて、80 番ポートで耳を澄ましているところまで来た。では、ウェブサーバのメインループに入ろう:

```
while 1:
    #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
```

(次のページに続く)

(前のページからの続き)

```
    ct = client_thread(clientsocket)
    ct.run()
```

このループには実際のところ、3 通りの一般的な動作方法がある - clientsocket を扱うようにスレッドを割り当てたり、clientsocket を扱う新しいプロセスを作ったり、あるいはノンブロッキングソケットを使うようにアプリを作り直して select で「サーバ」ソケットとアクティブな clientsocket の間を多重化したりするのだ。最後のについてはまた後にしよう。ここで理解しておくべき要点はこれだ: 以上が「サーバ」ソケットの仕事の **すべて** である。データは一切送信しないし、受信しない。「クライアント」ソケットを生み出すだけ。我々のバインドされているホストとポートに connect() してくる **他の** 「クライアント」ソケットに応える形で clientsocket を作り、作るや否や、さらなる接続を聞きに戻っていくのだ。このふたつの「クライアント」は、あとは勝手に喋っていればいい - 使うポートは動的に割り当てられ、会話が終わればリサイクルに廻される。

2.1 IPC

同一マシンのプロセス間で高速な IPC が必要なのであれば、そのプラットフォームが提供している何らかの共有メモリに目を向けるべきだ。共有メモリとロックやセマフォに基づいた簡素なプロトコルが断然一番速い。

ソケットを使うことにしたのであれば、「サーバ」ソケットは 'localhost' にバインドすることだ。こうすると、ほとんどのプラットフォームではネットワーク関連コードを何層かスキップすることになり、かなり速くなる。

第3章　ソケットの利用

はじめに憶えておくべきなのは、ウェブブラウザの「クライアント」ソケットとウェブサーバの「クライアント」ソケットがまったく同じ種族だということだ。つまり、これは「ピア・トゥ・ピア」(1 対 1) の会話である。別の言い方をすると、 **設計者として自分で会話のエチケット規則を決めなくてはいけない** ということでもある。通常は、 `connect` してくるソケットが要求あるいは宣言をして会話を始める。だが、それはそう設計しただけのことだ - ソケットの規則ではない。

さて、コミュニケーションに使う動詞は二組ある。 `send` と `recv` を使うこともできるし、クライアントソケットをファイルっぽい種族に変形して `read` と `write` を使っても良い。後者は Java のソケットの表現方法だ。ここで詳しく語るつもりはないが、その場合はソケットも `flush` しなければいけない、とだけ言っておく。これはバッファリングした「ファイル」なので、何かを `write` してすぐに返答を `read` するというのはよくある間違いだ。間に `flush` を入れないと、要求がまだ出力バッファにあって永遠に返事が来ない、という可能性がある。

さあ、ソケットの主要な難関に進もう - `send` と `recv` はネットワークバッファに働きかけるものだ。だから、手渡したもの (や返してもらいたいもの) を 1 バイトも残さず実際に処理してくれているとは限らない。一般的に言って、 `send` はバッファが埋まるまで、`recv` はバッファが空になるまで処理をして、そのバイト数を返す。メッセージが完全に処理されるまで繰り返し呼び出すのは **自分の** 責任なのだ。

`recv` が 0 バイトを返したときは、向こう側が接続を閉じてしまった (または閉じようとしている途中) という意味だ。もうこの接続でデータを受け取ることはない。永遠にだ。ただ、データ送信は成功するかもしれない; これについてはあとで語ることにしよう。

HTTP のようなプロトコルでは、ひとつのソケットを 1 回の転送にしか使わない。クライアントは要求を送り、返答を受ける。以上だ。これでソケットは破棄される。だからこの場合、クライアントは受信 0 バイトの時点で返答の末尾を検出することができる。

だが、以降の転送にもそのソケットを使い回すつもりなら、ソケットに EOT (End of Transfer) など **存在しない** ことを認識する必要がある。もう一度言おう: ソケットの `send` や `recv` が 0 バイト処理で返ってきたなら、その接続は終わっている。終わって **いない** なら、いつまで `recv` を待てばいいかは分からない。ソケットは「もう読むものが (今のところ) ないぜ」などと **言わない** のだから。このことを少し考えれば、ソケットの真実を悟ることになるだろう: **メッセージは必ず固定長か** (うげぇ) **区切り文字を使うか** (やれやれ) **長さ標識を付けておくか** (かなりマシ) **接続を閉じて終わらせるかのいずれかでなければいけない** のだ。選ぶ権利と責任はまったくもって自分にある (が、正しさの程度に違いはある)。

毎回接続を終わらせるのはイヤだとして、最も単純な解決策は固定長メッセージだろう:

```
class mysocket:
    '''demonstration class only
      - coded for clarity, not efficiency
    '''

    def __init__(self, sock=None):
        if sock is None:
```

(次のページに続く)

(前のページからの続き)

```
            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 == '':
                raise RuntimeError("socket connection broken")
            chunks.append(chunk)
            bytes_recd = bytes_recd + len(chunk)
        return ''.join(chunks)
```

この送信コードは、ほぼあらゆるメッセージ通信スキームで使える - 文字列を送るとき、Python なら長さを `len()` で見極めることができる (中に `\0` が埋め込まれていても大丈夫)。難しくしているのは、おもに受信コードである。 (なお、C でも事態はあまり悪くならないが、メッセージに `\0` が埋め込まれていると `strlen` が使えないのは面倒だ。)

最も簡単な改良法は、メッセージの最初の一文字をタイプ標識にして、そのタイプで長さを決定するというものだ。この場合ふたつの `recv` があることになる - 一番目でその一文字 (だけじゃなくても可) を取って長さを調べ、二番目でループして残りを取るのだ。あるいはもし区切り方式の道を行くのであれば、任意のサイズ (4096 か 8192 がネットワークバッファには最適なことが多い) で受信して区切り文字を走査していくことになる。

心に留めておくべき面倒な点がひとつ: 複数メッセージが次々に (何らかの返事を待たずに) 返ってくることのある会話プロトコルなら、そして任意のサイズを `recv` に渡しているなら、次のメッセージの冒頭部分まで読んでしまうことがあるかもしれない。そのときは、必要になるまで脇によけて、大切に保管しておく必要がある。

メッセージ冒頭に長さを (たとえば 5 桁の数字で) 付けるのは、それよりもさらに複雑になる。というのも、(信じられないかもしれないが) 一回の `recv` で 5 文字を全部受け取ることができるとは限らないからだ。お遊びでやっている間はごまかせても、高負荷ネットワークのもとでは、`recv` ループをふたつ使わないコードは、あっと言う間にダメになってしまう - 一番目は長さを見定める用で、二番目はデータ部分を受け取る用だ。うーむ、いやらしい。さらにこのとき、`send` も一発で全部を出し切れるとは限らないことに気付くだろう。なお、今こうやって読んでいても、いつか誰もが痛い目を見るのである!

紙面の都合および教育的配慮 (と著者の地位確保) のため、こうした改良は練習問題として残しておく。さあ片付けてしまおう。

3.1 バイナリデータ

バイナリデータはまったく問題なくソケットに乗せられる。問題は、すべてのマシンで同じ形式を使っているわけではないことにある。たとえば Motorola のチップなら 16 ビット整数の 1 という値をふたつの 16 進バイト列 00 01 で表現するが、Intel や DEC は逆バイトだ - 同じ 1 が 01 00 になるのだ。ソケットライブラリは 16 ビットや 32 ビット整数の変換用コールを持っている - `ntohl`, `htonl`, `ntohs`, `htons` である。”n” は *network*、 ”h” は *host* を意味する。 ”s” は *short* で ”l” は *long* だ。これらのコールは、「ネットワーク並び = ホスト並び」なら何もしないが、マシンが逆バイトならそれに合わせてぐるっと交換してくれる。

この 32 ビット時代、バイナリデータは ASCII 表現のほうが小さくなることが多い。というのも、long なのに値が 0 ばっかりでたまに 1 だとかいうことは驚くほど多いからだ。文字列なら ”0” は 2 バイトなのに、バイナリは 4 バイトも喰う。もちろんこれは固定長メッセージには合わないが。さあ、どうする、どうする。

第4章　切断

厳密には、ソケットを `close` する前には `shutdown` することになっている。 `shutdown` は相手ソケットに対する報告であり、渡す引数によって「これ以上こっちからは送らないけど、まだ聞いてるぜ」という意味になったり、「もう聞かない。せいせいした!」だったりする。しかしほとんどのソケットライブラリは、このエチケットを怠るプログラマに慣れてしまって、通常 `close` だけで `shutdown(); close()` と同じことになる。だから大抵はわざわざ `shutdown` しなくてもいい。

`shutdown` の効果的な使い方のひとつは、HTTP 風のやりとりだ。クライアントは要求を出してすぐに `shutdown(1)` する。これでサーバに、「クライアントは送信完了ですが、まだ受信可能です」と伝わる。サーバは 0 バイト受信で "EOF" を検出することができる。要求を残さず受け取ったことにして良いのだ。対してサーバは返答を送る。その `send` が成功したなら、クライアントは実際にまだ受信していたことになる。

Python はこの自動 shutdown をもう一歩進めて、ソケットが GC されるときに必要なら自動で `close` してくれると言っている。しかしこれに頼るクセをつけてはいけない。もしソケットが `close` せずに姿を消せば、相手ソケットはこちらが遅いだけだと思ってハングしてしまうかもしれない。 **お願いだから** 終わったらちゃんと `close` してくれ。

4.1 ソケットが死ぬと

ブロッキングソケットを使っていて一番いやなのは多分、相手側が意地悪く (`close` せずに) ダウンするときに起こる事柄だ。自分側のソケットは高確率でハングするだろう。SOCKSTREAM は信頼性の高いプロトコルなので、ずっとずっと待ち続けて、なかなか見捨てないのだ。スレッドを使っているのであれば、そのスレッド全体が根本から死んだ状態になる。こうなると、もう手の施しようがない。まあ、ブロッキング読み出しの間ロックし続けるといった馬鹿げたことをしていない限り、リソースの点ではたいして消費にならない。だから **ぜったいに** そのスレッドを殺そうとしてはいけない - プロセスよりスレッドが効率的である理由のひとつは、自動リソース回収にまつわるオーバヘッドを避けられるという点にあるのだ。つまり別の言い方をすると、どうにかしてそのスレッドを殺したなら、プロセス全体がぐちゃぐちゃになってしまうだろうということだ。

第5章　ノンブロッキングソケット

ここまで理解してきたなら、もうソケットの仕組みについて必要なことはほとんど知っていることになる。これからも同じコールを、ほぼ同じように使っていくだけ、それだけだ。これをちゃんとやっていれば、そのアプリはだいたい完璧であろう。

Python の場合、ノンブロッキングにするには `socket.setblocking(0)` を使う。 C ならもっと複雑だ (一例を挙げると、BSD 方式の `O_NONBLOCK` およびほぼ違いのない POSIX 方式 `O_NDELAY` のどちらを選ぶか決めなくてはならなくて、後者は `TCP_NODELAY` とは全然別物だったりする) が、考え方はまったく一緒だ。

構造上の大きな違いは、 `send`, `recv`, `connect`, `accept` が何もしないで戻ってくるかもしれないという点である。選択肢は (当然ながら) いくつかある。返り値とエラーコードをチェックするという方法もある。が、発狂すること請け合いだ。信じないなら、いつかやってみるといい。アプリは肥大化し、バグが増え、CPU を喰い尽くすだろう。だからそんな愚かな解法は飛ばして、正解に進もう。

`select` を使え。

C において `select` でコードを書くのはかなり面倒だが、Python なら造作もない。しかし Python で `select` を理解しておけば C でもほとんど問題なく書ける、という程度には似ている:

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

`select` に三つのリストを渡しているが、一番目にはあとで読みたくなるかもしれないソケットすべて、二番目には書き込みたくなるかもしれないソケットすべて、最後に (通常は空のままだが) エラーをチェックしたいソケットが入っている。ひとつのソケットが複数にまたがってリストされても構わないことを憶えておくと良い。なお、 `select` コールはブロックするが、時間制限を与えることができる。これは、やっておいて損はない - 特に理由がなければ、かなり長い (たとえば 1 分とかの) 時間制限を付けておくことだ。

戻り値として、三つのリストが手に入る。それぞれには、実際に読めるソケット、書けるソケット、エラー中のソケットが入っていて、渡したリストの部分集合 (空集合かもしれない) になっている。

出力のうち、readable リストにあるソケットについては、 `recv` がとりあえず **何か** を返すであろう、ということは史上最高度に確信できる。 writable リストも考え方は同じで、 **何か** は送れる。送りたいもの全体は無理かもしれないが、 **何も** ないよりはマシだろう。 (実のところ、ふつうに健康なソケットなら writable で返ってくることができる - それは外向きネットワークバッファに空きがあるというだけの意味しかないのだから)

「サーバ」ソケットは potential_readers リストに入れておこう。それが readable リストに入って出てきたら、 `accept` は (ほぼ) 確実に成功するはずだ。どこかへ `connect` するために作った新しいソケットは

potential_writers リストに入れる。それが writable リストに現れたら、接続が成功している可能性は高いと言える。

`select` の実にいやらしい問題がひとつ: 突然死したソケットが入力側リストのどこかにあれば、 `select` は失敗してしまう。そうなると、見つけるまですべてのリストでループしてソケットをひとつひとつ `select([sock],[],[],0)` していく必要がある。時間制限を 0 にしてあるので長くはかからないが、これは美しくない。

じつは `select` はブロッキングソケットにも便利に使える。それはブロックするかどうかを見極める方法のひとつである - バッファに何かがあれば readable として返ってくるのだ。しかしこれも、相手の用事がもう済んでいるのか、それとも単に他のことで忙しいだけなのかを見極める役には立たない。

非互換警報: Unix ではソケットにもファイルにも `select` が使える。これを Windows でやろうとしてはいけない。Windows で `select` はソケットにしか使えない。また C の場合、高度なソケットオプションの多くは、やり方が Windows では違っている。実際、Windows なら著者は通常、ソケットにスレッドを使っている (これは実に、実にうまくいく)。認めたくないが、何らかの性能を求めるなら Windows のコードは Unix のコードとはかなり異なるものになってしまうだろう。

5.1 性能

最速のソケットコードはノンブロッキングソケットを使って select で多重化するものだということに疑問の余地はない。 CPU に負荷をかけることなく LAN 接続を使いきるアプリは作れる。問題は、そうやって書かれたものはあまり他のことができなくなるということだ - いつでもデータをあちらこちらへ廻せるようにしている必要があるのだ。

もっと他にやることがあるアプリなのだとすれば、スレッドが最適解だ (さらにノンブロッキングソケットを使えばブロッキングより速い)。しかし残念ながら、各種 Unix のスレッド対応は API も品質もバラバラである。そのため Unix の一般解は、それぞれの接続を取り扱う子プロセスをフォークするというものだ。このオーバヘッドは甚大 (Windows ではプロセス生成のオーバヘッドが洒落にならないので無理) だ。それに、子プロセスがお互い完全に独立しているのでない限り、親子のプロセス間で通信するために他の IPC, たとえばパイプとか、共有メモリとセマフォだとかを使う必要が出てくる。

最後に、これを憶えておいてほしい。ブロッキングソケットはノンブロッキングよりも幾分遅いとはいえ、多くの場合そちらが「正解」である。結局のところ、ソケットから受け取るデータに基づいて動くアプリなら、 `recv` のかわりに `select` で待てるようにするためだけにロジックを複雑化する意味はあまりない。