








[首页](#)  
[所有文章](#)  
[观点与动态](#)  
[基础知识](#)  
[系列教程](#)  
[实践项目](#)  
[工具与框架](#)  
[工具资源](#)  
[Python小组](#)

- 导航条 -

[伯乐在线](#) > [Python - 伯乐在线](#) > [所有文章](#) > [系列教程](#) > 一起写一个Web服务器 (3)

## 一起写一个Web服务器 (3)

2015/07/30 · [系列教程](#) · [8 评论](#) · [Web服务器](#)

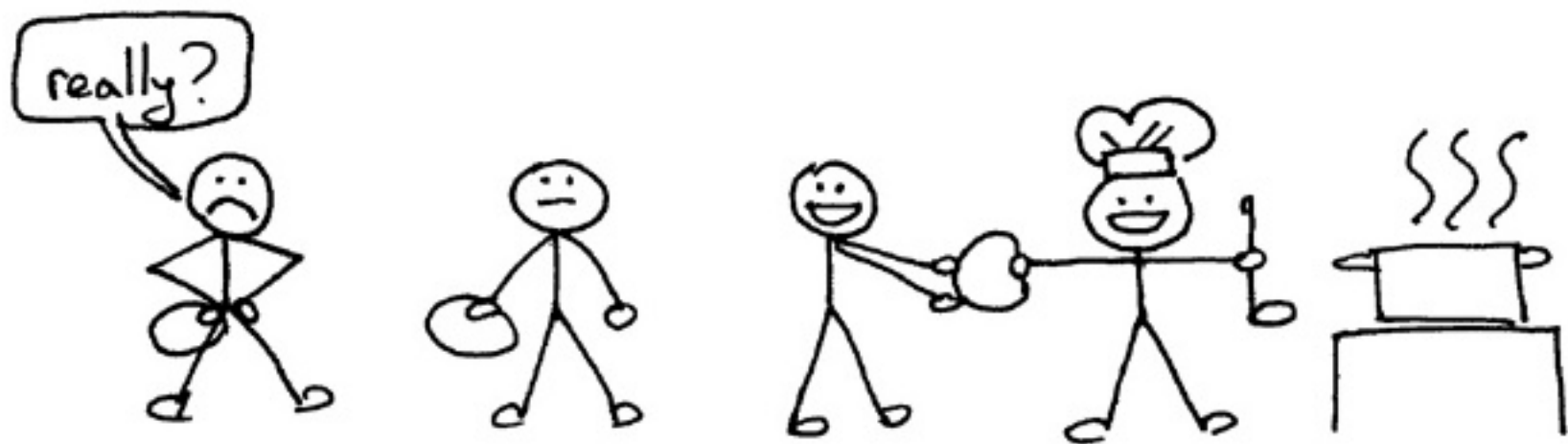
分享到:        1 本文由 [伯乐在线](#) - [高世界](#) 翻译, [黄利民](#) 校稿。未经许可, 禁止转载!  
英文出处: [ruslan spivak](#)。欢迎加入[翻译组](#)。

- 一起写一个 Web 服务器 (2)
- 一起写一个 Web 服务器 (1)

“发明创造时，我们学得最多”——Piaget

在[本系列第二部分](#)，你已经创造了一个可以处理基本的 HTTP GET 请求的 WSGI 服务器。我还问了你一个问题，“怎么让服务器在同一时间处理多个请求？”在本文中你将找到答案。那么，系好安全带加大马力。你马上就乘上快车啦。准备好Linux、Mac OS X（或任何类unix系统）和Python。本文的所有源码都能在GitHub上找到。

首先咱们回忆下一个基本的Web服务器长什么样，要处理客户端请求它得做什么。你在[第一部分](#)和[第二部分](#)创建的是一个迭代的服务  
器，每次处理一个客户端请求。除非已经处理了当前的客户端请求，否则它不能接受新的连接。有些客户端对此就不开心了，因为它们  
必须要排队等待，而且如果服务器繁忙的话，这个队伍会很长。



以下是迭代服务器webserver3a.py的代码:

```
#####  
# Iterative server - webserver3a.py #  
# #  
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #  
#####  
import socket
```

```
7 SERVER_ADDRESS = (HOST, PORT) = '', 8888
9 REQUEST_QUEUE_SIZE = 5
10
11 def handle_request(client_connection):
12     request = client_connection.recv(1024)
13     print(request.decode())
14     http_response = b"""
15 HTTP/1.1 200 OK
16
17 Hello, World!
18 """
19     client_connection.sendall(http_response)
20
21 def serve_forever():
22     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
23     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
24     listen_socket.bind(SERVER_ADDRESS)
25     listen_socket.listen(REQUEST_QUEUE_SIZE)
26     print('Serving HTTP on port {port} ...'.format(port=PORT))
27
28     while True:
29         client_connection, client_address = listen_socket.accept()
30         handle_request(client_connection)
31         client_connection.close()
32
33 if __name__ == '__main__':
34     serve_forever()
```

要观察服务器同一时间只处理一个客户端请求，稍微修改一下服务器，在每次发送给客户端响应后添加一个60秒的延迟。添加这行代码就是告诉服务器睡眠60秒。



以下是睡眠版的服务器webserver3b.py代码：

```
Python
1 #####
2 # Iterative server - webserver3b.py #
3 # #
4 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #
5 # #
6 # - Server sleeps for 60 seconds after sending a response to a client #
7 #####
8 import socket
9 import time
10
11 SERVER_ADDRESS = (HOST, PORT) = '', 8888
12 REQUEST_QUEUE_SIZE = 5
13
14 def handle_request(client_connection):
15     request = client_connection.recv(1024)
16     print(request.decode())
17     http_response = b"""
18 HTTP/1.1 200 OK
19
20 Hello, World!
21 """
22     client_connection.sendall(http_response)
23     time.sleep(60) # sleep and block the process for 60 seconds
24
25 def serve_forever():
26     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
27     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
28     listen_socket.bind(SERVER_ADDRESS)
29     listen_socket.listen(REQUEST_QUEUE_SIZE)
30     print('Serving HTTP on port {port} ...'.format(port=PORT))
31
32     while True:
33         client_connection, client_address = listen_socket.accept()
```

```
34         handle_request(client_connection)
35         client_connection.close()
36
37 if __name__ == '__main__':
38     serve_forever()
```

启动服务器：

```
1 $ python webserver3b.py
```

现在打开一个新的控制台窗口，运行以下curl命令。你应该立即就会看到屏幕上打印出了“Hello, World!”字符串：

```
1 $ curl http://localhost:8888/hello
2 Hello, World!
3
4 And without delay open up a second terminal window and run the same curl command:
```

立刻再打开一个控制台窗口，然后运行相同的curl命令：

```
1 $ curl http://localhost:8888/hello
```

如果你是在60秒内做的，那么第二个curl应该不会立刻产生任何输出，而是挂起。而且服务器也不会在标准输出打印出新请求体。在我的Mac上看起来像这样（在右下角的黄色高亮窗口表示第二个curl命令正挂起，等待服务器接受这个连接）：



当你等待足够长时间（大于60秒）后，你会看到第一个curl终止了，第二个curl在屏幕上打印出“Hello, World!”，然后挂起60秒，然后再终止：



```
(lsbaws)Ruslans-MacBook-Air:part3 rspivak$ python webserver3b.py
Serving HTTP on port 8888 ...
GET /hello HTTP/1.1
User-Agent: curl/7.37.1
Host: localhost:8888
Accept: */*

GET /hello HTTP/1.1
User-Agent: curl/7.37.1
Host: localhost:8888
Accept: */*

Ruslans-MacBook-Air:~ rspivak$ curl http://localhost:8888/hello
Hello, World!
Ruslans-MacBook-Air:~ rspivak$

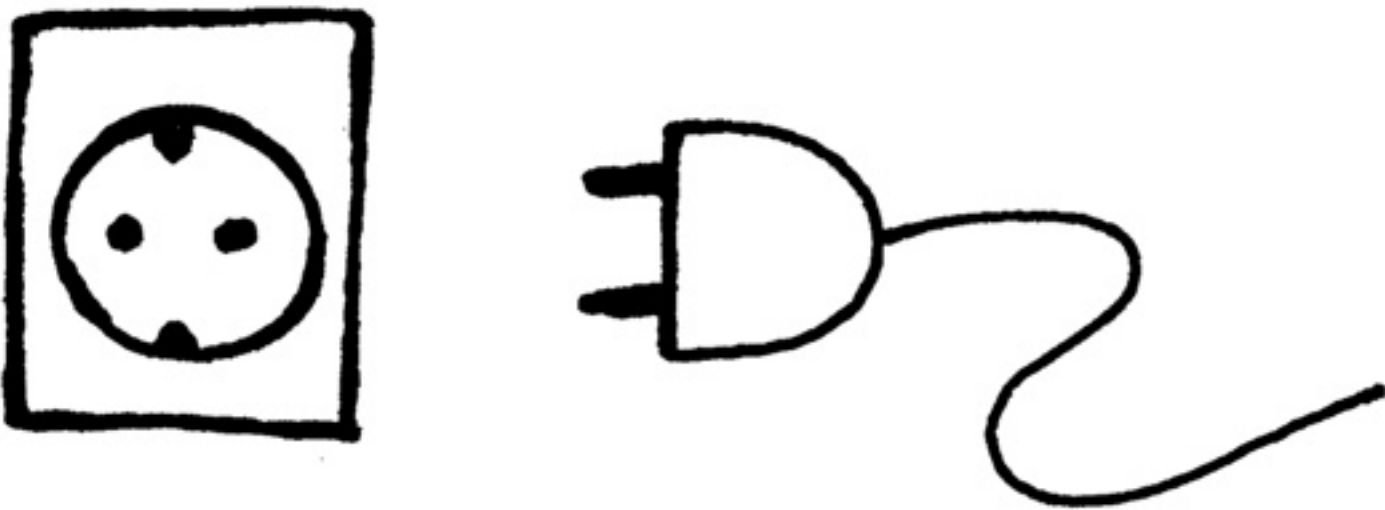
Ruslans-MacBook-Air:~ rspivak$ curl http://localhost:8888/hello
Hello, World!
Ruslans-MacBook-Air:~ rspivak$
```

Session: 0 3 2 1:bash 2:bash- 3:bash\*

"Ruslans-MacBook-Air.lo" 08:20 11-May-15

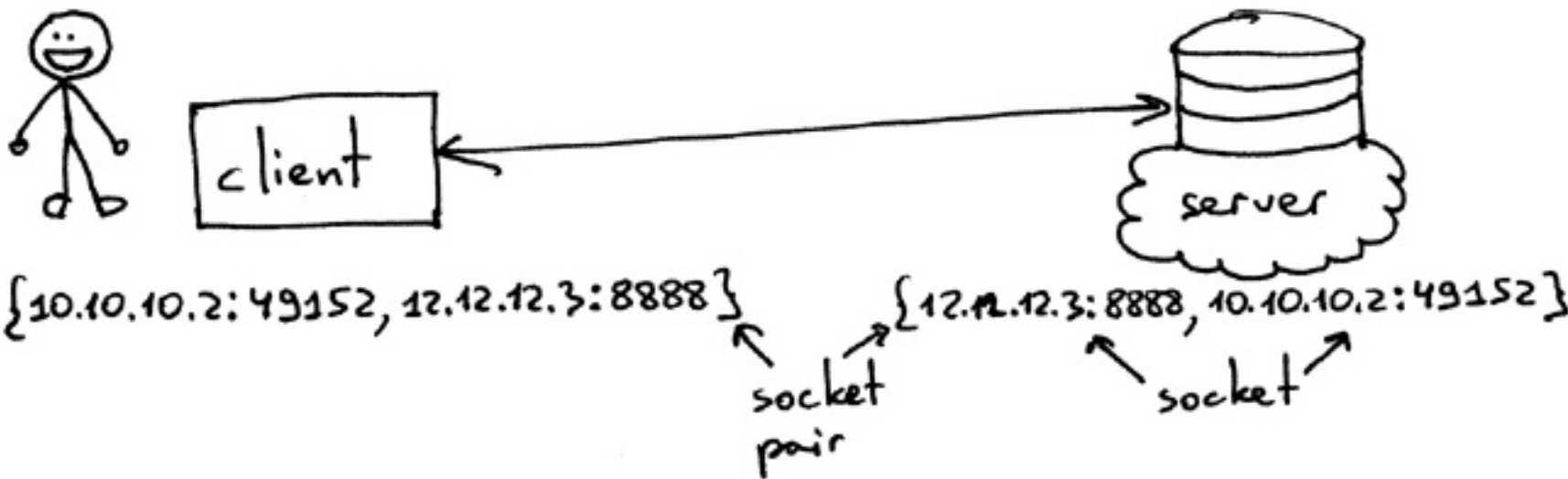
它是这么工作的，服务器完成处理第一个curl客户端请求，然后睡眠60秒后开始处理第二个请求。这些都是顺序地，或者迭代地，一步一步地，或者，在我们例子中是一次一个客户端请求地，发生。

咱们讨论点客户端和服务器的通信吧。为了让两个程序能够网络通信，它们必须使用socket。你在[第一部分](#)和[第二部分](#)已经见过socket了，但是，socket是什么呢？



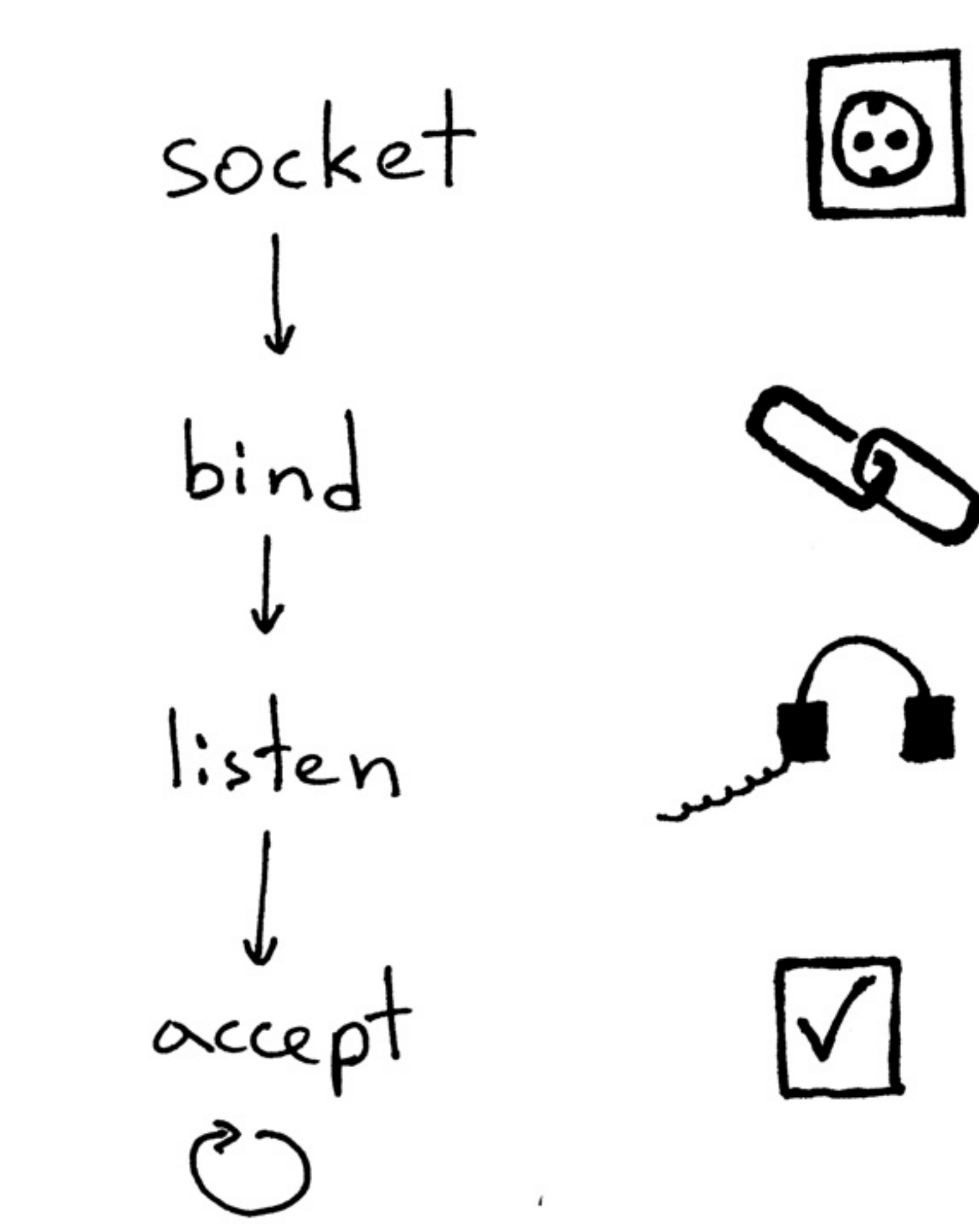
socket就是通信终端的一种抽象，它允许你的程序使用文件描述符和别的程序通信。本文我将详细谈谈在Linux/Mac OS X上的TCP/IP socket。理解socket的一个重要的概念是TCP socket对。

TCP的socket对是一个4元组，标识着TCP连接的两个终端：本地IP地址、本地端口、远程IP地址、远程端口。一个socket对唯一地标识着网络上的TCP连接。标识着每个终端的两个值，IP地址和端口号，通常被称为socket。



所以，元组{10.10.10.2:49152, 12.12.12.3:8888}是客户端TCP连接的唯一标识着两个终端的socket对。元组{12.12.12.3:8888, 10.10.10.2:49152}是服务器TCP连接的唯一标识着两个终端的socket对。标识TCP连接中服务器终端的两个值，IP地址12.12.12.3和端口8888，在这里就是指socket（同样适用于客户端终端）。

服务器创建一个socket并开始接受客户端连接的标准流程经历通常如下：



1. 服务器创建一个TCP/IP socket。在Python里使用下面的语句即可：

```
1 listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

2. 服务器可能会设置一些socket选项（这是可选的，上面的代码就设置了，为了在杀死或重启服务器后，立马就能再次重用相同的地址）。

```
1 listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

3. 然后，服务器绑定指定地址，bind函数分配一个本地地址给socket。在TCP中，调用bind可以指定一个端口号，一个IP地址，两者都，或者两者都不指定。

```
1 listen_socket.bind(SERVER_ADDRESS)
```

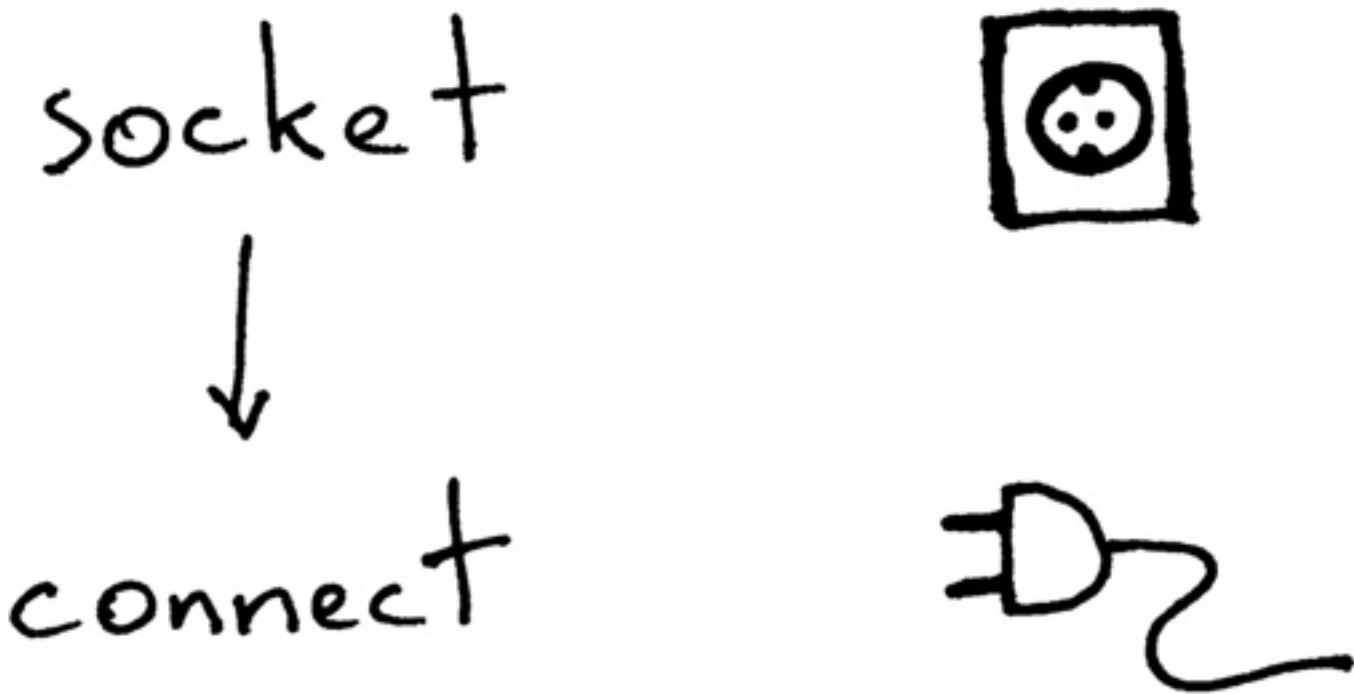
4. 然后，服务器让这个socket成为监听socket。

```
1 listen_socket.listen(REQUEST_QUEUE_SIZE)
```

listen方法只会被服务器调用。它告诉内核它要接受这个socket上的到来的连接请求了。

做完这些后，服务器开始循环地一次接受一个客户端连接。当有连接到达时，accept调用返回已连接的客户端socket。然后，服务器从这个socket读取请求数据，在标准输出上把数据打印出来，并回发一个消息给客户端。然后，服务器关闭客户端连接，准备好再次接受新的客户端连接。

下面是客户端使用TCP/IP和服务器通信要做的：



以下是客户端连接服务器，发送请求并打印响应的示例代码：

```
1 import socket
2
3 # create a socket and connect to a server
4 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5 sock.connect(('localhost', 8888))
6
7 # send and receive some data
8 sock.sendall(b'test')
9 data = sock.recv(1024)
10 print(data.decode())
```

创建socket后，客户端需要连接服务器。这是通过connect调用做到的：

```
1 sock.connect(('localhost', 8888))
```

客户端仅需提供要连接的远程IP地址或主机名和远程端口号即可。

可能你注意到了，客户端不用调用bind和accept。客户端没必要调用bind，是因为客户端不关心本地IP地址和本地端口号。当客户端调用connect时内核的TCP/IP栈自动分配一个本地IP址地和本地端口。本地端口被称为暂时端口（ephemeral port），也就是，short-lived 端口。





服务器上标识着一个客户端连接的众所周知的服务的端口被称为well-known端口（举例来说，80就是HTTP，22就是SSH）。操起Python shell，创建个连接到本地服务器的客户端连接，看看内核分配给你创建的socket的暂时的端口是多少（在这之前启动webserver3a.py或webserver3b.py）：

```
1 >>> import socket
2 >>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3 >>> sock.connect(('localhost', 8888))
4 >>> host, port = sock.getsockname()[:2]
5 >>> host, port
6 ('127.0.0.1', 60589)
```

上面这个例子中，内核分配了60589这个暂时端口。

在我开始回答第二部分提出的问题前，我需要快速讲一下几个重要的概念。你很快就知道为什么重要了。两个概念是进程和文件描述符。

什么是进程？进程就是一个正在运行的程序的实例。比如，当服务器代码执行时，它被加载进内存，运行起来的程序实例被称为进程。内核记录了进程的一堆信息用于跟踪，进程ID就是一个例子。当你运行服务器 webserver3a.py 或 webserver3b.py 时，你就在运行一个进程了。



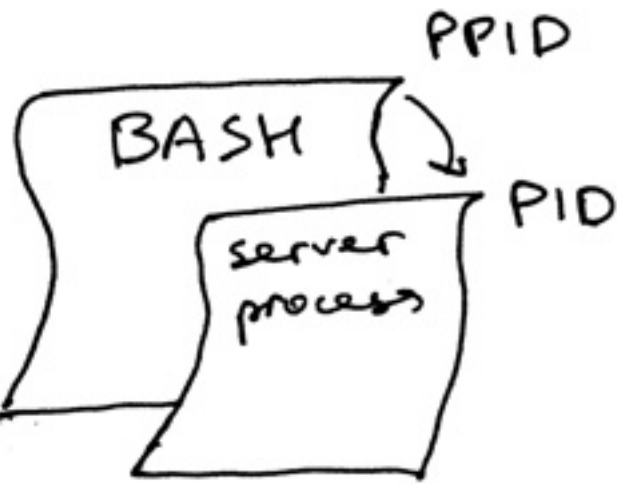
在控制台窗口运行webserver3b.py：

```
1 $ python webserver3b.py
```

在别的控制台窗口使用ps命令获取这个进程的信息：

```
1 $ ps | grep webserver3b | grep -v grep
2 7182 ttys003    0:00.04 python webserver3b.py
```

ps命令表示你确实运行了一个Python进程webserver3b。进程创建时，内核分配给它一个进程ID，也就是PID。在UNIX里，每个用户进程都有个父进程，父进程也有它自己的进程ID，叫做父进程ID，或者简称PPID。假设默认你是在BASH shell里运行的服务器，那新进程的父进程ID就是BASH shell的进程ID。



自己试试，看看它是怎么工作的。再启动Python shell，这将创建一个新进程，使用 os.getpid() 和 os.getppid() 系统调用获取Python shell进程的ID和父进程ID（BASH shell的PID）。然后，在另一个控制台窗口运行ps命令，使用grep查找PPID(父进程ID，我的是3148)。在下面的截图你可以看到在我的Mac OS X上，子Python shell进程和父BASH shell进程的关系：

```
>>> import os
>>> os.getpid()
10236
>>> os.getppid()
3148
>>>
```

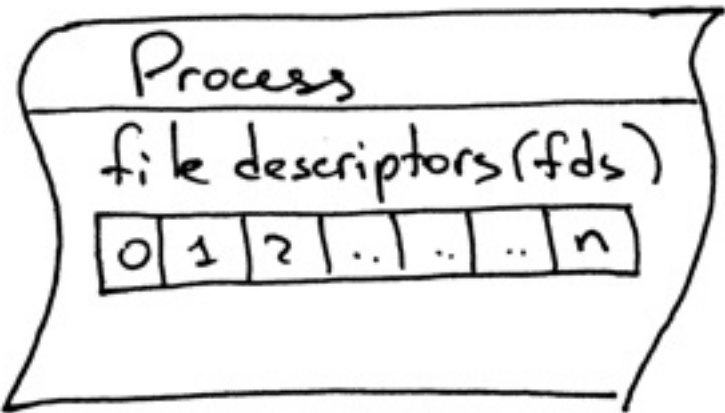
**PID** (points to 10236)  
**PPID** (points to 3148)

---

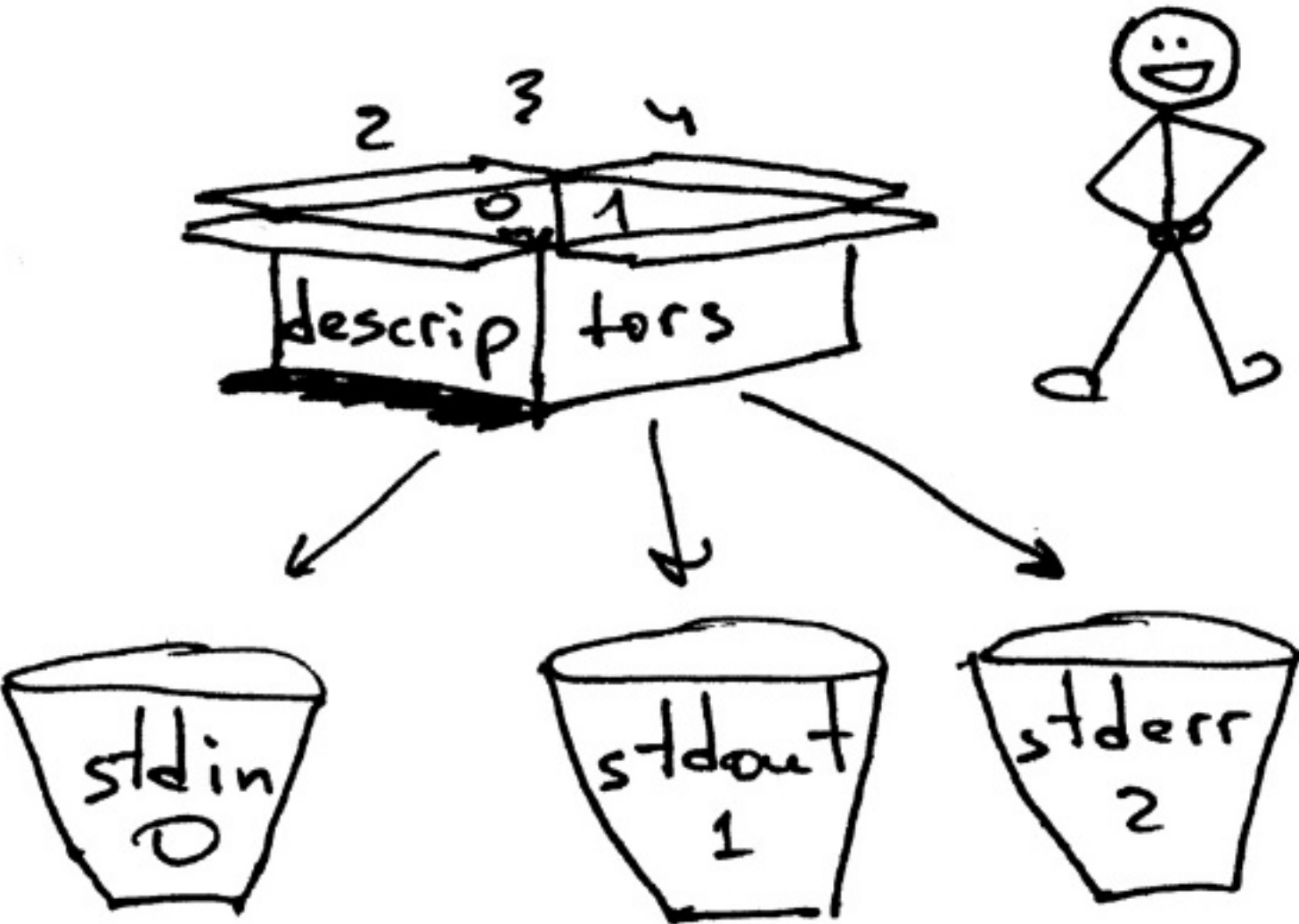
```
Russians-MacBook-Air:~ rspivak$ ps -opid,ppid,args | grep 3148 | grep -v grep
3148 1391 -bash
10236 3148 /usr/local/Cellar/python/2.7.9/Frameworks/Python.framework/Versions/2.7/Resources/Python.app/Contents/MacOS/Python
Russians-MacBook-Air:~ rspivak$
```

**PID** (points to 3148)  
**PPID** (points to 10236)

另一个要了解的重要概念是文件描述符。那么什么是文件描述符呢？文件描述符是当打开一个存在的文件，创建一个文件，或者创建一个socket时，内核返回的非负整数。你可能已经听过啦，在UNIX里一切皆文件。内核使用文件描述符来追踪进程打开的文件。当你需要读或写文件时，你就用文件描述符标识它好啦。Python给你包装成更高级别的对象来处理文件（和socket），你不必直接使用文件描述符来标识一个文件，但是，在底层，UNIX中是这样标识文件和socket的：通过它们的整数文件描述符。



默认情况下，UNIX shell分配文件描述符0给进程的标准输入，文件描述符1给进程的标准输出，文件描述符2给标准错误。



就像我前面说的，虽然Python给了你更高级别的文件或者类文件的对象，你仍然可以使用对象的fileno()方法来获取对应的文件描述符。回到Python shell来看看怎么做：



```
1 >>> import sys
2 >>> sys.stdin
3 <open file '<stdin>', mode 'r' at 0x102beb0c0>
4 >>> sys.stdin.fileno()
5 0
6 >>> sys.stdout.fileno()
7 1
8 >>> sys.stderr.fileno()
9 2
```

虽然在Python中处理文件和socket，通常使用高级的文件/socket对象，但有时候你需要直接使用文件描述符。下面这个例子告诉你如何使用write系统调用写一个字符串到标准输出，write使用整数文件描述符做为参数：

```
1 >>> import sys
2 >>> import os
3 >>> res = os.write(sys.stdout.fileno(), 'hellon')
4 hello
```

有趣的是——应该不会惊讶到你啦，因为你已经知道在UNIX里一切皆文件——socket也有一个分配给它的文件描述符。再说一遍，当你创建一个socket时，你得到的是一个对象而不是非负整数，但你也可以使用我前面提到的`fileno()`方法直接访问socket的文件描述符。

还有一件事我想说下：你注意到了吗？在第二个例子webserver3b.py中，当服务器进程在60秒的睡眠时你仍然可以用curl命令来连接。当然啦，curl没有立刻输出什么，它只是在那挂起。但为什么服务器不接受连接，客户端也不立刻被拒绝，而是能连接服务器呢？答案就是socket对象的listen方法和它的BACKLOG参数，我称它为REQUEST\_QUEUE\_SIZE(请求队列长度)。BACKLOG参数决定了内核为进入的连接请求准备的队列长度。当服务器webser3b.py睡眠时，第二个curl命令可以连接到服务器，因为内核在服务器socket的进入连接请求队列上有足够的可用空间。

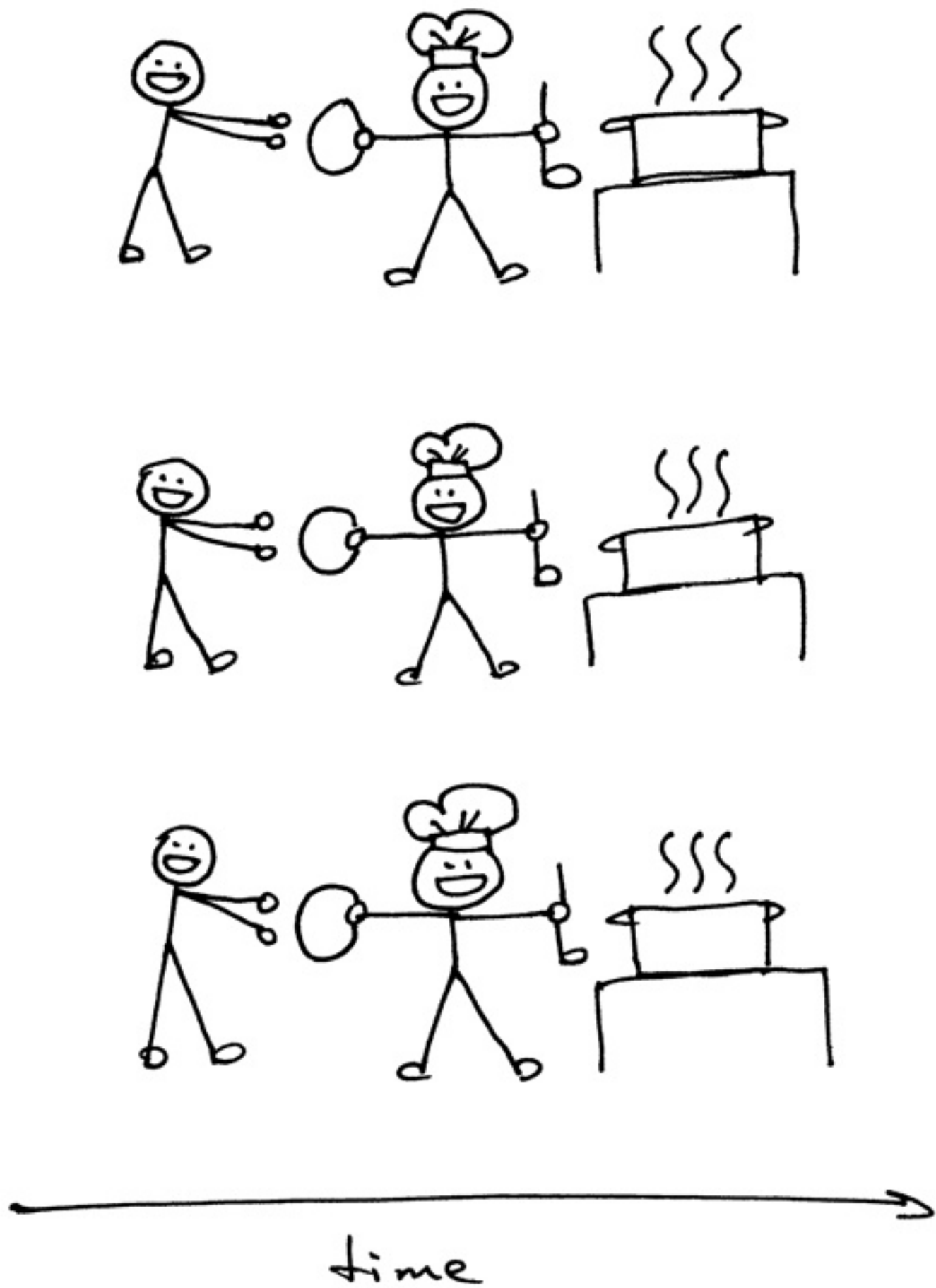
然而增加BACKLOG参数不会神奇地让服务器同时处理多个客户端请求，设置一个合理大点的backlog参数挺重要的，这样accept调用就不用等新连接建立起来，立刻就能从队列里获取新的连接，然后开始处理客户端请求啦。

吼吼！你已经了解了非常多的背景知识啦。咱们快速简要重述到目前为止你都学了什么（如果你都知道啦就温习一下吧）。



- 迭代服务器
- 服务器socket创建流程（socket, bind, listen, accept）
- 客户端连接创建流程（socket, connect）
- socket对
- socket
- 临时端口和众所周知端口
- 进程
- 进程ID（PID），父进程ID（PPID），父子关系。
- 文件描述符
- listen方法的BACKLOG参数的意义

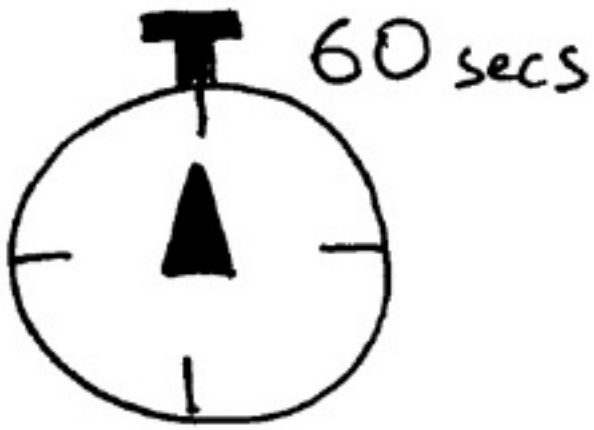
现在我准备回答第二部分问题的答案了：“怎样才能让服务器同时处理多个请求？”或者换句话说，“怎样写一个并发服务器？”



在Unix上写一个并发服务器最简单的方法是使用fork()系统调用。



下面就是新的牛逼闪闪的并发服务器webserver3c.py的代码，它能同时处理多个客户端请求（和咱们迭代服务器例子webserver3b.py一样，每个子进程睡眠60秒）：



```
1 #####
2 # Concurrent server - webserver3c.py #
3 # # #
4 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #
5 # # #
6 # - Child process sleeps for 60 seconds after handling a client's request #
7 # - Parent and child processes close duplicate descriptors #
8 # # #
9 #####
10 import os
11 import socket
12 import time
13
14 SERVER_ADDRESS = (HOST, PORT) = '', 8888
15 REQUEST_QUEUE_SIZE = 5
16
17 def handle_request(client_connection):
18     request = client_connection.recv(1024)
19     print(
20         'Child PID: {pid}. Parent PID {ppid}'.format(
21             pid=os.getpid(),
22             ppid=os.getppid(),
23         )
24     )
25     print(request.decode())
26     http_response = b"""
27 HTTP/1.1 200 OK
28
29 Hello, World!
30 """
31     client_connection.sendall(http_response)
32     time.sleep(60)
33
34 def serve_forever():
35     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
36     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
37     listen_socket.bind(SERVER_ADDRESS)
38     listen_socket.listen(REQUEST_QUEUE_SIZE)
39     print('Serving HTTP on port {port} ...'.format(port=PORT))
40     print('Parent PID (PPID): {pid}\n'.format(pid=os.getpid()))
41
42     while True:
43         client_connection, client_address = listen_socket.accept()
44         pid = os.fork()
45         if pid == 0: # child
46             listen_socket.close() # close child copy
47             handle_request(client_connection)
48             client_connection.close()
49             os._exit(0) # child exits here
50         else: # parent
51             client_connection.close() # close parent copy and loop over
52
53 if __name__ == '__main__':
54     serve_forever()
```

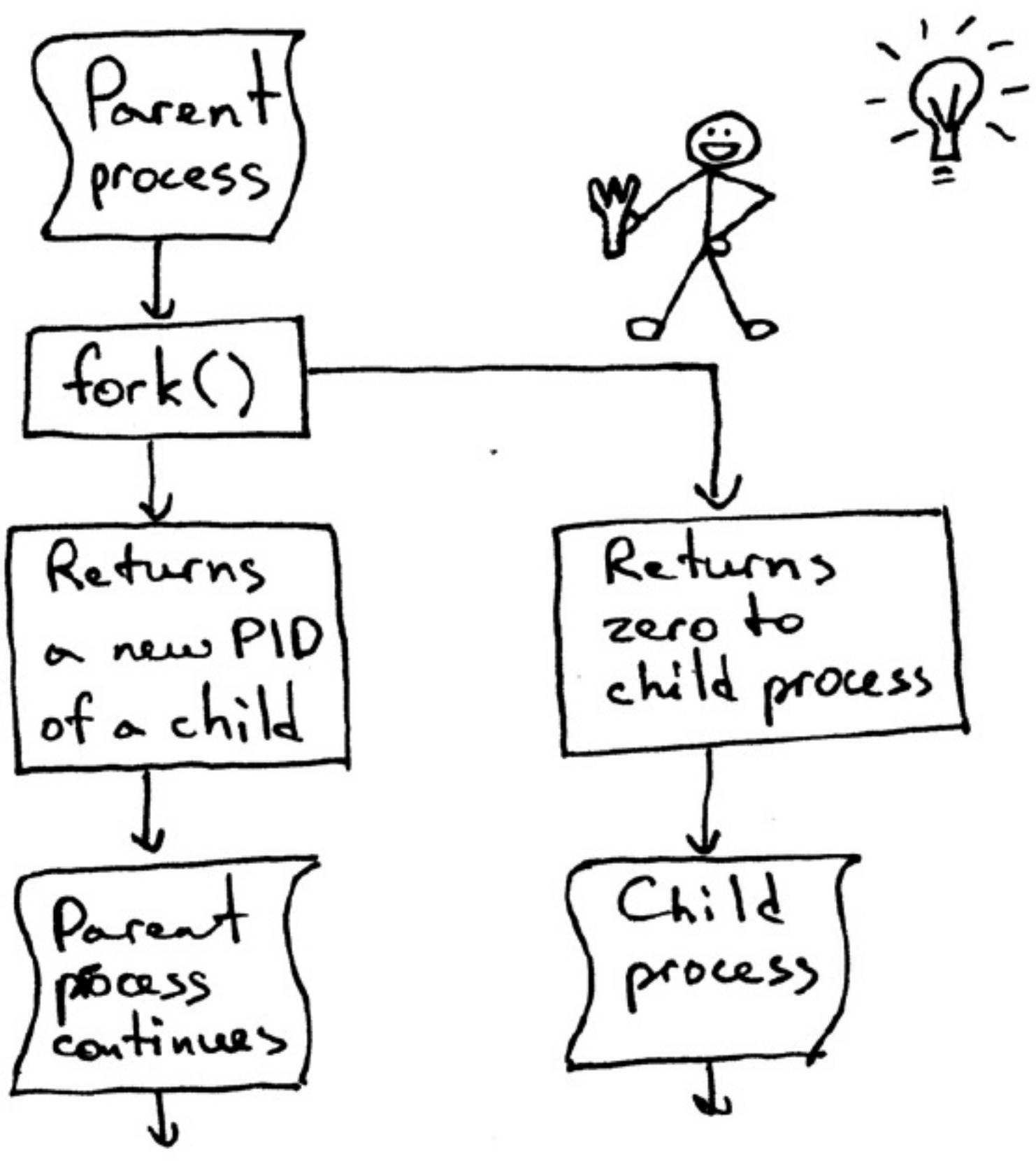
在深入讨论for如何工作之前，先自己试试，看看服务器确实可以同时处理多个请求，不像webserver3a.py和webserver3b.py。用下面命令启动服务器：

```
1 $ python webserver3c.py
```

像你以前那样试试用两个curl命令，自己看看，现在虽然服务器子进程在处理客户端请求时睡眠60秒，但不影响别的客户端，因为它们是被不同的完全独立的进程处理的。你应该能看到curl命令立刻就输出了“Hello, World!”，然后挂起60秒。你可以接着想运行多少curl命令就运行多少（嗯，几乎是任意多），它们都会立刻输出服务器的响应“Hello, Wrold”，而且不会有明显的延迟。试试看。

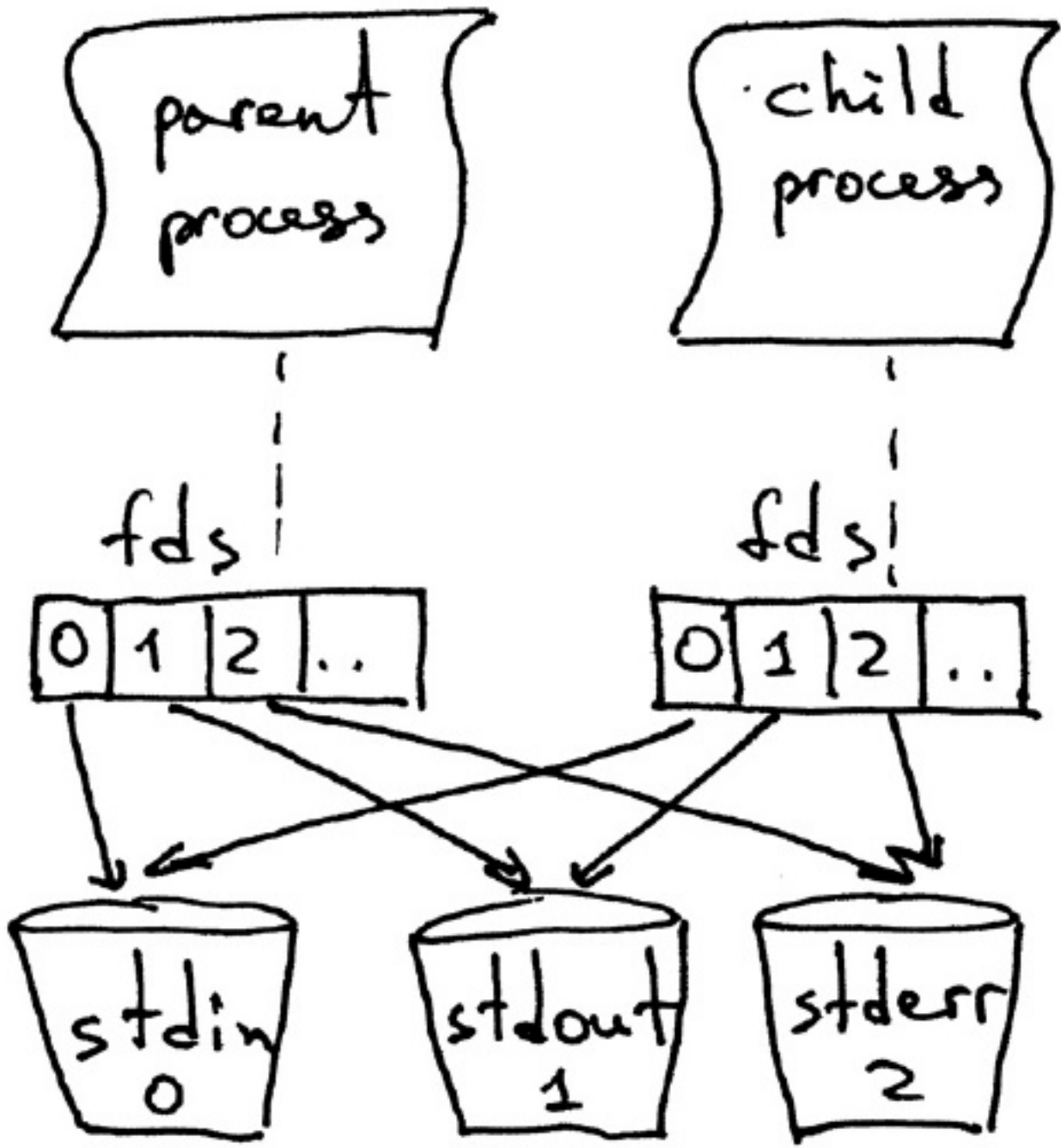


理解fork()的最重要的点是，你fork了一次，但它返回了两次：一个是在父进程里，一个是在子进程里。当你fork了一个新进程，子进程返回的进程ID是0。父进程里fork返回的是子进程的PID。



我仍然记得当我第一次知道它使用它时我对fork是有多着迷。它就像魔法一样。我正读着一段连续的代码，然后“duang”的一声：代码克隆了自己，然后就有两个相同代码的实例同时运行。我想除了魔法无法做到，我是认真哒。

当父进程fork了一个新的子进程，子进程就获取了父进程文件描述符的拷贝：



你可能已经注意到啦，上面代码里的父进程关闭了客户端连接：

```
1 else: # parent
2     client_connection.close() # close parent copy and loop over
```

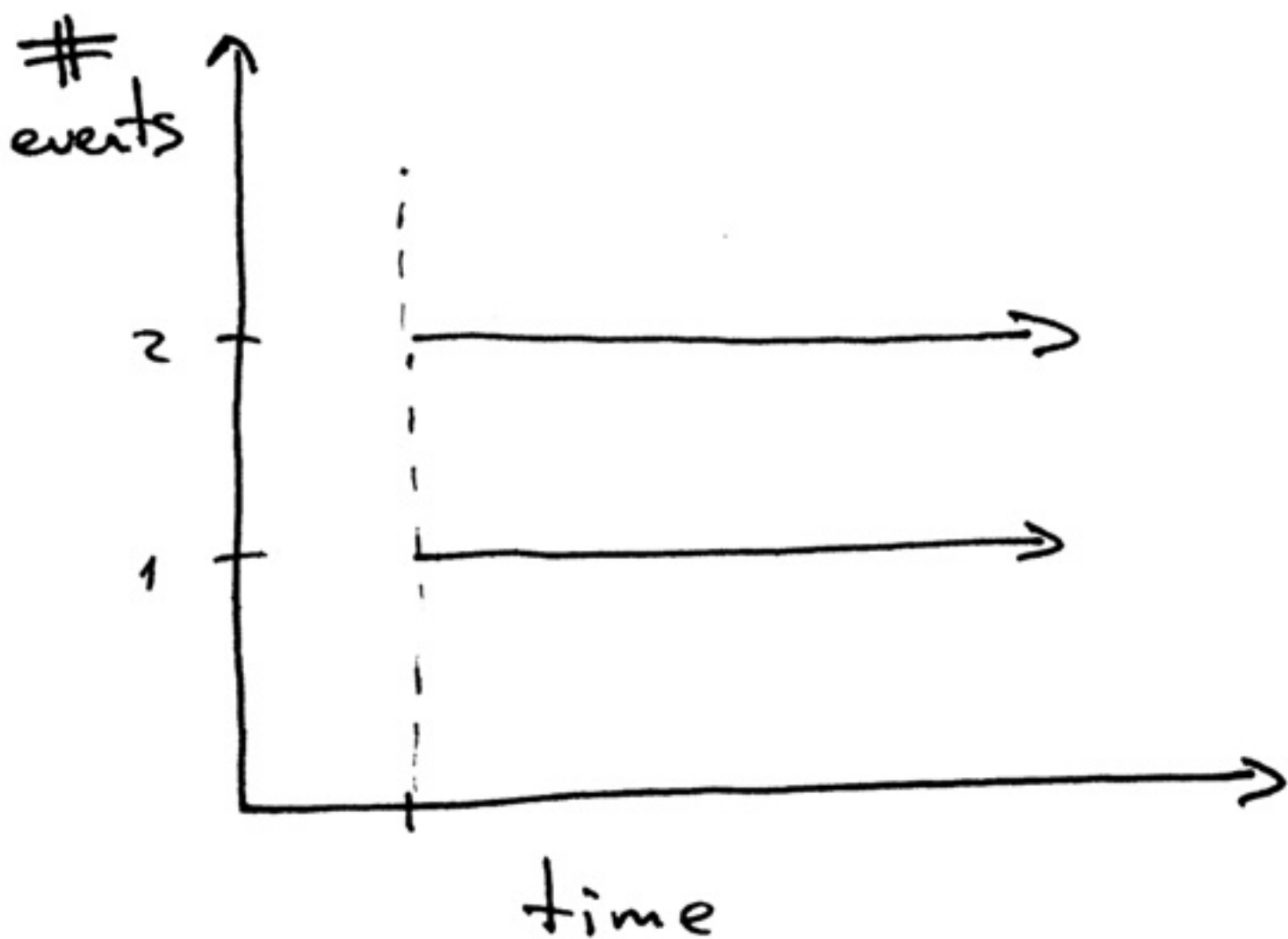
那么，如果它的父进程关闭了同一个socket，子进程为什么还能从客户端socket读取数据呢？答案就在上图。内核使用描述符引用计数来决定是否关闭socket。只有当描述符引用计数为0时才关闭socket。当服务器创建一个子进程，子进程获取了父进程的文件描述符拷贝，内核增加了这些描述符的引用计数。在一个父进程和一个子进程的场景中，客户端socket的描述符引用计数就成了2，当父进程关闭了客户端连接socket，它仅仅把引用计数减为1，不会引发内核关闭这个socket。子进程也把父进程的listen\_socket拷贝给关闭了，因为子进程不用管接受新连接，它只关心处理已经连接的客户端的请求：

```
1 listen_socket.close() # close child copy
```

本文后面我会讲下如果不关闭复制的描述符会发生什么。

你从并发服务器源码看到啦，现在服务器父进程唯一的角色就是接受一个新的客户端连接，fork一个新的子进程来处理客户端请求，然后重复接受另一个客户端连接，就没有别的事做啦。服务器父进程不处理客户端请求——它的小弟（子进程）干这事。

跑个题，我们说两个事件并发到底是什么意思呢？



当我们说两个事件并发时，我们通常表达的是它们同时发生。简单来说，这也不错，但你要知道严格定义是这样的：

```
Python
1 如果你不能通过观察程序来知道哪个先发生的，那么这两个事件就是并发的。
```

又到了简要重述目前为止已经学习的知识点和概念的时间啦.



- 在Unix下写一个并发服务器最简单的方法是使用fork()系统调用
- 当一个进程fork了一个新进程时，它就变成了那个新fork产生的子进程的父进程。
- 在调用fork后，父进程和子进程共享相同的文件描述符。
- 内核使用描述符引用计数来决定是否关闭文件/socket。
- 服务器父进程的角色是：现在它干的所有活就是接受一个新连接，fork一个子进来来处理这个请求，然后循环接受新连接。

咱们来看看，如果在父进程和子进程中你不关闭复制的socket描述符会发生什么吧。以下是个修改后的版本，服务器不关闭复制的描述符，webserver3d.py：

```
Python
1 #####
2 # Concurrent server - webserver3d.py #
3 # #
4 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #
5 #####
6 import os
7 import socket
8
9 SERVER_ADDRESS = (HOST, PORT) = '', 8888
10 REQUEST_QUEUE_SIZE = 5
11
12 def handle_request(client_connection):
13     request = client_connection.recv(1024)
14     http_response = b"""
15 HTTP/1.1 200 OK
16
17 Hello, World!
18 """
19     client_connection.sendall(http_response)
20
21 def serve_forever():
22     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
23     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
24     listen_socket.bind(SERVER_ADDRESS)
25     listen_socket.listen(REQUEST_QUEUE_SIZE)
```



```
26 print('Serving HTTP on port {port} ...'.format(port=PORT))
27
28 clients = []
29 while True:
30     client_connection, client_address = listen_socket.accept()
31     # store the reference otherwise it's garbage collected
32     # on the next loop run
33     clients.append(client_connection)
34     pid = os.fork()
35     if pid == 0: # child
36         listen_socket.close() # close child copy
37         handle_request(client_connection)
38         client_connection.close()
39         os._exit(0) # child exits here
40     else: # parent
41         # client_connection.close()
42         print(len(clients))
43
44 if __name__ == '__main__':
45     serve_forever()
```

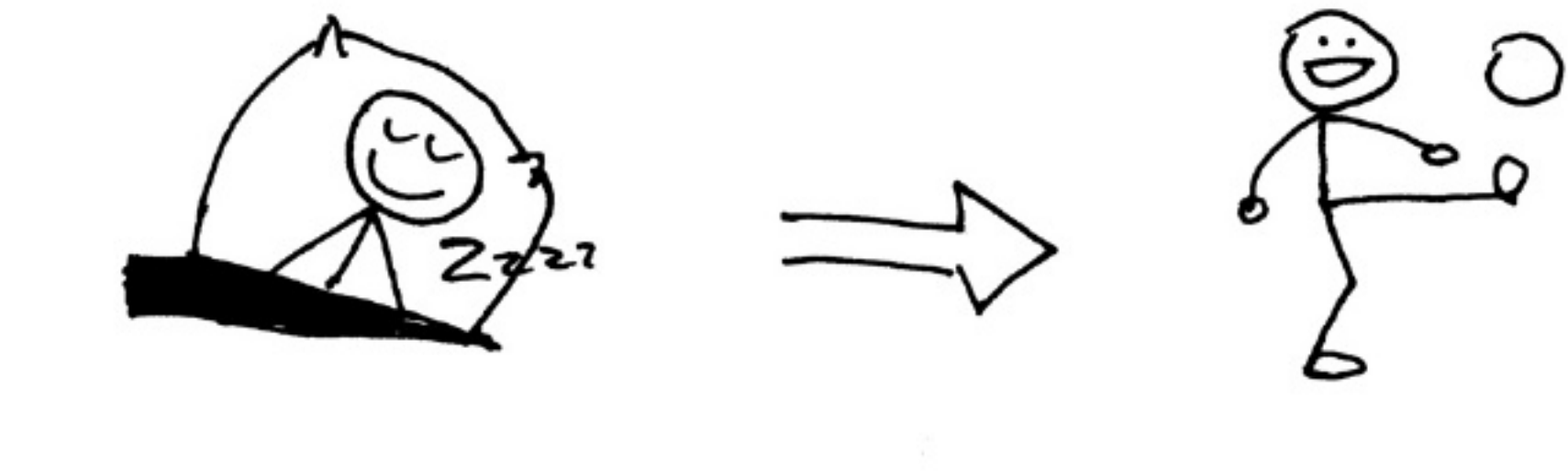
启动服务器：

```
Python
1 $ python webserver3d.py
```

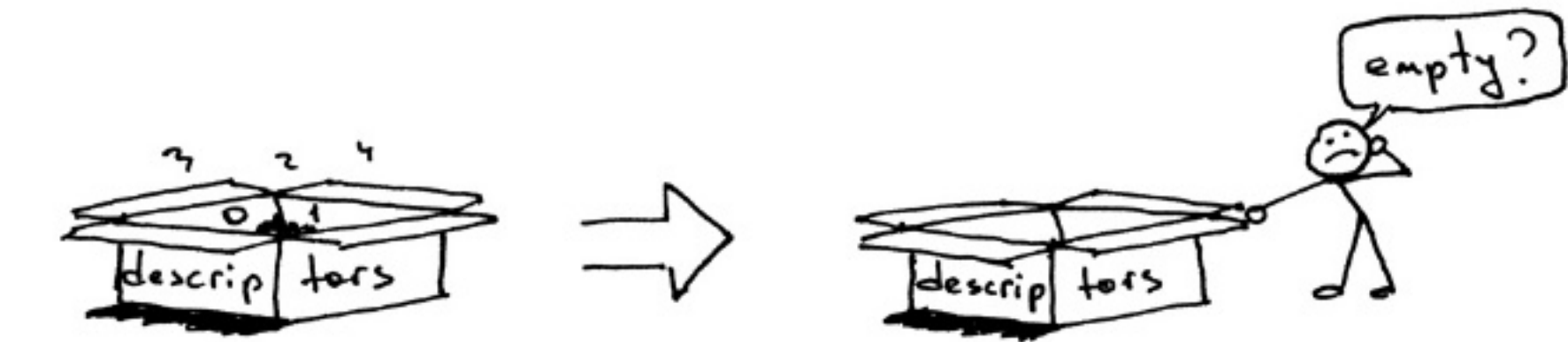
使用curl去连接服务器：

```
Python
1 $ curl http://localhost:8888/hello
2 Hello, World!
```

好的，curl打印出来并发服务器的响应，但是它不终止，一直挂起。发生了什么？服务器不再睡眠60秒了：它的子进程开心地处理了客户端请求，关闭了客户端连接然后退出啦，但是客户端curl仍然不终止。



那么，为什么curl不终止呢？原因就在于复制的文件描述符。当子进程关闭了客户端连接，内核减少引用计数，值变成了1。服务器子进程退出，但是客户端socket没有被内核关闭掉，因为引用计数不是0啊，所以，结果就是，终止数据包（在TCP/IP说法中叫做FIN）没有发送给客户端，所以客户端就保持在线啦。这里还有个问题，如果服务器不关闭复制的文件描述符然后长时间运行，最终会耗尽可用文件描述符。



使用Control-C停止webserver3d.py，使用shell内建的命令ulimit检查一下shell默认设置的进程可用资源：

```
Python
1 $ ulimit -a
2 core file size          (blocks, -c) 0
```

```
3 data seg size      (kbytes, -d) unlimited
4 scheduling priority      (-e) 0
5 file size            (blocks, -f) unlimited
6 pending signals       (-i) 3842
7 max locked memory     (kbytes, -l) 64
8 max memory size       (kbytes, -m) unlimited
9 open files            (-n) 1024
10 pipe size            (512 bytes, -p) 8
11 POSIX message queues  (bytes, -q) 819200
12 real-time priority    (-r) 0
13 stack size           (kbytes, -s) 8192
14 cpu time              (seconds, -t) unlimited
15 max user processes    (-u) 3842
16 virtual memory        (kbytes, -v) unlimited
17 file locks            (-x) unlimited
```

看到上面的了咩，我的Ubuntu上，进程的最大可打开文件描述符是1024。

现在咱们看看怎么让服务器耗尽可用文件描述符。在已存在或新的控制台窗口，调用服务器最大可打开文件描述符为256:

```
1 $ ulimit -n 256
```

在同一个控制台上启动webserver3d.py:

```
1 $ python webserver3d.py
```

使用下面的client3.py客户端来测试服务器。

```
1 #####
2 # Test client - client3.py #
3 # #
4 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #
5 #####
6 import argparse
7 import errno
8 import os
9 import socket
10
11 SERVER_ADDRESS = 'localhost', 8888
12 REQUEST = b"""
13 GET /hello HTTP/1.1
14 Host: localhost:8888
15
16 """
17
18 def main(max_clients, max_conns):
19     socks = []
20     for client_num in range(max_clients):
21         pid = os.fork()
22         if pid == 0:
23             for connection_num in range(max_conns):
24                 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
25                 sock.connect(SERVER_ADDRESS)
26                 sock.sendall(REQUEST)
27                 socks.append(sock)
28                 print(connection_num)
29                 os._exit(0)
30
31 if __name__ == '__main__':
32     parser = argparse.ArgumentParser(
33         description='Test client for LSBAWS.',
34         formatter_class=argparse.ArgumentDefaultsHelpFormatter,
35     )
36     parser.add_argument(
37         '--max-conns',
38         type=int,
39         default=1024,
40         help='Maximum number of connections per client.'
41     )
42     parser.add_argument(
43         '--max-clients',
44         type=int,
45         default=1,
46         help='Maximum number of clients.'
47     )
48     args = parser.parse_args()
49     main(args.max_clients, args.max_conns)
```

在新的控制台窗口里，启动client3.py，让它创建300个连接同时连接服务器。

```
Python
1 $ python client3.py --max-clients=300
```

很快服务器就崩了。下面是我电脑上抛异常的截图：

```
248
249
250
251
252
Traceback (most recent call last):
  File "webserver3d.py", line 58, in <module>
  File "webserver3d.py", line 43, in serve_forever
  File "/usr/lib/python2.7/socket.py", line 202, in accept
socket.error: [Errno 24] Too many open files
```

教训非常明显啦——服务器应该关闭复制的描述符。但即使关闭了复制的描述符，你还没有接触到底层，因为你的服务器还有个问题，僵尸！



是哒，服务器代码就是产生了僵尸。咱们看下是怎么产生的。再次运行服务器：

```
Python
1 $ python webserver3d.py
```

在另一个控制台窗口运行下面的curl命令：

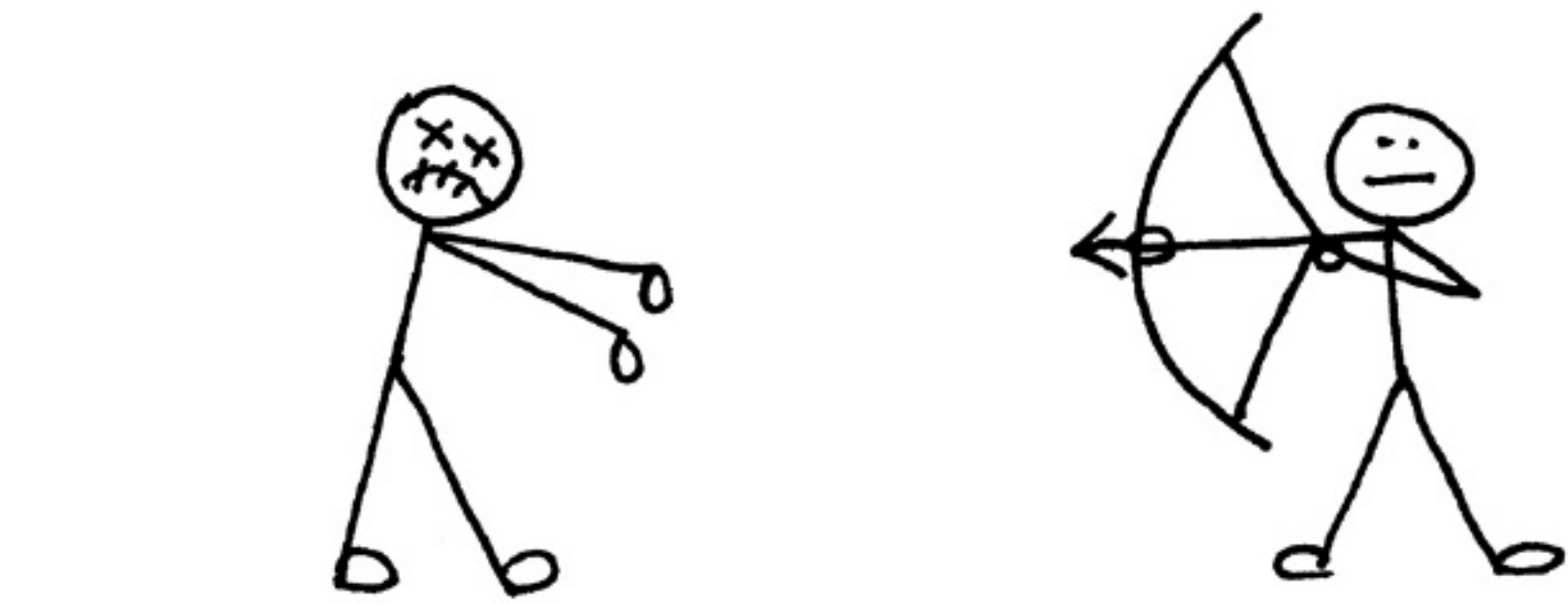
```
Python
1 $ curl http://localhost:8888/hello
```



现在运行ps命令，显示运行着的Python进程。以下是我的Ubuntu电脑上的ps输出：

```
1 $ ps auxw | grep -i python | grep -v grep
2 vagrant  9099  0.0  1.2 31804 6256 pts/0    S+   16:33   0:00 python webserver3d.py
3 vagrant  9102  0.0  0.0      0      0 pts/0    Z+   16:33   0:00 [python] <defunct>
```

你看到上面第二行了咩？ 它说Pid为9102的进程的状态是Z+，进程的名称是。这个就是僵尸啦。僵尸的问题在于，你杀死不了他们啊。



即使你试着用 \$ kill -9 来杀死僵尸，它们还是会幸存下来哒，自己试试看。

僵尸到底是什么呢？为什么咱们的服务器会产生它们呢？僵尸就是一个进程终止了，但是它的父进程没有等它，还没有接收到它的终止状态。当一个子进程比父进程先终止，内核把子进程转成僵尸，存储进程的一些信息，等着它的父进程以后获取。存储的信息通常就是进程ID，进程终止状态，进程使用的资源。嗯，僵尸还是有用的，但如果服务器不好好处理这些僵尸，系统就会越来越堵塞。咱们看看怎么做到的。首先停止服务器，然后新开一个控制台窗口，使用ulimit命令设置最大用户进程为400（确保设置打开文件更高，比如500吧）：

```
1 $ ulimit -u 400
2 $ ulimit -n 500
```

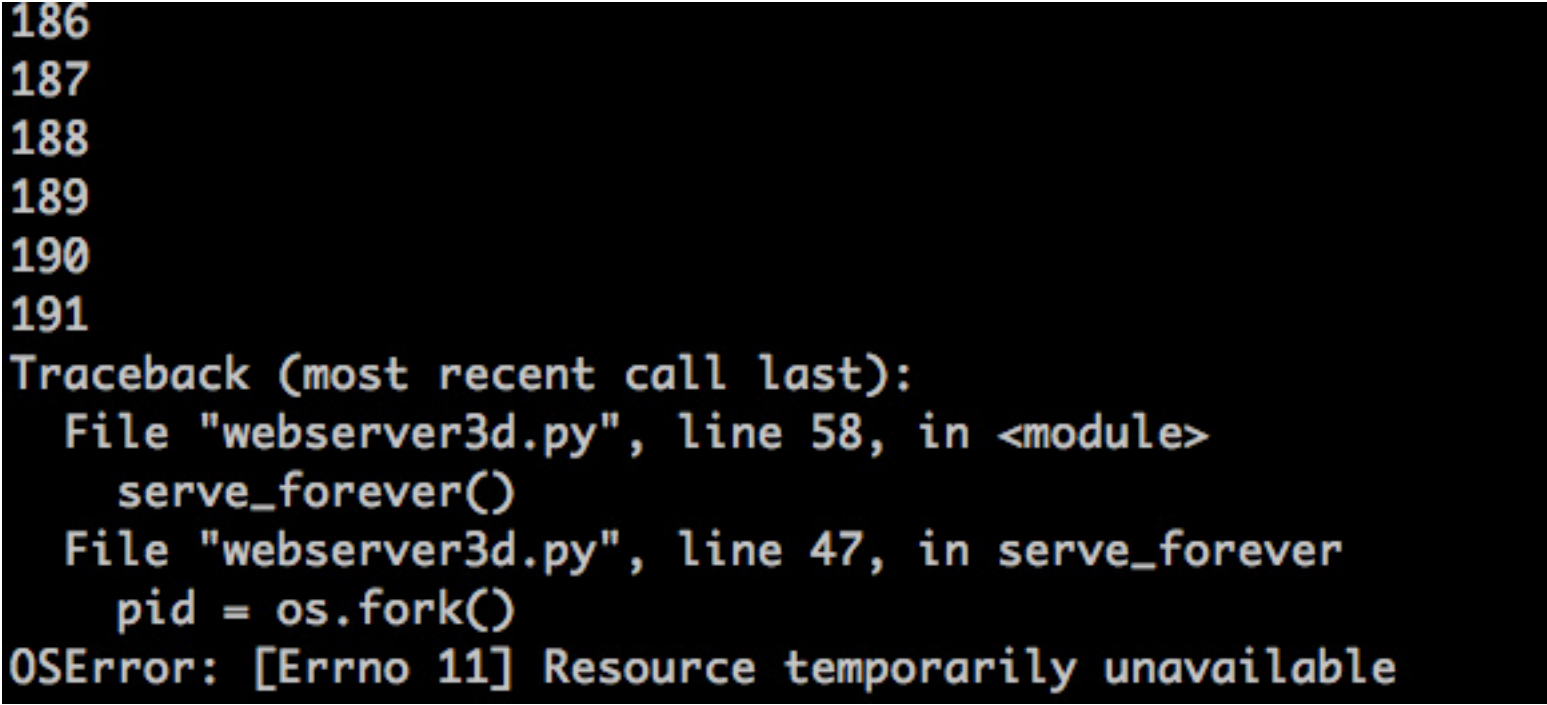
在同一个控制台窗口运行webserver3d.py：

```
1 $ python webserver3d.py
```

新开一个控制台窗口，启动client3.py，让它创建500个连接同时连接到服务器：

```
1 $ python client3.py --max-clients=500
```

然后，服务器又一次崩了，是OSError的错误：抛了资源临时不可用的异常，当试图创建新的子进程时但创建不了时，因为达到了最大子进程数限制。以下是我的电脑的截图：



看到了吧，如果你不处理好僵尸，服务器长时间运行就会出问题。我会简短讨论下服务器应该怎样处理僵尸问题。

咱们简要重述下目前为止你已经学习到主要知识点：

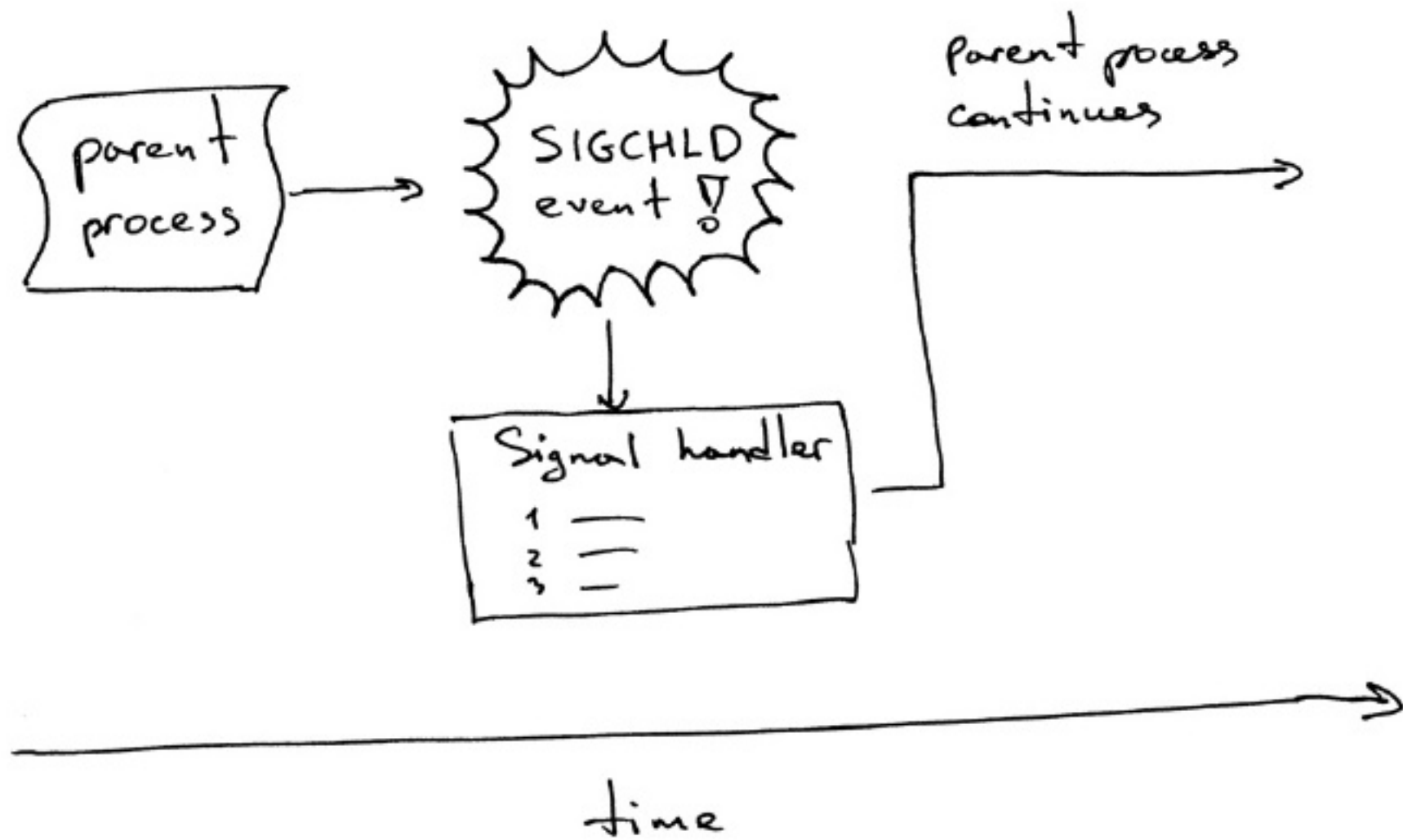


- 如果不关闭复制描述符，客户端不会终止，因为客户端连接不会关闭。
- 如果不关闭复制描述符，长时间运行的服务器最终会耗尽可用文件描述符（最大打开文件）。
- 当fork了一个子进程，然后子进程退出了，父进程没有等它，而且没有收集它的终止状态，它就变成僵尸了。
- 僵尸要吃东西，我们的场景中，就是内存。服务器最终会耗尽可用进程（最大用户进程），如果不处理好僵尸的话。
- 僵尸杀不死的，你需要等它们。

那么，处理好僵尸的话，要做什么呢？要修改服务器代码去等僵尸，获取它们的终止状态。通过调用wait系统调用就好啦。不幸的是，这不完美，因为如果调用wait，然而没有终止的子进程，wait就会阻塞服务器，实际上就是阻止了服务器处理新的客户端连接请求。有其他办法吗？当然有啦，其中之一就是使用信息处理器和wait系统调用组合。



以下是如何工作的。当一个子进程终止了，内核发送SIGCHLD信号。父进程可以设置一个信号处理器来异步地被通知，然后就能wait子进程获取它的终止状态，因此阻止了僵尸进程出现。



顺便说下，异步事件意味着父进程不会提前知道事件发生的时间。

修改服务器代码，设置一个SIGCHLD事件处理器，然后在事件处理器里wait终止的子进程。webserver3e.py代码如下：

```
1 #####
2 # Concurrent server - webserver3e.py
3 #
4 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X
5 #####
6 import os
7 import signal
8 import socket
9 import time
10
```



```
11 SERVER_ADDRESS = (HOST, PORT) = '', 8888
12 REQUEST_QUEUE_SIZE = 5
13
14 def grim_reaper(signum, frame):
15     pid, status = os.wait()
16     print(
17         'Child {pid} terminated with status {status}'
18         '\n'.format(pid=pid, status=status)
19     )
20
21 def handle_request(client_connection):
22     request = client_connection.recv(1024)
23     print(request.decode())
24     http_response = b"""
25 HTTP/1.1 200 OK
26
27 Hello, World!
28 """
29     client_connection.sendall(http_response)
30     # sleep to allow the parent to loop over to 'accept' and block there
31     time.sleep(3)
32
33 def serve_forever():
34     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
35     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
36     listen_socket.bind(SERVER_ADDRESS)
37     listen_socket.listen(REQUEST_QUEUE_SIZE)
38     print('Serving HTTP on port {port} ...'.format(port=PORT))
39
40     signal.signal(signal.SIGCHLD, grim_reaper)
41
42     while True:
43         client_connection, client_address = listen_socket.accept()
44         pid = os.fork()
45         if pid == 0: # child
46             listen_socket.close() # close child copy
47             handle_request(client_connection)
48             client_connection.close()
49             os._exit(0)
50         else: # parent
51             client_connection.close()
52
53 if __name__ == '__main__':
54     serve_forever()
```

启动服务器：

```
1 $ python webserver3e.py
```

使用老朋友curl给修改后的并发服务器发送请求：

```
1 $ curl http://localhost:8888/hello
```

观察服务器：

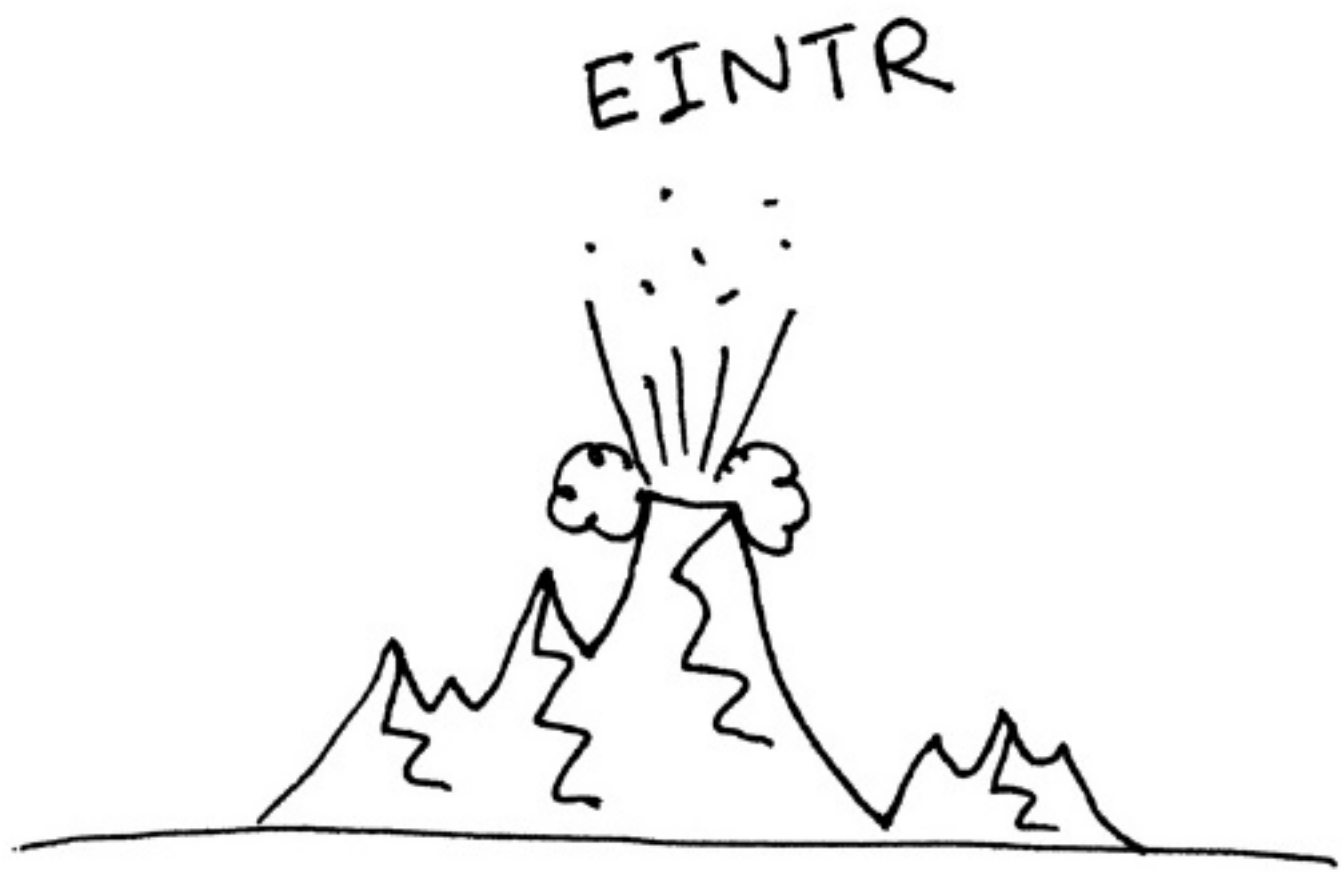
```
Serving HTTP on port 8888 ...
GET /hello HTTP/1.1
User-Agent: curl/7.35.0
Host: localhost:8888
Accept: */*

Child 9951 terminated with status 0

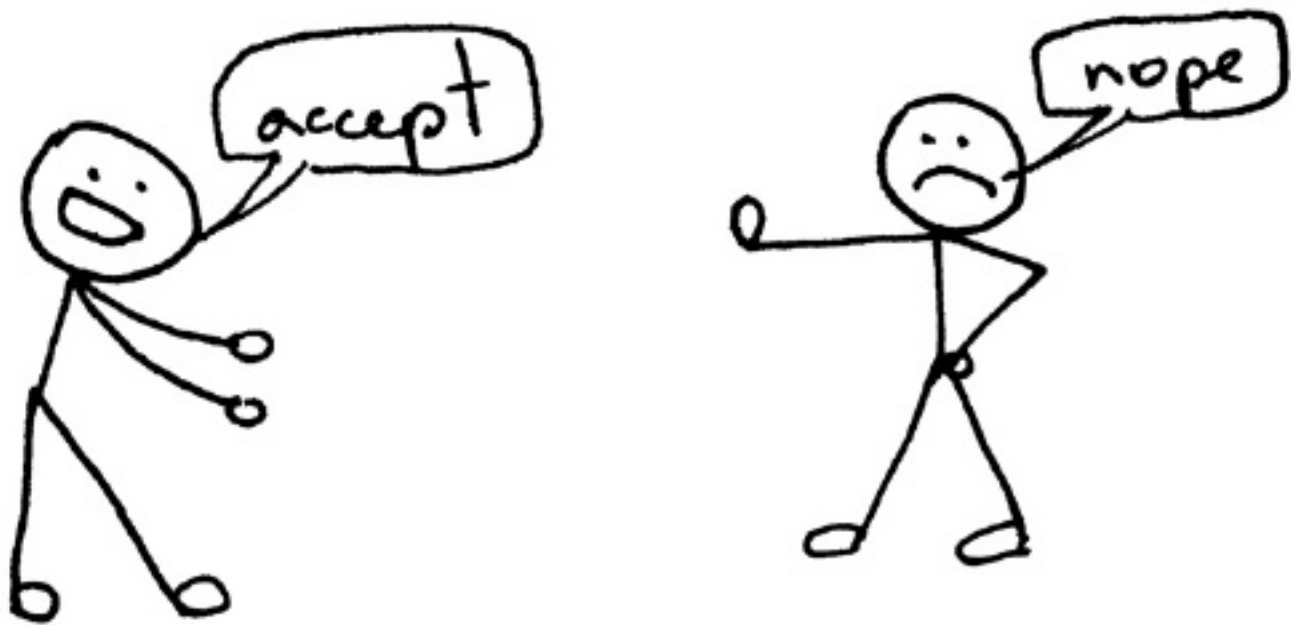
Traceback (most recent call last):
  File "webserver3e.py", line 62, in <module>
    serve_forever()
  File "webserver3e.py", line 51, in serve_forever
    client_connection, client_address = listen_socket.accept()
  File "/usr/lib/python2.7/socket.py", line 202, in accept
    sock, addr = self._sock.accept()
socket.error: [Errno 4] Interrupted system call
```



刚才发生了什么? accept调用失败了, 错误是EINTR。



当子进程退出, 引发SIGCHLD事件时, 父进程阻塞在accept调用, 这激活了事件处理器, 然后当事件处理器完成时, accept系统调用就中断了:



别着急, 这个问题很好解决. 你要做的就是重新调用accept。以下是修改后的代码:

```
1 #####
2 # Concurrent server - webserver3f.py #
3 # #
4 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #
5 #####
6 import errno
7 import os
8 import signal
9 import socket
10
11 SERVER_ADDRESS = (HOST, PORT) = '', 8888
12 REQUEST_QUEUE_SIZE = 1024
13
14 def grim_reaper(signum, frame):
15     pid, status = os.wait()
16
17 def handle_request(client_connection):
18     request = client_connection.recv(1024)
19     print(request.decode())
20     http_response = b"""
21 HTTP/1.1 200 OK
22
23 Hello, World!
24 """
25     client_connection.sendall(http_response)
26
27 def serve_forever():
28     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
29     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
30     listen_socket.bind(SERVER_ADDRESS)
31     listen_socket.listen(REQUEST_QUEUE_SIZE)
32     print('Serving HTTP on port {port} ...'.format(port=PORT))
33
34     signal.signal(signal.SIGCHLD, grim_reaper)
35
```

```
36 while True:
37     try:
38         client_connection, client_address = listen_socket.accept()
39     except IOError as e:
40         code, msg = e.args
41         # restart 'accept' if it was interrupted
42         if code == errno.EINTR:
43             continue
44         else:
45             raise
46
47     pid = os.fork()
48     if pid == 0: # child
49         listen_socket.close() # close child copy
50         handle_request(client_connection)
51         client_connection.close()
52         os._exit(0)
53     else: # parent
54         client_connection.close() # close parent copy and loop over
55
56 if __name__ == '__main__':
57     serve_forever()
```

启动修改后的webserver3f.py:

```
1 $ python webserver3f.py
```

使用curl给修改后的服务器发送请求:

```
1 $ curl http://localhost:8888/hello
```

看到了吗? 没有EINTR异常啦。现在, 验证一下吧, 没有僵尸了, 带wait的SIGCHLD事件处理器也能处理好子进程了。怎么验证呢? 只要运行ps命令, 看看没有Z+状态的进程(没有进程)。太棒啦! 没有僵尸在四周跳的感觉真安全呢!



- 如果fork了子进程并不wait它, 它就成了僵尸了。
- 使用SIGCHLD事件处理器来异步的wait终止了的子进程来获取它的终止状态
- 使用事件处理器时, 你要明白, 系统调用会被中断的, 你要做好准备对付这种情况

嗯, 目前为止, 一次都好。没有问题, 对吧? 好吧, 几乎滑。再次跑下webserver3f.py, 这次不用curl请求一次了, 改用client3.py来创建128个并发连接:

```
1 $ python client3.py --max-clients 128
```

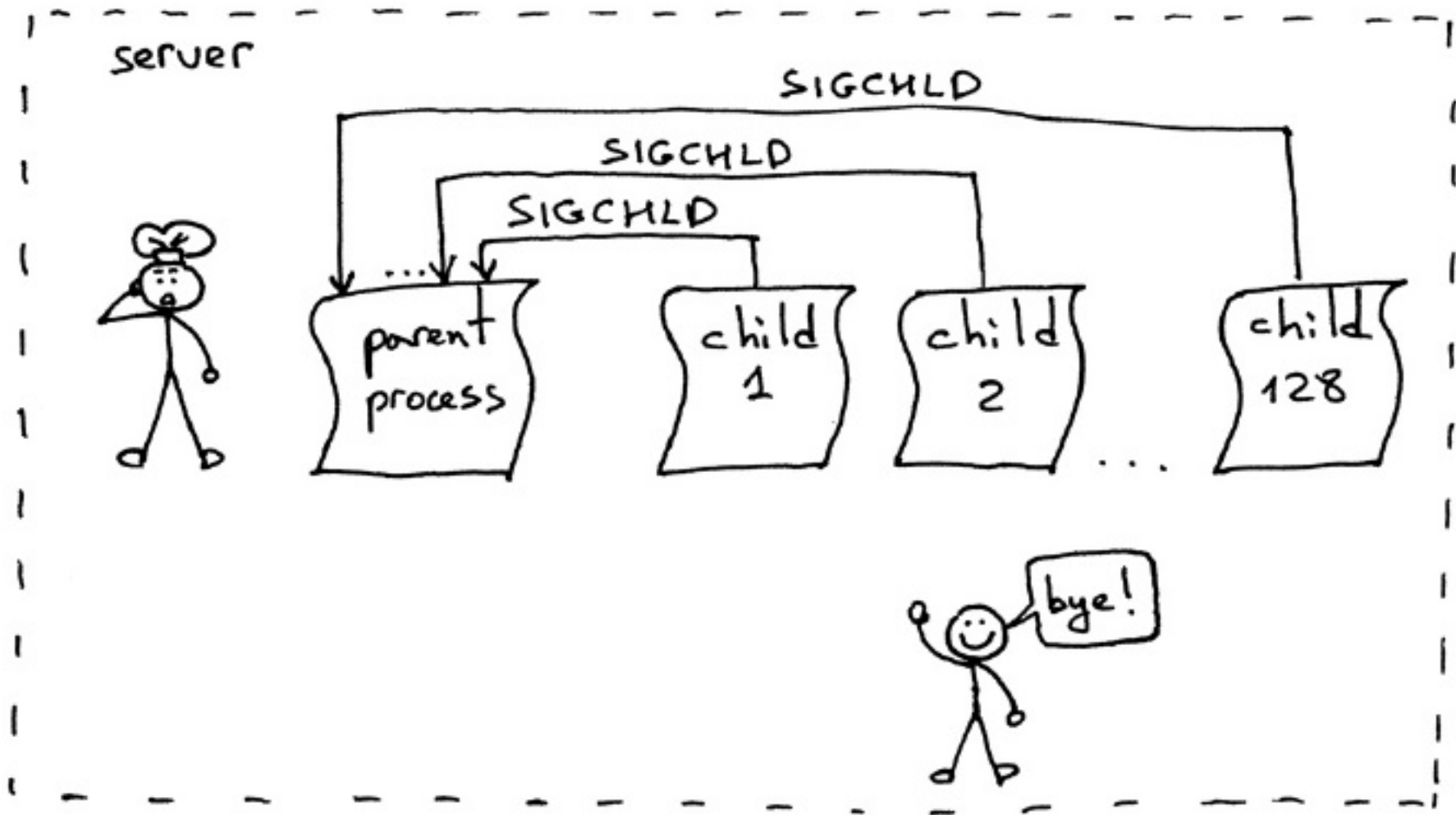
现在再运行ps命令

```
1 $ ps auxw | grep -i python | grep -v grep
```

看到了吧, 少年, 僵尸又回来了!



这次又出什么错了呢？当你运行128个并发客户端时，建立了128个连接，子进程处理了请求然后几乎同时终止了，这就引发了SIGCHLD信号洪水般的发给父进程。问题在于，信号没有排队，父进程错过了一些信号，导致了一些僵尸到处跑没人管：



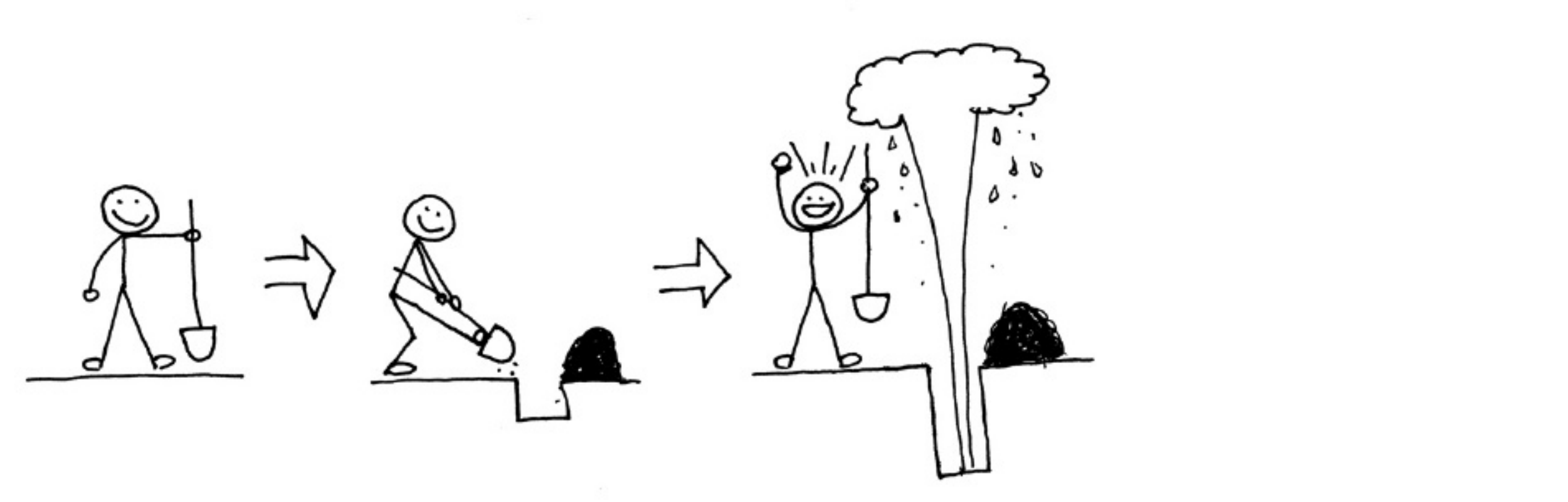
解决方案就是设置一个SIGCHLD事件处理器，但不用wait了，改用waitpid系统调用，带上WNOHANG参数，循环处理，确保所有的终止的子进程都被处理掉。以下是修改后的webserver3g.py：

```
1 #####
2 # Concurrent server - webserver3g.py
3 #
4 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X
5 #####
6 import errno
7 import os
8 import signal
9 import socket
10
11 SERVER_ADDRESS = (HOST, PORT) = '', 8888
12 REQUEST_QUEUE_SIZE = 1024
13
14 def grim_reaper(signum, frame):
15     while True:
16         try:
17             pid, status = os.waitpid(
18                 -1,
19                 os.WNOHANG # Do not block and return EWOULDBLOCK error
20             )
21         except OSError:
22             return
23
24         if pid == 0: # no more zombies
25             return
26
27 def handle_request(client_connection):
28     request = client_connection.recv(1024)
29     print(request.decode())
30     http_response = b"""
31 HTTP/1.1 200 OK
32
33 Hello, World!
34 """
35     client_connection.sendall(http_response)
```





去打好基础吧。质疑你已经知道的，保持深入研究。



如果你只学方法，你就依赖方法。但如果你学会原理，你可以发明自己的方法。—— 爱默生

以下是我挑出来对本文最重要的几本书。它们会帮你拓宽加深我提到的知识。我强烈建议你想言设法弄到这些书：从朋友那借也好，从本地图书馆借，或者从亚马逊买也行。它们是守护者：


1. Unix网络编程，卷1：socket网络API（第三版）
2. UNIX环境高级编程，第三版
3. Linux编程接口：Linux和UNIX系统编辑手册
4. TCP/IP详解，卷1：协议（第二版）
5. The Little Book of SEMAPHORES (2nd Edition): The Ins and Outs of Concurrency Control and Common Mistakes. Also available for free on the author’s site here.

 1 赞






 22 收藏

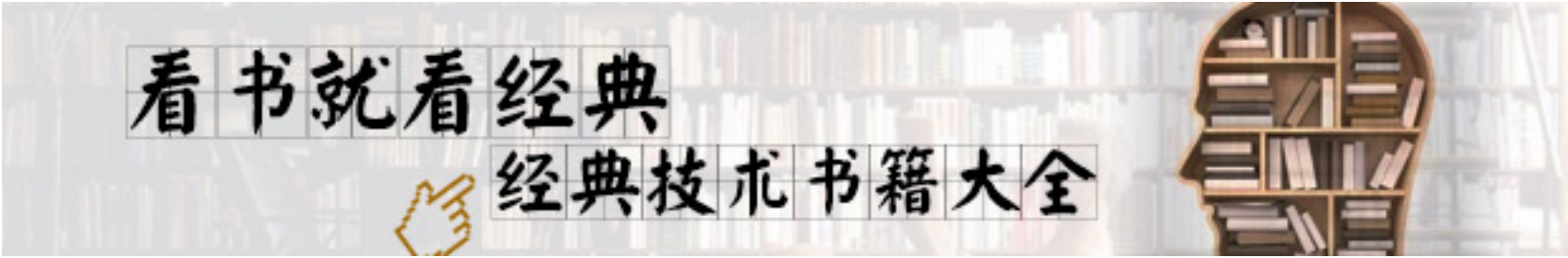


关于作者：高世界



我翻译得越多，发现知道的越少，我就要更多地翻译。论得的地的正确用法。我是php开发者，对python，c/c++，linux感兴趣。

 个人主页 ·  我的文章 ·  17 ·  











































## 相关文章

- [从零开始搭建论坛 \(2\)：Web服务器网关接口](#)
- [从零开始搭建论坛 \(1\)：Web服务器与Web框架](#) · [🔗 2](#)
- [一起写一个 Web 服务器 \(2\)](#) · [🔗 10](#)
- [一起写一个 Web 服务器 \(1\)](#) · [🔗 14](#)

## 可能感兴趣的话题

- [Java中的陷阱题-找奇数](#) · [🔗 2](#)
- [请大家各抒己见谈谈自己对现代战争中信息化的利用](#) · [🔗 1](#)
- [在北京做了5年开发，感觉无法突破自己，想去南方深圳发展，求建议](#) · [🔗 4](#)
- [父类和子类如何使用同一个装饰器呢、下面代码应该怎么改](#) · [🔗 2](#)
- [请杭州的程序员朋友帮推一份前端工程师的工作](#)
- [2016年链家网校招笔试（JAVA研发）：二叉树遍历](#)

最新评论

<div>  <div> <div>decli</div> <div>(  )</div> </div> </div> <div>2015/08/03</div>	<div> <div>非常不错的连载翻译，作者是个有心人，感谢翻译这么好的作品！</div> <div> <div> 赞</div> <div><a href="#">回复</a> </div> </div> </div>
<div>  <div> <div>高世界</div> <div>(  17 ·   )</div> </div> </div> <div>2015/08/03</div>	<div> <div>谢谢！你的名字看起来好好吃！</div> <div> <div> 赞</div> <div><a href="#">回复</a> </div> </div> </div>
<div>  <div> <div>leoliu</div> <div>(  1 )</div> </div> </div> <div>2015/08/03</div>	<div> <div>不错的翻译， 加油！</div> <div> <div> 赞</div> <div><a href="#">回复</a> </div> </div> </div>
<div>  <div> <div>英哲</div> <div>(  1 )</div> </div> </div> <div>2015/08/07</div>	<div> <div>好东西！</div> <div> <div> 赞</div> <div><a href="#">回复</a> </div> </div> </div>
<div>  <div> <div>JackPy</div> <div>(  1 ·   )</div> </div> </div> <div>2015/08/09</div>	<div> <div>翻译的好不好，就看它读起来流不流畅，很不错！</div> <div> <div> 赞</div> <div><a href="#">回复</a> </div> </div> </div>
<div>  <div> <div>Nicholas</div> <div>(  1 ·    )</div> </div> </div> <div>2016/01/03</div>	<div> <div>好文啊，我要去实践一下</div> <div> <div> 赞</div> <div><a href="#">回复</a> </div> </div> </div>
<div>  <div> <div>mie_mie_mie</div> <div>(  1 ·  )</div> </div> </div> <div>2016/08/18</div>	<div> <div>[然后“duang”的一声：代码克隆了自己，然后就有两个相同代码的实例同时运行。我想除了魔法无法做到，我是认真哒。] ~ ~ ~ 译者太有爱啦，赞赞赞！！！</div> <div> <div> 赞</div> <div><a href="#">回复</a> </div> </div> </div>
<div>  <div> <div>Ben_en_n</div> <div>(  )</div> </div> </div> <div>2016/10/25</div>	<div> <div>好文 让我回顾了之前学的东西，涉及的还挺多的</div> <div> <div> 赞</div> <div><a href="#">回复</a> </div> </div> </div>







[有没有非互联网行业的小伙伴自学编程...叫我小K咯](#) 发起 • 176 回复



[随着Python越来越火，自己也慢慢入了py...、O.o?](#) 发起 • 19 回复



[class的作用域](#)  
[day\\_day\\_up](#) 发起 • 3 回复



[明年找工作，求python大神指条明路太懒~](#) 发起 • 15 回复



[父类和子类如何使用同一个装饰器呢，...加瓦](#) 发起 • 2 回复



[如何使用多线程逐套下载多套图片？~桂~](#) 发起 • 1 回复



- [本周热门Python文章](#)
- [本月热门](#)
- [热门标签](#)

0 [python logging日志模块以及多进程...](#)

1 [Python标准库系列之Redis模块](#)



[Python工具资源](#)

[更多资源 »](#)



[Tryton：一个通用商务框架](#)  
[杂项](#)



[NLTK](#)：一个先进的用来处理自然语言数据的Python程序。  
[自然语言处理](#)



[PyMC](#)：马尔科夫链蒙特卡洛采样工具  
[科学计算与分析](#)



[statsmodels](#)：统计建模和计量经济学  
[科学计算与分析](#)



[Pylearn2](#)：一个基于Theano的机器学习库  
[机器学习](#) · [🔍1](#)

## 关于 Python 频道

Python频道分享 Python 开发技术、相关的行业动态。

- 快速链接
- [网站使用指南](#)»
  - [加入我们](#)»
  - [问题反馈与求助](#)»
  - [网站积分规则](#)»
  - [网站声望规则](#)»

## 关注我们

新浪微博：[@Python开发者](#)  
RSS：[订阅地址](#)

推荐微信号



合作联系  
Email：[bd@jobbole.com](mailto:bd@jobbole.com)  
QQ： 2302462408 （加好友请注明来意）

## 更多频道

- [小组](#) - 好的话题、有启发的回复、值得信赖的圈子
- [头条](#) - 分享和发现有价值的内容与观点
- [相亲](#) - 为IT单身男女服务的征婚传播平台
- [资源](#) - 优秀的工具资源导航
- [翻译](#) - 翻译传播优秀的外文文章
- [文章](#) - 国内外的精选文章
- [设计](#) - UI,网页，交互和用户体验
- [iOS](#) - 专注iOS技术分享
- [安卓](#) - 专注Android技术分享
- [前端](#) - JavaScript, HTML5, CSS
- [Java](#) - 专注Java技术分享
- [Python](#) - 专注Python技术分享

