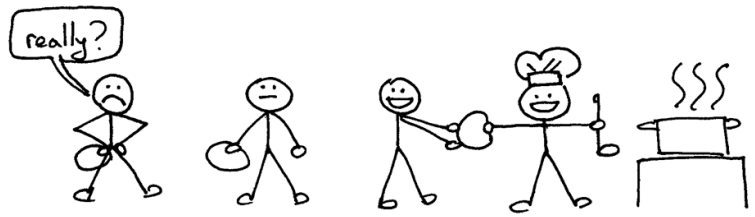
自己动手开发一个 Web 服务器（三）

2016-1-3 10:00 收藏: 1

来源：编程派 参考原文：http://ruslanspivak.com/lsbaws-part3/ 作者： Ruslan
编译文章：http://codingpy.com/article/build-a-simple-web-server-part-three/
文章地址：https://linux.cn/article-6817-1.html

在第二部分中，你开发了一个能够处理HTTPGET请求的简易WSGI服务器。在上一篇的最后，我问了你一个问题：“怎样让服务器一次处理多个请求？”读完本文，你就能够完美地回答这个问题。接下来，请你做好准备，因为本文的内容非常多，节奏也很快。文中的所有代码都可以在**Github仓库 <https://github.com/rspivak/lsbaws/blob/master/part3/>** 下载。

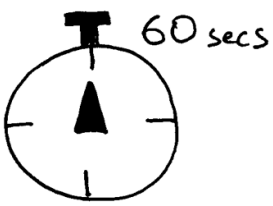
首先，我们简单回忆一下简易网络服务器是如何实现的，服务器要处理客户端的请求需要哪些条件。你在前面两部分文章中开发的服务器，是一个 迭代式(iterative)服务器(server) ，还只能一次处理一个客户端请求。只有在处理完当前客户端请求之后，它才能接收新的客户端连接。这样，有些客户端就必须要等待自己的请求被处理了，而对于流量大的服务器来说，等待的时间就会特别长。



下面是迭代式服务器 `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

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 5


def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)


def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    while True:
        client_connection, client_address = listen_socket.accept()
        handle_request(client_connection)
        client_connection.close()

if __name__ == '__main__':
    serve_forever()
```

如果想确认这个服务器每次只能处理一个客户端的请求，我们对上述代码作简单修改，在向客户端返回响应之后，增加60秒的延迟处理时间。这个修改只有一行代码，即告诉服务器在返回响应之后睡眠60秒。



下面就是修改之后的服务器代码：

```
 ##################################################################
# Iterative server - webserver3b.py                              #
#                                                                #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X  #
#                                                                #
# - Server sleeps for 60 seconds after sending a response to a client  #
##################################################################
import socket
```

```
import time

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 5


def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)
    time.sleep(60)  # sleep and block the process for 60 seconds


def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    while True:
        client_connection, client_address = listen_socket.accept()
        handle_request(client_connection)
        client_connection.close()

if __name__ == '__main__':
    serve_forever()
```

接下来，我们启动服务器：

```
$ python webserver3b.py
```

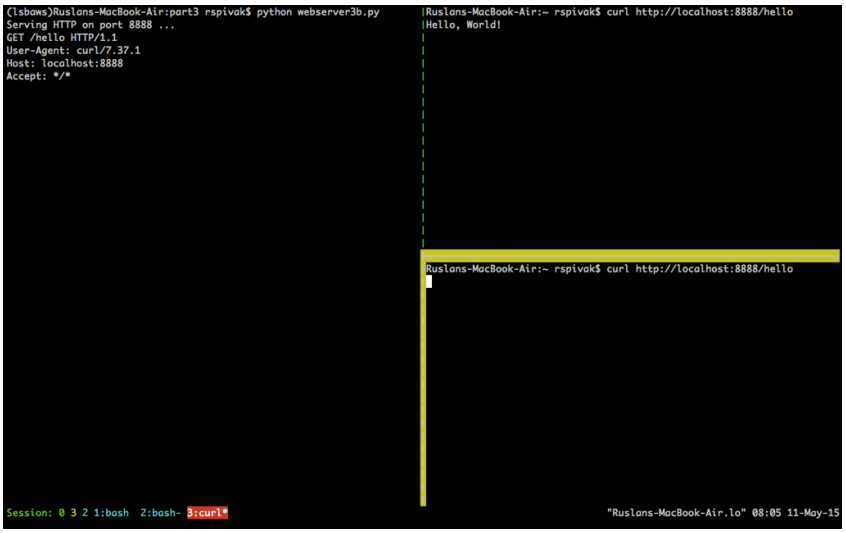
现在，我们打开一个新的终端窗口，并运行 `curl` 命令。你会立刻看到屏幕上打印出了“Hello, World!”这句话：

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

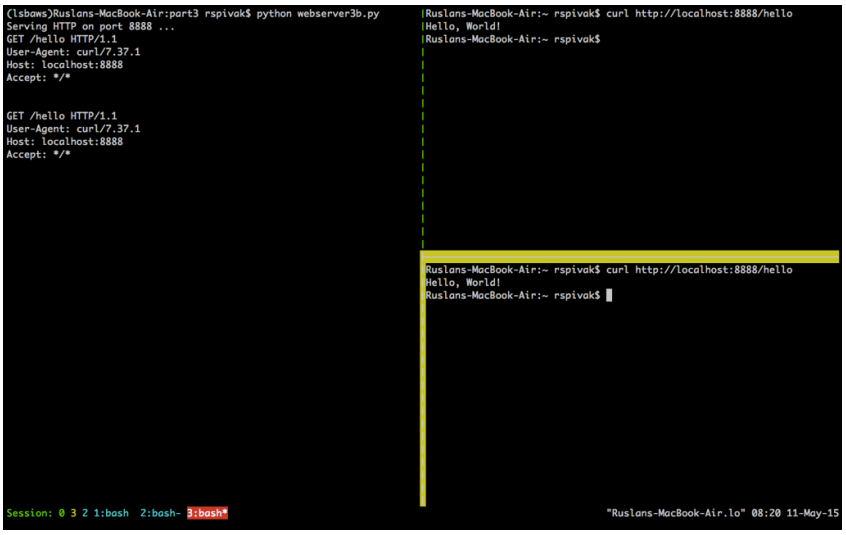
接着我们立刻再打开一个终端窗口，并运行 `curl` 命令：

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

如果你在60秒了完成了上面的操作，那么第二个 `curl` 命令应该不会立刻产生任何输出结果，而是处于挂死（hang）状态。服务器也不会在标准输出中打印这个新请求的正文。下面这张图就是我在自己的Mac上操作时的结果（右下角那个边缘高亮为黄色的窗口，显示的就是第二个 `curl` 命令挂死）：

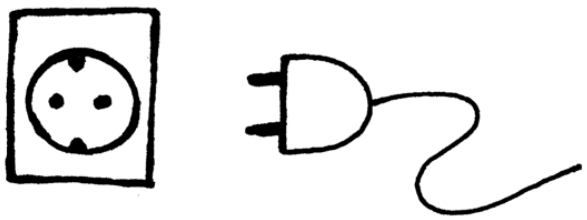


当然，你等了足够长时间之后（超过60秒），你会看到第一个 `curl` 命令结束，然后第二个 `curl` 命令会在屏幕上打印出“Hello, World!”，之后再挂死60秒，最后才结束：



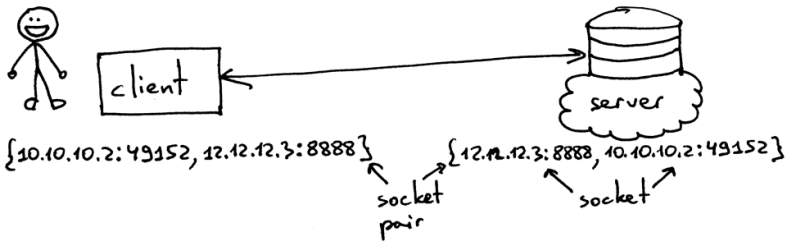
这背后的实现方式是，服务器处理完第一个 `curl` 客户端请求后睡眠60秒，才开始处理第二个请求。这些步骤是线性执行的，或者说迭代式一步一步执行的。在我们这个实例中，则是一次一个请求这样处理。

接下来，我们简单谈谈客户端与服务器之间的通信。为了让两个程序通过网络进行通信，二者均必须使用套接字。你在前两章中也看到过套接字，但到底什么是套接字？



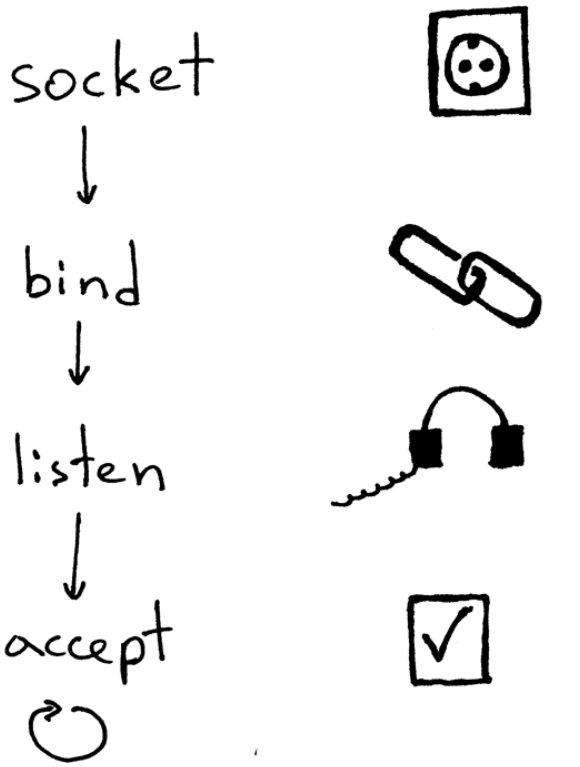
套接字是 通信端点（communication endpoint）的抽象形式，可以让一个程序通过 文件描述符（file descriptor）与另一个程序进行通信。在本文中，我只讨论Linux/Mac OS X平台上的TCP/IP套接字。其中，尤为重要的一个概念就是TCP 套接字对（socket pair）。

> TCP连接所使用的套接字对是一个 4元组 ，包括本地IP地址、本地端口、外部IP地址和外部端口。一个网络中的每一个TCP连接，都拥有独特的套接字对。IP地址和端口号通常被称为一个套接字，二者一起标识了一个网络端点。



因此， `{10.10.10.2:49152, 12.12.12.3:8888}` 元组组成了一个套接字对，代表客户端侧TCP连接的两个唯一端点， `{12.12.12.3:8888, 10.10.10.2:49152}` 元组组成另一个套接字对，代表服务器侧TCP连接的两个同样端点。构成TCP连接中服务器端点的两个值分别是IP地址 `12.12.12.3` 和端口号 `8888` ，它们在这里被称为一个套接字（同理，客户端端点的两个值也是一个套接字）。

服务器创建套接字并开始接受客户端连接的标准流程如下：



1. 服务器创建一个TCP/IP套接字。通过下面的Python语句实现：

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

2. 服务器可以设置部分套接字选项（这是可选项，但你会发现上面那行服务器代码就可以确保你重启服务器之后，服务器会继续使用相同的地址）。

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

3. 然后，服务器绑定地址。绑定函数为套接字指定一个本地协议地址。调用绑定函数时，你可以单独指定端口号或IP地址，也可以同时指定两个参数，甚至不提供任何参数也没问题。

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

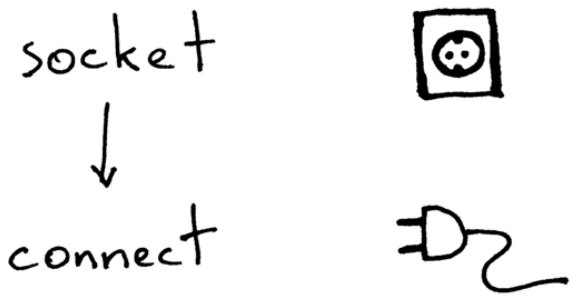
4. 接着，服务器将该套接字变成一个侦听套接字：

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

`listen` 方法只能由服务器调用，执行后会告知服务器应该接收针对该套接字的连接请求。

完成上面四步之后，服务器会开启一个循环，开始接收客户端连接，不过一次只接收一个连接。当有连接请求时， `accept` 方法会返回已连接的客户端套接字。然后，服务器从客户端套接字读取请求数据，在标准输出中打印数据，并向客户端返回消息。最后，服务器会关闭当前的客户端连接，这时服务器又可以接收新的客户端连接了。

要通过TCP/IP协议与服务器进行通信，客户端需要作如下操作：



下面这段示例代码，实现了客户端连接至服务器，发送请求，并打印响应内容的过程：

```
import socket

# create a socket and connect to a server
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 8888))

# send and receive some data
sock.sendall(b'test')
data = sock.recv(1024)
print(data.decode())
```

在创建套接字之后，客户端需要与服务器进行连接，这可以通过调用 `connect` 方法实现：

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

客户端只需要提供远程IP地址或主机名，以及服务器的远程连接端口号即可。

你可能已经注意到，客户端不会调用 `bind` 和 `accept` 方法。不需要调用 `bind` 方法，是因为客户端不关心本地IP地址和本地端口号。客户端调用 `connect` 方法时，系统内核中的TCP/IP栈会自动指定本地IP地址和本地端口。本地端口也被称为 ephemeral port 临时端口 。

服务器端有部分端口用于连接熟知的服务，这种端口被叫做“熟知端口”，例如，80用于HTTP传输服务，22用于SSH协议传输。接下来，我们打开Python shell，向在本地运行的服务器发起一个客户端连接，然后查看系统内核为你创建的客户端套接字指定了哪个临时端口（在进行下面的操作之前，请先运行 `webserver3a.py` 或 `webserver3b.py` 文件，启动服务器）：

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

在上面的示例中，我们看到内核为套接字指定的临时端口是60589。

在开始回答第二部分 **<http://codingpy.com/article/build-a-simple-web-server-part-two/>** 最后提的问题之前，我需要快速介绍一些其他的重要概念。稍后你就会明白我为什么要这样做。我要介绍的重要概念就是 进程 和 文件描述符 。

什么是进程？进程就是正在执行的程序的一个实例。举个例子，当服务器代码执行的时候，这些代码就被加载至内存中，而这个正在被执行的服务器的实例就叫做进程。系统内核会记录下有关进程的信息——包括进程ID，以便进行管理。所以，当你运行迭代式服务器 `webserver3a.py` 或 `webserver3b.py` 时，你也就开启了一个进程。

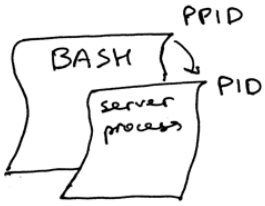
我们在终端启动 `webserver3a.py` 服务器：

```
$ python webserver3b.py
```

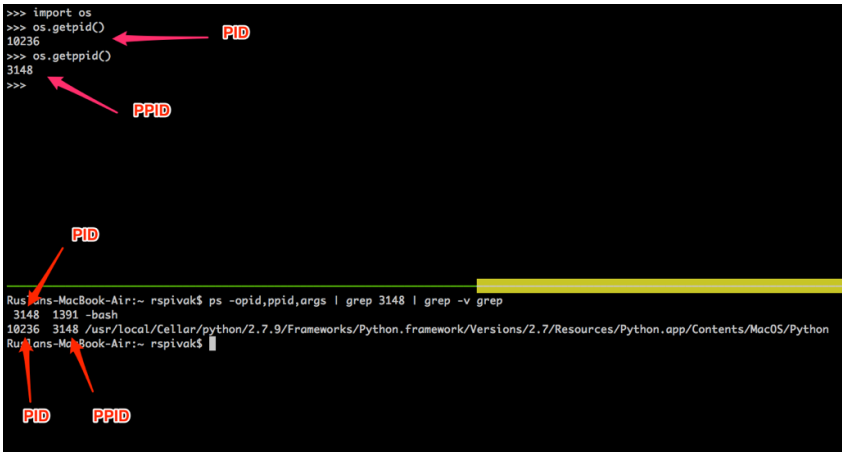
然后，我们在另一个终端窗口中，使用 `ps` 命令来获取上面那个服务器进程的信息：

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

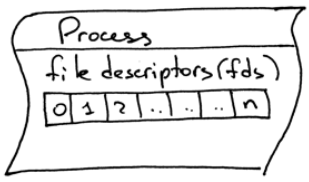
从 `ps` 命令的结果，我们可以看出你的确只运行了一个Python进程 `webserver3b` 。进程创建的时候，内核会给它指定一个进程ID——PID。在UNIX系统下，每个用户进程都会有一个 父进程 ，而这个父进程也有自己的进程ID，叫做父进程ID，简称PPID。在本文中，我默认大家使用的是BASH，因此当你启动服务器的时候，系统会创建服务器进程，指定一个PID，而服务器进程的父进程PID则是BASH shell进程的PID。



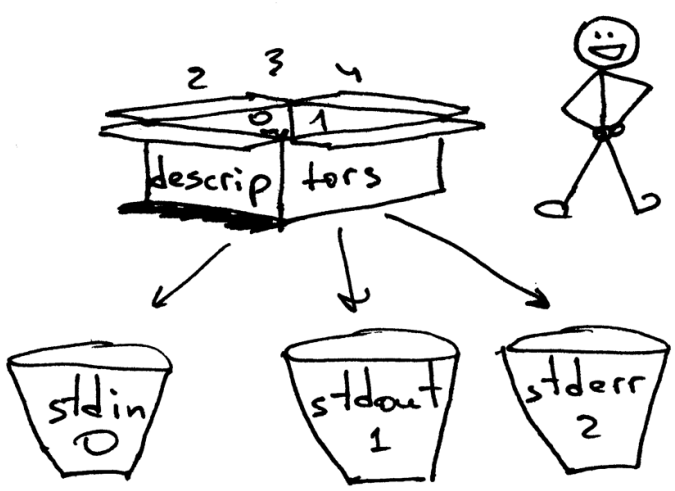
接下来请自己尝试操作一下。再次打开你的Python shell程序，这会创建一个新进程，然后我们通过 `os.gepid()` 和 `os.getppid()` 这两个方法，分别获得Python shell进程的PID及它的父进程PID（即BASH shell程序的PID）。接着，我们打开另一个终端窗口，运行 `ps` 命令， `grep` 检索刚才所得到的PPID（父进程ID，本操作时的结果是3148）。在下面的截图中，你可以看到我在Mac OS X上的操作结果：



另一个需要掌握的重要概念就是 文件描述符 。那么，到底什么是文件描述符？文件描述符指的就是当系统打开一个现有文件、创建一个新文件或是创建一个新的套接字之后，返回给进程的那个正整型数。系统内核通过文件描述符来追踪一个进程所打开的文件。当你需要读写文件时，你也通过文件描述符说明。Python语言中提供了用于处理文件（和套接字）的高层级对象，所以你不必直接使用文件描述符来指定文件，但是从底层实现来看，UNIX系统中就是通过它们的文件描述符来确定文件和套接字的。



一般来说，UNIX shell会将文件描述符0指定给进程的标准输出，文件描述富1指定给进程的标准输出，文件描述符2指定给标准错误。



正如我前面提到的那样，即使Python语言提供了高层及的文件或类文件对象，你仍然可以对文件对象使用 `fileno()` 方法，来获取该文件相应的文件描述符。我们回到Python shell中来试验一下。

```
 >>> import sys
>>> sys.stdin
<open file '<stdin>', mode 'r' at 0x102beb0c0>
>>> sys.stdin.fileno()
0
>>> sys.stdout.fileno()
1
>>> sys.stderr.fileno()
2
```

在Python语言中处理文件和套接字时，你通常只需要使用高层及的文件/套接字对象即可，但是有些时候你也可能需要直接使用文件描述符。下面这个示例演示了你如何通过 `write()` 方法向标准输出中写入一个字符串，而这个 `write` 方法就接受文件描述符作为自己的参数：

```
 >>> import sys
>>> import os
>>> res = os.write(sys.stdout.fileno(), 'hello\n')
hello
```

还有一点挺有意思——如果你知道Unix系统下一切都是文件，那么你就不会觉得奇怪了。当你在Python中创建一个套接字后，你获得的是一个套接字对象，而不是一个正整型数，但是你还是可以和上面演示的一样，通过 `fileno()` 方法直接访问这个套接字的文件描述符。

```
 >>> import socket
>>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> sock.fileno()
3
```

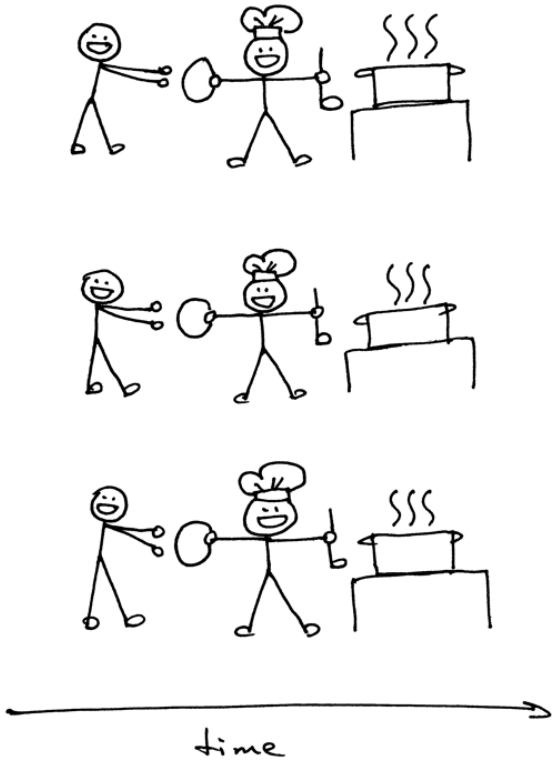
我还想再说一点：不知道大家有没有注意到，在迭代式服务器 `webserver3b.py` 的第二个示例中，我们的服务器在处理完请求后睡眠60秒，但是在睡眠期间，我们仍然可以通过 `curl` 命令与服务器建立连接？当然， `curl` 命令并没有立刻输出结果，只是出于挂死状态，但是为什么服务器既然没有接受新的连接，客户端也没有立刻被拒绝，而是仍然继续连接至服务器呢？这个问题的答案在于套接字对象的 `listen` 方法，以及它使用的 `BACKLOG` 参数。在示例代码中，这个参数的值被我设置为 `REQUEST_QUEQUE_SIZE` 。 `BACKLOG` 参数决定了内核中外部连接请求的队列大小。当 `webserver3b.py` 服务器睡眠时，你运行的第二个 `curl` 命令之所以能够连接服务器，是因为连接请求队列仍有足够的位置。

虽然提高 `BACKLOG` 参数的值并不会让你的服务器一次处理多个客户端请求，但是业务繁忙的服务器也应该设置一个较大的 `BACKLOG` 参数值，这样 `accept` 函数就可以直接从队列中获取新连接，立刻开始处理客户端请求，而不是还要花时间等待连接建立。

呜呼！到目前为止，已经给大家介绍了很多知识。我们现在快速回顾一下之前的内容。

- 迭代式服务器
- 服务器套接字创建流程（socket, bind, listen, accept）
- 客户端套接字创建流程（socket, connect）
- 套接字对
- 套接字
- 临时端口 与 熟知端口
- 进程
- 进程ID（PID），父进程ID（PPID）以及父子关系
- 文件描述符
- 套接字对象的 `listen` 方法中 `BACKLOG` 参数的意义

现在，我可以开始回答第二部分留下的问题了：如何让服务器一次处理多个请求？换句话说，如何开发一个并发服务器？



在Unix系统中开发一个并发服务器的最简单方法，就是调用系统函数 `fork()` 。

下面就是崭新的 `webserver3c.py` 并发服务器，能够同时处理多个客户端请求：

```
 ###########################################################################
# Concurrent server - webserver3c.py                                      #
#                                                                         #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X        #
#                                                                         #
# - Child process sleeps for 60 seconds after handling a client's request #
# - Parent and child processes close duplicate descriptors                #
#                                                                         #
###########################################################################
import os
import socket
import time

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 5


def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(
        'Child PID: {pid}. Parent PID {ppid}'.format(
            pid=os.getpid(),
            ppid=os.getppid(),
        )
    )
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)
    time.sleep(60)


def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))
    print('Parent PID (PPID): {pid}\n'.format(pid=os.getpid()))

    while True:
        client_connection, client_address = listen_socket.accept()
        pid = os.fork()
        if pid == 0:  # child
            listen_socket.close()  # close child copy
            handle_request(client_connection)
```

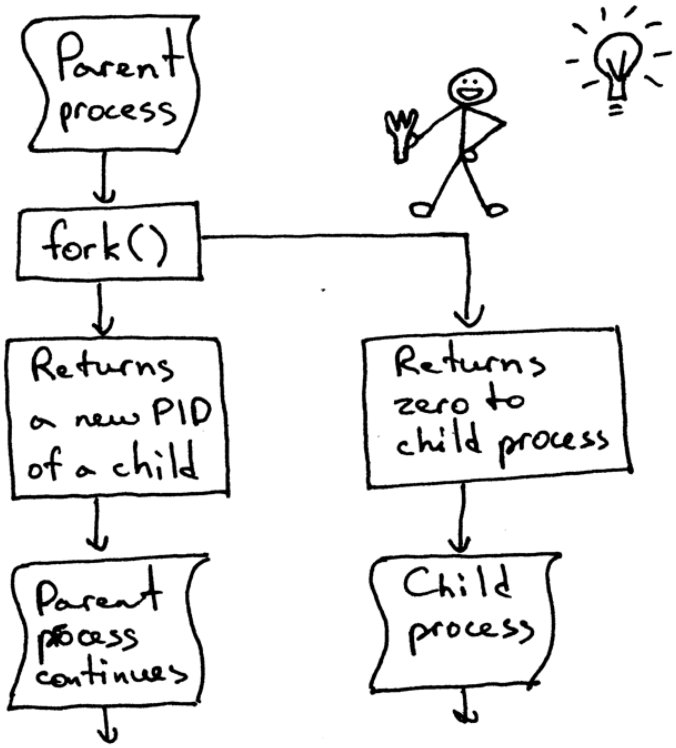
```
            client_connection.close()
            os._exit(0)  # child exits here
        else:  # parent
            client_connection.close()  # close parent copy and loop over

if __name__ == '__main__':
    serve_forever()
```

在讨论 `fork` 的工作原理之前，请测试一下上面的代码，亲自确认一下服务器是否能够同时处理多个客户端请求。我们通过命令行启动上面这个服务器：

```
$ python webserver3c.py
```

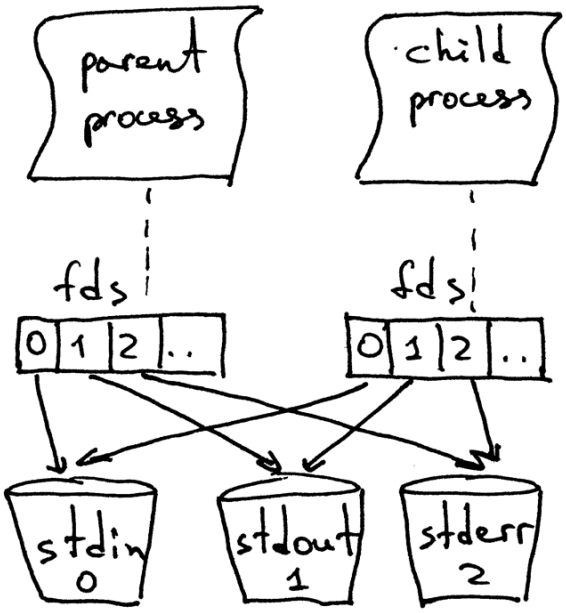
然后输入之前迭代式服务器示例中的两个 `curl` 命令。现在，即使服务器子进程在处理完一个客户端请求之后会睡眠60秒，但是并不会影响其他客户端，因为它们由不同的、完全独立的进程处理。你应该可以立刻看见 `curl` 命令输出“Hello, World”，然后挂死60秒。你可以继续运行更多的 `curl` 命令，所有的命令都会输出服务器的响应结果——“Hello, World”，不会有任何延迟。你可以试试。

关于 `fork()` 函数有一点最为重要，就是你调用 `fork` 一次，但是函数却会返回两次：一次是在父进程里返回，另一次是在子进程中返回。当你 `fork` 一个进程时，返回给子进程的PID是0，而 `fork` 返回给父进程的则是子进程的PID。



我还记得，第一次接触并使用 `fork` 函数时，自己感到非常不可思议。我觉得这就好像一个魔法。之前还是一个线性的代码，突然一下子克隆了自己，出现了并行运行的相同代码的两个实例。我当时真的觉得这和魔法也差不多了。

当父进程 `fork` 一个新的子进程时，子进程会得到父进程文件描述符的副本：



你可能也注意到了，上面代码中的父进程关闭了客户端连接：

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

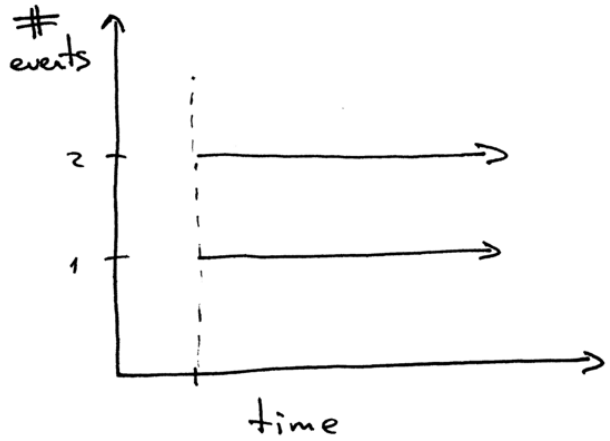
那为什么父进程关闭了套接字之后，子进程却仍然能够从客户端套接字中读取数据呢？答案就在上面的图片里。系统内核根据文件描述符计数 (descriptor reference counts) 来决定是否关闭套接字。系统只有在描述符计数变为0时，才会关闭套接字。当你的服务器创建一个子进程时，子进程就会获得父进程文件描述符的副本，系统内核则会增加这些文件描述符的计数。在一个父进程和一个子进程的情况下，客户端套接字的文件描述符计数为2。当上面代码中的父进程关闭客户端连接套接字时，只是让套接字的计数减为1，还不够让系统关闭套接字。子进程同样关闭了父进程侦听套接字的副本，因为子进程不关心要不要接收新的客户端连接，只关心如何处理连接成功的客户端所发出的请求。

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

稍后，我会给大家介绍如果不关闭重复的描述符的后果。

从上面并行服务器的源代码可以看出，服务器父进程现在唯一的作用，就是接受客户端连接， `fork` 一个新的子进程来处理该客户端连接，然后回到循环的起点，准备接受其他的客户端连接，仅此而已。服务器父进程并不会处理客户端请求，而是由它的子进程来处理。

谈得稍远一点。我们说两个事件是并行时，到底是什么意思？



我们说两个事件是并行的，通常指的是二者同时发生。这是简单的定义，但是你应该牢记它的严格定义：

> 如果你不能分辨出哪个程序会先执行，那么二者就是并行的。

现在又到了回顾目前已经介绍的主要观点和概念。

- Unix系统中开发并行服务器最简单的方法，就是调用 `fork()` 函数
- 当一个进程 `fork` 新进程时，它就成了新创建进程的父进程
- 在调用 `fork` 之后，父进程和子进程共用相同的文件描述符
- 系统内核通过描述符计数来决定是否关闭文件/套接字
- 服务器父进程的角色：它现在所做的只是接收来自客户端的新连接， `fork` 一个子进程来处理该客户端的请求，然后回到循环的起点，准备接受新的客户端连接

接下来，我们看看如果不关闭父进程和子进程中的重复套接字描述符，会发生什么情况。下面的并行服务器（webserver3d.py）作了一些修改，确保服务器不关闭重复的：

```
 ###########################################################################
# Concurrent server - webserver3d.py                                      #
#                                                                         #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X        #
###########################################################################
import os
import socket

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 5


def handle_request(client_connection):
    request = client_connection.recv(1024)
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
```

```
    client_connection.sendall(http_response)


def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    clients = []
    while True:
        client_connection, client_address = listen_socket.accept()
        # store the reference otherwise it's garbage collected
        # on the next loop run
        clients.append(client_connection)
        pid = os.fork()
        if pid == 0:  # child
            listen_socket.close()  # close child copy
            handle_request(client_connection)
            client_connection.close()
            os._exit(0)  # child exits here
        else:  # parent
            # client_connection.close()
            print(len(clients))

if __name__ == '__main__':
    serve_forever()
```

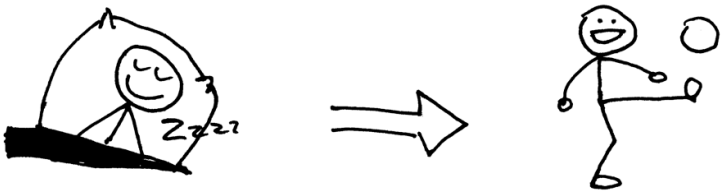
启动服务器：

```
$ python webserver3d.py
```

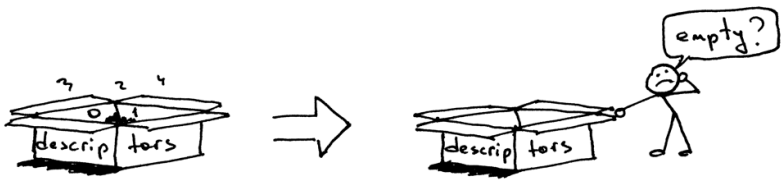
然后通过 `curl` 命令连接至服务器：

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

我们看到， `curl` 命令打印了并行服务器的响应内容，但是并没有结束，而是继续挂死。服务器出现了什么不同情况吗？服务器不再继续睡眠60秒：它的子进程会积极处理客户端请求，处理完成后就关闭客户端连接，然后结束运行，但是客户端的 `curl` 命令却不会终止。



那么为什么 `curl` 命令会没有结束运行呢？原因在于重复的 文件描述符 (duplicate file descriptor) 。当子进程关闭客户端连接时，系统内核会减少客户端套接字的计数，变成了1。服务器子进程结束了，但是客户端套接字并没有关闭，因为那个套接字的描述符计数并没有变成0，导致系统没有向客户端发送 终止包 (termination packet) （用TCP/IP的术语来说叫做FIN），也就是说客户端仍然在线。但是还有另一个问题。如果你一直运行的服务器不去关闭重复的文件描述符，服务器最终就会耗光可用的文件服务器：



按下 `Control-C` ，关闭 `webserver3d.py` 服务器，然后通过shell自带的 `ulimit` 命令查看服务器进程可以使用的默认资源：

```
 $ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 3842
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 3842
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

从上面的结果中，我们可以看到：在我这台Ubuntu电脑上，服务器进程可以使用的文件描述符（打开的文件）最大数量为1024。

现在，我们来看看如果服务器不关闭重复的文件描述符，服务器会不会耗尽可用的文件描述符。我们在现有的或新开的终端窗口里，将服务器可以使用的最大文件描述符数量设置为256：

```
 $ ulimit -n 256
```

在刚刚运行了 `$ ulimit -n 256` 命令的终端里，我们开启 `webserver3d.py` 服务器：

```
 $ python webserver3d.py
```

然后通过下面的 `client3.py` 客户端来测试服务器。

```
 ##################################################################
# Test client - client3.py                                       #
#                                                                #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X  #
##################################################################
import argparse
import errno
import os
import socket


SERVER_ADDRESS = 'localhost', 8888
REQUEST = b"""\
GET /hello HTTP/1.1
Host: localhost:8888
```

```
    """


    def main(max_clients, max_conns):
        socks = []
        for client_num in range(max_clients):
            pid = os.fork()
            if pid == 0:
                for connection_num in range(max_conns):
                    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                    sock.connect(SERVER_ADDRESS)
                    sock.sendall(REQUEST)
                    socks.append(sock)
                    print(connection_num)
                    os._exit(0)


    if __name__ == '__main__':
        parser = argparse.ArgumentParser(
            description='Test client for LSBAWS.',
            formatter_class=argparse.ArgumentDefaultsHelpFormatter,
        )
        parser.add_argument(
            '--max-conns',
            type=int,
            default=1024,
            help='Maximum number of connections per client.'
        )
        parser.add_argument(
            '--max-clients',
            type=int,
            default=1,
            help='Maximum number of clients.'
        )
        args = parser.parse_args()
        main(args.max_clients, args.max_conns)
```

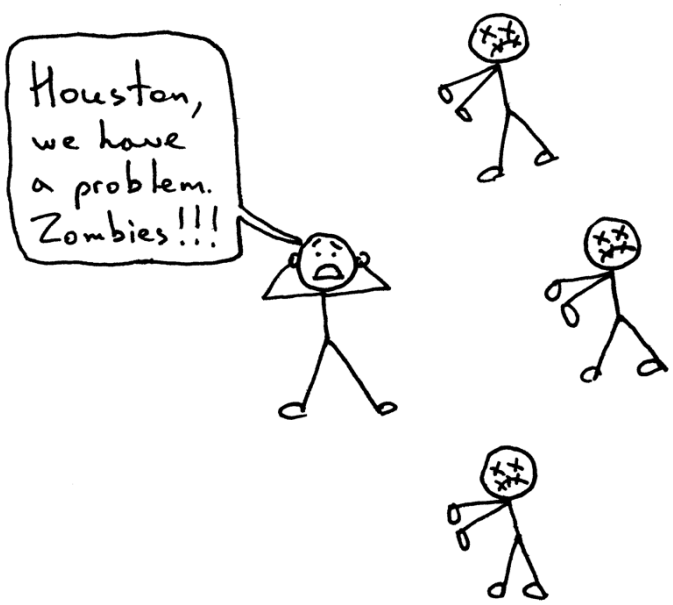
打开一个新终端窗口，运行 `client3.py` ，并让客户端创建300个与服务器的并行连接：

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

很快你的服务器就会崩溃。下面是我的虚拟机上抛出的异常情况：

```
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
```

问题很明显——服务器应该关闭重复的描述符。但即使你关闭了这些重复的描述符，你还没有彻底解决问题，因为你的服务器还存在另一个问题，那就是僵尸进程！



没错，你的服务器代码确实会产生僵尸进程。我们来看看这是怎么回事。再次运行服务器：

```
$ python webserver3d.py
```

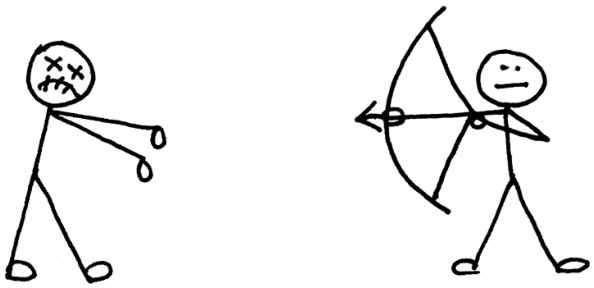
在另一个终端窗口中运行下面的 `curl` 命令：

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

现在，我们运行 `ps` 命令，看看都有哪些正在运行的Python进程。下面是我的Ubuntu虚拟机中的结果：

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

我们发现，第二行中显示的这个进程的PID为9102，状态是Z+，而进程的名称叫做 `<defunct>` 。这就是我们要找的僵尸进程。僵尸进程的问题在于你无法杀死它们。



即使你试图通过 `$ kill -9` 命令杀死僵尸进程，它们还是会存活下来。你可以试试看。

到底什么是僵尸进程，服务器又为什么会创建这些进程？僵尸进程其实是已经结束了的进程，但是它的父进程并没有等待进程结束，所以没有接收到进程结束的状态信息。当子进程在父进程之前退出，系统就会将子进程变成一个僵尸进程，保留原子进程的部分信息，方便父进程之后获取。系统所保留的信息通常包括进程ID、进程结束状态和进程的资源使用情况。好吧，这样说僵尸进程也有自己存在的理由，但是如果服务器不处理好这些僵尸进程，系统就会堵塞。我们来看看是否如此。

首先，停止正在运行的服务器，然后在新终端窗口中，使用 `ulimit` 命令将最大用户进程设置为400（还要确保将打开文件数量限制设置到一个较高的值，这里我们设置为500）。

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

然后在同一个窗口中启动 `webserver3d.py` 服务器：

```
$ python webserver3d.py
```

在新终端窗口中，启动客户端 `client3.py` ，让客户端创建500个服务器并行连接：

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

结果，我们发现很快服务器就因为OSError而崩溃：这个异常指的是暂时没有足够的资源。服务器试图创建新的子进程时，由于已经达到了系统所允许的最大可创建子进程数，所以抛出这个异常。下面是我的虚拟机上的报错截图。

```
186
187
188
189
190
191
Traceback (most recent call last):
  File "webserver3d.py", line 58, in <module>
    serve_forever()
  File "webserver3d.py", line 47, in serve_forever
    pid = os.fork()
OSError: [Errno 11] Resource temporarily unavailable
```

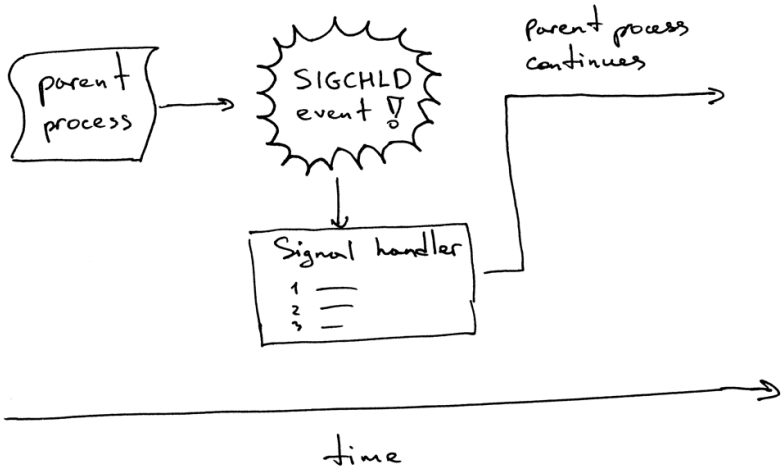
你也看到了，如果长期运行的服务器不处理好僵尸进程，将会出现重大问题。稍后我会介绍如何处理僵尸进程。

我们先回顾一下目前已经学习的知识点：

- 如果你不关闭重复的文件描述符，由于客户端连接没有中断，客户端程序就不会结束。
- 如果你不关闭重复的文件描述符，你的服务器最终会消耗完可用的文件描述符（最大打开文件数）
- 当你 `fork` 一个子进程后，如果子进程在父进程之前退出，而父进程又没有等待进程，并获取它的结束状态，那么子进程就会变成僵尸进程。
- 僵尸进程也需要消耗资源，也就是内存。如果不处理好僵尸进程，你的服务器最终会消耗完可用的进程数（最大用户进程数）。
- 你无法杀死僵尸进程，你需要等待子进程结束。

那么，你要怎么做才能处理掉僵尸进程呢？你需要修改服务器代码，等待僵尸进程返回其 termination status 结束状态 。要实现这点，你只需要在代码中调用 `wait` 系统函数即可。不过，这种方法并不是最理想的方案，因为如果你调用 `wait` 后，却没有结束了的子进程，那么 `wait` 调用将会阻塞服务器，相当于阻止了服务器处理新的客户端请求。那么还有其他的办法吗？答案是肯定的，其中一种办法就是将 `wait` 函数调用与 signal 信号处理函数 handler 结合使用。

这种方法的具体原理如下。当子进程退出时，系统内核会发送一个 `SIGCHLD` 信号。父进程可以设置一个信号处理函数，用于异步监测 `SIGCHLD` 事件，然后再调用 `wait` ，等待子进程结束并获取其结束状态，这样就可以避免产生僵尸进程。



顺便说明一下，异步事件意味着父进程实现并不知道该事件是否会发生。

接下来我们修改服务器代码，添加一个 `SIGCHLD` 事件处理函数，并在该函数中等待子进程结束。具体的代码见 `webserver3e.py` 文件：

```
 ##########################################################################
# Concurrent server - webserver3e.py                                     #
#                                                                        #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X       #
##########################################################################
import os
import signal
import socket
import time

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 5


def grim_reaper(signum, frame):
    pid, status = os.wait()
    print(
        'Child {pid} terminated with status {status}'
        '\n'.format(pid=pid, status=status)
    )


def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)
    # sleep to allow the parent to loop over to 'accept' and block there
    time.sleep(3)


def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
```

```
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    signal.signal(signal.SIGCHLD, grim_reaper)

    while True:
        client_connection, client_address = listen_socket.accept()
        pid = os.fork()
        if pid == 0:  # child
            listen_socket.close()  # close child copy
            handle_request(client_connection)
            client_connection.close()
            os._exit(0)
        else:  # parent
            client_connection.close()

if __name__ == '__main__':
    serve_forever()
```

启动服务器：

```
$ python webserver3e.py
```

再次使用 `curl` 命令，向修改后的并发服务器发送一个请求：

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

我们来看服务器的反应：

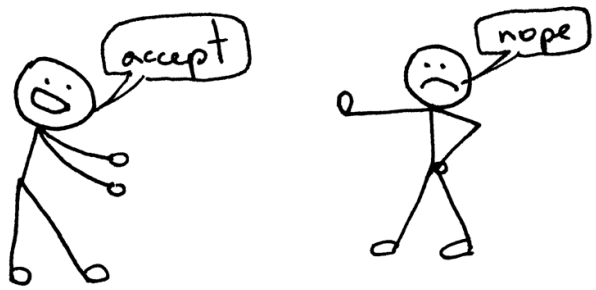
```
Serving HTTP on port 8888 ...
GET /hello HTTP/1.1
User-Agent: curl/7.35.0
Host: localhost:8888
Accept: */*


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
```

发生了什么事？ `accept` 函数调用报错了。

```
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
```

子进程退出时，父进程被阻塞在 `accept` 函数调用的地方，但是子进程的退出导致了 `SIGCHLD` 事件，这也激活了信号处

理函数。信号函数执行完毕之后，就导致了 `accept` 系统函数调用被中断：



别担心，这是个非常容易解决的问题。你只需要重新调用 `accept` 即可。下面我们再修改一下服务器代码（webserver3f.py），就可以解决这个问题：

```
 ##########################################################################
# Concurrent server - webserver3f.py                                     #
#                                                                        #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X       #
##########################################################################
import errno
import os
import signal
import socket

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 1024


def grim_reaper(signum, frame):
    pid, status = os.wait()


def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)


def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    signal.signal(signal.SIGCHLD, grim_reaper)

    while True:
        try:
            client_connection, client_address = listen_socket.accept()
        except IOError as e:
            code, msg = e.args
            # restart 'accept' if it was interrupted
            if code == errno.EINTR:
```

```
                continue
            else:
                raise

        pid = os.fork()
        if pid == 0:  # child
            listen_socket.close()  # close child copy
            handle_request(client_connection)
            client_connection.close()
            os._exit(0)
        else:  # parent
            client_connection.close()  # close parent copy and loop over


if __name__ == '__main__':
    serve_forever()
```

启动修改后的服务器：

```
$ python webserver3f.py
```

通过 `curl` 命令向服务器发送一个请求：

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

看到了吗？没有再报错了。现在，我们来确认下服务器没有再产生僵尸进程。只需要运行 `ps` 命令，你就会发现没有Python进程的状态是Z+了。太棒了！没有僵尸进程捣乱真是太好了。

- 如果你fork一个子进程，却不等待进程结束，该进程就会变成僵尸进程。
- 使用 `SIGCHLD` 时间处理函数来异步等待进程结束，获取其结束状态。
- 使用事件处理函数时，你需要牢记系统函数调用可能会被中断，要做好这类情况发生得准备。

好了，目前一切正常。没有其他问题了，对吗？呃，基本上是了。再次运行 `webserver3f.py` ，然后通过 `client3.py` 创建128个并行连接：

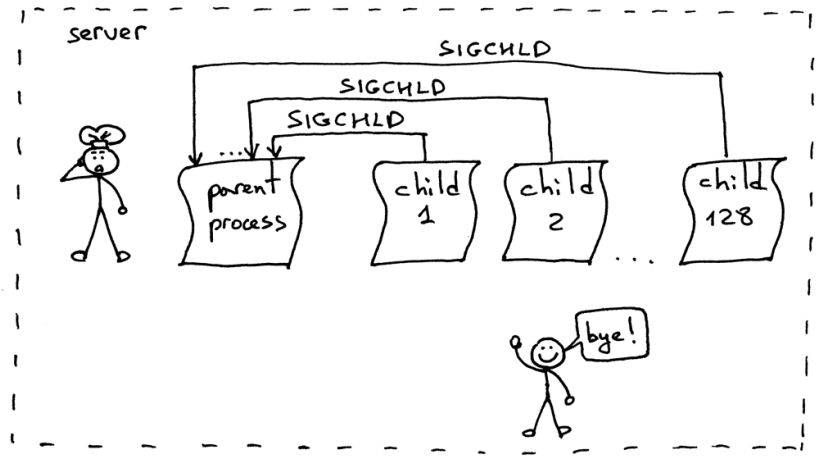
```
$ python client3.py --max-clients 128
```

现在再次运行 `ps` 命令：

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

噢，糟糕！僵尸进程又出现了！

这次又是哪里出了问题？当你运行128个并行客户端，建立128个连接时，服务器的子进程处理完请求，几乎是同一时间退出的，这就触发了一大波的 `SIGCHLD` 信号发送至父进程。但问题是这些信号并没有进入队列，所以有几个信号漏网，没有被服务器处理，这就导致出现了几个僵尸进程。



这个问题的解决方法，就是在 `SIGCHLD` 事件处理函数使用 `waitpid` ，而不是 `wait` ，再调用 `waitpid` 时增加 `WNOHANG` 选项，确保所有退出的子进程都会被处理。下面就是修改后的代码，webserver3g.py：

```
 ##########################################################################
# Concurrent server - webserver3g.py                                     #
#                                                                        #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X       #
##########################################################################
import errno
import os
import signal
import socket

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 1024


def grim_reaper(signum, frame):
    while True:
        try:
            pid, status = os.waitpid(
                -1,          # Wait for any child process
                 os.WNOHANG  # Do not block and return EWOULDBLOCK error
            )
        except OSError:
            return

        if pid == 0:  # no more zombies
            return


def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)
```

```
def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    signal.signal(signal.SIGCHLD, grim_reaper)

    while True:
        try:
            client_connection, client_address = listen_socket.accept()
        except IOError as e:
            code, msg = e.args
            # restart 'accept' if it was interrupted
            if code == errno.EINTR:
                continue
            else:
                raise

        pid = os.fork()
        if pid == 0:  # child
            listen_socket.close()  # close child copy
            handle_request(client_connection)
            client_connection.close()
            os._exit(0)
        else:  # parent
            client_connection.close()  # close parent copy and loop over

if __name__ == '__main__':
    serve_forever()
```

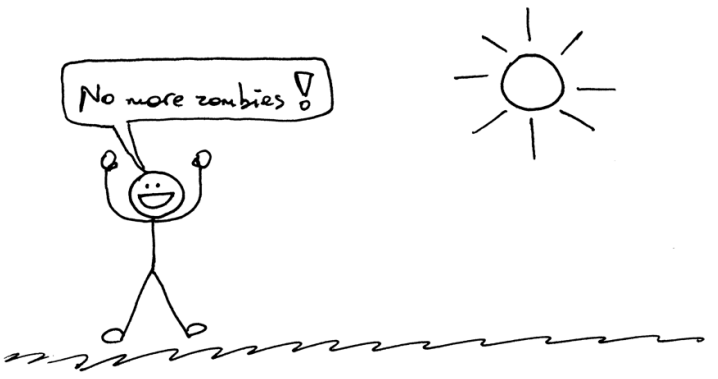
启动服务器：

```
$ python webserver3g.py
```

使用客户端 `client3.py` 进行测试：

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

现在请确认不会再出现僵尸进程了。

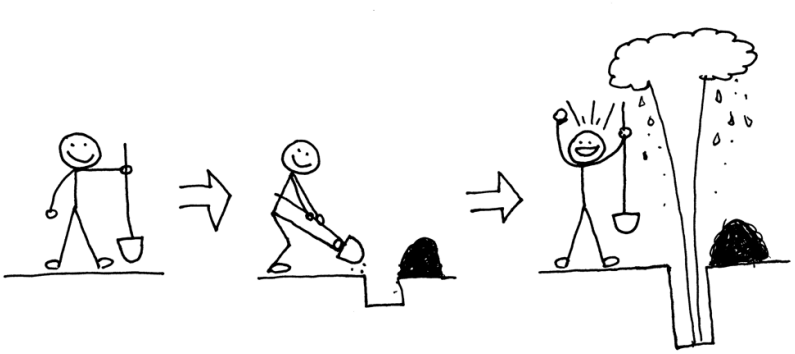


恭喜大家！现在已经自己开发了一个简易的并发服务器，这个代码可以作为你以后开发生产级别的网络服务器的基础。

最后给大家留一个练习题，把第二部分中的WSGI修改为并发服务器。最终的代码可以在这里查看。不过请你在自己实现了之后再查看。

接下来该怎么办？借用乔希·比林斯（19世纪著名幽默大师）的一句话：

> 要像一张邮票，坚持一件事情直到你到达目的地。



来源：编程派　参考原文：http://ruslanspivak.com/lsbaws-part3/　　作者： Ruslan
编译文章：http://codingpy.com/article/build-a-simple-web-server-part-three/

上一篇：黑客利用 Wi-Fi 攻击你的七种方法

发表评论

评论

最新评论　　我也要发表评论