用Python理解Web并发模型

<http://python.jobbole.com/85490/?utm_source=blog.jobbole.com&utm_medium=relatedPosts>

Java程序员进阶三条必经之路：数据库、虚拟机、异步通信。

**前言**

虽然异步是我们急需掌握的高阶技术，但是不积跬步无以至千里，同步技术的学习是不能省略的。今天这篇文章主要用Python来介绍Web并发模型，直观地展现同步技术的缺陷以及异步好在哪里。

**最简单的并发**

import socket

response = 'HTTP/1.1 200 OK\r\nConnection: Close\r\nContent-Length: 11\r\n\r\nHello World'

server = socket.socket()

server.bind(('0.0.0.0', 9527))

server.listen(1024)

while True:

client, clientaddr = server.accept() # blocking

request = client.recv(1024) # blocking

client.send(response) # maybe blocking

client.close()

上面这个例子太简单了，访问localhost:9527，返回“Hello World”。用ab来测试性能，数据如下：

ab -n 100000 -c 8 http://localhost:9527/

Time taken for tests: 1.568 seconds

发送10万个请求，8（我的CPU核数为8）个请求同时并发，耗时1.568秒。

性能瓶颈在哪里呢？就在上面的两个半阻塞。

accept和recv是完全阻塞的，而为什么send是半个阻塞呢？

在内核的 socket实现中，会有两个缓存 (buffer)。read buffer 和 write buffer 。当内核接收到网卡传来的客户端数据后，把数据复制到 read buffer ，这个时候 recv阻塞的进程就可以被唤醒。

当调用 send的时候，内核只是把 send的数据复制到 write buffer 里，然后立即返回。只有 write buffer 的空间不够时 send才会被阻塞，需要等待网卡发送数据腾空 write buffer 。在 write buffer的空间足够放下 send的数据时进程才可以被唤醒。

如果一个请求处理地很慢，其他请求只能排队，那么并发量肯定会受到影响。

**多进程**

每个请求对应一个进程倒是能解决上面的问题，但是进程太占资源，每个请求的资源都是独立的，无法共享，而且进程的上下文切换成本也很高。

import socket

import signal

import multiprocessing

response = 'HTTP/1.1 200 OK\r\nConnection: Close\r\nContent-Length: 11\r\n\r\nHello World'

server = socket.socket()

server.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server.bind(('0.0.0.0', 9527))

server.listen(1024)

def handler(client):

request = client.recv(1024)

client.send(response)

client.close()

#多进程里的子进程执行完后并不会死掉，而是变成僵尸进程，等待主进程挂掉后才会死掉，下面这条语句可以解决这个问题。

signal.signal(signal.SIGCHLD,signal.SIG\_IGN)

while True:

client, addr = server.accept()

process = multiprocessing.Process(target=handler, args=(client,))

process.start()

**Prefork**

这是多进程的改良版，预先分配好和CPU核数一样的进程数，可以控制资源占用，高效处理请求。

import socket

import multiprocessing

response = 'HTTP/1.1 200 OK\r\nConnection: Close\r\nContent-Length: 11\r\n\r\nHello World'

server = socket.socket()

server.bind(('0.0.0.0', 9527))

server.listen(1024)

def handler():

while True:

client, addr = server.accept()

request = client.recv(1024)

client.send(response)

client.close()

processors = 8

for i in range(0, processors):

process = multiprocessing.Process(target=handler, args=())

process.start()

**线程池**

import Queue

import socket

import threading

response = 'HTTP/1.1 200 OK\r\nConnection: Close\r\nContent-Length: 11\r\n\r\nHello World'

server = socket.socket()

server.bind(('0.0.0.0', 9527))

server.listen(1024)

def handler(queue):

while True:

client = queue.get()

request = client.recv(1024)

client.send(response)

client.close()

queue = Queue.Queue()

processors = 8

for i in range(0, processors):

thread = threading.Thread(target=handler, args=(queue,))

thread.daemon = True

thread.start()

while True:

client, clientaddr = server.accept()

queue.put(client)

耗时：3.901秒，大部分时间花在队列上，线程占用资源比进程少（资源可以共享），但是要考虑线程安全问题和锁的性能，而且python有臭名昭著的GIL，导致不能有效利用多核CPU。

**epoll**

最后祭出epoll大神，三大异步通信框架Netty、NodeJS、Tornado共同采用的通信技术，耗时1.582秒，但是要注意是单进程单线程哦。epoll真正发挥作用是在长连接应用里，单线程处理上万个长连接玩一样，占用资源极少。