

## 第32讲 | 不可忽视的多线程及并发问题

2018-08-07 蔡能

从0开始学游戏开发

[进入课程 >](#)



讲述：蔡能

时长 07:38 大小 3.50M



既然我们说到了服务器端的开发，我们就不得不提起多线程和并发的的问题，因为如果没有多线程和并发，是不可能做网络服务器端的，除非你的项目是 base 在 Nginx 或者 Apache 之上的。

### 多线程和并发究竟有什么区别和联系？

提到并发，不得不提到并行，所以我就讲这三个概念：并发、并行，以及多线程。作为初学者，你或许不太明白，多线程和并发究竟有什么区别和联系？下面我们就分别来看看。

**并发**出现在电脑只有一个 CPU 的情况下，那如果有多个线程需要操作，该怎么办呢？CPU 不可能一次只运行一个程序，运行完一个再运行第二个，这个效率任谁都忍受不了啊！所以，就想了个办法。

CPU 将运行的线程分成若干个 CPU 时间片来运行。不运行的那个线程就挂起，运行的时候，那个线程就活过来，切换地特别快，就好像是在同时运行一样。

你可以想象这个场景，有一个象棋大师，一个人对十个对手下棋，那十个人轮流和他下。大师从 1 号棋手这里开始下，下完 1 号走到 2 号的棋手面前，下 2 号棋手的棋，一直轮流走下去，直到再走向 1 号棋手这里再下一步。只要象棋大师下象棋下得足够快，然后他移动到下一位棋手这里又移动得足够快，大家都会觉得好像有十位象棋大师在和十个对手下棋。事实上只有一位象棋大师在下棋，只是他移动得很快而已。

**并行**和**并发**不同，并行是出现在多个物理 CPU 的情况下。在这种情况下，并行是真正的并发状态，是在物理状态下的并发运行。所以，并行是真的有几位象棋大师在应对几个对手。当然在并行的同时，CPU 也会进行并发运算。

而**多线程**是单个进程的切片，单个进程中的线程中的内存和资源都是共享的，所以线程之间进行沟通是很方便的。

多线程的意义，就好比一个厨师，他掌管了三个锅，一个锅在煮排骨，一个锅在烧鱼，另一个锅在煮面，这三个锅内容不同，火候不同，但是所有的调料和资源，包括菜、油、水、盐、味精、糖、酱油等等，都来自同一个地方（也就是资源共享），而厨师自己是一个进程，他分配了三个线程（也就是三个锅），这三个锅烧着不同的东西，三个食物或许不是同时出锅的，但是厨师心里有数，什么时候这个菜可以出锅，什么时候这个菜还需要煮。这就是多线程的一个比喻。

我们在编写网络服务器的时候，多线程和并发的是一定会考虑的。我们说的网络并发和 CPU 的并发可以说是异曲同工，也就是说，**网络并发的意义是，这个网络服务器可以同时支撑多少个用户同时登陆，或者同时在线操作。**

## 为什么 Python 用多个 CPU 的时候会出现问题？

那么我们又回头来看，为什么 Python、Ruby 或者 Node.js，在利用多个 CPU 的时候会出现问题呢？这是因为，它们是使用 C/C++ 语言编写的。是的，问题就在这里。

我们后续的内容还是会用 Python 来写，所以我们先来看看 Python 的多线程问题。Python 有个**GIL**（Global Interpreter Lock，全局解释锁），问题就出在 GIL 上。

使用 C 语言编写的 Python 版本（后面简称为 C-Python）的线程是操作系统的原生线程。在 Linux 上为 pthread，在 Windows 上为 Win thread，完全由操作系统调度线程的执行。

一个 Python 解释器进程内有一条主线程，以及多条用户程序的执行线程。即使在多核 CPU 平台上。由于 GIL 的存在，所以会禁止多线程的并行执行。这是为什么呢？

因为 Python 解释器进程内的多线程是合作多任务方式执行的。当一个线程遇到 I/O（输入输出）任务时，将释放 GIL 锁。计算密集型（以计算为主的逻辑代码）的线程在执行大约 100 次解释器的计步时，将释放 GIL 锁。你可以将计步看作是 Python 虚拟机的指令。计步实际上与 CPU 的时间片长度无关。我们可以通过 Python 的库 `sys.setcheckinterval()` 设置计步长度来控制 GIL 的释放事件。

在单核的 CPU 上，数百次间隔检查才会导致一次线程切换。在多核 CPU 上，就做不到这些了。从 Python 3.2 开始就使用新的 GIL 锁了。在新的 GIL 实现中，用一个固定的超时时间来指示当前的线程放弃全局锁。在当前线程保持这个锁，且其他线程请求这个锁的时候，当前线程就会在五毫秒后被强制释放掉这个锁。

我们如果要实现并行，利用 Python 的多线程效果不好，所以我们可以创建独立的进程来实现并行化。Python 2.6（含）以上版本引进了 multiprocessing 这个多进程包。


我们也可以把多线程的关键部分用 C/C++ 写成 Python 扩展，通过 ctypes 使 Python 程序直接调用 C 语言编译的动态库的导出函数来使用。

C-Python 的 GIL 的问题存在于 C-Python 的编写语言，原生语言 C 语言中，由于 GIL 为了保证 Python 解释器的顺利运行，所以事实上，多线程只是模拟了切换线程而已。这么做的话，如果你使用的是 IO 密集型任务的时候，就会提高速度。为什么这么说？

因为写文件读文件的时间完全可以将 GIL 锁给释放出来，而如果是计算密集型的任务，或许将会得到比单线程更慢的速度。为什么呢？事实上 GIL 是一个全局的排他锁，它并不能很好地利用 CPU 的多核，相反地，它会将多线程模拟成单线程进行上下文切换的形式进行运行。


我们来看一下，在计算密集型的代码中，单线程和多线程的比较。

## 单线程版本:

 复制代码

```
1 from threading import Thread
2 import time
3 def my_counter():
4     i = 0
5     for x in range(10000):
6         i = i + 1
7     return True
8 def run():
9     thread_array = {}
10    start_time = time.time()
11    for tt in range(2):
12        t = Thread(target=my_counter)
13        t.start()
14        t.join()
15    end_time = time.time()
16    print("count time: {}".format(end_time - start_time))
17 if __name__ == '__main__':
18     run()
```

## 多线程版本:

 复制代码

```
1 from threading import Thread
2 import time
3 def my_counter():
4     i = 0
5     for x in range(10000):
6         i = i + 1
7     return True
8 def run():
9     thread_array = {}
10    start_time = time.time()
11    for tt in range(2):
12        t = Thread(target=my_counter)
13        t.start()
14        thread_array[tid] = t
15    for i in range(2):
16        thread_array[i].join()
17    end_time = time.time()
18    print("count time: {}".format(end_time - start_time))
19 if __name__ == '__main__':
20     run()
```

当然，我们还可以把这个 ranger 的数字改得更大，看到更大的差异。

当计步完成后，将会达到一个释放锁的阈值，释放完后立刻又取得锁，然而这在单 CPU 环境下毫无问题，但是多 CPU 的时候，第二块 CPU 正要被唤醒线程的时候，第一块 CPU 的主线程又直接取得了主线程锁，这时候，就出现了第二块 CPU 不停地被唤醒，第一块 CPU 拿到了主线程锁继续执行内容，第二块继续等待锁，唤醒、等待，唤醒、等待。这样，事实上只有一块 CPU 在执行指令，浪费了其他 CPU 的时间。这就是问题所在。

这也就是 C 语言开发的 Python 语言的问题。当然如果是使用 Java 写成的 Python (Jython) 和 .NET 下的 Python (Iron Python)，并没有 GIL 的问题。事实上，它们其实连 GIL 锁都不存在。我们也可以使用新的 Python 实作项目 PyPy。所以，这些问题事实上是由于实现语言的差异造成的。

## 如何尽可能利用多线程和并发的优势？


我们来尝试另一种解决思路，我们仍然用的是 C-Python，但是我们要尽可能使之能利用多线程和并发的优势，这该怎么做呢？

multiprocess 是在 Python 2.6（含）以上版本的提供是为了弥补 GIL 的效率问题而出现的，不同的是它使用了多进程而不是多线程。每个进程有自己的独立的 GIL 锁，因此也不会出现进程之间，CPU 进行 GIL 锁的争抢问题，因为都是独立的进程。

当然 multiprocessing 也有不少问题。首先它会增加程序实现时线程间数据通信和同步的困难。

就拿计数器来举例子。如果我们要多个线程累加同一个变量，对于 thread 来说，申明一个 global 变量，用 thread.Lock 的 context 就可以了。而 multiprocessing 由于进程之间无法看到对方的数据，只能通过在主线程申明一个 Queue，put 再 get 或者用共享内存、共享文件、管道等等方法。

我们可以来看一下 multiprocess 的共享内容数据的方案。

 复制代码

```
1 from multiprocessing import Process, Queue
```

```
2 def f(q):
3     q.put([4031, 1024, 'my data'])
4 if __name__ == '__main__':
5     q = Queue()
6     p = Process(target=f, args=(q,))
7     p.start()
8     print q.get()
9     p.join()
```

这样的方案虽说可行，但是编码效率变得比较低下，但是也是一种权宜之计吧。

## 小结

我们来总结一下今天的内容。

我首先介绍了几个概念。并发是单个 CPU 之间切换多线程任务的操作。并行是多个 CPU 同时分配和运行多线程任务的操作。线程是进程内的独立任务单元，但是共享这个进程的所有资源。网络的并发指的是服务器同时可以承载多少数量的人数和任务。

而 C 语言编写的 Python 有 GIL 锁的问题，会让其多线程计算密集型的任务效率更低，解决方案有，利用多进程解决问题 或者 更换 Python 语言的实现版本，比如 PyPy 或者 JPython 等等。

给你留一个小问题，如果 Python 以多进程方式进行操作，那么如果我们网络服务器是用 Python 编写的，其中一个 Python 进程崩溃或者报错了，有什么办法可以让其复活？

欢迎留言说出你的看法。我在下一节的挑战中等你！

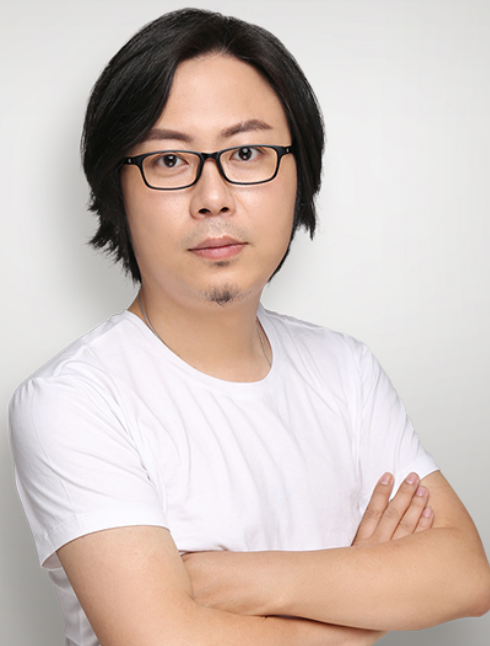


# 从0开始学游戏开发

你的游戏开发入门第一课

蔡能

原网易游戏引擎架构师  
资深游戏底层技术专家



新版升级：点击「👤 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 第31讲 | 热点剖析（八）：谈谈移动游戏的未来发展

下一篇 第33讲 | 如何判断心跳包是否离线？

## 精选留言 (1)

写留言



放羊大王

2018-08-08



监听信号量，重新 fork 。加锁什么是最烦的，go的sync包，现在都还没搞懂。能讲讲 ipc 通信方式与具体最小实现，伪代码也行，以及 actor 和 csp 。我看 elixir 也不错，就是用的人少。哎！感觉多线程其实就是运行同一份代码，通过if判断运行逻辑于是就有了 master worker。