预备知识

并发编程

所谓并发编程就是让程序中有多个部分能够并发或同时执行，并发编程带来的好处不言而喻，其中最为关键的两点是提升了执行效率和改善了用户体验。下面简单阐述一下Python中实现并发编程的三种方式：

1. 多线程：Python中通过`threading`模块的`Thread`类并辅以`Lock`、`Condition`、`Event`、`Semaphore`和`Barrier`等类来支持多线程编程。Python解释器通过GIL（全局解释器锁）来防止多个线程同时执行本地字节码，这个锁对于CPython（Python解释器的官方实现）是必须的，因为CPython的内存管理并不是线程安全的。因为GIL的存在，Python的多线程并不能利用CPU的多核特性。
2. 多进程：使用多进程可以有效的解决GIL的问题，Python中的`multiprocessing`模块提供了`Process`类来实现多进程，其他的辅助类跟`threading`模块中的类类似，由于进程间的内存是相互隔离的（操作系统对进程的保护），进程间通信（共享数据）必须使用管道、套接字等方式，这一点从编程的角度来讲是比较麻烦的，为此，Python的`multiprocessing`模块提供了一个名为`Queue`的类，它基于管道和锁机制提供了多个进程共享的队列。

```
"""
用下面的命令运行程序并查看执行时间，例如：
time python3 example06.py
real    0m20.657s
user    1m17.749s
sys     0m0.158s
使用多进程后实际执行时间为20.657秒，而用户时间1分17.749秒约为实际执行时间的4倍
这就证明我们的程序通过多进程使用了CPU的多核特性，而且这台计算机配置了4核的CPU
"""
import concurrent.futures
import math

PRIMES = [
    1116281,
    1297337,
    104395303,
    472882027,
    533000389,
    817504243,
    982451653,
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419
] * 5


def is_prime(num):
    """判断素数"""
    assert num > 0
    for i in range(2, int(math.sqrt(num)) + 1):
```

```
            if num % i == 0:
                return False
        return num != 1


def main():
    """主函数"""
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))


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

3. 异步编程（异步I/O）：所谓异步编程是通过调度程序从任务队列中挑选任务，调度程序以交叉的形式执行这些任务，我们并不能保证任务将以某种顺序去执行，因为执行顺序取决于队列中的一项任务是否愿意将CPU处理时间让位给另一项任务。异步编程通常通过多任务协作处理的方式来实现，由于执行时间和顺序的不确定，因此需要通过钩子函数（回调函数）或者`Future`对象来获取任务执行的结果。目前我们使用的Python 3通过`asyncio`模块以及`await`和`async`关键字（Python 3.5中引入，Python 3.7中正式成为关键字）提供了对异步I/O的支持。

```
import asyncio


async def fetch(host):
    """从指定的站点抓取信息(协程函数)"""
    print(f'Start fetching {host}\n')
    # 跟服务器建立连接
    reader, writer = await asyncio.open_connection(host, 80)
    # 构造请求行和请求头
    writer.write(b'GET / HTTP/1.1\r\n')
    writer.write(f'Host: {host}\r\n'.encode())
    writer.write(b'\r\n')
    # 清空缓存区(发送请求)
    await writer.drain()
    # 接收服务器的响应(读取响应行和响应头)
    line = await reader.readline()
    while line != b'\r\n':
        print(line.decode().rstrip())
        line = await reader.readline()
    print('\n')
    writer.close()


def main():
    """主函数"""
    urls = ('www.sohu.com', 'www.douban.com', 'www.163.com')
    # 获取系统默认的事件循环
    loop = asyncio.get_event_loop()
    # 用生成式语法构造一个包含多个协程对象的列表
    tasks = [fetch(url) for url in urls]
    # 通过asyncio模块的wait函数将协程列表包装成Task（Future子类）并等待其执行完成
    # 通过事件循环的run_until_complete方法运行任务直到Future完成并返回它的结果
    loop.run_until_complete(asyncio.wait(tasks))
    loop.close()
```

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

> 说明：目前大多数网站都要求基于HTTPS通信，因此上面例子中的网络请求不一定能收到正常的响应，也就是说响应状态码不一定是200，有可能是3xx或者4xx。当然我们这里的重点不在于获得网站响应的内容，而是帮助大家理解`asyncio`模块以及`async`和`await`两个关键字的使用。

我们对三种方式的使用场景做一个简单的总结。

以下情况需要使用多线程：

1. 程序需要维护许多共享的状态（尤其是可变状态），Python中的列表、字典、集合都是线程安全的，所以使用线程而不是进程维护共享状态的代价相对较小。
2. 程序会花费大量时间在I/O操作上，没有太多并行计算的需求且不需占用太多的内存。

以下情况需要使用多进程：

1. 程序执行计算密集型任务（如：字节码操作、数据处理、科学计算）。
2. 程序的输入可以并行的分成块，并且可以将运算结果合并。
3. 程序在内存使用方面没有任何限制且不强依赖于I/O操作（如：读写文件、套接字等）。

最后，如果程序不需要真正的并发性或并行性，而是更多的依赖于异步处理和回调时，异步I/O就是一种很好的选择。另一方面，当程序中有大量的等待与休眠时，也应该考虑使用异步I/O。

> 扩展：关于进程，还需要做一些补充说明。首先，为了控制进程的执行，操作系统内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程使之继续执行，这种行为被称为进程切换（也叫调度）。进程切换是比较耗费资源的操作，因为在进行切换时首先要保存当前进程的上下文（内核再次唤醒该进程时所需要的状态，包括：程序计数器、状态寄存器、数据栈等），然后还要恢复准备执行的进程的上下文。正在执行的进程由于期待的某些事件未发生，如请求系统资源失败、等待某个操作完成、新数据尚未到达等原因会主动由运行状态变为阻塞状态，当进程进入阻塞状态，是不占用CPU资源的。这些知识对于理解到底选择哪种方式进行并发编程也是很重要的。

I/O模式和事件驱动

对于一次I/O操作（以读操作为例），数据会先被拷贝到操作系统内核的缓冲区中，然后从操作系统内核的缓冲区拷贝到应用程序的缓冲区（这种方式称为标准I/O或缓存I/O，大多数文件系统的默认I/O都是这种方式），最后交给进程。所以说，当一个读操作发生时（写操作与之类似），它会经历两个阶段：(1)等待数据准备就绪；(2)将数据从内核拷贝到进程中。

由于存在这两个阶段，因此产生了以下几种I/O模式：

1. 阻塞 I/O（blocking I/O）：进程发起读操作，如果内核数据尚未就绪，进程会阻塞等待数据直到内核数据就绪并拷贝到进程的内存中。
2. 非阻塞 I/O（non-blocking I/O）：进程发起读操作，如果内核数据尚未就绪，进程不阻塞而是收到内核返回的错误信息，进程收到错误信息可以再次发起读操作，一旦内核数据准备就绪，就立即将数据拷贝到了用户内存中，然后返回。
3. 多路I/O复用（I/O multiplexing）：监听多个I/O对象，当I/O对象有变化（数据就绪）的时候就通知用户进程。多路I/O复用的优势并不在于单个I/O操作能处理得更快，而是在于能处理更多的I/O操作。
4. 异步 I/O（asynchronous I/O）：进程发起读操作后就可以去做别的事情了，内核收到异步读操作后会立即返回，所以用户进程不阻塞，当内核数据准备就绪时，内核发送一个信号给用户进程，告诉它读操作完成了。

通常，我们编写一个处理用户请求的服务器程序时，有以下三种方式可供选择：

1. 每收到一个请求，创建一个新的进程，来处理该请求；
2. 每收到一个请求，创建一个新的线程，来处理该请求；
3. 每收到一个请求，放入一个事件列表，让主进程通过非阻塞I/O方式来处理请求

第1种方式实现比较简单，但由于创建进程开销比较大，会导致服务器性能比较差；第2种方式，由于要涉及到线程的同步，有可能会面临竞争、死锁等问题；第3种方式，就是所谓事件驱动的方式，它利用了多路I/O复用和异步I/O的优点，虽然代码逻辑比前面两种都复杂，但能达到最好的性能，这也是目前大多数网络服务器采用的方式。