异步IO

在IO编程一节中，我们已经知道，CPU的速度远远快于磁盘、网络等IO。在一个线程中，CPU执行代码的速度极快，然而，一旦遇到IO操作，如读写文件、发送网络数据时，就需要等待IO操作完成，才能继续进行下一步操作。这种情况称为同步IO。

在IO操作的过程中，当前线程被挂起，而其他需要CPU执行的代码就无法被当前线程执行了。

因为一个IO操作就阻塞了当前线程，导致其他代码无法执行，所以我们必须使用多线程或者多进程来并发执行代码，为多个用户服务。每个用户都会分配一个线程，如果遇到IO导致线程被挂起，其他用户的线程不受影响。

多线程和多进程的模型虽然解决了并发问题，但是系统不能无上限地增加线程。由于系统切换线程的开销也很大，所以，一旦线程数量过多，CPU的时间就花在线程切换上了，真正运行代码的时间就少了，结果导致性能严重下降。

由于我们要解决的问题是CPU高速执行能力和IO设备的龟速严重不匹配，多线程和多进程只是解决这一问题的一种方法。

另一种解决IO问题的方法是异步IO。当代码需要执行一个耗时的IO操作时，它只发出IO指令，并不等待IO结果，然后就去执行其他代码了。一段时间后，当IO返回结果时，再通知CPU进行处理。

可以想象如果按普通顺序写出的代码实际上是没法完成异步IO的：

```
do_some_code()
f = open('/path/to/file', 'r')
r = f.read() # <== 线程停在此处等待IO操作结果
# IO操作完成后线程才能继续执行:
do_some_code(r)
```

所以，同步IO模型的代码是无法实现异步IO模型的。

异步IO模型需要一个消息循环，在消息循环中，主线程不断地重复“读取消息-处理消息”这一过程：

```
loop = get_event_loop()
while True:
    event = loop.get_event()
    process_event(event)
```

消息模型其实早在应用在桌面应用程序中了。一个GUI程序的主线程就负责不停地读取消息并处理消息。所有的键盘、鼠标等消息都被发送到GUI程序的消息队列中，然后由GUI程序的主线程处理。

由于GUI线程处理键盘、鼠标等消息的速度非常快，所以用户感觉不到延迟。某些时候，GUI线程在一个消息处理的过程中遇到问题导致一次消息处理时间过长，此时，用户会感觉到整个GUI程序停止响应了，敲键盘、点鼠标都没有反应。这种情况说明在消息模型中，处理一个消息必须非常迅速，否则，主线程将无法及时处理消息队列中的其他消息，导致程序看上去停止响应。

消息模型是如何解决同步IO必须等待IO操作这一问题的呢？当遇到IO操作时，代码只负责发出IO请求，不等待IO结果，然后直接结束本轮消息处理，进入下一轮消息处理过程。当IO操作完成后，将收到一条“IO完成”的消息，处理该消息时就可以直接获取IO操作结果。

在“发出IO请求”到收到“IO完成”的这段时间里，同步IO模型下，主线程只能挂起，但异步IO模型下，主线程并没有休息，而是在消息循环中继续处理其他消息。这样，在异步IO模型下，一个线程就可以同时处理多个IO请求，并且没有切换线程的操作。对于大多数IO密集型的应用程序，使用异步IO将大大提升系统的多任务处理能力。

协程

在学习异步IO模型前，我们先来了解协程。

协程，又称微线程，纤程。英文名Coroutine。

协程的概念很早就提出来了，但直到最近几年才在某些语言（如Lua）中得到广泛应用。

子程序，或者称为函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。

所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。

子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似CPU的中断。比如子程序A、B：

```
def A():
    print('1')
    print('2')
    print('3')

def B():
    print('x')
    print('y')
    print('z')
```

假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：

```
1
2
x
y
3
z
```

但是在A中是没有调用B的，所以协程的调用比函数调用理解起来要难一些。

看起来A、B的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python对协程的支持是通过generator实现的。

在generator中，我们不但可以通过`for`循环来迭代，还可以不断调用`next()`函数获取由`yield`语句返回的下一个值。

但是Python的`yield`不但可以返回一个值，它还可以接收调用者发出的参数。

来看例子：

传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。

如果改用协程，生产者生产消息后，直接通过`yield`跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

```
def consumer():
    r = ''
    while True:
        n = yield r
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        r = '200 OK'

def produce(c):
    c.send(None)
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()

c = consumer()
produce(c)
```

执行结果：

```
[PRODUCER] Producing 1...
[CONSUMER] Consuming 1...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 2...
[CONSUMER] Consuming 2...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 3...
[CONSUMER] Consuming 3...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 4...
[CONSUMER] Consuming 4...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 5...
[CONSUMER] Consuming 5...
[PRODUCER] Consumer return: 200 OK
```

注意到`consumer`函数是一个`generator`，把一个`consumer`传入`produce`后：

1. 首先调用`c.send(None)`启动生成器；
2. 然后，一旦生产了东西，通过`c.send(n)`切换到`consumer`执行；
3. `consumer`通过`yield`拿到消息，处理，又通过`yield`把结果传回；
4. `produce`拿到`consumer`处理的结果，继续生产下一条消息；
5. `produce`决定不生产了，通过`c.close()`关闭`consumer`，整个过程结束。

整个流程无锁，由一个线程执行，`produce`和`consumer`协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

最后套用Donald Knuth的一句话总结协程的特点：

“子程序就是协程的一种特例。”

asyncio

`asyncio`是Python 3.4版本引入的标准库，直接内置了对异步IO的支持。

`asyncio`的编程模型就是一个消息循环。我们从`asyncio`模块中直接获取一个`EventLoop`的引用，然后把需要执行的协程扔到`EventLoop`中执行，就实现了异步IO。

用`asyncio`实现`Hello world`代码如下：

```
import asyncio

@asyncio.coroutine
def hello():
    print("Hello world!")
    # 异步调用asyncio.sleep(1):
    r = yield from asyncio.sleep(1)
    print("Hello again!")

# 获取EventLoop:
loop = asyncio.get_event_loop()
# 执行coroutine
loop.run_until_complete(hello())
loop.close()
```

`@asyncio.coroutine`把一个generator标记为coroutine类型，然后，我们就把这个`coroutine`扔到`EventLoop`中执行。

`hello()`会首先打印出`Hello world!`，然后，`yield from`语法可以让我们方便地调用另一个`generator`。由于`asyncio.sleep()`也是一个`coroutine`，所以线程不会等待`asyncio.sleep()`，而是直接中断并执行下一个消息循环。当`asyncio.sleep()`返回时，线程就可以从`yield from`拿到返回值（此处是`None`），然后接着执行下一行语句。

把`asyncio.sleep(1)`看成是一个耗时1秒的IO操作，在此期间，主线程并未等待，而是去执行`EventLoop`中其他可以执行的`coroutine`了，因此可以实现并发执行。

我们用Task封装两个`coroutine`试试：

```
import threading
import asyncio

@asyncio.coroutine
def hello():
    print('Hello world! (%s)' % threading.currentThread())
    yield from asyncio.sleep(1)
    print('Hello again! (%s)' % threading.currentThread())

loop = asyncio.get_event_loop()
tasks = [hello(), hello()]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

观察执行过程：

```
Hello world! (<_MainThread(MainThread, started 140735195337472)>)
Hello world! (<_MainThread(MainThread, started 140735195337472)>)
(暂停约1秒)
Hello again! (<_MainThread(MainThread, started 140735195337472)>)
Hello again! (<_MainThread(MainThread, started 140735195337472)>)
```

由打印的当前线程名称可以看出，两个`coroutine`是由同一个线程并发执行的。

如果把`asyncio.sleep()`换成真正的IO操作，则多个`coroutine`就可以由一个线程并发执行。

我们用`asyncio`的异步网络连接来获取sina、sohu和163的网站首页：

```
import asyncio

@asyncio.coroutine
def wget(host):
    print('wget %s...' % host)
    connect = asyncio.open_connection(host, 80)
    reader, writer = yield from connect
    header = 'GET / HTTP/1.0\r\nHost: %s\r\n\r\n' % host
    writer.write(header.encode('utf-8'))
    yield from writer.drain()
    while True:
        line = yield from reader.readline()
        if line == b'\r\n':
            break
        print('%s header > %s' % (host, line.decode('utf-8').rstrip()))
    # Ignore the body, close the socket
    writer.close()

loop = asyncio.get_event_loop()
tasks = [wget(host) for host in ['www.sina.com.cn', 'www.sohu.com',
'www.163.com']]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

执行结果如下：

```
wget www.sohu.com...
wget www.sina.com.cn...
```

```
wget www.163.com...
(等待一段时间)
(打印出sohu的header)
www.sohu.com header > HTTP/1.1 200 OK
www.sohu.com header > Content-Type: text/html
...
(打印出sina的header)
www.sina.com.cn header > HTTP/1.1 200 OK
www.sina.com.cn header > Date: Wed, 20 May 2015 04:56:33 GMT
...
(打印出163的header)
www.163.com header > HTTP/1.0 302 Moved Temporarily
www.163.com header > Server: Cdn Cache Server V2.0
...
```

可见3个连接由一个线程通过`coroutine`并发完成。

小结

- `asyncio`提供了完善的异步IO支持；
- 异步操作需要在`coroutine`中通过`yield from`完成；
- 多个`coroutine`可以封装成一组Task然后并发执行。

async/await

用`asyncio`提供的`@asyncio.coroutine`可以把一个generator标记为coroutine类型，然后在coroutine内部用`yield from`调用另一个coroutine实现异步操作。

为了简化并更好地标识异步IO，从Python 3.5开始引入了新的语法`async`和`await`，可以让coroutine的代码更简洁易读。

请注意，`async`和`await`是针对coroutine的新语法，要使用新的语法，只需要做两步简单的替换：

1. 把`@asyncio.coroutine`替换为`async`；
2. 把`yield from`替换为`await`。

让我们对比一下上一节的代码：

```
@asyncio.coroutine
def hello():
    print("Hello world!")
    r = yield from asyncio.sleep(1)
    print("Hello again!")
```

用新语法重新编写如下：

```
async def hello():
    print("Hello world!")
    r = await asyncio.sleep(1)
    print("Hello again!")
```

剩下的代码保持不变。

小结

- Python从3.5版本开始为`asyncio`提供了`async`和`await`的新语法；
- 注意新语法只能用在Python 3.5以及后续版本，如果使用3.4版本，则仍需使用上一节的方案。

练习

将上一节的异步获取sina、sohu和163的网站首页源码用新语法重写并运行。

aiohttp

`asyncio`可以实现单线程并发IO操作。如果仅用在客户端，发挥的威力不大。如果把`asyncio`用在服务器端，例如Web服务器，由于HTTP连接就是IO操作，因此可以用单线程+`coroutine`实现多用户的高并发支持。

`asyncio`实现了TCP、UDP、SSL等协议，`aiohttp`则是基于`asyncio`实现的HTTP框架。

我们先安装`aiohttp`：

```
pip install aiohttp
```

然后编写一个HTTP服务器，分别处理以下URL：

- `/` - 首页返回`b'Index'`；
- `/hello/{name}` - 根据URL参数返回文本`hello, %s!`。

代码如下：

```
import asyncio

from aiohttp import web

async def index(request):
    await asyncio.sleep(0.5)
    return web.Response(body=b'<h1>Index</h1>')

async def hello(request):
    await asyncio.sleep(0.5)
    text = '<h1>hello, %s!</h1>' % request.match_info['name']
    return web.Response(body=text.encode('utf-8'))

async def init(loop):
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', index)
    app.router.add_route('GET', '/hello/{name}', hello)
    srv = await loop.create_server(app.make_handler(), '127.0.0.1', 8000)
    print('Server started at http://127.0.0.1:8000...')
    return srv

loop = asyncio.get_event_loop()
loop.run_until_complete(init(loop))
loop.run_forever()
```

注意`aiohttp`的初始化函数`init()`也是一个`coroutine`，`loop.create_server()`则利用`asyncio`创建TCP服务。