# Tornado 异步非阻塞的实现


什么是异步

- 同步 IO
- 异步 IO
- 协程
- 关于异步概念的误区

Python3 的异步实现

- asyncio
- async/await

Tornado 的异步实现

- 异步 handler
  - tornado.web.asynchronous
  - tornado.gen.coroutine
- 异步网络请求
  - Requests + ThreadPoolExecutor
- 多异步请求并发


参考资料：

---


1. 👍异步IO – 廖雪峰的官方网站
2. 👍协程：异步与并发
3. 使用tornado让你的请求异步非阻塞 – 董伟明博客
4. 使用Python进行并发编程–PoolExecutor篇 – 董伟明博客
5. 协程 – 维基百科
6. Tornado.gen – Generator-based coroutines
7. 👍Asynchronous and non-Blocking I/O
8. 真正的 Tornado 异步非阻塞


---

## 什么是异步

## 同步 IO

我们知道，CPU 的速度远远快于磁盘、网络等 IO。在一个线程中，当遇到 IO 操作时，如读写文件、发送网络请求，就需要等待 IO 操作完成，才能继续进行下一步的操作。这种称之为 **同步 IO**。

同步 IO 会在进行 IO 操作时，挂起当前线程，导致其他需要 CPU 执行的代码无法被当前线程处理，CPU 的高速执行能力和IO设备的龟速（与 CPU 相比）严重不匹配。为此，我们可以使用多进程和多线程解决，也可以使用 **异步 IO**。

## 异步 IO

异步 IO 的工作模式是，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。一段时间后，当 IO 返回结果时，再通知 CPU 进行处理。

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

## 协程

协程，又称微线程，纤程，英文 coroutine。

协程的概念可以和函数调用来类比理解：

- 子函数的入口只有一个，一旦退出即完成了函数的执行，子函数的一个实例只会返回一次。子函数与父函数之间是被调用者与调用者的关系；
- 协程可以通过 yield 调用其它协程。通过 yield 方式转移执行权的携程之间不是调用者与被调用者的关系，而是彼此对称、平等的。
- 函数调用的声明周期遵循后进先出（最后一个被调用的子例程最先返回）；相反，协程的生命周期完全由他们的使用需要决定。

以下是一个协程实现的生产者-消费者模型，生产者生产消息后，通过 yield 跳转到消费者开始执行，待消费者执行完毕后，切回生产者继续生产。

```python
def consumer():
    """消费者"""
    r = ""
    while True:
        n = yield r
        if not n:
            return
        print("[CONSUMER] Consuming %s ..." % n)
        r = "200 OK"
def producer(c):
    """生产者"""
    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)
```

输出为：

```
[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
```

## 关于异步概念的误区

1. **异步不能提升单个IO操作任务（web 请求）的响应速度**。异步通过在等待 IO 操作时执行其他代码，提高了 CPU 的利用率，进而提升了整体的运行效率，但是对单个任务（web 请求）来说，还是需要等待 IO 返回结果后才能继续操作（返回 HTTP 响应），不能提升单个任务的响应速度。
2. **同步程序不能通过协程异步封装的形式变为异步程序**。要写出有异步效果的程序，只有协程是不够的，还需要有底层 IO 的支持。在发生 IO 时，要将 IO 操作交给异步实现去执行，并让渡出协程的执行权，由调度去调度执行其他协程。因此如果底层 IO 未对协程调用做处理，其结果仍然会阻塞这个协程，不能实现异步的效果，比如 SQLAlchemy。

# Python3 的异步实现

## 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。

Python | 复制代码

```python
import threading
import asyncio
# @asyncio.coroutine把一个generator标记为coroutine类型
@asyncio.coroutine
def hello():
    print('Hello world! (%s)' % threading.currentThread())
    # 把asyncio.sleep(1)看成是一个耗时1秒的IO操作
    r = yield from asyncio.sleep(1)
    print('Hello again! (%s)' % threading.currentThread())
# 获取event_loop引用
loop = asyncio.get_event_loop()
# 把coroutine扔到EventLoop中执行
tasks = [hello(), hello()]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

1. tasks 中的第一个 `hello()` 执行时，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，即 `asyncio.sleep()` 。
2. 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，即 tasks 中的第二个 `hello()` 。
3. 第二个 `hello()` 的执行同第一个 `hello()` 一样。
4. 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，打印出 "Hello, again..."

执行结果如下，两个 `coroutine` 是由同一个线程“并发”执行的。

```
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)>)
```

## async/await

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。

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`。

举个🌰

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

用新语法可以编写为：

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

其他代码保持不变。

# Tornado 的异步实现

Tornado 是一个 Python Web 开发框架，也是异步网络请求库。通过使用非阻塞网络 IO，号称能够承载 10K 的请求量。

> Tornado is a Python web framework and asynchronous networking library, originally developed at FriendFeed. By using non-blocking network I/O, Tornado can scale to tens of thousands of open connections, making it ideal for long polling, WebSockets, and other applications that require a long-lived connection to each user.

常见的情况是，在 request handler 可能需要进行网络请求。此时我们应尽量使用异步 handler 和异步网络请求，以提高并发量和 Web 服务器效率。

## 异步 handler

异步 handler 实现主要有两种方式：`tornado.web.asynchronous` 和 `tornado.gen.coroutine` 。
**推荐使用 `tornado.gen.coroutine` 方式，不需要写回调函数，改造已有的同步代码也相对简单。**

### tornado.web.asynchronous

使用 `tornado.web.asynchronous` 装饰器修饰 handler，直接调用异步代码并使用回调函数处理响应。

Python | 复制代码

```python
import tornado.web
import tornado.httpclient
class MyRequestHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        """异步 handler 方法"""
        http_client = tornado.httpclient.AsyncHTTPClient()
        http_client.fetch("https://google.com/", self._on_download)
    def _on_download(self, response):
        """回调方法"""
        self.write("Downloaded!")
        # 使用了 asynchronous 需要手动 finish，否则一直 pending
        self.finish()
```

### tornado.gen.coroutine

使用 `tornado.gen.asynchr` 装饰器修饰 handler，使用 yield 调用异步代码并获取响应。

```python
import tornado.gen
import tornado.web
import tornado.httpclient
class MyRequestHandler(tornado.web.RequestHandler):
    @tornado.gen.coroutine
    def get(self):
        """异步 handler 方法"""
        http_client = tornado.httpclient.AsyncHTTPClient()
        response = yield http_client.fetch("https://google.com/")
        retunr self.write(response.body)
```

## 异步网络请求

在 Tornado 中进行异步网络请求，可以使用 Tornado 自带的 `tornado.httpclient.AsyncHTTPClient` 。
但在实际开发中，我们一般使用 `Requests` 库进行网络请求，相比前者，简直不能更好用。因此这里我们介绍一种将同步代码封装为“异步代码”的方法。

### Requests + ThreadPoolExecutor

Python3 中的 `concurrent.futures.ThreadPoolExecutor` 是对多线程的更高级封装，其内部实现中，返回的是 `concurrent.futures.Future` 对象。

Future 是常见的一种并发设计模式，在多个其他语言中都可以见到这种解决方案。一个 Future 对象代表了一些尚未就绪（完成）的结果，在「将来」的某个时间就绪了之后就可以获取到这个结果。**在 Future 模式下，调用方式改为异步**。

我们使用 `ThreadPoolExecutor` 对requests库进行封装，启动一个线程来执行阻塞的网络请求，假装自己是一个异步 IO。 `ThreadPoolExecutor` 的接口比 `threading` 模块要简单，有利于写出高效、异步、非阻塞的并行代码。

首先封装 requests 库

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from concurrent.futures import ThreadPoolExecutor
import tornado.gen
import requests
class AsyncRequests(object):
    """异步网络请求类
    使用线程池将同步网络请求转换为异步网络请求。
    使用requests进行网络请求，实现了get/post/put请求方法。
    """
    thread_pool = ThreadPoolExecutor(4)
    @classmethod
    @tornado.gen.coroutine
    def aget(cls, *args, **kw):
        """http get"""
        resp = yield cls.thread_pool.submit(requests.get, *args, **kw)
        return resp
```

在异步 handler 中调用异步网络请求 `AsyncRequests.aget`

```python
import tornado.web
import tornado.gen
class StatisticPlatformData(tornado.web.RequestHandler):
    """采购用户数据累计"""
    @tornado.web.authenticated
    @tornado.gen.coroutine
    def get(self):
        url = "https://httpbin.org/get"
        resp = yield AsyncAPIRequests.aget(url)
        jsondata = resp.json()
        return self.write(jsondata)
```

## 多异步请求并发

有时需要在一个 handler 里面同时进行多个网络请求，如果我们像下面这样写，那么网络请求是串行执行的，HTTP 响应时间会比较长。

```python
import tornado.gen
@tornado.gen.coroutine
def get(self):
    url = "https://httpbin.org/get"
    resp1 = yield AsyncAPIRequests.aget(url)
    resp2 = yield AsyncAPIRequests.aget(url)
    resp3 = yield AsyncAPIRequests.aget(url)
    return self.write("haha")
```

多异步请求并发执行：

```python
import tornado.gen
@tornado.gen.coroutine
def get(self):
    http_client = AsyncHTTPClient()
    # 使用 list 方式
    response1, response2 = yield [http_client.fetch(url1),
                                  http_client.fetch(url2)]
    # 使用 dict 方式
    response_dict = yield dict(response3=http_client.fetch(url3),
                               response4=http_client.fetch(url4))
    response3 = response_dict['response3']
    response4 = response_dict['response4']
```