Tornado Web Framework

Part 1:

同步 I/O && 异步 I/O、 阻塞 I/O && 非阻塞 I/P 区别：

首先一个 IO 操作其实分成了两个步骤：发起 IO 请求和实际的 IO 操作

同步 IO 指的是用户进程触发 I/O 操作请求之后并等待或者轮询的去查看 I/O 执行操作是否就绪。

异步 IO 是指用户进程触发 I/O 请求操作以后就立即返回，继续开始做自己的事情，而当 I/O 执行操作已经完成的时候， 用户进程会得到 I/O 完成的通知

--> 因此阻塞 IO、非阻塞 IO、IO 服用、信号驱动 IO 都是同步 IO

阻塞 IO 和非阻塞 IO 的区别在于第一步，发起 IO 请求是否会被阻塞(blocked)，如果阻塞直到完成那么就是传统的阻塞 IO，如果不阻塞，那么就是非阻塞 IO。但是无论是阻塞 I/O 还是 非阻塞 I/O 在请求操作之后都是通过轮询/等待的方式查看 I/O 执行操作是否就绪，因此都是同步 I/O。 --> 阻塞和非阻塞体现在是否能进去请求等待队列， 而同步和异步 I/O 为是否需要进去请求等待队列

Part2:

轮询、长轮询、长连接、Web Socket 区别：

Introduction:

Web 端即时通讯主要有四种方式，它们分别是轮询、长轮询(comet)、长连接(SSE)、Web Socket。它们大体可以分为两类，一种是在 HTTP 基础上实现的，包括短轮询、comet 和 SSE；另一种不是在 HTTP 基础上实现是，即 Web Socket。下面分别介绍一下这四种轮询方式，以及它们各自的优缺点。

轮询：

轮询的基本思路就是浏览器每隔一段时间向浏览器发送 http 请求，服务器端在收到请求后，不论是否有数据更新，都直接进行响应。这种方式实现的即时通信，本质上还是浏览器发送请求，服务器接受请求的一个过程，通过让客户端不断的进行请求，使得客户端能够模拟实时地收到服务器端的数据的变化。

优点： 比较简单，易于理解，实现起来也没有什么技术难点。

缺点： 由于需要不断的建立 http 连接，严重浪费了服务器端和客户端的资源。尤其是在客户端，距离来说，如果有数量级相对比较大的人同时位于基于短轮询的应用中，那么每一个用户的客户端都会疯狂的向服务器端发送 http 请求，而且不会间断。人数越多，服务器端压力越大，这是很不合理的。

使用： 因此轮询不适用于那些同时在线用户数量比较大，并且很注重性能的 Web 应用。

长轮询：

当服务器收到客户端发来的请求后,服务器端不会直接进行响应，而是先将这个请求挂起，然后判断服务器端数据是否有更新。如果有更新，则进行响应，如果一直没有数据，则

到达一定的时间限制(服务器端设置)才返回。 客户端 JavaScript 响应处理函数会在处理完服务器返回的信息后，再次发出请求，重新建立连接。

优点： 明显减少了很多不必要的 http 请求次数，相比之下节约了资源。

缺点： 连接挂起也会导致资源的浪费。

@ 轮询与长轮询都是基于 HTTP 的，两者本身存在着缺陷:轮询需要更快的处理速度；长轮询则更要求处理并发的能力;两者都是“被动型服务器”的体现:服务器不会主动推送信息，而是在客户端发送 ajax 请求后进行返回的响应。而理想的模型是"在服务器端数据有了变化后，可以主动推送给客户端",这种"主动型"服务器是解决这类问题的很好的方案。Web Sockets 就是这样的方案。

Web Sockets:

Web Socket 是 Html5 定义的一个新协议，与传统的 http 协议不同，该协议可以实现服务器与客户端之间全双工通信。简单来说，首先需要在客户端和服务器端建立起一个连接，这部分需要 http。连接一旦建立，客户端和服务器端就处于平等的地位，可以相互发送数据，不存在请求和响应的区别。

优点: 实现了双向通信

缺点: 是服务器端的逻辑非常复杂。现在针对不同的后台语言有不同的插件可以使用。

长连接（Server-Sent Events）：

SSE 最大的特点就是不需要客户端发送请求，可以实现只要服务器端数据有更新，就可以马上发送到客户端。

优点： 不需要建立或保持大量的客户端发往服务器端的请求，节约了很多资源，提升应用性能。SSE 的实现非常简单，并且不需要依赖其他插件。

Part 3:

WSGI（Web Server Gateway Interface）:

Introduction:

一个 Web 应用的本质：1.浏览器发送一个 HTTP 请求；2.服务器收到请求，生成一个 HTML 文档；3.服务器把 HTML 文档作为 HTTP 响应的 Body 发送给浏览器；4.浏览器收到 HTTP 响应，从 HTTP Body 取出 HTML 文档并显示。

所以，最简单的 Web 应用就是先把 HTML 用文件保存好，用一个现成的 HTTP 服务器软件，接收用户请求，从文件中读取 HTML，返回。Apache、Nginx、Lighttpd 等这些常见的静态服务器就是干这件事情的。

如果要动态生成 HTML，就需要把上述步骤自己来实现。不过，接受 HTTP 请求、解析 HTTP 请求、发送 HTTP 响应都是苦力活，如果我们自己来写这些底层代码，还没开始写动态 HTML 呢，就得花个把月去读 HTTP 规范。正确的做法是底层代码由专门的服务器软件实现，我们用 Python 专注于生成 HTML 文档。因为我们不希望接触到 TCP 连接、HTTP 原始请求和响应格式，所以，需要一个统一的接口，让我们专心用 Python 编写 Web 业务。这个接口就是 WSGI：Web Server Gateway Interface。

@无论多么复杂的 Web 应用程序，入口都是一个 WSGI 处理函数。HTTP 请求的所有输入信息都可以通过 environ 获得，HTTP 响应的输出都可以通过 start_response(header)加上函数返回值作为 Body。复杂的 Web 应用程序，光靠一个 WSGI 函数来处理还是太底层了，我

们需要在 WSGI 之上再抽象出 Web 框架，进一步简化 Web 开发。

Part4:

Implement a web server which has:
Simple mode:
latency HTTP 链接首字节传输的延迟。以毫秒表示。
speed 所有通过 chaos-proxy 的 HTTP 链接总速率限制。以字节数表示。

Advanced mode:
1. 随机的延迟
增加参数：latency-(min|max) HTTP
链接首字节传输的延迟。以毫秒表示。
2. 速率限制
增加参数：speed-(min|max)
所有通过 chaos-proxy 的 HTTP 链接总速率限制。以字节数表示。
3. 抖动
jitter-prob 抖动发生的概率。 0-1 之间
jitter-(min|max) 抖动（短时间断网）的时间长度。以毫秒表示。
4. 断网
reset-enable 启用主动断开 HTTP 链接（模拟断网）。
reset-prob 断网发生的概率。

Part 5: 协程

What is coroutine:

协程遇到 io 操作而阻塞时，立即切换到别的任务，如果操作完成则进行回调返回执行结果，提高了效率，同时这样也可以充分利用 CPU 和其他资源，这就是异步协程的优势，并且协程本质上是个单进程，相对于多进程来说，无需进程间上下文切换的开销，无需原子操作锁定及同步的开销，编程模型也非常简单。

async、await:

异步生成器迭代的流程：当生产者完成和返回之后，消费者便能从 await 挂起的地方继续往下跑，完成消费的过程。E.g:
经典实例：

```
async def buy_potatos():
    bucket = []
    async for p in take_potatos(50):
        bucket.append(p)
        print(f'Got potato {id(p)}...')

async def take_potatos(num):
    count = 0
    while True:
```

```
        if len(all_potatos) == 0:
            await ask_for_potato()
        potato = all_potatos.pop()
        yield potato
        count += 1
        if count == num:
            break

async def ask_for_potato():
    await asyncio.sleep(random.random())
    all_potatos.extend(Potato.make(random.randint(1, 10)))

Main function:
def main():
    import asyncio
    loop = asyncio.get_event_loop()
    res = loop.run_until_complete(buy_potatos())
    loop.close()
```

既然是异步的，在请求之后不一定要死等，而是可以做其他事情。比如除了土豆，我还想买番茄，这时只需要在事件循环中再添加一个过程：

```
def main():
    import asyncio
    loop = asyncio.get_event_loop()
    res = loop.run_until_complete(asyncio.wait([buy_potatos(),
buy_tomatos()]))
    loop.close()
```

yield from:

把生成器的操作委托给另一个生成器，生成器(gen)的调用方(main 函数)可以直接与子生成器(sub_gen)进行通信：

```
def sub_gen():
    yield 1
    yield 2
    yield 3

def gen():
    return (yield from sub_gen())

def main():
    for val in gen():
        print(val)
```

```
# 1
# 2
# 3
```

@ 用 yield from 容易在表示协程和生成器中混淆，没有良好的语义性，所以在 Python 3.5 推出了更新的 async/await 表达式来作为协程的语法：

```
def main():
    return (yield from coro())

def main():
    return (await coro())
```

asyncio 中 Future 经典实例：

```
import asyncio

future = asyncio.Future()

async def coro1():
    await asyncio.sleep(1)
    future.set_result('data')

async def coro2():
    print(await future)

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait([
    coro1(),
    coro2()
]))
loop.close()
```

两个协程在在事件循环中，协程 coro1 在执行第一句后挂起自身切到 asyncio.sleep，而协程 coro2 一直等待 future 的结果，让出事件循环，计时器结束后 coro1 执行了第二句设置了 future 的值，被挂起的 coro2 恢复执行，打印出 future 的结果'data'。

@ future 可以被 await 证明了 future 对象是一个 Awaitable

@ Knowledge about async、await:

1.Python 3.5 引入

2.使用这些关键字的 函数被称为”原生协程”

3.async def foo()代替 @gen.coroutine（装饰器）; await 代替 yield

4.async 和 await 运行起来更快

5.await 关键字比 yield 关键字功能要少一些. 例如,在一个使用 yield 的协程中， 你可以得到 Futures 列表, 但是在原生协程中, 必须把列表用 tornado.gen.multi 包起来.

6.可以使用 tornado.gen.convert_yielded 来把任何使用 yield 工作的代码转换成使用 await 的形式

总结：

1.完成异步的代码不一定要用 async/await，使用了 async/await 的代码也不一定能做到

异步

2.async/await 是协程的语法糖，使协程之间的调用变得更加清晰，使用 async 修饰的函数调用时会返回一个协程对象。

3.await 只能放在 async 修饰的函数里面使用，await 后面必须要跟着一个协程对象或 Awaitable

4.wait 的目的是等待协程控制流的返回，而实现暂停并挂起函数的操作是 yield。

Part 6:

协程执行者(coroutine runner):

接受任何来自其他框架的 awaitable 对象来进行协程调度

组合了多个框架的应用都使用 Tornado 的协程执行者来进行协程调度 --> 其他的协程运行时可能有很多限制(例如, asyncio 协程执行者不接受来自其他框架的协程). 为了能使用 Tornado 来调度执行 asyncio 的协程, 必须使用 tornado.platform.asyncio.to_asyncio_future 适配器

生成器(generator)、装饰器(@gen.coroutine):

生成器(generator)是一个包含了 yield 关键字的函数

所有的生成器都是异步的

当调用它们的时候,会返回一个生成器对象,而不是一个执行完的结果

@ 生成器通过 yield 表达式传递 Future 对象给装配器，装配器非阻塞的等待这个 Future 对象执行完成, 然后”解开(unwraps)” 这个 Future 对象并传回给生成器

Why we need it?

这种机制使得大多数异步代码从来不会直接接触 Future 类 除非 Future 立即通过异步函数返回给 yield 表达式（此时未解开 Future 类，因此生成器（异步代码）需要自己处理 Future 类）

如何调用协程：

几乎所有的情况下, 任何一个调用协程的函数都必须是协程它自身, 并且在调用的时候使用 yield 关键字 --> 协程一般不会抛出异常: 它们抛出的任何异常将被 Future 捕获，除非 Future 被 yield 解析，才能得到异常。E.g:

```
@gen.coroutine
def divide(x, y):
    return x / y

@gen.coroutine
def good_call():
    # yield 将会解开 divide() 返回的 Future 并且抛出异常
    yield divide(1, 0)
```

协程使用模式：

Callback:

使用回调(callback)而不是 Future 与异步代码进行交互, 把调用包在 Task 中. 这将为你添加一个回调参数并且返回一个可以 yield 的 Future :

```
@gen.coroutine
```

```
def call_task():
    # 注意这里没有传进来 some_function.
    # 这里会被 Task 翻译成
    #     some_function(other_args, callback=callback)
    yield gen.Task(some_function, other_args)
```

调用阻塞函数

从协程调用阻塞函数最简单的方式是使用 ThreadPoolExecutor, 它将返回和协程兼容的 Futures:

```
thread_pool = ThreadPoolExecutor(4)
@gen.coroutine
def call_blocking():
    yield thread_pool.submit(blocking_func, args)
```

并行

协程装饰器能识别列表或者字典对象中各自的 Futures, 并且并行的等待这些 Futures :

```
@gen.coroutine
def parallel_fetch(url1, url2):
    resp1, resp2 = yield [http_client.fetch(url1),
                          http_client.fetch(url2)]
@gen.coroutine
def parallel_fetch_many(urls):
    responses = yield [http_client.fetch(url) for url in urls]
    # 响应是和 HTTPResponses 相同顺序的列表
@gen.coroutine
def parallel_fetch_dict(urls):
    responses = yield {url: http_client.fetch(url)
                       for url in urls}
    # 响应是一个字典 {url: HTTPResponse}
```

交叉存取

有时候保存一个 Future 比立即 yield 它更有用, 所以你可以在等待之前 执行其他操作:

```
@gen.coroutine
def get(self):
    fetch_future = self.fetch_next_chunk()
    while True:
        chunk = yield fetch_future
        if chunk is None: break
        self.write(chunk)
        fetch_future = self.fetch_next_chunk()
        yield self.flush()
```

循环

协程的循环是棘手的, 因为在 Python 中没有办法在 for 循环或者 while 循环 yield 迭

代器,并且捕获 yield 的结果. 相反,你需要将 循环条件从访问结果中分离出来, 下面是一个使用 Motor 的例子:

```
import motor
db = motor.MotorClient().test
@gen.coroutine
def loop_example(collection):
    cursor = db.collection.find()
    while (yield cursor.fetch_next):
        doc = cursor.next_object()
```

在后台运行

PeriodicCallback 通常不使用协程. 相反,一个协程可以包含一个 while True: 循环并使用 tornado.gen.sleep:

```
@gen.coroutine
def minute_loop():
    while True:
        yield do_something()
        yield gen.sleep(60)
# Coroutines that loop forever are generally started with
# spawn_callback().
IOLoop.current().spawn_callback(minute_loop)
```

更复杂的循环:

例如, 上一个循环运行每次花费 60+N 秒, 其中 N 是 do_something() 花费的时间. 为了 准确的每 60 秒运行,使用上面的交叉模式:

```
@gen.coroutine
def minute_loop2():
    while True:
        nxt = gen.sleep(60)     # 开始计时.
        yield do_something()    # 计时后运行.
        yield nxt                 # 等待计时结束.
```

gevent:

gevent 是基于协程的 Python 网络库，它使用 greenlet 在 libev 或 libuv 事件循环的顶部提供高级同步 API。在 python2 以及 python3.3 时代，人们使用协程还得基于 greenlet 或者 gevent，greenlet 机制的主要思想是：生成器函数或者协程函数中的 yield 语句挂起函数的执行，直到稍后使用 next()或 send()操作进行恢复为止。可以使用一个调度器循环在一组生成器函数之间协作多个任务，它的缺点是必须通过安装三方库进行使用，使用时由于封装特性导致性能有一定的流失。

终于在 python3.4 中，我们迎来了 python 的原生协程关键字:Async 和 Await，它们的底层基于生成器函数，使得协程的实现更加方便。

Part6:

异步网络爬虫:

```
        Finally:
        如果存在 finally 子句，则 finally 子句将作为 try 语句结束前的最后一项任务被执行。 finally 子句不论 try 语句是否产生了异常都会被执行
        如果在执行 try 子句期间发生了异常，该异常可由一个 except 子句进行处理。 如果异常没有被某个 except 子句所处理，则该异常会在 finally 子句执行之后被重新引发。
        如果 finally 子句中包含一个 return 语句，则返回值将来自 finally 子句的某个 return 语句的返回值，而非来自 try 子句的 return 语句的返回值。
```

tornado.queues

实现了异步生产者/消费者模式的协程, 类似于通过 Python 标准库的 queue 实现线程模式.

一个 yield Queue.get 的协程直到队列中有值的时候才会暂停

yield Queue.put 的协程直到队列中有空间才会暂停(如果队列设置了最大长度)

Queue.join()等待直到所有任务都执行完毕(即所有元素都调用了 task_done())

E.g: 一个 Queue 从 0 开始对完成的任务进行计数. put 加计数; task_done 减少计数,并等待 join 的主协程取消暂停才完成