Async IO && WebSocket

Parallelism: 同时开始多个操作

Multiprocessing： 等同于高新的任务并行，将任务分为多个子任务，没个子任务再一个CPU(进程)上执行。

Concurrency： 比parallelism更加广泛一些，多个任务再执行的时候，回有重叠

Threading： 多个线程直接换着使用CPU(线程)，一个线程可以同时具备多个进程

> To recap the above, concurrency encompasses both multiprocessing (ideal for CPU-bound tasks) and threading (suited for IO-bound tasks). Multiprocessing is a form of parallelism, with parallelism being a specific type (subset) of concurrency. The Python standard library has offered longstanding support for both of thesethrough its `multiprocessing`, `threading`, and `concurrent.futures` packages.

Async IO: 并不是线程，或者进程，也不是在这两个基础之上的产物，而是单线程的，单进程的，协同式多任务 (cooperative multitasking)，是并发编程的一个风格(a style of concurrent programming)，但是并没有并行。

> 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。

子程序和协程

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。

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。

asyncio

`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

```
import asyncio

async def hello():
    print("Hello world!")
    // 阻塞，程序执行到这里之后，携程会停止执行这个函数(比较耗时的操作会先挂起来)，转而去执行 event loop
```

```
    // 中的其它任务，当所有阻塞的任务或者所有任务都执行结束之后，又会回来去执行这个(比较耗时的操作)任务
    r = await asyncio.sleep(1)
    print("Hello again!")

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

> `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。

```
async def do_some_work(x):
    print('Waiting {}'.format(x))
    return 'Done after {}s'.format(x)

start = now()

coroutine = do_some_work(2)
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(coroutine)

# run_until_complete 直接获取到task的执行结果 task.result()
loop.run_until_complete(task)

print('Task ret: {}'.format(task.result()))
print('TIME: {}'.format(now() - start))
```

协程遇到await，事件循环将会挂起该协程，执行别的协程，直到其他的协程也挂起或者执行完毕，再进行下一个协程的执行。

- event_loop 事件循环：程序开启一个无限的循环，程序员会把一些函数注册到事件循环上。当满足事件发生的时候，调用相应的协程函数。
- coroutine 协程：协程对象，指一个使用async关键字定义的函数，它的调用不会立即执行函数，而是会返回一个协程对象。协程对象需要注册到事件循环，由事件循环调用。
- task 任务：一个协程对象就是一个原生可以挂起的函数，任务则是对协程进一步封装，其中包含任务的各种状态。
- future： 代表将来执行或没有执行的任务的结果。它和task上没有本质的区别
- async/await 关键字：python3.5 用于定义协程的关键字，async定义一个协程，await用于挂起阻塞的异步调用接口。

同步与异步： 同步和异步关注的是**消息通信方式**

> **同步**：
>
> 你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，”我查一下”，然后开始查啊查，等查好了（可能是5秒，也可能是一天）告诉你结果（返回结果）。
>
> **异步**：
>
> 你打电话问书店老板有没有《分布式系统》这本书，，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。

阻塞和非阻塞：阻塞和非阻塞关注的是程序在**等待调用结果（消息，返回值）时的状态**

> 阻塞调用是指调用结果返回之前，当前线程会被挂起(主程序等待调用返回结果)。调用线程只有在得到结果之后才会返回。阻塞对应着同步的方式。 非阻塞调用指在不能立刻得到结果之前(发了任务，主程序就去干别的事情去了)，该调用不会阻塞当前线程。
>
> 你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了， 当然你也要偶尔过几分钟check一下老板有没有返回结果。 在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。
>
> **阻塞和await**
>
> **async**可以定义协程对象，使用**await**可以针对耗时的操作(**也就是会被阻塞的操作**)进行**挂起**，函数让出控制权。协程遇到await，事件循环将会挂起该协程，执行别的协程，直到其他的协程也挂起或者执行完毕，再进行下一个协程的执行。

并发和并行

并发通常指有多个任务需要同时进行，并行则是同一时刻有多个任务执行。

> 用上课来举例就是，并发情况下是一个老师在同一时间段辅助不同的人功课。并行则是好几个老师分别同时辅助多个学生功课。

asyncio实现并发，就需要多个协程来完成任务，每当有任务阻塞的时候就await，然后其他协程继续工作。

```
import asyncio

import time

now = lambda: time.time()

async def do_some_work(x):
    print('Waiting: ', x)

    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)
```

```
start = now()

coroutine1 = do_some_work(1)
coroutine2 = do_some_work(2)
coroutine3 = do_some_work(4)

tasks = [
    asyncio.ensure_future(coroutine1),
    asyncio.ensure_future(coroutine2),
    asyncio.ensure_future(coroutine3)
]

loop = asyncio.get_event_loop()

# run_until_complete 直接获取到task的执行结果 task.result()
loop.run_until_complete(asyncio.wait(tasks))

for task in tasks:
    print('Task ret: ', task.result())

print('TIME: ', now() - start)
```

协程嵌套

> 可以封装更多的io操作过程，这样就实现了嵌套的协程，即一个协程中await了另外一个协程，如此连接起来。

```
import asyncio

import time

now = lambda: time.time()

async def do_some_work(x):
    print('Waiting: ', x)

    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)

async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(4)

    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]
```

```
    dones, pendings = await asyncio.wait(tasks)

    for task in dones:
        print('Task ret: ', task.result())

start = now()

loop = asyncio.get_event_loop()
loop.run_until_complete(main())

print('TIME: ', now() - start)
```

如果使用的是 **asyncio.gather** 创建协程对象，那么await的返回值就是协程运行的结果。

```
async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(2)

    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]

    # 将携程的结果返回会去
    return await asyncio.gather(*tasks)

start = now()

loop = asyncio.get_event_loop()
# 那么最外层的run_until_complete将会返回main协程的结果。
results = loop.run_until_complete(main())

for result in results:
    print('Task ret: ', result)
```

或者返回使用asyncio.wait方式挂起协程。

```
async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(4)

    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
```

```
        asyncio.ensure_future(coroutine3)
    ]

    # 再执行的过程中将挂起的携程返回
    return await asyncio.wait(tasks)

start = now()

loop = asyncio.get_event_loop()
# 执行挂起的那些携程
done, pending = loop.run_until_complete(main())

for task in done:
    print('Task ret: ', task.result())
```

WebSocket

为什么会有WebSocket

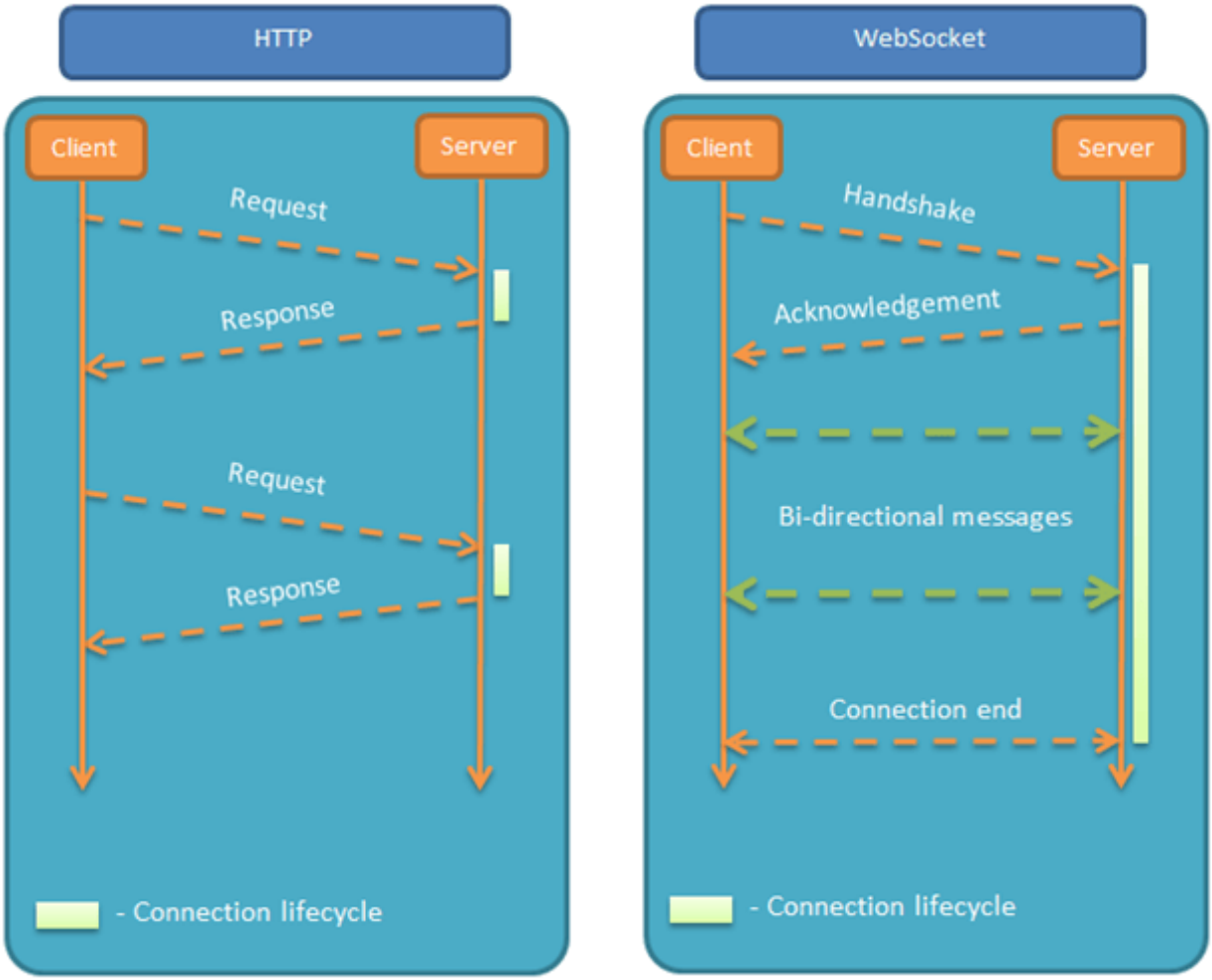
> 新场景的出现：
>
> 1. 客户端和服务端需要保持久长连接
> 2. 服务端去给服务端发消息

HTTP 可以解决上述的第一个场景

> **ajax轮询**: 让浏览器隔个几秒就发送一次请求，询问服务器是否有新信息。
>
> **long poll**: 跟 ajax轮询 差不多，都是采用轮询的方式，不过采取的是阻塞模型（一直打电话，没收到就不挂电话），也就是说，客户端发起连接后，如果没消息，就一直不返回Response给客户端。直到有消息才返回，返回完之后，客户端再次建立连接，周而复始。
>
> **ajax**和**long poll** 的方式都是不断地建立HTTP连接，然后等待服务端处理，可以体现HTTP协议的另外一个特点，**被动性**。总是会有一个 request，然后返回一个response，被动性：服务器不主动联系客户机，猪油客户机发起。
>
> **ajax**和**long poll** 都是很**消耗资源**的，

WebSocket 解决上述的问题

> **被动性** 服务端就可以主动推送信息给客户端啦。



1. 第一次发送HTTP 请求，客户端和服务端握手成功(HTTP只负责建立WebSocket连接)
2. 然后协议升级为WebSocket，然后服务器也是可以给客户端发送请求了。二者保持着长连接
3. 使用http协议断开联系

webSocket 是什么

> 可以把 WebSocket 看成是 HTTP 协议为了支持长连接所打的一个大补丁
>
> 约定了一个通信的规范，通过一个握手的机制，客户端和服务器之间能建立一个类似tcp的连接，从而方便它们之间的通信。在websocket出现之前，web交互一般是基于http协议的短连接或者长连接。websocket是一种全新的协议，不属于http无状态协议，协议名为"ws"，这意味着一个websocket连接地址会是这样的写法：ws://**。websocket协议本质上是一个基于tcp的协议。

webSocket Python 中怎么玩 : Django channels

> unning Django itself in a synchronous mode but handling connections and sockets asynchronously,

> 加入 channels之后，django项目就可以写异步编码了，而且项目不仅仅是处理http，更是能去处理websocket 等协议

Scopes and Events

> **scope is a set of details about a single incoming connection** - such as the path a web request was made from, or the originating IP address of a WebSocket, or the user messaging a chatbot - and persists throughout the connection.

> Consumers receive the connection's `scope` when they are initialised, which contains a lot of the information you'd find on the `request` object in a Django view. It's available as `self.scope` inside the consumer's methods.Scopes are part of the ASGI specification, but here are some common things you might want to use:
>
> - `scope["path"]`, the path on the request. *(HTTP and WebSocket)*
> - `scope["headers"]`, raw name/value header pairs from the request *(HTTP and WebSocket)*
> - `scope["method"]`, the method name used for the request. *(HTTP)*

What is a Consumer

> A consumer is the **basic unit** of Channels code. We call it a *consumer* as it *consumes events*, but you can think of it as its own tiny little application. When a request or new socket comes in, Channels will follow its routing table - we'll look at that in a bit - find the right consumer for that incoming connection, and start up a copy of it.

channel layers

> let them send messages between each other either one-to-one or via a broadcast system called groups.
>
> Channel layers allow you to talk between different instances of an application. They're a useful part of making a distributed realtime application if you don't want to have to shuttle all of your messages or events through a database.
>
> Channel layers have a purely async interface (for both send and receive); you will need to wrap them in a converter if you want to call them from synchronous code (see below).
>
> 通道不附带任何可以开箱即用的通道层(Channels does not ship with any channel layers you can use out of the box,）每一个都依赖于不同在网络间传输数据方式(each one depends on a different way of transporting data across a network)，官方推荐使用 **channels_redis** 方式，所以在执行程序的环境中需要有 *redis* 的数据库

By default the `send()`, `group_send()`, `group_add()` and other functions are async functions, meaning you have to `await` them. If you need to call them from synchronous code, you'll need to use the handy `asgiref.sync.async_to_sync` wrapper: