1 理论篇

1-1 课程介绍

1 课程核心内容：

- python异步编程：asyncio
- 理论讲解+源码阅读+实战操作

2 课程基础

- python基础
- python开发环境（推荐虚拟环境）

```
# 1 新建虚拟环境
python -m venv tutorial-env

# 2 激活虚拟环境windowns
tutorial-env\Scripts\activate.bat
```

1-2 什么是同步和异步

在正式进入课程前需要弄清楚三组概念：

- 同步和异步
- 串行和并行
- 并行和并发

1 同步和异步：

- 同步：做一件事情一定要看着它、把它做完，然后再开始下一件事。
- 异步：做一件事情可以把它交给别人做（不会一直等着它），继续做下一件事。

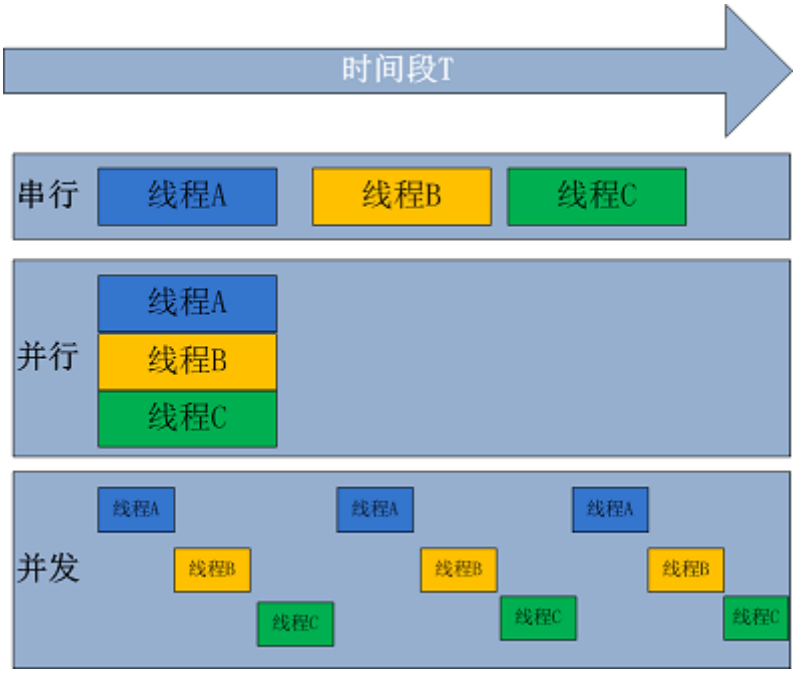
2 串行和并行：

- 串行：一件事一件事的做。
- 并行：两件事同时做，即同一个时间点有多个人在干活。

)

3 并行和并发

- 并行：同一时间点有多个任务在执行。
- 并发：一段时间内，有多个任务轮流执行；且同一个时间点依然只执行一个任务。



通常大家说的高并发指的就是【并发】这个概念，充分压榨CPU资源。

python中实现并发的手段有三个：进程、线程、协程。

1-3 什么是协程

- CPU在执行任务时，整整干活的是线程，当一个线程遇到IO时会交出执行权，切换到另个一个线程。
- 协程（Coroutine），也称为微线程，是一种用户态内的上下文切换技术，即在一个线程内切换被执行的代码块。代码级别的切换。

示例1：同步执行的代码，代码块依次执行

```
def f1():
    print(1)
    print(2)


def f2():
    print(3)
    print(4)


f1()
f2()

# 依次打印：1 2 3 4
```

示例2：使用`yield`实现的协程，实现代码块的切换执行。

```
def f1():
    yield 1
```

```
    yield from f2()
    yield 2


def f2():
    yield 3
    yield 4


generator = f1()        # f1()是生成器

for i in generator:
    print(i)            # 依次打印：1 3 4 2
```

1-4 第三方包实现协程

示例3：使用第三方包`greenlet`实现协程

```
from greenlet import greenlet


def f1():
    print(1)
    gr2.switch()
    print(2)
    gr2.switch()


def f2():
    print(3)
    gr1.switch()
    print(4)


gr1 = greenlet(f1)
gr2 = greenlet(f2)

gr1.switch()    # 依次打印：1 3 2 4
```

示例4：使用第三方包gevent实现协程（内部基于greenlet），遇到IO会自动切换。

```
import gevent


def f1():
    print(1)
    gevent.sleep(2)     # 注意不使用time.sleep(2)
    print(2)
```

```
def f2():
    print(3)
    gevent.sleep(2)
    print(4)


g1 = gevent.spawn(f1)
g2 = gevent.spawn(f2)


gevent.joinall([g1, g2])    # 等待所有函数运行结束，依次打印1 3 -睡2s- 2 4
```

> 1.gevent基于greenlet
>
> 2.猴子补丁：from gevent import monkey; monkey.patch_all()

1-5 asyncio实现协程

Python3.4之前官方未提供协程的类库，一般使用gevent实现协程达到异步编程的目的。

Python3.4之后官方推出asyncio模块，有如下几个特点：

- 单线程
- 事件循环
- 适用于IO密集型场景

示例：通过asyncio实现协程

```
import asyncio


@asyncio.coroutine
def f1():
    print(1)
    yield from asyncio.sleep(2)     # 遇到IO睡2s
    print(2)


@asyncio.coroutine
def f2():
    print(3)
    yield from asyncio.sleep(2)     # 遇到IO睡2s
    print(4)


tasks = [
    asyncio.ensure_future(f1()),
    asyncio.ensure_future(f2())
]

loop = asyncio.get_event_loop()                 # 得到一个事件循环
loop.run_until_complete(asyncio.wait(tasks))    # 等待所有任务执行完毕
```

```
# 打印结果：1 3 -睡2s- 2 4
```

- 注意1：asyncio内部是基于yield实现的协程，本质上在事件循环内切换执行任务。
- 注意2：@asyncio.coroutine这个装饰器在python3.8中将会被弃用，推荐使用async def

1-6 async/await实现协程

- Python3.5官方推出 async 和 await 关键字，目的是让协程代码可以更加简洁。

```
import asyncio


async def f1():
    print(1)
    await asyncio.sleep(2)       # 等 可以等待的对象
    print(2)


async def f2():
    print(3)
    await asyncio.sleep(2)
    print(4)


tasks = [
    asyncio.ensure_future(f1()),
    asyncio.ensure_future(f2())
]

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(tasks))
# 打印结果：1 3 -睡2s- 2 4
```

- 注意1：await只能用在async def 内，但在ipython中可以直接使用

1-7 体验async异步爬图片

- 协程的主要用途在IO密集型场景，本节通过爬虫带着大家体验下异步爬虫的高效。

示例1：同步爬虫

```
import requests


def download_img(url):
    file_name = url.rsplit('/')[-1]
    print(f"下载图片：{file_name}")
    response = requests.get(url)
    with open(file_name, mode='wb') as file:
```

```
        file.write(response.content)
    print(f"下载完成：{file_name}")


def main():
    urls = [
        "https://tenfei05.cfp.cn/creative/vcg/800/new/VCG41560336195.jpg",
        "https://tenfei03.cfp.cn/creative/vcg/800/new/VCG41688057449.jpg",
        ]
    for item in urls:
        download_img(item)


main()

# 遍历下载列表，第一张图片下载结束，再开始下载下一张图片。
```

示例2：使用asyncio异步下载图片

```
import asyncio
import aiohttp


async def download_img(session, url):
    file_name = url.rsplit('/')[-1]
    print(f"下载图片：{file_name}")
    response = await session.get(url, ssl=False)
    content = await response.content.read()
    with open(file_name, mode='wb') as file:
        file.write(content)
    print(f"下载完成：{file_name}")


async def main():
    urls = [
        "https://tenfei05.cfp.cn/creative/vcg/800/new/VCG41560336195.jpg",
        "https://tenfei03.cfp.cn/creative/vcg/800/new/VCG41688057449.jpg",
    ]
    async with aiohttp.ClientSession() as session:
        tasks = [asyncio.ensure_future(download_img(session, url)) for url in
urls]
        await asyncio.wait(tasks)


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

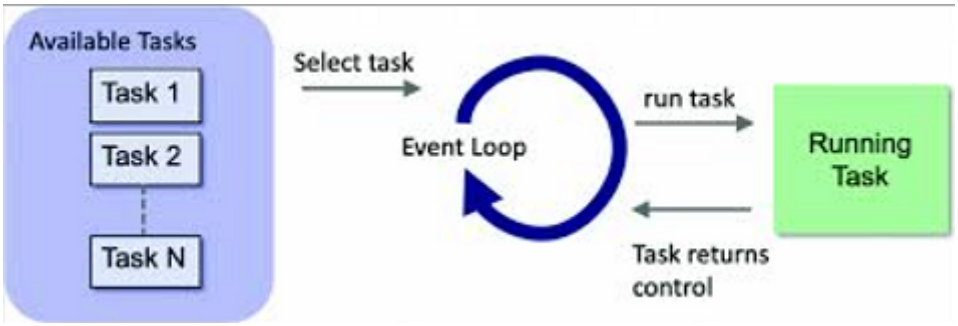
- 总结：IO密集型场景，使用协程优势明显。IO等待时切换执行其他任务，进而提高效率。
- 基于`async` & `await`关键字的协程可以实现异步编程，这也是目前的主流方式。

1-8 事件循环

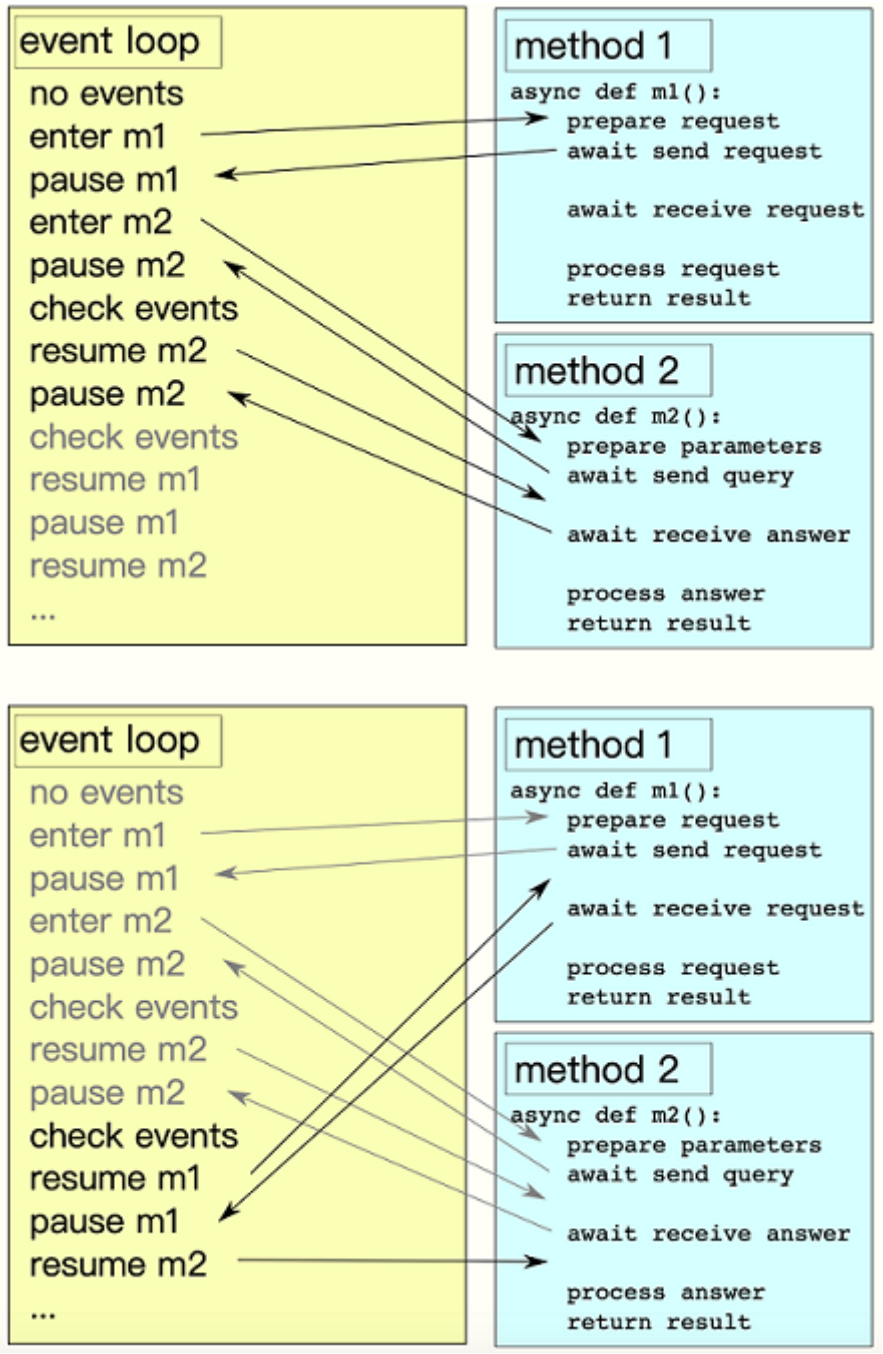
- 事件循环是asyncio的核心。在python异步应用中，每一个异步任务都是在事件循环中被执行的。
- 简单理解，可以把事件循环当成一个while循环，循环内部执行任务，当一个任务遇到IO挂起时，就会立即切换执行另一个任务，如此反复，在特定条件下会终止循环。

> 官网：https://docs.python.org/3/library/asyncio-eventloop.html

- 图解事件循环：



注意：程序遇到`await`关键词时，程序会挂起，事件循环会再找一个任务开始执行。



- 创建事件循环

```
import asyncio
asyncio.get_event_loop()
```

- 推荐使用 `asyncio.run()` 来运行事件循环

```python
import asyncio


async def f1():
    await asyncio.sleep(5)

# loop = asyncio.get_event_loop()
# loop.run_until_complete(f1())

asyncio.run(f1())               # run等价于上面两行代码
```

1-9 协程函数和协程对象

- 协程函数：通过`async def`定义的函数
- 协程对象：协程函数执行后返回的对象

注意：

- 调用协程函数，函数内部代码不会执行，只是会返回一个协程对象。
- 想要执行协程函数内部代码，需要配合事件循环。

```python
import asyncio


async def f1():
    await asyncio.sleep(5)

# loop = asyncio.get_event_loop()    # 获取一个事件循环
# loop.run_until_complete(f1())      # 将协程函数当做任务添加到事件循环中

asyncio.run(f1())               # run等价于上面两行代码
```

注意：

- run会新建一个事件循环，任务结束时会自动关闭它。
- run应该作为异步程序的唯一入口，只能被调用一次。

1-10 await关键字

- await关键字等待三种类型：协程对象、task对象、future对象（官网原话）
- await是一个只能在协程函数中使用的关键字，用于遇到IO操作时挂起当前任务，
- 当前任务挂起过程后，事件循环可以去执行其他的任务，
- 当前协程IO处理完成时，可以再次切换回来执行await之后的代码。

示例1：

```python
import asyncio


async def f1():
    print(1)
    await asyncio.sleep(2)
    print(2)

asyncio.run(f1())
```

示例2：接收返回值

```python
import asyncio


async def f1():
    print(1)
    await asyncio.sleep(2)
    print(2)
    return 123

async def f2():
    print(3)
    await asyncio.sleep(2)
    print(4)
    return 456


async def main():
    print("main start")
    res1 = await f1()       # 等f1执行后再执行f2，因为目前事件循环中只有一个任务（main）
    print(res1)
    res2 = await f2()
    print(res2)
    print("main end")


asyncio.run(main())
```

1-11 task对象

- 在程序想要创建多个任务对象，需要使用task对象来实现。
- task是用来调度协程并发执行的，也就是说事件循环中有多个task才能有并发的效果。

> 官网：https://docs.python.org/3/library/asyncio-task.html
>
> *Tasks* are used to schedule coroutines *concurrently*.
>
> When a coroutine is wrapped into a *Task* with functions like `asyncio.create_task()` the coroutine is automatically scheduled to run soon。

本质上，`asyncio.create_task()`是将协程对象封装成task对象，并立即加入事件循环，同时追踪协程的状态。

示例1：

```
import asyncio


async def f1():
    print(1)
    await asyncio.sleep(2)
    print(2)


async def f2():
    print(3)
    await asyncio.sleep(2)
    print(4)


async def main():

    print("main start")

    task1 = asyncio.create_task(f1())    # 事件循环加入task1
    task2 = asyncio.create_task(f2())    # 事件循环加入task2

    await task1     # 遇到await,挂起task1，寻找其他可执行的任务
    await task2     # 遇到await，挂起task2，寻找其他可执行的任务
    print("main end")


asyncio.run(main())
```

- 注意：示例1中，在并不是task1执行结束后才执行task2

示例2：

```
import asyncio


async def f1():
    print(1)
    await asyncio.sleep(1)
    print(2)


async def f2():
    print(3)
    await asyncio.sleep(1)
    print(4)
```

```python
async def main():

    print("main start")

    tasks = [
        asyncio.create_task(f1()),
        asyncio.create_task(f2()),
    ]

    await asyncio.wait(tasks)

    print("main end")


asyncio.run(main())
```

1-12 await错误使用

如果不理解asyncio协程，很容易错误使用await

- 错误1：create_task时需要存在一个已经在运行的事件循环。

```python
import asyncio


async def f1():
    print(1)
    await asyncio.sleep(1)
    print(2)


async def f2():
    print(3)
    await asyncio.sleep(3)
    print(4)


tasks = [
    asyncio.create_task(f1()),
    asyncio.create_task(f2()),
]

asyncio.run(asyncio.wait(tasks))
```

错误1解决办法：

```python
# 办法1：tasks内存协程对象，因为await时也会转成task
tasks = [f1(), f2()]
asyncio.run(asyncio.wait(tasks))

# 办法2：函数内部使用create_task，因为此时已经有loop了
async def main():
    tasks = [
        asyncio.create_task(f1()),
        asyncio.create_task(f2()),
    ]
    await asyncio.wait(tasks)

asyncio.run(main())
```

- 错误2：错误使用await造成同步执行

```python
import asyncio
import time


async def f1():
    print(1)
    await asyncio.sleep(2)
    print(2)


async def f2():
    print(3)
    await asyncio.sleep(2)
    print(4)


async def main():
    start = time.time()
    await f1()
    await f2()
    print(time.time() - start)


asyncio.run(main())
```

错误2解决办法：

```python
# 得到协程对象或者task，然后使用asyncio.wait() 或者asyncio.gather()
async def main():
    start = time.time()
    tasks = [f1(), f2()]
    await asyncio.wait(tasks)       # asyncio.gather(*tasks)
    print(time.time() - start)

asyncio.run(main())
```

1-13 wait和gather

相同点：

- 从功能上看，`asyncio.wait` 和 `asyncio.gather` 实现的效果是相同的。

不同点：

- 使用方式不一样，wait需要传一个可迭代对象，gather传多个元素
- wait比gather更底层，gather可以直接得到结果，wait先得到future再得到结果
- wait可以通过设置timeout和return_when来终止任务
- gather可以给任务分组，且支持组级别的取消任务

示例1：

```
import asyncio


async def f1():
    print(1)
    await asyncio.sleep(3)
    print(2)
    return "f1"


async def f2():
    print(3)
    await asyncio.sleep(1)
    print(4)
    return "f2"


async def main():
    tasks = [
        asyncio.create_task(f1()),
        asyncio.create_task(f2()),
    ]
    done, pending = await asyncio.wait(tasks, timeout=2)
    for d in done:
        print(d.done(), d.result())
    for p in pending:
        print(p.done())

    # done = await asyncio.gather(*tasks)
    # done=['f1', 'f2']


asyncio.run(main())
```

示例2：gather分组

```python
import asyncio


async def f1():
    print(1)
    await asyncio.sleep(3)
    print(2)
    return "f1"


async def f2():
    print(3)
    await asyncio.sleep(3)
    print(4)
    return "f2"


async def main():
    group1 = asyncio.gather(f1(), f1())
    group2 = asyncio.gather(f2(), f2())
    group1.cancel()
    all_groups = await asyncio.gather(group1, group2, return_exceptions=True)
    print(all_groups)

asyncio.run(main())
```

1-14 create_task和ensure_future

相同点：

- 都可以创建任务

不同点：

- create_task时必须有一个已经在运行的事件循环，内部调用loop.create_task()
- ensure_future时如果没有事件循环，内部会自动新建一个，且内部也调用loop.create_task()
- create_task可以给task取名字

```python
import asyncio


async def f1():
    print(1)
    await asyncio.sleep(2)
    print(2)
    return "f1"


async def f2():
    print(3)
    await asyncio.sleep(2)
    print(4)
    return "f2"
```

```python
async def main():
    tasks = [
        asyncio.create_task(f1(), name="t1"),
        asyncio.create_task(f2(), name="t2"),
        asyncio.ensure_future(f1())
    ]
    for t in tasks:
        print(t.get_name())
    await asyncio.wait(tasks)


asyncio.run(main())
```

> 注意：创建task对象，使用create_task和ensure_future，不要直接使用Task
>
> https://docs.python.org/3/library/asyncio-task.html#task-object

1-15 回调函数

- 给任务绑定回调函数：在任务执行完毕的时候可以获取任务的结果，也可以做一些其他操作。绑定方式：`task.add_done_callbac()`
- 回调函数的最后一个参数是future对象，通过该对象可以获取执行结果。
- 回调函数如果需要多个参数，可以使用偏函数`functools.partial`

```python
import asyncio
from functools import partial
from asyncio import Future


async def f1():
    print(1)
    await asyncio.sleep(2)
    print(2)
    return "f1"


def callback1(future: Future):
    print("结束了:", future.result())


def callback2(t, future):
    print("结束了:", t, future.result())


async def main():
    task1 = asyncio.create_task(f1())
    task1.add_done_callback(callback1)
    task1.add_done_callback(partial(callback2, 1111))
    await asyncio.gather(task1)

asyncio.run(main())
```

1-16 run_until_complete和run_forever

- run_forever 会一直循环，直到事件循环被终止。
- run_until_complete 会在传进去的future结束后自动终止事件循环。内部调用的run_forever ，通过给任务绑定回调函数，在任务结束后终止loop

示例1：run_until_complete 是因为传进去的future会绑定一个回调函数

```
future.add_done_callback(_run_until_complete_cb)

def _run_until_complete_cb(fut):
    if not fut.cancelled():
        exc = fut.exception()
        if isinstance(exc, (SystemExit, KeyboardInterrupt)):
            # Issue #22429: run_forever() already finished, no need to
            # stop it.
            return
    futures._get_loop(fut).stop()       # 得到事件循环并终止它
```

示例2：run_forever 我们也可以参考源码，绑定回调函数终止loop

```
import asyncio
from asyncio import Future


async def f1():
    print(1)
    await asyncio.sleep(3)
    print(2)
    return "f1"


def callback(f: Future):
    f.get_loop().stop()


loop = asyncio.get_event_loop()

task = loop.create_task(f1())
task.add_done_callback(callback)

loop.run_forever()
```

1-17 future对象

> 官网：
>
> A `Future` is a special **low-level** awaitable object that represents an **eventual result** of an asynchronous operation.
>
> When a Future object is *awaited* it means that the coroutine will wait until the Future is resolved in some other place.
>
> Future objects in asyncio are needed to allow callback-based code to be used with async/await.
>
> Normally **there is no need** to create Future objects at the application level code.

总结：

- Future对象是一个低层级的可以等待的对象
- 当future对象设置了返回值后，才能结束等待
- future对象可以设置回调函数，一般我们使用Task对象而不是用Future对象（Task是Future的子类）。Task对Future进行扩展，自动执行`set_result`，从而实现自动结束。

示例1：单纯等待future对象，将永远阻塞

```
import asyncio


async def main():
    loop = asyncio.get_running_loop()
    fut = loop.create_future()
    await fut


asyncio.run(main())
```

示例2：增加一个任务，2s后把future对象终止

```
import asyncio


async def set_result(fut):
    await asyncio.sleep(2)
    fut.set_result("可以结束了")


async def main():
    loop = asyncio.get_running_loop()
    fut = loop.create_future()

    task = asyncio.create_task(set_result(fut))

    data = await asyncio.gather(fut, task)
    print(data)
```

```
asyncio.run(main())
```

1-18 concurrent.futures实现并发

`concurrent.futures` 模块提供异步执行可调用对象高层接口，使用线程池 `ThreadPoolExecutor` 或进程池 `ProcessPoolExecutor` 来实现异步。目的是保证服务稳定运行的前提下提供最大的并发能力。

concurrent.futures的基本使用：

```
from concurrent.futures import Future
from concurrent.futures import ThreadPoolExecutor
import time


def task(a, b):
    time.sleep(2)
    return a + b


def callback(f: Future):
    print(f.result())


pool = ThreadPoolExecutor(5)    # 生成一个线程池，大小为5

print("开始了")
for i in range(5):
    pool.submit(task, i, i).add_done_callback(callback)
    # 异步执行，绑定回调函数获取结果

print("结束了")
```

concurrent.futures.Future和asyncio.Future本质上是没有联系的，用来处理不同的使用场景。

但是python在asyncio中提供了一个工具，可以将concurrent.futures.Future对象包装成asyncio.Future，如此一来的好处是：线程池/进程池 + 协程 搭配实现协程。
`loop.run_in_executor()`

```
    def run_in_executor(self, executor, func, *args):
        self._check_closed()
        if self._debug:
            self._check_callback(func, 'run_in_executor')
        if executor is None:
            executor = self._default_executor
            if executor is None:
                executor = concurrent.futures.ThreadPoolExecutor()
                self._default_executor = executor
        return futures.wrap_future(
            executor.submit(func, *args), loop=self)
```

1-19 线程池和协程混合实现异步案例

问题：asyncio是单线程的，通过事件循环实现异步，那为什么还需要使用线程池(多线程)?

- 其实，通过异步解决并发的方式有多种，比如多进程、多线程、协程。
- 这个asyncio实现的协程也是解决方案之一，并且在asyncio中也提供了配合使用多线程的途径。这样做的原因也很简单，**就asyncio中所有的操作都必须是异步非阻塞的，但是当我们不得不使用一个第三方包，而这个包又不支持asyncio，此时就需要配合线程池一块使用了。**
- 示例1：同步

```
import time

def download_img(url):
    print(f"下载图片：{url}")
    time.sleep(1)
    print(f"下载完成：{url}")


def main():
    for i in range(10):
        download_img(i)

main()
```

- 示例2：线程池

```
import time
from concurrent.futures import ThreadPoolExecutor


def download_img(url):
    print(f"下载图片：{url}")
    time.sleep(1)
    print(f"下载完成：{url}")


def main():
    executor = ThreadPoolExecutor(5)
    for i in range(10):
```

```
        executor.submit(download_img, i)

main()
```

- 示例3：run_in_executor

```
import asyncio
import time
from concurrent.futures import ThreadPoolExecutor


def download_img(url):
    print(f"下载图片：{url}")
    time.sleep(1)
    print(f"下载完成：{url}")


async def main():
    executor = ThreadPoolExecutor(5)
    loop = asyncio.get_running_loop()
    tasks = []
    for i in range(10):
        t = loop.run_in_executor(executor, download_img, i)
        tasks.append(t)

    await asyncio.wait(tasks)


asyncio.run(main())
```

2 实战篇

2-1 异步上下文管理器

- 回顾：什么是上下文管理器：实现了`__enter__`和`__exit__`的对象

```
import time


class ContextManager:
    def __init__(self):
        self.conn = None

    def action(self):
        return self.conn

    def __enter__(self):
        # 链接数据库
        time.sleep(1)
```

```
        self.conn = "OK"
        return self

    def __exit__(self, exc_type, exc, tb):
        # 关闭数据库链接
        self.conn = "CLOSE"


def main():
    with ContextManager() as cm:
        result = cm.action()
        print(result)


main()
```

- 异步上下文管理器：实现`__aenter__`和`__aexit__`，可以使用`async with`

```
import asyncio


class ContextManager:
    def __init__(self):
        self.conn = None

    async def action(self):
        return self.conn

    async def __aenter__(self):
        # 链接数据库
        await asyncio.sleep(1)
        self.conn = "OK"
        return self

    async def __aexit__(self, exc_type, exc, tb):
        # 关闭数据库链接
        self.conn = "CLOSE"


async def main():
    async with ContextManager() as cm:
        result = await cm.action()
        print(result)


asyncio.run(main())
```

2-2 异步迭代器

回顾：什么是迭代器：实现了`__iter__`和`__next__`的对象

```
# 自己实现一个range()
```

```
class MyRange:
    def __init__(self, total=0):
        self.total = total
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count < self.total:
            x = self.count
            self.count += 1
            return x
        else:
            raise StopIteration


for i in MyRange(10):
    print(i)
```

- 异步迭代器：需要实现`__aiter__`和`__anext__`，可以使用`async for`

```
import asyncio


class MyRange:
    def __init__(self, total=0):
        self.total = total
        self.count = 0

    def __aiter__(self):
        return self

    async def __anext__(self):
        if self.count < self.total:
            await asyncio.sleep(1)
            x = self.count
            self.count += 1
            return x
        else:
            raise StopAsyncIteration


async def main():
    async for i in MyRange(10):
        print(i)


asyncio.run(main())
```

2-3 异步操作mysql

在python中操作mysql我们通常使用pymysql，但是在异步的世界中我们使用aiomysql。这其中：连接、执行SQL、关闭都涉及网络IO请求，使用asycio异步的方式可以在IO等待时去做一些其他任务，从而提升性能。

> 官网：https://aiomysql.readthedocs.io/en/latest/index.html

- 安装`aiomysql`

```
pip install aiomysql
```

- 示例1：简单使用

```
import asyncio
import aiomysql


async def main():
    # 连接MySQL
    conn = await aiomysql.connect(
        host="127.0.0.1", port=3306, user="root", password="12345", db="mysql"
    )
    # 创建CURSOR
    cur = await conn.cursor()
    # 执行SQL
    await cur.execute("SELECT Host,User FROM user")
    # 获取SQL结果
    result = await cur.fetchall()
    print(result)
    # 关闭CURSOR
    await cur.close()
    # 关闭连接
    conn.close()


asyncio.run(main())
```

- 示例2：连接池

```
import asyncio
import aiomysql


async def go():
    pool = await aiomysql.create_pool(host='127.0.0.1', port=3306,
                                      user='root', password='12345',
                                      db='mysql', autocommit=False)

    async with pool.acquire() as conn:
        cur = await conn.cursor()
        await cur.execute("SELECT 10")
        ret = await cur.fetchone()
        print(ret)

    pool.close()
```

```
    await pool.wait_closed()


loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

2-4 异步操作redis

异步世界操作redis，一般常使用aioredis：https://aioredis.readthedocs.io/en/latest/

- 支持连接池、事务、管道等等。

```
pip install aioredis
```

- 示例

```
import asyncio
import aioredis


async def main():
    # Redis client bound to single connection (no auto reconnection).
    redis = aioredis.from_url(
        "redis://localhost", encoding="utf-8", decode_responses=True
    )
    async with redis.client() as conn:
        await conn.set("my-key", "value")
        val = await conn.get("my-key")
    print(val)


async def redis_pool():
    # Redis client bound to pool of connections (auto-reconnecting).
    redis = aioredis.from_url(
        "redis://localhost", encoding="utf-8", decode_responses=True
    )
    await redis.set("my-key", "value")
    val = await redis.get("my-key")
    print(val)


if __name__ == "__main__":
    asyncio.run(main())
    asyncio.run(redis_pool())
```

注意：

- 目前最先版已经启用了 `create_redis()`，`create_redis_pool()`，`create_pool()`
- from_url是一个函数，内部调用Redis.from_url（本身是一个类方法），返回一个连接池
- redis.client()返回一个具体的redis客户端，且可以使用异步上下文管理器
- 我们也可以自己通过Redis类实例化客户端（不推荐）
- 看看源码

```
    async def execute_command(self, *args, **options):
        """Execute a command and return a parsed response"""
        await self.initialize()
        pool = self.connection_pool
        command_name = args[0]
        conn = self.connection or await pool.get_connection(command_name,
**options)
        try:
            await conn.send_command(*args)
            return await self.parse_response(conn, command_name, **options)
        except (ConnectionError, TimeoutError) as e:
            await conn.disconnect()
            if not (conn.retry_on_timeout and isinstance(e, TimeoutError)):
                raise
            await conn.send_command(*args)
            return await self.parse_response(conn, command_name, **options)
        finally:
            if not self.connection:
                await pool.release(conn)
```

2-5 异步发请求

传统python代码中发请求(如爬虫)我们一般使用requests模板，但asyncio和Python的异步HTTP客户端，常用的两个包是 `aiohttp` 和 `httpx`

- aiohttp：**异步支持客户端和服务端，功能强大。**https://docs.aiohttp.org/en/stable/
- httpx：**httpx 既能发送同步请求，又能发送异步请求。**https://www.python-httpx.org/async/
- 性能上aiohttp优于httpx
- 基本使用

```
import asyncio

import aiohttp
import httpx


async def aiohttp_demo():
    print("start: aiohttp")
    async with aiohttp.ClientSession() as session:
        async with session.get('http://baidu.com') as response:

            print("Status:", response.status)
            print("Content-type:", response.headers['content-type'])

            html = await response.text()
            print("Body:", html[:15], "...")


async def httpx_demo():
    print("start: httpx")
    async with httpx.AsyncClient() as client:
        resp = await client.get('http://baidu.com')
```

```
        print("status-code:", resp.status_code)
        print(resp.text)


tasks = [aiohttp_demo(), httpx_demo()]
asyncio.run(asyncio.wait(tasks))
```

2-6 异步文件操作

传统代码中使用open操作文件，异步世界中使用aiofile，它和python原生open 一致，而且可以支持异步迭代。

> 官网：https://pypi.org/project/aiofile/

- 基本文件操作：async_open函数，返回类文件对象，有异步write、read、seek、readline等。

```
import asyncio

from aiofile import async_open


async def main():
    async with async_open("test.txt", 'w+') as afp:
        await afp.write("Hello ")
        await afp.write("world")
        afp.seek(0)

        print(await afp.read())

        await afp.write("Hello from\nasync world")
        afp.seek(0)
        print(await afp.readline())
        print(await afp.readline())


asyncio.run(main())
```

- 示例2：线性读取或写入文件

```
import asyncio
from aiofile import AIOFile, Reader, Writer


async def main():
    async with AIOFile("text.txt", 'w+') as afp:
        writer = Writer(afp)
        reader = Reader(afp, chunk_size=8)

        await writer("Hello")
        await writer(" ")
        await writer("World")
```

```
        await afp.fsync()

        async for chunk in reader:
            print(chunk)


asyncio.run(main())
```

- 示例3：如果希望按行读取则使用LineReader，不要使用async_open

```
import asyncio
from aiofile import AIOFile, LineReader, Writer


async def main():
    async with AIOFile("text.txt", 'w+') as afp:
        writer = Writer(afp)

        await writer("Hello")
        await writer(" ")
        await writer("World")
        await writer("\n")
        await writer("\n")
        await writer("From async world")
        await afp.fsync()

        async for line in LineReader(afp):
            print(line)


asyncio.run(main())
```

2-7 异步发邮件

异步世界的发邮件模块

> aiosmtplib is an asynchronous SMTP client for use with asyncio.
>
> 官网：https://aiosmtplib.readthedocs.io/en/stable/overview.html

- 快速使用

```
import asyncio
from email.message import EmailMessage

import aiosmtplib


async def main():
    message = EmailMessage()
```

```
    message["From"] = "root@localhost"
    message["To"] = "somebody@example.com"
    message["Subject"] = "Hello World!"
    message.set_content("Sent via aiosmtplib")
    await aiosmtplib.send(message, hostname="127.0.0.1", port=25)


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

- 登录认证：提供用户名和密码（如使用qq邮箱）

> QQ邮箱：
>
> hostname="smtp.qq.com"
>
> port=587
>
> username="你的QQ邮箱号"
>
> password="授权码"

```
await send(
    message,
    hostname="127.0.0.1",
    port=1025,
    username="test",
    password="test"
)
```

- 如果想在发邮件时，有更多的控制，可以直接使用 SMTP

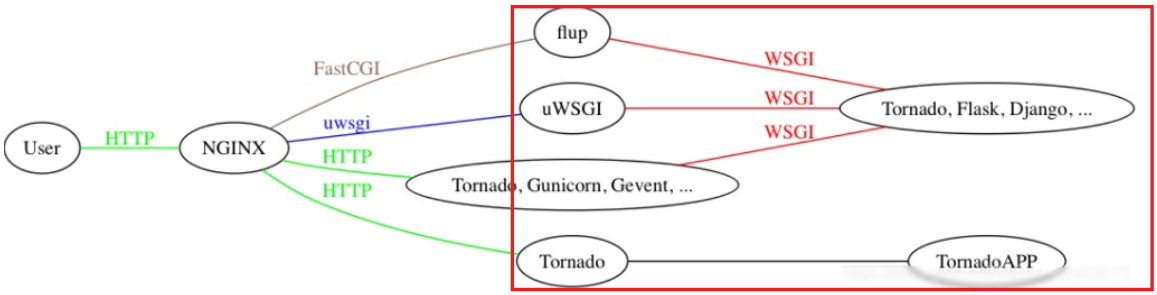
```
import asyncio
from email.message import EmailMessage

from aiosmtplib import SMTP


async def say_hello():
```

```
    message = EmailMessage()
    message["From"] = "root@localhost"
    message["To"] = "somebody@example.com"
    message["Subject"] = "Hello World!"
    message.set_content("Sent via aiosmtplib")

    smtp_client = SMTP(hostname="127.0.0.1", port=1025)
    async with smtp_client:
        await smtp_client.send_message(message)

event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(say_hello())
```

2-8 asgi

在说asgi之前，不得不回顾下什么是wsgi

wsgi是为 Python 定义的 Web 服务器和 Web 应用程序或框架之间的一种简单而通用的接口。



简单来说，wsgi规定了web服务器和web应用之间的通信协议。

```
def app(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return '<h1>Hello, web!</h1>'

# environ：一个包含所有HTTP请求信息的dict对象；
# start_response：一个发送HTTP响应的函数。
```

整个app()函数本身没有涉及到任何解析HTTP的部分，也就是说，底层代码不需要我们自己编写，我们只负责在更高层次上考虑如何响应请求就可以了。比如**Diango和Flask等等web框架属于web应用这层。**

app()函数必须由WSGI服务器来调用。有很多符合**WSGI规范的服务器，比如uwsgi、gunicorn、gevent**等，Python内置了一个WSGI服务器，这个模块叫wsgiref，它是用纯Python编写的WSGI服务器的参考实现。

> 补充：一般web框架不会使用函数式的app接口，会采用面向对象的方式封装成类的形式，比如Django内部定义了一个WSGIHandler类，通过 __call__(self, environ, start_redponse)实现了wsgi接口协议。

wsgi缺点：wsgi是一个单调用、同步接口，即输入一个请求，返回一个响应。这个模式无法支持长连接或者WebSocket这样的连接。即使我们想办法将应用改成异步，还有另一个限制：一个URL对应一个请求，而HTTP/2、Websocket等在一个URL里会出现多个请求。

> WSGI applications are a single, synchronous callable that takes a request and returns a response; this doesn't allow for long-lived connections, like you get with long-poll HTTP or WebSocket connections.
>
> Even if we made this callable asynchronous, it still only has a single path to provide a request, so protocols that have multiple incoming events (like receiving WebSocket frames) can't trigger this.

asgi：异步服务器网关接口（Asynchronous Server Gateway Interface）

> ASGI is a spiritual successor to WSGI, the long-standing Python standard for compatibility between web servers, frameworks, and applications.

实现asgi的app

```
# example.py
async def app(scope, receive, send):
    assert scope['type'] == 'http'

    await send({
        'type': 'http.response.start',
        'status': 200,
        'headers': [
            [b'content-type', b'text/plain'],
        ],
    })
    await send({
        'type': 'http.response.body',
        'body': b'Hello, world!',
    })
```

> - `scope` - A dictionary containing information about the incoming connection.
> - `receive` - A channel on which to receive incoming messages from the server.
> - `send` - A channel on which to send outgoing messages to the server.

运行asgi服务器：uvicorn（先下载 pip install uvicorn）

```
uvicorn example:app
```

也可以在应用中直接启动uvicorn

```
import uvicorn

async def app(scope, receive, send):
    ...

if __name__ == "__main__":
    uvicorn.run("example:app", host="127.0.0.1", port=5000, log_level="info")
```

- 基于类的app

```
class App:
    async def __call__(self, scope, receive, send):
        assert scope['type'] == 'http'
        ...

app = App()
```

常见的实现了asgi的应用：fastapi、starleete、django channels等等

2-9 uvloop

- 官网简介：https://uvloop.readthedocs.io/

uvloop是内置 asyncio 事件循环的快速、直接的替代方案。

uvloop和 asyncio 与 Python 3.5 中的 async/await 的强大功能相结合，使得在 Python 中编写高性能网络代码比以往任何时候都更加容易。

uvloop使异步速度更快。事实上，它至少比 nodejs、gevent 以及任何其他 Python 异步框架快 2 倍。基于 uvloop 的 asyncio 的性能接近 Go 程序。

- 为什么快：uvloop基于libuv，libuv是一个使用C语言实现的高性能异步I/O库，uvloop用来代替asyncio默认事件循环，可以进一步加快异步I/O操作的速度。
- 安装uvloop

```
pip install uvloop
```

- 基本使用

```
# 方式1
import asyncio
import uvloop
asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())

# 方式2 uvloop.install()


# 方式3
import asyncio
import uvloop
loop = uvloop.new_event_loop()
asyncio.set_event_loop(loop)
```

2-10 异步Web框架fastapi

- 官网：https://fastapi.tiangolo.com/
- 简介：*FastAPI framework, high performance, easy to learn, fast to code, ready for production*
- 特性：
  - **快速**：可与 **NodeJS** 和 **Go** 比肩的极高性能（归功于 Starlette 和 Pydantic）。最快的 Python web 框架之一。
  - **高效编码**：提高功能开发速度约 200% 至 300%。*
  - **更少 bug**：减少约 40% 的人为（开发者）导致错误。*
  - **智能**：极佳的编辑器支持。处处皆可自动补全，减少调试时间。typing hint
  - **简单**：设计的易于使用和学习，阅读文档的时间更短。
  - **简短**：使代码重复最小化。通过不同的参数声明实现丰富功能。bug 更少。
  - **健壮**：生产可用级别的代码。还有自动生成的交互式文档。
  - **标准化**：基于（并完全兼容）API 的相关开放标准：OpenAPI (以前被称为 Swagger) 和 JSON Schema。

- 下载：

```
pip install fastapi
pip install uvicorn
```

- 依赖：

```
- Python 3.6 及更高版本
- FastAPI 站在以下巨人的肩膀之上：
    - [Starlette](https://www.starlette.io/) 负责 web 部分。
    - [Pydantic](https://pydantic-docs.helpmanual.io/) 负责数据部分。
```

- 快速上手

```
# main.py

from typing import Optional

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
async def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
async def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

- 启动服务

```
uvicorn main:app --reload
```

2-11 异步Web框架starlette

fastapi内部依赖starlette实现web部分，所以想要玩转fastapi，最好对starlette有基本的了解。

官网：https://www.starlette.io/

- 示例

```
from starlette.applications import Starlette
from starlette.requests import Request
from starlette.responses import JSONResponse
from starlette.routing import Route


async def homepage(request: Request):
    print("url:", request.url)
    print("base_url:", request.base_url)
    return JSONResponse({'hello': 'world'})
```

```
app = Starlette(debug=True, routes=[
    Route('/', homepage),
])
```

特点：

- 每个模块都是独立的，容易理解，方便使用。
- Starlette被设计成既可以用做一个web框架，也可以当做一个工具箱，可以独立使用内部的模块。
- 开箱即用。

2-12 starlette之Application

Starlette 包含一个非常重要的类`Starlette`，它很好地将其所有其他组件联系在一起。

```
from starlette.applications import Starlette
from starlette.responses import PlainTextResponse
from starlette.routing import Route, Mount, WebSocketRoute
from starlette.staticfiles import StaticFiles


def homepage(request):
    return PlainTextResponse('Hello, world!')


def user_me(request):
    username = "John Doe"
    return PlainTextResponse('Hello, %s!' % username)


def user(request):
    username = request.path_params['username']
    return PlainTextResponse('Hello, %s!' % username)


async def websocket_endpoint(websocket):
    await websocket.accept()
    await websocket.send_text('Hello, websocket!')
    await websocket.close()


def startup():
    print('Ready to go')


def shutdown():
    print('The end')


routes = [
    Route('/', homepage),
```

```
    Route('/user/me', user_me),
    Route('/user/{username}', user),
    WebSocketRoute('/ws', websocket_endpoint),
    Mount('/static', StaticFiles(directory="static")),
]

app = Starlette(debug=True, routes=routes, on_startup=[startup], on_shutdown=
[shutdown])
```

- 常见应用程序，就是实例化一个Starlette对象

```
Starlette(
    debug=False,
    routes=None,
    middleware=None,
    exception_handlers=None,
    on_startup=None,
    on_shutdown=None,
    lifespan= None
)
```

- 参数解释

```
debug:                是否开启debug模式
routes:               为传入的 HTTP 和 WebSocket 请求提供服务的路由列表。
middleware:           中间件列表，默认包含两个中间件：ServerErrorMiddleware和
ExceptionMiddleware
exception_handlers:   一个字段：key是整数状态代码或异常类类型，value是可调用对象
on_startup:           在应用程序启动时运行的可调用对象列表。
on_shutdown:          在应用程序关闭时运行的可调用项列表。
```

2-13 starlette之Request

Starlette 包含一个`Request`类，有了这个它，但我们想要获取请求信息时，就不需要直接操作ASGI中的`scope`和`receive`

```
from starlette.requests import Request
from starlette.responses import Response


async def app(scope, receive, send):
    assert scope['type'] == 'http'
    request = Request(scope, receive)
    content = '%s %s' % (request.method, request.url.path)
    response = Response(content, media_type='text/plain')
    await response(scope, receive, send)
```

- 如果不需要请求体，实例化Request时不需要传 `receive`

- 常用属性和方法

```
- request.method
- request.url
    request.url.path, request.url.port, request.url.scheme

- request.headers                    # 一个字典对象
- request.query_params['search']     # 一个字典对象
- request.path_params                # 一个字典对象

- request.client
    request.client.host, request.client.port

- request.cookies.get('mycookie')       # cookie
```

- 请求体的方法

```
await request.body()    # bytes

await request.form()    # form data or multipart

await request.json()    # json

async for chunk in request.stream():
    pass # 流式获取请求体
```

- 请求文件：starlette.datastructures.UploadFile

```
from starlette.datastructures import UploadFile

form: UploadFile = await request.form()
filename = form["upload_file"].filename
contents = await form["upload_file"].read()
```

2-14 starlette之Response

Starlette 包含了一些响应类，这些响应类可以通过 `send` 通道返回合适的ASGI消息。Response类的签名如下：

```
Response(content, status_code=200, headers=None, media_type=None)

- content:          # 响应内容：字符串或者字节
- status_code:      # 状态码
- headers:          # 响应头
- media_type:       # eg. "text/html"
```

Starlette 将自动包含一个 Content-Length 标头。它还将包含一个 Content-Type 标头，基于 media_type 并为文本类型附加一个字符集。

- 实例化之后就可以使用

```
from starlette.responses import Response


async def app(scope, receive, send):
    assert scope['type'] == 'http'
    response = Response('Hello, world!', media_type='text/plain')
    await response(scope, receive, send)
```

常见的响应类：

- HTMLResponse

```
响应HTML页面，HTMLResponse是Response的子类，media_type = "text/html"
```

- PlainTextResponse

```
响应纯文本，PlainTextResponse是Response的子类，media_type = "text/plain"
```

- JSONResponse

```
响应json格式的数据，JSONResponse是Response的子类，media_type = "application/json"

# 自定义json序列化方式
from typing import Any

import orjson
from starlette.responses import JSONResponse


class OrjsonResponse(JSONResponse):
    def render(self, content: Any) -> bytes:
        return orjson.dumps(content)
```

- RedirectResponse

```
重定向，默认使用307状态码
```

- StreamingResponse

```
# 流式响应数据，数据时异步生成器或者普通生成器

from starlette.responses import StreamingResponse
import asyncio


async def slow_numbers(minimum, maximum):
    yield '<html><body><ul>'
    for number in range(minimum, maximum + 1):
        yield '<li>%d</li>' % number
        await asyncio.sleep(0.5)
    yield '</ul></body></html>'


async def app(scope, receive, send):
    assert scope['type'] == 'http'
    generator = slow_numbers(1, 10)
    response = StreamingResponse(generator, media_type='text/html')
    await response(scope, receive, send)
```

- FileResponse

```
# 异步流式响应文件

from starlette.responses import FileResponse


async def app(scope, receive, send):
    assert scope['type'] == 'http'
    response = FileResponse('main.py')
    await response(scope, receive, send)
```

2-15 starlette之Routing

路由，Starlette 有一个简单但功能强大的请求路由系统。该路由系统就是一个列表。

```
from starlette.applications import Starlette
from starlette.responses import PlainTextResponse
from starlette.routing import Route


def homepage(request):
    return PlainTextResponse("Homepage")

async def about(request):
    username = request.path_params['username']
    return PlainTextResponse(username)
```

```
routes = [
    Route("/", endpoint=homepage),
    Route("/about/{username:str}", endpoint=about),
]

app = Starlette(routes=routes)
```

- 注意：endpoint可以是基于函数的（接收一个Request对象），也可以是基于类的（实现了ASGI接口）

- 路径参数和转换器

```
Route('/users/{user_id:int}', user)
Route('/floating-point/{number:float}', floating_point)
Route('/uploaded/{rest_of_path:path}', uploaded)

# 注意：转换器中，冒号后面不能有空格
```

- 处理请求方式

```
Route('/users/{user_id:int}', user, methods=["GET", "POST"])
# 默认基于函数的Route只接收GET
```

- 子路由

```
routes = [
    Route('/', homepage),
    Mount('/users', routes=[
        Route('/', homepage),
        Route('/about/{username}', about),
    ])
]
```

- 路由优先级

当有请求进来时，会按照routes的顺序以此匹配，只要匹配到就立即执行匹配到的endpoint

在多个路由可以匹配传入路径的情况下，应该确保把具体路由放在前面，抽象路由放在后面。

```
# Don't do this: `/users/me` will never match incoming requests.
routes = [
    Route('/users/{username}', user),
    Route('/users/me', current_user),
]

# Do this: `/users/me` is tested first.
routes = [
    Route('/users/me', current_user),
    Route('/users/{username}', user),
]
```

2-16 starlette之Endpoint

Endpoint：Starlette 提供了基于类的视图用来处理HTTP请求，根据请求方式的不同来分配处理逻辑，这个东西就是Endpoint。Starlette中有两个Endpoint：`HTTPEndpoint`、`WebSocketEndpoint`

```
from starlette.applications import Starlette
from starlette.responses import PlainTextResponse
from starlette.endpoints import HTTPEndpoint
from starlette.routing import Route


class Homepage(HTTPEndpoint):
    async def get(self, request):
        return PlainTextResponse(request.base_url)


class User(HTTPEndpoint):
    async def get(self, request):
        username = request.path_params['username']
        return PlainTextResponse(f"GET: {username}")

    async def post(self, request):
        username = request.path_params['username']
        return PlainTextResponse(f"POST: {username}")


routes = [
    Route("/", Homepage),
    Route("/{username}", User)
]


app = Starlette(routes=routes)
```

- 注意：Route中路径后面传的参数是Endpoint类本身
- 如果没有匹配到会返回：405 Method not allowed

- 源码解读

```
class HTTPEndpoint:
    def __init__(self, scope: Scope, receive: Receive, send: Send) -> None:
        assert scope["type"] == "http"
        self.scope = scope
        self.receive = receive
        self.send = send

    def __await__(self) -> typing.Generator:
        return self.dispatch().__await__()

    async def dispatch(self) -> None:
        request = Request(self.scope, receive=self.receive)
        handler_name = "get" if request.method == "HEAD" else
request.method.lower()
        handler = getattr(self, handler_name, self.method_not_allowed)
        is_async = asyncio.iscoroutinefunction(handler)
        if is_async:
            response = await handler(request)
        else:
            response = await run_in_threadpool(handler, request)
        await response(self.scope, self.receive, self.send)
```

2-17 starlette之Background Task

后台任务，可以给响应上绑定一个任务，在响应被发送出去时把绑定的任务执行一次。比如使用在用户注册后发送邮件信息。

Starlette 中提供了`BackgroundTask`这个类实现后台任务。

- 基本使用

```
from starlette.applications import Starlette
from starlette.responses import JSONResponse
from starlette.routing import Route
from starlette.background import BackgroundTask


async def signup(request):
    data = await request.json()
    username, email = data['username'], data['email']
    task = BackgroundTask(send_welcome_email, to_address=email)
    message = {'status': 'Signup successful'}
    return JSONResponse(message, background=task)


async def send_welcome_email(to_address):
    print("模拟发邮件")


routes = [
    Route('/user/signup', endpoint=signup, methods=['POST'])
```

```
]

app = Starlette(routes=routes)
```

- 源码阅读

```
class Response:
    media_type = None
    charset = "utf-8"

    def __init__(
        self,
        content: typing.Any = None,
        status_code: int = 200,
        headers: dict = None,
        media_type: str = None,
        background: BackgroundTask = None,
    ) -> None:
        self.status_code = status_code
        if media_type is not None:
            self.media_type = media_type
        self.background = background
        self.body = self.render(content)
        self.init_headers(headers)

    async def __call__(self, scope: Scope, receive: Receive, send: Send) ->
None:
        await send(
            {
                "type": "http.response.start",
                "status": self.status_code,
                "headers": self.raw_headers,
            }
        )
        await send({"type": "http.response.body", "body": self.body})

        if self.background is not None:
            await self.background()          # 如果有后台任务则，执行backgroud实例


 class BackgroundTask:
    def __init__(
        self, func: typing.Callable, *args: typing.Any, **kwargs: typing.Any
    ) -> None:
        self.func = func
        self.args = args
        self.kwargs = kwargs
        self.is_async = asyncio.iscoroutinefunction(func)

    # 当任务实例被执行时，出发__call__执行。由此可知任务函数可以是普通函数，也可以是协程函数
    async def __call__(self) -> None:
        if self.is_async:
            await self.func(*self.args, **self.kwargs)
        else:
            await run_in_threadpool(self.func, *self.args, **self.kwargs)
```

2-18 课程总结

- 详见思维导图