不知道你是否发现，身边聊异步的人越来越多了，比如：FastAPI、Tornado、Sanic、Django 3、aiohttp等。

听说异步如何如何牛逼？性能如何吊炸天。。。。但他到底是咋回事呢？

本节要跟大家一起聊聊关于asyncio异步的那些事！

asyncio讲解教程：https://study.163.com/instructor/3525856.htm (https://study.163.com/instructor/3525856.htm)

博客园同步：https://www.cnblogs.com/wupeiqi/ (https://www.cnblogs.com/wupeiqi/)

1.协程

想学asyncio，得先了解协程，协程是根本呀！

协程（Coroutine），也可以被称为微线程，是一种用户态内的上下文切换技术。简而言之，其实就是通过一个线程实现代码块相互切换执行。例如：

```
def func1():
    print(1)
    ...
    print(2)

def func2():
    print(3)
    ...
    print(4)

func1()
func2()
```

上述代码是普通的函数定义和执行，按流程分别执行两个函数中的代码，并先后会输出：`1、2、3、4`。但如果介入协程技术那么就可以实现函数见代码切换执行，最终输入：`1、3、2、4`。

在Python中有多种方式可以实现协程，例如：

- greenlet，是一个第三方模块，用于实现协程代码（Gevent协程就是基于greenlet实现）
- yield，生成器，借助生成器的特点也可以实现协程代码。
- asyncio，在Python3.4中引入的模块用于编写协程代码。
- async & awiat，在Python3.5中引入的两个关键字，结合asyncio模块可以更方便的编写协程代码。

1.1 greenlet

greentlet是一个第三方模块，需要提前安装 `pip3 install greenlet` 才能使用。

```python
from greenlet import greenlet


def func1():
    print(1)        # 第1步：输出 1
    gr2.switch()    # 第3步：切换到 func2 函数
    print(2)        # 第6步：输出 2
    gr2.switch()    # 第7步：切换到 func2 函数，从上一次执行的位置继续向后执行


def func2():
    print(3)        # 第4步：输出 3
    gr1.switch()    # 第5步：切换到 func1 函数，从上一次执行的位置继续向后执行
    print(4)        # 第8步：输出 4


gr1 = greenlet(func1)
gr2 = greenlet(func2)
gr1.switch() # 第1步：去执行 func1 函数
```

注意：switch中也可以传递参数用于在切换执行时相互传递值。

1.2 yield

基于Python的生成器的yield和yield form关键字实现协程代码。

```
def func1():
    yield 1
    yield from func2()
    yield 2


def func2():
    yield 3
    yield 4


f1 = func1()
for item in f1:
    print(item)
```

注意：yield form关键字是在Python3.3中引入的。

1.3 asyncio

在Python3.4之前官方未提供协程的类库，一般大家都是使用greenlet等其他来实现。在Python3.4发布后官方正式支持协程，即：asyncio模块。

```python
import asyncio

@asyncio.coroutine
def func1():
    print(1)
    yield from asyncio.sleep(2)  # 遇到IO耗时操作，自动化切换到tasks中的其他任务
    print(2)


@asyncio.coroutine
def func2():
    print(3)
    yield from asyncio.sleep(2) # 遇到IO耗时操作，自动化切换到tasks中的其他任务
    print(4)


tasks = [
    asyncio.ensure_future( func1() ),
    asyncio.ensure_future( func2() )
]

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(tasks))
```

注意：基于asyncio模块实现的协程比之前的要更厉害，因为他的内部还集成了遇到IO耗时操作自动切花的功能。

1.4 async & awit

async & await 关键字在Python3.5版本中正式引入，基于他编写的协程代码其实就是 上一示例 的加强版，让代码可以更加简便。

Python3.8之后 `@asyncio (https://github.com/asyncio).coroutine` 装饰器就会被移除，推荐使用async & await 关键字实现协程代码。

```
import asyncio


async def func1():
    print(1)
    await asyncio.sleep(2)
    print(2)


async def func2():
    print(3)
    await asyncio.sleep(2)
    print(4)


tasks = [
    asyncio.ensure_future(func1()),
    asyncio.ensure_future(func2())
]

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(tasks))
```

1.5 小结

关于协程有多种实现方式，目前主流使用是Python官方推荐的asyncio模块和async&await关键字的方式，例如：在tonado、sanic、fastapi、django3 中均已支持。

接下来，我们也会针对 `asyncio模块` + `async & await` 关键字进行更加详细的讲解。

2.协程的意义

通过学习，我们已经了解到协程可以通过一个线程在多个上下文中进行来回切换执行。

但是，协程来回切换执行的意义何在呢？（网上看到很多文章舔协程，协程牛逼之处是哪里呢？）

1. 计算型的操作，利用协程来回切换执行，没有任何意义，来回切换并保存状态 反倒会降低性能。
2. IO型的操作，利用协程在IO等待时间就去切换执行其他任务，当IO操作结束后再自动回调，那么就会大大节省资源并提供性能，从而实现异步编程（不等待任务结束就可以去执行其他代码）。

2.1 爬虫案例

例如：用代码实现下载 `url_list` 中的图片。

- 方式一：同步编程实现

```
"""
下载图片使用第三方模块requests，请提前安装：pip3 install requests
"""
import requests

def download_image(url):
    print("开始下载:",url)
    # 发送网络请求，下载图片
    response = requests.get(url)
    print("下载完成")
    # 图片保存到本地文件
    file_name = url.rsplit('_')[-1]
    with open(file_name, mode='wb') as file_object:
        file_object.write(response.content)
if __name__ == '__main__':
    url_list = [
        'https://www3.autoimg.cn/newsdfs/g26/M02/35/A9/120x90_0_autohomecar__ChsEe12AXQ6AOOH_AAFocMs8nzU621.jpg',
        'https://www2.autoimg.cn/newsdfs/g30/M01/3C/E2/120x90_0_autohomecar__ChcCSV2BBICAUntfAADjJFd6800429.jpg',
        'https://www3.autoimg.cn/newsdfs/g26/M0B/3C/65/120x90_0_autohomecar__ChcCP12BFCmAIO83AAGq7vK0sGY193.jpg'
    ]
    for item in url_list:
        download_image(item)
```

- 方式二：基于协程的异步编程实现

```
"""
下载图片使用第三方模块aiohttp，请提前安装：pip3 install aiohttp
"""
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import aiohttp
import asyncio
async def fetch(session, url):
    print("发送请求：", url)
    async with session.get(url, verify_ssl=False) as response:
        content = await response.content.read()
        file_name = url.rsplit('_')[-1]
        with open(file_name, mode='wb') as file_object:
            file_object.write(content)
async def main():
    async with aiohttp.ClientSession() as session:
        url_list = [
            'https://www3.autoimg.cn/newsdfs/g26/M02/35/A9/120x90_0_autohomecar__ChsEe12AXQ6AOOH_AAFocMs8nzU621.jpg',
            'https://www2.autoimg.cn/newsdfs/g30/M01/3C/E2/120x90_0_autohomecar__ChcCSV2BBICAUntfAADjJFd6800429.jpg',
            'https://www3.autoimg.cn/newsdfs/g26/M0B/3C/65/120x90_0_autohomecar__ChcCP12BFCmAIO83AAGq7vK0sGY193.jpg'
        ]
        tasks = [asyncio.create_task(fetch(session, url)) for url in url_list]
        await asyncio.wait(tasks)

if __name__ == '__main__':
```

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

上述两种的执行对比之后会发现，`基于协程的异步编程` 要比 `同步编程` 的效率高了很多。因为：

- 同步编程，按照顺序逐一排队执行，如果图片下载时间为2分钟，那么全部执行完则需要6分钟。
- 异步编程，几乎同时发出了3个下载任务的请求（遇到IO请求自动切换去发送其他任务请求），如果图片下载时间为2分钟，那么全部执行完毕也大概需要2分钟左右就可以了。

2.2 小结

协程一般应用在有IO操作的程序中，因为协程可以利用IO等待的时间去执行一些其他的代码，从而提升代码执行效率。

生活中不也是这样的么，假设 你是一家制造汽车的老板，员工点击设备的【开始】按钮之后，在设备前需等待30分钟，然后点击【结束】按钮，此时作为老板的你一定希望这个员工在等待的那30分钟的时间去做点其他的工作。

3.异步编程

基于 `async` & `await` 关键字的协程可以实现异步编程，这也是目前python异步相关的主流技术。

想要真正的了解Python中内置的异步编程，根据下文的顺序一点点来看。

3.1 事件循环

事件循环，可以把他当做是一个while循环，这个while循环在周期性的运行并执行一些 `任务` ，在特定条件下终止循环。

```
# 伪代码

任务列表 = [ 任务1, 任务2, 任务3,... ]

while True:
    可执行的任务列表，已完成的任务列表 = 去任务列表中检查所有的任务，将'可执行'和'已完成'的任务返回

    for 就绪任务 in 已准备就绪的任务列表:
        执行已就绪的任务

    for 已完成的任务 in 已完成的任务列表:
        在任务列表中移除 已完成的任务

    如果 任务列表 中的任务都已完成，则终止循环
```

在编写程序时候可以通过如下代码来获取和创建事件循环。

```
import asyncio

loop = asyncio.get_event_loop()
```

3.2 协程和异步编程

协程函数，定义形式为 `async def` (https://docs.python.org/zh-cn/3.8/reference/compound_stmts.html#async-def) 的函数。

协程对象，调用 *协程函数* 所返回的对象。

```python
# 定义一个协程函数
async def func():
    pass

# 调用协程函数，返回一个协程对象
result = func()
```

注意：调用协程函数时，函数内部代码不会执行，只是会返回一个协程对象。

3.2.1 基本应用

程序中，如果想要执行协程函数的内部代码，需要 事件循环 和 协程对象 配合才能实现，如：

```python
import asyncio


async def func():
    print("协程内部代码")

# 调用协程函数，返回一个协程对象。
result = func()

# 方式一
# loop = asyncio.get_event_loop() # 创建一个事件循环
# loop.run_until_complete(result) # 将协程当做任务提交到事件循环的任务列表中，协程执行完成之后终止。

# 方式二
# 本质上方式一是一样的，内部先 创建事件循环 然后执行 run_until_complete，一个简便的写法。
# asyncio.run 函数在 Python 3.7 中加入 asyncio 模块
```

```
16. # asyncio.run 函数在 Python 3.7 中加入 asyncio 模块；
17. asyncio.run(result)
```

这个过程可以简单理解为：将 协程 当做任务添加到 事件循环 的任务列表，然后事件循环检测列表中的 协程 是否 已准备就绪（默认可理解为就绪状态），如果准备就绪则执行其内部代码。

3.2.2 await

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 当前协程（任务），当前协程（任务）挂起过程中 事件循环可以去执行其他的协程（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。代码如下：

示例1：

```
1. import asyncio
2.
3.
4. async def func():
5.     print("执行协程函数内部代码")
6.
7.     # 遇到IO操作挂起当前协程（任务），等IO操作完成之后再继续往下执行。
8.     # 当前协程挂起时，事件循环可以去执行其他协程（任务）。
9.     response = await asyncio.sleep(2)
10.
11.    print("IO请求结束，结果为：", response)
12.
13. result = func()
14.
15. asyncio.run(result)
```

示例2：

```python
import asyncio


async def others():
    print("start")
    await asyncio.sleep(2)
    print('end')
    return '返回值'


async def func():
    print("执行协程函数内部代码")

    # 遇到IO操作挂起当前协程（任务），等IO操作完成之后再继续往下执行。当前协程挂起时，事件循环可以去执行其他协程（任务）。
    response = await others()

    print("IO请求结束，结果为：", response)

asyncio.run( func() )
```

示例3：

```python
import asyncio


async def others():
    print("start")
    await asyncio.sleep(2)
    print('end')
    return '返回值'


async def func():
    print("执行协程函数内部代码")

    # 遇到IO操作挂起当前协程（任务），等IO操作完成之后再继续往下执行。当前协程挂起时，事件循环可以去执行其他协程（任务）。
    response1 = await others()
    print("IO请求结束，结果为：", response1)

    response2 = await others()
    print("IO请求结束，结果为：", response2)

asyncio.run( func() )
```

上述的所有示例都只是创建了一个任务，即：事件循环的任务列表中只有一个任务，所以在IO等待时无法演示切换到其他任务效果。

在程序想要创建多个任务对象，需要使用Task对象来实现。

3.2.3 Task对象

> *Tasks are used to schedule coroutines concurrently.*
>
> *When a coroutine is wrapped into a Task with functions like* `asyncio.create_task()` *(https://docs.python.org/3.8/library/asyncio-task.html#asyncio.create_task) the coroutine is automatically scheduled to run soon。*

Tasks用于并发调度协程，通过 `asyncio.create_task(协程对象)` 的方式创建Task对象，这样可以让协程加入事件循环中等待被调度执行。除了使用 `asyncio.create_task()` 函数以外，还可以用低层级的 `loop.create_task()` 或 `ensure_future()` 函数。不建议手动实例化 Task 对象。

本质上是将协程对象封装成task对象，并将协程立即加入事件循环，同时追踪协程的状态。

注意： `asyncio.create_task()` 函数在 Python 3.7 中被加入。在 Python 3.7 之前，可以改用低层级的 `asyncio.ensure_future()` 函数。

示例1：

```python
import asyncio


async def func():
    print(1)
    await asyncio.sleep(2)
    print(2)
    return "返回值"


async def main():
    print("main开始")

    # 创建协程，将协程封装到一个Task对象中并立即添加到事件循环的任务列表中，等待事件循环去执行（默认是就绪状态）。
    task1 = asyncio.create_task(func())

    # 创建协程，将协程封装到一个Task对象中并立即添加到事件循环的任务列表中，等待事件循环去执行（默认是就绪状态）。
    task2 = asyncio.create_task(func())

    print("main结束")

    # 当执行某协程遇到IO操作时，会自动化切换执行其他任务。
    # 此处的await是等待相对应的协程全都执行完毕并获取结果
    ret1 = await task1
    ret2 = await task2
    print(ret1, ret2)


```

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

示例2：

```python
import asyncio


async def func():
    print(1)
    await asyncio.sleep(2)
    print(2)
    return "返回值"


async def main():
    print("main开始")

    # 创建协程，将协程封装到Task对象中并添加到事件循环的任务列表中，等待事件循环去执行（默认是就绪状态）。
    # 在调用
    task_list = [
        asyncio.create_task(func(), name="n1"),
        asyncio.create_task(func(), name="n2")
    ]

    print("main结束")

    # 当执行某协程遇到IO操作时，会自动化切换执行其他任务。
    # 此处的await是等待所有协程执行完毕，并将所有协程的返回值保存到done
    # 如果设置了timeout值，则意味着此处最多等待的秒，完成的协程返回值写入到done中，未完成则写到pending中。
    done, pending = await asyncio.wait(task_list, timeout=None)
    print(done, pending)

```

```
29.
30. asyncio.run(main())
```

注意：`asyncio.wait` 源码内部会对列表中的每个协程执行ensure_future从而封装为Task对象，所以在和wait配合使用时task_list的值为 `[func(),func()]` 也是可以的。

示例3：

```
 1. import asyncio
 2.
 3.
 4. async def func():
 5.     print("执行协程函数内部代码")
 6.
 7.     # 遇到IO操作挂起当前协程（任务），等IO操作完成之后再继续往下执行。当前协程挂起时，事件循环可以去执行其他协程（任务）。
 8.     response = await asyncio.sleep(2)
 9.
10.     print("IO请求结束，结果为：", response)
11.
12.
13. coroutine_list = [func(), func()]
14.
15. # 错误：coroutine_list = [ asyncio.create_task(func()), asyncio.create_task(func()) ]
16. # 此处不能直接 asyncio.create_task，因为将Task立即加入到事件循环的任务列表，
17. # 但此时事件循环还未创建，所以会报错。
18.
19.
20. # 使用asyncio.wait将列表封装为一个协程，并调用asyncio.run实现执行两个协程
21. # asyncio.wait内部会对列表中的每个协程执行ensure_future，封装为Task对象。
22. done,pending = asyncio.run( asyncio.wait(coroutine_list) )
```

3.2.4 asyncio.Future对象

> *A* `Future` *is a special* ***low-level*** *awaitable object that represents an* ***eventual result*** *of an asynchronous operation.*

asyncio中的Future对象是一个相对更偏向底层的可对象，通常我们不会直接用到这个对象，而是直接使用Task对象来完成任务的并和状态的追踪。（ Task 是 Futrue的子类 ）

Future为我们提供了异步编程中的 最终结果 的处理（Task类也具备状态处理的功能）。

示例1:

```
async def main():
    # 获取当前事件循环
    loop = asyncio.get_running_loop()

    # # 创建一个任务（Future对象），这个任务什么都不干。
    fut = loop.create_future()

    # 等待任务最终结果（Future对象），没有结果则会一直等下去。
    await fut

asyncio.run(main())
```

示例2:

```python
import asyncio


async def set_after(fut):
    await asyncio.sleep(2)
    fut.set_result("666")


async def main():
    # 获取当前事件循环
    loop = asyncio.get_running_loop()

    # 创建一个任务（Future对象），没绑定任何行为，则这个任务永远不知道什么时候结束。
    fut = loop.create_future()

    # 创建一个任务（Task对象），绑定了set_after函数，函数内部在2s之后，会给fut赋值。
    # 即手动设置future任务的最终结果，那么fut就可以结束了。
    await loop.create_task(set_after(fut))

    # 等待 Future对象获取 最终结果，否则一直等下去
    data = await fut
    print(data)

asyncio.run(main())
```

Future对象本身函数进行绑定，所以想要让事件循环获取Future的结果，则需要手动设置。而Task对象继承了Future对象，其实就对Future进行扩展，他可以实现在对应绑定的函数执行完成之后，自动执行 `set_result` ，从而实现自动结束。

虽然，平时使用的是Task对象，但对于结果的处理本质是基于Future对象来实现的。

扩展：支持 `await 对象` 语 法的对象课成为可等待对象，所以 `协程对象` 、 `Task对象` 、 `Future对象` 都可以被成为可等待对象。

3.2.5 futures.Future对象

在Python的 `concurrent.futures` 模块中也有一个Future对象，这个对象是基于线程池和进程池实现异步操作时使用的对象。

```
import time
from concurrent.futures import Future
from concurrent.futures.thread import ThreadPoolExecutor
from concurrent.futures.process import ProcessPoolExecutor


def func(value):
    time.sleep(1)
    print(value)


pool = ThreadPoolExecutor(max_workers=5)
# 或 pool = ProcessPoolExecutor(max_workers=5)


for i in range(10):
    fut = pool.submit(func, i)
    print(fut)
```

两个Future对象是不同的，他们是为不同的应用场景而设计，例如： `concurrent.futures.Future` 不支持await语法 等。

官方提示两对象之间不同：

- unlike asyncio Futures, `concurrent.futures.Future` (https://docs.python.org/3.8/library/concurrent.futures.html#concurrent.futures.Future) instances cannot be awaited.
- `asyncio.Future.result()` (https://docs.python.org/3.8/library/asyncio-future.html#asyncio.Future.result) and `asyncio.Future.exception()` (https://docs.python.org/3.8/library/asyncio-future.html#asyncio.Future.exception) do not accept the *timeout* argument.
- `asyncio.Future.result()` (https://docs.python.org/3.8/library/asyncio-future.html#asyncio.Future.result) and `asyncio.Future.exception()` (https://docs.python.org/3.8/library/asyncio-future.html#asyncio.Future.exception) raise an `InvalidStateError` (https://docs.python.org/3.8/library/asyncio-exceptions.html#asyncio.InvalidStateError) exception when the Future is not *done*.
- Callbacks registered with `asyncio.Future.add_done_callback()` (https://docs.python.org/3.8/library/asyncio-future.html#asyncio.Future.add_done_callback) are not called immediately. They are scheduled with `loop.call_soon()` (https://docs.python.org/3.8/library/asyncio-eventloop.html#asyncio.loop.call_soon) instead.
- asyncio Future is not compatible with the `concurrent.futures.wait()` (https://docs.python.org/3.8/library/concurrent.futures.html#concurrent.futures.wait) and `concurrent.futures.as_completed()` (https://docs.python.org/3.8/library/concurrent.futures.html#concurrent.futures.as_completed) functions.

在Python提供了一个将 `futures.Future` 对象包装成 `asyncio.Future` 对象的函数 `asynic.wrap_future` 。

接下里你肯定问：为什么python会提供这种功能?

其实，一般在程序开发中我们要么统一使用 asycio 的协程实现异步操作、要么都使用进程池和线程池实现异步操作。但如果 `协程的异步` 和 `进程池/线程池的异步` 混搭时，那么就会用到此功能了。

```python
import time
import asyncio
import concurrent.futures

def func1():
    # 某个耗时操作
    time.sleep(2)
    return "SB"

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

    # 1. Run in the default loop's executor ( 默认ThreadPoolExecutor )
    # 第一步：内部会先调用 ThreadPoolExecutor 的 submit 方法去线程池中申请一个线程去执行func1函数，并返回一个concurrent.futures.Future对象
    # 第二步：调用asyncio.wrap_future将concurrent.futures.Future对象包装为asycio.Future对象。
    # 因为concurrent.futures.Future对象不支持await语法，所以需要包装为 asycio.Future对象 才能使用。
    fut = loop.run_in_executor(None, func1)
    result = await fut
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    # with concurrent.futures.ThreadPoolExecutor() as pool:
    #     result = await loop.run_in_executor(
    #         pool, func1)
    #     print('custom thread pool', result)

    # 3. Run in a custom process pool:
```

```
    # with concurrent.futures.ProcessPoolExecutor() as pool:
    #     result = await loop.run_in_executor(
    #         pool, func1)
    #     print('custom process pool', result)

asyncio.run(main())
```

应用场景：当项目以协程式的异步编程开发时，如果要使用一个第三方模块，而第三方模块不支持协程方式异步编程时，就需要用到这个功能，例如：

```python
import asyncio
import requests


async def download_image(url):
    # 发送网络请求，下载图片（遇到网络下载图片的IO请求，自动化切换到其他任务）
    print("开始下载:", url)

    loop = asyncio.get_event_loop()
    # requests模块默认不支持异步操作，所以就使用线程池来配合实现了。
    future = loop.run_in_executor(None, requests.get, url)

    response = await future
    print('下载完成')
    # 图片保存到本地文件
    file_name = url.rsplit('_')[-1]
    with open(file_name, mode='wb') as file_object:
        file_object.write(response.content)


if __name__ == '__main__':
    url_list = [
        'https://www3.autoimg.cn/newsdfs/g26/M02/35/A9/120x90_0_autohomecar__ChsEe12AXQ6AOOH_AAFocMs8nzU621.jpg',
        'https://www2.autoimg.cn/newsdfs/g30/M01/3C/E2/120x90_0_autohomecar__ChcCSV2BBICAUntfAADjJFd6800429.jpg',
        'https://www3.autoimg.cn/newsdfs/g26/M0B/3C/65/120x90_0_autohomecar__ChcCP12BFCmAIO83AAGq7vK0sGY193.jpg'
```

```
    ]

    tasks = [download_image(url) for url in url_list]

    loop = asyncio.get_event_loop()
    loop.run_until_complete( asyncio.wait(tasks) )
```

3.2.6 异步迭代器

什么是异步迭代器

实现了 `__aiter__()` (https://docs.python.org/zh-cn/3.8/reference/datamodel.html#object.__aiter__) 和 `__anext__()` (https://docs.python.org/zh-cn/3.8/reference/datamodel.html#object.__anext__) 方法的对象。 `__anext__` 必须返回一个 awaitable (https://docs.python.org/zh-cn/3.8/glossary.html#term-awaitable) 对象。 `async for` (https://docs.python.org/zh-cn/3.8/reference/compound_stmts.html#async-for) 会处理异步迭代器的 `__anext__()` (https://docs.python.org/zh-cn/3.8/reference/datamodel.html#object.__anext__) 方法所返回的可等待对象，直到其引发一个 `StopAsyncIteration` (https://docs.python.org/zh-cn/3.8/library/exceptions.html#StopAsyncIteration) 异常。由 **PEP 492** (https://www.python.org/dev/peps/pep-0492) 引入。

什么是异步可迭代对象?

可在 `async for` (https://docs.python.org/zh-cn/3.8/reference/compound_stmts.html#async-for) 语句中被使用的对象。必须通过它的 `__aiter__()` (https://docs.python.org/zh-cn/3.8/reference/datamodel.html#object.__aiter__) 方法返回一个 asynchronous iterator (https://docs.python.org/zh-cn/3.8/glossary.html#term-asynchronous-iterator)。由 **PEP 492** (https://www.python.org/dev/peps/pep-0492) 引入。

```python
import asyncio


class Reader(object):
    """ 自定义异步迭代器（同时也是异步可迭代对象） """

    def __init__(self):
        self.count = 0

    async def readline(self):
        # await asyncio.sleep(1)
        self.count += 1
        if self.count == 100:
            return None
        return self.count

    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == None:
            raise StopAsyncIteration
        return val


async def func():
    # 创建异步可迭代对象
```

```
    async_iter = Reader()
    # async for 必须要放在async def函数内，否则语法错误。
    async for item in async_iter:
        print(item)

asyncio.run(func())
```

异步迭代器其实没什么太大的作用，只是支持了async for语法而已。

3.2.6 异步上下文管理器

此种对象通过定义 `__aenter__()` (https://docs.python.org/zh-cn/3.8/reference/datamodel.html#object.__aenter__) 和 `__aexit__()` (https://docs.python.org/zh-cn/3.8/reference/datamodel.html#object.__aexit__) 方法来对 `async with` (https://docs.python.org/zh-cn/3.8/reference/compound_stmts.html#async-with) 语句中的环境进行控制。由 **PEP 492** (https://www.python.org/dev/peps/pep-0492) 引入。

```python
import asyncio


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

    async def do_something(self):
        # 异步操作数据库
        return 666

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

    async def __aexit__(self, exc_type, exc, tb):
        # 异步关闭数据库链接
        await asyncio.sleep(1)


async def func():
    async with AsyncContextManager() as f:
        result = await f.do_something()
        print(result)


asyncio.run(func())
```

这个异步的上下文管理器还是比较有用的，平时在开发过程中 打开、处理、关闭 操作时，就可以用这种方式来处理。

3.3 小结

在程序中只要看到 `async` 和 `await` 关键字，其内部就是基于协程实现的异步编程，这种异步编程是通过一个线程在IO等待时间去执行其他任务，从而实现并发。

以上就是异步编程的常见操作，内容参考官方文档。

- 中文版：https://docs.python.org/zh-cn/3.8/library/asyncio.html (https://docs.python.org/zh-cn/3.8/library/asyncio.html)
- 英文本：https://docs.python.org/3.8/library/asyncio.html (https://docs.python.org/3.8/library/asyncio.html)

4. uvloop

Python标准库中提供了 `asyncio` 模块，用于支持基于协程的异步编程。

uvloop是 asyncio 中的事件循环的替代方案，替换后可以使得asyncio性能提高。事实上，uvloop要比nodejs、gevent等其他python异步框架至少要快2倍，性能可以比肩Go语言。

安装uvloop

```
pip3 install uvloop
```

在项目中想要使用uvloop替换asyncio的事件循环也非常简单，只要在代码中这么做就行。

```
import asyncio
import uvloop
asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())

# 编写asyncio的代码，与之前写的代码一致。

# 内部的事件循环自动化会变为uvloop
```

```
7. # 内部的事件循环自动化会变为uvloop
8. asyncio.run(...)
```

注意：知名的asgi uvicorn内部就是使用的uvloop的事件循环。

5.实战案例

为了更好理解，上述所有示例的IO情况都是以 `asyncio.sleep` 为例，而真实的项目开发中会用到很多IO的情况。

5.1 异步Redis

当通过python去操作redis时，链接、设置值、获取值 这些都涉及网络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。

安装Python异步操作redis模块

```
1. pip3 install aioredis
```

示例1：异步操作redis。

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import asyncio
import aioredis


async def execute(address, password):
    print("开始执行", address)
    # 网络IO操作：创建redis连接
    redis = await aioredis.create_redis(address, password=password)

    # 网络IO操作：在redis中设置哈希值car，内部在设三个键值对，即： redis = { car:{key1:1,key2:2,key3:3}}
    await redis.hmset_dict('car', key1=1, key2=2, key3=3)

    # 网络IO操作：去redis中获取值
    result = await redis.hgetall('car', encoding='utf-8')
    print(result)

    redis.close()
    # 网络IO操作：关闭redis连接
    await redis.wait_closed()

    print("结束", address)


asyncio.run(execute('redis://47.93.4.198:6379', "root!2345"))
```

示例2：连接多个redis做操作（遇到IO会切换其他任务，提供了性能）。

```python
import asyncio
import aioredis


async def execute(address, password):
    print("开始执行", address)

    # 网络IO操作：先去连接 47.93.4.197:6379，遇到IO则自动切换任务，去连接47.93.4.198:6379
    redis = await aioredis.create_redis_pool(address, password=password)

    # 网络IO操作：遇到IO会自动切换任务
    await redis.hmset_dict('car', key1=1, key2=2, key3=3)

    # 网络IO操作：遇到IO会自动切换任务
    result = await redis.hgetall('car', encoding='utf-8')
    print(result)

    redis.close()
    # 网络IO操作：遇到IO会自动切换任务
    await redis.wait_closed()

    print("结束", address)


task_list = [
    execute('redis://47.93.4.197:6379', "root!2345"),
    execute('redis://47.93.4.198:6379', "root!2345")
]
```

```
29.
30. asyncio.run(asyncio.wait(task_list))
```

更多redis操作参考aioredis官网：https://aioredis.readthedocs.io/en/v1.3.0/start.html (https://aioredis.readthedocs.io/en/v1.3.0/start.html)

5.2 异步MySQL

当通过python去操作MySQL时，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。

安装Python异步操作redis模块

```
1. pip3 install aiomysql
```

示例1:

```python
import asyncio
import aiomysql


async def execute():
    # 网络IO操作：连接MySQL
    conn = await aiomysql.connect(host='127.0.0.1', port=3306, user='root', password='123', db='mysql', )

    # 网络IO操作：创建CURSOR
    cur = await conn.cursor()

    # 网络IO操作：执行SQL
    await cur.execute("SELECT Host,User FROM user")

    # 网络IO操作：获取SQL结果
    result = await cur.fetchall()
    print(result)

    # 网络IO操作：关闭链接
    await cur.close()
    conn.close()


asyncio.run(execute())
```

示例2：

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import asyncio
import aiomysql


async def execute(host, password):
    print("开始", host)
    # 网络IO操作：先去连接 47.93.40.197，遇到IO则自动切换任务，去连接47.93.40.198:6379
    conn = await aiomysql.connect(host=host, port=3306, user='root', password=password, db='mysql')

    # 网络IO操作：遇到IO会自动切换任务
    cur = await conn.cursor()

    # 网络IO操作：遇到IO会自动切换任务
    await cur.execute("SELECT Host,User FROM user")

    # 网络IO操作：遇到IO会自动切换任务
    result = await cur.fetchall()
    print(result)

    # 网络IO操作：遇到IO会自动切换任务
    await cur.close()
    conn.close()
    print("结束", host)


task_list = [
```

```
    execute('47.93.40.197', "root!2345"),
    execute('47.93.40.197', "root!2345")
]

asyncio.run(asyncio.wait(task_list))
```

5.3 FastAPI框架

FastAPI是一款用于构建API的高性能web框架，框架基于Python3.6+的 `type hints` 搭建。

接下里的异步示例以 `FastAPI` 和 `uvicorn` 来讲解（uvicorn是一个支持异步的asgi）。

安装FastAPI web 框架，

```
pip3 install fastapi
```

安装uvicorn，本质上为web提供socket server的支持的asgi（一般支持异步称asgi、不支持异步称wsgi）

```
pip3 install uvicorn
```

示例：

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import asyncio

import uvicorn
import aioredis
from aioredis import Redis
from fastapi import FastAPI

app = FastAPI()

REDIS_POOL = aioredis.ConnectionsPool('redis://47.193.14.198:6379', password="root123", minsize=1, maxsize=10)


@app.get("/")
def index():
    """ 普通操作接口 """
    return {"message": "Hello World"}


@app.get("/red")
async def red():
    """ 异步操作接口 """

    print("请求来了")

    await asyncio.sleep(3)
```

```
    # 连接池获取一个连接
    conn = await REDIS_POOL.acquire()
    redis = Redis(conn)

    # 设置值
    await redis.hmset_dict('car', key1=1, key2=2, key3=3)

    # 读取值
    result = await redis.hgetall('car', encoding='utf-8')
    print(result)

    # 连接归还连接池
    REDIS_POOL.release(conn)

    return result


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

在有多个用户并发请求的情况下，异步方式来编写的接口可以在IO等待过程中去处理其他的请求，提供性能。

例如：同时有两个用户并发来向接口 `http://127.0.0.1:5000/red` 发送请求，服务端只有一个线程，同一时刻只有一个请求被处理。异步处理可以提供并发是因为：当视图函数在处理第一个请求时，第二个请求此时是等待被处理的状态，当第一个请求遇到IO等待时，会自动切换去接收并处理第二个请求，当遇到IO时自动化切换至其他请求，一旦有请求IO执行完毕，则会再次回到指定请求向下继续执行其功能代码。

基于上下文管理，来实现自动化管理的案例：
示例1：redis

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import asyncio

import uvicorn
import aioredis
from aioredis import Redis
from fastapi import FastAPI

app = FastAPI()

REDIS_POOL = aioredis.ConnectionsPool('redis://47.193.14.198:6379', password="root123", minsize=1, maxsize=10)


@app.get("/")
def index():
    """ 普通操作接口 """
    return {"message": "Hello World"}


@app.get("/red")
async def red():
    """ 异步操作接口 """

    print("请求来了")

    async with REDIS_POOL.get() as conn:
```

```
        redis = Redis(conn)
        # 设置值
        await redis.hmset_dict('car', key1=1, key2=2, key3=3)

        # 读取值
        result = await redis.hgetall('car', encoding='utf-8')
        print(result)

    return result


if __name__ == '__main__':
    uvicorn.run("fast3:app", host="127.0.0.1", port=5000, log_level="info")
```

示例2：mysql

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import asyncio
import uvicorn
from fastapi import FastAPI
import aiomysql

app = FastAPI()

# 创建数据库连接池
pool = aiomysql.Pool(host='127.0.0.1', port=3306, user='root', password='123', db='mysql',
                     minsize=1, maxsize=10, echo=False, pool_recycle=-1, loop=asyncio.get_event_loop())


@app.get("/red")
async def red():
    """ 异步操作接口 """
    # 去数据库连接池申请链接
    async with pool.acquire() as conn:
        async with conn.cursor() as cur:
            # 网络IO操作：执行SQL
            await cur.execute("SELECT Host,User FROM user")
            # 网络IO操作：获取SQL结果
            result = await cur.fetchall()
            print(result)
            # 网络IO操作：关闭链接

    return {"result": "ok"}
```

```
29.
30.
31. if __name__ == '__main__':
32.     uvicorn.run("fast2:app", host="127.0.0.1", port=5000, log_level="info")
```

5.4 爬虫

在编写爬虫应用时，需要通过网络IO去请求目标数据，这种情况适合使用异步编程来提升性能，接下来我们使用支持异步编程的aiohttp模块来实现。

安装aiohttp模块

```
1. pip3 install aiohttp
```

示例：

```python
import aiohttp
import asyncio


async def fetch(session, url):
    print("发送请求：", url)
    async with session.get(url, verify_ssl=False) as response:
        text = await response.text()
        print("得到结果：", url, len(text))


async def main():
    async with aiohttp.ClientSession() as session:
        url_list = [
            'https://python.org',
            'https://www.baidu.com',
            'https://www.pythonav.com'
        ]
        tasks = [asyncio.create_task(fetch(session, url)) for url in url_list]

        await asyncio.wait(tasks)


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

总结

为了提升性能越来越多的框架都在向异步编程靠拢，例如：sanic、tornado、django3.0、django channels组件 等，用更少资源可以做处理更多的事，何乐而不为呢。

用户评论

M

为什么我运行最后一段代码，报错`RuntimeError: Event loop is closed`

milk_child 2020-08-18 08:03 回复 (/login/)

D

你把代码中的`url`改成`http`就不会报错了

djyyljz 2020-11-11 03:16 回复 (/login/)

G

武`Sir`，右边加个下拉滚动条感觉会更舒服哦~

gazh 2021-01-11 01:40 回复 (/login/)

⚠ 登录 (/login/) 或 注册 (/register/) 后才能发表评论

目录

- 垃圾回收机制剖析 (/wiki/detail/6/88/)
- **asyncio异步编程 (/wiki/detail/6/91/)**
    - 1.协程
        - 1.1 greenlet
        - 1.2 yield
        - 1.3 asyncio
        - 1.4 async & awit
        - 1.5 小结
    - 2.协程的意义
        - 2.1 爬虫案例
        - 2.2 小结
    - 3.异步编程
        - 3.1 事件循环
        - 3.2 协程和异步编程
            - 3.2.1 基本应用
            - 3.2.2 await
            - 3.2.3 Task对象
            - 3.2.4 asyncio.Future对象
            - 3.2.5 futures.Future对象
            - 3.2.6 异步迭代器
            - 3.2.6 异步上下文管理器
        - 3.3 小结
    - 4. uvloop
    - 5.实战案例
        - 5.1 异步Redis
        - 5.2 异步MySQL
        - 5.3 FastAPI框架
        - 5.4 爬虫
    - 总结
- 福利：优惠购买云服务器 (/wiki/detail/6/94/)
- 10分钟制作开源pip包 (/wiki/detail/6/95/)