一份详细的asyncio入门教程

爬虫工程...

> asyncio模块提供了使用协程构建并发应用的工具。它使用一种单线程单进程的的方式实现并发，应用的各个部分彼此合作, 可以显示的切换任务，一般会在程序阻塞I/O操作的时候发生上下文切换如等待读写文件,或者请求网络。同时asyncio也支持调度代码在将来的某个特定事件运行，从而支持一个协程等待另一个协程完成，以处理系统信号和识别其他一些事件。

异步并发的概念

对于其他的并发模型大多数采取的都是线性的方式编写。并且依赖于语言运行时系统或操作系统的底层线程或进程来适当地改变上下文，而基于asyncio的应用要求应用代码显示的处理上下文切换。
asyncio提供的框架以事件循环(event loop)为中心，程序开启一个无限的循环，程序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。

事件循环

事件循环是一种处理多并发量的有效方式，在维基百科中它被描述为「一种等待程序分配事件或消息的编程架构」，我们可以定义事件循环来简化使用轮询方法来监控事件，通俗的说法就是「当A发生时，执行B」。事件循环利用poller对象，使得程序员不用控制任务的添加、删除和事件的控制。事件循环使用回调方法来知道事件的发生。它是asyncio提供的「中央处理设备」，支持如下操作：

- 注册、执行和取消延迟调用（超时）
- 创建可用于多种类型的通信的服务端和客户端的Transports
- 启动进程以及相关的和外部通信程序的Transports

与事件循环交互的应用要显示地注册将运行的代码，让事件循环在资源可用时向应用代码发出必要的调用。如：一个套接字再没有更多的数据可以读取，那么服务器会把控制全交给事件循环。

Future

future是一个数据结构，表示还未完成的工作结果。事件循环可以监视Future对象是否完成。从而允许应用的一部分等待另一部分完成一些工作。

Task

task是Future的一个子类，它知道如何包装和管理一个协程的执行。任务所需的资源可用时，事件循环会调度任务允许，并生成一个结果，从而可以由其他协程消费。

异步方法

使用asyncio也就意味着你需要一直写异步方法。
一个标准方法是这样的：

```
def regular_double(x):
    return 2 * x
```

而一个异步方法：

```
async def async_double(x):
    return 2 * x
```

从外观上看异步方法和标准方法没什么区别只是前面多了个async。
“Async” 是“asynchronous”的简写，为了区别于异步函数，我们称标准函数为同步函数，
从用户角度异步函数和同步函数有以下区别：

要调用异步函数，必须使用**await**关键字。 因此，不要写regular_double(3)，而是写await async_double(3).
不能在同步函数里使用await，否则会出错。
句法错误：

```
def print_double(x):
    print(await async_double(x))   # <-- SyntaxError here
```

但是在异步函数中，await是被允许的：

```
async def print_double(x):
    print(await async_double(x))   # <-- OK!
```

协程

启动一个协程

一般异步方法被称之为协程(Coroutine)，asyncio事件循环可以通过多种不同的方法启动一个协

直接传入这个方

```
import asyncio


async def foo():
    print("这是一个协程")


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
        print("开始运行协程")
        coro = foo()
        print("进入事件循环")
        loop.run_until_complete(coro)
    finally:
        print("关闭事件循环")
        loop.close()
```

输出

```
开始运行协程
进入事件循环
这是一个协程
关闭事件循环
```

这就是最简单的一个协程的例子，下面让我们了解一下上面的代码.
第一步首先得到一个事件循环的应用也就是定义的对象loop。可以使用默认的事件循环，也可以实例化一个特定的循环类(比如uvloop),这里使用了默认循环run_until_complete(coro)方法用这个协程启动循环，协程返回时这个方法将停止循环。
run_until_complete的参数是一个futrue对象。当传入一个协程，其内部会自动封装成task，其中task是Future的子类。关于task和future后面会提到。

从协程中返回值

将上面的代码，改写成下面代码

```
import asyncio


async def foo():
    print("这是一个协程")
    return "返回值"


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
        print("开始运行协程")
        coro = foo()
        print("进入事件循环")
        result = loop.run_until_complete(coro)
        print(f"run_until_complete可以获取协程的{result}，默认输出None")
    finally:
        print("关闭事件循环")
        loop.close()
```

协程调用协程

一个协程可以启动另一个协程，从而可以任务根据工作内容，封装到不同的协程中。我们可以在协程中使用**await**关键字，链式的调度协程，来形成一个协程任务流。向下面的例子一样。

```
import asyncio


async def main():
    print("主协程")
    print("等待result1协程运行")
    res1 = await result1()
    print("等待result2协程运行")
    res2 = await result2(res1)
    return (res1,res2)


async def result1():
    print("这是result1协程")
    return "result1"


async def result2(arg):
    print("这是result2协程")
    return f"result2接收了一个参数,{arg}"


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
        result = loop.run_until_complete(main())
        print(f"获取返回值:{result}")
    finally:
        print("关闭事件循环")
        loop.close()
```

输出

```
主协程
等待result1协程运行
这是result1协程
等待result2协程运行
这是result2协程
获取返回值:('result1', 'result2接收了一个参数,result1')
关闭事件循环
```

协程中调用普通函数

在协程中可以通过一些方法去调用普通的函数。可以使用的关键字有call_soon,call_later, call_at。

call_soon

在下一个迭代的时间循环中立刻调用回调函数,大部分的回调函数支持位置参数，而不支持"关键字参数"，如果是想要使用关键字参数，则推荐使用functools.aprtial()对方法进一步包装.可选关键字context允许指定要运行的回调的自定义contextvars.Context。当没有提供上下文时使用当前上下文。在Python 3.7中， asyncio
协程加入了对上下文的支持。使用上下文就可以在一些场景下隐式地传递变量，比如数据库连接session等，而不需要在所有方法调用显示地传递这些变量。
下面来看一下具体的使用例子。

```
import asyncio
import functools


def callback(args, *, kwargs="defalut"):
    print(f"普通函数做为回调函数,获取参数:{args},{kwargs}")


async def main(loop):
    print("注册callback")
    loop.call_soon(callback, 1)
    wrapped = functools.partial(callback, kwargs="not defalut")
    loop.call_soon(wrapped, 2)
    await asyncio.sleep(0.2)


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(main(loop))
finally:
    loop.close()
```

输出结果

```
注册callback
普通函数做为回调函数,获取参数:1,defalut
普通函数做为回调函数,获取参数:2,not defalut
```

通过输出结果我们可以发现我们在协程中成功调用了一个普通函数，顺序的打印了1和2。

有时候我们不想立即调用一个函数，此时我们就可以call_later延时去调用一个函数了。

call_later

```
loop.call_later(delay, callback, *args, context=None)
```

首先简单的说一下它的含义，就是事件循环在delay多长时间之后才执行callback函数.
配合上面的call_soon让我们看一个小例子

```
import asyncio


def callback(n):
    print(f"callback {n} invoked")
```

```
    print("注册callbacks")
    loop.call_later(0.2, callback, 1)
    loop.call_later(0.1, callback, 2)
    loop.call_soon(callback, 3)
    await asyncio.sleep(0.4)


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(main(loop))
    finally:
        loop.close()
```

输出

```
注册callbacks
callback 3 invoked
callback 2 invoked
callback 1 invoked
```

通过上面的输出可以得到如下结果：
1.**call_soon**会在call_later之前执行，和它的位置在哪无关
2.**call_later**的第一个参数越小，越先执行。

call_at

```
loop.call_at(when, callback, *args, context=None)
```

call_at第一个参数的含义代表的是一个单调时间，它和我们平时说的系统时间有点差异，
这里的时间指的是事件循环内部时间，可以通过loop.time()获取，然后可以在此基础上进行操作。
后面的参数和前面的两个方法一样。实际上call_later内部就是调用的call_at。

```
import asyncio


def call_back(n, loop):
    print(f"callback {n} 运行时间点{loop.time()}")


async def main(loop):
    now = loop.time()
    print("当前的内部时间", now)
    print("循环时间", now)
    print("注册callback")
    loop.call_at(now + 0.1, call_back, 1, loop)
    loop.call_at(now + 0.2, call_back, 2, loop)
    loop.call_soon(call_back, 3, loop)
    await asyncio.sleep(1)


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
```

```
loop.close()
```

输出

```
进入事件循环
当前的内部时间 4412.152849525
循环时间 4412.152849525
注册callback
callback 3 运行时间点4412.152942526
callback 1 运行时间点4412.253202825
callback 2 运行时间点4412.354262512
关闭循环
```

因为call_later内部实现就是通过call_at所以这里就不多说了。

Future

获取Futrue里的结果

future表示还没有完成的工作结果。事件循环可以通过监视一个future对象的状态来指示它已经完成。future对象有几个状态：

- Pending
- Running
- Done
- Cancelled
  创建future的时候，task为pending，事件循环调用执行的时候当然就是running，调用完毕自然就是done，如果需要停止事件循环，就需要先把task取消，状态为cancel。

```
import asyncio


def foo(future, result):
    print(f"此时future的状态:{future}")
    print(f"设置future的结果:{result}")
    future.set_result(result)
    print(f"此时future的状态:{future}")


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
        all_done = asyncio.Future()
        loop.call_soon(foo, all_done, "Future is done!")
        print("进入事件循环")
        result = loop.run_until_complete(all_done)
        print("返回结果", result)
    finally:
        print("关闭事件循环")
        loop.close()
    print("获取future的结果", all_done.result())
```

输出

```
设置future的结果:Future is done!
此时future的状态:<Future finished result='Future is done!'>
返回结果 Future is done!
关闭事件循环
获取future的结果 Future is done!
```

`

可以通过输出结果发现，调用set_result之后future对象的状态由pending变为finished，Future的实例all_done会保留提供给方法的结果，可以在后续使用。

Future对象使用await

future和协程一样可以使用await关键字获取其结果。

```
import asyncio


def foo(future, result):
    print("设置结果到future", result)
    future.set_result(result)


async def main(loop):
    all_done = asyncio.Future()
    print("调用函数获取future对象")
    loop.call_soon(foo, all_done, "the result")

    result = await all_done
    print("获取future里的结果", result)


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(main(loop))
    finally:
        loop.close()
```

Future回调

Future 在完成的时候可以执行一些回调函数，回调函数按注册时的顺序进行调用:

```
import asyncio
import functools


def callback(future, n):
    print('{}: future done: {}'.format(n, future.result()))


async def register_callbacks(all_done):
    print('注册callback到future对象')
    all_done.add_done_callback(functools.partial(callback, n=1))
    all_done.add_done_callback(functools.partial(callback, n=2))
```

```
    print('设置future的结果')
    all_done.set_result('the result')

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
        all_done = asyncio.Future()
        loop.run_until_complete(main(all_done))
    finally:
        loop.close()
```

通过add_done_callback方法给funtrue任务添加回调函数，当funture执行完成的时候,就会调用回调函数。并通过参数future获取协程执行的结果。
到此为止，我们就学会了如何在协程中调用一个普通函数并获取其结果。

并发的执行任务

任务（Task）是与事件循环交互的主要途径之一。任务可以包装协程，可以跟踪协程何时完成。任务是Future的子类，所以使用方法和future一样。协程可以等待任务，每个任务都有一个结果，在它完成之后可以获取这个结果。
因为协程是没有状态的，我们通过使用create_task方法可以将协程包装成有状态的任务。还可以在任务运行的过程中取消任务。

```
import asyncio


async def child():
    print("进入子协程")
    return "the result"


async def main(loop):
    print("将协程child包装成任务")
    task = loop.create_task(child())
    print("通过cancel方法可以取消任务")
    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("取消任务抛出CancelledError异常")
    else:
        print("获取任务的结果", task.result())


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(main(loop))
    finally:
        loop.close()
```

输出

```
将协程child包装成任务
通过cancel方法可以取消任务
```

```
将协程child包装成任务
通过cancel方法可以取消任务
进入子协程
获取任务的结果 the result
```

另外出了使用loop.create_task将协程包装为任务外还可以使用asyncio.ensure_future(coroutine)建一个task。在python3.7中可以使用asyncio.create_task创建任务。

组合协程

一系列的协程可以通过await链式的调用，但是有的时候我们需要在一个协程里等待多个协程，比如我们在一个协程里等待1000个异步网络请求，对于访问次序有没有要求的时候，就可以使用另外的关键字wait或gather来解决了。wait可以暂停一个协程，直到后台操作完成。

等待多个协程

Task的使用

```
import asyncio


async def num(n):
    try:
        await asyncio.sleep(n*0.1)
        return n
    except asyncio.CancelledError:
        print(f"数字{n}被取消")
        raise


async def main():
    tasks = [num(i) for i in range(10)]
    complete, pending = await asyncio.wait(tasks, timeout=0.5)
    for i in complete:
        print("当前数字",i.result())
    if pending:
        print("取消未完成的任务")
        for p in pending:
            p.cancel()


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(main())
    finally:
        loop.close()
```

输出

```
当前数字 1
当前数字 2
当前数字 0
当前数字 4
```

```
数字9被取消
数字6被取消
数字8被取消
数字7被取消
```

可以发现我们的结果并没有按照数字的顺序显示，在内部wait()使用一个set保存它创建的Task实例。因为set是无序的所以这也就是我们的任务不是顺序执行的原因。wait的返回值是一个元组，包括两个集合，分别表示已完成和未完成的任务。wait第二个参数为一个超时值
达到这个超时时间后，未完成的任务状态变为pending，当程序退出时还有任务没有完成此时就会看到如下的错误提示。

```
Task was destroyed but it is pending!
task: <Task pending coro=<num() done, defined at 11.py:12> wait_for=<Future pe
Task was destroyed but it is pending!
task: <Task pending coro=<num() done, defined at 11.py:12> wait_for=<Future pe
Task was destroyed but it is pending!
task: <Task pending coro=<num() done, defined at 11.py:12> wait_for=<Future pe
```

此时我们可以通过迭代调用cancel方法取消任务。也就是这段代码

```
if pending:
        print("取消未完成的任务")
        for p in pending:
            p.cancel()
```

gather的使用

gather的作用和wait类似不同的是。
1.gather任务无法取消。
2.返回值是一个结果列表
3.可以按照传入参数的顺序，顺序输出。
我们将上面的代码改为gather的方式

```
import asyncio


async def num(n):
    try:
        await asyncio.sleep(n * 0.1)
        return n
    except asyncio.CancelledError:
        print(f"数字{n}被取消")
        raise


async def main():
    tasks = [num(i) for i in range(10)]
    complete = await asyncio.gather(*tasks)
    for i in complete:
        print("当前数字", i)


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
```

输出

```
当前数字 0
当前数字 1
....中间部分省略
当前数字 9
```

gather通常被用来阶段性的一个操作，做完第一步才能做第二步，比如下面这样

```
import asyncio

import time


async def step1(n, start):
    await asyncio.sleep(n)
    print("第一阶段完成")
    print("此时用时", time.time() - start)
    return n


async def step2(n, start):
    await asyncio.sleep(n)
    print("第二阶段完成")
    print("此时用时", time.time() - start)
    return n


async def main():
    now = time.time()
    result = await asyncio.gather(step1(5, now), step2(2, now))
    for i in result:
        print(i)
    print("总用时", time.time() - now)


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(main())
    finally:
        loop.close()
```

输出

```
第二阶段完成
此时用时 2.0014898777008057
第一阶段完成
此时用时 5.002960920333862
5
2
总用时 5.003103017807007
```

可以通过上面结果得到如下结论：

最长时间的那个。

as_complete是一个生成器，会管理指定的一个任务列表，并生成他们的结果。每个协程结束运行时一次生成一个结果。与wait一样，as_complete不能保证顺序，不过执行其他动作之前没有必要等待所以后台操作完成。

```
import asyncio
import time


async def foo(n):
    print('Waiting: ', n)
    await asyncio.sleep(n)
    return n


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

    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]
    for task in asyncio.as_completed(tasks):
        result = await task
        print('Task ret: {}'.format(result))


now = lambda : time.time()
start = now()

loop = asyncio.get_event_loop()
done = loop.run_until_complete(main())
print(now() - start)
```

输出

```
Waiting:  1
Waiting:  2
Waiting:  4
Task ret: 1
Task ret: 2
Task ret: 4
4.004292249679565
```

可以发现结果逐个输出。

到此为止第一部分就结束了，对于asyncio入门级学习来说这些内容就够了。如果想继续跟进asyncio的内容，敬请期待后面的内容。

参考资料

- The Python 3 Standard Library by Example
- docs.python.org/3/libra...

知乎 首发于 python学习开发

无障

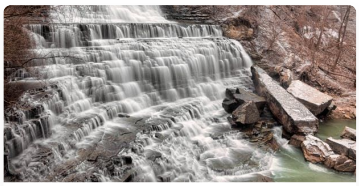
Python 异步 asyncio

文章被以下专栏收录

python学习开发
和我一起学习python

Python程序员
公众号：Python爱好者社区（python_shequ）欢迎投稿！

推荐阅读



python异步asyncio模块的使用

Dwzb

在python中如何以异步的方式调用第三方库提供的同步API

在 关于asyncio的基本用法中提到，asyncio并不是多线程。在协程中调用同步（阻塞函数），都占用同一线程的CPU时间，即当前线程会被阻塞（即协程只会在等待一个协程时可能出让CPU，如果是普...

杨勇 发表于解语科技

异步编程 101: 是什么、小试 Python asyncio

廖长江

python异步编习(二)

爬虫工程师陈

2 条评论 切换为时间排序

写下你的评论...

Craft 2021-06-27

写的怪详细，但是看不懂，还是要实践，结合实际例子理解。

赞

bestgopher 2019-04-19

再看了一遍

赞

▲ 赞同 43 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ···