# asyncio简介

## 一.什么是asyncio模块？

asyncio是原生协程关键字：async和await，它们的底层基于生成器函数；

asyncio模块也是在Python中实现协程的模块。

## 二.协程概念

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

协程的目的是通过一个线程利用其IO等待的时候，再去干点别的事情，别闲着！

协程不是被**[操作系统](https://so.csdn.net/so/search?q=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_38938108/article/details/_blank)内核**所管理，而完全是由程序所控制（也就是在用户态执行），是一种用户状态内的上下文切换技术，其实就是通过一个线程实现代码块相互切换执行，因此也称为轻量级的线程。

协程的切换完全由**程序控制**，而非通过操作系统内核来实现，因此对资源的开销更小；

**1.协程函数**

不是像普通函数那样直接调用运行的，必须添加到事件循环中，然后由事件循环去运行，单独运行协程函数是不会有结果的。

**2.通过事件循环运行协程函数的两种方式**

**方式一：**

创建事件循环对象loop，即asyncio.get\_event\_loop()，通过事件循环运行协程函数



**方式二：**

直接通过asyncio.run(function\_name)运行协程函数。但是需要注意的是，首先run函数是python3.7版本新添加的，前面的版本是没有的；其次，这个run函数总是会创建一个新的事件循环并在run结束之后关闭事件循环，所以，如果在同一个线程中已经有了一个事件循环，则不能再使用这个函数了，因为同一个线程不能有两个事件循环，而且这个run函数不能同时运行两次，因为他已经创建一个了。即同一个线程中是不允许有多个事件循环loop的。

**3.什么是事件循环？**

线程一直在各个协程方法之间永不停歇的游走，遇到一个yield from 或者await就悬挂起来，然后又走到另外一个方法，依次进行下去，直到事件循环所有的方法执行完毕。

**4.什么是awaitable对象——即可暂停等待的对象？**

有三类对象是可等待的，即coroutines，Tasks和Futures。

coroutine：本质上就是一个协程函数；

Tasks：任务，就是要完成某件事情，其实就是对协程函数进一步的封装；

Future：它是一个“更底层”的概念，代表一个异步操作的最终结果，因为异步操作一般用于耗时操作，结果不会立即得到，会在“将来”得到异步运行的结果，故而命名为Future。

三者的关系，coroutine可以自动封装成task，而Task是Future的子类。

**5.什么是task任务？**

Task用来并发调度的协程，即对协程函数的进一步包装；因为单纯的协程函数仅仅是一个函数而已，将其包装成任务，任务是可以包含各种状态的，[异步编程](https://so.csdn.net/so/search?q=%E5%BC%82%E6%AD%A5%E7%BC%96%E7%A8%8B&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_37674086/article/details/_blank)最重要的就是对异步操作状态的把控了。

（1）创建任务（两种方法）：

**方法一：**

task = asyncio.create\_task(coro())   # 这是3.7版本新添加的

**方法二：**

task = asyncio.ensure\_future(coro())

也可以使用：

loop.create\_future()

loop.create\_task(coro)

**5.什么是future？**

Future是一个较低层的可等待（awaitable）对象，他表示的是异步操作的最终结果，当一个Future对象被等待的时候，协程会一直等待，直到Future已经运算完毕。

Future是Task的父类，一般情况下，已不用去管它们两者的详细区别，也没有必要去用Future，用Task就可以了。

## 三.asyncio的应用场景

**网络编程：**在网络编程中，asyncio能够显著提高性能，因为它可以非阻塞地处理多个网络连接。例如，在编写Web服务器、聊天服务器、或者需要处理大量并发连接的其他网络应用时，使用asyncio可以高效地处理这些连接，而无需为每个连接创建一个新的线程或进程。

**I/O密集型任务：**对于需要从文件、数据库或网络读取大量数据的任务，asyncio能够显著减少等待时间。通过使用asyncio，你可以在等待一个I/O操作完成时，开始处理其他任务，从而实现真正的并发。

**并发控制：**在某些情况下，你可能需要控制并发任务的数量，以避免资源耗尽或性能下降。asyncio提供了对并发任务数量的精细控制，例如通过asyncio.Semaphore或asyncio.LimitedSemaphore等同步原语。

**GUI编程：**在图形用户界面（GUI）编程中，asyncio可以用于处理用户的输入和界面的更新，而不会阻塞主线程。这使得GUI能够保持响应性，同时执行后台任务。

**长时间运行的任务：**对于需要较长时间才能完成的任务，如大型计算或数据分析任务，asyncio可以帮助你编写非阻塞的代码，从而在处理这些任务的同时，还能响应用户的其他请求或执行其他任务。

**与其他异步库的集成：**许多现代Python库和框架，如aiohttp（异步HTTP客户端/服务器框架）、websockets（WebSocket客户端/服务器库）等，都支持asyncio。这使得你可以方便地使用这些库来构建高效、响应迅速的异步应用。

## 四.asyncio常见方法

**1、asyncio.run()**

用来运行[协程函数](https://so.csdn.net/so/search?q=%E5%8D%8F%E7%A8%8B%E5%87%BD%E6%95%B0&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_37871657/article/details/_blank)(异步函数)或协程对象。

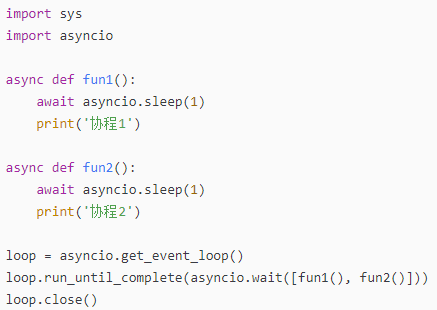


**2、asyncio.sleep(delay, result=None, \*, loop=None)**

用来阻塞指定的秒数，，也就是等待秒数。

**3、asyncio.get\_event\_loop()**

获取当前事件循环，常与run\_until\_complete()配合使用。



**注意：**

**loop=asyncio.get\_running\_loop()**

返回（获取）在当前线程中正在运行的事件循环，如果没有正在运行的事件循环，则会显示错误；它是python3.7中新添加的

**loop=asyncio.get\_event\_loop()**

获得一个事件循环，如果当前线程还没有事件循环，则创建一个新的事件循环loop；

**loop=asyncio.set\_event\_loop(loop)**

设置一个事件循环为当前线程的事件循环；

**loop=asyncio.new\_event\_loop()**

创建一个新的事件循环

**4、run\_until\_complete()**

运行直到future([Future](https://docs.python.org/zh-cn/3.8/library/asyncio-future.html" \l "asyncio.Future" \o "asyncio.Future)的实例)被完成。

格式：

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(协程)

**5、loop.close()**

关闭事件循环。

格式：

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(协程)

loop.close()

**6、asyncio.ensure\_future()**

用于在给定的事件循环中创建一个新的Future对象。Future是asyncio中用于表示尚未完成的操作的对象。你可以使用它来等待某个操作的结果，或者将结果设置到Future中以通知等待者。这个函数通常在较低级别的编程中使用，当你需要直接操作Future对象时。

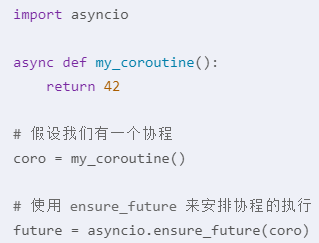
格式：

loop = asyncio.get\_event\_loop()

future = loop.create\_future()

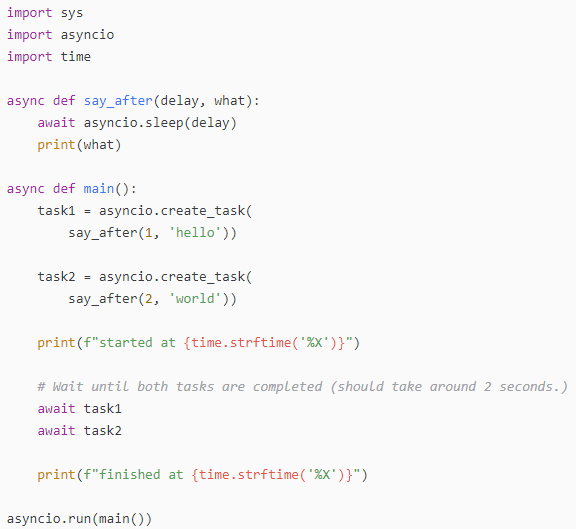
**7、create\_future()**

这个函数用于确保传入的参数(一个协程或Future对象)在事件循环中正确地安排执行。如果传入的是一个协程，ensure\_future()会将其封装在一个 Future 中，并安排其执行。如果传入的是一个Future对象，则直接返回该对象。这个函数提供了更高级别的抽象，使得开发者无需关心传入的是协程还是 Future，只需确保它被正确地安排执行。



**8、asyncio.create\_task()**

用来将一个协程打包为一个Task排入日程准备执行，并返回Task对象。

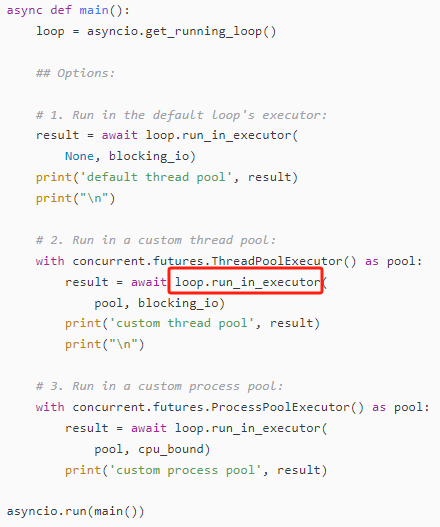


**9、add\_done\_callback()**



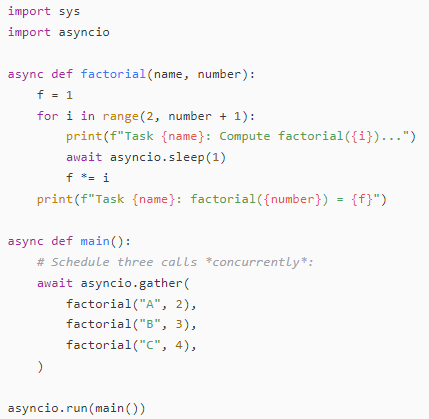
**10、run\_in\_executor()**

安排在指定的执行器中调用 func。



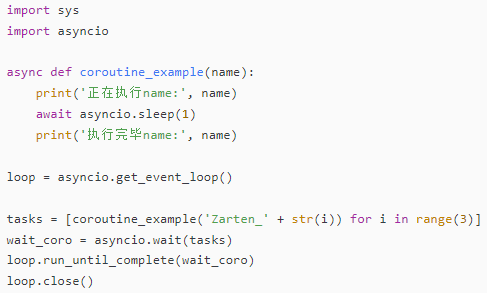
**11、gather**

用来并发运行 aws 序列中的可等待对象。如果 aws 中的某个可等待对象为协程，它将自动作为一个任务加入日程。



**12、wait\_for**

函数并发地运行 aws 可迭代对象中的可等待对象并进入阻塞状态直到满足 return\_when 所指定的条件。

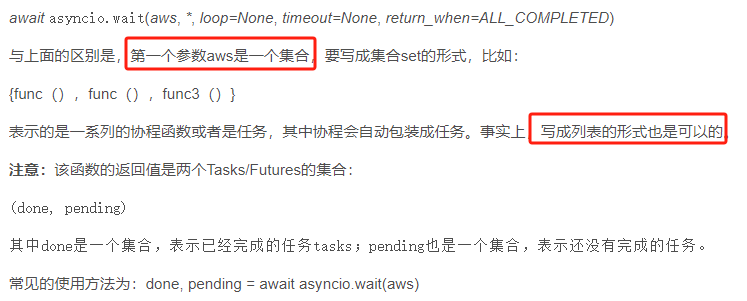


**13、run\_coroutine\_threadsafe**

向指定事件循环提交一个协程。线程安全。



**14、 asyncio.wait**



## 五.asyncio中一些方法的基本区别

**1.loop.stop() 与loop.close()的区别**

loop.stop()：是让事件循环在完成当前任务后自然停止，而不会立即释放资源。

loop.close()：是立即关闭事件循环并释放所有相关资源。

**2.loop.ensure\_future() 与loop.create\_task()的区别**

**参数类型**： ensure\_future()可以接受协程或Future对象，而create\_task()只接受协程。

**语义清晰度**：create\_task()的语义更清晰，专门用于创建协程任务。而ensure\_future()的用途更广泛，可以用于确保任何协程或Future在事件循环中执行。

**推荐使用**：由于create\_task()提供了更清晰的语义，并且在大多数情况下只用于协程，因此它通常是创建异步任务的首选方法。然而，如果你需要处理既有协程又有Future的情况，ensure\_future()仍然是一个有用的工具。

**3.asyncio.run() 与loop.run\_until\_complete()的区别**

**简化程度**： asyncio.run() 更简化，它隐藏了事件循环的获取和关闭的细节。而 asyncio.get\_event\_loop().run\_until\_complete() 需要你手动处理这些细节。

**错误处理**：使用 asyncio.run() 时，如果在协程中发生异常，它会直接传播到调用者。而使用 run\_until\_complete() 时，异常会被封装在一个 CancelledError 异常中，除非你使用 try/except 块来捕获它。

**Python版本**：asyncio.run() 是在 Python 3.7 中引入的，而 asyncio.get\_event\_loop().run\_until\_complete() 在更早的 Python 版本中就已经存在。

**上下文管理**：asyncio.run() 在其内部使用了一个上下文管理器来确保事件循环的正确关闭，即使在发生异常的情况下也是如此。而使用 run\_until\_complete() 时，你需要自己确保事件循环的正确关闭，特别是在异常处理时。

在大多数情况下，推荐使用 asyncio.run()，因为它更简单且更易于管理。但是，如果你需要更细粒度的控制事件循环（例如，在更复杂的应用程序中，你可能需要管理多个事件循环或需要更精细的异常处理），那么你可能需要使用 get\_event\_loop() 和 run\_until\_complete()。

**4.直接运行异步函数和使用asyncio.create\_task()的区别**

**阻塞与并发：**直接运行异步函数会阻塞当前事件循环，直到异步函数完成。而使用asyncio.create\_task()创建任务可以安排多个异步函数并发执行，而不会阻塞事件循环。

**执行控制：**直接运行异步函数时，你只能在异步函数完成后继续执行后续代码。而使用create\_task()可以让你在异步函数执行期间安排和执行其他任务或操作。

**结果获取：**直接运行异步函数会立即返回结果（如果使用了await）。而使用create\_task()创建的任务对象表示一个未来的结果，你需要显式地使用await来等待结果。

1. **asyncio.wait与asyncio.gather的区别**

**功能上的共同点**：

两者都用于等待一组异步任务（即协程）的完成。

两者都允许你并发地执行多个异步任务。

**返回值和处理方式的不同：**

asyncio.wait：

返回一个完成的（done）和未完成的（pending）任务集合。

对于完成的任务，你需要调用其 .result() 方法来获取结果。

它提供了更多的灵活性，例如可以等待特定的条件（如所有任务完成、第一个任务完成或第一个任务出错）。

asyncio.gather：

返回一个包含所有任务结果的迭代器。

你不需要对结果进行额外的处理或调用 .result()，因为 gather 已经为你收集了所有结果。

它简化了获取所有任务结果的过程。

错误处理：

asyncio.wait 允许你指定如何处理异常，例如通过提供一个异常处理函数。

asyncio.gather 会将第一个未捕获的异常传播给调用者，或者如果所有任务都成功完成，则返回所有结果。

**使用场景：**

如果你需要更细粒度的控制，例如只关心某些任务何时完成，或者想自定义异常处理逻辑，那么 asyncio.wait 可能更适合你。

如果你只是想简单地并发执行多个任务并收集它们的结果，那么 asyncio.gather 通常更为方便。

## 六.asyncio的事件循环函数

1、asyncio.get\_running\_loop()

函数返回当前 OS 线程中正在运行的事件循环。

2、asyncio.get\_event\_loop()

函数获取当前事件循环。

3、asyncio.set\_event\_loop(loop)

函数将 loop 设置为当前 OS 线程的当前事件循环。

4、asyncio.new\_event\_loop()

函数创建一个新的事件循环。

5、loop.run\_until\_complete(future)

函数运行直到 future (Future 的实例) 被完成。

6、loop.run\_forever()

函数运行事件循环直到 stop() 被调用。

7、loop.stop()

函数停止事件循环。

8、loop.is\_running()

函数返回 True 如果事件循环当前正在运行。

9、loop.is\_closed()

函数如果事件循环已经被关闭，返回 True 。

10、loop.close()

函数关闭事件循环。

11、loop.create\_future()

函数创建一个附加到事件循环中的 asyncio.Future 对象。

12、loop.create\_task(coro, \*, name=None)

函数安排一个 协程 的执行。返回一个 Task 对象。

13、loop.set\_task\_factory(factory)

函数设置一个 task 工厂 ， 被用于 loop.create\_task() 。

14、loop.get\_task\_factory()

函数返回一个任务工厂，或者如果是使用默认值则返回 None。

## 七.Future

1、asyncio.Future(\*, loop=None)

函数是一个 Future 代表一个异步运算的最终结果。线程不安全。

2、asyncio.isfuture(obj)

函数用来判断如果 obj 为一个 asyncio.Future类的示例、 asyncio.Task 类的实例或者一个具有 \_asyncio\_future\_blocking 属性的对象，返回 True。

3、asyncio.ensure\_future(obj, \*, loop=None)

函数创建新任务。

4、asyncio.wrap\_future(future, \*, loop=None)

函数将一个 concurrent.futures.Future 对象封装到 asyncio.Future 对象中。

**Future 对象相关函数：**

1、fut.result()

函数返回 Future 的结果。

2、fut.set\_result(result)

函数将 Future 标记为 完成 并设置结果。

3、fut.set\_exception(exception)

函数将 Future 标记为 完成 并设置一个异常。

4、fut.done()

函数如果 Future 为已 完成 则返回 True 。

5、fut.cancelled()

函数是如果 Future 已取消则返回 True

6、fut.add\_done\_callback(callback, \*, context=None)

函数添加一个在 Future 完成 时运行的回调函数。

7、fut.remove\_done\_callback(callback)

函数从回调列表中移除 callback 。

8、fut.cancel()

函数取消 Future 并调度回调函数。

9、fut.exception()

函数返回 Future 已设置的异常。

10、fut.get\_loop()

函数返回 Future 对象已绑定的事件循环。

## 八.调度回调和延迟回调

1、loop.call\_soon()

函数安排 callback 在事件循环的下一次迭代时附带 args 参数被调用。回调按其注册顺序被调用。每个回调仅被调用一次。方法不是线程安全的。

2、loop.call\_soon\_threadsafe()

函数是 call\_soon() 的线程安全变体。必须被用于安排 来自其他线程 的回调。

3、loop.call\_later()

函数安排 callback 在给定的 delay 秒（可以是 int 或者 float）后被调用。

4、loop.call\_at()

函数安排 callback 在给定的绝对时间戳的时间（一个 int 或者 float）被调用，使用与 loop.time() 同样的时间参考。

5、loop.time()

函数根据时间循环内部的单调时钟，返回当前时间， float 值。

示例：



## 九.socket连接和Streams函数

1、loop.create\_connection()

函数打开一个流式传输连接，连接到由 host 和 port 指定的地址。

2、loop.create\_server()

函数创建TCP服务 (socket 类型 SOCK\_STREAM ) 监听 host 地址的 port 端口。

3、loop.create\_unix\_server()

函数与 loop.create\_server() 类似但是专用于 AF\_UNIX 套接字族。path 是必要的 Unix 域套接字名称，除非提供了 sock 参数。 抽象的 Unix 套接字, str, bytes 和 Path 路径都是受支持的。

4、loop.connect\_accepted\_socket()

函数将已被接受的连接包装成一个传输/协议对。

5、loop.sock\_recv()

函数从 sock 接收至多 nbytes。 socket.recv() 的异步版本。

6、loop.sock\_recv\_into()

函数从 sock 接收数据放入 buf 缓冲区。 模仿了阻塞型的 socket.recv\_into() 方法。

7、loop.sock\_sendall()

函数将 data 发送到 sock 套接字。 socket.sendall() 的异步版本。

8、loop.sock\_accept()

函数接受一个连接。 模仿了阻塞型的 socket.accept() 方法。

9、loop.sock\_sendfile()

函数在可能的情况下使用高性能的 os.sendfile 发送文件。 返回所发送的字节总数。

10、asyncio.open\_connection()

函数建立网络连接并返回一对 (reader, writer) 对象。

11、asyncio.start\_server()

函数启动套接字服务。

12、asyncio.open\_unix\_connection()

函数建立一个 Unix 套接字连接并返回 (reader, writer) 这对返回值。与 open\_connection() 相似，但是操作在 Unix 套接字上。

13、asyncio.start\_unix\_server()

函数启动一个Unix socket服务。与 start\_server() 相似，但是是在 Unix 套接字上的操作。

14、asyncio.StreamReader

这个类表示一个提供api来从IO流中读取数据的读取器对象。

14.1、reader.read(n=-1)

函数读取 n 个byte. 如果没有设置 n , 则自动置为 -1 ，读至 EOF 并返回所有读取的byte。

14.2、reader.readline()

函数读取一行，其中“行”指的是以 \n 结尾的字节序列。如果读到EOF而没有找到 \n ，该方法返回部分读取的数据。如果读到EOF，且内部缓冲区为空，则返回一个空的 bytes 对象。

14.3、reader.readexactly(n)

函数精准读取 n 个 bytes，不能超过也不能少于。

14.4、reader.readuntil()

函数从流中读取数据直至遇到 分隔符成功后，数据和指定的separator将从内部缓冲区中删除(或者说被消费掉)。返回的数据将包括在末尾的指定separator。如果读取的数据量超过了配置的流限制，将引发 LimitOverrunError 异常，数据将留在内部缓冲区中并可以再次读取。如果在找到完整的separator之前到达EOF，则会引发 IncompleteReadError 异常，并重置内部缓冲区。 IncompleteReadError.partial 属性可能包含指定separator的一部分。

reader.at\_eof()

函数如果缓冲区为空并且 feed\_eof() 被调用，则返回 True 。

15、asyncio.StreamWriter

这个类表示一个写入器对象，该对象提供api以便于写数据至IO流中。

15.1、writer.write(data)

函数会尝试立即将 data 写入到下层的套接字。 如果写入失败，数据会被排入内部写缓冲队列直到可以被发送。

15.2、writer.writelines(data)

函数会立即尝试将一个字节串列表（或任何可迭代对象）写入到下层的套接字。 如果写入失败，数据会被排入内部写缓冲队列直到可以被发送。

15.3、writer.close()

函数会关闭流以及下层的套接字。

15.4、writer.can\_write\_eof()

函数如果下层的传输支持 write\_eof() 方法则返回``True``，否则返回 False。

15.5、writer.write\_eof()

函数在已缓冲的写入数据被刷新后关闭流的写入端。

15.6、writer.transport()

函数返回下层的 asyncio 传输。

15.7、writer.drain()

函数等待直到可以适当地恢复写入到流。

15.8、writer.is\_closing()

函数如果流已被关闭或正在被关闭则返回 True。

15.9、writer.wait\_closed()

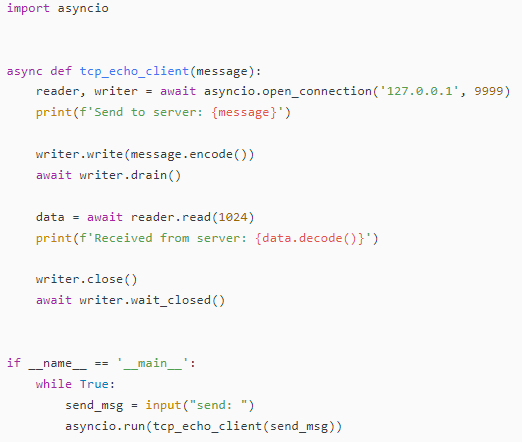
函数等待直到流被关闭。

实例：

服务端：



客户端：



## 十.错误处理API

1、loop.set\_exception\_handler(handler)

函数将 handler 设置为新的事件循环异常处理器。

1. loop.get\_exception\_handler()

函数返回当前的异常处理器，如果没有设置异常处理器，则返回 None 。

3、loop.default\_exception\_handler(context)

函数默认的异常处理器。

4、loop.call\_exception\_handler(context)

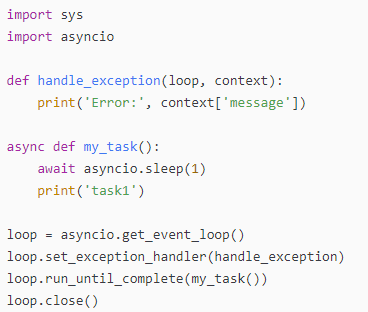
函数调用当前事件循环的异常处理器。

5、loop.get\_debug()

函数获取事件循环调试模式设置（bool）。

6、loop.set\_debug(enabled: bool)

函数设置事件循环的调试模式。



## 十一.阻塞与非阻塞

**阻塞**

阻塞状态指程序未得到所需计算资源时被挂起的状态。程序在等待某个操作完成期间，自身无法继续干别的事情，则称该程序在该操作上是阻塞的。

常见的阻塞形式有：网络 I/O 阻塞、磁盘 I/O 阻塞、用户输入阻塞等。阻塞是无处不在的，包括 CPU 切换上下文时，所有的进程都无法真正干事情，它们也会被阻塞。如果是多核 CPU 则正在执行上下文切换操作的核不可被利用。

**非阻塞**

程序在等待某操作过程中，自身不被阻塞，可以继续运行干别的事情，则称该程序在该操作上是非阻塞的。

非阻塞并不是在任何程序级别、任何情况下都可以存在的。仅当程序封装的级别可以囊括独立的子程序单元时，它才可能存在非阻塞状态。

非阻塞的存在是因为阻塞存在，正因为某个操作阻塞导致的耗时与效率低下，我们才要把它变成非阻塞的。

## 十二.同步与异步

**同步**

不同程序单元为了完成某个任务，在执行过程中需靠某种通信方式以协调一致，称这些程序单元是同步执行的。

例如购物系统中更新商品库存，需要用“行锁”作为通信信号，让不同的更新请求强制排队顺序执行，那更新库存的操作是同步的。

简言之，同步意味着有序。

**异步**

为完成某个任务，不同程序单元之间过程中无需通信协调，也能完成任务的方式，不相关的程序单元之间可以是异步的。

例如，爬虫下载网页。调度程序调用下载程序后，即可调度其他任务，而无需与该下载任务保持通信以协调行为。不同网页的下载、保存等操作都是无关的，也无需相互通知协调。这些异步操作的完成时刻并不确定。

简言之，异步意味着无序。

## 十三.协程与进程

**多进程**

多进程就是利用 CPU 的多核优势，在同一时间并行地执行多个任务，可以大大提高执行效率。

**协程**

协程，英文叫做 Coroutine，又称微线程，纤程，协程是一种用户态的轻量级线程。

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此协程能保留上一次调用时的状态，即所有局部状态的一个特定组合，每次过程重入时，就相当于进入上一次调用的状态。

协程本质上是个单进程，协程相对于多进程来说，无需线程上下文切换的开销，无需原子操作锁定及同步的开销，编程模型也非常简单。

我们可以使用协程来实现异步操作，比如在网络爬虫场景下，我们发出一个请求之后，需要等待一定的时间才能得到响应，但其实在这个等待过程中，程序可以干许多其他的事情，等到响应得到之后才切换回来继续处理，这样可以充分利用 CPU 和其他资源，这就是异步协程的优势。