链滴

Python 3.5的async和await特性(PEP492翻译)(转)

作者：jaz

原文链接：https://ld246.com/article/1457572884973

来源网站：链滴

<h2>原因：</h2>
 1，coroutine容易与正常的generators弄混
 2，一个function是否为coroutine由函数体内是否有yield 或者yield from 决定，这不学。
 3，如果在语法上允许yield的地方才能进行异步调用，那诸如with和for语句中都不能执异步了。
咋解决呢，把coroutine当成一个native的Python语言特性，与generator完全独立。
Native coroutines及其新的语法使得在异步条件下定义context manager(上下文管理器)和iteration protocols（迭代器协议）成为可能（也就是with和for了）。
通过 async with语句可以使得Python程序在进入和退出runtime context(运行时上下文)时,执行步调用；
通过 async for 语句使得可以在迭代器中执行异步调用。（老外真是的，老是 make it possible）

<h2>语法定义</h2>
假定你已经知道：
 * Python中coroutines的实现。 implementation of coroutines in Python (PEP 342 and PEP 380).
 * 一些要被改变的语法来自asyncio框架和"Cofunctions"提议（已经悲剧了）。Motivation for the syntax changes proposed here comes from the asyncio framework (PEP 3156) and the "Cofunctions" proposal (PEP 3152 , now rejected in favor of this specification).
<h3>新定义的coroutine</h3>

<pre class="brush: py">async def read_data(db):
    pass</pre>

native couroutines的关键特性：
* 使用async def定义的函数总是native coroutine，无论其中是否有await表达式。
* async函数中不允许有yield和yield from，将抛出SyntaxErro异常。
* 在内部呢，引入了两个新的code object flags.
     -- CO_COROUTINE用于标记native corroutine（也就是通过async def定义的）
     -- CO_ITERABLE_COROUTINE 用于的基于 生成器的coroutine与nativ coroutines兼容。
   所有的coroutine对象都有CO_GENERATOR标准。
* generator返回generator object， coroutines 返回 coroutine object
* 没有被await on的coroutine在gc时会抛出RuntimeWarning 。
<h3>Await表达式</h3>
await表达式用于获取一个coroutine的执行结果。

<pre class="brush: py">async def read_data(db):
    data = await db.fetch('SELECT ...')
    ...</pre>

await，与yield from类似（译注；其实知道的真是不多），将阻塞read_data的执行，直到db.fech这一awaitable的完成并返回数据。
awaitable（注：主要这个awaitable是名词，不是形容词）可以是：
1， 从一个native coroutine函数返回的native coroutine object.
2, 以types.coroutine 装饰的(decorated) 生成器函数返回的generator-based coroutine objec。
3，一个对象，该对象的__await__方法返回一个迭代器。
    如果__await__返回的不是iterator，则抛出TypeError。
4，CPython的C API定义 tp_as_async->am_await函数。
    如果await出现在async def函数以外，则抛出Syntax Error；

 将awaitable对象以外的任何东西传递给await表达式都会抛出TypeError。
。。。。。。。。。
...忽略严格的语法定义部分...
。。。。。。。。。
await表达式的优先级高于**，低于切片[]、函数调用()和attribute reference(属性引用，如x.attribute），

<h3>Asynchronous Context Managers and "async with"</h3>
asynchronous context manager(异步上下文管理器)是能够在enter和exit方法中阻塞（当前coroutine）执行的上下文管理器。又增加了两个魔力函数：__aenter__ 和__aexit__ ，两个函数都必须返一个awaitable对象。
举个例子：

<pre class="brush: py">class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

async def __aexit__(self, exc_type, exc, tb):</pre>

```
    await log('exiting context')</pre>

</div>
</div>
</div>
<h4>&amp;nbsp;新的语法：</h4>
<p>提出针对异步上下文管理器的新语法定义：</p>
<div>
<div id="highlighter_659738" class="syntaxhighlighter  python ">
<div class="line number1 index0 alt2">
<pre class="brush: py">async with EXPR as VAR:
    BLOCK</pre>
</div>
</div>
</div>
<p><span>在语法上等价于:</span></p>
<div>
<div id="highlighter_127062" class="syntaxhighlighter  python">
<div class="line number1 index0 alt2">
<pre class="brush: py">mgr = (EXPR)
aexit = type(mgr).__aexit__
aenter = type(mgr).__aenter__(mgr)
exc = True

VAR = await aenter
try:
BLOCK
except:
if not await aexit(mgr, *sys.exc_info()):
raise
else:
await aexit(mgr, None, None, None)</pre>

</div>
</div>
</div>
<p>和通常的with语句一样，可以在一个await with语句中指定多个上下文管理器。</p>
<h4>Example</h4>
<p>使用异步上下文管理器可以很容易的实现用于数据库事务管理器的coroutine.&amp;nbsp;</p>
<p>With asynchronous context managers it is easy to implement proper database transactio
 managers for coroutines:</p>
<div>
<div id="highlighter_72171" class="syntaxhighlighter  python">
<div class="line number1 index0 alt2">
<pre class="brush: py">async def commit(session, data):
    ...


async with session.transaction():
    ...
```

```
    await session.update(data)
    ...</pre>

</div>
</div>
</div>
<p>需要加锁的代码变得更加清晰：Code that needs locking also looks lighter:</p>
<div>
<div id="highlighter_58366" class="syntaxhighlighter  python">
<div class="line number1 index0 alt2">
<pre class="brush: py">async with lock:
    ...</pre>
</div>
</div>
</div>
<p>instead of:</p>
<div>
<div id="highlighter_338428" class="syntaxhighlighter  python">
<pre class="brush: py"> with (yield from lock):
    ...</pre>
</div>
</div>
<h3>异步迭代器和"async for"</h3>
<p><span>asynchronous iterable能够在其iter实现中调用异步代码，并且能够在其next方法中调异步代码。</span></p>
<p><span><span>* 必须实现__aiter__方法，该方法返回一个awaitable,并且该<span>awaitable</span>的结果必须 是一个<span>asynchronous iterator object。</span></span></span>An bject must implement an __aiter__ method returning an awaitable resulting in an asynchronou iterator object .</p>
<p>* asynchronous iterator object必须实现 __anext__ 成员函数，该成员函数返回 awaitable对象；</p>
<p><span>&amp;nbsp;* 为停止迭代，__anext__必须抛出<span>StopAsyncIteration 异常。</span></span></p>
<p>举个例子：</p>
<div>
<div id="highlighter_813082" class="syntaxhighlighter  python">
<div class="line number1 index0 alt2">
<pre class="brush: py">class AsyncIterable:
    async def __aiter__(self):
        return self


async def __anext__(self):
    data = await self.fetch_data()
    if data:
        return data
    else:
        raise StopAsyncIteration

async def fetch_data(self):</pre>

</div>
</div>
</div>
```

<h4>新语法</h4>
A new statement for iterating through asynchronous iterators is proposed:

<pre class="brush: py">async for TARGET in ITER:
    BLOCK
else:
    BLOCK2
等价于</pre>
<pre class="brush: py">iter = (ITER)
iter = await type(iter).__aiter__(iter)
running = True
while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        BLOCK
else:
    BLOCK2</pre>

如果用于async for的迭代器没有__aiter__ 成员函数，将抛出TypeError；
如果在async def函数以外使用async for将抛出SyntaxError错误。
如同通常的 for语句，async for也有可选的else子句。