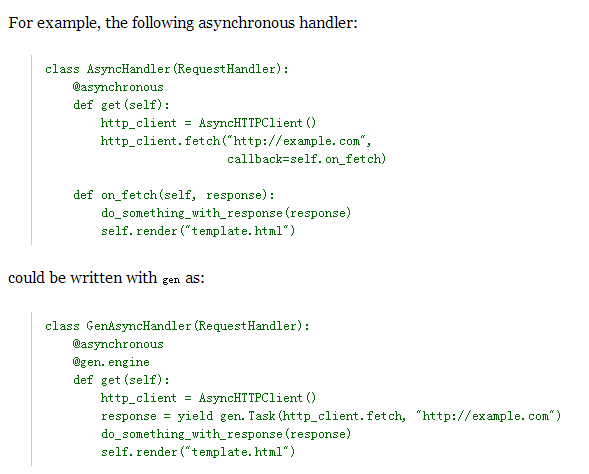
# tornado.gen

— Simplify asynchronous code

**注意：针对tornado是2.4.1版本 以下代码有删减，具体见源代码**

## 例子

使用 装饰器gen.engine 和 类gen.Task 封装异步回调



## 装饰器@asynchronous

Tornado/web.py

用@*asynchronous* 装饰的请求方法 不会立即返回Http请求，只有当处理对象调用self.finish()的时候，才返回结果。

其现很简单，就设置了一下 *self*.\_auto\_finish 的值，让其不直接返回结果。

def **asynchronous**(method):

*@functools.wraps*(method)

def **wrapper**(*self*, \*args, \*\*kwargs):

if *self*.application.\_wsgi:

raise Exception(*"@asynchronous is not supported for WSGI apps"*)

***self*.\_auto\_finish = False**

with stack\_context.ExceptionStackContext(

*self*.\_stack\_context\_handle\_exception):

return method(*self*, \*args, \*\*kwargs)

return wrapper

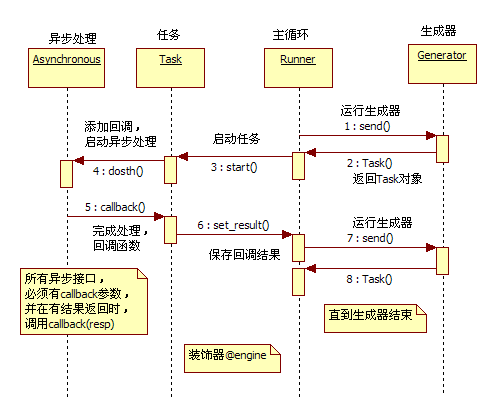
## 装饰器@engine

*tornado/gen.py*

基于生成器接口，可以让异步回调分片代码 合成在一个生成器函数里面。

要求yield Task对象

### 实现逻辑



### gen.Task 类

class **Task**(YieldPoint):

def **\_\_init\_\_**(*self*, func, \*args, \*\*kwargs):

assert *"callback"* not in kwargs

*self*.args = args

*self*.kwargs = kwargs

*self*.func = func

def **start**(*self*, runner):

*self*.runner = runner

*self*.key = object()

runner.register\_callback(*self*.key)

*self*.kwargs[*"callback"*] = runner.result\_callback(*self*.key)

*self*.func(\**self*.args, \*\**self*.kwargs)

上面例子 gen.Task 封装了异步函数http\_client.fetch("http://example.com")

在调用该函数之前，传人callback参数(注意，这就要求所有异步函数的回调参数名字一定要叫"callback" ) 即 调用 http\_client.fetch("http://example.com" , callback=runner \_callback)

这样 gen.Task 就可以得到 异步回调的结果了

### gen.engine

*tornado/gen.py*

def engine(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

gen = func(\*args, \*\*kwargs)

if isinstance(gen, types.GeneratorType):

runner = Runner(gen, deactivate)

runner.run()

return

# no yield, so we're done

return wrapper

调用Runner类的run方法

*Runner类*

调用生成器，直到生成器结束

注：yield\_point 即是gen.Task

*def run(self):*

*while True:*

#是否已经运行完回调---否有结果

*if not self.yield\_point.is\_ready():*

*return*

# 获取task 结果

*next = self.yield\_point.get\_result()*

*try:*

# 继续运行生成器，并把 结果(task 异步回调参数) 传给yield 返回值

*yielded = self.gen.send(next)*

*except StopIteration:*

*self.finished = True*

*return*

#如果还有yield Task , 开始运行

*self.yield\_point = yielded*

*self.yield\_point.start(self)*

上面在while True 第一个if 语句 如果Task 还没有结果（结束），会退出循环，那循环是如何恢复的？

是在*self.yield\_point.start(self)* 时候 传人的callback，当异步结果回调该callback，保存结果

*def set\_result(self, key, result):*

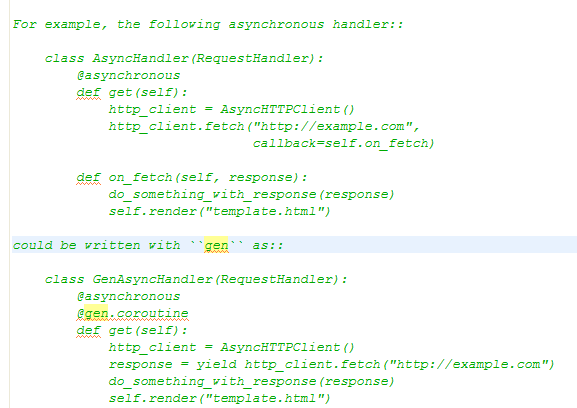
*self.results[key] = result*

*self.run()*

# tornado 3.0.1 改变

3.x 版本对新加入*gen.coroutine 装饰器*

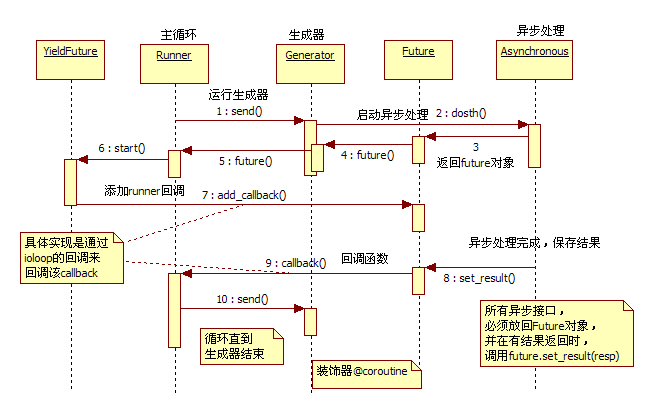
## gen.coroutine

**

和上面engine对比少了 gen.Task对象封装，直接yield.

其实现没有使用gen.Task 而是使用新对象Future

## 实现逻辑



## Future 类

tornado.concurrent.py

*Python 3.2 有futurres，如果不是使用下面对象*

if futures is None:

Future = \_DummyFuture

else:

Future = futures.Future

class **\_DummyFuture**(object):

def **\_\_init\_\_**(*self*):

*self*.\_done = False

*self*.\_result = None

*self*.\_exception = None

*self*.\_callbacks = []

def **result**(*self*, timeout=None):

*self*.\_check\_done()

if *self*.\_exception:

raise *self*.\_exception

return *self*.\_result

def **add\_done\_callback**(*self*, fn):

if *self*.\_done:

fn(*self*)

else:

*self*.\_callbacks.append(fn)

def **\_check\_done**(*self*):

if not *self*.\_done:

raise Exception(*"DummyFuture does not support blocking for results"*)

该类就是用了保存结果了回调函数的

## gen.coroutine

def **coroutine**(func):

*@functools.wraps*(func)

def **wrapper**(\*args, \*\*kwargs):

future = TracebackFuture() # 类Future 子类

if *'callback'* in kwargs: #如果有callback 把callback添加到future

callback = kwargs.pop(*'callback'*)

IOLoop.current().add\_future(

future, lambda future: callback(future.result()))

result = func(\*args, \*\*kwargs)

if isinstance(result, types.GeneratorType):

def **final\_callback**(value):

deactivate()

**future.set\_result(value)**

runner = Runner(result, final\_callback)

runner.run()

return future

future.set\_result(result)

return future

return wrapper

**与engine 不同，其返回值是future对象，（如果有回调函数调用回调函数）。**

## gen.YieldFuture

**engine的基于Task对象和异步函数回调callback来与Runner之间函数切换调用，进而实现协程**

**而coroutine 是基于YieldFuture 与future对象异步回调函数返回的对象（也就是要求异步函数要返回future对象）**

class **YieldFuture**(YieldPoint):

def **\_\_init\_\_**(*self*, future, io\_loop=None):

*self*.future = future

*self*.io\_loop = io\_loop or IOLoop.current()

def **start**(*self*, runner):

*self*.runner = runner

*self*.key = object()

runner.register\_callback(*self*.key)

#添加future完成回调

*self*.io\_loop.add\_future(*self*.future, runner.result\_callback(*self*.key))

Runner对象和上面的处理逻辑差不多，只不过添加了**YieldFuture 的支持，而不是仅仅是Task**