理解 Python 异步网络编程

- 视频后端 Wangningning

小问题?

- 什么是异步网络编程? 为什么它能提高 IO 性能
- Python的异步编程是如何工作的?
- 为什么使用协程就能用同步的方式编写异步代码

从一个 tcp socket 回显服务器说起

- 从一个简单的 TCP client server 引入
- 实现一个简单的回显服务器

In [ ]:

```
# Echo client program, code_demo/python_async/tcp_echo_client.py
import socket

HOST = '127.0.0.1'    # The remote host
PORT = 8888              # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

In [ ]:

```
# Echo server program, code_demo/python_async/tcp_echo_server.py
import socket

HOST = 'localhost'    # The remote host
PORT = 8888 # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((HOST, PORT))
    s.listen(50)
    while True:
        conn, addr = s.accept()
        print('Connected by', addr)
        with conn:
            while 1:
                data = conn.recv(1024)
                if not data:
                    break
                conn.sendall(data)
```

同步 socket 服务器的缺点

- 阻塞的
  - socket系统调用 accept/recv/sendall 等需要等待返回
- 同步的
  - client 发送请求后需要等待内核 IO 操作完成后才能继续执行，后续客户端需要等待

引入 Linux IO 多路复用机制

- Linux select/poll/epoll
- python2 select 模块
- python3 selectors 模块

Python3 selectors 模块

- 封装了操作系统的 IO 复用机制
- 替代 python2 偏底层的 select 模块
- 提供了更好的抽象和更简洁的 API

selectors 模块

- 事件类型：EVENT_READ, EVENT_WRITE
- DefaultSelector: 自动根据平台选取合适的 IO 模型
  - register(fileobj, events, data=None)
  - unregister(fileobj)
  - modify(fileobj, events, data=None)
  - select(timeout=None): returns[(key, events)]
  - close()

In [ ]:

```
# selectors 简单使用方式, code_demo/python_async/tcp_echo_server_callback.py
"""
python3 selectos 模块演示, 一个简单的异步 tcp 回显服务器
"""
import selectors
import socket

sel = selectors.DefaultSelector()  # 定义一个 selector


def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False) # 注意这里的 setblocking 为 False
    sel.register(conn, selectors.EVENT_READ, read)    # 注册监听可读事件的回调


def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)  # 取消对 conn socket 的事件监听
        conn.close()


sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)    # sock 可读的时候执行 accept 回调

while True:
    events = sel.select()    # 等待直到监听的socket 有注册的事件发生
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

什么是事件循环 EventLoop

- 异步编程中经常提到事件循环(EventLoop)的概念
- 其实刚才代码中的 while True 里就是事件循环
- 在一个死循环里等待 selector.select() 方法就绪
- 然后执行对应 socket 上注册的回调函数
- 抽象出 EventLoop 类

In [ ]:

```
import selectors

class EventLoop:

    def __init__(self, selector=None):
        if selector is None:
            selector = selectors.DefaultSelector()
        self.selector = selector

    def run_forever(self):
        while True:
            events = self.selector.select()
            for key, mask in events:
                if mask == selectors.EVENT_READ:
                    callback = key.data
                    callback(key.fileobj)
                else:
                    callback, msg = key.data
                    callback(key.fileobj, msg)
```

改写 tcp echo server

- 使用 selectors 模块改写 tcp 回显服务器
- 将 while True 循环改成 EventLoop 类

In [ ]:

```
# 基于回调方式的 tcp 回显 server
class TCPEchoServer:
    def __init__(self, host, port, loop):
        self.host = host
        self.port = port
        self._loop = loop
        self.s = socket.socket()

    def run(self):
        self.s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.s.bind((self.host, self.port))
        self.s.listen(128)
        self.s.setblocking(False)
        self._loop.selector.register(self.s, selectors.EVENT_READ, self._accept)
        self._loop.run_forever()

    def _accept(self, sock):
        conn, addr = sock.accept()
        print('accepted', conn, 'from', addr)
        conn.setblocking(False)
        self._loop.selector.register(conn, selectors.EVENT_READ, self._on_read)

    def _on_read(self, conn):
        msg = conn.recv(1024)
        if msg:
            print('echoing', repr(msg), 'to', conn)
            self._loop.selector.modify(conn, selectors.EVENT_WRITE, (self._on_wr
ite, msg))
        else:
            print('closing', conn)
            self._loop.selector.unregister(conn)
            conn.close()

    def _on_write(self, conn, msg):
        conn.sendall(msg)
        self._loop.selector.modify(conn, selectors.EVENT_READ, self._on_read)
```

In [ ]:

```
# 启动这个使用回调方式的异步 tcp 回显服务器, code_demo/python_async/tcp_echo_server_ca
llback_eventloop.py
event_loop = EventLoop()
echo_server = TCPEchoServer('localhost', 8888, event_loop)
echo_server.run()
```

回调的问题

- 代码逻辑割裂，不容易理解
- 多层嵌套，callback hell

回调的解决方式-使用协程

- 从 Python 生成器引入协程
- 什么是基于生成器的协程
- 如何使用异步编程中的 Future/Task
- 什么是原生协程

从生成器说起

- Python中生成器是用来生成值的函数 (包含 yield 的函数)
- 通常函数使用return返回值然后作用域被销毁，再次调用函数会重新执行
- 生成器可以yield一个值之后暂停函数执行，然后控制权交给调用者，之后我们可以恢复其执行并且获取下一个值

In [ ]:

```
# 生成器 generator 演示, generator_demo.py
def simple_gen():
    yield 'hello'
    yield 'world'

gen = simple_gen()
print(type(gen))    # 'generator' object
print(next(gen))    # 'hello'
print(next(gen))    # 'world'
print(next(gen))    # 协程结束抛出 StopIteration 异常
```

In [ ]:

```
# 注意生成器函数调用的时候不会直接返回值，而是返回一个类似于可迭代对象(iterable)的生成器对象(generator object)，我们可以对生成器对象调用next()函数来迭代值，或者使用for循环。
# 生成器常用来节省内存，比如我们可以使用生成器函数yield值来替代返回一个耗费内存的大序列：
def f(n):
    res = []
    for i in range(n):
        res.append(i)
    return res

def yield_n(n):
    for i in range(n):
        yield i

assert f(10) == list(yield_n(10))
```

什么是基于生成器的协程

- pep 342(Coroutines via Enhanced Generators)对生成器做了增强
- yield 关键字既可以用来获取(pull)数据，作为表达式在等号右边也可以发送数据（使用 send 方法）
- 还可以通过throw()向生成器内抛出异常以便随时终止生成器的运行。

In [ ]:

```
# 我们先看一个简单的例子, generator_based_coroutine.py
def coro():
    hello = yield 'hello'    # yield关键字在=右边作为表达式, 可以被send值
    yield hello


c = coro()
print(next(c))    # 输出 'hello', 这里调用 next 产出第一个值 'hello', 之后函数暂停
print(c.send('world'))    # 再次调用 send 发送值, 此时 hello 变量赋值为 'world', 然后
 yield 产出 hello 变量的值 'world'
# 之后协程结束, 后续再 send 值会抛异常 StopIteration
```

基于生成器的协程关键概念

- 协程需要使用 send(None) 或者 next(coroutine) 来『预激』(prime) 才能启动
- 在 yield 处协程会暂停执行
- 单独的 yield value 会产出值给调用方
- 可以通过 coroutine.send(value) 来给协程发送值，发送的值会赋值给 yield 表达式左边的变量 value = yield
- 协程执行完成后(没有遇到下一个 yield 语句)会抛出 StopIteration 异常

协程示例

- c=yield a+b，需要注意的是并不是把 a+b 的结果赋值 c

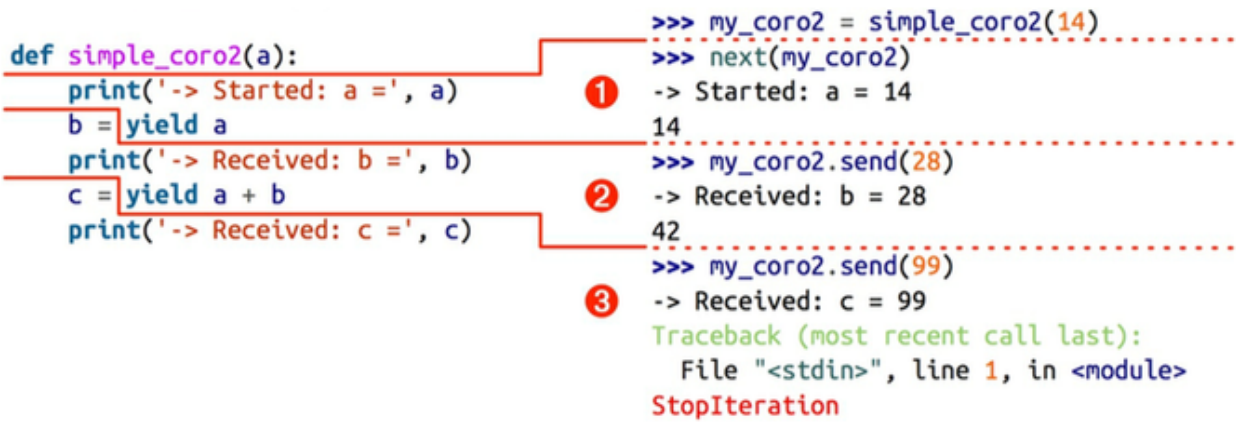

图 16-1：执行 **simple_coro2** 协程的 3 个阶段（注意，各个阶段都在 **yield** 表达式中结束，而且下一个阶段都从那一行代码开始，然后再把 **yield** 表达式的值赋给变量）

预激(prime)协程

```
from functools import wraps
def coroutine(func):  # 这样就不用每次都用 send(None)启动了
"""装饰器：向前执行到第一个`yield`表达式，预激`func`"""
    @wraps(func)
    def primer(*args,**kwargs):  ❶
        gen = func(*args,**kwargs)  ❷
        next(gen)  ❸
        return gen  ❹
    return primer
```

Python3 yield from

- Python3 引入了 yield from ， 它的主要作用有两个
  1. 链接子生成器
  2. 作为委派生成器用来当调用者和子生成器的通道

In [ ]:

```
>>> def gen():
...     for c in 'AB':
...         yield c
...     for i in range(1, 3):
...         yield i
...
>>> list(gen())
['A', 'B', 1, 2]

>>> def gen():
...     yield from 'AB'   # 用 yield from 帮我们省去了很多 for 模板代码
...     yield from range(1, 3)
...
>>> list(gen())
['A', 'B', 1, 2]
```

In [12]:

```
# 作为委派生成器
# “把迭代器当作生成器使用，相当于把子生成器的定义体内联在 yield from 表达式中。此外，子生成器可以执行 return 语句，返回一个值，而返回的值会成为 yield from 表达式的值。”
# code_demo/python_async/yield_from_demo.py
def coro1():
    """定义一个简单的基于生成器的协程作为子生成器"""
    word = yield 'hello'
    yield word
    return word    # 注意这里协程可以返回值了，返回的值会被塞到 StopIteration value 属性作为 yield from 表达式的返回值


def coro2():
    """委派生成器，起到了调用方和子生成器通道的作用，请仔细理解下边的描述。
    委派生成器会在 yield from 表达式处暂停，调用方可以直接发数据发给子生成器，
    子生成器再把产出的值发给调用方。
    子生成器返回后， 解释器抛出 StopIteration异常， 并把返回值附加到异常对象上，此时委派生成器恢复
    """
    # 子生成器返回后，解释器抛出 StopIteration 异常，返回值被附加到异常对象上，此时委派生成器恢复
    result = yield from coro1()  # 这里 coro2 会暂停并把调用者的 send 发送给 coro1() 协程，coro1() 返回后其return 的值会被赋值给 result
    print('coro2 result', result)


def main():  # 调用方，用来演示调用方通过委派生成器可以直接发送值给子生成器值。这里main 是调用者，coro2 是委派生成器，coro1 是子生成器
    c2 = coro2()  # 委派生成器
    print(next(c2))   # 这里虽然调用的是 c2 的send，但是会发送给 coro1，委派生成器进入 coro1 执行到第一个 yield 'hello' 产出 'hello'
    print(c2.send('world')) # 委派生成器发送给 coro1，word 赋值为 'world'，之后产出 'world'
    try:
        # 继续 send 由于 coro1 已经没有 yield 语句了，直接执行到了 return 并且抛出 StopIteration
        # 同时返回的结果作为 yield from 表达式的值赋值给左边的 result，接着 coro2() 里输出 "coro2 result world"
        c2.send(None)
    except StopIteration:
        pass

main()
```

```
  File "<ipython-input-12-1bf512329117>", line 18
    result = yield from coro1()  # 这里 coro2 会暂停并把调用者的 send 发送给 coro1() 协程，coro1() 返回后其return 的值会被赋值给 result
                         ^
SyntaxError: invalid syntax
```

In [ ]:

```
# RESULT = yield from EXPR 伪代码演示(为了简化，去掉了异常处理)
_i = iter(EXPR)  # ❶ EXPR 可以是任何可迭代的对象，因为获取迭代器 _i（这是子生成器）使用的是 iter() 函数。

try:
    _y = next(_i)  # ❷ 预激子生成器；结果保存在 _y 中，作为产出的第一个值。

except StopIteration as _e:
    _r = _e.value  # ❸ 如果抛出 StopIteration 异常，获取异常对象的 value 属性，赋值给 _r——这是最简单情况下的返回值（RESULT）。
else:
    while 1:  # ❹ 运行这个循环时，委派生成器会阻塞，只作为调用方和子生成器之间的通道。
        _s = yield _y  # ❺ 产出子生成器当前产出的元素；等待调用方发送 _s 中保存的值。注意，这个代码清单中只有这一个 yield 表达式。
        try:
            _y = _i.send(_s)  # ❻ 尝试让子生成器向前执行，转发调用方发送的 _s。
        except StopIteration as _e:  # ❼ 如果子生成器抛出 StopIteration 异常，获取 value 属性的值，赋值给 _r，然后退出循环，让委派生成器恢复运行。
            _r = _e.value
            break

RESULT = _r  # ❽ 返回的结果（RESULT）是 _r，即整个 yield from 表达式的值。

"""
_i（迭代器）

    子生成器

_y（产出的值）

    子生成器产出的值

_r（结果）

    最终的结果（即子生成器运行结束后 yield from 表达式的值）

_s（发送的值）

    调用方发给委派生成器的值，这个值会转发给子生成器

_e（异常）

    异常对象（在这段简化的伪代码中始终是 StopIteration 实例）

"""
```

In [ ]:

```
# 使用 yield from 改写之后的 TCPEchoServe 主体如下
class TCPEchoServer:
    def __init__(self, host, port, loop):
        self.host = host
        self.port = port
        self._loop = loop
        self.s = socket.socket()

    def run(self):
        self.s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.s.bind((self.host, self.port))
        self.s.listen(128)
        self.s.setblocking(False)

        while True:
            conn, addr = yield from self.accept()
            msg = yield from self.read(conn)
            if msg:
                yield from self.sendall(conn, msg)
            else:
                conn.close()
```

异步编程 Future 对象

- 如果不用回调，如何获取到异步调用的结果呢?
- Python异步框架中使用到了 Future 对象
- 作用：当异步调用执行完的时候，用来保存它的结果
- Future 对象的 result 属性用来保存结果
- set_result 用来设置 result 并且运行给 Future 对象添加的回调

In [ ]:

```
# Future 对象的定义
class Future:
    def __init__(self):
        self.result = None   # 保存结果
        self._callbacks = []  # 保存对 Future 的回调函数

    def add_done_callback(self, fn):
        self._callbacks.append(fn)

    def set_result(self, result):
        self.result = result
        for callback in self._callbacks:
            callback(self)

    def __iter__(self):
        """ 让 Future 对象支持 yield from"""
        yield self  # 产出自己
        return self.result   # yield from 将把 result 值返回作为 yield from 表达式的
值
```

In [ ]:

```
# 先来看个回调的例子，future_demo.py
# 这个例子中使用了多个嵌套回调， callback3 依赖 callback2 的结果，callback2 又依赖 callback1 的结果。
def callback1(a, b):
    c = a + b
    c = callback2(c)
    return c


def callback2(c):
    c *= 2
    callback3(c)
    return c


def callback3(c):
    print(c)


def caller(a, b):
    callback1(a, b)


caller(1, 2)  # 输出 6
```

In [ ]:

```
# 使用 yield from 和 Future 对象改写上面的例子
def callback_1(a, b):
    f = Future()

    def on_callback_1():
        f.set_result(a+b)

    on_callback_1()
    c = yield from f
    return c


def callback_2(c):
    f = Future()

    def on_callback_2():
        f.set_result(c*2)
    on_callback_2()
    c = yield from f
    return c


def callback_3(c):
    f = Future()

    def on_callback_3():
        f.set_result(c)
    on_callback_3()
    yield from f


def caller_use_yield_from(a, b):
    c1 = yield from callback_1(a, b)
    c2 = yield from callback_2(c1)
    yield from callback_3(c2)
    return c2
```

In [ ]:

```
# 然后你再执行以下 caller_use_yield_from(1, 2)，你会发现没有任何输出，直接调用它并没什么用
# 因为这个时候有了 yield from语句它成为了协程。
# 那我们怎么执行它呢？协程需要调用方来驱动执行，还记得我们之前说的 预激(prime) 吗？

c = caller_use_yield_from(1,2)  # coroutine
f1 = c.send(None)   # 产出第一个 future 对象
f2 = c.send(f1.result)  # 驱动运行到第二个 callback
f3 = c.send(f2.result)
try:
    f4 = c.send(None)
except StopIteration as e:
    print(e.value)   # 输出结果 6
```

In [ ]:

```
# 或者我们还可以用这种方式不断驱动它来执行，（后边我们会看到如何将它演变为 Task 类）：
# code_demo/python_async/future_demo.py

c = caller_use_yield_from(1, 2)  # coroutine
f = Future()
f.set_result(None)
next_future = c.send(f.result)
def step(future):
    next_future = c.send(future.result)
    next_future.add_done_callback(step)
while 1:
    try:
        step(f)
    except StopIteration as e:
        print(e.value)   # 输出结果 6
        break
```

协程编程的问题

- 代码看起来更复杂，不过在委派生成器这里，我们用协程和 Future 结合把回调给消除了
- 当一个函数的结果依赖另一个函数的时候，我们不需要一个函数回调另一个函数，而是通过协程把上一个协程的结果send(value)发送给下一个依赖它的值的协程
- 异步编程更加复杂，但是复杂性被框架掩盖了，业务层代码会大大简化

用协程和 yield from 改造 TCPEchoServer

- 使用 yield from 和 Future 重写
- 改写 accept/read/sendall 函数

In [ ]:

```
    def accept(self):
        f = Future()

        def on_accept():
            conn, addr = self.s.accept()
            print('accepted', conn, 'from', addr)
            conn.setblocking(False)
            f.set_result((conn, addr))  # accept 的 result 是接受连接的新对象 conn,
 addr
        self._loop.selector.register(self.s, selectors.EVENT_READ, on_accept)
        conn, addr = yield from f  # 委派给 future 对象，直到 future 执行了 socket.a
ccept() 并且把 result 返回
        self._loop.selector.unregister(self.s)
        return conn, addr

    def read(self, conn):
        f = Future()

        def on_read():
            msg = conn.recv(1024)
            f.set_result(msg)
        self._loop.selector.register(conn, selectors.EVENT_READ, on_read)
        msg = yield from f
        return msg

    def sendall(self, conn, msg):
        f = Future()

        def on_write():
            conn.sendall(msg)
            f.set_result(None)
            self._loop.selector.unregister(conn)
            conn.close()
        self._loop.selector.modify(conn, selectors.EVENT_WRITE, on_write)
        yield from f
```

In [ ]:

```
# 整个代码如下
class TCPEchoServer:
    def __init__(self, host, port, loop):
        self.host = host
        self.port = port
        self._loop = loop
        self.s = socket.socket()

    def run(self):   # 这是我们的应用层代码，看起来比起回调方便很多，但是内部 accept 等函数却做了很多工作
        self.s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.s.bind((self.host, self.port))
        self.s.listen(128)
        self.s.setblocking(False)

        while True:
            conn, addr = yield from self.accept()
            msg = yield from self.read(conn)
            if msg:
                yield from self.sendall(conn, msg)
            else:
                conn.close()

    def accept(self):
        f = Future()

        def on_accept():
            conn, addr = self.s.accept()
            print('accepted', conn, 'from', addr)
            conn.setblocking(False)
            f.set_result((conn, addr))  # accept 的 result 是接受连接的新对象 conn, addr
        self._loop.selector.register(self.s, selectors.EVENT_READ, on_accept)
        conn, addr = yield from f  # 委派给 future 对象，直到 future 执行了 socket.accept() 并且把 result 返回
        self._loop.selector.unregister(self.s)
        return conn, addr

    def read(self, conn):
        f = Future()

        def on_read():
            msg = conn.recv(1024)
            f.set_result(msg)
        self._loop.selector.register(conn, selectors.EVENT_READ, on_read)
        msg = yield from f
        return msg

    def sendall(self, conn, msg):
        f = Future()

        def on_write():
            conn.sendall(msg)
            f.set_result(None)
            self._loop.selector.unregister(conn)
            conn.close()
        self._loop.selector.modify(conn, selectors.EVENT_WRITE, on_write)
        yield from f
```

Task 驱动协程

- 使用 Future 和 yield from 将函数改造成了生成器协程
- 之前说过生成器需要由 send(None) 或者 next 来启动， 之后可以通过 send(value) 的方式发送值并且继续执行。

In [ ]:

```
# 前面我们曾经使用如下方式来运行 caller_use_yield_from 协程:

c = caller_use_yield_from(1, 2)  # coroutine
f = Future()
f.set_result(None)
next_future = c.send(f.result)
def step(future):
    next_future = c.send(future.result)
    next_future.add_done_callback(step)
while 1:
    try:
        step(f)
    except StopIteration as e:
        print(e.value)   # 输出结果 6
        break
```

In [ ]:

```
# 这里我们重构下这种方式，写一个 Task 对象驱动协程执行。
# 注意我们的 TCPEchoServer.run 方法已经成了协程，我们用 Task 驱动它执行。
# 我们创建一个 Task 来管理生成器的执行。

class Task:
    """管理生成器的执行"""

    def __init__(self, coro):
        self.coro = coro

    def step(self, future):
        next_future = self.coro.send(future.result)
        next_future.add_done_callback(self.step)

    def run(self):
        f = Future()
        f.set_result(None)
        while 1:
            try:
                self.step(f)
            except StopIteration as e:
                print(e.value)
                return

# 然后我们可以用如下方式使用它:
Task(caller_use_yield_from(1,2)).run()
```

In [ ]:

```
# 不过对于 TCPEchoServer 我们需要事件循环来驱动它，我们把 Task 修改成如下形式：
class Task:
    """管理生成器的执行"""
    def __init__(self, coro):
        self.coro = coro
        f = Future()
        f.set_result(None)
        self.step(f)

    def step(self, future):
        try:  # 把当前 future 的结果发送给协程作为 yield from 表达式的值，同时执行到下一
个 future 处
            next_future = self.coro.send(future.result)
        except StopIteration:
            return
        next_future.add_done_callback(self.step)
```

In [ ]:

```
# 然后是我们的 EventLoop 事件循环类：


class EventLoop:
    def __init__(self, selector=None):
        if selector is None:
            selector = selectors.DefaultSelector()
        self.selector = selector

    def create_task(self, coro):
        return Task(coro)

    def run_forever(self):
        while 1:
            events = self.selector.select()
            for event_key, event_mask in events:
                callback = event_key.data
                callback()
```

In [ ]:

```
# 好了，最后我们来启动 TCPEchoServer，

event_loop = EventLoop()
echo_server = TCPEchoServer('localhost', 8888, event_loop)
task = Task(echo_server.run())
event_loop.run_forever()
```

In [ ]:

```
# 所有代码如下：code_demo/python_async/tcp_echo_server_coroutine.py

import selectors
import socket


class Future:
    def __init__(self):
        self.result = None
        self._callbacks = []

    def add_done_callback(self, fn):
        self._callbacks.append(fn)

    def set_result(self, result):
        self.result = result
        for callback in self._callbacks:
            callback(self)

    def __iter__(self):
        yield self
        return self.result


class Task:
    """管理生成器的执行"""
    def __init__(self, coro):
        self.coro = coro
        f = Future()
        f.set_result(None)
        self.step(f)

    def step(self, future):
        try:  # 把当前 future 的结果发送给协程作为 yield from 表达式的值，同时执行到下一个 future 处
            next_future = self.coro.send(future.result)
        except StopIteration:
            return
        next_future.add_done_callback(self.step)


class TCPEchoServer:
    def __init__(self, host, port, loop):
        self.host = host
        self.port = port
        self._loop = loop
        self.s = socket.socket()

    def run(self):
        self.s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.s.bind((self.host, self.port))
        self.s.listen(128)
        self.s.setblocking(False)

        while True:
            conn, addr = yield from self.accept()
            msg = yield from self.read(conn)
            if msg:
                yield from self.sendall(conn, msg)
```

```
            else:
                conn.close()

    def accept(self):
        f = Future()

        def on_accept():
            conn, addr = self.s.accept()
            print('accepted', conn, 'from', addr)
            conn.setblocking(False)
            f.set_result((conn, addr))  # accept 的 result 是接受连接的新对象 conn, addr
        self._loop.selector.register(self.s, selectors.EVENT_READ, on_accept)
        conn, addr = yield from f  # 委派给 future 对象，直到 future 执行了 socket.accept() 并且把 result 返回
        self._loop.selector.unregister(self.s)
        return conn, addr

    def read(self, conn):
        f = Future()

        def on_read():
            msg = conn.recv(1024)
            f.set_result(msg)
        self._loop.selector.register(conn, selectors.EVENT_READ, on_read)
        msg = yield from f
        return msg

    def sendall(self, conn, msg):
        f = Future()

        def on_write():
            conn.sendall(msg)
            f.set_result(None)
            self._loop.selector.unregister(conn)
            conn.close()
        self._loop.selector.modify(conn, selectors.EVENT_WRITE, on_write)
        yield from f


class EventLoop:
    def __init__(self, selector=None):
        if selector is None:
            selector = selectors.DefaultSelector()
        self.selector = selector

    def create_task(self, coro):
        return Task(coro)

    def run_forever(self):
        while 1:
            events = self.selector.select()
            for event_key, event_mask in events:
                callback = event_key.data
                callback()


event_loop = EventLoop()
echo_server = TCPEchoServer('localhost', 8888, event_loop)
task = Task(echo_server.run())
event_loop.run_forever()
```

原生协程, async/await

- 到目前为止，我们仍然使用的是 基于生成器的协程(generators based coroutines)
- python3.5中，python增加了使用async/await语法的原生协程(native coroutines)，使用起来并没有功能上的差别
- 我们把之前的所有 yield from 改成 await，同时函数定义前面加上 async 就好了
- 注意 Future 需要定义 `__await__` 方法

In [ ]:

```
# code_demo/python_async/tcp_echo_server_async_await.py
class Future:
    def __init__(self):
        self.result = None
        self._callbacks = []

    def add_done_callback(self, fn):
        self._callbacks.append(fn)

    def set_result(self, result):
        self.result = result
        for callback in self._callbacks:
            callback(self)

    def __iter__(self):
        yield self
        return self.result

    __await__ = __iter__ # make compatible with 'await' expression

class Task:
    def __init__(self, coro):
        self.coro = coro
        f = Future()
        f.set_result(None)
        self.step(f)

    def step(self, future):
        try:
            next_future = self.coro.send(future.result)
        except StopIteration:
            return
        next_future.add_done_callback(self.step)


class TCPEchoServer:
    def __init__(self, host, port, loop):
        self.host = host
        self.port = port
        self._loop = loop
        self.s = socket.socket()

    async def run(self):
        self.s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.s.bind((self.host, self.port))
        self.s.listen(128)
        self.s.setblocking(False)

        while True:
            conn, addr = await self.accept()
            msg = await self.read(conn)
            if msg:
                await self.sendall(conn, msg)
            else:
                conn.close()

    async def accept(self):
        f = Future()
```

```
        def on_accept():
            conn, addr = self.s.accept()
            print('accepted', conn, 'from', addr)
            conn.setblocking(False)
            f.set_result((conn, addr))
        self._loop.selector.register(self.s, selectors.EVENT_READ, on_accept)
        conn, addr = await f
        self._loop.selector.unregister(self.s)
        return conn, addr

    async def read(self, conn):
        f = Future()

        def on_read():
            msg = conn.recv(1024)
            f.set_result(msg)
        self._loop.selector.register(conn, selectors.EVENT_READ, on_read)
        msg = await f
        return msg

    async def sendall(self, conn, msg):
        f = Future()

        def on_write():
            conn.sendall(msg)
            f.set_result(None)
            self._loop.selector.unregister(conn)
            conn.close()
        self._loop.selector.modify(conn, selectors.EVENT_WRITE, on_write)
        await f


class EventLoop:
    def __init__(self, selector=None):
        if selector is None:
            selector = selectors.DefaultSelector()
        self.selector = selector

    def create_task(self, coro):
        return Task(coro)

    def run_forever(self):
        while 1:
            events = self.selector.select()
            for event_key, event_mask in events:
                callback = event_key.data
                callback()


event_loop = EventLoop()
echo_server = TCPEchoServer('localhost', 8888, event_loop)
task = Task(echo_server.run())
event_loop.run_forever()
```

参考

- 《Fluent Python》
- 深入理解 Python 异步编程 (https://mp.weixin.qq.com/s/GgamzHPyZuSg45LoJKsofA)
- 从 asyncio 简单实现看异步是如何工作的 (https://www.4async.com/2016/02/simple-implement-asyncio-to-understand-how-async-works/)
- Python generators, coroutines, native coroutines and async/await ()

Thanks