# IOLoop模块

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

这个模块是异步机制的核心。

实现代码

tornado/ioloop.py

## poll选择

ioloop 关心三种类型的IO事件：READ，WRITE 和 ERROR。ERROR是默认自动添加的。

IO事件的监听使用epoll 、kqueue 、select 等（优先使用epoll）

代码实现

if hasattr(select, *"epoll"*):

# Python 2.6+ on Linux

\_poll = select.epoll

elif hasattr(select, *"kqueue"*):

# Python 2.6+ on BSD or Mac

\_poll = \_KQueue

else:

try:

# Linux systems with our C module installed

from tornado import epoll

\_poll = \_EPoll

except Exception:

# All other systems

import sys

if *"linux"* in sys.platform:

logging.warning(*"epoll module not found; using select()"*)

\_poll = \_Select

## 主循环：

ioloop 主循环逻辑

1、处理回调

2、处理定时器

3、处理io事件，回调处理函数

IOLoop.start

def start(self):

self.\_running = True

while True:

poll\_timeout = 3600.0

#处理回调函数

with self.\_callback\_lock:

callbacks = self.\_callbacks

self.\_callbacks = []

for callback in callbacks:

self.\_run\_callback(callback)

#处理定时器

if self.\_timeouts: # 定时器使用堆实现

now = time.time()

while self.\_timeouts:

if self.\_timeouts[0].callback is None:

heapq.heappop(self.\_timeouts)

elif self.\_timeouts[0].deadline <= now:

timeout = heapq.heappop(self.\_timeouts)

self.\_run\_callback(timeout.callback)

else:

seconds = self.\_timeouts[0].deadline - now

poll\_timeout = min(seconds, poll\_timeout) 更改poll时间

break

if self.\_callbacks:

poll\_timeout = 0.0 #还有回调，不等待

#控制循环退出

if not self.\_running:

break

。。。。

#获取IO信息

try:

event\_pairs = self.\_impl.poll(poll\_timeout)

except Exception, e:

if (getattr(e, 'errno', None) == errno.EINTR or

(isinstance(getattr(e, 'args', None), tuple) and

len(e.args) == 2 and e.args[0] == errno.EINTR)):

continue

else:

raise

。。。。

#处理IO事件

self.\_events.update(event\_pairs)

while self.\_events:

fd, events = self.\_events.popitem()

try:

self.\_handlers[fd](fd, events)

except (OSError, IOError), e:

if e.args[0] == errno.EPIPE:

pass

else:

logging.error("Exception in I/O handler for fd %s", fd, exc\_info=True)

except Exception:

logging.error("Exception in I/O handler for fd %s", fd, exc\_info=True)

## 主要对外提供的接口：

### 1、socketIO 事件处理——添加、更新、删除

def **add\_handler**(*self*, fd, handler, events):

*self*.\_handlers[fd] = stack\_context.wrap(handler)

*self*.\_impl.register(fd, events | *self*.ERROR)

def **update\_handler**(*self*, fd, events):

*self*.\_impl.modify(fd, events | *self*.ERROR)

def **remove\_handler**(*self*, fd):

*self*.\_handlers.pop(fd, None)

*self*.\_events.pop(fd, None)

try:

*self*.\_impl.unregister(fd)

except (OSError, IOError):

logging.debug(*"Error deleting fd from IOLoop"*, exc\_info=True)

### 2、添加回调

def **add\_callback**(*self*, callback):

*"""添加回调 在下一轮 I/O loop 迭代中 .他是线程安全的*

*add\_callback() may be used to transfer*

*control from other threads to the IOLoop's thread.*

*"""*

with *self*.\_callback\_lock:

list\_empty = not *self*.\_callbacks

*self*.\_callbacks.append(stack\_context.wrap(callback))

if list\_empty and thread.get\_ident() != *self*.\_thread\_ident:

# 多线程下唤醒ioloop.

*self*.\_waker.wake()

### 3、定时器——添加、删除（取消）

def **add\_timeout**(*self*, deadline, callback):

timeout = \_Timeout(deadline, stack\_context.wrap(callback))

heapq.heappush(*self*.\_timeouts, timeout)

return timeout

def **remove\_timeout**(*self*, timeout):

timeout.callback = None

## 例子

### TCP server

文档例子 Example usage for a simple TCP server::

#coding=utf-8

import socket

from tornado import ioloop

import functools

import errno

def handle\_connection(connection, address):

print address

connection.close()

def connection\_ready(sock, fd, events):

while True:

try:

connection, address = sock.accept()

except socket.error, e:

if e.args[0] not in (errno.EWOULDBLOCK, errno.EAGAIN):

raise

return

connection.setblocking(0)

handle\_connection(connection, address)

#创建socket

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM, 0)

sock.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

sock.setblocking(0)

sock.bind(("", 8888))

sock.listen(128)

#创建主循环

io\_loop = ioloop.IOLoop.instance()

callback = functools.partial(connection\_ready, sock)

#添加处理回调

io\_loop.add\_handler(sock.fileno(), callback, io\_loop.READ)

#开始循环处理事件、请求

io\_loop.start()

### 客户端：

import socket

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM, 0)

sock.connect(("",8888))

print 'connection'

sock.close()

# tornado 3.0.1 改变

3.x 版本对IOLoop 进行重构。把它拆成3个类，

Configurable 配置工厂抽象类——更改子类的具体创建生成方式

 IOLoop 变成一个抽象类

PollIOLoop 具体功能实现类

**Configurable**(object): **IOLoop**(Configurable): **PollIOLoop**(IOLoop):

## Configurable

tronado/util.py

class **Configurable**(object):

\_\_impl\_class = None

\_\_impl\_kwargs = None

def **\_\_new\_\_**(cls, \*\*kwargs): #更改类的创建方式

base = cls.configurable\_base() #配置基类

args = {}

if cls is base:

impl = cls.configured\_class() #获取实际要创建的类

if base.\_\_impl\_kwargs:

args.update(base.\_\_impl\_kwargs)

else:

impl = cls

args.update(kwargs)

instance = super(Configurable, cls).\_\_new\_\_(impl)

instance.initialize(\*\*args) #调用initialize初始化

return instance

*@classmethod*

def **configured\_class**(cls):

*"""Returns the currently configured class."""*

base = cls.configurable\_base()

if cls.\_\_impl\_class is None:

base.\_\_impl\_class = cls.configurable\_default() #默认实现

return base.\_\_impl\_class

 Configurable 子类 需要实现

`configurable\_base` 放回配置类的基类（为什么用这个，不太清楚）

`configurable\_default`, 默认创建类

`initialize` 类的初始化方法（注意不是 \_\_init\_\_）.

## IOLoop

tronado/ioloop.py

class **IOLoop**(Configurable):

*@classmethod*

def **configurable\_base**(cls):

return IOLoop

*@classmethod*

def **configurable\_default**(cls):#优先使用epoll 、kqueue

if hasattr(select, *"epoll"*):

from tornado.platform.epoll import EPollIOLoop

return EPollIOLoop

if hasattr(select, *"kqueue"*):

from tornado.platform.kqueue import KQueueIOLoop

return KQueueIOLoop

from tornado.platform.select import SelectIOLoop

return SelectIOLoop

可以看出IOLoop实际创建对象是 tornado.platform.epoll .EPollIOLoop

class **EPollIOLoop**(PollIOLoop):

def **initialize**(*self*, \*\*kwargs):

super(EPollIOLoop, *self*).initialize(impl=select.epoll(), \*\*kwargs)

## PollIOLoop

class **PollIOLoop**(IOLoop):

2.x 版本IOLoop 的实现多移动到这里实现

## IOLoop添加的一些新功能。

* 新函数[**IOLoop.current**](http://www.tornadoweb.org/en/stable/ioloop.html#tornado.ioloop.IOLoop.current)  当前线程的IOLoop 实例替换 [**IOLoop.instance**](http://www.tornadoweb.org/en/stable/ioloop.html#tornado.ioloop.IOLoop.instance) 主线程的实例,
* [**IOLoop.add\_callback**](http://www.tornadoweb.org/en/stable/ioloop.html#tornado.ioloop.IOLoop.add_callback) and [**add\_callback\_from\_signal**](http://www.tornadoweb.org/en/stable/ioloop.html#tornado.ioloop.IOLoop.add_callback_from_signal) 支持带callback 参数 \*args, \*\*kwargs .
* 新函数 [**IOLoop.add\_future**](http://www.tornadoweb.org/en/stable/ioloop.html#tornado.ioloop.IOLoop.add_future)  当[**Future**](http://www.tornadoweb.org/en/stable/concurrent.html#tornado.concurrent.Future) finishes 运行回调（添加 a callback on the IOLoop ）

def **add\_future**(*self*, future, callback):*.*

assert isinstance(future, Future)

callback = stack\_context.wrap(callback)

future.add\_done\_callback(

lambda future: *self*.add\_callback(callback, future))

详细见

<http://www.tornadoweb.org/en/stable/releases/v3.0.0.html>