# IOLoop模块

这个模块是异步机制的核心。

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

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:

# If any callbacks or timeouts called add\_callback,

# we don't want to wait in poll() before we run them.

poll\_timeout = 0.0

#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 事件处理等

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

update\_handler(self, fd, events):

remove\_handler(self, fd):

2、添加回调

add\_callback(self, callback):

3、定时器

add\_timeout(self, deadline, callback):# deadline 时间秒数或者是datetime.timedelta

remove\_timeout(self, timeout): #timeout 是一个类

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

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

*"""Calls the given callback on the next I/O loop iteration.*

*It is safe to call this method from any thread at any time.*

*Note that this is the \*only\* method in IOLoop that makes this*

*guarantee; all other interaction with the IOLoop must be done*

*from that IOLoop's thread. 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:

# If we're in the IOLoop's thread, we know it's not currently

# polling. If we're not, and we added the first callback to an

# empty list, we may need to wake it up (it may wake up on its

# own, but an occasional extra wake is harmless). Waking

# up a polling IOLoop is relatively expensive, so we try to

# avoid it when we can.

*self*.\_waker.wake()

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

*"""Registers the given handler to receive the given events for fd."""*

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

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

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

*"""Changes the events we listen for fd."""*

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

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

*"""Stop listening for events on 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)

## 例子

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()