

selectors – High-level I/O multiplexing

Introduction

This module allows high-level and efficient I/O multiplexing, built upon the select module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.

It defines a `BaseSelector` abstract base class, along with several concrete implementations (`KqueueSelector`, `EpollSelector`...), that can be used to wait for I/O readiness notification on multiple file objects. In the following, “file object” refers to any object with a `fileno()` method, or a raw file descriptor. See [file object](#).

`DefaultSelector` is an alias to the most efficient implementation available on the current platform: this should be the default choice for most users.

Classes

[Classes hierarchy](#):

`BaseSelector`

+-- `SelectSelector`

+-- `PollSelector`

+-- `EpollSelector`

+-- `DevpollSelector`

+-- `KqueueSelector`

In the following, `events` is a bitwise mask indicating which I/O events should be waited for on a given file object. It can be a combination of the modules constants below:

Constant	Meaning
<code>EVENT_READ</code>	Available for read

EVENT_WRITE	Available for write
-------------	---------------------

class selectors.SelectorKey

A SelectorKey is a namedtuple used to associate a file object to its underlying file descriptor, selected event mask and attached data. It is returned by several BaseSelector methods.

fileobj

File object registered.

fd

Underlying file descriptor.

events

Events that must be waited for on this file object.

data

Optional opaque data associated to this file object: for example, this could be used to store a per-client session ID.

class selectors.BaseSelector

A BaseSelector is used to wait for I/O event readiness on multiple file objects. It supports file stream registration, unregistration, and a method to wait for I/O events on those streams, with an optional timeout. It's an abstract base class, so cannot be instantiated. Use DefaultSelector instead, or one of SelectSelector, KqueueSelector etc. if you want to specifically use an implementation, and your platform supports it. BaseSelector and its concrete implementations support the context manager protocol.

abstractmethod register(fileobj, events, data=None)

Register a file object for selection, monitoring it for I/O events.

fileobj is the file object to monitor. It may either be an integer file descriptor or an object with a fileno() method. events is a bitwise mask of events to monitor. data is an opaque object.

This returns a new SelectorKey instance, or raises a ValueError in case of invalid event mask or file descriptor, or KeyError if the file object is already registered.

abstractmethod unregister(fileobj)

Unregister a file object from selection, removing it from monitoring. A file object shall be unregistered prior to being closed.

fileobj must be a file object previously registered.

This returns the associated SelectorKey instance, or raises a KeyError if fileobj is not registered. It will raise ValueError if fileobj is invalid (e.g. it has no fileno() method or its fileno() method has an invalid return value).

`modify(fileobj, events, data=None)`

Change a registered file object's monitored events or attached data.

This is equivalent to `BaseSelector.unregister(fileobj)()` followed by `BaseSelector.register(fileobj, events, data)()`, except that it can be implemented more efficiently.

This returns a new SelectorKey instance, or raises a ValueError in case of invalid event mask or file descriptor, or KeyError if the file object is not registered.

`abstractmethod select(timeout=None)`

Wait until some registered file objects become ready, or the timeout expires.

If `timeout > 0`, this specifies the maximum wait time, in seconds. If `timeout <= 0`, the call won't block, and will report the currently ready file objects. If `timeout` is `None`, the call will block until a monitored file object becomes ready.

This returns a list of (key, events) tuples, one for each ready file object.

key is the SelectorKey instance corresponding to a ready file object. events is a bitmask of events ready on this file object.

`close()`

Close the selector.

This must be called to make sure that any underlying resource is freed. The selector shall not be used once it has been closed.

`get_key(fileobj)`

Return the key associated with a registered file object.

This returns the SelectorKey instance associated to this file object, or raises KeyError if the file object is not registered.

`abstractmethod get_map()`

Return a mapping of file objects to selector keys.

This returns a Mapping instance mapping registered file objects to their associated SelectorKey instance.

`class selectors.DefaultSelector`

The default selector class, using the most efficient implementation available on the current platform. This should be the default choice for most users.

`class selectors.SelectSelector`

`select.select()`-based selector.

`class selectors.PollSelector`

`select.poll()`-based selector.

`class selectors.EpollSelector`

`select.epoll()`-based selector.

`fileno()`

This returns the file descriptor used by the underlying `select.epoll()` object.

`class selectors.DevpollSelector`

`select.devpoll()`-based selector.

`fileno()`

This returns the file descriptor used by the underlying `select.devpoll()` object.

`class selectors.KqueueSelector`

`select.kqueue()`-based selector.

`fileno()`

This returns the file descriptor used by the underlying `select.kqueue()` object.

Examples

Here is a simple echo server implementation:

```
import selectors
```

```
import socket
```

```
sel = selectors.DefaultSelector()
```

```
def accept(sock, mask):
```

```
    conn, addr = sock.accept() # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

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)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```