18.5.2. Event loops

Source code: Lib/asyncio/events.py

18.5.2.1. Event loop functions

The following functions are convenient shortcuts to accessing the methods of the global policy. Note that this provides access to the default policy, unless an alternative policy was set by calling `set_event_loop_policy()` earlier in the execution of the process.

`asyncio.`**`get_event_loop`**`()`

Equivalent to calling `get_event_loop_policy().get_event_loop()`.

`asyncio.`**`set_event_loop`**`(`*loop*`)`

Equivalent to calling `get_event_loop_policy().set_event_loop(loop)`.

`asyncio.`**`new_event_loop`**`()`

Equivalent to calling `get_event_loop_policy().new_event_loop()`.

18.5.2.2. Available event loops

asyncio currently provides two implementations of event loops: `SelectorEventLoop` and `ProactorEventLoop`.

class `asyncio.`**`SelectorEventLoop`**

Event loop based on the `selectors` module. Subclass of `AbstractEventLoop`.

Use the most efficient selector available on the platform.

On Windows, only sockets are supported (ex: pipes are not supported): see the MSDN documentation of select.

class `asyncio.`**`ProactorEventLoop`**

Proactor event loop for Windows using “I/O Completion Ports” aka IOCP. Subclass of `AbstractEventLoop`.

Availability: Windows.

See also: MSDN documentation on I/O Completion Ports.

Example to use a `ProactorEventLoop` on Windows:

```
import asyncio, sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

18.5.2.3. Platform support

The `asyncio` module has been designed to be portable, but each platform still has subtle differences and may not support all `asyncio` features.

18.5.2.3.1. Windows

Common limits of Windows event loops:

- `create_unix_connection()` and `create_unix_server()` are not supported: the socket family `socket.AF_UNIX` is specific to UNIX
- `add_signal_handler()` and `remove_signal_handler()` are not supported
- `EventLoopPolicy.set_child_watcher()` is not supported. `ProactorEventLoop` supports subprocesses. It has only one implementation to watch child processes, there is no need to configure it.

`SelectorEventLoop` specific limits:

- `SelectSelector` is used which only supports sockets and is limited to 512 sockets.
- `add_reader()` and `add_writer()` only accept file descriptors of sockets
- Pipes are not supported (ex: `connect_read_pipe()`, `connect_write_pipe()`)
- Subprocesses are not supported (ex: `subprocess_exec()`, `subprocess_shell()`)

`ProactorEventLoop` specific limits:

- `create_datagram_endpoint()` (UDP) is not supported
- `add_reader()` and `add_writer()` are not supported

The resolution of the monotonic clock on Windows is usually around 15.6 msec. The best resolution is 0.5 msec. The resolution depends on the hardware (availability of HPET) and on the Windows configuration. See asyncio delayed calls.

Changed in version 3.5: `ProactorEventLoop` now supports SSL.

18.5.2.3.2. Mac OS X

Character devices like PTY are only well supported since Mavericks (Mac OS 10.9). They are not supported at all on Mac OS 10.5 and older.

On Mac OS 10.6, 10.7 and 10.8, the default event loop is `SelectorEventLoop` which uses `selectors.KqueueSelector`. `selectors.KqueueSelector` does not support character devices on these versions. The `SelectorEventLoop` can be used with `SelectSelector` or `PollSelector` to support character devices on these versions of Mac OS X. Example:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

18.5.2.4. Event loop policies and the default policy

Event loop management is abstracted with a *policy* pattern, to provide maximal flexibility for custom platforms and frameworks. Throughout the execution of a process, a single global policy object manages the event loops available to the process based on the calling context. A policy is an object implementing the `AbstractEventLoopPolicy` interface.

For most users of `asyncio`, policies never have to be dealt with explicitly, since the default global policy is sufficient (see below).

The module-level functions `get_event_loop()` and `set_event_loop()` provide convenient access to event loops managed by the default policy.

18.5.2.5. Event loop policy interface

An event loop policy must implement the following interface:

class `asyncio.`**`AbstractEventLoopPolicy`**

Event loop policy.

`get_event_loop`()

Get the event loop for the current context.

Returns an event loop object implementing the `AbstractEventLoop` interface. In case called from coroutine, it returns the currently running event loop.

Raises an exception in case no event loop has been set for the current context and the current policy does not specify to create one. It must never return `None`.

Changed in version 3.6.

`set_event_loop`(*loop*)

Set the event loop for the current context to *loop*.

`new_event_loop`()

Create and return a new event loop object according to this policy's rules.

If there's need to set this loop as the event loop for the current context, `set_event_loop()` must be called explicitly.

The default policy defines context as the current thread, and manages an event loop per thread that interacts with `asyncio`. If the current thread doesn't already have an event loop associated with it, the default policy's `get_event_loop()` method creates one when called from the main thread, but raises `RuntimeError` otherwise.

18.5.2.6. Access to the global loop policy

`asyncio.`**`get_event_loop_policy`**()

Get the current event loop policy.

`asyncio.`**`set_event_loop_policy`**(*policy*)

Set the current event loop policy. If *policy* is `None`, the default policy is restored.

18.5.2.7. Customizing the event loop policy

To implement a new event loop policy, it is recommended you subclass the concrete default event loop policy `DefaultEventLoopPolicy` and override the methods for which you want to change behavior, for example:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```