asyncio_dispatch Documentation

Release 1.1.0

Mike Lenzen

November 12, 2015

Contents

aysincio_dispatch is a signal dispatcher for the asyncio event loop.

CHAPTER 1

Synopsis

Many callbacks can be connected to a Signal. When the Signal is triggered using its `send()` method, all connected callbacks will be scheduled for asynchronous execution.

Connections can optionally be made using with types of filters, `senders` and `keys`. If filters are used, the callback is only scheduled for execution if the Signal is sent with at least one matching `sender` or `key`. A `sender` can be any object, while a `key` is more likely to be a `string`. Under the hood, they use `id()` and `hash()` respectively.

Callbacks are invoked with keyword arguments that allow the callback to determine which Signal is calling it and which `senders` and `keys` were specified. Additional keyword arguments can be added to the Signal when it is instantiated, and their default values can be replaced when the Signal is sent.

Contents:

1.1 Introduction

1.1.1 Prerequisites

`asyncio_dispatch` works with the `asyncio` library found in python versions 3.4 and up.

1.1.2 Installation

You can install the most recent asyncio_dispatch release from pypi using pip:

```
pip install asyncio_dispatch
```

1.1.3 Contributions and Source

Source code is available at https://github.com/lenzenmi/asyncio_dispatch

Any contributions will be welcomed, especially those to improve testing and compatibility with the new python 3.5 asynchronous syntax, or asyncio backports like *trollius* and *tulip*.

Because the python syntax varies by python version, tests are run with tox against all supported versions.

> **Warning:** The tox command must be run with python version 3.5 or greater.

Run all tests

```
pip install tox
tox
```

Run specific tox envoronment tests

```
pip install tox
tox -e py35 # or py34
tox -e flake8
tox -e docs
```

Run all tests for the python environments you have installed on your system

```
pip install tox
tox --skip-missing-interpreters
```

1.1.4 License

Licensed under the MIT license.

1.2 Examples

The `asyncio_dispatch.Signal` class is used to connect and trigger callbacks.

Note: It's suggested to look at all of the examples, the last two are the most interesting.

1.2.1 Basic example

In this example, the callback is connected to the signal. When `Signal.send()` is called, all connected callbacks without `keys` or `senders` will be executed. Since our callback was connected without `key` or `sender` arguments, it will be run.

```
import asyncio
from asyncio_dispatch import Signal


@asyncio.coroutine
def callback(**kwargs):
    print('callback was called!')


loop = asyncio.get_event_loop()

# create the signal
signal = Signal(loop=loop)

# connect the callback to the Signal - This is a coroutine!
loop.run_until_complete(signal.connect(callback))

# send the signal - This is also a coroutine!
print('sending the signal.')
loop.run_until_complete(signal.send())
```

The above example prints the following output

```
sending the signal.
callback was called!
```

1.2.2 Basic example with async/await syntax

The same as above but with the new python 3.5 async/await syntax. The loop only runs long enough to send the signal and call the callback before exiting.

```
import asyncio
from asyncio_dispatch import Signal


async def callback(**kwargs):
    print('callback was called!')


async def connect_and_send_signal(signal, callback):
    await signal.connect(callback)
    print('sending the signal.')
    await signal.send()


loop = asyncio.get_event_loop()

# initialize the signal
signal = Signal(loop=loop)

# connect the callback to the signal and send the signal
loop.run_until_complete(connect_and_send_signal(signal, callback))
```

The above example prints the following output

```
sending the signal.
callback was called!
```

1.2.3 Mulitple types of callables

In this example, 5 different types of callable are all connected to the same signal. When the signal is sent, all 5 callbacks will be scheduled for execution.

```
import asyncio
from asyncio_dispatch import Signal


def callback1(**kwargs):
    print('callback1 was called')


@asyncio.coroutine
def callback2(**kwargs):
    print('callback2 was called')


async def callback3(**kwargs):
    print('callback3 was called')
```

```
class Test:
    def callback4(self, **kwargs):
        print('callback4 was called')

    @asyncio.coroutine
    def callback5(self, **kwargs):
        print('callback5 was called')

    async def callback6(self, **kwargs):
        print('callback6 was called')

    @classmethod
    def callback7(cls, **kwargs):
        print('callback7 was called')

    @staticmethod
    def callback8(**kwargs):
        print('callback8 was called')


loop = asyncio.get_event_loop()

# create the signal
signal = Signal(loop=loop)

# initialize the test class
test = Test()

tasks = [
    # connect the function and coroutines
    loop.create_task(signal.connect(callback1)),
    loop.create_task(signal.connect(callback2)),
    loop.create_task(signal.connect(callback3)),

    # connect the class methods
    loop.create_task(signal.connect(test.callback4)),
    loop.create_task(signal.connect(test.callback5)),
    loop.create_task(signal.connect(test.callback6)),
    loop.create_task(signal.connect(Test.callback7)),
    loop.create_task(signal.connect(Test.callback8)),
]

loop.run_until_complete(asyncio.wait(tasks))

# send the signal
loop.run_until_complete(signal.send())
```

The above example prints the following output

```
callback6 was called
callback1 was called
callback2 was called
callback8 was called
callback4 was called
callback3 was called
callback5 was called
callback7 was called
```

1.2.4 Example with kwargs

Callbacks receive several kwargs when called. The default keyword arguments are `signal`, `senders`, and `keys`. `signal` is the signal that called the callback. `senders` and `keys` each return a set containing all of the `senders` and `keys` specified when calling `asyncio_dispatch.Signal.send()`.

You can also add your own custom keyword arguments to a signal when it is instantiated. Each additional kwarg added to the signal has a default value. The value of the additional kwargs can be changed when the signal is sent.

```
import asyncio
from asyncio_dispatch import Signal


def callback(signal, senders, keys, my_kwarg, payload):
    print()
    print('-' * 50)

    if SIGNAL is signal:
        print('signals match as expected!')

    print('senders=', senders)
    print('keys=', keys)
    print('my_kwarg= {}'.format(my_kwarg))
    print('payload= {}'.format(payload))


loop = asyncio.get_event_loop()

# create the signal and define two custom keyword arguments: my_kwarg and payload
SIGNAL = Signal(loop=loop, my_kwarg='default', payload={})

# connect the callback to the signal - this method is a coroutine
loop.run_until_complete(SIGNAL.connect(callback))

# send the signal with default keyword arguments - this method is also a coroutine
loop.run_until_complete(SIGNAL.send())

# send the signal again with new values for my_kwarg and payload
loop.run_until_complete(SIGNAL.send(my_kwarg='changed with send',
                                    payload={'anything': 'a dict can hold!',
                                             'really': 'powerfull'}))
```

The above example prints the following output

```
--------------------------------------------------
signals match as expected!
senders= set()
keys= set()
my_kwarg= default
payload= {}

--------------------------------------------------
signals match as expected!
senders= set()
keys= set()
my_kwarg= changed with send
payload= {'really': 'powerfull', 'anything': 'a dict can hold!'}
```

1.2.5 Selective examples

Sometimes you only want to receive a signal if a certain condition occurs. This can be done by adding `senders` or `keys`.

```
import asyncio
import pprint

from asyncio_dispatch import Signal


class Somebody:
    '''
    Base class with a callback that will print the received arguments
    '''
    def message_received(self, signal, senders, keys, message, **kwargs):
        print('-' * 50)
        print('{} received a message'.format(self))
        print('-' * 50)
        if signal is SIGNAL1:
            print('SIGNAL #1 received')
        elif signal is SIGNAL2:
            print('SIGNAL #2 received')

        print('senders= ', end='')
        pprint.pprint(senders)
        print('keys= ', end='')
        pprint.pprint(keys)
        print('message= ', message)

    def __str__(self):
        return self.__class__.__name__

    def __repr__(self):
        return self.__str__()


class Mike(Somebody):
    pass


class Ashley(Somebody):
    pass


mike = Mike()
ashley = Ashley()

loop = asyncio.get_event_loop()

# create two signals
SIGNAL1 = Signal(loop=loop, message=None)
SIGNAL2 = Signal(loop=loop, message=None)

connection_tasks = [
    # connect the signals. Mike and Ashley are each connected using different keys and senders.
    # Their callbacks will only be executed if a Signal is sent with a matching key or sender.
    loop.create_task(SIGNAL1.connect(mike.message_received,
```

```
                                    sender=ashley,
                                    keys=['important', 'books'])),
    loop.create_task(SIGNAL2.connect(mike.message_received,
                                    keys=['logs'])),

    loop.create_task(SIGNAL1.connect(ashley.message_received,
                                    sender=mike,
                                    keys=['love-notes', 'music'])),
    loop.create_task(SIGNAL2.connect(ashley.message_received,
                                    keys=['alert']))
]

# ensure the connections are made before trying to send signals
loop.run_until_complete(asyncio.wait(connection_tasks))

send_tasks = [
    # Try to send the signals without senders or keys.
    # Nothing should happen as there are no matching senders or keys
    loop.create_task(SIGNAL1.send(message='nobody is listening')),
    loop.create_task(SIGNAL2.send(message='nobody is listening')),

    # Ashley sends Mike a message, and Mike replies. Matching by senders
    loop.create_task(SIGNAL1.send(sender=ashley,
                                  message='hello Mike!')),
    loop.create_task(SIGNAL1.send(sender=mike,
                                  message='hello Ashley!')),

    # an important message and alert come in. Matched by keys
    loop.create_task(SIGNAL1.send(key='important',
                                  message='important message for Mike')),
    loop.create_task(SIGNAL2.send(key='alert',
                                  message='alert for Ashley')),

    # then a book full of love notes. Matched keys rigger callbacks for both Mike and Ashley
    loop.create_task(SIGNAL1.send(keys=['books', 'love-notes'],
                                  message="Mike is waiting for books, Ashley for love-notes")),

    # if more than one sender or key match, only one execution of the callback is scheduled
    loop.create_task(SIGNAL1.send(keys=['important', 'books', 'logs'],
                                  message='Mike is subscribed to three matching keys,'
                                          ' but only one message is sent!')),

    # you get the idea
]

# run the event loop and see what happens.
loop.run_until_complete(asyncio.wait(send_tasks))
```

The above example prints the following output

```
------------------------------------------------
Mike received a message
------------------------------------------------
SIGNAL #1 received
senders= {Ashley}
keys= set()
message=  hello Mike!
------------------------------------------------
Ashley received a message
```

```
------------------------------------------------
SIGNAL #1 received
senders= {Mike}
keys= set()
message=  hello Ashley!
------------------------------------------------
Mike received a message
------------------------------------------------
SIGNAL #1 received
senders= set()
keys= {'important'}
message=  important message for Mike
------------------------------------------------
Ashley received a message
------------------------------------------------
SIGNAL #2 received
senders= set()
keys= {'alert'}
message=  alert for Ashley
------------------------------------------------
Mike received a message
------------------------------------------------
SIGNAL #1 received
senders= set()
keys= {'books', 'love-notes'}
message=  Mike is waiting for books, Ashley for love-notes
------------------------------------------------
Ashley received a message
------------------------------------------------
SIGNAL #1 received
senders= set()
keys= {'books', 'love-notes'}
message=  Mike is waiting for books, Ashley for love-notes
------------------------------------------------
Mike received a message
------------------------------------------------
SIGNAL #1 received
senders= set()
keys= {'important', 'logs', 'books'}
message=  Mike is subscribed to three matching keys, but only one message is sent!
```

1.3 API

The shorter import `asyncio_dispatch.Signal` is preferred over the longer `asyncio_dispatch.dispatcher.Signal`

CHAPTER 2

Indices and tables

- genindex
- modindex
- search