py-emit Documentation

Release 0.5.0

Aaron Yang

Oct 04, 2022

Contents:

CHAPTER 1

py-emit

light-weight asynchronous event system, support in-process communication, remote message server (currently redis only) communication and remote procedure call (RPC).

- Free software: MIT license
- Documentation: https://pyemit.readthedocs.io.

1.1 Features

- asynchronous support.
- support in-process communication
- support remote message server (currently redis, aioredis is required)
- RPC (Remote procedure call)

1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Stable release

To install py-emit, run this command in your terminal:

```
$ pip install pyemit
```

This is the preferred method to install py-emit, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for py-emit can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/zillionare/pyemit
```

Or download the tarball:

```
$ curl -OJL https://github.com/zillionare/pyemit/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use py-emit in a project

3.1 step 1. Import pyemit

```
from pyemit import emit
```

3.2 step 2. Register events

pyemit provides both annotation based registering and direct registering. Annotation can be used if the handler is not a class method.

Be noticed that the handler MUST be a coroutine function.

```
from pyemit import emit
@emit.on('event_name')
async def handler(msg):
    pass
```

or, if it's a class method, use `emit.register` instead:

```
from pyemit import emit
class Foo:
    async def bar(self, msg):
        pass

foo = Foo()
emit.register('event_name', foo.bar)
```

To know what causes the difference, please refer to $todo

3.3 step 3. Start the message pumping

you can start a local message pump by:

```
from pyemit import emit
async def init():
    await emit.start()
```

Or, if you're prefer a remote message server, use this:

```
from pyemit import emit
# aioredis is required. You can install it by running `pip install aioredis>=1.3.1
async def init(dsn):
    await emit.start(emit.Engine.REDIS, dsn=dsn)
```

3.4 step 4. Fire and consume events

Now you can fire and consume events. To emit a message, running:

```
from pyemit import emit
async def foo():
    # construct msg
    msg = {}
    await emit.emit('event_name', msg)
```

You can fire an event without providing any message. By doing so, be sure provide no parameter to the handler. The msg must be and dict and is json serializable.

CHAPTER 4

Troubeshooting

You can enable heartbeat when using REDIS engine:

```
from pyemit import emit
async def init(dsn):
    await emit.start(emit.Engine.REDIS, dsn=dsn, heart_beat=1)
```

and set logging level to DEBUG, this should print a heart beat msg every 1 second. If not seen, then check your configurations and Redis setup.

CHAPTER 5

Advanced topic

5.1 RPC call

```
from pyemit import emit as e
from pyemit.remote import Remote

class Sum(Remote):
    def __init__(self, to_be_sum):
        super().__init__()
        self.to_be_sum = to_be_sum

    async def server_impl(self, *args, **kwargs):
        result = sum(self.to_be_sum)
        await super().respond(result)

async def test_rpc():
    await e.start(e.Engine.REDIS, dsn="redis://localhost")
    foo = Sum([0, 1, 2])
    response = await foo.invoke()
    assert response == 3
```

step 1. Subclass from `Remote`, and implement `Remote.server_impl` method. This one is supposed to be executed on the server side. When calculation is done, then call `super().respond()` to send the result back to client

step 2. At client side, create instance of the subclass you defined (i.e., `foo` in the example), then by calling `await foo.invoke()` you will get what you want.

pass parameters during construct of Remote object if any, they can be access at server side by self object.

5.2 stop the message pump

call `emit.stop` to stop the whole machine. call `emit.unsubscribe` to remove a handler.

CHAPTER 6

pyemit

6.1 pyemit package

6.1.1 Submodules

6.1.2 pyemit.emit module

class pyemit.emit.**Engine**
Bases: enum.IntEnum

An enumeration.

IN_PROCESS = 0

REDIS = 1

pyemit.emit.**async_register**(*event: str*, *handler: Callable*, *exchange=''*)
‘handler‘‘event‘ :param event: :param handler: :param exchange: :return:

pyemit.emit.**emit**(*channel: str*, *message: Any = None*, *exchange=''*)
publish a message to channel. :param channel: the name of channel :param message: :return:

pyemit.emit.**on**(*event*, *exchange=''*)
Args:

event: exchange:

Returns:

pyemit.emit.**register**(*event: str*, *handler: Callable*, *exchange=''*)
‘event‘ :param event: :param handler: :param exchange: :return:

pyemit.emit.**rpc_respond**(*msg: dict*)

pyemit.emit.**rpc_send**(*remote*) → Any
emit msg and wait response back. The func will add __emit_sn__ to the dict, and the server should echo the serial number back. Args:

remote:

Returns:

pyemit.emit.**start**(*engine: pyemit.emit.Engine = <Engine.IN_PROCESS: 0>, start_server=False, heart_beat=0, **kwargs*)

Args: engine: start_server: heart_beat:

pyemit.emit.**stop**()
redis

pyemit.emit.**unsubscribe**(*channel: str, handler: Callable, exchange=''*)
stop subscribe message from channel :param channel: :param handler: :return:

6.1.3 pyemit.remote module

Author: Aaron-Yang [code@jieyu.ai] Contributors:

class pyemit.remote.**Remote**(*timeout=10*)
Bases: object

invoke()

static loads(*s*) → pyemit.remote.Remote

respond(*result: Any*)

server_impl()

sn

timeout

6.1.4 Module contents

Top-level package for py-emit.

CHAPTER 7

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs at https://github.com/zillionare/pyemit/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

7.1.4 Write Documentation

py-emit could always use more documentation, whether as part of the official py-emit docs, in docstrings, or even on the web in blog posts, articles, and such.

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/zillionare/pyemit/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Get Started!

Ready to contribute? Here's how to set up *pyemit* for local development.

1. Fork the *pyemit* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pyemit.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pyemit
$ cd pyemit/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 pyemit tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, 3.6, 3.7 and 3.8, and for PyPy. Check https://travis-ci.com/zillionare/pyemit/pull_requests and make sure that the tests pass for all supported Python versions.

7.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_pyemit
```

7.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 8

Credits

8.1 Development Lead

- Aaron Yang <aaron_yang@jieyu.ai>

8.2 Contributors

None yet. Why not be the first?

CHAPTER 9

History

9.1 0.1.0 (2020-04-23)

- First release on PyPI.

9.2 0.3.0 (2020-04-25)

- RPC feature implemented
- use pickle instead of json for serialization for better performance. Currently protocol 4 is used.

9.3 0.4.0 (2020-04-30)

- Change signature of *Remote.execute* to *Remote.invoke*

9.4 0.4.5 (2020-06-15)

- fixed #1

9.5 0.4.6 (2021-12-28)

- Fixed that the stop method did not empty hannlers

9.6 0.4.7 (2021-12-29)

- Fixed that the stop method did not empty hannlers, then add function error

9.7 0.4.8 (2021-12-29)

- Add asynchronous registration

9.8 0.5.0 (2021-12-30)

- Update aioredis version to 2.0.1

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index