aiothrift Documentation

Release 0.1

moonshadow

January 18, 2017

Contents

asyncio (**PEP 3156**) Thrift client and server library.

CHAPTER 1

Installation

The easiest way to install aiothrift is by using the packae on Pypi:

```
pip install aiothrift
```

CHAPTER 2

Requirements

- Python 3.4 +

CHAPTER 3

Contribute

- Issue Tracker: https://github.com/moonshadow/aiothrift/issues
- Source Code: https://github.com/moonshadow/aiothrift

Feel free to file an issue or make pull request if you find any bugs or have some suggestions for library improvement.

CHAPTER 4

User's Guide

4.1 Quickstart

This page gives you a good introduction to aiothrift. It assumes you already have aiothrift installed.

4.1.1 A Minimal Application

At first you should have a thrift file which defines at least one service. Go to create a thrift file named `pingpong.thrift`:

```
service PingPong {
    string ping(),
    i32 add(1:i32 a, 2:i32 b),
}
```

Now you can fire an asyncio thrift server easily:

```
import asyncio
import thriftpy
from aiothrift import create_server

class Dispatcher:
    def ping(self):
        return "pong"

    async def add(self, a, b):
        await asyncio.sleep(2)
        return a + b

pingpong_thrift = thriftpy.load('pingpong.thrift', module_name='pingpong_thrift')
loop = asyncio.get_event_loop()
server = loop.run_until_complete(create_server(pingpong_thrift.PingPong, Dispatcher(), loop=loop))
loop.run_forever()
```

let's have a look at what the code above does.

1. Frist we import the `thriftpy` module, which is used to parse a thrift file to a valid python module, thanks for the great job done by *thriftpy*, we don't have to generate thrift python sdk files manually.

2. We create a *Dispatcher* class as the namespace for all thrift rpc functions. Here we define a *ping* method which corresponds to the *ping* function defined in `pingpong.thrift`. You may notice that the *add* method is actually a

coroutine but a normal one. if you define the rpc function as a coroutine, it would scheduled by our thrift server and send the result back to client after the coroutine task is completed.

3. We then create the server by using *`create_server()`* function, and it returns a coroutine instance which can be scheduled by the event loop later.

4. Lastly we call `loop.run_forever()` to run the event loop to schedule the server task.

Just save it as `server.py` and then you can start the thrift server:

```
$ python3 server.py
```

It will listening at *localhost:6000* by default.

Now you'd like to visit the thrift server through a thrift client:

```
import asyncio
import thriftpy
from aiothrift import create_connection

pingpong_thrift = thriftpy.load('pingpong.thrift', module_name='pingpong_thrift')

loop = asyncio.get_event_loop()


async def create_client():
    conn = await create_connection(pingpong_thrift.PingPong, loop=loop, timeout=10)
    print(await conn.ping())
    conn.close()

loop.run_until_complete(create_client())
```

Look that *create_client* is the client task coroutine, this task would create a connection to the server we've created earlier, and make *ping* rpc call, print its result and close the connection.

Save it as `client.py`, and run the client by:

```
    $ python client.py
     * pong
```

That's all you need to make a minimal thrift application on both the server and client side, I hope you will enjoy it.

4.2 Examples of aiothrift usage

4.2.1 sample thrift file

`get source code`

```
service PingPong {
    string ping(),

    i64 add(1:i32 a, 2:i64 b),
}
```

4.2.2 aio thrift server

`get source code`

```
import asyncio
import thriftpy

from aiothrift.server import create_server

pingpong_thrift = thriftpy.load('pingpong.thrift', module_name='pingpong_thrift')


class Dispatcher:
    def ping(self):
        return "pong"

    async def add(self, a, b):
        await asyncio.sleep(2)
        return a + b


loop = asyncio.get_event_loop()

server = loop.run_until_complete(
    create_server(pingpong_thrift.PingPong, Dispatcher(), ('127.0.0.1', 6000), loop=loop, timeout=10)

print('server is listening on host {} and port {}'.format('127.0.0.1', 6000))

try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

4.2.3 aio thrift client

get source code

```
import thriftpy
import asyncio
import aiothrift

pingpong_thrift = thriftpy.load('pingpong.thrift', module_name='pingpong_thrift')

loop = asyncio.get_event_loop()


async def create_connection():
    conn = await aiothrift.create_connection(pingpong_thrift.PingPong, ('127.0.0.1', 6000), loop=loop
    print(await conn.ping())
    print(await conn.add(5, 6))
    conn.close()


loop.run_until_complete(create_connection())

loop.close()
```

4.2.4 connection pool sample

`get source code`

```
import thriftpy
import aiothrift
import asyncio

pingpong_thrift = thriftpy.load('pingpong.thrift', module_name='pingpong_thrift')


async def create_pool():
    return await aiothrift.create_pool(pingpong_thrift.PingPong, ('127.0.0.1', 6000), loop=loop, time


async def run_pool(pool):
    try:
        async with pool.get() as conn:
            print(await conn.add(5, 6))
            print(await conn.ping())
    except asyncio.TimeoutError:
        pass

    async with pool.get() as conn:
        print(await conn.ping())


loop = asyncio.get_event_loop()

pool = loop.run_until_complete(create_pool())
tasks = []
for i in range(10):
    tasks.append(asyncio.ensure_future(run_pool(pool)))

loop.run_until_complete(asyncio.gather(*tasks))
pool.close()
loop.run_until_complete(pool.wait_closed())

loop.close()
```

4.3 API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

4.3.1 API

This part of the documentation covers all the interfaces of aiothrift. For parts where aiothrift depends on external libraries, we document the most important right here and provide links to the canonical documentation.

ThriftConnection Object

class `aiothrift.`**`ThriftConnection`**(*service*, *, *iprot*, *oprot*, *address*, *loop=None*, *timeout=None*)

Thrift Connection.

`_init_rpc_apis`()

find out all apis defined in thrift service, and create corresponding method on the connection object, ignore it if some api name is conflicted with an existed attribute of the connection object, which you should call by using the `execute()` method.

`_recv`(*api*)

A coroutine which receive response from the thrift server

`execute`(*api*, **args*, ***kwargs*)

Execute a rpc call by api name. This is function is a coroutine.

Parameters

- **`api`** – api name defined in thrift file
- **`args`** – positional arguments passed to api function
- **`kwargs`** – keyword arguments passed to api function

Returns result of this rpc call

Raises `TimeoutError` if this task has exceeded the *timeout*

Raises `ThriftAppError` if thrift response is an exception defined in thrift.

Raises `ConnectionClosedError`: if server has closed this connection.

ThriftConnection Pool

class `aiothrift.`**`ThriftPool`**(*service*, *address*, ***, *minsize*, *maxsize*, *loop=None*, *timeout=None*)

Thrift connection pool.

`acquire`()

Acquires a connection from free pool.

Creates new connection if needed.

`clear`()

Clear pool connections.

Close and remove all free connections. this pattern is interesting

`close`()

Close all free and in-progress connections and mark pool as closed.

`fill_free`(***, *override_min*)

make sure at least *self.minsize* amount of connections in the pool if *override_min* is True, fill to the *self.maxsize*.

`freesize`

Current number of free connections.

`release`(*conn*)

Returns used connection back into pool.

When queue of free connections is full the connection will be dropped.

`size`

Current connection total num, acquiring connection num is counted

protocol

class `aiothrift.`**`TProtocol`**(*trans*, *strict_read=True*, *strict_write=True*, *decode_response=True*)

Base class for thrift protocols, subclass should implement some of the protocol methods, currently we only have *`TBinaryProtocol`* implemented for you.

class `aiothrift.`**`TBinaryProtocol`**(*trans*, *strict_read=True*, *strict_write=True*, *decode_response=True*)

Binary implementation of the Thrift protocol driver.

processor

class `aiothrift.`**`TProcessor`**(*service*, *handler*)

Base class for thrift rpc processor, which works on two streams.

server

class `aiothrift.`**`Server`**(*processor*, *protocol_cls=<class 'aiothrift.protocol.TBinaryProtocol'>*, *timeout=None*)

exceptions

class `aiothrift.`**`ThriftError`**

Base Exception defined by *aiothrift*

class `aiothrift.`**`ConnectionClosedError`**

Raised if connection to server was closed.

class `aiothrift.`**`PoolClosedError`**

Raised when operating on a closed thrift connection pool

class `aiothrift.`**`ThriftAppError`**(*type=0*, *message=None*)

Application level thrift exceptions.

Useful functions

`aiothrift.`**`create_server`**(*service*, *handler*, *address=('127.0.0.1', 6000)*, *loop=None*, *protocol_cls=<class 'aiothrift.protocol.TBinaryProtocol'>*, *timeout=None*)

create a thrift server. This function is a coroutine.

Parameters

- **`service`** – thrift Service
- **`handler`** – a dispatcher object which is a namespace for all thrift api functions.
- **`address`** – (host, port) tuple, default is ('127.0.0.1', 6000)
- **`loop`** – `Eventloop` instance
- **`protocol_cls`** – thrift protocol class, default is *`TBinaryProtocol`*
- **`timeout`** – server side timeout, default is None

Returns a *`Server`* object which can be used to stop the service `aiothrift.`**`create_connection`**(*service*, *address=('127.0.0.1', 6000)*, ***, *protocol_cls=<class 'aiothrift.protocol.TBinaryProtocol'>*, *timeout=None*, *loop=None*)

Create a thrift connection. This function is a coroutine.

Open a connection to the thrift server by address argument.

Parameters

- **`service`** – a thrift service object
- **`address`** – a (host, port) tuple
- **`protocol_cls`** – protocol type, default is *`TBinaryProtocol`*
- **`timeout`** – if specified, would raise *asyncio.TimeoutError* if one rpc call is longer than *timeout*
- **`loop`** – `Eventloop` instance, if not specified, default loop is used.

Returns newly created *`ThriftConnection`* instance.

`aiothrift.`**`create_pool`**(*service*, *address=('127.0.0.1', 6000)*, ***, *minsize=1*, *maxsize=10*, *loop=None*, *timeout=None*)

Create a thrift connection pool. This function is a coroutine.

Parameters

- **`service`** – service object defined by thrift file
- **`address`** – (host, port) tuple, default is ('127.0.0.1', 6000)
- **`minsize`** – minimal thrift connection, default is 1
- **`maxsize`** – maximal thrift connection, default is 10
- **`loop`** – targeting `eventloop`
- **`timeout`** – default timeout for each connection, default is None

Returns *`ThriftPool`* instance

4.4 Additional Notes

Information and changelog are here for the interested.

4.4.1 LICENSE

4.4.2 Changelog

Here you can see the full list of changes between each `aiothrift` release.

Version 0.1

First public release.

Python Module Index

Index