aioconsole

Nov 03, 2020

Contents

Asynchronous console and interfaces for asyncio

aioconsole provides:

- asynchronous equivalents to input, exec and code.interact
- an interactive loop running the asynchronous python console
- a way to customize and run command line interface using argparse
- stream support to serve interfaces instead of using standard streams
- the `apython` script to access asyncio code at runtime without modifying the sources

CHAPTER 1

Requirements

- Python >= 3.6

CHAPTER 2

Installation

aioconsole is available on PyPI and GitHub. Both of the following commands install the `aioconsole` package and the `apython` script.

```
$ pip3 install aioconsole   # from PyPI
$ python3 setup.py install  # or from the sources
$ apython -h
usage: apython [-h] [--serve [HOST:] PORT] [--no-readline]
               [--banner BANNER] [--locals LOCALS]
               [-m MODULE | FILE] ...
```

CHAPTER 3

Asynchronous console

The example directory includes a slightly modified version of the echo server from the asyncio documentation. It runs an echo server on a given port and save the received messages in `loop.history`.

It runs fine and doesn't use any `aioconsole` function:

```
$ python3 -m example.echo 8888
The echo service is being served on 127.0.0.1:8888
```

In order to access the program while it's running, simply replace `python3` with `apython` and redirect `stdout` so the console is not polluted by `print` statements (`apython` uses `stderr`):

```
$ apython -m example.echo 8888 > echo.log
Python 3.5.0 (default, Sep 7 2015, 14:12:03)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
---
This console is running in an asyncio event loop.
It allows you to wait for coroutines using the 'await' syntax.
Try: await asyncio.sleep(1, result=3, loop=loop)
---
>>>
```

This looks like the standard python console, with an extra message. It suggests using the `await` syntax (`yield from` for python 3.4):

```
>>> await asyncio.sleep(1, result=3, loop=loop)
# Wait one second...
3
>>>
```

The `locals` contain a reference to the event loop:

```
>>> dir()
['__doc__', '__name__', 'asyncio', 'loop']
```

(continues on next page)

(continued from previous page)

```
>>> loop
<InteractiveEventLoop running=True closed=False debug=False>
>>>
```

So we can access the `history` of received messages:

```
>>> loop.history
defaultdict(<class 'list'>, {})
>>> sum(loop.history.values(), [])
[]
```

Let's send a message to the server using a netcat client:

```
$ nc localhost 8888
Hello!
Hello!
```

The echo server behaves correctly. It is now possible to retrieve the message:

```
>>> sum(loop.history.values(), [])
['Hello!']
```

The console also supports `Ctrl-C` and `Ctrl-D` signals:

```
>>> ^C
KeyboardInterrupt
>>> # Ctrl-D
$
```

All this is implemented by setting `InteractiveEventLoop` as default event loop. It simply is a selector loop that schedules `aioconsole.interact()` coroutine when it's created.

CHAPTER 4

Serving the console

Moreover, `aioconsole.interact()` supports stream objects so it can be used along with asyncio.start_server to serve the python console. The `aioconsole.start_interactive_server` coroutine does exactly that. A backdoor can be introduced by simply adding the following line in the program:

```
server = await aioconsole.start_interactive_server(
    host='localhost', port=8000)
```

This is actually very similar to the eventlet.backdoor module. It is also possible to use the `--serve` option so it is not necessary to modify the code:

```
$ apython --serve :8889 -m example.echo 8888
The console is being served on 0.0.0.0:8889
The echo service is being served on 127.0.0.1:8888
```

Then connect using netcat and optionally, rlwrap:

```
$ rlwrap nc localhost 8889
Python 3.5.0 (default, Sep 7 2015, 14:12:03)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
---
This console is running in an asyncio event loop.
It allows you to wait for coroutines using the 'await' syntax.
Try: await asyncio.sleep(1, result=3, loop=loop)
---
>>>
```

Great! Anyone can now forkbomb your machine:

```
>>> import os
>>> os.system(':(){ :|:& };:')
```

CHAPTER 5

Command line interfaces

The package also provides an `AsychronousCli` object. It is initialized with a dictionary of commands and can be scheduled with the coroutine `async_cli.interact()`. A dedicated command line interface to the echo server is defined in example/cli.py. In this case, the command dictonary is defined as:

```
commands = {'history': (get_history, parser)}
```

where `get_history` is a coroutine and `parser` an ArgumentParser from the argparse module. The arguments of the parser will be passed as keywords arguments to the coroutine.

Let's run the command line interface:

```
$ python3 -m example.cli 8888 > cli.log
Welcome to the CLI interface of echo!
Try:
* 'help' to display the help message
* 'list' to display the command list.
>>>
```

The `help` and `list` commands are generated automatically:

```
>>> help
Type 'help' to display this message.
Type 'list' to display the command list.
Type '<command> -h' to display the help message of <command>.
>>> list
List of commands:
 * help [-h]
 * history [-h] [--pattern PATTERN]
 * list [-h]
>>>
```

The `history` command defined earlier can be found in the list. Note that it has an `help` option and a `pattern` argument:

```
>>> history -h
usage: history [-h] [--pattern PATTERN]

Display the message history

optional arguments:
  -h, --help            show this help message and exit
  --pattern PATTERN, -p PATTERN
                        pattern to filter hostnames
```

Example usage of the `history` command:

```
>>> history
No message in the history
>>> # A few messages later
>>> history
Host 127.0.0.1:
  0. Hello!
  1. Bye!
Host 192.168.0.3
  0. Sup!
>>> history -p 127.*
Host 127.0.0.1:
  0. Hello!
  1. Bye!
```

CHAPTER 6

Serving interfaces

Just like `asyncio.interact()`, `AsynchronousCli` can be initialized with any pair of streams. It can be used along with asyncio.start_server to serve the command line interface. The previous example provides this functionality through the `--serve-cli` option:

```
$ python3 -m example.cli 8888 --serve-cli 8889
The command line interface is being served on 127.0.0.1:8889
The echo service is being served on 127.0.0.1:8888
```

It's now possible to access the interface using netcat:

```
$ rlwrap nc localhost 8889
Welcome to the CLI interface of echo!
Try:
 * 'help' to display the help message
 * 'list' to display the command list.
>>>
```

It is also possible to combine the example with the `apython` script to add an extra access for debugging:

```
$ apython --serve 8887 -m example.cli 8888 --serve-cli 8889
The console is being served on 127.0.0.1:8887
The command line interface is being served on 127.0.0.1:8889
The echo service is being served on 127.0.0.1:8888
```

CHAPTER 7

Limitations

The python console exposed by aioconsole is quite limited compared to modern consoles such as IPython or ptpython. Luckily, those projects gained greater asyncio support over the years. In particular, the following use cases overlap with aioconsole capabilities:

- Embedding a ptpython console in an asyncio program
- Using the await syntax in an IPython console

CHAPTER 8

Contact

Vincent Michel: vxgmichel@gmail.com