greenado Documentation

Release 0.2.5

Author

Mar 07, 2018

Contents

Greenado is a utility library that provides greenlet-based coroutines for tornado. In tornado, coroutines allow you to perform asynchronous operations without using callbacks, providing a pseudo-synchronous flow in your functions.

When using Tornado's `@gen.coroutine` in a large codebase, you will notice that they tend to be 'infectious' from the bottom up. In other words, for them to be truly useful, callers of the coroutine should 'yield' to them, which requires them to be a coroutine. In turn, their callers need to 'yield', and so on.

Instead, greenado coroutines infect from the top down, and only requires the *`@greenado.groutine`* decorator *somewhere* in the call hierarchy, but it doesn't really matter where. Once the decorator is used, you can use *`greenado.gyield()`* to pseudo-synchronously wait for asynchronous events to occur. This reduces complexity in large codebases, as you only need to use the decorator at the very top of your call trees, and nowhere else.

CHAPTER 1

Documentation

Documentation can be found at http://greenado.readthedocs.org/en/latest/

CHAPTER 2

Installation & Requirements

Installation is easiest using pip:

```
$ pip install greenado
```

greenado should work using tornado 3.2, but I only actively use it in tornado 4+

I have only tested greenado on Linux & OSX, but I imagine that it would work correctly on platforms that tornado and greenlet support.

CHAPTER 3

Example usage

In the below examples, 'main_function' is your toplevel function in the call hierarchy that needs to call things that eventually call some asynchronous operation in tornado.

Normal tornado coroutine usage might look something like this:

```
from tornado import gen

@gen.coroutine
def do_long_operation():
    retval = yield long_operation()
    raise gen.Return(retval)

@gen.coroutine
def call_long_operation():
    retval = yield do_long_operation()
    raise gen.Return(retval)

@gen.coroutine
def main_function():
    retval = yield call_long_operation()
```

With greenado, it looks something like this instead:

```
import greenado

def do_long_operation():
    retval = greenado.gyield(long_operation())
    return retval

def call_long_operation():
    retval = do_long_operation()
    return retval

@greenado.groutine
```

```
def main_function():
    retval = call_long_operation()
```

Functions wrapped by *@greenado.groutine* return a tornado.concurrent.Future object which you must either yield, call result(), or use IOLoop.add_future on, otherwise you may risk swallowing exceptions.

3.1 Why can't I use the yield keyword?

Well, actually, if you use yet another decorator, you still can! Check out this example:

```
import greenado

    @greenado.generator
def do_long_operation():
    retval = yield long_operation()
    return retval

def call_long_operation():
    retval = do_long_operation()
    return retval

@greenado.groutine
def main_function():
    retval = call_long_operation()
```

You'll note that this is very similar to the coroutines available from tornado (and in fact, the implementation is mostly the same), but the difference is that (once again) you don't need to do anything special to call the do_long_operation function, other than make sure that *@greenado.groutine* is in the call stack somewhere.

CHAPTER 4

Testing

greenado.testing contains a function called gen_test which can be used exactly like `tornado.testing.gen_test()`:

```
import greenado

from greenado.testing import gen_test
from tornado.testing import AsyncTestCase

def something_that_yields():
    greenado.gyield(something())

class MyTest(AsyncTestCase):
    @gen_test
    def test_something(self):
        something_that_yields()
```

CHAPTER 5

Contributing new changes

1. Fork this repository
2. Create your feature branch (*git checkout -b my-new-feature*)
3. Test your changes (*tests/run_tests.sh*)
4. Commit your changes (*git commit -am 'Add some feature'*)
5. Push to the branch (*git push origin my-new-feature*)
6. Create new Pull Request

CHAPTER 6

Credit

Greenado is similar to and inspired by https://github.com/mopub/greenlet-tornado and https://github.com/Gawen/tornalet, but does not require that you use it from a tornado web handler as they do.

CHAPTER 7

Authors

Dustin Spicuzza (dustin@virtualroadside.com)

7.1 Function Reference

7.1.1 greenado.concurrent

exception greenado.concurrent.**TimeoutError**
Bases: Exception

Exception raised by gyield in timeout.

greenado.concurrent.**gcall**(*f*, **args*, ***kwargs*)
Calls a function, makes it asynchronous, and returns the result of the function as a tornado.concurrent.Future. The wrapped function may use *gyield()* to pseudo-synchronously wait for a future to resolve.

This is the same code that *@greenado.groutine* uses to wrap functions.

Parameters

- **f** – Function to call
- **args** – Function arguments
- **kwargs** – Function keyword arguments

Returns tornado.concurrent.Future

> **Warning:** You should not discard the returned Future or exceptions may be silently discarded, similar to a tornado coroutine. See @gen.coroutine for details.

greenado.concurrent.**generator**(*f*)
A decorator that allows you to use the 'yield' keyword in a function, without requiring callers to also yield this function.

The yield keyword can be used inside a decorated function on any function call that returns a future object, such as functions decorated by `@gen.coroutine`, and most of the tornado API as of tornado 4.0.

Similar to `@gen.coroutine`, in versions of Python before 3.3 you must raise `tornado.gen.Return` to return a value from this function.

This function must only be used by functions that either have a *`@greenado.groutine`* decorator, or functions that are children of functions that have the decorator applied.

New in version 0.1.7.

> **Warning:** You should not discard the returned Future or exceptions may be silently discarded, similar to a tornado coroutine. See `@gen.coroutine` for details.

`greenado.concurrent.`**`gmoment`**`()`
Similar to `tornado.gen.moment()`, yields the IOLoop for a single iteration from inside a groutine.

`greenado.concurrent.`**`groutine`**`(`*`f`*`)`
A decorator that makes a function asynchronous and returns the result of the function as a `tornado.concurrent.Future`. The wrapped function may use *`gyield()`* to pseudo-synchronously wait for a future to resolve.

The primary advantage to using this decorator is that it allows *all* called functions and their children to use *`gyield()`*, and doesn't require the use of generators.

If you are calling a groutine-wrapped function from a function with a *`@greenado.groutine`* decorator, you will need to use *`gyield()`* to wait for the returned future to resolve.

From a caller's perspective, this decorator is functionally equivalent to the `@gen.coroutine` decorator. You should not use this decorator and the `@gen.coroutine` decorator on the same function.

> **Warning:** You should not discard the returned Future or exceptions may be silently discarded, similar to a tornado coroutine. See `@gen.coroutine` for details.

`greenado.concurrent.`**`gsleep`**`(`*`timeout`*`)`
This will yield and allow other operations to occur in the background before returning.

Parameters **`timeout`** – Number of seconds to wait

New in version 0.1.9.

`greenado.concurrent.`**`gyield`**`(`*`future`*`, `*`timeout=None`*`)`
This is functionally equivalent to the 'yield' statements used in a `@gen.coroutine`, but doesn't require turning all your functions into generators – so you can use the return statement normally, and exceptions won't be accidentally discarded.

This can be used on any function that returns a future object, such as functions decorated by `@gen.coroutine`, and most of the tornado API as of tornado 4.0.

This function must only be used by functions that either have a *`@greenado.groutine`* decorator, or functions that are children of functions that have the decorator applied.

Parameters

- **`future`** – A `tornado.concurrent.Future` object
- **`timeout`** – Number of seconds to wait before raising a *`TimeoutError`*. Default is no timeout. *Parameter added in version 0.1.8.*

Returns The result set on the future object

Raises

- If an exception is set on the future, the exception will be thrown to the caller of gyield.
- If the timeout expires, *TimeoutError* will be raised.

Changed in version 0.1.8: Added timeout parameter

Changed in version 0.2.0: If a timeout occurs, the *TimeoutError* will not be set on the future object, but will only be raised to the caller.

7.1.2 greenado.testing

greenado.testing.**gen_test**(*func=None*, *timeout=None*)

An implementation of tornado.testing.gen_test() for *@greenado.groutine*

Example:

```
def something_that_yields():
    greenado.gyield(something())

class MyTest(AsyncTestCase):
    @greenado.testing.gen_test
    def test_something(self):
        something_that_yields()
```

7.2 Release notes

7.2.1 0.2.5 - 2018-03-06

- Fix compatibility with Tornado >= 5.0

7.2.2 0.2.4 - 2016-06-06

- Reorder gyield to optimize non-timeout case

7.2.3 0.2.3 - 2016-05-13

- tornado.gen.moment doesn't work, use gmoment instead

7.2.4 0.2.2 - 2016-04-20

- Retain current StackContext when using gcall or groutine

7.2.5 0.2.1 - 2016-04-19

- Fix StackContextInconsistentError when using gyield inside of a tornado.stack_context.StackContext context block

7.2.6 0.2.0 - 2016-04-06

- Breaking change: Changed behavior of gyield timeout to throw, instead of setting an exception on the yielded future

7.2.7 0.1.9 - 2015-08-05

- Added gsleep

7.2.8 0.1.8 - 2014-10-23

- Added sphinx documentation
- Added timeout parameter to *`gyield()`* (thanks Paul Fultz)

7.2.9 0.1.7 - 2014-09-11

- Added *`greenado.concurrent.generator()`* decorator to allow usage of the yield keyword instead of gyield

7.2.10 0.1.6 - 2014-09-04

- Use `tornado.concurrent.TracebackFuture` to show correct stack traces when exceptions occur
- Add CHANGELOG.md

7.2.11 0.1.5 - 2014-08-28

- Add a *`gen_test()`* implementation

7.2.12 0.1.4 - 2014-08-28

- Fix bug with nested groutines + double yield

7.2.13 0.1.3 - 2014-08-28

- Add *`gcall()`* to the API

7.2.14 0.1.2 - 2014-08-28

- Short-circuit futures that have already completed

7.2.15 0.1.1 - 2014-08-28

- Initial working version

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

G

T