Toro Documentation

Release 1.0.1.dev0

A. Jesse Jiryu Davis

January 27, 2016

Toro logo by Musho Rodney Alan Greenblat

With Tornado's gen module, you can turn Python generators into full-featured coroutines, but coordination among these coroutines is difficult without mutexes, semaphores, and queues.

Toro provides to Tornado coroutines a set of locking primitives and queues analogous to those that Gevent provides to Greenlets, or that the standard library provides to threads.

Important: Toro is completed and deprecated; its features have been merged into Tornado. Development of locks and queues for Tornado coroutines continues in Tornado itself.

CHAPTER 1

The Wait / Notify Pattern

Toro's *primitives* follow a "wait / notify pattern": one coroutine waits to be notified by another. Let's take `Condition` as an example:

```
>>> import toro
>>> from tornado import ioloop, gen
>>> loop = ioloop.IOLoop.current()
>>> condition = toro.Condition()
>>> @gen.coroutine
... def waiter():
...     print "I'll wait right here"
...     yield condition.wait()  # Yield a Future
...     print "I'm done waiting"
...
>>> @gen.coroutine
... def notifier():
...     print "About to notify"
...     condition.notify()
...     print "Done notifying"
...
>>> @gen.coroutine
... def runner():
...     # Yield two Futures; wait for waiter() and notifier() to finish
...     yield [waiter(), notifier()]
...     loop.stop()
...
>>> future = runner(); loop.start()
I'll wait right here
About to notify
Done notifying
I'm done waiting
```

Wait-methods take an optional `deadline` argument, which is either an absolute timestamp:

```
loop = ioloop.IOLoop.current()

# Wait up to 1 second for a notification
yield condition.wait(deadline=loop.time() + 1)
```

...or a `datetime.timedelta` for a deadline relative to the current time:

```
# Wait up to 1 second
yield condition.wait(deadline=datetime.timedelta(seconds=1))
```

If there's no notification before the deadline, the Toro-specific `Timeout` exception is raised.

CHAPTER 2

The Get / Put Pattern

Queue and its subclasses support methods *Queue.get()* and *Queue.put()*. These methods are each both a wait-method **and** a notify-method:

- *Queue.get()* waits until there is an available item in the queue, and may notify a coroutine waiting to put an item.
- *Queue.put()* waits until the queue has a free slot, and may notify a coroutine waiting to get an item.

Queue.get() and *Queue.put()* accept deadlines and raise *Timeout* if the deadline passes.

See the Producer-consumer example.

Additionally, *JoinableQueue* supports the wait-method *JoinableQueue.join()* and the notify-method *JoinableQueue.task_done()*.

CHAPTER 3

Contents

3.1 Examples

3.1.1 Producer-consumer example

A classic producer-consumer example for using *JoinableQueue*.

```
from tornado import ioloop, gen
import toro
q = toro.JoinableQueue(maxsize=3)


@gen.coroutine
def producer():
    for item in range(10):
        print 'Sending', item
        yield q.put(item)


@gen.coroutine
def consumer():
    while True:
        item = yield q.get()
        print '\t\t', 'Got', item
        q.task_done()


if __name__ == '__main__':
    producer()
    consumer()
    loop = ioloop.IOLoop.current()

    def stop(future):
        loop.stop()
        future.result()  # Raise error if there is one

    # block until all tasks are done
    q.join().add_done_callback(stop)
    loop.start()
```

3.1.2 `Lock` example - graceful shutdown

Graceful shutdown, an example use case for *`Lock`*.

`poll` continuously fetches http://tornadoweb.org, and after 5 seconds, `shutdown` stops the IOLoop. We want any request that `poll` has begun to complete before the loop stops, so `poll` acquires the lock before starting each HTTP request and releases it when the request completes. `shutdown` also acquires the lock before stopping the IOLoop.

(Inspired by a post to the Tornado mailing list.)

```
import datetime
from tornado import ioloop, gen, httpclient
import toro

lock = toro.Lock()
loop = ioloop.IOLoop.current()


@gen.coroutine
def poll():
    client = httpclient.AsyncHTTPClient()
    while True:
        with (yield lock.acquire()):
            print 'Starting request'
            response = yield client.fetch('http://www.tornadoweb.org/')
            print response.code

        # Wait a tenth of a second before next request
        yield gen.Task(loop.add_timeout, datetime.timedelta(seconds=0.1))


@gen.coroutine
def shutdown():
    # Get the lock: this ensures poll() isn't in a request when we stop the
    # loop
    print 'shutdown() is acquiring the lock'
    yield lock.acquire()
    loop.stop()
    print 'Loop stopped.'


if __name__ == '__main__':
    # Start polling
    poll()

    # Arrange to shutdown cleanly 5 seconds from now
    loop.add_timeout(datetime.timedelta(seconds=5), shutdown)
    loop.start()
```

3.1.3 `Event` example - a caching proxy server

An oversimplified caching HTTP proxy - start it, and configure your browser to use localhost:8888 as the proxy server. It doesn't do cookies or redirects, nor does it obey cache-control headers.

The point is to demonstrate *`Event`*. Imagine a client requests a page, and while the proxy is downloading the page from the external site, a second client requests the same page. Since the page is not yet in cache, an inefficient proxy would launch a second external request.

This proxy instead places an *Event* in the cache, and the second client request waits for the event to be set, thus requiring only a single external request.

```
from tornado import httpclient, gen, ioloop, web
import toro


class CacheEntry(object):
    def __init__(self):
        self.event = toro.Event()
        self.type = self.body = None

cache = {}


class ProxyHandler(web.RequestHandler):
    @web.asynchronous
    @gen.coroutine
    def get(self):
        path = self.request.path
        entry = cache.get(path)
        if entry:
            # Block until the event is set, unless it's set already
            yield entry.event.wait()
        else:
            print path
            cache[path] = entry = CacheEntry()

            # Actually fetch the page
            response = yield httpclient.AsyncHTTPClient().fetch(path)
            entry.type = response.headers.get('Content-Type', 'text/html')
            entry.body = response.body
            entry.event.set()

        self.set_header('Content-Type', entry.type)
        self.write(entry.body)
        self.finish()


if __name__ == '__main__':
    print 'Listening on port 8888'
    print
    print 'Configure your web browser to use localhost:8888 as an HTTP Proxy.'
    print 'Try visiting some web pages and hitting "refresh".'
    web.Application([('.*', ProxyHandler)], debug=True).listen(8888)
    ioloop.IOLoop.instance().start()
```

3.1.4 `Queue` and `Semaphore` example - a parallel web spider

A simple web-spider that crawls all the pages in http://tornadoweb.org.

`spider()` downloads the page at *base_url* and any pages it links to, recursively. It ignores pages that are not beneath *base_url* hierarchically.

This function demos two Toro classes: *JoinableQueue* and *BoundedSemaphore*. The *JoinableQueue* is a work queue; it begins containing only *base_url*, and each discovered URL is added to it. We wait for *join()* to complete before exiting. This ensures that the function as a whole ends when all URLs have been downloaded.

The *BoundedSemaphore* regulates concurrency. We block trying to decrement the semaphore before each download, and increment it after each download completes.

```
import HTMLParser
import time
import urlparse
from datetime import timedelta

from tornado import httpclient, gen, ioloop

import toro


@gen.coroutine
def spider(base_url, concurrency):
    q = toro.JoinableQueue()
    sem = toro.BoundedSemaphore(concurrency)

    start = time.time()
    fetching, fetched = set(), set()

    @gen.coroutine
    def fetch_url():
        current_url = yield q.get()
        try:
            if current_url in fetching:
                return

            print 'fetching', current_url
            fetching.add(current_url)
            urls = yield get_links_from_url(current_url)
            fetched.add(current_url)

            for new_url in urls:
                # Only follow links beneath the base URL
                if new_url.startswith(base_url):
                    yield q.put(new_url)

        finally:
            q.task_done()
            sem.release()

    @gen.coroutine
    def worker():
        while True:
            yield sem.acquire()
            # Launch a subtask
            fetch_url()

    q.put(base_url)

    # Start worker, then wait for the work queue to be empty.
    worker()
    yield q.join(deadline=timedelta(seconds=300))
    assert fetching == fetched
    print 'Done in %d seconds, fetched %s URLs.' % (
        time.time() - start, len(fetched))
```

```
@gen.coroutine
def get_links_from_url(url):
    """Download the page at `url` and parse it for links. Returned links have
    had the fragment after `#` removed, and have been made absolute so, e.g.
    the URL 'gen.html#tornado.gen.coroutine' becomes
    'http://www.tornadoweb.org/en/stable/gen.html'.
    """
    try:
        response = yield httpclient.AsyncHTTPClient().fetch(url)
        print 'fetched', url
        urls = [urlparse.urljoin(url, remove_fragment(new_url))
                for new_url in get_links(response.body)]
    except Exception, e:
        print e, url
        raise gen.Return([])

    raise gen.Return(urls)


def remove_fragment(url):
    scheme, netloc, url, params, query, fragment = urlparse.urlparse(url)
    return urlparse.urlunparse((scheme, netloc, url, params, query, ''))


def get_links(html):
    class URLSeeker(HTMLParser.HTMLParser):
        def __init__(self):
            HTMLParser.HTMLParser.__init__(self)
            self.urls = []

        def handle_starttag(self, tag, attrs):
            href = dict(attrs).get('href')
            if href and tag == 'a':
                self.urls.append(href)

    url_seeker = URLSeeker()
    url_seeker.feed(html)
    return url_seeker.urls


if __name__ == '__main__':
    import logging
    logging.basicConfig()
    loop = ioloop.IOLoop.current()

    def stop(future):
        loop.stop()
        future.result()  # Raise error if there is one

    future = spider('http://www.tornadoweb.org/en/stable/', 10)
    future.add_done_callback(stop)
    loop.start()
```

3.2 `toro` Classes

Contents

- *Primitives*
  - *AsyncResult*
  - *Lock*
  - *RWLock*
  - *Semaphore*
  - *BoundedSemaphore*
  - *Condition*
  - *Event*
- *Queues*
  - *Queue*
  - *PriorityQueue*
  - *LifoQueue*
  - *JoinableQueue*
- *Exceptions*
- *Class relationships*

3.2.1 Primitives

AsyncResult

class `toro.`**`AsyncResult`**(*io_loop=None*)

A one-time event that stores a value or an exception.

The only distinction between AsyncResult and a simple Future is that AsyncResult lets coroutines wait with a deadline. The deadline can be configured separately for each waiter.

An `AsyncResult` instance cannot be reset.

Parameters

- *io_loop*: Optional custom IOLoop.

`get`(*deadline=None*)

Get a value once `set()` is called. Returns a Future.

The Future's result will be the value. The Future raises `toro.Timeout` if no value is set before the deadline.

Parameters

- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

`get_nowait`()

Get the value if ready, or raise `NotReady`.

`set`(*value*)

Set a value and wake up all the waiters.

Lock

class `toro.`**`Lock`**(*io_loop=None*)

A lock for coroutines.

It is created unlocked. When unlocked, *acquire()* changes the state to locked. When the state is locked, yielding *acquire()* waits until a call to *release()*.

The *release()* method should only be called in the locked state; an attempt to release an unlocked lock raises RuntimeError.

When more than one coroutine is waiting for the lock, the first one registered is awakened by *release()*.

acquire() supports the context manager protocol:

```
>>> from tornado import gen
>>> import toro
>>> lock = toro.Lock()
>>>
>>> @gen.coroutine
... def f():
...     with (yield lock.acquire()):
...         assert lock.locked()
...
...     assert not lock.locked()
```

Note: Unlike with the standard threading.Lock, code in a single-threaded Tornado application can check if a *Lock* is *locked()*, and act on that information without fear that another thread has grabbed the lock, provided you do not yield to the IOLoop between checking *locked()* and using a protected resource.

See also:

Lock example - graceful shutdown

Parameters

- *io_loop*: Optional custom IOLoop.

acquire(*deadline=None*)
Attempt to lock. Returns a Future.

The Future raises *toro.Timeout* if the deadline passes.

Parameters

- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

locked()
`True` if the lock has been acquired

release()
Unlock.

If any coroutines are waiting for *acquire()*, the first in line is awakened.

If not locked, raise a RuntimeError.

RWLock

class toro.**RWLock**(*max_readers=1*, *io_loop=None*)
A reader-writer lock for coroutines.

It is created unlocked. When unlocked, *acquire_write()* always changes the state to locked. When unlocked, *acquire_read()* can changed the state to locked, if *acquire_read()* was called max_readers times. When the state is locked, yielding *acquire_read()*/meth:*acquire_write* waits until a call to *release_write()* in case of locking on write, or *release_read()* in case of locking on read.

The *release_read()* method should only be called in the locked-on-read state; an attempt to release an unlocked lock raises RuntimeError.

The *release_write()* method should only be called in the locked on write state; an attempt to release an unlocked lock raises RuntimeError.

When more than one coroutine is waiting for the lock, the first one registered is awakened by *release_read()*/*release_write()*.

acquire_read()/*acquire_write()* support the context manager protocol:

```
>>> from tornado import gen
>>> import toro
>>> lock = toro.RWLock(max_readers=10)
>>>
>>> @gen.coroutine
... def f():
...     with (yield lock.acquire_read()):
...         assert not lock.locked()
...
...     with (yield lock.acquire_write()):
...         assert lock.locked()
...
...     assert not lock.locked()
```

Parameters

- *max_readers*: Optional max readers value, default 1.
- *io_loop*: Optional custom IOLoop.

acquire_read(*deadline=None*)

Attempt to lock for read. Returns a Future.

The Future raises *toro.Timeout* if the deadline passes.

Parameters

- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

acquire_write(**args*, ***kwargs*)

Attempt to lock for write. Returns a Future.

The Future raises *toro.Timeout* if the deadline passes.

Parameters

- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

locked()

`True` if the lock has been acquired

release_read()
Releases one reader.

If any coroutines are waiting for *acquire_read()* (in case of full readers queue), the first in line is awakened.

If not locked, raise a RuntimeError.

release_write()
Releases after write.

The first in queue will be awakened after release.

If not locked, raise a RuntimeError.

Semaphore

class toro.**Semaphore**(*value=1*, *io_loop=None*)
A lock that can be acquired a fixed number of times before blocking.

A Semaphore manages a counter representing the number of release() calls minus the number of acquire() calls, plus an initial value. The acquire() method blocks if necessary until it can return without making the counter negative.

If not given, value defaults to 1.

acquire() supports the context manager protocol:

```
>>> from tornado import gen
>>> import toro
>>> semaphore = toro.Semaphore()
>>>
>>> @gen.coroutine
... def f():
...     with (yield semaphore.acquire()):
...         assert semaphore.locked()
...
...     assert not semaphore.locked()
```

Note: Unlike the standard threading.Semaphore, a *Semaphore* can tell you the current value of its *counter*, because code in a single-threaded Tornado app can check these values and act upon them without fear of interruption from another thread.

See also:

Queue and Semaphore example - a parallel web spider

Parameters

- *value*: An int, the initial value (default 1).
- *io_loop*: Optional custom IOLoop.

acquire(*deadline=None*)
Decrement *counter*. Returns a Future.

Block if the counter is zero and wait for a *release()*. The Future raises *toro.Timeout* after the deadline.

Parameters

- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

counter
An integer, the current semaphore value

locked()
True if *counter* is zero

release()
Increment *counter* and wake one waiter.

wait(*deadline=None*)
Wait for *locked* to be False. Returns a Future.

The Future raises *toro.Timeout* after the deadline.

Parameters

- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

BoundedSemaphore

class toro.**BoundedSemaphore**(*value=1*, *io_loop=None*)
A semaphore that prevents release() being called too often.

A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, `ValueError` is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug.

If not given, *value* defaults to 1.

See also:

Queue and Semaphore example - a parallel web spider

Condition

class toro.**Condition**(*io_loop=None*)
A condition allows one or more coroutines to wait until notified.

Like a standard Condition, but does not need an underlying lock that is acquired and released.

Parameters

- *io_loop*: Optional custom IOLoop.

notify(*n=1*)
Wake up *n* waiters.

Parameters

- *n*: The number of waiters to awaken (default: 1)

notify_all()
Wake up all waiters.

wait(*deadline=None*)
Wait for *notify()*. Returns a Future.

Timeout is executed after a timeout.

Parameters

- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

Event

class toro.**Event**(*io_loop=None*)
An event blocks coroutines until its internal flag is set to True.

Similar to threading.Event.

See also:

Event example - a caching proxy server

Parameters

- *io_loop*: Optional custom IOLoop.

clear()
Reset the internal flag to `False`. Calls to *wait()* will block until *set()* is called.

is_set()
Return `True` if and only if the internal flag is true.

set()
Set the internal flag to `True`. All waiters are awakened. Calling *wait()* once the flag is true will not block.

wait(*deadline=None*)
Block until the internal flag is true. Returns a Future.

The Future raises *Timeout* after a timeout.

Parameters

- *callback*: Function taking no arguments.
- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

3.2.2 Queues

Queue

class toro.**Queue**(*maxsize=0*, *io_loop=None*)
Create a queue object with a given maximum size.

If *maxsize* is 0 (the default) the queue size is unbounded.

Unlike the standard Queue, you can reliably know this Queue's size with *qsize()*, since your single-threaded Tornado application won't be interrupted between calling *qsize()* and doing an operation on the Queue.

Examples:

Producer-consumer example

Queue and Semaphore example - a parallel web spider

Parameters

- *maxsize*: Optional size limit (no limit by default).
- *io_loop*: Optional custom IOLoop.

empty()
Return `True` if the queue is empty, `False` otherwise.

full()
Return `True` if there are *maxsize* items in the queue.

Note: if the Queue was initialized with *maxsize=0* (the default), then *full()* is never `True`.

---

get(*deadline=None*)
Remove and return an item from the queue. Returns a Future.

The Future blocks until an item is available, or raises *toro.Timeout*.

Parameters

- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

get_nowait()
Remove and return an item from the queue without blocking.

Return an item if one is immediately available, else raise `queue.Empty`.

maxsize
Number of items allowed in the queue.

put(*item*, *deadline=None*)
Put an item into the queue. Returns a Future.

The Future blocks until a free slot is available for *item*, or raises *toro.Timeout*.

Parameters

- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

put_nowait(*item*)
Put an item into the queue without blocking.

If no free slot is immediately available, raise queue.Full.

qsize()
Number of items in the queue

PriorityQueue

class toro.**PriorityQueue**(*maxsize=0*, *io_loop=None*)
A subclass of *Queue* that retrieves entries in priority order (lowest first).

Entries are typically tuples of the form: `(priority number, data)`.

Parameters

- *maxsize*: Optional size limit (no limit by default).
- *initial*: Optional sequence of initial items.
- *io_loop*: Optional custom IOLoop.

LifoQueue

class `toro.`**`LifoQueue`**(*maxsize=0*, *io_loop=None*)

A subclass of *Queue* that retrieves most recently added entries first.

Parameters

- *maxsize*: Optional size limit (no limit by default).
- *initial*: Optional sequence of initial items.
- *io_loop*: Optional custom IOLoop.

JoinableQueue

class `toro.`**`JoinableQueue`**(*maxsize=0*, *io_loop=None*)

A subclass of *Queue* that additionally has *task_done()* and *join()* methods.

See also:

Queue and Semaphore example - a parallel web spider

Parameters

- *maxsize*: Optional size limit (no limit by default).
- *initial*: Optional sequence of initial items.
- *io_loop*: Optional custom IOLoop.

`join`(*deadline=None*)

Block until all items in the queue are processed. Returns a Future.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer calls *task_done()* to indicate that all work on the item is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

The Future raises *toro.Timeout* if the count is not zero before the deadline.

Parameters

- *deadline*: Optional timeout, either an absolute timestamp (as returned by `io_loop.time()`) or a `datetime.timedelta` for a deadline relative to the current time.

`task_done`()

Indicate that a formerly enqueued task is complete.

Used by queue consumers. For each *get* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put* into the queue).

Raises `ValueError` if called more times than there were items placed in the queue.

3.2.3 Exceptions

class `toro.`**`Timeout`**
Raised when a deadline passes before a Future is ready.

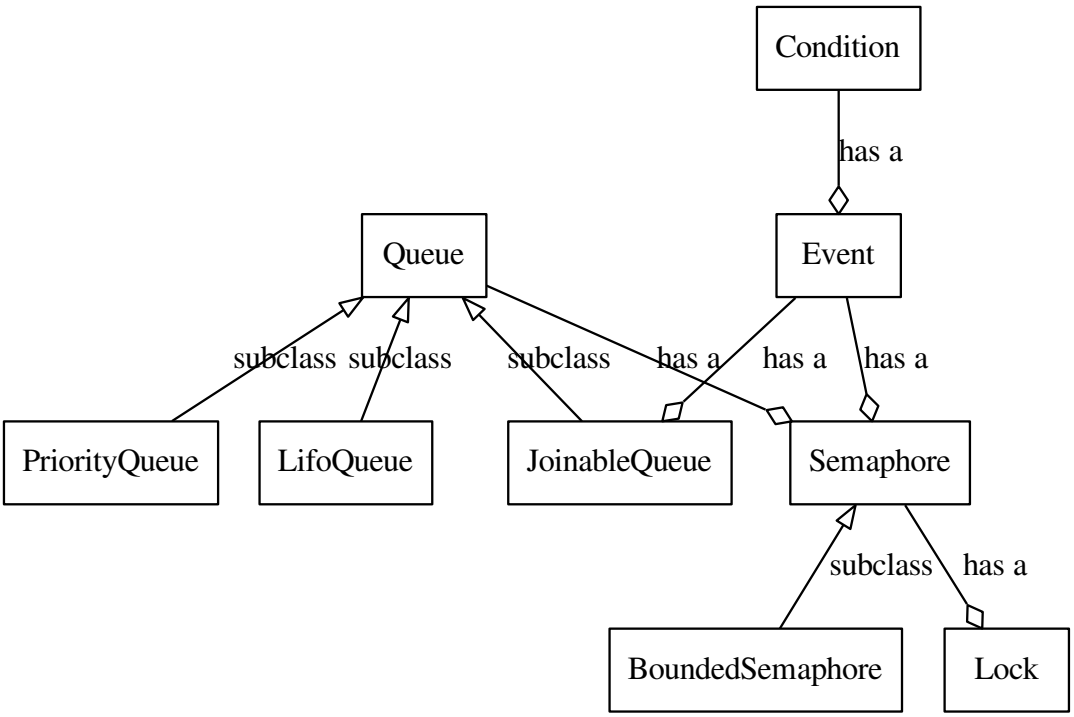
class `toro.`**`NotReady`**
Raised when accessing an *`AsyncResult`* that has no value yet.

class `toro.`**`AlreadySet`**
Raised when setting a value on an *`AsyncResult`* that already has one.

Toro also uses exceptions Empty and Full from the standard module Queue.

3.2.4 Class relationships

Toro uses some of its primitives in the implementation of others. For example, *`JoinableQueue`* is a subclass of *`Queue`*, and it contains an *`Event`*. (*`AsyncResult`* stands alone.)



3.3 Frequently Asked Questions

3.3.1 What's it for?

Toro makes it easy for Tornado coroutines–that is, functions decorated with gen.coroutine–to coordinate using Events, Conditions, Queues, and Semaphores. Toro supports patterns in which coroutines wait for notifications from others.

3.3.2 Why the name?

A coroutine is often called a "coro", and a library of primitives useful for managing coroutines is called "coros" in Gevent and "coro" in Shrapnel. So I call a library to manage Tornado coroutines "toro".

3.3.3 Why do I need synchronization primitives for a single-threaded app?

Protecting an object shared across coroutines is mostly unnecessary in a single-threading Tornado program. For example, a multithreaded app would protect `counter` with a Lock:

```
import threading

lock = threading.Lock()
counter = 0

def inc():
    lock.acquire()
    counter += 1
    lock.release()
```

This isn't needed in a Tornado coroutine, because the coroutine won't be interrupted until it explicitly yields. Thus Toro is *not* designed to protect shared state.

Instead, Toro supports complex coordination among coroutines with *The Wait / Notify Pattern*: Some coroutines wait at particular points in their code for other coroutines to awaken them.

3.3.4 Why no RLock?

The standard-library RLock (reentrant lock) can be acquired multiple times by a single thread without blocking, reducing the chance of deadlock, especially in recursive functions. The thread currently holding the RLock is the "owning thread."

In Toro, simulating a concept like an "owning chain of coroutines" would be over-complicated and under-useful, so there is no RLock, only a *Lock*.

3.3.5 Has Toro anything to do with Tulip?

Toro predates Tulip, which has very similar ideas about coordinating async coroutines using locks and queues. Toro's author implemented Tulip's queues, and version 0.5 of Toro strives to match Tulip's API.

The chief differences between Toro and Tulip are that Toro uses `yield` instead of `yield from`, and that Toro uses absolute deadlines instead of relative timeouts. Additionally, Toro's *Lock* and *Semaphore* aren't context managers (they can't be used with a `with` statement); instead, the Futures returned from *Lock.acquire()* and *Semaphore.acquire()* are context managers:

```
>>> from tornado import gen
>>> import toro
>>> lock = toro.Lock()
>>>
>>> @gen.coroutine
... def f():
...     with (yield lock.acquire()):
...         assert lock.locked()
...
...     assert not lock.locked()
```

3.4 Changelog

3.4.1 Changes in Version 1.0.1

Bug fix in *RWLock*: when max_readers > 1 *release_read()* must release one reader in case *acquire_read()* was called at least once:

```
@gen.coroutine
def coro():
    lock = toro.RWLock(max_readers=10)
    assert not lock.locked()

    yield lock.acquire_read()
    lock.release_read()
```

But, in old version *release_read()* raises RuntimeException if a lock in unlocked state, even if *acquire_read()* was already called several times.

Patch by Alexander Gridnev.

3.4.2 Changes in Version 1.0

This is the final release of Toro. Its features are merged into Tornado 4.2. Further development of locks and queues for Tornado coroutines will continue in Tornado.

For more information on the end of Toro, read my article. The Tornado changelog has comprehensive instructions on porting from Toro's locks and queues to Tornado 4.2 locks and queues.

Toro 1.0 has one new feature, an *RWLock* contributed by Alexander Gridnev. *RWLock* has *not* been merged into Tornado.

3.4.3 Changes in Version 0.8

Don't depend on "nose" for tests. Improve test quality and coverage. Delete unused method in internal `_TimeoutFuture` class.

3.4.4 Changes in Version 0.7

Bug fix in *Semaphore*: after a call to *acquire()*, *wait()* should block until another coroutine calls *release()*:

```
@gen.coroutine
def coro():
    sem = toro.Semaphore(1)
    assert not sem.locked()

    # A semaphore with initial value of 1 can be acquired once,
    # then it's locked.
    sem.acquire()
    assert sem.locked()

    # Wait for another coroutine to release the semaphore.
    yield sem.wait()
```

However, there was a bug and *wait()* returned immediately if the semaphore had **ever** been unlocked. I'm grateful to "abing" on GitHub for noticing the bug and contributing a fix.

3.4.5 Changes in Version 0.6

Queue now supports floating-point numbers for `maxsize`. A `maxsize` of 1.3 is now equivalent to a `maxsize` of 2. Before, it had been treated as infinite.

This feature is not intended to be useful, but to maintain an API similar to `asyncio` and the standard library Queue.

3.4.6 Changes in Version 0.5

Rewritten for Tornado 3.

Dropped support for Tornado 2 and Python 2.5.

Added support for Tornado 3's Futures:

- All Toro methods that took callbacks no longer take callbacks but return Futures.
- All Toro methods that took *optional* callbacks have been split into two methods: one that returns a Future, and a "nowait" method that returns immediately or raises an exception.
  - *AsyncResult.get_nowait()* can raise *NotReady*
  - *Queue.get_nowait()* can raise `Empty`
  - *Queue.put_nowait()* can raise `Full`
- All Toro methods that return Futures accept an optional `deadline` parameter. Whereas before each Toro class had different behavior after a timeout, all now return a Future that raises *toro.Timeout* after the deadline.

Toro's API aims to be very similar to Tulip, since Tulip will evolve into the Python 3.4 standard library:

- Toro's API has been updated to closely match the locks and queues in Tulip.
- The requirement has been dropped that a coroutine that calls *put()* resumes only *after* any coroutine it awakens. Similar for *get()*. The order in which the two coroutines resume is now unspecified.
- A Queue with maxsize 0 (the default) is no longer a "channel" as in Gevent but is an unbounded Queue as in Tulip and the standard library. `None` is no longer a valid maxsize.
- The `initial` argument to Queue() was removed.
- maxsize can no longer be changed after a Queue is created.

The chief differences between Toro and Tulip are that Toro uses `yield` instead of `yield from`, and that Toro uses absolute deadlines instead of relative timeouts. Additionally, Toro's *`Lock`* and *`Semaphore`* aren't context managers (they can't be used with a `with` statement); instead, the Futures returned from *`acquire()`* and *`acquire()`* are context managers.

3.4.7 Changes in Version 0.4

Bugfix in *`JoinableQueue`*, JoinableQueue doesn't accept an explicit IOLoop.

3.4.8 Changes in Version 0.3

Increasing the *`maxsize`* of a *`Queue`* unblocks callbacks waiting on *`put()`*.

Travis integration.

3.4.9 Changes in Version 0.2

Python 3 support.

Bugfix in *`Semaphore`*: `release()` shouldn't wake callbacks registered with `wait()` unless no one is waiting for `acquire()`.

Fixed error in the "Wait-Notify" table.

Added Lock example - graceful shutdown to docs.

3.4.10 Changes in Version 0.1.1

Fixed the docs to render correctly in PyPI.

3.4.11 Version 0.1

First release.

CHAPTER 4

Source

Is on GitHub: https://github.com/ajdavis/toro

CHAPTER 5

Bug Reports and Feature Requests

Also on GitHub: https://github.com/ajdavis/toro/issues

CHAPTER 6

Indices and tables

- genindex
- search

e

examples.event_example, 8
examples.lock_example, 8
examples.producer_consumer_example, 7
examples.web_spider_example, 9

t

toro, 3

A

acquire() (toro.Lock method), 13
acquire() (toro.Semaphore method), 15
acquire_read() (toro.RWLock method), 14
acquire_write() (toro.RWLock method), 14
AlreadySet (class in toro), 20
AsyncResult (class in toro), 12

B

BoundedSemaphore (class in toro), 16

C

clear() (toro.Event method), 17
Condition (class in toro), 16
counter (toro.Semaphore attribute), 16

E

empty() (toro.Queue method), 18
Event (class in toro), 17
examples.event_example (module), 8
examples.lock_example (module), 8
examples.producer_consumer_example (module), 7
examples.web_spider_example (module), 9

F

full() (toro.Queue method), 18

G

get() (toro.AsyncResult method), 12
get() (toro.Queue method), 18
get_nowait() (toro.AsyncResult method), 12
get_nowait() (toro.Queue method), 18

I

is_set() (toro.Event method), 17

J

join() (toro.JoinableQueue method), 19
JoinableQueue (class in toro), 19

L

LifoQueue (class in toro), 19
Lock (class in toro), 12
locked() (toro.Lock method), 13
locked() (toro.RWLock method), 14
locked() (toro.Semaphore method), 16

M

maxsize (toro.Queue attribute), 18

N

notify() (toro.Condition method), 16
notify_all() (toro.Condition method), 16
NotReady (class in toro), 20

P

PriorityQueue (class in toro), 18
put() (toro.Queue method), 18
put_nowait() (toro.Queue method), 18

Q

qsize() (toro.Queue method), 18
Queue (class in toro), 17

R

release() (toro.Lock method), 13
release() (toro.Semaphore method), 16
release_read() (toro.RWLock method), 14
release_write() (toro.RWLock method), 15
RWLock (class in toro), 13

S

Semaphore (class in toro), 15
set() (toro.AsyncResult method), 12
set() (toro.Event method), 17

T

task_done() (toro.JoinableQueue method), 19
Timeout (class in toro), 20

toro (module), 1, 21, 22

W

wait() (toro.Condition method), 16
wait() (toro.Event method), 17
wait() (toro.Semaphore method), 16