Readers/Writer Lock for Twisted Documentation

Release 1.0

Gaetan Semet

October 01, 2016

Twisted implementation of a Readers/Writer Lock.

- License: MIT
- Source: https://github.com/Stibbons/txrwlock
- Overview: http://www.great-a-blog.co/readerswriter-lock-for-twisted/

This synchronization primitive allows to lock a share depending on two access roles: "reader" which only access to the data without modifying it, and "writer" which may want to change the data in the share.

RW Lock features:

- Multiple readers can access to the data at the same time. There is no locking at all when only readers require access to the share
- When a write requires access to the share, it prevents any new reader request to fullfil and put these requests into a waiting queue. It will wait for all ongoing reads to finish
- Only one writer can act at the same time
- This Lock is well suited for share with more readers than writer. Write requests must be at least an order of magnitude less often that read requests

This implementation brings this mechanism to the Twisted's deferred. Please note they are independent with other multithreading RW locks.

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

CHAPTER 2

Source Documentation

2.1 Readers/Writer Deferred Lock

class `txrwlock.txrwlock.`**`ReadersWriterDeferredLock`**

Readers-Writer Lock for Twisted's Deferred

Many readers can simultaneously access a share at the same time, but a writer has an exclusive access to this share.

The following constraints should be met:

1. no reader should be kept waiting if the share is currently not opened to anyone or to only other readers.
2. only one writer can open the share at the same time, and when multiple writer request access, they will waiting for the execution of all previous writer access

Reads and Writes are executed from within the main twisted reactor. Do **NOT** call it from external threads (e.g., from synchronous method execute in thread with `deferToThread`).

Description

- "Readers" uses `readerAcquire` and `readerRelease`.
- "Writer" uses `writerAcquire` and `writerRelease`.

A "reader" is not blocked when one, two or more 'reads' are being executed.

A "reader" is blocked while a 'writer' is executing.

When a "write" starts, it blocks all new 'reads' and wait for the pending 'reads' to finish. If a new 'write" is requested, it will wait for running writes to finish as well.

Notes:

> Please be aware than `ReadersWriterDeferredLock.acquire*` and `ReadersWriterDeferredLock.release*` methods are deferred, which is different from `defer.DeferredLock`, where only the `defer.DeferredLock.acquire()` method is a deferred.

Usage

Threads that just need "read" access, use the following pattern:

```
@defer.inlineCallbacks
def aReaderMethod(...):
    try:
        yield rwlocker.readerAcquire()
        # ... any treatment ...
    finally:
        yield rwlocker.readerRelease()
```

Threads that just need "read" access, use the following pattern:

```
@defer.inlineCallbacks
def aWriterMethod(...):
    try:
        yield rwlocker.writerAcquire()
        # ... any treatment ...
    finally:
        yield rwlocker.writerRelease()
```

Example

```
from twisted.internet import defer
from txrwlock.txrwlock import ReadersWriterDeferredLock

class MySharedObject(object):

    def __init__(self):
        self._readWriteLock = DeferredReadersWriterLock()
        self._data = {}

    @defer.inlineCallbacks
    def performHeavyTreatmentOnData(self):
        try:
            yield rwlocker.readerAcquire()
            # self._data is read and need to stay coherent during the whole current method
            yield anyOtherVeryLongDeferredThatReadsData(self._data)
            # self._data is read again
        finally:
            yield rwlocker.readerRelease()

    @defer.inlineCallbacks
    def changeDataValue(self):
        try:
            yield rwlocker.writerAcquire()
            # Change self._data somehow
        finally:
            yield rwlocker.writerRelease()
```

There could be as many simultanous calls to `MySharedObject.performHeavyTreatmentOnData` at the same time (during `anyOtherVeryLongDeferredThatReadsData`, the reactor might schedule a new call to `MySharedObject.performHeavyTreatmentOnData`). Once `MySharedObject.changeDataValue` is called, all new call to `performHeavyTreatmentOnData` are blocked.

`isReading`

Is the lock acquired for read? (will return false if only required for writer)

`isWriting`

Is the lock acquired for write?

`readerAcquire``()`

Deferred to acquire the lock for a Reader.

Inside an inlineCallback, you need to yield this call.

If the lock has been acquire by only reader, this method will not block. If the lock has been requested by at least one writer, even if this writer is waiting for all ongoing readers to finish, this call will be blocked.

You need to enclose this call inside try/finally to ensure the lock is always released, even in case of exception.

Example:

```
@defer.inlineCallbacks
def aReaderMethod(...):
    try:
        yield rwlocker.readerAcquire()
        # ... any treatment ...
    finally:
        yield rwlocker.readerRelease()
```

readerRelease()

Release the lock by a reader.

Inside an inlineCallback, you need to yield this call.

This call is always non-blocking.

writerAcquire()

Acquire the lock for a Writer.

Inside an inlineCallback, you need to yield this call.

If at least one other reader is ongoing, this call will block any new reader request, and will wait for all reader to finish. If two writers request access to the lock, each one will wait so only one write has the lock at the a time.

You need to enclose this call inside try/finally to ensure the lock is always released, even in case of exception.

Example:

```
@defer.inlineCallbacks
def aWriterMethod(...):
    try:
        yield rwlocker.writerAcquire()
        # ... any treatment ...
    finally:
        yield rwlocker.writerRelease()
```

writerRelease()

Release the lock by a Writer.

Inside an inlineCallback, you need to yield this call.

This call is always non-blocking

2.2 Readers/Writer Deferred Lock TestCase

class txrwlock.txrwlocktestcase.**TxRWLockTestCase**(*methodName='runTest'*)

Unit test helper class for Twisted.

Provides useful methods to test exception cases, such as *assertRaisesWithMessage* and *assertInlineCallbacksRaisesWithMessage* in addition to *twisted.trial.unittest.TestCase*.

assertInlineCbRaises(*exceptionClass*, *inlineCallbacksFunc*, **args*, ***kw*)
Assert a given inlineCallbacks decorated method raises.

This replaces assertRaisesWithMessage for inlineCallbacks.

Note: this method is an inlineCallbacks and need to be yielded.

assertInlineCbRaisesWithMsg(*exceptionClass*, *expectedMessage*, *inlineCallbacksFunc*, **args*, ***kw*)
Assert a given inlineCallbacks decorated method raises with a given message.

This replaces assertRaisesWithMessage for inlineCallbacks.

Note: this method is an inlineCallbacks and need to be yielded.

assertRaisesWithMessage(*exceptionClass*, *expectedMessage*, *func*, **args*, ***kw*)
Check if a given function call (synchronous or deferred) raised with a given message.

Note: You cannot use an inlineCallbacks as func. Please use assertInlineCallbacksRaisesWithMessage.

Index