dirq Documentation

Release 1.2.2

Konstantin Skaburskas

February 04, 2013

CONTENTS

Contents:

CHAPTER

ONE

QUEUESIMPLE DOCUMENTATION

QueueSimple - object oriented interface to a simple directory based queue.

A port of Perl module Directory::Queue::Simple http://search.cpan.org/~lcons/Directory-Queue/ The documentation from Directory::Queue::Simple module was adapted for Python.

1.1 QueueSimple class

`QueueSimple` - simple directory based queue.

Usage:

```
from dirq.QueueSimple import QueueSimple

# sample producer

dirq = QueueSimple('/tmp/test')
for count in range(1,101):
    name = dirq.add("element %i\n" % count)
    print("# added element %i as %s" % (count, name))

# sample consumer

dirq = QueueSimple('/tmp/test')
for name in dirq:
    if not dirq.lock(name):
        continue
    print("# reading element %s" % name)
    data = dirq.get(name)
    # one could use dirq.unlock(name) to only browse the queue...
    dirq.remove(name)
```

1.1.1 Description

This module is very similar to dirq.queue, but uses a different way to store data in the filesystem, using less directories. Its API is almost identical.

Compared to `dirq.queue`, this module:

- is simpler
- is faster

- uses less space on disk
- can be given existing files to store
- does not support schemas
- can only store and retrieve byte strings
- is not compatible (at filesystem level) with Queue

Please refer to `dirq.queue` for general information about directory queues.

1.1.2 Directory Structure

The toplevel directory contains intermediate directories that contain the stored elements, each of them in a file.

The names of the intermediate directories are time based: the element insertion time is used to create a 8-digits long hexadecimal number. The granularity (see the constructor) is used to limit the number of new directories. For instance, with a granularity of 60 (the default), new directories will be created at most once per minute.

Since there is usually a filesystem limit in the number of directories a directory can hold, there is a trade-off to be made. If you want to support many added elements per second, you should use a low granularity to keep small directories. However, in this case, you will create many directories and this will limit the total number of elements you can store.

The elements themselves are stored in files (one per element) with a 14-digits long hexadecimal name SSSSSSSSMMMMMR where:

- SSSSSSSS represents the number of seconds since the Epoch
- MMMMM represents the microsecond part of the time since the Epoch
- R is a random digit used to reduce name collisions

A temporary element (being added to the queue) will have a *.tmp* suffix.

A locked element will have a hard link with the same name and the *.lck* suffix.

1.1.3 Author

Konstantin Skaburskas <konstantin.skaburskas@gmail.com>

1.1.4 License and Copyright

class `dirq.QueueSimple.`**`QueueSimple`**(*path*, *umask=None*, *granularity=60*)
QueueSimple

`add`(*data*)
Add data to the queue as a file.

Return: element name (<directory name>/<file name>).

add_path(*path*)
Add the given file (identified by its path) to the queue and return the corresponding element name, the file must be on the same filesystem and will be moved to the queue

add_ref(*data*)
Defined to comply with Directory::Queue interface.

count()
Return the number of elements in the queue, locked or not (but not temporary).

get(*name*)
Get locked element.

get_path(*name*)
Return the path given the name.

get_ref(*name*)
Get locked element. Defined to comply with Directory::Queue interface.

lock(*name*, *permissive=True*)
Lock an element.

Arguments: name - name of an element permissive - work in permissive mode

Return:

- true on success
- false in case the element could not be locked (in permissive mode)

purge(*maxtemp=300*, *maxlock=600*)
Purge the queue by removing unused intermediate directories, removing too old temporary elements and unlocking too old locked elements (aka staled locks); note: this can take a long time on queues with many elements.

maxtemp - maximum time for a temporary element (in seconds, default 300); if set to 0, temporary elements will not be removed

maxlock - maximum time for a locked element (in seconds, default 600); if set to 0, locked elements will not be unlocked

remove(*name*)
Remove a locked element from the queue.

unlock(*name*, *permissive=False*)
Unlock an element.

Arguments: name - name of an element permissive - work in permissive mode

Return:

- true on success
- false in case the element could not be unlocked (in permissive mode)

CHAPTER

TWO

QUEUE DOCUMENTATION

Directory based queue.

A port of Perl module Directory::Queue http://search.cpan.org/~lcons/Directory-Queue/ The documentation from Directory::Queue module was adapted for Python.

The goal of this module is to offer a simple queue system using the underlying filesystem for storage, security and to prevent race conditions via atomic operations. It focuses on simplicity, robustness and scalability.

This module allows multiple concurrent readers and writers to interact with the same queue.

2.1 Provides

Classes:

- `dirq.queue.Queue` directory based queue
- `dirq.QueueSimple.QueueSimple` simple directory based queue
- `dirq.queue.QueueSet` set of directory based queues
- `dirq.Exceptions.QueueError` exception

2.2 Documentation

2.2.1 Queue class

`dirq.queue.Queue` - directory based queue.

Usage:

```
from dirq.queue import Queue

# simple schema:
#  - there must be a "body" which is a string
#  - there can be a "header" which is a table/dictionary

schema = {"body": "string", "header": "table?"}
queuedir = "/tmp/test"

# sample producer
```

```
dirq = Queue(queuedir, schema=schema)
import os
for count in range(1,101):
    name = dirq.add({"body"   : "element %i"%count,
                     "header": dict(os.environ)})
    print("# added element %i as %s" % (count, name))

# sample consumer

dirq = Queue(queuedir, schema=schema)
name = dirq.first()
while name:
    if not dirq.lock(name):
        name = dirq.next()
        continue
    print("# reading element %s" % name)
    data = dirq.get(name)
    # one can use data['body'] and data['header'] here...
    # one could use dirq.unlock(name) to only browse the queue...
    dirq.remove(name)
    name = dirq.next()
```

Terminology

An element is something that contains one or more pieces of data. A simple string may be an element but more complex schemas can also be used, see the *Schema* section for more information.

A queue is a "best effort FIFO" collection of elements.

It is very hard to guarantee pure FIFO behavior with multiple writers using the same queue. Consider for instance:

. Writer1: calls the add() method . Writer2: calls the add() method . Writer2: the add() method returns . Writer1: the add() method returns

Who should be first in the queue, Writer1 or Writer2?

For simplicity, this implementation provides only "best effort FIFO", i.e. there is a very high probability that elements are processed in FIFO order but this is not guaranteed. This is achieved by using a high-resolution time function and having elements sorted by the time the element's final directory gets created.

Locking

Adding an element is not a problem because the add() method is atomic.

In order to support multiple processes interacting with the same queue, advisory locking is used. Processes should first lock an element before working with it. In fact, the get() and remove() methods raise an exception if they are called on unlocked elements.

If the process that created the lock dies without unlocking the ele- ment, we end up with a staled lock. The purge() method can be used to remove these staled locks.

An element can basically be in only one of two states: locked or unlocked.

A newly created element is unlocked as a writer usually does not need to do anything more with the element once dropped in the queue.

Iterators return all the elements, regardless of their states.

There is no method to get an element state as this information is usu- ally useless since it may change at any time. Instead, programs should directly try to lock elements to make sure they are indeed locked.

Constructor

For the signature of the Queue constructor see documentation to the respective __init__() method.

Schema

The schema defines how user supplied data is stored in the queue. It is only required by the add() and get() methods.

The schema must be a dictionary containing key/value pairs.

The key must contain only alphanumerical characters. It identifies the piece of data and will be used as file name when storing the data inside the element directory.

The value represents the type of the given piece of data. It can be:

binary the data is a sequence of binary bytes, it will be stored directly in a plain file with no further encoding

string the data is a text string (i.e. a sequence of characters), it will be UTF-8 encoded

table the data is a reference to a hash of text strings, it will be seri- alized and UTF-8 encoded before being stored in a file

By default, all pieces of data are mandatory. If you append a question mark to the type, this piece of data will be marked as optional. See the comments in the *Usage* section for more information.

To comply with Directory::Queue implementation it is allowed to append '*' (asterisk) to data type specification, which in Directory::Queue means switching to working with element references in add() and get() operations. This is irrelevant for the Python implementation.

Directory Structure

All the directories holding the elements and all the files holding the data pieces are located under the queue toplevel directory. This direc- tory can contain:

temporary the directory holding temporary elements, i.e. the elements being added

obsolete the directory holding obsolete elements, i.e. the elements being removed

NNNNNNNN an intermediate directory holding elements; NNNNNNNN is an 8-digits long hexadecimal number

In any of the above directories, an element is stored as a single directory with a 14-digits long hexadecimal name SSSSSSSSMMMMMR where:

SSSSSSSS represents the number of seconds since the Epoch

MMMMM represents the microsecond part of the time since the Epoch

R is a random digit used to reduce name collisions

Finally, inside an element directory, the different pieces of data are stored into different files, named according to the schema. A locked element contains in addition a directory named "locked".

Security

There are no specific security mechanisms in this module.

The elements are stored as plain files and directories. The filesystem security features (owner, group, permissions, ACLs...) should be used to adequately protect the data.

By default, the process' umask is respected. See the class constructor documentation if you want an other behavior.

If multiple readers and writers with different uids are expected, the easiest solution is to have all the files and directories inside the toplevel directory world-writable (i.e. umask=0). Then, the permissions of the toplevel directory itself (e.g. group-writable) are enough to control who can access the queue.

2.2.2 QueueSet class

`dirq.queue.QueueSet` - interface to a set of Queue objects

Usage:

```
from dirq.queue import Queue, QueueSet

dq1 = Queue("/tmp/q1")
dq2 = Queue("/tmp/q2")
dqset = QueueSet(dq1, dq2)
# dqs = [dq1, dq2]
# dqset = QueueSet(dqs)

(dq, elt) = dqset.first()
while dq:
    # you can now process the element elt of queue dq...
    (dq, elt) = dqset.next()
```

Description

This class can be used to put different queues into a set and browse them as one queue. The elements from all queues are merged together and sorted independently from the queue they belong to.

Constructor

For the signature of the QueueSet constructor see documentation to the respective `dirq.queue.QueueSet.__init__()` method.

Author

Konstantin Skaburskas <konstantin.skaburskas@gmail.com>

License and Copyright

class dirq.queue.**Queue**(*path*, *umask=None*, *maxelts=16000*, *schema={}*)
Directory based queue.

add(*data*)
Add a new element to the queue and return its name. Arguments:

data - element as a dictionary (should conform to the schema)

Raise:

QueueError - problem with schema definition or data OSError - problem putting element on disk

Note:

the destination directory must _not_ be created beforehand as it would be seen as a valid (but empty) element directory by another process, we therefore use rename() from a temporary directory

count()
Return the number of elements in the queue, regardless of their state.

Raise: OSError - can't list/stat element directories

dequeue(*ename*, *permissive=True*)
Dequeue an element from the queue. Removes element from the queue. Performs operations: lock(name), get(name), remove(name)

Arguments: ename - name of an element

Return: dictionary representing an element

Raise: QueueLockError - coulnd't lock element QueueError - problems with schema/data types/etc. OSError - problems opening/closing directory/file

enqueue(*data*)
Add a new element to the queue and return its name. Arguments:

data - element as a dictionary (should conform to the schema)

Raise:

QueueError - problem with schema definition or data OSError - problem putting element on disk

Note:

the destination directory must _not_ be created beforehand as it would be seen as a valid (but empty) element directory by another process, we therefore use rename() from a temporary directory

get(*ename*)
Get an element data from a locked element.

Arguments: ename - name of an element

Return: dictionary representing an element

Raise:

QueueError - schema is unknown; unexpected data type in the schema specification; missing mandatory file of the element

OSError - problems opening/closing file IOError - file read error

get_element(*ename*, *permissive=True*)
Get an element from the queue. Element will not be removed. Operations performed: lock(name), get(name), unlock(name)

Arguments: ename - name of an element

Raise: QueueLockError - couldn't lock element

lock (*ename*, *permissive=True*)
Lock an element.

Arguments: ename - name of an element permissive - work in permissive mode

Return:

- True on success
- False in case the element could not be locked (in permissive mode)

Raise: QueueError - invalid element name OSError - can't create lock (mkdir()/lstat() failed)

Note:

- locking can fail:
  - if the element has been locked by somebody else (EEXIST)
  - if the element has been removed by somebody else (ENOENT)
- if the optional second argument is true, it is not an error if the element cannot be locked (permissive mode), this is the default
- the directory's mtime will change automatically (after a successful mkdir()), this will later be used to detect stalled locks

purge (*maxtemp=300*, *maxlock=600*)
Purge the queue:

- delete unused intermediate directories
- delete too old temporary directories
- unlock too old locked directories

Arguments:

maxtemp - maximum time for a temporary element. If 0, temporary elements will not be removed.

maxlock - maximum time for a locked element. If 0, locked elements will not be unlocked.

Raise: OSError - problem deleting element from disk

Note: this uses first()/next() to iterate so this will reset the cursor

remove (*ename*)
Remove locked element from the queue.

Arguments: ename - name of an element

Raise:

QueueError - invalid element name; element not locked; unexpected file in the element directory

OSError - can't rename/remove a file/directory

Note: doesn't return anything explicitly (i.e. returns NoneType) or fails

unlock (*ename*, *permissive=False*)
Unlock an element.

Arguments: ename - name of an element permissive - work in permissive mode

Return:

- true on success
- false in case the element could not be unlocked (in permissive mode)

Raise: QueueError - invalid element name OSError - can't remove lock (rmdir() failed)

Note:

- unlocking can fail:
  - if the element has been unlocked by somebody else (ENOENT)
  - if the element has been removed by somebody else (ENOENT)
- if the optional second argument is true, it is not an error if the element cannot be unlocked (permissive mode), this is _not_ the default

class dirq.queue.**QueueSet**(**queues*)

Interface to elements on a set of directory based queues.

add(**queues*)

Add lists of queues to existing ones. Copies of the object instances are used.

Arguments: *queues - add([q1,..]/(q1,..)) or add(q1,..)

Raise: QueueError - queue already in the set TypeError - wrong queue object type provided

count()

Return the number of elements in the queue set, regardless of their state.

Raise: OSError - can't list/stat element directories

first()

Return the first element in the queue set and cache information about the next ones.

Raise: OSError - can't list directories

names()

Return iterator over element names on the set of queues.

next()

Return (queue, next element) tuple from the queue set, only using cached information.

Raise:

StopIteration - when used as Python iterator via __iter__() method

OSError - can't list element directories

remove(*given_queue*)

Remove a queue and its respective elements from in memory cache.

Arguments: queue - queue to be removed

Raise: TypeError - wrong queue object type provided

exception dirq.queue.**QueueError**

QueueError

CHAPTER

THREE

QUEUENULL DOCUMENTATION

QueueNull - object oriented interface to a null directory based queue.

A port of Perl module Directory::Queue::Null http://search.cpan.org/~lcons/Directory-Queue/ The documentation from Directory::Queue::Null module was adapted for Python.

3.1 QueueNull class

`QueueNull` - null directory based queue.

Usage:

```
from dirq.QueueNull import QueueNull

# sample producer

dirq = QueueNull()
for count in range(1,101):
    name = dirq.add("element %i\n" % count)
```

3.1.1 Description

The goal of this module is to offer a "null" queue system using the same API as the other directory queue implementations. The queue will behave like a black hole: added data will disappear immediately so the queue will therefore always appear empty.

This can be used for testing purposes or to discard data like one would do on Unix by redirecting output to */dev/null*.

Please refer to `dirq.queue` for general information about directory queues.

3.1.2 Author

Konstantin Skaburskas <konstantin.skaburskas@gmail.com>

3.1.3 License and Copyright

ASL 2.0

Copyright (C) 2010-2012

class dirq.QueueNull.**QueueNull**
QueueNull

add(*data*)
Add data to the queue, this does nothing.

add_path(*path*)
Add the given file (identified by its path) to the queue, this will therefore *remove the file*.

add_ref(*data*)
Defined to comply with Directory::Queue interface.

count()
Return the number of elements in the queue, which means it always return 0.

get(*name*)
Not supported method.

get_path(*name*)
Not supported method.

get_ref(*name*)
Get locked element. Defined to comply with Directory::Queue interface.

lock(*name*, *permissive=True*)
Not supported method.

purge(*maxtemp=300*, *maxlock=600*)
Purge the queue, this does nothing.

remove(*name*)
Not supported method.

unlock(*name*, *permissive=False*)
Not supported method.

CHAPTER

FOUR

QUEUEBASE DOCUMENTATION

Base class and common code for `dirq` package.

It is used internally by `dirq` modules and should not be used elsewhere.

4.1 Author

Konstantin Skaburskas <konstantin.skaburskas@gmail.com>

4.2 License and Copyright

ASL 2.0

Copyright (C) 2010-2012

class `dirq.QueueBase.`**`QueueBase`**(*path*, *umask=None*)

QueueBase

`copy`()

Copy/clone the object. Return copy of the object.

Note:

- the main purpose is to copy/clone the iterator cached state
- the other structured attributes (including schema) are not cloned

`first`()

Return the first element in the queue and cache information about the next ones.

Raise: OSError - can't list directories

`names`()

Return iterator over element names.

`next`()

Return name of the next element in the queue, only using cached information. When queue is empty, depending on the iterator protocol - return empty string or raise StopIteration.

Return: name of the next element in the queue

Raise:

StopIteration - when used as Python iterator via __iter__() method

OSError - can't list element directories

touch(*ename*)

Touch an element directory to indicate that it is still being used.

Note: this is only really useful for locked elements but we allow it for all.

Raises: EnvironmentError - on any IOError, OSError in utime()

TODO: this may not work on OSes with directories implemented not as files (eg. Windows). See doc for os.utime().

CHAPTER

FIVE

EXCEPTIONS DOCUMENTATION

Exceptions used in the module.

5.1 Author

Konstantin Skaburskas <konstantin.skaburskas@gmail.com>

5.2 License and Copyright

exception dirq.Exceptions.**QueueError**
QueueError

exception dirq.Exceptions.**QueueLockError**
QueueLockError

Directory based queue.

A port of Perl module Directory::Queue http://search.cpan.org/~lcons/Directory-Queue/ The documentation from Directory::Queue module was adapted for Python.

The goal of this module is to offer a simple queue system using the underlying filesystem for storage, security and to prevent race conditions via atomic operations. It focuses on simplicity, robustness and scalability.

This module allows multiple concurrent readers and writers to interact with the same queue.

For usage and implementation details see dirq.queue module.

CHAPTER

SIX

AUTHOR

Konstantin Skaburskas <konstantin.skaburskas@gmail.com>

CHAPTER

SEVEN

LICENSE AND COPYRIGHT

ASL 2.0

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

d