retools Documentation

Release 0.4.1

Ben Bangert

November 12, 2014

Contents

retools is a concise set of well-tested extensible Python Redis tools.

retools is available on PyPI at https://pypi.python.org/pypi/retools

- **`Caching`**
  - Hit/Miss Statistics
  - Regions for common expiration periods and invalidating batches of functions at once.
  - Write-lock to prevent the Thundering Herd
- **`Distributed Locking`**
  - Python context-manager with lock timeouts and retries
- **`Queuing`**
  - Simple *forking worker* based on Resque
  - Jobs stored as JSON in Redis for easy introspection
  - setproctitle used by workers for easy worker introspection on the command line
  - *Rich event system* for extending job processing behavior
- **`Limiter`**
  - Useful for making sure that only N operations for a given process happen at the same time
- **Well Tested** [1]
  - 100% statement coverage
  - 100% condition coverage (via instrumental)

CHAPTER 1

Reference Material

Reference material includes documentation for every *retools* API.

1.1 API Documentation

Comprehensive reference material for every public API exposed by `retools` is available within this chapter. The API documentation is organized alphabetically by module name.

1.1.1 retools.cache

Caching

Cache regions are used to simplify common expirations and group function caches.

To indicate functions should use cache regions, apply the decorator:

```
from retools.cache import cache_region

@cache_region('short_term')
def myfunction(arg1):
    return arg1
```

To configure the cache regions, setup the `CacheRegion` object:

```
from retools.cache import CacheRegion

CacheRegion.add_region("short_term", expires=60)
```

Constants

`retools.cache.`**`NoneMarker`**
: A module global returned to indicate no value is present in Redis rather than a `None` object.

Functions

`retools.cache.`**`cache_region`**(*region*, **deco_args*, ***kwargs*)
: Decorate a function such that its return result is cached, using a "region" to indicate the cache arguments.

Parameters

- **region** (*string*) – Name of the region to cache to
- ***deco_args** – Optional `str()`-compatible arguments which will uniquely identify the key used by this decorated function, in addition to the positional arguments passed to the function itself at call time. This is recommended as it is needed to distinguish between any two functions or methods that have the same name (regardless of parent class or not).

Note: The function being decorated must only be called with positional arguments, and the arguments must support being stringified with `str()`. The concatenation of the `str()` version of each argument, combined with that of the `*args` sent to the decorator, forms the unique cache key.

Example:

```
from retools.cache import cache_region

@cache_region('short_term', 'load_things')
def load(search_term, limit, offset):
    '''Load from a database given a search term, limit, offset.'''
    return database.query(search_term)[offset:offset + limit]
```

The decorator can also be used with object methods. The `self` argument is not part of the cache key. This is based on the actual string name `self` being in the first argument position:

```
class MyThing(object):
    @cache_region('short_term', 'load_things')
    def load(self, search_term, limit, offset):
        '''Load from a database given a search term, limit, offset.'''
        return database.query(search_term)[offset:offset + limit]
```

Classmethods work as well - use `cls` as the name of the class argument, and place the decorator around the function underneath `@classmethod`:

```
class MyThing(object):
    @classmethod
    @cache_region('short_term', 'load_things')
    def load(cls, search_term, limit, offset):
        '''Load from a database given a search term, limit, offset.'''
        return database.query(search_term)[offset:offset + limit]
```

Note: When a method on a class is decorated, the `self` or `cls` argument in the first position is not included in the "key" used for caching.

`retools.cache.`**`invalidate_region`**(*region*)

Invalidate all the namespace's in a given region

Note: This does not actually *clear* the region of data, but just sets the value to expire on next access.

Parameters **region** (*string*) – Region name

`retools.cache.`**`invalidate_function`**(*callable*, **args*)

Invalidate the cache for a callable

Parameters

- **callable** (*callable object*) – The callable that was cached

- ***args** – Arguments the function was called with that should be invalidated. If the args is just the differentiator for the function, or not present, then all values for the function will be invalidated.

Example:

```
@cache_region('short_term', 'small_engine')
def local_search(search_term):
    # do search and return it

@cache_region('long_term')
def lookup_folks():
    # look them up and return them

# To clear local_search for search_term = 'fred'
invalidate_function(local_search, 'fred')

# To clear all cached variations of the local_search function
invalidate_function(local_search)

# To clear out lookup_folks
invalidate_function(lookup_folks)
```

Classes

class retools.cache.**CacheKey**(*region*, *namespace*, *key*, *today=None*)
Cache Key object

Generator of cache keys for a variety of purposes once provided with a region, namespace, and key (args).

class retools.cache.**CacheRegion**
CacheRegion manager and configuration object

For organization sake, the CacheRegion object is used to configure the available cache regions, query regions for currently cached keys, and set batches of keys by region for immediate expiration.

Caching can be turned off globally by setting enabled to False:

```
CacheRegion.enabled = False
```

Statistics should also be turned on or off globally:

```
CacheRegion.statistics = False
```

However, if only some namespaces should have statistics recorded, then this should be used directly.

classmethod add_region(*name*, *expires*, *redis_expiration=604800*)
Add a cache region to the current configuration

Parameters

- **name** (*string*) – The name of the cache region
- **expires** (*integer*) – The expiration in seconds.
- **redis_expiration** (*integer*) – How long the Redis key expiration is set for. Defaults to 1 week.

classmethod invalidate(*region*)
Invalidate an entire region

Note: This does not actually *clear* the region of data, but just sets the value to expire on next access.

Parameters **region** (*string*) – Region name

classmethod `load`(*region*, *namespace*, *key*, *regenerate=True*, *callable=None*, *statistics=None*)
Load a value from Redis, and possibly recreate it

This method is used to load a value from Redis, and usually regenerates the value using the callable when provided.

If `regenerate` is `False` and a `callable` is not passed in, then `NoneMarker` will be returned.

Parameters

- **region** (*string*) – Region name
- **namespace** (*string*) – Namespace for the value
- **key** (*string*) – Key for this value under the namespace
- **regenerate** (*bool*) – If False, then existing keys will always be returned regardless of cache expiration. In the event that there is no existing key and no callable was provided, then a NoneMarker will be returned.
- **callable** – A callable to use when the cached value needs to be created
- **statistics** (*bool*) – Whether or not hit/miss statistics should be updated

1.1.2 `retools.exc`

retools exceptions

Exceptions

exception `retools.exc.`**`RetoolsException`**
retools package base exception

exception `retools.exc.`**`ConfigurationError`**
Raised for general configuration errors

exception `retools.exc.`**`CacheConfigurationError`**
Raised when there's a cache configuration error

exception `retools.exc.`**`QueueError`**
Raised when there's an error in the queue code

exception `retools.exc.`**`AbortJob`**
Raised to abort execution of a job

1.1.3 `retools.limiter`

Generic Limiter to ensure N parallel operations

Note: The limiter functionality is new. Please report any issues found on the retools Github issue tracker.

The limiter is useful when you want to make sure that only N operations for a given process happen at the same time, i.e.: concurrent requests to the same domain.

The limiter works by acquiring and releasing limits.

Creating a limiter:

```
from retools.limiter import Limiter

def do_something():
    limiter = Limiter(limit=10, prefix='my-operation')  # using default redis connection

    for i in range(100):
        if limiter.acquire_limit('operation-%d' % i):
            execute_my_operation()
            limiter.release_limit('operation-%d' % i)  # since we are releasing it synchronously
                                                       # all the 100 operations will be performed wi
                                                       # one of them locked at a time
```

Specifying a default expiration in seconds:

```
def do_something():
    limiter = Limiter(limit=10, expiration_in_seconds=45)  # using default redis connection
```

Specifying a redis connection:

```
def do_something():
    limiter = Limiter(limit=10, redis=my_redis_connection)
```

Every time you try to acquire a limit, the expired limits you previously acquired get removed from the set.

This way if your process dies in the mid of its operation, the keys will eventually expire.

Public API Classes

class retools.limiter.**Limiter**(*limit*, *redis=None*, *prefix='retools_limiter'*, *expiration_in_seconds=10*)

Configures and limits operations

__init__(*limit*, *redis=None*, *prefix='retools_limiter'*, *expiration_in_seconds=10*)

Initializes a Limiter.

Parameters

- **limit** – An integer that describes the limit on the number of items
- **redis** – A Redis instance. Defaults to the redis instance on the global_connection.
- **prefix** – The default limit set name. Defaults to 'retools_limiter'.
- **expiration_in_seconds** – The number in seconds that keys should be locked if not explicitly released.

acquire_limit(*key*, *expiration_in_seconds=None*, *retry=True*)

Tries to acquire a limit for a given key. Returns True if the limit can be acquired.

Parameters

- **key** – A string with the key to acquire the limit for. This key should be used when releasing.
- **expiration_in_seconds** – The number in seconds that this key should be locked if not explicitly released. If this is not passed, the default is used.
- **key** – Internal parameter that specifies if the operation should be retried. Defaults to True.

release_limit(*key*)
Releases a limit for a given key.

Parameters **key** – A string with the key to release the limit on.

1.1.4 retools.lock

A Redis backed distributed global lock

This code uses the formula here: https://github.com/jeffomatic/redis-exp-lock-js

It provides several improvements over the original version based on: http://chris-lamb.co.uk/2010/06/07/distributing-locking-python-and-redis/

It provides a few improvements over the one present in the Python redis library, for example since it utilizes the Lua functionality, it no longer requires every client to have synchronized time.

Classes

class retools.lock.**Lock**(*key*, *expires=60*, *timeout=10*, *redis=None*)

__init__(*key*, *expires=60*, *timeout=10*, *redis=None*)
Distributed locking using Redis Lua scripting for CAS operations.

Usage:

```
with Lock('my_lock'):
    print "Critical section"
```

Parameters

- **expires** – We consider any existing lock older than `expires` seconds to be invalid in order to detect crashed clients. This value must be higher than it takes the critical section to execute.
- **timeout** – If another client has already obtained the lock, sleep for a maximum of `timeout` seconds before giving up. A value of 0 means we never wait.
- **redis** – The redis instance to use if the default global redis connection is not desired.

Exceptions

exception retools.lock.**LockTimeout**
Raised in the event a timeout occurs while waiting for a lock

1.1.5 retools.queue

Queue worker and manager

Note: The queueing functionality is new, and has gone through some preliminary testing. Please report any issues found on the retools Github issue tracker.

Any function that takes keyword arguments can be a `job` that a worker runs. The `QueueManager` handles configuration and enqueing jobs to be run.

Declaring jobs:

```
# mypackage/jobs.py

# jobs

def default_job():
    # do some basic thing

def important(somearg=None):
    # do an important thing

# event handlers

def my_event_handler(sender, **kwargs):
    # do something

def save_error(sender, **kwargs):
    # record error
```

Running Jobs:

```
from retools.queue import QueueManager

qm = QueueManager()
qm.subscriber('job_failure', handler='mypackage.jobs:save_error')
qm.subscriber('job_postrun', 'mypackage.jobs:important',
              handler='mypackage.jobs:my_event_handler')
qm.enqueue('mypackage.jobs:important', somearg='fred')
```

Note: The events for a job are registered with the `QueueManager` and are encoded in the job's JSON blob. Updating events for a job will therefore only take effect for new jobs queued, and not existing ones on the queue.

Events

The retools queue has events available for additional functionality without having to subclass or directly extend retools. These functions will be run by the worker when the job is handled.

Available events to register for:

- **job_prerun**: Runs immediately before the job is run.
- **job_wrapper**: Wraps the execution of the job, these should be context managers.
- **job_postrun**: Runs after the job completes *successfully*, this will not be run if the job throws an exception.
- **job_failure**: Runs when a job throws an exception.

Event Function Signatures

Event functions have different call semantics, the following is a list of how the event functions will be called:

- **job_prerun**: (job=job_instance)
- **job_wrapper**: (job_function, job_instance, **job_keyword_arguments)
- **job_postrun**: (job=job_instance, result=job_function_result)
- **job_failure**: (job=job_instance, exc=job_exception)

Attributes of interest on the job instance are documented in the `Job.__init__()` method.

Running the Worker

After installing `retools`, a `retools-worker` command will be available that can spawn a worker. Queues to watch can be listed in order for priority queueing, in which case the worker will try each queue in order looking for jobs to process.

Example invocation:

```
$ retools-worker high,medium,main
```

Public API Classes

class `retools.queue.`**`QueueManager`**(*redis=None*, *default_queue_name='main'*, *serializer=<function dumps at 0x7fc895e94ed8>*, *deserializer=<function loads at 0x7fc895e94e60>*)

Configures and enqueues jobs

`__init__`(*redis=None*, *default_queue_name='main'*, *serializer=<function dumps at 0x7fc895e94ed8>*, *deserializer=<function loads at 0x7fc895e94e60>*)

Initialize a QueueManager

Parameters

- **redis** – A Redis instance. Defaults to the redis instance on the global_connection.
- **default_queue_name** – The default queue name. Defaults to 'main'.
- **serializer** – A callable to serialize json data, defaults to json.dumps().
- **deserializer** – A callable to deserialize json data, defaults to json.loads().

`enqueue`(*job*, ***kwargs*)

Enqueue a job

Parameters

- **job** – The pkg_resouce name of the function. I.e. retools.jobs:my_function
- **kwargs** – Keyword arguments the job should be called with. These arguments must be serializeable by JSON.

Returns The job id that was queued.

`set_queue_for_job`(*job_name*, *queue_name*)

Set the queue that a given job name will go to

Parameters

- **job_name** – The pkg_resource name of the job function. I.e. retools.jobs:my_function
- **queue_name** – Name of the queue on Redis job payloads should go to

`subscriber`(*event*, *job=None*, *handler=None*)

Set events for a specific job or for all jobs

Parameters

- **event** – The name of the event to subscribe to.
- **job** – Optional, a specific job to bind to.
- **handler** – The location of the handler to call.

Private API Classes

class retools.queue.**Job**(*queue_name*, *job_payload*, *redis*, *serializer=<function dumps at 0x7fc895e94ed8>*, *deserializer=<function loads at 0x7fc895e94e60>*)

__init__(*queue_name*, *job_payload*, *redis*, *serializer=<function dumps at 0x7fc895e94ed8>*, *deserializer=<function loads at 0x7fc895e94e60>*)

Create a job instance given a JSON job payload

Parameters

- **job_payload** – A JSON string representing a job.
- **queue_name** – The queue this job was pulled off of.
- **redis** – The redis instance used to pull this job.

A Job instance is created when the Worker pulls a job payload off the queue. The current_job global is set upon creation to indicate the current job being processed.

Attributes of interest for event functions:

•**job_id**: The Job's ID

•**job_name**: The Job's name (it's package + function name)

•**queue_name**: The queue this job came from

•**kwargs**: The keyword arguments the job is called with

•**state**: The state dict, this can be used by events to retain additional arguments. I.e. for a retry extension, retry information can be stored in the state dict.

•**func**: A reference to the job function

•**redis**: A redis.Redis instance.

•**serializer**: A callable to serialize json data, defaults to json.dumps().

•**deserializer**: A callable to deserialize json data, defaults to json.loads().

enqueue()

Queue this job in Redis

static load_events(*event_dict*)

Load all the events given the references

Parameters **event_dict** – A dictionary of events keyed by event name to a list of handlers for the event.

perform()

Runs the job calling all the job signals as appropriate

run_event(*event*, ***kwargs*)

Run all registered events for this job

class retools.queue.**Worker**(*queues*, *redis=None*, *serializer=<function dumps at 0x7fc895e94ed8>*, *deserializer=<function loads at 0x7fc895e94e60>*)

A Worker works on jobs

__init__(*queues*, *redis=None*, *serializer=<function dumps at 0x7fc895e94ed8>*, *deserializer=<function loads at 0x7fc895e94e60>*)

Create a worker

Parameters

- **queues** (*list*) – List of queues to process

- **redis** – Redis instance to use, defaults to the global_connection.

In the event that there is only a single queue in the list Redis list blocking will be used for lower latency job processing

done_working()
Called when we're done working on a job

immediate_shutdown(**args*)
Immediately shutdown the worker, kill child process if needed

kill_child(**args*)
Kill the child process immediately

pause_processing(**args*)
Cease pulling jobs off the queue for processing

perform()
Run the job and call the appropriate signal handlers

prune_dead_workers()
Prune dead workers from Redis

register_signal_handlers()
Setup all the signal handlers

register_worker()
Register this worker with Redis

reserve(*interval*, *blocking*)
Attempts to pull a job off the queue(s)

resume_processing(**args*)
Resume pulling jobs for processing off the queue

set_proc_title(*title*)
Sets the active process title, retains the retools prefic

startup()
Runs basic startup tasks

trigger_shutdown(**args*)
Graceful shutdown of the worker

unregister_worker(*worker_id=None*)
Unregister this worker with Redis

work(*interval=5*, *blocking=False*)
Work on jobs

This is the main method of the Worker, and will register itself with Redis as a Worker, wait for jobs, then process them.

Parameters

- **interval** (*int*) – Time in seconds between polling.
- **blocking** (*bool*) – Whether or not blocking pop should be used. If the blocking pop is used, then the worker will block for `interval` seconds at a time waiting for a new job. This affects how often the worker can respond to signals.

worker_id
Returns this workers id based on hostname, pid, queues

worker_pids()
Returns a list of all the worker processes

working_on()
Indicate with Redis what we're working on

1.2 Changelog

1.2.1 0.4.1 (02/19/2014)

Bug Fixes

- Properly support StrictRedis with ZADD (used in the limiter). Patch by Bernardo Heynemann.

1.2.2 0.4 (01/27/2014)

Features

- Added limiter functionality. Pull request #22, by Bernardo Heynemann.

1.2.3 0.3 (08/13/2012)

Bug Fixes

- Call redis.expire with proper expires value for RedisLock. Patch by Mike McCabe.
- Use functools.wraps to preserve doc strings for cache_region. Patch by Daniel Holth.

API Changes

- Added get_job/get_jobs methods to QueueManager class to get information on a job or get a list of jobs for a queue.

1.2.4 0.2 (02/01/2012)

Bug Fixes

- Critical fix for caching that prevents old values from being displayed forever. Thanks to Daniel Holth for tracking down the problem-aware.
- Actually sets the Redis expiration for a value when setting the cached value in Redis. This defaults to 1 week.

Features

- Statistics for the cache is now optional and can be disabled to slightly reduce the Redis queries used to store/retrieve cache data.
- Added first revision of worker/job Queue system, with event support.

Internals

- Heavily refactored `Connection` to not be a class singleton, instead a global_connection instance is created and used by default.
- Increased conditional coverage to 100% (via instrumental).

Backwards Incompatibilities

- Changing the default global Redis connection has changed semantics, instead of using `Connection.set_default`, you should set the global_connection's redis property directly:

```
import redis
from retools import global_connection

global_connection.redis = redis.Redis(host='myhost')
```

Incompatibilities

- Removed clear argument from invalidate_region, as removing keys from the set but not removing the hit statistics can lead to data accumulating in Redis that has no easy removal other than .keys() which should not be run in production environments.
- Removed deco_args from invalidate_callable (invalidate_function) as its not actually needed since the namespace is already on the callable to invalidate.

1.2.5 0.1 (07/08/2011)

Features

- Caching in a similar style to Beaker, with hit/miss statistics, backed by a Redis global write-lock with old values served to prevent the dogpile effect
- Redis global lock

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*

r

Symbols

A

C

D

E

I

J

K

L

N

P

Q

R

S

T

U

W