WorQ Documentation

Release 1.1.0

Daniel Miller

March 29, 2014

Contents

WorQ is a Python task queue that uses a worker pool to execute tasks in parallel. Workers can run in a single process, multiple processes on a single machine, or many processes on many machines. It ships with two backend options (memory and redis) and two worker pool implementations (multi-process and threaded). Task results can be monitored, waited on, or passed as arguments to another task.

WorQ has two main components:

- `TaskQueue`
- `WorkerPool`

WorQ ships with more than one implementation of each of these components.

- `worq.queue.memory.TaskQueue` - an in-memory (process local) task queue.
- `worq.queue.redis.TaskQueue` - a Redis-backed task queue that can scale to multiple servers.
- `worq.pool.thread.WorkerPool` - a multi-thread worker pool.
- `worq.pool.process.WorkerPool` - a multi-process worker pool.

These components can be mixed and matched as desired to meet the needs of your application. For example, an in-memory task queue can be used with a multi- process worker pool to to execute truely concurrent Python tasks on a single multi-core machine.

CHAPTER 1

An example with Redis and a multi-process worker pool

Create the following files.

tasks.py:

```
import logging
from worq import get_broker, TaskSpace

ts = TaskSpace(__name__)

def init(url):
    logging.basicConfig(level=logging.DEBUG)
    broker = get_broker(url)
    broker.expose(ts)
    return broker

@ts.task
def num(value):
    return int(value)

@ts.task
def add(values):
    return sum(values)
```

pool.py:

```
#!/usr/bin/env python
import sys
from worq.pool.process import WorkerPool
from tasks import init

def main(url, **kw):
    broker = init(url)
    pool = WorkerPool(broker, init, workers=2)
    pool.start(**kw)
    return pool

if __name__ == '__main__':
    main(sys.argv[-1])
```

main.py:

```
#!/usr/bin/env python
import sys
import logging
from worq import get_queue

def main(url):
    logging.basicConfig(level=logging.DEBUG)
    q = get_queue(url)

    # enqueue tasks to be executed in parallel
    nums = [q.tasks.num(x) for x in range(10)]

    # process the results when they are ready
    result = q.tasks.add(nums)

    # wait for the final result
    result.wait(timeout=30)

    print('0 + 1 + ... + 9 = {}'.format(result.value))

if __name__ == '__main__':
    main(sys.argv[-1])
```

Make sure Redis is accepting connections on port 6379. It is recommended, but not required, that you setup a virtualenv. Then, in a terminal window:

```
$ pip install "WorQ[redis]"
$ python pool.py redis://localhost:6379/0
```

And in a second terminal window:

```
$ python main.py redis://localhost:6379/0
```

Tasks may also be queued in in memory rather than using Redis. In this case the queue must reside in the same process that initiates tasks, but the work can still be done in separate processes. For example:

CHAPTER 2

Example with memory queue and a multi-process worker pool

In addition to the three files from the previous example, create the following:

mem.py:

```
#!/usr/bin/env python
import main
import pool

if __name__ == "__main__":
    url = "memory://"
    p = pool.main(url, timeout=2, handle_sigterm=False)
    try:
        main.main(url)
    finally:
        p.stop()
```

Then, in a terminal window:

```
$ python mem.py
```

See *Examples* for more things that can be done with WorQ.

CHAPTER 3

Links

- Documentation: http://worq.readthedocs.org/
- Source: https://github.com/millerdev/WorQ/
- PyPI: http://pypi.python.org/pypi/WorQ

CHAPTER 4

Running the tests

WorQ development is mostly done using TDD. Tests are important to verify that new code works. You may want to run the tests if you want to contribute to WorQ or simply just want to hack. Setup a virtualenv and run these commands where you have checked out the WorQ source code:

```
$ pip install nose
$ nosetests
```

The tests for some components (e.g., redis TaskQueue) are disabled unless the necessary requirements are available. For example, by default the tests look for redis at `redis://localhost:16379/0` (note non-standard port; you may customize this url with the `WORQ_TEST_REDIS_URL` environment variable).

CHAPTER 5

Change Log

v1.1.1, ??

- Add example using memory queue

v1.1.0, 2014-03-29

- Add support for Python 3

v1.0.2, 2012-09-07

- Allow clearing entire Queue with `del queue[:]`.
- Raise `DuplicateTask` (rather than the more generic `TaskFailure`) when trying to enqueue a task with an id matching that of another task in the queue.

v1.0.1, 2012-09-06

- Better support for managing more than one process.WorkerPool with a single pool manager process.
- Queue can be created with default task options.
- Can now check the approximate number of tasks in the queue with len(queue).
- Allow passing a completed Deferred as an argument to another task.
- Fix redis leaks.

v1.0.0, 2012-09-02 – Initial release.

CHAPTER 6

API Documentation

6.1 Examples

```
import worq.const as const
from worq import get_broker, get_queue, Task, TaskFailure, TaskSpace
from worq.tests.test_examples import example
from worq.tests.util import (assert_raises, eq_, eventually,
    thread_worker, TimeoutLock, WAIT)

@example
def simple(url):
    """A simple example demonstrating WorQ mechanics"""
    state = []

    def func(arg):
        state.append(arg)

    broker = get_broker(url)
    broker.expose(func)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        q.func('arg')

        eventually((lambda:state), ['arg'])


@example
def wait_for_result(url):
    """Efficiently wait for (block on) a task result.

    Use this feature wisely. Waiting for a result in a WorQ task
    could deadlock the queue.
    """

    def func(arg):
        return arg
```

```
    broker = get_broker(url)
    broker.expose(func)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        res = q.func('arg')

        completed = res.wait(WAIT)

        assert completed, repr(res)
        eq_(res.value, 'arg')
        eq_(repr(res), "<Deferred func [default:%s] success>" % res.id)


@example
def ignore_result(url):
    """Tell the queue to ignore the task result when the result is not
    important. This is done by creating a ''Task'' object with custom
    options for more efficient queue operation.
    """
    state = []

    def func(arg):
        state.append(arg)

    broker = get_broker(url)
    broker.expose(func)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        f = Task(q.func, ignore_result=True)
        res = f(3)

        eq_(res, None) # verify that we did not get a deferred result
        eventually((lambda:state), [3])


@example
def result_status(url):
    """Deferred results can be queried for task status.

    A lock is used to control state interactions between the producer
    and the worker for illustration purposes only. This type of
    lock-step interaction is not normally needed or even desired.
    """
    lock = TimeoutLock(locked=True)

    def func(arg):
        lock.acquire()
        return arg

    broker = get_broker(url)
    broker.expose(func)
    with thread_worker(broker, lock):
```

```
        # -- task-invoking code, usually another process --
        q = get_queue(url)

        res = q.func('arg')

        eventually((lambda:res.status), const.ENQUEUED)
        eq_(repr(res), "<Deferred func [default:%s] enqueued>" % res.id)

        lock.release()
        eventually((lambda:res.status), const.PROCESSING)
        eq_(repr(res), "<Deferred func [default:%s] processing>" % res.id)

        lock.release()
        assert res.wait(WAIT), repr(res)
        eq_(repr(res), "<Deferred func [default:%s] success>" % res.id)

        eq_(res.value, 'arg')


@example
def no_such_task(url):

    broker = get_broker(url)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        res = q.func('arg')
        assert res.wait(WAIT), repr(res)

        eq_(repr(res), '<Deferred func [default:%s] failed>' % res.id)
        with assert_raises(TaskFailure,
                'func [default:%s] no such task' % res.id):
            res.value


@example
def task_error(url):

    def func(arg):
        raise Exception('fail!')

    broker = get_broker(url)
    broker.expose(func)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        res = q.func('arg')
        assert res.wait(WAIT), repr(res)

        eq_(repr(res), '<Deferred func [default:%s] failed>' % res.id)
        with assert_raises(TaskFailure,
                'func [default:%s] Exception: fail!' % res.id):
            res.value
```

```
@example
def task_with_deferred_arguments(url):
    """A deferred result may be passed as an argument to another task. Tasks
    receiving deferred arguments will not be invoked until the deferred value
    is available. Notice that the value of the deferred argument, not the
    Deferred object itself, is passed to ``sum`` in this example.
    """

    def func(arg):
        return arg

    broker = get_broker(url)
    broker.expose(func)
    broker.expose(sum)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        res = q.sum([
            q.func(1),
            q.func(2),
            q.func(3),
        ])

        assert res.wait(WAIT), repr(res)
        eq_(res.value, 6)


@example
def more_deferred_arguments(url):
    from operator import add

    def func(arg):
        return arg

    broker = get_broker(url)
    broker.expose(func)
    broker.expose(sum)
    broker.expose(add)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        sum_123 = q.sum([
            q.func(1),
            q.func(2),
            q.func(3),
        ])

        sum_1234 = q.add(sum_123, q.func(4))

        assert sum_1234.wait(WAIT), repr(res)
        eq_(sum_1234.value, 10)


@example
```

```
def dependency_graph(url):
    """Dependency graph
                          |
              ____________|_____________
             /            |             \
            / \           / \           / \
           /   \         /   \         /   \
      left   right  left   right  left   right
         \   /         \   /         \   /
          \ /           \ /           \ /
         catch         catch         catch
           \             |            /
            \____________|___________/
                         |
                      combine
    """
    ts = TaskSpace()

    @ts.task
    def left(num):
        return ('left', num)

    @ts.task
    def right(num):
        return ('right', num)

    @ts.task
    def catch(left, right, num):
        return [num, left, right]

    @ts.task
    def combine(items):
        return {i[0]: i[1:] for i in items}

    broker = get_broker(url)
    broker.expose(ts)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        catches = []
        for num in [1, 2, 3]:
            left = q.left(num)
            right = q.right(num)

            catch = q.catch(left, right, num)

            catches.append(catch)

        res = q.combine(catches)
        assert res.wait(WAIT), repr(res)

        eq_(res.value, {
            1: [('left', 1), ('right', 1)],
            2: [('left', 2), ('right', 2)],
            3: [('left', 3), ('right', 3)],
        })
```

```
@example
def task_with_failed_deferred_arguments(url):
    """TaskFailure can be passed to the final task.

    By default, a task fails if any of its deferred arguments fail. However,
    creating a ``Task`` with ``on_error=Task.PASS`` will cause a ``TaskFailure``
    to be passed as the result of any task that fails.
    """

    def func(arg):
        if arg == 0:
            raise Exception('zero fail!')
        return arg

    broker = get_broker(url)
    broker.expose(func)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        items = [
            q.func(1),
            q.func(0),
            q.func(2),
        ]

        task = Task(q.func, on_error=Task.PASS)
        res = task(items)
        res.wait(timeout=WAIT)

        fail = TaskFailure(
            'func', 'default', items[1].id, 'Exception: zero fail!')
        eq_(res.value, [1, fail, 2])


@example
def named_queue(url):
    """Named queues facilitate discrete queues on a single backend."""

    foo_state = []
    def foo_func(arg):
        foo_state.append(arg)
    foo_broker = get_broker(url, 'foo')
    foo_broker.expose(foo_func)

    bar_state = []
    def bar_func(arg):
        bar_state.append(arg)
    bar_broker = get_broker(url, 'bar')
    bar_broker.expose(bar_func)

    with thread_worker(foo_broker), thread_worker(bar_broker):

        # -- task-invoking code, usually another process --
        f = get_queue(url, 'foo')
        f.foo_func(1)
```

```
        b = get_queue(url, 'bar')
        b.bar_func(2)

        eventually((lambda:(foo_state, bar_state)), ([1], [2]))


@example
def task_namespaces(url):
    """Task namepsaces are used to arrange tasks similar to the Python
    package/module hierarchy.
    """

    state = set()
    __name__ = 'module.path'

    ts = TaskSpace(__name__)

    @ts.task
    def foo():
        state.add('foo')

    @ts.task
    def bar(arg):
        state.add(arg)

    broker = get_broker(url)
    broker.expose(ts)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        q.module.path.foo()
        q.module.path.bar(1)

        eventually((lambda:state), {'foo', 1})


@example
def more_namespaces(url):
    state = set()

    foo = TaskSpace('foo')
    bar = TaskSpace('foo.bar')
    baz = TaskSpace('foo.bar.baz')

    @foo.task
    def join(arg):
        state.add('foo-join %s' % arg)

    @bar.task
    def kick(arg):
        state.add('bar-kick %s' % arg)

    @baz.task
    def join(arg):
        state.add('baz-join %s' % arg)
```

```
    @baz.task
    def kick(arg):
        state.add('baz-kick %s' % arg)

    broker = get_broker(url)
    broker.expose(foo)
    broker.expose(bar)
    broker.expose(baz)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)

        q.foo.join(1)
        q.foo.bar.kick(2)
        q.foo.bar.baz.join(3)
        q.foo.bar.baz.kick(4)

        eventually((lambda:state), {
            'foo-join 1',
            'bar-kick 2',
            'baz-join 3',
            'baz-kick 4',
        })


@example
def expose_method(url):
    """Object methods can be exposed too, not just functions."""

    class Database(object):
        """stateful storage"""
        value = None
        def update(self, value):
            self.value = value

    class TaskObj(object):
        """object with task definitions"""
        def __init__(self, db):
            self.db = db
        def update_value(self, value):
            self.db.update(value)

    db = Database()
    obj = TaskObj(db)
    broker = get_broker(url)
    broker.expose(obj.update_value)
    with thread_worker(broker):

        # -- task-invoking code, usually another process --
        q = get_queue(url)
        q.update_value(2)
        eventually((lambda:db.value), 2)
```

6.2 worq Package

6.2.1 worq Package

worq.**get_broker**(*url*, *name='default'*, **args*, ***kw*)
Create a new broker

Parameters

- **url** – Task queue URL.
- **name** – The name of the queue on which to expose or invoke tasks.

Returns An instance of worq.core.Broker.

worq.**get_queue**(*url*, *name='default'*, *target=''*, ***options*)
Get a queue for invoking remote tasks

Parameters

- **url** – Task queue URL.
- **name** – The name of the queue on which tasks should be invoked. Queued tasks will be invoked iff there is a worker listening on the named queue.
- **target** – Task namespace (similar to a python module) or name (similar to a python function). Defaults to the root namespace.
- ****options** – Default task options for tasks created with the queue. These can be overridden with worq.task.Task.

Returns An instance of worq.task.Queue.

6.2.2 task Module

class worq.task.**Deferred**(*broker*, *task*)
Bases: object

Deferred result object

Not thread-safe.

has_value()
Check for value without touching the broker

status
Get task status

value
Get the value returned by the task (if completed)

Returns The value returned by the task if it completed successfully.

Raises AttributeError if the task has not yet completed. TaskFailure if the task failed for any reason.

wait(*timeout*)
Wait for the task result.

Use this method wisely. In general a task should never wait on the result of another task because it may cause deadlock.

Parameters **timeout** – Number of seconds to wait. A value of `None` will wait indefinitely, but this is dangerous since the worker may go away without notice (due to loss of power, etc.) causing this method to deadlock.

Returns True if the result is available, otherwise False.

class `worq.task.`**`Queue`**(*broker*, *target=''*, ***options*)

Bases: `object`

Queue for invoking remote tasks

New `Queue` instances are generated through attribute access. For example:

```
>>> q = Queue(broker)
>>> q.foo
<Queue foo [default]>
>>> q.foo.bar
<Queue foo.bar [default]>
```

A `Queue` instance can be called like a function, which invokes a remote task identified by the target of the `Queue` instance. Example:

```
# Enqueue task 'func' in namespace 'foo' to be invoked
# by a worker listening on the 'default' queue.
>>> q = Queue(broker)
>>> q.foo.func(1, key=None)
```

The arrangement of queue tasks in `TaskSpaces` is similar to Python's package/module/function hierarchy.

NOTE two `Queue` objects are considered equal if they refer to the same broker (their targets may be different).

Parameters

- **broker** – A `Broker` instance.
- **target** – The task (space) path.
- ****options** – Default task options.

class `worq.task.`**`Task`**(*queue*, *id=None*, *on_error='fail'*, *ignore_result=False*, *result_timeout=3600*, *heartrate=30*)

Bases: `object`

Remote task handle

This class can be used to construct a task with custom options. A task is invoked by calling the task object.

Parameters

- **queue** – The `Queue` object identifying the task to be executed.
- **id** – A unique identifier string for this task, or a function that returns a unique identifier string when called with the task's arguments. If not specified, a global unique identifier is generated for each call. Only one task with a given id may exist in the queue at any given time. Note that a task with `ignore_result=True` will be removed from the queue before it is invoked.
- **on_error** – What should happen when a deferred argument's task fails. The `TaskFailure` exception will be passed as an argument if this value is `Task.PASS`, otherwise this will fail before it is invoked (the default action).
- **ignore_result** – Create a fire-and-forget task if true. Task invocation will return `None` rather than a `Deferred` object.

- **result_timeout** – Number of seconds to retain the result after the task has completed. The default is one hour. This is ignored by some `TaskQueue` implementations.
- **heartrate** – Number of seconds between task heartbeats, which are maintained by some `WorkerPool` implementations to prevent result timeout while the task is running. The default is 30 seconds.

with_options(*options*)
Clone this task with a new set of options

exception worq.task.**TaskFailure**
Bases: exceptions.Exception

Task failure exception class

Initialize with the following positional arguments:

1. Task name
2. Queue name
3. Task id
4. Error text

class worq.task.**TaskSpace**(*name=''*)
Bases: object

Task namespace container

task(*callable*, *name=None*)
Add a task to the namespace

This can be used as a decorator:

```
ts = TaskSpace(__name__)

@ts.task
def frob(value):
    db.update(value)
```

Parameters

- **callable** – A callable object, usually a function or method.
- **name** – Task name. Defaults to *callable.__name__*.

6.2.3 core Module

class worq.core.**Broker**(*taskqueue*)
Bases: object

A Broker controlls all interaction with the queue backend

deserialize(*message*, *task_id=None*)
Deserialize an object

Parameters

- **message** – A serialized object (string).
- **deferred** – When true load deferreds. When false raise an error if the message contains deferreds.

discard_pending_tasks()
Discard pending tasks from queue

expose(*obj*, *replace=False*)
Expose a TaskSpace or task callable.

Parameters

- **obj** – A TaskSpace or task callable.
- **replace** – Replace existing task if True. Otherwise (by default), raise ValueError if this would replace an existing task.

heartbeat(*task*)
Extend task result timeout

invoke(*task*, ***kw*)
Invoke the given task (normally only called by a worker)

next_task(*timeout=None*)
Get the next task from the queue.

Parameters **timeout** – See `AbstractTaskQueue.get`.

Returns A task object. `None` on timeout expiration or if the task could not be deserialized.

pop_result(*task*, *timeout=0*)
Pop and deserialize a task's result object

Parameters

- **task** – An object with `id` and `name` attributes representing the task.
- **timeout** – Length of time to wait for the result. The default behavior is to return immediately (no wait). Wait indefinitely if `None`.

Returns The deserialized result object.

Raises KeyError if the result was not available.

Raises TaskExpired if the task expired before a result was returned. A task normally only expires if the pool loses its ability to communicate with the worker performing the task.

queue(*target=''*, ***options*)
Get a Queue from the broker

serialize(*obj*, *deferred=False*)
Serialize an object

Parameters

- **obj** – The object to serialize.
- **deferred** – When this is true Deferred objects are serialized and their values are loaded on deserialization. When this is false Deferred objects are not serializable.

set_result(*task*, *result*)
Persist result object.

Parameters

- **task** – Task object for which to set the result.
- **result** – Result object.

status(*result*)
Get the status of a deferred result

task_failed(*task*)
Signal that the given task has failed.

class worq.core.**AbstractTaskQueue**(*url*, *name='default'*)
Bases: object

Message queue abstract base class

Task/result lifecycle

1.Atomically store non-expiring result placeholder and enqueue task.

2.Atomically pop task from queue and set timeout on result placeholder.

3.Task heartbeats extend result expiration as needed.

4.Task finishes and result value is saved.

All methods must be thread-safe.

Parameters

- **url** – URL used to identify the queue.
- **name** – Queue name.

defer_task(*result*, *message*, *args*)
Defer a task until its arguments become available

Parameters

- **result** – A Deferred result for the task.
- **message** – The serialized task message.
- **args** – A list of task identifiers whose results will be included in the arguments to the task.

discard_pending()
Discard pending tasks from queue

discard_result(*task_id*, *task_expired_token*)
Discard the result for the given task.

A call to pop_result after this is invoked should return a task expired response.

Parameters

- **task_id** – The task identifier.
- **task_expired_token** – A message that can be sent to blocking actors to signify that the task has expired.

enqueue_task(*result*, *message*)
Enqueue task

Parameters

- **result** – A Deferred result for the task.
- **message** – Serialized task message.

Returns True if the task was enqueued, otherwise False (duplicate task id).

get(*timeout=None*)
Atomically get a serialized task message from the queue

Task processing has started when this method returns, which means that the task heartbeat must be maintained if there could be someone waiting on the result. The result status is set to `worq.const.PROCESSING` if a result is being maintained for the task.

Parameters **timeout** – Number of seconds to wait before returning `None` if no task is available in the queue. Wait forever if timeout is `None`.

Returns A two-tuple (<task_id>, <serialized task message>) or `None` if timeout was reached before a task arrived.

get_arguments(*task_id*)
Get a dict of deferred arguments

Parameters **task_id** – The identifier of the task to which the arguments will be passed.

Returns A dict of serialized arguments keyed by argument id.

get_status(*task_id*)
Get the status of a task

Parameters **task_id** – Unique task identifier string.

Returns A task status value or `None`.

pop_result(*task_id*, *timeout*)
Pop serialized result message from persistent storage.

Parameters

- **task_id** – Unique task identifier string.
- **timeout** – Number of seconds to wait for the result. Wait indefinitely if `None`. Return immediately if timeout is zero (0).

Returns

One of the following:

- The result message.
- `worq.const.RESERVED` if another task depends on the result.
- `worq.const.TASK_EXPIRED` if the task expired before a result was available.
- `None` on timeout.

reserve_argument(*argument_id*, *deferred_id*)
Reserve the result of a task as an argument of a deferred task

Parameters

- **argument_id** – Identifier of a task whose result will be reserved for another task.
- **deferred_id** – Identifier of a deferred task who will get the reserved result as an argument.

Returns A two-tuple: (<bool>, <str>). The first item is a flag denoting if the argument was reserved, and the second is the serialized result if it was available else `None`.

set_argument(*task_id*, *argument_id*, *message*)
Set deferred argument for task

Parameters

- **task_id** – The identifier of the task to which the argument will be passed.
- **argument_id** – The argument identifier.
- **message** – The serialized argument value.

Returns True if all arguments have been set for the task.

set_result(*task_id*, *message*, *timeout*)
Persist serialized result message.

This also sets the result status to `worq.const.COMPLETED`.

Parameters

- **task_id** – Unique task identifier string.
- **message** – Serialized result message.
- **timeout** – Number of seconds to persist the result before discarding it.

Returns A deferred task identifier if the result has been reserved. Otherwise `None`.

set_task_timeout(*task_id*, *timeout*)
Set a timeout on the task result

Recursively set the timeout on the given task and all deferred tasks depending on this task's result.

size()
Return the approximate number of tasks in the queue

undefer_task(*task_id*)
Enqueue a deferred task

All deferred arguments must be available immediately.

6.2.4 const Module

6.2.5 Subpackages

pool Package

process Module

Multi-process worker pool

Processes in the `worq.pool.process` stack:

- Queue - enqueues tasks to be executed
- Broker - task queue and results backend (redis)
- WorkerPool - worker pool manager process
- Worker - worker process, which does the real work

class `worq.pool.process.`**PopenProcess**(*proc*)
Bases: `object`

Make a `subprocess.Popen` object more like `multiprocessing.Process`

class `worq.pool.process.`**WorkerPool**(*broker*, *init_func*, *init_args=()*, *init_kwargs=None*, *workers=None*, *max_worker_tasks=None*, *name=None*)
Bases: `object`

Multi-process worker pool

Parameters

- **broker** – Broker object.

- **init_func** – Worker initializer. This is called to initialize each worker on startup. It can be used to setup logging or do any other global initialization. The first argument will be a broker url, and the remaining will be `*init_args, **init_kwargs`. It must return a broker instance, and it must be pickleable.
- **init_args** – Additional arguments to pass to init_func.
- **init_kwargs** – Additional keyword arguments to pass to init_func.
- **workers** – Number of workers to maintain in the pool. The default value is the number returned by `multiprocessing.cpu_count`.
- **max_worker_tasks** – Maximum number of tasks to execute on each worker before retiring the worker and spawning a new one in its place.
- **name** – A name for this pool to distinguish its log output from that of other pools running in the same process.

join()
Wait for pool to stop (call after `.stop(join=False)`)

start(*timeout=10*, *handle_sigterm=True*)
Start the worker pool

Parameters

- **timeout** – Number of seconds to block while waiting for a new task to arrive on the queue. This timeout value affects pool stop time: a larger value means shutdown may take longer because it may need to wait longer for the consumer thread to complete.
- **handle_sigterm** – If true (the default) setup a signal handler and block until the process is signalled. This should only be called in the main thread in that case. If false, start workers and a pool manager thread and return.

stop(*join=True*)
Shutdown the pool

This is probably only useful when the pool was started with *handle_sigterm=False*.

worq.pool.process.**run_in_subprocess**(*_func*, **args*, ***kw*)
Call function with arguments in subprocess

All arguments to this function must be able to be pickled.

Use `subprocess.Popen` rather than `multiprocessing.Process` because we use threads, which do not play nicely with `fork`. This was originally written with `multiprocessing.Process`, which caused in intermittent deadlocks. See http://bugs.python.org/issue6721

Returns A `PopenProcess` object.

worq.pool.process.**start_pools**(**pools*, ***start_kwargs*)
Start one or more pools and wait indefinitely for SIGTERM or SIGINT

This is a blocking call, and should be run in the main thread.

thread Module

class worq.pool.thread.**WorkerPool**(*broker*, *workers=1*, *thread_factory=<class 'threading.Thread'>*)
Bases: `object`

Multi-thread worker pool

Parameters

- **broker** – Queue broker instance.
- **workers** – Number of workers in the pool.
- **thread_factory** – A factory function that creates a new thread object. This should have the same signature as `threading.Thread` and should return a thread object.

join()
Wait for all threads to stop (call `stop(join=False)` first)

start(*timeout=1*)
Start worker threads.

Parameters **timeout** – The number of seconds to wait for a task before checking if the pool has been asked to stop.

stop(*use_sentinel=False*, *join=True*)
Stop the worker pool

Parameters

- **use_sentinel** – Enqueue a no-op task for each worker if true. This will result in a more responsive shutdown if there are no other worker pools consuming tasks from the broker.
- **join** – Join each thread afer sending the stop signal.

queue Package

memory Module

In-memory message queue and result store.

class worq.queue.memory.**TaskQueue**(**args*, ***kw*)
Bases: worq.core.AbstractTaskQueue

Simple in-memory task queue implementation

defer_task(*result*, *message*, *args*)

discard_pending()

discard_result(*task_id*, *task_expired_token*)

enqueue_task(*result*, *message*)

classmethod factory(*url*, *name='default'*, **args*, ***kw*)

get(*timeout=None*)

get_arguments(*task_id*)

get_status(*task_id*)

pop_result(*task_id*, *timeout*)

reserve_argument(*argument_id*, *deferred_id*)

set_argument(*task_id*, *argument_id*, *message*)

set_result(*task_id*, *message*, *timeout*)

set_task_timeout(*task_id*, *timeout*)

size()

undefer_task(*task_id*)

redis Module

Redis message queue and result store.

class worq.queue.redis.**TaskQueue**(*url*, *name='default'*, *initial_result_timeout=60*, *redis_factory=<class 'redis.client.StrictRedis'>*)

Bases: worq.core.AbstractTaskQueue

Redis task queue

defer_task(*result*, *message*, *args*)

discard_pending()

discard_result(*task_id*, *task_expired_token*)

enqueue_task(*result*, *message*)

get(*timeout=0*)

get_arguments(*task_id*)

get_status(*task_id*)

log_all_worq(*show_expiring=False*)
debugging helper

ping()

pop_result(*task_id*, *timeout*)

reserve_argument(*argument_id*, *deferred_id*)

set_argument(*task_id*, *argument_id*, *message*)

set_result(*task_id*, *message*, *timeout*)

set_task_timeout(*task_id*, *timeout*)

size()

undefer_task(*task_id*)

worq.queue.redis.**utf8**(*value*)

tests Package

tests Package

worq.tests.**get_redis_url**()

worq.tests.**setup**()

worq.tests.**test_redis_should_be_installed**()

test_core Module

worq.tests.test_core.**Broker_duplicate_task_id**(*url*, *identifier*)

worq.tests.test_core.**test_Broker_duplicate_task_id_function**()

worq.tests.test_core.**test_Broker_duplicate_task_id_string**()

worq.tests.test_core.**test_Broker_task_failed**()

test_examples Module

worq.tests.test_examples.**example**(*func*)

worq.tests.test_examples.**test_examples**()

test_task Module

worq.tests.test_task.**test_Queue_default_options**()

worq.tests.test_task.**test_Queue_len**()

worq.tests.test_task.**test_clear_Queue**()

worq.tests.test_task.**test_completed_Deferred_as_argument**()

worq.tests.test_task.**test_deferred_task_fail_on_error**()

worq.tests.test_task.**test_worker_interrupted**()

util Module

class worq.tests.util.**TimeoutLock**(*locked=False*)
Bases: object

A lock with acquisition timeout

acquire(*timeout=10*)

release()

worq.tests.util.**assert_raises**(**args*, ***kwds*)

worq.tests.util.**eq_**(*value*, *other*)

worq.tests.util.**eventually**(*get_value*, *expect*, *timeout=10*, *poll_interval=0*)

worq.tests.util.**tempdir**(**args*, ***kwds*)
Create a temporary directory to be used in a test

If (optional keyword argument) 'delete' evaluates to True (the default value), the temporary directory and all files in it will be removed on context manager exit.

worq.tests.util.**thread_worker**(**args*, ***kwds*)

worq.tests.util.**with_urls**(*test=None*, *exclude=None*)

Subpackages

pool Package

test_process Module

worq.tests.pool.test_process.**WorkerPool_crashed_worker_init**(*url*)

worq.tests.pool.test_process.**WorkerPool_heartrate_init**(*url*)

worq.tests.pool.test_process.**WorkerPool_max_worker_tasks_init**(*url*)

worq.tests.pool.test_process.**WorkerPool_sigterm_init**(*url*, *tmp*, *logpath*)

worq.tests.pool.test_process.**WorkerPool_worker_shutdown_on_parent_die_init**(*url*, *tmp*, *logpath*)

worq.tests.pool.test_process.**discard_tasks**(**args*, ***kwds*)

worq.tests.pool.test_process.**force_kill_on_exit**(**args*, ***kwds*)

worq.tests.pool.test_process.**noop**()

worq.tests.pool.test_process.**pid_running**(*pid*)

worq.tests.pool.test_process.**printlog**(**args*, ***kwds*)

worq.tests.pool.test_process.**process_config**(*logpath*, *procname*)

worq.tests.pool.test_process.**reader**(**path*)

worq.tests.pool.test_process.**start_pool**(**args*, ***kwds*)

worq.tests.pool.test_process.**test_WorkerPool_crashed_worker**()

worq.tests.pool.test_process.**test_WorkerPool_heartrate**()

worq.tests.pool.test_process.**test_WorkerPool_max_worker_tasks**()

worq.tests.pool.test_process.**test_WorkerPool_sigterm**()

worq.tests.pool.test_process.**test_WorkerPool_start_twice**()

worq.tests.pool.test_process.**test_WorkerPool_worker_shutdown_on_parent_die**()

worq.tests.pool.test_process.**touch**(*path*, *data=''*)

worq.tests.pool.test_process.**verify_shutdown**(*proc*)

worq.tests.pool.test_process.**worker_pool**(*url*, *init_func*, *init_args*, *workers=1*)

CHAPTER 7

Indices and tables

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

CHAPTER 8

License

WorQ - Python task queue

Copyright (c) 2012 Daniel Miller

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Python Module Index

W