redicts Documentation

Release 1.0.0

adnymics

Dec 18, 2018

Contents

A utility package to save arbitrary nested Python dicts and objects in Redis.

This package can be used to save arbitrary values in a hierarchy. Each element of this hierarchy is referenced by a dotted path like this: `a.b.c`. When saving a nested dictionary, it's nested contents automatically get translated to such a dotted path by it's string keys:

```
# `23` can be read by specifying the path "a.b.c":
{
    "a": {
        "b": {
            "c": 23
        }
    }
}
```

A special feature of this package is concurrent access: It can be safely used from more than one process. The locking implementation is also separated and can be used on it's one if desirable. Also, the implementation is clever enough to not require a global lock if changes are done in different parts of the hierarchy.

You can store every object in `redicts` that works with `json.dumps()`.

CHAPTER 1

Installation

1.1 Over PyPI

```
$ pip install redicts
```

1.2 Manual installation

This requires `git` to be installed.

```
$ git clone https://github.com/adnymics/redicts
$ cd redicts
$ pip install -r requirements.txt
$ python setup.py install
```

If you want to run the tests you can also do:

```
$ pip install -r test_requirements.txt
$ pytest
```

It is recommended to do these steps in a virtual environment.

CHAPTER 2

Example Usage

Instead of boring you with an insanely long description of what this library is capable of, we'll keep it short and just give you examples. You'll learn the concepts along the way. All details also can be found in the *API Reference*.

2.1 Basics

2.1.1 Getting and setting values

All values in `redis` are accessed by instances of the *`redicts.Proxy`*. class. They represent a key that points to a value and can be asked to fetch it with *`redicts.Proxy.val`*:

```
>>> from redicts import Proxy, root
>>> p = Proxy("a.b.c")
>>> p.set("d", 42)
>>> p.get("d").val()
42
# set() also returns a proxy for the current value:
>>> p.set("x", {"y": "z"}).val()
{'x': {'y': 'z'}, 'd': 42}
>>> root().val()
{'a': {'b': {'c': {'x': {'y': 'z'}, 'd': 42}}}}
>>> # Not exsting values will yield None.
>>> root().get("who?").val()
None
```

Also observe that the values really live a hierarchy.

Warning:

Note that value access is not locked by default for performance reasons! Take a look at the next example to allow concurrent access.

2.1.2 Concurrent access to values

Every *redicts.Proxy* can be locked against concurrent access with it's *redicts.Proxy.acquire* and *redicts.Proxy.release* methods. This is optional, since locking can eat quite a bit of performance when done often. Take this example:

```
from redicts import Proxy
from multiprocessing import Process


Proxy("a.b.c").set("d", 0)

def increment():
    for _ in range(10000):
        with Proxy("a.b.c") as prox:
            prox.get("d").add(1)

p = Process(target=increment)
p.start()
increment()
p.join()
```

Try this example without the `with` to see the difference.

2.1.3 Other useful operations

Here are a few operations you can do on a *redicts.Proxy*:

- *redicts.Proxy.iter_children()*: Return a *redicts.Proxy*: for each direct child.
- *redicts.Proxy.delete()*: Delete a single subkey.
- *redicts.Proxy.exists()*: Check if a key has a value assigned.
- *redicts.Proxy.clear()*: Clear everthing below this prox.

Here they are in action:

```
>>> from redicts import Pool, root
>>> r = root()
>>> r.set('x', 1)
>>> r.set('y', {"z": 2})
>>> r.val()
{'y': {'z': 2}, 'x': 1}
>>> list(r)
>>> ["y", "x"]
>>> {p.key(): p.val() for p in root().iter_children()}
{'y.z': 2, 'x': 1}
>>> r.get("x").exists()
True
>>> r.delete("x")
>>> r.get("x").exists()
False
>>> r.clear()
>>> r.get("y").exists()
False
```

2.1.4 Different redis server

Everything related to connection details can be configured via the *redicts.Pool* singleton. It's responsible for keeping a pool of open connections and acts as central instance for configurations. Upon first use of anything network related *redicts.Pool* is instantiated with default connection details. If you like to use different connection details you can do this:

```
from redicts import Pool, root

Pool().reload(cfg=dict(
    host="localhost",
    port=6379,
    database=0,
    password="1234",
    max_connections=100,
    timeout_secs=50,
))
```

2.1.5 Using `fakredis`

Using a real instance of `redis` can be inconvinient for testing. In this case you can setup your tests with `fakeredis`:

```
from redicts import Pool, root

# Make sure to use `fakeredis`
Pool().reload(fake_redis=True)

# clear everything that was written by this library:
root().clear()
```

2.2 Advanced

Not all of the following features might be required during »daily« usage.

2.2.1 Using more than one database

If you'd like to use more than one database you can setup a mapping in the beginning:

```
from redicts import Pool, Proxy

# Assuming default setup:
Pool().reload(cfg={
    "default": 0,
    "names": {
        "persons": 1,
        "things": 2,
    }
})

# Later on you can use the human readable name for your database:
# All three values are stored in different redis db with different values.
Proxy("x").set("y", 1)
```

(continues on next page)

(continued from previous page)

```
Proxy("x", db_name="persons").set("y", 2)
Proxy("x", db_name="things").set("y", 3)
```

2.2.2 Time to live

You can tell `redis` to expire keys after some time. This is also possible with `redicts`:

```
import time
from redicts import Pool, root

# Expire this key in 10 seconds:
root().set("x", "still here!", expire=10)
time.sleep(1)
root().get("x").time_to_live()  # => 9
root().get("x").val()           # => "still here!"
time.sleep(10)
root().get("x").time_to_live()  # => -2
root().get("x").val()           # => None

# You can also alternatively set the expire time later:
root().set("x", "still here!")
root().get("x").expire(10)
```

CHAPTER 3

API Reference

This documentation is generated from the docstrings found in the source.

3.1 Main Interface

class redicts.**Proxy**(*path*, *lock_acquire_timeout=10*, *lock_expire_timeout=30*, *db_name=None*)

Create a new Proxy.

Parameters

- **str_or_iterable** (*path*) – The path where this value is stored. Can be a string (a dotted path) or an iterable of strings.
- **redis.Redis** (*rconn*) – Optional; the redis connection to use.

Returns Proxy The ready to use Proxy.

acquire()

Acquire a lock on this value and all of it's children.

add(*count*)

Convinience function to add a count to this value. If the value did not exist yet, it will be set to count. Will raise an ValueError if the key exists and does not support the add operator.

Parameters **count** – (int) The count to increment.

Returns The new total count.

clear()

Clear this level of the value tree including all children

delete(*key*)

Delete an existing key.

Parameters **key** – (str) A dotted path.

exists()
Return true if this value actually exists

expire(*seconds*)
Expire (i.e. delete) the key after a certain number of seconds. After this time .val() will return None and .exists() will return False.

Parameters **seconds** – (int) seconds after this value will no longer accessible.

get(*key*)
Return a lazy value for this key.

Parameters **key** – (str) A dotted path or simple

Returns A child Proxy.

is_locked()
Check if the node or any of its parents are locked

iter_children()
Iterate over all children including and below this node.

Returns A generator object, yielding ValueProxies.

key()
Return the key of this value in redis

release()
Release a previously acquired lock.

Note that this does not clear the locks of the children, if you locked those explicitly you have to release them.

set(*key*, *value*, *expire=None*)
Set a new value to this key.

Parameters

- **key** – (str) A dotted path.
- **value** – (object) Any value that can be passed to json.dumps.
- **expire** – (int) Time in seconds when to expire this key or None.

time_to_live()
Return the amount of seconds, this value will be accessible.

Returns int the amount to live in seconds.

val(*default=None*)
Get the actual value of this proxy

redicts.**root**(**args*, ***kwargs*)
Return the root Proxy

redicts.**section**(*name*, **args*, ***kwargs*)
Convience method for getting a Proxy for a first-level section. Try to to use a unique name, otherwise you might overwrite foreign keys. A good idiom is to use something like this to get a descriptive, but unique name for your module:

section(__name__)

Parameters **str** (*name*) – The section name. May not contain dots.

Returns A Proxy for the section.

class redicts.**Pool**(*cfg=None*)
Pool of redis connections

get_connection(*db_name=None*)
Get a new (or recycled) connection. Note: This function may block if there are too many connections open.

Return redis.StrictRedis A new redis connection.

reload(*cfg=None*, *fake_redis=False*)
Reload the pool, disconnecting previous connections and creating a new pool.

Parameters **(dict)** (*cfg*) – See documentation for __init__.

3.2 Locking

The locking implementation is available as separte class and can be used as multiprocess lock.

class redicts.**Lock**(*redis_conn*, *key*, *expire_timeout=30*, *acquire_timeout=10*)
Implement a distributed, thread-safe lock for redis.

The basics are described here: https://redis.io/topics/transactions This lock is more flexible than other locking implementations, since it supports tree-based locking by specifying a dotted path as key.

If you lock an element higher up in the hierarchy, all elements below it will be automatically locked too. It is still possible to lock elements below though, but those will only add their lock to the lock above.

This implementation uses optimistic locking, i.e. retry if some of the to be locked values changed.

acquire()
Acquire the lock and wait if needed

is_locked()
Return True if this lock was already acquired by someone

release()
Release the lock again

3.3 Exceptions

exception redicts.**InternalError**
Raised when the implementation got confused. This should only happen when somebody else tampers with the locking keys in redis.

exception redicts.**LockTimeout**
Raised when lock creation fails within the timeout

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index