python-memcached2 Documentation

Release 0.1

Sean Reifschneider

September 22, 2015

Contents

python-memcached2 is a next-generation implementation re-implementation of the python-memcached module. The primary goals are to get rid of some baggage, improve testability/maintainability/performance, and support Python 3. This codebase is regularly tested against Python 2.7 and Python 3.3.

The high level interface is dict-like: `ExceptionsAreMissesMapping`. It looks much like a dictionary but the back-end storage is memcache servers.

The low level `Memcache` class is complete and documented, see *Memcache Examples* for examples of use.

CHAPTER 1

Documentation Index

1.1 What's New in python-memcached2

- Renamed SelectorAvailableServers to SelectorRehashDownServers.

Wed Sep 04, 2013

- I'm going to tag this as 0.3, because it's at a pretty functional state, and I'm going to stat breaking things to work on separating the server connecting and re-connecting from the server selection.

Sat Aug 31, 2013

- I have finished the implementation of set_multi(), but it could use some more tests, particularly those verifying what happens on bad server responses.

Mon Aug 26, 2013

- I have completed the implementation of send_multi() and it passes a basic test. Need to decide what I want the return to be, and do more tests, but it's on its way.

Mon Jun 12, 2013

- Added a new Selector, *SelectorFractalSharding*, and removed the `SelectorRestirOnDownServer`. The new Selector improves on the old one in pretty much every way, and is the default for when more than 2 servers are listed.

Mon Jun 10, 2013

- Tagging as 0.2 as the functionality here is usable and stabilized and I want to start working on some significant changes that may break things for a while.

Sun Jun 09, 2013

- Added *SelectorConsistentHashing* that implements this algorithm for server selection.

Sun Jun 08, 2013

- Removed `SelectorRehashOnDownServer` and replaced it with the better `SelectorRestirOnDownServer`.

Tue Jun 04, 2013

- Fixing a bug if Memcache(selector+XXX) is used, hasher was not being set.
- Created SelectorRehashOnDownServer which will hash to the same server, unless that server is down in which case it will rehash among only the up servers.

Wed May 28, 2013

- Adding `memcached2.Memcache.delete_all()` and `memcached2.ValueSuperStr.sdelete_allet()`.

Wed May 27, 2013

- SelectorAvailableServers now can flush all servers when the topology changes. That's the situation it is most suited for, though it's also ideal for 2 server clusters.

Wed May 25, 2013

- Added get_multi which can get multiple keys at once.

Wed May 22, 2013

- Memcache now has a get_multi() method that will get multiple keys at once.

Wed May 20, 2013

- Memcache.cache() now takes varargs and kwargs, optionally, which will be passed to the compute function.

Wed May 19, 2013

- Memcache.cache() added which will call a function on a cache miss, then put the result in the cache.

Wed May 18, 2013

- Now have a ExceptionsAreMissesMemcache() class for lower-level access that treats exceptions as misses.

Wed May 15, 2013

- ValueSuperStr can now do a CAS refresh on `memcached2.ValueSuperStr.set()`.

Wed May 8, 2013

- MemcacheValue is now called ValueSuperStr, and it is no longer the default return type in Memcache(). It can be defined by passing ValueMemcache to Memcache() as the "value_wrapper". There's also a ValueDictionary now.
- Adding ValueDictionary class.
- Memcache() class no longer returns MemcacheValue class. It returns a normal string, unless you have specified a value_wrapper attribute during the creation of the Memcache object.

Tue May 7, 2013

- Adding MANIFEST.in file.
- Adding CASFailure to MemcacheValue methods.

Fri May 3, 2013

- I did a short performance test against the python-memcached library that this is meant to replace. This new module is around 10% faster (using the Memcache() class) at retrieving 10 byte values, and 16% faster at 1KB values. I was expecting more, but I also haven't done any performance tuning. If I just return normal strings instead of ValueSuperStr, that goes up to 23% faster, so that may be a point of optimization.
- Adding remaining methods to MemcacheValue.

Thu May 2, 2013

- MemcacheValue now has "set()" method.

Wed May 1, 2013

- I'm tagging a 0.2 but still not going to release to pypi yet. Server failure testing, related to ExceptionsAreMissesMapping, have located several exceptions that weren't being caught and translated into local module exceptions. Current functionality is solid, but I want to add a MemcacheCASValue class, which is kind of an API change.

- Improving Python 2 BrokenPipeError
- Catching more exceptions, more tests.

  Added more extensive testing to ExceptionsAsMissesMapping, including in the cases where the server disconnects. Through that, found places where more exceptions needed to be caught.

Tue Apr 30, 2013

- Trapping ServerDisconnected exception.

Mon Apr 29, 2013

- ObliviousMapping renamed ExceptionsAreMissesMapping

  ExceptionsAreMissesMapping suggested by Wes Winham. Thanks!

Sat Apr 27, 2013

- The module is usable, but if you do you should expect that the interfaces may change. The high level `Memcache` code is basically complete, documented, and well tested.
- Bringing back KeyError because d.get() is preferable.
- Renaming ObliviousDict to ObliviousMapping.

Fri Apr 26, 2013

- Adding ObliviousDict() tests and fixing "in".

1.2 Exception as Misses Mapping

1.2.1 Introduction

This is a subclass of *Memcache* which swallows exceptions and treats them like misses. This is meant to allow code to be a bit simpler, rather than catching all exceptions, you can just do things like the below example.

1.2.2 Examples

Basic example:

```
>>> import memcached2
>>> mc = memcached2.ExceptionsAreMissesMemcache(('memcached://localhost/',))
>>> data = mc.get('key')
>>> if not data:
>>>     data = [compute data]
>>>     mc.set('key', data)
>>> [use data]
```

1.2.3 Object Documentation

class memcached2.**ExceptionsAreMissesMemcache**(*servers*, *value_wrapper=None*, *selector=None*, *hasher=None*, *server_pool=None*)

A `Memcache` wrapper class which swallows server exceptions, except in the case of coding errors. This is meant for situations where you want to keep the code simple, and treat cache misses, server errors, and the like as cache misses. See *memcached2.Memcache()* for details of the use of this class, exceptions to that are noted here.

The methods that are protected against exceptions are those documented in this class. Everything should otherwise act like a *Memcache* instance.

Parameters

- **servers** (*list*) – One or more server URIs of the form: "memcache://hostname[:port]/"
- **value_wrapper** (ValueSuperStr or compatible object.) – (None) This causes values returned to be wrapped in the passed class before being returned. For example ValueSuperStr implements many useful additions to the string return.
- **selector** (SelectorBase) – (None) This code implements the server selector logic. If not specified, the default is used. The default is to use *SelectorFirst* if only one server is specified, and *SelectorRehashDownServers* if multiple servers are given.
- **hasher** (HasherBase) – (None) A "Hash" object which takes a key and returns a hash for persistent server selection. If not specified, it defaults to HasherZero if there is only one server specified, or HasherCMemcache otherwise.
- **server_pool** (ServerPool object.) – (None) A server connection pool. If not specified, a global pool is used.

delete(**args*, ***kwargs*)

Remove this key from the server.

Exceptions are swallowed and treated as memcached misses. See *delete()* for details on this method. Changes from the base function are:

Raises Exceptions are swallowed and treated a misses.

get(**args*, ***kwargs*)

Retrieve the specified key from a memcache server.

Exceptions are swallowed and treated as memcached misses. See *get()* for details on this method. Changes from the base function are:

Returns None if no value or exception, String, or "value_wrapper" as specified during object creation such as *~memcached2.ValueSuperStr*.

Raises Exceptions are swallowed and treated a misses.

set(**args*, ***kwargs*)

Update the value in the server. See *set()* for details on this method. Changes from the base function are:

Exceptions are swallowed and treated as memcached misses. See *set()* for details on this method. Changes from the base function are:

Raises Exceptions are swallowed and treated a misses.

set_multi(**args*, ***kwargs*)

Update multiple values in the server. See set_multi() for details on this method. Changes from the base function are:

Exceptions are swallowed and treated as memcached misses. See *set()* for details on this method. Changes from the base function are:

Raises Exceptions are swallowed and treated a misses.

1.3 Exception as Misses Mapping

1.3.1 Introduction

This is a dictionary-like interface to memcache, but it swallows server exceptions, except in the case of coding errors. This is meant for situations where you want to keep the code simple, and treat cache misses, server errors, and the like as cache misses.

On the instantiation you specify the servers, and at that point it can be accessed as a dictionary, including access, setting, and deleting keys. See the examples for a demonstration.

For functionality beyond what you can get from the dictionary interface, you need to use the `memcache` attribute, which is an *`Memcache`* instance. See that documentation for access to flusing servers, statistics, and other things not supported by the mapping interface.

Note that `NotImplementedException` will be raised for situations where there are code errors. So it's recommended that you don't just trap these, either catch and log them, or just let them raise up so that application users can report the bug.

1.3.2 Examples

Basic example:

```
>>> import memcached2
>>> mcd = memcached2.ExceptionsAreMissesMapping(('memcached://localhost/',))
>>> 'foo' in mcd
False
>>> mcd['foo'] = 'hello'
>>> 'foo' in mcd
True
>>> mcd['foo']
'hello'
>>> len(mcd)
1
>>> del(mcd['foo'])
>>> len(mcd)
0
```

1.3.3 Object Documentation

class memcached2.**ExceptionsAreMissesMapping**(*servers*, *selector=None*, *hasher=None*)

A dictionary-like interface which swallows server exceptions.

This is a dictionary-like interface to memcache, but it swallows server exceptions, except in the case of coding errors. This is meant for situations where you want to keep the code simple, and treat cache misses, server errors, and the like as cache misses.

See *ExceptionsAreMissesMapping Introduction* and *ExceptionsAreMissesMapping Examples* for more information.

1.4 Hasher Routines

class memcached2.**HasherBase**

Turn memcache keys into hashes, for use in server selection.

Normally, the python-memcached2 classes will automatically select a hasher to use. However, for special circumstances you may wish to use a different hasher or develop your own.

This is an abstract base class, here largely for documentation purposes. Hasher sub-classes such as *HasherZero* and *HasherCMemcache*, implement a *hash* method which does all the work.

See *hash()* for details of implementing a subclass.

hash(*key*)
Hash a key into a number.

This must persistently turn a string into the same value. That value is used to determine which server to use for this key.

Parameters **key** (*str*) – memcache key

Returns int – Hashed version of *key*.

class memcached2.**HasherZero**
Hasher that always returns 0, useful only for *SelectorFirst*.

hash(*key*)
See *memcached2.HasherBase.hash()* for details of this function.

class memcached2.**HasherCMemcache**
Hasher compatible with the C memcache hash function.

hash(*key*)
See *memcached2.HasherBase.hash()* for details of this function.

1.5 Selector Routines

class memcached2.**SelectorBase**
Select which server to use.

These classes implement a variety of algorithms for determining which server to use, based on the key being stored.

The selection is done based on a *key_hash*, as returned by the *memcached2.HasherBase.hash()* function.

Normally, the python-memcached2 classes will automatically pick a selector to use. However, for special circumstances you may wish to use a specific Selector or develop your own.

This is an abstract base class, here largely for documentation purposes. Selector sub-classes such as *SelectorFirst* and *SelectorRehashDownServers*, implement a *select* method which does all the work.

See *select()* for details of implementing a subclass.

select(*server_uri_list*, *hasher*, *key*, *server_pool*)
Select a server bashed on the *key_hash*.

Given the list of servers and a hash of of key, determine which of the servers this key is associated with on.

Parameters

- **server_uri_list** (*list of server URIs*) – A list of the server URIs to select among.
- **hasher** (memcache2.HasherBase.hash().) – Hasher function, such as memcache2.HasherBase.hash().
- **key** (*str*) – The key to hash.

- **server_pool** (ServerPool object.) – (None) A server connection pool. If not specified, a global pool is used.

Returns string – The server_uri to use.

Raises *NoAvailableServers*

class memcached2.**SelectorFirst**
Server selector that only returns the first server. Useful when there is only one server to select amongst.

select(*server_uri_list*, *hasher*, *key*, *server_pool*)
See *memcached2.SelectorBase.select()* for details of this function.

class memcached2.**SelectorRehashDownServers**(*hashing_retries=10*)
Select a server, if it is down re-hash up to *hashing_retries* times.

This was the default in the original python-memcached module. If the server that a key is housed on is down, it will re-hash the key after adding an (ASCII) number of tries to the key and try that server.

This is most suitable if you want to inter-operate with the old python-memcache client.

If no up server is found after *hashing_retries* attempts, *memcached2.NoAvailableServers* is raised.

Parameters **hashing_retries** (*int*) – Retry hashing the key looking for another server this many times.

select(*server_uri_list*, *hasher*, *key*, *server_pool*)
See *memcached2.SelectorBase.select()* for details of this function.

class memcached2.**SelectorFractalSharding**
On a down server, re-partition that servers keyspace to other servers.

This uses an algorithm that basically maps every key in the keyspace to a list of the servers that will answer queries for it. The first available server in that list will be used. The list is such that the keys that map to a server when it is up will get distributed across other servers evenly, stabally, and predictably.

I called it Fractal because when a server is down you dig deeper and see a new level of complexity in the keyspace mapping.

select(*server_uri_list*, *hasher*, *key*, *server_pool*)
See *memcached2.SelectorBase.select()* for details of this function.

class memcached2.**SelectorConsistentHashing**(*total_buckets=None*)
Predictably select a server, even if its normal server is down.

This implements the Consistent Hash algorithm as http://en.wikipedia.org/wiki/Consistent_hashing

This is done by splitting the key-space up into a number of buckets (more than the number of servers but probably no more than the number of servers squared). See Wikipedia for details on how this algorithm operates.

The downside of this mechanism is that it requires building a fairly large table at startup, so it is not suited to short lived code. It also is fairly expensive to add and remove servers from the pool (not implemented in this code). Note that it is NOT expensive to fail a server, only to completely remove it.

Parameters **total_buckets** (*int*) – How many buckets to create. Smaller values decrease the startup overhead, but also mean that a down server will tend to not evenly redistribute it's load across other servers. The default value of None means the default value of the number of servers squared.

select(*server_uri_list*, *hasher*, *key*, *server_pool*)
See *memcached2.SelectorBase.select()* for details of this function.

1.6 Reconnector Routines

1.7 String-like Memcache Value Object

class memcached2.**ValueSuperStr**
Wrapper around Memcache value results.

This acts as a string normally, containing the value read from the server. However, it is augmented with additional attributes representing additional data received from the server: *flags*, *key*, and *cas_unique* (which may be None if it was not requested from the server).

If this is constructed with the *memcache* ServerConnection instance, then additional methods may be used to update the value via this object. If *cas_unique* is given, these updates are done using the CAS value.

1.8 Dictionary-like Memcache Value Object

class memcached2.**ValueDictionary**(*value*, *key*, *flags*, *cas_unique=None*, *memcache=None*)
Encode the response as a dictionary.

This is a simple dictionary of the result data from the memcache server, including keys: "key", "value", "flags", and "cas_unique". This is a way of getting additional data from the memcache server for use in things like CAS updates.

Instantiate new instance.

Parameters

- **value** (*str*) – The memcache *value*, which is the value of this class when treated like a string.
- **key** (*str*) – The *key* associated with the *value* retrieved.
- **flags** (*int*) – *flags* associated with the *value* retrieved.
- **cas_unique** (*int*) – The *cas_unique* value, if it was queried, or None if no CAS information was retrieved.
- **memcache** (ServerConnection) – The memcache server instance, used for future operations on this key.

Returns *ValueSuperStr* instance

1.9 Low-Level Memcache() Interface

1.9.1 Introduction

This is a low-level interface to a group of memcache servers. This code tends to either return the requested data, or raises an exception if the data is not available or there is any sort of an error. If you want high level control, this is probably the interface for you. However, if you want something easy, like the old python-memcached module, you will want to wait for the higher level interfaces to be implemented.

1.9.2 Examples

Basic *get()* and exception example:

```
>>> import memcached2
>>> memcache = memcached2.Memcache(('memcached://localhost/',))
>>> try:
...     result = memcache.get('session_id')
...     print('Got cached results: {0}'.format(repr(result)))
... except memcached2.NoValue:
...     print('Cached value not available, need to recompute it')
...
Cached value not available, need to recompute it
```

Demonstrating *set()*, *get()* and *ValueSuperStr*:

```
>>> memcache.set('session_id', 'TEST SESSSION DATA')
>>> result = memcache.get('session_id')
>>> print('Got cached results: {0}'.format(repr(result)))
Got cached results: 'TEST SESSSION DATA'
>>> result.key
'session_id'
>>> result.flags
0
```

Example of `get_multi()` to retrieve multiple keys quickly:

```
>>> memcache.set('foo', '1')
>>> memcache.set('bar', '2')
>>> result = memcache.get_multi(['foo', 'bar', 'baz'])
>>> result.get('foo')
'1'
>>> result.get('bar')
'2'
>>> result.get('baz')
None
```

Usage of `cache()` to automatically cache values:

```
>>> numbers = range(10)
>>> calculate = lambda x: str(numbers.pop())
>>> memcache.flush_all()
>>> memcache.cache('foo', calculate)
'9'
>>> memcache.cache('foo', calculate)
'9'
>>> memcache.set('foo', 'hello')
>>> memcache.cache('foo', calculate)
'hello'
>>> memcache.flush_all()
>>> memcache.cache('foo', calculate)
'8'
```

Showing flags and expiration time and *touch()*:

```
>>> memcache.set('foo', 'xXx', flags=12, exptime=30)
>>> result = memcache.get('foo')
>>> result
'xXx'
>>> result.key
```

```
'foo'
>>> result.flags
12
>>> import time
>>> time.sleep(30)
>>> result = memcache.get('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "memcached2.py", line 411, in get
    raise NoValue()
memcached2.NoValue
>>> memcache.set('foo', 'bar', exptime=1)
>>> memcache.touch('foo', exptime=30)
>>> time.sleep(2)
>>> memcache.get('foo')
'bar'
```

Usage of *replace()*, *append()*, and *prepend()*:

```
>>> memcache.replace('unset_key', 'xyzzy')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "memcached2.py", line 502, in replace
    self._storage_command(command, key)
  File "memcached2.py", line 945, in _storage_command
    raise NotStored()
memcached2.NotStored
>>> memcache.set('unset_key', 'old_data', exptime=30)
>>> memcache.replace('unset_key', 'xyzzy')
>>> memcache.get('unset_key')
'xyzzy'
>>> memcache.append('unset_key', '>>>')
>>> memcache.prepend('unset_key', '<<<')
>>> memcache.get('unset_key')
'<<<xyzzy>>>'
```

Example of using CAS (Check And Set) atomic operations:

```
>>> memcache.set('foo', 'bar')
>>> result = memcache.get('foo', get_cas=True)
>>> result.cas_unique
5625
>>> memcache.set('foo', 'baz', cas_unique=result.cas_unique)
>>> memcache.get('foo', get_cas=True)
'baz'
>>> memcache.set('foo', 'qux', cas_unique=result.cas_unique)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "memcached2.py", line 464, in set
    self._storage_command(command, key)
  File "memcached2.py", line 947, in _storage_command
    raise CASFailure()
memcached2.CASFailure
>>> memcache.get('foo', get_cas=True)
'baz'
```

Usage of *incr()*/*decr()*:

```
>>> memcache.incr('incrtest', 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "memcached2.py", line 878, in incr
    return self._incrdecr_command(command, key)
  File "memcached2.py", line 915, in _incrdecr_command
    raise NotFound()
memcached2.NotFound
>>> memcache.set('incrtest', 'a')
>>> memcache.incr('incrtest', 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "memcached2.py", line 878, in incr
    return self._incrdecr_command(command, key)
  File "memcached2.py", line 919, in _incrdecr_command
    raise NonNumeric()
memcached2.NonNumeric
>>> memcache.set('incrtest', '1')
>>> memcache.incr('incrtest', 1)
2
>>> memcache.decr('incrtest', 1)
1
>>> memcache.get('incrtest')
'1'
```

Statistics sample information:

```
>>> import pprint
>>> pprint.pprint(memcache.stats())
[{'accepting_conns': '1',
  'auth_cmds': 0,
  'auth_errors': 0,
  'bytes': 201,
  'bytes_read': 173542,
  'bytes_written': 516341,
  'cas_badval': 49,
  'cas_hits': 49,
  'cas_misses': 0,
  'cmd_flush': 1154,
  'cmd_get': 880,
  'cmd_set': 5778,
  'cmd_touch': '148',
  'conn_yields': 0,
  'connection_structures': 9,
  'curr_connections': 5,
  'curr_items': 3,
  'decr_hits': 49,
  'decr_misses': 48,
  'delete_hits': 49,
  'delete_misses': 49,
  'evicted_unfetched': 0,
  'evictions': 0,
  'expired_unfetched': 0,
  'get_hits': '681',
  'get_misses': '199',
  'hash_bytes': 262144,
  'hash_is_expanding': '0',
  'hash_power_level': 16,
  'incr_hits': 49,
```

```
  'incr_misses': 49,
  'libevent': '2.0.19-stable',
  'limit_maxbytes': 67108864,
  'listen_disabled_num': '0',
  'pid': 22356,
  'pointer_size': 32,
  'reclaimed': 0,
  'reserved_fds': 20,
  'rusage_system': 7.568473,
  'rusage_user': 8.904556,
  'threads': 4,
  'time': 1366722131,
  'total_connections': 1545,
  'total_items': 5634,
  'touch_hits': 98,
  'touch_misses': 50,
  'uptime': 370393,
  'version': '1.4.14'}]
>>> pprint.pprint(memcache.stats_settings())
[{'auth_enabled_sasl': 'no',
  'binding_protocol': 'auto-negotiate',
  'cas_enabled': True,
  'chunk_size': 48,
  'detail_enabled': False,
  'domain_socket': 'NULL',
  'evictions': 'on',
  'growth_factor': 1.25,
  'hashpower_init': 0,
  'inter': '127.0.0.1',
  'item_size_max': 1048576,
  'maxbytes': 67108864,
  'maxconns': 1024,
  'maxconns_fast': False,
  'num_threads': 4,
  'num_threads_per_udp': 4,
  'oldest': 216734,
  'reqs_per_event': 20,
  'slab_automove': False,
  'slab_reassign': False,
  'stat_key_prefix': ':',
  'tcp_backlog': 1024,
  'tcpport': 11211,
  'udpport': 11211,
  'umask': 700,
  'verbosity': 0}]
>>> pprint.pprint(memcache.stats_items())
[{'1': {'age': 766,
        'evicted': 0,
        'evicted_nonzero': 0,
        'evicted_time': 0,
        'evicted_unfetched': 0,
        'expired_unfetched': 0,
        'number': 3,
        'outofmemory': 0,
        'reclaimed': 0,
        'tailrepairs': 0}}]
>>> pprint.pprint(memcache.stats_sizes())
[[(64, 1), (96, 2)]]
```

```
>>> pprint.pprint(memcache.stats_slabs())
[{'active_slabs': 1,
  'slabs': {'1': {'cas_badval': 49,
                  'cas_hits': 49,
                  'chunk_size': 80,
                  'chunks_per_page': 13107,
                  'cmd_set': 5778,
                  'decr_hits': 49,
                  'delete_hits': 49,
                  'free_chunks': 13104,
                  'free_chunks_end': 0,
                  'get_hits': 681,
                  'incr_hits': 49,
                  'mem_requested': 201,
                  'total_chunks': 13107,
                  'total_pages': 1,
                  'touch_hits': 98,
                  'used_chunks': 3}},
  'total_malloced': 1048560}]
```

How to *delete()*, *flush_all()*, and *close()* the connection:

```
>>> memcache.delete('foo')
>>> memcache.flush_all()
>>> memcache.close()
```

1.9.3 Object Documentation

class memcached2.**Memcache**(*servers*, *value_wrapper=None*, *selector=None*, *hasher=None*, *server_pool=None*)

Create a new memcache connection, to the specified servers.

The list of servers, specified by URL, are consulted based on the hash of the key, effectively "sharding" the key space.

This is a low-level memcache interface. This interface will raise exceptions when backend connections occur, allowing a program full control over handling of connection problems.

Example:

```
>>> from memcached2 import *                                    # noqa
>>> mc = Memcache(['memcached://localhost:11211/'])
>>> mc.set('foo', 'bar')
>>> mc.get('foo')
'bar'
```

Extensive examples including demonstrations of the statistics output is available in the documentation for *Memcache Examples*

Parameters

- **servers** (*list*) – One or more server URIs of the form: "memcache://hostname[:port]/"
- **value_wrapper** (ValueSuperStr or compatible object.) – (None) This causes values returned to be wrapped in the passed class before being returned. For example ValueSuperStr implements many useful additions to the string return.
- **selector** (SelectorBase) – (None) This code implements the server selector logic. If not specified, the default is used. The default is to use *SelectorFirst* if only one server is specified, and *SelectorRehashDownServers* if multiple servers are given.

- **hasher** (HasherBase) – (None) A "Hash" object which takes a key and returns a hash for persistent server selection. If not specified, it defaults to HasherZero if there is only one server specified, or HasherCMemcache otherwise.
- **server_pool** (ServerPool object.) – (None) A server connection pool. If not specified, a global pool is used.

add(*key*, *value*, *flags=0*, *exptime=0*)
Store, but only if the server doesn't already hold data for it.

Parameters

- **key** (*str*) – Key used to store value in memcache server and hashed to determine which server is used.
- **value** (*str*) – Value stored in memcache server for this key.
- **flags** (*int (32 bits)*) – If specified, the same value will be provided on get().
- **exptime** (*int*) – If non-zero, it specifies the expriation time, in seconds, for this value.

append(*key*, *value*)
Store data after existing data associated with this key.

Parameters

- **key** (*str*) – Key used to store value in memcache server and hashed to determine which server is used.
- **value** (*str*) – Value stored in memcache server for this key.

close()
Close the connection to all the backend servers.

decr(*key*, *value=1*)
Decrement the value for the key, treated as a 64-bit unsigned value.

Parameters

- **key** (*str*) – Key used to store value in memcache server and hashed to determine which server is used.
- **value** (*int (64 bit)*) – A numeric value (default=1) to add to the existing value.

Returns int – (64 bits) The new value after the decrement.

Raises *NotFound*, *NonNumeric*, NotImplementedError

delete(*key*)
Delete the key if it exists.

Parameters **key** (*str*) – Key used to store value in memcache server and hashed to determine which server is used.

Returns Boolean indicating if key was deleted.

Raises NotImplementedError, *NoAvailableServers*

flush_all()
Flush the memcache servers.

Note: An attempt is made to connect to all backend servers before running this command.

Raises NotImplementedError

get(*key*, *get_cas=False*)
Retrieve the specified key from a memcache server.

Parameters

- **key** (*str*) – The key to lookup in the memcache server.
- **get_cas** (*bool*) – If True, the "cas unique" is queried and the return object has the "cas_unique" attribute set.

Returns String, or "value_wrapper" as specified during object creation such as *~memcached2.ValueSuperStr*.

Raises *NoValue*, NotImplementedError, *NoAvailableServers*

incr(*key*, *value=1*)
Increment the value for the key, treated as a 64-bit unsigned value.

Parameters

- **key** (*str*) – Key used to store value in memcache server and hashed to determine which server is used.
- **value** (*int (64 bit)*) – A numeric value (default=1) to add to the existing value.

Returns int – (64 bits) The new value after the increment.

Raises *NotFound*, *NonNumeric*, NotImplementedError

prepend(*key*, *value*)
Store data before existing data associated with this key.

Parameters

- **key** (*str*) – Key used to store value in memcache server and hashed to determine which server is used.
- **value** (*str*) – Value stored in memcache server for this key.

replace(*key*, *value*, *flags=0*, *exptime=0*)
Store data, but only if the server already holds data for it.

Parameters

- **key** (*str*) – Key used to store value in memcache server and hashed to determine which server is used.
- **value** (*str*) – Value stored in memcache server for this key.
- **flags** (*int (32 bits)*) – If specified, the same value will be provided on get().
- **exptime** (*int*) – If non-zero, it specifies the expriation time, in seconds, for this value.

set(*key*, *value*, *flags=0*, *exptime=0*, *cas_unique=None*)
Set a key to the value in the memcache server.

Parameters

- **key** (*str*) – Key used to store value in memcache server and hashed to determine which server is used.
- **value** (*str*) – Value stored in memcache server for this key.
- **flags** (*int (32 bits)*) – If specified, the same value will be provided on get().
- **exptime** (*int*) – If non-zero, it specifies the expriation time, in seconds, for this value.

- **cas_unique** (*int (64 bits)*) – If specified as the 64-bit integer from `get()` with *cas_unique=True*, the value is only stored if the value has not been updated since the `get()` call.

stats()
Get general statistics about memcache servers.

Examples of the results of this function is available in the documentation as *Memcache Statistics Examples*

Note: An attempt is made to connect to all servers before issuing this command.

Returns

list – The statistics data is a dictionary of key/value pairs representing information about the server.

This data is returned as a list of statistics, one item for each server. If the server is not connected, *None* is returned for its position, otherwise data as mentioned above.

stats_items()
Get statistics about item storage per slab class from the memcache servers.

Examples of the results of this function is available in the documentation as *Memcache Statistics Examples*

Note: An attempt is made to connect to all servers before issuing this command.

Returns

list – The statistic information is a dictionary keyed by the "slab class", with the value another dictionary of key/value pairs representing the slab information.

This data is returned as a list of statistics, one item for each server. If the server is not connected, None is returned for its position, otherwise data as mentioned above.

stats_settings()
Gets statistics about settings (primarily from processing command-line arguments).

Examples of the results of this function is available in the documentation as *Memcache Statistics Examples*

Note: An attempt is made to connect to all servers before issuing this command.

Returns

list – The statistic information is a dictionary of key/value pairs.

This data is returned as a list of statistics, one item for each server. If the server is not connected, None is returned for its position, otherwise data as mentioned above.

stats_sizes()
Get statistics about object sizes.

Examples of the results of this function is available in the documentation as *Memcache Statistics Examples*

Warning: This operation locks the cache while it iterates over all objects. Returns a list of (size,count) tuples received from the server.

Note: An attempt is made to connect to all servers before issuing this command.

Returns

list – Each statistic is a dictionary of of size:count where the size is rounded up to 32-byte ranges.

This data is returned as a list of statistics, one item for each server. If the server is not connected, None is returned for its position, otherwise data as mentioned above.

stats_slabs()

Gets information about each of the slabs created during memcached runtime. Returns a dictionary of slab IDs, each contains a dictionary of key/value pairs for that slab.

Examples of the results of this function is available in the documentation as *Memcache Statistics Examples*

Note: An attempt is made to connect to all servers before issuing this command.

Returns

list – The statistic information is a dictionary keyed by the "slab class", with the value another dictionary of key/value pairs representing statistic information about each of the slabs created during the memcace runtime.

This data is returned as a list of statistics, one item for each server. If the server is not connected, None is returned for its position, otherwise data as mentioned above.

touch(*key*, *exptime*)

Update the expiration time on an item.

Parameters

- **key** (*str*) – Key used to store value in memcache server and hashed to determine which server is used.
- **exptime** (*int*) – If non-zero, it specifies the expriation time, in seconds, for this value. Note that setting exptime=0 causes the item to not expire based on time.

Raises *NotFound*, NotImplementedError, *NoAvailableServers*

1.10 Low-Level ServerConnection() Interface

class memcached2.**ServerConnection**(*uri*)

Low-level communication with the memcached server.

Data should be passed in as strings, and that is converted to *bytes* for sending to the backend, encoded as ASCII, if necessary. Data returned is likewise converted from *bytes*, also encoded as ASCII, if necessary.

This implments the connection to a server, sending messages and reading responses. This is largely intended to be used by other modules in the memcached2 module such as *Memcache()* rather than directly by consumers.

Note that this class buffers data read from the server, so you should **never** read data directly from the underlying socket, as it may confuse other software which uses this interface.

Parameters **uri** (*str*) – The URI of the backend server.

connect()

Connect to memcached server.

If already connected, this function returns immmediately. Otherwise, the connection is reset and a connection is made to the backend.

Raises *UnknownProtocol*

consume_from_buffer(*length*)
Retrieve the specified number of bytes from the buffer.

Parameters **length** (*int*) – Number of bytes of data to consume from buffer.

Returns str – Data from buffer.

parse_uri()
Parse a server connection URI.

Parses the *uri* attribute of this object.

Currently, the only supported URI format is of the form:

•memcached://<hostname>[:port]/ – A TCP socket connection to the host, optionally on the specified port. If *port* is not specified, port 11211 is used.

Returns dict – A dictionary with the key *protocol* and other protocol-specific keys. For *memcached* protocol the keys include *host*, and *port*.

Raises *InvalidURI*

read_length(*length*)
Read the specified number of bytes.

Parameters **length** (*int*) – Number of bytes of data to read.

Returns str – Data read from socket. Converted from *bytes* (as read from backend) with ASCII encoding, if necessary.

Raises *ServerDisconnect*

read_until(*search='\r\n'*)
Read data from the server until "search" is found.

Data is read in blocks, any remaining data beyond *search* is held in a buffer to be consumed in the future.

param search Read data from the server until *search* is found. This defaults to '

', so it acts like readline().

type search str

returns str – Data read, up to and including *search*. Converted from *bytes* (as read from backend) with ASCII encoding, if necessary.

raises *ServerDisconnect*

reset()
Reset the connection.

Flushes buffered data and closes the backend connection.

send_command(*command*)
Write an ASCII command to the memcached server.

Parameters **command** (*str*) – Data that is sent to the server. This is converted to a *bytes* type with ASCII encoding if necessary for sending across the socket.

Raises *ServerDisconnect*

1.11 python-memcached2 Exceptions

1.11.1 Overview

The classes that throw exceptions all tend to raise exceptions that are children of the MemcachedException. For storage-related exceptions, they are children of StoreException, and for retrieval they are children of RetrieveException.

If you use the exception-exposing interfaces ("Memcache()"), *will* need to catch these exceptions as part of your code. They are thrown on exceptional conditions, read the description of the exceptions for details on when they may be thrown.

In specific error cases that likely indicate bugs in the python-memcached2 module, or where the server replies with unexpected data, the NotImplementedError is raised. These situations are extremely unusual and almost certainly should be reported to the developers of either this python-memcached2 module or the developers of the memcached server you are using. You probably don't want to catch these

1.11.2 Exceptions

class memcached2.**MemcachedException**
Base exception that all other exceptions inherit from. This is never raised directly.

class memcached2.**UnknownProtocol**
An unknown protocol was specified in the memcached URI. Sublcass of *MemcachedException*.

class memcached2.**InvalidURI**
An error was encountered in parsing the server URI. Subclass of *MemcachedException*.

class memcached2.**ServerDisconnect**
The connection to the server closed. Subclass of *MemcachedException*.

class memcached2.**NoAvailableServers**
There are no servers available to cache on, probably because all are disconnected. This exception typically occurs after the code which would do a reconnection is run. Subclass of *MemcachedException*.

class memcached2.**StoreException**
Base class for storage related exceptions. Never raised directly. Subclass of *MemcachedException*.

class memcached2.**MultiStorageException**(*message=None*, *results={}*)
During a SET operation the server returned CLIENT_ERROR. This is probably due to too long of a key being used. Subclass of *StoreException*.

class memcached2.**NotStored**
Item was not stored, but not due to an error. Normally means the condition for an "add" or "replace" was not met.. Subclass of *StoreException*.

class memcached2.**CASFailure**
Item you are trying to store with a "cas" command has been modified since you last fetched it (result=EXISTS). Subclass of *StoreException*.

class memcached2.**CASRefreshFailure**
When trying to refresh a CAS from the memcached, the retrieved value did not match the value sent with the last update. This may happen if another process has updated the value. Subclass of *CASFailure*.

class memcached2.**NotFound**
Item you are trying to store with a "cas" command does not exist.. Subclass of *StoreException*.

class memcached2.**NonNumeric**
The item you are trying to incr/decr is not numeric.. Subclass of *StoreException*.

class memcached2.**RetrieveException**
Base class for retrieve related exceptions. This is never raised directly.. Subclass of *MemcachedException*.

class memcached2.**NoValue**
Server has no data associated with this key.. Subclass of *RetrieveException*.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

m

T

U

V