sprockets.clients.memcached

Release 1.1

Jan 09, 2018

Contents

Memcached client wrapper that is configured via environment variables

CHAPTER 1

Installation

sprockets.clients.memcached is available on the Python Package Index and can be installed via pip or easy_install:

```
pip install sprockets.clients.memcached
```

CHAPTER 2

Requirements

- python-memcached

CHAPTER 3

API Documentation

3.1 Memcached Client API

The memcached client API wraps the `memcache.Client` adding environment variable based configuration.

Example environment variable configuration:

```
<PREFIX>_MEMCACHED_SERVERS = '10.0.0.1:11211:64,10.0.0.2:11211:64'
```

class sprockets.clients.memcached.**Client**(*prefix=None*)

Wraps `memcache.Client`, passing in the environment variable prefix. If prefix is set, the environment variable key is in the format `<PREFIX>_MEMCACHED_SERVERS`. If the prefix is not set, the list will attempt to be retrieved from the `MEMCACHED_SERVERS`. If neither environment variable is set, the default value of `127.0.0.1:11211` is used.

The per server format in the comma separated list is:

```
[HOST]:[PORT]<:WEIGHT>
```

Where host and port are required but weight is optional.

Parameters **prefix** (*str*) – The environment variable prefix. Default: `None`

add(*key*, *val*, *time=0*, *min_compress_len=0*, *noreply=False*)

Add new key with value.

Like L{set}, but only stores in memcache if the key doesn't already exist.

@return: Nonzero on success. @rtype: int

append(*key*, *val*, *time=0*, *min_compress_len=0*, *noreply=False*)

Append the value to the end of the existing key's value.

Only stores in memcache if key already exists. Also see L{prepend}.

@return: Nonzero on success. @rtype: int

cas(*key*, *val*, *time=0*, *min_compress_len=0*, *noreply=False*)

Check and set (CAS)

Sets a key to a given value in the memcache if it hasn't been altered since last fetched. (See L{gets}).

The C{key} can optionally be an tuple, with the first element being the server hash value and the second being the key. If you want to avoid making this module calculate a hash value. You may prefer, for example, to keep all of a given user's objects on the same memcache server, so you could use the user's unique id as the hash value.

@return: Nonzero on success. @rtype: int

@param time: Tells memcached the time which this value should expire, either as a delta number of seconds, or an absolute unix time-since-the-epoch value. See the memcached protocol docs section "Storage Commands" for more info on <exptime>. We default to 0 == cache forever.

@param min_compress_len: The threshold length to kick in auto-compression of the value using the compressor routine. If the value being cached is a string, then the length of the string is measured, else if the value is an object, then the length of the pickle result is measured. If the resulting attempt at compression yields a larger string than the input, then it is discarded. For backwards compatibility, this parameter defaults to 0, indicating don't ever try to compress.

@param noreply: optional parameter instructs the server to not send the reply.

check_key (*key*, *key_extra_len=0*)
Checks sanity of key.

Fails if:

Key length is > SERVER_MAX_KEY_LENGTH (Raises MemcachedKeyLength). Contains control characters (Raises MemcachedKeyCharacterError). Is not a string (Raises MemcachedStringEncodingError) Is an unicode string (Raises MemcachedStringEncodingError) Is not a string (Raises MemcachedKeyError) Is None (Raises MemcachedKeyError)

decr (*key*, *delta=1*, *noreply=False*)
Decrement value for C{key} by C{delta}

Like L{incr}, but decrements. Unlike L{incr}, underflow is checked and new values are capped at 0. If server value is 1, a decrement of 2 returns 0, not -1.

@param delta: Integer amount to decrement by (should be zero or greater).

@param noreply: optional parameter instructs the server to not send the reply.

@return: New value after decrementing, or None for noreply or error. @rtype: int

delete (*key*, *time=None*, *noreply=False*)
Deletes a key from the memcache.

@return: Nonzero on success. @param time: number of seconds any subsequent set / update commands should fail. Defaults to None for no delay. @param noreply: optional parameter instructs the server to not send the

reply.

@rtype: int

delete_multi (*keys*, *time=None*, *key_prefix=''*, *noreply=False*)
Delete multiple keys in the memcache doing just one query.

```
>>> notset_keys = mc.set_multi({'a1' : 'val1', 'a2' : 'val2'})
>>> mc.get_multi(['a1', 'a2']) == {'a1' : 'val1','a2' : 'val2'}
1
>>> mc.delete_multi(['key1', 'key2'])
1
```

```
>>> mc.get_multi(['key1', 'key2']) == {}
1
```

This method is recommended over iterated regular L{delete}s as it reduces total latency, since your app doesn't have to wait for each round-trip of L{delete} before sending the next one.

@param keys: An iterable of keys to clear @param time: number of seconds any subsequent set / update commands should fail. Defaults to 0 for no delay. @param key_prefix: Optional string to prepend to each key when

sending to memcache. See docs for L{get_multi} and L{set_multi}.

@param noreply: optional parameter instructs the server to not send the reply.

@return: 1 if no failure in communication with any memcacheds. @rtype: int

flush_all()
Expire all data in memcache servers that are reachable.

forget_dead_hosts()
Reset every host in the pool to an "alive" state.

get(*key*)
Retrieves a key from the memcache.

@return: The value or None.

get_multi(*keys*, *key_prefix=''*)
Retrieves multiple keys from the memcache doing just one query.

```
>>> success = mc.set("foo", "bar")
>>> success = mc.set("baz", 42)
>>> mc.get_multi(["foo", "baz", "foobar"]) == {
...     "foo": "bar", "baz": 42
... }
1
>>> mc.set_multi({'k1' : 1, 'k2' : 2}, key_prefix='pfx_') == []
1
```

This looks up keys 'pfx_k1', 'pfx_k2', ... Returned dict will just have unprefixed keys 'k1', 'k2'.

```
>>> mc.get_multi(['k1', 'k2', 'nonexist'],
...              key_prefix='pfx_') == {'k1' : 1, 'k2' : 2}
1
```

get_multi [and L{set_multi}] can take str()-ables like ints / longs as keys too. Such as your db pri key fields. They're rotored through str() before being passed off to memcache, with or without the use of a key_prefix. In this mode, the key_prefix could be a table name, and the key itself a db primary key number.

```
>>> mc.set_multi({42: 'douglass adams',
...               46: 'and 2 just ahead of me'},
...              key_prefix='numkeys_') == []
1
>>> mc.get_multi([46, 42], key_prefix='numkeys_') == {
...     42: 'douglass adams',
...     46: 'and 2 just ahead of me'
... }
1
```

This method is recommended over regular L{get} as it lowers the number of total packets flying around your network, reducing total latency, since your app doesn't have to wait for each round-trip of L{get} before sending the next one.

See also L{set_multi}.

@param keys: An array of keys.

@param key_prefix: A string to prefix each key when we communicate with memcache. Facilitates pseudo-namespaces within memcache. Returned dictionary keys will not have this prefix.

@return: A dictionary of key/value pairs that were available. If key_prefix was provided, the keys in the returned dictionary will not have it present.

get_stats(*stat_args=None*)
Get statistics from each of the servers.

@param stat_args: Additional arguments to pass to the memcache "stats" command.

@return: A list of tuples (server_identifier, stats_dictionary). The dictionary contains a number of name/value pairs specifying the name of the status field and the string value associated with it. The values are not converted from strings.

gets(*key*)
Retrieves a key from the memcache. Used in conjunction with 'cas'.

@return: The value or None.

incr(*key*, *delta=1*, *noreply=False*)
Increment value for C{key} by C{delta}

Sends a command to the server to atomically increment the value for C{key} by C{delta}, or by 1 if C{delta} is unspecified. Returns None if C{key} doesn't exist on server, otherwise it returns the new value after incrementing.

Note that the value for C{key} must already exist in the memcache, and it must be the string representation of an integer.

```
>>> mc.set("counter", "20")  # returns 1, indicating success
1
>>> mc.incr("counter")
21
>>> mc.incr("counter")
22
```

Overflow on server is not checked. Be aware of values approaching 2**32. See L{decr}.

@param delta: Integer amount to increment by (should be zero or greater).

@param noreply: optional parameter instructs the server to not send the reply.

@return: New value after incrementing, no None for noreply or error. @rtype: int

prepend(*key*, *val*, *time=0*, *min_compress_len=0*, *noreply=False*)
Prepend the value to the beginning of the existing key's value.

Only stores in memcache if key already exists. Also see L{append}.

@return: Nonzero on success. @rtype: int

replace(*key*, *val*, *time=0*, *min_compress_len=0*, *noreply=False*)
Replace existing key with value.

Like L{set}, but only stores in memcache if the key already exists. The opposite of L{add}.

@return: Nonzero on success. @rtype: int

reset_cas()
Reset the cas cache.

This is only used if the Client() object was created with "cache_cas=True". If used, this cache does not expire internally, so it can grow unbounded if you do not clear it yourself.

set(*key*, *val*, *time=0*, *min_compress_len=0*, *noreply=False*)
Unconditionally sets a key to a given value in the memcache.

The C{key} can optionally be an tuple, with the first element being the server hash value and the second being the key. If you want to avoid making this module calculate a hash value. You may prefer, for example, to keep all of a given user's objects on the same memcache server, so you could use the user's unique id as the hash value.

@return: Nonzero on success. @rtype: int

@param time: Tells memcached the time which this value should expire, either as a delta number of seconds, or an absolute unix time-since-the-epoch value. See the memcached protocol docs section "Storage Commands" for more info on <exptime>. We default to 0 == cache forever.

@param min_compress_len: The threshold length to kick in auto-compression of the value using the compressor routine. If the value being cached is a string, then the length of the string is measured, else if the value is an object, then the length of the pickle result is measured. If the resulting attempt at compression yields a larger string than the input, then it is discarded. For backwards compatibility, this parameter defaults to 0, indicating don't ever try to compress.

@param noreply: optional parameter instructs the server to not send the reply.

set_multi(*mapping*, *time=0*, *key_prefix=''*, *min_compress_len=0*, *noreply=False*)
Sets multiple keys in the memcache doing just one query.

```
>>> notset_keys = mc.set_multi({'key1' : 'val1', 'key2' : 'val2'})
>>> keys = mc.get_multi(['key1', 'key2'])
>>> keys == {'key1': 'val1', 'key2': 'val2'}
True
```

This method is recommended over regular L{set} as it lowers the number of total packets flying around your network, reducing total latency, since your app doesn't have to wait for each round-trip of L{set} before sending the next one.

@param mapping: A dict of key/value pairs to set.

@param time: Tells memcached the time which this value should expire, either as a delta number of seconds, or an absolute unix time-since-the-epoch value. See the memcached protocol docs section "Storage Commands" for more info on <exptime>. We default to 0 == cache forever.

@param key_prefix: Optional string to prepend to each key when sending to memcache. Allows you to efficiently stuff these keys into a pseudo-namespace in memcache:

```
>>> notset_keys = mc.set_multi(
...     {'key1' : 'val1', 'key2' : 'val2'},
...     key_prefix='subspace_')
>>> len(notset_keys) == 0
True
>>> keys = mc.get_multi(['subspace_key1', 'subspace_key2'])
>>> keys == {'subspace_key1': 'val1', 'subspace_key2': 'val2'}
True
```

Causes key 'subspace_key1' and 'subspace_key2' to be set. Useful in conjunction with a higher-level layer which applies namespaces to data in memcache. In this case, the return result would be the list of notset original keys, prefix not applied.

@param min_compress_len: The threshold length to kick in auto-compression of the value using the compressor routine. If the value being cached is a string, then the length of the string is measured, else if the value is an object, then the length of the pickle result is measured. If the resulting attempt at compression yields a larger string than the input, then it is discarded. For backwards compatibility, this parameter defaults to 0, indicating don't ever try to compress.

@param noreply: optional parameter instructs the server to not send the reply.

@return: List of keys which failed to be stored [memcache out of memory, etc.].

@rtype: list

set_servers (*servers*)
Set the pool of servers used by this client.

@param servers: an array of servers. Servers can be passed in two forms:

1. Strings of the form C{"host:port"}, which implies a default weight of 1. 2. Tuples of the form C{("host:port", weight)}, where C{weight} is an integer weight value.

touch (*key*, *time=0*, *noreply=False*)
Updates the expiration time of a key in memcache.

@return: Nonzero on success. @param time: Tells memcached the time which this value should

expire, either as a delta number of seconds, or an absolute unix time-since-the-epoch value. See the memcached protocol docs section "Storage Commands" for more info on <exptime>. We default to 0 == cache forever.

@param noreply: optional parameter instructs the server to not send the reply.

@rtype: int

3.2 Examples

The following example sets the environment variables for connecting to memcached on `192.168.1.2` and `192.168.1.3` and subsequently issuing a few memcached commands:

```
import os

from sprockets.clients import memcached

os.environ['MEMCACHED_SERVERS'] = '192.168.1.2:11211,192.168.1.3:11211'

client = memcached.Client()
client.set('foo', 'bar')
print(client.get('foo'))
```

The next example uses a prefixed environment variable for configuration data:

```
import os

from sprockets.clients import memcached

os.environ['FOO_MEMCACHED_SERVERS'] = '192.168.1.2:11211'
```

```
client = memcached.Client('foo')
client.set('foo', 'bar')
print(client.get('foo'))
```

3.3 Version History

- 1.1.0 [2018-01-09]
  - Remove usage of python3-memcached since python-memcached supports both Python2 & Python3
  - Drop support for Python 2.6
- 1.0.0 [2014-09-03]
  - Initial release

CHAPTER 4

Version History

See *Version History*

CHAPTER 5

Issues

Please report any issues to the Github project at https://github.com/sprockets/sprockets.clients.memcached/issues

CHAPTER 6

Source

`sprockets.clients.memcached` source is available on Github at https://github.com/sprockets/sprockets.clients.memcached

CHAPTER 7

License

`sprockets.clients.memcached` is released under the 3-Clause BSD license.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index