msaStorageDict

storagedict contains a collection of dictionary classes backed by a data store (Redis, Zookeeper, memory, etc) suitable for use in a distributed system. Dictionary values are cached locally in the instance of the dictionary, and synced with their values and the backed data store when a value in the data store has changed.

Usage

storagedict contains various types of dictionary-like objects backed by a data store. All dict classes are located in the `msaStorageDict` package. Currently the following dicts are supported:

1. `msaStorageDict.redis.MSARedisDict` - Backed by Redis.
2. `msaStorageDict.zookeeper.MSAZookeeperDict` - Backed by Zookeeper.
3. `msaStorageDict.memory.MSAMemoryDict` - Backed by Memory.

Each dictionary class has a different `__init__` method which take different arguments, so consult their documentation for specific usage detail.

Once you have an instance of a storagedict, just use it like you would a normal dictionary:

```
from msaStorageDict import MSARedisDict
from redis import Redis

# Construct a new MSARedisDict object
settings = MSARedisDict('settings', Redis())

# Assign and retrieve a value from the dict
settings['foo'] = 'bar'
settings['foo']
# >>> 'bar'

# Assign and retrieve another value
settings['dont'] = 'trick'
settings['dont']
# >>> 'trick'

# Delete a value and access receives a KeyError
del settings['foo']
settings['foo']
# >>> KeyError
```

All dict types pickle their objects inside their durable data store, so any object that is \"pickleable\" can be saved to those stores.

Notes on Persistence, Consistency and the In-Memory Cache

Nearly all methods called on a msaStorageDict dictionary class (i.e. `MSARedisDict`) are proxied to an internal dict object that serves as a cache for access to dict values. This cache is only updated with fresh values from durable storage if there actually has been a change in the values stored in durable storage.

To check if the data in durable storage has changed, each msaStorageDict backend is responsible for providing a fast `last_updated()` method that quickly tells the dict the last time any value in the durable storage has been updated. For instance, the `msaStorageDict` constructor requires a `cache` object passed in as an argument, which provides implementations of cache-line interface methods for maintaining the `last_updated` state. A memcache client is a good candidate for this object.

Out of the box by default, all msaStorageDict classes will sync with their durable data store on all writes (insert, updates and deletes) as well as immediately before any read operation on the dictionary. This mode provides *high read consistency* of data at the expense of read speed. You can be guaranteed that any read operation on your dict, i.e. `settings['feature']`, will always use the most up to date data. If another consumer of your durable data store has modified a value in that store since you instantiated your object, you will immediately be able to read the new data with your dict instance.

Manually Control MSA Storage Dict Sync

As mentioned above in, the downside to syncing with durable storage before each read of dict data is it lowers your read performance. If you read 100 keys from your dictionary, that means 100 accesses to check `` `last_updated()` ``. Even with a data store as fast as memecache, that adds up quite quickly.

It therefore may be advantageous for you to *not* sync with durable storage before every read from the dict and instead control that syncing manually. To do so, pass `autosync=False` when you construct the dict, i.e.:

```
from msaStorageDict import MSARedisDict
from redis import Redis

# Construct a new RedisDict object that does not sync on reads
settings = MSARedisDict('settings', Redis(), autosync=False)
```

This causes the dictionary behave in the following way:

1. Like normal, the dictionary initializes from the durable data store upon instantiation.
2. Writes (both inserts and updates), along with deletes of values to the dictionary will still automatically sync with the data store each time the operation happens.
3. Any time a dictionary is read from, *only data current in the internal cache is used*. The dict *will not attempt to sync with its durable data store* before reads.
4. To force the dict to attempt to sync with its durable data store, you may call the `sync()` method on the dictionary. As with when `autosync` is false, if `last_update` says there are no changes, the dict will skip updating from durable storage.

A good use case for manual syncing is a read-heavy web application, where you\'re using a msaStorageDict for settings configuration. Very few requests actually *change* the dictionary contents - most simply read from the dictionary. In this situation, you would perhaps only `sync()` at the beginning of a user\'s web request to make sure the dict is up to date, but then not during the request in order to push the response to the user as fast as possible.

Encoding

All `msaStorageDict` implementations accept an `encoding` keyword argument, which defines the encoding object the dictionary should use when serializing data to and from the persistent data store. The overarching goal of the `encoding` is to serialize the dictionary value object into a format suitable for persisting to durable storage, and then at a later date reconstructing that object from its serialized representation into an object in memory.

By default, [msaStorageDict]{.title-ref} uses pickle as its encoding format, which allows it to serialize complex object easily at the expense of `known security implications`:security and other limitations. See `this IBM Developerworks`:devworks article for an overview of Pickle.

In addition to the built in `encoding.PickleEncoding`, `msaStorageDict` also features `encoding.JSONEncoding` which encodes the data as JSON and `encoding.NoOpEncoding` which does not encode the data at all (suitable only for the `MemoryDict` implementation).

Creating Your Own MSA Storage Dict

Creating your own MSA Storage dict is easy. All you need to do is subclass `msaStorageDict.base.MSAStorageDict` and implement the following required interface methods.

1. `persist(key, value)` - Persist `value` at `key` to your data store.
2. `depersist(key)` - Delete the value at `key` from your data store.
3. `durables()` - Return a `key=val` dict of all keys in your data store.
4. `last_updated()` - A comparable value of when the data in your data store was last updated.

You may also implement a couple optional dictionary methods, which `msaStorageDict.base.MSAStorageDict` will call when the actual non-underscored version is called on the dict.

1. `_pop(key[,default])` - If `key` is in the dictionary, remove it and return its value, else return `default`. If `default` is not given and `key` is not in the dictionary, a `KeyError` is raised.
2. `_setdefault(key[,default])` - If key is in the dictionary, return its value. If not, insert key with a value of `default` and return `default`. `default` defaults to `None`.

Last update: September 23, 2022

Created: September 22, 2022