rc Documentation

Release 0.3

Shipeng Feng

May 03, 2016

Contents

Welcome to rc's documentation. Caching can be fun and easy. This is one library that implements cache system for redis in Python. It is for use with web applications and Python scripts. It comes with really handy apis for easy drop-in use with common tasks, including caching decorators. It can be used to build a cache cluster which has a routing system that allows you to automatically set cache on different servers, sharding can be really easy now. You can use it to batch fetch multiple cache results back, for a cluster, it even does the job in parallel, we fetch results from all servers concurrently, which means much higher performance.

It uses the redis server, which is a in-memory key-value data structure server. It does not implement any other backends like filesystem and does not intend to do so. Mostly we want to use a key-value server like redis, if you have special needs, it is easy to write one cache decorator that stores everything in memory using a dict or you can check out other libraries.

CHAPTER 1

User's Guide

This part of the documentation begins with installation, followed by more instructions for doing cache with rc.

1.1 Foreword

Read this before you get started if you are not familiar with cache.

1.1.1 Why

Speed. The main problem with many applications is, they're slow. Each time we get the result, a lot of code are executed. And cache is the easiest way to speed up things. If you are serious about performance, use more caching.

1.1.2 How Caching Works

What does a cache do? Imagine we have a function that takes some time to complete, the idea is that we put the result of that expensive operation into a cache for some time.

Basically we have a cache object that is connected to a remote server or file system or memory. When the request comes in, you check if the result is in the cache, if so, you return it from the cache. Otherwise you execute the calculation and put it in the cache.

Here is a simple example:

```
def get_result():
    rv = cache.get('mykey')
    if rv is None:
        rv = calculate_result()
        cache.set('mykey', rv)
    return rv
```

1.2 Installation

Want to give rc a try quickly? Let's start by installing it, you need Python 2.6 or newer.

1.2.1 virtualenv

Virtualenv might be something you want to use for development! If you do not have it yet, try the following command:

```
$ sudo pip install virtualenv
```

Since we have virtualenv installed now, let's create one working environment:

```
$ mkdir myproject
$ cd myproject
$ virtualenv venv
$ . venv/bin/activate
```

It is time to get rc:

```
$ pip install rc
```

Done!

1.2.2 System Wide

Install it for all users on the system:

```
$ sudo pip install rc
```

1.2.3 Development Version

Try the latest version:

```
$ . venv/bin/activate
$ git clone http://github.com/fengsp/rc.git
$ cd rc
$ python setup.py develop
```

1.3 Quickstart

This page gives a introduction to RC.

1.3.1 A Simple Example

A minimal cache example looks like this:

```
from rc import Cache

cache = Cache()
assert cache.set('key', 'value')
assert cache.get('key') == 'value'
assert cache.get('foo') is None
assert cache.set('list', [1])
assert cache.get('list') == [1]
```

What we are doing here?

1. First we imported the *Cache* class. An instance of this class can be used to cache things with a single redis server.
2. We create one cache instance.
3. We set and get things based on a key.

1.3.2 Build A Cache Cluster

A cache cluster use a redis cluster as backend.

```
from rc import CacheCluster

cache = CacheCluster({
    'cache01': {'host': 'redis-host01'},
    'cache02': {'host': 'redis-host02'},
    'cache03': {'host': 'redis-host03'},
    'cache04': {'host': 'redis-host04', 'db': 1},
})
```

1.3.3 Cache Decorator

```
@cache.cache()
def load(name, offset):
    return load_from_database(name, offset)

rv = load('name', offset=10)
```

1.3.4 Batch Fetch Multiple Cache Results

```
assert cache.get_many('key', 'foo') == ['value', None]

# for cache decorated function
@cache.cache()
def cached_func(param):
    return param

results = []
# with the context manager, the function
# is executed and return a promise
with cache.batch_mode():
    for i in range(10):
        results.append(cached_func(i))
for i, rv in enumerate(results):
    assert rv.value == i
```

1.3.5 Cache Invalidation

```
cache.delete('key')
# for decorated function
cache.invalidate(load, 'name', offset=10)
```

1.4 Tutorial

In this tutorial, we will create a simple blog application. You can learn to cache with rc and Python here. In our blog application, anyone can add or update a post, view all posts.

1.4.1 Create Skeleton

For this simple web application, we choose to use Flask. Here is the basic skeleton:

```
# -*- coding: utf-8 -*-
import time

from flask import Flask
from flask import request, url_for, redirect, abort, render_template_string
from flask_sqlalchemy import SQLAlchemy
from rc import Cache


app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
db = SQLAlchemy(app)
cache = Cache()


def init_db():
    db.create_all()


if __name__ == '__main__':
    init_db()
    app.run()
```

1.4.2 Create Models

Let's declare the models and create the database schema here:

```
class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    content = db.Column(db.Text, nullable=False)
    created_ts = db.Column(db.Integer, nullable=False)
    updated_ts = db.Column(db.Integer, nullable=False)

    def __init__(self, title, content, created_ts, updated_ts):
        self.title = title
        self.content = content
        self.created_ts = created_ts
        self.updated_ts = updated_ts

    def __repr__(self):
        return '<Post %s>' % self.id

    @staticmethod
    def add(title, content):
        current_ts = int(time.time())
```

```
        post = Post(title, content, current_ts, current_ts)
        db.session.add(post)
        db.session.commit()
        cache.invalidate(Post.get_by_id, post.id)
        cache.invalidate(Post.get_all_ids)

    @staticmethod
    def update(post_id, title, content):
        post = Post.query.get(post_id)
        post.title = title
        post.content = content
        post.updated_ts = int(time.time())
        db.session.commit()
        cache.invalidate(Post.get_by_id, post.id)

    @staticmethod
    @cache.cache()
    def get_all_ids():
        posts = Post.query.all()
        return [post.id for post in posts]

    @staticmethod
    @cache.cache()
    def get_by_id(post_id):
        post = Post.query.get(post_id)
        return dict(id=post.id, title=post.title, content=post.content,
                    created_ts=post.created_ts, updated_ts=post.updated_ts)
```

1.4.3 View Functions

We will have four view functions here, they are used to add or update or view a single post, view all posts. The code explains itself:

```
@app.route('/add', methods=['POST'])
def add_post():
    title = request.form['title']
    content = request.form['content']
    Post.add(title, content)
    return redirect(url_for('show_all_posts'))


@app.route('/post/<int:post_id>')
def show_post(post_id):
    post = Post.get_by_id(post_id)
    if post is None:
        abort(404)
    return render_template_string(SHOW_POST_TEMPLATE, post=post)


@app.route('/post/<int:post_id>', methods=['POST'])
def edit_post(post_id):
    post = Post.get_by_id(post_id)
    if post is None:
        abort(404)
    title = request.form['title']
    content = request.form['content']
    Post.update(post_id, title, content)
```

```
    return redirect(url_for('show_all_posts'))


@app.route('/')
def show_all_posts():
    all_post_ids = Post.get_all_ids()
    all_posts = []
    with cache.batch_mode():
        for post_id in all_post_ids:
            all_posts.append(Post.get_by_id(post_id))
    all_posts = [p.value for p in all_posts]
    return render_template_string(ALL_POSTS_TEMPLATE, all_posts=all_posts)
```

1.4.4 Add The Templates

The template for showing all posts is here.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Blog</title>
  </head>
  <body>
    <h1>Blog</h1>
    <form action="{{ url_for('add_post') }}" method=post>
      <dl>
        <dt>Title:
        <dd><input type=text size=50 name=title>
        <dt>Content:
        <dd><textarea name=content rows=5 cols=40></textarea>
        <dd><input type=submit value=Post>
      </dl>
    </form>
    <ul>
    {% for post in all_posts %}
      <li>
        <a href="{{ url_for('show_post', post_id=post.id) }}">
          {{ post.title }}
        </a>
      </li>
    {% endfor %}
    </ul>
  </body>
</html>
```

The template for showing one post is here.

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{ post.title }}</title>
  </head>
  <body>
    <h1>{{ post.title }}</h1>
    <p>{{ post.content }}</p>
    <form action="{{ url_for('edit_post', post_id=post.id) }}" method=post>
      <dl>
```

```
        <dt>Title:
        <dd><input type=text size=50 name=title>
        <dt>Content:
        <dd><textarea name=content rows=5 cols=40></textarea>
        <dd><input type=submit value=Update>
      </dl>
    </form>
  </body>
</html>
```

If you want the full source code check out the tutorial source.

1.5 Cache

This page gives you some details on caching.

1.5.1 Create Cache Object

Keep one cache instance around so we can do caching easily. There are two types of cache in RC, if you are building a small project and one redis server is enough to hold your cache, go with *Cache*, if you are working on a website that is accessed millions of times per day, *CacheCluster* is the ideal solution.

1.5.2 Cache Global Namespace

Namespace is a global thing with one cache object. All cache object can have a namespace, by default, it is not set. The idea is simple, namespace is just one prefix that will be added to all keys set through this cache object. You can use this to distinguist usage A from usage B if you are sharing redis server resources on them. There is a parameter that is used to set this up, a simple demo:

```
from rc import Cache

models_cache = Cache(namespace='models')
templates_cache = Cache(namespace='templates')
```

1.5.3 Cache Function Result

There is one useful decorator api used to cache result for a function, check out *cache()*. Here is a simple example:

```
from rc import Cache

cache = Cache()

@cache.cache()
def load(name, offset):
    return load_from_database(name, offset)

rv = load('name', 10)
```

If you have two functions with same name inside one module, use *key_prefix* to distinguish them:

```
class Data(object):
    @cache.cache(key_prefix='another')
    def load(self, name, offset):
        return load_from_another_place(name, offset)
```

1.5.4 Cache Expiration Time

The cache expires automatically in time seconds, there is one *default_expire* that is used for all set on a cache object:

```
cache = Cache(default_expire=24 * 3600)  # one day
```

Of course you can change it on every cache set:

```
cache.set('key', 'value', expire=3600)  # one hour
```

1.5.5 Cache Invalidation

For a cache key that is set manually by you, simplily delete it:

```
cache.delete('key')
```

In order to invalidate a cached function with certain arguments:

```
@cache.cache()
def load(name, offset):
    return load_from_database(name, offset)

rv = load('name', offset=10)

# always invalidate using the same positional and keyword parameters
# as you call the function
cache.invalidate(load, 'name', offset=10)
```

What if you want to expire all results of this function with any parameters, since we have a *key_prefix*, just change it to a different value, like from `'version01'` to `'version02'`, the data of old version wouldn't be deleted immediately, however, they are going to be pushed out after expiration time.

1.5.6 Cache Batch Fetching

For a simple key usage, you just need *get_many()*, here is one simple example:

```
assert cache.get_many('key', 'foo') == ['value', None]
```

For cache decorated function, you need *batch_mode()*, check the api for more details. Basically we record all functions you want to execute and return a *Promise* object. When you leaves the batch context manager, the promise is resolved and the result value is there for you.

1.5.7 Bypass Values

New in version 0.3.

When you are using the cache decorator, sometimes you don't want to cache certain return value of decorated functions, you can bypass them:

```
cache = Cache(bypass_values=[None])

@cache.cache()
def load():
    # this won't be cached
    return None
```

1.6 Cache Config

This page gives you introductions on creating a *Cache* instance.

1.6.1 Basic Config

Cache takes parameters for basic redis server setup and cache setup. Here is one simple demo:

```
from rc import Cache

cache = Cache('redishost01', 6379, db=0, password='pass',
              socket_timeout=5)
```

There are other parameters you can config, you can specify your own customized serializer_cls, you can change default expiration time to any length you want, you can set a namespace for this cache instance, for more details, check out *Cache*.

1.6.2 Redis Options

There is one parameter called `redis_options`, you can use this to set other parameters to the underlying `redis.StrictRedis`. Here is a simple example:

```
from rc import Cache

cache = Cache(redis_options={'unix_socket_path': '/tmp/redis.sock'})
```

1.7 Cache Cluster Config

This page gives you introductions on how to create a *CacheCluster* instance.

1.7.1 Basic Config

Simple demo:

```
from rc import CacheCluster

cache = CacheCluster({
    0: {'host': 'redis-host-01'},
    1: {'host': 'redis-host-02', 'port': 6479},
    2: {'unix_socket_path': '/tmp/redis.sock'},
    3: {'host': 'redis-host-03', 'db': 1},
})
```

Basically *hosts* is just one dictionary that maps host name to parameters which are taken by *HostConfig*, excluding *host_name*.

Just like *Cache*, you can specify your own customized *serializer_cls*, you can change default expiration time to any length you want, you can set a namespace for this cache instance, for more details, check out *CacheCluster*.

1.7.2 Redis Connection Pool Config

By default we use `ConnectionPool`, specify your own connection pool class and options using parameters called `pool_cls` and `pool_options`.

1.7.3 Redis Router Config

By default we use *RedisCRC32HashRouter*, specify your own router class and options using parameters called `router_cls` and `router_options`. For more routers, check out *Redis Cluster Router*.

1.7.4 Concurrency Config

For operations on multiple keys like *get_many*, *set_many* and *delete_many*, we execute them in parallel. Under the hood, we does the parallel query using a select loop(select/poll/kqueue/epoll). Mostly, if we are using several remote redis servers, we achieve higher performance. You can specify `max_concurrency` and `poller_timeout` to control maximum concurrency and timeout for poller.

1.8 Serializer

This page we talk about serializers.

1.8.1 JSON Serializer

It is simple and fast. The downside is that it cannot serialize enough types of Python objects. For more details check out *JSONSerializer*.

1.8.2 Pickle Serializer

More Python types are supported. The downside is that it might be slower than JSON, unpickling can run arbitrary code, and using *pickle* to transfer data between programs in different languages is almost impossible, check out *PickleSerializer*.

1.8.3 Build Your Own Serializer

Subclass *BaseSerializer*, implement *dumps()* and *loads()*.

Here is one simple example:

```
import json

from rc.serializer import BaseSerializer

class IterEncoder(json.JSONEncoder):
```

```
    def default(self, o):
        try:
           iterable = iter(o)
        except TypeError:
           pass
        else:
           return list(iterable)
        return json.JSONEncoder.default(self, o)

class MyJSONSerializer(BaseSerializer):
    """One serializer that uses JSON and support arbitrary iterators"""

    def dumps(self, obj):
        return json.dumps(obj, cls=IterEncoder)

    def loads(self, string):
        if string is None:
            return
        return json.loads(string)
```

1.9 Redis Cluster Router

This page gives you introductions on redis router for cluster.

1.9.1 CRC32Hash Router

Router that just routes commands to redis node based on `crc32 % node_num`. For more details check out *RedisCRC32HashRouter*.

1.9.2 ConsistentHash Router

Router that routes to redis based on consistent hashing algorithm. For more details check out *RedisConsistentHashRouter*.

1.9.3 Build Your Own Router

Subclass *BaseRedisRouter*, implement *get_host_for_key()*.

Here is the builtin CRC32 router:

```
from binascii import crc32


class RedisCRC32HashRouter(BaseRedisRouter):
    """Use crc32 for hash partitioning."""

    def __init__(self, hosts):
        BaseRedisRouter.__init__(self, hosts)
        self._sorted_host_names = sorted(hosts.keys())

    def get_host_for_key(self, key):
        if isinstance(key, unicode):
```

```
            key = key.encode('utf-8')
        else:
            key = str(key)
        pos = crc32(key) % len(self._sorted_host_names)
        return self._sorted_host_names[pos]
```

1.10 Testing

Testing applications that use RC.

1.10.1 Null Cache

Simple idea, just create one *NullCache* object that does not cache at all when you are doing unit test.

1.10.2 Fake Redis

Use a fake redis as backend, this is existing for testing purposes only. It depends on the fakeredis library, install it first:

```
$ pip install fakeredis
```

For more details, check out *FakeRedisCache*.

1.11 Patterns On Caching

This part contains some snippets and patterns for caching.

1.11.1 Cache In Memory

RC doesn't support other backends, because mostly you want to use a cache server. But if you really need to put some cache in memory, it should be easy:

```
from functools import wraps

def cache(func):
    saved = {}
    @wraps(func)
    def newfunc(*args):
        if args in saved:
            return saved[args]
        result = func(*args)
        saved[args] = result
        return result
    return newfunc

@cache
def lookup(url):
    return url
```

CHAPTER 2

API Reference

This part of the documentation contains information on a specific function, class or method.

2.1 API

This page covers all interfaces of RC.

2.1.1 Base Cache System API

class `rc.cache.`**BaseCache**(*namespace=None*, *serializer_cls=None*, *default_expire=259200*, *bypass_values=[]*)

Baseclass for all redis cache systems.

Parameters

- **namespace** – a prefix that should be added to all keys
- **serializer_cls** – the serialization class you want to use.
- **default_expire** – default expiration time that is used if no expire specified on `set()`.
- **bypass_values** – a list of return values that would be ignored by the cache decorator and won't be cached at all.

New in version 0.3: The *bypass_values* parameter was added.

batch_mode()

Returns a context manager for cache batch mode. This is used to batch fetch results of cache decorated functions. All results returned by cache decorated function will be `Promise` object. This context manager runs the batch fetch and then resolves all promises in the end. Example:

```
results = []
with cache.batch_mode():
    for i in range(10):
        results.append(get_result(i))
results = map(lambda r: r.value, results)
```

Note: When you are using rc on this mode, rc is not thread safe.

cache(*key_prefix=None*, *expire=None*, *include_self=False*)

A decorator that is used to cache a function with supplied parameters. It is intended for decorator usage:

```
@cache.cache()
def load(name):
    return load_from_database(name)

rv = load('foo')
rv = load('foo') # returned from cache
```

The cache key doesn't need to be specified, it will be created with the name of the module + the name of the function + function arguments.

Parameters

- **key_prefix** – this is used to ensure cache result won't clash with another function that has the same name in this module, normally you do not need to pass this in
- **expire** – expiration time
- **include_self** – whether to include the *self* or *cls* as cache key for method or not, default to be False

---

Note: The function being decorated must be called with the same positional and keyword arguments. Otherwise, you might create multiple caches. If you pass one parameter as positional, do it always.

Note: Using objects as part of the cache key is possible, though it is suggested to not pass in an object instance as parameter. We perform a str() on the passed in objects so that you can provide a __str__ function that returns a identifying string for that object, the unique string will be used as part of the cache key.

Note: When a method on a class is decorated, the `self` or `cls` arguments is not included in the cache key. Starting from 0.2 you can control it with *include_self*. If you set *include_self* to True, remember to provide *__str__* method for the object, otherwise you might encounter random behavior.

New in version 0.2: The *include_self* parameter was added.

delete(*key*)
Deletes the value for the cache key.

Parameters **key** – cache key

Returns Whether the key has been deleted

delete_many(**keys*)
Deletes multiple keys.

Returns whether all keys has been deleted

get(*key*)
Returns the value for the cache key, otherwise *None* is returned.

Parameters **key** – cache key

get_client()
Returns the redis client that is used for cache.

get_many(**keys*)
Returns the a list of values for the cache keys.

invalidate(*func*, **args*, ***kwargs*)
Invalidate a cache decorated function. You must call this with the same positional and keyword arguments as what you did when you call the decorated function, otherwise the cache will not be deleted. The usage is simple:

```
@cache.cache()
def load(name, limit):
    return load_from_database(name, limit)

rv = load('foo', limit=5)

cache.invalidate(load, 'foo', limit=5)
```

Parameters

- **func** – decorated function to invalidate
- **args** – same positional arguments as you call the function
- **kwargs** – same keyword arguments as you call the function

Returns whether it is invalidated or not

set(*key*, *value*, *expire=None*)
Adds or overwrites key/value to the cache. The value expires in time seconds.

Parameters

- **key** – cache key
- **value** – value for the key
- **expire** – expiration time

Returns Whether the key has been set

set_many(*mapping*, *expire=None*)
Sets multiple keys and values using dictionary. The values expires in time seconds.

Parameters

- **mapping** – a dictionary with key/values to set
- **expire** – expiration time

Returns whether all keys has been set

2.1.2 Cache Object

class rc.**Cache**(*host='localhost'*, *port=6379*, *db=0*, *password=None*, *socket_timeout=None*, *namespace=None*, *serializer_cls=None*, *default_expire=259200*, *redis_options=None*, *bypass_values=[]*)
Uses a single Redis server as backend.

Parameters

- **host** – address of the Redis, this is compatible with the official Python StrictRedis cilent (redis-py).
- **port** – port number of the Redis server.
- **db** – db numeric index of the Redis server.

- **password** – password authentication for the Redis server.
- **socket_timeout** – socket timeout for the StrictRedis client.
- **namespace** – a prefix that should be added to all keys.
- **serializer_cls** – the serialization class you want to use. By default, it is *rc.JSONSerializer*.
- **default_expire** – default expiration time that is used if no expire specified on `set()`.
- **redis_options** – a dictionary of parameters that are useful for setting other parameters to the StrictRedis client.
- **bypass_values** – a list of return values that would be ignored by the cache decorator and won't be cached at all.

New in version 0.3: The *bypass_values* parameter was added.

2.1.3 Cache Cluster Object

class `rc.`**CacheCluster**(*hosts*, *namespace=None*, *serializer_cls=None*, *default_expire=259200*, *router_cls=None*, *router_options=None*, *pool_cls=None*, *pool_options=None*, *max_concurrency=64*, *poller_timeout=1.0*, *bypass_values=[]*)

The a redis cluster as backend.

Basic example:

```
cache = CacheCluster({
    0: {'port': 6379},
    1: {'port': 6479},
    2: {'port': 6579},
    3: {'port': 6679},
})
```

Parameters

- **hosts** – a dictionary of hosts that maps the host host_name to configuration parameters. The parameters are used to construct a *HostConfig*.
- **namespace** – a prefix that should be added to all keys.
- **serializer_cls** – the serialization class you want to use. By default, it is *JSONSerializer*.
- **default_expire** – default expiration time that is used if no expire specified on `set()`.
- **router_cls** – use this to override the redis router class, default to be *RedisCRC32HashRouter*.
- **router_options** – a dictionary of parameters that is useful for setting other parameters of router
- **pool_cls** – use this to override the redis connection pool class, default to be ConnectionPool
- **pool_options** – a dictionary of parameters that is useful for setting other parameters of pool
- **max_concurrency** – defines how many parallel queries can happen at the same time

- **poller_timeout** – for multi key operations we use a select loop as the parallel query implementation, use this to specify timeout for the underlying pollers (select/poll/kqueue/epoll).
- **bypass_values** – a list of return values that would be ignored by the cache decorator and won't be cached at all.

New in version 0.3: The *bypass_values* parameter was added.

2.1.4 Serializer

class rc.**BaseSerializer**
Baseclass for serializer. Subclass this to get your own serializer.

dumps(*obj*)
Dumps an object into a string for redis.

loads(*string*)
Read a serialized object from a string.

class rc.**JSONSerializer**
One serializer that uses JSON

class rc.**PickleSerializer**
One serializer that uses Pickle

2.1.5 Redis Router

The base router class provides a simple way to replace the router cls that cache cluster is using.

class rc.**BaseRedisRouter**(*hosts*)
Subclass this to implement your own router.

get_host_for_key(*key*)
Get host name for a certain key.

class rc.**RedisCRC32HashRouter**(*hosts*)
Use crc32 for hash partitioning.

class rc.**RedisConsistentHashRouter**(*hosts*)
Use ketama for hash partitioning.

2.1.6 Testing Objects

class rc.**NullCache**(**args*, ***kwargs*)
Use this for unit test. This doesn't cache.

delete(*key*)
Always return *True*

get(*key*)
Always return *None*

get_many(**keys*)
Always return a list of *None*

set(*key*, *value*, *time=None*)
Always return *True*

class rc.**FakeRedisCache**(*namespace=None*, *serializer_cls=None*, *default_expire=259200*)
Uses a fake redis server as backend. It depends on the fakeredis library.

Parameters

- **namespace** – a prefix that should be added to all keys.
- **serializer_cls** – the serialization class you want to use. By default, it is *rc.JSONSerializer*.
- **default_expire** – default expiration time that is used if no expire specified on `set()`.

2.1.7 Cluster Host Config

class rc.redis_cluster.**HostConfig**(*host_name*, *host='localhost'*, *port=6379*, *unix_socket_path=None*, *db=0*, *password=None*, *ssl=False*, *ssl_options=None*)

2.1.8 Promise Object

class rc.promise.**Promise**
A promise object. You can access `promise.value` to get the resolved value. Here is one example:

```
p = Promise()
assert p.is_pending
assert not p.is_resolved
assert p.value is None
p.resolve('value')
assert not p.is_pending
assert p.is_resolved
assert p.value == 'value'
```

is_pending
Return *True* if the promise is pending.

is_resolved
Return *True* if the promise is resolved.

resolve(*value*)
Resolves with value.

then(*on_resolve=None*)
Add one callback that is called with the resolved value when the promise is resolved, and return the promise itself. One demo:

```
p = Promise()
d = {}
p.then(lambda v: d.setdefault('key', v))
p.resolve('value')
assert p.value == 'value'
assert d['key'] == 'value'
```

CHAPTER 3

Additional Stuff

Changelog and license here if you are interested.

3.1 RC Changelog

3.1.1 Version 0.1

First public preview release.

3.1.2 Version 0.1.1

Bugfix release, released on Dec 18th 2015

- Make the cache decorated function always return result through the serializer, so we get consistent result

3.1.3 Version 0.2

Released on Dec 23th 2015

- Added include_self parameter for cache decorated function, now we can cache instance method.

3.1.4 Version 0.2.1

Enhancement release, released on Jan 15th 2016

- Only check has_self once to get rid of the inspect performance issue

3.1.5 Version 0.3

Released on May 3th 2016

- Added bypass_values parameter for cache object, now we can bypass certain return values of cache decorated function

3.2 License

r