rb documentation

Release 1.0

Function Software Inc.

Apr 03, 2023

Contents

Rb, the redis blaster, is a library that implements non-replicated sharding for redis. It implements a custom routing system on top of python redis that allows you to automatically target different servers without having to manually route requests to the individual nodes.

It does not implement all functionality of redis and does not attempt to do so. You can at any point get a client to a specific host, but for the most part the assumption is that your operations are limited to basic key/value operations that can be routed to different nodes automatically.

What you can do:

- automatically target hosts for single-key operations
- execute commands against all or a subset of nodes
- do all of that in parallel

CHAPTER 1

Installation

rb is available on PyPI and can be installed from there:

```
$ pip install rb
```

CHAPTER 2

Configuration

Getting started with rb is super easy. If you have been using py-redis before you will feel right at home. The main difference is that instead of connecting to a single host, you configure a cluster to connect to multiple:

```
from rb import Cluster

cluster = Cluster(hosts={
    0: {'port': 6379},
    1: {'port': 6380},
    2: {'port': 6381},
    3: {'port': 6382},
    4: {'port': 6379},
    5: {'port': 6380},
    6: {'port': 6381},
    7: {'port': 6382},
}, host_defaults={
    'host': '127.0.0.1',
})
```

In this case we set up 8 nodes on four different server processes on the same host. The *hosts* parameter is a mapping of hosts to connect to. The key of the dictionary is the host ID (an integer) and the value is a dictionary of parameters. The *host_defaults* is a dictionary of optional defaults that is filled in for all hosts. This is useful if you want to share some common defaults that repeat (in this case all hosts connect to localhost).

In the default configuration the *PartitionRouter* is used for routing.

CHAPTER 3

Routing

Now that the cluster is constructed we can use *Cluster.get_routing_client()* to get a redis client that automatically routes to the right redis nodes for each command:

```
client = cluster.get_routing_client()
results = {}
for key in keys_to_look_up:
    results[key] = client.get(key)
```

The client works pretty much exactly like a standard pyredis *StrictClient* with the main difference that it can only execute commands that involve exactly one key.

This basic operation however runs in series. What makes rb useful is that it can automatically build redis pipelines and send out queries to many hosts in parallel. This however changes the usage slightly as now the value is not immediately available:

```
results = {}
with cluster.map() as client:
    for key in keys_to_look_up:
        results[key] = client.get(key)
```

While it looks similar so far, instead of storing the actual values in the result dictionary, *Promise* objects are stored instead. When the map context manager ends they are guaranteed however to have been executed and you can access the *Promise.value* attribute to get the value:

```
for key, promise in results.iteritems():
    print '%s: %s' % (key, promise.value)
```

If you want to send a command to all participating hosts (for instance to delete the database) you can use the *Cluster.all()* method:

```
with cluster.all() as client:
    client.flushdb()
```

If you do that, the promise value is a dictionary with the host IDs as keys and the results as value. As an example:

```
with cluster.all() as client:
    results = client.info()
for host_id, info in results.iteritems():
    print 'host %s is running %s' % (host_id, info['os'])
```

To explicitly target some hosts you can use *Cluster.fanout()* which accepts a list of host IDs to send the command to.

CHAPTER 4

API

This is the entire reference of the public API. Note that this library extends the Python redis library so some of these classes have more functionality for which you will need to consult the py-redis library.

4.1 Cluster

class rb.**Cluster**(*hosts*, *host_defaults=None*, *pool_cls=None*, *pool_options=None*, *router_cls=None*, *router_options=None*)

The cluster is the core object behind rb. It holds the connection pools to the individual nodes and can be shared for the duration of the application in a central location.

Basic example of a cluster over four redis instances with the default router:

```
cluster = Cluster(hosts={
    0: {'port': 6379},
    1: {'port': 6380},
    2: {'port': 6381},
    3: {'port': 6382},
}, host_defaults={
    'host': '127.0.0.1',
})
```

hosts is a dictionary of hosts which maps the number host IDs to configuration parameters. The parameters correspond to the signature of the *add_host()* function. The defaults for these parameters are pulled from *host_defaults*. To override the pool class the *pool_cls* and *pool_options* parameters can be used. The same applies to *router_cls* and *router_options* for the router. The pool options are useful for setting socket timeouts and similar parameters.

add_host(*host_id=None*, *host='localhost'*, *port=6379*, *unix_socket_path=None*, *db=0*, *password=None*, *ssl=False*, *ssl_options=None*)

Adds a new host to the cluster. This is only really useful for unittests as normally hosts are added through the constructor and changes after the cluster has been used for the first time are unlikely to make sense.

all (*timeout=None*, *max_concurrency=64*, *auto_batch=True*)
Fanout to all hosts. Works otherwise exactly like *fanout()*.

Example:

```
with cluster.all() as client:
    client.flushdb()
```

disconnect_pools()
Disconnects all connections from the internal pools.

execute_commands (*mapping*, **args*, ***kwargs*)
Concurrently executes a sequence of commands on a Redis cluster that are associated with a routing key, returning a new mapping where values are a list of results that correspond to the command in the same position. For example:

```
>>> cluster.execute_commands({
...   'foo': [
...     ('PING',),
...     ('TIME',),
...   ],
...   'bar': [
...     ('CLIENT', 'GETNAME'),
...   ],
... })
{'bar': [<Promise None>],
 'foo': [<Promise True>, <Promise (1454446079, 418404)>]}
```

Commands that are instances of `redis.client.Script` will first be checked for their existence on the target nodes then loaded on the targets before executing and can be interleaved with other commands:

```
>>> from redis.client import Script
>>> TestScript = Script(None, 'return {KEYS, ARGV}')
>>> cluster.execute_commands({
...   'foo': [
...     (TestScript, ('key:1', 'key:2'), range(0, 3)),
...   ],
...   'bar': [
...     (TestScript, ('key:3', 'key:4'), range(3, 6)),
...   ],
... })
{'bar': [<Promise [['key:3', 'key:4'], ['3', '4', '5']]>],
 'foo': [<Promise [['key:1', 'key:2'], ['0', '1', '2']]>]}
```

Internally, *FanoutClient* is used for issuing commands.

fanout (*hosts=None*, *timeout=None*, *max_concurrency=64*, *auto_batch=True*)
Shortcut context manager for getting a routing client, beginning a fanout operation and joining over the result.

In the context manager the client available is a *FanoutClient*. Example usage:

```
with cluster.fanout(hosts='all') as client:
    client.flushdb()
```

get_local_client (*host_id*)
Returns a localized client for a specific host ID. This client works like a regular Python redis client and returns results immediately.

get_local_client_for_key(*key*)
Similar to *get_local_client_for_key()* but returns the client based on what the router says the key destination is.

get_pool_for_host(*host_id*)
Returns the connection pool for the given host.

This connection pool is used by the redis clients to make sure that it does not have to reconnect constantly. If you want to use a custom redis client you can pass this in as connection pool manually.

get_router()
Returns the router for the cluster. If the cluster reconfigures the router will be recreated. Usually you do not need to interface with the router yourself as the cluster's routing client does that automatically.

This returns an instance of *BaseRouter*.

get_routing_client(*auto_batch=True*)
Returns a routing client. This client is able to automatically route the requests to the individual hosts. It's thread safe and can be used similar to the host local client but it will refused to execute commands that cannot be directly routed to an individual node.

The default behavior for the routing client is to attempt to batch eligible commands into batch versions thereof. For instance multiple *GET* commands routed to the same node can end up merged into an *MGET* command. This behavior can be disabled by setting *auto_batch* to *False*. This can be useful for debugging because *MONITOR* will more accurately reflect the commands issued in code.

See *RoutingClient* for more information.

map(*timeout=None*, *max_concurrency=64*, *auto_batch=True*)
Shortcut context manager for getting a routing client, beginning a map operation and joining over the result. *max_concurrency* defines how many outstanding parallel queries can exist before an implicit join takes place.

In the context manager the client available is a *MappingClient*. Example usage:

```
results = {}
with cluster.map() as client:
    for key in keys_to_fetch:
        results[key] = client.get(key)
for key, promise in results.iteritems():
    print '%s => %s' % (key, promise.value)
```

remove_host(*host_id*)
Removes a host from the client. This only really useful for unittests.

4.2 Clients

class rb.**RoutingClient**(*cluster*, *auto_batch=True*)
A client that can route to individual targets.

For the parameters see *Cluster.get_routing_client()*.

execute_command(**args*, ***options*)
Execute a command and return a parsed response

fanout(*hosts=None*, *timeout=None*, *max_concurrency=64*, *auto_batch=None*)
Returns a context manager for a map operation that fans out to manually specified hosts instead of using the routing system. This can for instance be used to empty the database on all hosts. The context manager returns a *FanoutClient*. Example usage:

```
with cluster.fanout(hosts=[0, 1, 2, 3]) as client:
    results = client.info()
for host_id, info in results.value.iteritems():
    print '%s -> %s' % (host_id, info['is'])
```

The promise returned accumulates all results in a dictionary keyed by the *host_id*.

The *hosts* parameter is a list of *host_id*s or alternatively the string `'all'` to send the commands to all hosts.

The fanout APi needs to be used with a lot of care as it can cause a lot of damage when keys are written to hosts that do not expect them.

get_fanout_client(*hosts*, *max_concurrency=64*, *auto_batch=None*)
Returns a thread unsafe fanout client.

Returns an instance of *FanoutClient*.

get_mapping_client(*max_concurrency=64*, *auto_batch=None*)
Returns a thread unsafe mapping client. This client works similar to a redis pipeline and returns eventual result objects. It needs to be joined on to work properly. Instead of using this directly you shold use the *map()* context manager which automatically joins.

Returns an instance of *MappingClient*.

map(*timeout=None*, *max_concurrency=64*, *auto_batch=None*)
Returns a context manager for a map operation. This runs multiple queries in parallel and then joins in the end to collect all results.

In the context manager the client available is a *MappingClient*. Example usage:

```
results = {}
with cluster.map() as client:
    for key in keys_to_fetch:
        results[key] = client.get(key)
for key, promise in results.iteritems():
    print '%s => %s' % (key, promise.value)
```

class rb.**MappingClient**(*connection_pool*, *max_concurrency=None*, *auto_batch=True*)
The routing client uses the cluster's router to target an individual node automatically based on the key of the redis command executed.

For the parameters see *Cluster.map()*.

cancel()
Cancels all outstanding requests.

execute_command(**args*, ***options*)
Execute a command and return a parsed response

join(*timeout=None*)
Waits for all outstanding responses to come back or the timeout to be hit.

mget(*keys*, **args*)
Returns a list of values ordered identically to `keys`

For more information see https://redis.io/commands/mget

mset(**args*, ***kwargs*)
Sets key/values based on a mapping. Mapping is a dictionary of key/value pairs. Both keys and values should be strings or types that can be cast to a string via str().

For more information see https://redis.io/commands/mset

class rb.**FanoutClient**(*hosts*, *connection_pool*, *max_concurrency=None*, *auto_batch=True*)

This works similar to the `MappingClient` but instead of using the router to target hosts, it sends the commands to all manually specified hosts.

The results are accumulated in a dictionary keyed by the *host_id*.

For the parameters see `Cluster.fanout()`.

execute_command(**args*, ***options*)

Execute a command and return a parsed response

target(*hosts*)

Temporarily retarget the client for one call. This is useful when having to deal with a subset of hosts for one call.

target_key(*key*)

Temporarily retarget the client for one call to route specifically to the one host that the given key routes to. In that case the result on the promise is just the one host's value instead of a dictionary.

New in version 1.3.

4.3 Promise

class rb.**Promise**

A promise object that attempts to mirror the ES6 APIs for promise objects. Unlike ES6 promises this one however also directly gives access to the underlying value and it has some slightly different static method names as this promise can be resolved externally.

static all(*iterable_or_dict*)

A promise that resolves when all passed promises resolve. You can either pass a list or a dictionary of promises.

done(*on_success=None*, *on_failure=None*)

Attaches some callbacks to the promise and returns the promise.

is_pending

True if the promise is still pending, *False* otherwise.

is_rejected

True if the promise was rejected, *False* otherwise.

is_resolved

True if the promise was resolved, *False* otherwise.

reason

the reason for this promise if it's rejected.

reject(*reason*)

Rejects the promise with the given reason.

static rejected(*reason*)

Creates a promise object rejected with a certain value.

resolve(*value*)

Resolves the promise with the given value.

static resolved(*value*)

Creates a promise object resolved with a certain value.

then(*success=None*, *failure=None*)
A utility method to add success and/or failure callback to the promise which will also return another promise in the process.

value
the value that this promise holds if it's resolved.

4.4 Routers

class rb.**BaseRouter**(*cluster*)
Baseclass for all routers. If you want to implement a custom router this is what you subclass.

cluster
Reference back to the *Cluster* this router belongs to.

get_host_for_command(*command*, *args*)
Returns the host this command should be executed against.

get_host_for_key(*key*)
Perform routing and return host_id of the target.

Subclasses need to implement this.

get_key(*command*, *args*)
Returns the key a command operates on.

class rb.**ConsistentHashingRouter**(*cluster*)
Router that returns the host_id based on a consistent hashing algorithm. The consistent hashing algorithm only works if a key argument is provided.

This router requires that the hosts are gapless which means that the IDs for N hosts range from 0 to N-1.

get_host_for_key(*key*)
Perform routing and return host_id of the target.

Subclasses need to implement this.

class rb.**PartitionRouter**(*cluster*)
A straightforward router that just individually routes commands to single nodes based on a simple `crc32 % node_count` setup.

This router requires that the hosts are gapless which means that the IDs for N hosts range from 0 to N-1.

get_host_for_key(*key*)
Perform routing and return host_id of the target.

Subclasses need to implement this.

exception rb.**UnroutableCommand**
Raised if a command was issued that cannot be routed through the router to a single host.

4.5 Testing

class rb.testing.**TestSetup**(*servers=4*, *databases_each=8*, *server_executable='redis-server'*)
The test setup is a convenient way to spawn multiple redis servers for testing and to shut them down automatically. This can be used as a context manager to automatically terminate the clients.

rb.testing.**make_test_cluster**(**args*, ***kwargs*)
Convenient shortcut for creating a test setup and then a cluster from it. This must be used as a context manager:

```
from rb.testing import make_test_cluster
with make_test_cluster() as cluster:
    ...
```

Python Module Index

r

Index

T

U

V