<https://github.com/xetorthio/jedis/wiki/AdvancedUsage>

1. AdvancedUsage

KAWACHI Takashi edited this page on 15 Jul · [66 revisions](https://github.com/xetorthio/jedis/wiki/AdvancedUsage/_history)

1. **Advanced usage**
2. **Transactions**

To do transactions in Jedis, you have to wrap operations in a transaction block, very similar to pipelining:

jedis**.**watch **(**key1**,** key2**,** **...);**

Transaction t **=** jedis**.**multi**();**

t**.**set**(**"foo"**,** "bar"**);**

t**.**exec**();**

Note: when you have any method that returns values, you have to do like this:

Transaction t **=** jedis**.**multi**();**

t**.**set**(**"fool"**,** "bar"**);**

Response**<**String**>** result1 **=** t**.**get**(**"fool"**);**

t**.**zadd**(**"foo"**,** 1**,** "barowitch"**);** t**.**zadd**(**"foo"**,** 0**,** "barinsky"**);** t**.**zadd**(**"foo"**,** 0**,** "barikoviev"**);**

Response**<**Set**<**String**>>** sose **=** t**.**zrange**(**"foo"**,** 0**,** **-**1**);** *// get the entire sortedset*

t**.**exec**();** *// dont forget it*

String foolbar **=** result1**.**get**();** *// use Response.get() to retrieve things from a Response*

**int** soseSize **=** sose**.**get**().**size**();** *// on sose.get() you can directly call Set methods!*

*// List<Object> allResults = t.exec(); // you could still get all results at once, as before*

Note that a Response Object does not contain the result before t.exec() is called (it is a kind of a Future). Forgetting exec gives you exceptions. In the last lines, you see how transactions/pipelines were dealt with before version 2. You can still do it that way, but then you need to extract objects from a list, which contains also Redis status messages.

Note 2: Redis does not allow to use intermediate results of a transaction within that same transaction. This does not work:

*// this does not work! Intra-transaction dependencies are not supported by Redis!*

jedis**.**watch**(...);**

Transaction t **=** jedis**.**multi**();**

**if(**t**.**get**(**"key1"**).**equals**(**"something"**))**

t**.**set**(**"key2"**,** "value2"**);**

**else**

t**.**set**(**"key"**,** "value"**);**

However, there are some commands like setnx, that include such a conditional execution. Those are of course supported within transactions. You can build your own customized commands using eval/ LUA scripting.

1. **Pipelining**

Sometimes you need to send a bunch of different commands. A very cool way to do that, and have better performance than doing it the naive way, is to use pipelining. This way you send commands without waiting for response, and you actually read the responses at the end, which is faster.

Here is how to do it:

Pipeline p **=** jedis**.**pipelined**();**

p**.**set**(**"fool"**,** "bar"**);**

p**.**zadd**(**"foo"**,** 1**,** "barowitch"**);** p**.**zadd**(**"foo"**,** 0**,** "barinsky"**);** p**.**zadd**(**"foo"**,** 0**,** "barikoviev"**);**

Response**<**String**>** pipeString **=** p**.**get**(**"fool"**);**

Response**<**Set**<**String**>>** sose **=** p**.**zrange**(**"foo"**,** 0**,** **-**1**);**

p**.**sync**();**

**int** soseSize **=** sose**.**get**().**size**();**

Set**<**String**>** setBack **=** sose**.**get**();**

For more explanations see code comments in the transaction section.

1. **Publish/Subscribe**

To subscribe to a channel in Redis, create an instance of JedisPubSub and call subscribe on the Jedis instance:

**class** **MyListener** **extends** JedisPubSub **{**

**public** **void** **onMessage(**String channel**,** String message**)** **{**

**}**

**public** **void** **onSubscribe(**String channel**,** **int** subscribedChannels**)** **{**

**}**

**public** **void** **onUnsubscribe(**String channel**,** **int** subscribedChannels**)** **{**

**}**

**public** **void** **onPSubscribe(**String pattern**,** **int** subscribedChannels**)** **{**

**}**

**public** **void** **onPUnsubscribe(**String pattern**,** **int** subscribedChannels**)** **{**

**}**

**public** **void** **onPMessage(**String pattern**,** String channel**,**

String message**)** **{**

**}**

**}**

MyListener l **=** **new** **MyListener();**

jedis**.**subscribe**(**l**,** "foo"**);**

Note that subscribe is a blocking operation because it will poll Redis for responses on the thread that calls subscribe. A single JedisPubSub instance can be used to subscribe to multiple channels. You can call subscribe or psubscribe on an existing JedisPubSub instance to change your subscriptions.

1. **ShardedJedis**
2. **Motivation**

In the normal Redis master-slave approach, generally there is one master that serves write requests, and many slaves that serve read requests. This means, the user has to take care of effectively distributing the load on the slaves. Furthermore, only reads scale with the number of slaves, but writes do not, since there can be only one master! With ShardedJedis you achieve scalability for both reads and writes. Sharding uses a technique called "consistent hashing" and assigns the keys equally on a set of redis servers according to some hash algorithm (md5 and murmur, the latter being less standard, but faster). A node like this is then called a "shard". A further advantage is that each shards only needs to have RAM 1/n the size of the total dataset (for n being the number of participating slaves).

1. **The downside**

Since each shard is a separate master, sharding has limited functionality: i.e. you cannot use transactions, pipelining, pub/sub, especially not across shards! However, generally it is feasible to do a not allowed operation, as long as the concerned keys are on the same shard (check / ask the forum). You can influence which key go to which shard by keytags (see below). A further downside is that in the current standard implementation, shards cannot be added or removed from a running ShardedJedis. If you need this feature, there is an experimental reimplementation of ShardedJedis which allows adding and removing shards of a running ShardedJedis: [yaourt - dynamic sharding implementation](https://github.com/xetorthio/jedis/pull/174)

1. **General Usage:**
2. **1. Define your shards:**

List**<**JedisShardInfo**>** shards **=** **new** ArrayList**<**JedisShardInfo**>();**

JedisShardInfo si **=** **new** **JedisShardInfo(**"localhost"**,** 6379**);**

si**.**setPassword**(**"foobared"**);**

shards**.**add**(**si**);**

si **=** **new** **JedisShardInfo(**"localhost"**,** 6380**);**

si**.**setPassword**(**"foobared"**);**

shards**.**add**(**si**);**

Then, there are two ways of using ShardedJedis. Direct connections or by using ShardedJedisPool. For reliable operation, the latter has to be used in a multithreaded environment.

1. **2.a) Direct connection:**

ShardedJedis jedis **=** **new** **ShardedJedis(**shards**);**

jedis**.**set**(**"a"**,** "foo"**);**

jedis**.**disconnect**;**

1. **2.b) Pooled connection:**

ShardedJedisPool pool **=** **new** **ShardedJedisPool(new** **Config(),** shards**);**

ShardedJedis jedis **=** pool**.**getResource**();**

jedis**.**set**(**"a"**,** "foo"**);**

**....** *// do your work here*

pool**.**returnResource**(**jedis**);**

**....** *// a few moments later*

ShardedJedis jedis2 **=** pool**.**getResource**();**

jedis**.**set**(**"z"**,** "bar"**);**

pool**.**returnResource**(**jedis**);**

pool**.**destroy**();**

1. **3. Disconnect / returnRessource**

pool.returnResource should be called as soon as you are finished using jedis in a particular moment. If you don't, the pool may get slower after a while. getResource and returnResource are fast, since no new connection have to be created. Creation and destruction of a pool are slower, since theses are the actual network connections. Forgetting pool.destroy keeps the connection open until timeout is reached.

1. **Determine information of the shard of a particular key**

ShardInfo si **=** jedis**.**getShardInfo**(**key**);**

si**.**getHost**/**getPort**/**getPassword**/**getTimeout**/**getName

1. **Force certain keys to go to the same shard**

What you need is something called "keytags", and they are supported by Jedis. To work with keytags you just need to set a pattern when you instance ShardedJedis. For example:

ShardedJedis jedis **=** **new** **ShardedJedis(**shards**,**

ShardedJedis**.**DEFAULT\_KEY\_TAG\_PATTERN**);**

You can create your own pattern if you want. The default pattern is {}, this means that whatever goes inside curly brackets will be used to determine the shard.

So for example:

jedis**.**set**(**"foo{bar}"**,** "12345"**);**

and

jedis**.**set**(**"car{bar}"**,** "877878"**);**

will go to the same shard.

1. **Mixed approach**

If you want easy load distribution of ShardedJedis, but still need transactions/pipelining/pubsub etc, you can also mix the normal and the sharded approach: define a master as normal Jedis, the others as sharded Jedis. Then make all the shards slaveof master. In your application, direct your write requests to the master, the read requests to ShardedJedis. Your writes don't scale anymore, but you gain good read distribution, and you have transactions/pipelining/pubsub simply using the master. Dataset should fit in RAM of master. Remember that you can improve performance of the master a lot, if you let the slaves do the persistance for the master!

1. **Redis Cluster**

Sometime later 2011, there will be first versions of "redis cluster" which will be a much improved Sharded Jedis and should give back some if not all of the Redis functionalities you cannot have with shardedJedis. If you want to know more about redis cluster, youtube has a presentation of Salvatore Sanfilippo (the creator of Redis).

1. **Monitoring**

To use the monitor command you can do something like the following:

**new** **Thread(new** **Runnable()** **{**

**public** **void** **run()** **{**

Jedis j **=** **new** **Jedis(**"localhost"**);**

**for** **(int** i **=** 0**;** i **<** 100**;** i**++)** **{**

j**.**incr**(**"foobared"**);**

**try** **{**

Thread**.**sleep**(**200**);**

**}** **catch** **(**InterruptedException e**)** **{**

**}**

**}**

j**.**disconnect**();**

**}**

**}).**start**();**

jedis**.**monitor**(new** **JedisMonitor()** **{**

**public** **void** **onCommand(**String command**)** **{**

System**.**out**.**println**(**command**);**

**}**

**});**

1. **Misc**
2. **A note about String and Binary - what is native?**

Redis/Jedis talks a lot about Strings. And here <http://redis.io/topics/internals> it says Strings are the basic building block of Redis. However, this stress on strings may be misleading. Redis' "String" refer to the C char type (8 bit), which is incompatible with Java Strings (16-bit). Redis sees only 8-bit blocks of data of predefined length, so normally it doesn't interpret the data (it's "binary safe"). Therefore in Java, byte[] data is "native", whereas Strings have to be encoded before being sent, and decoded after being retrieved by the SafeEncoder. This has some minor performance impact. In short: if you have binary data, don't encode it into String, but use the binary versions.

1. **A note on Redis' master/slave distribution**

A Redis network consists of redis servers, which can be either masters or slaves. Slaves are synchronized to the master (master/slave replication). However, master and slaves look identical to a client, and slaves do accept write requests, but they will not be propagated "up-hill" and could eventually be overwritten by the master. It makes sense to route reads to slaves, and write demands to the master. Furthermore, being a slave doesn't prevent from being considered master by another slave.