Intro to Jedis – the Java Redis Client Library

Last modified: December 2, 2018

by baeldung (/author/baeldung/)

Persistence (/category/persistence/)

Redis (/tag/redis/)

1. Overview

This article is **an introduction to Jedis**, a client library in Java for Redis (http://redis.io/) – the popular in-memory data structure store that can persist on disk as well. It is driven by a keystore-based data structure to persist data and can be used as a database, cache, message broker, etc.

First, we are going to explain in which kind of situations Jedis is useful and what it is about.

In the subsequent sections we are elaborating on the various data structures and explaining transactions, pipelining and the publish/subscribe feature. We conclude with connection pooling and Redis Cluster.

2. Why Jedis?

Redis lists the most well-known client libraries on their official site (http://redis.io/clients#java/). There are multiple alternatives to Jedis, but only two more are currently worthy of their recommendation star, lettuce (http://redis.paluch.biz/), and Redisson (https://github.com/mrniko/redisson/).

These two clients do have some unique features like thread safety, transparent reconnection handling and an asynchronous API, all features of which Jedis lacks.

However, it is small and considerably faster than the other two. Besides, it is the client library of choice of the Spring Framework developers, and it has the biggest community of all three.

3. Maven Dependencies

Let's start by declaring the only dependency we will need in the *pom.xml*:

```xml
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.8.1</version>
</dependency>
```

If you're looking for the latest version of the library, check out this page (https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22redis.clients%22%20AND%20a%3A%22jedis%22).

4. Redis Installation

You will need to install and fire up one of the latest versions of Redis. We are running the latest stable version at this moment (3.2.1), but any post 3.x version should be okay.

Find here (http://redis.io/topics/quickstart) more information about Redis for Linux and Macintosh, they have very similar basic installation steps. Windows is not officially supported, but this port (https://github.com/MSOpenTech/redis) is well maintained.

After that we can directly dive in and connect to it from our Java code:

```
Jedis jedis = new Jedis();
```

The default constructor will work just fine unless you have started the service on a non-default port or a remote machine, in which case you can configure it correctly by passing the correct values as parameters into the constructor.

5. Redis Data Structures

Most of the native operation commands are supported and, conveniently enough, they normally share the same method name.

5.1. Strings

Strings are the most basic kind of Redis value, useful for when you need to persist simple key-value data types:

```
jedis.set("events/city/rome", "32,15,223,828");
String cachedResponse = jedis.get("events/city/rome");
```

The variable *cachedResponse* will hold the value *32,15,223,828*. Coupled with expiration support, discussed later, it can work as a lightning fast and simple to use cache layer for HTTP requests received at your web application and other caching requirements.

5.2. Lists

Redis Lists are simply lists of strings, sorted by insertion order and make it an ideal tool to implement, for instance, message queues:

```
jedis.lpush("queue#tasks", "firstTask");
jedis.lpush("queue#tasks", "secondTask");

String task = jedis.rpop("queue#tasks");
```

The variable *task* will hold the value *firstTask*. Remember that you can serialize any object and persist it as a string, so messages in the queue can carry more complex data when required.

5.3. Sets

Redis Sets are an unordered collection of Strings that come in handy when you want to exclude repeated members:

```java
jedis.sadd("nicknames", "nickname#1");
jedis.sadd("nicknames", "nickname#2");
jedis.sadd("nicknames", "nickname#1");

Set<String> nicknames = jedis.smembers("nicknames");
boolean exists = jedis.sismember("nicknames", "nickname#1");
```

The Java Set *nicknames* will have a size of 2, the second addition of *nickname#1* was ignored. Also, the *exists* variable will have a value of *true*, the method *sismember* enables you to check for the existence of a particular member quickly.

5.4. Hashes

Redis Hashes are mapping between *String* fields and *String* values:

```java
jedis.hset("user#1", "name", "Peter");
jedis.hset("user#1", "job", "politician");

String name = jedis.hget("user#1", "name");

Map<String, String> fields = jedis.hgetAll("user#1");
String job = fields.get("job");
```

As you can see, hashes are a very convenient data type when you want to access object's properties individually since you do not need to retrieve the whole object.

5.5. Sorted Sets

Sorted Sets are like a Set where each member has an associated ranking, that is used for sorting them:

```java
Map<String, Double> scores = new HashMap<>();

scores.put("PlayerOne", 3000.0);
scores.put("PlayerTwo", 1500.0);
scores.put("PlayerThree", 8200.0);

scores.entrySet().forEach(playerScore -> {
    jedis.zadd(key, playerScore.getValue(), playerScore.getKey());
});

String player = jedis.zrevrange("ranking", 0, 1).iterator().next();
long rank = jedis.zrevrank("ranking", "PlayerOne");
```

The variable *player* will hold the value *PlayerThree* because we are retrieving the top 1 player and he is the one with the highest score. The *rank* variable will have a value of 1 because *PlayerOne* is the second in the ranking and the ranking is zero-based.

6. Transactions

Transactions guarantee atomicity and thread safety operations, which means that requests from other clients will never be handled concurrently during Redis transactions:

```java
String friendsPrefix = "friends#";
String userOneId = "4352523";
String userTwoId = "5552321";

Transaction t = jedis.multi();
t.sadd(friendsPrefix + userOneId, userTwoId);
t.sadd(friendsPrefix + userTwoId, userOneId);
t.exec();
```

You can even make a transaction success dependent on a specific key by "watching" it right before you instantiate your *Transaction*:

```java
jedis.watch("friends#deleted#" + userOneId);
```

If the value of that key changes before the transaction is executed, the transaction will not be completed successfully.

7. Pipelining

When we have to send multiple commands, we can pack them together in one request and save connection overhead by using pipelines, it is essentially a network optimization. As long as the operations are mutually independent, we can take advantage of this technique:

```java
String userOneId = "4352523";
String userTwoId = "4849888";

Pipeline p = jedis.pipelined();
p.sadd("searched#" + userOneId, "paris");
p.zadd("ranking", 126, userOneId);
p.zadd("ranking", 325, userTwoId);
Response<Boolean> pipeExists = p.sismember("searched#" + userOneId, "paris");
Response<Set<String>> pipeRanking = p.zrange("ranking", 0, -1);
p.sync();

String exists = pipeExists.get();
Set<String> ranking = pipeRanking.get();
```

Notice we do not get direct access to the command responses, instead, we're given a *Response* instance from which we can request the underlying response after the pipeline has been synced.

8. Publish/Subscribe

We can use the Redis messaging broker functionality to send messages between the different components of our system. Make sure the subscriber and publisher threads do not share the same Jedis connection.

8.1. Subscriber

Subscribe and listen to messages sent to a channel:

```java
Jedis jSubscriber = new Jedis();
jSubscriber.subscribe(new JedisPubSub() {
    @Override
    public void onMessage(String channel, String message) {
        // handle message
    }
}, "channel");
```

Subscribe is a blocking method, you will need to unsubscribe from the *JedisPubSub* explicitly. We have overridden the *onMessage* method but there are many more useful methods (http://javadox.com/redis.clients/jedis/2.8.0/redis/clients/jedis/JedisPubSub.html) available to override.

8.2. Publisher

Then simply send messages to that same channel from the publisher's thread:

```java
Jedis jPublisher = new Jedis();
jPublisher.publish("channel", "test message");
```

9. Connection Pooling

It is important to know that the way we have been dealing with our Jedis instance is naive. In a real-world scenario, you do not want to use a single instance in a multi-threaded environment as a single instance is not thread-safe.

Luckily enough we can easily create a pool of connections to Redis for us to reuse on demand, a pool that is thread safe and reliable as long as you return the resource to the pool when you are done with it.

Let's create the *JedisPool*:

```
final JedisPoolConfig poolConfig = buildPoolConfig();
JedisPool jedisPool = new JedisPool(poolConfig, "localhost");

private JedisPoolConfig buildPoolConfig() {
    final JedisPoolConfig poolConfig = new JedisPoolConfig();
    poolConfig.setMaxTotal(128);
    poolConfig.setMaxIdle(128);
    poolConfig.setMinIdle(16);
    poolConfig.setTestOnBorrow(true);
    poolConfig.setTestOnReturn(true);
    poolConfig.setTestWhileIdle(true);
    poolConfig.setMinEvictableIdleTimeMillis(Duration.ofSeconds(60).toMillis());
    poolConfig.setTimeBetweenEvictionRunsMillis(Duration.ofSeconds(30).toMillis());
    poolConfig.setNumTestsPerEvictionRun(3);
    poolConfig.setBlockWhenExhausted(true);
    return poolConfig;
}
```

Since the pool instance is thread safe, you can store it somewhere statically but you should take care of destroying the pool to avoid leaks when the application is being shutdown.

Now we can make use of our pool from anywhere in the application when needed:

```
try (Jedis jedis = jedisPool.getResource()) {
    // do operations with jedis resource
}
```

We used the Java try-with-resources statement to avoid having to manually close the Jedis resource, but if you cannot use this statement you can also close the resource manually in the *finally* clause.

Make sure you use a pool like we have described in your application if you do not want to face nasty multi-threading issues. You can obviously play with the pool configuration parameters to adapt it to the best setup in your system.

10. Redis Cluster

This Redis implementation provides easy scalability and high availability, we encourage you to read their official specification (http://redis.io/topics/cluster-spec) if you are not familiar with it. We will not cover Redis cluster setup since that is a bit out of the scope for this article, but you should have no problems in doing so when you are done with its documentation.

Once we have that ready, we can start using it from our application:

```
try (JedisCluster jedisCluster = new JedisCluster(new HostAndPort("localhost", 6379))) {
    // use the jedisCluster resource as if it was a normal Jedis resource
} catch (IOException e) {}
```

We only need to provide the host and port details from one of our master instances, it will auto-discover the rest of the instances in the cluster.

This is certainly a very powerful feature but it is not a silver bullet. When using Redis Cluster you cannot perform transactions nor use pipelines, two important features on which many applications rely for ensuring data integrity.

Transactions are disabled because, in a clustered environment, keys will be persisted across multiple instances. Operation atomicity and thread safety cannot be guaranteed for operations that involve command execution in different instances.

Some advanced key creation strategies will ensure that data that is interesting for you to be persisted in the same instance will get persisted that way. In theory, that should enable you to perform transactions successfully using one of the underlying Jedis instances of the Redis Cluster.

Unfortunately, currently you cannot find out in which Redis instance a particular key is saved using Jedis (which is actually supported natively by Redis), so you do not know which of the instances you must perform the transaction operation. If you are interested about this, you can find more information here (https://groups.google.com/forum/#!topic/redis-db/4I0ELYnf3bk).

11. Conclusion

The vast majority of the features from Redis are already available in Jedis and its development moves forward at a good pace.

It gives you the ability to integrate a powerful in-memory storage engine in your application with very little hassle, just do not forget to set up connection pooling to avoid thread safety issues.

You can find code samples in the GitHub project (https://github.com/eugenp/tutorials/tree/master/persistence-modules/redis).