jedis 使用指南

概述

本文基于 jedis-2.1.0 和 commons-pool-1.5.5。

本文首先会剖析 jedis 对 redis 的支持，透过 jedis 群集之后，那些 redis 的能力会受到限制，因此第一个部分会重点介绍 jedis 的关键特性。

对象池的配置会决定 jedis 的最终性能，而 jedis 的对象池的实现是基于 apache 的 commons-pool。本文会重点介绍 commons-pool 的配置能力，以便为 jedis 群集提供更好的配置参数。

另外，jedis 还有一些扩展包，可以在特定场景下增强 jedis 的能力，使我们面对特定的应用场景的时候，会有更好的选择。所以，文档的最后会涉及如何使用 jedis 的扩展包。

jedis 的关键特性

主要是 API 支持范围，pipeline 和 transaction 的使用，以及对 pub/sub 的支持。最后，介绍 jedis 的客户端群集：ShardedJedis。

完整的 redis API 支持

jedis 首先是 redis 的 java 客户端，jedis 提供了完整的 redis 的 java API，可以支持完整的 redis 客户端 API。

例如，通过 jedis，可以设置 redis 的主从关系：

```
jedis.slaveOf("localhost", 6379);  //  if the master is on the same PC which runs your code
jedis.slaveOf("192.168.1.35", 6379);
```

使用 pipeline

如果希望一次发送一批 redis 命令，一种有效的方式是使用 pipeline。

jedis 使用 pipeline 的代码如下：

```
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();
```

使用 transaction

如果希望一些命令一起执行而不被干扰，可以通过 transaction 将命令打包到一起执行：

```
jedis.watch (key1, key2, ...);
BinaryTransaction t = jedis.multi();
t.set("foo", "bar");
t.exec();
```

如果需要得到返回值，可以参考下面的代码：

```
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!
```

Publish/Subscribe

如果需要订阅 redis 的 channel，可以创建一个 JedisPubSub 的派生类的实例，并在 jedis 上调用其 subscribe 方法：

```
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");
```

subscribe 是一个阻塞的操作。一个 JedisPubSub 的实例可以订阅多个 redis 的 channel。

ShardedJedis

简单的说，ShardedJedis 是一种帮助提高读/写并发能力的群集，群集使用一致性 hash 来确保一个 key 始终被指向相同的 redis server。每个 redis server 被称为一个 shard。

因为每个 shard 都是一个 master，因此使用 sharding 机制会产生一些限制：不能在 sharding 中直接使用 jedis 的 transactions、pipelining、pub/sub 这些 API，基本的原则是不能跨越 shard。但 jedis 并没有在 API 的层面上禁止这些行为，但是这些行为会有不确定的结果。一种可能的方式是使用 keytags 来干预 key 的分布，当然，这需要手工的干预。

另外一个限制是正在使用的 shards 是不能被改变的，因为所有的 sharding 都是预分片的。

注：
如果希望使用可以改变的 shards， 可以使用 yaourt - dynamic sharding implementation（一个 jedis 的实现分支）。

ShardedJedis 的使用方法：

1. 定义 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);
```

2.a) 直接使用:

```
ShardedJedis jedis = new ShardedJedis(shards);
jedis.set("a", "foo");
jedis.disconnect;
```

2.b) 或 使用连接池:

```
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();
```

判断使用的是那个 shards：

```
ShardInfo si = jedis.getShardInfo(key);
si.getHost/getPort/getPassword/getTimeout/getName
```

也可以通过 keytags 来确保 key 位于相同的 shard。如：

```
ShardedJedis jedis = new ShardedJedis(shards,
ShardedJedis.DEFAULT_KEY_TAG_PATTERN);
```

这样，默认的 keytags 是”{}”，这表示在”{}”内的字符会用于决定使用那个 shard。
如：

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

和

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

会使用同一个 shard。

注：
如果 key 和 keytag 不匹配，会使用原来的 key 作为选择 shard 的 key。

使用 ShardedJedisPipeline

ShardedJedisPipeline 其实是一个很鸡肋的功能。

为了能在 ShardedJedis 中平滑的支持 redis 的 pipeline 的功能，ShardedJedis 通过 ShardedJedisPipeline 类对 pipeline 提供了支持。

简单的说，ShardedJedis 是通过向每个用到的 shard 发起 pipeline 来实现 ShardedJedisPipeline 的功能，这种方式如果累积的 key 不够多，很难达到提高效率的目的。

如果需要在 ShardedJedis 中使用 pipeline，还是建议尽量通过 keytag 将关联的 key 放到同一 shard 之中。

ShardedJedisPipeline 简单的示例代码如下：

```
ShardedJedis jedis = new ShardedJedis(shards);
ShardedJedisPipeline p = jedis.pipelined();
p.set("foo", "bar");
p.get("foo");
List<Object> results = p.syncAndReturnAll();

//assertEquals(2, results.size());
//assertEquals("OK", results.get(0));
//assertEquals("bar", results.get(1));
```

ShardedJedisPipeline 相对复杂的示例代码：

```
ShardedJedis jedis = new ShardedJedis(shards);
jedis.set("string", "foo");
jedis.lpush("list", "foo");
jedis.hset("hash", "foo", "bar");
jedis.zadd("zset", 1, "foo");
```

```
jedis.sadd("set", "foo");
ShardedJedisPipeline p = jedis.pipelined();
Response<String> string = p.get("string");
Response<Long> del = p.del("string");
Response<String> emptyString = p.get("string");
Response<String> list = p.lpop("list");
Response<String> hash = p.hget("hash", "foo");
Response<Set<String>> zset = p.zrange("zset", 0, -1);
Response<String> set = p.spop("set");
Response<Boolean> blist = p.exists("list");
Response<Double> zincrby = p.zincrby("zset", 1, "foo");
Response<Long> zcard = p.zcard("zset");
p.lpush("list", "bar");
Response<List<String>> lrange = p.lrange("list", 0, -1);
Response<Map<String, String>> hgetAll = p.hgetAll("hash");
p.sadd("set", "foo");
Response<Set<String>> smembers = p.smembers("set");
Response<Set<Tuple>> zrangeWithScores = p.zrangeWithScores("zset", 0,
        -1);
p.sync();

assertEquals("foo", string.get());
assertEquals(Long.valueOf(1), del.get());
assertNull(emptyString.get());
assertEquals("foo", list.get());
assertEquals("bar", hash.get());
assertEquals("foo", zset.get().iterator().next());
assertEquals("foo", set.get());
assertFalse(blist.get());
assertEquals(Double.valueOf(2), zincrby.get());
assertEquals(Long.valueOf(1), zcard.get());
assertEquals(1, lrange.get().size());
assertNotNull(hgetAll.get().get("foo"));
assertEquals(1, smembers.get().size());
assertEquals(1, zrangeWithScores.get().size());
```

使用 jedis 的对象池

jedis 通过 commons-pool 来提供其对象池的功能，其对象池类有 JedisPool 和 ShardedJedisPool，面向普通的 redis 连接池和 pre-sharding 的 redis 连接池。

在连接池的使用和配置层面，这两个类基本没什么差别。

配置 jedis 的连接池，一般通过 JedisPoolConfig 类完成，其提供了一个不同于基类的默认值，当然也可以通过 org.apache.commons.pool.impl.GenericObjectPool.Config 类来配置，这个类的默认值我们可以在 commons-pool 对象池配置的小节中看到。

对象池的使用

jedis 创建对象池的方式：

```
JedisPool pool = new JedisPool(new JedisPoolConfig(), "localhost");
```

使用池中的对象，是通过 JedisPool 的 getResource 和 returnResource 来得到和归还资源：

```
Jedis jedis = pool.getResource();
try {
   /// ... do stuff here ... for example
   jedis.set("foo", "bar");
   String foobar = jedis.get("foo");
   jedis.zadd("sose", 0, "car"); jedis.zadd("sose", 0, "bike");
   Set<String> sose = jedis.zrange("sose", 0, -1);
} catch (JedisConnectionException e) {
       // returnBrokenResource when the state of the object is unrecoverable
       if (null != jedis) {
           pool.returnBrokenResource(jedis);
           jedis = null;
       }
} finally {
   /// ... it's important to return the Jedis instance to the pool once you've finished using it
   if (null != jedis)
    pool.returnResource(jedis);
}
/// ... when closing your application:
pool.destroy();
```

commons-pool 对象池配置

jedis 的对象池是通过 apache 的 commons-pool 实现的。其对象池的配置是通过 **org.apache.commons.pool.impl.GenericObjectPool.Config 类**完成。

Config 是一个简单的值对象类，其成员都有预设的默认值。

我们将 Config 类的各个成员的配置含义描述如下：

■ maxActive

控制池中对象的最大数量。
默认值是 8，如果是负值表示没限制。

■ maxIdle

控制池中空闲的对象的最大数量。
默认值是 8，如果是负值表示没限制。

■ minIdle

控制池中空闲的对象的最小数量。
默认值是 0。

■ whenExhaustedAction

指定池中对象被消耗完以后的行为，有下面这些选择：
>> WHEN_EXHAUSTED_FAIL 0
>> WHEN_EXHAUSTED_GROW 2
>> WHEN_EXHAUSTED_BLOCK 1

如果是 WHEN_EXHAUSTED_FAIL，当池中对象达到上限以后，继续 borrowObject 会抛出 NoSuchElementException 异常。
如果是 WHEN_EXHAUSTED_GROW，当池中对象达到上限以后，会创建一个新对象，并返回它。
如果是 WHEN_EXHAUSTED_BLOCK，当池中对象达到上限以后，会一直等待，直到有一个对象可用。这个行为还与 maxWait 有关，如果 maxWait 是正数，那么会等待 maxWait 的毫秒的时间，超时会抛出 NoSuchElementException 异常；如果 maxWait 为负值，会永久等待。

whenExhaustedAction 的默认值是 WHEN_EXHAUSTED_BLOCK，maxWait 的默认值是-1。

■ maxWait

whenExhaustedAction 如果是 WHEN_EXHAUSTED_BLOCK，指定等待的毫秒数。如果 maxWait 是正数，那么会等待 maxWait 的毫秒的时间，超时会抛出 NoSuchElementException 异常；如果 maxWait 为负值，会永久等待。
maxWait 的默认值是-1。

■ testOnBorrow

如果 testOnBorrow 被设置，pool 会在 borrowObject 返回对象之前使用 PoolableObjectFactory 的 validateObject 来验证这个对象是否有效，要是对象没通过验证，这个对象会被丢弃，然

后重新选择一个新的对象。

testOnBorrow 的默认值是 false。

■ testOnReturn

如果 testOnReturn 被设置，pool 会在 returnObject 的时候通过 PoolableObjectFactory 的 validateObject 方法验证对象，如果对象没通过验证，对象会被丢弃，不会被放到池中。

testOnReturn 的默认值是 false。

■ testWhileIdle

指定 idle 对象是否应该使用 PoolableObjectFactory 的 validateObject 校验，如果校验失败，这个对象会从对象池中被清除。
这个设置仅在 timeBetweenEvictionRunsMillis 被设置成正值（>0）的时候才会生效。
testWhileIdle 的默认值是 false。

■ timeBetweenEvictionRunsMillis

指定驱逐线程的休眠时间。如果这个值不是正数（>0），不会有驱逐线程运行。
timeBetweenEvictionRunsMillis 的默认值是-1。

■ numTestsPerEvictionRun

设置驱逐线程每次检测对象的数量。
这个设置仅在 timeBetweenEvictionRunsMillis 被设置成正值（>0）的时候才会生效。
numTestsPerEvictionRun 的默认值是 3。

■ minEvictableIdleTimeMillis

指定最小的空闲驱逐的时间间隔（空闲超过指定的时间的对象，会被清除掉）。
这个设置仅在 timeBetweenEvictionRunsMillis 被设置成正值（>0）的时候才会生效。
minEvictableIdleTimeMillis 默认值是 30 分钟。

■ softMinEvictableIdleTimeMillis

与 minEvictableIdleTimeMillis 类似，也是指定最小的空闲驱逐的时间间隔（空闲超过指定的时间的对象，会被清除掉），不过会参考 minIdle 的值，只有 idle 对象的数量超过 minIdle 的值，对象才会被清除。
这个设置仅在 timeBetweenEvictionRunsMillis 被设置成正值（>0）的时候才会生效，并且这个配置能被 minEvictableIdleTimeMillis 配置取代（minEvictableIdleTimeMillis 配置项的优先级更高）。

softMinEvictableIdleTimeMillis 的默认值是-1。

■ lifo

pool 可以被配置成 LIFO 队列（last-in-first-out）或 FIFO 队列（first-in-first-out），来指定空闲对象被使用的次序。
lifo 的默认值是 true。

JedisPoolConfig 的调整

jedis 的对象池是通过 commons-pool 实现的，对对象池的配置应该通过 JedisPoolConfig 来完成，jedis 提供了自己的配置参数：

```
public class JedisPoolConfig extends Config {
    public JedisPoolConfig() {
        // defaults to make your life with connection pool easier :)
        setTestWhileIdle(true);
        setMinEvictableIdleTimeMillis(60000);
        setTimeBetweenEvictionRunsMillis(30000);
        setNumTestsPerEvictionRun(-1);
    }
```

简单的说，是启用了 commons-pool 的驱逐线程，并配置了驱逐线程的轮询参数。