# Jedis

使用阻塞的I/O，且其方法调用都是同步的，程序流需要等到sockets处理完I/O才能执行，不支持异步。Jedis客户端实例不是线程安全的，所以需要通过连接池来使用Jedis。

## 解析版本

1. <dependency>
2. <groupId>redis.clients</groupId>
3. <artifactId>jedis</artifactId>
4. <version>2.9.0</version>
5. </dependency>

## [常用方法](http://www.redis.cn/commands)

|  |  |
| --- | --- |
| 操作 | 具体函数 |
| 初始化链接 | new Jedis(ip,port) |
| 清空数据 | Jedis.flushDB() |
| 新增键值为(key,value) | Jedis.set(String key,String value) |
| 获取所有key |  |

## 关键源码解析

### 初始化

#### 工作模式

Jedis对应Redis的四种工作模式初始化类，对应关系如下：

| **Jedis主要模块** | **Redis工作模式** |
| --- | --- |
| Jedis | Redis Standalone（单节点模式） |
| JedisCluster | Redis Cluster（集群模式） |
| JedisSentinel | Redis Sentinel（哨兵模式） |
| ShardedJedis | Redis Sharding（分片模式） |

#### 三种请求模式

Jedis实例有3种请求模式：Client、Pipeline和Transaction；Jedis实例通过Socket建立客户端与服务端的长连接，往outputStream发送命令，从inputStream读取回复：

[Jredis 请求模式图](Jedis%2001.png)

* **Client模式**：客户端发一个命令，阻塞等待服务端执行，然后读取返回结果；
* **Pipeline模式**：一次性发送多个命令，最后一次取回所有的返回结果，这种模式通过减少网络的往返时间和IO的读写次数，大幅度提高通信性能；
  + 原生批命令（mset、mget）与Pipeline对比：
    - 原生批量命令是原子性，Pipeline是非原子性的
    - 原生批量命令是一个命令对应多个key，Pipeline支持多个命令
    - 原生批量命令是Redis服务端支持实现的，而Pipeline需要服务端与客户端的共同实现
* **Transaction模式**：Transaction模式即开启Redis的事务管理，Redis事务可以一次执行多个命令， 并且带有以下三个重要的保证：
  + 批量操作在发送EXEC命令前被放入队列缓存
  + 收到EXEC命令后进入事务执行，事务中任意命令执行失败，其余的命令依然被执行
  + 在事务执行过程，其他客户端提交的命令请求不会插入到事务执行命令序列中
  + 一个事务从开始到执行会经历以下三个阶段：
    - 开始事务
    - 命令入队
    - 执行事务

[Jedis类结构](Jedis%2002.png)

#### 初始化流程

1. Jedis jedis = new Jedis("47.105.133.123", 6379);

Jedis通过传入Redis服务器地址（host、port）开始初始化，然后在BinaryJedis里实例化Client。Client通过Socket维持客户端与Redis服务器的连接。Transaction和Pipeline继承同一个基类MultiKeyPipelineBase。Transaction在实例化的时候，就自动发送MULTI命令，开启事务模式；Pipeline按情况手动开启。它们均依靠Client发送命令。

##### Client模式

1. public String get(final String key) {
2. *// 校验是否是transaction模式或者pipeline模式*
3. checkIsInMultiOrPipeline();
4. *// 1)调用client发送命令*
5. client.get(key);
6. *// 2)从inputStream里读取回复*
7. return client.getBulkReply();
8. }

代码1)处会调用Connection的sendCommand()方法，再调用Protocol的sendCommand()方法按照Redis的同一请求协议组织Redis命令，写入outputStream：

1. public void sendCommand(final ProtocolCommand cmd, final byte[]... args) {
2. try {
3. *// 连接socket,如果已连接,跳过*
4. connect();
5. *// 按照redis的同一请求协议组织redis命令,写入outputStream*
6. Protocol.sendCommand(outputStream, cmd, args);
7. pipelinedCommands++;
8. return this
9. } catch (JedisConnectionException ex) {
10. */\**
11. \* When client send request which formed by invalid protocol, Redis send back error message
12. \* before close connection. We try to read it to provide reason of failure.
13. \*/
14. try {
15. String errorMessage = Protocol.readErrorLineIfPossible(inputStream);
16. if (errorMessage != null && errorMessage.length() > 0) {
17. ex = new JedisConnectionException(errorMessage, ex.getCause());
18. }
19. } catch (Exception e) {
20. */\**
21. \* Catch any IOException or JedisConnectionException occurred from InputStream#read and just
22. \* ignore. This approach is safe because reading error message is optional and connection
23. \* will eventually be closed.
24. \*/
25. }
26. *// Any other exceptions related to connection?*
27. broken = true;
28. throw ex;
29. }
30. ｝

代码2)处调用Connection的getBulkReply()方法，再调用Protocol的read()方法从inputStream中读取服务器的回复，此处会阻塞等待

1. public String getBulkReply() {
2. flush();
3. pipelinedCommands--;
4. final byte[] resp = (byte[]) readProtocolWithCheckingBroken();
5. if (null == resp) {
6. return null;
7. } else {
8. return SafeEncoder.encode(resp);
9. }
10. }
11. protected Object readProtocolWithCheckingBroken() {
12. try {
13. *// 从inputStream中读取服务器的回复,此处阻塞等待*
14. return Protocol.read(inputStream);
15. } catch (JedisConnectionException exc) {
16. broken = true;
17. throw exc;
18. }
19. }

[Client模式调用链](Jedis%2003.png)

##### Pipeline模式和Transaction模式

* Pipeline和Transaction都继承自MultiKeyPipelineBase
* MultiKeyPipelineBase和PipelineBase的区别在于处理的命令不同，内部均调用Client发送命令
* Pipeline有一个内部类对象MultiResponseBuilder，当调用Pipeline的sync()之前，存储所有返回结果

Pipeline的使用方法：

1. Pipeline pipeline = jedis.pipelined();
2. Response<String> key1 = pipeline.get("key1");
3. Response<String> key2 = pipeline.get("key2");
4. Response<String> key3 = pipeline.get("key3");
5. pipeline.sync();
6. System.out.println("value1:" + key1.get() + ",value2:" + key2.get() + ",value3:" + key3.get());

Transaction的使用方法：

1. Transaction transaction = jedis.multi();
2. Response<String> key1 = transaction.get("key1");
3. Response<String> key2 = transaction.get("key2");
4. Response<String> key3 = transaction.get("key3");
5. transaction.exec();
6. System.out.println("value1:" + key1.get() + ",value2:" + key2.get() + ",value3:" + key3.get());

以get(key)为例，Pipeline代码如下：

1. **public** Response<String> get(**final** String key) {
2. // 调用client发送命令
3. getClient(key).get(key);
4. // 新建response,放入消息队列queue,此时response没有数据
5. **return** getResponse(BuilderFactory.STRING);
6. }
7. **protected** <T> Response<T> getResponse(Builder<T> builder) {
8. Response<T> lr = **new** Response<T>(builder);
9. pipelinedResponses.add(lr);
10. **return** lr;
11. }

Pipeline的sync()方法代码如下：

1. **public** **void** sync() {
2. // 判断消息队列是否为空,是否发出请求
3. **if** (getPipelinedResponseLength() > 0) {
4. // 1)从inputStream中获取回复消息,消息塞入消息队列的response中
5. List<Object> unformatted = client.getAll();
6. **for** (Object o : unformatted) {
7. generateResponse(o);
8. }
9. }

代码1)会调用Connection的getMany()方法，代码如下：

1. **public** List<Object> getAll(**int** except) {
2. List<Object> all = **new** ArrayList<Object>();
3. flush();
4. **while** (pipelinedCommands > except) {
5. **try** {
6. all.add(readProtocolWithCheckingBroken());
7. } **catch** (JedisDataException e) {
8. all.add(e);
9. }
10. pipelinedCommands--;
11. }
12. **return** all;
13. }
15. **protected** Object readProtocolWithCheckingBroken() {
16. **try** {
17. // 从inputStream中读取服务器的回复,此处阻塞等待
18. **return** Protocol.read(inputStream);
19. } **catch** (JedisConnectionException exc) {
20. broken = **true**;
21. **throw** exc;
22. }
23. }

Pipeline的sync()方法中调用Queable的方法如下：

1. **protected** Response<?> generateResponse(Object data) {
2. Response<?> response = pipelinedResponses.poll();
3. **if** (response != **null**) {
4. response.set(data);
5. }
6. **return** response;
7. }

[Pipeline调用链](Jedis%2004.png)

Transaction的exec()方法和Pipeline的sync()很相似，代码如下：

1. **public** List<Object> exec() {
2. // 发送EXEC指令,让服务端执行所有命令
3. client.exec();
4. // 放弃除最后一个答复以外的所有答复
5. client.getAll(1);
6. // 事务结束
7. inTransaction = **false**;
9. // 从inputStream中读取所有回复
10. List<Object> unformatted = client.getObjectMultiBulkReply();
11. **if** (unformatted == **null**) {
12. **return** Collections.emptyList();
13. }
14. // 处理响应结果
15. List<Object> formatted = **new** ArrayList<Object>();
16. **for** (Object o : unformatted) {
17. **try** {
18. formatted.add(generateResponse(o).get());
19. } **catch** (JedisDataException e) {
20. formatted.add(e);
21. }
22. }
23. **return** formatted;
24. }

### Protocol解析

Protocol是一个通讯工具类，将Redis的各类执行关键字存储为静态变量，可以直观调用命令，例如Protocol.Command.GET。同时，将命令包装成符合Redis的统一请求协议，回复消息的处理也是在这个类进行，先通过通讯协提取出当次请求的回复消息，将Object类型的消息，格式化为String、List等具体类型，如果回复消息有Error则以异常的形式抛出。

#### 命令的发送方法

1. **private** **static** **void** sendCommand(**final** RedisOutputStream os, **final** **byte**[] command, **final** **byte**[]... args) {
2. **try** {
3. os.write(ASTERISK\_BYTE);
4. os.writeIntCrLf(args.length + 1);
5. os.write(DOLLAR\_BYTE);
6. os.writeIntCrLf(command.length);
7. // 命令枚举，可返回字节数组
8. os.write(command);
9. os.writeCrLf();
11. **for** (**final** **byte**[] arg : args) {
12. os.write(DOLLAR\_BYTE);
13. os.writeIntCrLf(arg.length);
14. os.write(arg);
15. os.writeCrLf();
16. }
17. } **catch** (IOException e) {
18. **throw** **new** JedisConnectionException(e);
19. }
20. }

可以看出redis的[命令格式](#_RedisOutputStream)：

* [\*号][消息元素个数]\r\n (消息元素个数 = 参数个数 + 1个命令)
* [$号][命令字节个数]\r\n
* [命令内容]\r\n
* [$号][参数字节个数]\r\n
* [参数内容]\r\n
* [$号][参数字节个数]\r\n
* [参数内容]\r\n

#### 接收解析方法

1. **public** **static** Object read(**final** RedisInputStream is) {
2. **return** process(is);
3. }
5. **private** **static** Object process(**final** RedisInputStream is) {
7. **final** **byte** b = is.readByte();
8. **if** (b == PLUS\_BYTE) {
9. **return** processStatusCodeReply(is);
10. } **else** **if** (b == DOLLAR\_BYTE) {
11. **return** processBulkReply(is);
12. } **else** **if** (b == ASTERISK\_BYTE) {
13. **return** processMultiBulkReply(is);
14. } **else** **if** (b == COLON\_BYTE) {
15. **return** processInteger(is);
16. } **else** **if** (b == MINUS\_BYTE) {
17. processError(is);
18. **return** **null**;
19. } **else** {
20. **throw** **new** JedisConnectionException("Unknown reply: " + (**char**) b);
21. }
22. }
24. **private** **static** **byte**[] processStatusCodeReply(**final** RedisInputStream is) {
25. **return** is.readLineBytes();
26. }
28. **private** **static** **byte**[] processBulkReply(**final** RedisInputStream is) {
29. **final** **int** len = is.readIntCrLf();
30. **if** (len == -1) {
31. **return** **null**;
32. }
34. **final** **byte**[] read = **new** **byte**[len];
35. **int** offset = 0;
36. **while** (offset < len) {
37. **final** **int** size = is.read(read, offset, (len - offset));
38. **if** (size == -1) **throw** **new** JedisConnectionException(
39. "It seems like server has closed the connection.");
40. offset += size;
41. }
43. // read 2 more bytes for the command delimiter
44. is.readByte();
45. is.readByte();
47. **return** read;
48. }
50. **private** **static** List<Object> processMultiBulkReply(**final** RedisInputStream is) {
51. **final** **int** num = is.readIntCrLf();
52. **if** (num == -1) {
53. **return** **null**;
54. }
55. **final** List<Object> ret = **new** ArrayList<Object>(num);
56. **for** (**int** i = 0; i < num; i++) {
57. **try** {
58. ret.add(process(is));
59. } **catch** (JedisDataException e) {
60. ret.add(e);
61. }
62. }
63. **return** ret;
64. }
66. **private** **static** Long processInteger(**final** RedisInputStream is) {
67. **return** is.readLongCrLf();
68. }
70. **private** **static** **void** processError(**final** RedisInputStream is) {
71. String message = is.readLine();
72. // TODO: I'm not sure if this is the best way to do this.
73. // Maybe Read only first 5 bytes instead?
74. **if** (message.startsWith(MOVED\_RESPONSE)) {
75. String[] movedInfo = parseTargetHostAndSlot(message);
76. **throw** **new** JedisMovedDataException(message, **new** HostAndPort(movedInfo[1],
77. Integer.valueOf(movedInfo[2])), Integer.valueOf(movedInfo[0]));
78. } **else** **if** (message.startsWith(ASK\_RESPONSE)) {
79. String[] askInfo = parseTargetHostAndSlot(message);
80. **throw** **new** JedisAskDataException(message, **new** HostAndPort(askInfo[1],
81. Integer.valueOf(askInfo[2])), Integer.valueOf(askInfo[0]));
82. } **else** **if** (message.startsWith(CLUSTERDOWN\_RESPONSE)) {
83. **throw** **new** JedisClusterException(message);
84. } **else** **if** (message.startsWith(BUSY\_RESPONSE)) {
85. **throw** **new** JedisBusyException(message);
86. } **else** **if** (message.startsWith(NOSCRIPT\_RESPONSE) ) {
87. **throw** **new** JedisNoScriptException(message);
88. }
89. **throw** **new** JedisDataException(message);
90. }

通过返回数据的第一个字节来判断返回的数据类型，调用不同的[处理函数](#_RedisInputStream)：

[-号] 错误信息

[\*号] 多个数据 结构和发送命令的结构一样

[:号] 一个整数

[$号] 一个数据 结构和发送命令的结构一样

[+号] 一个状态码

### Jedis流的实现

#### RedisOutputStream

该类实现了一个非线程安全的缓冲输出流，还有一些特殊操作，如字符串编码。此流完全忽略标记/重置，不应在jedis之外使用。

写入redis命令方法：

1. **public** **void** write(**final** **byte** b) **throws** IOException {
2. **if** (count == buf.length) {
3. flushBuffer();
4. }
5. buf[count++] = b;
6. }
8. **public** **void** writeIntCrLf(**int** value) **throws** IOException {
9. **if** (value < 0) {
10. write((**byte**) '-');
11. value = -value;
12. }
14. **int** size = 0;
15. **while** (value > sizeTable[size])
16. size++;
18. size++;
19. **if** (size >= buf.length - count) {
20. flushBuffer();
21. }
23. **int** q, r;
24. **int** charPos = count + size;
26. **while** (value >= 65536) {
27. q = value / 100;
28. r = value - ((q << 6) + (q << 5) + (q << 2));
29. value = q;
30. buf[--charPos] = DigitOnes[r];
31. buf[--charPos] = DigitTens[r];
32. }
34. **for** (;;) {
35. q = (value \* 52429) >>> (16 + 3);
36. r = value - ((q << 3) + (q << 1));
37. buf[--charPos] = digits[r];
38. value = q;
39. **if** (value == 0) **break**;
40. }
41. count += size;
43. writeCrLf();
44. }
46. **public** **void** writeCrLf() **throws** IOException {
47. **if** (2 >= buf.length - count) {
48. flushBuffer();
49. }
51. buf[count++] = '\r';
52. buf[count++] = '\n';
53. }

Jedis内部特殊处理，为了符合redis发送命令格式。且重写了刷新方法，用于刷新自身缓存内的数据。

1. **public** **void** flush() **throws** IOException {
2. flushBuffer();
3. out.flush();
4. }
6. **private** **void** flushBuffer() **throws** IOException {
7. **if** (count > 0) {
8. out.write(buf, 0, count);
9. count = 0;
10. }
11. }

#### RedisInputStream

这是jedis自己编写的一个输入流。这个类假设（在某种程度上）我们正在读取返回流。它还假设，如果协议层需要一个字节，那么如果该字节不存在，则是流错误。

读取单一标识字节：

1. **public** **byte** readByte() **throws** JedisConnectionException {
2. ensureFill();
3. **return** buf[count++];
4. }
5. /\*\*
6. \* 此方法假定有需要读取的字节。如果我们不能再读取字节，就会抛出一个异常，
7. \* 以快速确定流是否比预期的小。
8. \*/
9. **private** **void** ensureFill() **throws** JedisConnectionException {
10. **if** (count >= limit) {
11. **try** {
12. limit = in.read(buf);
13. count = 0;
14. **if** (limit == -1) {
15. **throw** **new** JedisConnectionException("Unexpected end of stream.");
16. }
17. } **catch** (IOException e) {
18. **throw** **new** JedisConnectionException(e);
19. }
20. }
21. }

读取命令个数也相当于读取整数：

1. **public** **long** readLongCrLf() {
2. **final** **byte**[] buf = **this**.buf;
4. ensureFill();
6. **final** **boolean** isNeg = buf[count] == '-';
7. **if** (isNeg) {
8. ++count;
9. }
11. **long** value = 0;
12. **while** (**true**) {
13. ensureFill();
15. **final** **int** b = buf[count++];
16. **if** (b == '\r') {
17. ensureFill();
19. **if** (buf[count++] != '\n') {
20. **throw** **new** JedisConnectionException("Unexpected character!");
21. }
23. **break**;
24. } **else** {
25. value = value \* 10 + b - '0';
26. }
27. }

### JedisPool

#### 初始化的基础配置

JedisPool的配置参数大部分是由JedisPoolConfig的对应项来赋值的。

1. **public** **class** JedisPoolConfig **extends** GenericObjectPoolConfig {
2. **public** JedisPoolConfig() {
3. // defaults to make your life with connection pool easier :)
4. setTestWhileIdle(**true**);
5. setMinEvictableIdleTimeMillis(60000);
6. setTimeBetweenEvictionRunsMillis(30000);
7. setNumTestsPerEvictionRun(-1);
8. }
9. }
10. **public** **class** GenericObjectPoolConfig **extends** BaseObjectPoolConfig {
11. **public** **static** **final** **int** DEFAULT\_MAX\_TOTAL = 8;
12. **public** **static** **final** **int** DEFAULT\_MAX\_IDLE = 8;
13. **public** **static** **final** **int** DEFAULT\_MIN\_IDLE = 0;
14. **private** **int** maxTotal = 8;
15. **private** **int** maxIdle = 8;
16. **private** **int** minIdle = 0;
17. }
18. **public** **abstract** **class** BaseObjectPoolConfig **implements** Cloneable {
19. **public** **static** **final** **boolean** DEFAULT\_LIFO = **true**;
20. **public** **static** **final** **boolean** DEFAULT\_FAIRNESS = **false**;
21. **public** **static** **final** **long** DEFAULT\_MAX\_WAIT\_MILLIS = -1L;
22. **public** **static** **final** **long** DEFAULT\_MIN\_EVICTABLE\_IDLE\_TIME\_MILLIS = 1800000L;
23. **public** **static** **final** **long** DEFAULT\_SOFT\_MIN\_EVICTABLE\_IDLE\_TIME\_MILLIS = -1L;
24. **public** **static** **final** **int** DEFAULT\_NUM\_TESTS\_PER\_EVICTION\_RUN = 3;
25. **public** **static** **final** **boolean** DEFAULT\_TEST\_ON\_CREATE = **false**;
26. **public** **static** **final** **boolean** DEFAULT\_TEST\_ON\_BORROW = **false**;
27. **public** **static** **final** **boolean** DEFAULT\_TEST\_ON\_RETURN = **false**;
28. **public** **static** **final** **boolean** DEFAULT\_TEST\_WHILE\_IDLE = **false**;
29. **public** **static** **final** **long** DEFAULT\_TIME\_BETWEEN\_EVICTION\_RUNS\_MILLIS = -1L;
30. **public** **static** **final** **boolean** DEFAULT\_BLOCK\_WHEN\_EXHAUSTED = **true**;
31. **public** **static** **final** **boolean** DEFAULT\_JMX\_ENABLE = **true**;
32. **public** **static** **final** String DEFAULT\_JMX\_NAME\_PREFIX = "pool";
33. **public** **static** **final** String DEFAULT\_JMX\_NAME\_BASE = **null**;
34. **public** **static** **final** String DEFAULT\_EVICTION\_POLICY\_CLASS\_NAME = "org.apache.commons.pool2.impl.DefaultEvictionPolicy";
35. **private** **boolean** lifo = **true**;
36. **private** **boolean** fairness = **false**;
37. **private** **long** maxWaitMillis = -1L;
38. **private** **long** minEvictableIdleTimeMillis = 1800000L;
39. **private** **long** softMinEvictableIdleTimeMillis = 1800000L;
40. **private** **int** numTestsPerEvictionRun = 3;
41. **private** String evictionPolicyClassName = "org.apache.commons.pool2.impl.DefaultEvictionPolicy";
42. **private** **boolean** testOnCreate = **false**;
43. **private** **boolean** testOnBorrow = **false**;
44. **private** **boolean** testOnReturn = **false**;
45. **private** **boolean** testWhileIdle = **false**;
46. **private** **long** timeBetweenEvictionRunsMillis = -1L;
47. **private** **boolean** blockWhenExhausted = **true**;
48. **private** **boolean** jmxEnabled = **true**;
49. **private** String jmxNamePrefix = "pool";
50. **private** String jmxNameBase;
51. }

部分参数解析：

|  |  |  |
| --- | --- | --- |
| 配置参数 | 意义 | 默认值 |
| maxTotal | 对象总数 | 8 |
| maxIdle | 最大空闲对象数 | 8 |
| minIdle | 最小空闲对象书 | 0 |
| lifo | 对象池借还是否采用lifo | true |
| fairness | 对于借对象的线程阻塞恢复公平性 | false |
| maxWaitMillis | 借对象阻塞最大等待时间 | -1 |
| minEvictableIdleTimeMillis | 最小驱逐空闲时间 | 30分钟 |
| numTestsPerEvictionRun | 每次驱逐数量 | 3 |
| testOnCreate | 创建后有效性测试 | false |
| testOnBorrow | 出借前有效性测试 | false |
| testOnReturn | 还回前有效性测试 | false |
| testWhileIdle | 空闲有效性测试 | false |
| timeBetweenEvictionRunsMillis | 驱逐定时器周期 | false |
| blockWhenExhausted | 对象池耗尽是否block | true |

#### pool池中借对象

从jedisPool中获取Jedis实例路径：

1. **public** Jedis getResource() {
2. Jedis jedis = **super**.getResource();
3. jedis.setDataSource(**this**);
4. **return** jedis;
5. }
7. **public** T getResource() {
8. **try** {
9. **return** internalPool.borrowObject();
10. } **catch** (NoSuchElementException nse) {
11. **throw** **new** JedisException("Could not get a resource from the pool", nse);
12. } **catch** (Exception e) {
13. **throw** **new** JedisConnectionException("Could not get a resource from the pool", e);
14. }
15. }
17. **public** T borrowObject() **throws** Exception {
18. **return** **this**.borrowObject(**this**.getMaxWaitMillis());
19. }

AbandonedConfig 遗弃对象管理机制：

1. AbandonedConfig ac = **this**.abandonedConfig;
2. **if** (ac != **null** && ac.getRemoveAbandonedOnBorrow() && **this**.getNumIdle()
3. < 2 && **this**.getNumActive() > **this**.getMaxTotal() - 3) {
4. **this**.removeAbandoned(ac);
5. }

进行一个逻辑判断，一共4个条件：

* ac不为空
* ac的removeAbandonedOnBorrow属性为true
* 当前空闲对象的数量 < 2
* 当前活跃对象的数量 >（对象池最大数量-3）

后两个条件都为真的时候，空闲对象最多1个，可供新申请的对象资源最多2个：说明对象池中的资源使用趋于饱和了，资源快耗尽了。这个时候新的的请求过来，如果请求量较大，1、2个剩余资源被争抢后，剩下的请求就只能排队等待或者直接返回异常了。

AbandonedConfig配置类的参数解析：

1. **public** **class** AbandonedConfig {
3. /\*\*
4. \* Whether or not borrowObject performs abandoned object removal.
5. \* 如果设置为true，就意味着，在borrowObject方法中（从对象池中申请对象的时候）就可以进行遗弃对象的相关清理逻辑。
6. \* （当然是否能触发清理，还受限于其他条件）
7. \*/
8. **private** **boolean** removeAbandonedOnBorrow = **false**;
10. /\*\*
11. \* Whether or not pool maintenance (evictor) performs abandoned object removal.
12. \* 对象池本省可以通过GenericObjectPoolConfig配置后台异步清理任务，但是后台清理任务的主要职责是关注空闲（状态为空闲）对象的检测和清理。
13. \* 如果removeAbandonedOnMaintenance设置为true，就意味着，在对象池的异步清理任务中，也可以进行遗弃（状态为活跃）对象的相关清理。
14. \*/
15. **private** **boolean** removeAbandonedOnMaintenance = **false**;
17. /\*\*
18. \* Timeout in seconds before an abandoned object can be removed.
19. \* 这个时间，默认300s。如果对象池的一个对象被占用了超过300s还没有被释放，就认为是被调用方遗弃了。
20. \*/
21. **private** **int** removeAbandonedTimeout = 300;

24. /\*\*
25. \* Determines whether or not to log stack traces for application code
26. \* which abandoned an object.
27. \* 如果处理发了遗弃对象的回收和清理，是否要打印该对象的调用堆栈。
28. \* 生产环境十分不建议设为true，会很消耗性能。
29. \*/
30. **private** **boolean** logAbandoned = **false**;
32. /\*\*
33. \* PrintWriter to use to log information on abandoned objects.
34. \* Use of default system encoding is deliberate.
35. \* 这是一个日志输出器，用来定义调用堆栈日志的输出行为（输出到控制台、文件等）。默认输出到控制台。
36. \* 如果logAbandoned为false，就不会有输出行为。
37. \*/
38. **private** PrintWriter logWriter = **new** PrintWriter(System.out);
40. /\*\*
41. \* If the pool implements {@link UsageTracking}, should the pool record a
42. \* stack trace every time a method is called on a pooled object and retain
43. \* the most recent stack trace to aid debugging of abandoned objects?
44. \* 这个UsageTracking从名字上来直观翻译，叫做使用追踪，指的就是池中对象的使用追踪。UsageTracking也是一个接口，GenericObjectPool实现了该接口。关于UsageTracking我们后续单独详细解析，
45. \* 什么叫使用？
46. \*  就是只要是调用了池中对象的任何方法，就算使用了池中对象。
47. \* 什么叫追踪？
48. \*  就是在池中某个对象的任何一个方法被调用时，都会创建一个调用堆栈快照。
49. \*  logAbandoned为true时，useUsageTracking也为true时，那么回收被遗弃的对象时，就会打印该对象最后一次的调用堆栈信息了。
50. \*  如果useUsageTracking为true，即便是logAbandoned为false，那么每次对象的方法调用，一样还是会创建调用堆栈对象。只不过最终被回收时不会打印输出。
51. \*  生产环境该属性也不建议设置为true。
52. \*/
53. **private** **boolean** useUsageTracking = **false**;
54. }

removeAbandoned方法解析：

1. **private** **void** removeAbandoned(AbandonedConfig ac) {
2. **long** now = System.currentTimeMillis();
3. **long** timeout = now - (**long**)ac.getRemoveAbandonedTimeout() \* 1000L;
4. ArrayList<PooledObject<T>> remove = **new** ArrayList();
5. Iterator it = **this**.allObjects.values().iterator();
7. **while**(it.hasNext()) {
8. PooledObject<T> pooledObject = (PooledObject)it.next();
9. **synchronized**(pooledObject) {
10. /\*
11. 如果该对象的状态为ALLOCATED，也就是使用中状态（使用中不一定就是执行中，也不一定真的还在被使用）
12. pooledObject.getLastUsedTime()返回的就是对象的lastUseTime属性值，见名知意，就是该对象上一次被使用的时间
13. 这个值我们在解析borrowObject时提到过一点，就是在调用对象的allocte方法时，会把当前的毫秒时间戳赋值给lastUseTime
14. 所以如果：lastUseTime < timeout ,说明该对象已经在（removeAbandonedTimeout -> 默认300秒）的时间里没有被再次使用了。
15. \*/
16. **if** (pooledObject.getState() == PooledObjectState.ALLOCATED && pooledObject.getLastUsedTime() <= timeout) {
17. // 把该对象标记为ABANDONED（可舍弃）
18. pooledObject.markAbandoned();
19. // 把该对象添加到待移除列表中
20. remove.add(pooledObject);
21. }
22. }
23. }
25. Iterator itr = remove.iterator();
27. **while**(itr.hasNext()) {
28. PooledObject<T> pooledObject = (PooledObject)itr.next();
29. **if** (ac.getLogAbandoned()) {
30. pooledObject.printStackTrace(ac.getLogWriter());
31. }
33. **try** {
34. // 调用invalidateObject方法，来作废（销毁、释放资源）该对象
35. **this**.invalidateObject(pooledObject.getObject());
36. } **catch** (Exception var11) {
37. var11.printStackTrace();
38. }
39. }
41. }

借用空闲队列中的实例，并加入数据库选择：

1. PooledObject<T> p = **null**;
2. **boolean** blockWhenExhausted = **this**.getBlockWhenExhausted();
3. **long** waitTime = System.currentTimeMillis();
5. **while**(**true**) {
6. create = **false**;
7. // 判断对象池是否耗尽
8. **if** (blockWhenExhausted) {
9. // 队列中添加
10. p = idleObjects.pollFirst();
11. **if** (p == **null**) {
12. p = create();
13. **if** (p != **null**) {
14. create = **true**;
15. }
16. }
17. **if** (p == **null**) {
18. **if** (borrowMaxWaitMillis < 0) {
19. p = idleObjects.takeFirst();
20. } **else** {
21. p = idleObjects.pollFirst(borrowMaxWaitMillis,
22. TimeUnit.MILLISECONDS);
23. }
24. }
25. **if** (p == **null**) {
26. **throw** **new** NoSuchElementException(
27. "Timeout waiting for idle object");
28. }
29. **if** (!p.allocate()) {
30. p = **null**;
31. }
32. } **else** {
33. p = idleObjects.pollFirst();
34. **if** (p == **null**) {
35. p = create();
36. **if** (p != **null**) {
37. create = **true**;
38. }
39. }
40. **if** (p == **null**) {
41. **throw** **new** NoSuchElementException("Pool exhausted");
42. }
43. **if** (!p.allocate()) {
44. p = **null**;
45. }
46. }
48. **if** (p != **null**) {
49. **try** {
50. factory.activateObject(p);
51. } **catch** (Exception e) {
52. **try** {
53. destroy(p);
54. } **catch** (Exception e1) {
55. // Ignore - activation failure is more important
56. }
57. p = **null**;
58. **if** (create) {
59. NoSuchElementException nsee = **new** NoSuchElementException(
60. "Unable to activate object");
61. nsee.initCause(e);
62. **throw** nsee;
63. }
64. }
65. **if** (p != **null** && (getTestOnBorrow() || create && getTestOnCreate())) {
66. **boolean** validate = **false**;
67. Throwable validationThrowable = **null**;
68. **try** {
69. validate = factory.validateObject(p);
70. } **catch** (Throwable t) {
71. PoolUtils.checkRethrow(t);
72. validationThrowable = t;
73. }
74. **if** (!validate) {
75. **try** {
76. destroy(p);
77. destroyedByBorrowValidationCount.incrementAndGet();
78. } **catch** (Exception e) {
79. // Ignore - validation failure is more important
80. }
81. p = **null**;
82. **if** (create) {
83. NoSuchElementException nsee = **new** NoSuchElementException(
84. "Unable to validate object");
85. nsee.initCause(validationThrowable);
86. **throw** nsee;
87. }
88. }
89. }
90. }
91. // 更新池的出借状态
92. updateStatsBorrow(p, System.currentTimeMillis() - waitTime);
93. }

## 版本迭代

### 2.9.0

* 添加特定异常用于evalsha的未定义脚本
* 添加ipv6支持
* 添加集群的Hash Tag版本扫描方法
* 添加BITFIELD命令
* 添加集群支持密码
* 添加SSL支持

### 3.0.1

* [#1910](https://github.com/redis/jedis/issues/1910) JedisSentinelPool.initPool() 中的竞争条件导致 getResource() 上的 JedisConnectionException
* [#1920](https://github.com/redis/jedis/issues/1920) JedisPool耗尽问题

### 3.1.0-m4

* 添加对迁移时传递身份验证的支持

### 3.1.0-rc

* 升级到jdk1.8

### 3.1.0

* 添加redis流的处理（[#1880](https://github.com/redis/jedis/pull/1880/files)）
* 集群支持SSL（[#1896](https://github.com/redis/jedis/pull/1896)）
* 添加MEMORY DOCTOR命令

### 3.2.0

* 添加zpopmax和zpopmin命令
* 在事务和管道中添加缺失UNWATCH
* 添加在读取redis回应时阻塞
* 在JedisSentinentalPool中异常时，资源照样回到池里

### 3.3.0

* 添加Xinfo和BITFIELD\_RO命令
* 实现reids6的ACL
* 添加对自定义套接字工厂的支持
* 如果设置了sentinel密码，则发送验证消息（[#2154](https://github.com/redis/jedis/pull/2154)）
* 修改MasterListener中的sentinel配置参数

### 3.4.0

* 添加CLIENT、ACL、LOG、lpos、MEMORY USAGE、SET … GET option、SMISMEMBER、ZMSCORE、KEEPTTL 命令，更新SLOWLOG GET
* 当Redis从XClaim对已删除的条目返回nil时返回null

### 3.5.0

* 添加LPOP和RPOP的count参数支持
* 添加对sendCommand阻塞版本的支持

### 3.5.2

* 修复 PipelineBase 中的 xrevrange的start和end反向了

### 3.6.0

* 不再支持jdk1.7
* 引入配置模式
* Sentinel支持TLS
* 添加在JedisFactory中重置密码的功能
* 超时时间实现为double（[#2481](https://github.com/redis/jedis/pull/2481)）
* 添加对使用池使用JedisSocketFactory（自定义套接字）的支持
* 添加一些命令以及其参数的支持
* 修复了阻止所有集群的初始化时，插槽请求发送到特定实例的逻辑
* 添加CLIENT INFO 和CLIENT LIST对特殊id的支持
* 添加对RESTORE命令的ABSTTL、IDLETIME和FREQ参数支持
* 在 CLIENT KILL中添加对本地地址的支持
* 添加了对COPY、LMOVE、BLMOVE、ZINTER、HRANDFIELD、clientUnblock、ZDIFFSTORE、ZUNION等命令以及参数的支持
* 在redis群集连接失败时使用回退重试（[#2358](https://github.com/redis/jedis/pull/2358)）
* 修复参数类型和返回类型

### 3.6.3

* 在Pipeline支持WAIT

### 3.7.0

* 添加XAUTOCLAIM、CLIENT LIST、CLIENT PAUSE、More Cluster、ROLE、FAILOVER、stralgo、SCAN、SHUTDOWN的命令以及参数支持
* CONFIG SET返回OK状态
* 减少锁粒度

### 3.8.0

* 添加了aclLogReset（）方法以支持ACL LOG RESET子命令。这将替换aclLog（byte []）和aclLog（String）方法
* 更新插槽缓存时刷新节点
* 使用slf4j simple替换log4j实现

### 4.0.0

* 介绍了JedisPool的一种替代方案，它实现了与JedisCluster相同的接口，允许在JedisCluster和JedisPool之间轻松切换。
* 使用IllegalStateException替换无效的JedisDataException
* Sentinel支持TLS
* 从EVAL和EVALSHA命令中删除无限超时的用法
* 从事务中移除监听

<https://github.com/redis/jedis/releases>

### 4.0.1

* 修复事务中的zrevrangeByScore最大/最小参数顺序问题
* 添加重新搜索管道命令

# Redisson

基于Netty框架的事件驱动的通信层，其方法调用是异步的。Redisson的API是线程安全的，所以可以操作单个Redisson连接来完成各种操作。

# Lettuce

基于Netty框架的事件驱动的通信层，其方法调用是异步的。Lettuce的API是线程安全的，所以可以操作单个Lettuce连接来完成各种操作