第5章

Redis基础应用


视频讲解

除了可以作为缓存和基于键值对存储的非关系数据库来使用，Redis 还有一些基础应用，如简单的消息队列、可以一次性执行多条命令的流水线及地理位置信息处理等。本章将针对这些基础应用进行介绍。

5.1 发布-订阅

Redis 2.0 及以后的版本支持基于发布-订阅模式的消息机制。在这种模式下，消息发布者和订阅者不直接进行通信。发布者创建一个频道，并在上面发送消息，所有订阅该频道的客户端都能收到消息。Redis 的消息发布-订阅机制如图 5-1 所示，消息订阅者（订阅者 1、订阅者 2 和订阅者 3）订阅频道 Channel 1，消息发布者将消息发布到频道 Channel 1，该消息会被发送给三个订阅者。消息发布者在发送消息后无须关注有多少消息订阅者，更无须关注消息订阅者处理消息的细节，反过来看，消息订阅者也无须关注消息发布者的细节。也就是说，通过消息发布-订阅机制能够最大限度地实现发布者和订阅者间的解耦。同时，可以减少等待消息时不必要的轮询。发布-订阅的应用场景有即时聊天室、公众号订阅等。Redis 适合小型应用系统的发布-订阅，如果是大型应用，还应使用 RabbitMQ 或者 Kafka 等更专业的消息队列软件。

聊天室、公告板可以利用发布-订阅实现消息服务解耦。下面以简单的服务解耦进行说明。如图 5-2 所示，图中有两套业务，上面为朋友圈管理服务，负责管理朋友圈信息；下面为朋友圈频道，用户可以通过各种终端（手机、浏览器等）获取到朋友圈信息。

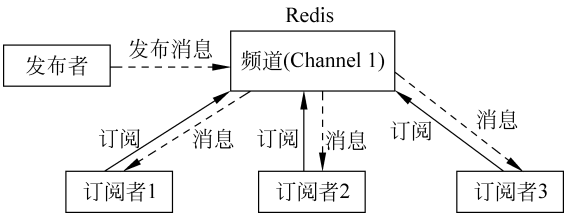

图 5-1 Redis 的消息发布-订阅机制

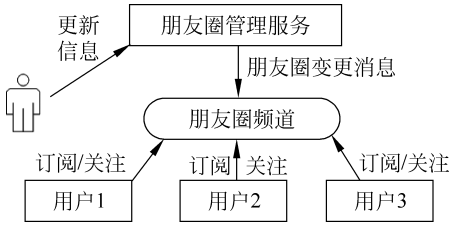

图 5-2 朋友圈中的发布-订阅

假如发布者更新了朋友圈信息，可以利用发布-订阅模式将更新后的信息发布到指定的频道，其他用户订阅这个频道可以及时更新朋友圈信息，通过这种方式可以有效解决朋友圈管理和朋友圈消息订阅两个业务的耦合性。

5.1.1 常用命令

表 5-1 列出了 Redis 提供的一系列用于发布-订阅机制的常用命令。

表 5-1 Redis 发布-订阅机制的常用命令

常 用 命 令	描　　述
PSUBSCRIBE	订阅一个或多个符合指定模式的频道
PUBSUB	查看发布和订阅系统的状态，由多个不同格式的子命令组成
PUBLISH	将消息发送到指定的频道
PUNSUBSCRIBE	退订所有指定模式的频道
SUBSCRIBE	订阅指定的一个或多个频道
UNSUBSCRIBE	退订指定的频道

下面演示发布-订阅是如何工作的。

首先，打开一个客户端连接 Redis 服务器，作为订阅者接收消息。创建一个订阅频道，命名为 redisChat。

```
redis > SUBSCRIBE redisChat
Reading messages... (press Ctrl - C to quit)
1) "subscribe"
2) "redisChat"
3) (integer) 1
```

其次，重新开启一个 Redis 客户端，作为发布者发送消息，然后在同一个频道 redisChat 发布两次消息，订阅者就可以接收消息。

```
redis > PUBLISH redisChat "message 1"
(integer) 1
redis > PUBLISH redisChat "message 2"
(integer) 1
```

订阅者的客户端会显示如下消息：

```
redis > SUBSCRIBE redisChat
Reading messages... (press Ctrl - C to quit)
1) "subscribe"
2) "redisChat"
3) (integer) 1
1) "message"
2) "redisChat"
3) "message 1"
1) "message"
2) "redisChat"
3) "message 2"
```

电商平台经常有传送订单数据的需求。下面，利用 Jedis 的 JedisPubSub 类来模拟两个模块间通过 Redis 消息队列交互的动作，来实现订单数据的发送与接收。具体步骤如下。

1. 引入相关依赖

在 Maven 项目中引入 Jedis 和 GSON 的依赖包。

```
1    < dependency >
2        < groupId > redis.clients </groupId >
```

```
        <artifactId>jedis</artifactId>
        <version>3.8.0</version>
    </dependency>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.8.6</version>
    </dependency>
```

2. 创建发布者

创建一个名为 Publisher.java 的文件，作为发布者，用于向频道中发送订单数据。代码如文件 5-1 所示。

【文件 5-1】　Publisher.java

```
1   public class Publisher {
2      public static void main(String[] args) {
3         HashMap<String,String> orderHM = new HashMap<String, String>();
4         orderHM.put("id","001");
5         orderHM.put("owner","Peter");
6         orderHM.put("amount","1000");
7         Gson gson = new Gson();
8         String jsonStr = gson.toJson(orderHM);
9         Jedis jedis = new Jedis("localhost",6379);
10        jedis.publish("MQChannel",jsonStr.toString());
11     }
12  }
```

如文件 5-1 所示，第 3～6 行首先创建了 HashMap 类型的订单对象 orderHM，并通过 put()方法模拟实际业务向订单对象中存放数据。由于消息队列一般只传输字符串，在发送前需要将 HashMap 对象转换为 JSON 类型的字符串(第 8 行)。在创建向 localhost:6379 的 Redis 连接后(第 9 行)，通过 Redis 连接对象 jedis 向 MQChannel 频道发送已转换为 JSON 格式的订单数据(第 10 行)。

3. 创建订阅者

创建一个名为 Subscriber.java 的文件，作为订阅者，用于从频道中读取订单数据。代码如文件 5-2 所示。

【文件 5-2】　Subscriber.java

```
1   public class Subscriber extends JedisPubSub {
2      //消息到来时触发
3      public void onMessage(String channel, String message) {
4         Gson gson = new Gson();
5         HashMap<String,String> orderMap =
6               gson.fromJson(message, HashMap.class);
7         System.out.println(orderMap.get("id"));
8         System.out.println(orderMap.get("owner"));
9         System.out.println(orderMap.get("amount"));
10     }
11
```

```
12        //订阅频道时触发
13        public void onSubscribe(String channel,
14            int subscribedChannels) {
15           System.out.println("subscribe the channel : " + channel);
16        }
17
18        //取消订阅时触发
19        public void onUnsubscribe(String channel,
20            int subscribedChannels) {
21           System.out.println("unsubscribe the channel : " + channel);
22        }
23
24        public static void main(String[] args) {
25           Subscriber subscriber = new Subscriber();
26           Jedis jedis = new Jedis("localhost",6379);
27           jedis.subscribe(subscriber, "MQChannel");
28        }
29     }
```

如文件 5-2 所示，第 1 行定义 Subscriber 类时继承了 JedisPubSub 类，并分别在第 3 行、第 13 行和第 19 行重写了 JedisPubSub 类的 onMessage()、onSubscribe()和 onUnsubscribe()方法。这三个方法分别在收到频道中的消息、订阅频道、取消订阅时触发。

由于在文件 5-1 中，订单对象以 JSON 格式发送到频道 QMChannel 中。因此，在 onMessage()方法中会通过第 5、6 行的代码把接收到的 JSON 格式的消息转换为 HashMap 对象，随后第 7～9 行的输出动作可以被理解为对订单对象的处理。在第 24 行的 main()方法中，在创建 Redis 连接对象 jedis 后（第 26 行），调用 Jedis 类的 subscribe()方法订阅 MQChannel 频道（第 27 行）。注意，该方法的第一个参数 subscriber 是一个消息处理类，设置后，该对象的 onMessage()、onSubscribe()和 onUnsubscribe()方法会在特定的时候被触发。

代码完成后，可以首先运行 Subscriber.java，可以在控制台看到如下输出。

```
subscribe the channel : MQChannel
```

通过该输出能够确认 Subscriber 成功地订阅了 MQChannel 频道。此时，Subscriber 还在继续侦听 MQChannel 频道。接下来可以运行 Publisher.java，该段代码没有输出，但是切换到 Subscriber.java 的控制台，输出内容如图 5-3 所示。

图 5-3　发布订单数据后，Subscriber 在控制台的输出

其中，第 2～4 行的输出结果是运行 Publisher.java 以后得到的，这说明 Subscriber.java 程序成功地接收到 Publisher.java 通过 MQChannel 频道发送过来的订单信息。

5.1.2 消息队列

Spring Data 为 Redis 提供专用的消息传递功能。Redis 消息传递可以分为两个领域：

(1) 发布或制作消息。

(2) 订阅或消费消息。

对于发布消息，既可以使用 RedisConnection 接口继承自 RedisPubSubCommands 接口(org. springframework. data. redis. connection. RedisPubSubCommands)的 publish()方法，也可以使用 RedisTemplate 类的 convertAndSend()方法。例如：

```
byte[] msg = ...
byte[] channel = ...
con.publish(msg, channel);
```

或

```
RedisTemplate template = ...
template.convertAndSend("hello!", "world");
```

其中，RedisConnection 接口需要原始数据(字节数组)作为参数，而 RedisTemplate 允许将任意对象作为消息传递。

对于订阅消息，RedisConnection 接口提供了 subscribe()和 pSubscribe()方法，分别按通道和模式进行订阅。Spring Data Redis 中的订阅命令是阻塞的。也就是说，调用 subscribe()方法会导致当前线程在开始等待消息时阻塞，只有在取消订阅时才会释放该线程。

对于阻塞式消息订阅，由于要对每个消息侦听器进行连接和线程管理，编码烦琐且性能表现不佳。Spring Data Redis 提供了消息侦听器容器类(RedisMessageListenerContainer)，实现异步处理消息。RedisMessageListenerContainer 充当消息侦听器容器。它用于从 Redis 接收消息，并驱动注入其中的消息侦听器(MessageListener)实例。侦听器容器负责管理接收消息的所有线程，并将消息分派到侦听器中进行处理。这使开发人员可以编写与接收消息相关的业务逻辑(并对其做出处理)，并将一些 Redis 基础架构问题委托给 Spring 框架。此外，为了使应用程序占用空间最少，RedisMessageListenerContainer 让多个侦听器共享一个连接和一个线程。表 5-2 列举了 Redis 的异步消息处理组件。

表 5-2 Redis 的异步消息处理组件

类 或 接 口	描 述
org. springframework. data. redis. listener. RedisMessageListenerContainer(类)	为 Redis 消息侦听器提供异步行为的容器。处理侦听、转换和消息调度的细节问题。与底层 Redis(每个订阅一个连接)相反，容器只使用一个连接，该连接对所有注册的侦听器进行“多路复用”，消息调度通过任务执行器完成
org. springframework. data. redis. connection. MessageListener(接口)	消息侦听器
org. springframework. data. redis. listener. adapter. MessageListenerAdapter(类)	消息侦听器适配器，通过反射将消息处理委托给目标侦听器方法，并具有灵活的消息类型转换能力。允许侦听器方法对消息内容进行操作，完全独立于 Redis API

下面介绍两种利用 Spring Data Redis 实现 Redis 发布-订阅的方法：第一种，利用消息侦听器；第二种，利用消息侦听器适配器。下面的例子模拟社交网络中关注公众号一类操

作。具体步骤如下。

1. 配置消息侦听器类

创建一个名为 RedisMessageListener 的消息侦听器类，实现 MessageListener 接口。代码如文件 5-3 所示。

【文件 5-3】 RedisMessageListener. java

```
package com.example.redis.message.spring;
//import 部分略
@Component
public class RedisMessageListener implements MessageListener {
    @Autowired
    private RedisTemplate template;

    @Override
    public void onMessage(Message message, byte[] pattern) {
      //获取消息
      byte[] messageBody = message.getBody();
      //使用值序列化器转换
      Object msg = template.getValueSerializer()
           .deserialize(messageBody);
      //获取侦听的频道
      byte[] channelByte = message.getChannel();
      //使用字符串序列化器转换
      Object channel = template.getStringSerializer()
           .deserialize(channelByte);
      //频道名称转换
      String patternStr = new String(pattern);
      System.out.println("--pattern--:" + patternStr);
      System.out.println("--频道--:" + channel);
      System.out.println("--消息内容:--" + msg);
    }
}
```

2. 配置侦听容器并订阅频道

可在文件 3-6 基础上增加消息侦听器容器的配置。修改后的完整代码如文件 5-4 所示。

【文件 5-4】 RedisConfig. java

```
package com.example.redis.message.spring;
//import 部分略
@Configuration
@ComponentScan(basePackages = "com.example.redis.message.spring")
public class RedisConfig {

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
         //代码见文件 3-6
    }
    @Bean
    public RedisTemplate template(@Autowired
        RedisConnectionFactory rcf){
        //代码见文件 3-6
```

```
15        }
16
17        @Bean
18        public RedisMessageListenerContainer container(
19            RedisConnectionFactory rcf, RedisMessageListener rml) {
20            RedisMessageListenerContainer container =
21                new RedisMessageListenerContainer();
22            container.setConnectionFactory(rcf);
23            container.addMessageListener(rml,
24                new ChannelTopic("redis.life"));
25            container.addMessageListener(rml,
26                new ChannelTopic("redis.news"));
27            /*container.addMessageListener(rml,
28                new PatternTopic("redis.*")); */
29            return container;
30        }
31    }
```

如文件 5-4 所示，在创建消息侦听器容器对象后（第 20～22 行），订阅 redis.life 和 redis.news 两个频道（第 23～26 行）。可以向容器中添加多个消息侦听器。除了按频道名称实现订阅，也可以按模式实现订阅，如第 27、28 行，订阅以 redis.开头的所有频道。

3. 编写测试代码

可以使用 RedisTemplate 的 convertAndSend()方法向指定频道发布消息，同时查看消息处理情况。测试代码如文件 5-5 所示。

【文件 5-5】　TestMessageAsync.java

```
1     //import 部分略
2
3     @RunWith(SpringJUnit4ClassRunner.class)
4     @ContextConfiguration(classes = RedisConfig.class)
5     public class TestMessageAsync {
6        @Autowired
7        private RedisTemplate template;
8
9        @Test
10       public void publish(){
11          template.convertAndSend("redis.life","this is life");
12          template.convertAndSend("redis.news","this is news");
13       }
14    }
```

如文件 5-5 所示，可以利用 RedisTemplate 的 convertAndSend()方法向指定的 redis.life 和 redis.news 频道发送消息。运行测试代码后，控制台输出的消息处理结果如图 5-4 所示。

```
16:58:30.009 [main] DEBUG org.springframework.
--pattern--:redis.news
--pattern--:redis.life
16:58:30.009 [main] DEBUG org.springframework.
16:58:30.009 [main] DEBUG org.springframework.
16:58:30.009 [main] DEBUG org.springframework.
--频道--: redis.news
--频道--: redis.life
--消息内容:--this is news
--消息内容:--this is life
```

图 5-4　消息处理结果

此外，还可以利用消息侦听器适配器实现消息处理。具体步骤如下。

1）定义一个消息接收器类

创建一个名为 MessageReceiver 的消息接收

器类，代码如文件 5-6 所示。

【文件 5-6】 MessageReceiver. java

```
@Component
public class MessageReceiver {
    public void receiveMessage(String message, String channel){
        System.out.println(" -- 频道 -- " + channel);
        System.out.println(" -- 消息 -- " + message);
    }
}
```

2）配置消息侦听器适配器

可以在文件 5-4 中增加一个消息侦听器适配器 Bean，代码如文件 5-7 所示。

【文件 5-7】 RedisConfig. java 中增加的侦听器适配器 Bean

```
@Bean
public MessageListenerAdapter listenerAdapter(
    MessageReceiver receiver) {
    return new MessageListenerAdapter(receiver, "receiveMessage");
}
```

其中，MessageListenerAdapter 的构造方法指定两个参数：第一个参数 receiver 指定消息处理器的代理，即第一步中定义的消息接收器对象；第二个参数指定代理中用来处理消息的方法名，即第一步中 MessageReceiver 类中用来处理消息的方法 receiveMessage()。

3）配置侦听器容器

可以将文件 5-4 中侦听器容器的配置参数由消息侦听器改为消息侦听器适配器，如文件 5-8 所示。

【文件 5-8】 RedisConfig. java 中配置的消息侦听器适配器

```
@Bean
public RedisMessageListenerContainer container(
  RedisConnectionFactory factory, MessageListenerAdapter adapter){
    RedisMessageListenerContainer container = new
      RedisMessageListenerContainer();
    container.setConnectionFactory(factory);
    container.addMessageListener(adapter, new PatternTopic
      ("redis.*"));
    return container;
}
```

4）运行测试

执行上述修改后，重新运行文件 5-5 的测试代码，可实现同样的消息处理效果。

至此，关于使用 Spring Data Redis API 实现 Redis 发布-订阅应用程序开发已介绍完毕。和很多专业的消息队列系统（例如 Kafka、RabbitMQ）相比，Redis 的发布-订阅略显粗糙，例如无法实现消息堆积和回溯。但 Redis 的发布-订阅足够简单，如果当前场景可以容忍这些缺点，那么使用 Redis 作为消息队列也是一个不错的选择。

5.2　Redis 流

5.2.1　Redis 流简介

Redis 流(Redis Stream)(以下称流)是 Redis 5.0 新增加的数据结构。流的作用类似于只追加(Append-only)日志,主要用于消息队列(Message Queue,MQ)。Redis 有一个发布-订阅机制来实现消息队列的功能,但它有一个缺点就是消息无法持久化。如果出现网络断开、Redis 宕机等情况,消息就会被丢弃。简单来说,发布-订阅可以分发消息,但无法记录历史消息。Redis 流是日志形式的存储结构,可以添加数据。流将为每个数据生成一个时间戳 ID。可以使用这些 ID 检索其关联的条目,或读取和处理流中所有后续条目。因此,流适用于消息队列和时间序列存储。流主要有以下使用场景。

(1) 事件源(例如,跟踪用户操作、点击等)。

(2) 传感器监测(例如,现场设备的读数)。

(3) 通知(例如,在单独的流中存储每个用户的通知记录)。

流的结构如图 5-5 所示。

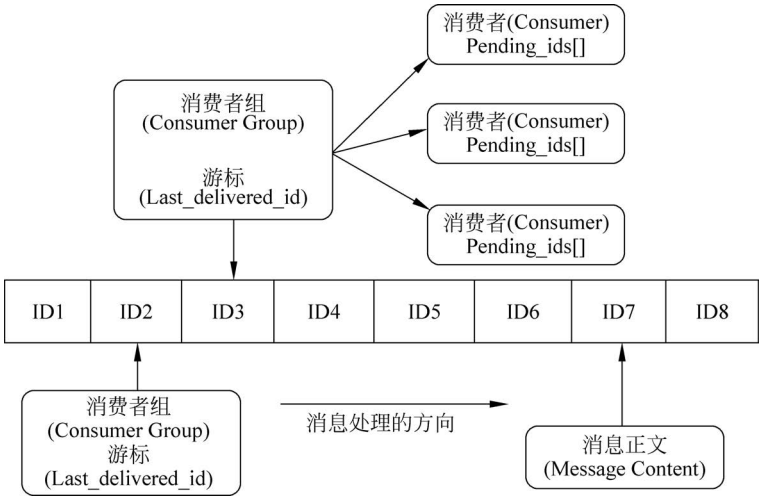

图 5-5　流的结构

如图 5-5 所示,流维护了一个消息链表,将所有加入的消息都串起来,每个消息都有一个唯一的 ID 和对应的内容。其中:

(1) 消息正文(Message Content):存放了每条消息的真实数据。

(2) 消费组(Consumer Group):使用 XGROUP CREATE 命令创建,一个消费组有多个消费者(Consumer)。

(3) 游标(Last_delivered_id):每个消费组会有个游标 Last_delivered_id,任意一个消费者读取了消息都会使游标 Last_delivered_id 往前移动。

(4) 消费者(Consumer):消息的最终消费者,从队列中获取消息进行消费。

(5) 消费者状态变量(Pending_ids):记录了当前已经被客户端读取的消息,但是还没有确认,作用是维护消费者的未确认的 ID。

在介绍流的应用案例前，先了解有关流的几个简单的命令。

（1）XADD：添加消息到末尾。语法：XADD key ID field value[field value...]。注意，如果将ID设置为*，则表示由Redis自动生成ID。

（2）XLEN：获取流包含的元素数量，即消息队列的长度。语法：XLEN key。

（3）XRANGE：获取消息列表，会自动过滤已经删除的消息。语法：XRANGE key start end[COUNT count]。其中，start为“-”表示最小值，end为“+”表示最大值。COUNT选项可指定返回的条目数。

下面列举一个应用流命令的简单示例。

（1）向流中添加一条消息。

```
redis > XADD mystream * name LiuZongYuan age 16
"1674825991997 - 0"
```

（2）获取key为mystream的流的长度。

```
redis > XLEN mystream
(integer) 1
```

（3）获取key为mystream的流中的所有消息。

```
redis > XRANGE mystream - +
1) 1) "1674825991997 - 0"
   2) 1) "name"
      2) "LiuZongYuan"
      3) "age"
      4) "16"
```

Redis流操作可分为两个功能：追加记录与消费记录。这种模式与发布-订阅有相似之处，其主要区别在于消息的持久性以及消息的使用方式。发布-订阅依赖于瞬时消息的广播（即，如果你不听，你就会错过一条消息），但流使用一种持久的、只追加的数据类型，可以在流被修剪之前保留消息。消息使用方式的区别是发布-订阅注册了服务器端订阅，Redis将到达的消息推送到客户端，而流需要主动轮询。在Spring Data Redis中，流的主要操作由org.springframework.data.redis.connection包和org.springframework.data.redis.stream包中的类或接口提供。

5.2.2 Redis流操作之追加

在Spring Data Redis中，如果要向流追加记录，与其他操作一样，可以使用RedisConnection接口或StreamOperations<K，HK，HV>接口（org.springframework.data.redis.core.StreamOperations<K，HK，HV>）来实现。这两个接口分别提供了xAdd()和add()方法用于向流中追加记录。其中RedisConnection接口提供的xAdd()方法以原始数据（字节数组）作为参数，而StreamOperations接口提供的add()方法允许将任意对象作为记录传入。

1. 使用RedisConnection接口执行追加

下面的例子模拟利用流存储传感器传送的实时数据，如温度、压力等。创建一个名为StreamRedisConfig.java的类。代码如文件5-9所示。

【文件 5-9】　StreamRedisConfig.java

```
1    @Configuration
2    public class StreamRedisConfig {
3        @Bean
4        public RedisConnectionFactory redisConnectionFactory() {
5          //代码可见文件 3-6
6        }
7    }
```

如文件 5-9 所示，在 StreamRedisConfig 配置类中创建 RedisConnectionFactory 的配置元数据，代码可参考文件 3-6。随后，编写测试代码，利用 RedisConnection 向流中追加记录，测试代码如文件 5-10 所示。

【文件 5-10】　TestStreamsDemo.java

```
1    @RunWith(SpringJUnit4ClassRunner.class)
2    @ContextConfiguration(classes = StreamRedisConfig.class)
3    public class TestStreamsDemo {
4        @Autowired
5        private RedisConnectionFactory rcf;
6
7        @Test
8        public void testAppendByConnection() {
9            RedisConnection conn = rcf.getConnection();
10           byte[] stream = "my-stream".getBytes();
11           Map<byte[],byte[]> map = new HashMap<>();
12           map.put("sensor".getBytes(),"tp-sensor".getBytes());
13           map.put("temperature".getBytes(),"21.0C".getBytes());
14           map.put("pressure".getBytes(),"3500Kpa".getBytes());
15           ByteRecord record =
16               StreamRecords.rawBytes(map).withStreamKey(stream);
17           conn.xAdd(record);
18       }
19   }
```

如文件 5-10 所示，第 9 行利用 RedisConnectionFactory 对象获取 RedisConnection 对象。由于 RedisConnection 只能接受字节数组作为参数，第 10～14 行将追加到流中的键（第 10 行）、字段和值（第 12～14 行）都转换为字节数组。第 15、16 行首先调用 StreamRecords 类的静态方法 rawBytes()，为给定的原始字段值对创建新的 ByteRecord，其中 ByteRecord 接口代表流中由二进制字段值对集合所支持的记录。随后，调用 ByteRecord 对象的 withStreamKey(byte[]key)方法，创建一个新的指定了键的 ByteRecord 对象。第 17 行调用 RedisConnection 接口继承自 RedisStreamCommands 接口的 xAdd()方法。xAdd()方法的原型如下：

```
@Nullable
default RecordId xAdd(byte[] key, Map<byte[],byte[]> content)
```

该方法的功能是向键 key 指定的流中追加一条新的记录，记录的内容由 Map 对象中的字段和值组成。运行此测试代码后，可以在 Redis 的客户端利用 XRANGE 命令查看运行结果：

```
redis > XRANGE my - stream  -  +
1) 1) "1674890806976 - 0"
   2) 1) "temperature"
      2) "21.0C"
      3) "sensor"
      4) "tp - sensor"
      5) "pressure"
      6) "3500Kpa"
```

2. 使用 StreamOperations 接口执行追加

对于文件 5-10 中的测试数据，也可以利用 StreamOperations 接口提供的 add()方法追加到流。而 StreamOperations 对象需要调用 RedisTemplate 类的 opsForStreams()方法获取。因此，可在文件 5-9 的基础上增加 RedisTemplate 类的配置元数据，代码如文件 5-11 所示。

【文件 5-11】 文件 5-9 增加的配置元数据

```
1    @Bean
2    public RedisTemplate < String, Map < String,String >> template(
3      @Autowired RedisConnectionFactory rcf){
4      RedisTemplate < String, Map < String,String >> template  =  new
5        RedisTemplate <>();
6      template.setConnectionFactory(rcf);
7      template.setDefaultSerializer(RedisSerializer.string());
8      return template;
9    }
```

Spring Data Redis 中，所有发送到流的记录都需要序列化为二进制格式。因此，文件 5-11 的第 7 行将 Redis 默认的序列化器设置为 StringRedisSerializer。

随后，可以编写测试代码，利用 StreamOperations 对象完成追加。可在文件 5-10 的基础上增加一个测试用例，如文件 5-12 所示。

【文件 5-12】 文件 5-10 增加的测试用例

```
1    @Test
2    public void testAppendByTemplate() {
3      Map < String,String > map  =  new HashMap <>();
4      map.put("sensor","tp - sensor2");
5      map.put("temperature","62.0C");
6      map.put("pressure","4300Kpa");
7      StreamOperations < String,String,String > ops  =
8        template.opsForStream();
9      ops.add("my - stream2",map);
10   }
```

5.2.3 Redis 流操作之消费

消费者可以消费一个或多个流。流提供读取命令，允许从已知流中的任意位置消费流（随机访问），并且可以超出流末端消费新的流记录。RedisConnection 接口从 RedisStreamCommands 接口继承了 xRead()和 xReadGroup()方法，分别对应 Redis 命令 XREAD 和 XREADGROUP，

从一个或多个流中读取数据以及在消费者组内进行读取。Redis 中的订阅命令可能会被阻塞。也就是说，在 Redis 连接上调用 xRead()方法会导致当前线程在开始等待消息时阻塞。只有当读取命令超时或收到消息时，线程才会被释放。要使用流消息，可以轮询应用程序代码中的消息，或者通过消息侦听器容器使用两种异步接收方式中的一种，即命令式或响应式。每次新记录到达时，容器都会通知应用程序代码。接下来，本节从同步接收、异步接收和消息偏移量三方面介绍流的消费操作。

1. 同步接收消息

虽然流消费通常与异步处理相关，但也可以从流中同步消费消息。StreamOperations 接口中重载的 read()方法可以实现这个功能。在同步接收期间，调用线程可能会阻塞，直到消息可用为止。例如，准备消费流中已追加的数据，可以利用文件 5-13 所示的代码消费流数据(流中的数据已在文件 5-10 和文件 5-12 中完成追加)。

【文件 5-13】　部分消费者代码

```
1    @Test
2    public void testRecMessageSynchronously() {
3      StreamOperations < String,String,String > ops =
4         template.opsForStream();
5      List < MapRecord < String, String, String >> messages = ops.read(
6        StreamReadOptions.empty(),StreamOffset.fromStart("my - stream1"));
7      if(messages!= null)
8         messages.forEach(msg - > System.out.println(msg.getStream() + ":" +
9         msg.getId() + " : " + msg.getValue()));
10   }
```

如文件 5-13 所示，第 3、4 行获取 StreamOperations 对象。第 5 行调用 StreamOperations 接口的 read()方法消费(读取)流中的数据。read()方法中指定了两个参数：第一个指定读操作的参数，通过调用 StreamReadOptions 类的静态方法 empty()创建一个空的 StreamReadOptions 对象；第二个参数指定要读取的流。第 6 行指定从 my-stream1 流的起始位置开始读取。对于读取到的流中的记录(MapRecord)列表，利用 Lambda 表达式对列表中的每个元素逐一处理(第 8、9 行)。其中的记录都是由字段值对(Field-Value Pairs)组成的。MapRecord 接口继承了 Record 接口的三个方法，其中 getStream()方法返回流在 Redis 中的键；getId()方法返回流中实体的 ID；getValue()方法返回流中实体的值。此测试用例的运行结果如图 5-6 所示。

```
my-stream1 : 1674913205847-0 : {sensor=tp-sensor1, temperature=21.0C, pressure=3500Kpa}
```

图 5-6　文件 5-13 测试用例运行结果

在某些应用场景中，需要的不是向许多客户端提供相同的消息流，而是向客户端提供同一消息流的不同子集。例如，对于一个处理速度慢的消息，可以让 N 个不同的工作人员接收消息流的不同部分，这就需要将不同的消息路由到不同的工作人员来处理消息。为了实现这一点，Redis 使用了一个叫作消费者组的概念。从实现的角度来看，Redis 消费者组与 Kafka 消费者组无关，尽管它们在功能上是相似的。消费者组就像一个从流中获取数据的伪消费者，实际上为多个消费者提供服务，并做出如下保证。

(1) 每个消息都提供给不同的消费者，不可能将同一消息传递给多个消费者。

（2）在消费者组中，每个消费者需要通过名称来标识。名称是区分大小写的字符串，每个消费者必须指定名称。

（3）每个消费者组都具有一个“从未使用过的第一个 ID”的概念，其作用相当于书签。因此，当消费者请求新消息时，流可以只提供以前未传递的消息。

（4）使用（或消费）消息需要使用 XACK 命令进行确认。Redis 将确认解释为：此消息已正确处理，可以将其从消费者组中移除。

（5）消费者组会跟踪当前挂起的所有消息，即已传递给消费者组中的某个消费者但尚未确认为已处理的消息。当查看流消息的历史记录时，每个消费者只能看到传递给它自己的消息。

要使用消费者组处理消息，首先要在 Redis 中创建消费者组。可以使用下述命令创建消费者组：

```
XGROUP CREATE mystream mygroup $
```

其中，mystream 表示已创建流的键，mygroup 表示指定的消费者组的名称，$ 是消费者组的 ID。$ 的含义为只有到达流中的新消息才会提供给组中的消费者。如果将 ID 指定为 0，则消费者组将使用流历史记录中的所有消息。当然，也可以将 ID 指定为其他的合法值。可利用文件 5-10 或文件 5-12 创建的流创建消费者组 myGroup，并指定 myGroup 消费流中的所有消息，编写的测试代码如文件 5-14 所示。

【文件 5-14】 消费者组测试代码

```
1   @Test
2   public void testRecMessageSynchronouslyByGroup() {
3     StreamOperations<String, String, String> ops =
4       template.opsForStream();
5     List<MapRecord<String, String, String>> messages = ops.read(
6       Consumer.from("myGroup","myConsumer"),
7       StreamReadOptions.empty(),
8       StreamOffset.create("my-stream1",ReadOffset.lastConsumed()));
9     if(messages!= null){
10      messages.forEach(msg -> { System.out.println(
11        msg.getStream() + " : " + msg.getId() + " : " + msg.getValue());
12        ops.acknowledge("myGroup",msg);
13    });}
14  }
```

如文件 5-14 所示，第 5 行调用 StreamOperations 接口的 read()方法消费（读取）流数据，需要指定 read()方法的三个参数（第 6～8 行）。第 6 行调用 Consumer 类的静态方法 from()创建一个新的消费者，指定消费者组为 myGroup、消费者名称为 myConsumer。第 8 行指定从 my-stream1 流中消费消息。第 10、11 行用于消费消息。通过消费组读取消息时，服务器将记录给定的消息已发送，并将其添加到已发送但尚未确认的消息列表（Pending Entries List，PEL）中。第 12 行调用 StreamOperations 接口的 acknowledge()方法向 Redis 服务器确认当前消息已被处理，可以将其从消费者组中移除。

运行文件 5-14 的测试代码，控制台的输出如图 5-6 所示。此外，还可以通过 RedisInsight 查看第 12 行发送确认命令（XACK 命令）前后 Redis 消息队列的变化情况。图 5-7 和图 5-8

分别展示了发送确认命令前后的情况。

图 5-7　发送确认命令前的情况

图 5-8　发送确认命令后的情况

如图 5-7 所示，如果不发送确认命令（移除文件 5-14 第 12 行代码），Redis 服务器显示有一条消息挂起（Pending）。当执行确认命令后，则该消息从流中移除，挂起的消息数量为 0，如图 5-8 所示。

2. 异步接收消息

由于其阻塞特性，同步接收消息没有吸引力，因为它需要对每个消费者进行连接和线程管理。为了解决这个问题，Spring Data 提供了消息侦听器，它可以实现异步接收消息。Spring Data 提供了两种针对不同编程模型的消息侦听器容器。

（1）StreamMessageListenerContainer 接口充当命令式编程模型的消息侦听器容器。它用于消费来自流的记录，并驱动注入其中的 StreamListener 实例。

（2）StreamReceiver 接口提供了消息侦听器的响应式变体。它可以将来自流的消息作为潜在的无限流使用，并通过 Flux（一个利用单向数据流实现的应用架构）发出流消息。

StreamMessageListenerContainer 和 StreamReceiver 负责将接收消息和发送消息的所有线程交由侦听器处理。消息侦听器容器（或消息接收器）是消息驱动 POJO（Message Driven POJOs，MDPs）和消息提供者之间的中介，负责注册以接收消息、获取和释放资源、执行异常转换等。这使得应用程序开发人员可以编写与接收消息相关的业务逻辑，其余细节问题交给框架处理。

这两个容器都允许更改运行时配置，以便在应用程序运行时添加或删除订阅。此外，容器使用延迟订阅方法，仅在需要时使用 RedisConnection。如果所有侦听器都被取消订阅，那么容器会自动执行清理，并释放线程。

3. 消息偏移量

执行流读取操作时，Redis 从给定的偏移量开始读取流记录。ReadOffset 类（org.springframework.data.redis.connection.stream.ReadOffset）定义了读取偏移量（文件 5-14 第 8 行）。Redis 支持 3 种偏移量，具体取决于是独立使用流还是在消费者组内使用流。

（1）ReadOffset.latest()：读取最新消息。

（2）ReadOffset.from()：读取特定消息 ID 之后的消息。

（3）ReadOffset.lastConsumed()：读取最后使用的消息 ID 之后的消息（仅针对消费者组）。

5.2.4　Redis 流操作之序列化

所有发送到流中的记录都需要序列化。由于流与哈希数据结构非常接近，因此流的键、字段和值均使用了 RedisTemplate 配置的序列化器。下面列出了相关的序列化器。

（1）键使用序列化器 keySerializer，用于 Record#getStream()方法。

（2）字段使用序列化器 hashKeySerializer，用于序列化哈希中的每个字段。

（3）值使用序列化器 hashValueSerializer，用于序列化哈希中的每个值。

如果不使用任何序列化程序，则需要确保记录值已经是二进制的。对于 5.2.2 节和 5.2.3 节的例子，已在文件 5-11 中设置使用默认的序列化器 StringRedisSerializer。

向流中添加一个复杂值可以通过三种方式完成。

（1）将复杂值转换为简单值，如使用 JSON 字符串表示。

（2）使用合适的序列化器序列化值。

（3）使用 HashMapper 将值转换为适合序列化的 Map。

其中，第一种方式是最直接的，但忽略了流结构所提供的字段值，流中的值对其他消费者来说仍然是可读的。第二种方式的优点与第一种方式相同，但要求所有的消费者都必须实现相似的序列化机制。第三种方式，使用 HashMapper 接口是一种更复杂的方法，它使用了流的哈希结构，但对源代码进行了扁平化处理。只要选择了合适的序列化程序，其他用户仍然能够读取记录。

HashMapper 接口可将 Java 对象转换为 Redis 的哈希类型或 Map 类型（java.util.Map）。需要使用能够序列化和反序列化的哈希键和哈希值序列化器。如，要将一个普通的 User 对象发送到流中，可按以下步骤执行。

第一步，创建 User 类，代码如文件 5-15 所示。

【文件 5-15】 User.java

```
@Data
public class User {
  private int id;
  private String name;
  private String pass;
  public User(int id, String name, String pass) {
     this.id = id;
     this.name = name;
     this.pass = pass;
  }
  //此处省略了 toString()方法
}
```

第二步，编写测试代码，向流中发送 User 对象，如文件 5-16 所示。

【文件 5-16】 部分测试代码

```
@Test
public void testStreamForComplexValue() {
  StreamOperations<String,String,User> ops = template.opsForStream();
  ObjectRecord<String, User> record = StreamRecords.newRecord()
     .in("user-stream").ofObject(new User(1,"Tom","angel"));
  ops.add(record);
  List<ObjectRecord<String,User>> records = ops.read(
     User.class,StreamOffset.fromStart("user-stream"));
  records.forEach(rec->System.out.println("id=" + rec.getId() +
  ",stream=" + rec.getStream() + ",value=" + rec.getValue().toString()));
}
```

如文件 5-16 所示，第 4 行调用 StreamRecords 类的 newRecord()方法创建记录构建器(org. springframework. data. redis. connection. stream. StreamRecords. RecordBuilder < S >)用以在流中创建记录。随后，调用 RecordBuilder 类的 in()方法来配置流的键。再调用 RecordBuilder 类的 ofObject()方法创建一个 Object 类型的记录(第 5 行)。第 6 行调用 add()方法向流中发送 User 对象。第 8、9 行从流中取出记录并消费(第 9、10 行)。运行测试代码后的控制台输出如图 5-9 所示。

```
id =1674999570815-0, stream=user-stream,value= User{id=1, name='Tom', pass='angel'}
```

图 5-9　控制台输出

5.3　流水线

Redis 流水线(Pipelining)技术是一种提高 Redis 性能的技术，它允许同时发出多个命令而无须等待每个命令的单独响应，可以在全部命令执行结束后一次性返回运行结果。Redis 是一个使用客户-服务器(C/S)模型的 TCP 服务器，它采用与 HTTP 类似的请求-响应协议。一般情况下，用户每执行一个 Redis 命令，客户端与服务器都需要进行一次通信。客户端将命令发送给服务器，随后客户端会阻塞并等待 Redis 服务器处理，服务器将命令执行的结果以响应报文的形式返回给客户端。执行一条 Redis 命令分为如下 4 个步骤：

(1) 发送命令。

(2) 命令排队。

(3) 执行命令。

(4) 返回结果。

这 4 个步骤的时间消耗称为往返时间(Round Trip Time，RTT)。大部分的 Redis 命令是不支持批量操作的。例如，要执行 n 次 HGETALL 命令，需要消耗 n 次往返时间。实际项目中，Redis 的客户端和服务器可能部署在不同的机器上。例如，客户端在 A 地，Redis 服务器在 B 地，两地直线距离约为 2600km，那么 1 次 RTT 时间＝2600×2/(300 000×2/3)＝26ms(光在真空中的传播速度为每秒 3×10^8 m，这里假设光在光纤中的传播速度为真空光速的 2/3)，那么客户端在 1 秒内大约能执行 40 条命令，这与 Redis 的高并发高吞吐特性背道而驰。

Redis 的流水线机制能够改善上述问题。流水线能够将一组 Redis 命令进行组装，一次向服务器发送多条命令而无须等待回复，然后在一个步骤中读取回复。在没有使用流水线时执行 n 条命令的情形如图 5-10 所示，使用流水线时执行 n 条命令的情形如图 5-11 所示。

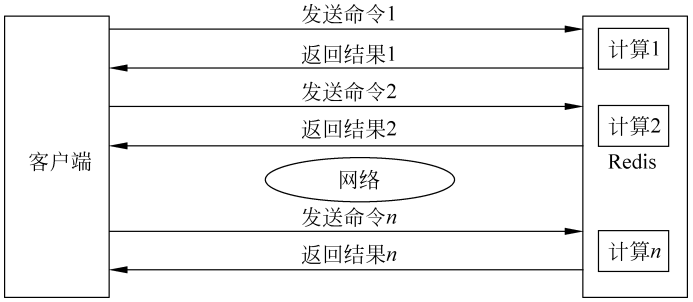

图 5-10　没有使用流水线时执行 n 条命令的情形

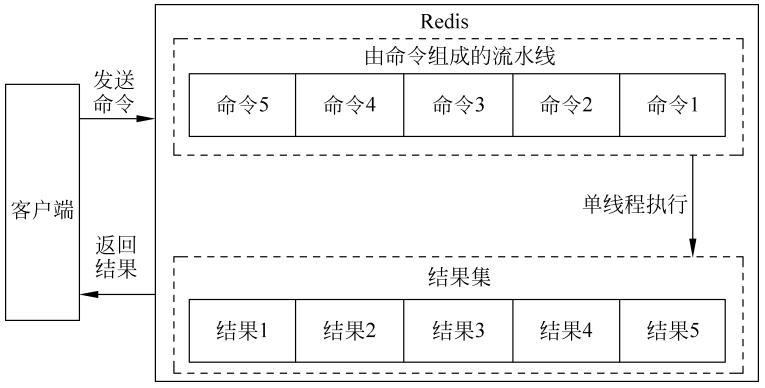

图 5-11　使用流水线时执行 n 条命令的情形

在没有使用流水线的情况下，执行 n 条命令耗时 n * RTT；而使用流水线执行 n 条命令时，耗时 RTT。流水线方式可以大量节省命令和结果的传输时间，从而提高程序性能。

可以使用 RedisTemplate 来执行流水线操作，RedisTemplate 提供的流水线方法如表 5-3 所示。

表 5-3　RedisTemplate 提供的流水线方法

方 法 原 型	说　　明
List < Object > executePipelined(RedisCallback <?> action)	在流水线上执行给定的操作 action 并返回结果
List < Object > executePipelined(RedisCallback <?> action, @Nullable RedisSerializer <?> resultSerializer)	在流水线上执行给定的操作 action 并使用专用的序列化器 resultSerializer 返回结果
List < Object > executePipelined(SessionCallback <?> session)	在流水线上执行 Redis 会话 session
List < Object > executePipelined(SessionCallback <?> session, @Nullable RedisSerializer <?> resultSerializer)	在流水线上执行 Redis 会话 session 并使用专用的序列化器 resultSerializer 返回结果

在下面的范例中，向 Redis 中批量插入 10 000 个键，比较了不采用流水线方式和采用流水线方式时的性能差异。代码如文件 5-17 所示。

【文件 5-17】　TestPipelineDemo. java

```
1   @RunWith(SpringJUnit4ClassRunner.class)
2   @ContextConfiguration(classes = RedisConfig.class)
3   public class TestPipelineDemo {
4
5       @Autowired
6       private StringRedisTemplate template;
7
8       private final int batchSize = 10000;
9       @Test
10      public void testPipeline() {
11          long start = System.currentTimeMillis();
12          for(int cnt = 0; cnt < batchSize;cnt++){
13              template.opsForValue().set("key" + cnt, String.valueOf(cnt));
14              template.opsForValue().get("key" + cnt);
```

```
15          }
16          long end = System.currentTimeMillis();
17          System.out.println("time cost by Non-pipeline :"
18                + (end-start) + " ms.");
19
20          long s = System.currentTimeMillis();
21          template.executePipelined((RedisCallback<Object>) conn -> {
22            for(int i=0; i<batchSize; i++) {
23           conn.set(("key" + i).getBytes(),String.valueOf(i)
24               .getBytes());
25           conn.get(("key" + i).getBytes());
26          }
27          return null;
28          });
29          long e = System.currentTimeMillis();
30          System.out.println("time cost by pipeline :" + (e-s) + " ms.");
31       }
32    }
```

其中，文件 5-17 中的配置类 RedisConfig（第 2 行）的内容见文件 3-6，此处不再赘述。在获取 StringRedisTemplate 的实例后（第 5、6 行），执行了 10 000 次读写操作（非流水线作业），并在第 17、18 行输出这 10 000 次操作所耗费的时间。

随后，在第 21 行调用 RedisTemplate 类的 executePipelined()方法实现流水线作业。该方法的用法与表 3-7 中的 execute()方法用法类似，可以接收的参数为 RedisCallback 对象或 SessionCallback 对象（第 21 行）。第 22～26 行执行 10 000 次读写操作（流水线作业）。由于操作的是字符串，RedisTemplate 会自动用值序列化器在返回响应数据前反序列化所有结果。从 RedisCallback 对象返回的值必须为 null（第 27 行），因为该值会被丢弃，从而支持返回流水线命令的执行结果。

上述测试代码的运行结果如图 5-12 所示。可见非流水线方式执行了多次 Redis 连接的建立与关闭，程序运行耗时 3137ms，而流水线方式只执行了一次连接建立与关闭，程序运行耗时 264ms。

```
16:43:11.202 [main] DEBUG org.springfr
16:43:11.202 [main] DEBUG org.springfr
16:43:11.202 [main] DEBUG org.springfr
16:43:11.202 [main] DEBUG org.springfr
16:43:11.202 [main] DEBUG org.springfr
time cost by Non-pipeline :3137 ms.
16:43:11.210 [main] DEBUG org.springfr
16:43:11.466 [main] DEBUG org.springfr
time cost by pipeline :264 ms.
```

图 5-12　文件 5-17 测试代码运行结果

目前，大多数的 Redis 客户端已支持流水线操作。流水线操作可以在一定程度上提升程序性能，但是每次由流水线组装的命令个数不能没有限制。一次组装在流水线中的数据量过大，一方面会增加客户端的等待时间，另一方面会造成一定程度的网络拥塞。可以将一个包含大量命令的流水线拆分成多个较小的流水线来完成。同时，流水线只能操作一个 Redis 实例。即使在分布式 Redis 场景中，也可以作为批量操作的重要优化手段。

5.4　事务与 Lua

Redis 事务是一组命令的集合，事务支持一次执行多条命令。Redis 事务具有以下特征：事务中的所有命令都会按顺序执行，其他客户端提交的命令不会插入事务执行的命令序列中，并且事务中所有命令都会被序列化；这就保证了事务中的一组命令成为一个独立

的执行单元。即 Redis 事务就是一次性、顺序性、排他性地执行一个队列中的一组命令。简单地说，事务表示一组动作要么全部执行，要么全部不执行。例如，在社交网站上用户 A 关注了用户 B，那么需要在用户 A 的关注列表中加入用户 B，并且在用户 B 的关注者列表中添加用户 A，这两个动作要么全部执行，要么全部不执行，否则会出现数据不一致的情况。

Redis 事务没有隔离级别的概念，也不保证事务的原子性。为保证多条命令组合的原子性，Redis 借助 Lua 脚本来解决这个问题。本节首先介绍 Redis 事务的使用方法和它的局限性，之后介绍 Lua 脚本的基本使用方法，以及如何将 Redis 和 Lua 脚本集成，最后给出 Spring Data Redis 操作 Lua 的相关例子。

5.4.1 Redis 事务

Redis 提供了简单的事务功能，表 5-4 列出了 Redis 事务的常用命令及描述。

表 5-4 Redis 事务的常用命令及描述

命令	描　述
DISCARD	取消事务，取消执行事务块内的所有命令
EXEC	执行所有事务块内的所有命令
MULTI	标记一个事务块的开始
UNWATCH	取消 WATCH 命令对所有键的监视
WATCH	监视所有的键，如果在事务执行之前这些键被其他命令所改动，那么事务将被打断

利用 Redis 处理事务时，将一组需要一起执行的命令放到 MULTI 和 EXEC 两个命令之间。这两个命令之间的命令是按顺序执行的原子操作。例如，下面的操作实现了上述的用户关注问题。

```
redis > MULTI
OK
redis > SADD user:a:follow user:b
QUEUED
redis > SADD user:b:fans user:a
QUEUED
```

此时，可以看到 SADD 命令的返回结果是 QUEUED，代表命令并没有真正执行，而是暂时保存在 Redis 中。此时，如果另一个客户端执行命令 SISMEMBER user:a:follow user:b 则返回结果应该是 0。

```
redis > SISMEMBER user:a:follow user:b
(integer) 0
```

只有当 EXEC 命令执行后，用户 A 关注用户 B 的行为才算完成。如下命令序列所示。

```
redis > MULTI
OK
redis > SADD user:a:follow user:b
QUEUED
redis > SADD user:b:fans user:a
QUEUED
redis > EXEC
1) (integer) 1
2) (integer) 1
```

```
redis > SISMEMBER user:a:follow user:b
(integer) 1
```

如果要停止事务的执行，要使用 DISCARD 命令代替 EXEC 命令。

```
redis > DISCARD
OK
redis > SISMEMBER user:a:follow user:b
(integer) 0
```

有些应用场景需要在事务执行前，确保事务中的键没有被其他客户端修改过才能执行事务，否则不执行(类似乐观锁)。Redis 提供了 WATCH 命令来解决这类问题，表 5-5 展示了事务中 WATCH 命令执行时序。

表 5-5　事务中 WATCH 命令执行时序

时　间　点	客户端 1	客户端 2
T1	SET num 10	
T2		WATCH num
T3	WATCH num	
T4	MULTI	
T5		MULTI
T6		INCRBY num 5
T7	INCRBY num 8	
T8		EXEC (success)
T9	EXEC (nil)	

如表 5-5 所示，客户端 1 和客户端 2 均在执行 MULTI 命令之前执行了 WATCH 命令，用于监控 num 的值。T6 时刻，客户端 2 修改了 num 的值。客户端 1 在 T7 时刻也修改了 num 的值。但在 T8 时刻，客户端 2 执行 EXEC 命令，完成对 num 值的修改。而 T9 时刻，客户端 1 执行 EXEC 命令，此时，客户端 1 对 num 值的修改无效，事务回滚(EXEC 命令执行结果为 nil)。WATCH 可以同时监控多个键，在监控期间只要有一个键被其他客户端修改，则整个事务回滚。因此，这番操作后，num 的值应为 15。并且，WATCH 命令的生命周期只和事务关联，一个事务执行完毕，相应的 WATCH 命令的生命周期就会结束。

Redis 对事务的支持是很弱的，也无法实现命令之间的逻辑关系计算。5.4.2 节要介绍的 Lua 脚本同样可以实现事务的相关功能，但功能要强大很多。

5.4.2　Lua 脚本

Lua 是一种轻量级的脚本语言，是巴西一所大学的研究小组发明的，其设计目的是作为嵌入式应用程序移植到其他应用。结合 Redis 和 Lua 脚本语言的特性，如果在 Redis 应用中遇到如下需求，就可以引入 Lua 脚本：

(1) 重复执行相同类型的命令。如要在内存中缓存数字 1～1000。

(2) 在高并发场景下减少网络开销。如要向 Redis 服务器发送多条命令，就可以把这些命令放入 Lua 脚本里，这样通过调用脚本只耗费一次网络开销就能执行多条 Redis 命令。

(3) Redis 会把 Lua 脚本作为一个整体来执行，天然具有原子性。所以，在一些需要原子操作的场景里，Lua 脚本非常适合。

作为编程语言，Lua 本身就包含了变量、操作符、循环等语法。本节只给出与 Redis 应用相关的 Lua 语法。

1. 数据类型及逻辑处理

Lua 提供了如下几种数据类型：booleans（布尔）、numbers（数值）、strings（字符串）、tables（表格）。下面结合例子对 Lua 的基本数据类型和逻辑处理进行说明。

（1）字符串。

下面定义一个字符串类型的数据：

```
local strings s = "world"
```

其中，local 代表 s 是一个局部变量，如果没有 local，则代表全局变量。print()函数可以输出变量的值，例如下面代码将输出 world，其中--是 Lua 的注释。

```
--输出 world
print(s)
```

（2）数组。

在 Lua 中，如果要使用类似数组的功能，可以用 tables 类型。下面代码定义了一个 tables 类型的变量 myArr，但和大多数编程语言不同的是，Lua 数组的下标从 1 开始：

```
local tables myArr = {"redis", "jedis", true, 80.0}
--输出 true
print(myArr[3])
```

如果要遍历整个数组，可以使用 for 和 while，这些关键字和许多编程语言是一致的。

① for。

下面代码会计算 1～100 的和，循环体以 end 作为结束符：

```
local int sum = 0
for i = 1, 100
do
    sum = sum + i
end
--输出结果为 5050
print(sum)
```

要遍历 myArr 数组，首先要知道 tables 的长度，只需要在变量前加一个＃即可：

```
for i = 1, #myArr
do
    print(myArr[i])
end
```

除此之外，Lua 还提供了内置函数 ipairs()，使用 for index，value in ipairs(tables)可以遍历所有的索引下标和值：

```
for index, value in ipairs(myArr)
  do
      print(index)
```

```
4          print(value)
5    end
```

② while。

下面的代码同样会计算 1～100 的和。

```
1    local int sum = 0
2    local i = 0
3    while i <= 100
4    do
5        sum = sum + i
6        i = i+1
7    end
8    --结果为 5050
9    print(sum)
```

③ if else。

要确定数组 myArr 中是否包含 jedis,可以使用下面的 Lua 代码:

```
 1    local tables myArr = {"redis","jedis",true,80.9}
 2    for i=1, #myArr
 3    do
 4        if myArr[i] == "jedis"
 5        then
 6            print("true")
 7            break
 8        else
 9            --do nothing
10        end
11    end
```

(3) 哈希。

如果要使用类似哈希的功能,同样可以使用 tables 类型,例如下面代码定义了一个 tables,每个元素包含了键和值,其中运算符..的作用是连接两个字符串。

```
1    local tables user_1 = {age=28, name="Tome"}
2    print("user_1 ages is "..user_1["age"])
```

如果要遍历 user_1,可以使用 Lua 的内置函数 pairs():

```
1    for key,value in pairs(user_1)
2      do
3          print(key..value)
4      end
```

2. 函数

在 Lua 中,函数以 function 开头,以 end 结尾,funcName 是函数名,中间部分是函数体:

```
function funcName()
    ...
end
```

例如，自定义函数 contact()可以将两个字符串拼接：

```
function contact(str1,str2)
    return str1..str2
end
```

5.4.3 应用案例

Redis 结合 Lua 脚本可以确保 Redis 命令执行时的顺序性和原子性，所以在开发一些高并发场景的应用时，可以采用两者结合的方式实现 Redis 事务，下面给出相关的范例。

1. 模拟抽奖

模拟一个抽奖场景，从奖池中进行随机抽奖。要求如下：

(1) 中奖的人只能从奖池中抽取。

(2) 每个人只能中奖一次。

(3) 中奖总人数不能超过设置的奖项数。

(4) 生成中奖名单。

可以使用 Redis 中的集合(SET)数据类型实现奖池。因为集合可以保证存放的元素无重复且无序，这样可以满足抽奖的随机性和奖池候选人的唯一性。同时集合还提供了很多操作来满足抽奖的需要。

第一步，设置奖池。向集合中添加参与抽奖的元素(人员)。代码如下：

```
public void preparation(){
   for(int i = 1;i < 8;i++)
      template.opsForSet().add("lottery","u" + i);
}
```

如上述代码所示，将集合 lottery 设置为奖池，并向奖池中添加 u1～u7 共 7 个元素。

第二步，编写抽奖脚本。从 lottery 集合中抽出一定数量的元素放入另外一个集合 chosen 中，遍历 chosen 集合即可得到中奖人员名单。这一功能需要执行多条 Redis 命令，可以通过编写 Lua 脚本实现。

可以在 Maven 项目的 src/main/resources 目录下创建名为 META-INF/scripts 的子目录，并将 lottery.lua 脚本文件存放在该子目录中，代码如文件 5-18 所示。

【文件 5-18】 lottery.lua

```
-- 奖池的 key
local lottery_key = KEYS[1]
-- 中奖名单的 key
local chosen_key = KEYS[2]
-- 预定抽奖的人数
local lottery_count = ARGV[1]

-- 如果预定抽奖的人数大于 0 则开始抽奖
if tonumber(lottery_count) > 0 then
    -- 奖池中抽奖,返回的是被抽中的人组成的数组
    local chosen_list = redis.call('SRANDMEMBER', lottery_key,
```

```
12            lottery_count);
13        -- 将抽中的人添加到中奖名单中,返回中奖的人数
14        if chosen_list then
15            return redis.call('SADD', chosen_key, unpack(chosen_list))
16        else
17            return 0
18        end
19    else
20        return 0
21    end
```

其中,KEYS 和 ARGV(第 2、4、6 行)是两个全局变量,它们可以将 Java 参数应用于 Lua 脚本。KEYS 表(数组)预先填充了在脚本执行之前提供给脚本的所有键名参数,而 ARGV 表(数组)的用途类似,但用于常规参数。

Lua 脚本与 Redis 交互可以通过调用 redis.call()函数或 redis.pcall()函数实现。这两个函数都可以执行格式正确的 Redis 命令(包括提供的参数)。它们之间的区别在于处理运行时错误(例如语法错误)的方式。执行 Redis 命令引发错误时,redis.call()直接返回给执行它的客户端。相反,redis.pcall()将返回到脚本的上下文,而不进行进一步处理。

例如,可以在 Redis 客户端利用 EVAL 命令执行 Lua 脚本:

```
redis > EVAL "return redis.call('SET',KEYS[1], ARGV[1])" 1 foo bar
OK
redis > get foo
"bar"
```

文件 5-18 的第 15 行的 unpack()函数接受一个数组作为参数,并返回数组的所有元素(数组下标默认从 1 开始)。在 Lua 5.1 中,unpack()是全局函数,可以直接使用,但是在 Lua 5.2 中,unpack()被移到 table.unpack,所以在 Lua 5.2 以后要用 table.unpack()替代 unpack()。

文件 5-18 的 Lua 脚本的返回值类型为 long。脚本的返回类型应为 long、boolean、list 或反序列化值类型之一,也可以为空。

第三步,调用抽奖脚本。Redis 2.6 及更高版本支持通过 EVAL 和 EVALSHA 命令运行 Lua 脚本。每当执行 EVAL 命令时,还会在请求中包含脚本的源代码。重复利用 EVAL 命令执行同一组参数化脚本时,既浪费网络带宽,又有一些 Redis 开销。为了节省网络和计算资源,Redis 为 Lua 脚本提供了缓存机制。

可以通过调用 RedisTemplate 的 execute()方法运行 Lua 脚本(该方法的全部重载形式可见表 3-7)。本案例选择的重载形式如下:

```
public < T > T execute(RedisScript < T > script, List < K > keys, Object... args) {
    return scriptExecutor.execute(script, keys, args);
}
```

其中:

(1) 参数 script 用来加载 Lua 脚本。

(2) 参数 keys 对应 Lua 脚本中的 KEYS,用来传入 Redis 的 KEY,在 Lua 脚本中可以通过 KEYS[索引]来取值,例如取第一个值 KEYS[1]。

（3）参数 args 用来向 Lua 脚本传递其他的参数，在 Lua 脚本中可以通过 ARGV[索引]来取值。

调用 Lua 脚本的 Java 代码如下：

```
public void drawLotteries(RedisScript<Long> script, List<String> keys,
    Object... args) {
    template.execute(script, keys, args);
}
```

第四步，修改配置类。可修改文件 3-6，增加一个 RedisScript<Long> Bean 的定义，代码如下：

```
@Bean
public RedisScript<Long> script() {
   RedisScript<Long> redisScript = RedisScript.of(new
    ClassPathResource("META-INF/scripts/lottery.lua"), Long.class);
   return redisScript;
}
```

第五步，编写测试代码。

```
@Test
public void testLottery() {
   raffleDemo.preparation();
   raffleDemo.drawLotteries(script, Arrays.asList("lottery","chosen")
       , "3");
   System.out.println(template.opsForSet().members("chosen"));
}
```

```
[u2, u6, u1]
```

图 5-13 抽奖案例运行结果

如测试代码所示，首先执行初始化工作，调用第一步定义的 preparation()方法设置奖池，随后从奖池 lottery 中随机抽取 3 个元素，并将这 3 个元素作为中奖者存入 chosen 集合。最后遍历 chosen 集合输出获奖者名单。运行测试代码后，控制台输出如图 5-13 所示。

2. 限流

限流是指某应用程序需要限制指定 IP 的主机在单位时间内的访问次数。互联网应用往往是高并发场景，其特性就是流量可能瞬时激增。此时，如果没有流量管控机制，就很容易导致系统崩溃。

限流是用来保证系统稳定性的常用手段，当系统遭遇瞬时流量激增时，很可能会因系统资源耗尽导致宕机。限流可以把超出系统承受能力的请求直接拒绝掉，允许其他请求正常访问，从而保证在系统可承受的范围内提供正常服务。例如，在某高并发场景中，对会员查询模块的限流需求是在 10 秒内最多允许有 1000 个请求。常用的限流算法有漏桶算法、令牌桶算法、计数法等。计数法限流的做法是，提供服务的模块会统计服务请求模块在单位时间内发送请求的次数，如果已达到限流标准，则不予提供服务。

下面给出用 Lua 脚本实现的基于计数法限流的案例。

第一步，创建执行限流的 Lua 脚本，代码如文件 5-19 所示。

【文件 5-19】　limiter.lua

```
local key = KEYS[1]
local limit = tonumber(KEYS[2])
local length = tonumber(KEYS[3])

local current = redis.call('GET', key)
if current == false then
  redis.call('SET', key,1)
  redis.call('EXPIRE',key,length)
  return '1'
else
  local num_current = tonumber(current)
  if num_current + 1 > limit then
     return '0'
  else
     redis.call('INCRBY',key,1)
     return '1'
  end
end
```

如文件 5-19 所示，该限流脚本有 3 个参数（第 1～3 行）：参数 key 接收待限流的对象，并将该对象的名字作为键保存到 Redis 中；参数 limit 表示限流的次数，即服务请求模块在单位时间内发送请求的次数如果不超过 limit，则其发出的请求可被正常处理；参数 length 表示限流的时长，即在 length 时间单位内只允许服务请求模块最多发出 limit 次请求。首先利用 Redis 的 GET 命令获取待限流的对象的访问次数（第 5 行）。如果获取不到（第 6～9 行），则认为当前对象第一次发送请求，在记录当前对象发送请求的次数后（第 7 行），设置当前请求的计数键在 Redis 中的过期时间（第 8 行），并返回字符串“1”（第 9 行），用以表示当前请求可以被正常处理。如果可以获取到待限流对象的访问次数（第 10～18 行），则首先调用 Lua 的 tonumber()函数将字符串形式的计数值转为数值型（第 11 行），随后判断当前计数值是否达到限流标准。如果达到限流标准（第 12、13 行），则返回字符串“0”，用以表示当前请求要被限流，不予提供服务；如果没达到限流标准（第 14～17 行），将当前计数值增 1（第 15 行），用以表示待限流对象在 length 时间内又一次发送请求，并返回字符串“1”。

第二步，以多线程的形式模拟多个并发请求，每个线程在规定时间内发送 5 次请求，以此查看限流效果，代码如文件 5-20 所示。

【文件 5-20】　LimiterDemo.java

```
public class LimiterDemo extends Thread {
   private StringRedisTemplate template;
   private RedisScript<String> script;
   public LimiterDemo(StringRedisTemplate template,
         RedisScript<String> script){
      this.template = template;
      this.script = script;
   }

   public void run(){
      List<String> keyList = Arrays.asList(
```

```
12                Thread.currentThread().getName(),"3","10");
13          for(int i = 0;i < 5;i++){
14             String res = template.execute(script, keyList, "none");
15             if (res.equals("1"))
16                System.out.println(Thread.currentThread().getName() +
17                  " can visit");
18             else
19                System.out.println(Thread.currentThread().getName() +
20                  " sorry! be slow");
21          }
22       }
23    }
```

如文件 5-20 所示，以当前线程的名字作为键，将其发送请求的计数值存入 Redis。同时，指定每个线程在 10 秒内可以发送 3 次请求，如果在 10 秒的限流期内发送请求的次数超过 3 次，则该线程将被限流。Redis 缓存键（线程名）的时长设置为 10 秒（第 11、12 行）。而每个线程在程序运行期间将发送 5 个请求（第 13 行）。对照之前的分析可知，每个线程都会被执行两次限流。

第三步，为测试用多线程模拟多客户端发送请求，并被限流的效果，还需要编写测试代码，如文件 5-21 所示。

【文件 5-21】 TestLimiterDemo.java

```
1     @RunWith(SpringJUnit4ClassRunner.class)
2     @ContextConfiguration(classes = RedisConfig.class)
3     public class TestLimiterDemo {
4        @Autowired
5        private StringRedisTemplate template;
6        @Autowired
7        private RedisScript < String > script;
8
9        @Test
10       public void testLimiter() {
11          try {
12             for(int i = 0;i < 5;i++) {
13                Thread.sleep(200);
14                new LimiterDemo(template, script).start();
15             }
16              //主线程挂起，等待所有线程运行完毕
17              Thread.sleep(6000);
18          } catch (InterruptedException e) {
19                e.printStackTrace();
20          }
21       }
22    }
```

如文件 5-21 所示，第 2 行引入配置类 RedisConfig，增加一个 RedisScript < String > Bean 的定义，代码可参考模拟抽奖案例的第四步，此处不再赘述。随后创建 5 个线程（第 12～15 行），模拟 5 个客户端给服务模块发送请求，测试限流效果，代码运行结果如图 5-14 所示。

```
Thread-1 can visit
Thread-0 can visit
Thread-1 can visit
Thread-0 can visit
Thread-1 can visit
Thread-0 can visit
Thread-1 sorry! be slow
Thread-0 sorry! be slow
Thread-1 sorry! be slow
Thread-0 sorry! be slow
Thread-2 can visit
Thread-2 can visit
Thread-2 can visit
Thread-2 sorry! be slow
Thread-2 sorry! be slow
Thread-3 can visit
Thread-3 can visit
Thread-3 can visit
Thread-3 sorry! be slow
Thread-3 sorry! be slow
Thread-4 can visit
Thread-4 can visit
Thread-4 can visit
```

图 5-14　限流案例代码运行结果

5.5　Geo

Redis 3.2 及以后的版本提供了 Geo(Geospatial,地理空间数据)类型,该类型可以存储地理位置信息,用来实现诸如附近位置、摇一摇等这类依赖于地理位置信息的功能。Redis Geo 并不是一种新的数据结构,而是基于有序集合实现的。有序集合保存的数据形式是键-分数(Key-Score),即一个元素对应一个分数,默认是根据分数排序的,且可以按范围查询。但是经纬度是一个数据对,如(106.053 218,32.437 117),Redis 是如何将经纬度转换为分数值的呢? 转换为分数值之后,又是如何保证分数值相邻的元素距离也相近的呢? 这一切依赖于 GeoHash 编码。该编码可将经纬度转换为字符串,而且距离越近字符串的相似程度越高。GeoHash 编码可通过 Redis 提供的 Geo 相关命令得到。Geo 相关命令包括添加、计算位置之间的距离,根据中心点坐标和距离范围来查询地理位置集合等,如表 5-6 所示。

表 5-6　Geo 相关命令

命　　令	说　　明
GEOADD	添加地理位置的坐标
GEOPOS	获取地理位置的坐标
GEODIST	计算两个位置之间的距离
GEORADIUS	根据用户给定的经纬度坐标来获取指定范围内的地理位置集合
GEORADIUSBYMEMBER	根据存储在位置集合里面的某个地点获取指定范围内的地理位置集合
GEOHASH	返回有效的 GeoHash 字符串,该字符串表示一个或多个元素在表示地理空间索引的排序集中的位置(其中元素是使用 GEOADD 添加的)

操作 Redis Geo 的方法与操作 Redis 字符串的方法类似,要调用 RedisTemplate 的 opsForGeo()方法创建 GeoOperations<K,M>子接口对象,再调用 GeoOperations<K,M>子接口的相关方法。下面介绍 GeoOperations<K,M>子接口中的常用方法。

（1）方法原型：@Nullable Long add(K key,Point point,M member)；功能：将具有给定成员名称的点(地理坐标值)添加到键 key；对应的 Redis 命令：GEOADD。其中,Point 类(org. springframework. data. geo. Point)代表某点的地理坐标值；泛型 M 为 GeoOperations 接口的类型参数。

（2）方法原型：@Nullable Distance distance(K key,M member1,M member2,Metric metric)；功能：获取成员 member1 和成员 member2 之间的距离；对应的 Redis 命令：GEODIST。其中,Metric 接口(org. springframework. data. geo. Metric)是一个用于米制的度量接口。

（3）方法原型：@Nullable List < Point > position(K key,M...members)；功能：获取一个或多个成员的位置的点(地理坐标值)表示；对应的 Redis 命令：GEOPOS。

（4）方法原型：@Nullable GeoResults < RedisGeoCommands. GeoLocation < M >> radius(K key,M member,Distance distance)；功能：获取给定圆边界内的成员；对应的 Redis 命令：GEORADIUS。

（5）方法原型：@Nullable GeoResults < RedisGeoCommands. GeoLocation < M >> radius(K key,M member,Distance distance)；功能：基于米制单位,获取由成员坐标和给定半径定义的圆内的成员；对应的 Redis 命令：GEORADIUSBYMEMBER。

（6）方法原型：@Nullable List < String > hash(K key,M...members)；功能：获取一个或多个成员的位置的 GeoHash 表示；对应的 Redis 命令：GEOHASH。

案例：结合表 5-7 给出的城市的地理位置信息,完成如下要求。

表 5-7　5 个城市的地理位置信息

城　　市	经　　度	纬　　度	成　员　名
北京	116.28	39.55	BeiJing
天津	117.12	39.08	TianJin
石家庄	114.29	38.02	ShiJiaZhuang
唐山	118.01	39.38	TangShan
保定	115.29	38.51	BaoDing

（1）以 city:location 为键,将表 5-7 中的城市位置信息加入 Redis。可以调用 GeoOperations < K,M >子接口的 add()方法,测试代码如文件 5-22 所示。

【文件 5-22】　TestGeoDemo. java

```
1    @RunWith(SpringJUnit4ClassRunner.class)
2    @ContextConfiguration(classes = RedisConfig.class)
3    public class TestGeoDemo {
4       @Autowired
5       private RedisTemplate template;
6       private double[][] position = {{116.28,39.55},{117.12,39.08},
7             {114.29,38.02},{118.01,39.38},{115.29,38.51}};
8       private String[] member =
9         {"BeiJing","TianJin","ShiJiaZhuang","TangShan","BaoDing"};
10
11      @Test
12      public void testGeoAdd(){
```

```
13          Long num = 0L;
14          for(int i = 0;i < member.length; i++){
15              Point point = new Point(position[i][0],position[i][1]);
16              num = template.opsForGeo()
17                .add("city:location",point,member[i]);
18          }
19          Assert.assertEquals(1,num.intValue());
20       }
21    }
```

(2) 获取天津的位置信息并输出。可调用 GeoOperations<K,M>子接口的 position()方法,在文件 5-22 中添加一个测试用例,代码如下。

```
1    @Test
2    public void testGeoPos(){
3       List<Point> list = new ArrayList<Point>();
4       list = template.opsForGeo().position("city:location",member[1]);
5       list.forEach(System.out::println);
6    }
```

该测试代码的运行结果如图 5-15 所示。

```
Point [x=117.120000, y=39.080000]
```

图 5-15　(2)测试代码运行结果

(3) 计算北京和天津间的直线距离,单位:km。可调用 GeoOperations<K,M>子接口的 distance()方法,并设置计算结果的单位为千米。在文件 5-22 中添加一个测试用例,代码如下。

```
1    @Test
2    public void testGeoDist() {
3       Distance distance = template.opsForGeo().distance("city:location",
4        member[0],member[1],RedisGeoCommands.DistanceUnit.KILOMETERS);
5       System.out.println(distance);
6    }
```

其中,第 4 行的参数 RedisGeoCommands.DistanceUnit 是一个静态枚举,它已实现 Metric 接口。从 RedisGeoCommands.DistanceUnit 选择常量 KILOMETERS 用来指定计算结果的单位为 km。完成要求(3)也可以使用 distance()方法的另一种重载形式,即省略第 4 行中的第 3 个参数,这样计算出的两个成员间的距离单位为 m。

(4) 找出距离北京 150km 以内的城市。可调用 GeoOperations<K,M>子接口提供的 radius()方法。在文件 5-22 中添加一个测试用例,代码如下。

```
1    @Test
2    public void testGeoRadius() {
3       Distance distance = new Distance(150, Metrics.KILOMETERS);
4       GeoResults<RedisGeoCommands.GeoLocation<String>> results =
5        template.opsForGeo().radius("city:location",member[0],distance);
6       System.out.println(results);
7    }
```

此测试代码的运行结果如图 5-16 所示。

当然,也可使用 radius()方法的另一种重载形式:@Nullable。

```
GeoResults: [averageDistance: 0.0 KILOMETERS, results: GeoResult
 [content: RedisGeoCommands.GeoLocation(name=BeiJing, point=null),
 distance: 0.0 KILOMETERS, ],GeoResult [content: RedisGeoCommands
 .GeoLocation(name=TianJin, point=null), distance: 0.0 KILOMETERS,
 ],GeoResult [content: RedisGeoCommands.GeoLocation(name=TangShan,
 point=null), distance: 0.0 KILOMETERS, ],GeoResult [content:
 RedisGeoCommands.GeoLocation(name=BaoDing, point=null), distance:
 0.0 KILOMETERS, ]]
```

图 5-16 （4）测试代码运行结果

GeoResults < RedisGeoCommands. GeoLocation < M >> radius（K key，Circle within）完成此要求。读者可自行编写代码。

（5）利用键绑定子接口操作键为 bound-geo 的 Geo 类型的数据。分别执行添加、测距操作。例如，测量 BeiJing 和 BaoDing 两个城市间的距离，单位为 km，部分代码如下。

```
1    @Test
2    public void testGeoBoundOperations() {
3       BoundGeoOperations < String,String > ops =
4          template.boundGeoOps("bound - geo");
5       ops.add(new Point(116.28,39.55),"BeiJing");
6       ops.add(new Point(115.29,38.51),"BaoDing");
7       Distance dis = ops.distance("BeiJing","BaoDing",
8          RedisGeoCommands.DistanceUnit.KILOMETERS);
9       if(dis != null)
10         System.out.println("The distance between BeiJing and BaoDing
11         is " + dis.getValue() + dis.getUnit());
12   }
```

此测试代码的运行结果如图 5-17 所示。

```
The distance between BeiJing and BaoDing is 143.8646km
```

图 5-17 （5）测试代码运行结果

（6）获取北京和天津的 GeoHash 值。可调用 GeoOperations < K，M >子接口的 hash() 方法。在文件 5-22 中添加一个测试用例，代码如下。

```
1    @Test
2    public void testGeoHash() {
3       List < String > list = template.opsForGeo()
4          .hash("city:location",member[0],member[1]);
5       list.forEach(System.out::println);
6    }
```

```
wx48ypbe2q0
wwgq34k1tb0
```

图 5-18 （6）测试代码运行结果

Redis 使用 GeoHash 将二维经纬度转换为一维字符串。上述测试代码的运行结果如图 5-18 所示。

由上述案例可知，GeoHash 具有如下特点：

（1）Geo 的基础数据类型为有序集合，Redis 将所有地理位置信息的 GeoHash 存放在有序集合中。

（2）字符串越长，表示的位置越精确。例如，GeoHash 的长度为 9 时，精度在 2m 左右。

（3）两个字符串越相似，它们之间的距离越近。Redis 利用字符串前缀匹配算法实现相

关的命令。

(4) GeoHash 编码和经纬度是可以相互转换的。

5.6 小结

本章介绍了 Redis 的一些基础(非缓存)应用。如 Redis 可以实现消息的发布-订阅,可以作为简单的消息队列使用。此外,Redis 还提供了以日志形式存储消息的数据结构——流。这样,就支持消费者以同步或异步方式获取消息队列中的消息。为了提高命令的执行效率,Redis 提供了流水线机制。它允许将多条命令集中起来,一次性发给 Redis 服务器执行,而后一次性返回命令执行结果。作为 NoSQL 数据库,Redis 只提供了简单的事务管理功能,结合了 Lua 脚本后,Redis 才拥有强大的事务管理能力。最后,本章介绍了 Redis 提供的基于有序集合的新的数据类型——Geo。利用 Geo 类型,可以执行地理位置相关的计算。

对于以上 5 方面的基础应用,Redis 都提供了相关命令。读者可查找 Redis 官方提供的命令手册学习更多相关命令的使用方法。