第 3 章

Spring Redis Template

视频讲解

视频讲解

RedisTemplate 类可以用于简化 Redis 操作，是 Spring Data Redis 对 Redis 支持的核心类。它可以负责序列化和连接管理，用户无须关心此类细节。此外，该类提供了 Redis 操作视图(依据 Redis 命令分组)，这些视图提供了丰富的通用接口，可以以程序的方式执行 Redis 命令。RedisTemplate 一旦配置好就是线程安全的。

本章将围绕 RedisTemplate 类的连接建立、Redis 操作视图、键绑定、序列化等内容展开。

3.1 Java Redis 客户端

为了能够在应用程序中使用 Redis，通过 Redis 命令操作是不现实的。Redis 提供了适配不同编程语言的客户端实现通过编程操作 Redis。Java 有很多优秀的 Redis 客户端，如 Lettuce、Jedis、Redisson 和 JRedis 等。表 3-1 列举了常用的 Java Redis 客户端的特性。

表 3-1 常用的 Java Redis 客户端的特性

客户端	框架整合	介　　绍
Lettuce	Spring Data Redis	Lettuce 是基于 Netty 实现的，支持同步、异步和响应编程方式并且是线程安全的，支持 Redis 的哨兵模式、集群模式和流水线
Jedis	Spring Data Redis	以 Redis 命令作为方法名称，学习成本低，简单实用。Jedis 实例不是线程安全的，多线程环境下需要基于连接池来使用
Redisson	/	Redisson 是分布式 Redis 客户端，底层使用 Netty 框架，支持 Redis 的哨兵模式、主从模式和单节点模式

使用 Java 操作 Redis 最常用的是使用 Jedis 客户端。如果在项目中使用了 Jedis，但是后来决定弃用 Jedis 改用其他的 Redis 客户端就比较麻烦了。因为不同的 Java Redis 客户端是无法兼容的。Spring Data Redis 是 Spring Data 模块的一部分，专门用来支持在 Spring 管理的项目中对 Redis 的操作。Spring Data Redis 提供了 Redis 的 Java 客户端的抽象，在开发中可以忽略由于切换 Redis 客户端所带来的影响，而且它本身就属于 Spring 的一部分，比起单纯的使用 Jedis 更加稳定，管理起来更加自动化。

使用 Spring Data Redis 的首要任务之一是通过 Spring 容器连接到 Redis。为此，需要创建一个 Java 连接器。无论开发者选择哪个 Java Redis 客户端，只需要使用一组 Spring Data Redis API(在所有连接器中表现一致)。即使用 org.springframework.data.redis.

connection 包中 RedisConnection 和 RedisConnectionFactory 接口。这两个接口用于处理和获取 Redis 的活动连接。

RedisConnection 接口为应用程序与 Redis 通信提供了核心组件。因为它处理与 Redis 后端的通信。它还自动将底层连接库异常转换为与 Spring 一致的 DAO 异常层次结构，这样就可以在不更改任何代码的情况下切换连接器，因为操作语义保持不变。

RedisConnection 对象是通过 RedisConnectionFactory（工厂）创建的。此外，工厂充当 PersistenceExceptionTranslator 对象。这意味着一旦声明，PersistenceExceptionTranslator 对象就允许开发者进行透明的异常转换——例如，通过使用@Repository 注解和 AOP 进行异常转换。使用 RedisConnectionFactory 的最简单的方式就是通过 Spring 容器配置一个合适的连接器并将连接器注入给需要使用它的类。

Spring Data Redis 主要有以下特性：

（1）提供了一个可以跨越多个客户端（如 Jedis、Lettuce）的底层抽象连接包。

（2）针对数据的序列化和反序列化，提供了多种方案供开发者选择。

（3）提供了一个 RedisTemplate 类，该类对 Redis 的各种操作、异常转换和序列化都实现了高层封装。

（4）支持 Redis 的哨兵模式和集群模式。

3.2 创建 Redis 连接

针对不同的 Redis 客户端，使用程序连接 Redis 也有不同的方式。本节讲授如何利用 Lettuce、Jedis 客户端以及 Redis Template 类创建 Redis 连接。

3.2.1 Lettuce

Lettuce 是一个可扩展的线程安全 Redis 客户端，用于同步、异步和响应式使用。如果多个线程不使用阻塞和事务操作（如 BLPOP 和 MULTI/EXEC），则它们可能共享一个连接。Lettuce 是基于 Netty 构建的。Lettuce 支持 Redis 的高级功能，如哨兵（Sentinel）、集群（Cluster）、流水线（Pipelining）、自动重新连接和 Redis 数据模型。

要利用 Spring Data Redis 配置 Lettuce 连接器，首先在 Maven 项目中引入相关依赖，内容如下：

```
1   <!-- Spring 核心包 spring-context -->
2   <dependency>
3       <groupId>org.springframework</groupId>
4       <artifactId>spring-context</artifactId>
5       <version>5.3.18</version>
6   </dependency>
7   <dependency>
8       <groupId>org.springframework.data</groupId>
9       <artifactId>spring-data-redis</artifactId>
10      <version>2.7.1</version>
11  </dependency>
12  <!-- Lettuce -->
13  <dependency>
```

```
14        <groupId>io.lettuce</groupId>
15        <artifactId>lettuce-core</artifactId>
16        <version>6.1.8.RELEASE</version>
17    </dependency>
```

利用 Lettuce 配置 Redis 连接器有两种方案。第一,创建 Lettuce 连接工厂,通过连接工厂获取连接;第二,直接用 Lettuce 创建 Redis 连接。

1. Lettuce 连接工厂

可以创建一个名为 LettuceConfig 的类来配置 Lettuce 连接工厂,代码如文件 3-1 所示。

【文件 3-1】 LettuceConfig.java

```
1    @Configuration
2    public class LettuceConfig {
3      @Bean
4      public LettuceConnectionFactory redisConnectionFactory() {
5        return new LettuceConnectionFactory(
6             new RedisStandaloneConfiguration("127.0.0.1", 6379));
7      }
8    }
```

在创建了 LettuceConnectionFactory 实例后(第 5、6 行),可以调用该类的 getConnection()方法获取 Redis 连接对象(org.springframework.data.redis.connection.RedisConnection)。同时,可以编写测试类查看获取的连接是否有效。测试代码如文件 3-2 所示。

【文件 3-2】 TestLettuce.java

```
1    @RunWith(SpringJUnit4ClassRunner.class)
2    @ContextConfiguration(classes = LettuceConfig.class)
3    public class TestLettuce {
4        @Autowired
5        private LettuceConnectionFactory factory;
6        @Test
7        public void testLettuceConnectionFactory(){
8            RedisConnection conn = factory.getConnection();
9            //向 Redis 发送 PING 命令,并断言得到 PONG
10           Assert.assertEquals("PONG",conn.ping());
11       }
12   }
```

如文件 3-2 所示,第 10 行调用 RedisConnectionCommands(RedisConnection 接口的父接口)接口的 ping()方法,用于向 Redis 发送 PING 命令(见 1.4.2 节 PING 命令部分)。

2. Lettuce 直接连接 Redis

创建一个名为 LettuceConnection 的类,代码如文件 3-3 所示。

【文件 3-3】 LettuceConnection.java

```
1    public class LettuceConnection {
2          public static void main(String[] args) {
3             //步骤 1: 连接信息
4             RedisURI redisURI = RedisURI.builder()
5                .withHost("127.0.0.1")
```

```
6                 .withPort(6379)
7                 //.withPassword(new char[]{'a', 'b', 'c'})
8                 .withTimeout(Duration.ofSeconds(10))
9                 .build();
10            //步骤 2: 创建 Redis 客户端
11            RedisClient client = RedisClient.create(redisURI);
12            //步骤 3: 建立连接
13            StatefulRedisConnection<String, String> connection =
14                  client.connect();
15            //向 Redis 发送操作命令,相关代码略
16            //关闭连接
17            connection.close();
18            client.shutdown();
19        }
20    }
```

3.2.2 Jedis

Jedis 是利用 Java 操作 Redis 的工具，功能类似于 JDBC。Jedis 是 Redis 官方推荐的 Java 客户端开发包。Jedis 支持 Redis 命令、事务和流水线。配置 Jedis 连接器，首先要在项目中添加 Jedis 的依赖，内容如下：

```
1    <!-- Jedis -->
2    <dependency>
3        <groupId>redis.clients</groupId>
4        <artifactId>jedis</artifactId>
5        <version>3.8.0</version>
6    </dependency>
```

利用 Jedis 作为 Redis 连接器有三种实现方案：Jedis 连接工厂、Jedis 直接连接和 Jedis 连接池。

1. Jedis 连接工厂

创建一个 JedisConfig 类，代码如文件 3-4 所示。

【文件 3-4】 JedisConfig.java

```
1    @Configuration
2    public class JedisConfig {
3        @Bean
4        public JedisConnectionFactory redisConnectionFactory() {
5            RedisStandaloneConfiguration config =
6                new RedisStandaloneConfiguration("127.0.0.1", 6379);
7            return new JedisConnectionFactory(config);
8        }
9    }
```

如文件 3-4 所示，在获取了 JedisConnectionFactory 实例后（第 7 行），可以调用该类的 getConnection()方法获取 Redis 连接。

2. Jedis 直接连接

所谓直接连接是指 Jedis 在每次发送 Redis 操作命令前都会新建 TCP 连接，使用后再

断开连接，如图 3-1 所示。对于频繁访问 Redis 的场景这种方案，显然不是高效的使用方式。

利用 Jedis 直接创建 Redis 连接非常简单，只需创建 Jedis 的实例并指定相关参数即可。下述代码描述了图 3-1 的完整过程。

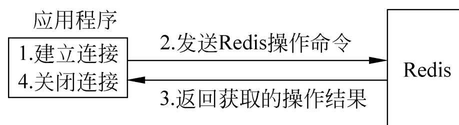

图 3-1 Jedis 直接连接 Redis

```
1    //1.建立连接
2    Jedis jedis = new Jedis("localhost",6379);
3    //2.发送 Redis 操作命令
4    jedis.set("username","zhangsan");
5    //3.返回获取的操作结果
6    String username = jedis.get("username");
7    //4.关闭连接
8    jedis.close();
```

3. Jedis 连接池

在生产环境中，从提升性能的角度考虑，一般使用连接池方式管理 Jedis 连接。具体做法是将所有的 Jedis 连接对象预先存放在连接池中，每次连接 Redis 时，只需要从连接池中借用活动连接，用后再将连接归还给连接池，如图 3-2 所示。

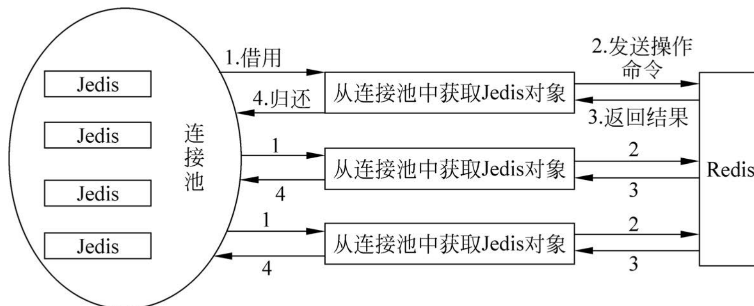

图 3-2 Jedis 连接池

客户端连接 Redis 使用的协议是 TCP。直接连接的方式每次都需要建立 TCP 连接，而连接池方式是可以预先创建好 Redis 连接，所以每次只需从连接池中借用即可。而借用和归还操作都是在本地进行的，只有少量的并发同步开销，这个开销远远小于新建 TCP 连接的开销。另外，直接连接方式无法限制 Jedis 对象的个数，在极端情况下会造成连接泄露，而连接池方式可以有效地保护和控制资源的使用。表 3-2 给出了两种方式各自的优势和劣势。

表 3-2 Jedis 直接连接方式和连接池方式对比

	优 点	缺 点
直接连接	简单方便，适用于少量长期连接的场景	(1) 存在每次新建、关闭 TCP 连接的开销； (2) 资源无法控制，极端情况会出现连接泄露； (3) Jedis 对象线程不安全
连接池	(1) 无须每次连接都生成 Jedis 对象，降低开销； (2) 保护和控制资源的使用	使用相对麻烦，尤其在资源管理上需要很多参数来保证，一旦规划不合理就会出现问题

创建 Redis 连接池代码如文件 3-5 所示。

【文件 3-5】 JedisConnectionPool.java

```
1    public class JedisConnectionPool {
2        private static final JedisPool jedisPool;
3        static{
4          GenericObjectPoolConfig jedisPoolConfig =
5                new GenericObjectPoolConfig();
6          //连接池中的最大连接数
7          jedisPoolConfig.setMaxTotal(8);
8          //连接池中的最大空闲连接数
9          jedisPoolConfig.setMaxIdle(8);
10        //连接池中的最少空闲连接数
11        jedisPoolConfig.setMinIdle(0);
12          //当连接资源耗尽后,调用者的最大等待时间,单位: 毫秒. -1 表示永不超时
13          jedisPoolConfig.setMaxWait(Duration.ofMillis(200));
14          jedisPool =
15              new JedisPool(jedisPoolConfig,"127.0.0.1",6379,1000);
16        }
17        //从连接池中借用连接
18        public static Jedis getJedis(){
19            return jedisPool.getResource();
20        }
21   }
```

3.2.3 RedisTemplate

目前,Jedis 客户端在编程实施方面存在以下一些不足。

(1) 连接管理无法自动化,连接池的设计缺少必要的容器支持。

(2) 数据操作需要关注序列化和反序列化,因为 Jedis 的客户端 API 接受的数据类型为 String 和 Byte,对结构化数据(JSON、XML、POJO 等)操作需要额外的支持。

(3) 事务操作为硬编码。

(4) 对于发布-订阅功能缺乏必要的设计模式支持,对于开发者而言需要关注的内容太多。

RedisTemplate 类(org.springframework.data.redis.core.RedisTemplate)是 Spring Data Redis 中对 Jedis API 的高度封装。Spring Data Redis 相对于 Jedis 来说可以方便地更换 Redis 的 Java 客户端。与 Jedis 相比,Spring Data Redis 进行了如下改进。

(1) 连接池自动管理,提供了一个高度封装的 RedisTemplate 类。

(2) 针对 Jedis 客户端大量的 API 进行了归类封装。遵循 Redis 命令参考(见 Redis 官方网站)中的分组,RedisTemplate 提供包含丰富的通用子接口的操作视图。这些视图可用于针对特定类型或特定键(通过键绑定接口)进行操作。

(3) 由容器封装并控制事务操作。

(4) 针对数据的序列化和反序列化提供了多种可选择的序列化器(RedisSerializer)。

(5) 基于设计模式和 JMS(Java Message Service,Java 消息服务)开发思路,将发布-订阅的编程接口进行了封装,使开发更加便捷。

(6) RedisTemplate 是线程安全的。

本书的后续案例将使用 RedisTemplate 来实现编程执行 Redis 操作。要获取 RedisTemplate 实例，首先要创建连接工厂对象，再借助连接工厂构建 RedisTemplate。如果操作的值是 String 类型，也可以使用 RedisTemplate 类的子类 StringRedisTemplate。由于程序中要频繁使用 RedisTemplate 对象，可以将其设置为 Spring 管理的 Bean，然后由 Spring 将该对象注入需要的地方。代码如文件 3-6 所示。

【文件 3-6】 RedisTemplateConfig. java

```
1    package com.example.redis.template;
2    //import 部分略
3    @Configuration
4    public class RedisTemplateConfig {
5        @Bean
6        publicRedisConnectionFactory redisConnectionFactory() {
7            RedisStandaloneConfiguration rsc =
8                new RedisStandaloneConfiguration();
9            rsc.setHostName("127.0.0.1");
10           rsc.setDatabase(0);
11           rsc.setPort(6379);
12           return new JedisConnectionFactory(rsc);
13       }
14       @Bean
15       public StringRedisTemplate redisTemplate(@Autowired
16       RedisConnectionFactory rcf){
17           StringRedisTemplate template = new StringRedisTemplate();
18           template.setConnectionFactory(rcf);
19           return template;
20       }
21   }
```

从 2.0 版本开始，Spring Data Redis 已经不推荐直接显式设置连接的信息了，一方面为了使配置信息与建立连接工厂解耦，另一方面抽象出 Standalone、Sentinel 和 RedisCluster 三种模式的环境配置类与一个统一的 Jedis 客户端连接配置类（用于配置连接池和 SSL（Secure Socket Layer，安全套接字层）连接），这样可以更加灵活、方便地根据实际业务场景需要来配置连接信息。文件 3-6 以 Standalone 方式为例，展示了在不使用连接池的情况下，如何实例化 RedisTemplate。首先创建 RedisStandaloneConfiguration 实例并设置参数（第 7～11 行），然后根据该配置实例来初始化 Jedis 连接工厂（第 12 行）。

文件 3-6 的配置使用的是直接连接 Redis 的方式，即每次需要时都会创建新的连接。当并发量剧增时，这会带来性能上的开销，同时由于没有对连接数进行限制，可能使服务器崩溃导致无法响应。所以一般会建立连接池，事先初始化一组连接，供需要 Redis 连接的线程取用。采用连接池方式需要更改文件 3-6 的第 5～13 行，具体如下：

```
1    @Bean
2    public RedisConnectionFactory redisConnectionFactory() {
3       JedisPoolConfig jpc = new JedisPoolConfig();
4       jpc.setMaxTotal(8);
5       jpc.setMaxIdle(8);
6       jpc.setMinIdle(0);
```

```
7          jpc.setMaxWait(Duration.ofMillis(200));
8          //Redis 连接配置
9          RedisStandaloneConfiguration redisStandaloneConfiguration =
10             new RedisStandaloneConfiguration();
11         //设置 Redis 服务器的 IP
12         redisStandaloneConfiguration.setHostName("127.0.0.1");
13         //设置 Redis 服务器的端口号
14         redisStandaloneConfiguration.setPort(6379);
15         //连接的数据库
16         redisStandaloneConfiguration.setDatabase(0);
17         //JedisConnectionFactory 配置 jedisPoolConfig
18         JedisClientConfiguration.JedisClientConfigurationBuilder
19           jedisClientConfiguration = JedisClientConfiguration.builder();
20         //指定连接池
21         jedisClientConfiguration.usePooling().poolConfig(jpc);
22         //创建工厂对象
23         RedisConnectionFactory factory = new JedisConnectionFactory(
24           redisStandaloneConfiguration,jedisClientConfiguration.build());
25         return factory;
26     }
```

在默认情况下，Redis 服务器在启动时会创建 16 个数据库，编号从 0 到 15。不同的应用可以连接到不同的数据库上。上述代码的第 16 行通过 setDatabase()方法选择编号为 0 的数据库。此外，Spring Data Redis 提供的采用连接池创建 RedisTemplate 对象的方式并不优雅。如果要采用连接池创建 RedisTemplate 对象，推荐使用 Spring Boot。

RedisTemplate 默认使用 Java 的序列化程序。通过 RedisTemplate 写入或读取的任何对象都是通过 Java 序列化和反序列化的。可以通过 org.springframework.data.redis.serializer 包中提供的接口更改默认的序列化机制的设置，内容可见 3.12 节。

3.3 Spring 操作 Redis 字符串

在创建了 RedisTemplate(或 StringRedisTemplate)对象后，可以编程执行 Redis 操作了。Redis 可以存取多种不同类型的数据，其中有 5 种基础数据类型：字符串、列表、哈希、集合和有序集合。此外，还有流、地理空间数据、位图等。RedisTemplate 对基础数据类型的大部分操作都是借助表 3-3 中的方法和子接口完成的。

表 3-3 RedisTemplate 操作基础数据类型的主要方法和子接口

方法	子接口	描述
opsForValue()	ValueOperations<K,V>	操作字符串类型的条目
opsForList()	ListOperations<K,V>	操作列表类型的条目
opsForSet()	SetOperations<K,V>	操作集合类型的条目
opsForHash()	HashOperations<K,V>	操作哈希类型的条目
opsForZSet()	ZSetOperations<K,V>	操作有序集合类型的条目
opsForStream()	StreamOperations<K,HK,HM>	执行流操作命令的接口
opsForHyperLogLog()	HyperLogLogOperations<K,V>	操作超级日志
opsForGeo()	GeoOperations<K,M>	操作地理空间数据类型的条目

要操作 Redis 字符串，需要调用 RedisTemplate 类的 opsForValue()方法创建 ValueOperations 子接口对象，再调用 ValueOperations 子接口的相关方法。本节介绍 ValueOperations 子接口(org.springframework.data.redis.core.ValueOperations<K,V>)中的主要方法的使用。

(1) 方法原型：void set(K key,V value)；功能：设置键 key 的值 value；对应 Redis 命令：SET。

(2) 方法原型：@Nullable V get(Object key)；功能：返回键 key 的值，当键的值不存在或在流水线(或事务)中使用该方法时，返回 null；对应 Redis 命令：GET。

【例 3-1】 利用键 user 保存值 hello,redis。可以通过 ValueOperations 子接口提供的 set(String key,String value)方法实现，代码如文件 3-7 所示。

【文件 3-7】 MyRedisStringTest.java

```
1   @RunWith(SpringJUnit4ClassRunner.class)
2   @ContextConfiguration(classes = RedisTemplateConfig.class)
3   public class MyRedisStringTest {
4     @Autowired
5     private StringRedisTemplate redis;
6     @Test
7     public void testEx1() {
8         redis.opsForValue().set("user","hello,redis");
9         String str = redis.opsForValue().get("user");
10        assertEquals("hello,redis",str);
11    }
12  }
```

如文件 3-7 所示，本节的案例由于操作的值都是 String 类型的，因此采用了 RedisTemplate 类的子类 StringRedisTemplate。StringRedisTemplate 对象在文件 3-6 的第 14～20 行完成实例化，并注入测试类中(第 4、5 行)。表 3-3 中的子接口对象可以通过 RedisTemplate 或 StringRedisTemplate 创建。即通过调用 RedisTemplate 的 opsForValue()方法获得子接口引用(第 8 行)，再调用子接口中对应的方法完成字符串操作。第 8 行调用 set()方法向 Redis 中以 user 为键，写入值 hello,redis。第 9 行通过键 user 取出对应的值。

(3) 方法原型：void set(K key,V value,long timeout,TimeUnit unit)；功能：设置键 key 的值 value 和过期超时时长 timeout；对应 Redis 命令：SETEX。

【例 3-2】 以 name 为键将值 Tom 写入 Redis，并设置写入键值的有效时长为 10 秒。测试代码如文件 3-8 所示。

【文件 3-8】 例 3-2 测试代码

```
1   @Test
2   public void testEx2() throws Interrupted Exception {
3      redis.opsForValue().set("name","Tom",10,TimeUnit.SECONDS);
4      Thread.sleep(11000);
5      String str = redis.opsForValue().get("name");
6      assertEquals("Tom",str);
7   }
```

在设置 name 键值后，等待 10 秒再去获取 name 键的值，get()方法将返回 null，测试代码运行结果如图 3-3 所示。

```
java.lang.AssertionError:
Expected :Tom
Actual   :null
```

图 3-3　例 3-2 测试代码运行结果

（4）方法原型：void set(K key,V value,long offset)；功能：对于键 key 所存储的字符串；从指定偏移量 offset 开始用给定值 value 替代对应 Redis 命令：SETRANGE。

【例 3-3】 向 Redis 中以 key 为键存入字符串 hello，world，随后将该字符串替换为 hello，redis。测试代码如文件 3-9 所示。

【文件 3-9】 例 3-3 测试代码

```
1   @Test
2   public void testEx3(){
3       redis.opsForValue().set("key","hello,world");
4       redis.opsForValue().set("key","redis",6);
5       String str = redis.opsForValue().get("key");
6       assertEquals("hello,redis",str);
7   }
```

（5）方法原型：@Nullable Boolean setIfAbsent(K key,V value)；功能：如果缺少键 key，则设置以键 key 保存字符串值 value；对应的 Redis 命令：SETNX。

【例 3-4】 以 abs 为键，调用 setIfAbsent()方法保存字符串 absent，保存后将值改为 present。测试代码如文件 3-10 所示。

【文件 3-10】 例 3-4 测试代码

```
1   @Test
2   public void testEx4(){
3       assertTrue(redis.opsForValue().setIfAbsent("abs","absent"));
4       assertFalse(redis.opsForValue().setIfAbsent("abs","present"));
5   }
```

如文件 3-10 所示，如果 Redis 中不存在键 abs，则第 3 行代码执行成功。第 4 行准备将键 abs 的值修改为 present。由于键 abs 已存在，因此此次更改失败。

（6）方法原型：void multiSet(Map<? extends K,? extends V>map)；功能：使用元组中提供的键值对将多个键设置为多个值；对应的 Redis 命令：MSET。

（7）方法原型：@Nullable List<V>multiGet(Collection<K>keys)；功能：返回所有给定键的值，值按键的请求顺序返回；对应的 Redis 命令：MGET。

【例 3-5】 将字符串 aaa、bbb 和 ccc 一次性存入 Redis，这三个字符串对应的键分别为 multi1、multi2 和 multi3，再通过各自的键将它们取出。测试代码如文件 3-11 所示。

【文件 3-11】 例 3-5 测试代码

```
1   @Test
2   public void testEx5(){
3       Map<String, String> map = new HashMap<String, String>();
4       map.put("multi1","aaa");
5       map.put("multi2","bbb");
6       map.put("multi3","ccc");
7       redis.opsForValue().multiSet(map);
8
```

```
9        List<String> list = new ArrayList<String>();
10       list.add("multi1");
11       list.add("multi2");
12       list.add("multi3");
13       List<String> values = redis.opsForValue().multiGet(list);
14       values.forEach(System.out::println);
15   }
```

（8）方法原型：@Nullable V getAndSet(K key,V value)；功能：设置键key的值并返回其旧值；对应的Redis命令：GETSET。

【例3-6】 应用getAndSet()方法。测试代码如文件3-12所示。

【文件3-12】 例3-6测试代码

```
1   @Test
2   public void testEx6(){
3      redis.opsForValue().set("getset","test-11");
4      String str = redis.opsForValue().getAndSet("getset","test-22");
5      assertEquals("test-11",str);
6   }
```

（9）方法原型：@Nullable Long increment(K key,long delta)；功能：将存储在键key下的字符串的整数值按增量delta递增，如果key指定的值不存在，那么key的值会先被初始化为0，然后再执行递增；对应的Redis命令：INCRBY。

【例3-7】 点赞是社交网络中最常用的功能。本例模拟社交网络中对作品的点赞功能，并输出当前作品获赞的数量。测试代码如文件3-13所示。

【文件3-13】 例3-7测试代码

```
1    @Test
2    public void testEx7(){
3        boolean laud_flag = true;
4        Long l = 0L;
5        if(laud_flag)
6            l = redis.opsForValue().increment("articleId",1);
7        else
8            l = redis.opsForValue().increment("articleId",-1);
9        System.out.println(l);
10   }
```

将点赞数量存入Redis，以作品Id(标识符属性，字符串类型)为键。如果用户点赞，则变量laud_flag取值为true，对应的点赞数增加1；反之，用户取消点赞，变量laud_flag取值为false，对应的点赞数减1。

（10）方法原型：@Nullable Boolean setBit(K key,long offset,boolean value)；功能：对键key所存储的字符串值，设置或清除指定偏移量上的位；对应的Redis命令：SETBIT。

（11）方法原型：@Nullable Boolean getBit(K key,long offset)；功能：获取键对应值的ASCII码在offsest处的值；对应的Redis命令：GETBIT。

【例3-8】 利用键bit存入字符串a，并利用位运算将该字符串改为b。测试代码如文件3-14所示。

【文件 3-14】 例 3-8 测试代码

```
1    @Test
2    public void testEx8(){
3        redis.opsForValue().set("bit","a");
4        assertTrue(redis.opsForValue().getBit("bit",7));
5        redis.opsForValue().setBit("bit",6,true);
6        redis.opsForValue().setBit("bit",7,false);
7        assertEquals("b",redis.opsForValue().get("bit"));
8    }
```

上述测试代码第 3 行以 bit 为键存入字符串 a。字符'a'的 ASCII 码是 97，其二进制形式为 01100001。这样，第 4 行获取第 7 位的 ASCII 码值，得到二进制 1。1 代表 true，0 代表 false。因此，第 4 行结果应为 true。第 5、6 行分别将字符'a'的 ASCII 码的第 6、7 位设置为 1 和 0，这样键 bit 对应的 ASCII 码被改为 01100010，即字符'b'的 ASCII 码。

目前的软件系统（包括电商和社交网络）经常有这样的需求：根据用户提供的手机号码发送验证码，实现登录。下面的案例模拟实现手机验证码的发送功能。

【例 3-9】 利用 Redis 实现模拟手机验证码登录之验证码发送功能。对于验证码的发送，通常有如下要求：

（1）发送的手机验证码几分钟内（本例设定为 1 分钟）有效。

（2）每天向每个手机号码发送的验证码的次数有限（本例设定为 24 小时内最多发送 3 次）。

为实现上述两项要求，可以利用 Redis 保存两个值，并设置它们的生存时间：一个用于保存发送给用户的验证码，生存时长为 1 分钟；另一个用来保存给用户发送验证码的次数，生存时间为 24 小时。测试代码如文件 3-15 所示。

【文件 3-15】 ShortMessageSender.java

```
1    public class ShortMessageSender {
2        public String sendCode(StringRedisTemplate template,String phone){
3            String codeKey = phone + "_CODE";
4            String countKey = phone + "_COUNT";
5            Integer count = 0;
6            try {
7                count = Integer.parseInt(
8                    template.opsForValue().get(countKey));
9            } catch(NumberFormatException nfe) {
10               count = 0;
11           }
12           if(count > 2) {
13               System.out.println("24 小时内发送次数已达 3 次,24 小时后重试");
14               return "retry";
15           }
16           Boolean hasCodeKey = template.hasKey(codeKey);
17           if(hasCodeKey){
18               Long codeTTL = template.getExpire(codeKey);
19               System.out.println("验证码 1 分钟内有效,请在" + codeTTL +
20                   "秒之后再次发送");
21               return "tip";
```

```
22          }
23        String code = RandomUtil.randomNumbers(6);
24        System.out.println("CODE IS : " + code);
25        template.opsForValue().set(codeKey,code,60, TimeUnit.SECONDS);
26        long timeout = 24 * 60 * 60;
27        if(count!= 0)
28            timeout = template.getExpire(countKey);
29         template.opsForValue().set(countKey,String.valueOf(count + 1),
30             timeout,TimeUnit.SECONDS);
31         System.out.println("验证码发送成功");
32         return "success";
33     }
34 }
```

如文件 3-15 所示，对于用户提交的手机号码，以参数 phone 传入 sendCode()方法，并省去了对手机号码合法性的校验。第 3、4 行分别以手机号码附带后缀的形式定义了两个键，这两个键对应的值分别为验证码和发送验证码的次数。第 7、8 行试图从 Redis 中获取发送验证码的次数，当 Redis 中不存在键 countKey 时，将记录验证码发送次数的变量 count 计为 0(第 9～11 行)。第 23 行采用 RandomUtil 工具随机生成包含 6 位数字的验证码。要使用 RandomUtil 工具，需要在 pom.xml 文件中添加相关依赖：

```
1   <!-- hutool -->
2   <dependency>
3       <groupId>cn.hutool</groupId>
4       <artifactId>hutool-all</artifactId>
5       <version>5.8.20</version>
6   </dependency>
```

第 25 行将验证码保存到 Redis，以便用户提交验证码后进行比对。同时，设置了验证码在 Redis 中的保存时间为 60 秒。第 29 行用同样的方法将验证码的发送次数保存到 Redis，同时设置该值在 Redis 中的保存时间为 24 小时。

随后，模拟向号码为 15612345678 的手机发送验证码，测试代码如文件 3-16 所示。

【文件 3-16】 TestSMSVerification.java

```
1   @RunWith(SpringJUnit4ClassRunner.class)
2   @ContextConfiguration(classes = RedisTemplateConfig.class)
3   public class TestSMSVerification {
4       @Autowired
5       private StringRedisTemplate template;
6
7       @Test
8       public void testSender(){
9           ShortMessageSender sender = new ShortMessageSender();
10          sender.sendCode(template,"15612345678");
11      }
12  }
```

测试代码运行结果如图 3-4 所示，再次运行测试代码，结果如图 3-5 所示。

```
✔ Tests passed: 1 of 1 test – 158 ms
/Library/Java/JavaVirtualMac
SLF4J: Failed to load class
SLF4J: Defaulting to no-oper
SLF4J: See http://www.slf4j.
CODE IS : 124828
验证码发送成功

Process finished with exit c
```

图 3-4　文件 3-16 测试代码运行结果

```
✔ Tests passed: 1 of 1 test – 159 ms
/Library/Java/JavaVirtualMachines/
SLF4J: Failed to load class "org.s
SLF4J: Defaulting to no-operation
SLF4J: See http://www.slf4j.org/co
验证码1分钟内有效，请在33秒之后再次发送

Process finished with exit code 0
```

图 3-5　再次运行测试代码的结果

3.4　Spring 操作 Redis 列表

在 Redis 中，列表类型是按照元素插入的顺序排序的字符串列表。可以向列表的头部(左边)或尾部(右边)添加、删除元素。操作 Redis 列表的方法与操作 Redis 字符串的方法类似，要调用 RedisTemplate 类的 opsForList()方法创建 ListOperations 子接口对象，再调用 ListOperations 子接口的相关方法。本节介绍 ListOperations 子接口(org. springframework. data. redis. core. ListOperations < K,V >)中的主要方法的使用。

(1) 方法原型：@Nullable Long leftPush(K key,V value)；功能：将一个或多个值 value 插入列表 key 的表头，特点是先进后出，可以作为栈使用；返回值：执行插入操作后的列表长度；对应的 Redis 命令：LPUSH。

(2) 方法原型：@Nullable Long rightPush(K key,V value)；功能：将一个或多个值 value 插入列表 key 的表尾，特点是先进先出，可以作为队列使用；返回值：执行插入操作后的列表长度；对应的 Redis 命令：RPUSH。

(3) 方法原型：@Nullable List < V > range(K key,long start,long end)；功能：获取指定范围内的元素列表，参数 start 和 end 分别代表开始索引和结束索引。索引从左到右分别为 0 到 $N-1$，从右到左为 -1 到 $-N$。并且，end 参数包含了自身。对应的 Redis 命令：LRANGE。

【例 3-10】 将数字 1～10 以 strs 为键保存到 Redis 中，并倒序输出。测试代码如文件 3-17 所示。

【文件 3-17】　例 3-10 测试代码

```
1    @Test
2    public void testEx9() {
3        for(int i = 1;i < 11;i++)
4            redis.opsForList().leftPush("strs",String.valueOf(i));
5        redis.opsForList().range("strs", 0, -1)
6            .forEach(e -> System.out.print(e + " "));
7    }
```

```
10 9 8 7 6 5 4 3 2 1
```

图 3-6　例 3-10 测试代码运行结果

执行测试代码，运行结果如图 3-6 所示。从控制台输出可见 leftPush()方法实现了栈操作。读者可自行修改代码，实现正序输出。

（4）方法原型：@Nullable Long rightPushAll(K key,Collection < V > values)；功能：将一组值插入列表 key 的尾部；返回值：执行插入后的列表长度；对应的 Redis 命令：RPUSH。

（5）方法原型：@Nullable Long leftPushAll(K key,Collection < V > values)；功能：将一组值插入列表 key 的头部；返回值：执行插入后的列表长度；对应的 Redis 命令：LPUSH。

【例 3-11】 将数字 1～10 以 strs 为键批量保存到 Redis 中，并正序输出。测试代码如文件 3-18 所示。

【文件 3-18】 例 3-11 测试代码

```
1    @Test
2    public void testEx10() {
3        List<String> strs = new ArrayList<String>();
4        for(int i = 1;i < 11;i++)
5            strs.add(String.valueOf(i));
6        redis.opsForList().rightPushAll("strs",strs);
7        redis.opsForList().range("strs", 0, -1)
8            .forEach(System.out::println);
9    }
```

（6）方法原型：void trim(K key,long start,long end)；功能：修剪列表，使其保留 start 到 end 之间的值；对应的 Redis 命令：LTRIM。

【例 3-12】 修剪例 3-10 中的 strs 列表，保留 6～10。测试代码如文件 3-19 所示。

【文件 3-19】 例 3-12 测试代码

```
1    @Test
2    public void testEx11() {
3        System.out.println(redis.opsForList().range("strs",0,-1));
4        redis.opsForList().trim("strs",5,-1);
5        System.out.println(redis.opsForList().range("strs",0,-1));
6    }
```

测试代码运行结果如图 3-7 所示。

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[6, 7, 8, 9, 10]
```

图 3-7　例 3-12 测试代码运行结果

（7）方法原型：@Nullable V leftPop(K key)；功能：移除并返回列表 key 中的第一个元素；对应的 Redis 命令：LPOP。

（8）方法原型：@Nullable V rightPop(K key)；功能：移除并返回列表 key 中的最后一个元素；对应的 Redis 命令：RPOP。

【例 3-13】 利用 Redis List 数据类型实现栈和队列。测试代码如文件 3-20 所示。

【文件 3-20】 例 3-13 测试代码

```
1    @Test
2    public void testEx12(){
3        String s = "abcde";
4        for(int i = 0;i < s.length();i++)
5            redis.opsForList().rightPush("letters",
```

```
6              String.valueOf(s.charAt(i)));
7        System.out.println("the current list:"
8            + redis.opsForList().range("letters",0,-1));
9        System.out.println("As a stack");
10       for(int i = 0;i < 5;i++)
11           //Stack
12           System.out.println(redis.opsForList().rightPop("letters"));
13           //Queue
14           //System.out.println(redis.opsForList().leftPop("letters"));
15   }
```

如文件 3-20 所示，要实现栈，只要保证数据在列表的同一端执行插入和删除操作。为使字母入栈顺序与字母表序一致，采用了 rightPush()方法(第 5、6 行)在右侧执行插入操作。同时，在右侧执行删除操作(第 12 行)。测试代码运行结果如图 3-8 所示。

```
the current list:[a, b, c, d, e]
As a stack :
e d c b a
```

图 3-8 例 3-13 测试代码运行结果

(9) 方法原型：@Nullable V move(K sourceKey, RedisListCommands.Direction from, K destinationKey, RedisListCommands.Direction to)；功能：自动返回并删除存储在列表 sourceKey 中的第一个或最后一个元素(表头或表尾，取决于 from 参数)，并将该元素推送到列表 destinationKey 的第一个或最后一个元素(表头或表尾，取决于 to 参数)，该方法需要 Redis 6.2.0 及以上版本支持；对应的 Redis 命令：LMOVE。

【例 3-14】 向 list1 中添加三个值，分别为 one、two 和 three，随后将 three 和 one 从 list1 中移除，并分别移入 list2 的表头和表尾。测试代码如文件 3-21 所示。

【文件 3-21】 例 3-14 测试代码

```
1    @Test
2    public void testEx13() {
3        String[] s = {"one","two","three"};
4        redis.opsForList().rightPushAll("list1",s);
5        redis.opsForList().move("list1", RedisListCommands.Direction.RIGHT
6            ,"list2",RedisListCommands.Direction.LEFT);
7        redis.opsForList().move("list1",RedisListCommands.Direction.LEFT
8            ,"list2",RedisListCommands.Direction.RIGHT);
9        System.out.println(redis.opsForList().range("list1",0,-1));
10       System.out.println(redis.opsForList().range("list2",0,-1));
11   }
```

上述代码的第 5、6 行调用 move()方法，首先将 list1 中的尾部元素 three 删除，随后将其移入 list2 的头部。类似地，第 7、8 行调用 move()方法将 list1 中的头部元素 one 删除，随后将其移入 list2 的尾部。运行此测试代码，结果如图 3-9 所示。

```
[two]
[three, one]
```

图 3-9 例 3-14 测试代码运行结果

Redis 通常用作消息传递服务器，用于处理后台作业或其他类型的消息传递任务。一种简单的消息队列形式通常是将值推送到生产者端的列表中，等待消费者端使用 RPOP(使用轮询)命令使用该值。然而，这种消息队列并不可靠，因为消息可能会丢失。例如，网络存在传输问题的情况，或者如果消费者在收到消息后不久崩溃，但消息尚未被处理。

LMOVE(或 BLMOVE 用于阻塞变体)命令提供了一种避免此问题的方法：消费者获取消息，同时将其推送到待处理列表中。一旦消息被处理，消费者将使用 LREM 命令从处理列表中删除消息。还可以使用另一个客户端监视处理列表中的项目是否保留太长时间，并在需要时将这些超时的项目再次推送到消息队列中。

此外，利用 move()方法还可以实现访问 N 个元素列表中的每个元素，而无须使用 LRANGE 命令将完整列表从服务器传输到客户端。

【例 3-15】 利用 move()方法遍历列表 cirList。测试代码如文件 3-22 所示。

【文件 3-22】 例 3-15 测试代码

```
1    @Test
2    public void testEx14() {
3       String[] s = {"one","two","three"};
4       redis.opsForList().rightPushAll("cirList",s);
5       Long size = redis.opsForList().size("cirList");
6       for(int i = 0;i < size;i++)
7          System.out.println(redis.opsForList()
8                .move("cirList",RedisListCommands.Direction.RIGHT
9                ,"cirList",RedisListCommands.Direction.LEFT));
10   }
```

其中，第 5 行调用 size()方法获取列表 cirList 中元素的个数。其方法原型为@Nullable Long size(K key)。运行此测试代码，结果如图 3-10 所示。

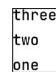

图 3-10 例 3-15 测试代码运行结果

(10) 方法原型：void set(K key,long index,V value)；功能：在索引 index 处设置列表元素的值；对应的 Redis 命令：LSET。

(11) 方法原型：@Nullable V index(K key,long index)；功能：获取列表 key 中索引为 index 的元素的值。对应的 Redis 命令：LINDEX。

【例 3-16】 将例 3-12 中结果列表的最后一个元素的值改为 100，并获取修改后的元素值。测试代码如文件 3-23 所示。

【文件 3-23】 例 3-16 测试代码

```
1    @Test
2    public void testEx15() {
3       redis.opsForList().set("strs",4,"100");
4       String n = redis.opsForList().index("strs",4);
5       assertEquals("100",n);
6    }
```

本章将 Redis 操作中的值都设定为 String 类型。因此，在上述代码的第 3 行，更改的新值也以字符串的形式表示。

(12) 方法原型：@Nullable Long remove(K key,long count,Object value)；功能：根据参数 count 的值，移除列表中与参数 value 相等的元素。count 的值可以是以下几种：

① count>0：从表头开始向表尾搜索，移除值与 value 相等的元素，数量为 count。

② count<0：从表尾开始向表头搜索，移除值与 value 相等的元素，数量为 count 的绝对值。

③ count＝0：移除表中所有值与 value 相等的值。

④ 返回值：被移除的元素的数量。因为不存在的 key 被视作空表，所以当 key 不存在时，该方法返回 0。对应的 Redis 命令：LREM。

【例 3-17】 移除列表中的重复值。测试代码如文件 3-24 所示。

【文件 3-24】 例 3-17 测试代码

```
1    @Test
2    public void testEx16() {
3        String[] s = {"hello","hello","foo","hello"};
4        redis.opsForList().rightPushAll("lrm",s);
5        System.out.println(redis.opsForList().range("lrm",0, - 1));
6        redis.opsForList().remove("lrm", - 2,"hello");
7        System.out.println(redis.opsForList().range("lrm",0, - 1));
8    }
```

```
[hello, hello, foo, hello]
[hello, foo]
```

图 3-11 例 3-17 测试代码运行结果

运行此测试代码，结果如图 3-11 所示。

社交网络（或电商系统）中经常有这样的需求，用户可以查看浏览内容的历史记录。如果只是要求保留用户的浏览记录，则可以用列表来实现。

【例 3-18】 Id 为 101 的用户某时段的浏览记录为{a.html，b.html，…，g.html}，要求将用户最近的 5 条浏览记录保留 3 天。测试代码如文件 3-25 所示。

【文件 3-25】 例 3-18 测试代码

```
1    @Test
2    public void testViewed(){
3        String[] pages = {"a.html","b.html","c.html","d.html","e.html",
4            "f.html","g.html"};
5         //保留最近 5 条浏览记录
6        int viewed_page_counter = 5;
7        int offset = pages.length - viewed_page_counter;
8        for(int i = 0;i < pages.length;i++) {
9            if(i < offset) {
10               template.opsForList().rightPush("101:20241011:viewed",
11                   pages[i]);
12               template.opsForList().leftPop("101:20241011:viewed");
13           } else {
14               template.opsForList().leftPush("101:20241011:viewed",
15                    pages[i]);
16               template.expire("101:20241011:viewed",3 * 24 * 60,
17                    TimeUnit.MINUTES);
18             }
19       }
20       System.out.println("浏览记录为：");
21       List < String > viewedPages = template.opsForList().range(
22           "101:20241011:viewed",0, - 1);
23       viewedPages.forEach(System.out::println);
24   }
```

测试代码运行结果如图 3-12 所示。

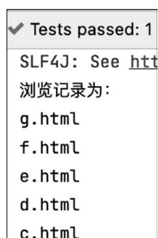

图 3-12 例 3-18 测试代码运行结果

3.5 Spring 操作 Redis 哈希

几乎所有的编程语言都提供了哈希类型。Redis 的哈希类型值是一个键值对结构，形如 value={{$field_1$, $value_1$},…,{$field_n$, $value_n$}}，因此哈希类型特别适合存储对象。Redis 是以键值对的形式存储数据的。Redis 键值对和哈希类型二者的关系可以用图 3-13 表示。

如图 3-13 所示，普通哈希类型数据< name，Tom >与< age，28 >以键 user：1 存储在 Redis 中。其映射关系在 Redis 中叫作字段值(field-value)，注意这里的值是指字段(field)对应的值，不是键对应的值。操作 Redis 哈希的方法与操作 Redis 字符串的方法类似，要调用 RedisTemplate 类的 opsForHash()方法创建 HashOperations 子接口对象，再调用 HashOperations 子接口的相关方法。本节介绍 HashOperations 子接口(org. springframework. data. redis. core. HashOperations < K，V >)中的主要方法的使用。

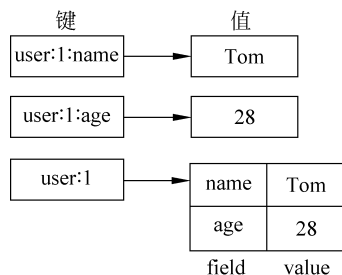

图 3-13 Redis 键值对和哈希类型的关系

(1) 方法原型：void put(H key，HK hashKey，HV value)；功能：将哈希 key 中的字段 hashKey 的值设置为 value；对应的 Redis 命令：HSET。

(2) 方法原型：@Nullable HV get(H key，Object hashKey)；功能：从哈希 key 中根据字段 hashKey 取出值；对应的 Redis 命令：HGET。

(3) 方法原型：Map < HK，HV > entries(H key)；功能：根据 key 获取整个哈希存储的值；对应的 Redis 命令：HGETALL。

(4) 方法原型：Set < HK > keys(H key)；功能：获取哈希 key 的所有字段名的集合；对应的 Redis 命令：HKEYS。

(5) 方法原型：List < HV > values(H key)；功能：获取哈希 key 的所有字段的值；对应的 Redis 命令：HVALS。

(6) 方法原型：Long size(H key)；功能：返回哈希中字段的数量；对应的 Redis 命令：HLEN。

(7) 方法原型：Boolean hasKey(H key，Object hashKey)；功能：判断哈希 key 中给定的字段 hashKey 是否存在；对应的 Redis 命令：HEXISTS。

【例 3-19】 Redis 哈希基础操作 1。要求：①将哈希数据< name：Tom >、< age：26 >、

<class:3>存储到哈希 rHash 中;②返回 rHash 中字段的数量;③取出年龄值;④取出 rHash 中存储的全部值;⑤取出全部字段名;⑥取出全部字段值;⑦判断 rHash 中是否存在 age 字段和 ttt 字段。测试代码如文件 3-26 所示。

【文件 3-26】 例 3-19 测试代码

```
1   @Test
2   public void testEx17() {
3       //①
4       template.opsForHash().put("rHash", "name", "Tom");
5       template.opsForHash().put("rHash", "age", "26");
6       template.opsForHash().put("rHash", "class", "6");
7       //②
8       assertEquals(3, template.opsForHash().size("rHash").intValue());
9       //③
10       assertEquals("26", template.opsForHash().get("rHash", "age"));
11      //④
12      System.out.println("stored Hash:" +
13          template.opsForHash().entries("rHash"));
14      //⑤
15      System.out.println("fields:" +
16          template.opsForHash().keys("rHash"));
17      //⑥
18      System.out.println("values:" +
19         template.opsForHash().values("rHash"));
20      //⑦
21      assertTrue(template.opsForHash().hasKey("rHash", "age"));
22      assertFalse(template.opsForHash().hasKey("rHash", "ttt"));
23  }
```

```
stored Hash:{name=Tom, age=26, class=6}
fields:[class, name, age]
values:[6, Tom, 26]
```

图 3-14 例 3-19 测试代码运行结果

运行上述测试代码,结果如图 3-14 所示。

(8) 方法原型:Cursor<Map.Entry<HK,HV>> scan(H key,ScanOptions options);功能:用于增量迭代哈希 key 中的键值对,参数 ScanOptions 是用于 SCAN 命令的选项,目前的常量值为 NONE(对扫描模式不做限制);对应的 Redis 命令:HSCAN。

该命令支持增量迭代,即每次执行都只会返回少量元素,所以该命令可以用于生产环境,而不会出现像 KEYS 命令带来的问题:当 KEYS 命令被用于处理一个大的数据库时,可能会阻塞服务器达数秒之久。在对键进行增量式迭代的过程中,键可能会被修改,所以增量式迭代命令只能对被返回的元素提供有限的保证。

(9) 方法原型:Long increment(H key,HK hashKey,long delta);功能:为哈希 key 中的字段 hashKey 的值加上增量 delta。增量也可以为负数,相当于对给定字段进行减法操作。如果 key 不存在,则会创建一个新的哈希并执行该方法。如果域 hashKey 不存在,那么在执行该方法前,字段的值被初始化为 0。对一个存储字符串值的字段执行该方法将造成一个错误。对应的 Redis 命令:HINCRBY。

由于内部序列化器的设置不同,因此此方法要求配合 StringRedisTemplate 类使用。

(10) 方法原型:void putAll(H key,Map<?extends HK,?extends HV>m);功能:将

多个字段值对同时设置到哈希 key 中。执行此方法会覆盖哈希中已存在的字段。如果 key 不存在，则创建一个空的哈希并执行该方法；对应的 Redis 命令：HMSET。

（11）方法原型：List<HV>multiGet(H key,Collection<HK>hashKeys)；功能：返回哈希 key 中，一个或多个给定字段 hashKeys 的值。如果给定的字段不存在于哈希，则返回一个 nil 值。对应的 Redis 命令：HMGET。

（12）方法原型：Long delete(H key,Object...hashKeys)；功能：删除哈希 key 中一个或多个指定的字段 hashKeys。对应的 Redis 命令：HDEL。

【例 3-20】 Redis 哈希基础操作 2。要求：①使用 scan()方法遍历例 3-19 中的哈希 rHash，并输出其全部值；②将 age 字段的值增加 1；③将哈希数据<name:Bob>、<age:28>、<class:2>一次性加入哈希 rHash2 中；④从哈希 rHash 中取出 name 字段和 age 字段的值；⑤删除哈希 rHash 中的字段 name。测试代码如文件 3-27 所示。

【文件 3-27】 例 3-20 测试代码

```
1    @Test
2    public void testEx18 () {
3        //①
4        template.opsForHash().put("rHash", "name", "Tom");
5        template.opsForHash().put("rHash", "age", "26");
6        template.opsForHash().put("rHash", "class", "6");
7        System.out.println("all the values in rHash are:");
8        Cursor<Map.Entry<Object,Object>> cursor =
9             template.opsForHash().scan("rHash", ScanOptions.NONE);
10       cursor.forEachRemaining(entry -> System.out.println(
11            entry.getKey() + ":" + entry.getValue()));
12       //②
13       //需注入 StringRedisTemplate
14       assertEquals("27",template.opsForHash().increment(
15           "rHash","age",1).toString());
16       //③
17       Map<String,Object> tempMap = new HashMap<String,Object>();
18       tempMap.put("name","Bob");
19       tempMap.put("age","28");
20       tempMap.put("class","2");
21       template.opsForHash().putAll("rHash2",tempMap);
22       System.out.println("rHash:"
23            + template.opsForHash().entries("rHash"));
24       System.out.println("rHash2:"
25            + template.opsForHash().entries("rHash2"));
26       //④
27       List<Object> ks = new ArrayList<Object>();
28       ks.add("name");
29       ks.add("age");
30       System.out.println("values for field name and age for rHash:");
31       System.out.println(template.opsForHash().multiGet("rHash",ks));
32       //⑤
33       template.opsForHash().delete("rHash","name");
34       System.out.println("name field has been removed. Now rHash looks
35          like this: " + template.opsForHash().entries("rHash"));
36   }
```

运行上述测试代码，结果如图 3-15 所示。

```
all the values in rHash are:
name:Tom
class:6
age:26
----------
rHash:{age=26, class=6, name=Tom}
rHash2:{age=28, name=Bob, class=2}
----------
values for field name and age for rHash:
[Tom, 26]
----------
name field has been removed. Now rHash looks like this: {class=6, age=26}
```

图 3-15　例 3-20 测试代码运行结果

目前的软件系统（包括电商和社交网络）经常有这样的需求：根据用户提供的手机号码发送验证码，实现登录。下面的例子在例 3-9 的基础上模拟手机验证码的登录验证功能。

【例 3-21】 模拟手机验证码登录功能，用户提交手机上收到的验证码并完成登录验证。对于验证成功的用户，将其登录信息保存到 Redis 中，测试代码如文件 3-28 所示。

【文件 3-28】 UserLogin.java

```
1    public class UserLogin {
2        public String login(StringRedisTemplate template, String phone,
3            String userCode){
4            String codeKey = phone + "_CODE";
5            String cacheCode = template.opsForValue().get(codeKey);
6            if(cacheCode == null || ! cacheCode.equals(userCode)) {
7                System.out.println("验证码错误");
8                return "FAIL";
9            }
10           User user = new User();
11           user.setUserId(Integer.valueOf(133));
12           user.setName("admin");
13           user.setPhone(phone);
14           String token = UUID.randomUUID().toString();
15           Map<String,Object> map = BeanUtil.beanToMap(user,
16               new HashMap<>(), CopyOptions.create()
17               .setIgnoreNullValue(true)
18               .setFieldValueEditor((fieldName,fieldValue)
19               -> fieldValue.toString()));
20           System.out.println(map);
21           String tokenKey = phone + "_" + token;
22           template.opsForHash().putAll(tokenKey,map);
23           template.expire(tokenKey,30, TimeUnit.MINUTES);
24           template.delete(codeKey);
25           return "OK";
26       }
27   }
```

如文件 3-28 所示，本例在例 3-9 基础上完成登录验证任务。第 4 行指定 Redis 中缓存的已发送给用户的验证码的键 codeKey。第 5 行根据 codeKey 获取发送给用户的验证码。第 6～9 行将已发送给用户的验证码和用户提交的验证码进行比对，若比对失败则报告错误

并退出。第 10～13 行先实例化 User 类(代码见本书配套源代码),再对其属性分别赋值。这些操作用来模拟通过手机号在数据库中检索用户的相关信息。第 14 行生成随机令牌,用来保存用户会话(Session)信息。第 15～19 行将 User 类的对象 user 转换为 Map,目的是准备将 Map 对象存入 Redis 哈希中。其中的 BeanUtil 是 hutool 提供的工具类,引入的依赖见例 3-9。第 22 行将用户信息存入 Redis 哈希。第 23 行设置用户信息的过期时间。用户登录验证成功,删除缓存的用户验证码(第 24 行)。

本例中,在用户登录验证成功后,将用户信息写入 Redis。在 Web 应用系统开发中,用户登录验证成功后,通常将用户信息写入 Session。当 Web 应用系统部署在一台 Tomcat 服务器上时,这样做是可行的。而工程上,为应对大量的并发请求,往往将 Web 应用系统部署到 Tomcat 集群上。而一个 Session 对象不能在多个 Tomcat 上使用,这样用户登录信息只能保存在一个 Tomcat 上。因为集群的存在,用户的请求会被分配到不同的 Tomcat 上处理。这样,当登录过的用户再次向系统发送请求时,请求可能会被分配到没有保存用户信息的 Tomcat。这时,该 Tomcat 就会要求用户重新登录,这样就会极大降低用户的体验度。并且,Tomcat 集群中也会保存大量冗余的用户登录信息,造成资源浪费。本例中,将用户的登录信息保存到 Redis 中就是对上述问题的一个解决方案。一方面,Tomcat 集群可以从 Redis 中获取用户的登录信息,实现数据共享；另一方面,由于 Redis 具有良好的读写性能,可以从容应对众多用户登录时带来的大量的并发请求。同时,本例给出的解决方案还有一个不足,就是第 23 行中设置了用户信息的保存时间为 30 分钟。如果用户的操作时长超过 30 分钟,会因为 Redis 中缓存的信息过期而被要求重新登录。这个问题需要其他技术手段来解决,此处不再详述。

随后,测试用户的登录验证功能,在文件 3-16 的基础上增加一个测试用例,测试代码如文件 3-29 所示。

【文件 3-29】 增加的测试用例

```
1    @Test
2    public void testLogin(){
3        UserLogin userLogin = new UserLogin();
4        userLogin.login(template,"15612345678","******");
5    }
```

测试时,需要先接收验证码。因此,要先执行文件 3-16 中的测试用例,将验证码填写到文件 3-29 第 4 行的“*”处。如果验证成功,则可利用 Redis 客户端查看 Redis 中保存的用户信息。Redis 中保存的用户信息及程序运行结果如图 3-16 所示。

图 3-16 Redis 中保存的用户信息及程序运行结果

3.6 Spring 操作 Redis 集合

与列表类型数据相似，集合类型数据也可以在同一个键下存储一个或多个元素。与列表类型不同的是，集合中不允许有重复元素，并且集合中的元素是无序的，不能通过索引获取元素。Redis 的集合是字符串类型元素的无序集合。Redis 除了支持集合内的增、删、改、查操作，还支持多个集合的交、并、差运算。

操作 Redis 集合的方法与操作 Redis 字符串的方法类似，要调用 RedisTemplate 类的 opsForSet()方法创建 SetOperations 子接口对象，再调用 SetOperations 子接口的相关方法。本节介绍 SetOperations 子接口(org.springframework.data.redis.core.SetOperations <K,V>)中的主要方法的使用。

(1) 方法原型：@Nullable Long add(K key,V...values)；功能：向集合 key 中添加元素，并返回添加的个数；对应的 Redis 命令：SADD。

(2) 方法原型：@Nullable Boolean move(K key,V value,K destKey)；功能：将元素 value 从集合 key 移动到集合 destKey；对应的 Redis 命令：SMOVE。

(3) 方法原型：@Nullable Set <V> members(K key)；功能：返回集合 key 中的所有成员，不存在的 key 被视为空集合；对应的 Redis 命令：SMEMBERS。

(4) 方法原型：@Nullable List <V> pop(K key,long count)；功能：从集合 key 中删除并返回 count 个随机成员；对应的 Redis 命令：SPOP。

(5) 方法原型：@Nullable Long remove(K key,Object...values)；功能：移除集合 key 中的一个或多个元素 value，不存在的元素 value 会被忽略。当 key 不是集合类型时，返回一个错误；对应的 Redis 命令：SREM。

(6) 方法原型：@Nullable Long size(K key)；功能：返回集合 key 的基数(集合中元素的数量)；对应的 Redis 命令：SCARD。

(7) 方法原型：@Nullable Boolean isMember(K key,Object o)；功能：检测集合 key 是否包含元素 o；对应的 Redis 命令：SISMEMBER。

(8) 方法原型：Cursor <V> scan(K key,ScanOptions options)；功能：支持增量式迭代集合中的元素；对应的 Redis 命令：SSCAN。

【例 3-22】 给定两个集合 ball 和 ball2，对两个集合执行下述操作：①向集合 ball 中添加 4 个元素，分别为 football、volleyball、basketball 和 pingpong；②输出集合 ball 中的所有元素；③迭代输出集合 ball 中的元素；④将集合 ball 中的元素 pingpong 移至集合 ball2 中，检查集合 ball 中元素的个数，并检查元素 basketball 是否属于集合 ball；⑤从集合 ball 中随机取出两个元素并输出；⑥将元素 pingpong 从集合 ball2 中移除并输出集合 ball2 的剩余元素。测试代码如文件 3-30 所示。

【文件 3-30】 例 3-22 测试代码

```
1    @Test
2    public void testEx19() {
3        //①
4        String[] s = {"football","volleyball","basketball","pingpong"};
5        template.opsForSet().add("ball",s);
```

```
 6        //②
 7        System.out.println("the members returned by members() method:");
 8        System.out.println(template.opsForSet().members("ball"));
 9        //③
10        System.out.println("----------------");
11        System.out.println("the members returned by scan() method:");
12        template.opsForSet().scan("ball", ScanOptions.NONE)
13           .forEachRemaining(System.out::println);
14        //④
15        template.opsForSet().move("ball","pingpong","ball2");
16        assertEquals(3,template.opsForSet().size("ball").intValue());
17        assertTrue(template.opsForSet().isMember("ball","basketball"));
18        //⑤
19        System.out.println("----------------");
20        System.out.println("obtain 2 items randomly from ball set");
21        System.out.println(template.opsForSet().pop("ball",2));
22        //⑥
23        System.out.println("----------------");
24        System.out.println("remove pingpong from ball2 set");
25        template.opsForSet().remove("ball2","pingpong");
26        System.out.println("the remaining elements in ball2 set :"
27            + template.opsForSet().members("ball2"));
28    }
```

运行上述测试代码,运行结果如图 3-17 所示。

```
the members returned by members() method:
[basketball, football, volleyball, pingpong]
----------------
the members returned by scan() method:
basketball
football
volleyball
pingpong
----------------
obtain 2 items randomly from ball set
[football, basketball]
----------------
remove pingpong from ball2 set
the remaining elements in ball2 set :[]
```

图 3-17 例 3-22 测试代码运行结果

(9) 方法原型:@Nullable Set<V> intersect(K key,Collection<K> otherKeys);功能:求集合 key 与其他多个集合 otherKeys 的交集,不存在的 key 被视为空集。该方法的另外两种重载形式:@Nullable Set<V> intersect(K key,K otherKey)和@Nullable Set<V> intersect(Collection<K> keys);对应的 Redis 命令:SINTER。

(10) 方法原型:@Nullable Long intersectAndStore(K key,K otherKey,K destKey);功能:求集合 key 与集合 otherKey 的交集,并将产生的交集元素存入集合 destKey 中。如果集合 destKey 已经存在,则将其覆盖。集合 destKey 可以是集合 key 本身。该方法的另外两种重载形式:@Nullable Long intersectAndStore(K key,Collection<K> otherKeys,K destKey)和@Nullable Long intersectAndStore(Collection<K> keys,K destKey);对应

的 Redis 命令：SINTERSTORE。

（11）方法原型：@Nullable Set < V > union(K key,K otherKey)；功能：求集合 key 与集合 otherKey 的并集，不存在的 key 被视为空集。该方法的另外两种重载形式：@Nullable Set < V > union(K key,Collection < K > otherKeys)和@Nullable Set < V > union(Collection < K > keys)；对应的 Redis 命令：SUNION。

（12）方法原型：@Nullable Long unionAndStore(K key,K otherKey,K destKey)；功能：求集合 key 和集合 otherKey 的并集，并将产生的并集元素存入集合 destKey 中。如果集合 destKey 已经存在，则将其覆盖。集合 destKey 可以是集合 key 本身。该方法的另外两种重载形式：@Nullable Long unionAndStore(K key,Collection < K > otherKeys,K destKey)和@Nullable Long unionAndStore(Collection < K > keys,K destKey)；对应的 Redis 命令：SUNIONSTORE。

（13）方法原型：@Nullable Set < V > difference(K key,K otherKey)；功能：求集合 key 与集合 otherKey 的差集，不存在的 key 被视为空集。该方法的另外两种重载形式：@Nullable Set < V > difference(K key,Collection < K > otherKeys)和@Nullable Set < V > difference(Collection < K > keys)；对应的 Redis 命令：SDIFF。

（14）方法原型：@Nullable Long differenceAndStore(K key,K otherKey,K destKey)；功能：求集合 key 和集合 otherKey 的差集，并将产生的差集元素存入集合 destKey 中。如果集合 destKey 已经存在，则将其覆盖。集合 destKey 可以是集合 key 本身。该方法的另外两种重载形式：@Nullable Long differenceAndStore(K key,Collection < K > otherKeys,K destKey)和@Nullable Long differenceAndStore(Collection < K > keys,K destKey)；对应的 Redis 命令：SDIFFSTORE。

【例 3-23】 在利用 Redis 存储社交关系时，可能会有如下需求：①在微博中 zhangsan 有一批好友，lisi 有另外一批好友，现需要查询 zhangsan 和 lisi 的共同好友；②要得到 zhangsan、lisi 和 wangwu 关注的所有公众号；③要得到 zhangsan 关注的但 lisi 和 wangwu 没有关注的公众号。测试代码如文件 3-31 所示。

【文件 3-31】 例 3-23 测试代码

```
1    @Test
2    public void testEx20(){
3        //准备好友数据
4        template.opsForSet().add("zhangsan-friend",
5           "friend1","friend2","friend3");
6        template.opsForSet().add("lisi-friend",
7           "friend1","friend3","friend4");
8        //①
9        System.out.println("Mutual friends of zhangsan & lisi:"
10          + template.opsForSet().intersect(
11       "zhangsan-friend","lisi-friend"));
12       //准备公众号数据
13       template.opsForSet().add("zhangsan-concern",
14          "pub1","pub2","pub3","pub4");
15       template.opsForSet().add("lisi-concern",
16          "pub1","pub2","pub5","pub6");
17       template.opsForSet().add("wangwu-concern",
```

```
18              "pub1","pub7","pub8","pub9");
19          //②
20          System.out.println("All concerns are :");
21          List < String > list = new ArrayList < String >();
22          list.add("lisi - concern");
23          list.add("wangwu - concern");
24          template.opsForSet().union("zhangsan - concern",list).iterator()
25            .forEachRemaining(e - > System.out.print(e + " "));
26          System.out.println();
27          //③
28          template.opsForSet().differenceAndStore("zhangsan - concern",
29             list,"zhangsanonly");
30          System.out.println("only zhangsan's concern :"
31              + template.opsForSet().members("zhangsanonly"));
32      }
```

如文件 3-31 所示，第 4～7 行准备 zhangsan 和 lisi 的好友数据。两者的共同好友可以通过求解两者好友集合的交集得到（第 9～11 行）。第 13～18 行准备 zhangsan、lisi 和 wangwu 的关注公众号数据。要得到三个人关注的所有公众号，只要求解三者关注公众号集合的并集即可。在求解并集时，调用的是 union()方法的一种重载形式：@Nullable Set < V > union(K key，Collection < K > otherKeys)（第 24 行）。最后遍历结果集合，输出并集的全部元素（第 25、26 行）。要得到只有 zhangsan 关注的公众号，只需求解 zhangsan 关注的公众号集合与其他两人关注的公众号集合的差集即可（第 28、29 行）。运行此测试代码，运行结果如图 3-18 所示。

```
Mutual friends of zhangsan & lisi:[friend3, friend1]
All concerns are :
pub1 pub4 pub6 pub5 pub2 pub3 pub7 pub8 pub9
only zhangsan's concern :[pub3, pub4]
```

图 3-18 例 3-23 测试代码运行结果

3.7 Spring 操作 Redis 有序集合

有序集合（Sorted set，也称 ZSet）同集合有一定的相似性，也是字符串类型元素的集合，并且都不能出现重复元素。在有序集合里，每个数据都会对应一个 double 类型的参数 score（分数）。Redis 正是按照分数来为集合中的元素进行升序排列。有序集合的元素是唯一的，但分数却可以重复。表 3-4 给出了 Redis 列表、集合和有序集合的异同点。

表 3-4 Redis 列表、集合和有序集合的异同点

数据结构	是否允许重复元素	是否有序	有序实现方式	应用场景
列表	是	是	索引（下标）	时间轴、消息队列等
集合	否	否	无	标签、社交网络等
有序集合	否	是	分数	排行榜、社交网络等

操作 Redis 有序集合的方法与操作 Redis 字符串的方法类似，要调用 RedisTemplate 类的 opsForZSet()方法创建 ZSetOperations 子接口对象，再调用 ZSetOperations 子接口的相关方法。本节介绍 ZSetOperations 子接口（org. springframework. data. redis. core. ZSetOperations

<K,V>)中的主要方法的使用。

3.7.1 对单个集合的操作

（1）方法原型：@Nullable Boolean add(K key,V value,double score)；功能：将一个或多个元素 value 及其分数 score 加入有序集合 key 中。如果某个元素 value 已经是有序集合的成员，那么更新这个元素 value 的分数 score，并通过重新插入这个元素来保证其在正确的位置上。分数 score 可以是整数值或双精度浮点数。如果有序集合 key 不存在，则创建一个空的有序集并执行 add()方法。当 key 存在但不是有序集合类型时，返回一个错误。该方法的另外一种重载形式：@Nullable Long add(K key,Set <ZSetOperations. TypedTuple <V>> tuples)；返回值：被成功添加的新成员的数量，不包括那些被更新的、已经存在的成员；对应的 Redis 命令：ZADD。

（2）方法原型：@Nullable Set <V> range(K key,long start,long end)；功能：返回有序集合在指定区间[start,end]内的成员；下标参数 start 和 stop 都以 0 表示有序集合的第一个成员，以 1 表示有序集合的第二个成员，以此类推。也可以使用负数下标，以 -1 表示最后一个成员，-2 表示倒数第二个成员，以此类推。超出范围的下标并不会引起错误。例如，当 start 的值比有序集合的最大下标还要大或 start＞end 时，该方法只是简单地返回一个空列表。另外，假如参数 end 的值比有序集的最大下标还要大，那么 Redis 将 end 当作最大下标来处理。对应的 Redis 命令：ZRANGE。

（3）方法原型：@Nullable Set <ZSetOperations. TypedTuple <V>> rangeWithScores(K key,long start,long end)；功能：返回有序集合 key 的下标在指定区间[start,end]内的成员对象，其中有序集成员按分数值递增顺序排列；对应的 Redis 命令：ZRANGE。

（4）方法原型：@Nullable default Set <V> reverseRangeByLex(K key,RedisZSetCommands. Range range)；功能：从有序集合 key 中获取 range 范围内的具有反向字典顺序的所有元素；其中的 RedisZSetCommands. Range 对应的 Redis 命令：ZREVRANGEBYLEX。

（5）方法原型：@Nullable Set <ZSetOperations. TypedTuple <V>> popMax(K key, long count)；功能：从有序集合 key 中返回并移除 count 个分数最大的元素。

（6）方法原型：@Nullable Long zCard(K key)；功能：获取有序集合 key 的成员数；对应的 Redis 命令：ZCARD。

（7）方法原型：@Nullable Long removeRange(K key,long start,long end)；功能：移除有序集合 key 中指定排名区间[start,end]内的所有成员。下标参数 start 和 end 都以 0 表示有序集合的第一个成员，1 表示有序集合的第二个成员，以此类推。也可以使用负数下标，-1 表示最后一个成员，-2 表示倒数第二个成员，以此类推。返回值：被移除的成员数量；对应的 Redis 命令：ZREMRANGEBYRANK。

【例 3-24】 有序集合基本操作 1(模拟学生成绩管理)。要求：①向有序集合 students 中添加 4 个元素，元素名字分别为 SuXun、SuShi、SuZhe、HanYu，分数分别为 80.0、81.0、81.0、88.0；②输出 students 集合中的全部元素；③将 students 中的元素按分数由低到高排序并输出；④将 students 中的元素按分数由高到低排列，在分数相同的情况下，按反字母表序排列；⑤取出分数最高的元素；⑥删除 students 集合中的剩余元素。测试代码如文件 3-32 所示。

【文件 3-32】 例 3-24 测试代码

```
1    @Test
2    public void testEx21() {
3        ZSetOperations.TypedTuple<String> tuple1 =
4            new DefaultTypedTuple<String>("SuXun",80.0);
5        ZSetOperations.TypedTuple<String> tuple2 =
6            new DefaultTypedTuple<String>("SuShi",81.0);
7        ZSetOperations.TypedTuple<String> tuple3 =
8            new DefaultTypedTuple<String>("SuZhe",81.0);
9        ZSetOperations.TypedTuple<String> tuple4 =
10           new DefaultTypedTuple<String>("HanYu",88.0);
11       Set<ZSetOperations.TypedTuple<String>> t =
12           new HashSet<ZSetOperations.TypedTuple<String>>();
13       t.add(tuple1);    t.add(tuple2);
14       t.add(tuple3);    t.add(tuple4);
15       //①
16       template.opsForZSet().add("students",t);
17       //②
18       System.out.println(template.opsForZSet()
19            .range("students", -4, -1));
20       //③
21       System.out.println(template.opsForZSet()
22           .rangeWithScores("students",0, -1));
23       //④
24       RedisZSetCommands.Range range = new RedisZSetCommands.Range();
25       System.out.println(template.opsForZSet()
26           .reverseRangeByLex("students", range.gte("H")));
27       //⑤
28       System.out.println(template.opsForZSet().popMax("students",1));
29
30       assertEquals(3,template.opsForZSet()
31           .zCard("students").intValue());
32       //⑥
33       template.opsForZSet().removeRange("students",0, -1);
34       System.out.println(template.opsForZSet()
35           .range("students",0, -1));
36   }
```

其中,文件 3-32 的第 26 行调用 reverseRangeByLex()方法。该方法用于获取满足非分数的排序取值。这个排序只有在分数相同的情况下才能使用,如果有不同的分数则返回值不确定。运行此测试代码,运行结果如图 3-19 所示。

```
集合中的全部元素: [SuXun, SuShi, SuZhe, HanYu]
----------
分数由高到低排序输出:
[DefaultTypedTuple [score=80.0, value=SuXun],
 DefaultTypedTuple [score=81.0, value=SuShi],
 DefaultTypedTuple [score=81.0, value=SuZhe],
 DefaultTypedTuple [score=88.0, value=HanYu]]
----------
分数由高到低排序, 分数相同的情况下, 按字母表反序:
[HanYu, SuZhe, SuShi, SuXun]
----------
最高分:
[DefaultTypedTuple [score=88.0, value=HanYu]]
----------
移除集合中的所有元素:
[]
```

图 3-19 例 3-24 测试代码运行结果

(8) 方法原型:@Nullable Long count(K key, double min, double max);功能:返回分数在[min, max]区间内的成员个数;对应的 Redis 命令:ZCOUNT。

(9) 方法原型:@Nullable Long remove(K key, Object...values);功能:移除有序集合 key 中的一个

或多个成员，不存在的成员将被忽略。当 key 存在但不是有序集合类型时，返回一个错误。返回被成功移除的成员的数量，不包括被忽略的成员。对应的 Redis 命令：ZREM。

（10）方法原型：@Nullable Long rank(K key,Object o)；功能：返回有序集合 key 中指定成员 o 的排名，其中有序集合成员按分数值递增顺序排列；对应的 Redis 命令：ZRANK。

（11）方法原型：@Nullable Long reverseRank(K key,Object o)；功能：返回有序集合 key 中指定成员 o 的排名，其中有序集合成员按分数值递减顺序排列；对应的 Redis 命令：ZREVRANK。

（12）方法原型：@Nullable Set<ZSetOperations.TypedTuple<V>> reverseRangeWithScores(K key,long start,long end)；功能：返回有序集合 key 在指定区间[start,end]内的成员对象，其中有序集合成员按分数值递减顺序排列；对应的 Redis 命令：ZREVRANGEBYSCORE。

（13）方法原型：@Nullable Double score(K key,Object o)；功能：获取指定成员的 score 值；对应的 Redis 命令：ZSCORE。

（14）方法原型：@Nullable Set<V> distinctRandomMembers(K key,long count)；功能：从有序集合 key 中随机获取 count 个不同元素，该方法需要 Redis 6.2.0 及以上版本支持；对应的 Redis 命令：ZRANDMEMBER。

（15）方法原型：@Nullable Double incrementScore(K key,V value,double delta)；功能：将有序集合 key 中值为 value 的元素的分数增加增量 delta；对应的 Redis 命令：ZINCRBY。

（16）方法原型：Cursor<ZSetOperations.TypedTuple<V>> scan(K key,ScanOptions options)；功能：使用光标(Cursor)在有序集合 key 上迭代元素（包括元素成员和元素分数）；对应的 Redis 命令：ZSCAN。

【例 3-25】 有序集合基本操作 2（模拟视频网站的一些基础操作）。要求：①向有序集合 zset 中添加三个元素，元素名字分别为 zset-1、zset-2 和 zset-3，分数分别为 9.9、9.6、9.1；②对 zset 中的元素按分数升序排列并输出排序结果；③计算分数在 9.3 分以上的元素数量；④从集合 zset 中随机抽取一个元素，并获取其排名及分数；⑤将 zset-1 的分数提高到 10.1。测试代码如文件 3-33 所示。

【文件 3-33】 例 3-25 测试代码

```
1    @Test
2    public void testEx22() {
3        ZSetOperations.TypedTuple<String> objectTypedTuple1 =
4            new DefaultTypedTuple<String>("zset-1", 9.9);
5        ZSetOperations.TypedTuple<String> objectTypedTuple2 =
6            new DefaultTypedTuple<String>("zset-2", 9.6);
7        ZSetOperations.TypedTuple<String> objectTypedTuple3 =
8            new DefaultTypedTuple<String>("zset-3", 9.1);
9        Set<ZSetOperations.TypedTuple<String>> tuples =
10           new HashSet<ZSetOperations.TypedTuple<String>>();
11       tuples.add(objectTypedTuple1);
12       tuples.add(objectTypedTuple2);
13       tuples.add(objectTypedTuple3);
14       //①
```

```
15          template.opsForZSet().add("zset", tuples);
16          //②
17          tuples = template.opsForZSet().reverseRangeWithScores(
18              "zset", 0, -1);
19          Cursor cursor = template.opsForZSet()
20              .scan("zset", ScanOptions.NONE);
21          cursor.forEachRemaining(System.out::println);
22          //③
23          assertEquals(2, template.opsForZSet()
24              .count("zset", 9.3, 10.0).intValue());
25          //④
26          Set set = template.opsForZSet().distinctRandomMembers("zset", 1);
27          Iterator<String> iterator = set.iterator();
28          iterator.forEachRemaining(item -> System.out.println(item
29               + " 排名: " + String.valueOf(template.opsForZSet()
30              .rank("zset", item).intValue() + 1)
31               + " 分数: " + template.opsForZSet().score("zset", item)));
32          //⑤
33          assertEquals(10.1, template.opsForZSet()
34              .incrementScore("zset", "zset-1", 0.2).doubleValue(), 0.01);
35      }
```

运行测试代码,运行结果如图 3-20 所示。

```
按分数升序排列的结果:
DefaultTypedTuple [score=9.1, value=zset-3]
DefaultTypedTuple [score=9.6, value=zset-2]
DefaultTypedTuple [score=9.9, value=zset-1]
----------
随机抽取一个元素, 排名及分数:
zset-2 排名: 11 分数: 9.6
----------
```

图 3-20　例 3-25 测试代码运行结果

3.7.2　对多个集合的操作

(1) 方法原型:@Nullable Long unionAndStore(K key,Collection < K > otherKeys,K destKey,RedisZSetCommands. Aggregate aggregate,RedisZSetCommands. Weights weights);功能:计算有序集合 key 和 otherKeys 的并集,并将结果存储于集合 destKey 中。其中,参数 aggregate 表示并集选项,默认值为 SUM,表示结果集合 destKey 中元素的分数为该元素在各集合中分数的和;参数 aggregate 的另两个值为 MAX 和 MIN,分别表示结果集合 destKey 中的元素的分数为该元素在各集合中分数的最大和最小值。参数 weights 用于指定若干乘数因子,参与并集运算的每个集合中的元素的分数与指定的乘法因子相乘,得到该元素在结果集合中的分数,该参数的默认值为 1。对应的 Redis 命令:ZUNIONSTORE。

(2) 方法原型:@Nullable Set < ZSetOperations. TypedTuple < V >> unionWithScores(K key,Collection < K > otherKeys,RedisZSetCommands. Aggregate aggregate,RedisZSetCommands weights);功能:计算两个有序集合的并集,其中参数 aggregate 和 weights 的含义同(1),该方法需要 Redis 6.2.0 及以上版本支持;对应的 Redis 命令:ZUNION。

(3) 方法原型:@Nullable Long intersectAndStore(K key,Collection < K > otherKeys,

K destKey，RedisZSetCommands. Aggregate aggregate，RedisZSetCommands. Weights weights)；功能：计算有序集合 key 和 otherKeys 的交集，并将结果存储于集合 destKey 中，其中参数 aggregate 和 weights 的含义同(1)；对应的 Redis 命令：ZINTERSTORE。

(4) 方法原型：@Nullable Set < ZSetOperations. TypedTuple < V >> intersectWithScores(K key，Collection < K > otherKeys，RedisZSetCommands. Aggregate aggregate，RedisZSetCommands Weights weights)；功能：计算两个有序集合的交集，其中参数 aggregate 和 weights 的含义同(1)；该方法需要 Redis 6.2.0 及以上版本支持；对应的 Redis 命令：ZINTER。

(5) 方法原型：@Nullable Long differenceAndStore(K key，Collection < K > otherKeys，K destKey)；功能：求解有序集合 key 和 otherKeys 的差集，并将结果存放于集合 destKey 中，该方法需要 Redis 6.2.0 及以上版本支持；对应的 Redis 命令：ZDIFFSTORE。

(6) 方法原型：@Nullable Set < ZSetOperations. TypedTuple < V >> differenceWithScores (K key，Collection < K > otherKeys)；功能：计算两个有序集合的差集，该方法需要 Redis 6.2.0 及以上版本支持；对应的 Redis 命令：ZDIFF。

【例 3-26】 公司决定调整岗位工资。目前的岗位情况如下：LiBai、XinQiji 为创作岗位，岗位工资分别为 2300、2100；YueFei、XinQiji 为管理岗位，岗位工资分别为 3300 和 3700。要求：①查看只供职于创作岗位的人员及其岗位工资；②查看供职于多个岗位的人员及其最高档位的岗位工资；③对所有人员的岗位工资普遍调整，创作岗位人员在原岗位工资基础上翻倍，管理岗位人员在原岗位工资基础上上浮 20%，对兼任多个岗位的人员，岗位工资按各自岗位工资调整细则调整后就高，输出岗位工资调整后的所有人员的岗位工资。测试代码如文件 3-34 所示。

【文件 3-34】 例 3-26 测试代码

```
1   @Test
2   public void testEx23(){
3      ZSetOperations.TypedTuple < String > p1 =
4          new DefaultTypedTuple < String >("LiBai", 2300.0);
5      ZSetOperations.TypedTuple < String > p2 =
6          new DefaultTypedTuple < String >("XinQiji", 2100.0);
7      Set < ZSetOperations.TypedTuple < String >> ptuples =
8          new HashSet < ZSetOperations.TypedTuple < String >>();
9      ptuples.add(p1);        ptuples.add(p2);
10     template.opsForZSet().add("poet",ptuples);
11     ZSetOperations.TypedTuple < String > m1 =
12         new DefaultTypedTuple < String >("YueFei", 3300.0);
13     ZSetOperations.TypedTuple < String > m2 =
14         new DefaultTypedTuple < String >("XinQiji", 3700.0);
15     Set < ZSetOperations.TypedTuple < String >> mtuples =
16         new HashSet < ZSetOperations.TypedTuple < String >>();
17     mtuples.add(m1);        mtuples.add(m2);
18     template.opsForZSet().add("general",mtuples);
19     List < String > sets = new ArrayList < String >();
20     sets.add("general");
21     //①
22     System.out.println("创作岗位人员：");
23     template.opsForZSet().differenceAndStore("poet",sets,"res");
24     System.out.println(template.opsForZSet()
```

```
25            .rangeWithScores("res",0,-1));
26        //②
27        System.out.println("兼任多个岗位人员：");
28        System.out.println(template.opsForZSet().intersectWithScores(
29            "poet",sets,RedisZSetCommands.Aggregate.MAX));
30        //③
31        System.out.println("普调后的岗位工资：");
32        template.opsForZSet().unionAndStore("poet",sets,"person",
33            RedisZSetCommands.Aggregate.MAX,
34            RedisZSetCommands.Weights.of(2.0,1.2));
35        System.out.println(template.opsForZSet()
36            .rangeWithScores("person",0,-1));
37    }
```

其中，文件3-34的第3～20行为准备数据阶段。对于要求①，通过计算集合间的差集可求解（第23行）。对于要求②，只要求解两类人员集合的交集即可。第28、29行在调用intersectWithScores()方法求解时，指定了参数aggregate的值为MAX。对于要求③，基本思路是求解并集，但在执行并集操作前，需要将两个集合中的元素的分数分别与两个乘法因子相乘，第34行调用了RedisZSetCommands.Weights类的静态方法of()为每个参与并运算的集合分别设置乘法因子。其中，of()方法的原型如下：

```
public static RedisZSetCommands.Weights of(double... weights)
```

运行测试代码，运行结果如图3-21所示。

```
创作岗位人员：
[DefaultTypedTuple [score=2300.0, value=LiBai]]
兼任多个岗位人员：
[DefaultTypedTuple [score=3700.0, value=XinQiji]]
普调后的岗位工资：
[DefaultTypedTuple [score=3960.0, value=YueFei],
 DefaultTypedTuple [score=4440.0, value=XinQiji],
 DefaultTypedTuple [score=4600.0, value=LiBai]]
```

图3-21　例3-26测试代码运行结果

3.8　Spring 操作 HyperLogLog

HyperLogLog（超级日志，以下简称HLL）是Redis 2.8.9版本增加的数据结构，它可以在不保存集合元素的情况下进行集合基数（集合中元素的个数）的统计。在Redis中，每个HLL键只需要花费12KB内存，就可以计算接近2^{64}个不同元素的基数。HLL的工作原理是HLL概率算法。HLL的统计规则是基于概率的，不会非常准确，其标准差为0.81%。这个误差在工程上是完全可以接受的。

对于HLL的应用，通常有这样的场景：如要统计今天连接到网站的IP地址的数量，同一个IP多次访问计数为1。此问题可抽象化：如原始的数据记录为{1,1,2,3,3,3,3,4,4,5}，去重后的数据访问记录集合为Visitors={1,2,3,4,5}，Visitors集合的基数为5。对于计算集合基数的问题，一般可采用集合类型（Java和Redis中都有集合类型）。对于数据量小的应用场景来说，这是可行的。但是随着数据量增大，集合只会无限扩张，最后变得很庞大，占

用大量内存空间。此时，可以用 HLL 来解决问题。因为 HLL 只会根据输入元素来计算基数，而不会存储元素本身。

操作 HLL 的方法与操作 Redis 字符串的方法类似，要调用 RedisTemplate 类的 opsForHyperLogLog( ) 方法创建 HyperLogLogOperations 子接口对象，再调用 HyperLogLogOperations 子接口的相关方法。本节介绍 HyperLogLogOperations<K,V> 子接口(org.springframework.data.redis.core.HyperLogLogOperations<K,V>)中的主要方法的使用。

(1) 方法原型：Long add(K key,V...values)；功能：将给定的一个或多个值 values 添加到键 key。返回值：至少有一个值添加到键 key 时返回 1，否则返回 0；在流水线或事务中使用时返回 null。对应的 Redis 命令：PFADD。

(2) 方法原型：void delete(K key)；功能：删除给定的键 key。

(3) 方法原型：Long size(K...keys)；功能：获取键 keys 中当前元素的数量，在流水线或事务中使用时返回 null。对应的 Redis 命令：PFCOUNT。

(4) 方法原型：Long union(K destination,K...sourceKeys)；功能：将键 sourceKeys 的所有值合并到键 destination 中；对应的 Redis 命令：PFMERGE。

【例 3-27】 分别以 u1 和 u2 为键，向 HyperLogLog 各存入 10 000 000(1000 万)个范围在[0,99 999]的随机整数。输出 u1 和 u2 中元素的个数，并将 u1 和 u2 合并为 u，输出 u 中元素的个数。最后删除 u1、u2 和 u。测试代码如文件 3-35 所示。

【文件 3-35】 例 3-27 测试代码

```
1    @Test
2    public void testHyperLogLogOperations() {
3        HyperLogLogOperations<String, String> hyperLogLog =
4     template.opsForHyperLogLog();
5        String[] r = new String[5000];
6        String[] s = new String[5000];
7        for(int i = 0;i < 10000000;i++) {
8            int p = i % 5000;
9            r[p] = String.valueOf((int)(Math.random() * 100000));
10           if(p == 4999)
11               hyperLogLog.add("u1",r);
12       }
13       for(int j = 0;j < 10000000;j++){
14           int q = j % 5000;
15           s[q] = String.valueOf((int)(Math.random() * 100000));
16           if(q == 4999)
17               hyperLogLog.add("u2",s);
18       }
19       System.out.println("the size of u1 is " + hyperLogLog.size("u1"));
20       System.out.println("the size of u2 is " + hyperLogLog.size("u2"));
21
22       hyperLogLog.union("u","u1","u2");
23       System.out.println("the size of union is " + hyperLogLog.size("u"));
24       /*
25       hyperLogLog.delete("u1");
26       hyperLogLog.delete("u2");
```

```
27        hyperLogLog.delete("u");
28        */
29    }
```

在运行此测试代码前，可利用 info memory 命令查看 Redis 的内存占用情况，如图 3-22 所示。

```
127.0.0.1:6379> info memory
# Memory
used_memory:725368
used_memory_human:708.37K
```

图 3-22 运行测试代码前 Redis 的内存占用情况

运行测试代码后，控制台输出如图 3-23 所示。从输出结果可知，HLL 能够实现元素的自动去重，效果和集合类似。此外，union()方法（第 22 行）将两个 HLL 进行了合并，并且去掉了重复的元素，然后返回不重复元素个数。再次使用 info memory 命令检查 Redis 的内存占用情况，如图 3-24 所示。从代码运行前后的内存占用情况来看，向 Redis 中写入两千万条数据后（97 781 490B，约 93.25MB），内存占用由 725 368B 增加至 771 352B，只增加了 44.9KB。

```
the size of u1 is 99565
the size of u2 is 99565
the size of union is 99565
```

图 3-23 例 3-27 测试代码运行结果

```
127.0.0.1:6379> info memory
# Memory
used_memory:771352
used_memory_human:753.27K
```

图 3-24 运行测试代码后 Redis 的内存占用情况

HLL 的一项重要应用就是计数器。对于 Web 应用程序来讲，持续收集信息是一件非常重要的事。例如，已知网站在 5 分钟内得到 11 000 次点击，数据库在 5 秒内处理了 200 次写入和 600 次读取请求。这些数据都是非常重要的。因为通过在一段时间内持续地记录这些信息，可以了解网站流量的增减情况，进而根据这些数据预测何时需要对服务器进行升级，从而防止系统因为负载增加而宕机。对于网站来说，与流量相关的指标点有 UV（Unique Visitor，限定时段内同一客户端多次访问计为 1 次）、IP（用户的 IP 地址，限定时段内同一 IP 地址多次访问计为 1 次）等。

【例 3-28】 要求记录 0～24 时访问网站的 IP 地址的数量（同一 IP 地址多次访问计为 1 次），记录保留 24 小时。如果选用集合作为存储 IP 地址的数据类型，则会因为 IP 地址数量的增加导致集合占用的内存空间不断增大。本例只需要统计去重后的 IP 地址的数量，而无须记录每个 IP 地址，因此可以采用 HLL 来完成此任务。测试代码如文件 3-36 所示。

【文件 3-36】 例 3-28 测试代码

```
1    @Test
2    public void testIPCounter(){
3       HyperLogLogOperations<String, String> hLLOps =
4          template.opsForHyperLogLog();
5       String remoteAddr = "",backAddr = "",date_key = "";
6       Random random = new Random();
7       StringBuilder ipBuilder = new StringBuilder();
8       for(int j = 0; j < 10; j++) {
9          for (int i = 0; i < 4; i++) {
10            ipBuilder.append(random.nextInt(256));
11            if (i != 3)
12               ipBuilder.append(".");
13         }
14         if( j < 8 ) {
```

```
15                remoteAddr = ipBuilder.toString();
16                if(j == 1)
17                    backAddr = remoteAddr;
18            }
19            else
20                remoteAddr = backAddr;
21            System.out.println("Remote IP : " + remoteAddr);
22            date_key = new SimpleDateFormat(
23                 "yyyy - MM - dd HH:mm").format(System.currentTimeMillis());
24            hLLOps.add(date_key,remoteAddr);
25            template.expire(date_key,24,TimeUnit.HOURS);
26            ipBuilder.setLength(0);
27        }
28        System.out.println("IP 地址数: " + hLLOps.size(date_key));
29    }
```

```
Remote IP : 201.104.221.132
Remote IP : 110.124.243.96
Remote IP : 164.253.37.25
Remote IP : 61.150.245.217
Remote IP : 223.125.92.78
Remote IP : 30.149.115.231
Remote IP : 145.227.120.104
Remote IP : 45.113.215.205
Remote IP : 110.124.243.96
Remote IP : 110.124.243.96
IP 地址数: 8
```

图 3-25 例 3-28 测试代码运行结果

如文件 3-36 所示，第 9～20 行随机产生 10 个 IP 地址。为模拟同一 IP 地址多次访问，第 16～17 行将某次（本例为第 2 次）产生的 IP 地址做备份，随后将该备份地址作为第 9 次和第 10 次的 IP 地址（第 19、20 行）。第 22、23 行将当前系统时间作为 HLL 的键，第 24 行将 IP 地址保存到 HLL 中，执行去重计数。第 25 行设定记录在 Redis 中保存的时长。该测试代码的运行结果如图 3-25 所示。

3.9 Spring 操作 Redis 位图

位图是通过计算机存储数据的最小单位——位来表示某个元素对应的值或者状态的方法。一个位的值或者是 0，或者是 1，也就是说一个位只能表示两种状态。Redis 位图是字符串类型的扩展，可以将字符串视为位向量，进而对一个或多个字符串执行逐位操作。例如，字符串"big"，字母 b、i 和 g 对应的 ASCII 码分别为 0x62(01100010B)、0x69(01101001B)和 0x67(01100111B)，则字符串"big"在 Redis 的存储情况为：

<table>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="8">'b'</td><td colspan="8">'i'</td><td colspan="8">'g'</td></tr>
</table>

由于位图是字符串类型的扩展，因此位图类型的数据也可以使用字符串类型的命令，主要是 SET 和 GET。可以通过 SETBIT 命令设置各位的值，来保存字符串。如，在 Redis 中执行下述命令：

```
redis > SETBIT bk 0 0
(integer) 0
redis > SETBIT bk 1 1
(integer) 0
redis > SETBIT bk 2 1
(integer) 0
redis > SETBIT bk 3 0
```

```
(integer) 0
redis > SETBIT bk 4 0
(integer) 0
redis > SETBIT bk 5 0
(integer) 0
redis > SETBIT bk 6 1
(integer) 0
redis > SETBIT bk 7 0
(integer) 0
redis > GET bk
"b"
```

SETBIT 命令的格式为：SETBIT key offset value。功能：针对 key 存储的字符串值，设置或清除指定偏移量 offset 上的位，位的设置或清除取决于 value 值，即 1 或 0。当 key 不存在时，会创建一个新的字符串。而且这个字符串的长度会伸展，直到可以满足指定的偏移量 offset($0 \leqslant offset < 2^{32}$)，在伸展过程中，新增的位的值被设置为 0。如果想要设置位图的非零初值，一种方式就是将每个位逐个设置为 0 或 1，但是这种方式比较麻烦；另一种方式是可以直接使用 SET 命令存储一个字符串。如准备设置一个 8 位长度的位图的初值，其中的第 2、3、4 位为 1，其余位为 0，即初值为 00111000，此初值为字符'8'的 ASCII 码。因此，可通过下述命令完成初值设置。

```
redis > set mk "8"
OK
redis > GETBIT mk 2
(integer) 1
redis > GETBIT mk 3
(integer) 1
redis > GETBIT mk 4
(integer) 1
redis > GETBIT mk 1
(integer) 0
```

GETBIT 命令的格式为：GETBIT key offset。功能：返回 key 对应的字符串在 offset 位置的位，当 offset 大于值的长度时，返回 0；当 key 不存在时，可以认为 value 为空字符串，此时 offset 肯定大于空字符串长度，返回 0。Redis 中位图相关的常用命令如表 3-5 所示。

表 3-5 Redis 中位图相关的常用命令

命　令	含　义
GETBIT key offset	返回以键(key)存储的字符串值中偏移量(offset)处的位值
SETBIT key key offset value	设置或清除键(key)处存储的字符串值中的偏移位。根据值(value 可以是 1 或 0)设置或清除位
BITCOUNT key [start end]	获取位图指定范围中位值为 1 的个数，如果不指定 start 与 end，则取所有
BITOP op destKey key1 [key2…]	执行多个 BitMap 的 AND(交集)、OR(并集)、NOT(非)、XOR(异或)操作并将结果保存在 destKey 中

位图基于位进行存储，所以具有节省空间、操作快速、方便扩容等优点。位图有很多应用场景，如：

(1) 记录用户在线状态，只需要一个键，将用户 id 作为偏移量。如果用户在线就设置为 1，不在线就设置为 0，3 亿用户只需要 36MB 的空间。命令示例如下：

```
$status = 1;
$redis->setBit('online', $uid, $status);
$redis->getBit('online', $uid);
```

（2）统计活跃用户数，使用时间作为缓存的键，用户 id 作为偏移量。如果当日活跃过就设置为 1，通过简单的计算就可得到用户在某时段的活跃情况。命令示例如下：

```
$status = 1;
$redis->setBit('active_20220708', $uid, $status);
$redis->setBit('active_20220709', $uid, $status);
$redis->bitOp('AND', 'active', 'active_20220708', 'active_20220709');
```

（3）用户签到。假设某网站有 1000 万用户，平均每人每年签到次数为 10 次，如果用关系数据库保存签到数据的话，1 年将产生 1 亿条签到数据。这样就需要保存和处理大量的意义不大的数据。如果将用户每日签到信息保存到 Redis 中，则以位图形式处理就会节约很多空间。如果用户签到以 1 表示，未签到则以 0 表示。在执行按月统计用户签到信息时，只需要 32 位数据，每个用户 4B 即可保存。1000 万用户一年的签到数据仅仅 480MB。如，12 月前 10 天某用户的签到情况如下：除 12 月 3 日、4 日、7 日和 9 日没有签到，其余已签到。签到数据可以用位图表示如下：

1	1	0	0	1	1	0	1	0	1
12-1	12-2	12-3	12-4	12-5	12-6	12-7	12-8	12-9	12-10

要统计该用户在 12 月前 10 天签到的总天数，只要计算位图结构中 1 的数量即可。处理用户签到程序的代码如文件 3-37 所示。

【文件 3-37】 TestBitmapOperations.java

```
1   @RunWith(SpringJUnit4ClassRunner.class)
2   @ContextConfiguration(classes = RedisTemplateConfig.class)
3   public class TestBitmapOperations {
4       @Autowired
5       private StringRedisTemplate template;
6
7       @Test
8       public void testAttendance() throws NoSuchAlgorithmException {
9           String uid = "1001";
10          LocalDateTime now = LocalDateTime.now();
11          String date =
12              now.format(DateTimeFormatter.ofPattern("yyyyMM"));
13          String key = uid + ":" + date;
14          SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
15          random.setSeed(2L);
16          for(int i = 0;i < 31;i++) {
17              int a = random.nextInt(2);
18              template.opsForValue().setBit(key, i, a == 1);
19              System.out.print(a + " ");
20          }
21          List<Long> result = template.opsForValue().bitField(key,
22            BitFieldSubCommands.create().get(BitFieldSubCommands
23              .BitFieldType.unsigned(31)).valueAt(0));
```

```
24            Long num = result.get(0);
25            int count = 0;
26            while (num > 0) {
27            //让这个数字与 1 做与运算,得到数字的最后一个位判断这个数字是否为 0
28               if ((num & 1) == 1)
29                  //如果为 1,则表示签到 1 次
30                  count++;
31              num >>> = 1;
32            }
33            System.out.println("num = " + count);
34         }
35     }
```

如文件 3-37 所示，第 2 行指定了元数据的配置类为 RedisTemplateConfig. class（代码可参考文件 3-6）。第 9～13 行设定存入 Redis 中的键的格式为“用户 id:年月”。第 14、15 行随机产生 1、0 两个数字，用来模拟用户签到数据。第 16～20 行将模拟的签到数据（按每月 31 天计算）存入 Redis 的位图中，同时将签到数据输出到控制台，以便核对。第 21～34 行进行当月签到次数的统计，即计算位图中数字 1 出现的次数。其中，第 22、23 行获取本月的全部签到数据，其返回结果是一组十进制数字。随后，用这个数字的每一位与数字 1 做与运算，根据与运算的结果判定当天是否签到（第 27～33 行）。最后，经过与运算得到的 1 的个数即为签到的天数。程序运行结果如图 3-26 所示。

```
0 1 0 0 1 0 1 1 0 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 0 1 0 1 1 0 0 签到天数 = 16
```

图 3-26 签到统计程序运行结果

3.10 键绑定操作子接口

前面几节分别介绍了 ValueOperations、ListOperations 等操作接口。这些操作接口有一个共性，在进行某个具体操作时，都需要指定键，例如，使用这些操作接口添加值时，如下：

```
1    SetOperations -> add(K key, V... values)
2    ZSetOperations -> add(K key, Set < ZSetOperations.TypedTuple < V >> tuples)
3    ValueOperations -> set(K key, V value)
4    ListOperations -> leftPush(K key, V value)
```

其实在整个操作过程中，可能并没有更换键，只是反复对一个键进行设置、取值、删除操作。于是，RedisTemplate 类提供了一套便捷操作子接口——键绑定操作子接口，这些子接口提前将某个键和操作接口进行绑定。这样，在使用键绑定操作子接口进行操作时，就不需要传递键了。RedisTemplate 类提供的键绑定子接口如表 3-6 所示。

表 3-6 RedisTemplate 类提供的键绑定子接口

子 接 口	说 明
BoundGeoOperations	地理空间数据键绑定操作
BoundHashOperations	哈希键绑定操作
BoundKeyOperations	键绑定操作

续表

子 接 口	说 明
BoundListOperations	列表键绑定操作
BoundSetOperations	集合键绑定操作
BoundValueOperations	字符串(或值)键绑定操作
BoundZSetOperations	有序集合键绑定操作

下面通过一些示例演示怎样利用键绑定操作子接口操作哈希、列表、集合、字符串、有序集合数据类型。

【例 3-29】 利用键绑定子接口操作键为 bound-value 的字符串类型(String)的数据。分别执行添加、修改和追加值操作。测试代码如文件 3-38 所示。

【文件 3-38】 例 3-29 测试代码

```
1    @Test
2    public void testValueBoundOperations() throws InterruptedException{
3      BoundValueOperations<String,String> ops =
4         template.boundValueOps("bound-value");
5      //添加值
6      ops.set("value1");
7      System.out.println(ops.get());
8      //修改值
9      ops.set("value2");
10     System.out.println(ops.get());
11     //追加值
12     ops.append(" append");
13     System.out.println(ops.get());
14     ops.set("value3", Duration.ofMillis(3000));
15     System.out.println(ops.get());
16     Thread.sleep(5000);
17     System.out.println(ops.get());
18   }
```

```
value1
value2
value2 append
value3
null
```

图 3-27 例 3-29 测试代码运行结果

如文件 3-38 所示，第 14 行在设置值 value3 的同时，指定了该值在 Redis 中的缓存时长。第 16 行设置当前程序挂起 5 秒，等待键 bound-value 对应的值 value3 过期，第 17 行再次获取键 bound-value 对应的值。运行此测试代码，运行结果如图 3-27 所示。

【例 3-30】 利用键绑定子接口操作键为 bound-list 的列表类型(List)的数据。分别执行添加、修改和删除操作。测试代码如文件 3-39 所示。

【文件 3-39】 例 3-30 测试代码

```
1    @Test
2    public void testListBoundOperations() {
3       BoundListOperations<String, String> ops =
4          template.boundListOps("bound-list");
5       //添加
6       ops.leftPush("value1");
```

```
7        ops.rightPush("value2");
8        ops.rightPushAll("value3","value4");
9        System.out.println(ops.range(0, - 1));
10       //修改
11       ops.set(0,"new value");
12       System.out.println(ops.range(0, - 1));
13       //删除
14       ops.remove(1,"value3");
15       System.out.println(ops.range(0, - 1));
16   }
```

运行此测试代码,运行结果如图 3-28 所示。

```
[value1, value2, value3, value4]
[new value, value2, value3, value4]
[new value, value2, value4]
```

图 3-28 例 3-30 测试代码运行结果

【例 3-31】 利用键绑定子接口操作键为 bound-hash 的哈希类型的数据。分别执行添加、修改和扫描操作。测试代码如文件 3-40 所示。

【文件 3-40】 例 3-31 测试代码

```
1    @Test
2    public void testHashBoundOperations() {
3      BoundHashOperations < String, String,String > ops =
4         template.boundHashOps("bound - hash");
5      //添加
6      ops.put("k1","v1");
7      Map < String,String > map = new HashMap <>();
8      map.put("k2","v2");  map.put("k3","100");
9      ops.putAll(map);
10     System.out.println(ops.entries()); System.out.println(" -------- ");
11     //修改
12     ops.increment("k3",10);
13     System.out.println("k3 = " + ops.get("k3"));
14     ops.delete("k1");
15     System.out.println(ops.entries()); System.out.println(" -------- ");
16     //扫描
17     Cursor < Map.Entry < String,String >> cursor =
18       ops.scan(ScanOptions.NONE);
19     cursor.forEachRemaining(entry - > System.out.println(
20       entry.getKey() + " = " + entry.getValue()));
21   }
```

运行此测试代码,运行结果如图 3-29 所示。

```
{k1=v1, k3=100, k2=v2}
--------
k3 = 110
{k3=110, k2=v2}
--------
k2=v2
k3=110
```

图 3-29 例 3-31 测试代码运行结果

【例 3-32】 利用键绑定子接口操作键为 bound-set 的集合类型的数据。分别执行添加、移除和扫描操作。测试代码如文件 3-41 所示。

【文件 3-41】 例 3-32 测试代码

```
1    @Test
2    public void testSetBoundOperations() {
3        BoundSetOperations<String,String> ops =
4          template.boundSetOps("bound-set");
5        ops.add("v1","v2","v3","v4","v5");
6        System.out.println(ops.members());
7        System.out.println("--------");
8        ops.remove("v3");
9        Cursor<String> cursor = ops.scan(ScanOptions.NONE);
10       if(cursor != null)
11         cursor.forEachRemaining(e -> System.out.print(e+" "));
12   }
```

```
[v5, v1, v2, v3, v4]
--------
v5 v1 v2 v4
```

图 3-30　例 3-32 测试代码运行结果

运行此测试代码，运行结果如图 3-30 所示。

【例 3-33】 利用键绑定子接口操作键为 bound-zset 的有序集合类型的数据。分别执行添加、移除和扫描操作。测试代码如文件 3-42 所示。

【文件 3-42】 例 3-33 测试代码

```
1    @Test
2    public void testZSetBoundOperations() {
3        BoundZSetOperations<String, String> ops =
4          template.boundZSetOps("bound-zset");
5        ops.add("GuanYu",32.0D);
6        ops.add("LiuBei",21.9D);
7        ops.add("ZhangFei",41.0D);
8        ops.add("CaoCao",100.0D);
9        System.out.println(ops.range(0,-1));
10       ops.remove("CaoCao");
11       System.out.println("--------");
12       Cursor<ZSetOperations.TypedTuple<String>> tups  =
13         ops.scan(ScanOptions.NONE);
14       tups.forEachRemaining(tup -> System.out.println(
15         tup.getValue()+"="+tup.getScore()));
16   }
```

运行此测试代码，运行结果如图 3-31 所示。

```
[LiuBei, GuanYu, ZhangFei, CaoCao]
--------
LiuBei=21.9
GuanYu=32.0
ZhangFei=41.0
```

图 3-31　例 3-33 测试代码运行结果

在电商或社交网络中，经常有自动补全的需求。用户在文本框输入的文字时，会自动弹出一个下拉框，给用户提供候选词，方便输入。例如，在百度主页输入“大学”两个字，会有相应的候选词出现，如图 3-32 所示。

关于自动补全这个功能，可以使用 Redis 的有序集合实现。有序集合会默认将存入的字符串按分数进行升序排列。如果存入的字符串的分数相等，则按照字符串中字母的字典序升序排列。如存入分数相同的三个字符串"aab"、"abb"、"aaba"，则这三个字符串在有序集合中的顺序为"aab"、"aaba"、"abb"。这样，就可以把提供给用户的候选词存放到有序集合中，并且保持这些词的分数相同。那么，要查找带有前缀为 abc 的词，就是查找前缀介于 abbz 和 abd 之间的词，如图 3-33 所示，即锁定图中 p 和 q 的位置。

图 3-32 百度主页提供的候选词

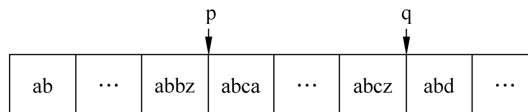

图 3-33 有序集合中存放的候选词

要锁定 p 和 q 的位置，以查找前缀为 abc 的词(不包括 abc 本身)为例，只需要向有序集合中插入 abc\`和 abc{两个元素。因为在 ASCII 编码中，排在字符 a 前面的字符是反引号(\`)字符，排在字符 z 后面的是左花括号({)字符。即 abc\`会排在所有拥有 abca 前缀的词的前面，而位于所有拥有 abbz 前缀的词的后面。同理，abc{会位于所有拥有 abcz 前缀的词的后面，而位于所有拥有 abd 前缀的词的前面。两个元素进入有序集合后，就会占据图 3-33 中 p 和 q 指向的位置，如图 3-34 所示。

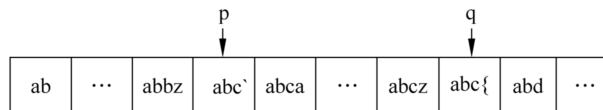

图 3-34 有序集合中插入两个元素后的情况

随后，通过 p 和 q，即 abc\`和 abc{在有序集合中的位置便可确定提供给用户的候选词集合。

【例 3-34】 准备一组候选词，根据用户输入的单词前缀给出最多 10 个候选词，实现自动补全功能。测试代码如文件 3-43 所示。

【文件 3-43】 AutoCompletion. java

```
1    public class AutoCompletion {
2        public Set < String > doAutoCompletion(StringRedisTemplate template,
3            String prefix){
4            String[ ] names = {
5                    "astra","astrid","astrix","athena","athene","atlanta",
6                    "barb","barbara","barbe","barbey","barrie","barry",
7                    "caden","cadesa","cafesse","cagey","caril","carine"
```

```
8           };
9           BoundZSetOperations<String, String> ops =
10              template.boundZSetOps("words");
11          for(int i = 0;i<names.length;i++)
12              ops.add(names[i],0);
13          String p = prefix+"~";
14          String q = prefix+"{";
15          ops.add(p,0);
16          ops.add(q,0);
17          Long begin_index = ops.rank(p);
18          Long end_index = ops.rank(q);
19          if(end_index-begin_index>10)
20              end_index = begin_index+11;
21          Set<String> tips = ops.range(begin_index+1,end_index-1);
22          ops.remove(p);
23          ops.remove(q);
24          return tips;
25      }
26  }
```

如文件 3-43 所示，第 4～8 行将一组候选词存放到 names 数组中。第 9、10 行绑定操作有序集合的键 words。第 11、12 行将 names 数组中存放的候选词加入有序集合，同时设定所有元素的分数为 0。这就意味着所有加入有序集合的候选词会按照字母表顺序升序排列。第 13、14 行根据用户输入的词语前缀生成两个定位符 p 和 q。第 15、16 行将 p、q 加入有序集合。第 17、18 行获取两个定位符在有序集合中的下标（位置），这样，在两个定位符之间的元素组成了返回给用户的候选词集合。第 21 行从有序集合中取出一定数量的候选词，随后删除两个定位符（第 22、23 行）。

根据文件 3-43 中给出的候选词，编写测试代码，从候选词中查找前缀为 ast 的单词作为提供给用户的候选词（最多取出 10 个）。测试代码如文件 3-44 所示。

【文件 3-44】 TestAutoCompletion.java

```
1   @RunWith(SpringJUnit4ClassRunner.class)
2   @ContextConfiguration(classes = RedisTemplateConfig.class)
3   public class TestAutoCompletion {
4       @Autowired
5       private StringRedisTemplate template;
6
7       @Test
8       public void testAutoCompletion(){
9           AutoCompletion app = new AutoCompletion();
10          Set<String> tips = app.doAutoCompletion(template,"ast");
11          tips.forEach(System.out::println);
12      }
13  }
```

运行测试代码，从文件 3-43 提供的候选词中选择出含有 ast 前缀的词，测试结果如图 3-35 所示。

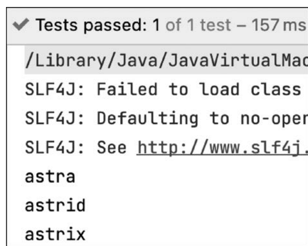

图 3-35 自动补全功能测试结果

3.11 RedisTemplate 类的通用方法

RedisTemplate 类除了提供 Redis 常用数据类型的操作接口外，还提供了操作缓存的方法，如删除键、判断键是否存在、键设置过期时间、键移动等方法。本节将介绍这些方法。

（1）方法原型：List<RedisClientInfo> getClientList()；功能：获取已连接的客户端的信息。

（2）方法原型：Boolean expire(K key,long timeout,TimeUnit unit)；功能：设置指定键 key 的生存时间。

（3）方法原型：Boolean persist(K key)；功能：移除指定键 key 的过期时间；对应的 Redis 命令：PERSIST。

（4）方法原型：void rename(K oldKey,K newKey)；功能：重命名键；对应的 Redis 命令：RENAME。

（5）方法原型：Boolean hasKey(K key)；功能：判断键 key 是否存在；对应的 Redis 命令：EXISTS。

（6）方法原型：Boolean delete(K key)；重载形式：Long delete(Collection<K> keys)；功能：删除指定的键 key(s)；对应的 Redis 命令：DEL。

【例 3-35】 应用上述方法完成以下要求：①获取客户端的信息；②将键 expire-key 的过期时间设置为 3 秒；③移除 expire-key 的过期时间；④将键 no-key 重命名为 has-key，并判断 has-key 和 no-key 是否存在；⑤删除键 delete-key。测试代码如文件 3-45 所示。

【文件 3-45】 例 3-35 测试代码

```
1    @Test
2    public void testCommonOperations1() {
3        //①获取客户端连接信息
4        System.out.println("---- exercise 1 ----");
5        List<RedisClientInfo> list = template.getClientList();
6        if(list != null)
7            list.forEach(System.out::println);
8        //②设置过期时间
9        System.out.println("---- exercise 2 ----");
10       template.opsForValue().set("expire-key","data");
11       template.expire("expire-key", Duration.ofMillis(3000));
12       System.out.println("expire-key exprired within "
13           + template.getExpire("expire-key") + " ms");
```

```
14          //③清理过期时间
15          System.out.println("---- exercise 3 ----");
16          template.persist("expire-key");
17          System.out.println("expire = " + template.getExpire("expire-key"));
18          //④重命名键
19          System.out.println("---- exercise 4 ----");
20          template.opsForValue().set("no-key","value");
21          template.rename("no-key","has-key");
22          //⑤判断键是否存在
23          System.out.println("---- exercise 5 ----");
24          template.opsForValue().set("has-key","data");
25          Boolean flag = template.hasKey("has-key");
26          if(flag != null && flag)
27             System.out.println("键 has-key 存在");
28          else
29             System.out.println("键 has-key 不存在");
30          System.out.println(template.hasKey("no-key")?"键 no-key 存在":
31             "键 no-key 不存在");
32          //⑥删除键
33          System.out.println("---- exercise 6 ----");
34          template.opsForValue().set("delete-key","value");
35          System.out.print("删除前：");
36          System.out.println(template.hasKey("delete-key")?
37             "delete-key 存在":"delete-key 不存在");
38          template.delete("delete-key");
39          System.out.print("删除后：");
40          System.out.println(template.hasKey("delete-key")?
41             "delete-key 存在":"delete-key 不存在");
42      }
```

如文件 3-45 所示，本例使用的 template 是 RedisTemplate<String,String>的实例，其配置代码可参考文件 3-6。运行此测试代码，控制台的输出如图 3-36 所示。

```
---- exercise 1 ----
{sub=0, flags=N, multi=-1, qbuf=0, id=211, addr=127.0.0.1:51841,
 events=r, psub=0, idle=7, qbuf-free=0, oll=0, omem=0, name=, obl=0,
 cmd=command, fd=14, age=7, db=0}
{sub=0, flags=N, multi=-1, qbuf=26, id=212, addr=127.0.0.1:51848,
 events=r, psub=0, idle=0, qbuf-free=32742, oll=0, omem=0, name=,
 obl=0, cmd=client, fd=11, age=0, db=0}
---- exercise 2 ----
expire-key exprired within 3 ms
---- exercise 3 ----
expire = -1
---- exercise 4 ----
键has-key存在
键no-key不存在
---- exercise 5 ----
删除前： delete-key存在
删除后： delete-key不存在
```

图 3-36 例 3-35 测试代码运行后控制台的输出

（7）方法原型：Set<K> keys(K pattern)；功能：查找与给定模式匹配的所有键；对应的 Redis 命令：KEYS。

其中，关于参数 pattern 的定义有以下几种情况：

① 问号(?)匹配符：表示仅匹配一个字母，如 h?llo 可匹配 hello、hallo 和 hxllo 等。

② 星号(*)匹配符：表示匹配 0 个或多个字母，如 h*llo 可匹配 hllo 和 heeeello 等。

③ 列表([])匹配：表示仅匹配列表中的一个字符，如 h[ae]llo 可匹配 hello 和 hallo，但不能匹配 hillo。也可以在列表中指定一个有序序列的首尾字符，则可以匹配包括首尾字符在内的一个字符，如 h[a-b]llo 可匹配 hallo 和 hbllo。此外，可以用“^”符号结合列表符号“[”和“]”实现排除某个字符，如 h[^e]llo 可匹配 hallo、hbllo 等，但不能匹配 hello。

此外，方法原型定义中的返回值类型 Set 中的泛型 K 与 RedisTemplate<K,V>中定义的泛型 K 类型一致。在本节中，已将泛型 K 定义为 String。

【例 3-36】 将<CaoCao,v1>、<CaoPi,v21>、<CaoZhang,v22>和<CaoXiong,v23>分别加入 Redis。要求：①找到 Cao 开头的所有键及其值；②找到 Cao 开头的跟随两个字母的键及其值；③找到 Cao 开头的随后为 P 或 C 的键及其值；④找到以 Cao 开头的随后为 X 或 Z，并以 g 为结尾的键及其值；⑤找到以 Cao 开头的随后不含字母 C 的键及其值。测试代码如文件 3-46 所示。

【文件 3-46】 例 3-36 测试代码

```
1    @Test
2    public void testCommonOperations2() {
3        template.opsForValue().set("CaoCao","v1");
4        template.opsForValue().set("CaoPi","v21");
5        template.opsForValue().set("CaoZhang","v22");
6        template.opsForValue().set("CaoXiong","v23");
7        //1-1 pattern 1 "*"
8        template.keys("Cao*").iterator().forEachRemaining(e->
9            System.out.println(e+" : "+template.opsForValue().get(e)));
10       //1-2 pattern 2 "?"
11       template.keys("Cao??").iterator().forEachRemaining(e->
12           System.out.println(e+" : "+template.opsForValue().get(e)));
13       //1-3 pattern 3 "[PC]*"
14       template.keys("Cao[PC]*").iterator().forEachRemaining(e->
15           System.out.println(e+" : "+template.opsForValue().get(e)));
16       //1-4 pattern 4 "Cao[X-Z]???g" or "Cao[X-Z]*"
17       template.keys("Cao[X-Z]*").iterator().forEachRemaining(e->
18           System.out.println(e+" : "+template.opsForValue().get(e)));
19       //1-5 pattern 5 "Cao[^C]*"
20       template.keys("Cao[^C]*").iterator().forEachRemaining(e->
21           System.out.println(e+" : "+template.opsForValue().get(e)));
22   }
```

运行此测试代码，控制台的输出如图 3-37 所示。

(8) 方法原型：K randomKey()；功能：从键空间中随机返回一个键；对应的 Redis 命令：RANDOMKEY。

(9) 方法原型：DataType type(K key)；功能：确定键存储的数据的类型；对应的 Redis 命令：TYPE。

【例 3-37】 向 key0～key7 共 8 个键中存入 8 个随机选择的字符串，再从此 8 个键中随机取出 3 个键，并判断对应的值的类型。测试代码如文件 3-47 所示。

```
CaoXiong : v23
CaoPi : v21
CaoCao : v1
CaoZhang : v22
--------
CaoPi : v21
--------
CaoPi : v21
CaoCao : v1
--------
CaoXiong : v23
CaoZhang : v22
--------
CaoXiong : v23
CaoPi : v21
CaoZhang : v22
```

图 3-37 例 3-36 测试代码运行后控制台的输出

【文件 3-47】 例 3-37 测试代码

```
1    @Test
2    public void testCommonOperations3() {
3       for(int i = 0;i < 8;i++)
4          template.opsForValue().set(
5             "key" + i,UUID.randomUUID().toString());
6       for(int j = 0;j < 3;j++){
7          String rk = template.randomKey();
8          System.out.println(rk + ":" + template.opsForValue().get(rk));
9          System.out.println("type is:" + template.type(rk).name());
10      }
11   }
```

运行此测试代码，控制台的输出如图 3-38 所示。

```
key5:62277d05-f24f-4a1a-b065-5db948043859
type is : STRING
key6:64b964e9-138c-4503-9cac-77b3d6874c24
type is : STRING
key0:9e760887-1f96-4a83-a0c2-6a03debd9ad5
type is : STRING
```

图 3-38 例 3-37 测试代码运行后控制台的输出

（10）方法原型：Boolean move(K key,int dbIndex)；功能：将键 key 移动到由参数 dbIndex 指定的数据库；对应的 Redis 命令：MOVE；Redis 默认配置中共有 16 个数据库，索引为 0～15。在 Redis 命令提示符下可以用

```
select dbIndex
```

命令切换不同的数据库。不同数据库之间的数据没有任何关联，甚至可以存在相同的键。

（11）方法原型：byte[]dump(K key)；功能：执行数据备份；对应的 Redis 命令：DUMP。该方法可执行 Redis 的转储命令并返回结果。Redis 使用非标准的序列化机制并包含校验和信息，因此该方法返回原始字节，而没有使用 ValueSerializer 进行反序列化。可使用转储的返回值(byte[])作为要备份的值的参数。

（12）方法原型：public void restore(K key,byte[] value,long timeToLive,TimeUnit unit,boolean replace)；功能：执行 Redis 还原命令。其中，参数 key 指定要还原的键，即目

标键；参数 value 指定转储对象返回的要还原的值；参数 timeToLive 指定目标键的到期时间，0 表示无到期时间；参数 replace 指定为 true 表示替换可能存在的值，默认值 false。对应的 Redis 命令：RESTORE。

【例 3-38】 完成如下两个要求：①将 move-key 键移动到索引为 4 的数据库；②将 backup-key 键备份到 backup-key-bak 键，并设置键 backup-key-bak 的生存时间为 30 分钟。测试代码如文件 3-48 所示。

【文件 3-48】 例 3-38 测试代码

```
1   @Test
2   public void testCommonOperations4() {
3     template.opsForValue().set("move - key","data");
4     System.out.println("移动前" + template.opsForValue()
5      .get("move - key"));
6     template.move("move - key",4);
7     System.out.println("移动后" + template.opsForValue()
8       .get("move - key"));
9     template.opsForValue().set("backup - key","this is important");
10    byte[] backup = template.dump("backup - key");
11    if(backup != null) {
12      template.restore("backup - key - bak", backup, 30,
13           TimeUnit.MINUTES);
14      System.out.println("backup data: " +
15           template.opsForValue().get("backup - key - bak"));
16      }
17  }
```

执行移动操作后(第 6 行)，move-key 从 0 号数据库被移动到 4 号数据库。当再次执行获取值操作时(第 7、8 行)，返回 null。此时，在 Redis 客户端执行 select 4 命令，可以在 4 号数据库找到被移动过来的 move-key 键：

```
redis > select 4
OK
redis[4]> keys *
1) "move - key"
```

运行此测试代码，控制台的输出如图 3-39 所示。

```
移动前data
移动后null
backup data: this is important
```

图 3-39 例 3-38 测试代码运行后控制台的输出

(13) 方法原型：List<V> sort(SortQuery<K> query)；功能：对要查询的元素进行排序；对应的 Redis 命令：SORT。

【例 3-39】 对给定的 8 位散文家(SuXun，HanYu，Wang'anShi，SuShi，LiuZongYuan，SuZhe，OuYangXiu，ZengGong)按名字的字母序升序排列，输出排序后的前三位。测试代码如文件 3-49 所示。

【文件 3-49】 例 3-39 测试代码

```
1   @Test
2   public void testCommonOperations3() {
3      String sortKey = "sortKey";
4      String[] names = {"SuXun","HanYu","Wang'anShi","SuShi",
5         "LiuZongYuan","SuZhe","OuYangXiu","ZengGong"};
```

```
6        String[] marks = {"1009","768","1021","1037",
7           "773","1039","1007","1019"};
8        template.delete(sortKey);
9        if (!template.hasKey(sortKey)) {
10          for (int i = 0; i < 8; i++) {
11             template.boundSetOps(sortKey).add(String.valueOf(i));
12             String hashKey = "hash" + i,
13                pid = String.valueOf(i),
14                pname = names[i],
15                pmark = marks[i];
16             template.boundHashOps(hashKey).put("id", pid);
17             template.boundHashOps(hashKey).put("name", pname);
18             template.boundHashOps(hashKey).put("mark", pmark);
19             System.out.printf("%s:{\"_id\":%s,\"Name\":
20                  %s,\"Mark\",%s}\n",hashKey, pid, pname, pmark);
21          }
22       }
23       SortQuery<String> sortQuery = SortQueryBuilder.sort(sortKey)
24          .by("hash*->name")
25          .alphabetical(true)
26          .limit(new SortParameters.Range(0,3))
27          .order(SortParameters.Order.ASC)
28          .get("hash*->id")
29          .get("hash*->name")
30          .get("hash*->mark").build();
31       System.out.println("----  ----");
32       List<String> list = template.sort(sortQuery);
33       for(int j = 0;j < list.size();j += 3)
34          System.out.printf("{\"ID\": %s, \"Name\": %s, \"Mark\", %s}\n",
35                list.get(j), list.get(j + 1), list.get(j + 2));
36    }
```

排序运行原理可以了解 Redis 的 SORT 命令（见 1.4.1 节）。运行此测试代码，控制台的输出如图 3-40 所示。

```
hash0 : {"_id": 0, "Name": SuXun, "Mark", 1009}
hash1 : {"_id": 1, "Name": HanYu, "Mark", 768}
hash2 : {"_id": 2, "Name": Wang'anShi, "Mark", 1021}
hash3 : {"_id": 3, "Name": SuShi, "Mark", 1037}
hash4 : {"_id": 4, "Name": LiuZongYuan, "Mark", 773}
hash5 : {"_id": 5, "Name": SuZhe, "Mark", 1039}
hash6 : {"_id": 6, "Name": OuYangXiu, "Mark", 1007}
hash7 : {"_id": 7, "Name": ZengGong, "Mark", 1019}
---- ----
{"ID": 1, "Name": HanYu, "Mark", 768}
{"ID": 4, "Name": LiuZongYuan, "Mark", 773}
{"ID": 6, "Name": OuYangXiu, "Mark", 1007}
```

图 3-40　例 3-39 测试代码运行后控制台的输出

在 RedisTemplate 中，定义了几个 execute()方法，这些方法是 RedisTemplate 的核心方法。RedisTemplate 中很多其他方法均是通过调用 execute()方法来执行具体的操作。表 3-7 列举了 execute()方法的 6 种重载形式。

表 3-7 RedisTemplate 定义的 execute()方法

方法原型	说明
<T> T execute(RedisCallback<T> action)	在 Redis 连接中执行给定的操作
<T> T execute(RedisCallback<T> action, boolean exposeConnection)	在 Redis 连接中执行给定的操作,参数 exposeConnection 表示是否要暴露当前连接,如果为 true,那么就可以在回调函数中使用当前连接对象
<T> T execute(RedisCallback<T> action, boolean exposeConnection,boolean pipeline)	在连接中执行给定的操作,参数 exposeConnection 的含义同上,参数 pipeline 表示是否开启流水线(见 5.3 节)
<T> T execute(RedisScript<T> script,List<K> keys,Object... args)	执行给定的 Redis 脚本(Redis Script)
<T> T execute(RedisScript<T> script, RedisSerializer<?> argsSerializer, RedisSerializer<T> resultSerializer, List<K> keys,Object... args)	执行给定的 Redis 脚本,使用提供的 RedisSerializer 序列化脚本参数和结果
<T> T execute(SessionCallback<T> session)	执行 Redis 会话

使用 RedisTemplate 直接调用 opsFor**()方法来操作 Redis 时,每执行一条命令时都要重新获取一个连接,因此很耗资源。可以调用 execute()方法,让一个连接直接执行多次 Redis 操作语句。

【例 3-40】 利用一个 Redis 连接完成字符串数据的存取操作。测试代码如文件 3-50 所示。

【文件 3-50】 TestExecuteMethod.java

```
1   @RunWith(SpringJUnit4ClassRunner.class)
2   @ContextConfiguration(classes = ExecuteMethodConfig.class)
3   public class TestExecuteMethod {
4       @Autowired
5       private RedisTemplate<String,String> template;
6       @Test
7       public void testExecuteByRedisCallback(){
8           template.execute((RedisCallback<Object>) connection -> {
9               connection.stringCommands().set("key".getBytes(),
10                      "hello,redis".getBytes());
11              byte[] res = connection.stringCommands()
12                      .get("key".getBytes());
13              System.out.println(new String(res));
14              return null;
15          });
16      }
17  }
```

如文件 3-50 所示,第 8 行指定传递给 execute()方法的参数为 RedisCallback(接口)类型的对象。而 RedisCallback 接口中只定义了一个 doInRedis(RedisConnection connection)方法。因此,第 9~12 行可调用 RedisConnection 接口中定义或继承的方法。RedisConnection 接口代表了一个与 Redis 服务器的连接,它是各种 Redis 客户端(或驱动程序)的抽象。并且,RedisConnection 接口继承了 RedisStringCommands、RedisKeyCommands 等众多 Redis 操作接口,可以编程形式完成绝大部分的 Redis 操作。第 9 行调用 RedisConnection 接口的

stringCommands()方法获取 RedisStringCommands 接口类型对象，并调用 RedisStringCommands 接口的 Boolean set(byte[]key，byte[]value)和 byte[]get(byte[]key)方法完成字符串存取操作。运行此代码，控制台输出 hello，redis 字符串。

【例 3-41】 利用一个 Redis 会话完成字符串数据的存取操作。测试代码如文件 3-51 所示。

【文件 3-51】 例 3-41 测试代码

```
1   @Test
2   public void testExecuteBySessionCallback(){
3       template.execute(new SessionCallback < Object >(){
4           public String execute(RedisOperations operations)
5               throws DataAccessException {
6               BoundValueOperations < String, String > bops =
7                   operations.boundValueOps("key3");
8               bops.set("hello,world");
9               System.out.println(bops.get());
10              return null;
11          }
12      });
13  }
```

运行此测试代码，控制台输出 hello，world 字符串。用于执行 Redis 脚本的 execute()方法的案例见 5.4.3 节。

3.12 序列化和反序列化

虽然 Redis 支持各种数据类型，但 Redis 中存储的数据只有字节。因此，要利用 Redis 存储 Java 程序中的对象，就要将对象转换为字节数组或字符串再保存到 Redis 中。将对象转换为可传输(或可存储)的字节序列或字符串的过程称为序列化；将字节序列或字符串还原为对象的过程称为反序列化。进行序列化就是为了对象能够通过网络传输和跨平台存储。不同的计算机系统能够识别和处理的数据的通用格式是二进制数据(或纯文本数据)。因此，需要将对象按照一定规则转换为字节数组或字符串才能实现跨平台传输和存储的目的，这就是序列化。当需要对象时，再按这个规则把对象还原出来，这就是反序列化。作为键值型数据库，Redis 写入数据时，可以分别指定键和值的序列化机制。此外，还可以使用 Redis 哈希类型来实现更复杂的结构化对象映射，Spring Data Redis 提供了将 Java 对象映射到哈希的各种策略。本节将分别介绍键值序列化机制和对象-哈希序列化机制。

3.12.1 内置序列化器

将 Java 对象存储到 Redis 中时，需要进行序列化操作，如将 Java 对象序列化为 JSON 字符串。此时，Redis 中保存的内容为序列化后的 JSON 字符串。同样，如果要将序列化后的 JSON 字符串从 Redis 中取出，再转换为存储前的 Java 对象则需要反序列化操作，将 JSON 字符串转换为 Java 对象。在 Spring Data Redis 中，Java 对象和二进制数据之间的转换(反之亦然)可利用 org.springframework.data.redis.serializer 包(以下简称为 serializer

包)中提供的接口或类处理。

1. 序列化器

该包提供了两种类型的序列化器。

(1) 基于 RedisSerializer 接口的双向序列化器。

(2) 使用 RedisElementReader 接口和 RedisElementWriter 接口的元素读写器。

以上两种序列化器的主要区别是 RedisSerializer 接口主要序列化为字节数组(byte[]),而读写器使用字节缓冲区(ByteBuffer)。

2. RedisSerializer 接口的实现类

在 Spring Data Redis 中,serializer 包提供了以下几个 RedisSerializer 接口的实现类。

(1) JdkSerializationRedisSerializer 类:使用 JDK 的序列化器,将对象通过 ByteArrayOutputStream 类和 ByteArrayInputStream 类进行序列化和反序列化,最终 Redis 中将存储字节序列。该类是 RedisCache 类和 RedisTemplate 类默认的序列化器。限制:被序列化的类需要实现 Serializable 接口,而且 Redis 中存储的数据很不直观,序列化后的内容为十六进制数字或乱码。

(2) StringRedisSerializer 类:在键或值为字符串时,该类根据指定的字符集将字符串转换为字节序列(byte[]),也可以执行字节序列到字符串的反序列化。该类通过 String 类的 String.getBytes(Charset charset)和 String(byte[]byte,Charset charset)方法实现序列化和反序列化,是最轻量级和最高效的序列化器。

(3) Jackson2JsonRedisSerializer 类:使用 Jackson 和 Jackson Databind ObjectMapper 读取和写入 JSON 的序列化器。该序列化器可用于绑定到类型化的 Bean 或非类型化的 HashMap 实例。注意,空对象被序列化为空数组,反之亦然。

(4) GenericJackson2JsonRedisSerializer 类:使用 Jackson 实现的序列化器。该类实现 Java 对象到 JSON 字符串的序列化,以及 JSON 字符串到 Java 对象的反序列化。使用该类执行序列化时,会保存序列化的对象的包名和类名,反序列化时以包名和类名作为标识就可以还原成指定的对象。

(5) GenericToStringSerializer 类:该序列化器使用 Spring 的 ConversionService,使用默认的字符集 UTF-8 将对象转换为字符串,从而完成序列化。反之亦然。

(6) OxmSerializer 类:可实现对象与 XML 之间的相互转换。使用此序列化器,编程将会有些难度,而且效率最低,不建议使用。此外,该序列化器需要 Spring-OXM 模块的支持。

3. RedisTemplate 序列化相关的方法

RedisTemplate 类提供的与序列化相关的方法如表 3-8 所示。

表 3-8 RedisTemplate 类提供的与序列化相关的方法

方法原型	说明
RedisSerializer <?> getDefaultSerializer()	获取当前模板(RedisTemplate)默认的序列化器
setDefaultSerializer(RedisSerializer <?> serializer)	设置用于当前模板的默认序列化器
RedisSerializer <?> getHashKeySerializer()	返回哈希键序列化器
void setHashKeySerializer(RedisSerializer <?> hashKeySerializer)	设置哈希键序列化器

续表

方法原型	说　明
RedisSerializer <?> getHashValueSerializer()	返回哈希值序列化器
void setHashValueSerializer(RedisSerializer <?> hashValueSerializer)	设置哈希值序列化器
RedisSerializer <?> getKeySerializer()	返回当前模板使用的键序列化器
void setKeySerializer(RedisSerializer <?> serializer)	设置当前模板使用的键序列化器
RedisSerializer < String > getStringSerializer()	返回字符串序列化器
void setStringSerializer(RedisSerializer < String > stringSerializer)	设置字符串序列化器
RedisSerializer <?> getValueSerializer()	返回当前模板使用的值序列化器
void setValueSerializer(RedisSerializer <?> serializer)	设置当前模板使用的值序列化器

4. RedisSerializer 接口的实现类的应用

下面结合例子介绍 serializer 包提供的几个 RedisSerializer 接口的实现类的应用。

（1）JdkSerializationRedisSerializer。

JdkSerializationRedisSerializer 序列化器是 RedisCache 类和 RedisTemplate 类默认的序列化器，采用 Java 语言的序列化机制。该序列化器将对象保存成二进制格式，执行序列化的效率不是最差的，但结果的可读性较差。下面的例子演示如何利用 JdkSerializationRedisSerializer 序列化器将一个对象进行序列化。

第一步，创建一个类 Barrel，代码如文件 3-52 所示。

【文件 3-52】 Barrel. java

```
1   @Data
2   public class Barrel implements Serializable {
3     private String material;
4     private double capacity;
5     //此处省略了 toString()方法
6   }
```

第二步，配置 RedisTemplate，代码如文件 3-53 所示。

【文件 3-53】 RedisSerializerConfig. java

```
1    @Configuration
2    public class RedisSerializerConfig {
3      //RedisConnectionFactory 配置元数据定义省略，代码见文件 3 - 6
4      @Bean
5      public RedisTemplate < String,Object > redisTemplate(
6          RedisConnectionFactory factory) {
7        RedisTemplate < String,Object > redisTemplate =
8          new RedisTemplate <>();
9        redisTemplate.setConnectionFactory(factory);
10       //指定值的默认的序列化器 JdkSerializationRedisSerializer
11       redisTemplate.setValueSerializer(RedisSerializer.java());
12       return redisTemplate;
13     }
14   }
```

第三步，编写测试代码，如文件 3-54 所示。

【文件 3-54】 TestSerializer.java

```
1    @RunWith(SpringJUnit4ClassRunner.class)
2    @ContextConfiguration(classes = RedisSerializerConfig.class)
3    public class TestSerializer {
4        @Autowired
5        private RedisTemplate<String, Object> template;
6        @Test
7        public void testJdkSerializer(){
8            Barrel barrel = new Barrel();
9            barrel.setMaterial("plastic");
10           barrel.setCapacity(60);
11           template.opsForValue().set("jdk",barrel);
12           System.out.println(template.opsForValue().get("jdk"));
13       }
14   }
```

运行此测试代码后，将 Barrel 对象序列化后保存到 Redis。控制台的输出为 Barrel{material='plastic', capacity=60.0}。利用 RedisInsight 客户端查看 Redis 保存的内容，如图 3-41 所示。

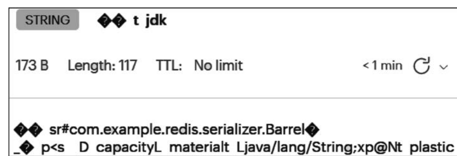

图 3-41 Redis 中保存的内容 1

（2）Jackson2JsonRedisSerializer。

为了解决图 3-41 中出现的乱码问题，可以替换默认的序列化接口实现机制。例如，使用 Jackson2JsonRedisSerializer 序列化器。Jackson2JsonRedisSerializer 是可以使用 Jackson 读取和写入 JSON 的序列化器。当要存储的值为字符串类型时，也可以采用 StringRedisSerializer 序列化器。本例采用 Jackson2JsonRedisSerializer 对 Barrel 对象进行序列化。

第一步，修改 RedisTemplate 类的默认序列化器，可利用表 3-8 中的 setKeySerializer() 方法和 setValueSerializer() 方法分别指定键和值的序列化器，以替代默认的序列化器。部分代码如文件 3-55 所示。

【文件 3-55】 RedisSerializerConfig.java

```
1    @Bean
2    public RedisTemplate<String,Object> redisTemplate(
3       RedisConnectionFactory factory) {
4      RedisTemplate<String,Object> template = new RedisTemplate<>();
5      template.setConnectionFactory(factory);
6      template.setKeySerializer(RedisSerializer.string());
7      template.setValueSerializer(new
8         Jackson2JsonRedisSerializer<Object>(Object.class));
9      template.afterPropertiesSet();
10     return template;
11   }
```

如文件 3-55 所示，第 6 行指定键的序列化器为 StringRedisSerializer，字符编码为默认的 UTF-8，第 7、8 行指定值的序列化器为 Jackson2JsonRedisSerializer。

第二步，引入 jackson-databind 依赖。当前的 Jackson2JsonRedisSerializer 并不要求持久化类（本例中为 Barrel 类）显式实现 Serializable 接口。运行文件 3-54 的测试代码后，控制台的输出为{material = plastic，capacity=60.0}，利用 RedisInsight 客户端查看 Redis 保存的内容，如图 3-42 所示。

图 3-42　Redis 中保存的内容 2

（3）GenericJackson2JsonRedisSerializer。

该序列化器可以将 Java 对象序列化为 JSON 字符串，以及 JSON 字符串反序列化为 Java 对象。要使用 GenericJackson2JsonRedisSerializer 序列化器，可对文件 3-55 稍作修改，部分代码如文件 3-56 所示。

【文件 3-56】　RedisSerializerConfig.java 部分修改

```
1    @Bean
2    public RedisTemplate<String,Object> redisTemplate(
3      RedisConnectionFactory factory) {
4        RedisTemplate<String,Object> redisTemplate =
5          new RedisTemplate<>();
6        redisTemplate.setConnectionFactory(factory);
7        redisTemplate.setKeySerializer(RedisSerializer.string());
8        redisTemplate.setValueSerializer(RedisSerializer.json());
9        redisTemplate.afterPropertiesSet();
10       return redisTemplate;
11   }
```

GenericJackson2JsonRedisSerializer 序列化器并不要求持久化类显式实现 Serializable 接口。运行文件 3-54 中的测试代码后，控制台输出为 Barrel{material='plastic',capacity=60.0}，利用 RedisInsight 客户端查看 Redis 保存的内容，如图 3-43 所示。对比控制台输出和 Redis 中保存的内容可知，与 StringRedisSerializer 和 Jackson2JsonRedisSerializer 序列化器不同，在从 Redis 中获取键对应的值时，GenericJackson2JsonRedisSerializer 序列化器执行了反序列化操作。由 JSON 字符串反序列化为 Java 对象时，要求持久化类（本例中为 Barrel 类）提供公有的无参数构造方法。

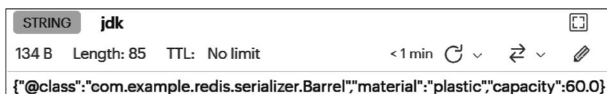

图 3-43　Redis 中保存的内容 3

3.12.2　HashMapper 接口

HashMapper 接口是 Spring Data Redis 提供的实现 Java 对象和 Redis Hash（Redis 的哈希类型）之间转换的核心接口。Redis Hash 一般具有如下结构：

```
Key:{
     filed: value,
     filed: value,
     filed: value,
     ....
}
```

这个结构和Java中的对象非常相似，但是不能按照Java对象的结构直接存储进Redis Hash。因为Java对象中的字段(field)是可以嵌套的，而Redis Hash不支持嵌套结构。为此，Spring Data Redis提供了将Java对象映射到Redis Hash的三种策略。

（1）直接映射。可以使用HashOperations接口和相关的序列化程序，如Jackson2JsonRedisSerializer，进行直接映射。

（2）使用Redis仓库。Redis仓库可以应用定制的映射策略转换和存储Redis Hash中的域对象，并且可以使用辅助索引(见8.5节)。

（3）使用HashMapper接口和HashOperations接口。

本节主要介绍上述方法(3)执行Java对象与Redis Hash的映射。Spring Data Redis提供了HashMapper接口的三个实现类，分别为BeanUtilsHashMapper、ObjectHashMapper和Jackson2HashMapper。下面结合例子分别介绍ObjectHashMapper类和Jackson2HashMapper类的使用方法。

1. ObjectHashMapper类

电商网站中经常有这样的需求：记录用户注册时填写的基础信息及其注册地址信息，实现这一需求的具体步骤如下。

第一步，创建两个持久化类Person和Address，分别封装用户的基础信息和注册地址信息，代码如文件3-57和文件3-58所示。

【文件3-57】 Person.java

```
1    @Data
2    public class Person {
3        private String personId;
4        private String firstname;
5        private String lastname;
6        private Address address;
7        private LocalDateTime localDateTime;
8    }
```

【文件3-58】 Address.java

```
1    @Data
2    public class Address {
3        private String city;
4        private String country;
5        //此处省略了带参数的构造方法
6    }
```

第二步，编写配置类，代码如文件3-59所示。

【文件3-59】 HashMapperConfig.java

```
1    public class HashMapperConfig {
2        //此处省略了连接工厂的配置,内容见文件3-6
3        @Bean
4        public RedisTemplate<String,Map<byte[],byte[]>>
5            hashMapperRedisTemplate(
```

```
6          RedisConnectionFactory factory) {
7          RedisTemplate<String, Map<byte[],byte[]>> template =
8             new RedisTemplate<>();
9          template.setConnectionFactory(factory);
10         //设置键序列化方式
11         template.setKeySerializer(RedisSerializer.string());
12         //设置简单类型值的序列化方式
13         template.setValueSerializer(RedisSerializer.byteArray());
14         //设置哈希类型键的序列化方式
15         template.setHashKeySerializer(RedisSerializer.byteArray());
16         //设置哈希类型值的序列化方式
17         template.setHashValueSerializer(RedisSerializer.byteArray());
18         template.afterPropertiesSet();
19         return template;
20      }
21   }
```

如文件3-59所示，第11行将键的序列化器指定为StringRedisSerializer，默认的字符编码为UTF-8。第12～17行分别将字符串类型值、哈希键和哈希值的序列化器指定为ByteArrayRedisSerializer，该序列化器是一个只使用字节数组（byte[]）的原始序列化器（RedisSerializer），代码如下：

```
1    package org.springframework.data.redis.serializer;
2    enum ByteArrayRedisSerializer implements RedisSerializer<byte[]> {
3        INSTANCE;
4
5        @Nullable
6        @Override
7        public byte[] serialize(@Nullable byte[] bytes)
8           throws SerializationException {
9             return bytes;
10       }
11
12       @Nullable
13       @Override
14       public byte[] deserialize(@Nullable byte[] bytes)
15          throws SerializationException {
16            return bytes;
17       }
18   }
```

第三步，建立person对象与Redis Hash的映射，编写的测试代码如文件3-60所示。

【文件3-60】 TestHashMapper.java

```
1    @RunWith(SpringJUnit4ClassRunner.class)
2    @ContextConfiguration(classes= HashMapperConfig.class)
3    public class TestHashMapper {
4       @Autowired
5       private RedisTemplate<String, Map<byte[],byte[]>> redisTemplate;
6
```

```
7       private final HashMapper<Object, byte[], byte[]> hashMapper =
8         new ObjectHashMapper();
9
10      @Test
11      public void save(){
12         Person person = new Person();
13         person.setPersonId("person-address");
14         person.setFirstname("Simth");
15         person.setLastname("Qiong");
16         person.setLocalDateTime(LocalDateTime.now());
17         person.setAddress(new Address("DaLian","China"));
18         Map<byte[], byte[]> map = hashMapper.toHash(person);
19         HashOperations<String,byte[],byte[]> ops =
20           redisTemplate.opsForHash();
21         ops.putAll(person.getPersonId(),map);
22         map.entrySet().iterator().forEachRemaining(entry ->
23            System.out.println(new String(entry.getKey()) + " = " +
24            new String(entry.getValue())));
25      }
26   }
```

如文件 3-60 所示，第 11 行开始的测试用例是实现将 person 对象映射为 Redis Hash 并存入 Redis。因此，需要调用 HashMapper 接口的 toHash()方法。该方法的原型为：

```
Map<K,V> toHash(T object)
```

其功能是将 Java 对象（本例中为 person 对象）转换为可以被 Redis Hash 使用的映射（第 18 行）。同时，指定了 Redish Hash 的字段和值的类型均为 byte[]（字节数组）。与此对应的是第 7、8 行，在创建 ObjectHashMapper 的实例时，指明 HashMapper 的三个泛型，第一个为 Java 对象的类型，第二个和第三个为 Redis Hash 中的字段和值的类型。第 19～21 行将 person 对象映射成的 map 对象存入 Redis，并且约定，该 map 对象的键的类型为 String。作为测试，第 22～24 行将转换后的 map 对象的内容在控制台输出。运行此测试代码后，控制台输出和 Redis 中保存的内容分别如图 3-44 和图 3-45 所示。

```
_class=com.example.redis.hashmapper.entity.Person
address.city=DaLian
address.country=China
firstname=Simth
lastname=Qiong
localDateTime=2023-02-03T16:15:26.742473400
personId=person-address
```

图 3-44　控制台输出的内容

Field	Value
_class	com.example.redis.hashmapper...
address.city	DaLian
address.country	China
firstname	Simth
lastname	Qiong
localDateTime	2023-02-03T16:15:26.742473400
personId	person-address

图 3-45　Redis 中保存的内容 4

第四步，验证 Redis Hash 向 Java 对象的（反向）映射。可调用 HashMapper 接口的 fromHash()方法实现（反向）映射。该方法的原型为：

```
T fromHash(Map<K,V> hash)
```

可在文件 3-60 的基础上增加一个测试用例，代码如下：

```
1    @Test
2    public void find() {
3      HashOperations<String,byte[],byte[]> ops =
4         redisTemplate.opsForHash();
5      Map<byte[],byte[]> map = ops.entries("person-address");
6      Person person = (Person)hashMapper.fromHash(map);
7      System.out.println(new Gson().toJson(person));
8    }
```

运行此测试用例，可在控制台看到以JSON字符串形式输出的person对象，如图3-46所示。

```
{"personId":"person-address","firstname":"Simth","lastname":"Qiong",
"address":{"city":"DaLian","country":"China"},
"localDateTime":{"date":{"year":2023,"month":2,"day":3},
"time":{"hour":16,"minute":15,"second":26,"nano":742473400}}}
```

图3-46　控制台输出的JSON字符串

2. Jackson2HashMapper类

Jackson2HashMapper类通过使用Faster XML Jackson为Java对象提供Redis Hash映射。Jackson2HashMapper可以将顶级属性映射为哈希字段名，还可以选择将结构扁平化。扁平化是指为所有嵌套属性创建单独的哈希项（字段和值），并尽可能将复杂类型解析为简单类型。以文件3-57定义的持久化类为例，数据在非扁平化映射中的显示方式如表3-9所示。

表3-9　数据在非扁平化映射中的显示方式

字　段	值
firstname	Jon
lastname	Snow
address	{ "city" ："Castle Black"，"country" ："The North" }
localDateTime	2018-01-02T12:13:14

经过扁平化处理后，显示方式如表3-10所示。

表3-10　数据在扁平化映射中的显示方式

字　段	值
firstname	Jon
lastname	Snow
address.city	Castle Black
address.country	The North
localDateTime	2018-01-02T12:13:14

对于文件3-57定义的Person类的对象，也可以用Jackson2JsonRedisSerializer类完成其与Redis Hash的映射。

第一步，可以修改文件3-59，配置RedisTemplate的相应的序列化器。部分代码如文件3-61所示。

【文件 3-61】 HashMapperConfig.java 的部分修改

```
1   @Bean
2   public RedisTemplate<String, Map<String,Object>>
3     hashMapperRedisTemplate(RedisConnectionFactory factory) {
4       RedisTemplate<String,Map<String,Object>> redisTemplate =
5           new RedisTemplate<>();
6       redisTemplate.setConnectionFactory(factory);
7       // 设置键序列化方式
8       redisTemplate.setKeySerializer(RedisSerializer.string());
9       redisTemplate.setValueSerializer(new
10        Jackson2JsonRedisSerializer<Object>(Object.class));
11      redisTemplate.setHashKeySerializer(new
12        Jackson2JsonRedisSerializer<Object>(Object.class));
13      redisTemplate.setHashValueSerializer(new
14        Jackson2JsonRedisSerializer<Object>(Object.class));
15      redisTemplate.afterPropertiesSet();
16      return redisTemplate;
17  }
```

第二步，执行 person 对象到 Redis Hash 的映射，部分测试代码如文件 3-62 所示。

【文件 3-62】 部分测试代码

```
1   @Test
2   public void save(){
3     Person person = new Person();
4     person.setPersonId("person-address");
5     person.setFirstname("Simth");
6     person.setLastname("Qiong");
7     person.setLocalDateTime(LocalDateTime.now());
8     person.setAddress(new Address("DaLian","China"));
9     HashMapper<Object, String, Object> hashMapper =
10      new Jackson2HashMapper(true);
11    Map<String, Object> map = hashMapper.toHash(person);
12    HashOperations<String,String,Object> ops =
13      redisTemplate.opsForHash();
14    ops.putAll(person.getPersonId(),map);
15    map.entrySet().iterator().forEachRemaining(entry ->
16      System.out.println(entry.getKey() + " = " + entry.getValue()));
17  }
```

如文件 3-62 所示，第 9、10 行实例化 Jackson2HashMapper 映射器，该映射器的构造方法有布尔型参数，取值为 true 意味着采用扁平化方式处理数据；反之取值为 false。此外，Jackson2HashMapper 类实现的 HashMapper<K,T,V>接口中，已将 K、T 和 V 三个泛型指定为 Object、String、Object。即该映射器的定义为：

```
1   public class Jackson2HashMapper implements HashMapper<Object, String,
2     Object>, HashObjectReader<String, Object> {
3         ……
4   }
```

因此，实例化该映射器时必须沿用该映射器的泛型定义。为了便于处理 LocalDateTime 类型的数据，需要在 pom.xml 文件中增加依赖：

```
1  <dependency>
2      <groupId>com.fasterxml.jackson.datatype</groupId>
3      <artifactId>jackson-datatype-jsr310</artifactId>
4      <version>2.13.3</version>
5  </dependency>
```

第三步，执行此测试代码，控制台的输出如图 3-47 所示。

```
localDateTime=2023-02-03T20:27:58.7118286
firstname=Simth
@class=com.example.redis.hashmapper.entity.Person
address.city=DaLian
personId=person-address
lastname=Qiong
address.@class=com.example.redis.hashmapper.entity.Address
address.country=China
```

图 3-47　控制台的输出

第四步，执行 Redis Hash 到 person 对象的映射。部分测试代码如文件 3-63 所示。

【文件 3-63】部分测试代码

```
1   @Test
2   public void find() {
3     HashOperations<String,String,Object> ops =
4       redisTemplate.opsForHash();
5     Map<String,Object> map = ops.entries("person-address");
6     HashMapper<Object, String, Object> hashMapper =
7       new Jackson2HashMapper(true);
8     Person person = (Person)hashMapper.fromHash(map);
9     System.out.println(new Gson().toJson(person));
10  }
```

注意，为保证反序列化能够成功执行，需要 Person 类和 Address 类都提供公有的无参数构造方法。

3.13　小结

本章着重介绍了 Spring Data Redis 提供的 RedisTemplate 类。RedisTemplate 类是在 RedisConnection 接口(org.springframework.data.redis.connection.RedisConnection)基础上，将 Redis 操作进行了更高层次的封装。由于 RedisTemplate 来自于 Spring，因此可以在程序中利用 IoC、AOP 等 Spring 的特性优雅地、简单地操作 Redis。本章涉及的操作 Redis 的常用类和接口的关系如图 3-48 所示。

对于操作 Redis，可以使用 Lettuce、Jedis、Redisson 等客户端，也可以利用 Spring 框架的 RedisConnectionFactory 接口创建与 Redis 的连接，再使用 RedisTemplate 类操作 Redis。图 3-48 只列举了 RedisTemplate 类的一些方法。如果在 Redis 中保存的键和值都是字符串类型，也可以使用 RedisTemplate 类的子类 StringRedisTemplate。

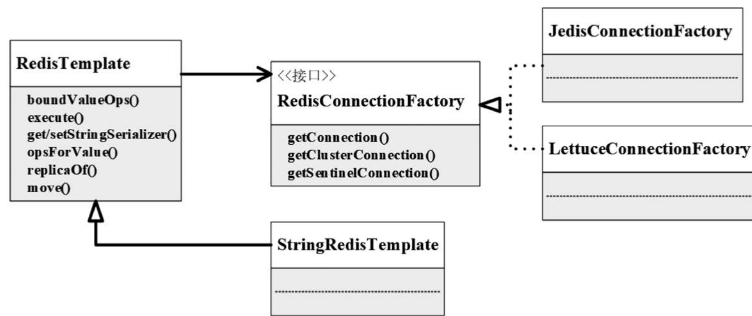

图 3-48 操作 Redis 的常用类和接口的关系

作为 Spring Data Redis 提供的模板类，RedisTemplate 提供了大量的 API 用以封装 Redis 操作。如，数据的序列化和反序列化操作、执行 Lua 脚本(见 5.4.2 节)、操作 Redis 分片集群(见 7.3.3 节)等。同时 RedisTemplate 类还提供了操作字符串、列表、哈希等数据结构的专属操作接口。可以说，RedisTemplate 类是使用 Spring 开发 Redis 应用的首选。