Why Spring Data Redis？

首先Spring Data Redis是Spring框架提供的用于操作Redis的客户端。

Spring框架是一个全栈Java程序框架，通过DI、AOP和便携的服务抽象提供一个轻量的容器和非侵入编程模型支持。

NoSQL存储为传统的关系型数据库提供了横向扩展和速度上的替代，Key-Value存储是目前NoSQL领域的主要成员。

Spring Data Redis（SDR）框架通过消除冗余的、重复的集成代码，使Spring程序能简单的使用Redis的Key-Value存储。

Spring Data Redis目前支持两种驱动:Jedis和Lettuce，可以看成是这两种驱动的统一封装，以高度统一的形式屏蔽了底层驱动的操作细节，向用户提供一种统一的API，大致结构如下：



Spring Data Redis与Spring Data家族的其他数据操作如：Spring Data JDBC，以类似的结构进行封装，降低用户的学习成本；只要使用过其中一种Spring Data技术，对其他的数据源操作能依葫芦画瓢的进行，封装的整体结构如下：



将封装的层次分为4部分:

1. 底层的工厂层和连接层,这里会依据驱动的不同而有不同的实现;连接层可以直接与Redis Server进行交互，采用二进制数据和命令进行。
2. 模板层是对连接层的进一步封装，对Redis的每种数据类型，都封装一个操作类，与Redis Server进行交互，如String类型的ValueOperations；而RedisTemplate持有所有类型操作的引用，所以可以直接通过定义RedisTemplate对Redis进行操作；RedisTemplate封装了通用的操作，如删除key，设置过期时间，对不同数据类型的操作通过具体的操作类进行。
3. 键绑定层是对模板层的更进一步封装，每个类型的实例只能操作单一的键值(实例化时已经指定了键)

Spring Data Redis的系统要求：Spring Data Redis 1.x 🡺 JDK 1.6 + ,Redis 2.6.x +

常用的Redis Java客户端有：Jedis、Lettuce、Redison、Spring Data Redis，对应的学习资料如下：

Jedis：<https://github.com/xetorthio/jedis/wiki>

Lettuce：<https://github.com/lettuce-io/lettuce-core/wiki>

Redison：<https://github.com/redisson/redisson>

SPR：

<https://docs.spring.io/spring-data/redis/docs/2.0.2.RELEASE/reference/html/>

更多客户端参照：<http://www.redis.net.cn/clients/#java>

基于Jedis的Spring Data Redis使用入门：底层API示例

使用底层API：RedisConnectionFactory和RedisConnection可以直接操作Redis，下面是一个简单的例子：

1. Maven依赖

<properties>

<jedis.version>2.9.0</jedis.version>

<spring.redis.version>2.0.0.RELEASE</spring.redis.version>

</properties>

<dependencies>

<!-- jedis -->

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>${jedis.version}</version>

</dependency>

<!-- spring-data-redis -->

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-redis</artifactId>

<version>${spring.redis.version}</version>

</dependency>

</dependencies>

1. 配置redis.properties

redis.host=127.0.0.1

redis.port=6379

1. 配置Spring-data-redis.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<!-- 读取properties文件 -->

<bean id=*"propertyConfigurerForProject1"*

class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>

<property name=*"order"* value=*"1"* />

<property name=*"systemPropertiesModeName"* value=*"SYSTEM\_PROPERTIES\_MODE\_OVERRIDE"* />

<property name=*"ignoreResourceNotFound"* value=*"true"* />

<property name=*"ignoreUnresolvablePlaceholders"* value=*"true"*/>

<property name=*"locations"*>

<list>

<value>classpath\*:redis.properties</value>

</list>

</property>

</bean>

<!-- Jedis ConnectionFactory -->

<bean id=*"jedisConnectionFactory"* class=*"org.springframework.data.redis.connection.jedis.JedisConnectionFactory"*

p:host-name=*"${redis.host}"*

p:port=*"${redis.port}"* />

</beans>

1. 操作Redis

**public** **static** **void** main( String[] args )

{

ApplicationContext context = **new** ClassPathXmlApplicationContext("classpath:spring-data-redis.xml");

// 直接通过连接进行操作

JedisConnectionFactory jedisConnectionFactory = context.getBean(JedisConnectionFactory.**class**);

RedisConnection redisConnection = jedisConnectionFactory.getConnection();

// 设置值

redisConnection.set("name".getBytes(),"wangdh".getBytes());

// 获取值

**byte**[] value = redisConnection.get("name".getBytes());

System.***out***.println(**new** String(value));

}

基于Jedis的Spring Data Redis使用入门：RedisTemplate示例

使用底层API：RedisConnection操作Redis，需要对数据进行手动转换(String 🡨🡪byte)，需要进行多数重复性工作，效率低下；

org.springframework.data.redis.core.RedisTemplate类提供了与Redis交互的高级抽象，该类负责序列化和连接管理，对大多数重复性工作进行了封装。

Remark：RedisTemplate是一个线程安全的类。

此外，模板还提供了用于处理特定类型或特定键的操作视图。包含的接口见下表：

|  |  |
| --- | --- |
| 接口 | 描述 |
| ValueOperations | 操作String类型 |
| ListOperations | 操作List类型 |
| SetOperations | 操作Set类型 |
| ZSetOperations | 操作ZSet类型 |
| HashOperations | 操作Hash类型 |
| HyperLogLogOperations | 操作HyperLogLog类型 |
| GeoOperations | 操作Geo类型 |
| BoundValueOperations | 操作String类型特定key |
| BoundListOperations | 操作List类型特定key |
| BoundSetOperations | 操作Set类型特定key |
| BoundZSetOperations | 操作ZSet类型特定key |
| BoundHashOperations | 操作Hash类型特定key |
| BoundGeoOperations | 操作Geo类型特定key |

例子：上一节第三步的配置文件添加下面内容：

<bean id=*"redisTemplate"* class=*"org.springframework.data.redis.core.RedisTemplate"*>

<property name=*"connectionFactory"* ref=*"jedisConnectionFactory"*/>

</bean>

测试RedisTemplate：

// RedisTemplate 操作

RedisTemplate redisTemplate = context.getBean(RedisTemplate.**class**);

// 获取String类型的操作类

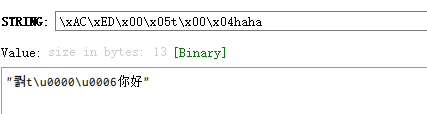
ValueOperations<String, String> valueOperations = redisTemplate.opsForValue();

valueOperations.set("haha", "你好");

System.***out***.println(valueOperations.get("haha"));

基于Jedis的Spring Data Redis使用入门：数据序列化

RedisTemplate默认使用的是基于JDK的序列化器，所以存储在Redis的数据如果不经过相应的反序列化，看到的结果是这个样子的：



可以看到，出现了乱码，在程序层面上，不会影响程序的运行，但当出现数据错误，对数据进行排查时，就无从下手了。

在Spring Data Redis中，用户自定义类型和存储数据之间的转换（反之亦然）由org.springframework.data.redis.serializer包下的类进行处理。

这个包包含两种类型的序列化程序，它们负责序列化过程：

* 基于RedisSerializer的双向序列化器
* 通过RedisElementReader和RedisElementWriter进行元素的读写(2.0引进的)

这两种类型的序列化器主要区别在于，RedisSerializer主要序列化为byte[]，RedisElementReader/RedisElementWriter使用ByteBuffer。

基于RedisSerializer的序列化器有多种实现，可以直接使用，如：

* StringRedisSerializer
* JdkSerializationRedisSerializer
* OxmSerializer(依赖于Spring OXM)
* Jackson2JsonRedisSerializer/GenericJackson2JsonRedisSerializer(依赖于jackson)

序列化的范围

序列化可以应用到Key，Value，HashKey，HashValue上，在RedisTemplate中，有四个属于与之对应：

keySerializer、valueSerializer、hashKeySerializer、hashValueSerializer

基于Json的序列化

引入Jackson的maven依赖：

<jackson.databind.version>2.8.5</jackson.databind.version>

<!-- jackson-databind -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>${jackson.databind.version}</version>

</dependency>

配置序列化器：

<!-- 缓存序列化方式 -->

<!--对key的默认序列化器。默认值是StringSerializer -->

<bean id=*"stringRedisSerializer"*

class=*"org.springframework.data.redis.serializer.StringRedisSerializer"* />

<!--是对value的默认序列化器，默认值是取自DefaultSerializer的JdkSerializationRedisSerializer。 -->

<bean id=*"genericJackson2JsonRedisSerializer"*

class=*"org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer"* />

重新配置RedisTemplate

<bean id=*"redisTemplate"* class=*"org.springframework.data.redis.core.RedisTemplate"*>

<property name=*"connectionFactory"* ref=*"jedisConnectionFactory"*/>

<property name=*"keySerializer"* ref=*"stringRedisSerializer"* />

<property name=*"valueSerializer"* ref=*"stringRedisSerializer"*/>

<property name=*"hashKeySerializer"* ref=*"stringRedisSerializer"*/>

<property name=*"hashValueSerializer"* ref=*"genericJackson2JsonRedisSerializer"* />

</bean>

重新运行上一节的示例，得到如下结果：



Redis连接池配置

完善redis.properties配置文件：

#redis的服务器地址  
redis.host=127.0.0.1  
#redis的服务端口  
redis.port=6379  
#客户端超时时间单位是毫秒  
redis.timeout=100000  
#最大建立连接等待时间  
redis.maxWaitMillis=1000  
#最小空闲数  
redis.minIdle=5  
#最大空闲数  
redis.maxIdle=20  
#最大连接数  
redis.maxTotal=100

xml配置

<!-- Jedis连接池的配置对象 -->  
<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">  
 <!--最大空闲数 -->  
 <property name="maxIdle" value="${redis.maxIdle}" />  
 <!--最小空闲数 -->  
 <property name="minIdle" value="${redis.minIdle}" />  
 <!--最大连接数 -->  
 <property name="maxTotal" value="${redis.maxTotal}" />  
 <property name="testOnBorrow" value="true" />  
 <!--最大建立连接等待时间 -->  
 <property name="maxWaitMillis" value="${redis.maxWaitMillis}" />  
</bean>  
  
<!--jedis服务器信息 -->  
<bean id="jedisShardInfo" class="redis.clients.jedis.JedisShardInfo">  
 <constructor-arg index="0" value="${redis.host}" />  
 <constructor-arg index="1" value="${redis.port}" type="int" />  
 <constructor-arg index="2" value="${redis.timeout}"  
 type="int" />  
</bean>  
  
<!--jedis连接池 -->  
<bean id="shardedJedisPool" class="redis.clients.jedis.ShardedJedisPool">  
 <constructor-arg index="0" ref="jedisPoolConfig" />  
 <constructor-arg index="1">  
 <list>  
 <ref bean="jedisShardInfo" />  
 </list>  
 </constructor-arg>  
</bean>  
  
<!-- Redis连接工厂 -->  
<bean id="jedisConnectionFactory"  
 class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">  
 <property name="shardInfo" ref="jedisShardInfo" />  
 <property name="poolConfig" ref="jedisPoolConfig" />  
</bean>

Redis集群和哨兵配置

哨兵配置

为了处理高可用性的Redis，可以使用RedisSentinelConfiguration支持Redis Sentinel。

/\*\*

\* jedis

\*/

@Bean

public RedisConnectionFactory jedisConnectionFactory() {

RedisSentinelConfiguration sentinelConfig = new RedisSentinelConfiguration()

.master("mymaster")

.sentinel("127.0.0.1", 26379)

.sentinel("127.0.0.1", 26380);

return new JedisConnectionFactory(sentinelConfig);

}

可以配置的属性：

spring.redis.sentinel.master：master节点的名字

spring.redis.sentinel.nodes：主机:端口的逗号分隔列表

集群配置

@Bean  
public JedisConnectionFactory redisConnectionFactory() {  
 JedisConnectionFactory connectionFactory = new JedisConnectionFactory();  
 // 集群配置  
 RedisClusterConfiguration redisClusterConfiguration = new RedisClusterConfiguration();  
 RedisNode redisNode7000 = new RedisNode("10.1.21.4", 7000);  
 RedisNode redisNode7001 = new RedisNode("10.1.21.4", 7001);  
 RedisNode redisNode7002 = new RedisNode("10.1.21.4", 7002);  
 RedisNode redisNode7003 = new RedisNode("10.1.21.4", 7003);  
 RedisNode redisNode7004 = new RedisNode("10.1.21.4", 7004);  
 RedisNode redisNode7005 = new RedisNode("10.1.21.4", 7005);  
 redisClusterConfiguration.addClusterNode(redisNode7000);  
 redisClusterConfiguration.addClusterNode(redisNode7001);  
 redisClusterConfiguration.addClusterNode(redisNode7002);  
 redisClusterConfiguration.addClusterNode(redisNode7003);  
 redisClusterConfiguration.addClusterNode(redisNode7004);  
 redisClusterConfiguration.addClusterNode(redisNode7005);  
 connectionFactory = new JedisConnectionFactory(redisClusterConfiguration);  
  
 // 设置连接池配置  
 connectionFactory.setPoolConfig(jedisConfig());  
  
 // 设置数据库，默认0  
 connectionFactory.setDatabase(2);  
  
 // 设置使用连接池  
 connectionFactory.setUsePool(true);  
  
 return connectionFactory;  
}

xml配置和JavaCode配置，根据个人喜好选择

自1.7开始，Spring Data Redis针对Redis的三种模式：单机/集群/哨兵，分别提供了单独的配置类：RedisStandaloneConfiguration(2.x引入)、RedisClusterConfiguration、RedisSentinelConfiguration；

并且自2.0开始，废弃了JedisConnectionFactory的HostName，Port，Password等属性的配置。

最新的配置示例如下：

单机模式

/\*\*

\* 连接工厂类

\* **@return**

\*/

@Bean

**public** JedisConnectionFactory jedisConnectionFactory(){

// 单机配置

RedisStandaloneConfiguration standaloneConfig = **new** RedisStandaloneConfiguration();

standaloneConfig.setHostName("127.0.01");

standaloneConfig.setPort(6379);

JedisConnectionFactory connectionFactory = **new** JedisConnectionFactory(standaloneConfig);

**return** connectionFactory;

}

集群模式与哨兵模式参考上面的配置。

Spring Data Redis操作字符串

// 注入模板操作实例

@Autowired

**private** RedisTemplate template;

// 从模板中取出对应的操作类实例

@Resource(name = "redisTemplate")

**private** ValueOperations valueOps;

由于存储在Redis中的键和值通常是java.lang.String，因此Redis模块为RedisConnection和RedisTemplate提供了两个扩展，分别是StringRedisConnection（及其DefaultStringRedisConnection实现）和StringRedisTemplate(相当于RedisTemplate<String, String>)：

org.springframework.data.redis.core.StringRedisTemplate

**public** **class** StringRedisTemplate **extends** RedisTemplate<String, String> {

/\*\*

\* Constructs a new <code>StringRedisTemplate</code> instance. {@link #setConnectionFactory(RedisConnectionFactory)}

\* and {@link #afterPropertiesSet()} still need to be called.

\*/

**public** StringRedisTemplate() {

RedisSerializer<String> stringSerializer = **new** StringRedisSerializer();

setKeySerializer(stringSerializer);

setValueSerializer(stringSerializer);

setHashKeySerializer(stringSerializer);

setHashValueSerializer(stringSerializer);

}

/\*\*

\* Constructs a new <code>StringRedisTemplate</code> instance ready to be used.

\*

\* **@param** connectionFactory connection factory for creating new connections

\*/

**public** StringRedisTemplate(RedisConnectionFactory connectionFactory) {

**this**();

setConnectionFactory(connectionFactory);

afterPropertiesSet();

}

**protected** RedisConnection preProcessConnection(RedisConnection connection, **boolean** existingConnection) {

**return** **new** DefaultStringRedisConnection(connection);

}

}

实际就是继承自RedisTemplate<String, String>并且对应的序列化器是StringRedisSerializer。

Spring Data Redis操作Hash

对像在Redis中存储为hash类型，可以有两种方式，将每个对象实例作为一个hash进行存储，则对象的每个属性作为hash的field；同种类型的对象实例存储为一个hash，每个对象分配一个field，将对象序列化后，作为该field的值；

假设有Person类：

public class Person {

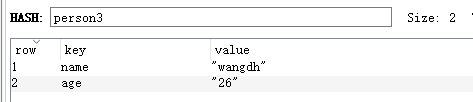
private String name;

private String age;

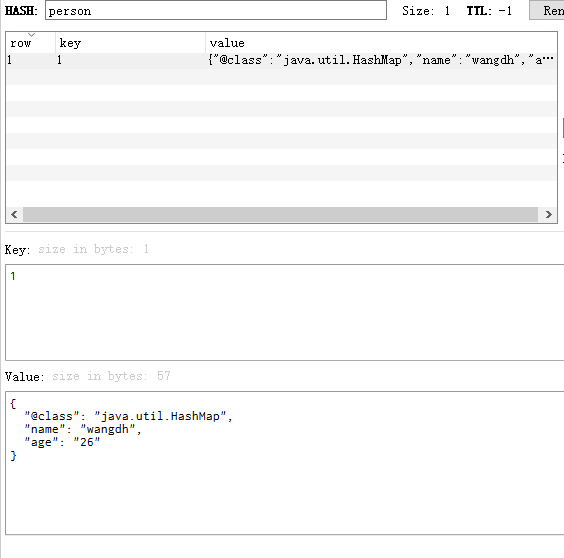
……

}

第一种方式存储的结果为：



第二种方式存储的结果为：



因此，Spring Data Redis处理hash类型数据一般有两种策略：

* 使用HashOperations和Json序列化器直接映射
* 使用HashMapper和HashOperations

@Autowired

**private** RedisTemplate template;

@Resource(name = "redisTemplate")

**private** HashOperations<String, String, Object> hashOps;

**public** **void** test(){

// 使用HashOperations和json序列化器，存储hash

Map<String, String> map = **new** HashMap<String, String>();

map.put("name", "wangdh");

map.put("age", "26");

// 会将map进行json序列化，存储为field为1，对应的值

hashOps.put("person", "1", map);

// 使用HashMapper和HashOperations，存储hash

// 存储的数据会多出一个键：@class，标识序列化的类型

HashMapper<Object, String, Object> mapper = **new** Jackson2HashMapper(**false**);

Map<String, Object> mappedHash = mapper.toHash(map);

hashOps.putAll("person2",mappedHash);

// 直接将map作为hash存储

hashOps.putAll("person3",map);

}

HashMappers

HashMappers使对象和Map <K，V>进行相互转换，有多种默认的实现方式可供使用：

* 基于Spring’s BeanUtils的BeanUtilsHashMapper
* 使用Object to Hash Mapping的ObjectHashMapper(使用JDK序列化器)
* 使用FasterXML Jackson的Jackson2HashMapper(常用，使用Json序列化器)

Jackson2HashMapper可以将数据顶级属性映射为哈希字段名称，并可选地将结构展平。 简单的类型映射到简单的值。复杂类型（嵌套对象，集合，映射）表示为嵌套JSON。

展平会为所有嵌套属性创建单独的哈希条目，并尽可能将复杂类型解析为简单类型。

展平的形式：

展开的形式：

{

"firstname":"Jon",

"lastname":"Snow",

"address.city":"Castle Black",

"address.country":"The North"

}

不展平的形式：

{

"firstname":"Jon",

"lastname":"Snow",

"address":

{

"city" : "Castle Black",

"country" : "The North"

}

}

Redis作为Spring Cache的后端存储

SPR使用1.8.2.RELEASE，2.X版本配置不友好

关于Spring Cache的相关内容参照《Spring Cache.docx》

该缓存的实现方式是将数据存储为String类型，对象存成value，则需要进行相应的序列化配置，建议配置json序列化。

配置类：需要启动缓存配置

**package** com.wangdh.spring.data.redis.demo.config;

**import** org.springframework.beans.factory.annotation.Configurable;

**import** org.springframework.cache.CacheManager;

**import** org.springframework.cache.annotation.EnableCaching;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.data.redis.cache.RedisCacheManager;

**import** org.springframework.data.redis.connection.RedisConnectionFactory;

**import** org.springframework.data.redis.connection.jedis.JedisConnectionFactory;

**import** org.springframework.data.redis.core.RedisTemplate;

**import** org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer;

**import** org.springframework.data.redis.serializer.StringRedisSerializer;

/\*\*

\* Spring Cache 配置

\*

\* **@author** PC

\*

\*/

@Configurable

@EnableCaching

@ComponentScan

**public** **class** SpringCacheConfig {

/\*\*

\* String序列化器

\*

\* **@return**

\*/

@Bean

**public** StringRedisSerializer stringRedisSerializer() {

**return** **new** StringRedisSerializer();

}

/\*\*

\* Json序列化器

\*

\* **@return**

\*/

@Bean

**public** GenericJackson2JsonRedisSerializer genericJackson2JsonRedisSerializer() {

**return** **new** GenericJackson2JsonRedisSerializer();

}

/\*\*

\* 连接工厂类

\*

\* **@return**

\*/

@Bean

**public** JedisConnectionFactory jedisConnectionFactory() {

// 单机配置

JedisConnectionFactory connectionFactory = **new** JedisConnectionFactory();

connectionFactory.setHostName("127.0.0.1");

connectionFactory.setPort(6379);

connectionFactory.setUsePool(**true**);

**return** connectionFactory;

}

/\*\*

\* Redis操作模板

\*

\* **@return**

\*/

@Bean

**public** RedisTemplate<?, ?> redisTemplate(RedisConnectionFactory redisConnectionFactory) {

RedisTemplate<?, ?> redisTemplate = **new** RedisTemplate();

redisTemplate.setConnectionFactory(redisConnectionFactory);

redisTemplate.setKeySerializer(stringRedisSerializer());

redisTemplate.setHashKeySerializer(stringRedisSerializer());

redisTemplate.setValueSerializer(genericJackson2JsonRedisSerializer());

redisTemplate

.setHashValueSerializer(genericJackson2JsonRedisSerializer());

**return** redisTemplate;

}

@Bean

**public** CacheManager cacheManager(RedisTemplate<?, ?> redisTemplate) {

RedisCacheManager cacheManager = **new** RedisCacheManager(redisTemplate);

cacheManager.setUsePrefix(**true**);

**return** cacheManager;

}

}

数据缓存：

@Service

@CacheConfig(cacheNames="wangdh:test")

**public** **class** SpringCacheDemo {

@Cacheable(key="#id")

**public** Person get(String id){

System.***out***.println("query from db...");

Person person = **new** Person();

person.setName("wangdh" + id);

person.setAge("26");

Address address = **new** Address();

address.setCity("GZ");

address.setCountry("ZH");

person.setAddress(address);

**return** person;

}

}

程序运行测试：

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** AnnotationConfigApplicationContext(SpringCacheConfig.**class**);

SpringCacheDemo springCacheDemo = context.getBean(SpringCacheDemo.**class**);

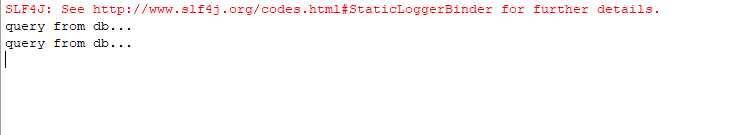
springCacheDemo.get("123");

springCacheDemo.get("123");

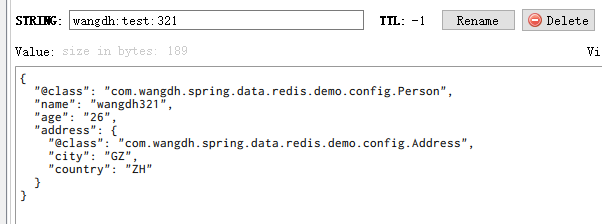
springCacheDemo.get("321");

}

结果：



只输出两次，一次从缓存中获取数据，缓存数据结果为：



RedisTemplate

|  |  |
| --- | --- |
| 接口 | 描述 |
| ValueOperations | 操作String类型 |
| ListOperations | 操作List类型 |
| SetOperations | 操作Set类型 |
| ZSetOperations | 操作ZSet类型 |
| HashOperations | 操作Hash类型 |
| HyperLogLogOperations | 操作HyperLogLog类型 |
| GeoOperations | 操作Geo类型 |
| BoundValueOperations | 操作String类型特定key |
| BoundListOperations | 操作List类型特定key |
| BoundSetOperations | 操作Set类型特定key |
| BoundZSetOperations | 操作ZSet类型特定key |
| BoundHashOperations | 操作Hash类型特定key |
| BoundGeoOperations | 操作Geo类型特定key |

默认情况下，RedisTemplate基于Java进行数据序列化；

单机配置：

集群配置：

哨兵配置：