Redis

配置lettuce连接器

Lettuce是一个基于Netty的开源连接器，由Spring Data Redis通过org.springframework.data.redis.connection.lettuce包支持。以下示例显示如何创建新的Lettuce连接工厂

```
@Configuration
class AppConfig {

  @Bean
  public LettuceConnectionFactory redisConnectionFactory() {

    return new LettuceConnectionFactory(new RedisStandaloneConfiguration("server", 6379));
  }
}
```

还有一些lettuce的连接参数可以调整。默认情况下，由LettuceConnectionFactory创建的所有LettuceConnection实例都为所有非阻塞和非事务操作共享相同的线程安全本机连接。要每次使用专用连接，请将ShareNativeConnection设置为false。如果shareNativeConnection设置为false，也可以将lettuceConnectionFactory配置为使用lettucePool进行池阻塞和事务连接或所有连接。

lettuce与netty的本地传输相集成，允许您使用Unix域套接字与redis通信。确保包含与运行时环境匹配的适当本机传输依赖项。下面的示例演示如何在/var/run/redis.sock为UNIX域套接字创建lettuce连接工厂：

```
@Configuration
class AppConfig {

  @Bean
  public LettuceConnectionFactory redisConnectionFactory() {

    return new LettuceConnectionFactory(new RedisSocketConfiguration("/var/run/redis.sock"));
  }
}
```

配置Jedis连接器

jedis是一个社区驱动的连接器，由Spring Data Redis模块通过org.springframework.data.redis.connection.jedis包支持。在最简单的形式中，Jedis配置如下所示：

```
@Configuration
class AppConfig {

  @Bean
  public JedisConnectionFactory redisConnectionFactory() {
    return new JedisConnectionFactory();
  }
}
```

同时你还想设置密码等host

Redis Sentinel支持

对于处理高可用性的Redis，春季数据Redis的具有用于支撑Redis的哨兵，使用`RedisSentinelConfiguration`，作为显示在下面的例子：

```
/**
 * 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);
}

/**
 * Lettuce
 */
@Bean
public RedisConnectionFactory lettuceConnectionFactory() {
  RedisSentinelConfiguration sentinelConfig = new RedisSentinelConfiguration()
  .master("mymaster")
  .sentinel("127.0.0.1", 26379)
  .sentinel("127.0.0.1", 26380);
  return new LettuceConnectionFactory(sentinelConfig);
}
```

**	**`RedisSentinelConfiguration`也可以使用a定义`PropertySource`，它允许您设置以下属性：配置属性`spring.redis.sentinel.master`：主节点的名称。`spring.redis.sentinel.nodes`：逗号分隔的主机：端口对列表。**

有时，需要与其中一个Sentinels直接交互。使用`RedisConnectionFactory.getSentinelConnection()`或`RedisConnection.getSentinelCommands()`允许您访问配置的第一个活动Sentinel。

通过RedisTemplate处理对象

大多数用户可能会使用`RedisTemplate`它及其相应的包`org.springframework.data.redis.core`。事实上，该模板是Redis模块的核心类，因为它具有丰富的功能集。该模板为Redis交互提供了高级抽象。虽然`RedisConnection`提供了接受和返回二进制值（`byte`数组）的低级方法，但模板负责序列化和连接管理，使用户无需处理这些细节。

此外，模板提供操作视图（在Redis命令参考的分组之后），提供丰富的，通用的接口，用于处理特定类型或某些密钥（通过`KeyBound`接口），如下表所述：

接口	描述
密钥类型操作	
`GeoOperations`	Redis的地理空间操作的，比如`GEOADD`，`GEORADIUS`...
`HashOperations`	Redis哈希操作
`HyperLogLogOperations`	Redis的HyperLogLog操作，例如`PFADD`，`PFCOUNT`，...
`ListOperations`	Redis列表操作
`SetOperations`	Redis设置了操作
`ValueOperations`	Redis字符串（或值）操作
`ZSetOperations`	Redis zset（或排序集）操作
键绑定操作	
`BoundGeoOperations`	Redis键绑定地理空间操作
`BoundHashOperations`	Redis散列键绑定操作
`BoundKeyOperations`	Redis键绑定操作
`BoundListOperations`	Redis列出键绑定操作
`BoundSetOperations`	Redis设置键绑定操作
`BoundValueOperations`	Redis字符串（或值）键绑定操作
`BoundZSetOperations`	Redis zset（或有序集）键绑定操作

配置完成后，模板是线程安全的，可以跨多个实例重用。

`RedisTemplate`在大多数操作中使用基于Java的序列化程序。这意味着模板编写或读取的任何对象都通过Java进行序列化和反序列化。您可以更改模板上的序列化机制，Redis模块提供了多个实现，这些实现在`org.springframework.data.redis.serializer`包中提供。有关更多信息，请参阅序列化器 您还可以将任何序列化程序设置为null，并通过将`enableDefaultSerializer`属性设置为将RedisTemplate与原始字节数组一起使用`false`。请注意，模板要求所有键都为非null。但是，只要底层序列化程序接受它们，值就可以为null。阅读每个序列化程序的Javadoc以获取更多信息。

对于需要特定模板视图的情况，请将视图声明为依赖项并注入模板。容器会自动执行转换，从而消除`opsFor[X]`调用，如以下示例所示：

```
<?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">

  <bean id="jedisConnectionFactory"
class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory" p:use-pool="true"/>
  <!-- redis template definition -->
  <bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate" p:connection-factory-
ref="jedisConnectionFactory"/>
  ...

</beans>
```

```
public class Example {

  // inject the actual template
  @Autowired
  private RedisTemplate<String, String> template;

  // inject the template as ListOperations
  @Resource(name="redisTemplate")
  private ListOperations<String, String> listOps;

  public void addLink(String userId, URL url) {
    listOps.leftPush(userId, url.toExternalForm());
  }
}
```

RedisTemplate的使用功能：

RedisTemplate是操作redis的一个API:先看下RedisTemplate的原码:具体的方法使看原码中的注释

RedisTemplate帮助程序类，用于简化redis数据访问代码。

在给定对象和redis存储中的基础二进制数据之间执行自动序列化/反序列化。默认情况下，它使用对象的Java序列化（通过JDKSerialIdaseReDeSerialZER）。对于字符串密集型操作，请考虑使用专用的StringRedisTemplate.

中心方法是执行execute()，支持redis访问代码实现redisallback接口。它提供了redis connection处理，这样，无论是redisallback实现还是调用代码都不需要显式地关心检索/关闭redis连接，或者处理连接生命周期异常。对于典型的单步动作，有各种方便的方法。

一旦配置，这个类就是线程安全的。

请注意，虽然模板是通用的，但要正确地将给定对象转换为二进制数据或从二进制数据转换为给定对象，则取决于序列化程序/反序列化程序。

文章中的最主要的方法如下：

在Redis连接中执行给定的操作。只要可能，操作对象引发的应用程序异常就会传播到调用方（只能取消选中）。Redis异常被转换为适当的DAO异常。允许返回结果对象，即域对象或域对象集合。对适合于redis存储的二进制数据的给定对象执行自动序列化/反序列化。注意：回调代码不应该处理事务本身！使用适当的事务管理器。通常，回调代码不能接触任何连接生命周期方法，比如close，以让模板完成其工作。 public T execute(RedisCallback action, boolean exposeConnection, boolean pipeline) {}

执行一个事务，使用默认的redisserializer来反序列化任何属于byte[]s或byte[]s或元组的集合或映射的结果。其他结果类型（long、boolean等）与转换后的结果相同。如果在RedisonConnectionFactory中禁用了Tx结果的转换，则将返回exec的结果，而不进行反序列化。此检查主要是为了与1.0向后兼容。

public List exec() {} 执行事务，使用提供的redisserializer反序列化任何字节为[]s的结果或字节为[]s的集合。如果结果是映射，则提供的redisserializer将用于键和值。其他结果类型（long、boolean等）与转换后的结果相同。元组结果将自动转换为typedtuples。 public List exec(RedisSerializer<?> valueSerializer) { }

删除 public Boolean delete(K key) {}

删除集合中的key public Long delete(Collection keys) { } 确定给定的键是否存在。 public Boolean hasKey(K key) { } 为给定的密钥设置生存时间。 public Boolean expire(K key, final long timeout, final TimeUnit unit) { }

为给定的密钥设置生存时间。 public Boolean expireAt(K key, final Date date) {}

将给定消息发布到给定通道。 public void convertAndSend(String channel, Object message) { }

以秒为单位获得钥匙的生存时间。 public Long getExpire(K key) { }

获取生存时间并将其转换为给定的时间单位。 public Long getExpire(K key, final TimeUnit timeUnit) { }

查找与给定模式匹配的所有键。 public Set keys(K pattern) { }

从给定的密钥中删除过期。 public Boolean persist(K key) {}

将给定的键移动到具有索引的数据库。 public Boolean move(K key, final int dbIndex) { }

从键空间返回随机键。 public K randomKey() {}

重命名key public void rename(K oldKey, K newKey) { }

仅当newkey不存在时，才将key olename重命名为newkey。 public Boolean renameIfAbsent(K oldKey, K newKey) {}

确定存储在键处的类型。 public DataType type(K key) { }

执行redis dump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，因此返回原始字节，而不是使用valueserializer进行反序列化。使用dump的返回值作为要还原的值参数 public byte[] dump(K key) { }

执行redis还原命令。传入的值应该是从dump（object）返回的准确序列化数据，因为redis使用非标准序列化机制。 public void restore(K key, final byte[] value, long timeToLive, TimeUnit unit) {} 标记事务块的开始命令将排队，然后可以通过调用redisperations.exec（）或使用redisperations.discard（）回滚来执行。 public void multi() { }

放弃在redisperations.multi（）之后发出的所有命令。 public void discard() { }

在用redisperations.multi（）启动的事务期间，观察给定的键是否有修改。 public void watch(K key) { }

刷新所有以前的重读操作。观察（对象）键。

public void unwatch() { }

排序：

为查询对元素排序。 public List sort(SortQuery query) { return sort(query, valueSerializer); }

关闭由客户端中给定的ip:port标识的给定客户端连接。

public void killClient(final String host, final int port) { } 请求有关已连接客户端的信息和统计信息。 @Override public List getClientList() { return execute(RedisServerCommands::getClientList); }

将redis复制设置更改为新的主服务器。 @Override public void slaveOf(final String host, final int port) {}

将服务器更改为主服务器。 public void slaveOfNoOne() { }

返回特定于群集的操作接口 @Override public ClusterOperations opsForCluster() {

返回特定于地理空间的操作接口。

public GeoOperations opsForGeo() { }

返回绑定到给定键的特定于地理空间的操作接口。 public BoundGeoOperations boundGeoOps(K key) {}

返回对哈希值执行的操作。 public HashOperations opsForHash() { }

返回对list值执行的操作。 public ListOperations opsForList() {}

返回对绑定到给定键的列表值执行的操作。 public BoundListOperations boundListOps(K key) { }

返回对绑定到给定键的set值执行的操作。 public BoundSetOperations boundSetOps(K key) { }

返回对set值执行的操作。 public SetOperations opsForSet() { }

返回对zSet值执行的操作。

public BoundZSetOperations boundZSetOps(K key) {}

测试：

以下是方法的测试及结果：

redisTemplate.opsForValue();//操作字符串， redisTemplate.opsForHash();//操作hash
redisTemplate.opsForList();;//操作list redisTemplate.opsForSet();//操作set
redisTemplate.opsForZSet();//操作有序set

1.set void set(K key, V value);

```
使用：redisTemplate.opsForValue().set("name","tom");
结果：redisTemplate.opsForValue().get("name")  输出结果为tom
```

2. set void set(K key, V value, long timeout, TimeUnit unit);

```
使用：redisTemplate.opsForValue().set("name","tom",10, TimeUnit.SECONDS);
结果：redisTemplate.opsForValue().get("name")由于设置的是10秒失效，十秒之内查询有结果，十秒之后返回为null
```

3. set void set(K key, V value, long offset); 该方法是用 value 参数覆写(overwrite)给定 key 所储存的字符串值，从偏移量 offset 开始

使用：`template.opsForValue().set("key","helloworld");template.opsForValue().set("key","redis",6);System.out.println("**********"+template.opsForValue().get("key"));`结果：`**********helloredis`

4.setIfAbsent Boolean setIfAbsent(K key, V value);

```
使用：System.out.println(template.opsForValue().setIfAbsent("multi1","multi1"));//false  multi1之前已经存在

System.out.println(template.opsForValue().setIfAbsent("multi111","multi111"));//true  multi111之前不存在
 结果：false
```

```
  true
```

5.multiSet void multiSet(Map<? extends K, ? extends V> m); 为多个键分别设置它们的值

```
使用：Map<String,String> maps = new HashMap<String, String>();

                        maps.put("multi1","multi1");

                        maps.put("multi2","multi2");

                        maps.put("multi3","multi3");

                        template.opsForValue().multiSet(maps);

                        List<String> keys = new ArrayList<String>();

                        keys.add("multi1");

                        keys.add("multi2");

                        keys.add("multi3");

                        System.out.println(template.opsForValue().multiGet(keys));

                结果：[multi1, multi2, multi3]
```

6. multiSetIfAbsent Boolean multiSetIfAbsent(Map<? extends K, ? extends V> m);　为多个键分别设置它们的值，如果存在则返回false，不存在返回true

```
使用：Map<String,String> maps = new HashMap<String, String>();
                        maps.put("multi11","multi11");
                        maps.put("multi22","multi22");
                        maps.put("multi33","multi33");
                        Map<String,String> maps2 = new HashMap<String, String>();
                        maps2.put("multi1","multi1");
                        maps2.put("multi2","multi2");
                        maps2.put("multi3","multi3");
                        System.out.println(template.opsForValue().multiSetIfAbsent(maps));
                        System.out.println(template.opsForValue().multiSetIfAbsent(maps2));
                结果：true
                false
```

7.get V get(Object key);

```
使用：template.opsForValue().set("key","hello world");
                        System.out.println("***************"+template.opsForValue().get("key"));
                结果：***************hello world
```

8.getAndSet V getAndSet(K key, V value); 设置键的字符串值并返回其旧值

```
使用：template.opsForValue().set("getSetTest","test");

                        System.out.println(template.opsForValue().getAndSet("getSetTest","test2"));

                结果：test
```

9.multiGet List multiGet(Collection keys);为多个键分别取出它们的值

```
  使用：Map<String,String> maps = new HashMap<String, String>();
                        maps.put("multi1","multi1");
                        maps.put("multi2","multi2");
                        maps.put("multi3","multi3");
                        template.opsForValue().multiSet(maps);
                        List<String> keys = new ArrayList<String>();
                        keys.add("multi1");
                        keys.add("multi2");
                        keys.add("multi3");
                        System.out.println(template.opsForValue().multiGet(keys));
                结果：[multi1, multi2, multi3]
```

10.get String get(K key, long start, long end);截取key所对应的value字符串

```
使用：appendTest对应的value为Helloworld
System.out.println("*****"+template.opsForValue().get("appendTest",0,5));
 结果：*****Hellow
使用：System.out.println("*****"+template.opsForValue().get("appendTest",0,-1));
 结果：*****Helloworld
 使用：System.out.println("*****"+template.opsForValue().get("appendTest",-3,-1));
```

以字符串为中心的便捷类

由于Redis中存储的密钥和值非常常见`java.lang.String`，因此Redis模块分别为（及其实现）提供了两个扩展，`RedisConnection`并为密集的String操作提供了方便的一站式解决方案。除了绑定到键之外，模板和连接使用底层，这意味着存储的键和值是人类可读的（假设在Redis和您的代码中使用相同的编码）。以下列表显示了一个示例：`RedisTemplate``StringRedisConnection``DefaultStringRedisConnection``StringRedisTemplate``String``StringRedisSerializer`

```
<?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">

  <bean id="jedisConnectionFactory"
class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory" p:use-pool="true"/>

  <bean id="stringRedisTemplate" class="org.springframework.data.redis.core.StringRedisTemplate"
p:connection-factory-ref="jedisConnectionFactory"/>
  ...
</beans>
```

```
public class Example {

  @Autowired
  private StringRedisTemplate redisTemplate;

  public void addLink(String userId, URL url) {
    redisTemplate.opsForList().leftPush(userId, url.toExternalForm());
  }
}
```

由于与其他Spring模板， `RedisTemplate` 并 `StringRedisTemplate` 让你直接通过交谈Redis的 `RedisCallback` 界面。此功能为您提供完全控制，因为它直接与您对话 `RedisConnection` 。请注意，回调接收使用 `StringRedisConnection` a时的实例 `StringRedisTemplate` 。以下示例显示如何使用该 `RedisCallback` 接口：

```
public void useCallback() {

  redisTemplate.execute(new RedisCallback<Object>() {
    public Object doInRedis(RedisConnection connection) throws DataAccessException {
      Long size = connection.dbSize();
      // Can cast to StringRedisConnection if using a StringRedisTemplate
      ((StringRedisConnection)connection).set("key", "value");
    }
   });
}
```

串行器

从框架的角度来看，存储在Redis中的数据只是字节数。虽然Redis本身支持各种类型，但大多数情况下，这些类型指的是数据的存储方式而不是它所代表的方式。由用户决定信息是否被翻译成字符串或任何其他对象。

在Spring Data中，用户（自定义）类型和原始数据之间的转换（反之亦然）在 `org.springframework.data.redis.serializer` 包中处理Redis 。

该软件包包含两种类型的序列化程序，顾名思义，它们负责序列化过程：

- 基于的双向序列化器 `RedisSerializer` 。
- 使用 `RedisElementReader` 和的元素读者和作者 `RedisElementWriter` 。

这些变体之间的主要区别在于， `RedisSerializer` 主要是 `byte[]` 在读者和作者使用时序列化 `ByteBuffer` 。

可以使用多种实现（包括本文档中已经提到的两种实现）：

- `JdkSerializationRedisSerializer` ，默认用于 `RedisCache` 和 `RedisTemplate` 。
- 这个 `StringRedisSerializer` 。

但是，可以 `OxmSerializer` 通过Spring OXM支持用于对象/ XML映射， `Jackson2JsonRedisSerializer` 或者 `GenericJackson2JsonRedisSerializer` 用于以JSON格式存储数据。

请注意，存储格式不仅限于值。它可以用于键，值或哈希，没有任何限制。

**	**默认情况下，`RedisCache`并`RedisTemplate`配置为使用Java本机序列化。众所周知，Java本机序列化允许由利用易受攻击的库和类注入未经验证的字节码的有效负载引起的远程代码执行。在反序列化步骤中，操作输入可能导致应用程序中不需要的代码执行。因此，请勿在不受信任的环境中使用序列化。通常，我们强烈建议使用任何其他消息格式（例如JSON）。如果您担心Java序列化导致的安全漏洞，请考虑核心JVM级别的通用序列化过滤机制，最初为JDK 9开发但向后移植到JDK 8,7和6：过滤传入的序列化数据。JEP 290。OWASP：不受信任数据的反序列化。**

哈希映射

可以使用Redis中的各种数据结构存储数据。`Jackson2JsonRedisSerializer`可以转换JSON格式的对象。理想情况下，可以使用普通键将JSON存储为值。您可以使用Redis哈希实现更复杂的结构化对象映射。Spring Data Redis提供了各种将数据映射到哈希的策略（取决于用例）：

- 通过使用`HashOperations`和序列化器直接映射
- 使用Redis存储库
- 使用`HashMapper`和`HashOperations`

哈希映射器

哈希映射器是地图对象到a `Map<K, V>`和back的转换器。`HashMapper`适用于Redis Hashes。

有多种实现方式：

- `BeanUtilsHashMapper`使用Spring的BeanUtils。
- `ObjectHashMapper`使用对象哈希映射。
- `Jackson2HashMapper`使用FasterXML Jackson。

以下示例显示了实现哈希映射的一种方法：

```
public class Person {
  String firstname;
  String lastname;

  // …
}

public class HashMapping {

  @Autowired
  HashOperations<String, byte[], byte[]> hashOperations;

  HashMapper<Object, byte[], byte[]> mapper = new ObjectHashMapper();

  public void writeHash(String key, Person person) {

    Map<byte[], byte[]> mappedHash = mapper.toHash(person);
    hashOperations.putAll(key, mappedHash);
  }

  public Person loadHash(String key) {

    Map<byte[], byte[]> loadedHash = hashOperations.entries("key");
    return (Person) mapper.fromHash(loadedHash);

  }
```

```
}
```

Jackson2HashMapper

`Jackson2HashMapper`使用FasterXML Jackson为域对象提供Redis Hash映射。 `Jackson2HashMapper`可以将顶级属性映射为哈希字段名称，并可选择展平结构。简单类型映射到简单值。复杂类型（嵌套对象，集合，映射等）表示为嵌套JSON。

展平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。

考虑以下类及其包含的数据结构：

```
public class Person {
  String firstname;
  String lastname;
  Address address;
}

public class Address {
  String city;
  String country;
}
```

下表显示了前一类中的数据如何在法线贴图中显示：

哈希场	值
名字	`Jon`
姓	`Snow`
地址	`{ "city" : "Castle Black", "country" : "The North" }`

下表显示了前一类中的数据如何在平面映射中显示：

哈希场	值
名字	`Jon`
姓	`Snow`
address.city	`Castle Black`
address.country	`The North`

**	**展平需要所有属性名称不会干扰JSON路径。使用展平时，不支持在地图键中使用点或括号或作为属性名称。生成的哈希不能映射回Object。**

Redis消息（Pub / Sub）

Spring Data为Redis提供了专用的消息传递集成，功能类似，并命名为Spring Framework中的JMS集成。

Redis消息传递大致可分为两个功能区域：

- 发布或制作消息
- 订阅或消费消息

这是通常称为Publish / Subscribe（简称Pub / Sub）的模式示例。所述`RedisTemplate`类用于消息生成。对于类似于Java EE的消息驱动bean样式的异步接收，Spring Data提供了一个专用的消息监听器容器，用于创建消息驱动的POJO（MDP），以及用于同步接收的`RedisConnection`合同。

在`org.springframework.data.redis.connection`和`org.springframework.data.redis.listener`软件包提供了对Redis的消息的核心功能。

发布（发送消息）

要发布消息，您可以像使用其他操作一样使用低级别`RedisConnection`或高级别`RedisTemplate`。两个实体都提供该`publish`方法，该方法接受消息和目标通道作为参数。虽然`RedisConnection`需要原始数据（字节数组），但`RedisTemplate`允许任意对象作为消息传入，如以下示例所示：

```
// send message through connection RedisConnection con = ...
byte[] msg = ...
byte[] channel = ...
con.publish(msg, channel); // send message through RedisTemplate
RedisTemplate template = ...
template.convertAndSend("hello!", "world");
```

订阅（接收消息）

在接收方，可以通过直接命名或使用模式匹配来订阅一个或多个通道。后一种方法非常有用，因为它不仅可以使用一个命令创建多个订阅，还可以监听尚未在订阅时创建的通道（只要它们与模式匹配）。

在低级别，`RedisConnection`提供映射Redis命令的方法`subscribe`和`pSubscribe`方法，分别按通道或模式进行订阅。请注意，可以使用多个通道或模式作为参数。要更改连接的订阅或查询是否正在侦听，请`RedisConnection`提供`getSubscription`和`isSubscribed`方法。

**	**Spring Data Redis中的订阅命令是阻止的。也就是说，在连接上调用subscribe会导致当前线程在开始等待消息时阻塞。只有在取消订阅时才会释放该线程，这在另一个线程调用时`unsubscribe`或`pUnsubscribe`在同一连接上发生。有关此问题的解决方案，请参阅“消息侦听器容器”（本文档后面部分）。**

如前所述，一旦订阅，连接就开始等待消息。仅允许添加新订阅，修改现有订阅和取消现有订阅的命令。调用比其他任何东西`subscribe`，`pSubscribe`，`unsubscribe`，或`pUnsubscribe`抛出异常。

为了订阅消息，需要实现`MessageListener`回调。每次新消息到达时，都会调用回调并由该`onMessage`方法运行用户代码。该接口不仅可以访问实际消息，还可以访问通过它接收的通道以及订阅用于匹配通道的模式（如果有）。此信息使被叫方不仅可以通过内容区分各种消息，还可以检查其他详细信息。

消息侦听器容器

由于其阻塞性质，低级订阅不具吸引力，因为它需要每个单个侦听器的连接和线程管理。为了缓解这个问题，Spring Data提供了`RedisMessageListenerContainer`所有繁重的工作。如果您熟悉EJB和JMS，那么您应该找到熟悉的概念，因为它的设计尽可能接近Spring Framework及其消息驱动的POJO（MDP）中的支持。

`RedisMessageListenerContainer`充当消息监听器容器。它用于从Redis通道接收消息并驱动`MessageListener`注入其中的实例。侦听器容器负责消息接收的所有线程并将其分派到侦听器进行处理。消息监听器容器是MDP和消息传递提供者之间的中介，并负责注册以接收消息，资源获取和释放，异常转换等。这使您作为应用程序开发人员可以编写与接收消息（并对其做出反应）相关联的（可能是复杂的）业务逻辑，并将样板Redis基础结构关注委托给框架。

此外，为了最小化应用程序占用空间，`RedisMessageListenerContainer`即使多个侦听器不共享订阅，也允许多个侦听器共享一个连接和一个线程。因此，无论应用程序跟踪多少个侦听器或通道，运行时成本在其整个生命周期内保持不变。此外，容器允许更改运行时配置，以便您可以在应用程序运行时添加或删除侦听器，而无需重新启动。此外，容器使用延迟订阅方法，`RedisConnection`仅在需要时使用。如果所有侦听器都已取消订阅，则会自动执行清理，并释放该线程。

为了帮助消息的异步性，容器需要一个`java.util.concurrent.Executor`（或Spring `TaskExecutor`）来分派消息。根据负载，侦听器数量或运行时环境，您应该更改或调整执行程序以更好地满足您的需求。特别是，在托管环境（例如应用程序服务器）中，强烈建议选择适当的`TaskExecutor`方式来利用其运行时。

MessageListenerAdapter

本`MessageListenerAdapter`类是Spring的异步支持消息的最后一个组件。简而言之，它允许您将几乎**任何**类暴露为MDP（尽管存在一些约束）。

请考虑以下接口定义：

```
public interface MessageDelegate {
  void handleMessage(String message);
  void handleMessage(Map message); void handleMessage(byte[] message);
  void handleMessage(Serializable message);
  // pass the channel/pattern as well
  void handleMessage(Serializable message, String channel);
 }
```

请注意，虽然接口不扩展`MessageListener`接口，但仍可以通过使用`MessageListenerAdapter`该类将其用作MDP 。还要注意如何使用各种消息处理方法是根据强类型的**内容**不同的`Message`，他们可以接收和处理类型。此外，发送消息的通道或模式可以作为类型的第二个参数传递给方法`String`：

```
public class DefaultMessageDelegate implements MessageDelegate {
  // implementation elided for clarity...
}
```

注意上面的`MessageDelegate`接口实现（上面的`DefaultMessageDelegate`类）根本**没有** Redis依赖。它确实是我们在MDP中使用以下配置制作的POJO：

```
<?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:redis="http://www.springframework.org/schema/redis"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/redis http://www.springframework.org/schema/redis/spring-
redis.xsd">

<!-- the default ConnectionFactory -->
<redis:listener-container>
  <!-- the method attribute can be skipped as the default method name is "handleMessage" -->
  <redis:listener ref="listener" method="handleMessage" topic="chatroom" />
</redis:listener-container>

<bean id="listener" class="redisexample.DefaultMessageDelegate"/>
 ...
<beans>
```

****	**侦听器主题可以是通道（例如`topic="chatroom"`）或模式（例如`topic="*room"`）**

前面的示例使用Redis命名空间声明消息侦听器容器并自动将POJO注册为侦听器。完整的bean定义如下：

```
<bean id="messageListener" class="org.springframework.data.redis.listener.adapter.MessageListenerAdapter">
  <constructor-arg>
    <bean class="redisexample.DefaultMessageDelegate"/>
  </constructor-arg>
</bean>

<bean id="redisContainer" class="org.springframework.data.redis.listener.RedisMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory"/>
  <property name="messageListeners">
    <map>
      <entry key-ref="messageListener">
        <bean class="org.springframework.data.redis.listener.ChannelTopic">
          <constructor-arg value="chatroom">
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

每次收到消息时，适配器都会自动透明地执行`RedisSerializer`低级格式和所需对象类型之间的转换（使用已配置）。由方法调用引起的任何异常都由容器捕获和处理（默认情况下，会记录异常）。

Redis事务

Redis的提供支持的交易通过`multi`，`exec`和`discard`命令。这些操作可用`RedisTemplate`。但是，`RedisTemplate`不保证在具有相同连接的事务中执行所有操作。

Spring Data Redis提供了`SessionCallback`在需要执行多个操作时使用的接口`connection`，例如使用Redis事务时。以下示例使用以下`multi`方法：

```
//execute a transaction
List<Object> txResults = redisTemplate.execute(new SessionCallback<List<Object>>() {
  public List<Object> execute(RedisOperations operations) throws DataAccessException {
    operations.multi();
    operations.opsForSet().add("key", "value1");

    // This will contain the results of all operations in the transaction
    return operations.exec();
  }
});
System.out.println("Number of items added to set: " + txResults.get(0));
```

`RedisTemplate`使用其值，散列键和散列值序列化程序来反序列化`exec`返回之前的所有结果。还有一种`exec`方法可以让您为事务结果传递自定义序列化程序。

**	**从版本1.1开始，`exec`对`RedisConnection`和的方法进行了重大改变`RedisTemplate`。以前，这些方法直接从连接器返回事务的结果。这意味着数据类型通常不同于从方法返回的数据类型`RedisConnection`。例如，`zAdd`返回一个布尔值，指示元素是否已添加到有序集合中。大多数连接器将此值作为long返回，Spring Data Redis执行转换。另一个常见的区别是大多数连接器返回状态回复（通常是字符串`OK`），用于诸如的操作`set`。Spring Data Redis通常会丢弃这些回复。在1.1之前，这些转换没有在结果上执行`exec`。此外，结果未反序列化`RedisTemplate`，所以他们经常包括原始字节数组。如果这一变化打破了你的应用程序，设置`convertPipelineAndTxResults`到`false`您`RedisConnectionFactory`禁用此行为。**

@Transactional支持

默认情况下，事务支持已禁用，必须`RedisTemplate`通过设置为每个使用中显式启用`setEnableTransactionSupport(true)`。这样做会强制将电流绑定`RedisConnection`到`Thread`触发的电流`MULTI`。如果事务完成且没有错误，`EXEC`则调用。否则`DISCARD`被叫。进入后`MULTI`，`RedisConnection`队列写入操作。所有`readonly`操作，例如`KEYS`，都通过管道传输到新的（非线程绑定）`RedisConnection`。

以下示例显示了如何配置事务管理：

示例1.启用事务管理的配置

```
@Configuration
@EnableTransactionManagement
public class RedisTxContextConfiguration {

  @Bean
  public StringRedisTemplate redisTemplate() {
    StringRedisTemplate template = new StringRedisTemplate(redisConnectionFactory());
    // explicitly enable transaction support
    template.setEnableTransactionSupport(true);
    return template;
  }

  @Bean
  public RedisConnectionFactory redisConnectionFactory() {
    // jedis || Lettuce
  }

  @Bean
  public PlatformTransactionManager transactionManager() throws SQLException {
    return new DataSourceTransactionManager(dataSource());
  }

  @Bean
  public DataSource dataSource() throws SQLException {
    // ...
  }
}
```

**	**配置Spring Context以启用声明式事务管理。**
**	配置`RedisTemplate`通过绑定到当前线程的连接来参与事务。
**	交易管理需要一个`PlatformTransactionManager`。Spring Data Redis不附带`PlatformTransactionManager`实现。假设您的应用程序使用JDBC，Spring Data Redis可以使用现有的事务管理器参与事务。

以下示例均演示了使用限制：

示例2.使用限制

```
// must be performed on thread-bound connection
template.opsForValue().set("thing1", "thing2");

// read operation must be executed on a free (not transaction-aware) connection
template.keys("*");

// returns null as values set within a transaction are not visible
template.opsForValue().get("thing1");
```

流水线

Redis支持流水线操作，包括向服务器发送多个命令而无需等待回复，然后在一个步骤中读取回复。当您需要连续发送多个命令时，流水线操作可以提高性能，例如向同一个List添加许多元素。

Spring Data Redis提供了几种`RedisTemplate`在管道中执行命令的方法。如果你不关心流水线操作的结果，你可以使用标准`execute`方法，传递`true`的`pipeline`参数。所述`executePipelined`方法运行所提供的`RedisCallback`或`SessionCallback`在管道中，并返回结果，如显示在下面的例子：

```
//pop a specified number of items from a queue
List<Object> results = stringRedisTemplate.executePipelined(
  new RedisCallback<Object>() {
    public Object doInRedis(RedisConnection connection) throws DataAccessException {
      StringRedisConnection stringRedisConn = (StringRedisConnection)connection;
      for(int i=0; i< batchSize; i++) {
        stringRedisConn.rPop("myqueue");
      }
    return null;
  }
});
```

前面的示例从管道中的队列运行批量右侧弹出的项目。在`results` `List`包含了所有的弹出项目。`RedisTemplate`使用其值，散列键和散列值序列化程序在返回之前反序列化所有结果，因此前面示例中返回的项是字符串。还有其他`executePipelined`方法可以为流水线结果传递自定义序列化程序。

请注意，从中返回的值`RedisCallback`必须为null，因为此值将被丢弃，以支持返回流水线命令的结果。

**	**从版本1.1开始，`exec`对`RedisConnection`和的方法进行了重大改变`RedisTemplate`。以前，这些方法直接从连接器返回事务的结果。这意味着数据类型通常不同于从方法返回的数据类型`RedisConnection`。例如，`zAdd`返回一个布尔值，指示元素是否已添加到有序集合中。大多数连接器将此值作为long返回，Spring Data Redis执行转换。另一个常见的区别是大多数连接器返回状态回复（通常是字符串`OK`），用于诸如的操作`set`。Spring Data Redis通常会丢弃这些回复。在1.1之前，这些转换没有在结果上执行`exec`。此外，结果未反序列化`RedisTemplate`，所以他们经常包括原始字节数组。如果这一变化打破了你的应用程序，设置`convertPipelineAndTxResults`到`false`您`RedisConnectionFactory`禁用此行为。**

Redis脚本

Redis 2.6及更高版本通过eval和evalsha命令为执行Lua脚本提供支持。Spring Data Redis为脚本执行提供高级抽象，处理序列化并自动使用Redis脚本缓存。

可以通过调用and 的`execute`方法来运行脚本。两者都使用可配置（或）来运行提供的脚本。默认情况下，（或）负责序列化提供的键和参数以及反序列化脚本结果。这是通过模板的键和值序列化程序完成的。还有一个额外的重载，允许您为脚本参数和结果传递自定义序列化程序。
`RedisTemplate``ReactiveRedisTemplate``ScriptExecutor``ReactiveScriptExecutor``ScriptExecutor``ReactiveScriptExecutor`

默认情况下，`ScriptExecutor`通过检索脚本的SHA1并尝试首先运行来优化性能，如果脚本尚未存在于Redis脚本缓存中`evalsha`，`eval`则返回到该脚本。

以下示例使用Lua脚本运行常见的“检查和设置”方案。这是Redis脚本的理想用例，因为它要求以原子方式运行一组命令，并且一个命令的行为受另一个命令的结果的影响。

```
@Bean
public RedisScript<Boolean> script() {

  ScriptSource scriptSource = new ResourceScriptSource(new ClassPathResource("META-
INF/scripts/checkandset.lua");
  return RedisScript.of(scriptSource, Boolean.class);
}
```

```
public class Example {

  @Autowired
  RedisScript<Boolean> script;

  public boolean checkAndSet(String expectedValue, String newValue) {
    return redisTemplate.execute(script, singletonList("key"), asList(expectedValue, newValue));
  }
}
```

```
-- checkandset.lua
local current = redis.call('GET', KEYS[1])
if current == ARGV[1]
  then redis.call('SET', KEYS[1], ARGV[2])
  return true
end
return false
```

上面的代码配置`RedisScript`指向一个名为的文件`checkandset.lua`，该文件应该返回一个布尔值。该脚本`resultType`应该是一个`Long`，`Boolean`，`List`或反序列化的值类型。它也可以`null`是脚本返回丢弃状态（具体而言`OK`）。

**	**`DefaultRedisScript`在应用程序上下文中 配置单个实例是理想的，以避免在每次脚本执行时重新计算脚本的SHA1。**

然后`checkAndSet`上面的方法运行脚本。脚本可以`SessionCallback`作为事务或管道的一部分在一个内部运行。有关详细信息，请参阅“ Redis Transactions ”和“ Pipelining ”。

Spring Data Redis提供的脚本支持还允许您使用Spring Task和Scheduler抽象来安排Redis脚本以定期执行。有关更多详细信息，请参阅Spring Framework文档。

支持类

Package `org.springframework.data.redis.support`提供了各种可重用的组件，这些组件依赖Redis作为后备存储。目前，该软件包在Redis之上包含各种基于JDK的接口实现，例如原子计数器和JDK 集合。

原子计数器可以轻松地包装Redis密钥增量，而集合允许以最小的存储空间或API泄漏轻松管理Redis密钥。特别是，`RedisSet`和`RedisZSet`接口提供了对Redis支持的集合操作的轻松访问，例如`intersection`和`union`。`RedisList`实现`List`，`Queue`以及`Deque`上Redis的顶部合同（和它们的等效阻断兄弟姐妹），露出存储作为FIFO（先入先出），LIFO（后进先出），或者用最少的配置封端的集。以下示例显示了使用以下bean的bean的配置`RedisList`：

```
<?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">

  <bean id="queue" class="org.springframework.data.redis.support.collections.DefaultRedisList">
    <constructor-arg ref="redisTemplate"/>
    <constructor-arg value="queue-key"/>
  </bean>

</beans>
```

以下示例显示了以下的Java配置示例`Deque`：

```
public class AnotherExample {

  // injected
  private Deque<String> queue;

  public void addTag(String tag) {
    queue.push(tag);
  }
}
```

如前面的示例所示，使用代码与实际存储实现分离。实际上，没有迹象表明Redis被用在了下面。这使得从开发环境转移到生产环境变得透明并大大提高了可测试性（Redis实现可以替换为内存实现）。

支持Spring Cache Abstraction

**	改变了2.0

Spring Redis 通过包提供Spring 缓存抽象的`org.springframework.data.redis.cache`实现。要将Redis用作后备实现，请添加`RedisCacheManager`到您的配置中，如下所示：

```
@Bean
public RedisCacheManager cacheManager(RedisConnectionFactory connectionFactory) {
    return RedisCacheManager.create(connectionFactory);
}
```

`RedisCacheManager`可以配置行为`RedisCacheManagerBuilder`，允许您设置默认`RedisCacheConfiguration`，事务行为和预定义的缓存。

```
RedisCacheManager cm = RedisCacheManager.builder(connectionFactory)
    .cacheDefaults(defaultCacheConfig())
    .withInitialCacheConfigurations(singletonMap("predefined",
defaultCacheConfig().disableCachingNullValues()))
    .transactionAware()
    .build();
```

如前面的示例所示，`RedisCacheManager`允许基于每个缓存定义配置。

`RedisCache`创建的行为`RedisCacheManager`定义为`RedisCacheConfiguration`。通过该配置，您可以设置密钥到期时间，前缀和`RedisSerializer`实现，以便与二进制存储格式进行转换，如以下示例所示：

```
RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig()
    .entryTtl(Duration.ofSeconds(1))
    .disableCachingNullValues();
```

`RedisCacheManager`默认为无锁，`RedisCacheWriter`用于读取和写入二进制值。无锁缓存可提高吞吐量。缺少条目锁定可能导致`putIfAbsent`和`clean`方法的重叠非原子命令，因为那些需要将多个命令发送到Redis。锁定对应物通过设置显式锁定键并检查是否存在此键来防止命令重叠，这会导致其他请求和潜在的命令等待时间。

可以按如下方式选择锁定行为：

```
RedisCacheManager cm = RedisCacheManager.build(RedisCacheWriter.lockingRedisCacheWriter())
    .cacheDefaults(defaultCacheConfig())
    ...
```

默认情况下，任何`key`缓存条目都以实际缓存名称为后缀，后跟两个冒号。此行为可以更改为静态和计算前缀。

以下示例显示如何设置静态前缀：

```
// static key prefix
RedisCacheConfiguration.defaultCacheConfig().prefixKeysWith("( ͡° ᴥ ͡°)");

The following example shows how to set a computed prefix:

// computed key prefix
RedisCacheConfiguration.defaultCacheConfig().computePrefixWith(cacheName -> "¯\_(ツ)_/¯" + cacheName);
```

下表列出了以下内容的默认设置`RedisCacheManager`：

设置	值
缓存编写器	非锁定
缓存配置	`RedisCacheConfiguration#defaultConfiguration`
初始缓存	没有
Trasaction Aware	没有

下表列出了以下内容的默认设置`RedisCacheConfiguration`：

密钥到期	没有
高速缓存`null`	是
前缀键	是
默认前缀	实际的缓存名称
密钥序列化器	`StringRedisSerializer`
价值序列化器	`JdkSerializationRedisSerializer`
转换服务	`DefaultFormattingConversionService`使用默认缓存密钥转换器

Reactive Redis支持

本节介绍了Redis的反应支持以及如何入门。Reactive Redis支持自然与Redis支持的必要重叠。

Redis要求

Spring Data Redis目前与Lettuce集成，是唯一的反应式Java连接器。Project Reactor用作反应性组合库。

使用反应驱动程序连接到Redis

使用Redis和Spring时的首要任务之一是通过IoC容器连接到商店。为此，需要Java连接器（或绑定）。无论您选择哪个库，都必须使用`org.springframework.data.redis.connection`它`ReactiveRedisConnection`及其`ReactiveRedisConnectionFactory`接口和接口来处理和检索`connections`Redis的活动。

Redis操作模式

Redis可以作为独立服务器运行，使用Redis Sentinel运行，也可以在Redis群集模式下运行。 Lettuce支持所有前面提到的连接类型。

`ReactiveRedisConnection`和`ReactiveRedisConnectionFactory`

`ReactiveRedisConnection`是Redis通信的核心，因为它处理与Redis后端的通信。它还会自动将底层驱动程序异常转换为Spring的一致DAO异常层次结构，因此您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。

`ReactiveRedisConnectionFactory`创建活动`ReactiveRedisConnection`实例。此外，工厂充当`PersistenceExceptionTranslator`实例，这意味着，一旦声明，它们就会让您进行透明的异常转换 - 例如，通过使用`@Repository`注释和AOP进行异常转换。有关更多信息，请参阅Spring Framework文档中的专用部分。

**	**根据基础配置，工厂可以返回新连接或现有连接（如果使用池或共享本机连接）。**

**	**使用a的最简单方法`ReactiveRedisConnectionFactory`是通过IoC容器配置适当的连接器并将其注入using类。**

配置lettuce连接器

Spring Data Redis通过`org.springframework.data.redis.connection.lettuce`软件包支持生菜。

您可以`ReactiveRedisConnectionFactory`按如下方式设置生菜：

```
@Bean
public ReactiveRedisConnectionFactory connectionFactory() {
  return new LettuceConnectionFactory("localhost", 6379);
}
```

以下示例显示了一个更复杂的配置，包括SSL和超时，它使用`LettuceClientConfigurationBuilder`：

```
@Bean
public ReactiveRedisConnectionFactory lettuceConnectionFactory() {

  LettuceClientConfiguration clientConfig = LettuceClientConfiguration.builder()
    .useSsl().and()
    .commandTimeout(Duration.ofSeconds(2))
    .shutdownTimeout(Duration.ZERO)
    .build();

  return new LettuceConnectionFactory(new RedisStandaloneConfiguration("localhost", 6379), clientConfig);
}
```

有关更详细的客户端配置调整，请参阅`LettuceClientConfiguration`。

通过ReactiveRedisTemplate处理对象

大多数用户可能会使用`ReactiveRedisTemplate`它及其相应的包`org.springframework.data.redis.core`。由于其丰富的功能集，模板实际上是Redis模块的中心类。该模板为Redis交互提供了高级抽象。虽然`ReactiveRedisConnection`提供了接受和返回二进制值（`ByteBuffer`）的低级方法，但模板负责序列化和连接管理，使您无需处理这些细节。

此外，模板提供操作视图（在Redis命令参考的分组之后），提供丰富的，通用的接口，用于处理特定类型，如下表所述：

接口	**描述**
密钥类型操作	
ReactiveGeoOperations	如Redis的地理空间操作的`GEOADD`，`GEORADIUS`和其他人）
ReactiveHashOperations	Redis哈希操作
ReactiveHyperLogLogOperations	Redis的HyperLogLog操作，如（`PFADD`，`PFCOUNT`，等）
ReactiveListOperations	Redis列表操作
ReactiveSetOperations	Redis设置了操作
ReactiveValueOperations	Redis字符串（或值）操作
ReactiveZSetOperations	Redis zset（或排序集）操作

配置完成后，模板是线程安全的，可以跨多个实例重用。

`ReactiveRedisTemplate`在大多数操作中使用基于Java的序列化程序。这意味着模板编写或读取的任何对象都通过`RedisElementWriter`或序列化或反序列化`RedisElementReader`。序列化上下文在构造时传递给模板，Redis模块提供了`org.springframework.data.redis.serializer`包中可用的多个实现。有关更多信息，请参阅序列化器

以下示例显示了`ReactiveRedisTemplate`用于返回的一个`Mono`：

```
@Configuration
class RedisConfiguration {

  @Bean
  ReactiveRedisTemplate<String, String> reactiveRedisTemplate(ReactiveRedisConnectionFactory factory) {
    return new ReactiveRedisTemplate<>(factory, RedisSerializationContext.string());
  }
}
```

```
public class Example {

  @Autowired
  private ReactiveRedisTemplate<String, String> template;

  public Mono<Long> addLink(String userId, URL url) {
    return template.opsForList().leftPush(userId, url.toExternalForm());
  }
}
```

以字符串为中心的便捷类

因为它是存储在Redis的是一个键和值相当普遍`java.lang.String`，Redis的模块提供了一个基于字符串的扩展`ReactiveRedisTemplate`：`ReactiveStringRedisTemplate`。对于密集型`String`操作，它是一种方便的一站式解决方案。除了绑定到`String`键之外，模板还使用基于字符串的方式`RedisSerializationContext`，这意味着存储的键和值是人类可读的（假设在Redis和代码中使用相同的编码）。以下示例显示`ReactiveStringRedisTemplate`正在使用中：

```
@Configuration
class RedisConfiguration {

  @Bean
  ReactiveStringRedisTemplate reactiveRedisTemplate(ReactiveRedisConnectionFactory factory) {
    return new ReactiveStringRedisTemplate<>(factory);
  }
}
```

```
public class Example {

  @Autowired
  private ReactiveStringRedisTemplate redisTemplate;

  public Mono<Long> addLink(String userId, URL url) {
    return redisTemplate.opsForList().leftPush(userId, url.toExternalForm());
  }
}
```

Redis Messaging / PubSub

Spring Data为Redis提供了专用的消息传递集成，在功能和命名方面与Spring Framework中的JMS集成非常相似; 事实上，熟悉Spring中JMS支持的用户应该感到宾至如归。

Redis消息传递大致可以分为两个功能区域，即消息的生成或发布以及消费或订阅，因此快捷方式pubsub（发布/订阅）。所述`ReactiveRedisTemplate`类用于消息生成。对于异步接收，Spring Data提供了一个专用的消息监听器容器，用于使用消息流。仅仅`ReactiveRedisTemplate`为了使用监听器容器订阅提供的替代方案。

该软件包`org.springframework.data.redis.connection`并`org.springframework.data.redis.listener`提供使用Redis消息传递的核心功能。

发送/发布消息

要发布消息，可以像其他操作一样使用低级别`ReactiveRedisConnection`或高级别`ReactiveRedisTemplate`。两个实体都提供一种发布方法，该方法接受需要发送的消息以及目标通道作为参数。虽然`ReactiveRedisConnection`需要原始数据，但`ReactiveRedisTemplate`允许任意对象作为消息传递：

```
// send message through ReactiveRedisConnection
ByteBuffer msg = …
ByteBuffer channel = …
Mono<Long> publish = con.publish(msg, channel);

// send message through ReactiveRedisTemplate
ReactiveRedisTemplate template = …
Mono<Long> publish = template.convertAndSend("channel", "message");
```

接收/订阅消息

在接收方，可以通过直接命名或使用模式匹配来订阅一个或多个通道。后一种方法非常有用，因为它不仅允许使用一个命令创建多个订阅，而且还可以监听尚未在订阅时创建的通道（只要它们与模式匹配）。

在低级别，`ReactiveRedisConnection`提供`subscribe`和`pSubscribe`方法，映射Redis命令分别按模式按通道订阅。请注意，可以使用多个通道或模式作为参数。要更改订阅，只需查询其中的渠道和模式即可`ReactiveSubscription`。

**	**Spring Data Redis中的反向订阅命令是非阻塞的，可以在不发出元素的情况下终止。**

如上所述，一旦订阅连接就开始等待消息。除了添加新订阅或修改/取消现有订阅外，不能在其上调用其他命令。比其他命令`subscribe`，`pSubscribe`，`unsubscribe`，或者`pUnsubscribe`是非法的，将导致异常。

为了接收消息，需要获取消息流。请注意，订阅仅发布针对该特定订阅注册的频道和模式的消息。消息流本身是一个热门序列，可以在不考虑需求的情况下生成元素。确保注册足够的需求以避免耗尽消息缓冲区。

消息侦听器容器

Spring Data提供`ReactiveRedisMessageListenerContainer`代表用户完成所有繁重的转换和订阅状态管理。

`ReactiveRedisMessageListenerContainer`充当消息监听器容器。它用于从Redis通道接收消息，并公开应用反序列化发出通道消息的消息流。它负责注册接收消息，资源获取和释放，异常转换等。这允许您作为应用程序开发人员编写与接收消息（并对其做出反应）相关联的（可能是复杂的）业务逻辑，并将样板Redis基础结构关注委托给框架。消息流在发布者订阅时在Redis中注册订阅，如果订阅被取消则注销。

此外，为了最小化应用程序占用空间，`ReactiveRedisMessageListenerContainer`允许多个侦听器共享一个连接和一个线程，即使它们不共享订阅。因此，无论应用程序跟踪多少个侦听器或通道，运行时成本在其生命周期内将保持不变。此外，容器允许运行时配置更改，因此可以在应用程序运行时添加或删除侦听器，而无需重新启动。此外，容器使用延迟订阅方法，`ReactiveRedisConnection`仅在需要时使用 - 如果所有侦听器都已取消订阅，则会自动执行清理。

消息侦听器容器本身不需要外部线程资源。它使用驱动程序线程来发布消息。

```
ReactiveRedisConnectionFactory factory = …
ReactiveRedisMessageListenerContainer container = new ReactiveRedisMessageListenerContainer(factory);

Flux<ChannelMessage<String, String>> stream = container.receive(ChannelTopic.of("my-chanel"));
```

通过模板API订阅

如上所述，您可以直接使用`ReactiveRedisTemplate`订阅频道/模式。这种方法提供了一种直接的，虽然有限的解决方案，因为您放弃了在初始订阅后添加订阅的选项。尽管如此，您仍然可以通过返回`Flux`使用例如控制消息流。`take(Duration)`。完成读取后，在出错或取消时，将再次释放所有绑定资源。

```
redisTemplate.listenToChannel("channel1", "channel2").doOnNext(msg -> {
    // message processing ...
}).subscribe();
```

反应性脚本

通过响应式基础架构执行Redis脚本可以使用`ReactiveScriptExecutor`访问的最佳通道来完成`ReactiveRedisTemplate`。

```
public class Example {

  @Autowired
  private ReactiveRedisTemplate<String, String> template;

  public Flux<Long> theAnswerToLife() {
```

```
    DefaultRedisScript<Long> script = new DefaultRedisScript<>();
    script.setLocation(new ClassPathResource("META-INF/scripts/42.lua"));
    script.setResultType(Long.class);

    return reactiveTemplate.execute(script);
  }
}
```

有关脚本命令的更多详细信息，请参阅脚本部分。

Redis集群

使用Redis群集需要Redis Server 3.0+版。有关详细信息，请参阅群集教程。

启用Redis群集

群集支持基于与非群集通信相同的构建块。`RedisClusterConnection`，扩展`RedisConnection`，处理与Redis群集的通信，并将错误转换为Spring DAO异常层次结构。`RedisClusterConnection`使用`RedisConnectionFactory`必须使用关联设置的实例创建实例`RedisClusterConfiguration`，如以下示例所示：

示例3. Redis群集的RedisConnectionFactory配置示例

```
@Component
@ConfigurationProperties(prefix = "spring.redis.cluster")
public class ClusterConfigurationProperties {

    /*
     * spring.redis.cluster.nodes[0] = 127.0.0.1:7379
     * spring.redis.cluster.nodes[1] = 127.0.0.1:7380
     * ...
     */
    List<String> nodes;

    /**
     * Get initial collection of known cluster nodes in format {@code host:port}.
     *
     * @return
     */
    public List<String> getNodes() {
        return nodes;
    }

    public void setNodes(List<String> nodes) {
        this.nodes = nodes;
    }
}

@Configuration
public class AppConfig {

    /**
     * Type safe representation of application.properties
     */
    @Autowired ClusterConfigurationProperties clusterProperties;

    public @Bean RedisConnectionFactory connectionFactory() {
```

```
        return new JedisConnectionFactory(
            new RedisClusterConfiguration(clusterProperties.getNodes()));
    }
}
```

**	**`RedisClusterConfiguration`也可以通过定义`PropertySource`并具有以下属性：配置属性`spring.redis.cluster.nodes`：逗号分隔的主机：端口对列表。`spring.redis.cluster.max-redirects`：允许的群集重定向数。**

**	**初始配置将驱动程序库指向一组初始集群节点。实时群集重新配置导致的更改仅保留在本机驱动程序中，不会写回配置。**

使用Redis群集连接

如前所述，Redis群集的行为与单节点Redis或甚至Sentinel监控的主副本环境不同。这是因为自动分片将密钥映射到16384个插槽之一，这些插槽分布在节点上。因此，涉及多个密钥的命令必须断言所有密钥映射到完全相同的插槽，以避免跨时隙执行错误。单个群集节点仅提供一组专用密钥。针对一个特定服务器发出的命令仅返回该服务器所服务的密钥的结果。举个简单的例子，考虑一下`KEYS`命令。当发布到群集环境中的服务器时，它仅返回请求发送到的节点所服务的密钥，而不一定返回群集中的所有密钥。因此，要获取群集环境中的所有密钥，必须从所有已知主节点读取密钥。

虽然驱动程序库处理特定键到相应的插槽服务节点的重定向，但是更高级别的功能（例如跨节点收集信息或向集群中的所有节点发送命令）都包括在内`RedisClusterConnection`。从前面提取密钥示例，这意味着该`keys(pattern)`方法获取集群中的每个主节点，同时`KEYS`在每个主节点上执行该命令，同时获取结果并返回累积的密钥集。仅请求单个节点的密钥`RedisClusterConnection`为这些方法提供重载（例如，`keys(node, pattern)`）。

`RedisClusterNode`可以`RedisClusterConnection.clusterGetNodes`通过使用主机和端口或节点Id 来获得或者可以构造A.

以下示例显示了在集群中运行的一组命令：

示例4.跨群集运行命令的示例

```
redis-cli@127.0.0.1:7379 > cluster nodes

6b38bb... 127.0.0.1:7379 master - 0 0 25 connected 0-5460
7bb78c... 127.0.0.1:7380 master - 0 1449730618304 2 connected 5461-10922
164888... 127.0.0.1:7381 master - 0 1449730618304 3 connected 10923-16383
b8b5ee... 127.0.0.1:7382 slave 6b38bb... 0 1449730618304 25 connected
```

```
RedisClusterConnection connection = connectionFactory.getClusterConnnection();

connection.set("thing1", value);
connection.set("thing2", value);

connection.keys("*");

connection.keys(NODE_7379, "*");
connection.keys(NODE_7380, "*");
connection.keys(NODE_7381, "*");
connection.keys(NODE_7382, "*");
```

**	**主节点服务时隙0到5460在7382复制到副本**
**	主节点服务时隙5461到10922
**	主节点服务时隙10923到16383
**	副本节点在7379处持有主节点的副本
**	请求路由到服务时隙12182的7381处的节点
**	请求路由到7379服务时隙5061的节点
**	请求路由到节点7379,7380,7381→[thing1，thing2]
**	请求路由到节点7379→[thing2]
**	请求路由到7380→[]节点
**	请求路由到节点7381→[thing1]
**	请求路由到7382→[thing2]节点

当所有键映射到同一插槽时，本机驱动程序库会自动为跨插槽请求提供服务，例如`MGET`。但是，如果不是这种情况，则对插槽服务节点`RedisClusterConnection`执行多个并行`GET`命令，并再次返回累积结果。这比单槽执行效率低，因此应谨慎使用。如果有疑问，请考虑通过在大括号中提供前缀（例如`{my-prefix}.thing1`和）将密钥固定到同一个插槽`{my-prefix}.thing2`，这两个前缀都将映射到相同的插槽号。以下示例显示了跨槽请求处理：

示例5.交叉槽请求处理的示例

```
redis-cli@127.0.0.1:7379 > cluster nodes

6b38bb... 127.0.0.1:7379 master - 0 0 25 connected 0-5460
7bb...
```

```
RedisClusterConnection connection = connectionFactory.getClusterConnnection();

connection.set("thing1", value);           // slot: 12182
connection.set("{thing1}.thing2", value);  // slot: 12182
connection.set("thing2", value);           // slot:  5461

connection.mGet("thing1", "{thing1}.thing2");

connection.mGet("thing1", "thing2");
```

**	**与之前的示例相同的配置。**
**	键映射到相同的插槽→127.0.0.1：7381 MGET thing1 {thing1} .thing2
**	键映射到不同的插槽并分成单个插槽，路由到相应的节点→127.0.0.1:7379 GET thing2 →127.0.0.1：7381 GET thing1

**	**前面的示例演示了Spring Data Redis遵循的一般策略。请注意，某些操作可能需要将大量数据加载到内存中以计算所需的命令。此外，并非所有跨时隙请求都可以安全地移植到多个单插槽请求，如果误用，则会出错（例如`PFCOUNT`）。**

`RedisTemplate`和`ClusterOperations`

有关通用，配置和用法的信息，请参阅“ 使用RedisTemplate处理对象”部分`RedisTemplate`。

**	**`RedisTemplate#keySerializer`使用任何JSON进行 设置时要小心`RedisSerializers`，因为更改JSON结构会立即影响哈希槽计算。**

`RedisTemplate`通过`ClusterOperations`接口提供对特定于集群的操作的访问，该接口可以从中获取`RedisTemplate.opsForCluster()`。这使您可以在集群中的单个节点上显式运行命令，同时保留为模板配置的序列化和反序列化功能。它还提供管理命令（例如`CLUSTER MEET`）或更多高级操作（例如，重新分片）。

下面的示例演示了如何访问`RedisClusterConnection`有`RedisTemplate`：

示例6. `RedisClusterConnection`使用。访问`RedisTemplate`

```
ClusterOperations clusterOps = redisTemplate.opsForCluster();
clusterOps.shutdown(NODE_7379);
```

**	**在7379关闭节点并交叉手指有一个可以接管的复制品。**

Jedis vs Lettuce

参考https://juejin.im/post/5ba0a098f265da0adb30c684

下面以Jedis客户端为例，再来总结下 客户端直连方式和连接池方式的对比

	优点	缺点
直连	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连接开销 2. 资源无法控制，极端情况下出现连接泄漏 3. Jedis对象线程不安全(Lettuce对象是线程安全的)
连接池	1. 无需每次连接生成Jedis对象，降低开销 2. 使用连接池的形式保护和控制资源的使用	相对于直连，使用更加麻烦，尤其在资源的管理上需要很多参数来保证，一旦规划不合理也会出现问题

Lettuce是一个可扩展的线程安全的Redis客户端，用于同步，异步和反应式使用。如果多个线程避免阻塞和事务操作（例如`BLPOP`和`MULTI`/），则它们可以共享一个连接`EXEC`。生菜是用netty建造的。支持先进的Redis功能，如Sentinel，Cluster，Pipelining，Auto-Reconnect和Redis数据模型。

基本用法：

示例1.基本用法

```
RedisClient客户端=  RedisClient 。create (“ redis：// localhost ”) ;  (1)

StatefulRedisConnection < String, String > connection = client 。连接 () ;  (2)

RedisCommands < String, String > commands = connection 。同步 () ;  (3)

String value = commands 。得到 (“ foo ”) ;  (4)

...

连接。关 () ;  (5)

客户。关掉 () ;  (6)
```

1. 创建`RedisClient`实例并提供指向localhost的Redis URI，端口6379（默认端口）。
2. 打开Redis独立连接。端点用于初始化`RedisClient`
3. 获取用于同步执行的命令API。Lettuce也支持异步和反应执行模型。
4. 发出`GET`命令以获取密钥`foo`。
5. 完成后关闭连接。这通常发生在您的应用程序的最后。连接设计为长寿命。
6. 关闭客户端实例以释放线程和资源。这通常发生在您的应用程序的最后。

每个Redis命令都由一个或多个方法实现，其名称与小写Redis命令名称相同。具有更改结果类型的多个修饰符的复杂命令包括CamelCased修饰符作为命令名称的一部分，例如`zrangebyscore`和`zrangebyscoreWithScores`。

Redis连接设计为长寿命和线程安全的，如果连接丢失，则重新连接直到`close()`被调用。在成功重新连接后将（重新）发送尚未超时的待命命令。所有连接都从其RedisClient继承默认超时，并且`RedisException`在超时到期之前，非阻塞命令无法返回结果时抛出。超时默认为60秒，可以在RedisClient中或每个连接中更改。`RedisCommandExecutionException`如果Redis响应错误，同步方法将抛出。当Redis响应错误时，异步连接不会抛出异常。

RedisURI

RedisURI包含主机/端口，可以携带身份验证/数据库详细信息。在成功连接上，您将获得身份验证，然后选择数据库。这也适用于在连接丢失后重新建立连接之后。还可以从URI字符串创建Redis URI。支持的格式是：

- `redis://[password@]host[:port][/databaseNumber]` Plaintext Redis连接
- `rediss://[password@]host[:port][/databaseNumber]` SSL Redis连接
- `redis-sentinel://[password@]host[:port][,host2[:port2]][/databaseNumber]#sentinelMasterId`使用Redis Sentinel
- `redis-socket:///path/to/socket`与Redis的Unix域套接字连接

例外

如果Redis发生异常/错误响应，您将收到`RedisException`包含错误消息的消息。`RedisException`是一个`RuntimeException`。

例子

示例2.使用主机和端口并将默认超时设置为20秒

```
RedisClient客户端=  RedisClient 。创建 (RedisURI 。创建 (“本地主机”, 6379) ) ;
客户。setDefaultTimeout (20, TIMEUNIT 。  SECONDS) ;

// ...

客户。关掉 () ;
```

示例3.使用RedisURI

```
RedisURI redisUri =  RedisURI 。生成器。redis (“ localhost ”)
                                .withPassword (“身份验证”)
                                .withDatabase (2)
                                。建立 () ;
RedisClient客户端=  RedisClient 。创建 (redisUri) ;

// ...

客户。关掉 () ;
```

示例4. SSL RedisURI

```
RedisURI redisUri =  RedisURI 。生成器。redis (“ localhost ”)
                                .withSsl (true)
                                .withPassword (“身份验证”)
                                .withDatabase (2)
                                。建立 () ;
RedisClient客户端=  RedisClient 。创建 (redisUri) ;

// ...

客户。关掉 () ;
```

示例5.字符串RedisURI

```
RedisURI redisUri =  RedisURI 。create (“ redis：// authentication @ localhost / 2 ”) ;
RedisClient客户端=  RedisClient 。创建 (redisUri) ;

// ...

客户。关掉 () ;
```

Lettuce 是 一种可伸缩，线程安全，完全非阻塞的Redis客户端，多个线程可以共享一个RedisConnection,它利用Netty NIO框架来高效地管理多个连接，从而提供了异步和同步数据访问方式，用于构建非阻塞的反应性应用程序。

在 springboot 1.5.x版本的默认的Redis客户端是 Jedis实现的，springboot 2.x版本中默认客户端是用 lettuce实现的。

下面介绍 `springboot 2.0` 分别使用 `jedis` 和 `lettuce` 集成 redis服务

官方推荐使用：

```
/**
 * @author Mark Paluch
 */
public class ConnectToRedis {
```

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

        // Syntax: redis://[password@]host[:port][/databaseNumber]
        RedisClient redisClient = RedisClient.create("redis://password@localhost:6379/0");
        StatefulRedisConnection<String, String> connection = redisClient.connect();

        System.out.println("Connected to Redis");

        connection.close();
        redisClient.shutdown();
    }
}
```

```
/**
 * @author Mark Paluch
 */
public class ConnectToRedisCluster {

    public static void main(String[] args) {

        // Syntax: redis://[password@]host[:port]
        RedisClusterClient redisClient = RedisClusterClient.create("redis://password@localhost:7379");

        StatefulRedisClusterConnection<String, String> connection = redisClient.connect();

        System.out.println("Connected to Redis");

        connection.close();
        redisClient.shutdown();
    }
}
```

springboot 2.0 通过 lettuce集成Redis服务

导入依赖

```
<dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>
    </dependencies>
```

application.properties配置文件

```
#host服务器
spring.redis.host=localhost
#端口
spring.redis.port=6379
#连接名
spring.redis.password=root
# 连接池最大连接数(使用负值表示没有限制) 默认为8

spring.redis.lettuce.pool.max-active=8
```

```
# 连接池最大阻塞等待时间(使用负值表示没有限制) 默认为-1
spring.redis.lettuce.pool.max-wait=-1ms
# 连接池中的最大空闲连接 默认为8
spring.redis.lettuce.pool.max-idle=8
# 连接池中的最小空闲连接 默认为 0
spring.redis.lettuce.pool.min-idle=0
```

application.yml文件:

```
spring:
  redis:
    lettuce:
      pool:
        max-active: 8
        max-idle: 8
        max-wait: -1ms
        min-idle: 0
    sentinel:
      master: mymaster
      nodes: 192.168.147.132:6379
    #########sentinel集群####
  redis:
    lettuce:
      pool:
        max-active: 8
        max-idle: 8
        max-wait: -1ms
        min-idle: 0
    sentinel:
      master: mymaster
      nodes: 192.168.147.132:26379, 192.168.147.132:26380, 192.168.147.132:26381
```

自定义 RedisTemplate方式1

默认情况下的模板只能支持 `RedisTemplate<String,String>`，只能存入字符串，很多时候，我们需要自定义 RedisTemplate，设置序列化器，这样我们可以很方便的操作实例对象。如下所示：

```
@Configuration
public class RedisConfig {
    @Bean
    public RedisTemplate<String, Serializable> redisTemplate(LettuceConnectionFactory connectionFactory) {
        RedisTemplate<String, Serializable> redisTemplate = new RedisTemplate<>();
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        redisTemplate.setValueSerializer(new GenericJackson2JsonRedisSerializer());
        redisTemplate.setConnectionFactory(connectionFactory);
        return redisTemplate;
    }
}
复制代码
```

自定义 RedisTemplate方式2:

```
/**
```

```
 * @Author kay三石
 * @date:2019/6/30
 * 定义序列化的操作，所有的类和对象
 */
public class RedisObjectSerializer implements RedisSerializer<Object> {
    //为了方便对象与数组的转换使用这两个对象转换器
    private Converter<Object ,byte[]> serializingConverter=new SerializingConverter();
    private Converter<byte[],Object> deserializingConverter=new DeserializingConverter();

    @Override
    public byte[] serialize(Object object) throws SerializationException {
        if (object==null){
            return new byte[0];
        }
        //序列化对象
        return this.serializingConverter.convert(object);
    }

    @Override
    public Object deserialize(byte[] bytes) throws SerializationException {
        if (bytes==null || bytes.length == 0){
            return null;
        }
        return this.deserializingConverter.convert(bytes);
    }
}
/**
 * @Author kay三石
 * @date:2019/6/30
 * 为了让RedisTemplate操作模板知道有这样的一个序列程序类存在，这里定义配置
 */
@Configuration
public class RedisConfig {

    @Bean
    public RedisTemplate<String,Object> getRedisTemplate(RedisConnectionFactory factory){
        RedisTemplate<String,Object> redisTemplate=new RedisTemplate <>();
        redisTemplate.setConnectionFactory(factory);
        redisTemplate.setKeySerializer(new StringRedisSerializer()); //key的序列化类型
        redisTemplate.setValueSerializer(new RedisObjectSerializer());//value的序列化类型
        return  redisTemplate;
    }
}
```

测试:

```
package com.kayleoi.springbootdataredis;

import com.kayleoi.springbootdataredis.bean.Member;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.context.web.WebAppConfiguration;

import javax.annotation.Resource;

@RunWith(SpringRunner.class)
```

```
@WebAppConfiguration
@SpringBootTest(classes =SpringBootDataRedisApplication.class)
public class SpringBootDataRedisApplicationTests {

    @Resource
    private RedisTemplate<String,Object> redisTemplate;

    @Test
    public void contextLoads() {
        this.redisTemplate.opsForValue().set("student","james");
        System.out.println(this.redisTemplate.opsForValue().get("student"));

    }

    /**
     * 这里使用的是对象转换为string类型
     */
    @Test
    public void testSet() {
        Member vo = new Member() ;
        vo.setMid("studyjava");
        vo.setAge(19);
        this.redisTemplate.opsForValue().set("study", vo);
    }

    /**
     * 测试得到的值
     */
    @Test
    public void getSet() {
        System.out.println(this.redisTemplate.opsForValue().get("study"));
        //结果：Member{mid='studyjava', age=19}
    }


}
```

springboot 2.0 通过 jedis 集成Redis服务

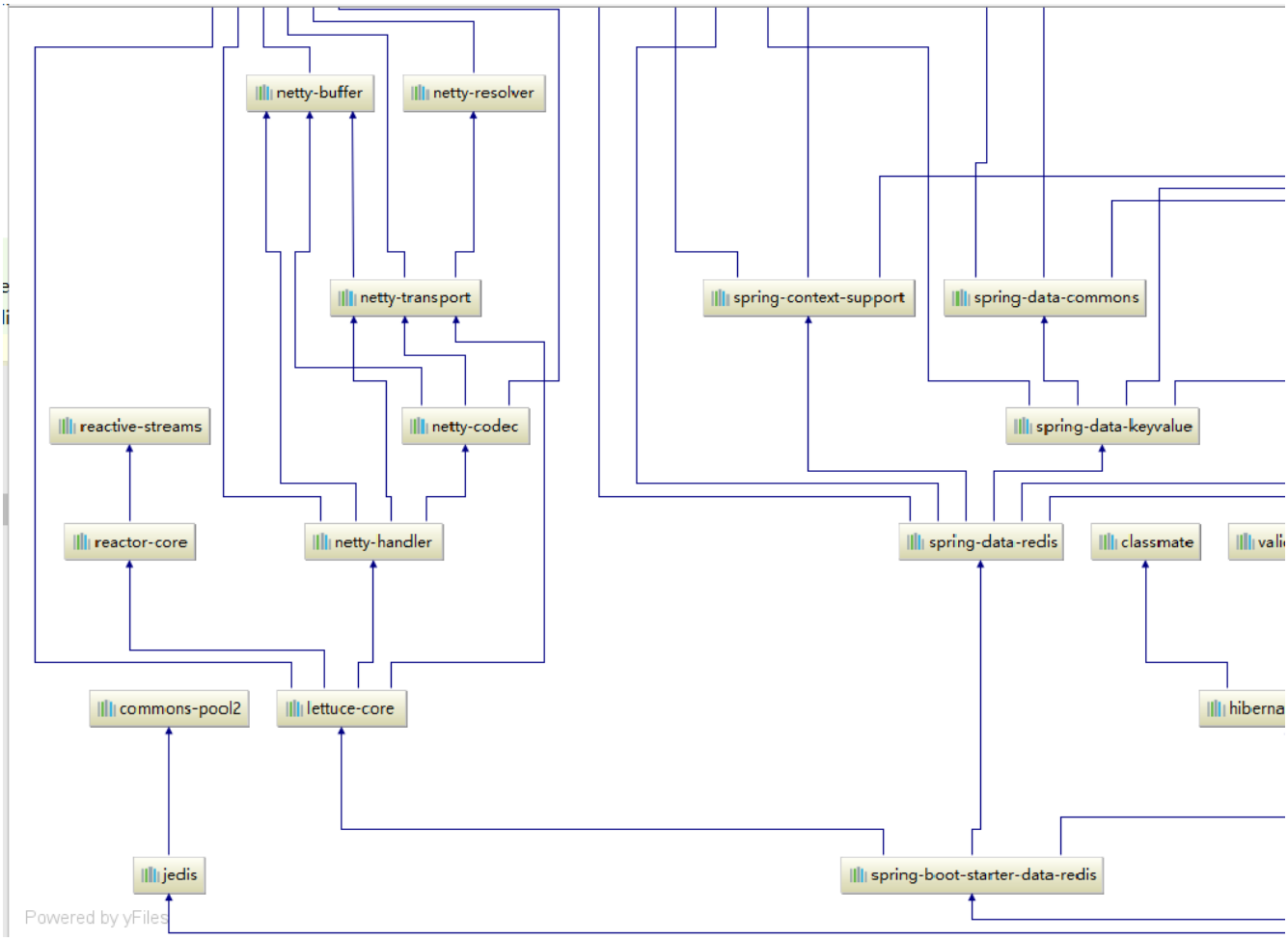
导入依赖

因为 springboot2.0中默认是使用 Lettuce来集成Redis服务，`spring-boot-starter-data-redis`默认只引入了`Lettuce`包，并没有引入`jedis`包支持。所以在我们需要手动引入`jedis`的包，并排除掉`lettuce`的包，pom.xml配置如下:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <exclusions>
        <exclusion>
            <groupId>io.lettuce</groupId>
            <artifactId>lettuce-core</artifactId>
        </exclusion>

    </exclusions>
```

```
</dependency>
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
```



application.properties配置

使用jedis的连接池

```
spring.redis.host=localhost
spring.redis.port=6379
spring.redis.password=root
spring.redis.jedis.pool.max-idle=8
spring.redis.jedis.pool.max-wait=-1ms
spring.redis.jedis.pool.min-idle=0
spring.redis.jedis.pool.max-active=8
```

使用application.yml配置：

```
spring:
  redis:

    host: 192.168.122.123
```

```
    port:
    password:
    timeout:
    database:
    pool:
      max-active: 10
      max-idle: 8
      min-idle: 2
      max-wait: 100

  redis-two:
    host: 192.168.122.124
      port:
      password:
      timeout:
      database:
      pool:
        max-active: 10
        max-idle: 8
        min-idle: 2
        max-wait: 100
```

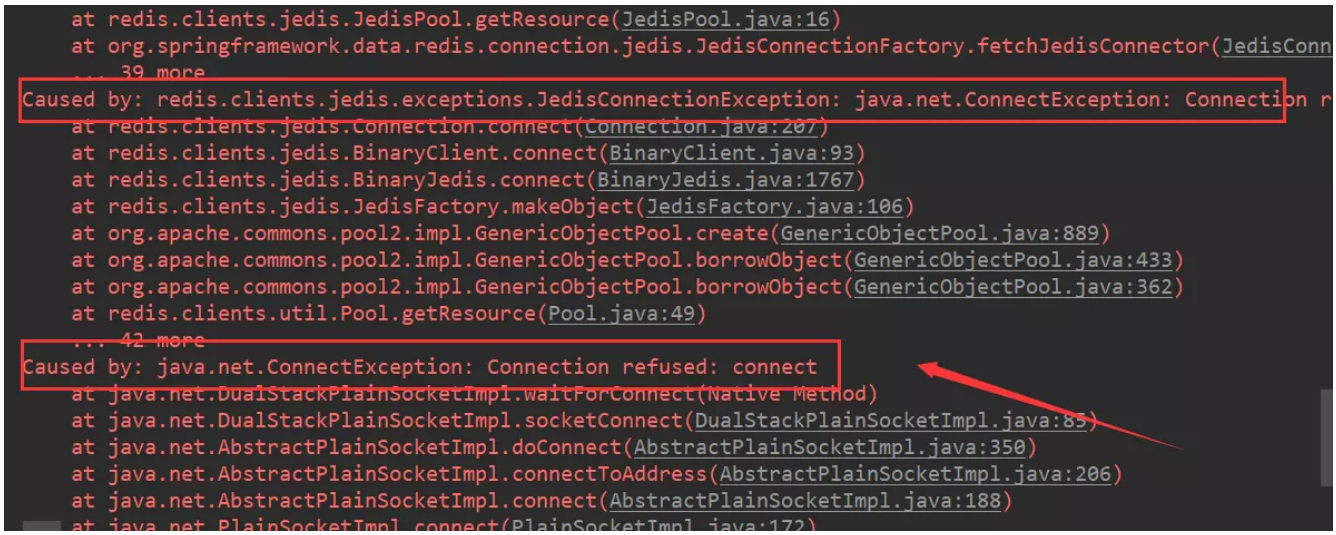
配置 JedisConnectionFactory

因为在 springoot 2.x版本中，默认采用的是 Lettuce实现的，所以无法初始化出 Jedis的连接对象 JedisConnectionFactory，所以我们需要手动配置并注入使用spring官方的

```
/**
 * @Author kay三石
 * @date:2019/7/1
 * Jedis是一个社区驱动的连接器，
 * 由Spring Data Redis模块通过org.springframework.data.redis.connection.jedis包支持。在最简单的形式中，Jedis配置如下所示
 */
@Configuration
public class Appconfig {
    /**
     * 标准配置
     * @return
     */
//    @Bean
//    public JedisConnectionFactory redisConnectionFactory() {
//        return new JedisConnectionFactory();
//    }
    /**
     *  但是，对于生产用途，您可能需要调整主机或密码等设置，如以下示例所示：
     */
    @Bean
    public JedisConnectionFactory redisConnectionFactory() {

        RedisStandaloneConfiguration config = new RedisStandaloneConfiguration("server", 6379);
        return new JedisConnectionFactory(config);
    }
}
```

但是启动项目后发现报出了如下的异常：



redis连接失败，springboot2.x通过以上方式集成Redis并不会读取配置文件中的 `spring.redis.host` 等这样的配置，需要手动配置,如下：

```
@Configuration
public class RedisConfig2 {
    @Value("${spring.redis.host}")
    private String host;
    @Value("${spring.redis.port}")
    private int port;
    @Value("${spring.redis.password}")
    private String password;
    @Bean
    public RedisTemplate<String, Serializable> redisTemplate(JedisConnectionFactory connectionFactory) {
        RedisTemplate<String, Serializable> redisTemplate = new RedisTemplate<>();
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        redisTemplate.setValueSerializer(new GenericJackson2JsonRedisSerializer());
        redisTemplate.setConnectionFactory(jedisConnectionFactory());
        return redisTemplate;
    }
    @Bean
    public JedisConnectionFactory jedisConnectionFactory() {
        RedisStandaloneConfiguration config = new RedisStandaloneConfiguration();
        config.setHostName(host);
        config.setPort(port);
        config.setPassword(RedisPassword.of(password));
        JedisConnectionFactory connectionFactory = new JedisConnectionFactory(config);
        return connectionFactory;
    }
}
```

通过以上方式就可以连接上 redis了，不过这里要提醒的一点就是，在springboot 2.x版本中 `JedisConnectionFactory` 设置连接的方法已过时

在 `springboot 2.x` 版本中推荐使用 `RedisStandaloneConfiguration` 类来设置连接的端口，地址等属性

然后是单元测试

Jedis集群，单机，连接池版

```java
package com.kayleoi.springbootdataredis.jedis;

import java.util.List;

/**
 * @Author kay三石
 * @date:2019/7/1
 * 使用jedis操作
 */
public interface JedisClient {
    String set(String key, String value);
    String get(String key);
    Boolean exists(String key);
    Long expire(String key, int seconds);
    Long ttl(String key);
    Long incr(String key);
    Long hset(String key, String field, String value);
    String hget(String key, String field);
    Long hdel(String key, String... field);
    Boolean hexists(String key, String field);
    List<String> hvals(String key);
    Long del(String key);
}
package com.kayleoi.springbootdataredis.jedis;

import redis.clients.jedis.JedisCluster;

import java.util.List;

/**
 * @Author kay三石
 * @date:2019/7/1
 * 使用jedis操作集群方式
 */
public class JedisClientCluster implements  JedisClient {


    private JedisCluster jedisCluster;


    public JedisCluster getJedisCluster() {
        return jedisCluster;
    }

    public void setJedisCluster(JedisCluster jedisCluster) {
        this.jedisCluster = jedisCluster;
    }

    @Override
    public String set(String key, String value) {
        return jedisCluster.set(key, value);
    }

    @Override
    public String get(String key) {
        return jedisCluster.get(key);

    }
```

```
    @Override
    public Boolean exists(String key) {
        return jedisCluster.exists(key);
    }

    @Override
    public Long expire(String key, int seconds) {
        return jedisCluster.expire(key, seconds);
    }

    @Override
    public Long ttl(String key) {
        return jedisCluster.ttl(key);
    }

    @Override
    public Long incr(String key) {
        return jedisCluster.incr(key);
    }

    @Override
    public Long hset(String key, String field, String value) {
        return jedisCluster.hset(key, field, value);
    }

    @Override
    public String hget(String key, String field) {
        return jedisCluster.hget(key, field);
    }

    @Override
    public Long hdel(String key, String... field) {
        return jedisCluster.hdel(key, field);
    }

    @Override
    public Boolean hexists(String key, String field) {
        return jedisCluster.hexists(key, field);
    }

    @Override
    public List<String> hvals(String key) {
        return jedisCluster.hvals(key);
    }

    @Override
    public Long del(String key) {
        return jedisCluster.del(key);
    }
}
package com.kayleoi.springbootdataredis.jedis;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;

import java.util.List;
```

```java
/**
 * @Author kay三石
 * @date:2019/7/1
 * 使用jedis操作jedis连接池
 */
public class JedisClientPool  implements  JedisClient{
    /**
     *在不同的线程中使用相同的Jedis实例会发生奇怪的错误。但是创建太多的实现也不好因为这意味着会建立很多sokcet连接，
     * 也会导致奇怪的错误发生。单一Jedis实例不是线程安全的。为了避免这些问题，可以使用JedisPool，
     * JedisPool是一个线程安全的网络连接池。
     * 可以用JedisPool创建一些可靠Jedis实例，可以从池中拿到Jedis的实例。这种方式可以解决那些问题并且会实现高效的性
能.
     */
    private JedisPool jedisPool;

    public JedisPool getJedisPool() {
        return jedisPool;
    }

    public void setJedisPool(JedisPool jedisPool) {
        this.jedisPool = jedisPool;
    }

    @Override
    public String set(String key, String value) {
        Jedis jedis = jedisPool.getResource();
        String result = jedis.set(key, value);
        jedis.close();
        return result;
    }

    @Override
    public String get(String key) {
        Jedis jedis = jedisPool.getResource();
        String result = jedis.get(key);
        jedis.close();
        return result;
    }

    @Override
    public Boolean exists(String key) {
        Jedis jedis = jedisPool.getResource();
        Boolean result = jedis.exists(key);
        jedis.close();
        return result;
    }

    @Override
    public Long expire(String key, int seconds) {
        Jedis jedis = jedisPool.getResource();
        Long result = jedis.expire(key, seconds);
        jedis.close();
        return result;
    }

    @Override
    public Long ttl(String key) {
        Jedis jedis = jedisPool.getResource();
```

```
        Long result = jedis.ttl(key);
        jedis.close();
        return result;
    }

    @Override
    public Long incr(String key) {
        Jedis jedis = jedisPool.getResource();
        Long result = jedis.incr(key);
        jedis.close();
        return result;
    }

    @Override
    public Long hset(String key, String field, String value) {
        Jedis jedis = jedisPool.getResource();
        Long result = jedis.hset(key, field, value);
        jedis.close();
        return result;
    }

    @Override
    public String hget(String key, String field) {
        Jedis jedis = jedisPool.getResource();
        String result = jedis.hget(key, field);
        jedis.close();
        return result;
    }

    @Override
    public Long hdel(String key, String... field) {
        Jedis jedis = jedisPool.getResource();
        Long result = jedis.hdel(key, field);
        jedis.close();
        return result;
    }

    @Override
    public Boolean hexists(String key, String field) {
        Jedis jedis = jedisPool.getResource();
        Boolean result = jedis.hexists(key, field);
        jedis.close();
        return result;
    }

    @Override
    public List<String> hvals(String key) {
        Jedis jedis = jedisPool.getResource();
        List<String> result = jedis.hvals(key);
        jedis.close();
        return result;
    }

    @Override
    public Long del(String key) {
        Jedis jedis = jedisPool.getResource();
        Long result = jedis.del(key);
        jedis.close();
```

```
        return result;
    }
}
package com.kayleoi.springbootdataredis.jedis;

import org.junit.Test;
import redis.clients.jedis.HostAndPort;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisCluster;
import redis.clients.jedis.JedisPool;

import java.util.HashSet;
import java.util.Set;

/**
 * @Author kay三石
 * @date:2019/7/1
 */
public class JedisTest {
    /**
     * 测试单机的jedis
     * @throws Exception
     */
    @Test
    public void testJedis() throws Exception {
        //创建一个连接Jedis对象，参数：host、port
        Jedis jedis = new Jedis("localhost", 6379);
        //直接使用jedis操作redis。所有jedis的命令都对应一个方法。
        jedis.set("test123", "my first jedis test");
        String string = jedis.get("test123");
        System.out.println(string);
        //关闭连接
        jedis.close();
    }

    /**
     * 测试jedis连接池形式
     * @throws Exception
     */
    @Test
    public void testJedisPool() throws Exception {
        //创建一个连接池对象，两个参数host、port
        JedisPool jedisPool = new JedisPool("localhost", 6379);
        //从连接池获得一个连接，就是一个jedis对象。
        Jedis jedis = jedisPool.getResource();
        //使用jedis操作redis
        String string = jedis.get("test123");
        System.out.println(string);
        //关闭连接，每次使用完毕后关闭连接。连接池回收资源。
        jedis.close();
        //关闭连接池。
        jedisPool.close();
    }

    /**
     * 测试集群的方式
     * @throws Exception
     */
```

```
    @Test
    public void testJedisCluster() throws Exception {
        //创建一个JedisCluster对象。有一个参数nodes是一个set类型。set中包含若干个HostAndPort对象。
        Set<HostAndPort> nodes = new HashSet<>();
        nodes.add(new HostAndPort("192.168.25.162", 7001));
        nodes.add(new HostAndPort("192.168.25.162", 7002));
        nodes.add(new HostAndPort("192.168.25.162", 7003));
        nodes.add(new HostAndPort("192.168.25.162", 7004));
        nodes.add(new HostAndPort("192.168.25.162", 7005));
        nodes.add(new HostAndPort("192.168.25.162", 7006));
        JedisCluster jedisCluster = new JedisCluster(nodes);
        //直接使用JedisCluster对象操作redis。
        jedisCluster.set("test", "123");
        String string = jedisCluster.get("test");
        System.out.println(string);
        //关闭JedisCluster对象
        jedisCluster.close();
    }

}
```

本文章参考博文：

https://blog.csdn.net/qq_37256896/article/details/94155681

https://blog.csdn.net/qq_37256896/article/details/94153205

https://lettuce.io/docs/getting-started.html

https://juejin.im/post/5ba0a098f265da0adb30c684

https://docs.spring.io/spring-data/redis/docs/2.1.9.RELEASE/reference/html/