Spring Data Redis

Costin Leau, Jennifer Hickey, Christoph Strobl, Thomas Darimont, Mark PaluchVersion 2.0.3.RELEASE,2018-01-24

© 2011-2016 The original authors.

Translator: 杨德军

目录

[Spring Data Redis 1](#_Toc505119384)

[Preface 2](#_Toc505119385)

[1. New Features 2](#_Toc505119386)

[1.1. New in Spring Data Redis 2.0 2](#_Toc505119387)

[1.2. New in Spring Data Redis 1.8 2](#_Toc505119388)

[1.3. New in Spring Data Redis 1.7 3](#_Toc505119389)

[1.4. New in Spring Data Redis 1.6 3](#_Toc505119390)

[1.5. New in Spring Data Redis 1.5 3](#_Toc505119391)

[Introduction 3](#_Toc505119392)

[2. Why Spring Data Redis? 3](#_Toc505119393)

[3. Requirements 4](#_Toc505119394)

[4. Getting Started 4](#_Toc505119395)

[4.1. First Steps 4](#_Toc505119396)

[4.2. Need Help? 5](#_Toc505119397)

[4.3. Following Development 5](#_Toc505119398)

[Reference Documentation 6](#_Toc505119399)

[Document structure 6](#_Toc505119400)

[5. Redis support 6](#_Toc505119401)

[5.1. Redis Requirements 7](#_Toc505119402)

[5.2. Redis Support High Level View 7](#_Toc505119403)

[5.3. Connecting to Redis 7](#_Toc505119404)

[5.4. Redis Sentinel Support 9](#_Toc505119405)

[5.5. Working with Objects through RedisTemplate 10](#_Toc505119406)

[5.6. String-focused convenience classes 13](#_Toc505119407)

[5.7. Serializers 14](#_Toc505119408)

[5.8. Hash mapping 15](#_Toc505119409)

[5.9. Redis Messaging/PubSub 18](#_Toc505119410)

[5.10. Redis Transactions 22](#_Toc505119411)

[5.11. Pipelining 24](#_Toc505119412)

[5.12. Redis Scripting 25](#_Toc505119413)

[5.13. Support Classes 27](#_Toc505119414)

[6. Reactive Redis support 30](#_Toc505119415)

[6.1. Redis Requirements 30](#_Toc505119416)

[6.2. Connecting to Redis using a reactive driver 31](#_Toc505119417)

[6.3. Working with Objects through ReactiveRedisTemplate 32](#_Toc505119418)

[6.4. Reactive Scripting 34](#_Toc505119419)

[7. Redis Cluster 34](#_Toc505119420)

[7.1. Enabling Redis Cluster 35](#_Toc505119421)

[7.2. Working With Redis Cluster Connection 36](#_Toc505119422)

[7.3. Working With RedisTemplate and ClusterOperations 39](#_Toc505119423)

[8. Redis Repositories 39](#_Toc505119424)

[8.1. Usage 39](#_Toc505119425)

[8.2. Object to Hash Mapping 41](#_Toc505119426)

[8.3. Keyspaces 45](#_Toc505119427)

[8.4. Secondary Indexes 46](#_Toc505119428)

[8.5. Time To Live 50](#_Toc505119429)

[8.6. Persisting References 52](#_Toc505119430)

[8.7. Persisting Partial Updates 52](#_Toc505119431)

[8.8. Queries and Query Methods 53](#_Toc505119432)

[8.9. Redis Repositories running on Cluster 55](#_Toc505119433)

[8.10. CDI integration 56](#_Toc505119434)

[Appendixes 58](#_Toc505119435)

[Appendix Document structure 58](#_Toc505119436)

[Appendix A: Schema 58](#_Toc505119437)

[Core schema 58](#_Toc505119438)

[Appendix B: Command Reference 64](#_Toc505119439)

[Supported commands 64](#_Toc505119440)

前言

Spring Data Redis项目将核心Spring概念应用于使用key-value样式数据存储的解决方案开发。 我们提供了一个“template”作为发送和接收消息的高级抽象(high-level)。 你会注意到Spring框架中JDBC支持的相似之处。

1. 新功能

新的和最值得注意的最新版本。

1.1. Spring Data Redis 2.0中的新功能

* 升级到 Java 8
* 升级到 Lettuce 5.0.
* 删除了对 SRP 和 JRedis drivers 的支持.
* [Reactive connection support using Lettuce](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis:reactive).
* 为RedisConnection引入Redis feature-specific interfaces
* 通过JedisClientConfiguration 和LettuceClientConfiguration改进了 RedisConnectionFactory配置
* 修订了RedisCache实现
* 用Redis 3.2.的count命令添加SPOP

1.2. Spring Data Redis 1.8中的新功能

* 升级到Jedis 2.9.
* 升级到 Lettuce 4.2 (Note: Lettuce 4.2 需要 Java 8).
* 支持Redis [GEO](http://redis.io/commands#geo) 命令.
* 使用Spring Data Repository 抽象支持Geospatial Indexes (请参阅： [Geospatial Index](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.indexes.geospatial)).
* MappingRedisConverter 基于 HashMapper 实现(请参阅：[Hash mapping](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.hashmappers.root)).
* 支持存储库支持中的 PartialUpdate (请参阅： [Persisting Partial Updates](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.partial-updates)).
* SSL支持连接到Redis集群(cluster)。
* 使用Jedis时，通过ConnectionFactory  支持客户端名称。

1.3. Spring Data Redis 1.7中的新功能

* 支持 [RedisCluster](http://redis.io/topics/cluster-tutorial).
* 支持Spring Data Repository抽象(请参阅： [Redis Repositories](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories)).

1.4. Spring Data Redis 1.6的新功能

* Lettuce Redis driver 从[wg/lettuce](https://github.com/wg/lettuce) 切换到 [mp911de/lettuce](https://github.com/mp911de/lettuce).
* 支持 ZRANGEBYLEX.
* 增强了ZSET的范围操作，包括 +inf / -inf.
* RedisCache 中的性能改进现在可以更早的释放连接。
* 通用 Jackson2 RedisSerializer 利用Jackson的多台反序列化。

1.5. Spring Data Redis 1.5新功能

* 添加对Redis HyperLogLog 命令PFADD, PFCOUNT 和 PFMERGE的支持.
* 基于Jackson的RedisSerializers 的可配置 JavaType 查找
* 用于连接到Redis Sentinel 的基于PropertySource 的配置 (请参阅: [Redis Sentinel Support](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis:sentinel)).

介绍

本文档是Spring Data Redis（SDR）支持的参考指南。 它解释了Key Value module的概念和语义以及各种商店命名空间(stores namespaces)的语法。

有关key value stores or Spring, or Spring Data examples的介绍，请参阅 [Getting Started](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#get-started)  - 本文档仅涉及Spring Data Redis支持，并假定用户熟悉关键值存储和Spring概念。

2. Why Spring Data Redis?

Spring框架是领先的全堆栈Java / JEE应用程序框架。 它通过使用依赖注入，AOP和便携式服务抽象提供了一个轻量级的容器和一个非侵入式的编程模型。

[NoSQL](https://en.wikipedia.org/wiki/NoSQL) 存储为经典RDBMS提供了一种水平可伸缩性和速度的替代方案。 在实现方面，Key Value stores是NoSQL领域最大（也是最老）的成员之一。

通过Spring的Redis（或SDR）框架，通过Spring的卓越的基础设施支持，可以轻松编写使用Redis key value store的Spring应用程序，从而消除了与store交互所需的冗余任务和boiler plate代码。

3. Requirements

Spring Data Redis 1.x二进制文件需要JDK级别6.0及更高版本，以及 [Spring Framework](https://projects.spring.io/spring-framework/) 5.0.3.RELEASE及更高版本。

在key value stores方面，[Redis](http://redis.io/) 2.6.x或更高版本是必需的。 Spring Data Redis目前正在针对最新的3.2版本进行测试。

4. Getting Started

学习新的框架并不总是直截了当的。 在这一节中，我们（Spring Data团队）试图提供一个简单易懂的从Spring Data Redis模块开始的指南。 当然，如果可能的话，随意创建自己的学习“path”，如果可能的话，请报告任何可以帮助他人的文档的改进。

4.1. First Steps

正如“ [Why Spring Data Redis?](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#why-spring-redis)”中所解释的那样，Spring Data Redis（SDR）提供了Spring框架与Redis key value store之间的集成。 因此，熟悉这两种框架（存储或环境，取决于你想如何命名）是非常重要的。 在整个SDR文档中，每个部分提供与相关资源的链接，但是最好事先熟悉这些主题。

4.1.1. Knowing Spring

Spring Data大量使用Spring框架的 [core](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html)功能，如 [IoC](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html)容器， [resource](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#resources) abstract or [AOP](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#aop)infrastructure。 虽然了解Spring API并不重要，但理解它们背后的概念是。 IoC背后的想法至少应该是熟悉的。 也就是说，Spring对Spring的了解越多，她就会越快地拿起Spring Data Redis。 除了详细解释Spring框架的非常全面（有时是解除武装）的文档之外，还有很多关于这个问题的文章，博客文章和书籍 - 请参阅Spring Guides [home page](https://spring.io/guides)以获取更多信息。 一般来说，这应该是开发者想要试用Spring DR的起点。

4.1.2. Knowing NoSQL and Key Value stores

NoSQL stores已经风靡了整个存储世界。 这是一个有着大量解决方案，术语和模式的广阔领域（即使术语本身具有多重意义( [meanings](https://www.google.com/search?q=nosoql+acronym))，情况也会变得更糟）。 尽管一些原则是常见的，但用户在某种程度上熟悉SDR所支持的stores是至关重要的。 了解这些解决方案的最好方法是阅读他们的文档，并遵循他们的例子 - 通常不需要超过5-10分钟的时间，如果你来自RDMBS专用的背景，这些练习很多次 可以成为大开眼界。

4.1.3. Trying Out The Samples

可以在 [http://github.com/spring-projects/spring-data-keyvalue-examples](https://github.com/spring-projects/spring-data-keyvalue-examples)上的专用示例回购中找到关键值存储的各种样本。 对于Spring Data Redis，感兴趣的是retwisj示例，这是一个构建在Redis之上的Twitter克隆，可以在本地运行或部署到云中。 请参阅其 [documentation](http://static.springsource.org/spring-data/data-keyvalue/examples/retwisj/current/)，以下博客[entry](http://blog.springsource.com/2011/04/27/getting-started-redis-spring-cloud-foundry/) 或 [live instance](http://retwisj.cloudfoundry.com/)以获取更多信息。

4.2. Need Help?

如果遇到问题，或者您只是在寻求建议，请随时使用以下链接之一：

4.2.1. Community Support

[Stackoverflow](https://stackoverflow.com/questions/tagged/spring-data)上的Spring Data标签是所有Spring Data（不仅仅是Redis）用户共享信息和互相帮助的信息板。 请注意，注册仅用于发布。

4.2.2. Professional Support

[Pivotal Software, Inc.](https://www.pivotal.io/)是Spring Data和Spring旗下公司，提供专业的，源代码的支持，保证响应时间。

4.3. Following Development

有关Spring数据源代码库的信息，每晚构建和快照构件，请参阅Spring Data home [page](https://spring.io/spring-data)。

您可以通过在 [spring-data](https://stackoverflow.com/questions/tagged/spring-data) or [spring-data-redis](https://stackoverflow.com/questions/tagged/spring-data-redis)上与Stackoverflow上的开发人员交互来帮助Spring Data最好地满足Spring社区的需求。

如果遇到错误或想要改进，请在Spring Data issue  [tracker](https://jira.springsource.org/browse/DATAREDIS)上创建一张ticket。

要了解Spring生态系统的最新消息和最新消息，请订阅Spring Community  [Portal](https://spring.io/)。

最后，您可以在Twitter上关注Spring [blog](https://spring.io/blog/)或项目团队([@SpringData](https://twitter.com/SpringData))。

参考文档

文档结构

参考文档的这一部分介绍了Spring Data Redis提供的核心功能。

[Redis support](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis) 引入了Redis模块功能集。

5. Redis 支持

Spring Data支持的一个key value stores是 [Redis](http://redis.io/)。 引用项目主页：

*Redis是一个先进的键值存储。 它与memcached类似，但数据集不易变，值可以是字符串，就像在memcached中一样，也可以是lists, sets, and ordered sets。 所有这些数据类型都可以使用原子操作来操作push / pop元素，add/remove元素，执行服务器端union, intersection, difference between sets等等。 Redis支持不同类型的排序功能。*

Spring Data Redis提供了从Spring应用程序轻松配置和访问Redis的功能。 它提供了与store互动的低级别和高级别抽象，使用户免受基础设施问题的困扰。

5.1. Redis Requirements

Spring Redis需要Redis 2.6或更高版本以及Java SE 8.0或更高版本。 在语言绑定（或连接器）方面，Spring Redis集成了两个受欢迎的Redis开源Java库 [Jedis](https://github.com/xetorthio/jedis) and [Lettuce](https://github.com/lettuce-io/lettuce-core)。

5.2. Redis Support High Level View

Redis支持提供了几个组件（按依赖顺序）：

对于大多数任务来说，高层次的抽象和支持服务是最好的选择。 请注意，在任何时候，可以在层(layers)之间移动 - 例如，容易保持低层连接(low level connection)（甚至是本地库）与Redis直接通信。

5.3. 连接 Redis

使用Redis和Spring的首要任务之一是通过IoC容器连接到store。 为此，需要Java连接器（或binding）。 无论选择哪个库，只有一组Spring Data Redis API需要使用，它们在所有连接器（即org.springframework.data.redis.connection包及其RedisConnection和RedisConnectionFactory接口）之间运行一致 并检索到Redis的活动连接。

5.3.1. RedisConnection 和 RedisConnectionFactory

RedisConnection为Redis通信提供构建块，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Spring一致的DAO异常层次结构( [hierarchy](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/data-access.html#dao-exceptions))，因为操作语义保持不变，所以可以在不更改任何代码的情况下切换连接器。

|  |  |
| --- | --- |
|  | 对于需要本地库API的情况，RedisConnection提供了一个专用方法getNativeConnection，它返回用于通信的原始底层对象。 |

活动的RedisConnection是通过RedisConnectionFactory创建的。 另外，工厂充当PersistenceExceptionTranslator，意味着一旦声明，他们允许做透明异常转换。 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文档中的专用章节 [section](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/data-access.html#orm-exception-translation)。

|  |  |
| --- | --- |
|  | 根据底层配置，工厂可以返回一个新的连接或一个现有的连接（如果使用池或共享本地连接）。 |

使用RedisConnectionFactory的最简单方法是通过IoC容器配置适当的连接器，并将其注入到使用的类中。

|  |  |
| --- | --- |
|  | 不幸的是，目前并非所有的连接器都支持所有的Redis功能。 调用基础库不支持的Connection API上的方法时，会引发UnsupportedOperationException。 随着各种连接器的成熟，这种情况将来可能会得到解决。 |

5.3.2. 配置 Jedis connector

[Jedis](https://github.com/xetorthio/jedis) 是Spring Data Redis模块通过org.springframework.data.redis.connection.jedis包支持的连接器之一。 以最简单的形式，Jedis配置如下所示：

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

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

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

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

<!-- Jedis ConnectionFactory -->

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

</beans>

但是，对于生产用途，可能需要调整host或password等设置：

<?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:host-name="server" p:port="6379" />

</beans>

5.3.3. 配置 Lettuce connector

[Lettuce](https://github.com/mp911de/lettuce) 是Spring Data Redis通过org.springframework.data.redis.connection.lettuce包支持的[netty](http://netty.io/)-based的开源连接器。

它的配置可能很容易猜到：

<?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="lettuceConnectionFactory" class="org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory" p:host-name="server" p:port="6379"/>

</beans>

还有一些可以调整的Lettuce-specific connection parameters。 默认情况下，由LettuceConnectionFactory创建的所有LettuceConnection共享所有非阻塞和非事务操作的相同的线程安全本机连接。 将shareNativeConnection设置为false，以便每次都使用专用连接。 LettuceConnectionFactory也可以使用LettucePool进行配置，以用于合并阻塞和事务连接，或者在shareNativeConnection设置为false时使用所有连接。

5.4. Redis Sentinel 支持

为了处理高可用性的Redis，使用RedisSentinelConfiguration支持[Redis Sentinel](http://redis.io/topics/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);

}

/\*\*

\* 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 可以通过 PropertySource来定义.  *Configuration Properties*   * spring.redis.sentinel.master: master node 的名称. * spring.redis.sentinel.nodes: host:port. |

有时需要与Sentinels直接互动。 使用RedisConnectionFactory.getSentinelConnection()or RedisConnection.getSentinelCommands()可以访问配置的第一个活动Sentinel。

5.5. 通过RedisTemplate处理对象

大多数用户可能使用 RedisTemplate及其相应的包org.springframework.data.redis.core - 由于其丰富的功能集，该模板实际上是Redis模块的中心类。 该模板提供了Redis交互的高级抽象。 虽然RedisConnection 提供接受和返回二进制值(byte arrays)的底层方法，但是模板负责序列化和连接管理，使用户不必处理这些细节。

此外，该模板还提供了操作视图（来自Redis命令参考 [reference](http://redis.io/commands)的分组），该视图提供丰富的，generified interfaces，用于处理特定类型或特定键（通过KeyBound interfaces），如下所述：

| *Table 1. Operational views* | |
| --- | --- |
| **Interface** | **Description** |
| *Key Type Operations* | |
| GeoOperations | Redis geospatial operations like GEOADD, GEORADIUS,…​) |
| HashOperations | Redis hash operations |
| HyperLogLogOperations | Redis HyperLogLog operations like (PFADD, PFCOUNT,…​) |
| ListOperations | Redis list operations |
| SetOperations | Redis set operations |
| ValueOperations | Redis string (or value) operations |
| ZSetOperations | Redis zset (or sorted set) operations |
| *Key Bound Operations* | |
| BoundGeoOperations | Redis key bound geospatial operations. |
| BoundHashOperations | Redis hash key bound operations |
| BoundKeyOperations | Redis key bound operations |
| BoundListOperations | Redis list key bound operations |
| BoundSetOperations | Redis set key bound operations |
| BoundValueOperations | Redis string (or value) key bound operations |
| BoundZSetOperations | Redis zset (or sorted set) key bound operations |

配置完成后，该模板是线程安全的，可以在多个实例中重复使用。

开箱即用，RedisTemplate在其大部分操作中使用基于Java的序列化程序。 这意味着模板写入或读取的任何对象都将通过Java进行序列化/反序列化。 序列化机制可以很容易地在模板上进行更改，而Redis模块在org.springframework.data.redis.serializer包中提供了几个可用的实现 - 请参阅 [Serializers](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis:serializer)程序以获取更多信息。 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false来对原始字节数组使用RedisTemplate。 请注意，模板要求所有的键都是非空的 - 只要底层的序列化程序接受它们，值就可以为空; 阅读每个序列化器的javadoc以获取更多信息。

对于需要特定模板视图的情况，将视图声明为依赖项并注入模板：容器将自动执行转换，从而消除opsFor [X]调用：

<?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());

}

}

5.6. String-focused convenience classes

由于Redis中存储的键和值通常是java.lang.String，因此Redis模块分别为RedisConnection和RedisTemplate（分别为StringRedisConnection（及其DefaultStringRedisConnection实现）和StringRedisTemplate提供了两个扩展，作为便捷的一站式解决方案 密集的字符串操作 除了绑定到字符串键，模板和连接使用StringRedisSerializer之下，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）。 例如：

<?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());

}

}

与其他Spring模板一样，RedisTemplate和StringRedisTemplate允许开发人员通过RedisCallback接口直接与Redis进行通话。 当它直接与RedisConnection对话时，这给了开发者完全的控制权。 请注意，使用StringRedisTemplate时，回调接收到StringRedisConnection的实例。

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");

}

});

}

5.7. Serializers

从框架的角度来看，存储在Redis中的数据只是字节。 虽然Redis本身支持各种类型，但大部分是指数据存储的方式，而不是它所代表的内容。 由用户决定是否将信息转换为字符串或任何其他对象。

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

这个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`RedisSerializer`的双向穿行器(Two-way).
* Element readers and writers 使用 RedisElementReader and RedisElementWriter.

这些变体之间的主要区别在于，RedisSerializer 主要序列化为byte[]，而读者和作者使用ByteBuffer。

可以直接使用多种实现，其中的两个已经在本文档中提到过了：

* StringRedisSerializer
* JdkSerializationRedisSerializer

但是可以通过Spring [OXM](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/data-access.html#oxm) 支持使用OxmSerializer进行Object / XML映射，或者使用Jackson2JsonRedisSerializer或GenericJackson2JsonRedisSerializer以 [JSON](https://en.wikipedia.org/wiki/JSON)格式存储数据。

请注意，存储格式不仅限于values - 它可以用于keys，值或散列没有任何限制。

5.8. Hash mapping

数据可以使用Redis中的各种数据结构进行存储。 您已经了解到可以转换 [JSON](https://en.wikipedia.org/wiki/JSON)格式的对象的Jackson2JsonRedisSerializer。 JSON可以理想地存储为使用普通键的值。 使用Redis哈希可以实现更复杂的结构化对象映射。 Spring Data Redis提供了根据用例将数据映射到散列的各种策略。

1. 使用 HashOperations 和一个 [serializer](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis:serializer)
2. 使用 [Redis Repositories](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories)
3. 使用 HashMapper 和 HashOperations

5.8.1. Hash mappers

Hash mappers是将对象映射到Map<K, V>和back. HashMapper。 HashMapper旨在与Redis哈希一起使用。

开箱即可使用多种实施方式：

1. BeanUtilsHashMapper using Spring’s [BeanUtils](https://docs.spring.io/spring/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/BeanUtils.html).
2. ObjectHashMapper using [Object to Hash Mapping](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.mapping).
3. [Jackson2HashMapper](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.hashmappers.jackson2) using [FasterXML Jackson](https://github.com/FasterXML/jackson).

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);

}

}

5.8.2. Jackson2HashMapper

Jackson2HashMapper使用 [FasterXML Jackson](https://github.com/FasterXML/jackson)为域对象提供Redis哈希映射。 Jackson2HashMapper可以将数据映射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（nested objects, collections, maps）表示为嵌套JSON。

展平(Flattening)会为所有嵌套属性创建单独的哈希条目(hash entries)，并尽可能将复杂类型解析为简单类型。

public class Person {

String firstname;

String lastname;

Address address;

}

public class Address {

String city;

String country;

}

| *Table 2. Normal Mapping* | |
| --- | --- |
| **Hash Field** | **Value** |
| firstname | Jon |
| lastname | Snow |
| address | { "city" : "Castle Black", "country" : "The North" } |

| *Table 3. Flat Mapping* | |
| --- | --- |
| **Hash Field** | **Value** |
| firstname | Jon |
| lastname | Snow |
| address.city | Castle Black |
| address.country | The North |

|  |  |
| --- | --- |
|  | 展平(Flattening)需要所有属性名称不会干扰JSON路径。 在拼图键(map keys)中使用点或括号或作为属性名称使用拼合不支持。 生成的散列无法映射回对象。 |

5.9. Redis Messaging/PubSub

Spring Data provides dedicated messaging integration for Redis, very similar in functionality and naming to the JMS integration in Spring Framework; in fact, users familiar with the JMS support in Spring should feel right at home.

Redis messaging can be roughly divided into two areas of functionality, namely the production or publication and consumption or subscription of messages, hence the shortcut pubsub (Publish/Subscribe). The RedisTemplate class is used for message production. For asynchronous reception similar to Java EE’s message-driven bean style, Spring Data provides a dedicated message listener container that is used to create Message-Driven POJOs (MDPs) and for synchronous reception, the RedisConnection contract.

The package org.springframework.data.redis.connection and org.springframework.data.redis.listener provide the core functionality for using Redis messaging.

5.9.1. Sending/Publishing messages

To publish a message, one can use, as with the other operations, either the low-level RedisConnection or the high-level RedisTemplate. Both entities offer the publish method that accepts as an argument the message that needs to be sent as well as the destination channel. While RedisConnection requires raw-data (array of bytes), the RedisTemplate allow arbitrary objects to be passed in as messages:

// send message through connection RedisConnection con = ...

byte[] msg = ...

byte[] channel = ...

con.publish(msg, channel); // send message through RedisTemplate

RedisTemplate template = ...

template.convertAndSend("hello!", "world");

5.9.2. Receiving/Subscribing for messages

On the receiving side, one can subscribe to one or multiple channels either by naming them directly or by using pattern matching. The latter approach is quite useful as it not only allows multiple subscriptions to be created with one command but to also listen on channels not yet created at subscription time (as long as they match the pattern).

At the low-level, RedisConnection offers subscribe and pSubscribe methods that map the Redis commands for subscribing by channel respectively by pattern. Note that multiple channels or patterns can be used as arguments. To change the subscription of a connection or simply query whether it is listening or not, RedisConnection provides getSubscription and isSubscribedmethod.

|  |  |
| --- | --- |
|  | Subscription commands in Spring Data Redis are blocking. That is, calling subscribe on a connection will cause the current thread to block as it will start waiting for messages - the thread will be released only if the subscription is canceled, that is an additional thread invokes unsubscribe or pUnsubscribe on the **same**connection. See [message listener container](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis:pubsub:subscribe:containers) below for a solution to this problem. |

As mentioned above, once subscribed a connection starts waiting for messages. No other commands can be invoked on it except for adding new subscriptions or modifying/canceling the existing ones. That is, invoking anything other then subscribe, pSubscribe, unsubscribe, or pUnsubscribe is illegal and will throw an exception.

In order to subscribe for messages, one needs to implement the MessageListener callback: each time a new message arrives, the callback gets invoked and the user code executed through onMessage method. The interface gives access not only to the actual message but to the channel it has been received through and the pattern (if any) used by the subscription to match the channel. This information allows the callee to differentiate between various messages not just by content but also through data.

Message Listener Containers

Due to its blocking nature, low-level subscription is not attractive as it requires connection and thread management for every single listener. To alleviate this problem, Spring Data offers RedisMessageListenerContainer which does all the heavy lifting on behalf of the user - users familiar with EJB and JMS should find the concepts familiar as it is designed as close as possible to the support in Spring Framework and its message-driven POJOs (MDPs)

RedisMessageListenerContainer acts as a message listener container; it is used to receive messages from a Redis channel and drive the MessageListener s that are injected into it. The listener container is responsible for all threading of message reception and dispatches into the listener for processing. A message listener container is the intermediary between an MDP and a messaging provider, and takes care of registering to receive messages, resource acquisition and release, exception conversion and the like. This allows you as an application developer to write the (possibly complex) business logic associated with receiving a message (and reacting to it), and delegates boilerplate Redis infrastructure concerns to the framework.

Furthermore, to minimize the application footprint, RedisMessageListenerContainer allows one connection and one thread to be shared by multiple listeners even though they do not share a subscription. Thus no matter how many listeners or channels an application tracks, the runtime cost will remain the same through out its lifetime. Moreover, the container allows runtime configuration changes so one can add or remove listeners while an application is running without the need for restart. Additionally, the container uses a lazy subscription approach, using a RedisConnection only when needed - if all the listeners are unsubscribed, cleanup is automatically performed and the used thread released.

To help with the asynch manner of messages, the container requires a java.util.concurrent.Executor ( or Spring’s TaskExecutor) for dispatching the messages. Depending on the load, the number of listeners or the runtime environment, one should change or tweak the executor to better serve her needs - in particular in managed environments (such as app servers), it is highly recommended to pick a a proper TaskExecutor to take advantage of its runtime.

The MessageListenerAdapter

The MessageListenerAdapter class is the final component in Spring’s asynchronous messaging support: in a nutshell, it allows you to expose almost **any** class as a MDP (there are of course some constraints).

Consider the following interface definition. Notice that although the interface doesn’t extend the MessageListener interface, it can still be used as a MDP via the use of the MessageListenerAdapter class. Notice also how the various message handling methods are strongly typed according to the **contents** of the various Message types that they can receive and handle. In addition, the channel or pattern to which a message is sent can be passed in to the method as the second argument of type String:

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);

}

public class DefaultMessageDelegate implements MessageDelegate {

// implementation elided for clarity...

}

In particular, note how the above implementation of the MessageDelegate interface (the above DefaultMessageDelegate class) has **no** Redis dependencies at all. It truly is a POJO that we will make into an MDP via the following configuration.

<?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>

|  |  |
| --- | --- |
|  | The listener topic can be either a channel (e.g. topic="chatroom") or a pattern (e.g. topic="\*room") |

The example above uses the Redis namespace to declare the message listener container and automatically register the POJOs as listeners. The full blown, **beans** definition is displayed below:

<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>

Each time a message is received, the adapter automatically performs translation (using the configured RedisSerializer) between the low-level format and the required object type transparently. Any exception caused by the method invocation is caught and handled by the container (by default, being logged).

5.10. Redis Transactions

Redis provides support for [transactions](http://redis.io/topics/transactions) through the multi, exec, and discard commands. These operations are available on RedisTemplate, however RedisTemplate is not guaranteed to execute all operations in the transaction using the same connection.

Spring Data Redis provides the SessionCallback interface for use when multiple operations need to be performed with the same connection, as when using Redis transactions. For example:

//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 ops in the transaction

return operations.exec();

}

});

System.out.println("Number of items added to set: " + txResults.get(0));

RedisTemplate will use its value, hash key, and hash value serializers to deserialize all results of exec before returning. There is an additional exec method that allows you to pass a custom serializer for transaction results.

|  |  |
| --- | --- |
|  | An important change has been made to the exec methods of RedisConnection and RedisTemplate in version 1.1. Previously these methods returned the results of transactions directly from the connectors. This means that the data types often differed from those returned from the methods of RedisConnection. For example, zAddreturns a boolean indicating that the element has been added to the sorted set. Most connectors return this value as a long and Spring Data Redis performs the conversion. Another common difference is that most connectors return a status reply (usually the String "OK") for operations like set. These replies are typically discarded by Spring Data Redis. Prior to 1.1, these conversions were not performed on the results of exec. Also, results were not deserialized in RedisTemplate, so they often included raw byte arrays. If this change breaks your application, you can set convertPipelineAndTxResults to false on your RedisConnectionFactory to disable this behavior. |

5.10.1. @Transactional Support

Transaction Support is disabled by default and has to be explicitly enabled for each RedisTemplate in use by setting setEnableTransactionSupport(true). This will force binding the RedisConnection in use to the current Thread triggering MULTI. If the transaction finishes without errors, EXEC is called, otherwise DISCARD. Once in MULTI, RedisConnection would queue write operations, all readonly operations, such as KEYS are piped to a fresh (non thread bound) RedisConnection.

/\*\* Sample Configuration \*\*/

@Configuration

public class RedisTxContextConfiguration {

@Bean

public StringRedisTemplate redisTemplate() {

StringRedisTemplate template = new StringRedisTemplate(redisConnectionFactory());

// explicitly enable transaction support

template.setEnableTransactionSupport(true);

return template;

}

@Bean

public PlatformTransactionManager transactionManager() throws SQLException {

return new DataSourceTransactionManager(dataSource());

}

@Bean

public RedisConnectionFactory redisConnectionFactory( // jedis || lettuce);

@Bean

public DataSource dataSource() throws SQLException { // ... }

}

/\*\* Usage Constrainsts \*\*/

// executed on thread bound connection

template.opsForValue().set("foo", "bar");

// read operation executed on a free (not tx-aware)

connection template.keys("\*");

// returns null as values set within transaction are not visible

template.opsForValue().get("foo");

5.11. Pipelining

Redis provides support for [pipelining](http://redis.io/topics/pipelining), which involves sending multiple commands to the server without waiting for the replies and then reading the replies in a single step. Pipelining can improve performance when you need to send several commands in a row, such as adding many elements to the same List.

Spring Data Redis provides several RedisTemplate methods for executing commands in a pipeline. If you don’t care about the results of the pipelined operations, you can use the standard execute method, passing true for the pipeline argument. The executePipelined methods will execute the provided RedisCallback or SessionCallback in a pipeline and return the results. For example:

//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;

}

});

The example above executes a bulk right pop of items from a queue in a pipeline. The results List contains all of the popped items. RedisTemplate uses its value, hash key, and hash value serializers to deserialize all results before returning, so the returned items in the above example will be Strings. There are additional executePipelined methods that allow you to pass a custom serializer for pipelined results.

Note that the value returned from the RedisCallback is required to be null, as this value is discarded in favor of returning the results of the pipelined commands.

|  |  |
| --- | --- |
|  | An important change has been made to the closePipeline method of RedisConnection in version 1.1. Previously this method returned the results of pipelined operations directly from the connectors. This means that the data types often differed from those returned by the methods of RedisConnection. For example, zAddreturns a boolean indicating that the element has been added to the sorted set. Most connectors return this value as a long and Spring Data Redis performs the conversion. Another common difference is that most connectors return a status reply (usually the String "OK") for operations like set. These replies are typically discarded by Spring Data Redis. Prior to 1.1, these conversions were not performed on the results of closePipeline. If this change breaks your application, you can set convertPipelineAndTxResults to false on your RedisConnectionFactory to disable this behavior. |

5.12. Redis Scripting

Redis versions 2.6 and higher provide support for execution of Lua scripts through the [eval](http://redis.io/commands/eval) and [evalsha](http://redis.io/commands/evalsha) commands. Spring Data Redis provides a high-level abstraction for script execution that handles serialization and automatically makes use of the Redis script cache.

Scripts can be run through the execute methods of RedisTemplate and ReactiveRedisTemplate. Both use a configurable ScriptExecutor / ReactiveScriptExecutor to run the provided script. By default, the ScriptExecutor takes care of serializing the provided keys and arguments and deserializing the script result. This is done via the key and value serializers of the template. There is an additional overload that allows you to pass custom serializers for the script arguments and result.

The default ScriptExecutor optimizes performance by retrieving the SHA1 of the script and attempting first to run evalsha, falling back to eval if the script is not yet present in the Redis script cache.

Here’s an example that executes a common "check-and-set" scenario using a Lua script. This is an ideal use case for a Redis script, as it requires that we execute a set of commands atomically and the behavior of one command is influenced by the result of another.

@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

The code above configures a RedisScript pointing to a file called checkandset.lua, which is expected to return a boolean value. The script resultType should be one of Long, Boolean, List, or deserialized value type. It can also be null if the script returns a throw-away status (i.e "OK"). It is ideal to configure a single instance of DefaultRedisScript in your application context to avoid re-calculation of the script’s SHA1 on every script execution.

The checkAndSet method above then executes th Scripts can be executed within a SessionCallback as part of a transaction or pipeline. See [Redis Transactions](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#tx) and [Pipelining](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#pipeline) for more information.

The scripting support provided by Spring Data Redis also allows you to schedule Redis scripts for periodic execution using the Spring Task and Scheduler abstractions. See the Spring Framework documentation for more details.

5.13. Support Classes

Package org.springframework.data.redis.support offers various reusable components that rely on Redis as a backing store. Currently the package contains various JDK-based interface implementations on top of Redis such as [atomic](http://download.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html) counters and JDK[Collections](http://download.oracle.com/javase/8/docs/api/java/util/Collection.html).

The atomic counters make it easy to wrap Redis key incrementation while the collections allow easy management of Redis keys with minimal storage exposure or API leakage: in particular the RedisSet and RedisZSet interfaces offer easy access to the **set**operations supported by Redis such as intersection and union while RedisList implements the List, Queue and Dequecontracts (and their equivalent blocking siblings) on top of Redis, exposing the storage as a *FIFO (First-In-First-Out)*, *LIFO (Last-In-First-Out)* or *capped collection* with minimal configuration:

<?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>

public class AnotherExample {

// injected

private Deque<String> queue;

public void addTag(String tag) {

queue.push(tag);

}

}

As shown in the example above, the consuming code is decoupled from the actual storage implementation - in fact there is no indication that Redis is used underneath. This makes moving from development to production environments transparent and highly increases testability (the Redis implementation can just as well be replaced with an in-memory one).

5.13.1. Support for Spring Cache Abstraction

|  |  |
| --- | --- |
|  | Changed in 2.0 |

Spring Redis provides an implementation for Spring [cache abstraction](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/integration.html#cache) through the org.springframework.data.redis.cachepackage. To use Redis as a backing implementation, simply add RedisCacheManager to your configuration:

@Bean

public RedisCacheManager cacheManager(RedisConnectionFactory connectionFactory) {

return RedisCacheManager.create(connectionFactory);

}

RedisCacheManager behavior can be configured via RedisCacheManagerBuilder allowing to set the default RedisCacheConfiguration, transaction behaviour and predefined caches.

RedisCacheManager cm = RedisCacheManager.builder(connectionFactory)

.cacheDefaults(defaultCacheConfig())

.initialCacheConfigurations(singletonMap("predefined", defaultCacheConfig().disableCachingNullValues()))

.transactionAware()

.build();

Behavior of RedisCache created via RedisCacheManager is defined via RedisCacheConfiguration. The configuration allows to set key expiration times, prefixes and RedisSerializers for converting to and from the binary storage format. As shown above RedisCacheManager allows definition of configurations on a per cache base.

RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig()

.entryTtl(Duration.ofSeconds(1))

.disableCachingNullValues();

RedisCacheManager defaults to a lock-free RedisCacheWriter for reading & writing binary values. Lock-free caching improves throughput. The lack of entry locking can lead to overlapping, non atomic commands, for putIfAbsent and clean methods as those require multiple commands sent to Redis. The locking counterpart prevents command overlap by setting an explicit lock key and checking against presence of this key, which leads to additional requests and potential command wait times.

It is possible to opt in to the locking behavior as follows:

RedisCacheManager cm = RedisCacheManager.build(RedisCacheWriter.lockingRedisCacheWriter())

.cacheDefaults(defaultCacheConfig())

...

| *Table 4. RedisCacheManager defaults* | |
| --- | --- |
| **Setting** | **Value** |
| Cache Writer | non locking |
| Cache Configuration | RedisCacheConfiguration#defaultConfiguration |
| Initial Caches | none |
| Trasaction Aware | no |

| *Table 5. RedisCacheConfiguration defaults* | |
| --- | --- |
| **Key Expiration** | **none** |
| Cache null | yes |
| Prefix Keys | yes |
| Default Prefix | the actual cache name |
| Key Serializer | StringRedisSerializer |
| Value Serializer | JdkSerializationRedisSerializer |
| Conversion Service | DefaultFormattingConversionService with default cache key converters |

6. Reactive Redis support

This section covers reactive Redis support and how to get started. You will find certain overlaps with the [imperative Redis support](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis).

6.1. Redis Requirements

Spring Data Redis requires Redis 2.6 or above and Java SE 8.0 or above. In terms of language bindings (or connectors), Spring Data Redis currently integrates with [Lettuce](https://github.com/lettuce-io/lettuce-core) as the only reactive Java connector. [Project Reactor](https://projectreactor.io/) is used as reactive composition library.

6.2. Connecting to Redis using a reactive driver

One of the first tasks when using Redis and Spring is to connect to the store through the IoC container. To do that, a Java connector (or binding) is required. No matter the library one chooses, there is only one set of Spring Data Redis API that one needs to use that behaves consistently across all connectors, namely the org.springframework.data.redis.connection package and its ReactiveRedisConnection and ReactiveRedisConnectionFactory interfaces for working with and retrieving active connections to Redis.

6.2.1. Redis Operation Modes

Redis can be run as standalone server, with [Redis Sentinel](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis:sentinel) or in [Redis Cluster](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#cluster) mode. [Lettuce](https://github.com/lettuce-io/lettuce-core) supports all above mentioned connection types.

6.2.2. ReactiveRedisConnection and ReactiveRedisConnectionFactory

ReactiveRedisConnection provides the building block for Redis communication as it handles the communication with the Redis back-end. It also automatically translates the underlying driver exceptions to Spring’s consistent DAO exception [hierarchy](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/data-access.html#dao-exceptions) so one can switch the connectors without any code changes as the operation semantics remain the same.

Active ReactiveRedisConnections are created through ReactiveRedisConnectionFactory. In addition, the factories act as PersistenceExceptionTranslators, meaning once declared, they allow one to do transparent exception translation. For example, exception translation through the use of the @Repository annotation and AOP. For more information see the dedicated [section](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/data-access.html#orm-exception-translation) in Spring Framework documentation.

|  |  |
| --- | --- |
|  | Depending on the underlying configuration, the factory can return a new connection or an existing connection (in case a pool or shared native connection is used). |

The easiest way to work with a ReactiveRedisConnectionFactory is to configure the appropriate connector through the IoC container and inject it into the using class.

6.2.3. Configuring Lettuce connector

[Lettuce](https://github.com/lettuce-io/lettuce-core) is supported by Spring Data Redis through the org.springframework.data.redis.connection.lettuce package.

Setting up ReactiveRedisConnectionFactory for Lettuce can be done as follows:

@Bean

public ReactiveRedisConnectionFactory connectionFactory() {

return new LettuceConnectionFactory("localhost", 6379);

}

A more sophisticated configuration, including SSL and timeouts, using LettuceClientConfigurationBuilder might look like below:

@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);

}

For more detailed client configuration tweaks have a look at LettuceClientConfiguration.

6.3. Working with Objects through ReactiveRedisTemplate

Most users are likely to use ReactiveRedisTemplate and its corresponding package org.springframework.data.redis.core - the template is in fact the central class of the Redis module due to its rich feature set. The template offers a high-level abstraction for Redis interactions. While ReactiveRedisConnection offers low level methods that accept and return binary values (ByteBuffer), the template takes care of serialization and connection management, freeing the user from dealing with such details.

Moreover, the template provides operation views (following the grouping from Redis command [reference](http://redis.io/commands)) that offer rich, generified interfaces for working against a certain type as described below:

| *Table 6. Operational views* | |
| --- | --- |
| **Interface** | **Description** |
| *Key Type Operations* | |
| ReactiveGeoOperations | Redis geospatial operations like GEOADD, GEORADIUS,…​) |
| ReactiveHashOperations | Redis hash operations |
| ReactiveHyperLogLogOperations | Redis HyperLogLog operations like (PFADD, PFCOUNT,…​) |
| ReactiveListOperations | Redis list operations |
| ReactiveSetOperations | Redis set operations |
| ReactiveValueOperations | Redis string (or value) operations |
| ReactiveZSetOperations | Redis zset (or sorted set) operations |

Once configured, the template is thread-safe and can be reused across multiple instances.

Out of the box, ReactiveRedisTemplate uses a Java-based serializer for most of its operations. This means that any object written or read by the template will be serialized/deserialized through RedisElementWriter respective RedisElementReader. The serialization context is passed to the template upon construction, and the Redis module offers several implementations available in the org.springframework.data.redis.serializer package - see [Serializers](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis:serializer) for more information.

@Configuration

class RedisConfiguration {

@Bean

ReactiveRedisTemplate<String, String> reactiveRedisTemplate(ReactiveRedisConnectionFactory factory) {

return new ReactiveRedisTemplate<>(connectionFactory, 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());

}

}

6.4. Reactive Scripting

Executing Redis scripts via the reactive infrastructure can be done using the ReactiveScriptExecutor accessed best via 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);

}

}

Please refer to the [scripting section](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#scripting) for more details on scripting commands.

7. Redis Cluster

Working with [Redis Cluster](http://redis.io/topics/cluster-spec) requires a Redis Server version 3.0+ and provides a very own set of features and capabilities. Please refer to the [Cluster Tutorial](http://redis.io/topics/cluster-tutorial) for more information.

7.1. Enabling Redis Cluster

Cluster support is based on the very same building blocks as non clustered communication. RedisClusterConnection an extension to RedisConnection handles the communication with the Redis Cluster and translates errors into the Spring DAO exception hierarchy. RedisClusterConnection 's are created via the RedisConnectionFactory which has to be set up with the according RedisClusterConfiguration.

*Example 1. Sample RedisConnectionFactory Configuration for Redis Cluster*

@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 can also be defined via PropertySource.  *Configuration Properties*   * spring.redis.cluster.nodes: Comma delimited list of host:port pairs. * spring.redis.cluster.max-redirects: Number of allowed cluster redirections. |
|  | The initial configuration points driver libraries to an initial set of cluster nodes. Changes resulting from live cluster reconfiguration will only be kept in the native driver and not be written back to the configuration. |

7.2. Working With Redis Cluster Connection

As mentioned above Redis Cluster behaves different from single node Redis or even a Sentinel monitored master slave environment. This is reasoned by the automatic sharding that maps a key to one of 16384 slots which are distributed across the nodes. Therefore commands that involve more than one key must assert that all keys map to the exact same slot in order to avoid cross slot execution errors. Further on, hence a single cluster node, only serves a dedicated set of keys, commands issued against one particular server only return results for those keys served by the server. As a very simple example take the KEYScommand. When issued to a server in cluster environment it only returns the keys served by the node the request is sent to and not necessarily all keys within the cluster. So to get all keys in cluster environment it is necessary to read the keys from at least all known master nodes.

While redirects for to a specific keys to the corresponding slot serving node are handled by the driver libraries, higher level functions like collecting information across nodes, or sending commands to all nodes in the cluster that are covered by RedisClusterConnection. Picking up the keys example from just before, this means, that the keys(pattern) method picks up every master node in cluster and simultaneously executes the KEYS command on every single one, while picking up the results and returning the cumulated set of keys. To just request the keys of a single node RedisClusterConnection provides overloads for those (like keys(node, pattern) ).

A RedisClusterNode can be obtained from RedisClusterConnection.clusterGetNodes or it can be constructed using either host and port or the node Id.

*Example 2. Sample of Running Commands Across the Cluster*

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("foo", value);

connection.set("bar", value);

connection.keys("\*");

connection.keys(NODE\_7379, "\*");

connection.keys(NODE\_7380, "\*");

connection.keys(NODE\_7381, "\*");

connection.keys(NODE\_7382, "\*");

|  |  |
| --- | --- |
|  | Master node serving slots 0 to 5460 replicated to slave at 7382 |
|  | Master node serving slots 5461 to 10922 |
|  | Master node serving slots 10923 to 16383 |
|  | Slave node holding replicates of master at 7379 |
|  | Request routed to node at 7381 serving slot 12182 |
|  | Request routed to node at 7379 serving slot 5061 |
|  | Request routed to nodes at 7379, 7380, 7381 → [foo, bar] |
|  | Request routed to node at 7379 → [bar] |
|  | Request routed to node at 7380 → [] |
|  | Request routed to node at 7381 → [foo] |
|  | Request routed to node at 7382 → [bar] |

Cross slot requests such as MGET are automatically served by the native driver library when all keys map to the same slot. However once this is not the case RedisClusterConnection executes multiple parallel GET commands against the slot serving nodes and again returns a cumulated result. Obviously this is less performing than the single slot execution and therefore should be used with care. In doubt please consider pinning keys to the same slot by providing a prefix in curly brackets like {my-prefix}.foo and {my-prefix}.bar which will both map to the same slot number.

*Example 3. Sample of Cross Slot Request Handling*

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("foo", value); // slot: 12182

connection.set("{foo}.bar", value); // slot: 12182

connection.set("bar", value); // slot: 5461

connection.mGet("foo", "{foo}.bar");

connection.mGet("foo", "bar");

|  |  |
| --- | --- |
|  | Same Configuration as in the sample before. |
|  | Keys map to same slot → 127.0.0.1:7381 MGET foo {foo}.bar |
|  | Keys map to different slots and get split up into single slot ones routed to the according nodes → 127.0.0.1:7379 GET bar → 127.0.0.1:7381 GET foo |
|  | The above provided simple examples to demonstrate the general strategy followed by Spring Data Redis. Be aware that some operations might require loading huge amounts of data into memory in order to compute the desired command. Additionally not all cross slot requests can safely be ported to multiple single slot requests and will error if misused (eg. PFCOUNT). |

7.3. Working With RedisTemplate and ClusterOperations

Please refer to the section [Working with Objects through RedisTemplate](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis:template) to read about general purpose, configuration and usage of RedisTemplate.

|  |  |
| --- | --- |
|  | Please be careful when setting up RedisTemplate#keySerializer using any of the JSON RedisSerializers as changing json structure has immediate influence on hash slot calculation. |

RedisTemplate provides access to cluster specific operations via the ClusterOperations interface that can be obtained via RedisTemplate.opsForCluster(). This allows to execute commands explicitly on a single node within the cluster while retaining de-/serialization features configured for the template and provides administrative commands such as CLUSTER MEET or more high level operations for eg. resharding.

*Example 4. Accessing RedisClusterConnection via RedisTemplate*

ClusterOperations clusterOps = redisTemplate.opsForCluster();

clusterOps.shutdown(NODE\_7379);

|  |  |
| --- | --- |
|  | Shut down node at 7379 and cross fingers there is a slave in place that can take over. |

8. Redis Repositories

Working with Redis Repositories allows to seamlessly convert and store domain objects in Redis Hashes, apply custom mapping strategies and make use of secondary indexes.

|  |  |
| --- | --- |
|  | Redis Repositories requires at least Redis Server version 2.8.0. |

8.1. Usage

To access domain entities stored in a Redis you can leverage repository support that eases implementing those quite significantly.

*Example 5. Sample Person Entity*

@RedisHash("persons")

public class Person {

@Id String id;

String firstname;

String lastname;

Address address;

}

We have a pretty simple domain object here. Note that it has a property named id annotated with org.springframework.data.annotation.Id and a @RedisHash annotation on its type. Those two are responsible for creating the actual key used to persist the hash.

|  |  |
| --- | --- |
|  | Properties annotated with @Id as well as those named id are considered as the identifier properties. Those with the annotation are favored over others. |

To now actually have a component responsible for storage and retrieval we need to define a repository interface.

*Example 6. Basic Repository Interface To Persist Person Entities*

public interface PersonRepository extends CrudRepository<Person, String> {

}

As our repository extends CrudRepository it provides basic CRUD and finder operations. The thing we need in between to glue things together is the according Spring configuration.

*Example 7. JavaConfig for Redis Repositories*

@Configuration

@EnableRedisRepositories

public class ApplicationConfig {

@Bean

public RedisConnectionFactory connectionFactory() {

return new JedisConnectionFactory();

}

@Bean

public RedisTemplate<?, ?> redisTemplate() {

RedisTemplate<byte[], byte[]> template = new RedisTemplate<byte[], byte[]>();

return template;

}

}

Given the setup above we can go on and inject PersonRepository into our components.

*Example 8. Access to Person Entities*

@Autowired PersonRepository repo;

public void basicCrudOperations() {

Person rand = new Person("rand", "al'thor");

rand.setAddress(new Address("emond's field", "andor"));

repo.save(rand);

repo.findOne(rand.getId());

repo.count();

repo.delete(rand);

}

|  |  |
| --- | --- |
|  | Generates a new id if current value is null or reuses an already set id value and stores properties of type Personinside the Redis Hash with key with pattern keyspace:id in this case eg. persons:5d67b7e1-8640-4475-beeb-c666fab4c0e5. |
|  | Uses the provided id to retrieve the object stored at keyspace:id. |
|  | Counts the total number of entities available within the keyspace *persons* defined by @RedisHash on Person. |
|  | Removes the key for the given object from Redis. |

8.2. Object to Hash Mapping

The Redis Repository support persists Objects in Hashes. This requires an Object to Hash conversion which is done by a RedisConverter. The default implementation uses Converter for mapping property values to and from Redis native byte[].

Given the Person type from the previous sections the default mapping looks like the following:

\_class = org.example.Person

id = e2c7dcee-b8cd-4424-883e-736ce564363e

firstname = rand

lastname = al’thor

address.city = emond's field

address.country = andor

|  |  |
| --- | --- |
|  | The \_class attribute is included on root level as well as on any nested interface or abstract types. |
|  | Simple property values are mapped by path. |
|  | Properties of complex types are mapped by their dot path. |
| *Table 7. Default Mapping Rules* | | |
| **Type** | **Sample** | **Mapped Value** |
| Simple Type (eg. String) | String firstname = "rand"; | firstname = "rand" |
| Complex Type (eg. Address) | Address adress = new Address("emond’s field"); | address.city = "emond’s field" |
| List of Simple Type | List<String> nicknames = asList("dragon reborn", "lews therin"); | nicknames.[0] = "dragon reborn", nicknames.[1] = "lews therin" |
| Map of Simple Type | Map<String, String> atts = asMap({"eye-color", "grey"}, {"…​ | atts.[eye-color] = "grey", atts.[hair-color] = "…​ |
| List of Complex Type | List<Address> addresses = asList(new Address("em…​ | addresses.[0].city = "emond’s field", addresses.[1].city = "…​ |
| Map of Complex Type | Map<String, Address> addresses = asMap({"home", new Address("em…​ | addresses.[home].city = "emond’s field", addresses.[work].city = "…​ |

Mapping behavior can be customized by registering the according Converter in RedisCustomConversions. Those converters can take care of converting from/to a single byte[] as well as Map<String,byte[]> whereas the first one is suitable for eg. converting one complex type to eg. a binary JSON representation that still uses the default mappings hash structure. The second option offers full control over the resulting hash. Writing objects to a Redis hash will delete the content from the hash and re-create the whole hash, so not mapped data will be lost.

*Example 9. Sample byte[] Converters*

@WritingConverter

public class AddressToBytesConverter implements Converter<Address, byte[]> {

private final Jackson2JsonRedisSerializer<Address> serializer;

public AddressToBytesConverter() {

serializer = new Jackson2JsonRedisSerializer<Address>(Address.class);

serializer.setObjectMapper(new ObjectMapper());

}

@Override

public byte[] convert(Address value) {

return serializer.serialize(value);

}

}

@ReadingConverter

public class BytesToAddressConverter implements Converter<byte[], Address> {

private final Jackson2JsonRedisSerializer<Address> serializer;

public BytesToAddressConverter() {

serializer = new Jackson2JsonRedisSerializer<Address>(Address.class);

serializer.setObjectMapper(new ObjectMapper());

}

@Override

public Address convert(byte[] value) {

return serializer.deserialize(value);

}

}

Using the above byte[] Converter produces eg.

\_class = org.example.Person

id = e2c7dcee-b8cd-4424-883e-736ce564363e

firstname = rand

lastname = al’thor

address = { city : "emond's field", country : "andor" }

*Example 10. Sample Map<String,byte[]> Converters*

@WritingConverter

public class AddressToMapConverter implements Converter<Address, Map<String,byte[]>> {

@Override

public Map<String,byte[]> convert(Address source) {

return singletonMap("ciudad", source.getCity().getBytes());

}

}

@ReadingConverter

public class MapToAddressConverter implements Converter<Address, Map<String, byte[]>> {

@Override

public Address convert(Map<String,byte[]> source) {

return new Address(new String(source.get("ciudad")));

}

}

Using the above Map Converter produces eg.

\_class = org.example.Person

id = e2c7dcee-b8cd-4424-883e-736ce564363e

firstname = rand

lastname = al’thor

ciudad = "emond's field"

|  |  |
| --- | --- |
|  | Custom conversions have no effect on index resolution. [Secondary Indexes](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.indexes) will still be created even for custom converted types. |

8.3. Keyspaces

Keyspaces define prefixes used to create the actual *key* for the Redis Hash. By default the prefix is set to getClass().getName(). This default can be altered via @RedisHash on aggregate root level or by setting up a programmatic configuration. However, the annotated keyspace supersedes any other configuration.

*Example 11. Keyspace Setup via @EnableRedisRepositories*

@Configuration

@EnableRedisRepositories(keyspaceConfiguration = MyKeyspaceConfiguration.class)

public class ApplicationConfig {

//... RedisConnectionFactory and RedisTemplate Bean definitions omitted

public static class MyKeyspaceConfiguration extends KeyspaceConfiguration {

@Override

protected Iterable<KeyspaceSettings> initialConfiguration() {

return Collections.singleton(new KeyspaceSettings(Person.class, "persons"));

}

}

}

*Example 12. Programmatic Keyspace setup*

@Configuration

@EnableRedisRepositories

public class ApplicationConfig {

//... RedisConnectionFactory and RedisTemplate Bean definitions omitted

@Bean

public RedisMappingContext keyValueMappingContext() {

return new RedisMappingContext(

new MappingConfiguration(

new MyKeyspaceConfiguration(), new IndexConfiguration()));

}

public static class MyKeyspaceConfiguration extends KeyspaceConfiguration {

@Override

protected Iterable<KeyspaceSettings> initialConfiguration() {

return Collections.singleton(new KeyspaceSettings(Person.class, "persons"));

}

}

}

8.4. Secondary Indexes

[Secondary indexes](http://redis.io/topics/indexes) are used to enable lookup operations based on native Redis structures. Values are written to the according indexes on every save and are removed when objects are deleted or [expire](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.expirations).

8.4.1. Simple Property Index

Given the sample Person entity we can create an index for *firstname* by annotating the property with @Indexed.

*Example 13. Annotation driven indexing*

@RedisHash("persons")

public class Person {

@Id String id;

@Indexed String firstname;

String lastname;

Address address;

}

Indexes are built up for actual property values. Saving two Persons eg. "rand" and "aviendha" results in setting up indexes like below.

SADD persons:firstname:rand e2c7dcee-b8cd-4424-883e-736ce564363e

SADD persons:firstname:aviendha a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56

It is also possible to have indexes on nested elements. Assume Address has a *city* property that is annotated with @Indexed. In that case, once person.address.city is not null, we have Sets for each city.

SADD persons:address.city:tear e2c7dcee-b8cd-4424-883e-736ce564363e

Further more the programmatic setup allows to define indexes on map keys and list properties.

@RedisHash("persons")

public class Person {

// ... other properties omitted

Map<String,String> attributes;

Map<String Person> relatives;

List<Address> addresses;

}

|  |  |
| --- | --- |
|  | SADD persons:attributes.map-key:map-value e2c7dcee-b8cd-4424-883e-736ce564363e |
|  | SADD persons:relatives.map-key.firstname:tam e2c7dcee-b8cd-4424-883e-736ce564363e |
|  | SADD persons:addresses.city:tear e2c7dcee-b8cd-4424-883e-736ce564363e |
|  | Indexes will not be resolved on [References](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.references). |

Same as with *keyspaces* it is possible to configure indexes without the need of annotating the actual domain type.

*Example 14. Index Setup via @EnableRedisRepositories*

@Configuration

@EnableRedisRepositories(indexConfiguration = MyIndexConfiguration.class)

public class ApplicationConfig {

//... RedisConnectionFactory and RedisTemplate Bean definitions omitted

public static class MyIndexConfiguration extends IndexConfiguration {

@Override

protected Iterable<IndexDefinition> initialConfiguration() {

return Collections.singleton(new SimpleIndexDefinition("persons", "firstname"));

}

}

}

*Example 15. Programmatic Index setup*

@Configuration

@EnableRedisRepositories

public class ApplicationConfig {

//... RedisConnectionFactory and RedisTemplate Bean definitions omitted

@Bean

public RedisMappingContext keyValueMappingContext() {

return new RedisMappingContext(

new MappingConfiguration(

new KeyspaceConfiguration(), new MyIndexConfiguration()));

}

public static class MyIndexConfiguration extends IndexConfiguration {

@Override

protected Iterable<IndexDefinition> initialConfiguration() {

return Collections.singleton(new SimpleIndexDefinition("persons", "firstname"));

}

}

}

8.4.2. Geospatial Index

Assume the Address type contains a property location of type Point that holds the geo coordinates of the particular address. By annotating the property with @GeoIndexed those values will be added using Redis GEO commands.

@RedisHash("persons")

public class Person {

Address address;

// ... other properties omitted

}

public class Address {

@GeoIndexed Point location;

// ... other properties omitted

}

public interface PersonRepository extends CrudRepository<Person, String> {

List<Person> findByAddressLocationNear(Point point, Distance distance);

List<Person> findByAddressLocationWithin(Circle circle);

}

Person rand = new Person("rand", "al'thor");

rand.setAddress(new Address(new Point(13.361389D, 38.115556D)));

repository.save(rand);

repository.findByAddressLocationNear(new Point(15D, 37D), new Distance(200));

|  |  |
| --- | --- |
|  | Query method declaration on nested property using Point and Distance. |
|  | Query method declaration on nested property using Circle to search within. |
|  | GEOADD persons:address:location 13.361389 38.115556 e2c7dcee-b8cd-4424-883e-736ce564363e |
|  | GEORADIUS persons:address:location 15.0 37.0 200.0 km |

In the above example the lon/lat values are stored using GEOADD using the objects id as the member’s name. The finder methods allow usage of Circle or Point, Distance combinations for querying those values.

|  |  |
| --- | --- |
|  | It is **not** possible to combine near/within with other criteria. |

8.5. Time To Live

Objects stored in Redis may only be valid for a certain amount of time. This is especially useful for persisting short lived objects in Redis without having to remove them manually when they reached their end of life. The expiration time in seconds can be set via @RedisHash(timeToLive=…​) as well as via KeyspaceSettings (see [Keyspaces](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.keyspaces)).

More flexible expiration times can be set by using the @TimeToLive annotation on either a numeric property or method. However do not apply @TimeToLive on both a method and a property within the same class.

*Example 16. Expirations*

public class TimeToLiveOnProperty {

@Id

private String id;

@TimeToLive

private Long expiration;

}

public class TimeToLiveOnMethod {

@Id

private String id;

@TimeToLive

public long getTimeToLive() {

return new Random().nextLong();

}

}

|  |  |
| --- | --- |
|  | Annotating a property explicitly with @TimeToLive will read back the actual TTL or PTTL value from Redis. -1 indicates that the object has no expire associated. |

The repository implementation ensures subscription to [Redis keyspace notifications](http://redis.io/topics/notifications) via RedisMessageListenerContainer.

When the expiration is set to a positive value the according EXPIRE command is executed. Additionally to persisting the original, a *phantom* copy is persisted in Redis and set to expire 5 minutes after the original one. This is done to enable the Repository support to publish RedisKeyExpiredEvent holding the expired value via Springs ApplicationEventPublisher whenever a key expires even though the original values have already been gone. Expiry events will be received on all connected applications using Spring Data Redis repositories.

By default, the key expiry listener is disabled when initializing the application. The startup mode can be adjusted in @EnableRedisRepositories or RedisKeyValueAdapter to start the listener with the application or upon the first insert of an entity with a TTL. See EnableKeyspaceEvents for possible values.

The RedisKeyExpiredEvent will hold a copy of the actually expired domain object as well as the key.

|  |  |
| --- | --- |
|  | Delaying or disabling the expiry event listener startup impacts RedisKeyExpiredEvent publishing. A disabled event listener will not publish expiry events. A delayed startup can cause loss of events because the delayed listener initialization. |
|  | The keyspace notification message listener will alter notify-keyspace-events settings in Redis if those are not already set. Existing settings will not be overridden, so it is left to the user to set those up correctly when not leaving them empty. Please note that CONFIG is disabled on AWS ElastiCache and enabling the listener leads to an error. |

|  |  |
| --- | --- |
|  | Redis Pub/Sub messages are not persistent. If a key expires while the application is down the expiry event will not be processed which may lead to secondary indexes containing still references to the expired object. |

8.6. Persisting References

Marking properties with @Reference allows storing a simple key reference instead of copying values into the hash itself. On loading from Redis, references are resolved automatically and mapped back into the object.

*Example 17. Sample Property Reference*

\_class = org.example.Person

id = e2c7dcee-b8cd-4424-883e-736ce564363e

firstname = rand

lastname = al’thor

mother = persons:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56

|  |  |
| --- | --- |
|  | Reference stores the whole key (keyspace:id) of the referenced object. |
|  | Referenced Objects are not subject of persisting changes when saving the referencing object. Please make sure to persist changes on referenced objects separately, since only the reference will be stored. Indexes set on properties of referenced types will not be resolved. |

8.7. Persisting Partial Updates

In some cases it is not necessary to load and rewrite the entire entity just to set a new value within it. A session timestamp for last active time might be such a scenario where you just want to alter one property. PartialUpdate allows to define set and delete actions on existing objects while taking care of updating potential expiration times of the entity itself as well as index structures.

*Example 18. Sample Partial Update*

PartialUpdate<Person> update = new PartialUpdate<Person>("e2c7dcee", Person.class)

.set("firstname", "mat")

.set("address.city", "emond's field")

.del("age");

template.update(update);

update = new PartialUpdate<Person>("e2c7dcee", Person.class)

.set("address", new Address("caemlyn", "andor"))

.set("attributes", singletonMap("eye-color", "grey"));

template.update(update);

update = new PartialUpdate<Person>("e2c7dcee", Person.class)

.refreshTtl(true);

.set("expiration", 1000);

template.update(update);

|  |  |
| --- | --- |
|  | Set the simple property *firstname* to *mat*. |
|  | Set the simple property *address.city* to *emond’s field* without having to pass in the entire object. This does not work when a custom conversion is registered. |
|  | Remove the property *age*. |
|  | Set complex property *address*. |
|  | Set a map/collection of values removes the previously existing map/collection and replaces the values with the given ones. |
|  | Automatically update the server expiration time when altering [Time To Live](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.expirations). |
|  | Updating complex objects as well as map/collection structures requires further interaction with Redis to determine existing values which means that it might turn out that rewriting the entire entity might be faster. |

8.8. Queries and Query Methods

Query methods allow automatic derivation of simple finder queries from the method name.

*Example 19. Sample Repository finder Method*

public interface PersonRepository extends CrudRepository<Person, String> {

List<Person> findByFirstname(String firstname);

}

|  |  |
| --- | --- |
|  | Please make sure properties used in finder methods are set up for indexing. |
|  | Query methods for Redis repositories support only queries for entities and collections of entities with paging. |

Using derived query methods might not always be sufficient to model the queries to execute. RedisCallback offers more control over the actual matching of index structures or even custom added ones. All it takes is providing a RedisCallback that returns a single or Iterable set of *id* values.

*Example 20. Sample finder using RedisCallback*

String user = //...

List<RedisSession> sessionsByUser = template.find(new RedisCallback<Set<byte[]>>() {

public Set<byte[]> doInRedis(RedisConnection connection) throws DataAccessException {

return connection

.sMembers("sessions:securityContext.authentication.principal.username:" + user);

}}, RedisSession.class);

Here’s an overview of the keywords supported for Redis and what a method containing that keyword essentially translates to.

| *Table 8. Supported keywords inside method names* | | |
| --- | --- | --- |
| **Keyword** | **Sample** | **Redis snippet** |
| And | findByLastnameAndFirstname | SINTER …:firstname:rand …:lastname:al’thor |
| Or | findByLastnameOrFirstname | SUNION …:firstname:rand …:lastname:al’thor |
| Is,Equals | findByFirstname,findByFirstnameIs,findByFirstnameEquals | SINTER …:firstname:rand |
| Top,First | findFirst10ByFirstname,findTop5ByFirstname |  |

8.9. Redis Repositories running on Cluster

Using the Redis repository support in a clustered Redis environment is fine. Please see the [Redis Cluster](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#cluster) section for ConnectionFactory configuration details. Still some considerations have to be done as the default key distribution will spread entities and secondary indexes through out the whole cluster and its slots.

| **key** | **type** | **slot** | **node** |
| --- | --- | --- | --- |
| persons:e2c7dcee-b8cd-4424-883e-736ce564363e | id for hash | 15171 | 127.0.0.1:7381 |
| persons:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56 | id for hash | 7373 | 127.0.0.1:7380 |
| persons:firstname:rand | index | 1700 | 127.0.0.1:7379 |

Some commands like SINTER and SUNION can only be processed on the Server side when all involved keys map to the same slot. Otherwise computation has to be done on client side. Therefore it be useful to pin keyspaces to a single slot which allows to make use of Redis serverside computation right away.

| **key** | | **type** | **slot** | **node** |
| --- | --- | --- | --- | --- |
| {persons}:e2c7dcee-b8cd-4424-883e-736ce564363e | | id for hash | 2399 | 127.0.0.1:7379 |
| {persons}:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56 | | id for hash | 2399 | 127.0.0.1:7379 |
| {persons}:firstname:rand | | index | 2399 | 127.0.0.1:7379 |
|  | Define and pin keyspaces via `@RedisHash("{yourkeyspace}") to specific slots when using Redis cluster. | | | | |

8.10. CDI integration

Instances of the repository interfaces are usually created by a container, which Spring is the most natural choice when working with Spring Data. There’s sophisticated support to easily set up Spring to create bean instances. Spring Data Redis ships with a custom CDI extension that allows using the repository abstraction in CDI environments. The extension is part of the JAR so all you need to do to activate it is dropping the Spring Data Redis JAR into your classpath.

You can now set up the infrastructure by implementing a CDI Producer for the RedisConnectionFactory and RedisOperations:

class RedisOperationsProducer {

@Produces

RedisConnectionFactory redisConnectionFactory() {

JedisConnectionFactory jedisConnectionFactory = new JedisConnectionFactory(new RedisStandaloneConfiguration());

jedisConnectionFactory.afterPropertiesSet();

return jedisConnectionFactory;

}

void disposeRedisConnectionFactory(@Disposes RedisConnectionFactory redisConnectionFactory) throws Exception {

if (redisConnectionFactory instanceof DisposableBean) {

((DisposableBean) redisConnectionFactory).destroy();

}

}

@Produces

@ApplicationScoped

RedisOperations<byte[], byte[]> redisOperationsProducer(RedisConnectionFactory redisConnectionFactory) {

RedisTemplate<byte[], byte[]> template = new RedisTemplate<byte[], byte[]>();

template.setConnectionFactory(redisConnectionFactory);

template.afterPropertiesSet();

return template;

}

}

The necessary setup can vary depending on the JavaEE environment you run in.

The Spring Data Redis CDI extension will pick up all Repositories available as CDI beans and create a proxy for a Spring Data repository whenever a bean of a repository type is requested by the container. Thus obtaining an instance of a Spring Data repository is a matter of declaring an @Injected property:

class RepositoryClient {

@Inject

PersonRepository repository;

public void businessMethod() {

List<Person> people = repository.findAll();

}

}

A Redis Repository requires RedisKeyValueAdapter and RedisKeyValueTemplate instances. These beans are created and managed by the Spring Data CDI extension if no provided beans are found. You can however supply your own beans to configure the specific properties of RedisKeyValueAdapter and RedisKeyValueTemplate.

Appendixes

Appendix Document structure

Various appendixes outside the reference documentation.

[Schema](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#appendix:schema) defines the schemas provided by Spring Data Redis.

Appendix A: Schema

Core schema

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

<xsd:schema xmlns="http://www.springframework.org/schema/redis"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

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

xmlns:tool="http://www.springframework.org/schema/tool"

targetNamespace="http://www.springframework.org/schema/redis"

elementFormDefault="qualified"

attributeFormDefault="unqualified">

<xsd:import namespace="http://www.springframework.org/schema/tool" schemaLocation="http://www.springframework.org/schema/tool/spring-tool.xsd"/>

<xsd:annotation>

<xsd:documentation><![CDATA[

Defines the configuration elements for the Spring Data Redis support.

Allows for configuring Redis listener containers in XML 'shortcut' style.

]]></xsd:documentation>

</xsd:annotation>

<xsd:element name="listener-container">

<xsd:annotation>

<xsd:documentation><![CDATA[

Container of Redis listeners. All listeners will be hosted by the same container.

]]></xsd:documentation>

<xsd:appinfo>

<tool:annotation>

<tool:exports type="org.springframework.data.redis.listener.RedisMessageListenerContainer"/>

</tool:annotation>

</xsd:appinfo>

</xsd:annotation>

<xsd:complexType>

<xsd:sequence>

<xsd:element name="listener" type="listenerType" minOccurs="0" maxOccurs="unbounded"/>

</xsd:sequence>

<xsd:attribute name="connection-factory" type="xsd:string" default="redisConnectionFactory">

<xsd:annotation>

<xsd:documentation><![CDATA[

A reference to the Redis ConnectionFactory bean.

Default is "redisConnectionFactory".

]]></xsd:documentation>

<xsd:appinfo>

<tool:annotation kind="ref">

<tool:expected-type type="org.springframework.data.redis.connection.ConnectionFactory"/>

</tool:annotation>

</xsd:appinfo>

</xsd:annotation>

</xsd:attribute>

<xsd:attribute name="task-executor" type="xsd:string">

<xsd:annotation>

<xsd:documentation><![CDATA[

A reference to a Spring TaskExecutor (or standard JDK 1.5 Executor) for executing

Redis listener invokers. Default is a SimpleAsyncTaskExecutor.

]]></xsd:documentation>

<xsd:appinfo>

<tool:annotation kind="ref">

<tool:expected-type type="java.util.concurrent.Executor"/>

</tool:annotation>

</xsd:appinfo>

</xsd:annotation>

</xsd:attribute>

<xsd:attribute name="subscription-task-executor" type="xsd:string">

<xsd:annotation>

<xsd:documentation><![CDATA[

A reference to a Spring TaskExecutor (or standard JDK 1.5 Executor) for listening

to Redis messages. By default reuses the 'task-executor' value.

]]></xsd:documentation>

<xsd:appinfo>

<tool:annotation kind="ref">

<tool:expected-type type="java.util.concurrent.Executor"/>

</tool:annotation>

</xsd:appinfo>

</xsd:annotation>

</xsd:attribute>

<xsd:attribute name="topic-serializer" type="xsd:string">

<xsd:annotation>

<xsd:documentation><![CDATA[

A reference to the RedisSerializer strategy for converting Redis channels/patterns to

serialized format. Default is a StringRedisSerializer.

]]></xsd:documentation>

<xsd:appinfo>

<tool:annotation kind="ref">

<tool:expected-type type="org.springframework.data.redis.serializer.RedisSerializer"/>

</tool:annotation>

</xsd:appinfo>

</xsd:annotation>

</xsd:attribute>

<xsd:attribute name="phase" type="xsd:string">

<xsd:annotation>

<xsd:documentation><![CDATA[

The lifecycle phase within which this container should start and stop. The lower

the value the earlier this container will start and the later it will stop. The

default is Integer.MAX\_VALUE meaning the container will start as late as possible

and stop as soon as possible.

]]></xsd:documentation>

</xsd:annotation>

</xsd:attribute>

</xsd:complexType>

</xsd:element>

<xsd:complexType name="listenerType">

<xsd:attribute name="ref" type="xsd:string" use="required">

<xsd:annotation>

<xsd:documentation><![CDATA[

The bean name of the listener object, implementing

the MessageListener interface or defining the specified listener method.

Required.

]]></xsd:documentation>

<xsd:appinfo>

<tool:annotation kind="ref"/>

</xsd:appinfo>

</xsd:annotation>

</xsd:attribute>

<xsd:attribute name="topic" type="xsd:string">

<xsd:annotation>

<xsd:documentation><![CDATA[

The topics(s) to which the listener is subscribed. Can be (in Redis terminology) a

channel or/and a pattern. Multiple values can be specified by separating them with

spaces. Patterns can be specified by using the '\*' character.

]]></xsd:documentation>

</xsd:annotation>

</xsd:attribute>

<xsd:attribute name="method" type="xsd:string">

<xsd:annotation>

<xsd:documentation><![CDATA[

The name of the listener method to invoke. If not specified,

the target bean is supposed to implement the MessageListener

interface or provide a method named 'handleMessage'.

]]></xsd:documentation>

</xsd:annotation>

</xsd:attribute>

<xsd:attribute name="serializer" type="xsd:string">

<xsd:annotation>

<xsd:documentation><![CDATA[

A reference to the RedisSerializer strategy for converting Redis Messages to

listener method arguments. Default is a StringRedisSerializer.

]]></xsd:documentation>

<xsd:appinfo>

<tool:annotation kind="ref">

<tool:expected-type type="org.springframework.data.redis.serializer.RedisSerializer"/>

</tool:annotation>

</xsd:appinfo>

</xsd:annotation>

</xsd:attribute>

</xsd:complexType>

<xsd:element name="collection">

<xsd:annotation>

<xsd:documentation><![CDATA[

Factory creating collections on top of Redis keys.

]]></xsd:documentation>

<xsd:appinfo>

<tool:annotation>

<tool:exports type="org.springframework.data.redis.support.collections.RedisCollectionFactoryBean"/>

</tool:annotation>

</xsd:appinfo>

</xsd:annotation>

<xsd:complexType>

<xsd:attribute name="id" type="xsd:ID">

<xsd:annotation>

<xsd:documentation><![CDATA[

The name of the Redis collection.]]></xsd:documentation>

</xsd:annotation>

</xsd:attribute>

<xsd:attribute name="key" type="xsd:string" use="optional">

<xsd:annotation>

<xsd:documentation><![CDATA[

Redis key of the created collection. Defaults to bean id.

]]></xsd:documentation>

</xsd:annotation>

</xsd:attribute>

<xsd:attribute name="template" type="xsd:string" default="redisTemplate">

<xsd:annotation>

<xsd:documentation><![CDATA[

A reference to a RedisTemplate bean.Default is "redisTemplate".

]]></xsd:documentation>

<xsd:appinfo>

<tool:annotation kind="ref">

<tool:expected-type type="org.springframework.data.redis.core.RedisTemplate"/>

</tool:annotation>

</xsd:appinfo>

</xsd:annotation>

</xsd:attribute>

<xsd:attribute name="type" default="LIST" use="optional">

<xsd:annotation>

<xsd:documentation><![CDATA[

The collection type (default is list).

If the key exists, its type takes priority. The type is used to disambiguate the collection type (map vs properties) or

specify one in case the key is missing.]]></xsd:documentation>

</xsd:annotation>

<xsd:simpleType>

<xsd:restriction base="xsd:string">

<xsd:enumeration value="LIST"/>

<xsd:enumeration value="SET"/>

<xsd:enumeration value="ZSET"/>

<xsd:enumeration value="MAP"/>

<xsd:enumeration value="PROPERTIES"/>

</xsd:restriction>

</xsd:simpleType>

</xsd:attribute>

</xsd:complexType>

</xsd:element>

</xsd:schema>

Appendix B: Command Reference

Supported commands

| *Table 9. Redis commands supported by RedisTemplate.* | |
| --- | --- |
| **Command** | **Template Support** |
| APPEND | X |
| AUTH | X |
| BGREWRITEAOF | X |
| BGSAVE | X |
| BITCOUNT | X |
| BITOP | X |
| BLPOP | X |
| BRPOP | X |
| BRPOPLPUSH | X |
| CLIENT KILL | X |
| CLIENT GETNAME | X |
| CLIENT LIST | X |
| CLIENT SETNAME | X |
| CLUSTER SLOTS | - |
| COMMAND | - |
| COMMAND COUNT | - |
| COMMAND GETKEYS | - |
| COMMAND INFO | - |
| CONFIG GET | X |
| CONFIG RESETSTAT | X |
| CONFIG REWRITE | - |
| CONFIG SET | X |
| DBSIZE | X |
| DEBUG OBJECT | - |
| DEBUG SEGFAULT | - |
| DECR | X |
| DECRBY | X |
| DEL | X |
| DISCARD | X |
| DUMP | X |
| ECHO | X |
| EVAL | X |
| EVALSHA | X |
| EXEC | X |
| EXISTS | X |
| EXPIRE | X |
| EXPIREAT | X |
| FLUSHALL | X |
| FLUSHDB | X |
| GET | X |
| GETBIT | X |
| GETRANGE | X |
| GETSET | X |
| HDEL | X |
| HEXISTS | X |
| HGET | X |
| HGETALL | X |
| HINCRBY | X |
| HINCRBYFLOAT | X |
| HKEYS | X |
| HLEN | X |
| HMGET | X |
| HMSET | X |
| HSCAN | X |
| HSET | X |
| HSETNX | X |
| HVALS | X |
| INCR | X |
| INCRBY | X |
| INCRBYFLOAT | X |
| INFO | X |
| KEYS | X |
| LASTSAVE | X |
| LINDEX | X |
| LINSERT | X |
| LLEN | X |
| LPOP | X |
| LPUSH | X |
| LPUSHX | X |
| LRANGE | X |
| LREM | X |
| LSET | X |
| LTRIM | X |
| MGET | X |
| MIGRATE | - |
| MONITOR | - |
| MOVE | X |
| MSET | X |
| MSETNX | X |
| MULTI | X |
| OBJECT | - |
| PERSIST | X |
| PEXIPRE | X |
| PEXPIREAT | X |
| PFADD | X |
| PFCOUNT | X |
| PFMERGE | X |
| PING | X |
| PSETEX | X |
| PSUBSCRIBE | X |
| PTTL | X |
| PUBLISH | X |
| PUBSUB | - |
| PUBSUBSCRIBE | - |
| QUIT | X |
| RANDOMKEY | X |
| RENAME | X |
| RENAMENX | X |
| RESTORE | X |
| ROLE | - |
| RPOP | X |
| RPOPLPUSH | X |
| RPUSH | X |
| RPUSHX | X |
| SADD | X |
| SAVE | X |
| SCAN | X |
| SCARD | X |
| SCRIPT EXITS | X |
| SCRIPT FLUSH | X |
| SCRIPT KILL | X |
| SCRIPT LOAD | X |
| SDIFF | X |
| SDIFFSTORE | X |
| SELECT | X |
| SENTINEL FAILOVER | X |
| SENTINEL GET-MASTER-ADD-BY-NAME | - |
| SENTINEL MASTER | - |
| SENTINEL MASTERS | X |
| SENTINEL MONITOR | X |
| SENTINEL REMOVE | X |
| SENTINEL RESET | - |
| SENTINEL SET | - |
| SENTINEL SLAVES | X |
| SET | X |
| SETBIT | X |
| SETEX | X |
| SETNX | X |
| SETRANGE | X |
| SHUTDOWN | X |
| SINTER | X |
| SINTERSTORE | X |
| SISMEMBER | X |
| SLAVEOF | X |
| SLOWLOG | - |
| SMEMBERS | X |
| SMOVE | X |
| SORT | X |
| SPOP | X |
| SRANDMEMBER | X |
| SREM | X |
| SSCAN | X |
| STRLEN | X |
| SUBSCRIBE | X |
| SUNION | X |
| SUNIONSTORE | X |
| SYNC | - |
| TIME | X |
| TTL | X |
| TYPE | X |
| UNSUBSCRIBE | X |
| UNWATCH | X |
| WATCH | X |
| ZADD | X |
| ZCARD | X |
| ZCOUNT | X |
| ZINCRBY | X |
| ZINTERSTORE | X |
| ZLEXCOUNT | - |
| ZRANGE | X |
| ZRANGEBYLEX | - |
| ZREVRANGEBYLEX | - |
| ZRANGEBYSCORE | X |
| ZRANK | X |
| ZREM | X |
| ZREMRANGEBYLEX | - |
| ZREMRANGEBYRANK | X |
| ZREVRANGE | X |
| ZREVRANGEBYSCORE | X |
| ZREVRANK | X |
| ZSCAN | X |
| ZSCORE | X |
| ZUNINONSTORE | X |

Version 2.0.3.RELEASE  
Last updated 2018-01-24 13:43:51 MEZ