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.

**声明：此文档为个人翻译官方文档，未经允许请勿外传。**

**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为Redis提供了专门的消息集成，在功能和命名方面与Spring Framework中的JMS集成非常相似; 实际上，熟悉Spring的JMS支持的用户应该感到宾至如归。

Redis消息可以大致分为两个功能区域，即生产或发布和消费或订阅消息，因此是快捷方式pubsub（发布/订阅）。 RedisTemplate类用于消息生成。 对于类似于Java EE的消息驱动Bean风格的异步接收，Spring Data提供了一个专用的消息侦听器容器，用于创建消息驱动的POJO（MDP）和同步接收RedisConnection协定。

package org.springframework.data.redis.connection 和 org.springframework.data.redis.listener 提供了使用Redis消息传递的核心功能。

5.9.1. Sending/Publishing messages

要发布消息，与其他操作一样，可以使用低级RedisConnection或高级RedisTemplate。 两个实体都提供发布方法，该方法接受需要发送的消息以及目标频道(destination channel)作为参数。 虽然RedisConnection需要原始数据（array of bytes），但RedisTemplate允许任意对象作为消息传入：

// 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

在接收方，可以通过直接命名或使用模式匹配来订阅一个或多个频道(channels)。 后一种方法非常有用，因为它不仅允许使用一个命令创建多个订阅，而且还可以在订阅时尚未创建的频道上进行侦听（只要它们与模式匹配）。

在底层，RedisConnection提供了subscribe和pSubscribe方法，它们分别按照模式映射Redis命令用于按频道订阅。 请注意，多个通道或模式可以用作参数。 要更改连接的订阅或仅查询是否正在侦听，RedisConnection提供了getSubscription和isSubscribe方法。

|  |  |
| --- | --- |
|  | Spring Data Redis中的订阅命令是阻塞的。 也就是说，在一个连接上调用订阅将导致当前线程阻塞，因为它将开始等待消息 - 只有当订阅被取消时，线程才会被释放，这是一个额外的线程在相同的连接上调用取消订阅或取消订阅。 有关此问题的解决方案，请参阅下面的消息监听器容器( [message listener container](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis:pubsub:subscribe:containers))。 |

如上所述，一旦订阅连接开始等待消息。 除了添加新的订阅或修改/取消现有的订阅之外，不能调用其他命令。 也就是说，调用任何其他方法，然后订阅，pSubscribe，unsubscribe或pUnsubscribe都是非法的，并且会抛出异常。

为了订阅消息，需要实现MessageListener回调：每当新消息到达时，回调被调用，用户代码通过onMessage 方法执行。 该interface不仅可以访问实际的消息，还可以访问已经收到的通道以及订阅所使用的与通道匹配的模式（如果有的话）。 该信息允许被叫方区分各种消息，不仅仅是内容，而且还通过数据。

Message Listener Containers

由于其阻塞性质，低级订阅没有吸引力，因为它需要为每个单独的监听者进行连接和线程管理。 为了缓解这个问题，Spring Data提供了RedisMessageListenerContainer ，它代表用户(user - users)完成所有繁重的工作 - 熟悉EJB和JMS的用户应该找到熟悉的概念，因为它尽可能地接近Spring框架中的支持， POJO（MDPs）

RedisMessageListenerContainer充当消息侦听器容器; 它用于接收来自Redis通道的消息，并驱动注入到其中的MessageListener。 侦听器容器负责所有的消息接收线程，并派发到侦听器进行处理。 消息监听器容器是MDP和消息提供者之间的中介，负责注册接收消息，资源获取和释放，异常转换等。 这使您可以作为应用程序开发人员编写与接收消息（并对其作出响应）相关的（可能是复杂的）业务逻辑，并将模板化的Redis基础架构问题委托给框架。

此外，为了最大限度地减少应用程序的占用空间，RedisMessageListenerContainer允许一个连接和一个线程被多个监听器共享，即使它们不共享订阅。 因此，无论应用程序跟踪多少个监听器或通道，运行时成本在其整个生命周期中都将保持不变。 此外，容器允许运行时配置更改，以便在应用程序运行时添加或删除侦听器，而无需重新启动。 此外，容器使用延迟订阅方式，仅在需要时才使用RedisConnection - 如果所有侦听器都已取消订阅，则会自动执行清除操作，并释放所用的线程。

为了帮助消息的异步方式，容器需要一个java.util.concurrent.Executor（或者Spring的TaskExecutor）来调度消息。 根据负载，监听器数量或运行时环境，应该更改或调整执行程序以更好地满足其需求 - 特别是在托管环境（如应用程序服务器）中，强烈建议选择合适的TaskExecutor 它的运行时间的优势。

The MessageListenerAdapter

MessageListenerAdapter类是Spring异步消息传递支持中的最后一个组件：简而言之，它允许您将几乎任何类作为MDP公开（当然还有一些限制）。

考虑下面的接口定义。 请注意，虽然接口不扩展MessageListener接口，但仍可以通过使用MessageListenerAdapter类将其用作MDP。 还要注意各种消息处理方法是如何根据它们可以接收和处理的各种消息类型的内容进行强类型化的。 另外，消息发送到的通道或模式可以作为String类型的第二个参数传递给方法：

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...

}

特别的，请注意MessageDelegate接口（上面的DefaultMessageDelegate类）的上述实现如何根本没有Redis依赖关系。 这是一个POJO，我们将通过以下配置将其制作成MDP。

<?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>

|  |  |
| --- | --- |
|  | listener topic 可以是一个 channel  (e.g. topic="chatroom") 或一个 pattern (e.g. topic="\*room") |

上面的示例使用Redis名称空间来声明消息侦听器容器，并自动将POJO注册为侦听器。 完整的bean定义如下所示：

<bean id="messageListener" class="org.springframework.data.redis.listener.adapter.MessageListenerAdapter">

<constructor-arg>

<bean class="redisexample.DefaultMessageDelegate"/>

</constructor-arg>

</bean>

<bean id="redisContainer" class="org.springframework.data.redis.listener.RedisMessageListenerContainer">

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

<property name="messageListeners">

<map>

<entry key-ref="messageListener">

<bean class="org.springframework.data.redis.listener.ChannelTopic">

<constructor-arg value="chatroom">

</bean>

</entry>

</map>

</property>

</bean>

每次收到消息时，适配器都会自动执行低级格式与所需对象类型之间的透明转换（使用配置的RedisSerializer）。 由方法调用引起的任何异常都被容器捕获并处理（默认情况下，被记录）。

5.10. Redis Transactions(事务)

Redis通过multi，exec和discard命令提供对 [transactions](http://redis.io/topics/transactions)的支持。 这些操作在RedisTemplate上可用，但RedisTemplate不能保证使用相同的连接执行事务中的所有操作。

Spring Data Redis提供SessionCallback接口，以便在需要使用相同连接执行多个操作时使用，就像使用Redis事务一样。 例如：

//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));

在返回之前，RedisTemplate将使用其value，hash key和hash value序列化器来反序列化exec的所有结果。 还有一个额外的exec方法，允许您传递一个自定义序列化程序的事务结果。

|  |  |
| --- | --- |
|  | 版本1.1中对RedisConnection和RedisTemplate的exec方法做了一个重要的改变。 以前，这些方法直接从连接器返回事务结果。 这意味着数据类型往往不同于RedisConnection方法返回的数据类型。 例如，zAdd返回一个布尔值，指示元素已添加到已排序的集合中。 大多数连接器将这个值作为一个long值返回，并且Spring Data Redis执行转换。 另一个常见的区别是，大多数连接器返回状态回复（通常是字符串“OK”）像set操作。 这些回复通常被Spring Data Redis丢弃。 在1.1之前，这些转换不是在exec的结果上执行的。 另外，结果在RedisTemplate中没有反序列化，所以它们通常包含原始字节数组。 如果此更改中断了您的应用程序，则可以在RedisConnectionFactory上将convertPipelineAndTxResults设置为false以禁用此行为。 |

5.10.1. @Transactional Support

事务支持默认是禁用的，必须通过设置setEnableTransactionSupport(true)为每个正在使用的RedisTemplate显式启用。 这将强制绑定正在使用的RedisConnection到当前的线程触发MULTI。 如果事务完成没有错误，则调用EXEC，否则DISCARD。 一旦进入MULTI，RedisConnection会对写入操作进行排队，所有只读操作（例如KEYS）都被传送到新的（非线程绑定的）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(流水线)

Redis提供对 [pipelining](http://redis.io/topics/pipelining)的支持，这包括向服务器发送多个命令，而不用等待回复，然后单步读取回复。 当您需要连续发送多个命令时，管道传输可以提高性能，例如将多个元素添加到同一个List中。

Spring Data Redis提供了多个用于在管道中执行命令的RedisTemplate方法。 如果您不关心pipelined操作的结果，则可以使用标准的execute方法，对pipeline参数传递true。 executePipelined方法将在管道中执行提供的RedisCallback或SessionCallback并返回结果。 例如：

//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;

}

});

上面的例子执行pipeline队列中项目的批量权限弹出。 结果列表包含所有弹出的项目。 RedisTemplate在返回之前使用它的值，hash key 和 hash value序列化器来反序列化所有的结果，所以上面例子中返回的项目就是Strings。 还有其他的executePipelined方法可以让你传递一个自定义的序列化器来进行pipelined结果。

请注意，从RedisCallback返回的值必须为空，因为此值将被丢弃，以便返回pipelined命令的结果。

|  |  |
| --- | --- |
|  | 版本1.1中对RedisConnection的closePipeline方法做了一个重要的改变。 以前，此方法直接从连接器返回流水线操作的结果。 这意味着数据类型通常不同于RedisConnection方法返回的数据类型。 例如，zAdd返回一个布尔值，指示元素已添加到已排序的集合中。 大多数连接器将这个值作为一个long值返回，并且Spring Data Redis执行转换。 另一个常见的区别是，大多数连接器返回状态回复（通常是字符串“OK”）像集操作。 这些回复通常被Spring Data Redis丢弃。 在1.1之前，不会对closePipeline的结果执行这些转换。 如果此更改中断了您的应用程序，则可以在RedisConnectionFactory上将convertPipelineAndTxResults设置为false以禁用此行为。 |

5.12. Redis Scripting

Redis 2.6及更高版本通过 [eval](http://redis.io/commands/eval) 和 [evalsha](http://redis.io/commands/evalsha) 命令为Lua脚本的执行提供支持。 Spring Data Redis为处理序列化的脚本执行提供高级抽象，并自动使用Redis脚本缓存。

脚本可以通过RedisTemplate和ReactiveRedisTemplate的执行方法运行。 两者都使用可配置的ScriptExecutor / ReactiveScriptExecutor运行提供的脚本。 默认情况下，ScriptExecutor负责序列化提供的键和参数(keys and arguments)，并反序列化脚本结果。 这是通过模板的键和值序列化器完成的。 还有一个额外的重载允许你传递脚本参数和结果的自定义序列化器。

默认的ScriptExecutor通过检索脚本的SHA1并尝试首先运行evalsha来优化性能，如果脚本还没有出现在Redis脚本缓存中，则会回退到eval。

下面是一个使用Lua脚本执行常见的“check-and-set”场景的示例。 对于Redis脚本来说，这是一个理想的用例，因为它要求我们自动执行一组命令，而一个命令的行为受到另一个命令的影响。

@Bean

public RedisScript<Boolean> script() {

ScriptSource scriptSource = new ResourceScriptSource(new ClassPathResource("META-INF/scripts/checkandset.lua");

return RedisScript.of(scriptSource, Boolean.class);

}

public class Example {

@Autowired

RedisScript<Boolean> script;

public boolean checkAndSet(String expectedValue, String newValue) {

return redisTemplate.execute(script, singletonList("key"), asList(expectedValue, newValue));

}

}

-- checkandset.lua local

current = redis.call('GET', KEYS[1])

if current == ARGV[1]

then redis.call('SET', KEYS[1], ARGV[2])

return true

end

return false

上面的代码配置一个RedisScript指向一个名为checkandset.lua的文件，这个文件预计会返回一个布尔值。 脚本resultType应该是Long，Boolean，List或反序列化的值类型之一。 如果脚本返回丢弃状态（即“OK”），则也可以为null。 在应用程序上下文中配置DefaultRedisScript的单个实例是非常理想的，以避免在每个脚本执行过程中重新计算脚本的SHA1。

上面的checkAndSet方法然后执行脚本可以作为事务或管道的一部分在SessionCallback中执行。 有关更多信息，请参阅 [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) 。

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

5.13. Support Classes

Package org.springframework.data.redis.support提供了各种可重用组件，这些组件依赖于Redis作为后备存储。 目前，该软件包在Redis之上包含各种基于JDK的接口实现，如 [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)。

atomic counters可以轻松地包装Redis密钥增量(key incrementation)，而集合允许以最小的存储空间或API leakage轻松管理Redis keys：特别是RedisSet和RedisZSet接口可以轻松访问Redis支持的seuperations，例如intersection 和 union RedisList在Redis之上实现List，Queue和Dequecontracts（及其等价的阻塞同胞），将存储作为FIFO（*First-In-First-Out*先进先出），LIFO（*Last-In-First-Out*后进先出） 最小配置：

<?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="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);

}

}

如上例所示，消费代码与实际的存储实现是分离的 - 事实上，没有任何迹象表明在下面使用了Redis。 这使得从开发到生产环境变得透明并且极大地提高了可测试性（Redis的实现可以用内存中(in-memory)的代替）。

5.13.1. Support for Spring Cache Abstraction

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

Spring Redis通过org.springframework.data.redis.cachepackage提供了 [cache abstraction](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/integration.html#cache)抽象的实现。 要使用Redis作为后台实现，只需将RedisCacheManager添加到您的配置：

@Bean

public RedisCacheManager cacheManager(RedisConnectionFactory connectionFactory) {

return RedisCacheManager.create(connectionFactory);

}

RedisCacheManager 行为可以通过 RedisCacheManagerBuilder 配置，允许设置默认的 RedisCacheConfiguration,事务行为和预定义的缓存。

RedisCacheManager cm = RedisCacheManager.builder(connectionFactory)

.cacheDefaults(defaultCacheConfig())

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

.transactionAware()

.build();

通过RedisCacheManager创建的RedisCache行为通过RedisCacheConfiguration定义。 该配置允许设置密钥到期时间(key expiration times)，前缀(prefixes)和 RedisSerializer以转换为二进制存储格式和从二进制存储格式转换。 如上所示，RedisCacheManager允许定义每个缓存库上的配置。

RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig()

.entryTtl(Duration.ofSeconds(1))

.disableCachingNullValues();

RedisCacheManager默认使用无锁RedisCacheWriter来读写二进制值。 无锁(Lock-free)缓存提高了吞吐量。 缺少入口锁定可能会导致putIfAbsent和clean方法的重叠，非原子命令，因为这些方法需要多个发送到Redis的命令。 锁定副本通过设置显式锁定键并检查是否存在此键来防止命令重叠，这导致了额外的请求和潜在的命令等待时间。

可以选择锁定行为如下：

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

本节介绍Redis支持以及如何入门。 你会发现与命令式Redis支持有一定的重叠( [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需要Redis 2.6或更高版本以及Java SE 8.0或更高版本。 就语言绑定（或连接器）而言，Spring Data Redis目前与 [Lettuce](https://github.com/lettuce-io/lettuce-core)集成为唯一的反应式Java连接器。  [Project Reactor](https://projectreactor.io/)被用作反应性组分库。

6.2. 使用reactive driver 连接到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（或绑定）。 无论选择哪个库，只需要使用一组Spring Data Redis API，即可在所有连接器（即org.springframework.data.redis.connection包及其ReactiveRedisConnection和ReactiveRedisConnectionFactory接口）中保持一致的行为， 并检索到Redis的活动连接。

6.2.1. Redis Operation Modes

Redis可以作为独立的服务器， [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.模式运行。  [Lettuce](https://github.com/lettuce-io/lettuce-core) 支持上面提到的所有连接类型。

6.2.2. ReactiveRedisConnection 和 ReactiveRedisConnectionFactory

ReactiveRedisConnection为Redis通信提供构建块，因为它处理与Redis后端的通信。 它还自动将底层驱动程序异常转换为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) )，因此可以在不更改任何代码的情况下切换连接器，因为操作语义保持不变。

Active ReactiveRedisConnections通过ReactiveRedisConnectionFactory创建。 另外，工厂作为PersistenceExceptionTranslators，意味着一旦声明，它允许一个做透明的异常转换。 例如，通过使用@Repository注释和AOP进行异常转换。 有关更多信息，请参阅Spring Framework文档( [section](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/data-access.html#orm-exception-translation))中的专用章节。

|  |  |
| --- | --- |
|  | 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ReactiveRedisConnectionFactory最简单的方法是通过IoC容器配置适当的连接器，并将其注入到using类中。

6.2.3. Configuring Lettuce connector

Spring Data Redis通过org.springframework.data.redis.connection.lettuce包支持[Lettuce](https://github.com/lettuce-io/lettuce-core) 。

为Lettuce设置ReactiveRedisConnectionFactory可以做如下：

@Bean

public ReactiveRedisConnectionFactory connectionFactory() {

return new LettuceConnectionFactory("localhost", 6379);

}

使用LettuceClientConfigurationBuilder的更复杂的配置（包括SSL和超时）可能如下所示：

@Bean

public ReactiveRedisConnectionFactory lettuceConnectionFactory() {

LettuceClientConfiguration clientConfig = LettuceClientConfiguration.builder()

.useSsl().and()

.commandTimeout(Duration.ofSeconds(2))

.shutdownTimeout(Duration.ZERO)

.build();

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

}

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

6.3. 通过ReactiveRedisTemplate处理对象

大多数用户可能使用ReactiveRedisTemplate及其相应的包org.springframework.data.redis.core - 由于其丰富的功能集，模板实际上是Redis模块的中心类。 该模板提供了Redis交互的高级抽象。

虽然ReactiveRedisConnection提供接受和返回二进制值（ByteBuffer）的低级方法，但是模板负责序列化和连接管理，使用户无需处理这些细节。

此外，该模板提供了操作视图（来自Redis命令参考的分组之后 [reference](http://redis.io/commands)），该视图提供丰富的，通用的接口，用于针对某种类型进行处理，如下所述：

| *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 |

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，ReactiveRedisTemplate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将通过RedisElementWriter和RedisElementReader进行序列化/反序列化。 序列化上下文在构建时传递给模板，Redis模块在org.springframework.data.redis.serializer包中提供了几个可用实现 - 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。

@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

通过reactive基础架构执行Redis脚本可以使用ReactiveScriptExecutor通过ReactiveRedisTemplate最好地访问。

public class Example {

@Autowired

private ReactiveRedisTemplate<String, String> template;

public Flux<Long> theAnswerToLife() {

DefaultRedisScript<Long> script = new DefaultRedisScript<>();

script.setLocation(new ClassPathResource("META-INF/scripts/42.lua"));

script.setResultType(Long.class);

return reactiveTemplate.execute(script);

}

}

有关脚本命令的更多详细信息，请参阅脚本部分( [scripting section](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#scripting))。

7. Redis Cluster

使用 [Redis Cluster](http://redis.io/topics/cluster-spec)需要Redis服务器版本3.0+，并提供了一套自己的特性和功能。 有关更多信息，请参阅 [Cluster Tutorial](http://redis.io/topics/cluster-tutorial)。

7.1. Enabling Redis Cluster

群集支持基于与非群集通信相同的构件块。 RedisClusterConnection是RedisConnection的扩展，用于处理与Redis集群的通信，并将错误转换为Spring DAO异常层次结构。 RedisClusterConnection是通过RedisConnectionFactory创建的，必须使用相应的RedisClusterConfiguration进行设置。

*Example 1. Redis Cluster 的示例RedisConnectionFactory 配置*

@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 可以通过PropertySource来定义.  *Configuration Properties*   * spring.redis.cluster.nodes: host:port 对. * spring.redis.cluster.max-redirects: 允许的集群重定向的数量 |
|  | 初始配置将驱动程序库指向最初的一组集群节点。 活动群集重新配置所产生的更改只会保留在本机驱动程序中，而不会写回到配置中。 |

7.2. Working With Redis Cluster Connection

如上所述，Redis集群的行为与单个节点Redis或甚至Sentinel监控的主从属环境有所不同。 这是由自动分片的原因，该分片将key映射到分布在节点上的16384个插槽(slots)之一。 因此，涉及多于一个key的命令必须声明所有key映射到完全相同的时隙(slot)以避免交叉时隙执行错误(cross slot execution errors)。 进一步说，因此一个集群节点只提供专用的一组key，针对一个特定的服务器发出的命令只返回服务器提供的那些key的结果。 作为一个非常简单的例子，使用KEYS命令。 当发布到集群环境中的服务器时，它只返回请求发送到的节点所服务的key，而不一定是集群内的所有key。 因此，要获得群集环境中的所有key，至少需要从所有已知主节点读取key。

虽然重定向到对应的槽(slot)服务节点的特定key由驱动程序库处理，但高级功能（如收集跨节点的信息）或将命令发送到群集中由RedisClusterConnection覆盖的所有节点。 从上面的例子中，我们可以看到keys（pattern）方法cluster中的每个主节点，并同时对每个主节点执行KEYS命令，同时拿起结果并返回累积的一组键。 为了只请求单个节点的密钥，RedisClusterConnection为那些（比如keys(node, pattern) )提供了重载。

RedisClusterNode可以从RedisClusterConnection.clusterGetNodes获取，也可以使用主机和端口或节点ID构建。

*Example 2. 跨集群运行命令的示例*

redis-cli@127.0.0.1:7379 > cluster nodes

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

7bb78c... 127.0.0.1:7380 master - 0 1449730618304 2 connected 5461-10922

164888... 127.0.0.1:7381 master - 0 1449730618304 3 connected 10923-16383

b8b5ee... 127.0.0.1:7382 slave 6b38bb... 0 1449730618304 25 connected

RedisClusterConnection connection = connectionFactory.getClusterConnnection();

connection.set("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] |

所有key映射到同一个插槽(slot)时，本地驱动程序库会自动为跨插槽(slot)请求（如MGET）提供服务。 但是，一旦情况不是这样，RedisClusterConnection会针对插槽服务节点执行多个并行GET命令，并再次返回累计结果。 显然这比单个时隙执行的性能差，因此应该小心使用。 如有疑问，请考虑在同一个插槽上提供一个前缀，如{my-prefix}.foo 和 {my-prefix}.bar，它们都映射到相同的插槽编号(slot number)。

*Example 3. Sample of Cross(跨域) 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");

|  |  |
| --- | --- |
|  | 与之前的示例中的配置相同。 |
|  | 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 |
|  | 以上提供了简单的例子来演示Spring Data Redis所遵循的一般策略。 请注意，某些操作可能需要将大量的数据加载到内存中才能计算所需的命令。 此外，并非所有的交叉插槽请求都可以安全地移植到多个单插槽请求，并且如果使用不当（例如，PFCOUNT），则会出错。 |

7.3. 使用 RedisTemplate 和 ClusterOperations

请参阅( [Working with Objects through RedisTemplate](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis:template))通过RedisTemplate使用对象一节来了解RedisTemplate的一般用途，配置和用法。

|  |  |
| --- | --- |
|  | 使用任何JSON RedisSerializer设置RedisTemplate＃keySerializer时请小心，因为更改json结构会对散列槽(hash slot)计算产生直接影响。 |

RedisTemplate通过可通过Template.opsForCluster()获取的ClusterOperations接口提供对集群特定操作的访问。 这允许在集群内的单个节点上显式地执行命令，同时保留为模板配置的de-/serialization功能，并提供管理命令，例如CLUSTER MEET或更高级别的操作。例如: resharding。

*Example 4. 通过RedisTemplate访问 RedisClusterConnection*

ClusterOperations clusterOps = redisTemplate.opsForCluster();

clusterOps.shutdown(NODE\_7379);

|  |  |
| --- | --- |
|  | 在7379关闭节点，cross fingers有一个可以接管的slave。 |

8. Redis Repositories(储存库)

使用Redis存储库可以在Redis哈希中无缝转换和存储域对象，应用自定义映射策略并使用二级索引。

|  |  |
| --- | --- |
|  | Redis存储库至少需要Redis服务器版本2.8.0。 |

8.1. Usage

要访问存储在Redis中的域实体，您可以利用存储库支持，从而轻松实现这些实现。

*Example 5. Sample Person Entity*

@RedisHash("persons")

public class Person {

@Id String id;

String firstname;

String lastname;

Address address;

}

我们在这里有一个非常简单的域对象。 请注意，它有一个名为id的属性，用org.springframework.data.annotation.Id注解，并在其类型上注释@RedisHash。 这两个负责创建用于保存散列的实际key。

|  |  |
| --- | --- |
|  | 用@Id注解的属性以及那些名为id的属性被视为标识符属性。 那些带有注释的人比其他人更喜欢。 |

现在实际上有一个负责存储和检索(storage and retrieval)的组件，我们需要定义一个存储库接口。

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

public interface PersonRepository extends CrudRepository<Person, String> {

}

由于我们的知识库扩展了CrudRepository ，它提供了基本的CRUD和查找(finder)器操作。 我们之间需要粘合在一起的东西是相应的Spring配置。

*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;

}

}

鉴于上面的设置，我们可以继续并将PersonRepository注入到组件中。

*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);

}

|  |  |
| --- | --- |
|  | 如果当前值为null，则生成一个新的ID或者重用已经设置的id值，并存储Personinside Redis Hash类型的属性，此时带有模式keyspace:id的键。persons:5d67b7e1-8640-4475-beeb-c666fab4c0e5。 |
|  | 使用提供的ID检索存储在keyspace:id处的对象。 |
|  | 计算@RedisHash在Person上定义的keyspace人员中可用实体的总数。 |
|  | 从Redis中移除给定对象的键。 |

8.2. Object to Hash Mapping

Redis Repository支持持久化Hashes中的对象。 这需要由RedisConverter完成的对象哈希转换。 默认实现使用Converter将属性值映射到和来自Redis本地byte []。

从前面的部分给出Person类型，默认的映射如下所示：

\_class = org.example.Person

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

firstname = rand

lastname = al’thor

address.city = emond's field

address.country = andor

|  |  |
| --- | --- |
|  | \_class属性包含在根级别(root level)以及任何嵌套的接口或抽象类型中。 |
|  | 简单的属性值由路径映射。 |
|  | 复杂类型的属性由它们的点路径(dot path)映射。 |
| *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 = "…​ |

映射行为可以通过在RedisCustomConversions中注册相应的Converter来定制。 这些转换器可以处理单个byte[]以及Map <String，byte []>的转换，而第一个适用于例如。 将一个复杂类型转换为例如。 二进制JSON表示仍然使用默认映射哈希结构。 第二个选项提供完全控制结果散列。 将对象写入Redis哈希将从哈希中删除内容并重新创建整个哈希，因此不会丢失映射的数据。

*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);

}

}

使用上述byte[]转换器产生例如。

\_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"

|  |  |
| --- | --- |
|  | 自定义转换对索引解析没有影响。 即使对于自定义转换类型，[Secondary Indexes](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.indexes)仍将被创建。 |

8.3. Keyspaces

Keyspaces定义用于为Redis哈希创建实际key的前缀。 默认情况下，前缀设置为getClass().getName()。 这个默认值可以通过聚合根级的@RedisHash或者通过设置编程配置来改变。 但是，annotated keyspace将取代任何其他配置。

*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) 用于启用基于本机Redis结构的查找操作。 值在每次保存时写入相应的索引，并在删除或过期( [expire](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.expirations))时删除。

8.4.1. Simple Property Index

鉴于示例Person实体，我们可以通过使用@Indexed注解属性来为firstname创建一个索引。

*Example 13. Annotation driven indexing*

@RedisHash("persons")

public class Person {

@Id String id;

@Indexed String firstname;

String lastname;

Address address;

}

索引是为实际属性值而建立的。 保存两个人，例如。 “rand”和“aviendha”导致设置如下的索引。

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

此外，编程设置允许定义keys 和 list properties属性上的索引。

@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). |

与*keyspaces* 相同，可以配置索引而不需要注解实际的域类型。

*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(地理空间) Index

假定地址类型包含一个类型为Point的属性位置，该位置保存特定地址的地理坐标。 通过使用@GeoIndexed注释属性，将使用Redis GEO命令添加这些值。

@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 |

在上面的例子中，使用对象id作为成员的名字，使用GEOADD存储lon/lat值。 查找方法允许使用Circle 或Point，距离组合(Distance combinations)来查询这些值。

|  |  |
| --- | --- |
|  | 不可能将near/within与其他标准组合在一起。 |

8.5. Time To Live

存储在Redis中的对象只能在一定的时间内有效。 这对于在Redis中保存短暂的对象特别有用，而不必在达到其寿命时手动删除它们。 以秒为单位的到期时间可以通过@RedisHash(timeToLive=…​)以及通过KeyspaceSettings来设置（请参阅 [Keyspaces](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.keyspaces)）。

可以通过在数字属性或方法上使用@TimeToLive注释来设置更灵活的到期时间。 但是，不要在同一个类中的方法和属性上同时应用@TimeToLive。

*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();

}

}

|  |  |
| --- | --- |
|  | 用@TimeToLive显式注释一个属性会从Redis回读实际的TTL或PTTL值。 -1表示该对象没有过期关联。 |

存储库实现确保通过RedisMessageListenerContainer订阅 [Redis keyspace notifications](http://redis.io/topics/notifications)。

当到期被设置为正值时，执行相应的EXPIRE命令。 除了保留原始文件外，幻影副本在Redis中仍然存在，并在原始文件后5分钟到期。 这样做是为了使存储库支持在密钥过期时通过Springs ApplicationEventPublisher发布RedisKeyExpiredEvent持有过期值，即使原始值已经消失。 所有连接的应用程序将使用Spring Data Redis存储库接收到期事件。

默认情况下，初始化应用程序时，key到期监听器被禁用。 可以在@EnableRedisRepositories或RedisKeyValueAdapter中调整启动模式，以启动应用程序的侦听器，或者在首次插入具有TTL的实体时启动侦听器。 有关可能的值，请参阅EnableKeyspaceEvents。

RedisKeyExpiredEvent将保存实际到期的域对象的副本以及key。

|  |  |
| --- | --- |
|  | 延迟或禁用到期事件侦听器启动会影响RedisKeyExpiredEvent发布。 禁用的事件侦听器不会发布到期事件。 由于延迟的侦听器初始化，延迟的启动可能会导致事件丢失。 |
|  | keyspace通知消息侦听器将改变Redis中的notify-keyspace-events设置（如果尚未设置的话）。 现有的设置不会被覆盖，所以留给用户的时候不要将它们留空。 请注意，在AWS ElastiCache上禁用了CONFIG，并使监听器导致错误。 |

|  |  |
| --- | --- |
|  | Redis Pub/Sub 消息不是持久的。 如果在应用程序关闭期间某个键过期，将不会处理到期事件，这可能会导致二级索引包含对已过期对象的静态引用。 |

8.6. Persisting References

使用@Reference标记属性允许存储简单的键引用，而不是将值复制到散列本身。 在从Redis加载时，引用会自动解析并映射回对象。

*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

|  |  |
| --- | --- |
|  | 引用存储引用对象的整个键(keyspace:id)。 |
|  | 引用对象在保存引用对象时不会保留更改。 请确保分开保存对引用对象的更改，因为只有引用将被存储。 在引用类型的属性上设置的索引将不会被解析。 |

8.7. Persisting Partial Updates

在某些情况下，不需要加载和重写整个实体，只需在其中设置一个新的值即可。 最后一次活动时间的会话时间戳可能是这种情况下，你只是想改变一个属性。 PartialUpdate允许定义对现有对象的设置和删除操作，同时考虑更新实体本身以及索引结构的潜在到期时间。

*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);

|  |  |
| --- | --- |
|  | 将简单属性firstname设置为mat。 |
|  | 将简单的属性address.city设置为emond的字段，而不必传入整个对象。 这在注册自定义转换时不起作用。 |
|  | Remove the property *age*. |
|  | 设置复杂的地址。 |
|  | 设置一个map/collection的集合删除以前存在的map/collection，并用给定的值替换值。 |
|  | 更改生存时间时自动更新服务器到期时间[Time To Live](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#redis.repositories.expirations)。 |
|  | 更新复杂对象以及映射/集合结构需要与Redis进一步交互以确定现有值，这意味着可能会更快地重写整个实体。  U |

8.8. Queries 和 Query Methods

查询方法允许从方法名称自动派生简单的查找器查询。

*Example 19. Sample Repository finder Method*

public interface PersonRepository extends CrudRepository<Person, String> {

List<Person> findByFirstname(String firstname);

}

|  |  |
| --- | --- |
|  | 请确保在查找器方法中使用的属性设置为索引。 |
|  | Redis存储库的查询方法仅支持查询具有分页的实体和实体集合。 |

使用派生查询方法可能并不总是足以对要执行的查询建模。 RedisCallback提供了对索引结构的实际匹配或者甚至自定义添加的更多控制。 所需要的是提供一个RedisCallback，它返回一个或一组Iterable的id值。

*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);

以下是关于Redis支持的keywords的概述，以及包含keyword本质的翻译方法。

| *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

在群集的Redis环境中使用Redis存储库支持很好。 有关ConnectionFactory配置详细信息，请参阅 [Redis Cluster](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#cluster)。 仍然需要考虑一些因素，因为默认的密钥分配会将实体和二级索引分散到整个集群及其插槽(slots)中。

| **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 |

当所有相关密钥映射到同一个插槽时，像SINTER和SUNION这样的命令只能在服务器端进行处理。 否则，计算必须在客户端完成。 因此，将密钥固定到单个插槽是非常有用的，它允许立即使用Redis服务器计算。

| **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

版本库接口的实例通常由一个容器创建，Spring是使用Spring Data时最自然的选择。 有复杂的支持来轻松设置Spring来创建bean实例。 Spring Data Redis附带一个自定义CDI扩展，允许在CDI环境中使用存储库抽象。 该扩展是JAR的一部分，所以您只需要将Spring Data Redis JAR放入类路径即可。

您现在可以通过为RedisConnectionFactory和RedisOperations实现CDI Producer来设置基础结构：

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;

}

}

必要的设置可以根据您运行的JavaEE环境而有所不同。

Spring Data Redis CDI扩展将拾取作为CDI bean提供的所有存储库，并在容器请求存储库类型的bean时为Spring Data存储库创建代理。 因此，获取Spring Data存储库的一个实例是声明一个@Injected属性的事情：

class RepositoryClient {

@Inject

PersonRepository repository;

public void businessMethod() {

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

}

}

Redis存储库需要RedisKeyValueAdapter和RedisKeyValueTemplate实例。 如果没有找到提供的bean，则这些bean由Spring Data CDI扩展创建和管理。 但是，您可以提供自己的Bean来配置RedisKeyValueAdapter和RedisKeyValueTemplate的特定属性。

附录

附录文档结构

参考文档以外的各种附录。

[Schema](https://docs.spring.io/spring-data/redis/docs/2.0.3.RELEASE/reference/html/#appendix:schema) 定义了Spring Data Redis提供的模式。

附录 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>

附录 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