Spring Data Redis

Costin Leau, Jennifer Hickey, Christoph Strobl, Thomas Darimont, Mark Paluch, Jay BryantVersion 1.8.18.RELEASE,2019-01-10

© 2011-2019 The original authors.

|  |  |
| --- | --- |
|  | Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.  本文档的副本可供您自己使用并分发给他人，前提是您不对此类副本收取任何费用，并且每份副本均包含本版权声明，无论是以印刷版还是电子版分发。 |

Table of Contents

* [*Preface*](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_preface)
  + [1. New Features](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#new-features)
    - [1.1. New in Spring Data Redis 1.8](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#new-in-1.8.0)
    - [1.2. New in Spring Data Redis 1.7](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#new-in-1.7.0)
    - [1.3. New in Spring Data Redis 1.6](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#new-in-1-6-0)
    - [1.4. New in Spring Data Redis 1.5](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#new-in-1-5-0)
* [*Introduction*](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#introduction)
  + [2. Why Spring Data Redis?](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#why-spring-redis)
  + [3. Requirements](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#requirements)
  + [4. Getting Started](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#get-started)
    - [4.1. First Steps](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#get-started:first-steps)
      * [4.1.1. Learning Spring](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#get-started:first-steps:spring)
      * [4.1.2. Learning NoSQL and Key Value Stores](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#get-started:first-steps:nosql)
      * [4.1.3. Trying out the Samples](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#get-started:first-steps:samples)
    - [4.2. Need Help?](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#get-started:help)
      * [4.2.1. Community Support](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#get-started:help:community)
      * [4.2.2. Professional Support](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#get-started:help:professional)
    - [4.3. Following Development](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#get-started:up-to-date)
* [*Reference Documentation*](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#reference)
  + [5. Redis support](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis)
    - [5.1. Redis Requirements](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:requirements)
    - [5.2. Redis Support High-level View](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:architecture)
    - [5.3. Connecting to Redis](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:connectors)
      * [5.3.1. RedisConnection and RedisConnectionFactory](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:connectors:connection)
      * [5.3.2. Configuring the Jedis Connector](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:connectors:jedis)
      * [5.3.3. Configuring JRedis connector (Deprecated since 1.7)](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:connectors:jredis)
      * [5.3.4. Configuring SRP connector (Deprecated since 1.7)](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:connectors:srp)
      * [5.3.5. Configuring the Lettuce connector](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:connectors:lettuce)
    - [5.4. Redis Sentinel Support](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:sentinel)
    - [5.5. Working with Objects through RedisTemplate](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:template)
    - [5.6. String-focused Convenience Classes](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:string)
    - [5.7. Serializers](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:serializer)
    - [5.8. Hash mapping](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.hashmappers.root)
      * [5.8.1. Hash Mappers](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_hash_mappers)
      * [5.8.2. Jackson2HashMapper](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.hashmappers.jackson2)
    - [5.9. Redis Messaging (Pub/Sub)](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#pubsub)
      * [5.9.1. Publishing (Sending Messages)](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:pubsub:publish)
      * [5.9.2. Subscribing (Receiving Messages)](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:pubsub:subscribe)
    - [5.10. Redis Transactions](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#tx)
      * [5.10.1. @Transactional Support](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#tx.spring)
    - [5.11. Pipelining](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#pipeline)
    - [5.12. Redis Scripting](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#scripting)
    - [5.13. Support Classes](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:support)
      * [5.13.1. Support for the Spring Cache Abstraction](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:support:cache-abstraction)
  + [6. Redis Cluster](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#cluster)
    - [6.1. Enabling Redis Cluster](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_enabling_redis_cluster)
    - [6.2. Working With Redis Cluster Connection](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_working_with_redis_cluster_connection)
    - [6.3. Working with RedisTemplate and ClusterOperations](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_working_with_code_redistemplate_code_and_code_clusteroperations_code)
  + [7. Redis Repositories](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories)
    - [7.1. Usage](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.usage)
    - [7.2. Object-to-Hash Mapping](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.mapping)
      * [7.2.1. Customizing Type Mapping](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_customizing_type_mapping)
    - [7.3. Keyspaces](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.keyspaces)
    - [7.4. Secondary Indexes](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.indexes)
      * [7.4.1. Simple Property Index](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.indexes.simple)
      * [7.4.2. Geospatial Index](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.indexes.geospatial)
    - [7.5. Time To Live](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.expirations)
    - [7.6. Persisting References](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.references)
    - [7.7. Persisting Partial Updates](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.partial-updates)
    - [7.8. Queries and Query Methods](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.queries)
    - [7.9. Redis Repositories Running on a Cluster](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.cluster)
    - [7.10. CDI Integration](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.cdi-integration)
    - [7.11. Redis Repositories Anatomy](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_redis_repositories_anatomy)
      * [7.11.1. Insert new](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_insert_new)
      * [7.11.2. Replace existing](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_replace_existing)
      * [7.11.3. Save Geo Data](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_save_geo_data)
      * [7.11.4. Find using simple index](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_find_using_simple_index)
      * [7.11.5. Find using Geo Index](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_find_using_geo_index)
* [*Appendixes*](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#appendixes)
  + [Appendix A: Schema](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#appendix:schema)
  + [Appendix B: Command Reference](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#appendix:command-reference)
    - [Supported Commands](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#_supported_commands)

Preface

The Spring Data Redis project applies core Spring concepts to the development of solutions by using a key-value style data store. We provide a “template” as a high-level abstraction for sending and receiving messages. You may notice similarities to the JDBC support in the Spring Framework.

Spring Data Redis项目通过使用键值样式数据存储将核心Spring概念应用于解决方案的开发。 我们提供“模板”作为发送和接收消息的高级抽象。 您可能会注意到Spring Framework中与JDBC支持的相似之处。

1. New Features

This section briefly covers items that are new and noteworthy in the latest releases.

本节简要介绍最新版本中新增和值得注意的项目。

1.1. New in Spring Data Redis 1.8

* Upgrade to Jedis 2.9.
* 升级到Jedis 2.9。
* Upgrade to Lettuce 4.2 (Note: Lettuce 4.2 requires Java 8).
* 升级到Lettuce 4.2（注意：Lettuce 4.2需要Java 8）。
* Support for Redis [GEO](http://redis.io/commands#geo) commands.
* 支持Redis GEO命令。
* Support for Geospatial Indexes using Spring Data Repository abstractions (see [Geospatial Index](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.indexes.geospatial)).
* 使用Spring Data Repository抽象支持地理空间索引（请参阅地理空间索引）。
* MappingRedisConverter-based HashMapper implementation (see [Hash mapping](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.hashmappers.root)).
* 基于MappingRedisConverter的HashMapper实现（请参阅哈希映射）。
* Support for PartialUpdate in repositories (see [Persisting Partial Updates](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.partial-updates)).
* 支持存储库中的PartialUpdate（请参阅保留部分更新）。
* SSL support for connections to Redis cluster.
* SSL支持与Redis群集的连接。
* Support for client name through ConnectionFactory when using Jedis.
* 使用Jedis时通过ConnectionFactory支持客户端名称。

1.2. New in Spring Data Redis 1.7

* Support for [RedisCluster](http://redis.io/topics/cluster-tutorial).
* Support for Spring Data Repository abstractions (see [Redis Repositories](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories)).

1.3. New in Spring Data Redis 1.6

* The Lettuce Redis driver switched from [wg/lettuce](https://github.com/wg/lettuce) to [mp911de/lettuce](https://github.com/mp911de/lettuce).
* Support for ZRANGEBYLEX.
* Enhanced range operations for ZSET, including +inf / -inf.
* Performance improvements in RedisCache, now releasing connections earlier.
* Generic Jackson2 RedisSerializer making use of Jackson’s polymorphic deserialization.

1.4. New in Spring Data Redis 1.5

* Add support for Redis HyperLogLog commands: PFADD, PFCOUNT, and PFMERGE.
* 添加对Redis HyperLogLog命令的支持：PFADD，PFCOUNT和PFMERGE。
* Configurable JavaType lookup for Jackson-based RedisSerializers.

基于Jackson的RedisSerializers的可配置JavaType查找。

* PropertySource-based configuration for connecting to Redis Sentinel (see: [Redis Sentinel Support](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:sentinel)).

用于连接Redis Sentinel的基于PropertySource的配置（请参阅：Redis Sentinel支持）。

Introduction-介绍

This document is the reference guide for Spring Data Redis (SDR) Support. It explains Key-Value module concepts and semantics and the syntax for various stores namespaces.

本文档是Spring Data Redis（SDR）支持的参考指南。 它解释了键值模块的概念和语义以及各种商店命名空间的语法。

For an introduction to key-value stores, Spring, or Spring Data examples, see [Getting Started](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#get-started). This documentation refers only to Spring Data Redis Support and assumes the user is familiar with key-value storage and Spring concepts.

有关键值存储，Spring或Spring Data示例的介绍，请参阅“入门”。 本文档仅涉及Spring Data Redis支持，并假定用户熟悉键值存储和Spring概念。

2. Why Spring Data Redis?- 为什么Spring Data Redis？

The Spring Framework is the leading full-stack Java/JEE application framework. It provides a lightweight container and a non-invasive programming model enabled by the use of dependency injection, AOP, and portable service abstractions.

Spring Framework是领先的全栈Java / JEE应用程序框架。 它提供了一个轻量级容器和一个非侵入式编程模型，它通过使用依赖注入，AOP和便携式服务抽象来实现。

[NoSQL](https://en.wikipedia.org/wiki/NoSQL) storage systems provide an alternative to classical RDBMS for horizontal scalability and speed. In terms of implementation, key-value stores represent one of the largest (and oldest) members in the NoSQL space.

NoSQL存储系统为经典RDBMS提供了水平可扩展性和速度的替代方案。 在实现方面，键值存储代表NoSQL空间中最大（和最老）的成员之一。

The Spring Data Redis (SDR) framework makes it easy to write Spring applications that use the Redis key-value store by eliminating the redundant tasks and boilerplate code required for interacting with the store through Spring’s excellent infrastructure support.

Spring Data Redis（SDR）框架通过Spring的优秀基础架构支持消除了与商店交互所需的冗余任务和样板代码，可以轻松编写使用Redis键值存储的Spring应用程序。

3. Requirements -要求

Spring Data Redis 2.x binaries require JDK level 8.0 and above and [Spring Framework](https://projects.spring.io/spring-framework/) 4.3.22.RELEASE and above.

Spring Data Redis 2.x二进制文件需要JDK 8.0及以上版本以及Spring Framework 4.3.22.RELEASE及以上版本。

In terms of key-value stores, [Redis](http://redis.io/) 2.6.x or higher is required. Spring Data Redis is currently tested against the latest 4.0 release.

在键值存储方面，需要Redis 2.6.x或更高版本。 Spring Data Redis目前已针对最新的4.0版本进行了测试。

4. Getting Started -入门

This section provides an easy-to-follow guide for getting started with the Spring Data Redis module.

本节提供了一个易于遵循的Spring Data Redis模块入门指南。

4.1. First Steps -第一步

As explained in [Why Spring Data Redis?](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#why-spring-redis), Spring Data Redis (SDR) provides integration between the Spring framework and the Redis key-value store. Consequently, you should become acquainted with both of these frameworks. Throughout the SDR documentation, each section provides links to relevant resources. However, you should become familiar with these topics before reading this guide.

正如Spring Data Data Redis中所解释的那样，Spring Data Redis（SDR）提供了Spring框架和Redis键值存储之间的集成。 因此，您应该熟悉这两个框架。 在整个SDR文档中，每个部分都提供了相关资源的链接。 但是，在阅读本指南之前，您应该熟悉这些主题。

4.1.1. Learning Spring-学习spring

Spring Data uses Spring framework’s [core](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/spring-core.html) functionality, such as the [IoC](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html) container, [resource](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/resources.html) abstract, and the [AOP](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html)infrastructure. While it is not important to know the Spring APIs, understanding the concepts behind them is important. At a minimum, the idea behind IoC should be familiar. That being said, the more knowledge you have about the Spring, the faster you can pick up Spring Data Redis. In addition to the Spring Framework’s comprehensive documentation, there are a lot of articles, blog entries, and books on the matter. The Spring Guides [home page](https://spring.io/guides) offer a good place to start. In general, this should be the starting point for developers wanting to try Spring Data Redis.

Spring Data使用Spring框架的核心功能，例如IoC容器，[resource](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/resources.html) 抽象和AOP 基础设施。 虽然了解Spring API并不重要，但了解它们背后的概念非常重要。 至少，IoC背后的想法应该是熟悉的。 话虽这么说，你对Spring的了解越多，你获得Spring Data Redis的速度就越快。 除了Spring Framework的综合文档之外，还有很多关于此事的文章，博客文章和书籍。 Spring Guides主页提供了一个很好的起点。 一般来说，这应该是想要尝试Spring Data Redis的开发人员的起点。

4.1.2. Learning NoSQL and Key Value Stores-学习NoSQL和Key Value对存储

NoSQL stores have taken the storage world by storm. It is a vast domain with a plethora of solutions, terms, and patterns (to make things worse, even the term itself has multiple [meanings](https://www.google.com/search?q=nosoql+acronym)). While some of the principles are common, it is crucial that you be familiar to some degree with the stores supported by SDR. The best way to get acquainted with these solutions is to read their documentation and follow their examples. It usually does not take more then five to ten minutes to go through them and, if you come from an RDMBS-only background, many times these exercises can be eye-openers.

NoSQL存储风靡了存储世界。 这是一个涉及众多解决方案，术语和模式的庞大领域（更糟糕的是，甚至术语本身也有多重含义）。 虽然一些原则很常见，但在某种程度上熟悉SDR支持的存储至关重要。 熟悉这些解决方案的最佳方式是阅读他们的文档并按照他们的示例进行操作。 它通常不需要花费五到十分钟来完成它们，如果你来自RDMBS背景，很多时候这些练习可以让人大开眼界。

4.1.3. Trying out the Samples – 简单尝试

One can find various samples for key-value stores in the dedicated Spring Data example repo, at [http://github.com/spring-projects/spring-data-keyvalue-examples](https://github.com/spring-projects/spring-data-keyvalue-examples). For Spring Data Redis, you should pay particular attention to the retwisj sample, a Twitter-clone built on top of Redis that can be run locally or be deployed into the cloud. See its [documentation](http://static.springsource.org/spring-data/data-keyvalue/examples/retwisj/current/), the following blog [entry](http://blog.springsource.com/2011/04/27/getting-started-redis-spring-cloud-foundry/) for more information.

可以在http://github.com/spring-projects/spring-data-keyvalue-examples的专用Spring Data示例仓库中找到各种键值存储样本。 对于Spring Data Redis，您应该特别注意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。 有关详细信息，请参阅其文档，以下博客条目。

4.2. Need Help? -需要帮忙？

If you encounter issues or you are just looking for advice, use one of the links below:

如果您遇到问题或者您只是在寻找建议，请使用以下链接之一：

4.2.1. Community Support -社区支持

The Spring Data tag on [Stack Overflow](https://stackoverflow.com/questions/tagged/spring-data) is a message board for all Spring Data (not just Redis) users to share information and help each other. Note that registration is needed **only** for posting.

Stack Overflow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留言板。 请注意，仅发布时才需要注册。

4.2.2. Professional Support -专业支持

Professional, from-the-source support, with guaranteed response time, is available from [Pivotal Software, Inc.](http://www.gopivotal.com/), the company behind Spring Data and Spring.

专业的，源外支持，保证响应时间，可从Pivotal Software，Inc。（Spring Data and Spring公司）获得。

4.3. Following Development –后续发展

For information on the Spring Data source code repository, nightly builds, and snapshot artifacts, see the Spring Data home [page](https://spring.io/spring-data).

开发之后有关Spring Data源代码库，夜间构建和快照构件的信息，请参阅Spring Data主页。

You can help make Spring Data best serve the needs of the Spring community by interacting with developers on Stack Overflow at either [spring-data](https://stackoverflow.com/questions/tagged/spring-data) or [spring-data-redis](https://stackoverflow.com/questions/tagged/spring-data-redis).

通过在Spring-data或spring-data-redis上与Stack Overflow上的开发人员交互，您可以帮助Spring Data最好地满足Spring社区的需求。

If you encounter a bug or want to suggest an improvement (including to this documentation), please create a ticket on the Spring Data issue [tracker](https://jira.spring.io/browse/DATAREDIS).

如果您遇到错误或想要建议改进（包括本文档），请在Spring Data问题跟踪器上创建一个票证。

To stay up to date with the latest news and announcements in the Spring eco system, subscribe to the Spring Community [Portal](https://spring.io/).

要及时了解Spring eco系统中的最新新闻和公告，请订阅Spring社区门户。

Lastly, you can follow the Spring [blog](https://spring.io/blog/) or the project team ([Thomas](https://twitter.com/thomasdarimont) and [Christoph](https://twitter.com/stroblchristoph)) on Twitter.

最后，您可以在Twitter上关注Spring博客或项目团队（Thomas和Christoph）。

Reference Documentation

参考文档

Document structure- 文档结构

This part of the reference documentation explains the core functionality offered by Spring Data Redis.

这部分参考文档解释了Spring Data Redis提供的核心功能。

[Redis support](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis) introduces the Redis module feature set.

Redis支持引入了Redis模块功能集。

5. Redis support –redis支持

One of the key-value stores supported by Spring Data is [Redis](http://redis.io/). To quote the Redis project home page:

Spring Data支持的其中一个键值存储是Redis。 引用Redis项目主页：

*Redis is an advanced key-value store. It is similar to memcached but the dataset is not volatile, and values can be strings, exactly like in memcached, but also lists, sets, and ordered sets. All this data types can be manipulated with atomic operations to push/pop elements, add/remove elements, perform server side union, intersection, difference between sets, and so forth. Redis supports different kind of sorting abilities.*

*Redis是一家高级键值存储库。 它与memcached类似，但数据集不是易失性的，值可以是字符串，与memcached完全相同，但也可以是列表，集合和有序集。 所有这些数据类型都可以通过原子操作来操作，以推送/弹出元素，添加/删除元素，执行服务器端并集，交集，集合之间的差异等等。 Redis支持不同类型的排序功能。*

Spring Data Redis provides easy configuration and access to Redis from Spring applications. It offers both low-level and high-level abstractions for interacting with the store, freeing the user from infrastructural concerns.

Spring Data Redis提供了从Spring应用程序轻松配置和访问Redis的功能。 它提供了与存储仓库交互的低级和高级抽象，使用户免于基础设施问题。

5.1. Redis Requirements- Redis要求

Spring Data Redis requires Redis 2.6 or above and Java SE 6.0 or above . In terms of language bindings (or connectors), Spring Redis integrates with [Jedis](https://github.com/xetorthio/jedis), [JRedis](https://github.com/alphazero/jredis) (Deprecated since 1.7), [SRP](https://github.com/spullara/redis-protocol) (Deprecated since 1.7) and [Lettuce](https://github.com/wg/lettuce), four popular open-source Java libraries for Redis. If you are aware of any other connector that we should be integrating with please send us feedback.

Spring Data Redis需要Redis 2.6或更高版本以及Java SE 6.0或更高版本。 在语言绑定（或连接器）方面，Spring Redis集成了Jedis，JRedis（自1.7以来已弃用），SRP（自1.7以来已弃用）和Lettuce，这是Redis的四个流行的开源Java库。 如果您了解我们应该集成的任何其他连接器，请向我们发送反馈。

5.2. Redis Support High-level View- Redis支持高级视图

The Redis support provides several components. For most tasks, the high-level abstractions and support services are the best choice. Note that, at any point, you can move between layers. For example, you can get a low-level connection (or even the native library) to communicate directly with Redis.

Redis支持提供了几个组件。 对于大多数任务，高级抽象和支持服务是最佳选择。 请注意，在任何时候，您都可以在图层之间移动。 例如，您可以获得与Redis直接通信的低级连接（甚至是本机库）。

5.3. Connecting to Redis-连接到Redis

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 you choose, you need to use only one set of Spring Data Redis APIs (which behaves consistently across all connectors): the org.springframework.data.redis.connection package and its RedisConnection and RedisConnectionFactory interfaces for working with and retrieving active connections to Redis.

使用Redis和Spring时的首要任务之一是通过IoC容器连接到存储仓库。 为此，需要Java连接器（或绑定）。 无论您选择哪个库，都只需要使用一组Spring Data Redis API（在所有连接器中表现一致）：org.springframework.data.redis.connection包及其RedisConnection和RedisConnectionFactory接口，用于处理和检索 与Redis的活动连接。

5.3.1. RedisConnection and RedisConnectionFactory

RedisConnection和RedisConnectionFactory

RedisConnection provides the core building block for Redis communication, as it handles the communication with the Redis back end. It also automatically translates the underlying connecting library exceptions to Spring’s consistent DAO exception [hierarchy](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/dao.html#dao-exceptions) so that you can switch the connectors without any code changes, as the operation semantics remain the same.

RedisConnection为Redis通信提供核心构建块，因为它处理与Redis后端的通信。它还会自动将底层连接库异常转换为Spring的一致DAO异常层次结构，以便您可以在不更改任何代码的情况下切换连接器，因为操作语义保持不变。

For the corner cases where the native library API is required, RedisConnection provides a dedicated method (getNativeConnection) that returns the raw, underlying object used for communication.

对于需要本机库API的极端情况，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。

Active RedisConnection objects are created through RedisConnectionFactory. In addition, the factory acts as PersistenceExceptionTranslator objects, meaning that, once declared, they let you do transparent exception translation. For example, you can do exception translation through the use of the @Repository annotation and AOP. For more information, see the dedicated [section](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/orm.html#orm-exception-translation) in the Spring Framework documentation.

激活RedisConnection对象是通过RedisConnectionFactory创建的。此外，工厂充当PersistenceExceptionTranslator对象，这意味着，一旦声明，它们就会让您进行透明的异常转换。例如，您可以通过使用@Repository注释和AOP进行异常转换。有关更多信息，请参阅Spring Framework文档中的专用部分。

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

根据基础配置，工厂可以返回新连接或现有连接（使用池或共享本机连接时）。

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

使用RedisConnectionFactory最简单的方法是通过IoC容器配置相应的连接器并将其注入using类。

Unfortunately, currently, not all connectors support all Redis features. When invoking a method on the Connection API that is unsupported by the underlying library, an UnsupportedOperationException is thrown.

不幸的是，目前并非所有连接器都支持所有Redis功能。在基础库不支持的Connection API上调用方法时，将引发UnsupportedOperationException。

5.3.2. Configuring the Jedis Connector-配置Jedis连接器

[Jedis](https://github.com/xetorthio/jedis) is one of the connectors supported by the Spring Data Redis module through the org.springframework.data.redis.connection.jedis package. In its simplest form, the Jedis configuration looks as follow:

Jedis是Spring Data Redis模块通过org.springframework.data.redis.connection.jedis包支持的连接器之一。 在最简单的形式中，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>

For production use, however, you might want to tweak settings such as the host or password, as shown in the following example:

但是，对于生产用途，您可能需要调整主机或密码等设置，如以下示例所示：

<?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. Configuring JRedis connector (Deprecated since 1.7)- 配置JRedis连接器（自1.7起不推荐使用）

[JRedis](https://github.com/alphazero/jredis) is another popular, open-source connector supported by Spring Data Redis through the org.springframework.data.redis.connection.jredis package.

JRedis是另一种流行的开源连接器，由Spring Data Redis通过org.springframework.data.redis.connection.jredis包支持。

A typical JRedis configuration can looks like this:

典型的JRedis配置可能如下所示：

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

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

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

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="jredisConnectionFactory" class="org.springframework.data.redis.connection.jredis.JredisConnectionFactory" p:host-name="server" p:port="6379"/>

</beans>

The configuration is quite similar to Jedis, with one notable exception. By default, the JedisConnectionFactory pools connections. In order to use a connection pool with JRedis, configure the JredisConnectionFactory with an instance of JredisPool. For example:

配置与Jedis非常相似，但有一个值得注意的例外。 默认情况下，JedisConnectionFactory池连接。 要使用JRedis连接池，请使用JredisPool实例配置JredisConnectionFactory。 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"

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

<bean id="jredisConnectionFactory" class="org.springframework.data.redis.connection.jredis.JredisConnectionFactory">

<constructor-arg>

<bean class="org.springframework.data.redis.connection.jredis.DefaultJredisPool">

<constructor-arg value="localhost" />

<constructor-arg value="6379" />

</bean>

</constructor-arg>

</bean>

</beans>

5.3.4. Configuring SRP connector (Deprecated since 1.7)- 配置SRP连接器（自1.7起不推荐使用）

[SRP](https://github.com/spullara/redis-protocol) (an acronym for Sam’s Redis Protocol) is the third open-source connector supported by Spring Data Redis through the org.springframework.data.redis.connection.srp package.

SRP（Sam的Redis协议的首字母缩写）是Spring Data Redis通过org.springframework.data.redis.connection.srp包支持的第三个开源连接器。

By now, its configuration is probably easy to guess:

到目前为止，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="srpConnectionFactory" class="org.springframework.data.redis.connection.srp.SrpConnectionFactory" p:host-name="server" p:port="6379"/>

</beans>

Needless to say, the configuration is quite similar to that of the other connectors.

不用说，该配置与其他连接器的配置非常相似。

5.3.5. Configuring the Lettuce connector-配置Lettuce连接器

[Lettuce](https://github.com/mp911de/lettuce) is the fourth open-source connector supported by Spring Data Redis through the org.springframework.data.redis.connection.lettuce package.

Lettuce是Spring Data Redis通过org.springframework.data.redis.connection.lettuce包支持的第四个开源连接器。

Its configuration is probably easy to guess:

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>

There are also a few Lettuce-specific connection parameters that can be tweaked. By default, all LettuceConnection s created by the LettuceConnectionFactory share the same thread-safe native connection for all non-blocking and non-transactional operations. Set shareNativeConnection to false to use a dedicated connection each time. LettuceConnectionFactory can also be configured with a LettucePool to use for pooling blocking and transactional connections, or all connections if shareNativeConnection is set to false.

还有一些可以调整的特定生菜特定连接参数。 默认情况下，LettuceConnectionFactory创建的所有LettuceConnection都为所有非阻塞和非事务操作共享相同的线程安全本机连接。 将shareNativeConnection设置为false以每次使用专用连接。 LettuceConnectionFactory也可以使用LettucePool配置以用于池化阻塞和事务连接，或者如果shareNativeConnection设置为false则配置所有连接。

5.4. Redis Sentinel Support- Redis Sentinel支持

For dealing with high-availability Redis, Spring Data Redis has support for [Redis Sentinel](http://redis.io/topics/sentinel), using RedisSentinelConfiguration, as shown in the following example:

为了处理高可用性Redis，Spring Data Redis使用RedisSentinelConfiguration支持Redis Sentinel，如以下示例所示：

Please note that currently only [Jedis](https://github.com/xetorthio/jedis) and lettuce [Lettuce](https://github.com/mp911de/lettuce) support Redis Sentinel.

请注意，目前只有Jedis和lettuce lettuce 支持Redis Sentinel。

/\*\*

\* jedis

\*/

@Bean

public RedisConnectionFactory jedisConnectionFactory() {

RedisSentinelConfiguration sentinelConfig = new RedisSentinelConfiguration() .master("mymaster")

.sentinel("127.0.0.1", 26379) .sentinel("127.0.0.1", 26380);

return new JedisConnectionFactory(sentinelConfig);

}

/\*\*

\* 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 can also be defined with a PropertySource, which lets you set the following properties:

RedisSentinelConfiguration也可以使用PropertySource定义，它允许您设置以下属性：

Configuration Properties

配置属性

spring.redis.sentinel.master: name of the master node.

spring.redis.sentinel.master：主节点的名称。

spring.redis.sentinel.nodes: Comma delimited list of host:port pairs.

spring.redis.sentinel.nodes：逗号分隔的主机：端口对列表。

Sometimes, direct interaction with one of the Sentinels is required. Using RedisConnectionFactory.getSentinelConnection() or RedisConnection.getSentinelCommands() gives you access to the first active Sentinel configured.

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

5.5. Working with Objects through RedisTemplate

通过RedisTemplate处理对象

Most users are likely to use RedisTemplate and its coresponding 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 RedisConnection offers low-level methods that accept and return binary values (byte arrays), the template takes care of serialization and connection management, freeing the user from dealing with such details.

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

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

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

| *Table 1. Operational views* | |
| --- | --- |
| **Interface** | **Description** |
| *Key Type Operations* | |
| GeoOperations | Redis geospatial operations, such as GEOADD, GEORADIUS,…​  Redis地理空间操作，例如GEOADD，GEORADIUS，...... |
| HashOperations | Redis hash operations  Redis哈希操作 |
| HyperLogLogOperations | Redis HyperLogLog operations, such as PFADD, PFCOUNT,…​  Redis HyperLogLog操作，例如PFADD，PFCOUNT，...... |
| ListOperations | Redis list operations  Redis列表操作 |
| SetOperations | Redis set operations  Redis set操作 |
| ValueOperations | Redis string (or value) operations - Redis字符串（或值）操作 |
| ZSetOperations | Redis zset (or sorted set) operations |
| HashOperations | Redis hash operations |
| HyperLogLogOperations | Redis HyperLogLog operations like (pfadd, pfcount,…​) |
| GeoOperations | Redis geospatial operations like GEOADD, GEORADIUS,…​) |
| *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 |
| BoundHashOperations | Redis hash key bound operations |
| BoundGeoOperations | Redis key bound geospatial operations. |

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

配置完成后，模板是线程安全的，可以跨多个实例重用。

RedisTemplate uses a Java-based serializer for most of its operations. This means that any object written or read by the template is serialized and deserialized through Java. You can change the serialization mechanism on the template, and the Redis module offers several implementations, which are available in the org.springframework.data.redis.serializer package. See [Serializers](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:serializer) for more information. You can also set any of the serializers to null and use RedisTemplate with raw byte arrays by setting the enableDefaultSerializer property to false. Note that the template requires all keys to be non-null. However, values can be null as long as the underlying serializer accepts them. Read the Javadoc of each serializer for more information.

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

For cases where you need a certain template view, declare the view as a dependency and inject the template. The container automatically performs the conversion, eliminating the opsFor[X] calls, as shown in the following example:

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

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

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

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

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

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

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

<!-- redis template definition -->

<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate" p:connection-factory-ref="jedisConnectionFactory"/>

...

</beans>

public class Example {

// inject the actual template

@Autowired

private RedisTemplate<String, String> template;

// inject the template as ListOperations

@Resource(name="redisTemplate")

private ListOperations<String, String> listOps;

public void addLink(String userId, URL url) {

listOps.leftPush(userId, url.toExternalForm());

}

}

5.6. String-focused Convenience Classes

以字符串为中心的便捷类

Since it is quite common for the keys and values stored in Redis to be java.lang.String, the Redis modules provides two extensions to RedisConnection and RedisTemplate, respectively the StringRedisConnection (and its DefaultStringRedisConnection implementation) and StringRedisTemplate as a convenient one-stop solution for intensive String operations. In addition to being bound to String keys, the template and the connection use the StringRedisSerializerunderneath, which means the stored keys and values are human-readable (assuming the same encoding is used both in Redis and your code). The following listings show an example:

由于存储在Redis中的键和值通常是java.lang.String，因此Redis模块提供了对RedisConnection和RedisTemplate的两个扩展，分别是StringRedisConnection（及其DefaultStringRedisConnection实现）和StringRedisTemplate作为方便的一站式解决方案 用于密集的String操作。 除了绑定到String键之外，模板和连接还使用StringRedisSerializerunderneath，这意味着存储的键和值是人类可读的（假设在Redis和代码中都使用相同的编码）。 以下列表显示了一个示例：

<?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());

}

}

As with the other Spring templates, RedisTemplate and StringRedisTemplate let you talk directly to Redis through the RedisCallback interface. This feature gives complete control to you, as it talks directly to the RedisConnection. Note that the callback receives an instance of StringRedisConnection when a StringRedisTemplate is used. The following example shows how to use the RedisCallback interface:

与其他Spring模板一样，RedisTemplate和StringRedisTemplate允许您通过RedisCallback接口直接与Redis对话。 此功能为您提供完全控制，因为它直接与RedisConnection对话。 请注意，当使用StringRedisTemplate时，回调会接收StringRedisConnection的实例。 以下示例显示如何使用RedisCallback接口：

public void useCallback() {

redisTemplate.execute(new RedisCallback<Object>() {

public Object doInRedis(RedisConnection connection) throws DataAccessException {

Long size = connection.dbSize();

// Can cast to StringRedisConnection if using a StringRedisTemplate

((StringRedisConnection)connection).set("key", "value");

}

});

}

5.7. Serializers-序列化

From the framework perspective, the data stored in Redis is only bytes. While Redis itself supports various types, for the most part, these refer to the way the data is stored rather than what it represents. It is up to the user to decide whether the information gets translated into strings or any other objects.

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

In Spring Data, the conversion between the user (custom) types and raw data (and vice-versa) is handled Redis in the org.springframework.data.redis.serializer package.

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

This package contains two types of serializers that, as the name implies, take care of the serialization process:

该软件包包含两种类型的序列化程序，顾名思义，它们负责序列化过程：

* Two-way serializers based on RedisSerializer.

基于RedisSerializer的双向序列化程序。

* Element readers and writers that use RedisElementReader and RedisElementWriter.

使用RedisElementReader和RedisElementWriter的元素读者和编写者。

The main difference between these variants is that RedisSerializer primarily serializes to byte[] while readers and writers use ByteBuffer.

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

Multiple implementations are available (including two that have been already mentioned in this documentation):

可以使用多种实现（包括本文档中已经提到的两种实现）：

* JdkSerializationRedisSerializer, which is used by default for RedisCache and RedisTemplate.

JdkSerializationRedisSerializer，默认情况下用于RedisCache和RedisTemplate。

* the StringRedisSerializer.

StringRedisSerializer。

However one can use OxmSerializer for Object/XML mapping through Spring [OXM](https://docs.spring.io/spring/docs/4.3.22.RELEASE/spring-framework-reference/data-access.html#oxm) support or Jackson2JsonRedisSerializeror GenericJackson2JsonRedisSerializer for storing data in [JSON](https://en.wikipedia.org/wiki/JSON) format.

但是，可以使用OxmSerializer通过Spring OXM支持进行对象/ XML映射，或使用Jackson2JsonRedisSerializeror GenericJackson2JsonRedisSerializer以JSON格式存储数据。

Do note that the storage format is not limited only to values. It can be used for keys, values, or hashes without any restrictions.

请注意，存储格式不仅限于值。 它可以用于键，值或哈希，没有任何限制。

By default, RedisCache and RedisTemplate are configured to use Java native serialization. Java native serialization is known for allowing remote code execution caused by payloads that exploit vulnerable libraries and classes injecting unverified bytecode. Manipulated input could lead to unwanted code execution in the application during the deserialization step. As a consequence, do not use serialization in untrusted environments. In general, we strongly recommend any other message format (such as JSON) instead.

默认情况下，RedisCache和RedisTemplate配置为使用Java本机序列化。 众所周知，Java本机序列化允许由利用易受攻击的库和类注入未经验证的字节码的有效负载引起的远程代码执行。 在反序列化步骤中，操作输入可能导致应用程序中不需要的代码执行。 因此，请勿在不受信任的环境中使用序列化。 通常，我们强烈建议使用任何其他消息格式（例如JSON）。

If you are concerned about security vulnerabilities due to Java serialization, consider the general-purpose serialization filter mechanism at the core JVM level, originally developed for JDK 9 but backported to JDK 8, 7, and 6:

如果您担心Java序列化导致的安全漏洞，请考虑核心JVM级别的通用序列化过滤机制，最初为JDK 9开发但向后移植到JDK 8,7和6：

• Filter Incoming Serialization Data.

过滤传入的序列化数据。

• JEP 290.

JEP 290。

• OWASP: Deserialization of untrusted data.

OWASP：不受信任数据的反序列化。

5.8. Hash mapping-哈希映射

Data can be stored by using various data structures within Redis. Jackson2JsonRedisSerializer can convert objects in [JSON](https://en.wikipedia.org/wiki/JSON)format. Ideally, JSON can be stored as a value by using plain keys. You can achieve a more sophisticated mapping of structured objects by using Redis hashes. Spring Data Redis offers various strategies for mapping data to hashes (depending on the use case):

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

* Direct mapping, by using HashOperations and a [serializer](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:serializer)

使用HashOperations和序列化程序直接映射

* Using [Redis Repositories](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories)

使用Redis存储库

* Using HashMapper and HashOperations

使用HashMapper和HashOperations

5.8.1. Hash Mappers -哈希映射器

Hash mappers are converters of map objects to a Map<K, V> and back. HashMapper is intended for using with Redis Hashes.

散列映射器是地图对象到Map <K，V>和后面的转换器。 HashMapper旨在与Redis Hashes一起使用。

Multiple implementations are available:

有多种实现方式：

* BeanUtilsHashMapper using Spring’s [BeanUtils](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/BeanUtils.html).

使用Spring的BeanUtils的BeanUtilsHashMapper。

* ObjectHashMapper using [Object-to-Hash Mapping](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.mapping).

ObjectHashMapper使用Object-to-Hash Mapping。

* [Jackson2HashMapper](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.hashmappers.jackson2) using [FasterXML Jackson](https://github.com/FasterXML/jackson).

Jackson2HashMapper使用FasterXML Jackson。

The following example shows one way to implement hash mapping:

以下示例显示了实现哈希映射的一种方法：

public class Person {

String firstname;

String lastname;

// …

}

public class HashMapping {

@Autowired

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

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

public void writeHash(String key, Person person) {

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

hashOperations.putAll(key, mappedHash);

}

public Person loadHash(String key) {

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

return (Person) mapper.fromHash(loadedHash);

}

}

5.8.2. Jackson2HashMapper

Jackson2HashMapper provides Redis Hash mapping for domain objects by using [FasterXML Jackson](https://github.com/FasterXML/jackson). Jackson2HashMapper can map top-level properties as Hash field names and, optionally, flatten the structure. Simple types map to simple values. Complex types (nested objects, collections, maps, and so on) are represented as nested JSON.

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

Flattening creates individual hash entries for all nested properties and resolves complex types into simple types, as far as possible.

展平为所有嵌套属性创建单独的哈希条目，并尽可能将复杂类型解析为简单类型。

Consider the following class and the data structure it contains:

考虑以下类及其包含的数据结构：

public class Person {

String firstname;

String lastname;

Address address;

}

public class Address {

String city;

String country;

}

The following table shows how the data in the preceding class would appear in normal mapping:

下表显示了前一类中的数据如何在法线贴图中显示：

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

The following table shows how the data in the preceding class would appear in flat mapping:

下表显示了前一类中的数据如何在平面映射中显示：

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

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

5.9. Redis Messaging (Pub/Sub)- Redis消息（Pub / Sub）

Spring Data provides dedicated messaging integration for Redis, similar in functionality and naming to the JMS integration in Spring Framework.

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

Redis messaging can be roughly divided into two areas of functionality:

Redis消息传递大致可分为两个功能区域：

* Publication or production of messages

发布或制作消息

* Subscription or consumption of messages

订阅或消费消息

This is an example of the pattern often called Publish/Subscribe (Pub/Sub for short). 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.

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

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

org.springframework.data.redis.connection和org.springframework.data.redis.listener包为Redis消息传递提供核心功能。

5.9.1. Publishing (Sending Messages)- 发布（发送消息）

To publish a message, you can use, as with the other operations, either the low-level RedisConnection or the high-level RedisTemplate. Both entities offer the publish method, which accepts the message and the destination channel as arguments. While RedisConnection requires raw data (array of bytes), the RedisTemplate lets arbitrary objects be passed in as messages, as shown in the following example:

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

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

byte[] msg = ...

byte[] channel = ...

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

RedisTemplate template = ...

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

5.9.2. Subscribing (Receiving Messages)- 订阅（接收消息）

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 lets multiple subscriptions be created with one command but can also listen on channels not yet created at subscription time (as long as they match the pattern).

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

At the low-level, RedisConnection offers the subscribe and pSubscribe methods that map the Redis commands for subscribing by channel or by pattern, respectively. Note that multiple channels or patterns can be used as arguments. To change the subscription of a connection or query whether it is listening, RedisConnection provides the getSubscription and isSubscribed methods.

在低级别，RedisConnection提供subscribe和pSubscribe方法，分别映射Redis命令以按通道或按模式进行订阅。请注意，可以使用多个通道或模式作为参数。要更改连接或查询是否正在侦听的订阅，RedisConnection将提供getSubscription和isSubscribed方法。

Subscription commands in Spring Data Redis are blocking. That is, calling subscribe on a connection causes the current thread to block as it starts waiting for messages. The thread is released only if the subscription is canceled, which happens when another thread invokes unsubscribe or pUnsubscribe on the **same** connection. See “[Message Listener Containers](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:pubsub:subscribe:containers)” (later in this document) for a solution to this problem.

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

As mentioned earlier, once subscribed, a connection starts waiting for messages. Only commands that add new subscriptions, modify existing subscriptions, and cancel existing subscriptions are allowed. Invoking anything other than subscribe, pSubscribe, unsubscribe, or pUnsubscribe throws an exception.

如前所述，一旦订阅，连接就开始等待消息。仅允许添加新订阅，修改现有订阅和取消现有订阅的命令。调用除subscribe，pSubscribe，unsubscribe或pUnsubscribe之外的任何内容都会引发异常。

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

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

Message Listener Containers-消息侦听器容器

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. If you are familiar with EJB and JMS, you should find the concepts familiar, as it is designed to be as close as possible to the support in Spring Framework and its message-driven POJOs (MDPs).

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

RedisMessageListenerContainer acts as a message listener container. It is used to receive messages from a Redis channel and drive the MessageListener instances 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 lets you as an application developer write the (possibly complex) business logic associated with receiving a message (and reacting to it) and delegates boilerplate Redis infrastructure concerns to the framework.

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

Furthermore, to minimize the application footprint, RedisMessageListenerContainer lets one connection and one thread 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 remains the same throughout its lifetime. Moreover, the container allows runtime configuration changes so that you can add or remove listeners while an application is running without the need for a 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 thread is released.

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

To help with the asynchronous nature 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, you should change or tweak the executor to better serve your needs. In particular, in managed environments (such as app servers), it is highly recommended to pick a proper TaskExecutor to take advantage of its runtime.

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

The MessageListenerAdapter

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

MessageListenerAdapter类是Spring异步消息传递支持的最后一个组件。 简而言之，它允许您将几乎任何类暴露为MDP（尽管存在一些约束）。

Consider the following interface definition:

请考虑以下接口定义：

public interface MessageDelegate {

void handleMessage(String message);

void handleMessage(Map message); void handleMessage(byte[] message);

void handleMessage(Serializable message);

// pass the channel/pattern as well

void handleMessage(Serializable message, String channel);

}

Notice that, although the interface does not extend the MessageListener interface, it can still be used as a MDP by using 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:

请注意，虽然接口不扩展MessageListener接口，但仍可以使用MessageListenerAdapter类将其用作MDP。 还要注意各种消息处理方法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。 此外，发送消息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 class DefaultMessageDelegate implements MessageDelegate {

// implementation elided for clarity...

}

Notice 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 make into an MDP with the following configuration:

注意MessageDelegate接口的上述实现（上面的DefaultMessageDelegate类）根本没有Redis依赖。 它确实是我们在MDP中使用以下配置制作的POJO：

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

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

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

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

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

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

<!-- the default ConnectionFactory -->

<redis:listener-container>

<!-- the method attribute can be skipped as the default method name is "handleMessage" -->

<redis:listener ref="listener" method="handleMessage" topic="chatroom" />

</redis:listener-container>

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

...

<beans>

The listener topic can be either a channel (for example, topic="chatroom") or a pattern (for example, topic="\*room")

侦听器主题可以是通道（例如，topic =“chatroom”）或模式（例如，topic =“\* room”）

The preceding example uses the Redis namespace to declare the message listener container and automatically register the POJOs as listeners. The full blown beans definition follows:

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

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

<constructor-arg>

<bean class="redisexample.DefaultMessageDelegate"/>

</constructor-arg>

</bean>

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

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

<property name="messageListeners">

<map>

<entry key-ref="messageListener">

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

<constructor-arg value="chatroom">

</bean>

</entry>

</map>

</property>

</bean>

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

每次收到消息时，适配器都会自动且透明地在低级格式和所需对象类型之间执行转换（使用配置的RedisSerializer）。 由方法调用引起的任何异常都由容器捕获并处理（默认情况下，会记录异常）。

5.10. Redis Transactions- Redis交易

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 with the same connection.

Redis通过multi，exec和discard命令为事务提供支持。 RedisTemplate上提供了这些操作。 但是，不保证RedisTemplate使用相同的连接执行事务中的所有操作。

Spring Data Redis provides the SessionCallback interface for use when multiple operations need to be performed with the same connection, such as when using Redis transactions. The following example uses the multi method:

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，例如使用Redis事务时。 以下示例使用multi方法：

//execute a transaction

List<Object> txResults = redisTemplate.execute(new SessionCallback<List<Object>>() {

public List<Object> execute(RedisOperations operations) throws DataAccessException {

operations.multi();

operations.opsForSet().add("key", "value1");

// This will contain the results of all operations in the transaction

return operations.exec();

}

});

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

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

RedisTemplate使用其值，散列键和散列值序列化程序在返回之前反序列化exec的所有结果。还有一个额外的exec方法，允许您为事务结果传递自定义序列化程序。

As of version 1.1, an important change has been made to the exec methods of RedisConnection and RedisTemplate. 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, zAdd returns a boolean indicating whether 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 such as 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, set convertPipelineAndTxResults to false on your RedisConnectionFactory to disable this behavior.

从版本1.1开始，RedisConnection和RedisTemplate的exec方法发生了重大变化。以前，这些方法直接从连接器返回事务的结果。这意味着数据类型通常与RedisConnection方法返回的数据类型不同。例如，zAdd返回一个布尔值，指示元素是否已添加到有序集合中。大多数连接器将此值作为long返回，Spring Data Redis执行转换。另一个常见的区别是大多数连接器为诸如set之类的操作返回状态答复（通常是字符串，OK）。 Spring Data Redis通常会丢弃这些回复。在1.1之前，没有对exec的结果进行这些转换。此外，RedisTemplate中的结果未反序列化，因此它们通常包含原始字节数组。如果此更改破坏了您的应用程序，请在RedisConnectionFactory上将convertPipelineAndTxResults设置为false以禁用此行为。

5.10.1. @Transactional Support

By default, transaction Support is disabled and has to be explicitly enabled for each RedisTemplate in use by setting setEnableTransactionSupport(true). Doing so forces binding the current RedisConnection to the current Thread that is triggering MULTI. If the transaction finishes without errors, EXEC is called. Otherwise DISCARD is called. Once in MULTI, RedisConnection queues write operations. All readonly operations, such as KEYS, are piped to a fresh (non-thread-bound) RedisConnection.

The following example shows how to configure transaction management:

*Example 1. Configuration enabling Transaction Management*

@Configuration

@EnableTransactionManagement

public class RedisTxContextConfiguration {

@Bean

public StringRedisTemplate redisTemplate() {

StringRedisTemplate template = new StringRedisTemplate(redisConnectionFactory());

// explicitly enable transaction support

template.setEnableTransactionSupport(true);

return template;

}

@Bean

public RedisConnectionFactory redisConnectionFactory() {

// jedis || Lettuce || srp || ...

}

@Bean

public PlatformTransactionManager transactionManager() throws SQLException {

return new DataSourceTransactionManager(dataSource());

}

@Bean

public DataSource dataSource() throws SQLException {

// ...

}

}

|  |  |
| --- | --- |
|  | Configures a Spring Context to enable [declarative transaction management](https://docs.spring.io/spring/docs/4.3.22.RELEASE/spring-framework-reference/data-access.html#transaction-declarative). |
|  | Configures RedisTemplate to participate in transactions by binding connections to the current thread. |
|  | Transaction management requires a PlatformTransactionManager. Spring Data Redis does not ship with a PlatformTransactionManager implementation. Assuming your application uses JDBC, Spring Data Redis can participate in transactions by using existing transaction managers. |

The following examples each demonstrate a usage constraint:

*Example 2. Usage Constraints*

// must be performed on thread-bound connection

template.opsForValue().set("thing1", "thing2");

// read operation must be executed on a free (not transaction-aware) connection

template.keys("\*");

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

template.opsForValue().get("thing1");

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 do not care about the results of the pipelined operations, you can use the standard execute method, passing true for the pipeline argument. The executePipelined methods run the provided RedisCallback or SessionCallback in a pipeline and return the results, as shown in the following 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 preceding example runs 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 preceding example are Strings. There are additional executePipelined methods that let you 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.

|  |  |
| --- | --- |
|  | As of version 1.1, an important change has been made to the exec methods of RedisConnection and RedisTemplate. 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, zAdd returns a boolean indicating whether 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 such as 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, 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 uses the Redis script cache.

Scripts can be run by calling the execute methods of RedisTemplate. It uses a configurable ScriptExecutor 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 through the key and value serializers of the template. There is an additional overload that lets you pass custom serializers for the script arguments and the 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.

The following example runs a common “check-and-set” scenario by using a Lua script. This is an ideal use case for a Redis script, as it requires that running a set of commands atomically, and the behavior of one command is influenced by the result of another.

@Bean

public RedisScript<Boolean> script() {

DefaultRedisScript<Boolean> redisScript = new DefaultRedisScript<Boolean>();

redisScript.setScriptSource(new ResourceScriptSource(new ClassPathResource("META-INF/scripts/checkandset.lua")));

redisScript.setResultType(Boolean.class);

}

public class Example {

@Autowired

RedisScript<Boolean> script;

public boolean checkAndSet(String expectedValue, String newValue) {

return redisTemplate.execute(script, Collections.singletonList("key"), 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 preceding code 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 a deserialized value type. It can also be null if the script returns a throw-away status (specifically, 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 runs the scripts. Scripts can be run within a SessionCallback as part of a transaction or pipeline. See “[Redis Transactions](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#tx)” and “[Pipelining](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#pipeline)” for more information.

The scripting support provided by Spring Data Redis also lets you schedule Redis scripts for periodic execution by using the Spring Task and Scheduler abstractions. See the [Spring Framework](https://projects.spring.io/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](https://download.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html) counters and JDK [Collections](https://download.oracle.com/javase/6/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. 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. The following example shows the configuration for a bean that uses a RedisList:

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

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

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

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

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

<bean id="queue" class="org.springframework.data.redis.support.collections.DefaultRedisList">

<constructor-arg ref="redisTemplate"/>

<constructor-arg value="queue-key"/>

</bean>

</beans>

The following example shows a Java configuration example for a Deque:

public class AnotherExample {

// injected

private Deque<String> queue;

public void addTag(String tag) {

queue.push(tag);

}

}

As shown in the preceding example, 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 be replaced with an in-memory one).

5.13.1. Support for the Spring Cache Abstraction

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

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

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

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

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

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

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

<!-- turn on declarative caching -->

<cache:annotation-driven />

<!-- declare Redis Cache Manager -->

<bean id="cacheManager" class="org.springframework.data.redis.cache.RedisCacheManager" c:template-ref="redisTemplate"/>

</beans>

|  |  |
| --- | --- |
|  | By default RedisCacheManager will lazily initialize RedisCache whenever a Cache is requested. This can be changed by predefining a Set of cache names. |
|  | By default RedisCacheManager will not participate in any ongoing transaction. Use setTransactionAware to enable transaction support. |

|  |  |
| --- | --- |
|  | By default RedisCacheManager does not prefix keys for cache regions, which can lead to an unexpected growth of a ZSET used to maintain known keys. It’s highly recommended to enable the usage of prefixes in order to avoid this unexpected growth and potential key clashes using more than one cache region. |
|  | By default RedisCache will not cache any null values as keys without a value get dropped by Redis itself. However you can explicitly enable null value caching via RedisCacheManager which will store org.springframework.cache.support.NullValue as a placeholder. |

6. Redis Cluster

Working with [Redis Cluster](http://redis.io/topics/cluster-spec) requires Redis Server version 3.0+. See the [Cluster Tutorial](http://redis.io/topics/cluster-tutorial) for more information.

|  |  |
| --- | --- |
|  | Redis Cluster is only supported by [jedis](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:connectors:jedis) and [lettuce](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:connectors:lettuce). |

6.1. Enabling Redis Cluster

Cluster support is based on the 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 instances are created with the RedisConnectionFactory, which has to be set up with the associated RedisClusterConfiguration, as shown in the following example:

*Example 3. 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 through PropertySource and has the following properties:  *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 are kept only in the native driver and are not written back to the configuration. |

6.2. Working With Redis Cluster Connection

As mentioned earlier, Redis Cluster behaves differently from single-node Redis or even a Sentinel-monitored master-slave environment. This is because the automatic sharding maps a key to one of 16384 slots, which are distributed across the nodes. Therefore, commands that involve more than one key must assert all keys map to the exact same slot to avoid cross-slot execution errors. A single cluster node serves only a dedicated set of keys. Commands issued against one particular server return results only for those keys served by that server. As a simple example, consider the KEYS command. When issued to a server in a cluster environment, it returns only 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 a cluster environment, you must read the keys from all the known master nodes.

While redirects for specific keys to the corresponding slot-serving node are handled by the driver libraries, higher-level functions, such as collecting information across nodes or sending commands to all nodes in the cluster, are covered by RedisClusterConnection. Picking up the keys example from earlier, this means that the keys(pattern) method picks up every master node in the cluster and simultaneously executes the KEYS command on every master node 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 methods (for example, keys(node, pattern)).

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

The following example shows a set of commands being run across the cluster:

*Example 4. 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("thing1", value);

connection.set("thing2", 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 replicants of the 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 → [thing1, thing2] |
|  | Request routed to node at 7379 → [thing2] |
|  | Request routed to node at 7380 → [] |
|  | Request routed to node at 7381 → [thing1] |
|  | Request routed to node at 7382 → [thing2] |

When all keys map to the same slot, the native driver library automatically serves cross-slot requests, such as MGET. However, once this is not the case, RedisClusterConnection executes multiple parallel GET commands against the slot-serving nodes and again returns an accumulated result. This is less performant than the single-slot execution and, therefore, should be used with care. If in doubt, consider pinning keys to the same slot by providing a prefix in curly brackets, such as {my-prefix}.thing1 and {my-prefix}.thing2, which will both map to the same slot number. The following example shows cross-slot request handling:

*Example 5. 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("thing1", value); // slot: 12182

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

connection.set("thing2", value); // slot: 5461

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

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

|  |  |
| --- | --- |
|  | Same Configuration as in the sample before. |
|  | Keys map to same slot → 127.0.0.1:7381 MGET thing1 {thing1}.thing2 |
|  | Keys map to different slots and get split up into single slot ones routed to the according nodes → 127.0.0.1:7379 GET thing2 → 127.0.0.1:7381 GET thing1 |
|  | The preceding examples demonstrate the general strategy followed by Spring Data Redis. Be aware that some operations might require loading huge amounts of data into memory to compute the desired command. Additionally, not all cross-slot requests can safely be ported to multiple single slot requests and error if misused (for example, PFCOUNT). |

6.3. Working with RedisTemplate and ClusterOperations

See the [Working with Objects through RedisTemplate](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis:template) section for information about the general purpose, configuration, and usage of RedisTemplate.

|  |  |
| --- | --- |
|  | 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 through the ClusterOperations interface, which can be obtained from RedisTemplate.opsForCluster(). This lets you explicitly run commands on a single node within the cluster while retaining the serialization and deserialization features configured for the template. It also provides administrative commands (such as CLUSTER MEET) or more high-level operations (for example, resharding).

The following example shows how to access RedisClusterConnection with RedisTemplate:

*Example 6. Accessing RedisClusterConnection with 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. |

7. Redis Repositories- Redis存储库

Working with Redis Repositories lets you seamlessly convert and store domain objects in Redis Hashes, apply custom mapping strategies, and use secondary indexes.

使用Redis存储库可以在Redis Hashes中无缝转换和存储域对象，应用自定义映射策略以及使用二级索引。

Redis Repositories require at least Redis Server version 2.8.0 and do not work with transactions. Make sure to use a RedisTemplate with [disabled transaction support](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#tx.spring).

Redis存储库至少需要Redis Server版本2.8.0，不能用于事务。 确保使用具有禁用事务支持的RedisTemplate。

7.1. Usage-用法

Spring Data Redis lets you easily implement domain entities, as shown in the following example:

Spring Data Redis允许您轻松实现域实体，如以下示例所示：

*Example 7. Sample Person Entity*

*示例7.示例人员实体*

@RedisHash("people")

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 @RedisHash annotation on its type and a property named idthat is annotated with org.springframework.data.annotation.Id. Those two items are responsible for creating the actual key used to persist the hash.

我们这里有一个非常简单的域对象。 请注意，它在其类型上有一个@RedisHash注释，并且使用org.springframework.data.annotation.Id注释了一个名为idthat的属性。 这两个项目负责创建用于保持哈希的实际密钥。

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

使用@Id注释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受青睐。

To now actually have a component responsible for storage and retrieval, we need to define a repository interface, as shown in the following example:

现在实际上有一个负责存储和检索的组件，我们需要定义一个存储库接口，如下例所示：

*Example 8. Basic Repository Interface To Persist Person Entities*

*示例8.持久保存人员实体的基本存储库接口*

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 corresponding Spring configuration, shown in the following example:

由于我们的存储库扩展了CrudRepository，它提供了基本的CRUD和finder操作。 我们需要将事物粘合在一起的是相应的Spring配置，如下例所示：

*Example 9. JavaConfig for Redis Repositories*

*示例9. Redis存储库的JavaConfig*

@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 preceding setup, we can inject PersonRepository into our components, as shown in the following example:

鉴于前面的设置，我们可以将PersonRepository注入到我们的组件中，如以下示例所示：

*Example 10. Access to Person Entities*

*示例10.访问人员实体*

@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 the current value is null or reuses an already set id value and stores properties of type Person inside the Redis Hash with a key that has a pattern of keyspace:id — in this case, it might be people:5d67b7e1-8640-4475-beeb-c666fab4c0e5.

如果当前值为null或重新使用已设置的id值，则生成一个新的id，并使用具有keyspace模式的键在Redis Hash中存储Person类型的属性：id - 在这种情况下，它可能是people：5d67b7e1-8640-4475-BEEB-c666fab4c0e5。

Uses the provided id to retrieve the object stored at keyspace:id.

使用提供的id来检索存储在keyspace：id的对象。

Counts the total number of entities available within the keyspace, people, defined by @RedisHash on Person.

计算密钥空间中可用的实体总数，由@RedisHash on Person定义的人员。

Removes the key for the given object from Redis.

从Redis中删除给定对象的键。

7.2. Object-to-Hash Mapping-对象到哈希映射

The Redis Repository support persists Objects to 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[].

Redis存储库支持将对象持久化为哈希。 这需要一个由RedisConverter完成的Object-to-Hash转换。 默认实现使用Converter将属性值映射到Redis本地byte []。

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

给定前面部分中的Person类型，默认映射如下所示：

\_class = org.example.Person

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

firstname = rand

lastname = al’thor

address.city = emond's field

address.country = andor

The \_class attribute is included on the root level as well as on any nested interface or abstract types.

\_class属性包含在根级别以及任何嵌套接口或抽象类型上。

Simple property values are mapped by path.

简单属性值由路径映射。

Properties of complex types are mapped by their dot path.

复杂类型的属性由其点路径映射。

The following table describes the default mapping rules:

下表描述了默认的映射规则：

| *Table 4. Default Mapping Rules*  *表4.默认映射规则* | | |
| --- | --- | --- |
| **Type** | **Sample** | **Mapped Value** |
| Simple Type (for example, String) | String firstname = "rand"; | firstname = "rand" |
| Complex Type (for example, Address) | Address address = 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 = "…​ |

Due to the flat representation structure, Map keys need to be simple types, such as String or Number.

由于平面表示结构，Map键必须是简单类型，例如String或Number。

Mapping behavior can be customized by registering the corresponding Converter in RedisCustomConversions. Those converters can take care of converting from and to a single byte[] as well as Map<String,byte[]>. The first one is suitable for (for example) converting a complex type to (for example) a binary JSON representation that still uses the default mappings hash structure. The second option offers full control over the resulting hash.

可以通过在RedisCustomConversions中注册相应的Converter来自定义映射行为。 这些转换器可以负责转换单个字节[]以及Map <String，byte []>。 第一个适用于（例如）将复杂类型转换为（例如）仍使用默认映射哈希结构的二进制JSON表示。 第二个选项提供对结果哈希的完全控制。

Writing objects to a Redis hash deletes the content from the hash and re-creates the whole hash, so data that has not been mapped is lost

将对象写入Redis哈希会从哈希中删除内容并重新创建整个哈希，因此未映射的数据将丢失

The following example shows two sample byte array converters:

以下示例显示了两个示例字节数组转换器：

*Example 11. Sample byte[] Converters*

*示例11.示例byte []转换器*

@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 preceding byte array Converter produces output similar to the following:

使用前面的字节数组Converter生成类似于以下内容的输出：

\_class = org.example.Person

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

firstname = rand

lastname = al’thor

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

The following example shows two examples of Map converters:

以下示例显示了两个Map转换器示例：

*Example 12. Sample Map<String,byte[]> Converters*

*示例12.示例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 preceding Map Converter produces output similar to the following:

使用前面的Map Converter生成类似于以下内容的输出：

\_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/1.8.18.RELEASE/reference/html/#redis.repositories.indexes) are still created, even for custom converted types.

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，仍会创建辅助索引。

7.2.1. Customizing Type Mapping

If you want to avoid writing the entire Java class name as type information and would rather like to use a key, you can use the @TypeAlias annotation on the entity class being persisted. If you need to customize the mapping even more, look at the [TypeInformationMapper](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/convert/TypeInformationMapper.html) interface. An instance of that interface can be configured at the DefaultRedisTypeMapper, which can be configured on MappingRedisConverter.

The following example shows how to define a type alias for an entity:

*Example 13. Defining @TypeAlias for an entity*

@TypeAlias("pers")

class Person {

}

The resulting document contains pers as the value in a \_class field.

Configuring Custom Type Mapping

The following example demonstrates how to configure a custom RedisTypeMapper in MappingRedisConverter:

*Example 14. Configuring a custom RedisTypeMapper via Spring Java Config*

class CustomRedisTypeMapper extends DefaultRedisTypeMapper {

//implement custom type mapping here

}

@Configuration

class SampleRedisConfiguration {

@Bean

public MappingRedisConverter redisConverter(RedisMappingContext mappingContext,

RedisCustomConversions customConversions, ReferenceResolver referenceResolver) {

MappingRedisConverter mappingRedisConverter = new MappingRedisConverter(mappingContext, null, referenceResolver,

customTypeMapper());

mappingRedisConverter.setCustomConversions(customConversions);

return mappingRedisConverter;

}

@Bean

public RedisTypeMapper customTypeMapper() {

return new CustomRedisTypeMapper();

}

}

7.3. Keyspaces

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

The following example shows how to set the keyspace configuration with the @EnableRedisRepositories annotation:

*Example 15. 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, "people"));

}

}

}

The following example shows how to programmatically set the keyspace:

*Example 16. 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, "people"));

}

}

}

7.4. 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/1.8.18.RELEASE/reference/html/#redis.repositories.expirations).

7.4.1. Simple Property Index

Given the sample Person entity shown earlier, we can create an index for firstname by annotating the property with @Indexed, as shown in the following example:

*Example 17. Annotation driven indexing*

@RedisHash("people")

public class Person {

@Id String id;

@Indexed String firstname;

String lastname;

Address address;

}

Indexes are built up for actual property values. Saving two Persons (for example, "rand" and "aviendha") results in setting up indexes similar to the following:

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

SADD people: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, as shown in the following example:

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

Furthermore, the programmatic setup lets you define indexes on map keys and list properties, as shown in the following example:

@RedisHash("people")

public class Person {

// ... other properties omitted

Map<String,String> attributes;

Map<String Person> relatives;

List<Address> addresses;

}

|  |  |
| --- | --- |
|  | SADD people:attributes.map-key:map-value e2c7dcee-b8cd-4424-883e-736ce564363e |
|  | SADD people:relatives.map-key.firstname:tam e2c7dcee-b8cd-4424-883e-736ce564363e |
|  | SADD people:addresses.city:tear e2c7dcee-b8cd-4424-883e-736ce564363e |
|  | Indexes cannot be resolved on [References](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.references). |

As with keyspaces, you can configure indexes without needing to annotate the actual domain type, as shown in the following example:

*Example 18. Index Setup with @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("people", "firstname"));

}

}

}

Again, as with keyspaces, you can programmatically configure indexes, as shown in the following example:

*Example 19. 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("people", "firstname"));

}

}

}

7.4.2. Geospatial Index

Assume the Address type contains a location property of type Point that holds the geo coordinates of the particular address. By annotating the property with @GeoIndexed, Spring Data Redis adds those values by using Redis GEO commands, as shown in the following example:

@RedisHash("people")

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

In the preceding example the, longitude and latitude values are stored by using GEOADD that use the object’s 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 and within with other criteria. |

7.5. Time To Live

Objects stored in Redis may be valid only 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 reach their end of life. The expiration time in seconds can be set with @RedisHash(timeToLive=…​) as well as by using KeyspaceSettings (see [Keyspaces](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#redis.repositories.keyspaces)).

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

*Example 20. 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 reads back the actual TTL or PTTL value from Redis. -1 indicates that the object has no associated expiration. |

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 corresponding EXPIRE command is executed. In addition to persisting the original, a phantom copy is persisted in Redis and set to expire five minutes after the original one. This is done to enable the Repository support to publish RedisKeyExpiredEvent, holding the expired value in Spring’s ApplicationEventPublisherwhenever a key expires, even though the original values have already been removed. Expiry events are received on all connected applications that use 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](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/api/org/springframework/data/redis/core/RedisKeyValueAdapter.EnableKeyspaceEvents.html) for possible values.

The RedisKeyExpiredEvent holds a copy of the expired domain object as well as the key.

|  |  |
| --- | --- |
|  | Delaying or disabling the expiry event listener startup impacts RedisKeyExpiredEvent publishing. A disabled event listener does not publish expiry events. A delayed startup can cause loss of events because of the delayed listener initialization. |
|  | The keyspace notification message listener alters notify-keyspace-events settings in Redis, if those are not already set. Existing settings are not overridden, so you must set up those settings correctly (or leave them empty). 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 is not processed, which may lead to secondary indexes containing references to the expired object. |

7.6. 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, as shown in the following example:

*Example 21. Sample Property Reference*

\_class = org.example.Person

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

firstname = rand

lastname = al’thor

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

|  |  |
| --- | --- |
|  | Reference stores the whole key (keyspace:id) of the referenced object. |
|  | Referenced Objects are not persisted when the referencing object is saved. You must persist changes on referenced objects separately, since only the reference is stored. Indexes set on properties of referenced types are not resolved. |

7.7. Persisting Partial Updates

In some cases, you need not load and rewrite the entire entity just to set a new value within it. A session timestamp for the last active time might be such a scenario where you want to alter one property. PartialUpdate lets you define set and deleteactions on existing objects while taking care of updating potential expiration times of both the entity itself and index structures. The following example shows a partial update:

*Example 22. 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 firstname property to mat. |
|  | Set the simple 'address.city' property to 'emond’s field' without having to pass in the entire object. This does not work when a custom conversion is registered. |
|  | Remove the age property. |
|  | Set complex address property. |
|  | Set a map of values, which removes the previously existing map 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/1.8.18.RELEASE/reference/html/#redis.repositories.expirations). |
|  | Updating complex objects as well as map (or other collection) structures requires further interaction with Redis to determine existing values, which means that rewriting the entire entity might be faster. |

7.8. Queries and Query Methods-查询和查询方法

Query methods allow automatic derivation of simple finder queries from the method name, as shown in the following example:

查询方法允许从方法名称自动派生简单的查找程序查询，如以下示例所示：

*Example 23. Sample Repository finder Method*

*示例23.样本存储库查找器方法*

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.

请确保在finder方法中使用的属性已设置为索引。

Query methods for Redis repositories support only queries for entities and collections of entities with paging.

Redis存储库的查询方法仅支持对具有分页的实体和实体集合的查询。

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 indexes. To do so, provide a RedisCallback that returns a single or Iterable set of id values, as shown in the following example:

使用派生查询方法可能并不总是足以模拟要执行的查询。 RedisCallback提供了对索引结构甚至自定义索引的实际匹配的更多控制。 为此，请提供RedisCallback，它返回单个或Iterable的一组id值，如以下示例所示：

*Example 24. Sample finder using RedisCallback*

*示例24.使用RedisCallback的样本查找器*

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);

The following table provides an overview of the keywords supported for Redis and what a method containing that keyword essentially translates to:

下表概述了Redis支持的关键字以及包含该关键字的方法实质上转换为：

| *Table 5. Supported keywords inside method names*  *表5.方法名称中支持的关键字* | | |
| --- | --- | --- |
| **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 |  |

7.9. Redis Repositories Running on a Cluster- Redis存储库在群集上运行

You can use the Redis repository support in a clustered Redis environment. See the “[Redis Cluster](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#cluster)” section for ConnectionFactory configuration details. Still, some additional configuration must be done, because the default key distribution spreads entities and secondary indexes through out the whole cluster and its slots.

The following table shows the details of data on a cluster (based on previous examples):

| **Key** | **Type** | **Slot** | **Node** |
| --- | --- | --- | --- |
| people:e2c7dcee-b8cd-4424-883e-736ce564363e | id for hash | 15171 | 127.0.0.1:7381 |
| people:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56 | id for hash | 7373 | 127.0.0.1:7380 |
| people:firstname:rand | index | 1700 | 127.0.0.1:7379 |

Some commands (such as 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 is useful to pin keyspaces to a single slot, which lets make use of Redis server side computation right away. The following table shows what happens when you do (note the change in the slot column and the port value in the node column):

| **Key** | | **Type** | **Slot** | **Node** |
| --- | --- | --- | --- | --- |
| {people}:e2c7dcee-b8cd-4424-883e-736ce564363e | | id for hash | 2399 | 127.0.0.1:7379 |
| {people}:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56 | | id for hash | 2399 | 127.0.0.1:7379 |
| {people}:firstname:rand | | index | 2399 | 127.0.0.1:7379 |
|  | Define and pin keyspaces by using @RedisHash("{yourkeyspace}") to specific slots when you use Redis cluster. | | | | |

7.10. CDI Integration

Instances of the repository interfaces are usually created by a container, for which Spring is the most natural choice when working with Spring Data. Spring offers sophisticated for creating bean instances. Spring Data Redis ships with a custom CDI extension that lets you use the repository abstraction in CDI environments. The extension is part of the JAR, so, to activate it, drop the Spring Data Redis JAR into your classpath.

You can then set up the infrastructure by implementing a CDI Producer for the RedisConnectionFactory and RedisOperations, as shown in the following example:

class RedisOperationsProducer {

@Produces

RedisConnectionFactory redisConnectionFactory() {

JedisConnectionFactory jedisConnectionFactory = new JedisConnectionFactory();

jedisConnectionFactory.setHostName("localhost");

jedisConnectionFactory.setPort(6379);

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 your JavaEE environment.

The Spring Data Redis CDI extension picks up all available repositories as CDI beans and creates 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, as shown in the following example:

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.

7.11. Redis Repositories Anatomy- Redis存储库解剖

Redis as a store itself offers a very narrow low-level API leaving higher level functions, such as secondary indexes and query operations, up to the user.

This section provides a more detailed view of commands issued by the repository abstraction for a better understanding of potential performance implications.

Consider the following entity class as the starting point for all operations:

*Example 25. Example entity*

@RedisHash("people")

public class Person {

@Id String id;

@Indexed String firstname;

String lastname;

Address hometown;

}

public class Address {

@GeoIndexed Point location;

}

7.11.1. Insert new

repository.save(new Person("rand", "al'thor"));

HMSET "people:19315449-cda2-4f5c-b696-9cb8018fa1f9" "\_class" "Person" "id" "19315449-cda2-4f5c-b696-9cb8018fa1f9" "firstname" "rand" "lastname" "al'thor"

SADD "people" "19315449-cda2-4f5c-b696-9cb8018fa1f9"

SADD "people:firstname:rand" "19315449-cda2-4f5c-b696-9cb8018fa1f9"

SADD "people:19315449-cda2-4f5c-b696-9cb8018fa1f9:idx" "people:firstname:rand"

|  |  |
| --- | --- |
|  | Save the flattened entry as hash. |
|  | Add the key of the hash written in <1> to the helper index of entities in the same keyspace. |
|  | Add the key of the hash written in <2> to the secondary index of firstnames with the properties value. |
|  | Add the index of <3> to the set of helper structures for entry to keep track of indexes to clean on delete/update. |

7.11.2. Replace existing

repository.save(new Person("e82908cf-e7d3-47c2-9eec-b4e0967ad0c9", "Dragon Reborn", "al'thor"));

DEL "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"

HMSET "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9" "\_class" "Person" "id" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9" "firstname" "Dragon Reborn" "lastname" "al'thor"

SADD "people" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"

SMEMBERS "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9:idx"

TYPE "people:firstname:rand"

SREM "people:firstname:rand" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"

DEL "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9:idx"

SADD "people:firstname:Dragon Reborn" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"

SADD "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9:idx" "people:firstname:Dragon Reborn"

|  |  |
| --- | --- |
|  | Remove the existing hash to avoid leftovers of hash keys potentially no longer present. |
|  | Save the flattened entry as hash. |
|  | Add the key of the hash written in <1> to the helper index of entities in the same keyspace. |
|  | Get existing index structures that might need to be updated. |
|  | Check if the index exists and what type it is (text, geo, …). |
|  | Remove a potentially existing key from the index. |
|  | Remove the helper holding index information. |
|  | Add the key of the hash added in <2> to the secondary index of firstnames with the properties value. |
|  | Add the index of <6> to the set of helper structures for entry to keep track of indexes to clean on delete/update. |

7.11.3. Save Geo Data

Geo indexes follow the same rules as normal text based ones but use geo structure to store values. Saving an entity that uses a Geo-indexed property results in the following commands:

GEOADD "people:hometown:location" "13.361389" "38.115556" "76900e94-b057-44bc-abcf-8126d51a621b"

SADD "people:76900e94-b057-44bc-abcf-8126d51a621b:idx" "people:hometown:location"

|  |  |
| --- | --- |
|  | Add the key of the saved entry to the the geo index. |
|  | Keep track of the index structure. |

7.11.4. Find using simple index

repository.findByFirstname("egwene");

SINTER "people:firstname:egwene"

HGETALL "people:d70091b5-0b9a-4c0a-9551-519e61bc9ef3"

HGETALL ...

|  |  |
| --- | --- |
|  | Fetch keys contained in the secondary index. |
|  | Fetch each key returned by <1> individually. |

7.11.5. Find using Geo Index

repository.findByHometownLocationNear(new Point(15, 37), new Distance(200, KILOMETERS));

GEORADIUS "people:hometown:location" "15.0" "37.0" "200.0" "km"

HGETALL "people:76900e94-b057-44bc-abcf-8126d51a621b"

HGETALL ...

|  |  |
| --- | --- |
|  | Fetch keys contained in the secondary index. |
|  | Fetch each key returned by <1> individually. |

Appendixes

Appendix Document Structure

The appendix contains various additional detail that complements the information in the rest of the reference documentation:

* “[Schema](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#appendix:schema)” defines the schemas provided by Spring Data Redis.
* “[Command Reference](https://docs.spring.io/spring-data/redis/docs/1.8.18.RELEASE/reference/html/#appendix:command-reference)” details which commands are supported by RedisTemplate.

Appendix A: Schema

[Spring Data Redis Schema (redis-namespace)](http://www.springframework.org/schema/redis/spring-redis-1.0.xsd)

Appendix B: Command Reference

Supported Commands

| *Table 6. 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 1.8.18.RELEASE