Spring Data Redis

Costin Leau, Jennifer Hickey, Christoph Strobl, Thomas Darimont, Mark Paluch, Jay Bryant

Version 2.3.6.RELEASE, 2021-05-14

Table of Contents

□□□□□□□□□□□□□□□,□□□□□□□□□□,□□□□□□□□□□□□□□□□□□□,□□□□□□□□□□□□□□□□□□□(□□□□□□□□□□□□□□□□) .

前言

Spring Data Redis 项目通过使用 key-value 风格的数据存储将 Spring 的核心概念应用于开发解决方案。我们提供了一个 发送和接收消息的 "模板” 作为高级抽象。您可能会注意到与 Spring Framework 中的 JDBC 支持相似。

本文档是 Spring Data Redis(SDR) 模块支持的参考指南。

Chapter 1. 学习 Spring

Spring Data 使用 Spring 框架的核心功能，包括：

- IoC 容器
- 类型转换系统
- 语言表达式
- JMX 集成
- DAO 异常层次.

虽然了解 Spring API 并不重要,但了解它们背后的概念很重要. 至少,应该了解 IoC 背后的想法,并且您应该熟悉选择使用的任何IoC容器.

Redis 支持的核心功能可以直接使用，而无需调用 Spring 容器的 IoC 服务. 这非常类似于 `JdbcTemplate`，它可以在没有 Spring 容器的任何其他服务的情况下 "独立" 使用. 为了利用 Spring Data Redis 的所有功能，例如存储库支持，您需要配置库的某些部分以使用 Spring.

话虽如此,您对 Spring 的了解越多,您就能更快地获取 Spring Data Redis. 可以参考 Spring 框架的参考文档,关于该主题有很多文章，博客和书籍. 有关更多信息，请参见 Spring 框架 home page.

通常,这应该是想要尝试使用 Spring Data Redis 的开发人员的起点.

Chapter 2. 学习 NoSQL 和 Key Value 存储

NoSQL 存储席卷了存储世界. 它是一个广阔的领域,具有许多解决方案,术语和模式(更糟糕的是,甚至术语本身也具有 多种含义). 尽管某些原则很普遍,但一定程度上要熟悉 SDR 支持的处处,这一点至关重要. 熟悉这些解决方案的最佳方法是阅读其文档并遵循其示例. 通常需要花费五到十分钟的时间来完成这些步骤,而且,如果您具有 RDMBS 的背景,那么这些练习通常会使人大开眼界.

2.1. 尝试示例

您可以在 Spring Data 示例存储库中找到各种用于键值存储的示例,网址为 https://github.com/spring-projects/spring-data-keyvalue-examples. 对于 Spring Data Redis,您应该特别注意 `retwisj` 示例,该示例是在 Redis 之上构建的 Twitter-clone,可以在本地运行或部署到云中. 有关更多信息,请参阅其 文档,以下 博客.

Chapter 3. 要求

Spring Data Redis 2.x 需要 JDK 8.0 或更高版本以及 Spring Framework {springVersion} 或更高版本.

就 key-value 存储而言, 需要 Redis 2.6.x 或更高版本. Spring Data Redis 当前已针对最新的 4.0 版本进行了测试.

Chapter 4. 其他帮助资源

学习新框架并不总是那么简单。在本节中，我们提供开始学习 Spring Data Redis 的简要指南。但是，如果遇到问题或需要建议，请随时使用以下链接之一：

社区论坛

Stack Overflow 上的 Spring Data 标签是一个留言板,供所有 Spring Data(不仅仅是文档)用户共享信息并互相帮助。请注意,只有注册了才能发布。

Professional Support

可从 Spring Data 和 Spring 背后的公司 Pivotal Software, Inc. 获得专业的支持。

Chapter 5. 后续发展

有关 Spring Data 源代码存储库,每夜构建和 snapshot artifacts 的信息,请参见 Spring Data 主页.

通过在 spring-data 或 spring-data-redis 上的 Stack Overflow 上与开发人员进行交互,可以帮助使 Spring Data 最好地满足 Spring 社区的需求.

如果您遇到错误或要提出改进建议(包括本文档中的内容),请在 Spring Data 问题跟踪器上创建 tracker.

要了解 Spring 生态系统中的最新新闻和公告,请订阅 Spring Community Portal.

最后,您可以在 Twitter 上关注 Spring blog 或项目团队(@SpringData).

Chapter 6. 新特性

本节简要介绍了最新版本中的新内容和值得注意的内容.

6.1. New in Spring Data Redis 2.3

- Template API 方法优化 `Duration` 和 `Instant`.
- Stream Commands 扩展.

6.2. Spring Data Redis 2.2

- Redis Streams
- 接受单个 keys 集合的 `union`/`diff`/`intersect` 操作方法
- 更新至 Jedis 3.
- 添加对使用 Jedis Cluster 的脚本命令的支持.

6.3. Spring Data Redis 2.1

- 使用 Lettuce 的 Unix domain socket 连接
- Lettuce 从主库写,从库读
- 查询示例
- `@TypeAlias` 支持 Redis 存储库.
- Cluster-wide `SCAN` using Lettuce and `SCAN` execution on a selected node supported by both drivers.
- Reactive Pub/Sub to send and receive a message stream.
- `BITFIELD`, `BITPOS`, and `OBJECT` command support.
- Align return types of `BoundZSetOperations` with `ZSetOperations`.
- Reactive `SCAN`, `HSCAN`, `SSCAN`, and `ZSCAN` support.
- Usage of `IsTrue` and `IsFalse` keywords in repository query methods.

6.4. New in Spring Data Redis 2.0

- Upgrade to Java 8.
- Upgrade to Lettuce 5.0.
- Removed support for SRP and JRedis drivers.
- Reactive connection support using Lettuce.
- Introduce Redis feature-specific interfaces for `RedisConnection`.
- Improved `RedisConnectionFactory` configuration with `JedisClientConfiguration` and `LettuceClientConfiguration`.

- Revised `RedisCache` implementation.
- Add `SPOP` with count command for Redis 3.2.

6.5. New in Spring Data Redis 1.8

- Upgrade to Jedis 2.9.
- Upgrade to `Lettuce` 4.2 (Note: Lettuce 4.2 requires Java 8).
- Support for Redis GEO commands.
- Support for Geospatial Indexes using Spring Data Repository abstractions (see □□□□□□).
- `MappingRedisConverter`-based `HashMapper` implementation (see Hash □□).
- Support for `PartialUpdate` in repositories (see □□□□□□).
- SSL support for connections to Redis cluster.
- Support for `client name` through `ConnectionFactory` when using Jedis.

6.6. New in Spring Data Redis 1.7

- Support for RedisCluster.
- Support for Spring Data Repository abstractions (see Redis □□□).

6.7. New in Spring Data Redis 1.6

- The `Lettuce` Redis driver switched from wg/lettuce to 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.

6.8. New in Spring Data Redis 1.5

- Add support for Redis HyperLogLog commands: `PFADD`, `PFCOUNT`, and `PFMERGE`.
- Configurable `JavaType` lookup for Jackson-based `RedisSerializers`.
- `PropertySource`-based configuration for connecting to Redis Sentinel (see: Redis Sentinel □□).

Chapter 7. 依赖

由于各个 Spring Data 模块的初始日期不同,它们中的大多数都带有不同的主版本号和次版本号. 寻找兼容版本的最简单方法是依靠我们随定义的兼容版本提供的 Spring Data Release BOM. 在 Maven 项目中,您将在 `<dependencyManagement />` POM 的部分声明这种依赖,如下所示:

Example 1. 使用 Spring Data 发行版 BOM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-releasetrain</artifactId>
      <version>{releasetrainVersion}</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

目前的发行版本是 `{releasetrainVersion}`. 名称按字母顺序上升,目前可用的 列车在 此处 列出. 版本名称遵循以下模式: `${name}-${release}`,其中发布可以是下列之一:

- `BUILD-SNAPSHOT`: 当前快照
- `M1`, `M2`, 等: 里程碑
- `RC1`, `RC2`, 发布候选
- `RELEASE`: GA 版本
- `SR1`, `SR2`, 等: 服务版本

在我们的 Spring Data 示例存储库 中可以找到使用 BOM 的一个工作示例. 有了这个,你可以在你的模块中声明 Spring Data 模块而不需要版本 `<dependencies />`,如下所示:

Example 2. 声明一个依赖 Spring Data 模块

```
<dependencies>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
  </dependency>
<dependencies>
```

7.1. Spring Boot 的依赖管理

Spring Boot 为您选择最新版本的 Spring Data 模块. 如果您仍想升级到较新版本,请将该属性配置为您要使用 `spring-data-releasetrain.version` 的

火车名称迭代 .

7.2. Spring Framework

当前版本的 Spring Data 模块需要版本 5.2.12.RELEASE 或更高版本的 Spring Framework. 这些模块也可能与该次要版本的旧版错误修复版本一起工作. 但是,强烈建议使用该代中的最新版本.

参考文档

文档结构

本文档是 Spring Data Redis(SDR) 提供的核心功能. 它解释了 Key-Value 模块的概念和语义以及各种 stores namespaces 的语法.

有关 Key-Value 存储,Spring 或 Spring Data 示例的介绍, 请查看 [get-started]. 本文档仅涉及 Spring Data Redis 支持,并且你熟悉 Key-Value 存储和 Spring 相关概念.

Redis 支持 介绍 Redis 模块功能.

“Redis 存储库” 介绍了 Redis 的存储库支持.

本文档是 Spring Data Redis (SDR) 支持的参考指南.

Chapter 8. 为什么使用 Spring Data Redis?

Spring 框架是领先的全栈 Java/JEE 应用程序框架. 它提供了一个轻量级的容器和一个非侵入性编程模型,该模型通过使用依赖注入,AOP和可移植服务抽象来实现.

NoSQL 存储系统提供了传统 RDBMS 的替代产品,以实现水平可伸缩性和速度. 就实现而言,键值存储是 NoSQL 中使用最多的类型之一.

Spring Data Redis (SDR) 框架由 Spring 基础架构支持,消除了与存储交互的所需的冗余任务和样板代码,从而使编写使用 Redis 键值存储的 Spring 应用程序变得容易.

Chapter 9. Redis 支持

Spring Data 支持的 key-value 存储之一是 Redis. 下面这句话引用自 Redis 项目主页:

> Redis 是高级键值存储. 它类似于 memcached,但数据集是不易失的,值可以是字符串,就像在 memcached 中一样,也可以是列表,集合和有序集合. 所有这些数据类型都可以通过原子操作进行操作,以 push/pop 元素, add/remove 元素,包括 union,intersection,集合之间的差异等. Redis 支持不同种类的排序功能.

Spring Data Redis 提供了简单的配置,并可以从 Spring 应用程序访问 Redis. 它提供了与 store 交互的低层和高层抽象,使用户摆脱了基础配置的困扰.

9.1. 开始使用

设置工作环境的一种简单方法是在 STS 中创建一个基于 Spring 的项目.

首先, 您需要设置一个正在运行的 Redis 服务器.

在 STS 中创建 Spring 项目:

1. File → New → Spring Template Project → Simple Spring Utility Project, 然后在出现提示时按 Yes. 然后输入项目和包名称, 例如 `org.spring.redis.example`. .将以下内容添加到 pom.xml 文件 `dependencies` 元素中:

```
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>2.3.6.RELEASE</version>
  </dependency>

</dependencies>
```

2. 在 pom.xml 中将 Spring 的版本更改为

```
<spring.framework.version>{springVersion}</spring.framework.version>
```

3. 将 Spring Milestone 存储库添加到 `pom.xml` 文件中, 与 `<dependencies/>` 元素处于同一级别:

```
<repositories>
  <repository>
    <id>spring-milestone</id>
    <name>Spring Maven MILESTONE Repository</name>
    <url>https://repo.spring.io/libs-milestone</url>
  </repository>
</repositories>
```

也可以在这里 预览此存储库.

9.2. Redis 要求

Spring Redis 需要 Redis 2.6 或更高版本,Spring Data Redis 与 Lettuce 和 Jedis 集成,这是两个流行的 Redis 开源 Java 库.

9.3. Redis 支持高级视图

Redis 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,您可以随时在视图之间移动. 例如,您可以获得底层连接(甚至本地库)以直接与 Redis 通信.

9.4. 连接到 Redis

使用 Redis 和 Spring 时的首要任务之一是通过IoC容器连接到 store. 为此,需要 Java 连接器(或绑定). 无论选择哪种库,您都只需要使用一组 Spring Data Redis API(在所有连接器上都保持一致): 使用 `org.springframework.data.redis.connection` 包下的 `RedisConnection` 和 `RedisConnectionFactory` 接口即可自动检索与 Redis 的活动连接.

9.4.1. RedisConnection 和 RedisConnectionFactory

`RedisConnection` 提供了 Redis 通信的核心模块,因为它处理与 Redis 服务端的通信. 它还会自动将底层连接库异常转换为 Spring 的 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.

对于需要本地库API的特殊情况,`RedisConnection` 提供了专用方法(`getNativeConnection`),该方法返回用于通信的原始底层对象.

激活的 `RedisConnection` 对象是通过 `RedisConnectionFactory` 创建的. 另外,工厂还充当 `PersistenceExceptionTranslator` 对象,这意味着一旦声明,它们就可以让您进行透明的异常转换. 例如,您可以使用 `@Repository` 注解和AOP进行异常转换. 有关更多信息,请参见 Spring Framework 文档中的相关部分.

根据底层配置,工厂可以返回新连接或现有连接(使用池或共享本地连接时).

使用 `RedisConnectionFactory` 的最简单方法是通过IoC容器配置适当的连接器,并将其注入 using 类.

不幸的是,当前,并非所有连接器都支持所有 Redis 功能. 在底层库不支持的 Connection API 上调用方法时,将抛出 `UnsupportedOperationException`.

Table 1. Feature Availability across Redis Connectors

Supported Feature	**Lettuce**	**Jedis**
Standalone Connections	X	X
Master/Replica Connections	X	

Supported Feature	Lettuce	Jedis
Redis Sentinel	Master Lookup, Sentinel Authentication, Replica Reads	Master Lookup
Redis Cluster	Cluster Connections, Cluster Node Connections, Replica Reads	Cluster Connections, Cluster Node Connections
Transport Channels	TCP, OS-native TCP (epoll, kqueue), Unix Domain Sockets	TCP
Connection Pooling	X (using `commons-pool2`)	X (using `commons-pool2`)
Other Connection Features	Singleton-connection sharing for non-blocking commands	`JedisShardInfo` support
SSL Support	X	X
Pub/Sub	X	X
Pipelining	X	X
Transactions	X	X
Datatype support	Key, String, List, Set, Sorted Set, Hash, Server, Stream, Scripting, Geo, HyperLogLog	Key, String, List, Set, Sorted Set, Hash, Server, Scripting, Geo, HyperLogLog
Reactive (non-blocking) API	X	

9.4.2. 配置 Lettuce 连接器

Lettuce 是一个基于 Netty 的开源连接器.Spring Data Redis 通过 `org.springframework.data.redis.connection.lettuce` 包提供支持. 以下示例显示了如何创建新的 Lettuce 连接工厂:

Add the following to the pom.xml files `dependencies` element:

```
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>io.lettuce</groupId>
    <artifactId>lettuce-core</artifactId>
    <version>{lettuce}</version>
  </dependency>

</dependencies>
```

以下示例显示了如何创建新的 Lettuce 连接工厂:

```
@Configuration
class AppConfig {

  @Bean
  public LettuceConnectionFactory redisConnectionFactory() {

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

还可以调整 Lettuce 连接参数.`LettuceConnectionFactory` 创建的所有 `LettuceConnection` 实例并且对于所有非阻塞和非事务操作共享相同的线程安全本地连接. 要每次使用特定连接,请将 `shareNativeConnection` 设置为 `false`. 如果将 `shareNativeConnection` 设置为 `false`,则还可以将 `LettuceConnectionFactory` 配置为使用 `LettucePool` 来池化阻塞和事务连接或所有连接.

Lettuce 与 Netty 的 native transports 集成,可让您使用Unix域套接字与Redis通信. 确保包括与您的运行时环境匹配的适当的本地传输依赖. 以下示例显示了如何在 `/var/run/redis.sock` 上为Unix域套接字创建 Lettuce Connection 工厂:

```
@Configuration
class AppConfig {

  @Bean
  public LettuceConnectionFactory redisConnectionFactory() {

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

Netty 当前支持用于OS本地传输的 epoll(Linux)和 kqueue (BSD/macOS)接口.

9.4.3. 配置 Jedis 连接器

Jedis 是一个社区驱动的连接器, Spring Data Redis 模块通过 `org.springframework.data.redis.connection.jedis` 包支持.

Add the following to the pom.xml files `dependencies` *element:*

```
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>{jedis}</version>
  </dependency>

</dependencies>
```

Jedis 配置以最简单的形式如下所示:

```
@Configuration
class AppConfig {

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

但是,对于生产用途,您可能想要调整主机或密码等设置,如以下示例所示:

```
@Configuration
class RedisConfiguration {

  @Bean
  public JedisConnectionFactory redisConnectionFactory() {

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

9.4.4. 主从复制

Redis 主从复制设置 —— 没有自动故障转移(有关自动故障转移,请参见:Sentinel)允许将数据安全地存储在更多节点上. 通过使用 Lettuce,可以将数据写入 Master,然后从副本中读取. 您可以使用 `LettuceClientConfiguration` 设置 读/写策略,如以下示例所示:

```
@Configuration
class WriteToMasterReadFromReplicaConfiguration {

  @Bean
  public LettuceConnectionFactory redisConnectionFactory() {

    LettuceClientConfiguration clientConfig = LettuceClientConfiguration.builder()
      .readFrom(SLAVE_PREFERRED)
      .build();

    RedisStandaloneConfiguration serverConfig = new
RedisStandaloneConfiguration("server", 6379);

    return new LettuceConnectionFactory(serverConfig, clientConfig);
  }
}
```

对于通过 INFO 信息报告非公共地址的环境(例如,在使用AWS时),请使用 RedisStaticMasterReplicaConfiguration 而不是 RedisStandaloneConfiguration. 请注意,RedisStaticMasterReplicaConfiguration 不支持发布/订阅,因为缺少在各个服务器之间传播的 Pub/Sub 消息.

9.5. Redis Sentinel 支持

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:

```
/**
 * 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);
}
```

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

Configuration Properties

- `spring.redis.sentinel.master`: 主节点的名称
- `spring.redis.sentinel.nodes`: 以逗号分隔的 host:port 列表
- `spring.redis.sentinel.password`: 使用 Redis Sentinel 进行身份验证时要应用的密码

有时,需要与其中一个哨兵进行直接互动. 使用 `RedisConnectionFactory.getSentinelConnection()` 或 `RedisConnection.getSentinelCommands()` 可以访问配置的第一个活动 Sentinel.

Sentinel身份验证仅在使用 Lettuce 时可用.

9.6. 通过 RedisTemplate 处理对象

大多数用户可能会使用 RedisTemplate 及其相应的包 `org.springframework.data.redis.core`. 实际上,由于 template 具有丰富的功能,因此它是Redis模块的核心类. 该 template 为 Redis 交互提供了高级抽象. 尽管 `RedisConnection` 提供了接受和返回二进制值(字节数组)的底层方法,但是 template 负责序列化和连接管理,使用户无需处理此类细节.

此外,该 template 提供了操作视图(根据 Redis 命令 参考进行分组),提供了丰富,通用的接口,用于处理特定类型或特定键(通过 KeyBound 接口),如下表所述:

Table 2. Operational 视图

Interface	Description
Key Type Operations	
`GeoOperations`	Redis 地理空间操作, 例如 `GEOADD`, `GEORADIUS`,…
`HashOperations`	Redis hash 操作
`HyperLogLogOperations`	Redis HyperLogLog 操作, 例如 `PFADD`, `PFCOUNT`,…
`ListOperations`	Redis list 操作
`SetOperations`	Redis set 操作
`ValueOperations`	Redis string (or value) 操作
`ZSetOperations`	Redis zset (or sorted set) 操作
Key 绑定 Operations	
`BoundGeoOperations`	Redis key bound geospatial 操作
`BoundHashOperations`	Redis hash key bound 操作
`BoundKeyOperations`	Redis key bound 操作
`BoundListOperations`	Redis list key bound 操作
`BoundSetOperations`	Redis set key bound 操作
`BoundValueOperations`	Redis string (or value) key bound 操作
`BoundZSetOperations`	Redis zset (or sorted set) key bound 操作

配置后,该 template 是线程安全的,并且可以在多个实例之间重用.

`RedisTemplate` 的大多数操作都使用基于 Java 的序列化. 这意味着 template 读写任何对象都将通过 Java 进行序列化和反序列化. 您可以更改 template 上的序列化机制,`Redis` 模块提供了几种实现,这些实现在 `org.springframework.data.redis.serializer` 包中可用. 有关更多信息,请参见序列化器. 您还可以将任何序列化器设置为 `null`,并通过将 `enableDefaultSerializer` 属性设置为 `false` 来将 RedisTemplate 与原始字节数组一起使用. 请注意,template 要求所有 key 都不为空. 但是,只要基础序列化器接受这些值,它们就可以为空. 阅读每个序列化器的 Javadoc,以获取更多信息.

对于需要特定 template 视图的情况,请将视图声明为依赖并注入template. 容器自动执行转换,从而消除了 `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
https://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());
  }
}
```

9.7. String 的便捷操作

由于 Redis 中存储的键和值通常是 java.lang.String,因此 Redis 模块提供了 RedisConnection 和 RedisTemplate 的两个扩展,分别是 StringRedisConnection(及其 DefaultStringRedisConnection 实现)和 StringRedisTemplate, 这是一种便捷的一站式解决方案用于密集的String操作. 除了绑定到字符串键之外,template 和连接还使用下面的 StringRedisSerializer,这意味着存储的键和值是人类可读的(假定Redis和您的代码中使用相同的编码). 以下清单显示了一个示例:

```
<?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
https://www.springframework.org/schema/beans/spring-beans.xsd">

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

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

```
public class Example {

  @Autowired
  private StringRedisTemplate redisTemplate;

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

与其他 Spring templates 一样,RedisTemplate 和 StringRedisTemplate 允许您通过 RedisCallback 接口直接与 Redis 交互. 此功能直接与 RedisConnection 对话,因此可以进行完全控制. 请注意,当使用 StringRedisTemplate 时,回调将接收 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");
    }
   });
}
```

9.8. 序列化器

从框架的角度来看,Redis 中存储的数据仅为字节. 尽管 Redis 本身支持各种类型,但在大多数情况下,它们是指数据的存储方式,而不是其表示的内容. 由用户决定是否将信息转换为字符串或任何其他对象.

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)由 `org.springframework.data.redis.serializer` 包中的 Redis 处理.

该包中包含两种类型的序列化器,顾名思义,它们负责序列化过程:

- 基于 `RedisSerializer` 的双向序列化器.
- 使用 `RedisElementReader` 和 `RedisElementWriter` 的元素 readers 和 writers.

他们之间的主要区别在于,`RedisSerializer` 主要序列化为 `byte[]`,而 readers 和 writers 则使用 `ByteBuffer`.

有多种实现方式(包括本文档中已经提到的两种):

- `JdkSerializationRedisSerializer`, 默认情况下用于 `RedisCache` 和 `RedisTemplate`.
- `StringRedisSerializer`.

但是,可以通过 Spring OXM 支持或使用 `Jackson2JsonRedisSerializer` 或 `GenericJackson2JsonRedisSerializer` 将 `OxmSerializer` 用于 对象/XML映射,以 JSON 格式存储数据.

请注意,存储格式不仅限于 values. 它可以不受限制地用于 keys,values 或 hashes.

> 默认情况下,`RedisCache` 和 `RedisTemplate` 配置为使用 Java 的序列化机制. Java 的序列化机制允许 payloads 引发的远程代码执行,这些 payloads 利用易受攻击的库和类注入未验证的字节码. 在反序列化步骤中,操纵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此序列化. 通常,我们强烈建议您使用其他任何消息格式(例如JSON).
>
> 如果您担心由 Java 序列化引起的安全漏洞,请考虑在核心JVM级别上使用通用序列化过滤器机制,该机制最初是为 JDK 9 开发的,但后来又移植到 JDK 8´7 和 6:
>
> - Filter Incoming Serialization Data.
> - JEP 290.
> - OWASP: Deserialization of untrusted data.

9.9. Hash 映射

可以通过在 Redis 中使用各种数据结构来存储数据. `Jackson2JsonRedisSerializer` 可以转换 JSON 格式的对象. 理想情况下,json数据作为一个字符创 key 的值. 您可以使用 Redis hash 来实现结构化对象的更复杂的映射. Spring Data Redis 提供了各种将数据映射到哈希的策略(取决于用例):

- 通过使用 `HashOperations` 和 serializer 进行直接映射
- 使用 Redis 存储库
- 使用 `HashMapper` 和 `HashOperations`

9.9.1. Hash Mappers

Hash mappers 是将 map 对象 转为 `Map<K,V>` 并返回. HashMapper 一般都与 Redis 哈希 一起使用.

有多种实现方式:

- `BeanUtilsHashMapper` 使用 Spring 的 BeanUtils.
- `ObjectHashMapper` 使用 对象到哈希的映射.

- `Jackson2HashMapper` 使用 FasterXML Jackson.

以下示例显示了一种实现哈希映射的方法:

```
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);
  }
}
```

9.9.2. Jackson2HashMapper

`Jackson2HashMapper` 通过使用 FasterXML Jackson 为域对象提供 Redis 哈希映射. `Jackson2HashMapper` 可以将顶级属性映射为 Hash 字段名称,并且可以选择将结构展平. 简单类型映射到简单值. 复杂类型(嵌套对象,集合,地图等)表示为嵌套JSON.

拼合为所有嵌套属性创建单个哈希条目,并尽可能将复杂类型解析为简单类型.

考虑以下类及其包含的数据结构:

```
public class Person {
  String firstname;
  String lastname;
  Address address;
  Date date;
  LocalDateTime localDateTime;
}

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

映射结果:

Table 3. Normal Mapping

Hash Field	Value
firstname	Jon
lastname	Snow
address	{ "city" : "Castle Black", "country" : "The North" }
date	1561543964015
localDateTime	2018-01-02T12:13:14

Flat Mapping

Table 4. Flat Mapping

Hash Field	Value
firstname	Jon
lastname	Snow
address.city	Castle Black
address.country	The North
date	1561543964015
localDateTime	2018-01-02T12:13:14

Flattening 要求所有属性名称都不得干扰JSON路径. 使用 flattening 时,不支持在 map keys 中使用 点 或方括号或将其用作属性名称. 结果哈希不能映射为一个对象.

java.util.Date 和 java.util.Calendar 用毫秒表示. 假如 jackson-datatype-jsr310 在类路径上,JSR-310 Date/Time 类型的将被序列化为他们的 toString 形式.

9.10. Redis 消息 (Pub/Sub)

Spring Data 为 Redis 提供了专用的消息传递集成,其功能和命名与 Spring Framework 中的 JMS 集成相似.

Redis 消息传递可以大致分为两个功能区域:

- 消息的发布或产生
- 订阅或消费消息

这是通常称为 "发布/订阅”(简称 "发布/订阅”)的模式的示例. `RedisTemplate` 类用于消息生成. 对于类似于 Java EE 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,该容器用于创建消息驱动的 POJO(MDP),并用于同步接收 `RedisConnection` 契约.

`org.springframework.data.redis.connection` 和 `org.springframework.data.redis.listener` 包提供了 Redis 消息的核心功能

9.10.1. Publishing (发送消息)

要发布消息,可以与其他操作一起使用底层 `RedisConnection` 或高级别 `RedisTemplate`. 这两个实体都提供了 `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递,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");
```

9.10.2. Subscribing (接收消息)

在接收端,可以直接命名一个 channels 或使用模式匹配来订阅一个或多个 channels. 后一种方法非常有用,因为它不仅允许使用一个命令创建多个订阅,而且还可以监听在订阅时尚未创建的 channel(只要它们与模式匹配).

在低层,`RedisConnection` 提供了 `subscription` 和 `pSubscribe` 方法,它们分别映射 Redis 命令以分别按通道或按模式进行订阅. 注意,可以将多个通道或模式用作参数. 要更改订阅的连接或查询连接是否在监听,`RedisConnection` 提供了 `getSubscription` 和 `isSubscribed` 方法.

> Spring Data Redis 中的订阅命令是阻塞的. 也就是说,在连接上调用订阅会导致当前线程阻塞,等待消息返回. 仅当取消订阅时才释放线程,这是在另一个线程在同一连接上调用 `unsubscribe` 或 `pUnsubscribe` 时发生的. 有关此问题的解决方案,请参见 “消息监听容器” (在本文档后面).

如前所述,一旦订阅,连接即开始等待消息. 仅允许添加新订阅,修改现有订阅以及取消现有订阅的命令. 调用 `subscribe`,`pSubscribe`, `unsubscribe` 或 `pUnsubscribe` 之外的其他任何操作都会引发异常.

为了订阅消息,需要实现 `MessageListener` 回调. 每次收到新消息时,都会通过 `onMessage` 方法调用回调并运行用户代码. 该接口不仅可以访问实际消息,还可以访问已通过其接收的通道,以及订阅使用的与通道匹配的模式(如果有). 该信息使被叫方不仅可以按内容来区分各种消息,还可以检查其他细节.

消息监听容器

由于其阻塞性质,底层订阅并不吸引人,因为它要求每个监听器都具有连接和线程管理. 为了减轻这个问题,Spring Data 提供了

`RedisMessageListenerContainer`,它可以完成所有繁重的工作. 如果您熟悉 EJB 和J MS,则应该熟悉这些概念,因为它被设计为尽可能接近 Spring Framework 及其消息驱动的 POJO(MDP)的支持.

`RedisMessageListenerContainer` 充当消息监听器容器. 它用于接收来自 Redis 通道的消息,并驱动注入到其中的 `MessageListener` 实例. 监听器容器负责消息接收的所有线程,并分派到监听器中进行处理. 消息监听器容器是MDP与消息传递提供程序之间的中介,并负责注册接收消息,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(并对消息做出响应)相关的(可能很复杂的)业务逻辑,并将样板Redis基础结构问题委托给框架.

此外,为了最大限度地减少应用程序占用空间,`RedisMessageListenerContainer` 允许一个连接和一个线程由多个监听器共享,即使它们不共享订阅.因此,无论应用程序跟踪多少监听器或通道,运行时间成本在整个生命周期中都保持不变. 此外,该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监听器,而无需重新启动.此外,容器使用延迟订阅方法,仅在需要时才使用 `RedisConnection`.如果所有监听器都未订阅,则将自动执行清除,然后释放线程.

为了帮助解决消息的异步特性,容器需要使用 `java.util.concurrent.Executor`(或 Spring 的 `TaskExecutor`)来调度消息.根据负载,监听器的数量或运行时环境,应更改或调整执行程序,以更好地满足您的需求. 特别是在托管环境(例如应用服务器)中,强烈建议选择合适的 `TaskExecutor` 以利用其运行时.

MessageListenerAdapter

`MessageListenerAdapter` 类是 Spring 异步消息支持中的最后一个组件. 简而言之,它使您几乎可以将任何类暴露为MDP(尽管存在一些约束).

考虑以下接口定义:

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

请注意,尽管该接口未扩展 `MessageListener` 接口,但仍可以通过使用 `MessageListenerAdapter` 类将其用作MDP. 还请注意,如何根据各种消息处理方法可以接收和处理的各种消息类型的内容来强类型化. 另外,可以将发送消息的通道或模式作为 String 类型的第二个参数传递给该方法:

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

注意上面的 `MessageDelegate` 接口的实现(上面的 `DefaultMessageDelegate` 类)如何完全没有 Redis 依赖. 这确实是我们使用以下配置将 MPO 制成的 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
https://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/redis
https://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>
```

监听器主题可以是通道(for example, `topic="chatroom"`) 或模式(for example, `topic="*room"`)

前面的示例使用 Redis 命名空间声明消息监听器容器,并自动将POJO注册为监听器. 完整的定义如下:

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

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

每次接收到消息时,适配器都会自动且透明地(使用配置的 `RedisSerializer`)在底层格式和所需对象类型之间执行转换. 容器捕获并处理由方法调用引起的任何异常(默认情况下,异常会被记录).

9.11. Redis Streams

Redis Streams 以更抽象的方式模拟日志数据结构。就像一个日志文件,通常实现为以只附加模式打开的文件,从一开始就在随机位置或通过流式传输新消息来使用日志。

> Redis 参考文档中获取有关 Redis Stream 的文档。

Redis Streams 可以大致分为两个功能区域:

- 追加记录
- 消费记录

尽管此模式与发布/订阅有相似之处,但主要区别在于消息的持久性及其使用方式。

尽管发布/订阅依赖于广播瞬态消息(即,如果您不收听,则会错过一条消息),但 Redis Stream 使用一种持久的,仅追加的数据类型,该类型将保留消息,直到流被修剪为止。消费方面的另一个区别是Pub/Sub 注册了服务器端订阅。Redis Stream 需要活动轮询时,Redis 将到达的消息推送到客户端。

`org.springframework.data.redis.connection` 和 `org.springframework.data.redis.stream` 包提供 Redis Streams 的核心功能。

> Redis Stream 支持当前仅可通过 Lettuce 客户端获得,因为 Jedis 尚不支持。

9.11.1. 追加

要发送记录,可以与其他操作一起使用底层 `RedisConnection` 或高级 `StreamOperations`。这两个实体都提供 `add` (`xAdd`) 方法,该方法接受记录和目标流作为参数。尽管 `RedisConnection` 需要原始数据(字节数组),但 `StreamOperations` 允许将任意对象作为记录传递,如以下示例所示:

```
// append message through connection
RedisConnection con = …
byte[] stream = …
ByteRecord record = StreamRecords.rawBytes(…).withStreamKey(stream);
con.xAdd(record);

// append message through RedisTemplate
RedisTemplate template = …
StringRecord record = StreamRecords.string(…).withStreamKey("my-stream");
template.streamOps().add(record);
```

Stream 记录携带 `Map`,键值元组作为其有效负载。将记录追加到流将返回 RecordId,该 RecordId 可用作进一步参考。

9.11.2. 消费

在消费方面,可以消费一个或多个流。Redis 流提供读取命令,该命令允许从已知流内容内的任意位置(流访问之外)的任意位置(随机访问)使用流,以消耗新的流记录。

在低层,`RedisConnection` 提供了 `xRead` 和 `xReadGroup` 方法,它们分别映射 Redis 命令以在使用者组中进行读取和读取。请注意,可以将多个流用作参数。

> Redis 中的订阅命令可能被阻塞。也就是说,在连接上调用 `xRead` 会导致当前线程在开始等待消息并阻塞。仅当读取命令超时或收到消息时才释放线程。

要使用流消息,可以在应用程序代码中轮询消息,或者通过消息监听器容器使用两种 通过消息监听器容器进行异步接收 之一,即命令式或响应式。每当有新记录到达时,容器都会通知应用程序代码。

同步接收

虽然消费流通常与异步处理相关联,但是可以同步消费消息. 重载的 `StreamOperations.read(…)` 方法提供了此功能. 在同步接收期间,调用线程可能会阻塞,直到消息可用为止. `StreamReadOptions.block` 属性指定接收者在放弃等待消息之前应该等待多长时间.

```
// Read message through RedisTemplate
RedisTemplate template = …

List<MapRecord<K, HK, HV>> messages =
template.streamOps().read(StreamReadOptions.empty().count(2),
                StreamOffset.latest("my-stream"));

List<MapRecord<K, HK, HV>> messages = template.streamOps().read(Consumer.from("my-
group", "my-consumer"),
                StreamReadOptions.empty().count(2),
                StreamOffset.create("my-stream", ReadOffset.lastConsumed()))
```

通过消息监听器容器进行异步接收

由于其阻塞性质,低级别轮询没有吸引力,因为它需要每个单个使用者进行连接和线程管理. 为了减轻这个问题,Spring Data 提供了消息监听器,它可以完成所有繁重的工作. 如果您熟悉 EJB 和 JMS,则应该熟悉这些概念,因为它被设计为尽可能接近 Spring Framework 及其消息驱动的POJO(MDP)的支持.

Spring Data 附带了两种针对所用编程模型的实现:

- `StreamMessageListenerContainer` 充当命令式编程模型的消息监听器容器. 它用于消耗Redis流中的记录并驱动注入其中的 `StreamListener` 实例.
- `StreamReceiver` 提供了消息监听器的响应式变体. 它用于将来自Redis流的消息作为潜在的无限流使用,并通过 `Flux` 发出流消息.

`StreamMessageListenerContainer` 和 `StreamReceiver` 负责消息接收的所有线程,并分派到监听器中进行处理. 消息监听器容器/接收器是MDP与消息传递提供程序之间的中介,并负责注册接收消息,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(并对消息做出响应)相关的(可能很复杂的)业务逻辑,并将样板 Redis 基础结构问题委托给框架.

这两个容器都允许更改运行时配置,因此您可以在应用程序运行时添加或删除订阅,而无需重新启动. 此外,容器使用延迟订阅方法,仅在需要时才使用 `RedisConnection`. 如果所有监听器都未订阅,它将自动执行清除,然后释放线程.

`StreamMessageListenerContainer`

流驱动POJO(SDP)以类似于EJB世界中的消息驱动Bean(MDB)的方式充当流消息的接收者. SDP 的一个限制是它必须实现 `org.springframework.data.redis.stream.StreamListener` 接口. 还请注意,在POJO在多个线程上接收消息的情况下,确保您的实现是线程安全的很重要.

```
class ExampleStreamListener implements StreamListener<String, MapRecord<String,
String, String>> {

    @Override
    public void onMessage(MapRecord<String, String, String> message) {

        System.out.println("MessageId: " + message.getId());
        System.out.println("Stream: " + message.getStream());
        System.out.println("Body: " + message.getValue());
    }
}
```

StreamListener 表示一个功能接口,因此可以使用其 Lambda 形式重写实现:

```
message -> {

    System.out.println("MessageId: " + message.getId());
    System.out.println("Stream: " + message.getStream());
    System.out.println("Body: " + message.getValue());
};
```

一旦实现了 StreamListener,就可以创建一个消息监听器容器并注册订阅了:

```
RedisConnectionFactory connectionFactory = …
StreamListener<String, MapRecord<String, String, String>> streamListener = …

StreamMessageListenerContainerOptions<String, MapRecord<String, String, String>>
containerOptions = StreamMessageListenerContainerOptions
            .builder().pollTimeout(Duration.ofMillis(100)).build();

StreamMessageListenerContainer<String, MapRecord<String, String, String>> container =
StreamMessageListenerContainer.create(connectionFactory,
                containerOptions);

Subscription subscription = container.receive(StreamOffset.fromStart("my-stream"),
streamListener);
```

请参阅各种消息监听器容器的 Javadoc,以获取每个实现所支持功能的完整说明.

Reactive StreamReceiver

流式数据源的 Reactive consumption 通常通过 Flux 的事件或消息发生. 响应接收器实现由 StreamReceiver 和重载的 receive(…) 消息提供. 与 StreamMessageListenerContainer 相比,被动方法需要更少的基础结构资源,例如线程,因为它利用了驱动程序提供的线程资源. 接收流是 StreamMessage 的需求驱动的发布者:

```
Flux<MapRecord<String, String, String>> messages = …

return messages.doOnNext(it -> {
    System.out.println("MessageId: " + message.getId());
    System.out.println("Stream: " + message.getStream());
    System.out.println("Body: " + message.getValue());
});
```

现在我们需要创建 `StreamReceiver` 并注册一个订阅以使用流消息:

```
ReactiveRedisConnectionFactory connectionFactory = …

StreamReceiverOptions<String, MapRecord<String, String, String>> options =
StreamReceiverOptions.builder().pollTimeout(Duration.ofMillis(100))
                .build();
StreamReceiver<String, MapRecord<String, String, String>> receiver =
StreamReceiver.create(connectionFactory, options);

Flux<MapRecord<String, String, String>> messages =
receiver.receive(StreamOffset.fromStart("my-stream"));
```

请参阅各种消息监听器容器的 Javadoc,以获取每个实现所支持功能的完整说明.

需求驱动的消耗使用背压信号来激活和停用轮询. 如果满足需求,则 `StreamReceiver` 订阅将暂停轮询,直到订阅者发出进一步的请求. 根据 `ReadOffset` 策略,这可能导致消息被跳过.

`Acknowledge` 策略

当您通过 `Consumer Group` 阅读邮件时,服务器将记住已传递给定邮件,并将其添加到"待处理条目列表"(PEL)中. 已发送但尚未确认的邮件列表. 消息必须通过 `StreamOperations.acknowledge` 进行确认,以便从待处理条目列表中删除,如下面的代码片段所示.

```
StreamMessageListenerContainer<String, MapRecord<String, String, String>>
container = ...

container.receive(Consumer.from("my-group", "my-consumer"), ①
    StreamOffset.create("my-stream", ReadOffset.lastConsumed()),
    msg -> {

        // ...
        redisTemplate.opsForStream().acknowledge("my-group", msg); ②
    });
```

① 从 *my-group* 组读取为 *my-consumer*. 收到的消息不被确认.

② 处理后确认消息.

要在接收时自动确认消息,请使用 `receiveAutoAck` 而不是 `receive`.

`ReadOffset` 策略

流读取操作接受读取偏移量规范以从给定偏移量开始消耗消息. `ReadOffset` 表示读取偏移量规范. Redis支持三种偏移量,具体取决于您是独立使用流还是在使用者组中使用流:

- `ReadOffset.latest()` — 阅读最新消息.
- `ReadOffset.from(…)` — 在特定消息ID之后阅读.
- `ReadOffset.lastConsumed()` — 在最后消耗的消息ID之后读取(仅针对消费者组).

在基于消息容器的使用情况下,我们在使用消息时需要提高(或增加)读取偏移量. 前进取决于请求的 `ReadOffset` 和消费模式(有/无消费组). 以下矩阵说明了容器如何提高 `ReadOffset`:

Table 5. ReadOffset Advancing

Read offset	Standalone	Consumer Group
Latest	Read latest message	Read latest message
Specific Message Id	Use last seen message as the next MessageId	Use last seen message as the next MessageId
Last Consumed	Use last seen message as the next MessageId	Last consumed message as per consumer group

从特定消息ID和最后使用的消息读取可以被视为安全操作,可确保消耗附加到流中的所有消息. 使用最新消息进行读取可以跳过在轮询操作处于停滞时间状态时添加到流中的消息. 轮询会引入一个停滞时间,在该停滞时间内消息可以在各个轮询命令之间到达. 流消耗不是线性连续读取, 而是分成重复的 `XREAD` 调用.

序列化

发送到流的任何记录都需要序列化为其二进制格式. 由于流与哈希数据结构非常接近,因此流键,字段名称和值使用在 `RedisTemplate` 上配置的相应序列化器.

Table 6. Stream Serialization

Stream Property	Serializer	Description
key	keySerializer	used for `Record#getStream()`
field	hashKeySerializer	used for each map key in the payload
value	hashValueSerializer	used for each map value in the payload

请确保查看使用中的Redis `RedisSerializer`,并注意,如果您决定不使用任何序列化器,则需要确保这些值已经是二进制的.

Object Mapping

简单的值

`StreamOperations` 允许通过 `ObjectRecord` 将简单值直接附加到流,而不必将这些值放入 `Map` 结构中. 然后将该值分配给有效负载字段,并在读回该值时可以将其提取.

```
ObjectRecord<String, String> record = StreamRecords.newRecord()
    .in("my-stream")
    .ofObject("my-value");

redisTemplate()
    .opsForStream()
    .add(record); ①

List<ObjectRecord<String, String>> records = redisTemplate()
    .opsForStream()
    .read(String.class, StreamOffset.fromStart("my-stream"));
```

① XADD my-stream * "_class" "java.lang.String" "_raw" "my-value"

`ObjectRecords` 与所有其他记录通过相同的序列化过程,因此 Record 也可以使用返回 `MapRecord` 的无类型读取操作获得。

复杂的值

可以通过3种方式将复杂的值添加到流中:

- 使用转换为简单值。字符串JSON表示形式。
- 用合适的 `RedisSerializer` 序列化该值。
- 使用 `HashMapper` 将值转换为适合于序列化的 `Map`。

第一个变体是最直接的变体,但忽略了流结构提供的字段值功能,但流中的值仍可供其他使用者读取。第二个选项具有与第一个选项相同的好处,但是可能会导致非常特殊的使用方限制,因为所有使用方都必须实现完全相同的序列化机制。 `HashMapper` 方法是一种更复杂的方法,它使用了 Steam 哈希结构,但是却使源代码变得平坦。只要选择了合适的序列化器组合,其他使用者仍然可以读取记录。

`HashMappers` 将有效负载转换为具有特定类型的Map。确保使用能够(反)序列化哈希的哈希键和哈希值序列化程序。

```
ObjectRecord<String, User> record = StreamRecords.newRecord()
    .in("user-logon")
    .ofObject(new User("night", "angel"));

redisTemplate()
    .opsForStream()
    .add(record); ①

List<ObjectRecord<String, User>> records = redisTemplate()
    .opsForStream()
    .read(User.class, StreamOffset.fromStart("user-logon"));
```

① XADD user-logon * "_class" "com.example.User" "firstname" "night" "lastname" "angel"

默认情况下,`StreamOperations` 使用 ObjectHashMapper。获取 `StreamOperations` 时,可以提供适合您要求的 `HashMapper`。

```
redisTemplate()
    .opsForStream(new Jackson2HashMapper(true))
    .add(record); ①
```

① XADD user-logon * "firstname" "night" "@class" "com.example.User" "lastname" "angel"

StreamMessageListenerContainer 可能并不知道 domain 类型上使用的任何 @TypeAlias, 因为需要通过 MappingContext 进行解析. 确保使用 initialEntitySet 初始化 RedisMappingContext.

```
@Bean
RedisMappingContext redisMappingContext() {
    RedisMappingContext ctx = new RedisMappingContext();
    ctx.setInitialEntitySet(Collections.singleton(Person.class));
    return ctx;
}

@Bean
RedisConverter redisConverter(RedisMappingContext mappingContext) {
    return new MappingRedisConverter(mappingContext);
}

@Bean
ObjectHashMapper hashMapper(RedisConverter converter) {
    return new ObjectHashMapper(converter);
}

@Bean
StreamMessageListenerContainer
streamMessageListenerContainer(RedisConnectionFactory
connectionFactory, ObjectHashMapper hashMapper) {
    StreamMessageListenerContainerOptions<String, ObjectRecord<String,
Object>> options = StreamMessageListenerContainerOptions.builder()
            .objectMapper(hashMapper)
            .build();

    return StreamMessageListenerContainer.create(connectionFactory,
options);
}
```

9.12. Redis 事务

Redis通过 multi, exec, 和 discard 命令为 事务 提供支持. 这些操作在 RedisTemplate 上可用. 但是,不能保证 RedisTemplate 使用相同的连接来执行事务中的所有操作.

当需要在同一连接上执行多个操作时,例如在使用 Redis 事务时,Spring Data Redis 提供了 SessionCallback 接口. 下面的示例使用 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` 使用其值,哈希键和哈希值序列化程序在返回之前反序列化 `exec` 的所有结果. 还有一个附加的 `exec` 方法,可让您传递自定义的序列化程序以获取事务结果.

从版本1.1开始,已对 `RedisConnection` 和 `RedisTemplate` 的 `exec` 方法进行了重要更改. 以前,这些方法直接从连接器返回事务处理的结果. 这意味着数据类型通常不同于 `RedisConnection` 方法返回的数据类型. 例如,`zAdd` 返回一个布尔值,指示该元素是否已添加到排序集中. 大多数连接器都会将此值返回为 `long`,而 Spring Data Redis 将执行转换. 另一个常见的区别是,大多数连接器针对诸如 `set` 之类的操作返回状态答复 (通常为字符串 `OK`). 这些答复通常被 Spring Data Redis 丢弃. 在1.1之前的版本中,未对 `exec` 的结果执行这些转换. 此外,结果未在 `RedisTemplate` 中反序列化,因此它们通常包含原始字节数组. 如果此更改使您的应用程序中断,请在 `RedisConnectionFactory` 上将 `convertPipelineAndTxResults` 设置为 `false` 以禁用此行为.

9.12.1. @Transactional 支持

默认情况下,事务支持是禁用的,必须通过设置 `setEnableTransactionSupport(true)` 为正在使用的每个 `RedisTemplate` 显式启用. 这样做会强制将当前 `RedisConnection` 绑定到触发 `MULTI` 的当前线程. 如果事务顺利完成,则调用 `EXEC`. 否则调用 `DISCARD`. 一旦进入 `MULTI`,`RedisConnection` 会将写入操作排队. 所有只读操作(例如 `KEYS`)都通过管道传递到新的(非线程绑定)`RedisConnection`.

以下示例显示如何配置事务管理:

Example 3. Configuration enabling Transaction Management

```
@Configuration
@EnableTransactionManagement                                    ①
public class RedisTxContextConfiguration {

  @Bean
  public StringRedisTemplate redisTemplate() {
    StringRedisTemplate template = new
StringRedisTemplate(redisConnectionFactory());
    // explicitly enable transaction support
    template.setEnableTransactionSupport(true);                 ②
    return template;
  }

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

  @Bean
  public PlatformTransactionManager transactionManager() throws SQLException {
    return new DataSourceTransactionManager(dataSource());     ③
  }

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

① 配置一个 Spring Context 来启用 声明式事务管理.

② 通过将连接绑定到当前线程来配置 `RedisTemplate` 参与事务.

③ 事务管理需要 `PlatformTransactionManager`. Spring Data Redis 不附带 `PlatformTransactionManager` 实现. 假设您的应用程序使用J DBC,Spring Data Redis 可以使用现有的事务管理器参与事务.

以下示例每个都演示了使用约束:

Example 4. 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");
```

9.13. 流水线

Redis 提供了对 流水线 的支持,该流水线涉及到向服务器发送多个命令,而无需等待答复,然后在一个步骤中读取答复. 当您需要连续发送多个命令(例如将多个元素添加到同一List)时,流水线可以提高性能.

Spring Data Redis 提供了几种 `RedisTemplate` 方法来执行流水线中的命令. 如果您不关心流水线操作的结果,则可以使用标准的 `execute` 方法,将流水线参数传递为 `true`. `executePipelined` 方法在流水线中运行提供的 `RedisCallback` 或 `SessionCallback` 并返回结果,如以下示例所示:

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

前面的示例从流水线中的队列中运行项目的批量右弹出. 结果列表包含所有弹出项目. `RedisTemplate` 使用其值,哈希键和哈希值序列化器在返回之前对所有结果进行反序列化,因此前面示例中返回的项目为字符串. 还有其他的 `executePipelined` 方法,可让您传递自定义序列化程序以获取流水线结果.

请注意,从 `RedisCallback` 返回的值必须为 `null`,因为为了支持返回流水线命令的结果而将其丢弃.

Lettuce 驱动程序支持细粒度的刷新控制,该控件可以刷新出现的命令,在连接关闭时对其进行缓冲或发送.

```
LettuceConnectionFactory factory = // ...
factory.setPipeliningFlushPolicy(PipeliningFlushPolicy.buffered(3)); ①
```

① 在第三个命令后本地缓冲并刷新.

从版本1.1开始,已对 `RedisConnection` 和 `RedisTemplate` 的 `exec` 方法进行了重要更改. 以前,这些方法直接从连接器返回事务处理的结果. 这意味着数据类型通常不同于 `RedisConnection` 方法返回的数据类型. 例如,`zAdd` 返回一个布尔值,指示该元素是否已添加到排序集中. 大多数连接器都会将此值返回为 `long`,而 Spring Data Redis 将执行转换. 另一个常见的区别是,大多数连接器针对诸如 `set` 之类的操作返回状态答复 (通常为字符串 `OK`). 这些答复通常被 Spring Data Redis 丢弃. 在1.1之前的版本中,未对 `exec` 的结果执行这些转换. 此外,结果未在 `RedisTemplate` 中反序列化,因此它们通常包含原始字节数组. 如果此更改使您的应用程序中断,请在 `RedisConnectionFactory` 上将 `convertPipelineAndTxResults` 设置为 `false` 以禁用此行为.

9.14. Redis 脚本

Redis 2.6及更高版本通过 eval 和 evalsha 命令提供了对 Lua 脚本执行的支持. Spring Data Redis 为脚本执行提供了高级抽象,该抽象处理了序列化并自动使用 Redis 脚本缓存.

可以通过调用 `RedisTemplate` 和 `ReactiveRedisTemplate` 的 `execute` 方法来运行脚本.两者都使用可配置的 `ScriptExecutor`(或 `ReactiveScriptExecutor`)来运行提供的脚本.默认情况下,`ScriptExecutor`(或 `ReactiveScriptExecutor`)负责序列化提供的键和参数并反序列化脚本结果.这是通过模板的键和值序列化程序完成的,还有一个额外的重载,可让您传递脚本参数和结果的自定义序列化程序.

默认的 `ScriptExecutor` 通过检索脚本的SHA1并首先尝试运行 evalsha 来优化性能,如果Redis脚本缓存中还没有该脚本,则回退到eval.

以下示例通过使用 Lua 脚本运行常见的 “check-and-set” 方案,这是 Redis 脚本的理想用例,因为它需要原子地运行一组命令,并且一个命令的行为会受到另一个命令的结果的影响.

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

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

```
public class Example {

  @Autowired
  RedisScript<Boolean> script;

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

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

前面的代码配置 `RedisScript` 指向一个名为 `checkandset.lua` 的文件,该文件应返回布尔值。脚本 `resultType` 应该是 `Long`,`Boolean`,`List` 或反序列化值类型之一。如果脚本返回抛弃状态(特别是 `OK`),则它也可以为 `null`。

理想的是在您的应用程序上下文中配置 `DefaultRedisScript` 的单个实例,以避免在每次执行脚本时重新计算脚本的 SHA1。

然后,上面的 `checkAndSet` 方法运行脚本。脚本可以在 `SessionCallback` 内作为事务或管道的一部分运行。有关更多信息,请参见 “Redis 事务” 和 “流水线” 。

Spring Data Redis 提供的脚本支持还允许您通过使用 Spring Task 和 Scheduler 抽象来调度 Redis 脚本以定期执行。有关更多详细信息,请参见 Spring Framework 文档。

9.15. 支持的类

`org.springframework.data.redis.support` 包供了各种依赖 Redis 的可重用组件。当前,该包在 Redis 之上包含各种基于JDK的接口实现,例如 atomic和 JDK Collections.

通过原子计数器,可以轻松的使 Redis key 递增,而集合则可以轻松管理 Redis key,并最大程度地减少存储风险或API泄漏。特别是,`RedisSet` 和 `RedisZSet` 接口可轻松访问 Redis 支持的设置操作,例如交集和并集。`RedisList` 在 Redis 之上实现 List,Queue 和 Deque 契约(及其等效的阻塞同级),将存储作为 FIFO(先进先出),LIFO(先进先出)或加盖 只需最少的配置即可收集。以下示例显示了使用 `RedisList` 的bean的配置:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p" xsi:schemaLocation="
  http://www.springframework.org/schema/beans
https://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>
```

使用 Java 配置 `Deque`:

```
public class AnotherExample {

  // injected
  private Deque<String> queue;

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

如前面的示例所示,使用代码与实际的存储实现分离. 实际上,没有迹象表明在下面使用了 Redis. 这使得从开发环境到生产环境的迁移变得透明,并大大提高了可测试性(Redis实施可替换为内存中的实施).

9.15.1. 支持 Spring Cache 抽象

Changed in 2.0

Spring Redis 通过 `org.springframework.data.redis.cache` 包提供了 Spring 缓存抽象的实现. 要将 Redis 用作实现,请将 `RedisCacheManager` 添加到您的配置中,如下所示:

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

可以使用 `RedisCacheManagerBuilder` 配置 `RedisCacheManager` 行为,让您设置默认的 `RedisCacheConfiguration`,事务行为和预定义的缓存.

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

如前面的示例所示,`RedisCacheManager` 允许在每个缓存的基础上定义配置.

由 `RedisCacheManager` 创建的 `RedisCache` 的行为由 `RedisCacheConfiguration` 定义. 通过该配置,您可以设置 key 到期时间,前缀和 `RedisSerializer` 实现,以实现与二进制存储格式之间的相互转换,如以下示例所示:

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

`RedisCacheManager` 默认使用无锁 `RedisCacheWriter` 来读写二进制值. 无锁缓存提高了吞吐量. 缺少条目锁定会导致 `putIfAbsent` 和 `clean` 方法的重叠,非原子性命令重叠,因为这些方法需要将多个命令发送到R edis. 锁定对方通过设置显式锁定键并检查该键是否存在来防止命令重叠,从而导致其他请求和潜在的命令等待时间.

可以选择如下锁定行为:

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

默认情况下,缓存条目的任何 key 都以实际的缓存名称作为前缀,后跟两个冒号. 此行为可以更改为静态前缀和计算前缀.

以下示例显示了如何设置静态前缀:

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

The following example shows how to set a computed prefix:

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

下表列出了 `RedisCacheManager` 的默认设置

Table 7. `RedisCacheManager` defaults

Setting	Value
Cache Writer	Non-locking
Cache Configuration	`RedisCacheConfiguration#defaultConfiguration`
Initial Caches	None
Transaction Aware	No

下表列出了 `RedisCacheConfiguration` 的默认设置:

Table 8. RedisCacheConfiguration defaults

Key Expiration	None
Cache `null`	Yes
Prefix Keys	Yes
Default Prefix	The actual cache name
Key Serializer	`StringRedisSerializer`
Value Serializer	`JdkSerializationRedisSerializer`
Conversion Service	`DefaultFormattingConversionService` with default cache key converters

Chapter 10. Reactive Redis 支持

本节介绍了响应式Redis支持以及入门方法. 响应式 Redis 支持与 命令式 Redis 支持 有某些重叠.

10.1. Redis 要求

Spring Data Redis当前与 Lettuce 集成为唯一的响应式 Java 连接器. Project Reactor 用作响应式库.

10.2. 使用响应式驱动程序连接到 Redis

使用 Redis 和 Spring 时的首要任务之一是通过IoC容器连接到 store. 为此,需要Java连接器(或绑定). 无论选择哪种库,都必须使用 `org.springframework.data.redis.connection` 包及其 `ReactiveRedisConnection` 和 `ReactiveRedisConnectionFactory` 接口来使用和检索与Redis的活动连接.

10.2.1. Redis操作模式

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

10.2.2. `ReactiveRedisConnection` 和 `ReactiveRedisConnectionFactory`

`ReactiveRedisConnection` 是 Redis 通信的核心,因为它处理与 Redis 后端的通信. 它还会自动将底层驱动程序异常转换为 Spring 一致的DAO异常层次结构,因此您可以在不更改任何代码的情况下切换连接器,因为操作语义保持不变.

`ReactiveRedisConnectionFactory` 创建活动的 `ReactiveRedisConnection` 实例. 另外,工厂还充当 `PersistenceExceptionTranslator` 实例,这意味着工厂一旦声明,就可以进行透明的异常转换-例如,通过使用 `@Repository` 注解和AOP进行异常转换. 有关更多信息,请参见Spring Framework文档中的 特定部分 .

> 根据基础配置,工厂可以返回新连接或现有连接(如果使用池或共享本地连接).

> 使用 `ReactiveRedisConnectionFactory` 的最简单方法是通过IoC容器配置适当的连接器,并将其注入 using 类.

10.2.3. 配置 Lettuce 连接器

Spring Data Redis 通过 `org.springframework.data.redis.connection.lettuce` 包支持 Lettuce.

您可以按以下步骤配置 Lettuce `ReactiveRedisConnectionFactory`:

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

以下示例显示了使用 `LettuceClientConfigurationBuilder` 的更复杂的配置,包括SSL和超时:

```
@Bean
public ReactiveRedisConnectionFactory lettuceConnectionFactory() {

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

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

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

10.3. 通过 ReactiveRedisTemplate 处理对象

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

此外,该模板提供了操作视图(按照 Redis 命令参考进行分组之后),这些操作视图提供了丰富的通用接口来针对某种类型进行工作,如下表所述:

Table 9. Operational views

Interface	Description
Key Type Operations	
ReactiveGeoOperations	Redis geospatial operations such as `GEOADD`, `GEORADIUS`, and others)
ReactiveHashOperations	Redis hash operations
ReactiveHyperLogLogOperations	Redis HyperLogLog operations such as (`PFADD`, `PFCOUNT`, and others)
ReactiveListOperations	Redis list operations
ReactiveSetOperations	Redis set operations
ReactiveValueOperations	Redis string (or value) operations
ReactiveZSetOperations	Redis zset (or sorted set) operations

配置后,该模板是线程安全的,并且可以在多个实例之间重用.

`ReactiveRedisTemplate` 的大多数操作都使用基于 Java 的序列化器. 这意味着通过 `RedisElementWriter` 或 `RedisElementReader` 序列化或反序列化由模板写入或读取的任何对象. 序列化上下文在构建时传递给模板,Redis 模块在 `org.springframework.data.redis.serializer` 包中提供了几种可用的实现. 有关更多信息,请参见 序列化器.

以下示例显示了用于返回 `Mono` 的 `ReactiveRedisTemplate`:

```
@Configuration
class RedisConfiguration {

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

```
public class Example {

  @Autowired
  private ReactiveRedisTemplate<String, String> template;

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

10.4. String 的便捷操作类

由于 Redis 中存储的键和值通常是 java.lang.String,因此 Redis 模块为 ReactiveRedisTemplate 提供了基于 String 的扩展:`ReactiveStringRedisTemplate`. 这是用于密集型 String 操作的便捷的一站式解决方案. 除了绑定到字符串键之外, 该模板还使用基于字符串的 RedisSerializationContext,这意味着存储的键和值是人类可读的(假定Redis和您的代码使用相同的编码). 以下示例显示了正在使用的 ReactiveStringRedisTemplate:

```
@Configuration
class RedisConfiguration {

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

```
public class Example {

  @Autowired
  private ReactiveStringRedisTemplate redisTemplate;

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

10.5. Redis Messaging/PubSub

Spring Data 为 Redis 提供了专用的消息传递集成,其功能和命名与 Spring Framework 中的JMS集成非常相似; 实际上,熟悉 Spring 的 JMS 支持的用户应该有宾至如归的感觉.

Redis消息传递可以大致分为两个功能区域,即消息的产生或发布以及消息的使用或订阅,因此简称为 pubsub(发布/订阅). `ReactiveRedisTemplate` 类用于消息生成. 对于异步接收,Spring Data 提供了一个专用的消息监听器容器,该容器用于消耗消息流. 出于订阅目的,`ReactiveRedisTemplate` 提供了使用监听器容器的简化选择.

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

10.5.1. Sending/Publishing messages

要发布消息,可以与其他操作一样使用底层的 `ReactiveRedisConnection` 或高级 `ReactiveRedisTemplate`. 这两个实体都提供一种发布方法,该方法接受需要发送的消息以及目标通道作为参数. 尽管 `ReactiveRedisConnection` 需要原始数据,但 `ReactiveRedisTemplate` 允许将任意对象作为消息传递:

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

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

10.5.2. Receiving/Subscribing for messages

在接收端,可以直接命名一个通道或使用模式匹配来订阅一个或多个通道. 后一种方法非常有用,因为它不仅允许使用一个命令创建多个订阅,而且还可以监听在订阅时尚未创建的通道(只要它们与模式匹配).

在低层,`ReactiveRedisConnection` 提供了 subscription 和 `pSubscribe` 方法,它们映射Redis命令以分别按模式按通道进行订阅. 注意,可以将多个通道或模式用作参数. 要更改订阅,只需查询 `ReactiveSubscription` 的通道和模式.

Spring Data Redis中的响应式订阅命令是非阻塞的,可以终止而不会发出任何元素.

如上所述,一旦订阅,连接就会开始等待消息. 除了添加新订阅或修改/取消现有订阅之外,无法在其上调用其他命令. 除 `subscribe`,`pSubscribe`,`unsubscribe` 或 `pUnsubscribe` 以外的命令是非法的,并且会导致异常.

为了接收消息,需要获得消息流. 请注意,订阅仅发布有关在该特定订阅中注册的通道和模式的消息. 消息流本身是一个热序列,它在不考虑需求的情况下生成元素. 确保注册足够的需求,以免耗尽消息缓冲区.

Message 监听容器

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

`ReactiveRedisMessageListenerContainer` 充当消息监听器容器. 它用于从 Redis 通道接收消息并暴露消息流,该消息流发出应用反序列化的通道消息. 它负责注册接收消息,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(并对消息做出响应)相关的(可能很复杂)业务逻辑,并将样板Redis基础结构问题委托给框架. 消息流在发布者订阅后在 Redis 中注册一个订阅,如果订阅被取消,则注销.

此外,为了最大程度地减少应用程序占用空间,`ReactiveRedisMessageListenerContainer` 允许一个连接和一个线程由多个监听器共享,即使它们不共享订阅. 因此,无论应用程序跟踪多少个监听器或通道,整个生命周期的运行时间成本都将保持不变. 此外,该容器允许更改运行时配置,因此可以在应用程序运行时添加或删除监听器,而无需重新启动. 此外,容器使用惰性订阅方法,仅在需要时才使用 `ReactiveRedisConnection`---- 如果所有监听器都未订阅,则将自动执行清除.

消息监听器容器本身不需要外部线程资源. 它使用驱动程序线程来发布消息.

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

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

通过模板API订阅

如上所述,您可以直接使用 `ReactiveRedisTemplate` 订阅通道/模式. 这种方法提供了一种直接但有限的解决方案,因为您松开了在初始订阅之后添加订阅的选项. 不过,您仍然可以使用例如返回的 `Flux` 控制消息流. 服用(持续时间). 完成读取后,一旦出错或取消,所有绑定的资源将再次释放.

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

10.6. Reactive 脚本

您可以通过使用 `ReactiveScriptExecutor` 在响应式的基础上运行 Redis 脚本, 最好通过 `ReactiveRedisTemplate` 访问.

```
public class Example {

  @Autowired
  private ReactiveRedisTemplate<String, String> template;

  public Flux<Long> theAnswerToLife() {

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

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

有关脚本命令的更多详细信息,请参见 脚本部分.

Chapter 11. Redis 集群

使用 Redis Cluster 需要Redis Server 3.0+版本. 有关更多信息,请参见 集群教程.

11.1. 启用 Redis 集群

群集支持基于与非群集通信相同的构造块. `RedisClusterConnection` 是 `RedisConnection` 的扩展,用于处理与 Redis 群集的通信,并将错误转换为 Spring DAO 异常层次结构. `RedisClusterConnection` 实例是使用 `RedisConnectionFactory` 创建的, 必须使用关联的 `RedisClusterConfiguration` 对其进行设置,如以下示例所示:

Example 5. Redis 集群的示例 RedisConnectionFactory 配置

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

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

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

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

@Configuration
public class AppConfig {

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

    public @Bean RedisConnectionFactory connectionFactory() {

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

RedisClusterConfiguration 也可以通过 PropertySource 定义,并具有以下属性:

配置属性

- spring.redis.cluster.nodes: 以逗号分隔的 host:port 列表.
- spring.redis.cluster.max-redirects: 允许的集群重定向数.

初始配置将驱动程序库指向一组初始的群集节点。由实时群集重新配置导致的更改仅保留在本地驱动程序中,而不会写回到配置中。

11.2. 使用 Redis 群集连接

如前所述,Redis 群集的行为与单节点 Redis 甚至是 Sentinel 监视的主副本环境不同。 这是因为自动分片将密钥映射到16384个slots之一,该 slots 分布在节点上。因此,涉及多个键的命令必须断言所有键都映射到完全相同的 slots ,以避免跨 slots 执行错误。单个群集节点仅提供一组专用密钥。针对一台特定服务器发出的命令仅针对该服务器提供的那些键返回结果。作为一个简单的示例,请考虑 `KEYS` 命令。 当在集群环境中发布给服务器时,它仅返回请求发送到的节点所服务的密钥,而不一定返回集群中的所有密钥。因此,要在群集环境中获取所有密钥,必须从所有已知的主节点读取密钥。

虽然将特定密钥重定向到相应的 slots 服务节点是由驱动程序库处理的,但是 `RedisClusterConnection` 涵盖了更高级别的功能,例如跨节点收集信息或向集群中的所有节点发送命令。拾取前面的密钥示例,这意味着keys(pattern)方法将拾取群集中的每个主节点, 并同时在每个主节点上执行 `KEYS` 命令,同时拾取结果并返回累积的键集。仅请求单个节点的键,`RedisClusterConnection` 为这些方法提供了重载(例如,`keys(node, pattern)`)。

While redirects for specific keys to the corresponding slot-serving node are handled by the driver libraries, higher-level functions, such a s 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` co mmand 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` provide s overloads for those methods (for example, `keys(node, pattern)`).

可以从 `RedisClusterConnection.clusterGetNodes` 获得 `RedisClusterNode`,也可以使用主机和端口或节点ID来构造它。

以下示例显示了在集群中运行的一组命令:

Example 6. 跨集群运行命令的示例

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

6b38bb... 127.0.0.1:7379 master - 0 0 25 connected 0-5460                    ①
7bb78c... 127.0.0.1:7380 master - 0 1449730618304 2 connected 5461-10922     ②
164888... 127.0.0.1:7381 master - 0 1449730618304 3 connected 10923-16383    ③
b8b5ee... 127.0.0.1:7382 slave 6b38bb... 0 1449730618304 25 connected        ④
```

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

connection.set("thing1", value);                                             ⑤
connection.set("thing2", value);                                             ⑥

connection.keys("*");                                                        ⑦

connection.keys(NODE_7379, "*");                                             ⑧
connection.keys(NODE_7380, "*");                                             ⑨
connection.keys(NODE_7381, "*");                                             ⑩
connection.keys(NODE_7382, "*");                                             ⑪
```

① Master node serving slots 0 to 5460 replicated to replica at 7382

② Master node serving slots 5461 to 10922

③ Master node serving slots 10923 to 16383

④ Replica node holding replicants of the master at 7379

⑤ Request routed to node at 7381 serving slot 12182

⑥ Request routed to node at 7379 serving slot 5061

⑦ Request routed to nodes at 7379, 7380, 7381 → [thing1, thing2]

⑧ Request routed to node at 7379 → [thing2]

⑨ Request routed to node at 7380 → []

⑩ Request routed to node at 7381 → [thing1]

⑪ Request routed to node at 7382 → [thing2]

当所有键都映射到同一 slot 时,本地驱动程序库会自动处理跨 slots 请求,例如 `MGET`. 但是,一旦不是这种情况,`RedisClusterConnection` 会对 slots 服务节点执行多个并行的 `GET` 命令,然后再次返回累积的结果. 这比单 slot 执行的性能要差,因此应谨慎使用. 如有疑问,请考虑通过在大括号中提供前缀(例如 `{my-prefix}.thing1` 和 `{my-prefix}.thing2`)来将密钥固定到同一 slot ,这两者都将映射到相同的 slot 号. 以下示例显示了跨槽请求处理:

Example 7. 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                    ①
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");                                  ②

connection.mGet("thing1", "thing2");                                           ③
```

① 与之前示例中的配置相同。

② Keys map to same slot → 127.0.0.1:7381 MGET thing1 {thing1}.thing2

③ 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

前面的示例演示了Spring Data Redis遵循的一般策略。 请注意,某些操作可能需要将大量数据加载到内存中才能计算所需的命令。此外,并非所有跨槽请求都可以安全地移植到多个单个槽请求,如果使用不当,则会出错(例如 `PFCOUNT`)。

11.3. 使用 `RedisTemplate` 和 `ClusterOperations`

有关 通过 RedisTemplate 处理对象 的一般用途,配置和用法的信息,请参见通过 `RedisTemplate` 使用对象。

使用任何 JSON `RedisSerializer` 设置 `RedisTemplate#keySerializer` 时要小心,因为更改JSON结构会立即影响哈希槽的计算。

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

以下示例显示如何使用 `RedisTemplate` 访问 `RedisClusterConnection`:

Example 8. Accessing `RedisClusterConnection` *with* `RedisTemplate`

```
ClusterOperations clusterOps = redisTemplate.opsForCluster();
clusterOps.shutdown(NODE_7379);                                    ①
```

① Shut down node at 7379 and cross fingers there is a replica in place that can take over.

Chapter 12. Redis 存储库

通过使用 Redis 存储库,您可以无缝地在 Redis 哈希中转换和存储域对象,应用自定义映射策略以及使用二级索引.

Redis存储库至少需要 Redis Server 版本2.8.0,并且不适用于事务. 确保使用的 `RedisTemplate` 禁用了事务支持.

12.1. 使用

Spring Data Redis 使您可以轻松实现域实体,如以下示例所示:

Example 9. Sample Person Entity

```
@RedisHash("people")
public class Person {

  @Id String id;
  String firstname;
  String lastname;
  Address address;
}
```

我们在这里有一个非常简单的域对象. 请注意,它的类型上带有 `@RedisHash` 注解,并带有 `org.springframework.data.annotation.Id` 注解的名为 `id` 的属性. 这两个选项负责创建实际的 key.

用 `@Id` 注解的属性以及名为 `id` 的属性被视为标识符属性.

现在要真正拥有负责存储和检索的组件,我们需要定义一个存储库接口,如以下示例所示:

Example 10. Basic Repository Interface To Persist Person Entities

```
public interface PersonRepository extends CrudRepository<Person, String> {

}
```

随着我们的存储库扩展 `CrudRepository`,它提供了基本的 CRUD 和 finder 操作.我们需要将它们粘合在一起的是对应的 Spring 配置,如以下示例所示:

Example 11. JavaConfig for Redis Repositories

```
@Configuration
@EnableRedisRepositories
public class ApplicationConfig {

  @Bean
  public RedisConnectionFactory connectionFactory() {
    return new JedisConnectionFactory();
  }

  @Bean
  public RedisTemplate<?, ?> redisTemplate() {

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

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

Example 12. Access to Person Entities

```
@Autowired PersonRepository repo;

public void basicCrudOperations() {

  Person rand = new Person("rand", "al'thor");
  rand.setAddress(new Address("emond's field", "andor"));

  repo.save(rand);                                    ①

  repo.findOne(rand.getId());                         ②

  repo.count();                                       ③

  repo.delete(rand);                                  ④
}
```

① 如果当前值为 `null` 或已经存在 `id` 值,并在 Redis Hash 中存储类型为 Person 的属性,并使用 `keyspace:id` 的 key,则生成新的id:id-在这种情况下,可能是 `people:5d67b7e1-8640-4475-beeb-c666fab4c0e5`.

② 使用提供的 `ID` 在 `keyspace:id` 中搜索对象 .

③ 计算 keyspace 可用的实体总数,由 `@RedisHash` 在 `Person` 上定义.

④ 从Redis中删除给定对象的 key.

12.2. 对象映射基础

本节介绍了Spring Data对象映射，对象创建，字段和属性访问，可变性和不可变性的基础知识。请注意，本节仅适用于不使用基础数据存储（例如JPA）的对象映射的Spring Data模块。另外，请确保参考存储库特定的部分以获取存储库特定的对象映射，例如索引，自定义列或字段名称等。

Spring Data对象映射的核心职责是创建域对象的实例，并将存储本地数据结构映射到这些实例上。这意味着我们需要两个基本步骤：

1. 使用暴露的构造函数之一创建实例。
2. 实例填充以实现所有暴露的属性。

12.2.1. 创建对象

Spring Data会自动尝试检测要用于实现该类型对象的持久性实体的构造函数。解析算法的工作原理如下：

1. 如果有一个无参数的构造函数，它将被使用。其他构造函数将被忽略。
2. 如果只有一个构造函数接受参数，则将使用它。
3. 如果有多个接受参数的构造函数，则Spring Data要使用的那个必须用 `@PersistenceConstructor` 注解。

值解析假定构造函数参数名称与实体的属性名称匹配，即，解析将像要填充该属性一样执行，包括映射中的所有自定义项（不同的数据存储列或字段名称等）。这也需要在类文件中可用的参数名称信息或在构造函数上存在的 `@ConstructorProperties` 注解。

可以通过使用特定于存储的SpEL表达式使用Spring Framework的 `@Value` 值注解来自定义值解析。请参阅有关特定存储映射的部分以获取更多详细信息。

创建内部对象

为了避免反射的开销,Spring Data对象的创建使用默认情况下在运行时生成的工厂类,该工厂类将直接调用 domain 类的构造函数. 即 对于此示例类型:

```
class Person {
  Person(String firstname, String lastname) { … }
}
```

我们将在运行时创建一个在语义上与此工厂等效的工厂类:

```
class PersonObjectInstantiator implements ObjectInstantiator {

  Object newInstance(Object... args) {
    return new Person((String) args[0], (String) args[1]);
  }
}
```

与反射相比,这使我们的性能提高了约10%. 为了使域类有资格进行此类优化,它需要遵守一组约束:

- 它不能是 private 类
- 它不能是非静态内部类
- 它不能是 CGLib 代理类
- Spring Data 使用的构造函数不能为 private

如果这些条件中的任何一个匹配,Spring Data将通过反射回退到实体实例化.

12.2.2. 属性 population

创建实体的实例后,Spring Data会填充该类的所有剩余持久性属性. 除非实体的构造函数已经填充了该实体 (即通过其构造函数参数列表使用) ,否则将首先填充identifier属性,以允许解析循环对象引用. 之后,在实体实例上设置所有尚未由构造函数填充的非临时属性. 为此,我们使用以下算法:

1. 如果属性是不可变的,但使用 wither 方法 (请参见下文) ,则我们使用 `with…` 方法创建具有新属性值的新实体实例.
2. 如果定义了属性访问权限 (即通过 getter 和 setter 的访问) ,则我们正在调用 setter 方法.
3. 如果属性是可变的,则直接设置该字段.
4. 如果属性是不可变的,那么我们将使用持久化操作使用的构造函数 (请参见 创建对象) 来创建实例的副本.
5. 默认情况下,直接设置字段值.

Property population internals

与我们在构造对象的优化 类似,我们还使用Spring Data运行时生成的访问器类与实体实例进行交互.

```
class Person {

  private final Long id;
  private String firstname;
  private @AccessType(Type.PROPERTY) String lastname;

  Person() {
    this.id = null;
  }

  Person(Long id, String firstname, String lastname) {
    // Field assignments
  }

  Person withId(Long id) {
    return new Person(id, this.firstname, this.lastame);
  }

  void setLastname(String lastname) {
    this.lastname = lastname;
  }
}
```

Example 13. 生成的属性访问器

```
class PersonPropertyAccessor implements PersistentPropertyAccessor {

  private static final MethodHandle firstname;              ②

  private Person person;                                    ①

  public void setProperty(PersistentProperty property, Object value) {

    String name = property.getName();

    if ("firstname".equals(name)) {
      firstname.invoke(person, (String) value);             ②
    } else if ("id".equals(name)) {
      this.person = person.withId((Long) value);            ③
    } else if ("lastname".equals(name)) {
      this.person.setLastname((String) value);              ④
    }
  }
}
```

① PropertyAccessor持有底层对象的一个可变实例。这是为了允许对其他不可变的属性进行更改。

② 默认情况下,Spring数据使用字段访问来读写属性值。根据 `private` 字段的可见性规则,使用 `MethodHandles` 与字段交互。

③ 该类暴露了一个 `withId(…)` 方法,该方法用于设置标识符,例如,当一个实例被插入到数据存储并生成了一个标识符时。调用 `withId(…)` 创建一个新的 `Person` 对象。所有后续的突变都将在新实例中发生,而不影响先前的实例。

④ 使用属性访问允许直接调用方法而不使用 `MethodHandles`。

与反射相比,这使我们的性能提高了约25%。为了使域类有资格进行此类优化,它需要遵守一组约束:

- Types 不得位于默认值或java包下。
- 类型及其构造函数必须是 `public` 的
- 内部类的类型必须是静态的。
- 使用的Java运行时必须允许在原始 `ClassLoader` 中声明类。Java 9和更高版本强加了某些限制。

默认情况下,Spring Data尝试使用生成的属性访问器,如果检测到限制,则回退到基于反射的属性访问器。

让我们看一下以下实体:

Example 14. A sample entity

```
class Person {

  private final @Id Long id;                                               ①
  private final String firstname, lastname;                                ②
  private final LocalDate birthday;
  private final int age;                                                   ③

  private String comment;                                                  ④
  private @AccessType(Type.PROPERTY) String remarks;                        ⑤

  static Person of(String firstname, String lastname, LocalDate birthday) { ⑥

    return new Person(null, firstname, lastname, birthday,
      Period.between(birthday, LocalDate.now()).getYears());
  }

  Person(Long id, String firstname, String lastname, LocalDate birthday, int age)
{ ⑥

    this.id = id;
    this.firstname = firstname;
    this.lastname = lastname;
    this.birthday = birthday;
    this.age = age;
  }

  Person withId(Long id) {                                                 ①
    return new Person(id, this.firstname, this.lastname, this.birthday, this.age);
  }

  void setRemarks(String remarks) {                                        ⑤
    this.remarks = remarks;
  }
}
```

① 标识符属性是 `final`,但在构造函数中设置为 `null`. 该类暴露用于设置标识符的 withId(⋯)`方法,例如 将实例插入数据存储区并已生成标识符时. 创建新实例后,原始 `Person 实例保持不变. 通常将相同的模式应用于存储管理的其他属性,但可能需要为持久性操作进行更改.wither 方法是可选的,因为构造函数 (请参见6) 实际上是一个复制的构造函数,设置该属性将转换为使用新的值创建一个新实例.

② `firstname` 和 `lastname` 属性是可能通过getter暴露的普通不可变属性.

③ `age` 属性是一个不变的,但从 `birthday` 属性扩展的属性. 通过显示的设计,数据库值将胜过默认值,因为 Spring Data 使用唯一声明的构造函数. 即使意图是首选计算,此构造函数也必须将 `age` 作为参数 (可能会忽略它) ,这一点很重要,因为否则属性填充步骤将尝试设置 `age` 字段并由于其不可变而失败,并且没有 `with⋯` 方法存在.

④ 通过直接设置其字段可以填充 `comment` 属性是可变的.

⑤ `remarks` 属性是可变的,可通过直接设置 `comment` 字段或通过调用setter方法来填充

⑥ 该类暴露用于对象创建的工厂方法和构造函数. 这里的核心思想是使用工厂方法而不是其他构造函数,以避免通过 `@PersistenceConstructor` 消除构造函数歧义的需要.

相反,属性的默认设置是在工厂方法中处理的.

12.2.3. 一般建议

- 尝试坚持不可变的对象 — 不可变的对象很容易创建,因为实现一个对象只需调用其构造函数即可. 同样,这避免了用允许客户端代码操纵对象状态的setter方法乱扔您的域对象. 如果需要它们,则最好使它们受到程序包保护,以便只能由有限数量的同一位置类型调用它们. 仅限构造函数的实现比属性填充快30%.
- 提供一个全参数的构造函数 — 即使您不能或不希望将实体建模为不可变的值,仍然可以提供一个将实体的所有属性作为参数 (包括可变属性) 作为参数的构造函数,因为这样做可以对象映射以跳过属性填充以获得最佳性能.
- 使用工厂方法而不是重载的构造函数来避免 `@PersistenceConstructor` — 为了获得最佳性能,需要使用全参数构造函数,我们通常希望暴露更多特定于应用程序用例的构造函数,从而省略了诸如自动生成的标识符等内容. 使用静态工厂方法暴露 `all-args` 构造函数的这些变体.
- 确保您遵守允许使用生成的实例化器和属性访问器类的约束 —
- 对于要生成的标识符,仍然将 *final* 字段与全参数构造函数 `with…` 方法结合使用 —
- 使用 *Lombok* 来避免样板代码 — 由于持久性操作通常需要构造函数使用所有参数,因此它们的声明成为对字段分配的样板参数的繁琐重复,最好使用Lombok的 `@AllArgsConstructor` 来避免.

12.2.4. Kotlin 支持

Spring Data 修改了 Kotlin 的细节以允许对象创建和变异.

Kotlin 对象创建

支持实例化 Kotlin 类,默认情况下所有类都是不可变的,并且需要显式属性声明来定义可变属性. 考虑以下 `data` 类 `Person`:

```
data class Person(val id: String, val name: String)
```

上面的类使用显式构造函数编译为典型类. 我们可以通过添加另一个构造函数来定制该类,并使用 `@PersistenceConstructor` 对其进行注解以指示构造函数的首选项:

```
data class Person(var id: String, val name: String) {

    @PersistenceConstructor
    constructor(id: String) : this(id, "unknown")
}
```

Kotlin通过允许在未提供参数的情况下使用默认值来支持参数的可选性. 当Spring Data检测到带有参数默认值的构造函数时,如果数据存储区不提供值 (或简单地返回 `null`) ,则它将使这些参数不存在,因此Kotlin可以应用参数默认值. 考虑下面的类,该类将参数默认值用作 `name`

```
data class Person(var id: String, val name: String = "unknown")
```

每次 `name` 参数不是结果的一部分或值为 `null` 时, name` 默认为 `unknown`.

Property population of Kotlin data classes

在Kotlin中,所有类默认都是不可变的,并且需要显式的属性声明来定义可变属性. 考虑以下 `data` 类Person:

```
data class Person(val id: String, val name: String)
```

该类实际上是不可变的. 当Kotlin生成 `copy(…)` 方法时,它可以创建新的实例,该方法创建新的对象实例,该对象实例从现有对象复制所有属性值,并将作为参数提供的属性值应用于该方法.

12.3. 对象到哈希的映射

Redis 存储库支持将对象保留为哈希. 这需要由 `RedisConverter` 完成的对象到哈希的转换. 默认实现使用 `Converter` 将属性值与 Redis 本地 `byte[]` 之间进行映射.

给定前面几节中的 `Person` 类型,默认映射如下所示:

```
_class = org.example.Person                    ①
id = e2c7dcee-b8cd-4424-883e-736ce564363e
firstname = rand                               ②
lastname = al'thor
address.city = emond's field                   ③
address.country = andor
```

① `_class` 属性包含在根级别以及任何嵌套接口或抽象类型中.

② 简单属性值按路径映射.

③ 复杂类型的属性按其点路径进行映射.

下表描述了默认映射规则:

Table 10. 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] = "…

Type	Sample	Mapped Value
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 = "…

由于采用平面表示结构,因此 `Map` 的 key 必须是简单的类型,例如 `String` 或 `Number`.

可以通过在 `RedisCustomConversions` 中注册相应的 `Converter` 来自定义映射行为。这些转换器可以处理与单个 `byte[]` 和 `Map<String,byte[]>` 之间的转换。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表示形式。第二个选项提供对生成的哈希的完全控制。

将对象写入 Redis 哈希会删除哈希中的内容,并重新创建整个哈希,因此尚未映射的数据会丢失。

下面的示例显示了两个示例字节数组转换器:

Example 15. Sample byte[] Converters

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

  private final Jackson2JsonRedisSerializer<Address> serializer;

  public AddressToBytesConverter() {

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

  @Override
  public byte[] convert(Address value) {
    return serializer.serialize(value);
  }
}

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

  private final Jackson2JsonRedisSerializer<Address> serializer;

  public BytesToAddressConverter() {

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

  @Override
  public Address convert(byte[] value) {
    return serializer.deserialize(value);
  }
}
```

使用前面的字节数组 `Converter` 产生的输出类似于以下内容：

```
_class = org.example.Person
id = e2c7dcee-b8cd-4424-883e-736ce564363e
firstname = rand
lastname = al'thor
address = { city : "emond's field", country : "andor" }
```

以下示例显示了两个 `Map` 转换器示例：

Example 16. Sample Map<String,byte[]> Converters

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

  @Override
  public Map<String,byte[]> convert(Address source) {
    return singletonMap("ciudad", source.getCity().getBytes());
  }
}

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

  @Override
  public Address convert(Map<String,byte[]> source) {
    return new Address(new String(source.get("ciudad")));
  }
}
```

使用前面的 Map Converter 产生的输出类似于以下内容:

```
_class = org.example.Person
id = e2c7dcee-b8cd-4424-883e-736ce564363e
firstname = rand
lastname = al'thor
ciudad = "emond's field"
```

自定义转换对索引解析没有影响. 即使对于自定义转换类型, 二级索引 仍会创建.

12.3.1. 自定义类型映射

如果要避免将整个 Java 类名称写为类型信息,并且希望使用键,则可以在要保留的实体类上使用 @TypeAlias 注解. 如果您需要进一步自定义映射,请查看 TypeInformationMapper 接口. 可以在 DefaultRedisTypeMapper 上配置该接口的实例,该实例可以在 MappingRedisConverter 上配置.

以下示例显示如何为实体定义类型别名:

Example 17. 为实体定义 `@TypeAlias`

```
@TypeAlias("pers")
class Person {

}
```

生成的文档将 `pers` 作为 `_class` 字段中的值.

配置自定义类型映射

以下示例演示如何在 `MappingRedisConverter` 中配置自定义 `RedisTypeMapper`:

Example 18. 通过*Spring Java Config*配置自定义 `RedisTypeMapper`

```
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();
  }
}
```

12.4. Keyspaces

Keyspaces 定义用于为 Redis 哈希创建实际 key 的前缀.

默认情况下,前缀设置为 `getClass().getName()`. 您可以通过在聚合根级别上设置 `@RedisHash` 或设置程序配置来更改此默认设置. 但是,带注解的 keyspaces 将取代任何其他配置.

下面的示例演示如何使用 `@EnableRedisRepositories` 注解设置密钥空间配置:

Example 19. 通过 `@EnableRedisRepositories` 进行 *Keyspaces* 设置

```
@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"));
    }
  }
}
```

下面的示例演示如何以编程方式设置 keyspace:

Example 20. Programmatic Keyspace setup

```
@Configuration
@EnableRedisRepositories
public class ApplicationConfig {

  //... RedisConnectionFactory and RedisTemplate Bean definitions omitted

  @Bean
  public RedisMappingContext keyValueMappingContext() {
    return new RedisMappingContext(
      new MappingConfiguration(new IndexConfiguration(), new
MyKeyspaceConfiguration()));
  }

  public static class MyKeyspaceConfiguration extends KeyspaceConfiguration {

    @Override
    protected Iterable<KeyspaceSettings> initialConfiguration() {
      return Collections.singleton(new KeyspaceSettings(Person.class, "people"));
    }
  }
}
```

12.5. 二级索引

二级索引 用于启用基于 本地 Redis结构的查找操作. 每次保存时,值都会写入相应的索引,并在删除或 过期 对象时将其删除..

12.5.1. 简单属性索引

给定前面显示的示例 `Person` 实体,我们可以通过使用 `@Indexed` 注解属性来为 `firstname` 创建索引,如以下示例所示:

Example 21. Annotation driven indexing

```
@RedisHash("people")
public class Person {

  @Id String id;
  @Indexed String firstname;
  String lastname;
  Address address;
}
```

为实际属性值建立索引. 保存两个人(例如,"rand" 和 "aviendha")会导致建立类似于以下内容的索引:

```
SADD people:firstname:rand e2c7dcee-b8cd-4424-883e-736ce564363e
SADD people:firstname:aviendha a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56
```

也可以在嵌套元素上使用索引.假定 `Address` 具有以 `@Indexed` 注解的 `city` 属性.在这种情况下,一旦 `person.address.city` 不为 `null`,我们将为每个城市设置 `Set`,如以下示例所示:

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

此外,通过编程设置,您可以定义 map keys 和列表属性的索引,如以下示例所示:

```
@RedisHash("people")
public class Person {

  // ... other properties omitted

  Map<String,String> attributes;      ①
  Map<String Person> relatives;       ②
  List<Address> addresses;            ③
}
```

① `SADD people:attributes.map-key:map-value e2c7dcee-b8cd-4424-883e-736ce564363e`

② `SADD people:relatives.map-key.firstname:tam e2c7dcee-b8cd-4424-883e-736ce564363e`

③ `SADD people:addresses.city:tear e2c7dcee-b8cd-4424-883e-736ce564363e`

索引无法在 References 上解析.

与 keyspaces 一样,您可以配置索引而无需注解实际的 domain ,如以下示例所示:

Example 22. 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"));
    }
  }
}
```

再次,与 keyspaces 一样,您可以以编程方式配置索引,如以下示例所示:

Example 23. 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"));
    }
  }
}
```

12.5.2. 地理空间索引

假设 `Address` 类型包含 `Point` 类型的 `location` 属性,其中包含特定地址的地理坐标. 通过使用 `@GeoIndexed` 注解属性,Spring Data Redis 使用 Redis GEO 命令添加这些值,如以下示例所示:

```
@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);     ①
  List<Person> findByAddressLocationWithin(Circle circle);                    ②
}

Person rand = new Person("rand", "al'thor");
rand.setAddress(new Address(new Point(13.361389D, 38.115556D)));

repository.save(rand);                                                        ③

repository.findByAddressLocationNear(new Point(15D, 37D), new Distance(200)); ④
```

① 使用 `Point` 和 `Distance` 在嵌套属性上的查询方法声明, .

② 嵌套属性上的查询方法声明, 使用 `Circle` 进行搜索.

③ `GEOADD people:address:location 13.361389 38.115556 e2c7dcee-b8cd-4424-883e-736ce564363e`

④ `GEORADIUS people:address:location 15.0 37.0 200.0 km`

在前面的示例中,经度和纬度值是通过使用 `GEOADD` 存储的,而 `GEOADD` 使用对象的 `id` 作为成员的名称.查找器方法允许使用 `Circle` 或 `Point, Distance` 组合查询这些值.

不可能将附近和内部与其他条件组合.

12.6. 查询例子

12.6.1. 简介

本章对 "按示例查询" 进行了介绍,并说明了如何使用它.

示例查询 (QBE) 是一种具有简单界面的用户友好查询技术. 它允许动态查询创建,并且不需要您编写包含字段名称的查询. 实际上,"示例查询" 根本不需要您使用存储库特定的查询语言编写查询.

12.6.2. 使用

示例查询由三部分组成:

- Probe: 带有填充字段的 domain 对象的实际示例.
- `ExampleMatcher`: `ExampleMatcher` 包含有关如何匹配特定字段的详细信息. 可以在多个示例中重复使用它.
- `Example`: 示例包括探针和 `ExampleMatcher`. 它用于创建查询.

示例查询非常适合几种用例:

- 使用一组静态或动态约束来查询数据存储.
- 频繁重构 domain 对象,而不必担心破坏现有查询.
- 独立于基础数据存储 API 进行工作.

示例查询也有一些限制:

- 不支持嵌套或分组属性约束,例如 `firstname = ?0 or (firstname = ?1 and lastname = ?2)`.
- 仅支持字符串的开始/包含/结束/正则表达式匹配,以及其他属性类型的完全匹配.

在开始使用示例查询之前,您需要具有一个 domain 对象. 首先,为您的存储库创建一个接口,如以下示例所示:

Example 24. Sample Person 对象

```
public class Person {

  @Id
  private String id;
  private String firstname;
  private String lastname;
  private Address address;

  // … getters and setters omitted
}
```

前面的示例显示了一个简单的域对象. 您可以使用它来创建一个 `Example`. 默认情况下,具有 `null` 的字段将被忽略,并且使用存储特定的默认值来匹配字符串. 可以使用工厂方法或使用 `ExampleMatcher` 构建示例. 例子是一成不变的. 以下清单显示了一个简单的示例:

Example 25. Simple Example

```
Person person = new Person();                         ①
person.setFirstname("Dave");                          ②

Example<Person> example = Example.of(person);         ③
```

① 创建 domain 对象的新实例.

② 设置要查询的属性.

③ 创建 `Example`.

最好在存储库中执行示例。为此,让您的存储库接口扩展 `QueryByExampleExecutor<T>`。以下清单显示了 `QueryByExampleExecutor` 接口:

Example 26. `QueryByExampleExecutor`

```
public interface QueryByExampleExecutor<T> {

  <S extends T> S findOne(Example<S> example);

  <S extends T> Iterable<S> findAll(Example<S> example);

  // … more functionality omitted.
}
```

12.6.3. Example 匹配

示例不限于默认设置。您可以使用 `ExampleMatcher` 为字符串匹配,空值处理和特定于属性的设置指定自己的默认值,如以下示例所示:

Example 27. 具有定制匹配的例子匹配器

```
Person person = new Person();                          ①
person.setFirstname("Dave");                           ②

ExampleMatcher matcher = ExampleMatcher.matching()     ③
  .withIgnorePaths("lastname")                         ④
  .withIncludeNullValues()                             ⑤
  .withStringMatcherEnding();                          ⑥

Example<Person> example = Example.of(person, matcher); ⑦
```

① 创建域对象的新实例.

② 设置属性.

③ 创建一个 `ExampleMatcher` 以期望所有值都匹配. 即使没有进一步的配置,它也可以在此阶段使用.

④ 构造一个新的 `ExampleMatcher` 以忽略 `lastname` 属性路径.

⑤ 构造一个新的 `ExampleMatcher` 以忽略 `lastname` 属性路径并包含空值.

⑥ 构造一个新的 `ExampleMatcher` 以忽略 `lastname` 属性路径,包括空值,并执行后缀字符串匹配.

⑦ 基于域对象和配置的 `ExampleMatcher` 创建一个新的 `Example`.

默认情况下,`ExampleMatcher` 期望设置的所有值都匹配. 如果要获取与隐式定义的任何断言匹配的结果,请使用 `ExampleMatcher.matchingAny()`.

您可以为单个属性 (例如 "firstname" 和 "lastname",或者对于嵌套属性,"address.city") 指定行为. 您可以使用匹配选项和区分大小写对其进行调整,如以下示例所示:

Example 28. 配置匹配器选项

```
ExampleMatcher matcher = ExampleMatcher.matching()
  .withMatcher("firstname", endsWith())
  .withMatcher("lastname", startsWith().ignoreCase());
}
```

配置匹配器选项的另一种方法是使用 lambda (在Java 8中引入) . 此方法创建一个回调,要求实现者修改匹配器. 您无需返回匹配器,因为配置选项保存在匹配器实例中. 以下示例显示了使用lambda的匹配器:

Example 29. 用*lambdas*配置匹配器选项

```
ExampleMatcher matcher = ExampleMatcher.matching()
  .withMatcher("firstname", match -> match.endsWith())
  .withMatcher("firstname", match -> match.startsWith());
}
```

由 `Example` 创建的查询使用配置的合并视图. 可以在 `ExampleMatcher` 级别上设置默认的匹配设置,而可以将单个设置应用于特定的属性路径. 除非明确定义,否则

`ExampleMatcher` 上设置的设置将由属性路径设置继承。属性修补程序上的设置优先于默认设置。下表描述了各种 `ExampleMatcher` 设置的范围：

表4. `ExampleMatcher` 设置的范围

Table 11. Scope of `ExampleMatcher` settings

Setting	Scope
Null-handling	`ExampleMatcher`
String matching	`ExampleMatcher` and property path
Ignoring properties	Property path
Case sensitivity	`ExampleMatcher` and property path
Value transformation	Property path

12.6.4. 执行一个例子

以下示例对存储库使用“按示例查询”：

Example 30. 使用存储库按示例查询

```
interface PersonRepository extends QueryByExampleExecutor<Person> {
}

class PersonService {

  @Autowired PersonRepository personRepository;

  List<Person> findPeople(Person probe) {
    return personRepository.findAll(Example.of(probe));
  }
}
```

Redis 存储库及其二级索引支持 Spring Data 的按示例查询功能的子集。特别是,仅使用精确的,区分大小写的和非 `null` 的值来构造查询。

二级索引使用基于集合的操作(集合相交,集合并集)来确定匹配的键。 向没有索引的查询添加属性不会返回任何结果,因为不存在索引。 “按示例查询” 支持人员检查索引配置,以仅包括查询中由索引覆盖的属性。这是为了防止意外包含未索引的属性。

不区分大小写的查询和不受支持的 `StringMatcher` 实例在运行时被拒绝。

以下列表显示了受支持的“按示例查询”选项：

- 区分大小写,简单和嵌套属性的精确匹配
- Any/All 匹配模式
- 标准值的值转换
- 从条件中排除 `null` 值

以下列表显示了 “按示例查询” 不支持的属性：

- 不区分大小写的匹配
- 正则表达式,前缀/包含/后缀字符串匹配
- 查询关联,集合和类似地图的属性
- 从条件中包含 `null` 值
- `findAll` 与排序

12.7. 生存时间

Redis 中存储的对象可能仅在一定时间内有效。这对于在 Redis 中持久保存短寿命的对象特别有用,而无需在寿命到期时手动将其删除。可以使用 `@RedisHash(timeToLive=…)` 以及使用 `KeyspaceSettings`(请参见 Keyspaces)来设置以秒为单位的到期时间。

可以通过在数字属性或方法上使用 `@TimeToLive` 注解来设置更灵活的到期时间。 但是,不要将 `@TimeToLive` 应用于同一类中的方法和属性。 以下示例显示了属性和方法上的 `@TimeToLive` 注解:

Example 31. 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();
  }
}
```

使用 `@TimeToLive` 显式注解属性会从 Redis 中读取实际的 `TTL` 或 `PTTL` 值。-1 表示该对象没有关联的到期时间。

存储库实现可确保通过 `RedisMessageListenerContainer` 订阅 Redis keyspace notifications

当到期设置为正值时,将执行相应的 `EXPIRE` 命令。 除了保留原始副本外,幻影副本还将保留在 Redis 中,并设置为在原始副本后五分钟到期。 这样做是为了使存储库支持能够发布 `RedisKeyExpiredEvent`,并在键过期时将过期值保存在Spring的 `ApplicationEventPublisher` 中, 即使原始值已被删除也是如此。 在使用 Spring Data Redis 存储库的所有已连接应用程序上都会收到到期事件。

默认情况下,初始化应用程序时禁用密钥过期监听器。 可以在 `@EnableRedisRepositories` 或 `RedisKeyValueAdapter` 中调整启动模式,以通过应用程序或在第一次插入带有TTL的实体时启动监听器。 有关可能的值,请参见 `EnableKeyspaceEvents`。

RedisKeyExpiredEvent 包含过期的域对象以及密钥的副本.

延迟或禁用到期事件监听器的启动会影响 RedisKeyExpiredEvent 发布.禁用的事件监听器不会发布到期事件.由于延迟的监听器初始化,延迟的启动可能导致事件丢失.

keyspace notification 消息监听器会更改Redis中的 notify-keyspace-events 设置(如果尚未设置). 现有设置不会被覆盖,因此您必须正确设置这些设置(或将其保留为空). 请注意,在AWS ElastiCache 上禁用了 CONFIG,并且启用监听器会导致错误.

Redis Pub/Sub messages 消息不是持久性的.如果在应用程序关闭时密钥过期,则不会处理过期事件,这可能会导致二级索引包含对过期对象的引用.

12.8. 持久引用

使用 @Reference 标记属性允许存储简单的键引用,而不是将值复制到哈希本身中.从 Redis 加载时,引用将自动解析并映射回该对象,如以下示例所示:

Example 32. Sample 属性参考

```
_class = org.example.Person
id = e2c7dcee-b8cd-4424-883e-736ce564363e
firstname = rand
lastname = al'thor
mother = people:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56      ①
```

① Reference stores the whole key (keyspace:id) of the referenced object.

保存引用对象时,引用对象不会保留.由于仅存储引用,因此您必须分别对引用的对象进行更改.无法解析在引用类型的属性上设置的索引

12.9. 持续部分更新

在某些情况下,您无需加载和重写整个实体就可以在其中设置新值. 最后一个活动时间的会话时间戳可能就是您要更改一个属性的情况. PartialUpdate 使您可以定义现有对象的 set 和 delete 操作,同时注意更新实体本身和索引结构的潜在到期时间. 以下示例显示了部分更新:

Example 33. Sample Partial Update

```
PartialUpdate<Person> update = new PartialUpdate<Person>("e2c7dcee", Person.class)
  .set("firstname", "mat")
①
  .set("address.city", "emond's field")
②
  .del("age");
③

template.update(update);

update = new PartialUpdate<Person>("e2c7dcee", Person.class)
  .set("address", new Address("caemlyn", "andor"))
④
  .set("attributes", singletonMap("eye-color", "grey"));
⑤

template.update(update);

update = new PartialUpdate<Person>("e2c7dcee", Person.class)
  .refreshTtl(true);
⑥
  .set("expiration", 1000);

template.update(update);
```

① 将 `firstname` 属性设置为 `mat`.

② 将 'address.city' 属性设置为 'emond’s field' 而不必传入整个对象. 注册自定义转换时,此功能不起作用.

③ 删除 `age` 属性.

④ 设置复杂的 `address` 属性.

⑤ 设置值的映射,该映射将删除先前存在的映射,并将值替换为给定的值.

⑥ 更改生存时间时,将自动更新服务器的到期时间.Automatically update the server expiration time when altering .

更新复杂对象以及地图(或其他集合)结构需要与Redis进行进一步交互以确定现有值,这意味着重写整个实体可能会更快.

12.10. 查询和查询方法

查询方法允许从方法名称自动继承 简单查找程序查询,如以下示例所示:

Example 34. Sample Repository finder Method

```
public interface PersonRepository extends CrudRepository<Person, String> {

  List<Person> findByFirstname(String firstname);
}
```

请确保设置查找程序方法中使用的属性以建立索引。

Redis存储库的查询方法仅支持对实体的查询以及带分页的实体集合。

使用继承 的查询方法可能并不总是足以为要执行的查询建模。 `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，提供一个 `RedisCallback`，它返回一组或一组可迭代的 `id` 值，如以下示例所示：

Example 35. Sample finder using RedisCallback

```
String user = //...

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

  public Set<byte[]> doInRedis(RedisConnection connection) throws
DataAccessException {
    return connection
      .sMembers("sessions:securityContext.authentication.principal.username:" +
user);
  }}, RedisSession.class);
```

下表概述了Redis支持的关键字以及包含该关键字的方法的基本含义：

Table 12. 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
IsTrue	FindByAliveIsTrue	SINTER …:alive:1
IsFalse	findByAliveIsFalse	SINTER …:alive:0
Top,First	findFirst10ByFirstname,findTop5ByFirstname	

12.11. 在集群上运行的Redis存储库

您可以在集群 Redis 环境中使用 Redis 存储库支持. 有关 ConnectionFactory 配置的详细信息,请参见 "Redis 集群" 部分. 但是,还必须进行一些其他配置,因为默认的密钥分发会在整个群集及其 slot 中扩展实体和二级索引.

下表显示了群集上数据的详细信息(基于先前的示例):

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

当所有涉及的键都映射到同一 slot 时,某些命令(例如 SINTER 和 SUNION)只能在服务器端处理. 否则,必须在客户端进行计算. 因此,将 keyspaces 固定到单个 slot 非常有用,这样可以立即利用 Redis 服务器端计算. 下表显示了您执行此操作时发生的情况(请注意 slot 列中的更改以及节点列中的端口值):

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

使用Redis集群时,通过使用 @RedisHash("{yourkeyspace}") 将 keyspaces 定义和固定到特定 slots.

12.12. CDI 集成

存储库接口的实例通常由容器创建,在使用 Spring Data 时,Spring 是最自然的选择. Spring 提供了用于创建bean实例的高级工具. Spring Data Redis 附带了一个自定义 CDI 扩展,使您可以在 CDI 环境中使用存储库抽象. 该扩展是JAR的一部分,因此要激活它,请将Spring Data Redis JAR放到您的类路径中.

然后,您可以通过为 RedisConnectionFactory 和 RedisOperations 实现CDI生产者来设置基础结构,如以下示例所示:

```
class RedisOperationsProducer {


  @Produces
  RedisConnectionFactory redisConnectionFactory() {

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

    return jedisConnectionFactory;
  }

  void disposeRedisConnectionFactory(@Disposes RedisConnectionFactory
redisConnectionFactory) throws Exception {

    if (redisConnectionFactory instanceof DisposableBean) {
      ((DisposableBean) redisConnectionFactory).destroy();
    }
  }

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

    RedisTemplate<byte[], byte[]> template = new RedisTemplate<byte[], byte[]>();
    template.setConnectionFactory(redisConnectionFactory);
    template.afterPropertiesSet();

    return template;
  }

}
```

必要的设置可能会有所不同,具体取决于您的 JavaEE 环境.

每当容器请求存储库类型的bean时,Spring Data Redis CDI 扩展都将所有可用的存储库用作 CDI bean,并为 Spring Data 存储库创建代理. 因此,获取 Spring Data 存储库的实例只需声明一个 @Injected 属性即可,如以下示例所示:

```
class RepositoryClient {

  @Inject
  PersonRepository repository;

  public void businessMethod() {
    List<Person> people = repository.findAll();
  }
}
```

Redis存储库需要 `RedisKeyValueAdapter` 和 `RedisKeyValueTemplate` 实例. 如果找不到提供的 bean,那么这些 bean 由 Spring Data CDI 扩展创建和管理. 但是,您可以提供自己的 bean 来配置 `RedisKeyValueAdapter` 和 `RedisKeyValueTemplate` 的特定属性.

12.13. Redis存储库原理

Redis作为村书库本身提供了一个非常狭窄的低级API,留给用户的是高级功能,例如二级索引和查询操作.

本节提供了由存储库抽象发出的命令的更详细视图,以更好地了解潜在的性能影响.

将以下实体类视为所有操作的起点:

Example 36. Example entity

```
@RedisHash("people")
public class Person {

  @Id String id;
  @Indexed String firstname;
  String lastname;
  Address hometown;
}

public class Address {

  @GeoIndexed Point location;
}
```

12.13.1. 插入新值

```
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" ①
SADD  "people" "19315449-cda2-4f5c-b696-9cb8018fa1f9"                           ②
SADD  "people:firstname:rand" "19315449-cda2-4f5c-b696-9cb8018fa1f9"            ③
SADD  "people:19315449-cda2-4f5c-b696-9cb8018fa1f9:idx" "people:firstname:rand" ④
```

① 将展开的条目另存为哈希.

② 将用 <1> 编写的哈希键添加到同一 keyspaces 中的实体的辅助索引中.

③ 将用<2>编写的哈希键添加到具有属性值的名字的二级索引中.

④ 将<3>的索引添加到帮助程序结构集以供输入,以跟踪删除/更新时要清理的索引.

12.13.2. 替换存在的

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

```
DEL       "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"
①
HMSET     "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9" "_class" "Person" "id"
"e82908cf-e7d3-47c2-9eec-b4e0967ad0c9" "firstname" "Dragon Reborn" "lastname"
"al'thor" ②
SADD      "people" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"
③
SMEMBERS  "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9:idx"
④
TYPE      "people:firstname:rand"
⑤
SREM      "people:firstname:rand" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"
⑥
DEL       "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9:idx"
⑦
SADD      "people:firstname:Dragon Reborn" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"
⑧
SADD      "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9:idx"
"people:firstname:Dragon Reborn" ⑨
```

① 删除现有的散列,以避免可能不再存在的散列键剩余.

② 将展开的条目另存为哈希.

③ 将用<1>编写的哈希键添加到同一 keyspaces 中的实体的辅助索引中.

④ 获取可能需要更新的现有索引结构.

⑤ 检查索引是否存在以及它是什么类型(文本,地理位置,...).

⑥ 从索引中删除潜在的现有键.

⑦ 删除辅助程序持有索引信息.

⑧ 将在<2>中添加的哈希键添加到具有属性值的名字的二级索引中.

⑨ 将索引<6>添加到帮助程序结构集中,以进行输入,以跟踪删除/更新时要清除的索引.

12.13.3. 保存 Geo 数据

地理索引遵循与基于普通文本的规则相同的规则,但是使用地理结构来存储值. 保存使用地理索引属性的实体将导致以下命令:

```
GEOADD "people:hometown:location" "13.361389" "38.115556" "76900e94-b057-44bc-
abcf-8126d51a621b"  ①
SADD   "people:76900e94-b057-44bc-abcf-8126d51a621b:idx"
"people:hometown:location"              ②
```

① 将已保存条目的键添加到地理位置索引

② 跟踪索引结构.

12.13.4. 使用简单索引查找

```
repository.findByFirstname("egwene");
```

```
SINTER  "people:firstname:egwene"                    ①
HGETALL "people:d70091b5-0b9a-4c0a-9551-519e61bc9ef3" ②
HGETALL ...
```

① 提取二级索引中包含的键.

② 分别获取<1>返回的每个键.

12.13.5. 使用地理位置索引查找

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

```
GEORADIUS "people:hometown:location" "15.0" "37.0" "200.0" "km" ①
HGETALL   "people:76900e94-b057-44bc-abcf-8126d51a621b"        ②
HGETALL   ...
```

① 提取二级索引中包含的键.

② 分别获取<1>返回的每个键.

附录

Appendix 文档结构

附录包含各种其他详细信息,可补充参考文档其余部分中的信息:

- “Schema” 定义了 Spring Data Redis 提供的 schema.
- “命令参考” 详细介绍了 RedisTemplate 支持哪些命令.

Appendix A: Schema

Spring Data Redis Schema (redis-namespace)

Appendix B: □□□□

□□□□□

Table 13. Redis commands supported by `RedisTemplate`

Command	Template Support
APPEND	X
AUTH	X
BGREWRITEAOF	X
BGSAVE	X
BITCOUNT	X
BITFIELD	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

Command	Template Support
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

Command	Template Support
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	-

Command	Template Support
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

Command	Template Support
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

Command	Template Support
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