Redis高级客户端Lettuce详解

前提

`Lettuce` 是一个 `Redis` 的 `Java` 驱动包，初识她的时候是使用 `RedisTemplate` 的时候遇到点问题 `Debug` 到底层的一些源码，发现 `spring-data-redis` 的驱动包在某个版本之后替换为 `Lettuce` 。 `Lettuce` 翻译为 生菜 ，没错，就是吃的那种生菜，所以它的 `Logo` 长这样：

既然能被 `Spring` 生态所认可， `Lettuce` 想必有过人之处，于是笔者花时间阅读她的官方文档，整理测试示例，写下这篇文章。编写本文时所使用的版本为 `Lettuce 5.1.8.RELEASE` ， `SpringBoot 2.1.8.RELEASE` ， `JDK [8,11]` 。超长警告：这篇文章断断续续花了两周完成，超过4万字.....

Lettuce简介

`Lettuce` 是一个高性能基于 `Java` 编写的 `Redis` 驱动框架，底层集成了 `Project Reactor` 提供天然的反应式编程，通信框架集成了 `Netty` 使用了非阻塞 `IO` ， `5.x` 版本之后融合了 `JDK1.8` 的异步编程特性，在保证高性能的同时提供了十分丰富易用的 `API` ， `5.1` 版本的新特性如下：

- 支持 `Redis` 的新增命令 `ZPOPMIN, ZPOPMAX, BZPOPMIN, BZPOPMAX` 。
- 支持通过 `Brave` 模块跟踪 `Redis` 命令执行。
- 支持 `Redis Streams` 。
- 支持异步的主从连接。
- 支持异步连接池。
- 新增命令最多执行一次模式（禁止自动重连）。
- 全局命令超时设置（对异步和反应式命令也有效）。
- .....等等

注意一点： `Redis` 的版本至少需要 `2.6` ，当然越高越好， `API` 的兼容性比较强大。

只需要引入单个依赖就可以开始愉快地使用 `Lettuce` ：

- Maven

```
<dependency>
    <groupId>io.lettuce</groupId>
    <artifactId>lettuce-core</artifactId>
    <version>5.1.8.RELEASE</version>
</dependency>
```

- Gradle

```
dependencies {
  compile 'io.lettuce:lettuce-core:5.1.8.RELEASE'
}
```

连接Redis

单机、哨兵、集群模式下连接 `Redis` 需要一个统一的标准去表示连接的细节信息，在 `Lettuce` 中这个统一的标准是 `RedisURI` 。可以通过三种方式构造一个 `RedisURI` 实例：

- 定制的字符串 `URI` 语法：

```
RedisURI uri = RedisURI.create("redis://localhost/");
```

- 使用建造器（ `RedisURI.Builder` ）：

```
RedisURI uri = RedisURI.builder().withHost("localhost").withPort(6379).build();
```

- 直接通过构造函数实例化：

```
RedisURI uri = new RedisURI("localhost", 6379, 60, TimeUnit.SECONDS);
```

定制的连接URI语法

- 单机（前缀为 `redis://` ）

```
格式:redis://[password@]host[:port][/databaseNumber][?[timeout=timeout[d|h|m|s|ms|us|ns]]
完整:redis://mypassword@127.0.0.1:6379/0?timeout=10s
简单:redis://localhost
```

- 单机并且使用 `SSL` （前缀为 `rediss://` ） <== 注意后面多了个 `s`

```
格式:rediss://[password@]host[:port][/databaseNumber][?[timeout=timeout[d|h|m|s|ms|us|ns]]
完整:rediss://mypassword@127.0.0.1:6379/0?timeout=10s
简单:rediss://localhost
```

- 单机 `Unix Domain Sockets` 模式（前缀为 `redis-socket://` ）

```
格式:redis-socket://path[?[timeout=timeout[d|h|m|s|ms|us|ns]][&_database=database_]]
完整:redis-socket:///tmp/redis?timeout=10s&_database=0
```

- 哨兵（前缀为 `redis-sentinel://` ）

```
格式:redis-sentinel://[password@]host[:port][,host2[:port2]][/databaseNumber][?[timeout=timeout[d|h|m|s|ms|us|ns]]#sentinelMasterId
完整:redis-sentinel://mypassword@127.0.0.1:6379,127.0.0.1:6380/0?timeout=10s#mymaster
```

超时时间单位：

- d 天
- h 小时
- m 分钟
- s 秒钟
- ms 毫秒
- us 微秒
- ns 纳秒

个人建议使用 `RedisURI` 提供的建造器，毕竟定制的 `URI` 虽然简洁，但是比较容易出现人为错误。鉴于笔者没有 `SSL` 和 `Unix Domain Socket` 的使用场景，下面不对这两种连接方式进行列举。

基本使用

`Lettuce` 使用的时候依赖于四个主要组件：

- `RedisURI` ：连接信息。
- `RedisClient` ： `Redis` 客户端，特殊地，集群连接有一个定制的 `RedisClusterClient` 。
- `Connection` ： `Redis` 连接，主要是 `StatefulConnection` 或者 `StatefulRedisConnection` 的子类，连接的类型主要由连接的具体方式（单机、哨兵、集群、订阅发布等等）选定，比较重要。
- `RedisCommands` ： `Redis` 命令 `API` 接口，基本上覆盖了 `Redis` 发行版本的所有命令，提供了同步（ `sync` ）、异步（ `async` ）、反应式（ `reative` ）的调用方式，对于使用者而言，会经常跟 `RedisCommands` 系列接口打交道。

一个基本使用例子如下：

```java
@Test
public void testSetGet() throws Exception {
    RedisURI redisUri = RedisURI.builder()                  // <1> 创建单机连接的连接信息
            .withHost("localhost")
            .withPort(6379)
            .withTimeout(Duration.of(10, ChronoUnit.SECONDS))
            .build();
    RedisClient redisClient = RedisClient.create(redisUri);   // <2> 创建客户端
    StatefulRedisConnection<String, String> connection = redisClient.connect();     // <3> 创建线程安全的连接
    RedisCommands<String, String> redisCommands = connection.sync();                // <4> 创建同步命令
    SetArgs setArgs = SetArgs.Builder.nx().ex(5);
    String result = redisCommands.set("name", "throwable", setArgs);
    Assertions.assertThat(result).isEqualToIgnoringCase("OK");
    result = redisCommands.get("name");
    Assertions.assertThat(result).isEqualTo("throwable");
    // ... 其他操作
    connection.close();   // <5> 关闭连接
    redisClient.shutdown();  // <6> 关闭客户端
}
```

注意：

- <5>：关闭连接一般在应用程序停止之前操作，一个应用程序中的一个 `Redis` 驱动实例不需要太多的连接（一般情况下只需要一个连接实例就可以，如果有多个连接的需要可以考虑使用连接池，其实 `Redis` 目前处理命令的模块是单线程，在客户端多个连接多线程调用理论上没有效果）。
- <6>：关闭客户端一般应用程序停止之前操作，如果条件允许的话，基于 后开先闭 原则，客户端关闭应该在连接关闭之后操作。

API

`Lettuce` 主要提供三种 `API`：

- 同步（`sync`）：`RedisCommands`。
- 异步（`async`）：`RedisAsyncCommands`。
- 反应式（`reactive`）：`RedisReactiveCommands`。

先准备好一个单机 `Redis` 连接备用：

```java
private static StatefulRedisConnection<String, String> CONNECTION;
private static RedisClient CLIENT;

@BeforeClass
public static void beforeClass() {
    RedisURI redisUri = RedisURI.builder()
            .withHost("localhost")
            .withPort(6379)
            .withTimeout(Duration.of(10, ChronoUnit.SECONDS))
            .build();
    CLIENT = RedisClient.create(redisUri);
    CONNECTION = CLIENT.connect();
}

@AfterClass
public static void afterClass() throws Exception {
    CONNECTION.close();
    CLIENT.shutdown();
}
```

`Redis` 命令 `API` 的具体实现可以直接从 `StatefulRedisConnection` 实例获取，见其接口定义：

```java
public interface StatefulRedisConnection<K, V> extends StatefulConnection<K, V> {

    boolean isMulti();

    RedisCommands<K, V> sync();

    RedisAsyncCommands<K, V> async();

    RedisReactiveCommands<K, V> reactive();
}
```

值得注意的是，在不指定编码解码器 `RedisCodec` 的前提下，`RedisClient` 创建的 `StatefulRedisConnection` 实例一般是泛型实例 `StatefulRedisConnection<String,String>`，也就是所有命令 `API` 的 `KEY` 和 `VALUE` 都是 `String` 类型，这种使用方式能满足大部分的使用场景。当然，必要的时候可以定制编码解码器 `RedisCodec<K,V>`。

同步API

先构建 `RedisCommands` 实例：

```java
private static RedisCommands<String, String> COMMAND;

@BeforeClass
public static void beforeClass() {
    COMMAND = CONNECTION.sync();
}
```

基本使用：

```java
@Test
public void testSyncPing() throws Exception {
   String pong = COMMAND.ping();
   Assertions.assertThat(pong).isEqualToIgnoringCase("PONG");
}


@Test
public void testSyncSetAndGet() throws Exception {
    SetArgs setArgs = SetArgs.Builder.nx().ex(5);
    COMMAND.set("name", "throwable", setArgs);
    String value = COMMAND.get("name");
    log.info("Get value: {}", value);
}

// Get value: throwable
```

同步 `API` 在所有命令调用之后会立即返回结果。如果熟悉 `Jedis` 的话，`RedisCommands` 的用法其实和它相差不大。

异步API

先构建 `RedisAsyncCommands` 实例：

```java
private static RedisAsyncCommands<String, String> ASYNC_COMMAND;

@BeforeClass
public static void beforeClass() {
    ASYNC_COMMAND = CONNECTION.async();
}
```

基本使用：

```java
@Test
```

```
public void testAsyncPing() throws Exception {
    RedisFuture<String> redisFuture = ASYNC_COMMAND.ping();
    log.info("Ping result:{}", redisFuture.get());
}
// Ping result:PONG
```

`RedisAsyncCommands` 所有方法执行返回结果都是 `RedisFuture` 实例，而 `RedisFuture` 接口的定义如下：

```
public interface RedisFuture<V> extends CompletionStage<V>, Future<V> {

    String getError();

    boolean await(long timeout, TimeUnit unit) throws InterruptedException;
}
```

也就是， `RedisFuture` 可以无缝使用 `Future` 或者 `JDK` 1.8中引入的 `CompletableFuture` 提供的方法。举个例子：

```
@Test
public void testAsyncSetAndGet1() throws Exception {
    SetArgs setArgs = SetArgs.Builder.nx().ex(5);
    RedisFuture<String> future = ASYNC_COMMAND.set("name", "throwable", setArgs);
    // CompletableFuture#thenAccept()
    future.thenAccept(value -> log.info("Set命令返回:{}", value));
    // Future#get()
    future.get();
}
// Set命令返回:OK

@Test
public void testAsyncSetAndGet2() throws Exception {
    SetArgs setArgs = SetArgs.Builder.nx().ex(5);
    CompletableFuture<Void> result =
            (CompletableFuture<Void>) ASYNC_COMMAND.set("name", "throwable", setArgs)
                    .thenAcceptBoth(ASYNC_COMMAND.get("name"),
                            (s, g) -> {
                                log.info("Set命令返回:{}", s);
                                log.info("Get命令返回:{}", g);
                            });
    result.get();
}
// Set命令返回:OK
// Get命令返回:throwable
```

如果能熟练使用 `CompletableFuture` 和函数式编程技巧，可以组合多个 `RedisFuture` 完成一些列复杂的操作。

反应式API

`Lettuce` 引入的反应式编程框架是Project Reactor，如果没有反应式编程经验可以先自行了解一下 `Project Reactor` 。

构建 `RedisReactiveCommands` 实例：

```
private static RedisReactiveCommands<String, String> REACTIVE_COMMAND;

@BeforeClass
public static void beforeClass() {
    REACTIVE_COMMAND = CONNECTION.reactive();
}
```

根据 `Project Reactor` ， `RedisReactiveCommands` 的方法如果返回的结果只包含0或1个元素，那么返回值类型是 `Mono` ，如果返回的结果包含0到N（N大于0）个元素，那么返回值是 `Flux` 。举个例子：

```
@Test
public void testReactivePing() throws Exception {
    Mono<String> ping = REACTIVE_COMMAND.ping();
    ping.subscribe(v -> log.info("Ping result:{}", v));
    Thread.sleep(1000);
}
// Ping result:PONG

@Test
public void testReactiveSetAndGet() throws Exception {
    SetArgs setArgs = SetArgs.Builder.nx().ex(5);
    REACTIVE_COMMAND.set("name", "throwable", setArgs).block();
    REACTIVE_COMMAND.get("name").subscribe(value -> log.info("Get命令返回:{}", value));
    Thread.sleep(1000);
}
// Get命令返回:throwable

@Test
public void testReactiveSet() throws Exception {
    REACTIVE_COMMAND.sadd("food", "bread", "meat", "fish").block();
    Flux<String> flux = REACTIVE_COMMAND.smembers("food");
    flux.subscribe(log::info);
    REACTIVE_COMMAND.srem("food", "bread", "meat", "fish").block();
    Thread.sleep(1000);
}
// meat
// bread
// fish
```

举个更加复杂的例子，包含了事务、函数转换等：

```
@Test
public void testReactiveFunctional() throws Exception {
    REACTIVE_COMMAND.multi().doOnSuccess(r -> {
        REACTIVE_COMMAND.set("counter", "1").doOnNext(log::info).subscribe();
        REACTIVE_COMMAND.incr("counter").doOnNext(c -> log.info(String.valueOf(c))).subscribe();
    }).flatMap(s -> REACTIVE_COMMAND.exec())
            .doOnNext(transactionResult -> log.info("Discarded:{}", transactionResult.wasDiscarded()))
            .subscribe();
    Thread.sleep(1000);
}
// OK
// 2
// Discarded:false
```

这个方法开启一个事务，先把 `counter` 设置为1，再将 `counter` 自增1。

发布和订阅

非集群模式下的发布订阅依赖于定制的连接 `StatefulRedisPubSubConnection` ，集群模式下的发布订阅依赖于定制的连接 `StatefulRedisClusterPubSubConnection` ，两者分别来源于 `RedisClient#connectPubSub()` 系列方法和 `RedisClusterClient#connectPubSub()` ：

- 非集群模式：

```
// 可能是单机、普通主从、哨兵等非集群模式的客户端
RedisClient client = ...
StatefulRedisPubSubConnection<String, String> connection = client.connectPubSub();
connection.addListener(new RedisPubSubListener<String, String>() { ... });

// 同步命令
```

```
RedisPubSubCommands<String, String> sync = connection.sync();
sync.subscribe("channel");

// 异步命令
RedisPubSubAsyncCommands<String, String> async = connection.async();
RedisFuture<Void> future = async.subscribe("channel");

// 反应式命令
RedisPubSubReactiveCommands<String, String> reactive = connection.reactive();
reactive.subscribe("channel").subscribe();

reactive.observeChannels().doOnNext(patternMessage -> {...}).subscribe()
```

- 集群模式：

```
// 使用方式其实和非集群模式基本一致
RedisClusterClient clusterClient = ...
StatefulRedisClusterPubSubConnection<String, String> connection = clusterClient.connectPubSub();
connection.addListener(new RedisPubSubListener<String, String>() { ... });
RedisPubSubCommands<String, String> sync = connection.sync();
sync.subscribe("channel");
// ...
```

这里用单机同步命令的模式举一个 `Redis` 键空间通知（Redis Keyspace Notifications）的例子：

```
@Test
public void testSyncKeyspaceNotification() throws Exception {
    RedisURI redisUri = RedisURI.builder()
            .withHost("localhost")
            .withPort(6379)
            // 注意这里只能是0号库
            .withDatabase(0)
            .withTimeout(Duration.of(10, ChronoUnit.SECONDS))
            .build();
    RedisClient redisClient = RedisClient.create(redisUri);
    StatefulRedisConnection<String, String> redisConnection = redisClient.connect();
    RedisCommands<String, String> redisCommands = redisConnection.sync();
    // 只接收键过期的事件
    redisCommands.configSet("notify-keyspace-events", "Ex");
    StatefulRedisPubSubConnection<String, String> connection = redisClient.connectPubSub();
    connection.addListener(new RedisPubSubAdapter<>() {

        @Override
        public void psubscribed(String pattern, long count) {
            log.info("pattern:{},count:{}", pattern, count);
        }

        @Override
        public void message(String pattern, String channel, String message) {
            log.info("pattern:{},channel:{},message:{}", pattern, channel, message);
        }
    });
    RedisPubSubCommands<String, String> commands = connection.sync();
    commands.psubscribe("__keyevent@0__:expired");
    redisCommands.setex("name", 2, "throwable");
    Thread.sleep(10000);
    redisConnection.close();
    connection.close();
    redisClient.shutdown();
}
// pattern:__keyevent@0__:expired,count:1
// pattern:__keyevent@0__:expired,channel:__keyevent@0__:expired,message:name
```

实际上，在实现 `RedisPubSubListener` 的时候可以单独抽离，尽量不要设计成匿名内部类的形式。

事务和批量命令执行

事务相关的命令就是 `WATCH`、`UNWATCH`、`EXEC`、`MULTI` 和 `DISCARD`，在 `RedisCommands` 系列接口中有对应的方法。举个例子：

```
// 同步模式
@Test
public void testSyncMulti() throws Exception {
    COMMAND.multi();
    COMMAND.setex("name-1", 2, "throwable");
    COMMAND.setex("name-2", 2, "doge");
    TransactionResult result = COMMAND.exec();
    int index = 0;
    for (Object r : result) {
        log.info("Result-{}:{}", index, r);
        index++;
    }
}
// Result-0:OK
// Result-1:OK
```

`Redis` 的 `Pipeline` 也就是管道机制可以理解为把多个命令打包在一次请求发送到 `Redis` 服务端，然后 `Redis` 服务端把所有的响应结果打包好一次性返回，从而节省不必要的网络资源（最主要是减少网络请求次数）。`Redis` 对于 `Pipeline` 机制如何实现并没有明确的规定，也没有提供特殊的命令支持 `Pipeline` 机制。`Jedis` 中底层采用 `BIO`（阻塞IO）通讯，所以它的做法是客户端缓存将要发送的命令，最后需要触发然后同步发送一个巨大的命令列表包，再接收和解析一个巨大的响应列表包。`Pipeline` 在 `Lettuce` 中对使用者是透明的，由于底层的通讯框架是 `Netty`，所以网络通讯层面的优化 `Lettuce` 不需要过多干预，换言之可以这样理解：`Netty` 帮 `Lettuce` 从底层实现了 `Redis` 的 `Pipeline` 机制。但是，`Lettuce` 的异步 `API` 也提供了手动 `Flush` 的方法：

```
@Test
public void testAsyncManualFlush() {
    // 取消自动flush
    ASYNC_COMMAND.setAutoFlushCommands(false);
    List<RedisFuture<?>> redisFutures = Lists.newArrayList();
    int count = 5000;
    for (int i = 0; i < count; i++) {
        String key = "key-" + (i + 1);
        String value = "value-" + (i + 1);
        redisFutures.add(ASYNC_COMMAND.set(key, value));
        redisFutures.add(ASYNC_COMMAND.expire(key, 2));
    }
    long start = System.currentTimeMillis();
    ASYNC_COMMAND.flushCommands();
    boolean result = LettuceFutures.awaitAll(10, TimeUnit.SECONDS, redisFutures.toArray(new RedisFuture[0]));
    Assertions.assertThat(result).isTrue();
    log.info("Lettuce cost:{} ms", System.currentTimeMillis() - start);
}
// Lettuce cost:1302 ms
```

上面只是从文档看到的一些理论术语，但是现实是骨感的，对比了下 `Jedis` 的 `Pipeline` 提供的方法，发现了 `Jedis` 的 `Pipeline` 执行耗时比较低：

```
@Test
public void testJedisPipeline() throws Exception {
    Jedis jedis = new Jedis();
    Pipeline pipeline = jedis.pipelined();
    int count = 5000;
    for (int i = 0; i < count; i++) {
        String key = "key-" + (i + 1);
```

```
        String value = "value-" + (i + 1);
        pipeline.set(key, value);
        pipeline.expire(key, 2);
    }
    long start = System.currentTimeMillis();
    pipeline.syncAndReturnAll();
    log.info("Jedis cost:{} ms", System.currentTimeMillis()  - start);
}
// Jedis cost:9 ms
```

个人猜测 `Lettuce` 可能底层并非合并所有命令一次发送（甚至可能是单条发送），具体可能需要抓包才能定位。依此来看，如果真的有大量执行 `Redis` 命令的场景，不妨可以使用 `Jedis` 的 `Pipeline` 。

注意：由上面的测试推断 `RedisTemplate` 的 `executePipelined()` 方法是 假的 `Pipeline` 执行方法，使用 `RedisTemplate` 的时候请务必注意这一点。

Lua脚本执行

`Lettuce` 中执行 `Redis` 的 `Lua` 命令的同步接口如下：

```
public interface RedisScriptingCommands<K, V> {

    <T> T eval(String var1, ScriptOutputType var2, K... var3);

    <T> T eval(String var1, ScriptOutputType var2, K[] var3, V... var4);

    <T> T evalsha(String var1, ScriptOutputType var2, K... var3);

    <T> T evalsha(String var1, ScriptOutputType var2, K[] var3, V... var4);

    List<Boolean> scriptExists(String... var1);

    String scriptFlush();

    String scriptKill();

    String scriptLoad(V var1);

    String digest(V var1);
}
```

异步和反应式的接口方法定义差不多，不同的地方就是返回值类型，一般我们常用的是 `eval()` 、 `evalsha()` 和 `scriptLoad()` 方法。举个简单的例子：

```
private static RedisCommands<String, String> COMMANDS;
private static String RAW_LUA = "local key = KEYS[1]\n" +
        "local value = ARGV[1]\n" +
        "local timeout = ARGV[2]\n" +
        "redis.call('SETEX', key, tonumber(timeout), value)\n" +
        "local result = redis.call('GET', key)\n" +
        "return result;";
private static AtomicReference<String> LUA_SHA = new AtomicReference<>();

@Test
public void testLua() throws Exception {
    LUA_SHA.compareAndSet(null, COMMANDS.scriptLoad(RAW_LUA));
    String[] keys = new String[]{"name"};
    String[] args = new String[]{"throwable", "5000"};
    String result = COMMANDS.evalsha(LUA_SHA.get(), ScriptOutputType.VALUE, keys, args);
    log.info("Get value:{}", result);
}
// Get value:throwable
```

高可用和分片

为了 `Redis` 的高可用，一般会采用普通主从（ `Master/Replica` ，这里笔者称为普通主从模式，也就是仅仅做了主从复制，故障需要手动切换）、哨兵和集群。普通主从模式可以独立运行，也可以配合哨兵运行，只是哨兵提供自动故障转移和主节点提升功能。普通主从和哨兵都可以使用 `MasterSlave` ，通过入参包括 `RedisClient` 、编码解码器以及一个或者多个 `RedisURI` 获取对应的 `Connection` 实例。

这里注意一点， `MasterSlave` 中提供的方法如果只要求传入一个 `RedisURI` 实例，那么 `Lettuce` 会进行 拓扑发现机制，自动获取 `Redis` 主从节点信息；如果要求传入一个 `RedisURI` 集合，那么对于普通主从模式来说所有节点信息是静态的，不会进行发现和更新。

拓扑发现的规则如下：

- 对于普通主从（ `Master/Replica` ）模式，不需要感知 `RedisURI` 指向从节点还是主节点，只会进行一次性的拓扑查找所有节点信息，此后节点信息会保存在静态缓存中，不会更新。
- 对于哨兵模式，会订阅所有哨兵实例并侦听订阅/发布消息以触发拓扑刷新机制，更新缓存的节点信息，也就是哨兵天然就是动态发现节点信息，不支持静态配置。

拓扑发现机制的提供 `API` 为 `TopologyProvider` ，需要了解其原理的可以参考具体的实现。

对于集群（ `Cluster` ）模式， `Lettuce` 提供了一套独立的 `API` 。

另外，如果 `Lettuce` 连接面向的是非单个 `Redis` 节点，连接实例提供了 数据读取节点偏好（ `ReadFrom` ）设置，可选值有：

- `MASTER` ：只从 `Master` 节点中读取。
- `MASTER_PREFERRED` ：优先从 `Master` 节点中读取。
- `SLAVE_PREFERRED` ：优先从 `Slavor` 节点中读取。
- `SLAVE` ：只从 `Slavor` 节点中读取。
- `NEAREST` ：使用最近一次连接的 `Redis` 实例读取。

普通主从模式

假设现在有三个 `Redis` 服务形成树状主从关系如下：

- 节点一：localhost:6379，角色为Master。
- 节点二：localhost:6380，角色为Slavor，节点一的从节点。
- 节点三：localhost:6381，角色为Slavor，节点二的从节点。

首次动态节点发现主从模式的节点信息需要如下构建连接：

```
@Test
public void testDynamicReplica() throws Exception {
    // 这里只需要配置一个节点的连接信息，不一定需要是主节点的信息，从节点也可以
    RedisURI uri = RedisURI.builder().withHost("localhost").withPort(6379).build();
    RedisClient redisClient = RedisClient.create(uri);
    StatefulRedisMasterSlaveConnection<String, String> connection = MasterSlave.connect(redisClient, new Utf8StringCodec(), uri);
    // 只从从节点读取数据
    connection.setReadFrom(ReadFrom.SLAVE);
    // 执行其他Redis命令
    connection.close();
    redisClient.shutdown();
}
```

如果需要指定静态的 `Redis` 主从节点连接属性，那么可以这样构建连接：

```
@Test
public void testStaticReplica() throws Exception {
    List<RedisURI> uris = new ArrayList<>();
    RedisURI uri1 = RedisURI.builder().withHost("localhost").withPort(6379).build();
    RedisURI uri2 = RedisURI.builder().withHost("localhost").withPort(6380).build();
    RedisURI uri3 = RedisURI.builder().withHost("localhost").withPort(6381).build();
```

```
    uris.add(uri1);
    uris.add(uri2);
    uris.add(uri3);
    RedisClient redisClient = RedisClient.create();
    StatefulRedisMasterSlaveConnection<String, String> connection = MasterSlave.connect(redisClient,
            new Utf8StringCodec(), uris);
    // 只从主节点读取数据
    connection.setReadFrom(ReadFrom.MASTER);
    // 执行其他Redis命令
    connection.close();
    redisClient.shutdown();
}
```

哨兵模式

由于 `Lettuce` 自身提供了哨兵的拓扑发现机制，所以只需要随便配置一个哨兵节点的 `RedisURI` 实例即可：

```
@Test
public void testDynamicSentinel() throws Exception {
    RedisURI redisUri = RedisURI.builder()
            .withPassword("你的密码")
            .withSentinel("localhost", 26379)
            .withSentinelMasterId("哨兵Master的ID")
            .build();
    RedisClient redisClient = RedisClient.create();
    StatefulRedisMasterSlaveConnection<String, String> connection = MasterSlave.connect(redisClient, new Utf8StringCodec(), redisUri);
    // 只允许从从节点读取数据
    connection.setReadFrom(ReadFrom.SLAVE);
    RedisCommands<String, String> command = connection.sync();
    SetArgs setArgs = SetArgs.Builder.nx().ex(5);
    command.set("name", "throwable", setArgs);
    String value = command.get("name");
    log.info("Get value:{}", value);
}
// Get value:throwable
```

集群模式

鉴于笔者对 `Redis` 集群模式并不熟悉， `Cluster` 模式下的 `API` 使用本身就有比较多的限制，所以这里只简单介绍一下怎么用。先说几个特性：

下面的API提供跨槽位（ `Slot` ）调用的功能：

- `RedisAdvancedClusterCommands` 。
- `RedisAdvancedClusterAsyncCommands` 。
- `RedisAdvancedClusterReactiveCommands` 。

静态节点选择功能：

- `masters` ：选择所有主节点执行命令。
- `slaves` ：选择所有从节点执行命令，其实就是只读模式。
- `all nodes` ：命令可以在所有节点执行。

集群拓扑视图动态更新功能：

- 手动更新，主动调用 `RedisClusterClient#reloadPartitions()` 。
- 后台定时更新。
- 自适应更新，基于连接断开和 `MOVED/ASK` 命令重定向自动更新。

`Redis` 集群搭建详细过程可以参考官方文档，假设已经搭建好集群如下（ `192.168.56.200` 是笔者的虚拟机Host）：

- 192.168.56.200:7001 => 主节点，槽位0-5460。
- 192.168.56.200:7002 => 主节点，槽位5461-10922。
- 192.168.56.200:7003 => 主节点，槽位10923-16383。
- 192.168.56.200:7004 => 7001的从节点。
- 192.168.56.200:7005 => 7002的从节点。
- 192.168.56.200:7006 => 7003的从节点。

简单的集群连接和使用方式如下：

```
@Test
public void testSyncCluster(){
    RedisURI uri = RedisURI.builder().withHost("192.168.56.200").build();
    RedisClusterClient redisClusterClient = RedisClusterClient.create(uri);
    StatefulRedisClusterConnection<String, String> connection = redisClusterClient.connect();
    RedisAdvancedClusterCommands<String, String> commands = connection.sync();
    commands.setex("name",10, "throwable");
    String value = commands.get("name");
    log.info("Get value:{}", value);
}
// Get value:throwable
```

节点选择：

```
@Test
public void testSyncNodeSelection() {
    RedisURI uri = RedisURI.builder().withHost("192.168.56.200").withPort(7001).build();
    RedisClusterClient redisClusterClient = RedisClusterClient.create(uri);
    StatefulRedisClusterConnection<String, String> connection = redisClusterClient.connect();
    RedisAdvancedClusterCommands<String, String> commands = connection.sync();
//  commands.all();  // 所有节点
//  commands.masters();  // 主节点
    // 从节点只读
    NodeSelection<String, String> replicas = commands.slaves();
    NodeSelectionCommands<String, String> nodeSelectionCommands = replicas.commands();
    // 这里只是演示,一般应该禁用keys *命令
    Executions<List<String>> keys = nodeSelectionCommands.keys("*");
    keys.forEach(key -> log.info("key: {}", key));
    connection.close();
    redisClusterClient.shutdown();
}
```

定时更新集群拓扑视图（每隔十分钟更新一次，这个时间自行考量，不能太频繁）：

```
@Test
public void testPeriodicClusterTopology() throws Exception {
    RedisURI uri = RedisURI.builder().withHost("192.168.56.200").withPort(7001).build();
    RedisClusterClient redisClusterClient = RedisClusterClient.create(uri);
    ClusterTopologyRefreshOptions options = ClusterTopologyRefreshOptions
            .builder()
            .enablePeriodicRefresh(Duration.of(10, ChronoUnit.MINUTES))
            .build();
    redisClusterClient.setOptions(ClusterClientOptions.builder().topologyRefreshOptions(options).build());
    StatefulRedisClusterConnection<String, String> connection = redisClusterClient.connect();
    RedisAdvancedClusterCommands<String, String> commands = connection.sync();
    commands.setex("name", 10, "throwable");
    String value = commands.get("name");
    log.info("Get value:{}", value);
    Thread.sleep(Integer.MAX_VALUE);
```

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

自适应更新集群拓扑视图：

```
@Test
public void testAdaptiveClusterTopology() throws Exception {
    RedisURI uri = RedisURI.builder().withHost("192.168.56.200").withPort(7001).build();
    RedisClusterClient redisClusterClient = RedisClusterClient.create(uri);
    ClusterTopologyRefreshOptions options = ClusterTopologyRefreshOptions.builder()
            .enableAdaptiveRefreshTrigger(
                    ClusterTopologyRefreshOptions.RefreshTrigger.MOVED_REDIRECT,
                    ClusterTopologyRefreshOptions.RefreshTrigger.PERSISTENT_RECONNECTS
            )
            .adaptiveRefreshTriggersTimeout(Duration.of(30, ChronoUnit.SECONDS))
            .build();
    redisClusterClient.setOptions(ClusterClientOptions.builder().topologyRefreshOptions(options).build());
    StatefulRedisClusterConnection<String, String> connection = redisClusterClient.connect();
    RedisAdvancedClusterCommands<String, String> commands = connection.sync();
    commands.setex("name", 10, "throwable");
    String value = commands.get("name");
    log.info("Get value:{}", value);
    Thread.sleep(Integer.MAX_VALUE);
    connection.close();
    redisClusterClient.shutdown();
}
```

动态命令和自定义命令

自定义命令是 `Redis` 命令有限集，不过可以更细粒度指定 `KEY` 、 `ARGV` 、命令类型、编码解码器和返回值类型，依赖于 `dispatch()` 方法：

```
// 自定义实现PING方法
@Test
public void testCustomPing() throws Exception {
    RedisURI redisUri = RedisURI.builder()
            .withHost("localhost")
            .withPort(6379)
            .withTimeout(Duration.of(10, ChronoUnit.SECONDS))
            .build();
    RedisClient redisClient = RedisClient.create(redisUri);
    StatefulRedisConnection<String, String> connect = redisClient.connect();
    RedisCommands<String, String> sync = connect.sync();
    RedisCodec<String, String> codec = StringCodec.UTF8;
    String result = sync.dispatch(CommandType.PING, new StatusOutput<>(codec));
    log.info("PING:{}", result);
    connect.close();
    redisClient.shutdown();
}
// PING:PONG

// 自定义实现Set方法
@Test
public void testCustomSet() throws Exception {
    RedisURI redisUri = RedisURI.builder()
            .withHost("localhost")
            .withPort(6379)
            .withTimeout(Duration.of(10, ChronoUnit.SECONDS))
            .build();
    RedisClient redisClient = RedisClient.create(redisUri);
    StatefulRedisConnection<String, String> connect = redisClient.connect();
    RedisCommands<String, String> sync = connect.sync();
    RedisCodec<String, String> codec = StringCodec.UTF8;
    sync.dispatch(CommandType.SETEX, new StatusOutput<>(codec),
            new CommandArgs<>(codec).addKey("name").add(5).addValue("throwable"));
    String result = sync.get("name");
    log.info("Get value:{}", result);
    connect.close();
    redisClient.shutdown();
}
// Get value:throwable
```

动态命令是基于 `Redis` 命令有限集，并且通过注解和动态代理完成一些复杂命令组合的实现。主要注解在 `io.lettuce.core.dynamic.annotation` 包路径下。简单举个例子：

```
public interface CustomCommand extends Commands {

    // SET [key] [value]
    @Command("SET ?0 ?1")
    String setKey(String key, String value);

    // SET [key] [value]
    @Command("SET :key :value")
    String setKeyNamed(@Param("key") String key, @Param("value") String value);

    // MGET [key1] [key2]
    @Command("MGET ?0 ?1")
    List<String> mGet(String key1, String key2);
    /**
     * 方法名作为命令
     */
    @CommandNaming(strategy = CommandNaming.Strategy.METHOD_NAME)
    String mSet(String key1, String value1, String key2, String value2);
}


@Test
public void testCustomDynamicSet() throws Exception {
    RedisURI redisUri = RedisURI.builder()
            .withHost("localhost")
            .withPort(6379)
            .withTimeout(Duration.of(10, ChronoUnit.SECONDS))
            .build();
    RedisClient redisClient = RedisClient.create(redisUri);
    StatefulRedisConnection<String, String> connect = redisClient.connect();
    RedisCommandFactory commandFactory = new RedisCommandFactory(connect);
    CustomCommand commands = commandFactory.getCommands(CustomCommand.class);
    commands.setKey("name", "throwable");
    commands.setKeyNamed("throwable", "doge");
    log.info("MGET ===> " + commands.mGet("name", "throwable"));
    commands.mSet("key1", "value1","key2", "value2");
    log.info("MGET ===> " + commands.mGet("key1", "key2"));
    connect.close();
    redisClient.shutdown();
}
// MGET ===> [throwable, doge]
// MGET ===> [value1, value2]
```

高阶特性

`Lettuce` 有很多高阶使用特性，这里只列举个人认为常用的两点：

- 配置客户端资源。
- 使用连接池。

更多其他特性可以自行参看官方文档。

配置客户端资源

客户端资源的设置与 `Lettuce` 的性能、并发和事件处理相关。线程池或者线程组相关配置占据客户端资源配置的大部分（ `EventLoopGroups` 和 `EventExecutorGroup` ），这些线程池或者线程组是连接程序的基础组件。一般情况下，客户端资源应该在多个 `Redis` 客户端之间共享，并且在不再使用的时候需要自行关闭。笔者认为，客户端资源是面向 `Netty` 的。 注意 ：除非特别熟悉或者花长时间去测试调整下面提到的参数，否则在没有经验的前提下凭直觉修改默认值，有可能会踩坑。

客户端资源接口是 `ClientResources` ，实现类是 `DefaultClientResources` 。

构建 `DefaultClientResources` 实例：

```
// 默认
ClientResources resources = DefaultClientResources.create();

// 建造器
ClientResources resources = DefaultClientResources.builder()
                        .ioThreadPoolSize(4)
                        .computationThreadPoolSize(4)
                        .build()
```

使用：

```
ClientResources resources = DefaultClientResources.create();
// 非集群
RedisClient client = RedisClient.create(resources, uri);
// 集群
RedisClusterClient clusterClient = RedisClusterClient.create(resources, uris);
// ......
client.shutdown();
clusterClient.shutdown();
// 关闭资源
resources.shutdown();
```

客户端资源基本配置：

属性	描述	默认值
`ioThreadPoolSize`	`I/O` 线程数	`Runtime.getRuntime().availableProcessors()`
`computationThreadPoolSize`	任务线程数	`Runtime.getRuntime().availableProcessors()`

客户端资源高级配置：

属性	描述	默认值
`eventLoopGroupProvider`	`EventLoopGroup` 提供商	-
`eventExecutorGroupProvider`	`EventExecutorGroup` 提供商	-
`eventBus`	事件总线	`DefaultEventBus`
`commandLatencyCollectorOptions`	命令延时收集器配置	`DefaultCommandLatencyCollectorOptions`
`commandLatencyCollector`	命令延时收集器	`DefaultCommandLatencyCollector`
`commandLatencyPublisherOptions`	命令延时发布器配置	`DefaultEventPublisherOptions`
`dnsResolver`	`DNS` 处理器	JDK或者 `Netty` 提供
`reconnectDelay`	重连延时配置	`Delay.exponential()`
`nettyCustomizer`	`Netty` 自定义配置器	-
`tracing`	轨迹记录器	-

非集群客户端 `RedisClient` 的属性配置：

`Redis` 非集群客户端 `RedisClient` 本身提供了配置属性方法：

```
RedisClient client = RedisClient.create(uri);
client.setOptions(ClientOptions.builder()
                        .autoReconnect(false)
                        .pingBeforeActivateConnection(true)
                        .build());
```

非集群客户端的配置属性列表：

属性	描述	默认值
`pingBeforeActivateConnection`	连接激活之前是否执行 `PING` 命令	false
`autoReconnect`	是否自动重连	true
`cancelCommandsOnReconnectFailure`	重连失败是否拒绝命令执行	false
`suspendReconnectOnProtocolFailure`	底层协议失败是否挂起重连操作	false
`requestQueueSize`	请求队列容量	2147483647(Integer#MAX_VALUE)
`disconnectedBehavior`	失去连接时候的行为	`DEFAULT`
`sslOptions`	`SSL`配置	-
`socketOptions`	`Socket` 配置	`10 seconds Connection-Timeout, no keep-alive, no TCP noDelay`
`timeoutOptions`	超时配置	-
`publishOnScheduler`	发布反应式信号数据的调度器	使用 `I/O` 线程

集群客户端属性配置：

`Redis` 集群客户端 `RedisClusterClient` 本身提供了配置属性方法：

```
RedisClusterClient client = RedisClusterClient.create(uri);
ClusterTopologyRefreshOptions topologyRefreshOptions = ClusterTopologyRefreshOptions.builder()
                .enablePeriodicRefresh(refreshPeriod(10, TimeUnit.MINUTES))
                .enableAllAdaptiveRefreshTriggers()
                .build();

client.setOptions(ClusterClientOptions.builder()
                        .topologyRefreshOptions(topologyRefreshOptions)
                        .build());
```

集群客户端的配置属性列表：

属性	描述	默认值
`enablePeriodicRefresh`	是否允许周期性更新集群拓扑视图	false
`refreshPeriod`	更新集群拓扑视图周期	60秒

属性	描述	默认值
`enableAdaptiveRefreshTrigger`	设置自适应更新集群拓扑视图触发器 `RefreshTrigger`	-
`adaptiveRefreshTriggersTimeout`	自适应更新集群拓扑视图触发器超时设置	30秒
`refreshTriggersReconnectAttempts`	自适应更新集群拓扑视图触发重连次数	5
`dynamicRefreshSources`	是否允许动态刷新拓扑资源	true
`closeStaleConnections`	是否允许关闭陈旧的连接	true
`maxRedirects`	集群重定向次数上限	5
`validateClusterNodeMembership`	是否校验集群节点的成员关系	true

使用连接池

引入连接池依赖 `commons-pool2` ：

```xml
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
    <version>2.7.0</version>
</dependency>
```

基本使用如下：

```java
@Test
public void testUseConnectionPool() throws Exception {
    RedisURI redisUri = RedisURI.builder()
            .withHost("localhost")
            .withPort(6379)
            .withTimeout(Duration.of(10, ChronoUnit.SECONDS))
            .build();
    RedisClient redisClient = RedisClient.create(redisUri);
    GenericObjectPoolConfig poolConfig = new GenericObjectPoolConfig();
    GenericObjectPool<StatefulRedisConnection<String, String>> pool
            = ConnectionPoolSupport.createGenericObjectPool(redisClient::connect, poolConfig);
    try (StatefulRedisConnection<String, String> connection = pool.borrowObject()) {
        RedisCommands<String, String> command = connection.sync();
        SetArgs setArgs = SetArgs.Builder.nx().ex(5);
        command.set("name", "throwable", setArgs);
        String n = command.get("name");
        log.info("Get value:{}", n);
    }
    pool.close();
    redisClient.shutdown();
}
```

其中，同步连接的池化支持需要用 `ConnectionPoolSupport` ，异步连接的池化支持需要用 `AsyncConnectionPoolSupport` （ `Lettuce` 5.1之后才支持）。

几个常见的渐进式删除例子

渐进式删除Hash中的域-属性：

```java
@Test
public void testDelBigHashKey() throws Exception {
    // SCAN参数
    ScanArgs scanArgs = ScanArgs.Builder.limit(2);
    // TEMP游标
    ScanCursor cursor = ScanCursor.INITIAL;
    // 目标KEY
    String key = "BIG_HASH_KEY";
    prepareHashTestData(key);
    log.info("开始渐进式删除Hash的元素...");
    int counter = 0;
    do {
        MapScanCursor<String, String> result = COMMAND.hscan(key, cursor, scanArgs);
        // 重置TEMP游标
        cursor = ScanCursor.of(result.getCursor());
        cursor.setFinished(result.isFinished());
        Collection<String> fields = result.getMap().values();
        if (!fields.isEmpty()) {
            COMMAND.hdel(key, fields.toArray(new String[0]));
        }
        counter++;
    } while (!(ScanCursor.FINISHED.getCursor().equals(cursor.getCursor()) && ScanCursor.FINISHED.isFinished() == cursor.isFinished()));
    log.info("渐进式删除Hash的元素完毕,迭代次数:{} ...", counter);
}

private void prepareHashTestData(String key) throws Exception {
    COMMAND.hset(key, "1", "1");
    COMMAND.hset(key, "2", "2");
    COMMAND.hset(key, "3", "3");
    COMMAND.hset(key, "4", "4");
    COMMAND.hset(key, "5", "5");
}
```

渐进式删除集合中的元素：

```java
@Test
public void testDelBigSetKey() throws Exception {
    String key = "BIG_SET_KEY";
    prepareSetTestData(key);
    // SCAN参数
    ScanArgs scanArgs = ScanArgs.Builder.limit(2);
    // TEMP游标
    ScanCursor cursor = ScanCursor.INITIAL;
    log.info("开始渐进式删除Set的元素...");
    int counter = 0;
    do {
        ValueScanCursor<String> result = COMMAND.sscan(key, cursor, scanArgs);
        // 重置TEMP游标
        cursor = ScanCursor.of(result.getCursor());
        cursor.setFinished(result.isFinished());
        List<String> values = result.getValues();
        if (!values.isEmpty()) {
            COMMAND.srem(key, values.toArray(new String[0]));
        }
        counter++;
    } while (!(ScanCursor.FINISHED.getCursor().equals(cursor.getCursor()) && ScanCursor.FINISHED.isFinished() == cursor.isFinished()));
    log.info("渐进式删除Set的元素完毕,迭代次数:{} ...", counter);
}

private void prepareSetTestData(String key) throws Exception {
    COMMAND.sadd(key, "1", "2", "3", "4", "5");
}
```

渐进式删除有序集合中的元素：

```java
@Test
```

```
public void testDelBigZSetKey() throws Exception {
    // SCAN参数
    ScanArgs scanArgs = ScanArgs.Builder.limit(2);
    // TEMP游标
    ScanCursor cursor = ScanCursor.INITIAL;
    // 目标KEY
    String key = "BIG_ZSET_KEY";
    prepareZSetTestData(key);
    log.info("开始渐进式删除ZSet的元素...");
    int counter = 0;
    do {
        ScoredValueScanCursor<String> result = COMMAND.zscan(key, cursor, scanArgs);
        // 重置TEMP游标
        cursor = ScanCursor.of(result.getCursor());
        cursor.setFinished(result.isFinished());
        List<ScoredValue<String>> scoredValues = result.getValues();
        if (!scoredValues.isEmpty()) {
            COMMAND.zrem(key, scoredValues.stream().map(ScoredValue<String>::getValue).toArray(String[]::new));
        }
        counter++;
    } while (!(ScanCursor.FINISHED.getCursor().equals(cursor.getCursor()) && ScanCursor.FINISHED.isFinished() == cursor.isFinished()));
    log.info("渐进式删除ZSet的元素完毕,迭代次数:{} ...", counter);
}

private void prepareZSetTestData(String key) throws Exception {
    COMMAND.zadd(key, 0, "1");
    COMMAND.zadd(key, 0, "2");
    COMMAND.zadd(key, 0, "3");
    COMMAND.zadd(key, 0, "4");
    COMMAND.zadd(key, 0, "5");
}
```

在SpringBoot中使用Lettuce

个人认为，`spring-data-redis` 中的 `API` 封装并不是很优秀，用起来比较重，不够灵活，这里结合前面的例子和代码，在 `SpringBoot` 脚手架项目中配置和整合 `Lettuce`。先引入依赖：

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.1.8.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
            <dependency>
        <groupId>io.lettuce</groupId>
        <artifactId>lettuce-core</artifactId>
        <version>5.1.8.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.10</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

一般情况下，每个应用应该使用单个 `Redis` 客户端实例和单个连接实例，这里设计一个脚手架，适配单机、普通主从、哨兵和集群四种使用场景。对于客户端资源，采用默认的实现即可。对于 `Redis` 的连接属性，比较主要的有 `Host` 、 `Port` 和 `Password` ，其他可以暂时忽略。基于约定大于配置的原则，先定制一系列属性配置类（其实有些配置是可以完全共用，但是考虑到要清晰描述类之间的关系，这里拆分多个配置属性类和多个配置方法）：

```
@Data
@ConfigurationProperties(prefix = "lettuce")
public class LettuceProperties {

    private LettuceSingleProperties single;
    private LettuceReplicaProperties replica;
    private LettuceSentinelProperties sentinel;
    private LettuceClusterProperties cluster;


}

@Data
public class LettuceSingleProperties {

    private String host;
    private Integer port;
    private String password;
}

@EqualsAndHashCode(callSuper = true)
@Data
public class LettuceReplicaProperties extends LettuceSingleProperties {

}

@EqualsAndHashCode(callSuper = true)
@Data
public class LettuceSentinelProperties extends LettuceSingleProperties {

    private String masterId;
}

@EqualsAndHashCode(callSuper = true)
@Data
public class LettuceClusterProperties extends LettuceSingleProperties {

}
```

配置类如下，主要使用 `@ConditionalOnProperty` 做隔离，一般情况下，很少有人会在一个应用使用一种以上的 `Redis` 连接场景：

```
@RequiredArgsConstructor
@Configuration
@ConditionalOnClass(name = "io.lettuce.core.RedisURI")
@EnableConfigurationProperties(value = LettuceProperties.class)
public class LettuceAutoConfiguration {

    private final LettuceProperties lettuceProperties;

    @Bean(destroyMethod = "shutdown")
    public ClientResources clientResources() {
        return DefaultClientResources.create();
```

```
    }

    @Bean
    @ConditionalOnProperty(name = "lettuce.single.host")
    public RedisURI singleRedisUri() {
        LettuceSingleProperties singleProperties = lettuceProperties.getSingle();
        return RedisURI.builder()
                .withHost(singleProperties.getHost())
                .withPort(singleProperties.getPort())
                .withPassword(singleProperties.getPassword())
                .build();
    }

    @Bean(destroyMethod = "shutdown")
    @ConditionalOnProperty(name = "lettuce.single.host")
    public RedisClient singleRedisClient(ClientResources clientResources, @Qualifier("singleRedisUri") RedisURI redisUri) {
        return RedisClient.create(clientResources, redisUri);
    }

    @Bean(destroyMethod = "close")
    @ConditionalOnProperty(name = "lettuce.single.host")
    public StatefulRedisConnection<String, String> singleRedisConnection(@Qualifier("singleRedisClient") RedisClient singleRedisClient) {
        return singleRedisClient.connect();
    }

    @Bean
    @ConditionalOnProperty(name = "lettuce.replica.host")
    public RedisURI replicaRedisUri() {
        LettuceReplicaProperties replicaProperties = lettuceProperties.getReplica();
        return RedisURI.builder()
                .withHost(replicaProperties.getHost())
                .withPort(replicaProperties.getPort())
                .withPassword(replicaProperties.getPassword())
                .build();
    }

    @Bean(destroyMethod = "shutdown")
    @ConditionalOnProperty(name = "lettuce.replica.host")
    public RedisClient replicaRedisClient(ClientResources clientResources, @Qualifier("replicaRedisUri") RedisURI redisUri) {
        return RedisClient.create(clientResources, redisUri);
    }

    @Bean(destroyMethod = "close")
    @ConditionalOnProperty(name = "lettuce.replica.host")
    public StatefulRedisMasterSlaveConnection<String, String> replicaRedisConnection(@Qualifier("replicaRedisClient") RedisClient replicaRedis
Client,
                                                                                      @Qualifier("replicaRedisUri") RedisURI redisUri) {
        return MasterSlave.connect(replicaRedisClient, new Utf8StringCodec(), redisUri);
    }

    @Bean
    @ConditionalOnProperty(name = "lettuce.sentinel.host")
    public RedisURI sentinelRedisUri() {
        LettuceSentinelProperties sentinelProperties = lettuceProperties.getSentinel();
        return RedisURI.builder()
                .withPassword(sentinelProperties.getPassword())
                .withSentinel(sentinelProperties.getHost(), sentinelProperties.getPort())
                .withSentinelMasterId(sentinelProperties.getMasterId())
                .build();
    }

    @Bean(destroyMethod = "shutdown")
    @ConditionalOnProperty(name = "lettuce.sentinel.host")
    public RedisClient sentinelRedisClient(ClientResources clientResources, @Qualifier("sentinelRedisUri") RedisURI redisUri) {
        return RedisClient.create(clientResources, redisUri);
    }

    @Bean(destroyMethod = "close")
    @ConditionalOnProperty(name = "lettuce.sentinel.host")
    public StatefulRedisMasterSlaveConnection<String, String> sentinelRedisConnection(@Qualifier("sentinelRedisClient") RedisClient sentinelRe
disClient,
                                                                                       @Qualifier("sentinelRedisUri") RedisURI redisUri) {
        return MasterSlave.connect(sentinelRedisClient, new Utf8StringCodec(), redisUri);
    }

    @Bean
    @ConditionalOnProperty(name = "lettuce.cluster.host")
    public RedisURI clusterRedisUri() {
        LettuceClusterProperties clusterProperties = lettuceProperties.getCluster();
        return RedisURI.builder()
                .withHost(clusterProperties.getHost())
                .withPort(clusterProperties.getPort())
                .withPassword(clusterProperties.getPassword())
                .build();
    }

    @Bean(destroyMethod = "shutdown")
    @ConditionalOnProperty(name = "lettuce.cluster.host")
    public RedisClusterClient redisClusterClient(ClientResources clientResources, @Qualifier("clusterRedisUri") RedisURI redisUri) {
        return RedisClusterClient.create(clientResources, redisUri);
    }

    @Bean(destroyMethod = "close")
    @ConditionalOnProperty(name = "lettuce.cluster")
    public StatefulRedisClusterConnection<String, String> clusterConnection(RedisClusterClient clusterClient) {
        return clusterClient.connect();
    }
}
```

最后为了让 `IDE` 识别我们的配置，可以添加 `IDE` 亲缘性， `/META-INF` 文件夹下新增一个文件 `spring-configuration-metadata.json` ，内容如下：

```
{
  "properties": [
    {
      "name": "lettuce.single",
      "type": "club.throwable.spring.lettuce.LettuceSingleProperties",
      "description": "单机配置",
      "sourceType": "club.throwable.spring.lettuce.LettuceProperties"
    },
    {
      "name": "lettuce.replica",
      "type": "club.throwable.spring.lettuce.LettuceReplicaProperties",
      "description": "主从配置",
      "sourceType": "club.throwable.spring.lettuce.LettuceProperties"
    },
    {
      "name": "lettuce.sentinel",
      "type": "club.throwable.spring.lettuce.LettuceSentinelProperties",
      "description": "哨兵配置",
      "sourceType": "club.throwable.spring.lettuce.LettuceProperties"
    },
    {
      "name": "lettuce.single",
```

```
      "type": "club.throwable.spring.lettuce.LettuceClusterProperties",
      "description": "集群配置",
      "sourceType": "club.throwable.spring.lettuce.LettuceProperties"
    }
  ]
}
```

如果想 `IDE` 亲缘性做得更好，可以添加 `/META-INF/additional-spring-configuration-metadata.json` 进行更多细节定义。简单使用如下：

```
@Slf4j
@Component
public class RedisCommandLineRunner implements CommandLineRunner {

    @Autowired
    @Qualifier("singleRedisConnection")
    private StatefulRedisConnection<String, String> connection;

    @Override
    public void run(String... args) throws Exception {
        RedisCommands<String, String> redisCommands = connection.sync();
        redisCommands.setex("name", 5, "throwable");
        log.info("Get value:{}", redisCommands.get("name"));
    }
}
// Get value:throwable
```

小结

本文算是基于 `Lettuce` 的官方文档，对它的使用进行全方位的分析，包括主要功能、配置都做了一些示例，限于篇幅部分特性和配置细节没有分析。 `Lettuce` 已经被 `spring-data-redis` 接纳作为官方的 `Redis` 客户端驱动，所以值得信赖，它的一些 `API` 设计确实比较合理，扩展性高的同时灵活性也高。个人建议，基于 `Lettuce` 包自行添加配置到 `SpringBoot` 应用用起来会得心应手，毕竟 `RedisTemplate` 实在太笨重，而且还屏蔽了 `Lettuce` 一些高级特性和灵活的 `API` 。

参考资料：

- Lettuce Reference Guide

链接

- Github Page：http://www.throwable.club/2019/09/28/redis-client-driver-lettuce-usage
- Coding Page：http://throwable.coding.me/2019/09/28/redis-client-driver-lettuce-usage

（本文完 c-14-d e-a-20190928 最近事太多...）

技术公众号（《Throwable文摘》），不定期推送笔者原创技术文章（绝不抄袭或者转载）：

posted @ 2019-09-28 09:33 throwable 阅读(45410) 评论(9) 编辑 收藏