Please visit website: http://cxyroad.com

Redis项目中竟然有这么多种使用场景!!
==========

> 下面蜗牛哥依次对Redis 12种业务场景举例说明和解释：

1. 排行榜和计数器

针对Redis作为排行榜和计数器的使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redis来实现一个简单的文章点赞功能，并将点赞数用作排行榜的依据。

场景描述

假设我们正在开发一个博客平台，用户可以对文章进行点赞。

我们希望根据文章的点赞数来显示一个实时更新的热门文章排行榜。

创建Spring Boot项目

配置Redis连接

在src/main/resources/application.properties中配置Redis 服务器的连接信息：

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 定义文章实体类

```
public class Article {
    private String id;
    private String title;
    private int likeCount;
    // 省略构造函数、getter和setter方法
}

```

2. 创建文章服务接口和实现类

```
@Service
public class ArticleService {

    @Autowired
    private StringRedisTemplate redisTemplate;

    public void likeArticle(String articleId) {
        // 增加文章的点赞数
        redisTemplate.opsForValue().increment(articleId, 1);
    }

    public List<Article> getTopLikedArticles(int topN) {
        // 获取topN个点赞数最多的文章
        Set<String> articleIds = redisTemplate.keys("article:*:likeCount");
        List<Article> topArticles = new ArrayList<>();
        for (String id : articleIds) {
            int likeCount = (Integer) redisTemplate.opsForValue().get(id);
            Article article = new Article();
            article.setId(id.replace("article:", "").replace(":likeCount", ""));
            article.setTitle("文章标题待查询");
            article.setLikeCount(likeCount);
            topArticles.add(article);
        }
        // 根据点赞数排序
        topArticles.sort((a1, a2) -> a2.getLikeCount() - a1.getLikeCount());
        return topArticles.subList(0, topN);
    }
}
```

```

3. 创建控制器



```
@RestController
@RequestMapping("/articles")
public class ArticleController {

    @Autowired
    private ArticleService articleService;

    @PostMapping("/{id}/like")
    public ResponseEntity<String> likeArticle(@PathVariable String id) {
        articleService.likeArticle(id);
        return ResponseEntity.ok("点赞成功");
    }

    @GetMapping("/top/{topN}")
    public ResponseEntity<List<Article>> getTopLikedArticles(@PathVariable int topN) {
        List<Article> topArticles = articleService.getTopLikedArticles(topN);
        return ResponseEntity.ok(topArticles);
    }
}

```

**详细解释**![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/0747448a2cbc41eeb95af64d73b2c4ca~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=662&h=387&s=37424&e=webp&b=fefdfd)

通过这种方式，我们可以利用Redis的原子操作和高性能特性来实现一个高效的点赞和排行榜功能。

每次用户点赞时，Redis都会原子性地更新点赞数，而获取排行榜时，我们可以快速地从Redis中检索和排序数据，从而提供实时的热门文章排行。

2. 实时分析
-------
```

针对Redis作为实时分析使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redis的Sorted Set来实现一个简单的用户在线时长统计和分析功能。

场景描述

假设我们正在开发一个在线教育平台，需要统计每个用户的在线时长，并根据这些数据生成实时的在线时长排行榜。

创建Spring Boot项目

配置Redis连接

在src/main/resources/application.properties中配置Redis服务器的连接信息：

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 用户在线时长服务接口和实现类

```
@Service
public class OnlineDurationService {

    @Autowired
    private StringRedisTemplate redisTemplate;

    public void updateUserOnlineDuration(String userId, long duration) {
        // 使用Sorted Set存储用户ID和在线时长
        redisTemplate.opsForZSet().incrementScore("user:online:duration",
userId, duration);
    }
```

```
    public Set<String> getTopUsersByOnlineDuration(int topN) {
        // 获取在线时长最长的前N个用户
        Set<String> topUsers = redisTemplate.opsForZSet().reverseRange("
user:online:duration", 0, topN - 1);
        return topUsers;
    }
}
```

2. 用户登录和登出逻辑

```
@Controller
public class UserController {

    @Autowired
    private OnlineDurationService onlineDurationService;

    @PostMapping("/user/{userId}/login")
    public ResponseEntity<String> userLogin(@PathVariable String userId)
 {
        // 用户登录逻辑，可以是任何触发登录的事件
        return ResponseEntity.ok("User " + userId + " logged in");
    }

    @PostMapping("/user/{userId}/logout")
    public ResponseEntity<String> userLogout(@PathVariable String userI
d) {
        // 用户登出时记录在线时长
        long duration = // 计算用户在线时长的逻辑
        onlineDurationService.updateUserOnlineDuration(userId, duration);
        return ResponseEntity.ok("User " + userId + " logged out");
    }
}
```

3. 获取在线时长排行榜

```
@RestController
@RequestMapping("/users")
```

```
public class UserRankController {

    @Autowired
    private OnlineDurationService onlineDurationService;

    @GetMapping("/online-duration/top/{topN}")
    public ResponseEntity<Set<String>> getTopUsersByOnlineDuration(@PathVariable int topN) {
        Set<String> topUsers = onlineDurationService.getTopUsersByOnlineDuration(topN);
        return ResponseEntity.ok(topUsers);
    }
}

```

详细解释

![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/65b748ce11114168abf78f0afaa6944f~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=552&h=341&s=33728&e=webp&b=fefdfd)

通过这种方式，我们可以利用Redis的Sorted Set来存储和排序用户的在线时长，实现一个高效的实时在线时长统计和分析功能。

每当用户登出时，系统都会更新用户的在线时长，并可以快速地根据在线时长对用户进行排名，从而提供一个动态的在线时长排行榜。这对于在线教育平台等需要监控用户活跃度的业务场景非常有用。

3. 分布式锁

针对Redis作为分布式锁的使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redisson作为客户端来实现分布式锁。

场景描述

假设我们有一个高流量的电子商务网站，需要执行一些资源密集型的操作，比如生成日报表。为了防止多个实例同时执行这些操作，我们需要一个分布式锁来确保每次只有一个实例可以执行这些操作。

创建Spring Boot项目

配置Redisson连接

在src/main/resources/application.properties或application.yml中配置Redisson客户端连接到Redis服务器：

```
# application.properties
redisson.address=redis://localhost:6379
```

或者

```
# application.yml
redisson:
  address: "redis://localhost:6379"
```

编写业务代码

1. 配置Redisson

创建一个配置类来配置Redisson客户端。

```
@Configuration
public class RedissonConfig {

    @Bean(destroyMethod = "shutdown")
    public RedissonClient redissonClient() {
        RedissonClientConfig config = new RedissonClientConfig();
        config.useSingleServer().setAddress(redisson.address);
```

```
        return Redisson.create(config);
    }

    @Value("${redisson.address}")
    private String redissonAddress;
}
```

2. 使用分布式锁

创建一个服务类来执行需要分布式锁保护的资源密集型操作。

```
@Service
public class ReportService {

    @Autowired
    private RedissonClient redissonClient;

    public void generateDailyReport() {
        RLock lock = redissonClient.getLock("dailyReportLock");
        try {
            // 尝试获取锁，最多等待3秒，锁的自动过期时间设置为10秒
            if (lock.tryLock(3, 10, TimeUnit.SECONDS)) {
                // 执行生成日报表的操作
                System.out.println("Generating daily report...");
                // 模拟长时间运行的任务
                TimeUnit.SECONDS.sleep(5);
                System.out.println("Daily report generated.");
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            // 释放锁
            lock.unlock();
        }
    }
}
```

3. 创建控制器

创建一个控制器来触发生成日报表的操作。

```
@RestController
@RequestMapping("/reports")
public class ReportController {

    @Autowired
    private ReportService reportService;

    @GetMapping("/daily")
    public ResponseEntity<String> generateDailyReport() {
        reportService.generateDailyReport();
        return ResponseEntity.ok("Daily report generation triggered.");
    }
}
```

详细解释

![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/f963c84a56b34c7dafb14b0091c4b589~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=584&h=263&s=20782&e=webp&b=fcfcfc)

通过这种方式，我们可以确保在分布式系统中，即使有多个实例运行，也只有一个实例可以执行生成日报表的操作，从而避免资源冲突和重复劳动。

Redisson客户端简化了Redis分布式锁的使用，使得在Spring Boot应用中实现分布式锁变得简单而高效。

4. 限流

针对Redis作为限流功能的使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redis来实现API的限流。

场景描述

假设我们正在开发一个公共API服务，该服务需要对外部请求进行限流，以防止滥用和过载。我们希望对每个IP地址每分钟的请求次数进行限制。

创建Spring Boot项目

配置Redis连接

在src/main/resources/application.properties中配置Redis服务器的连接信息：

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 创建限流注解

定义一个自定义注解，用于标识需要限流的API。

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface RateLimit {
    int limit() default 10; // 默认每分钟请求次数限制
    long timeout() default 60; // 默认时间窗口为60秒
}
```

2. 创建限流拦截器

实现一个拦截器来检查请求频率。

```
public class RateLimiterInterceptor implements HandlerInterceptor {

    private final RedisTemplate<String, Integer> redisTemplate;
    private final String rateLimitKeyPrefix = "rate_limit:";

    public RateLimiterInterceptor(RedisTemplate<String, Integer> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        String ip = request.getRemoteAddr();
        String methodName = ((MethodSignature) (handler)).getMethod().getName();
        String rateLimitKey = rateLimitKeyPrefix + methodName + ":" + ip;

        int currentCount = redisTemplate.opsForValue().increment(rateLimitKey);
        if (currentCount > 1) {
            // 如果当前计数大于1，则说明请求已超过限制
            response.sendError(HttpServletResponse.SC_TOO_MANY_REQUESTS, "Too many requests, please try again later.");
            return false;
        }

        // 设置过期时间
        redisTemplate.expire(rateLimitKey, RateLimit.class.cast(((MethodSignature) handler).getMethod().getAnnotation(RateLimit.class)).timeout(), TimeUnit.SECONDS);
        return true;
    }
}
```

3. 配置拦截器

配置拦截器以应用于所有控制器。

```
@Configuration
```

```
public class WebConfig implements WebMvcConfigurer {

    @Autowired
    private RateLimiterInterceptor rateLimiterInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(rateLimiterInterceptor);
    }
}
```

4. 应用限流注解

在需要限流的API上应用自定义的RateLimit注解。

```
@RestController
public class ApiController {

    @RateLimit(limit = 5, timeout = 60) // 每分钟最多5个请求
    @GetMapping("/api/resource")
    public ResponseEntity<String> getLimitedResource() {
        return ResponseEntity.ok("Access to limited resource");
    }
}
```

详细解释

![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/29fae48056714f49868801765c579e84~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=559&h=299&s=25568&e=webp&b=fefdfd)

通过这种方式，我们可以利用Redis的原子递增操作和键过期特性来实现API的限流。

每次请求都会检查当前IP的请求计数，如果超过限制，则返回429错误码（Too Many Requests）。

这有助于保护API免受滥用，并确保服务的稳定性和可用性。

5. 全页缓存

针对Redis作为全页缓存的使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redis来缓存整个页面的HTML内容。

场景描述

假设我们正在开发一个新闻网站，该网站的首页包含多个新闻文章的摘要信息。由于首页访问频率很高，我们希望将整个首页的内容缓存起来，以减少数据库的查询次数和页面渲染时间。

创建Spring Boot项目

配置Redis连接

在src/main/resources/application.properties中配置Redis服务器的连接信息：

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 创建新闻文章服务

```
@Service
public class NewsService {
```

```
    // 假设有一个方法来获取新闻列表
    public List<Article> getNewsList() {
        // 这里是获取新闻列表的逻辑
        return Collections.emptyList();
    }
}
```

2. 配置Redis缓存

创建一个配置类来设置Spring Cache和Redis缓存。

```
@Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public CacheManager cacheManager(RedisConnectionFactory connec
tionFactory) {
        RedisCacheManager cacheManager = new RedisCacheManager(co
nnectionFactory);
        // 设置缓存过期时间（例如5分钟）
        cacheManager.setDefaultExpiration(300);
        return cacheManager;
    }
}
```

3. 创建控制器和视图

创建一个控制器来返回首页，并使用Redis缓存整个页面。

```
@Controller
public class NewsController {

    @Autowired
    private NewsService newsService;
```

```
    @Autowired
    private CacheManager cacheManager;

    @GetMapping("/")
    @Cacheable(value = "homePage", condition = "#root.caches[0].name
== 'homePage'")
    public String homePage(Model model) {
        // 尝试从缓存中获取页面
        model.addAttribute("newsList", newsService.getNewsList());
        return "home";
    }
}
```

4. 创建Thymeleaf模板

创建一个Thymeleaf模板home.html来渲染首页。

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>首页</title>
</head>
<body>
    <h1>新闻首页</h1>
    <div th:each="article : ${newsList}">
        <h2 th:text="${article.title}"></h2>
        <p th:text="${article.summary}"></p>
    </div>
</body>
</html>
```

详细解释

![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/a8f8c9b46ba04d319141d4bc491a2598~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=560&h=334&s=31862&e=webp&b=fefdfd)

通过这种方式，我们可以利用Redis来缓存整个页面的内容。

首页的访问非常频繁，通过缓存可以显著减少数据库的查询次数和页面渲染时间，提高网站的响应速度和性能。

此外，Spring的缓存抽象和Thymeleaf模板使得实现全页缓存变得简单而高效。

6. 社交功能

针对Redis作为社交功能存储的使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redis来存储用户的社交关系信息，如好友列表和用户状态更新。

场景描述

假设我们正在开发一个社交网络平台，用户可以添加好友，并且可以发布状态更新。我们需要存储每个用户的好友列表以及状态更新的时间线。

创建Spring Boot项目

配置Redis连接

在src/main/resources/application.properties中配置Redis服务器的连接信息：

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 定义用户和状态更新实体类

```
public class User {
    private String id;
    private String name;
    // 省略构造函数、getter和setter方法
}

public class StatusUpdate {
    private String userId;
    private String content;
    private Instant timestamp;
    // 省略构造函数、getter和setter方法
}
```

2. 创建社交服务

```
@Service
public class SocialService {

    @Autowired
    private StringRedisTemplate redisTemplate;

    public void addFriend(String userOneId, String userTwoId) {
        // 使用集合存储用户的好友列表
        redisTemplate.opsForSet().add("friends:" + userOneId, userTwoId);
        redisTemplate.opsForSet().add("friends:" + userTwoId, userOneId);
    }

    public Set<String> getFriends(String userId) {
        // 获取用户的好友列表
        return redisTemplate.opsForSet().members("friends:" + userId);
    }

    public void postStatusUpdate(String userId, String content) {
        // 使用列表存储用户的状态更新时间线
        StatusUpdate statusUpdate = new StatusUpdate(userId, content, In
stant.now());
        redisTemplate.opsForList().rightPush("timeline:" + userId, statusUp
date);
```

```
    }

    public List<StatusUpdate> getStatusUpdates(String userId) {
        // 获取用户的状态更新时间线
        return redisTemplate.opsForList().range("timeline:" + userId, 0, -1);
    }
}

```

3. 创建控制器

```
@RestController
@RequestMapping("/social")
public class SocialController {

    @Autowired
    private SocialService socialService;

    @PostMapping("/addFriend")
    public ResponseEntity<String> addFriend(@RequestParam String user
OneId, @RequestParam String userTwoId) {
        socialService.addFriend(userOneId, userTwoId);
        return ResponseEntity.ok("Friends added successfully");
    }

    @GetMapping("/friends/{userId}")
    public ResponseEntity<Set<String>> getFriends(@PathVariable String
userId) {
        Set<String> friends = socialService.getFriends(userId);
        return ResponseEntity.ok(friends);
    }

    @PostMapping("/status")
    public ResponseEntity<String> postStatusUpdate(@RequestParam Stri
ng userId, @RequestParam String content) {
        socialService.postStatusUpdate(userId, content);
        return ResponseEntity.ok("Status updated successfully");
    }

    @GetMapping("/timeline/{userId}")
    public ResponseEntity<List<StatusUpdate>> getStatusUpdates(@Path
Variable String userId) {
        List<StatusUpdate> updates = socialService.getStatusUpdates(user
Id);
```

```
        return ResponseEntity.ok(updates);
    }
}

```

详细解释

![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/6178dd05393343dfa969fe530fe3bbc0~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=568&h=320&s=25116&e=webp&b=fcfcfc)

通过这种方式，我们可以利用Redis的高性能和数据结构特性来实现社交网络平台中的社交功能。

Redis的Set和List数据结构非常适合存储和管理好友关系和状态更新时间线，能够提供快速的读写性能，满足社交网络平台的需求。

7. 实时推荐系统

针对Redis作为实时推荐系统存储的使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redis来存储用户行为数据和偏好，以及提供一个简单的推荐功能。

场景描述

假设我们正在开发一个电子商务平台，我们希望根据用户的浏览和购买历史来推荐商品。我们将使用Redis来存储用户的这些行为数据，并根据这些数据生成推荐。

创建Spring Boot项目

配置Redis连接

在src/main/resources/application.properties中配置Redis服务器的连接信息：

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 创建商品和用户实体类

```
public class Product {
    private String id;
    private String name;
    // 省略构造函数、getter和setter方法
}

public class User {
    private String id;
    private String username;
    // 省略构造函数、getter和setter方法
}
```

2. 创建推荐服务

```
@Service
public class RecommendationService {

    @Autowired
    private StringRedisTemplate redisTemplate;

    public void recordView(String userId, String productId) {
        // 记录用户查看的商品
        redisTemplate.opsForList().leftPush("user:" + userId + ":views", pro
ductId);
    }
```

```
    public List<String> recommendProducts(String userId) {
        // 简单推荐算法：返回用户查看次数最多的商品
        Set<String> viewedProducts = redisTemplate.opsForSet().members(
”user:” + userId + ”:views”);
        Map<String, Long> productViewCounts = new HashMap<>();
        viewedProducts.forEach(productId -> {
            long count = redisTemplate.opsForValue().decrement(”user:” + u
serId + ”:views:” + productId);
            productViewCounts.put(productId, count);
        });

        return productViewCounts.entrySet().stream()
                .sorted(Map.Entry.<String, Long>comparingByValue().reversed
())
                .map(Map.Entry::getKey)
                .collect(Collectors.toList());
    }
}
```

3. 创建控制器

```
@RestController
@RequestMapping(”/recommendations”)
public class RecommendationController {

    @Autowired
    private RecommendationService recommendationService;

    @PostMapping(”/view”)
    public ResponseEntity<String> recordProductView(@RequestParam St
ring userId, @RequestParam String productId) {
        recommendationService.recordView(userId, productId);
        return ResponseEntity.ok(”View recorded”);
    }

    @GetMapping(”/products”)
    public ResponseEntity<List<String>> getRecommendations(@Request
Param String userId) {
        List<String> recommendedProducts = recommendationService.reco
mmendProducts(userId);
        return ResponseEntity.ok(recommendedProducts);
    }
}
```

```

**详细解释**

![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/686abf75c7e543fc8b37ddcd117fec25~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=566&h=483&s=39364&e=webp&b=fefdfd)

通过这种方式，我们可以利用Redis的高性能和简单的数据结构来快速记录用户行为并生成推荐。

虽然这里的推荐算法非常简单，但它展示了如何使用Redis来实现实时推荐系统的基础功能。

在实际应用中，推荐算法可能会更复杂，涉及机器学习模型和更丰富的用户行为数据。

8. 地理位置信息
---

针对Redis作为地理位置信息存储的使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redis的Geospatial索引来实现基于地理位置的推荐功能。

**场景描述**

假设我们正在开发一款基于位置的社交应用，用户可以查看附近的其他用户或地点。我们需要存储用户的地理位置，并能够查询给定位置附近的用户。

**创建Spring Boot项目**

**配置Redis连接**

在src/main/resources/application.properties中配置Redis服务器的连接信息：
```

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 创建用户实体类

```
public class User {
    private String id;
    private String name;
    private double longitude;
    private double latitude;
    // 省略构造函数、getter和setter方法
}
```

2. 创建地理位置服务

```
@Service
public class GeoLocationService {

    @Autowired
    private RedisTemplate<String, User> redisTemplate;

    public void addLocation(String userId, double longitude, double latitud
e) {
        User user = new User(userId, "username", longitude, latitude);
        // 使用Geospatial索引存储用户位置
        redisTemplate.opsForGeo().add("userLocations", new GeoLocation(
user.getLongitude(), user.getLatitude()), userId);
    }

    public List<User> getUsersNearby(double longitude, double latitude, d
ouble radius) {
        // 查询给定位置附近的用户
```

```
        List<GeoWithin> nearbyUsersGeo = redisTemplate.opsForGeo().radius("userLocations",
                new Circle(new GeoCoordinate(latitude, longitude), radius),
                RedisGeoCommands.GeoRadiusCommandArgs.newGeoRadiusArgs());

        List<User> nearbyUsers = new ArrayList<>();
        for (GeoWithin geoWithin : nearbyUsersGeo) {
            nearbyUsers.add(redisTemplate.opsForValue().get(geoWithin.getMember()));
        }
        return nearbyUsers;
    }
}

```

3. 创建控制器

```
@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private GeoLocationService geoLocationService;

    @PostMapping("/addLocation")
    public ResponseEntity<String> addLocation(@RequestParam String userId,
                                              @RequestParam double longitude,
                                              @RequestParam double latitude) {
        geoLocationService.addLocation(userId, longitude, latitude);
        return ResponseEntity.ok("User location added");
    }

    @GetMapping("/nearby")
    public ResponseEntity<List<User>> getUsersNearby(@RequestParam double longitude,
                                                     @RequestParam double latitude,
                                                     @RequestParam double radius) {
        List<User> nearbyUsers = geoLocationService.getUsersNearby(longitude, latitude, radius);
        return ResponseEntity.ok(nearbyUsers);
    }
}
```

```

**详细解释**

![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/e705eb09bc18403e844fe214ce6795a8~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=555&h=454&s=32474&e=webp&b=fefdfd)

通过这种方式，我们可以利用Redis的Geospatial索引来存储和查询地理位置信息。这对于需要基于地理位置提供服务的应用非常有用，如社交网络、共享出行、本地服务推荐等。

Redis的Geospatial索引提供了高效的邻近查询功能，可以快速找到指定范围内的用户或其他地理位置相关的实体。

9. 时间序列数据
---

针对Redis作为时间序列数据存储的使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redis来存储和查询时间序列数据。

**场景描述**

假设我们正在开发一个监控系统，需要记录服务器的CPU使用率随时间变化的数据。我们将使用Redis的时间序列数据结构来存储这些监控数据，并能够查询任意时间范围内的CPU使用率。

**创建Spring Boot项目**

**配置Redis连接**

在src/main/resources/application.properties中配置Redis服务器的连接信息：
```

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 创建监控数据实体类

```
public class CpuUsageData {
    private Instant timestamp;
    private double cpuUsage;

    // 省略构造函数、getter和 setter 方法
}
```

2. 创建监控服务

```
@Service
public class MonitoringService {

    @Autowired
    private LettuceConnectionFactory connectionFactory;

    public void logCpuUsage(String serverId, double cpuUsage) {
        // 记录CPU使用率数据
        CpuUsageData data = new CpuUsageData(Instant.now(), cpuUsage
);
        // 使用Lettuce客户端的命令执行器来与RedisTimeSeries模块交互
        StatefulRedisConnection<String, CpuUsageData> connection = con
nectionFactory.connect();
        try {
            RedisTimeSeriesCommands<String, CpuUsageData> ts = conne
ction.sync();
            ts.add(serverId, data.getTimestamp().toEpochMilli() / 1000, data)
;
        } finally {
            connection.close();
```

```
        }
    }

    public List<CpuUsageData> getCpuUsageHistory(String serverId, Instant start, Instant end) {
        // 查询指定时间范围内的CPU使用率历史数据
        List<CpuUsageData> history = new ArrayList<>();
        StatefulRedisConnection<String, CpuUsageData> connection = connectionFactory.connect();
        try {
            RedisTimeSeriesCommands<String, CpuUsageData> ts = connection.sync();
            Range range = Range.create(start.toEpochMilli() / 1000, end.toEpochMilli() / 1000);
            Cursor<CpuUsageData> cursor = ts.rangeRead(serverId, range);
            while (cursor.hasNext()) {
                history.add(cursor.next().getValue());
            }
        } finally {
            connection.close();
        }
        return history;
    }
}

```

3. 创建控制器

```
@RestController
@RequestMapping("/monitoring")
public class MonitoringController {

    @Autowired
    private MonitoringService monitoringService;

    @PostMapping("/logCpuUsage")
    public ResponseEntity<String> logCpuUsage(@RequestParam String serverId, @RequestParam double cpuUsage) {
        monitoringService.logCpuUsage(serverId, cpuUsage);
        return ResponseEntity.ok("CPU usage logged");
    }

    @GetMapping("/cpuUsageHistory")
    public ResponseEntity<List<CpuUsageData>> getCpuUsageHistory(@
```

```
RequestParam String serverId,
                                                        @RequestParam Instant sta
rt,
                                                        @RequestParam Instant en
d) {
        List<CpuUsageData> history = monitoringService.getCpuUsageHist
ory(serverId, start, end);
        return ResponseEntity.ok(history);
    }
}

```

详细解释

通过这种方式，我们可以利用Redis的RedisTimeSeries模块来存储和查询时间序列数据。这对于需要监控和分析随时间变化的数据的应用非常有用，如服务器监控、网站访问量分析等。

RedisTimeSeries提供了高效的时间序列数据存储和查询功能，可以快速插入和检索大量时间戳数据。

10. 任务调度

针对Redis作为任务调度的使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redis的延迟队列特性来实现任务调度。

场景描述

假设我们正在开发一个定时任务管理系统，需要安排一些任务在将来的某个时间点执行。我们将使用Redis的schedule命令来安排任务的执行。

配置Redis连接

在src/main/resources/application.properties中配置Redis服务器的连接信息：

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 创建任务调度服务

```
@Service
public class TaskSchedulingService {

    @Autowired
    private RedisTemplate<String, Runnable> redisTemplate;

    public void scheduleTask(Runnable task, long delay, TimeUnit timeUnit) {
        // 将任务和延迟时间存储到Redis中
        redisTemplate.opsForValue().set(
            "task:" + task.hashCode(),
            task,
            timeUnit.toSeconds(delay),
            timeUnit
        );
        // 使用schedule命令安排任务在未来执行
        String scheduleCommand = String.format(
            "SCHEDULE %d %s",
            System.currentTimeMillis() + timeUnit.toMillis(delay),
            "task:" + task.hashCode()
        );
        redisTemplate.execute((RedisConnection connection) -> {
            connection.schedule(scheduleCommand);
            return null;
        });
    }
}
```

```

2. 创建具体的任务



```
public class SampleTask implements Runnable {
    @Override
    public void run() {
        System.out.println("Task is running: " + LocalDateTime.now());
        // 执行任务逻辑
    }
}

```

3. 创建控制器



```
@RestController
@RequestMapping("/tasks")
public class TaskController {

    @Autowired
    private TaskSchedulingService taskSchedulingService;

    @PostMapping("/schedule")
    public ResponseEntity<String> scheduleTask(@RequestParam long de
lay, @RequestParam TimeUnit timeUnit) {
        taskSchedulingService.scheduleTask(new SampleTask(), delay, time
Unit);
        return ResponseEntity.ok("Task scheduled for execution at " + Loc
alDateTime.now().plusNanos(timeUnit.toNanos(delay)));
    }
}

```

**详细解释**


![图片](https://p3-juejin.byteimg.com/tos-cn-i-
k3u1fbpfcp/49f8b32969ac497eb8d6cabe9e43b694~tplv-k3u1fbpfcp-jj-
mark:3024:0:0:0:q75.awebp#?w=566&h=285&s=23436&e=webp&b=fefdf
d)
```

通过这种方式，我们可以利用Redis的schedule命令来安排任务的执行。这对于需要执行定时任务的应用非常有用，如定时数据备份、定时发送通知等。

通过Redis的延迟队列特性，我们可以简化任务调度的复杂性，并且能够灵活地安排任务在未来的任意时间点执行。

11. 数据共享

针对Redis作为数据共享的使用场景，下面是一个Java Spring Boot应用的案例，其中使用Redis来实现微服务架构中的服务间数据共享。

场景描述

假设我们有一个电商平台，它由多个微服务组成，比如用户服务、产品服务和订单服务。这些服务需要共享购物车数据，以确保用户在平台上的购物体验是连贯的。我们将使用Redis来存储和共享购物车数据。

创建Spring Boot项目

配置Redis连接

在src/main/resources/application.properties中配置Redis服务器的连接信息：

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 创建购物车项实体类

```
public class CartItem {
    private String productId;
    private int quantity;
    // 省略构造函数、getter和setter方法
}
```

2. 创建购物车服务

```
@Service
public class CartService {

    @Autowired
    private StringRedisTemplate redisTemplate;

    public void addToCart(String cartId, String productId, int quantity) {
        // 将购物车项存储到Redis的Hash结构中
        redisTemplate.opsForHash().put("cart:" + cartId, productId, quantity
);
    }

    public Map<String, Integer> getCart(String cartId) {
        // 从Redis获取购物车内容
        return redisTemplate.opsForHash().entries("cart:" + cartId);
    }
}
```

3. 创建控制器

```
@RestController
@RequestMapping("/cart")
public class CartController {

    @Autowired
    private CartService cartService;
```

```
    @PostMapping("/{cartId}/items")
    public ResponseEntity<String> addToCart(@PathVariable String cartId
,
                                            @RequestParam String productId,
                                            @RequestParam int quantity) {
        cartService.addToCart(cartId, productId, quantity);
        return ResponseEntity.ok("Item added to cart");
    }

    @GetMapping("/{cartId}")
    public ResponseEntity<Map<String, Integer>> getCart(@PathVariable
String cartId) {
        Map<String, Integer> cart = cartService.getCart(cartId);
        return ResponseEntity.ok(cart);
    }
}
```

详细解释

![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/ee4ddde190834df6887c54c5f7338d4c~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=564&h=400&s=24332&e=webp&b=fdfdfd)

通过这种方式，我们可以利用Redis的高性能和数据共享能力来实现微服务架构中的服务间数据共享。

购物车数据被存储在Redis中，可以被不同的微服务实例访问和修改，确保了数据的一致性和实时性。

这对于需要高度协同工作的分布式系统非常有用，如电商平台、在线协作工具等。

12. 持久化

针对Redis作为任务调度使用场景，下面是一个Java Spring Boot应用的案例，其中使用Spring的@Scheduled注解与Redisson结合来实现任务调度。

场景描述

假设我们有一个自动化的营销平台，需要定期（例如每天凌晨1点）执行一些任务，比如发送时事通讯邮件给订阅用户。我们将使用Spring的定时任务功能结合Redisson来确保分布式环境下任务的准时和准确执行。

创建Spring Boot项目

配置Redis连接

在src/main/resources/application.properties中配置Redis服务器的连接信息：

```
spring.redis.host=localhost
spring.redis.port=6379
```

编写业务代码

1. 创建任务执行服务

```
@Service
public class ScheduledTaskService {

    public void executeTask() {
        // 执行任务的逻辑，例如发送邮件
        System.out.println("Executing scheduled task: " + LocalDateTime.now());
    }
}
```

2. 配置Redisson

创建一个配置类来配置Redisson客户端。

```
@Configuration
public class RedissonConfig {

    @Bean(destroyMethod = "shutdown")
    public RedissonClient redissonClient() {
        RedissonClientConfig config = new RedissonClientConfig();
        config.useSingleServer().setAddress("redis://" + spring.redis.host +
 ":" + spring.redis.port);
        return Redisson.create(config);
    }

    @Value("${spring.redis.host}")
    private String redisHost;

    @Value("${spring.redis.port}")
    private int redisPort;
}
```

3. 创建定时任务配置

使用Redisson的RedissonScheduledExecutorService来创建一个分布式的调度器。

```
@Configuration
public class ScheduledConfig {

    @Bean
    public RedissonScheduledExecutorService redissonScheduledExecuto
rService(RedissonClient redissonClient) {
        return redissonClient.getExecutorService("myScheduler");
    }
}
```

4. 创建定时任务

使用Spring的@Scheduled注解和Redisson的调度器来执行定时任务。

```
@Component
public class ScheduledTasks {

    @Autowired
    private ScheduledTaskService taskService;

    @Autowired
    private RedissonScheduledExecutorService scheduler;

    @Scheduled(cron = "0 0 1 * * ?") // 每天凌晨1点执行
    public void scheduledTask() {
        scheduler.schedule(() -
> taskService.executeTask(), 0, TimeUnit.SECONDS);
    }
}

```

详细解释

![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/932160cfe0544d27a724da97d84bee47~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=579&h=339&s=23048&e=webp&b=fdfdfd)

通过这种方式，我们可以利用Spring的定时任务功能和Redisson的分布式调度器来实现任务调度。

这确保了即使在分布式系统中，任务也能准时和准确地执行，避免了任务执行的冲突和重复。

这对于需要定时执行的任务，如发送时事通讯、数据备份、报告生成等场景非常有用。

最后说一句(求!别白嫖！)

如果这篇文章对您有所帮助，或者有所启发的话，求一键三连：点赞、转发、在看。

公众号：**woniuxgg**，在公众号中回复：笔记 就可以获得蜗牛为你精心准备的java实战语雀笔记，回复面试、开发手册、有超赞的粉丝福利！

原文链接: https://juejin.cn/post/7379060209367253033