```
            // 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

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> redisTe
mplate) {
        this.redisTemplate = redisTemplate;
    }

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

        int currentCount = redisTemplate.opsForValue().increment(rateLimit
Key);
```

```
        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);
    }
}
```

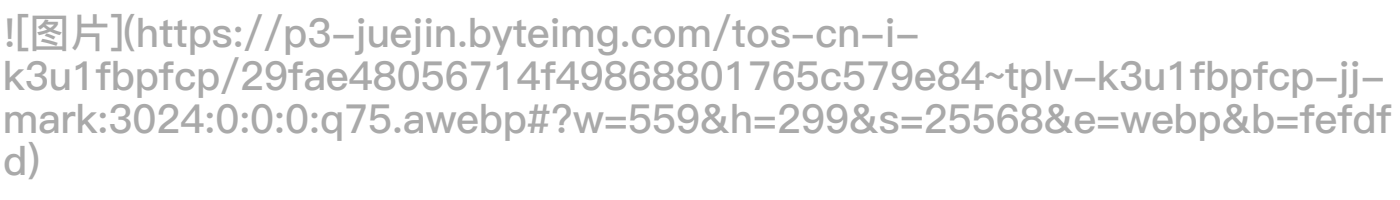
4. 应用限流注解

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



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。

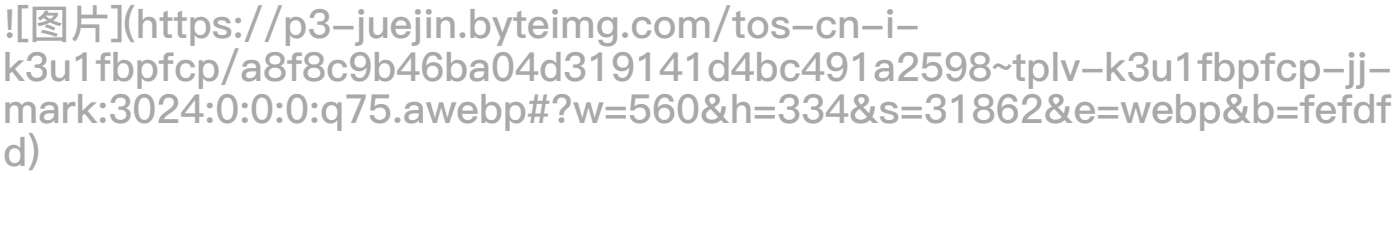
```
@Controller
public class NewsController {

    @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



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 y, 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

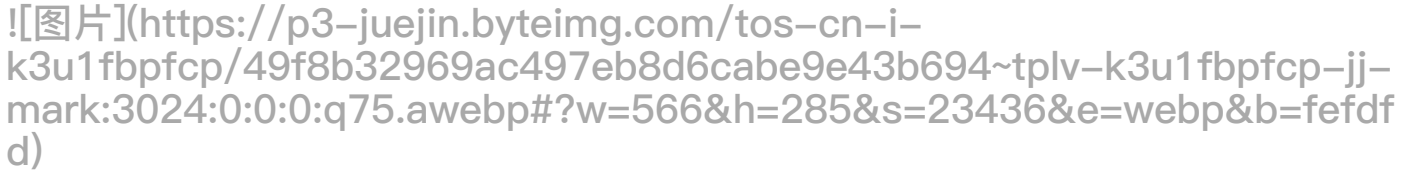
```
public class SampleTask implements Runnable {
    @Override
    public void run() {
        System.out.println("Task is running: " + LocalDateTime.now());
        // 执行任务逻辑
    }
}
```

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 " + Local
DateTime.now().plusNanos(timeUnit.toNanos(delay)));
    }
}
```

详细解释



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
}
```

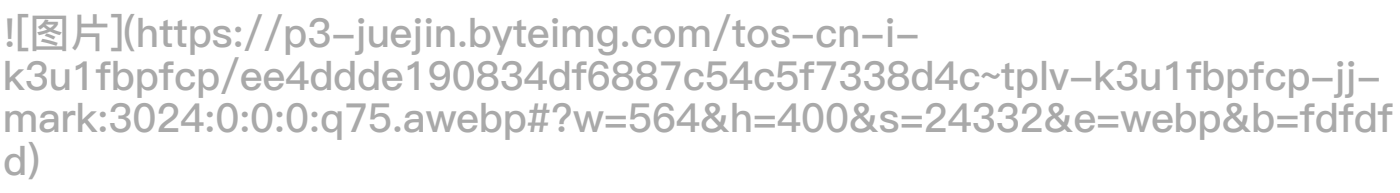
2. 创建购物车服务

```
@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**



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.n
ow());
    }
}

```

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。

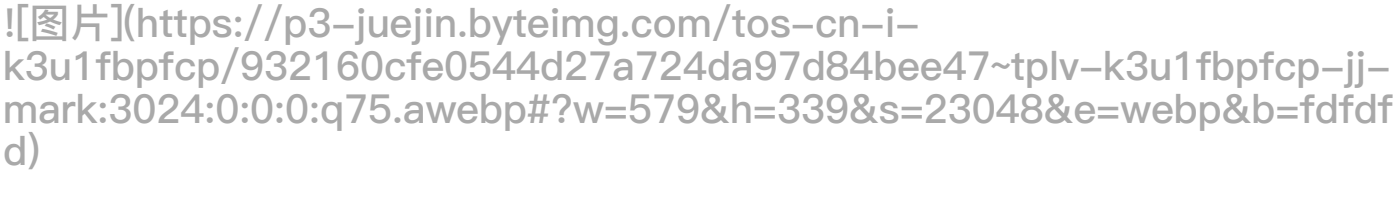
```
@Configuration
public class ScheduledConfig {

    @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



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