

[🏠 / 开发指南 / 后端手册](#)[👤 芋道源码](#) [📅 2022-04-09](#)

幂等性（防重复提交）

`yudao-spring-boot-starter-protection`  技术组件，由它的 `idempotent`  包，提供声明式的幂等特性，可防止重复请求。例如说，用户快速的双击了某个按钮，前端没有禁用该按钮，导致发送了两次重复的请求。

```
// UserController.java
```

```
@Idempotent(timeout = 10, timeUnit = TimeUnit.SECONDS, message = "正在添加用户中，")
@PostMapping("/user/create")
public String createUser(User user){
    userService.createUser(user);
    return "添加成功";
}
```

- 每 10 秒钟，所有用户，只能操作一次

疑问：如果想按照每个用户，或者每个 IP，限制请求呢？

可设置该注解的 `keyResolver` 属性，可选择的有：

- `DefaultIdempotentKeyResolver`：全局级别
- `UserIdempotentKeyResolver`：用户级别
- `ExpressionIdempotentKeyResolver`：自定义级别，通过 `keyArg` 属性指定 Spring EL 表达式

1. 实现原理

友情提示：

它的实现原理，和《[请求限流（RateLimiter）](#)》比较接近哈。

它的实现原理非常简单，针对相同参数的方法，一段时间内，有且仅能执行一次。执行流程如下：

- ① 在方法执行前，根据参数对应的 Key 查询是否存在：
 - 如果**存在**，说明正在执行中，则进行报错。

- 如果**不在**，则计算参数对应的 Key，存储到 Redis 中，并设置过期时间，即标记正在执行中。

默认参数的 Redis Key 的计算规则由 [DefaultIdempotentKeyResolver](#) 实现，使用 MD5(方法名 + 方法参数)，避免 Redis Key 过长。

- ② 方法执行完成，**不会**主动删除参数对应的 Key。

如果希望会**主动删除** Key，可以使用《开发指南——分布式锁》提供的 `@Lock` 来实现幂等性。

😊 从本质上来说，`idempotent` 包提供的幂等特性，本质上也是基于 Redis 实现的分布式锁。

- ③ 如果方法执行时间较长，超过 Key 的过期时间，则 Redis 会自动删除对应的 Key。因此，需要大概评估下，避免方法的执行时间超过过期时间。

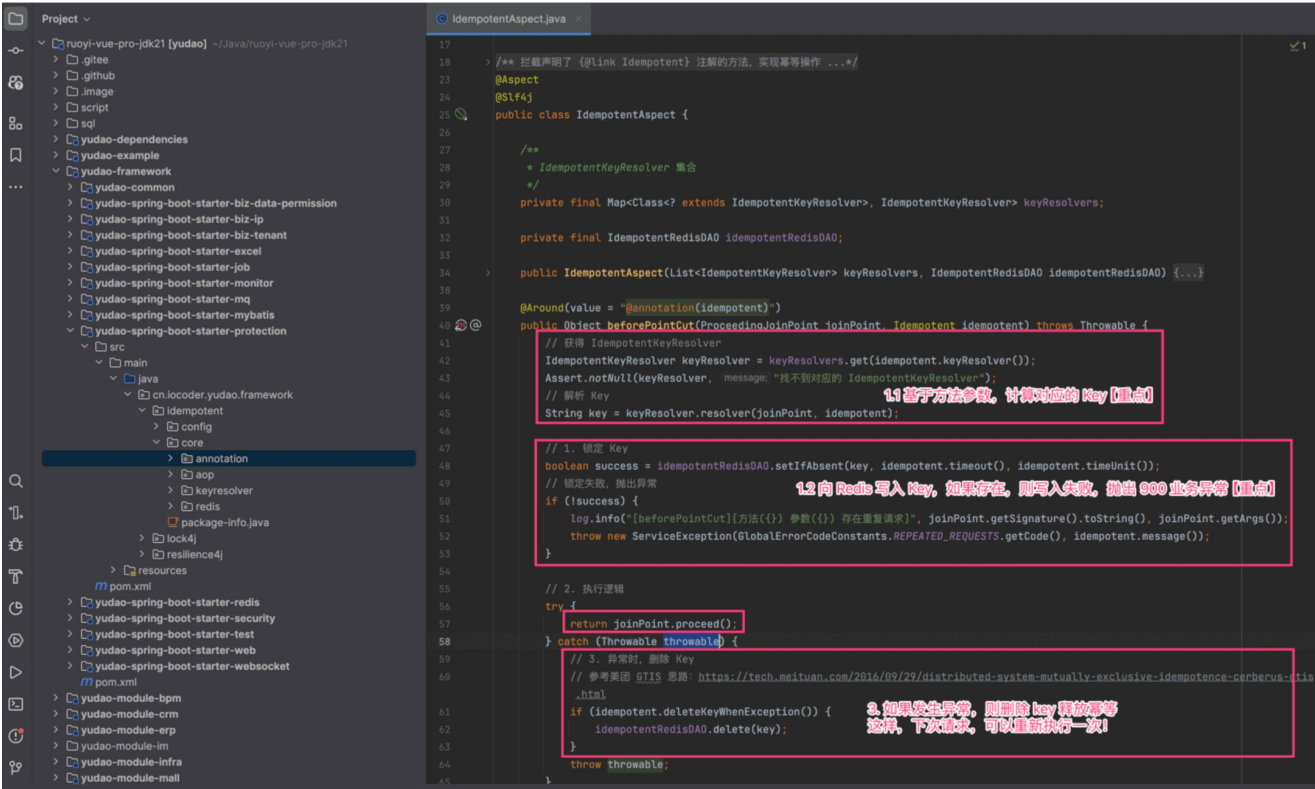
- ④ 如果方法执行发生 Exception 异常时，默认会删除 Key，避免下次请求无法正常执行，此处参考《美团 GTIS》。

2. @Idempotent 注解

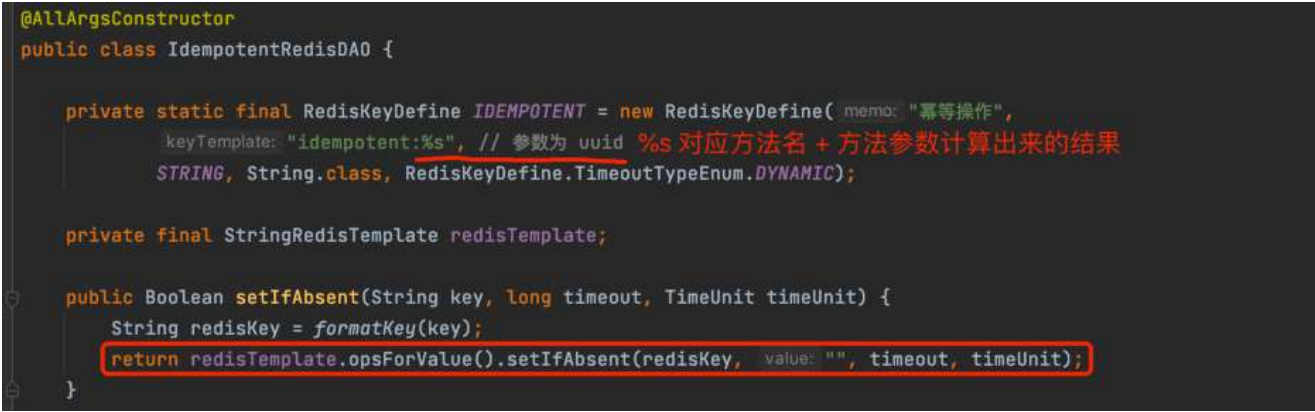
`@Idempotent` 注解，声明在方法上，表示该方法需要开启幂等性。代码如下：

```
13 幂等注解
14  Author: 芋道源码
15
16  @Target({ElementType.METHOD})
17  @Retention(RetentionPolicy.RUNTIME)
18  public @interface Idempotent {
19
20      /**
21       * 幂等的超时时间，默认为 1 秒
22       *
23       * 注意，如果执行时间超过它，请求还是会进来
24       */
25      int timeout() default 1;
26
27      /** 时间单位，默认为 SECONDS 秒 */
28      TimeUnit timeUnit() default TimeUnit.SECONDS;
29
30      /** 提示信息，正在执行中的提示 */
31      String message() default "重复请求，请稍后重试";
32
33      /**
34       * 使用的 Key 解析器
35       *
36       * @see DefaultIdempotentKeyResolver 全局级别
37       * @see UserIdempotentKeyResolver 用户级别
38       * @see ExpressionIdempotentKeyResolver 自定义表达式，通过 {@link #keyArg()} 计算
39       */
40      Class<? extends IdempotentKeyResolver> keyResolver() default DefaultIdempotentKeyResolver.class;
41
42      /** 使用的 Key 参数 */
43      String keyArg() default "";
44
45      /**
46       * 删除 Key，当发生异常时
47       *
48       * 问题：为什么发生异常时，需要删除 Key 呢？
49       * 回答：发生异常时，说明业务发生错误，此时需要删除 Key，避免下次请求无法正常执行。
50       *
51       * 问题：为什么不搞 deleteWhenSuccess 执行成功时，需要删除 Key 呢？
52       * 回答：这种情况下，本质上是分布式锁，推荐使用 @Lock4j 注解
53       */
54      boolean deleteKeyWhenException() default true;
55  }
```

- ① 对应的 AOP 切面是 `IdempotentAspect` 类，核心就 10 行左右的代码，如下图所示：



② 对应的 Redis Key 的前缀是 `idempotent:%s`，可见 `IdempotentRedisDAO` 类，如下图所示：



3. 使用示例

本小节，我们实现 `/admin-api/infra/test-demo/get` RESTful API 接口的幂等性。

① 在 `pom.xml` 文件中，引入 `yudao-spring-boot-starter-protection` 依赖。

```
<dependency>
  <groupId>cn.iocoder.boot</groupId>
  <artifactId>yudao-spring-boot-starter-protection</artifactId>
</dependency>
```

② 在该 API 接口的对应方法上，添加 `@Idempotent` 注解。代码如下：

```
// TestDemoController.java
```

```
@GetMapping("/get")
@Idempotent(timeout = 10, message = "重复请求，请稍后重试")
public CommonResult<TestDemoRespVO> getTestDemo(@RequestParam("id") Long id) {
    // ... 省略代码
}
```

③ 调用该 API 接口，执行成功。

A terminal window showing a Redis command: `127.0.0.1:6379> keys idempotent*`. The text "考虑到 Redis Key 的长度，所以是" is overlaid in red. Below the command, there is a large number of keys listed, indicating a successful search for keys matching the pattern.

④ 再次调用该 API 接口，被幂等性拦截，执行失败。

```
{
  "code": 900,
  "data": null,
  "msg": "重复请求，请稍后重试"
}
```

← [分布式锁](#)

[请求限流 \(RateLimiter\)](#) →



Theme by [Vdoing](#) | Copyright © 2019-2024 芋道源码 | MIT License