芋道源码 2022-04-05

分布式锁

`yudao-spring-boot-starter-protection` 技术组件，使用 Redis 实现分布式锁的功能，它有 2 种使用方式：

- 编程式锁：基于 Redisson 框架提供的各种 分布式锁
- 声明式锁：基于 Lock4j 框架的 `@Lock4j` 注解

> **Redis 分布式锁的实现原理？**
>
> 参见 《Redis 实现原理与源码解析系列》 文章。

1. 编程式锁

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson-spring-boot-starter</artifactId>
</dependency>
```

1.1 Redisson 配置

无需配置。因为在 Redis 缓存 中，进行了 Spring Data Redis + Redisson 的配置。

1.2 实战案例

`yudao-module-pay` 模块的 `notify` 功能，使用到分布式锁，确保**每个**支付通知任务有且仅有一个在执行。下面，来看看这个案例是如何实现的。

> **友情提示：**
>
> 建议你已经阅读过 《开发指南 —— Redis 缓存》 文档。

① 在 RedisKeyConstants 类中，定义通知任务使用的分布式锁的 Redis Key。如下图所示：

```
public interface RedisKeyConstants {

    RedisKeyDefine PAY_NOTIFY_LOCK = new RedisKeyDefine( memo: "通知任务的分布式锁",
            keyTemplate: "pay_notify:lock:", // 参数来自 DefaultLockKeyBuilder 类
            RedisKeyDefine.KeyTypeEnum.HASH, RLock.class, RedisKeyDefine.TimeoutTypeEnum.DYNAMIC); // Redisson 的 Lock 锁，使用 Hash 数据结构

}
```

② 创建 PayNotifyLockRedisDAO 类，使用 RedisClient 实现分布式锁的加锁与解锁。如下图所示：

```java
@Repository
public class PayNotifyLockRedisDAO {

    @Resource
    private RedissonClient redissonClient;

    public void lock(Long id, Long timeoutMillis, Runnable runnable) {
        String lockKey = formatKey(id);
        RLock lock = redissonClient.getLock(lockKey);
        try {
            lock.lock(timeoutMillis, TimeUnit.MILLISECONDS); // ① 添加 Redis 分布式锁，时长通过参数
            // 执行逻辑
            runnable.run(); // ② 执行逻辑
        } finally {
            lock.unlock(); // ③ 释放 Redis 分布式锁
        }
    }

    private static String formatKey(Long id) {
        return String.format(PAY_NOTIFY_LOCK.getKeyTemplate(), id);
    }

}
```

为什么通过 runnable 传递逻辑呢？
让调用的 Service 不用关心分布式锁的实现，完成自己的业务逻辑就好

③ 在 PayNotifyServiceImpl 执行指定的支付通知任务时，通过 PayNotifyLockRedisDAO 获得分布式锁。如下图所示：

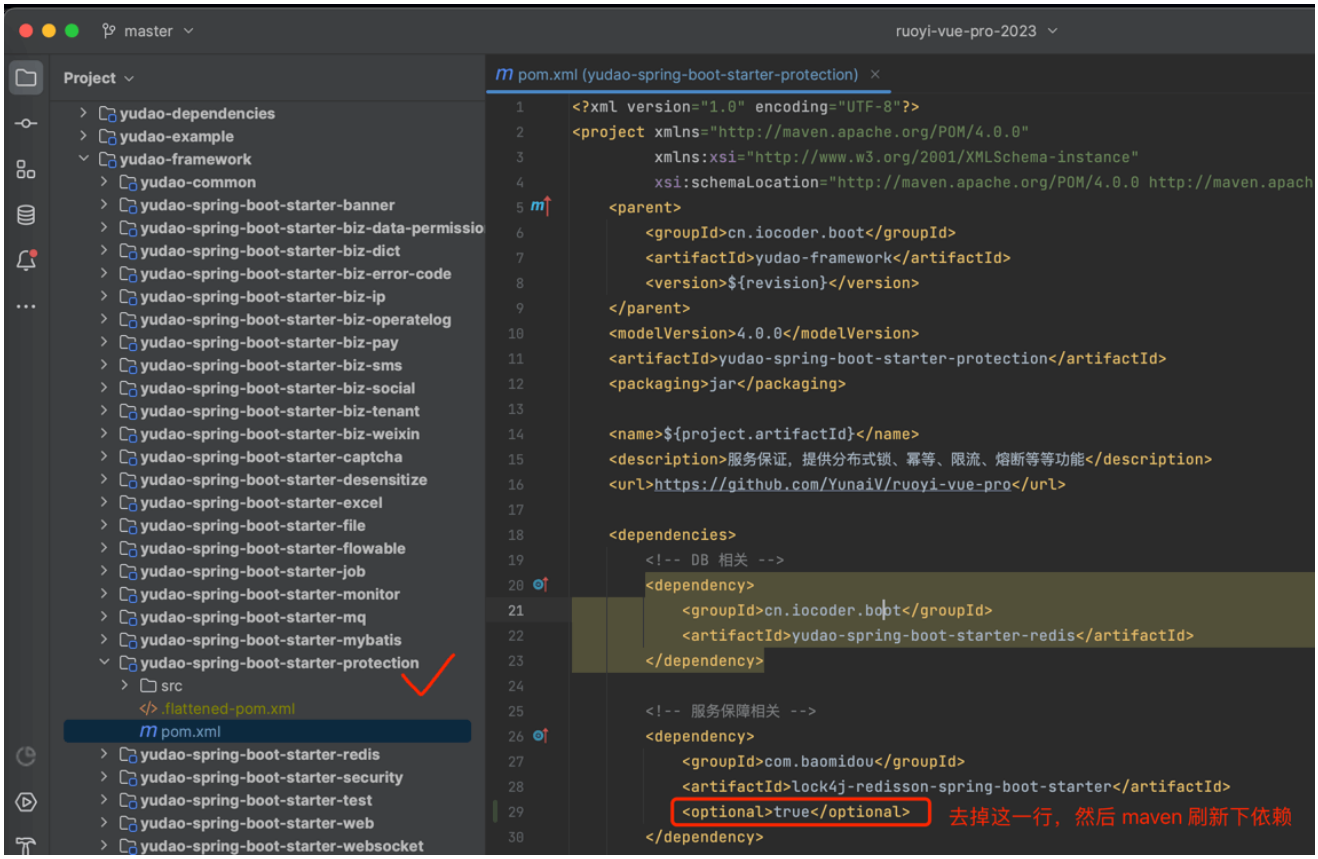
```java
    /** {@link #NOTIFY_TIMEOUT} 的毫秒 */
    public static final long NOTIFY_TIMEOUT_MILLIS = 120 * DateUtils.SECOND_MILLIS;

    @Resource
    private PayNotifyLockRedisDAO payNotifyLockCoreRedisDAO; // ① 注入 PayNotifyLockRedisDAO

    /**
     * 同步执行单个支付通知
     *
     * @param task 通知任务
     */
    public void executeNotifySync(PayNotifyTaskDO task) {
        // 分布式锁，避免并发问题
        payNotifyLockCoreRedisDAO.lock(task.getId(), NOTIFY_TIMEOUT_MILLIS, () -> { // ② 加 Redis 分布式锁
            // 校验，当前任务是否已经被通知过
            // 虽然已经通过分布式加锁，但是可能同时满足通知的条件，然后都去获得锁。此时，第一个执行完后，第二个还是能拿到锁，然后会再执行一次。
            PayNotifyTaskDO dbTask = payNotifyTaskCoreMapper.selectById(task.getId());
            if (DateUtils.afterNow(dbTask.getNextNotifyTime())) {
                log.info("[executeNotify][dbTask({}) 任务被忽略，原因是未到达下次通知时间，可能是因为并发执行了]", JsonUtils.toJsonString(dbTask));
                return;
            }

            // 执行通知
            executeNotify(dbTask); // ④ 真正执行业务逻辑
        });
    }
```

③ 获得到锁后，需要判断下，是否前一个获得锁，已经执行完成了。

> **技术选型：为什么不使用 Lock4j 提供的 LockTemplate 实现编程式锁？**
>
> 两者各有优势，选择 Redisson 主要考虑它支持的 Redis 分布式锁的类型较多：可靠性较高的红锁、性能较好的读写锁等等。
>
> Lock4j 的 LockTemplate 也是不错的选择，一方面不强依赖 Redisson 框架，一方面支持 ZooKeeper 等等。

2. 声明式锁

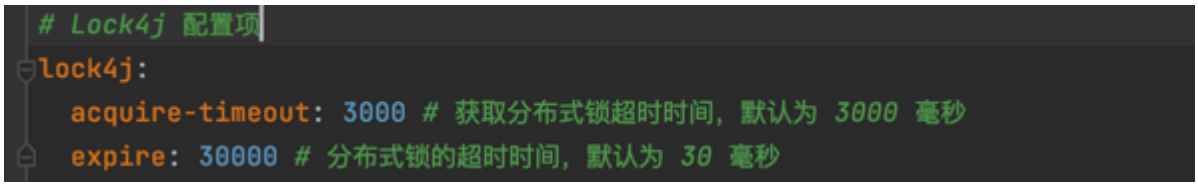
考虑到不是所有人都会使用 Lock4j 组件，所以默认项目未引入 `lock4j-redisson-spring-boot-starter` 。如果你想要实用，可按照下图进行引入：



2.1 Lock4j 配置

> **友情提示：以 yudao-module-system 服务为例子。**

在 `application-local.yaml` 配置文件中，通过 `lock4j` 配置项，添加 Lock4j 全局默认的分布式锁配置。如下图所示：



```yaml
# Lock4j 配置项
lock4j:
  acquire-timeout: 3000 # 获取分布式锁超时时间，默认为 3000 毫秒
  expire: 30000 # 分布式锁的超时时间，默认为 30 毫秒
```

2.2 使用案例

在需要使用到分布式锁的方法上，添加 `@Lock4j` 注解，非常方便。示例代码如下：

```
@Service
public class DemoService {
```

```
    // 默认使用 lock4j 配置项
    @Lock4j
    public void simple() {
        //do something
    }

    // 完全配置，支持 Spring EL 表达式
    @Lock4j(keys = {"#user.id", "#user.name"}, expire = 60000, acquireTimeout =
    public User customMethod(User user) {
        return user;
    }

}
```

← 单元测试 幂等性（防重复提交） →