

od0609

[异步消息](#)

[Spring事件监听机制](#)

[@Scheduled源码解析](#)

[SpringBoot定时任务](#)

[浅谈Redis分布式锁\(下\)](#)

[浅谈Redis分布式锁\(中\)](#)

[浅谈Redis分布式锁\(上\)](#)

[华为od机试 2025B卷 - 乘坐保密电梯 \(C++ & Python & JAVA & JS\)](#)

[华为OD机试 2025 B卷 - 战场索敌 \(C++ & Python & JAVA & JS &](#)

[华为OD 机试 2025 B卷 - 区间连接器 \(C++ & Python & JAVA & JS\)](#)

[华为od机试 2025 B卷 - 基站维修工程师 \(C++ & Python & JAVA & JS\)](#)

[华为OD机试 2025 B卷 - We are a Team \(C++ & Python & JAV](#)

异步消息

在日常开发中，偶尔需要在主营业务逻辑之外做一些附加操作，比如下单成功后通知商家、课程报名成功后通知老师、简历投递成功后通知HR。一般来讲，这些业务是不适合放在主线程中的：

```
Java | ▾

1  @Slf4j
2  @SpringBootTest
3  public class AsyncNotifyTest {
4
5      @Test
6      public void testAsyncNotify() throws InterruptedException {
7
8          long start = System.currentTimeMillis();
9
10         // 投递简历，插入投递记录
11         TimeUnit.SECONDS.sleep(2);
12         log.info("插入投递记录完毕...");
13
14         // 发送短信通知HR，并留存发送记录
15         notifyHR("bravo1988", "叉车师傅");
16         writeMsg("bravo1988", "叉车师傅");
17
18         log.info("耗时:{}毫秒", System.currentTimeMillis() - start);
19     }
20
21     public void notifyHR(String username, String jobName) throws InterruptedException {
22         TimeUnit.SECONDS.sleep(1);
23         log.info("【发送消息】HR你好，用户：{}, 投递你的岗位：{}", username, jobName);
24     }
25
26     public void writeMsg(String username, String jobName) {
27         // 留存消息发送记录
28         log.info("【保存消息】保存到数据库，用户：{}, 岗位：{}", username, jobName);
29     }
30
31 }
```

结果

```
[main] INFO - 插入投递记录完毕...
[main] INFO - 【发送消息】 HR你好， 用户:bravo1988, 投递你的岗位:叉车师傅
[main] INFO - 【保存消息】 保存到数据库， 用户:bravo1988, 岗位:叉车师傅
[main] INFO - com.bravo.happy.AsyncNotifyTest - 耗时:3019毫秒
```

消息通知等附加操作为什么不适合放在主流程呢？

- 首先，消息通知相对没那么重要，即使发送失败了，一般还有发送记录，重新发送或者只要能追溯即可
- 其次，在主流程中加入消息通知会减慢响应速度
- 最后，万一消息发送失败，还可能导致事务回滚，但系统本身其实是没有问题的

多线程异步消息

一个解决办法是使用多线程，把消息发送的逻辑单独放在一个异步线程中执行，主流程处理完毕直接返回即可。为了尽可能简单，这里就不配置线程池或使用@Async了，换CompletableFuture做演示：

```
1  @Slf4j
2  @SpringBootTest
3  public class AsyncNotifyTest {
4
5      @Test
6      public void testAsyncNotify() throws InterruptedException {
7
8          long start = System.currentTimeMillis();
9
10         // 投递简历，插入投递记录
11         TimeUnit.SECONDS.sleep(2);
12         log.info("插入投递记录完毕...");
13
14         // 异步发送短信通知HR，并留存发送记录
15         CompletableFuture.runAsync(() -> {
16             try {
17                 notifyHR("bravo1988", "叉车师傅");
18                 writeMsg("bravo1988", "叉车师傅");
19             } catch (InterruptedException e) {
20                 e.printStackTrace();
21             }
22         });
23
24         log.info("耗时:{}毫秒", System.currentTimeMillis() - start);
25
26         // 为了观察到异步线程里的打印信息，主线程sleep一会儿
27         TimeUnit.SECONDS.sleep(2);
28     }
29
30     public void notifyHR(String username, String jobName) throws InterruptedException {
31         TimeUnit.SECONDS.sleep(1);
32         log.info("【发送消息】HR你好，用户：{}, 投递你的岗位：{}", username, jobName);
33     }
34
35     public void writeMsg(String username, String jobName) {
36         // 留存消息发送记录
37         log.info("【保存消息】保存到数据库，用户：{}, 岗位：{}", username, jobName);
38     }
39
40 }
```

结果

[main] INFO - 插入投递记录完毕...

[main] INFO - 耗时:2145毫秒

[ForkJoinPool.commonPool-worker-9] INFO - 【发送消息】 HR你好， 用户:bravo1988, 投递你的岗位:叉车师傅

[ForkJoinPool.commonPool-worker-9] INFO - 【保存消息】 保存到数据库, 用户:bravo1988, 岗位:叉车师傅

可以看到，主线程耗时变成了2秒，如此一来用户整个投递简历的响应时间缩短了。

Spring事件监听机制

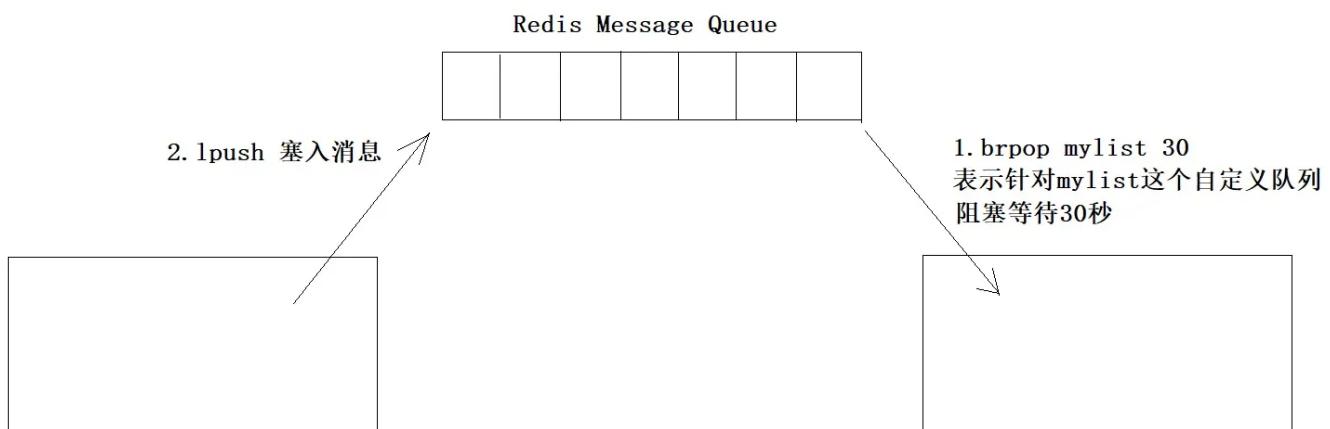
具体请参考[Spring事件监听机制](#)，本质上和多线程异步消息是一样的。

Redis实现消息队列

上面多线程版本的异步消息其实已经挺不错了，但小概率的情况下可能会出现消息丢失（虽然当前情境下无所谓）：

- 情况1：消息过多，线程数不够触发拒绝策略
- 情况2：异步线程宕机了，消息丢失（类似于消费者挂了）

此时可以考虑使用Redis做一个简单的消息队列，数据类型可以选择List。



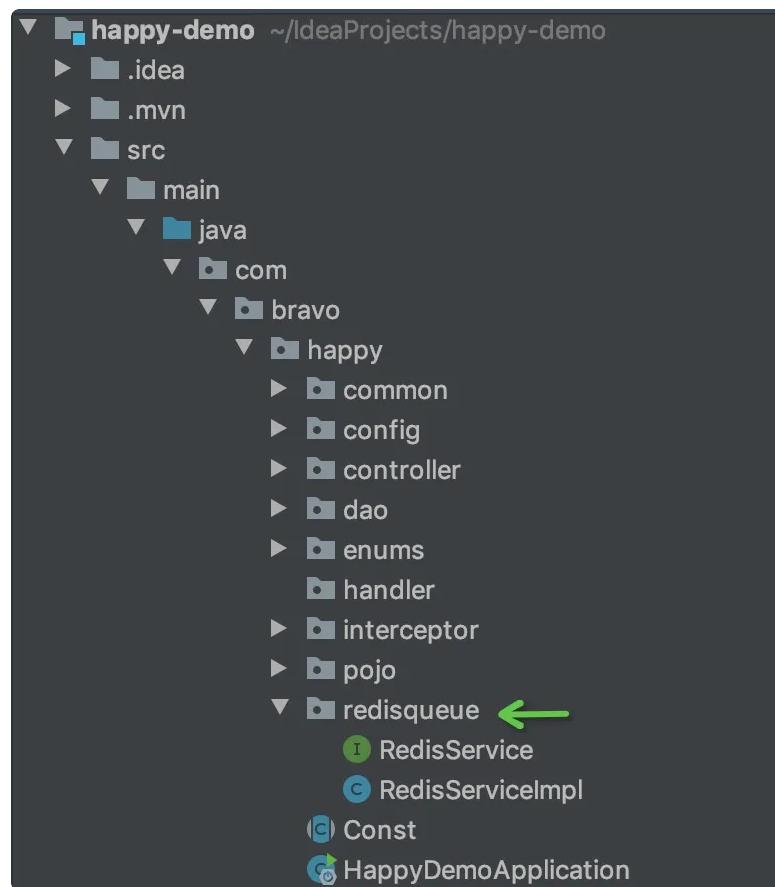


对于Redis的List，如果使用lpush+rpop即可实现先进先出的简单队列，而如果配合brpop则可实现阻塞队列。所谓的brpop，其实就是blocking right pop，即阻塞等待队列中的消息，一旦有消息被push进队列就从右边取出消费。

注意，rpop和brpop的区别是，rpop发现队列为空直接返回null，不会等待，也不能设置等待时间：



lpush和brpop反映到Java代码里，可以使用RedisTemplate或StringRedisTemplate实现。



没有安装redis的同学，可以用我阿里云的（性能很垃圾，请尽量不要分享）：

YAML |

```
1 spring:  
2   redis:  
3     host: # 参考小册《阿里云账号》  
4     password: # 参考小册《阿里云账号》  
5     database: 1
```

Java |

```
1 public interface RedisService {  
2  
3   /**  
4    * 向队列插入消息  
5    *  
6    * @param queue 自定义队列名称  
7    * @param obj    要存入的消息  
8    */  
9   void pushQueue(String queue, Object obj);  
10  
11  /**  
12   * 从队列取出消息  
13   *  
14   * @param queue    自定义队列名称  
15   * @param timeout  最长阻塞等待时间  
16   * @param timeUnit 时间单位  
17   * @return  
18   */  
19   Object popQueue(String queue, long timeout, TimeUnit timeUnit);  
20 }
```

```
1  @Component
2  public class RedisServiceImpl implements RedisService {
3
4      @Autowired
5      private StringRedisTemplate redisTemplate;
6      @Autowired
7      private ObjectMapper objectMapper;
8
9      @Override
10     public void pushQueue(String queue, Object obj) {
11         try {
12             redisTemplate.opsForList().leftPush(queue, objectMapper.writeValueAsString(obj));
13         } catch (JsonProcessingException e) {
14             e.printStackTrace();
15         }
16     }
17
18     @Override
19     public Object popQueue(String queue, long timeout, TimeUnit timeUnit)
20     {
21         return redisTemplate.opsForList().rightPop(queue, timeout, timeUnit);
22     }

```

```
1  Slf4j
2  @RunWith(SpringRunner.class)
3  @SpringBootTest
4  public class RedisQueueTest {
5
6      @Autowired
7      private RedisService redisService;
8
9      public static final String ORDER_MESSAGE = "order_message";
10
11     @Test
12     public void testRedisBlockingQueue() throws InterruptedException {
13         // 订单服务
14         orderService("bravo1988", 10086L);
15
16         // 启动消费者，取出消息，逐一发送
17         new Thread(this::consumeMsg).start();
18
19         // 10秒后再发一条消息，模拟第二次下单
20         TimeUnit.SECONDS.sleep(10);
21         orderService("bravo2020", 99999L);
22
23         // 等待一会儿，观察第二条消息
24         TimeUnit.SECONDS.sleep(10);
25     }
26
27     public void orderService(String username, Long orderId) {
28         // 1.操作数据库，插入订单
29
30         // 2.其他操作
31
32         // 3.发送消息
33         redisService.pushQueue(ORDER_MESSAGE, new Order(username, orderId));
34     }
35
36     public void consumeMsg() {
37         for (; ; ) {
38             Object order = redisService.popQueue(ORDER_MESSAGE, 5, TimeUnit.SECONDS);
39             log.info("每隔5秒循环获取，期间for循环阻塞");
40             if (order != null) {
41                 log.info("order:{}", order.toString());
42             }
43         }
44     }
45 }
```

```
44     }
45
46     @Data
47     @AllArgsConstructor
48     static class Order {
49         private String username;
50         private Long resumeId;
51     }
52 }
53 }
```



Redis实现消息队列的好处是，可以把消息存起来慢慢消费，而且项目挂了不影响已经存入的消息，重新启动后仍可继续消费：



可能有同学不禁要问：如果消息还没发送到队列中就丢失了呢？发送方也无法感知（没有应答机制），所以Redis作为消息队列还是存在很多问题的。

那为什么要在这章节安排Redis实现消息队列呢？

首先，就我个人的感受而言，入行后很长一段时间我都对Redis很抵触、很畏惧，导致自己一直停滞不前，其实Redis并没有我们想的那么难，只要你敢动手去敲，就会迅速熟悉起来（其他技术也是如此）。

其次，实际开发中一些小项目还是会用，主要是从系统复杂性考虑，不轻易引入MQ，所以仍有学习的必要。尤其是对于一些消息通知，丢了就丢了，影响不是特别大，而订单操作就不适合用Redis这么简陋的消息队列了。

上面的demo仅仅用作学习，大家有兴趣可以自行拓展，Redis还提供了消息的发布/订阅模式：

用过MQ的同学会觉得上面的模式很熟悉！

小坑

我在Redis实现消息通知的代码中，留了一个小坑，是大家平时可能会不小心犯的。

提示：和SpringBoot定时任务故事中我同事遇到的类似的坑。

详见左侧目录答案。

在执行上面的代码时可能遇到：

```
nested exception is io.lettuce.core.RedisException: io.lettuce.core.RedisException: Connection closed
```

这是因为主线程结束导致Redis断开，而两个for循环还在操作队列。实际生产环境一般不会有这个问题，因为线程是一直跑着的。

| 来自：异步消息

Spring事件监听机制

发短信案例问题分析

假设现在有这么一个业务场景：

用户在京西商城下单成功后，平台要发送短信通知用户下单成功

我们最直观的想法是直接在order()方法中添加发送短信的业务代码：

```
Java

1 public void order(){
2     // 下单操作
3     log.info("下单成功, 订单号:{}" , orderNumber);
4     // 发送短信
5     sendSms();
6 }
7
8 public void sendSms(){
9     // 省略...
10 }
```

乍一看没什么不妥，但随着时间推移，上面的代码就会暴露出局限性：

一个月后，京西搞了自建物流体系，用户下单成功后，还需要通知物流系统发货

于是你又要打开OrderService修改order()方法：

```
Java

1 public void order(){
2     // 下单成功
3     log.info("下单成功, 订单号:{}" , orderNumber);
4     // 发送短信
5     sendSms();
6     // 通知车队发货
7     notifyCar();
8 }
```

嗯，完美。

又过了一个月，东哥被抓了，股价暴跌，决定卖掉自己的车队，所以下单后就不用通知车队了
重新修改OrderService：

```
1 ▼ public void order(){
2     // 下单成功
3     log.info("下单成功, 订单号:{}" , orderNumber);
4     // 发送短信
5     sendSms();
6     // 车队没了, 注释掉这行代码
7     // notifyCar();
8 }
```

又过了一个月，东哥明尼苏达州荣耀归来：回来做我的兄弟一起开车吧。

好的，东哥。

```
1 ▼ public void order(){
2     // 下单成功
3     log.info("下单成功, 订单号:{}" , orderNumber);
4     // 发送短信
5     sendSms();
6     // 车队买回来了, 放开这段代码
7     notifyCar()
8 }
```

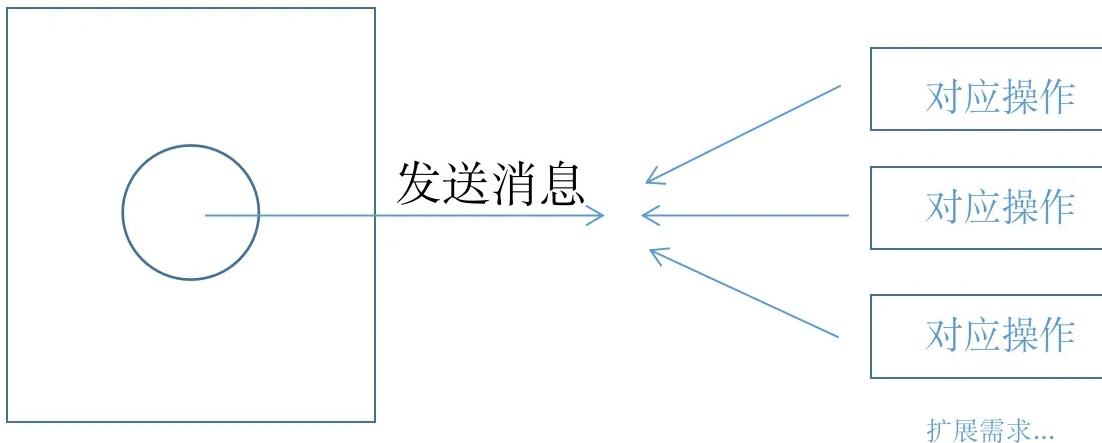
车队回来了，你却受不了这大起大落异常刺激的生活，决定离职。

就在这时候，组长拉住了你，语重心长地和你说：

小伙子，知道什么叫“以增量的方式应对变化的需求”吗？听过Spring事件监听机制吗？

说时迟那时快，组长拿来一支笔和一张纸，唰唰唰画了一张图：

某个服务

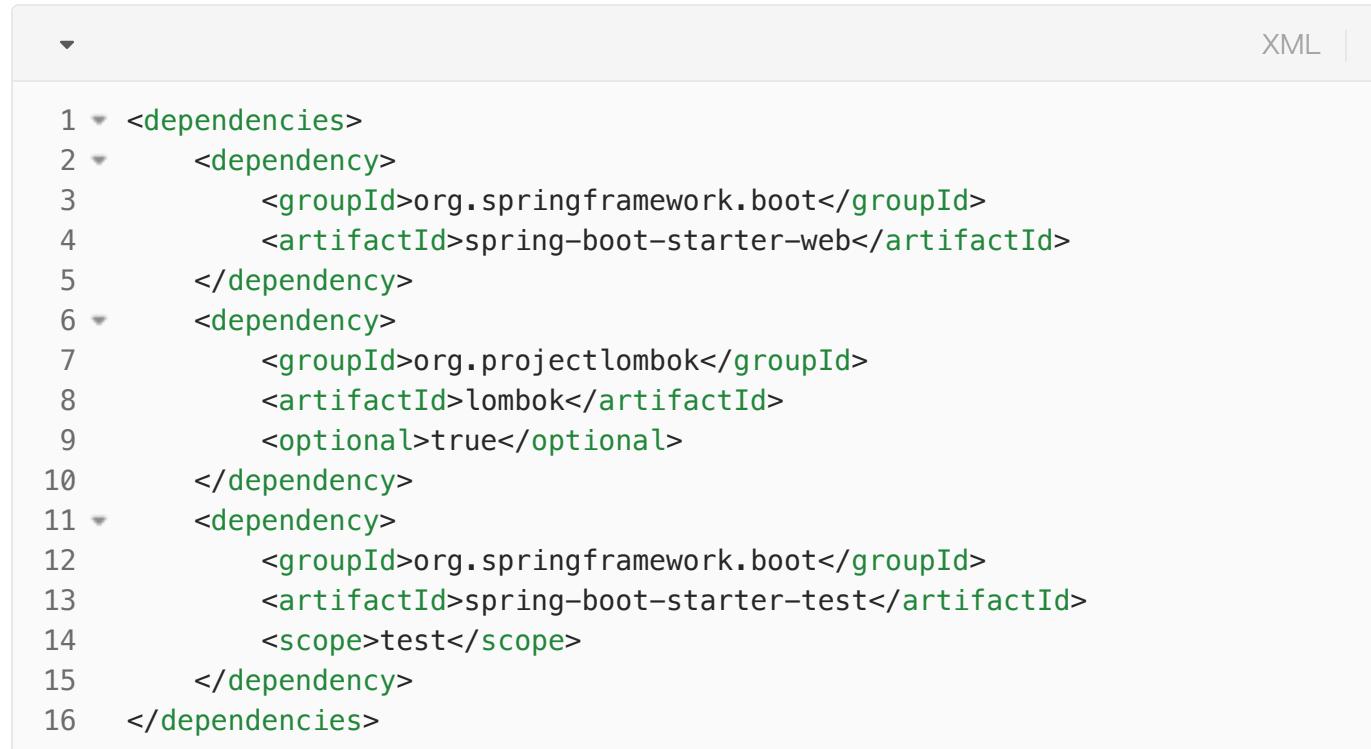
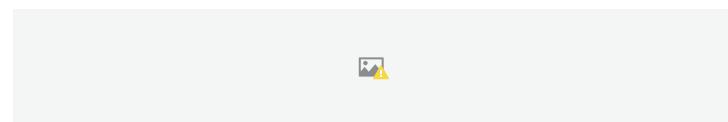


你顿时心领神会，扑通一声跪在了地上，开始敲起了代码。

利用Spring事件机制完成需求

环境准备

还是用之前的SpringBoot项目即可，只要你引入了spring-boot-starter-web。



```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-web</artifactId>
5   </dependency>
6   <dependency>
7     <groupId>org.projectlombok</groupId>
8     <artifactId>lombok</artifactId>
9     <optional>true</optional>
10  </dependency>
11  <dependency>
12    <groupId>org.springframework.boot</groupId>
13    <artifactId>spring-boot-starter-test</artifactId>
14    <scope>test</scope>
15  </dependency>
16 </dependencies>
```

代码

OrderService

```
Java |  
1  /**  
2   * 订单服务  
3   */  
4   @Service  
5   public class OrderService {  
6  
7       @Autowired  
8       private ApplicationContext applicationContext;  
9  
10      public void order() {  
11          // 下单成功  
12          System.out.println("下单成功...");  
13          // 发布通知 (传入了当前对象)  
14          applicationContext.publishEvent(new OrderSuccessEvent(this));  
15          System.out.println("main线程结束...");  
16      }  
17  }
```

OrderSuccessEvent (继承ApplicationEvent, 自定义事件)

```
Java |  
1  public class OrderSuccessEvent extends ApplicationEvent {  
2  
3      /**  
4       * Create a new ApplicationEvent.  
5       *  
6       * @param source the object on which the event initially occurred (never {@code null})  
7       */  
8      public OrderSuccessEvent(Object source) {  
9      super(source);  
10     }  
11  }
```

SmsService (实现ApplicationListener, 监听OrderSuccessEvent)

Java

```
1  /**
2   * 短信服务, 监听OrderSuccessEvent
3   */
4  @Service
5  public class SmsService implements ApplicationListener<OrderSuccessEvent>
{
6
7      @Override
8      public void onApplicationEvent(OrderSuccessEvent event) {
9          this.sendSms();
10     }
11
12     /**
13     * 发送短信
14     */
15     public void sendSms() {
16         System.out.println("发送短信...");
17     }
18 }
```

Test

Java

```
1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class Test {
4
5      @Autowired
6      private OrderService orderService;
7
8      @Test
9      public void testSpringEvent() {
10         orderService.order();
11     }
12 }
```

输出:

下单成功...

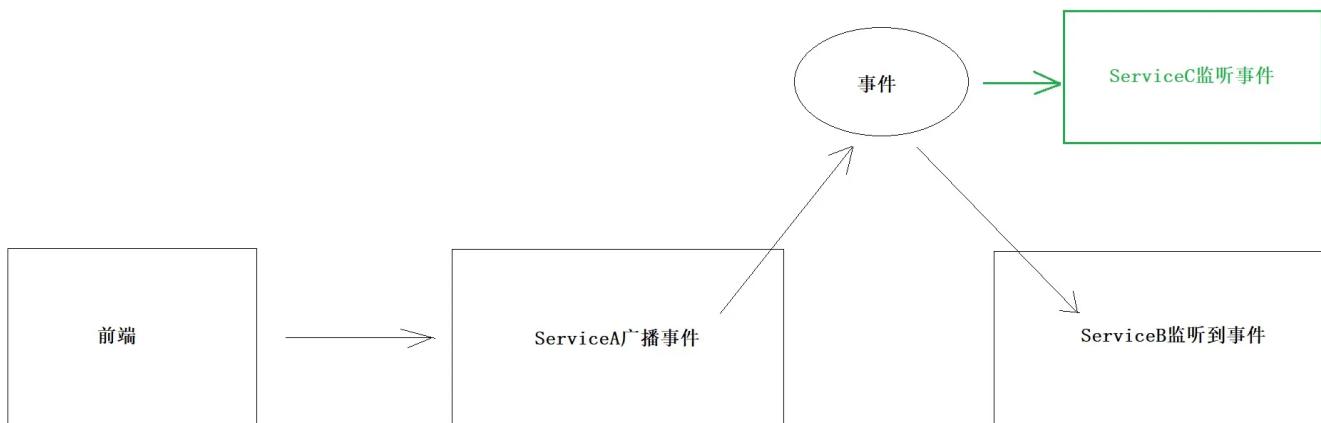
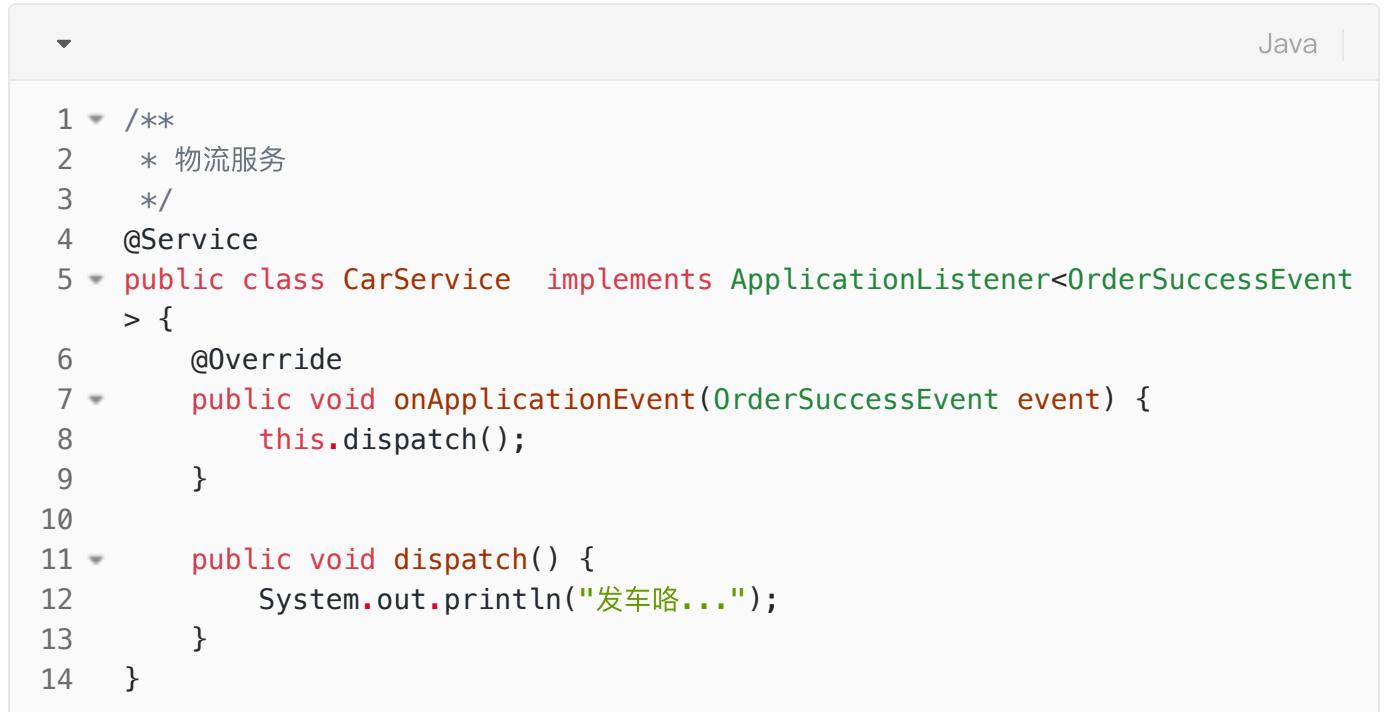
发送短信...

main线程结束...

流程示意图（为什么 SmsService 能监听到？更详细的流程会在山寨版 Spring 事件监听机制中介绍）：



如果后期针对下单成功有新的操作，可以新写一个事件监听类：



这就是以增量的方式应对变化的需求，而不是去修改已有的代码（ServiceA）。同样的，对老项目改造时也是如此，如果你不知道原来的接口是干嘛的，最好不要去动它，宁愿新写一个接口，即一般提倡“对扩展开放，对修改关闭”。

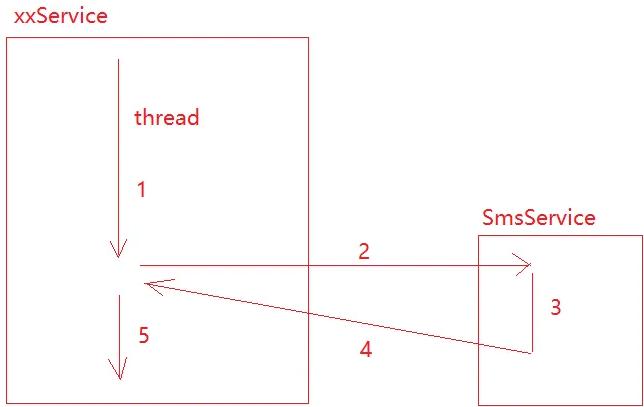
上面SmsService既是一个服务，还是一个Listener，因为它既有@Service又实现了ApplicationListener接口。

但是仅仅是为了一个监听回调方法而实现一个接口，未免麻烦，所以Spring提供了注解的方式：

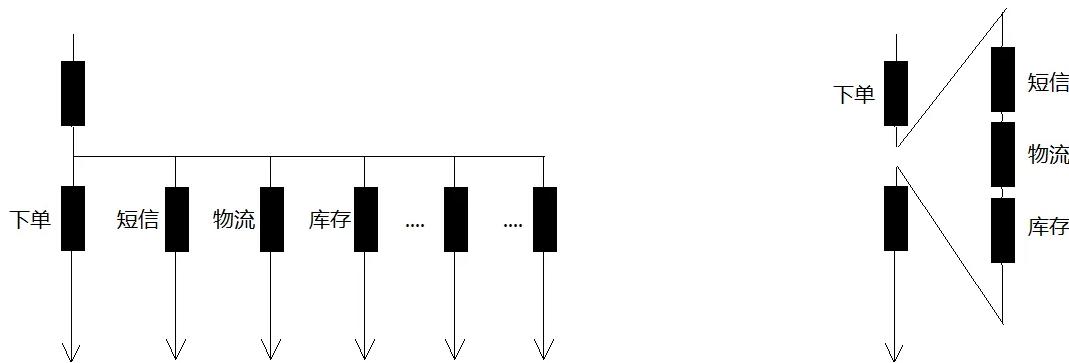
```
Java |  
1  /**  
2   * 短信服务，监听OrderSuccessEvent，但不用实现ApplicationListener  
3   */  
4   @Service  
5  public class SmsService {  
6  
7      /**  
8   * 发送短信 @EventListener指定监听的事件  
9   */  
10  @EventListener(OrderSuccessEvent.class)  
11  public void sendSms() {  
12  
13      try {  
14      Thread.sleep(1000L * 5);  
15      } catch (InterruptedException e) {  
16      e.printStackTrace();  
17  }  
18  
19  System.out.println("发送短信...");  
20}  
21  
22 }
```

Spring发布异步事件

看似很完美了，但是你注意到Spring默认的事件机制是同步的：



如果针对OrderService下单成功后的操作越来越多，比如下单后需要完成的对应操作有十几个，那么等十几个其他服务都调用完毕，东哥可能又去明尼苏达州了。



所以，你必须想办法把Spring的事件机制改成异步的，尽可能快地返回下单的结果本身，而不是等其他附属服务全部完成（涉及到其他问题暂时按下不表）。

要想把Spring事件机制改造成异步通知，最粗暴的方法是：

OrderService

```
1  /**
2   * 订单服务
3   */
4   @Service
5  public class OrderService {
6
7     @Autowired
8     private ApplicationContext applicationContext;
9
10    public void order() {
11        // 下单成功
12        System.out.println("下单成功...");
13        // 发布通知
14        new Thread(() ->{
15            applicationContext.publishEvent(new OrderSuccessEvent(this));
16        }).start();
17        System.out.println("main线程结束...");
18        // 等SmsService结束
19        try {
20            Thread.sleep(1000L * 5);
21        } catch (InterruptedException e) {
22            e.printStackTrace();
23        }
24    }
25 }
```

SmsService

```
1  /**
2   * 短信服务，监听OrderSuccessEvent
3   */
4   @Service
5  public class SmsService implements ApplicationListener<OrderSuccessEvent>
{
6
7      @Override
8      public void onApplicationEvent(OrderSuccessEvent event) {
9          this.sendSms();
10     }
11
12     /**
13     * 发送短信
14     */
15    public void sendSms() {
16        try {
17            Thread.sleep(1000L * 3);
18        } catch (InterruptedException e) {
19            e.printStackTrace();
20        }
21        System.out.println("发送短信...");
22    }
23 }
```

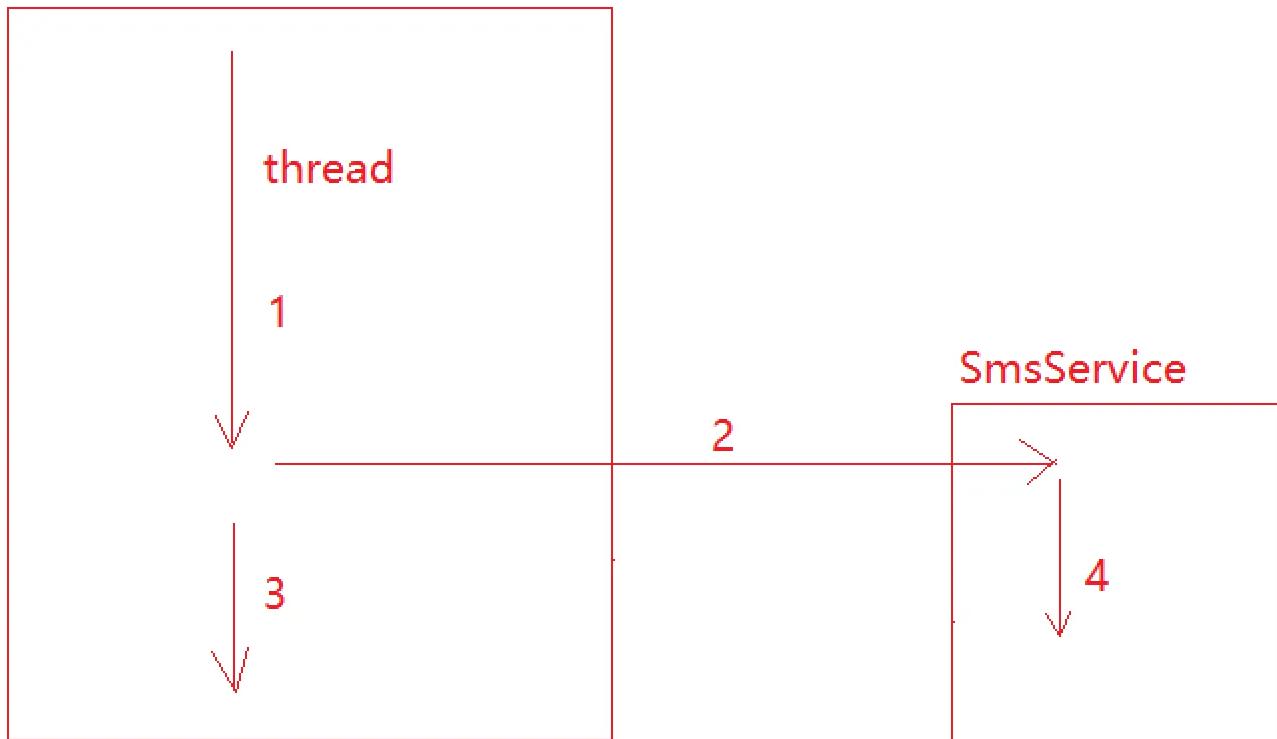
输出：

下单成功...

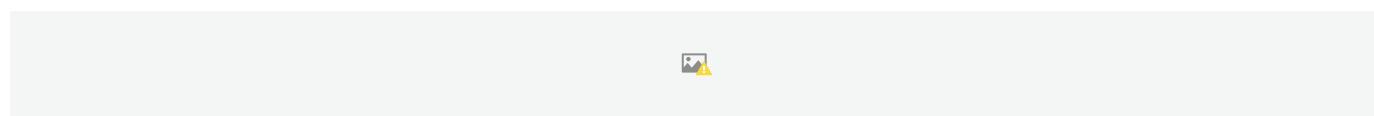
main线程结束...

发送短信...

xxService



当然，这种做法其实违背了Spring事件机制的设计初衷。人家会想不到你要搞异步通知？



当SimpleApplicationEventMulticaster中的Executor不为null，就会执行异步通知。

```

1  @Configuration
2  public class AsyncEventConfig {
3
4      @Bean(name = "applicationEventMulticaster")
5      public ApplicationEventMulticaster simpleApplicationEventMulticaster()
6      {
7          SimpleApplicationEventMulticaster eventMulticaster
8              = new SimpleApplicationEventMulticaster();
9
10         eventMulticaster.setTaskExecutor(new SimpleAsyncTaskExecutor());
11         return eventMulticaster;
12     }
13 }

```

把OrderService改回来：

```

1  @Service
2  public class OrderService {
3
4      @Autowired
5      private ApplicationContext applicationContext;
6
7      public void order() {
8          // 下单成功
9          System.out.println("下单成功...");
10         // 发布通知
11         applicationContext.publishEvent(new OrderSuccessEvent(this));
12         System.out.println("main线程结束...");
13         // 等SmsService结束
14         try {
15             Thread.sleep(1000L * 5);
16         } catch (InterruptedException e) {
17             e.printStackTrace();
18         }
19     }
20 }

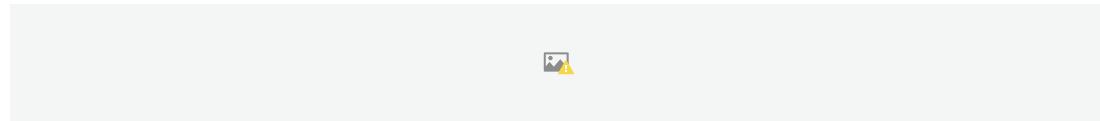
```

此时仍然是异步的。

最后说一句，Spring事件机制适合单体应用，同一个JVM且并发不大的情况，如果是分布式应用，推荐使用MQ。Spring事件监听机制和MQ有相似的地方，也有不同的地方。MQ允许跨JVM，因为它本身是独立于项目之外的，切提供了消息持久化的特性，而Spring事件机制哪怕使用了异步，本质是还是一种方法调用，宕机了就没了。

细节

之前在知乎，有知友留言，为什么OrderService发布事件要放入this，感觉没什么用：



大家有注意到这个细节吗~你们觉得有什么用呢？

拓展阅读：[spring 事件监听同时支持同步事件及异步事件](#)

| 来自：[Spring事件监听机制](#)

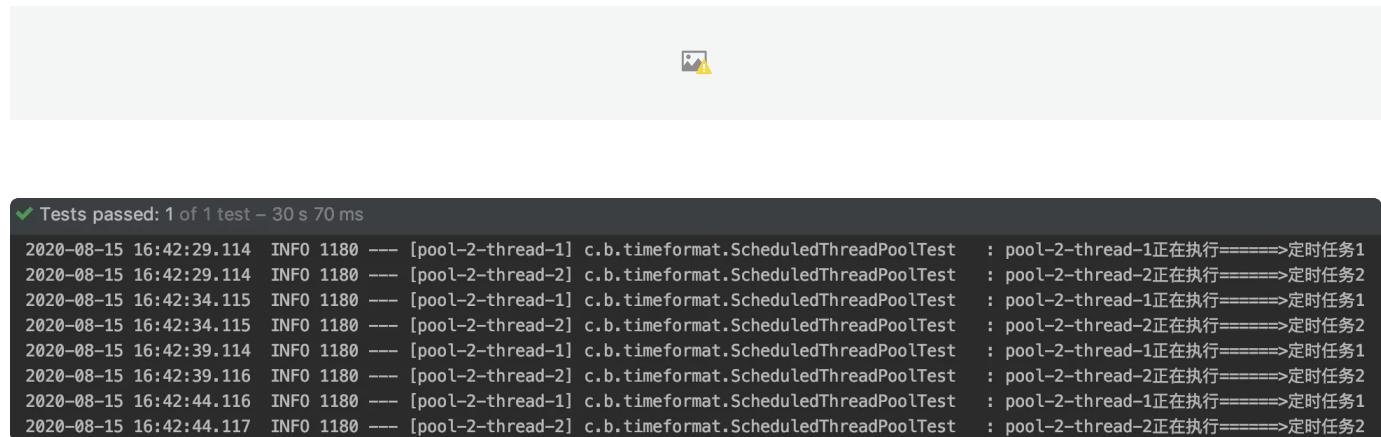
@Scheduled源码解析

首先，丝毫不用怀疑，定时任务绝对不是Spring首创的，JDK本身就提供了很多种实现定时任务的方式。

来看一种最简单的实现：

```
public static void main(String[] args) {
    Timer timer = new Timer();
    timer.schedule(new TimerTask() {
        @Override
        public void run() {
            System.out.println("====任务执行====");
        }
    }, delay: 1000L, period: 2000L);
}
```

实际上JDK还提供了定时任务线程池ScheduledThreadPool，我们可以直接通过Executors工具类获取：



```
Tests passed: 1 of 1 test - 30 s 70 ms
2020-08-15 16:42:29.114 INFO 1180 --- [pool-2-thread-1] c.b.timeformat.ScheduledThreadPoolTest : pool-2-thread-1正在执行=====>定时任务1
2020-08-15 16:42:29.114 INFO 1180 --- [pool-2-thread-2] c.b.timeformat.ScheduledThreadPoolTest : pool-2-thread-2正在执行=====>定时任务2
2020-08-15 16:42:34.115 INFO 1180 --- [pool-2-thread-1] c.b.timeformat.ScheduledThreadPoolTest : pool-2-thread-1正在执行=====>定时任务1
2020-08-15 16:42:34.115 INFO 1180 --- [pool-2-thread-2] c.b.timeformat.ScheduledThreadPoolTest : pool-2-thread-2正在执行=====>定时任务2
2020-08-15 16:42:39.114 INFO 1180 --- [pool-2-thread-1] c.b.timeformat.ScheduledThreadPoolTest : pool-2-thread-1正在执行=====>定时任务1
2020-08-15 16:42:39.116 INFO 1180 --- [pool-2-thread-2] c.b.timeformat.ScheduledThreadPoolTest : pool-2-thread-2正在执行=====>定时任务2
2020-08-15 16:42:44.116 INFO 1180 --- [pool-2-thread-1] c.b.timeformat.ScheduledThreadPoolTest : pool-2-thread-1正在执行=====>定时任务1
2020-08-15 16:42:44.117 INFO 1180 --- [pool-2-thread-2] c.b.timeformat.ScheduledThreadPoolTest : pool-2-thread-2正在执行=====>定时任务2
```

和一般的线程池不一样的是，ScheduledThreadPool会不断地、定时地执行提交的任务。

定时任务方法的命名方式都大同小异，都叫scheduleXxx()，后面还会在别处见到这种命名方式，注意一下。

那Spring是如何实现@Scheduled的呢？

还是从@Scheduled注解入手：



我们发现@Scheduled注解是由ScheduledAnnotationBeanPostProcessor这个后置处理器处理的：



框出来的内容意思是：

当前这个PostProcessor负责注册那些加了@Scheduled注解的方法，然后根据@Scheduled注解的fixedRate、fixedDelay等属性分别交给TaskScheduler（定时任务线程池）执行。

现在有两个问题：

- 这个后置处理器怎么被加入Spring容器的？
- @Scheduled标注的方法是如何被TaskScheduler定时任务线程池处理的？

@EnableScheduling的作用

先解决第一个问题。

我们从类注释可以看到，ScheduledAnnotationBeanPostProcessor可以通过XML形式的<task:annotation-driven>标签或者注解形式的@EnableScheduling导入到Spring容器。我们主要观察@EnableScheduling：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Import(SchedulingConfiguration.class) ←
@Documented
public @interface EnableScheduling {

}
```

发现@EnableScheduling导入了一个配置类：

```
@Configuration
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
public class SchedulingConfiguration {

    @Bean(name = TaskManagementConfigUtils.SCHEDULED_ANNOTATION_PROCESSOR_BEAN_NAME)
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public ScheduledAnnotationBeanPostProcessor scheduledAnnotationProcessor() {
        return new ScheduledAnnotationBeanPostProcessor();
    }

}
```

哦，原来@EnableScheduling经过一连串的套娃操作后最终会通过@Bean的方式把ScheduledAnnotationBeanPostProcessor加入到Spring容器。

@Scheduled方法的执行流程

接下来解决第二个问题：@Scheduled标注的方法是如何被定时执行的？

分两步：

- 收集@Scheduled方法
- 执行定时任务

还是以上一篇的定时任务为例，我们通过debug的方式过一遍流程好了。

```
@Slf4j
@Component
@EnableScheduling
public class TaskOne {

    @Async("taskExecutor")
    @Scheduled(cron = "*/10 * * * * ?")
    public void test1() {
        log.info("=====test1任务启动=====");
        try {
            Thread.sleep( millis: 2 * 1000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        log.info("=====test1任务结束=====");
    }
}
```

收集@Scheduled方法

每个Bean（包括TaskOne）会经过@EnableScheduling导入的
ScheduledAnnotationBeanPostProcessor：



forEach会调用本类的另一个processScheduled方法，它会根据每个定时任务@Scheduled注解中的fixedRate、fixedDelay以及cron属性，**分别**将每个任务存储到ScheduledTaskRegistrar的不同TaskList。



跟进ScheduledTaskRegistrar#scheduleCronTask()看看：



好了，至此所有@Scheduled方法都已经被包装成Task存储到不同的任务列表中了。那么，这些任务怎么被定时调用的呢？

执行定时任务

在之前使用Java的ScheduledThreadPool执行定时任务时，都存在一个提交任务的动作(scheduleXxx)：

而Spring的@Scheduled底层也是用线程池的。我们来找找。

由于ScheduledAnnotationBeanPostProcessor实现了ApplicationListener的onApplicationEvent()，所以在整个Spring容器启动完毕后最终会调用finishRegistration()，在所有定时任务注册完毕后做一些事。

你会发现，finishRegistration()内部会调用registrar.afterPropertiesSet()。

嗯？registrar？就是收集了各种定时任务的那个ScheduledTaskRegistrar吗？

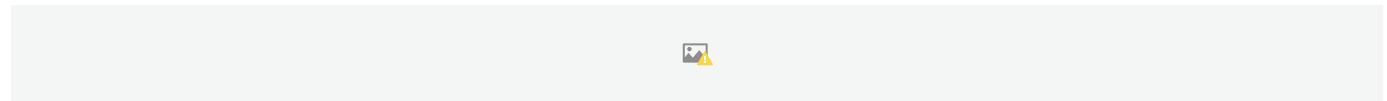
没错，你猜到了，接下来就是把ScheduledTaskRegistrar的任务逐个提交给taskScheduler线程池即可。afterPropertiesSet()调用了scheduleTasks()，这个方法会把每个任务提交给线程池执行。



Executors.newSingleThreadScheduledExecutor(), 这解释了为什么SpringBoot默认的定时任务是单线程的。

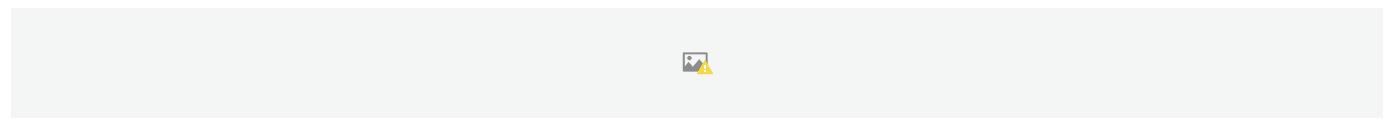


addScheduledTask()内部调用了scheduleCronTask(task), 把任务提交给线程池：



定时任务线程池不是这么配的！

其实，在上一篇文章中，我自己也犯了一个错误。在注意到SpringBoot默认的定时任务是单线程后，想都不想直接配置了一个线程池：



还在定时任务上加了@Async：



实际上这并不是一个很好的选择。

如果我们在ScheduledTaskRegistrar#scheduleTasks()上打断点观察，会发现其实我们配置的线程池并没有赋值给taskExecutor：



此时定时任务线程池仍然是默认的SingleThreadScheduledExecutor。



你可能会想：你是不是搞错了，我明明看到控制台打印的线程名称是自定义的线程池啊。

是的，正是因为打印的是自定义线程名称，我才疏忽了。虽然从最终效果看，确实走了自定义线程池的线程，但这是因为我们在方法上加了@Async。



此时定时任务还是单线程，只不过当这个方法执行时，刚好因为加了@Async，所以走了异步。
SpringBoot单线程的定时任务只需要把每个任务推到异步线程池就可以扭头执行下一个了，时间很短，看起来好像不是串行的！

那么，怎样才能真正替换默认的SingleThreadScheduledExecutor呢？

还是要回到ScheduledAnnotationBeanPostProcessor#finishRegistration()方法：

```

1  private void finishRegistration() {
2      if (this.scheduler != null) {
3          this.registrar.setScheduler(this.scheduler);
4      }
5
6      // 1 --- 查找是否有SchedulingConfigurer类型的自定义的bean (后面解释)
7      if (this.beanFactory instanceof ListableBeanFactory) {
8          Map<String, SchedulingConfigurer> beans =
9              ((ListableBeanFactory) this.beanFactory).getBeansOfType(Sched
10     ulerConfigurer.class);
11     List<SchedulingConfigurer> configurers = new ArrayList<>(beans.values());
12     AnnotationAwareOrderComparator.sort(configurers);
13     for (SchedulingConfigurer configurer : configurers) {
14         configurer.configureTasks(this.registrar);
15     }
16
17     // 2 --- 如果经过上一步registrar中的taskScheduler仍然未被赋值
18     if (this.registrar.hasTasks() && this.registrar.getScheduler() == null
19 ) {
20         Assert.state(this.beanFactory != null, "BeanFactory must be set t
21 o find scheduler by type");
22         // 2.1 --- 查找Spring容器中TaskScheduler类型的Bean
23         try {
24             this.registrar.setTaskScheduler(resolveSchedulerBean(this.bean
25     Factory, TaskScheduler.class, false));
26         }
27         // 2.2 --- 存在多个TaskScheduler类型的Bean
28         catch (NoUniqueBeanDefinitionException ex) {
29             logger.debug("Could not find unique TaskScheduler bean", ex);
30             // 2.1.2 --- 由于存在多个, 这次改为按名字来确定, DEFAULT_TASK_SCHEDU
31             LER_BEAN_NAME = "taskScheduler";
32             try {
33                 this.registrar.setTaskScheduler(resolveSchedulerBean(this.
34     beanFactory, TaskScheduler.class, true));
35             }
36             // 2.1.3 --- 找不到名为"taskScheduler"的Bean, 抛异常
37             catch (NoSuchBeanDefinitionException ex2) {
38                 // 注意下面的异常信息, 解释得很清楚了, 甚至告诉我们如何自定义定时任务
39                 线程池
40                 if (logger.isInfoEnabled()) {
41                     logger.info("More than one TaskScheduler bean exists w
42             ithin the context, and " +
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
948
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1088
1089
1090
1091
1092
1093
1094
1095
1096
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1287
1288
1289
1290
1291
1292
1293
1294
1295
1295
1296
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1387
1388
1389
1389
1390
1391
1392
1393
1394
1395
1395
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1488
1489
1490
1491
1492
1493
1494
1495
1495
1496
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1588
1589
1590
1591
1592
1593
1594
1595
1595
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1688
1689
1690
1691
1692
1693
1694
1695
1695
1696
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1788
1789
1790
1791
1792
1793
1794
1795
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1890
1891
1892
1893
1894
1895
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1990
1991
1992
1993
1994
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2090
2091
2092
2093
2094
2095
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2188
2189
2190
2191
2192
2193
2194
2195
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
220
```

```

37                     "none is named 'taskScheduler'. Mark one of th
38 em as primary or name it 'taskScheduler' " +
39                     "(possibly as an alias); or implement the Sche
40 dulingConfigurer interface and call " +
41                     "ScheduledTaskRegistrar#setScheduler explicitl
42 y within the configureTasks() callback: " +
43                     ex.getBeanNamesFound());
44             }
45         }
46     }
47     // 2.3 --- 不存在TaskScheduler类型的Bean
48     catch (NoSuchBeanDefinitionException ex) {
49         logger.debug("Could not find default TaskScheduler bean", ex);
50         // 2.3.1 --- 查找Spring容器中ScheduledExecutorService类型的Bean
51         try {
52             this.registrar.setScheduler(resolveSchedulerBean(this.bean
53 Factory, ScheduledExecutorService.class, false));
54         }
55         // 2.3.2 --- 存在多个ScheduledExecutorService类型的Bean
56         catch (NoUniqueBeanDefinitionException ex2) {
57             logger.debug("Could not find unique ScheduledExecutorServ
58 ice bean", ex2);
59             // 2.3.2 --- 由于存在多个, 这次改为按名字来确定, DEFAULT_TASK_SC
60 HEDULER_BEAN_NAME = "taskScheduler";
61             try {
62                 this.registrar.setScheduler(resolveSchedulerBean(this.
63 beanFactory, ScheduledExecutorService.class, true));
64             }
65             // 2.3.3 --- 找不到名为"taskScheduler"的Bean, 抛异常
66             catch (NoSuchBeanDefinitionException ex3) {
67                 // 注意下面的异常信息, 解释得很清楚了, 甚至告诉我们如何自定义定
68 时任务线程池
69                 if (logger.isInfoEnabled()) {
70                     logger.info("More than one ScheduledExecutorServic
71 e bean exists within the context, and " +
72                     "none is named 'taskScheduler'. Mark one o
73 f them as primary or name it 'taskScheduler' " +
74                     "(possibly as an alias); or implement the
75 SchedulingConfigurer interface and call " +
76                     "ScheduledTaskRegistrar#setScheduler expli
77 citly within the configureTasks() callback: " +
78                     ex2.getBeanNamesFound());
79             }
80         }
81     }
82     catch (NoSuchBeanDefinitionException ex2) {
83         logger.debug("Could not find default ScheduledExecutorServ
84 ice bean", ex2);

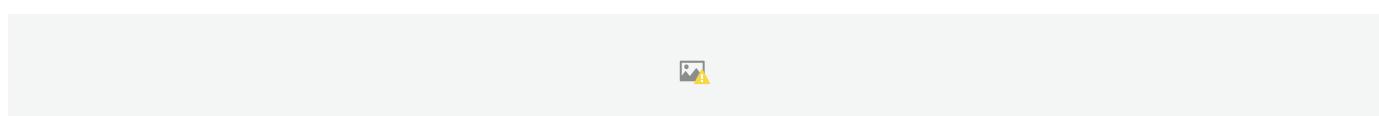
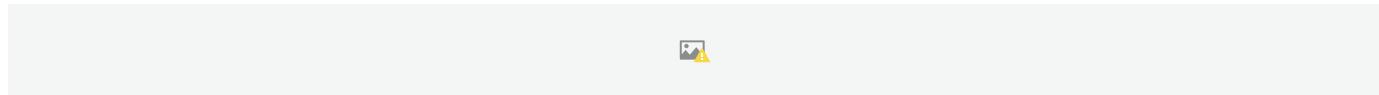
```

```
72                     // Giving up -> falling back to default scheduler within t
73             he registrar...
74                     // 翻译：放弃，沿用registrar内部默认的scheduler（单线程）
75                     logger.info("No TaskScheduler/ScheduledExecutorService bea
76             n found for scheduled processing");
77                     }
78             }
79         this.registrar.afterPropertiesSet();
}
```

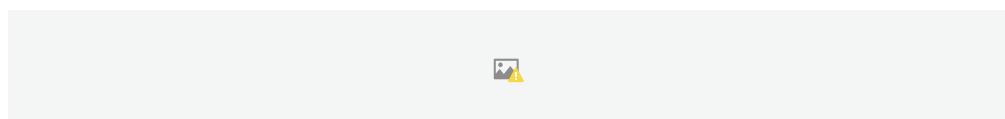
从源码可以看出来，SpringBoot对外暴露了三种自定义定时任务线程池的方法：

- 如果存在SchedulingConfigurer类型的Bean，用SchedulingConfigurer
- 如果存在TaskScheduler类型的Bean，用TaskScheduler
- 如果存在ScheduledExecutorService类型的Bean，用ScheduledExecutorService
- 以上都没有，就用默认的

找到以上任意一种定时任务线程池，都会调用ScheduledTaskRegistrar#setScheduler()为taskScheduler赋值：



而我们之前配置线程池是这样的，不属于上面任意一种：



所以，最终会使用默认的单线程定时任务，然后通过@Async走异步，看起来好像不是串行。





配置定时任务线程池的3种方式

分析源码后，我们很容易得到以下三种配置定时任务线程池的方式（请把@Async去掉，不需要）。



方式1：重写SchedulingConfigurer#configureTasks()

```

1  @Slf4j
2  @Configuration
3  public class ThreadPoolTaskConfig implements WebMvcConfigurer {
4
5      // 方式1: SchedulingConfigurer 其实内部还是设置一个TaskScheduler
6      @Bean
7      public SchedulingConfigurer schedulingConfigurer() {
8          return new MySchedulingConfigurer();
9      }
10
11     static class MySchedulingConfigurer implements SchedulingConfigurer {
12
13         @Override
14         public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
15             ThreadPoolTaskScheduler taskScheduler = new ThreadPoolTaskScheduler();
16             taskScheduler.setPoolSize(3);
17             taskScheduler.setThreadNamePrefix("schedule-task-");
18             taskScheduler.setRejectedExecutionHandler(
19                 new RejectedExecutionHandler() {
20                     /**
21                      * 自定义线程池拒绝策略（模拟发送告警邮件）
22                      */
23                     @Override
24                     public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
25                         log.info("发送告警邮件=====:>嘿沙雕，线上定时任务
卡爆了，当前线程名称为:{}，当前线程池队列长度为:{}",
26                                         r.toString(),
27                                         executor.getQueue().size());
28                     }
29                 });
30             taskScheduler.initialize();
31             taskRegistrar.setScheduler(taskScheduler);
32         }
33     }
34 }

```



这样看起来有点别扭，其实可以把MySchedulingConfigurer单独作为一个类，加上@Component即可。

```
1  @Slf4j
2  @Component
3  class SchedulingConfig implements SchedulingConfigurer {
4
5      @Override
6      public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
7          ThreadPoolTaskScheduler taskScheduler = new ThreadPoolTaskScheduler();
8          taskScheduler.setPoolSize(3);
9          taskScheduler.setThreadNamePrefix("schedule-task-");
10         taskScheduler.setRejectedExecutionHandler(
11             new RejectedExecutionHandler() {
12                 /**
13                  * 自定义线程池拒绝策略 (模拟发送告警邮件)
14                  */
15                 @Override
16                 public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
17                     log.info("发送告警邮件=====:>嘿沙雕, 线上定时任务卡爆了, 当前线程名称为:{}，当前线程池队列长度为:{}",
18                         r.toString(),
19                         executor.getQueue().size());
20                 }
21             });
22         taskScheduler.initialize();
23         taskRegistrar.setScheduler(taskScheduler);
24     }
25 }
```

方式2：@Bean + ThreadPoolTaskScheduler

```
1  @Slf4j
2  @Configuration
3  public class ThreadPoolTaskConfig implements WebMvcConfigurer {
4
5      // 方式2: taskScheduler
6      @Bean("taskScheduler")
7      public Executor taskScheduler() {
8          ThreadPoolTaskScheduler taskScheduler = new ThreadPoolTaskScheduler();
9          taskScheduler.setPoolSize(3);
10         taskScheduler.setThreadNamePrefix("schedule-task-");
11         taskScheduler.setRejectedExecutionHandler(
12             new RejectedExecutionHandler() {
13                 /**
14                  * 自定义线程池拒绝策略 (模拟发送告警邮件)
15                  */
16                 @Override
17                 public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
18                     log.info("发送告警邮件=====:>嘿沙雕, 线上定时任务卡爆了, 当前线程名称为:{}，当前线程池队列长度为:{}",
19                                         r.toString(),
20                                         executor.getQueue().size());
21                 }
22             });
23         taskScheduler.initialize();
24         return taskScheduler;
25     }
26 }
```



方式3: @Bean + ScheduledThreadPoolExecutor

```

1  @Slf4j
2  @Configuration
3  public class ThreadPoolTaskConfig implements WebMvcConfigurer {
4
5      // 方式3
6      @Bean("taskScheduler")
7      public Executor taskScheduler() {
8          ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(
9              3,
10             new RejectedExecutionHandler() {
11                 /**
12                  * 自定义线程池拒绝策略 (模拟发送告警邮件)
13                  */
14                 @Override
15                 public void rejectedExecution(Runnable r, ThreadPoolEx-
ecutor executor) {
16                     log.info("发送告警邮件=====:>嘿沙雕，线上定时任务卡爆
17                         了，当前线程名称为:{}，当前线程池队列长度为:{}",
18                                     r.toString(),
19                                     executor.getQueue().size());
20                 }
21             );
22             executor.setMaximumPoolSize(5);
23             executor.setKeepAliveTime(60, TimeUnit.SECONDS);
24             return executor;
25         }
26     }
27 }
```

个人比较推荐方式1和方式2。

方式1的另一个作用是可以实现动态配置定时任务的时间，但真的很麻烦。

详情可参考：<http://mbcoder.com/dynamic-task-scheduling-with-spring/>

其他公众号上看到的动态配置：https://mp.weixin.qq.com/s/lFURReSuVoQ_kWbAi0ANAoQ

通过配置文件也可改变定时任务线程数

上面几种方式看起来比较复杂，因为我们采用的是自定义线程池的方式，除了改变线程数还重写了拒绝策略等。如果仅仅想要增加定时任务线程池的线程数，可以直接在配置文件中更改：



最后再次强调SpringBoot定时任务和@Async没有任何关系，不需要加@Async。

写这一篇的目的不是为了深究SpringBoot定时任务，因为上一篇说了，它有先天的不足，最致命的就是不支持分布式部署。但我希望通过这一篇的源码分析，大家能再次回顾BeanPostProcessor的机制以及重视线程池的分类。

| 来自: [@Scheduled源码解析](#)

SpringBoot定时任务

定时任务是实际开发中非常普遍的需求，比如定时统计报表、定时更新用户状态等。如果你使用SpringBoot开发项目，那么只需加上`@EnableScheduling + @Scheduled`两个注解即可启用定时任务。

但是SpringBoot提供的定时任务也存在一些小小的坑以及诸多不足，今天我们一起来了解它。

为了避免大家觉得我偷懒，先放几篇上来。其实这些应该安排在另一些文章后，不然一部分读者看起来会有点懵。到时都放上来了我再微调一下。

定时任务示例

```
Java

1   */**
2   * @author qiyu
3   * @date 2020-08-13
4   */
5  @Slf4j
6  @Component
7  @EnableScheduling
8  public class TaskOne {
9
10      /**
11      * 每隔10秒执行test1()
12      */
13      @Scheduled(cron = "*/10 * * * ?")
14      public void test1() {
15          log.info("=====test1任务启动=====");
16          try {
17              Thread.sleep(7 * 1000L);
18          } catch (InterruptedException e) {
19              e.printStackTrace();
20          }
21          log.info("=====test1任务结束=====");
22      }
23
24 }
```

```
2020-08-13 13:31:10.004 INFO 40492 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne      : =====test1任务启动=====  
2020-08-13 13:31:17.005 INFO 40492 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne      : =====test1任务结束=====  
2020-08-13 13:31:20.004 INFO 40492 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne      : =====test1任务启动=====  
2020-08-13 13:31:27.008 INFO 40492 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne      : =====test1任务结束=====  
2020-08-13 13:31:30.000 INFO 40492 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne      : =====test1任务启动=====
```

test1()每隔10秒启动，所以启动时间是13:31:10、13:31:20、13:31:30。由于内部调用Sleep睡眠了7秒，所以每次任务都是在启动7秒后结束。

另外，@EnableScheduling放在Application启动类上也可以，方便整体把控，有可能随着项目的开发，会出现多个定时任务类。

@Scheduled的几个属性

@Schedule提供了多个属性供我们使用，不同的属性有不同的功效。

```
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Repeatable(Schedules.class)
public @interface Scheduled {

    /** A cron-like expression, extending the usual UN*X definition to include triggers ...*/
    String cron() default "";

    /** A time zone for which the cron expression will be resolved. By default, this ...*/
    String zone() default "";

    /** Execute the annotated method with a fixed period in milliseconds between the ...*/
    long fixedDelay() default -1;

    /** Execute the annotated method with a fixed period in milliseconds between the ...*/
    String fixedDelayString() default "";

    /** Execute the annotated method with a fixed period in milliseconds between ...*/
    long fixedRate() default -1;

    /** Execute the annotated method with a fixed period in milliseconds between ...*/
    String fixedRateString() default "";

    /** Number of milliseconds to delay before the first execution of a ...*/
    long initialDelay() default -1;

    /** Number of milliseconds to delay before the first execution of a ...*/
    String initialDelayString() default "";

}
```

fixedDelay: 距离上一次结束的时间

```

12  @Slf4j
13  @Component
14  @EnableScheduling
15  public class TaskOne {
16
17      @Scheduled(fixedDelay = 5 * 1000L)
18      public void test1() {
19          log.info("=====test1任务启动=====");
20          try {
21              Thread.sleep( 2 * 1000L );
22          } catch (InterruptedException e) {
23              e.printStackTrace();
24          }
25          log.info("=====test1任务结束=====");
26      }
27
28  }

```

Debug: TimeFormatApplication TimeFormatApplication

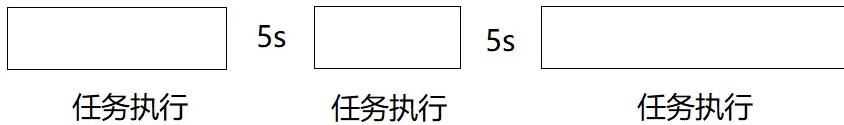
Debugger Console Endpoints

2020-08-13 14:33:48.538 INFO 41629 --- [main] s.a.ScheduledAnnotationBeanPostProcessor : No TaskScheduler/ScheduledExecutorService bean found for scheduled processing
2020-08-13 14:33:48.541 INFO 41629 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne : =====test1任务启动===== 14:33:48
2020-08-13 14:33:48.560 INFO 41629 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 6060 (http) with context path ''
2020-08-13 14:33:48.563 INFO 41629 --- [main] com.bravo.TimeFormatApplication : Started TimeFormatApplication in 9.383 seconds (JVM running for 10.19)
2020-08-13 14:33:50.546 INFO 41629 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne : =====test1任务结束===== 14:33:50
2020-08-13 14:33:55.550 INFO 41629 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne : =====test1任务启动===== 14:33:55
2020-08-13 14:33:57.553 INFO 41629 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne : =====test1任务结束===== 14:33:57

第一个任务从48秒开始，执行任务耗时2秒，在50秒时结束第，第二个任务在第一个任务结束后5秒后开始。

所以，fixDelay=5*1000L 的含义是上一个任务结束5秒后开始下一个任务。

受网络影响，每次任务执行的时间可能会有变化，但不管怎样，下一个任务会在上一个任务结束5秒后开始。



fixedRate: 距离上一次开始的时间

```

12  @Slf4j
13  @Component
14  @EnableScheduling
15  public class TaskOne {
16
17      @Scheduled(fixedRate = 5 * 1000L)
18      public void test1() {
19          log.info("=====test1任务启动=====");
20          try {
21              Thread.sleep( 2 * 1000L );
22          } catch (InterruptedException e) {
23              e.printStackTrace();
24          }
25          log.info("=====test1任务结束=====");
26      }
27  }

```

Debug: TimeFormatApplication

```

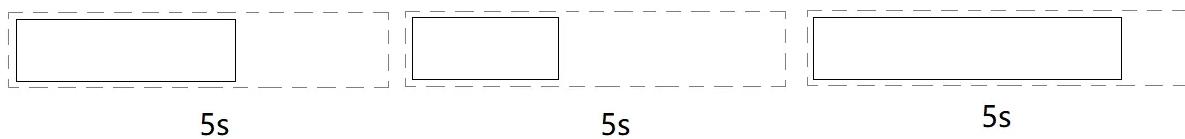
↑ 2020-08-13 14:40:45.517 INFO 41729 --- [main] s.a.ScheduledAnnotationBeanPostProcessor : No TaskScheduler/ScheduledExecutorService bean found for scheduled processing
↓ 2020-08-13 14:40:45.522 INFO 41729 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne : =====test1任务启动===== 14:40:45
2020-08-13 14:40:45.543 INFO 41729 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 6060 (http) with context path ''
: Started TimeFormatApplication in 9.636 seconds (JVM running for 10.259)
2020-08-13 14:40:45.548 INFO 41729 --- [pool-2-thread-1] com.bravo.TimeFormatApplication : =====test1任务结束===== 14:40:47
2020-08-13 14:40:47.526 INFO 41729 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne : =====test1任务启动===== 14:40:50
2020-08-13 14:40:50.526 INFO 41729 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne : =====test1任务结束===== 14:40:52
2020-08-13 14:40:52.531 INFO 41729 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne

```

第一个任务从45秒开始，执行任务耗时2秒，在50秒时结束第，第二个任务在第一个任务结束3秒后就开始了。

所以，fixRate=5*1000L 的含义是上一个任务开始5秒后开始下一个任务。

不论上一个任务何时结束，下一个任务会在上一个任务开始5秒后开始。



通过上面这幅图，我们很容易想到里面的长方形很有可能超出外面的虚线框，即任务耗时比设定fixRate时间长。比如，即使上一个任务耗时6秒，那么按理来说下一个任务应该还是要在第5秒开始。

```

12  @Slf4j
13  @Component
14  @EnableScheduling
15  public class TaskOne {
16
17      @Scheduled(fixedRate = 5 * 1000L)
18      public void test1() {
19          log.info("=====test1任务启动=====");
20          try {
21              Thread.sleep( 6 * 1000L );
22          } catch (InterruptedException e) {
23              e.printStackTrace();
24          }
25          log.info("=====test1任务结束=====");
26      }
27  }

```

TaskOne

Debug: TimeFormatApplication

```

↑ 2020-08-13 14:57:51.783 INFO 42021 --- [main] s.a.ScheduledAnnotationBeanPostProcessor : No TaskScheduler/ScheduledExecutorService bean found for scheduled processing
↓ 2020-08-13 14:57:51.787 INFO 42021 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne : =====test1任务启动===== 14:57:51
2020-08-13 14:57:51.804 INFO 42021 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 6060 (http) with context path ''
: Started TimeFormatApplication in 12.732 seconds (JVM running for 13.506)
2020-08-13 14:57:51.808 INFO 42021 --- [pool-2-thread-1] com.bravo.TimeFormatApplication : =====test1任务结束===== 14:57:57
2020-08-13 14:57:57.792 INFO 42021 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne : =====test1任务启动===== 14:57:57
2020-08-13 14:58:03.797 INFO 42021 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne : =====test1任务结束===== 14:58:03
2020-08-13 14:58:03.797 INFO 42021 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne

```

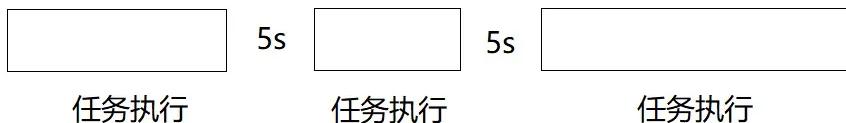
然而实际情况是，由于上一个任务执行了6秒才结束，导致下一个任务多等待了1秒。

不是说在fixedRate设定下，下一个任务会在上一个任务开始后固定时间启动吗？

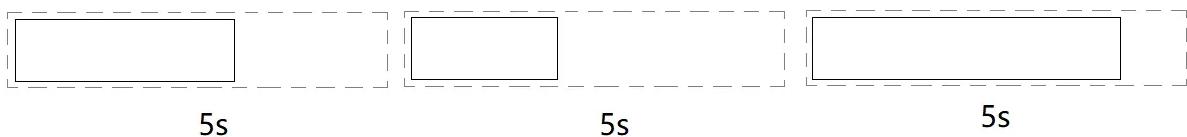
其实并不是fixRate的问题，而是因为**SpringBoot的定时任务默认是单线程的**（从截图可以看出始终只有pool-2-thread-1在执行当前定时任务）。

好了，我们先停下来梳理一下：

- fixDelay：用于指定上一个任务结束与下一个任务开始的时间间隔



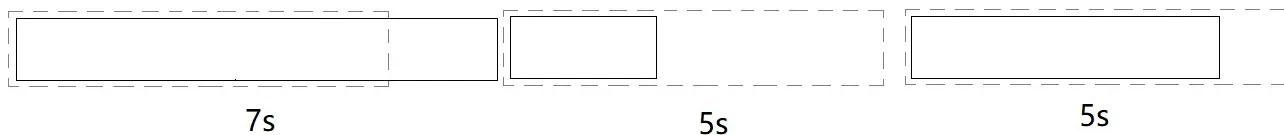
- fixRate：用于指定上一个任务开始与下一个任务开始的时间间隔



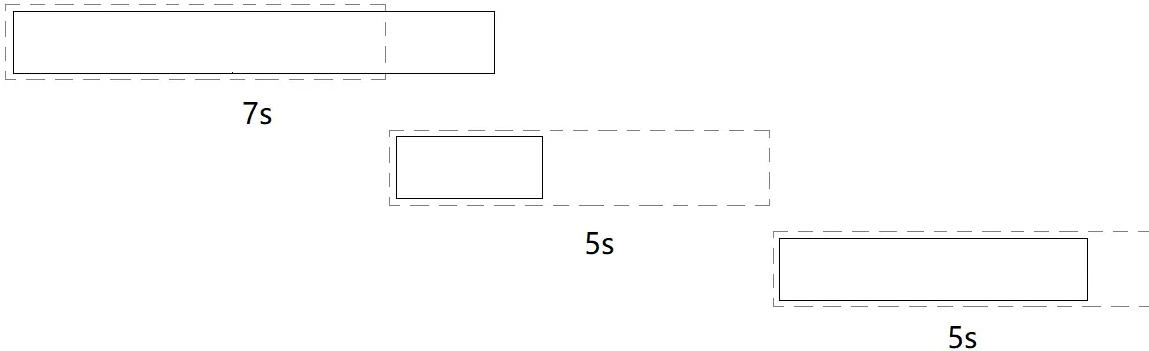
对于fixDelay，任务实际的执行耗时不会影响整个流程，它只保证两个任务首尾的时间间隔。

对于fixRate，它预期每隔多久就启动一个新任务，不论上一个任务执行结束与否。

但很遗憾，**SpringBoot默认定时任务是单线程的**，如果任务执行时间较短，那么fixRate可以保证每个任务开始时间的间隔稳定性，但如果上一个任务耗时异常，那么下一个任务会被往后顶。



解决办法：为定时任务指定线程池，每个任务都跑在独立的线程中，不存在谁把谁往后顶的情况。



关于如何将SpringBoot的定时任务配置为多线程模式，我们会在后面介绍。

initialDelay：启动多少秒后开始首次任务

不论fixDelay还是fixRate，默认都是项目启动就立即执行，随后按照指定间隔时间重复。
如果希望推迟首次执行时间，可以用initialDelay指定。

```
@Scheduled(initialDelay = 20 * 1000L, fixedRate = 5 * 1000L)
public void test1() {
    log.info("=====test1任务启动=====");
    try {
        Thread.sleep( millis: 2 * 1000L);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    log.info("=====test1任务结束=====");
}
```

```
2020-08-13 15:26:03.512 INFO 42477 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 6060 (http) with context path ''
2020-08-13 15:26:03.516 INFO 42477 --- [           main] com.bravo.TimeFormatApplication      : Started TimeFormatApplication in 9.536 seconds (JVM running for 10.239)
2020-08-13 15:26:23.497 INFO 42477 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne   : ======test1任务启动=====
2020-08-13 15:26:25.501 INFO 42477 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne   : ======test1任务结束=====
2020-08-13 15:26:28.497 INFO 42477 --- [pool-2-thread-1] com.bravo.timeformat.task.TaskOne   : ======test1任务启动=====
```

项目在15:26:03启动完毕，而定时任务由于设定了initialDelay=20*1000L，初次启动往后延迟20秒。

cron：定时执行（最常用）

cron是实际工作中最常用的，什么fixDelay和fixRate往往用的很少。

指定cron表达式，一般只要写6位即可，分别代表@Schedule(cron = "秒 分 时 日 月 星期 [年]"), 第7位[年]可以不写。

表达式的书写规则大家去[在线Cron表达式生成器](#)玩一下就知道了。个人觉得最实用的就是记住以下两点：

- */number表示“每隔...”，是最实用的
- 逗号表示“或”，比如 8,13,18 表示 8或13或18

比如，在[秒]的位置写上 */10，表示每隔10秒执行一次。

Cron 表 达式: 反解析到UI

最近5次运行时间:

2020-08-13 15:33:30
2020-08-13 15:33:40
2020-08-13 15:33:50
2020-08-13 15:34:00
2020-08-13 15:34:10

在[分]的位置写上 */10，表示每隔10分钟执行一次。

Cron 表 达式: 反解析到UI

最近5次运行时间:

2020-08-13 15:40:00
2020-08-13 15:50:00
2020-08-13 16:00:00
2020-08-13 16:10:00
2020-08-13 16:20:00

此时记得把[秒]位置的设置为0，表示“每隔10分钟，且整分才执行”，你如果设定为*，而*表示任意，含义就变成“每隔10分钟，且任意秒都执行”。

Cron 表 达式: 反解析到UI

最近5次运行时间:

2020-08-13 15:50:00
2020-08-13 15:50:01
2020-08-13 15:50:02
2020-08-13 15:50:03
2020-08-13 15:50:04

为了验证这个解释，我特意把[秒]改为5，意思就是“每隔10分钟，且秒数为5才执行”：

Cron 表 达式: 反解析到UI

最近5次运行时间:

2020-08-13 15:50:05
2020-08-13 16:00:05
2020-08-13 16:10:05
2020-08-13 16:20:05
2020-08-13 16:30:05

接下来，举个日常最普遍的例子，在[时]的位置设置 0,8,18 表示：每天0点、8点、18点更新

Cron 表 达式: 反解析到UI

最近5次运行时间:

2020-08-13 18:00:00
2020-08-14 00:00:00
2020-08-14 08:00:00
2020-08-14 18:00:00
2020-08-15 00:00:00

一个线上的坑

我相信，很多人实际开发都写过下面这样的定时任务，它们原本的设定都是在同一个时刻开始的。

```

1  @Slf4j
2  @Component
3  @EnableScheduling
4  public class TaskOne {
5
6      @Scheduled(cron = "*/10 * * * * ?")
7  public void test1() {
8      log.info("=====test1任务启动=====");
9      try {
10          Thread.sleep(2 * 1000L);
11      } catch (InterruptedException e) {
12          e.printStackTrace();
13      }
14      log.info("=====test1任务结束=====");
15  }
16
17  @Scheduled(cron = "*/10 * * * * ?")
18  public void test2() {
19      log.info("=====test2任务启动=====");
20      try {
21          // 模拟远程服务卡死
22          Thread.sleep(1000 * 1000L);
23      } catch (InterruptedException e) {
24          e.printStackTrace();
25      }
26      log.info("=====test2任务结束=====");
27  }
28
29  @Scheduled(cron = "*/10 * * * * ?")
30  public void test3() {
31      log.info("=====test3任务启动=====");
32      try {
33          Thread.sleep(2 * 1000L);
34      } catch (InterruptedException e) {
35          e.printStackTrace();
36      }
37      log.info("=====test3任务结束=====");
38  }
39
40  }

```

一般情况下是没有问题的，甚至你根本没意识到它们其实不是同时开始的~因为你可能都不知道SpringBoot定时任务默认单线程。但由于每个任务执行时间短且一般需求对时间准确度要求并不特别严

格，串行执行慢个2~3秒都可以接受。

但某次项目发布后，同事告诉我他的定时任务不跑了。我看了半天，才发现他没有配置线程池，而且又因为引入了Redis分布式锁，不小心发生了死锁卡在那了，最终导致其他任务都没法启动（串行化）。

配置线程池

```
1  /**
2   * 线程池配置
3   * @author qiyu
4   *
5   */
6  @EnableAsync // 来了，这里挖了一个坑
7  @Configuration
8  public class ThreadPoolTaskConfig implements WebMvcConfigurer {
9
10     @Bean("taskExecutor")
11     public Executor taskExecutor() {
12         ThreadPoolExecutor executor = new ThreadPoolExecutor();
13         executor.setCorePoolSize(3);
14         executor.setMaxPoolSize(3);
15         executor.setQueueCapacity(5);
16         executor.setKeepAliveSeconds(10);
17         executor.setThreadNamePrefix("async-task-");
18
19         // 线程池对拒绝任务的处理策略
20         executor.setRejectedExecutionHandler(new ThreadPoolExecutor.Caller
21             RunsPolicy());
22         // 初始化
23         executor.initialize();
24         return executor;
25     }
26 }
```

这里只设定了3个线程数（配置可以抽取到yml）。

然后给每个定时任务的方法上加上@Async("taskExecutor")即可。

```
@Async("taskExecutor")
@scheduled(cron = "*/10 * * * * ?")
public void test1() {
    log.info("=====test1任务启动=====");
    try {
        Thread.sleep( millis: 2 * 1000L );
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    log.info("=====test1任务结束=====");
}
```

由于项目中可能配置多个线程池，所以个人建议创建线程池时最好指定有意义的名字（比如async-task-），不然线上日志就炸了，根本不知道这个线程是干啥的。另外，使用@Async时最好明确使用哪个线程池，比如@Async("taskExecutor")，因为项目中的线程池一般都是“专池专用”，这是一个好习惯。

这样就没问题了吗？



看起来还行...任务2卡在那并不影响其他两个任务执行。

但我们把时间拉长，就会发现最终线程池的全部3个线程都卡在任务2上了：



因为每到时间点，都需要去执行3个任务。而线程池总共就3个线程，其中一个是大坑，3个线程随机分配给每个任务，最终3个线程都会掉坑里出不来。

如何保证定时任务可用

所以啊，对于定时任务异常，光靠配置定时任务线程池还是不行的，最终线程池仍会枯竭，导致所有定时任务阻塞。除非你每个定时任务都专门配一个线程池…

所以发生定时任务耗时异常这种情况，最重要的是及时发现并修复。

在配置线程池时，我们可以指定拒绝策略（线程池队列满了之后触发），SpringBoot默认提供了4种：

- CallerRunsPolicy：不使用异步线程，直接主线程执行
- AbortPolicy：丢弃当前任务，直接抛异常
- DiscardPolicy：丢弃当前任务，无声无息（不抛异常）
- DiscardOldestPolicy：丢弃队列里最老的任务，执行当前任务

这4种拒绝策略被定义在ThreadPoolExecutor类的内部，是静态内部类。

在前面几篇文章中，大家会发现我经常是这样写的：

但实际开发建议自定义拒绝策略。为什么呢？

实际发现线程池不够用了，你直接跑主线程吗？还记得Tomcat被卡爆的案例吗？直接丢弃？你确定对业务没影响吗？如果业务本身不在乎请求失败，那是没关系的，否则丢弃策略就不合适了。

一般可以选择在丢弃策略里使用MQ（延后缓冲）或者发邮件告警（及时发现），只要实现RejectedExecutionHandler接口即可：

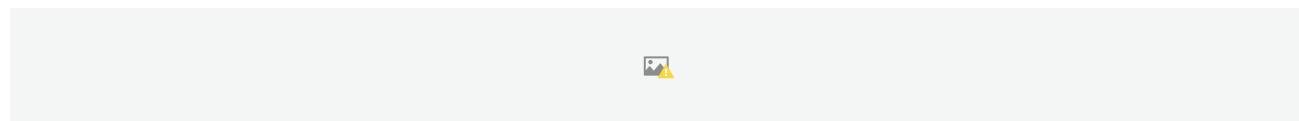
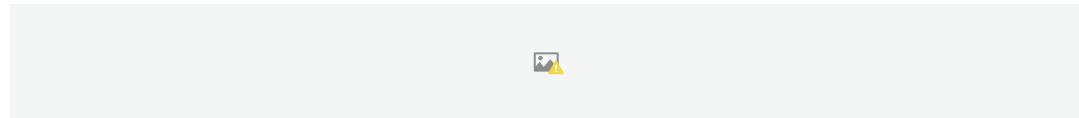
```
1  /**
2   * 线程池配置
3   * @author qiyu
4   *
5   */
6  @Slf4j
7  @EnableAsync // 第二次提示这个坑
8  @Configuration
9  public class ThreadPoolTaskConfig implements WebMvcConfigurer {
10
11     @Bean("taskExecutor")
12     public Executor taskExecutor() {
13         ThreadPoolExecutor executor = new ThreadPoolExecutor();
14         executor.setCorePoolSize(3);
15         executor.setMaxPoolSize(3);
16         executor.setQueueCapacity(5);
17         executor.setKeepAliveSeconds(10);
18         executor.setThreadNamePrefix("async-task-");
19
20         // 线程池对拒绝任务的处理策略
21         executor.setRejectedExecutionHandler(new RejectedExecutionHandler()
22         ){
23             /**
24              * 自定义线程池拒绝策略（模拟发送告警邮件）
25              */
26             @Override
27             public void rejectedExecution(Runnable r, ThreadPoolExecutor e
28                 xecutor) {
29                 log.info("发送告警邮件=====:>嘿沙雕，线上定时任务卡爆了，当前线
30                 程名称为:{}，当前线程池队列长度为:{}",
31                 r.toString(),
32                 executor.getQueue().size());
33             }
34         );
35         // 初始化
36         executor.initialize();
37         return executor;
38     }
39 }
```



好了，这样我们就具备完善的报警机制，可以及时发现线上的问题。

而我的同事最终也发现了线上的bug：

由于项目是多节点部署，为了不重复执行，他引入了Redis分布式锁，不知道为什么产生了死锁。



同事引入分布式锁的目的是保证某一时刻不同节点的定时任务不会重复执行（谁获取锁谁执行）。

SpringBoot定时任务的不足

其实，SpringBoot的定时任务是很鸡肋的：

- 不支持集群时单节点启动（同事使用Redis分布式锁就是为了解决避免多节点重复执行）
- 不支持分片任务
- 不支持失败重试（一个任务失败了就失败了，不会重试）
- 动态调整比较繁琐（我曾经做过一个项目，要求前端页面可以动态配置任务启动的时间点）
- ...

所以，对于多节点部署或者分布式项目，还是乖乖用Elastic-Job或者XXL-Job吧。

看到这，似乎很完美。其实我配错了。上面线程池的配置严格意义上来说不是定时任务线程池，而是异步线程池。定时任务理论上只要加@Scheduled，@Async是异步线程相关的。

不要觉得我在钻牛角尖，我以前犯的错，大部分人也会犯。我想说，线程池本身体系蛮复杂的，很多人其实都分不清哪个是哪个，所以我想借这个错误让大家正视线程池。

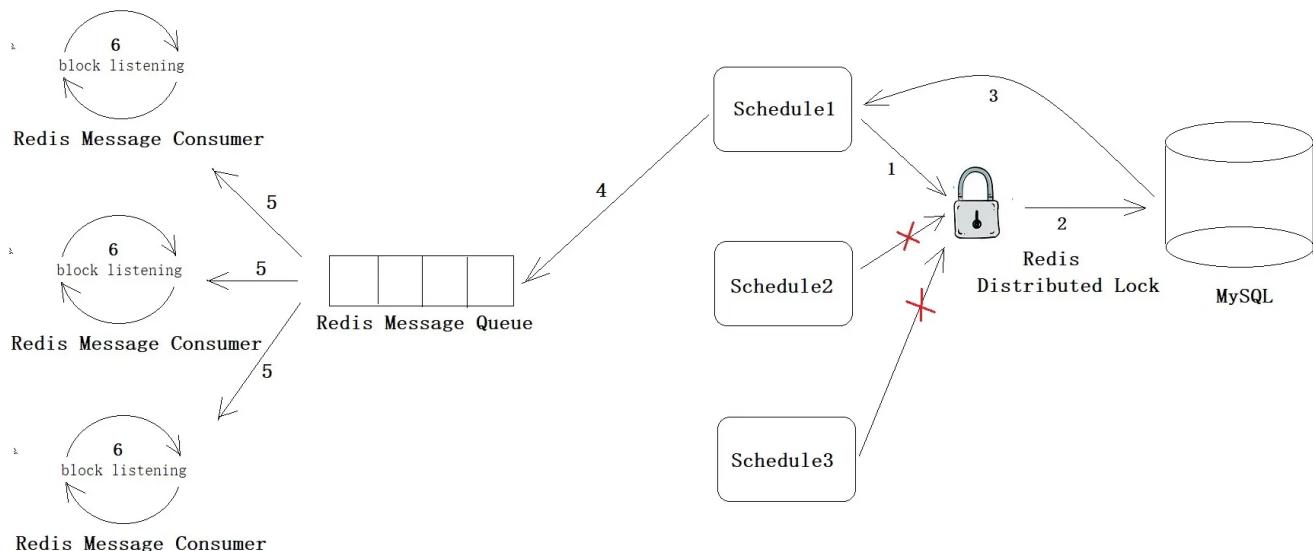
正确的配置方法请看下一篇@Scheduled源码解析。

| 来自: [SpringBoot定时任务](#)

浅谈Redis分布式锁(下)

自定义Redis分布式锁的弊端

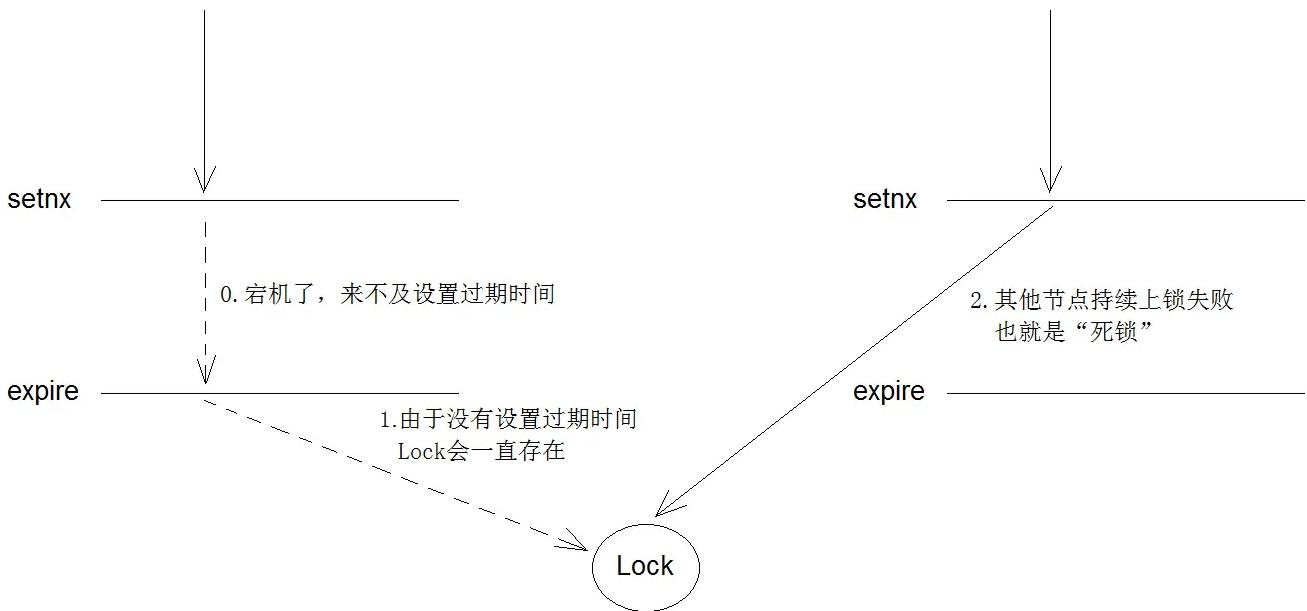
在上一篇我们自定义了一个Redis分布式锁，用来解决多节点定时任务的拉取问题（避免任务重复执行）：



但仍然存在很多问题：

- 加锁操作不是原子性的 (setnx和expire两步操作不是原子性的，中间宕机会导致死锁)

```
Java |  
1  public boolean tryLock(String lockKey, String value, long expireTime, TimeUnit timeUnit) {  
2      // 1.先setnx  
3      Boolean lock = redisTemplate.opsForValue().setIfAbsent(lockKey, value)  
;  
4      if (lock != null && lock) {  
5          // 2.再expire  
6          redisTemplate.expire(lockKey, expireTime, timeUnit);  
7          return true;  
8      } else {  
9          return false;  
10     }  
11 }
```



当然啦，高版本的SpringBoot Redis依赖其实提供了加锁的原子性操作：

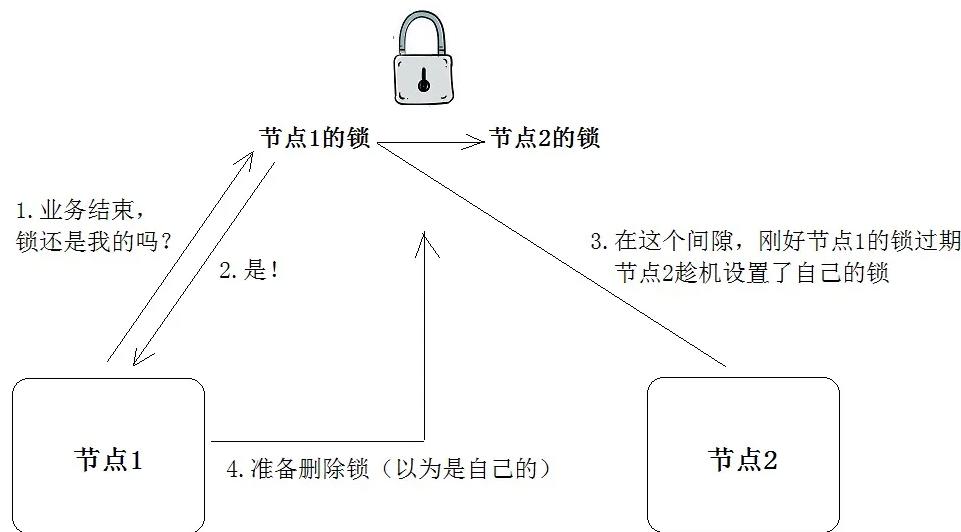
```

1  /**
2   * 尝试上锁: setNX + expire
3   *
4   * @param lockKey      锁
5   * @param value        对应的值
6   * @param expireTime  过期时间
7   * @param timeUnit    时间单位
8   * @return
9   */
10  @Override
11  public boolean tryLock(String lockKey, String value, long expireTime, TimeUnit timeUnit) {
12      try {
13          // 高版本SpringBoot的setIfAbsent可以设置4个参数，一步到位
14          redisTemplate.opsForValue().setIfAbsent(lockKey, value, expireTime,
15          , timeUnit);
16          return true;
17      } catch (Exception e) {
18          e.printStackTrace();
19      }
20      return false;
}

```

从 Redis 2.6.12 版本开始（现在6.x了...），`SET` 命令的行为可以通过一系列参数来修改，也因为 `SET` 命令可以通过参数来实现和 `SETNX`、`SETEX` 和 `PSETEX` 三个命令的效果，所以将来的 Redis 版本可能会废弃并最终移除 `SETNX`、`SETEX` 和 `PSETEX` 这三个命令。

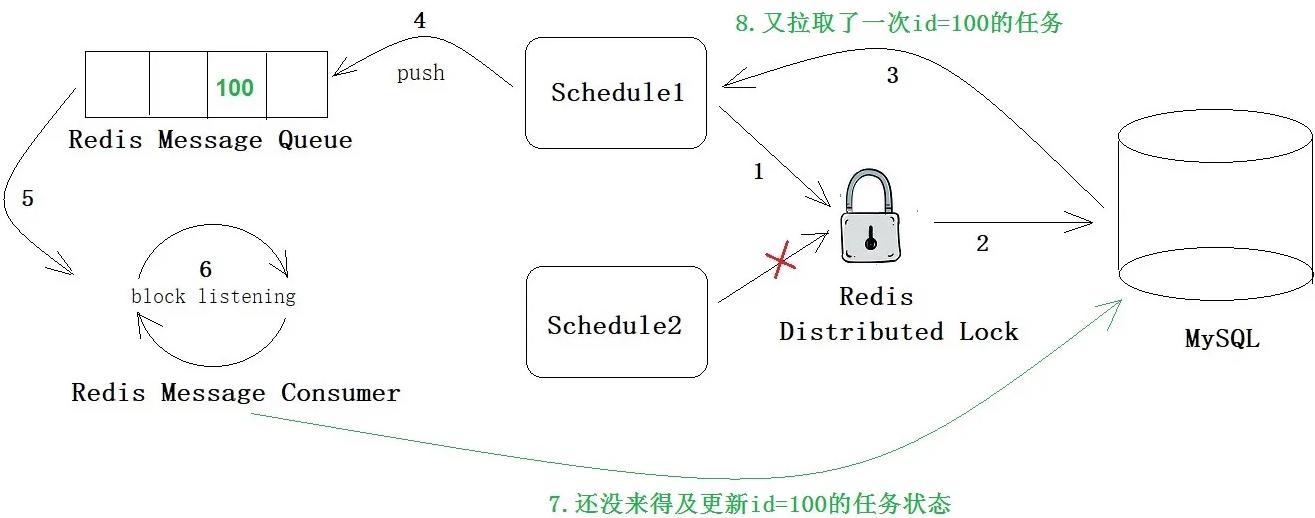
- 解锁操作不是原子性的（可能造成不同节点之间互相删锁）



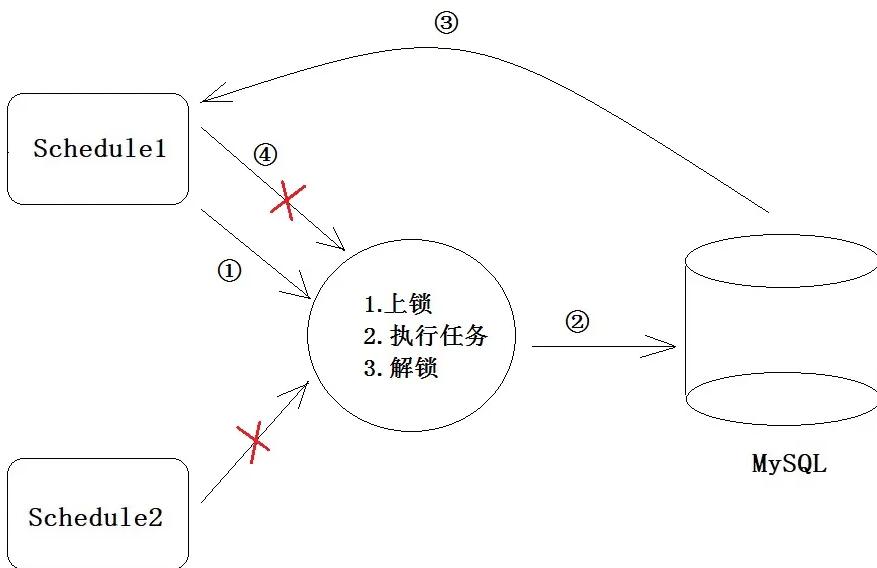
虽然上一篇设计的unLock()不是原子操作，但可以避免不同节点之间互相删锁

```
1  public boolean unLock(String lockKey, String value) {
2      // 1. 获取锁的value, 存的是MACHINE_ID
3      String machineId = (String) redisTemplate.opsForValue().get(lockKey);
4      if (StringUtils.isNotEmpty(machineId) && machineId.equals(value)) {
5          // 2. 只能删除当前节点设置的锁
6          redisTemplate.delete(lockKey);
7          return true;
8      }
9      return false;
10 }
```

- 畏难情绪作祟，不想考虑锁续期的问题，企图采用队列的方式缩减定时任务执行时间，直接把任务丢到队列中。但实际上可能存在任务堆积，个别情况下会出现：上次已经拉取某个任务并丢到Redis队列中，但由于队列比较繁忙，该任务还未被执行，数据库状态也尚未更改为status=1（已执行），结果下次又拉取一遍，重复执行（简单的解决策略是：虽然无法阻止入队，但是出队消费时可以判断where status=0后执行）



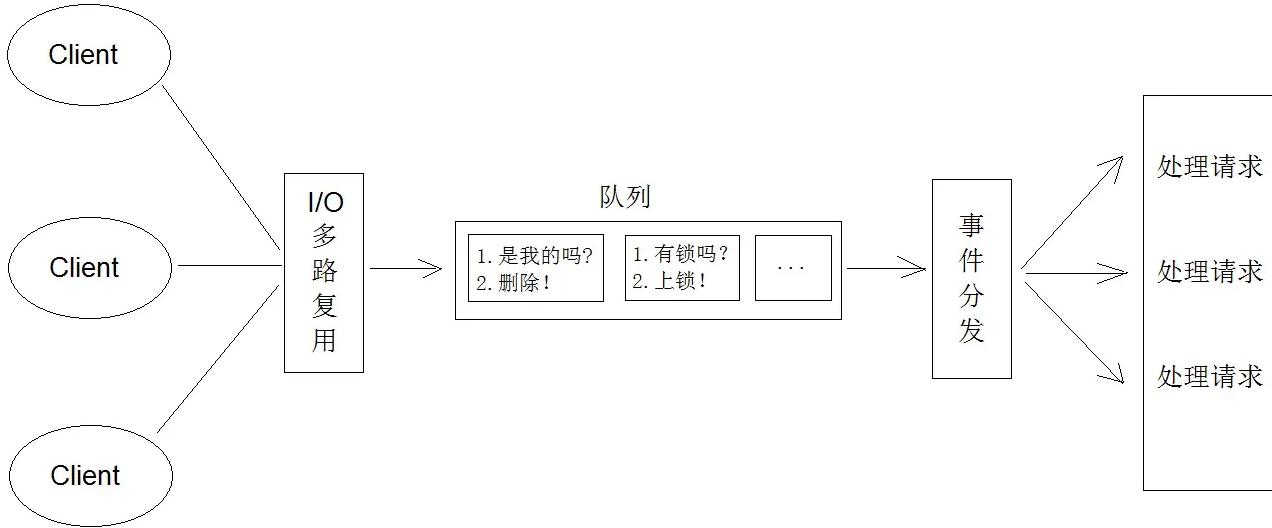
引入Redis Message Queue会让系统变得更加复杂，我之前就因为使用了上面的模型导致各种偶发性的BUG，非常不好排查。一般来说，定时任务应该设计得简单点：



也就是说，绕来绕去，想要设计一个较完备的Redis分布式锁，必须至少解决3个问题：

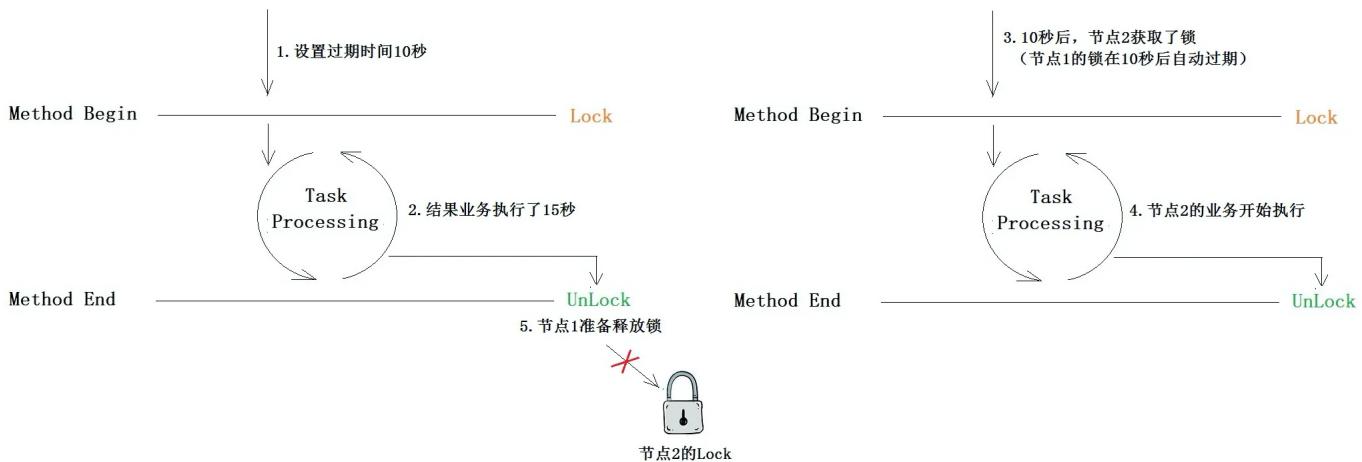
- 加锁原子性（setnx和expire要保证原子性，否则会容易发生死锁）
- 解锁原子性（不能误删别人的锁）
- 需要考虑业务/定时任务执行的时间，并为锁续期

如果不考虑性能啥的，加解锁原子性都可以通过lua脚本实现（利用Redis单线程的特性）：



一次执行一个脚本，要么成功要么失败，不会和其他指令交错执行。

最难的是如何根据实际业务的执行时间给锁续期！虽然我们已经通过判断MACHINE_ID避免了不同节点互相删除锁：



但本质上我们需要的是：



本文我们的主要目标就是实现锁续期！

好在Redisson已经实现了，所以目标又变成：了解Redisson的锁续期机制。

Redisson案例

Redisson环境搭建

```
▼ YAML |  
1 server:  
2   port: 8080  
3  
4 spring:  
5   redis:  
6     host: # 见小册开头《阿里云服务账号》  
7     password: # 见小册开头《阿里云服务账号》  
8     database: 1  
9  
10 # 调整控制台日志格式，稍微精简一些（非必要操作）  
11 logging:  
12   pattern:  
13     console: "%d{yyyy-MM-dd HH:mm:ss} - %thread - %msg%n"
```

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-web</artifactId>
5   </dependency>
6   <dependency>
7     <groupId>org.projectlombok</groupId>
8     <artifactId>lombok</artifactId>
9     <optional>true</optional>
10    </dependency>
11   <dependency>
12     <groupId>org.springframework.boot</groupId>
13     <artifactId>spring-boot-starter-test</artifactId>
14     <scope>test</scope>
15   </dependency>
16   <dependency>
17     <groupId>org.springframework.boot</groupId>
18     <artifactId>spring-boot-starter-data-redis</artifactId>
19   </dependency>
20   <!--大家也可以单独引入Redisson依赖，然后通过@Configuration自己配置RedissonClient-->
21   <dependency>
22     <groupId>org.redisson</groupId>
23     <artifactId>redisson-spring-boot-starter</artifactId>
24     <version>3.13.6</version>
25   </dependency>
26 </dependencies>
```

然后就可以在test包下测试了~

lock()方法初探

```
1  @Slf4j
2  @RunWith(SpringRunner.class)
3  @SpringBootTest
4  public class RLockTest {
5
6      @Autowired
7      private RedissonClient redissonClient;
8
9      @Test
10     public void testRLock() throws InterruptedException {
11         new Thread(this::testLockOne).start();
12         new Thread(this::testLockTwo).start();
13
14         TimeUnit.SECONDS.sleep(200);
15     }
16
17     public void testLockOne(){
18         try {
19             RLock lock = redissonClient.getLock("bravo1988_distributed_loc
k");
20             log.info("testLockOne尝试加锁...");
21             lock.lock();
22             log.info("testLockOne加锁成功...");
23             log.info("testLockOne业务开始...");
24             TimeUnit.SECONDS.sleep(50);
25             log.info("testLockOne业务结束...");
26             lock.unlock();
27             log.info("testLockOne解锁成功...");
28         } catch (InterruptedException e) {
29             e.printStackTrace();
30         }
31     }
32
33     public void testLockTwo() {
34         try {
35             RLock lock = redissonClient.getLock("bravo1988_distributed_loc
k");
36             log.info("testLockTwo尝试加锁...");
37             lock.lock();
38             log.info("testLockTwo加锁成功...");
39             log.info("testLockTwo业务开始...");
40             TimeUnit.SECONDS.sleep(50);
41             log.info("testLockTwo业务结束...");
42             lock.unlock();
43             log.info("testLockTwo解锁成功...");
```

```
44         } catch (InterruptedException e) {
45             e.printStackTrace();
46         }
47     }
48 }
49 }
```

结果

```
2020-11-21 14:24:33 - Thread-3 - testLockTwo尝试加锁...
2020-11-21 14:24:33 - Thread-2 - testLockOne尝试加锁...
=====> testLockOne()执行过程中, testLockTwo()一直阻塞 <=====
2020-11-21 14:24:33 - Thread-2 - testLockOne加锁成功...
2020-11-21 14:24:33 - Thread-2 - testLockOne业务开始...
2020-11-21 14:25:23 - Thread-2 - testLockOne业务结束...
2020-11-21 14:25:23 - Thread-2 - testLockOne解锁成功...
=====> testLockOne()执行结束释放锁, testLockTwo()抢到锁 <=====
2020-11-21 14:25:23 - Thread-3 - testLockTwo加锁成功...
2020-11-21 14:25:23 - Thread-3 - testLockTwo业务开始...
2020-11-21 14:26:13 - Thread-3 - testLockTwo业务结束...
2020-11-21 14:26:13 - Thread-3 - testLockTwo解锁成功...
```

通过上面的代码，我们有以下疑问：

- lock()方法是原子性的吗？
- lock()有设置过期时间吗？是多少？
- lock()实现锁续期了吗？
- lock()方法怎么实现阻塞的？又怎么被唤醒？

先忘了这些，跟着我们走一遍lock()源码就明白了。

lock()源码解析

lock()加锁，去除异常的情况，无非加锁成功、加锁失败两种情况，我们先看加锁成功的情况。

流程概览

我们从这段最简单的代码入手：

```
1  @Slf4j
2  @RunWith(SpringRunner.class)
3  @SpringBootTest
4  public class RLockTest {
5
6      @Autowired
7      private RedissonClient redissonClient;
8
9      @Test
10     public void testLockSuccess() throws InterruptedException {
11         RLock lock = redissonClient.getLock("bravo1988_distributed_lock");
12         log.info("准备加锁...");
13         lock.lock();
14         log.info("加锁成功...");
15         TimeUnit.SECONDS.sleep(300);
16     }
17 }
```

大家跟着我们先打几个断点（SpringBoot2.3.4）：



The screenshot shows a Java code editor with the following code:

```
26
27  ►  @Test
28  public void testLockSuccess() throws InterruptedException {
29      RLock lock = redissonClient.getLock( name: "bravo1988_distributed_lock");
30      log.info("准备加锁...");
31  ⚡  lock.lock();
32      log.info("加锁成功...");
33      TimeUnit.SECONDS.sleep( timeout: 300);
34 }
```

A red circular icon with a white dot, indicating a breakpoint, is positioned next to the line number 31. The line containing the breakpoint is highlighted with a brown background.

```
143
144     @Override
145     public void lock() {
146         try {
147             lock(leaseTime: -1, unit: null, interruptibly: false);
148         } catch (InterruptedException e) {
149             throw new IllegalStateException();
150         }
151     }
```



```
217     // get(lockAsync(leaseTime, unit));
218 }
219
220     private Long tryAcquire(long waitTime, long leaseTime, TimeUnit unit, long threadId) { waitTime: -1 leaseTime: -1 unit: null threadId: 1
221         return get(tryAcquireAsync(waitTime, leaseTime, unit, threadId)); waitTime: -1 leaseTime: -1 unit: null threadId: 1
222     }
```



注意啊，把截图中能看到的断点都打上。

OK，接着大家自己启动DEBUG，感受一下大致流程，然后看下面的注释：

```

1 // redisson.lock()
2 Override
3 public void lock() {
4     try {
5         lock(-1, null, false);
6     } catch (InterruptedException e) {
7         throw new IllegalStateException();
8     }
9 }
10
11 // 为了方便辨认，我直接把传进来的参数写在参数列表上
12 private void lock(long leaseTime=-1, TimeUnit unit=null, boolean interrup-
tibly=false) throws InterruptedException {
13     // 获取当前线程id
14     long threadId = Thread.currentThread().getId();
15     // 尝试上锁。上锁成功返回null，上锁失败返回ttl
16     Long ttl = tryAcquire(-1, leaseTime=-1, unit=null, threadId=666);
17     // 上锁成功，方法结束，回到主线程执行业务啦（后台有个定时任务在给当前锁续期）
18     if (ttl == null) {
19         return;
20     }
21
22     // 上锁成功就不走下面的流程了，所以这里直接省略
23     // 略：加锁失败后续流程...
24 }
25
26 // 尝试上锁。上锁成功返回null，上锁失败返回【当前已经存在的锁】的ttl，方便调用者判断多
久之后能重新获取锁
27 private Long tryAcquire(long waitTime=-1, long leaseTime=-1, TimeUnit uni-
t=null, long threadId=666) {
28     /**
29      * 有两次调用：1.tryAcquireAsync()返回Future 2.从Future获取异步结果（异步结
果就是ttl）
30      * 重点是tryAcquireAsync()
31      */
32     return get(tryAcquireAsync(waitTime=-1, leaseTime=-1, unit=null, thre-
adId=666));
33 }
34
35 // 获取过期时间（非重点）
36 protected final <V> V get(RFuture<V> future) {
37     return commandExecutor.get(future);
38 }
39
40

```

```

41 // 重点, 加锁后返回RFuture, 内部包含ttl。调用本方法可能加锁成功, 也可能加锁失败, 外界
42 // 可以通过ttl判断
43 private <T> RFuture<Long> tryAcquireAsync(long waitTime=-1, long leaseTim
44 e=-1, TimeUnit unit=null, long threadId=666) {
45
46     // lock()默认leaseTime=-1, 所以会跳过if
47     if (leaseTime != -1) {
48         return tryLockInnerAsync(waitTime, leaseTime, unit, threadId, Red
49 isCommands.EVAL_LONG);
50     }
51
52     // 执行lua脚本, 尝试加锁并返回RFuture。这个方法是异步的, 其实是把任务提交给线程
53 池
54     RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(
55         waitTime=-1,
56         commandExecutor.getConnection
57 Manager().getCfg().getLockWatchdogTimeout()=30秒,
58         TimeUnit.MILLISECONDS,
59         threadId=666,
60         RedisCommands.EVAL_LONG);
61
62     // 设置回调方法, 异步线程与Redis交互得到结果后会回调BiConsumer#accept()
63     ttlRemainingFuture.onComplete((ttlRemaining, e) -> {
64         // 发生异常时直接return
65         if (e != null) {
66             return;
67         }
68
69         // 说明加锁成功
70         if (ttlRemaining == null) {
71             // 启动额外的线程, 按照一定规则给当前锁续期
72             scheduleExpirationRenewal(threadId);
73         }
74     });
75
76     // 返回RFuture, 里面有ttlRemaining
77     return ttlRemainingFuture;
78 }
79
80 // 执行lua脚本尝试上锁
81 <T> RFuture<T> tryLockInnerAsync(long waitTime=-1, long leaseTime=30*1000
82 , TimeUnit unit=毫秒, long threadId=666, RedisStrictCommand<T> command) {
83     internalLockLeaseTime = unit.toMillis(leaseTime);
84
85     /**
86      * 大家去看一下evalWriteAsync()的参数列表, 看看每个参数都代表什么, 就能理解KEY
87      * S[]和ARGV[]以及整个脚本什么意思了

```

```

81     * 如果你仔细看lua脚本，就会明白：加锁成功时返回ttlRemaining=null，加锁失败时
82     * 返回ttlRemaining=xxx（上一个锁还剩多少时间）
83     *
84     * 另外，我们自定义的Redis分布式锁采用了IdUtil生成节点id，和getLockName(thre
85     adId)本质是一样的
86     */
87     return evalWriteAsync(getName(), LongCodec.INSTANCE, command,
88         "if (redis.call('exists', KEYS[1]) == 0) then " +
89             "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
90             "redis.call('pexpire', KEYS[1], ARGV[1]); " +
91             "return nil; " +
92             "end; " +
93             "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) th
en " +
94             "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
95             "redis.call('pexpire', KEYS[1], ARGV[1]); " +
96             "return nil; " +
97             "end; " +
98             "return redis.call('pttl', KEYS[1]);",
99             Collections.singletonList(getName()), internalLockLeaseTime,
100            getLockName(threadId));
101    }
102
103    // 向Redis服务器发送脚本并返回RFuture，大家可以近似看成：往线程池提交一个任务，然后将
104    // 异步结果封装到CompletableFuture
105    protected <T> RFuture<T> evalWriteAsync(String key, Codec codec, RedisCom
mand<T> evalCommandType, String script, List<Object> keys, Object... para
ms) {
106        CommandBatchService executorService = createCommandBatchService();
107        RFuture<T> result = executorService.evalWriteAsync(key, codec, evalCo
mmandType, script, keys, params);
108        if (!(commandExecutor instanceof CommandBatchService)) {
109            executorService.executeAsync();
110        }
111        return result;
112    }

```

示意图：



整个流程比较简单，只有两个难点：

- lua脚本写了啥
- ttlRemainingFuture.onComplete()有什么作用

lua脚本解读

大家可以通过evalWriteAsync()的参数列表推导出KEYS、ARGV分别是什么：

KEYS[] => Collections.singletonList(getName())

ARGV[] => internalLockLeaseTime, getLockName(threadId)

```
1 -- 如果不存在锁: "bravo1988_distributed_lock"
2 if (redis.call('exists', KEYS[1]) == 0) then
3     -- 使用hincrby设置锁: hincrby bravo1988_distributed_lock a1b2c3d4:666 1
4     redis.call('hincrby', KEYS[1], ARGV[2], 1);
5     -- 设置过期时间。ARGV[1]==internalLockLeaseTime
6     redis.call('pexpire', KEYS[1], ARGV[1]);
7     -- 返回null
8     return nil;
9 end;
10
11 -- 如果当前节点已经设置"bravo1988_distributed_lock" (注意, 传了ARGV[2]==节点id)
12 if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then
13     -- 就COUNT++, 可重入锁
14     redis.call('hincrby', KEYS[1], ARGV[2], 1);
15     -- 设置过期时间。ARGV[1]==internalLockLeaseTime
16     redis.call('pexpire', KEYS[1], ARGV[1]);
17     -- 返回null
18     return nil;
19 end;
20
21 -- 已经存在锁, 且不是当前节点设置的, 就返回锁的过期时间ttl
22 return redis.call('pttl', KEYS[1]);
```

总的来说，Redisson设计的分布式锁是采用hash结构：

LOCK_NAME (锁的KEY) + **CLIENT_ID** (节点ID) + **COUNT** (重入次数)



回调函数的作用

之前我们已经学过CompletableFuture的回调机制：

RFuture#onComplete()和它很相似：



```
Java |
```

```
1 - ttlRemainingFuture.onComplete((ttlRemaining, e) -> {
2     // 发生异常时直接return
3     if (e != null) {
4         return;
5     }
6
7     // 说明加锁成功
8     if (ttlRemaining == null) {
9         // 启动额外的线程，按照一定规则给当前锁续期
10        scheduleExpirationRenewal(threadId);
11    }
12});
```

onComplete()应该也是把回调函数推到stack中，方便后面异步线程弹栈执行。

至此，我们已经解决了之前的两个问题：

- lua脚本是什么意思（见注释）
- ttlRemainingFuture.onComplete()有什么作用（设置回调函数，等会儿会有线程调用）

虽然在CompletableFuture中已经强调过，这里还是要提一下：**被回调的不是onComplete(BiConsumer)，而是BiConsumer#accept()**。主线程在调用onComplete(BiConsumer)时把它作为参数传入，然后被推入栈中：

```

1 BiConsumer<Object, Object> consumer = (ttlRemaining, e) -> {
2     // 发生异常时直接return
3     if (e != null) {
4         return;
5     }
6
7     // 说明加锁成功
8     if (ttlRemaining == null) {
9         // 启动额外的线程, 按照一定规则给当前锁续期
10        scheduleExpirationRenewal(threadId);
11    }
12}

```

Redisson异步回调机制

现在已经确定了尝试加锁后会返回RFuture，并且我们可以通过RFuture做两件事：

- 通过RFuture获取ttlRemaining，也就是上一个锁的过期时间，如果为null则本次加锁成功，否则加锁失败，需要等待
- 通过RFuture设置回调函数

现在疑问是：

- 异步线程是谁，哪来的？
- onComplete()设置的回调函数是干嘛的？
- 回调时的参数(ttlRemaining, e)哪来的？

1、3两个问题非常难，源码比较绕，这里就带大家感性地体验一下，有兴趣可以自己跟源码了解。清除刚才的全部断点，只留下：

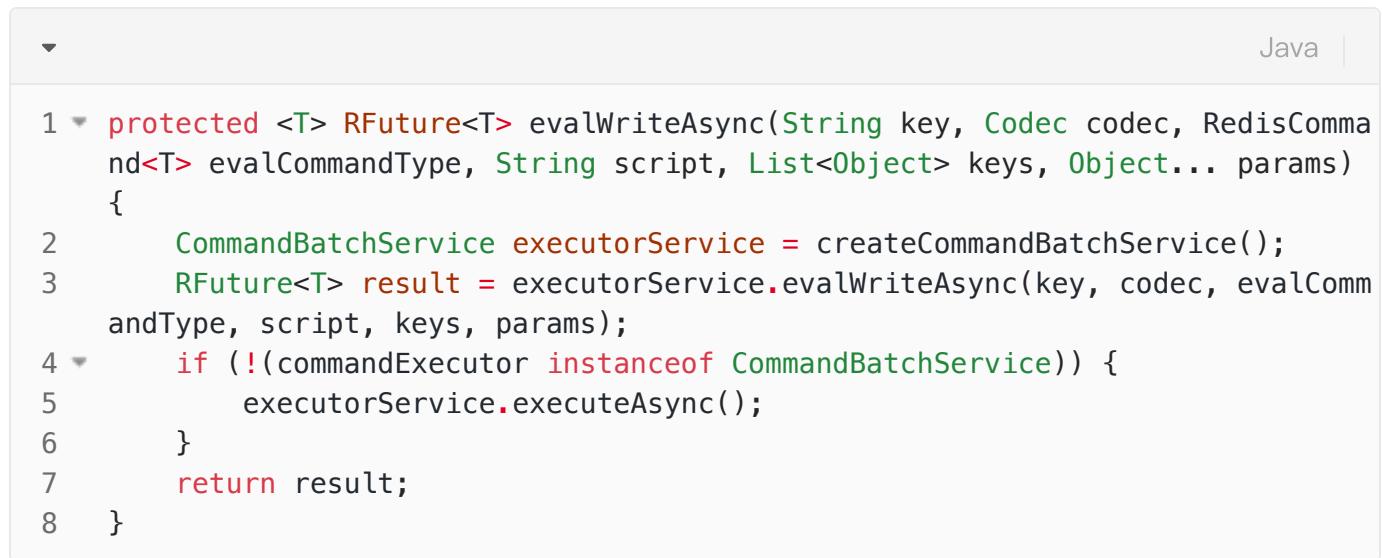


再次DEBUG，线程会先到达return ttlRemainingFuture，随后回调BiConsumer#accept()：



回调时线程变了：

大家有兴趣可以自己顺着调用栈逆推回去，还是比较复杂的，涉及到NIO、Promise等，源头还是在线程池，但其中又设计了Listeners的收集和循环唤醒：



```
Java
```

```
1 protected <T> RFuture<T> evalWriteAsync(String key, Codec codec, RedisCommandType evalCommandType, String script, List<Object> keys, Object... params) {
2     CommandBatchService executorService = createCommandBatchService();
3     RFuture<T> result = executorService.evalWriteAsync(key, codec, evalCommandType, script, keys, params);
4     if (!(commandExecutor instanceof CommandBatchService)) {
5         executorService.executeAsync();
6     }
7     return result;
8 }
```

总之，目前为止我们只需要知道：



```
我们虽然不知道onComplete()具体如何实现回调（比CompletableFuture复杂得多），但是我们知道锁续期和RFuture的回调机制相关！
```

Redisson如何实现锁续期

最终会进入：

```

1  private void renewExpiration() {
2      ExpirationEntry ee = EXPIRATION_RENEWAL_MAP.get(getEntryName());
3      if (ee == null) {
4          return;
5      }
6
7      /**
8       * 启动一个定时器: Timeout newTimeout(TimerTask task, long delay, TimeUnit unit);
9       * 执行规则是: 延迟internalLockLeaseTime/3后执行
10      * 注意啊, 每一个定时任务只执行一遍, 而且是延迟执行。
11      *
12      * 那么问题就来了:
13      * 1. internalLockLeaseTime/3是多久呢?
14      * 2. 如果定时任务只执行一遍, 似乎解决不了问题啊, 本质上和我们手动设置过期时间一样:
15      * 多久合适呢?
16      */
17      Timeout task = commandExecutor.getConnectionManager().newTimeout(new TimerTask() {
18          @Override
19          public void run(Timeout timeout) throws Exception {
20              ExpirationEntry ent = EXPIRATION_RENEWAL_MAP.get(getEntryName());
21              if (ent == null) {
22                  return;
23              }
24              Long threadId = ent.getFirstThreadId();
25              if (threadId == null) {
26                  return;
27
28              // 定时任务的目的是: 重新执行一遍lua脚本, 完成锁续期, 把锁的ttl拨回到30s
29              RFuture<Boolean> future = renewExpirationAsync(threadId);
30              // 设置了一个回调
31              future.onComplete((res, e) -> {
32                  if (e != null) {
33                      log.error("Can't update lock " + getName() + " expiration", e);
34                      // 如果宕机了, 就不会续期了
35                      return;
36                  }
37                  // 如果锁还存在 (没有unLock, 说明业务还没结束), 递归调用当前方法,
38                  // 不断续期
39                  if (res) {
36                      // reschedule itself

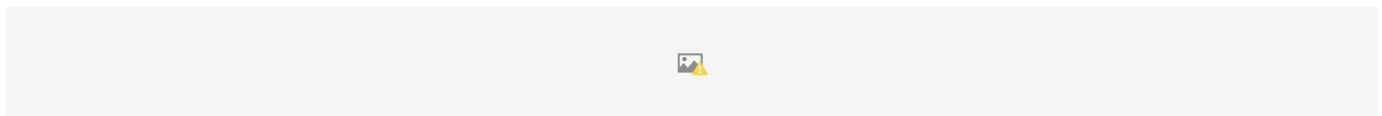
```

```

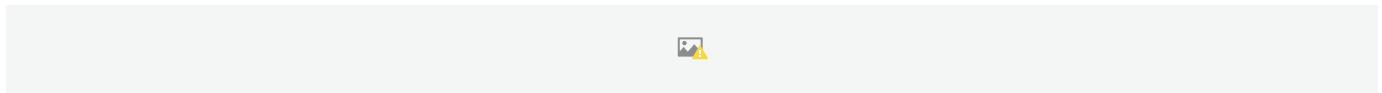
40                     renewExpiration();
41                 }
42             });
43         }
44     }, internalLockLeaseTime / 3, TimeUnit.MILLISECONDS);
45
46     ee.setTimeout(task);
47 }
48
49
50 /**
51 * 重新执行evalWriteAsync(), 和加锁时的lua脚本比较类似, 但有点不同
52 * 这里设置expire的参数也是internalLockLeaseTime
53 *
54 * 看来我们不得不去调查一下internalLockLeaseTime了!
55 */
56 protected RFuture<Boolean> renewExpirationAsync(long threadId) {
57     return evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands.EVA
58 L_BOOLEAN,
59     "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
60     "redis.call('pexpire', KEYS[1], ARGV[1]); " +
61     "return 1; " +
62     "end; " +
63     "return 0;",
64     Collections.singletonList(getName()),
65     internalLockLeaseTime, getLockName(threadId));
}

```

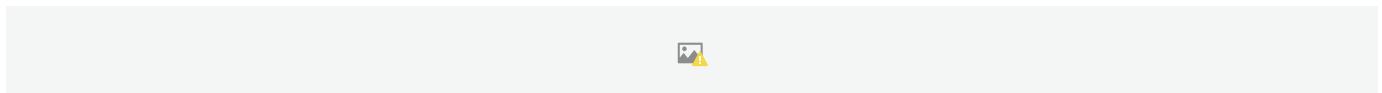
如果你给renewExpirationAsync()打上断点, 会发现每隔10秒, 定时任务就会执行一遍:



联想到定时任务的delay是internalLockLeaseTime/3, 所以推测internalLockLeaseTime为30秒。
点击internalLockLeaseTime, 很容易跳转到对应的字段:



再顺着getLockWatchdogTimeout()跳转, 很快就会发现





确实是30秒。

梳理一下所谓的Watchdog锁续期机制：

- lock()第一次成功加锁时，设置的锁过期时间默认30秒，这个值来自Watchdog变量

```

1 // 重点
2 -> private <T> RFuture<Long> tryAcquireAsync(long waitTime=-1, long leaseTime
   =-1, TimeUnit unit=null, long threadId=666) {
3
4     // lock()默认leaseTime=-1, 所以会跳过if
5 ->     if (leaseTime != -1) {
6         return tryLockInnerAsync(waitTime, leaseTime, unit, threadId, RedisCommands.EVAL_LONG);
7     }
8
9     // 执行lua脚本加锁, 返回RFuture。第二个参数就是leaseTime, 来自LockWatchdogTi
10    meout! ! !
11    RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(
12                                waitTime=-1,
13                                commandExecutor.getConnectionM
14                                anager().getCfg().getLockWatchdogTimeout()=30秒,
15                                TimeUnit.MILLISECONDS,
16                                threadId=666,
17                                RedisCommands.EVAL_LONG);
18
19    // 设置回调方法
20 ->    ttlRemainingFuture.onComplete((ttlRemaining, e) -> {
21        // 发生异常时直接return
22        if (e != null) {
23            return;
24        }
25        // 说明加锁成功
26        if (ttlRemaining == null) {
27            // 启动额外的线程, 按照一定规则给当前锁续期
28            scheduleExpirationRenewal(threadId);
29        }
30    });
31
32    // 返回RFuture, 里面有ttlRemaining
33    return ttlRemainingFuture;
34 }
35
36 -> <T> RFuture<T> tryLockInnerAsync(long waitTime=-1, long leaseTime=30*1000,
37   TimeUnit unit=毫秒, long threadId=666, RedisStrictCommand<T> command) {
38     // 略...
39 }
```

- `onComplete()`设置回调，等Redis调用回来后，异步线程回调`BiConsumer#accept()`，进入`scheduleExpirationRenewal(threadId)`，开始每隔`internalLockLeaseTime/3`时间就给锁续期



和加锁一样，执行lua脚本其实很快，所以这里的`future.onComplete()`虽说是异步，但很快就会被调用，然后就会递归调用`renewExpiration()`，然后又是一个`TimerTask()`，隔`internalLockLeaseTime/3`后又给锁续期。

也就是说，Redisson的Watchdog定时任务虽然只延迟执行一次，但每次调用都会递归，所以相当于：重复延迟执行。

还记得之前学习CompletableFuture时我写的一行注释吗：



也就是说，只要主线程的任务不结束，就会一直给锁续期。

锁释放有两种情况：

- 任务结束，主动`unLock()`删除锁

```
▼ Java  
1 redisson.lock();  
2 task();  
3 redisson.unLock();
```

- 任务结束，不调用`unLock()`，但由于守护线程已经结束，不会有后台线程继续给锁续期，过了30秒自动过期

上面我们探讨的都是加锁成功的流程，直接`ttl=null`就返回了，后面一大块都是加锁失败时的判断逻辑，其中涉及到：

- `while(true)`死循环
- 阻塞等待
- 释放锁时Redis的Publish通知（在后面的`unLock`流程会看到）
- 其他节点收到锁释放的信号后重新争抢锁



整个过程还是非常复杂的，大家有精力可以自行百度了解，后面介绍unLock()时也会涉及一部分加锁失败相关内容。

unLock()源码解析

有了lock()的经验，unLock()就简单多了：



相信大家还是能推断出KEYS[]和ARGV[], 这里就直接给出答案了：

```

1  -- 参数解释:
2  -- KEYS[1] => "bravo1988_distributed_lock"
3  -- KEYS[2] => getChannelName()
4  -- ARGV[1] => LockPubSub.UNLOCK_MESSAGE
5  -- ARGV[2] => internalLockLeaseTime
6  -- ARGV[3] => getLockName(threadId)
7
8  -- 锁已经不存在, 返回null
9  if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then
10    return nil;
11  end;
12
13  -- 锁还存在, 执行COUNT-- (重入锁的反向操作)
14  local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1);
15
16  -- COUNT--后仍然大于0 (之前可能重入了多次)
17  if (counter > 0) then
18    -- 设置过期时间
19    redis.call('pexpire', KEYS[1], ARGV[2]);
20    return 0;
21  -- COUNT--后小于等于0, 删除锁, 并向对应的Channel发送消息 (NIO), 消息类型是LockPubS
22  else
23    redis.call('del', KEYS[1]);
24    redis.call('publish', KEYS[2], ARGV[1]);
25    return 1;
26  end;
27
28  return nil;

```

也就是说, 当一个锁被释放时, 原先持有锁的节点会通过NIO的Channel发送 LockPubSub.UNLOCK_MESSAGE, 告诉其他订阅的Client: 我已经释放锁啦, 快来抢啊! 此时原本阻塞的其他节点就会重新竞争锁。

而所谓重入和反重入，简单来说就是：

Java

```
1 // 加锁三次
2 redisson.lock();
3 redisson.lock();
4 redisson.lock();
5 // 执行业务
6 executeTask();
7 // 相应的，就要解锁三次
8 redisson.unLock();
9 redisson.unLock();
10 redisson.unLock();
```

实际开发不会这样调用，但有时会出现子父类方法调用或者同一个线程反复调用使用同一把锁的多个方法，就会发生锁的重入（COUNT++），而当这些方法执行完毕逐个弹栈的过程中就会逐个unLock()解锁（COUNT--）。

lock(leaseTime, unit): 自定义过期时间、且不续期

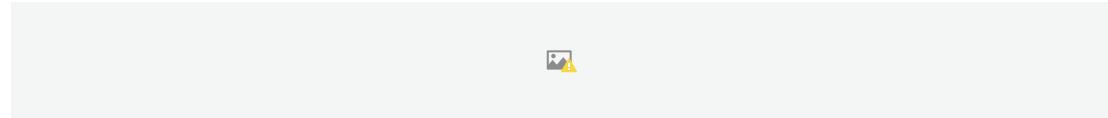
lock()默认会开启定时任务对锁进行续期，但Redisson还提供了另一个lock方法：

Java

```
1 redisson.lock();
2 redisson.lock(-1, null);
```

这两种写法其实一样。

当然了，通常会传入有意义的leaseTime：



这种写法除了更改了锁的默认ttl时间外，还阉割了锁续期功能。也就是说，10秒后如果任务还没执行完，就会和我们手写的Redis分布式锁一样，自动释放锁。

为什么锁续期的功能失效了呢？留给大家自己解答，这里只给出参考答案：

```
1 // 重点
2 - private <T> RFuture<Long> tryAcquireAsync(long waitTime=-1, long leaseTime
3 =-1, TimeUnit unit=null, long threadId=666) {
4
5     // lock()默认leaseTime=-1, 会跳过这个if执行后面的代码。但如果是lock(10, Time
6     Unit.SECONDS), 会执行if并跳过后面的代码。
7     if (leaseTime != -1) {
8         // 其实和下面的tryLockInnerAsync()除了时间不一样外, 没什么差别
9         return tryLockInnerAsync(waitTime, leaseTime, unit, threadId, Redis
10    Commands.EVAL_LONG);
11    }
12
13    // 但由于上面直接return了, 所以下面的都不会执行! !
14    /*
15
16        RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(
17
18            waitTime=-1,
19            commandExecutor.getConnectionM
20        anager().getCfg().getLockWatchdogTimeout()=30秒,
21
22            TimeUnit.MILLISECONDS,
23            threadId=666,
24            RedisCommands.EVAL_LONG);
25
26
27    // 设置回调方法 (不会执行! ! )
28    ttlRemainingFuture.onComplete((ttlRemaining, e) -> {
29
30        // 发生异常时直接return
31        if (e != null) {
32            return;
33        }
34
35        // 说明加锁成功
36        if (ttlRemaining == null) {
37
38            // 启动额外的线程, 按照一定规则给当前锁续期
39            scheduleExpirationRenewal(threadId);
40
41        }
42    });
43
44
45    // 不会执行! !
46    return ttlRemainingFuture;
47
48    */
49
50 }
```

```
42     <T> RFuture<T> tryLockInnerAsync(long waitTime=-1, long leaseTime=30*1000,  
43         TimeUnit unit=毫秒, long threadId=666, RedisStrictCommand<T> command) {  
44         // 略...  
45     }
```

也就是说，直接执行lua加锁就返回了，没有机会启动定时任务和递归...

tryLock()系列：让调用者自行决定加锁失败后的操作

之前我们已经观察到，如果多个节点都调用lock()，那么没获取到锁的节点线程会阻塞，直到原先持有锁的节点删除锁并publish LockPubSub.UNLOCK_MESSAGE。

但如果调用者不希望阻塞呢？他有可能想着：如果加锁失败，我就直接放弃。

是啊，毕竟尝试加锁的目的可能完全相反：

- 在保证线程安全的前提下，尽量让所有线程都执行成功
- 在保证线程安全的前提下，只让一个线程执行成功

前者适用于秒杀、下单等操作，希望尽最大努力达成；后者适用于定时任务，只要让一个节点去执行，没有获取锁的节点应该fast-fail（快速失败）。

也就是说，节点获锁失败后，理论上可以有各种各样的处理方式：

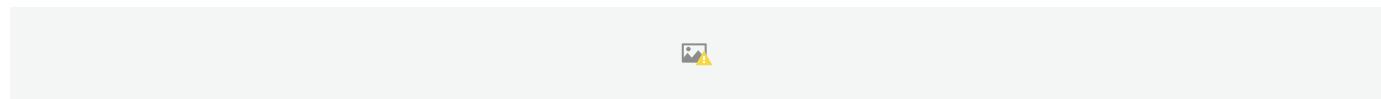
- 阻塞等待
- 直接放弃
- 试N次再放弃
- ...

但lock、lock(leaseTime, timeUnit)替我们写死了：阻塞等待。即使lock(leaseTime, unit)，其实也是阻塞等待，只不过不会像lock()一样不断续期。

究其原因，主要是lock()这些方法对于加锁失败的判断是在内部写死的：



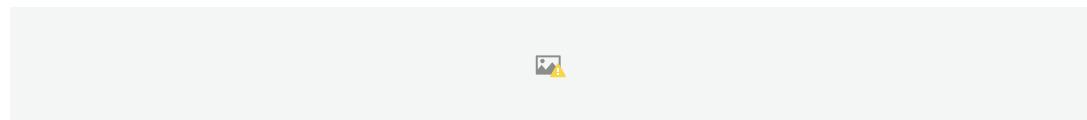
而tryLock()方法则去掉了这层中间判断，把结果直接呈递到调用者面前，让调用者自己决定加锁失败后如何处理：



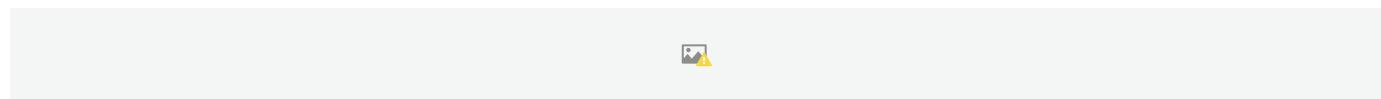
tryLock()直接返回true（加锁成功）和false（加锁失败），后续如何处理，全凭各个节点自己做出决定。

```
1  @Test
2  public void testTryLock() {
3      RLock lock = redissonClient.getLock("bravo1988_distributed_lock");
4      boolean b = lock.tryLock();
5      if (b) {
6          // 业务操作...
7      }
8
9      // 调用立即结束，不阻塞
10 }
```

这样讲可能有点抽象，大家可以分别点进lock()和tryLock()，自行体会。总之，tryLock()中间少了一大块逻辑，因为它不插手结果的判断。



另外，tryLock()在加锁成功的情况下，其实和lock()是一样的，也会触发锁续期：



如果你不希望触发锁续期，可以像lock(leaseTime, unit)一样指定过期时间，还可以指定加锁失败后等待多久：

```

1  @Test
2  public void testLockSuccess() throws InterruptedException {
3      RLock lock = redissonClient.getLock("bravo1988_distributed_lock");
4      // 基本等同于lock(), 加锁成功也【会自动锁续期】，但获锁失败【立即返回false】，交
5      // 给调用者判断是否阻塞或放弃
6      lock.tryLock();
7      // 加锁成功仍然【会自动锁续期】，但获锁失败【会等待10秒】，看看这10秒内当前锁是否释
8      // 放，如果是否则尝试加锁
9      lock.tryLock(10, TimeUnit.SECONDS);
10     // 加锁成功【不会锁续期】，加锁失败【会等待10秒】，看看这10秒内当前锁是否释放，如果
11     // 是否则尝试加锁
12     lock.tryLock(10, 30, TimeUnit.SECONDS);
13 }

```

注意哈，只传两个参数时，那个time其实是传给waitTime的：



我们之前操作的都是leaseTime，此时还是-1，也就是说如果加锁成功，还是会锁续期。



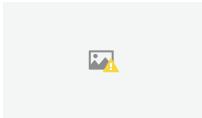
那waitTime是用来控制什么的呢？



简而言之：

- tryLock()加锁失败会立即返回false，而加了waitTime可以手动指定阻塞等待的时间（等一等，万一行呢）
- leaseTime的作用没变，控制的是加锁成功后要不要续期





至此，分布式锁章节暂时告一段段落。大家有兴趣的话，可以把上一篇花里胡哨的定时任务用Redisson改写，去掉Redis Message Queue（但定时任务最好还是用xxl-job等）。

Redisson的具体使用方法可以参考尚硅谷雷丰阳老师的讲解：

<https://www.bilibili.com/video/BV18a4y1L7nv?p=57>

Redisson分布式锁的缺陷

在哨兵模式或者主从模式下，如果master实例宕机，可能导致多个节点同时完成加锁。

以主从模式为例，由于所有的写操作都是先在master上进行，然后再同步给各个slave节点，所以master与各个slave节点之间的数据具有一定的延迟性。对于Redisson分布式锁而言，比如客户端刚对master写入Redisson锁，然后master异步复制给各个slave节点，但这个过程中master节点宕机了，其中一个slave节点经过选举变成了master节点，好巧不巧，这个slave还没同步到Redisson锁，所以其他客户端可能再次加锁。

具体情况，大家可以百度看看，解决方案也比较多。

还是那句话，但凡涉及到分布式，都没那么简单。有时引入一个解决方案后，我们不得不面对另一个问题。

| 来自：浅谈Redis分布式锁(下)

浅谈Redis分布式锁(中)

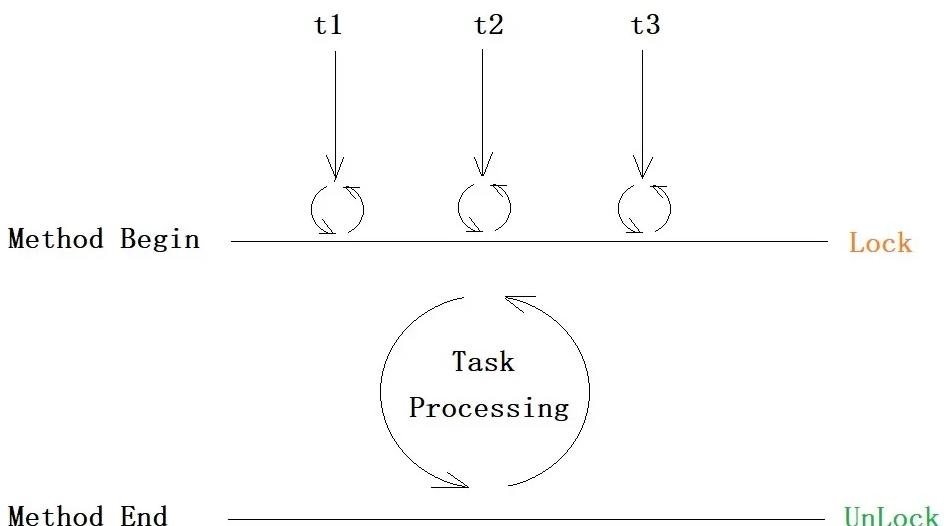
我们在不久前介绍了SpringBoot定时任务，最近又一起探究了如何使用Redis实现简单的消息队列，都是一些不错的小知识点。为了能跟前面的内容产生联动，这次我们打算把Redis分布式锁相关的介绍融合进定时任务的案例中，学起来更带劲~

Demo构思

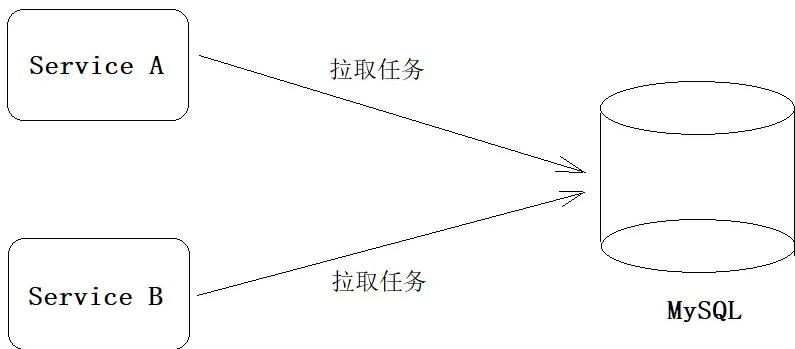
在我看来，同样需要使用锁，动机可能完全相反：

- 在保证线程安全的前提下，尽量让所有线程都执行成功
- 在保证线程安全的前提下，只让一个线程执行成功

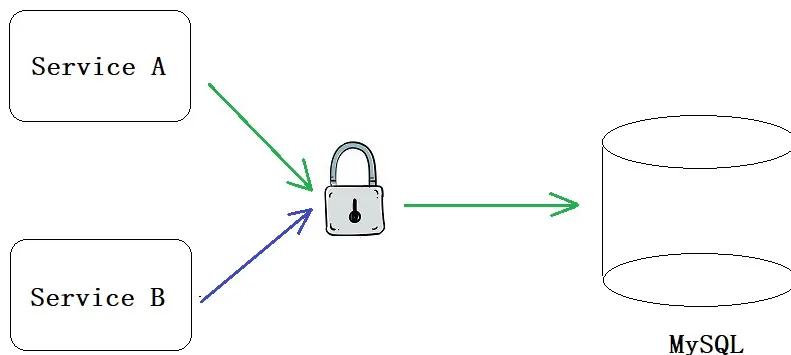
前者适用于秒杀等场景。作为商家，当然希望在不发生线程安全问题的前提下，让每一个订单都生效，直到商品售罄。此时分布式锁的写法可以是“不断重试”或“阻塞等待”，即：递归或while true循环尝试获取、阻塞等待。



而后者适用于分布式系统或多节点项目的定时任务，比如同一份代码部署在A、B两台服务器上，而数据库共用同一个。如果不做限制，那么在同一时刻，两台服务器都会去拉取列表执行，会发生任务重复执行的情况。

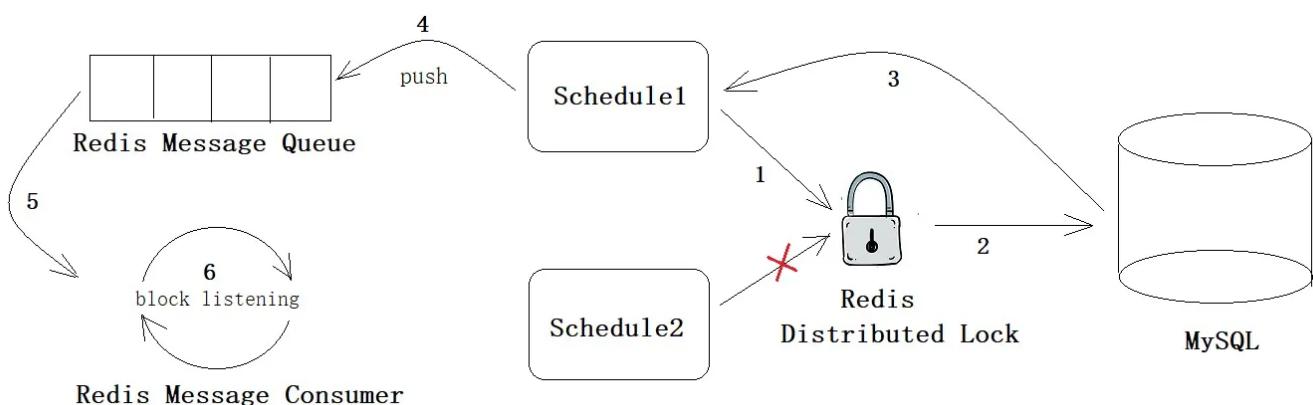


此时可以考虑使用分布式锁，在cron触发的时刻只允许一个线程去往数据库拉取任务：

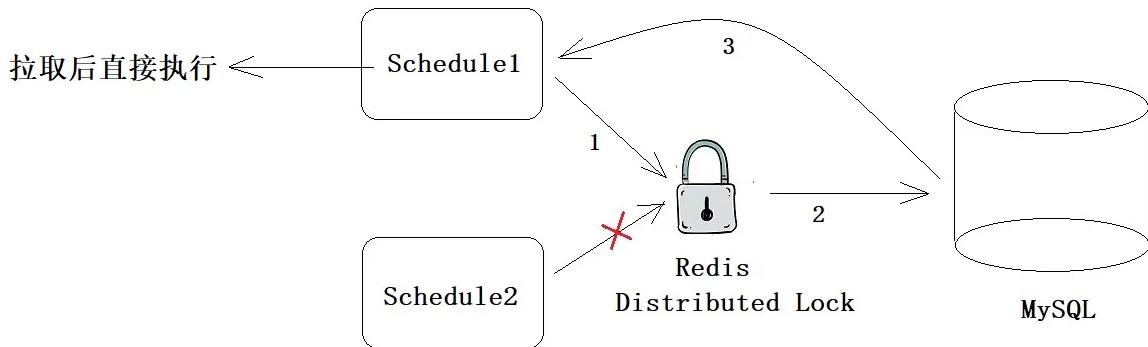


在实现Redis分布式锁控制定时任务唯一性的同时，我们引入之前的Redis消息队列。注意，这与Redis分布式锁本身无关，就是顺便复习一遍Redis消息队列而已，大家可以只实现Redis分布式锁+定时任务的部分。

整个Demo的结构大致如图：



当然，实际项目中一般是这样的：



分布式锁为什么难设计？

首先，要和大家说一下，但凡牵涉到分布式的处理，没有一个是简单的，上面的Demo设计也不过是玩具，用来启发 大家的思路。

为什么要把Demo设计得这么复杂呢？哈哈，因为这是我在上一家公司自己设计的，遇到了很多坑...拿出来自嘲一番，与各位共勉。

我当时的设计思路是：

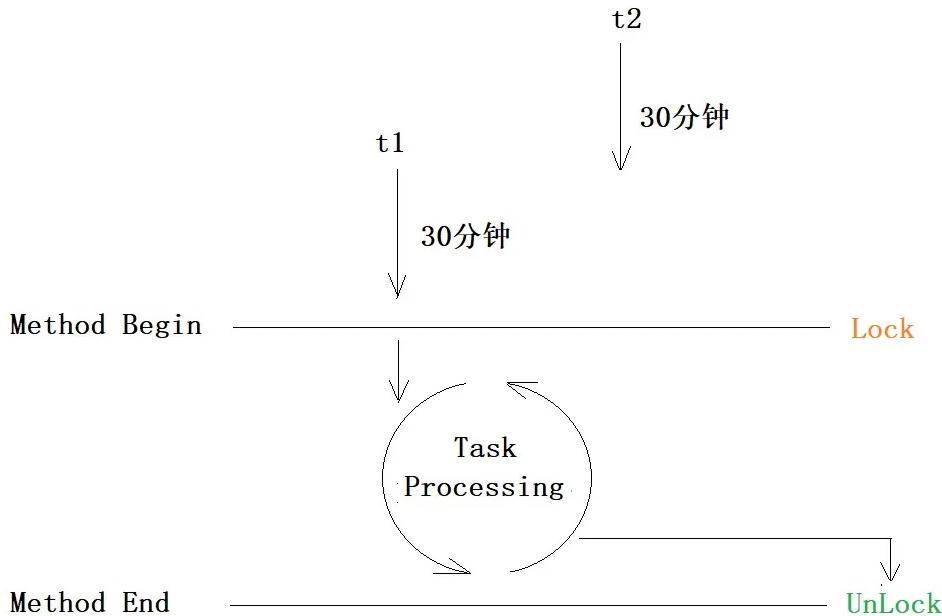
由于小公司没有用什么Elastic-Job啥的，就是很普通的多节点部署。为了避免任务重复执行，我想设计一个分布式锁。但因为当时根本不知道Redisson，所以就自己百度了Redis实现分布式锁的方式，然后依葫芦画瓢自己手写了一个。

但我写完Redis分布式锁后，在实际测试过程中发现还需要考虑锁的失效时间...

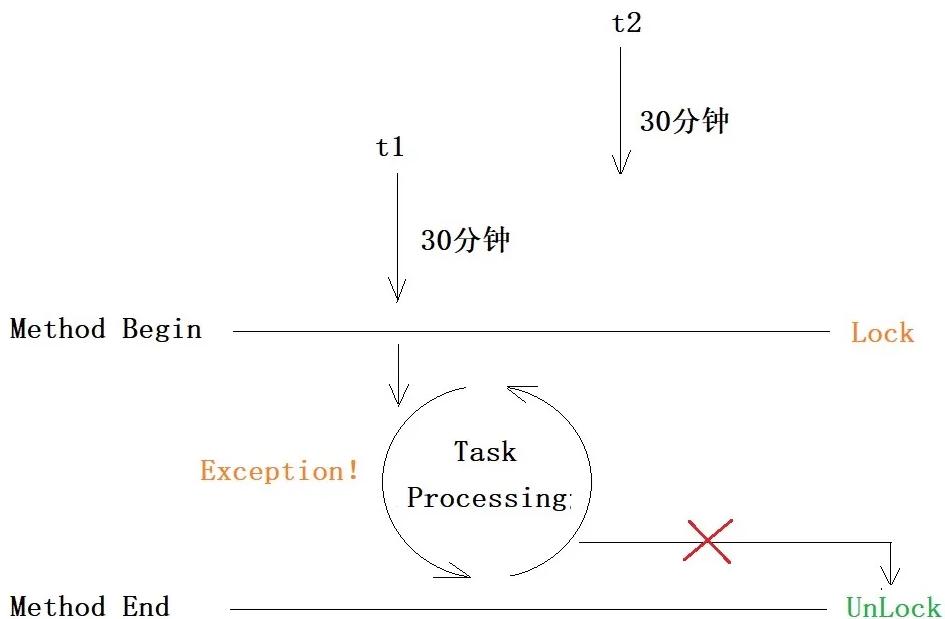
这里有两个问题：

- 为什么要设置锁的过期时间？
- 锁的过期时间设置多久合适？

最简单的实现方案是这样的，一般没问题：



但极端的情况下（项目在任务进行时重启或意外宕机），可能当前任务来不及解锁就挂了（死锁），那么下一个任务就会一直被锁在方法外等待。就好比厕所里有人被熏晕了，没法开门，而外面的人又进不去…



此时需要装一个自动解锁的门，到时间自动开门，也就是要给锁设置一个过期时间。但紧接着又会有第二个问题：锁的失效时间设多长合适？

很难定。

因为随着项目的发展，定时任务的执行时间很可能是变化的。

如果设置时间过长，极端点，定为365天。假设任务正常执行，比如10分钟就结束，那么线程继续往下就会执行unLock()主动解锁。但万一和上面一样宕机了，那么这个锁就要等365天后才解开。注意，宕机可不像JVM异常，它压根不会去执行finally里的unLock()。这种情况好比有个人在厕所里上大号直接掉坑里了，而自动门默认365天打开...所以，锁过期时间设置过长的坏处，本质是一旦发生宕机来不及解锁，那么过期时间越长，影响面越广，会导致其他操作阻滞。

如果设置时间过短，上一个人还没拉完，门就“咔嚓”一声开了，尴尬不，重复执行了。

综上所述，我当时之所以设计得这么复杂，就是想尽量缩短任务执行的时间，让它尽可能短（拉取后直接丢给队列，自己不处理），这样锁的时间一般设置30分钟就没啥问题。另外，对于死锁问题，我当时没有考虑宕机的情况，只考虑了意外重启...问题还有很多，文末会再总结。

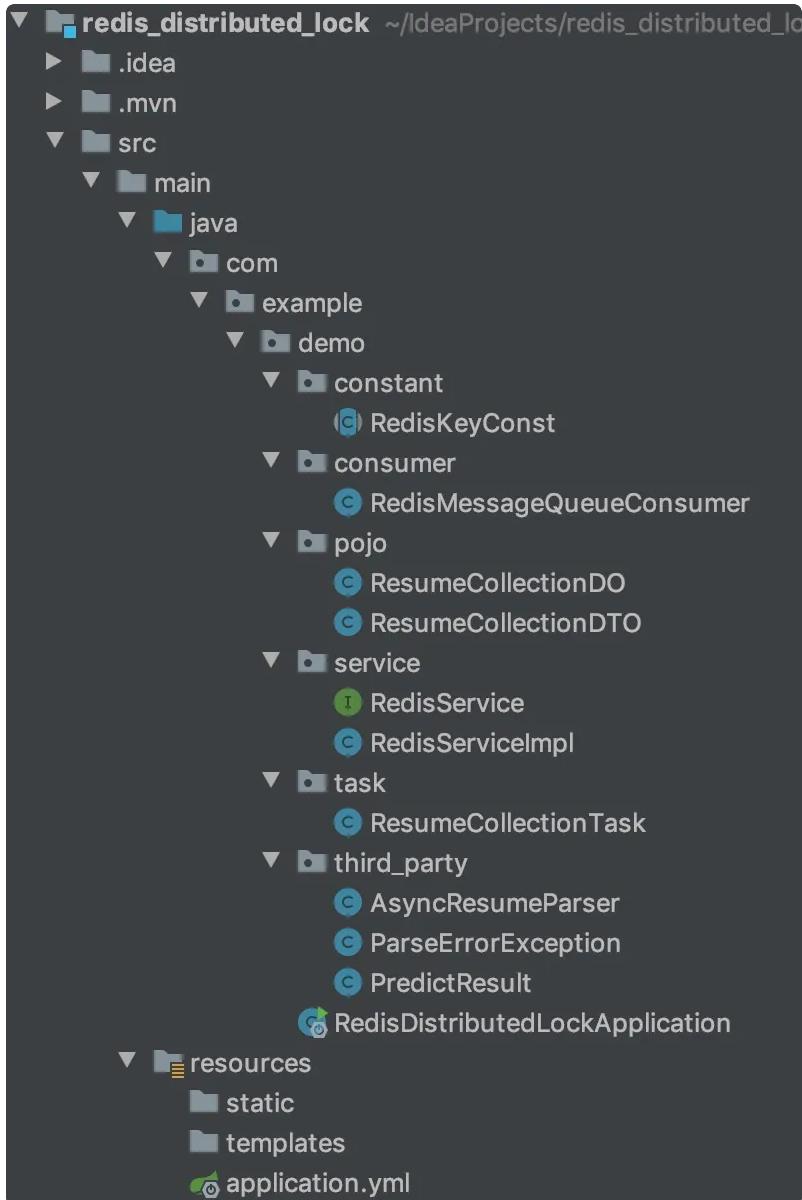
请大家阅读下面代码时思考两个问题：

- Demo如何处理锁的过期时间
- Demo如何防止死锁

项目搭建

新建一个空的SpringBoot项目。

拷贝下方代码，构建工程：



构建完以后，拷贝一份，修改端口号为8081，避免和原先的冲突



统一管理Redis Key： RedisKeyConst

```
1   */
2  * 统一管理Redis Key
3  *
4  * @author qiyu
5  */
6  public final class RedisKeyConst {
7   */
8  * 分布式锁的KEY
9  */
10 public static final String RESUME_PULL_TASK_LOCK = "resume_pull_task_lock";
11  */
12 * 简历异步解析任务队列
13 */
14 public static final String RESUME_PARSE_TASK_QUEUE = "resume_parse_task_queue";
15 }
```

Redis消息队列：RedisMessageQueueConsumer


```

74                     retryCount = 0;
75                     rePullCount = 0;
76                     break;
77                 }
78             } catch (InterruptedException e) {
79                 discardTask(resumeCollectionDTO);
80                 log.info("<<<<<<<<<<任务中断
81             异常，简历:{}", resumeCollectionDTO.getName());
82                     rePullCount = 0;
83                     retryCount = 0;
84                     break;
85                 }
86             } catch (Exception e) {
87                 if (retryCount > 3) {
88                     discardTask(resumeCollectionDTO);
89                     log.info("<<<<<<<<<<简历:{}重试{}次后放弃，rePullCount:{}，retryCount:{}",
90                         resumeCollectionDTO.getName(), r
91                         etryCount, rePullCount, retryCount);
92                     rePullCount = 0;
93                     retryCount = 0;
94                     break;
95                 }
96                 retryCount++;
97                 log.info("简历:{}远程调用异常，准备进行第{}次重
98                 试....", resumeCollectionDTO.getName(), retryCount);
99             }
100         }
101     });
102 }
103
104 private void discardTask(ResumeCollectionDTO task) {
105     // 根据asyncPredictId删除任务...
106     log.info("丢弃任务:{}...", task.getName());
107 }
108
109 }
```

实体类：DO+DTO

```
1  @Data
2  @NoArgsConstructor
3  @AllArgsConstructor
4  public class ResumeCollectionDO {
5      /**
6       * 简历id
7       */
8      private Long id;
9      /**
10     * 简历名称
11     */
12     private String name;
13 }
```

```
1  @Data
2  @NoArgsConstructor
3  @AllArgsConstructor
4  public class ResumeCollectionDTO implements Serializable {
5      /**
6       * 简历id
7       */
8      private Long id;
9      /**
10     * 异步解析id，稍后根据id可获取最终解析结果
11     */
12     private Long asyncPredictId;
13      /**
14       * 简历名称
15       */
16     private String name;
17 }
```

分布式锁：RedisService

```
1  public interface RedisService {  
2  
3      /**  
4          * 向队列插入消息  
5          *  
6          * @param queue 自定义队列名称  
7          * @param obj    要存入的消息  
8          */  
9      void pushQueue(String queue, Object obj);  
10  
11     /**  
12         * 从队列取出消息  
13         *  
14         * @param queue    自定义队列名称  
15         * @param timeout  最长阻塞等待时间  
16         * @param timeUnit 时间单位  
17         * @return  
18         */  
19     Object popQueue(String queue, long timeout, TimeUnit timeUnit);  
20  
21     /**  
22         * 尝试上锁  
23         *  
24         * @param lockKey  
25         * @param value  
26         * @param expireTime  
27         * @param timeUnit  
28         * @return  
29         */  
30     boolean tryLock(String lockKey, String value, long expireTime, TimeUnit timeUnit);  
31  
32     /**  
33         * 根据MACHINE_ID解锁（只能解自己的）  
34         *  
35         * @param lockKey  
36         * @param value  
37         * @return  
38         */  
39     boolean unLock(String lockKey, String value);  
40  
41     /**  
42         * 释放锁，不管是自己的  
43         *  
44         * @param lockKey
```

```
45     * @param value
46     * @return
47     */
48     boolean releaseLock(String lockKey, String value);
49 }
50 }
```

```
1  @Slf4j
2  @Component
3  public class RedisServiceImpl implements RedisService {
4
5      @Autowired
6      private RedisTemplate redisTemplate;
7
8      /**
9       * 向队列插入消息
10      *
11      * @param queue 自定义队列名称
12      * @param obj    要存入的消息
13      */
14     @Override
15     public void pushQueue(String queue, Object obj) {
16         redisTemplate.opsForList().leftPush(queue, obj);
17     }
18
19     /**
20      * 从队列取出消息
21      *
22      * @param queue    自定义队列名称
23      * @param timeout  最长阻塞等待时间
24      * @param timeUnit 时间单位
25      * @return
26      */
27     @Override
28     public Object popQueue(String queue, long timeout, TimeUnit timeUnit)
29     {
30         return redisTemplate.opsForList().rightPop(queue, timeout, timeUnit);
31     }
32     /**
33      * 尝试上锁
34      *
35      * @param lockKey
36      * @param value
37      * @param expireTime
38      * @param timeUnit
39      * @return
40      */
41     @Override
42     public boolean tryLock(String lockKey, String value, long expireTime,
        TimeUnit timeUnit) {
```

```
43     Boolean lock = redisTemplate.opsForValue().setIfAbsent(lockKey, va
44     lue);
45     if (Boolean.TRUE.equals(lock)) {
46         redisTemplate.expire(lockKey, expireTime, timeUnit);
47         return true;
48     } else {
49         return false;
50     }
51 }
52
53 /**
54 * 根据MACHINE_ID解锁（只能解自己的）
55 *
56 * @param lockKey
57 * @param value
58 * @return
59 */
60 @Override
61 public boolean unLock(String lockKey, String value) {
62     String machineId = (String) redisTemplate.opsForValue().get(lockKe
63     y);
64     if (StringUtils.isNotEmpty(machineId) && machineId.equals(value))
65     {
66         redisTemplate.delete(lockKey);
67         return true;
68     }
69 }
70 /**
71 * 释放锁，不管是自己的
72 *
73 * @param lockKey
74 * @param value
75 * @return
76 */
77 @Override
78 public boolean releaseLock(String lockKey, String value) {
79     Boolean delete = redisTemplate.delete(lockKey);
80     if (Boolean.TRUE.equals(delete)) {
81         log.info("Spring启动, 节点:{}成功释放上次简历汇聚定时任务锁", value)
82     ;
83         return true;
84     }
85     return false;
86 }
```

定时任务：ResumeCollectionTask

```

1  @Slf4j
2  @Component
3  @EnableScheduling
4  public class ResumeCollectionTask implements ApplicationListener<ContextRefreshedEvent> {
5
6      /**
7       * 当这份代码被部署到不同的服务器，启动时为每台机器分配一个唯一的机器ID
8       */
9      private static final String MACHINE_ID = IdUtil.randomUUID();
10
11     @Autowired
12     private RedisService redisService;
13     @Autowired
14     private AsyncResumeParser asyncResumeParser;
15
16     @Scheduled(cron = "0 */1 * * * ?")
17     //     @Scheduled(fixedDelay = 60 * 1000L)
18     public void resumeSchedule() {
19         // 尝试上锁，返回true或false，锁的过期时间设置为10分钟（实际要根据项目调整，这也是自己实现Redis分布式锁的难点之一）
20         boolean lock = redisService.tryLock(RedisKeyConst.RESUME_PULL_TASK_LOCK, MACHINE_ID, 10, TimeUnit.MINUTES);
21
22         // 如果当前节点成功获取锁，那么整个系统只允许当前程序去MySQL拉取待执行任务
23         if (lock) {
24             log.info("节点:{}获取锁成功，定时任务启动", MACHINE_ID);
25             try {
26                 collectResume();
27             } catch (Exception e) {
28                 log.info("定时任务异常:", e);
29             } finally {
30                 redisService.unLock(RedisKeyConst.RESUME_PULL_TASK_LOCK, M
31 ACHINE_ID);
32                 log.info("节点:{}释放锁，定时任务结束", MACHINE_ID);
33             }
34         } else {
35             log.info("节点:{}获取锁失败，放弃定时任务", MACHINE_ID);
36         }
37
38     /**
39      * 任务主体：
40      * 1.从数据库拉取符合条件的HR邮箱
41      * 2.从HR邮箱拉取附件简历

```

```
42     * 3. 调用远程服务异步解析简历
43     * 4. 插入待处理任务到数据库，作为记录留存
44     * 5. 把待处理任务的id丢到Redis Message Queue，让Consumer去异步处理
45     */
46     private void collectResume() throws InterruptedException {
47         // 跳过1、2两步，假设已经拉取到简历
48         log.info("节点:{}从数据库拉取任务简历", MACHINE_ID);
49         List<ResumeCollectionDO> resumeCollectionList = new ArrayList<>();
50         resumeCollectionList.add(new ResumeCollectionDO(1L, "张三的简历.pdf"));
51         resumeCollectionList.add(new ResumeCollectionDO(2L, "李四的简历.html"));
52         resumeCollectionList.add(new ResumeCollectionDO(3L, "王五的简历.doc"));
53         // 模拟数据库查询耗时
54         TimeUnit.SECONDS.sleep(3);
55
56         log.info("提交任务到消息队列:{}",
57             resumeCollectionList.stream().map(ResumeCollectionDO::getName).collect(Collectors.joining(",")));
58
59         for (ResumeCollectionDO resumeCollectionDO : resumeCollectionList) {
60             // 上传简历异步解析，得到异步结果id
61             Long asyncPredictId = asyncResumeParser.asyncParse(resumeCollectionDO);
62
63             // 把任务插入数据库
64             // 略...
65
66             // 把任务丢到Redis Message Queue
67             ResumeCollectionDTO resumeCollectionDTO = new ResumeCollectionDTO();
68             BeanUtils.copyProperties(resumeCollectionDO, resumeCollectionDTO);
69             resumeCollectionDTO.setAsyncPredictId(asyncPredictId);
70             redisService.pushQueue(RedisKeyConst.RESUME_PARSE_TASK_QUEUE,
71             resumeCollectionDTO);
72         }
73     }
74
75     /**
76      * 项目重启后先尝试删除之前的锁（如果存在），防止死锁等待
77      *
78      * @param event the event to respond to
79      */
80     @Override
```

```
81     public void onApplicationEvent(ContextRefreshedEvent event) {
82         redisService.releaseLock(RedisKeyConst.RESUME_PULL_TASK_LOCK, MACH
83         INE_ID);
84     }
85 }
```

模拟第三方服务（异步）

```
1  /**
2   * 第三方提供给的简历解析服务
3   *
4   * @author qiyu
5   */
6  @Service
7  public class AsyncResumeParser {
8
9      @Autowired
10     private ObjectMapper objectMapper;
11
12     /**
13     * 模拟分配异步任务结果id，不用深究，没啥意义，反正每个任务都会得到一个id，稍后根据
14     * id返回最终解析结果
15     */
16     private static final AtomicLong ASYNC_RESULT_ID = new AtomicLong(1000)
17     ;
18
19     /**
20     * 解析结果
21     */
22     private static final Map<Long, String> results = new HashMap<>();
23
24     /**
25     * 模拟第三方服务异步解析，返回解析结果
26     *
27     * @param resumeCollectionDO
28     * @return
29     */
30     public Long asyncParse(ResumeCollectionDO resumeCollectionDO) {
31         long asyncPredictId = ASYNC_RESULT_ID.getAndIncrement();
32         try {
33             String resultJson = objectMapper.writeValueAsString(resumeCollectionDO);
34             results.put(asyncPredictId, resultJson);
35             return asyncPredictId;
36         } catch (JsonProcessingException e) {
37             e.printStackTrace();
38         }
39     }
40
41     /**
42     * 根据异步id返回解析结果，但此时未必已经解析成功
43     * <p>
44     * 解析状态
```

```
43     * 0 初始化
44     * 1 处理中
45     * 2 调用成功
46     * 3 调用失败
47     *
48     * @param asyncPredictId
49     * @return
50     */
51     public PredictResult getResult(Long asyncPredictId) throws ParseErrorException, InterruptedException {
52         // 随机模拟异步解析的状态
53         int value = ThreadLocalRandom.current().nextInt(100);
54         if (value >= 85) {
55             // 模拟解析完成
56             TimeUnit.SECONDS.sleep(1);
57             String resultJson = results.get(asyncPredictId);
58             return new PredictResult(resultJson, 2);
59         } else if (value <= 5) {
56             // 模拟解析异常
57             TimeUnit.SECONDS.sleep(1);
58             throw new ParseErrorException("简易解析异常");
59         }
60         // 如果时间过短, 返回status=1, 表示解析中
61         TimeUnit.SECONDS.sleep(1);
62         return new PredictResult("", 1);
63     }
64 }
65 }
```

```
1  /**
2   * 解析异常
3   *
4   * @author qiyu
5   */
6  public class ParseErrorException extends Exception {
7      /**
8   * Constructs a new exception with {@code null} as its detail message.
9   * The cause is not initialized, and may subsequently be initialized b
y a
10  * call to {@link #initCause}.
11  */
12     public ParseErrorException() {
13  }
14
15     /**
16   * Constructs a new exception with the specified detail message. The
17   * cause is not initialized, and may subsequently be initialized by
18   * a call to {@link #initCause}.
19   *
20   * @param message the detail message. The detail message is saved for
21   *                 later retrieval by the {@link #getMessage()} method.
22   */
23     public ParseErrorException(String message) {
24     super(message);
25  }
26 }
```

```
1  /**
2   * 第三方返回值
3   *
4   * @author qiyu
5   */
6  @Data
7  @NoArgsConstructor
8  @AllArgsConstructor
9  public class PredictResult {
10     /**
11   * 解析结果
12   */
13   private String resultJson;
14     /**
15   * 解析状态
16   * 0 初始化
17   * 1 处理中
18   * 2 调用成功
19   * 3 调用失败
20   */
21   private Integer status;
22 }
```

模拟异常

在项目运行过程中，启动这个测试类的方法，即可观察不一样的现象。

```

1  @SpringBootTest
2  class RedisDistributedLockApplicationTests {
3
4      @Autowired
5      private RedisService redisService;
6
7      /**
8       * 作为失败案例（因为不存在777L这个解析任务，AsyncResumeParse.results会返回nu
9       * ll）
10      */
11     @Test
12     void contextLoads() {
13         ResumeCollectionDTO resumeCollectionDTO = new ResumeCollectionDTO(
14             );
15         resumeCollectionDTO.setId(666L);
16         resumeCollectionDTO.setAsyncPredictId(777L);
17         resumeCollectionDTO.setName("测试1号");
18
19         redisService.pushQueue(RedisKeyConst.RESUME_PARSE_TASK_QUEUE, resu
20         meCollectionDTO);
21
22     }

```

pom.xml

```

1  server:
2      port: 8080
3
4  spring:
5      redis:
6          host: # 参考小册《阿里云账号》
7          password: # 参考小册《阿里云账号》
8          database: 2

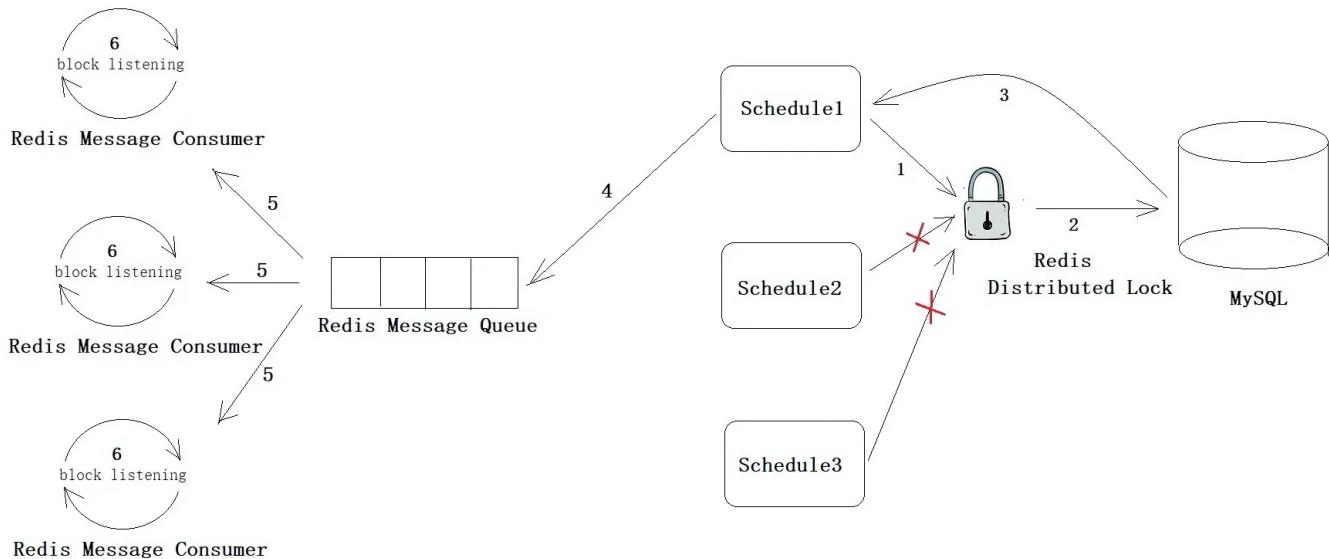
```

效果展示

啥都不说了，都在jiu代码里了。大家自己拷贝到本地，动手玩一下，加深对Redis锁和Redis消息队列的理解。

此处为语雀视频卡片，点击链接查看：[Kapture 2020-11-04 at 21.31.00.mp4](#)

只有一个定时任务能去数据库拉取任务，到时多节点部署大致是下面这样（redis一般是独立部署的，和节点代码无关）：



后话

上面展示的代码其实存在很多问题，我们会在下一篇指出并讨论解决方案。

本文仅提供思路，开阔大家的眼界，千万别在自己项目中使用！！！！我当年被这个坑惨了，花里胡哨的，尤其Consumer里一大堆的sleep()，是非常low的！！

对于异步调用的结果，不要循环等待，而应该分为几步：

1. 调用异步接口，得到异步结果唯一id
2. 将结果id保存到任务表中，作为一个任务
3. 启动定时任务，根据id拉取最终结果（如果还没有结果，跳过当前任务，等下一个定时任务处理）

分布式定时任务可以考虑xxl-job或elastic-job，分布式锁推荐使用[redisson](#)。

| 来自：浅谈Redis分布式锁(中)

浅谈Redis分布式锁(上)

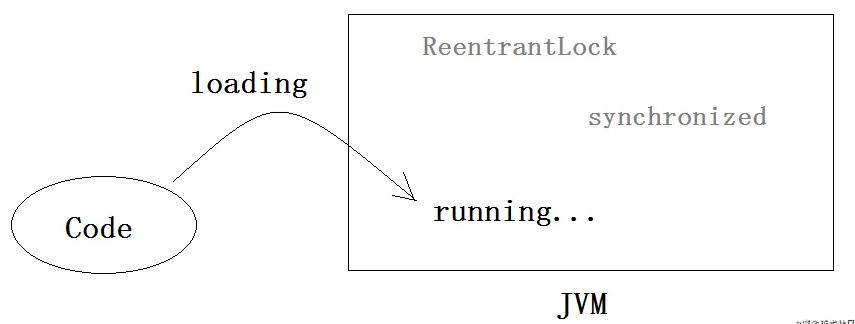
不论面试还是实际工作中，Redis都是避无可避的技术点。在我心里，MySQL和Redis是衡量一个程序员是否“小有所成”的两把标尺。如果他能熟练使用MySQL和Redis，以小化大，充分利用现有资源出色地完成当下需求，说明他已经成长了。

本篇文章我们一起来探讨Redis分布式锁相关的内容。

说到锁，大家第一时间想到的应该是synchronized关键字或ReentrantLock，随即想到偏向锁、自旋锁、重量级锁或者CAS甚至AQS。一般来说，我不喜欢一下子引入这么多概念，可能会把问题弄复杂，但为了方便大家理解Redis分布式锁，这里稍微提一下。

JVM锁

所谓JVM锁，其实指的是诸如synchronized关键字或者ReentrantLock实现的锁。之所以统称为JVM锁，是因为我们的项目其实都是跑在JVM上的。理论上每一个项目启动后，就对应一片JVM内存，后续运行时数据的生离死别都在这一片土地上。



什么是锁、怎么锁？

明白了“JVM锁”名字的由来，我们再来聊什么是“锁”，以及怎么“锁”。

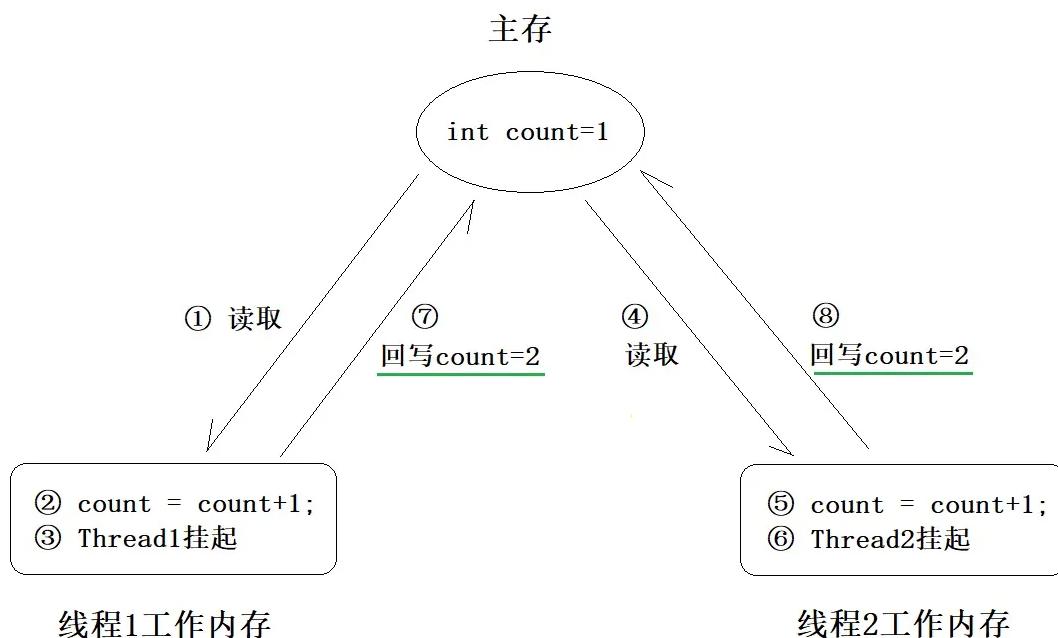
有时候我们很难阐述清楚某个事物是什么，但很容易解释它能干什么，JVM锁也是这个道理。JVM锁的出现，就是为了解决线程安全问题。所谓线程安全问题，可以简单地理解为数据不一致（与预期不一致）。

什么时候可能出现线程安全问题呢？

当同时满足以下三个条件时，才可能引发线程安全问题：

- 多线程环境
- 有共享数据
- 有多条语句操作共享数据/单条语句本身非原子操作（比如`i++`虽然是单条语句，但并非原子操作）

比如线程A、B同时对`int count`进行+1操作（初始值假设为1），在一定的概率下两次操作最终结果可能为2，而不是3。



那么加锁为什么能解决这个问题呢？

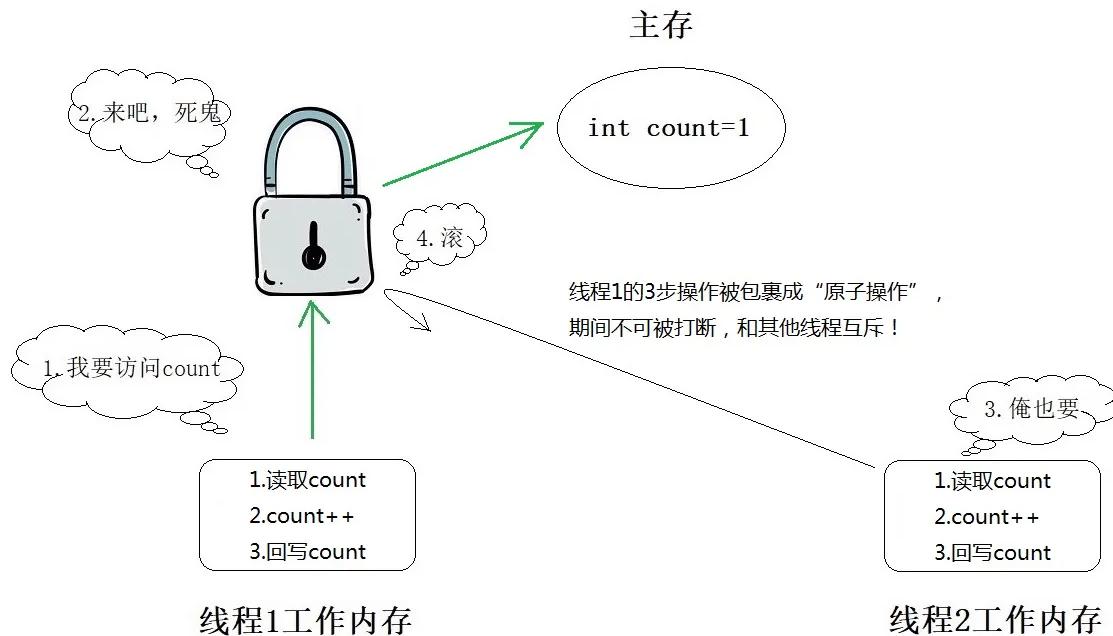
如果不考虑原子性、内存屏障等晦涩的名词，加锁之所以能保证线程安全，核心就是“互斥”。所谓互斥，就是字面意思上的互相排斥。这里的“互相”是指谁呢？就是多线程之间！

怎么实现多线程之间的互斥呢？

引入“中间人”即可。

注意，这是个非常简单且伟大的思想。在编程世界中，通过引入“中介”最终解决问题的案例不胜枚举，包括但不限于Spring、MQ。在码农之间，甚至流传着一句话：没有什么问题是引入中间层解决不了的。

而JVM锁其实就是线程和线程彼此的“中间人”，多个线程在操作加锁数据前都必须征求“中间人”的同意：



锁在这里扮演的角色其实就是守门员，是唯一的访问入口，所有的线程都要经过它的拷问。在JDK中，锁的实现机制最常见的就是两种，分别是两个派系：

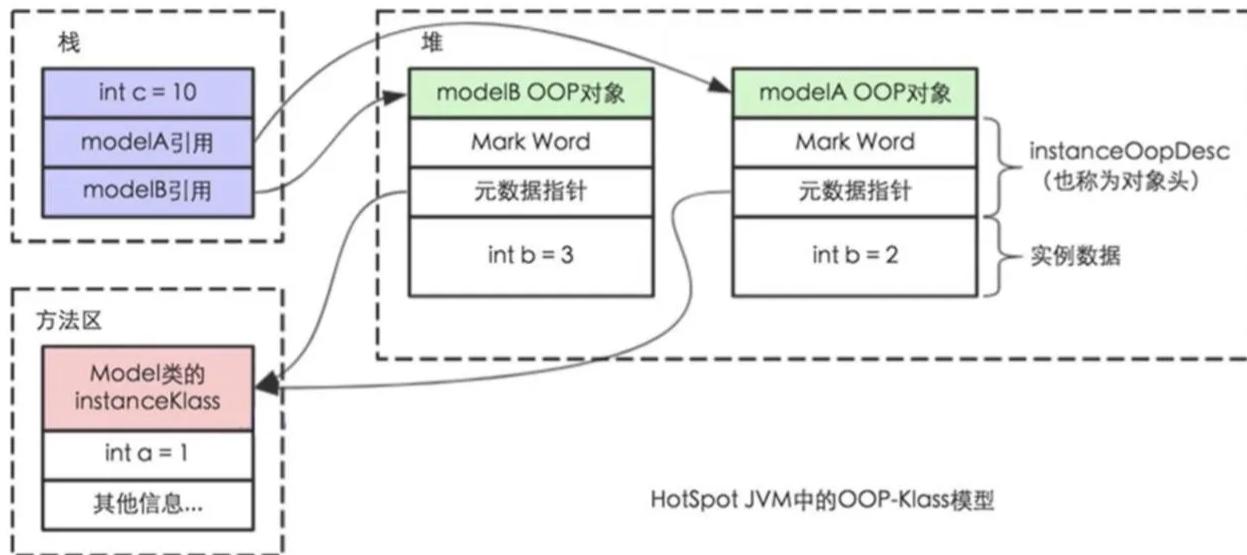
- synchronized关键字
- AQS

个人觉得synchronized关键字要比AQS难理解，但AQS的源码比较抽象。这里简要介绍一下Java对象内存结构和synchronized关键字的实现原理。

Java对象内存结构

要了解synchronized关键字，首先要知道Java对象的内存结构。强调一遍，是Java对象的内存结构。

它的存在仿佛向我们抛出一个疑问：如果有机会解剖一个Java对象，我们能看到什么？



右上图画了两个对象，只看其中一个即可。我们可以观察到，Java对象内存结构大致分为几块：

- Mark Word (锁相关)
- 元数据指针 (class pointer, 指向当前实例所属的类)
- 实例数据 (instance data, 我们平常看到的仅仅是这一块)
- 对齐 (padding, 和内存对齐有关)

如果此前没有了解过Java对象的内存结构，你可能会感到吃惊：天呐，我还以为Java对象就只有属性和方法！

是的，我们最熟悉实例数据这一块，而且以为只有这一块。也正是这个观念的限制，导致一部分初学者很难理解synchronized。比如初学者经常会疑惑：

- 为什么任何对象都可以作为锁？
- Object对象锁和类锁有什么区别？
- synchronized修饰的普通方法使用的锁是什么？
- synchronized修饰的静态方法使用的锁是什么？

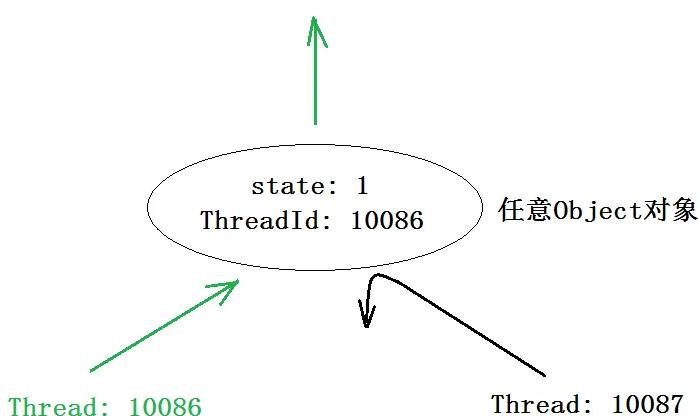
这一切的一切，其实都可以在Java对象内存结构中的Mark Word找到答案：

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁	略			0	01
偏向锁	线程ID	略		1	01
轻量级锁	指向栈中锁记录指针			00	
重量级锁	指向重量级锁指针			10	

很多同学可能是第一次看到这幅图，会感到有点懵，没关系，我也很头大，都一样的。

Mark Word包含的信息还是蛮多的，但这里我们只需要简单地把它理解为记录锁信息的标记即可。上图展示的是32位虚拟机下的Java对象内存，如果你仔细数一数，会发现全部bit加起来刚好是32位。64位虚拟机下的结构大同小异，就不特别介绍。

Mark Word从有限的32bit中划分出2bit，专门用作锁标志位，通俗地讲就是标记当前锁的状态。



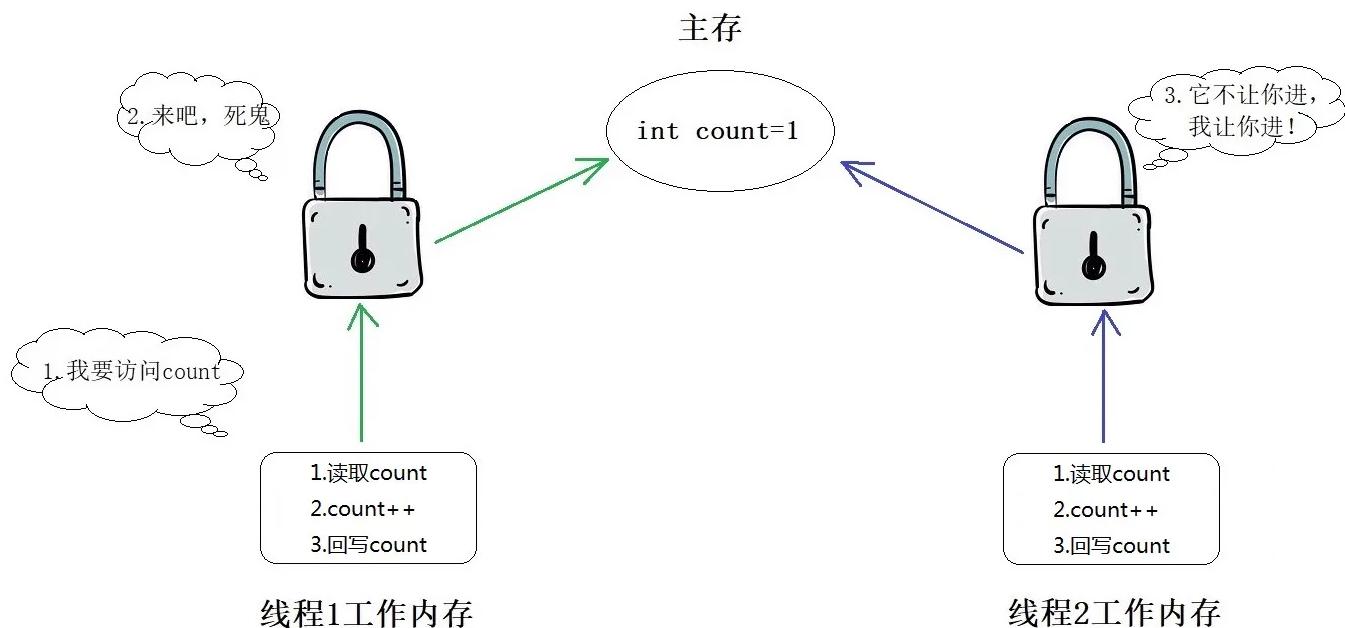
正因为每个Java对象都有Mark Word，而Mark Word能标记锁状态（把自己当做锁），所以Java中任意对象都可以作为synchronized的锁：

```
1 synchronized(person){
2 }
3 synchronized(student){
4 }
```

Java

所谓的this锁就是当前对象，而Class锁就是当前对象所属类的Class对象，本质也是Java对象。
synchronized修饰的普通方法底层使用当前对象作为锁，synchronized修饰的静态方法底层使用Class对象作为锁。

但如果要保证多个线程互斥，最基本的条件是它们使用同一把锁：



对同一份数据加两把不同的锁是没有意义的，实际开发时应该注意避免下面的写法：

```
1 -> synchronized(Person.class){  
2     // 操作count  
3 }  
4  
5 -> synchronized(person){  
6     // 操作count  
7 }
```

或者

```
1 public synchronized void method1(){
2     // 操作count
3 }
4
5 public static synchronized void method1(){
6     // 操作count
7 }
```

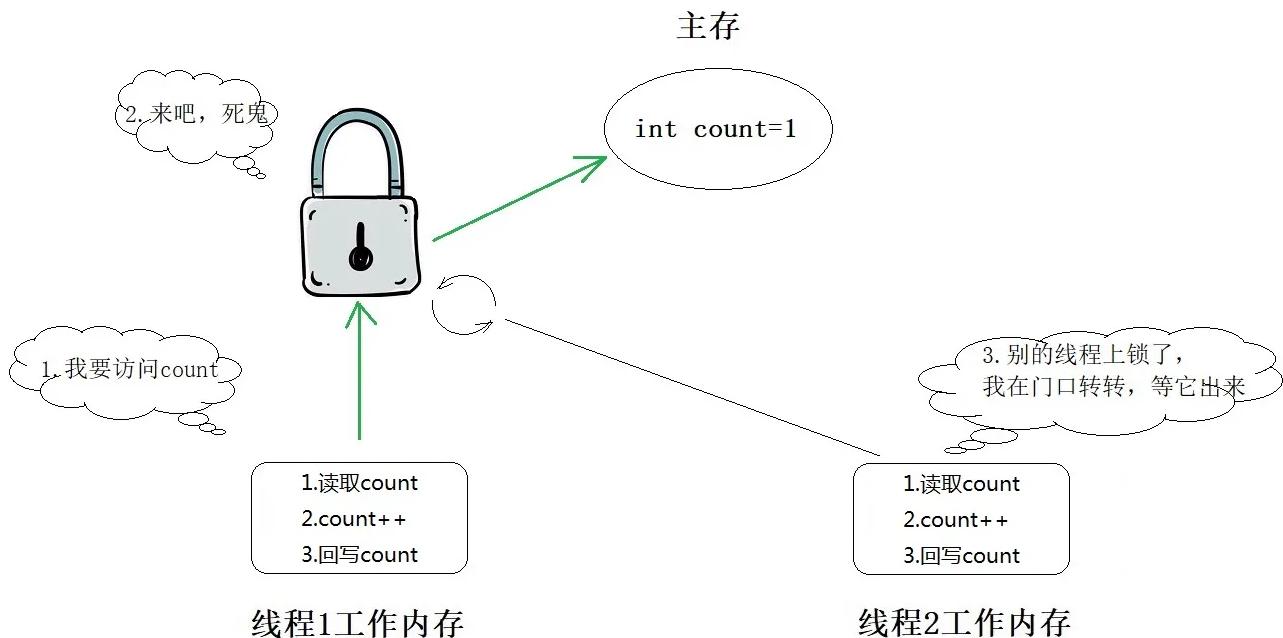
synchronized与锁升级

大致介绍完Java对象内存结构后，我们再来解决一个新疑问：

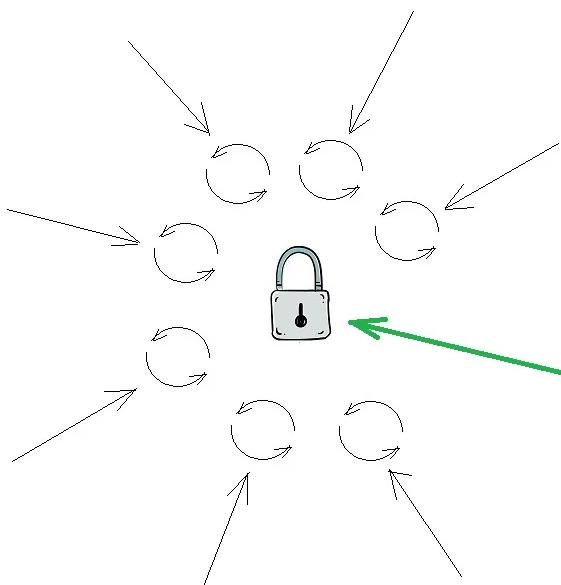
为什么需要标记锁的状态呢？是否意味着synchronized锁有多种状态呢？

在JDK早期版本中，synchronized关键字的实现是直接基于重量级锁的。只要我们在代码中使用了synchronized，JVM就会向操作系统申请锁资源（不论当前是否真的是多线程环境），而向操作系统申请锁是比较耗费资源的，其中涉及到用户态和内核态的切换等，总之就是比较费事，且性能不高。

JDK为了解决JVM锁性能低下的问题，引入了ReentrantLock，它基于CAS+AQS，类似自旋锁。自旋的意思就是，在发生锁竞争的时候，未争取到锁的线程会在门外采取自旋的方式等待锁的释放，谁抢到谁执行。



自旋锁的好处是，不需要兴师动众地切换到内核态申请操作系统的重量级锁，在JVM层面即可实现自旋等待。但世界上并没有百利而无一害的灵丹妙药，CAS自旋虽然避免了状态切换等复杂操作，却要耗费部分CPU资源，尤其当可预计上锁的时间较长且并发较高的情况下，会造成几百上千个线程同时自旋，极大增加CPU的负担。



`synchronized`毕竟JDK亲儿子，所以大概在JDK1.6或者更早期的版本，官方对`synchronized`做了优化，提出了“锁升级”的概念，把`synchronized`的锁划分为多个状态，也就是上图中提到的：

- 无锁
- 偏向锁
- 轻量级锁（自旋锁）
- 重量级锁

无锁就是一个Java对象刚new出来的状态。当这个对象第一次被一个线程访问时，该线程会把自己的线程id“贴到”它的头上（Mark Word中部分位数被修改），表示“你是我的”：

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁	略			0	01
偏向锁	线程ID ←	略		1	01
轻量级锁	指向栈中锁记录指针				00
重量级锁	指向重量级锁指针				10

此时是不存在锁竞争的，所以并不会有什么阻塞或等待。

为什么要设计“偏向锁”这个状态呢？

大家回忆一下，项目中并发的场景真的这么多吗？并没有吧。大部分项目的大部分时候，某个变量都是单个线程在执行，此时直接向操作系统申请重量级锁显然没有必要，因为根本不会发生线程安全问题。

而一旦发生锁竞争时，synchronized便会在一定条件下升级为轻量级锁，可以理解为一种自旋锁，具体自旋多少次以及何时放弃自旋，JDK也有一套相关的控制机制，大家可以自行了解。

同样是自旋，所以synchronized也会遇到ReentrantLock的问题：如果上锁时间长且自旋线程多，又该如何？

此时就会再次升级，变成传统意义上的重量级锁，本质上操作系统会维护一个队列，用空间换时间，避免多个线程同时自旋等待耗费CPU性能，等到上一个线程结束时唤醒等待的线程参与新一轮的锁竞争即可。

拓展阅读（没太大必要）：

[线程安全\(中\)--彻底搞懂synchronized\(从偏向锁到重量级锁\)](#)

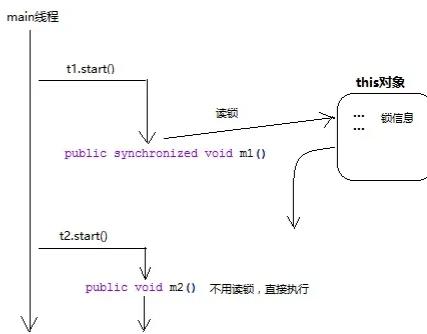
[死磕Synchronized底层实现--偏向锁](#)

synchronized案例

让我们一起来看几个案例，加深对synchronized的理解。

- 同一个类中的synchronized method m1和method m2互斥吗？

```
/**  
 * 同步和非同步方法是否可以同时调用?  
 * @author mashibing  
 */  
  
package yxxy.c_007;  
  
public class T {  
    // synchronized加在非静态方法上，锁对象即为this  
  
    public synchronized void m1() {  
        System.out.println(Thread.currentThread().getName() + " m1 start...");  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println(Thread.currentThread().getName() + " m1 end");  
    }  
  
    public void m2() {  
        try {  
            Thread.sleep(500);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println(Thread.currentThread().getName() + " m2 ");  
    }  
  
    public static void main(String[] args) {  
        T t = new T();  
  
        new Thread(t::m1, "t1").start();  
        new Thread(t::m2, "t2").start();  
    }  
}
```



t1线程执行m1方法时要去读this对象锁，但是t2线程并不需要读锁，两者各管各的，没有交集（不共用一把锁）

- 同一个类中synchronized method m1中可以调用synchronized method m2吗？

```


    /**
     * 一个同步方法可以调用另外一个同步方法，一个线程已经拥有某个对象的锁，再次申请的时候仍然会得到该对象的锁。
     * 也就是说synchronized获得的锁是可重入的
     * @author mashibing
     */
    package yxxy.c_009;

    import java.util.concurrent.TimeUnit;

    public class T {
        ①↓ ②
        synchronized void m1() {
            System.out.println("m1 start");
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            ③
            m2();
        }

        synchronized void m2() {
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("m2");
        }
    }


```

synchronized是可重入锁，可以粗浅地理解为同一个线程在已经持有该锁的情况下，可以再次获取锁，并且会在某个状态量上做+1操作（ReentrantLock也支持重入）

- 子类同步方法synchronized method m可以调用父类的synchronized method m吗？

```


    /**
     * 这里是继承中有可能发生的情形，子类调用父类的同步方法
     * @author mashibing
     */
    package yxxy.c_010;

    import java.util.concurrent.TimeUnit;

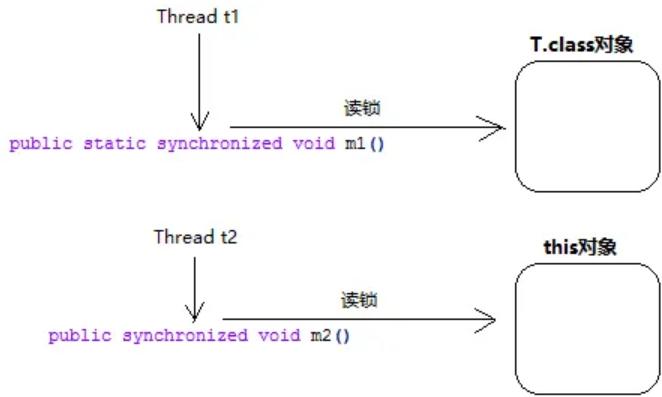
    public class T {
        synchronized void m() {
            System.out.println("m start");
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("m end");
        }
    }

    class TT extends T {
        @Override
        synchronized void m() {
            System.out.println("child m start");
            super.m();
            System.out.println("child m end");
        }
    }


```

子类对象初始化前，会调用父类构造方法，在结构上相当于包裹了一个父类对象，用的都是this锁对象

- 静态同步方法和非静态同步方法互斥吗？



各玩各的，不是同一把锁，谈不上互斥

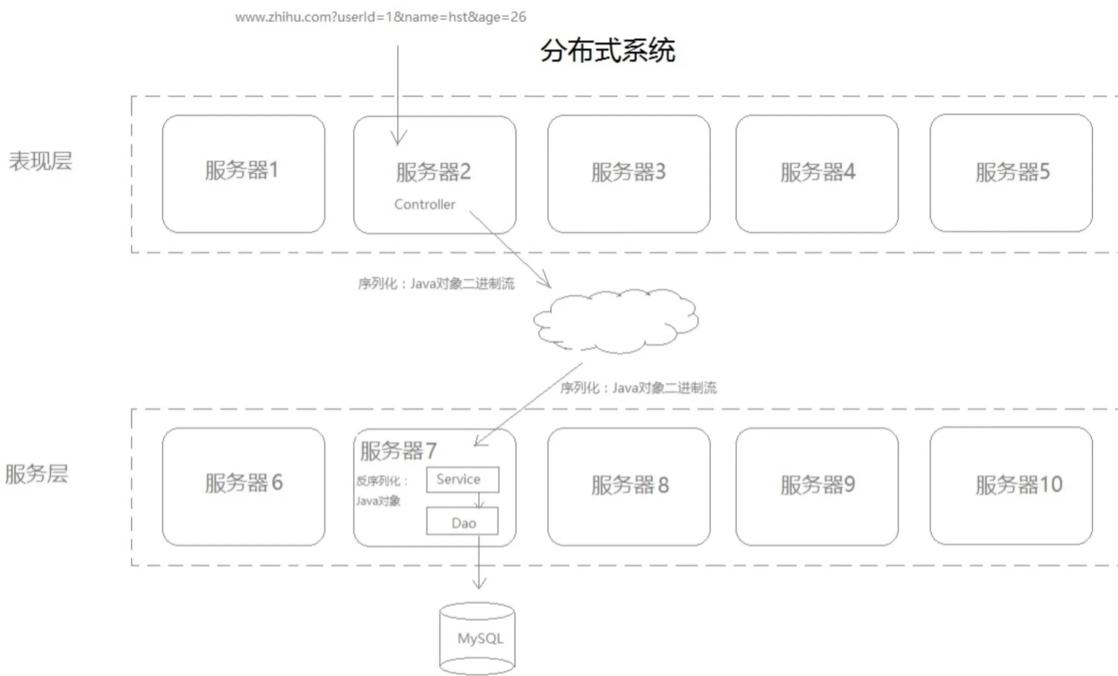
Redis分布式锁的概念

谈到Redis分布式锁，总是会有这样或那样的疑问：

- 什么是分布式
- 什么是分布式锁
- 为什么需要分布式锁
- Redis如何实现分布式锁

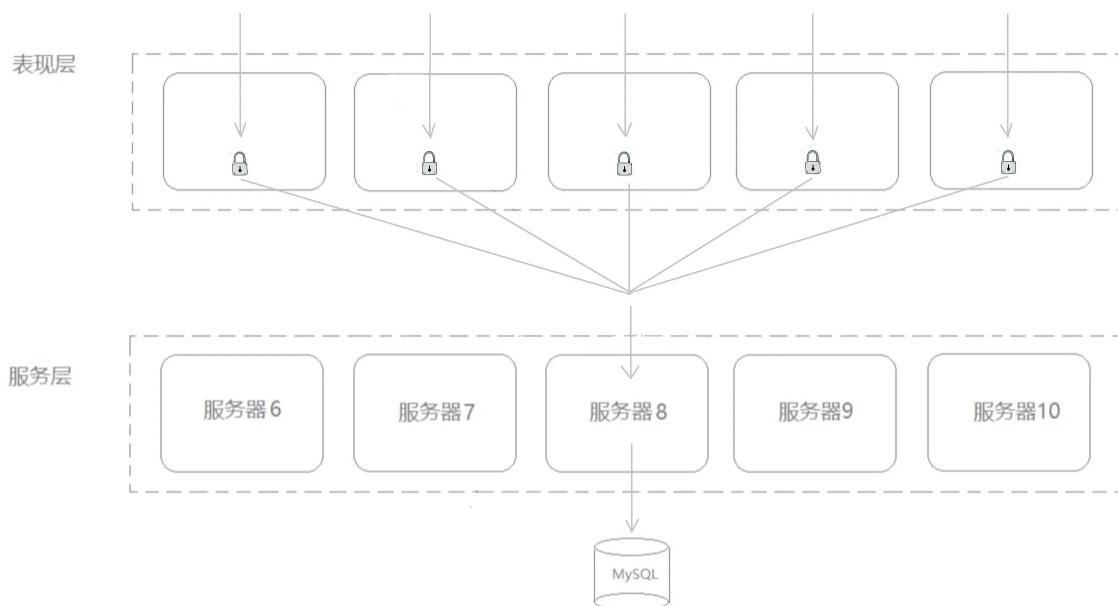
前3个问题其实可以一起回答，至于Redis如何实现分布式锁，我们放在下一篇。

什么是分布式？这是个很复杂的概念，我也很难说准确，所以干脆画个图，大家各花入各眼吧：



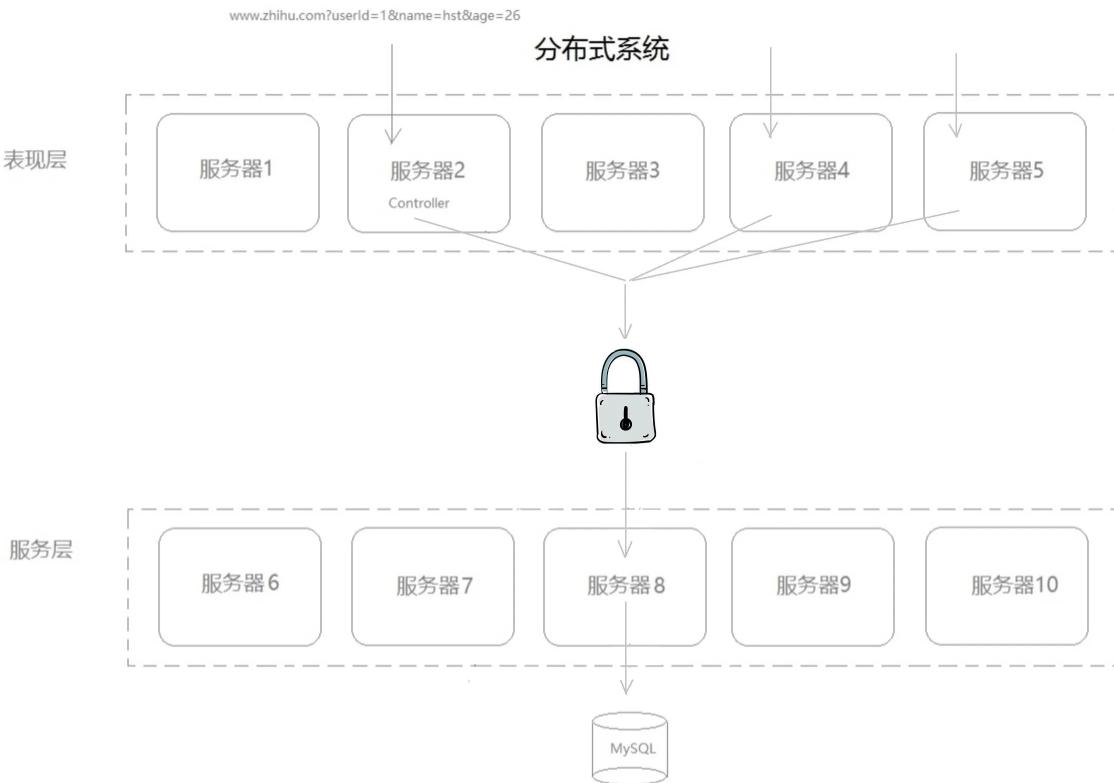
分布式有个很显著的特点是，Service A和Service B极有可能并不是部署在同一个服务器上，所以它们也不共享同一片JVM内存。而上面介绍了，要想实现线程互斥，必须保证所有访问的线程使用的是同一把锁（JVM锁此时就无法保证互斥）。

对于分布式项目，有多少台服务器就有多少片JVM内存，即使每片内存中各设置一把“独一无二”的锁，从整体来看项目中的锁就不是唯一的。



此时，如何保证每一个JVM上的线程共用一把锁呢？

答案是：把锁抽取出来，让线程们在同一片内存相遇。



但锁是不能凭空存在的，本质还是要在内存中，此时可以使用Redis缓存作为锁的宿主环境，这就是Redis能构造分布式锁的原因。

Redis的锁长啥样

`synchronized`关键字和`ReentrantLock`，它们都是实实在在已经实现的锁，而且还有标志位啥的。但Redis就是一个内存...怎么作为锁呢？

有一点大家要明确，Redis之所以能用来做分布式锁，肯定不只是因为它是一片内存，否则JVM本身也占有内存，为什么无法自己实现分布式锁呢？

我个人的理解是，要想自定义一个分布式锁，必须至少满足几个条件：

- 多进程可见（独立于多节点系统之外的一片内存）
- 互斥（可以通过单线程，或者某种顺序机制）
- 可重入

还有个条件，默认要支持：只有持有这把锁的客户端才能解锁

以上三点Redis都能满足。在上面三个条件下，其实怎么设计锁，完全取决于个人如何定义锁。就好比现实生活中，通常我们理解的锁就是有个钥匙孔、需要插入钥匙的金属小物件。然而锁的形态可不止这么一种，随着科技的发展，什么指纹锁、虹膜锁层出不穷，但归根结底它们之所以被称为“锁”，是因为都保证了“互斥”（我行，你不行）。

如果我们能设计一种逻辑，它能造成某个场景下的“互斥事件”，那么它就可以被称为“锁”。比如，某家很有名的网红店，一天只接待一位客人。门口没有营业员，就放了一台取号机，里面放了一张票。你如果去迟了，票就没了，你就进不了这家店。这个场景下，没票的顾客进不去，被锁在门外。此时，取票机造成了“互斥事件”，那么它就可以叫做“锁”。

而Redis提供了setnx指令，如果某个key当前不存在则设置成功并返回true，否则不再重复设置，直接返回false。这不就是编程界的取号机吗？当然，实际用到的命令可不止这一个，具体如何实现，请看下一篇
~

这一篇从JVM锁聊到了Redis分布式锁，还介绍了Java的对象内存结构及synchronized底层的原理，相信大家对“锁”已经有了自己的感性认识。下一篇我们将通过分布式定时任务的案例介绍Redis分布式锁的使用场景。

下次见。

思考一个问题：分布式系统是否一定要分布式锁？

分布式系统如果要加锁是否一定要使用分布式锁呢？

可能未必。

如果你需要的是写锁，那么可能确实需要分布式锁保证单一线程处理数据，而如果是为了防止缓存穿透（热点数据定时失效），那么使用JVM本地锁也没有太大关系。比如某个服务有10个节点，在使用JVM锁的情况下，即使某一时刻每个节点各自涌入1000个请求，虽然总共有1w个请求，但最终打到数据库的也只有10个，数据库层面是完全可以抗拒这点请求量的，又由于本身是查询，所以不会造成线程安全问题。

| 来自: 浅谈Redis分布式锁(上)

华为OD机试2025B卷 - 乘坐保密电梯 (C++ & Python & JAVA & JS)

乘坐保密电梯

2025B卷目录点击查看：[华为OD机试2025B卷真题题库目录 | 机考题库 + 算法考点详解](#)

| 2025B卷 100分题型

题目描述

有一座保密大楼，你从0楼到达指定楼层m，必须这样的规则乘坐电梯：

给定一个数字序列，每次根据序列中的数字n，上升n层或者下降n层，前后两次的方向必须相反，规定首次的方向向上，自行组织序列的顺序按规定操作到达指定楼层。

求解到达楼层的序列组合，如果不能到达楼层，给出小于该楼层的最近序列组合。

输入描述

第一行：期望的楼层，取值范围[1,50]; 序列总个数，取值范围[1,23]

第二行：序列，每个值取值范围[1,50]

输出描述

能够达到楼层或者小于该楼层最近的序列

备注

- 操作电梯时不限定楼层范围。
- 必须对序列中的每个项进行操作，不能只使用一部分。

用例1

输入

输出

说明

| 1 2 6, 6 2 1均为可行解，按先处理大值的原则结果为6 2 1

题解

思路： 递归回溯

- 首先确定需要选取上升的数量,规定第一个为上升并且交叉上升和下降,那么选取上升的数量为 $(n + 1) / 2$,向上取整。
- 将输入数组降序排序。
- 使用 递归回溯 枚举所有选取 `exceptCount` 上升数, 记录其中出现 上升高度 $\leq m$, 并且最接近 m 的方案。
- 将每个候选方案对应的数组构建出来, 选取其中字典序最大的数组就是结果。

C++

```
1 #include<iostream>
2 #include<vector>
3 #include<string>
4 #include <utility>
5 #include <sstream>
6 #include<algorithm>
7 #include<list>
8 #include<queue>
9 #include<map>
10 #include<set>
11 using namespace std;
12
13 bool cmp(int x, int y) {
14     return x > y;
15 }
16 // 整体序列和
17 int totalSum;
18 vector<int> res;
19 int minDiff;
20 int m,n;
21
22 // 递归回溯枚举 选取exceptCount的数
23 void DFS(vector<int>& ans, int index , int currentSum, int count, int exceptCount, int& visited) {
24     if (count > exceptCount) {
25         return;
26     }
27     if (exceptCount == count) {
28         // 净增楼层
29         int diff = currentSum - (totalSum - currentSum);
30         // 上升超过m的不考虑
31         if (diff > m) {
32             return;
33         }
34
35         // 找最小差距
36         diff = abs(diff - m);
37         if (diff < minDiff) {
38             res.clear();
39             minDiff = diff;
40             res.push_back(visited);
41         } else if (diff == minDiff) {
42             res.push_back(visited);
43         }
44     }
45 }
```

```

45     }
46
47     for (int i = index; i < n; i++) {
48         int num = ans[i];
49         int nextCurrentSum = currentSum + num;
50         // 剪枝 最终会超过m层
51         if (nextCurrentSum - (totalSum - nextCurrentSum) > m) {
52             continue;
53         }
54         // 递归回溯
55         visited |= 1 << (n - i -1);
56         DFS(ans, i + 1, nextCurrentSum, count + 1, exceptCount, visited);
57         visited &= ~(1 << (n - i - 1));
58
59     }
60 }
61
62 // 交叉构建结果集
63 vector<int> buildArr(vector<int>& ans, int num) {
64     vector<bool> visted(n, false);
65     for (int i = 0; i < n; i++) {
66         visted[i] = 1 & (num >> (n-1-i));
67     }
68     vector<int> s(n);
69     int pos = 0;
70     for (int i = 0; i < n; i++) {
71         if (visted[i]) {
72             s[pos] = ans[i];
73             pos += 2;
74         }
75     }
76     pos = 1;
77     for (int i = 0; i < n; i++) {
78         if (!visted[i]) {
79             s[pos] = ans[i];
80             pos += 2;
81         }
82     }
83     return s;
84 }
85
86 // 在所有mask生成的数组中，找出字典序最大的
87 vector<int> findBestArray(vector<int>& ans) {
88     vector<int> best;
89
90     for (const int& mask : res) {
91         vector<int> curr = buildArr(ans, mask);
92         if (best.empty() || curr > best) {

```

```
93         best = curr;
94     }
95 }
96 return best;
97 }

99 int main() {
100     cin >> m >> n;
101     vector<int> ans(n);
102     int sum = 0;
103     for (int i = 0; i < n; i++) {
104         cin >> ans[i];
105         sum += ans[i];
106     }
107     //
108     totalSum = sum;
109     minDiff = sum + m;
110     // 从大到小排序
111     sort(ans.begin(), ans.end(), cmp);
112     // 向上取整 向上的数量
113     int exceptUpCount = (n + 1) / 2;
114

115     int status = 0;
116     DFS(ans, 0, 0, 0, exceptUpCount, status);
117

118     vector<int> anwser = findBestArray(ans);
119     for (int i = 0; i < n; i++) {
120         cout << anwser[i];
121         if (i != n-1) {
122             cout << " ";
123         }
124     }
125     return 0;
126 }
```

JAVA

```
1 import java.util.*;
2
3 public class Main {
4     static int totalSum;
5     static List<Integer> res = new ArrayList<>();
6     static int minDiff;
7     static int m, n;
8
9     // 比较函数, 从大到小排序
10    static boolean cmp(int x, int y) {
11        return x > y;
12    }
13
14    // 递归回溯枚举 选取exceptCount的数
15    static void DFS(List<Integer> ans, int index, int currentSum, int cou-
nt, int exceptCount, int visited) {
16        if (count > exceptCount) {
17            return;
18        }
19        if (exceptCount == count) {
20            // 净增楼层
21            int diff = currentSum - (totalSum - currentSum);
22            // 上升超过m的不考虑
23            if (diff > m) {
24                return;
25            }
26            // 找最小差距
27            diff = Math.abs(diff - m);
28            if (diff < minDiff) {
29                res.clear();
30                minDiff = diff;
31                res.add(visited);
32            } else if (diff == minDiff) {
33                res.add(visited);
34            }
35            return;
36        }
37
38        for (int i = index; i < n; i++) {
39            int num = ans.get(i);
40            int nextCurrentSum = currentSum + num;
41            // 剪枝 最终会超过m层
42            if (nextCurrentSum - (totalSum - nextCurrentSum) > m) {
43                continue;
44            }
45        }
46    }
47}
```

```

45         // 递归回溯
46         visited |= 1 << (n - i - 1);
47         DFS(ans, i + 1, nextCurrentSum, count + 1, exceptCount, visit
48     ed);
49         visited &= ~(1 << (n - i - 1));
50     }
51 }
52
53 // 交叉构建结果集
54 static List<Integer> buildArr(List<Integer> ans, int num) {
55     boolean[] visited = new boolean[n];
56     for (int i = 0; i < n; i++) {
57         visited[i] = (1 & (num >> (n - 1 - i))) == 1;
58     }
59     List<Integer> s = new ArrayList<>(Collections.nCopies(n, 0));
60     int pos = 0;
61     for (int i = 0; i < n; i++) {
62         if (visited[i]) {
63             s.set(pos, ans.get(i));
64             pos += 2;
65         }
66     }
67     pos = 1;
68     for (int i = 0; i < n; i++) {
69         if (!visited[i]) {
70             s.set(pos, ans.get(i));
71             pos += 2;
72         }
73     }
74     return s;
75 }
76
77 // 在所有mask生成的数组中，找出字典序最大的
78 static List<Integer> findBestArray(List<Integer> ans) {
79     List<Integer> best = new ArrayList<>();
80     for (int mask : res) {
81         List<Integer> curr = buildArr(ans, mask);
82         if (best.isEmpty() || compare(curr, best) > 0) {
83             best = curr;
84         }
85     }
86     return best;
87 }
88
89 // 手动比较两个列表的字典序
90 static int compare(List<Integer> a, List<Integer> b) {
91     for (int i = 0; i < a.size(); i++) {
92         if (!a.get(i).equals(b.get(i))) {

```

```

92             return a.get(i) - b.get(i);
93         }
94     }
95     return 0;
96 }
97
98 public static void main(String[] args) {
99     Scanner sc = new Scanner(System.in);
100    m = sc.nextInt();
101    n = sc.nextInt();
102    List<Integer> ans = new ArrayList<>();
103    totalSum = 0;
104    for (int i = 0; i < n; i++) {
105        int x = sc.nextInt();
106        ans.add(x);
107        totalSum += x;
108    }
109    minDiff = totalSum + m;
110    // 从大到小排序
111    ans.sort((a, b) -> b - a);
112
113    int exceptUpCount = (n + 1) / 2;
114    int status = 0;
115    DFS(ans, 0, 0, 0, exceptUpCount, status);
116
117    List<Integer> answer = findBestArray(ans);
118    for (int i = 0; i < answer.size(); i++) {
119        System.out.print(answer.get(i));
120        if (i != answer.size() - 1) {
121            System.out.print(" ");
122        }
123    }
124 }
125 }
```

Python

```
1 totalSum = 0
2
3 res = []
4
5 minDiff = 0
6 m, n = 0, 0
7
8 def DFS(ans, index, currentSum, count, exceptCount, visited):
9     global minDiff, res, totalSum, m, n
10    if count > exceptCount:
11        return
12    if count == exceptCount:
13
14        diff = currentSum - (totalSum - currentSum)
15
16        if diff > m:
17            return
18
19        diff = abs(diff - m)
20        if diff < minDiff:
21            res.clear()
22            minDiff = diff
23            res.append(visited)
24        elif diff == minDiff:
25            res.append(visited)
26        return
27
28    for i in range(index, n):
29        num = ans[i]
30        nextCurrentSum = currentSum + num
31
32        if nextCurrentSum - (totalSum - nextCurrentSum) > m:
33            continue
34
35        visited |= 1 << (n - i - 1)
36        DFS(ans, i + 1, nextCurrentSum, count + 1, exceptCount, visited)
37        visited &= ~(1 << (n - i - 1))
38
39
40 def buildArr(ans, mask):
41     visited = [(mask >> (n - 1 - i)) & 1 for i in range(n)]
42     s = [0] * n
43     pos = 0
44     for i in range(n):
45         if visited[i]:
```

```

46         s[pos] = ans[i]
47         pos += 2
48     pos = 1
49     for i in range(n):
50         if not visited[i]:
51             s[pos] = ans[i]
52             pos += 2
53     return s
54
55
56 def findBestArray(ans):
57     best = []
58     for mask in res:
59         curr = buildArr(ans, mask)
60         if not best or curr > best:
61             best = curr
62     return best
63
64 if __name__ == "__main__":
65     m, n = map(int, input().split())
66     ans = list(map(int, input().split()))
67     totalSum = sum(ans)
68     minDiff = totalSum + m
69
70     ans.sort(reverse=True)
71
72     exceptUpCount = (n + 1) // 2
73     status = 0
74     DFS(ans, 0, 0, 0, exceptUpCount, status)
75     answer = findBestArray(ans)
76     print(*answer)

```

JavaScript

```
1 const readline = require('readline');
2
3 const rl = readline.createInterface({
4     input: process.stdin,
5     output: process.stdout
6 });
7
8 let input = [];
9 rl.on('line', function (line) {
10     input.push(...line.trim().split(' ').map(Number));
11 }).on('close', function () {
12     let idx = 0;
13     const m = input[idx++];
14     const n = input[idx++];
15     const ans = [];
16     for (let i = 0; i < n; i++) {
17         ans.push(input[idx++]);
18     }
19     solve(m, n, ans);
20 });
21
22 let totalSum;
23 let res = [];
24 let minDiff;
25 let m_, n_;
26
27 function solve(m, n, ans) {
28     m_ = m;
29     n_ = n;
30     totalSum = ans.reduce((a, b) => a + b, 0);
31     minDiff = totalSum + m;
32
33     ans.sort((a, b) => b - a);
34
35     let exceptUpCount = Math.floor((n + 1) / 2);
36     let status = 0;
37     DFS(ans, 0, 0, 0, exceptUpCount, status);
38
39     const answer = findBestArray(ans);
40     console.log(answer.join(' '));
41 }
42
43
44 function DFS(ans, index, currentSum, count, exceptCount, visited) {
45     if (count > exceptCount) {
```

```

46         return;
47     }
48     if (count === exceptCount) {
49
50         let diff = currentSum - (totalSum - currentSum);
51
52         if (diff > m_) {
53             return;
54         }
55
56         diff = Math.abs(diff - m_);
57         if (diff < minDiff) {
58             res = [];
59             minDiff = diff;
60             res.push(visited);
61         } else if (diff === minDiff) {
62             res.push(visited);
63         }
64         return;
65     }
66     for (let i = index; i < n_; i++) {
67         let num = ans[i];
68         let nextCurrentSum = currentSum + num;
69
70         if (nextCurrentSum - (totalSum - nextCurrentSum) > m_) {
71             continue;
72         }
73
74         visited |= 1 << (n_ - i - 1);
75         DFS(ans, i + 1, nextCurrentSum, count + 1, exceptCount, visited);
76         visited &= ~(1 << (n_ - i - 1));
77     }
78 }
79
80
81 function buildArr(ans, mask) {
82     let visited = Array.from({length: n_}, (_, i) => (mask >> (n_ - 1 - i)) & 1);
83     let s = Array(n_).fill(0);
84     let pos = 0;
85     for (let i = 0; i < n_; i++) {
86         if (visited[i]) {
87             s[pos] = ans[i];
88             pos += 2;
89         }
90     }
91     pos = 1;
92     for (let i = 0; i < n_; i++) {

```

```

93         if (!visited[i]) {
94             s[pos] = ans[i];
95             pos += 2;
96         }
97     }
98     return s;
99 }
100
101
102 function findBestArray(ans) {
103     let best = [];
104     for (let mask of res) {
105         let curr = buildArr(ans, mask);
106         if (best.length === 0 || compare(curr, best) > 0) {
107             best = curr;
108         }
109     }
110     return best;
111 }
112
113 function compare(a, b) {
114     for (let i = 0; i < a.length; i++) {
115         if (a[i] !== b[i]) {
116             return a[i] - b[i];
117         }
118     }
119     return 0;
120 }

```

Go

```
1 package main
2
3 import (
4     "fmt"
5     "sort"
6 )
7
8 var (
9     totalSum int
10    res      []int
11    minDiff  int
12    m, n     int
13 )
14
15 func main() {
16     fmt.Scan(&m, &n)
17     ans := make([]int, n)
18     sum := 0
19     for i := 0; i < n; i++ {
20         fmt.Scan(&ans[i])
21         sum += ans[i]
22     }
23     totalSum = sum
24     minDiff = sum + m
25
26     sort.Slice(ans, func(i, j int) bool {
27         return ans[i] > ans[j]
28     })
29
30     exceptUpCount := (n + 1) / 2
31     status := 0
32     DFS(ans, 0, 0, 0, exceptUpCount, status)
33     answer := findBestArray(ans)
34     for i := 0; i < n; i++ {
35         fmt.Print(answer[i])
36         if i != n-1 {
37             fmt.Print(" ")
38         }
39     }
40     fmt.Println()
41 }
42
43
44 func DFS(ans []int, index, currentSum, count, exceptCount, visited int) {
45     if count > exceptCount {
```

```

46     return
47 }
48 if count == exceptCount {
49
50     diff := currentSum - (totalSum - currentSum)
51
52     if diff > m {
53         return
54     }
55
56     if d := abs(diff - m); d < minDiff {
57         res = []int{}
58         minDiff = d
59         res = append(res, visited)
60     } else if d == minDiff {
61         res = append(res, visited)
62     }
63     return
64 }
65
66 for i := index; i < n; i++ {
67     num := ans[i]
68     nextCurrentSum := currentSum + num
69
70     if nextCurrentSum-(totalSum-nextCurrentSum) > m {
71         continue
72     }
73
74     visited |= 1 << (n - i - 1)
75     DFS(ans, i+1, nextCurrentSum, count+1, exceptCount, visited)
76     visited &= ^(1 << (n - i - 1))
77 }
78 }
79
80
81 func buildArr(ans []int, mask int) []int {
82     visited := make([]bool, n)
83     for i := 0; i < n; i++ {
84         visited[i] = (mask>>(n-1-i))&1 == 1
85     }
86     s := make([]int, n)
87     pos := 0
88     for i := 0; i < n; i++ {
89         if visited[i] {
90             s[pos] = ans[i]
91             pos += 2
92         }
93     }

```

```

94     pos = 1
95     for i := 0; i < n; i++ {
96         if !visited[i] {
97             s[pos] = ans[i]
98             pos += 2
99         }
100    }
101    return s
102 }
103
104
105 func findBestArray(ans []int) []int {
106     var best []int
107     for _, mask := range res {
108         curr := buildArr(ans, mask)
109         if best == nil || compare(curr, best) > 0 {
110             best = curr
111         }
112     }
113     return best
114 }
115
116 func compare(a, b []int) int {
117     for i := 0; i < len(a); i++ {
118         if a[i] != b[i] {
119             return a[i] - b[i]
120         }
121     }
122     return 0
123 }
124
125 func abs(x int) int {
126     if x < 0 {
127         return -x
128     }
129     return x
130 }
```

来自: 华为od 机试 2025B卷 – 乘坐保密电梯 (C++ & Python & JAVA & JS & GO)_华为机考a卷乘坐
保密电梯–CSDN博客

华为OD机试 2025 B卷 - 战场索敌 (C++ & Python & JAVA & JS &

战场索敌

华为OD机试真题目录点击查看: [华为OD机试2025B卷真题题库目录 | 机考题库 + 算法考点详解](#)

华为OD机试2025B卷 200分题型

题目描述

有一个大小是N*M的战场地图，被墙壁‘#’分隔成大小不同的区域，上下左右四个方向相邻的空地‘.’属于同一个区域，只有空地上可能存在敌人‘E’，
请求出地图上总共有多少区域里的敌人数小于K。

输入描述

第一行输入为N,M,K；

- N表示地图的行数，M表示地图的列数，K表示目标敌人数量
- N, M<=100

之后为一个NxM大小的字符数组。

输出描述

敌人数小于K的区域数量

示例1

输入

输出

说明

地图被墙壁分为两个区域，左边区域有1个敌人，右边区域有3个敌人，符合条件的区域数量是1

题解

思路： [DFS/BFS](#)

1. 接收输入的战场地图，以及指定要求的K。
2. 接下来从上往下 从左往右顺序遍历输入的地图，当遇到 E 字符时使用 DFS/BFS 统计当前区域的敌人数。如果敌人数量 $\leq k$,则满足要求区域 + 1. BFS访问过程中可以把访问过的 E 重新赋值为 . 来防止重复访问 (也可以定义visited数组来防止重复访问)。
3. 通过2逻辑，遍历完地图之后。输出满足要求的区域数量即可。

C++

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int rowCount, colCount, enemyThreshold;
7 vector<vector<char>> grid;
8 vector<vector<int>> visited;
9 int currentEnemyCount;
10
11
12 void exploreRegion(int x, int y) {
13     visited[x][y] = 1;
14
15     if (grid[x][y] == 'E') {
16         currentEnemyCount++;
17     }
18
19
20     vector<vector<int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
21
22     for (const auto& dir : directions) {
23         int newX = x + dir[0];
24         int newY = y + dir[1];
25
26         if (newX >= 0 && newX < rowCount && newY >= 0 && newY < colCount &
27             &
28             visited[newX][newY] == 0 && grid[newX][newY] != '#') {
29             exploreRegion(newX, newY);
30         }
31     }
32 }
33
34 int main() {
35     cin >> rowCount >> colCount >> enemyThreshold;
36
37     grid.resize(rowCount, vector<char>(colCount));
38     visited.resize(rowCount, vector<int>(colCount));
39
40     for (int i = 0; i < rowCount; i++) {
41         string row;
42         cin >> row;
43         for (int j = 0; j < colCount; j++) {
```

```
45         grid[i][j] = row[j];
46     }
47 }
48
49 int validRegions = 0;
50
51
52 for (int i = 0; i < rowCount; i++) {
53     for (int j = 0; j < colCount; j++) {
54         if (visited[i][j] != 0 || grid[i][j] == '#') {
55             continue;
56         }
57         currentEnemyCount = 0;
58         exploreRegion(i, j);
59
60
61         if (currentEnemyCount < enemyThreshold) {
62             validRegions++;
63         }
64     }
65 }
66
67 cout << validRegions << endl;
68
69 return 0;
}
```

Java

```
1 import java.util.*;
2
3 public class Main {
4     static int rowCount, colCount, enemyThreshold;
5     static char[][] grid;
6     static boolean[][] visited;
7     static int currentEnemyCount;
8
9
10    static void exploreRegion(int x, int y) {
11        visited[x][y] = true;
12
13        if (grid[x][y] == 'E') {
14            currentEnemyCount++;
15        }
16
17
18        int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
19
20        for (int[] dir : directions) {
21            int newX = x + dir[0];
22            int newY = y + dir[1];
23
24
25            if (newX >= 0 && newX < rowCount && newY >= 0 && newY < colCount &&
26                !visited[newX][newY] && grid[newX][newY] != '#') {
27                exploreRegion(newX, newY);
28            }
29        }
30    }
31
32    public static void main(String[] args) {
33        Scanner scanner = new Scanner(System.in);
34        rowCount = scanner.nextInt();
35        colCount = scanner.nextInt();
36        enemyThreshold = scanner.nextInt();
37        scanner.nextLine();
38
39        grid = new char[rowCount][colCount];
40        visited = new boolean[rowCount][colCount];
41
42
43        for (int i = 0; i < rowCount; i++) {
44            String row = scanner.nextLine();
```

```
45         grid[i] = row.toCharArray();
46     }
47
48     int validRegions = 0;
49
50
51     for (int i = 0; i < rowCount; i++) {
52         for (int j = 0; j < colCount; j++) {
53             if (!visited[i][j] && grid[i][j] != '#') {
54                 currentEnemyCount = 0;
55                 exploreRegion(i, j);
56
57                 if (currentEnemyCount < enemyThreshold) {
58                     validRegions++;
59                 }
60             }
61         }
62     }
63
64     System.out.println(validRegions);
65 }
66 }
```

Python

```
1 import sys
2
3
4 rowCount, colCount, enemyThreshold = map(int, sys.stdin.readline().split())
5
6
7 grid = [list(sys.stdin.readline().strip()) for _ in range(rowCount)]
8 visited = [[False] * colCount for _ in range(rowCount)]
9
10 def explore_region(x, y):
11     """ 深度优先搜索，统计当前区域的敌人数 """
12     global current_enemy_count
13     visited[x][y] = True
14
15     if grid[x][y] == 'E':
16         current_enemy_count += 1
17
18
19     directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
20
21     for dx, dy in directions:
22         new_x, new_y = x + dx, y + dy
23
24
25         if 0 <= new_x < rowCount and 0 <= new_y < colCount and not visited[new_x][new_y] and grid[new_x][new_y] != '#':
26             explore_region(new_x, new_y)
27
28 valid_regions = 0
29
30
31 for i in range(rowCount):
32     for j in range(colCount):
33         if not visited[i][j] and grid[i][j] != '#':
34             current_enemy_count = 0
35             explore_region(i, j)
36
37             if current_enemy_count < enemyThreshold:
38                 valid_regions += 1
39
40 print(valid_regions)
```

```
1 const readline = require("readline");
2
3 const rl = readline.createInterface({
4     input: process.stdin,
5     output: process.stdout
6 });
7
8 let inputLines = [];
9 let rowCount, colCount, enemyThreshold;
10 let grid = [];
11 let visited = [];
12 let currentEnemyCount = 0;
13
14
15 rl.on("line", (line) => {
16     inputLines.push(line);
17 }).on("close", () => {
18     [rowCount, colCount, enemyThreshold] = inputLines[0].split(" ").map(Number);
19
20     grid = inputLines.slice(1, rowCount + 1).map(row => row.split(""));
21     visited = Array.from({ length: rowCount }, () => Array(colCount).fill(false));
22
23     function exploreRegion(x, y) {
24         visited[x][y] = true;
25
26         if (grid[x][y] === 'E') {
27             currentEnemyCount++;
28         }
29
30         const directions = [[-1, 0], [1, 0], [0, -1], [0, 1]];
31
32         for (const [dx, dy] of directions) {
33             let newX = x + dx, newY = y + dy;
34
35             if (newX >= 0 && newX < rowCount && newY >= 0 && newY < colCount &&
36                 !visited[newX][newY] && grid[newX][newY] !== "#") {
37                 exploreRegion(newX, newY);
38             }
39         }
40     }
41
42     let validRegions = 0;
```

```
43
44    for (let i = 0; i < rowCount; i++) {
45        for (let j = 0; j < colCount; j++) {
46            if (!visited[i][j] && grid[i][j] !== "#") {
47                currentEnemyCount = 0;
48                exploreRegion(i, j);
49
50                if (currentEnemyCount < enemyThreshold) {
51                    validRegions++;
52                }
53            }
54        }
55    }
56
57    console.log(validRegions);
58});
```

Go

```
1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "os"
7 )
8
9 var rowCount, colCount, enemyThreshold int
10 var grid [][]byte
11 var visited [][]bool
12 var currentEnemyCount int
13
14
15 func exploreRegion(x, y int) {
16     visited[x][y] = true
17
18     if grid[x][y] == 'E' {
19         currentEnemyCount++
20     }
21
22     directions := []int{{-1, 0}, {1, 0}, {0, -1}, {0, 1}}
23
24     for _, dir := range directions {
25         newX, newY := x+dir[0], y+dir[1]
26
27         if newX >= 0 && newX < rowCount && newY >= 0 && newY < colCount &&
28             !visited[newX][newY] && grid[newX][newY] != '#' {
29             exploreRegion(newX, newY)
30         }
31     }
32 }
33
34 func main() {
35     scanner := bufio.NewScanner(os.Stdin)
36     fmt.Scan(&rowCount, &colCount, &enemyThreshold)
37
38     grid = make([][]byte, rowCount)
39     visited = make([][]bool, rowCount)
40
41     for i := 0; i < rowCount; i++ {
42         scanner.Scan()
43         grid[i] = []byte(scanner.Text())
44         visited[i] = make([]bool, colCount)
45     }
}
```

```
46
47     validRegions := 0
48
49     for i := 0; i < rowCount; i++ {
50         for j := 0; j < colCount; j++ {
51             if !visited[i][j] && grid[i][j] != '#' {
52                 currentEnemyCount = 0
53                 exploreRegion(i, j)
54
55                 if currentEnemyCount < enemyThreshold {
56                     validRegions++
57                 }
58             }
59         }
60     }
61
62     fmt.Println(validRegions)
63 }
```

| 来自: 华为OD机试 2025 B卷 – 战场索敌 (C++ & Python & JAVA & JS & GO)-CSDN博客

华为OD机试2025B卷 - 区间连接器(C++ & Python & Java & JS)

区间连接器

华为OD机试真题目录点击查看: [华为OD机试2025B卷真题题库目录 | 机考题库 + 算法考点详解](#)

华为OD机试2025B卷 200分题型

题目描述

有一组区间 $[a_0, b_0], [a_1, b_1], \dots$ (a, b 表示起点, 终点), 区间有可能重叠、相邻, 重叠或相邻则可以合并为更大的区间;

给定一组连接器 $[x_1, x_2, x_3, \dots]$ (x 表示连接器的最大可连接长度, 即 $x \geq gap$), 可用于将分离的区间连接起来, 但两个分离区间之间只能使用1个连接器;

请编程实现使用连接器后, 最少的区间数结果。

备注:

- 区间数量 < 10000 , a, b 均 ≤ 10000
- 连接器数量 < 10000 ; $x \leq 10000$

输入描述

无

输出描述

无

用例1

输入

```
1  [1,10],[15,20],[18,30],[33,40]
2  [5,4,3,2]
```

输出

说明

合并后：[1,10], [15,30], [33,40]，使用5, 3两个连接器连接后只剩下[1,40]。

用例2

输入

▼

Plain Text |

```
1 [1,2],[3,5],[7,10],[15,20],[30,100]
2 [5,4,3,2,1]
```

输出

说明

无重叠和相邻，使用1, 2, 5三个连接器连接后只剩下[1,20], [30,100]

题解

思路： 合并区间 + 贪心分配

1. 合并输入的区间,得到一组不重叠的区间。得到按照起始值升序排序不重叠的区间 `merge` 数组。
2. 计算相邻区间中间存在空闲距离, 存入 `dist` 数组。将 `dist` 数组进行排序。
3. 贪心将连接器距离大 贪心 分配给 `dist` 中大的间隔, 使用 `count` 统计可以分配的数量。
4. 结果就为 `merge.size() - count`

C++

```
1 #include<iostream>
2 #include<vector>
3 #include<string>
4 #include <utility>
5 #include <sstream>
6 #include<algorithm>
7 #include<list>
8 #include<queue>
9 #include<map>
10 #include<set>
11 using namespace std;
12
13 // 通用 split 函数
14 vector<string> split(const string& str, const string& delimiter) {
15     vector<string> result;
16     size_t start = 0;
17     size_t end = str.find(delimiter);
18     while (end != string::npos) {
19         result.push_back(str.substr(start, end - start));
20         start = end + delimiter.length();
21         end = str.find(delimiter, start);
22     }
23     // 添加最后一个部分
24     result.push_back(str.substr(start));
25     return result;
26 }
27
28
29 // 区域合并
30 vector<pair<int, int>> merged(vector<pair<int, int>> segs) {
31     // 按起点排序，方便合并
32     sort(segs.begin(), segs.end());
33     // 合并区间
34     vector<pair<int, int>> merged;
35     pair<int, int> cur = segs[0];
36     int n = segs.size();
37     for (int i = 1; i < n; i++) {
38         if (segs[i].first <= cur.second) {
39             cur.second = max(cur.second, segs[i].second);
40         } else {
41             merged.push_back(cur);
42             cur = segs[i];
43         }
44     }
45     merged.push_back(cur);
```

```

46     return merged;
47 }
48
49 int main() {
50     string input1, input2;
51     getline(cin, input1);
52     getline(cin, input2);
53     // 存储区间
54     vector<pair<int, int>> ans;
55
56     // 处理区间输入字符串 [18,30], [33,40] => 18,30], [33,40
57     input1 = input1.substr(1, input1.size() - 2);
58     vector<string> tmpArr = split(input1, "[", "]");
59     for (int i = 0; i < tmpArr.size(); i++) {
60         string tmp = tmpArr[i];
61         vector<string> startAndEnd = split(tmp, ",");
62         int start = stoi(startAndEnd[0]);
63         int end = stoi(startAndEnd[1]);
64         ans.push_back({start, end});
65     }
66
67     // 处理连接器输入字符串
68     input2 = input2.substr(1, input2.size() - 2);
69     vector<string> joinStrArr = split(input2, ",");
70     vector<int> joinArr;
71     for (int i = 0; i < joinStrArr.size(); i++) {
72         joinArr.push_back(stoi(joinStrArr[i]));
73     }
74
75     // 输入数据区间进行区间合并
76     vector<pair<int, int>> merge = merged(ans);
77
78     // 计算每个相邻区间之间的距离
79     vector<int> dist;
80     for (int i = 1; i < merge.size(); i++) {
81         dist.push_back(merge[i].first - merge[i-1].second);
82     }
83     sort(dist.begin(), dist.end());
84     sort(joinArr.begin(), joinArr.end());
85
86
87     int i = dist.size() - 1, j = joinArr.size() - 1;
88     // 合并次数
89     int count = 0;
90     // 贪心优先把连接器数量大的分配给距离大
91     while (i >= 0 && j >= 0) {
92         if (dist[i] <= joinArr[j]) {
93             count++;

```

```
94         i--;
95         j--;
96     } else {
97         i--;
98     }
99 }
100 // 区间数 - 合并
101 cout << dist.size() + 1 - count;
102 return 0;
103 }
```

JAVA

```
1 import java.util.*;
2
3 public class Main {
4     // 区间合并函数
5     public static List<int[]> merged(List<int[]> segs) {
6         // 按起点排序
7         segs.sort(Comparator.comparingInt(a -> a[0]));
8         List<int[]> merged = new ArrayList<>();
9         int[] cur = segs.get(0);
10        int n = segs.size();
11        for (int i = 1; i < n; i++) {
12            if (segs.get(i)[0] <= cur[1]) {
13                cur[1] = Math.max(cur[1], segs.get(i)[1]);
14            } else {
15                merged.add(cur);
16                cur = segs.get(i);
17            }
18        }
19        merged.add(cur);
20        return merged;
21    }
22
23    public static void main(String[] args) {
24        Scanner sc = new Scanner(System.in);
25        String input1 = sc.nextLine();
26        String input2 = sc.nextLine();
27
28        // 去除开头和结尾的中括号
29        input1 = input1.substring(1, input1.length() - 1);
30        String[] tmpArr = input1.split("\\],\\[" );
31
32        // 解析区间
33        List<int[]> ans = new ArrayList<>();
34        for (String s : tmpArr) {
35            String[] parts = s.split(",");
36            int start = Integer.parseInt(parts[0]);
37            int end = Integer.parseInt(parts[1]);
38            ans.add(new int[]{start, end});
39        }
40
41        // 解析连接器
42        input2 = input2.substring(1, input2.length() - 1);
43        String[] joinStrArr = input2.split(",");
44        List<Integer> joinArr = new ArrayList<>();
45        for (String s : joinStrArr) {
```

```
46         joinArr.add(Integer.parseInt(s));
47     }
48
49     // 合并区间
50     List<int[]> merge = merged(ans);
51
52     // 计算每对相邻区间之间的间距
53     List<Integer> dist = new ArrayList<>();
54     for (int i = 1; i < merge.size(); i++) {
55         dist.add(merge.get(i)[0] - merge.get(i - 1)[1]);
56     }
57
58     // 排序准备贪心合并
59     Collections.sort(dist);
60     Collections.sort(joinArr);
61
62     int i = dist.size() - 1, j = joinArr.size() - 1;
63     int count = 0;
64     while (i >= 0 && j >= 0) {
65         if (dist.get(i) <= joinArr.get(j)) {
66             count++;
67             i--;
68             j--;
69         } else {
70             i--;
71         }
72     }
73
74     // 原始段数 - 合并次数
75     System.out.println(dist.size() + 1 - count);
76 }
77 }
```

Python

```
1 def merged(segs):
2     segs.sort()
3     merged_list = []
4     cur = segs[0]
5     for i in range(1, len(segs)):
6         if segs[i][0] <= cur[1]:
7             cur[1] = max(cur[1], segs[i][1])
8         else:
9             merged_list.append(cur)
10            cur = segs[i]
11    merged_list.append(cur)
12    return merged_list
13
14
15 input1 = input().strip()
16 input2 = input().strip()
17
18
19 input1 = input1[1:-1]
20 segments = input1.split(",") , [")
21 ans = []
22 for s in segments:
23     start, end = map(int, s.split(','))
24     ans.append([start, end])
25
26
27 input2 = input2[1:-1]
28 joinArr = list(map(int, input2.split(',')))
29
30
31 merge = merged(ans)
32
33
34 dist = []
35 for i in range(1, len(merge)):
36     dist.append(merge[i][0] - merge[i-1][1])
37
38 dist.sort()
39 joinArr.sort()
40
41
42 i, j = len(dist) - 1, len(joinArr) - 1
43 count = 0
44 while i >= 0 and j >= 0:
45     if dist[i] <= joinArr[j]:
```

```
46         count += 1
47         i -= 1
48         j -= 1
49     else:
50         i -= 1
51
52
53 print(len(dist) + 1 - count)
```

JavaScript

```
1 const readline = require("readline");
2 const rl = readline.createInterface({
3     input: process.stdin,
4     output: process.stdout
5 });
6
7 let inputs = [];
8 rl.on("line", line => {
9     inputs.push(line.trim());
10    if (inputs.length === 2) rl.close();
11 });
12
13 rl.on("close", () => {
14     let input1 = inputs[0];
15     let input2 = inputs[1];
16
17     input1 = input1.slice(1, -1);
18     const segStrs = input1.split("[", "]");
19     let segments = segStrs.map(s => s.split(",")).map(Number));
20
21
22     segments.sort((a, b) => a[0] - b[0]);
23     let merged = [];
24     let cur = segments[0];
25     for (let i = 1; i < segments.length; i++) {
26         if (segments[i][0] <= cur[1]) {
27             cur[1] = Math.max(cur[1], segments[i][1]);
28         } else {
29             merged.push(cur);
30             cur = segments[i];
31         }
32     }
33 }
34 merged.push(cur);
35
36
37 input2 = input2.slice(1, -1);
38 let joinArr = input2.split(",").map(Number);
39
40
41 let dist = [];
42 for (let i = 1; i < merged.length; i++) {
43     dist.push(merged[i][0] - merged[i - 1][1]);
44 }
45
```

```
46     dist.sort((a, b) => a - b);
47     joinArr.sort((a, b) => a - b);
48
49     let i = dist.length - 1, j = joinArr.length - 1, count = 0;
50     while (i >= 0 && j >= 0) {
51         if (dist[i] <= joinArr[j]) {
52             count++;
53             i--;
54             j--;
55         } else {
56             i--;
57         }
58     }
59
60     console.log(dist.length + 1 - count);
61 });
});
```

Go

```
1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "os"
7     "sort"
8     "strconv"
9     "strings"
10    )
11
12
13 func merged(segs [][][2]int) [][]int {
14     sort.Slice(segs, func(i, j int) bool {
15         return segs[i][0] < segs[j][0]
16     })
17     var result [][]int
18     cur := segs[0]
19     for i := 1; i < len(segs); i++ {
20         if segs[i][0] <= cur[1] {
21             if segs[i][1] > cur[1] {
22                 cur[1] = segs[i][1]
23             }
24         } else {
25             result = append(result, cur)
26             cur = segs[i]
27         }
28     }
29     result = append(result, cur)
30     return result
31 }
32
33 func main() {
34     reader := bufio.NewReader(os.Stdin)
35     input1, _ := reader.ReadString('\n')
36     input2, _ := reader.ReadString('\n')
37     input1 = strings.TrimSpace(input1)
38     input2 = strings.TrimSpace(input2)
39
40
41     input1 = input1[1 : len(input1)-1]
42     parts := strings.Split(input1, "],[")
43     var segments [][]int
44     for _, s := range parts {
45         nums := strings.Split(s, ",")
```

```

46     start, _ := strconv.Atoi(nums[0])
47     end, _ := strconv.Atoi(nums[1])
48     segments = append(segments, [2]int{start, end})
49 }
50
51
52     input2 = input2[1 : len(input2)-1]
53     joinStrs := strings.Split(input2, ",")
54     var joinArr []int
55     for _, s := range joinStrs {
56         num, _ := strconv.Atoi(s)
57         joinArr = append(joinArr, num)
58     }
59
60
61     mergedSegs := merged(segments)
62
63
64     var dist []int
65     for i := 1; i < len(mergedSegs); i++ {
66         d := mergedSegs[i][0] - mergedSegs[i-1][1]
67         dist = append(dist, d)
68     }
69
70     sort.Ints(dist)
71     sort.Ints(joinArr)
72
73     i, j := len(dist)-1, len(joinArr)-1
74     count := 0
75     for i >= 0 && j >= 0 {
76         if dist[i] <= joinArr[j] {
77             count++
78             i--
79             j--
80         } else {
81             i--
82         }
83     }
84
85
86     fmt.Println(len(dist) + 1 - count)
87 }
```

| 来自: 华为OD 机试 2025 B卷 – 区间连接器 (C++ & Python & JAVA & JS & GO)-CSDN博客

华为OD机试 2025 B卷 - 基站维修工程师 (C++ & Python & JAVA & JS)

基站维修工程师

华为OD机试真题目录点击查看: [华为OD机试2025B卷真题题库目录 | 机考题库 + 算法考点详解](#)

华为OD机试2025B卷 200分题型

题目描述

小王是一名基站维护工程师，负责某区域的基站维护。某地方有 n 个基站($1 < n < 10$)，已知各基站之间的距离 $s(0 < s < 500)$ ，并且基站 x 到基站 y 的距离，与基站 y 到基站 x 的距离并不一定会相同。小王从基站 1 出发，途经每个基站 1 次，然后返回基站 1，需要请你为他选择一条距离最短的路

输入描述

站点数n和各站点之间的距离(均为整数)，如：3 {站点数} 0 2 1 {站点1到各站点的路程} 1 0 2 {站点2到各站点的路程} 2 1 0 {站点3到各站点的路程}

输出描述

最短路程的数值

用例1

输入

输出

用例2

输入

▼

Plain Text |

```
1 4
2 0 2 1 3
3 1 0 2 5
4 2 1 0 4
5 3 2 6 0
```

输出

题解

思路： 递归回溯

1. 本题数据量比较小站点数量 `1 <= n <= 10`，可以直接使用 递归回溯 枚举出所有从1号站出发访问到所有站所有可能路径。
2. 计算所有 (可能路径的距离 + 对应终点站回到起点站的距离)，结果就为其中的最小值。

小经验：建议看题的时候，终点关注题目给出的数据量(不同数据范围，要求的**时间复杂度**不同)，小数据量的题，一般采用枚举方式就能全部通过，。

C++

```
1 #include<iostream>
2 #include<vector>
3 #include<string>
4 #include <utility>
5 #include <sstream>
6 #include<algorithm>
7 #include<list>
8 #include<queue>
9 #include<climits>
10 using namespace std;
11
12
13 // 递归生成所有可能的路径
14 void dfs(int n, vector<vector<int>> &distance, vector<int> &currentPath, v
15   ector<vector<int>> & paths, vector<bool> &visited) {
16     if (currentPath.size() == n - 1) {
17       paths.push_back(vector<int>(currentPath.begin(), currentPath.end
18         ()));
19       return;
20     }
21     for (int i = 1; i < n; i++) {
22       if (!visited[i]) {
23         currentPath.push_back(i);
24         visited[i] = true;
25
26         dfs(n, distance, currentPath, paths, visited);
27         // 递归回溯
28         visited[i] = false;
29         currentPath.pop_back();
30       }
31     }
32
33 int getMinPath(int n, vector<vector<int>> distance) {
34   vector<bool> visited(n, false);
35   vector<vector<int>> paths;
36   vector<int> currentPath;
37   dfs(n, distance, currentPath, paths, visited);
38   int res = INT_MAX;
39   // 求出所有路径中的最小值
40   for (int i = 0; i < paths.size(); i++) {
41     vector<int> path = paths[i];
42     int tmp = distance[0][path[0]];
43     for (int j = 0; j < path.size() - 1; j++) {
```

```
44         tmp += distance[path[j]][path[j+1]];
45     }
46     // 返回1号站
47     tmp += distance[path[path.size()-1]][0];
48     res = min(res, tmp);
49 }
50 return res;
51 }
52
53
54
55 int main() {
56     int n;
57     cin >> n;
58     vector<vector<int>> distance(n, vector<int>(n));
59     for (int i = 0; i < n; i++) {
60         for (int j = 0; j < n; j++) {
61             int tmp;
62             cin >> tmp;
63             distance[i][j] = tmp;
64         }
65     }
66     int res = getMinPath(n, distance);
67     cout << res;
68     return 0;
69 }
```

JAVA

```
1 import java.util.*;
2
3 public class Main {
4     // 递归生成所有可能的路径
5     private static void dfs(int n, int[][] distance, List<Integer> current
6     Path, List<List<Integer>> paths, boolean[] visited) {
7         if (currentPath.size() == n - 1) {
8             paths.add(new ArrayList<>(currentPath));
9             return;
10        }
11        for (int i = 1; i < n; i++) {
12            if (!visited[i]) {
13                currentPath.add(i);
14                visited[i] = true;
15
16                dfs(n, distance, currentPath, paths, visited);
17
18                // 递归回溯
19                visited[i] = false;
20                currentPath.remove(currentPath.size() - 1);
21            }
22        }
23    }
24
25    private static int getMinPath(int n, int[][] distance) {
26        boolean[] visited = new boolean[n];
27        List<List<Integer>> paths = new ArrayList<>();
28        List<Integer> currentPath = new ArrayList<>();
29        dfs(n, distance, currentPath, paths, visited);
30
31        int res = Integer.MAX_VALUE;
32        // 求出所有路径中的最小值
33        for (List<Integer> path : paths) {
34            int tmp = distance[0][path.get(0)];
35            for (int j = 0; j < path.size() - 1; j++) {
36                tmp += distance[path.get(j)][path.get(j + 1)];
37            }
38            // 返回1号站
39            tmp += distance[path.get(path.size() - 1)][0];
40            res = Math.min(res, tmp);
41        }
42        return res;
43    }
44
45    public static void main(String[] args) {
```

```
45     Scanner scanner = new Scanner(System.in);
46     int n = scanner.nextInt();
47     int[][] distance = new int[n][n];
48
49     for (int i = 0; i < n; i++) {
50         for (int j = 0; j < n; j++) {
51             distance[i][j] = scanner.nextInt();
52         }
53     }
54     scanner.close();
55
56     int res = getMinPath(n, distance);
57     System.out.println(res);
58 }
59 }
```

Python

```
1 import sys
2 import itertools
3
4
5 def get_min_path(n, distance):
6     res = float('inf')
7
8
9     for path in itertools.permutations(range(1, n)):
10         tmp = distance[0][path[0]]
11         for i in range(len(path) - 1):
12             tmp += distance[path[i]][path[i + 1]]
13         tmp += distance[path[-1]][0]
14         res = min(res, tmp)
15
16     return res
17
18 def main():
19     n = int(sys.stdin.readline().strip())
20     distance = [list(map(int, sys.stdin.readline().split())) for _ in range(n)]
21
22     res = get_min_path(n, distance)
23     print(res)
24
25 if __name__ == "__main__":
26     main()
```

JavaScript

```
1 const readline = require("readline");
2
3
4 function getMinPath(n, distance) {
5     let res = Infinity;
6
7
8     function permute(arr, l, r) {
9         if (l === r) {
10             let tmp = distance[0][arr[0]];
11             for (let i = 0; i < arr.length - 1; i++) {
12                 tmp += distance[arr[i]][arr[i + 1]];
13             }
14
15             tmp += distance[arr[arr.length - 1]][0];
16             res = Math.min(res, tmp);
17             return;
18         }
19         for (let i = l; i <= r; i++) {
20             [arr[l], arr[i]] = [arr[i], arr[l]];
21             permute(arr, l + 1, r);
22             [arr[l], arr[i]] = [arr[i], arr[l]];
23         }
24     }
25
26     let nodes = Array.from({ length: n - 1 }, (_, i) => i + 1);
27     permute(nodes, 0, nodes.length - 1);
28
29     return res;
30 }
31
32 const rl = readline.createInterface({
33     input: process.stdin,
34     output: process.stdout
35 });
36
37 let input = [];
38 rl.on("line", (line) => {
39     input.push(line.trim());
40 }).on("close", () => {
41     let n = parseInt(input[0]);
42     let distance = input.slice(1).map(row => row.split(" ")).map(Number));
43
44     let res = getMinPath(n, distance);
45     console.log(res);
```

46 });

Go

```
1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "os"
7     "strconv"
8     "strings"
9 )
10
11
12 func getMinPath(n int, distance [][]int) int {
13     res := int(1e9)
14
15     nodes := make([]int, n-1)
16     for i := 1; i < n; i++ {
17         nodes[i-1] = i
18     }
19
20     var permute func([]int, int)
21     permute = func(arr []int, l int) {
22         if l == len(arr) {
23             tmp := distance[0][arr[0]]
24             for i := 0; i < len(arr)-1; i++ {
25                 tmp += distance[arr[i]][arr[i+1]]
26             }
27
28             tmp += distance[arr[len(arr)-1]][0]
29             if tmp < res {
30                 res = tmp
31             }
32             return
33         }
34         for i := l; i < len(arr); i++ {
35             arr[l], arr[i] = arr[i], arr[l]
36             permute(arr, l+1)
37             arr[l], arr[i] = arr[i], arr[l]
38         }
39     }
40 }
41
42     permute(nodes, 0)
43     return res
44 }
45 }
```

```
46 func main() {
47     scanner := bufio.NewScanner(os.Stdin)
48     scanner.Scan()
49     n, _ := strconv.Atoi(scanner.Text())
50
51     distance := make([][]int, n)
52     for i := 0; i < n; i++ {
53         scanner.Scan()
54         line := strings.Fields(scanner.Text())
55         distance[i] = make([]int, n)
56         for j := 0; j < n; j++ {
57             distance[i][j], _ = strconv.Atoi(line[j])
58         }
59     }
60
61     res := getMinPath(n, distance)
62     fmt.Println(res)
63 }
```

| 来自: 华为od机试 2025 B卷 – 基站维修工程师 (C++ & Python & JAVA & JS & GO)-CSDN博客

华为OD机试 2025 B卷 - We are a Team (C++ & Python & JAV)

We are a Team

华为OD机试真题目录点击查看: [华为OD机试2025B卷真题题库目录 | 机考题库 + 算法考点详解](#)

华为OD机试2025B卷 100分题型

题目描述

总共有 n 个人在机房，每个人有一个标号 ($1 \leq \text{标号} \leq n$)，他们分成了多个团队，需要你根据收到的 m 条消息判定指定的两个人是否在一个团队中，具体的：

1. 消息构成为 $a\ b\ c$ ，整数 a 、 b 分别代表两个人的标号，整数 c 代表指令
2. $c == 0$ 代表 a 和 b 在一个团队内
3. $c == 1$ 代表需要判定 a 和 b 的关系，如果 a 和 b 是一个团队，输出一行'we are a team'，如果不是，输出一行'we are not a team'
4. c 为其他值，或当前行 a 或 b 超出 $1 \sim n$ 的范围，输出'da pian zi'

输入描述

1. 第一行包含两个整数 $n, m (1 \leq n, m < 100000)$ ，分别表示有 n 个人和 m 条消息
2. 随后的 m 行，每行一条消息，消息格式为： $a\ b\ c (1 \leq a, b \leq n, 0 \leq c \leq 1)$

输出描述

1. $c == 1$ ，根据 a 和 b 是否在一个团队中输出一行字符串，在一个团队中输出'we are a team'，不在一个团队中输出'we are not a team'
2. c 为其他值，或当前行 a 或 b 的标号小于 1 或者大于 n 时，输出字符串'da pian zi'
3. 如果第一行 n 和 m 的值超出约定的范围时，输出字符串"Null"。

示例1

输入

▼

Plain Text |

```
1 5 7
2 1 2 0
3 4 5 0
4 2 3 0
5 1 2 1
6 2 3 1
7 4 5 1
8 1 5 1
```

输出

▼

Plain Text |

```
1 we are a team
2 we are a team
3 we are a team
4 we are not a team
```

示例2

输入

▼

Plain Text |

```
1 5 6
2 1 2 0
3 1 2 1
4 1 5 0
5 2 3 1
6 2 5 1
7 1 3 2
```

输出

▼

Plain Text |

```
1 we are a team
2 we are not a team
3 we are a team
4 da pian zi
```

题解

思路： 并查集

1. 初始化一个数组 `ans`，`ans[i]`中存储了 `i` 所处团队的队长的序号。团队的队长的为本团队中最小的序号的成员。初始假设所有人都是单独一个团队并为自己的队长，在代码中的体现为初始所有 `ans[i] = i`。
2. 将 `a` 和 `c` 加入一个团队相当于，将 `a` 所处的团队和 `b` 所处的团队进行合并称为一个团队。合并团队，合并之后的团队的队长为原始两个团队队长的序号较小的那个。
3. 判断两个人是否是一个团队就转换为两个人的队长是否相同。

并查集是一个比较基础且考取频率相对较高的算法，建议熟练掌握。并查集也是一个模板算法，代码套路比较固定。

C++ 源码实现

```
1 #include<iostream>
2 #include<vector>
3 using namespace std;
4
5
6 int find(int a, vector<int> &ans) {
7     if (ans[a] != a) {
8         ans[a] = find(ans[a], ans);
9     }
10    return ans[a];
11 }
12
13
14 void merge(int a, int b, vector<int> &ans) {
15     int rootA = find(a, ans);
16     int rootB = find(b, ans);
17     int newRoot = min(rootA, rootB);
18     ans[rootA] = newRoot;
19     ans[rootB] = newRoot;
20 }
21
22 int main() {
23     int n, m;
24     cin >> n >> m;
25
26     if (1 > n || m < 1 || n >= 100000 || m >= 100000) {
27         cout << "Null" << endl;
28         return 0;
29     }
30
31     vector<int> ans(n+1);
32
33     for (int i = 0; i <= n; i++) {
34         ans[i] = i;
35     }
36
37     while (m--) {
38         int a,b,c;
39         cin >> a >> b >> c;
40
41         if (a > n || b > n || a < 1 || b < 1) {
42             cout << "da pian zi" << endl;
43             continue;
44         }
45     }
```

```
46     if (c == 0) {
47
48         merge(a, b, ans);
49     } else if (c == 1) {
50         if (find(a, ans) == find(b, ans)) {
51             cout << "we are a team" << endl;
52         } else{
53             cout << "we are not a team" << endl;
54         }
55     } else {
56         cout << "da pian zi" << endl;
57     }
58 }
59
60 }
```

python实现源码

```
1  class UnionFind:
2      def __init__(self, n):
3          self.ans = list(range(n + 1))
4
5      def find(self, a):
6
7          if self.ans[a] != a:
8              self.ans[a] = self.find(self.ans[a])
9
10         return self.ans[a]
11
12     def merge(self, a, b):
13
14         root_a = self.find(a)
15         root_b = self.find(b)
16         new_root = min(root_a, root_b)
17         self.ans[root_a] = new_root
18         self.ans[root_b] = new_root
19
20
21     def main():
22
23         n, m = map(int, input().split())
24
25
26         if n < 1 or m < 1 or n >= 100000 or m >= 100000:
27             print("Null")
28             return
29
30
31         uf = UnionFind(n)
32
33         for _ in range(m):
34
35             inputs = input().split()
36             a, b, c = map(int, inputs)
37
38
39             if a > n or b > n or a < 1 or b < 1:
40                 print("da pian zi")
41                 continue
42
43             if c == 0:
44
45                 uf.merge(a, b)
```

```
46     elif c == 1:
47
48         if uf.find(a) == uf.find(b):
49             print("we are a team")
50         else:
51             print("we are not a team")
52     else:
53         print("da pian zi")
54
55
56 if __name__ == "__main__":
57     main()
```

JAVA实现源码

```
1 import java.util.*;
2
3
4 class Team {
5     private int[] ans;
6
7     public Team(int n) {
8
9         ans = new int[n + 1];
10        for (int i = 0; i <= n; i++) {
11            ans[i] = i;
12        }
13    }
14
15    public int find(int a) {
16
17        if (ans[a] != a) {
18            ans[a] = find(ans[a]);
19        }
20        return ans[a];
21    }
22
23    public void merge(int a, int b) {
24
25        int rootA = find(a);
26        int rootB = find(b);
27        int newRoot = Math.min(rootA, rootB);
28        ans[rootA] = newRoot;
29        ans[rootB] = newRoot;
30    }
31 }
32
33 public class Main {
34     public static void main(String[] args) {
35         Scanner scanner = new Scanner(System.in);
36
37
38         int n = scanner.nextInt();
39         int m = scanner.nextInt();
40
41
42         if (n < 1 || m < 1 || n >= 100000 || m >= 100000) {
43             System.out.println("Null");
44             scanner.close();
45             return;
```

```
46     }
47
48
49     Team uf = new Team(n);
50
51     for (int i = 0; i < m; i++) {
52
53         int a = scanner.nextInt();
54         int b = scanner.nextInt();
55         int c = scanner.nextInt();
56
57
58         if (a > n || b > n || a < 1 || b < 1) {
59             System.out.println("da pian zi");
60             continue;
61         }
62
63         if (c == 0) {
64
65             uf.merge(a, b);
66         } else if (c == 1) {
67
68             if (uf.find(a) == uf.find(b)) {
69                 System.out.println("we are a team");
70             } else {
71                 System.out.println("we are not a team");
72             }
73         } else {
74             System.out.println("da pian zi");
75         }
76     }
77
78     scanner.close();
79 }
80 }
```

go实现源码

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 type Team struct {
8     parent []int
9 }
10
11
12 func NewTeam(n int) *Team {
13     uf := &Team{
14         parent: make([]int, n+1),
15     }
16     for i := 0; i <= n; i++ {
17         uf.parent[i] = i
18     }
19     return uf
20 }
21
22
23 func (uf *Team) Find(a int) int {
24     if uf.parent[a] != a {
25         uf.parent[a] = uf.Find(uf.parent[a])
26     }
27     return uf.parent[a]
28 }
29
30
31 func (uf *Team) Merge(a, b int) {
32     rootA := uf.Find(a)
33     rootB := uf.Find(b)
34     newRoot := min(rootA, rootB)
35     uf.parent[rootA] = newRoot
36     uf.parent[rootB] = newRoot
37 }
38
39 func min(a, b int) int {
40     if a < b {
41         return a
42     }
43     return b
44 }
45
```

```
46 func main() {
47     var n, m int
48     fmt.Scan(&n, &m)
49
50
51     if n < 1 || m < 1 || n >= 100000 || m >= 100000 {
52         fmt.Println("Null")
53         return
54     }
55
56
57     uf := NewTeam(n)
58
59     for i := 0; i < m; i++ {
60         var a, b, c int
61         fmt.Scan(&a, &b, &c)
62
63
64         if a > n || b > n || a < 1 || b < 1 {
65             fmt.Println("da pian zi")
66             continue
67         }
68
69         if c == 0 {
70
71             uf.Merge(a, b)
72         } else if c == 1 {
73
74             if uf.Find(a) == uf.Find(b) {
75                 fmt.Println("we are a team")
76             } else {
77                 fmt.Println("we are not a team")
78             }
79         } else {
80             fmt.Println("da pian zi")
81         }
82     }
83 }
```

来自: 华为OD机试 2025 B卷 – We are a Team (C++ & Python & JAVA & JS & GO)-CSDN博客