## Spring循环依赖

基于构造器的循环依赖，就更不用说了，官方文档都摊牌了，你想让构造器注入支持循环依赖，是不存在的，不如把代码改了。

那么默认单例的属性注入场景，Spring是如何支持循环依赖的？

先说明前提：原型(Prototype)的场景是不支持循环依赖的，通常会走到AbstractBeanFactory类中下面的判断，抛出异常。

**if** (isPrototypeCurrentlyInCreation(beanName)) {  
  **throw** **new** BeanCurrentlyInCreationException(beanName);  
}

原因很好理解，创建新的A时，发现要注入原型字段B，又创建新的B发现要注入原型字段A...

# =====springboot=====

## springboot启动过程中做了哪些事情？

<https://www.jianshu.com/p/dc12081b3598>

## 注解 @spring boot Application说明？

1. Springboot如何判断当前应用是否时web应用？
2. <https://www.jianshu.com/p/dc12081b3598>

4、Spring boot整合jsp的流程，需要注意哪些点

# =====**SpringCloud**=====

1、SpringCloud和dubbo的区别

2、项目中用到了哪些组件

3、eureka的原理，如何保证高可用性，和Zookeeper有什么区别

4、feign如何调用的

5、处理生产环境上配置生效问题

6、hystrix的降级策略有哪些

7、Springcloud eureka是如何注册服务、如何监测心跳的，它注册的流程是怎么样的

8、在分布式环境中如何快速发现某一台服务有问题

9、分布式集群系统对外提供接口的时候如何验证对方的身份

## 10、Eureka和zookeeper作为注册中心有什么区别

<https://blog.csdn.net/weixin_33725126/article/details/94308610?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase>

1.ZooKeeper保证的是CP,Eureka保证的是AP

ZooKeeper在选举期间注册服务瘫痪,虽然服务最终会恢复,但是选举期间不可用的  
Eureka各个节点是平等关系,只要有一台Eureka就可以保证服务可用,而查询到的数据并不是最新的自我保护机制会导致

Eureka不再从注册列表移除因长时间没收到心跳而应该过期的服务  
Eureka仍然能够接受新服务的注册和查询请求,但是不会被同步到其他节点(高可用)  
当网络稳定时,当前实例新的注册信息会被同步到其他节点中(最终一致性)  
Eureka可以很好的应对因网络故障导致部分节点失去联系的情况,而不会像ZooKeeper一样使得整个注册系统瘫痪

1. ZooKeeper有Leader和Follower角色,Eureka各个节点平等  
   3.ZooKeeper采用过半数存活原则,Eureka采用自我保护机制解决分区问题  
   4.Eureka本质上是一个工程,而ZooKeeper只是一个进程

# =====分布式锁=====

## 我们真的需要锁么？

需要锁的条件：

1. 多任务环境下。（进程，线程）

2. 任务都对同一共享资源进行写操作。

3. 对资源的访问是互斥的。

**操作周期：**

1. 竞争锁。获取锁后才能对资源进行操作。

2. 占有锁。操作中。

3. 其他竞争者，任务阻塞。

4. 占有锁者，释放锁。继续从1开始。

**JVM 锁 解决不了分布式环境中的加锁问题。**

分布式锁应用场景：服务集群，比如N个订单服务，接受到大量司机的发送的对一个订单的抢单请求。如果是单个服务，可以用jvm锁控制，但是服务集群，jvm 就不行了。因为不在一个jvm中。

### 应该具备的条件

在分析分布式锁的三种实现方式之前，先了解一下分布式锁应该具备哪些条件：

1、在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行；   
2、高可用的获取锁与释放锁；   
3、高性能的获取锁与释放锁；   
4、具备可重入特性；   
5、具备锁失效机制，防止死锁；   
6、具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败。

### 三种实现：

基于数据库实现分布式锁；   
基于缓存（Redis等）实现分布式锁；   
基于Zookeeper实现分布式锁；

### 基于数据库的实现方式

基于数据库的实现方式的核心思想是：在数据库中创建一个表，表中包含**方法名**等字段，并在**方法名字段上创建唯一索引**，想要执行某个方法，就使用这个方法名向表中插入数据，成功插入则获取锁，执行完成后删除对应的行数据释放锁。

<https://mp.weixin.qq.com/s/gVdUNYQjVZFI78oTNVOgHg>

问题：

1、如果中间出异常了，如何释放锁，用存储过程，还是可以解决。

2、mysql 并发是由限制的。不适合高并发场景。

### Redis实现

redis：内存存储的数据结构服务器，内存数据库。可用于：数据库，高速缓存，消息队列。采用单线程模型，并发能力强大。10万并发没问题。

分布锁知识：

redis的单进程单线程。

缓存有效期。有效期到，删除数据。

setnx。当key存在，不做任何操作，key不存在，才设置。

**代码：**

|  |
| --- |
| *//生成key* String lock = **"order\_"**+(orderId+**""**);  //  **boolean** lockStatus = **stringRedisTemplate**.opsForValue().setIfAbsent(lock.intern(), driverId+**""**, 30L, TimeUnit.***SECONDS***);  **if**(!lockStatus) {  **return null**;  }    **try** {  System.***err***.println(**"司机:"**+driverId+**" 执行抢单逻辑"**);  **boolean** b = **orderService**.grab(orderId, driverId);  **if**(b) {  System.***err***.println(**"---------->司机:"**+driverId+**" 抢单成功"**);  }**else** {  System.***err***.println(**"---------->司机:"**+driverId+**" 抢单失败"**);  }  } **finally** {  */\*\*  \* 这种释放锁有，可能释放了别人的锁。  \*/ // stringRedisTemplate.delete(lock.intern());  /\*\*  \* 下面代码避免释放别人的锁  \*/* **if**((driverId+**""**).equals(**stringRedisTemplate**.opsForValue().get(lock.intern()))) {  **stringRedisTemplate**.delete(lock.intern());  }  } |

注意点：

**1，设置超时时间**

加锁时候，一定要设置过期时间。设置过期时间和set方法一定在一行完成，如果在2行，加了锁执行完后，设置过期时间没执行就宕机了，这个锁就没法释放了。

1. 不要释放了别人的锁

必须给key设置一个value。value保证每个线程不一样。如果value在每个线程间一样。会发生 误解锁的问题。

释放锁一定要释放自己的 锁，取出锁，判断一下是不是自己加的锁。

例外场景：业务代码需要执行5秒，锁过期时间是3秒，线程1先抢到锁，3秒后锁释放了，线程2又抢到锁，又过了2秒，线程1直接去删除锁，会删掉线程2的锁。

1. 释放时间长短问题

如果业务正常执行，没问题，可以开个子线程，比如原来过期时间是Ns，开个子线程，每隔N/3秒去检查一下锁是否释放，没有释放就续约N秒

存在问题：

1. redis故障问题。

如果redis故障了，所有客户端无法获取锁，服务变得不可用。为了提高可用性。我们给redis 配置主从。当master不可用时，系统切换到slave，由于Redis的主从复制（replication）是异步的，这可能导致丧失锁的安全性。

1.客户端1从Master获取了锁。

2.Master宕机了，存储锁的key还没有来得及同步到Slave上。

3.Slave升级为Master。

4.客户端2从新的Master获取到了对应同一个资源的锁。

客户端1和客户端2同时持有了同一个资源的锁。锁的安全性被打破。

2. 如果在过期时间内，程序没有执行完，是不能让key过期的，所以要延时。

这个算法中出现的锁的有效时间(lock validity time)，设置成多少合适呢？如果设置太短的话，锁就有可能在客户端完成对于共享资源的访问之前过期，从而失去保护；如果设置太长的话，一旦某个持有锁的客户端释放锁失败，那么就会导致所有其它客户端都无法获取锁，从而长时间内无法正常工作。应该设置稍微短一些，如果线程持有锁，开启线程自动延长有效期。

### -----> Redisson实现

Redlock算法

它基于N个完全独立的Redis节点（通常N=5）。独立的，不是集群，不是哨兵 不是主从

自动续约超时时间，

运行Redlock算法的客户端依次执行下面各个步骤，来完成 获取锁 的操作：

1. 获取当前时间（毫秒数）。

2. 按顺序依次向N个Redis节点执行【获取锁】 的操作。

这个获取操作跟前面基于单Redis节点的 \*\*获取锁\*\* 的过程相同，包含value driverId ，也包含过期时间(比如 `PX 30000` ，即锁的有效时间)。为了保证在某个Redis节点不可用的时候算法能够继续运行，这个 \*\*获取锁\*\* 的操作还有一个超时时间(time out)，它要远小于锁的有效时间（几十毫秒量级）。客户端在向某个Redis节点获取锁失败以后，应该立即尝试下一个Redis节点。这里的失败，应该包含任何类型的失败，比如该Redis节点不可用，或者该Redis节点上的锁已经被其它客户端持有（注：Redlock原文中这里只提到了Redis节点不可用的情况，但也应该包含其它的失败情况）。

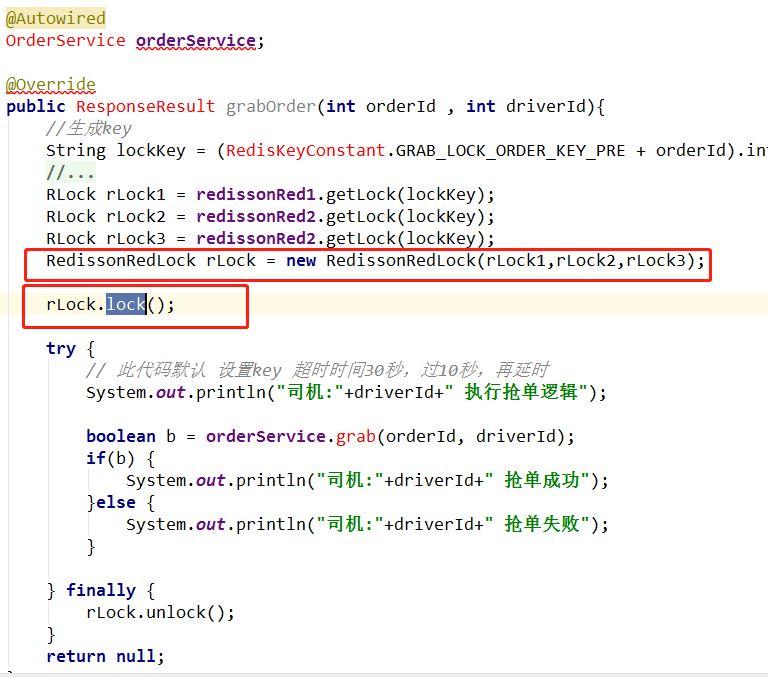
3. 如果从 >= N/2+1个节点成功获取到锁，且没超过过期时间，就算获取锁成功

计算整个获取锁的过程总共消耗了多长时间，计算方法是用当前时间减去第1步记录的时间。如果客户端从大多数Redis节点（>= N/2+1）成功获取到了锁，并且获取锁总共消耗的时间没有超过锁的有效时间(lock validity time)，那么这时客户端才认为最终获取锁成功；否则，认为最终获取锁失败。

4. 如果最终获取锁成功了，那么这个锁的有效时间应该重新计算，它等于最初的锁的有效时间减去第3步计算出来的获取锁消耗的时间。

5. 如果最终获取锁失败了（可能由于获取到锁的Redis节点个数少于N/2+1，或者整个获取锁的过程消耗的时间超过了锁的最初有效时间），那么客户端应该立即向所有Redis节点发起 \*\*释放锁\*\* 的操作（即前面介绍的Redis Lua脚本）。

当然，上面描述的只是 获取锁 的过程，而 释放锁 的过程比较简单：客户端向所有Redis节点发起 释放锁 的操作，不管这些节点当时在获取锁的时候成功与否。



问题：

由于N个Redis节点中的大多数能正常工作就能保证Redlock正常工作，因此理论上它的可用性更高。我们前面讨论的单Redis节点的分布式锁在failover的时候锁失效的问题，在Redlock中不存在了，但如果有节点发生崩溃重启，还是会对锁的安全性有影响的。具体的影响程度跟Redis对数据的持久化程度有关。

假设一共有5个Redis节点：A, B, C, D, E。设想发生了如下的事件序列：

1. 客户端1成功锁住了A, B, C， \*\*获取锁\*\* 成功（但D和E没有锁住）。

2. 节点C崩溃重启了，但客户端1在C上加的锁没有持久化下来，丢失了。

3. 节点C重启后，客户端2锁住了C, D, E， \*\*获取锁\*\* 成功。

这样，客户端1和客户端2同时获得了锁（针对同一资源）。

在默认情况下，Redis的AOF持久化方式是每秒写一次磁盘（即执行fsync），因此最坏情况下可能丢失1秒的数据。为了尽可能不丢数据，Redis允许设置成每次修改数据都进行fsync，但这会降低性能。当然，即使执行了fsync也仍然有可能丢失数据（这取决于系统而不是Redis的实现）。所以，上面分析的由于节点重启引发的锁失效问题，总是有可能出现的。为了应对这一问题，antirez又提出了 延迟重启 (delayed restarts)的概念。也就是说，一个节点崩溃后，先不立即重启它，而是等待一段时间再重启，这段时间应该大于锁的有效时间(lock validity time)。这样的话，这个节点在重启前所参与的锁都会过期，它在重启后就不会对现有的锁造成影响。

关于Redlock还有一点细节值得拿出来分析一下：在最后 释放锁 的时候，antirez在算法描述中特别强调，客户端应该向所有Redis节点发起 释放锁 的操作。也就是说，即使当时向某个节点获取锁没有成功，在释放锁的时候也不应该漏掉这个节点。这是为什么呢？设想这样一种情况，客户端发给某个Redis节点的 获取锁 的请求成功到达了该Redis节点，这个节点也成功执行了 `SET`操作，但是它返回给客户端的响应包却丢失了。这在客户端看来，获取锁的请求由于超时而失败了，但在Redis这边看来，加锁已经成功了。因此，释放锁的时候，客户端也应该对当时获取锁失败的那些Redis节点同样发起请求。实际上，这种情况在异步通信模型中是有可能发生的：客户端向服务器通信是正常的，但反方向却是有问题的。

### 基于ZooKeeper的实现方式

ZooKeeper是一个为分布式应用提供一致性服务的开源组件，它内部是一个分层的文件系统目录树结构，规定同一个目录下只能有一个唯一文件名。基于ZooKeeper实现分布式锁的步骤如下：

（1）创建一个目录mylock；

（2）线程A想获取锁就在mylock目录下创建临时顺序节点；

（3）获取mylock目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则说明当前线程顺序号最小，获得锁；

（4）线程B获取所有节点，判断自己不是最小节点，设置监听比自己次小的节点；

（5）线程A处理完，删除自己的节点，线程B监听到变更事件，判断自己是不是最小的节点，如果是则获得锁。

这里推荐一个Apache的开源库Curator，它是一个ZooKeeper客户端，Curator提供的InterProcessMutex是分布式锁的实现，acquire方法用于获取锁，release方法用于释放锁。

优点：具备高可用、可重入、阻塞锁特性，可解决失效死锁问题。

缺点：因为需要频繁的创建和删除节点，性能上不如Redis方式。

# =====分布式事务=====