# 持续更新的分布式组件



# 分布式异步方法

## 源起场景

大量数据插入,导致延时很高,但是任务本本身不需要实时响应.
比如应用要插入100000条数据,但是如果直接写入,数据库处理要一段时间,而用户并不需要立刻看到结果.
于是有了分布式异步方法解决方案.

> 在本地调用方法,会跳过该方法,并且往消息队列发送任务消息,然后消费端处理该任务即可;
> 从任务执行的角度,该任务是异步,并且是无响应的(对于调用方法).
> 消费成功的保障是消息队列,一旦消费失败,则重新消费,直至成功.(100%成功保障)

## 概念介绍

就是将**任务异步无回调执行,保证100%成功**.

## 接入

### 普通方法接入

在需要的method上加上@MqMethond注解,并且填上对应参数即可.

```
必填参数--gitCloneUrl
当前项目的git url(https或者git都行),所有method只需写一次就行

必填参数--taskName
任务名称,每个任务名必须不一样
```

### 消费失败处理

方法执行失败直接抛异常即可,会重新执行该方法.
但是防止因为本身的代码逻辑问题导致无限消费,用户最好自己处理,这里的消费成功保证更多是的解决网络问题导致的消费失败.

# 分布式实时对象

## 源起场景

游戏中实时对战,一方数据更新,另一方需要收到实时更新消息.
双方的数据保持始终一致性.
分布式实时对象解决方案于是产生了

> 分布式共享对象,其实就是共享同一个rediss对象
> 对java对象实现代理,每次set/get都会映射一个rediss操作,这样保证对象数据的实时性.
> 实时更新消息其实就是订阅/发布模型
> 每次set方法,都会发送一个更新消息,其余订阅者收信息即可.
> 注:开发无需关注发送消息的实现,只需关注收消息即可.

## 概念介绍

概念类似多线程的共享对象,只不过概念放大,是分布式环境的共享对象.

但是单纯的共享对象是没有更新回调的,于是引入更新回调的接口.该接口自动接收更新对象信息,便于应用感知.

# 接入

## 将一个普通的Java对象“升级”成了一个分布式实时对象

需要为一个类添加一个@REntity注释，然后再为其中的一个字段添加一个@RId注释即可。

```
ds
@REntity
public class MyLiveObject {
    @RId
    private String name;
    //其他字段
    ...
    ...

    //get和set方法
    ...
    ...
}
```

## 更新回调---实现DCallback接口

该接口会将更新的类名以及唯一id传回来
其余使用方式和redisson的分布式实时对象一样.

## 使用样例

```
//首先获取服务实例
RLiveObjectService service = redisson.getLiveObjectService();

//通过服务实例构造RLO实例
MyObject standardObject1 = new MyObject();
standardObject1.setName("liveObjectId");
MyObject liveObject1 = service.<MyObject, String>persist(standardObject1);
//服务实例会首先通过单一参数为条件查找构造函数，如果能找到就尝试采用"liveObjectId"作为参数
//来构造实例，如果没有找到就采用默认构造函数，然后调用setName("liveObjectId")赋值，最后再
//返回对象。
```

```
liveObject1.setValue("abc");
//“abc”作为字段值，储存在Redis里，而不是在JVM内存堆里。（虽然它会在字符串池里出现，但是没有被
//对象引用，因此不会影响垃圾回收。）

System.out.println(liveObject1.getValue());
//控制台输出内容和上面一样，但值是从Redis里获取出来的。
```

### 注意点

**手动new的对象不是代理对象,service.get返回的对象才是**

## 其余参考文档

https://github.com/redisson/redisson/wiki/9.-%E5%88%86%E5%B8%83%E5%BC%8F%E6%9C%8D%E5%8A%A1#92-%E5%88%86%E5%B8%83%E5%BC%8F%E5%AE%9E%E6%97%B6%E5%AF%B9%E8%B1%A1live-object%E6%9C%8D%E5%8A%A1

# 分布式锁

直接使用redisson的分布式锁

## 源起

分布式环境下,锁的使用

## 概念介绍

本地锁的升级版,jdk锁接口实现,体验和使用本地锁一样

## 公平锁（Fair Lock）

基于Redis的Redisson分布式可重入公平锁也是实现了java.util.concurrent.locks.Lock接口的一种RLock对象。它保证了当多个Redisson客户端线程同时请求加锁时，优先分配给先发出请求的线程。

```
RLock fairLock = redisson.getFairLock("anyLock");
// 最常见的使用方法
fairLock.lock();
```

大家都知道，如果负责储存这个分布式锁的Redis节点宕机以后，而且这个锁正好处于锁住的状态时，这个锁会出现锁死的状态。为了避免这种情况的发生，Redisson内部提供了一个监控锁的看门狗，它的作用是在Redisson实例被关闭前，不断的延长锁的有效期。默认情况下，看门狗的检查锁的超时时间是30秒钟，也可以通过修改Config.lockWatchdogTimeout来另行指定。

另外Redisson还通过加锁的方法提供了leaseTime的参数来指定加锁的时间。超过这个时间后锁便自动解开了。

```
// 10秒钟以后自动解锁
// 无需调用unlock方法手动解锁
fairLock.lock(10, TimeUnit.SECONDS);

// 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
boolean res = fairLock.tryLock(100, 10, TimeUnit.SECONDS);
...
fairLock.unlock();
```

Redisson同时还为分布式可重入公平锁提供了异步执行的相关方法：

```
RLock fairLock = redisson.getFairLock("anyLock");
fairLock.lockAsync();
fairLock.lockAsync(10, TimeUnit.SECONDS);
Future<Boolean> res = fairLock.tryLockAsync(100, 10, TimeUnit.SECONDS);
```

## 其余锁文档地址

https://github.com/redisson/redisson/wiki/8.-%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81%E5%92%8C%E5%90%8C%E6%AD%A5%E5%99%A8