分布式读写锁使用场景和ZK实现方案

读写锁介绍

读写锁允许同一时刻被多个读线程访问，但是在写线程访问时，所有的读线程和其他的写线程都会被阻塞

1. 写锁：互斥，同一时刻，只有一个进程，持有写锁
2. 读锁：共享，同一时刻，可以多个进程，持有读锁
3. 组合 ：  
   * 读锁：所有的「写锁」都失效时，可以「加读锁」，读锁共享
   * 写锁：所有的「读锁」和「写锁」都失效时，可以「加写锁」，写锁互斥，跟所有的读写动作都互斥

读写锁特性

**—->**读共享、写互斥

影响读写锁性能需要考虑的几点

1. 释放优先：当一个操作「释放写锁」时，并且队列中同时存在读线程和写线程时，那么是读线程优先获得锁，还是写线程，或者说是最先发出请求的线程
2. 读线程插队：如果当读线程「持有读锁」时，有写线程在等待，那么新到达的读线程能否立即获得访问权，还是应该在写线程后面等待？
   * 如果允许读线程插队到线程前面，那么将提高并发性，但却可能造成写线程发生饥饿问题。
3. 重入性：读锁、写锁是否允许重入。

* 广义上的可重入锁指的是可重复可递归调用的锁，在外层使用锁之后，在内层仍然可以使用，并且不发生死锁（前提得是同一个对象或者class），这样的锁就叫做可重入锁。

1. 降级：如果一个线程「持有写锁」，那么它能否在不释放锁的情况下「降级成读锁」？
2. 升级：「持有读锁」的线程能否优于其他正在等待的读线程和写线程而「升级成写锁」？  
   * 在大多数的「读**-**写锁」实现中，并不支持升级，因为很容易造成死锁（如果两个读线程同时升级为写锁，那么二者都不会释放读取锁）

**ZooKeeper** 中，读写锁的实现

1. 创建 /zookeeperLock/sharedLock/{ip}-{type}-{id} 的「临时顺序节点」，来代表读写锁
   * + type枚举值 ： R：读锁（共享）W：写锁（互斥）
     + id：临时顺序节点自增序号
     + 获取「读锁」，会在ZooKeeper 上，创建类似节点：/sharedLock/10.0.10.1-R-0000000001。
     + 获取「写锁」，会在ZooKeeper 上，创建类似节点：/sharedLock/10.0.10.1-W-0000000002。
2. 客户端调用 getChildren 接口来获取所有已创建的子节点**[**前驱节点**]**列表。
3. 判断是否获得锁，对于读请求如果所有前驱节点都是读请求或者没有前驱节点，表明已经成功获取共享锁，同时开始执行读逻辑。对于写请求，如果自己不是序号最小的子节点，那么就进入等待。
4. 如果没有获取到共享锁，读请求向比自己序号小的最后一个写请求节点注册 watcher 监听，写请求向比自己序号小的最后一个节点注册watcher 监听。避免「惊群效应」

实际开发过程中，可以 curator 工具包封装的API帮助我们实现分布式锁。

<dependency>

<groupId>org.apache.curator</groupId>

<artifactId>curator-recipes</artifactId>

<version>x.x.x</version>

</dependency>

实践建议

「读**-**写锁」允许对共享数据，进行更高的并发访问。

在实践中，「读**-**写锁」只有在多处理器上，高频读 + 低频写 场景下，才能提高性能。

而在其他情况下，「读**-**写锁」的性能却比「独占锁」的性能要差一点，这是因为「读**-**写锁」的复杂性更高。

所以，要对程序进行分析，判断「读-写锁」是否能提高性能，特别是，大多数场景下，高频读的场景，可依赖读取缓存，提升并发能力。

参考

<https://mp.weixin.qq.com/s/hPi2pp4WKIAYGI7SQ0fl0g>

<https://juejin.cn/post/6844903601546985485>

<https://blog.csdn.net/rickiyeat/article/details/78314451>

<https://www.runoob.com/w3cnote/zookeeper-locks.html>