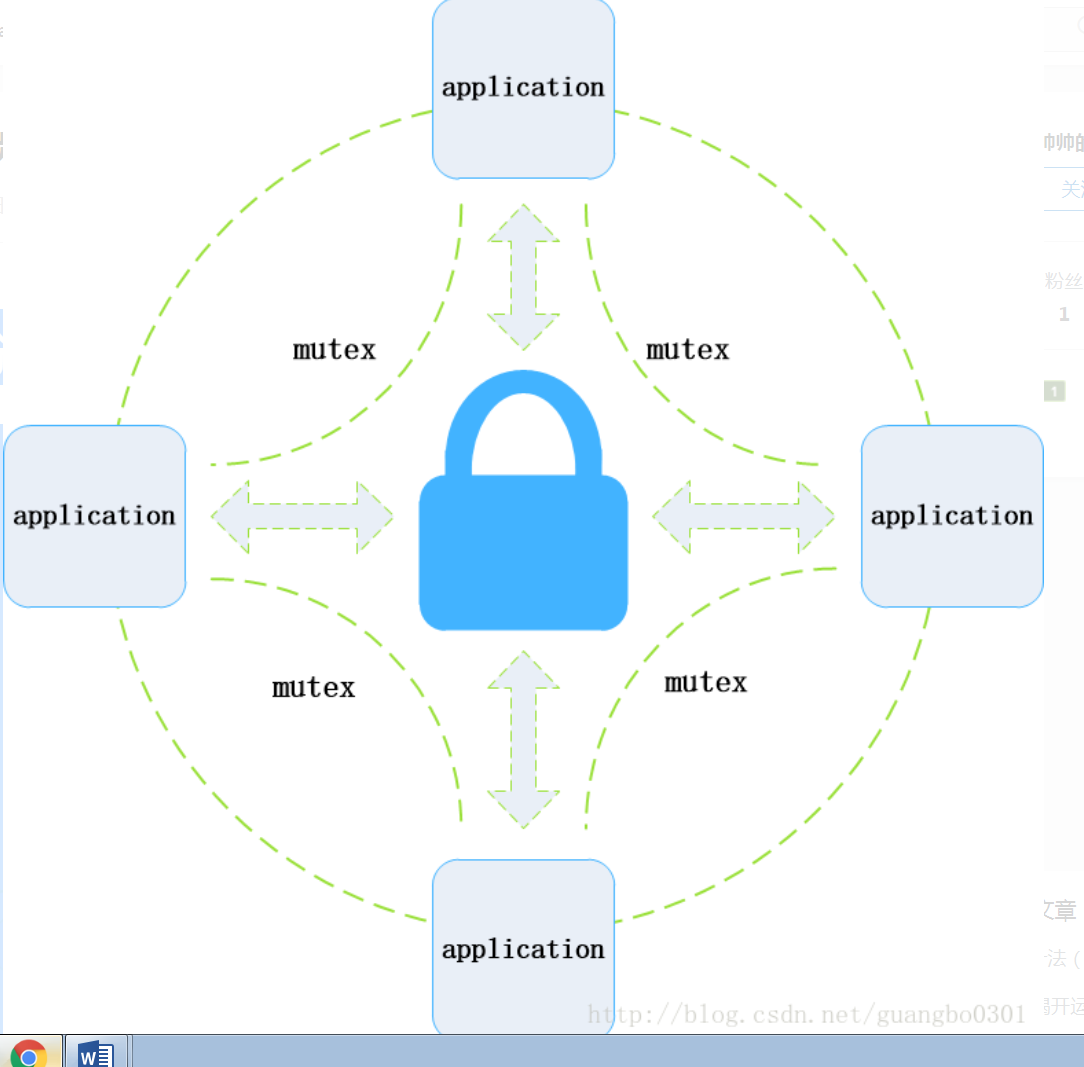
redis的分布式锁设计思路



**1.概念**  
我们在开发时最常用的一个是java给我们提供的基于jvm的锁，锁的获取和释放由jvm来管理，我们只需要标注synchronized就可以。   
 另外还有Lock，需要显式的调用锁定和解锁。这两种锁的作用范围是一个jvm进程，也就是我们的一个系统中；   
 在分布式系统中，一个集群内的不同主机或者不同集群同时访问共享的资源，会出现竞态条件(两个或多个线程竞争同一资源时，如果对资源  的访问顺序敏感就称为竞态条件)，使用传统的锁就没有办法 处理，此时就需要使用分布式锁来解决。   
 分布式锁主要就是解决分布式系统中共享资源的竞态条件问题。

**2.使用场景**  
多个应用有操作共享资源的情景   
**3.需要具备的功能**  
   先看下我们所熟悉的锁都具备什么功能   
   1. 作为锁所要具备的最基本的功能其一是获取锁，其二是释放锁；   
       synchronized关键字在jvm中使用了字节码指令monitorenter和monitorexitlock来获取和释放对象锁，   
       这两个字节码指令隐式的调用了lock和unlock操作。锁的获取和释放无需我们关心，jvm进程挂掉资源回收；   
       java提供的Lock需要我们显示的调用锁定和解锁。底层使用cas算法，控制原子变量的状态，来标记锁的获取与释放，  
       同样jvm进程挂掉资源进行回收；   
       分布式锁同样也需要提供获取和释放锁的功能。   
   2. 处理死锁  
       当出现死锁情况时最好能在一定时间内打破死锁的状态，否则会一直占用锁，占用资源。  
   3. 重入性   
       一般锁都会具备的特性，可重复获取已经获取到的锁。  
   4. 锁的性能等  
       锁性能的好坏直接影响系统及程序的执行，如synchronized持续优化，锁粒度及锁升级策略都是为了获得更好的性能。

**4.实现方式**  
  **方式：incr、decr 原子操作**  
   **加锁：**在需要使用的地方执行该key的incr操作，如果返回值是1，则获取锁   
   **解锁：**在finally块中将key做decr操作   
   **设置过期时间：**如果进程挂掉，导致锁没有释放，自动过期删除   
   **优点：**操作简单易行   
   **缺点：**   
   1.设置过期时间正确   
      获取锁的客户端进程在执行过程中挂掉，没有走finally块减一，那其他进程只能等redis的ttl自动删除;   
      该过期时间设置的长短难以把控，如果我们的请求因为其他原因阻塞了没有处理完，但已经到了redis的过期时间，  
      其他进程可以获得锁进行   处理，结果。。。   
   2.设置过期时间不正确   
      设置key的自增和设置过期时间不是原子操作，假如前者设置成功了，而过期时间因为各种原因没有设置成功，  
      一旦该锁的计数出现错误，那么所有进程都无法获取到锁，结果。。。

**总结上面方式所存在的问题：**  
**1.操作非原子性**  
**2.网络中断、命令发送失败**  
**3.死锁**  
**4.互斥**

**实现分布式锁需要考虑的几点：**  
 **1.线程和锁关系**  
    拿最常用的互斥锁来说   
    synchronized内置锁是作用于对象，java中每个对象是唯一存在的，   
    每个对象的对象头中包含获取该对象锁的线程ID，那就保证了线程对该对象锁的唯一性。   
    ReentrantLock内含Sync对象，其继承自同步器对象，同步器对象继承自 AbstractOwnableSynchronizer，   
    该对象可以设置获取该锁的独占线程。   
    从上面两种锁可以看出，锁的标识要与线程保持唯一性的关系   
    不例外的分布式锁也该如此，需要明确当前锁被哪个线程占有，也就是要维护锁与线程的关系。

**2.如何防止死锁的产生**   
    对于synchronized产生的死锁，似乎我们无能为力，即死锁状态无法解除；   
    一个线程已经获得了对象锁，其他线程访问共享对象的时就必须无限期等待，不能中断那些获取锁的线 程。   
    因此我们编码时让线程按照相同的顺序获得一组锁进行预防。   
    而Lock提供了更加灵活的方法，如果当前锁可用则返回true，否则返回false，并且可以设置获取锁的超时 时间，超时退出，防止死锁。   
    对于分布式的锁第一要设置锁的超时时间，让锁能及时释放掉。   
    其次还要设置客户端请求锁的超时时间，以防止通信过程出现问题，客户端线程一直等待锁响应。

**3.保证互斥性和重入性**   
    哪一个线程可以获得锁(独占锁)？哪些线程可以获取锁(共享锁)？线程之间互斥锁与共享锁是如何保证的？   
    我们先来看下传统锁是如何来处理的   
    synchronized是互斥锁，只需维护互斥关系，对象唯一，锁唯一；   
    重入性也是维护计数器，累加；   
    Lock既有互斥锁也有共享锁   
    其互斥特性是利用int变量state来判断，当然state是用volatile修饰。   
    判断state是否为0，如果为0，设置当前线程为独占锁拥有者，并将state加1；   
    重入特性实现也比较简单，state一直累加即可。   
    redis如何保证互斥性   
    使用过期时间，如果当前应用的线程获取锁的过期时间是null，设置锁的过期时间，并返回null，   
    当然该操作是原子操作（如何保证原子性，请看下文）；   
    如果在该锁过期时间内，其他线程获取的过期时间不为空，也就没有获得锁。   
    重入性：使用hset，设定当前锁的value值，如果重复获取锁，则累加；释放锁递减。

**4.如何保证原子性**  
    上面几种实现方式中，存在的原子性的问题。那如何来保证操作的原子性？   
     貌似通过我们的java程序保证是无济于事的；   
     redis在2.6版本以后，可以使用lua语言编写脚本传到redis服务端执行，  
     将我们之前多次与服务端交互才能完成的功能放到一次脚本中处理，   
     那我们所担心的原子性问题迎刃而解，还可以减少网络传输。

**5.未获取锁的线程如何处理**   
    得到锁的线程如愿以偿的去执行临界区内代码了，那未得到锁的线程去哪里了？   
    有两种方式:   
    非阻塞：未获取到锁的线程一直循环看锁的持有者是否释放锁。   
    优点：处理相对简单 缺点：占用cpu资源，容易产生死锁   
    阻塞：未获得锁，将线程本身进行阻塞。   
    优点：对线程统一管理调度 缺点：逻辑相对复杂   
    Lock把未得到锁的线程封装成Node节点，放入其构造的虚拟双向队列中，该队列是FIFO队列,   
    并进行阻塞操作，等待东山再起，；将队列符合条件的线程调用park()方法挂起阻塞。   
    当获取锁的线程释放锁时，会调用unpark()唤醒队列中第一个阻塞节点，使程序在阻塞处继续执行，   
    让队列head节点的下一个节点持有的线程获得锁，并且将该节点设为head节点。过程见下图： 

   synchronized也是封装Node节点，构造虚拟队列，与Lock不同的是，该队列是LIFO队列，所有请求锁的线程都被放入队列中，将符合条     件的线程移到另外的队列，具体过程不再赘述；也是采用阻塞的方式处理线程。   
    那分布式redis锁该如何处理未获得到锁的线程呢？   
    采用非阻塞方式，没得到锁的线程，不断的轮询来获取锁，很明显这样的方式会增加额外的无用功，会增加redis服务端节点的压力。   
    如何进行优化？   
    一个应用里面可能会有多个线程竞争该锁，可以控制应用里线程对锁的申请频率   
    1.让线程sleep一段时间再请求锁，那sleep多长时间呢，时间不好把控。   
    2.单个应用进程里可以使用信号量来限制，那就需要对信号量进行增减操作，来控制一定数量的线程。  
    没有信号量的线程阻塞，直到某个线程释放锁后信号量加1，线程获得信号量后来竞争锁。  
    那就需要监听线 程释放锁的操作，如何监听呢？可以使用redis的pub/sub方式，异步处理，增加吞吐量；   
    首先线程订阅某个锁的topic，如果获取不到锁，就发起sub操作，并阻塞当前线程，一旦有线程释放锁，pub消息给订阅的客户端，  
    客户端进行信号量的处理，使阻塞的线程获取许可来竞争锁。对于某个锁有哪些线程需要，也就转移到redis服务端记录。   
    具体过程如下图： 

     除此之外还需要考虑网络因素，命令发送失败重试(使用netty通讯)，单点因素等。

**一、RDB概述**

RDB是在某个时间点将数据写入一个临时文件，持久化结束后，用这个临时文件替换上次持久化的文件，达到数据恢复。   
优点：**使用单独子进程来进行持久化，主进程不会进行任何IO操作，保证了redis的高性能**   
缺点：**RDB是间隔一段时间进行持久化，如果持久化之间redis发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候**

这里说的这个执行数据写入到临时文件的时间点是可以通过配置来自己确定的，通过配置**redis在n秒内如果超过m个key被修改**这执行一次RDB操作。这个操作就类似于在这个时间点来保存一次Redis的所有数据，一次快照数据。所有这个持久化方法也通常叫做snapshots。

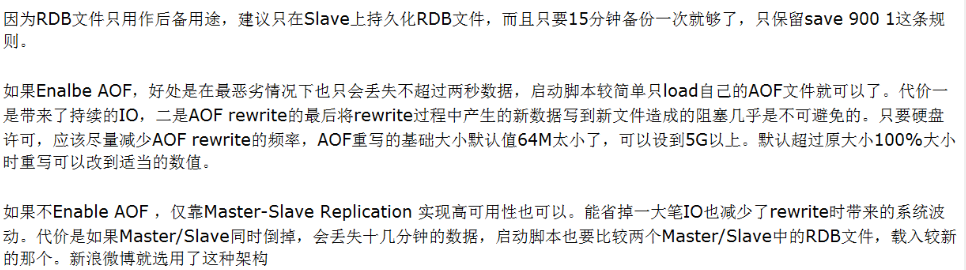
snapshot触发的时机，是有“间隔时间”和“变更次数”共同决定，同时符合2个条件才会触发snapshot,否则“变更次数”会被继续累加到下一个“间隔时间”上。snapshot过程中并不阻塞客户端请求。snapshot首先将数据写入临时文件，当成功结束后，将临时文件重名为dump.rdb。

**二、AOF概述**

Append-only file，将“操作 + 数据”以格式化指令的方式追加到操作日志文件的尾部，在append操作返回后(已经写入到文件或者即将写入)，才进行实际的数据变更，“日志文件”保存了历史所有的操作过程；当server需要数据恢复时，可以直接replay此日志文件，即可还原所有的操作过程。AOF相对可靠，它和mysql中bin.log、apache.log、zookeeper中txn-log简直异曲同工。AOF文件内容是字符串，非常容易阅读和解析。   
优点：**可以保持更高的数据完整性，如果设置追加file的时间是1s，如果redis发生故障，最多会丢失1s的数据；且如果日志写入不完整支持redis-check-aof来进行日志修复；AOF文件没被rewrite之前（文件过大时会对命令进行合并重写），可以删除其中的某些命令（比如误操作的flushall）。**   
缺点：**AOF文件比RDB文件大，且恢复速度慢。**

* 1) AOF更加安全，可以将数据更加及时的同步到文件中，但是AOF需要较多的磁盘IO开支，AOF文件尺寸较大，文件内容恢复数度相对较慢。   
  \*2) snapshot，安全性较差，它是“正常时期”数据备份以及master-slave数据同步的最佳手段，文件尺寸较小，恢复数度较快。
* [**暗里着迷**](http://www.cnblogs.com/wangfajun/)

## [Redis之RDB与AOF 笔记](http://www.cnblogs.com/wangfajun/p/5787077.html)

* AOF定义：以日志的形式记录每个操作，将Redis执行过的所有指令全部记录下来（读操作不记录），只许追加文件但不可以修改文件，Redis启动时会读取AOF配置文件重构数据
* 换句话说，就是Redis重启就会根据日志内容从头到尾执行一次来完成数据的恢复工作。
* Tip:
* 一.RDB与AOF同时开启  默认先加载AOF的配置文件
* 二.相同数据集，AOF文件要远大于RDB文件，恢复速度慢于RDB
* 三.AOF运行效率慢于RDB,但是同步策略效率好，不同步效率和RDB相同
* **1.RDB持久化(以快照的方式) 策略（默认）：**
* save 900 1       （15分钟变更一次）  
  　　save 300 10     （5分钟变更10次）  
  　　save 60 10000  （1分钟变更1万次）  
  **2.RDB默认配置文件名称：**
* dbfilename dump.rdb
* **3.表示是否开启AOF持久化：**
* appendonly yes(默认no,关闭)
* **4.AOF持久化配置文件的名称：**
* appendfilename "appendonly.aof"
* **5.AOF持久化策略(默认每秒)：**
* appendfsync always (同步持久化，每次发生数据变更会被立即记录到磁盘，性能差但数据完整性比较好)
* appendfsync everysec (异步操作，每秒记录，如果一秒钟内宕机，有数据丢失)
* appendfsync no  （不同步）
* **6.AOF配置文件损坏修复方法：**
* 进入redis安装路径 执行 redis-check-aof --fix AOF配置文件名称
* **7.AOF的Rewrite(重写) ：**
* 定义：AOF采用文件追加的方式持久化数据，所以文件会越来越大，为了避免这种情况发生，增加了重写机制
* 当AOF文件的大小超过了配置所设置的阙值时，Redis就会启动AOF文件压缩，只保留可以恢复数据的最小指令集，可以使用命令bgrewriteaof
* 原理：当AOF增长过大时，会fork出一条新的进程将文件重写(也是先写临时文件最后rename)，遍历新进程的内存数据，每条记录有一条set语句。
* 重写AOF文件并没有操作旧的AOF文件，而是将整个内存中的数据内容用命令的方式重写了一个新的aof文件（有点类似快照）
* 触发机制：Redis会记录上次重写时的AOF文件大小，默认配置时当AOF文件大小是上次rewrite后大小的一倍且文件大于64M时触发
* auto-aof-rewrite-percentage 100  （一倍）  
  　　　　   auto-aof-rewrite-min-size 64mb  
  **8.RDB与AOF的选择：**
* 做备份：当数据量大，且对恢复速度有要求，并且数据的一致性要求不高的话，可以只使用RDB
* 只做缓存：不用开启任何的持久化方式
* 两者都开启的建议：RDB数据不实时，同时使用两者时服务器只会找AOF文件，可不可以只使用AOF?作者建议不要，因为RDB更适合备份数据库(AOF在不断变化，不好备份)
* 快速重启，而且不会又AOF可能潜在的BUG,留作万一的手段。
* **9.优化：**
* ****
* 分类: [Redis](http://www.cnblogs.com/wangfajun/category/868301.html)