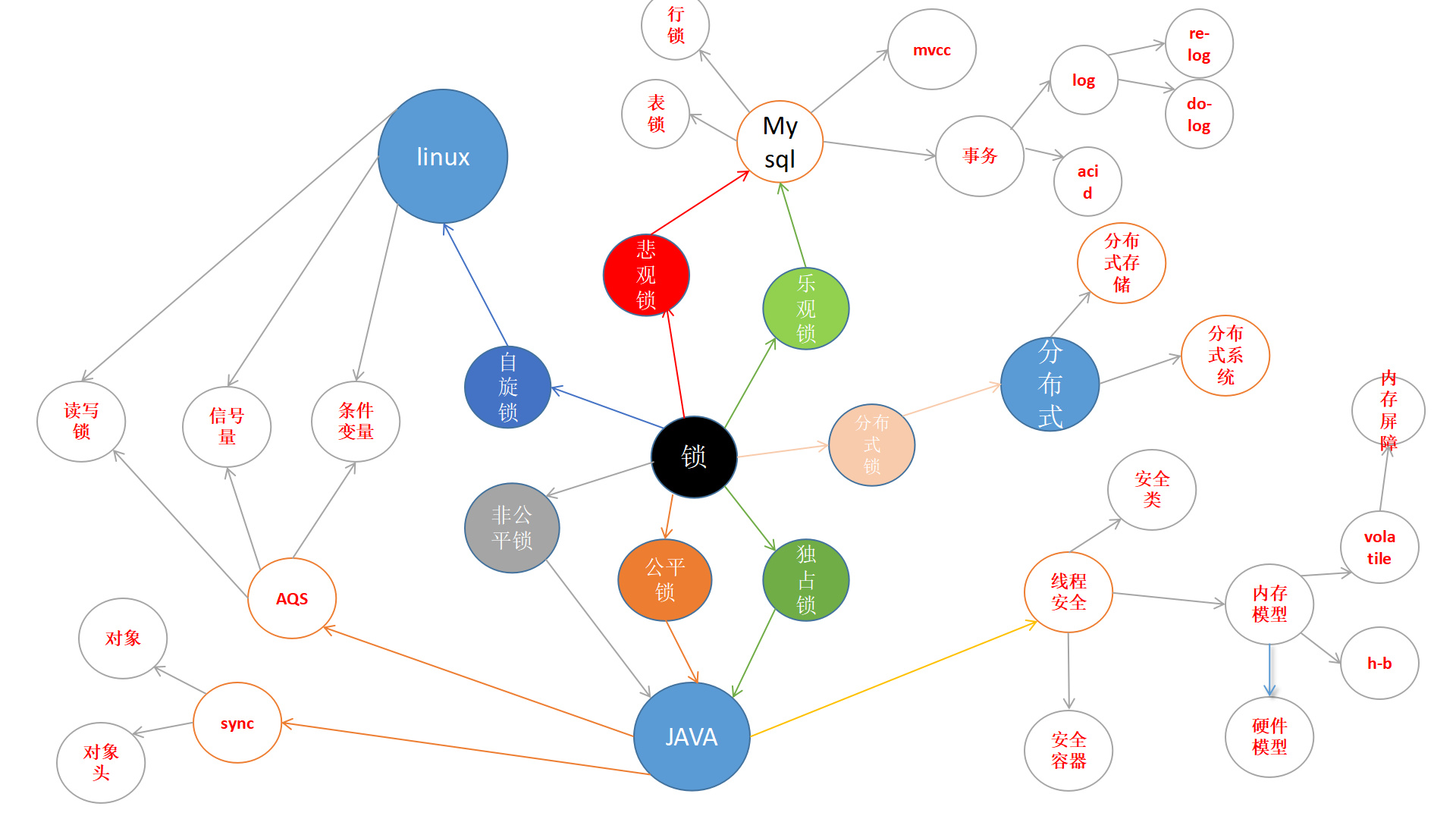
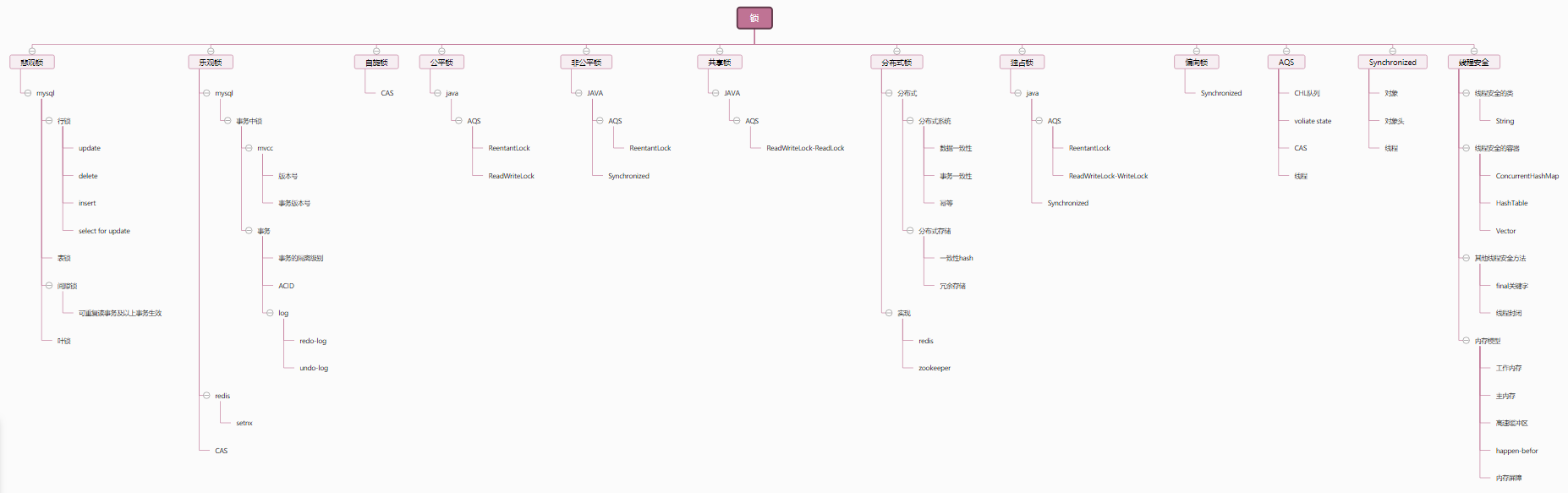
# 两程序员玩“锁”，一人抢救无效身亡

<https://zhuanlan.zhihu.com/p/34510121>



# 锁:



## 一、乐观锁

总是认为不会产生并发问题，每次去取数据的时候总认为不会有其他线程对数据进行修改，因此不会上锁，但是在更新时会判断其他线程在这之前有没有对数据进行修改，一般会使用版本号机制或CAS操作实现。

version方式：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

核心SQL代码：

update table set x=x+1, version=version+1 where id=#{id} and version=#{version};

CAS操作方式：即compare and swap 或者 compare and set，涉及到三个操作数，数据所在的内存值，预期值，新值。当需要更新时，判断当前内存值与之前取到的值是否相等，若相等，则用新值更新，若失败则重试，一般情况下是一个自旋操作，即不断的重试。

乐观锁的两种实现方式

什么场景下需要使用锁？

在多节点部署或者多线程执行时，同一个时间可能有多个线程更新相同数据，产生冲突，这就是并发问题。这样的情况下会出现以下问题：

更新丢失：一个事务更新数据后，被另一个更新数据的事务覆盖。

脏读：一个事务读取另一个事物为提交的数据，即为脏读。

其次还有幻读。。

针对并发引入并发控制机制，即加锁。

加锁的目的是在同一个时间只有一个事务在更新数据，通过锁独占数据的修改权。

锁的实现方式

          1、悲观锁，前提是，一定会有并发抢占资源，强行独占资源，在整个数据处理过程中，将数据处于锁定状态。

          2、乐观锁，前提是，不会发生并发抢占资源，只有在提交操作的时候检查是否违反数据完整性。只能防止脏读后数据的提交，不能解决脏读。

当然，还有其他的锁机制，暂时不多介绍，着重于乐观锁的实现。

乐观锁，使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。

           记录1，id,status1,status2,stauts3,version，表示有三个不同的状态，以及数据当前的版本

           操作1：update table set status1=1,status2=0,status3=0 where id=111;

           操作2：update table set status1=0,status2=1,status3=0 where id=111;

           操作3：update table set status1=0,status2=0,status3=1 where id=111;

没有任何控制的情况下，顺序执行3个操作，最后前两个操作会被直接覆盖。

加上version字段，每一次的操作都会更新version，提交时如果version不匹配，停止本次提交，可以尝试下一次的提交，以保证拿到的是操作1提交后的结果。

          这是一种经典的乐观锁实现。

          另外，java中的compare and swap即cas，解决多线程并行情况下使用锁造成性能损耗的一种机制。

          CAS操作包含三个操作数，内存位置（V）,预期原值（A）和新值（B）。如果内存位置的值与预期原值相匹配，那么处理器会西东将该位置值更新为新值。否则，处理器不做任何操作。

记录2: id，stauts，status 包含3种状态值 1，2，3

           操作，update status＝3 where id＝111 and status＝1；

           即 如果内存值为1，预期值为1，则修改新值。对于没有执行的操作则丢弃。

思考：这两种方式有什么区别？

## 一、悲观锁

 总是假设最坏的情况，每次取数据时都认为其他线程会修改，所以都会加锁（读锁、写锁、行锁等），当其他线程想要访问数据时，都需要阻塞挂起。可以依靠数据库实现，如行锁、读锁和写锁等，都是在操作之前加锁，在Java中，synchronized的思想也是悲观锁。

总结Mysql中悲观锁是怎么使用的

这俩天看了很多有关高并发电商网站的解决方案。本文主要考虑在实现在与数据服务器交互的部分。乐观锁主要有2种解决方法，version和时间戳，从别的资料中很容易理解。但涉及悲观锁的具体解决方案，一直理解的不够达到实现程度。以下分享一下自己的经验，很多知识材料来自其他博客大佬们的无私分享~

在一般查询sql中，是这样的写查询：

1.select \* from table where id=1;

此时并没有提交该事务1

这样做是无法起到上锁的目的，也就是说其他事务中可能有

2.update table set name='wind' where id=1;

事务2如上文修改了数据，并commit   
此时事务1执行下一句

3.select \* from table where id=1;

发现会发现name字段2次结果不一致，这就是数据库事务并发时可能发生的不可重复读情况

解决方法是

select \* from table where id=1 LOCK IN SHARE MODE;

这种写会为事务1该行加上共享锁。   
事务2此时想要执行update语句就会阻塞，因为他需要为id=1的那条记录加排它锁，而排它锁是与共享锁无法共存的。

\*注：

共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。   
排他锁（X)：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。\*

　　mysql也给用户提供了加锁的机会，只要在sql后加LOCK IN SHARE MODE 或FOR UPDATE

　　共享锁（S）：SELECT \* FROM table\_name WHERE ... LOCK IN SHARE MODE  
　　排他锁（X）：SELECT \* FROM table\_name WHERE ... FOR UPDATE

**Mysql的InnoDB引擎支持事务，其默认事务隔离级别是REPEATABLE\_READ**   
1.READ\_UNCOMMITED   
这是事务最低的隔离级别，它充许另外一个事务可以看到这个事务未提交的数据。   
2.READ\_COMMITED   
保证一个事务修改的数据提交后才能被另外一个事务读取，即另外一个事务不能读取该事务未提交的数据。   
3.REPEATABLE\_READ  
保证一个事务相同条件下前后两次获取的数据是一致的   
4.SERIALIZABLE   
事务被处理为顺序执行。

通过在一条记录上设共享锁，可以在该事务期间保证2次读取同一记录的结果一致，也保证其他事务不会更改相同记录,这一方法用于在并发事务中保证查询sql的数据一致性。

通常我们在电商并发网站中，会涉及对同一个商品number数的更新，   
事务1中这样写，代表了一个用户在服务器在一个数据库连接执行的任务。   
begin;   
update goods set number=number-1 where good\_id=#｛id｝ and number>0;   
insert into detail(order\_id,good\_id ,user\_name) values(null,1,”wind”);   
commit;

另一个用户在一个数据库连接里需要执行的事务2此时是   
begin;   
⑴update goods set number=number-1 where good\_id=#｛id｝ and number>0;   
insert into detail(order\_id,good\_id ,user\_name) values(null,1,”sea”);   
commit;

因为他们请求同一个商品也就是说good\_id一致，但因为是update语句所以都需要上为该记录上排他锁，此时如果事务1执行过程中，事务2虽然是并发执行，但事务2会阻塞在⑴这句，等事务1commit后，释放行锁，事务2才会成功加排它锁继续执行。这样实现了并发事务在对同一记录的访问中，保证安全，不会超买。

突然思路没了…这是我的一些理解，希望能对大家有所帮助。

## 间隙锁

前段时间系统老是出现insert死锁，很是纠结。经过排查发现是间隙锁！间隙锁是innodb中行锁的一种， 但是这种锁锁住的却不止一行数据，他锁住的是多行，是一个数据范围。间隙锁的主要作用是为了防止出现幻读，但是它会把锁定范围扩大，有时候也会给我们带来麻烦，我们就遇到了。 在数据库参数中， 控制间隙锁的参数是：innodb\_locks\_unsafe\_for\_binlog， 这个参数默认值是OFF， 也就是启用间隙锁， 他是一个bool值， 当值为true时表示disable间隙锁。那为了防止间隙锁是不是直接将innodb\_locaks\_unsafe\_for\_binlog设置为true就可以了呢？ 不一定！而且这个参数会影响到主从复制及灾难恢复， 这个方法还尚待商量。  
  
间隙锁的出现主要集中在同一个事务中先delete 后 insert的情况下， 当我们通过一个参数去删除一条记录的时候， 如果参数在数据库中存在， 那么这个时候产生的是普通行锁， 锁住这个记录， 然后删除， 然后释放锁。如果这条记录不存在，问题就来了， 数据库会扫描索引，发现这个记录不存在， 这个时候的delete语句获取到的就是一个间隙锁，然后数据库会向左扫描扫到第一个比给定参数小的值， 向右扫描扫描到第一个比给定参数大的值， 然后以此为界，构建一个区间， 锁住整个区间内的数据， 一个特别容易出现死锁的间隙锁诞生了。

## 自旋锁

自旋锁原理非常简单，如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样就**避免用户线程和内核的切换的消耗**。

但是线程自旋是需要消耗cup的，说白了就是让cup在做无用功，线程不能一直占用cup自旋做无用功，所以需要设定一个自旋等待的最大时间。

如果持有锁的线程执行的时间超过自旋等待的最大时间扔没有释放锁，就会导致其它争用锁的线程在最大等待时间内还是获取不到锁，这时争用线程会停止自旋进入阻塞状态。

自旋锁的优缺点

自旋锁尽可能的减少线程的阻塞，这对于锁的竞争不激烈，且占用锁时间非常短的代码块来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起操作的消耗！

但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用cpu做无用功，占着XX不XX，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要cup的线程又不能获取到cpu，造成cpu的浪费。

自旋锁时间阈值

自旋锁的目的是为了占着CPU的资源不释放，等到获取到锁立即进行处理。但是如何去选择自旋的执行时间呢？如果自旋执行时间太长，会有大量的线程处于自旋状态占用CPU资源，进而会影响整体系统的性能。因此自旋的周期选的额外重要！

JVM对于自旋周期的选择，jdk1.5这个限度是一定的写死的，在1.6引入了适应性自旋锁，适应性自旋锁意味着自旋的时间不在是固定的了，而是由前一次在同一个锁上的自旋时间以及锁的拥有者的状态来决定，基本认为一个线程上下文切换的时间是最佳的一个时间，同时JVM还针对当前CPU的负荷情况做了

较多的优化

* 如果平均负载小于CPUs则一直自旋
* 如果有超过(CPUs/2)个线程正在自旋，则后来线程直接阻塞
* 如果正在自旋的线程发现Owner发生了变化则延迟自旋时间（自旋计数）或进入阻塞
* 如果CPU处于节电模式则停止自旋
* 自旋时间的最坏情况是CPU的存储延迟（CPU A存储了一个数据，到CPU B得知这个数据直接的时间差）
* 自旋时会适当放弃线程优先级之间的差异

自旋锁的开启

JDK1.6中-XX:+UseSpinning开启；   
JDK1.7后，去掉此参数，由jvm控制；

## 偏向锁

Java偏向锁(Biased Locking)是Java6引入的一项多线程优化。   
偏向锁，顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，这种情况下，就会给线程加一个偏向锁。   
如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM会消除它身上的偏向锁，将锁恢复到标准的轻量级锁。

它通过消除资源无竞争情况下的同步原语，进一步提高了程序的运行性能。

偏向锁的实现

偏向锁获取过程：

* 访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01，确认为可偏向状态。
* 如果为可偏向状态，则测试线程ID是否指向当前线程，如果是，进入步骤5，否则进入步骤3。
* 如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行5；如果竞争失败，执行4。
* 如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。（撤销偏向锁的时候会导致stop the word）
* 执行同步代码。

注意：第四步中到达安全点safepoint会导致stop the word，时间很短。

偏向锁的释放：

偏向锁的撤销在上述第四步骤中有提到。偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动去释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

偏向锁的适用场景

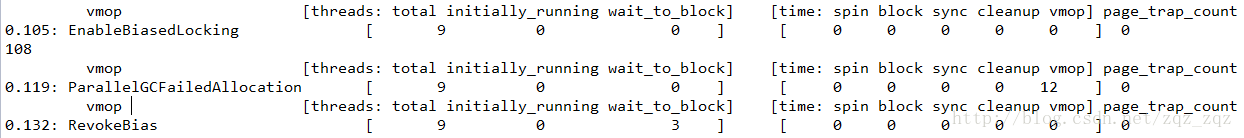
始终只有一个线程在执行同步块，在它没有执行完释放锁之前，没有其它线程去执行同步块，在锁无竞争的情况下使用，一旦有了竞争就升级为轻量级锁，升级为轻量级锁的时候需要撤销偏向锁，撤销偏向锁的时候会导致stop the word操作；   
在有锁的竞争时，偏向锁会多做很多额外操作，尤其是撤销偏向所的时候会导致进入安全点，安全点会导致stw，导致性能下降，这种情况下应当禁用；

查看停顿–安全点停顿日志

要查看安全点停顿，可以打开安全点日志，通过设置JVM参数 -XX:+PrintGCApplicationStoppedTime 会打出系统停止的时间，添加-XX:+PrintSafepointStatistics -XX:PrintSafepointStatisticsCount=1 这两个参数会打印出详细信息，可以查看到使用偏向锁导致的停顿，时间非常短暂，但是争用严重的情况下，停顿次数也会非常多；

注意：安全点日志不能一直打开：   
1. 安全点日志默认输出到stdout，一是stdout日志的整洁性，二是stdout所重定向的文件如果不在/dev/shm，可能被锁。   
2. 对于一些很短的停顿，比如取消偏向锁，打印的消耗比停顿本身还大。   
3. 安全点日志是在安全点内打印的，本身加大了安全点的停顿时间。

所以安全日志应该只在问题排查时打开。   
如果在生产系统上要打开，再再增加下面四个参数：   
-XX:+UnlockDiagnosticVMOptions -XX: -DisplayVMOutput -XX:+LogVMOutput -XX:LogFile=/dev/shm/vm.log   
打开Diagnostic（只是开放了更多的flag可选，不会主动激活某个flag），关掉输出VM日志到stdout，输出到独立文件,/dev/shm目录（内存文件系统）。



此日志分三部分：   
第一部分是时间戳，VM Operation的类型   
第二部分是线程概况，被中括号括起来   
total: 安全点里的总线程数   
initially\_running: 安全点时开始时正在运行状态的线程数   
wait\_to\_block: 在VM Operation开始前需要等待其暂停的线程数

第三部分是到达安全点时的各个阶段以及执行操作所花的时间，其中最重要的是vmop

* spin: 等待线程响应safepoint号召的时间；
* block: 暂停所有线程所用的时间；
* sync: 等于 spin+block，这是从开始到进入安全点所耗的时间，可用于判断进入安全点耗时；
* cleanup: 清理所用时间；
* vmop: 真正执行VM Operation的时间。

可见，那些很多但又很短的安全点，全都是RevokeBias， 高并发的应用会禁用掉偏向锁。

jvm开启/关闭偏向锁

开启偏向锁：-XX:+UseBiasedLocking -XX:BiasedLockingStartupDelay=0

关闭偏向锁：-XX:-UseBiasedLocking

## 重量级锁Synchronized

Synchronized的作用

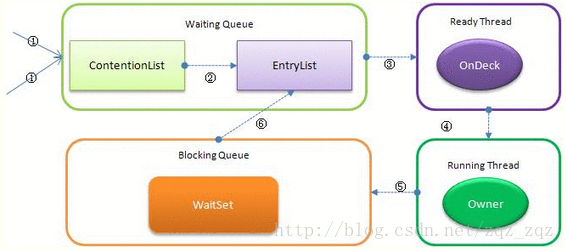
在JDK1.5之前都是使用synchronized关键字保证同步的，Synchronized的作用相信大家都已经非常熟悉了；

它可以把任意一个非NULL的对象当作锁。

* 作用于方法时，锁住的是对象的实例(this)；
* 当作用于静态方法时，锁住的是Class实例，又因为Class的相关数据存储在永久带PermGen（jdk1.8则是metaspace），永久带是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程；
* synchronized作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。

Synchronized的实现

实现如下图所示；



它有多个队列，当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。

* Contention List：竞争队列，所有请求锁的线程首先被放在这个竞争队列中；
* Entry List：Contention List中那些有资格成为候选资源的线程被移动到Entry List中；
* Wait Set：哪些调用wait方法被阻塞的线程被放置在这里；
* OnDeck：任意时刻，最多只有一个线程正在竞争锁资源，该线程被成为OnDeck；
* Owner：当前已经获取到所资源的线程被称为Owner；
* !Owner：当前释放锁的线程。

JVM每次从队列的尾部取出一个数据用于锁竞争候选者（OnDeck），但是并发情况下，ContentionList会被大量的并发线程进行CAS访问，为了降低对尾部元素的竞争，JVM会将一部分线程移动到EntryList中作为候选竞争线程。Owner线程会在unlock时，将ContentionList中的部分线程迁移到EntryList中，并指定EntryList中的某个线程为OnDeck线程（一般是最先进去的那个线程）。Owner线程并不直接把锁传递给OnDeck线程，而是把锁竞争的权利交给OnDeck，OnDeck需要重新竞争锁。这样虽然牺牲了一些公平性，但是能极大的提升系统的吞吐量，在JVM中，也把这种选择行为称之为“竞争切换”。

OnDeck线程获取到锁资源后会变为Owner线程，而没有得到锁资源的仍然停留在EntryList中。如果Owner线程被wait方法阻塞，则转移到WaitSet队列中，直到某个时刻通过notify或者notifyAll唤醒，会重新进去EntryList中。

处于ContentionList、EntryList、WaitSet中的线程都处于阻塞状态，该阻塞是由操作系统来完成的（[Linux](http://lib.csdn.net/base/linux)内核下采用pthread\_mutex\_lock内核函数实现的）。

**Synchronized是非公平锁。** Synchronized在线程进入ContentionList时，等待的线程会先尝试自旋获取锁，如果获取不到就进入ContentionList，这明显对于已经进入队列的线程是不公平的，还有一个不公平的事情就是自旋获取锁的线程还可能直接抢占OnDeck线程的锁资源。

# Java锁有哪些种类，以及区别

|  |  |  |
| --- | --- | --- |
| **一、公平锁/非公平锁** 公平锁是指多个线程按照申请锁的顺序来获取锁。 非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿现象。 对于Java ReentrantLock而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。 对于Synchronized而言，也是一种非公平锁。由于其并不像ReentrantLock是通过AQS的来实现线程调度，所以并没有任何办法使其变成公平锁。 **二、可重入锁** 可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。说的有点抽象，下面会有一个代码的示例。 对于Java ReentrantLock而言, 他的名字就可以看出是一个可重入锁，其名字是Re entrant Lock重新进入锁。 对于Synchronized而言,也是一个可重入锁。可重入锁的一个好处是可一定程度避免死锁。  [Java] 纯文本查看 复制代码  [?](https://www.52pojie.cn/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7 | synchronized void setA() throws Exception{  Thread.sleep(1000);  setB();  }  synchronized void setB() throws Exception{  Thread.sleep(1000);  } |   上面的代码就是一个可重入锁的一个特点，如果不是可重入锁的话，setB可能不会被当前线程执行，可能造成死锁。 **三、独享锁/共享锁** 独享锁是指该锁一次只能被一个线程所持有。 共享锁是指该锁可被多个线程所持有。 对于Java ReentrantLock而言，其是独享锁。但是对于Lock的另一个实现类ReadWriteLock，其读锁是共享锁，其写锁是独享锁。 读锁的共享锁可保证并发读是非常高效的，读写，写读 ，写写的过程是互斥的。 独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。 对于Synchronized而言，当然是独享锁。 **四、互斥锁/读写锁** 上面讲的独享锁/共享锁就是一种广义的说法，互斥锁/读写锁就是具体的实现。 互斥锁在Java中的具体实现就是ReentrantLock 读写锁在Java中的具体实现就是ReadWriteLock **五、乐观锁/悲观锁** 乐观锁与悲观锁不是指具体的什么类型的锁，而是指看待并发同步的角度。 悲观锁认为对于同一个数据的并发操作，一定是会发生修改的，哪怕没有修改，也会认为修改。因此对于同一个数据的并发操作，悲观锁采取加锁的形式。悲观的认为，不加锁的并发操作一定会出问题。 乐观锁则认为对于同一个数据的并发操作，是不会发生修改的。在更新数据的时候，会采用尝试更新，不断重新的方式更新数据。乐观的认为，不加锁的并发操作是没有事情的。 从上面的描述我们可以看出，悲观锁适合写操作非常多的场景，乐观锁适合读操作非常多的场景，不加锁会带来大量的性能提升。 悲观锁在Java中的使用，就是利用各种锁。 乐观锁在Java中的使用，是无锁编程，常常采用的是CAS算法，典型的例子就是原子类，通过CAS自旋实现原子操作的更新。 **六、分段锁** 分段锁其实是一种锁的设计，并不是具体的一种锁，对于ConcurrentHashMap而言，其并发的实现就是通过分段锁的形式来实现高效的并发操作。 我们以ConcurrentHashMap来说一下分段锁的含义以及设计思想，ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap(JDK7与JDK8中HashMap的实现)的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表;同时又是一个ReentrantLock(Segment继承了ReentrantLock)。 当需要put元素的时候，并不是对整个hashmap进行加锁，而是先通过hashcode来知道他要放在那一个分段中，然后对这个分段进行加锁，所以当多线程put的时候，只要不是放在一个分段中，就实现了真正的并行的插入。 但是，在统计size的时候，可就是获取hashmap全局信息的时候，就需要获取所有的分段锁才能统计。 分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。 **七、偏向锁/轻量级锁/重量级锁** 这三种锁是指锁的状态，并且是针对Synchronized。在Java 5通过引入锁升级的机制来实现高效Synchronized。这三种锁的状态是通过对象监视器在对象头中的字段来表明的。 偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价。 轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。 重量级锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。重量级锁会让其他申请的线程进入阻塞，性能降低。 **八、自旋锁** 在Java中，自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。典型的自旋锁实现的例子，可以参考自旋锁的实现. |