# [关于java锁机制的优化](http://willpower.javaeye.com/blog/255944)

JVM 级别的锁机制的优化主要针对这两种情况:   
1> 大多数情况不会存在竞争的锁.   
2> 被频繁竞争的锁.   
  
以下采用事例的方式加以说明. 前三种方法是针对情况1>, 最后一种是针对情况2>.   
  
1. lock elision

Java代码 [复制代码](http://willpower.javaeye.com/blog/255944)

1. public String getStoogeNames() {
2. Vector v = new Vector();
3. v.add("Moe");
4. v.add("Larry");
5. v.add("Curly");
6. return v.toString();
7. }

public String getStoogeNames() {

Vector v = new Vector();

v.add("Moe");

v.add("Larry");

v.add("Curly");

return v.toString();

}

我们知道Vector的实现是线程安全的,也就是加了synchronized. 但是在上面的这段代码中, 显然Vector v 不会被其他的线程所共享. 这个时候Vector实现中的锁是可以完全去掉的. 这种情况下, java 在compile的时候就能根据变量的作用域来进行相应的优化.比如:在生成的bytecode中去除掉所有的lock,unlock逻辑.   
  
2. Biased Locking

Java代码 [复制代码](http://willpower.javaeye.com/blog/255944)

1. public void getStoogeNames(Vector v) {
2. String[] names = {"Moe", "Larry", "Curly"};
3. for (int i=0; i<names.length; i++)
4. {
5. v.add(names[i]);
6. }
7. }

public void getStoogeNames(Vector v) {

String[] names = {"Moe", "Larry", "Curly"};

for (int i=0; i<names.length; i++)

{

v.add(names[i]);

}

}

请注意代码中的循环. 由于Vector v 这里非局部变量, 所以compiler不会对这段代码进行第一种优化. 这时, 一般情况下, 在每次循环进入v.add()时会做lock处理, 在退出时做unlock处理. 但是, 如果假设Vector v 在大多数情况下都只会被单个线程所占用, 那么JVM可以做这样的处理: 当前占有该锁的线程只在有其他线程请求到来时才进行解锁操作, 从而降低当前线程再次获得锁的overhead.   
显然, 要做这种处理, JVM需要先动态的得到统计数据表明该锁将在大部分时间内被单个线程占用.   
  
  
3. Lock coarsening

Java代码 [复制代码](http://willpower.javaeye.com/blog/255944)

1. public void addStooges(Vector v) {
2. v.add("Moe");
3. v.add("Larry");
4. v.add("Curly");
5. }

public void addStooges(Vector v) {

v.add("Moe");

v.add("Larry");

v.add("Curly");

}

这个问题其实和例2差不多. 这里就不赘述. 这次我们采用另一种方法: 将所有的add操作   
并入一个lock-unlock块中, 也就是所谓的粗化锁粒度.   
其实也可以用这种方式来处理方法2种的例子. 但是他们的不同是, 这种方法可能会导致其他线程长时间的得不到锁, 因为这种方式可能会产生出非常长的一段critical section, 系统总的吞吐量从而降低.    
  
4. Adaptive lock   
这是这里面唯一的一种处理被多线程频繁访问的锁的. 在这种情况下一般有两种竞争锁的方式:   
1> 当某个线程未能得到锁时, 它将不断的询问(spin lock), 如下例所示:

Java代码 [复制代码](http://willpower.javaeye.com/blog/255944)

1. while (lockStillInUse)
2. ;

while (lockStillInUse)

;

这种方式的优点是: 当临界段的处理很快时会有很快得到响应. 缺点是:会大量占用cpu资源.   
2> 当某线程未能得到锁时, 将其挂起, 并等待重新调度(suspension lock)   
这种方式刚好和方式1>互补.   
  
那么既然如此, 为什么不同时利用这两种方法的优点呢?   
这就是adaptive lock要做的事. JVM 可以先尝试着用spin lock, 如果发现线程都能在这种方式下很快的获得锁,那么就继续使用它. 如果发现超过了特定的时间线程还是未能得到锁则切换到suspension lock. 像这样经过一段时间的探索, JVM就可确定使用哪种方式.   
  
好了, 这是我对JVM级别锁优化的认识. 欢迎大家和我讨论.   
  
参考文献:   
[1] Java SE 6 Performance White Paper   
http://java.sun.com/performance/reference/whitepapers/6\_performance.html#2.1.2   
[2] Java theory and practice: Synchronization optimizations in Mustang   
http://www.ibm.com/developerworks/java/library/j-jtp10185/index.html   
[3] Java Synchronization Optimizations   
http://blogs.sun.com/dagastine/entry/java\_synchronization\_optimizations\_in\_mustang   
[4] Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing   
http://java.sun.com/javase/technologies/hotspot/publications/biasedlocking\_oopsla2006\_wp.pdf