# 28 JVM 对锁进行了哪些优化?

本课时我们主要讲解 JVM 对锁进行了哪些优化呢?

相比于 JDK 1.5，在 JDK 1.6 中 HotSopt 虚拟机对 synchronized 内置锁的性能进行了很多优化，包括自适应的自旋、锁消除、锁粗化、偏向锁、轻量级锁等。有了这些优化措施后，synchronized 锁的性能得到了大幅提高，下面我们分别介绍这些具体的优化。

## 自适应的自旋锁

首先，我们来看一下自适应的自旋锁。先来复习一下自旋的概念和自旋的缺点。“自旋”就是不释放 CPU，一直循环尝试获取锁，如下面这段代码所

```
public final long getAndAddLong(Object var1, long var2, long var4) {
    long var6;
    do {
        var6 = this.getLongVolatile(var1, var2);
    } while(!this.compareAndSwapLong(var1, var2, var6, var6 + var4));
    return var6;
}
```

代码中使用一个 do-while 循环来一直尝试修改 long 的值。自旋的缺点在于如果自旋时间过长，那么性能开销是很大的，浪费了 CPU 资源。

在 JDK 1.6 中引入了自适应的自旋锁来解决长时间自旋的问题。自适应意味着自旋的时间不再固定，而是会根据最近自旋尝试的成功率、失败率，以及当前锁的拥有者的状态等多种因素来共同决定。自旋的持续时间是变化的，自旋锁变“聪明”了。比如，如果最近尝试自旋获取某一把锁成功了，那么下一次可能还会继续使用自旋，并且允许自旋更长的时间；但是如果最近自旋获取某一把锁失败了，那么可能会省略掉自旋的过程，以便减少无用的自旋，提高效率。

## 锁消除

第二个优化是锁消除。首先我们来看下面的代码：

```
public class Person {

    private String name;

    private int age;

    public Person(String personName, int personAge) {

        name = personName;

        age = personAge;

    }

    public Person(Person p) {

        this(p.getName(), p.getAge());

    }

    public String getName() {

        return name;

    }

    public int getAge() {

        return age;

    }

}

class Employee {

    private Person person;

    // makes a defensive copy to protect against modifications by caller

    public Person getPerson() {

        return new Person(person);

    }

    public void printEmployeeDetail(Employee emp) {

        Person person = emp.getPerson();
```

```
        // this caller does not modify the object, so defensive copy was unnecessar
        System.out.println("Employee's name: " + person.getName() + "; age: " + per
    }
}
```

在这段代码中，我们看到下方的 Employee 类中的 getPerson() 方法，这个方法中使用了类里面的 person 对象，并且新建一个和它属性完全相同的新的 person 对象，目的是防止方法调用者修改原来的 person 对象。但是在这个例子中，其实是没有任何必要新建对象的，因为我们的 printEmployeeDetail() 方法没有对这个对象做出任何的修改，仅仅是打印，既然如此，我们其实可以直接打印最开始的 person 对象，而无须新建一个新的。

如果编译器可以确定最开始的 person 对象不会被修改的话，它可能会优化并且消除这个新建 person 的过程。

根据这样的思想，接下来我们就来举一个锁消除的例子，经过逃逸分析之后，如果发现某些对象不可能被其他线程访问到，那么就可以把它们当成栈上数据，栈上数据由于只有本线程可以访问，自然是线程安全的，也就无需加锁，所以会把这样的锁给自动去除掉。

例如，我们的 StringBuffer 的 append 方法如下所示：

```
@Override
public synchronized StringBuffer append(Object obj) {
    toStringCache = null;
    super.append(String.valueOf(obj));
    return this;
}
```

从代码中可以看出，这个方法是被 synchronized 修饰的同步方法，因为它可能会被多个线程同时使用。

但是在大多数情况下，它只会在一个线程内被使用，如果编译器能确定这个 StringBuffer 对象只会在一个线程内被使用，就代表肯定是线程安全的，那么我们的编译器便会做出优化，把对应的 synchronized 给消除，省去加锁和解锁的操作，以便增加整体的效率。

## 锁粗化

接下来，我们来介绍一下锁粗化。如果我们释放了锁，紧接着什么都没做，又重新获取锁，例如下面这段代码所示：

```
public void lockCoarsening() {
    synchronized (this) {
        //do something
    }
    synchronized (this) {
        //do something
    }
    synchronized (this) {
        //do something
    }
}
```

那么其实这种释放和重新获取锁是完全没有必要的，如果我们把同步区域扩大，也就是只在最开始加一次锁，并且在最后直接解锁，那么就可以把中间这些无意义的解锁和加锁的过程消除，相当于是把几个 synchronized 块合并为一个较大的同步块。这样做的好处在于在线程执行这些代码时，就无须频繁申请与释放锁了，这样就减少了性能开销。

不过，我们这样做也有一个副作用，那就是我们会让同步区域变大。如果在循环中我们也这样做，如代码所示：

```
for (int i = 0; i < 1000; i++) {
    synchronized (this) {
        //do something
    }
}
```

也就是我们在第一次循环的开始，就开始扩大同步区域并持有锁，直到最后一次循环结束，才结束同步代码块释放锁的话，这就会导致其他线程长时间无法获得锁。所以，这里的锁粗化不适用于循环的场景，仅适用于非循环的场景。

锁粗化功能是默认打开的，用 -XX:-EliminateLocks 可以关闭该功能。

## 偏向锁/轻量级锁/重量级锁

下面我们来介绍一下偏向锁、轻量级锁和重量级锁。这个锁在我们之前介绍锁的种类的时候也介绍过。这三种锁是特指 synchronized 锁的状态的，通过在对象头中的 mark word 来表明锁的状态。

- 偏向锁

对于偏向锁而言，它的思想是如果自始至终，对于这把锁都不存在竞争，那么其实就没必要上锁，只要打个标记就行了。一个对象在被初始化后，如果还没有任何线程来获取它的锁时，它就是可偏向的，当有第一个线程来访问它尝试获取锁的时候，它就记录下来这个线程，如果后面尝试获取锁的线程正是这个偏向锁的拥有者，就可以直接获取锁，开销很小。

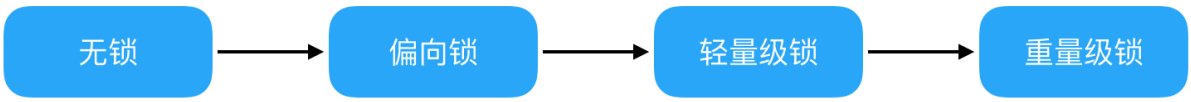
- 轻量级锁

JVM 的开发者发现在很多情况下，synchronized 中的代码块是被多个线程交替执行的，也就是说，并不存在实际的竞争，或者是只有短时间的锁竞争，用 CAS 就可以解决。这种情况下，重量级锁是没必要的。轻量级锁指当锁原来是偏向锁的时候，被另一个线程所访问，说明存在竞争，那么偏向锁就会升级为轻量级锁，线程会通过自旋的方式尝试获取锁，不会阻塞。

- 重量级锁这种锁利用操作系统的同步机制实现，所以开销比较大。当多个线程直接有实际竞争，并且锁竞争时间比较长的时候，此时偏向锁和轻量级锁都不能满足需求，锁就会膨胀为重量级锁。重量级锁会让其他申请却拿不到锁的线程进入阻塞状态。

## 锁升级的路径

最后，我们看下锁的升级路径，如图所示。从无锁到偏向锁，再到轻量级锁，最后到重量级锁。结合前面我们讲过的知识，偏向锁性能最好，避免了 CAS 操作。而轻量级锁利用自旋和 CAS 避免了重量级锁带来的线程阻塞和唤醒，性能中等。重量级锁则会把获取不到锁的线程阻塞，性能最差。




JVM 默认会优先使用偏向锁，如果有必要的话才逐步升级，这大幅提高了锁的性能。

上一页 下一页