Java 偏向锁、轻量级锁和重量级锁

前言

最开始听到偏向锁、轻量级锁和重量级锁的概念的时候，我还以为是 `Java` 中提供了相应的类库来实现的，结果了解后才发现， 这三个原来是虚拟机底层对 `synchronized` 代码块的不同加锁方式。

因此，不了解这三者的概念其实是不影响 `synchronized` 的使用的（大概），但是，了解它们对自身的提升来说却是必要的。

这里，就来看看它们是怎么回事吧！

同步代码块和同步方法

在 `Java` 中，关键字 `synchronized` 通常有两种使用方式，一是直接修饰在方法上定义同步方法，二是修饰单个对象，定义同步代码块：

```
public synchronized void syncMethod() {
  System.out.println("Sync method");
}

public void syncCodeBlock() {
  synchronized (this) {
    System.out.println("Sync code block");
  }
}
复制代码
```

对于同步代码块来说，Javac 编译时会在同步代码块的前后插入 `monitorenter` 和 `monitorexit` 指令，同时保证只要执行了 `monitorenter` 指令，就必然会执行 `monitorexit` 指令。

比如说上面的 `syncCodeBlock` 方法，它的编译结果为：

```
public void syncCodeBlock();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
       0: aload_0
       1: dup
       2: astore_1
  -->  3: monitorenter
       4: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #5                  // String Sync code block
       9: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
      12: aload_1
  --> 13: monitorexit
      14: goto          22
      17: astore_2
      18: aload_1
```

```
  --> 19: monitorexit
      20: aload_2
      21: athrow
      22: return
  Exception table:
   from    to  target type
  -->  4    14    17   any
      17    20    17   any
```
复制代码

可以看到，编译器在插入一个 `monitorenter` 后却插入了两个 `monitorexit` 指令，通过 `Exception table` 可以发现，当第 `4` 至 `14` 间的代码执行出现异常时，就会跳转到第 `17` 行执行， 此时，第 `17` 行后依然还有一个 `monitorexit` 指令保证同步代码块的退出。

但是对于同步方法来说，就不需要编译器添加 `monitorenter` 和 `monitorexit` 指令了，而是直接添加 `ACC_SYNCHRONIZED` 方法访问标志，方法的同步交由虚拟机完成：

```
public synchronized void syncMethod();
   descriptor: ()V
-> flags: ACC_PUBLIC, ACC_SYNCHRONIZED
   Code:
     stack=2, locals=1, args_size=1
        0: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
        3: ldc           #3                  // String Sync method
        5: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
        8: return
```
复制代码

虽然说同步方法和同步代码块编译出来的结果不一样，但是，它们最后实现同步的方式还是一样的。

锁对象和 Mark Word

对象头里面的 `Mark Word` 是了解 `synchronized` 实现原理时绕不开的东西，为了节约内存，这个 `Mark Word` 在不同锁状态下存储的内容是不一样的，大致如下图：

锁状态	25Bit		4Bit	1Bit	2Bit
	23Bit	2Bit		是否偏向锁	锁标志位
无锁	对象hashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向重量级锁的指针				10

其中，较为关键的便是最后的两位锁标志位了，根据其值的不同，虚拟机加锁时会做出不同的操作。

而锁对象，则是在获取锁和释放锁时需要关注的对象，对于同步代码块来说就是被 `synchronized` 关键字修饰的对象，对于同步方法来说，静态方法的锁对象是该类对应的 `java.lang.Class` 对象， 而普通方法则是相应的实例对象。

重量级锁

重量级锁指的就是一般意义上 `synchronized` 的同步方式，通过对象内部的监视器（monitor）实现，其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock 实现，操作系统实现线程之间的切换需要从用户态到内核态的切换， 切换成本非常高。

获取重量级锁后，会在对象头中保存指向重量级锁对象的指针，并将锁标志位的值设为 10，当其他线程过来尝试获得锁时，就会进入等待，直到重量级锁释放。

由于将线程挂起同样需要系统调用，存在用户态和内核态之间的转换，为了减少这种操作，对于获取重量级锁失败的线程来说，还可以通过 **自旋锁** 来等待获取锁成功的线程执行完成释放锁。

而自旋锁就是一个忙循环，因为很多同步块的执行时间并不是很长，因此通过一个忙循环等待来替代线程挂起是值得尝试的操作。

轻量级锁

获取释放重量级锁的消耗都是极为巨大的，如果临界区经常有几个线程同时访问，那么，这个消耗还可以接受，但是，如果临界区同一时间只有一个线程访问呢？这个时候还用重量级锁不就亏了？

因此，为了针对这一情况进行优化，虚拟机实现了轻量级锁，通过虚拟机自身在 **用户态** 下的 `CAS` 操作来替换获取释放重量级锁时的用户态内核态切换，其获取流程为：

1. 判断当前对象是否处于无锁状态（偏向锁标志为 0，锁标志位为 01），若是，则在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的 Mark Word 的拷贝，否则执行步骤（3）
2. 通过 CAS 操作尝试将对象的 Mark Word 更新为指向 Lock Record 的指针，如果成功表示竞争到锁，将锁标志位变成 00，执行同步操作代码，如果失败则执行步骤（3）
3. 判断当前对象的 Mark Word 是否指向当前线程的栈帧，如果是则表示当前线程已经持有当前对象的锁，则直接执行同步代码块，否则只能说明该锁对象已经被其他线程抢占了，这时轻量级锁需要膨胀为重量级锁

在执行完同步代码后，轻量级锁会被主动释放，释放流程如下：

1. 取出在获取轻量级锁保存在 Lock Record 中的数据
2. 用 CAS 操作将取出的数据替换到当前对象的 Mark Word 中，如果成功，则说明释放锁成功，否则执行步骤 (3)
3. 如果 CAS 操作替换失败，说明有其他线程尝试获取该锁，这时需要将该锁升级为重量级锁，并释放

轻量级锁的关键思路就在于通过 CAS 操作代替消耗大的系统调用，但是在频繁存在多个线程同时进入临界区的情况时，轻量级锁反而会带来额外的消耗。因此， 轻量级锁更适合不存在多个线程同时竞争同一个资源的情况。

偏向锁

虽然说轻量级锁通过 CAS 代替了系统调用减小了同步消耗，但是，如果临界区通常只有一个线程会进入呢？这时，是可以通过偏向锁进一步减小同步消耗的。

偏向锁通过如下措施进一步的减少了轻量级锁的消耗：

1. 在对象头中记录获取偏向锁成功的线程 ID，当该线程再次获取偏向锁时，发现线程 ID 一样，就可以直接通过判断执行同步代码，减少获取锁时的消耗
2. 不主动释放偏向锁，仅在出现竞争时才是否偏向锁，减小释放锁的消耗

获取偏向锁的过程为：

1. 检测 Mark Word 是否为可偏向状态（锁标志位为 01）
2. 若为可偏向状态，则测试线程 ID 是否为当前线程 ID，如果是，则执行步骤 (5)，否则执行步骤 (3)
3. 如果线程 ID 不为当前线程 ID，则通过 CAS 操作竞争锁，竞争成功，则将 Mark Word 的线程 ID 替换为当前线程 ID，否则执行线程 (4)
4. 通过 CAS 竞争锁失败，证明当前存在多线程竞争情况，当到达全局安全点，获得偏向锁的线程被挂起，撤销偏向锁，升级为轻量级锁，升级完成后被阻塞在安全点的线程继续往下执行同步代码
5. 执行同步代码块

偏向锁不会主动释放，只有当其它线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，释放过程为：

1. 暂停拥有偏向锁的线程，判断锁对象是否还处于被锁定状态
2. 撤销偏向锁，恢复到无锁状态(01)或者轻量级锁(00)的状态

偏向锁在 JDK 1.6 之后默认启用，可以通过 `XX:-UseBiasedLocking=false` 参数关闭偏向锁。

使用场景

虽然说从重量级锁到偏向锁的过程中，获取和释放锁的消耗在逐渐减少，但是，各自适用的场景也越来越特殊：

- 重量级锁，适用于多个线程 **同时** 进入临界区的场景
- 轻量级锁，适用于多个线程 **交替** 进入临界区
- 偏向锁，适用于 **只有一个** 线程进入临界区临界区的情况

当然了，使用那个锁是由虚拟机在运行时决定的，我们需要了解的是它们各自的实现原理，为什么要那么做，带来了什么好处，又有什么坏处。

结语

总的来说，这几个锁的概念比我想象的要容易一些，但也还是存在一些细节上的东西不是很清楚，其中一个就是锁膨胀的过程和重量级锁的具体实现。

这些东西后面还需要慢慢学习啊 (` ・ω・ ´)

参考链接

- 深入探究 JVM | Safepoint 及 GC 的触发条件 | 「浮生若梦」 - sczyh30's blog
- java 对象在内存中的结构（HotSpot虚拟机） - duanxz - 博客园
- Java Synchronized 实现原理 | big data decode club
- JVM 源码分析之 synchronized 实现 - 简书