

漫画：什么是 CAS 机制？

原创 2018-01-02 永远爱大家的 程序员小灰

点击上方“程序员小灰”，选择“置顶公众号”
有趣有内涵的文章第一时间送达！

老板，我明天去欧洲旅游，
请一天事假。



去吧去吧，好好玩。



第二天

小灰是吧？请简单介绍一下
你自己。



好的！
blah blah blah



下面考考你的多线程基础，知道
CAS 机制吗？



知... 知道一点，好像跟
线程安全有关系。



OK，那么请你来说一说，CAS 和
Synchronized 的区别是什么？适合什么
样的场景？有什么样的优点和缺点？



.....



.....



嘿嘿，不知道.....



呵呵，没关系，
回家等通知去吧！



小灰，听说你去面试了？
结果怎么样？



哎.....



大黄，你给我讲讲 Java 中的
CAS 机制呗？



好吧，在理解 CAS 之前，让我们
先来看一段程序。



示例程序：启动两个线程，每个线程中让静态变量count循环累加100次。

```
public static int count = 0;

public static void main(String[] args){
    //开启两个线程
    for(int i=0;i<2;i++){
        new Thread(
            new Runnable() {
                public void run() {
                    try{
                        Thread.sleep(10);
                    }catch(InterruptedException e){
                        e.printStackTrace();
                    }
                    //每个线程当中让count值自增100次
                    for(int j=0;j<100;j++){
                        count++;
                    }
                }
            }
        ).start();
    }
    try{
        Thread.sleep(2000);
    }catch(InterruptedException e){
        e.printStackTrace();
    }
    System.out.print("count="+count);
}
```

最终输出的count结果是什么呢？一定会是200吗？

这个我知道，因为这段代码不是线程安全，所以最终的自增结果很可能会小于 200 ！



说得很对，所以我们可以加上 Synchronized 同步锁，再看一看效果。




```
public static int count = 0;

public static void main(String[] args){
    //开启两个线程
    for(int i=0;i<2;i++){
        new Thread(
            new Runnable() {
                public void run() {
                    try{
                        Thread.sleep(10);
                    }catch(InterruptedException e){
                        e.printStackTrace();
                    }
                    //每个线程当中让count值自增100次
                    for(int j=0;j<100;j++){
                        synchronized (CountTest.class){
                            count++;
                        }
                    }
                }
            }
        ).start();
    }
    try{
        Thread.sleep(2000);
    }catch(InterruptedException e){
        e.printStackTrace();
    }
    System.out.print("~count="+count);
}
```

加了同步锁之后，count自增的操作变成了原子性操作，所以最终的输出一定是**count=200**，代码实现了线程安全。



Synchronized 的确保证了线程安全，但是在某些情况下，却不是一个最优选择。



为什么这么说呢？关键在于**性能**问题。

Synchronized关键字会让没有得到锁资源的线程进入**BLOCKED**状态，而后在争夺到锁资源后恢复为**RUNNABLE**状态，这个过程中涉及到操作系统**用户模式**和**内核模式**的转换，代价比较高。

尽管Java1.6为Synchronized做了优化，增加了从**偏向锁**到**轻量级锁**再到**重量级锁**的过度，但是在最终转变为重量级锁之后，性能仍然较低。

这样子啊... 那么有什么方式可以
替代同步锁呢？



你有没有听过，Java 当中的
「原子操作类」？



所谓原子操作类，指的是java.util.concurrent.atomic包下，一系列以Atomic开头的包装类。例如**AtomicBoolean**，**AtomicInteger**，**AtomicLong**。它们分别用于Boolean，Integer，Long类型的原子性操作。

现在我们尝试在代码中引入AtomicInteger类：

```
public static AtomicInteger count = new AtomicInteger(0);

public static void main(String[] args){
    //开启两个线程
    for(int i=0;i<2;i++){
        new Thread(
            new Runnable() {
                public void run() {
                    try{
                        Thread.sleep(10);
                    }catch(InterruptedException e){
                        e.printStackTrace();
                    }
                    //每个线程当中让count值自增100次
                    for(int j=0;j<100;j++){
                        count.incrementAndGet();
                    }
                }
            }
        ).start();
    }
    try{
        Thread.sleep(2000);
    }catch(InterruptedException e){
        e.printStackTrace();
    }
    System.out.print("count="+count.get());
}
```

使用AtomicInteger之后，最终的输出结果同样可以保证是200。并且在**某些情况下**，代码的性能会比Synchronized更好。

Wow！Atomic 操作类这么厉害，
底层究竟利用了什么手段呀？



Atomic 操作类的底层，正是利用了我们讲的「CAS 机制」！



什么是CAS？

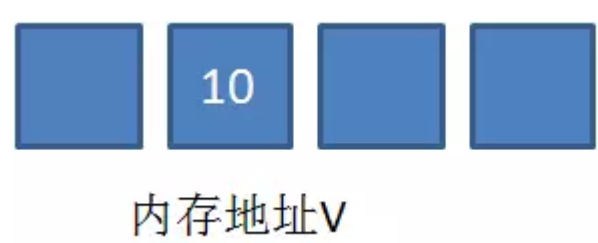
CAS是英文单词**Compare And Swap**的缩写，翻译过来就是比较并替换。

CAS机制当中使用了3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。

更新一个变量的时候，只有当变量的预期值A和内存地址V当中的实际值相同时，才会将内存地址V对应的值修改为B。

这样说或许有些抽象，我们来看一个例子：

1.在内存地址V当中，存储着值为10的变量。



2.此时线程1想要把变量的值增加1。对线程1来说，旧的预期值A=10，要修改的新值B=11。



内存地址V

线程1: A = 10 B = 11

3.在线程1要提交更新之前，另一个线程2抢先一步，把内存地址V中的变量值率先更新成了11。



内存地址V

线程1: A = 10 B = 11

线程2: 把变量值更新为11

4.线程1开始提交更新，首先进行A和地址V的实际值比较（Compare），发现A不等于V的实际值，提交失败。



内存地址V

线程1: A = 10 B = 11
A != V的值 (10!=11)
提交失败！

线程2: 把变量值更新为11

5.线程1重新获取内存地址V的当前值，并重新计算想要修改的新值。此时对线程1来说，A=11，B=12。这个重新尝试的过程被称为自旋。



内存地址V

线程1: A = 11 B = 12

6.这一次比较幸运，没有其他线程改变地址V的值。线程1进行**Compare**，发现A和地址V的实际值是相等的。



内存地址V

线程1: A = 11 B = 12
A == V的值 (11 == 11)

7.线程1进行**SWAP**，把地址V的值替换为B，也就是12。



内存地址V

线程1: A = 11 B = 12
A == V的值 (11 == 11)
地址V的值更新为12

从思想上来说，Synchronized属于**悲观锁**，悲观地认为程序中的并发情况严重，所以严防死守。CAS属于**乐观锁**，乐观地认为程序中的并发情况不那么严重，所以让线程不断去尝试更新。

原来 CAS 机制这么巧妙！那我以后再也不用 Synchronized 了，专门使用 CAS。



傻孩子，这两种机制没有绝对的好与坏，关键看使用场景。在并发量非常高的情况下，反而用同步锁更合适一些。



那么，在 Java 当中都有哪些地方应用到了 CAS 机制呢？



有许多地方都会用到，包括刚才说的 Atomic 系列类，以及 Lock 系列类的底层实现。



甚至在 Java1.6 以上版本，
Synchronized 转变为重量级锁
之前，也会采用 CAS 机制。



最后问一个问题：CAS 机制存
在什么样的缺点呢？



说起 CAS 机制的缺点，还真能
列出不少：



CAS的缺点：

1.CPU开销较大

在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给CPU带来很大的压力。

2.不能保证代码块的原子性

CAS机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证3个变量共同进行原子性的更新，就不得不使用Synchronized了。

3.ABA问题

这是CAS机制最大的问题所在。
什么是**ABA**问题？怎么解决？我们下一期来详细介绍。

几点补充：

本漫画纯属娱乐，还请大家尽量珍惜当下的工作，切勿模仿小灰的行为哦。

END

喜欢本文的朋友们，欢迎长按下图关注订阅号**程序员小灰**，收看更多精彩内容



投诉