**java synchronized的使用**

在多线程应用中，经常要考虑到线程同步问题，Java中内置了语言级的同步原语synchronized。

1. **synchronized的使用**

synchronized的使用主要有方法同步、块同步、对象同步和类同步。

1. 方法同步

class Account {  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​    String name;  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​    float amount; ​   
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​    public Account(String name, float amount) {  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​        this.name = name;  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​        this.amount = amount;  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​    }https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​    public  void deposit(float amt) {  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​        float tmp = amount;  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​        tmp += amt; https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​          
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​        try {  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​            Thread.sleep(100);//模拟其它处理所需要的时间，比如刷新数据库等  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​        } catch (InterruptedException e) {  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​            // ignore  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​        }  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​        amount = tmp;  
https://imgconvert.csdnimg.cn/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS9JbWFnZXMvT3V0bGluaW5nSW5kaWNhdG9ycy9Ob25lLmdpZg​    }

}

注：

1. 某个对象实例内，synchronized aMethod(){}可以防止多个线程同时访问这个对象的synchronized方法（如果一个对象有多个synchronized方法，只要一个线程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法）。这时，不同的对象实例的synchronized方法是不相干扰的。也就是说，其它线程照样可以同时访问相同类的另一个对象实例中的synchronized方法；
2. 是某个类的范围，synchronized static myStaticMethod{}防止多个线程同时访问这个类中的synchronized static 方法。它可以对类的所有对象实例起作用；
3. synchronized关键字是不能继承的，也就是说，基类的方法synchronized myMethod(){} 在继承类中并不自动是synchronizedmyMethod(){}，而是变成了myMethod(){}。继承类需要你显式的指定它的某个方法为synchronized方法。
4. 块同步：

通过 synchronized关键字来声明synchronized 块。语法如下：    
synchronized(syncObject) {    
 //允许访问控制的代码  
}    
synchronized 块是这样一个代码块，其中的代码必须获得对象 syncObject （如前所述，可以是类实例或类）的锁方能执行，具体机制同前所述。由于可以针对任意代码块，且可任意指定上锁的对象，故灵活性较高。

注：synchronized(this)的一些理解

1. 当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。
2. 当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同步代码块。
3. 当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将被阻塞。
4. 第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。
5. 以上规则对其它对象锁同样适用
6. 对象同步：

public void method3(SomeObject so){

    synchronized(so){   
       //…..   
    }

}

这时，锁就是so这个对象，谁拿到这个锁谁就可以运行它所控制的那段代码。

注：如果你的类中有一个对象同步方法，这个方法可以被两个不同的线程同时执行，只要每个线程自己创建一个的该类的实例对象即可。

类同步：

要实现真正的断面，你必须同步一个全局对象或者对类进行同步。

class Foo extends Thread{   
　private int val;   
　public Foo(int v){   
　　val = v;   
　}   
　public void printVal(int v){   
　　synchronized(Foo.class) {   
　　　while(true)  
　　　　System.out.println(v);   
　　}   
　}   
　public void run(){   
　　printVal(val);   
　}

}

注：上面的类不再对个别的类实例同步而是对类进行同步。对于类Foo而言，它只有唯一的类定义，两个线程在相同的锁上同步，因此只有一个线程可以执行printVal方法。

对共享资源的同步访问更加安全的技巧：

1． 定义private 的instance变量+它的 get方法，而不要定义public/protected的instance变量。如果将变量定义为public，对象在外界可以绕过同步方法的控制而直接取得它，并改动它。这也是JavaBean的标准实现方式之一。

2． 如果instance变量是一个对象，如数组或ArrayList什么的，那上述方法仍然不安全，因为当外界对象通过get方法拿到这个instance对象的引用后，又将其指向另一个对象，那么这个private变量也就变了，岂不是很危险。 这个时候就需要将get方法也加上synchronized同步，并且，只返回这个private对象的clone()――这样，调用端得到的就是对象副本的引用了

1. **锁优化**

线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒对CPU来说是一件负担很重的工作，势必会给系统的并发性能带来很大的压力。因此需要对锁进行优化。

1. 自旋锁

让该线程等待一段时间，不会被立即挂起，看持有锁的线程是否会很快释放锁。

虽然它避免线程切换带来的开销，但它占用了处理器的时间。如果持有锁的线程很快就释放了锁，那么自旋的效率就非常好，反之，自旋的线程就会白白消耗掉处理的资源，它不会做任何有意义的工作，反而会带来性能上的浪费。因此自旋等待的时间（自旋的次数）必须要有一个限度，如果自旋超过了定义的时间仍然没有获取到锁，则应该被挂起。

1. 自适应自旋锁

JDK 1.6引入了更加聪明的自旋锁，即自适应自旋锁。

线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能会再次成功，那么它就会允许自旋等待持续的次数更多。反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要或者这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源。

1. 锁消除

使用同步锁时，需要让同步块的作用范围尽可能小——仅在共享数据的实际作用域中才进行同步，这样做的目的是为了使需要同步的操作数量尽可能缩小，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。

1. 锁粗化

一系列的连续加锁解锁操作，可能会导致不必要的性能损耗。锁粗化就是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。

1. 偏向锁

首先检查这个对象头的MarkWord是不是储存着这个线程的ID。如果是，那么直接进去而不需要任何别的操作。如果不是，那么分为两种情况。1，对象的偏向锁标志位为0（当前不是偏向锁），说明发生了竞争，已经膨胀为轻量级锁，这时使用CAS操作尝试获得锁。2，偏向锁标志位为1，说明还是偏向锁不过请求的线程不是原来那个了。这时只需要使用CAS尝试把对象头偏向锁从原来那个线程指向目前求锁的线程。

如果CAS尝试失败，也就是说发生了竞争，有别的线程和它抢锁并且抢赢了，那么这个情况下，它就会要求撤销偏向锁（因为发生了竞争）。

1. 轻量锁

JVM会先在当前线程的栈帧中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中，官方成为Displaced Mark Word。然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。

cas如果成功，当前线程获得锁；如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋锁来获取锁；如果仍未获取到锁，则升级为重量级锁。

1. 重量级锁

重量级锁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。

1. **CAS**

CAS通过调用JNI的代码实现的。compareAndSwapInt就是借助C来调用CPU底层指令实现的。

CAS的缺点：

1. ABA问题。通过AtomicStampedReference或者在变量前面追加上版本号，每次变量更新的时候把版本号加一的方法解决；
2. 循环时间长开销大；
3. 只能保证一个共享变量的原子操作，要实现多个共享变量的原子操作，可以把多个共享变量合并成一个共享变量来操作。
4. **锁底层原理**
5. 处理器自动保证基本内存操作的原子性

处理器保证从系统内存当中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。

1. 使用总线锁保证原子性

总线锁就是使用处理器提供的一个LOCK＃信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住,那么该处理器可以独占使用共享内存。

1. 使用缓存锁保证原子性

总线锁定把CPU和内存之间通信锁住了，这使得锁定期间，其他处理器不能操作其他内存地址的数据，所以总线锁定的开销比较大。而缓存锁在同一时刻我们只需保证对某个内存地址的操作是原子性即可。

“缓存锁定”就是如果缓存在处理器缓存行中内存区域在LOCK操作期间被锁定，当它执行锁操作回写内存时，处理器不在总线上声言LOCK＃信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改被两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时会使缓存行无效

有两种情况下处理器不会使用缓存锁定：

1. 当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行（cache line），则处理器会调用总线锁定。
2. 有些处理器不支持缓存锁定。对于Inter486和奔腾处理器,就算锁定的内存区域在处理器的缓存行中也会调用总线锁定。

参考：<http://www.cnblogs.com/devinzhang/archive/2011/12/14/2287675.html>

<https://blog.csdn.net/qq_37933685/article/details/81088073>