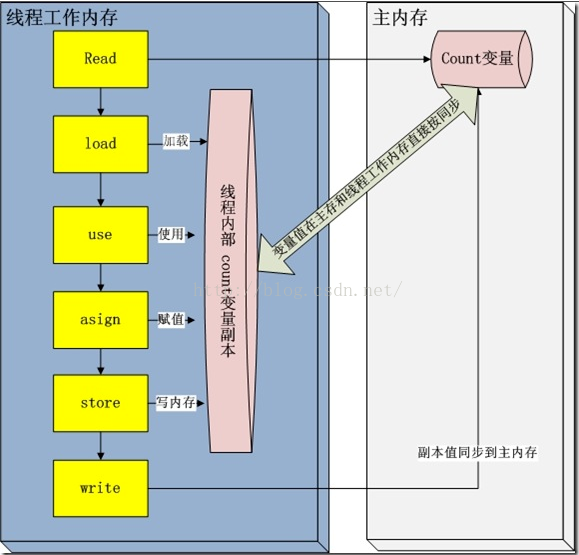
### 并发线程安全

#### 一 线程安全

#### 轻量级锁

这个关键字，保证了线程之间变量的可见性。原本变量在线程之间是独立的，也就是说，各自的线程用有的变量是各自的，所以总会出现一种情况，一个共享的资源，被线程A修改了，此时线程B来读取这个共享资源的时候，共享资源的值是修改之前的值。这是因为线程A没有把共享资源改变以后的值，进行直接回写。所以线程B看到共享资源还是原内存中的资源。Violate关键字的作用就是在保证线程BD读取共享资源之前，线程A对共享资源的修改，及时回写到主内存。这样对其他线程来说就是可见的了。



如图示线程的工作内存与主内存的区别。加了violate关键字以后,就能保证线程内存中的变量及时回写。但是这种情况一般适用于低并发，并且内存操作不涉及底层机器指令操作的情况下。对于如下所示的代码

**public** **class** ThreadVoalite {

**private** **static** **volatile** **long** *index*;

**static** **class** ThreadEG **implements** Runnable{

@Override

**public** **void** run() {

**for**(**int** i=0;i<1000;i++){

*index*=*index*+1;

}

}

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Thread[] ts=**new** Thread[10];

**for**(**int** i=0;i<ts.length;i++){

ts[i]=**new** Thread(**new** ThreadEG());

ts[i].start();

}

**for**(**int** i=0;i<ts.length;i++){

ts[i].join();

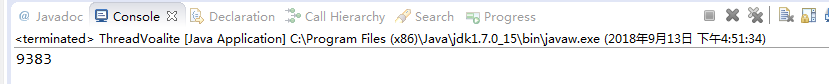
}

System.***out***.println(*index*);

}

}

从上面的代码中可以看出，也许你预期的结果是10000。然而结果却超出了你的想象范围。



这是为什么呢？这个时候或许就得引起我们的反思了。本质原因还是index=index+1并不是原子性的。即使我们换成index++也一样。Index++也是可以拆分成多条独立的指令来执行的。所以也不是原子性的。这个时候我们说violate()这个轻量级锁已经解决不了这样的并发问题了。如果这样的代码出现在100万行的代码量级中。是不是通宵加班都搞不定了。所以这些东西就是我们要注意的。

#### 重量级锁

Synchronized 关键字就是重量级锁了，那么我们应该怎么来用重量级锁呢。这个时候我给大家展示两种代码的写法。

##### 1.2.1 方式一 (错误)

**public** **class** ThreadVoalite {

**private** **static** **volatile** **long** *index*;

**static** **class** ThreadEG **implements** Runnable{

**public** **synchronized** **void** increase(){

*index*++;

}

@Override

**public** **void** run() {

**for**(**int** i=0;i<1000;i++){

increase();

}

}

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Thread[] ts=**new** Thread[10];

**for**(**int** i=0;i<ts.length;i++){

ts[i]=**new** Thread(**new** ThreadEG());

ts[i].start();

}

**for**(**int** i=0;i<ts.length;i++){

ts[i].join();

}

System.***out***.println(*index*);

}

}

##### 1.2.2 方式二 (错误)

**public** **class** ThreadVoalite {

**private** **static** **volatile** **long** *index*;

**static** **class** ThreadEG **implements** Runnable{

**private** Object lock=**new** Object();

@Override

**public** **void** run() {

**for**(**int** i=0;i<1000;i++){

**synchronized** (lock) {

*index*++;

}

}

}

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Thread[] ts=**new** Thread[10];

**for**(**int** i=0;i<ts.length;i++){

ts[i]=**new** Thread(**new** ThreadEG());

ts[i].start();

}

**for**(**int** i=0;i<ts.length;i++){

ts[i].join();

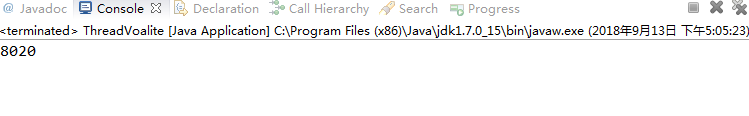
}

System.***out***.println(*index*);

}

}

大家可能认为结果应该是正确的，那么揭开神秘的面纱了



这两种运行方式都是错误的。那么很多同学就会怀疑了。难道重量级锁失效了吗？。那么大家可以看我上面的代码，然后想想为什么。对于线程来说，每一个线程它都是有自己的内存空间的。那么定义在线程内部的字段，就属于它的内存变量。每创建一个线程，那么变量就相当于又被创建了一次，那么又怎么能做为锁呢。这个时候让我来给你肯定的回答并没有。那么怎样写才能得到正确的结果呢。好的下面我给大家展示一下方式三。

##### 1.2.3 方式三 (正确)

**public** **class** ThreadVoalite {

**private** **static** **volatile** **long** *index*;

**static** **class** ThreadEG **implements** Runnable{

**private** Object lock=**null**;

**public** ThreadEG(Object lock) {

**this**.lock=lock;

}

@Override

**public** **void** run() {

**for**(**int** i=0;i<1000;i++){

**synchronized** (lock) {

*index*++;

}

}

}

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Object lock=**new** Object();

Thread[] ts=**new** Thread[10];

**for**(**int** i=0;i<ts.length;i++){

ts[i]=**new** Thread(**new** ThreadEG(lock));

ts[i].start();

}

**for**(**int** i=0;i<ts.length;i++){

ts[i].join();

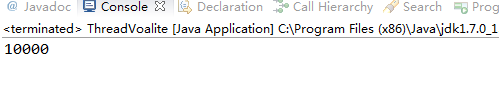
}

System.***out***.println(*index*);

}

}

这里如上代码段所示，这个时候你传入的Object lock是10个线程共有的资源。可以说是共享资源。这个时候它就可以用来做锁



##### 1.2.4 方式四 (正确)

**public** **class** ThreadVoalite {

**private** **static** **volatile** **long** *index*;

**static** **class** ThreadEG **implements** Runnable{

@Override

**public** **void** run() {

**for**(**int** i=0;i<1000;i++){

**synchronized** (ThreadEG.**class**) {

*index*++;

}

}

}

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Thread[] ts=**new** Thread[10];

**for**(**int** i=0;i<ts.length;i++){

ts[i]=**new** Thread(**new** ThreadEG());

ts[i].start();

}

**for**(**int** i=0;i<ts.length;i++){

ts[i].join();

}

System.***out***.println(*index*);

}

}

如上代码段所示也是正确的，那么为什么呢。关注1.3

#### 类锁，对象锁，变量锁

##### 1.3.1 类锁

就是对整个静态的class文件加锁，也就是说一个地方用到了这个class文件，其他地方，就要等待获取到class的锁。典型的比如我们的单例模式。就是控制类级别的锁。

**class** ThreadEF {

**public** **void** MethodA(){

**synchronized**(ThreadEF.**class**){ }

}

**public** **static** **synchronized** **void** MethodB(){}

}

如上代码所示，如果一个地方正在使用这个类，那么其他地方就不能够再使用这个类了。这就是类级别的锁。然而方法A,与B就是互斥的。也就是说这两种写法都是类锁的写法。

##### 1.3.2 对象锁

**class** ThreadEG {

**public** **void** MethodA(){

**synchronized**(**this**){ }

}

**public** **synchronized** **void** MethodB(){}

}

如上所示的对象加锁方法就是对象锁，一个类可以有多个实例，然而多个实例直接各自调用各自的方法是不会影响的，但是如上所示加了对象锁之后，那么这个实例是不可以对同时调用A与B方法的，这两个方法是同步的。但是如果是两个实例的话，这两个方法就是异步的。如果两个实例要实现同步，那么就得把锁上升到类锁。

##### 1.3.3 变量锁

**class** ThreadEH {

**private** Object object=**new** Object();

**public** **void** MethodA(){

**synchronized**(object){}

}

**public** **void** MethodB(){

**synchronized** (object){ }

}

}

如上代码所示，是通过变量来实现，A,B方法的同步的，然而这样的同步其实也是对象锁的特例。它无非还是让对象的A,B两个方法不能同时执行。那么上面的index++问题，大家应该知道怎么处理了吧。那就是类锁来解决。

#### 并发容器

##### 1.4.1并发下的ArrayList

**public** **class** ThreadArrayList {

**private** **static** List<Integer> *list*=**new** ArrayList<Integer>();

**static** **class** AddThread **implements** Runnable{

@Override

**public** **void** run() {

**for**(**int** i=0;i<10000;i++){

*list*.add(i);

}

}

}

**public** **static** **void** main(String[] args) {

Thread t1=**new** Thread(**new** AddThread());

Thread t2=**new** Thread(**new** AddThread());

t1.start();

t2.start();

t1.join();

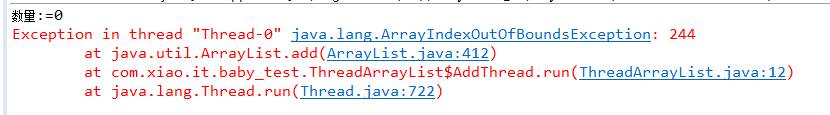
t2.join();

System.***out***.println("数量:="+*list*.size());

}

}

如上代码所示，可能你觉得这段代码会输出20000个数，然而结果是什么



异常了，数组越界。这样的原因就是因为内部ArrayList在扩容的过程中，出现了数据不一致的错误问题。当然及时如果不出现数据一致性的问题，实际的数值也将会是一个小于20000的数值。所以我们为什么说ArrayList是线程不安全的。那么要怎么处理呢，这个请继续关注我的博客。

##### 1.4.2并发下的HashMap

**public** **class** ThreadHashMap {

**private** **static** Map<Integer,Integer> *map*=**new** HashMap<Integer,Integer>();

**static** **class** AddThread **implements** Runnable{

@Override

**public** **void** run() {

**for**(**int** i=0;i<10000;i++){

*map*.put(i, i);

}

}

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Thread t1=**new** Thread(**new** AddThread());

Thread t2=**new** Thread(**new** AddThread());

t1.start();

t2.start();

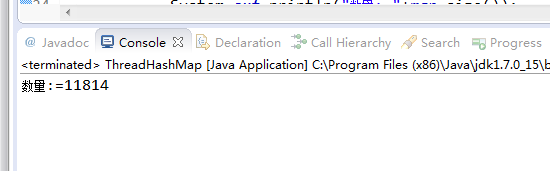
t1.join();

t2.join();

System.***out***.println("数量:="+*map*.size());

}

}



正如你所看见的，并发下的hashmap集合也未必是理想的是吧。都是一样的原因，因为他们在并发下都是线程不安全的。所示产生了与你预想不符合的现象是吧。如上种种问题，正式我们的新手不会深揪的问题。放在1000000的代码项目中。那就加班吧。那么怎么去注意和解决这样的问题。请继续关注我的博客。