java 并发基础!

作者：lee528066
原文链接：https://ld246.com/article/1502774133956
来源网站：链滴

当一个对象或变量可以被多个线程共享的时候，就有可能使得程序的逻辑出现问题。 在一个对象中有个变量i=0，有两个线程A，B都想对i加1，这个时候便有问题显现出来，关键就是对i加1的这个过程是原子操作。要想对i进行递增，第一步就是获取i的值，当A获取i的值为0，在A将新的值写入A之前，也获取了A的值0，然后A写入，i变成1，然后B也写入i，i这个时候依然是1.
当然java的内存模型没有上面这么简单，在Java Memory Model中，Memory分为两类，main memory和working memory，main memory为所有线程共享，working memory中存放的是线程所需的变量的拷贝（线程要对main memory中的内容进行操作的话，首先需要拷贝到自己的working memory，一般为了速度，working memory一般是在cpu的cache中的）。volatile的变量在被操作的时不会产生working memory的拷贝，而是直接操作main memory，当然volatile虽然解决了变量的可性问题，但没有解决变量操作的原子性的问题，这个还需要synchronized或者CAS相关操作配合进行。

多线程中几个重要的概念:

可见性

也就说假设一个对象中有一个变量i，那么i是保存在main memory中的，当某一个线程要操作i的时，首先需要从main memory中将i 加载到这个线程的working memory中，这个时候working memory中就有了一个i的拷贝，这个时候此线程对i的修改都在其working memory中，直到其将i从working memory写回到main memory中，新的i的值才能被其他线程所读取。从某个意义上说，可见性保证各个线程的working memory的数据的一致性。 可见性遵循下面一些规则：

- 当一个线程运行结束的时候，所有写的变量都会被flush回main memory中。
- 当一个线程第一次读取某个变量的时候，会从main memory中读取最新的。
- volatile的变量会被立刻写到main memory中的，在jsr133中，对volatile的语义进行增强，后面会到
- 当一个线程释放锁后，所有的变量的变化都会flush到main memory中，然后一个使用了这个相同同步锁的进程，将会重新加载所有的使用到的变量，这样就保证了可见性。

原子性

还拿上面的例子来说，原子性就是当某一个线程修改i的值的时候，从取出i到将新的i的值写给i之间不有其他线程对i进行任何操作。也就是说保证某个线程对i的操作是原子性的，这样就可以避免数据脏。 通过锁机制或者CAS（Compare And Set 需要硬件CPU的支持）操作可以保证操作的原子性。

有序性

假设在main memory中存在两个变量i和j，初始值都为0，在某个线程A的代码中依次对i和j进行自增作（i，j的操作不相互依赖）

```
i++;
j++;
```

由于，所以i,j修改操作的顺序可能会被重新排序。那么修改后的ij写到main memory中的时候，顺序能就不是按照i，j的顺序了，这就是所谓的reordering，在单线程的情况下，当线程A运行结束的后i，的值都加1了，在线程自己看来就好像是线程按照代码的顺序进行了运行（这些操作都是基于as-if-serial语义的），即使在实际运行过程中，i，j的自增可能被重新排序了，当然计算机也不能帮你乱排序，在上下逻辑关联的运行顺序肯定还是不会变的。但是在多线程环境下，问题就不一样了，比如另一个程B的代码如下

```
if(j == 1){
System.out.println(i);
```

```
}
```

按照我们的思维方式，当j为1的时候那么i肯定也是1，因为代码中i在j之前就自增了，但实际的情况有能当j为1的时候i还是为0。这就是reordering产生的不好的后果，所以我们在某些时候为了避免这样问题需要一些必要的策略，以保证多个线程一起工作的时候也存在一定的次序。JMM提供了happens before 的排序策略。这样我们可以得到多线程环境下的as-if-serial语义。 这里不对happens-before行详细解释了,详细的请看这里http://www.ibm.com/developerworks/cn/java/j-jtp03304/，这里要讲一下volatile在新的java内存模型下的变化，在jsr133之前，下面的代码可能会出现问题

```
Map configOptions;

char[] configText;

volatile boolean initialized = false;

// In Thread A

configOptions = new HashMap();

configText = readConfigFile(fileName);

processConfigOptions(configText, configOptions);

initialized = true;

// In Thread B

while (!initialized)

sleep();

// use configOptions
```

jsr133之前，虽然对 volatile 变量的读和写不能与对其他 volatile 变量的读和写一起重新排序，但是们仍然可以与对 nonvolatile 变量的读写一起重新排序，所以上面的Thread A的操作，就可能initializd变成true的时候，而configOptions还没有被初始化，所以initialized先于configOptions被线程B到，就产生问题了。

JSR 133 Expert Group 决定让 volatile 读写不能与其他内存操作一起重新排序，新的内存模型下，果当线程 A 写入 volatile 变量 V 而线程 B 读取 V 时，那么在写入 V 时，A 可见的所有变量值现在都以保证对 B 是可见的。

结果就是作用更大的 volatile 语义，代价是访问 volatile 字段时会对性能产生更大的影响。这一点在oncurrentHashMap中的统计某个segment元素个数的count变量中使用到了。