**学习多线程的前提：了解JVM之中的JMM(内存模型)**

在Java虚拟机规范中试图定义一种Java内存模型（Java Memory Model，JMM）来屏蔽各个硬件平台和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。那么Java内存模型规定了哪些东西呢，它定义了程序中变量的访问规则，往大一点说是定义了程序执行的次序。注意，为了获得较好的执行性能，Java内存模型并没有限制执行引擎使用处理器的寄存器或者高速缓存来提升指令执行速度，也没有限制编译器对指令进行重排序。也就是说，在java内存模型中，也会存在缓存一致性问题和指令重排序的问题。

Java内存模型规定所有的变量都是存在主存当中（类似于计算机组成原理说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）。线程对变量的所有操作都必须在工作内存中进行，而不能直接对主存进行操作。并且每个线程不能访问其他线程的工作内存。

1. **多线程的三个概念**

* 原子性
* 可见性(多线程操作同一个数据)
* 顺序性(指令重排序，导致代码语句执行步骤和我们直接写的并不一样)

1. **创建线程的三种方式**

* 先说类之间的关系，Thread类实现Runnable接口。Runnable接口中有run()方法，Thread类实现了Runnable接口，实现了run()方法。开启一个线程调用的是Thread中的start()方法，start()方法中调用了run()方法，也就是说run方法代码段里写的是线程运行时要执行的代码。start()方法会先根据threadStatus线程状态来判断线程是否已经被开启，若已经被开启，则报异常。然后调用start0方法(把当前线程装入线程池？)，把started置为true。
* 1.继承Thread类，并且重写run方法
* 2.实现Runnable接口，并且实现run方法，然后把实现Runnable接口的类作为参数放进一个新的new Thread(“参数”)实例中去，然后调用new Thread().start()方法
* 3.创建匿名内部类，直接Thread thread = new Thread(){

public void run(){

}

};

PS：

new Thread()的时候，Thread类有一个构造函数，是接收一个Runnable对象作为参数，因此我们可以new一个新的Runnable对象作为参数传入new Thread()里，然后在Runnable的匿名内部类里实现run方法

1. **线程同步**

* 当多个线程同时操作一个数据的时候，若数据保存和数据修改产生冲突，就会产生脏数据。如何避免呢？应该让数据的保存和修改，在一种有规则秩序的执行顺序进行下去才行。这就是线程的同步。
* 线程的同步使用关键词synchronized，使用方式如下

1. 声明一个Object，让两个线程的运行条件是必须占有这个Object线程才能运行，简单来说，就是一个线程运行的时候，这个对象被这个线程所占有，当第一个线程执行完毕时，释放该对象，然后第二个线程才能继续运行。

private final Object object = new Object();

Thread thread = new Thread(){

public void run(){

synchronized(object){

代码xxx

}

}

};

1. 可以把object替换为this，指代当前对象，其实任何对象都可以作为一个类似于“锁”的存在。
2. 在要调用的方法体内部声明synchronized
3. 在方法名前声明synchronized，其实效果等同于4，在方法内部声明synchronized(this)，只不过锁对象变为调用那个方法的那个类的实例了
4. 在静态方法前声明synchronized，所对象变为类名.class，类的字节码文件在内存中也是一个对象
5. 总的来说，想让两段代码同步，必须让同一个对象当锁
6. **线程安全的集合类**

HashMap VS HashTable

StringBuffer VS StringBuilder

ArrayList VS Vector

1. **线程之间的交互 wait和notify**

* wait和notify方法都是Object类里的方法
* wait方法的意思的让当前线程等待，并且释放当前线程占有对象上的锁
* notify方法的意思是唤醒等待的线程，并且让等待的线程重新占有对象锁
* 再次提醒，和之前的在方法名前添加synchronized一样，同步对象都是调用该方法的对象（注意，并不是类的字节码文件）
* 什么时候同步对象为类的字节码文件呢？当在静态方法前声明synchronized的时候

1. **ThreadLocal：线程内的数据共享**

* 声明一个static的整形变量，在多个线程之间修改这个整形变量的值，然后统一输出，发现两个线程里的static类型的变量值是不同的(因此说每一个线程单独开启一个内存？)
* ThreadLocal实现了在一个线程内存储数据，从而让这个线程之内的数据可以共享，而这个数据在线程间是独立的，互斥的
* 打开Thread的源码，我们可以看到存在一个名为ThreadLocalMap的变量，此变量其实维护了一张哈希表，以键值对的形式存储了多个Entry的，其中的key就是当前ThreadLocal，值就是我们要在线程中共享的值。底层是一个Entry数组，初始长度也是16，扩容为1.5。

但是有个问题，我们来看一下Entry数组是如何定义的：

**static class** Entry **extends** WeakReference<ThreadLocal<?>> {  
 */\*\* The value associated with this ThreadLocal. \*/* Object **value**;  
  
 Entry(ThreadLocal<?> k, Object v) {  
 **super**(k);  
 **value** = v;  
 }  
}

* 我们可以看出来，ThreadLocal的Entry中key是继承了WeakReference的，因此ThreadLocalMap中的Entry对象本质上是一个WeakReference<ThreadLocal>，也就是说ThreadLocalMap里保存的实际上是把ThreadLocal作为弱引用对象保存在key中，只不过Entry中还有value。

既然存在弱引用的问题了，那么这个ThreadLocalMap中的key随时有可能被回收，变为null，那么这个entry中的value值就不会被回收，从而造成内存泄露。事实真的如此吗？其实在set/get/remove方法中就已经判断了，遍历Entry数组的时候如果有当前key为空的情况，就把value也置为空。这样就不存在内存泄露的问题了(其实还是有的)。

其实网上也都这么说的，那我们再思考一个问题：

如果ThreadLocal是强引用而不是弱引用呢？

如果ThreadLocal是强引用，那么再对应的ThreadLocal被GC之后，ThreadLocalMap中的ThreadLocal也并不会被GC，也并没有被手动清除，久而久之就导致了内存泄露。

如果使用了弱引用，在ThreadLocal被GC之后，ThreadLocalMap对应的key就为空，然后在调用ThreadLocal的set/get等等方法的时候。就会让key值为空的Entry直接GC掉，就避免了内存泄露的发生。

因此，究竟有没有内存泄露，问题的关键在于ThreadLocalMap的生命周期和Thread的声明周期是一样的，而不是ThreadLocal是弱引用的。

PS：只要当前线程被GC了，那么这些ThreadLocalMap啊，ThreadLocal啊，Entry啊就都被GC了，也就不存在那些什么内存泄露的问题了。

* ThreadLocal的set方法

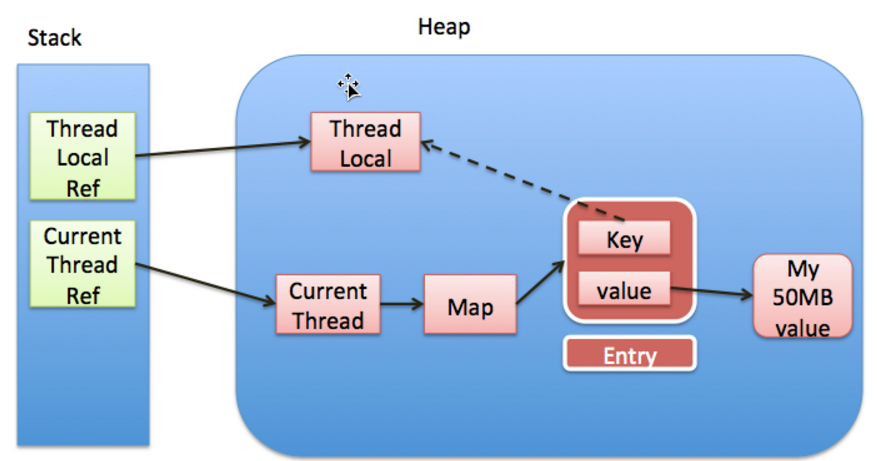
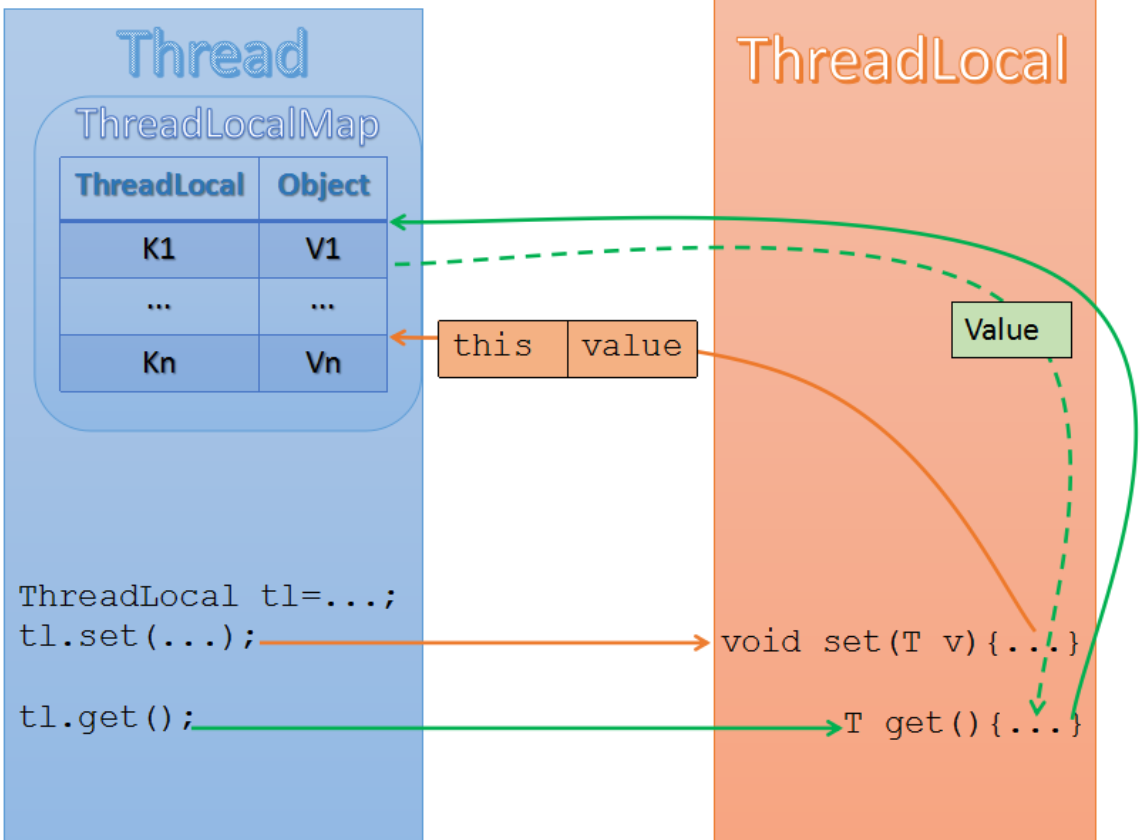
我们先理一理，ThreadLocal是Thread类中的一个属性，ThreadLocal的set(T)函数中，首先是拿到当前线程Thread对象中的ThreadLocalMap对象实例threadLocals ，然后再将需要保存的值保存到threadLocals里面。

换句话说，每个线程引用的ThreadLocal副本值都是保存在当前线程Thread对象里面的。存储结构为ThreadLocalMap类型，ThreadLocalMap保存的键类型为ThreadLocal，值为副本值。

* ThreadLocal的get方法

**public** T get() {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**) {  
 ThreadLocalMap.Entry e = map.getEntry(**this**);  
 **if** (e != **null**) {  
 @SuppressWarnings(**"unchecked"**)  
 T result = (T)e.**value**;  
 **return** result;  
 }  
 }  
 **return** setInitialValue();  
}

此方法是从当前线程中取出ThreadLocalMap的实例，然后通过threadlocal实例(在当前类中就直接使用this)作为key取出对应的value值

* 

Thread中有一张ThreadLocalMap表，表中为弱引用的Entry对象，Entry的key为当前的threadLocal

1. **多个线程共享数据**

* 上次我们说到使用ThreadLocal在线程内共享数据，接下来我们讨论如何在线程之间共享数据

1. 让一个类实现Runnable接口，然后多个线程把这个实现了Runnable接口的类当做参数，用来实例化一个线程。

众所周知，实例化一个Thread时，我们是可以把一个实现了Runnable接口的类当做参数传递进去的，这时候，启动线程之后调用的是实现了Runnable接口之后的run()方法。只要我们把操作的数据放在run()方法里，我们就可以实现在多个线程之间共享数据了。

1. 把数据操作的方法设置为同步的
2. **Volatile关键字的使用**

* 首先明白一点，volatile是干啥的？为什么要用？

我们根据一段代码，先来阐述一个现象：

**public class** A8\_19 {  
 **private static int** *a* = 0;  
 **public static void** main(String[] args){  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 *a*++;  
 System.***out***.println(Thread.*currentThread*().getName() + **"启动"**);  
 }  
 }).start();  
  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 *a*++;  
 System.***out***.println(Thread.*currentThread*().getName() + **"启动"**);  
 }  
 }).start();  
 **try** {  
 Thread.*sleep*(100);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(*a*);  
 }

**结果： Thread-1启动**

**Thread-0启动**

**1**

当然结果只可能是1或2，大部分情况下，a的值都是2，但是也存在上述结果，这是为什么呢？

(和**指令重排序**是否有关呢？？)

这里首先要理解一个**主存**和**缓存**的概念。从硬件的角度去理解，当我们执行一条语句比如说i = i + 1；的时候，先从物理主存中读取i的值，然后把i的值放入高速缓存中，然后根据CPU指令对i进行+1操作，然后将数据写入高速缓存，最后把高速缓存之中的数据放入主存之。

这里有一个问题，如果是多核多线程的情况下，第二条线程在第一条线程并未结束(还没有把操作完成的数据写入主存)的情况下，从主存之中拿到的i的值一定是0，这样等到第二条线程把数据写入主存以后，最后得到的结果仍然是1。

这个问题就被称为**缓存一致性**问题，在多线程中，若多条线程想要**共享操作**一个变量，就可能会出现缓存一致性问题。

那么我们该怎么解决呢？.

当然**硬件**层面上有人在解决这个问题了，但我们身为一个Java程序员，上边那种情况也是举个例子说说而已，考虑的更多的应该是Java本身是怎么做的。

（*顺便摘抄一句话：在Java虚拟机规范中试图定义一种Java内存模型（Java Memory Model，****JMM****）来屏蔽各个硬件平台和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。但是Java程序仍然存在缓存一致性和指令重排序的问题。*

）

如果把硬件层面会出现问题和Java对比来说，Java中把所有变量都定义在主存中，这个主存就相当于硬件中的物理内存，每个线程也有自己的工作内存，这个工作内存就相当于上边说的缓存。（所以说在Java中出现缓存一致性问题和指令重排问题是很正常的啦，也就解释了上边那段代码）

在Java中，我们管这方面叫做可见性。什么是可见性呢？就是指**多个线程同时操作同一个变量**的时候，一个线程操作数据之后，其他线程是不是能**立即**看到前一个线程**修改**的值。

下面来说一下解决方法和volatile关键字：

当我们用volatile修饰符来修饰一个变量的时候，不管在哪个线程之中，只要这个变量被修改，被**修改之后**的数据会**立即更新**到主存之中。

**把修改之后的数据立即更新到主存之中，为什么就能保证其他线程也能访问到被修改的数据？**

1. 使用volatile修饰符修饰的数据被修改后会**强制**更新在主存之中
2. 此数据在**其他线程**中的缓存行中的**缓存变量无效**
3. 由于其他线程读取此数据的时候，发现缓存无效，其他线程就会从主存中**重新读取**数据，也就能读取到主存中更新的数据了

可是仍然存在一个问题，我们并不能保证其他线程是什么时候从主存之中拿到数据的。有可能人家别的线程是你**更新数据之前（没写到主存之前）**就拿到数据的旧值了，这样仍然**不能保证可见性**。因为数据的读取，修改，并且更新到主存之中，是**三个**操作，不是**原子性**的。只有在最后一步把数据成功更新至主存中，才会让其他线程的**缓存行无效**。因此只有在一个线程**操作完一**个数据并放回主存之后，另一个线程才有权利从主存中获取这个值才行。这就需要利用**synchronized**关键字或者**lock**对象来解决了，我们就可以保证**只有一个方法**能操作数据，访问主存。

此外volatile还有一个作用，那就是保证顺序性，禁止指令重排。

以上我们可以发现：volatile并不能取代synchronized，虽然synchronized的效率并不高，但是synchronized可以保证同一段时间只有一个线程在执行一段代码，从而保证原子性，而volatile不可以保证原子性。

1. **线程池ThreadPool**