对于ThreadLocal、Volatile、synchronized、Atomic这四个关键字，我想一提及到大家肯定都想到的是解决在多线程并发环境下资源的共享问题，但是要细说每一个的特点、区别、应用场景、内部实现等，却可能模糊不清，说不出个所以然来，所以，本文就对这几个关键字做一些作用、特点、实现上的讲解。

# Atomic

对于原子操作类，[Java](http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "http://blog.csdn.net/u010687392/article/details/_blank)的concurrent并发包中主要为我们提供了这么几个常用的：**AtomicInteger、AtomicLong、AtomicBoolean、AtomicReference<T>**。

这些原子操作类最大的特点是在多线程并发操作同一个资源的情况下，采用Lock-Free算法替代锁+原子操作指令实现并发情况下资源的安全、完整、一致性，这样开销小、速度快，对于原子操作类是采用原子操作指令实现的，从而可以保证操作的原子性。什么是原子性？比如一个操作i++；实际上这是三个原子操作，先把i的值读取、然后修改(+1)、最后写入给i。所以使用Atomic原子类操作数，比如：i++；那么它会在这步操作都完成情况下才允许其它线程再对它进行操作，而这个实现则是通过Lock-Free+原子操作指令来确定的 

**Atomic的用法**

比如在多个线程操作一个count变量的情况下，则可以把count定义为AtomicInteger，如下：

**public class** Counter {  
 **private** AtomicInteger **count** = **new** AtomicInteger();  
  
 **public int** getCount() {  
 **return count**.get();  
 }  
  
 **public void** increment() {  
 **count**.incrementAndGet();  
 }  
}

在每个线程中通过increment()来对count进行计数增加的操作，或者其它一些操作。这样每个线程访问到的将是安全、完整的count。

# Volatile

Volatile可以看做是一个轻量级的synchronized，它可以在多线程并发的情况下保证变量的“可见性”，什么是可见性？就是在一个线程的工作内存中修改了该变量的值，该变量的值立即能回显到主内存中，从而保证所有的线程看到这个变量的值是一致的。所以在处理同步问题上它大显作用，而且它的开销比synchronized小、使用成本更低。

对一个共享变量使用Volatile关键字保证了线程间对该数据的可见性，即不会读到脏数据。

注意：

1. 可见性：对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入

2. 原子性：在对boolean和int的基本类型get和set的时候，java保证原子性，但是对没有volatile修饰的long和double类型不保证，原因是long和double是64位的，而且类似于volatile++这种复合操作不具有原子性。

3. volatile修饰的变量如果是对象或数组之类的，其含义是对象获数组的地址具有可见性，但是数组或对象内部的成员改变不具备可见性。

举个例子：在写单例模式中，除了用静态内部类外，还有一种写法也非常受欢迎，就是Volatile+DCL：

**public class** Singleton {  
 **private static volatile** Singleton *instance*;  
  
 **private** Singleton() {  
 }  
  
 **public static** Singleton getInstance() {  
 **if** (*instance* == **null**) {  
 **synchronized** (Singleton.**class**) {  
 **if** (*instance* == **null**) {  
 *instance* = **new** Singleton();  
 }  
 }  
 }  
 **return** *instance*;  
 }  
}

这样单例不管在哪个线程中创建的，所有线程都是共享这个单例的。

虽说这个Volatile关键字可以解决多线程环境下的同步问题，不过这也是相对的，因为volatile不具有操作的原子性，也就是它不适合在对该变量的写操作依赖于变量本身自己。举个最简单的栗子：在进行计数操作时count++，实际是count=count+1;，count最终的值依赖于它本身的值。所以使用volatile修饰的变量在进行这么一系列的操作的时候，就有并发的问题   
举个例子：

*//volatile变量自增运算测试***public class** VolatileTest {  
  
 **public static volatile int** *race* = 0;  
 **private static final int *THREADS\_COUNT*** = 20;  
  
 **public static void** main(String[] args) {  
 Thread[] threads = **new** Thread[***THREADS\_COUNT***];  
 **for** (**int** i = 0; i < ***THREADS\_COUNT***; i++) {  
 threads[i] = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **for** (**int** i = 0; i < 1000; i++) {  
 *increase*();  
 }  
 }  
 });  
 threads[i].start();  
 }  
  
 **try** {  
 *//主线程停止10秒，让20个累加线程都执行完成后打印在打印race* Thread.*sleep*(10000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(*race*);  
 }  
  
 **public static void** increase() {  
 *race*++;  
 }  
  
}

这段代码发起了20个线程，每个线程对race变量进行1000自增，如果这段代码能够正确并发的话，正确的结果应该是20000，但实际运行结果总是小于20000。为什么呢？问题就出现在自增运算“race++”，race++并不是一个原子操作，我们可以把它在细分为rece+1运算和赋值两部分运算操作。所以，volatile变量只能保证可见性，在不符合以下两条规则的运算场景，我们仍然要通过加锁（或使用synchronized或java.util.concurrent中的原子类）来保证原子性。

1. 写入变量不依赖此变量的值（例如：i++，实质上是i=i+1;我们对i进行赋值时不能使用到i它本身），或者只有一个线程修改此变量
2. 变量的状态不需要与其它变量共同参与不变约束

而要想race能在并发环境中保持数据的一致性，则可以在increase()中加synchronized同步锁修饰，如：

**public synchronized static void** increase() {  
 *race*++;  
}

**另外，volatile还可以禁止指令重排序优化：**volatile的变量附近的代码不会被指令重排序打乱。

volatile屏蔽指令重排序的语义在JDK1.5中才被完全修复，此前的JDK中即使将变量声明为volatile也仍然不能完全避免重排序所导致的问题（主要是volatile变量前后的代码仍然存在排序问题），这点也是JDK1.5之前的Java中无法安全地使用DCL（双检锁）来实现单例模式的原因。

在性能上，volatile与普通变量上在写操作可能会慢些，因为需要插入内存屏障来保证不乱序执行。不过比sync高效很多。

# synchronized

synchronized叫做同步锁，是Lock的一个简化版本，由于是简化版本，那么性能肯定是不如Lock的，不过它操作起来方便，只需要在一个方法或把需要同步的代码块包装在它内部，那么这段代码就是同步的了，所有线程对这块区域的代码访问必须先持有锁才能进入，否则则拦截在外面等待正在持有锁的线程处理完毕再获取锁进入，正因为它基于这种阻塞的策略，所以它的性能不太好，但是由于操作上的优势，只需要简单的声明一下即可，而且被它声明的代码块也是具有操作的原子性。

**public synchronized void** increment(){  
 count++;  
}  
  
**public void** increment(){  
 **synchronized** (Counte.**class**){  
 count++;  
 }  
}

通过在increment方法前加入synchronized关键字，使多个线程在执行increment方法时，以排队的方式进行处理。当一个线程调用increment前，先判断increment方法有没有被上锁，如果上锁，说明有其他线程正在调用increment方法，必须等其他线程对increment方法调用结束后才可以开始执行run方法。这样就实现了排队调用increment方法的目的。Synchronized可以在任意对象及方法上加锁，而加锁的这段代码称为“互斥区”或“临界区”。当一个线程想要执行同步方法里面的代码时，线程首先尝试去拿掉这把锁，如果能拿掉，那么这个线程可以执行synchronized里的代码，如果不能拿掉，就一直尝试着拿掉，直到能拿掉为止，而且是有多个线程同时去争抢这把锁。

# ThreadLocal

ThreadLocal不是一个线程，而是线程的一个本地化对象，当工作于多个线程中的对象使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程分配一个独立的变量副本。所以每一个线程都可以独立地改变自己的副本，而不会影响其他线程所对应的副本。而在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。概括来说，同步机制采用了“以时间换空间”的方式访问串行化，对象共享化；而ThreadLocal采用了“以空间换时间”的方式访问并行化，对象独享化。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

关于ThreadLocal，这个类的出现并不是用来解决在多线程并发环境下资源的共享问题的，它和其它三个关键字不一样，其它三个关键字都是从线程外来保证变量的一致性，这样使得多个线程访问的变量具有一致性，可以更好的体现出资源的共享。而ThreadLocal的设计，**并不是解决资源共享的问题**，而是用来提供线程内的局部变量，这样每个线程都自己管理自己的局部变量，别的线程操作的数据不会对我产生影响，互不影响，所以不存在解决资源共享这么一说，如果是解决资源共享，那么其它线程操作的结果必然我需要获取到，而ThreadLocal则是自己管理自己的，相当于封装在Thread内部了，供线程自己管理。

**ThreadLocal的实现原理：**

其实实现的思路很简单：在ThreadLocal类中有一个Map，用于存储每一个线程的变量副本，Map中元素的键为线程对象，而值对应线程的变量副本。

ThreadLocal类接口只有4个方法：

* void set(Object value)：设置当前线程局部变量的值
* public Object get()：获取当前线程局部变量的值
* public void remove()：删除当前线程局部变量的值，当线程结束后，对应的线程局部变量将自动被垃圾回收，所以显示调用该方法不是必须的操作，但他可以加快内存回收的速度
* protected Object initialValue()：初始当前线程局部变量的值，该方法是一个protected的方法，显然是为了让子类覆盖设计的，这个方法是一个延迟调用方法，在线程第一次调用get()或set()时才执行，并且仅执行一次。

一般使用ThreadLocal，官方建议我们定义为private static ，至于为什么要定义成静态的，这和内存泄露有关，后面再讲。 它有三个暴露的方法，set、get、remove。

**public class** ThreadLocalDemo {  
 **private static** ThreadLocal<String> *threadLocal* = **new** ThreadLocal<String>(){  
 @Override  
 **protected** String initialValue() {  
 **return "hello"**;  
 }  
 };  
 **static class** MyRunnable **implements** Runnable{  
 **private int num**;  
 **public** MyRunnable(**int** num){  
 **this**.**num** = num;  
 }  
 @Override  
 **public void** run() {  
 *threadLocal*.set(String.*valueOf*(**num**));  
 System.***out***.println(**"threadLocalValue:"** + *threadLocal*.get());  
 }  
 }  
  
 **public static void** main(String[] args){  
 **for**(**int** i=0;i<10;i++){  
 Thread t = **new** Thread(**new** MyRunnable(i));  
 t.start();  
 }  
 }  
}

**执行结果：**

threadLocalValue:0

threadLocalValue:4

threadLocalValue:3

threadLocalValue:2

threadLocalValue:5

threadLocalValue:1

threadLocalValue:6

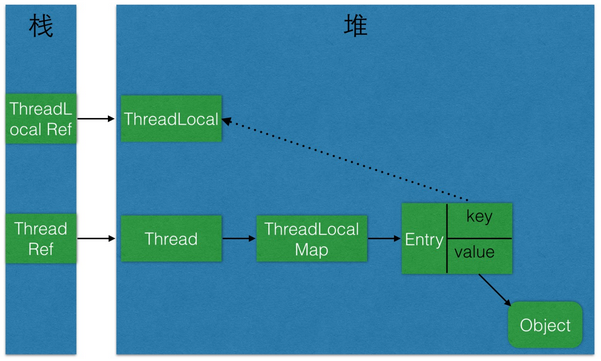
threadLocalValue:8

threadLocalValue:7

threadLocalValue:9

## ThreadLocal内存泄露的问题

对于ThreadLocal，一直涉及到内存的泄露问题，即当该线程不需要再操作某个ThreadLocal内的值时，应该手动的remove掉，为什么呢？我们来看看ThreadLocal与Thread的联系图：



其中虚线表示弱引用，从该图可以看出，一个Thread维持着一个ThreadLocalMap对象，而该Map对象的key又由提供该value的ThreadLocal对象弱引用提供，所以这就有这种情况：   
如果ThreadLocal不设为static的，由于Thread的生命周期不可预知，这就导致了当系统gc时将会回收它，而ThreadLocal对象被回收了，此时它对应key必定为null，这就导致了该key对应得value拿不出来了，而value之前被Thread所引用，所以就存在key为null、value存在强引用导致这个Entry回收不了，从而导致内存泄露。

所以避免内存泄露的方法，是对于ThreadLocal要设为static静态的，除了这个，还必须在线程不使用它的值是手动remove掉该ThreadLocal的值，这样Entry就能够在系统gc的时候正常回收，而关于ThreadLocalMap的回收，会在当前Thread销毁之后进行回收。

# 总结

关于Volatile关键字具有可见性，但不具有操作的原子性，而synchronized比volatile对资源的消耗稍微大点，但可以保证变量操作的原子性，保证变量的一致性，最佳实践则是二者结合一起使用。

1、对于synchronized的出现，是解决多线程资源共享的问题，同步机制采用了“以时间换空间”的方式：访问串行化，对象共享化。同步机制是提供一份变量，让所有线程都可以访问。

2、对于Atomic的出现，是通过原子操作指令+Lock-Free完成，从而实现非阻塞式的并发问题。

3、对于Volatile，为多线程资源共享问题解决了部分需求，在非依赖自身的操作的情况下，对变量的改变将对任何线程可见。

4、对于ThreadLocal的出现，并不是解决多线程资源共享的问题，而是用来提供线程内的局部变量，省去参数传递这个不必要的麻烦，ThreadLocal采用了“以空间换时间”的方式：访问并行化，对象独享化。ThreadLocal是为每一个线程都提供了一份独有的变量，各个线程互不影响。