高并发线程

# 4、ThreadLocal 一人一只笔

## 解释

　　ThreadLocal翻译成中文比较准确的叫法应该是：线程局部变量。

### 这个玩意有什么用处，或者说为什么要有这么一个东东？先解释一下，

在并发编程的时候，成员变量如果不做任何处理其实是线程不安全的，比如SimpleDateFormat是线程不安全的，在多个线程中共享这个变量是不安全的，因此共享这个对象肯定会出问题，各个线程都在操作同一个变量，显然是不行的，并且我们也知道volatile这个关键字也是不能保证线程安全的。那么在有一种情况之下，我们需要满足这样一个条件：变量是同一个，但是每个线程都使用同一个初始值，也就是使用同一个变量的一个新的副本。这种情况之下ThreadLocal就非常使用，比如说DAO的数据库连接，我们知道DAO是单例的，那么他的属性Connection就不是一个线程安全的变量。而我们每个线程都需要使用他，并且各自使用各自的。这种情况，ThreadLocal就比较好的解决了这个问题。

## 举例代码

|  |
| --- |
| public final class ConnectionUtil {  private ConnectionUtil() {}  private static final ThreadLocal<Connection> conn = new ThreadLocal<>();  public static Connection getConn() {  Connection con = conn.get();  if (con == null) {  try {  Class.forName("com.mysql.jdbc.Driver");  con = DriverManager.getConnection("url", "userName", "password");  conn.set(con);  } catch (ClassNotFoundException | SQLException e) {  // ...  }  }  return con;  }  } |

这样子，都是用同一个连接，但是每个连接都是新的，是同一个连接的副本。

那么实现机制是如何的呢？

1、先看set方法，先获取当前线程，然后通过getMap(Thread t)方法获取一个和当前线程相关的ThreadLocalMap，然后将变量的值设置到这个ThreadLocalMap对象中，当然如果获取到的ThreadLocalMap对象为空，就通过createMap方法创建。  
  
 线程隔离的秘密，就在于ThreadLocalMap这个类。

ThreadLocalMap是ThreadLocal类的一个静态内部类，它实现了键值对的设置和获取（对比Map对象来理解），

每个线程中都有一个独立的ThreadLocalMap副本，它所存储的值，只能被当前线程读取和修改。ThreadLocal类通过操作每一个线程特有的ThreadLocalMap副本，从而实现了变量访问在不同线程中的隔离。因为每个线程的变量都是自己特有的，完全不会有并发错误。还有一点就是，

ThreadLocalMap存储的键值对中的键是this对象指向的ThreadLocal对象，而值就是你所设置的对象了。

|  |
| --- |
| **public** **void** set(T value) {  Thread t = Thread.*currentThread*();  ThreadLocalMap map = getMap(t);  **if** (map != **null**)  map.set(**this**, value);  **else**  createMap(t, value); //如果没有就创建这个map  }  ThreadLocalMap getMap(Thread t) {  **return** t.threadLocals;  }  **void** createMap(Thread t, T firstValue) {  t.threadLocals = **new** ThreadLocalMap(**this**, firstValue); //this是ThreadLocal对象  }  ThreadLocal.ThreadLocalMap threadLocals = null; （在Thread类中） |

### 2、当我们在调用get()方法的时候，先获取当前线程，然后获取到当前线程的ThreadLocalMap对象，如果非空，那么取出ThreadLocal的value，否则进行初始化，初始化就是将initialValue的值（null）set到ThreadLocal中。

|  |
| --- |
| **public** T get() {  Thread t = Thread.*currentThread*();  ThreadLocalMap map = getMap(t);  **if** (map != **null**) {  ThreadLocalMap.Entry e = map.getEntry(**this**);  **if** (e != **null**)  **return** (T)e.value;  }  **return** setInitialValue();  }  **protected** T initialValue() {  **return** **null**;  }  **private** T setInitialValue() {  T value = initialValue();  Thread t = Thread.*currentThread*();  ThreadLocalMap map = getMap(t);  **if** (map != **null**)  map.set(**this**, value);  **else**  createMap(t, value);  **return** value;  }  ThreadLocalMap getMap(Thread t) {  **return** t.threadLocals;  } |

### 3、总结：当我们调用get方法的时候，其实每个当前线程中都有一个ThreadLocalMap。每次获取或者设置都是对该ThreadLocal进行的操作，是与其他线程分开的。从本质来讲，就是每个线程都维护了一个map，而这个map的key就是threadLocal，而值就是我们set的那个值，每次线程在get的时候，都从自己的变量中取值，既然从自己的变量中取值，那肯定就不存在线程安全问题，总体来讲，ThreadLocal这个变量的状态根本没有发生变化，他仅仅是充当一个key的角色，另外提供给每一个线程一个初始值

### 4、应用场景：当很多线程需要多次使用同一个对象，并且需要该对象具有相同初始化值的时候最适合使用ThreadLocal。

### 5、[ThreadLocal 与 Synchronized区别](http://www.cnblogs.com/xhyouyou/p/6932286.html)

**相同**：ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。  
**不同**：Synchronized同步机制采用了“以时间换空间”的方式，仅提供一份变量，让不同的线程排队访问；而ThreadLocal采用了“以空间换时间”的方式，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。

**以时间换空间**->即枷锁方式，某个区域代码或变量只有一份节省了内存，但是会形成很多线程等待现象，因此浪费了时间而节省了空间。  
**以空间换时间**->为每一个线程提供一份变量，多开销一些内存，但是呢线程不用等待，可以一起执行而相互之间没有影响。

**小结**：ThreadLocal是解决线程安全问题一个很好的思路，它通过为每个线程提供一个独立的变量副本解决了变量并发访问的冲突问题。在很多情况下，ThreadLocal比直接使用synchronized同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

# 15、倒计时器CountDownLatch

## 解释：

把门锁起来，不让里面的线程跑出来，这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。

下面代码意思就是，等待上面的10个线程全部结束，再执行主线程代码

|  |
| --- |
| **public** **class** CountDownLatchDemo **implements** Runnable {  //计数数量为10，表示需要有10个线程完成任务等待在CountDownLatch 上的线程才能执行  **static** **final** CountDownLatch *end* = **new** CountDownLatch(10);  **static** **final** CountDownLatchDemo *demo* = **new** CountDownLatchDemo();  @Override  **public** **void** run() {  **try** {  System.*out*.println(Thread.*currentThread*().getName());  Thread.*sleep*(**new** Random().nextInt(3) \* 1000);  System.*out*.println("check complete");  *end*.countDown(); //通知CountDownLatch 一个线程已经完成了任务 倒计时可以减一了  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **public** **static** **void** main(String args[]) **throws** InterruptedException {  ExecutorService executorService = Executors.*newFixedThreadPool*(10); //固定开启10个线程  **for** (**int** i = 0; i < 10; i++) {  executorService.submit(*demo*);  }  //等待检查  *end*.await(); //要求主线程等待所有的10个检查任务全部完成，主线程才能继续运行  //发射火箭  System.*out*.println("Fire!");  executorService.shutdown();  }  } |

# 16、循环栅栏 CyclicBarrier

CyclicBarrier 是一种多线程并发控制工具，和CountDownLatch,相似，但是它更强大，它可以理解为循环栅栏，栅栏就是一种障碍物，比如通常在私人宅邸周围有栅栏，阻止闲杂人等进入，这里当然是用来阻止线程继续运行，要求线程在栅栏处等待前面Cyclec意为循环，也就是说这个计时器可以反复使用，比如，假设计数器设置为10，那么凑齐10个后，计数器归0，紧接着开始凑齐下一批10个线程

# 可见性与原子性

   可见性：一个线程对共享变量的修改，更够及时的被其他线程看到  
   原子性：即不可再分了，不能分为多步操作。比如赋值或者return。比如"a = 1;"和 "return a;"这样的操作都具有原子性。类似"a += b"这样的操作不具有原子性，在某些JVM中"a += b"可能要经过这样三个步骤：  
①　取出a和b  
②　计算a+b  
③　将计算结果写入内存

**(1)Synchronized：保证可见性和原子性**  
    Synchronized能够实现原子性和可见性；在Java内存模型中，synchronized规定，线程在加锁时，先清空工作内存→在主内存中拷贝最新变量的副本到工作内存→执行完代码→将更改后的共享变量的值刷新到主内存中→释放互斥锁。

**(2)Volatile：保证可见性，但不保证操作的原子性**

**[java]**  [copy](http://blog.csdn.net/guyuealian/article/details/52525724)

1. Private **int** Num=0;
2. Num++;//Num不是原子操作

Num不是原子操作，因为其可以分为：读取Num的值，将Num的值+1，写入最新的Num的值。  
    对于Num++;操作，线程1和线程2都执行一次，最后输出Num的值可能是：1或者2  
   【解释】

输出结果1的解释：

当线程1执行Num++;语句时，先是读入Num的值为0，倘若此时让出CPU执行权，线程获得执行，线程2会重新从主内存中，读入Num的值还是0，然后线程2执行+1操作，最后把Num=1刷新到主内存中； 线程2执行完后，线程1由开始执行，但之前已经读取的Num的值0，所以它还是在0的基础上执行+1操作，也就是还是等于1，并刷新到主内存中。所以最终的结果是1

    一般在多线程中使用volatile变量，为了安全，对变量的写入操作不能依赖当前变量的值：如Num++或者Num=Num\*5这些操作。

**(3)Synchronized和Volatile的比较**  
    1）Synchronized保证内存可见性和操作的原子性  
    2）Volatile只能保证内存可见性  
    3）Volatile不需要加锁，比Synchronized更轻量级，并不会阻塞线程（volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。）  
    4）volatile标记的变量不会被编译器优化,而synchronized标记的变量可以被编译器优化（如编译器重排序的优化）.  
    5）volatile是变量修饰符，仅能用于变量，而synchronized是一个方法或块的修饰符。  
      volatile本质是在告诉JVM当前变量在寄存器中的值是不确定的，使用前，需要先从主存中读取，因此可以实现可见性。而对n=n+1,n++等操作时，volatile关键字将失效，不能起到像synchronized一样的线程同步（原子性）的效果。

### volatile关键字的作用

一个非常重要的问题，是每个学习、应用多线程的Java程序员都必须掌握的。理解volatile关键字的作用的前提是要理解Java内存模型，这里就不讲Java内存模型了，可以参见第31点，volatile关键字的作用主要有两个：

（1）多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据

         深入来说：通过加入内存屏障和禁止重排序优化来实现的。

* 对volatile变量执行写操作时，会在写操作后加入一条store屏障指令
* 对volatile变量执行读操作时，会在读操作前加入一条load屏障指令

         通俗地讲：volatile变量在每次被线程访问时，都强迫从主内存中重读该变量的值，而当该变量发生变化时，又会强迫线程将最新的值刷新到主内存。这样任何时刻，不同的线程总能看到该变量的最新值。

# 17、Th