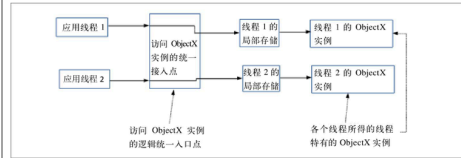
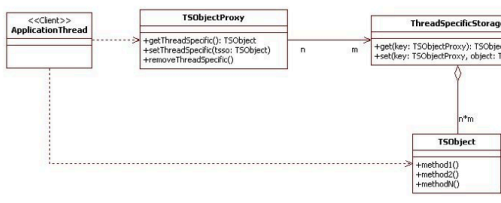
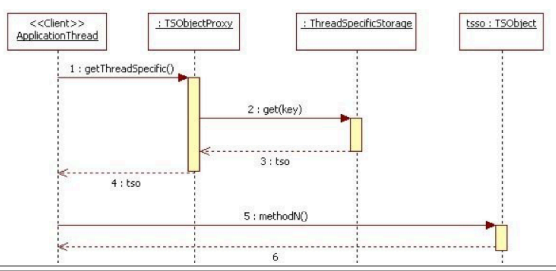


## CH10 Thread Specific Storage

2017年4月30日 星期日  
下午2:32

功能	线程私有变量局限	1.通用性不强 2.可通过run0之外的方法来访问变量，可能导致私有变量暴露给其他线程
	方法	<p>如果多个线程都需要使用某个类TObject，而为了实现线程安全和避免与锁相关的问题我们又不希望这些线程共享TObject的实例。那么我们可以使这些线程中的每一个线程都获得一个且仅一个TObject实例。这样的TObject实例就被称为线程特有对象（Thread Specific Object），它只会被一个线程持有，该线程对其状态的修改不会影响到其他线程。并且，为了隐藏相关实现细节以便于这些线程获取所需的线程特有对象，我们引入一个线程特有对象的代理类TObjectProxy。不同的线程使用同一个TObjectProxy实例可以获得所需的线程特有对象，这样就形成了一种效果：不同的线程使用统一的访问接入点（TObjectProxy）可以获取该线程所特有的TObject实例。这就是Thread Specific Storage模式的核心思想，如图10-1所示。</p>  <p>图10-1. Thread Specific Storage模式核心思想示意图</p> <p>Java 1.2引入的标准库类java.lang.ThreadLocal就相当于图10-1中的访问线程特有变量的逻辑统一入口点。读者如果对ThreadLocal已经有所理解，也不妨继续往下看。本章其他节的内容也有助于读者深入理解ThreadLocal，并了解在实战中应用ThreadLocal需要注意的重要问题。</p>
类图	 <p>图10-2. Thread Specific Storage模式的类图</p> <ul style="list-style-type: none"><li>● <b>ApplicationThread</b>：Thread Specific Storage模式的客户端，表示各个应用线程。</li><li>● <b>TObjectProxy</b>：用于访问线程特有对象的代理对象。其主要方法及职责如下。<ul style="list-style-type: none"><li>■ <b>getThreadSpecific</b>：获取与其所属TObjectProxy实例相关联的线程特有对象实例。</li><li>■ <b>setThreadSpecific</b>：建立其所属TObjectProxy实例与指定线程特有对象实例的关联。</li><li>■ <b>removeThreadSpecific</b>：删除其所属TObjectProxy实例与线程特有对象实例的关联。</li></ul></li><li>● <b>TObject</b>：表示线程特有对象。具体的类型由应用决定。</li><li>● <b>ThreadSpecificStorage</b>：线程特有存储。该参与者实例可以理解为一个Map，每个线程都有这样一个Map。这个Map存储了TObjectProxy实例到TObject实例的映射。其主要方法及职责如下。<ul style="list-style-type: none"><li>■ <b>get</b>：获取与指定TObjectProxy实例关联的TObject实例。</li><li>■ <b>set</b>：设置指定TObjectProxy实例与指定TObject实例的关联关系。</li></ul></li></ul>	
	 <p>第1步：客户端代码调用TObjectProxy实例的getThreadSpecific方法。</p> <p>第2、3步：getThreadSpecific方法调用与当前线程关联的ThreadSpecificStorage实例的get方法，并得到其返回值tso。</p> <p>第4步：getThreadSpecific方法将tso作为其返回值返回。</p> <p>第5、6步：客户端代码调用tso的相关方法。</p>	
使用时考虑点	优点	<p>Thread Specific Storage模式提升了计算效率。Thread Specific Storage模式使得我们可以在不使用锁的情况下实现线程安全，从而避免了锁的开销以及由锁带来的相关问题，如上下文切换、死锁等。</p> <p>Thread Specific Storage模式易于使用。Thread Specific Storage模式通过引入TObjectProxy这个参与者隐藏了其相关实现细节，客户端代码只需要与TObjectProxy参与者实例打交道即可获取所需的线程特有对象。</p>
	缺点	<p>Thread Specific Storage模式隐藏了系统结构。Thread Specific Storage模式的TObjectProxy参与者一方面使得客户端代码变得简单，另一方面也隐藏了应用中的各个对象间的关系，从而可能使应用更加难于理解。</p> <p>Thread Specific Storage模式鼓励使用全局对象。Thread Specific Storage模式的客户端代码通常是多个线程共用一个TObjectProxy（ThreadLocal）实例获取其所需的线程特有对象。这个共用的TObjectProxy实例就相当于一个全局变量，而全局变量的使用与目前广为接受的观点是相左的。</p>
场景	场景1	<p>场景一、需要使用非线性程安全对象，但又不希望引入锁。如果多个线程需要使用非线性程安全的对象，而我们又不希望该对象被多个线程共享，因为共享往往意味着需要引入锁以保证线程安全。此时可以使用Thread Specific Storage模式，使得各个线程拥有其特有的非线性程安全对象实例。该场景的一个典型例子是将非线性程安全的SimpleDateFormat改用ThreadLocal包装，以供多个线程使用而不必引入锁，如清单10-4所示。</p> <p>例子：电子书 2299</p>
	场景2	<p>场景二、使用线程安全对象，但希望避免其使用的锁的开销和相关问题：线程安全的对象虽然可以被多个线程共享，但是由于其可能使用了锁来保证线程安全，而某些情况下我们可能不希望看到锁的开销以及由锁可能引起的相关问题（如死锁）。此时，我们可以将线程安全的对象看成非线性程安全的对象来应用Thread Specific Storage模式。因此，这种场景就转化成场景一。只不过，此时使用Thread Specific Storage模式的主要意图在于避免锁的开销，当然线程安全也是有保障的。本章案例对Thread Specific Storage模式的使用就属于这种场景。</p>
实现要领	场景3	<p>场景三、隐式参数传递：线程特有对象在一个具体的线程中，它是线程全局可见的。某个类的方法中设置的线程特有对象对于该方法调用的其他类的方法也是可见的。这就可以形成隐式传递参数的效果，即一个类的方法调用另一个类的方法时，前者向后传递数据可以借助ThreadLocal而不必通过方法参数传递。不过，也有的观点认为隐式参数传递使得系统难于理解。如清单10-5所示代码展示了这种使用场景。</p> <p>例子：电子书 2315</p>
	场景4	<p>场景四、特定于线程的单例（Singleton）模式：广为使用的单例模式所实现的是，对于一个JVM中的一个类加载器而言，某个类有且仅有一个实例。如果对于某个类，希望每个线程有且仅有该类的一个实例，那么就可以使用Thread Specific Storage模式。Thread Specific Storage模式中，同一个应用线程多次调用同一个TObjectProxy实例所得到的线程特有对象实例都是同一个实例（只要该线程没有调用TObjectProxy实例的setThreadSpecific方法替换过线程特有对象实例）。</p>
伪内存池	预防死锁	<p>在线程池环境下使用Thread Specific Storage模式需要谨慎。这是因为线程池环境下使用线程特有对象可能导致数据错乱的现象。线程池环境下，一个工作者线程往往先后用于执行多个不同的任务。对于线程池中的一个工作者线程而言，如果它所执行的一个任务更换或者修改了该线程的线程特有对象，那么该工作者线程继续执行其他任务的时候，这些任务所“看到”的是一个新的线程特有对象。这种场景下，如果线程特有对象的状态与当前工作者线程所处理的任务有关，则会导致其他任务在执行的时候“看到”了本不属于它们的数据，产生了错乱。</p> <p>因此，在线程池环境下使用线程特有对象需要考虑在适当的时间和地方清理线程特有对象，以便同一个工作者线程在处理其他任务的时候不会产生数据错乱。清理线程特有对象只需要调用获取相应线程特有对象实例时所用的ThreadLocal实例的remove方法即可。当然，如果我们进一步分析就不难发现，这种情形下使用Thread Specific Storage模式所得到的线程特有对象其实更像是“任务特有对象”。因为每个任务都需要自己的一个线程特有对象实例。</p>
	伪内存池	<p>空而相制</p>

漏及内存泄漏	key是WeakReference、value是强引用	
	<p>首先，我们先看下ThreadLocal是如何实现Thread Specific Storage模式的。JDK标准库类java.lang.Thread的实例变量threadLocals会引用一个ThreadLocal.ThreadLocalMap实例。该实例相当于Thread Specific Storage模式的ThreadSpecificStorage参与者实例。从效果上看可以将其理解为一个WeakHashMap，该WeakHashMap包含若干条目（Entry）。每个条目的键（Key）为指向一个ThreadLocal实例的弱引用（Weak Reference），值（Value）指向线程特有对象（TObject）实例，如图10-4所示。</p> <p>ThreadLocalMap的条目对ThreadLocal实例的引用是弱引用，因此它不会阻止垃圾回收器（Garbage Collector）将其引用的ThreadLocal实例回收。而ThreadLocalMap的条目对线程特有对象实例的引用是强引用（Strong Reference），因此它会阻止垃圾回收器将其引用的线程特有对象实例回收。</p>	
	<p>对于某个ThreadLocal实例而言，如果在某一个时间段内该实例除了ThreadLocalMap条目对其有可达的引用（Reachable Reference）外，没有其他可达的引用，那么垃圾回收器就可以将该ThreadLocal实例回收。此时，先前引用该ThreadLocal实例的ThreadLocalMap条目由于其Key的值变为null，就成了一条无效的条目（Stale Entry）。这种无效的条目（即Key值为null，Value值指向一个线程特有对象）在其所属的ThreadLocalMap有新增条目时可能被删除掉。如果某个线程引用的ThreadLocalMap实例产生无效条目后，某段时间内该线程处于非运行状态，则在该ThreadLocalMap实例就没有新增的条目，因此其所有的无效条目在该时间段内也无法被删除。如果该线程一直处于非运行状态，则在该线程引用的ThreadLocalMap实例的无效条目永远无法被删除。所以，这种情形就会导致伪内存泄漏。</p> <p>对于某个ThreadLocal实例而言，如果在某一时间段内系统中除了ThreadLocalMap条目对其有可达的引用（弱引用）外，还有其他可达的强引用，那么，相应的ThreadLocalMap条目不会被删除，再加上ThreadLocalMap条目对线程特有对象的引用是强引用，此时只要引用该ThreadLocalMap实例的线程存在，则该条目会阻止垃圾回收器将相应的线程特有对象实例回收。如果引用该ThreadLocalMap条目的线程永远存在，则该ThreadLocalMap条目引用的ThreadLocal实例及其对应的线程特有对象实例都无法被垃圾回收，这就产生了内存泄漏。</p>	
例子	Tomcat 6.0.37 内存泄漏的和伪内存泄漏的场景 电子书 2372	
可复用代码	<p>JDK 1.2引入的标准库类java.lang.ThreadLocal可以看成是Thread Specific Storage模式的可复用实现。ThreadLocal类相当于Thread Specific Storage模式的TObjectProxy参与者。其类型参数T相当于TObject参与者。利用ThreadLocal实现Thread Specific Storage模式，应用代码只需要完成以下几件事情。</p> <p>1.【必需】创建ThreadLocal的子类（或者匿名子类）。</p> <p>2.【可选，但通常是需要的】在ThreadLocal的子类中覆盖其父类的initialValue方法，用于定义初始的线程特有对象实例。</p> <p>需要注意的是，类型为ThreadLocal的变量，其声明通常采用static final修饰。不同的线程采用同一个ThreadLocal实例即可获取所需的线程特有对象实例，因此类型为ThreadLocal的变量定义为类变量（用static修饰）即可，而无须定义为实例变量（不使用static修饰）。</p>	
JDK伪例	<p>JDK 1.7中引入的标准库类java.util.concurrent.ThreadLocalRandom就使用了Thread Specific Storage模式。ThreadLocalRandom是ThreadLocal的一个子类，其current方法返回一个属于当前线程的ThreadLocalRandom实例。这里，current方法的返回值就是一个线程特有对象实例。因此，不同线程调用ThreadLocalRandom实例的相关方法获取下一个随机数的时候，相互之间不影响。这比多个线程共享一个java.math.Random实例来获取随机数效率要高，尽管Random类也是线程安全的。</p>	
例子	背景	<p>1.复用expensive object; 2.避免互斥</p> <p>某系统需要支持验证码短信功能。该系统的用户进行一些重要操作的时候，该系统会生成一个验证码，并将其通过短信发送给用户。验证码是一个6位数的随机数字。这里，为了提高安全性，验证码的生成需要使用java.security.SecureRandom这种强随机数生成器，而非java.math.Random这种伪随机数生成器。但是，使用SecureRandom可能会涉及以下几个问题。</p> <p>一、SecureRandom实例的初始化（主要是初始化种子）可能比较耗时间。这点在JVM的宿主操作系统为Linux系统时更为明显。这与SecureRandom的内部实现有关。因此，我们希望能够复用SecureRandom实例，而不是每次需要生成一个验证码的时候就生成一个SecureRandom实例。</p> <p>二、SecureRandom用于生成随机整数的nextInt方法最终会调用一个由SecureRandom自身定义的synchronized方法。这意味着，nextInt方法的调用实际上会涉及锁。因此，如果多个线程共用同一个SecureRandom实例，那么当一个线程正在调用该实例的nextInt方法生成随机数的时候，其他线程只能等待。但是，我们不想看到这种等待：由于验证码生成后需要通过短信发送给用户，而该系统下发短信给用户涉及网络I/O这种相对慢的操作，我们希望验证码的生成能够尽量快，从而不耽误短信的下发。</p>
代码	<pre>public class ThreadSpecificSecureRandom {     //该类的唯一实例     public static final ThreadSpecificSecureRandom INSTANCE = new     ThreadSpecificSecureRandom();      /*     SECURE_RANDOM相当于模式角色: ThreadSpecificStorage.TObjectProxy.     SecureRandom相当于模式角色: ThreadSpecificStorage.TObject.     */     private static final ThreadLocal&lt;SecureRandom&gt; SECURE_RANDOM = new     ThreadLocal&lt;SecureRandom&gt;() {          @Override         protected SecureRandom initialValue() {             SecureRandom srnd;             try {                 srnd = SecureRandom.getInstance("SHA1PRNG");             } catch (NoSuchAlgorithmException e) {                 e.printStackTrace();                 srnd = new SecureRandom();             }             return srnd;         }     };      // 私有构造器     private ThreadSpecificSecureRandom() {      }      public int nextInt(int upperBound) {         SecureRandom secureRnd = SECURE_RANDOM.get();         return secureRnd.nextInt(upperBound);     }      public void setSeed(long seed) {         SecureRandom secureRnd = SECURE_RANDOM.get();         secureRnd.setSeed(seed);     } }</pre>	
	<pre>public class SmsVerificationCodeSender {     private static final ExecutorService EXECUTOR = new ThreadPoolExecutor(1,     Runtime.getRuntime().availableProcessors(), 60, TimeUnit.SECONDS,     new SynchronousQueue&lt;Runnable&gt;(), new ThreadFactory() {         @Override         public Thread newThread(Runnable r) {             Thread t = new Thread(r, "VerfCodeSender");             t.setDaemon(true);             return t;         }     }, new ThreadPoolExecutor.DiscardPolicy());      public static void main(String[] args) {         SmsVerificationCodeSender client = new SmsVerificationCodeSender();         client.sendVerificationSms("18912345678");         client.sendVerificationSms("18712345679");         client.sendVerificationSms("18612345676");          try {             Thread.sleep(100);         } catch (InterruptedException e) {             ;         }     } }</pre>	

	<pre>public void sendVerificationSms(final String msisdn) {     Runnable task = new Runnable() {         @Override         public void run() {             //生成随机数验证码             int verificationCode = ThreadSpecificSecureRandom.INSTANCE                 .nextInt(9999999);             DecimalFormat df = new DecimalFormat("0000000");             String txtVerCode = df.format(verificationCode);              //发送验证码短信             sendSms(msisdn, txtVerCode);         }     };      EXECUTOR.submit(task); }  private void sendSms(String msisdn, String verificationCode) {     System.out.println("Sending verification code " + verificationCode + " to "         + msisdn); }  // 省略其他代码 }</pre>

使用 Microsoft OneNote for Mac 创建。