**ThreadLocal是什么**

早在JDK 1.2的版本中就提供java.lang.ThreadLocal，ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序。

ThreadLocal很容易让人望文生义，想当然地认为是一个“本地线程”。其实，ThreadLocal并不是一个Thread，而是Thread的局部变量，也许把它命名为ThreadLocalVariable更容易让人理解一些。

当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

从线程的角度看，目标变量就象是线程的本地变量，这也是类名中“Local”所要表达的意思。

线程局部变量并不是Java的新发明，很多语言（如IBM IBM XL FORTRAN）在语法层面就提供线程局部变量。在Java中没有提供在语言级支持，而是变相地通过ThreadLocal的类提供支持。

所以，在Java中编写线程局部变量的代码相对来说要笨拙一些，因此造成线程局部变量没有在Java开发者中得到很好的普及。

ThreadLocal的接口方法

ThreadLocal类接口很简单，只有4个方法，我们先来了解一下：

* void set(Object value)

设置当前线程的线程局部变量的值。

* public Object get()

该方法返回当前线程所对应的线程局部变量。

* public void remove()

将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是JDK 5.0新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。

* protected Object initialValue()

返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次。ThreadLocal中的缺省实现直接返回一个null。

值得一提的是，在JDK5.0中，ThreadLocal已经支持泛型，该类的类名已经变为ThreadLocal<T>。API方法 也相应进行了调整，新版本的API方法分别是void set(T value)、T get()以及T initialValue()。

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？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。我们自己就可以提供一个简单的实现版本：

代码清单1 SimpleThreadLocal

public class SimpleThreadLocal {

private Map valueMap = Collections.synchronizedMap(new HashMap());

public void set(Object newValue) {

valueMap.put(**Thread.currentThread(), newValue**);**①键为线程对象，值为本线程的变量副本**

}

public Object get() {

Thread currentThread = Thread.currentThread();

Object o = valueMap.get(currentThread);**②返回本线程对应的变量**

if (o == null && !valueMap.containsKey(currentThread)) {**③如果在Map中不存在，放到Map**

**中保存起来。**

o = initialValue();

valueMap.put(currentThread, o);

}

return o;

}

public void remove() {

valueMap.remove(Thread.currentThread());

}

public Object initialValue() {

return null;

}

}

虽然代码清单9‑3这个ThreadLocal实现版本显得比较幼稚，但它和JDK所提供的ThreadLocal类在实现思路上是相近的。

一个TheadLocal实例

下面，我们通过一个具体的实例了解一下ThreadLocal的具体使用方法。

代码清单2 SequenceNumber

package com.baobaotao.basic;

public class SequenceNumber {

**①通过匿名内部类覆盖ThreadLocal的initialValue()方法，指定初始值**

private static ThreadLocal<Integer> seqNum = new **ThreadLocal<Integer>**(){

public Integer initialValue(){

return 0;

}

};

**②获取下一个序列值**

public int getNextNum(){

seqNum.set(seqNum.get()+1);

return seqNum.get();

}

public static void main(String[] args)

{

SequenceNumber **sn** = new SequenceNumber();

**③ 3个线程共享sn，各自产生序列号**

TestClient t1 = new TestClient(**sn**);

TestClient t2 = new TestClient(**sn**);

TestClient t3 = new TestClient(**sn**);

t1.start();

t2.start();

t3.start();

}

private static class TestClient extends Thread

{

private SequenceNumber sn;

public TestClient(SequenceNumber sn) {

this.sn = sn;

}

public void run()

{

for (int i = 0; i < 3; i++) {**④每个线程打出3个序列值**

System.out.println("thread["+Thread.currentThread().getName()+

"] sn["+**sn.getNextNum()**+"]");

}

}

}

}

通常我们通过匿名内部类的方式定义ThreadLocal的子类，提供初始的变量值，如例子中①处所示。TestClient线程产生一组序列号， 在③处，我们生成3个TestClient，它们共享同一个SequenceNumber实例。运行以上代码，在控制台上输出以下的结果：

thread[Thread-2] sn[1]

thread[Thread-0] sn[1]

thread[Thread-1] sn[1]

thread[Thread-2] sn[2]

thread[Thread-0] sn[2]

thread[Thread-1] sn[2]

thread[Thread-2] sn[3]

thread[Thread-0] sn[3]

thread[Thread-1] sn[3]

考察输出的结果信息，我们发现每个线程所产生的序号虽然都共享同一个SequenceNumber实例，但它们并没有发生相互干扰的情况，而是各自产生独立的序列号，这是因为我们通过ThreadLocal为每一个线程提供了单独的副本。

Thread同步机制的比较

ThreadLocal和线程同步机制相比有什么优势呢？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。

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。这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

而ThreadLocal则从另一个角度来解决多线程的并发访问。ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线 程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编 写多线程代码时，可以把不安全的变量封装进ThreadLocal。

由于ThreadLocal中可以持有任何类型的对象，低版本JDK所提供的get()返回的是Object对象，需要强制类型转换。但JDK 5.0通过泛型很好的解决了这个问题，在一定程度地简化ThreadLocal的使用，代码清单 9 2就使用了JDK 5.0新的ThreadLocal<T>版本。

概括起来说，对于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，而ThreadLocal采用了“以空间换时间”的方式。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。

Spring使用ThreadLocal解决线程安全问题

我们知道在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用 域。就是因为Spring对一些Bean（如RequestContextHolder、 TransactionSynchronizationManager、LocaleContextHolder等）中非线程安全状态采用 ThreadLocal进行处理，让它们也成为线程安全的状态，因为有状态的Bean就可以在多线程中共享了。

一般的Web应用划分为展现层、服务层和持久层三个层次，在不同的层中编写对应的逻辑，下层通过接口向上层开放功能调用。在一般情况下，从接收请求到返回响应所经过的所有程序调用都同属于一个线程，如图9‑2所示：

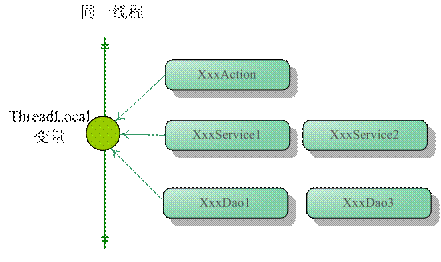


图1同一线程贯通三层

这样你就可以根据需要，将一些非线程安全的变量以ThreadLocal存放，在同一次请求响应的调用线程中，所有关联的对象引用到的都是同一个变量。

下面的实例能够体现Spring对有状态Bean的改造思路：

代码清单3 TopicDao：非线程安全

public class TopicDao {

private Connection conn;**①一个非线程安全的变量**

public void addTopic(){

Statement stat = conn.createStatement();**②引用非线程安全变量**

…

}

}

由于①处的conn是成员变量，因为addTopic()方法是非线程安全的，必须在使用时创建一个新TopicDao实例（非singleton）。下面使用ThreadLocal对conn这个非线程安全的“状态”进行改造：

代码清单4 TopicDao：线程安全

import java.sql.Connection;

import java.sql.Statement;

public class TopicDao {

**①使用ThreadLocal保存Connection变量**

private static **ThreadLocal<Connection>** connThreadLocal = new ThreadLocal<Connection>();

public static Connection getConnection(){

**②如果connThreadLocal没有本线程对应的Connection创建一个新的Connection，**

**并将其保存到线程本地变量中。**

if (connThreadLocal.get() == null) {

Connection conn = ConnectionManager.getConnection();

connThreadLocal.set(conn);

return conn;

}else{

return connThreadLocal.get();**③直接返回线程本地变量**

}

}

public void addTopic() {

**④从ThreadLocal中获取线程对应的Connection**

Statement stat = getConnection().createStatement();

}

}

不同的线程在使用TopicDao时，先判断connThreadLocal.get()是否是null，如果是null，则说明当前线程还没有对 应的Connection对象，这时创建一个Connection对象并添加到本地线程变量中；如果不为null，则说明当前的线程已经拥有了 Connection对象，直接使用就可以了。这样，就保证了不同的线程使用线程相关的Connection，而不会使用其它线程的 Connection。因此，这个TopicDao就可以做到singleton共享了。

当然，这个例子本身很粗糙，将Connection的ThreadLocal直接放在DAO只能做到本DAO的多个方法共享Connection时 不发生线程安全问题，但无法和其它DAO共用同一个Connection，要做到同一事务多DAO共享同一Connection，必须在一个共同的外部类 使用ThreadLocal保存Connection。

**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。

**ThreadLocal的几种误区**

 最近由于需要用到ThreadLocal，在网上搜索了一些相关资料，发现对ThreadLocal经常会有下面几种**误解**  
  
 一、ThreadLocal是java线程的一个实现  
      ThreadLocal的确是和java线程有关，不过它并不是java线程的一个实现，它只是用来维护本地变量。针对每个线程，提供自己的变量版本，主要是为了避免线程冲突，每个线程维护自己的版本。彼此独立，修改不会影响到对方。

 二、ThreadLocal是相对于每个session的

        ThreadLocal顾名思义，是针对线程。在java web编程上，每个用户从开始到会话结束，都有自己的一个session标识。但是ThreadLocal并不是在会话层上。其 实，Threadlocal是独立于用户session的。它是一种服务器端行为，当服务器每生成一个新的线程时，就会维护自己的 ThreadLocal。对于这个误解，个人认为应该是开发人员在本地基于一些应用服务器测试的结果。众所周知，一般的应用服务器都会维护一套线程池，也 就是说，对于每次访问，并不一定就新生成一个线程。而是自己有一个线程缓存池。对于访问，先从缓存池里面找到已有的线程，如果已经用光，才去新生成新的线 程。所以，由于开发人员自己在测试时，一般只有他自己在测，这样服务器的负担很小，这样导致每次访问可能是共用同样一个线程，导致会有这样的误解：每个 session有一个ThreadLocal

 三、ThreadLocal是相对于每个线程的，用户每次访问会有新的ThreadLocal

  理 论上来说，ThreadLocal是的确是相对于每个线程，每个线程会有自己的ThreadLocal。但是上面已经讲到，一般的应用服务器都会维护一套 线程池。因此，不同用户访问，可能会接受到同样的线程。因此，在做基于TheadLocal时，需要谨慎，避免出现ThreadLocal变量的缓存，导 致其他线程访问到本线程变量

 四、对每个用户访问，ThreadLocal可以多用  
        可以说，ThreadLocal是一把双刃剑，用得来的话可以起到非常好的效果。但是，ThreadLocal如果用得不好，就会跟全局变量一样。代码不 能重用，不能独立测试。因为，一些本来可以重用的类，现在依赖于ThreadLocal变量。如果在其他没有ThreadLocal场合，这些类就变得不 可用了。个人觉得ThreadLocal用得很好的几个应用场合，值得参考

  1、存放当前session用户：quake want的jert

  2、存放一些context变量，比如webwork的ActionContext

  3、存放session，比如Spring hibernate orm的session