# ThreadLocal设计模式

转载自：

http://lgbolgger.iteye.com/blog/2117216

ThreadLocal设计模式使用的很频繁，我们会经常在各大框架中找到它们的踪影，比如Spring AOP等。ThreadLocal可以理解为：

（1）ThreadLocal所操作的数据是线程间不共享的，它不是用来解决多个线程竞争同一资源的多线程问题。

（2）ThreadLocal所操作的数据主要用于线程内共享数据，可以避免同一线程内函数间的传参问题。

ThreadLocal更像是一个操作线程数据的工具类，哪个线程调用它，它就操作哪个线程的数据。下面看两个案例。

## 案例分析

案例一：ThreadLocal所操作的数据是线程间不共享的，是线程间互不干扰的。

package com.kevin.threadlocal;  
  
*/\*\*  
 \** ***@author*** *kevin  
 \** ***@version*** *1.0  
 \** ***@des*** *\** ***@data*** *2017/6/2 16:41  
 \*/*public class ThreadLocalTest1 {  
  
 public static void main(String[] args) {  
 *test1*();  
 }  
  
 private static void test1() {  
 Thread t1 = new Thread() {  
 @Override  
 public void run() {  
 System.*out*.println(this.getName() + "开始值: " + ThreadUtil.*getName*());  
 ThreadUtil.*setName*("线程0");  
 *sleepTime*(1000 \* 4);  
 System.*out*.println(this.getName() + "设定值: " + ThreadUtil.*getName*());  
 }  
 };  
  
 Thread t2 = new Thread() {  
 @Override  
 public void run() {  
 *sleepTime*(1000 \* 2);  
 System.*out*.println(this.getName() + "开始值: " + ThreadUtil.*getName*());  
 ThreadUtil.*setName*("线程1");  
 System.*out*.println(this.getName() + "设定值: " + ThreadUtil.*getName*());  
 }  
 };  
  
 t1.start();  
 t2.start();  
 }  
  
 private static void sleepTime(long millis) {  
 try {  
 Thread.*sleep*(millis);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

其中ThreadUtil就是用ThreadLocal来实现的。ThreadLocal完全可以这样理解：它就是操作线程数据的工具类，哪个线程调用它的get或set方法，它就会操作调用它的线程的数据。

package com.kevin.threadlocal;  
  
*/\*\*  
 \** ***@author*** *kevin  
 \** ***@version*** *1.0  
 \** ***@des*** *\** ***@data*** *2017/6/2 16:45  
 \*/*public class ThreadUtil {  
  
 private static ThreadLocal<String> *nameLocal* = new ThreadLocal<>();  
  
 public static String getName() {  
 return *nameLocal*.get();  
 }  
  
 public static void setName(String name) {  
 *nameLocal*.set(name);  
 }  
}

运行结果如下：

Thread-0开始值: null

Thread-1开始值: null

Thread-1设定值: 线程1

Thread-0设定值: 线程0

分析如下：

（1）每个线程都有一个map用于存放数据。首先是线程t1调用ThreadUtil.getName()，即调用nameLocal.get()。由于此时调用nameLocal.get()方法的线程是t1，因此nameLocal会从线程t1的map集合中取数据，其中key为nameLocal对象，由于之前未存放数据，因此返回null。

（2）紧接着t1调用ThreadUtil.setName("线程0")，即调用nameLocal.set("线程0")。由于此时调用nameLocal.set()方法的线程是t1，因此nameLocal将数据存放到线程t1的map集合中，其中key为nameLocal对象，然后线程t1休眠4s。

（3）线程t2刚开始休眠2s，然后调用ThreadUtil.getName()，即调用nameLocal.get()。由于此时调用nameLocal.get()方法的线程是t2，因此nameLocal会从线程t2的map集合中取数据，其中key为nameLocal对象，由于之前未存放数据，因此返回null。

（4）紧接着t2调用ThreadUtil.setName("线程1")，即调用nameLocal.set("线程1")。由于此时调用nameLocal.set()方法的线程是t2，因此nameLocal将数据存放到线程t2的map集合中，其中key为nameLocal对象。

（5）然后线程t2调用ThreadUtil.getName()，即调用nameLocal.get()。由于此时调用nameLocal.get()方法的线程是t2，因此nameLocal会从线程t2的map集合中取数据，其中key为nameLocal对象，取到数据“线程1”。

（6）最后线程t1调用ThreadUtil.getName()，即调用nameLocal.get()。由于此时调用nameLocal.get()方法的线程是t1，因此nameLocal会从线程t1的map集合中取数据，其中key为nameLocal对象，取到数据“线程0”。

从以上过程可以看出，ThreadLocal对象所操作的数据是线程所独有的数据，虽然两个线程使用的是同一个ThreadLocal对象，但该ThreadLocal对象所操作的数据是线程间不共享、互不干扰的，所以ThreadLocal设计模式不适用于多个线程竞争同一资源的情况。

案例二：ThreadLocal所操作的数据主要用于线程内共享数据。

package com.kevin.threadlocal;  
  
*/\*\*  
 \** ***@author*** *kevin  
 \** ***@version*** *1.0  
 \** ***@des*** *\** ***@data*** *2017/6/2 17:02  
 \*/*public class ThreadLocalTest2 {  
  
 public static void main(String[] args) {  
 *test2*();  
 }  
  
 private static void test2() {  
 Thread t1 = new Thread() {  
 @Override  
 public void run() {  
 func1();  
 func2();  
 }  
  
 private void func2() {  
 System.*out*.println("在函数2中获取到当前线程的name数据为:" + ThreadUtil.*getName*());  
 }  
  
 private void func1() {  
 String name = "my own name";  
 ThreadUtil.*setName*(name);  
 System.*out*.println("在函数1设置当前线程的name数据为: " + name);  
 }  
 };  
  
 t1.start();  
 }  
}

运行结果如下：

在函数1设置当前线程的name数据为: my own name

在函数2中获取到当前线程的name数据为:my own name

分析如下；

（1）func1和func2都在线程t1中执行。在func1中调用ThreadUtil.setName()，即调用nameLocal.set()方法，由于此时调用nameLocal.set()方法的线程是t1，因此nameLocal将数据存放到线程t1的map集合中，其中key为nameLocal对象。

（2）在func2中调用ThreadUtil.getName()，即调用nameLocal.get()方法，由于此时调用nameLocal.get()方法的线程是t1，因此nameLocal会从线程t1的map集合中取数据，其中key为nameLocal对象，取到数据“my own name”。

综上所述，通过ThreadLocal对象可以把线程内要共享的数据存放到该线程的map集合中，方便随时获取，实现共享。不然的话，要想实现数据的共享，则需要fun1将数据作为参数传给fun2，考虑到实际情况是某个线程中几十个函数嵌套调用，如果不将线程内共享的数据放进该线程的map集合中，而是将共享数据作为参数传递来传递去，将会一片混乱，杂乱无章，有些函数需要这些参数，有些函数不需要。采用ThreadLocal模式，将数据放进线程的map集合中，想要获取时随时都可以通过ThreadLocal对象来获取，方便快捷。

## 源码分析

下面从源码的角度出发，来说明ThreadLocal对象操作的是调用它的线程的map集合中的数据，这就需要看ThreadLocal的get和set方法。

get()方法如下：

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();  
}

（1）首先是获取调用该ThreadLocal对象的当前线程，例如线程t1调用了ThreadLocal对象的get方法，则此时的Thread t就是t1。

（2）Thread类拥有一个ThreadLocalMap属性，如下：

ThreadLocal.ThreadLocalMap threadLocals = null;

getMap(t)就是获取线程t的threadLocals，如下：

ThreadLocalMap getMap(Thread t) {  
 return t.threadLocals;  
}

如果线程t中的ThreadLocalMap不为空，则以this（即当前的ThreadLocal对象）作为key来获取对应的value。

如果线程t中的ThreadLocalMap为空，则调用setInitialValue()进行初始化，如下：

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;  
}

首先通过initialValue()来获取初始值，默认为null；然后判断当前线程中的ThreadLocalMap是否为空，如果为空则创建一个，同时将初始值设置进ThreadLocalMap中，其中key即使ThreadLocal对象本身。

initialValue()如下：

protected T initialValue() {  
 return null;  
}

我们可以重写该方法来设置一个初始值。

set()方法如下：

public void set(T value) {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 if (map != null)  
 map.set(this, value);  
 else  
 createMap(t, value);  
}

也很清晰明了，首先获取当前线程，然后获取该线程的ThreadLocalMap对象，最后向其中存放数据。