# TODO

一、    jdk源码

（一）代码

|  |
| --- |
| package java**.**lang**;**  **import** java**.**lang**.**ref**.\*;**  **import** java**.**util**.**concurrent**.**atomic**.**AtomicInteger**;**    public class ThreadLocal**<**T**>** **{**        private final int threadLocalHashCode **=** nextHashCode**();**  //ThreadLocal实例hash值，用来区分不同实例        private static AtomicInteger nextHashCode **=**         //可以看作hash值的一个基值  **new** AtomicInteger**();**        private static final int HASH\_INCREMENT **=** 0x61c88647**;**   //hash值每次增加量        private static int nextHashCode**()** **{**  **return** nextHashCode**.**getAndAdd**(**HASH\_INCREMENT**);**  **}**      //初始化函数      protected T initialValue**()** **{**  **return** **null;**  **}**      //无参构造函数      public ThreadLocal**()** **{**  **}**      //注意：每个线程中都是有一个ThreadLocalMap对象，它属于Map类型，其中key为ThreadLocal对象，value为某个对象。     //从当前线程的ThreadLocalMap中取出 key为当前ThreadLocal对象 的value对象，其实key值与ThreadLocal的threadLocalHashCode值有关      public T get**()** **{**          Thread t **=** Thread**.**currentThread**();**   //得到当前线程          ThreadLocalMap map **=** getMap**(**t**);**     //得到当前线程的ThreadLocalMap对象  **if** **(**map **!=** **null)** **{**      //如果map不为null,              ThreadLocalMap**.**Entry e **=** map**.**getEntry**(this);**  //得到map中Entry实体对象；  **if** **(**e **!=** **null)**   //如果e不为空，则取出Entry对象中的value值，然后返回  **return** **(**T**)**e**.**value**;**  **}**  **return** setInitialValue**();**  //如果map为null,则创建ThreadLocalMap对象，                                     //并且创建一个空的T对象放到map中，最后返回null  **}**          private T setInitialValue**()** **{**          T value **=** initialValue**();**          Thread t **=** Thread**.**currentThread**();**  //得到当前线程          ThreadLocalMap map **=** getMap**(**t**);**  //得到当前线程的ThreadLocalMap对象  **if** **(**map **!=** **null)**          //map不为空，则将value放到map              map**.**set**(this,** value**);**  **else**              createMap**(**t**,** value**);**   //否则创建map,然后将value放到map中  **return** value**;**  **}**      //把value放到当前线程的ThreadLocalMap对象中去，其中key值与当前ThreadLocal对象的threadLocalHashCode值有关      public void set**(**T value**)** **{**          Thread t **=** Thread**.**currentThread**();**          ThreadLocalMap map **=** getMap**(**t**);**  **if** **(**map **!=** **null)**              map**.**set**(this,** value**);**  **else**              createMap**(**t**,** value**);**  **}**      //删除当前线程的 ThreadLocalMap对象中 key为当前ThreadLocal 的Entry(包含key/value)      public void remove**()** **{**           ThreadLocalMap m **=** getMap**(**Thread**.**currentThread**());**  **if** **(**m **!=** **null)**               m**.**remove**(this);**  **}**        //取得TheadLocalMap      ThreadLocalMap getMap**(**Thread t**)** **{**  **return** t**.**threadLocals**;**  **}**        //创建TheadLocalMap      void createMap**(**Thread t**,** T firstValue**)** **{**          t**.**threadLocals **=** **new** ThreadLocalMap**(this,** firstValue**);**  **}**          static ThreadLocalMap createInheritedMap**(**ThreadLocalMap parentMap**)** **{**  **return** **new** ThreadLocalMap**(**parentMap**);**  **}**          T childValue**(**T parentValue**)** **{**  **throw** **new** UnsupportedOperationException**();**  **}**      //静态内部类ThreadLcoalMap      static class ThreadLocalMap **{**            static class Entry **extends** WeakReference**<**ThreadLocal**>** **{**              /\*\* The value associated with this ThreadLocal. \*/              Object value**;**                Entry**(**ThreadLocal k**,** Object v**)** **{**  **super(**k**);**                  value **=** v**;**  **}**  **}**            private static final int INITIAL\_CAPACITY **=** 16**;**              private Entry**[]** table**;**            private int size **=** 0**;**            private int threshold**;** // Default to 0            private void setThreshold**(**int len**)** **{**              threshold **=** len **\*** 2 **/** 3**;**  **}**            private static int nextIndex**(**int i**,** int len**)** **{**  **return** **((**i **+** 1 **<** len**)** **?** i **+** 1 **:** 0**);**  **}**            private static int prevIndex**(**int i**,** int len**)** **{**  **return** **((**i **-** 1 **>=** 0**)** **?** i **-** 1 **:** len **-** 1**);**  **}**            ThreadLocalMap**(**ThreadLocal firstKey**,** Object firstValue**)** **{**              table **=** **new** Entry**[**INITIAL\_CAPACITY**];**              int i **=** firstKey**.**threadLocalHashCode **&** **(**INITIAL\_CAPACITY **-** 1**);**              table**[**i**]** **=** **new** Entry**(**firstKey**,** firstValue**);**              size **=** 1**;**              setThreshold**(**INITIAL\_CAPACITY**);**  **}**            private ThreadLocalMap**(**ThreadLocalMap parentMap**)** **{**              Entry**[]** parentTable **=** parentMap**.**table**;**              int len **=** parentTable**.**length**;**              setThreshold**(**len**);**              table **=** **new** Entry**[**len**];**    **for** **(**int j **=** 0**;** j **<** len**;** j**++)** **{**                  Entry e **=** parentTable**[**j**];**  **if** **(**e **!=** **null)** **{**                      ThreadLocal key **=** e**.**get**();**  **if** **(**key **!=** **null)** **{**                          Object value **=** key**.**childValue**(**e**.**value**);**                          Entry c **=** **new** Entry**(**key**,** value**);**                          int h **=** key**.**threadLocalHashCode **&** **(**len **-** 1**);**  **while** **(**table**[**h**]** **!=** **null)**                              h **=** nextIndex**(**h**,** len**);**                          table**[**h**]** **=** c**;**                          size**++;**  **}**  **}**  **}**  **}**            private Entry getEntry**(**ThreadLocal key**)** **{**              int i **=** key**.**threadLocalHashCode **&** **(**table**.**length **-** 1**);**              Entry e **=** table**[**i**];**  **if** **(**e **!=** **null** **&&** e**.**get**()** **==** key**)**  **return** e**;**  **else**  **return** getEntryAfterMiss**(**key**,** i**,** e**);**  **}**              private Entry getEntryAfterMiss**(**ThreadLocal key**,** int i**,** Entry e**)** **{**              Entry**[]** tab **=** table**;**              int len **=** tab**.**length**;**    **while** **(**e **!=** **null)** **{**                  ThreadLocal k **=** e**.**get**();**  **if** **(**k **==** key**)**  **return** e**;**  **if** **(**k **==** **null)**                      expungeStaleEntry**(**i**);**  **else**                      i **=** nextIndex**(**i**,** len**);**                  e **=** tab**[**i**];**  **}**  **return** **null;**  **}**            private void set**(**ThreadLocal key**,** Object value**)** **{**                // We don't use a fast path as with get() because it is at              // least as common to use set() to create new entries as              // it is to replace existing ones, in which case, a fast              // path would fail more often than not.                Entry**[]** tab **=** table**;**              int len **=** tab**.**length**;**              int i **=** key**.**threadLocalHashCode **&** **(**len**-**1**);**    **for** **(**Entry e **=** tab**[**i**];**                   e **!=** **null;**                   e **=** tab**[**i **=** nextIndex**(**i**,** len**)])** **{**                  ThreadLocal k **=** e**.**get**();**    **if** **(**k **==** key**)** **{**                      e**.**value **=** value**;**  **return;**  **}**    **if** **(**k **==** **null)** **{**                      replaceStaleEntry**(**key**,** value**,** i**);**  **return;**  **}**  **}**                tab**[**i**]** **=** **new** Entry**(**key**,** value**);**              int sz **=** **++**size**;**  **if** **(!**cleanSomeSlots**(**i**,** sz**)** **&&** sz **>=** threshold**)**                  rehash**();**  **}**            private void remove**(**ThreadLocal key**)** **{**              Entry**[]** tab **=** table**;**              int len **=** tab**.**length**;**              int i **=** key**.**threadLocalHashCode **&** **(**len**-**1**);**  **for** **(**Entry e **=** tab**[**i**];**                   e **!=** **null;**                   e **=** tab**[**i **=** nextIndex**(**i**,** len**)])** **{**  **if** **(**e**.**get**()** **==** key**)** **{**                      e**.**clear**();**                      expungeStaleEntry**(**i**);**  **return;**  **}**  **}**  **}**            private void replaceStaleEntry**(**ThreadLocal key**,** Object value**,**                                         int staleSlot**)** **{**              Entry**[]** tab **=** table**;**              int len **=** tab**.**length**;**              Entry e**;**                // Back up to check for prior stale entry in current run.              // We clean out whole runs at a time to avoid continual              // incremental rehashing due to garbage collector freeing              // up refs in bunches (i.e., whenever the collector runs).              int slotToExpunge **=** staleSlot**;**  **for** **(**int i **=** prevIndex**(**staleSlot**,** len**);**  **(**e **=** tab**[**i**])** **!=** **null;**                   i **=** prevIndex**(**i**,** len**))**  **if** **(**e**.**get**()** **==** **null)**                      slotToExpunge **=** i**;**                // Find either the key or trailing null slot of run, whichever              // occurs first  **for** **(**int i **=** nextIndex**(**staleSlot**,** len**);**  **(**e **=** tab**[**i**])** **!=** **null;**                   i **=** nextIndex**(**i**,** len**))** **{**                  ThreadLocal k **=** e**.**get**();**                    // If we find key, then we need to swap it                  // with the stale entry to maintain hash table order.                  // The newly stale slot, or any other stale slot                  // encountered above it, can then be sent to expungeStaleEntry                  // to remove or rehash all of the other entries in run.  **if** **(**k **==** key**)** **{**                      e**.**value **=** value**;**                        tab**[**i**]** **=** tab**[**staleSlot**];**                      tab**[**staleSlot**]** **=** e**;**                        // Start expunge at preceding stale entry if it exists  **if** **(**slotToExpunge **==** staleSlot**)**                          slotToExpunge **=** i**;**                      cleanSomeSlots**(**expungeStaleEntry**(**slotToExpunge**),** len**);**  **return;**  **}**                    // If we didn't find stale entry on backward scan, the                  // first stale entry seen while scanning for key is the                  // first still present in the run.  **if** **(**k **==** **null** **&&** slotToExpunge **==** staleSlot**)**                      slotToExpunge **=** i**;**  **}**                // If key not found, put new entry in stale slot              tab**[**staleSlot**].**value **=** **null;**              tab**[**staleSlot**]** **=** **new** Entry**(**key**,** value**);**                // If there are any other stale entries in run, expunge them  **if** **(**slotToExpunge **!=** staleSlot**)**                  cleanSomeSlots**(**expungeStaleEntry**(**slotToExpunge**),** len**);**  **}**              private int expungeStaleEntry**(**int staleSlot**)** **{**              Entry**[]** tab **=** table**;**              int len **=** tab**.**length**;**                // expunge entry at staleSlot              tab**[**staleSlot**].**value **=** **null;**              tab**[**staleSlot**]** **=** **null;**              size**--;**                // Rehash until we encounter null              Entry e**;**              int i**;**  **for** **(**i **=** nextIndex**(**staleSlot**,** len**);**  **(**e **=** tab**[**i**])** **!=** **null;**                   i **=** nextIndex**(**i**,** len**))** **{**                  ThreadLocal k **=** e**.**get**();**  **if** **(**k **==** **null)** **{**                      e**.**value **=** **null;**                      tab**[**i**]** **=** **null;**                      size**--;**  **}** **else** **{**                      int h **=** k**.**threadLocalHashCode **&** **(**len **-** 1**);**  **if** **(**h **!=** i**)** **{**                          tab**[**i**]** **=** **null;**                            // Unlike Knuth 6.4 Algorithm R, we must scan until                          // null because multiple entries could have been stale.  **while** **(**tab**[**h**]** **!=** **null)**                              h **=** nextIndex**(**h**,** len**);**                          tab**[**h**]** **=** e**;**  **}**  **}**  **}**  **return** i**;**  **}**            private boolean cleanSomeSlots**(**int i**,** int n**)** **{**              boolean removed **=** **false;**              Entry**[]** tab **=** table**;**              int len **=** tab**.**length**;**  **do** **{**                  i **=** nextIndex**(**i**,** len**);**                  Entry e **=** tab**[**i**];**  **if** **(**e **!=** **null** **&&** e**.**get**()** **==** **null)** **{**                      n **=** len**;**                      removed **=** **true;**                      i **=** expungeStaleEntry**(**i**);**  **}**  **}** **while** **(** **(**n **>>>=** 1**)** **!=** 0**);**  **return** removed**;**  **}**            private void rehash**()** **{**              expungeStaleEntries**();**                // Use lower threshold for doubling to avoid hysteresis  **if** **(**size **>=** threshold **-** threshold **/** 4**)**                  resize**();**  **}**            private void resize**()** **{**              Entry**[]** oldTab **=** table**;**              int oldLen **=** oldTab**.**length**;**              int newLen **=** oldLen **\*** 2**;**              Entry**[]** newTab **=** **new** Entry**[**newLen**];**              int count **=** 0**;**    **for** **(**int j **=** 0**;** j **<** oldLen**;** **++**j**)** **{**                  Entry e **=** oldTab**[**j**];**  **if** **(**e **!=** **null)** **{**                      ThreadLocal k **=** e**.**get**();**  **if** **(**k **==** **null)** **{**                          e**.**value **=** **null;** // Help the GC  **}** **else** **{**                          int h **=** k**.**threadLocalHashCode **&** **(**newLen **-** 1**);**  **while** **(**newTab**[**h**]** **!=** **null)**                              h **=** nextIndex**(**h**,** newLen**);**                          newTab**[**h**]** **=** e**;**                          count**++;**  **}**  **}**  **}**                setThreshold**(**newLen**);**              size **=** count**;**              table **=** newTab**;**  **}**            private void expungeStaleEntries**()** **{**              Entry**[]** tab **=** table**;**              int len **=** tab**.**length**;**  **for** **(**int j **=** 0**;** j **<** len**;** j**++)** **{**                  Entry e **=** tab**[**j**];**  **if** **(**e **!=** **null** **&&** e**.**get**()** **==** **null)**                      expungeStaleEntry**(**j**);**  **}**  **}**  **}**  **}** |

二、测试代码

（一）源码

|  |
| --- |
|  |

Why ThreadLocal?

无论如何，要编写一个多线程安全(Thread-safe)的程序是困难的，为了让线程共享资源，必须小心地对共享资源进行同步，同步带来一定的效能延迟，而另一方面，在处理同步的时候，又要注意对象的锁定与释放，避免产生死结，种种因素都使得编写多线程程序变得困难。

尝试从另一个角度来思考多线程共享资源的问题，既然共享资源这么困难，那么就干脆不要共享，何不为每个线程创造一个资源的复本。将每一个线程存取数据的行为加以隔离，实现的方法就是给予每个线程一个特定空间来保管该线程所独享的资源

什么是ThreadLocal？

顾名思义它是local variable（线程局部变量）。它的功用非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，是每一个线程都可以独立地改变自己的副本，而不会和其它线程的副本冲突。从线程的角度看，就好像每一个线程都完全拥有该变量。

首先，本文先谈一下对ThreadLocal的理解，然后根据ThreadLocal类的源码分析了其实现原理和使用需要注意的地方，最后给出了两个应用场景。

　　以下是本文目录大纲：

　　一.对ThreadLocal的理解

　　二.深入解析ThreadLocal类

　　三.ThreadLocal的应用场景：

<http://www.cnblogs.com/dolphin0520/p/3920407.html>

**一.对ThreadLocal的理解**

　　ThreadLocal，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，其实意思差不多。可能很多朋友都知道ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

　　这句话从字面上看起来很容易理解，但是真正理解并不是那么容易。

　　我们还是先来看一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class ConnectionManager {        private static Connection connect = null;        public static Connection openConnection() {          if(connect == null){              connect = DriverManager.getConnection();          }          return connect;      }        public static void closeConnection() {          if(connect!=null)              connect.close();      }  } |

 　　假设有这样一个数据库链接管理类，这段代码在单线程中使用是没有任何问题的，但是如果在多线程中使用呢？很显然，在多线程中使用会存在线程安全问题：第一，这里面的2个方法都没有进行同步，很可能在openConnection方法中会多次创建connect；第二，由于connect是共享变量，那么必然在调用connect的地方需要使用到同步来保障线程安全，因为很可能一个线程在使用connect进行数据库操作，而另外一个线程调用closeConnection关闭链接。

　　所以出于线程安全的考虑，必须将这段代码的两个方法进行同步处理，并且在调用connect的地方需要进行同步处理。

　　这样将会大大影响程序执行效率，因为一个线程在使用connect进行数据库操作的时候，其他线程只有等待。

　　那么大家来仔细分析一下这个问题，这地方到底需不需要将connect变量进行共享？事实上，是不需要的。假如每个线程中都有一个connect变量，各个线程之间对connect变量的访问实际上是没有依赖关系的，即一个线程不需要关心其他线程是否对这个connect进行了修改的。

　　到这里，可能会有朋友想到，既然不需要在线程之间共享这个变量，可以直接这样处理，在每个需要使用数据库连接的方法中具体使用时才创建数据库链接，然后在方法调用完毕再释放这个连接。比如下面这样：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | class ConnectionManager {        private  Connection connect = null;        public Connection openConnection() {          if(connect == null){              connect = DriverManager.getConnection();          }          return connect;      }        public void closeConnection() {          if(connect!=null)              connect.close();      }  }      class Dao{      public void insert() {          ConnectionManager connectionManager = new ConnectionManager();          Connection connection = connectionManager.openConnection();            //使用connection进行操作            connectionManager.closeConnection();      }  } |

 　　这样处理确实也没有任何问题，由于每次都是在方法内部创建的连接，那么线程之间自然不存在线程安全问题。但是这样会有一个致命的影响：导致服务器压力非常大，并且严重影响程序执行性能。由于在方法中需要频繁地开启和关闭数据库连接，这样不尽严重影响程序执行效率，还可能导致服务器压力巨大。

　　那么这种情况下使用ThreadLocal是再适合不过的了，因为ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

　　但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以要考虑它对资源的消耗，比如内存的占用会比不使用ThreadLocal要大。

**二.深入解析ThreadLocal类**

　　在上面谈到了对ThreadLocal的一些理解，那我们下面来看一下具体ThreadLocal是如何实现的。

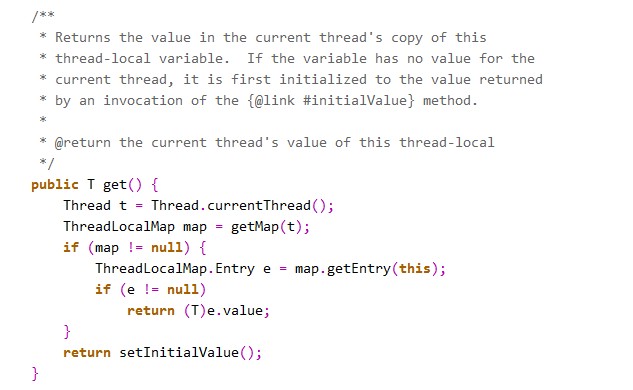
　　先了解一下ThreadLocal类提供的几个方法：

|  |  |
| --- | --- |
| 1  2  3  4 | public T get() { }  public void set(T value) { }  public void remove() { }  protected T initialValue() { } |

 　　get()方法是用来获取ThreadLocal在当前线程中保存的变量副本，set()用来设置当前线程中变量的副本，remove()用来移除当前线程中变量的副本，initialValue()是一个protected方法，一般是用来在使用时进行重写的，它是一个延迟加载方法，下面会详细说明。

　　首先我们来看一下ThreadLocal类是如何为每个线程创建一个变量的副本的。

　　先看下get方法的实现：



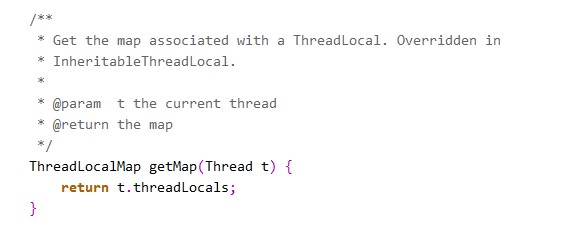
 　　第一句是取得当前线程，然后通过getMap(t)方法获取到一个map，map的类型为ThreadLocalMap。然后接着下面获取到<key,value>键值对，注意这里获取键值对传进去的是  this，而不是当前线程t。

　　如果获取成功，则返回value值。

　　如果map为空，则调用setInitialValue方法返回value。

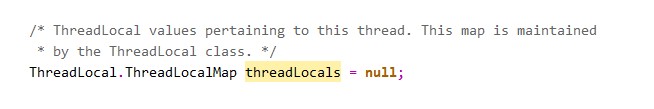
　　我们上面的每一句来仔细分析：

　　首先看一下getMap方法中做了什么：

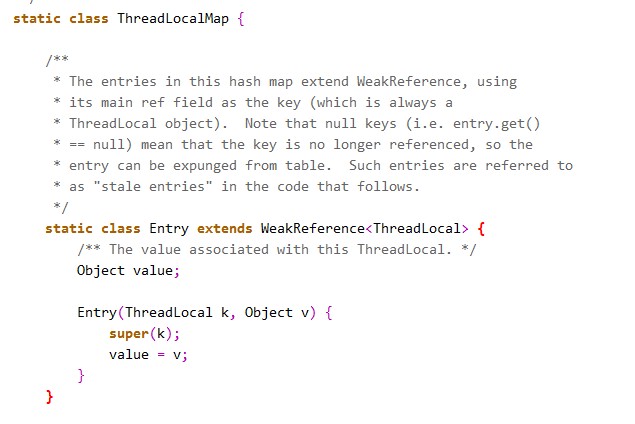


　　可能大家没有想到的是，在getMap中，是调用当期线程t，返回当前线程t中的一个成员变量threadLocals。

　　那么我们继续取Thread类中取看一下成员变量threadLocals是什么：



　　实际上就是一个ThreadLocalMap，这个类型是ThreadLocal类的一个内部类，我们继续取看ThreadLocalMap的实现：

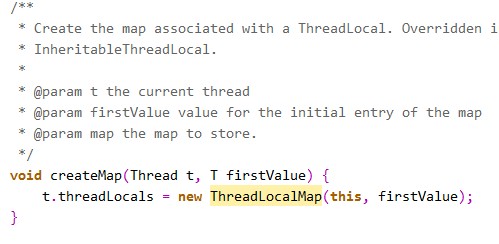


　　可以看到ThreadLocalMap的Entry继承了WeakReference，并且使用ThreadLocal作为键值。

　　然后再继续看setInitialValue方法的具体实现：



　　很容易了解，就是如果map不为空，就设置键值对，为空，再创建Map，看一下createMap的实现：



　　至此，可能大部分朋友已经明白了ThreadLocal是如何为每个线程创建变量的副本的：

　　首先，在每个线程Thread内部有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，这个threadLocals就是用来存储实际的变量副本的，键值为当前ThreadLocal变量，value为变量副本（即T类型的变量）。

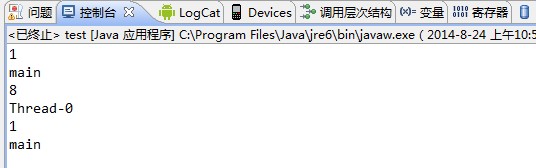
　　初始时，在Thread里面，threadLocals为空，当通过ThreadLocal变量调用get()方法或者set()方法，就会对Thread类中的threadLocals进行初始化，并且以当前ThreadLocal变量为键值，以ThreadLocal要保存的副本变量为value，存到threadLocals。

　　然后在当前线程里面，如果要使用副本变量，就可以通过get方法在threadLocals里面查找。

　　下面通过一个例子来证明通过ThreadLocal能达到在每个线程中创建变量副本的效果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | public class Test {      ThreadLocal<Long> longLocal = new ThreadLocal<Long>();      ThreadLocal<String> stringLocal = new ThreadLocal<String>();          public void set() {          longLocal.set(Thread.currentThread().getId());          stringLocal.set(Thread.currentThread().getName());      }        public long getLong() {          return longLocal.get();      }        public String getString() {          return stringLocal.get();      }        public static void main(String[] args) throws InterruptedException {          final Test test = new Test();              test.set();          System.out.println(test.getLong());          System.out.println(test.getString());              Thread thread1 = new Thread(){              public void run() {                  test.set();                  System.out.println(test.getLong());                  System.out.println(test.getString());              };          };          thread1.start();          thread1.join();            System.out.println(test.getLong());          System.out.println(test.getString());      }  } |

 　　这段代码的输出结果为：



　　从这段代码的输出结果可以看出，在main线程中和thread1线程中，longLocal保存的副本值和stringLocal保存的副本值都不一样。最后一次在main线程再次打印副本值是为了证明在main线程中和thread1线程中的副本值确实是不同的。

　　总结一下：

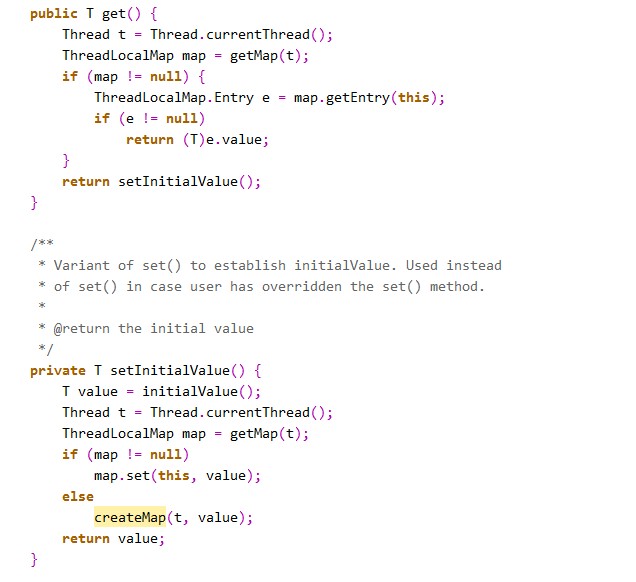
　　1）实际的通过ThreadLocal创建的副本是存储在每个线程自己的threadLocals中的；

　　2）为何threadLocals的类型ThreadLocalMap的键值为ThreadLocal对象，因为每个线程中可有多个threadLocal变量，就像上面代码中的longLocal和stringLocal；

　　3）在进行get之前，必须先set，否则会报空指针异常；

　　    如果想在get之前不需要调用set就能正常访问的话，必须重写initialValue()方法。

　　　 因为在上面的代码分析过程中，我们发现如果没有先set的话，即在map中查找不到对应的存储，则会通过调用setInitialValue方法返回i，而在setInitialValue方法中，有一个语句是T value = initialValue()， 而默认情况下，initialValue方法返回的是null。



　　看下面这个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class Test {      ThreadLocal<Long> longLocal = new ThreadLocal<Long>();      ThreadLocal<String> stringLocal = new ThreadLocal<String>();        public void set() {          longLocal.set(Thread.currentThread().getId());          stringLocal.set(Thread.currentThread().getName());      }        public long getLong() {          return longLocal.get();      }        public String getString() {          return stringLocal.get();      }        public static void main(String[] args) throws InterruptedException {          final Test test = new Test();            System.out.println(test.getLong());          System.out.println(test.getString());            Thread thread1 = new Thread(){              public void run() {                  test.set();                  System.out.println(test.getLong());                  System.out.println(test.getString());              };          };          thread1.start();          thread1.join();            System.out.println(test.getLong());          System.out.println(test.getString());      }  } |

 　　在main线程中，没有先set，直接get的话，运行时会报空指针异常。

　　但是如果改成下面这段代码，即重写了initialValue方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | public class Test {      ThreadLocal<Long> longLocal = new ThreadLocal<Long>(){          protected Long initialValue() {              return Thread.currentThread().getId();          };      };      ThreadLocal<String> stringLocal = new ThreadLocal<String>(){;          protected String initialValue() {              return Thread.currentThread().getName();          };      };          public void set() {          longLocal.set(Thread.currentThread().getId());          stringLocal.set(Thread.currentThread().getName());      }        public long getLong() {          return longLocal.get();      }        public String getString() {          return stringLocal.get();      }        public static void main(String[] args) throws InterruptedException {          final Test test = new Test();            test.set();          System.out.println(test.getLong());          System.out.println(test.getString());              Thread thread1 = new Thread(){              public void run() {                  test.set();                  System.out.println(test.getLong());                  System.out.println(test.getString());              };          };          thread1.start();          thread1.join();            System.out.println(test.getLong());          System.out.println(test.getString());      }  } |

 　　就可以直接不用先set而直接调用get了。

**三.ThreadLocal的应用场景**

　　最常见的ThreadLocal使用场景为 用来解决 数据库连接、Session管理等。

　　如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | private static ThreadLocal<Connection> connectionHolder  = new ThreadLocal<Connection>() {  public Connection initialValue() {      return DriverManager.getConnection(DB\_URL);  }  };    public static Connection getConnection() {  return connectionHolder.get();  } |

 　　下面这段代码摘自：

<http://www.iteye.com/topic/103804>

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | private static final ThreadLocal threadSession = new ThreadLocal();    public static Session getSession() throws InfrastructureException {      Session s = (Session) threadSession.get();      try {          if (s == null) {              s = getSessionFactory().openSession();              threadSession.set(s);          }      } catch (HibernateException ex) {          throw new InfrastructureException(ex);      }      return s;  } |

参考：<http://www.cnblogs.com/dolphin0520/p/3920407.html>

# REF