**ThreadLocal 不是用来解决共享对象的多线程访问问题的，一般情况下，通过ThreadLocal.set() 到线程中的对象是该线程自己使用的对象，其他线程是不需要访问的，也访问不到的。**各个线程中访问的是不同的对象。

**另外，说ThreadLocal使得各线程能够保持各自独立的一个对象，并不是通过ThreadLocal.set()来实现的，而是通过每个线程中的new 对象 的操作来创建的对象，每个线程创建一个，不是什么对象的拷贝或副本。**通过ThreadLocal.set()将这个新创建的对象的引用保存到各线程的自己的一个map中，每个线程都有这样一个map，执行 ThreadLocal.get()时，各线程从自己的map中取出放进去的对象，因此取出来的是各自自己线程中的对象，ThreadLocal实例是作为map的key来使用的。

如果ThreadLocal.set()进去的东西本来就是多个线程共享的同一个对象，那么多个线程的ThreadLocal.get()取得的还是这个共享对象本身，还是有并发访问问题。

下面来看一个hibernate中典型的ThreadLocal的应用：

代码

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

可以看到在getSession()方法中，首先判断当前线程中有没有放进去session，如果还没有，那么通过sessionFactory ().openSession()来创建一个session，再将session set到线程中，实际是放到当前线程的ThreadLocalMap这个map中，这时，对于这个session的唯一引用就是当前线程中的那个 ThreadLocalMap（下面会讲到），而threadSession作为这个值的key，要取得这个session可以通过 threadSession.get()来得到，里面执行的操作实际是先取得当前线程中的ThreadLocalMap，然后将 threadSession作为key将对应的值取出。这个session相当于线程的私有变量，而不是public的。   
显然，其他线程中是取不到这个session的，他们也只能取到自己的ThreadLocalMap中的东西。要是session是多个线程共享使用的，那还不乱套了。   
试想如果不用ThreadLocal怎么来实现呢？可能就要在action中创建session，然后把session一个个传到service 和dao中，这可够麻烦的。或者可以自己定义一个静态的map，将当前thread作为key，创建的session作为值，put到map中，应该也行，这也是一般人的想法，但事实上，ThreadLocal的实现刚好相反，它是在每个线程中有一个map，而将ThreadLocal实例作为key，这样每个map中的项数很少，而且当线程销毁时相应的东西也一起销毁了，不知道除了这些还有什么其他的好处。

总之，ThreadLocal不是用来解决对象共享访问问题的，而主要是提供了保持对象的方法和避免参数传递的方便的对象访问方式。归纳了两点：   
1。每个线程中都有一个自己的ThreadLocalMap类对象，可以将线程自己的对象保持到其中，各管各的，线程可以正确的访问到自己的对象。   
2。将一个共用的ThreadLocal静态实例作为key，将不同对象的引用保存到不同线程的ThreadLocalMap中，然后在线程执行的各处通过这个静态ThreadLocal实例的get()方法取得自己线程保存的那个对象，避免了将这个对象作为参数传递的麻烦。

当然如果要把本来线程共享的对象通过ThreadLocal.set()放到线程中也可以，可以实现避免参数传递的访问方式，但是要注意get() 到的是那同一个共享对象，并发访问问题要靠其他手段来解决。但一般来说线程共享的对象通过设置为某类的静态变量就可以实现方便的访问了，似乎没必要放到线程中。

ThreadLocal的应用场合，我觉得最适合的是按线程多实例（每个线程对应一个实例）的对象的访问，并且这个对象很多地方都要用到。

下面来看看ThreadLocal的实现原理（jdk1.5源码）

代码

1. **public** **class** ThreadLocal<T> {
2. /\*\*
3. \* ThreadLocals rely on per-thread hash maps attached to each thread
4. \* (Thread.threadLocals and inheritableThreadLocals).  The ThreadLocal
5. \* objects act as keys, searched via threadLocalHashCode.  This is a
6. \* custom hash code (useful only within ThreadLocalMaps) that eliminates
7. \* collisions in the common case where consecutively constructed
8. \* ThreadLocals are used by the same threads, while remaining well-behaved
9. \* in less common cases.
10. \*/
11. **private** **final** **int** threadLocalHashCode = nextHashCode();
13. /\*\*
14. \* The next hash code to be given out. Accessed only by like-named method.
15. \*/
16. **private** **static** **int** nextHashCode = 0;
18. /\*\*
19. \* The difference between successively generated hash codes - turns
20. \* implicit sequential thread-local IDs into near-optimally spread
21. \* multiplicative hash values for power-of-two-sized tables.
22. \*/
23. **private** **static** **final** **int** HASH\_INCREMENT = 0x61c88647;
25. /\*\*
26. \* Compute the next hash code. The static synchronization used here
27. \* should not be a performance bottleneck. When ThreadLocals are
28. \* generated in different threads at a fast enough rate to regularly
29. \* contend on this lock, memory contention is by far a more serious
30. \* problem than lock contention.
31. \*/
32. **private** **static** **synchronized** **int** nextHashCode() {
33. **int** h = nextHashCode;
34. nextHashCode = h + HASH\_INCREMENT;
35. **return** h;
36. }
38. /\*\*
39. \* Creates a thread local variable.
40. \*/
41. **public** ThreadLocal() {
42. }
44. /\*\*
45. \* Returns the value in the current thread's copy of this thread-local
46. \* variable.  Creates and initializes the copy if this is the first time
47. \* the thread has called this method.
48. \*
49. \* @return the current thread's value of this thread-local
50. \*/
51. **public** T get() {
52. Thread t = Thread.currentThread();
53. ThreadLocalMap map = getMap(t);
54. **if** (map != **null**)
55. **return** (T)map.get(**this**);
57. // Maps are constructed lazily.  if the map for this thread
58. // doesn't exist, create it, with this ThreadLocal and its
59. // initial value as its only entry.
60. T value = initialValue();
61. createMap(t, value);
62. **return** value;
63. }
65. /\*\*
66. \* Sets the current thread's copy of this thread-local variable
67. \* to the specified value.  Many applications will have no need for
68. \* this functionality, relying solely on the {@link #initialValue}
69. \* method to set the values of thread-locals.
70. \*
71. \* @param value the value to be stored in the current threads' copy of
72. \*        this thread-local.
73. \*/
74. **public** **void** set(T value) {
75. Thread t = Thread.currentThread();
76. ThreadLocalMap map = getMap(t);
77. **if** (map != **null**)
78. map.set(**this**, value);
79. **else**
80. createMap(t, value);
81. }
83. /\*\*
84. \* Get the map associated with a ThreadLocal. Overridden in
85. \* InheritableThreadLocal.
86. \*
87. \* @param  t the current thread
88. \* @return the map
89. \*/
90. ThreadLocalMap getMap(Thread t) {
91. **return** t.threadLocals;
92. }
94. /\*\*
95. \* Create the map associated with a ThreadLocal. Overridden in
96. \* InheritableThreadLocal.
97. \*
98. \* @param t the current thread
99. \* @param firstValue value for the initial entry of the map
100. \* @param map the map to store.
101. \*/
102. **void** createMap(Thread t, T firstValue) {
103. t.threadLocals = **new** ThreadLocalMap(**this**, firstValue);
104. }
106. .......
108. /\*\*
109. \* ThreadLocalMap is a customized hash map suitable only for
110. \* maintaining thread local values. No operations are exported
111. \* outside of the ThreadLocal class. The class is package private to
112. \* allow declaration of fields in class Thread.  To help deal with
113. \* very large and long-lived usages, the hash table entries use
114. \* WeakReferences for keys. However, since reference queues are not
115. \* used, stale entries are guaranteed to be removed only when
116. \* the table starts running out of space.
117. \*/
118. **static** **class** ThreadLocalMap {
120. ........
122. }
124. }

可以看到ThreadLocal类中的变量只有这3个int型：

代码

1. **private** **final** **int** threadLocalHashCode = nextHashCode();
2. **private** **static** **int** nextHashCode = 0;
3. **private** **static** **final** **int** HASH\_INCREMENT = 0x61c88647;

而作为ThreadLocal实例的变量只有 threadLocalHashCode 这一个，nextHashCode 和HASH\_INCREMENT 是ThreadLocal类的静态变量，实际上HASH\_INCREMENT是一个常量，表示了连续分配的两个ThreadLocal实例的 threadLocalHashCode值的增量，而nextHashCode 的表示了即将分配的下一个ThreadLocal实例的threadLocalHashCode 的值。

可以来看一下创建一个ThreadLocal实例即new ThreadLocal()时做了哪些操作，从上面看到构造函数ThreadLocal()里什么操作都没有，唯一的操作是这句：

代码

1. **private** **final** **int** threadLocalHashCode = nextHashCode();

那么nextHashCode()做了什么呢：

代码

1. **private** **static** **synchronized** **int** nextHashCode() {
2. **int** h = nextHashCode;
3. nextHashCode = h + HASH\_INCREMENT;
4. **return** h;
5. }

就是将ThreadLocal类的下一个hashCode值即nextHashCode的值赋给实例的threadLocalHashCode，然后nextHashCode的值增加HASH\_INCREMENT这个值。

因此ThreadLocal实例的变量只有这个threadLocalHashCode，而且是final的，用来区分不同的 ThreadLocal实例，ThreadLocal类主要是作为工具类来使用，那么ThreadLocal.set()进去的对象是放在哪儿的呢？

看一下上面的set()方法，两句合并一下成为

代码

1. ThreadLocalMap map = Thread.currentThread().threadLocals;

这个ThreadLocalMap 类是ThreadLocal中定义的内部类，但是它的实例却用在Thread类中：

代码

1. **public** **class** Thread **implements** Runnable {
2. ......
4. /\* ThreadLocal values pertaining to this thread. This map is maintained
5. \* by the ThreadLocal class. \*/
6. ThreadLocal.ThreadLocalMap threadLocals = **null**;
7. ......
8. }

再看这句：

代码

1. **if** (map != **null**)
2. map.set(**this**, value);

也就是将该ThreadLocal实例作为key，要保持的对象作为值，设置到当前线程的ThreadLocalMap 中，get()方法同样大家看了代码也就明白了，ThreadLocalMap 类的代码太多了，我就不帖了，自己去看源码吧。

写了这么多，也不知讲明白了没有，有什么不当的地方还请大家指出来

**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。

　　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。

　　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：

**[java]** [view plain](http://blog.csdn.net/lufeng20/article/details/24314381) [copy](http://blog.csdn.net/lufeng20/article/details/24314381)

[print](http://blog.csdn.net/lufeng20/article/details/24314381)[?](http://blog.csdn.net/lufeng20/article/details/24314381)[在CODE上查看代码片](https://code.csdn.net/snippets/306406)

1. **package** com.test;
3. **public** **class** TestNum {
4. // ①通过匿名内部类覆盖ThreadLocal的initialValue()方法，指定初始值
5. **private** **static** ThreadLocal<Integer> seqNum = **new** ThreadLocal<Integer>() {
6. **public** Integer initialValue() {
7. **return** 0;
8. }
9. };
11. // ②获取下一个序列值
12. **public** **int** getNextNum() {
13. seqNum.set(seqNum.get() + 1);
14. **return** seqNum.get();
15. }
17. **public** **static** **void** main(String[] args) {
18. TestNum sn = **new** TestNum();
19. // ③ 3个线程共享sn，各自产生序列号
20. TestClient t1 = **new** TestClient(sn);
21. TestClient t2 = **new** TestClient(sn);
22. TestClient t3 = **new** TestClient(sn);
23. t1.start();
24. t2.start();
25. t3.start();
26. }
28. **private** **static** **class** TestClient **extends** Thread {
29. **private** TestNum sn;
31. **public** TestClient(TestNum sn) {
32. **this**.sn = sn;
33. }
35. **public** **void** run() {
36. **for** (**int** i = 0; i < 3; i++) {
37. // ④每个线程打出3个序列值
38. System.out.println("thread[" + Thread.currentThread().getName() + "] --> sn["
39. + sn.getNextNum() + "]");
40. }
41. }
42. }
43. }

 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TestNum实例。运行以上代码，在控制台上输出以下的结果：

thread[Thread-0] --> sn[1]  
thread[Thread-1] --> sn[1]  
thread[Thread-2] --> sn[1]  
thread[Thread-1] --> sn[2]  
thread[Thread-0] --> sn[2]  
thread[Thread-1] --> sn[3]  
thread[Thread-2] --> sn[2]  
thread[Thread-0] --> sn[3]  
thread[Thread-2] --> 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TestNum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。

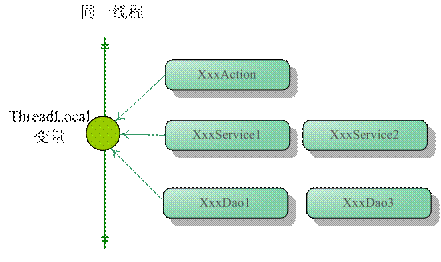
　　而ThreadLocal则从另一个角度来解决多线程的并发访问。ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

　　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：



　　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 TestDao：非线程安全

**[java]** [view plain](http://blog.csdn.net/lufeng20/article/details/24314381) [copy](http://blog.csdn.net/lufeng20/article/details/24314381)

[print](http://blog.csdn.net/lufeng20/article/details/24314381)[?](http://blog.csdn.net/lufeng20/article/details/24314381)[在CODE上查看代码片](https://code.csdn.net/snippets/306406)

1. **package** com.test;
3. **import** java.sql.Connection;
4. **import** java.sql.SQLException;
5. **import** java.sql.Statement;
7. **public** **class** TestDao {
8. **private** Connection conn;// ①一个非线程安全的变量
10. **public** **void** addTopic() **throws** SQLException {
11. Statement stat = conn.createStatement();// ②引用非线程安全变量
12. // …
13. }
14. }

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 TestDao：线程安全

**[java]** [view plain](http://blog.csdn.net/lufeng20/article/details/24314381) [copy](http://blog.csdn.net/lufeng20/article/details/24314381)

[print](http://blog.csdn.net/lufeng20/article/details/24314381)[?](http://blog.csdn.net/lufeng20/article/details/24314381)[在CODE上查看代码片](https://code.csdn.net/snippets/306406)

1. **package** com.test;
3. **import** java.sql.Connection;
4. **import** java.sql.SQLException;
5. **import** java.sql.Statement;
7. **public** **class** TestDaoNew {
8. // ①使用ThreadLocal保存Connection变量
9. **private** **static** ThreadLocal<Connection> connThreadLocal = **new** ThreadLocal<Connection>();
11. **public** **static** Connection getConnection() {
12. // ②如果connThreadLocal没有本线程对应的Connection创建一个新的Connection，
13. // 并将其保存到线程本地变量中。
14. **if** (connThreadLocal.get() == **null**) {
15. Connection conn = getConnection();
16. connThreadLocal.set(conn);
17. **return** conn;
18. } **else** {
19. **return** connThreadLocal.get();// ③直接返回线程本地变量
20. }
21. }
23. **public** **void** addTopic() **throws** SQLException {
24. // ④从ThreadLocal中获取线程对应的Connection
25. Statement stat = getConnection().createStatement();
26. }
27. }

　　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。

ConnectionManager.java

**[java]** [view plain](http://blog.csdn.net/lufeng20/article/details/24314381) [copy](http://blog.csdn.net/lufeng20/article/details/24314381)

[print](http://blog.csdn.net/lufeng20/article/details/24314381)[?](http://blog.csdn.net/lufeng20/article/details/24314381)[在CODE上查看代码片](https://code.csdn.net/snippets/306406)

1. **package** com.test;
3. **import** java.sql.Connection;
4. **import** java.sql.DriverManager;
5. **import** java.sql.SQLException;
7. **public** **class** ConnectionManager {
9. **private** **static** ThreadLocal<Connection> connectionHolder = **new** ThreadLocal<Connection>() {
10. @Override
11. **protected** Connection initialValue() {
12. Connection conn = **null**;
13. **try** {
14. conn = DriverManager.getConnection(
15. "jdbc:mysql://localhost:3306/test", "username",
16. "password");
17. } **catch** (SQLException e) {
18. e.printStackTrace();
19. }
20. **return** conn;
21. }
22. };
24. **public** **static** Connection getConnection() {
25. **return** connectionHolder.get();
26. }
28. **public** **static** **void** setConnection(Connection conn) {
29. connectionHolder.set(conn);
30. }
31. }

**java.lang.ThreadLocal<T>的具体实现**

那么到底ThreadLocal类是如何实现这种“为每个线程提供不同的变量拷贝”的呢？先来看一下ThreadLocal的set()方法的源码是如何实现的：

**[java]** [view plain](http://blog.csdn.net/lufeng20/article/details/24314381) [copy](http://blog.csdn.net/lufeng20/article/details/24314381)

[print](http://blog.csdn.net/lufeng20/article/details/24314381)[?](http://blog.csdn.net/lufeng20/article/details/24314381)[在CODE上查看代码片](https://code.csdn.net/snippets/306406)

1. /\*\*
2. \* Sets the current thread's copy of this thread-local variable
3. \* to the specified value.  Most subclasses will have no need to
4. \* override this method, relying solely on the {@link #initialValue}
5. \* method to set the values of thread-locals.
6. \*
7. \* @param value the value to be stored in the current thread's copy of
8. \*        this thread-local.
9. \*/
10. **public** **void** set(T value) {
11. Thread t = Thread.currentThread();
12. ThreadLocalMap map = getMap(t);
13. **if** (map != **null**)
14. map.set(**this**, value);
15. **else**
16. createMap(t, value);
17. }

在这个方法内部我们看到，首先通过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。  
  
  
为了加深理解，我们接着看上面代码中出现的getMap和createMap方法的实现：

**[java]** [view plain](http://blog.csdn.net/lufeng20/article/details/24314381) [copy](http://blog.csdn.net/lufeng20/article/details/24314381)

[print](http://blog.csdn.net/lufeng20/article/details/24314381)[?](http://blog.csdn.net/lufeng20/article/details/24314381)[在CODE上查看代码片](https://code.csdn.net/snippets/306406)

1. /\*\*
2. \* Get the map associated with a ThreadLocal. Overridden in
3. \* InheritableThreadLocal.
4. \*
5. \* @param  t the current thread
6. \* @return the map
7. \*/
8. ThreadLocalMap getMap(Thread t) {
9. **return** t.threadLocals;
10. }
12. /\*\*
13. \* Create the map associated with a ThreadLocal. Overridden in
14. \* InheritableThreadLocal.
15. \*
16. \* @param t the current thread
17. \* @param firstValue value for the initial entry of the map
18. \* @param map the map to store.
19. \*/
20. **void** createMap(Thread t, T firstValue) {
21. t.threadLocals = **new** ThreadLocalMap(**this**, firstValue);
22. }

接下来再看一下ThreadLocal类中的get()方法:

**[java]** [view plain](http://blog.csdn.net/lufeng20/article/details/24314381) [copy](http://blog.csdn.net/lufeng20/article/details/24314381)

[print](http://blog.csdn.net/lufeng20/article/details/24314381)[?](http://blog.csdn.net/lufeng20/article/details/24314381)[在CODE上查看代码片](https://code.csdn.net/snippets/306406)

1. /\*\*
2. \* Returns the value in the current thread's copy of this
3. \* thread-local variable.  If the variable has no value for the
4. \* current thread, it is first initialized to the value returned
5. \* by an invocation of the {@link #initialValue} method.
6. \*
7. \* @return the current thread's value of this thread-local
8. \*/
9. **public** T get() {
10. Thread t = Thread.currentThread();
11. ThreadLocalMap map = getMap(t);
12. **if** (map != **null**) {
13. ThreadLocalMap.Entry e = map.getEntry(**this**);
14. **if** (e != **null**)
15. **return** (T)e.value;
16. }
17. **return** setInitialValue();
18. }

再来看setInitialValue()方法：

**[java]** [view plain](http://blog.csdn.net/lufeng20/article/details/24314381) [copy](http://blog.csdn.net/lufeng20/article/details/24314381)

[print](http://blog.csdn.net/lufeng20/article/details/24314381)[?](http://blog.csdn.net/lufeng20/article/details/24314381)[在CODE上查看代码片](https://code.csdn.net/snippets/306406)

1. /\*\*
2. \* Variant of set() to establish initialValue. Used instead
3. \* of set() in case user has overridden the set() method.
4. \*
5. \* @return the initial value
6. \*/
7. **private** T setInitialValue() {
8. T value = initialValue();
9. Thread t = Thread.currentThread();
10. ThreadLocalMap map = getMap(t);
11. **if** (map != **null**)
12. map.set(**this**, value);
13. **else**
14. createMap(t, value);
15. **return** value;
16. }

　　获取和当前线程绑定的值时，ThreadLocalMap对象是以this指向的ThreadLocal对象为键进行查找的，这当然和前面set()方法的代码是相呼应的。  
  
  
　　进一步地，我们可以创建不同的ThreadLocal实例来实现多个变量在不同线程间的访问隔离，为什么可以这么做？因为不同的ThreadLocal对象作为不同键，当然也可以在线程的ThreadLocalMap对象中设置不同的值了。通过ThreadLocal对象，在多线程中共享一个值和多个值的区别，就像你在一个HashMap对象中存储一个键值对和多个键值对一样，仅此而已。

**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。

ConnectionManager.java

**[java]** [view plain](http://blog.csdn.net/lufeng20/article/details/24314381) [copy](http://blog.csdn.net/lufeng20/article/details/24314381)

[print](http://blog.csdn.net/lufeng20/article/details/24314381)[?](http://blog.csdn.net/lufeng20/article/details/24314381)[在CODE上查看代码片](https://code.csdn.net/snippets/306406)

1. **package** com.test;
3. **import** java.sql.Connection;
4. **import** java.sql.DriverManager;
5. **import** java.sql.SQLException;
7. **public** **class** ConnectionManager {
9. **private** **static** ThreadLocal<Connection> connectionHolder = **new** ThreadLocal<Connection>() {
10. @Override
11. **protected** Connection initialValue() {
12. Connection conn = **null**;
13. **try** {
14. conn = DriverManager.getConnection(
15. "jdbc:mysql://localhost:3306/test", "username",
16. "password");
17. } **catch** (SQLException e) {
18. e.printStackTrace();
19. }
20. **return** conn;
21. }
22. };
24. **public** **static** Connection getConnection() {
25. **return** connectionHolder.get();
26. }
28. **public** **static** **void** setConnection(Connection conn) {
29. connectionHolder.set(conn);
30. }
31. }

**后记**

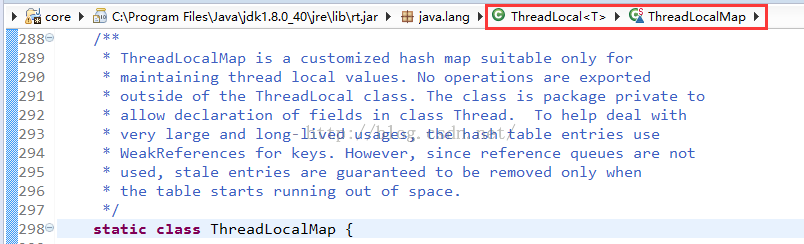
　　看到网友评论的很激烈，甚至关于ThreadLocalMap不是ThreadLocal里面的，而是Thread里面的这种评论都出现了，于是有了这个后记，下面先把jdk源码贴上，源码最有说服力了。

**[java]** [view plain](http://blog.csdn.net/lufeng20/article/details/24314381) [copy](http://blog.csdn.net/lufeng20/article/details/24314381)

[print](http://blog.csdn.net/lufeng20/article/details/24314381)[?](http://blog.csdn.net/lufeng20/article/details/24314381)[在CODE上查看代码片](https://code.csdn.net/snippets/306406)

1. /\*\*
2. \* ThreadLocalMap is a customized hash map suitable only for
3. \* maintaining thread local values. No operations are exported
4. \* outside of the ThreadLocal class. The class is package private to
5. \* allow declaration of fields in class Thread.  To help deal with
6. \* very large and long-lived usages, the hash table entries use
7. \* WeakReferences for keys. However, since reference queues are not
8. \* used, stale entries are guaranteed to be removed only when
9. \* the table starts running out of space.
10. \*/
11. **static** **class** ThreadLocalMap {...}

　　源码就是以上，这源码自然是在ThreadLocal里面的，有截图为证。



　　本文是自己在学习ThreadLocal的时候，一时兴起，深入看了源码，思考了此类的作用、使用范围，进而联想到对传统的synchronize共享变量线程安全的问题进行比较，而总结的博文，总结一句话就是一个是锁机制进行时间换空间，一个是存储拷贝进行空间换时间。

(全文完)