概述

相信读者在网上也看了很多关于ThreadLocal的资料，很多博客都这样说：ThreadLocal为解决多线程程序的并发问题提供了一种新的思路；ThreadLocal的目的是为了解决多线程访问资源时的共享问题。如果你也这样认为的，那现在给你10秒钟，清空之前对ThreadLocal的**错误**的认知！

看看JDK中的源码是怎么写的：

> This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its {@code get} or {@code set} method) has its own, independently initialized copy of the variable. {@code ThreadLocal} instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

翻译过来大概是这样的(英文不好，如有更好的翻译，请留言说明)：

> ThreadLocal类用来提供线程内部的局部变量。这种变量在多线程环境下访问(通过get或set方法访问)时能保证各个线程里的变量相对独立于其他线程内的变量。ThreadLocal实例通常来说都是private static类型的，用于关联线程和线程的上下文。

可以总结为一句话：**ThreadLocal的作用是提供线程内的局部变量，这种变量在线程的生命周期内起作用，减少同一个线程内多个函数或者组件之间一些公共变量的传递的复杂度。**
举个例子，我出门需要先坐公交再做地铁，这里的坐公交和坐地铁就好比是同一个线程内的两个函数，我就是一个线程，我要完成这两个函数都需要同一个东西：公交卡（北京公交和地铁都使用公交卡），那么我为了不向这两个函数都传递公交卡这个变量（相当于不是一直带着公交卡上路），我可以这么做：将公交卡事先交给一个机构，当我需要刷卡的时候再向这个机构要公交卡（当然每次拿的都是同一张公交卡）。这样就能达到只要是我(同一个线程)需要公交卡，何时何地都能向这个机构要的目的。

有人要说了：你可以将公交卡设置为全局变量啊，这样不是也能何时何地都能取公交卡吗？但是如果有很多个人（很多个线程）呢？大家可不能都使用同一张公交卡吧(我们假设公交卡是实名认证的)，这样不就乱套了嘛。现在明白了吧？这就是ThreadLocal设计的初衷：提供线程内部的局部变量，在本线程内随时随地可取，隔离其他线程。

ThreadLocal基本操作

构造函数

ThreadLocal的构造函数签名是这样的：

```
1/**
2* Creates a thread local variable.
3* @see #withInitial(java.util.function.Supplier)
4*/
```

```
5 public ThreadLocal() {
6 }
```

内部啥也没做。

initialValue函数

initialValue函数用来设置ThreadLocal的初始值，函数签名如下：

```
1 protected T initialValue() {
2     return null;
3 }
```

该函数在调用get函数的时候会第一次调用，但是如果一开始就调用了set函数，则该函数不会被调用。通常该函数只会被调用一次，除非手动调用了remove函数之后又调用get函数，这种情况下，get函数中还是会调用initialValue函数。该函数是protected类型的，很显然是建议在子类重载该函数的，所以通常该函数都会以匿名内部类的形式被重载，以指定初始值，比如：

```
1 package com.winwill.test;
2
3 /**
4 * @author qifuguang
5 * @date 15/9/2 00:05
6 */
7 public class TestThreadLocal {
8     private static final ThreadLocal<Integer> value = new ThreadLocal<Integer>() {
9         @Override
10        protected Integer initialValue() {
11            return Integer.valueOf(1);
12        }
13    };
14
```

get函数

该函数用来获取与当前线程关联的ThreadLocal的值，函数签名如下：

```
1 public T get()
```

如果当前线程没有该ThreadLocal的值，则调用initialValue函数获取初始值返回。

set函数

set函数用来设置当前线程的该ThreadLocal的值，函数签名如下：

```
public void set(T value)
```

设置当前线程的ThreadLocal的值为value。

remove函数

remove函数用来将当前线程的ThreadLocal绑定的值删除，函数签名如下：

```
public void remove()
```

在某些情况下需要手动调用该函数，防止内存泄露。

代码演示

学习了最基本的操作之后，我们用一段代码来演示ThreadLocal的用法，该例子实现下面这个场景：

> 有5个线程，这5个线程都有一个值value，初始值为0，线程运行时用一个循环往value值相加数字。

代码实现：

```
package com.winwill.test;

/**
 * @author qifuguang
 * @date 15/9/2 00:05
 */
public class TestThreadLocal {
    private static final ThreadLocal<Integer> value = new ThreadLocal<Integer>() {
        @Override
        protected Integer initialValue() {
            return 0;
        }
    };

```

```
15  public static void main(String[] args) {
16      for (int i = 0; i < 5; i++) {
17          new Thread(new MyThread(i)).start();
18      }
19  }
20
21  static class MyThread implements Runnable {
22      private int index;
23
24      public MyThread(int index) {
25          this.index = index;
26      }
27
28      public void run() {
29          System.out.println("线程" + index + "的初始value:" + value.get());
30          for (int i = 0; i < 10; i++) {
31              value.set(value.get() + i);
32          }
33          System.out.println("线程" + index + "的累加value:" + value.get());
34      }
35  }
36
```

执行结果为：

> 线程0的初始value:0
> 线程3的初始value:0
> 线程2的初始value:0
> 线程2的累加value:45
> 线程1的初始value:0
> 线程3的累加value:45
> 线程0的累加value:45
> 线程1的累加value:45
> 线程4的初始value:0
> 线程4的累加value:45

可以看到，各个线程的value值是相互独立的，本线程的累加操作不会影响到其他线程的值，真正达到了线程内部隔离的效果。

如何实现的

看了基本介绍，也看了最简单的效果演示之后，我们更应该好好研究下ThreadLocal内部的实现原理。如果给你设计，你会怎么设计？相信大部分人会有这样的想法：

> 每个ThreadLocal类创建一个Map，然后用线程的ID作为Map的key，实例对象作为Map的value，这样就能达到各个线程的值隔离的效果。

没错，这是最简单的设计方案，JDK最早期的ThreadLocal就是这样设计的。JDK1.3（不确定是否是1.3）之后ThreadLocal的设计换了一种方式。

我们先看看JDK8的ThreadLocal的get方法的源码:

```
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();
}
```

其中getMap的源码：

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

setInitialValue函数的源码：

```
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;
}
```

createMap函数的源码：

```
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

简单解析一下，get方法的流程是这样的：

1. 首先获取当前线程
2. 根据当前线程获取一个Map
3. 如果获取的Map不为空，则在Map中以ThreadLocal的引用作为key来在Map中获取对应的value e，否则转到5
4. 如果e不为null，则返回e.value，否则转到5
5. Map为空或者e为空，则通过initialValue函数获取初始值value，然后用ThreadLocal的引用和value作为firstKey和firstValue创建一个新的Map

然后需要注意的是Thread类中包含一个成员变量：

```
ThreadLocal.ThreadLocalMap threadLocals = null;
```

所以，可以总结一下ThreadLocal的设计思路：
每个Thread维护一个ThreadLocalMap映射表，这个映射表的key是ThreadLocal实例本身，value是真正需要存储的Object。
这个方案刚好与我们开始说的简单的设计方案相反。查阅了一下资料，这样设计的主要有以下几点优势：

- 这样设计之后每个Map的Entry数量变小了：之前是Thread的数量，现在是ThreadLocal的数量，能提高性能，据说性能的提升不是一点两点(没有亲测)
- 当Thread销毁之后对应的ThreadLocalMap也就随之销毁了，能减少内存使用量。

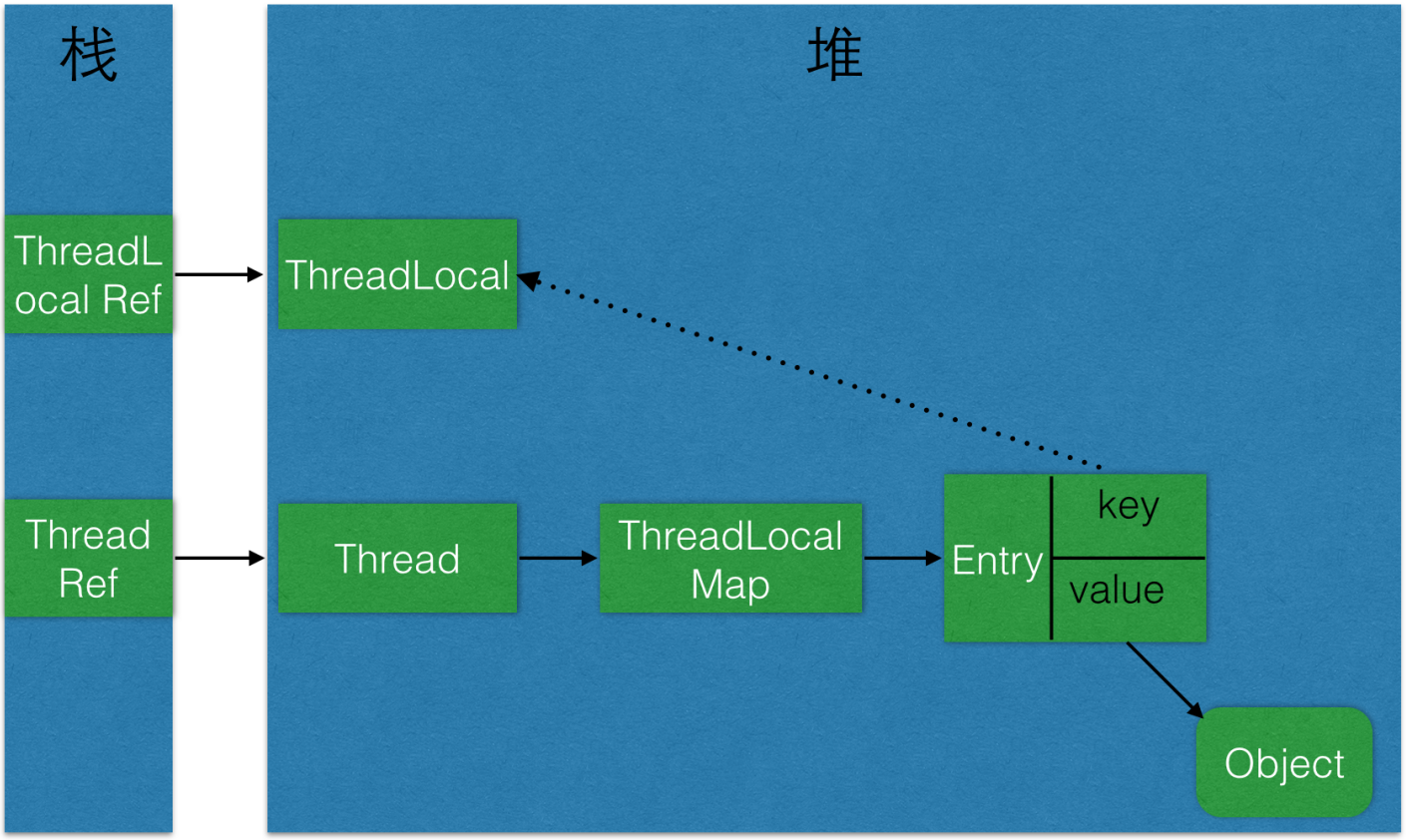
再深入一点

先交代一个事实：**ThreadLocalMap是使用ThreadLocal的弱引用作为Key的**：

```
static class ThreadLocalMap {
```

```
2
3       /**
4        * The entries in this hash map extend WeakReference, using
5        * its main ref field as the key (which is always a
6        * ThreadLocal object).  Note that null keys (i.e. entry.get()
7        * == null) mean that the key is no longer referenced, so the
8        * entry can be expunged from table.  Such entries are referred to
9        * as "stale entries" in the code that follows.
10       */
11      static class Entry extends WeakReference<ThreadLocal<?>> {
12          /** The value associated with this ThreadLocal. */
13          Object value;
14
15          Entry(ThreadLocal<?> k, Object v) {
16              super(k);
17              value = v;
18          }
19      }
20      ...
21      ...
22
```

下图是本文介绍到的一些对象之间的引用关系图，实线表示强引用，虚线表示弱引用：



By: http://qifuguang.me

然后网上就传言，ThreadLocal会引发内存泄露，他们的理由是这样的：

> 如上图，ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用引用他，那么系统gc的时候，这个ThreadLocal势必会被回收，这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value，如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链：
> **Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value**
> 永远无法回收，造成内存泄露。

我们来看看到底会不会出现这种情况。
其实，在JDK的ThreadLocalMap的设计中已经考虑到这种情况，也加上了一些防护措施，下面是ThreadLocalMap的getEntry方法的源码：

```
1 private Entry getEntry(ThreadLocal<?> key) {
2    int i = key.threadLocalHashCode & (table.length - 1);
3    Entry e = table[i];
4    if (e != null && e.get() == key)
5        return e;
6    else
7        return getEntryAfterMiss(key, i, e);
8 }
```

getEntryAfterMiss函数的源码：

```
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;
}
```

expungeStaleEntry函数的源码：

```
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) {
```

```
18                 e.value = null;
19                 tab[i] = null;
20                 size--;
21             } else {
22                 int h = k.threadLocalHashCode & (len - 1);
23                 if (h != i) {
24                     tab[i] = null;
25 
26                     // Unlike Knuth 6.4 Algorithm R, we must scan until
27                     // null because multiple entries could have been stale.
28                     while (tab[h] != null)
29                         h = nextIndex(h, len);
30                     tab[h] = e;
31                 }
32             }
33         }
34         return i;
35     }
```

整理一下ThreadLocalMap的getEntry函数的流程：

1. 首先从ThreadLocal的直接索引位置(通过ThreadLocal.threadLocalHashCode & (len-1)运算得到)获取Entry e，如果e不为null并且key相同则返回e；
2. 如果e为null或者key不一致则向下一个位置查询，如果下一个位置的key和当前需要查询的key相等，则返回对应的Entry，否则，如果key值为null，则擦除该位置的Entry，否则继续向下一个位置查询

在这个过程中遇到的key为null的Entry都会被擦除，那么Entry内的value也就没有强引用链，自然会被回收。仔细研究代码可以发现，set操作也有类似的思想，将key为null的这些Entry都删除，防止内存泄露。
但是光这样还是不够的，上面的设计思路依赖一个前提条件：**要调用ThreadLocalMap的genEntry函数或者set函数。**这当然是不可能任何情况都成立的，所以很多情况下需要使用者手动调用ThreadLocal的remove函数，手动删除不再需要的ThreadLocal，防止内存泄露。所以JDK建议将ThreadLocal变量定义成private static的，这样的话ThreadLocal的生命周期就更长，由于一直存在ThreadLocal的强引用，所以ThreadLocal也就不会被回收，也就能保证任何时候都能根据ThreadLocal的弱引用访问到Entry的value值，然后remove它，防止内存泄露。

声明

本文为作者自己的个人见解，如理解有误，请留言相告，谢谢。
转载请注明出处：
http://qifuguang.me/2015/09/02/[Java%E5%B9%B6%E5%8F%91%E5%8C%85%E5%AD%A6%E4%B9%A0%E4%B8%83]%E8%A7%A3%E5%AF%86ThreadLocal/