<https://blog.csdn.net/puppylpg/article/details/80433271>

# ThreadLocal原理及内存泄露预防

2018年05月24日 13:38:21 [puppylpg](https://me.csdn.net/puppylpg) 阅读数：805 标签： [ThreadLocal](http://so.csdn.net/so/search/s.do?q=ThreadLocal&t=blog)[WeakReference](http://so.csdn.net/so/search/s.do?q=WeakReference&t=blog) 更多

个人分类： [Java](https://blog.csdn.net/puppylpg/article/category/2720027)

版权声明：本文为博主原创文章，欢迎转载欢迎传播，注不注明出处随你，只要对更多人有所帮助就行。 https://blog.csdn.net/puppylpg/article/details/80433271

* [前言](https://blog.csdn.net/puppylpg/article/details/80433271#%E5%89%8D%E8%A8%80)
* [原理](https://blog.csdn.net/puppylpg/article/details/80433271#%E5%8E%9F%E7%90%86)
  + [为什么key使用弱引用](https://blog.csdn.net/puppylpg/article/details/80433271#%E4%B8%BA%E4%BB%80%E4%B9%88key%E4%BD%BF%E7%94%A8%E5%BC%B1%E5%BC%95%E7%94%A8)
* [内存泄露](https://blog.csdn.net/puppylpg/article/details/80433271#%E5%86%85%E5%AD%98%E6%B3%84%E9%9C%B2)
* [线程池](https://blog.csdn.net/puppylpg/article/details/80433271#%E7%BA%BF%E7%A8%8B%E6%B1%A0)
* [附：强引用-软引用-弱引用](https://blog.csdn.net/puppylpg/article/details/80433271#%E9%99%84%E5%BC%BA%E5%BC%95%E7%94%A8-%E8%BD%AF%E5%BC%95%E7%94%A8-%E5%BC%B1%E5%BC%95%E7%94%A8)

参阅：<http://www.importnew.com/22039.html>

# 前言

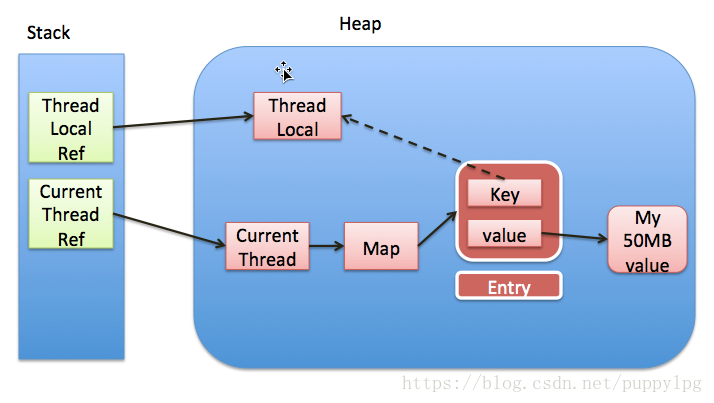
ThreadLocal提供了线程独有的局部变量，可以在整个线程存活的过程中随时取用，极大地方便了一些逻辑的实现。常见的ThreadLocal用法有：   
- 存储单个线程上下文信息。比如存储id等；   
- 使变量线程安全。变量既然成为了每个线程内部的局部变量，自然就不会存在并发问题了；   
- 减少参数传递。比如做一个trace工具，能够输出工程从开始到结束的整个一次处理过程中所有的信息，从而方便debug。由于需要在工程各处随时取用，可放入ThreadLocal。

# 原理

ThreadLocal里类型的变量，其实是放入了当前Thread里。每个Thread都有一个{@link Thread#threadLocals}，它是一个map：{@link java.lang.ThreadLocal.ThreadLocalMap}。这个map的entry是{@link java.lang.ThreadLocal.ThreadLocalMap.Entry}，具体的key和value类型分别是{@link ThreadLocal}和 {@link Object}。

**（注：实际是ThreadLocal的弱引用*WeakReference<ThreadLocal<?>>*，但可以先简单理解为ThreadLocal。）**

当设置一个ThreadLocal变量时，这个map里就多了一对ThreadLocal -> Object的映射。



通过一个简单程序来说明上图：

package example.concurrency.tl;

/\*\*

\* @author liuhaibo on 2018/05/23

\*/

public class ThreadLocalDemo {

private static final ThreadLocal<Integer> TL\_INT = ThreadLocal.withInitial(() -> 6);

private static final ThreadLocal<String> TL\_STRING = ThreadLocal.withInitial(() -> "Hello, world");

public static void main(String... args) {

// 6

System.out.println(TL\_INT.get());

TL\_INT.set(TL\_INT.get() + 1);

// 7

System.out.println(TL\_INT.get());

TL\_INT.remove();

// 会重新初始化该value，6

System.out.println(TL\_INT.get());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* Stack-ThreadLocalRef：TL\_INT;
* Stack-CurrentThreadRef： 当前线程在栈中的引用；
* Heap-ThreadLocal：TL\_INT引用所对应的ThreadLocal实例；
* Heap-CurrentThread：当前线程实例；
* Heap-Map：当前线程内部的threadLocals变量所对应的map实例；
* Heap-Entry：上述map的entry；
* Heap-Entry-Key：上述entry的键的**弱引用**；
* Heap-Entry-Value：上述entry的值的**强引用**；

对于上述程序，实际上我们在当前线程的threadlocals这个map里放了如下内容：

| TL\_INT -> 6 |

| TL\_STRING -> "Hello, world"|

* 1
* 2

对于一个普通的map，取其中某个key对应的值分两步：   
1. 找到这个map；   
2. 在map中，给出key，得到value。

想取出我们存放在当前线程里的map里的值同样需要这两步。但是，我们不需要告诉jvm map在哪儿，因为jvm知道当前线程，也知道其局部变量map。所以最终的get操作只需要知道key就行了：int localInt = TL\_INT.get();。   
看起来有些奇怪，不同于常规的map的get操作的接口的样子。

## 为什么key使用弱引用

不妨反过来想想，如果使用强引用，当ThreadLocal对象（假设为ThreadLocal@123456）的引用（即：TL\_INT，是一个强引用，指向ThreadLocal@123456）被回收了，ThreadLocalMap本身依然还持有ThreadLocal@123456的强引用，如果没有手动删除这个key，则ThreadLocal@123456不会被回收，所以只要当前线程不消亡，ThreadLocalMap引用的那些对象就不会被回收，可以认为这导致Entry内存泄漏。

那使用弱引用的好处呢？

如果使用弱引用，那指向ThreadLocal@123456对象的引用就两个：TL\_INT强引用，和ThreadLocalMap中Entry的弱引用。一旦TL\_INT被回收，则指向ThreadLocal@123456的就只有弱引用了，在下次gc的时候，这个ThreadLocal@123456就会被回收。

那么问题来了，ThreadLocal@123456对象只是作为ThreadLocalMap的一个key而存在的，现在它被回收了，但是它对应的value并没有被回收，内存泄露依然存在！而且key被删了之后，变成了null，value更是无法被访问到了！针对这一问题，ThreadLocalMap类的设计本身已经有了这一问题的解决方案，那就是在每次get()/set()/remove()ThreadLocalMap中的值的时候，会自动清理key为null的value。如此一来，value也能被回收了。

既然对key使用弱引用，能使key自动回收，那为什么不对value使用弱引用？答案显而易见，假设往ThreadLocalMap里存了一个value，gc过后value便消失了，那就无法使用ThreadLocalMap来达到存储全线程变量的效果了。（但是再次访问该key的时候，依然能取到value，此时取得的value是该value的初始值。即在删除之后，如果再次访问，取到null，会重新调用初始化方法。）

# 内存泄露

总结一下内存泄露（本该回收的无用对象没有得到回收）的原因：   
- 弱引用一定程度上回收了无用对象，但前提是开发者手动清理掉ThreadLocal对象的强引用（如TL\_INT）。只要线程一直不死，ThreadLocalMap的key-value一直在涨。

**解决方法：当某个ThreadLocal变量（比如：TL\_INT）不再使用时，记得TL\_INT.remove()，删除该key。**

在上例中，ThreadLocalDemo持有static的ThreadLocal类型：TL\_INT，导致TL\_INT的生命周期跟ThreadLocalDemo类的生命周期一样长。意味着TL\_INT不会被回收，弱引用形同虚设，所以当前线程无法通过ThreadLocalMap的防护措施清除TL\_INT所对应的value（Integer）的强引用。

通常，我们需要**保证作为key的TL\_INT类型能够被全局访问到**，同时也必须**保证其为单例**，因此，在一个类中将其设为static类型便成为了惯用做法。

# 线程池

使用了线程池，可以达到“线程复用”的效果。但是归还线程之前记得清除ThreadLocalMap，要不然再取出该线程的时候，ThreadLocal变量还会存在。这就不仅仅是内存泄露的问题了，整个业务逻辑都可能会出错。

解决方法参考：

/\*\*

\* Method invoked upon completion of execution of the given Runnable.

\* This method is invoked by the thread that executed the task. If

\* non-null, the Throwable is the uncaught {@code RuntimeException}

\* or {@code Error} that caused execution to terminate abruptly.

\*

\* <p>This implementation does nothing, but may be customized in

\* subclasses. Note: To properly nest multiple overridings, subclasses

\* should generally invoke {@code super.afterExecute} at the

\* beginning of this method.

\*

... some deleted ...

\*

\* @param r the runnable that has completed

\* @param t the exception that caused termination, or null if

\* execution completed normally

\*/

protected void afterExecute(Runnable r, Throwable t) { }

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

override {@link ThreadPoolExecutor#afterExecute(r, t)}方法，对ThreadLocalMap进行清理，比如：

protected void afterExecute(Runnable r, Throwable t) {

// you need to set this field via reflection.

Thread.currentThread().threadLocals = null;

}

* 1
* 2
* 3
* 4

参考：<https://stackoverflow.com/a/30328722/7676237>

所以ThreadLocal最好还是不要和线程池一起使用，就没这么多问题了:D

# 附：强引用-软引用-弱引用

* 强引用：普通的引用，强引用指向的对象不会被回收；
* 软引用：仅有软引用指向的对象，只有发生gc且内存不足，才会被回收；
* 弱引用：仅有弱引用指向的对象，只要发生gc就会被回收。

看一个例子就明白强引用、软引用、弱引用的区别：

package example.reference;

import java.lang.ref.SoftReference;

import java.lang.ref.WeakReference;

/\*\*

\* @author liuhaibo on 2018/03/06

\*/

public class WeakRefDemo {

public static void main(String... args) {

// all these objects have a strong reference

Object a = new Object();

Object b = new Object();

Object c = new Object();

// other references to these objects

Object strongA = a;

SoftReference<Object> softB = new SoftReference<>(b);

WeakReference<Object> weakC = new WeakReference<>(c);

// free the former strong references to these objects:

// there is still a strong reference(strongA) to the first object

a = null;

// only a soft reference(softB) refers to the second object

b = null;

// only a weak reference(weakC) refers to the third object

c = null;

System.out.println("Before gc...");

System.out.println(String.format("strongA = %s, softB = %s, weakC = %s", strongA, softB.get(), weakC.get()));

System.out.println("Run GC...");

System.gc();

// object with only soft reference will be cleaned only if memory is not enough: 用来做缓存很不错

// object with only weak reference will be cleaned after a gc operation:

System.out.println("After gc...");

System.out.println(String.format("strongA = %s, softB = %s, weakC = %s", strongA, softB.get(), weakC.get()));

}

}

* 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

Output：

Before gc...

strongA = java.lang.Object@3af49f1c, softB = java.lang.Object@19469ea2, weakC = java.lang.Object@13221655

Run GC...

After gc...

strongA = java.lang.Object@3af49f1c, softB = java.lang.Object@19469ea2, weakC = null