ThreadLocal 的误解

可能大部分人的想法和我当初的想法一样，都是以为在 ThreadLocal 里使用一个 Map，这个 Map 的键为 Thread，值为绑定的变量。其实如果这样做是有问题的：

1. 就是当线程回收时，该线程绑定的变量不能被自动的回收，因为变量存储在 ThreadLocal 里，必须显式的去回收。如果此变量存储在线程里，那么线程回收时，这个变量没有被其他引用指向的话，它便随着线程一起回收。
2. 另外不这样做还有一个好处：如果 Map 在 ThreadLocal 里，那你必须得考虑线程同步访问这个 Map，但是这确实没有必要，因为线程访问自己的变量，和其他线程没有直接的关系，所以把 Map 放在线程里，就不需要做同步的处理，这样即加快了访问的速度。

其实实现不是这样的：每个线程都包含一个 ThreadLocal.ThreadLocalMap 变量 threadLocals（延迟创建的），这个映射（Map）目的就是为每个线程存储关联到使用到的不同 ThreadLocal 的变量，这个很好理解，因为，一个线程可能使用到多个不同的 ThreadLocal 对象，每一个 ThreadLocal 对象的值都被认为是不同的。于是，每次调用 ThreadLocal 的 get() 方法，其实就是获取当前线程（Thread.currentThread()），然后从 threadLocals 映射里，根据 ThreadLocal 对象，找出其关联的拷贝，这个值便是当前线程的，隔离于其他线程的值。

我们知道，ThreadLocal.ThreadLocalMap 映射使用的键是被 WeakReference 包装的 ThreadLocal 对象，如果 ThreadLocal 对象没有其他强引用和软引用指向时，该线程也不会继续持有 ThreadLocal 对象，因为根据 JVM 规范，它会被垃圾回收器下次回收时销毁，这一定程度避免了内存泄露，但不表示不会出现内存泄露，关于 ThreadLocal 引起的内存泄露，特别是导致 ClassLoader 不能被回收，网上有很多文章都在讨论。在 Java 1.5 开始，加入了 remove()方法，这样我们可以显式的调用此方法，释放内存，所以使用 ThreadLocal 要特别注意内存泄露的问题。

看来大家对 ThreadLocal 内存泄露的原因有点误解，其导致 Classloader 内存泄露的原因在这篇博文写的很清楚了 Classloader leaks:，我在这里补充一下，jvm 规范下的 classloader 能被回收的条件是，所有该 classloader 产生的所有对象都被回收了：我们知道，对象有一个隐式的引用指向它的类型 class 对象，而 class 对象有个隐式的引用指向它的 classloader，所以如果有一个对象不回收，那么可能导致整个 classloader 不能够被回收。为什么是可能？这个是 jvm 规范规定的规范，但是实现并不尽相同，sun 的 hotspot 目前版本是不会对还有不能回收对象的 classloader 做回收的，但是其他的 jvm 实现，比如 Jrocket，ibm 的 jvm 可能会产生回收那些无用的 class 对象，但目前没有看到源码，只能从现象猜测。线程池产生泄漏的一个原因主要可能是对线程重用产生的，这点读者有条件可以去测试。