<https://mp.weixin.qq.com/s/pj2pTD2TgCbYF-oWcwD5mw>

# 一文搞懂 ThreadLocal 原理

当多线程访问共享可变数据时，涉及到线程间同步的问题，并不是所有时候，都要用到共享数据，所以就需要线程封闭出场了。

数据都被封闭在各自的线程之中，就不需要同步，这种通过将数据封闭在线程中而避免使用同步的技术称为**线程封闭**。

本文主要介绍线程封闭中的其中一种体现：ThreadLocal，将会介绍什么是 ThreadLocal；从 ThreadLocal 源码角度分析，最后介绍 ThreadLocal 的应用场景。

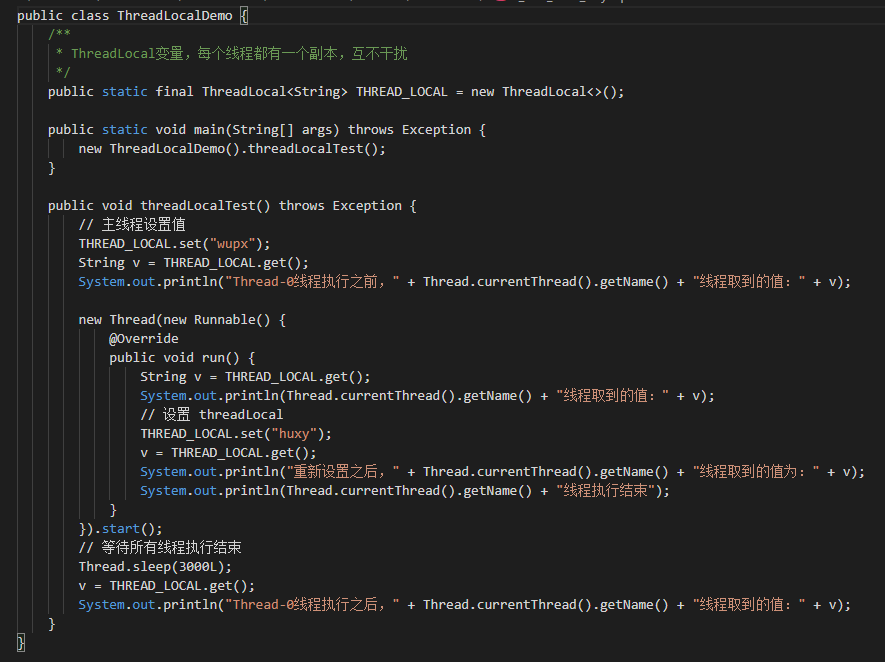
## **什么是 ThreadLocal？**

ThreadLocal 是 Java 里一种特殊变量，它是一个线程级别变量，每个线程都有一个 ThreadLocal 就是每个线程都拥有了自己独立的一个变量，竞态条件被彻底消除了，在并发模式下是绝对安全的变量。

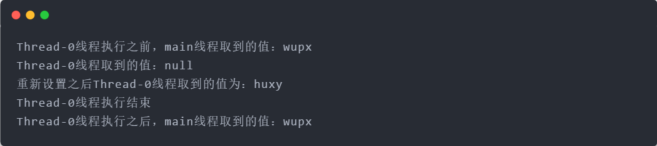
可以通过 ThreadLocal<T> value = new ThreadLocal<T>(); 来使用。

会自动在每一个线程上创建一个 T 的副本，副本之间彼此独立，互不影响，可以用 ThreadLocal 存储一些参数，以便在线程中多个方法中使用，用以代替方法传参的做法。

下面通过例子来了解下 ThreadLocal：



首先通过 static final 定义了一个 THREAD\_LOCAL 变量，其中 static 是为了确保全局只有一个保存 String 对象的 ThreadLocal 实例；final 确保 ThreadLocal 的实例不可更改，防止被意外改变，导致放入的值和取出来的不一致，另外还能防止 ThreadLocal 的内存泄漏。上面的例子是演示在不同的线程中获取它会得到不同的结果，运行结果如下：

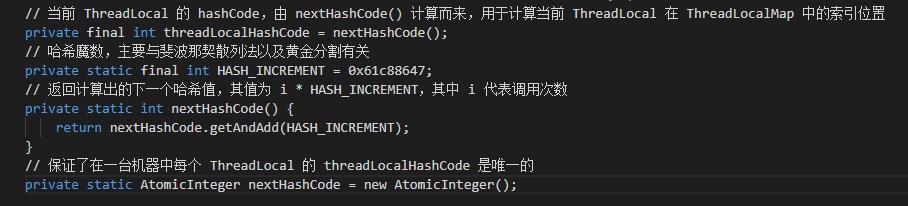


首先在 Thread-0 线程执行之前，先给 THREAD\_LOCAL 设置为 wupx，然后可以取到这个值，然后通过创建一个新的线程以后去取这个值，发现新线程取到的为 null，意外着这个变量在不同线程中取到的值是不同的，不同线程之间对于 ThreadLocal 会有对应的副本，接着在线程 Thread-0 中执行对 THREAD\_LOCAL 的修改，将值改为 huxy，可以发现线程 Thread-0 获取的值变为了 huxy，主线程依然会读取到属于它的副本数据 wupx，这就是线程的封闭。

看到这里，我相信大家一定会好奇 ThreadLocal 是如何做到多个线程对同一对象 set 操作，但是 get 获取的值还都是每个线程 set 的值呢，接下来就让我们进入源码解析环节：

## **ThreadLocal 源码解析**

首先看下 ThreadLocal 都有哪些重要属性：

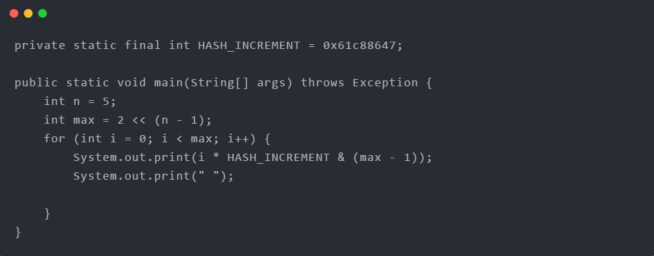


其中的 HASH\_INCREMENT 也不是随便取的，它转化为十进制是 1640531527，2654435769 转换成 int 类型就是 -1640531527，2654435769 等于 (√5-1)/2 乘以 2 的 32 次方。(√5-1)/2 就是黄金分割数，近似为 0.618，也就是说 0x61c88647 理解为一个黄金分割数乘以 2 的 32 次方，它可以保证 nextHashCode 生成的哈希值，均匀的分布在 2 的幂次方上，且小于 2 的 32 次方。

下面是 javaspecialists 中一篇文章对它的介绍：

This number represents the golden ratio (sqrt(5)-1) times two to the power of 31 ((sqrt(5)-1) \* (2^31)). The result is then a golden number, either 2654435769 or -1640531527.

下面用例子来证明下：

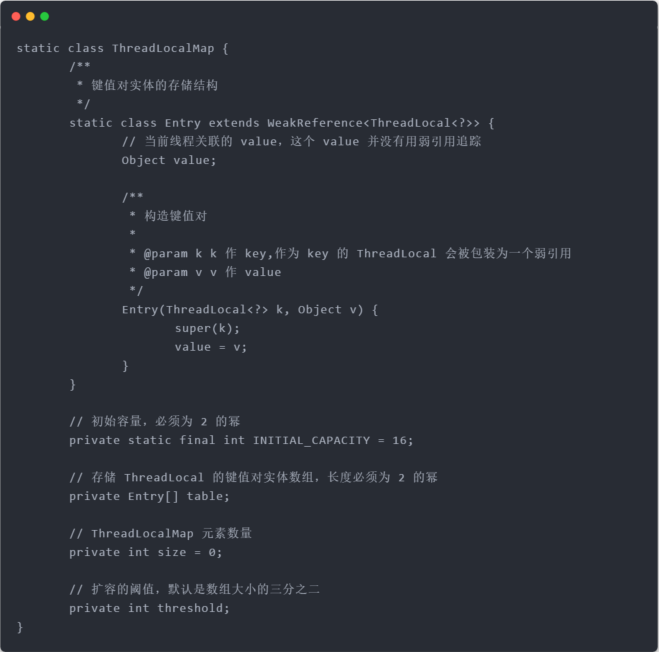


运行结果为：0 7 14 21 28 3 10 17 24 31 6 13 20 27 2 9 16 23 30 5 12 19 26 1 8 15 22 29 4 11 18 25

可以发现元素索引值完美的散列在数组当中，并没有出现冲突。

### **ThreadLocalMap**

除了上述属性外，还有一个静态内部类 ThreadLocalMap，ThreadLocalMap 是 ThreadLocal 的静态内部类，当一个线程有多个 ThreadLocal 时，需要一个容器来管理多个 ThreadLocal，ThreadLocalMap 的作用就是管理线程中多个 ThreadLocal，源码如下：



从源码中看到 ThreadLocalMap 其实就是一个简单的 Map 结构，底层是数组，有初始化大小，也有扩容阈值大小，数组的元素是 Entry，**Entry 的 key 就是 ThreadLocal 的引用，value 是 ThreadLocal 的值**。ThreadLocalMap 解决 hash 冲突的方式采用的是**线性探测法**，如果发生冲突会继续寻找下一个空的位置。

这样的就有可能会发生内存泄漏的问题，下面让我们进行分析：

### **ThreadLocal 内存泄漏**

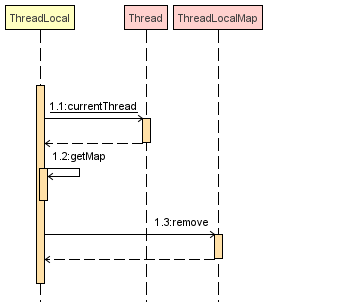
ThreadLocal 在没有外部强引用时，发生 GC 时会被回收，那么 ThreadLocalMap 中保存的 key 值就变成了 null，而 Entry 又被 threadLocalMap 对象引用，threadLocalMap 对象又被 Thread 对象所引用，那么当 Thread 一直不终结的话，value 对象就会一直存在于内存中，也就导致了内存泄漏，直至 Thread 被销毁后，才会被回收。

那么如何避免内存泄漏呢？

在使用完 ThreadLocal 变量后，需要我们手动 remove 掉，防止 ThreadLocalMap 中 Entry 一直保持对 value 的强引用，导致 value 不能被回收，其中 remove 源码如下所示：



remove 方法的时序图如下所示：

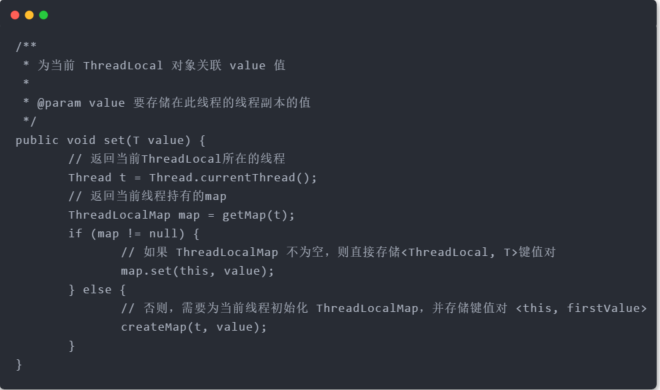


remove 方法是先获取到当前线程的 ThreadLocalMap，并且调用了它的 remove 方法，从 map 中清理当前 ThreadLocal 对象关联的键值对，这样 value 就可以被 GC 回收了。

那么 ThreadLocal 是如何实现线程隔离的呢？

### **ThreadLocal 的 set 方法**

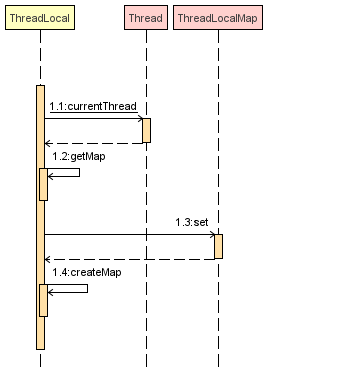
我们先去看下 ThreadLocal 的 set 方法，源码如下：



set 方法的作用是把我们想要存储的 value 给保存进去。set 方法的流程主要是：

* 先获取到当前线程的引用
* 利用这个引用来获取到 ThreadLocalMap
* 如果 map 为空，则去创建一个 ThreadLocalMap
* 如果 map 不为空，就利用 ThreadLocalMap 的 set 方法将 value 添加到 map 中

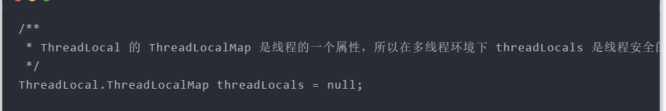
set 方法的时序图如下所示：



其中 map 就是我们上面讲到的 ThreadLocalMap，可以看到它是通过当前线程对象获取到的 ThreadLocalMap，接下来我们看 getMap方法的源代码：



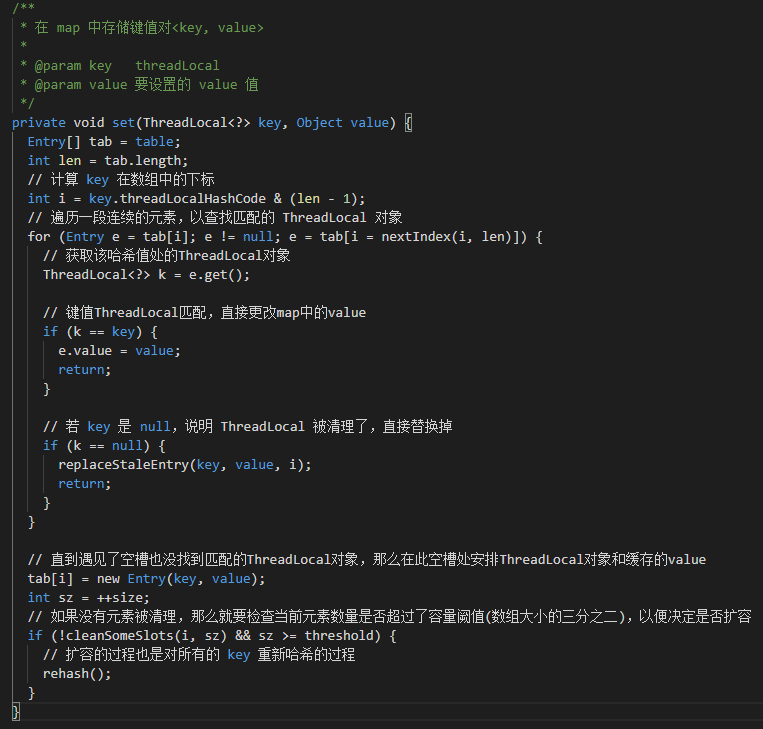
getMap 方法的作用主要是获取当前线程内的 ThreadLocalMap 对象，原来这个 ThreadLocalMap 是线程的一个属性，下面让我们看看 Thread 中的相关代码：



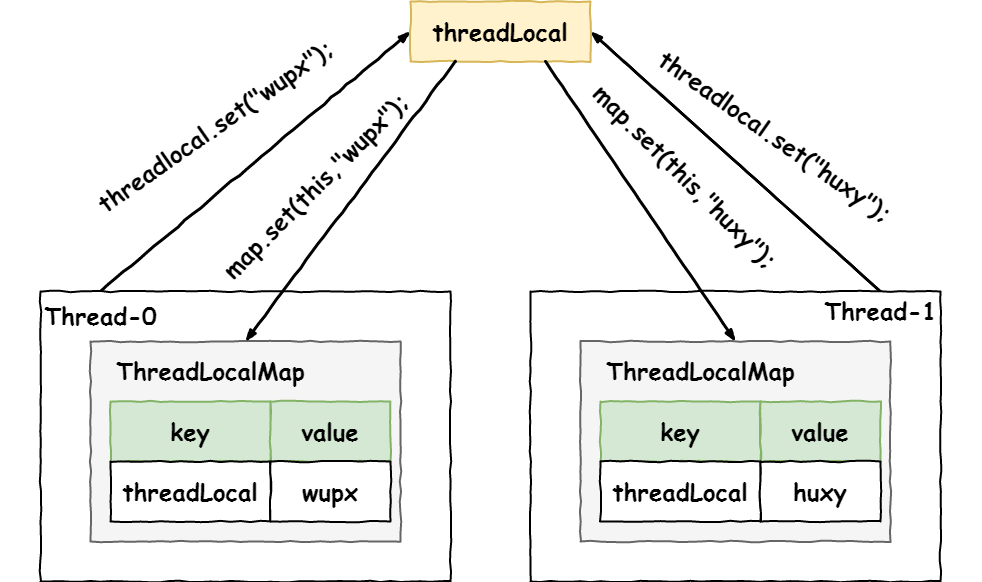
可以看出每个线程都有 ThreadLocalMap 对象，被命名为 threadLocals，默认为 null，所以每个线程的 ThreadLocals 都是隔离独享的。

调用 ThreadLocalMap.set() 时，会把当前 threadLocal 对象作为 key，想要保存的对象作为 value，存入 map。

其中 ThreadLocalMap.set() 的源码如下：



相信到这里，大家应该对 Thread、ThreadLocal 以及 ThreadLocalMap 的关系有了进一步的理解，下图为三者之间的关系：



### **ThreadLocal 的 get 方法**

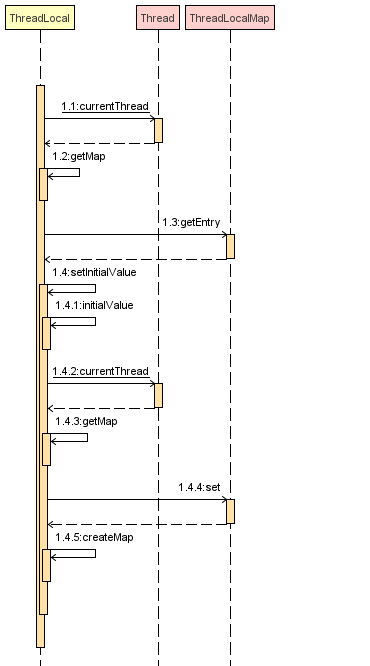
了解完 set 方法后，让我们看下 get 方法，源码如下：



get 方法的主要流程为：

* 先获取到当前线程的引用
* 获取当前线程内部的 ThreadLocalMap
* 如果 map 存在，则获取当前 ThreadLocal 对应的 value 值
* 如果 map 不存在或者找不到 value 值，则调用 setInitialValue() 进行初始化

get 方法的时序图如下所示：



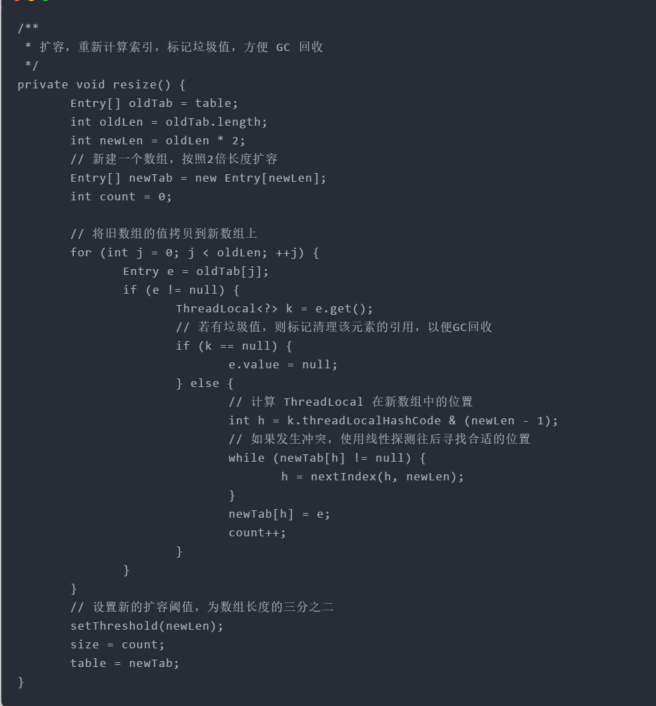
其中每个 Thread 的 ThreadLocalMap 以 threadLocal 作为 key，保存自己线程的 value 副本，也就是保存在每个线程中，并没有保存在 ThreadLocal 对象中。

其中 ThreadLocalMap.getEntry() 方法的源码如下：



### **ThreadLocalMap 的 resize 方法**

当 ThreadLocalMap 中的 ThreadLocal 的个数超过容量阈值时，ThreadLocalMap 就要开始扩容了，我们一起来看下 resize 的源代码：



resize 方法主要是进行扩容，同时会将垃圾值标记方便 GC 回收，扩容后数组大小是原来数组的两倍。

## **ThreadLocal 应用场景**

ThreadLocal 的特性也导致了应用场景比较广泛，主要的应用场景如下：

* 线程间数据隔离，各线程的 ThreadLocal 互不影响
* 方便同一个线程使用某一对象，避免不必要的参数传递
* 全链路追踪中的 traceId 或者流程引擎中上下文的传递一般采用 ThreadLocal
* Spring 事务管理器采用了 ThreadLocal
* Spring MVC 的 RequestContextHolder 的实现使用了 ThreadLocal