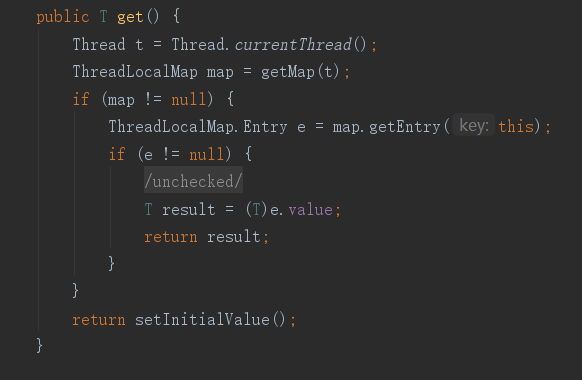
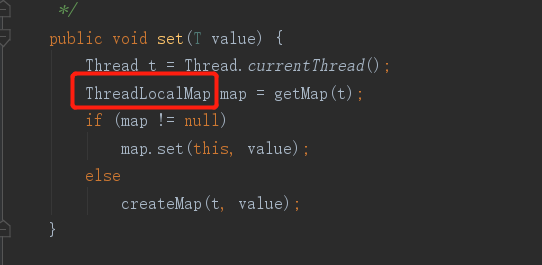
# ThreadLocal

## ThreadLocal是什么

首先，它是一个数据结构，有点像HashMap，可以保存"key : value"键值对，但是一个ThreadLocal只能保存一个，并且各个线程的数据互不干扰。





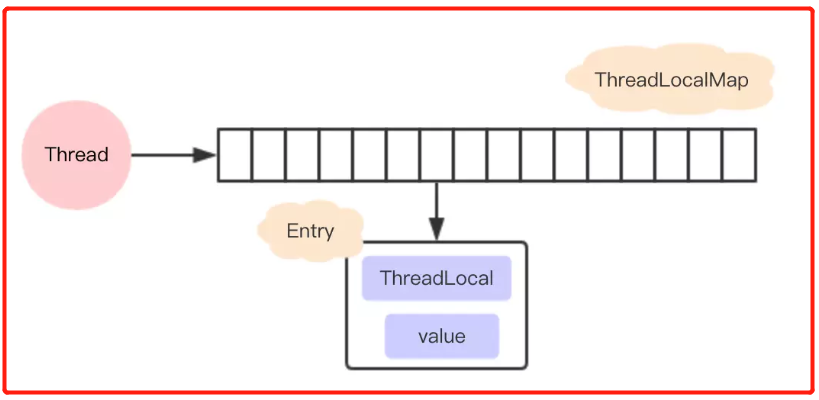
可以发现，每个线程中都有一个ThreadLocalMap数据结构，当执行set方法时，其值是保存在当前线程的threadLocals变量中，当执行方法get时，是从当前线程的threadLocals变量获取。

所以在线程1中set的值，对线程2来说是摸不到的，而且在线程2中重新set的话，也不会影响到线程1中的值，保证了线程之间不会相互干扰。

## ThreadLoalMap

从名字上看，可以猜到它也是一个类似HashMap的数据结构，但是在ThreadLocal中，并没实现Map接口。

在ThreadLocalMap中，也是初始化一个带下16的Entry数组，Entry对象用来保存每一个Key-Value键值对，只不过这里的key永远是ThreadLocal对象，通过ThreadLocal对象的Set方法，结果把ThreadLocal对象自己当做Key,放进ThreadLocalMap中

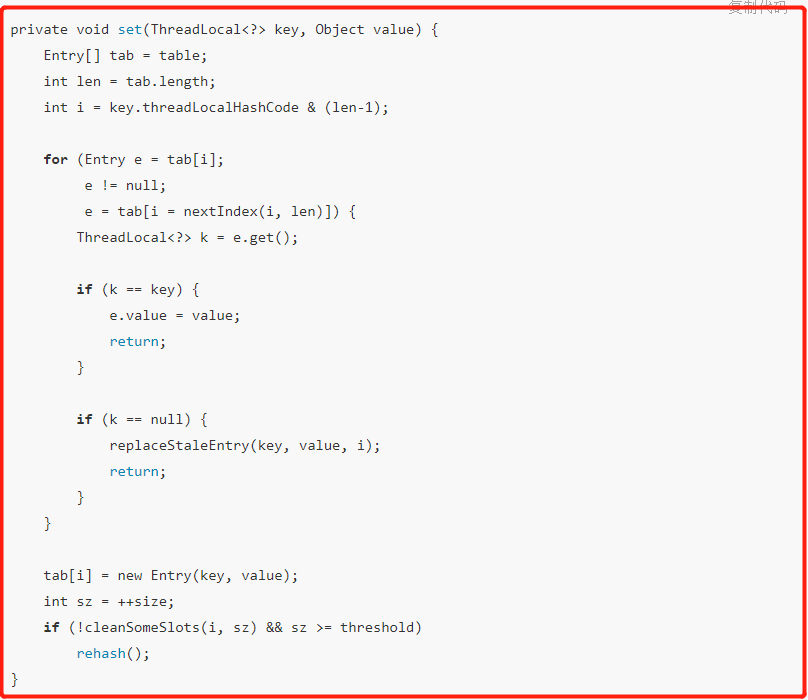


这里需要注意的是，ThreadLoalMap的Entry是继承WeakReference,和HashMap很大的区别死，Entry没有Next字段，所以就不存在链表

## hash冲突

没有链表结构，那发生hash冲突了怎么办？

先看看ThreadLoalMap中插入一个key-value的实现



每个ThreadLocal对象都有一个hash值threadLocalHashCode，每初始化一个ThreadLocal对象，hash值就增加一个固定的大小0x61c88647。

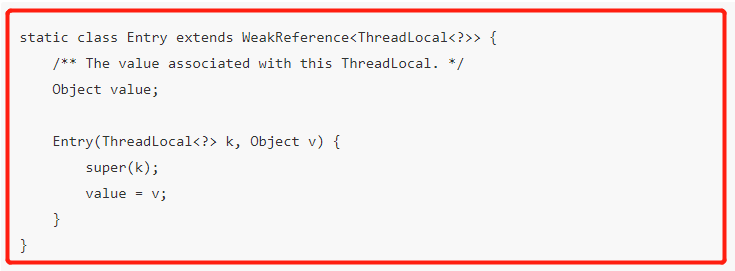
在插入过程中，根据ThreadLocal对象的hash值，定位到table中的位置i，过程如下：1、如果当前位置是空的，那么正好，就初始化一个Entry对象放在位置i上；2、不巧，位置i已经有Entry对象了，如果这个Entry对象的key正好是即将设置的key，那么重新设置Entry中的value； 3、很不巧，位置i的Entry对象，和即将设置的key没关系，那么只能找下一个空位置；

这样的话，在get的时候，也会根据ThreadLocal对象的hash值，定位到table中的位置，然后判断该位置Entry对象中的key是否和get的key一致，如果不一致，就判断下一个位置

可以发现，set和get如果冲突严重的话，效率很低，因为ThreadLoalMap是Thread的一个属性，所以即使在自己的代码中控制了设置的元素个数，但还是不能控制其它代码的行为。

## 内存泄露

ThreadLocal可能导致内存泄漏，为什么？ 先看看Entry的实现：



通过之前的分析已经知道，当前使用ThreadLoccalMap中的数组插入一个Entry对象，按理说Key-Value都应该可以以强引用保存在Entry中，但在ThreadLocalMap的实现中，Key被保存到了WeakReference中。

这就导致了一个问题，ThreadLocal在没有外部强引用时，发生GC时会被回收，如果创建ThreadLocal的线程一直持续运行，那么这个Entry对象中的value就有可能一直得不到回收，发生内存泄露。

## 如何避免内存泄露

既然已经发现有内存泄露的隐患，自然有应对的策略，在调用ThreadLocal的get()、set()可能会清除ThreadLocalMap中key为null的Entry对象，这样对应的value就没有GC Roots可达了，下次GC的时候就可以被回收，当然如果调用remove方法，肯定会删除对应的Entry对象。

一般在监听器监听到回话关闭的时候进行Remove操作

