@来源 我把 ThreadLocal 能问的，都写了

你好，我是yes。

今天我们再来盘一盘 ThreadLocal ，这篇力求对 ThreadLocal 一网打尽，彻底弄懂 ThreadLocal 的机制。

有了这篇基础之后，下篇再来盘一盘 ThreadLocal 的进阶版，等我哈。

话不多说，本文要解决的问题如下：

- 为什么需要 ThreadLocal
- 应该如何设计 ThreadLocal
- 从源码看ThreadLocal 的原理
- ThreadLocal 内存泄露之为什么要用弱引用
- ThreadLocal 的最佳实践
- InheritableThreadLocal

好了，开车！

为什么需要 ThreadLocal

最近不是开放三胎政策嘛，假设你有三个孩子。

现在你带着三个孩子出去逛街，路过了玩具店，三个孩子都看中了一款变形金刚。

所以你买了一个变形金刚，打算让三个孩子轮着玩。

回到家你发现，孩子因为这个玩具吵架了，三个都争着要玩，谁也不让着谁。

这时候怎么办呢？你可以去拉架，去讲道理，说服孩子轮流玩，但这很累。

所以一个简单的办法就是出去再买两个变形金刚，这样三个孩子都有各自的变形金刚，世界就暂时得到了安宁。

映射到我们今天的主题，变形金刚就是共享变量，孩子就是程序运行的线程。

有多个线程(孩子)，争抢同一个共享变量(玩具)，就会产生冲突，而程序的解决办法是加锁(父母说服，讲道理，轮流玩)，但加锁就意味着性能的消耗(父母比较累)。

所以有一种解决办法就是避免共享(让每个孩子都各自拥有一个变形金刚)，这样线程之间就不需要竞争共享变量(孩子之间就不会争抢)。

所以为什么需要 ThreadLocal?

就是为了通过本地化资源来避免共享，避免了多线程竞争导致的锁等消耗。

这里需要强调一下，不是说任何东西都能直接通过避免共享来解决，因为有些时候就必须共享。

举个例子：当利用多线程同时累加一个变量的时候，**此时就必须共享**，因为一个线程的对变量的修改需要影响要另个线程，不然累加的结果就不对了。

再举个不需要共享的例子：比如现在每个线程需要判断当前请求的用户来进行权限判断，那这个用户信息其实就不需要共享，因为每个线程只需要管自己当前执行操作的用户信息，跟别的用户不需要有交集。

好了，道理很简单，这下子想必你已经清晰了 ThreadLocal 出现的缘由了。

再来看一下 ThreadLocal 使用的小 demo。

```java
public class YesThreadLocal {
    private static final ThreadLocal<String> threadLocalName =
ThreadLocal.withInitial(() -> Thread.currentThread().getName());
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            new Thread(() -> {
                System.out.println("threadName: " + threadLocalName.get());
            }, "yes-thread-" + i).start();
        }
    }
}

```

输出结果如下：

```
threadName: yes-thread-2
threadName: yes-thread-3
threadName: yes-thread-4
threadName: yes-thread-0
threadName: yes-thread-1
```

可以看到，我在 new 线程的时候，设置了每个线程名，每个线程都操作**同一个 ThreadLocal 对象**的 get 却返回的各自的线程名，是不是很神奇？

应该如何设计 ThreadLocal？

那应该怎么设计 ThreadLocal 来实现以上的操作，即本地化资源呢？

我们的目标已经明确了，就是用 ThreadLocal 变量来实现线程隔离。

从代码上看，可能最直接的实现方法就是将 ThreadLocal 看做一个 map，然后每个线程是 key，这样每个线程去调用 `ThreadLocal.get` 的时候，将自身作为 key 去 map 找，这样就能获取各自的值了。

听起来很完美？错了！

这样 ThreadLocal 就变成共享变量了，多个线程竞争 ThreadLocal，那就得保证 ThreadLocal 的并发安全，那就得加锁了，这样绕了一圈就又回去了。

所以这个方案不行，那应该怎么做？

答案其实上面已经讲了，**是需要在每个线程的本地都存一份值**，说白了就是每个线程需要有个变量，来存储这些需要本地化资源的值，并且值有可能有多个，所以怎么弄呢？

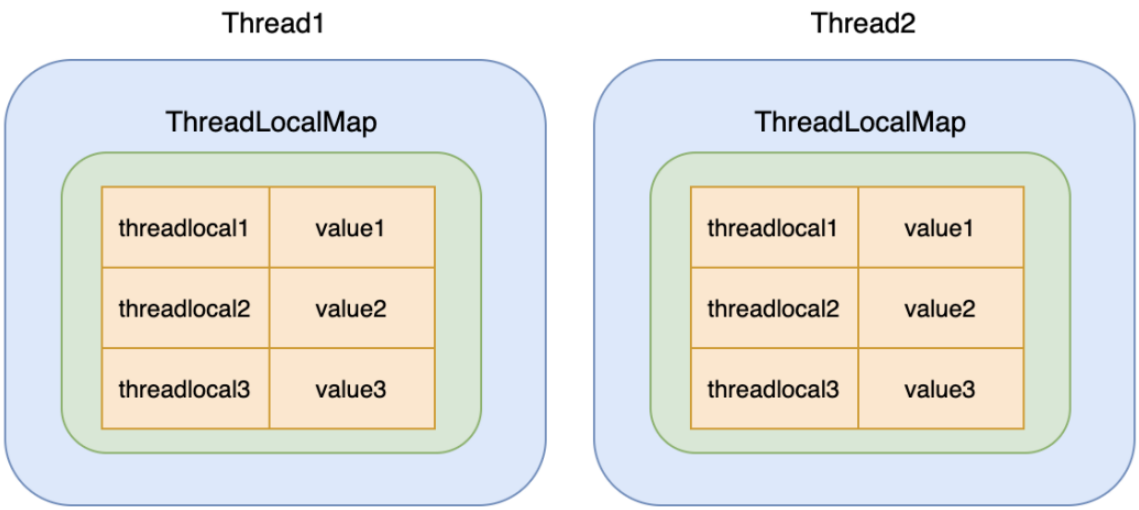
在线程对象内部搞个 map，把 ThreadLocal 对象自身作为 key，把它的值作为 map 的值。

这样每个线程可以利用同一个对象作为 key，去各自的 map 中找到对应的值。

这不就完美了嘛！比如我现在有 3 个 ThreadLocal 对象，2 个线程。

```java
ThreadLocal<String> threadLocal1 =  new ThreadLocal<>();
ThreadLocal<Integer> threadLocal2 =  new ThreadLocal<>();
ThreadLocal<Integer> threadLocal3 =  new ThreadLocal<>();
```

那此时 ThreadLocal 对象和线程的关系如下图所示：



这样一来就满足了本地化资源的需求，每个线程维护自己的变量，互不干扰，实现了变量的线程隔离，同时也满足存储多个本地变量的需求，完美！

JDK就是这样实现的！我们来看看源码。

从源码看ThreadLocal 的原理

前面我们说到 Thread 对象里面会有个 map，用来保存本地变量。

我们来看下 jdk 的 Thread 实现

```
public class Thread implements Runnable {
     // 这就是我们说的那个 map 。
    ThreadLocal.ThreadLocalMap threadLocals = null;
}
```

可以看到，确实有个 map ，不过这个 map 是 ThreadLocal 的静态内部类，记住这个变量的名字 threadLocals，下面会有用的哈。

看到这里，想必有很多小伙伴会产生一个疑问。

> 竟然这个 map 是放在 Thread 里面使用，那为什么要定义成 ThreadLocal 的静态内部类呢?

首先内部类这个东西是编译层面的概念，就像语法糖一样，经过编译器之后其实内部类会提升为外部顶级类，和平日里外部定义的类没有区别，也就是说**在 JVM 中是没有内部类这个概念的。**

一般情况下非静态内部类用在内部类，跟其他类无任何关联，专属于这个外部类使用，并且也便于调用外部类的成员变量和方法，比较方便。

而静态外部类其实就等于一个顶级类，可以独立于外部类使用，所以**更多的只是表明类结构和命名空间。**

所以说这样定义的用意就是说明 ThreadLocalMap 是和 ThreadLocal 强相关的，专用于保存线程本地变量。

现在我们来看一下 ThreadLocalMap 的定义：

```
static class ThreadLocalMap {

    /** The entries in this hash map extend WeakReference, using ...*/
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }

    /**
     * The initial capacity -- MUST be a power of two.
     */
    private static final int INITIAL_CAPACITY = 16;

    /**
     * The table, resized as necessary.
     * table.length MUST always be a power of two.
     */
    private Entry[] table;

    /**
     * The number of entries in the table.
     */
    private int size = 0;
```

重点我已经标出来了，首先可以看到这个 ThreadLocalMap 里面有个 Entry 数组，熟悉 HashMap 的小伙伴可能有点感觉了。

这个 Entry 继承了 WeakReference 即弱引用。这里需要注意，**不是说 Entry 自己是弱引用**，看到我标注的 Entry 构造函数的 `super(k)` 没，这个 key 才是弱引用。

所以 ThreadLocalMap 里有个 Entry 的数组，这个 Entry 的 key 就是 ThreadLocal 对象，value 就是我们需要保存的值。

> 那是如何通过 key 在数组中找到 Entry 然后得到 value 的呢？

这就要从上面的 `threadLocalName.get()` 说起，不记得这个代码的滑上去看下示例，其实就是调用 ThreadLocal 的 get 方法。

此时就进入 `ThreadLocal#get` 方法中了，这里就可以得知为什么不同的线程对同一个 ThreadLocal 对象调用 get 方法竟然能得到不同的值了。

这个中文注释想必很清晰了吧！

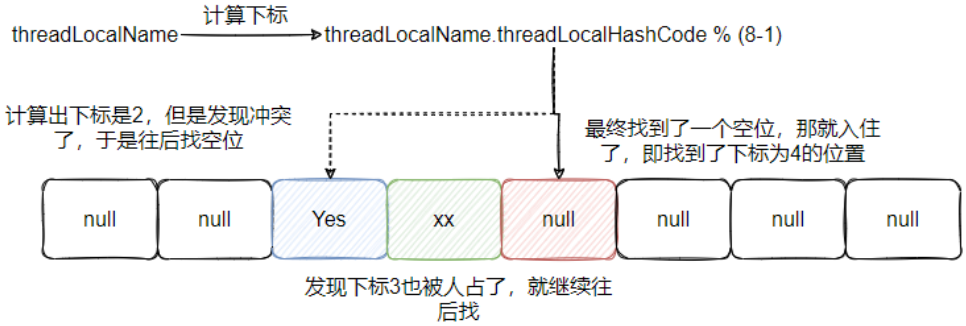
`ThreadLocal#get` 方法首先获取当前线程，然后得到当前线程的 ThreadLocalMap 变量即 threadLocals，然后将自己作为 key 从 ThreadLocalMap 中找到 Entry ，最终返回 Entry 里面的 value 值。

这里我们再看一下 key 是如何从 ThreadLocalMap 中找到 Entry 的，即 `map.getEntry(this)` 是如何实现的，其实很简单。

可以看到 ThreadLocalMap 虽然和 HashMap 一样，都是基于数组实现的，但是它们对于 Hash 冲突的解决方法不一样。

HashMap 是通过链表(红黑树)法来解决冲突，而 ThreadLocalMap 是通过**开放寻址法来解决冲突。**

听起来好像很高级，其实道理很简单，我们来看一张图就很清晰了。



所以说，如果通过 key 的哈希值得到的下标无法直接命中，则会将下标 +1，即继续往后遍历数组查找 Entry ，直到找到或者返回 null。

可以看到，这种 hash 冲突的解决效率其实不高，但是一般 ThreadLocal 也不会太多，所以用这种简单的办法解决即可。

至于代码中的 `expungeStaleEntry` 我们等下再分析，先来看下 `ThreadLocalMap#set` 方法，看看写入的怎样实现的，来看看 hash 冲突的解决方法是否和上面说的一致。

```
        */
        private void set(ThreadLocal<?> key, Object value) {

            //...

            Entry[] tab = table;
            int len = tab.length;                                  熟悉的操作，先通过key的hash值，
            int i = key.threadLocalHashCode & (len-1);             得到一个数组下标

            for (Entry e = tab[i];
                 e != null;  从这个条件可以看到，for循环是为了找到数组中的空位
                 e = tab[i = nextIndex(i, len)]) {  递增i，即数组下标实现数组往后遍历
                ThreadLocal<?> k = e.get();

                                在循环中判断，当前不为Null 的Entry 是否就是要寻找的Entry，
                if (k == key) {  如果是，那就是需要执行更新操作，则更新完，直接返回
                    e.value = value;
                    return;
                }

                if (k == null) {  如果当前的 Entry 的 key 是空的，则执行一波覆盖，然后返回
                    replaceStaleEntry(key, value, i);
                    return;
                }
            }

            tab[i] = new Entry(key, value);  新建一个 entry 给数组的空位
            int sz = ++size;  累加 Entry 数量
            if (!cleanSomeSlots(i, sz) && sz >= threshold)      如果无法清理一些Entry且
                rehash();  则进行扩容                            Entry数量大于阈值
        }
```

可以看到 set 的逻辑也很清晰。

先通过 key 的 hash 值计算出一个数组下标，然后看看这个下标是否被占用了，如果被占了看看是否就是要找的 Entry 。

如果是则进行更新，如果不是则下标++，即往后遍历数组，查找下一个位置，找到空位就 new 个 Entry 然后把坑给占用了。

当然，这种数组操作一般免不了阈值的判断，如果超过阈值则需要进行扩容。

上面的清理操作和 key 为空的情况，下面再做分析，这里先略过。

至此，我们已经分析了 ThreadLocalMap 的**核心操作 get 和 set**，想必你对 ThreadLocalMap 的原理已经从源码层面清晰了！

可能有些小伙伴对 key 的哈希值的来源有点疑惑，所以我再来补充一下 `key.threadLocalHashCode` 的分析。

```
public class ThreadLocal<T> {
    /** ThreadLocals rely on per-thread linear-probe hash maps attached ...*/
    private final int threadLocalHashCode = nextHashCode();

    /** The next hash code to be given out. Updated atomically. Starts at ...*/
    private static AtomicInteger nextHashCode =  定义一个原子类
        new AtomicInteger();

    /** The difference between successively generated hash codes - turns ...*/
    private static final int HASH_INCREMENT = 0x61c88647;  定义一个看起来很奇怪的步长

    /**
     * Returns the next hash code.
     */
    private static int nextHashCode() {   每来一个 key 就将原子类增加步长返回
        return nextHashCode.getAndAdd(HASH_INCREMENT);
    }
```

可以看到 `key.threadLocalHashCode` 其实就是调用 nextHashCode 进行一个原子类的累加。

注意看上面都是静态变量和静态方法，所以在 ThreadLocal 对象之间是共享的，然后通过固定累加一个奇怪的数字 `0x61c88647` 来分配 hash 值。

这个数字当然不是乱写的，是实验证明的一个值，即通过 `0x61c88647` 累加生成的值与 2 的幂取模的结果，可以较为均匀地分布在 2 的幂长度的数组中，这样可以减少 hash 冲突。

有兴趣的小伙伴可以深入研究一下，反正我没啥兴趣。

ThreadLocal 内存泄露之为什么要用弱引用

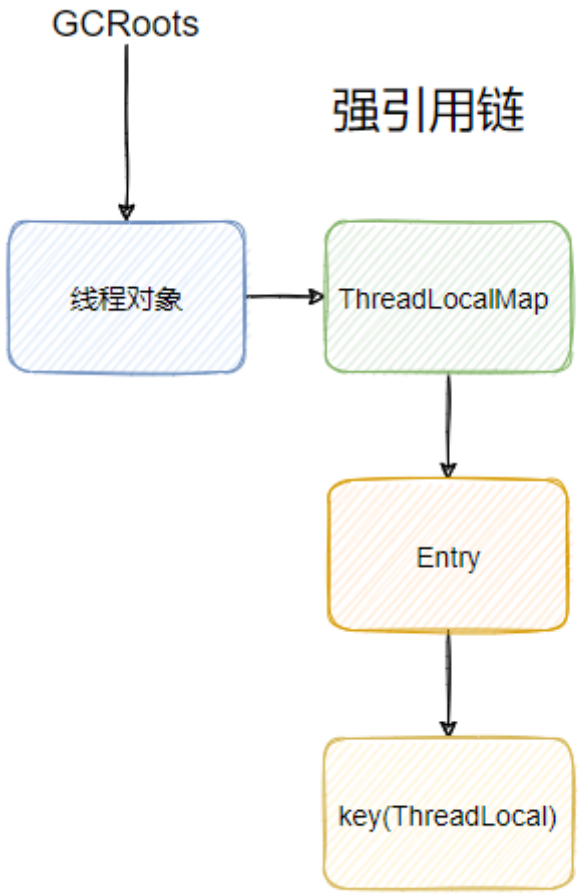
接下来就是要解决上面挖的坑了，即 key 的弱引用、Entry 的 key 为什么可能为 null、还有清理 Entry 的操作。

之前提到过，Entry 对 key 是弱引用，**那为什么要弱引用呢?**

我们知道，如果一个对象没有强引用，**只有弱引用的话**，这个对象是活不过一次 GC 的，所以这样的设计就是为了让当外部没有对 ThreadLocal 对象有强引用的时候，可以将 ThreadLocal 对象给清理掉。

> **那为什么要这样设计呢?**

假设 Entry 对 key 的引用是强引用，那么来看一下这个引用链：



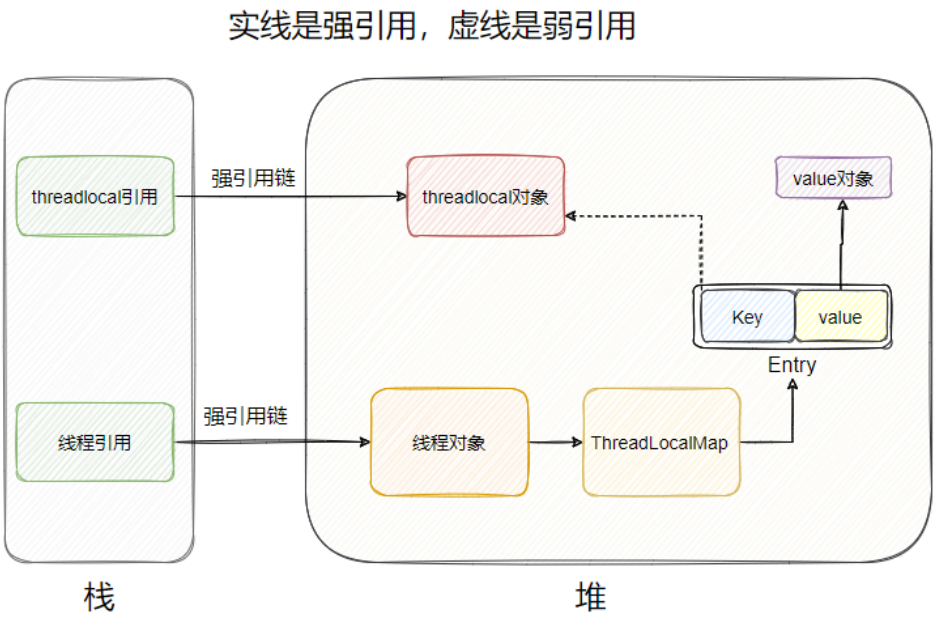
从这条引用链可以得知，如果线程一直在，那么相关的 ThreadLocal 对象肯定会一直在，因为它一直被强引用着。

看到这里，可能有人会说那线程被回收之后就好了呀。

重点来了！**线程在我们应用中，常常是以线程池的方式来使用的**，比如 Tomcat 的线程池处理了一堆请求，而线程池中的线程一般是不会被清理掉的，**所以这个引用链就会一直在**，那么 ThreadLocal 对象即使没有用了，也会随着线程的存在，而一直存在着！

所以这条引用链需要弱化一下，而能操作的只有 Entry 和 key 之间的引用，所以它们之间用弱引用来实现。

与之对应的还有一个条引用链，我结合着上面的线程引用链都画出来：



另一条引用链就是栈上的 ThreadLocal 引用指向堆中的 ThreadLocal 对象，这个引用是强引用。

如果有这条强引用存在，那说明此时的 ThreadLocal 是有用的，此时如果发生 GC 则 ThreadLocal 对象不会被清除，因为有个强引用存在。

当随着方法的执行完毕，相应的栈帧也出栈了，此时这条强引用链就没了，如果没有别的栈有对 ThreadLocal 对象的引用，那么说明 ThreadLocal 对象无法再被访问到(定义成静态变量的另说)。

那此时 ThreadLocal 只存在与 Entry 之间的弱引用，那此时发生 GC 它就可以被清除了，因为它无法被外部使用了，那就等于没用了，是个垃圾，应该被处理来节省空间。

至此，想必你已经明白为什么 Entry 和 key 之间要设计为弱引用，就是因为平日线程的使用方式基本上都是线程池，所以线程的生命周期就很长，可能从你部署上线后一直存在，而 ThreadLocal 对象的生命周期可能没这么长。

所以为了能让已经没用 ThreadLocal 对象得以回收，所以 Entry 和 key 要设计成弱引用，不然 Entry 和 key是强引用的话，ThreadLocal 对象就会一直在内存中存在。

但是这样设计就可能产生内存泄漏。

> **那什么叫内存泄漏?**

就是指：程序中已经无用的内存无法被释放，造成系统内存的浪费。

当 Entry 中的 key 即 ThreadLocal 对象被回收了之后，**会发生 Entry 中 key 为 null 的情况**，其实这个 Entry 就已经没用了，但是又无法被回收，因为有 Thread->ThreadLocalMap ->Entry 这条强引用在，这样没用的内存无法被回收就是内存泄露。

> **那既然会有内存泄漏还这样实现?**

这里就要填一填上面的坑了，也就是涉及到的关于 `expungeStaleEntry` 即清理过期的 Entry 的操作。

设计者当然知道会出现这种情况，所以在多个地方都做了清理无用 Entry ，即 key 已经被回收的 Entry 的操作。

比如通过 key 查找 Entry 的时候，如果下标无法直接命中，那么就会向后遍历数组，此时遇到 key 为 null 的 Entry 就会清理掉，再贴一下这个方法：

```
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;
}
```

遍历的时候清理无用 Entry

这个方法也很简单，我们来看一下它的实现：

```
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) {
            e.value = null;
            tab[i] = null;
            size--;
        } else {
            int h = k.threadLocalHashCode & (len - 1);
            if (h != i) {
                tab[i] = null;

                // Unlike Knuth 6.4 Algorithm R, we must scan until
                // null because multiple entries could have been stale.
                while (tab[h] != null)
                    h = nextIndex(h, len);
                tab[h] = e;
            }
        }
    }
    return i;
}
```

置空操作，且 size --

然后再往后遍历继续清理，
直到遇到 null 才结束

同样也是置空操作

所以在查找 Entry 的时候，就会顺道清理无用的 Entry ，这样就能防止一部分的内存泄露啦！

还有像扩容的时候也会清理无用的 Entry：

```
private void rehash() {
    expungeStaleEntries();

    // Use lower threshold for doubling to avoid hysteresis
    if (size >= threshold - threshold / 4)
        resize();
}
```

其它还有，我就不贴了，反正知晓设计者是做了一些操作来回收无用的 Entry 的即可。

ThreadLocal 的最佳实践

当然，等着这些操作被动回收不是最好的方法，假设后面没人调用 get 或者调用 get 都直接命中或者不会发生扩容，那无用的 Entry 岂不是一直存在了吗？所以上面说只能防止一部分的内存泄露。

所以，最佳实践是用完了之后，调用一下 remove 方法，手工把 Entry 清理掉，这样就不会发生内存泄漏了！

```
void yesDosth {
    threadlocal.set(xxx);
    try {
        // do sth
    } finally {
        threadlocal.remove();
    }
}
```

这就是使用 Threadlocal 的一个正确姿势啦，即不需要的时候，显示的 remove 掉。

当然，如果不是线程池使用方式的话，其实不用关系内存泄漏，反正线程执行完了就都回收了，**但是一般我们都是使用线程池的**，可能只是你没感觉到。

比如你用了 tomcat ，其实请求的执行用的就是 tomcat 的线程池，这就是隐式使用。

还有一个问题，关于 withInitial 也就是初始化值的方法。

由于类似 tomcat 这种隐式线程池的存在，即线程第一次调用执行 Threadlocal 之后，如果没有显示调用 remove 方法，则这个 Entry 还是存在的，那么下次这个线程再执行任务的时候，不会再调用 withInitial 方法，**也就是说会拿到上一次执行的值。**

但是你以为执行任务的是新线程，会初始化值，然而它是线程池里面的老线程，这就和预期不一致了，所以这里需要注意。

InheritableThreadLocal

这个其实之前文章写过了，不过这次竟然写了 threadlocal 就再拿出来。

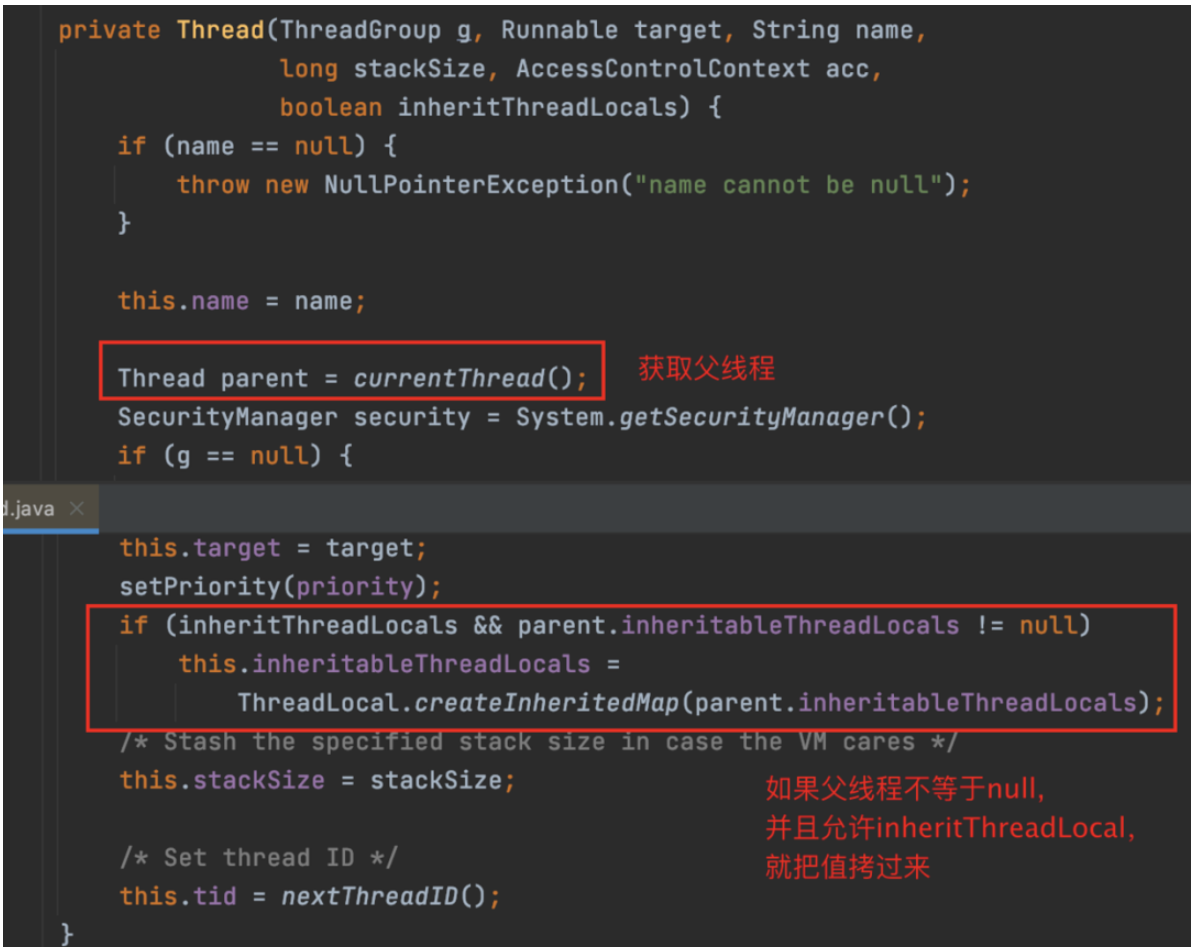
这玩意可以理解为就是可以把父线程的 threadlocal 传递给子线程，所以如果要这样传递就用 InheritableThreadLocal ，不要用 threadlocal。

原理其实很简单，在 Thread 中已经包含了这个成员：

```
/* ThreadLocal values pertaining to this thread. This map is maintained
 * by the ThreadLocal class. */
ThreadLocal.ThreadLocalMap threadLocals = null;

/*
 * InheritableThreadLocal values pertaining to this thread. This map is
 * maintained by the InheritableThreadLocal class.
 */
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
```

在父线程创建子线程的时候，子线程的构造函数可以得到父线程，然后判断下父线程的 InheritableThreadLocal 是否有值，如果有的话就拷过来。



这里要注意，只会在线程创建的时会拷贝 InheritableThreadLocal 的值，之后父线程如何更改，子线程都不会受其影响。

最后

至此有关 ThreadLocal 的知识点就差不多了。

想必你已经清楚 ThreadLocal 的原理，包括如何实现，为什么 key 要设计成弱引用，并且关于在线程池中使用的注意点等等。

其实本没打算写 ThreadLocal 的，因为最近在看 Netty，所以想写一下 FastThreadLocal，但是前置知识点是 ThreadLocal，所以就干了这篇。

消化了这篇之后，出去面试 ThreadLocal 算是没问题了吧，最后再留个小小的思考题。

那为什么 Entry 中的 value 不弱引用?

这个题目来自群友的一个面试题哈，想必看完这篇文章之后，这个题目难不倒你，欢迎留言区写出答案！

等我下篇的 ThreadLocal 进阶版！