[IMG_256](https://www.cnblogs.com/bingxinshuo/)

# [秉心说](https://www.cnblogs.com/bingxinshuo/)

* [博客园](https://www.cnblogs.com/)
* [首页](https://www.cnblogs.com/bingxinshuo/)
* [新随笔](https://i.cnblogs.com/EditPosts.aspx?opt=1)
* [联系](https://msg.cnblogs.com/send/%E7%A7%89%E5%BF%83%E8%AF%B4)
* [订阅](https://www.cnblogs.com/bingxinshuo/rss/)
* [管理](https://i.cnblogs.com/)

# [深入理解 ThreadLocal](https://www.cnblogs.com/bingxinshuo/p/11503601.html)

## **前言**

上篇文章 [深入理解 Handler 消息机制](https://juejin.im/post/5d712cedf265da03ea5a9ecf) 中提到了获取线程的 Looper 是通过 ThreadLocal 来实现的：

public static @Nullable Looper myLooper() {

return sThreadLocal.get();

}

每个线程都有自己的 Looper，它们之间不应该有任何交集，互不干扰，我们把这种变量称为 **线程局部变量** 。而 ThreadLocal 的作用正是存储线程局部变量，每个线程中存储的都是独立存在的数据副本。如果你还是不太理解，看一下下面这个简单的例子：

public static void main(String[] args) throws InterruptedException {

ThreadLocal<Boolean> threadLocal = new ThreadLocal<Boolean>();

threadLocal.set(true);

Thread t1 = new Thread(() -> {

threadLocal.set(false);

System.out.println(threadLocal.get());

});

Thread t2 = new Thread(() -> {

System.out.println(threadLocal.get());

});

t1.start();

t2.start();

t1.join();

t2.join();

System.out.println(threadLocal.get());

}

执行结果是：

falsenulltrue

可以看到，我们在不同的线程中调用同一个 ThreadLocal 的 get() 方法，获得的值是不同的，看起来就像 ThreadLocal 为每个线程分别存储了不同的值。那么这到底是如何实现的呢？一起来看看源码吧。

以下源码基于 [JDK 1.8](https://github.com/lulululbj/jdk8u) , 相关文件：

[Thread.java](https://github.com/lulululbj/jdk8u/blob/master/jdk/src/share/classes/java/lang/Thread.java)

[ThreadLocal.java](https://github.com/lulululbj/jdk8u/blob/master/jdk/src/share/classes/java/lang/ThreadLocal.java)

## **ThreadLocal**

首先 ThreadLocal 是一个泛型类，public class ThreadLocal<T>，支持存储各种数据类型。它对外暴露的方法很少，基本就 get() 、set() 、remove() 这三个。下面依次来看一下。

### **set()**

public void set(T value) {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t); // 获取当前线程的 ThreadLocalMap

if (map != null)

map.set(this, value);

else

createMap(t, value); // 创建 ThreadLocalMap

}

这里出现了一个新东西 ThreadLocalMap，暂且就把他当做一个普通的 Map。从 map.set(this, value) 可以看出来这个 map 的键是 ThreadLocal 对象，值是要存储的 value 对象。其实看到这，ThreadLocal 的原理你应该基本都明白了。

每一个 Thread 都有一个 ThreadLocalMap ，这个 Map 以 ThreadLocal 对象为键，以要保存的线程局部变量为值。这样就做到了为每个线程保存不同的副本。

首先通过 getMap() 函数获取当前线程的 ThreadLocalMap ：

ThreadLocalMap getMap(Thread t) {

return t.threadLocals;

}

原来 Thread 还有这么一个变量 threadLocals ：

/\* ThreadLocal values pertaining to this thread. This map is maintained

\* by the ThreadLocal class.

\*

\* 存储线程私有变量，由 ThreadLocal 进行管理

\*/

ThreadLocal.ThreadLocalMap threadLocals = null;

默认为 null，所以第一次调用时返回 null ，调用 createMap(t, value) 进行初始化：

void createMap(Thread t, T firstValue) {

t.threadLocals = new ThreadLocalMap(this, firstValue);

}

### **get()**

set() 方法是向 ThreadLocalMap 中插值，那么 get() 就是在 ThreadLocalMap 中取值了。

public T get() {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t); // 获取当前线程的 ThreadLocalMap

if (map != null) {

ThreadLocalMap.Entry e = map.getEntry(this);

if (e != null) {

@SuppressWarnings("unchecked")

T result = (T)e.value;

return result; // 找到值，直接返回

}

}

return setInitialValue(); // 设置初始值

}

首先获取 ThreadLocalMap，在 Map 中寻找当前 ThreadLocal 对应的 value 值。如果 Map 为空，或者没有找到 value，则通过 setInitialValue() 函数设置初始值。

private T setInitialValue() {

T value = initialValue(); // 为 null

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

map.set(this, value);

else

createMap(t, value);

return value;

}

protected T initialValue() {

return null;

}

setInitialValue() 和 set() 逻辑基本一致，只不过 value 是 null 而已。这也解释了文章开头的例子会输出 null。当然，在 ThreadLocal 的子类中，我们可以通过重写 setInitialValue() 来提供其他默认值。

### **remove()**

public void remove() {

ThreadLocalMap m = getMap(Thread.currentThread());

if (m != null)

m.remove(this);

}

remove() 就更简单了，根据键直接移除对应条目。

看到这里，ThreadLocal 的原理好像就说完了，其实不然。ThreadLocalMap 是什么样的一个哈希表呢？它是如何解决哈希冲突的？它是如何添加，获取和删除元素的？可能会导致内存泄露吗？

其实 ThreadLocalMap 才是 ThreadLocal 的核心。ThreadLocal 仅仅只是提供给开发者的一个工具而已，就像 Handler 一样。带着上面的问题，来阅读 ThreadLocalMap 的源码，体会 JDK 工程师的鬼斧神工。

## **ThreadLocalMap**

### **Entry**

ThreadLocalMap 是 ThreadLocal 的静态内部类，它没有直接使用 HashMap，而是一个自定义的哈希表，使用数组实现，数组元素是 Entry。

static class Entry extends WeakReference<ThreadLocal<?>> {

/\*\* The value associated with this ThreadLocal. \*/

Object value;

Entry(ThreadLocal<?> k, Object v) {

super(k);

value = v;

}

}

Entry 类继承了 WeakReference<ThreadLocal<?>>，我们可以把它看成是一个键值对。键是当前的 ThreadLocal 对象，值是存储的对象。注意 ThreadLocal 对象的引用是弱引用，值对象 value 的引用是强引用。ThreadLocal 使用弱引用其实很好理解，源码注释中也告诉了我们答案：

To help deal with very large and long-lived usages, the hash table entries use WeakReferences for keys

Thread 持有 ThreadLocalMap 的强引用，ThreadLocalMap 中的 Entry 的键是 ThreadLocal 引用。如果线程长期存活或者使用了线程池，而 ThreadLocal 在外部又没有任何强引用了，这种情况下如果 ThreadLocalMap 的键仍然使用强引用 ThreadLocal，就会导致 ThreadLocal 永远无法被垃圾回收，造成内存泄露。

图片来源：https://www.jianshu.com/p/a1cd61fa22da

那么，使用弱引用是不是就万无一失了呢？答案也是否定的。同样是上面说到使用情况，线程长期存活，由于 Entry 的 key 使用了弱引用，当 ThreadLocal 不存在外部强引用时，可以在 GC 中被回收。但是根据可达性分析算法，仍然存在着这么一个引用链：

Current Thread -> ThreadLocalMap -> Entry -> value

key 已经被回收了，此时 key == null。那么，value 呢？如果线程长期存在，这个针对 value 的强引用也会一直存在，外部是否对 value 指向的对象还存在其他强引用也不得而知。所以这里还是有几率发生内存泄漏的。就算我们不知道外部的引用情况，但至少在这里应该是可以切断 value 引用的。

所以，为了解决可能存在的内存泄露问题，我们有必要对于这种 key 已经被 GC 的过期 Entry 进行处理，手动释放 value 引用。当然，JDK 中已经为我们处理了，而且处理的十分巧妙。下面就来看看 ThreadLocalMap 的源码。

### **构造函数**

ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {

table = new Entry[INITIAL\_CAPACITY];

int i = firstKey.threadLocalHashCode & (INITIAL\_CAPACITY - 1);

table[i] = new Entry(firstKey, firstValue);

size = 1;

setThreshold(INITIAL\_CAPACITY);

}

table 是存储 Entry 的数组，初始容量 INITIAL\_CAPACITY 是 16。

firstKey.threadLocalHashCode & (INITIAL\_CAPACITY - 1) 是 ThreadLocalMap 计算哈希的方式。&(2^n-1) 其实等同于 % 2^n，位运算效率更高。

threadLocalHashCode 是如何计算的呢？看下面的代码：

private static final int HASH\_INCREMENT = 0x61c88647;

private static AtomicInteger nextHashCode = new AtomicInteger();

private final int threadLocalHashCode = nextHashCode();

private static int nextHashCode() {

return nextHashCode.getAndAdd(HASH\_INCREMENT);

}

0x61c88647 是一个增量，每次取哈希都要再加上这个数字。又是一个神奇的数字，让我想到了 Integer 源码中的 52429 这个数字，见 [走进 JDK 之 Integer](https://juejin.im/post/5c76ad1ae51d4572c95835d0" \l "heading-11) 。0x61c88647 背后肯定也有它的数学原理，总之肯定是为了效率。

原理就不去探究了，其实我也不知道是啥原理。不过我们可以试用一下，看看效果如何。按照上面的方式来计算连续几个元素的哈希值，也就是在 Entry 数组中的位置。代码如下：

public class Test {

private static final int INITIAL\_CAPACITY = 16;

private static final int HASH\_INCREMENT = 0x61c88647;

private static AtomicInteger nextHashCode = new AtomicInteger();

private static int nextHashCode() {

return nextHashCode.getAndAdd(HASH\_INCREMENT);

}

private static int hash() {

return nextHashCode() & (INITIAL\_CAPACITY - 1);

}

public static void main(String[] args) {

for (int i = 0; i < 8; i++) {

System.out.println(hash());

}

}

}

运算结果如下：

0

7

14

5

12

3

10

1

计算结果分布还是比较均匀的。既然是哈希表，肯定就会存在哈希冲突的情况。那么，ThreadLocalMap 是如何解决哈希冲突呢？很简单，看一下 nextIndex() 方法。

private static int nextIndex(int i, int len) {

return ((i + 1 < len) ? i + 1 : 0);

}

在不超过 len 的情况下直接加 1，否则置 0。其实这样又可以看成一个环形数组。

接下来看看 ThreadLocalMap 的数据是如何存储的。

### **set()**

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

// We don't use a fast path as with get() because it is at

// least as common to use set() to create new entries as

// it is to replace existing ones, in which case, a fast

// path would fail more often than not.

Entry[] tab = table;

int len = tab.length;

int i = key.threadLocalHashCode & (len-1); // 当前 key 的哈希，即在数组 table 中的位置

for (Entry e = tab[i];

e != null; // 循环直到碰到空 Entry

e = tab[i = nextIndex(i, len)]) {

ThreadLocal<?> k = e.get();

if (k == key) { // 更新 key 对应的值

e.value = value;

return;

}

if (k == null) { // 替代过期 entry

replaceStaleEntry(key, value, i);

return;

}

}

tab[i] = new Entry(key, value);

int sz = ++size;

if (!cleanSomeSlots(i, sz) && sz >= threshold)

rehash();

}

1. 通过 key.threadLocalHashCode & (len-1) 计算出初始的哈希值
2. 不断调用 nextIndex() 直到找到空 Entry

在第二步遍历过程中的每个元素，要处理两种情况：

(1). k == key，说明当前 key 已存在，直接更新值即可，直接返回

1. (2). k == null, 注意这里的前置条件是 entry != null。说明遇到过期 Entry，直接替换
2. 不属于 3 中的两种情况，则将参数中的键值对插入空 Entry 处

cleanSomeSlots()/rehash()

先来看看第三步中的第二种特殊情况。Entry 不为空，但其中的 key 为空，什么时候会发生这种情况呢？对，就是前面说到内存泄漏时提到的 **过期 Entry**。我们都知道 Entry 的 key 是弱引用的 ThreadLocal，当外部没有它的任何强引用时，下次 GC 时就会将其回收。所以这时候的 Entry 理论上也是无效的了。

由于这里是在 set() 方法插入元素的过程中发现了过期 Entry，所以只要将要插入的 Entry 直接替换这个 key==null 的 Entry 就可以了，这就是 replaceStaleEntry() 的核心逻辑。

### **replaceStaleEntry()**

private void replaceStaleEntry(ThreadLocal<?> key, Object value, int staleSlot) {

Entry[] tab = table;

int len = tab.length;

Entry e;

// Back up to check for prior stale entry in current run.

// We clean out whole runs at a time to avoid continual

// incremental rehashing due to garbage collector freeing

// up refs in bunches (i.e., whenever the collector runs).

// 向前找到第一个过期条目

int slotToExpunge = staleSlot;

for (int i = prevIndex(staleSlot, len);

(e = tab[i]) != null;

i = prevIndex(i, len))

if (e.get() == null)

slotToExpunge = i; // 记录前一个过期条目的位置

// Find either the key or trailing null slot of run, whichever occurs first

// 向后查找，直到找到 key 或者 空 Entry

for (int i = nextIndex(staleSlot, len);

(e = tab[i]) != null;

i = nextIndex(i, len)) {

ThreadLocal<?> k = e.get();

// If we find key, then we need to swap it

// with the stale entry to maintain hash table order.

// The newly stale slot, or any other stale slot

// encountered above it, can then be sent to expungeStaleEntry

// to remove or rehash all of the other entries in run.

if (k == key) {

// 如果在向后查找过程中发现 key 相同的 entry 就覆盖并且和过期 entry 进行交换

e.value = value;

tab[i] = tab[staleSlot];

tab[staleSlot] = e;

// Start expunge at preceding stale entry if it exists

// 如果在查找过程中还未发现过期 entry，那么就以当前位置作为 cleanSomeSlots 的起点

if (slotToExpunge == staleSlot)

slotToExpunge = i;

cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);

return;

}

// If we didn't find stale entry on backward scan, the

// first stale entry seen while scanning for key is the

// first still present in the run.

// 如果向前未搜索到过期 entry，而在向后查找过程遇到过期 entry 的话，后面就以此时这个位置

// 作为起点执行 cleanSomeSlots

if (k == null && slotToExpunge == staleSlot)

slotToExpunge = i;

}

// If key not found, put new entry in stale slot

// 如果在查找过程中没有找到可以覆盖的 entry，则将新的 entry 插入在过期 entry

tab[staleSlot].value = null;

tab[staleSlot] = new Entry(key, value);

// If there are any other stale entries in run, expunge them

// 在上面的代码运行过程中，找到了其他的过期条目

if (slotToExpunge != staleSlot)

cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);

}

看起来挺累人的。在我理解，replaceStaleEntry 只是做一个标记的作用，在各种情况下最后都会调用 cleanSomeSlots 来真正的清理过期条目。

你可以看到 ``

### **cleanSomeSlots()**

private boolean cleanSomeSlots(int i, int n) {

boolean removed = false;

Entry[] tab = table;

int len = tab.length;

do {

i = nextIndex(i, len);

Entry e = tab[i];

if (e != null && e.get() == null) {

n = len;

removed = true;

i = expungeStaleEntry(i); // 需要清理的 Entry

}

} while ( (n >>>= 1) != 0);

return removed;

}

参数 n 表示扫描控制。初始情况下扫描 log2(n) 次，如果遇到过期条目，会再扫描 log2(table.length)-1 次。在 set() 方法中调用，参数 n 表示元素的个数。在 replaceStaleEntry 中调用，参数 n 表示的是数组 table 的长度。

注意 do 循环里面的判断条件：e != null && e.get() == null ，还是那些 Entry 不为空，key 为空的过期条目。发现过期条目之后，调用 expungeStaleEntry() 去清理。

### **expungeStaleEntry()**

private int expungeStaleEntry(int staleSlot) {

Entry[] tab = table;

int len = tab.length;

// expunge entry at staleSlot

// 清空 staleSlot 处的 过期 entry

// 将 value 置空，保证不会因为这里的强引用造成 memory leak

tab[staleSlot].value = null;

tab[staleSlot] = null;

size--;

// Rehash until we encounter null

// 继续搜索直到遇到 tab 中的空 entry

Entry e;

int i;

for (i = nextIndex(staleSlot, len);

(e = tab[i]) != null;

i = nextIndex(i, len)) {

ThreadLocal<?> k = e.get();

if (k == null) { // 搜索过程中遇到过期条目，直接清理

e.value = null;

tab[i] = null;

size--;

} else {

// key 还没有被回收

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; // 此时从 staleSlot 到 i 之间不存在过期条目

}

直接将 entry.value 和 entry 都置空，消除内存泄露的隐患。注意这里仅仅只是置空，并不是回收对象。因为你不知道 value 在外部的引用情况，只需要管好自己的引用就可以了。

除此之外，不甘寂寞的 expungeStaleEntry() 又发起了一次扫描，直到碰到空 Entry未知。期间遇到的过期 Entry 要置空。

整个 set() 方法就看完了，原理很简单，但是其中关于内存泄漏的预防处理十分复杂，看的我一度放弃了，也让我对源码阅读产生了一些疑问。有些时候是不是没有必要逐行去玩去完全理解？比如这一系列关于内存泄露的处理，核心思想就是清理 Entry 不为 null 但 key 为 null 的过期条目。理解了核心思想，对于其中复杂的细节处理是不是没有必要去深究？不知道你怎么看，欢迎在评论区写下你的看法。

下面来看一看 getgetEntry 方法。

### **getEntry()**

private Entry getEntry(ThreadLocal<?> key) {

int i = key.threadLocalHashCode & (table.length - 1);

Entry e = table[i];

if (e != null && e.get() == key) // 直接命中

return e;

else

// 未直接命中，线性探测，继续往后找

return getEntryAfterMiss(key, i, e);

}

getEntry() 比较粗暴，上来直接根据哈希值查找 table 数组，如果直接命中，就返回。未直接命中，调用 getEntryAfterMiss() 继续查找。

private Entry getEntryAfterMiss(ThreadLocal<?> key, int i, Entry e) {

Entry[] tab = table;

int len = tab.length;

// 向后查找直到遇到空 entry

while (e != null) {

ThreadLocal<?> k = e.get();

if (k == key) // get it

return e;

if (k == null) // key 等于 null，清理过期 entry

expungeStaleEntry(i);

else

i = nextIndex(i, len); // 继续向后查找

e = tab[i];

}

return null;

}

调用 nextIndex() 向后查找，直到遇到 空 Entry，也就是队尾：

* k==key，说明找到了对应 Entry
* k==null，说明遇到了过期 Entry，调用 expungeStaleEntry() 处理

对过期 Entry 的处理真的是无处不在，就是为了最大程度的降低内存泄漏发生的几率。那么有没有什么一劳永逸的办法呢？那就是 ThreadLocalMap 的 remove() 方法。

### **remove()**

private void remove(ThreadLocal<?> key) {

Entry[] tab = table;

int len = tab.length;

int i = key.threadLocalHashCode & (len-1);

for (Entry e = tab[i];

e != null;

e = tab[i = nextIndex(i, len)]) {

if (e.get() == key) {

e.clear();

expungeStaleEntry(i);

return;

}

}

}

直接清除当前 ThreadLocal 对应的 Entry，根本上避免了发生内存泄露。所以，当我们不再需要使用 ThreadLocal 中的相应数据时，调用一下 remove() 方法肯定是个好习惯。

虽然在长期存活的线程（例如线程池）中使用 ThreadLocal 并发生内存泄漏是一个小概率事件，但 JDK 开发者却为此多写了很多代码。我们在使用中也要多加注意，仔细考虑是否会涉及到内存泄露的问题。

## **End**

最后说说在网上看到的一个观点，**ThreadLocal 比 Synchronized 更适合解决线程同步问题。**

首先这个问题本身就不是那么严谨。ThreadLocal 是用来解决线程同步问题的吗？表面上看，ThreadLocal 的机制的确是线程安全的，但它并不是为了解决多线程访问同一个变量的竞争问题，而是给每一个线程都提供单独的变量，有些文章称之为 **数据备份**，但它们并不是备份，每一个都是独立存在的，互不干扰，并不存在什么同步问题。

ThreadLocal 和 Synchronized 的应用场景也是千差万别的。例如银行的转账场景，涉及多个账户同时转账的多线程同步问题，ThreadLocal 根本就没法解决，即使每个线程都单独保存着用户的余额也没法解决并发问题。ThreadLocal 在 Android 中的典型应用就是 Looper，每个线程都有自己的 Looper 对象，它们都是独立工作，互不干扰的。

关于 ThreadLocal 就说到这里了。后续分享的方向主要集中在两块，一方面是 AOSP 源码的阅读和解析，另一方面是 Kotlin 和 Java 相关特性的对比，敬请期待！

文章首发微信公众号： 秉心说 ， 专注 Java 、 Android 原创知识分享，LeetCode 题解。

更多最新原创文章，扫码关注我吧！



**标签:**[java](https://www.cnblogs.com/bingxinshuo/tag/java/)**,**[ThreadLocal](https://www.cnblogs.com/bingxinshuo/tag/ThreadLocal/)

**[好文要顶](javascript:void(0);)** **[关注我](javascript:void(0);)** **[收藏该文](javascript:void(0);)** **[IMG_258](javascript:void(0);)** **[IMG_259](javascript:void(0);)**

[秉心说](https://home.cnblogs.com/u/bingxinshuo/)  
[关注 - 3](https://home.cnblogs.com/u/bingxinshuo/followees/)  
[粉丝 - 2](https://home.cnblogs.com/u/bingxinshuo/followers/)

[+加关注](javascript:void(0);)

1

0

[«](https://www.cnblogs.com/bingxinshuo/p/11470573.html)上一篇： [深入理解 Handler 消息机制](https://www.cnblogs.com/bingxinshuo/p/11470573.html" \o "发布于 2019-09-05 23:43)  
[»](https://www.cnblogs.com/bingxinshuo/p/11546151.html)下一篇： [Null is your friend, not a mistake](https://www.cnblogs.com/bingxinshuo/p/11546151.html" \o "发布于 2019-09-18 22:50)

posted @ 2019-09-10 22:44  [秉心说](https://www.cnblogs.com/bingxinshuo/)  阅读(542)  评论(0)  [编辑](https://i.cnblogs.com/EditPosts.aspx?postid=11503601)  [收藏](javascript:void(0))

[刷新评论](javascript:void(0);)[刷新页面](https://www.cnblogs.com/bingxinshuo/p/11503601.html)[返回顶部](https://www.cnblogs.com/bingxinshuo/p/11503601.html" \l "top)

**注册用户登录后才能发表评论，请 [登录](javascript:void(0);) 或 [注册](javascript:void(0);)， [访问](https://www.cnblogs.com/) 网站首页。**

[【推荐】超50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库](http://www.ucancode.com/index.htm" \t "/Users/liu123/Documents\\x/_blank)  
[【推荐】阿里云双11返场来袭，热门产品低至一折等你来抢！](http://click.aliyun.com/m/1000081987/" \t "/Users/liu123/Documents\\x/_blank)  
[【活动】开发者上云必备，腾讯云1核4G 2M云服务器11元/月起](https://cloud.tencent.com/act/developer?fromSource=gwzcw.3196335.3196335.3196335&utm_medium=cpc&utm_id=gwzcw.3196335.3196335.3196335" \t "/Users/liu123/Documents\\x/_blank)  
[【推荐】百度智能云岁末感恩季，明星产品低至1元新老用户畅享](https://cloud.baidu.com/campaign/Promotion-20191111/index.html?track=cp:dsp|pf:pc|pp:chui-bokeyuan-huodong-19shuangshiyiganenji-BCC-cpaxingshi-191210|pu:cpa-xingshi|ci:2019syj|kw:2172212" \t "/Users/liu123/Documents\\x/_blank)  
[【活动】京东云限时优惠1.5折购云主机，最高返价值1000元礼品！](https://www.jdcloud.com/cn/activity/newUser?utm_source=DMT_cnblogs&utm_medium=CH&utm_campaign=09vm&utm_term=Virtual-Machines" \t "/Users/liu123/Documents\\x/_blank)  
[【推荐】大数据技术之flink百问，阿里云开发者社区一线收集、吐血整理](https://developer.aliyun.com/ask/258515?utm_content=g_1000095381" \t "/Users/liu123/Documents\\x/_blank)  
[【推荐】史上最全python数据爬取面试问答合集 | 开发者福利](https://developer.aliyun.com/ask/257367?utm_content=g_1000095382" \t "/Users/liu123/Documents\\x/_blank)

**相关博文：**  
· [ThreadLocal原理深入解析](https://www.cnblogs.com/takumicx/p/9320881.html" \o "ThreadLocal原理深入解析" \t "/Users/liu123/Documents\\x/_blank)  
· [【Java】深入理解ThreadLocal](https://www.cnblogs.com/lqminn/p/3751206.html" \o "【Java】深入理解ThreadLocal" \t "/Users/liu123/Documents\\x/_blank)  
· [彻底理解ThreadLocal](https://www.cnblogs.com/xzwblog/p/7227509.html" \o "彻底理解ThreadLocal" \t "/Users/liu123/Documents\\x/_blank)  
· [Android开发之ThreadLocal原理深入理解](https://www.cnblogs.com/alex-mh/p/6761233.html" \o "Android开发之ThreadLocal原理深入理解" \t "/Users/liu123/Documents\\x/_blank)  
· [Java并发编程：深入剖析ThreadLocal](https://www.cnblogs.com/dolphin0520/p/3920407.html" \o "Java并发编程：深入剖析ThreadLocal" \t "/Users/liu123/Documents\\x/_blank)  
» [更多推荐...](https://recomm.cnblogs.com/blogpost/11503601" \t "/Users/liu123/Documents\\x/_blank)

[精品问答：大数据常见问题之 flink 五十问](https://developer.aliyun.com/ask/258515?utm_content=g_1000088951" \t "/Users/liu123/Documents\\x/_blank)

**最新 IT 新闻**:  
· [世界级难题“三体”出现曙光！Nature：牛顿350年前遗留之谜迎新解](https://news.cnblogs.com/n/653691/" \t "/Users/liu123/Documents\\x/_blank)  
· [5年股价翻13倍！AMD是怎样一年大翻身的？](https://news.cnblogs.com/n/653690/" \t "/Users/liu123/Documents\\x/_blank)  
· [缺乏管理职业化，中国的企业管理SaaS很难做](https://news.cnblogs.com/n/653689/" \t "/Users/liu123/Documents\\x/_blank)  
· [计算中的上帝：全球首个“活体机器人”研究团队在线答疑，无须恐慌！](https://news.cnblogs.com/n/653688/" \t "/Users/liu123/Documents\\x/_blank)  
· [暗中观察，没有「头环」：AI摄像头就可以看出你上课是否走神](https://news.cnblogs.com/n/653687/" \t "/Users/liu123/Documents\\x/_blank)  
» [更多新闻...](https://news.cnblogs.com/" \o "IT 新闻" \t "/Users/liu123/Documents\\x/_blank)

### 公告

昵称： [秉心说](https://home.cnblogs.com/u/bingxinshuo/)  
园龄： [4年4个月](https://home.cnblogs.com/u/bingxinshuo/" \o "入园时间：2015-08-19)  
粉丝： [2](https://home.cnblogs.com/u/bingxinshuo/followers/)  
关注： [3](https://home.cnblogs.com/u/bingxinshuo/followees/)

[+加关注](javascript:void(0))

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | [<](javascript:void(0);) | 2020年1月 | [>](javascript:void(0);) | | | | | | | |
| **日** | **一** | **二** | **三** | **四** | **五** | **六** |
| 29 | 30 | 31 | 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 | 1 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |

### 搜索

### 我的标签

* [aosp](https://www.cnblogs.com/bingxinshuo/tag/aosp/)(5)
* [Jetpack Compose](https://www.cnblogs.com/bingxinshuo/tag/Jetpack Compose/)(4)
* [kotlin](https://www.cnblogs.com/bingxinshuo/tag/kotlin/)(2)
* [android](https://www.cnblogs.com/bingxinshuo/tag/android/)(2)
* [Android 9.0](https://www.cnblogs.com/bingxinshuo/tag/Android 9.0/)(1)
* [ANR](https://www.cnblogs.com/bingxinshuo/tag/ANR/)(1)
* [ActivityManagerService](https://www.cnblogs.com/bingxinshuo/tag/ActivityManagerService/)(1)
* [Box](https://www.cnblogs.com/bingxinshuo/tag/Box/)(1)
* [Corotines](https://www.cnblogs.com/bingxinshuo/tag/Corotines/)(1)
* [handler](https://www.cnblogs.com/bingxinshuo/tag/handler/)(1)
* [更多](https://www.cnblogs.com/bingxinshuo/tag/)

### 随笔档案

* [2019年12月(5)](https://www.cnblogs.com/bingxinshuo/archive/2019/12.html" \t "/Users/liu123/Documents\\x/)
* [2019年11月(2)](https://www.cnblogs.com/bingxinshuo/archive/2019/11.html" \t "/Users/liu123/Documents\\x/)
* [2019年10月(5)](https://www.cnblogs.com/bingxinshuo/archive/2019/10.html" \t "/Users/liu123/Documents\\x/)
* [2019年9月(4)](https://www.cnblogs.com/bingxinshuo/archive/2019/09.html" \t "/Users/liu123/Documents\\x/)
* [2019年8月(1)](https://www.cnblogs.com/bingxinshuo/archive/2019/08.html" \t "/Users/liu123/Documents\\x/)

### 最新评论

* [1. Re:秉心说，不一样的 2019](https://www.cnblogs.com/bingxinshuo/p/12052126.html" \l "4462716)
* 人生赢家
* --getEleven
* [2. Re:秉心说，不一样的 2019](https://www.cnblogs.com/bingxinshuo/p/12052126.html" \l "4461200)
* 666
* --滴答的雨
* [3. Re:Android 世界中，谁喊醒了 Zygote ？](https://www.cnblogs.com/bingxinshuo/p/11681943.html" \l "4392496)
* 支持支持
* --牛腩
* [4. Re:Null is your friend, not a mistake](https://www.cnblogs.com/bingxinshuo/p/11546151.html" \l "4363571)
* @ 可少kotlin 的类型系统相比较于 Java 确实更加健壮一些。...
* --秉心说
* [5. Re:Null is your friend, not a mistake](https://www.cnblogs.com/bingxinshuo/p/11546151.html" \l "4363005)
* 吗的 居然是Kotlin 的广告
* --可少

### 阅读排行榜

* [1. 如何正确的在 Android 上使用协程 ？(593)](https://www.cnblogs.com/bingxinshuo/p/11717209.html)
* [2. 深入理解 ThreadLocal(538)](https://www.cnblogs.com/bingxinshuo/p/11503601.html)
* [3. 秉心说，不一样的 2019(495)](https://www.cnblogs.com/bingxinshuo/p/12052126.html)
* [4. 细数 SharedPreferences 的那些槽点 ！(353)](https://www.cnblogs.com/bingxinshuo/p/11427208.html)
* [5. Java 世界的盘古和女娲 —— Zygote(336)](https://www.cnblogs.com/bingxinshuo/p/11605340.html)

### 评论排行榜

* [1. 深入理解 Handler 消息机制(2)](https://www.cnblogs.com/bingxinshuo/p/11470573.html)
* [2. Null is your friend, not a mistake(2)](https://www.cnblogs.com/bingxinshuo/p/11546151.html)
* [3. 秉心说，不一样的 2019(2)](https://www.cnblogs.com/bingxinshuo/p/12052126.html)
* [4. Android 世界中，谁喊醒了 Zygote ？(1)](https://www.cnblogs.com/bingxinshuo/p/11681943.html)

### 推荐排行榜

* [1. 秉心说，不一样的 2019(3)](https://www.cnblogs.com/bingxinshuo/p/12052126.html)
* [2. Java 世界的盘古和女娲 —— Zygote(2)](https://www.cnblogs.com/bingxinshuo/p/11605340.html)
* [3. Zygote家的大儿子 —— SystemServer(1)](https://www.cnblogs.com/bingxinshuo/p/11669031.html)
* [4. Android 世界中，谁喊醒了 Zygote ？(1)](https://www.cnblogs.com/bingxinshuo/p/11681943.html)
* [5. 深入理解 ThreadLocal(1)](https://www.cnblogs.com/bingxinshuo/p/11503601.html)

Copyright © 2020 秉心说  
Powered by .NET Core 3.1.0 on Linux