

## @来源 [原来这就是比 ThreadLocal 更快的玩意](#)

你好，我是yes。

继上一篇之后我把 [ThreadLocal 能问的，都写了](#)，咱们再来盘一盘 FastThreadLocal，这个算是 ThreadLocal 的进阶版，是 Netty 针对 ThreadLocal 自己造的轮子，所以对 ThreadLocal 没有完全理解的话，建议先看上一篇文章，打个基础。

那了解 FastThreadLocal 之后呢，对平日的一些优化可能可以提供一些思路，或者面试就能装个x。

面试官：ThreadLocal 竟然有xxx这个缺点，那怎么优化啊？

你就把 FastThreadLocal 的实现 BB 一遍，这不就稳妥了嘛！

所以，今天我们就来看看 Netty 是如何实现 FastThreadLocal 的，话不多说，本文大纲如下：

- 数数 ThreadLocal 的缺点。
- 应该如何针对 ThreadLocal 缺点改进？
- FastThreadLocal 的原理。
- FastThreadLocal VS ThreadLocal 的实操。

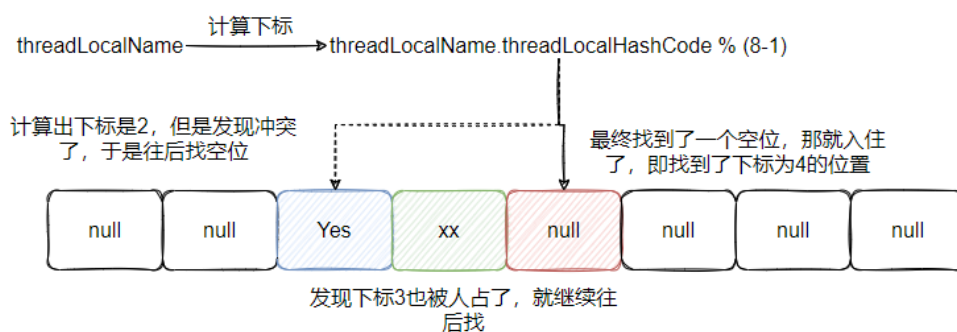
这篇下来，进阶版 ThreadLocal 基本拿下，下篇我会基于这篇做一个延伸，一个比较底层的延伸，属于绝对装x的那种，等下看文章你就知道了，我会埋坑的，哈哈。

预告一下，这篇是个长文，源码也有点多，但是耐心看完肯定会有收获的。

发车发车！

### 数数 ThreadLocal 的缺点

看完上篇文章的同学，应该都很清楚了 ThreadLocal 的一个缺点：hash 冲突用的是线性探测法，效率低。



可以看到，图上显示的是经过两个遍历找到了空位，假设冲突多了，需要遍历的次数就多了。并且下次 get 的时候，hash 直接命中的位置发现不是要找的 Entry，于是就接着遍历向后找，所以说这个效率低。

而像 HashMap 是通过链表法来解决冲突，并且为了防止链表过长遍历的开销变大，在一定条件之后又会转变成红黑树来查找，这样的解决方案在频繁冲突的条件下，肯定是优于线性探测法，所以这是一个优化方向。

不过 FastThreadLocal 不是这样优化的，我们下面再说。

还有一个缺点是 ThreadLocal 使用了 WeakReference 以保证资源可以被释放，但是这可能会产生一些 Entry 的 key 为 null，即无用的 Entry 存在。

所以调用 ThreadLocal 的 get 或 set 方法时，会主动清理无用的 Entry，减轻内存泄漏的发生。

```
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;
}
```

会主动清理无用内存

这其实等于把清理的开销弄到了 get 和 set 上，万一 get 的时候清理的无用 Entry 特别多，那这次 get 相对而言就比较慢了。

还有一个就是内存泄漏的问题了，当然这个问题只存在于用线程池使用的时候，并且上面也提到了 get 和 set 的时候也能清理一些无用的 Key，所以没有那么的夸张，只要记得用完后调用 ThreadLocal#remove 就不会有内存泄漏的问题了。

大致就这么几点。

## 应该如何针对 ThreadLocal 缺点改进

所以怎么改呢？

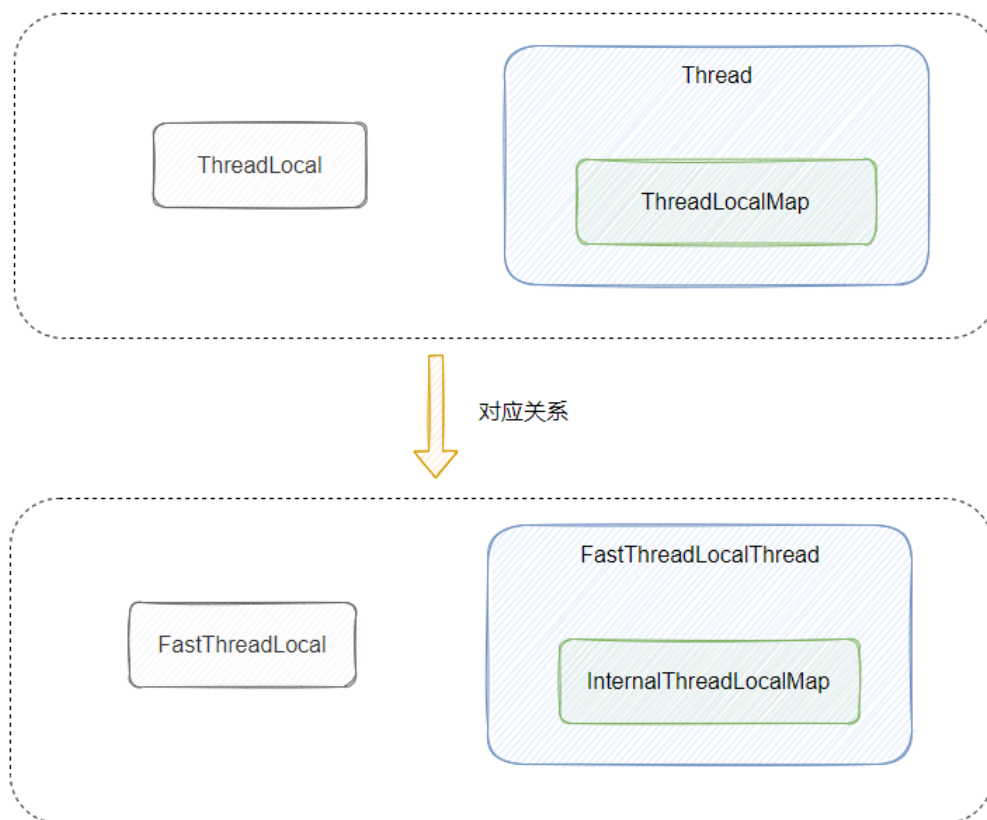
前面提到 ThreadLocal hash 冲突的线性探测法不好，还有 Entry 的弱引用可能会发生内存泄漏，这些都和 ThreadLocalMap 有关，所以需要搞个新的 map 来替换 ThreadLocalMap。

而这个 ThreadLocalMap 又是 Thread 里面的一个成员变量，这么一看 Thread 也得动一动，但是我们无法修改 Thread 的代码，所以配套的还得弄个新的 Thread。

所以我们不仅得弄个新的 ThreadLocal、ThreadLocalMap 还得弄个配套的 Thread 来用上新的 ThreadLocalMap。

所以如果想改进 ThreadLocal，就需要动这三个类。

对应到 Netty 的实现就是 FastThreadLocal、InternalThreadLocalMap、FastThreadLocalThread



然后发散一下思维，既然 Hash 冲突的想线性探测效果不好，你可能比较容易想到的就是上面提到的链表法，然后再基于链表法说个改成红黑树，这个确实是一方面，但是可以再想想。

比如，让 Hash 不冲突，所以设计一个不会冲突的 hash 算法？不存在的！

所以怎么样才不会产生冲突呢？

### 各自取号入座

什么意思？就是每往 InternalThreadLocalMap 中塞入一个新的 FastThreadLocal 对象，就给这个对象发个唯一的下标，然后让这个对象记住这个下标，到时候去 InternalThreadLocalMap 找 value 的时候，直接通过下标去取对应的 value。

这样不就不会冲突了？

这就是 FastThreadLocal 给出的方案，具体下面分析。

还有个内存泄漏的问题，这个其实只要规范的使用即用完后 remove 就好了，其实也没太好的解决方案，不过 FastThreadLocal 曲线救国了一下，这个也且看下面的分析！

## FastThreadLocal 的原理

以下 Netty 基于 4.1 版本分析

先来看下 FastThreadLocal 的定义：

```
public class FastThreadLocal<V> {  
    private static final int variablesToRemoveIndex = InternalThreadLocalMap.nextVariableIndex();  
    private final int index;  
    public FastThreadLocal() {  
        index = InternalThreadLocalMap.nextVariableIndex();  
    }  
}
```

可以看到有个叫 `variablesToRemoveIndex` 的类成员，并且用 `final` 修饰的，所以等于每个 `FastThreadLocal` 都有个共同的不可变 `int` 值，值为多少等下分析。

然后看到这个 `index` 没，在 `FastThreadLocal` 构造的时候就被赋值了，且也被 `final` 修饰，所以也不可  
变，**这个 `index` 就是我上面说的给每个新 `FastThreadLocal` 都发个唯一的下标**，这样每个 `index` 就都知道自己的位置了。

上面两个 `index` 都是通过 `InternalThreadLocalMap.nextVariableIndex()` 赋值的，盲猜一下，这个肯定是用原子类递增实现的。

我们来看一下实现：

```
private static final AtomicInteger nextIndex = new AtomicInteger();

public static int nextVariableIndex() {
    int index = nextIndex.getAndIncrement();
    if (index < 0) {
        nextIndex.decrementAndGet();
        throw new IllegalStateException("too many thread-local indexed variables");
    }
    return index;
}
```

确实，在 `InternalThreadLocalMap` 也定义了一个静态原子类，每次调用 `nextVariableIndex` 就返回且递增，没有什么别的赋值操作，从这里也可以得知 `variablesToRemoveIndex` 的值为 0，因为它属于常量赋值，第一次调用时 `nextIndex` 的值为 0。

看到这，不知道大家是否已经感觉到一丝不对劲了。好像有点浪费空间的意思，我们继续往下看。

`InternalThreadLocalMap` 对标的就是之前的 `ThreadLocalMap` 也就是 `ThreadLocal` 缺点集中的类，需要重点看下。

我们再回顾一下 `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. ... */
    private Entry[] table;

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

它是个 `Entry` 数组，然后 `Entry` 里面弱引用了 `ThreadLocal` 作为 `Key`。

而 InternalThreadLocalMap 有点不太一样：

```
public final class InternalThreadLocalMap extends UnpaddedInternalThreadLocalMap {  
  
    private static final ThreadLocal<InternalThreadLocalMap> slowThreadLocalMap = 这个等下分析  
        new ThreadLocal<>();  
  
    private static final int INDEXED_VARIABLE_TABLE_INITIAL_SIZE = 32;  
  
    public static final Object UNSET = new Object();  
  
    /** Used by {@link FastThreadLocal} */  
    private Object[] indexedVariables; 这里是通过 Object 数组来实现的  
  
    private InternalThreadLocalMap() {  
        indexedVariables = newIndexedVariableTable();  
    }  
  
    private static Object[] newIndexedVariableTable() { 构造默认生成32长度的数组，并且填充默认值  
        Object[] array = new Object[INDEXED_VARIABLE_TABLE_INITIAL_SIZE];  
        Arrays.fill(array, UNSET);  
        return array;  
    }  
}
```

可以看到，InternalThreadLocalMap 好像放弃了 map 的形式，没用定义 key 和 value，而是一个 Object 数组？

那它是如何通过 Object 来存储 FastThreadLocal 和对应的 value 的呢？我们从 FastThreadLocal#set 开始分析：

```
public final void set(V value) {  
    if (value != InternalThreadLocalMap.UNSET) { 如果要塞入的值不是默认值  
        InternalThreadLocalMap threadLocalMap = InternalThreadLocalMap.get(); 这个肯定是从线程对象  
        setKnownNotUnset(threadLocalMap, value); 里面拿到 map  
    } else {  
        remove(); 如果是默认值，则执行删除操作，这个等下在看  
    }  
}  
  
/** @see InternalThreadLocalMap#setIndexedVariable(int, Object). ... */  
private void setKnownNotUnset(InternalThreadLocalMap threadLocalMap, V value) {  
    if (threadLocalMap.setIndexedVariable(index, value)) { index就是构造时被分配得到的下标  
        addToVariablesToRemove(threadLocalMap, variable: this);  
    }  
    从名字来看，添加到待删除列表？ this就是 fastThreadlocal 对象  
}
```

因为我们已经熟悉 ThreadLocal 的套路，所以我们知道 InternalThreadLocalMap 肯定是 FastThreadLocalThread 里面的一个变量。

然后我们从对应的 FastThreadLocalThread 里面拿到了 map 之后，就要执行塞入操作即 setKnownNotUnset。

我们先看一下塞入操作里面的 setIndexedVariable 方法：

```

public boolean setIndexedVariable(int index, Object value) {
    Object[] lookup = indexedVariables; 前面看到的那个 Object 数组
    if (index < lookup.length) {
        Object oldValue = lookup[index]; 直接通过索引找到位置，进行替换
        lookup[index] = value;
        return oldValue == UNSET;
    } else {
        expandIndexedVariableTableAndSet(index, value); 如果index大于数组 size，
        return true; 进行扩容操作
    }
}

```

可以看到，根据传入构造 FastThreadLocal 生成的唯一 index 可以直接从 Object 数组里面找到下标并且进行替换，这样一来压根就不会产生冲突，逻辑很简单，完美。

那如果塞入的 value 不是 UNSET(默认值)，则执行 `addToVariablesToRemove` 方法，这个方法又有什么用呢？

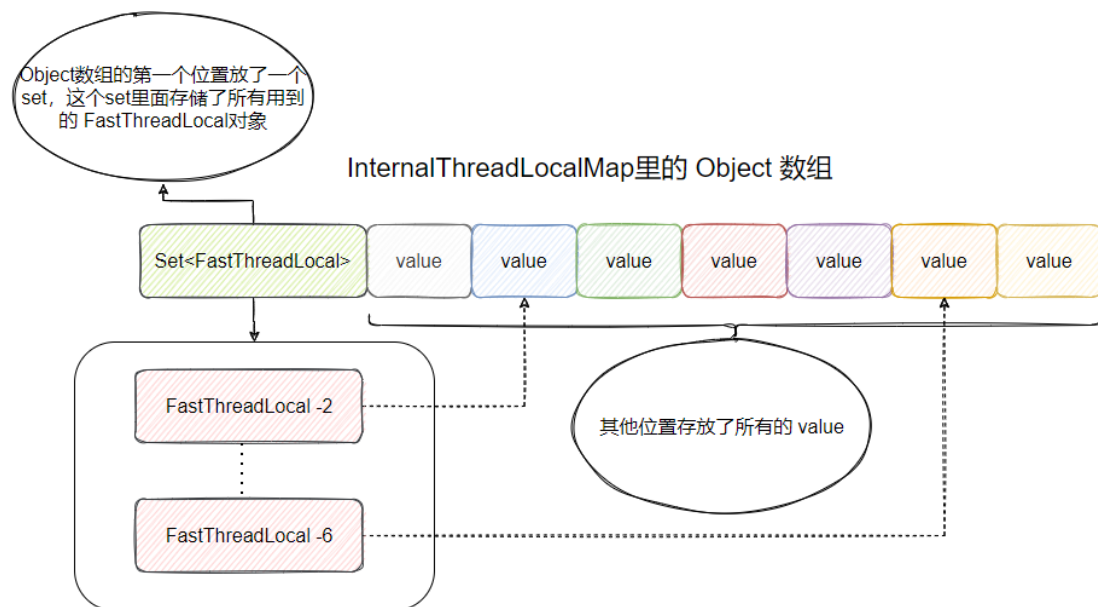
```

@SuppressWarnings("unchecked")
private static void addToVariablesToRemove(InternalThreadLocalMap threadLocalMap, FastThreadLocal<?> variable) {
    Object v = threadLocalMap.indexedVariable(variablesToRemoveIndex); index是上面分析的常量，值为 0，
    Set<FastThreadLocal<?>> variablesToRemove; 所以从数组的 0 下标处获取一个 v
    if (v == InternalThreadLocalMap.UNSET || v == null) { 如果v不是默认值，且 v == null
        variablesToRemove = Collections.newSetFromMap(new IdentityHashMap<FastThreadLocal<?>, Boolean>()); 构建一个 set
        threadLocalMap.setIndexedVariable(variablesToRemoveIndex, variablesToRemove); 将这个 set 塞到 Object 数组的
    } else { 第 0 个位置
        variablesToRemove = (Set<FastThreadLocal<?>>) v; 反之，将 v 强转成 set
    }

    variablesToRemove.add(variable); 将传入的 FastThreadLocal 塞到这个 set 中
}

```

是不是看着有点奇怪？这是啥操作？别急，看我画个图来解释解释：



这就是 Object 数组的核心关系图了，第一个位置放了一个 set，set 里面存储了所有使用的 FastThreadLocal 对象，然后数组后面的位置都放 value。

那为什么要放一个 set 保存所有使用的 FastThreadLocal 对象？

**用于删除**，你想想看，假设现在要清空线程里面的所有 FastThreadLocal，那必然得有一个地方来存放这些 FastThreadLocal 对象，这样才能找到这些家伙，然后干掉。



所以刚好就把数组的第一个位置腾出来放一个 set 来保存这些 FastThreadLocal 对象，如果要删除全部 FastThreadLocal 对象的时候，只需要遍历这个 set，得到 FastThreadLocal 的 index 找到数组对应的位置将 value 置空，然后把 FastThreadLocal 从 set 中移除即可。

刚好 FastThreadLocal 里面实现了这个方法，我们来看下：

```
public static void removeAll() {
    InternalThreadLocalMap threadLocalMap = InternalThreadLocalMap.getIfSet();
    if (threadLocalMap == null) {
        return;
    }

    try {    得到 v，v 就是那个set
        Object v = threadLocalMap.indexedVariable(variablesToRemoveIndex);
        if (v != null && v != InternalThreadLocalMap.UNSET) {
            @SuppressWarnings("unchecked")
            Set<FastThreadLocal<?>> variablesToRemove = (Set<FastThreadLocal<?>>) v;
            FastThreadLocal<?>[] variablesToRemoveArray = 将 set 转成数组遍历
                variablesToRemove.toArray(new FastThreadLocal[0]);
            for (FastThreadLocal<?> tlv: variablesToRemoveArray) {
                tlv.remove(threadLocalMap); 分别调用 remove
            }
        }
    } finally {
        InternalThreadLocalMap.remove(); 这个是将线程里面的 map 置空，完成整体的移除
    }
}
```

内容可能有点多了，我们做下小结，理一理上面说的：

首先 InternalThreadLocalMap 没有采用 ThreadLocalMap k-v形式的存储方式，而是用 Object 数组来存储 FastThreadLocal 对象和其 value，具体是在第一个位置存放了一个包含所使用的 FastThreadLocal 对象的 set，然后后面存储所有的 value。

之所以需要个 set 是为了存储所有使用的 FastThreadLocal 对象，这样就能找到这些对象，便于后面的删除工作。

之所以数组其他位置可以直接存储 value，是因为每个 FastThreadLocal 构造的时候已经被分配了一个唯一的下标，这个下标对应的就是 value 所处的下标。

看到这里，不知道大家是否有感受到空间的浪费？

我举个例子。

假设系统里面一个 new 了 100 个 FastThreadLocal，那第 100 个 FastThreadLocal 的下标就是 100，这个应该没有疑义。

从上面的 set 方法可以得知，只有调用 set 的时候，才会从当前线程中拿出 InternalThreadLocalMap，然后往这个 map 的数组里面塞入 value，这里我们再回顾一下 set 的方法。

```
public boolean setIndexedVariable(int index, Object value) {
    Object[] lookup = indexedVariables; Object 数组
    if (index < lookup.length) { 重点：默认数组是32，如果index大于数组
        Object oldValue = lookup[index];
        lookup[index] = value;
        return oldValue == UNSET;
    } else {
        expandIndexedVariableTableAndSet(index, value); 就会进行扩容
        return true;
    }
}
```

那这里是什么意思呢？

如果我这个线程之前都没塞过 FastThreadLocal，此时要塞入**第一个** FastThreadLocal，构造出来的数组长度是32，但是这个 FastThreadLocal 的下标已经涨到了 100 了，所以**这个线程第一次塞值，也仅仅只有这么一个值，数组就需要扩容。**

看到没，这就是我所说的浪费，空间被浪费了。

Netty 相关实现者知道这样会浪费空间，**所以数组的扩容是基于 index 而不是原先数组的大小**，你看看如果是基于原先数组的扩容，那么第一次扩容 2 倍，32 变成 64，还是塞不下下标 100 的数据，所以还得扩容一次，这就不美了。

所以可以看到扩容传进去的参数是 index。

```
private void expandIndexedVariableTableAndSet(int index, Object value) {
    Object[] oldArray = indexedVariables;
    final int oldCapacity = oldArray.length;
    int newCapacity = index;
    newCapacity |= newCapacity >>> 1;
    newCapacity |= newCapacity >>> 2;
    newCapacity |= newCapacity >>> 4;
    newCapacity |= newCapacity >>> 8;
    newCapacity |= newCapacity >>> 16;
    newCapacity++;
    Object[] newArray = Arrays.copyOf(oldArray, newCapacity);
    Arrays.fill(newArray, oldCapacity, newArray.length, UNSET);
    newArray[index] = value;
    indexedVariables = newArray;
}
```

直接数组拷贝，不需要rehash，这也是优于ThreadLocalMap的地方

可以看到，直接基于 index 的向上 2 次幂取整。然后就是扩容的拷贝，这里是直接进行数组拷贝，**不需要进行 rehash**，而 ThreadLocalMap 的扩容需要进行rehash，也就是重新基于 key 的 hash 值进行位置的分配，所以**这个也是 FastThreadLocal 优于ThreadLocal 的一个点。**

对了，上面那个向上 2 次幂取整的操作，不知道你们熟悉不熟悉，这个和 HashMap 的实现是一致的。

```
/**
 * Returns a power of two size for the given target capacity.
 */
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

咳咳，但是我没有证据，只能说优秀的代码，就是**源远流长**。

所以从上面的实现可以得知 Netty 就是特意这样设计的，用多余的空间去换取不会冲突的 set 和 get，这样写入和获取的速度就更快了，这就是典型的**空间换时间**。

好了，想必此时你已经弄懂了 FastThreadLocal 的核心原理了，我们再来看看 get 方法的实现，我想你应该能脑补这个实现了。



```

public final V get() {
    InternalThreadLocalMap threadLocalMap = InternalThreadLocalMap.get();
    Object v = threadLocalMap.indexedVariable(index);
    if (v != InternalThreadLocalMap.UNSET) {
        return (V) v;
    }

    return initialize(threadLocalMap);
}

public Object indexedVariable(int index) {
    Object[] lookup = indexedVariables;
    return index < lookup.length? lookup[index] : UNSET;
}

```

从线程里拿到 map

值不是初始值，就返回

如果是初始值，就执行初始化方法赋值

直接进行一个数组的下标获取

是吧，没啥难度，index 就是 FastThreadLocal 构造时候预先分配好的那个下标，然后直接进行一个数组下标查找，如果没找到就调用 init 方法进行初始化。

我们这里再继续探究一下 `InternalThreadLocalMap.get()`，这里面做了一个兼容。不过我要先介绍一下 `FastThreadLocalThread`，就是这玩意替代了 `Thread`。

```

public class FastThreadLocalThread extends Thread {
    // This will be set to true if we have a chance to wrap the Runnable.
    private final boolean cleanupFastThreadLocals;

    private InternalThreadLocalMap threadLocalMap;
}

```

可以看到它继承了 `Thread`，并且弄了一个成员变量就是我们前面说的 `InternalThreadLocalMap`。

然后我们再来看一下 `get` 方法，我截了好几个，不过逻辑很简单。

```

public static InternalThreadLocalMap get() {
    Thread thread = Thread.currentThread(); 获取当前线程
    if (thread instanceof FastThreadLocalThread) { 如果当前线程是 fastthread类型
        return fastGet((FastThreadLocalThread) thread); 就执行一个 fast get
    } else {
        return slowGet(); 否则就 slowget
    }
}

private static InternalThreadLocalMap fastGet(FastThreadLocalThread thread) {
    InternalThreadLocalMap threadLocalMap = thread.threadLocalMap();
    if (threadLocalMap == null) { 为空就 new 一个 拿到 InternalThreadlocalmap
        thread.setThreadLocalMap(threadLocalMap = new InternalThreadLocalMap());
    }
    return threadLocalMap; 直接返回
}

private static InternalThreadLocalMap slowGet() {
    InternalThreadLocalMap ret = slowThreadLocalMap.get(); 从这个map 拿到 InternalThreadlocalmap
    if (ret == null) {
        ret = new InternalThreadLocalMap();
        slowThreadLocalMap.set(ret); 不然就new一个, (这里吐槽一下: 为啥一样的赋值功能, 这个分两行, 上面就一行, 不统一一下)
    }
    return ret;
}

private static final ThreadLocal<InternalThreadLocalMap> slowThreadLocalMap =
    new ThreadLocal<>();

public static void remove() {
    Thread thread = Thread.currentThread();
    if (thread instanceof FastThreadLocalThread) {
        ((FastThreadLocalThread) thread).setThreadLocalMap(null);
    } else {
        slowThreadLocalMap.remove();
    }
}

```

这里之所以分了 fastGet 和 slowGet 是为了做一个兼容，假设有个不熟悉的人，他用了 FastThreadLocal 但是没有配套使用 FastThreadLocalThread，然后调用 FastThreadLocal#get 的时候去 Thread 里面找 InternalThreadLocalMap 那不就傻了吗，会报错的。

所以就再弄了个 slowThreadLocalMap，它是个 ThreadLocal，里面保存 InternalThreadLocalMap 来兼容一下这个情况。

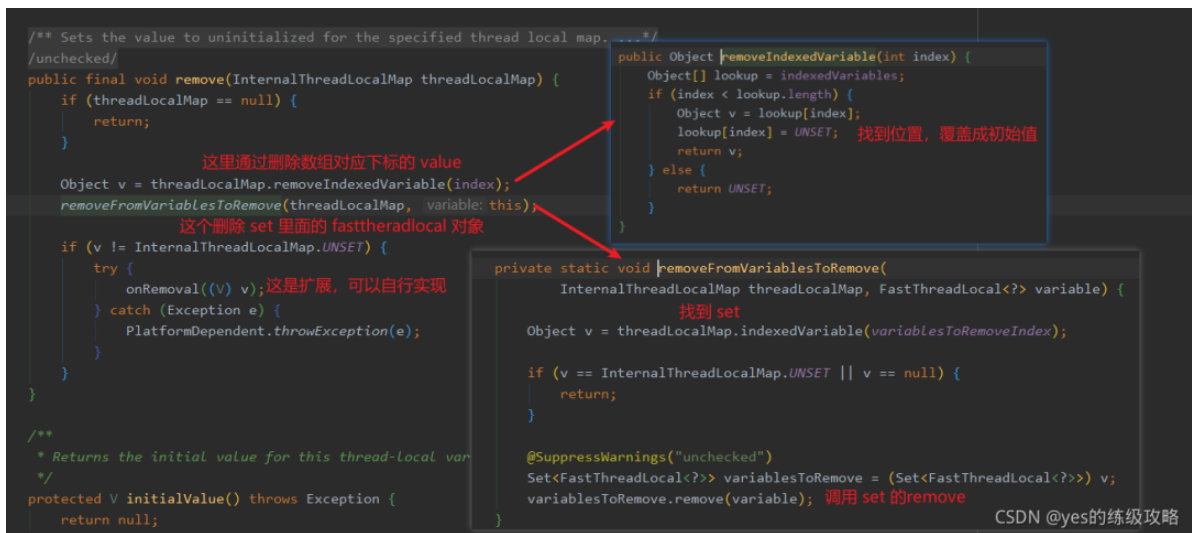
从这里我们也能得知，FastThreadLocal 最好和 FastThreadLocalThread 配套使用，不然就隔了一层了。

```

1 FastThreadLocal<String> threadLocal = new FastThreadLocal<String>();
2 Thread t = new FastThreadLocalThread(new Runnable() { //记得要 new
    FastThreadLocalThread
3     public void run() {
4         threadLocal.get();
5         ....
6     }
7 });

```

好了，get 和 set 这两个核心操作都分析完了，我们最后再来看一下 remove 操作吧。



很简单对吧，把数组里的 value 给覆盖了，然后再到 set 里把对应的 `FastThreadLocal` 对象给删了。

不过看到这里，可能有人会发出疑惑，内存泄漏相关的点呢？

其实吧，可以看到 `FastThreadLocal` 就没用弱引用，所以它把无用 `FastThreadLocal` 的清理就寄托到规范使用上，即没用了就主动调用 `remove` 方法。

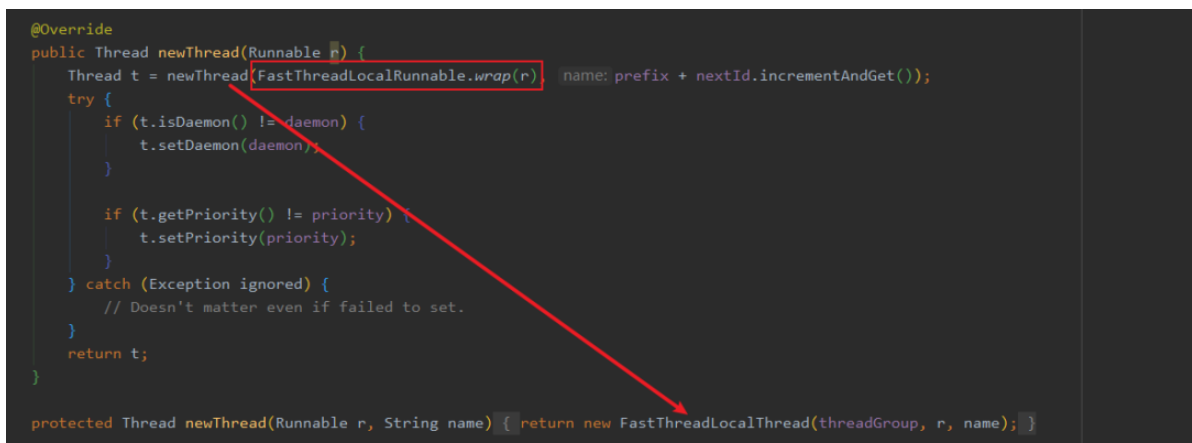
但是它曲线救国了一下，我们来看一下 `FastThreadLocalRunnable` 这个类：



我已经把重点画出来了，可以看到这个 `Runnable` 执行完毕之后，会主动调用 `FastThreadLocal.removeAll()` 来清理所有的 `FastThreadLocal`，这就是我说的曲线救国，怕你完了调用 `remove`，没事我帮你封装一下，就是这么贴心。

当然，这个前提是你不能用 `Runnable` 而是用 `FastThreadLocalRunnable`。不过这里 `Netty` 也是做了封装的。

`Netty` 实现了一个 `DefaultThreadFactory` 工厂类来创建线程。



你看，你传入 Runnable 是吧，没事，我把它包成 FastThreadLocalRunnable，并且我 new 回去的线程是 FastThreadLocalThread 类型，这样就能在很大程度上避免使用的错误，也减少了使用的难度。

这也是工厂方法这个设计模式的好处之一啦。所以工程上如果怕对方没用对，我们就封装了再给别人使用，这样也屏蔽了一些细节，**他好你也好**。

所以说多看看开源框架的源码，有很多可以学习的地方！好了，FastThreadLocal 原理大致就说到这里。

## FastThreadLocal VS ThreadLocal

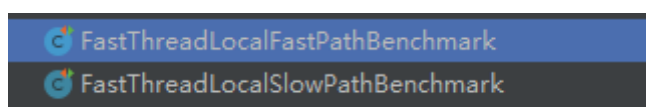
到此，我们已经充分了解了两者之间的不同，但是 Fast 到底有多 Fast 呢？

我们用实验说话，Netty 源码里面已经有 benchmark 了，我们直接跑就行了

里面有两个实验：

FastPath 对应的是使用 FastThreadLocalThread 线程对象。

SlowPath 对应的是使用 Thread 线程对象。



两个实验都是分别定义了 ThreadLocal 和 FastThreadLocal：

```
@Threads(4)
@Measurement(iterations = 10, batchSize = 100)
public class FastThreadLocalFastPathBenchmark extends AbstractMicrobenchmark {

    private static final Random rand = new Random();

    /unchecked/
    private static final ThreadLocal<Integer>[] jdkThreadLocals = new ThreadLocal[128];
    /unchecked/
    private static final FastThreadLocal<Integer>[] fastThreadLocals = new FastThreadLocal[jdkThreadLocals.length];

    static {
        for (int i = 0; i < jdkThreadLocals.length; i++) {
            final int num = rand.nextInt();
            jdkThreadLocals[i] = new ThreadLocal<Integer>() {
                @Override
                protected Integer initialValue() {
                    return num;
                }
            };
            fastThreadLocals[i] = new FastThreadLocal<Integer>() {
                @Override
                protected Integer initialValue() {
                    return num;
                }
            };
        }
    }

    @Benchmark
    public void jdkThreadLocalGet(Blackhole bh) {
        for (ThreadLocal<Integer> i: jdkThreadLocals) {
            bh.consume(i.get());
        }
    }

    @Benchmark
    public void fastThreadLocal(Blackhole bh) {
        for (FastThreadLocal<Integer> i: fastThreadLocals) {
            bh.consume(i.get());
        }
    }
}
```

我们来看一下执行的结果：

FastPath:

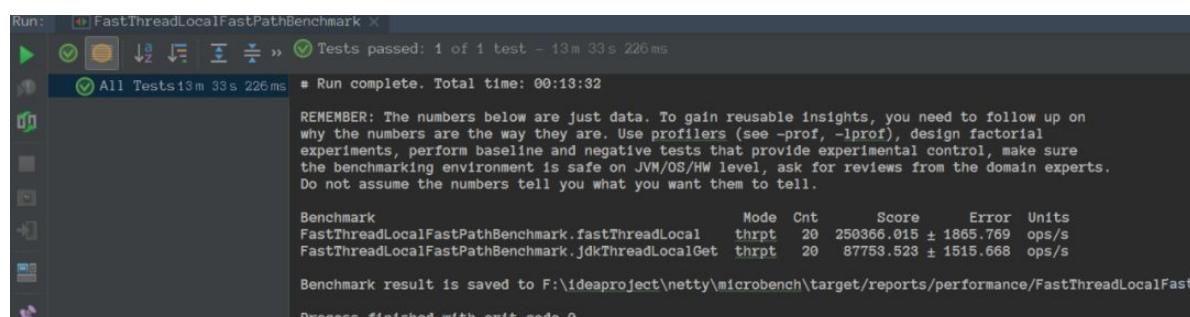
Benchmark	Mode	Cnt	Score	Error	Units
FastThreadLocalFastPathBenchmark.fastThreadLocal	thrpt	20	94638.527 ± 192.274		ops/s
FastThreadLocalFastPathBenchmark.jdkThreadLocalGet	thrpt	20	75701.163 ± 496.036		ops/s

SlowPath:

Benchmark	Mode	Cnt	Score	Error	Units
FastThreadLocalSlowPathBenchmark.fastThreadLocal	thrpt	20	61609.618 ± 655.548		ops/s
FastThreadLocalSlowPathBenchmark.jdkThreadLocalGet	thrpt	20	76051.448 ± 1882.671		ops/s

可以看到搭配 FastThreadLocalThread 来使用 FastThreadLocal 吞吐确实比使用 ThreadLocal 大，但是好像也没大太多？

不过，我在网上有看别比人的 benchmark 对比，同样的代码，他的结果是大了三倍。



```
Run: FastThreadLocalFastPathBenchmark x
Tests passed: 1 of 1 test - 13m 33s 226ms
All Tests 13m 33s 226ms * Run complete. Total time: 00:13:32

REMEMBER: The numbers below are just data. To gain reusable insights, you need to follow up on why the numbers are the way they are. Use profilers (see -prof, -lprof), design factorial experiments, perform baseline and negative tests that provide experimental control, make sure the benchmarking environment is safe on JVM/OS/HW level, ask for reviews from the domain experts. Do not assume the numbers tell you what you want them to tell.

Benchmark                                     Mode  Cnt   Score    Error   Units
FastThreadLocalFastPathBenchmark.fastThreadLocal  thrpt   20 250366.015 ± 1865.769 ops/s
FastThreadLocalFastPathBenchmark.jdkThreadLocalGet thrpt   20  87753.523 ± 1515.668 ops/s

Benchmark result is saved to F:\ideaproject\netty\microbench\target\reports\performance\FastThreadLocalFast
Process finished with exit code 0
```

我反正又跑了几遍，每次都比原生的 ThreadLocal 吞吐好，但是也没好那么多...有点奇怪。

至于 FastThreadLocal 搭配 Thread 则吞吐比 ThreadLocal 都少，说明 FastThreadLocal 的使用必须得搭配 FastThreadLocalThread，不然就是**反向优化**了。

代码在 netty 的 microbench 这个项目里，有兴趣的可以自己 down 下来跑一跑看看。

## 最后

我们再来总结一下：

- FastThreadLocal 通过分配下标直接定位 value，不会有 hash 冲突，效率较高。
- FastThreadLocal 采用空间换时间的方式来提高效率。
- FastThreadLocal 需要配套 FastThreadLocalThread 使用，不然还不如原生 ThreadLocal。
- FastThreadLocal 使用最好配套 FastThreadLocalRunnable，这样执行完任务后会主动调用 removeAll 来移除所有 FastThreadLocal，防止内存泄漏。
- FastThreadLocal 的使用也是推荐用完之后，主动调用 remove。

这就是 Netty 实现的加强版 ThreadLocal，如果你看过 Netty 源码，你会发现内部是有挺多使用 ThreadLocal 的场景，所以这个优化还是有必要的。

并且 Netty work 线程池默认线程数是两倍 CPU 核心数，所以线程不会太多，那么空间的浪费其实也不会很多，所以这波空间换时间影响不大。

好了，文章就到这了。挖个坑，我在 InternalThreadLocalMap 这个类里面发现了一些奇怪的 long 变量。

```
/** @deprecated These padding fields will be removed in the future. */
public long rp1, rp2, rp3, rp4, rp5, rp6, rp7, rp8, rp9;
```

懂行的同学看着可能知道，这是为了填充 Cache Line，避免伪共享问题的产生。

ok，那为什么被标记了@deprecated? 并且说将来的版本要被移除?

且听下回分解。