23 Netty的FastThreadLocal到底快在哪里

更新时间：2020-08-11 09:46:09

> 天才就是这样，终身努力，便成天才。——门捷列夫

前言

你好，我是彤哥。

前面几节，我们一起学习了 Netty 中的内存池和对象池相关的知识，不管是内存池还是对象池，它们底层都有使用一种叫作线程本地缓存的东西，我们知道，在 Java 中有 ThreadLocal 可以存储线程本地变量，但是，在 Netty 中，并没有使用 Java 原生的 ThreadLocal，而是创建了一个新的类 ——FastThreadLocal，从名字上就可以看出，它很快，那么，它究竟是不是真的正如其名，很 Fast 呢？它相比于 Java 原生的 ThreadLocal，除了快还有哪些特性呢？

今天，就让我们一起来学习 Netty 中鼎鼎大名的快男 ——FastThreadLocal。

问题

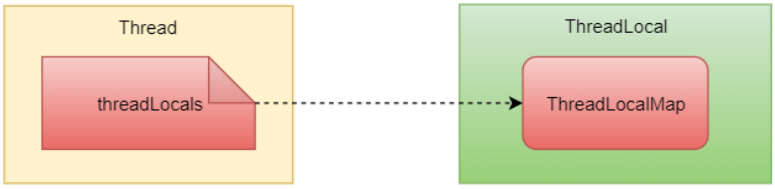
关于 FastThreadLocal 的学习，我想问你这么几个问题：

1. Java 原生的 ThreadLocal 是如何实现的？有什么缺点？
2. FastThreadLocal 是如何实现的？有什么优点和缺点？
3. FastThreadLocal 有哪些使用的姿势？
4. FastThreadLocal 一定比 ThreadLocal 快吗？
5. FastThreadLocal 除了快还有什么优势？

ThreadLocal 简介

在正式介绍 FastThreadLocal 之前，让我们先从 Java 原生的 ThreadLocal 开始说起。

在 ThreadLocal 体系中，有三个特别重要的类：Thread、ThreadLocal、ThreadLocalMap，其中 ThreadLocalMap 是 ThreadLocal 的内部类。



Thread，线程类，就是我们创建线程的那个类，这里面有一个字段叫作 threadLocals，它的类型是 ThreadLocalMap。

ThreadLocal，线程本地类，用于存放线程本地变量，每一个 ThreadLocal 中存储的值在每一个线程中都有一个副本，这个副本是保存在 Thread 的 threadLocals 中的。

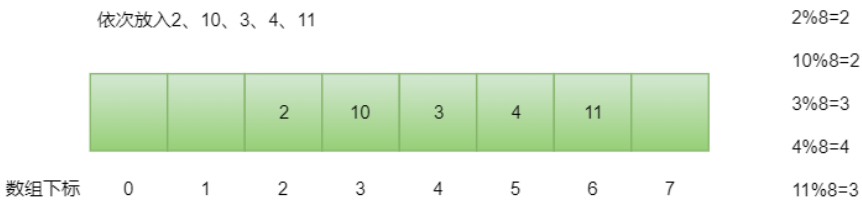
ThreadLocalMap，它是一种使用线性探测法实现的哈希表，可以理解成就是一个 HashMap，不过这个哈希表的 key 是 ThreadLocal，value 是 ThreadLocal 存储的值，这么来说可能比较难理解，我用个伪代码来表示一下：

```
ThreadLocalMap<ThreadLocal, Object> map = new ThreadLocalMap<>();
ThreadLocal threadLocal = new ThreadLocal();
User user = new User();
map.put(threadLocal, user);
```

当然，实际使用肯定是不能这样用的，这里仅用于描述 ThreadLocalMap 的组成结构。

什么是线性探测法呢？

使用线性探测法的哈希表使用数组存储元素，每次添加元素的时候，如果 hash 到的位置已经有元素了，则向后移动一位检测是否有元素，如果还有元素，继续往后移直到一个没有元素的位置把当前要添加的元素放在那个位置。比如，对于一个容量为 8 的哈希表，我们依次放入 2、10、3、4、11 这么几个元素：



整个过程是这样的：

1. 2%8=2，所以放在下标为 2 的位置；
2. 10%8=2，所以放在下标为 2 的位置，但是下标 2 的位置有元素了，往后移一位，到下标为 3 的位置，没有元素，所以 10 放在了下标为 3 的位置；
3. 3%8=3，所以放在下标为 3 的位置，但是下标 3 的位置有元素了，往后移一位，到下标为 4 的位置，没有元素，所以 3 放在了下标为 4 的位置；

4. 4%8=4，所以放在下标为 4 的位置，但是下标 4 的位置有元素了，往后移一位，到下标为 5 的位置，没有元素，所以 4 放在了下标为 5 的位置；
5. 11%8=3，所以放在下标为 3 的位置，但是下标 3 的位置有元素了，往后移一位，到下标为 4 的位置，也有元素了，继续后移，到下标为 5 的位置，依然有元素，继续后移，到下标为 6 的位置，没有元素，所以 11 放在了下标为 6 的位置；

可以看到，使用线性探测法实现的哈希表非常容易出现哈希冲突，且解决冲突的过程时间复杂度非常高，最高可以达到 O (n) 的时间复杂度。

既然线性探测法性能这么差，为什么 ThreadLocal 还要使用线性探测法呢？我认为原因有两点：

1. ThreadLocal 是 JDK1.2 就加入的类，比较早，那时候性能还不是主要的关注点；
2. 线程本地变量并不会存在很多，比如，使用 Spring 的情况下，一个线程也就差不多 30 多个 ThreadLocal 变量，所以，对性能影响并不是特别明显；

> 关于哈希表的更多内容，可以参考这篇文章《哈希表》。

另外，ThreadLocalMap 中的 Entry 是一个弱引用，它引用的对象就是它的 key 值，即 ThreadLocal 变量，所以，当这个 key 不具有强引用时，下一次垃圾回收时，这个弱引用本身会进入到引用队列中，等待被回收，关于弱引用的相关知识，可以参考《再谈 ByteBuffer》那一章的讲解。

那么，我们该如何使用 ThreadLocal 呢？

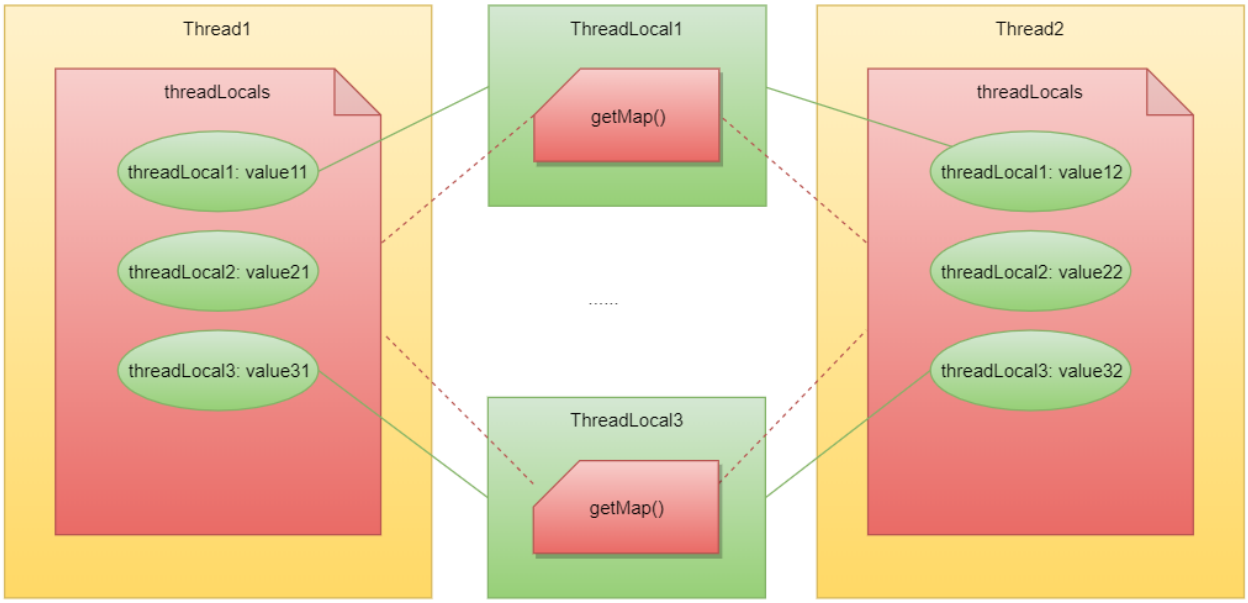
让我们来看看 ThreadLocal 的作者怎么说：

> ThreadLocal instances are typically *private static* fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

作者告诉我们，我们应该把 ThreadLocal 作为类的静态私有变量来使用，让我们看一个例子：

```
public class ThreadLocalTest {
    // threadLocal1
    private static ThreadLocal<String> STRING_THREAD_LOCAL = ThreadLocal.withInitial(() -> Thread.currentThread().getName());
    // threadLocal2
    private static ThreadLocal<Long> LONG_THREAD_LOCAL = ThreadLocal.withInitial(() -> Thread.currentThread().getId());
    // threadLocal3
    private static ThreadLocal<User> USER_THREAD_LOCAL = new ThreadLocal<>();

    public static void main(String[] args) {
        // thread1
        new Thread(() -> {
            User user = new User(1L, "test001");
            USER_THREAD_LOCAL.set(user);

            System.out.println("001：threadName: " + STRING_THREAD_LOCAL.get());
            System.out.println("001：threadId: " + LONG_THREAD_LOCAL.get());
            System.out.println("001：" + USER_THREAD_LOCAL.get());
        }, "thread-001").start();

        // thread2
        new Thread(() -> {
            User user = new User(2L, "test002");
            USER_THREAD_LOCAL.set(user);

            System.out.println("002：threadName: " + STRING_THREAD_LOCAL.get());
            System.out.println("002：threadId: " + LONG_THREAD_LOCAL.get());
            System.out.println("002：" + USER_THREAD_LOCAL.get());
        }, "thread-002").start();
    }

    static class User {
        Long id;
        String name;

        public User(Long id, String name) {
            this.id = id;
            this.name = name;
        }

        @Override
        public String toString() {
            return "id=" + id + ", name=" + name;
        }
    }

}
```

在这个示例中，我们创建了三个 ThreadLocal，他们的值分别为 String、Long、User，然后在 main () 方法中分别创建了两个 Thread，那么，在这两个 Thread 生命周期结束之前，这三个 ThreadLocal 分别是怎么存储的呢？



在调用 ThreadLocal 的 set ()/setInitialValue () 等方法时，会获取到 Thread 中的 threadLocals 这个 Map，然后调用它的 set () 方法（类似于 HashMap 的 put () 方法）把当前 ThreadLocal 本身作为 key，添加到这个 Map 中，最终的存储结构如上图所示。

可以看到，每一个 ThreadLocal 在每一个线程中都保存了一份，而且它们对应的 value 不同。

那么，在生产中，是否可以使用到 ThreadLocal 呢？

答案是当然的，比如一些具有状态的大对象的创建，我们完全可以使用 ThreadLocal 来保存，使得每一个线程都有一个不同的副本，各副本之间的状态不会产生影响，比如经典的就是 SimpleDateFormat、Random 等对象的使用。另外，在诸如 Spring 等 Web 应用场景中，我们也可以使用 ThreadLocal 来保存用户信息，比如，前置拦截器中获取一次用户信息把它保存在 ThreadLocal 中，这样在后面的调用过程中都可以直接拿来使用（无异步调用），在后置拦截器中再把这个用户信息清除即可。此外，还有分布式 ID 的追踪等场景。

既然，ThreadLocal 这么好用，那么，Netty 为什么还要自己造一个 FastThreadLocal 呢？

这就要归结于 ThreadLocal 本身的一些缺陷了，它有哪些缺陷呢？我归纳了一下，主要是以下两点：

- 存储数据的 ThreadLocalMap 使用的是线性探测法，效率低下；
- 使用结束未及时 remove () 掉 ThreadLocal 中的值，容易造成内存泄漏，这种情况只能依靠下一次调用 ThreadLocal 的时候来清除无用的数据，可以参考 ThreadLocal 的 set ()/get ()/remove () 中的相关代码；

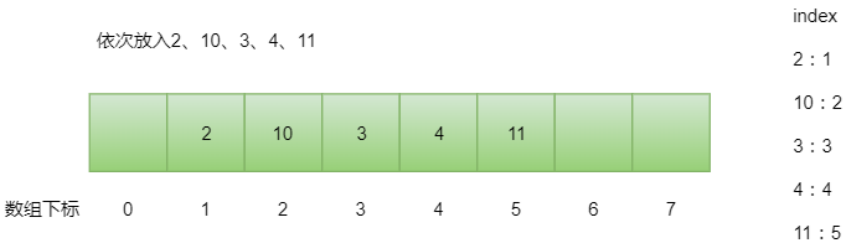
那么，Netty 是如何改造 ThreadLocal 的呢？下面我们就来一起学习 FastThreadLocal 这个非常 Fast 的 ThreadLocal。

FastThreadLocal 简介

在 Netty 中，为了配合 FastThreadLocal 的使用，还定义了另外两个非常重要的类 ——InternalThreadLocalMap 和 FastThreadLocalThread。

我们先来看看 InternalThreadLocalMap 类。

InternalThreadLocalMap，它是对原生 ThreadLocalMap 的优化，它不再使用线性探测法实现，而是显式地给每个 ThreadLocal 分配一个 index，使用这个 index 定位 ThreadLocal 在数组中的精确位置，可以让时间复杂度降低到 O (1)，这个性能提升是很明显的，比如，我们同样放入 2、10、3、4、11 这么几个元素：



整个过程非常简单，每个元素创建的时候都给它分配一个 index，对于 2、10、3、4、11，它们的索引分别是 1、2、3、4、5，这样，它们在放入数组的时候直接按这个 index 作为下标去数组对应的位置即可，也不会出现 hash 冲突，当索引大小达到了数组长度扩容就好了。

不过，这样使用也有个问题，就是当一个元素回收的时候，它的 index 并不会回收，也就是数组对应的位置不能重复利用，但是，这又不是一个问题，因为上面也说过了，ThreadLocal 一般作为类的静态属性来使用，正常情况下，是永远不会被回收的，所以，在 ThreadLocal 的使用场景下，这样的 Map 完全没有问题。

我们再来看看 FastThreadLocalThread 类。

FastThreadLocalThread，它是对 Thread 的包装，目的是为了配合 FastThreadLocal 的使用。它里面封装了一个 InternalThreadLocalMap 字段，这样的话，以前使用 Thread 的 threadLocals 存储元素就变成了使用这个 InternalThreadLocalMap 来存储元素，以达到上面说的利用 InternalThreadLocalMap 高性能。

另外，FastThreadLocalThread 中的构造方法还对任务进行了包装，使得线程任务运行结束的时候可以自动清理掉本线程相关的本地变量。

OK，讲了这么多，怎么能没有源码呢？

FastThreadLocal 源码剖析

调试用例

调试用例非常简单，FastThreadLocal 的使用基本上与 ThreadLocal 完全一致，所以，我们只需要拿上面 ThreadLocal 的使用简单修改即可，这也是 Netty 比较厉害的地方：

```
public class FastThreadLocalTest {

    private static FastThreadLocal<User> USER_THREAD_LOCAL = new FastThreadLocal<>();

    public static void main(String[] args) {
        // thread1
        new FastThreadLocalThread(() -> {
            User user = new User(1L, "test001");
            USER_THREAD_LOCAL.set(user);

            System.out.println("001：" + USER_THREAD_LOCAL.get());
        }, "thread-001").start();

        // thread2
        new FastThreadLocalThread(() -> {
            User user = new User(2L, "test002");
            USER_THREAD_LOCAL.set(user);

            System.out.println("002：" + USER_THREAD_LOCAL.get());
        }, "thread-002").start();
    }
}
```

为了简单一点，我们这个用例中只保留一个线程本地变量。然后，进入调试模式，在 USER_THREAD_LOCAL 声明的地方打个断点，并跟踪进去：

```
// 创建一个FastThreadLocal
public FastThreadLocal() {
    // 给它一个index
    index = InternalThreadLocalMap.nextVariableIndex();
}
// InternalThreadLocalMap#nextVariableIndex
public static int nextVariableIndex() {
    // static final AtomicInteger nextIndex = new AtomicInteger();
    // 这个index是从一个AtomicInteger中获取的
    int index = nextIndex.getAndIncrement();
    if (index < 0) {
        nextIndex.decrementAndGet();
        throw new IllegalStateException("too many thread-local indexed variables");
    }
    return index;
}
```

可以看到，在 FastThreadLocal 创建的时候给它分配了一个 index，这个 index 是从一个 AtomicInteger 中获取的，理论上来说，index 的值应该从 0 开始，但是 FastThreadLocal 中有一个常量字段 variablesToRemoveIndex，它会在 FastThreadLocal 加载的时候就从 AtomicInteger 中拿走了第一个值 0，所以，实际上，FastThreadLocal 中的 index 是从 1 开始的。

此时，我们就创建好了一个 FastThreadLocal，让我们再把断点打在创建第一个 FastThreadLocalThread 的地方，并跟踪进去：

```
public FastThreadLocalThread(Runnable target, String name) {
    // 1. 包装任务
    // 2. 调用父类Thread的构造方法
    super(FastThreadLocalRunnable.wrap(target), name);
    cleanupFastThreadLocals = true;
}
```

可以看到，在创建 FastThreadLocalThread 的时候对其任务进行了再包装，那么，这个包装后的任务长啥样呢？让我们看看 FastThreadLocalRunnable 这个类：

```
final class FastThreadLocalRunnable implements Runnable {
    // 原始任务
    private final Runnable runnable;

    private FastThreadLocalRunnable(Runnable runnable) {
        this.runnable = ObjectUtil.checkNotNull(runnable, "runnable");
    }

    @Override
    public void run() {
        // 对原始任务的run()方法进行了包装
        try {
            runnable.run();
        } finally {
            // run()结束后清理掉FastThreadLocal
            FastThreadLocal.removeAll();
        }
    }

    // 包装方法
    static Runnable wrap(Runnable runnable) {
        return runnable instanceof FastThreadLocalRunnable ? runnable : new FastThreadLocalRunnable(runnable);
    }
}
```

这里的包装也非常简单，只是对原始任务的 run () 方法进行包装，使其运行完毕后，清理掉 FastThreadLocal，那么，这个清理到底清理了什么呢？我们后面再看。

OK，包装任务并调用父类 Thread 的构造方法之后，我们的 FastThreadLocalThread 就创建完毕了，此时，执行它的 start () 方法，将进入到线程的 run () 方法中，所以，我们现在在 Thread 的 run () 方法中打一个断点：

```
// Thread#run
@Override
public void run() {
    // 不为空，是上面传进来的包装后的任务
    if (target != null) {
        target.run();
    }
}
// FastThreadLocalRunnable#run
@Override
public void run() {
    try {
        // 1. 运行原始任务
        runnable.run();
    } finally {
        // 2. 清理工作
        FastThreadLocal.removeAll();
    }
}
```

这里运行到了 FastThreadLocalRunnable 这个类里面，这里主要干两件事，一是运行原始任务，即我们传进去的 Runnable，二是清理工作。此时，继续跟踪将进入 main () 方法中我们传入的 Runnable 中，即下面这块代码：

```
new FastThreadLocalThread(() -> {
    User user = new User(1L, "test001");
    USER_THREAD_LOCAL.set(user);

    System.out.println("001：" + USER_THREAD_LOCAL.get());
}, "thread-001").start();
```

在我们的 Runnable 中主要对 USER_THREAD_LOCAL 进行了 set () 和 /get () 方法的调用，我们先跟踪到 set () 方法中：

```
// FastThreadLocal#set(V)
public final void set(V value) {
    // 如果value不等于UNSET，就把它set进去
    // 初始化时InternalThreadLocalMap中的数组的每个元素都会被初始化为UNSET
    if (value != InternalThreadLocalMap.UNSET) {
        // 1. 获取存储元素的InternalThreadLocalMap
        InternalThreadLocalMap threadLocalMap = InternalThreadLocalMap.get();
        // 2. 设置值
        setKnownNotUnset(threadLocalMap, value);
    } else {
        remove();
    }
}
```

在 set () 方法里面，首先，获取到当前线程存储本地变量的 Map，然后，往这个 Map 中设置值。我们先来看看是如何获取到当前线程对应的 Map 的：

```
public static InternalThreadLocalMap get() {
    Thread thread = Thread.currentThread();
    // 判断当前线程是不是FastThreadLocalThread
    if (thread instanceof FastThreadLocalThread) {
        // 如果是，使用fastGet
        return fastGet((FastThreadLocalThread) thread);
    } else {
        // 如果否，使用slowGet
        return slowGet();
    }
}
// fast
private static InternalThreadLocalMap fastGet(FastThreadLocalThread thread) {
    // 这个Map是存储在FastThreadLocalThread中的，变量名为threadLocalMap
    InternalThreadLocalMap threadLocalMap = thread.threadLocalMap();
    if (threadLocalMap == null) {
        // 初始化Map
        thread.setThreadLocalMap(threadLocalMap = new InternalThreadLocalMap());
    }
    return threadLocalMap;
}
// slow
private static InternalThreadLocalMap slowGet() {
    // UnpaddedInternalThreadLocalMap是InternalThreadLocalMap的父类
    // 里面存储了一个Java原生的ThreadLocal，它的值是InternalThreadLocalMap
    // 这里是把这个Map取出来
    // 也就是说，如果当前线程对象不是FastThreadLocalThread
    // 就使用Java原生的ThreadLocal来存储一个InternalThreadLocalMap
    // 跟Netty相关的本地变量还是存储在InternalThreadLocalMap中
    ThreadLocal<InternalThreadLocalMap> slowThreadLocalMap = UnpaddedInternalThreadLocalMap.slowThreadLocalMap;
    InternalThreadLocalMap ret = slowThreadLocalMap.get();
    if (ret == null) {
        // 初始化Map
        ret = new InternalThreadLocalMap();
        slowThreadLocalMap.set(ret);
    }
    return ret;
}
```

获取存储本地线程变量的 Map 分为两种情况：

1. 如果当前线程是 FastThreadLocalThread，则直接取 FastThreadLocalThread 的 threadLocalMap 属性即可；
2. 如果当前线程不是 FastThreadLocalThread，则在 UnpaddedInternalThreadLocalMap 中保存一个

ThreadLocal，在这个 ThreadLocal 里面保存了一个线程本地变量 InternalThreadLocalMap，再把 Netty 相关的线程本地变量放到这个 Map 中，这里的逻辑有点绕，请仔细体会。

OK，此时，我们拿到了这个 Map，再看看是怎么把值设置进去的呢？

```
// FastThreadLocal#setKnownNotUnset
private void setKnownNotUnset(InternalThreadLocalMap threadLocalMap, V value) {
    // 取当前FastThreadLocal的索引index
    if (threadLocalMap.setIndexedVariable(index, value)) {
        // key，将当前FastThreadLocal添加到待清理的Set中
        addToVariablesToRemove(threadLocalMap, this);
    }
}
// InternalThreadLocalMap#setIndexedVariable
public boolean setIndexedVariable(int index, Object value) {
    // 存储元素的数组
    Object[] lookup = indexedVariables;
    // 容量还够
    if (index < lookup.length) {
        // 直接找到数组的位置把值设置进去，时间复杂度为O(1)
        Object oldValue = lookup[index];
        lookup[index] = value;
        return oldValue == UNSET;
    } else {
        // 容量不够，先扩容再设置值
        expandIndexedVariableTableAndSet(index, value);
        return true;
    }
}
private void expandIndexedVariableTableAndSet(int index, Object value) {
    // 旧数组
    Object[] oldArray = indexedVariables;
    // 旧容量
    final int oldCapacity = oldArray.length;

    // 找到大于index的最小的2次方
    // 比如，index=3，则为4
    // index=16，则为32
    // 因为数组下标是从0开始的，16个元素的数组最大的下标为15，所以16需要32个元素的数组
    // 如何实现取大于等于自己的最小2次方呢？只需要把这里改成 index-1 即可
    // 有兴趣的同学可以看看HashMap的tableSizeFor()方法
    int newCapacity = index;
    newCapacity |= newCapacity >>>  1;
    newCapacity |= newCapacity >>>  2;
    newCapacity |= newCapacity >>>  4;
    newCapacity |= newCapacity >>>  8;
    newCapacity |= newCapacity >>> 16;
    newCapacity ++;

    // 创建新数组，并把旧数组的元素全部拷贝过来
    Object[] newArray = Arrays.copyOf(oldArray, newCapacity);
    // 把新数组不包含旧数组的后面部分都初始化为UNSET
    Arrays.fill(newArray, oldCapacity, newArray.length, UNSET);
    // 设置当前的值
    newArray[index] = value;
    // 把新数组赋值给InternalThreadLocalMap存储元素的数组
    indexedVariables = newArray;
}
```

设置值这里也分成两种情况：

1. 容量足够，直接把数组对应下标位置的值设置成新值即可，时间复杂度为 O (1)；
2. 容量不够，先扩容，再设置新值；

扩容这里有个疑问：为什么使用 index 作为基准来扩容，而不是直接扩容为旧数组容量的 2 倍呢？

我举个例子你就明白了，在 Netty 中，默认这个数组的容量是 32，假设我们有 100 个 FastThreadLocal 变量，且它们都没有 set () 过任何值，此时，这个数组大小依然是 32，现在如果要设置第 100 个 FastThreadLocal 的值呢？根据前面的逻辑，我们知道，它的 index 为 100，那么，此时，如果按旧数组的容量扩容为 2 倍，依然无法承载 index 为 100 的这个元素，所以，需要按 index 作为基准来进行扩容。

到这里，元素就已经添加到 Map 中了。在上面，我们看到在添加成功之后，还有一步操作是把当前 ThreadLocal 添加到待清理的 Set 中，这里是怎么实现的呢？让我们跟踪进去看看：

```
private static void addToVariablesToRemove(InternalThreadLocalMap threadLocalMap, FastThreadLocal<?> variable) {
    // variablesToRemoveIndex是FastThreadLocal中的常量，它的值为0
    // 它对应的值同样存储在InternalThreadLocalMap中，是下标为0的元素
    // 这里也就解释了为什么FastThreadLocal的index是从0开始的了，因为被variablesToRemoveIndex占用了
    Object v = threadLocalMap.indexedVariable(variablesToRemoveIndex);
    // 0号元素是一个Set
    Set<FastThreadLocal<?>> variablesToRemove;
    if (v == InternalThreadLocalMap.UNSET || v == null) {
        // 初始化这个Set
        variablesToRemove = Collections.newSetFromMap(new IdentityHashMap<FastThreadLocal<?>, Boolean>());
        // 设置到0号元素中
        threadLocalMap.setIndexedVariable(variablesToRemoveIndex, variablesToRemove);
    } else {
        variablesToRemove = (Set<FastThreadLocal<?>>) v;
    }

    // 把当前FastThreadLocal添加到这个Set中
    variablesToRemove.add(variable);
}
```

这里把当前 FastThreadLocal 添加到了 InternalThreadLocalMap 中数组的 0 号元素对应的 Set 中了，添加到这里干什么呢？

让我们的代码继续前进，这里 set 完元素之后，在我们的 Runnable 中，下一步就是 get () 方法的调用了，里面的代码比较简单，就交给你自己了。

在 get () 方法获取到值且打印完成之后，我们的 Runnable 任务就完成了，此时，会再次回到 FastThreadLocalRunnable 的 run () 方法中，也就是下面这个方法：

```
@Override
public void run() {
    try {
        // 1. 运行原始任务
        runnable.run();
    } finally {
        // 2. 清理工作
        FastThreadLocal.removeAll();
    }
}
```

在我们的 Runnable 任务运行完毕后，相当于当前线程的生命周期已经趋近于结束，所以，这里有个 finally，里面会执行清理工作，那么，都清理了什么东西呢？跟踪进去：

```
public static void removeAll() {
    // 获取Map，getIfSet()方法跟上面的get()方法类似，只是内部不会再做初始化的操作
    InternalThreadLocalMap threadLocalMap = InternalThreadLocalMap.getIfSet();
    if (threadLocalMap == null) {
        // 如果为空，说明还未初始化，线程任务没有操作本地变量，直接返回就好了
        return;
    }

    try {
        // 从0号元素取出待清理的FastThreadLocal
        Object v = threadLocalMap.indexedVariable(variablesToRemoveIndex);
        // 已初始化过
        if (v != null && v != InternalThreadLocalMap.UNSET) {
            // 转换为Set
            @SuppressWarnings("unchecked")
            Set<FastThreadLocal<?>> variablesToRemove = (Set<FastThreadLocal<?>>) v;
            // 转换为数组
            FastThreadLocal<?>[] variablesToRemoveArray =
                variablesToRemove.toArray(new FastThreadLocal[0]);
            // 遍历数组
            for (FastThreadLocal<?> tlv: variablesToRemoveArray) {
                // 1. 调用FastThreadLocal的remove()方法
                tlv.remove(threadLocalMap);
            }
        }
    } finally {
        // 2. 调用InternalThreadLocalMap的remove()方法
        InternalThreadLocalMap.remove();
    }
}
```

这里的逻辑比较简单，会把所有存储在待清理 Set 中的所有 FastThreadLocal 全部清除，即调用它们的 remove () 方法，同时，在最后会统一调用 InternalThreadLocalMap 的 remove () 方法，这两个 remove () 方法有什么区别呢？

```
// 1. FastThreadLocal#remove
public final void remove(InternalThreadLocalMap threadLocalMap) {
    if (threadLocalMap == null) {
        return;
    }
    // 从Map中移除，这里是重置为UNSET元素
    Object v = threadLocalMap.removeIndexedVariable(index);
    // 从待清理的Set中移除
    removeFromVariablesToRemove(threadLocalMap, this);

    if (v != InternalThreadLocalMap.UNSET) {
        try {
            // 执行钩子方法
            onRemoval((V) v);
        } catch (Exception e) {
            PlatformDependent.throwException(e);
        }
    }
}
// 2. InternalThreadLocalMap#remove
public static void remove() {
    Thread thread = Thread.currentThread();
    if (thread instanceof FastThreadLocalThread) {
        // 移除Map本身
        ((FastThreadLocalThread) thread).setThreadLocalMap(null);
    } else {
        // 从ThreadLocal中移除Map本身
        slowThreadLocalMap.remove();
    }
}
```

可以看到，第一个 remove () 方法是把当前 FastThreadLocal 从 threadLocalMap 和待清理 Set 中移除，并执行钩子方法 onRemoval ()；第二个 remove () 方法是把 threadLocalMap 本身移除，当然，这里根据 Thread 的类型判断是从 FastThreadLocalThread 中移除还是从 ThreadLocal 中移除。

在上面，我们提到了一个钩子方法 onRemoval ()，它是干什么的呢？它实际上是一个空方法，这是 Netty 为我们预留的一个方法，我们可以继承自 FastThreadLocal 并实现 onRemoval () 方法，在移除 FastThreadLocal 的时候做一些我们自己的逻辑，比如清理我们自定义的资源等。其实，在 JDK 中，也有很多这种钩子方法，比如线程池 ThreadPoolExecutor.runWorker () 方法的逻辑，HashMap.putVal () 方法的逻辑（LinkedHashMap 就是使用这些钩子方法实现的），等等。

其它

其实，到这里，FastThreadLocal 的源码剖析过程基本结束了，但是，细心的同学可能会发现，InternalThreadLocalMap 还有个父类叫作 UnpaddedInternalThreadLocalMap，从名字可以看出多了一个 Unpadded，这个单词是什么意思呢？

Unpadded，顾名思义，它是 Padded 的反义词，Padded 表示的是缓存行补齐，用于消除伪共享带来的性能问题，它的补齐是通过声明下面一堆 long 类型变量来实现的。

```
public final class InternalThreadLocalMap extends UnpaddedInternalThreadLocalMap {
    public long rp1, rp2, rp3, rp4, rp5, rp6, rp7, rp8, rp9;
}
```

但是，在 InternalThreadLocalMap 中，并没有明确的证据表明这些变量能消除伪共享。经过查询相关的 issue 发现，在早期，添加这些变量确实会补齐缓存行，但是，后面 InternalThreadLocalMap 和 UnpaddedInternalThreadLocalMap 这两个类分别添加了 cleanerFlags 和 arrayList 这两个变量，而这里的 long 类型并没有对应的修改，导致现在是超出缓存行的状态，当然，这不是主要问题，主要问题是，添加这些变量是为了防止哪个变量伪共享的，我们也不得而知。

为了更好地验证这些变量是否真的能够消除伪共享，我做了测试，测试结果是在我的机器上添加这些变量并没有明显的性能提升，同学们也可以在自己的机器上测试下，具体的测试方法见文后。

> 关于伪共享的问题，我们这里就不展开了，有兴趣的同学可以看看这篇文章《伪共享》。

开篇回顾

在开篇，我们提了几个问题，这里根据源码的剖析结果再回顾一下：

FastThreadLocal 有哪些使用的姿势？

FastThreadLocal 正常来说是跟 FastThreadLocalThread 联合使用的，但是，Netty 为了兼容性，也可以跟普通的 Thread 一起使用，只是会使用一种 slow 的方式来运行，这个 slow () 主要体现在 InternalThreadLocalMap 的存储上，使用 FastThreadLocalThread 时，它是存储在 FastThreadLocalThread 中的，使用普通的 Thread 时，它是存储在 Java 原生的 ThreadLocal 中的。

FastThreadLocal 一定比 ThreadLocal 快吗？

不一定，当 FastThreadLocal 与普通 Thread 一起使用的时候，可能会比直接使用 ThreadLocal 还慢，这得根据 ThreadLocal 值的多少和 InternalThreadLocalMap 中值的多少来判断，比如，ThreadLocal 只存储了 InternalThreadLocalMap 一个值而 InternalThreadLocalMap 中存储了 100 个值 和 ThreadLocal 存储了 100 个值而 InternalThreadLocalMap 中只存储了一个值，这两种场景的结果肯定不一样。

FastThreadLocal 除了快还有什么优势？

FastThreadLocal 除了快还能帮助我们自动清理相关资源，这是通过包装 Runnable 来实现的。

其它问题，诸如 ThreadLocal、FastThreadLocal 的实现以及优缺点等，我们这里就不一一解答了。

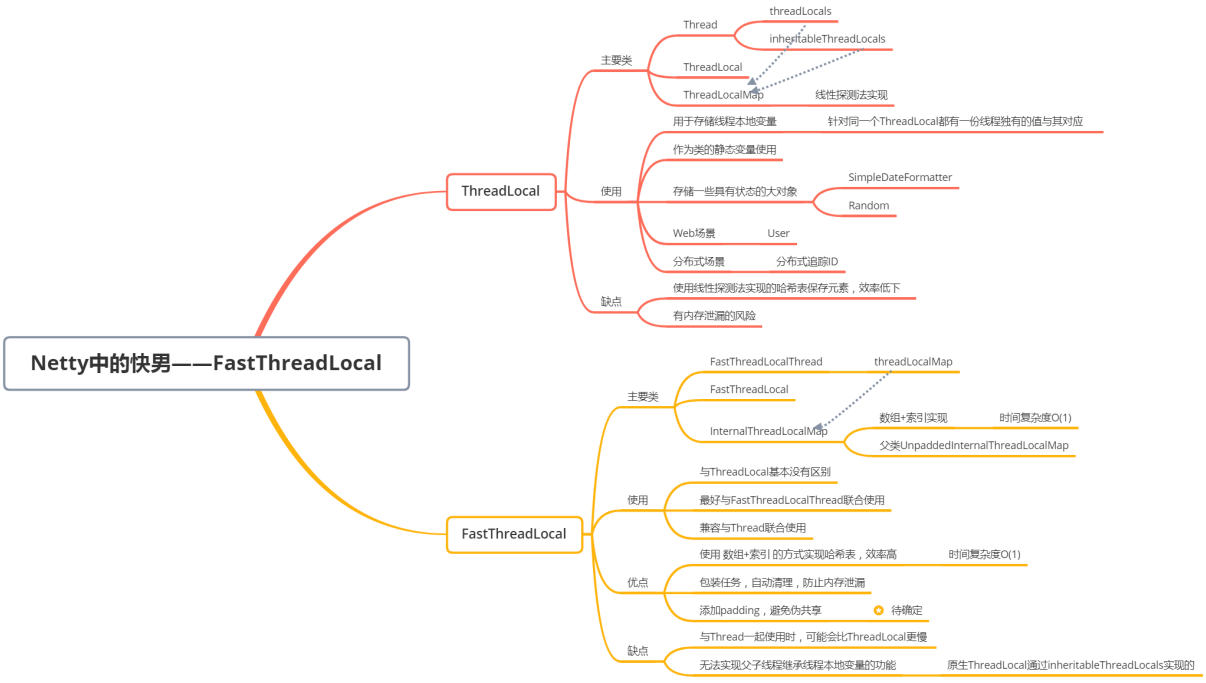
后记

本节，我们一起学习了 Netty 中的一个快类 ——FastThreadLocal，从使用上来说，它与使用 Java 原生的 ThreadLocal 并无明显区别，从实现上来说，它也完全兼容与普通 Thread 的联合使用。这也正是 Netty 的魅力所在。

另外，细心的同学会发现，它的相关代码完全遵循面向对象的思想，领域划分非常明确，比如，上文提到的两个 remove () 方法。

下一节，我们将学习内存池中使用到的另一个非常精彩的类 ——MpscArrayQueue—— 多生产者单消费者队列，敬请期待。

思维导图



关于 InternalThreadLocalMap 的测试

测试用例

```
public class InternalThreadLocalMapPaddingTest {

    /**
     * 单线程运行20次
     */
    @Test
    public void testPadding() {
        List<FastThreadLocal<Boolean>> list = new ArrayList<FastThreadLocal<Boolean>>();
        for (int i = 0; i < 1000000; i++) {
            list.add(new FastThreadLocal<Boolean>());
        }

        Boolean value = Boolean.TRUE;
        for (int j = 0; j < 20; j++) {
            long start = System.currentTimeMillis();
            for (int i = 0; i < 1000000; i++) {
                FastThreadLocal<Boolean> fastThreadLocal = list.get(i);
                fastThreadLocal.set(value=!value);
            }
            for (int i = 0; i < 1000000; i++) {
                FastThreadLocal<Boolean> fastThreadLocal = list.get(i);
                fastThreadLocal.get();
            }
            FastThreadLocal.removeAll();
            long end = System.currentTimeMillis();
            System.out.println(end - start);
        }

    }

    /**
     * 多线程运行20个线程
     */
    @Test
    public void testPaddingMultiThread() throws InterruptedException {
        final List<FastThreadLocal<Boolean>> list = new ArrayList<FastThreadLocal<Boolean>>();
        for (int i = 0; i < 100000; i++) {
            list.add(new FastThreadLocal<Boolean>());
        }

        final CountDownLatch countDownLatch = new CountDownLatch(20);
        for (int k = 0; k < 20; k++) {
            new FastThreadLocalThread(new Runnable() {
                @Override
                public void run() {
                    long start = System.currentTimeMillis();
                    Boolean value = Boolean.TRUE;
                    for (int i = 0; i < 100000; i++) {
                        FastThreadLocal<Boolean> fastThreadLocal = list.get(i);
                        fastThreadLocal.set(value=!value);
                    }
                    for (int i = 0; i < 100000; i++) {
                        FastThreadLocal<Boolean> fastThreadLocal = list.get(i);
                        fastThreadLocal.get();
                    }
                    FastThreadLocal.removeAll();
                    long end = System.currentTimeMillis();
                    System.out.println(end - start);

                    countDownLatch.countDown();
                }
            }).start();
        }
        countDownLatch.await();
        System.out.println("finished");
    }

}
```

测试方法

下载 netty 源码，把这个类放到 netty-common 工程的 test 下面，通过修改 InternalThreadLocalMap 源码，即把 `public long rp1, rp2, rp3, rp4, rp5, rp6, rp7, rp8, rp9;` 注释掉，对比注释前后的运行时间，记住多运行几次观察结果。

另外，在 64 位机器上，还有个指针压缩的概念，可以通过参数 `-XX:+UseCompressedOops` 和 `-XX:-UseCompressedOops` 来启用和禁用指针压缩，然后再结合上面的注释前后进行对比，观察结果。

测试结论

在我的机器上，测试发现，注释前后与是否启用指针压缩，都没有带来明显的性能提升，试想，测试用例中使用了 1000000 和 100000 个 FastThreadLocal 都没有明显的提升，那么，实际生产中最多也就几十几百个 FastThreadLocal，更是可以忽略不计了。

当然，以上测试结论仅限于我的机器，linux 和其它电脑并没有测试过，所以，这个结论并不严谨，仅作参考。

}

22 Netty的对象池又是如何实现的

24 Netty的队列有何不一样