我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro

- NETTY

精尽 Netty 源码解析 —— Util 之 FastThreadLocal

笔者先把 Netty 主要的内容写完，所以关于 FastThreadLocal 的分享，先放在后续的计划里。

> 老芳芳：其实是因为，自己想去研究下 Service Mesh ，所以先简单收个小尾。

当然，良心如我，还是为对这块感兴趣的胖友，先准备好了一篇不错的文章：

- 莫那一鲁道 《Netty 高性能之道 FastThreadLocal 源码分析（快且安全）》
  - 😈 我的好基友，可以关注下他的简书。
- 暗夜君王 《【源起Netty 外传】FastThreadLocal怎么Fast？》

为避免可能 《Netty 高性能之道 FastThreadLocal 源码分析（快且安全）》 被作者删除，笔者这里先复制一份作为备份。

666. 备份

前言

Netty 作为高性能框架，对 JDK 中的很多类都进行了封装了和优化，例如 Thread 类，Netty 使用了 FastThreadLocalRunnable 对所有 DefaultThreadFactory 创建出来的 Runnable 都进行了包装。包装的目的是 run 方法的不同，看代码：

```
public void run() {
    try {
        runnable.run();
    } finally {
        FastThreadLocal.removeAll();
    }
}
```

可以看到，多了一行 FastThreadLocal.removeAll()，众所周知，JDK 中自带的 ThreadLocal 在线程池使用环境中，有内存泄漏的风险，很明显，Netty 为了避免这个 bug，重新进行了封装，而且这个封装线程的名字叫做 FastThreadLocalRunnable，语义很明显：快速的 ThreadLocal！意思说 JDK 自带的慢喽？那我们今天就来看看到底快在哪里？对 ThreadLocal 内存泄漏不清楚或者对 ThreadLoca 不清楚的可以移步 并发编程之 ThreadLocal 源码剖析。

1. 如何使用？

测试用例

运行结果

2. 构造方法解析

构造方法

构造方法中定义了两个变量。 index 和 cleanerFlagIndex，这两个变量且都是 int final 的。且都是通过 InternalThreadLocalMap.nextVariableIndex() 方法而来。

InternalThreadLocalMap.nextVariableIndex() 方法

nextIndex 变量

该方法通过一个原子 int 变量自增得到。也就是说，cleanerFlagIndex 变量比 index 大1，这两个变量的作用稍后我们会看到他们如何使用。这里暂且不表。

3. set 方法解析

set（） 方法

该方法步骤如下：

1. 判断设置的 value 值是否是缺省值，如果是，则调用 remove 方法。
2. 如果不是，则获取道当前线程的 InternalThreadLocalMap。然后将该 FastThreadLocal 对应的 index 下标的 value 替换成新的 value。老的 value 设置成缺省值。

小小的一个 set 方法，内部可是非常的复杂，非战斗人员请尽快撤离！

实际上，这里调用了4个方法：

1. InternalThreadLocalMap.get();
2. setKnownNotUnset(threadLocalMap, value);
3. registerCleaner(threadLocalMap);
4. remove();

让我们慢慢说道说道。

1. InternalThreadLocalMap.get();

代码如下：

```
public static InternalThreadLocalMap get() {
    Thread thread = Thread.currentThread();
    if (thread instanceof FastThreadLocalThread) {
        return fastGet((FastThreadLocalThread) thread);
    } else {
        return slowGet();
```

```
    }
}
```

首先是 InternalThreadLocalMap 的静态方法，方法逻辑很简单，主要是根据当前线程是否是 Netty 的 FastThreadLocalThread 来调用不同的方法，一个是 fast 的，一个 是 slow 的（不是 Netty 的线程就是 slow 的）。哈哈哈，Netty 的作者命名还真是犀利。那我们就看看 fastGet 方法是什么?

```
private static InternalThreadLocalMap fastGet(FastThreadLocalThread thread) {
    InternalThreadLocalMap threadLocalMap = thread.threadLocalMap();
    if (threadLocalMap == null) {
        thread.setThreadLocalMap(threadLocalMap = new InternalThreadLocalMap());
    }
    return threadLocalMap;
}
```

逻辑很简单，获取当前线程的 InternalThreadLocalMap，如果没有，就创建一个。我们看看他的构造方法。

```
public static final Object UNSET = new Object();

private InternalThreadLocalMap() {
    super(newIndexedVariableTable());
}

private static Object[] newIndexedVariableTable() {
    Object[] array = new Object[32];
    Arrays.fill(array, UNSET);
    return array;
}

UnpaddedInternalThreadLocalMap(Object[] indexedVariables) {
    this.indexedVariables = indexedVariables;
}
```

楼主将 3 个关联的方法都放在一起了，方便查看，首先，InternalThreadLocalMap 调用的父类 UnpaddedInternalThreadLocalMap 的构造方法，并传入了一个数组，而这个数组默认大小是 32，里面填充32 个空对象的引用。

那 slowGet 方法又是什么样子的呢？代码如下：

```
static final ThreadLocal<InternalThreadLocalMap> slowThreadLocalMap = new ThreadLocal<InternalThreadLo

private static InternalThreadLocalMap slowGet() {
    ThreadLocal<InternalThreadLocalMap> slowThreadLocalMap = UnpaddedInternalThreadLocalMap.slowThread
    InternalThreadLocalMap ret = slowThreadLocalMap.get();
    if (ret == null) {
        ret = new InternalThreadLocalMap();
        slowThreadLocalMap.set(ret);
    }
    return ret;
}
```

代码还是很简单的，我们分析一下：首先使用 JDK 的 ThreadLocal 获取一个 Netty 的 InternalThreadLocalMap，如果没有就创建一个，并将这个 InternalThreadLocalMap 设置到 JDK 的 ThreadLocal 中，然后返回这个 InternalThreadLocalMap。从这

里可以看出，为了提高性能，Netty 还是避免使用了JDK 的 threadLocalMap，他的方式是曲线救国：在JDK 的 threadLocal 中设置 Netty 的 InternalThreadLocalMap ，然后，这个 InternalThreadLocalMap 中设置 Netty 的 FastThreadLcoal。

好，到这里，我们的 InternalThreadLocalMap.get() 方法就看完了，主要是获取当前线程的 InternalThreadLocalMap，如果没有，就创建一个，这个 Map 内部维护的是一个数组，和 JDK 不同，JDK
维护的是一个使用线性探测法的 Map，可见，从底层数据结构上，JDK 就已经输了，他们的读取速度相差很大，特别是当数据量很大的时候，Netty 的数据结构速度依然不变，而 JDK 由于使用线性探测法，速度会相应的下降。

2. setKnownNotUnset(threadLocalMap, value);

当 InternalThreadLocalMap.get() 返回了 一个 InternalThreadLocalMap，这个时候调用 setKnownNotUnset(threadLocalMap, value); 方法进行操作。代码如下：

```
private boolean setKnownNotUnset(InternalThreadLocalMap threadLocalMap, V value) {
    if (threadLocalMap.setIndexedVariable(index, value)) {
        addToVariablesToRemove(threadLocalMap, this);
        return true;
    }
    return false;
}
```

看方法名称，是设置一个值，但不是 unset，也就是那个空对象。通过 threadLocalMap.setIndexedVariable(index, value) 进行设置。如果返回 true，则调用 addToVariablesToRemove(threadLocalMap, this) 。这两个方法，我们一起看看。先看第一个：

setIndexedVariable 方法

```
public boolean setIndexedVariable(int index, Object value) {
    Object[] lookup = indexedVariables;
    if (index < lookup.length) {
        Object oldValue = lookup[index];
        lookup[index] = value;
        return oldValue == UNSET;
    } else {
        expandIndexedVariableTableAndSet(index, value);
        return true;
    }
}
```

首先，拿到那个 32 长度的数组，如果 FastThreadLocal 的 index 属性小于数组长度，则将值设定到指定槽位。将原来槽位的值设置为空对象。如果原来的对象也是空对象，则返回 true，否则返回 false。

如果不够呢？调用 expandIndexedVariableTableAndSet(index, value) 方法。进入该方法查看。看方法名称是扩大索引并设置值。

```
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;
}
```

这里代码很熟悉，HashMap 中也有这样的代码，我们去看看：

HashMap 中的 tableSizeFor 方法

这段代码的作用就是按原来的容量扩容2倍。并且保证结果是2的幂次方。这里 Netty 的做法和 HashMap 一样，按照原来的容量扩容到最近的 2 的幂次方大小，比如原来32，就扩容到64，然后，将原来数组的内容填充到新数组中，剩余的填充 空对象 ，然后将新数组赋值给成员变量 indexedVariables。完成了一次扩容。

回到 setKnownNotUnset 方法中，setIndexedVariable 方法什么情况下会返回 ture 呢？扩容了，或者没扩容，但插入的对象没有替换掉别的对象，也就是原槽位是空对象。换句话说，只有更新了对象才会返回 false。

也就是说，当新增了对象的时候，会调用 addToVariablesToRemove 方法，如同方法名，添加变量然后删除。我们看看 addToVariablesToRemove(threadLocalMap, this) 方法逻辑：

```
private static void addToVariablesToRemove(InternalThreadLocalMap threadLocalMap, FastThreadLocal<?> v
    // 该变量是 static final 的，因此通常是 0
    Object v = threadLocalMap.indexedVariable(variablesToRemoveIndex);
    Set<FastThreadLocal<?>> variablesToRemove;
    if (v == InternalThreadLocalMap.UNSET || v == null) {
        // 创建一个基于 IdentityHashMap 的 Set，泛型是 FastThreadLocal
        variablesToRemove = Collections.newSetFromMap(new IdentityHashMap<FastThreadLocal<?>, Boolean>
        // 将这个 Set 放到这个 Map 数组的下标 0 处
        threadLocalMap.setIndexedVariable(variablesToRemoveIndex, variablesToRemove);
    } else {
        // 如果拿到的不是 UNSET ，说明这是第二次操作了，因此可以强转为 Set
        variablesToRemove = (Set<FastThreadLocal<?>>) v;
    }

    // 最后的目的就是将 FastThreadLocal 放置到 Set 中
    variablesToRemove.add(variable);
}
```

这个方法的目的是将 FastThreadLocal 对象保存到一个 Set 中，因为 Netty 的 Map 只是一个数组，没有键，所以保存到一个 Set 中，这样就可以判断是否 set 过这个 map，例如 Netty 的 isSet 方法就是根据这个判断的。

说完了 setKnownNotUnset 方法，我们再说说 registerCleaner 方法。

3. registerCleaner(threadLocalMap);

这个方法可以说有点复杂了，请耐住性子，这里是 ftl（FastThreadLocal） 的精髓。

首先说下该方法的作用：将这个 ftl 注册到一个 清理线程 中，当 thread 对象被 gc 的时候，则会自动清理掉 ftl，防止 JDK 的内存泄漏问题。

让我们进入该方法查看：

```
private void registerCleaner(final InternalThreadLocalMap threadLocalMap) {
    Thread current = Thread.currentThread();
```

```
    if (FastThreadLocalThread.willCleanupFastThreadLocals(current) ||
        threadLocalMap.indexedVariable(cleanerFlagIndex) != InternalThreadLocalMap.UNSET) {
        return;
    }
    threadLocalMap.setIndexedVariable(cleanerFlagIndex, Boolean.TRUE);
    ObjectCleaner.register(current, new Runnable() {
        public void run() {
            remove(threadLocalMap);
        }
    });
}
```

楼主删除了源码中的注释，我们来好好说说这个方法：

1. 获取当前线程，如果当前线程是 FastThreadLocalThread 类型 且 cleanupFastThreadLocals 是 true，则返回 true，直接 return。也就是说，Netty 线程池里面创建的线程都符合这条件，只有用户自定义的线程池不符合。
   当然还有一个条件：如果这个 ftl 的 index + 1 在 map 中的值不是空对象，则已经注册过了，也直接 return，不再重复注册。
2. 当不符合上面的条件的时候，将 Map 中对应的 ftl 的 index + 1 位置的值设置为 TRUE。根据上面的判断，防止重复注册。
3. 调用 ObjectCleaner 的 register 方法，注册一个任务，任务的内容就是调用 remove 方法，删除 ftl 在 map 中的对象和相应的内容。

问题来了，怎么注册的呢？为什么还带着一个 current 当前线程呢？

我们看看源码：

```
public static void register(Object object, Runnable cleanupTask) {
    AutomaticCleanerReference reference = new AutomaticCleanerReference(object,
            ObjectUtil.checkNotNull(cleanupTask, "cleanupTask"));
    LIVE_SET.add(reference);

    // Check if there is already a cleaner running.
    if (CLEANER_RUNNING.compareAndSet(false, true)) {
        final Thread cleanupThread = new FastThreadLocalThread(CLEANER_TASK);
        cleanupThread.setPriority(Thread.MIN_PRIORITY);
        AccessController.doPrivileged(new PrivilegedAction<Void>() {
            public Void run() {
                cleanupThread.setContextClassLoader(null);
                return null;
            }
        });
        cleanupThread.setName(CLEANER_THREAD_NAME);
        cleanupThread.setDaemon(true);
        cleanupThread.start();
    }
}
```

首先创建一个 AutomaticCleanerReference 自动清洁对象，继承了 WeakReference，先不看他的构造方法，先看下面，将这个构造好的实例放入到 LIVE_SET 中，实际上，这是一个 Netty 封装的 ConcurrentSet，然后判断清除线程是否在运行。如果没有，并且CAS改状态成功。就创建一个线程，任务是 定义好的 CLEANER_TASK，线程优先级是最低，上下位类加载器是 null，名字是 objectCleanerThread，并且是后台线程。然后启动这个线程。运行 CLEANER_TASK。

一步一步来看看。

首先 AutomaticCleanerReference 的构造方法如下：

```
private static final ReferenceQueue<Object> REFERENCE_QUEUE = new ReferenceQueue<Object>();

AutomaticCleanerReference(Object referent, Runnable cleanupTask) {
    super(referent, REFERENCE_QUEUE);
    this.cleanupTask = cleanupTask;
}

void cleanup() {
   cleanupTask.run();
}
```

ReferenceQueue 的作用是，当对象被回收的时候，会将这个对象添加进这个队列，就可以跟踪这个对象。设置可以复活这个对象。也就是说，当这个 Thread 对象被回收的时候，会将这个对象放进这个引用队列，放进入干嘛呢？什么时候取出来呢？我们看看什么时候取出来：

代码如下：

```
private static final Runnable CLEANER_TASK = new Runnable() {
    @Override
    public void run() {
        for (;;) {
            while (!LIVE_SET.isEmpty()) {
                final AutomaticCleanerReference reference = (AutomaticCleanerReference) REFERENCE_QUEU

                if (reference != null) {
                    try {
                        reference.cleanup();
                    } catch (Throwable ignored) {
                    }
                    LIVE_SET.remove(reference);
                }
            }
            CLEANER_RUNNING.set(false);
            if (LIVE_SET.isEmpty() || !CLEANER_RUNNING.compareAndSet(false, true)) {
                break;
            }
        }
    }
};
```

巧了！！！！正是 CLEANER_TASK 在使用这个 ReferenceQueue！！！！别激动，我们还是慢慢看看这个任务到底是做什么的：

1. 死循环，如果 ConcurrentSet 不是空（还记得我们将 AutomaticCleanerReference 放进这里吗），尝试从 REFERENCE_QUEUE 中取出 AutomaticCleanerReference，也就是我们刚刚放进入的。这是标准的跟踪 GC 对象的做法。因为当一个对象被 GC 时，会将保证这个对象的 Reference 放进指定的引用队列，这是 JVM 做的。
2. 如果不是空，就调用应用的 cleanUp 方法，也就是我们传进去的任务，什么任务？就是那个调用 ftl 的 remove 方法的任务。随后从 Set 中删除这个引用。
3. 如果 Set 是空的话，将清理线程状态（原子变量） 设置成 fasle。
4. 继续判断，如果Set 还是空，或者 Set 不是空 且 设置 CAS 设置状态为true 失败（说明其他线程改了这个状态）则跳出循环，结束线程。

有点懵？那我们就好好总结这里为什么这么做：

> 当我们在一个非 Netty 线程池创建的线程中使用 ftl 的时候，Netty 会注册一个垃圾清理线程（因为 Netty 线程池创建的线程最终都会执行 removeAll 方法，不会出现内存泄漏），用于清理这个线程这个 ftl 变量，从上面的代码中，我们知道，非 Netty 线程如果使用 ftl，Netty 仍然会借助 JDK 的 ThreadLocal，只是只借用一个槽位，放置 Netty 的 Map， Map 中再放置 Netty 的 ftl 。所以，在使用线程池的情况下可能会出现内存泄漏。**Netty** 为了解决这个问题，在每次使用新的 **ftl** 的时候，都将这个 **ftl** 注册到和线程对象绑定到一个 **GC** 引用上， 当这个线程对象被回收的时候，也会顺便清理掉他的 **Map** 中的 所有 **ftl**，解决了该问题，就像解决 **JDK Nio bug** 一样。

好，到这里，Netty 的 FastThreadLocal 的精华我们基本就全部吸取了。ftl 不仅快，而且安全。快在使用数组代替线性探测法的 Map，安全在每次线程回收的时候都清理 ftl，不用担心内存泄漏。

剩下的方法都是很简单的。我们一起看完吧

4. remove();

每次 Set 一个空对象的时候，就是调用remove 方法，我们看看该方法，源码如下：

```
public final void remove() {
     remove(InternalThreadLocalMap.getIfSet());
 }

 public static InternalThreadLocalMap getIfSet() {
     Thread thread = Thread.currentThread();
     if (thread instanceof FastThreadLocalThread) {
         return ((FastThreadLocalThread) thread).threadLocalMap();
     }
     return slowThreadLocalMap.get();

 }

 public final void remove(InternalThreadLocalMap threadLocalMap) {
     if (threadLocalMap == null) {
         return;
     }
     // 删除并返回 Map 数组中当前 ThreadLocal index 对应的 value
     Object v = threadLocalMap.removeIndexedVariable(index);
     // 从 Map 数组下标 0 的位置取出 Set ，并删除当前的 ThreadLocal
     removeFromVariablesToRemove(threadLocalMap, this);

     if (v != InternalThreadLocalMap.UNSET) {
         try {
             // 默认啥也不做，用户可以继承 FastThreadLocal 重定义这个方法。
             onRemoval((V) v);
         } catch (Exception e) {
```

```
                PlatformDependent.throwException(e);
            }
        }
    }
```

楼主将这3个方法都合并在一起了，首先获取当前线程的 threadLocalMap，然后就像注释中写的：删除 ftl 对应下标中 map 的 value，然后删除 map 下标0 处 Set 中的 ftl。防止 isSet 方法误判。最后，如果用户重写了 onRemoval 方法，就调用，默认是个空方法。用户可以重写 onRemoval 方法和 initialize 方法。

4. get 方法解析

get 方法就更简单了，代码如下：

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

    V value = initialize(threadLocalMap);
    registerCleaner(threadLocalMap);
    return value;
}
```

首先获取当前线程的map，然后根据 ftl 的 index 获取 value，然后返回，如果是空对象，也就是没有设置，则通过 initialize 返回，initialize 方法会将返回值设置到 map 的槽位中，并放进 Set 中。最后，尝试注册一个清洁器。

5. remove All方法解析

这个方法在 Netty 的默认线程的 finally 块中调用。代码如下：

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

    try {
        Object v = threadLocalMap.indexedVariable(variablesToRemoveIndex);
        if (v != null && v != InternalThreadLocalMap.UNSET) {
            @SuppressWarnings("unchecked")
            Set<FastThreadLocal<?>> variablesToRemove = (Set<FastThreadLocal<?>>) v;
            FastThreadLocal<?>[] variablesToRemoveArray =
                    variablesToRemove.toArray(new FastThreadLocal[variablesToRemove.size()]);
            for (FastThreadLocal<?> tlv: variablesToRemoveArray) {
                tlv.remove(threadLocalMap);
            }
        }
    } finally {
        InternalThreadLocalMap.remove();
    }
}
```

非常简单，首先获取当前线程map，然后获取 Set，将 Set 转成数组，遍历数组，调用 ftl 的 remove 方法。最后，删除线程中的 map 属性。

总结

现在我们来总结一下 FastThreadLocal 。

之所以称之为 Fast，因为没有使用 JDK 的使用线性探测法的 Map，如果你使用的是Netty 线程池工厂创建的线程，搭配 Netty 的 ftl，性能非常好，如果你使用自定义的线程，搭配 ftl，性能也会比 JDK 的好，注意： ftl 没有 JDK 的内存泄露的风险。

但做到这些不是没有代价的，由于每一个 ftl 都是一个唯一的下标，而这个下标是每次创建一个 ftl 对象都是递增 2，当你的下标很大，你的线程中的 Map 相应的也要增大，可以想象，如果创建了海量的 ftl 对象，这个数组的浪费是非常客观的。很明显，这是一种空间换时间的做法。

通常，ftl 都是静态对象，所以不会有我们假设的那么多。如果使用不当，确实会浪费大量内存。

但这个风险带来的好处是明显的，在楼主的机器上测试，ftl 的读取性能是 JDK 的 5 倍左右，写入的速度也要快 20% 左右。

FastThreadLocal 人如其名，快且安全！

今天就到这里，good luck！！！！