**Netty如何监控内存泄露**

一般而言，在Netty程序中都会采用池化的ByteBuf，也就是PooledByteBuf以提高程序性能。但是PooledByteBuf需要在使用完毕后手工释放，否则就会因为PooledByteBuf申请的内存空间没有归还进而造成内存泄露，最终OOM。而一旦泄露发生，在复杂的应用程序中找到未手工释放的ByteBuf并不是一个简单的活计，在没有工具辅助的情况只能白盒检查所有源码，效率无疑十分低下。

为了解决这个问题，Netty设计了专门的泄露检测接口用于实现对需要手动释放的资源对象的监控。

**JDK的弱引用和引用队列**

在分析Netty的泄露监控功能之前，先来复习下其中会用到的JDK知识：引用。

在java中存在4中引用类型，分别是强引用，软引用，弱引用，虚引用。

**强引用**

强引用，是我们写程序最经常使用的方式。比如一个将一个值赋给一个变量，那这个对象值就被该变量强引用了。除非设置为null，否则java的内存回收不会回收该对象。就算是内存不足异常发生也不会。

**软引用**

软引用所引用的对象会在java内存不足的时候，被gc回收。如果gc发生的时候，java的内存还充足则不会回收这个对象 使用的方式如下

* SoftReference ref = new SoftReference(new Date());
* Date tmp = ref.get(); //如果对象没有被回收，则这个get操作会返回初始化的值。如果被回收了之后，则返回null

**弱引用**

弱引用则比软引用更差一些。只要是gc发生的时候，弱引用的对象都会被回收。使用方式上和软引用类似，如下

* WeakReference re = new WeakReference(new Date());
* re.get();

**虚引用**

虚引用和前面的软引用、弱引用不同，它并不影响对象的生命周期。在java中用java.lang.ref.PhantomReference类表示。如果一个对象与虚引用关联，则跟没有引用与之关联一样，在任何时候都可能被垃圾回收器回收。

除了强引用之外，其余的引用都有一个引用队列可以与之配合。当java清理调用不必要的引用后，会将这个引用本身（不是引用指向的值对象）添加到队列之中。代码如下

ReferenceQueue**<**Date**>** queue **=** **new** ReferenceQueue**<>();**

WeakReference**<**Date**>** re **=** **new** WeakReference**<**Date**>(new** Date**(),** queue**);**

Reference**<?** **extends** Date**>** moved **=** queue**.**poll**();**

从上面的介绍可以看出引用队列的一个适用场景：**与弱引用或虚引用配合，监控一个对象是否被GC回收**。

**Netty的实现思路**

针对需要手动关闭的资源对象，Netty设计了一个接口io.netty.util.ResourceLeakTracker来实现对资源对象的追踪。该接口提供了一个release方法。在资源对象关闭需要调用release方法。如果从未调用release方法则被认为存在资源泄露。

该接口只有一个实现，就是io.netty.util.ResourceLeakDetector.DefaultResourceLeak，该实现继承了WeakReference。每一个DefaultResourceLeak会与一个需要监控的资源对象关联，同时关联着一个引用队列。

当资源对象被GC回收后，与之关联的DefaultResourceLeak就会进入引用队列。通过检查引用队列中的DefaultResourceLeak实例的状态（release方法的调用会导致状态变更），就能确定在资源对象被GC前，是否执行了手动关闭的相关方法，从而判断是否存在泄漏可能。

**代码实现**

**分配监控对象**

当进行ByteBuf的分配的时候，比如方法io.netty.buffer.PooledByteBufAllocator#newHeapBuffer，查看代码如下

**protected** ByteBuf **newHeapBuffer(int** initialCapacity**,** **int** maxCapacity**)** **{**

PoolThreadCache cache **=** threadCache**.**get**();**

PoolArena**<byte[]>** heapArena **=** cache**.**heapArena**;**

**final** ByteBuf buf**;**

**if** **(**heapArena **!=** **null)** **{**

buf **=** heapArena**.**allocate**(**cache**,** initialCapacity**,** maxCapacity**);**

**}** **else** **{**

buf **=** PlatformDependent**.**hasUnsafe**()** **?**

**new** UnpooledUnsafeHeapByteBuf**(this,** initialCapacity**,** maxCapacity**)** **:**

**new** UnpooledHeapByteBuf**(this,** initialCapacity**,** maxCapacity**);**

**}**

**return** toLeakAwareBuffer**(**buf**);**

**}**

当实际持有内存区域的ByteBuf生成，通过方法io.netty.buffer.AbstractByteBufAllocator#toLeakAwareBuffer(io.netty.buffer.ByteBuf)加持监控泄露的能力。该方法代码如下

**protected** **static** ByteBuf **toLeakAwareBuffer(**ByteBuf buf**)** **{**

ResourceLeakTracker**<**ByteBuf**>** leak**;**

**switch** **(**ResourceLeakDetector**.**getLevel**())** **{**

**case** SIMPLE**:**

leak **=** AbstractByteBuf**.**leakDetector**.**track**(**buf**);**

**if** **(**leak **!=** **null)** **{**

buf **=** **new** SimpleLeakAwareByteBuf**(**buf**,** leak**);**

**}**

**break;**

**case** ADVANCED**:**

**case** PARANOID**:**

leak **=** AbstractByteBuf**.**leakDetector**.**track**(**buf**);**

**if** **(**leak **!=** **null)** **{**

buf **=** **new** AdvancedLeakAwareByteBuf**(**buf**,** leak**);**

**}**

**break;**

**default:**

**break;**

**}**

**return** buf**;**

**}**

根据不同的监控级别生成不同的监控等级对象。Netty对监控分为4个等级：

1. 关闭：这种模式下不进行泄露监控。
2. 简单：这种模式下以1/128的概率抽取ByteBuf进行泄露监控。默认。
3. 增强：在简单的基础上，每一次对ByteBuf的调用都会尝试记录调用轨迹，消耗较大。
4. 偏执：在增强的基础上，对每一个ByteBuf都进行泄露监控，消耗最大。

一般而言，在项目的初期使用简单模式进行监控，如果没有问题一段时间后就可以关闭。否则升级到增强或者偏执模式尝试确认泄露位置。

**追踪和检查泄露**

泄露的检查和追踪主要依靠两个类io.netty.util.ResourceLeakDetector.DefaultResourceLeak和io.netty.util.ResourceLeakDetector.前者用于追踪一个资源对象，并且记录对应的调用轨迹；后者则负责管理和生成DefaultResourceLeak对象。

**DefaultResourceLeak**

首先来看用于追踪资源对象的监控对象。该类继承了WeakReference，有几个重要的属性，如下

//存储着最新的调用轨迹信息，record内部通过next指针形成一个单向链表

**private** **volatile** Record head**;**

//调用轨迹不会无限制的存储，有一个上限阀值。超过了阀值会抛弃掉一些调用轨迹信息。

**private** **volatile** **int** droppedRecords**;**

//存储着所有的追踪对象，用于确认追踪对象是否处于可用。

**private** **final** Set**<**DefaultResourceLeak**<?>>** allLeaks**;**

//记录追踪对象的hash值，用于后续操作中的对象对比。

**private** **final** **int** trackedHash**;**

这个类的作用有三个：

1. 调用record方法记录调用轨迹
2. 调用close方法结束追踪
3. 以及本身作为WeakReference，在追踪对象被GC回收后自身被入列到ReferenceQueue中。

先来看下record方法，代码如下

@Override

**public** **void** **record()** **{**

record0**(null);**

**}**

@Override

**public** **void** **record(**Object hint**)** **{**

record0**(**hint**);**

**}**

**private** **void** **record0(**Object hint**)** **{**

**if** **(**TARGET\_RECORDS **>** 0**)** **{**

Record oldHead**;**

Record prevHead**;**

Record newHead**;**

**boolean** dropped**;**

**do** **{**

**if** **((**prevHead **=** oldHead **=** headUpdater**.**get**(this))** **==** **null)** **{**

// already closed.

**return;**

**}**

**final** **int** numElements **=** oldHead**.**pos **+** 1**;**

**if** **(**numElements **>=** TARGET\_RECORDS**)** **{**

**final** **int** backOffFactor **=** Math**.**min**(**numElements **-** TARGET\_RECORDS**,** 30**);**

**if** **(**dropped **=** PlatformDependent**.**threadLocalRandom**().**nextInt**(**1 **<<** backOffFactor**)** **!=** 0**)** **{**

prevHead **=** oldHead**.**next**;**

**}**

**}** **else** **{**

dropped **=** **false;**

**}**

newHead **=** hint **!=** **null** **?** **new** Record**(**prevHead**,** hint**)** **:** **new** Record**(**prevHead**);**

**}** **while** **(!**headUpdater**.**compareAndSet**(this,** oldHead**,** newHead**));**

**if** **(**dropped**)** **{**

droppedRecordsUpdater**.**incrementAndGet**(this);**

**}**

**}**

**}**

方法record0的思路总结下也很简单，概括如下：

1. 使用CAS方式当前的调用轨迹对象Record设置为head属性的值。
2. Record对象中的pos属性记录着当前轨迹链的长度，当追踪对象的轨迹队链的长度超过配置值时，有一定的几率（1-1/2min(n-target\_record,30)）将最新的轨迹对象从链条中删除。
3. CAS成功后，如果有抛弃头部的轨迹对象，则抛弃计数+1。

步骤2中在链条过长时选择删除最新的轨迹对象是基于以下两点出发：

1. 一般泄漏都发生在最后一次使用后忘记调用释放方法造成，因此替换最新的归集对象，并不会造成判断信息的丢失
2. 一般而言，关注泄漏对象，也需要了解对象实例的申请位置，因此删除节点时不能从头开始删除。

在来看看close方法。代码如下

**public** **boolean** **close(**T trackedObject**)** **{**

**assert** trackedHash **==** System**.**identityHashCode**(**trackedObject**);**

**try** **{**

**return** close**();**

**}** **finally** **{**

reachabilityFence0**(**trackedObject**);**

**}**

**}**

**public** **boolean** **close()** **{**

**if** **(**allLeaks**.**remove**(this))** **{**

// Call clear so the reference is not even enqueued.

clear**();**

headUpdater**.**set**(this,** **null);**

**return** **true;**

**}**

**return** **false;**

**}**

**private** **static** **void** **reachabilityFence0(**Object ref**)** **{**

**if** **(**ref **!=** **null)** **{**

**synchronized** **(**ref**)** **{**

**}**

**}**

**}**

close方法本身没有什么，就是将资源进行了清除。需要解释的是方法reachabilityFence0。不过该方法需要在下文的报告泄露中才会具备作用，这边先暂留。

**ResourceLeakDetector**

该类用于按照规则进行追踪对象的生成，外部主要是调用其方法track，代码如下

**public** **final** ResourceLeakTracker**<**T**>** **track(**T obj**)** **{**

**return** track0**(**obj**);**

**}**

**private** DefaultResourceLeak **track0(**T obj**)** **{**

Level level **=** ResourceLeakDetector**.**level**;**

**if** **(**level **==** Level**.**DISABLED**)** **{**

**return** **null;**

**}**

**if** **(**level**.**ordinal**()** **<** Level**.**PARANOID**.**ordinal**())** **{**

**if** **((**PlatformDependent**.**threadLocalRandom**().**nextInt**(**samplingInterval**))** **==** 0**)** **{**

reportLeak**();**

**return** **new** DefaultResourceLeak**(**obj**,** refQueue**,** allLeaks**);**

**}**

**return** **null;**

**}**

reportLeak**();**

**return** **new** DefaultResourceLeak**(**obj**,** refQueue**,** allLeaks**);**

**}**

从生成策略来看，只要是小于PARANOID级别都是抽样生成。生成的追踪对象上一个章节已经分析过了，这边主要来看reportLeak方法，如下

**private** **void** **reportLeak()** **{**

**if** **(!**logger**.**isErrorEnabled**())** **{**

clearRefQueue**();**

**return;**

**}**

// Detect and report previous leaks.

**for** **(;;)** **{**

@SuppressWarnings**(**"unchecked"**)**

DefaultResourceLeak ref **=** **(**DefaultResourceLeak**)** refQueue**.**poll**();**

**if** **(**ref **==** **null)** **{**

**break;**

**}**

//返回true意味着资源没有调用close或者dispose方法结束追踪就被GC了，意味着该资源存在泄漏。

**if** **(!**ref**.**dispose**())** **{**

**continue;**

**}**

String records **=** ref**.**toString**();**

**if** **(**reportedLeaks**.**putIfAbsent**(**records**,** Boolean**.**TRUE**)** **==** **null)** **{**

**if** **(**records**.**isEmpty**())** **{**

reportUntracedLeak**(**resourceType**);**

**}** **else** **{**

reportTracedLeak**(**resourceType**,** records**);**

**}**

**}**

**}**

**}**

**boolean** io**.**netty**.**util**.**ResourceLeakDetector**.**DefaultResourceLeak#dispose**()** **{**

clear**();**

**return** allLeaks**.**remove**(this);**

**}**

可以看到，每次生成资源追踪对象时，都会遍历引用队列，如果发现泄漏对象，则进行日志输出。

这里面有个细节的设计点在于DefaultResourceLeak进入引用队列并不意味着一定内存泄露。判断追踪对象是否泄漏的规则是对象在被GC之前是否调用了DefaultResourceLeak的close方法。举个例子，PooledByteBuf只要将自身持有的内存释放回池化区就算是正确的释放，其后其实例对象可以被GC回收掉。

因此方法reportLeak在遍历引用队列时，需要通过调用dispose方法来确认追踪对象的dispose是否调用或者close方法是否被调用过。如果dispose方法返回true，则意味着被追踪对象未调用关闭方法就被GC，那就意味着造成了泄露。

上个章节曾提到的一个方法reachabilityFence0。

在JVM的规定中，如果一个实例对象不再被需要，则可以判定为可回收。即使该实例对象的一个具体方法正在执行过程中，也是可以的。更确切一些的说，如果一个实例对象的方法体中，不再需要读取或者写入实例对象的属性，则此时JVM可以回收该对象，即使方法还没有完成。

然而这样会导致一个问题，在close方法中，如果close方法还没有执行完毕，trackedObject对象实例就被GC回收了，就会导致DefaultResourceLeak对象被加入到引用队列中，从而可能在reportLeak方法调用中触发方法dispose，假设此时close方法才刚开始执行，则dispose方法可能返回true。程序就会判定这个对象出现了泄露，然而实际上却没有。

要解决这个问题，只需要让close方法执行完毕前，让对象不要回收即可。reachabilityFence0方法就完成了这个作用。