二

30 实践总结：Netty 在项目开发中的一些最佳实践

这是专栏的最后一节课，首先恭喜你持之以恒学习到现在，你已经离成为一个 Netty 高手不远啦！本节课我会结合自身的实践经验，整理出一些 Netty 的最佳实践，帮助你回顾之前课程的知识点以及进一步提升 Netty 的进阶技巧。

本节课我们的内容以知识点列表的方式呈现，仅仅对 Netty 的核心要点进行提炼，更多详细的实现原理需要你课后深入研究源码。

性能篇

网络参数优化

Netty 提供了 ChannelOption 以便于我们优化 TCP 参数配置，为了提高网络通信的吞吐量，一些可选的网络参数我们有必要掌握。在之前的课程中我们已经介绍了一些常用的参数，我们在此基础上再做一些详细地扩展。

- **SO_SNDBUF/SO_RCVBUF**

TCP 发送缓冲区和接收缓冲区的大小。为了能够达到最大的网络吞吐量，SO_SNDBUF 不应当小于带宽和时延的乘积。SO_RCVBUF 一直会保存数据到应用进程读取为止，如果 SO_RCVBUF 满了，接收端会通知对端 TCP 协议中的窗口关闭，保证 SO_RCVBUF 不会溢出。

SO_SNDBUF/SO_RCVBUF 大小的设置建议参考消息的平均大小，不要按照最大消息来进行设置，这样会造成额外的内存浪费。更灵活的方式是可以动态调整缓冲区的大小，这时候就体现出 ByteBuf 的优势，Netty 提供的 ByteBuf 是可以支持动态调整容量的，而且提供了开箱即用的工具，例如可动态调整容量的接收缓冲区分配器 AdaptiveRecvByteBufAllocator。

- **TCP_NODELAY**

是否开启 Nagle 算法。Nagle 算法通过缓存的方式将网络数据包累积到一定量才会发送，从而避免频繁发送小的数据包。Nagle 算法 在海量流量的场景下非常有效，但是会造成一

定的数据延迟。如果对数据传输延迟敏感，那么应该禁用该参数。

- **SO_BACKLOG**

已完成三次握手的请求队列最大长度。同一时刻服务端可能会处理多个连接，在高并发海量连接的场景下，该参数应适当调大。但是 SO_BACKLOG 也不能太大，否则无法防止 SYN-Flood 攻击。

- **SO_KEEPALIVE**

连接保活。启用了 TCP SO_KEEPALIVE 属性，TCP 会主动探测连接状态，Linux 默认设置了 2 小时的心跳频率。TCP KEEPALIVE 机制主要用于回收死亡时间交长的连接，不适合实时性高的场景。

在海量连接的场景下，也许你会遇到类似 "too many open files" 的报错，所以 Linux 操作系统最大文件句柄数基本是必须要调优参数。可以通过 vi /etc/security/limits.conf，添加如下配置：

```
* soft nofile 1000000
* hard nofile 1000000
```

修改保存以后，执行 sysctl -p 命令使配置生效，然后通过 ulimit -a 命令查看参数是否生效。

业务线程池的必要性

Netty 是基于 Reactor 线程模型实现的，I/O 线程数量固定且资源珍贵，ChannelPipeline 负责所有事件的传播，如果其中任何一个 ChannelHandler 处理器需要执行耗时的操作，其中那么 I/O 线程就会出现阻塞，甚至整个系统都会被拖垮。所以推荐的做法是在 ChannelHandler 处理器中自定义新的业务线程池，将耗时的操作提交到业务线程池中执行。以 RPC 框架为例，在服务提供者处理 RPC 请求调用时就是将 RPC 请求提交到自定义的业务线程池中执行，如下所示：

```
public class RpcRequestHandler extends SimpleChannelInboundHandler<MiniRpcProtocol<
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, MiniRpcProtocol<MiniRpcR
        RpcRequestProcessor.submitRequest(() -> {
            // 处理 RPC 请求
```

```
        });
    }
}
```

共享 ChannelHandler

我们经常使用以下 new HandlerXXX() 的方式进行 Channel 初始化，在每建立一个新连接的时候会初始化新的 HandlerA 和 HandlerB，如果系统承载了 1w 个连接，那么就会初始化 2w 个处理器，造成非常大的内存浪费。

```
ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .localAddress(new InetSocketAddress(port))
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        public void initChannel(SocketChannel ch) {
                            ch.pipeline()
                                    .addLast(new HandlerA())
                                    .addLast(new HandlerB());
                        }
                    });
```

为了解决上述问题，Netty 提供了 @Sharable 注解用于修饰 ChannelHandler，标识该 ChannelHandler 全局只有一个实例，而且会被多个 ChannelPipeline 共享。所以我们必须要注意的是，@Sharable 修饰的 ChannelHandler 必须都是无状态的，这样才能保证线程安全。

设置高低水位线

高低水位线 WRITE_BUFFER_HIGH_WATER_MARK 和 WRITE_BUFFER_LOW_WATER_MARK 是两个非常重要的流控参数。Netty 每次添加数据时都会累加数据的字节数，然后判断缓存大小是否超过所设置的高水位线，如果超过了高水

位，那么 Channel 会被设置为不可写状态。直到缓存的数据大小低于低水位线以后，Channel 才恢复成可写状态。Netty 默认的高低水位线配置是 32K ~ 64K，可以根据发送端和接收端的实际情况合理设置高低水位线，如果你没有足够的测试数据作为参考依据，建议不要随意更改高低水位线。高低水位线的设置方式如下：

```
// Server
ServerBootstrap bootstrap = new ServerBootstrap();
bootstrap.childOption(ChannelOption.WRITE_BUFFER_HIGH_WATER_MARK, 32 * 1024);
bootstrap.childOption(ChannelOption.WRITE_BUFFER_LOW_WATER_MARK, 8 * 1024);
// Client
Bootstrap bootstrap = new Bootstrap();
bootstrap.option(ChannelOption.WRITE_BUFFER_HIGH_WATER_MARK, 32 * 1024);
bootstrap.option(ChannelOption.WRITE_BUFFER_LOW_WATER_MARK, 8 * 1024);
```

当缓存超过了高水位，Channel 会被设置为不可写状态，调用 isWritable() 方法会返回 false。建议在 Channel 写数据之前，使用 isWritable() 方法来判断缓存水位情况，防止因为接收方处理较慢造成 OOM。推荐的使用方式如下：

```
if (ctx.channel().isActive() && ctx.channel().isWritable()) {
    ctx.writeAndFlush(message);
} else {
    // handle message
}
```

GC 参数优化

对不同场景下的网络应用程序进行 JVM 参数调优，可以取得很好的性能提升，以及避免 OOM 风险。因为不同业务系统的特性是不一样的，在此我只能给你分享一些重要的注意事项。

- **堆内存**：-Xms 和 -Xmx 参数，-Xmx 用于控制 JVM Heap 的最大值，必须设置其大小，合理调整 -Xmx 有助于降低 GC 开销，提升系统吞吐量。-Xms 表示 JVM Heap 的初始值，对于生产环境的服务端来说 -Xms 和 -Xmx 最好设置为相同值。
- **堆外内存**：DirectByteBuffer 最容易造成 OOM 的情况，DirectByteBuffer 对象的回收需

要依赖 Old GC 或者 Full GC 才能触发清理。如果长时间没有 Old GC 或者 Full GC 执行，那么堆外内存即使不再使用，也会一直在占用内存不释放。我们最好通过 JVM 参数 -XX:MaxDirectMemorySize 指定堆外内存的上限大小，当堆外内存的大小超过该阈值时，就会触发一次 Full GC 进行清理回收，如果在 Full GC 之后还是无法满足堆外内存的分配，那么程序将会抛出 OOM 异常。

- **年轻代**：-Xmn 调整新生代大小，-XX:SurvivorRatio 设置 SurvivorRatio 和 eden 区比例。我们经常遇到 YGC 频繁的情况，应该清楚程序中对象的基本分布情况，如果存在大量朝生夕灭的对象，应适当调大新生代；反之应适当调大老年代。例如在类似百万长连接、推送服务等延迟敏感的场景中，老年代的内存增长缓慢，优化年轻代的空间大小以及各区的比例可以带来更大的收益。

内存池 & 对象池

从内存分配的角度来看，ByteBuf 可以分为堆内存 HeapByteBuf 和堆外内存 DirectByteBuf。DirectByteBuf 相比于 HeapByteBuf，虽然分配和回收的效率较慢，但是在 Socket 读写时可以少一次内存拷贝，性能更佳。

为了减少堆外内存的频繁创建和销毁，Netty 提供了池化类型的 PooledDirectByteBuf。Netty 提前申请一块连续内存作为 ByteBuf 内存池，如果有堆外内存申请的需求直接从内存池里获取即可，使用完之后必须重新放回内存池，否则会造成严重的内存泄漏。Netty 中启用内存池可以在创建客户端或者服务端的时候指定，示例代码如下：

```
bootstrap.option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);

bootstrap.childOption(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);
```

对象池与内存池的都是为了提高 Netty 的并发处理能力，通常在项目开发中我们会将一些通用的对象缓存起来，当需要该对象时，优先从对象池中获取对象实例。通过重用对象，不仅避免频繁地创建和销毁所带来的性能损耗，而且对 JVM GC 是友好的。如果你是一个高性能的网络应用系统，不妨试下 Netty 提供的 Recycler 对象池。Recycler 对象池如何使用在之前的课程有介绍过，在此我们一起回顾下。假设我们有一个 User 类，需要实现 User 对象的复用，具体实现代码如下：

```
public class UserCache {

    private static final Recycler<User> userRecycler = new Recycler<User>() {

        @Override

        protected User newObject(Handle<User> handle) {

            return new User(handle);
```

```
        }
    };
    static final class User {
        private String name;
        private Recycler.Handle<User> handle;
        public void setName(String name) {
            this.name = name;
        }
        public String getName() {
            return name;
        }
        public User(Recycler.Handle<User> handle) {
            this.handle = handle;
        }
        public void recycle() {
            handle.recycle(this);
        }
    }
    public static void main(String[] args) {
        User user1 = userRecycler.get(); // 1、从对象池获取 User 对象
        user1.setName("hello"); // 2、设置 User 对象的属性
        user1.recycle(); // 3、回收对象到对象池
        User user2 = userRecycler.get(); // 4、从对象池获取对象
        System.out.println(user2.getName());
        System.out.println(user1 == user2);
    }
}ß
```

由此可见，Netty 内存池和 Recycler 对象池优化的核心目标都是为了减少资源分配的开销，避免大量朝生夕灭的对象造成严重的内存消耗和 GC 压力。关于内存池和对象池的原理可以复习下之前课程《举一反三：Netty 高性能内存管理设计》《轻量级对象回收站：Recycler 对象池技术解析》，值得我们反复消化理解。

Native 支持

从 4.0.16 版本起，Netty 提供了用 C++ 编写 JNI 调用的 Socket Transport，相比 JDK NIO 具备更高的性能和更低的 GC 成本，并且支持更多的 TCP 参数。

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-transport-native-epoll</artifactId>
    <version>4.1.42.Final</version>
</dependency>
```

使用 Netty Native 非常简单，只需要替换相应的类即可：

NIO	Epoll
NioEventLoopGroup	EpollEventLoopGroup
NioEventLoop	EpollEventLoop
NioServerSocketChannel	EpollServerSocketChannel
NioSocketChannel	EpollSocketChannel

线程绑定

如果是经常关注系统性能调优，一定挖掘过 Linux 操作系统 CPU 亲和性的黑科技招数。CPU 亲和性是指在多核 CPU 的机器上线程可以被强制运行在某个 CPU 上，而不会调度到其他 CPU，也被称为绑核。当绑定线程到某个固定的 CPU 后，不仅可以避免 CPU 切换的开销，而且可以提高 CPU Cache 命中率，对系统性能有一定提升。

在 C/C++、Golang 中实现绑核操作是非常容易的事，遗憾的是在 Java 中是比较麻烦的。

目前 Java 中有一个开源 affinity 类库，GitHub 地址https://github.com/OpenHFT/Java-Thread-Affinity。如果你的项目想引入使用它，需要先引入 Maven 依赖：

```
<dependency>
    <groupId>net.openhft</groupId>
    <artifactId>affinity</artifactId>
    <version>3.0.6</version>
</dependency>
```

affinity 类库可以和 Netty 轻松集成，比较常用的方式是创建一个 AffinityThreadFactory，然后传递给 EventLoopGroup，AffinityThreadFactory 负责创建 Worker 线程并完成绑核。代码实现如下所示：

```
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
ThreadFactory threadFactory = new AffinityThreadFactory("worker", AffinityStrategie
EventLoopGroup workerGroup = new NioEventLoopGroup(4, threadFactory);
ServerBootstrap serverBootstrap = new ServerBootstrap().group(bossGroup, workerGrou
```

高可用篇

连接空闲检测 + 心跳检测

连接空闲检测是指每隔一段时间检测连接是否有数据读写，如果服务端一直能收到客户端连接发送过来的数据，说明连接处于活跃状态，对于假死的连接是收不到对端发送的数据的。如果一段时间内没收到客户端发送的数据，并不能说明连接一定处于假死状态，有可能客户端就是长时间没有数据需要发送，但是建立的连接还是健康状态，所以服务端还需要通过心跳检测的机制判断客户端是否存活。

客户端可以定时向服务端发送一次心跳包，如果有 N 次没收到心跳数据，可以判断当前客户端已经下线或处于不健康状态。由此可见，连接空闲检测和心跳检测是应对连接假死的一种有效手段，通常空闲检测时间间隔要大于 2 个周期的心跳检测时间间隔，主要是为了排除网络抖动的造成心跳包未能成功收到。

TCP 中已经有 SO_KEEPALIVE 参数，为什么我们还要在应用层加入心跳机制呢？心跳机制不仅能说明应用程序是活跃状态，更重要的是可以判断应用程序是否还在正常工作。然而 TCP KEEPALIVE 是有严重缺陷的，KEEPALIVE 设计初衷是为了清除和回收处于死亡状态

的连接，实时性不高。KEEPALIVE 只能检查连接是否活跃，但是不能判断连接是否可用，例如服务端如果处于高负载假死状态，但是连接依然是处于活跃状态的。

解码器保护

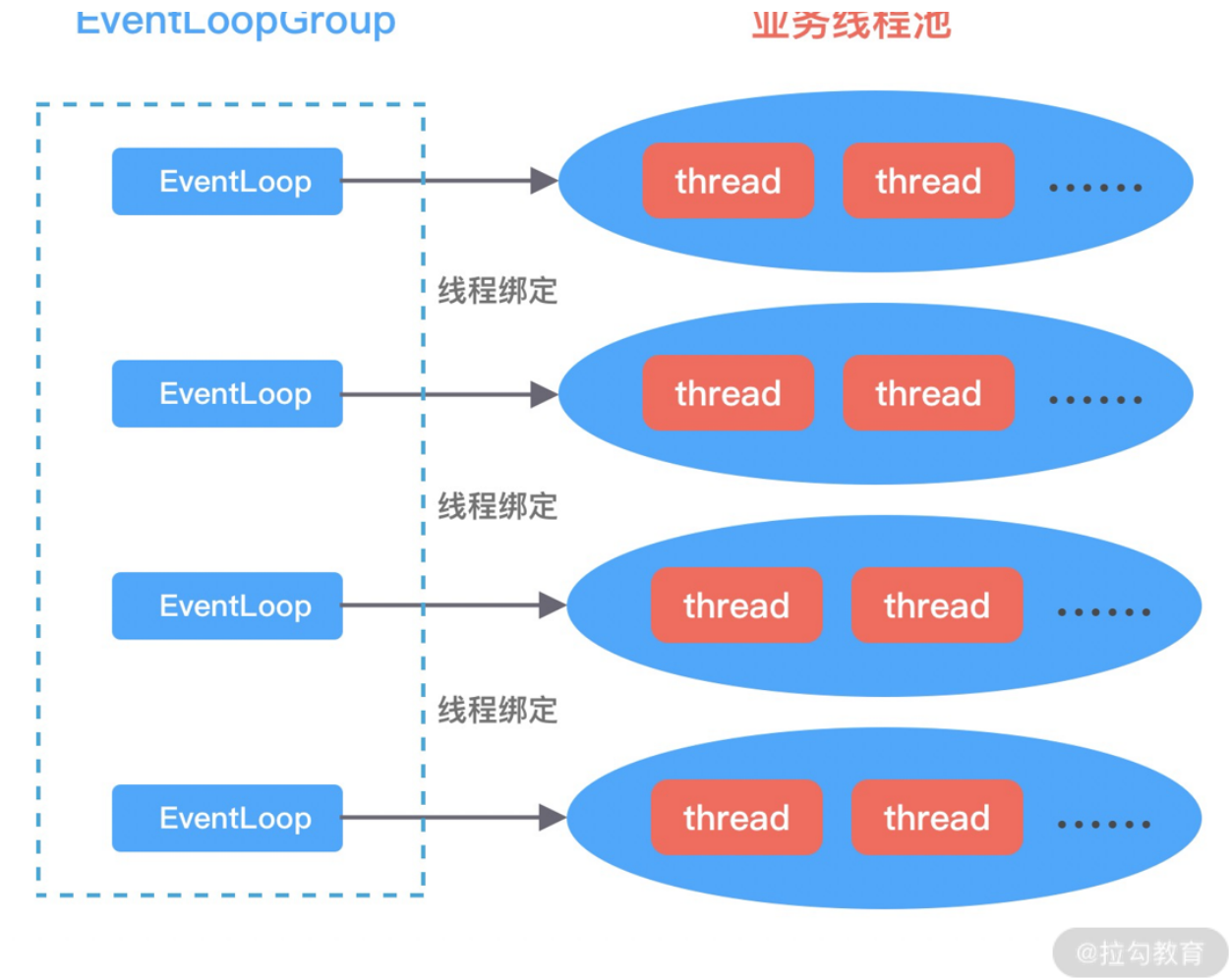
Netty 在实现数据解码时，需要等待到缓冲区有足够多的字节才能开始解码。为了避免缓冲区缓存太多数据造成内存耗尽，我们可以在解码器中设置一个最大字节的阈值，然后结合 Netty 提供的 TooLongFrameException 异常通知 ChannelPipeline 中其他 ChannelHandler。示例如下：

```
public class MyDecoder extends ByteToMessageDecoder {

    private static final int MAX_FRAME_LIMIT = 1024;

    @Override

    public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {

        int readable = in.readableBytes();

        if (readable > MAX_FRAME_LIMIT) {

            in.skipBytes(readable);

            throw new TooLongFrameException("too long frame");

        }

        // decode

    }

}
```

检测缓冲区可读字节是否大于 MAX_FRAME_LIMIT，如果超过忽略这些可读字节，对于应用程序在特定的场景下是一种有效的保护措施。

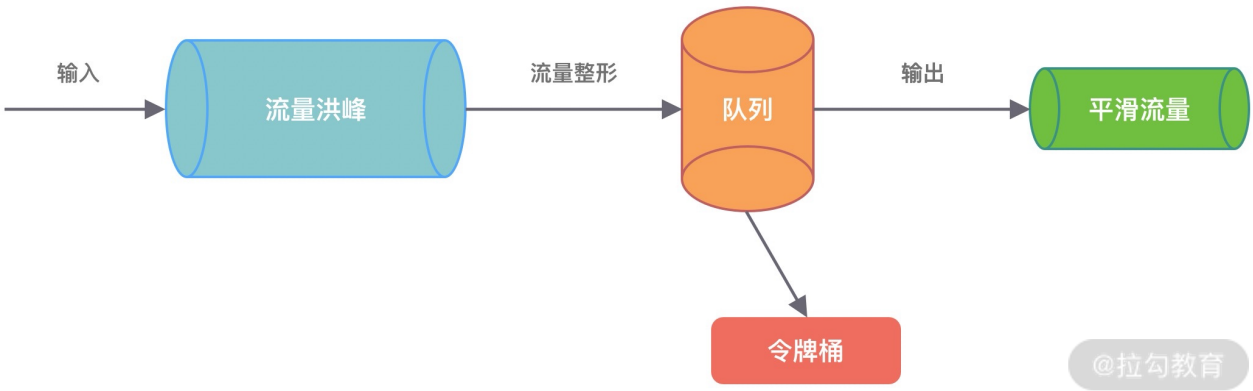
线程池隔离

我们知道，如果有复杂且耗时的业务逻辑，推荐的做法是在 ChannelHandler 处理器中自定义新的业务线程池，将耗时的操作提交到业务线程池中执行。建议根据业务逻辑的核心等级拆分出多个业务线程池，如果某类业务逻辑出现异常造成线程池资源耗尽，也不会影响到其他业务逻辑，从而提高应用程序整体可用率。对于 Netty I/O 线程来说，每个 EventLoop 可以与某类业务线程池绑定，避免出现多线程锁竞争。如下图所示：



流量整形

流量整形（Traffic Shaping）是一种主动控制服务流量输出速率的措施，保证下游服务能够平稳处理。流量整形和流控的区别在于，流量整形不会丢弃和拒绝消息，无论流量洪峰有多大，它都会采用令牌桶算法控制流量以恒定的速率输出，如下图所示。



Netty 通过实现流量整形的抽象类 AbstractTrafficShapingHandler，提供了三种类型的流量整形策略：GlobalTrafficShapingHandler、ChannelTrafficShapingHandler 和

GlobalChannelTrafficShapingHandler，它们之间的关系如下：

```
GlobalTrafficShapingHandler = ChannelTrafficShapingHandler + GlobalChannelTrafficSh
```

全局流量整形 GlobalChannelTrafficShapingHandler 作用范围是所有 Channel，用户可以设置全局报文的接收速率、发送速率、整形周期。Channel 级流量整形 ChannelTrafficShapingHandler 作用范围是单个 Channel，可以对不同的 Channel 设置流量整形策略。举个简单的例子，火爆的旅游景区不仅在大门口对游客限流（相当于 GlobalChannelTrafficShapingHandler），而且在景区内部不同的小景点也对游客有限流（相当于 ChannelTrafficShapingHandler），这两个流量整形策略加起来就是 GlobalTrafficShapingHandler。

流量整形并不能保证系统处于安全状态，当流量洪峰过大，数据会一直积压在内存中，所以流量整形和流控应该结合使用才能保证系统的高可用。

堆外内存泄漏排查思路

堆外内存泄漏问题是 Netty 应用程序的热点问题，经常遇到 Java 进程占用内存很高，但是堆内存并不高的情况。这里给你分享一些排查堆外内存泄漏的基本思路：

堆外内存回收

jmap -histo:live `<pid>` 手动触发 FullGC, 观察堆外内存是否被回收，如果正常回收很可能是因为堆外设置太小，可以通过 -XX:MaxDirectMemorySize 调整。当然这无法排除堆外内存缓慢泄漏的情况，需要借助其他工具进行分析。

堆外内存代码监控

前面的课程我们介绍过堆外内存回收原理，建议你再回过头复习下。JDK 默认采用 Cleaner 回收释放 DirectByteBuffer，Cleaner 继承于 PhantomReference，因为依赖 GC 进行处理，所以回收的时间是不可控的。对于 hasCleaner 的 DirectByteBuffer，Java 提供了一系列不同类型的 MXBean 用于获取 JVM 进程线程、内存等监控指标，代码实现如下：

```
BufferPoolMXBean directBufferPoolMXBean = ManagementFactory.getPlatformMXBeans(Buff

LOGGER.info("DirectBuffer count: {}, MemoryUsed: {} K", directBufferPoolMXBean.getC
```

对于 Netty 中 noCleaner 的 DirectByteBuffer，直接通过 PlatformDependent.usedDirectMemory() 读取即可。

Netty 自带检测工具

Netty 提供了自带的内存泄漏检测工具，我们可以通过以下命令启用堆外内存泄漏检测工具：

```
-Dio.netty.leakDetection.level=paranoid
```

Netty 一共提供了四种检测级别：

1. disabled，关闭堆外内存泄漏检测；
2. simple，以 1% 的采样率进行堆外内存泄漏检测，消耗资源较少，属于默认的检测级别；
3. advanced，以 1% 的采样率进行堆外内存泄漏检测，并提供详细的内存泄漏报告；
4. paranoid，追踪全部堆外内存的使用情况，并提供详细的内存泄漏报告，属于最高的检测级别，性能开销较大，常用于本地调试排查问题。

Netty 会检测 ByteBuf 是否已经不可达且引用计数大于 0，判定内存泄漏的位置并输出到日志中，你需要关注日志中 LEAK 关键字。

MemoryAnalyzer 内存分析

我们可以通过传统 Dump 内存的方法排查堆外内存泄漏问题，运行如下命令：

```
jmap -dump:format=b,file=heap.dump pid
```

Dump 完内存堆栈之后，将其导入 MemoryAnalyzer 工具进行分析内存泄漏的可疑点，最终定位到代码源头。关于如何 MemoryAnalyzer 工具我在此就不展开了，需要你自行学习研究，这是每一个 Java 程序员的必备技能。

Btrace 神器

Btrace 是一款通过字节码检测 Java 程序的排障神器，它可以获取程序在运行过程中的一切信息，与 AOP 的使用方式类似。我们可以通过如下方式追踪 DirectByteBuffer 的堆外内存申请的源头：

```
@BTrace
public class TraceDirectAlloc {

    @OnMethod(clazz = "java.nio.Bits", method = "reserveMemory")
```

```
    public static void printThreadStack() {
        jstack();
    }
}
```

二分排查法：笨方法解决大问题

堆外内存泄漏问题有时候非常隐蔽，并不是很容易定位发现。为了提高问题排查的效率，我们最好能够在本地模拟复现出堆外内存泄漏问题，如果本地能够成功复现，那么已经成功了一半了。

我们可以根据近期代码变更的记录，通过二分法对代码进行回滚，然后再次尝试是否可以复现出堆外内存泄漏问题，最终可以定位出有问题的代码 commit。该思路虽然是一种笨方法，但是很多场景下可以有效解决问题。

总结

以上都是项目实践中的一些重要技巧，对于我们上手 Netty 应用程序开发已经足够使用，还有更多 Netty 的技巧和使用心得需要我们去自己在实践中探索。纸上得来终觉浅，绝知此事要躬行，当你积累了丰富的经验，不管是项目开发还是问题排障，都会越来越得心应手。

上一页 下一页