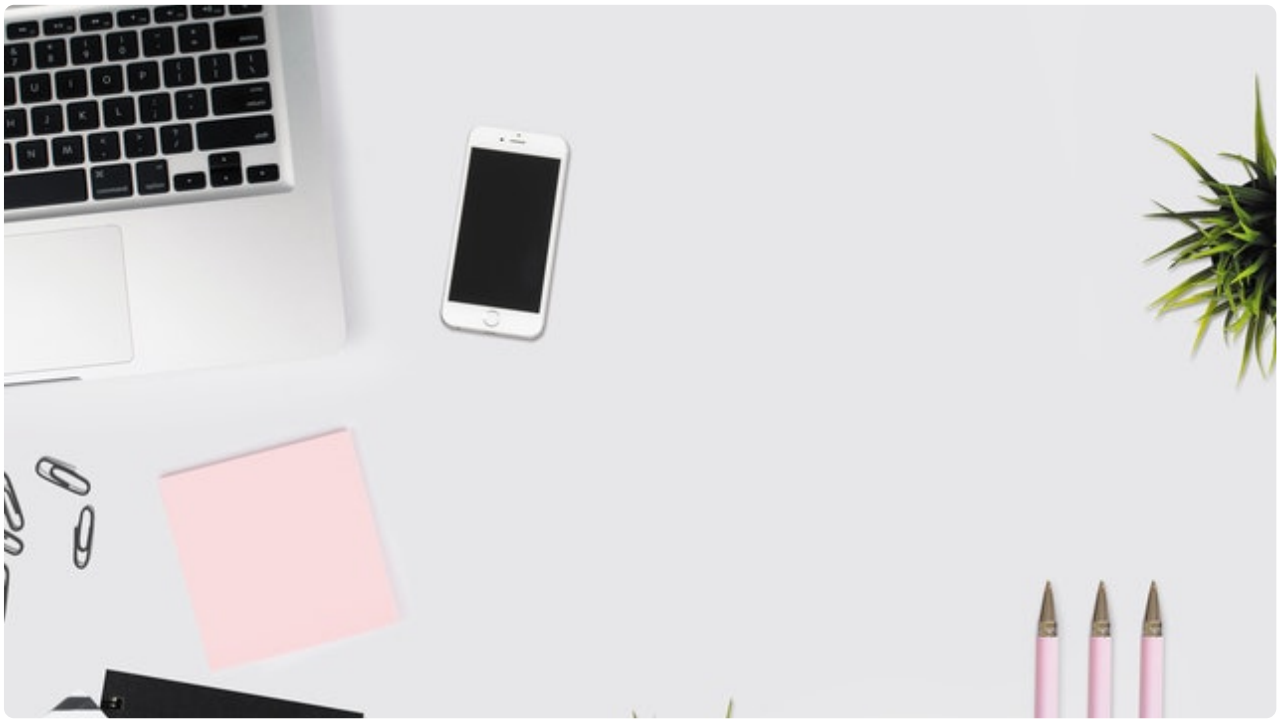
37 如何调优参数

更新时间：2020-08-27 10:06:15



> 我们有力的道德就是通过奋斗取得物质上的成功；这种道德既适用于国家，也适用于个人。——罗素

前言

你好，我是彤哥。

上一节，我们一起从安全性的角度对实战项目进行了改造，可以说，现在这个项目已经基本具有上线的标准了。

不过，还能够再压榨压榨，使它更健壮，那么，还有哪些手段呢？

本节，我们将从一些小的方面对实战项目进行调优，这些小的方面包括：

- 注解支持
- Native 支持
- 系统参数
- Netty 参数

其中，前两项跟参数关系不大，不过内容太零散，就放在这节一起了，请知悉。

好了，让我们进入今天的学习吧。

注解支持

说起 Netty 的注解，就不得不提 `@Sharable` 了，在 Netty 中，`@Sharable` 添加在 Handler 上，表示的是这个 Handler 没有线程安全的问题，可以被多个 Channel 共享，简单点讲，就是这个 Handler 我们可以创建为单例。

那么，该如何使用 `@Sharable` 注解呢？

其实，非常简单，加在 Handler 上面即可，这样，在往 Pipeline 中添加 Handler 的时候就可以提取出去添加为单例了，以 MahjongServerHandler 为例：

```
@ChannelHandler.Sharable
public class MahjongServerHandler extends SimpleChannelInboundHandler<MahjongProtocol> {
    // ...省略代码
}
public class MahjongServer {
    public static void main(String[] args) throws Exception {
        try {
            // 在外部创建一个实例
            MahjongServerHandler mahjongServerHandler = new MahjongServerHandler();

            ServerBootstrap serverBootstrap = new ServerBootstrap();

            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws Exception {
                    ChannelPipeline p = ch.pipeline();
                    // 添加到pipeline中
                    // 如果是在这里new，是每来一个Channel都会创建一个实例的
                    p.addLast(mahjongServerHandler);
                }
            });
        }
    }
}
```

我们编写的大部分 Handler 都应该是 Sharable 的，不应该在 Handler 中添加状态信息，这有点类似于 Spring 中的 Service，无状态才能作为单例来使用。

除了 @Sharable 注解，还有个 @Skip 注解，它是注解在方法上的，表示这个方法在 Pipeline 中执行的时候是跳过的，一般是放在适配类（父类）中使用，不过，一旦子类重写了这个方法，这个注解就失效了，比如 ChannelInboundHandlerAdapter 它的所有方法都加了这个注解：

```
public class ChannelInboundHandlerAdapter extends ChannelHandlerAdapter implements ChannelInboundHandler {
    @Skip
    @Override
    public void channelRegistered(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelRegistered();
    }
    @Skip
    @Override
    public void channelUnregistered(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelUnregistered();
    }
    @Skip
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelActive();
    }
    @Skip
    @Override
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelInactive();
    }
    @Skip
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        ctx.fireChannelRead(msg);
    }
    @Skip
    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelReadComplete();
    }
    @Skip
    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
        ctx.fireUserEventTriggered(evt);
    }
    @Skip
    @Override
    public void channelWritabilityChanged(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelWritabilityChanged();
    }
    @Skip
    @Override
    @SuppressWarnings("deprecation")
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
            throws Exception {
        ctx.fireExceptionCaught(cause);
    }
}
```

Native 支持

所谓 Native 支持，是指在不同平台开启对应平台的特性。

比如，Linux 系统有更优的多路复用模型 Epoll，那么，我们在部署到生产环境的时候完全可以修改为使用 Epoll。

> select 和 epoll 有什么区别呢？
>
> 简单点讲，select 是轮询的方式，epoll 是回调的方式。比如，有 1000 个连接，select 是一个一个去询问有没有准备好的 IO，epoll 是当有准备好的 IO 会主动通知它，所以，select 的时间复杂度是 O (n)，epoll 的时间复杂度是 O (1)，epoll 在连接多的情况下更高效。

好了，让我们看看 Netty 如何开启 Epoll 吧。

其实也非常简单，在 `netty-transport-native-epoll` 工程下面提供了一个 `Epoll` 类，它里面有一个 `isAvailable()` 方法判断是否支持 Epoll，所以，我们只需要简单的修改启动类就可以了：

```
public class MahjongServer {

    public static void main(String[] args) throws Exception {
        // 1. 声明线程池******
        EventLoopGroup bossGroup = Epoll.isAvailable()? new EpollEventLoopGroup(1) : new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = Epoll.isAvailable()? new EpollEventLoopGroup() : new NioEventLoopGroup();
        try {
            // 2. 服务端引导器
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            // 3. 设置线程池
            serverBootstrap.group(bossGroup, workerGroup)
                    // 4. 设置ServerSocketChannel的类型*****
                    .channel(Epoll.isAvailable()? EpollServerSocketChannel.class : NioServerSocketChannel.class);

            // ...省略其他代码
        } finally {

        }
    }
}
```

把创建 bossGroup、workerGroup 以及设置 channel 的地方修改为 Epoll 的相关实现即可，其他地方都不用修改，就能享受到 Linux 系统带来的性能提升，可以说是非常值得的。

系统参数

Netty 对系统参数的支持主要集中在 `ChannelOption` 及其子类中：

```
public static final ChannelOption<Boolean> SO_BROADCAST = valueOf("SO_BROADCAST");
public static final ChannelOption<Boolean> SO_KEEPALIVE = valueOf("SO_KEEPALIVE");
public static final ChannelOption<Integer> SO_SNDBUF = valueOf("SO_SNDBUF");
public static final ChannelOption<Integer> SO_RCVBUF = valueOf("SO_RCVBUF");
public static final ChannelOption<Boolean> SO_REUSEADDR = valueOf("SO_REUSEADDR");
public static final ChannelOption<Integer> SO_LINGER = valueOf("SO_LINGER");
public static final ChannelOption<Integer> SO_BACKLOG = valueOf("SO_BACKLOG");
public static final ChannelOption<Integer> SO_TIMEOUT = valueOf("SO_TIMEOUT");

public static final ChannelOption<Boolean> TCP_NODELAY = valueOf("TCP_NODELAY");
```

这些参数包括了 UDP、TCP 等各种各样的参数，这些参数都是标准参数，我这里挑几个比较重要的参数讲解一下。

SO_KEEPALIVE

TCP 的心跳机制，默认关闭，我们已经有了应用层的心跳检测机制，也不需要开启此参数。

为什么不直接使用 TCP 的心跳机制呢？

原因主要有以下三点：

- TCP 的 keepalive 只能检测连接是否存在，不能检测连接是否可用。比如一方负载过高导致无法响应请求但是连接还在，此时无法判断连接是否可用。
- 超时时间过长，默认超时时间为 120 分钟，太长了。

- 如果一方无故断了，另一方会尝试重传，重传消息的优先级是大于 keepalive 的，导致连接迟迟无法断开。

所以，还是使用应用层的心跳，自己控制比较好一些。

SO_BACKLOG

用于配置最大的等待连接数，Linux 系统默认值是 128，Windows 系统默认值为 200。

这个参数的主要作用是防止一次性过来太多客户端连接，服务器处理不过来的场景，比较典型的例子是服务端重启，所有客户端都要重新建立连接，为了防止惊群效应，可以把这个参数调大一些，比如 1024 等。

TCP_NODELAY

TCP/IP 协议中针对 TCP 默认开启了 Nagle 算法。Nagle 算法通过减少需要传输的数据包，来优化网络。在内核实现中，数据包的发送和接受会先做缓存，分别对应于写缓存和读缓存。

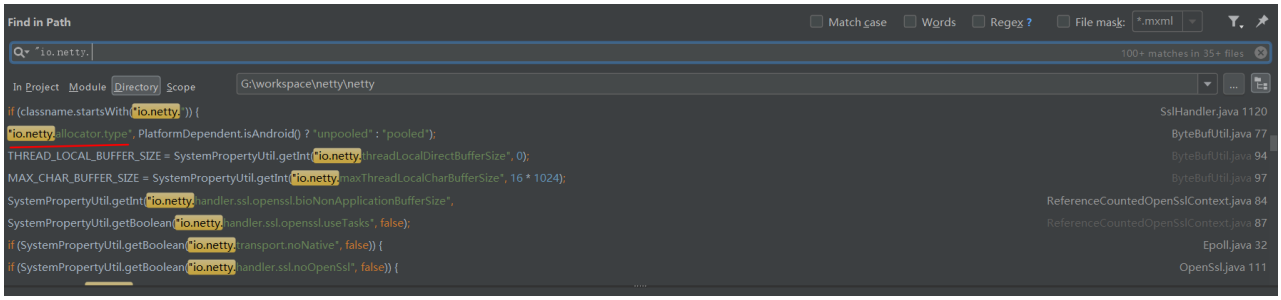
启动 TCP_NODELAY，就意味着禁用了 Nagle 算法，允许小包的发送。

这个参数默认为 false，可以理解为延迟写，如果有非常小的报文需要发送，则需要开启该参数。

好了，关于系统参数，我就介绍这么多，还有很多其它的参数，有兴趣的同学可以自行查阅。

Netty 参数

相比于 Netty 对系统参数的支持，Netty 本身的参数显然要凌乱的多，基本上是哪里使用哪里定义，使用 IDEA 的同学可以使用全局搜索（CTRL+SHIRT+F），输入 "io.netty.（前面加个引号）：



这部分参数可以配置在程序启动参数上，比如：-Dio.netty.allocator.type=unpooled。

另外，还有一部分参数在 ChannelOption 中：

```
public static final ChannelOption<ByteBufAllocator> ALLOCATOR = valueOf("ALLOCATOR");
public static final ChannelOption<RecvByteBufAllocator> RCVBUF_ALLOCATOR = valueOf("RCVBUF_ALLOCATOR");
public static final ChannelOption<MessageSizeEstimator> MESSAGE_SIZE_ESTIMATOR = valueOf("MESSAGE_SIZE_ESTIMATOR");
public static final ChannelOption<Integer> CONNECT_TIMEOUT_MILLIS = valueOf("CONNECT_TIMEOUT_MILLIS");
public static final ChannelOption<Integer> WRITE_SPIN_COUNT = valueOf("WRITE_SPIN_COUNT");
public static final ChannelOption<WriteBufferWaterMark> WRITE_BUFFER_WATER_MARK = valueOf("WRITE_BUFFER_WATER_MARK");
public static final ChannelOption<Boolean> ALLOW_HALF_CLOSURE = valueOf("ALLOW_HALF_CLOSURE");
public static final ChannelOption<Boolean> AUTO_READ = valueOf("AUTO_READ");
public static final ChannelOption<Boolean> AUTO_CLOSE = valueOf("AUTO_CLOSE");
public static final ChannelOption<Boolean> SINGLE_EVENTEXECUTOR_PER_GROUP = valueOf("SINGLE_EVENTEXECUTOR_PER_GROUP");
```

对于这些参数，大部分情况下，我们都是不需要修改的，不过，我们还是应该做到大致了解，心中有数，这里我简单介绍几个：

io.netty.allocator.type

指定是否使用池化的分配器，默认值是 pooled，如果要使用非池化，则添加这个参数并赋值为 unpooled 即可。

io.netty.maxDirectMemory

设置 Netty 可使用的最大的直接内存空间，小于 0 表示它使用 JDK 的定义的直接内存大小，默认为 2 倍的堆内存大小，大于 0 表示 Netty 可使用的最大直接内存大小，它跟 JDK 可使用的没有任何关系，等于 0 表示它也使用 JDK 的直接内存大小，同时还会使用 JDK 的清理方式，即 Cleaner。

io.netty.noPreferDirect

不偏向于使用直接内存，默认为 false，即使用直接内存，注意这个参数的名称是 “不偏向于”，即使开启了该参数也不代表就不使用直接内存了。

io.netty.noUnsafe

不使用 Unsafe，默认为 false，即使用 Unsafe，这里的不使用 Unsafe 只代表 Netty 不使用 Unsafe，若开启该参数，则 Netty 自己不会使用 Unsafe 去操作底层数组或者直接内存，但是它调用的 JDK 的类还是有可能会使用到 Unsafe 的，比如 Java 原生的直接内存 Buffer。

CONNECT_TIMEOUT_MILLIS

客户端连接超时时间，默认为 30 秒，30 秒有点长了，可以设置短一点，比如 10 秒。

好了，我们就简单介绍这么几个参数，其实，大多数参数都是不需要修改的，了解这些参数，也能更全面地了解 Netty，在看源码的时候看到这些参数也会会心一笑。

后记

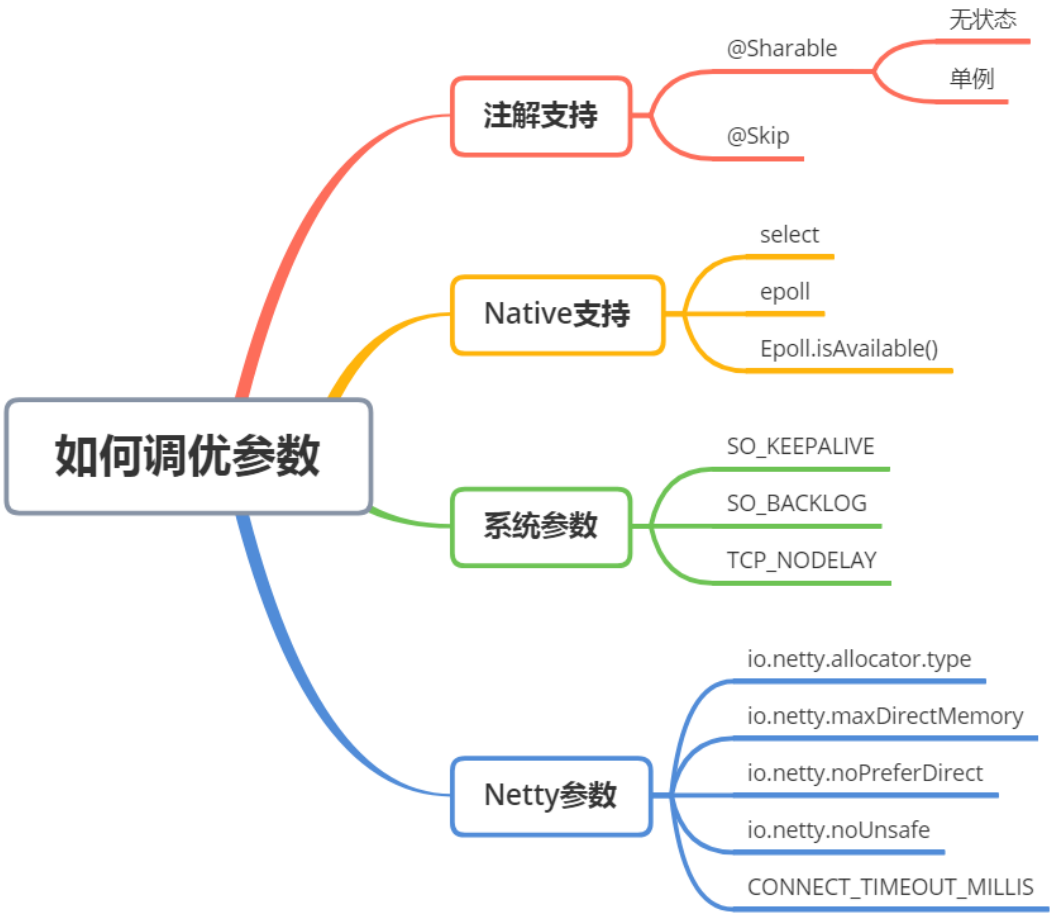
本节，我们一起从注解支持、Native 支持、系统参数、Netty 参数等四个维度更深地挖掘了一下 Netty，通过进一步改造，也能更进一步提升 Netty 应用的性能。

到这里，我们的实战项目基本上算是达到了上线的标准，不过，你敢现在就把项目部署到生产环境吗？

我想，还可以做些什么事，毕竟现在应用对于我们还是个黑盒子，使用了多少内存？连接数有多少？对于这些问题，我们一无所知。

所以，我们还需要给应用加上监控，有了监控我们心里才底，因此，下一节，我将介绍如何对 Netty 应用进行监控，敬请期待。

思维导图



}

36 如何增加安全性

38 如何增加监控