服务端启动流程

这一小节，我们来学习一下如何使用 Netty 来启动一个服务端应用程序，以下是服务端启动的一个非常精简的 Demo:

NettyServer.java

```
public class NettyServer {
    public static void main(String[] args) {
        NioEventLoopGroup bossGroup = new
NioEventLoopGroup();
        NioEventLoopGroup workerGroup = new
NioEventLoopGroup();

        ServerBootstrap serverBootstrap = new
ServerBootstrap();
        serverBootstrap
                .group(bossGroup, workerGroup)

.channel(NioServerSocketChannel.class)
                .childHandler(new
ChannelInitializer<NioSocketChannel>() {
                    protected void
initChannel(NioSocketChannel ch) {
                    }
                });

        serverBootstrap.bind(8000);
    }
}
```

- 首先看到，我们创建了两个NioEventLoopGroup，这两个对象可以看做是传统IO编程模型的两大线程组，bossGroup表示监听端口，accept 新连接的线程组，workerGroup表示处理每一条连接的数据读写的线程组，不理解的同学可以看一下上一小节《Netty是什么》(https://juejin.im/book/5b4bc28bf265da0f60130116/s
用生活中的例子来讲就是，一个工厂要运作，必然要有一个老板负责从外面接活，然后有很多员工，负责具体干活，老板就是bossGroup，员工们就是workerGroup，bossGroup接收完连接，扔给workerGroup去处理。
- 接下来 我们创建了一个引导类 ServerBootstrap，这个类将引导我们进行服务端的启动工作，直接new出来开搞。
- 我们通过.group(bossGroup, workerGroup)给引导类配置两大线程组，这个引导类的线程模型也就定型了。
- 然后，我们指定我们服务端的 IO 模型为NIO，我们通过.channel(NioServerSocketChannel.class)来指定 IO 模型，当然，这里也有其他的选择，如果你想指定 IO 模型为 BIO，那么这里配置上OioServerSocketChannel.class类型即可，当然通常我们也不会这么做，因为Netty的优势就在于NIO。
- 接着，我们调用childHandler()方法，给这个引导类创建一个ChannelInitializer，这里主要就是定义后续每条连接的数据读写，业务处理逻辑，不理解没关系，在后面我们会详细分析。ChannelInitializer这个类中，我们注意到有一个泛型参数NioSocketChannel，这个类呢，就是 Netty 对 NIO 类型的连接的抽象，而我们前面NioServerSocketChannel也是对 NIO 类型的连接的抽象，NioServerSocketChannel和NioSocketChannel的概念可以和 BIO 编程模型中的ServerSocket以及Socket两个概念对应上

我们的最小化参数配置到这里就完成了，我们总结一下就是，要启动一个Netty服务端，必须要指定三类属性，分别是线程模型、IO 模型、连接读写处理逻辑，有了这三者，之后在调用bind(8000)，我们就可以在本地绑定一个 8000 端口启动起来，以上这段代码读者可以直接拷贝到你的 IDE 中运行。

自动绑定递增端口

在上面代码中我们绑定了 8000 端口，接下来我们实现一个稍微复杂一点的逻辑，我们指定一个起始端口号，比如 1000，然后呢，我们从1000号端口往上找一个端口，直到这个端口能够绑定成功，比如 1000 端口不可用，我们就尝试绑定 1001，然后 1002，依次类推。

serverBootstrap.bind(8000);这个方法呢，它是一个异步的方法，调用之后是立即返回的，他的返回值是一个ChannelFuture，我们可以给这个ChannelFuture添加一个监听器GenericFutureListener，然后我们在GenericFutureListener的operationComplete方法里面，我们可以监听端口是否绑定成功，接下来是监测端口是否绑定成功的代码片段

```
serverBootstrap.bind(8000).addListener(new
GenericFutureListener<Future<? super Void>>() {
    public void operationComplete(Future<? super
Void> future) {
        if (future.isSuccess()) {
            System.out.println("端口绑定成功!");
        } else {
            System.err.println("端口绑定失败!");
        }
    }
});
```

我们接下来从 1000 端口号，开始往上找端口号，直到端口绑定成功，我们要做的就是在 if (future.isSuccess())的else逻辑里面重新绑定一个递增的端口号，接下来，我们把这段绑定逻辑抽取出一个bind方法

```
private static void bind(final ServerBootstrap
serverBootstrap, final int port) {
    serverBootstrap.bind(port).addListener(new
GenericFutureListener<Future<? super Void>>() {
        public void operationComplete(Future<?
super Void> future) {
            if (future.isSuccess()) {
                System.out.println("端口[" + port
+ "]绑定成功!");
            } else {
                System.err.println("端口[" + port
+ "]绑定失败!");
                bind(serverBootstrap, port + 1);
            }
        }
    });
}
```

然后呢，以上代码中最关键的就是在端口绑定失败之后，重新调用自身方法，并且把端口号加一，然后，在我们的主流程里面，我们就可以直接调用

```
bind(serverBootstrap, 1000)
```

读者可以自定修改代码，运行之后可以看到效果，最终会发现，端口成功绑定了在1024，从 1000 开始到 1023，端口均绑定失败了，这是因为在我的 MAC 系统下，1023 以下的端口号都是被系统保留了，需要 ROOT 权限才能绑定。

以上就是自动绑定递增端口的逻辑，接下来，我们来一起学习一下，服务端启动，我们的引导类ServerBootstrap除了指定线程模型，IO 模型，连接读写处理逻辑之外，他还可以干哪些事情?

服务端启动其他方法

handler() 方法

```
serverBootstrap.handler(new
ChannelInitializer<NioServerSocketChannel>() {
    protected void
initChannel(NioServerSocketChannel ch) {
        System.out.println("服务端启动中");
    }
})
```

handler()方法呢，可以和我们前面分析的childHandler()方法对应起来，childHandler()用于指定处理新连接数据的读写处理逻辑，handler()用于指定在服务端启动过程中的一些逻辑，通常情况下呢，我们用不着这个方法。

attr() 方法

```
serverBootstrap.attr(AttributeKey.newInstance("se
rverName"), "nettyServer")
```

attr()方法可以给服务端的 channel，也就是NioServerSocketChannel指定一些自定义属性，然后我们可以通过channel.attr()取出这个属性，比如，上面的代码我们指定我们服务端channel的一个serverName属性，属性值为nettyServer，其实说白了就是给NioServerSocketChannel维护一个map而已，通常情况下，我们也用不上这个方法。

那么，当然，除了可以给服务端 channel NioServerSocketChannel指定一些自定义属性之外，我们还可以给每一条连接指定自定义属性

childAttr() 方法

```
serverBootstrap.childAttr(AttributeKey.newInstance("clientKey"), "clientValue")
```

上面的childAttr可以给每一条连接指定自定义属性，然后后续我们可以通过channel.attr()取出该属性。

childOption() 方法

```
serverBootstrap
        .childOption(ChannelOption.SO_KEEPALIVE, true)
        .childOption(ChannelOption.TCP_NODELAY, true)
```

childOption()可以给每条连接设置一些TCP底层相关的属性，比如上面，我们设置了两种TCP属性，其中

- ChannelOption.SO_KEEPALIVE表示是否开启TCP底层心跳机制，true为开启
- ChannelOption.TCP_NODELAY表示是否开启Nagle算法，true表示关闭，false表示开启，通俗地说，如果要求高实时性，有数据发送时就马上发送，就关闭，如果需要减少发送次数减少网络交互，就开启。

其他的参数这里就不一一讲解，有兴趣的同学可以去这个类里面自行研究。

option() 方法

除了给每个连接设置这一系列属性之外，我们还可以给服务端channel设置一些属性，最常见的就是so_backlog，如下设置

```
serverBootstrap.option(ChannelOption.SO_BACKLOG, 1024)
```

表示系统用于临时存放已完成三次握手的请求的队列的最大长度，如果连接建立频繁，服务器处理创建新连接较慢，可以适当调大这个参数

总结

- 本文中，我们首先学习了 Netty 服务端启动的流程，一句话来说就是：创建一个引导类，然后给他指定线程模型，IO模型，连接读写处理逻辑，绑定端口之后，服务端就启动起来了。
- 然后，我们学习到 bind 方法是异步的，我们可以通过这个异步机制来实现端口递增绑定。
- 最后呢，我们讨论了 Netty 服务端启动额外的参数，主要包括给服务端 Channel 或者客户端 Channel 设置属性值，设置底层 TCP 参数。

如果，你觉得这个过程比较简单，想深入学习，了解服务端启动的底层原理，可参考[这里](https://coding.imooc.com/class/chapter/230.html#Anchor)。