昵称：Tiny&zzh
园龄：5年2个月
粉丝：50
关注：15
+加关注

< 2017年3月 >

日	一	二	三	四	五	六
26	27	28	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

Netty(11)
silverlight(7)
java(7)
Unity3D(7)
中文教程(5)
Netty 4(5)
NIO(4)
Okra(3)
Socket(3)
C#(3)
更多

随笔分类

Java(4)
Netty(7)
Okra(3)
Unity3D(5)

随笔档案

2016年6月 (1)
2016年4月 (4)
2015年6月 (1)
2015年3月 (1)
2015年2月 (1)
2015年1月 (1)
2014年10月 (2)
2014年9月 (2)
2014年8月 (3)
2014年7月 (2)
2014年6月 (1)
2014年4月 (5)
2014年1月 (2)
2013年12月 (4)
2013年11月 (2)
2013年10月 (1)
2013年6月 (2)
2013年4月 (1)
2012年11月 (1)
2012年9月 (2)
2012年5月 (1)
2012年3月 (2)
2012年2月 (1)
2012年1月 (1)
2011年12月 (1)

相册

Netty4.x中文教程系列**(**六**)** 从头开始**Bootstrap**

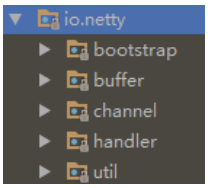
Netty4.x中文教程系列(六) 从头开始Bootstrap

其实自从中文教程系列（五）一直不知道自己到底想些什么。加上忙着工作上出现了一些问题。本来想就这么放弃维护了。没想到有朋友和我说百度搜索推荐了我的文章。瞬间有点小激动啊。决定自己要把这个教程系列完善下去。这里诚挚的想支持我的盆友们道歉。真的是让你们失望了。我居然有想放弃的这种丧心病狂的念头。以后绝对不会了。

其实伴随着对Netty的逐步深入学习。感觉自己对netty的了解仍然有所欠缺。加上笔者语文课是美术老师教的。所以。。说多了都是泪啊。~~o(>_<)o ~~

下面开始正文：

纵览Netty框架的包结构，不难看出。其实Netty是有五大模块组成。

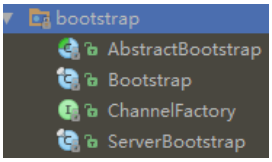


1. Bootstrap负责启动引导
2. Buffer是Netty自己封装的缓存器
3. Channel负责管理和建立连接
4. Handler是责任链路模式中的处理者
5. Util是Netty提供和使用到的一些工具

如何启动**Netty**服务器

Netty的启动服务器相关的类全部都在bootstrap包里面。所以本章我们从头开始，从bootstrap包里面的内容开始。从创建一个Netty服务器开始为大家逐步讲解Netty的应用。

相比于第五章的ChannelHandler里面的编解码器bootstrap里面可以说是内容少的可怜。来看一下他的包内容：



简简单单的三个类，一个接口。

Bootstrap是客户端的启动程序类。

ServerBootstrap是服务端的启动程序类

Bootstrap和ServerBootstrap继承AbstractBootstrap。

ChannelFactory则是AbstractBootstrap中用于创建Channel的接口

以下代码以服务端的启动程序启动为例：

步骤一：实例化**ServerBootstrap**

首先我们需要实例化一个ServerBootstrap服务端启动引导程序。如下图：

```
ServerBootstrap bootstrap = new ServerBootstrap();
```

步骤二：设置它的线程组

创建两个NioEventLoopGroup，一个是父线程（Boss线程），一个是子线程(work线程)。

```
NioEventLoopGroup parentBosser = new NioEventLoopGroup();
NioEventLoopGroup childWorker = new NioEventLoopGroup();
```

设置bootstrap的线程组

```
bootstrap.group(parentBosser, childWorker);
```

设置线程组主要的目的是为了处理Channel中的事件和IO操作。

下图为ServerBootstrap的group方法的源码：

```
public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup) {
    super.group(parentGroup);
    if (childGroup == null) {
        throw new NullPointerException("childGroup");
    }
    if (this.childGroup != null) {
        throw new IllegalStateException("childGroup set already");
    }
    this.childGroup = childGroup;
    return this;
}
```

父线程组被传递到父类中。详细的解释在最后面。涉及的东西太多。在后面在进行解释。

步骤三：设置**Channel**类型

设置Channel类型：

```
bootstrap.channel(NioServerSocketChannel.class);
```

下图ServerBootstrap中channel()方法的源码：

```
public B channel(Class<? extends C> channelClass) {
    if (channelClass == null) {
        throw new NullPointerException("channelClass");
    }
    return channelFactory(new BootstrapChannelFactory<C>(channelClass));
}
```

我们可以看到创建并设置了一个Channel工厂。

下图是BootstrapChannelFactory的源码。它是一个终态的静态的类。实现ChannelFactory。作用是根据初始设置的Channel类型，创建并返回一个新的Channel。

```
private static final class BootstrapChannelFactory<T extends Channel> implements ChannelFactory<T> {
    private final Class<? extends T> clazz;

    BootstrapChannelFactory(Class<? extends T> clazz) {
        this.clazz = clazz;
    }

    @Override
    public T newChannel() {
        try {
            return clazz.newInstance();
        } catch (Throwable t) {
            throw new ChannelException("Unable to create Channel from class " + clazz, t);
        }
    }

    @Override
    public String toString() { return StringUtil.simpleClassName(clazz) + ".class"; }
}
```

步骤四：设置责任链路

责任链模式是Netty的核心部分。每个处理者只负责自己有关的东西。然后将处理结果根据责任链传递下去。

```
bootstrap.childHandler(new ChannelInitializer<NioSocketChannel>() {
```

我们要在初始的设置一个责任链路。当一个Channel被创建之后初始化的时候将被设置。下图是ServerBootstrap在init（）方法的源码：

```
final EventLoopGroup currentChildGroup = childGroup;
final ChannelHandler currentChildHandler = childHandler;
final Entry<ChannelOption<?>, Object>[] currentChildOptions;
final Entry<AttributeKey<?>, Object>[] currentChildAttrs;
synchronized (childOptions) {
    currentChildOptions = childOptions.entrySet().toArray(newOptionArray(childOptions.size()));
}
synchronized (childAttrs) {
    currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(childAttrs.size()));
}

p.addLast(new ChannelInitializer<Channel>() {
    @Override
    public void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new ServerBootstrapAcceptor(
                currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));
    }
});
```

创建一个Channel，在初始化的设置管道里面的处理者。

步骤五：绑定并监听端口

绑定并设置监听端口。

```
try {
    // 绑定并监听端口
    ChannelFuture future = bootstrap.bind(9002).sync();
    // 等待关闭事件
    future.channel().closeFuture().sync();
} catch (InterruptedException e) {
    throw new Exception("");
} finally {
    // 释放资源
    parentBosser.shutdownGracefully();
    childWorker.shutdownGracefully();
}
```

经过以上的5个步骤，我们的服务器就足以启动了。很多的设置都是Netty默认的。我们想设置自己的参数怎么办呢？Netty提供了这个方法。

步骤六：其他设置

1. 设置Channel选项配置：

在Netty 以前的版本中都是以字符串来配置的。4.x版本发布之后统一修改为使用ChannelOption类来实现配置。

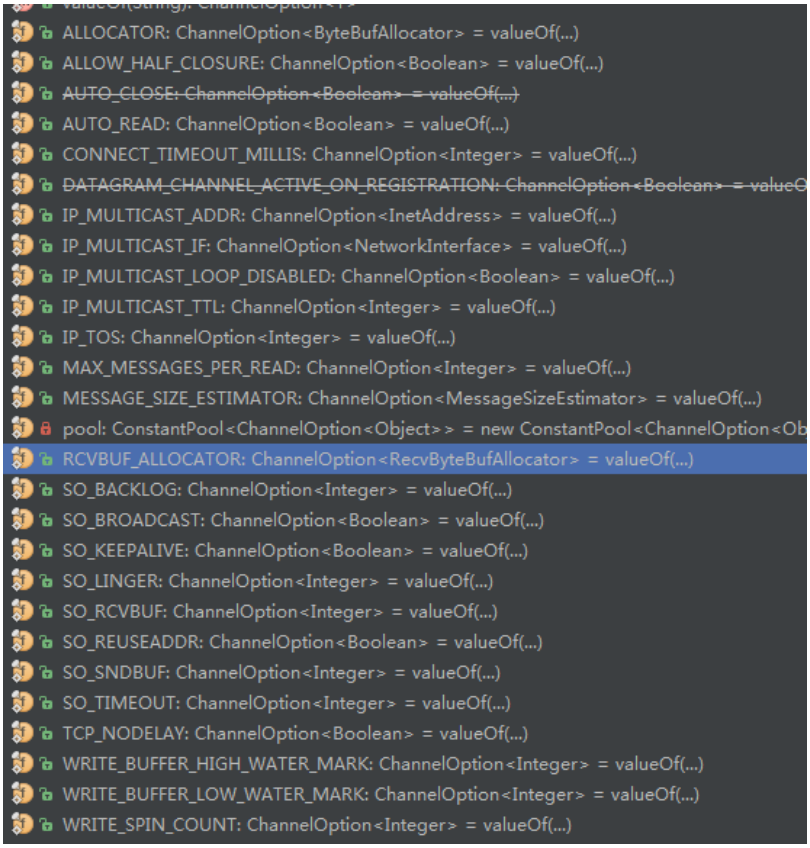
例如：

```
bootstrap.option(ChannelOption.SO_KEEPALIVE, true);
```

Socket连接是否保存连接:

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

还有很多其他的参数。如下图所示：



这里不详细讲了。参考：io.netty.channel.ChannelOption

2. 设置子Channel的属性：

```
bootstrap.childAttr()
```

设置子Channel的属性。当值为null是，属性将被删除。

解释**EventLoopGroup**

这里解释一下我们上面创建的两个完全一样的线程组的作用。

Netty的架构使用了非常复杂的主从式Reactor线程模型。简单的说就是。父线程组（代码中的parentBosser）担任（acceptor）的角色。负责接收客户端的连接请求，处理完成请求，创建一个Channel并注册到子线程组（代码中的childWorker）中的某个线程上面，然后这个线程将负责Channel的读写，编解码等操作。

源代码查看：

在步骤四中我们设置了责任链路。这里是Channel初始化和注册。在这里的init就是Channel的初始化。初始化完成之后。Group()则是获取在步骤一种的设置父线程组，并将这个新的Channel注册进来。

下图是AbstractBootstrap的initAndRegister方法

```
final ChannelFuture initAndRegister() {
    final Channel channel = channelFactory().newChannel();
    try {
        init(channel);
    } catch (Throwable t) {
        channel.unsafe().closeForcibly();
        // as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor
        return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);
    }

    ChannelFuture regFuture = group().register(channel);
    if (regFuture.cause() != null) {
        if (channel.isRegistered()) {
            channel.close();
        } else {
            channel.unsafe().closeForcibly();
        }
    }

    // If we are here and the promise is not failed, it's one of the following cases:
    // 1) If we attempted registration from the event loop, the registration has been completed at this point.
    //    i.e. It's safe to attempt bind() or connect() now because the channel has been registered.
    // 2) If we attempted registration from the other thread, the registration request has been successfully
    //    added to the event loop's task queue for later execution.
    //    i.e. It's safe to attempt bind() or connect() now:
    //         because bind() or connect() will be executed *after* the scheduled registration task is executed
    //         because register(), bind(), and connect() are all bound to the same thread.

    return regFuture;
}
```

方法Init()实现在ServerBootstrap中。代码如下：

```
void init(Channel channel) throws Exception {
    final Map<ChannelOption<?>, Object> options = options();
    synchronized (options) {
        channel.config().setOptions(options);
    }

    final Map<AttributeKey<?>, Object> attrs = attrs();
    synchronized (attrs) {
        for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {
            @SuppressWarnings("unchecked")
            AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();
            channel.attr(key).set(e.getValue());
        }
    }

    ChannelPipeline p = channel.pipeline();
    if (handler() != null) {
        p.addLast(handler());
    }

    final EventLoopGroup currentChildGroup = childGroup;
    final ChannelHandler currentChildHandler = childHandler;
    final Entry<ChannelOption<?>, Object>[] currentChildOptions;
    final Entry<AttributeKey<?>, Object>[] currentChildAttrs;
    synchronized (childOptions) {
        currentChildOptions = childOptions.entrySet().toArray(newOptionArray(childOptions.size()));
    }
    synchronized (childAttrs) {
        currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(childAttrs.size()));
    }

    p.addLast(new ChannelInitializer<Channel>() {
        @Override
        public void initChannel(Channel ch) throws Exception {
            ch.pipeline().addLast(new ServerBootstrapAcceptor(
                    currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));
        }
    });
}
```

看到下面的代码是不是有种和熟悉的感觉？没错。就是在步骤四中设置责任链路的那段代码。这里将注册新创建的Channel到子线程组

Ps: 完。。。O(∩_∩)O哈哈~。。。写的好辛苦的说。。。附上我的测试示例代码。好累。。写这么多字。希望能帮助到大家

```
1 import io.netty.bootstrap.ServerBootstrap;
2 import io.netty.channel.*;
3 import io.netty.channel.nio.NioEventLoopGroup;
4 import io.netty.channel.socket.nio.NioServerSocketChannel;
5 import io.netty.channel.socket.nio.NioSocketChannel;
6 import io.netty.handler.codec.LengthFieldBasedFrameDecoder;
7 import io.netty.handler.codec.LengthFieldPrepender;
```

```
 8
 9 /**
10  * 测试。。O(∩_∩)O哈哈~
11  * Created by TinyZ on 2014/8/12.
12  */
13 public class MainTest {
14
15     public static void main(String[] args) throws Exception {
16
17         NioEventLoopGroup parentBosser =new NioEventLoopGroup();
18         NioEventLoopGroup childWorker =new NioEventLoopGroup();
19
20         ServerBootstrap bootstrap =new ServerBootstrap();
21         bootstrap.group(parentBosser, childWorker);
22         bootstrap.channel(NioServerSocketChannel.class);
23         bootstrap.childHandler(new ChannelInitializer<NioSocketChannel>() {
24             @Override
25             protected void initChannel(NioSocketChannel ch) throws Exception {
26                 ChannelPipeline cp =ch.pipeline();
27                 // 基于长度的解码器
28                 cp.addLast("framer",new
LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 0, 2, 0, 2));
29                 cp.addLast("prepender",new LengthFieldPrepender(4));
30                 //
31                 cp.addLast("handler",new SimpleChannelInboundHandler<Object>() {
32
33                     @Override
34                     protected void channelRead0(ChannelHandlerContext ctx, Object
msg) throws Exception {
35
36                         System.out.println();
37                         ctx.channel().writeAndFlush(msg);
38
39                     }
40                 });
41             }
42         });
43         bootstrap.option(ChannelOption.SO_KEEPALIVE,true);
44         //bootstrap.childAttr()
45         try {
46             // 绑定并监听端口
47             ChannelFuture future = bootstrap.bind(9002).sync();
48             // 等待关闭事件
49             future.channel().closeFuture().sync();
50         } finally {
51             // 释放资源
52             parentBosser.shutdownGracefully();
53             childWorker.shutdownGracefully();
54         }
55     }
56 }
```


分类: Netty

标签: Netty, java

« 上一篇：Unity3D中简单的C#异步Socket实现
» 下一篇：简单的异步Socket实现——SimpleSocket_V1.1

posted @ 2014-08-12 18:24 Tiny&zzh 阅读(5016) 评论(1) 编辑 收藏

评论列表

#1楼 2016-02-19 10:22 onlyxx

感谢楼主分享，解决了我很久的一些疑问

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。