客户端启动流程

上一小节，我们已经学习了 Netty 服务端启动的流程，这一小节，我们来学习一下 Netty 客户端的启动流程。

客户端启动 Demo

对于客户端的启动来说，和服务端的启动类似，依然需要线程模型、IO 模型，以及 IO 业务处理逻辑三大参数，下面，我们来看一下客户端启动的标准流程

NettyClient.java

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

        Bootstrap bootstrap = new Bootstrap();
        bootstrap
                // 1.指定线程模型
                .group(workerGroup)
                // 2.指定 IO 类型为 NIO
                .channel(NioSocketChannel.class)
                // 3.IO 处理逻辑
                .handler(new
ChannelInitializer<SocketChannel>() {
                    @Override
                    public void
initChannel(SocketChannel ch) {
                    }
                });
        // 4.建立连接
        bootstrap.connect("juejin.im",
80).addListener(future -> {
            if (future.isSuccess()) {
                System.out.println("连接成功!");
            } else {
                System.err.println("连接失败!");
            }

        });
    }
}
```

从上面代码可以看到，客户端启动的引导类是 Bootstrap，负责启动客户端以及连接服务端，而上一小节我们在描述服务端的启动的时候，这个辅导类是 ServerBootstrap，引导类创建完成之后，下面我们描述一下客户端启动的流程

1. 首先，与服务端的启动一样，我们需要给它指定线程模型，驱动着连接的数据读写，这个线程的概念可以和第一小节Netty是什么(https://juejin.im/book/5b4bc28bf265da0f60130116/se中的 IOClient.java 创建的线程联系起来
2. 然后，我们指定 IO 模型为 NioSocketChannel，表示 IO 模型为 NIO，当然，你可以可以设置 IO 模型为 OioSocketChannel，但是通常不会这么做，因为 Netty 的优势在于 NIO
3. 接着，给引导类指定一个 handler，这里主要就是定义连接的业务处理逻辑，不理解没关系，在后面我们会详细分析
4. 配置完线程模型、IO 模型、业务处理逻辑之后，调用 connect 方法进行连接，可以看到 connect 方法有两个参数，第一个参数可以填写 IP 或者域名，第二个参数填写的是端口号，由于 connect 方法返回的是一个 Future，也就是说这个方是异步的，我们通过 addListener 方法可以监听到连接是否成功，进而打印出连接信息

到了这里，一个客户端的启动的 Demo 就完成了，其实只要和 客户端 Socket 编程模型对应起来，这里的三个概念就会显得非常简单，遗忘掉的同学可以回顾一下 Netty是什么(https://juejin.im/book/5b4bc28bf265da0f60130116/section中的 IOClient.java 再回来看这里的启动流程哦

失败重连

在网络情况差的情况下，客户端第一次连接可能会连接失败，这个时候我们可能会尝试重新连接，重新连接的逻辑写在连接失败的逻辑块里

```
bootstrap.connect("juejin.im",
80).addListener(future -> {
    if (future.isSuccess()) {
        System.out.println("连接成功!");
    } else {
        System.err.println("连接失败!");
        // 重新连接
    }
});
```

重新连接的时候，依然是调用一样的逻辑，因此，我们把建立连接的逻辑先抽取出来，然后在重连失败的时候，递归调用自身

```
private static void connect(Bootstrap bootstrap,
String host, int port) {
    bootstrap.connect(host,
port).addListener(future -> {
        if (future.isSuccess()) {
            System.out.println("连接成功!");
        } else {
            System.err.println("连接失败，开始重
连");
            connect(bootstrap, host, port);
        }
    });
}
```

上面这一段便是带有自动重连功能的逻辑，可以看到在连接建立失败的时候，会调用自身进行重连。

但是，通常情况下，连接建立失败不会立即重新连接，而是会通过一个指数退避的方式，比如每隔 1 秒、2 秒、4 秒、8 秒，以 2 的幂次来建立连接，然后到达一定次数之后就放弃连接，接下来我们就来实现一下这段逻辑，我们默认重试 5 次

```
connect(bootstrap, "juejin.im", 80, MAX_RETRY);

private static void connect(Bootstrap bootstrap,
String host, int port, int retry) {
    bootstrap.connect(host,
port).addListener(future -> {
        if (future.isSuccess()) {
            System.out.println("连接成功!");
        } else if (retry == 0) {
            System.err.println("重试次数已用完，放弃
连接！");
        } else {
            // 第几次重连
            int order = (MAX_RETRY - retry) + 1;
            // 本次重连的间隔
            int delay = 1 << order;
            System.err.println(new Date() + ": 连
接失败，第" + order + "次重连……");

bootstrap.config().group().schedule(() ->
connect(bootstrap, host, port, retry - 1), delay,
TimeUnit
                    .SECONDS);
        }
    });
}
```

从上面的代码可以看到，通过判断连接是否成功以及剩余重试次数，分别执行不同的逻辑

1. 如果连接成功则打印连接成功的消息
2. 如果连接失败但是重试次数已经用完，放弃连接
3. 如果连接失败但是重试次数仍然没有用完，则计算下一次重连间隔 delay，然后定期重连

在上面的代码中，我们看到，我们定时任务是调用 bootstrap.config().group().schedule(), 其中 bootstrap.config() 这个方法返回的是 BootstrapConfig，他是对 Bootstrap 配置参数的抽象，然后 bootstrap.config().group() 返回的就是我们在一开始的时候配置的线程模型 workerGroup，调 workerGroup 的 schedule 方法即可实现定时任务逻辑。

在 schedule 方法块里面，前面四个参数我们原封不动地传递，最后一个重试次数参数减掉一，就是下一次建立连接时候的上下文信息。读者可以自行修改代码，更改到一个连接不上的服务端 Host 或者 Port，查看控制台日志就可以看到5次重连日志。

以上就是实现指数退避的客户端重连逻辑，接下来，我们来一起学习一下，客户端启动，我们的引导类Bootstrap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?

客户端启动其他方法

attr() 方法

```
bootstrap.attr(AttributeKey.newInstance("clientName"), "nettyClient")
```

attr() 方法可以给客户端 Channel，也就是NioSocketChannel绑定自定义属性，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客户端 Channel 的一个clientName属性，属性值为nettyClient，其实说白了就是给NioSocketChannel维护一个 map 而已，后续在这个NioSocketChannel 通过参数传来传去的时候，就可以通过他来取出设置的属性，非常方便。

option() 方法

```
Bootstrap

.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 
5000)
        .option(ChannelOption.SO_KEEPALIVE, true)
        .option(ChannelOption.TCP_NODELAY, true)
```

option() 方法可以给连接设置一些 TCP 底层相关的属性，比如上面，我们设置了三种 TCP 属性，其中

- ChannelOption.CONNECT_TIMEOUT_MILLIS 表示连接的超时时间，超过这个时间还是建立不上的话则代表连接失败
- ChannelOption.SO_KEEPALIVE 表示是否开启 TCP 底层心跳机制，true 为开启
- ChannelOption.TCP_NODELAY 表示是否开始 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，就设置为 true 关闭，如果需要减少发送次数减少网络交互，就设置为 false 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。

总结

- 本文中，我们首先学习了 Netty 客户端启动的流程，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，连接上特定主机和端口，客户端就启动起来了。
- 然后，我们学习到 connect 方法是异步的，我们可以通过这个异步回调机制来实现指数退避重连逻辑。
- 最后呢，我们讨论了 Netty 客户端启动额外的参数，主要包括给客户端 Channel 绑定自定义属性值，设置底层 TCP 参数。

> 本小节涉及到的源码已放置 github仓库 (https://github.com/lightningMan/flash-netty/tree/%E5%AE%A2%E6%88%B7%E7%AB%AF%E5%90%A clone 到本地之后切换到本小节对应分支即可

思考题

与服务端启动相比，客户端启动的引导类少了哪些方法，为什么不需要这些方法？欢迎留言讨论。