Netty学习资料

```
1. 使用场景
2. “HelloWorld” 程序分析
3. 核心API介绍
4. 项目需求介绍&功能分析
5. 项目总体架构设计
6. 预备知识
```

1️⃣ 使用场景

1. 互联网领域：构建高性能RPC框架基础通信组件，阿里分布式服务框架 Dubbo 的 RPC 框架使用 Dubbo 协议进行节点间通信，Dubbo 协议默认使用 Netty 作为基础通信组件，用于实现各进程节点之间的内部通信
2. 大数据领域：经典的 Hadoop 的高性能通信和序列化组件 Avro 的 RPC 框架，默认采用 Netty 进行跨节点通信，它的 Netty Service 基于 Netty 框架二次封装实现
3. 游戏行业：无论是手游服务端、还是大型的网络游戏，Java 语言得到了越来越广泛的应用。Netty 作为高性能的基础通信组件，它本身提供了 TCP/UDP 和 HTTP 协议栈，非常方便定制和开发私有协议栈，账号登陆服务器、地图服务器之间可以方便的通过 Netty 进行高性能的通信
4. 企业软件：企业和 IT 集成需要 ESB，Netty 对多协议支持、私有协议定制的简洁性和高性能是 ESB RPC 框架的首选通信组件。事实上，很多企业总线厂商会选择 Netty 作为基础通信组件，用于企业的 IT 集成。
5. 通信行业：Netty 的异步高性能、高可靠性和高成熟度的优点，使它在通信行业得到了大量的应

2️⃣ “HelloWorld” 程序分析

需求分析：客户端发送

项目结构：

├—— client ├—— Client.java -- 客户端启动类 ├—— ClientHandler.java -- 客户端逻辑处理类 ├—— ClientHandler.java -- 客户端初始化类 ├—— server ├—— Server.java -- 服务端启动类 ├—— ServerHandler.java -- 服务端逻辑处理类 ├—— ServerInitializer.java -- 服务端初始化类

服务端

首先是编写服务端的启动类。

```
public final class Server {
public  static void main(String[] args) throws Exception {
//Configure the server
//创建两个EventLoopGroup对象
//创建boss线程组 用于服务端接受客户端的连接
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
// 创建 worker 线程组 用于进行 SocketChannel 的数据读写
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
// 创建 ServerBootstrap 对象
ServerBootstrap b = new ServerBootstrap();
//设置使用的EventLoopGroup
b.group(bossGroup,workerGroup)
//设置要被实例化的为 NioServerSocketChannel 类
.channel(NioServerSocketChannel.class)
// 设置 NioServerSocketChannel 的处理器
.handler(new LoggingHandler(LogLevel.INFO))
// 设置连入服务端的 Client 的 SocketChannel 的处理器
.childHandler(new ServerInitializer());
// 绑定端口，并同步等待成功，即启动服务端
ChannelFuture f = b.bind(8888);
// 监听服务端关闭，并阻塞等待
f.channel().closeFuture().sync();
} finally {
// 优雅关闭两个 EventLoopGroup 对象
bossGroup.shutdownGracefully();
workerGroup.shutdownGracefully();
}
}
}
```

- 第6到8行： 创建两个EventLoopGroup对象。
  - boss 线程组: 用于服务端接受客户端的 **连接** 。
  - worker 线程组： 用于进行客户端的SocketChannel的 **数据读写** 。
- 第11行: 创建 ServerBootstrap 对象，用于设置服务端的启动配置。
  - 第13行: 调用 `#group(EventLoopGroup parentGroup, EventLoopGroup childGroup)` 方法，设置使用的 EventLoopGroup 。
  - 第15行: 调用 `#channel(Class<? extends C> channelClass)` 方法，设置要被实例化的 Channel 为 NioServerSocketChannel 类。在下文中，我们会看到该 Channel 内嵌了 `java.nio.channels.ServerSocketChannel` 对象。
  - 第17行: 调用 `#handler(ChannelHandler handler)` 方法，设置 NioServerSocketChannel 的处理器。在本示例中，使用了 `io.netty.handler.logging.LoggingHandler` 类，用于打印服务端的每个事件。
  - 第19行: 调用 `#childHandler(ChannelHandler handler)` 方法，设置连入服务端的 Client 的 SocketChannel 的处理器。在本实例中，使用 ServerInitializer() 来初始化连入服务端的 Client 的 SocketChannel 的处理器。

- 第21行: **先** 调用 `#bind(int port)` 方法，绑定端口， **后** 调用 `ChannelFuture#sync()` 方法，阻塞等待成功。这个过程，就是“ **启动服务端** ”。
- 第23行: **先** 调用 `#closeFuture()` 方法， **监听** 服务器关闭， **后** 调用 `ChannelFuture#sync()` 方法，阻塞等待成功。: 注意，此处不是关闭服务器，而是“ **监听** ”关闭。
- 第26到27行: 执行到此处，说明服务端已经关闭，所以调用 `EventLoopGroup#shutdownGracefully()` 方法，分别关闭两个 EventLoopGroup 对象。

服务端主类编写完毕之后，我们再来设置下相应的过滤条件。 这里需要继承Netty中 **ChannelInitializer** 类，然后重写 **initChannel** 该方法，进行添加相应的设置，传输协议设置，以及相应的业务实现类。

```
public class ServerInitializer extends ChannelInitializer<SocketChannel> {
private static final StringDecoder DECODER = new StringDecoder();
private static final StringEncoder ENCODER = new StringEncoder();
private static final ServerHandler SERVER_HANDLER = new ServerHandler();
@Override
public void initChannel(SocketChannel ch) throws Exception {
ChannelPipeline pipeline = ch.pipeline();
// 添加帧限定符来防止粘包现象
pipeline.addLast(new DelimiterBasedFrameDecoder(8192,
Delimiters.lineDelimiter()));
// 解码和编码，应和客户端一致
pipeline.addLast(DECODER);
pipeline.addLast(ENCODER);
// 业务逻辑实现类
pipeline.addLast(SERVER_HANDLER);
}
}
```

服务相关的设置的代码写完之后，我们再来编写主要的业务代码。 使用Netty编写 [业务层]的代码，我们需要继承 **ChannelInboundHandlerAdapter** 或 **SimpleChannelInboundHandler** 类，在这里顺便说下它们两的区别吧。 继承 **SimpleChannelInboundHandler** 类之后，会在接收到数据后会自动 **release** 掉数据占用的 **Bytebuffer** 资源。并且继承该类需要指定数据格式。 而继承**ChannelInboundHandlerAdapter** 则不会自动释放，需要手动调用 ReferenceCountUtil.release() 等方法进行释放。继承该类不需要指定数据格式。 所以在这里，个人推荐服务端继承 ChannelInboundHandlerAdapter ，手动进行释放，防止数据未处理完就自动释放了。而且服务端可能有多个客户端进行连接，并且每一个客户端请求的数据格式都不一致，这时便可以进行相应的处理。 客户端根据情况可以继承 **SimpleChannelInboundHandler** 类。好处是直接指定好传输的数据格式，就不需要再进行格式的转换了。

```
@Sharable
```

```
public class ServerHandler extends SimpleChannelInboundHandler<String>
{
/**

- 建立连接时，发送一条庆祝消息
  */
  @Override
  public void channelActive(ChannelHandlerContext ctx) throws
Exception {
        // 为新连接发送庆祝
        ctx.write("Welcome to " +
InetAddress.getLocalHost().getHostName() + "!/r/n");
        ctx.write("It is " + new Date() + " now./r/n");
         ctx.flush();
  }
  //业务逻辑处理
  @Override
  public void channelRead0(ChannelHandlerContext ctx, String request)
throws Exception  {
    // Generate and write a response.
     String response;
     boolean close = false;
    if (request.isEmpty()) {
        response = "Please type something./r/n";
    } else if ("bye".equals(request.toLowerCase())) {
        response = "Have a good day!/r/n";
        close = true;
    } else {
        response = "Did you say '" + request + "'?/r/n";
    }
        ChannelFuture future = ctx.write(response);
    if (close) {
        future.addListener(ChannelFutureListener.CLOSE);
    }
  }
  @Override
  public void channelReadComplete(ChannelHandlerContext ctx) {
       ctx.flush();
  }
  //异常处理
  @Override
  public void exceptionCaught(ChannelHandlerContext ctx, Throwable
cause) {
         cause.printStackTrace();
         ctx.close();
  }
}
```

客户端

```
public static void main(String[] args) throws Exception {
  EventLoopGroup group = new NioEventLoopGroup();
    try {
        Bootstrap b = new Bootstrap();
        b.group(group)
         .channel(NioSocketChannel.class)
         .handler(new ClientInitializer());
        Channel ch = b.connect("127.0.0.1",8888).sync().channel();
        ChannelFuture lastWriteFuture = null;
        BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
        for (;;) {
        String line = in.readLine();
            if (line == null) {
                break;
            }
        // Sends the received line to the server.
        lastWriteFuture = ch.writeAndFlush(line + "/r/n");
        // If user typed the 'bye' command, wait until the server closes
        // the connection.
            if ("bye".equals(line.toLowerCase())) {
                ch.closeFuture().sync();
                break;
            }
        }
        // Wait until all messages are flushed before closing the channel.
        if (lastWriteFuture != null) {
            lastWriteFuture.sync();
        }
    } finally {
        group.shutdownGracefully();
    }
}
```

客户端过滤其这块基本和服务端一致。不过需要注意的是，传输协议、编码和解码应该一致

```
public class ClientInitializer extends ChannelInitializer<SocketChannel> {
    private static final StringDecoder DECODER = new StringDecoder();
    private static final StringEncoder ENCODER = new StringEncoder();
    private static final ClientHandler CLIENT_HANDLER = new
ClientHandler();

    @Override
        public void initChannel(SocketChannel ch) {
            ChannelPipeline pipeline = ch.pipeline();
```

```
            pipeline.addLast(new DelimiterBasedFrameDecoder(8192,
Delimiters.lineDelimiter()));
            pipeline.addLast(DECODER);
            pipeline.addLast(ENCODER);
            pipeline.addLast(CLIENT_HANDLER);
        }
}
```

客户端的业务代码逻辑。

主要时打印读取到的信息。

这里有个注解， 该注解 **Sharable** 主要是为了多个handler可以被多个channel安全地共享，也就是保证线程安全。代码如下：

```
@Sharable
public class ClientHandler extends SimpleChannelInboundHandler<String> {
    //打印读取到的数据
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String msg)
throws   Exception       {
        System.err.println(msg);
    }
    //异常数据捕获
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
{
      cause.printStackTrace();
      ctx.close();
    }
}
```

3️⃣.核心API介绍

可先行自己学习，后续会在写项目案例的时候介绍

https://netty.io/4.1/api/index.html

4️⃣.项目需求介绍&功能分析

配送抢单功能需求分析

一、功能介绍：

配送服务是一种众包物流的模式，致力于解决O2O最后3公里配送痛点，通过召集社会兼职人员，通过抢单的形式接受饿了吗平台商家的订单配送任务，并按单赚取酬劳。在这里仅分析配送员端APP的抢单的功能（不考虑商家端及配送功能和钱包功能）。

二、需求列表：

作为众包物流平台，其配送员端的目标人群包括两类，一类是以兼职为目的的兼职配送员，一类是以全职为目的驻店配送员，下表通过分析这两类人群的特点及其需求和痛点，整理了抢单需求列表。

> **三、需求分类整合：**

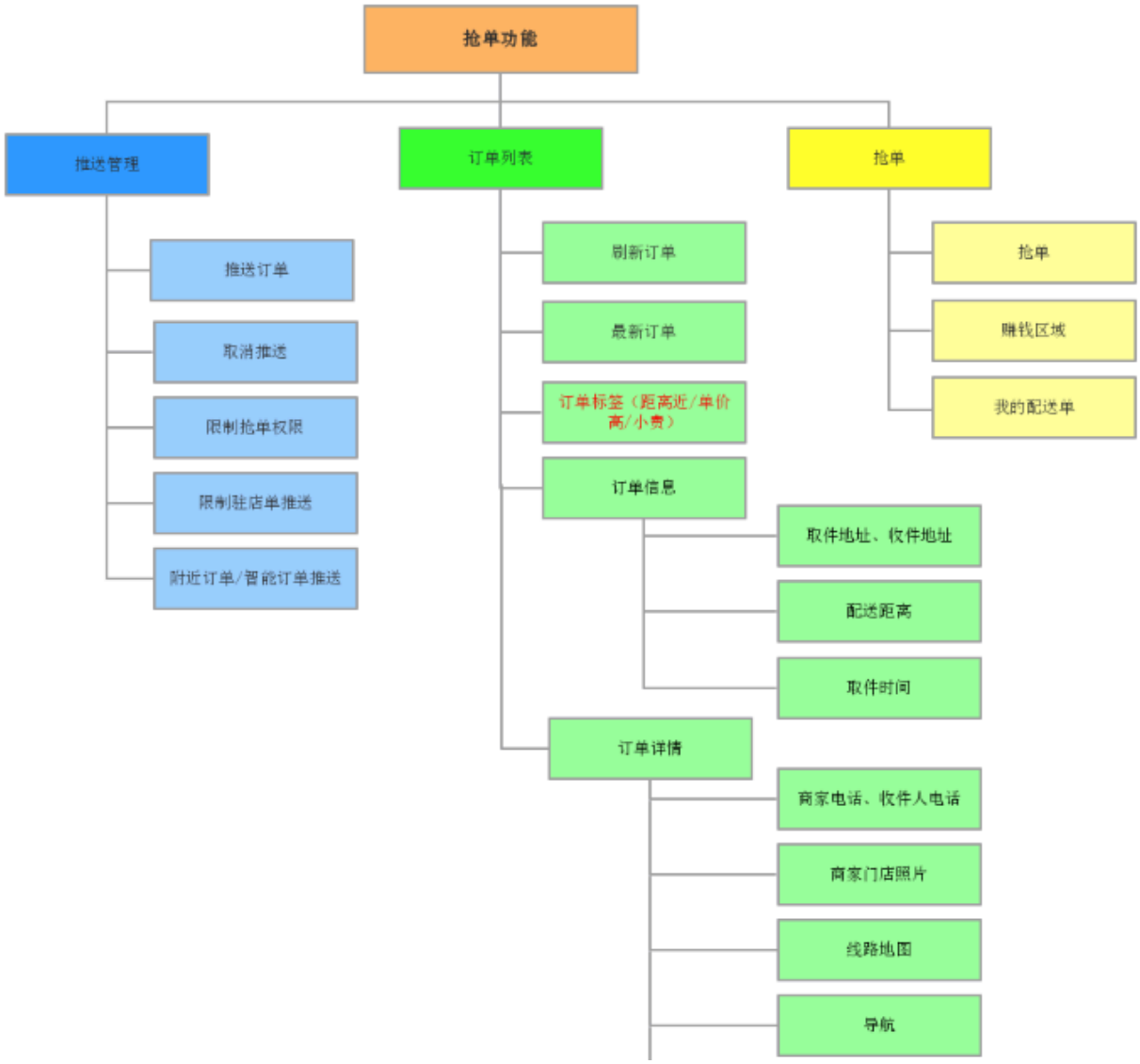
在需求列表中根据角色及其诉求穷举的需求是散乱的，在这里根据功能的属性将其进行归类整合，使其形成清晰的信息脉络。

考虑到一个订单承载的信息量过多，无法同时在APP订单列表中显示，分为“订单信息”和“订单详情”两类，将重要信息归纳在“订单信息”中在订单列表显示，将不重要的信息隐藏在“订单详情”中，点击进入才能查看。其中，取件地址、收货地址、配送费、配送距离、取件时间是配送员决定是否抢单的重要参考因素，因此需要将这些信息的层级提升。

角色	角色特点	角色需求及痛点	需求项
兼职配送员	1.有较多空闲时间，以兼职为目的 2.通过抢单接活，按单计算酬劳 3.要求对所配送区域道路情况比较熟悉	1.希望接到更多优质订单（例如距离近、单价高、有小费等），赚取更多酬劳	1. 订单推送**（推送3公里之内的订单）2** 抢单**3** 配送费** 4 **订单标签**（距离近/单价高/小费）
		2.希望能抢到多个路线相近的订单，可以同时进行配送，增加每次配送的效益	**附近订单****/智能订单推送**
		3.希望推送的订单都是有效的订单，不希望已过时的已被抢的订单	1.**刷新订单** 2.**最新订单** 3.**取消推送**（已被抢的、无效的订单取消推送）
		4.希望能知道哪个区域订单多，直接到那个区域配送，容易抢到更多订单	**赚钱区域**
		5.希望能有准确的路线指引或者导航帮助，避免绕路或者走错路耽误配送	1 **配送距离** 2 **线路地图** 3 **导航**
		6.希望到达取件地点就能快速找到商家店面，不用浪费时间找	**商家门店照片**
		7.希望到达店面时餐已做好可直接取餐，不用浪费时间等待	**取餐时间**（店家约定取餐时间，配送员该时间到达即可取餐）
		8.希望有更详细的取货商家信息和收货人信息，方便第一时间找到他们，联系到他们	1.**取件地址** 2.**收件地址** 3.**商家电话** 4.**收件人电话**
		9.能确认是否抢单成功，能查看自己的配送单	**我的配送单**

在需求列表中根据角色及其诉求穷举的需求是散乱的，在这里根据功能的属性将其进行归类整合，使其形成清晰的信息脉络。

考虑到一个订单承载的信息量过多，无法同时在APP订单列表中显示，分为“订单信息”和“订单详情”两类，将重要信息归纳在“订单信息”中在订单列表显示，将不重要的信息隐藏在“订单详情”中，点击进入才能查看。其中，取件地址、收货地址、配送费、配送距离、取件时间是配送员决定是否抢单的重要参考因素，因此需要将这些信息的层级提升。



四、需求优先级考虑

考虑需求优先级，将需求划分为三个层次：基本型需求、期望型需求和兴奋型需求。基本型需求是必须全力以赴去满足的；期望型需求是尽量去满足，提升用户满意度的；兴奋型需求是最后去考虑满足的。

基本型需求：推送订单、取消订单、刷新订单、抢单、订单信息、订单详情、我的配送单

期望型需求：限制抢单权限、限制驻店单推送、订单标签、最新订单、赚钱区域

兴奋型需求：附近订单/智能订单推送、智能线路地图

当然，目前只是考虑优先级的其中一个参考维度，实际过程中还要考虑技术实现难度、商业风险等。例如，在配送员的角度，一次配送越多单，能赚到更多的配送费，平台在考虑满足他们的期望的同时也要考虑如果配送员同时接过多订单，导致配送超时，降低服务质量的问题。

5️⃣ 项目总体架构设计

上课过程中补充

6️⃣ 预备知识

一、需要有redis使用以及相关API的开发经验

二、需要有高并发的相关经验

三、需要有zookeeper使用以及相关API的开发经验