

# 多线程抓包实验报告

---

## 1. 开发环境

## 2. 项目设计思路

## 3. 项目架构模块

### 3.1. 模块划分

### 3.2. 模块关系图

## 4. 项目模块说明

### 4.1. header

#### 4.1.1. 构成

#### 4.1.2. 类之间的关系

#### 4.1.3. 功能介绍

##### 4.1.3.1. BaseHeader

##### 4.1.3.2. IPHeader

##### 4.1.3.3. TCPHeader

### 4.2. networkInterface

#### 4.2.1. 构成

#### 4.2.2. 功能介绍

### 4.3. strategy

#### 4.3.1. 构成

#### 4.3.2. 类之间的关系

#### 4.3.3. 功能介绍

### 4.4. packet

#### 4.4.1. 构成

#### 4.4.2. 功能介绍

### 4.5. exceptions

#### 4.5.1. 构成

#### 4.5.2. 功能介绍

### 4.6. enums

#### 4.6.1. 构成

- 4.6.2. 功能介绍
- 4.7. factory
  - 4.7.1. 构成
  - 4.7.2. 功能介绍
- 4.8. frame
  - 4.8.1. 构成
  - 4.8.2. 类/模块之间的关系
  - 4.8.3. 功能介绍
    - 4.8.3.1. component
    - 4.8.3.2. panel
      - 4.8.3.2.1. ButtonPanel
      - 4.8.3.2.2. CheckBoxPanel
      - 4.8.3.2.3. InputPanel
      - 4.8.3.2.4. MainPanel
    - 4.8.3.3. MainFrame
    - 4.8.3.4. 设计的好处
- 4.9. thread
  - 4.9.1. 构成
  - 4.9.2. 功能介绍
- 5. 核心代码设计
  - 5.1. 监听函数
    - 5.1.1. 网络接口监听显示
    - 5.1.2. 抓包监听
  - 5.2. 抓包函数
- 6. 产品使用说明书
  - 6.1. 程序主界面
  - 6.2. 网络接口选择
  - 6.3. IP地址选择
  - 6.4. 协议解析选择
  - 6.5. 抓包操作
  - 6.6. 抓包效果展示

本报告将从以下方面对该系统展开描述，分别为：

- 1.介绍本次系统设计的基本开发环境，包括开发工具，编程语言的使用等等。
- 2.将介绍说明本项目的基本设计思路，以及预计要使用的设计模式思想。
- 3.说明该项目的基本模块划分以及各模块在项目中起到的作用，逻辑关系。
- 4.本次报告的核心部分，将对每一个模块的设计展开详细的介绍，包括类的设计，类与类之间的关系，所用到 的设计模式思想等等。
- 5.将会对该系统的核心代码部分展开详细的说明。
- 6.介绍如何正确使用该产品以及运行效果示意图。

## 1. 开发环境

开发工具：IntelliJ IDEA

编程语言：JAVA(JDK 17)

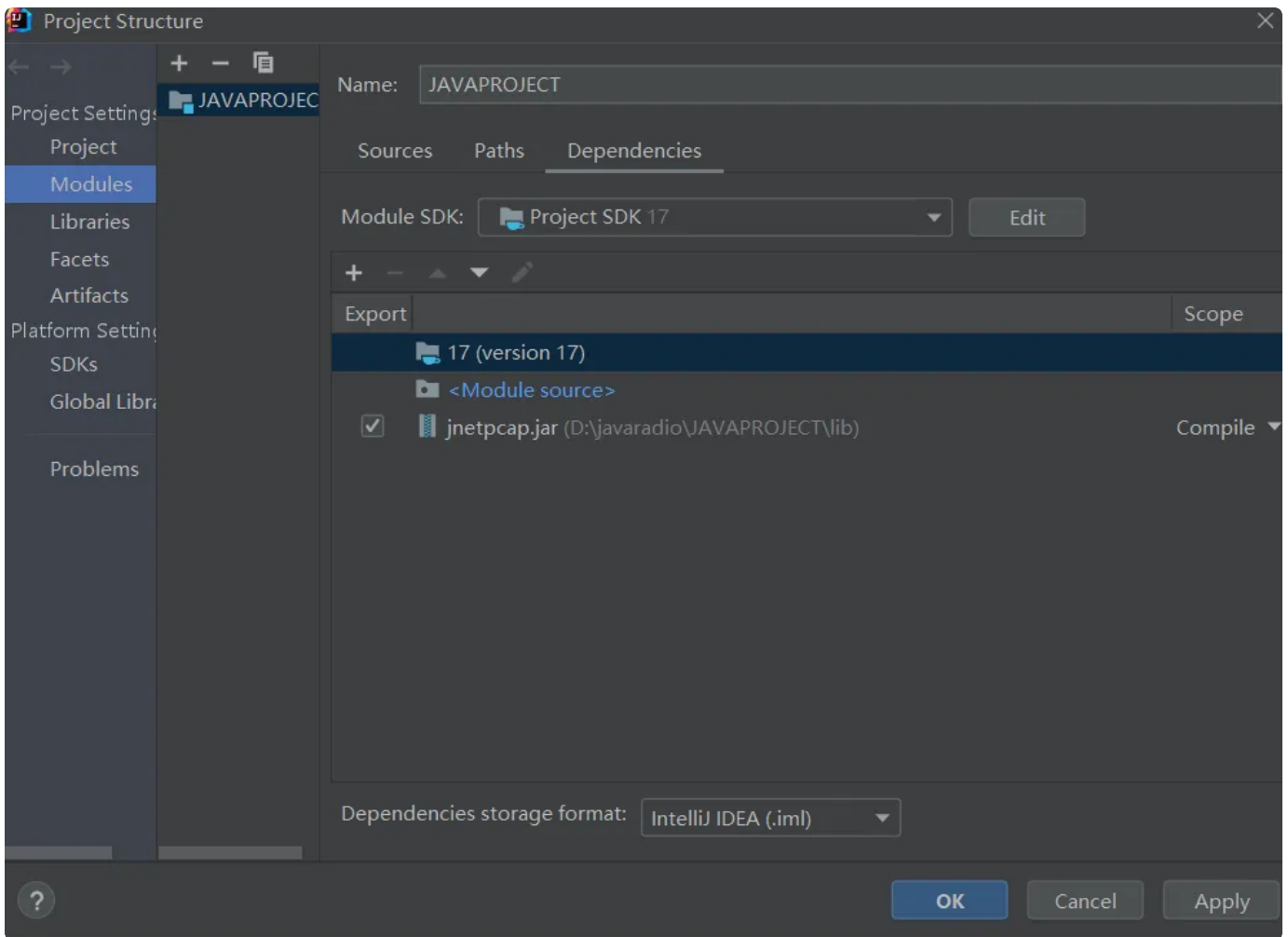
网络数据包捕获库和驱动程序：npcap

导入的第三方库：jnetpcap.jar

图形化界面：swing

注意：

要在Project Settings中对于lib目录中的jar包中进行依赖选择jnetpcap.jar包，并且将lib目录下的dll扩展文件放在JDK 17中的bin目录当中去：



## 2. 项目设计思路

首先想到的就是由于不知道用户要进行解析的是IP头还是TCP头，因此我们要设计2个类TCPHeader以及IPHeader，为了达到最大程度的解耦合，我们采用设计模式中的策略模式思想，让两个类都继承自BaseHeader类，方便后面利用多态的特性进行抓包。

抓包的过程中会遇到网络抓包的相关异常，因此我们要自定义一个异常类用于我们的系统。

考虑到分析的模式我们不是IP就是TCP，因此可以增加一个枚举类。

网络接口也可以单独作为一个模块使用，最起码的成员一定会有网络接口以及IP地址。

由于涉及到不同的抓包策略，因此我们跟IP头的做法类似，定义一个PacketStrategy类，再定义IP与TCP两种策略继承自该抽象类，实现不同的抓包策略。

由于该系统需要多线程的支持，因此我们需要自定义一个thread模块，里面的类可以创建一个线程池用于多线程并发操作。

packet包下的类设计首先考虑到我们要多用组装，少用继承的思想，因此我们可以将网络接口，抓包策略都作为该类的成员变量。

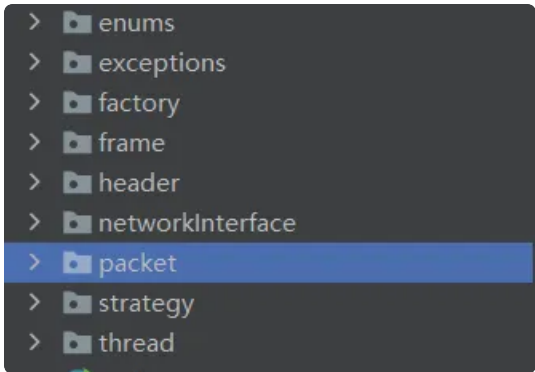
factory模块我们考虑通过设计模式中的工厂模式，客户端代码与具体的对象创建过程解耦合。客户端代码只需要知道如何使用工厂方法来获取所需的对象，而无需关心对象的创建细节。这种解耦合提高了系统的灵活性和可维护性，使得系统更容易进行修改和扩展。

最后我们需要一个frame模块用于展示所有的可视化布局组件元素。

### 3. 项目架构模块

#### 3.1. 模块划分

该系统主要分为了如下几个模块：



模块	作用
enums	枚举模块，主要是区分TCP与IP
exceptions	自定义的异常模块

factory	用于根据不同的策略创建不同类型包的工厂模块
frame	用于可视化显示的模块
header	用于解析接收包头部的模块
networkInterface	网络接口模块
packet	用来接解析抓取得到的包的模块
strategy	抓包策略模块
thread	多线程工具模块

程序的Main方法入口：

▼

Java

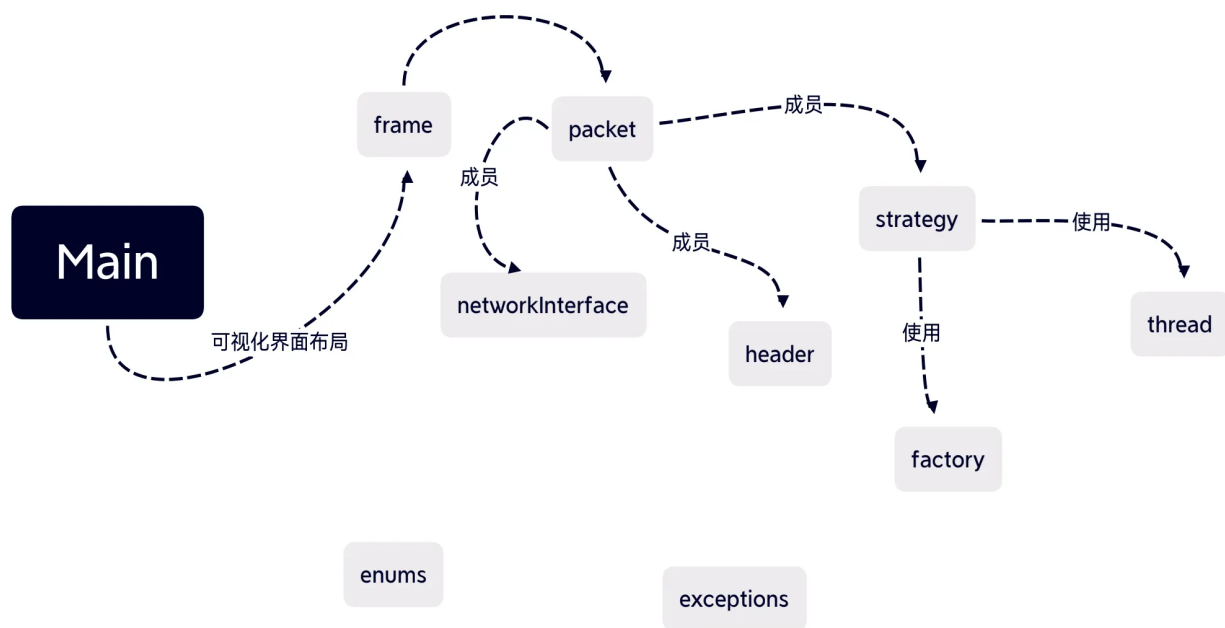
```

1  public class Main {
2      public static void main(String[] args) {
3          MainFrame mainFrame=new MainFrame();
4          mainFrame.setVisible(true);
5      }
6  }

```

只有最简单的一个创建对象的过程作为程序的入口，最大化面向对象编程的设计思想。

## 3.2. 模块关系图



frame作为可视化界面模块是该系统的基层建筑，其他模块或多或少都要依赖于该模块。

packet模块中的Packet类成员变量与networkInterface模块，factory模块，thread模块相关联在一起。

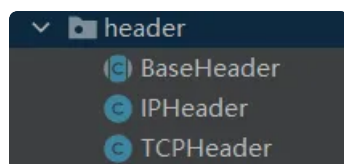
enums与exceptions则是作为工具类型模块。

## 4. 项目模块说明

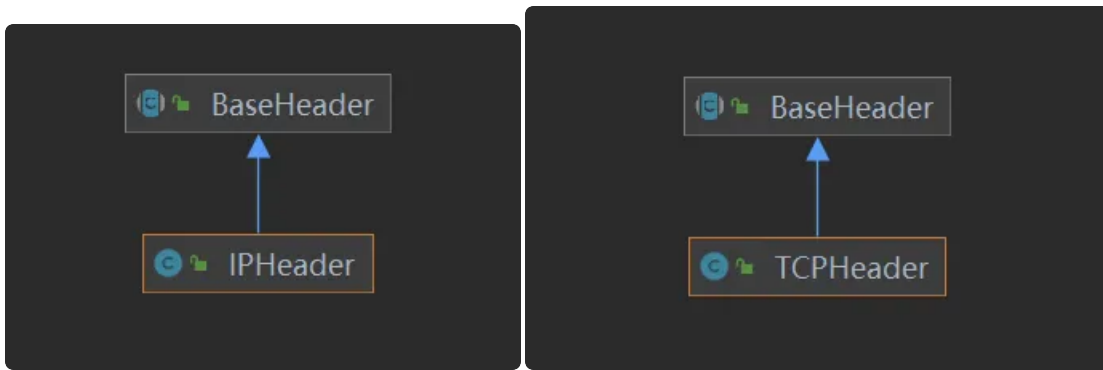
### 4.1. header

#### 4.1.1. 构成

由BaseHeader，IPHeader，TCPHeader构成：



#### 4.1.2. 类之间的关系



IPHeader与TCPHeader都继承自抽象类BaseHeader

### 4.1.3. 功能介绍

注：IPHeader与TCPHeader都重写了toString方法，为了更好地为用户展示抓包结果

#### 4.1.3.1. BaseHeader

*在该系统中，BaseHeader为一个没有任何方法的空抽象类。*

将IPHeader和TCPHeader继承自空抽象类BaseHeader是为了利用面向对象编程中的继承机制来实现代码的重用和组织结构的清晰性。

主要考虑到以下好处：

1.统一接口：通过将IPHeader和TCPHeader都继承自BaseHeader，可以确保它们都具有一致的接口。这样做有助于提高代码的可维护性和可扩展性，因为调用方可以统一使用BaseHeader定义的接口，而无需关心具体是哪个子类（例如后面策略模式的使用）。

2.提高可读性：通过这种继承结构，代码的组织结构更加清晰，可以更容易地理解系统的设计和功。BaseHeader作为一个抽象类，清晰地表明了它是一个基类，而IPHeader和TCPHeader作为具体的子类，表示它们是在基类的基础上进行了特定功能的扩展。

其实还有一个好处就是代码重用，如果IPHeader和TCPHeader具有一些共同的属性或方法，将它们放在BaseHeader中，可以避免在每个子类中重复编写相同的代码。这样可以减少代码冗余，并且在需要修改这些共同特性时，只需在BaseHeader中进行一次修改即可，而不必在多个地方修改。但是由于目前的设计中还没有IP头与TCP头共用的方法，因此只是设计为了空类。

#### 4.1.3.2. IPHeader

该类的作用是提供了一个用于表示IP协议头部信息的数据结构，并且提供了一些方法用于处理IP地址和协议类型的转换。从而用于构建和解析网络数据包的头部信息。



头部信息主要包括：版本 头部长度的 服务类型 总长度 标识 标志和偏移 生存时间 协议 头部校验和 源IP地址 目的IP地址，同时编写了该类的set，get，toString方法。

为了便于将抓包获得的数据进行转化，这里还自定义了几个函数：

- integerToIp：用于转化IP地址
- protocolToString：用于根据协议号得出所用的协议
- typeOfServiceToString：用于解析服务类型

### 4.1.3.3. TCPHeader

该类的作用是提供一个数据结构来存储和操作TCP协议头部的信息，同时提供了一些方法来解析和转换TCP头部的各个字段。

- 头部信息主要包括：源端口 目标端口 序列号 应答号 数据偏移和标志位 窗口大小 校验和 紧急指针
- 同时自定义了函数：
- parseFlags：用于转化标志位。

## 4.2. networkInterface

### 4.2.1. 构成

该包下只有一个类，即NetworkInterface

### 4.2.2. 功能介绍

该类维护了两个重要的成员变量，分别为：

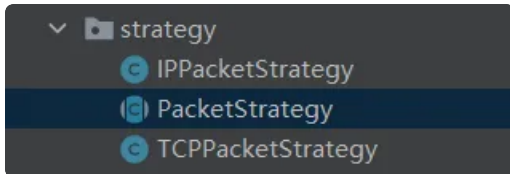
变量名称	类型	含义
name	String	接口名称

ipAddress	String	指定IP地址
-----------	--------	--------

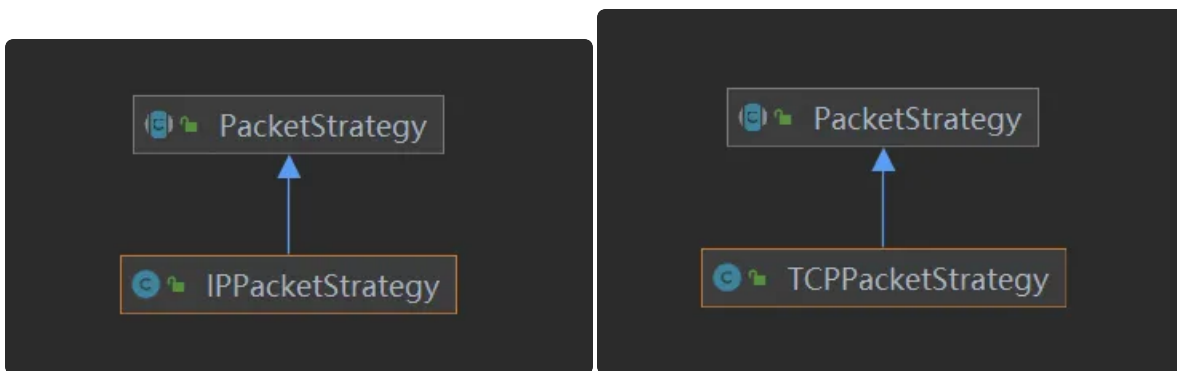
## 4.3. strategy

### 4.3.1. 构成

该包主要由IPPacketStrategy, TCPPacketStrategy, PacketStrategy构成：



### 4.3.2. 类之间的关系



IPPacketStrategy与TCPacketStrategy都继承了PacketStrategy抽象类

PacketStrategy：

```

1 public abstract class PacketStrategy {
2
3     private volatile AtomicInteger capturedCount;
4
5     abstract void processPacket(JTextArea outPutArea, Packet packetToUse,
6     PcapPacket packet, int packetOrder);
7
8     public Packet capture(JTextArea outPutArea, Packet packetToUse, int ca
9     ptureNum, Pcap pcap){
10         ...
11         processPacket(...)    ...//抓包细节请参考核心代码设计
12         ...
13     }
14
15     //转化十六进制
16     protected String bytesToHex(byte[] bytes) {
17         StringBuilder hexString = new StringBuilder();
18         for (int i = 0; i < bytes.length; i++){
19             String hex = Integer.toHexString(0xFF & bytes[i]);
20             if (hex.length() == 1) {
21                 hexString.append('0');
22             }
23             hexString.append(hex);
24             if (i < bytes.length - 1) {
25                 hexString.append(' ');
26             }
27         }
28         return hexString.toString();
29     }
30 }

```

PacketStrateg抽象类有一个原子变量用于计数，processPacket具体抓包逻辑用于留给子类实现，该抽象类不负责。这么做是参考了设计模式中的工厂模式，有许多好处，例如：

- 1.统一抓包逻辑：将抓包的核心逻辑封装在抽象类中，可以实现多个子类共享相同的抓包逻辑，避免了代码重复。
- 2.解耦抓包细节：抽象类负责抓包的整体流程控制，而具体的抓包细节由子类去实现，实现了逻辑的解耦。这样，当需要修改抓包细节时，只需要修改子类的实现，而不需要修改抽象类中的代码。
- 3.原子计数保证线程安全：使用原子变量 capturedCount 进行计数，保证了多线程环境下的计数操作的线程安全性，避免了多线程环境下的竞态条件问题。

- 4.灵活性和可扩展性：通过抽象类和子类的组合，可以实现不同的抓包策略，例如 IPPacketStrategy 和 TCPPacketStrategy，而且可以轻松地添加新的抓包策略子类，以满足不同的需求。
- 5.易于维护和扩展：将抓包逻辑封装在抽象类中，使得代码结构清晰，易于理解和维护。同时，基于抽象类和多态的设计，可以方便地扩展新的抓包策略，而不影响现有的代码逻辑。

### 4.3.3. 功能介绍

IPPacketStrategy与TCPStrategy的作用都是相同的，都是用于实际的抓包过程，只不过一个用于TCP解析，另外一个用于IP解析。

例如IP：

```
Java |
1 public class IPPacketStrategy extends PacketStrategy {
2
3     public void processPacket(JTextArea outPutArea, Packet packetToUse, Packet packet, int packetOrder){
4         ...
5         getElement(...);
6         ...
7     }
8
9     private void getElement(Packet packetToUse, Ip4 ip4){
10         ...
11     }
12
13
14 }
```

TCP类的结构完全一样。

getElement方法则是用于对抓取得到的包进行解析。

## 4.4. packet

### 4.4.1. 构成

该包下只有一个类：即Packet

### 4.4.2. 功能介绍

Packet中的私有成员如下：

变量名称	类型	含义
baseHeader	BaseHeader	头部数据
payloadData	String	数据负载
networkInterface	NetworkInterface	网络接口
packetStrategy	PacketStrategy	抓包策略
snapMaxLen	int	最大数据包长度
flags	int	抓包模式
timeout	int	最大等待超时时间
errBuf	StringBuilder	错误信息

在Packet的设计中，我们采用了设计模式的一大法则：**多用组装，少用继承** **【出自Head First设计模式】**

在Packet中使用了组合关系来包含一个BaseHeader对象。通过组合，Packet类可以利用BaseHeader提供的头部数据，而无需继承BaseHeader类。

这么做的好处有：

- 1.松耦合性：使用组合关系降低了Packet类与BaseHeader类之间的耦合度。Packet类不再依赖于BaseHeader类的具体实现细节，而只是利用了其提供的头部数据。这样的设计使得修改BaseHeader类不会影响到Packet类的实现，也使得Packet类更加灵活和可维护。
- 2.代码重用：通过组合关系，Packet类可以重用BaseHeader类中已经实现的功能和属性，而无需重新实现或继承BaseHeader类。这样可以减少重复代码的编写，提高了代码的复用性。
- 3.灵活性：使用组合关系可以轻松地在Packet类中替换或升级BaseHeader类的实现，而不会影响到Packet类的其他部分。这种灵活性使得系统更易于扩展和维护，能够应对未来的需求变化。
- 4.可定制性：通过组合关系，Packet类可以与不同版本或实现不同功能的BaseHeader类进行组合，从而实现不同的定制需求。这种灵活性使得可以根据具体场景选择合适的BaseHeader类，以满足特定的业务需求。

该总结同样参考 **【Head First设计模式】神书**

PacketStrategy类型的变量也是一样的道理，我们在后面可以通过**策略模式**的手段最大程度解耦程序

其他变量的作用主要用于实际的抓包过程，我们在构造方法中便会初始化这些成员：

```
Java |  
1      public Packet(BaseHeader baseHeader, NetworkInterface networkInterface  
2      ,  
3          PacketStrategy packetStrategy){  
4          this.baseHeader=baseHeader;  
5          this.networkInterface=networkInterface;  
6          this.packetStrategy=packetStrategy;  
7          this.snapMaxLen=64*1024;  
8          this.flags= Pcap.MODE_PROMISCUOUS;  
9          this.timeout=10*1000;  
10         this.errBuf=new StringBuilder();  
11     }
```

综上所述，通过使用组合关系来包含一个BaseHeader对象以及PacketStrategy对象，Packet类能够充分利用BaseHeader与PacketStrategy提供的功能和属性，同时保持了松耦合、代码重用、灵活性和可定制性。同时其他成员变量用于Packet的实际抓包过程。

## 4.5. exceptions

### 4.5.1. 构成

exceptions包下只有一个类：NetException

### 4.5.2. 功能介绍

```
1 public class NetException extends Exception{
2     public NetException(){}
3
4     public NetException(String errorMessage){
5         super(errorMessage);
6     }
7 }
```

继承自Exception类，由于在网络抓包的过程中会遇到许多异常需要处理的情况，因此这里自定义了一个异常类用于处理这些异常。

## 4.6. enums

### 4.6.1. 构成

改包下只有一个类：Choose

### 4.6.2. 功能介绍

由于我们要分析的类型只有TCP与IP，符合枚举的特性，因此在Choose中声明枚举类，使得代码更加清晰：

```
1 public enum Choose {
2     TCP("TCP"),
3     IP("IP");
4
5     private final String type;
6
7     Choose(String type) {
8         this.type=type;
9     }
10
11     public String getType() {
12         return type;
13     }
14 }
```

## 4.7. factory

### 4.7.1. 构成

该包下只有一个类：PacketFactory

### 4.7.2. 功能介绍

这段代码定义了一个名为PacketFactory的工厂类，用于根据传入的BaseHeader对象和NetworkInterface对象创建对应的Packet对象。工厂类中的createPacket方法根据BaseHeader的类型来决定使用哪种Packet策略来创建。

createPacket是一个静态方法，因此无需创建PacketFactory的实例即可调用该方法。

这个类的作用是根据给定的BaseHeader对象和NetworkInterface对象来创建对应的Packet对象，并根据BaseHeader的类型选择适当的抓包策略。这样的设计使得Packet对象的创建过程更加灵活，并且将对象的创建逻辑封装在工厂类中，提高了代码的可维护性和可扩展性。

在这里我们使用的是工厂模式的设计思想，他有许多好处，比如：

1.封装创建逻辑：工厂方法封装了对象的创建逻辑，使得调用方无需了解对象的创建过程和细节。这样可以简化调用方的代码，并提高了代码的可读性和可维护性。

2.解耦合：通过工厂方法，客户端代码与具体的对象创建过程解耦合。客户端代码只需要知道如何使用工厂方法来获取所需的对象，而无需关心对象的创建细节。这种解耦合提高了系统的灵活性和可维护性，使得系统更容易进行修改和扩展。

3.灵活性：工厂方法根据传入的参数来确定要创建的对象类型，从而使得系统更加灵活。例如，如果需要新增其他类型的BaseHeader或者Packet对象，只需修改工厂方法中的逻辑即可，而不需要修改调用方的代码。这种灵活性使得系统更易于扩展和维护。

4.隐藏实现细节：工厂方法隐藏了对象的创建细节，使得客户端代码无需关心具体的对象创建过程。这样可以降低客户端代码与具体对象的耦合度，同时提高了代码的可维护性和可重用性。



```

1 public class PacketFactory {
2     public static Packet createPacket(BaseHeader header,
3                                     NetworkInterface networkInterface) {
4         if (header instanceof IPHeader) {
5             return new Packet(header, networkInterface, new IPPacketStrateg
6 y());
7         } else if (header instanceof TCPHeader) {
8             return new Packet(header, networkInterface, new TCPHeaderStrate
9 gy());
10        }
11        return null;
12    }
13 }

```

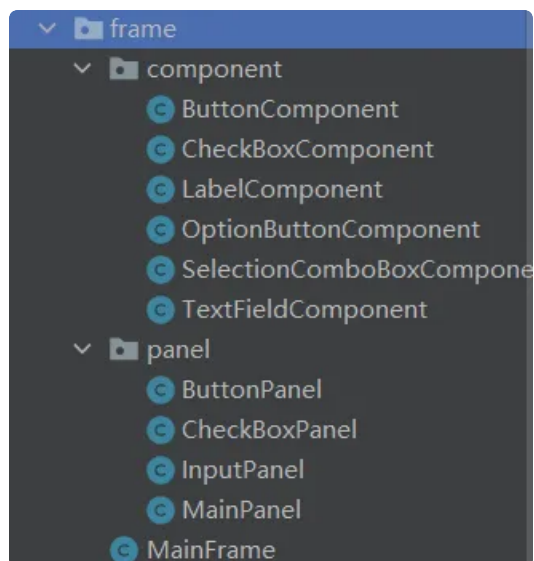
该工厂方法与策略模式可以紧密结合在一起，工厂方法用于创建不同类型的Packet对象，而策略模式则用于定义不同的抓包策略。通过工厂方法，根据不同的BaseHeader类型选择不同的策略对象，从而实现了对象的动态创建和策略的动态切换。

## 4.8. frame

该包主要用于swing图形化，且最大程度提高的代码的解耦度以及可扩展度

### 4.8.1. 构成

该模块构成较为复杂，主要由component， panel包以及一个类Main\_Frame构成：



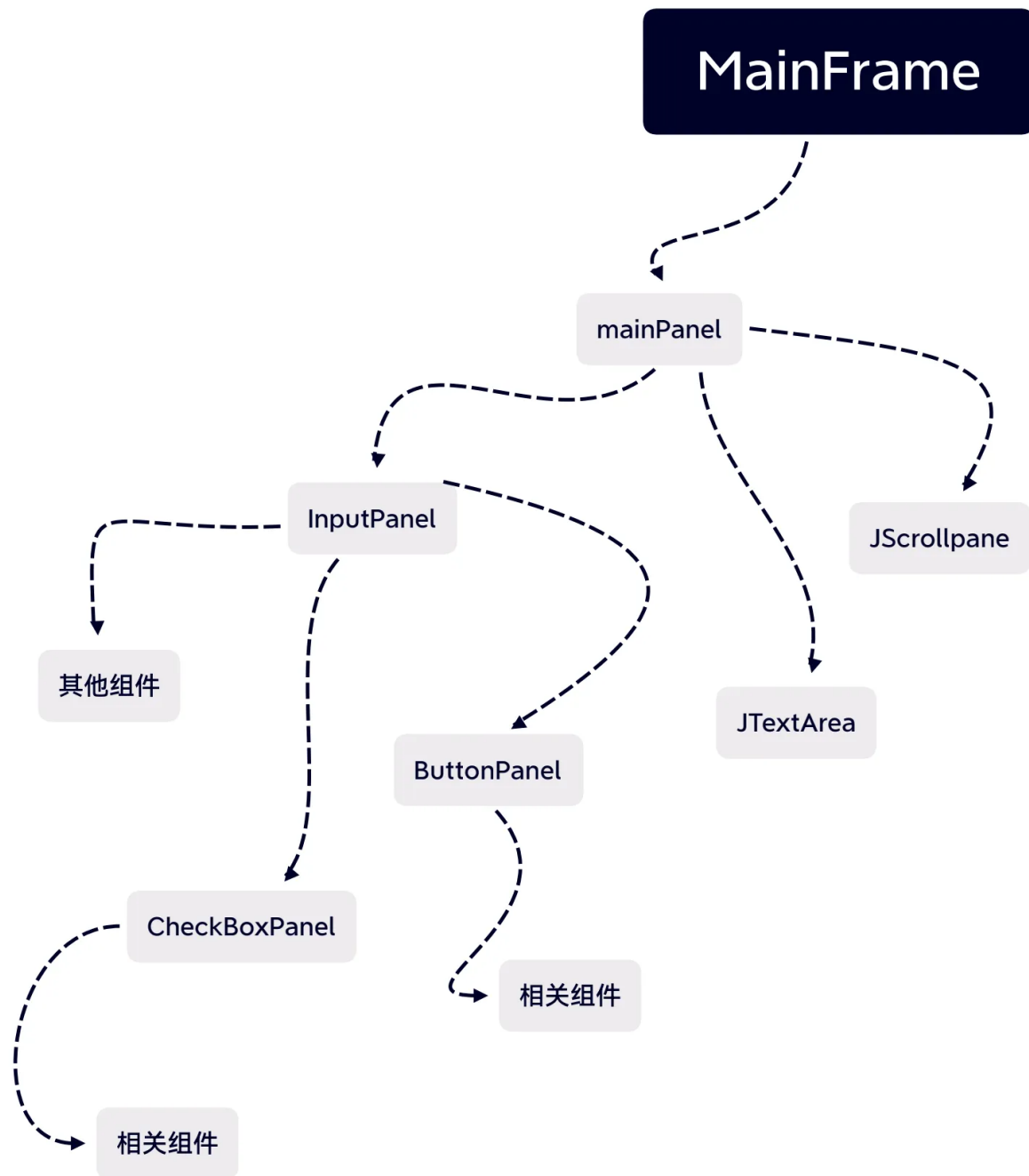
### 4.8.2. 类/模块之间的关系

component模块中主要定义了一组组件元素以及他们的布局情况

panel模块主要用于定义一组面板，面板嵌套着component组件，以及其他的面板

总的来说，Component表示可视化用户界面元素，Panel用于组织和管理这些元素，两者共同协作，帮助我构建出具有良好布局和交互的Swing用户界面。

整体关系图大体如下：



### 4.8.3. 功能介绍

4.8.3.1. component

组件类	名称	作用
ButtonComponent	按钮组件	用来让用户开始抓包的按钮
CheckBoxComponent	复选框组件	用来让用户指定是否选中指定的IP地址
LabelComponent	标签组件	用来标识其他组件的含义
OptionButtonComponent	选项按钮组件	用来选择解析TCP还是IP协议
SelectionComboBoxComponent	下拉框组件	用来列出本机的所有网络接口
TextFieldComponent	文本框组件	用来填写网络接口与IP地址

这6个组件类都包含两个成员变量，一个是原生swing库提供的对应组件，例如JButton，JCheckBox等等；另外一个则是GridBagConstraints类的constraints。前者就是最核心的组件，后者则是用来指定他们的布局，以ButtonComponent为例：

```

1 public class ButtonComponent {
2     private JButton buttonComponent;
3
4     private GridBagConstraints constraints;
5
6
7     public ButtonComponent(String words,int gridx,int gridy,int gridwidth,
8 int gridheight) {
9         constraints=new GridBagConstraints();
10        constraints.fill = GridBagConstraints.HORIZONTAL; // 设置组件水平填充
11        constraints.insets = new Insets(5, 5, 5, 5); // 设置组件之间的间距
12        constraints.gridx=gridx;
13        constraints.gridy=gridy;
14        constraints.gridheight=gridheight;
15        constraints.gridwidth=gridwidth;
16        buttonComponent=new JButton(words);
17    }
18    .....//省略其他方法
19
20    public void setStartActionListener(JTextArea outPutArea, ButtonPanel buttonPanel,
21                                     TextFieldComponent interfaceTextField,
22                                     TextFieldComponent ipTextField,
23                                     SelectionComboBoxComponent selectionComboBoxComponent) {
24
25        buttonComponent.addActionListener(new ActionListener() {
26            @Override
27            public void actionPerformed(ActionEvent e) {
28                ...//此处进行省略，详细请见第五节核心代码设计
29            }
30        });
31    }
32 }

```

ButtonComponent类是一个封装了按钮组件的创建、配置和事件处理逻辑的类，**通过构造函数初始化按钮的文本内容和布局约束信息**，并提供方法用于添加点击事件监听器，实现按钮点击时的特定抓包操作，从而使得按钮的创建和交互变得简单且灵活。

其他类跟该类的设计十分相近。

#### 4.8.3.2. panel

面板类名	名称	作用
ButtonPanel	按钮面板	主要用于存放选中TCP/IP的选项组件
CheckBoxPanel	复选框面板	主要用于存放让用户选中是否指定IP地址的面板
InputPanel	输入面板	核心面板，初始化所有的组件以及相关面板
MainPanel	主面板	主要存放InputPanel以及JTextArea抓包结果显示区域

注：

- 1.每一个面板几乎都在构造函数中定义好了面板中的组件布局，构造方法以面板中的Init方法结尾，该方法用于将所有的这些组件加入到面板当中去。
- 2.每一个面板类都继承了JPanel类。

这几个面板相互交叉在一起，我们进行细致的分析：

#### 4.8.3.2.1. ButtonPanel

成员变量如下：

类	名称	作用
LabelComponent	标签	用于提示用户
ButtonGroup	TCP/IP按钮组	确保TCP与IP同时只有一个被选中
OptionButtonComponent	TCP选框	选中TCP
OptionButtonComponent	IP选框	选中IP
GridBagComconstrainits	布局	用于布局

整个ButtonPanel类封装了一组用于选择模型的标签和两个选项按钮，通过GridBagLayout布局管理器实现布局，提供了一系列方法用于获取和设置组件以及布局信息。

在构造方法中我们已经定义好了他们的布局情况：

```

1  public ButtonPanel(String words,String buttonOne,String buttonSecond,int
   gridx,int gridy,int gridwidth,int gridheight){
2      constraints=new GridBagConstraints();
3      buttonGroup=new ButtonGroup();
4      constraints.fill = GridBagConstraints.HORIZONTAL; // 设置组件水平填
   充
5      constraints.insets = new Insets(5, 5, 5, 5); // 设置组件之间的间距
6      constraints.gridx=gridx;
7      constraints.gridy=gridy;
8      constraints.gridheight=gridheight;
9      constraints.gridwidth=gridwidth;
10     modelChooseLabel=new LabelComponent(words,1,3,1,1);
11     IPButton=new OptionButtonComponent(buttonOne,2,3,1,1);
12     TCPButton=new OptionButtonComponent(buttonSecond,3,3,1,1);
13     buttonGroup.add(IPButton.getOptionButtonComponent());
14     buttonGroup.add(TCPButton.getOptionButtonComponent());
15     this.Init();
16 }

```

Init方法用于将组件加入到面板当中去：

```

1  public void Init(){
2      this.setLayout(new GridBagLayout());
3      this.add(modelChooseLabel.getLabel(),modelChooseLabel.getConstraint
   s());
4      this.add(IPButton.getOptionButtonComponent(),IPButton.getConstraint
   s());
5      this.add(TCPButton.getOptionButtonComponent(),TCPButton.getConstrai
   nts());
6  }

```

#### 4.8.3.2.2. CheckBoxPanel

成员变量为：

类	名称	作用
CheckBoxComponent	IP选中框	用于让用户指定是否指定IP地址
GridBagComconstraints	布局	用于布局

设计思想与ButtonPanel一样，这里不加以赘述。

4.8.3.2.3. InputPanel

核心面板，成员变量为：

类	名称	作用
LabelComponent	网络接口标签	提示用户
LabelComponent	IP地址标签	提示用户
LabelComponent	网卡选择标签	提示用户
TextFieldComponent	网络接口文本框	用户可输入网络接口
TextFieldComponent	IP地址文本框	用户可指定IP地址
SelectionComboBoxComponent	网卡选择	用户可在众多网卡中进行选择
CheckBoxPanel	IP是否选中面板	用户可选择是否指定IP地址
ButtonPanel	TCP/IP按钮面板	用户可选择解析TCP/IP
LabelComponent	抓包数目标签	提示用户
TextFieldComponent	抓包数目	用户可指定抓包数目
ButtonComponent	抓包按钮	用户点击即可抓包

在该面板的构造方法中我们通ButtonPanel进行了初始化工作：



```

1  public InputPanel(){
2      //第0行
3      interfaceLabel=new LabelComponent("网络接口:",0,0,1,1);
4      interfaceTextField=new TextFieldComponent(30,1,0,1,1);
5      interfaceChooseLabel=new LabelComponent("本地网络接口选择:",2,0,1,1)
6      ;
7      selectionComboBoxComponent = new SelectionComboBoxComponent(3,0,1,
8      1);
9      selectionComboBoxComponent.setActionListener(interfaceTextField);
10
11     //第1行
12     ipLabel=new LabelComponent("目标IP地址:",0,1,1,1);
13     ipTextField=new TextFieldComponent(30,1,1,1,1);
14     ipLabel.getLabel().setVisible(false);
15     ipTextField.getTextFieldComponent().setVisible(false);
16
17     //第2行
18     checkBoxPanel=new CheckBoxPanel("是否指定IP",2,2,1,1);
19     checkBoxPanel.setActionListener(ipLabel,ipTextField);
20
21     //第3行
22     buttonPanel=new ButtonPanel("选择解析模式:", "IP", "TCP", 3,3,1,1);
23
24     //第4行
25     captureNumLabel=new LabelComponent("抓包数目:",0,4,1,1);
26     captureNumTextField=new TextFieldComponent(30,1,4,1,1);
27
28     //第5行
29     buttonStart=new ButtonComponent("点击开始抓包",2,5,1,1);
30     this.Init();
31 }

```

在Init方法中我们将这一系列组件或面板都加入到了input面板当中去。

这样最大程度提高了程序的解耦程度以及复用性。

#### 4.8.3.2.4. MainPanel

成员变量如下：

类	名称	作用
InputPanel	输入面板	核心面板，显示大部分组件
JTextArea	显示区域	用于显示抓包结果

JScrollPane	滚动面板对象	支持滚动查看抓包结果
-------------	--------	------------

大体思想与ButtonPanel一样，这里不加以赘述。

#### 4.8.3.3. MainFrame

只有一个成员变量，即MainPanel：

```
Java |
1  public MainFrame(){
2      this.setTitle("网络抓包程序");
3      this.setSize(1000, 500);
4      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5      mainPanel=new MainPanel();
6      this.add(mainPanel);
7  }
```

简单且清晰明了的设计。

#### 4.8.3.4. 设计的好处

这样设计采用了面向对象的思想，将界面的不同组件拆分成了独立的类，通过组合的方式构建复杂的界面。同时拥有以下的好处：

- 1.模块化：每个组件（如标签、文本框、按钮等）都有其专门的类来管理，使得代码结构清晰，易于维护和扩展。每个组件的功能都在相应的类中实现，降低了代码的耦合度。
- 2.复用性：每个组件都是独立的，可以在不同的界面中重复使用。例如，如果其他界面也需要一个类似的输入面板，只需实例化InputPanel对象即可，无需重复编写相同的代码。
- 3.可维护性：由于代码结构清晰，每个组件的功能都被封装在单独的类中，因此易于定位和修复bug。同时，当需要对界面进行修改或添加新功能时，也更容易进行管理和维护。
- 4.可扩展性：由于每个组件都是独立的，可以根据需求方便地添加新的组件或修改现有组件的行为，而不会对其他部分造成影响。这种设计使得系统具有良好的灵活性，能够适应不断变化的需求。

**一句化概括，就是核心思想是面向对象进行设计而不是面向过程进行设计。**

## 4.9. thread

### 4.9.1. 构成

该包下只有一个类：ThreadPool

### 4.9.2. 功能介绍

```
1 public class ThreadPool {
2     private static final ExecutorService threadPool;
3
4     private static final Integer threadNum=3;
5
6     static{
7         threadPool= Executors.newFixedThreadPool(threadNum);
8     }
9
10    public static void execute(Runnable runnable){
11        threadPool.execute(runnable);
12    }
13 }
```

我们实现了一个简单的线程池，其中包括一个静态成员变量 threadPool 以及一个静态方法 execute。通过静态初始化块，在类加载时初始化了一个固定大小的线程池，大小由 threadNum 变量指定。execute 方法允许向线程池提交任务，使得任务在后台线程中执行，从而实现了任务的异步执行和线程池资源的有效利用。

## 5. 核心代码设计

### 5.1. 监听函数

#### 5.1.1. 网络接口监听显示

本地网络接口的显示主要与SelectionComboBoxComponent组件相关联在一起

当在构造函数中初始化的时候：

```

1  public SelectionComboBoxComponent(int gridx,int gridy,int gridwidth,int
   t gridheight) {
2      ...
3      ...
4  try {
5      netInit();
6  } catch (NetException e) {
7      e.printStackTrace();
8  }
9
10 }
```

最后会进行一个叫做netInit的方法：

```

1  public void netInit() throws NetException{
2      StringBuilder errBuf = new StringBuilder();
3      List<PcapIf> allDevs = new ArrayList<PcapIf>(); // Will be filled
   with
4      int r = Pcap.findAllDevs(allDevs, errBuf);
5  if (r == Pcap.NOT_OK || allDevs.isEmpty()) {
6      throw new NetException("Can't read list of devices, error is %
   s"+ errBuf);
7  }
8      // 迭代找到的所有网卡
9      int i = 0;
10 for (PcapIf device : allDevs) {
11     String description = (device.getDescription() != null) ? device
   e
12         .getDescription() : "No description available";
13     System.out.printf("#%d: %s [%s]\n", i++, device.getName(),
14         description);
15     selectionComboBoxComponent.addItem(device.getName());
16 }
17 }
```

该方法将迭代找出本机的所有网络接口并且显示在文本框中，这也是为什么程序启动的时候就可以显示出来所有的网络接口。

### 5.1.2. 抓包监听

该功能主要与ButtonComponent组件相结合在一起，核心代码就在于：

```

1  public void setStartActionListener(JTextArea outPutArea, ButtonPanel butto
   nPanel,
2      TextFieldComponent interfaceTextField, TextFieldComponent i
   pTextField,
3      SelectionComboBoxComponent selectionComboBoxComponent,
4      TextFieldComponent captureNumTextFi
   eld) {
5
6  buttonComponent.addActionListener(new ActionListener() {
7      @Override
8      public void actionPerformed(ActionEvent e) {
9          Packet packet=null;
10         Choose buttonSelected=buttonPanel.getIPButton().getOptionButtonCom
   ponent().
11             isSelected()==true?Choose.IP:Choose.TCP;
12         if(buttonSelected==Choose.IP){
13             packet = PacketFactory.createPacket(new IPHeader(),
14             new NetworkInterface(interfaceTextField.getTextFieldCo
   mponent().getText(),
15             ipTextField.getTextFieldComponent().getTex
   t()));
16         }else{
17             packet = PacketFactory.createPacket(new TCPHeader(),
18             new NetworkInterface(interfaceTextField.getTextFieldCo
   mponent().getText(),
19             ipTextField.getTextFieldComponent().getTex
   t()));
20     }
}

```

当用户一点击开始抓包的按钮，首先会根据用户的选择确认是TCP解析还是IP解析，之后就会根据不同的策略调用PacketFactory工厂类的静态方法createPacket创建出来的不同包，最后调用Packet中的capture方法开始抓包逻辑的进行。我们可以看出来这一段代码很好体现了策略模式以及工厂模式的精华所在：

- 1.灵活性和可扩展性：通过策略模式，根据用户选择的不同，动态地选择合适的策略（即不同的包类型），从而实现了灵活的抓包行为。如果需要添加新的包类型或者修改现有的包类型，只需要修改PacketFactory 工厂类，而不需要修改其他部分的代码。
- 2.解耦合：策略模式将不同的算法或行为封装在独立的类中，使得每个类都只关注特定的任务，实现了代码的解耦合。这样，当需要修改或者扩展某个特定的抓包策略时，不会影响其他部分的代码，提高了代码的可维护性和可扩展性。
- 3.简化代码：工厂模式通过封装对象的创建过程，隐藏了对象的创建细节，使得客户端代码更加简洁清晰。在这个例子中，通过工厂模式创建不同类型的包对象，避免了客户端代码直接依赖于具体的包类，

降低了代码的耦合度。

4.符合开闭原则：策略模式和工厂模式都符合开闭原则，即对扩展开放，对修改关闭。通过新增具体策略类或者工厂方法，可以轻松地扩展系统的功能，而不需要修改原有的代码。

参考自《Head First设计模式》中的开闭原则描述

## 5.2. 抓包函数

Packet中的capture函数：

```
1 public void capture(JTextArea outPutArea,String captureNumString) throws Ne  
   tException {  
2     int captureNum = Integer.parseInt(captureNumString);  
3     Pcap pcap = netInit(outPutArea, this);  
4     packetStrategy.capture(outPutArea,this,captureNum,pcap);  
5 }
```

首先会进行一个netInit方法初始化操作：

```

1  protected Pcap netInit(JTextArea outPutArea,Packet packetToUse) throws Net
   Exception {
2      Pcap pcap = Pcap.openLive(packetToUse.getNetworkInterface().getNam
   e(), packetToUse.getSnapMaxLen(),
3          packetToUse.getFlags(), packetToUse.getTimeout(),
4          packetToUse.getErrBuf());
5      if (pcap == null) {
6          outPutArea.append("无效网卡, 请重新填写.\n");
7          throw new NetException("Error while opening device for captur
   e: " + packetToUse.getErrBuf());
8      }
9      if(!packetToUse.getNetworkInterface().getIpAddresses().equals(""))
   {
10         // 设置过滤器
11         PcapBpfProgram filter = new PcapBpfProgram();
12         String expression = "host " + packetToUse.getNetworkInterface(
   ).getIpAddresses();
13         int optimize = 0; // 0表示不优化
14         int netmask = 0xFFFFFFFF; // 掩码, 这里使用默认掩码
15         int compileResult = pcap.compile(filter, expression, optimize,
   netmask);
16         if (compileResult != Pcap.OK) {
17             throw new NetException("Error setting filter: "+ packetToU
   se.getErrBuf());
18         }
19         pcap.setFilter(filter);
20     }
21     return pcap;
22 }

```

在这个方法中打开了与网络接口的通道并且根据用户的选择决定是否要开启对于网络IP地址的过滤器, 如果有错误的话则打印出来。

PacketStrategy中的capture抓包函数:

```

1 public Packet capture(JTextArea outPutArea, Packet packetToUse, int captureNum, Pcap pcap){
2     capturedCount=new AtomicInteger(captureNum);
3     new Thread(() -> {
4         PcapPacketHandler<String> packetHandler = new PcapPacketHandler<String>() {
5             @Override
6             public void nextPacket(PcapPacket packet, String user) {
7                 int packetSum = capturedCount.decrementAndGet();
8                 ThreadPool.execute(()->processPacket(outPutArea,packetToUse,packet,captureNum-packetSum));
9                 if(packetSum==0){
10                     pcap.breakloop();
11                 }
12             }
13         };
14         pcap.loop(Pcap.LOOP_INFINITE,packetHandler, "start capturing!!!");
15         pcap.close();
16     }).start();
17     return null;
18 }

```

首先初始化AtomicInteger成员变量表示要抓包的数量，开启一个线程去处理抓包逻辑，避免对于主线程的堵塞。在该子线程中将抓包程序processPacket交给线程池去运行，然后判断是否已经抓到了指定数目的包，如果是则停止loop抓包循环，关闭网络接口通道。

processPacket:

TCP与IP具体的实现逻辑在这个方法中不尽相同但是大同小异，这里以ip为例子进行说明：



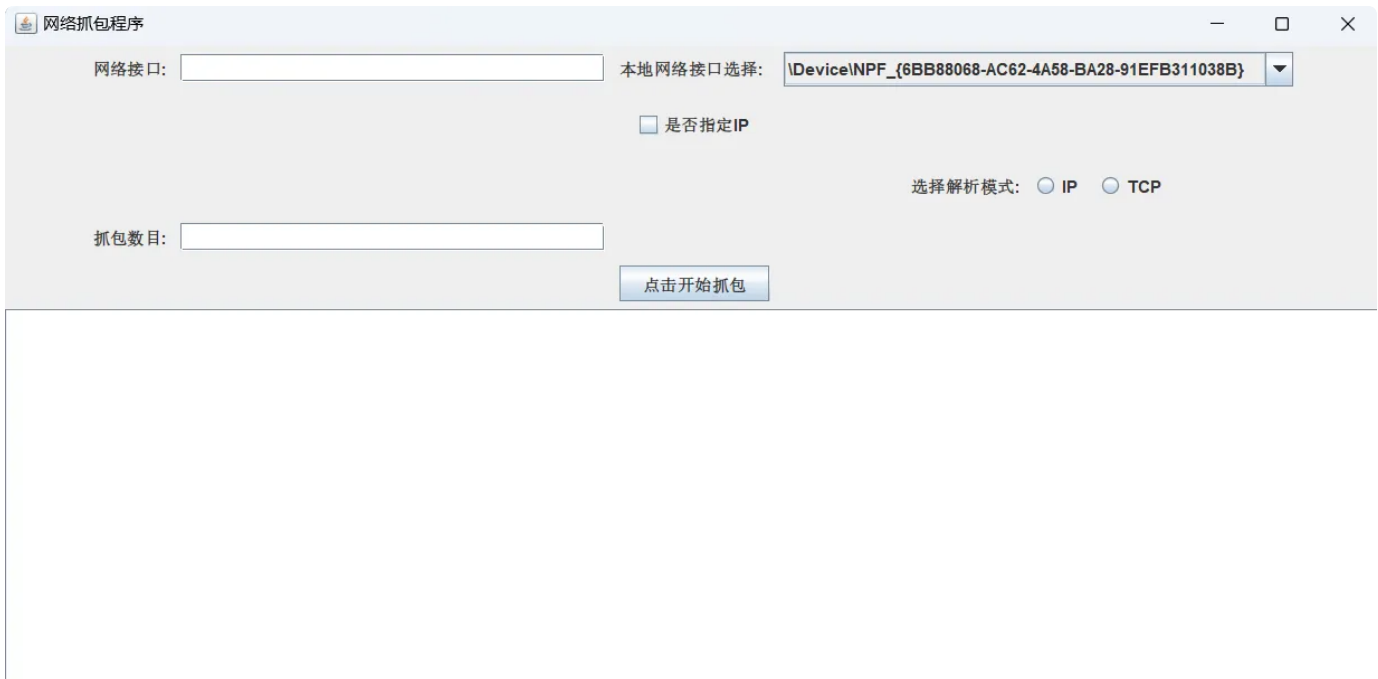
```
1      public void processPacket(JTextArea outPutArea, Packet packetToUse,
2                                PcapPacket packet,int packetOrder){
3          Ip4 ip4 = new Ip4();
4          if (packet.hasHeader(ip4)) {
5              getElement(packetToUse,ip4);
6              SwingUtilities.invokeLater(() -> {
7                  outPutArea.append("第"+packetOrder+"个包:\n"+
8                                  packetToUse.getBaseHeader()+"负载数据:"+
9                                  packetToUse.getPayloadData()+"\n-----\n");
10             });
11          }
12      }
```

主要就是实现了一个抓包处理方法，首先检查原始包中是否存在 IP 包头部，如果存在则提取关键信息填充到指定的包对象中，并在 GUI 界面中显示包的顺序、头部信息和载荷数据。（getElement即为填充方法，这里不加以赘述）。

## 6. 产品使用说明书

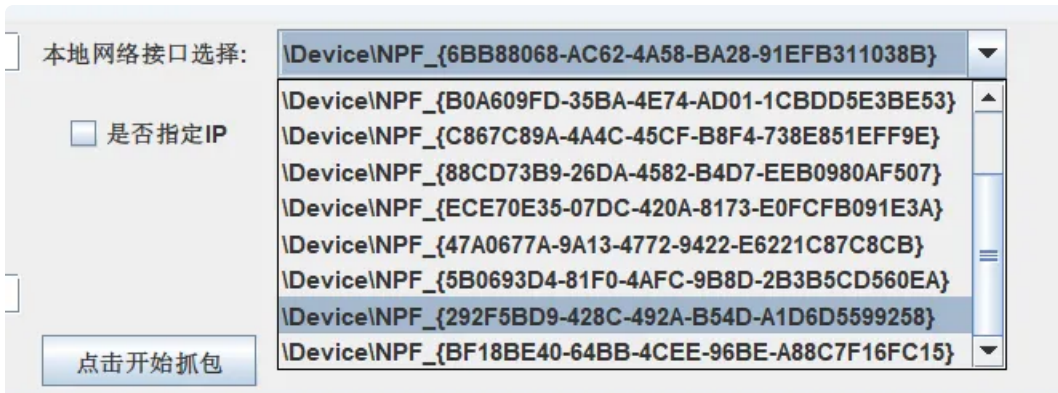
### 6.1. 程序主界面

程序主界面如下，上半部分用于用户操作，下半部分用于显示抓包结果：



## 6.2. 网络接口选择

本地网络接口选择可以下拉开来进行选择，一旦选中则会自动复制到左侧的网络接口文本框中：

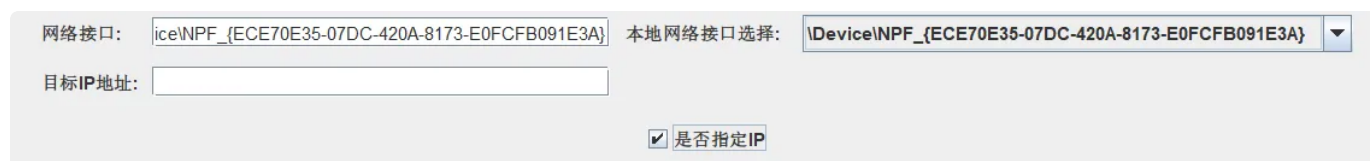


**注：最好使用WIFI的网络接口，否则不能保证有流量通过，接口的信息已经打印在了命令行中。**

我们使用\Device\NPF\_{ECE70E35-07DC-420A-8173-E0FCFB091E3A}该网络接口进行演示。

## 6.3. IP地址选择

勾选是否指定IP，即可填写指定的ip地址：

A network configuration interface with a light gray background. It contains two rows of labels and input fields. The first row has '网络接口:' followed by a text box containing 'ice\NPF\_{ECE70E35-07DC-420A-8173-E0FCFB091E3A}' and '本地网络接口选择:' followed by a dropdown menu showing the same path. The second row has '目标IP地址:' followed by an empty text box. At the bottom right, there is a checkbox labeled '是否指定IP' which is checked.

网络接口: ice\NPF\_{ECE70E35-07DC-420A-8173-E0FCFB091E3A} 本地网络接口选择: \Device\NPF\_{ECE70E35-07DC-420A-8173-E0FCFB091E3A} ▼

目标IP地址:

☒ 是否指定IP

这里将会过滤排除掉那些源或目的地不是用户目标IP地址填写的包，如若不想过滤不要勾选即可。

## 6.4. 协议解析选择

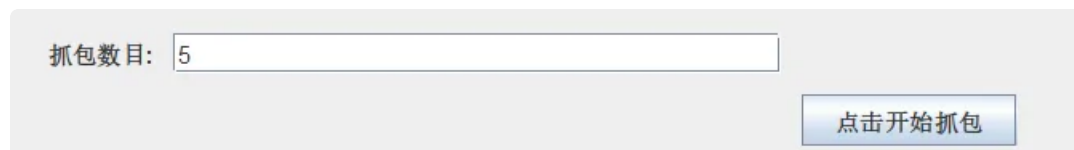
A light gray rectangular box containing the text '选择解析模式:' followed by two radio buttons. The first radio button is selected and labeled 'IP'. The second radio button is unselected and labeled 'TCP'.

选择解析模式: ☒ IP ☐ TCP

在主界面右中部可以选择解析IP头部又或是TCP头部。

## 6.5. 抓包操作

用户可以填写想要抓包的数目，然后点击开始抓包即可：

A light gray rectangular box containing a label '抓包数目:' followed by a text box with the number '5'. To the right of the text box is a blue button with white text that says '点击开始抓包'.

抓包数目:

点击开始抓包

## 6.6. 抓包效果展示

```
第5个包:(由于是多线程抓包,因此该顺序为乱序)
IP 头部信息:
版本:4
头部长度:20 字节
服务类型:0 (Priority: 0, Normal Delay, Normal Throughput)
总长度:63 字节
标识:64239
标志和偏移:0
生存时间:64 跳
协议:17 (UDP)
头部校验和:29106
源IP地址:10.129.240.130
目标IP地址:10.3.9.6
负载数据:00 35 ee 1b 00 ee 74 da ca 55 81 80 00 01 00 05 00 00 00 00 06 70 64 73 61 70 69 06 61 6c 69 70 61 6e 03 63 6f 6d 00 00 01 00 01 c0 0c 00 05 00 01 00 00 00 50 00 2c 07 62 68
```

如图即为IP头部分析

```
第4个包:(由于是多线程抓包,因此该顺序为乱序)
TCP 头部信息:
源端口: 12075
目标端口: 80
序列号及其raw: 3958975879
应答号及其raw: 3487978756
数据偏移和标志位: 24
ACK标志位被设置 PSH标志位被设置
窗口大小: 514字节
校验和: 17928
紧急指针: 0
负载数据:50 4f 53 54 20 2f 73 63 61 6e 20 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a 20 73 2e 66 2e 33 36 30 2e 63 6e 0d 0a 41 63 63 65 70 74 3a 20 2a 2f 2a 0d 0a 43 6f 6e 6e 65 63 74
```

如图为TCP头部解析

注：由于是多线程抓包，因此包的号码并没有实际参考价值

用户可以在在最右侧向上或向下进行翻滚查看其他抓取得到的包：

