# 第10讲 网络抓包与发包程序设计(上)

[需要2个教学单位完成，此为第1个单元]

**教学与实践目的：**学会网络抓包及发送特定网络包的程序设计技术。

通常情况下网卡（工作在链路层）在接收数据时往网络层传递3类包：广播包、与自己IP地址一致的单播包、已加入某组的组播包，在这种情况下，高层应用只能收到上述3类数据包。

我们前几讲的程序设计教学中，包括基于TCP Socket的网络应用（网络对话、FTP、Email及HTTP），以及基于UDP Socket的网络通信应用中，网卡只往网络层传递与自己IP地址一致的单播包。

**抓包的思想是**：流经网卡所有的有效包（共6种有效包，见《计算机网络》的链路层）都要抓，所以抓包程序都是工作在链路层或网络层（不考虑应用层的端口参数问题）。网络抓包软件或工具，也叫网络嗅探器，目前网上有许多免费的嗅探软件。

java语言虽然在TCP/UDP传输方面给予了良好的定义，但无法直接访问底层网络数据，但对于网络层以下的控制无能为力。Jpcap扩展包弥补了这一点，Jpcap是一个可以让java工作在链路层的类库； Jpcap实际上并非真正实现了对数据链路层的控制，而是一个中间件，Jpcap通过调用WinPcap/libpcap，给java语言提供一个公共的接口，从而实现了平台无关性。

这两个教学单元掌握基于Jpcap类库包（第三方软件包）自己动手开发抓包发包工具软件。

## 一、Jpcap简介

### 1. Jpcap的工作环境需求

（1）安装JDK;

（2）安装WinPcap（部分机房已经安装）或LibPcap：

* 对于Windows平台（http://www.winpcap.org/)需要安装[WinPcap](https://oc.our-web.org/download?file=WinPcap_4_1_3.exe)4.1.3，但其对windows 10的系统支持不好，需要管理员模式下运行net start npf解决兼容问题。也可以用[Npcap](https://oc.our-web.org/download?file=npcap-1.00.exe)代替（安装的时候勾选“Install Npcap in WinPcap API-compatiable Mode”，其它选项保持默认）；
* 对于UNIX平台（http://ee.lbl.gov/)需要安装 LibPcap。

### 2. Jpcap的类库结构

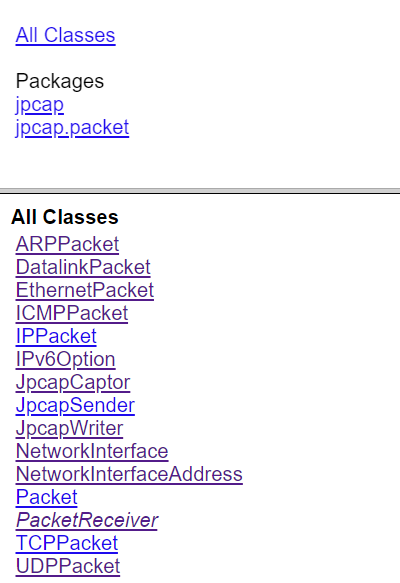
Jpcap0.7版本共有15个类(接口)，如图10.1所示，分别简介如下： 

图10.1 Jpcap所有的类

JpcapCaptor类：继承了java.lang.Object，它是Jpcap的核心类，一个JpcapCaptor对象实例相当于是对一个网卡的实现，通过对JpcapCaptor对象的调用，实现网络数据包的抓取；它提供了一系列静态方法调用如：获取网卡列表，获取某个网卡上的JpcapCaptor对象；

Packet类：是所有被捕获的数据包的基类，继承了java.lang.Object；

ARPPacket类：描述了ARP/RARP包，继承了Packet类；

IPPacket类：描述了IP包，继承了Packet类，支持IPv4和IPv6；

TCPPacket类：描述TCP包，继承了IPPacket类；

UDPPacket类：描述了UDP包，继承了IPPacket类；

ICMPPacket类：描述了ICMP包，继承了IPPacket类；

DatalinkPacket类：抽象类，描述了数据链路层的包，它继承了java.lang.Object；

EthernetPacket类：描述了以太帧包，继承DatalinkPacket类；

IPAddress类：继承了java.lang.Object，描述了IPv4和IPv6地址，包含了将IP地址转换为域名的方法；

IPv6Option类：继承了java.lang.Object，描述了IPv6选项报头；

JpcapSender类 ：用来发送可以用来发送 IPPacket及其子类，继承了java.lang.Object；

JpcapWriter类 ：用来将一个被捕获的数据包保存到文件，继承了java.lang.Object；

PacketReceiver接口：数据包处理器接口定义，要处理收到的数据包，必须编写这个接口的实现类，在JpcapCaptor对象的loopPacket或processPacket方法中调用。

**关于Jpcap的API详细索引，**可下载[Jpcap API Doc参考文档](https://oc.our-web.org/download?file=JPcap_javadoc.zip)（下载解压后，打开index.html）。

## 二、Jpcap的安装

登录[OC课堂资源网站](https://oc.our-web.org/)下载jpcap32\_64.zip，压缩包中有32位和64位版本的两个文件夹，每个文件夹下都有Jpcap.dll和jpcap.jar两个文件，根据自己的操作系统进行选择。针对Windows 环境：

（1） 复制＂Jpcap.dll＂到java jdk安装目录的[jdk目录]\bin，以及[jre目录]\bin，对于安装了多个jdk(jre)版本的系统，复制到你项目使用的那个jdk(jre)目录下的bin中；

（2） 将＂jpcap.jar＂导入到自己项目中，idea添加外部jar的方法参见第7讲的扩展练习二；

**注意F&Q：**

上述工作完成后，如果后续编写的程序运行不正常，查找是否出现下列问题:

（1）若错误提示是“no dependence libray”, WinPcap没有安装或未装好, 重装WinPcap;

（2）如果出现”no Jpcap in …java library.path”或“no ipcap.dll in jdk/bin……”等原因是没有拷贝Jpcap.dll到正确的位置，或者jpcap.jar没有导入到项目中；

（3）若出现“不是有效的Win32应用程序”，可能是WinPcap的版本问题，应下载一个64/32位兼容的版本。

（4）程序能运行，但检测不到可用的网卡，可能是运行在windows 10系统中，应该选择Npcap代替（或者管理员模式下的cmd中，运行net start npf）

## 三、Jpcap的基本用法

下面的测试代码，需要导入以下的包：

import jpcap.JpcapCaptor;  
import jpcap.NetworkInterface;  
import jpcap.NetworkInterfaceAddress;  
import jpcap.PacketReceiver;  
import jpcap.packet.Packet;

### 1. 获取网卡列表及相关信息

要进行抓包，第一步工作就是要获取机器中可用的网卡列表，JpcapCaptor.getDeviceList()方法可用返回NetworkInterface 类型的数组。NetworkInterface对象包含了对应网卡的信息，对我们有用的主要是网卡的GUID值、网卡的描述信息、IP、MAC等。例如：

NetworkInterface[] devices = JpcapCaptor.*getDeviceList*();  
for (int i = 0; i < devices.length; i++) {  
 *//print out its GUID information and description* System.*out*.println(i+": "+devices[i].name + devices[i].description);  
 *//print out its MAC address* String mac = "";  
for (byte b : devices[i].mac\_address) {  
 *//mac地址6段，每段是8位，所以只保留低 8位，和0xff相与* mac = mac + Integer.*toHexString*(b & 0xff) + ":";  
}  
System.*out*.println("MAC address:" + mac.substring(0, mac.length() - 1));

*//print out its IP address, subnet mask and broadcast address* for (NetworkInterfaceAddress addr : devices[i].addresses) {  
 System.*out*.println(" address:"+addr.address + " " + addr.subnet + " "+ addr.broadcast )

}  
}

### 2. 打开网卡接口

获得网卡接口列表后，打开一个可用的网卡返回JpcapCaptor对象（该对象实例你可以近似看作是个网卡的抽象实现），用于捕获数据。例如：

JpcapCaptor jpcapCaptor = JpcapCaptor.*openDevice*(devices[0], 1514, true, 20);

* 第一个参数表示NetworkInterface对象；
* 第二个参数表示捕获的数据字节数，一般最小不小于68，避免连首部数据都抓不全，一般可设置为1514（以太网帧MTU=1500）就够用了，特殊情况还可设置更大的值（巨型帧的情况）；
* 第三个参数表示是否开启网卡的混杂模式；
* 第四个参数是毫秒为单位的超时设置；

### 3. 数据抓包

获得JpcapCaptor的对象实例jpcapCaptor后，就可以用来进行抓包操作。

**（1）单步抓包方式：**

Packet packet = jpcapCaptor.getPacket();  
System.*out*.println(packet);

jpcapCaptor.close();

每执行一次，捕获一个数据包。

**（2）自动回调方式：**

首先需要实现PacketReceiver接口：

class PacketHandler implements PacketReceiver {  
 @Override  
 public void receivePacket(Packet packet) {

System.*out*.println(packet); *//输出抓取的包的原始信息*

*//其它相关处理代码*

……  
 }  
}

然后将其对象实例作为参数传递给jpcapCaptor.loopPacket(int n, PacketReceiver packetHandler)

或 jpcapCaptor.processPacket(int n, PacketReceiver packetHandler)，

这样就会在有数据包被捕获的时候，自动执行receivePacket ()方法中的代码。

以上两个方法中的参数 int n，表示需要捕获的数据包的个数，-1表示无限制地捕获。注意，打开网络接口的方法openDevice中的超时设置，只对processPacket方法有效。loopPacket和procesPacket方法很相似，的区别主要是：前者是阻塞方法，没有包被捕获就一直阻塞；而后者是非阻塞方法，支持超时设置。

**（3）数据包信息的显示**

捕获的数据包为Packet或其子类型（TCPPacket、UDPPacket等），直接System.*out*.println(packet); 就可以输出抓包的原始信息，显示结果类似：

*1603870741:133888 /192.168.1.106->/14.215.177.38 protocol(6) priority(0) hop(128) offset(0) ident(9507) TCP 2913 > 443 seq(1890118981) win(1024) ack 3804762716 S*

以上信息其实就是IP数据报及TCP报头的一些相关信息，例如*S* 表示SYN=1的标识，*seq(1890118981)*中的数字表示序号，*ack 3804762716*中的数字表示确认号。

如果要显示抓包内的数据部分，可以通过packet.data返回字节数组内容，例如这样构造一个字符串来显示：

new String(packet.data,0,packet.data.length,"utf-8");

（当然，数据部分可能根本不是明文字符串，例如是http返回的gzip压缩格式，显示出来的会是一堆不可读的乱字符）

**（3）数据包过滤**

捕获结果内容过多，如何只显示感兴趣的内容？可以使用jpacpCaptor.setFilter()方法，例如，只捕获TCP/IPv4的数据包：

jpcapCaptor.setFilter("ip and tcp", true);

(更多的过滤规则可参考[Designing Capture Filters for Ethereal / Wireshark](https://cosy.univ-reims.fr/~lsteffenel/cours/Master1/Reseaux/0910/captureFilters.htm))

## 四、程序设计第一步：创建网卡参数配置对话框

抓包之前需要获取网卡列表，通过传入一系列参数返回JpcapCaptor对象后才能开始抓包，由于抓包是大量尝试性性工作，需要经常修改参数、设置不同过滤内容、在不同机器上切换不同网卡等，每次都到程序中手动更改低效繁琐，所以可以创建一个网卡参数配置对话框，方便使用。该对话框作为主程序窗体的子窗体使用，由主窗体唤出，这个对话框代码可以在主窗体程序中编写，但过于耦合，不利于重用。我们单独创建一个类，在这个类中实现一个Stage，供主程序窗体使用，参考界面见10.1。

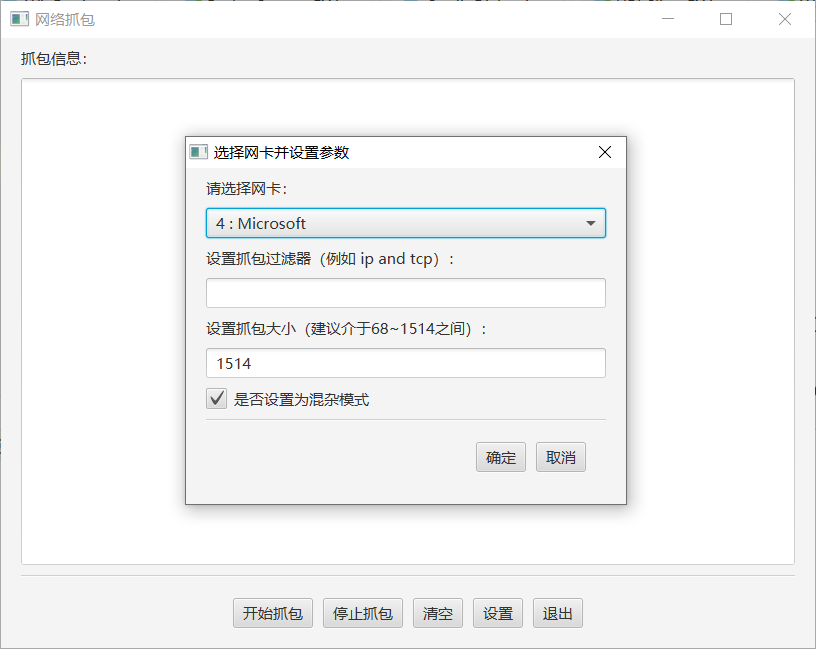


图10.1 网卡参数配置对话框

### 1.创建ConfigDialog类

在项目中新建chapter10包，在该包下新建一个类：ConfigDialog，注意，是采用创建普通类的方式，而不是以往的创建JavaFXApplication的方式。

### 2. ConofigDialog参考代码

public class ConfigDialog {  
 private JpcapCaptor jpcapCaptor;*//用于返回给主窗体*

*//网卡列表* private NetworkInterface[] devices = JpcapCaptor.*getDeviceList*();  
 private Stage stage = new Stage();*//对话框窗体*

*//parentStage表示主程序的stage，传值可通过这种构造方法参数的方式*

public ConfigDialog(Stage parentStage) {  
 *//设置该对话框的父窗体为调用者的那个窗体*

stage.initOwner(parentStage);  
 *//设置为模态窗体，即不关闭就不能切换焦点* stage.initModality(Modality.*WINDOW\_MODAL*);  
 stage.setResizable(false);  
 stage.setTitle("选择网卡并设置参数");  
  
 *//窗体主容器* VBox vBox = new VBox();  
 ......

......  
 *//网卡选择列表，使用组合下拉框控件* ComboBox<String> cob = new ComboBox<>();  
 cob.setMaxWidth(800);  
 for (int i = 0; i < devices.length; i++) {  
 cob.getItems().add( i + " : " + devices[i].description);  
 }  
 *//默认选择第一项* cob.getSelectionModel().selectFirst();  
  
 *//设置抓包过滤* TextField tfFilter = new TextField();  
  
 *//设置抓包大小（一般建议在68-1514之间，默认1514）* TextField tfSize = new TextField("1514");  
  
 *//是否设置混杂模式* CheckBox cb = new CheckBox("是否设置为混杂模式");  
 cb.setSelected(true); *//默认选中*

*//底部确定和取消按钮* HBox hBoxBottom = new HBox();  
 ......

......

Button btnConfirm = new Button("确定");  
 Button btnCancel = new Button("取消");  
 hBoxBottom.getChildren().addAll(btnConfirm,btnCancel);  
  
 *//将各组件添加到主容器* vBox.getChildren().addAll(new Label("请选择网卡："),cob,  
 new Label("设置抓包过滤器（例如 ip and tcp）："),tfFilter,  
 new Label("设置抓包大小（建议介于68~1514之间）："),tfSize,cb,  
 new Separator(),hBoxBottom);  
  
 Scene scene = new Scene(vBox);  
 stage.setScene(scene);  
 *//stage.show(); //不要显示对话框，由主窗体调用显示  
  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**   
 *//确定按钮动作事件* btnConfirm.setOnAction(event -> {  
 try {  
 int index = cob.getSelectionModel().getSelectedIndex();  
 *//选择的网卡接口* NetworkInterface networkInterface = devices[index];  
 *//抓包大小* int snapLen = Integer.*parseInt*(tfSize.getText().trim());  
 *//是否混杂模式* boolean promisc = cb.isSelected();  
 jpcapCaptor = JpcapCaptor.*openDevice*(networkInterface,snapLen,  
 promisc,20);  
 jpcapCaptor.setFilter(tfFilter.getText().trim(),true);  
 stage.hide();  
  
 } catch (Exception e) {  
 new Alert(Alert.AlertType.*ERROR*,e.getMessage()).showAndWait();  
 }  
 });  
  
 *//取消按钮动作事件* btnCancel.setOnAction(event -> {  
 stage.hide();  
 });  
 }

*//主程序调用，获取设置了参数的JpcapCaptor对象*  
 public JpcapCaptor getJpcapCaptor() {  
 return jpcapCaptor;  
 }

*//主程序调用，阻塞式显示界面* public void showAndWait() {  
 stage.showAndWait();  
 }  
}

## 五、程序设计第二步：创建网络抓包主程序

### 1. 创建PacketCaptureFX类

在chapter10包中，新建网络抓包的JavaFX主程序，命名为PacketCaptureFX.java，参考界面如图10.2所示。

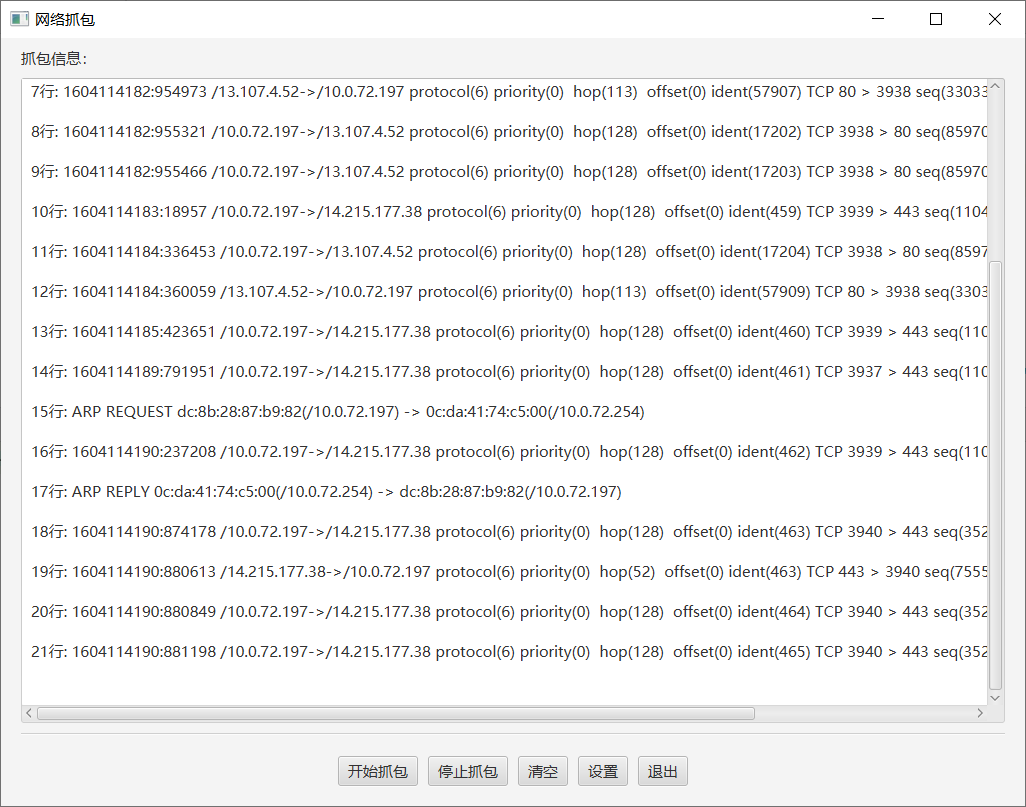


图10.2 网络抓包主程序界面

### 2. 设置各个按钮的动作响应事件

（1）“设置”按钮动作事件响应：

在该动作事件响应中，实例化ConfigDialog（如果还没实例化），显示设置对话框，并获取该对话框创建的jpcapCaptor对象，部分代码如下（自行完成省略的部分）：

*//设置按钮动作事件*btnOpenConfigDialog.setOnAction(event -> {  
 *//还没有实例化对话框，则先实例化* if (configDialog == null) {  
 ......  
 }  
 *//阻塞式显示，等待设置窗体完成设置* ......

*//获取设置后的JpcapCaptor对象实例* jpcapCaptor = configDialog.getJpcapCaptor();  
});

（2）“开始抓包”按钮动作事件响应：

* 如果还没有jpcapCaptor对象，则先调用“设置”按钮中的代码；
* 如果已经存在抓包的线程，则先停止该线程；
* 开启一个命名为"captureThread"的新线程来进行抓包的操作；

如何停止指定的线程？我们可以新建一个interrupt方法，对于需要被停止的线程，在创建时指定名称（不同线程名称可以相同），然后可以调用该方法，interrupt方法参考代码如下：

*/\*\*  
 \* 循环遍历指定线程名的线程列表，并声明关闭，需要关闭的线程需要在构造时指定线程名，作为参数传入  
 \* @param threadName  
 \*/*private void interrupt(String threadName) {  
 ThreadGroup currentGroup =  
 Thread.*currentThread*().getThreadGroup();  
 int noThreads = currentGroup.activeCount();  
 Thread[] lstThreads = new Thread[noThreads];  
 currentGroup.enumerate(lstThreads);*//将活跃线程复制到线程数组* for (int i = 0; i < noThreads; i++) {  
 if (lstThreads[i].getName().equals(threadName)) {  
 lstThreads[i].interrupt();*//声明线程关闭* }  
 }  
}

以上方法只是声明线程进行关闭，并不是真正关闭了线程，实质只是设置了一个isInterrupt逻辑变量为真，在线程中的run方法中，就可以根据这个变量来完成结束线程执行的操作。

“开始抓包”按钮事件响应参考代码如下：

*// 开始抓包 按钮动作事件*btnStart.setOnAction(event -> {  
 *//还没有jpcapCaptor对象实例，则打开设置对话框* if (jpcapCaptor == null) {  
 ......  
 return;  
 }  
  
 *//停止还没结束的抓包线程* ......  
  
 *//开线程名为"captureThread"的新线程进行抓包* new Thread(() -> {  
 while (true) {  
 *//如果声明了本线程被中断，则退出循环* if(Thread.*currentThread*().isInterrupted())  
 break;  
  
 *//每次抓一个包，交给内部类PacketHandler的实例处理  
 // PacketHandler为接口PacketReceiver的实现类* jpcapCaptor.processPacket(1, new PacketHandler());  
 }  
 },"captureThread").start();  
});

PacketHandler类的部分参考代码如下：

class PacketHandler implements PacketReceiver {  
 @Override  
 public void receivePacket(Packet packet) {  
 Platform.*runLater*(()->{  
 *// 在显示区显示抓包原始信息*

......

......   
 });  
 }  
}

（3）停止抓包 按钮事件响应：其实就是停止抓包的线程。

（4）清空按钮事件响应：将显示区的内容清空。

（5）退出按钮事件响应：完成退出功能。

程序编写完成后，可以按照过滤语法设置过滤器，看看抓包的效果。

## 六、课堂计分

（1）在自己的机器上，先启动第8讲的URLClientFX程序；抓包程序设置合理的过滤器后开始抓包，例如综合使用 port、src host、dst host等过滤词，过滤语法见Jpcap基本用法部分提供的链接；

（2）用URLClientFX程序访问https://www.gdufs.edu.cn，在抓取结果中按顺序找出建立TCP连接的这3个TCP连接包（即三次握手过程）。注意，第（1）步骤中设置合理的过滤器，就是为了过滤掉无关的抓包，方便找到这3个TCP包；

（3）将捕获的这3个包的信息按顺序和自己的学号姓名等信息构成一个UDP包。信息共四行，第一行是自己的学号&姓名，其余三行是抓包的原始信息，注意每行之间插入一个\n换行符，将构建的UDP包发送到**202.116.195.71:6868**服务器。（**注意：**TextField是单行文本框，不能识别\n等控制符，而且需要拼接的字符串很长，所以发送信息的时候，不要使用读取文本框的方式，而是直接使用StringBuffer类来进行拼接，再将其包含的字符串发送出去）

## 七、扩展练习一

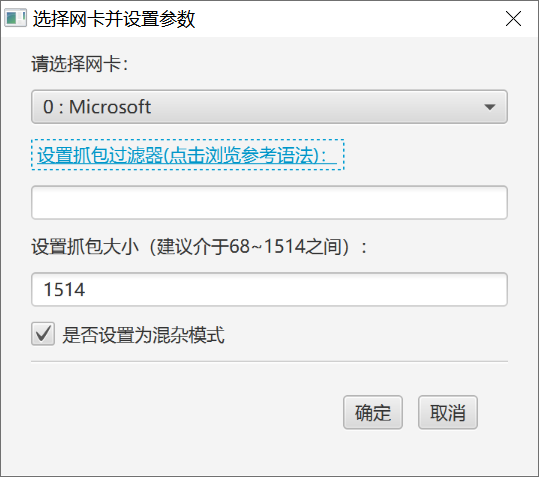
Jpcap过滤器的语法与Wireshark使用的语法几乎一致，最好能在设置界面提供一个超链接，需要的时候可以链接到语法参考界面，如图10.3所示。

图10.3 为过滤器语法参考提供超链接

实现这个功能只需要将之前用来显示提示信息的Label，换成Hyperlink控件，然后实现该控件的动作事件响应，就可以响应用户点击操作。

这个超链接控件并不提供直接打开浏览器浏览URL地址的功能，如果希望用系统默认的浏览器打开指定地址，可以使用：

Desktop.getDesktop().browse(new URI(url));

过滤器语法页面的url地址为：

<https://cosy.univ-reims.fr/~lsteffenel/cours/Master1/Reseaux/0910/captureFilters.htm>

## 八、扩展练习二

在选择网卡界面中，可能不只一个网卡，例如有线网卡、无线网卡，如果有蓝牙设备、安装了Vmware、VirtualBox等配置的虚拟网卡，那么网卡列表就更长。一般大家都知道自己使用的是哪个网卡，但部分电脑从下拉表中却不容易找出自己使用的那块网卡，例如这台笔记本的网卡列表（图10.4）：

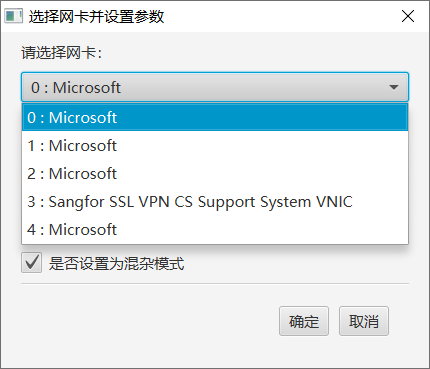


图10.4 不易区分的网卡下拉列表

这个和网卡生产商有关，有些没有这个问题，有些却显示一些同名的名称。在windows系统中，Jpcap的NetworkInerface的description方法得到的是注册表中设备的ProviderName信息，而更准确的信息是注册表中的DriverDesc信息，有些设备这两个信息一致，所以显示正常，但不少设备的ProviderName都是Microsoft，就无从区分网卡设备，如图10.5的注册表截图：

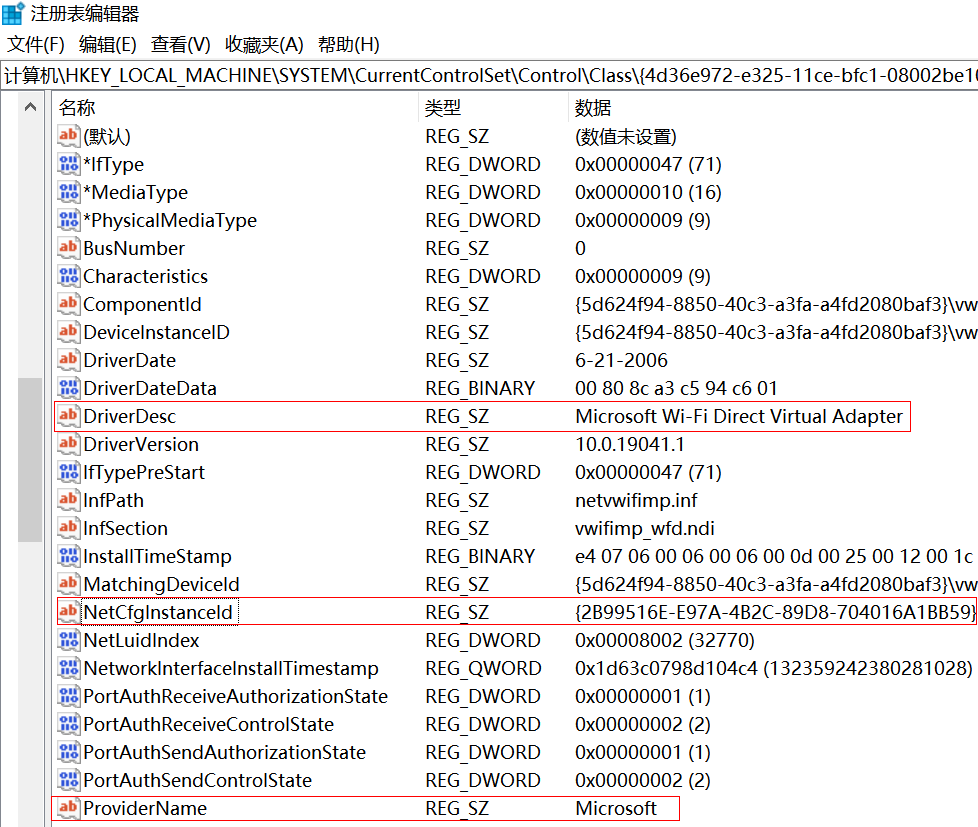


图10.5 网卡信息的注册表截图

为了解决windows系统下的这个问题，可以从注册表入手，NetworkInterface类的name方法中，包含一个guid值，就是图10.5中的NetCfgInstanceId的值，可以通过搜索注册表特定位置下子目录中的该值，匹配成功后再获取对应的DriverDesc值，就可以得到真实的设备信息，例如程序优化后网卡下拉列表就是如图10.6所示（如果DriverDesc和 ProvideName不同，则将ProviderName放在后面小括号中），可以发现列表中第2个设备才是可用的网卡。

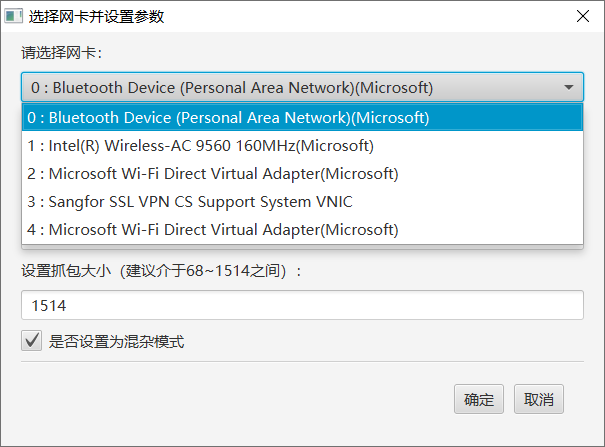


图10.6 优化后的网卡信息下拉表

解决方案大致如下：

（1）通过Jpcap的JpcapCaptor.getDeviceList()获得NetworkInterface数组，然后遍历，通过其name方法，分离出GUID值（GUID是用花括号包围住的，这个可以作为切分标识）；

（2）使用合适的操作注册表的类库，在注册表特定位置开始搜索NetCfgInstanceId的值，匹配上一步的GUID值;

注册表实际就像有五个根节点的树，关键就是要知道在注册表哪个位置开始搜索NetCfgInstanceId键对应的值，毕竟注册表太大，从最顶层根节点开始查找太慢。建议从这个位置开始搜索：

HKEY\_LOCAL\_MACHINE \SYSTEM\CurrentControlSet\Control\Class

（3）找到对应的位置，再获取同一个目录下的DriverDesc的值。

大家可以自行搜索操作注册表的工具类库，了解相应用法。这里也给大家提供一个简单的纯Java编写的注册表操作类[WinRegistry.java](https://oc.our-web.org/download?file=WinRegistry.java)，可自行下载研究。如果使用WinRegistry类，这里给出一些核心使用方法：

int *DRIVER\_CLASS\_ROOT* = WinRegistry.*HKEY\_LOCAL\_MACHINE*;  
String *DRIVER\_CLASS\_PATH* = "SYSTEM\\CurrentControlSet\\Control\\Class";  
String *NETCFG\_INSTANCE\_KEY* = "NetCfgInstanceId";*//这个值就是JPcap返回的device.name包含的guid值*

*... ...*

*... ...*

*////在注册表中循环遍历搜索对应的guid值*

for (String driverClassSubkey : WinRegistry.*readStringSubKeys*(*DRIVER\_CLASS\_ROOT*, *DRIVER\_CLASS\_PATH*, 0)) {  
 for (String driverSubkey : WinRegistry.*readStringSubKeys*(*DRIVER\_CLASS\_ROOT*, *DRIVER\_CLASS\_PATH* + "\\" + driverClassSubkey, 0)) {  
 String path = *DRIVER\_CLASS\_PATH* + "\\" + driverClassSubkey + "\\" + driverSubkey;  
 String netCfgInstanceId = WinRegistry.*readString*(*DRIVER\_CLASS\_ROOT*, path, *NETCFG\_INSTANCE\_KEY*, 0);  
  
 if (netCfgInstanceId != null && netCfgInstanceId.equalsIgnoreCase(guid)) {  
 theDriverName = *trimOrDefault*(WinRegistry.*readString*(*DRIVER\_CLASS\_ROOT*, path, "DriverDesc", 0), "");  
 *... ...* break;  
 }  
 }  
}

## 附录 TCP连接的三次握手示意图

