路由器原理仿真与实现报告文档

路由器原理仿真与实现报告文档

姓　名： 李嘉梁
学　号： 211302104
班　级： 计算机 211

1. 网络体系结构

1.1 网络分析与抽象

网络分析：

本次任务的核心目的是对路由器原理的仿真与实现，因此只需将网络层级自底向上约束到网络层即可，并且不妨让物理层对我们透明，所以以下分析乃至整个任务中所涉及的网络的定义域仅限于网络层和链路层；

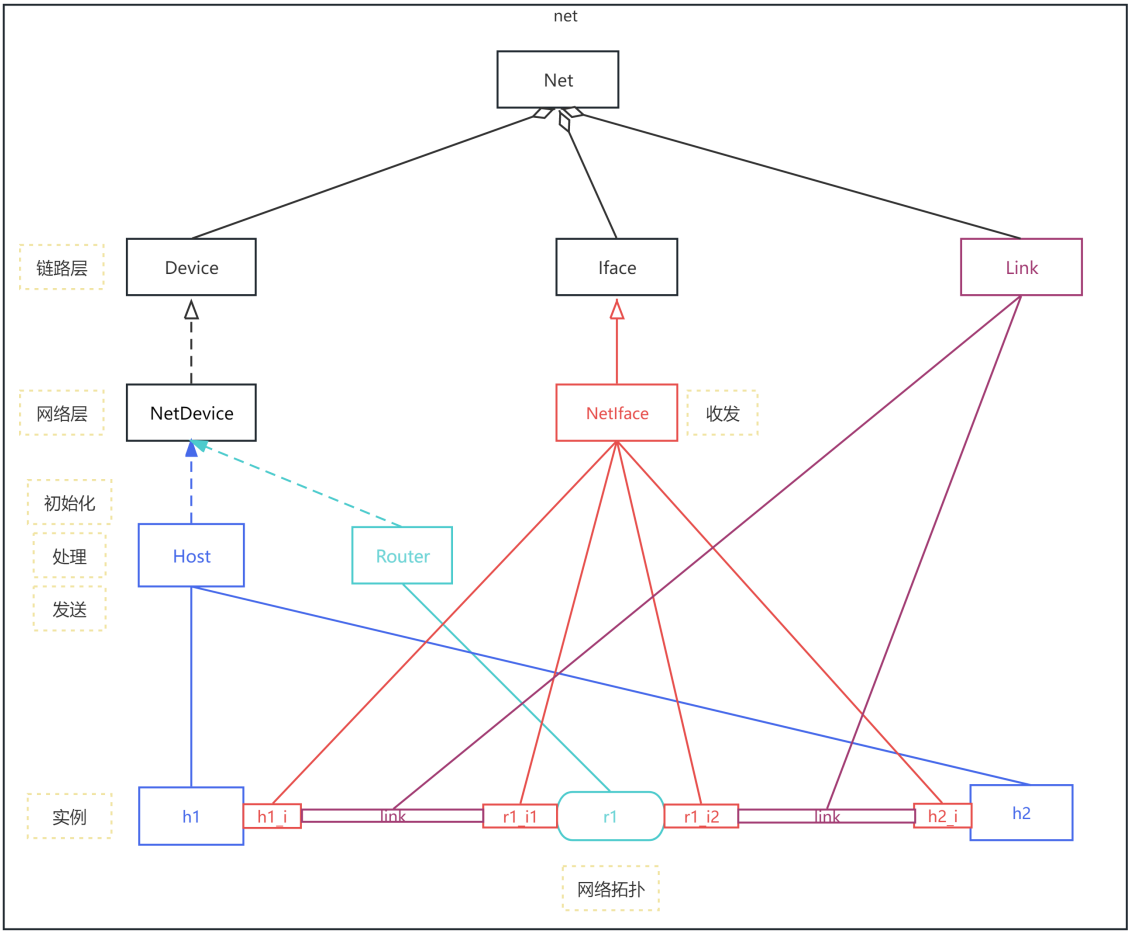
进一步分析，由于上一层建立在下一层之上并享受下一层提供的服务，可以推出某一实体的网络层抽象继承自它的链路层抽象，因此以下分析乃至整个任务中所涉及的某一层实体的语义均代表最高达到某一层的实体，如网络层实体代表的是高至网络层的实体。

网络抽象：

经过上述分析，我们可以将实体进一步泛化。

- 从链路层的视角看，主要可以分为三大类实体：设备、接口和链路；
- 从网络层的视角看，又可以分为网络层设备、网络层接口，网络层设备又可以分为路由器和主机；
    - 网络层接口备区别于链路层接口，在于它有 IP 地址、 所在子网子网掩码;
    - 网络层设备区别于链路层设备，在于它有 ARP 缓存（链路层设备显然不需要 ”IP-MAC“ 映射服务）；
        - 路由器区别于基本网络层设备在于它有路由表；
        - 主机区别于基本网络层设备在于它有网关 IP 地址（可以由 DNS 得到）；

以下是参考UML类图关系线但还没标注属性的网络抽象图，需要特殊说明的关系线是，无箭头实线代表 “类-实例” 关系：



1.2 网络设计与实现

网络设计：

经过上述抽象，我们可以将实体进一步具化。

以下是主要的类及其属性：

- 网络
  - 网络类
    - 链路集合
    - 设备集合
    - 接口集合
- 链路
  - 链路类
    - 端点接口（两个，一侧一个）
- 设备
  - 链路层接口类
    - 接口名
    - MAC 地址
    - 输入缓存
    - 输出缓存
  - 链路层设备类
    - 主机名
    - 接口集合

    - 网络层接口类
        - ...
        - IP 地址
        - 所在子网子网掩码
    - 网络层设备类
        - ...
        - ARP 缓存
    - ARP 缓存和缓存项的类
    - 路由器
        - 路由器类
            - ...
            - 路由表
        - 路由表和表项的类
    - 主机
        - 主机类
            - ...
            - 网关 IP 地址
- 数据包
    - 数据包类
        - 头部字段
        - 有效负载

网络实现：

经过上述设计，我们可以映射出对应的类。

以下是网络内核的包和类：

- `com.netapp.net`
    - 网络类（Net）
- `com.netapp.link`
    - 链路类（Link）
- `com.netapp.device`
    - 链路层接口类（Iface）
    - 链路层设备类（Device）
    - 网络层接口类（NetIface）
    - 网络层设备类（NetDevice）
    - ARP 缓存和缓存项的类（ArpCache, ArpEntry）
    - `com.netapp.device.router`
        - 路由器类（Router）
        - 路由表和表项的类（RouteTable, RouteEntry）
    - `com.netapp.device.host`
        - 主机类（Host）
- `com.netapp.packet`
    - 数据包类 （Ethernet, ARP, IPv4, ICMP, Data）

此外，还有网络应用的包和类和文件：

- `com.netapp`
  - 网络层应用启动类 （APP）
- `com.netapp.config`
  - 用于网络和设备配置的常量类 （NetConfig, DeviceConfig）
  - 用于网络和设备装配的工厂类 （NetFactory, DeviceFactory）
- `src/main/resources/config`
  - `src/main/resources/config/arp_cache`
    - ARP 缓存表
  - `src/main/resources/config/route_table`
    - 路由表
  - `src/main/resources/config/topo`
    - 网络拓扑表

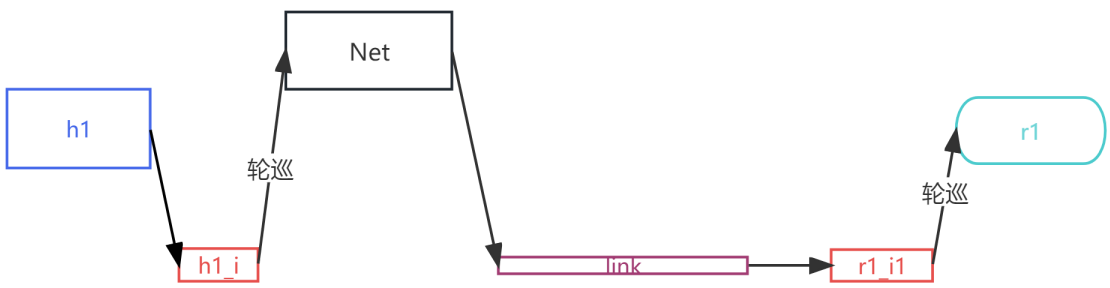
2. 网络数据流与控制流

2.1 网络数据流

一台设备如果想给另外一台设备发送数据包，至少需要经过如下步骤:

- 发送设备找到要发出的接口
- 在找到的接口中发送数据包（放入输出缓存）
- 网络轮巡中发现有接口发送数据包
- 网络查询与该接口相连的链路
- 通过以太帧头部判断是否是广播 MAC 地址
  - 如果是，则发送给每一个链路另一端的接口
  - 如果不是，则在相连链路中迭代判断是否是网络层接口
    - 如果是，则判断该接口的 MAC 地址是否是对应地址
      - 如果是，则将数据包放入其输入缓存
    - 如果不是，则这个接口是链路层接口，这个设备是交换机，直接将数据包放入其输入缓存
- 接收设备在轮巡中发现并接收接口中的数据包

以下是网络的数据流图，示例是h1发送给r1，实线箭头代表了数据包的流向：



以下是`Net.java`中`run(...)`函数的网络轮巡代码：

```
while(true){
            // 轮巡 interfaces
            for (Iface iface : interfaces.values()) {
                if(iface.peekOutputPacket() != null){
                    Ethernet etherPacket = iface.pollOutputPacket();
                    // 按 links 放入对应 inputQueue
                    List<Iface> lkdIfaces = findLkdIfaces(iface);
                    if(!lkdIfaces.isEmpty()){
                        if(etherPacket.isBroadcast()){ // 是广播以太网数据包
                            // 广播相连接口
                            for (Iface lkdIface : lkdIfaces){
                                lkdIface.putInputPacket(etherPacket);
                            }
                        }else{
                            // 查找 MAC 匹配的接口
                            for (Iface lkdIface : lkdIfaces){
                                if( lkdIface instanceof NetIface ){
                                    // lkdIface 是网络层设备，需辨别 MAC

 if(lkdIface.getMacAddress().equals(etherPacket.getDestinationMAC())){
                                        lkdIface.putInputPacket(etherPacket);
                                    }
                                    else {
                                        System.out.println(
                                                "``````````````````Net: " +
                                                lkdIface.getMacAddress() +
                                                " != " +
                                                etherPacket.getDestinationMAC()
                                        );
                                    }
                                }
                                else{
                                    // lkdIface 是链路层交换机，不需辨别 MAC
                                    lkdIface.putInputPacket(etherPacket);
                                }
                            }
                        }
                    }
                }
            }
        }
```

2.2 链路层接口控制流

接口作为发送、接收缓存并不具有控制主动性，主要为设备和网络提供通信服务，提供输入、输出的接口方法。

以下是`Iface.java`主要的成员变量和方法：

```
    protected String iName;             // 接口名称
    protected String macAddress;  // MAC地址
    protected BlockingQueue<Ethernet> inputQueue; // 输入队列
    protected BlockingQueue<Ethernet> outputQueue; // 输出队列
    public void putInputPacket(Ethernet etherPacket);// 由 Net 放入数据包
    public void putOutputPacket(Ethernet etherPacket);// 由 Device 放入数据包
    public Ethernet peekInputPacket();// 由 Device 查看数据包
    public Ethernet peekOutputPacket();// 由 Net 查看数据包
    public Ethernet pollInputPacket();// 由 Device 取出数据包
    public Ethernet pollOutputPacket();// 由 Net 取出数据包
```

2.3 网络层接口控制流

同链路层接口控制流，只是额外有 IP 地址和所在子网子网掩码。

以下是`NetIface.java`主要的成员变量和方法：

```
    protected String ipAddress;   // IP地址
    protected String subnetMask;    // 所在子网子网掩码
```

2.4 链路层设备控制流

链路层设备有自己的接口集合，当链路层设备接收到数据包时（即当`run(...)`接口轮巡时发现接口的接收缓存中有数据包时），将调用`handlePacket(...)`函数；当想要发送数据包时，调用`sendPacket(...)`函数。其中，`handlePacket(...)`是抽象方法，因为处理逻辑因设备而异。

以下是`NetIface.java`主要的成员变量和方法：

```
    /** 设备的主机名 */
    protected String hostname;

    /** 设备的接口列表；将接口名称映射到接口对象 */
    protected Map<String, Iface> interfaces;

    /**
     * 发送以太网数据包到特定接口。
     * @param etherPacket 包含所有字段、封装头和有效载荷的以太网数据包
     * @param iface 要发送数据包的接口
     * @return 如果成功发送数据包，则为 true；否则为 false
     */
    public void sendPacket(Ethernet etherPacket, Iface iface)
    { iface.putOutputPacket(etherPacket); }

    /**
     * 处理接收到的以太网数据包的抽象方法。
     * @param etherPacket 接收到的以太网数据包
     * @param inIface 接收数据包的接口
     */
    public abstract void handlePacket(Ethernet etherPacket, Iface inIface);
```

```
    @Override
    public void run() {
        while (true) {
            interfaces.forEach((iName, iface) -> {
                if(iface.peekInputPacket() != null){
                    Ethernet etherPacket = iface.pollInputPacket();
                    handlePacket(etherPacket, iface);
                }
            });
        }

    }
```

2.5 网络层设备控制流

网络层设备在链路层设备之上，拥有 ARP 缓存，可以在初始化时加载 ARP 缓存：

```
    /** ARP 缓存 */
    protected AtomicReference<ArpCache> atomicCache;

    /** 为ARP设置的输出缓存区 （不知道目的 MAC 的目的 IP） --> （对应数据包队列） */
    protected HashMap<String, BlockingQueue<Ethernet>> outputQueueMap;

    /**
     * 创建设备。
     * @param hostname 设备的主机名
     * @param interfaces 接口映射
     */
    public NetDevice(String hostname, Map<String, Iface> interfaces)
    {
        super(hostname, interfaces);
        this.atomicCache = new AtomicReference<>(new ArpCache());
        this.outputQueueMap = new HashMap<>();
        this.loadArpCache(ARP_CACHE_PREFFIX + this.hostname + ARP_CACHE_SUFFIX);
    }
```

网络层设备也可以以网络层设备的方式处理以太帧（通过调用 `handlePacket(...)` 函数），其主要逻辑是：

- 检验目的 MAC
  - 如果既不是接收接口的 MAC 也不是广播地址，则弃帧
- 检验以太帧头部校验和
  - 如果检验不正确，则弃帧
  - 如果检验正确则通过头部类型号进一步判断怎么处理，在这里分为两种情况：
    - 如果类型是 IPv4，则调用 `handleIPPacket(...)` 函数处理
    - 如果类型是 ARP，则调用 `handleARPPacket(...)` 函数处理

```
    /**
     * 处理在特定接口接收到的以太网数据包。
```

```
     * @param etherPacket 接收到的以太网数据包
     * @param inIface 接收数据包的接口
     */
    public void handlePacket(Ethernet etherPacket, Iface inIface) {
        System.out.println(this.hostname + " is receiving Ether packet: " +
etherPacket.toString());

        /********************************************************************/

        /* 检验 MAC                                                         */
        if(!inIface.getMacAddress().equals(etherPacket.getDestinationMAC()) &&
                !etherPacket.isBroadcast()){
            return;
        }

        /* 检验校验和                                                       */
        int origCksum = etherPacket.getChecksum();
        etherPacket.updateChecksum();
        int calcCksum = etherPacket.getChecksum();
        if (origCksum != calcCksum) {
            System.out.println(this.hostname + " found Ether packet's checksum
is wrong: ");
            return;
        }
        /* 处理数据包                                                       */
        switch (etherPacket.getEtherType()) {
            case Ethernet.TYPE_IPv4:
                this.handleIPPacket(etherPacket, inIface);
                break;
            case Ethernet.TYPE_ARP:
                this.handleARPPacket(etherPacket, inIface);
                break;
            // 暂时忽略其他数据包类型
        }

        /********************************************************************/
    }
```

网络层设备也可以以一致的方式处理 ARP 数据包（通过调用 `handleARPPacket(...)` 函数），其主要逻辑是：

- 获取以太帧的有效负载后，判断ARP的头部操作码
  - 如果是 ARP 请求
    - 判断如果目的 IP 不是接收接口的 IP 则丢包
    - 创建 ARP 数据包并设置源 MAC 和 IP 是自己的，目的 MAC 和 IP 是请求者的
    - 封装 ARP 作为以太帧的有效负载并发送
  - 如果是 ARP 响应
    - 缓存发送者 “IP-MAC” 映射
    - 发送设备中发送缓存的所有以此 IP为目的地址的数据包

```
/**
     * 处理 ARP 数据包。
     * @param etherPacket 接收到的以太网数据包
```

```
     * @param inIface 接收数据包的接口
     */
    protected void handleARPPacket(Ethernet etherPacket, Iface inIface) {
        if (etherPacket.getEtherType() != Ethernet.TYPE_ARP) {
            return;
        }

        ARP arpPacket = (ARP) etherPacket.getPayload();
        System.out.println(this.hostname + " is handling ARP packet: " +
arpPacket);

        if (arpPacket.getOpCode() != ARP.OP_REQUEST) {
            if (arpPacket.getOpCode() == ARP.OP_REPLY) { // 收到的是 ARP 响应数据包

                // 放入 ARP 缓存
                String srcIp = arpPacket.getSenderProtocolAddress();
                atomicCache.get().insert(srcIp,
arpPacket.getSenderHardwareAddress());

                Queue<Ethernet> packetsToSend = outputQueueMap.get(srcIp); //
outputQueueMap 中目的 IP 是响应源 IP 的数据包队列
                while(packetsToSend != null && packetsToSend.peek() != null){
                    Ethernet packet = packetsToSend.poll();

 packet.setDestinationMAC(arpPacket.getSenderHardwareAddress());
                    packet.updateChecksum();
                    this.sendPacket(packet, inIface);
                }
            }
            return;
        }

        // ARP 请求数据包

        String targetIp = arpPacket.getTargetProtocolAddress();
        if (!Objects.equals(targetIp, ((NetIface) inIface).getIpAddress())) // 不
是对应接口 IP 则不处理
            return;

        Ethernet ether = new Ethernet();
        ether.setEtherType(Ethernet.TYPE_ARP);
        ether.setSourceMAC(inIface.getMacAddress());
        ether.setDestinationMAC(etherPacket.getSourceMAC());

        ARP arp = new ARP();
        arp.setHardwareType(ARP.HW_TYPE_ETHERNET);
        arp.setProtocolType(ARP.PROTO_TYPE_IP);
        arp.setOpCode(ARP.OP_REPLY);
        arp.setSenderHardwareAddress(inIface.getMacAddress());
        arp.setSenderProtocolAddress(((NetIface)inIface).getIpAddress());
        arp.setTargetHardwareAddress(arpPacket.getSenderHardwareAddress());
        arp.setTargetProtocolAddress(arpPacket.getSenderProtocolAddress());

        ether.setPayload(arp);
```

```
        System.out.println(this.hostname + " is sending ARP packet:" + ether);

        ether.updateChecksum();
        this.sendPacket(ether, inIface);
        return;
    }
```

网络层设备也可以以一致的方式发送 ARP 数据包（通过调用 `sendARPPacket(...)` 函数），其主要逻辑是：

- 创建 ARP 数据包并设置源 MAC 和 IP 是自己的，目的 IP 是请求者的，MAC是广播 MAC
- 如果设备的输出缓存中没有以此 IP 为目的地址的缓存队列则创建，有则不用
- 将暂时不能发送的数据包放入队列
- 发送 ARP 请求
- 过一秒检查 ARP 缓存中有没有要找的条目，如果有则返回，无则返回上一个步骤，最多循环三次
- 如果三次过后还是找不到则放弃，并发送 ICMP 目的主机不可达报文告知请求主机

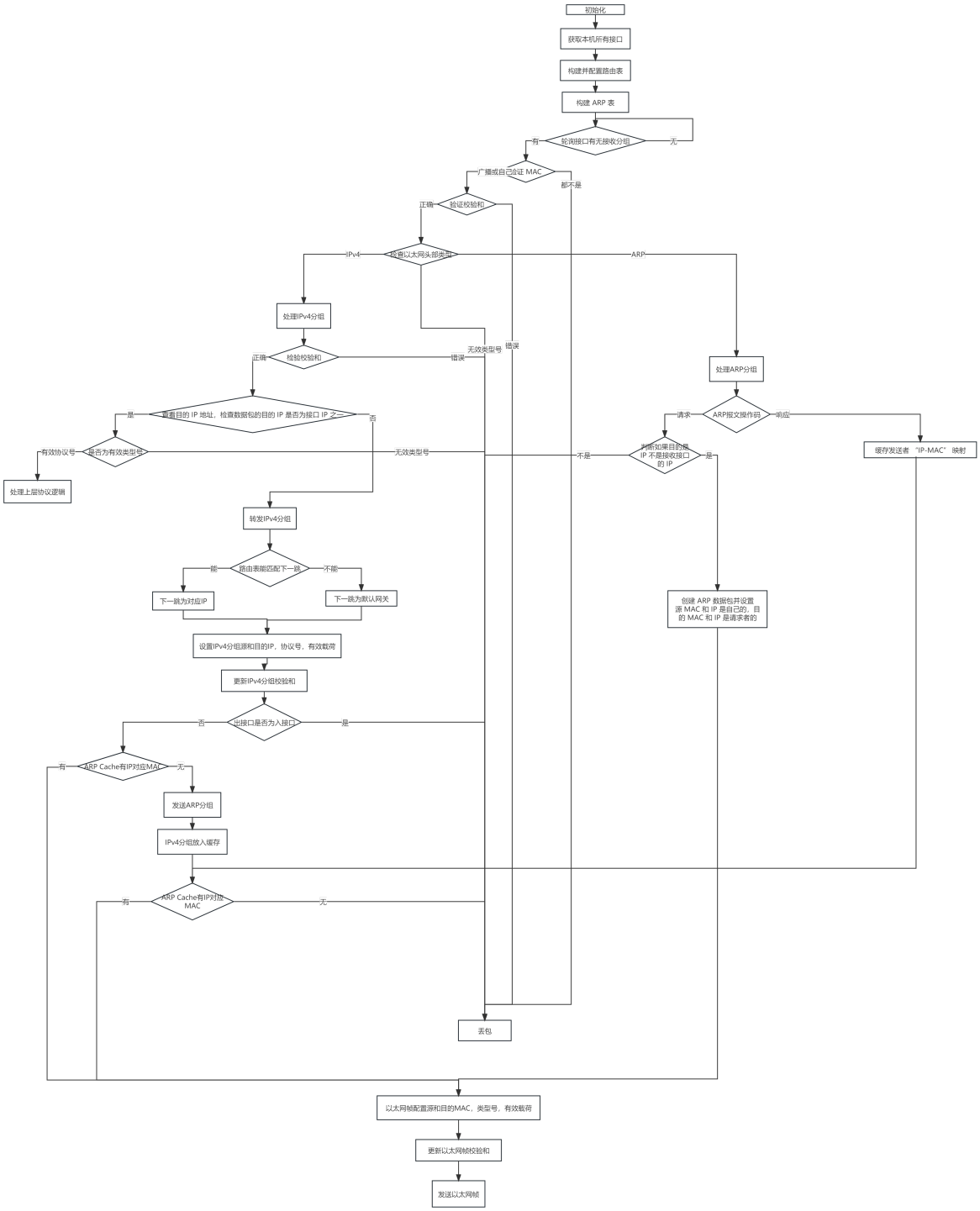
```
/**
     * 发送 ARP 数据包。
     * @param etherPacket 接收到的以太网数据包
     * @param dstIp ICMP 类型
     * @param outIface 发送数据包的接口
     */
    protected void sendARPPacket(Ethernet etherPacket, String dstIp, Iface
outIface){
        ARP arp = new ARP();
        arp.setHardwareType(ARP.HW_TYPE_ETHERNET);
        arp.setProtocolType(ARP.PROTO_TYPE_IP);
        arp.setOpCode(ARP.OP_REQUEST);
        arp.setSenderHardwareAddress(outIface.getMacAddress());
        arp.setSenderProtocolAddress(((NetIface)outIface).getIpAddress());
        arp.setTargetHardwareAddress(null);
        arp.setTargetProtocolAddress(dstIp);

        final AtomicReference<Ethernet> atomicEtherPacket = new
AtomicReference<>(new Ethernet());
        final AtomicReference<Iface> atomicIface = new AtomicReference<>
(outIface);
        final AtomicReference<Ethernet> atomicInPacket = new AtomicReference<>
(etherPacket);

        atomicEtherPacket.get().setEtherType(Ethernet.TYPE_ARP);
        atomicEtherPacket.get().setSourceMAC(outIface.getMacAddress());

        atomicEtherPacket.get().setPayload(arp);
        atomicEtherPacket.get().setDestinationMAC(Ethernet.BROADCAST_MAC); // 广
播 ARP 请求数据包
        atomicEtherPacket.get().updateChecksum();


        if (!outputQueueMap.containsKey(dstIp)) {
            outputQueueMap.put(dstIp, new LinkedBlockingQueue<>());
            System.out.println(hostname + " is making a new buffer queue for: "
+ dstIp);
        }
```

```
        BlockingQueue<Ethernet> nextHopQueue = outputQueueMap.get(dstIp);

        // 放入（不阻塞）
        try {
            nextHopQueue.put(etherPacket);
        } catch (InterruptedException e) {
//              System.out.println(this.hostname + " blocked a sending Ether
packet: " + etherPacket);
            e.printStackTrace();
        }

        final AtomicReference<BlockingQueue<Ethernet>> atomicQueue = new
AtomicReference<BlockingQueue<Ethernet>>(nextHopQueue); // 线程安全

        Thread waitForReply = new Thread(new Runnable() {

            public void run() {

                try {
                    System.out.println(hostname + " is sending ARP packet:" +
atomicEtherPacket.get());
                    sendPacket(atomicEtherPacket.get(), atomicIface.get());
                    Thread.sleep(1000);
                    if (atomicCache.get().lookup(dstIp) != null) {
                        System.out.println(hostname + ": Found it: " +
atomicCache.get().lookup(dstIp));
                        return;
                    }
                    System.out.println(hostname + " is sending ARP packet:" +
atomicEtherPacket.get());
                    sendPacket(atomicEtherPacket.get(), atomicIface.get());
                    Thread.sleep(1000);
                    if (atomicCache.get().lookup(dstIp) != null) {
                        System.out.println(hostname + ": Found it: " +
atomicCache.get().lookup(dstIp));
                        return;
                    }
                    System.out.println(hostname + " is sending ARP packet:" +
atomicEtherPacket.get());
                    sendPacket(atomicEtherPacket.get(), atomicIface.get());
                    Thread.sleep(1000);
                    if (atomicCache.get().lookup(dstIp) != null) {
                        System.out.println(hostname + ": Found it: " +
atomicCache.get().lookup(dstIp));
                        return;
                    }

                    // 都发了 3 次了，实在是真的是找不着 MAC，那就放弃吧，发送一个`目的主机
不可达`的 ICMP
                    System.out.println(hostname + ": Not found: " + dstIp);

                    while (atomicQueue.get() != null && atomicQueue.get().peek()
!= null) {
                        atomicQueue.get().poll();
                    }
```

```
                    sendICMPPacket(atomicInPacket.get(), atomicIface.get(), 3,
1, false);
                    return;
                } catch (InterruptedException e) {
                    System.out.println(e);
                }
            }
        });
        waitForReply.start();
        return;
    }
```

2.6 路由器控制流

以下是路由器的工作流程图：



以下是任务书要求的关键控制流解释：

(1) 初始化：

(1.1) 获取本机 IP 地址和 MAC 地址；

路由器通过在构造函数参数中传入它所有的接口（“接口名-接口” 映射），因为传入的是网络层接口，每个接口都有 IP 地址和 MAC 地址地址，因此路由器的 IP 地址和 MAC 地址就是它所有接口对应地址的集合。

以下是路由器（`Router.java`）的构造方法：

```
/**
     * 创建路由器。
     * @param hostname 设备的主机名
     * @param interfaces 接口映射
     */
    public Router(String hostname, Map<String, Iface> interfaces) {
        super(hostname, interfaces);
        routeTable = new RouteTable();
        this.loadRouteTable(ROUTE_TABLE_PREFFIX + this.hostname +
ROUTE_TABLE_SUFFIX);
    }
```

(1.2) 构建并配置路由表；

在路由器的构造函数中，路由器会创建路由表对象并调用`loadRouteTable(...)`函数从文件中加载路由表。

以下是路由器（`Router.java`）的`loadRouteTable(...)`方法：

```
/**
     * 从文件加载路由表。
     * @param routeTableFile 包含路由表的文件名
     */
    public void loadRouteTable(String routeTableFile) {
        if (!routeTable.load(routeTableFile, this)) {
            System.err.println("Error setting up routing table from file " +
routeTableFile);
            System.exit(1);
        }

        System.out.println(this.hostname + " loaded static route table");
        System.out.println("-------------------------------------------------");
        System.out.print(this.routeTable.toString());
        System.out.println("-------------------------------------------------");
    }
```

`loadRouteTable(...)`函数会再次调用`RouteTable`中的`load(...)`从文件中加载路由表；主要方法是通过正则表达式匹配，格式为`(dstIp, gwIp, maskIp, iface)`：目的IP，网关IP，子网掩码，对应接口；然后将记录通过`insert(...)`方法插入到路由表中。

以下是路由表（`RouteTable.java`）的`load(...)`方法：

```
/**
     * 从文件中加载静态路由表填充路由表。
     * @param filename 包含静态路由表的文件的名称
     * @param router 与路由表相关联的路由器
     * @return 如果成功加载路由表则返回 true，否则返回 false
     */
    public boolean load(String filename, Router router) {
        // 打开文件
        BufferedReader reader;
        try {
```

```
            FileReader fileReader = new FileReader(filename);
            reader = new BufferedReader(fileReader);
        } catch (FileNotFoundException e) {
            System.err.println(e.toString());
            return false;
        }

        while (true) {
            // 从文件读取路由表项
            String line = null;
            try {
                line = reader.readLine();
            } catch (IOException e) {
                System.err.println(e.toString());
                try {
                    reader.close();
                } catch (IOException f) {
                }
                return false;
            }

            // 如果已经到达文件末尾，则停止
            if (null == line) {
                break;
            }

            // 解析用于路由表项的字段
            String ipPattern = "(\\d+\\.\\d+\\.\\d+\\.\\d+)";
            String ifacePattern = "([a-zA-Z0-9_]+)";
            Pattern pattern = Pattern.compile(String.format(
                    "%s\\s+%s\\s+%s\\s+%s",
                    ipPattern, ipPattern, ipPattern, ifacePattern));
            Matcher matcher = pattern.matcher(line);
            if (!matcher.matches() || matcher.groupCount() != 4) {
                System.err.println("路由表文件中存在无效条目");
                System.out.println(line);
                try {
                    reader.close();
                } catch (IOException f) {
                }
                return false;
            }

            String dstIp = matcher.group(1);
            if (null == dstIp) {
                System.err.println("加载路由表时出错，无法将 "
                        + matcher.group(1) + " 转换为有效的 IP");
                try {
                    reader.close();
                } catch (IOException f) {
                }
                return false;
            }

            String gwIp = matcher.group(2);
```

```
            String maskIp = matcher.group(3);
            if (null == maskIp) {
                System.err.println("加载路由表时出错，无法将 "
                        + matcher.group(3) + " 转换为有效的 IP");
                try {
                    reader.close();
                } catch (IOException f) {
                }
                return false;
            }

            String ifaceName = matcher.group(4).trim();
            Iface iface = router.getInterface(ifaceName);
            if (null == iface) {
                System.err.println("加载路由表时出错，无效的接口 "
                        + matcher.group(4));
                try {
                    reader.close();
                } catch (IOException f) {
                }
                return false;
            }

            // 将条目添加到路由表
            this.insert(dstIp, gwIp, maskIp, iface);
        }

        // 关闭文件
        try {
            reader.close();
        } catch (IOException f) {
        }

        return true;

    }
```

以下是路由表（`RouteTable.java`）的`insert(...)`方法：

```
    /**
     * 向路由表中插入一条条目。
     * @param dstIp 目标 IP
     * @param gwIp 网关 IP
     * @param maskIp 子网掩码
     * @param iface 通过该接口发送数据包以到达目标或网关
     */
    public void insert(String dstIp, String gwIp, String maskIp, Iface iface) {
        RouteEntry entry = new RouteEntry(dstIp, gwIp, maskIp, iface);
        synchronized (this.entries) {
            this.entries.add(entry);
        }
    }
```

以下是路由表条目（`RouteEntry.java`）的构造函数：

```
/**
     * 创建一个新的路由表条目。
     * @param destinationAddress 目标 IP 地址
     * @param gatewayAddress 网关 IP 地址
     * @param maskAddress 子网掩码
     * @param iface 通过该接口发送数据包以达到目标或网关
     */
    public RouteEntry(String destinationAddress, String gatewayAddress, String
maskAddress, Iface iface) {
        this.destinationAddress = destinationAddress;
        this.gatewayAddress = gatewayAddress;
        this.maskAddress = maskAddress;
        this.iface = iface;
    }
```

以下是路由表文件（`r1.rt`）的内容：

```
2.0.0.2 0.0.0.0 255.255.255.0 r1_i2
1.0.0.2 0.0.0.0 255.255.255.0 r1_i1
0.0.0.0 0.0.0.0 0.0.0.0 r1_i1
```

(1.3) 构建 ARP 表。

在网络层设备的构造函数中，网络层设备会创建 ARP 缓存对象并调用`loadArpCache(...)`函数从文件中加载 ARP 缓存。

以下是网络层设备（`NetDevice.java`）的构造函数

```
/**
     * 创建设备。
     * @param hostname 设备的主机名
     * @param interfaces 接口映射
     */
    public NetDevice(String hostname, Map<String, Iface> interfaces)
    {
        super(hostname, interfaces);
        this.atomicCache = new AtomicReference<>(new ArpCache());
        this.outputQueueMap = new HashMap<>();
        this.loadArpCache(ARP_CACHE_PREFFIX + this.hostname + ARP_CACHE_SUFFIX);
    }
```

以下是网络层设备（`NetDivice.java`）的`loadArpCache(...)`方法：

```
/**
     * 从文件加载 ARP 缓存。
     * @param arpCacheFile 包含 ARP 缓存的文件名
     */
    public void loadArpCache(String arpCacheFile) {
        if (!atomicCache.get().load(arpCacheFile)) {
            System.err.println("Error setting up ARP cache from file " +
arpCacheFile);
            System.exit(1);
```

```
        }

        System.out.println(this.hostname + " loaded static ARP cache");
        System.out.println("----------------------------------");
        System.out.print(this.atomicCache.get().toString());
        System.out.println("----------------------------------");
    }
```

`loadArpCache(...)` 函数会再次调用 `ArpCache` 中的 `load(...)` 从文件中加载 ARP 缓存；主要方法是通过正则表达式匹配，格式为 `(ip,mac)`：目的 IP，网关IP，子网掩码，对应接口；然后将记录通过 `insert(...)` 方法插入到 ARP 缓存中。

以下是路由表（`ArpCache.java`）的`load(...)`方法：

```
/**
     * 从文件中加载ARP缓存。
     * @param filename 包含ARP缓存的文件的名称
     * @return 如果ARP缓存成功加载，则返回true；否则返回false
     */
    public boolean load(String filename)
    {
        // 打开文件
        BufferedReader reader;
        try
        {
            FileReader fileReader = new FileReader(filename);
            reader = new BufferedReader(fileReader);
        }
        catch (FileNotFoundException e)
        {
            System.err.println(e.toString());
            return false;
        }

        while (true)
        {
            // 从文件中读取ARP条目
            String line = null;
            try
            { line = reader.readLine(); }
            catch (IOException e)
            {
                System.err.println(e.toString());
                try { reader.close(); } catch (IOException f) {};
                return false;
            }

            // 如果已经到达文件末尾，则停止
            if (null == line)
            { break; }

            // 解析ARP条目的字段
            String ipPattern = "(\\d+\\.\\d+\\.\\d+\\.\\d+)";
            String macByte = "[a-fA-F0-9]{2}";
            String macPattern = "("+macByte+":"+macByte+":"+macByte
```

```
                    +":"+macByte+":"+macByte+":"+macByte+")";
            Pattern pattern = Pattern.compile(String.format(
                    "%s\\s+%s", ipPattern, macPattern));
            Matcher matcher = pattern.matcher(line);
            if (!matcher.matches() || matcher.groupCount() != 2)
            {
                System.err.println("ARP缓存文件中存在无效条目");
                try { reader.close(); } catch (IOException f) {};
                return false;
            }

            String ip = matcher.group(1);
            if (null == ip)
            {
                System.err.println("加载ARP缓存时出错，无法将 "
                        + matcher.group(1) + " 转换为有效的IP");
                try { reader.close(); } catch (IOException f) {};
                return false;
            }

           String mac = null;
            try
            { mac = matcher.group(2); }
            catch(IllegalArgumentException iae)
            {
                System.err.println("加载ARP缓存时出错，无法将 "
                        + matcher.group(2) + " 转换为有效的MAC");
                try { reader.close(); } catch (IOException f) {};
                return false;
            }

            // 向ARP缓存添加条目
            this.insert(ip,mac);
        }

        // 关闭文件
        try { reader.close(); } catch (IOException f) {};

        return true;
    }
```

以下是 ARP 缓存表条目（`ArpeEntry.java`）的构造函数：

```
/**
     * 创建一个将IP地址映射到MAC地址的ARP表条目。
     * @param ip 对应于MAC地址的IP地址
     * @param mac 对应于IP地址的MAC地址
     */
    public ArpEntry(String ip, String mac)
    {
        this.ip = ip;
        this.mac = mac;
        this.timeAdded = System.currentTimeMillis();
    }
```

以下是 ARP 缓存表文件（`h1.ac`）的内容：

```
1.0.0.2 AA:BB:CC:DD:EE:FF
1.0.0.1 00:11:22:33:44:55
```

(2) 处理接收的消息：

(2.1) 接收 IPv4 类型的消息；

路由器作为网络层设备调用父类网络层设备的方法处理以太帧（通过调用 `handlePacket(...)` 函数），其主要逻辑是：（在网络层设备部分也有说明）

- 检验目的 MAC
  - 如果既不是接收接口的 MAC 也不是广播地址，则弃帧
- 检验以太帧头部校验和
  - 如果检验不正确，则弃帧
  - 如果检验正确则通过头部类型号进一步判断怎么处理，在这里分为两种情况：
    - 如果类型是 IPv4，则调用 `handleIPPacket(...)` 函数处理
    - 如果类型是 ARP，则调用 `handleARPPacket(...)` 函数处理

以下是网络层设备（`NetDevice.java`）的 `handlePacket(...)` 函数：

```
    /**
     * 处理在特定接口接收到的以太网数据包。
     * @param etherPacket 接收到的以太网数据包
     * @param inIface 接收数据包的接口
     */
    public void handlePacket(Ethernet etherPacket, Iface inIface) {
        System.out.println(this.hostname + " is receiving Ether packet: " +
etherPacket.toString());

        /********************************************************************/

        /* 检验 MAC                                                         */
        if(inIface.getMacAddress().equals(etherPacket.getDestinationMAC())){
            inIface.putInputPacket(etherPacket);
        }

        /* 检验校验和                                                       */
        int origCksum = etherPacket.getChecksum();
        etherPacket.updateChecksum();
        int calcCksum = etherPacket.getChecksum();
        if (origCksum != calcCksum) {
            System.out.println(this.hostname + " found Ether packet's checksum
is wrong: ");
            return;
        }
        /* 处理数据包                                                       */
        switch (etherPacket.getEtherType()) {
            case Ethernet.TYPE_IPv4:
                this.handleIPPacket(etherPacket, inIface);
                break;
```

```
            case Ethernet.TYPE_ARP:
                this.handleARPPacket(etherPacket, inIface);
                break;
            // 暂时忽略其他数据包类型
        }

        /********************************************************************/
    }
```

(2.2) 查看目的 MAC 地址和目的 IP 地址并做出相应操作，包括检验校验和；

检验校验和，如果检验不正确，则弃帧：

```
// 检验校验和
        int origCksum = ipPacket.getChecksum();
        ipPacket.updateChecksum();
        int calcCksum = ipPacket.getChecksum();
        if (origCksum != calcCksum) {
            System.out.println(this.hostname + " found IP packet's checksum is
wrong: ");
            return;
        }
```

查看目的 IP 地址，检查数据包的目的 IP 是否为接口 IP 之一：

- 如果是，则查看头部协议号
  - 各种协议号逻辑
  - 如果都不是，则丢包
- 如果不是，检查路由表并转发

```
// 检查数据包的目的 IP 是否为接口 IP 之一
        for (Iface iface : this.interfaces.values()) {
            if (Objects.equals(ipPacket.getDestinationIP(), ((NetIface)
iface).getIpAddress())) {
                byte protocol = ipPacket.getProtocol();
//                System.out.println("ipPacket protocol: " + protocol);
                if (protocol == IPv4.PROTOCOL_ICMP) {
                    ICMP icmp = (ICMP) ipPacket.getPayload();
                    System.out.println(this.hostname + " accepted message: " +
icmp);
                    if (icmp.getIcmpType() == 8) {
                        this.sendICMPPacket(etherPacket, inIface, 0, 0, true);
                    }
                }
                else if (protocol == IPv4.PROTOCOL_DEFAULT){
                    Data data = (Data) ipPacket.getPayload();
                    System.out.println(this.hostname + " accepted message: " +
data.getData());
                }
                return;
            }
        }
 // 检查路由表并转发
        this.forwardIPPacket(etherPacket, inIface);
```

查看目的 MAC 地址，如果既不是接收接口的 MAC 也不是广播地址，则弃帧：

```
        /* 检验 MAC                                                    */
        if(!inIface.getMacAddress().equals(etherPacket.getDestinationMAC()) &&
                !etherPacket.isBroadcast()){
            return;
        }
```

以下是完整的路由器（`Router.java`）的 `handleIPPacket(...)` 函数：

```
/**
     * 处理 IP 数据包。
     * @param etherPacket 接收到的以太网数据包
     * @param inIface 接收数据包的接口
     */
    @Override
    protected void handleIPPacket(Ethernet etherPacket, Iface inIface) {
        if (etherPacket.getEtherType() != Ethernet.TYPE_IPv4) {
            return;
        }

        IPv4 ipPacket = (IPv4) etherPacket.getPayload();
        System.out.println(this.hostname + " is handling IP packet: " +
ipPacket);

        // 检验校验和
        int origCksum = ipPacket.getChecksum();
        ipPacket.updateChecksum();
        int calcCksum = ipPacket.getChecksum();
        if (origCksum != calcCksum) {
            System.out.println(this.hostname + " found IP packet's checksum is
wrong: ");
            return;
        }

        // TTL-1
        ipPacket.setTtl((ipPacket.getTtl() - 1));
        if (0 == ipPacket.getTtl()) {
            this.sendICMPPacket(etherPacket, inIface, 11, 0, false);
            return;
        }

        // 更新校验和
        ipPacket.updateChecksum();

        // 检查数据包的目的 IP 是否为接口 IP 之一
        for (Iface iface : this.interfaces.values()) {
            if (Objects.equals(ipPacket.getDestinationIP(), ((NetIface)
iface).getIpAddress())) {
                byte protocol = ipPacket.getProtocol();
//                System.out.println("ipPacket protocol: " + protocol);
                if (protocol == IPv4.PROTOCOL_ICMP) {
                    ICMP icmp = (ICMP) ipPacket.getPayload();
```

```
                    System.out.println(this.hostname + " accepted message: " +
icmp);
                    if (icmp.getIcmpType() == 8) {
                        this.sendICMPPacket(etherPacket, inIface, 0, 0, true);
                    }
                }
                else if (protocol == IPv4.PROTOCOL_DEFAULT){
                    Data data = (Data) ipPacket.getPayload();
                    System.out.println(this.hostname + " accepted message: " +
data.getData());
                }
                return;
            }
        }

        // 检查路由表并转发
        this.forwardIPPacket(etherPacket, inIface);
    }
```

(2.3) 对于要转发的数据报，查看目的 IP 地址，根据路由表做出相应操作；

路由器通过 `RouteTable` 类中的 `lookup(...)` 函数查看目的 IP 地址的下一跳，该函数应返回与给定 IP 地址具有最长前缀匹配的 `RouteEntry` 对象。如果没有匹配的条目，则函数返回默认网关条目。路由器得到条目后判断发送接口是否等于接收接口，如果是则丢包，如果不是则进一步处理。

以下是路由表（`RouteTable.java`）的 `lookup(...)` 函数：

```
/**
     * 查找与给定 IP 地址匹配的路由条目。
     * @param ipAddress IP 地址
     * @return 匹配的路由条目，如果不存在则返回 null
     */
    public RouteEntry lookup(String ipAddress) {

        int ip = IPv4.toIPv4Address(ipAddress);

        synchronized (this.entries) {
            /*********************************************************************/
            /* 找到具有最长前缀匹配的路由条目                                    */

            // 初始化最佳匹配为null
            RouteEntry bestMatch = null;
            // 初始化默认匹配为null
            RouteEntry defaultMatch = null;

            // 遍历所有路由条目
            for (RouteEntry entry : this.entries) {

                // 暂时忽略默认网关
                if(entry.getDestinationAddress().equals(IPv4.DEFAULT_IP) &&
                        entry.getMaskAddress().equals(IPv4.DEFAULT_IP))
                {
                    defaultMatch = entry;
```

```
                    continue;
                }

                // 使用路由条目的掩码对目标IP进行掩码操作
                int maskedDst = ip &
                        IPv4.toIPv4Address(entry.getMaskAddress());
                // 获取路由条目的子网地址
                int entrySubnet =
IPv4.toIPv4Address(entry.getDestinationAddress()) &
                        IPv4.toIPv4Address(entry.getMaskAddress());

                // 如果掩码后的目标IP与子网地址匹配
                if (maskedDst == entrySubnet) {
                    // 如果当前匹配是第一个或者当前路由条目的掩码更长
                    if ((bestMatch == null) ||
                            (IPv4.toIPv4Address(entry.getMaskAddress()) >

 IPv4.toIPv4Address(bestMatch.getMaskAddress()))) {
                        bestMatch = entry; // 更新最佳匹配
                    }
                }
            }

            // 如果找不到最佳匹配，  则最佳匹配就是默认匹配
            if(bestMatch == null){
                bestMatch = defaultMatch;
                System.out.println("Can't find best match, best match set
default match: " + defaultMatch);
            }

            return bestMatch; // 返回最佳匹配的路由条目

            /************************************************************/
        }
    }
```

以下是完整的路由器（`Router.java`）的 `forwardIPPacket(...)` 函数：

```
 /**
     * 转发 IP 数据包。
     * @param etherPacket 接收到的以太网数据包
     * @param inIface 接收数据包的接口
     */
    private void forwardIPPacket(Ethernet etherPacket, Iface inIface) {
        // 确保是 IP 数据包
        if (etherPacket.getEtherType() != Ethernet.TYPE_IPv4) {
            return;
        }
        IPv4 ipPacket = (IPv4) etherPacket.getPayload();
        System.out.println(this.hostname + " is forwarding IP packet: "  +
ipPacket);

        // 获取目的 IP
        String dstIp = ipPacket.getDestinationIP();
```

```
        // 查找匹配的路由表项
        RouteEntry bestMatch = this.routeTable.lookup(dstIp);

        // 这种情况会转发到默认网关
        /*
        // 如果没有匹配的项，则发 ICMP
        if (null == bestMatch) {
            this.sendICMPPacket(etherPacket, inIface, 3, 0, false);
            return;
        }

         */

        // 确保不将数据包发送回它进入的接口
        Iface outIface =  bestMatch.getInterface();
        if (outIface == inIface) {
            return;
        }

        // 设置以太网头部中的源 MAC 地址
        String srcMac = outIface.getMacAddress();
        etherPacket.setSourceMAC(srcMac);

        // 如果没有网关，那么下一跳是 IP 目的地，设置目的 MAC 的时候可以直接设置目的地的
MAC，否则设置网关的MAC
        String nextHop = bestMatch.getGatewayAddress();
        if (IPv4.DEFAULT_IP.equals(nextHop)) {
            nextHop = dstIp;
        }

        // 设置以太网头部中的目标 MAC 地址
        ArpEntry arpEntry = this.atomicCache.get().lookup(nextHop);
        if (null == arpEntry) {

            System.out.println(this.hostname + " can't find arp entry for: " +
dstIp);
            sendARPPacket(etherPacket, nextHop, outIface);

            return;
        } else
            etherPacket.setDestinationMAC(arpEntry.getMac());

        etherPacket.updateChecksum();
        this.sendPacket(etherPacket, outIface);
    }
```

(2.4) 根据下一跳 IP 地址和 ARP 表做出相应操作，包括处理 ARP 分组。

根据查找到的路由条目判断有没有网关，如果没有网关，那么下一跳就是目的地 IP ，用 ARP 解析下一跳 MAC 的时候可以直接解析目的地的 IP ，否则需解析网关的 IP，如果 ARP 缓存中找不到对应条目，则发送 ARP 请求，否则直接将找到的 MAC 设为目的 MAC 。

处理ARP分组：通过调用 `handleARPPacket(...)` 函数，其主要逻辑是：（网络层设备部分也有说明）

- 获取以太帧的有效负载后，判断ARP的头部操作码
  - 如果是 ARP 请求
    - 判断如果目的 IP 不是接收接口的 IP 则丢包
    - 创建 ARP 数据包并设置源 MAC 和 IP 是自己的，目的 MAC 和 IP 是请求者的
    - 封装 ARP 作为以太帧的有效负载并发送
  - 如果是 ARP 响应
    - 缓存发送者 “IP-MAC” 映射
    - 发送设备中发送缓存的所有以此 IP为目的地址的数据包

```
/**
     * 处理 ARP 数据包。
     * @param etherPacket 接收到的以太网数据包
     * @param inIface 接收数据包的接口
     */
    protected void handleARPPacket(Ethernet etherPacket, Iface inIface) {
        if (etherPacket.getEtherType() != Ethernet.TYPE_ARP) {
            return;
        }

        ARP arpPacket = (ARP) etherPacket.getPayload();
        System.out.println(this.hostname + " is handling ARP packet: " +
arpPacket);

        if (arpPacket.getOpCode() != ARP.OP_REQUEST) {
            if (arpPacket.getOpCode() == ARP.OP_REPLY) { // 收到的是 ARP 响应数据包

                // 放入 ARP 缓存
                String srcIp = arpPacket.getSenderProtocolAddress();
                atomicCache.get().insert(srcIp,
arpPacket.getSenderHardwareAddress());

                Queue<Ethernet> packetsToSend = outputQueueMap.get(srcIp); //
outputQueueMap 中目的 IP 是响应源 IP 的数据包队列
                while(packetsToSend != null && packetsToSend.peek() != null){
                    Ethernet packet = packetsToSend.poll();

 packet.setDestinationMAC(arpPacket.getSenderHardwareAddress());
                    packet.updateChecksum();
                    this.sendPacket(packet, inIface);
                }
            }
            return;
        }

        // ARP 请求数据包

        String targetIp = arpPacket.getTargetProtocolAddress();
        if (!Objects.equals(targetIp, ((NetIface) inIface).getIpAddress())) // 不
是对应接口 IP 则不处理
            return;

        Ethernet ether = new Ethernet();
        ether.setEtherType(Ethernet.TYPE_ARP);
```

```
        ether.setSourceMAC(inIface.getMacAddress());
        ether.setDestinationMAC(etherPacket.getSourceMAC());

        ARP arp = new ARP();
        arp.setHardwareType(ARP.HW_TYPE_ETHERNET);
        arp.setProtocolType(ARP.PROTO_TYPE_IP);
        arp.setOpCode(ARP.OP_REPLY);
        arp.setSenderHardwareAddress(inIface.getMacAddress());
        arp.setSenderProtocolAddress(((NetIface)inIface).getIpAddress());
        arp.setTargetHardwareAddress(arpPacket.getSenderHardwareAddress());
        arp.setTargetProtocolAddress(arpPacket.getSenderProtocolAddress());

        ether.setPayload(arp);

        System.out.println(this.hostname + " is sending ARP packet:" + ether);

        ether.updateChecksum();
        this.sendPacket(ether, inIface);
        return;
    }
```

(3) 组装新帧：

(3.1) 修改 IP 数据报的首部并做出相应操作，包括发送 ICMP 分组、计算校验和；

修改 TTL， 如果为0，则发送 ICMP 超时报文：

```
// TTL-1
        ipPacket.setTtl((ipPacket.getTtl() - 1));
        if (0 == ipPacket.getTtl()) {
            this.sendICMPPacket(etherPacket, inIface, 11, 0, false);
            return;
        }
```

计算校验和：

```
// 更新校验和
        ipPacket.updateChecksum();
```

(3.2) 修改以太帧的首部。

修改源 MAC 为发送接口的 MAC：

```
 // 设置以太网头部中的源 MAC 地址
        String srcMac = outIface.getMacAddress();
        etherPacket.setSourceMAC(srcMac);
```

设置目的 MAC 为 ARP 得到的 MAC：

```
etherPacket.setDestinationMAC(arpEntry.getMac());
```

(4) 转发消息，等待接收新的消息。

更新以太帧校验和并转发以太帧：

```
etherPacket.updateChecksum();
this.sendPacket(etherPacket, outIface);
```

2.7 主机控制流

以下是任务书要求的关键控制流解释：

(1) 构造 IPv4 分组；

主机构造IPv4 分组的逻辑主要如下：

- 创建IPv4 分组
- 设置 Data 数据包为有效负载
- 设置目的 IP
- 获取并设置输出接口的 IP
- 更新 IPv4 分组校验和

```
        IPv4 ip = new IPv4();
        Data data = new Data(message);
        ip.setPayload(data);

//        int ttl = 64;
        ip.setTtl(ttl);
        ip.setDestinationIP(dstIp);

        Iface outIface = this.getDefaultInterface();

        // 在 ICMP Echo 回应中：源 IP 是上一次请求的接收方主机的 IP 地址
        ip.setSourceIP(((NetIface)outIface).getIpAddress());

        // 更新校验和
        ip.updateChecksum();
```

(2) 构造以太网帧。

主机构造以太网帧的逻辑主要如下：

- 创建以太帧
- 设置 IP 数据包为有效负载
- 修改源 MAC 为发送接口的 MAC
- 判断目的 IP 属不属于自己所属子网
    - 如果直接 ARP 解析目的地的 IP
    - 否则需解析网关的 IP
- 如果 ARP 缓存中找不到对应条目， 则发送 ARP 请求，否则直接将找到的 MAC 设为目的 MAC

- 更新以太帧校验和

```
 Ethernet ether = new Ethernet();
        ether.setPayload(ip);

        ether.setSourceMAC(outIface.getMacAddress());

        String nextHop = null;
        // 判断属不属于自己所属子网：
        if (isInSubnet(dstIp, outIface))
        { // 如果属于自己所属子网则直接设置目的 MAC 为目的地 MAC
            nextHop = dstIp;
            System.out.println(this.hostname + " found dstIp is in the subnet,
nexHop: " + nextHop);
        }
        else
        { // 否则将其设置为网关的 MAC
            nextHop = this.gatewayAddress;
            System.out.println(this.hostname + " found dstIp is not in the
subnet, nexHop: " + nextHop);
        }

        ArpEntry arpEntry = this.atomicCache.get().lookup(nextHop);
        if (null == arpEntry) {

            System.out.println(this.hostname + " can't find arp entry for: " +
nextHop);
            sendARPPacket(ether, nextHop, outIface);
            return;

        } else
            ether.setDestinationMAC(arpEntry.getMac());

        ether.updateChecksum();
```

以下是完整的主机（`Host.java`）的 `sendIPPacket(...)` 函数：

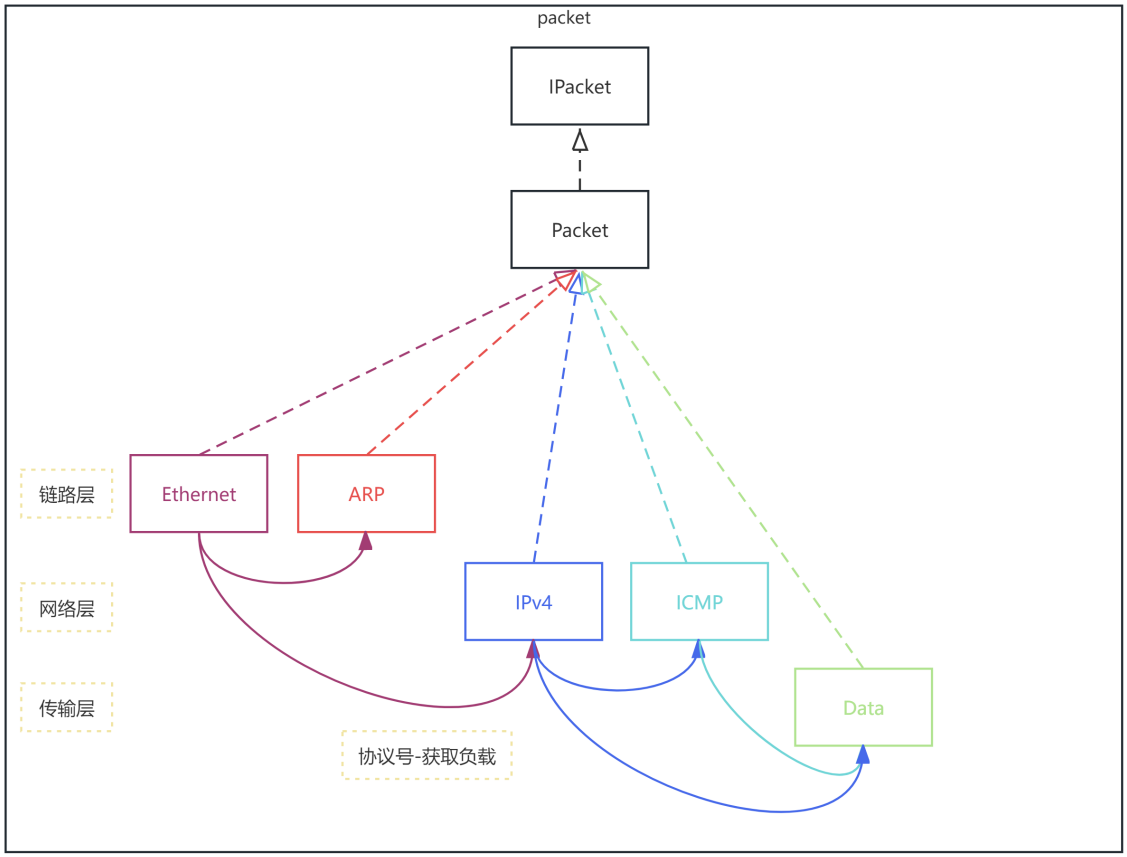
```
/**
     * 发送 IP 数据包。
     * @param message 模拟的IP数据包载荷
     */
    public void sendIPPacket(String dstIp, String message, int ttl) {
        Ethernet ether = new Ethernet();
        IPv4 ip = new IPv4();
        Data data = new Data(message);
        ether.setPayload(ip);
        ip.setPayload(data);

        ether.setEtherType(Ethernet.TYPE_IPv4);

//        int ttl = 64;
        ip.setTtl(ttl);
        ip.setDestinationIP(dstIp);
```

```
        Iface outIface = this.getDefaultInterface();

        // 在 ICMP Echo 回应中：源 IP 是上一次请求的接收方主机的 IP 地址
        ip.setSourceIP(((NetIface)outIface).getIpAddress());

        // 更新校验和
        ip.updateChecksum();

        System.out.println(this.hostname + " is sending IP packet: " + ip);

        ether.setSourceMAC(outIface.getMacAddress());

        String nextHop = null;
        // 判断属不属于自己所属子网：
        if (isInSubnet(dstIp, outIface))
        { // 如果属于自己所属子网则直接设置目的 MAC 为目的地 MAC
            nextHop = dstIp;
            System.out.println(this.hostname + " found dstIp is in the subnet,
nexHop: " + nextHop);
        }
        else
        { // 否则将其设置为网关的 MAC
            nextHop = this.gatewayAddress;
            System.out.println(this.hostname + " found dstIp is not in the
subnet, nexHop: " + nextHop);
        }

        ArpEntry arpEntry = this.atomicCache.get().lookup(nextHop);
        if (null == arpEntry) {

            System.out.println(this.hostname + " can't find arp entry for: " +
nextHop);
            sendARPPacket(ether, nextHop, outIface);
            return;

        } else
            ether.setDestinationMAC(arpEntry.getMac());

        ether.updateChecksum();
        this.sendPacket(ether, outIface);
    }
```

3. 网络协议与数据包

3.1 网络协议层级与数据包抽象

本网络所涉及的协议仅限于链路层和网络层，按网络层级划分和负载所属关系可得如下结果：



所有的数据包都继承自基本数据包，它们的共性是都具有有效负载，父级包可以通过协议号解析并通过获取负载得到子级包。

经过上述抽象，可以抽象出数据包接口和基本数据包类。

以下是数据包接口（`IPacket.java`）的方法：

```
    /**
     * 获取有效负载。
     * @return 返回实现 IPacket 接口的对象，表示有效负载。
     */
    public IPacket getPayload();

    /**
     * 设置有效负载。
     * @param packet 要设置的有效负载对象，必须实现 IPacket 接口。
     * @return 返回设置后的 IPacket 对象。
     */
    public IPacket setPayload(IPacket packet);
```

以下是基本数据包（`Packet.java`）的主要成员变量和方法：

```
    protected IPacket payload;

    /**
     * 获取有效负载。
     * @return 返回实现 IPacket 接口的对象，表示有效负载。
```

```
     */
    @Override
    public IPacket getPayload() {
        return payload;
    }

    /**
     * 设置有效负载。
     * @param payload 要设置的有效负载对象，必须实现 IPacket 接口。
     * @return 返回设置后的 IPacket 对象。
     */
    @Override
    public IPacket setPayload(IPacket payload) {
        this.payload = payload;
        return this;
    }
```

3.2 Ethernet协议与数据包

以下是Ethernet数据包（`Ethernet.java`）的头部和常量：

```
    public static final String BROADCAST_MAC = "FF:FF:FF:FF:FF:FF";
    public static final short TYPE_ARP = 0x0806;
    public static final short TYPE_IPv4 = 0x0800;
    public static Map<Short, Class<? extends IPacket>> etherTypeClassMap;

    static {
        etherTypeClassMap = new HashMap<Short, Class<? extends IPacket>>();
        etherTypeClassMap.put(TYPE_ARP, ARP.class);
        etherTypeClassMap.put(TYPE_IPv4, IPv4.class);
    }

    private String sourceMAC;
    private String destinationMAC;
    private short etherType;
    private int checksum;
```

3.3 ARP协议与数据包

以下是ARP数据包（`ARP.java`）的头部和常量：

```
    public static short HW_TYPE_ETHERNET = 0x1;
    public static short PROTO_TYPE_IP = 0x800;

    public static final short OP_REQUEST = 0x1;
    public static final short OP_REPLY = 0x2;

    protected short hardwareType;
    protected short protocolType;
    protected short opCode; // 1->REQ : 2-> REPLY
```

```
    protected String senderHardwareAddress;
    protected String senderProtocolAddress;
    protected String targetHardwareAddress;
    protected String targetProtocolAddress;
```

3.4 IPv4协议与数据包

以下是IPv4数据包（`IPv4.java`）的头部和常量：

```
    public static final String BROADCAST_IP = "255.255.255.255";
    public static final String DEFAULT_IP = "0.0.0.0";
    public static final byte PROTOCOL_DEFAULT = 0x0;
    public static final byte PROTOCOL_ICMP = 0x1;
    public static final byte PROTOCOL_TCP = 0x6;
    public static final byte PROTOCOL_UDP = 0x11;
    public static Map<Byte, Class<? extends IPacket>> protocolClassMap;

    static {
        protocolClassMap = new HashMap<Byte, Class<? extends IPacket>>();
        protocolClassMap.put(PROTOCOL_ICMP, ICMP.class);
    }

    private byte version;
    private String sourceIP;
    private String destinationIP;
    private int ttl;
    private byte protocol;
    private int checksum;
```

3.5 ICMP协议与数据包

以下是ICMP数据包（`ICMP.java`）的头部和常量：

```
    private byte icmpType;
    private byte icmpCode;
    private int checksum;

    public static final String ECHO_REQUEST = "ECHO_REQUEST";
    public static final String ECHO_REPLY = "ECHO_REPLY";
    public static final String DESTINATION_NETWORK_UNREACHABLE =
"DESTINATION_NETWORK_UNREACHABLE";
    public static final String DESTINATION_HOST_UNREACHABLE =
"DESTINATION_HOST_UNREACHABLE";
    public static final String TIME_EXCEEDED = "TIME_EXCEEDED";
```

4. 运行结果

4.1 常规运行结果

测试：h1 需要经过 r1 给 h2 发送信息，h1 和 r1 的ARP 缓存和路由表均有对应条目：

```
/**
         * h1 -> h2                @TEST: h1 -> r1 ->h2
         * */
        net.service.sendIPPacket(H1_HOSTNAME, H2_I_IP, message, 64);
```

理想结果： h2 能收到 h1 发送的信息。

测试结果：


```
    INPUT YOUR MESSAGE HERE: hi :)
/**`YOUR MESSAGE IS SENDING!`*/
h1 is sending IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='2.0.0.2', ttl=64, protocol=0, checksum=9788,
payload=Data{data='hi :)'}}
h1 found dstIp is not in the subnet, nexHop: 1.0.0.1
h1_i is sending Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18870,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=64,
protocol=0, checksum=9788, payload=Data{data='hi :)'}}}
r1_i1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18870,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=64,
protocol=0, checksum=9788, payload=Data{data='hi :)'}}}
r1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18870,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=64,
protocol=0, checksum=9788, payload=Data{data='hi :)'}}}
r1 is handling IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='2.0.0.2', ttl=64, protocol=0, checksum=9788,
payload=Data{data='hi :)'}}
r1 is forwarding IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='2.0.0.2', ttl=63, protocol=0, checksum=9787,
payload=Data{data='hi :)'}}
r1_i2 is sending Ether packet: Ethernet{sourceMAC='11:22:33:44:55:00',
destinationMAC='BB:CC:DD:EE:FF:AA', etherType=2048, checksum=18868,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=63,
protocol=0, checksum=9787, payload=Data{data='hi :)'}}}
```

```
h2_i is receiving Ether packet: Ethernet{sourceMAC='11:22:33:44:55:00',
destinationMAC='BB:CC:DD:EE:FF:AA', etherType=2048, checksum=18868,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=63,
protocol=0, checksum=9787, payload=Data{data='hi :)'}}}
h2 is receiving Ether packet: Ethernet{sourceMAC='11:22:33:44:55:00',
destinationMAC='BB:CC:DD:EE:FF:AA', etherType=2048, checksum=18868,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=63,
protocol=0, checksum=9787, payload=Data{data='hi :)'}}}
h2 is handling IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='2.0.0.2', ttl=63, protocol=0, checksum=9787,
payload=Data{data='hi :)'}}
h2 accepted message: hi :)
/**``````````````````````````````````````````````````````````````````*/
```

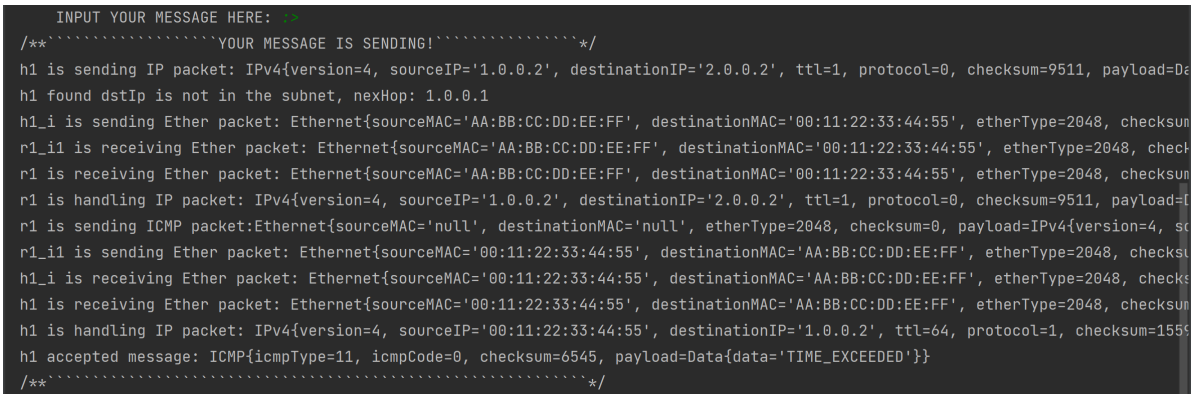
4.2 超时运行结果

测试：h1 需要经过 r1 给 h2 发送信息，h1 和 r1 的ARP 缓存和路由表均有对应条目，但是 IP 数据包的 TTL = 1：

```
/**
         *  h1 -> h2               @TEST: ICMP_DESTINATION_TIME_EXCEEDED
         *  */
        net.service.sendIPPacket(H1_HOSTNAME, H2_I_IP, message, 1);
```

理想结果： r1 给 h1 发送 ICMP 超时报文。

测试结果：




```
    INPUT YOUR MESSAGE HERE: hi :>
/**``````````````````YOUR MESSAGE IS SENDING!````````````````*/
h1 is sending IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='2.0.0.2', ttl=1, protocol=0, checksum=9752, payload=Data{data='hi
:>'}}
h1 found dstIp is not in the subnet, nexHop: 1.0.0.1
h1_i is sending Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18825,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=1,
protocol=0, checksum=9752, payload=Data{data='hi :>'}}}
```

```
r1_i1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18825,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=1,
protocol=0, checksum=9752, payload=Data{data='hi :>'}}}
r1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18825,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=1,
protocol=0, checksum=9752, payload=Data{data='hi :>'}}}
r1 is handling IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='2.0.0.2', ttl=1, protocol=0, checksum=9752, payload=Data{data='hi
:>'}}
r1 is sending ICMP packet:Ethernet{sourceMAC='null', destinationMAC='null',
etherType=2048, checksum=0, payload=IPv4{version=4,
sourceIP='00:11:22:33:44:55', destinationIP='1.0.0.2', ttl=64, protocol=1,
checksum=15595, payload=ICMP{icmpType=11, icmpCode=0, checksum=6545,
payload=Data{data='TIME_EXCEEDED'}}}}
r1_i1 is sending Ether packet: Ethernet{sourceMAC='00:11:22:33:44:55',
destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2048, checksum=24718,
payload=IPv4{version=4, sourceIP='00:11:22:33:44:55', destinationIP='1.0.0.2',
ttl=64, protocol=1, checksum=15595, payload=ICMP{icmpType=11, icmpCode=0,
checksum=6545, payload=Data{data='TIME_EXCEEDED'}}}}
h1_i is receiving Ether packet: Ethernet{sourceMAC='00:11:22:33:44:55',
destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2048, checksum=24718,
payload=IPv4{version=4, sourceIP='00:11:22:33:44:55', destinationIP='1.0.0.2',
ttl=64, protocol=1, checksum=15595, payload=ICMP{icmpType=11, icmpCode=0,
checksum=6545, payload=Data{data='TIME_EXCEEDED'}}}}
h1 is receiving Ether packet: Ethernet{sourceMAC='00:11:22:33:44:55',
destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2048, checksum=24718,
payload=IPv4{version=4, sourceIP='00:11:22:33:44:55', destinationIP='1.0.0.2',
ttl=64, protocol=1, checksum=15595, payload=ICMP{icmpType=11, icmpCode=0,
checksum=6545, payload=Data{data='TIME_EXCEEDED'}}}}
h1 is handling IP packet: IPv4{version=4, sourceIP='00:11:22:33:44:55',
destinationIP='1.0.0.2', ttl=64, protocol=1, checksum=15595,
payload=ICMP{icmpType=11, icmpCode=0, checksum=6545,
payload=Data{data='TIME_EXCEEDED'}}}
h1 accepted message: ICMP{icmpType=11, icmpCode=0, checksum=6545,
payload=Data{data='TIME_EXCEEDED'}}
/**``````````````````````````````````````````````````````````*/
```

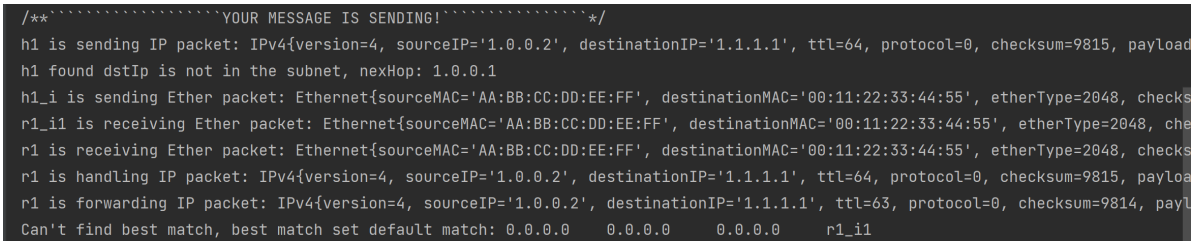
4.3 路由表项缺失运行结果

测试：h1 给子网外并且 r1 路由表中没有对应路由表项的 IP 发送信息。

```
        /**
         *  h1 -> ?                @TEST: FORWARD_DEFAULT_GATEWAY
         * */
        net.service.sendIPPacket(H1_HOSTNAME,"1.1.1.1", message, 64);
```

理想结果：r1 将数据包转发到默认网关。

测试结果：

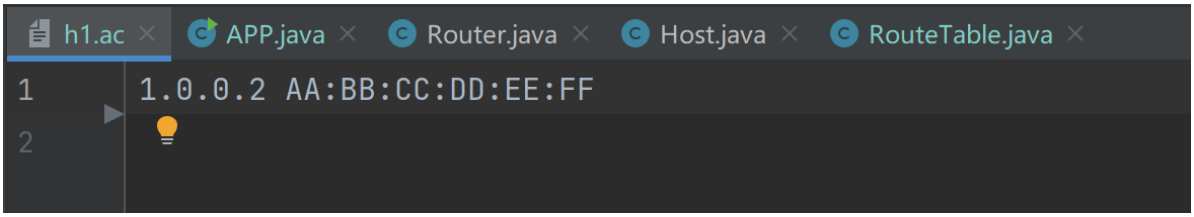


```
    INPUT YOUR MESSAGE HERE: hi :D
/**``````````````````YOUR MESSAGE IS SENDING!````````````````*/
h1 is sending IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='1.1.1.1', ttl=64, protocol=0, checksum=9815,
payload=Data{data='hi :D'}}
h1 found dstIp is not in the subnet, nexHop: 1.0.0.1
h1_i is sending Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18888,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='1.1.1.1', ttl=64,
protocol=0, checksum=9815, payload=Data{data='hi :D'}}}
r1_i1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18888,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='1.1.1.1', ttl=64,
protocol=0, checksum=9815, payload=Data{data='hi :D'}}}
r1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18888,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='1.1.1.1', ttl=64,
protocol=0, checksum=9815, payload=Data{data='hi :D'}}}
r1 is handling IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='1.1.1.1', ttl=64, protocol=0, checksum=9815,
payload=Data{data='hi :D'}}
r1 is forwarding IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='1.1.1.1', ttl=63, protocol=0, checksum=9814,
payload=Data{data='hi :D'}}
Can't find best match, best match set default match: 0.0.0.0    0.0.0.0
0.0.0.0     r1_i1
```

4.4 ARP缓存项缺失运行结果

测试：h1 需要经过 r1 给 h2 发送信息，但是 h1 的ARP 缓存中没有对应网关 r1 的条目。

注意：需要先删除 `src/main/resources/config/arp_cache/h1.ac` 中的 `1.0.0.1 00:11:22:33:44:55` 条目，并删除空行，保证底部最多有一个空行。



```
/**
         *  h1 ?-> h2               @TEST: ARP_REQ
         *  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
         *  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
         *  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
         *  NOTIFICATION:
         *  DELETE THE ENTRY `1.0.0.1 00:11:22:33:44:55`
         *  IN THE FILE `src/main/resources/config/arp_cache/h1.ac`
         *  MANUALLY,
         *  MAKE SURE THERE IS <=1 BLANK LINE AT THE BOTTOM
         *  AND TRY THIS AGAIN:
         *  */
        net.service.sendIPPacket(H1_HOSTNAME, H2_I_IP, message, 64);
```

理想结果：h1 发送 ARP 请求获得 r1 的 MAC 后，进行常规发送。

测试结果：

```
h1 is sending ARP packet:Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF', destinationMAC='FF:FF:FF:FF:FF:FF', etherType=2054, checksum=2655
h1_i is sending Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF', destinationMAC='FF:FF:FF:FF:FF:FF', etherType=2054, checksum
r1_i1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF', destinationMAC='FF:FF:FF:FF:FF:FF', etherType=2054, check
r1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF', destinationMAC='FF:FF:FF:FF:FF:FF', etherType=2054, checksum
r1 is handling ARP packet: ARP{hardwareType=1, protocolType=2048, opCode=1, senderHardwareAddress='AA:BB:CC:DD:EE:FF', senderProto
r1 is sending ARP packet:Ethernet{sourceMAC='00:11:22:33:44:55', destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2054, checksum=0, p
r1_i1 is sending Ether packet: Ethernet{sourceMAC='00:11:22:33:44:55', destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2054, checksu
h1_i is receiving Ether packet: Ethernet{sourceMAC='00:11:22:33:44:55', destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2054, checks
h1 is receiving Ether packet: Ethernet{sourceMAC='00:11:22:33:44:55', destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2054, checksum
h1 is handling ARP packet: ARP{hardwareType=1, protocolType=2048, opCode=2, senderHardwareAddress='00:11:22:33:44:55', senderProto
h1_i is sending Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF', destinationMAC='00:11:22:33:44:55', etherType=2048, checksum
r1_i1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF', destinationMAC='00:11:22:33:44:55', etherType=2048, check
r1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF', destinationMAC='00:11:22:33:44:55', etherType=2048, checksum
r1 is handling IP packet: IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=64, protocol=0, checksum=9840, payload=
r1 is forwarding IP packet: IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=63, protocol=0, checksum=9839, payloa
r1_i2 is sending Ether packet: Ethernet{sourceMAC='11:22:33:44:55:00', destinationMAC='BB:CC:DD:EE:FF:AA', etherType=2048, checksu
h2_i is receiving Ether packet: Ethernet{sourceMAC='11:22:33:44:55:00', destinationMAC='BB:CC:DD:EE:FF:AA', etherType=2048, checks
h2 is receiving Ether packet: Ethernet{sourceMAC='11:22:33:44:55:00', destinationMAC='BB:CC:DD:EE:FF:AA', etherType=2048, checksum
h2 is handling IP packet: IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=63, protocol=0, checksum=9839, payload=
h2 accepted message: hi :]
/**````````````````````````````````````````````````````````````*/
h1: Found it: 1.0.0.1   00:11:22:33:44:55
```

```
    INPUT YOUR MESSAGE HERE: hi :]
/**````````````````````YOUR MESSAGE IS SENDING!````````````````````*/
h1 is sending IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='2.0.0.2', ttl=64, protocol=0, checksum=9840,
payload=Data{data='hi :]'}}
h1 found dstIp is not in the subnet, nexHop: 1.0.0.1
h1 can't find arp entry for: 1.0.0.1
h1 is making a new buffer queue for: 1.0.0.1
h1 is sending ARP packet:Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='FF:FF:FF:FF:FF:FF', etherType=2054, checksum=26555,
payload=ARP{hardwareType=1, protocolType=2048, opCode=1,
senderHardwareAddress='AA:BB:CC:DD:EE:FF', senderProtocolAddress='1.0.0.2',
targetHardwareAddress='null', targetProtocolAddress='1.0.0.1', payload=null}}
h1_i is sending Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='FF:FF:FF:FF:FF:FF', etherType=2054, checksum=26555,
payload=ARP{hardwareType=1, protocolType=2048, opCode=1,
senderHardwareAddress='AA:BB:CC:DD:EE:FF', senderProtocolAddress='1.0.0.2',
targetHardwareAddress='null', targetProtocolAddress='1.0.0.1', payload=null}}
```

```
r1_i1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='FF:FF:FF:FF:FF:FF', etherType=2054, checksum=26555,
payload=ARP{hardwareType=1, protocolType=2048, opCode=1,
senderHardwareAddress='AA:BB:CC:DD:EE:FF', senderProtocolAddress='1.0.0.2',
targetHardwareAddress='null', targetProtocolAddress='1.0.0.1', payload=null}}
r1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='FF:FF:FF:FF:FF:FF', etherType=2054, checksum=26555,
payload=ARP{hardwareType=1, protocolType=2048, opCode=1,
senderHardwareAddress='AA:BB:CC:DD:EE:FF', senderProtocolAddress='1.0.0.2',
targetHardwareAddress='null', targetProtocolAddress='1.0.0.1', payload=null}}
r1 is handling ARP packet: ARP{hardwareType=1, protocolType=2048, opCode=1,
senderHardwareAddress='AA:BB:CC:DD:EE:FF', senderProtocolAddress='1.0.0.2',
targetHardwareAddress='null', targetProtocolAddress='1.0.0.1', payload=null}
r1 is sending ARP packet:Ethernet{sourceMAC='00:11:22:33:44:55',
destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2054, checksum=0,
payload=ARP{hardwareType=1, protocolType=2048, opCode=2,
senderHardwareAddress='00:11:22:33:44:55', senderProtocolAddress='1.0.0.1',
targetHardwareAddress='AA:BB:CC:DD:EE:FF', targetProtocolAddress='1.0.0.2',
payload=null}}
r1_i1 is sending Ether packet: Ethernet{sourceMAC='00:11:22:33:44:55',
destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2054, checksum=26775,
payload=ARP{hardwareType=1, protocolType=2048, opCode=2,
senderHardwareAddress='00:11:22:33:44:55', senderProtocolAddress='1.0.0.1',
targetHardwareAddress='AA:BB:CC:DD:EE:FF', targetProtocolAddress='1.0.0.2',
payload=null}}
h1_i is receiving Ether packet: Ethernet{sourceMAC='00:11:22:33:44:55',
destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2054, checksum=26775,
payload=ARP{hardwareType=1, protocolType=2048, opCode=2,
senderHardwareAddress='00:11:22:33:44:55', senderProtocolAddress='1.0.0.1',
targetHardwareAddress='AA:BB:CC:DD:EE:FF', targetProtocolAddress='1.0.0.2',
payload=null}}
h1 is receiving Ether packet: Ethernet{sourceMAC='00:11:22:33:44:55',
destinationMAC='AA:BB:CC:DD:EE:FF', etherType=2054, checksum=26775,
payload=ARP{hardwareType=1, protocolType=2048, opCode=2,
senderHardwareAddress='00:11:22:33:44:55', senderProtocolAddress='1.0.0.1',
targetHardwareAddress='AA:BB:CC:DD:EE:FF', targetProtocolAddress='1.0.0.2',
payload=null}}
h1 is handling ARP packet: ARP{hardwareType=1, protocolType=2048, opCode=2,
senderHardwareAddress='00:11:22:33:44:55', senderProtocolAddress='1.0.0.1',
targetHardwareAddress='AA:BB:CC:DD:EE:FF', targetProtocolAddress='1.0.0.2',
payload=null}
h1_i is sending Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18911,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=64,
protocol=0, checksum=9840, payload=Data{data='hi :]'}}}
r1_i1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18911,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=64,
protocol=0, checksum=9840, payload=Data{data='hi :]'}}}
r1 is receiving Ether packet: Ethernet{sourceMAC='AA:BB:CC:DD:EE:FF',
destinationMAC='00:11:22:33:44:55', etherType=2048, checksum=18911,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=64,
protocol=0, checksum=9840, payload=Data{data='hi :]'}}}
```

```
r1 is handling IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='2.0.0.2', ttl=64, protocol=0, checksum=9840,
payload=Data{data='hi :]'}}
r1 is forwarding IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='2.0.0.2', ttl=63, protocol=0, checksum=9839,
payload=Data{data='hi :]'}}
r1_i2 is sending Ether packet: Ethernet{sourceMAC='11:22:33:44:55:00',
destinationMAC='BB:CC:DD:EE:FF:AA', etherType=2048, checksum=18918,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=63,
protocol=0, checksum=9839, payload=Data{data='hi :]'}}}
h2_i is receiving Ether packet: Ethernet{sourceMAC='11:22:33:44:55:00',
destinationMAC='BB:CC:DD:EE:FF:AA', etherType=2048, checksum=18918,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=63,
protocol=0, checksum=9839, payload=Data{data='hi :]'}}}
h2 is receiving Ether packet: Ethernet{sourceMAC='11:22:33:44:55:00',
destinationMAC='BB:CC:DD:EE:FF:AA', etherType=2048, checksum=18918,
payload=IPv4{version=4, sourceIP='1.0.0.2', destinationIP='2.0.0.2', ttl=63,
protocol=0, checksum=9839, payload=Data{data='hi :]'}}}
h2 is handling IP packet: IPv4{version=4, sourceIP='1.0.0.2',
destinationIP='2.0.0.2', ttl=63, protocol=0, checksum=9839,
payload=Data{data='hi :]'}}
h2 accepted message: hi :]
/**``````````````````````````````````````````````````````````*/
h1: Found it: 1.0.0.1   00:11:22:33:44:55
```

5. 总结与感谢

任务的实现汇聚了不仅限于一个课设周一个人的苦思冥想，而是来自于一学期的深入钻研，同时也凝结了很多其他人的思想精华：

- 感谢我的计算机网络王苏老师，无法忘记她课上的详细讲解和认真耐心地解答我一学期超级多的学习疑惑；
- 感谢《计算机网络 自顶向下方法》的作者，他们生动有趣的讲解很大程度上加深了我对计算机网络的理解；
- 感谢 ChatGPT 老师，它在24h任何我有困惑的时候都提供全领域逻辑清晰的解答，尽管回答的未必完全对，但开导了我的思路，引导我提出更正确和更上层的问题；
- 感谢清华大学、威斯康星大学麦迪逊分校、斯坦福大学的开源项目，它们也引导了我的实现思路和补充了我的实现逻辑；
- 感谢无数网络博客；
- 感谢计算机网络这个深奥而又伟大的创造；