1 定义的结构和变量

1.1 typedef struct Node

表示路由表的节点。其中有四个部分：

- unsigned int dest：子网地址
- unsigned int maskLen：掩码长度（可以不用存，因为下面存了掩码mask）
- unsigned int mask： 子网掩码：为1的部分表示子网地址，为0的部分表示主机地址
- unsigned int nexthop：下一跳的ip地址

1.2 vector<Node> RouteList

路由表。

本来想用树状路由表，但由于路由匹配算法是最长匹配前缀，需要遍历，所以就改成了vector。

1.3 函数中的变量

将在每个函数中进行解释。

2 函数及实现逻辑

2.1 void stud_Route_Init()

函数作用、函数参数和函数返回值已经在指导书中说明，故这里只阐述实现逻辑。

调用vector::clear()函数，将RouteList中的内容清空。

2.2 void stud_route_add(stud_route_msg *proute)

函数作用、函数参数和函数返回值已经在指导书中说明，故这里只阐述实现逻辑。

- 首先利用ntohl函数将proute中的网络序全部转成字节序
- 再利用masklen得到掩码mask
- 最后将四个变量合并成Node插入RouteList中。

2.3 stud_fwd_deal(char *pBuffer, int length)

函数作用、函数参数和函数返回值已经在指导书中说明，故这里只阐述实现逻辑。

- 先从ip头部得到头部长度headLen
- 根据头部长度建立headBuffer数组，unsigned int类型， 存放从网络序转成字节序后的ip头部
- 再从ip头部得到TTL和目的地址，检查TTL是否出错（<=0），若出错调用fwd_DiscardPkt函数丢弃pBuffer并指明错误类型，返回1
- 若目的地址就是路由器地址，则调用fwd_LocalRcv函数接受ip分组，返回0
- 若目的地址不是路由器地址，遍历路由表寻找下一跳地址，注意这里按照最长匹配前缀算法寻找
- 若找不到，则调用fwd_DiscardPkt函数丢弃pBuffer并指明错误类型，返回1
- 若找到下一跳的ip地址，则进行如下操作：
  - TTL值-1，并同步更新到ip头部中
  - 将ip头部的校验码位全部归0
  - 重新计算ip头部的校验码，填入对应位置中
  - 将ip头部所有字节序全部转为网络序
  - 更新ip头部
- 最后，调用fwd_SendtoLower函数发送给下一跳路由器或目的地址。

3 实验过程的重点难点及遇到的问题

- 关于网络序和主机序的转换：

  

  htonl是按照int大小转换的，所以我们在将头部分组是指针类型是unsigned int *。

- 这里的位数是按照从右到左从低到高。

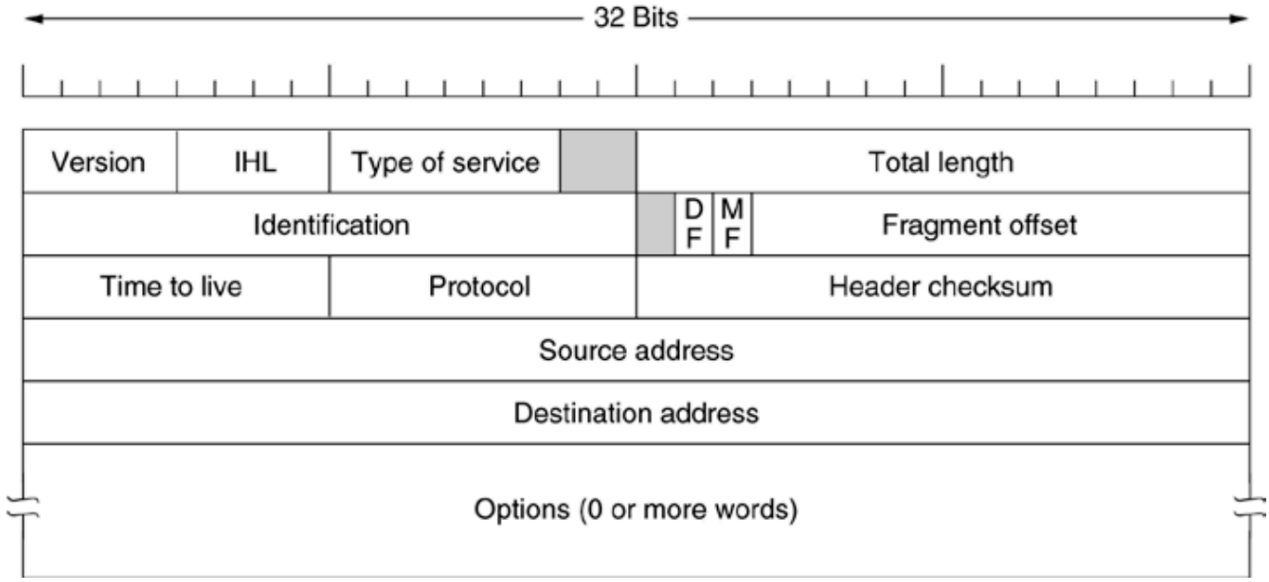

图 1.3　IPv4 分组头部格式

- 注意headLen是有固定的20字节，但下面也可能有可变字节，不能默认headLen就是20，要根据IHL中记录的数字确定headLen。

- 校验码要先取反再转为unsigned short，否则取反操作后结果默认为为short，结果不为0，没法和0比。

- TTL检验要转化成char检验是否 < 0。

4 感想和建议

这次试验和ipv4收发实验比较相似，但我们需要额外做的工作是自己构建路由表。

不同路由表的构建和搜索算法会导致寻找下一跳地址的时间不同，从而影响ip转发的效率。

可以利用堆排序优先寻找masklen长的路由表项进行匹配；也可以先对RouteList按照masklen排序，按照masklen从长到短的顺序匹配路由表项。