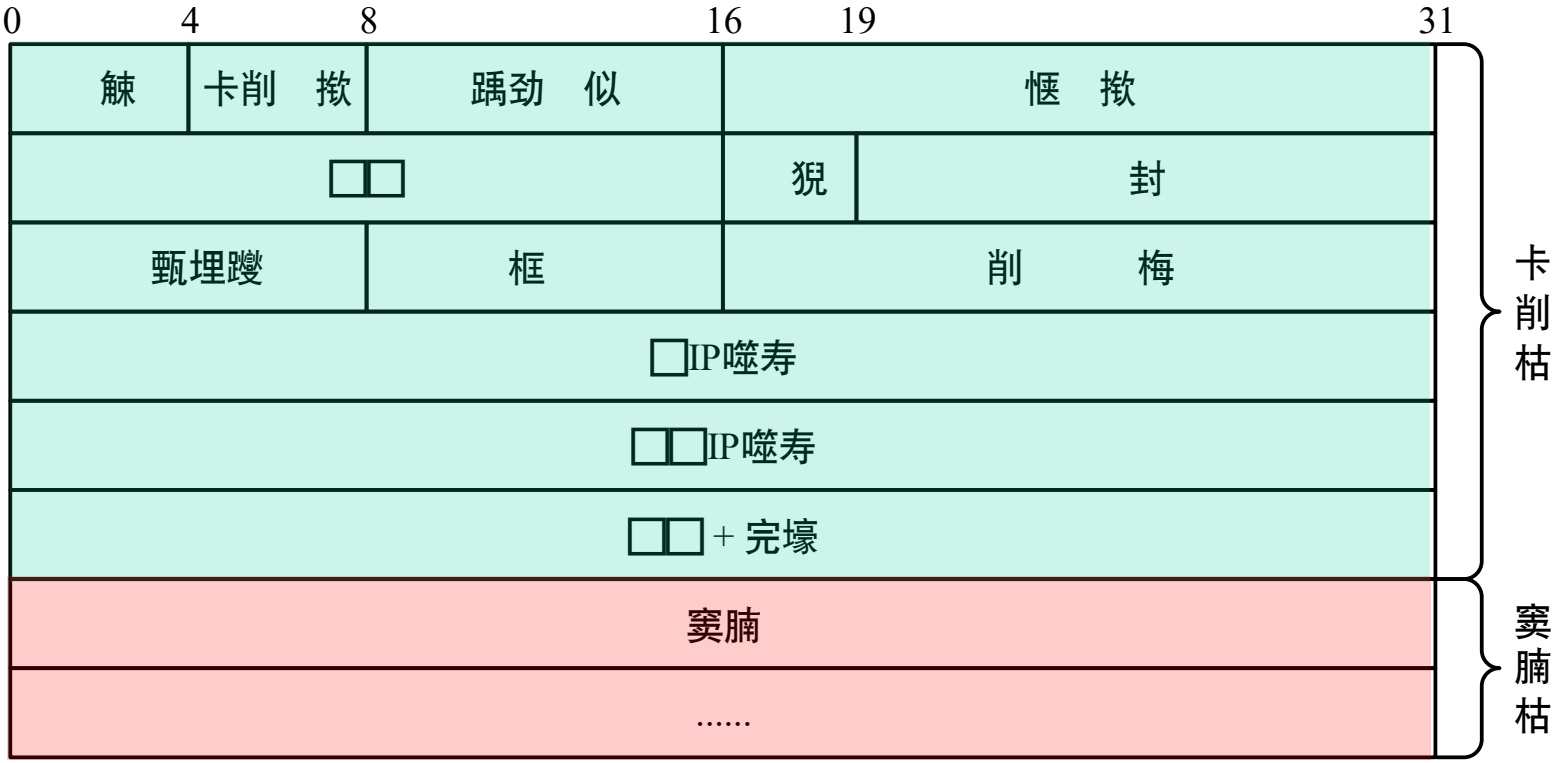
计算机网络技术与应用

第6章

IP数据包

IP数据包的格式

- IP数据包：报头区和数据区
  - 数据区：高层传输的数据
  - 包头区：为了正确传输高层数据而增加的控制信息



主要字段（1/2）

- 版本与协议类型
  - 版本：数据包对应的IP协议版本号（4和6）
  - 协议类型：数据区数据的高级协议类型（如TCP、UDP等）
- 长度
  - 包头长度：包头区的长度（以32bit双字为单位）
  - 总长度：整个IP数据包的长度（以8bit字节为单位）
- 服务类型
  - 转发过程中对该数据包的处理方式

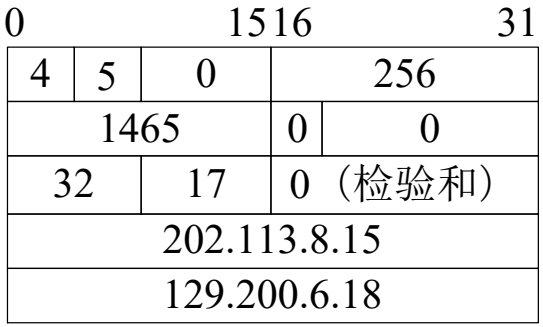
主要字段（2/2）

- 生存周期：IP数据包在互联网中的存活时间（避免死循环）
- 头部校验和：保证IP数据包包头的完整性
- 地址： IP地址采用32位的地址形式
  - 源IP地址：数据包的发送者
  - 目的IP地址：数据包的接收者

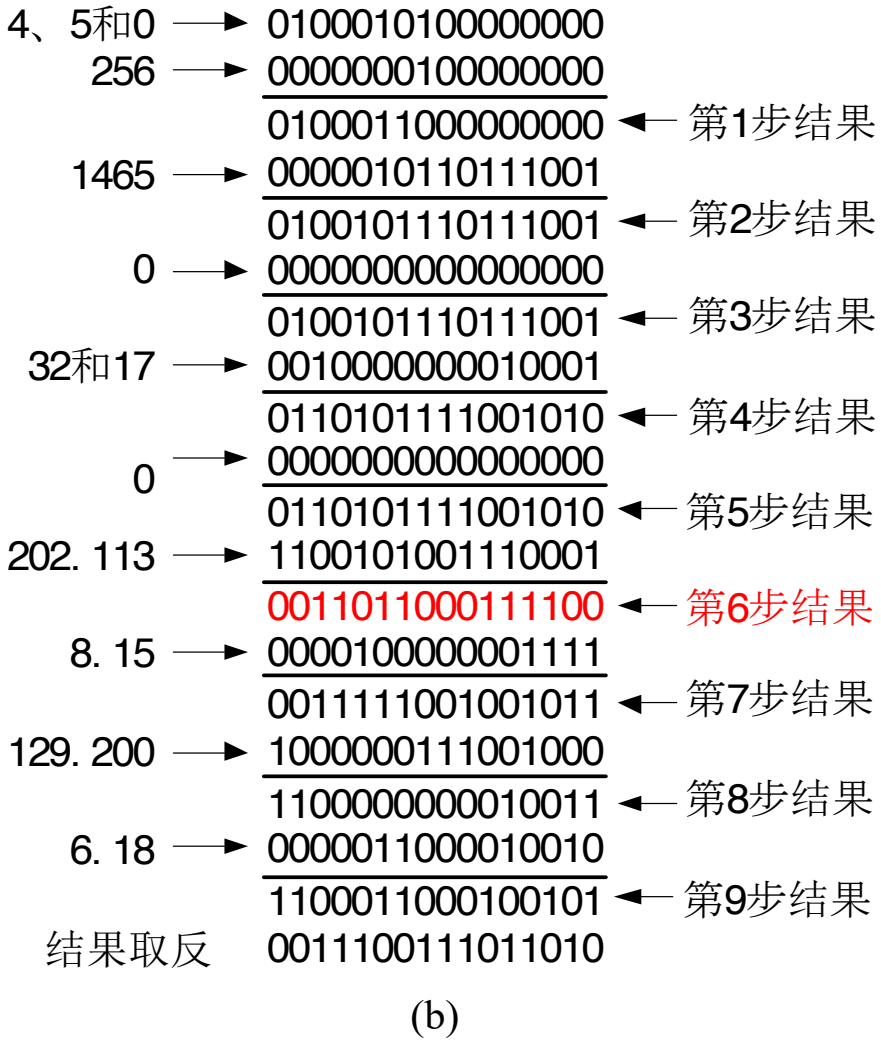
IP头部校验和算法（1/2）

- 将IP头部看成16位字组成的二进制数据序列，把头部校验和字段置为0
- 对IP头部中每个16位字进行求和运算。如果求和过程中遇到溢出则进行回卷（即如果求和过程中遇到进位，则将进位加至结果的最低位）
- 对求和的结果取反，得到最终的头部校验和值

IP头部校验和算法（2/2）

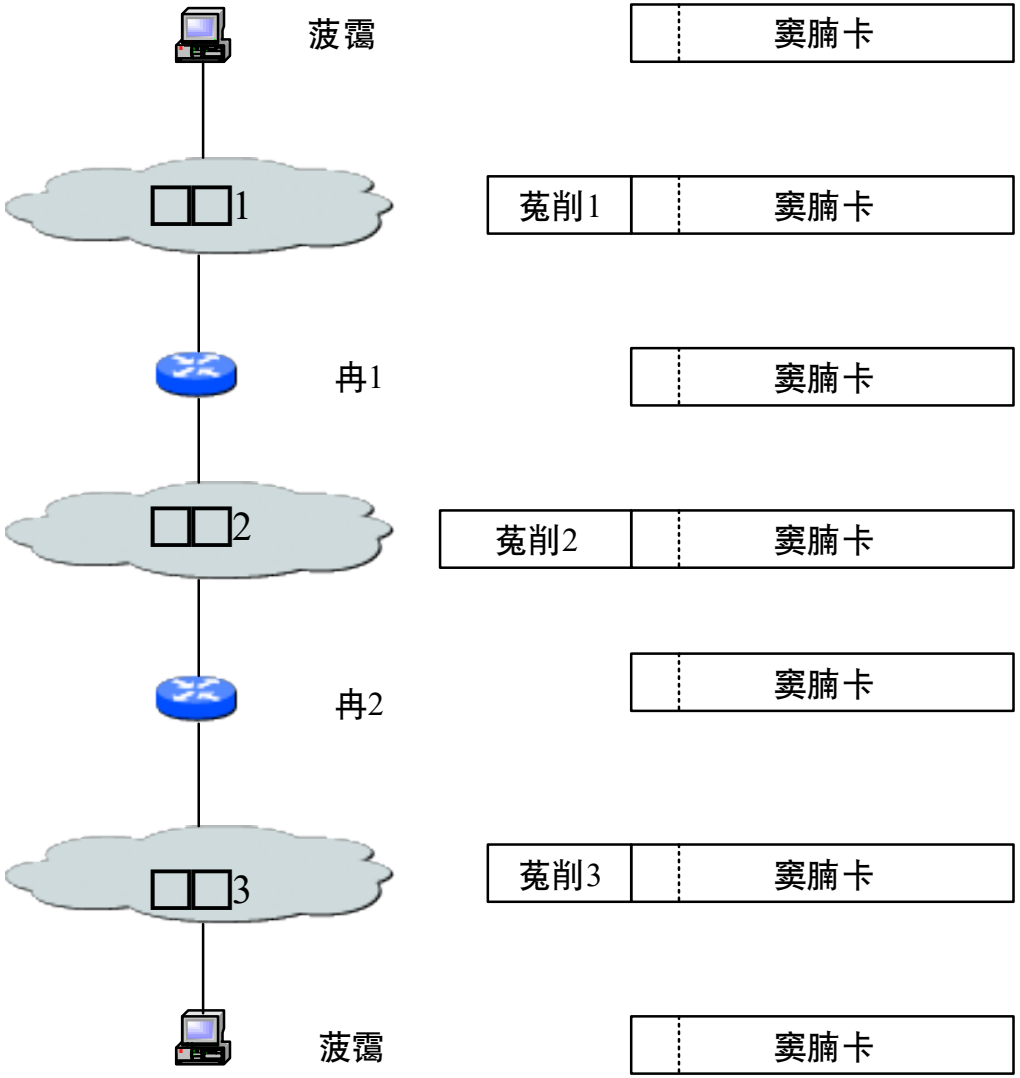


(a)



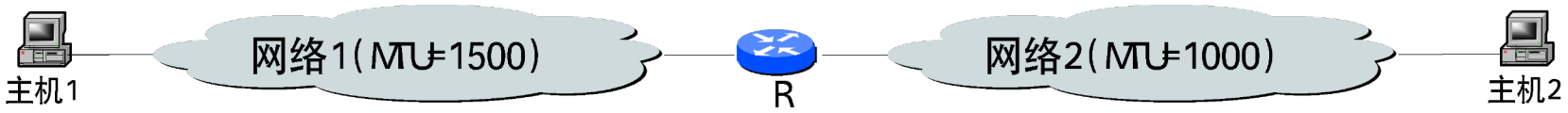
(b)

IP数据包的封装



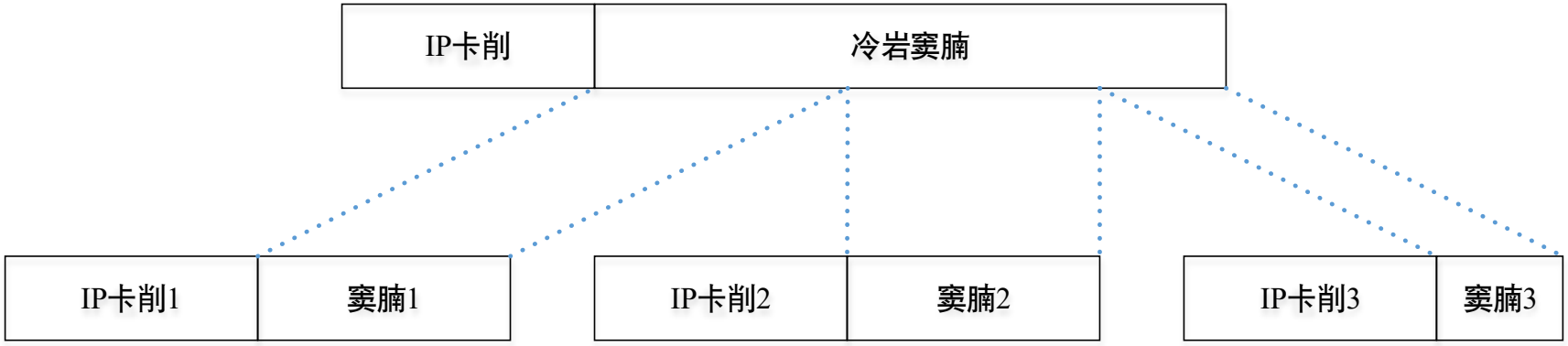
MTU

- MTU：一个帧最多能够携带的数据量
- IP数据包的长度只有小于或等于网络的MTU，才能在这个网络传输
- 与路由器连接的各个网络的MTU可能不同



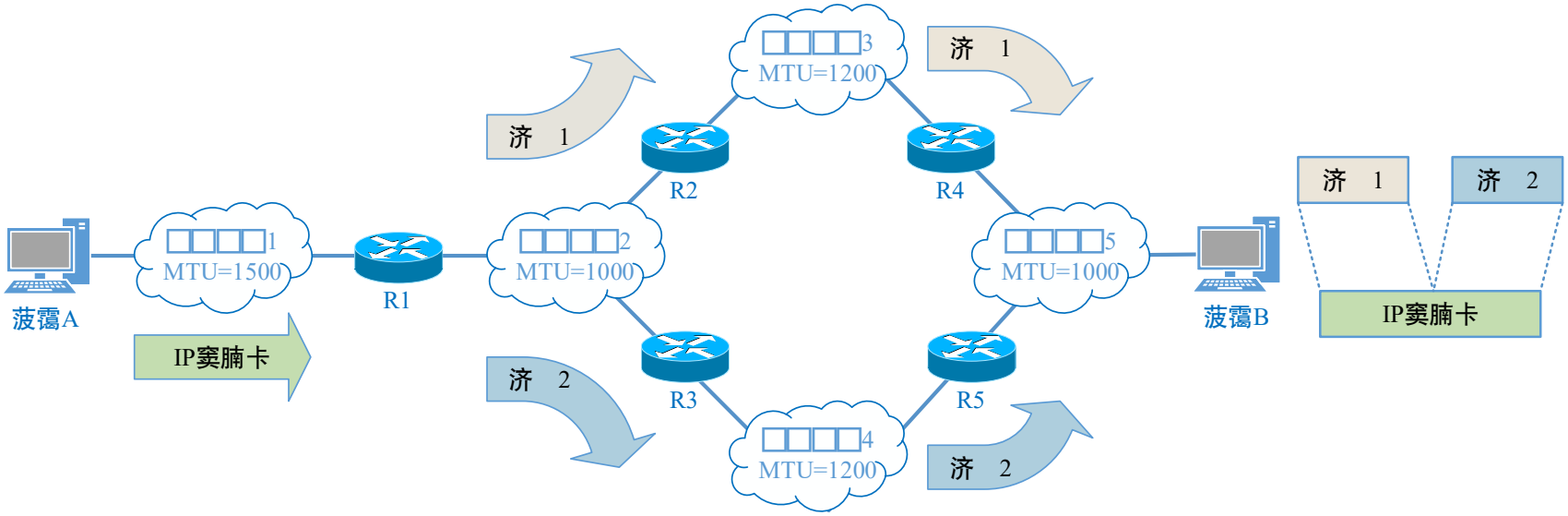
分片

- 分片：IP数据包的尺寸大于将发往网络的MTU值时，将IP数据包分成若干多个较小数据包的过程
- 分片由包头区和数据区两部分构成



分片的转发与投递

- 每个分片独立进行路由选择，最终到达目的主机



重组

- 重组：收到所有分片后，设备对分片进行重新组装的过程
- 目的主机负责重组
  - 减少了中间路由器的计算量
  - 路由器可以为每个分片独立选路
- 中间路由器不需要对分片进行重组，也不可能对分片进行重组

分片控制

- 标识：源主机赋予IP数据包的标识符
  - 需复制到新分片的包头中
  - 目的主机利用此字段和目的地址判断分片属于哪个数据包
- 标志：是否已经分片，是否是最后一个分片
- 片偏移
  - 本片数据在初始IP数据包数据区的位置
  - 偏移量以8个字节为单位

IP数据包选项

- 功能：主要用于控制和测试
- 用户可以使用也可以不使用IP选项，但所有实现IP协议的系统必须能处理IP选项
- 选项组成：选项码、长度和选项数据

源路由选项

- 源路由：IP数据包穿越互联网所经过的路径是由源主机指定的
- 应用场合：测试特定网络的吞吐率、使数据包绕开出错网络等
- 分类：
  - 严格源路由：规定IP数据包需经过的每个路由器
  - 松散源路由：给出IP数据包需经过的一些“要点”

记录路由选项

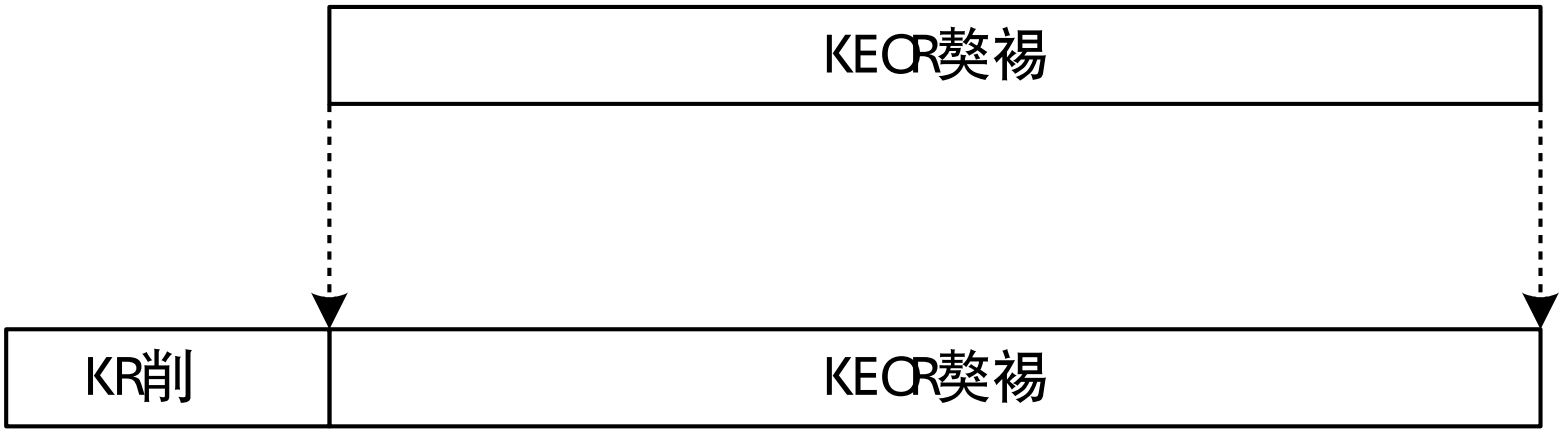
- 记录路由：记录IP数据包从源主机到目的主机所经过路径上各个路由器的IP地址
- 应用场合
  - 查看IP数据包传输过程中所经过的路径
  - 测试路由器的路由配置是否正确

时间戳选项

- 时间戳：记录IP数据包经过每一路由器时的当地时间
- 应用场合：分析网络吞吐率、拥塞情况、负载情况等

差错与控制报文

- IP互联网利用ICMP传输控制报文和差错报文
- ICMP报文封装在IP数据包中传递



ICMP差错控制

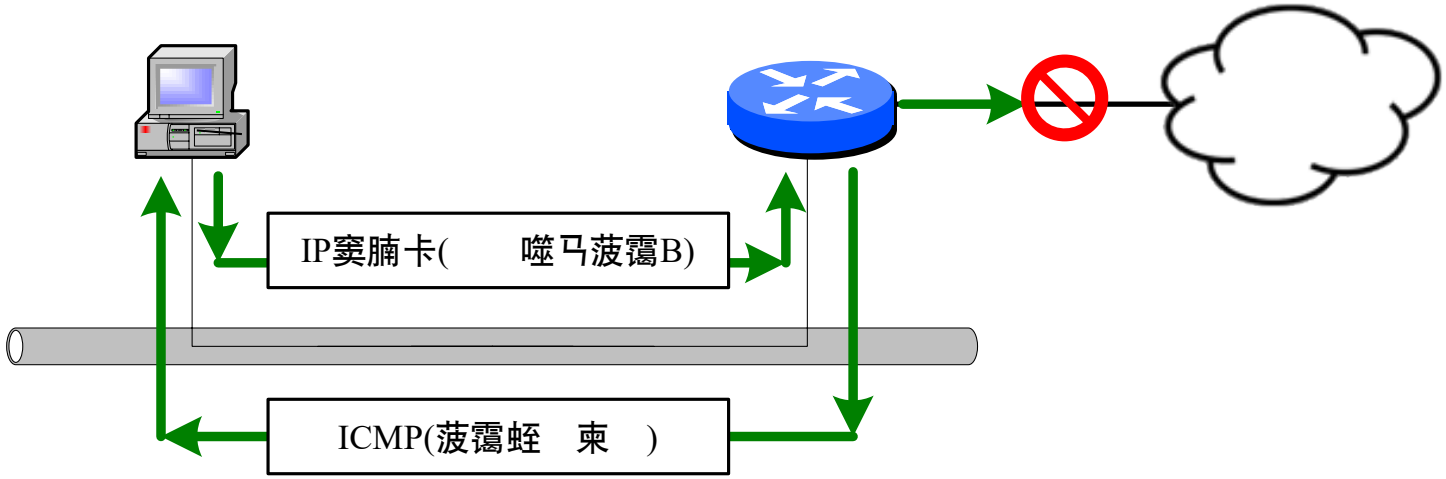
- 提供差错报告是ICMP的基本功能之一
- ICMP不严格规定如何处理差错
- ICMP差错报告采用路由器到源主机模式
  - IP数据包只包含源主机地址和目的主机地址，错误报告给目的主机没有意义（有时也不可能）
  - 路由器独立选路，发现错误的路由器不知道数据包经过的路径，无法通知相应路由器

ICMP差错报文的主要特点

- ICMP差错报告作为一般数据传输，不享受特别优先权和可靠性
- ICMP差错报告数据：故障IP数据包报头 + 故障IP数据包数据区的前64bit数据
- ICMP差错报告是伴随着抛弃出错IP数据包而产生的

ICMP主要差错报告类型

- 目的地不可达报告：网络不可达、主机不可达、协议和端口不可达等



- 超时报告：头部TTL阈值超时、分片到达超时
- 参数出错报告：头部校验和错误等

ICMP控制报文

- 拥塞控制与源抑制报文
- 路由控制与重定向报文
- ICMP请求/应答报文对

拥塞控制与源抑制报文（1/2）

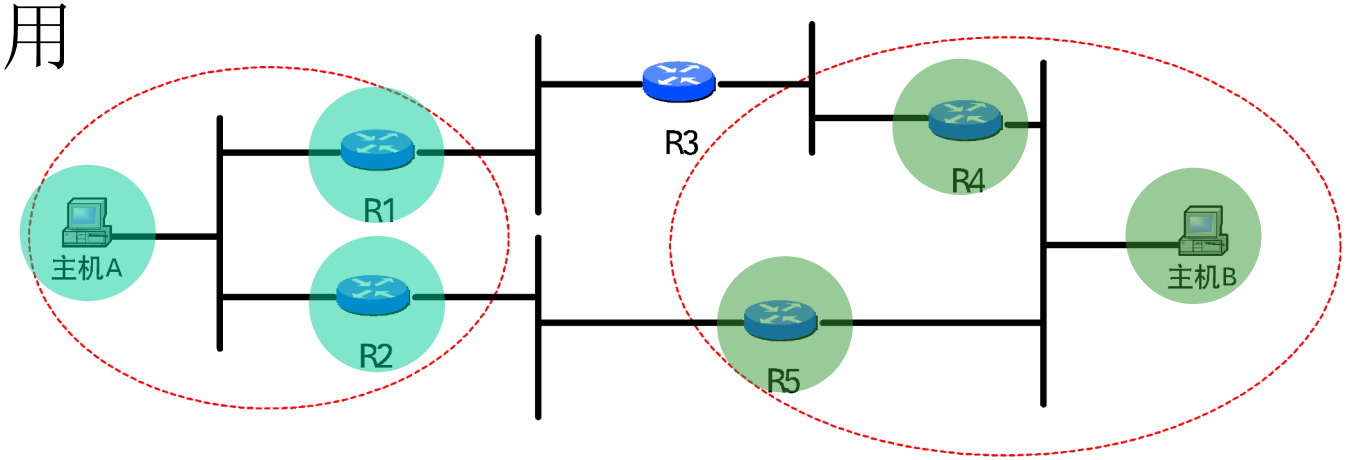
- 拥塞：路由器被大量涌入的IP数据包“淹没”的现象
- 拥塞原因
  - 路由器的处理速度太慢
  - 路由器传入数据速率大于传出数据速率
- 拥塞控制：源站抑制（抑制源主机发送速率）

拥塞控制与源抑制报文（2/2）

- 发送源站抑制报文策略
  - 输出队列溢出后，抛弃新来的数据包，而后发送
  - 设置阈值，超过后抛弃新来的数据包，而后发送
  - 有选择地抑制IP数据包发送率较高的源主机
- 接收源站抑制报文
  - 可降低发送IP数据包的速率
  - 注意：拥塞解除后路由器不主动通知源主机

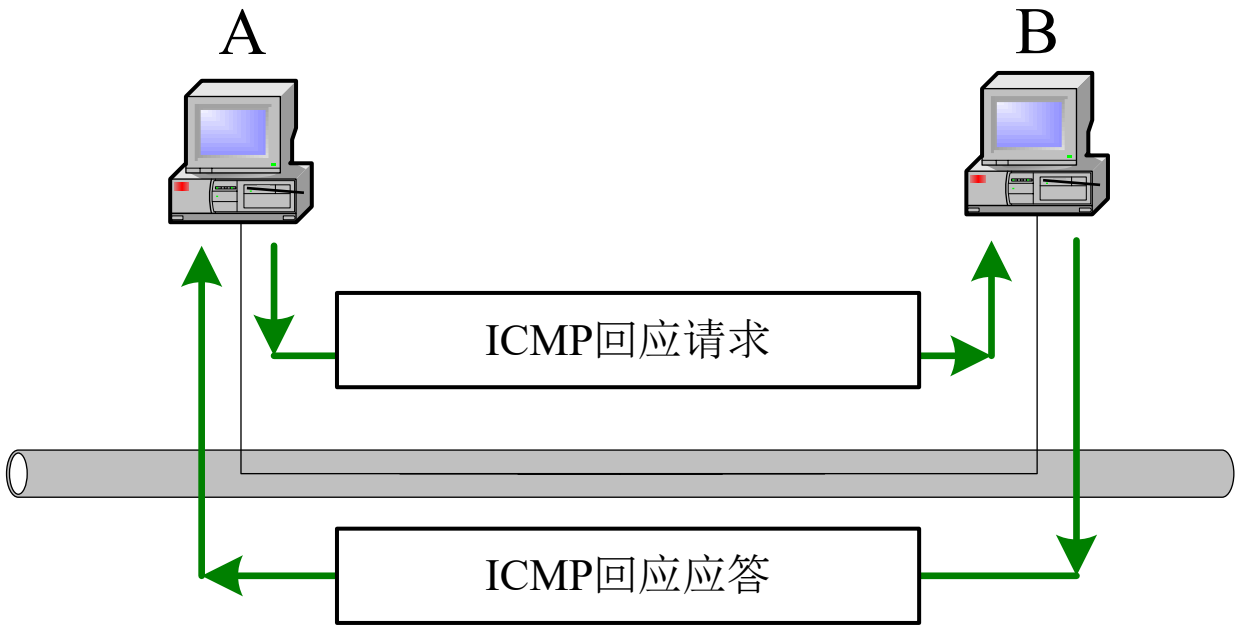
路由控制与重定向报文

- ICMP重定向机制
  - 主机在启动时具有一定的路由信息，但不一定是最优的
  - 路由器若检测到IP数据包经非优路由传输，则通知主机去往该目的地的最优路径
  - 功能：保证主机拥有动态的、既小且优的路由表
- ICMP重定向机制只能在同一网络的路由器与主机之间使用



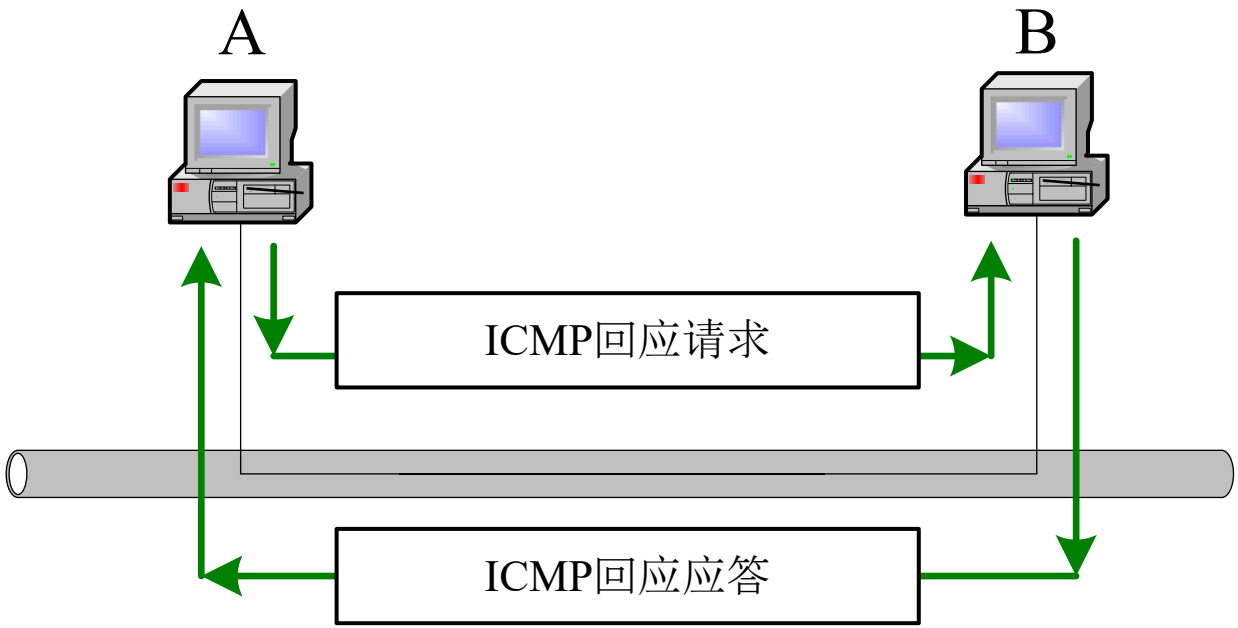
ICMP请求/应答报文对

- 回应请求与应答：测试目的主机或路由器的可达性
- 时戳请求与应答：获取其他设备的当前时间
- 掩码请求与应答：从路由器获取本网的子网掩码



回应请求与应答ICMP的机理

- 请求者向特定目的IP地址发送包含任选数据区的回应请求报文
- 目的主机或路由器收到后响应应答报文（包含请求报文中任选数据的拷贝）



请求者成功收到应答可说明什么？

- 目的主机（或路由器）可以到达
- 源主机与目的主机（或路由器）的ICMP软件和IP 软件工作正常
- 回应请求与应答ICMP报文经过的中间路由器的路由选择功能正常

实验： IP数据包捕获与分析

- 实验环境：以太网环境
- 实验方法
  - 监听与分析工具（如Wireshark、snort、tcpdump等）
  - 利用Npcap或LibPcap编写捕获与分析程序
- 实验目的
  - 学习网络数据包捕获方法
  - 初步掌握网络监听与分析技术的实现过程
  - 理解IP数据包校验和计算方法

Npcap

- 是一个开源的、运行于Windows的数据包捕获与发送函数库
- 主要功能：数据包捕获、发送和网络分析
  - Packet.dll：内核级、低层次的包过滤动态连接库
  - wpcap.dll：高级别系统无关函数库
- 安装和使用：http://npcap.com
  - 安装Npcap驱动程序和DLL程序
  - 开发工具包：库文件、包含文件、简单的示例程序代码和帮助文件

获取设备列表

```
int pcap_findalldevs_ex(
                char *source,
                struct  pcap_rmtauth auth,
                pcap_if_t **alldevs,
                char *errbuf
);

Typedef struct pcap_if pcap_if_t;
struct pcap_if {
                struct pcap_if *next;
                char *name;
                char *description;
                struct pcap_addr *addresses;
                u_int flags;
};
struct pcap_addr {
                struct pcap_addr *next;
                struct sockaddr *addr;
                struct sockaddr *netmask;
                struct sockaddr *broadaddr;
                struct sockaddr *dstaddr;
};
```

释放设备列表

```
void pcap_freealldevs(pcap_if_t *alldevsp);
```

例：本机接口和IP地址的获取

```
pcap_if_t          *alldevs;                              //指向设备链表首部的指针
pcap_if_t          *d;
pcap_addr_t        *a;
char               errbuf[PCAP_ERRBUF_SIZE]; //错误信息缓冲区
//获得本机的设备列表
if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING,          //获取本机的接口设备
                   NULL,                             //无需认证
                   &alldevs,                         //指向设备列表首部
                   errbuf                            //出错信息保存缓存区
                   ) == -1)
{          ……          //错误处理          }

for(d= alldevs; d != NULL; d= d->next)          //显示接口列表
{
          ……          //利用d->name获取该网络接口设备的名字
          ……          //利用d->description获取该网络接口设备的描述信息
          //获取该网络接口设备的IP地址信息
          for(a=d->addresses; a!=NULL; a=addr->next)
                   if (a->addr->sa_family==AF_INET)   //判断该地址是否IP地址
                   {
                             ……          //利用a->addr获取IP地址
                             ……          //利用a->netmask获取网络掩码
                             ……          //利用a->broadaddr获取广播地址
                             ……          //利用a->dstaddr)获取目的地址
                   }
}
pcap_freealldevs(alldevs); //释放设备列表
```

打开网络接口

```
pcap_t* pcap_open(
                    const char *source,
                    int snaplen,
                    int flags,
                    int read_timeout,
                    struct pcap_rmtauth *auth,
                    char *errbuf
);
```

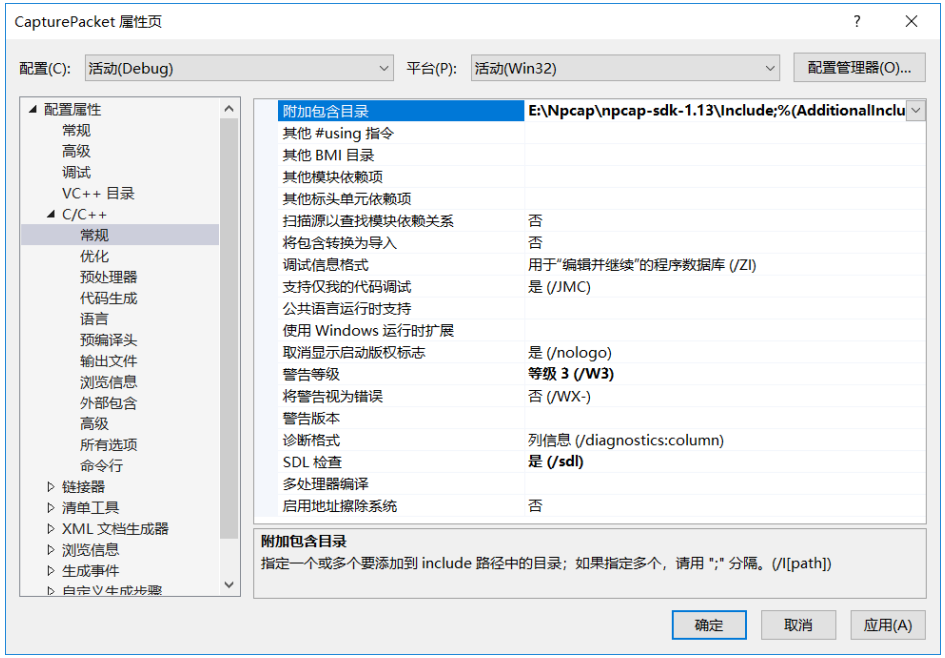
捕获网络数据包

- 利用回调函数捕获
  - pcap_dispatch()：read_timeout到时返回
  - pcap_loop()：捕获到cnt个数据包后返回
- 直接捕获
  - pcap_next_ex()：read_timeout到时返回

```
int pcap_next_ex(
                    pcap_t* p,
                    struct pcap_pkthdr **pkt_header,
                    u_char ** pkt_data
};
```

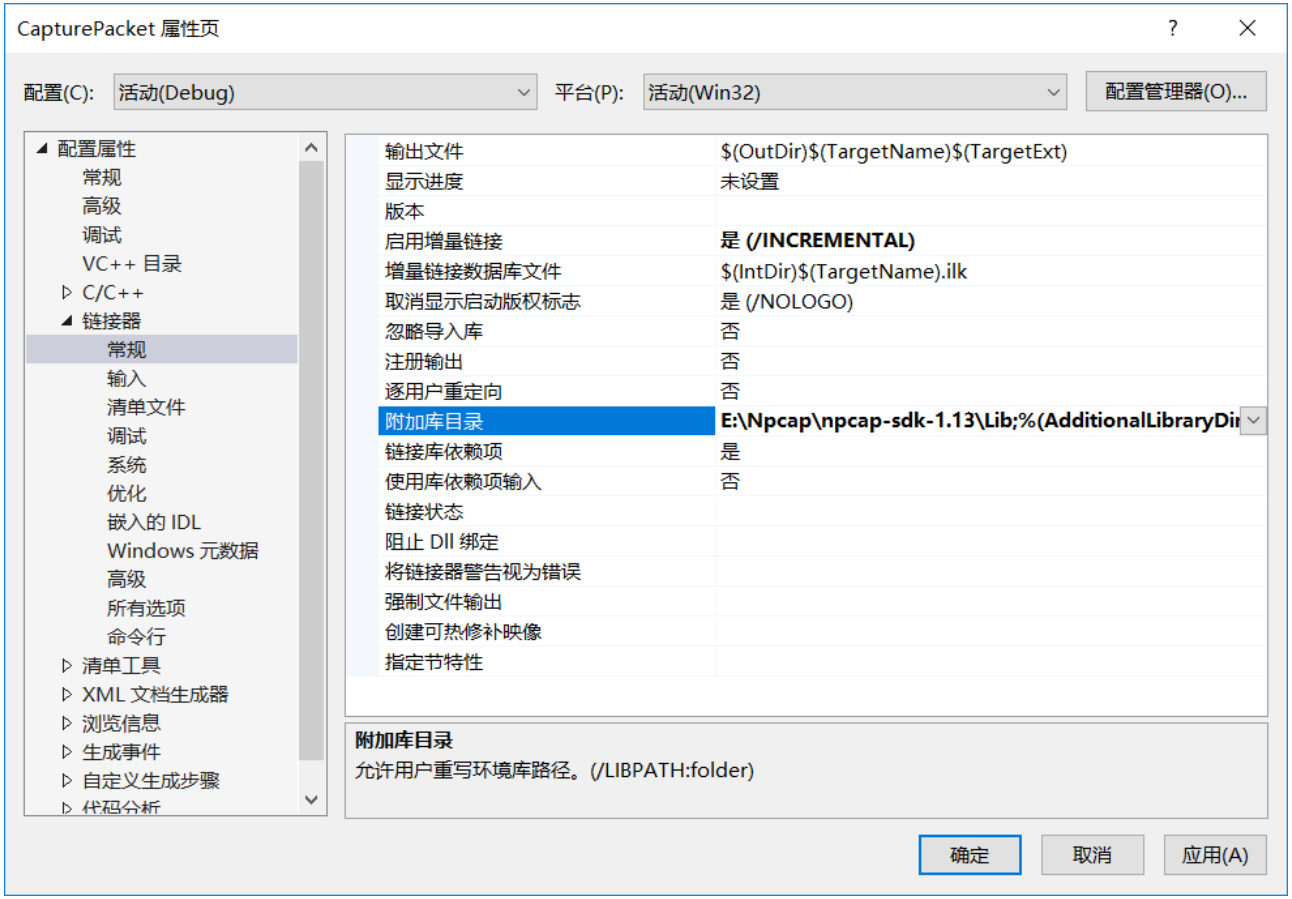
创建基于WinPcap的应用程序

- 添加pcap.h包含文件
  - 所有使用Npcap函数的源文件中都需添加pcap.h包含文件：
    #include "pcap.h
- 添加包含文件目录
  - 项目属性 - 配置属性 - C/C++ - 常规 - 附加包含目录



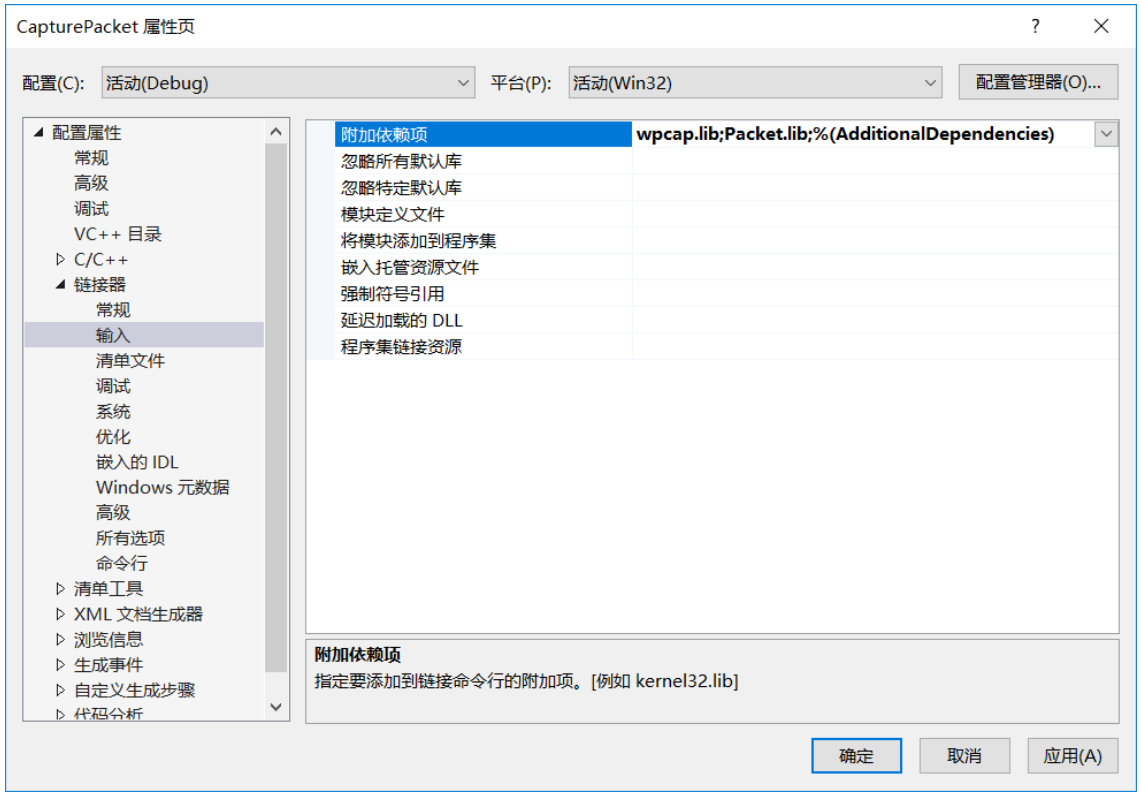
创建基于WinPcap的应用程序

- 添加库文件目录
  - 项目 - 属性 - 配置属性 - 连接器 - 常规 - 附加库目录



创建基于WinPcap的应用程序

- 添加链接时使用的库文件：二选一
  - 源文件中添加：#pragma comment(lib,"xxx.lib")
  - 项目 - 属性 - 配置属性 - 连接器 - 输入 - 附加依赖项



字节顺序

- 网络序→主机序
  - u_short ntohs(u_short netshort)
  - u_long ntohl(u_long netlong)
- 主机序→网络序
  - u_short htons(u_short hostshort)
  - u_long htonl(u_long hostlong)

以太网帧和IP数据包的结构定义

```
#pragma pack(1)                    //进入字节对齐方式
typedef struct FrameHeader_t  {        //帧首部
    BYTE        DesMAC[6];      // 目的地址
    BYTE        SrcMAC[6];      // 源地址
    WORD        FrameType;      // 帧类型
} FrameHeader_t;
typedef struct IPHeader_t {            //IP首部
        BYTE    Ver_HLen;
        BYTE    TOS;
        WORD    TotalLen;
        WORD    ID;
        WORD    Flag_Segment;
        BYTE    TTL;
        BYTE    Protocol;
        WORD    Checksum;
        ULONG   SrcIP;
        ULONG   DstIP;
} IPHeader_t;
typedef struct Data_t {        //包含帧首部和IP首部的数据包
        FrameHeader_t  FrameHeader;
        IPHeader_t             IPHeader;
} Data_t;
#pragma pack() //恢复缺省对齐方式
```

例：提取源IP地址和目的IP地址

```
Data_t     * IPPacket;
ULONG      SourceIP,DestinationIP;
……
IPPacket = (Data_t *) pkt_data;
……
SourceIP = ntohl(IPPacket->IPHeader.SrcIP);
DestinationIP = ntohl(IPPacket-
>IPHeader.DstIP);
……
```

捕获IP数据包并验证其正确性