

第10章 IP的分片与重装

10.1 引言

我们将第8章的IP的分片与重装处理问题推迟到本章来讨论。

IP具有一种重要功能，就是当分组过大而不适合在所选硬件接口上发送时，能够对分组进行分片。过大的分组被分成两个或多个大小适合在所选网络上发送的 IP分片。而在去目的主机的路途中，分片还可能被中间的路由器继续分片。因此，在目的主机上，一个 IP数据报可能放在一个IP分组内，或者，如果在发送时被分片，就放在多个 IP分组内。因为各个分片可能以不同的路径到达目的主机，所以只有目的主机才有机会看到所有分片。因此，也只有目的主机才能把所有分片重装成一个完整的数据报，提交给合适的运输层协议。

图8-5显示在被接收的分组中，0.3%(72 786/27 881 978)是分片，0.12% (264 484/(29 447 726-796 084))的数据报是被分片后发送的。在 world.std.com上，被接收分组的9.5%是被分片的。world有更多的NFS活动，这是IP分片的主要来源。

IP首部内有三个字段实现分片和重装：标识字段(ip_id)、标志字段(ip_off的3个高位比特)和偏移字段(ip_off的13个低位比特)。标志字段由三个1 bit标志组成。比特0是保留的，必须为0；比特1是“不分片”(DF)标志；比特2是“更多分片”(MF)标志。Net/3中，标志和偏移字段结合起来，由ip_off访问，如图10-1所示。

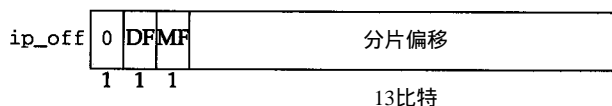


图10-1 ip_off 控制IP分组的分片

Net/3通过用IP_DF和 IP_MF掩去ip_off来访问DF和MF。IP实现必须允许应用程序请求在输出的数据报中设置DF比特。

当使用UDP或TCP时，Net/3并不提供对DF比特的应用程序级的控制。

进程可以用原始IP接口(第32章)构造和发送它自己的IP首部。运输层必须直接设置DF比特。例如，当TCP运行“路径MTU发现(path MTU discovery)”时。

ip_off的其他13 bit指出在原始数据报内分片的位置，以8字节为单元计算。因而，除最后一个分片外，其他每个分片都希望是一个8字节倍数的数据，从而使后面的分片从8字节边界开始。图10-2显示了在原始数据报内的字节偏移关系，以及在分片的IP首部内分片的偏移(ip_off的低位13 bit)。

图10-2显示了把一个最大的IP数据报分成8190个分片，除最后一个分片包含3个字节外，其他每个分片都包含8个字节。图中还显示，除最后一个分片外，设置了其余分片的MF比特。这是一个不太理想的例子，但它说明了一些实现中存在的问题。

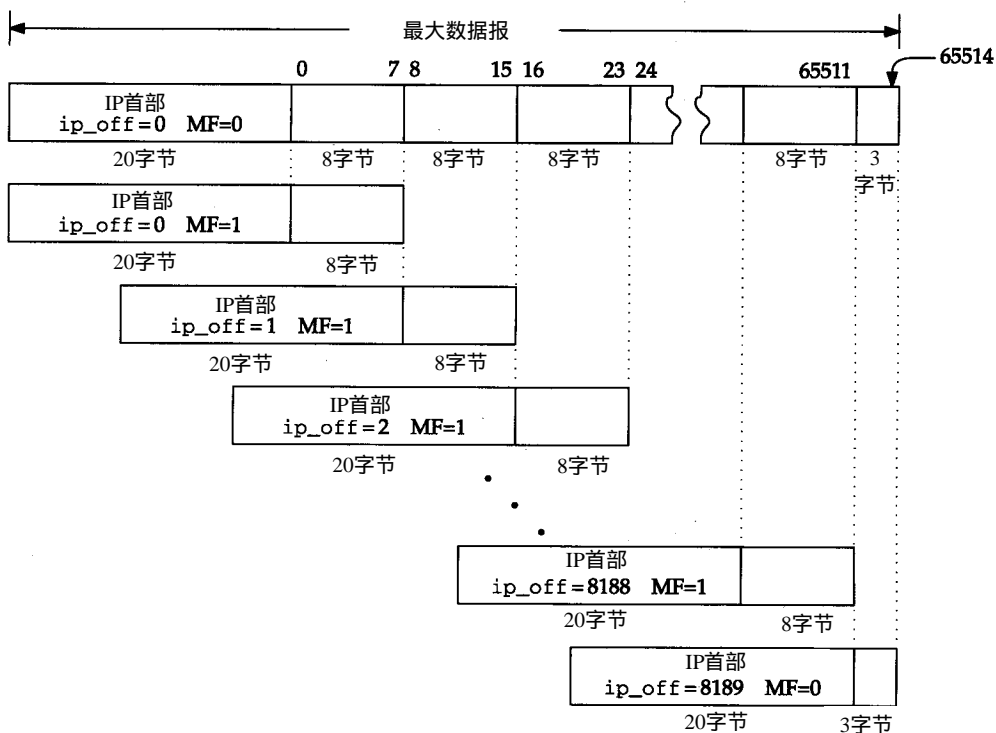


图10-2 65535字节的数据报的分片

原始数据报上面的数字是该数据部分在数据报内的字节偏移。分片偏移 (ip_off) 是从数据报的数据部分开始计算的。分片不可能含有偏移超过 65514 的字节，因为如果这样的话，重装的数据报会大于 65535 字节——这是 ip_len 字段的最大值。这就限制了 ip_off 的最大值为 8189 ($8189 \times 8 = 65512$)，只为最后一个分片留下 3 字节空间。如果有 IP 选项，则偏移还要小些。

因为 IP 互联网是无连接的，所以，在目的主机上，来自一个数据报的分片必然会与来自其他数据报的分片交错。ip_id 唯一地标识某个特定数据报的分片。源系统用相同的源地址 (ip_src)、目的地址 (ip_dst) 和协议 (ip_p) 值，作为数据报在互联网上生命期的值，把每个数据报的 ip_id 设置成一个唯一的值。

总而言之，ip_id 标识了特定数据报的分片，ip_off 确定了分片在原始数据报内的位置，除最后一个分片外，MF 标识每个分片。

10.2 代码介绍

重装数据结构出现在一个头文件里。两个 C 文件中有重装和分片处理的代码。这三个文件列在图 10-3 中。

文 件	描 述
netinet/ip_var.h	重装数据结构
netinet/ip_output.c	分片代码
netinet/ip_input.c	重装代码

图10-3 本章讨论的文件

10.2.1 全局变量

本章中只有一个全局变量，`ipq`。如图10-4所示。

变 量	类 型	描 述
<code>ipq</code>	<code>struct ipq *</code>	重装表

图10-4 本章介绍的全局变量

10.2.2 统计量

分片和重装代码修改的统计量如图 10-5所示。它们是图 8-4的`ipstat`结构中所包含统计量的子集。

ipstat成员	描 述
<code>ips_cantfrag</code>	要求分片但被DF比特禁止而没有发送的数据报数
<code>ips_odropped</code>	因为内存不够而被丢弃的分组数
<code>ips_ofragments</code>	被发送的分片数
<code>ips_fragmented</code>	为输出分片的分组数

图10-5 本章收集的统计量

10.3 分片

我们现在返回到`ip_output`，分析分片代码。记得在图 8-25中，如果分组正好适合选定出接口的MTU，就在一个链路级帧中发送它。否则，必须对分组分片，并在多个帧中将其发送。分组可以是一个完整的数据报或者它自己也是前边系统创建的分片。我们分三个部分讨论分片代码：

- 确定分片大小(图10-6)；
- 构造分片表(图10-7)；以及
- 构造第一个分片并发送分片 (图10-8)。

```

253      /*
254      * Too large for interface; fragment if possible.
255      * Must be able to put at least 8 bytes per fragment.
256      */
257      if (ip->ip_off & IP_DF) {
258          error = EMSGSIZE;
259          ipstat.ips_cantfrag++;
260          goto bad;
261      }
262      len = (ifp->if_mtu - hlen) & ~7;
263      if (len < 8) {
264          error = EMSGSIZE;
265          goto bad;
266      }

```

ip_output.c

图10-6 函数`ip_output`：确定分片大小

253-261 分片算法很简单，但由于对mbuf结构和链的操作使实现非常复杂。如果DF比特禁

止分片，则 `ip_output` 丢弃该分组，并返回 `EMSGSIZE`。如果该数据报是在本地生成的，则运输层协议把该错误传回该进程；但如果分组是被转发的，则 `ip_forward` 生成一个 ICMP 目的地不可达差错报文，并指出不分片就无法转发该分组（图8-21）。

Net/3 没有实现“路径 MTU 发现”算法，该算法用来搜索到目的主机的路径，并发现所有中间网络支持的最大传送单元。卷 1 的 11.8 节和 24.2 节讨论了 UDP 和 TCP 的路径 MTU 发现。

262-266 每个分片中的数据字节数，`len` 的计算是用接口的 MTU 减去分组首部的长度后，舍去低位的 3 个比特（`&~7`）。后成为 8 字节倍数。如果 MTU 太小，使每个分片中无法含有 8 字节的数据，则 `ip_output` 返回 `EMSGSIZE`。

每个新的分片中都包含：一个 IP 首部、某些原始分组中的选项以及最多 `len` 长度的数据。

图 10-7 中的代码，以一个 C 的复合语句开始，构造了从第 2 个分片开始的分片表。在表生成后（图 10-8），原来的分组被转换成第一个分片。

```

267      {
268          int      mhlen, firstlen = len;
269          struct mbuf **mnext = &m->m_nextpkt;

270          /*
271           * Loop through length of segment after first fragment,
272           * make new header and copy data of each part and link onto chain.
273           */
274          m0 = m;
275          mhlen = sizeof(struct ip);
276          for (off = hlen + len; off < (u_short) ip->ip_len; off += len) {
277              MGETHDR(m, M_DONTWAIT, MT_HEADER);
278              if (m == 0) {
279                  error = ENOBUFS;
280                  ipstat.ips_odropped++;
281                  goto sendorfree;
282              }
283              m->m_data += max_linkhdr;
284              mhip = mtod(m, struct ip *);
285              *mhip = *ip;
286              if (hlen > sizeof(struct ip)) {
287                  mhlen = ip_optcopy(ip, mhip) + sizeof(struct ip);
288                  mhip->ip_hl = mhlen >> 2;
289              }
290              m->m_len = mhlen;
291              mhip->ip_off = ((off - hlen) >> 3) + (ip->ip_off & ~IP_MF);
292              if (ip->ip_off & IP_MF)
293                  mhip->ip_off |= IP_MF;
294              if (off + len >= (u_short) ip->ip_len)
295                  len = (u_short) ip->ip_len - off;
296              else
297                  mhip->ip_off |= IP_MF;
298              mhip->ip_len = htons((u_short) (len + mhlen));
299              m->m_next = m_copy(m0, off, len);
300              if (m->m_next == 0) {
301                  (void) m_free(m);
302                  error = ENOBUFS; /* ??? */
303                  ipstat.ips_odropped++;
304                  goto sendorfree;
305              }
306              m->m_pkthdr.len = mhlen + len;

```

ip_output.c

图 10-7 函数 `ip_output`：构造分片表

```

307         m->m_pkthdr.rcvif = (struct ifnet *) 0;
308         mhip->ip_off = htons((u_short) mhip->ip_off);
309         mhip->ip_sum = 0;
310         mhip->ip_sum = in_cksum(m, mhlen);
311         *mnext = m;
312         mnext = &m->m_nextpkt;
313         ipstat.ips_ofragments++;
314     }

```

ip_output.c

图10-7 (续)

```

315     /*
316      * Update first fragment by trimming what's been copied out
317      * and updating header, then send each fragment (in order).
318      */
319     m = m0;
320     m_adj(m, hlen + firstlen - (u_short) ip->ip_len);
321     m->m_pkthdr.len = hlen + firstlen;
322     ip->ip_len = htons((u_short) m->m_pkthdr.len);
323     ip->ip_off = htons((u_short) (ip->ip_off | IP_MF));
324     ip->ip_sum = 0;
325     ip->ip_sum = in_cksum(m, hlen);
326     sendorfree:
327     for (m = m0; m; m = m0) {
328         m0 = m->m_nextpkt;
329         m->m_nextpkt = 0;
330         if (error == 0)
331             error = (*ifp->if_output) (ifp, m,
332                                     (struct sockaddr *) dst, ro->ro_rt);
333         else
334             m_freem(m);
335     }
336     if (error == 0)
337         ipstat.ips_fragmented++;
338 }

```

ip_output.c

图10-8 函数ip_output：发送分片

267-269 外部块允许在函数中离使用点更近一点的地方定义 mhlen、firstlen 和 mnext。这些变量的范围一直到块的末尾，它们隐藏其他在块外定义的有相同名字的变量。

270-276 因为原来的缓存链现在成了第一个分片，所以 for 循环从第2个分片的偏移开始：hlen+len。对每个分片，ip_output 采取以下动作：

- 277-284 分配一个新的分组缓存，调整 m_data 指针，为一个 16 字节链路层首部 (max_linkhdr) 腾出空间。如果 ip_output 不这么做，则网络接口驱动器就必须再分配一个 mbuf 来存放链路层首部或移动数据。两种工作都很耗时，在这里就很容易避免。
- 285-290 从原来的分组中把 IP 首部和 IP 选项复制到新的分组中。前者复制在一个结构中。ip_optcopy 只复制那些将被复制到每个分片中的选项 (10.4 节)。
- 291-297 设置分片包括 MF 比特的偏移字段 (ip_off)。如果原来分组中已设置了 MF 比特，则在所有分片中都把 MF 置位。如果原来分组中没有设置 MF 比特，则除了最后一个分片外，其他所有分片中的 MF 都置位。
- 298 为分片设置长度，解决首部小一些 (ip_optcopy 可能没有复制所有选项)，以及最后一个分片的数据区小一些的问题。以网络字节序存储长度。

- 299-305 从原始分组中把数据复制到该分片中。如果必要，`m_copy`会再分配一个 `mbuf`。如果 `m_copy` 失败，则发出 `ENOBUFS`。`sendorfree` 丢弃所有已被分配的缓存。
- 306-314 调整新创建的分片的 `mbuf` 分组首部，使其具有正确的全长。把新分片的接口指针清零，把 `ip_off` 转换成网络字节序，计算新分片的检验和。通过 `m_nextpkt` 把该分片与前面的分片链接起来。

在图10-8中，`ip_output`构造了第一个分片，并把每个分片传递到接口层。

315-325 把末尾多余的数据截断后，原来的分组就被转换成第一个分片，同时设置 MF 比特，把 `ip_len` 和 `ip_off` 转换成网络字节序，计算新的检验和。在这个分片中，保留所有的 IP 选项。在目的主机重装时，只保留数据报的第一个分片的 IP 选项(图10-28)。某些选项，如源路由选项，必须被复制到每个分片中，即使在重装时都被丢弃了。

326-338 此时，`ip_output`可能有一个完整的分片表，或者已经产生了错误，都必须把生成的那部分分片表丢弃。`for` 循环遍历该表，发送分片或者由于 `error` 而丢弃分片。在发送期间遇到的所有错误都会使后面的分片被丢弃。

10.4 ip_optcopy函数

在分片过程中，`ip_optcopy`(图10-9)复制到达分组(如果分组是被转发的)或者原始数据报中(如果该数据报是在本地生成的)中的选项到外出的分片中。

```

395 int
396 ip_optcopy(ip, jp)
397 struct ip *ip, *jp;
398 {
399     u_char *cp, *dp;
400     int     opt, optlen, cnt;

401     cp = (u_char *) (ip + 1);
402     dp = (u_char *) (jp + 1);
403     cnt = (ip->ip_hl << 2) - sizeof(struct ip);
404     for (; cnt > 0; cnt -= optlen, cp += optlen) {
405         opt = cp[0];
406         if (opt == IPOPT_EOL)
407             break;
408         if (opt == IPOPT_NOP) {
409             /* Preserve for IP mcast tunnel's LSRR alignment. */
410             *dp++ = IPOPT_NOP;
411             optlen = 1;
412             continue;
413         } else
414             optlen = cp[IPOPT_OLEN];
415         /* bogus lengths should have been caught by ip_dooptions */
416         if (optlen > cnt)
417             optlen = cnt;
418         if (IPOPT_COPIED(opt)) {
419             bcopy((caddr_t) cp, (caddr_t) dp, (unsigned) optlen);
420             dp += optlen;
421         }
422     }
423     for (optlen = dp - (u_char *) (jp + 1); optlen & 0x3; optlen++)
424         *dp++ = IPOPT_EOL;
425     return (optlen);
426 }

```

ip_output.c

ip_output.c

图10-9 函数：确定分片大小

395-422 `ip_optcopy`的参数是：`ip`，一个指向外出分组的IP首部的指针；`jp`，一个指向新生成的分片的IP首部的指针；`ip_optcopy`初始化`cp`和`dp`指向每个分组的第一个选项，并在处理每个选项时把`cp`和`dp`向前移动。第一个`for`循环在每次重复时复制一个选项，当它遇到EOL选项或已经检查完所有选项时。NOP选项被复制，用来维持后继选项的对齐限制。

Net/2版本废除了NOP选项。

如果`IPOPT_COPIED`指示`copied`比特被置位，则`ip_optcopy`把选项复制到新片中。图9-5显示了哪些选项的`copied`比特是被置位的。如果某个选项的长度太大，就被截断；`ip_dooptions`应该已经发现这种错误了。

423-426 第2个`for`循环把选项表填充到4字节的边界。由于分组首部长度(`ip_hlen`)是以4字节为单位计算的，所以需要这个操作。这也保证了后面跟着的运输层首部与4字节边界对齐。这样会提高性能，因为在许多运输层协议的设计中，如果运输层首部从一个32 bit边界开始，那么32 bit首部字段将按照32 bit边界对齐。在某些机器上，CPU访问32 bit对齐的字有困难，这时，这种字节安排就提高了CPU的性能。

图10-10显示了`ip_optcopy`的运行。

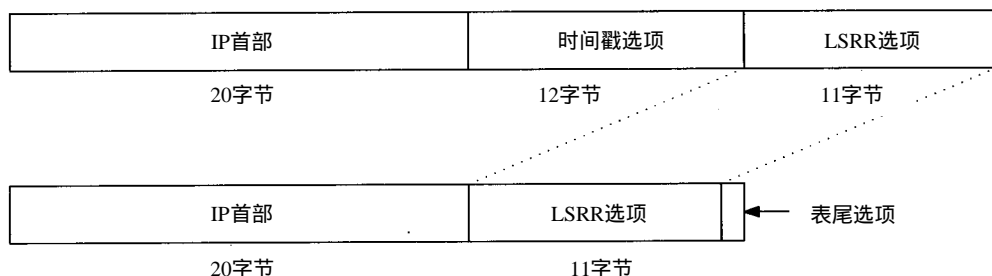


图10-10 在分片中并不复制所有选项

在图10-10中，我们看到`ip_optcopy`不复制时间戳选项(它的`copied`比特为0)，但却复制LSRR选项(它的`copied`比特为1)。为了把新选项与4字节边界对齐，`ip_optcopy`也增加了一个EOL选项。

10.5 重装

到目前为止，我们已经讨论了数据报(或片)的分片，现在再回到`ipintr`讨论重装过程。在图8-15中，我们把`ipintr`中的重装代码省略了，并推迟对它的讨论。`ipintr`可以把数据报整个地交给运输层处理。`ipintr`接收的分片被传给`ip_reass`，由它尝试把分片重装成一个完整的数据报。图10-11显示了`ipintr`的代码。

271-279 我们知道`ip_off`包含DF比特、MF比特以及分片偏移。如果MF比特或分片偏移非零，则DF就被掩盖掉了，分组就是一个必须被重装的分片。如果两者都为零，则分组就是一个完整的数据报。跳过重装代码，执行图10-11中最后的`else`语句，它从全部数据报长度中排除了首部长度的。

280-286 `m_pullup`把位于外部簇上的数据移动到`mbuf`的数据区。我们记得，如果一个缓存区无法容纳外部簇上的一个IP分组，则SLIP接口(5.3节)可能会把该分组整个返回。`m_devget`也会全部返回外部簇上的某个IP分组(2.6节)。在`mtod`宏(2.6节)开始工作之前，

m_pullup必须把IP首部从外部簇上移到mbuf的数据区中去。

```

271  ours:
272  /*
273   * If offset or IP_MF are set, must reassemble.
274   * Otherwise, nothing need be done.
275   * (We could look in the reassembly queue to see
276   * if the packet was previously fragmented,
277   * but it's not worth the time; just let them time out.)
278   */
279  if (ip->ip_off & ~IP_DF) {
280      if (m->m_flags & M_EXT) { /* XXX */
281          if ((m = m_pullup(m, sizeof(struct ip))) == 0) {
282              ipstat.ips_toosmall++;
283              goto next;
284          }
285          ip = mtod(m, struct ip *);
286      }
287      /*
288       * Look for queue of fragments
289       * of this datagram.
290       */
291      for (fp = ipq.next; fp != &ipq; fp = fp->next)
292          if (ip->ip_id == fp->ipq_id &&
293              ip->ip_src.s_addr == fp->ipq_src.s_addr &&
294              ip->ip_dst.s_addr == fp->ipq_dst.s_addr &&
295              ip->ip_p == fp->ipq_p)
296              goto found;
297      fp = 0;
298  found:
299      /*
300       * Adjust ip_len to not reflect header,
301       * set ip_mff if more fragments are expected,
302       * convert offset of this to bytes.
303       */
304      ip->ip_len -= hlen;
305      ((struct ipasfrag *) ip)->ipf_mff &= ~1;
306      if (ip->ip_off & IP_MF)
307          ((struct ipasfrag *) ip)->ipf_mff |= 1;
308      ip->ip_off <<= 3;
309      /*
310       * If datagram marked as having more fragments
311       * or if this is not the first fragment,
312       * attempt reassembly; if it succeeds, proceed.
313       */
314      if (((struct ipasfrag *) ip)->ipf_mff & 1 || ip->ip_off) {
315          ipstat.ips_fragments++;
316          ip = ip_reass((struct ipasfrag *) ip, fp);
317          if (ip == 0)
318              goto next;
319          ipstat.ips_reassembled++;
320          m = dtom(ip);
321      } else if (fp)
322          ip_freem(fp);
323  } else
324      ip->ip_len -= hlen;

```

ip_input.c

图10-11 函数ipintr：分片处理

287-297 Net/3在一个全局双向链表 `ipq`上记录不完整的数据报。这个名字可能容易产生误解，因为这个数据结构并不是一个队列。也就是说，可以在表的任何地方插入和删除，并不限制一定要在末尾。我们将用名词“表 (*list*)”来强调这个事实。

`ipintr`对表进行线性搜索，为当前分片找到合适的数据报。记住分片是由4元组{`ip_id`、`ip_src`、`ip_dst`和`ip_p`}唯一标识的。`ipq`的每个入口是一个分片表，如果 `ipintr`找到一个匹配，则`fp`指向匹配的表。

Net/3采用线性搜索来访问它的许多数据结构。尽管简单，但是当主机支持大量网络连接时，这种方法就成为瓶颈。

298-303 在`found`语句，`ipintr`为方便重装，修改了分组：

- 304 `ipintr`修改`ip_len`，从中减去标准IP首部和任何选项。我们必须牢记这一点，以免混淆对标准 `ip_len`解释的理解。标准 `ip_len`中包含了标准首部、选项和数据。如果跳过重装代码，`ip_len`也会被改变，因为这个分组不是一个分片。
- 305-307 `ipintr`把MF标志复制到 `ipf_mff`的低位，把`ip_tos`覆盖掉(&=1只清除低位)。注意，在 `ipf_mff`成为一个有效成员之前，必须把 `ip`指一个`ipasfrag`结构。10.6节和图10-14描述了 `ipasfrag`结构。

尽管RFC 1122要求IP层提供某种机制，允许运输层为每个外出的数据报设置 `ip_tos`比特。但它只推荐，在目的主机，IP层把 `ip_tos`值传给运输层。因为TOS字段的低位字节必须总是0，所以当重装算法使用 `ip_off`(通常在这里找到MF比特)时，可以得到MF比特。

现在，可以把 `ip_off`作为一个16 bit偏移，而不是3个标志比特和一个13 bit偏移来访问了。

- 308 用8乘`ip_off`，把它从以8字节为单元转换成以1字节为单元。

`ipf_mff`和`ip_off`决定 `ipintr`是否应该重装。图10-12描述了不同的情况及相应的动作，其中`fp`指向的是系统以前为该数据报接收的分片表。许多工作是由 `ip_reass`做的。

309-322 如果 `ip_reass`通过把当前分片与以前收到的分片组合在一起，能重装成一个完整的数据报，它就返回指向该重装好的数据报的指针。如果没有重装好，则 `ip_reass`保存该分片，`ipintr`跳到`next`去处理下一个分片(图8-12)。

<code>ip_off</code>	<code>ipf_mff</code>	<code>fp</code>	描 述	动 作
0	假	空	完整数据报	没有要求重装
0	假	非空	完整数据报	丢弃前面的分片
任意	真	空	新数据报的分片	用这个分片初始化新的分片表
任意	真	非空	不完整新数据报的分片	插入到已有的分片表中，尝试重装
非零	假	空	新数据报的末尾分片	初始化新的分片表
非零	假	非空	不完整新数据报的末尾分片	插入到已有的分片表中，尝试重装

图10-12 `ipintr` 和 `ip_reass` 中的IP分片处理

323-324 当到达一个完整的数据报时，就选择这个 `else`分支，并按照前面的叙述修改 `ip_hlen`。这是一个普通的流，因为收到的大多数数据报都不是分片。

如果重装处理产生一个完整的数据报，`ipintr`就把这个完整的数据报上传给合适的运输

层协议(图8-15)：

```
(*inetsw[ip_protox[ip->ip_p]].pr_input)(m,hlen);
```

10.6 ip_reass函数

ipintr把一个要处理的分片和一个指针传给 ip_reass，其中指针指向的是 ipq中匹配的重装首部。ip_reass可能重装成功并返回一个完整的数据报，可能把该分片链接到数据报的重装链表上，等待其他分片到达后重装。每个重装链表的表头是一个 ipq结构，如图10-13所示。

```
52 struct ipq {
53     struct ipq *next, *prev;    /* to other reass headers */
54     u_char ipq_ttl;            /* time for reass q to live */
55     u_char ipq_p;              /* protocol of this fragment */
56     u_short ipq_id;            /* sequence id for reassembly */
57     struct ipasfrag *ipq_next, *ipq_prev;
58     /* to ip headers of fragments */
59     struct in_addr ipq_src, ipq_dst;
60 };
```

ip_var.h

ip_var.h

图10-13 ipq 结构

52-60 用来标识一个数据报分片的四个字段，ip_id、ip_src、ip_dst和ip_p，被保存在每个重装链表表头的 ipq结构中。Net/3用next和prev构造数据报链表，用ipq_next和ipq_prev构造分片的链表。

到达分组的IP首部在被放在重装链表之前，首先被转换成一个 ipasfrag结构(图10-14)。

```
66 struct ipasfrag {
67     #if BYTE_ORDER == LITTLE_ENDIAN
68         u_char ip_hl:4,
69             ip_v:4;
70     #endif
71     #if BYTE_ORDER == BIG_ENDIAN
72         u_char ip_v:4,
73             ip_hl:4;
74     #endif
75     u_char ipf_mff;    /* XXX overlays ip_tos: use low bit
76                        * to avoid destroying tos;
77                        * copied from (ip_off&IP_MF) */
78     short ip_len;
79     u_short ip_id;
80     short ip_off;
81     u_char ip_ttl;
82     u_char ip_p;
83     u_short ip_sum;
84     struct ipasfrag *ipf_next; /* next fragment */
85     struct ipasfrag *ipf_prev; /* previous fragment */
86 };
```

ip_var.h

ip_var.h

图10-14 ipasfrag 结构

66-86 ip_reass在一个由ipf_next和ipf_prev链接起来的双向循环链表上，收集某个数据报的分片。这些指针覆盖了IP首部的源地址和目的地址。ipf_mff成员覆盖ip结构中的

ip_tos。其他成员是相同的。

图10-15显示了分片首部链表(ipq)和分片(ipasfrag)之间的关系。

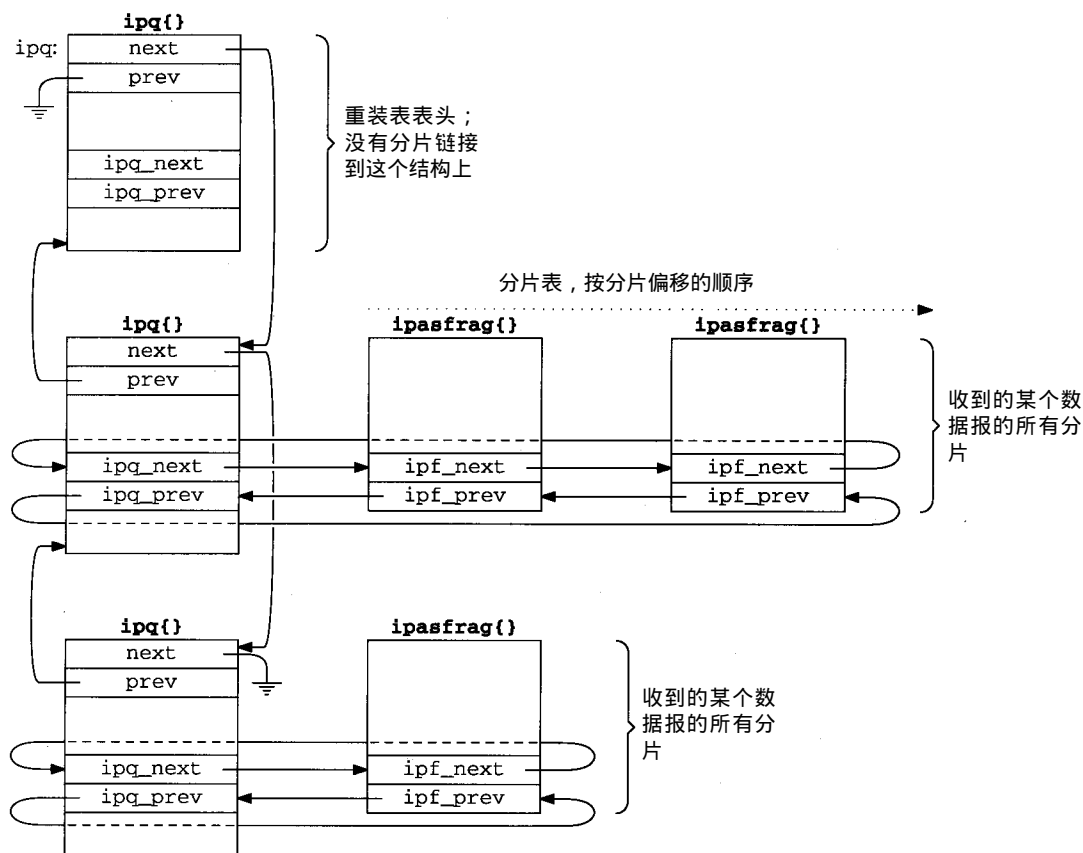


图10-15 分片首部链表 ipq 和分片

图10-15的左下部是重装首部的链表。表中第一个节点是全局 ipq 结构，ipq。它永远不会有自己的相关分片表。ipq 表是一个双向链表，用于支持快速插入和删除。next 和 prev 指针指向前一个和后一个 ipq 结构，用终止结构的角上的箭头表示。

图10-15仍然没有显示重装结构的所有复杂性。重装代码很难跟踪，因为它完全依靠把指针指向底层 mbuf 上的三个不同的结构。我们已经接触过这个技术了，例如，当一个 ip 结构覆盖某个缓存的数据部分时。

图10-16显示了 mbuf、ipq 结构、ipasfrag 结构和 ip 结构之间的关系。

图10-16中含有大量信息：

- 所有结构都放在一个 mbuf 的数据区内。
- ipq 链表由 next 和 prev 链接起来的 ipq 结构组成。每个 ipq 结构保存了唯一标识一个 IP 数据报的四个字段(图10-16中的阴影部分)。
- 当作为分片链表的头访问时，每个 ipq 结构被看成是一个 ipasfrag 结构。这些分片由 ipf_next 和 ipf_prev 链接起来，分别覆盖了 ipq 结构的 ipq_next 和 ipq_prev 成员。
- 每个 ipasfrag 结构都覆盖了到达分片的 ip 结构，与分片一起到达的数据在缓存中跟在该结构之后。ipasfrag 结构的阴影部分的成员的含义与其在 ip 结构中不太相同。

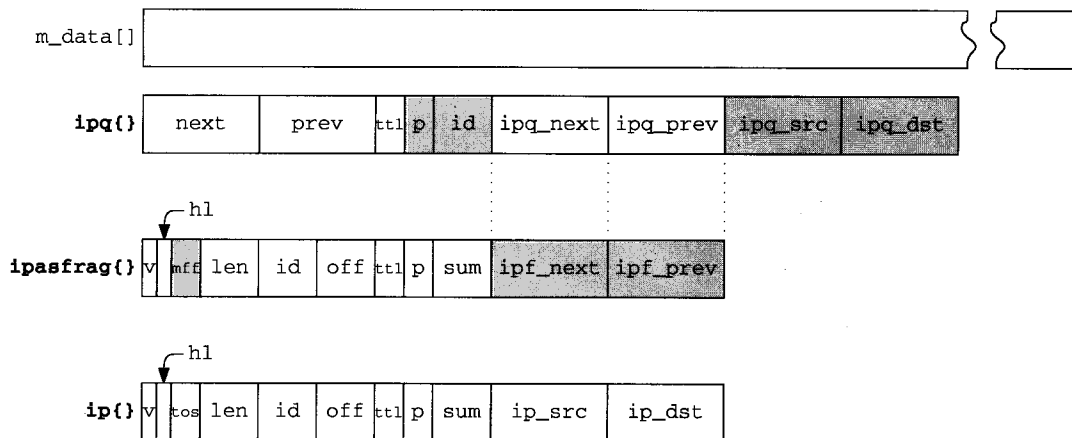


图10-16 可通过多种结构访问的一段内存区

图10-15显示了这些重装结构之间的物理连接,图10-16显示了`ip_reass`使用的覆盖技术。图10-17从逻辑的观点说明重装结构:该图显示了三个数据报的重装,以及`ipq`链表和`ipasfrag`结构之间的关系。

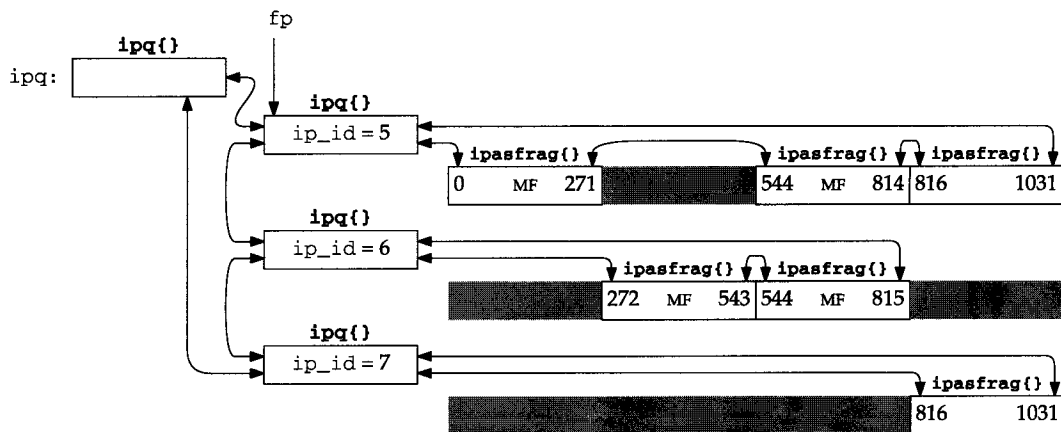


图10-17 三个IP数据报的重装

每个重装链表的表头包含原始数据报的标识符、协议、源和目的地址。图中只显示了`ip_id`字段。分片表通过偏移字段排序,如果MF比特被置位,则用MF标志分片,缺少的分片出现在阴影里。每个分片内的数字显示了该分片的开始和结束字节相对于原始数据报数据区的相对偏移,而不是相对于原始数据报的IP首部。

这个例子是用来说明三个没有IP选项的UDP数据报,其中每个数据报都有1024字节的数据。每个数据报的全长是1052(20+8+1024)字节,正好适合1500字节以太网MTU。在到目的主机的途中,这些数据报会遇到一个SLIP链路,该链路上的路由器对数据报分片,使其大小适于放在典型的296字节的SLIP MTU中。每个数据报分4个分片到达。第1个分片中包含一个标准的20字节IP首部,8字节UDP首部和264字节数据。第2个和第3个分片中包含一个20字节IP首部和272字节数据。最后一个分片中有一个20字节首部和216字节数据($1032=272 \times 3+216$)。

在图10-17中,数据报5缺少一个包含272~543字节的分片。数据报6缺少第一个分片,

0~271字节，以及最后一个从偏移816开始的分片。数据报7缺少前三个分片，0~815。

图10-18列出了ip_reass。前面讲到，当目的地是本主机的某个IP分片到达时，在处理完所有选项后，ipintr会调用ip_reass。

```

337 /*
338  * Take incoming datagram fragment and try to
339  * reassemble it into whole datagram. If a chain for
340  * reassembly of this datagram already exists, then it
341  * is given as fp; otherwise have to make a chain.
342  */
343 struct ip *
344 ip_reass(ip, fp)
345 struct ipasfrag *ip;
346 struct ipq *fp;
347 {
348     struct mbuf *m = dtom(ip);
349     struct ipasfrag *q;
350     struct mbuf *t;
351     int hlen = ip->ip_hl << 2;
352     int i, next;
353
354     /*
355      * Presence of header sizes in mbufs
356      * would confuse code below.
357      */
358     m->m_data += hlen;
359     m->m_len -= hlen;
360
361     /*
362      * If a chain of fragments already exists,
363      * then we must have a pointer to the first
364      * fragment in the chain.
365      */
366     if (fp)
367         return (fp);
368
369     /*
370      * No chain exists, so we must have a pointer
371      * to the first fragment in the chain.
372      */
373     if (ip->ip_off & 1)
374         return (0);
375
376     /*
377      * Create a new chain.
378      */
379     q = malloc(sizeof(struct ipasfrag));
380     if (q == 0)
381         return (0);
382     q->ip = ip;
383     q->next = 0;
384     fp = q;
385
386     /*
387      * If the fragment is the first in the chain,
388      * then we must have a pointer to the first
389      * fragment in the chain.
390      */
391     if (ip->ip_off & 1)
392         return (0);
393
394     /*
395      * If the fragment is the last in the chain,
396      * then we must have a pointer to the last
397      * fragment in the chain.
398      */
399     if (ip->ip_off & 0x3fff)
400         return (0);
401
402     /*
403      * If the fragment is the first in the chain,
404      * then we must have a pointer to the first
405      * fragment in the chain.
406      */
407     if (ip->ip_off & 1)
408         return (0);
409
410     /*
411      * If the fragment is the last in the chain,
412      * then we must have a pointer to the last
413      * fragment in the chain.
414      */
415     if (ip->ip_off & 0x3fff)
416         return (0);
417
418     /*
419      * If the fragment is the first in the chain,
420      * then we must have a pointer to the first
421      * fragment in the chain.
422      */
423     if (ip->ip_off & 1)
424         return (0);
425
426     /*
427      * If the fragment is the last in the chain,
428      * then we must have a pointer to the last
429      * fragment in the chain.
430      */
431     if (ip->ip_off & 0x3fff)
432         return (0);
433
434     /*
435      * If the fragment is the first in the chain,
436      * then we must have a pointer to the first
437      * fragment in the chain.
438      */
439     if (ip->ip_off & 1)
440         return (0);
441
442     /*
443      * If the fragment is the last in the chain,
444      * then we must have a pointer to the last
445      * fragment in the chain.
446      */
447     if (ip->ip_off & 0x3fff)
448         return (0);
449
450     /*
451      * If the fragment is the first in the chain,
452      * then we must have a pointer to the first
453      * fragment in the chain.
454      */
455     if (ip->ip_off & 1)
456         return (0);
457
458     /*
459      * If the fragment is the last in the chain,
460      * then we must have a pointer to the last
461      * fragment in the chain.
462      */
463     if (ip->ip_off & 0x3fff)
464         return (0);
465
466     dropfrag:
467     ipstat.ips_fragdropped++;
468     m_freem(m);
469     return (0);
470 }

```

图10-18 函数ip_reass：数据报重装

343-358 当调用ip_reass时，ip指向分片fp或者指向匹配的ipq结构或者为空。

因为重装只涉及每个分片的数据部分，所以ip_reass调整含有该分片的mbuf的m_data和m_len，减去每个分片的IP首部。

465-469 在重装过程中，如果产生错误，该函数就跳到dropfrag。dropfrag增加ips_fragdropped，丢弃该分片，并返回一个空指针。

在运输层丢弃分片通常会严重降低性能，因为必须重传整个数据报。TCP谨慎地避免分片，但是UDP应用程序必须采取步骤以避免对自己分片。[Kent和Mogul 1987]解释了为什么应该避免分片。

所有IP实现必须能够重装最多576字节的数据报。没有通用的方法来确定远程主机能重装的最大的数据报的大小。我们将在27.5节中看到TCP提供了一个机制，可以确定远程主机所能处理的最大数据报的大小。UDP没有这样的机制，所以许多基于UDP的协议(例如，RIP、TFTP、BOOTP、SNMP和DNS)，都限制在576字节左右。

我们将分7个部分显示重装代码，从图10-19开始。

```

359  /*
360  * If first fragment to arrive, create a reassembly queue.
361  */
362  if (fp == 0) {
363      if ((t = m_get(M_DONTWAIT, MT_FTABLE)) == NULL)
364          goto dropfrag;
365      fp = mtod(t, struct ipq *);
366      insque(fp, &ipq);
367      fp->ipq_ttl = IPFRAGTTL;
368      fp->ipq_p = ip->ip_p;
369      fp->ipq_id = ip->ip_id;
370      fp->ipq_next = fp->ipq_prev = (struct ipasfrag *) fp;
371      fp->ipq_src = ((struct ip *) ip)->ip_src;
372      fp->ipq_dst = ((struct ip *) ip)->ip_dst;
373      q = (struct ipasfrag *) fp;
374      goto insert;
375  }

```

ip_input.c

图10-19 函数ip_reass：创建重装表

1. 创建重装表

359-366 当fp为空时，ip_reass用新的数据报的第一个分片创建一个重装表。它分配一个mbuf来存放新表的头(一个ipq结构)，并调用insque，将该结构插入到重装表的链表中。

图10-20列出了操作数据报和分片链表的函数。

Net/3的386版本是在machdep.c文件中定义insque和remque函数的。每台机器都有自己的文件，在其中定义核心函数，通常是为提高性能。该文件也包括与机器体系结构有关的函数，包括中断处理支持、CPU和设备配置以及内存管理函数。

insque和remque的存在主要是为了维护内核执行队列。Net/3可把它们用于数据报重装链表，因为链表具有下一个和前一个两个指针，分别作为各自节点结构的前两个成员。对任何类型结构的链表，这些函数同样可以用，尽管编译器可能会发出一些警告。这也是另一个通过两个不同结构访问内存的例子。

在所有内核结构里，下一个指针通常位于前一个指针的前面（例如，图10-14）。这是因为insque和remque最早是在VAX上用insque和remque硬件指令实现的，这些指令要求前向和后向指针具有这种顺序。

分片表不是用ipasfrag结构的前两个成员链接起来的(图10-14)，所以Net/3调用ip_deq和ip_enq而不是insque和remque。

函 数	描 述
insque	紧接在prev后面插入node void insque (void *node, void* prev);
Remque	把node从表中移走 void remque (void *node);
ip_enq	紧接在分片prev后面插入分片p void ip_enq (struct ipasfrag*, struct ipasfrag *prev);
ip_deq	移走分片p void ip_deq (struct ipasfrag*);

图10-20 ip_reass 采用的队列函数

2. 重装超时

367 RFC 1122要求有生命期字段(`ipq_ttl`), 并限制Net/3等待分片以完成一个数据报的时间。这与IP首部的TTL字段是不同的, IP首部的TTL字段是为了限制分组在互联网中循环的时间。重用IP首部的TTL字段作为重装超时的原因在于, 一旦分片到达它的最终目的地, 就不再需要首部TTL。

在Net/3中, 重装超时的初始值设为60 (`IPFRAGTTL`)。因为每次内核调用`ip_slowtimo`时, `ipq_ttl`就减去1, 而内核每500 ms调用`ip_slowtimo`一次。如果系统在接收到数据报的任一分片30秒后, 还没有组装好一个完整的IP数据报, 那么系统就丢弃该IP重装链表。重装定时器在链表被创建后的第一次调用`ip_slowtimo`时开始计时。

RFC 1122推荐重装时间在60~120秒内, 并且当收到数据报的第一个分片且定时器超时, 向源主机发出一个ICMP超时差错报文。重装后, 总是丢弃其他分片的首部和选项, 并且在ICMP差错报文中必须包含出错数据报的前64 bit(或者, 如果该数据报短于8字节, 就可以少一些)。所以, 如果内核还没有接收到分片0, 它就不能发ICMP报文。

Net/3的定时器要短一些, 并且当丢弃分片时, Net/3不发送ICMP报文。要求返回数据报的前64 bit保证含有运输层首部的前端, 这样就可以把差错报文返回给发生错误的应用程序。注意, 因为这个原因, TCP和UDP有意把它们的端口号放在首部的前8个字节。

3. 数据报标识符

368-375 `ip_reass`在分配给该数据报的`ipq`结构中保存`ip_p`、`ip_id`、`ip_src`和`ip_dst`, 让`ipq_next`和`ipq_prev`指针指向该`ipq`结构(也就是说, 它用一个节点构造一个循环链表), 让`q`指向这个结构, 并跳到`insert`(图10-25), 把第一个分片`ip`插入到新的重装表中。

`ip_reass`的下一个部分(图10-21)是当`fp`不空, 并已当前表中为新的分片找到正确位置后执行的。

```
376      /*  
377      * Find a fragment which begins after this one does.  
378      */  
379      for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = q->ipf_next)  
380          if (q->ip_off > ip->ip_off)  
381              break;
```

ip_input.c

ip_input.c

图10-21 函数`ip_reass` : 在重装链表中找位置

376-381 因为`fp`不空, 所以`for`循环搜索数据报的分片链表, 找到一个偏移大于`ip_off`的分片。

在目的主机上, 分片包含的字节范围可能会相互覆盖。发生这种情况的原因是, 当一个运输层协议重传某个数据报时, 采用与原来数据报不同的路由; 而且, 分片的模式也可能不同, 这就导致在目的主机上的相互覆盖。传输协议必须能强制IP使用原来的ID字段, 确保目的主机识别出该数据报是重传的。

Net/3并不为运输层协议提供机制保证在重传的数据报中重用IP ID字段。在准备

新数据报时，ip_output通过增加全局整数ip_id来赋一个新值(图8-22)。尽管如此，Net/3系统也能从让运输层用相同ID字段重传IP数据报的系统上接收重叠的分片。

图10-22说明分片可能会以不同的方式与已经到达的分片重叠。分片是按照它们到达目的主机的顺序编号的。重装的分片在图10-22底部显示，分片的阴影部分是被丢弃的多余字节。

在下面的讨论中，早到(earlier)分片是指先前到达主机的分片。

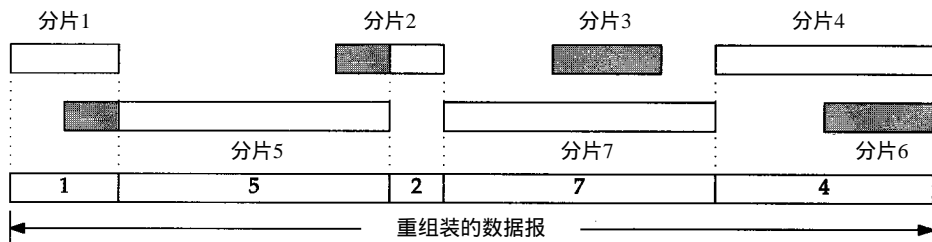


图10-22 可能会在目的主机重叠的分片的字节范围

图10-23中代码截断或丢弃到达的分片。

382-396 ip_reass把新片中与早到分片末尾重叠的字节丢弃，截断重复的部分(图10-22中分片5的前部)，或者，如果新分片的所有字节已经在早先的分片中(分片4)出现，就丢弃整个新分片(分片6)。

```

382  /*
383   * If there is a preceding fragment, it may provide some of
384   * our data already. If so, drop the data from the incoming
385   * fragment. If it provides all of our data, drop us.
386   */
387  if (q->ipf_prev != (struct ipasfrag *) fp) {
388      i = q->ipf_prev->ip_off + q->ipf_prev->ip_len - ip->ip_off;
389      if (i > 0) {
390          if (i >= ip->ip_len)
391              goto dropfrag;
392          m_adj(dtom(ip), i);
393          ip->ip_off += i;
394          ip->ip_len -= i;
395      }
396  }

```

ip_input.c

图10-23 函数ip_reass：截断到达分组

图10-24中的代码截断或丢弃已有的分片。

397-412 如果当前分片部分地与早到分片的前端部分重叠，就把早到分片中重复的数据截掉(图10-22中分片2的前部)。丢弃所有与当前分片完全重叠的早到分片(分片3)。

图10-25中，到达分片被插入到重装链表。

413-426 在截断后，ip_enq将该分片插入链表，并扫描整个链表，确定是否所有分片全部到达。如果还缺少分片，或链表最后一个分片的ipf_mff被置位，ip_reass就返回0，并等待更多的分片。

当目前的分片完成一个数据报后，整个链表被图10-26所示的代码转换成一个mbuf链。

427-440 如果某个数据报的所有分片都被接收下来，while循环用m_cat把分片重新构造

成数据报。

```

397  /*-----ip_input.c
398  * While we overlap succeeding fragments trim them or,
399  * if they are completely covered, dequeue them.
400  */
401  while (q != (struct ipasfrag *) fp && ip->ip_off+ip->ip_len > q->ip_off){
402      i = (ip->ip_off + ip->ip_len) - q->ip_off;
403      if (i < q->ip_len) {
404          q->ip_len -= i;
405          q->ip_off += i;
406          m_adj(dtom(q), i);
407          break;
408      }
409      q = q->ipf_next;
410      m_freem(dtom(q->ipf_prev));
411      ip_deq(q->ipf_prev);
412  }

```

图10-24 函数ip_reass：截断已有分组

```

413  insert:-----ip_input.c
414  /*
415  * Stick new fragment in its place;
416  * check for complete reassembly.
417  */
418  ip_enq(ip, q->ipf_prev);
419  next = 0;
420  for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = q->ipf_next) {
421      if (q->ip_off != next)
422          return (0);
423      next += q->ip_len;
424  }
425  if (q->ipf_prev->ipf_mff & 1)
426      return (0);

```

图10-25 函数ip_reass：插入分组

```

427  /*-----ip_input.c
428  * Reassembly is complete; concatenate fragments.
429  */
430  q = fp->ipq_next;
431  m = dtom(q);
432  t = m->m_next;
433  m->m_next = 0;
434  m_cat(m, t);
435  q = q->ipf_next;
436  while (q != (struct ipasfrag *) fp) {
437      t = dtom(q);
438      q = q->ipf_next;
439      m_cat(m, t);
440  }

```

图10-26 函数ip_reass：重装数据报

图10-27显示了一个有三个分片的数据报的mbuf和ipq结构之间的关系。

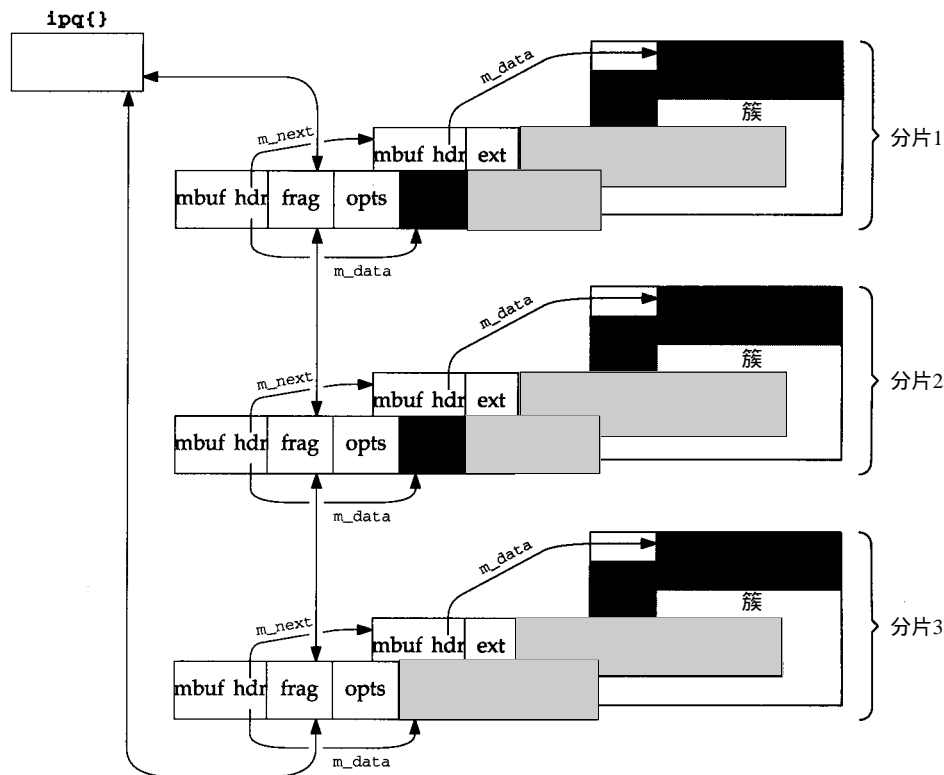


图10-27 m_cat 重装缓存内的分片

图中最暗的区域是分组的数据部分，稍淡的阴影部分是 mbuf 中未用的部分。有三个分片，每个分片都被存放在一个有两个 mbuf 的链上：一个分组首部和一个簇。每个分片的第一个缓存上的 m_data 指针指向分组数据，而不是分组的首部。因此，由 m_cat 构造的缓存链只包含分片的数据部分。

当一个分片含有多于 208 字节的数据时，情况通常是这样的 (2.6 节)。缓存的 “frag” 部分是分片的 IP 首部。由于图 10-18 中的代码，各缓存链第一个缓存的 m_data 指针指向 “opts” 之后。

图 10-28 显示了用所有分片的缓存重装的数据报。注意，分片 2 和 3 的 IP 部分和选项不在重装的数据报里。

第一个分片的首部仍然被用作 ipasfrag 结构。它被图 10-29 中的代码恢复成一个有效的 IP 数据报首部。

4. 重建数据报首部

441-456 ip_reass 把 ip 指向链表的第一个分片，将 ipasfrag 结构恢复成 ip 结构：把数据报长度恢复成 ip_len，源站地址恢复成 ip_src，目的地址恢复成 ip_dst；并把 ipf_mff 的低位清零 (从图 10-14 可以知道，ipasfrag 结构的 ipf_mff 覆盖了 ip 结构的 ipf_tos)。

ip_reass 用 remque 把整个分组从重装链表中移走，丢弃链表头 ipq 结构，调整第一个缓存中的 m_len 和 m_data，将前面被隐藏起来的第一个分片的首部和选项包含进来。

5. 计算分组长度

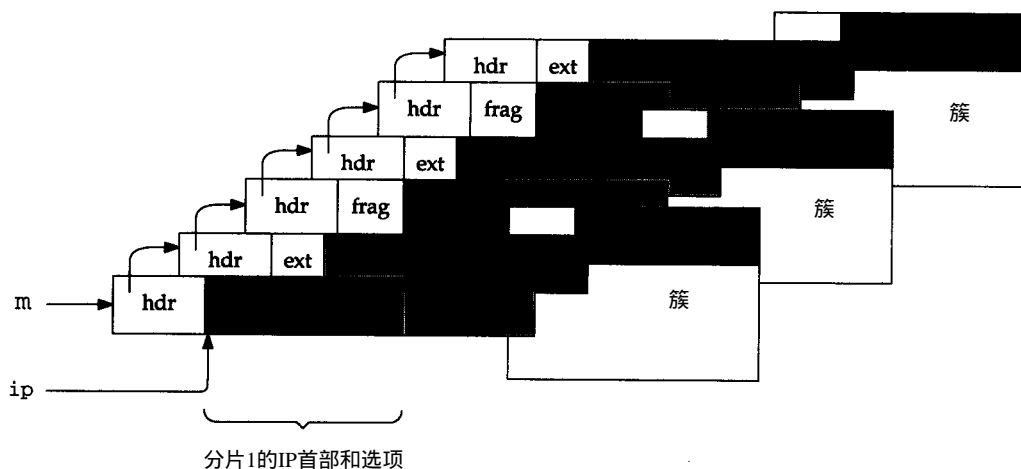


图10-28 重装的数据报

```

441      /*
442      * Create header for new ip packet by
443      * modifying header of first packet;
444      * dequeue and discard fragment reassembly header.
445      * Make header visible.
446      */
447      ip = fp->ipq_next;
448      ip->ip_len = next;
449      ip->ipf_mff &= ~1;
450      ((struct ip *) ip)->ip_src = fp->ipq_src;
451      ((struct ip *) ip)->ip_dst = fp->ipq_dst;
452      remque(fp);
453      (void) m_free(dtom(fp));
454      m = dtom(ip);
455      m->m_len += (ip->ip_hl << 2);
456      m->m_data -= (ip->ip_hl << 2);
457      /* some debugging cruft by sklower, below, will go away soon */
458      if (m->m_flags & M_PKTHDR) { /* XXX this should be done elsewhere */
459          int      plen = 0;
460          for (t = m; m; m = m->m_next)
461              plen += m->m_len;
462          t->m_pkthdr.len = plen;
463      }
464      return ((struct ip *) ip);

```

ip_input.c

图10-29 函数ip_reass：数据报重装

457-464 此处的代码总被执行，因为数据报的第一个缓存总是一个分组首部。for循环计算缓存链中数据的字节数，并把值保存在m_pkthdr.len中。

在选项类型字段中，copied比特的意义现在应该很明白了。因为目的主机只保留那些出现在第一个分片中的选项，而且只有那些在分组去往目的主机的途中控制分组处理的选项才被复制下来。不复制那些在传送过程中收集信息的选项，因为当分组在目的主机上被重装时，所有收集的信息都被丢弃了。

10.7 ip_slowtimo函数

如7.4节所述，Net/3的各项协议可能指定每500 ms调用一个函数。对IP而言，这个函数是ip_slowtimo，如图10-30所示，为重装链表上的分片计时。

```

515 void
516 ip_slowtimo(void)
517 {
518     struct ipq *fp;
519     int      s = splnet();

520     fp = ipq.next;
521     if (fp == 0) {
522         splx(s);
523         return;
524     }
525     while (fp != &ipq) {
526         --fp->ipq_ttl;
527         fp = fp->next;
528         if (fp->prev->ipq_ttl == 0) {
529             ipstat.ips_fragtimeout++;
530             ip_freef(fp->prev);
531         }
532     }
533     splx(s);
534 }

```

ip_input.c

图10-30 ip_slowtimo 函数

515-534 ip_slowtimo遍历部分数据报的链表，减少重装TTL字段。当该字段减为0时，就调用ip_freef，把与该数据报相关的分片都丢弃。在splnet处运行ip_slowtimo，避免到达分组修改链表。

ip_freef显示如图10-31。

470-486 ip_freef移走并释放链表上fp指向的各分片，然后释放链表本身。

```

474 void
475 ip_freef(fp)
476 struct ipq *fp;
477 {
478     struct ipasfrag *q, *p;

479     for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = p) {
480         p = q->ipf_next;
481         ip_deq(q);
482         m_freem(dtom(q));
483     }
484     remque(fp);
485     (void) m_free(dtom(fp));
486 }

```

ip_input.c

图10-31 ip_freef 函数

ip_drain函数

在图7-14中，我们讲到IP把ip_drain定义成一个当内核需要更多内存时才调用的函数。

这种情况通常发生在我们讨论过的(图2-13)分配缓存时。ip_drain显示如图10-32。

```
538 void
539 ip_drain()
540 {
541     while (ipq.next != &ipq) {
542         ipstat.ips_fragdropped++;
543         ip_freef(ipq.next);
544     }
545 }
```

ip_input.c

ip_input.c

图10-32 ip_drain 函数

538-545 IP释放内存的最简单办法就是丢弃重装链表上的所有IP分片。对属于某个TCP报文段的分片，TCP最终会重传该数据。属于UDP数据报的IP分片就丢失了，基于UDP的协议必须在应用程序层处理这种情况。

10.8 小结

本章展示了当一个外出的数据报过大而不适于在选定网络上传送时，ip_output如何对数据报分片。由于分片在向目的地传送的途中可能会被继续分片，也有可能走不同的路径，所以只有目的主机才能组装原来的数据报。

ip_reass接收到达分片，并试图重装数据报。如果重装成功，就把数据报传回ipintr，然后提交给恰当的运输层协议。所有IP实现必须能够重装最多576字节的数据报。Net/3的唯一限制就是可以得到的mbuf的个数。如果在一段合理的时间内，没有接收完数据报的所有分片，ip_slowtimo就丢弃不完整的数据报。

习题

- 10.1 修改ip_slowtimo，当它丢弃一个不完整数据报时(图11-1)，发出一个ICMP超时差错报文。
- 10.2 在分片的数据报中，各分片记录的路由可能互不相同。在目的主机上重装某个数据报时，返回给运输层的哪一个路由？
- 10.3 画一个图说明图10-17中ID为7的分片的ipq结构所涉及的mbuf和相关的分片链表。
- 10.4 [Auerbach 1994] 建议在对数据报分片后，应该首先传送最后的分片。如果接收系统先收到最后的分片，它就可以利用偏移值为数据报分配一个大小合适的缓冲区。请修改ip_output，首先发送最后的分片。

[Auerbach 1994] 注意到某些商用TCP/IP实现如果先收到最后的分片，就会出现崩溃。

- 10.5 用图8-5中的统计回答下面的问题。什么是每个重装的数据报的平均分片数？当一个外出的数据报被分片时，创建的平均分片数是多少？
- 10.6 如果ip_off的保留比特被置位，分组会发生什么情况？