

第2章 mbuf：存储器缓存

2.1 引言

网络协议对内核的存储器管理能力提出了很多要求。这些要求包括能方便地操作可变量缓存，能在缓存头部和尾部添加数据（如低层封装来自高层的数据），能从缓存中移去数据（如，当数据分组向上经过协议栈时要去掉首部），并能尽量减少为这些操作所做的数据复制。内核中的存储器管理调度直接关系到联网协议的性能。

在第1章我们介绍了普遍应用于Net/3内核中的存储器缓存：mbuf，它是“memory buffer”的缩写。在本章，我们要查看 mbuf和内核中用于操作它们的函数的更多的细节，在本书中几乎每一页我们都会遇到 mbuf。要理解本书的其他部分必须先要理解 mbuf。

mbuf的主要用途是保存在进程和网络接口间互相传递的用户数据。但 mbuf也用于保存其他各种数据：源与目标地址、插口选项等等。

图2-1显示了我们要遇到的四种不同类型的 mbuf，它们依据在成员 `m_flags` 中填写的不同标志 `M_PKTHDR` 和 `M_EXT` 而不同。图2-1中四个mbuf的区别从左到右罗列如下：

- 1) 如果 `m_flags` 等于0，mbuf只包含数据。在mbuf中有108字节的数据空间(`m_dat`数组)。指针 `m_data` 指向这108字节缓存中的某个位置。我们所示的 `m_data` 指向缓存的起始，但它能指向缓存中的任意位置。成员 `m_len` 指示了从 `m_data` 开始的数据的字节数。图1-6是这类mbuf的一个例子。

在图2-1中，结构 `m_hdr` 中有六个成员，它的总长是20字节。当我们查看此结构的C语言定义时，会看见前四个成员每个占用4字节而后两个成员每个占用2字节。在图2-1中我们没有区分4字节成员和2字节成员。

- 2) 第二类mbuf的 `m_flags` 值是 `M_PKTHDR`，它指示这是一个分组首部，描述一个分组数据的第一个mbuf。数据仍然保存在这个mbuf中，但是由于分组首部占用了8字节，只有100字节的数据可存储在这个mbuf中(在 `m_pktdat` 数组中)。图1-10是这种mbuf的一个例子。

成员 `m_pkthdr.len` 的值是这个分组的mbuf链表中所有数据的总长度：即所有通过 `m_next` 指针链接的mbuf的 `m_len` 值的和，如图1-8所示。输出分组没有使用成员 `m_pkthdr.rcvif`，但对于接收的分组，它包含一个指向接收接口 `ifnet` 结构(图3-6)的指针。

- 3) 下一种mbuf不包含分组首部(没有设置 `K_PKTHDR`)，但包含超过208字节的数据，这时用到一个叫“簇”的外部缓存(设置 `M_EXT`)。在此mbuf中仍然为分组首部结构分配了空间，但没有用——在图2-1中，我们用阴影显示出来。Net/3分配一个大小为1024或2048字节的簇，而不是使用多个mbuf来保存数据(第一个带有100字节数据，其余的每个带有108字节数据)。在这个mbuf中，指针 `m_data` 指向这个簇中的某个位置。

Net/3版本支持七种不同的结构。定义了四种1024字节的簇(惯例值)，三种2048字节的

簇。传统上用1024字节的原因是为了节约存储器：如果簇的大小是2048，对于以太网分组(最大1500字节)，每个簇大约有四分之一没有用。在27.5节中我们会看到Net/3 TCP发送的每个TCP报文段从来不超过一簇大小，因为当簇的大小为1024时，每个1500字节的以太网帧几乎三分之一未用。但是[Mogul 1993, 图15-15]显示了当在以太网中发送最大帧而不是1024字节的帧时能明显提高以太网的性能。这就是一种性能/存储器互换。老的系统使用1024字节簇来节约存储器，而拥有廉价存储器的新系统用2048字节的簇来提高性能。在本书中我们假定一簇的大小是2048字节。

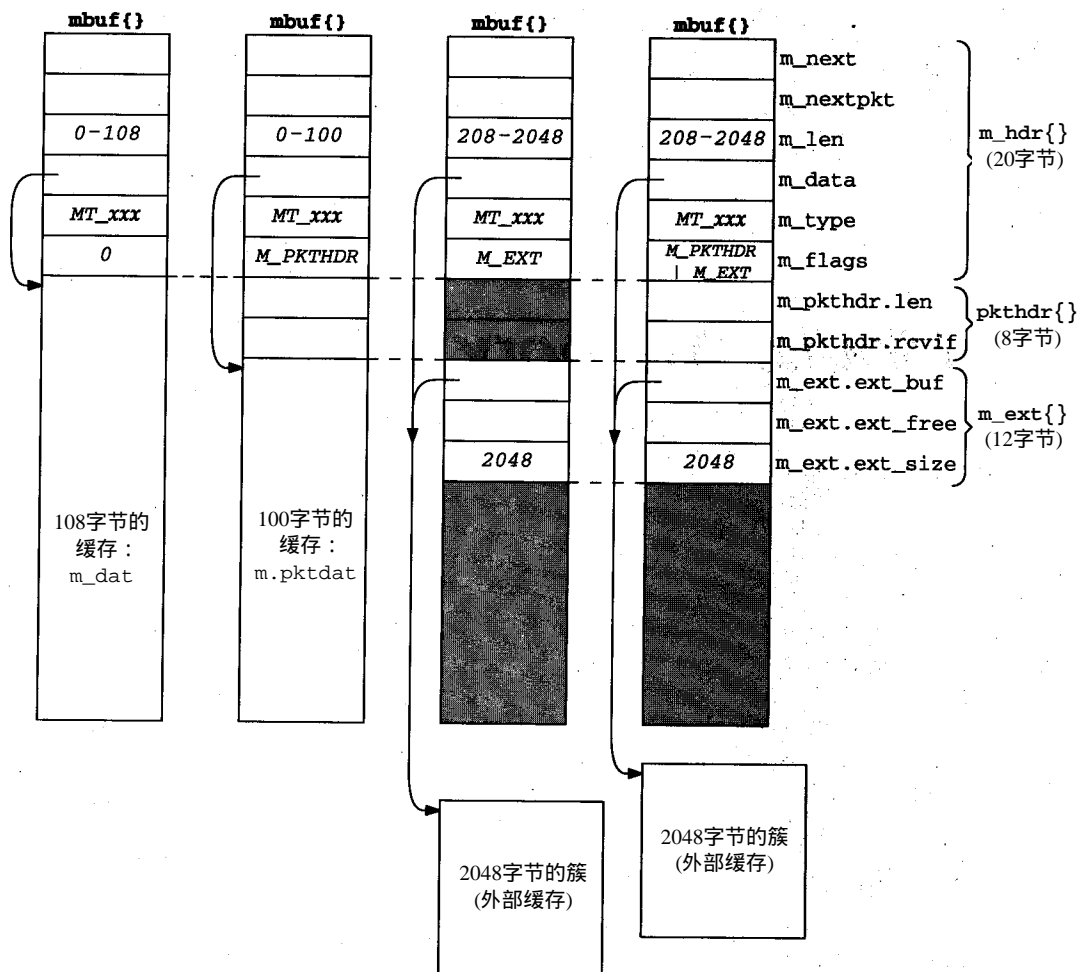


图2-1 根据不同m_flags值的四种不同类型的mbuf

不幸的是，我们所说的“簇(cluster)”用过不同的名字。常量MCLBYTES是这些缓存(1024或2048)的大小，操作这些缓存的宏的名字是MCLGET、MCLALLOC和MCLFREE。这就是为什么称它们为“簇”的原因。但我们还看到mbuf的标志是M_EXT，它代表“外部的”缓存。最后，[Leffler et al. 1989]称它们为映射页(mapped page)。这后一种称法来源于它们的实现，在2.9节我们会看到当要求一个副本时，这些簇是可以共享的。

我们可能会希望这种类型的mbuf的m_len的最小值是209而不是我们在图中所示

- 带有簇的mbuf总是包含缓存的起始地址(m_ext.ext_buf)和它的大小(m_ext.ext_size)。我们在本书采用的大小为2048。成员m_data和m_ext.ext_buf值是不同的(如我们所示),除非m_data也指向缓存的第一个字节。结构 m_ext的第三个成员 ext_free, Net/3当前未用。
- 指针m_next把mbuf链接在一起,把一个分组(记录)形成一条mbuf链表,如图1-8所示。
- 指针m_nextpkt把多个分组(记录)链接成一个mbuf链表队列。在队列中的每个分组可以是一个单独的mbuf,也可以是一个mbuf链表。每个分组的第一个mbuf包含一个分组首部。如果多个mbuf定义一个分组,只有第一个mbuf的成员m_nextpkt被使用——链表中其他mbuf的成员m_nextpkt全是空指针。

图2-2所示的是在一个队列中的两个分组的例子。它是图1-8的一个修改版。我们已经把UDP数据报放到接口输出队列中(显示出14字节的以太网首部已经添加到链表中第一个mbuf的IP首部前面),并且第二个分组已经被添加到队列中:TCP段包含1460字节的用户数据。TCP数据包含在一个簇中,并且有一个mbuf包含了它的以太网、IP与TCP首部。通过这个簇,我们可以看到指向簇的数据指针(m_data)不需要指向簇的起始位置。我们所示的队列有一个头指针和一个尾指针。这就是Net/3处理接口输出队列的方法。我们给有M_EXT标志的mbuf还添加了一个m_ext结构,并且用阴影表示这个mbuf中未用的pkthdr结构。

带有UDP数据报分组首部的第一个mbuf的类型是MT_DATA,但带有TCP报文段分组首部的第一个mbuf的类型是MT_HEADER。这是由于UDP和TCP采用了不同的方式往数据中添加首部造成的,但没有什么不同。这两种类型的mbuf本质上一致。链表中第一个mbuf的m_flags的值M_PKTHDR指示了它是一个分组首部。

仔细的读者可能会注意到我们显示一个mbuf的图(Net/3 mbuf,图2-1)与显示一个Net/1 mbuf的图[Leffler et al. 1989, p.290]的区别。这个变化是在Net/2中造成的:添加了成员m_flags,把指针m_act改名为m_nextpkt,并把这个指针移到这个mbuf的前面。

在第一个mbuf中,UDP与TCP协议首部位置的不同是由于UDP调用M_PREPEND(图23-15和习题23.1)而TCP调用MGETHDR(图26-25)造成的。

2.2 代码介绍

mbuf函数在一个单独的C文件中,并且mbuf宏与各种mbuf定义都在一个单独的头文件中,如图2-3所示。

文 件	说 明
sys/mbuf.h	mbuf结构、mbuf宏与定义
kern/uipc_mbuf.c	mbuf函数

图2-3 本章讨论的文件

2.2.1 全局变量

在本章中有一个全局变量要介绍,如图2-4所示。

变 量	数 据 类 型	说 明
mbstat	struct mbstat	mbuf的统计信息(图2-5)

图2-4 本章介绍的全局变量

2.2.2 统计

在全局结构mbstat中维护的各种统计，如图2-5所示。

mbstat成员	说 明
m_clfree	自由簇
m_clusters	从页池中获得的簇
m_drain	调用协议的drain函数来回收空间的次数
m_drops	寻找空间(未用)失败的次数
m_mbufs	从页池(未用)中获得的mbuf数
m_mtypes[256]	当前mbuf的分配数：MT_xxx索引
m_spare	剩余空间(未用)
m_wait	等待空间(未用)的次数

图2-5 在结构mbstat 中维护的mbuf统计

这个结构能被命令netstat -m检测；图2-6所示的是一些输出示例。关于所用映射页的数量的两个值是：m_clusters(34)减m_clfree(32)——当前使用的簇数(2)和m_clusters(34)。

分配给网络的存储器的千字节数是 mbuf存储器(99 × 128字节)加上簇存储器(34 × 2048字节)再除以1024。使用百分比是mbuf存储器(99 × 128字节)加上所用簇的存储器(2 × 2048字节)除以网络存储器总数(80千字节)，再乘100。

netstat -m output	mbstat member
99 mbufs in use:	
1 mbufs allocated to data	m_mtypes[MT_DATA]
43 mbufs allocated to packet headers	m_mtypes[MT_HEADER]
17 mbufs allocated to protocol control blocks	m_mtypes[MT_PCB]
20 mbufs allocated to socket names and addresses	m_mtypes[MT_SONAME]
18 mbufs allocated to socket options	m_mtypes[MT_SOOPTS]
2/34 mapped pages in use	(see text)
80 Kbytes allocated to network (20% in use)	(see text)
0 requests for memory denied	m_drops
0 requests for memory delayed	m_wait
0 calls to protocol drain routines	m_drain

图2-6 mbuf统计例子

2.2.3 内核统计

mbuf统计显示了在Net/3源代码中的一种通用技术。内核在一个全局变量(在本例中是结构mbstat)中保持对某些统计信息的跟踪。当内核在运行时，一个进程(在本例中是netstat程序)可以检查这些统计。

不是提供系统调用来获取由内核维护的统计，而是进程通过读取链接编辑器在内核建立时保存的信息来获得所关心的数据结构在内核中的地址。然后进程调用函数kvm(3)，通过使用特殊文件/dev/mem读取在内核存储器中的相应位置。如果内核数据结构从一个版本改变

为下一版本，任何读取这个结构的程序也必须改变。

2.3 mbuf的定义

处理mbuf时，我们会反复遇到几个常量。它们的值显示在图 2-7中。除了MCLBYTES定义在文件/usr/include/machine/param.h中外，其他所有常量都定义在文件mbuf.h中。

常 量	值(字节数)	说 明
<i>MCLBYTES</i>	2048	一个mbuf簇(外部缓存)的大小
<i>MHLEN</i>	100	带分组首部的mbuf的最大数据量
<i>MINCLSIZE</i>	208	存储到簇中的最小数据量
<i>MLEN</i>	108	在正常mbuf中的最大数据量
<i>MSIZE</i>	128	每个mbuf的大小

图2-7 mbuf.h 中的mbuf常量

2.4 mbuf结构

图2-8所示的是mbuf结构的定义。

```

60 /* header at beginning of each mbuf: */
61 struct m_hdr {
62     struct mbuf *mh_next;          /* next buffer in chain */
63     struct mbuf *mh_nextpkt;       /* next chain in queue/record */
64     int mh_len;                    /* amount of data in this mbuf */
65     caddr_t mh_data;               /* pointer to data */
66     short mh_type;                  /* type of data (Figure 2.10) */
67     short mh_flags;                 /* flags (Figure 2.9) */
68 };

69 /* record/packet header in first mbuf of chain; valid if M_PKTHDR set */
70 struct pkthdr {
71     int len;                        /* total packet length */
72     struct ifnet *rcvif;            /* receive interface */
73 };

74 /* description of external storage mapped into mbuf, valid if M_EXT set */
75 struct m_ext {
76     caddr_t ext_buf;                /* start of buffer */
77     void (*ext_free) ();            /* free routine if not the usual */
78     u_int ext_size;                 /* size of buffer, for ext_free */
79 };

80 struct mbuf {
81     struct m_hdr m_hdr;
82     union {
83         struct {
84             struct pkthdr MH_pkthdr; /* M_PKTHDR set */
85             union {
86                 struct m_ext MH_ext; /* M_EXT set */
87                 char MH_databuf[MHLEN];
88             } MH_dat;
89         } MH;
90         char M_databuf[MLEN]; /* !M_PKTHDR, !M_EXT */
91     } M_dat;
92 };

```

图2-8 mbuf 结构

```

93 #define m_next      m_hdr.mh_next
94 #define m_len       m_hdr.mh_len
95 #define m_data       m_hdr.mh_data
96 #define m_type       m_hdr.mh_type
97 #define m_flags      m_hdr.mh_flags
98 #define m_nextpkt    m_hdr.mh_nextpkt
99 #define m_act        m_nextpkt
100 #define m_pkthdr     M_dat.MH.MH_pkthdr
101 #define m_ext        M_dat.MH.MH_dat.MH_ext
102 #define m_pktdata    M_dat.MH.MH_dat.MH_databuf
103 #define m_dat        M_dat.M_databuf

```

mbuf.h

图2-8 (续)

结构mbuf是用一个m_hdr结构跟着一个联合来定义的。如注释所示，联合的内容依赖于标志M_PKTHDR和M_EXT。

93-103 这11个#define语句简化了对mbuf结构中的结构与联合的成员的访问。我们会看到这种技术普遍应用于Net/3源代码中，只要是一个结构包含其他结构或联合这种情况。

我们在前面说明了在结构 mbuf中前两个成员的目的：指针 m_next把mbuf链接成一个mbuf链表，而指针m_nextpkt把mbuf链表链接成一个mbuf队列。

图1-8显示了每个mbuf的成员m_len与分组首部中的成员 m_pkthdr.len的区别。后者是链表中所有mbuf的成员m_len的和。

图2-9所示的是成员m_flags的五个独立的值。

m_flags	说 明
M_BCAST	作为链路层广播发送 / 接收
M_EOR	记录结束
M_EXT	此mbuf带有簇 (外部缓存)
M_MCAST	作为链路层多播发送 / 接收
M_PKTHDR	形成一个分组 (记录) 的第一个mbuf
M_COPYFLAGS	M_PKTHDR/M_EOR/M_BCAST/M_MCAST

图2-9 m_flags 值

我们已经说明了标志 M_EXT和M_PKTHDR。M_EOR在一个包含记录尾的 mbuf中设置。Internet协议(例如TCP)从来不设置这个标志，因为TCP提供一个无记录边界的字节流服务。但是OSI与XNS运输层要用这个标志。在插口层我们会遇到这个标志，因为这一层是协议无关的，并且它要处理来自或发往所有运输层的数据。

当要往一个链路层广播地址或多播地址发送分组，或者要从一个链路层广播地址或多播地址接收一个分组时，在这个 mbuf中要设置接下来的两个标志 M_BCAST和M_MCAST。这两个常量是协议层与接口层之间的标志(图1-3)。

对于最后一个标志值 M_COPYFLAGS，当一个mbuf包含一个分组首部的副本时，这个标志表明这些标志是复制的。

图2-10所示的常量MT_xxx用于成员m_type，指示存储在mbuf中的数据的类型。虽然我们总认为一个mbuf是用来存放要发送或接收的用户数据，但 mbuf可以存储各种不同的数据结构。回忆图1-6中的一个mbuf被用来存放一个插口地址结构，其中的目标地址用于系统调用 sendto。它的m_type成员被设置为MT_SONAME。

不是图2-10中所有的mbuf类型值都用于Net/3。有些已不再使用(MT_HTABLE)，还有一些不用于TCP/IP代码中，但用于内核的其他地方。例如，MT_OOBDATA用于OSI和XNS协议，但是TCP用不同方法来处理带外(out-of-band)数据(我们在29.7节说明)。当我们在本书的后面遇到其他mbuf类型时会说明它们的用法。

Mbuf m_type	用于Net/3 TCP/IP代码	说 明	存储类型
MT_CONTROL	•	外部数据协议报文	M_MBUF
MT_DATA	•	动态数据分配	M_MBUF
MT_FREE		应在自由列表中	M_FREE
MT_FTABLE	•	分片重组首部	M_FTABLE
MT_HEADER	•	分组首部	M_MBUF
MT_HTABLE		IMP主机表	M_HTABLE
MT_IFADDR		接口地址	M_IFADDR
MT_OOBDATA		加速(带外)数据	M_MBUF
MT_PCB		协议控制块	M_PCB
MT_RIGHTS		访问权限	M_MBUF
MT_RTABLE		路由表	M_RTABLE
MT_SONAME	•	插口名称	M_MBUF
MT_SOOPTS	•	插口选项	M_SOOPTS
MT_SOCKET		插口结构	M_SOCKET

图2-10 成员m_type 的值

本图的最后一列所示的M_xxx值与内核为不同类型mbuf分配的存储器片有关。这里有大约60个可能的M_xxx值指派给由内核函数 malloc和宏MALLOC分配的不同类型的存储器空间。图2-6所示的是来源于命令 netstat -m的mbuf分配统计信息，它包括每种MT_xxx类型的统计。命令vmstat -显示了内核的存储分配统计，包括每个M_xxx类型的统计。

由于mbuf有一个固定长度(128字节)，因此对于mbuf的使用有一个限制——包含的数据不能超过108字节。Net/3用一个mbuf来存储一个TCP协议控制块(在第24章我们会涉及到)，这个mbuf的类型为MT_PCB。但是4.4BSD把这个结构的大小从108字节增加到140字节，并为这个结构使用一种不同的内核存储器分配类型。

仔细的读者会注意到图2-10中我们表明未使用MT_PCB类型的mbuf，而图2-6显示这个类型的计数不为零。Unix域协议使用这种类型的mbuf，并且mbuf的统计功能用于所有协议，而不只是Internet协议，记住这一点很重要。

2.5 简单的mbuf宏和函数

有超过两打的宏和函数来处理 mbuf(分配一个mbuf，释放一个mbuf，等等)。让我们来查看几个宏与函数的源代码，看看它们是如何实现的。

有些操作既提供了宏也提供了函数。宏版本的名称是以 M开头的大写字母名称，而函数是以m_开始的小写字母名称。两者的区别是一种典型的时间-空间互换。宏版本在每个被用到的地方都被C预处理器展开(要求更多的代码空间)，但是它在执行时更快，因为它不需要执行函数调用(对于有些体系结构，这是费时的)。而对于函数版本，它在每个被调用的地方变成了一些指令(参数压栈，调用函数等)，要求较少的代码空间，但会花费更多的执行时间。

2.5.1 m_get函数

让我们先看一下图2-11中分配mbuf的函数：`m_get`。这个函数仅仅就是宏MGET的展开。

```

134 struct mbuf *
135 m_get(nowait, type)
136 int      nowait, type;
137 {
138     struct mbuf *m;
139     MGET(m, nowait, type);
140     return (m);
141 }

```

uipc_mbuf.c

图2-11 `m_get` 函数：分配一个mbuf

注意，Net/3代码不使用ANSI C参数声明。但是，如果使用一个ANSI C编译器，所有Net/3系统头文件为所有的内核函数都提供了ANSI C函数原型。例如，`<sys/mbuf.h>`头文件中包含这样的行：

```
struct mbuf *m_get(int, int);
```

这些函数原型为所有内核函数的调用提供编译期间的参数与返回值的检查。

这个调用表明参数`nowait`的值为`M_WAIT`或`M_DONTWAIT`，它取决于在存储器不可用时是否要求等待。例如，当插口层请求分配一个mbuf来存储`sendto`系统调用(图1-6)的目标地址时，它指定`M_WAIT`，因为在此阻塞是没有问题的。但是当以太网设备驱动程序请求分配一个mbuf来存储一个接收的帧时(图1-10)，它指定`M_DONTWAIT`，因为它是作为一个设备中断处理来执行的，不能进入睡眠状态来等待一个mbuf。在这种情况下，若存储器不可用，设备驱动程序丢弃这个帧比较好。

2.5.2 MGET宏

图2-12所示的是MGET宏。调用MGET来分配存储`sendto`系统调用(图1-6)的目标地址的mbuf如下所示：

```

MGET(m, M_WAIT, MT_SONAME);
if (m == NULL)
    return(ENOBUFS);

```

```

154 #define MGET(m, how, type) { \
155     MALLOC((m), struct mbuf *, MSIZE, mtypes[type], (how)); \
156     if (m) { \
157         (m)->m_type = (type); \
158         MBUFLOCK(mbstat.m_mtypes[type]++); \
159         (m)->m_next = (struct mbuf *)NULL; \
160         (m)->m_nextpkt = (struct mbuf *)NULL; \
161         (m)->m_data = (m)->m_dat; \
162         (m)->m_flags = 0; \
163     } else \
164         (m) = m_retry((how), (type)); \
165 }

```

mbuf.h

图2-12 MGET 宏

虽然调用指定了M_WAIT，但返回值仍然要检查，因为，如图2-13所示，等待一个mbuf并不保证它是可用的。

154-157 MGET一开始调用内核宏 MALLOC，它是通用内核存储器分配器进行的。数组mbtypes把mbuf的MT_XXX值转换成相应的M_XXX值(图2-10)。若存储器被分配，成员m_type被设置为参数中的值。

158 用于跟踪统计每种mbuf类型的内核结构加1(mbstat)。当执行这句时，宏MBUFLOCK把它作为参数来改变处理器优先级(图1-13)，然后把优先级恢复为原值。这防止在执行语句mbstat.mtypes[type]++；时被网络设备中断，因为mbuf可能在内核中的各层中被分配。考虑这样一个系统，它用三步来实现一个C中的++运算：(1)把当前值装入一个寄存器；(2)寄存器加1；(3)把寄存器值存入存储器。假设计数器值为77并且MGET在插口层执行。假设执行了步骤1和2(寄存器值为78)，并且一个设备中断发生。若设备驱动也执行MGET来获得同种类型的mbuf，在存储器中取值(77)，加1(78)，并存回在存储器。当被中断执行的MGET的步骤3继续执行时，它将寄存器的值(78)存入存储器。但是计数器应为79，而不是78，这样计数器就被破坏了。

159-160 两个mbuf指针，m_next和m_nextpkt，被设置为空指针。若必要，由调用者把这个mbuf加入到一个链或队列。

161-162 最后，数据指针被设置为指向108字节的mbuf缓存的起始，而标志被设置为0。

163-164 若内核的存储器分配调用失败，调用m_retry(图2-13)。第一个参数是M_WAIT或M_DONTWAIT。

2.5.3 m_retry函数

图2-13所示的是m_retry函数。

```

92 struct mbuf *
93 m_retry(i, t)
94 int i, t;
95 {
96     struct mbuf *m;
97     m_reclaim();
98 #define m_retry(i, t) (struct mbuf *)0
99     MGET(m, i, t);
100 #undef m_retry
101     return (m);
102 }

```

uipc_mbuf.c

uipc_mbuf.c

图2-13 m_retry 函数

92-97 被m_retry调用的第一个函数是m_reclaim。在7.4节我们会看到每个协议都能定义一个“drain”函数，在系统缺乏可用存储器时能被m_reclaim调用。在图10-32中我们还会发现当IP的drain函数被调用时，所有等待重新组成IP数据报的IP分片被丢弃。TCP的drain函数什么都不做，而UDP甚至就没有定义一个drain函数。

98-102 因为在调用了m_reclaim后有可能有机会得到更多的存储器，因此再次调用宏MGET，试图获得mbuf。在展开宏MGET(图2-12)之前，m_retry被定义为一个空指针。这可以防止当存储器仍然不可用时的无休止的循环：这个MGET展开会把m设置为空指针而不是调用m_retry函数。在MGET展开以后，这个m_retry的临时定义就被取消了，以防在此之后

有对MGET的其他引用。

2.5.4 mbuf锁

在本节中我们所讨论的函数和宏并不调用 spl函数，而是调用图2-12中的MBUFLOCK来保护这些函数和宏不被中断。但在宏 MALLOC的开始包含一个 splimp，在结束时有一个 splx。宏MFREE中包含同样的保护机制。由于 mbuf在内核的所有层中被分配和释放，因此内核必须保护那些用于存储器分配的数据结构。

另外，用于分配和释放 mbuf簇的宏MCLALLOC与MCLFREE要用一个 splimp和一个 splx包括起来，因为它们修改的是一个可用簇链。

因为存储器分配与释放及簇分配与释放的宏被保护起来防止被中断，我们通常在 MGET和 m_get这样的函数和宏的前后不再调用 spl函数。

2.6 m_devget和m_pullup函数

我们在讨论IP、ICMP、IGMP、UDP和TCP的代码时会遇到函数 m_pullup。它用来保证指定数目的字节（相应协议首部的大小）在链表的第一个 mbuf中紧挨着存放；即这些指定数目的字节被复制到一个新的 mbuf并紧挨着存放。为了理解 m_pullup的用法，必须查看它的实现及相关的函数 m_devget和宏mtod与dtom。在分析这些问题的同时我们还可以再次领会 Net/3中mbuf的用法。

2.6.1 m_devget函数

当接收到一个以太网帧时，设备驱动程序调用函数 m_devget来创建一个mbuf链表，并把设备中的帧复制到这个链表中。根据所接收的帧的长度（不包括以太网首部），可能导致4种不同的mbuf链表。图2-14所示的是前两种。

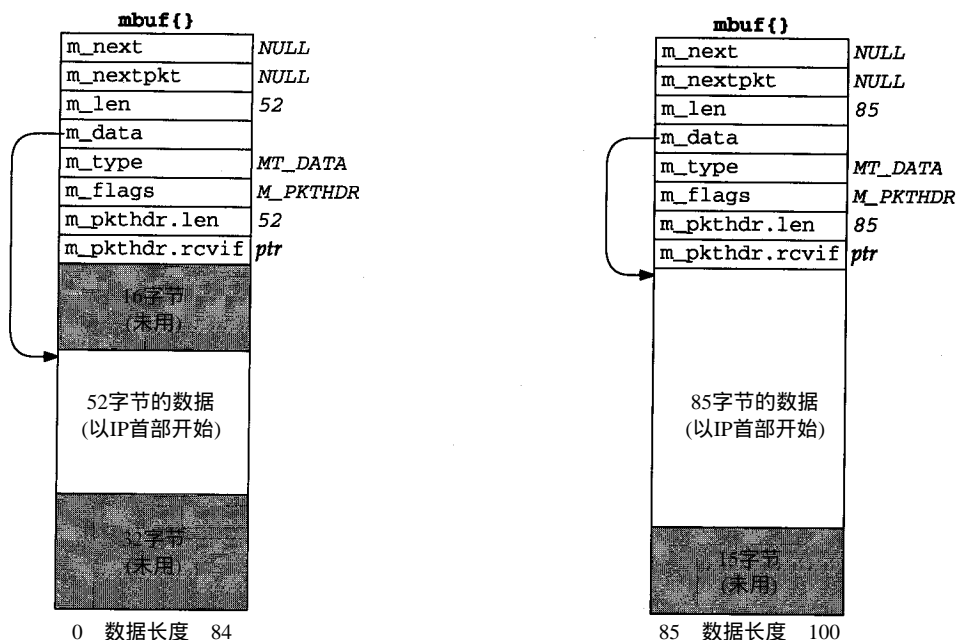


图2-14 m_devget 创建的前两种类型的mbuf

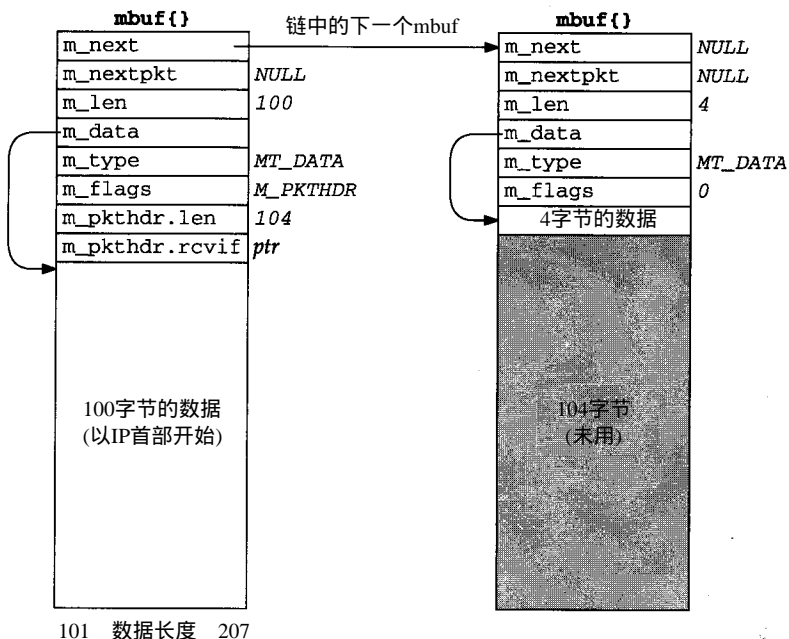


图2-15 m_devget 创建的第3种mbuf

1) 图2-14左边的mbuf用于数据的长度在0~84字节之间的情况。在这个图中，我们假定有52字节的数据：一个20字节的IP首部和一个32字节的TCP首部(标准的20字节的TCP首部加上12字节的TCP选项)，但不包括TCP数据。既然m_devget返回的mbuf数据从IP首部开始，m_len的实际最小值是28：20字节的IP首部，8字节的UDP首部和一个0长度的UDP数据报。m_devget在这个mbuf的开始保留了16字节未用。虽然14字节的以太网首部不存放在这里，还是分配了一个14字节的用于输出的以太网首部，这是同一个mbuf，用于输出。我们会遇到两个函数：icmp_reflect和tcp_respond，它们通过把接收到的mbuf作为输出mbuf来产生一个应答。在这两种情况中，接收的数据报应该少于84字节，因此很容易在前面保留16字节的空间，这样在建立输出数据报时可以节省时间。分配16字节而不是14字节的原因是为了在mbuf中用长字对准方式存储IP首部。

2) 如果数据在85~100字节之间，就仍然存放在一个分组首部mbuf中，但在开始没有16字节

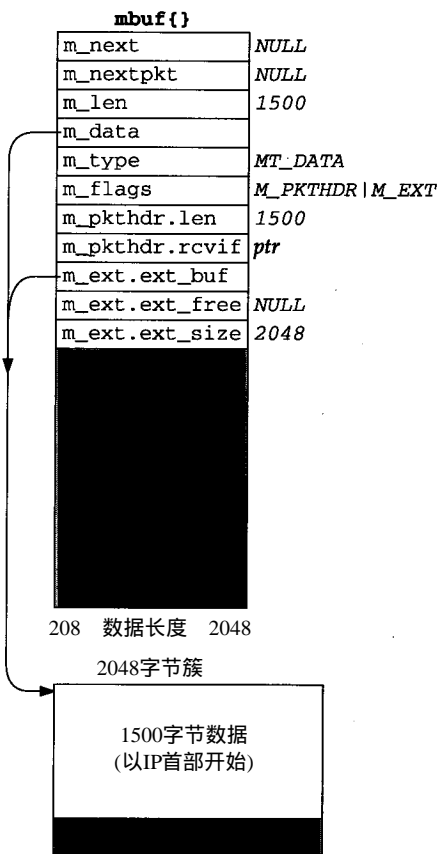


图2-16 m_devget 创建的第4种mbuf

的空间。数据存储在数组 `m_pktdat` 的开始，并且任何未用的空间放在这个数组的后面。例如在图2-14的右边的mbuf显示的就是这个例子，假设有85字节数据。

- 3) 图2-15所示的是 `m_devget` 创建的第三种mbuf。当数据在101~207字节之间时，要求有两个mbuf。前100字节存放在第一个mbuf中(有分组首部的mbuf)，而剩下的存放在第二个mbuf中。在此例中，我们显示的是一个104字节的数据报。在第一个mbuf的开始没有保留16字节的空间。
- 4) 图2-16所示的是 `m_devget` 创建的第四种mbuf。如果数据超过或等于208字节(MINCLBYTES)，要用一个或多个簇。图中的例子假设了一个1500字节的以太网帧。如果使用1024字节的簇，本例子需要两个mbuf，每个mbuf都有标志 `M_EXT`，和指向一个簇的指针。

2.6.2 mtod和dtom宏

宏 `mtod` 和 `dtom` 也定义在文件 `mbuf.h` 中。它们简化了复杂的mbuf结构表达式。

```
#define mtod(m,t) ((t)((m)->m_data))
#define dtom(x) ((struct mbuf *)((int)(x) & ~(MSIZE-1)))
```

`mtod(“mbuf到数据”)` 返回一个指向mbuf数据的指针，并把指针声名为指定类型。例如代码

```
struct mbuf *m;
struct ip *ip;

ip = mtod(m, struct ip *);
ip->ip_v = IPVERSION;
```

存储在mbuf的数据(`m_data`)指针 `ip` 中。C编译器要求进行类型转换，然后代码用指针 `ip` 引用IP首部。我们可以看到当一个C结构(通常是一个协议首部)存储在一个mbuf中时会用到这个宏。当数据存放在mbuf本身(图2-14和图2-15)或存放在一个簇中(图2-16)时，可以用这个宏。

宏 `dtom(“数据到mbuf”)` 取得一个存放在一个mbuf中任意位置的数据的指针，并返回这个mbuf结构本身的一个指针。例如，若我们知道 `ip` 指向一个mbuf的数据区，下面的语句序列

```
struct mbuf *m;
struct ip *ip;

m = dtom(ip);
```

把指向这个mbuf开始的指针存放到 `m` 中。我们知道 `MSIZE(128)` 是2的幂，并且内核存储器分配器总是为mbuf分配连续的 `MSIZE` 字节的存储块，`dtom` 仅仅是清除参数中指针的低位来发现这个mbuf的起始位置。

宏 `dtom` 有一个问题：当它的参数指向一个簇，或在一个簇内，如图2-16时，它不能正确执行。因为那里没有指针从簇内指回mbuf结构，`dtom` 不能被使用。这导致了下一个函数：`m_pullup`。

2.6.3 m_pullup函数和连续的协议首部

函数 `m_pullup` 有两个目的。第一个是当一个协议(IP、ICMP、IGMP、UDP或TCP)发现在第一个mbuf的数据量(`m_len`)小于协议首部的最小长度(例如：IP是20，UDP是8，TCP是20)时。调用 `m_pullup` 是基于假定协议首部的剩余部分存放在链表中的下一个mbuf。

`m_pullup`重新安排mbuf链表，使得前 N 字节的数据被连续地存放在链表的第一个mbuf中。 N 是这个函数的一个参数，它必须小于或等于100(MHLEN)。如果前 N 字节连续存放在第一个mbuf中，则可以使用宏`mtod`和`dtom`。

例如，我们在IP输入例程中会遇到下面这样的代码：

```
if (m->m_len < sizeof(struct ip) &&
    (m = m_pullup(m, sizeof(struct ip))) == 0) {
    ipstat.ips_toosmall++;
    goto next;
}
ip = mtod(m, struct ip *);
```

如果第一个mbuf中的数据少于20(标准IP首部的大小)，`m_pullup`被调用。函数`m_pullup`有两个原因会失败：(1)如果它需要其他mbuf并且调用`MGET`失败；或者(2)如果整个mbuf链表中的数据总数少于要求的连续字节数(即我们所说的 N ，在本例中是20)。通常，失败是因为第二个原因。在此例中，如果`m_pullup`失败，一个IP计数器加1，并且此IP数据报被丢弃。注意，这段代码假设失败的原因是mbuf链表中数据少于20字节。

实际上，在这种情况下，`m_pullup`很少能被调用(注意，C语言的`&&`操作符仅当mbuf长度小于期待值时才调用它)，并且当它被调用时，它通常会失败。通过查看图2-14~图2-16，我们可以找到它的原因：在第一个mbuf中，或在簇中，从IP首部开始有至少100字节的连续字节。这允许60字节的最大IP首部，并且后面跟着40字节的TCP首部(其他协议——ICMP，IGMP和UDP——它们的协议首部不到40字节)。如果mbuf链表中的数据可用(分组不小于协议要求的最小值)，则所要求的字节数总能连续地存放在第一个mbuf中。但是，如果接收的分组太小(`m_len`小于期待的最小值)，则`m_pullup`被调用，并且它返回一个差错，因为在mbuf链表中没有所要求数目的可用数据。

源于伯克利的内核维护一个叫`MPFail`的变量，每次`m_pullup`失败时，它都加1。在一个Net/3系统中曾经接收了超过2700万的IP数据报，而`MPFail`只有9。计数器`ipstat.ips_toosmall`也是9，并且所有其他协议计数器(ICMP、IGMP、UDP和TCP等)所计的`m_pullup`失败次数为0。这证实了我们的断言：大多数`m_pullup`的失败是因为接收的IP数据报太小。

2.6.4 `m_pullup`和IP的分片与重组

使用`m_pullup`的第二个用途涉及到IP和TCP的重组。假定IP接收到一个长度为296的分组，这个分组是一个大的IP数据报的一个分片。这个从设备驱动程序传到IP输入的mbuf看起来像我们在图2-16中所示的一个mbuf：296字节的数据存放在一个簇中。我们将这显示在图2-17中。

问题在于，IP的分片算法将各分片都存放在一个双向链表中，使用IP首部中的源与目标IP地址来存放向前与向后链表指针(当然，这两个IP地址要保存在这个链表的表头中，因为它们还要放回到重组的数据报中。我们在第10章讨论这个问题)。但是如果这个IP首部在一个簇中，如图2-17所示，这些链表指针会存放在这个簇中，并且当以后遍历链表时，指向IP首部的指针(即指向这个簇的起始的指针)不能被转换成指向mbuf的指针。这是我们在本节前面提到的问题：如果`m_data`指向一个簇时不能使用宏`dtom`，因为没有从簇指回mbuf的指针。IP分片

不能如图2-17所示的把链指针存储在簇中。

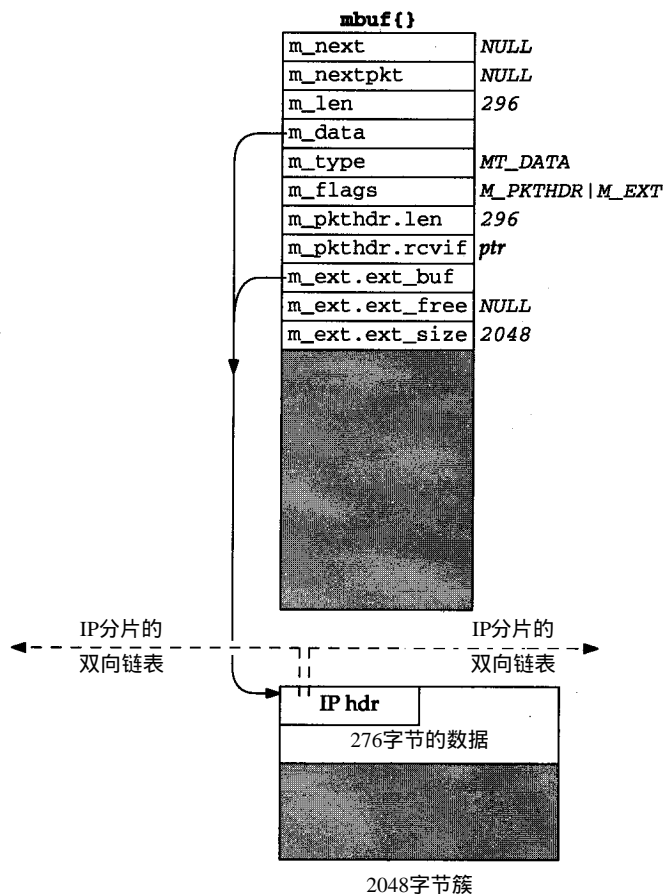


图2-17 一个长度为296的IP分片

为解决这个问题，当接收到一个分片时，若分片存放在一个簇中，IP分片例程总是调用 `m_pullup`。它强行将20字节的IP首部放到它自己的mbuf中。代码如下：

```
if (m->m_flags & M_EXT) {
    if ((m = m_pullup(m, sizeof(struct ip))) == 0) {
        ipstat.ips_toosmall++;
        goto next;
    }
    ip = mtoad(m, struct ip *);
}
```

图2-18所示的是在调用了 `m_pullup` 后得到的mbuf链表。 `m_pullup` 分配了一个新的mbuf，挂在链表的前面，并从簇中取走40字节放入到这个新的mbuf中。之所以取40字节而不是仅要求的20字节，是为了保证以后在IP把数据报传给一个高层协议（例如：ICMP，IGMP，UDP或TCP）时，高层协议能正确处理。采用不可思议的40（图7-17中的 `max_protohdr`）是因为最大协议首部通常是一个20字节的IP首部和20字节的TCP首部的组合（这假设其他协议族，例如OSI协议，并不编译到内核中）。

在图2-18中，IP分片算法在左边的mbuf中保存了一个指向IP首部的指针，并且可以用 `dtom` 将这个指针转换成一个指向mbuf本身的指针。

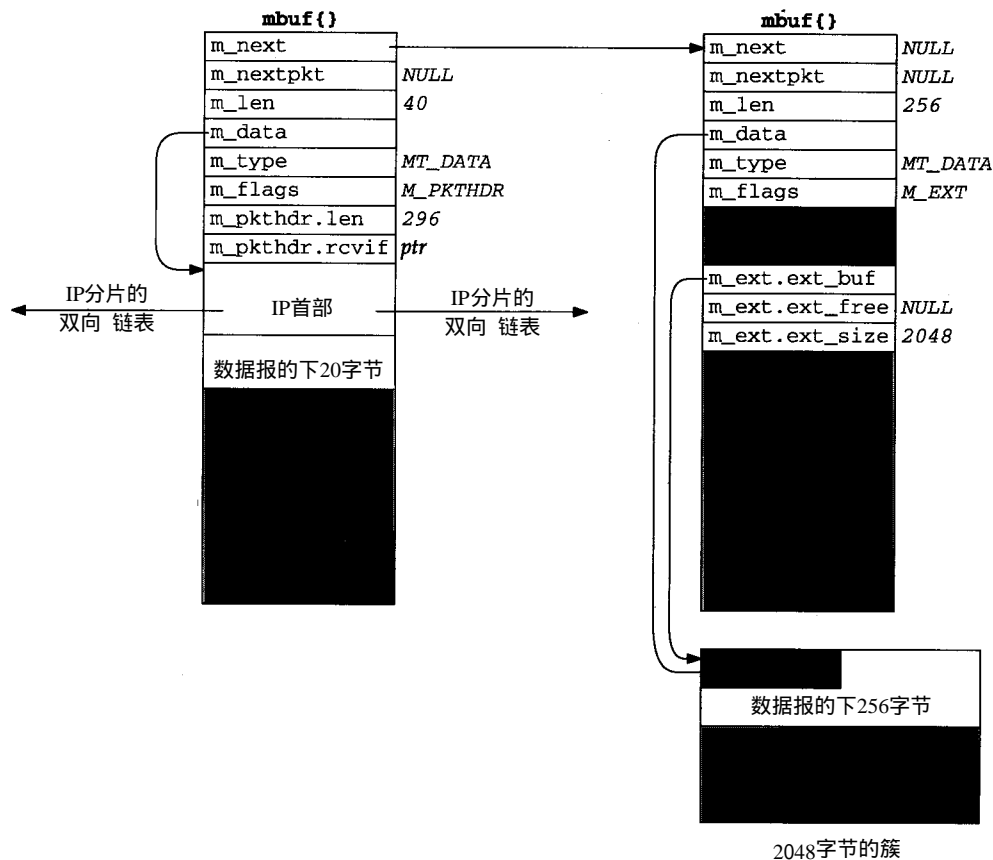


图2-18 调用m_pullup 后的长度为296的IP分片

2.6.5 TCP重组避免调用m_pullup

重组TCP报文段使用一个不同的技术而不是调用 m_pullup。这是因为 m_pullup 开销较大：分配存储器并且数据从一个簇复制到一个 mbuf 中。TCP 试图尽可能地避免数据的复制。

卷1的第19章提到大约有一半的 TCP 数据是批量数据（通常每个报文段有 512 或更多字节的数据），并且另一半是交互式数据（这里面有大约 90% 的报文段包含不到 10 字节的数据）。因此，当 TCP 从 IP 接收报文段时，它们通常是如图 2-14 左边所示的格式（一个小量的交互数据，存储在 mbuf 本身）或图 2-16 所示的格式（批量数据，存储在一个簇中）。当 TCP 报文段失序到达时，它们被 TCP 存储到一个双向链表中。如 IP 分片一样，在 IP 首部的字段用于存放链表的指针，既然这些字段在 TCP 接收了 IP 数据报后不再需要，这完全可行。但当 IP 首部存放在一个簇中，要将一个链表指针转换成一个相应的 mbuf 指针时，会引起同样的问题（图 2-17）。

为解决这个问题，在 27.9 节中我们会看到 TCP 把 mbuf 指针存放在 TCP 首部中的一些未用的字段中，提供一个从簇指回 mbuf 的指针，来避免对每个失序的报文段调用 m_pullup。如果 IP 首部包含在 mbuf 的数据区（图 2-18），则这个回指指针是无用的，因为宏 dtom 对这个链表指针会正常工作。但如果 IP 首部包含在一个簇中，这个回指指针将被使用。当我们在讨论 27.9 节的 tcp_reass 时，会研究实现这种技术的源代码。

2.6.6 m_pullup使用总结

我们已经讨论了关于使用 `m_pullup` 的三种情况：

- 大多数设备驱动程序不把一个 IP 数据报的第一部分分割到几个 mbuf 中。假设协议首部都紧挨着存放，则在每个协议 (IP、ICMP、IGMP、UDP 和 TCP) 中调用 `m_pullup` 的可能性很小。如果调用 `m_pullup`，通常是因为 IP 数据报太小，并且 `m_pullup` 返回一个差错，这时数据报被丢弃，并且差错计数器加 1。
- 对于每个接收到的 IP 分片，当 IP 数据报被存放在一个簇中时，`m_pullup` 被调用。这意味着，几乎对于每个接收的分片都要调用 `m_pullup`，因为大多数分片的长度大于 208 字节。
- 只要 TCP 报文段不被 IP 分片，接收一个 TCP 报文段，不论是否失序，都不需调用 `m_pullup`。这是避免 IP 对 TCP 分片的一个原因。

2.7 mbuf宏和函数的小结

在操作 mbuf 的代码中，我们会遇到图 2-19 中所列的宏和图 2-20 中所列的函数。图 2-19 中的宏以函数原型的形式显示，而不是以 `#define` 形式来显示参数的类型。由于这些宏和函数主要用于处理 mbuf 数据结构并且不涉及联网问题，因此我们不查看实现它们的源代码。还有另外一些 mbuf 宏和函数用于 Net/3 源代码的其他地方，但由于我们在本书中不会遇到它们，因此没有把它们列于图中。

宏	描 述
MCLGET	<p>获得一个簇(一个外部缓存)并将 <code>m</code> 指向的 mbuf 中的数据指针 (<code>m_data</code>) 设置为指向这个簇。如果存储器不可用，返回时不设置 mbuf 中的 <code>M_EXT</code> 标志</p> <pre>void MCLGET(struct mbuf *m, int nowait);</pre>
MFREE	<p>释放一个 <code>m</code> 指向的 mbuf。若 <code>m</code> 指向一个簇(设置了 <code>M_EXT</code>)，这个簇的引用计数器减 1，但这个簇并不被释放，直到它的引用计数器降为 0 (如 2.9 节所述)。返回 <code>m</code> 的后继 (由 <code>m->m_next</code> 指向，可以为空) 存放在 <code>n</code> 中</p> <pre>void MFREE(struct mbuf *m, struct mbuf *n);</pre>
MGETHDR	<p>分配一个 mbuf，并把它初始化为一个分组首部。这个宏与 <code>MGET</code> (图 2-12) 相似，但设置了标志 <code>M_PKTHDR</code>，并且数据指针 (<code>m_data</code>) 指向紧接分组首部后的 100 字节的缓存</p> <pre>void MGETHDR(struct mbuf *m, int nowait, int type);</pre>
MH_ALIGN	<p>设置包含一个分组首部的 mbuf 的 <code>m_data</code>，在这个 mbuf 数据区的尾部为一个长度为 <code>len</code> 字节的对象提供空间。这个数据指针也是长字对准方式的</p> <pre>void MH_ALIGN(struct mbuf *m, int len);</pre>
M_PREPEND	<p>在 <code>m</code> 指向的 mbuf 中的数据的前面添加 <code>len</code> 字节的数据。如果 mbuf 有空间，则仅把指针 (<code>m_data</code>) 减 <code>len</code> 字节，并将长度 (<code>m_len</code>) 增加 <code>len</code> 字节。如果没有足够的空间，就分配一个新的 mbuf，它的 <code>m_next</code> 指针被设置为 <code>m</code>。一个新 mbuf 的指针存放在 <code>m</code> 中。并且新 mbuf 的数据指针被设置，这样 <code>len</code> 字节的数据放置到这个 mbuf 的尾部(例如，调用 <code>MH_ALIGN</code>)。如果一个新 mbuf 被分配，并且原来的 mbuf 的分组首部标志被设置，则分组首部从老 mbuf 中移到新 mbuf 中</p> <pre>void M_PREPEND(struct mbuf *m, int len, int nowait);</pre>
dtom	<p>将指向一个 mbuf 数据区中某个位置的指针 <code>x</code> 转换成一个指向这个 mbuf 的起始的指针。</p> <pre>struct mbuf *dtom(void *x);</pre>
mtod	<p>将 <code>m</code> 指向的 mbuf 的数据区指针的类型转换成 <code>type</code> 类型</p> <pre>type mtod(struct mbuf *m, type);</pre>

图2-19 我们在本书中会遇到的 mbuf宏

函 数	说 明
m_adj	从 m 指向的mbuf中移走 len 字节的数据。如果 len 是正数，则所操作的是紧排在这个mbuf的开始的 len 字节数据；否则是紧排在这个mbuf的尾部的 len 绝对值字节数据 void m_adj (struct mbuf * m , int len);
m_cat	把由 n 指向的mbuf链表链接到由 m 指向的mbuf链表的尾部。当我们讨论IP重组时(第10章)会遇到这个函数 void m_cat (struct mbuf * m , struct mbuf * n);
m_copy	这是m_copym的三参数版本，它隐含的第4个参数的值为M_DONTWAIT struct mbuf * m_copy (struct mbuf * m , int $offset$, int len);
m_copydata	从 m 指向的mbuf链表中复制 len 字节数据到由 cp 指向的缓存。从mbuf链表数据区起始的 $offset$ 字节开始复制 void m_copydata (struct mbuf * m , int $offset$, int len , caddr_t cp);
m_copyback	从 cp 指向的缓存复制 len 字节的数据到由 m 指向的mbuf，数据存储在mbuf链表起始 $offset$ 字节后。必要时，mbuf链表可以用其他mbuf来扩充 void m_copyback (struct mbuf * m , int $offset$, int len , caddr_t cp);
m_copym	创建一个新的mbuf链表，并从 m 指向的mbuf链表的开始 $offset$ 处复制 len 字节的数据。一个新mbuf链表的指针作为此函数的返回值。如果 len 等于常量M_COPYALL，则从这个mbuf链表的 $offset$ 开始的所有数据都将被复制。在2.9节中，我们会更详细地介绍这个函数 struct mbuf * m_copym (struct mbuf * m , int $offset$, int len , int $nowait$);
m_devget	创建一个带分组首部的mbuf链表，并返回指向这个链表的指针。这个分组首部的 len 和 $rcvif$ 字段被设置为 len 和 ifp 。调用函数copy从设备接口(由 buf 指向)将数据复制到mbuf中。如果 $copy$ 是一个空指针，调用函数bcopy。由于尾部协议不再被支持， off 为0。我们在2.6节讨论了这个函数 struct mbuf * m_devget (char * buf , int len , int off , struct ifnet * ifp , void (* $copy$)(const void *, void *, u_int));
m_free	宏MFREE的函数版本 struct mbuf * m_free (struct mbuf * m);
m_freem	释放 m 指向的链表中的所有mbuf void m_freem (struct mbuf * m);
m_get	宏MGET的函数版本。我们在图2-12中显示过此函数 struct mbuf * m_get (int $nowait$, int $type$);
m_getclr	此函数调用宏MGET来得到一个mbuf，并把108字节的缓存清零 struct mbuf * m_getclr (int $nowait$, int $type$);
m_gethdr	宏MGETHDR的函数版本 struct mbuf * m_gethdr (int $nowait$, int $type$);
m_pullup	重新排列由 m 指向的mbuf中的数据，使得前 len 字节的数据连续地存储在链表中的第一个mbuf中。如果这个函数成功，则宏mtod能返回一个正好指向这个大小为 len 的结构。我们在2.6节讨论了这个函数 struct mbuf m_pullup (struct mbuf * m , int len);

图2-20 在本书中我们要遇到的mbuf函数

所有原型的参数 $nowait$ 是M_WAIT或M_DONTWAIT，参数 $type$ 是图2-10中所示的MT_XXX中的一个。

M_PREPEND的一个例子是，从图 1-7 转换到图 1-8 的过程中，当 IP 和 UDP 首部被添加到数据的前面时要调用这个宏，因为另一个 mbuf 要被分配。但当这个宏再次被调用（从图 1-8 转换成图 2-2）来添加以太网首部时，在那个 mbuf 中已有存放这个首部的空间。

`M_copydata`的最后一个参数的类型是 `caddr_t`，它代表“内核地址”。这个数据类型通常定义在 `<sys/types.h>` 中，为 `char *`。它最初在内核中使用，但被某些系统调用使用时被外露出来。例如，`mmap` 系统调用，不论是 4.4BSD 或 SVR4 都把 `caddr_t` 作为第一个参数的类型并作为返回值类型。

2.8 Net/3联网数据结构小结

本节总结我们在Net/3联网代码中要遇到的数据结构类型。在Net/3内核中用到其他数据结构(感兴趣的读者可以查看头文件<sys/queue.h>),但下面这些是我们在本书中要遇到的。

- 1) 一个mbuf链：一个通过m_next指针链接的mbuf链表。我们已经看过几个这样的例子。
- 2) 只有一个头指针的mbuf链的链表。mbuf链通过每个链的第一个mbuf中的m_nextpkt指针链接起来。

图2-21所示的就是这种链表。这种数据结构的例子是一个插口发送缓存和接收缓存。

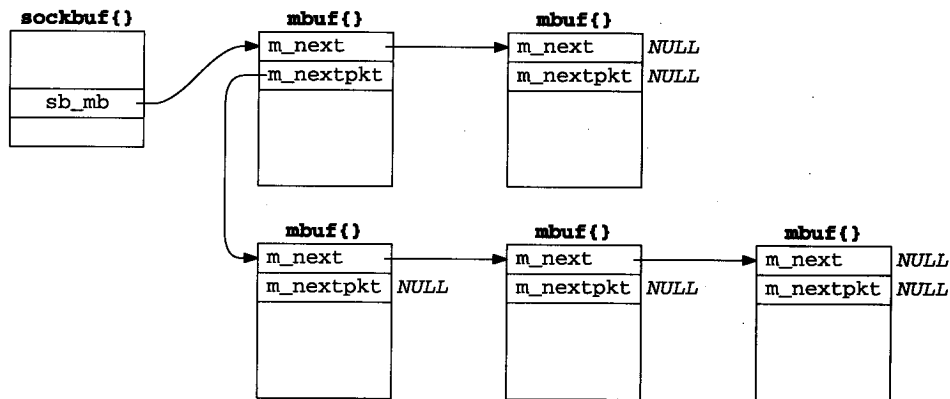


图2-21 只有头指针的mbuf链的链表

顶部的两个mbuf形成这个队列中的第一个记录，底下三个mbuf形成这个队列的第二个记录。对于一个基于记录的协议，如UDP，我们在每个队列中能遇到多个记录。但对于像TCP这样的协议，它没有记录的边界，每个队列我们只能发现一个记录（一个mbuf链可能包含多个mbuf）。

把一个 mbuf 追加到队列的第一个记录中要遍历所有第一个记录的 mbuf，直到遇到 m_next 为空的 mbuf。而追加一个包含新记录的 mbuf 链到这个队列中，要查找所有记录直到遇到 m_nextpkt 为空的记录。

- 3) 一个有头指针和尾指针的 mbuf 链的链表。

图2-22显示的是这种类型的链表。我们在接口队列中会遇到它（图3-13），并且在图2-2中已显示过它的一个例子。

在图2-21中仅有一点改变：增加了一个尾指针，来简化增加一个新记录的操作。

- #### 4) 双向循环链表。

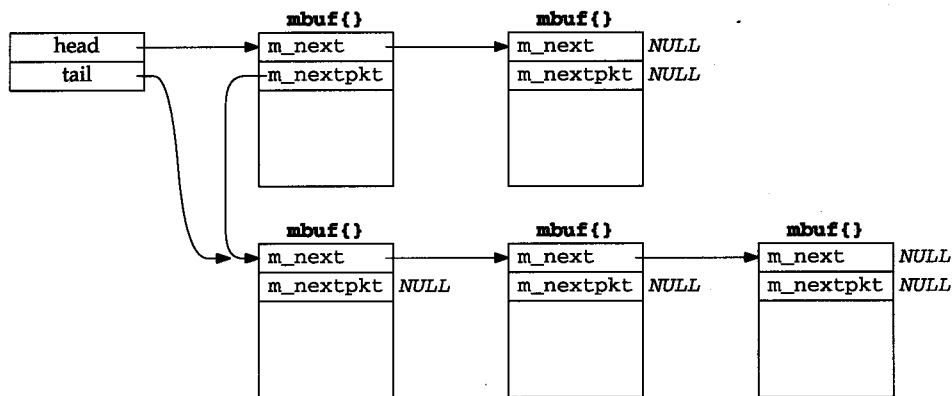


图2-22 有头指针和尾指针的链表

图2-23所示的是这种类型的链表，我们在IP分片与重装(第10章)、协议控制块(第22章)及TCP失序报文段队列(第27.9节)中会遇到这种数据结构。

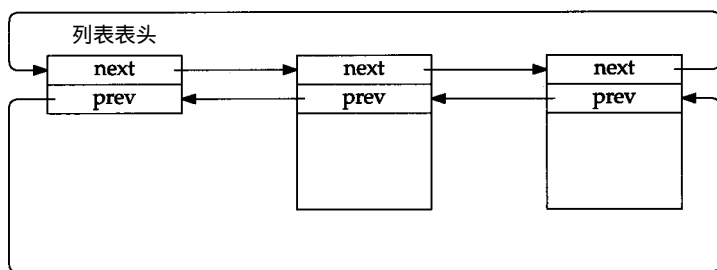


图2-23 双向循环链表

在这个链表中的元素不是mbuf——它们是一些定义了两个相邻的指针的结构：一个next指针跟着一个previous指针。两个指针必须在结构的起始处。如果链表为空，表头的next和previous指针都指向这个表头本身。

在图中我们简单地把向后指针指向另一个向后指针。显然所有的指针应包含它所指向的结构地址，即向前指针的地址(因为向前和向后指针总是放在结构的起始处)。

这种类型的数据结构能方便地向前向后遍历，并允许方便地在链表中任何位置进行插入与删除。

函数insque和remque(图10-20)被调用来对这个链表进行插入和删除。

2.9 m_copy和簇引用计数

使用簇的一个明显的好处就是在要求包含大量数据时能减少mbuf的数目。例如，如果不使用簇，要有10个mbuf才能包含1024字节的数据：第一个mbuf带有100字节的数据，后面8个每个存放108字节数据，最后一个存放60字节数据。分配并链接10个mbuf比分配一个包含1024字节簇的mbuf开销要大。簇的一个潜在缺点是浪费空间。在我们的例子中使用一个簇(2048 + 128)要2176字节，而1280字节不到1簇(10 × 128)。

簇的另外一个好处是在多个mbuf间可以共享一个簇。在TCP输出和m_copy函数中我们遇到过这种情况，但现在我们要更详细地说明这个问题。

例如，假设应用程序执行一个 `write`，把 4096 字节写到 TCP 插口中。假设插口发送缓存原来是空的，接收窗口至少有 4096，则会发生以下操作。插口层把前 2048 字节的数据放在一个簇中，并且调用协议的发送例程。TCP 发送例程把这个 mbuf 追加到它的发送缓存后，如图 2-24 所示，并调用 `tcp_output`。结构 `socket` 中包含 `sockbuf` 结构，这个结构中存储着发送缓存 mbuf 链的链表的表头：`so_snd.sb_mb`。

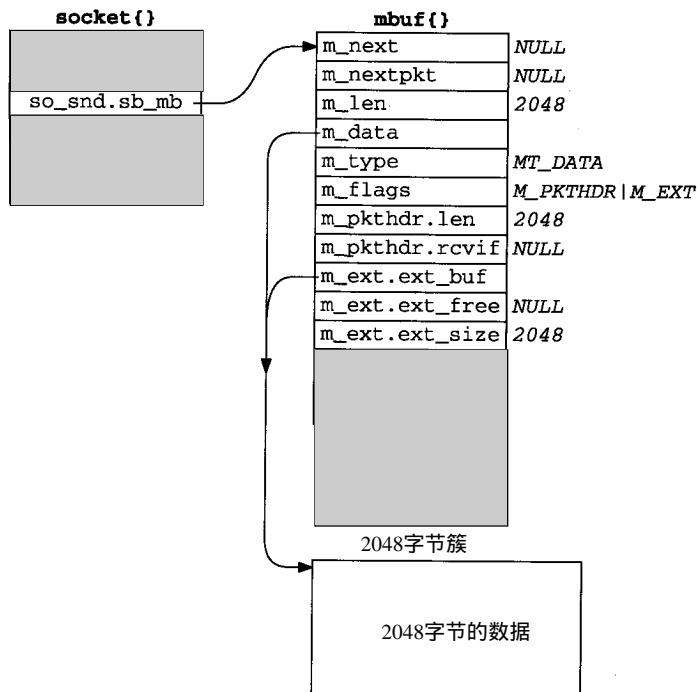


图2-24 包含2048字节数据的TCP插口发送缓存

假设这个连接(典型的是以太网)的一个TCP最大报文段大小(MSS)为1460，`tcp_output` 建立一个报文段来发送包含前 1460 字节的数据。它还建立一个包含 IP 和 TCP 首部的 mbuf，为链路层首部(16 字节)预留了空间，并将这个 mbuf 链传给 IP 输出。在接口输出队列尾部的 mbuf 链显示在图 2-25 中。

在 1.9 节的 UDP 例子中，UDP 用 mbuf 链来存放数据报，在前面添加一个 mbuf 来存放协议首部，并把此链传给 IP 输出。UDP 并不把这个 mbuf 保存在它的发送缓存中。而 TCP 不能这样做，因为 TCP 是一个可靠协议，并且它必须维护一个发送数据的副本，直到数据被对方确认。

在这个例子中，`tcp_output` 调用函数 `m_copy`，请求复制 1460 字节的数据，从发送缓存起始位置开始。但由于数据被存放在一个簇中，`m_copy` 创建一个 mbuf(图 2-25 的右下侧)并且对它初始化，将它指向那个已存在的簇的正确位置(此例中是簇的起始处)。这个 mbuf 的长度是 1460，虽然有另外 588 字节的数据在簇中。我们所示的这个 mbuf 链的长度是 1514，包括以太网首部、IP 首部和 TCP 首部。

在图 2-25 的右下侧我们还显示了这个 mbuf 包含一个分组首部，但它不是链中的第一个 mbuf。当 `m_copy` 复制一个包含一个分组首部的 mbuf 并且从原来 mbuf 的起始地址开始复制时，分组首部也被复制下来。因为这个 mbuf 不是链中的第一个 mbuf，

这个额外的分组首部被忽略。而在这个额外的分组首部中的 `m_pkthdr.len` 的值 2048 也被忽略。

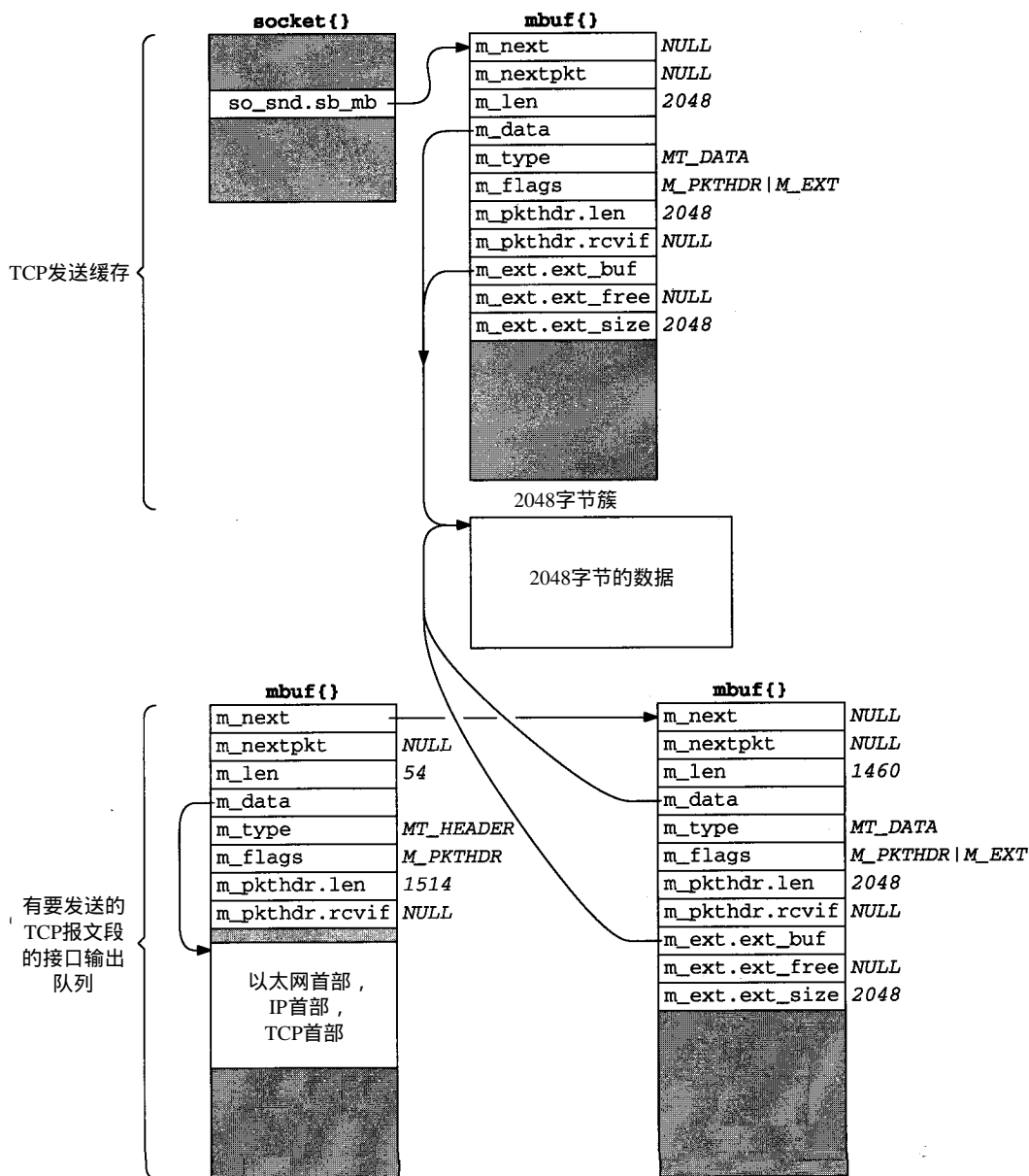


图2-25 TCP插口发送缓存和接口输出队列中的报文段

这个共享的簇避免了内核将数据从一个 mbuf 复制到另一个 mbuf 中——这节约了很多开销。它是通过为每个簇提供一个引用计数来实现的，每次另一个 mbuf 指向这个簇时计数加 1，当一个簇释放时计数减 1。仅当引用计数到达 0 时，被这个簇占用的存储器才能被其他程序使用(见习题 2.4)。

例如，当图 2-25 底部的 mbuf 链到达以太网设备驱动程序并且它的内容已被复制给这个设备时，驱动程序调用 `m_freem`。这个函数释放带有协议首部的第一个 mbuf，并注意到链中第

二个mbuf指向一个簇。簇引用计数减1，但由于它的值变成了1，它仍然保存在存储器中。它不能被释放，因为它仍在TCP发送缓存中。

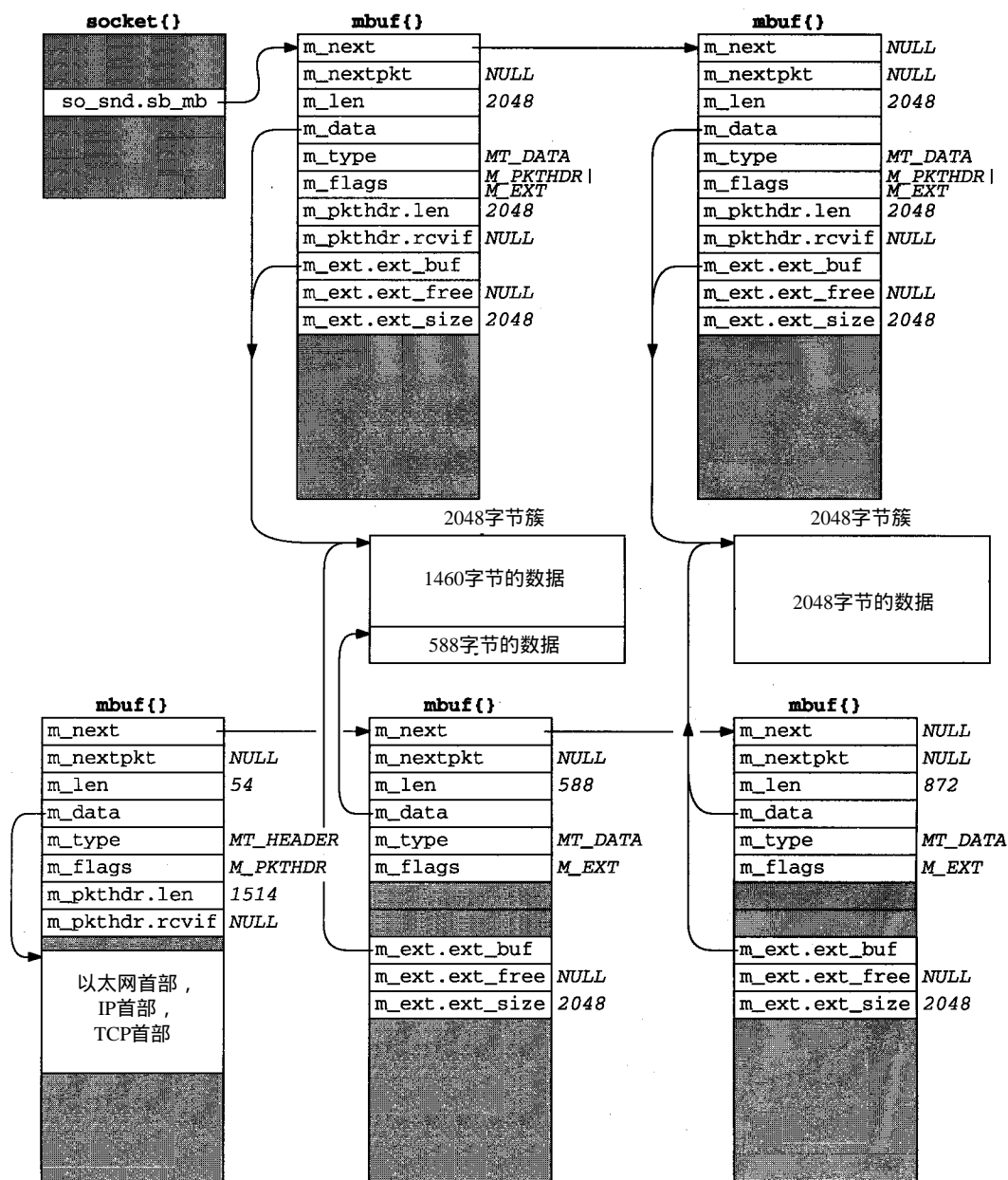


图2-26 用于发送1460字节TCP报文段的mbuf链

继续我们的例子，由于在发送缓存中剩余的588字节不能组成一个报文段，tcp_output在把1460字节的报文段传给IP后返回(在第26章我们要详细说明在这种条件下tcp_output发送数据的细节)。插口层继续处理来自应用程序的数据：剩下的2048字节被存放到一个带有一个簇的mbuf中，TCP发送例程再次被调用，并且新的mbuf被追加到插口发送缓存中。因为能发送一个完整的报文段，tcp_output建立另一个带有协议首部和1460字节数据的mbuf链表。

`m_copy`的参数指定了1460字节的数据在发送缓存中的起始位移和长度(1460字节)。这显示在图2-26中,并假设这个mbuf链在接口输出队列中(这个链中的第一个mbuf的长度反映了以太网首部、IP首部及TCP首部)。

这次1460字节的数据来自两个簇:前588字节来自发送缓存的第一个簇而后面的872字节来自发送缓存的第二个簇。它用两个mbuf来存放1460字节,但`m_copy`还是不复制这1460字节的数据——它引用已存在的簇。

这次我们没有在图2-26右下侧的任何mbuf中显示一个分组首部。原因是调用`m_copy`的起始位移为零。但在插口发送缓存中的第二个mbuf包含一个分组首部,而不是链中的第一个mbuf。这是函数`sosend`的特点,这个额外的分组首部被简单地忽略了。

我们在通篇中会多次遇到函数`m_copy`。虽然这个名字隐含着对数据进行物理复制,但如果数据被包含在一个簇中,却是仅引用这个簇而不是复制。

2.10 其他选择

mbuf远非完美,并且时常遭到批评。但不管怎样,它们形成了所有今天正使用着的伯克利联网代码的基础。

一种由Van Jacobson [Partridge 1993]完成的Internet协议的研究实现,它废除了支持大量连续缓存的复杂的mbuf数据结构。[Jacobson 1993]提出了一种速度能提高一到两个数量级的改进方案,还包括其他改进,及废除mbuf。

这个mbuf的复杂性是一种权衡,以避免分配大的固定长度的缓存,这样的大缓存很少能被装满。而在这种情况下,mbuf要进行设计,一个VAX-11/780有4兆存储器,是一个大系统,并且存储器是昂贵的资源,需要仔细分配。今天存储器已不昂贵了,而焦点已经转向更高的性能和代码的简单性。

mbuf的性能基于存放在mbuf中数据量。[Hutchinson and Peterson 1991]显示了处理mbuf的时间与数据量不是线性关系。

2.11 小结

在本书几乎所有的函数中我们都会遇到mbuf。它们的主要用途是在进程和网络接口之间传递用户数据时用来存放用户数据,但mbuf还用于保存其他各种数据:源地址和目标地址、插口选项等等。

根据`M_PKTHDR`和`M_EXT`标志是否被设置,这里有4种类型的mbuf:

- 无分组首部,mbuf本身带有0~108字节数据;
- 有分组首部,mbuf本身带有0~100字节数据;
- 无分组首部,数据在簇(外部缓存)中;
- 有分组首部,数据在簇(外部缓存)中。

我们查看了几个mbuf宏和函数的源代码,但不是所有的mbuf例程源代码。图2-19和图2-20提供了所有我们在本书中遇到的mbuf例程的函数原型和说明。

查看了我们要遇到的两个函数的操作:`m_devget`,很多网络设备驱动程序调用它来存

储一个收到的帧；`m_pullup`，所有输入例程调用它把协议首部连续放置在一个 `mbuf` 中。

由一个 `mbuf` 指向的簇（外部缓存）能通过 `m_copy` 被共享。例如，用于 TCP 输出，因为一个被传输的数据的副本要被发送端保存，直到数据被对方确认。比起进行物理复制来说，通过引用计数，共享簇提高了性能。

习题

- 2.1 在图2-9中定义了 `M_COPYFLAGS`。为什么不复制标志 `M_EXT`？
- 2.2 在2.6节中，我们列出了两个 `m_pullup` 失败的原因。实际上有三个原因。查看这个函数的源代码（附录B），并发现另外一个原因。
- 2.3 为避免宏 `dtom` 遇到在2.6节中我们所讨论的问题——当数据在簇中时，为什么不仅仅给每个簇加一个指向 `mbuf` 的回指指针？
- 2.4 既然一个 `mbuf` 簇的大小是2的幂（典型的是1024或2048），簇内的空间不能用于引用计数。查看 `Net/3` 的源代码（附录B），并确定这些引用计数存储在什么地方。
- 2.5 在图2-5中，我们注意到两个计数器 `m_drops` 和 `m_wait` 现在没有实现。修改 `mbuf` 例程增加这些计数器。