2.mbuf：存储器缓存

四种mbuf

- m_flags=0，mbuf只包含数据
- m_flags为M_PKTHDR，指示这是一个分组首部
- m_flags为M_EXT，指针m_data指向分配的簇的位置

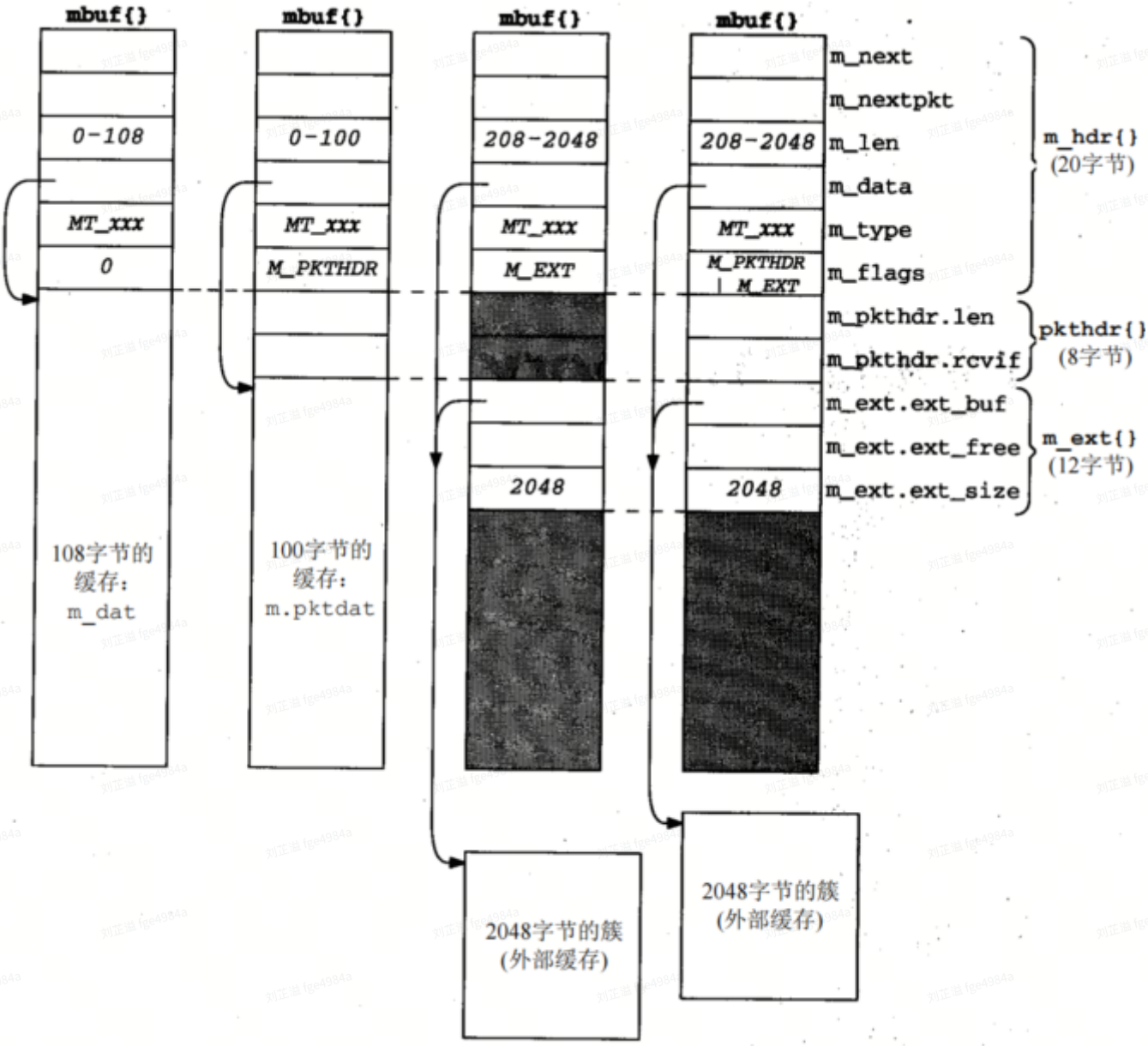

图2-1　根据不同m_flags 值的四种不同类型的mbuf

- mbuf结构总是128Byte
- 可以存在m_len=0的数据缓存

- 每个mbuf中的成员m_data指向相应的缓存的开始，但是可以指向相应缓存的任意位置，不一定是起始
- m_nextpkt可以把多个分组链接成一个mbuf链表队列

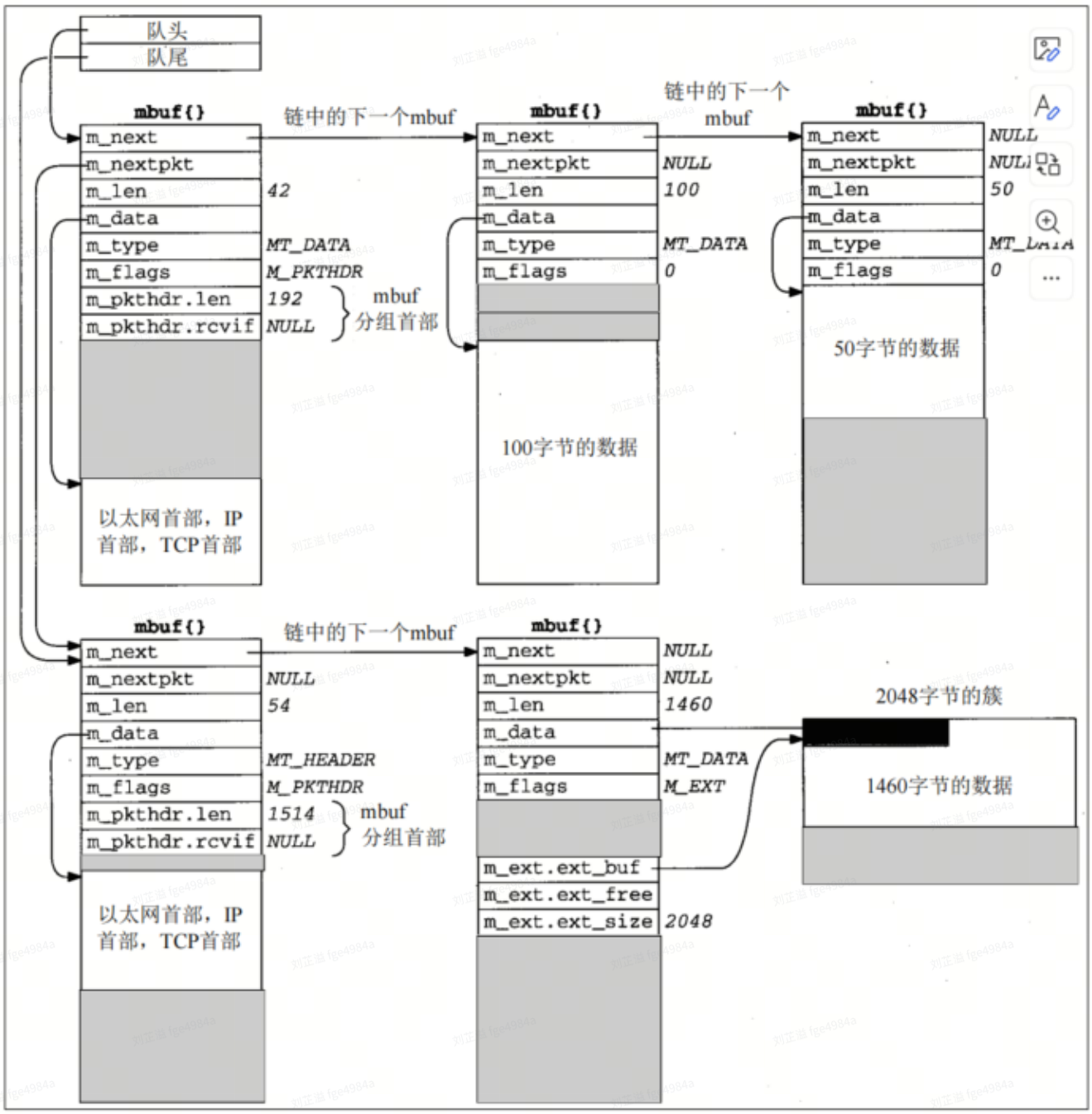

图2-2　在一个队列中的两个分组：第一个带有192字节数据，第二个带有1514字节数据

mbuf结构

mbuf.h

```
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

获取mbuf结构

MGET宏

- MBUFLOCK：先提高优先级，然后恢复优先级；防止执行语句++时被网络中断

mbuf.h

```
154 #define MGET(m, how, type) { \
155     MALLOC((m), struct mbuf *, MSIZE, mbtypes[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

m_retry函数

- 每个协议会定义一个drain协议；在系统缺乏可用存储器时可以被m_reclaim调用

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

uipc_mbuf.c

```
 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

图2-13 m_retry 函数

m_get函数

uipc_mbuf.c

```
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。

mbuf锁

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。

m_devget函数和m_pullup函数

- m_pullup指定数目的字节在链表第一个mbuf中紧挨着存放

m_devget

- 接收到一个以太网帧时，设备驱动程序调用函数m_devget创建一个mbuf链表；将设备的帧复制到该链表
- 数据长度小于84字节，分配了一个14字节(16是为了对齐)用于输出以太网首部；icmp_reflect和tcp_respond，建立输出报文时它们会产生一个应答，预留空间可以节省时间
- 数据在85~100字节之间，粗那种一个分组首部mbuf中，但是开始没有16字节空间

- 数据在101~207字节之间，前100字节存放在第一个mbuf，剩下的存放在第二个mbuf
- 数据如果大于等于208个字节，使用一个或多个簇

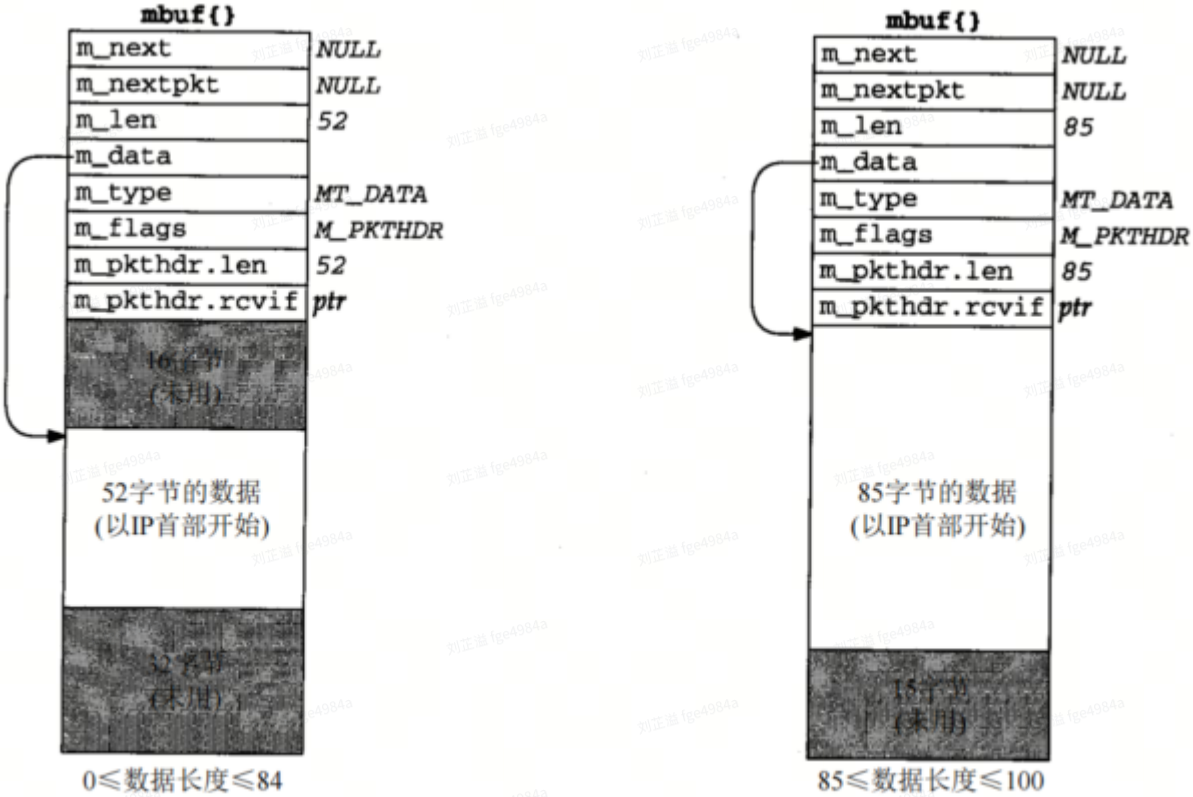

图2-14 m_devget 创建的前两种类型的mbuf

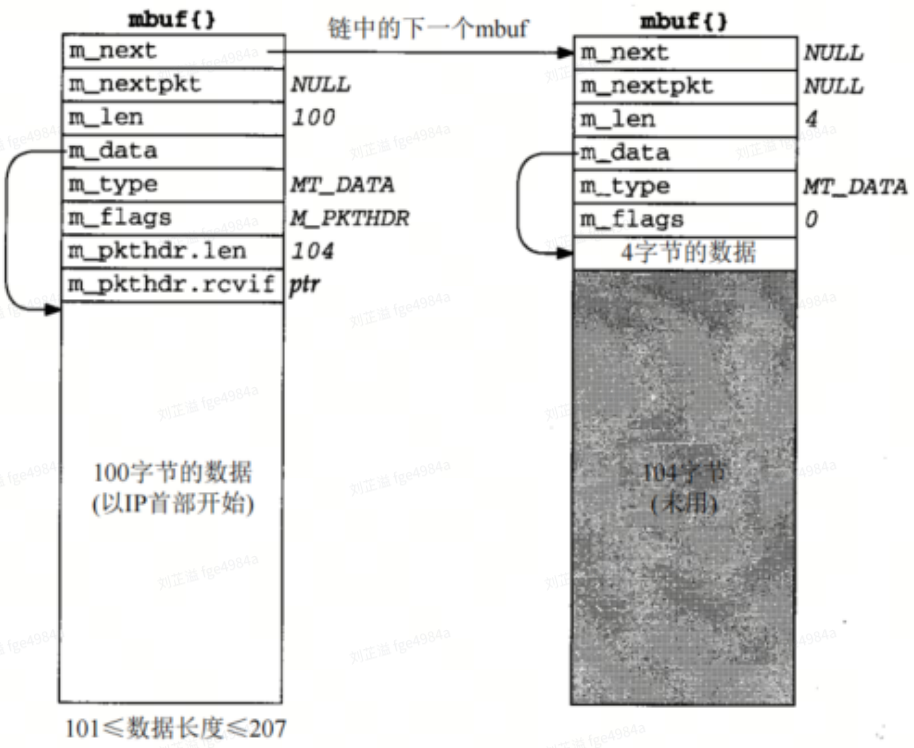

图2-15 m_devget 创建的第3种mbuf

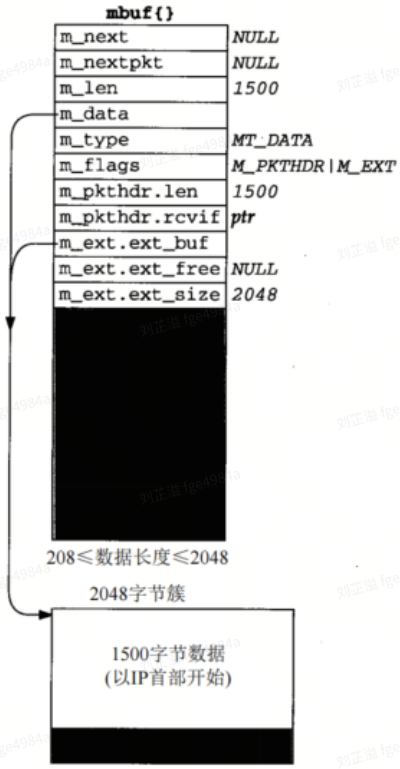

图2-16 m_devget 创建的第4种mbuf

mtod和dtom宏

- mtod返回一个指向mbuf数据的指针，并将指针声明为指定类型

- dtom取得一个存放在mbuf中任意位置的指针，返回这个mbuf结构本身的一个指针
  - 内核存储分配器总是为mbuf连续分配MSIZE字节的存储块，dtom仅仅是清除参数中指针的低位来发现该位置
  - 但dtom的参数指向一个簇或者在簇内，该宏不能正确执行

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

m_pullup函数

- 一个协议发现第一个mbuf的数据量小于协议首部的最小长度
  - 假定协议首部的剩余部分在链表中的下一个mbuf；m_pullup使得前N字节的数据被连续地存放在第一个mbuf
- IP的重组
  - 分配一个新的mbuf，挂在mbuf链表前面，并从簇中取走40字节放入这个新的mbuf中(20字节IP首部和20字节TCP首部)

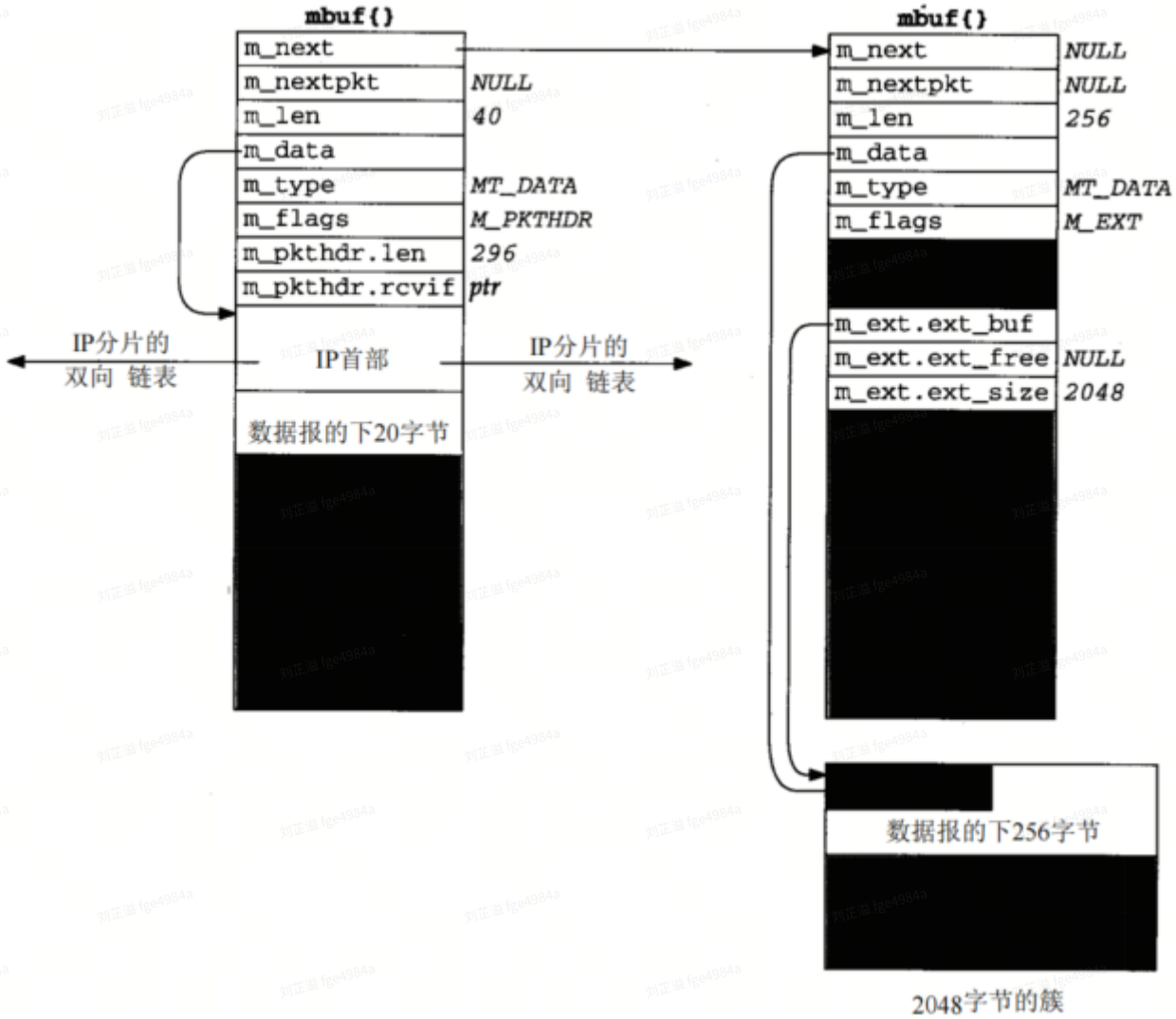

图2-18 调用m_pullup 后的长度为296的IP分片

- TCP的重组避免调用m_pullup；TCP将mbuf指针存放在TCP首部一些未用的字段中，提供一个从簇指回mbuf的指针，避免对每个失序的报文段调用m_pullup

m_pullup失败

- 需要其他mbuf并调用MGET失败
- 整个mbuf链表中的数据总数小于要求的连续字节数；通常因为该原因失败

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。

mbuf宏

宏	描　述
MCLGET	获得一个簇(一个外部缓存)并将*m*指向的mbuf中的数据指针(m_data)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M_EXT标志 void **MCLGET**(struct mbuf *m, int *nowait*);
MFREE	释放一个*m*指向的mbuf。若*m*指向一个簇(设置了M_EXT)，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*m*的后继(由m->m_next指向，可以为空)存放在*n*中 void **MFREE**(struct mbuf *m, struct mbuf *n);
MGETHDR	分配一个mbuf，并把它初始化为一个分组首部。这个宏与MGET(图2-12)相似，但设置了标志M_PKTHDR，并且数据指针(m_data)指向紧接分组首部后的100字节的缓存 void **MGETHDR**(struct mbuf *m, int *nowait*, int *type*);
MH_ALIGN	设置包含一个分组首部的mbuf的m_data，在这个mbuf数据区的尾部为一个长度为len字节的对象提供空间。这个数据指针也是长字对准方式的 void **MH_ALIGN**(struct mbuf *m, int *len*);
M_PREPEND	在*m*指向的mbuf中的数据的前面添加*len*字节的数据。如果mbuf有空间，则仅把指针(m_data)减len字节，并将长度(m_len)增加*len*字节。如果没有足够的空间，就分配一个新的mbuf，它的m_next指针被设置为*m*。一个新mbuf的指针存放在*m*中。并且新mbuf的数据指针被设置，这样*len*字节的数据放置到这个mbuf的尾部(例如，调用MH_ALIGN)。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 void **M_PREPEND**(struct mbuf *m, int *len*, int *nowait*);
dtom	将指向一个mbuf数据区中某个位置的指针*x*转换成一个指向这个mbuf的起始的指针。 struct mbuf ***dtom**(void *x);
mtod	将*m*指向的mbuf的数据区指针的类型转换成*type*类型 type **mtod**(struct mbuf *m*, *type*);

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

数据结构

- mbuf链表
- 只有一个头指针的mbuf链表
  - socket的发送缓存和接收缓存

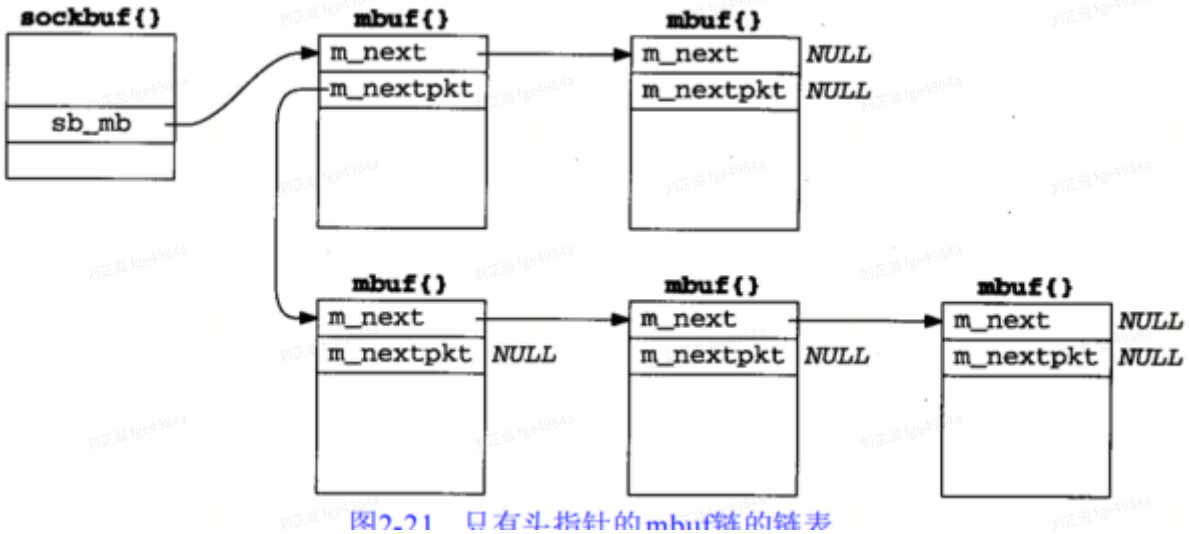

图2-21 只有头指针的mbuf链的链表

- 一个有头指针和尾指针的mbuf链的链表
  - 在接口队列会遇到

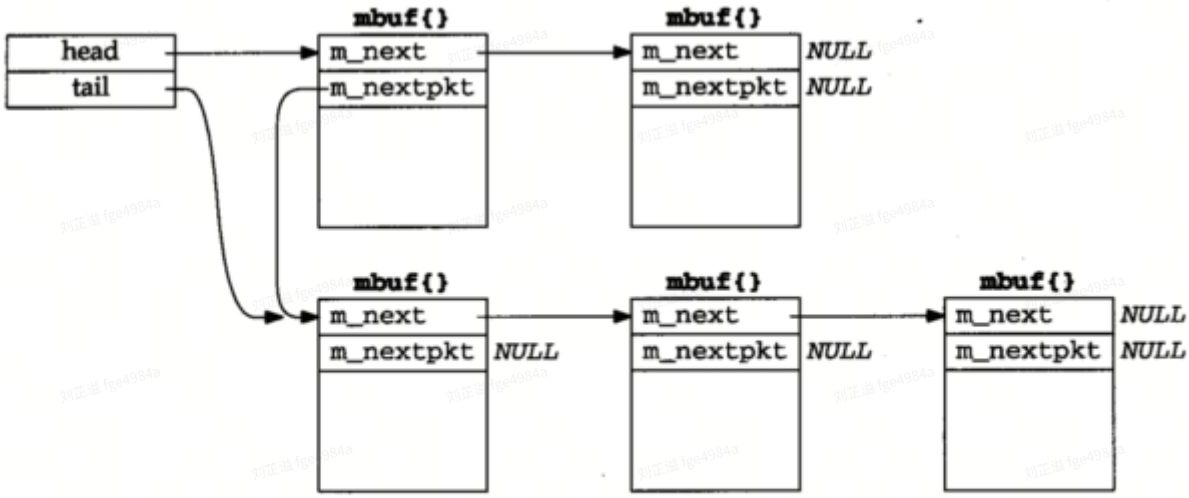

图2-22 有头指针和尾指针的链表

- 双向循环链表
  - 在IP分片与重装、协议控制块和TCP失序报文段队列会遇到

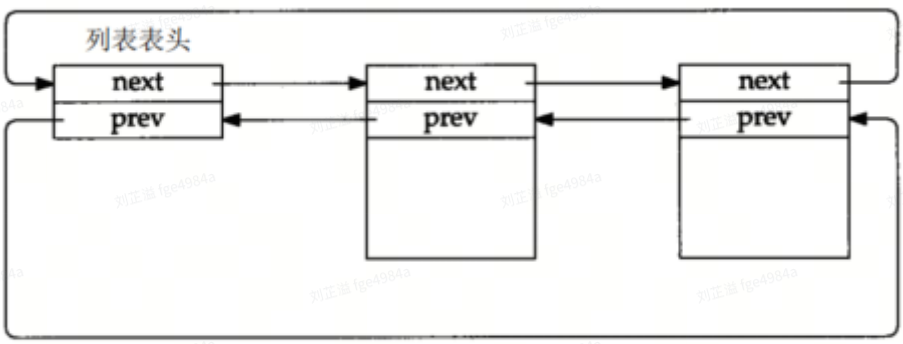

图2-23 双向循环链表

mcopy和簇引用计数

- 簇可以实现多个mbuf共享一个簇
- UDP并不把mbuf保存在它的发送缓存中；TCP必须维护一个发送数据的副本，直到数据被对方确认
- 共享簇避免了内核将数据从一个mbuf复制到另一个mbuf中

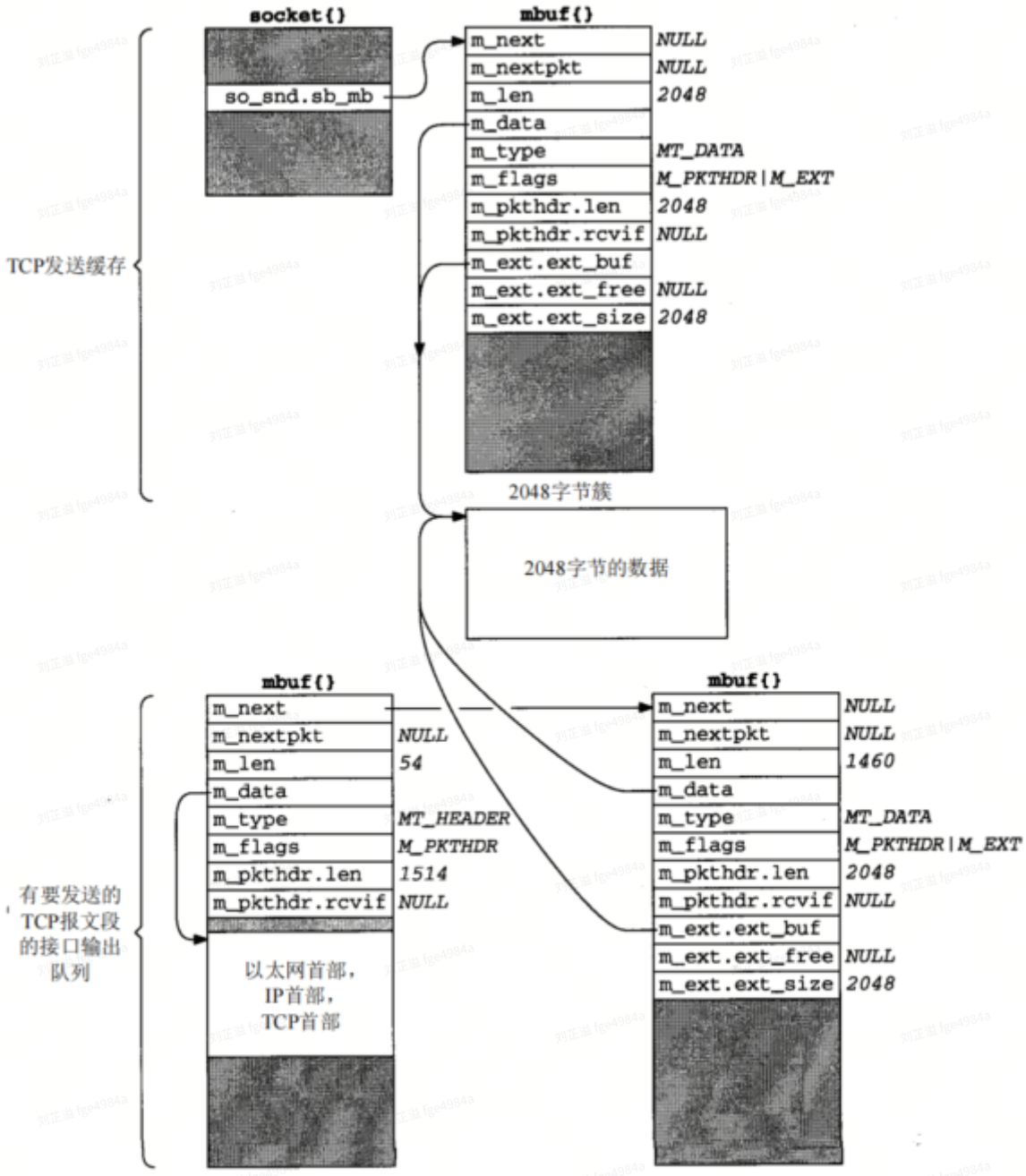

图2-25　TCP插口发送缓存和接口输出队列中的报文段

- 发送缓存中剩余的588字节不能组成一个报文段；tcp_output建立另一个带有协议首部和和1460数据报的mbuf链表
  - 发送缓存第一个簇的588字节
  - 发送缓存第二个簇的872字节
  - m_copy不复制这1460字节数据，引用已经存在的簇

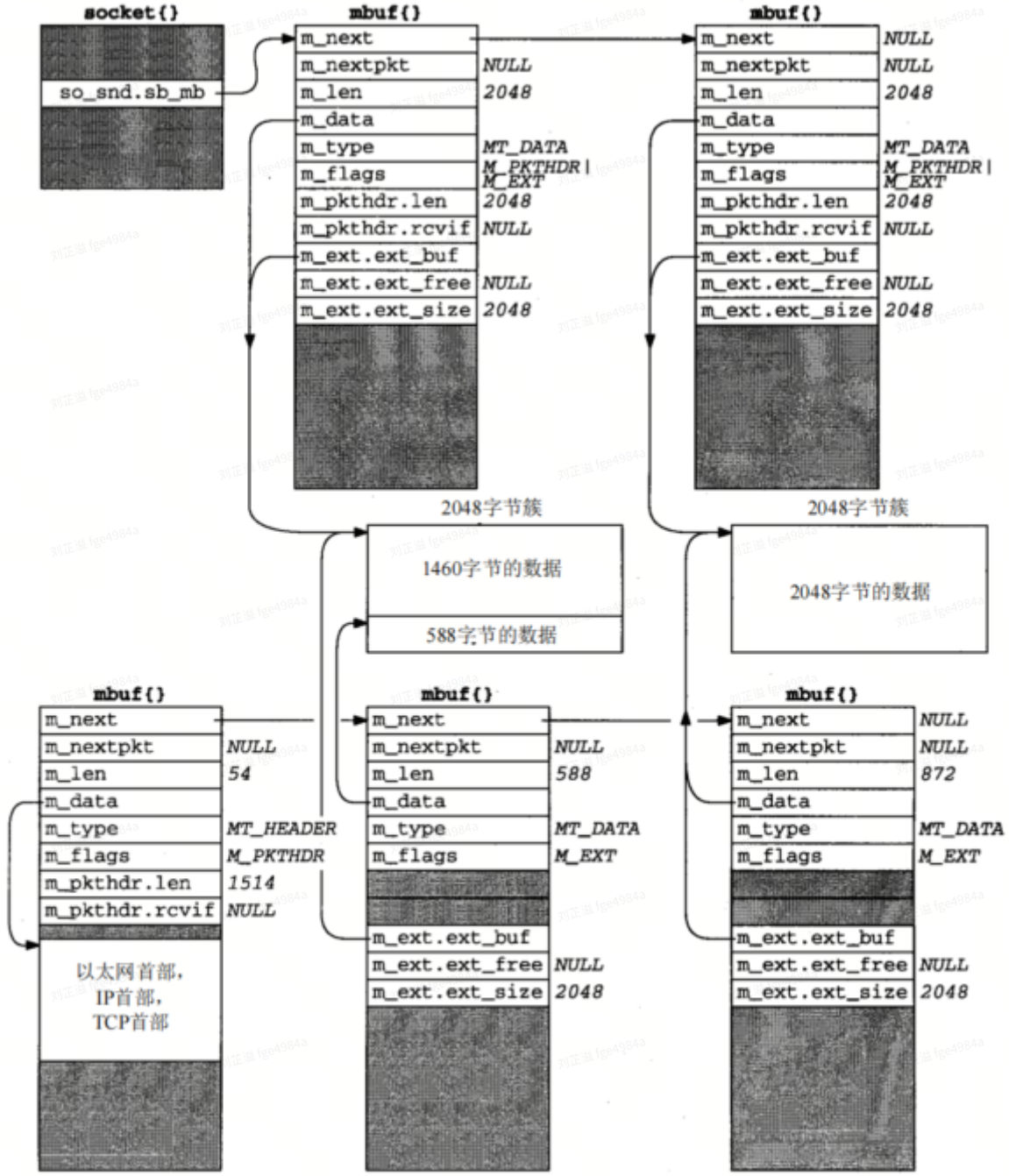

图2-26　用于发送1460字节TCP报文段的mbuf链