使用 malloc 子系统来分配系统内存

使用 **malloc** 子系统将内存分配给应用程序。

malloc 子系统是一个由以下子例程组成的内存管理 API：

- **malloc**
- **calloc**
- **realloc**
- **free**
- **mallopt**
- **mallinfo**
- **alloca**
- **valloc**
- **posix_memalign**

malloc 子系统管理称为*堆*的逻辑内存对象。堆是一个内存区域，驻留在应用程序地址空间中编译器分配的数据的最后一个字节和该数据区结束处之间。堆是内存对象，从中可以分配内存，同时也是 **malloc** 子系统 API 返回内存的目标。

malloc 子系统执行以下基本内存操作：

- 分配：
  由 **malloc**、**calloc valloc**、**alloca** 和 **posix_memalign** 子例程执行。
- 释放：
  由 **free** 子例程执行。
- 重新分配：
  由 **realloc** 子例程执行。

为了与系统 V 兼容，支持 **mallopt** 和 **mallinfo** 子例程。在程序开发期间，**mallinfo** 子例程可用于获取由 **malloc** 子例程管理的堆的信息。**mallopt** 子例程可用于放弃页对齐、页大小可用空间以及启用和禁用缺省的分配器。与 **malloc** 子例程类似的是 **valloc** 子例程，提供它是为了要与 Berkeley 兼容性库相兼容。

更多其他信息，请参阅以下几个部分：

处理进程堆

_edata 是一个符号，其地址是初始化的程序数据最后一个字节后面的第一个字节。**_edata** 符号指示进程堆的启动，当分配第一块数据时由 **malloc** 子系统扩大。**malloc** 子系统通过增加进程 brk 值扩大进程堆，该值表示进程堆的末端。这是通过调用 **sbrk** 子例程完成的。**malloc** 子系统按应用程序指示的要求对进程堆扩展。

进程堆分为*已分配的*和*已释放的*内存块。空闲池由可用于后续分配的内存组成。通过先将一个内存块从空闲池中除去，然后再返回到调用函数，使一个指针指向该块，分配就完成了。通过先分配新的大小的内存块、再将原来块中的数据移动到新块中，然后释放原来的块，来完成重新分配。已分配的内存块由正被应用程序使用的多个进程堆组成。因为内存块实际并没有从堆中除去（只是它们的状态从已分配更

改为空闲），所以进程堆的大小在应用程序释放内存后不会减少。

32 位应用程序中的进程地址空间

在系统中运行的 32 位应用程序具有一个地址空间，该地址空间分为以下几段：

分段	描述
0x00000000 到 0x0fffffff	包含内核。
0x10000000 到 0x1fffffff	包含应用程序文本。
0x20000000 到 0x2fffffff	包含应用程序数据、进程堆和应用程序堆栈。
0x30000000 到 0xcfffffff	可供共享内存或 **mmap** 服务使用。
0xd0000000 到 0xdfffffff	包含共享库文本。
0xe0000000 到 0xefffffff	可供共享内存或 **mmap** 服务使用。
0xf0000000 到 0xffffffff	包含应用程序共享库数据。

64 位应用程序中的进程地址空间

在系统中运行的 64 位应用程序具有一个地址空间，该地址空间分为以下几段：

分段	描述
0x0000 0000 0000 0000 到 0x0000 0000 0fff ffff	包含内核。
0x0000 0000 1000 0000 到 0x0000 0000 efff ffff	保留为直接 shmat 和 mmap
0x0000 0000 f000 0000 到 0x0000 0000 ffff ffff	保留的。
0x0000 0001 0000 0000 到 0x07ff ffff ffff ffff	包含应用程序文本、应用程序数据、进程堆以及共享内存或 **mmap** 服务。
0x0800 0000 0000 0000 到 0x08ff ffff ffff ffff	私自装入的对象。
0x0900 0000 0000 0000 到 0x09ff ffff ffff ffff	共享库文本和数据。
0x0f00 0000 0000 0000 到 0x0fff ffff ffff ffff	应用程序堆栈。

> **注：**
>
> AIX® 对分配给应用程序的存储使用延迟页面调度时间片分配技术。当存储以子例程（例如 **malloc**）分配给应用程序时，只有在访问该存储后才会将调页空间分配给该存储。这个技术对分配大量稀疏内存段的应用程序有用。但是，这个技术可能影响分配大量内存的应用程序的可移植性。如果应用程序希望当没有足够备份存储支持内存请求时对 malloc 的调用会失败，那么应用程序可能会分配太多的内存。当稍后访问该内存时，机器很快耗尽了调页空间并且操作系统杀死了那些进程以使系统不完全耗尽虚拟内存。分配内存的应用程序必须确保对于正被分配的存储有备份存储存在。将 **PSALLOC** 环境变量设为 PSALLOC=early 会将调页空间分配技术更改为早期的分配算法。在早期的分配中，一旦请求了内存就分配调页空间。有关更多信息，请参阅*操作系统与设备管理*中的『调页空间和虚拟内存』。

了解系统分配策略

*分配策略*指的是一组数据结构和算法，用于代表堆并实现分配、取消分配以及重新分配。**malloc** 子系统支持几个不同分配策略，包括缺省分配策略、watson 分配策略、malloc 3.1 分配策略和用户定义的分配策略。用于访问 **malloc** 子系统的 API 对所有分配策略都是相同的；仅底层的实现不同。

可使用以下环境变量来指定分配策略和该策略的任何正常或调试选项：

- **MALLOCTYPE** 指定分配策略。
- **MALLOCOPTIONS** 指定选定分配策略的正常选项。
- **MALLOCDEBUG** 指定选定分配策略的调试选项。
- **MALLOCALIGN** 指定程序外部的缺省 **malloc** 对齐。

缺省的分配策略一般情况下更有效率，因此是大多数应用程序的首选。如比较不同分配策略中所述，其他分配策略具有一些独特的行为特征，在特定情况下，这些特征可能是有益的。

各种分配策略的一些选项相互相容且可顺序使用。顺序使用选项时，请使用逗号 (,) 来分隔 **MALLOCOPTIONS** 和 **MALLOCDEBUG** 环境变量指定的选项。

可将 **MALLOCALIGN** 环境变量设为每个 **malloc()** 分配所需的缺省对齐格式。例如，

```
MALLOCALIGN=16;  export MALLOCALIGN
```

MALLOCALIGN 环境变量可设为 2 的任何次幂，该值大于或等于相应运行方式（4 个字节对应于 32 位方式，8 个字节对应于 64 位方式）下的指针大小。对于 32 位向量启用的程序，可将此环境变量设为 16，这样，如果需要，将针对向量数据类型适当地对齐所有 **malloc()**。请注意，64 位向量程序将接收 16 字节对齐的分配。

另外，在程序内部，该程序可使用 **mallopt(M_MALIGN, 16)** 例程来更改缺省 **malloc()** 以提供 16 字节对齐的分配。**mallopt(M_MALIGN)** 例程使程序能够在运行时动态控制缺省 malloc 对齐。

了解缺省分配策略

缺省分配策略将堆中的可用空间维护为 *cartesian* 二分搜索树中的节点，其中按地址从左到右（向右则地址增大）并按长度从高到低（这样没有子代比父代大）对节点排序。该数据结构没有对树所支持的块大小的数量施加任何限制，使得可能的块大小的范围非常广泛。树的重新组织技术可缩短对节点位置、插入、删除的访问时间，并且还能防止碎片的产生。

缺省的分配策略支持以下可选的功能：

分配

需要少量开销来为分配请求提供服务。这归因于对元数据前缀的需要和对每个内存块适当对齐的需要。所有分配的元数据前缀的大小对于 32 位和 64 位程序分别为 8 和 16 字节。每个块必须在 16 或 32 字节边界上对齐，这样大小 n 的一个分配所需的内存总量为：

```
size = roundup(n + prefix_size, alignment requirement)
```

例如，在 32 位进程中大小 37 的分配将需要 `roundup(37 + 8, 16)`，等于 48 字节。

具有最低地址且大于或等于所需大小的树节点将从树中除去。如果找到的块大于所需的大小，那么块将被分为两块：其中一块是所需的大小，第二块是剩下的大小。第二块称为 *runt*，将返回到空闲树以供将来分配。第一块则返回至调用程序。

如果在空闲树中未找到足够大小的块，那么对堆进行扩展，所需扩展大小的块将添加到空闲树中，然后如前面所述继续分配。

释放

使用 **free** 子例程释放的内存块将返回到树中的根节点。至新节点插入点路径上的每个节点均将被检查，以查看它是否与将被插入的节点相邻。如果相邻，那么这两个节点将合并，且新近合并的节点将重新放置在树中。如果未找到任何相邻的块，那么仅将节点插入树中的合适位置。合并相邻块可以显著地减少堆碎片。

重新分配

如果重新分配的块的大小大于原始的块，那么使用 **free** 子例程将原始的块返回到空闲树，以便任何可能的合并能够发生。接着分配所需大小的新块，数据从原始的块移动到新的块，然后新的块返回至调用程序。

如果重新调用的块的大小小于原始的块，那么该块将被分割，其中较小的块返回到空闲树。

限制

缺省分配策略支持以下选项：

- malloc 多堆
- malloc 存储区
- Malloc 回收
- Malloc 线程高速缓存
- 了解 no_overwrite 选项

了解 watson 分配策略

Watson 分配策略将堆中的可用空间维护为两个独立红黑树中的节点：一个按地址排序，另一个按大小排序。红黑树提供比缺省分配器的 cartesian 树更简单和更有效的树操作，这样 watson 分配策略通常比缺省情况下更快。

分配

Watson 分配策略具有和缺省分配策略相同的开销需求。

在大小树中搜索大于等于所需大小的最小可能块。然后从大小树中除去这个块。如果找到的块大于所需的大小，那么块将被分为两块：其中一块是剩下的大小，第二块是所需的大小。第一块称为 *runt*，将返回到大小树以供将来分配。第二块则返回至调用者。如果大小树中找到的块正好是所需的大小，那么从大小树和地址树中除去该块，然后返回至调用者。

如果在空闲树中未找到足够大小的块，那么对进程堆进行扩展，该扩展大小的块将添加到大小树和地址树中，然后如前面所述继续分配。

释放

使用 **free** 子例程释放的内存块将返回到地址树中的根节点。至新节点插入点路径上的每个节点均将被检查，以查看它是否与将被插入的节点相邻。如果相邻，那么这两个节点将合并，且新近合并的节点将重新放置在大小树中。如果未找到任何相邻的块，那么仅将节点插入地址树和大小树中的合适位置。

插入后，必须检查红黑树是否达到了正确的平衡。

重新分配

如果重新分配的块的大小将大于原始的块，那么使用 **free** 子例程将原始的块返回到空闲树，以便能够发生任何可能的合并。接着分配所需大小的新块，数据从原始的块移动到新的块，然后新的块返回至调用程序。

如果重新分配的块的大小小于原始的块，那么该块将被分割，剩余部分返回到空闲树。

限制

Watson 分配策略支持以下选项：

- malloc 多堆
- Malloc 回收
- Malloc 线程高速缓存
- 了解 no_overwrite 选项

了解 malloc 3.1 分配策略

可通过在进程启动之前设置 `MALLOCTYPE=3.1` 来选择 malloc 3.1 分配策略。在这之后，所有由 shell 运行的 32 位程序将使用 malloc 3.1 分配策略（64 位程序将继续使用缺省的分配策略）。

malloc 3.1 分配策略作为一组 28 个散列存储区来维护堆，其中每个存储区均指向一个链接的列表。每个链接的列表包含多个特定大小的块。散列存储区的索引指出了链接列表中块的大小。块的大小可通过以下公式来计算：

$$size = 2^{i+4}$$

其中 *i* 表示存储区。这意味着由存储区 0 建锚的列表中块的长度为 20+4 = 16 字节长度。因此，如果前缀是 8 个字节，那么这些块可以满足 0 到 8 个字节长度的块要求。下表说明了如何在各个存储区中分配要求的大小。

> **注：**
>
> 此算法可以使用两倍于应用程序实际请求的内存数。大于 4096 字节的存储区还需要一页，因为大于或等于 4096 字节的页的对象是页对齐的。因为前缀紧靠块之前，所以仅前缀就需要一整页。

存储区	块大小	已映射的大小	已使用的页
0	16	0 ... 8	
1	32	9 ... 24	
2	64	25 ... 56	
3	128	57 ... 120	
4	256	121 ... 248	
5	512	249 ... 504	

6	1 K	505 ... 1 K-8	
7	2K	1 K-7 ... 2 K-8	
8	4 K	2 K-7 ... 4 K-8	2
9	8 K	4 K-7 ... 8 K-8	3
10	16 K	8 K-7 ... 16 K-8	5
11	32 K	16 K-7 ... 32 K-8	9
12	64 K	32 K-7 ... 64 K-8	17
13	128 K	64 K-7 ... 128 K-8	33
14	256 K	128 K-7 ... 256 K-8	65
15	512 K	256 K-7 ... 512 K-8	129
16	1 M	256 K-7 ... 1 M-8	257
17	2 M	1 M-7 ... 2 M-8	513
18	4 M	2 M-7 ... 4 M-8	1 K + 1
19	8 M	4 M-7 ... 8 M-8	2 K + 1
20	16 M	8 M-7 ... 16 M-8	4 K + 1
21	32 M	16 M-7 ... 32 M-8	8 K + 1
22	64 M	32 M-7 ... 64 M-8	16 K + 1
23	128 M	64 M-7 ... 128 M-8	32 K + 1
24	256 M	128 M-7 ... 256 M-8	64 K + 1
25	512 M	256 M-7 ... 512 M-8	128 K + 1
26	1024 M	512 M-7 ... 1024 M-8	256 K + 1
27	2048 M	1024 M-7 ... 2048 M-8	512 K + 1

分配

通过首先将请求的字节数转换为存储区阵列中的索引可从空闲池中分配块，请使用以下方程式：

```
needed = requested + 8

If needed <= 16,
then
bucket = 0

If needed > 16,
then
bucket = (log(needed)/log(2) rounded down to the nearest integer) - 3
```

列表中存储区的每块的大小是块的大小 = $2^{\text{存储区} + 4}$。如果存储区中的列表为空，那么通过使用 **sbrk** 子例程将块添加到列表中来分配内存。如果块小于页，那么使用 **sbrk** 子例程来分配页，而通过将块大小分成页大小而获得的块数量将添加到该列表中。如果块等于或大于页，那么使用 **sbrk** 子例程来分配所需内存，单个块将添加到存储区的空闲列表中。如果空闲列表非空，那么列表头处的块将返回至调用程序。列表中的下一个块则成为新的头。

释放

当内存块返回至空闲池以后，分配将计算在存储区索引内。然后要释放的块将添加到存储区空闲列表的头中。

重新分配

库存块重新分配后，所需的大小将与现有的块大小比较。由于单个存储区处理的块在大小上有很大差异，因此新块的大小通常映射为同一存储区中的原始块大小。在这些情况下，前缀的长度将被更新，以反映新的大小，同时相同的块将返回。如果所需大小大于现有的块，那

么块将被释放，新的块分配自新的存储区，然后数据从旧的块移动到新的块中。

限制

设置 MALLOCTYPE=3.1 将仅启用适用于 32 位程序的 malloc 3.1 策略。对于 64 位程序，要使用 malloc 3.1 策略，必须将 **MALLOCTYPE** 环境变量显式地设为 MALLOCTYPE=3.1_64BIT。该分配策略效率比缺省值更低，对于大多数情况，不建议使用。

malloc 3.1 分配策略支持以下选项：

- Malloc 回收
- 了解 no_overwrite 选项

了解池分配策略

当管理存储对象小于 523 字节时，**Malloc** 池对于 libc 函数 **malloc**、**calloc**、**free**、**posix_memalign** 和 **realloc** 是一个高性能的前端。这种性能能够显著地缩短路径长度和提高数据高速缓存的利用率。对于多线程应用程序，更为有利的是可以使用线程局部池锚点来避免原子操作。该前端可用于 libc（*yorktown* 和 *watson*）中当前提供的所有存储管理方案的结合。

要使用 **malloc** 池，请运行以下命令：

```
export MALLOCOPTIONS=pool<:max_size>
```

当指定该选项时，将在 **malloc** 初始化过程中创建池集合，其中每个池都是固定大小对象的链接列表。最小的池可容纳指针大小（如 32 位应用程序的 4 个字节或 64 位应用程序的 8 个字节）的对象。每个连续池可容纳比前一个池大一个指针大小的对象。这意味着 32 位应用程序具有 128 个池，64 位应用程序具有 64 个池。池集合表示为指向链接列表的指针数组。

Malloc 池使用其自身内存（池堆），该池堆不与标准 **malloc** 共享。当已指定时，**max_size** 选项将再增加 2 MB 并用于控制池堆的大小。**max_size** 选项可指定为十进制或前缀为 0x 或 0X 的十六进制（例如，export MALLOCOPTIONS=pool:0x1700000 在取整之后将 **max_size** 设为 24 MB）。

对于 32 位应用程序，池堆大小最小为 2 MB。如果需要更多存储器并且总池堆存储器小于 **max_size**，可增加 2 MB。每个 2 MB 区域将在 2 MB 的边界上，但不必与其他 2 MB 区域相邻。对于 64 位应用程序，在 **malloc** 初始化时分配 **max_size** 大小的单一连续池堆并且不再扩展。如果未指定 **max_size**，那么缺省值为 512 MB（对于 32 位应用程序）或 32 MB（对于 64 位应用程序）。 如果指定了一个较大的大小，那么对于 32 位和 64 位方式，都将 **max_size** 设为 512 MB。对于 32 位方式，**max_size** 设为 512 MB，而对于 64 位方式，**max_size** 设为 3.7 GB（如果指定较大的大小）。

存储器利用率

所有池锚点都初始设为 NULL 或空。当 **malloc** 池发出请求且相应的池为空时，将调用例程在 1024 字节边界上以连续的 1024 字节块从池堆中分配存储区。此时会创建多个具有请求大小的对象。返回第一个对象的地址以满足请求，同时余下的对象链接在一起并放置在池锚点上。对于每个 1024 字节块，在辅助表中都存在一个 2 字节的条目，可自由使用该条目来确定返回对象的大小。

当 **malloc** 池释放一个对象后，仅将其“推”到相应的池锚点上。并未试图合并块以创建更大的对象。

由于这种行为，**malloc** 池可比其他格式的 **malloc** 使用更多的存储器。

对齐

必须通过相应地设置 **MALLOCALIGN** 环境变量以指定 **malloc()**、**calloc()** 和 **realloc()** 子例程的缺省对齐。即使未设置 **MALLOCALIGN** 环境变量，**posix_memalign()** 子例程仍起作用。如果 **MALLOCALIGN** 大于 512，那么不使用 **malloc** 池。

高速缓存效率

使用 **malloc 池**分配的存储器对象没有前缀或后缀。因此，数据高速缓存行中的应用程序可用数据的压缩将更具紧密。每个大小为 2 的 N 次方的存储器对象都将在同等大小的边界上对齐，并且每个对象都包含在最小数量高速缓冲行中。**malloc** 和 **free** 子例程不扫描树或链接列表，因此不会使用高速缓存。

多线程支持

Malloc 池在多线程场景中可显著地提高性能，因为它减少了锁争用和对原子操作的需求。

负载均衡支持

在某些多线程场景中，一个线程的空闲池可能由于动态分配内存的重复释放而变得非常大。但是，其他线程可能无法使用此内存。

在每个池达到阈值后，负载均衡支持会促使线程将每个池中的一半内存释放到全局池，以便其他线程可以使用内存。可以调节将重新调整

线程池的阈值。

要开启负载均衡支持，必须导出以下选项：

```
export MALLOCOPTIONS=pool:0x80000000,pool_balanced
```

```
export MALLOCFREEPOOL=min_size<-max_size>:threshold_value<,min_size<-max_size>:
threshold_value, ... >,default:threshold
```

以下示例为提供 0-16 字节块内存的池设置阈值（256 字节块），而且为提供 32 字节块内存的池设置阈值（512 字节块）。对于其他池，128 字节块是阈值。

```
export MALLOCFREEPOOL=0-16:256,32:512,default:128
```

调试支持

该高性能前端没有调试版本。如果设置了 **MALLOCDEBUG** 环境变量，那么忽略池选项。在激活池之前，应使用 **malloc** 调试应用程序。

了解用户定义的分配策略

malloc 子系统提供了一个机制，通过它用户可以开发他们自己的算法用于管理系统堆和分配内存。

了解 no_overwrite 选项

所有分配策略可用的附加选项为 no_overwrite。为了减少 malloc 子系统内 glink 代码的开销，**malloc** 子系统 API 函数描述符将被实际底层实现的函数描述符覆盖。因为某些程序（例如第三方调试器）在函数指针以这种方式修改后可能无法运行，所以 no_overwrite 选项可用来禁用该优化。

要禁用该优化，请在进程启动前设置 MALLOCOPTIONS=no_overwrite。

比较各种分配策略

当以支持的方式单独或组合使用时，以上详述的各种 malloc 分配策略为应用程序开发者提供灵活性。开发者负责识别应用程序的独特需求并且以有用的方式调整各种分配策略参数。

比较缺省分配策略和 malloc 3.1 分配策略

因为 malloc 3.1 分配策略将每个分配请求的大小向上舍入到下一个 2 的幂次，所以它会产生相当大的虚拟内存碎片和实内存碎片以及较差的引用局部性。通常，缺省分配策略是较好的选择，因为它精确地分配了所需的空间量，且能更有效地收回先前已使用的内存块。

但不幸的是，某些应用程序可能无意中会依靠 malloc 3.1 分配策略的副作用来获得可接受的性能甚至是正常的运行。例如，当使用 malloc 3.1 分配器时，超载运行到矩阵最后的程序可能会正确运行，这只是因为向上舍入进程提供了附加空间。如果使用缺省分配器，那么相同的程序可能出现错误的行为甚至运行失败，这是因为缺省的分配器仅分配所需的字节数。

另外举个例子，由于 malloc 3.1 算法在空间回收方面的低效率，应用程序接收到的空间几乎始终已设为零（当进程在其工作段中第一次碰到一给定页时，该页已设为零）。应用程序可能要依赖该副作用才能正常执行。实际上，将已分配空间置零不是 **malloc** 子例程的指定功能，并且会导致那些只要按需要初始化且可能不置零的程序发生不必要的性能损耗。因为缺省分配器在重新使用空间方面更加主动，所以那些依赖于从 **malloc** 中接收已置零的存储空间的程序可能在使用缺省分配器时失败。

类似的情况，如果程序不断地将一个结构重新分配得稍大一些，那么 malloc 3.1 分配器可能不需要经常移动该结构。在许多情况下，**realloc** 子例程能够利用隐含在 malloc 3.1 分配算法中的舍入运算所提供的额外空间。通常，缺省分配器必须将结构移动到稍大一些的区域中，因为其他的一些空间可能已被它上面的 **malloc** 子例程所调用。如果使用缺省分配器而非 malloc 3.1 分配器，**realloc** 子例程的性能看上去可能会下降。实际上，它是应用程序结构所隐含的表面消耗。

调试应用程序对系统堆的管理错误

malloc 子系统提供了一组调试工具，旨在帮助应用程序开发者调试并纠正程序的堆管理中的错误。这些调试工具通过 **MALLOCDEBUG** 环境变量控制。

malloc 环境变量和选项的摘要

下表说明了 **MALLOCTYPE** 和 **MALLOCOPTIONS** 环境变量之间的兼容性。

*表 1. **MALLOCTYPE** 和 **MALLOCOPTIONS** 环境变量之间的兼容性*

	<缺省分配器>	**3.1**	**Watson**	**用户：**	**multiheap（和子选项）**

<缺省分配器>		否	否	否	是
3.1	否		否	否	否
Watson	否	否		否	是
用户：	否	否	否		否
multiheap（和子选项）	是	否	是	否	
buckets（和子选项）	是	否	否	否	是
Thread Cache	是	否	是	否	是
disclaim	是	是	是	否	是
no_overwrite	是	是	是	是	是

*表 2. **MALLOCTYPE** 和 **MALLOCOPTIONS** 环境变量之间的兼容性（续）*

	buckets（和子选项）	**Thread Cache**	**disclaim**	**no_overwrite**
<缺省分配器>	是	是	是	是
3.1	否	否	是	是
Watson	否	是	是	是
用户：	否	否	否	是
multiheap（和子选项）	是	是	是	是
buckets（和子选项）		是	是	是
Thread Cache	是		是	是
disclaim	是	是		是
no_overwrite	是	是	是	

下表说明了 **MALLOCTYPE**、**MALLOCOPTIONS** 和 **MALLOCDEBUG** 环境变量之间的兼容性。

*表 3. **MALLOCTYPE**、**MALLOCOPTIONS** 和 **MALLOCDEBUG** 环境变量之间的兼容性*

	<缺省分配器>	**3.1**	**Watson**	**用户：**	**multiheap（和子选项）**
catch_overflow（和子选项）	是	否	是	否	是[1]
report_allocations	是	否	是	否	是
postfree_checking	是	否	是	否	是
validate_ptrs	是	否	是	否	是
trace	是	否	是	否	是
log	是	否	是	否	是
verbose	是	否	是	否	是
check_arena	是	否	是	否	是

注：

1. 不会调试存储区分配。

*表 4. **MALLOCTYPE**、**MALLOCOPTIONS** 和 **MALLOCDEBUG** 环境变量之间的兼容性（续）*

	buckets（和子选项）	Thread Cache	disclaim	no_overwrite
catch_overflow（和子选项）	是	是[1]	是	是
report_allocations	是	是	是	是
postfree_checking	是	是	是	是
validate_ptrs	是	是	是	是
trace	是	是	是	是
log	是	是	是	是
verbose	是	是	是	是
check_arena	是	是	是	是

注：

1. 不会调试线程高速缓存分配。

More topics

同步多线程技术

动态逻辑分区

sed 程序信息

共享库和共享内存

AIX 向量编程

编写重入和线程安全代码

为安装封装软件

源代码控制系统

子例程，示例程序和库