第 7 章

Delphi 的内存管理器

最小化内核中并不包含内存管理器，这表明使用最小化内核开发应用程序，与使用汇编一样“原始”：需要使用标准的 Win32 API 来分配、使用和管理内存块。但是这通常会存在效率问题，也可能会导致内存漏洞。

如果代码中使用到需要动态内存的数据类型，那么这些数据类型的内部例程将访问 Delphi 的内存管理器。这些数据类型中就包括了 `AnsiString`。这意味着绝大多数的 Delphi 程序都需要内存管理器。

除了缺省的内存管理器外，Delphi 还实现了 *ShareMem.pas* 和相应的内存管理模块 *borlndmm.dll*，以同步 DLL 与 HOST 程序中的内存管理器。

与缺省的内存管理器一样，共享内存管理器(*borlndmm.dll*)并不是最有效率的。因此某些时候，选择第三方的内存管理器可能会得到更好的性能。

开发人员随时可以写一个内存管理器来嘲笑一下那个叫 *borlndmm.dll* 的东西。但在开始动手之前，请确认你已理解本章所述的全部细节，否则你可能不得不面对一次次的进程崩溃，并无可奈何地在 `uses` 关键字的后面加入 `ShareMem`。

知己知彼，才能无往不利。否则，笑到最后的还会是 Borland。

7.1 Delphi 的内存管理器实现框架

Delphi 在 *GetMem.inc* 中实现了自己的内存管理器。对于应用程序模块来说，除了变量、常量和系统内存的数据结构之外，其他任何时候的内存分配行为，只可能是如下三种情况之一：

- 通过操作系统 API 进行的进程内内存分配。
- 使用 Delphi 内存管理例程进行的堆分配。

☞　通过汇编指令或者局部变量定义进行的栈分配[①]。

通过操作系统 API 进行的内存分配主要有虚地址内存分配和堆分配。前者是指使用 API：VirtualAlloc()进行的分配，后者是指用 API：HeapAlloc()进行的分配。

Delphi内存管理例程中的所谓“堆分配”，与操作系统中的“堆”概念并不一致。Delphi中指的是使用“堆”这种数据结构进行内存管理的方法，而操作系统中指的是一个堆内存区。[②]

在 *GetMem.inc* 中，Delphi 封装了操作系统 API，使得用户可以直接分配内存，而无须考虑内存具体在哪个虚地址空间、或者内存页中。

使用 Delphi 的开发人员不必了解任何有关操作系统内存分配的细节。对于这些开发者来说：任何一块(既定长度的)内存都是连续的，可以通过字节序遍历一个数据结构，或者在长度边界相容的情况下进行强制转换。

Delphi 通过图 7.1 所示的内存管理器框架对 API 实行封装：

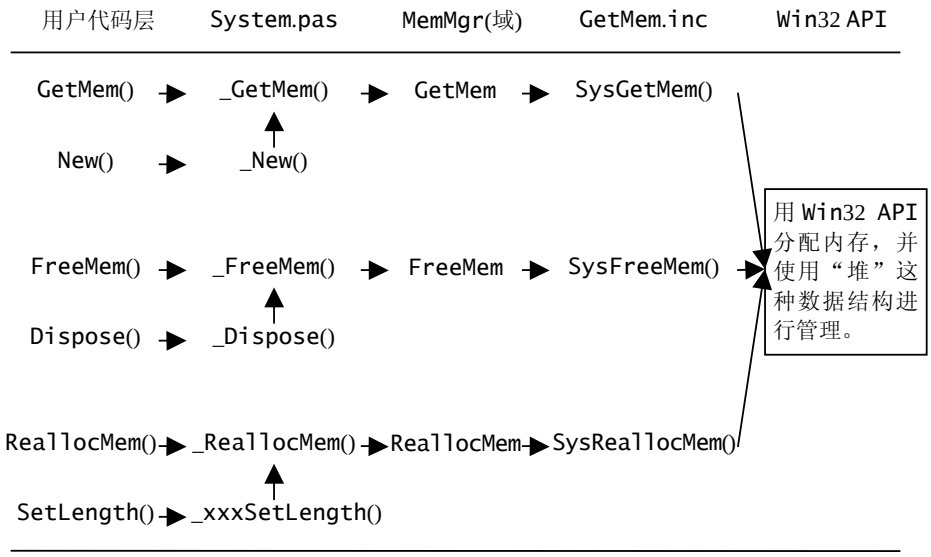

图 7.1　Delphi 内存管理器实现框架

在实现框架中，存在一些例外。例如_WStrSetLength()实现时，就不是通过调用_ReallocMem()来实现内存重分配的，而是通过_NewWideString()例程，调用

① 栈分配的相关内容请参见第 2 章和第 3 章。

② 除非特别指明，以下叙述的“在堆上分配内存”，都是指Delphi的“堆”内存管理方法，而不是操作系统的堆分配。

oleaut32.dll 中的例程 SysAllocStringLen()来实现。

7.2 内存页管理

内存页管理并非 Delphi 需要实现的管理机制，它是由操作系统实现的。

在 Win32 操作系统中，内存以“页”为单位进行分配。但内存页与内存虚地址空间是分开管理的。这表明，至少需要两步操作，才能得到一个可合法存取的内存块：

- 从整个虚地址空间中保留一个区间(Reserve)。
- 将一个内存块提交到虚地址空间上(Commit)。

这两步操作都使用同一个 API 来完成：

```
function VirtualAlloc(lpAddress: Pointer;
  dwSize, flAllocationType, flProtect: DWORD): Pointer; stdcall;
  external kernel name 'VirtualAlloc';
```

参数 flAllocationType 决定了进行哪一步的操作。其可能的取值为：

- MEM_RESERVE
  不分配任何内存空间，仅保留进程的虚地址空间的一个区间。

- MEM_COMMIT
  为指定的页范围分配物理存储[①]。具体是从物理内存或者是磁盘交换文件中分配，是由操作系统决定的。

另一个重要的参数是 lpAddress。其可能的取值为：

- nil
  在保留(Reserve)虚地址空间时，系统自行决定一个未分配的地址空间。

- 一个虚地址指针
  在保留(Reserve)虚地址空间时，以该值为参考，将地址舍入到下一个 64K 字节边界上。在提交(Commit)内存时，以该值为参考，将所分配的页舍入到下一个页边界。这意味着实际分配的页，将是能够包含 lpAddress .. DWORD(lpAddress) + cbSize

① 与大多数人所想象的不一样，Win32 API中所说的物理存储(physical storage)，并不是机器内存插槽中的那些芯片(physical memory)。这里所说的物理存储，仅是相对于虚拟地址空间而言的。它实际上是指由物理内存、磁盘交换文件(*PageFile.sys*)所组成的系统可用的物理内存空间。

范围的所有页。

依据操作系统的内存页管理机制，保留虚地址空间和提交内存都不一定正好是用户请求的地址和长度。因此，VirtualAlloc()例程将保证实际的分配大于或者等于用户所请求的虚地址空间或内存页。

正是因为这样的问题，所以在内存管理器实现中，将不得不随时修正堆块(Heap Block)的内存块边界。以充分利用 VirtualAlloc()提交的内存页上的未用空间。

7.3　堆

这里简要地说一下“堆”这种数据结构。

堆的主要构成：堆中包含许多大小不确定的块(Block)。

堆的主要操作：从堆中获取一个(至少为)指定大小的块；将一个空闲的块回收到堆；合并一些堆中的块以得到更大的块。

初始状态下，堆仅有一个块，即堆本身。经过一段时间地取用和回收以后，堆中将可能只剩下一些“切割”后残余的“碎片”，且这些碎片可能已经无法再合并。此时，如果一个新的请求大于任何一个碎片，那么就必须再申请一个新的、大的块放在堆中。

堆的使用，永远是一个“拆东墙补西墙”的过程。如果东墙拆得没有了，就去搬城砖。最终的结果是西墙不倒，或者城砖搬尽。

在最后一种情况下，Delphi 提示“Out of memory”。

7.4　MemoryManager 及相关例程

在 *System.pas* 中，MemoryManager 是一个有初值的记录变量。其定义如下：

```
var
  MemoryManager: TMemoryManager = (
    GetMem: SysGetMem;
    FreeMem: SysFreeMem;
    ReallocMem: SysReallocMem);
```

该变量初值状态为：记录的三个域分别指向 SysGetMem()、SysFreeMem()和

SysReallocMem()。根据Delphi对有初值变量的处理原则[①]，在开始运行应用程序代码之前，内存管理器实际已经指向了有效的处理例程。这个动作发生在操作系统调用模块入口代码之前，当然也在单元初始化之前。

MemoryManager 是系统内部变量，不能够直接访问。*System.pas* 中提供三个与 MemoryManager 变量相关的例程：

```
procedure GetMemoryManager(var MemMgr: TMemoryManager);
procedure SetMemoryManager(const MemMgr: TMemoryManager);
function IsMemoryManagerSet: Boolean;
```

例程 GetMemoryManager()简单地复制 MemoryManager 记录的一个备份，而 SetMemoryManager()修改 GetMemoryManager()的出口变量 MemMgr 的域值，不会影响 MemoryManager 变量。

替换缺省内存管理器的惟一方法，就是调用例程 SetMemoryManager()。该例程只是简单地将用户的 MemMgr 复制到 MemoryManager 记录。这样的实现方法可以保证开发人员不能直接地操作 MemoryManager 变量。

Delphi 使用 GetMemoryManager()和 SetMemoryManager()这两个例程来实现 MemoryManager 的存取，仅是出于安全的考虑。

例程 IsMemoryManagerSet()用于检测是否有第三方的内存管理器设置：逐一比较 MemoryManager 的各个域，如果任意一个与初始不同，则表明已经修改了预设的内存管理器，返回 True。

据官方文档，如果需要替换内存管理器，则应当在调用 IsMemoryManagerSet()之前检测当前系统已分配内存的值。这是两个系统全局变量：

```
var
  AllocMemCount: Integer;   { 已分配内存的块数             }
  AllocMemSize: Integer;    { 已分配内存块的总长度(字节数)  }
```

下面这样的内存管理器替换代码是安全的：

```
var
  NewMemMgr, OldMemMgr : TMemoryManager;

  // ...
```

① 参见PE结构中有关DATA节的描述。

```
// 设置NewMemMgr的域到有效的内存管理例程

GetMemoryManager(OldMemMgr);
if (AllocMemSize = 0) then
  SetMemoryManager(NewMemMgr);
```

大多数开发人员并不确知检测 AllocMemSize 的重要性。事实上，不检测 AllocMemSize 将导致极为严重的后果。

替换内存管理器的基本原则是：不能使用一个管理器例程去处理另一个(未知)管理器所分配的内存。如果 AllocMemSize<>0，则表明缺省的内存管理器已经启动，已经有内存被分配并存在于系统之中。此时替换到新内存管理器，则新内存管理器无法确知：后续代码是否会在切换回缺省内存管理器之前，尝试释放该内存块。因而这样的系统存在安全隐患。

一个简单的实例是：如果有字符串分别在新、旧两个内存管理器环境中被添加到 TStringList，那么在任何一个环境中调用 Free()，都会导致系统崩溃。

除非能够确知被替换的内存管理器的所有细节，标识出它所分配的内存并在需要的时候调用到 OldMemMgr 来处理，否则应当总是在检测 AllocMemSize 后替换内存管理器。

7.5　GetMem.inc 中的重要例程

MemoryManager 的三个域值例程实际位于 *GetMem.inc* 中。考虑到多线程环境，它们都使用了临界区变量 heapLock。因此在多线程环境中，Delphi 缺省的内存管理器的性能会相对较差一些。

GetMem.inc 首先针对“堆”这种数据结构，实现了堆块的操作例程。然后基于堆，使用三个层次结构来实现内存管理器，这包括：

- 虚地址空间(Address space)管理。
- 已提交的内存空间(Committed space)管理。
- 用户调用例程(actually calls)的实现。

7.5.1 堆块及其管理例程

■ 堆块：TBlockDesc(块描述)记录

每个堆块是某个双向链表的结点记录，数据类型定义为：

```
type
  PBlockDesc = ^TBlockDesc;
  TBlockDesc = packed record
    next: PBlockDesc;
    prev: PBlockDesc;
    addr: PChar;   // 并不表明这是一个PChar字符串，使用PChar类型仅为了方便地址运算
    size: Integer;
  end;
```

每 100 个TBlockDesc作为一个数据被存放在TBlockDescBlock中[①]，然后定义了基于TBlockDescBlock的单向链表blockDescBlockList。

另一个结构是 blockDescFreeList。这是一个依赖 TBlockDesc.next 维护的单向链表，结点的其他域均未使用。更确切地说，这是一个空闲块描述结点的后入先出队列(LIFO)。

blockDescFreeList 总是指向表的头部。如果 blockDescFreeList=nil，则表明系统初始化或者系统所有申请的 n*100 个块已用尽，不再有空闲的 TBlockDesc。此时，例程 GetBlockDesc()将再次申请 TBlockDescBlock 结构，这意味着 100 个新的 BlockDesc 在进程的本地堆中产生：

```
// code from GetMem.inc, GetBlockDesc().

// ...
    bdb := LocalAlloc(LMEM_FIXED, sizeof(bdb^));
    if bdb = nil then begin
      result := nil;
      exit;
    end;
```

GetBlockDesc()接下来将这个 TBlockDescBlock 插入到 blockDescBlockList 链表的头部。然后，将初始化 *TBlockDescBlock.data*，使得该数组的初始情况为：

- *data[0].next* 等于 nil。

① 定义TBlockDescBlock的原因是有序表的访问会比链表访问更快。但这会导致SizeOf(TBlockDesc) * 100个字节的开销。

- *data[1..99].next* 指向上一个 BlockDesc，即 data[0..98]。

结束初始化循环的时候，blockDescFreeList 正好指向 data[99]，它将被 GetBlockDesc()作为返回值。这表明 Delphi 是以逆序访问 TBlockDescBlock.data 数组，取得空闲的 BlockDesc 结点的。

GetBlockDesc()在退出例程前，将返回结点的 next 域赋给 blockDescFreeList。显然，如果返回结点正好是 data[0]，则 blockDescFreeList = data[0].next，亦即是指向 nil。这使得下次调用 GetBlockDesc()时，将开始另一轮分配 100 个 BlockDesc 结点的循环。

```
function GetBlockDesc: PBlockDesc;
//...
  if blockDescFreeList = nil then begin
    // 分配100个BlockDesc元素的数组，并初始化
    // blockDescFreeList指向最末一个BlockDesc，即data[99]
  end;
  bd := blockDescFreeList;
  blockDescFreeList := bd.next;
  result := bd;
end;
```

■ 非空闲的块描述结点

取得 BlockDesc 并不表明一个堆块的内存已分配。BlockDesc 仅用于记录堆中内存块或地址空间的相关信息。BlockDesc 也可以回收和合并，并总是以如下规则维护已用块描述链表：

- 若链表只有一个节点，则该节点的 prev 和 next 均指向自身。
- 若有 n 个节点，则：
  Node[n].prev->Node[n-1];
  Node[n].next->Node[n+1]。
  特例：
  Node[0].prev->Node[n];
  Node[n].next->Node[0]。

很明显，这构造了一个环形的双向链表。*GetMem.inc* 中用于维护这样的链表的例程有：

```
// 链表清空，使之初始化为只有一个(块描述)结点的链表
procedure MakeEmpty(bd: PBlockDesc);
```

```
// 为b所指向的一个内存块建立一个块描述, 并将块描述插入到prev块描述结点之后
function AddBlockAfter(prev: PBlockDesc; const b: TBlock): Boolean;

// 从链表中取出指定块描述, 然后将它插入到空闲块列表blockDescFreeList的头部
procedure DeleteBlock(bd: PBlockDesc);

// 从一个节点开始遍历链表, 试图将表中与内存块b相邻的块合并到b中.
// 如果能够合并, 则调用DeleteBlock(), 将被合并的块描述符移到blockDescFreeList.
// 完成(尝试)合并操作后, 将调用AddBlockAfter()将结果内存块b插入到prev块描述结点之后.
// 以下情况返回无效结果值result.addr = nil
//   1. 无法获取一个可用的块描述结点, 因而无法将内存块b插入到prev块描述结点之后
function MergeBlockAfter(prev: PBlockDesc; const b: TBlock) : TBlock;

// 从一个节点开始遍历链表, 检测每个块描述节点指定的地址空间是否能包含内存块b
// 如果内存块b被包含在某一节点(指定的范围)中, 则从块描述节点中截去内存块b的地址空间
// 以下情况之一返回false:
//   1. 从链表中截去内存块b的地址空间后, 剩余部分不能正确地插入到链表中
//   2. 无法在链表中找到符合条件的块描述结点
function RemoveBlock(bd: PBlockDesc; const b: TBlock): Boolean;
```

双向环形链表的特点是：无须计数链表的总长度，从任意结点(ANode)依序(next)访问结点，直到 Node.next 指向 ANode，则表明整个链表被遍历(以 prev 顺序访问的规则类同)。因此在例程 MergeBlockAfter()和 RemoveBlock()中，使用类似如下的代码，从入口参数 prev 结点开始遍历链表：

```
function MergeBlockAfter(prev: PBlockDesc; const b: TBlock) : TBlock;
var
  bd, bdNext: PBlockDesc;
begin
  bd := prev.next;
  repeat
    bdNext := bd.next;

    //...
    // 如果内存块b与bd.addr是地址相邻的, 则合并

    bd := bdNext;
  until bd = prev;  // 如果bd=prev, 则表明使用next序完成遍历

//...
```

用于保存块描述的双向环形链表在 *GetMem.inc* 中共有三个：

```
  spaceRoot: TBlockDesc;        // 已分配的内存虚地址空间, 其空间大小等于后两者的和
  decommittedRoot: TBlockDesc;  // 未提交的物理内存大小(不能使用)
```

```
  committedRoot: TBlockDesc;    // 已提交的物理内存大小(可用)
```

大多数情况下，*GetMem.inc* 使用上述变量的地址来作为链表的头部指针，并展开遍历。例如在 GetCommitted()例程中：

```
  if MergeBlockAfter(@decommittedRoot, result).addr = nil then begin
    //...
  end;
```

任何一个 TBlockDesc 记录被创建之后就不会被释放，它总是存在于上述三个环形链表和 blockDescFreeList 单向链表四者之一中。

除了上述三个环形链表的首结点之外，所有的 TBlockDesc 记录都通过例程 GetBlockDesc()在本地堆中创建。所以，系统总的块描述符结点个数为 n*100+3 个。

TBlockDesc 结构及其相关的(对四个链表的)管理与内存分配本身是无关的。*GetMem.inc* 中仅用它们来管理内存地址，这包括“虚地址空间”和“提交的内存地址”两个方面。至于真正的内存分配，是在第三层“用户调用例程”中实现的。

7.5.2　虚地址空间(Address space)管理

GetMem.inc 中实现了如下五个虚地址空间管理例程：

```
// 调用VirtualAlloc()保留虚地址空间
function GetSpace(minSize: Integer): TBlock;

// 调用VirtualAlloc(), 从指定虚地址开始保留虚地址空间
function GetSpaceAt(addr: PChar; minSize: Integer): TBlock;

// 释放虚地址空间
function FreeSpace(addr: Pointer; maxSize: Integer): TBlock;

// 提交在(addr, minSize)地址区间中的所有块(所指定的页区域)
function Commit(addr: Pointer; minSize: Integer): TBlock;

// 收回提交在(addr, minSize)地址区间中的所有块(所指定的页区域)
function Decommit(addr: Pointer; maxSize: Integer): TBlock;
```

结果值 Result.addr<>0，表明调用 GetSpace()和 GetSpaceAt()成功，且返回的 Result 块已经通过调用 AddBlockAfter()例程添加到 spaceRoot 链表中。但 Result.Size 并不一定总等于 minSize 值。Delphi 根据如下的对齐规则来设定虚地址空间分配的实际值：

- 如果 minSize 小于 cSpaceMin(1M Bytes)，则设为 1M Bytes；
- 如果 minSize 大于 1M，则在最小能容纳 minSize 字节的 cSpaceAlign(4K Bytes)边界上对齐。

FreeSpace()例程在 spaceRoot 链表中查找能够包含参数“addr, maxSize”所指定地址区间的块，并调用 VirtualFree()释放其保留的空间。虚地址空间释放后，即调用 DeleteBlock()将查找到的块从 spaceRoot 链表中清除。例程的返回值 Result 保存实际释放空间的起始地址和长度。

Commit()和 Decommit()都将进行边界对齐运算，提交和回收的值都是边界对齐后的地址和长度。返回值是实际操作的地址区间。

7.5.3 已提交的内存空间(Committed space)管理

对于已使用 Commit()提交的空间，将使用如下例程进行管理：

```
// 取至少为minSize长度的已提交块
function GetCommitted(minSize: Integer): TBlock;

// 取以addr为开始地址的，最小长度为minSize的已提交块
function GetCommittedAt(addr: PChar; minSize: Integer): TBlock;

// 释放已提交的地址空间
function FreeCommitted(addr: PChar; maxSize: Integer): TBlock;
```

以上三个例程实际上都是以 cCommitAlign 边界对齐值提交或释放。

GetCommitted 操作将首先遍历 decommittedRoot 链表，如果不能找到可供提交的空间，则调用 GetSpace()来保留一个新的地址空间。

由于调用 GetSpace()时，使用类似这样的代码：

```
  result := GetSpace(minSize);
  if MergeBlockAfter(@decommittedRoot, result).addr = nil then begin
    // 若地址空间无法登记到decommittedRoot, 则调用FreeSpace()释放它.
    FreeSpace(result.addr, result.size);
    result.addr := nil;
    exit;
  end
  else
    // ok. 重新进行GetCommitted操作.
    // ...
```

在 GetSpace()例程中，一个新的地址空间总是被加入到 spaceRoot 链表。因此在 MergeBlockAfter()调用后，未提交的地址空间会在 decommittedRoot 和 spaceRoot 两个链表中存在重叠。而一个无法合并到 decommittedRoot 链表的空间将被立即释放——这通常是因为无法申请到新的 BlockDesc。

在 GetCommitted()和 GetCommittedAt()中使用这样的代码来完成提交：

```
function GetCommitted(minSize: Integer): TBlock;
var
  bd: PBlockDesc;
//...
    result := Commit(bd.addr, minSize);
    if result.addr = nil then
      exit;
    Inc(bd.addr, result.size);
    Dec(bd.size, result.size);
    if bd.size = 0 then
      DeleteBlock(bd);
//...
```

这使得提交以后，在 decommittedRoot 链表的块描述结点 bd 中，将剩下未被提交的高位地址空间。如果 bd.size = 0，表明相应的地址空间已经全部提交，则将其从链表中删除(使块描述空闲)。

在例程 FreeCommitted()中，如果地址空间成功释放，则这个空间将被合并到 decommittedRoot 链表中。

7.5.4　用户调用例程(actually calls)的实现

如果地址空间被保留和提交，则意味着 Delphi 内核已经向操作系统成功地申请内存。接下来，*GetMem.inc* 中的大量代码用于实现 Delphi 内部的内存分配机制。

■　内存边界对齐与实际分配长度

Delphi 向操作系统申请一个内存块的实际长度，总是大于用户代码中申请的值。除了内存边界对齐的因素之外，还因为 Delphi 需要使用一个 TFree 结构来保存内存块的大小和使用状态。因此如果申请的长度不能够容纳 TFree 结构，则将至少分配 SizeOf(TFree)字节。这使得该内存块被释放时，能够作为一个 TFree 记录被缓存。

但是这并不是内存块的实际状态。如果一个内存块被申请并使用，那么该内存块将会包含

一个 TUsed 结构。这个结构用来标识内存块的大小，以使内存块回收的时候，可以辨识是缓存还是调用 API 释放物理内存。这两个结构定义如下：

```
type
  PFree = ^TFree;
  TFree = packed record
    prev: PFree;
    next: PFree;
    size: Integer;
  end;
  PUsed = ^TUsed;
  TUsed = packed record
    sizeFlags: Integer;
  end;
```

对于任何一个已分配的内存块来说，一个 TUsed 结构总是被放其头部。举例来说，一个“GetLength(Str)”为 10 的字符串，其内存占用的实际大小为：

```
  StrSize   = 引用计数位 + 长度计数位 + 10Bytes + NULL结束符
            = SizeOf(Integer) * 2 + 10 + 1
            = 19
```

因此将调用 SysGetMem(19)来分配内存，在 SysGetMem 中加上 TUsed 长度：

```
AllocSize   = TUsed结构 + 19
            = 4 + 19
            = 23
```

但这仍然不是最终向操作系统申请的内存块长度。因为接下来还要进行边界对齐运算：

```
// 对齐值cAlign = 4
AlignSize   = (23 + (cAlign-1)) and not (cAlign-1)
            = (23 + 3) and not 3
            = 24
```

按照这样的内存分配规则，可以写出实际分配长度的计算函数，也可以从内存块指针的负偏移上得到这个长度值：

```
// define in GetMem.inc
type
  PUsed = ^TUsed;
  TUsed = packed record
    sizeFlags: Integer; // 必须通过运算才能得到实际长度
  end;
```

```
const
  cAlign        = 4;
  cThisUsedFlag = 2;
  cFillerFlag   = Integer($80000000);
  cPrevFreeFlag = 1;
  cFlags        = cThisUsedFlag or cPrevFreeFlag or cFillerFlag;

// 取指针所指向内存块的对齐长度
function GetAlignSize(const p : pointer) : integer; overload;
var
  u : PUsed;
begin
  u := p;
  dec(u);
  result := u.sizeFlags and not cFlags;
end;

// 取字符串的内存占用的对齐长度
function GetAlignSize(const Str : string) : integer; overload;
var
  p : PInteger;
begin
  p := Pointer(Str);
  dec(p, 2);
  Result := GetAlignSize(Pointer(p));
end;

// 计算对齐长度
function CalcAlignSize(Size : Integer) : Integer;
begin
  Result := (Size + SizeOf(TUsed) + (cAlign-1)) and not (cAlign-1);
  if Result < 12 then  // SizeOf(TFree) = 12
   Result := 12;
end;

// ...
begin
  GetMem(p, 10);
  Writeln('直接获取: ', GetAlignSize(p));
  Writeln('  计算值: ', CalcAlignSize(10));

  Str := '0123456789';
  Writeln('直接获取: ', GetAlignSize(Str));
  Writeln('  计算值: ', CalcAlignSize(SizeOf(Integer)*2 + Length(Str) + 1));
//...
```

在上例中，*TUsed.sizeFlags* 按照如图 7.2 所示的格式存储，从该域提取内存长度值时，依赖于一些标志位的运算。

GetMem.inc 中通常使用 BlockSize 来表明对齐长度(AlignSize)。基于 Delphi 的内存实现机制，将无法确切地知道对齐前的长度值。

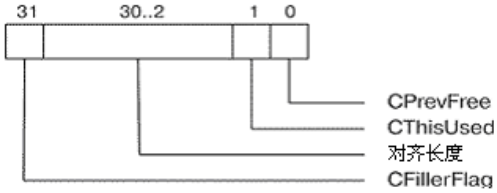

图 7.2 TUsed.sizeFlags 的存储格式

■ 内存块运行期状态

如前所述，用户每次申请的内存块会在负偏移处有一个 TUsed 结构，这使得整个内存块大小(BlockSize)为“SizeOf(TUsed) + AllocSize”的对齐值。

一个被释放的内存块除了头部被修改成 TFree 结构之外，它的尾部也会被修改成一个 TFree 记录。这使得不同状态的内存块在运行期结构如图 7.3 所示。

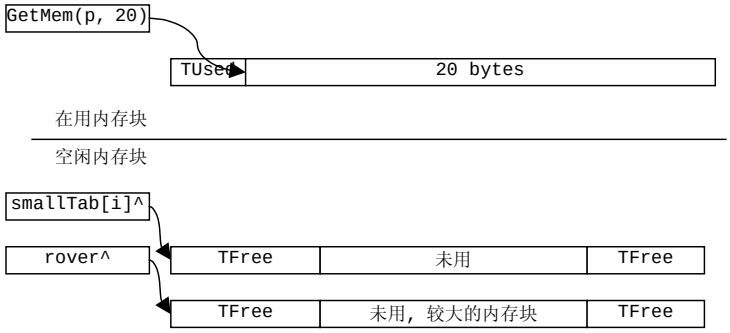

图 7.3 不同状态下的内存块结构

在目前仅见的一些关于内存管理器的资料中[①]，空闲内存块尾部被描述成TUsed结构。这并不正确。

① 截止成书之日(2004.06.23)，我能查到的惟一一份资料是俄文的，它发表在《RSDN Magazine #2》(2003.02.21)。我当然不会俄文，但是依据它所提供的图例来看，它对空闲内存块尾部结构的描述是有误的。

cFillerFlag 也同时置位。

- G2：仅 cThisUsedFlag、cFillerFlag 标志置位。是用于填充的普通 Gap。

在 FillAfterGap()例程中，如果 GapSize >= 12，那么使用如下代码填写在用内存块头部的 TUsed 结构：

```
PUsed(a).sizeFlags := size or cThisUsedFlag;
```

这样，cPrevFreeFlag 位缺省值就是 0，这使得接下来的 InternalFreeMem()操作中，不会试图向前合并。同样的原因，必须考虑：如果 Gap 足够小，则将来需要释放 AfterGap 之后的 Next Block 时，也不能试图向前合并。因此 FillAfterGap()使用如下的代码来强制填 Next Block 的 sizeFlags 域：

```
// 下一个内存块的cPrevFreeFlag取消置位
Inc(a,size);
PUsed(a).sizeFlags := PUsed(a).sizeFlags and not cPrevFreeFlag;
```

■ 内存分配(GetMem)

开发人员总有一些固有的操作内存块的习惯。例如下面的这样一段代码：

```
procedure TestMemAlloc;
var
  Str : string;
begin
  Str := '0123456789';
  // ...
end;
```

在这个例程中，需要在堆上分配块长度为 24 字节的内存。但是例程结束时，如果字符串Str没有被引用，那么它所占的内存就将被立即释放。由此可见，在应用程序中，“释放一个刚分配的内存块”的概率是很高的[①]。然而如果每次释放都将它放到smallTab或者avail链表，然后需要时再查找，则效率将会变得很低。

因此，curAlloc 指针被用来保存当前分配的一个内存块：

```
VAR
  remBytes      : Integer;      // curAlloc指向内存块的大小
```

① 这样的情况经常发生在循环调用例程或者调用递归过程中。

在上例中，*TUsed.sizeFlags* 按照如图 7.2 所示的格式存储，从该域提取内存长度值时，依赖于一些标志位的运算。

GetMem.inc 中通常使用 BlockSize 来表明对齐长度(AlignSize)。基于 Delphi 的内存实现机制，将无法确切地知道对齐前的长度值。

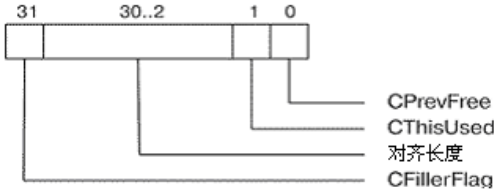

图 7.2 TUsed.sizeFlags 的存储格式

■ 内存块运行期状态

如前所述，用户每次申请的内存块会在负偏移处有一个 TUsed 结构，这使得整个内存块大小(BlockSize)为“SizeOf(TUsed) + AllocSize”的对齐值。

一个被释放的内存块除了头部被修改成 TFree 结构之外，它的尾部也会被修改成一个 TFree 记录。这使得不同状态的内存块在运行期结构如图 7.3 所示。

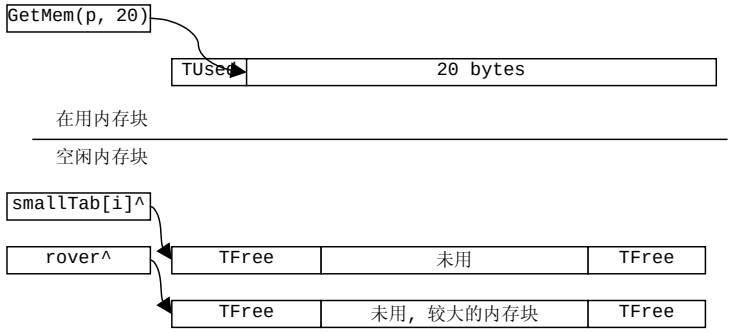

图 7.3 不同状态下的内存块结构

在目前仅见的一些关于内存管理器的资料中[①]，空闲内存块尾部被描述成TUsed结构。这并不正确。

① 截止成书之日(2004.06.23)，我能查到的惟一一份资料是俄文的，它发表在《RSDN Magazine #2》(2003.02.21)。我当然不会俄文，但是依据它所提供的图例来看，它对空闲内存块尾部结构的描述是有误的。

GetMem.inc 中所理解的 *TFree.size* 与 TUsed.sizeFlags 是不同的。前者是一个真实的长度值，后者却包含了三个标志位，需要使用 sizeFlags and not cFlags 运算才能得到有效的长度值。但在内存块末尾的数据结构的长度值并不是 sizeFlags，而是确切的长度值，因此它是 TFree 结构。

但是这样将导致一个更难以理解的问题：如果尾部是 TFree 结构，那么要容纳首尾两个 TFree，则块长度至少需要 12*2 个字节，而实际最小的块长度 CalcAlignSize(1)仅为 12 个字节。

对于长度为 12 字节的块，它首、尾的两个 TFree 结构是重叠的，因此无须讨论。而接下来的对齐长度为 16 字节，这便是尾部结构被误解为 TUsed 的原因了。其实，尾部 TFree 结构的 prev 和 next 两个值是根本不用的，所以在对齐长度为 16、20 字节的情况下，尾部与首部的 TFree 的部分域重叠，却也不会冲突。——这里，对结构的不同理解都是为了方便地址运算。

■

在内存块的尾部添加一个 TFree 的目的，是使得 Delphi 可以通过尾部的 TFree.size 来查找到头部的 TFree。这样，在释放一个内存块时，GetMem.inc 中的 SysFreeMem()就可以根据下面的规则来实现向前合并：

- 如果当前内存块的 TUsed.sizeFlags 的 cPrevFreeFlag 置位，则低地址端相邻的内存块是空闲的；
- 从(低地址端)相邻的内存块尾部的 TFree 结构中取得 size 值；
- 查找到(低地址端)相邻内存块头部的 TFree 结构；
- 合并两个内存块，并试图释放合并后的内存块。

很显然，由于当前内存块的 cPrevFreeFlag 可以表明相邻的前一个内存块的状态，所以总是能安全地访问前一个内存块。但是除了向前合并之外，在释放内存块时，Delphi 还试图向后合并。那么 Delphi 又如何知道后一个内存块的状态呢？

后一个内存块头部的四个字节，只可能存在两种情况：

- 一个空闲内存块头部的 TFree.prev 域；
- 一个在用内存块头部的 TUsed.sizeFlags 域。

由于 TFree.prev 指向的总是另一个内存块的地址值(或 nil)，而地址值总是以 4 Bytes(cAlign)为边界对齐的。因此，如果是 TFree.prev，则其低二位为 0。对于 TUsed.sizeFlags 域来说，低二位分别为 cPrevFreeFlag 和 cThisUsedFlag 标志，那么，如果内存块是“在用的”，则 TUsed.sizeFlags 的 cThisUsedFlag 标志位为 1。因此，

一个空闲内存块被释放时，它总是被插入到当前rover指针之前，然后重置rover指针到新释放的内存块。这保证了rover指针总是指向最近释放到avail链表中的空闲内存块[①]，而距离rover指针的next序越远，则表明该内存块越早被释放。这样，在遍历avail链表时，只要以rover指针开始访问，就可以有最大的几率利用操作系统的内存缓存机制。

对于长度大于或等于 15K 的空闲内存块，InsertFree()例程将试图调用DecommitFree()，在DecommitFree()中则调用 FreeCommitted()来实现释放物理存储和地址空间的功能。

接下来的问题将因为内存分页和对齐机制变得越来越复杂。

首先，FreeCommitted()在释放已提交空间时，需要进行边界对齐运算。因此，入口时的 addr 和 maxSize 就可能不是真实释放的值。该例程返回 TBlock 类型的值来存放实际释放的 addr 和 size。如果不能成功释放，返回 Result.addr=nil，这时，InsertFree()会将这个块插入到 avail 链表中。

因此，一个(大的)内存块在释放时可能发生的情况如图 7.4 所示。

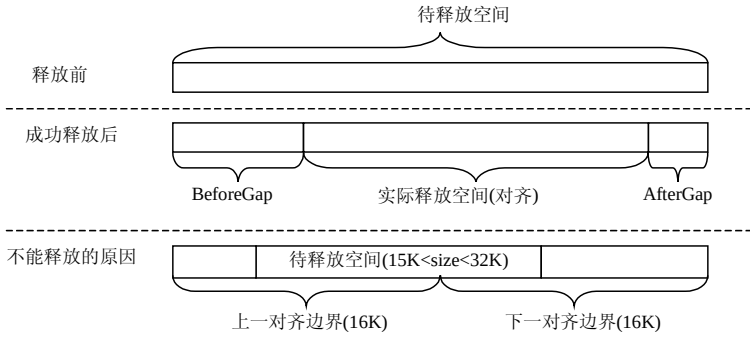

图 7.4 调用 FreeCommitted()例程的实际效果

内存管理器必须有一种机制来识别和处理这些空隙(Gap)。这就是 *GetMem.inc* 中下面两个例程的设计目的：

```
// 填前面的Gap
procedure FillBeforeGap(a: PChar; size: Integer);
```

① 基于操作系统的内存管理机制，重新访问最近操作过的内存块，总是效率最高。例如一个*pagefile.sys*中的内存页可能被缓存在物理内存中，如果重新访问它，则不必再从磁盘中读取。此外，CPU会进行高速缓存地址转换(虚地址->页面地址)，重复访问同一地址就可以不须要重新转换。

```
// 填后面的Gap
procedure FillAfterGap(a: PChar; size: Integer);
```

对于 BeforeGap：

- 如果 Gap 大小在 4~15 字节，则处理成一个 Gap 记录⑧；
- 如果 Gap 大小超过 15 字节，则处理成一个空闲内存块(TFree)，并在尾部补充一个 Gap 记录。然后调用 InsertFree()，插入到 smallTab 或者 avail 链表中。

对于 AfterGap：

- 如果 Gap 大小在 4~11 字节，则处理成一个 Gap 记录，且后一个内存块的 TUsed.sizeFlags的cPrevFreeFlag位将被清除①；
- 如果 Gap 大小超过 11 字节，则处理成一个在用内存块(TUsed)，然后调用 InternalFreeMem()例程重新释放。

因此，在空隙处理之后，原来的待释放空间将变成如图 7.5 所示的结构。

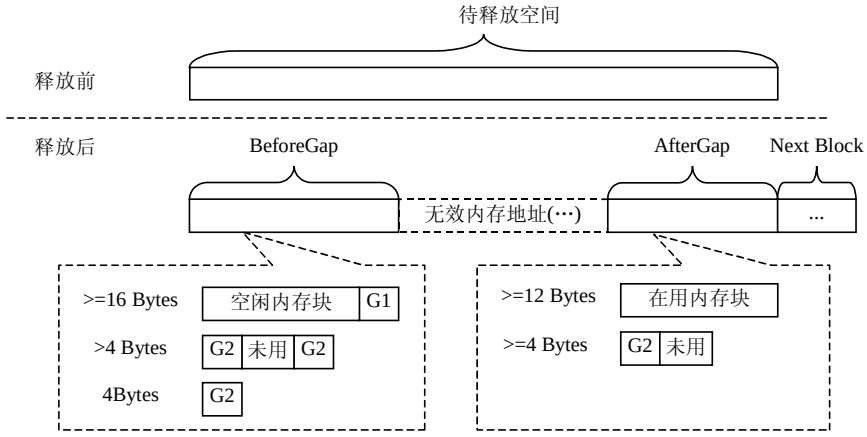

图 7.5　空隙处理后的效果

Gap 记录实际是 TUsed 结构，因此 Gap 记录也保存着 Gap 的长度值。图 7.5 中，G1 和 G2 是指不同的 Gap 类型，这主要表现在 sizeFlags 域的一些标志上：

- G1：cPrevFreeFlag 置位。这用于指明 G1 前有一个空闲内存块。cThisUsedFlag、

① 很明显，这两处存在了不可再分配的碎片。只有出现新的空闲块，并能够将它们重新合并，才有可能重新被分配或者释放地址空间。

cFillerFlag 也同时置位。

- G2：仅 cThisUsedFlag、cFillerFlag 标志置位。是用于填充的普通 Gap。

在 FillAfterGap()例程中，如果 GapSize >= 12，那么使用如下代码填写在用内存块头部的 TUsed 结构：

```
PUsed(a).sizeFlags := size or cThisUsedFlag;
```

这样，cPrevFreeFlag 位缺省值就是 0，这使得接下来的 InternalFreeMem()操作中，不会试图向前合并。同样的原因，必须考虑：如果 Gap 足够小，则将来需要释放 AfterGap 之后的 Next Block 时，也不能试图向前合并。因此 FillAfterGap()使用如下的代码来强制填 Next Block 的 sizeFlags 域：

```
// 下一个内存块的cPrevFreeFlag取消置位
Inc(a,size);
PUsed(a).sizeFlags := PUsed(a).sizeFlags and not cPrevFreeFlag;
```

■ 内存分配(GetMem)

开发人员总有一些固有的操作内存块的习惯。例如下面的这样一段代码：

```
procedure TestMemAlloc;
var
  Str : string;
begin
  Str := '0123456789';
  // ...
end;
```

在这个例程中，需要在堆上分配块长度为 24 字节的内存。但是例程结束时，如果字符串Str没有被引用，那么它所占的内存就将被立即释放。由此可见，在应用程序中，“释放一个刚分配的内存块”的概率是很高的[①]。然而如果每次释放都将它放到smallTab或者avail链表，然后需要时再查找，则效率将会变得很低。

因此，curAlloc 指针被用来保存当前分配的一个内存块：

```
VAR
  remBytes      : Integer;      // curAlloc指向内存块的大小
```

① 这样的情况经常发生在循环调用例程或者调用递归过程中。

```
  curAlloc       : PChar;         // 当前(上一次)分配的内存块指针，用于快速访问

// 释放当前(上一次)分配的内存块。有两种情况会导致CurAlloc的释放:
//   1. 当前请求的内存块大于CurAlloc，需要重新提交内存时;
//   2. 释放内存时，合并后的CurAlloc大于cDecommitMin。
// 释放操作实际上是将块插入到samllTab、avail或者调用DecommitFree()释放。
procedure FreeCurAlloc;
```

这样，有内存块释放时，先试图合并到这个内存块上，直到它大于 15K，才调用 FreeCurAlloc()例程释放其物理存储。如果有新的内存块需要分配(GetMem)，(在访问 avail 链表之前，)也总是尽可能先在 curAlloc 中进行。这使得总长度小于 15K 的连续的内存分配、释放动作变得非常高效。

用于内存分配的主要例程包括：

```
// GetMem()例程
function SysGetMem(size: Integer): Pointer;

// 从smallTab, avail链表中分配指定长度的内存，或重新向OS申请一个内存块
function TryHarder(size: Integer): Pointer;

// 在smallTab中查找至少为size字节的块
function SearchSmallBlocks(size: Integer): PFree;

// 提交新的物理存储，新提交的块总是出现在curAlloc指针上
function NewCommit(minSize: Integer): Boolean;

// 合并新提交的块到committedRoot，并将块首地址赋给curAllocr指针
function MergeCommit(b: TBlock): Boolean;
```

GetMem.inc 中，内存分配总是尽可能遵循这样的一些原则：

- 避免从大内存块中切分内存，以防止碎片产生；
- 避免访问 avail 链表，因为这是个无序的链表，访问效率较低；
- 避免调用 NewCommit()重新提交物理存储，因为这将导致内存开销；
- 在 NewCommit()中，避免调用 GetSpace()重新划分空间，因为这除了导致内存开销之外，还将导致新一个页的地址空间占用。

因此，SysGetMem()例程：

- 总是先访问 smallTab，直接查找正好满足长度需求的小内存块；如果不成功，则，
- 访问 curAlloc 指针，直接从头部截取指定长度的内存块；如果不成功，则，
- 调用 TryHarder()，访问 avail 链表，试图查找满足长度需求的内存块，切分；

如果不成功，则，

- 再次访问 smallTab，查找大于需求长度的内存块，切分；如果不成功，则，
- 调用 NewCommit()提交一个新的块，如果出错则退出 SysGetMem()；否则查找新块是否分配在 curAlloc 指针中；如果不成功，则，
- 重复上述的分配规则。

可以想象，avail 中总是比较大的块，所以通常情况下，avail 中找不到的块，在 smallTab 中也不可能找到。那么 TryHarder()例程中，为什么在访问 avail 链表之后，还要再调用 SearchSmallBlocks()来访问 smallTab 呢？这是因为存在一种特殊情况：avail 表只有一个初始结点，该结点 avail.size = 0。这时就有必要访问 smallTab 了。

在 TryHarder()中，访问 avail 链表时并没有采用这样的结构：

```
if size < cDecommitMin then
begin
  // 遍历avail链表
end;
```

这是因为在调用 InsertFree()时会进行边界对齐运算，因此会将长度不小于 cDecommitMin 的、但调用 DecommitFree()又无法释放的块插入到 avail 链表，使得 avail 链表中存在比 cDecommitMin 值更大的块。

但是，依据对齐的原理，一个块如果不小于 2*cCommitAlign，则一定会被释放(部分或全部)。因此，在 avail 中查找这样的块是没有意义的。没有对此加以排除，这是 TryHarder()例程在效率上的一个疏忽。

如果在 avail 和 smallTab 中都得不到一个可用的内存块，则 TryHarder()将调用 NewCommit()来分配新的内存块，新的内存块总是出现在 curAlloc 指针上，原 curAlloc 指针所指向的内存块，将通过调用 FreeCurAlloc()例程插入到 samllTab、avail 或者调用 DecommitFree()释放。

任何一个通过 NewCommit()得到的内存块，都将先试图与 committedRoot 链表中已提交的块合并，这是通过 MergeCommit()例程来实现的。这个合并操作与上一小节中提到的 DecommitFree()例程的机制息息相关。一个极端的例子是：应用程序先释放了一些物理存储，调用 DecommitFree()时正好释放了一个页；接下来，应用程序又需要申请一个新的页空间(如果新内存块的需求足够大)，则上一次释放的页正好被重新申请到。这时，堆中出现的情况如图 7.6 所示。

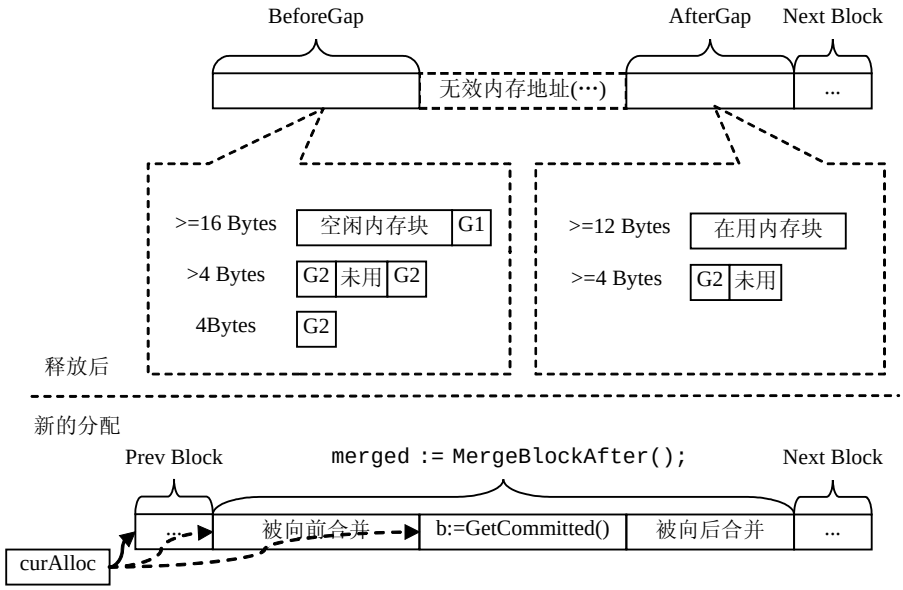

图 7.6　调用 NewCommit()的实际效果

在这个图例中，AfterGap 中的“在用内存块”最终会被调用 InternalFreeMem()释放，这使得它也将成为一个空闲内存块并被合并到 merged。

MergeCommit()例程用来调用 MergeBlockAfter()并检测合并的结果：

- merged.addr < b.addr，向前合并；
- merged.addr + merged.size > b.addr + b.size，向后合并；
- merged.addr + merged.size = b.addr + b.size，合并后块 b 的末端正好在对齐边界上(需要在末端则填一个 Gap)。

这里发生的向前、向后合并是调用 MergeBlockAfter()时，通过遍历 committedRoot 形成的。接下来调用的两个例程用来检测合并的有效性：

```
// 检测BeforeGap的有效性并试图合并Prev Block
function FillerSizeBeforeGap(a: PChar): Integer;

// 检测AfterGap的有效性并试图合并Next Block
function FillerSizeAfterGap(a: PChar): Integer;
```

除了检测合并的有效性之外，这两个例程也试图根据 BeforeGap 和 AfterGap 的 sizeFlags 域中的长度和标志信息来定位 Prev Block 和 Next Block。如果这些块是空闲的，那么这两个例程也将会进一步向前、向后合并。

调用这两个例程后并修正过 addr 和 size 值之后，块 b 的起始地址被赋值给 curAlloc，而长度值记录在 remBytes 中。至此，一块新的物理存储提交完成，它的首地址总是记录在 curAlloc 中。这存在三种可能：

- 调用 MergeBlockAfter()时未合并 BeforeGap，则 curAlloc 指向 b.addr；
- 调用 FillerSizeBeforeGap()时未合并 Prev Block，则指向 merged.addr；
- 调用 FillerSizeBeforeGap()时发生了向前合并，则指向 Prev Block。

回到 TryHarder()。由于 curAlloc 指向新分配的物理存储，因此在调用 NewCommit()后，紧接着访问 curAlloc，并从前面截取指定长度的块即可。而后续的代码，则用于处理从 avail 链表与 smallTab 中切分内存块的情况。

■ 内存再分配(ReallocMem)

内存再分配，实际上是指对一个在用的内存块进行长度调整。根据这个内存块在内存中的后续块的情况，可能有以下处理方案：

- 如果新长度小于原内存长度，则截去原内存的尾端部分，并试图回收；
- 如果新长度大于原内存长度，且后续内存块空闲，则合并后续内存块；
- 如果新长度大于原内存长度，但没有足够后续空闲内存块，则调用重新分配内存，并将原内存块中的内存复制到新内存块中。

很显然，无论新长度的大小如何，需要实现内存再分配的最简单的方法是这样的：

```
n := SysGetMem(size);
oldSize := (PUsed(PChar(p)-sizeof(PUsed)).sizeFlags and not cFlags) -
  sizeof(TUsed);
if oldSize > size then
  oldSize := size;
if n <> nil then begin
  Move(p^, n^, oldSize);
  SysFreeMem(p);
end;
result := n;
```

不过，这在 SysReallocMem()中是备选的方案。对于内存再分配来说，最消耗时间的操作是：

- 重新提交内存的 NewCommit()操作；
- 重新分配内存的 SysGetMem()操作；
- 复制原有数据的 Move()操作。

而上面的代码就占了其中两项，因此效率很低。为此，SysReallocMem()中引入了一个 ResizeInPlace()例程，它用于在指针的原有位置上，通过合并和切分内存块来实现重分配。

```
// ReallocMem()例程
function SysReallocMem(p: Pointer; size: Integer): Pointer;

// 通过截去多余内存或合并后续空闲内存块来实现内存再分配
function ResizeInPlace(p: Pointer; newSize: Integer): Boolean;
```

对于缩小空间的操作，ResizeInPlace()例程主要实现的操作是回收多余的内存空间。如果这些空间在 curAlloc 指向的负偏移处，则简单地调整 curAlloc 的指针即可；否则，先将这些空间标注为“在用的”，然后调用 InternalFreeMem()例程回收。

对于增大空间的操作来说，ResizeInPlace()则主要通过合并后续(连续)空闲内存块来获取足够的空间。如果发现后续内存块是 Gap 记录，则调用 NewCommitAt()来提交 Gap 记录后的物理存储。

仅在指定内存块后没有足够的连续空闲内存，且没有 Gap 记录(或者 Gap 记录后的能提交的空间小于请求的长度)时，ResizeInPlace()例程才返回 False，通知 SysReallocMem()使用前面所述的备选方案来实现内存再分配。

7.5.5　初始化、结束化与其他辅助例程

Delphi 内部的内存分配机制由以下两个例程来实现初始化和结束化：

```
function InitAllocator: Boolean;
procedure UninitAllocator;
```

缺省情况下，Delphi 并不初始化内部内存管理器。而是在第一次调用 GetMem()、FreeMem()或 ReallocMem()例程时，实时检测初始化状态 initialized，并调用 InitAllocator()例程。因此内核中只需要将 UninitAllocator()被加在 *System.pas* 的单元结束化节。

在 *GetMem.inc* 中，被公开的例程还有 GetHeapStatus()，这是相关的例程代码：

```
// 双向环形链表中的地址空间(TBlockDesc.size)的和
function BlockSum(root: PBlockDesc): Integer;

// 获取内存堆状态信息记录
function GetHeapStatus: THeapStatus;
```

GetHeapStatus()用于反馈 Delphi 的内存堆的状态，这是一个 THeapStatus 记录。主

要的信息包括：

- TotalAddrSpace：spaceRoot 链表中，全部保留的地址空间大小；
- TotalUncommitted：decommittedRoot 链表中，全部的未提交内存大小；
- TotalCommitted：committedRoot 链表中，全部的已提交内存大小；
- TotalAllocated：用户代码通过 GetMem()分配的内存总长度；
- TotalFree：FreeSmall + FreeBig + Unused;
- FreeSmall：smallTab 中所有空闲块的大小；
- FreeBig：avail 中所有空闲块的大小；
- Unused：curAlloc 指针所指向的内存块的大小；
- Overhead：所有 Gap 和在用内存块头部信息的总长度。

GetHeapStatus()中提供了一种遍历全部已提交内存块(即 committedRoot 链表)的方法，这事实上是对内存管理器的一些细节的总结。在下一节“遍历全部内存块”中也将使用类似的技术，源于 GetHeapStatus()的安全性，这些技术也将是安全的。

7.6 遍历全部内存块

在讨论内存管理器的替换时，曾经特别指出：由于新的内存管理器不能区分哪些内存块是原内存管理器分配的，所以应该保证替换前 AllocMemSize=0。

Delphi 的内存管理器并不提供任何例程来识别一个内存块指针是否归该内存管理器管理。这使得替换内存管理器惟一安全的方法是：将新的内存管理器单元放在项目文件的 uses 列表的头部。

编写一些调试、测试和分析软件时，可能需要列举每一个内存块并分析。因此，和编写第三方内存管理器一样，了解如何遍历 Delphi 缺省内存管理器的全部内存块，是一个必须克服的技术壁垒。

现在已经确知内存管理器包含了如下的一些结构：

- blockDescBlockList: PBlockDescBlock;
- blockDescFreeList: PBlockDesc;
- spaceRoot: TBlockDesc;
- decommittedRoot: TBlockDesc;
- committedRoot: TBlockDesc;
- smallTab: ^TSmallTab;
- avail: TFree;

- curAlloc: PChar;

一个块描述(TBlockDesc)总是存在于 blockDescBlockList 指向的结构中，而这个结构是在当前进程的堆上分配内存的，因此可以遍历进程的堆来找到全部 TBlockDescBlock 结构。接下来：

- 依据 TBlockDescBlock.Next 的关系，可以找到单向链表的第一个结点；
- 收集 TBlockDescBlock.Data 域，可得到包含全部 TBlockDesc 的列表。

一个块描述要么是未用的，要么就存在于 spaceRoot、committedRoot 和 decommittedRoot 三个环形链表之中。如果块描述未用，则它存在于 blockDescFreeList 指向的单向链表中。

如果从环形链表中的任何一个结点开始依序访问，那么最终会回到开始结点；而单向链表最终会找到一个 nil。因此，根据这个特点，可以将块描述分别列在一个单向链表和三个环形链表中。能确知的是，这个单向链表就是 blockDescFreeList。

接下来讨论如何区分三个环形链表。

首先，如果已确知一个已分配内存块的指针，那么这个指针必然不会存在于 decommittedRoot 中。因此，只需要用 GetMem()来获取一个内存块，并引入它的指针到三个环形链表中验证一下就可以了。

接下来，如果有一个存在于 decommittedRoot 中的无效指针，那么它必须存在于 spaceRoot 中(当然，它不会存在于 committedRoot 中)。因此，只需要从 decommittedRoot 所管理的地址空间中任意取一个指针，并引入到剩下的两个环形链表中验证，就可以找到 spaceRoot。

这里有一种特殊情况。如果所有的保留地址空间都被提交，即 decommittedRoot 链表为空，那么又如何区分 spaceRoot 与 committedRoot 呢？

在这种情况下，committedRoot 和 spaceRoot 所表示的空间是完全重叠的，尽管所包含的块描述(TBlockDesc)数量、记录域值存在不同，但并不会影响到下一步对内存块的遍历。

最后，在一个已经提交的内存存储中存在的内存块，要么是空闲的(TFree)，要么是在用的(TUsed)，或者就是 Gap 记录。而在 committedRoot 链表中遍历这些内存块的方法，在例程 GetHeapStatus()中有完整的实现代码。

由于在 committedRoot 中可以列举出全部的空闲内存块，因此是否查找剩下的一些结构

(例如 smallTab、avail 和 curAlloc)并不重要。但事实上也并不复杂：

- smallTab 是比较容易找到的，它也是在进程堆上进行分配的，其大小为 sizeof(TSmallTab)。因此也可以通过列举堆来查找。
- curAlloc 要么是提交空间的第一个块，要么就一定附在一个 Used 块之后，而且它一定是一个无效的块(没有有效的 TFree 头部)。
- 在 committedRoot 中，既不在 smallTab 中，且不是 curAlloc 的空闲内存块，就是 avail 链表中的一个结点。由于 avail 也是一个双向环形链表，所以只要遍历该链表，就可以找到 avail 指针(avail.size=0)。

上述思路的完整实现代码，请参阅随书光盘中的示例程序 MemMapper。

7.7 共享内存管理器

共享内存管理器是 Delphi 内存管理机制的一个重要组成部分。共享内存管理器由 *ShareMem.pas* 和 *Borlndmm.dll* 两个部分组成。

Delphi 的官方文档中指出：如果需要在 DLL 中导出例程，需要传递长字符串或动态数组参数，或这些类型的返回值(包括在记录和对象中使用它们)，那么需要使用到 *borlndmm.dll* 和 *ShareMem.pas* 单元。

这是因为 Delphi 中的长字符串、动态数组、变体、接口等都是使用引用计数机制来进行管理的。此外，在数组的元素、记录的域和类的字段中使用上述类型，也将会间接地与引用计数机制发生关系。

引用计数使得一块内存占用可能同时被多个变量引用，只有在所有变量都不再使用这块内存时，它才会被释放。释放这个内存块的操作，是在最后一个引用它的变量被释放的同时发生的。

然而，对于 Delphi 的编译器来说，它并不能识别一个参数是否是从 DLL 传递到 HOST 中来的(或者反之)，所以编译器也总是对它(返回值或入口参数)加入管理引用计数的代码，并按照同样的引用计数机制来管理。这样，一个在 DLL 中分配的内存块，可能在 HOST 中因为失去引用而被释放。例如：

```
// in dll
function GetStringFromDLL(Str: String) : String;
begin
  Result := Str + ' Sub String From Dll.';
```

```
end;

// in host
function GetString : String;
begin
  Result := GetStringFromDLL('Shared String.');
end;

// call GetString() from host app.
begin
  writeln(GetString);
end.
```

在 writeln()调用完成后，GetString()返回的字符串不再使用，所以 Delphi 编译器将加入一个_LStrClr()的调用来完成引用撤消(使引用计数减 1)。而_LStrClr()完成引用撤消之后，又会因为引用计数减至 0，调用 FreeMem()来释放内存。——这时，HOST 中试图释放在 DLL 中创建的字符串。因此，接下来的情况是：引发一个无效的指针操作(EInvalidPointer 异常)。

但这并不是问题的全部。因为即使是在 DLL 中传出一个 GetMem()获取的内存指针，并在 HOST 中释放它(或者反之)，也会有类似的错误。所以，问题真正的根源是在于“DLL 和 HOST 分别使用了不同的内存管理器”，至于在长字符串等数据类型中使用引用计数，不过是使得“在哪个模块中释放内存”变得更加不可预测而已。

Borlndmm.dll 可以解决上述问题。

Borlndmm.dll 的代码其实非常简单，我想是简单到了 Borland 无法公开这些代码的地步。如果要复原 *Borlndmm.dll* 的代码，大概只需要这样几行就可以了：

```
library Borlndmm;

// UNKNOWN
// procedure HeapAddRef;
// procedure HeapRelease;
procedure DumpBlocks;
begin
  // UNKNOWN
end;

function GetAllocMemCount: Integer;
begin
  Result := System.AllocMemCount;
end;
```

```
function GetAllocMemSize: Integer;
begin
  Result := System.AllocMemSize;
end;

exports
//  HeapAddRef name '@Borlndmm@HeapAddRef$qqrv',
//  HeapRelease name '@Borlndmm@HeapRelease$qqrv',
  DumpBlocks, GetHeapStatus, SysGetMem, SysFreeMem, SysReallocMem,
  System.SysGetMem name '@Borlndmm@SysGetMem$qqri',
  System.SysFreeMem name '@Borlndmm@SysFreeMem$qqrpv',
  System.SysReallocMem name '@Borlndmm@SysReallocMem$qqrpvi';

begin
  System.IsMultiThread := True;
end.
```

从原理上来说，只要有一个共用的 DLL 在其他 DLL 模块和 HOST 进程载入之前加载，并保证所有 DLL 与 HOST 进程都能使用共用 DLL(例如 *Borlndmm.dll*)中的内存分配例程，那么这个 DLL 就可以用来做共享内存管理器。

ShareMem.pas 并不包括实现内存分配例程的代码，它主要用来保证当前模块会静态装载 *Borlndmm.dll*。当然，它也包含为数不多的几行代码来调用 `SetMemoryManager()`，以使用 *Borlndmm.dll* 中的例程去替换缺省的内存管理器。

由于在每一个.DLL 或.EXE 中都引用了 *ShareMem.pas* 单元，所以 *Borlndmm.dll* 模块一定会比这些.DLL 和.EXE 更先载入内存，此后任何启动的.DLL 或.EXE 都将使用 *Borlndmm.dll* 模块中的内存管理器代码。由于无法确定后续模块中是否使用了多线程，因此 *Borlndmm.dll* 总是直接初始化 *`IsMultiThread`* 为 `True`。——关于这一点的真实细节仍不是非常清楚，也可能 *Borlndmm.dll* 中使用独立的 *GetMem.inc* 文件，而其中的代码并不检测 `IsMultiThread` 状态，就直接进入临界区。

直接使用临界区是导致 *Borlndmm.dll* 的内存管理器性能下降的罪魁祸首(极端的情况下，它速度只有缺省的内存管理器速度的 1/8)。除了在临界区使用上的细小的区别之外，*Borlndmm.dll* 中使用的代码与 *GetMem.inc* 完全一致。这也是上例中可以直接导出 *System.pas* 单元中的例程的原因。

7.8　第三方内存管理器

第三方的内存管理器是不多见的，其中包括：

- QStrings 的作者 Andrew Driazgov 发布的 QMemory 和 ShareQmm 2.01a。
- High Performance Delphi 发布的 HPMM。

其中，HPMM 尽管提出了较高性能的优化算法，却并没有实现 ShareMem 的接口。因此，HPMM 仅能在单独模块中使用(MultiMM 单元用于多线程环境)。

ShareQmm 实现共享内存管理器的思路与 Delphi 是一致的：它用一个 *Qmm.dll* 来替代 *Borlndmm.dll*。因此，使用 ShareQmm 时仍然得背上 *Qmm.dll* 的包袱。更加危险的是：没有太多人知道 *Qmm.dll* 的用途，因此被从硬盘上意外清除的可能性远比 *Borlndmm.dll* 大。

也有一些其他的专门用来替换 *Borlndmm.dll* 和 *ShareMem.pas* 的共享内存管理器。例如使用“内存桥”技术的 FastShareMem 和 ShareMemRep，它们都不须要另外加载模块，就能在 DLL 和 HOST 之间共享内存管理器。

FastShareMem 使用 API 在虚地址空间的高端，查找一个 64K 的页，然后提交它，并将一个当前的内存管理器的记录写到该地址上。在其他模块中初始化时，就可以通过列举内存页查找到这个记录，并使用它来替换当前的内存管理器。

ShareMemRep 是在 FastShareMem 的基础上发展起来的另一个 OpenSoruce 项目。不同的是，它通过导出表在 EXE 和 DLL 之间传递内存管理器地址(这个地址在编译期就被写入到导出表了)。除了内存管理器的地址之外，ShareMemRep 还导出了一个 SharedModules 记录，用来记录、管理所有被“桥接”的动态和静态模块。此外，ShareMemRep 还实现了内存管理器与其他模块间实现了 `IsMutilThread` 状态的同步，这表明 ShareMemRep 在多线程状态下也是安全的。

一旦 DLL 与 HOST 成功地建立了内存管理的“桥接”关系，那么 *Borlndmm.dll* 就不再是必需的了。因此，“内存桥”的技术核心，在于将内存管理器地址记录到某个位置，以供其他模块查找和替换。

关于 ShareMemRep 的详细实现代码及其实现机制，请参阅随书光盘中的 ShareMemRep 1.5 以及《共享 HOST 与 DLL 之间的内存管理器》一文。表 7-1 仅列出了 ShareMemRep 与 FastShareMem 之间的一些差异。

表 7-1 ShareMemRep 与 FastShareMem 之间的一些差异

解决方案 / 支持	解决方案(或程度)		备注
	FastShareMem	ShareMemRep	
共享MemMgr	记录在内存高端	通过导出表导出内存管理器地址	
多线程	不支持	可同步DLL与HOST的多线程状态	
模块识别	不能区分动态和静态模块	构造模块表,可识别静态和动态DLL载入	
静态DLL	可在静态DLL中共享MemMgr	可在静态DLL中载入MemMgr	
动态DLL	不能阻止共享其MemMgr	动态DLL不能共享它的MemMgr	
未卸载库的检测	通过引用计数识别	通过模块表识别	
异常	支持	支持	
第三方MemMgr	必须加入到第一个载入模块中	可以加入到静态模块和HOST模块中	①
替换的完整性	不完整	完整地实现了ShareMem.pas的全部接口	

7.9 小结

块描述来记录被保留的虚地址空间(spaceRoot)，以及虚地址空间中已提交(commitRoot)和未提交(decommitRoot)的部分。每次提交和取消提交都进行页边界对齐运算。

TBlockDescBlock 结构用于每次申请 100 个块描述结点。空闲的块描述被回收到 blockDescFreeList，这是一个后入先出队列(LIFO)。

被保留的虚地址空间称为“堆空间”，在堆中可能存在的块包括：已提交物理存储和未提交物理存储。如果堆空间用尽(未提交物理存储用尽)，则再保留新的虚地址空间。

已提交物理存储即是可用的内存块，包括在用、空闲和 Gap 三种状态。

在用内存块结构较为简单，使用 TUsed 记录来描述；空闲内存块结构复杂，在首尾两端存在 TFree 记录。

没有结构用于登记在用内存块，仅在 AllocMemCount 和 AllocMemSize 中保存了它们的数量和总长度。空闲的可用内存块则总是被登记在：

① Delphi的ShareMem不支持第三方内存管理器。

- smallTab 数组各元素所指向的双向环形链表；
- avail 双向环形链表；
- curAlloc 指针指向的内存块；

上述三者之一中。

已提交的物理存储，在其前后内存边界上会填充 Gap。Gap 是 cFillerFlag 为 1 的 TUsed 记录，它存在的目的，是用来标识该物理存储相邻的地址空间是未提交的，不能合并。

堆空间中的合并操作是指块描述所记录的地址空间的合并。内存块的合并操作则指的是通过 TFree 和 TUsed 记录实现的向前、向后合并。

只有在首次调用内存管理例程时才会初始化内存管理器，所以在此前替换内存管理器时并不会留下任何内存垃圾。正是因此， *ShareMem.pas* 和 *Borlndmm.dll* 在“节省内存”方面有很好的表现。但是由于 *Borlndmm.dll* 缺省时是按多线程方式来运行的，所以它的效率会差得很多。

ShareMem.pas 和 *Borlndmm.dll* 是为了实现“HOST 与 DLL 模块共用一个内存管理器”。FastShareMem 和 ShareMemRep 使用内存桥技术实现了相同的目的。与“抛弃 *Borlndmm.dll*”所带来的优点相对应的是：可能必须在一些条件限制下使用它们。