**探索 Linux 内存模型**

**理解 Linux 设计的第一步**

在这篇 Linux® 内存模型指南中，我们将学习如何构建和管理内存方面的基础知识。本指南介绍了内存控制单元、分页模型方面的内容，并详细介绍了物理内存区域方面的知识。

2https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/linux/l-memmod/#icomments)

[Vikram Shukla](https://www.ibm.com/developerworks/cn/linux/l-memmod/#authorN10019), 软件工程师, EMC

2006 年 2 月 23 日

* [expand](https://www.ibm.com/developerworks/cn/linux/l-memmod/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-l-memmod-_-sidebar)

理解 Linux 使用的内存模型是从更大程度上掌握 Linux 设计和实现的第一步，因此本文将概述 Linux 内存模型和管理。

Linux 使用的是单一整体式结构 (Monolithic)，其中定义了一组原语或系统调用以实现操作系统的服务，例如在几个模块中以超级模式运行的进程管理、并发控制和内存管理服务。尽管出于兼容性考虑，Linux 依然将*段控制单元模型 (segment control unit model)* 保持一种符号表示，但实际上已经很少使用这种模型了。

与内存管理有关的主要问题有：

* 虚拟内存的管理，这是介于应用程序请求与物理内存之间的一个逻辑层。
* 物理内存的管理。
* 内核虚拟内存的管理/内核内存分配器，这是一个用来满足对内存的请求的组件。这种对内存的请求可能来自于内核，也可能来自于用户。
* 虚拟地址空间的管理。
* 交换和缓存。

本文探讨了以下问题，可以帮助您从操作系统中内存管理的角度来理解 Linux 的内幕：

* 段控制单元模型，通常专用于 Linux
* 分页模型，通常专用于 Linux
* 物理内存方面的知识

虽然本文并没有详细介绍 Linux 内核管理内存的方法，但是介绍了有关整个内存模型的知识以及系统的寻址方式，这些介绍可为您进一步的学习提供一个框架。本文重点介绍的是 x86 架构，但本文中的知识对于其他硬件实现同样适用。

**x86 内存架构**

在 x86 架构中，内存被划分成 3 种类型的地址：

* *逻辑地址 (logical address)* 是存储位置的地址，它可能直接对应于一个物理位置，也可能不直接对应于一个物理位置。逻辑地址通常在请求控制器中的信息时使用。
* *线性地址 (linear address)* （或称为 *平面地址空间*）是从 0 开始进行寻址的内存。之后的每个字节都可顺序使用下一数字来引用（0、1、2、3 等），直到内存末尾为止。这就是大部分非 Intel CPU 的寻址方式。Intel® 架构使用了分段的地址空间，其中内存被划分成 64KB 的段，有一个段寄存器总是指向当前正在寻址的段的基址。这种架构中的 32 位模式被视为平面地址空间，不过它也使用了段。
* *物理地址 (physical address)* 是使用物理地址总线中的位表示的地址。物理地址可能与逻辑地址不同，内存管理单元可以将逻辑地址转换成物理地址。

CPU 使用两种单元将逻辑地址转换成物理地址。第一种称为*分段单元 (segmented unit)*，另外一种称为*分页单元 (paging unit)*。

**图 1. 转换地址空间使用的两种单元**

转换地址空间使用的两种单元

下面让我们来介绍一下段控制单元模型。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-memmod/#ibm-pcon)

**段控制单元模型概述**

这种分段模型背后的基本思想是将内存分段管理。从本质上来说，每个段就是自己的地址空间。段由两个元素构成：

* *基址 (base address)* 包含某个物理内存位置的地址
* *长度值 (length value)* 指定该段的长度

分段地址还包括两个组件 —— *段选择器 (segment selector)* 和*段内偏移量 (offset into the segment)*。段选择器指定了要使用的段（即基址和长度值），而段内偏移量组件则指定了实际内存位置相对于基址的偏移量。实际内存位置的物理地址就是这个基址值与偏移量之和。如果偏移量超过了段的长度，系统就会生成一个保护违例错误。

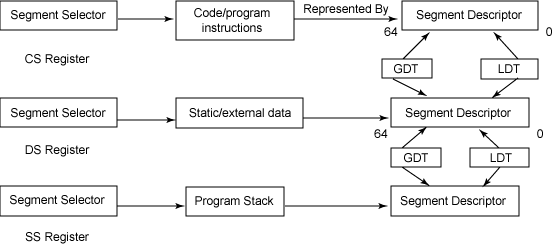
上述内容可小结如下：

分段单元可以表示成 -> 段: 偏移量 模型

也也可表示成 -> 段标识符: 偏移量

每个段都是一个 16 位的字段，称为*段标识符 (segment identifier)* 或*段选择器 (segment selector)*。x86 硬件包括几个可编程的寄存器，称为 *段寄存器 (segment register)*，段选择器保存于其中。这些寄存器为 cs（代码段）、ds（数据段）和 ss（堆栈段）。每个段标识符都代表一个使用 64 位（8 个字节）的*段描述符 (segment descriptor)* 表示的段。这些段描述符可以存储在一个 GDT（全局描述符表，global descriptor table）中，也可以存储在一个 LDT（本地描述符表，local descriptor table）中。

**图 2. 段描述符和段寄存器的相互关系**



每次将段选择器加载到段寄存器中时，对应的段描述符都会从内存加载到相匹配的不可编程 CPU 寄存器中。每个段描述符长 8 个字节，表示内存中的一个段。这些都存储到 LDT 或 GDT 中。段描述符条目中包含一个指针和一个 20 位的值（Limit 字段），前者指向由 Base 字段表示的相关段中的第一个字节，后者表示内存中段的大小。

其他某些字段还包含一些特殊属性，例如优先级和段的类型（cs 或 ds）。段的类型是由一个 4 位的 Type 字段表示的。

由于我们使用了不可编程寄存器，因此在将逻辑地址转换成线性地址时不引用 GDT 或 LDT。这样可以加快内存地址的转换速度。

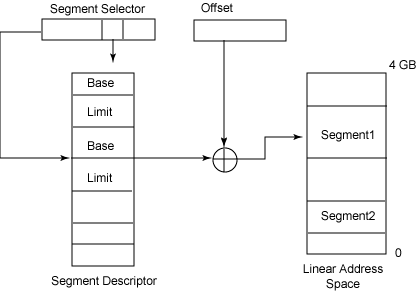
段选择器包含以下内容：

* 一个 13 位的索引，用来标识 GDT 或 LDT 中包含的对应段描述符条目
* TI (Table Indicator) 标志指定段描述符是在 GDT 中还是在 LDT 中，如果该值是 0，段描述符就在 GDT 中；如果该值是 1，段描述符就在 LDT 中。
* RPL (request privilege level) 定义了在将对应的段选择器加载到段寄存器中时 CPU 的当前特权级别。

由于一个段描述符的大小是 8 个字节，因此它在 GDT 或 LDT 中的相对地址可以这样计算：段选择器的高 13 位乘以 8。例如，如果 GDT 存储在地址 0x00020000 处，而段选择器的 Index 域是 2，那么对应的段描述符的地址就等于 (2\*8) + 0x00020000。GDT 中可以存储的段描述符的总数等于 (2^13 - 1)，即 8191。

图 3 展示了从逻辑地址获得线性地址。

**图 3. 从逻辑地址获得线性地址**



那么这在 Linux 环境下有什么不同呢？

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-memmod/#ibm-pcon)

**Linux 中的段控制单元**

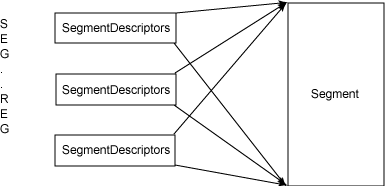
Linux 对这个模型稍微进行了修改。我注意到 Linux 以一种受限的方法来使用这种分段模型（主要是出于兼容性方面的考虑）。

在 Linux 中，所有的段寄存器都指向相同的段地址范围 —— 换言之，每个段寄存器都使用相同的线性地址。这使 Linux 所用的段描述符数量受限，从而可将所有描述符都保存在 GDT 之中。这种模型有两个优点：

* 当所有的进程都使用相同的段寄存器值时（当它们共享相同的线性地址空间时），内存管理更为简单。
* 在大部分架构上都可以实现可移植性。某些 RISC 处理器也可通过这种受限的方式支持分段。

图 4 展示了对模型的修改。

**图 4. 在 Linux 中，段寄存器指向相同的地址集**



**段描述符**

Linux 使用以下段描述符：

* 内核代码段
* 内核数据段
* 用户代码段
* 用户数据段
* TSS 段
* 默认 LDT 段

下面详细介绍这些段寄存器。

GDT 中的*内核代码段 (kernel code segment)* 描述符中的值如下：

* Base = 0x00000000
* Limit = 0xffffffff (2^32 -1) = 4GB
* G（粒度标志）= 1，表示段的大小是以页为单位表示的
* S = 1，表示普通代码或数据段
* Type = 0xa，表示可以读取或执行的代码段
* DPL 值 = 0，表示内核模式

与这个段相关的线性地址是 4 GB，S = 1 和 type = 0xa 表示代码段。选择器在 cs 寄存器中。Linux 中用来访问这个段选择器的宏是 \_KERNEL\_CS。

*内核数据段 (kernel data segment)* 描述符的值与内核代码段的值类似，惟一不同的就是 Type 字段值为 2。这表示此段为数据段，选择器存储在 ds 寄存器中。Linux 中用来访问这个段选择器的宏是 \_KERNEL\_DS。

*用户代码段 (user code segment)* 由处于用户模式中的所有进程共享。存储在 GDT 中的对应段描述符的值如下：

* Base = 0x00000000
* Limit = 0xffffffff
* G = 1
* S = 1
* Type = 0xa，表示可以读取和执行的代码段
* DPL = 3，表示用户模式

在 Linux 中，我们可以通过 \_USER\_CS 宏来访问此段选择器。

在 *用户数据段 (user data segment)* 描述符中，惟一不同的字段就是 Type，它被设置为 2，表示将此数据段定义为可读取和写入。Linux 中用来访问此段选择器的宏是 \_USER\_DS。

除了这些段描述符之外，GDT 还包含了另外两个用于每个创建的进程的段描述符 —— TSS 和 LDT 段。

每个 *TSS 段 (TSS segment)* 描述符都代表一个不同的进程。TSS 中保存了每个 CPU 的硬件上下文信息，它有助于有效地切换上下文。例如，在 U->K 模式的切换中，x86 CPU 就是从 TSS 中获取内核模式堆栈的地址。

每个进程都有自己在 GDT 中存储的对应进程的 TSS 描述符。这些描述符的值如下：

* Base = &tss （对应进程描述符的 TSS 字段的地址；例如 &tss\_struct）这是在 Linux 内核的 schedule.h 文件中定义的
* Limit = 0xeb （TSS 段的大小是 236 字节）
* Type = 9 或 11
* DPL = 0。用户模式不能访问 TSS。G 标志被清除

所有进程共享*默认 LDT 段*。默认情况下，其中会包含一个空的段描述符。这个默认 LDT 段描述符存储在 GDT 中。Linux 所生成的 LDT 的大小是 24 个字节。默认有 3 个条目：

LDT[0] = 空

LDT[1] = 用户代码段

LDT[2] = 用户数据/堆栈段描述符

**计算任务**

要计算 GDT 中最多可以存储多少条目，必须先理解 NR\_TASKS（这个变量决定了 Linux 可支持的并发进程数 —— 内核源代码中的默认值是 512，最多允许有 256 个到同一实例的并发连接）。

GDT 中可存储的条目总数可通过以下公式确定：

GDT 中的条目数 = 12 + 2 \* NR\_TASKS。

正如前所述，GDT 可以保存的条目数 = 2^13 -1 = 8192。

在这 8192 个段描述符中，Linux 要使用 6 个段描述符，另外还有 4 个描述符将用于 APM 特性（高级电源管理特性），在 GDT 中还有 4 个条目保留未用。因此，GDT 中的条目数等于 8192 - 14，也就是 8180。

任何情况下，GDT 中的条目数 8180，因此：

2 \* NR\_TASKS = 8180   
NR\_TASKS = 8180/2 = 4090

（为什么使用 *2 \* NR\_TASKS*？因为对于所创建的每个进程，都不仅要加载一个 TSS 描述符 —— 用来维护上下文切换的内容，另外还要加载一个 LDT 描述符。）

这种 x86 架构中进程数量的限制是 Linux 2.2 中的一个组件，但自 2.4 版的内核开始，这个问题已经不存在了，部分原因是使用了硬件上下文切换（这不可避免地要使用 TSS），并将其替换为进程切换。

接下来，让我们了解一下*分页模型*。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-memmod/#ibm-pcon)

**分页模型概述**

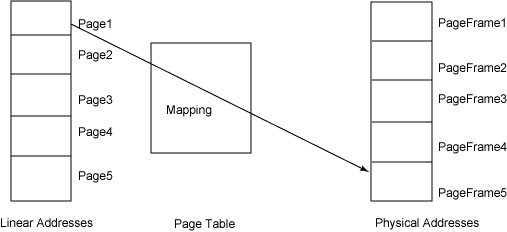
分页单元负责将线性地址转换成物理地址（请参见图 1）。线性地址会被分组成页的形式。这些线性地址实际上都是连续的 —— 分页单元将这些连续的内存映射成对应的连续物理地址范围（称为 *页框*）。注意，分页单元会直观地将 RAM 划分成固定大小的页框。

正因如此，分页具有以下优点：

* 为一个页定义的访问权限中保存了构成该页的整组线性地址的权限
* 页的大小等于页框的大小

将这些页映射成页框的数据结构称为*页表 (page table)*。页表存储在主存储器中，可由内核在启用分页单元之前对其进行恰当的初始化。图 5 展示了页表。

**图 5. 页表将页转换成页框**



注意，上图 Page1 中包含的地址集正好与 Page Frame1 中包含的地址集匹配。

在 Linux 中，分页单元的使用多于分段单元。前面介绍 Linux 分段模型时已提到，每个分段描述符都使用相同的地址集进行线性寻址，从而尽可能降低使用分段单元将逻辑地址转换成线性地址的需要。通过更多地使用分页单元而非分段单元，Linux 可以极大地促进内存管理及其在不同硬件平台之间的可移植性。

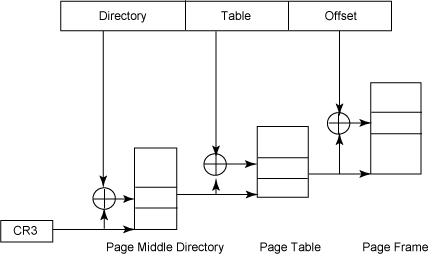
**分页过程中使用的字段**

下面让我们来介绍一下用于在 x86 架构中指定分页的字段，这些字段有助于在 Linux 中实现分页功能。分页单元进入作为分段单元输出结果的线性字段，然后进一步将其划分成以下 3 个字段：

* *Directory* 以 10 MSB 表示（Most Significant Bit，也就是二进制数字中值最大的位的位置 —— MSB 有时称为最左位）。
* *Table* 以中间的 10 位表示。
* *Offset* 以 12 LSB 表示。（Least Significant Bit，也就是二进制整数中给定单元值的位的位置，即确定这个数字是奇数还是偶数。LSB 有时称为最右位。这与数字权重最轻的数字类似，它是最右边位置处的数字。）

线性地址到对应物理位置的转换的过程包含两个步骤。第一步使用了一个称为*页目录 (Page Directory)* 的转换表（从页目录转换成页表），第二步使用了一个称为*页表 (Page Table)* 的转换表（即页表加偏移量再加页框）。图 6 展示了此过程。

**图 6. 分页字段**



开始时，首先将页目录的物理地址加载到 cr3 寄存器中。线性地址中的 Directory 字段确定页目录中指向恰当的页表条目。Table 字段中的地址确定包含页的页框物理地址所在页表中的条目。Offset 字段确定了页框中的相对位置。由于 Offset 字段为 12 位，因此每个页中都包含有 4 KB 数据。

下面小结物理地址的计算：

1. cr3 + Page Directory (10 MSB) = 指向 table\_base
2. table\_base + Page Table (10 中间位) = 指向 page\_base
3. page\_base + Offset = 物理地址 (获得页框)

由于 Page Directory 字段和 Page Table 段都是 10 位，因此其可寻址上限为 1024\*1024 KB，Offset 可寻址的范围最大为 2^12（4096 字节）。因此，页目录的可寻址上限为 1024\*1024\*4096（等于 2^32 个内存单元，即 4 GB）。因此在 x86 架构上，总可寻址上限是 4 GB。

**扩展分页**

扩展分页是通过删除页表转换表实现的；此后线性地址的划分即可在页目录 (10 MSB) 和偏移量 (22 LSB) 之间完成了。

22 LSB 构成了页框的 4 MB 边界（2^22）。扩展分页可以与普通的分页模型一起使用，并可用于将大型的连续线性地址映射为对应的物理地址。操作系统中删除页表以提供扩展页表。这可以通过设置 PSE (page size extension) 实现。

36 位的 PSE 扩展了 36 位的物理地址，可以支持 4 MB 页，同时维护一个 4 字节的页目录条目，这样就可以提供一种对超过 4 GB 的物理内存进行寻址的方法，而不需要对操作系统进行太大的修改。这种方法对于按需分页来说具有一些实际的限制。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-memmod/#ibm-pcon)

**Linux 中的分页模型**

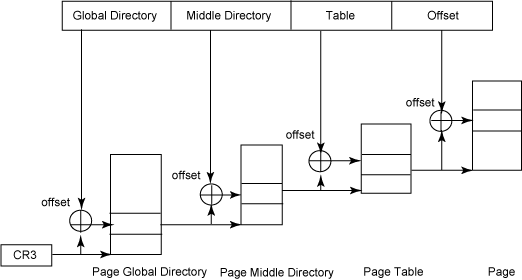
虽然 Linux 中的分页与普通的分页类似，但是 x86 架构引入了一种三级页表机制，包括：

* *页全局目录 (Page Global Directory)*，即 pgd，是多级页表的抽象最高层。每一级的页表都处理不同大小的内存 —— 这个全局目录可以处理 4 MB 的区域。每项都指向一个更小目录的低级表，因此 pgd 就是一个页表目录。当代码遍历这个结构时（有些驱动程序就要这样做），就称为是在“遍历”页表。
* *页中间目录 (Page Middle Directory)*,即 pmd，是页表的中间层。在 x86 架构上，pmd 在硬件中并不存在，但是在内核代码中它是与 pgd 合并在一起的。
* *页表条目 (Page Table Entry)*，即 pte，是页表的最低层，它直接处理页（参看 PAGE\_SIZE），该值包含某页的物理地址，还包含了说明该条目是否有效及相关页是否在物理内存中的位。

为了支持大内存区域，Linux 也采用了这种三级分页机制。在不需要为大内存区域时，即可将 pmd 定义成“1”，返回两级分页机制。

分页级别是在编译时进行优化的，我们可以通过启用或禁用中间目录来启用两级和三级分页（使用相同的代码）。32 位处理器使用的是 pmd 分页，而 64 位处理器使用的是 pgd 分页。

**图 7. 三级分页**



如您所知，在 64 位处理器中：

* 21 MSB 保留未用
* 13 LSB 由页面偏移量表示
* 其余的 30 位分为：
  + 10 位用于页表
  + 10 位用于页全局目录
  + 10 位用于页中间目录

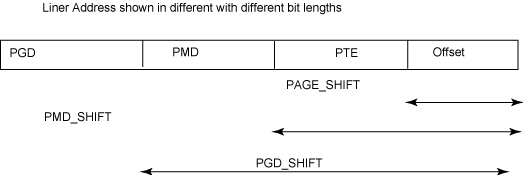
我们可以从架构中看到，实际上使用了 43 位进行寻址。因此在 64 位处理器中，可以有效使用的内存是 2 的 43 次方。

每个进程都有自己的页目录和页表。为了引用一个包含实际用户数据的页框，操作系统（在 x86 架构上）首先将 pgd 加载到 cr3 寄存器中。Linux 将 cr3 寄存器的内容存储到 TSS 段中。此后只要在 CPU 上执行新进程，就从 TSS 段中将另外一个值加载到 cr3 寄存器中。从而使分页单元引用一组正确的页表。

pgd 表中的每一条目都指向一个页框，其中中包含了一组 pmd 条目；pdm 表中的每个条目又指向一个页框，其中包含一组 pte 条目；pde 表中的每个条目再指向一个页框，其中包含的是用户数据。如果正在查找的页已转出，那么就会在 pte 表中存储一个交换条目，（在缺页的情况下）以定位将哪个页框重新加载到内存中。

图 8 说明我们连续为各级页表添加偏移量来映射对应的页框条目。我们通过进入作为分段单元输出的线性地址，再划分该地址来获得偏移量。要将线性地址划分成对应的每个页表元素，需要在内核中使用不同的宏。本文不详细介绍这些宏，下面我们通过图 8 来简单看一下线性地址的划分方式。

**图 8. 具有不同地址长度的线性地址**



**预留页框**

Linux 为内核代码和数据结构预留了几个页框。这些页*永远不会* 被转出到磁盘上。从 0x0 到 0xc0000000 （PAGE\_OFFSET） 的线性地址可由用户代码和内核代码进行引用。从 PAGE\_OFFSET 到 0xffffffff 的线性地址只能由内核代码进行访问。

这意味着在 4 GB 的内存空间中，只有 3 GB 可以用于用户应用程序。

**如何启用分页**

Linux 进程使用的分页机制包括两个阶段：

* 在启动时，系统为 8 MB 的物理内存设置页表。
* 然后，第二个阶段完成对其余物理地址的映射。

在启动阶段，startup\_32() 调用负责对分页机制进行初始化。这是在 arch/i386/kernel/head.S 文件中实现的。这 8 MB 的映射发生在 PAGE\_OFFSET 之上的地址中。这种初始化是通过一个静态定义的编译时数组 (swapper\_pg\_dir) 开始的。在编译时它被放到一个特定的地址（0x00101000）。

这种操作为在代码中静态定义的两个页 —— pg0 和 pg1 —— 建立页表。这些页框的大小默认为 4 KB，除非我们设置了页大小扩展位（有关 PSE 的更多内容，请参阅 [扩展分页](https://www.ibm.com/developerworks/cn/linux/l-memmod/#extend) 一节）。这个全局数组所指向的数据地址存储在 cr3 寄存器中，我认为这是为 Linux 进程设置分页单元的第一阶段。其余的页项是在第二阶段中完成的。

第二阶段由方法调用 paging\_init() 来完成。

在 32 位的 x86 架构上，RAM 映射到 PAGE\_OFFSET 和由 4GB 上限 (0xFFFFFFFF) 表示的地址之间。这意味着大约有 1 GB 的 RAM 可以在 Linux 启动时进行映射，这种操作是默认进行的。然而，如果有人设置了 HIGHMEM\_CONFIG，那么就可以将超过 1 GB 的内存映射到内核上 —— 切记这是一种临时的安排。可以通过调用 kmap() 实现。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-memmod/#ibm-pcon)

**物理内存区域**

我已经向您展示了（32 位架构上的） Linux 内核按照 3:1 的比率来划分虚拟内存：3 GB 的虚拟内存用于用户空间，1 GB 的内存用于内核空间。内核代码及其数据结构都必须位于这 1 GB 的地址空间中，但是对于此地址空间而言，更大的消费者是物理地址的虚拟映射。

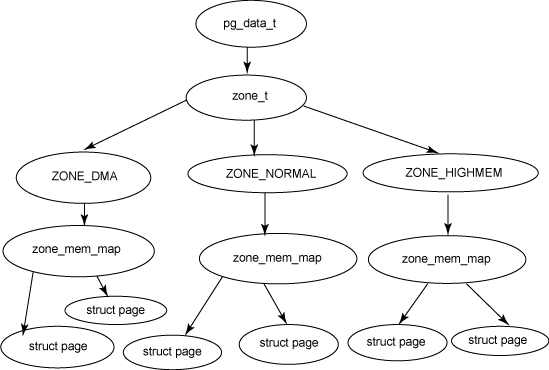
之所以出现这种问题，是因为若一段内存没有映射到自己的地址空间中，那么内核就不能操作这段内存。因此，内核可以处理的最大内存总量就是可以映射到内核的虚拟地址空间减去需要映射到内核代码本身上的空间。结果，一个基于 x86 的 Linux 系统最大可以使用略低于 1 GB 的物理内存。

为了迎合大量用户的需要，支持更多内存、提高性能，并建立一种独立于架构的内存描述方法，Linux 内存模型就必须进行改进。为了实现这些目标，新模型将内存划分成分配给每个 CPU 的空间。每个空间都称为一个 *节点*；每个节点都被划分成一些 *区域*。区域（表示内存中的范围）可以进一步划分为以下类型：

* ZONE\_DMA（0-16 MB）：包含 ISA/PCI 设备需要的低端物理内存区域中的内存范围。
* ZONE\_NORMAL（16-896 MB）：由内核直接映射到高端范围的物理内存的内存范围。所有的内核操作都只能使用这个内存区域来进行，因此这是对性能至关重要的区域。
* ZONE\_HIGHMEM（896 MB 以及更高的内存）：系统中内核不能映像到的其他可用内存。

节点的概念在内核中是使用 struct pglist\_data 结构来实现的。区域是使用 struct zone\_struct 结构来描述的。物理页框是使用 struct Page 结构来表示的，所有这些 Struct 都保存在全局结构数组 struct mem\_map 中，这个数组存储在 NORMAL\_ZONE 的开头。节点、区域和页框之间的基本关系如图 9 所示。

**图 9. 节点、区域和页框之间的关系**



当实现了对 Pentium II 的虚拟内存扩展的支持（在 32 位系统上使用 PAE —— Physical Address Extension —— 可以访问 64 GB 的内存）和对 4 GB 的物理内存（同样是在 32 位系统上）的支持时，高端内存区域就会出现在内核内存管理中了。这是在 x86 和 SPARC 平台上引用的一个概念。通常这 4 GB 的内存可以通过使用 kmap() 将 ZONE\_HIGHMEM 映射到 ZONE\_NORMAL 来进行访问。请注意在 32 位的架构上使用超过 16 GB 的内存是不明智的，即使启用了 PAE 也是如此。

（PAE 是 Intel 提供的内存地址扩展机制，它通过在宿主操作系统中使用 Address Windowing Extensions API 为应用程序提供支持，从而让处理器将可以用来寻址物理内存的位数从 32 位扩展为 36 位。）

这个物理内存区域的管理是通过一个 *区域分配器（zone allocator）* 实现的。它负责将内存划分为很多区域；它可以将每个区域作为一个分配单元使用。每个特定的分配请求都利用了一组区域，内核可以从这些位置按照从高到低的顺序来进行分配。

例如：

* 对于某个用户页面的请求可以首先从“普通”区域中来满足（ZONE\_NORMAL）；
* 如果失败，就从 ZONE\_HIGHMEM 开始尝试；
* 如果这也失败了，就从 ZONE\_DMA 开始尝试。

这种分配的区域列表依次包括 ZONE\_NORMAL、ZONE\_HIGHMEM 和 ZONE\_DMA 区域。另一方面，对于 DMA 页的请求可能只能从 DMA 区域中得到满足，因此这种请求的区域列表就只包含 DMA 区域。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-memmod/#ibm-pcon)

**结束语**

内存管理是一组非常庞大、复杂且耗时的任务，也是一个非常难以实现的任务，因为我们需要精雕细琢出一个模型，设计好系统如何在真实的多程序的环境中进行操作，这是一项非常艰难的工作。诸如调度、分页行为和多进程的交互组件都向我们提出了相当难度的挑战。我希望本文可以帮助您了解接受 Linux 内存管理挑战所需要的一些基本知识，并为您提供一个起点。