Linux 内核源代码解析

（基于 MIPS32 架构）

龙传慧　著

二 0 二一年四月

自　序

作者男，1984 年出生，江西赣州人，中共党员，2006 年本科毕业于南昌大学电子信息工程专业，获得工学学士学位，2011 年中国科学院光技术研究所硕士毕业，获得工学硕士学位。高考太紧张发挥的不好，本科上的大学不是很好，学的东西也比较杂，模电、数电、C 语言、C++、通信、信号处理等等，什么都学了点，什么也没学通，还是稀里糊涂。本科做毕业设计时，自学了一下 80C51 单片机，毕业设计做的是一个单片机的简单的控制电路，那时自己去电子市场花了一百多块钱买了个开发板，写了程序往里烧，程序跑起来的，控制了几个 LED 灯，那时觉得挺好玩，我也很好奇单片机里面到底是怎么工作的。大四那年，因为觉得上的本科不是很好，所以想努力考个好一点学校的研究生深造一下，结果选择了报考中国科学院光技术研究所（因为那时研究所不用交学费！！），结果居然还考上了。研究生阶段做的是一个模拟高压放大电路用于驱动压电陶瓷驱动器。虽然本科时学过模电、数电，但是那时真没学明白，一切感觉从零开始。研究生学习阶段学习环境、时间的自由，让我有时间慢慢地从新开始学习电路方面的知识，打牢基础，建立了模电、数电、处理器之间的联系，理解了其本质，最终在导师的指导下我顺利毕业了。到现在我都非常感激研究所给我提供的良好的学习环境，感谢导师们的指导，读研的三年感觉比我读四年大学学到的知识多多了。

研究生毕业回到了工作岗位，空闲时间总想学点什么，要不然我学的知识真的要荒废了。一直以来我都有一个梦想，梦想自己能在某一行当成为专家（小时候的理想是当科学家，现在估计是当不成了，成中年大叔了），曾经学过 C++、VB 等编程语言，还学过英语，甚至还学过几天俄语，现在感觉好搞笑！！研究生阶段做模电，买了一堆电子元器件，搭了些模拟电路玩，后来感觉玩不出什么名堂来，因为要做个印制电路板就得花一千多块钱，自己玩不起。想了想还是搞软件，买个开发板玩玩比较靠谱，于是又回到了大学时的情景，买了个 8051 单片机开发板玩。写了几天程序后，感觉单片机实在是太低级了，速度慢，内存小，连显示个小小的图片都要半天。后来，知道了 ARM 很火，又买了个 ARM 开发板，买回来发现开发资料里只有介绍怎么在 Linux 操作系统里写程序的教程，而我向来是个喜欢寻根问底的人。玩单片机时知道处理器是怎么执行每一条指令的，现在弄个操作系统把底层都屏蔽了，所以我把操作系统给刷了，写了一段时间的裸机程序。

在玩单片机的时候（大概是 2012 年的时候）我接触到了一本书，那是周立功写的一本书，书名我忘了，书的内容是讲在单片机中实现一个简单的操作系统。当时我感觉很惊讶，我知道的 Windows 操作系统装到电脑上要占好几个 G 的空间，小小的单片机如何实现操作系统？看完这本书之后，纠正了我错误的观念，Windows 是操作系统，但是操作系统不仅仅只有 Windows，凡是能实现进程调度，能让处理器运行多个程序的都是操作系统。后来我又了解到，Windows 只是商业上最成功的操作系统，国外讲操作系统的教材根本就没 Windows 什么事，C、C++语言也不一定非要在 Window 系统下编写运行。我发现我们的眼界在上大学的时候就被限制在了 Windows 操作系统上。讲操作系统必然要讲到 Unix 和 Linux，后者是前者的继承和发展，Linux 内核是一个开源的项目，任何人都可以直接从其官网上下载源代码，Linux 内核加上一些系统应用、用户应用软件就成了跟 Windows 一样的发行版操作系统了。Linux 内核尤其广泛地应用在嵌入式系统中，因为它是免费的、开源的、可裁剪的，我买的 ARM 开发板装的就是 Linux 操作系统。

ARM 开发板写了一段时间的裸机程序后，我觉得有必要学习一下操作系统，因为现在所有应用程序都是在操作系统之上运行的。我觉得连单片机都能跑操作系统，那么操作系统应该不难学，从此我就踏上了学习 Linux 内核的不归路。

要自学操作系统又是谈何容易，我先是看几本有关操作系统原理的书，后面还接触了一下用于教学用的 Minix 系统（美国佬用的），想先从一个简单的操作系统开始，后面再学复杂的 Linux。但是很不幸，我发现简单的我也学不来，还是直接回到 Linux。

Linux 系统在加载到内存运行前，系统要运行一段引导加载程序（如电脑中的 BIOS），用于将内核加载到内存。当时的想法是先学引导加载程序，后学 Linux 内核，掌握整个系统的运作。嵌入式系统中最常用的引导加载程序是 U-boot，我花了几乎将近一年的时间学习 U-boot，最终还是没有学完。最后我还是放弃了，以我的能力不可能求全，还是直接学 Linux 内核吧，能把内核学好就不错了。

Linux 内核的书买了一堆，最开始看的是《深入 Linux 内核架构》，我天真的以为直接学习内核源代码就能直接成为专家了，结果是书基本上看不懂。后来大概是到了 2016、2017 年的时候，反反复复地看书，感觉有点眉目了，开始阅读内核源代码。刚开始，内核数据结构之间错综复杂的关系，让我非常的迷茫，甚至买了一个笔记本来专门画出数据结构之间的关系。

随着学习的深入，我产生了写笔记的想法，毕竟好记性不如烂笔头，而且内核方方面面的内容庞杂，如果不做记录，过几天自己都忘了，又等于白学。于是，本书的雏形就在我脑子里酝酿，最终 2017 年 7 月的时候我决定开设微信公众号，把学习的心得和经验跟大家一起分享，一起学习、进步。在笔记的撰写过程中，不断地促进了我对内核的学习，逼着自己去弄清内核的组织结构和工作机制。经过几年的努力，现在终于感觉有种拨云见日的感觉，内核的架构在脑海里也清晰了起来。曾经也迷茫、困惑过，怀疑过写作笔记的意义，甚至想到过放弃，最终还是庆幸在众多关注微友的支持下坚持了下来，另外也要非常感谢家人对我的支持，坚持就是胜利。

近几年在学习 Linux 内核的过程中，我还在关注国产处理器、操作系统方面的信息，操作系统离不开处理器。目前，电脑系统基本是英特尔和微软的天下，手机则是 ARM 和安卓、IOS 的天下，没有一样是国产的，这给我们的信息安全带来了非常大的隐患，加上 2018 年的中兴事件，更是坚定了我们发展自主处理器和操作系统的决心。因此我又抛弃了 ARM 的学习，选择了国产龙芯处理器（MIPS 架构），支持国产才是正道，虽然个人的能力有限，但也要力所能及地支持一下。

回顾几年的内核学习经历，最开始是好奇，觉得很神奇，希望搞明白内核是如何工作的；接下来是困惑，真正开始接触内核了就不知道从何处下手，不知道内核是什么结构，如何工作，各组成部分、数据结构是什么关系；后面是欣喜，感觉掌握点眉目了，可以顺藤摸瓜了；最后是淡定和从容，理解了内核架构，整个内核在脑海里有一个整体的架构，可以去思考和研究各部分是怎么工作的，相互关系是什么，可以去研究某个函数做了些什么工作。学习内核最终的目的是将内核移植到目标系统中，应用内核，我正在为此努力。

我看了很多关于内核方面的书，国外讲内核源代码的书虽然写得很好，如《深入 Linux 内核架构》等，但讲解的内核版本太低，多是 2.6.x 版本，现在内核版本都更新到 5.0 了，而且线条比较粗，个人认为并不是很适合初学者。国内讲内核方面的书，多是讲驱动的移植，个人觉得讲内核都讲的不够深入，不够全面，只知然不知其所以然。虽然网上也有很多学习资料，但质量和系统性就不敢恭维了。所以我决定把笔记写成一本适合初学者、系统性讲解内核（源代码）的书。

内核代码量巨大，各部分内容庞杂，关系错综复杂，新手如何快速入门呢？内核说到底是编译链接成的一个单一的可执行目标文件，系统启动时被加载到内存中开始运行，内核启动完成后负责为用户进程提供服务。内核更多的是充当一个资源管理器的功能，通过某些数据结构管理和调配系统各种软、硬件资源。学习内核要先了解内核的组织架构，各功能部件的主要功能和相互关系，避免一开始就迷茫在代码细节中。本书在讲解内核的过程中，尽量地在章节的开始就建立本部分的组织架构，让读者对本部分的功能和组织结构有整体的认识，再顺着组织架构去阅读和理解内核源代码。

本书主要从内核组织架构及源代码讲解内核启动、内存管理、进程管理、文件系统、驱动程序、网络等各功能组件的工作机理，以 MIPS32 架构为基础介绍内核中特定于处理器架构的代码实现，并以龙芯 1B 处理器为基础介绍 Linux 内核移植的相关知识。

作者在 Windows 系统中采用 CodeLite 集成开发环境阅读源代码。CodeLite 是一款开源，支持 Windows、Linux 等操作系统的集成开发环境。移植内核前，读者需要在您的电脑上安装上 Linux 发行版操作系统，

以便对内核源代码进行配置和构建。作者主机安装的是国产 Deepin 操作系统，具体安装方法读者可参考 Deepin 操作系统官网。

学习本书之前读者应先了解操作系统基本知识、Linux 操作系统（发行版）的安装和常用命令、MIPS 架构（或其它处理器架构）、以及 C 语言和汇编语言（特定于处理器）等相关知识。由于作者水平有限，难免会有错误之处，敬请读者批评指正。

本书结构如下：

第 1 章 内核概述：简要介绍操作系统组成，Linux 内核框架和各组成部分的功用，移植内核的流程，以及内核源代码中常用的通用数据结构。

第 2 章 启动内核：介绍内核启动阶段启动函数所做的工作。

第 3 章 物理内存管理：介绍内核对物理内存的分配、释放等管理工作。

第 4 章 虚拟内存管理：介绍内核对进程虚拟地址空间的管理，以及内核自身地址空间的管理。

第 5 章 进程管理：介绍进程创建、调度、进程间通信等相关内容。

第 6 章 内核活动：介绍异常（中断）的处理、系统调用实现以及内核时间管理等。

第 7 章 文件系统：介绍虚拟文件系统的实现，几种具体文件系统类型的实现。

第 8 章 通用驱动模型：介绍内核对设备和驱动程序进行管理的驱动模型，以及常用总线驱动实现。

第 9 章 字符设备驱动程序：介绍常见字符设备驱动程序框架及实现。

第 10 章 块设备驱动程序：介绍块设备驱动程序框架及实现。

第 11 章 内存与块设备交互：介绍内核页缓存和块缓存、数据同步以及页回收和页交换的相关内容。

第 12 章 网络-套接字、传输层：介绍内核访问网络的套接字接口，简要介绍因特网 TCP/IP 分层协议，介绍传输层协议在内核的实现。

第 13 章 网络-网络层、数据链路层：主要介绍 TCP/IP 协议栈中网络层协议、数据链路层协议的实现，简要介绍 IPv6 协议的实现。

第 14 章 内核移植：以龙芯 1B 处理器为例，介绍移植公版 linux-4.2.4 内核到龙芯 1B 开发板的步骤和方法。

扫码关注微信公众号：

目　录

第 1 章 内核概述

翻开本书的读者，相信您已经对计算机体系结构，计算机编程等已经有了一定的了解。计算机就是能够自动执行事先编制好指令的机器。程序就是由指令组成的集合，程序加载到内存后处理器依次执行指令，完成人类赋予的工作。每种计算机体系统架构（如 MIPS）规定一个指令集，表示计算机能够执行的指令。

最初，计算机只能依次一个一个地执行程序，执行完一个后再执行下一个，这时的计算机没有操作系统，由人工输入一个程序，执行完后再由人工输入下一个要执行的程序。后来，计算机有了批处理系统，即人工输入多个程序，多个程序排成队列，计算机执行完一个之后，自动调入下一个程序执行。再后来，计算机有了操作系统，计算机可以同时执行多个程序（执行中的程序称为进程），处理器在多个进程间跳转，一个进程执行一段时间后，跳转到下一个进程执行，如此循环，以达到多个程序同时执行的效果。

操作系统不仅管理着运行中的程序，还管理着系统的其它资源，如处理器、外部设备、文件系统等。操作系统内核执行进程、系统资源管理和操作等核心功能，用户进程需要操作系统资源时，向内核提出申请，由内核代为执行，最后将结果返回给用户进程。内核本身可以看成一个程序，它提供多种功能供用户进程调用。内核加上一些必要的应用程序，如桌面管理程序、常用系统工具、文件系统等，就组成了一个完整的操作系统。

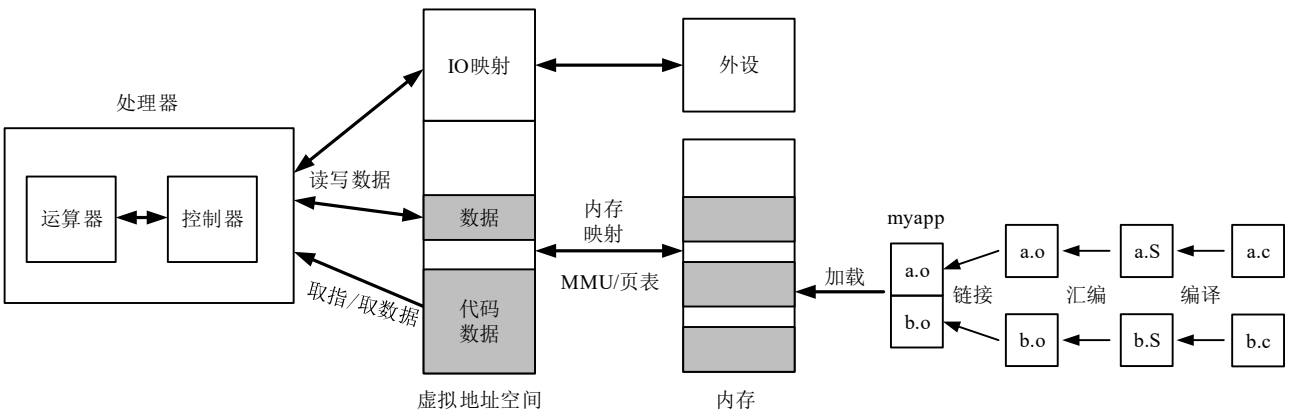
1.1 操作系统

操作系统最核心的功能就是管理多个同时执行的程序（进程），以及对系统软硬件资源的管理和操作。下面先介绍处理器如何执行一个程序，再介绍如何同时执行多个程序，最后介绍操作系统和内核的功用。

1.1.1 运行单个程序

计算机由五大部分组成：运算器、控制器、存储器、输入设备和输出设备。运算器和控制器位于处理器里，运算器主要完成算术和逻辑运算，里面包含寄存器和运算单元等，寄存器用于暂存运算数据。控制器用于控制指令、数据的读取，数据的写出，指令的解码，流水线控制等。存储器用于存放指令和数据。输入设备和输出设备称为外部设备，通常映射到处理器的虚拟地址空间，处理器可以像访问内存一样访问外部设备。

程序的执行如下图所示，程序的代码和数据保存在物理内存中，程序代码中的虚拟地址通过 MMU（内存管理单元）映射到处理器物理内存地址。代码和数据在物理内存中可以是分散存储的，MMU 可以将其映射到连续的虚拟内存区域。由于程序代码中使用的是虚拟地址，所以程序觉察不到物理内存是分散的。



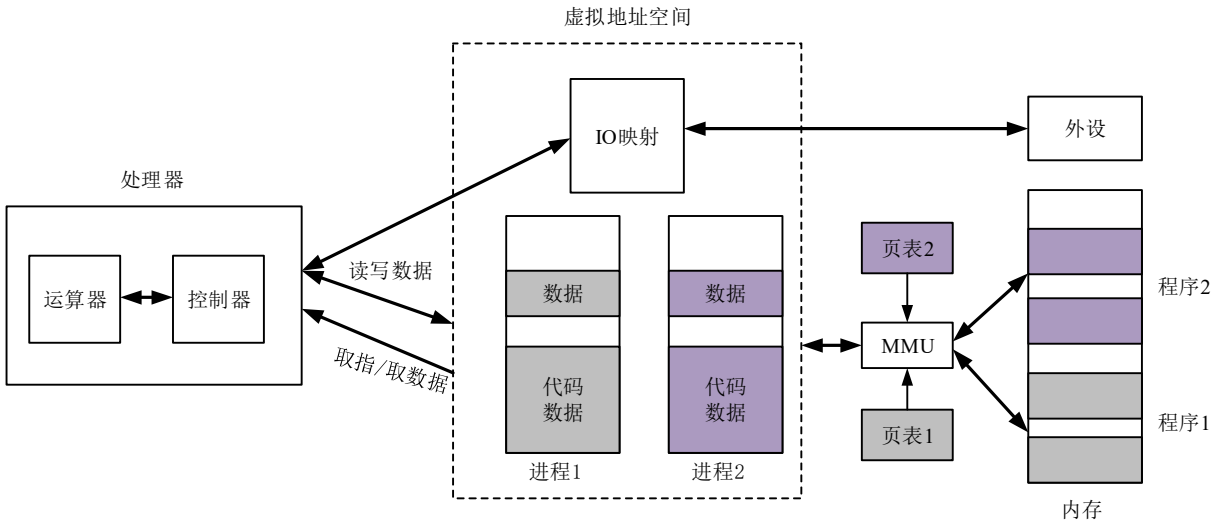
处理器有一个程序指针 PC，在硬件时钟的驱动下，顺序取出保存在代码区的指令，在控制器的作用下送入运算器执运算操作，指令执行完后通过控制器将结果写出内存或外设映射地址（寄存器）。

处理器只能识别二进制的机器指令，每条指令对应一个二进制编码，每条指令表示要执行的一个动作，如从内存取数据到运算器中的指定寄存器，将运算器指定寄存器中数据左移一位等。每条指令需要硬件逻辑的支持，每种体系结构的处理器只能识别、执行固定的一批指令，这些指令称为指令集（ISA）。指令集加上处理器中的控制逻辑的规范，中断处理的规范，处理器本身控制的一些规范等就组成了处理器体系结构。体系结构可视为实现具体处理器的规范或标准，流行的处理器体系结构有 x84_64、ARM 等，本书关注的是国产龙芯处理器采用的 MIPS 体系结架，详细内容读者可参考相关手册。

通常我们用 C、C++等高级语言编写应用程序，所有高级编程语言编写的程序代码都需要通过编译器编译成处理器的汇编源程序，然后通过汇编器将汇编源程序转换成能被处理器识别的二进制指令程序，最后通过链接器将各二进制指令程序合并成一个二进制可执行文件，以便加载到物理内存供处理器执行。在链接过程中通常将所有的代码段链接在一起，所有的数据段链接在一起（上图中只是简单地示意了将两个指令程序文件拼接），用户也可以在源代码中指定将代码或数据链接到某个指定的段。

1.1.2 运行多个程序

如果处理器只运行一个程序，那就不需要操作系统了。因为系统的所有资源都属于这个进程，不需要进行管理。当有多个程序同时运行时，首先需要解决的是处理器在各程序之间的跳转问题。下图示意了系统中同时运行两个程序时的情形：



上图中程序 1 和程序 2 的代码和数据都加载到了内存中，分别位于内存中的不同位置。程序中使用的是虚拟地址，程序 1 和程序 2 的虚拟地址是相同的。每个进程具有自己的页表，页表记录的是虚拟地址映射的物理地址，MMU 将虚拟地址转换成物理地址。不同的页表转换的物理地址不同，从而实现进程相同的虚拟地址映射到不同的物理地址。某一时刻，某个处理器核只能执行一个程序的代码（硬件超线程除外），如上图中所示，处理器在执行进程 1 时使用页表 1 访问物理内存，在执行进程 2 时使用页表 2 访问物理内存，以达到进程间地址的隔离。系统内需要保存页表 1 和页表 2 的信息，处理器在进程间切换时要保存当前进程的上下文信息，主要是此时处理器运算器中寄存器的信息，页表信息等，以便下次再次执行本进程时能够正确恢复前一次的状态。切换到下一个进程运行前，恢复下一个进程的上下文信息至处理器，并切换使用下一个进程的页表，而后处理器就可以执行下一个进程了。

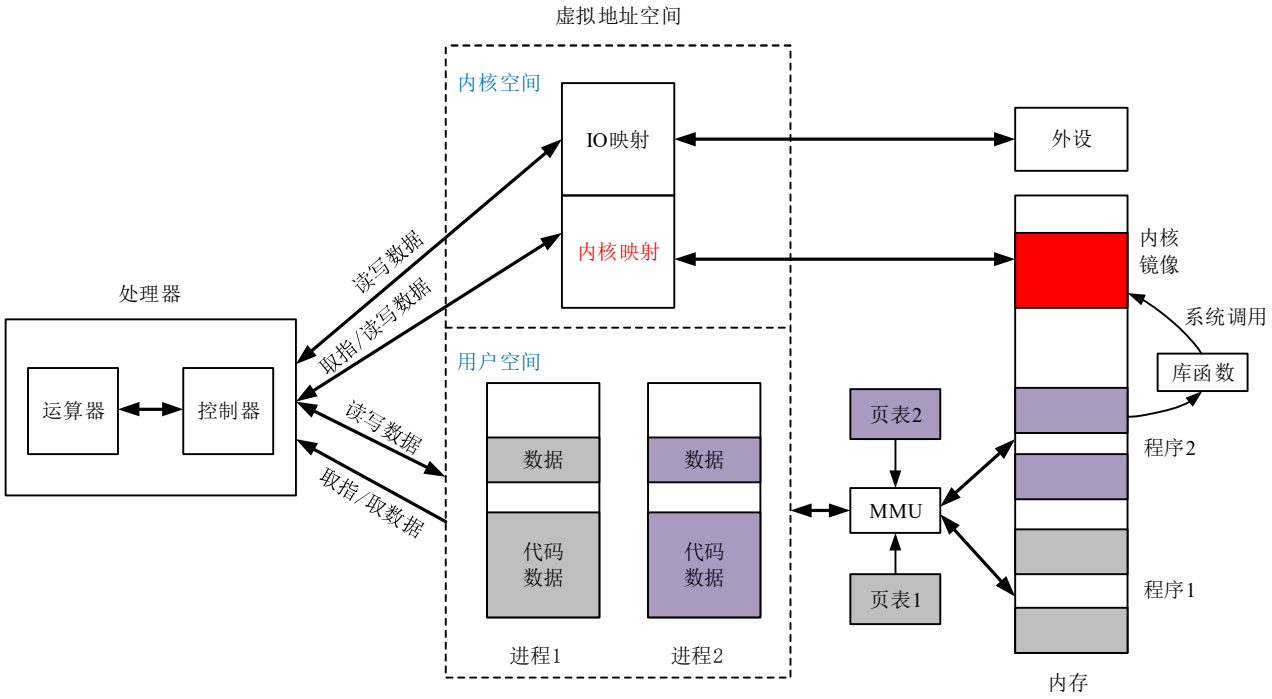
实现运行多个进程的一个简单方法是将多个程序加载到内存，并分别建立页表，在系统中设置一个定时器，在定时器中断处理程序中保存当前进程的上下文信息，恢复下一个进程上下文信息至处理器，切换下一个进程页表，中断返回。中断返回后执行的就是下一个进程了，也就是说由定时器中断来控制进程间的切换。

1.1.3 操作系统实现

系统中如果运行着多个进程，如果没有有效的管理势必会带来混乱，而且如何来实现进程的动态创建和退出呢？为此，系统中引入了一个进程，称为内核，用于对系统内运行的进程进行有效管理。内核负责进程的创建、退出、进程间的切换、调度等，内核还有一项重要的任务是对系统公共资源的管理和操作，如物理内存、文件系统管理和操作，外部设备的管理和操作等。

从内核自身的角度来说，内核可视为一个系统资源的管理器，它管理着系统内的软硬件资源。进程需要某项资源或对某项资源进行操作时，向内核发出申请，由内核代为执行，并将执行结果返回给进程。由内核统一执行对系统资源的操作可提高操作的效率、避免冲突，并简化用户程序的编程。

从进程（用户）的角度来说，内核相当于一个函数库，提供各项操作的接口函数，进程需要执行某项操作时，调用内核提供的接口函数即可。内核提供的接口函数称为系统调用，用户进程通过中断（异常）的方式进入内核代码，执行系统调用。内核与用户进程的关系如下图所示：



现代处理器地址空间通常分为内核空间和用户空间，处理器只有处于核心态时才能访问内核空间，处理器处于用户态时只能访问用户空间。处理器处于核心态时才具有访问系统特权资源的权限（如外设，文件系统等），内核在处理器的核心态下运行，内核代码映射到处理器内核空间，以保证只有内核才能访问系统特权资源，确保安全。

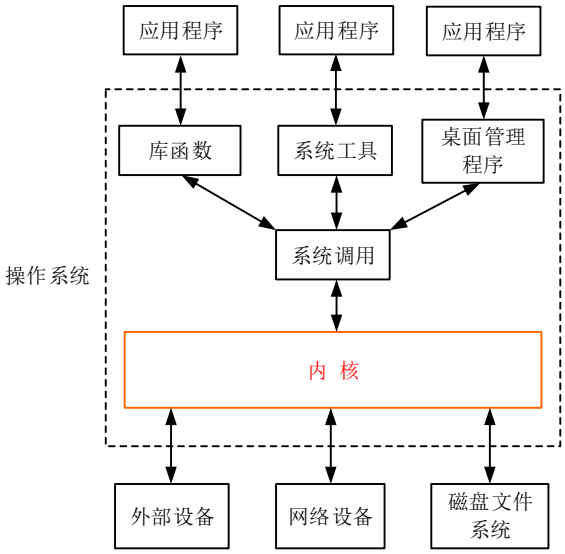
在系统内，内核只有一个，因此内核地址空间至物理内存（内核镜像、内核代码）的映射是固定的，而用户进程有多个，不同的进程通过页表将用户进程地址空间映射到不同的物理内存。

用户进程通过系统调用进入内核运行，调用内核功能。系统调用是内核与用户的接口。系统调用与普通的函数调用类似，每个系统调用具有一个编号和相关的参数。系统调用通常封装在库函数中，库函数在准备好系统调用编号值和相关参数后，执行一条系统调用异常（或中断）指令，PC 指针跳转到异常处理程序处取指执行，从而使用户进程进入内核运行。

内核将所有系统调用的实现函数指针放在一个列表中，系统调用异常处理程序根据编号查找列表，调用相应的实现函数，执行完后从异常（中断）返回用户空间（含返回值）。

除了向用户进程提供系统调用，对进程本身的管理也是内核一项非常重要的工作，例如：创建进程、进程执行新的代码、进程切换、进程调度（何时切换以及切换至哪一个进程）、进程资源的管理、进程间通信、进程退出等等。

内核只封装了最基本和最基础的资源管理和操作功能。发行版的操作系统在内核的基础上实现了一些常用和通用的系统工具，如创建目录、创建文件等，以方便用户空间编程。具有图形操作界面的操作系统还提供了一个桌面管理程序，以方便用户的操作。操作系统组成如下图所示：



本书主要介绍 Linux 内核（操作系统内核）的组成架构和功能，通过解读内核源代码使读者理解内核实现的机制，讲解设备驱动程的实现框架和内核移植的相关知识。

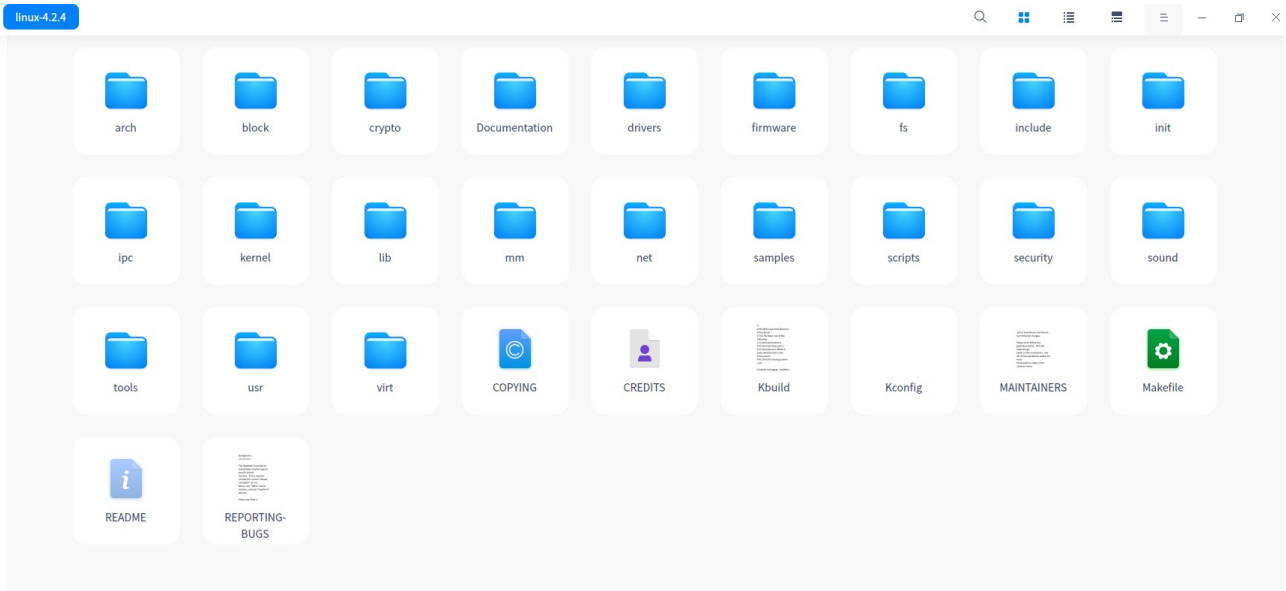
1.2 Linux 内核

操作系统内核可视为一个程序，它由一组源代码文件通过编译、链接生成一个单一的可执行目标文件（二进制指令文件），在系统启动时加载到内存中运行。

Linux 内核是一个广泛使用的操作系统内核。本节先介绍 Linux 内核源代码文件的组成，然后简要介绍 Linux 内核主要组成部分之间的关系和功能。本书后面如果没有特别说明，内核就是指 Linux 内核。

1.2.1 源代码目录

Linux 内核源码压缩包读者可从 www.kernel.org 网站直接下载，解压后得到内核源代码树，如下所示：



源代码根目录下包含的主要子目录及其内容简介如下：

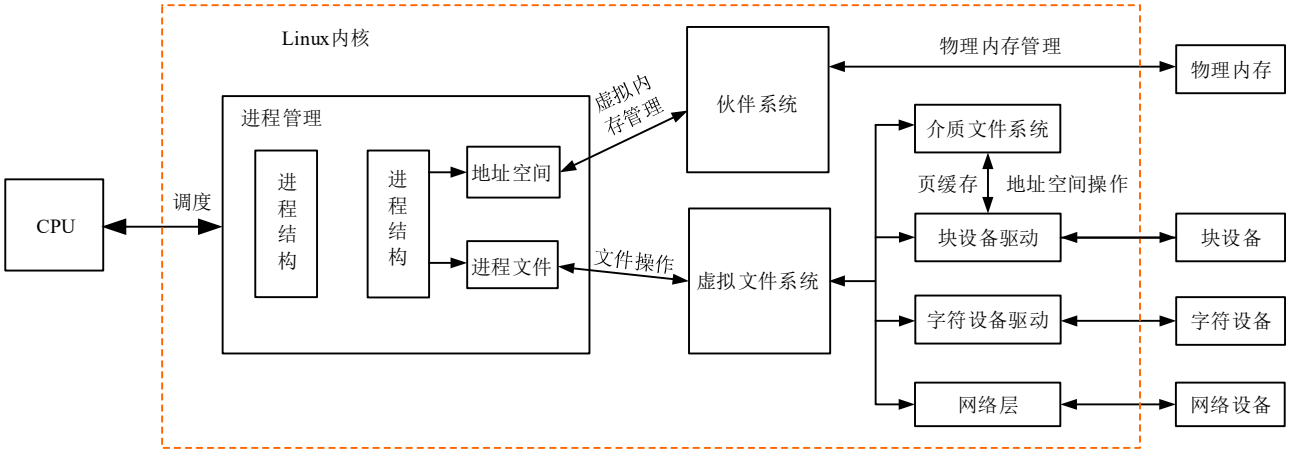
/arch	体系结构相关代码，每种体系结构对应其下一个子目录，如/arch/mips/。
/block	块设备驱动通用层代码。
/crypto	加密层文件，包含了各种密码算法的实现。
/drivers	设备驱动程序代码，包含通用驱动层代码和特定字符、块设备驱动程序代码。
/firmware	固件代码。
/fs	虚拟文件系统及具体文件系统类型代码。
/include	头文件。
/init	内核初始化代码。
/ipc	进程间通信机制实现代码。
/kernel	内核核心功能代码，如进程调度、中断等。
/lib	内核通用库例程代码，如基本数据结构操作的实现等。
/mm	内存管理代码，包括物理内存管理和虚拟内存管理。
/net	网络协议层代码。
/samples	内核功能举例。
/scripts	编译、链接内核或进行其它任务的脚本和实用程序。
/security	安全框架和密钥管理代码。
/sound	声卡驱动程序代码。
/tools	内核配置编译过程中使用的工具程序。
/usr	用户层代码，主要用于编译根文件系统。
/Documentation	内核说明文档。
Kconfig	顶层配置文件
Makefile	顶层构建文件

1.2.2 内核架构

系统在启动时，将由引导加载程序（固件中的程序）将内核目标文件（镜像）从外部存储介质中加载到物理内存中并运行。内核首先将执行初始化操作，初始化操作完成后将创建并运行用户进程。

内核是系统中的大总管，管理、调配系统中的所有软、硬件资源，管理和调度进程的运行。内核在启动初始化阶段，将会创建管理各种资源的数据结构，如红黑树、链表、根文件系统等，并执行各种资源的注册和初始化函数，以实现对资源的管理。内核通过系统调用为用户进程提供资源使用、操作的接口，并将操作结果返回给用户进程。

Linux 内核组织架构如下图所示：



Linux 内核按功能可分为进程管理、内存管理、文件系统、设备驱动程序、网络层等组成部分。下面分别简要介绍各部分的功能。

1 进程管理

内核最终是为进程服务的，内核首先是对进程本身进行管理，其次是为进程提供各种资源及操作接口。进程起初是不存在的，内核在启动后期将创建第一个用户进程（启动阶段会创建内核线程），并由此进程创建其它的用户进程。

每个进程在内核中由 task_struct 结构体实例表示，此结构体管理着进程所有的资源和信息。内核创建进程时，要为其创建虚拟地址空间管理结构，从物理内存中为其分配物理内存（建立页表），从外部文件系统中导入程序代码和数据至内存，并将 CPU 控制权交给此进程，此进程才能得以运行。

除内存外，进程另一个重要的资源是文件，Linux 奉行“万物皆文件”的哲学。进程通过文件接口访问外部介质文件系统中的文件、外部设备以及网络等资源。内核要为进程提供打开、关闭、操作文件的接口，普通文件系统、设备驱动程序、网络层都要实现文件操作的接口。

对进程本身的管理是影响系统性能的重要因素。内核提供了进程创建、退出、进程睡眠、进程间通信的机制。进程调度是进程管理的一项重要工作，进程调度是指何时进行进程切换，如何进行进程切换，以及切换的下一个进程如何选择等。系统内运行着多个进程，不同的进程重要性、紧迫性不同，CPU 运行时间如何在各种类型进程间分配是一个重要的问题。

此外，内核还需要解决不同进程访问同一共享资源的竞争问题，以及进程其它资源的管理。

2 内存管理

物理内存是程序运行的基础，用于存放程序代码和数据，CPU 从中读取指令和数据。系统中的进程在不断地创建和退出，因此需要不断地申请和释放内存。内核在运行过程中也需要不断地申请和释放内存。

内核为了管理物理内存，将物理内存按页进行划分，页大小通常为 4KB（也可以是其它大小），此物理内存页称为页帧。内核在初始化阶段为每个页帧分配了 page 结构体实例（数组），用于管理和跟踪页帧。在 page 实例的基础上内核通过伙伴系统分配和释放物理上连续的页帧，一次分配/释放页帧的数量为 2^{order}，order 称为分配阶，例如：order 为 0 表示分配 1 个页帧，order 为 3 表示分配 8 个连续的页帧。

内核总是按页（单页）为用户进程分配物理内存，内核自身可按不同的分配阶申请物理内存，只要不超过内核设置的最大分配阶即可。另外，内核运行过程中需要为各资源管理结构分配数据结构实例，此实例通常较小，用不了一页的物理内存。为此内核实现了分配小块内存的 slab/slub/slob 分配器，配置内核时在三者中选择其一，其中 slub 适用于大型系统，slob 适用于资源有限的嵌入式系统，slab 比较适中。这三个分配器都是从伙伴系统中申请内存页，再划分成小块分配给内存各子系统使用。

以上介绍的伙伴系统和 slab/slub/slob 分配器是内核对物理内存的管理。

用户进程在处理器用户态下运行，访问用户地址空间，用户地址空间总是通过页表转换映射到物理地址空间。实际进程通常并不需要使用到全部的用户地址空间，内核将进程使用到的用户地址空间区域划分为多个虚拟内存域，并对进程的虚拟内存域进行管理。用户进程的虚拟内存域通常并不是在进程运行时就全部映射到物理内存，而是在进程访问到此虚拟内存域时，再按需创建到物理内存的映射（按页创建）。用户地址空间的映射通常在缺页异常处理程序中完成。另外，还可以将文件等映射到进程虚拟内存域，以简化进程对文件等的操作。

内核在处理器内核态下运行，访问内核地址空间，内核地址空间分为直接映射区和间接映射区。直接映射区线性映射到连续的低端物理内存（物理内存底部），不通过页表转换（MIPS 体系结构是这样，其它的不一定，如 ARM 还是需要通过页表映射）。间接映射区通过页表可映射到物理内存的任意位置，这也是按页建立映射的。内核通过伙伴系统为直接映射区和间接映射区分配映射的页帧，不过直接映射区只能在低端物理内存中分配页帧，间接映射区可以在任意位置分配页帧，间接映射区分配页帧后需要修改内核页表，而直接映射区不需要。另外，slab/slub/slob 分配器也只能在低端内存中申请页帧，分配出的数据结构实例地址将线性映射到内核直接映射区。

用户进程虚拟内存域的管理、映射的创建和解除，以及内核映射区映射的创建和解除，统称为虚拟内存管理。

3 文件系统

Linux 奉行“万物皆文件”的哲学，系统内的许多资源都通过文件来表示，例如：普通文件、外部设备、网络等。内核通过虚拟文件系统（VFS）管理系统的文件资源，并提供文件操作的接口。虚拟文件系统中包含一个由目录项（由 dentry 结构表示）构成的树状层次结构，称为根文件系统，目录项可表示普通的目录，也可表示文件名称。内核在初始化阶段会创建树状目录的根目录，名称为“/”。每个目录项关联到表示对应文件的节点（inode）结构体实例（VFS 中文件由节点表示），普通的目录也关联节点，因为普通目录也是文件，只不过文件的内容是目录项而已。

根文件系统相当于系统中文件的仓库，目录项关联的文件节点结构中包含文件操作的接口，即 file_operations 结构体实例，内核通过此实例中的操作函数对文件进行操作。

实际文件保存在文件系统中，文件系统是按某种文件系统类型的格式组织的一个文件的集合。例如，磁盘分区中的 ext2 文件系统。内核还实现了基于内存盘和数据结构实例的文件系统。文件系统需要挂载到根文件系统某个目录项下，以导入内核，使其对内核可见。

不同类型文件系统中的文件保存形式各不相同，其读写操作方法也各不相同。内核抽象出了目录项、文件等的操作接口，具体文件系统类型需要实现这些接口，以执行特定于文件系统的操作。内核通过这些统一的接口操作文件系统中的目录项和文件。

内核支持几十种的文件系统类型，例如：ext2、ext3、ext4、FAT 等。另外，内核定义了许多基于内存、内核数据结构实例的文件系统类型，用于向用户进程提供设置内核参数、获取内核信息的渠道。内核越来越倾向于通过文件系统与用户进程进行通信。

进程在操作文件前需要执行打开操作。打开文件操作就是到内核根文件系统（文件仓库）中去查找所需的文件，查找文件时按路径（目录项）在根文件系统中搜索，如果文件尚未被打开过，则需要从实际文件系统（介质文件系统）中导入文件信息至内核根文件系统。进程通过 file 结构体实例建立与根文件系统中文件 inode 节点之间的关联，而进程 task_struct 结构体中包含打开文件 file 实例指针数组，数组索引值表示文件描述符（一个正整数）。因此对于用户进程来说，文件按名称打开后，就通过文件描述符对其进行访问了。

4 设备驱动程序

外部设备在内核中也由文件表示，称为设备文件。设备文件作为一种特殊文件，保存在某个实际的文件系统中。

在设备文件中保存了设备的类型，是字符设备还是块设备，以及设备号等信息。内核在打开设备文件时，从设备文件中获取设备类型和设备号，依此到设备数据库中查找设备驱动程序，获取设备文件操作接口实例。设备驱动程序需要实现虚拟文件系统中定义的文件操作接口 file_operations 实例，以此实现对设备的操作。对于内核来说，所有文件都是一样的，都采用相同的操作接口，它不会去区分是普通文件还是设备文件。

内核将外部设备分为字符设备和块设备两种类型，对每个设备赋予一个设备号，设备号由主设备号和从设备号组成。在创建设备文件时，设备号将写入设备文件节点中。内核分别为字符设备和块设备建立了设备数据库，字符设备由 cdev 结构体实例表示，块设备由 gendisk 结构体实例表示（两个结构体中包含设备号）。cdev 和 gendisk 结构体实例由设备驱动程序创建和注册，cdev 结构体中包含字符设备文件操作结构 file_operations 实例，块设备 gendisk 结构体实例中需要实现读写块设备的队列和执行 IO 操作的方法，内核定义了公共的块设备文件操作结构 file_operations 实例，其通过块设备驱动程序定义的块设备队列执行对块设备的操作。

字符设备驱动程序需要向内核申请设备号，创建并初始化 cdev 实例，定义 file_operations 实例并赋予 cdev 实例，最后向内核注册 cdev 实例（注册实例时通常创建设备文件）。

块设备是按数据块进行读写的设备，块设备驱动程序中需要申请设备号，创建并初始化 gendisk 结构体实例，实现块设备操作的请求队列，最后注册 gendisk 实例。在注册 gendisk 实例的过程中内核将会扫描块设备中的分区，并为各分区创建设备文件。

内核可以直接通过块设备文件对块设备进行操作，也可以通过文件系统对块设备进行操作。通过文件系统进行操作时，具体文件系统类型的代码会将对文件内容的操作转化成对块设备中数据块的操作。最终，所有对块设备的操作都会转化成对块设备中数据块的操作，并封装成请求，提交到块设备请求队列，由块设备驱动程序实现数据块数据的传输。由于块设备读写操作比较缓慢，内核在内存中建立了块设备数据（文件内容）的缓存，以提高访问效率，缓存由内存页组成，由地址空间结构进行管理。

5 网络

网络设备是比较特殊的设备，内核不是通过设备文件访问网络设备，而是通过套接字访问网络。在创建套接字时也会关联到 file 实例，并赋予文件操作结构 file_operations 实例，返回文件描述符。进程除了可以通过 file_operations 实例访问网络外，内核还定义了一组网络专用的系统调用，以实现不能合并到 file_operations 结构体中的操作。

内核网络层主要实现进程与进程间的通信，包括本机进程间的通信和不同主机进程间的通信。计算机网络用于将不同的主机连接在一起，以实现数据传输。不同的网络类型具有不同的数据传输协议，内核网络层主要是实现网络数据传输协议。

套接字对上实现进程操作网络的接口，对下连接网络传输协议，实现数据格式化和传输。网络设备负责将网络协议下传的数据发送出去，以及将接收到的数据提交给网络协议。网络协议层屏蔽了数据传输的细节，对于用户进程来说可以像操作普通文件一样操作网络数据。

最著名的计算机网络就是因特网，因特网将世界上数以亿计的主机连接在一起。因特网网络协议采用了分层的结构，主要分为应用层、传输层、网络层和数据链路层，内核主要实现后面三层，应用层由用户空间实现。传输层协议主要有 UDP 和 TCP，用于实现主机进程间的通信，主机内的进程通过端口号进行识别。网络层协议为 IPv4/IPv6，用于实现主机之间的通信，各主机通过 IP 地址寻址，网络中的路由器用于将数据发送到指定 IP 地址的主机上。数据链路层用于实现网络相邻节点间的数据传输，常用的协议有以

太网、WiFi 等，数据链路层各节点用设备 MAC 地址寻址。

发送网络数据时，通常需要指定目的主机的 IP 地址和端口号，即将数据发送到哪个主机上的哪个进程。因特网通过 IP 地址寻址到目的主机，在主机内由端口号寻找到接收进程，并将数据传递给该进程。

1.2.3 内核移植

学习内核最终是为了应用内核。内核其实就是一个程序，经过配置、编译、链接生成一个单一的可执行目标文件，可加载到目标机上运行。

Linux 内核支持几十种类型的计算机体系结构，将内核编译生成能够在目标机上运行的可执行目标文件，并加载到目标机上运行，称为内核移植。

移植内核，需要在用户主机上安装 Linux 发行版操作系统，下载内核源码压缩包，并解压获得内核源代码树。移植内核前需要做的主要工作是选择（或编写）适用于目标机的设备驱动程序，以及编写板级设备相关的设备描述源代码（文件）。

准备好内核源码后，移植内核的第一步是对内核进行配置。在配置时需要设置目标机体系结构，设置（交叉）编译工具前缀等信息，选择内核配置选项（选择内核特性、行为、参数等），以确定对哪些源文件进行编译链接、选择函数实现方式、源文件是持久编译入内核还是编译成模块、选择设备驱动程序，以及选择内核特性等等。

内核提供了 Kconfig 机制用于配置内核，配置选项由 Kconfig 语言编写，各内核源代码目录下几乎都存在一个 Kconfig 文件，用于设置内核的配置选项。在源代码根目录下执行 make menuconfig 等命令将启动内核配置，内核将向用户提供一个配置界面，用于选择配置选项，配置完成后将保存配置结果。

移植内核的第二步是构建内核，执行 make 命令，即根据配置选项，编译、链接内核源文件，生成可执行目标文件以及模块。内核通过 Kbuild 机制实现内核的构建，几乎每个内核源代码目录下都有一个 Makefile 文件，文件内根据配置结果，确定要编译、链接本目录和子目录下的哪些文件，确定是持久编译入内核还是编译成模块，或是不编译。最终，所有目录下编译生成的目标文件将链接成一个单一的可执行目标文件，用于加载到目标机内存中运行。

移植内核的第三步是制作目标机根文件系统，将内核可执行目标文件复制到根文件系统，在引导加载程序中设置正确的命令行参数，使引导加载程序从根文件系统中加载内核运行。

Linux 操作系统定义了一个标准的根文件系统目录（及文件）。根文件系统中除了需要包括内核可执行目标文件，还需要包含系统常用工具，第一个用户程序目标文件，系统配置文件等。在嵌入式系统中，通常通过 busybox 生成系统常用工具及用户程序（含 shell 程序）。

内核移植的详细操作读者可参考第 14 章。

1.3 基本数据结构

在讲解内核源代码之前，有必要先了解内核中常用的基本数据结构类型，它们在内核代码中广泛使用。基本数据结构主要用于管理、跟踪内核数据结构实例，如表示进程的 task_srtuct 结构体实例、表示内核文件的 inode 结构体实例等等。

内核并没有使用很复杂的数据结构，就是简单的单链表、双链表、散列表、红黑树、基数树等。本小节对以上数据结构做简要的介绍，为后面阅读内核源代码做准备。

1.3.1 容器

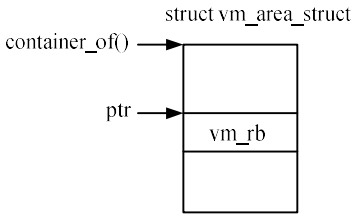
容器是指通过数据结构内嵌的某个成员的指针，获取数据结构实例的指针。

例如：vm_area_struct 结构体中包含红黑树节点 rb_node 结构体成员，成员名称为 vm_rb：

struct vm_area_struct {

```
    ...
    struct rb_node    vm_rb;
    ...
}
```

假设ptr为指向vm_area_struct 结构实例中vm_rb成员的指针，**container_of(ptr,vm_area_struct,vm_rb)**宏返回 ptr 指向 vm_rb 实例所在 vm_area_struct 结构体实例指针，如下图所示：



container_of(ptr, type, member)宏定义在/include/linux/kernel.h 头文件：

```
#define container_of(ptr, type, member) ({                  \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);   \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

type为数据结构名称，如上例中的vm_area_struct。member为type数据结构中成员名，如上例中的vm_rb。ptr 为指向 member 成员的指针，如 ptr 为指向 vm_area_struct.vm_rb 成员。

container_of()宏返回 type 数据结构实例指针。

1.3.2 链表

链表是内核广泛使用的简单数据结构，包括单链表、双链表、内核链表、散列表等。

1 单链表

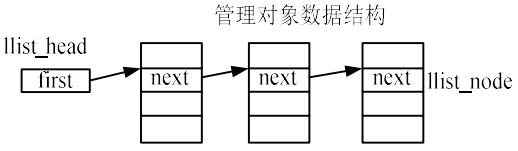
单链表用于将数据结构实例链接成单向的链表，单链表头数据结构定义在/include/linux/llist.h 头文件：

```
struct llist_head {
    struct llist_node *first;       /*第一个节点指针*/
};
```

单链表中其它节点成员由 llist_node 结构体表示，定义如下：

```
struct llist_node {
    struct llist_node *next;     /*指向下一节点指针*/
};
```

节点成员中只包含指向下一节点的指针成员，节点结构通常嵌入到被管理数据结构中。单链表结构如下图所示：



内核在/include/linux/llist.h 头文件定义了单链表的通用操作函数：

#define LLIST_HEAD_INIT(name) { NULL } /*初始化单链表头*/

#define LLIST_HEAD(name) struct llist_head name = LLIST_HEAD_INIT(name)

/*定义并初始化单链表头*/

●bool **llist_empty**(const struct llist_head *head)：测试单链表是否为空。

●struct llist_node ***llist_next**(struct llist_node *node)：返回 node 节点的下一个节点。

●bool **llist_add**(struct llist_node *new, struct llist_head *head)：将节点插入单链表头部，

返回插入节点前单链表是否为空。

●struct llist_node ***llist_del_first**(struct llist_head *head)：删除第一个节点。

●**llist_entry**(**ptr**, type, member)：ptr 为指向 llist_node 实例的指针，member 为 type 数据结构中 llist_node 结构体成员的名称，返回 ptr 指向 llist_node 实例所在 type 数据结构实例指针。

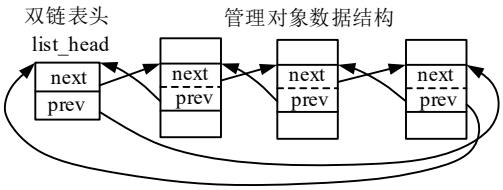
2 双链表

双链表是内核中大量使用的简单数据结构，以环形双向链表的形式管理对象。双链表表头及节点数据结构相同，定义在/include/linux/types.h 头文件：

```
struct list_head {
    struct list_head *next, *prev;    /*next：指向下一节点，prev：指向前一节点*/
};
```

list_head 结构体中包含两个成员，分别是指向前一个节点和后一个节点的指针。

list_head 结构体通常嵌入到被管理对象数据结构中，内核通常会单独定义表示链表头的 list_head 结构体实例。双链表结构如下图所示，双链表的首尾成员也会相互连接组成环形链表：



内核在/include/linux/list.h 头文件中定义了双链表的基本操作函数，例如：

#define LIST_HEAD_INIT(name) { &(name), &(name) }

/*双链表节点（表头）初始化，两个成员都指向同一实例*/

#define LIST_HEAD(name) \

struct list_head name = LIST_HEAD_INIT(name) /*定义并初始化双链表节点（表头）*/

#define list_entry(ptr, type, member) container_of(ptr, type, member)

/*由 list_head 指针获取被管理对象实例指针*/

```
#define list_for_each_entry(pos, head, member)                \
  for (pos = list_first_entry(head, typeof(*pos), member);  \
       &pos->member != (head);                              \
       pos = list_next_entry(pos, member))      /*遍历 head 为表头的管理对象实例，pos 为指针*/
```

●void **list_add**(struct list_head *new, struct list_head *head)：将 new 节点插入到 head 节点后面，head 通常表示链表头，此时函数将 new 节点的插入双链表的头部。

●void **list_add_tail**(struct list_head *new, struct list_head *head)：将 new 节点插入到 head 节点前面，head

为表头时，表示将 new 节点插入到双链表末尾。

●void **list_del**(struct list_head *entry)：将 entry 节点从双链表移除。

●void **list_splice**(const struct list_head *list,struct list_head *head)：将 list 双链表中成员插入到 head 双链表头部，实现两个双链表的合并。

●void **list_splice_init**(struct list_head *list,struct list_head *head)：将 list 双链表中成员插入到 head 双链表头部，实现两个双链表的合并，并初始化 list 双链表头。

●void **list_splice_tail**(struct list_head *list,struct list_head *head)：将 list 双链表中成员插入到 head 双链表尾部，实现两个双链表的合并。

●void **list_splice_tail_init**(struct list_head *list,struct list_head *head)：将 list 双链表中成员插入到 head 双链表尾部，实现两个双链表的合并，并初始化 list 双链表头。

3 内核链表

内核链表可认为是带自旋锁的双链表，带锁是为了防止多个进程同时访问双链表带来的冲突。自旋锁保证某一时刻只有一个进程在访问双链表。

内核链表头由 klist 结构体表示，定义在/include/linux/klist.h 头文件：

```
struct klist {
    spinlock_t          k_lock;                 /*自旋锁*/
    struct list_head    k_list;                 /*双链表头*/
    void        (*get)  (struct klist_node *);  /*增加链表中节点引用计数值的函数*/
    void        (*put)  (struct klist_node *);  /*减小链表中节点引用计数值的函数*/
} __attribute__ ((aligned (sizeof(void *))));
```

内核链表头结构中带有保护链表的自旋锁 k_lock 成员，get 和 put 为回调函数指针，用于增加或减少链表中节点成员的引用计数值。

内核链表节点由 klist_node 结构体表示，定义在/include/linux/klist.h 头文件：

```
struct klist_node {
    void            *n_klist;       /*指向表头 klist 结构体，不可直接访问*/
    struct list_head    n_node;     /*双链表节点成员*/
    struct kref         n_ref;      /*节点引用计数，原子变量*/
};
```

内核在/include/linux/klist.h 头文件定义了初始化链表头的宏：

```
#define   KLIST_INIT(_name, _get, _put)             \
{                                                   \
    .k_lock     = __SPIN_LOCK_UNLOCKED(_name.k_lock), \
    .k_list= LIST_HEAD_INIT(_name.k_list),          \
    .get        = _get,                             \
    .put        = _put,             \
}
#define DEFINE_KLIST(_name, _get, _put)         \           /*定义并初始化链表头*/
            struct klist _name = KLIST_INIT(_name, _get, _put)
```

内核在/lib/klist.c 文件内定义了内核链表操作的通用函数，例如：

●void **klist_add_head**(struct klist_node *n, struct klist *k)：将 n 节点添加到 k 表示内核链表的头部。

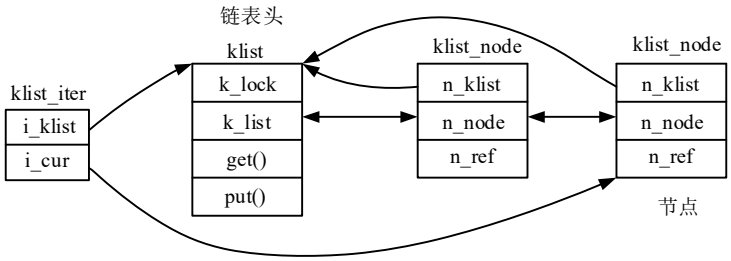
●void **klist_add_tail**(struct klist_node *n, struct klist *k)：将 n 节点添加到 k 表示内核链表的尾部。

●void **klist_del**(struct klist_node *n)：从内核链表中移除节点 n。

内核在/include/linux/klist.h 头文件还定义了 klist_iter 结构体，其内部包含指向内核链表头的指针成员，以及指向当前链表节点的指针成员，klist_iter 结构体定义如下：

```
struct klist_iter {
    struct klist        *i_klist;    /*内核链表头指针*/
    struct klist_node   *i_cur;     /*链表中当前访问节点指针*/
};
```

在引入 klist_iter 结构体后的内核链表，其结构如下图所示：



内核在/lib/klist.c 文件内定义了相关操作函数，例如：

●void klist_iter_init(struct klist *k, struct klist_iter *i)：初始化 i 指向的 klist_iter 实例，i->i_klist = k，i->i_cur =NULL。

●void klist_iter_init_node(struct klist *k, struct klist_iter *i,struct klist_node *n)：初始化 i 指向的 klist_iter 实例，i->i_klist = k，i->i_cur =n。

●struct klist_node *klist_next(struct klist_iter *i)：将前一节点引用计数减 1，后一节点引用计数加 1，返回后一节点指针。

4 散列表

散列表（哈希表）是内核中大量使用的数据结构，散列表可以认为是一个链表数组。它对管理对象进行分类，同类的对象由同一个链表管理。在查找（或添加）管理对象时，先确定对象的分类，然后到分类对应的链表中去查找（添加）对象，以提高效率。

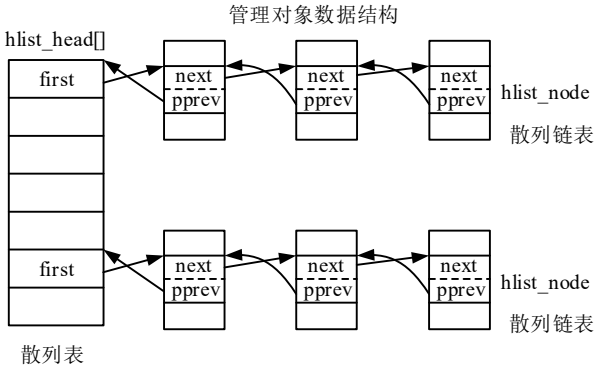
散列表由 hlist_head 结构体数组表示，结构体定义如下（/include/linux/types.h）：

```
struct hlist_head {                 /*散列链表头*/
    struct hlist_node *first;       /*指向第一个链表节点的指针*/
};
```

散列链表中的节点由 hlist_node 结构体表示，定义如下（/include/linux/types.h）：

```
struct hlist_node {
     struct hlist_node *next, **pprev;   /*next 指向下一个节点，pprev 指向前一节点的 next 成员*/
};
```

散列表结构如下图所示：



散列表头 hlist_head 结构中只包含指向节点 hlist_node 结构体的指针，在散列表（数组）较大时可减小内存占用量。散列表节点成员为 hlist_node 结构体，其中 next 成员指向下一节点 hlist_node 结构体，而 pprev 成员是指向指针的指针，其值保存的是前一节点 hlist_node 结构体中 next 成员的地址。

散列表节点 hlist_node 结构体通常内嵌到被管理对象数据结构的内部。内核要用散列表管理某类对象时，需要定义（分配）一个 hlist_head 结构体数组，并定义一个散列函数。散列函数就是通过对象的某个参数确定对象的分类，即添加到哪个散列链表（hlist_head 结构体数组项）。

例如：假设 hlist_head 结构体数组项数为 n，addr 表示对象的地址，散列函数为 hash(addr)=addr%n，即对象地址对 n 取模，表示按对象地址对其进行分类。

内核在/include/linux/list.h 头文件中定义了对散列链表的操作函数，例如：

```
#define    HLIST_HEAD_INIT { .first = NULL }          /*散列链表头初始化*/
#define    HLIST_HEAD(name) struct hlist_head name = {    .first = NULL } /*定义并初始化散列链表头*/
#define    INIT_HLIST_HEAD(ptr)    ((ptr)->first = NULL)      /*初始化散列链表头*/
#define    hlist_entry(ptr, type, member)    container_of(ptr,type,member)    /*hlist_node 转对象指针*/
#define    hlist_for_each(pos, head) \
           for (pos = (head)->first; pos ; pos = pos->next)         /*遍历散列链表中 hlist_node 节点*/

#define hlist_for_each_entry(pos, head, member)                    \
     for (pos = hlist_entry_safe((head)->first, typeof(*(pos)), member);\
          pos;                                          \
          pos = hlist_entry_safe((pos)->member.next, typeof(*(pos)), member))
                                /*遍历散列链表管理的对象实例，pos 为指向对象指针*/
```

●void **hlist_add_head**(struct hlist_node *n, struct hlist_head *h)：将 n 节点添加到散列链表 h 的头部。
●void **hlist_add_before**(struct hlist_node *n,struct hlist_node *next)：将节点 n 添加到节点 next 前面。
●void **hlist_add_behind**(struct hlist_node *n,struct hlist_node *prev)：将节点 n 添加到节点 prev 后面。
●void **hlist_del**(struct hlist_node *n)：从散列链表中移除 n 节点。

内核还定义了一个改进版的散列表，即在操作散列链表时需要获取锁，操作完成后释放锁。
改进版散列表相关数据结构定义在/include/linux/list_bl.h 头文件内（bl 意为 bit lock）：

```
struct hlist_bl_head {          /*散列链表头*/
    struct hlist_bl_node *first; /*指向链表第一个节点，链表中有成员时 first 成员 bit0 为 1（位锁）*/
};
```

```
struct hlist_bl_node {          /*散列链表节点*/
    struct hlist_bl_node *next, **pprev;
};
```

改进版散列链表操作函数简列如下（/include/linux/list_bl.h）：

●void hlist_bl_lock(struct hlist_bl_head *b)：获取散列链表锁。

●void hlist_bl_unlock(struct hlist_bl_head *b)：释放散列链表锁。

●void hlist_bl_add_head(struct hlist_bl_node *n,struct hlist_bl_head *h)：将 n 节点插入散列链表 h 头部。

●void hlist_bl_del(struct hlist_bl_node *n)：从散列链表中移除 n 节点。

```
#define hlist_bl_for_each_entry(tpos, pos, head, member)        \       /*遍历散列链表管理的对象实例*/
    for (pos = hlist_bl_first(head);                            \
         pos &&                                                 \
        ({ tpos = hlist_bl_entry(pos, typeof(*tpos), member); 1;}); \
         pos = pos->next)
```

1.3.3 基数树

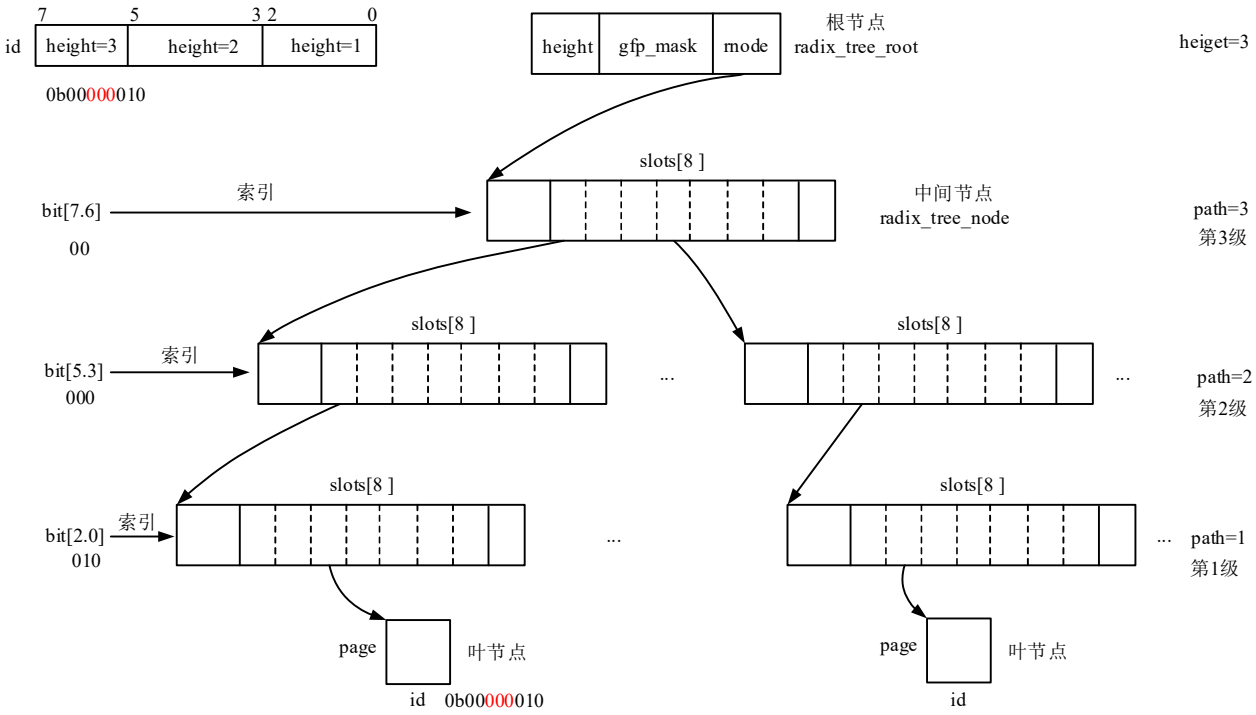
基数树是内核中以库方式实现的数据结构，基数树是一种树状层级结构，它通过数据结构的冗余来提高查找管理对象的速度。

基数树通常管理的是内核用连续整数依次标识（编号）的对象，例如文件地址空间中保存文件内容的页缓存。由于文件内容是按顺序依次保存的，因此缓存页（物理页）也需要按顺序依次编号。

下面先以简单的例子说明基数树的原理，然后列出内核基数树数据结构及接口函数，具体函数源代码请读者结合基数树原理自行阅读。

1 基数树原理

假设，内核用一个 8 比特的数标识某类对象，创建对象时，对象 id 值从 0 开始依次编号。



如上图中所示，8 比特的 id 值按 3 比特位宽，划分为 3 个位段（从低位往高位划分），最高位位段为 2 比特，即 bit[7.6]。基数树是一个层级的树状结构，包含一个根节点（由 radix_tree_root 结构体表示），若干个中间节点（由 radix_tree_node 结构体表示）和叶节点，叶节点即管理对象，且管理对象只能是叶节点。

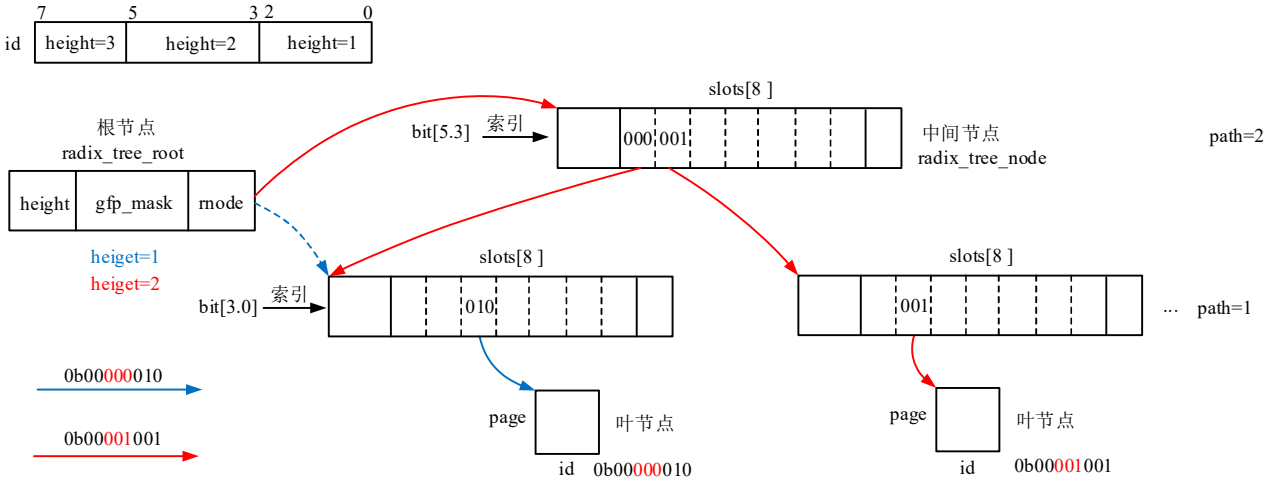
基数树中间节点中包含一个指针数组，数组项数为位段表示二进制数最大值加 1。例如，位段宽度为 3 比特时，数组项数为 2^3=8（数组项索引值最大为 7）。指针数组项指向下一级节点或叶节点。

管理对象根据其 id 值各位段的值，从基数树根节点指向的中间节点开始。从高到低，以各位段表示的二进制数值为索引值，在基数树中从上至下对应各节点指针数组项，第 1 级节点指针数组项指向对象。

基数树的最大高度 height 值为 id 位段数量值，高度值表示中间节点层级最大数量（不含叶节点和根节点）。中间节点所处的高度值，从下往上从 1 开始依次增加。

通常管理对象 id 值是从 0 开始依次分配的，基数树也是从底层往上依次扩展的。例如，假设基数树中只有一个 id 值为 0 的对象，则只需要一个中间节点即可，节点指针数组项中第 0 项指向对象，不需要其它层级的节点。下面以示例的形式，说明向基数树增加节点的方法，假设先增加 id 值为 0b00000010 的对象，然后增加 id 值为 0b00001001 的对象。

基数树初始状态为空，即只有根节点，根节点指向中间节点的指针为空。现在增加 id 值为 0b00000010 的对象，由于其在基数树 2、3 级对应位段值都为 0，不需要增加 2、3 级节点，第 1 级位段值为 2，因此只需要一个中间节点即可（第 1 级）。如下图中蓝色箭头所示，创建一个中间节点，根节点指向它，节点中指针数组第 2 个数组项指向对象，此时基数树高度为 1。



现在再向基数树增加 id 值为 0b00001001 的对象，其第 3 级位段值为 0（不需要增加此级节点），第 2 级位段值为 1，第 1 级位段值为 1，因此需要增加第 2、1 级节点。如上图中红色箭头所示，先增加第 2 级节点，根节点指向它，节点第 0 个数组项指向之前创建的节点，然后还要增加管理 0b00001001 对象的第 1 级节点，使其第 1 个指针数组项指向对象。

基数树的高度（中间节点级数）决定了基数树当前管理对象 id 值的最大值，如果添加的 id 值超过了最大值，则需要向上扩展基数树层级，否则可能只需要增加下级节点。

假设位段宽度为 RADIX_TREE_MAP_SHIFT，则高度为 1 的基数树能管理的对象 id 值最大值为（$2^{RADIX\_TREE\_MAP\_SHIFT}$-1），高度为 2 的基数树管理 id 值最大为（$2^{2*RADIX\_TREE\_MAP\_SHIFT}$-1），依此类推。基数树达到最大高度时，最大值受 id 值位宽的限制。

查找对象就比较简单了，首先被查找对象的 id 值不能超过基数树当前高度管理对象 id 值的最大值，然后从上至下依次取出 id 值中与节点层级对应的位段，以位段表示的二进制数值为索引值，依次检索节点指针数组项，到达第 1 级节点时，其指针数组项指向的就是查找对象。

2 内核实现

内核基数树相关数据结构和宏定义在/include/linux/radix-tree.h 头文件，基数树基本操作函数定义在库文件/lib/radix-tree.c。

内核中基数树参数定义如下（/include/linux/radix-tree.h）：

```
#ifdef __KERNEL__
        #define   RADIX_TREE_MAP_SHIFT      (CONFIG_BASE_SMALL ? 4 : 6)    /*位段宽度*/
                /*选择了 BASE_FULL，CONFIG_BASE_SMALL 为 0，否则为 1，/init/Kconfig*/
#else
        #define   RADIX_TREE_MAP_SHIFT   3     /* For more stressful testing */
#endif

#define   RADIX_TREE_MAP_SIZE      (1UL << RADIX_TREE_MAP_SHIFT)
                                                    /*节点 slots[]数组项数*/
#define   RADIX_TREE_MAP_MASK    (RADIX_TREE_MAP_SIZE-1)      /*位段掩码*/

#define   RADIX_TREE_INDEX_BITS    (8 * sizeof(unsigned long))
#define   RADIX_TREE_MAX_PATH    (DIV_ROUND_UP(RADIX_TREE_INDEX_BITS, \
                                        RADIX_TREE_MAP_SHIFT))       /*位段数量*/
```

内核定义了 **height_to_maxindex**[RADIX_TREE_MAX_PATH + 1]数组，表示基数树在各个高度时，管理对象 id 值的最大值（管理对象数量的最大值）。高度为 0 时，数组项值为 0，基数树初始高度为 0，表示管理对象为 0。

内核在 start_kernel()函数内调用 radix_tree_init(void)函数对基数树数据结构进行初始化，代码如下：

```
void __init radix_tree_init(void)
{
    radix_tree_node_cachep = kmem_cache_create("radix_tree_node",sizeof(struct radix_tree_node), 0,
            SLAB_PANIC | SLAB_RECLAIM_ACCOUNT,radix_tree_node_ctor);  /*创建 slab 缓存*/
    radix_tree_init_maxindex();    /*设置 height_to_maxindex[]数组项，/lib/radix-tree.c*/
    hotcpu_notifier(radix_tree_callback, 0);
}
```

内核基数树根节点由 radix_tree_root 结构体表示，定义如下：

```
struct radix_tree_root {
    unsigned int        height;             /*当前基数树的高度，即中间节点层级数*/
    gfp_t               gfp_mask;           /*分配节点标记（物理内存分配标记）*/
    struct radix_tree_node   __rcu *rnode;  /*指向节点*/
};
```

radix_tree_root 结构体一般内嵌于其它结构中，用于指向树根节点。初始化根节点的宏如下：

```
#define RADIX_TREE_INIT(mask)   {           \
    .height = 0,                            \       /*当前高度为 0*/
    .gfp_mask = (mask),                     \
    .rnode = NULL,                          \       /*指向 NULL*/
}

#define RADIX_TREE(name, mask) \
    struct radix_tree_root name = RADIX_TREE_INIT(mask)      /*定义并初始化根节点*/
```

内核基数树中间节点由 radix_tree_node 结构体表示，定义如下：

```
struct radix_tree_node {
    unsigned int    path;               /*节点所在层级，以及在父节点中指针数组项索引值*/
    unsigned int    count;              /*其下子节点数量*/
    union {
        struct {
            struct radix_tree_node *parent;     /*指向父节点*/
            void *private_data;                 /* For tree user */
        };
        struct rcu_head     rcu_head;           /*释放节点时的回调函数*/
    };
    struct list_head    private_list;               /* For tree user */
    void __rcu      *slots[RADIX_TREE_MAP_SIZE];    /*指针数组*/
    unsigned long tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS]; /*标签数组*/
```

};

radix_tree_node 结构体中主要成员简介如下：

●**path**：这个成员分两部分，低位部分表示节点所处层级，高位部分表示节点在其父节点指针数组中的索引值。

●**count**：其下子节点数量，即 slots[]指针数组项中不为空的项数。

●**slots[]**：指针数组，可能指向下一级中间节点，也可能指向管理对象。

●**tags[][]**：二维数组，表示标签值，见下文。

内核基数树操作函数简列如下（/lib/radix-tree.c）：

●int **radix_tree_insert**(struct radix_tree_root *root,unsigned long index, void *item)：向基数树添加 id 值为 index 的对象，item 为对象指针，root 表示根节点，成功返回 0。

在插入对象前函数会判断是否需要扩展基数树，如果需要则先扩展基数树，然后从根节点指向的节点开始往下逐级搜索，直至第 1 级，使第 1 级节点对应指针数组项指向对象，各级中尚不存在的节点将被创建。

●void ***radix_tree_lookup**(struct radix_tree_root *root, unsigned long index)：查找 id 值为 index 的对象，返回对象指针。

●void **radix_tree_delete**(struct radix_tree_root *root, unsigned long index)：从基数树中移除 id 值为 index 的对象，返回对象指针，其间可能需要释放中间节点 radix_tree_node 实例。

基数树节点 radix_tree_node 结构体中还有一个重要成员，那就是 tags[][]二维标签数组：

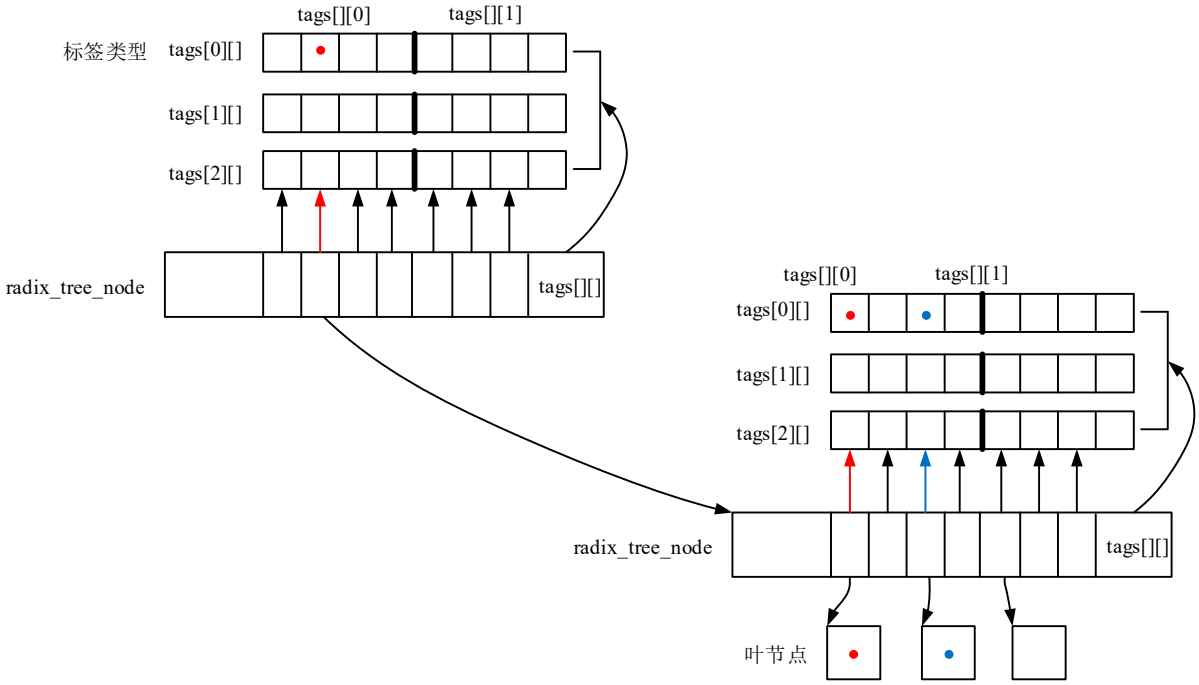
```
struct radix_tree_node {
    ...
    void __rcu      *slots[RADIX_TREE_MAP_SIZE];
    unsigned long tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS]; /*标签数组*/
};
```

标签数组项是一个二维数组，RADIX_TREE_MAX_TAGS 表示标签类型数量，每种标签类型对应一个位图。位图中每一位对应节点指针数组 slots[RADIX_TREE_MAP_SIZE]中的一项，指针数组项下的某个或多个对象具有该属性则设置相应标记位。

内核支持的标签类型数量为 3，定义如下：

```
#define   RADIX_TREE_MAX_TAGS   3      /*标签类型数量*/
#define   RADIX_TREE_TAG_LONGS    \    /*每类标签位图所占的整型数数量，位图*/
    ((RADIX_TREE_MAP_SIZE + BITS_PER_LONG - 1) / BITS_PER_LONG)
```

标签类型可以理解为节点下管理对象的某种属性，标签数组与指针数组对应关系如下图所示：



基数树中每个 radix_tree_node 实列中都包含二维的标签数组成员。若某一叶节点（对象）具有某个类型属性（上图中类型值为 0），则需要设置从根节点指向的节点开始直到叶节点的路径中的所有中间节点，对应类型的标签位，如上图中所示。设置标签位时，只要其下一个对象具有该属性就设置，清除标签位时，需要其下所有对象都不具有该属性时才清除。

设置和清除标签位的函数如下：

●void ***radix_tree_tag_set**(struct radix_tree_root *root,unsigned long index, unsigned int tag)：设置标签位，tag 参数为标签类型索引值，index 为对象 id 值，返回对象指针。

●int **radix_tree_tag_get**(struct radix_tree_root *root,unsigned long index, unsigned int tag)：获取 id 值为 index 对象的 tag 类型标签值，设置返回 1，否则返回 0。

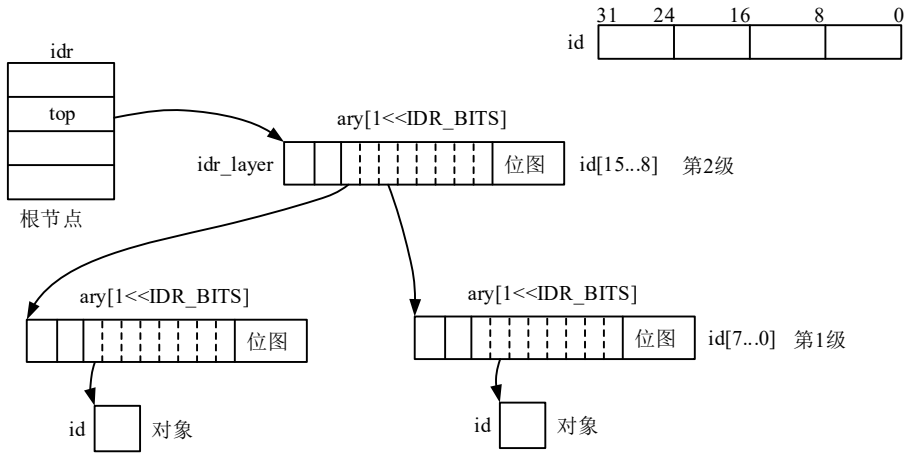
●void ***radix_tree_tag_clear**(struct radix_tree_root *root,unsigned long index, unsigned int tag)：清除 id 值为 index 对象的 tag 类型标签位，只有同一节点下的所有对象都清除了 tag 类型标签，才清除上一级节点中对应的标签位。

1.3.4 idr/ida

idr 数据结构与基数树非常类似，可以认为是一种简化版的基树数。内核固定 idr 中 id 值的位段宽度为 8 比特，以使一个节点能管理足够多的对象。对于对象比较少的应用，只需要一个中间节点即可。idr 最大高度为 4（32 位系统）。ida 是 idr 的一个包装器，在介绍 idr 之后将简要介绍 ida。

idr/ida 相关数据结构及宏定义在/include/linux/idr.h 头文件，相关函数定义在/lib/idr.c 文件内。

idr 结构如下图所示：



idr 中间节点由 idr_layer 结构体表示，其中包含指针数组，以及位图等信息。位图中的比特位与指针数组一一对应，表示指针数组项是否为空（是否已被使用），用于分配对象 id 值。指针数组项数为 1<<IDR_BITS，其中 IDR_BITS 为 8。

idr 根节点由 idr 结构体表示，结构体定义如下（/include/linux/idr.h）：

```
struct idr {
    struct idr_layer __rcu    *hint;     /*最后分配 id 值的中间节点*/
    struct idr_layer __rcu    *top;      /*最高层节点*/
    int               layers;    /*idr 高度*/
    int               cur;       /* current pos for cyclic allocation */
    spinlock_t            lock;
    int               id_free_cnt;
    struct idr_layer      *id_free;
};
```

根节点 idr 结构体通常嵌入到内核其它数据结构中，定义并初始化 idr 根节点的宏及函数，如下所示：

```
#define   DEFINE_IDR(name)   struct idr name = IDR_INIT(name)
#define   IDR_INIT(name)                  \
{                                         \
    .lock           = __SPIN_LOCK_UNLOCKED(name.lock),    \
}

void idr_init(struct idr *idp)      /*初始化根节点函数*/
{
    memset(idp, 0, sizeof(struct idr));
    spin_lock_init(&idp->lock);
}
```

idr 中间节点由 idr_layer 结构体表示，定义如下：

```
struct idr_layer {
    int               prefix;    /*id 值前缀*/
    int               layer;     /*节点所在层级*/
    struct idr_layer __rcu    *ary[1<<IDR_BITS];      /*指针数组*/
    int               count;     /*子节点数量*/
```

```
    union {
        DECLARE_BITMAP(bitmap, IDR_SIZE);   /*位图，表示指针数组空闲状态*/
        struct rcu_head         rcu_head;
    };
};
```

内核在/lib/idr.c 文件内定义了 idr 初始化函数，如下所示：

```
void __init idr_init_cache(void)
{
    idr_layer_cache = kmem_cache_create("idr_layer_cache",   \
            sizeof(struct idr_layer), 0, SLAB_PANIC, NULL);   /*为 idr_layer 结构体创建 slab 缓存*/
}
```

idr 与基数树最大区别可能就是 idr 中对象的 id 值是动态分配的，而不是像基数树一样是依次编号的。在向 idr 添加对象时，可指定对象 id 值的区间，由 idr 动态分配 id 值。idr 操作函数简列如下：

●int **idr_alloc(struct idr *idp, void *ptr, int start, int end, gfp_t gfp_mask)**：添加对象，并分配 id 值，idp 指向根节点，ptr 为对象指针，[start,end)表示可分配 id 值的区间，gfp_mask 为分配内存掩码，函数成功返回对象分配的 id 值。在分配 id 值时，可能需要扩展 idr，并创建中间节点。

●void ***idr_find**(struct idr *idr, int id)：在 idr 中查找指定 id 值的对象，返回对象指针，若对象为空，返回 NULL。

●void **idr_remove(struct idr *idp, int id)**：从 idr 中移除指定 id 值的对象。

ida 数据结构是 idr 数据结构的包装器，它只用来分配 id 值，不将对象关联到 idr 结构中。ida 根节点结构定义如下：

```
struct ida {
    struct idr      idr;        /*idr 根节点*/
    struct ida_bitmap   *free_bitmap;       /*ida_bitmap 结构体指针*/
};
```

ida 数据结构操作函数简列如下：

●void **ida_init**(struct ida *ida)：初始化 ida 实例，即初始化 idr 结构体成员。

●int **ida_get_new**(struct ida *ida, int *p_id)：分配 id 值，保存至 p_id 地址指向的内存。

●int **ida_simple_get**(struct ida *ida, unsigned int start, unsigned int end,gfp_t gfp_mask)：在[start,end)区间分配新 id 值，返回分配的 id 值（函数内在持有锁的情况下调用 ida_get_new()函数）。

●int **ida_get_new_above**(struct ida *ida, int starting_id, int *p_id)：分配大于等于 starting_id 值的新 id 值，p_id 保存 id 值，成功返回 0。

●void **ida_remove**(struct ida *ida, int id)：释放 id 值。

●void **ida_simple_remove**(struct ida *ida, unsigned int id)：在持有锁的情况下调用 ida_remove()函数释放 id 值（此 id 值由 ida_simple_get()函数分配）。

1.3.5 红黑树

红黑树是内核使用的一种非常重要的管理数据结构，内核非常多的重要的数据结构实例都由红黑树管理，它在速度和实现复杂度之间提供了一种很好的平衡。红黑树是同时具有以下特征的自平衡二叉查找树：

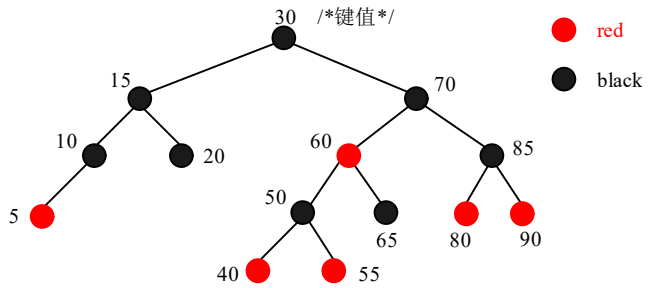
（1）树中节点是红色或者黑色；

（2）根节点是黑色的；

（3）红节点的所有子节点是黑色的；

（4）每条从根节点到叶节点的简单路径上包含相同数目的黑节点。

二叉树中每个节点最多只有两个子节点，且节点的左子节点键值比节点键值小，右子节点键值比节点键值大。红黑树满足二叉树的条件，另外每个节点还带着色，或红色或黑色。树中不能有两个连续的红色节点（父子节点不能同时为红色），但可以有任意连续的黑色节点。在红黑树中插入和删除节点时需要保持树的平衡性，即不能有某一节点的深度远大于其它节点。下图示意了一个红黑树的例子，节点旁边数字是节点管理对象的键值：



内核在/include/linux/rbtree.h 头文件内定义了红黑树根、节点数据结构，如下所示：

```
struct rb_root {        /*红黑树根节点*/
    struct rb_node *rb_node;      /*指向节点*/
};

struct rb_node {          /*红黑树节点*/
     unsigned   long   __rb_parent_color;  /*父节点颜色，包含父节点指针*/
     struct rb_node   *rb_right;       /*指向右子节点*/
     struct rb_node   *rb_left;        /*指向左子节点*/
} __attribute__((aligned(sizeof(long))));
```

红黑树节点结构__rb_parent_color 成员最低位用于表示父节点颜色（0 红，1 黑），屏蔽低两位后表示父节点指针。

内核中以库函数的形式实现红黑树的通用操作函数，如插入、删除节点等，函数实现在/lib/rbtree.c 文件内，下面列出其中主要的通用操作函数：

●void **rb_link_node**(struct rb_node *node, struct rb_node *parent,struct rb_node **rb_link)：关联父节点，node 节点父节点设为 parent，node 左右指针设为 NULL，*rb_link 设为 node 值（*rb_link = node）。在插入对象时，需先确定父对象，然后调用 rb_link_node()设置父节点，最后调用下面的 rb_insert_color()函数将节点插入红黑树。

●void **rb_insert_color**(struct rb_node *, struct rb_root *)：插入节点，插入前需调用 rb_link_node()函数关联父节点。

●void **rb_erase**(struct rb_node *, struct rb_root *)：从红黑树中移除节点。

●struct rb_node *rb_next(const struct rb_node *)：查找指定节点的下一个节点，遍历树时使用。

●struct rb_node *rb_prev(const struct rb_node *)：查找指定节点的前一个节点。

●struct rb_node *rb_first(const struct rb_root *)：查找树中第一个节点。

●struct rb_node *rb_last(const struct rb_root *)：查找树中最后一个节点。

红黑树节点结构 rb_node 一般内嵌在被管理数据结构内部，如进程虚拟内存域结构：

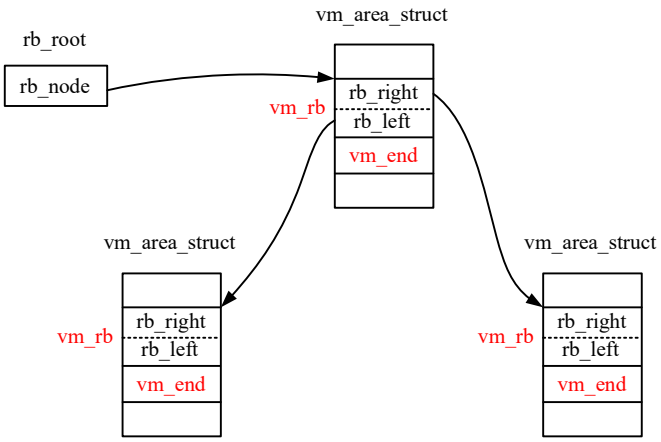
```
struct vm_area_struct {
    unsigned long    vm_start;          /*内存域起始地址*/
    unsigned long    vm_end;            /*内存域起始地址，键值*/
    ...
    struct rb_node   vm_rb;             /*红黑树节点*/
    ...
}
```

由 rb_node 实例指针通过容器机制可获得被管理数据结构实例指针：

```
#define rb_entry(ptr, type, member)    container_of(ptr, type, member)          /*/include/linux/rbtree.h*/
                                       /*ptr：rb_node 指针，type：被管理结构指针，rb_node 在 type 内成员名称*/
```

例如，假设 ptr 是指向 rb_node 实例指针，rb_entry(ptr,vm_area_struct,vm_rb)返回的就是包含 ptr 指向的 rb_node 实例的 vm_area_struct 实例指针。

通常管理对象数据结构中某一成员被当作键值，用作对象中红黑树节点成员在红黑树中排序的依据。例如：虚拟内存域 vm_area_struct 实例中以结束地址 vm_end 为键值，将所含的红黑树节点成员 vm_rb 按照实例中 vm_end 值从小到大在红黑树中从左至右排序。如下图所示，树中左侧节点 vm_end 值最小，右侧节点 vm_end 值最大。



查找红黑树节点操作由应用红黑树的代码实现，查找符合某一条件节点时，以给定键值为基准，从红黑树根节点开始查找管理对象实例，比给定键值小的对象在节点的左子树中，比给定键值大的对象在节点的右子树中，递归查找即可。例如：在 vm_area_struct 实例红黑树中查找结束地址在 addr 之后的第一个实例时，从根节点开始通过容器机制获取节点代表的 vm_area_struct 实例，再将实例中 vm_end 成员值与 addr 比较，vm_end 比 addr 小的实例在节点左子树中，比 addr 大的实例在节点右子树中，如此递归查找即可。

将管理对象数据结构实例插入红黑树之前，先要依键值查找红黑树确定插入的位置，即父节点，而后依次调用 rb_link_node()和 rb_insert_color()函数将节点插入红黑树。

删除操作在获取管理对象数据结构实例 rb_node 成员指针后，调用 rb_erase()函数从红黑树中移除节点。在通用的插入/移除节点操作函数中，在插入、移除对象后需要对树进行旋转、改变节点颜色等操作，以满足红黑树的条件，有兴趣的读者可阅读/lib/rbtree.c 文件内源代码。

若红黑树节点中包含额外的信息数据，可使用被称为扩展的红黑树。扩展的红黑树在向树中添加/移除节点时会回调用户提供的函数，以处理节点的扩展信息数据。若要使用扩展的红黑树，程序代码中需包含头文件/include/linux/rbtree_augmented.h，在头文件中定义了传递回调函数的数据结构：

```
struct rb_augment_callbacks {
    void (*propagate)(struct rb_node *node, struct rb_node *stop);
    void (*copy)(struct rb_node *old, struct rb_node *new);
    void (*rotate)(struct rb_node *old, struct rb_node *new);
};
```

rb_augment_callbacks 中各函数指针成员简介如下：

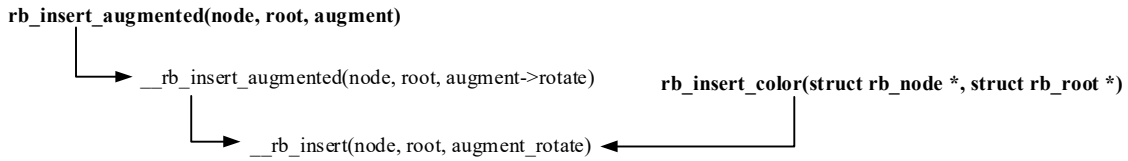
●**propagate**：更新 node 节点及其所有祖先节点的扩展信息（stop==NULL），或直至 stop 指向的节点；

●**copy**：将 old 节点的扩展信息复制给 new 节点；

●**rotate**：将 old 节点的扩展信息复制给 new 节点，并重新计算 old 节点的扩展信息。

扩展的红黑树与标准的红黑树之间的区别在于插入、删除节点的操作函数不同：

●void rb_insert_augmented(struct rb_node *node, struct rb_root *root,const struct rb_augment_callbacks *augment)：插入节点接口函数，augment 参数为 rb_augment_callbacks 实例指针，函数调用关系如下图所示：



函数内在执行标准的红黑树插入节点操作后，会调用 rb_augment_callbacks 实例内的 rotate()函数。标准的红黑树插入节点函数 rb_insert_color()在内部调用的也是__rb_insert()函数，只不过此时调用的 rotate()函数为空函数。

●void rb_erase_augmented(struct rb_node *node, struct rb_root *root,const struct rb_augment_callbacks *augment)：移除节点函数，内部可能会调用 rb_augment_callbacks 实例定义的三个函数。

内核在头文件/include/linux/rbtree_augmented.h 给出了定义 rb_augment_callbacks 实例的宏：

```
#define    RB_DECLARE_CALLBACKS(rbstatic, rbname, rbstruct, rbfield,    \
                                                    rbtype, rbaugmented, rbcompute)      \
/*
*rbstatic：rb_augment_callbacks 实例修饰符，通常为 static；rbname：rb_augment_callbacks 实例名称；
*rbstruct：节点管理数据结构名称，如 vm_area_struct；
*rbfield：rbstruct 结构中 rb_node 成员名称，如 vm_rb；
*rbtype：表示扩展信息的数据类型，如 unsigned long；
*rbaugmented：表示扩展信息的 rbstruct 数据结构成员，如 rb_subtree_gap；
*rbcompute：重新计算扩展信息的函数指针。
*/
static inline void                                                  \
rbname ## _propagate(struct rb_node *rb, struct rb_node *stop)      \    /*定义 propagate()函数*/
{                                                                   \
    while (rb != stop) {                                            \
        rbstruct *node = rb_entry(rb, rbstruct, rbfield);       \
        rbtype augmented = rbcompute(node);                     \
        if (node->rbaugmented == augmented)                     \
```

```
                break;                                          \
            node->rbaugmented = augmented;                      \
            rb = rb_parent(&node->rbfield);                     \
        }                                                       \
    }                                                           \
    static inline void                                          \
    rbname ## _copy(struct rb_node *rb_old, struct rb_node *rb_new)   \   /*定义 copy()函数*/
    {                                                           \
            rbstruct *old = rb_entry(rb_old, rbstruct, rbfield);    \
            rbstruct *new = rb_entry(rb_new, rbstruct, rbfield);    \
            new->rbaugmented = old->rbaugmented;                \
    }                                                           \
    static void                                                 \
    rbname ## _rotate(struct rb_node *rb_old, struct rb_node *rb_new)   \   /*定义 rotate()函数*/
    {                                                           \
            rbstruct *old = rb_entry(rb_old, rbstruct, rbfield);    \
            rbstruct *new = rb_entry(rb_new, rbstruct, rbfield);    \
            new->rbaugmented = old->rbaugmented;                \
            old->rbaugmented = rbcompute(old);                  \
    }                                                           \
    rbstatic const struct rb_augment_callbacks rbname = {           \   /*定义 rb_augment_callbacks 实例*/
            rbname ## _propagate, rbname ## _copy, rbname ## _rotate  \
    };
```

宏内部定义了 rb_augment_callbacks 结构体实例及其内部的三个函数，rbcompute 参数表示计算节点扩展信息的函数指针。应用扩展红黑树时，需定义 rbcompute()函数，调用以上宏定义 rb_augment_callbacks 结构体实例，随后即可调用 rb_insert_augmented()和 rb_erase_augmented()函数插入和删除节点，注意需要将 rb_augment_callbacks 实例的指针传递给函数。第 4 章进程虚拟内存域 vm_area_struct 实例的管理就是应用扩展红黑树的一个例子。

本节介绍了内核常用的管理数据结构，如单链表、双链表、散列表、基数树、idr、红黑树等，这些管理结构用来管理内核各种类型的对象。其中，双链表、散列表、红黑树在内核中使用尤其广泛，请读者理解以上管理结构，以便后面内核源代码的学习。

1.4 小结

本章介绍了计算机、操作系统的一些理论知识，以使读者理解操作系统是系统的大管家，管理着系统资源和进程。操作系统负责完成进程提出的各种资源操作申请，并对进程进行管理、调度等操作。

内核完成操作系统最基础、最核心的功能，内核配上一个根文件系统、用户工具、配置文件、用户程序等，就构成了一个完整的操作系统。

Linux 内核是使用广泛、免费、开源的操作系统内核。本章介绍了 Linux 内核的框架、各部分组成及功用，最后介绍了内核常用的管理数据结构，以便为后面阅读内核源代码打下基础。

通过本章的学习，希望读者能对内核的框架和主要功能有个整体的认识。内核源代码庞大而复杂，最好采取先整后分的学习方法，先对内核各部分（子系统）的框架和功用有个整体的认识，了解各部分之间的关系，然后再去详细研究各功能（子系统）代码。没有整体的认识，很容易就迷失在众多的代码细节之中。

内核源代码的学习是一项艰苦而卓绝的工程，需要超强的耐心和毅力，心平气和的心态，以及时刻清醒的头脑。下一章将真正开始内核源代码的学习，你准备好了吗？准备好了那就开始吧！