Linux 内核实践之路

——给那些想从 LiNuX 内核找点乐趣的人。

一个不能回避的尴尬问题：研究 Linux 内核是不是必须要通过研读那些错综复杂的“邪恶”代码，才能真正理解它？

关于方法

“术业要有专攻”。还记得大学时候训练英语的听说能力。每天到多媒体教室上一个多小时的课，但是一个学期下来，英语听说水平不但没有任何进展，还导致了对英语学习的厌恶，直到最后不了了之。后来实在不行，有时候需要用，用到时无法交流就是无比的尴尬，脸上绝对是“火辣辣的”！后来有个室友说背新概念管用，于是背了几篇，还是不行，交流的时候还是“哑口无声”，无话可说。室友又说这个要把一本书背得滚瓜烂熟才行，那样才能做到脱口而出。大道理谁都明白，能做得到的没几个，我相信在我没背滚瓜烂熟之前，我早就泄气了。

如果你学习一样东西，在较短的时间内都不能应用，不能让你体验一下小有成就的感觉，那么能够坚持下来的人就会很少。学习的动力一开始可能是兴趣，可能是好奇，也可能是某种非自愿。但无论如何一开始都是志在必得，胸有成竹的，至少是相信自己不会半途而废。但是不好的方法会很快耗尽最初的激情，而最坏的结果是让你放弃，让你认为这个东西太难了，可能根本不适合自己。这里不是讲革命大义，但是方法论在小事上依然有效。那么什么方法算得上是好的方法呢？适合这个人的方法就能适合其他人吗？那么每个人都要一套独到的方法了？好的方法总有一些通用性，不然也就不能认为是好的方法了。笔者认为它应该具有以下的特性：

1．提纲挈领，由浅入深，深入浅出。大部分当前还能存在的方法基本上都具有这个特点，如果你要学习英语，那么就要从 26 个英文字母开始学习，当然了对于英语国家的人，是从咿呀学语开始，但无论如何，没有人一开始就看虚拟语气，就探讨英美发音。好的开端，意味着成功的一半。

2．可持续性。保持学习的动力，进一步激发学习的兴趣。这一点是非常之重要。背新概念固然可以学习好的句法，逻辑性，当然还有些新的单词，但是如果用它来学习口语，则确实有些方法失当。我记得我最后学习口语的方法采用的是英语九百句，它把句子分类，比如交际问候，待人接物等，每一个分类都是一组对话：我累了，我烦了，我喜欢，我恨你，我受不了，我想骂人……然后一天学习几组对话，跟着 MP3 一句一句的重复，然后等到周末就可以去英语角找老外胡侃了，你知道当你敢于站在别人大声说出你背了几十遍的英语后，会感觉到多么的成功，尽管那又是多么的微不足道。最终这就促成了好的循环，这种循环不仅使你逃脱了学习过程的乏味，还让你感觉到自己找到了好的方法，树立了自信！在 Linux 内核代码学习中不好的方法之一就是看满篇代码引用的书籍或者资料。

3．循序渐进性，能够自然而合理的划分学习阶段，这与第二个特点相互呼应。多数时候这要根据个人的学习进展来决定，但是好的方法能够自然而然的体现学习的阶段性，阶梯型。与此同时在无形中要限制学习的速度，这一点是为了牵制大多数人急于求成的心态。很多人，包括笔者，之所以走弯路，最后走上放弃的道路，是因为“急功近利”。总想着走捷径，速成，最后就是“欲速则不达”！切记切记，慢即是快的道理。笔者一直认为将一本书看一百

遍和草草看一百本书的人是完全生活在不同的精神境界。再回到学英语的例子，那些不同的对话分类并不是随随便便的，而是有承前启后，步步上升作用的。比如一开始是家人之间的对话，简单到一两个单词，最后则是商业交际，涉及到金融贸易。在 Linux 内核代码学习中不好的方法之二就是好高骛远，一口气，两三天把想把问题搞个一清二楚。

4．具有普遍性，也即针对大多数人，它都是有效的。我想如果一个方法如果具备了以上的特性，那么这个方法就基本上具备了普遍性，至少适应一个地区，或者一个年龄层次的人没有问题。当然为了针对特定的人群，可能要一些方式的变化：比如 26 个英文字母，对于成年人只要给出音标，写法即可，但对于学龄前儿童，则要用图画颜色来表达。在 Linux 内核代码学习中，多画图，多总结则具有一般性。另外有的人有了很好的汇编或者硬件基础就可以从 Bootloader 开始看，有的人则不行，但是可以从相对简单的内核功能模块看起，这也具有一般性。

5．与时俱进。这是方法论的最可怕之处，也是理想方法的终极境界，对于个人的大多数事情，应该不需要到这个层次，当然达到这个境界的方法正无时无刻不在地球的各个地方发挥着作用，否则那些方法随着时间必将淘汰。与时俱进的方法体现在 Linux 内核代码学习上，需要到达一定的程度，此时已经具有了前瞻性，探索性。

好了，说了许多似乎和学习 Linux 内核还没有沾太大边呢！笔者并不建议一个只写过少许用户空间 C 程序的人直接去看内核代码，因为用户空间和内核空间相差的不仅仅是一个系统调用。笔者认为一个人有看内核代码的想法或者需求有以下几种：追踪 BUG 追到了内核；内核暴力不合作，抛出 oops 信息拒绝继续执行；Linux 到底干了啥的好奇心。还有一种可能就是看着别人研究内核，自己也要试试，盲从一把。不管是哪种情况，此时你可能要自顾自得折腾一番，然后大多数时候灰头土脸的回来了，然后在头大的同时得出一个结论：那帮家伙也太牛 x 了，这玩意咋搞出来的？此时你如果比较幸运就会开始思考学习内核的方法了，但是多数时候是胡乱买了一些书，然后看了几(“几”很有可能是“一”)章之后这本书便被束之高阁了。怎么办呢？有什么好方法？

如果你已经在 Linux 平台上工作或者学习了很久，至少大多数 Shell 命令用的还可以，对 Linux 这个黑盒子已经有了自己的感觉和理解，那么这确实是一个好的开端。很多计算机系的人在大学毕业时很可能都不具备这个基础。到这里，你如果自认为还没达到这个级别，那么请回头打牢基础，否则笔者就要害人走弯路了，这个阶段至少要有半年，记住“经济基础决定上层建筑”，现在你的基础有多深厚基本决定了你以后理解内核的效率，准确性以及能够达到的深度。

对 Linux 的初步感觉是，它和 Windows 基本上是两码事，这个东西的命令很强劲，但是要记住的东西也挺多。如果你平时的工作是在 Linux 上编写应用程序，那么已经自然而然进入第二阶段了。无论是从工作的积累，还是通过参考书籍，互联网资源以及同事间的探讨，这个阶段都要尽量做到以下几件事：

1． 能够熟练和准确的运用 C 语言的语法和技巧，熟悉 GCC 的一些特性，一些更深入的要求是能够研究一些 glibc 库函数的实现，这样会帮助你对神秘的 glibc 有实际的理解，与此同时它能使你养成严谨代码的写法。
2． 了解一些编译器的用法，如果是嵌入式开发，那么这是必须的。首先了解这些选项的作

用和优化的原理，在此基础上了解 GCC 编译的整个过程：预处理，汇编以及链接。更深一步的要求是对链接器的原理，ELF 文件的构造和特定 CPU 架构的汇编语言有正确的理解和全局的把握。

3. 尽管已经罗列了这么多，似乎还和 Linux 内核没沾上大的边。第三点就是在这个阶段编程时尽量广泛的使用 Linux 上不同应用的系统调用：

文件操作：读写，链接，权限控制，文件锁。

进程间通信：信号，管道，套接字。

网络应用：TCP/IP，UDP，RAW Socket，NETLINK 等。

多进程和线程的应用等。

其中进程间通信和网络应用是 Linux 的精华所在，如果错过了它们，可以说是一很大的遗憾。

尽管上面罗列的这些知识点已经让人望而却步，但是把它们放在一个较长的时间阶段里来实现，是相当现实的。如果你是一个刚刚接触 Linux 不久的新手，可能要惊叹：哇塞，还是算了吧，以后日子太苦了。没有对 Linux 的热情，那么又何必把精力放在这里呢，放在自己喜欢的领域将会有更大的成就。当然只有热情是远远不够的，如果没有长期的可实践的计划，那么和做一场梦有什么区别呢？切记“慢即是快”。

在第二个阶段完成的代码量至少要几万行，并且它们最终看起来和开源项目的代码间没有明显的区别。另外应该在这段时间形成自己的经验代码，最好形成一个或者几个库：针对网络的，文件操作的，锁机制的，进程间通信的等等！库中的代码应该清晰明了，和标准的函数库没有本质的区别，你可以熟练引用这些库函数来实现应用并能大概讲清楚它实现的流程。

好了，百川东到海，在经历了一年或者几年(具体的时间并不重要，但是要记住，欲速则不达——再次强调)的 Linux 平台的 C 语言编程之后，在此时，当然也很有可能是在第二阶段的某些时候，你肯定蠢蠢欲动了。最后的探宝之路开始了，并且你要相信过去的那些日子已经让你几乎最高效率的来到了埋藏宝贝的城堡门前，你已经走了一条捷径。

尽管大部分的项目已经不能难住你，还是要清楚的知道：这才是开始。只有抱有谦逊的心，才能走的更远。在此时：

1. 你应当(笔者认为是必须)准备一块开发板，它最好能够让你做一切事情，包括擦除 Bootloader 而不会报废，另外你应该(几乎是必须)拥有该开发板的所有软硬件资料，其中包含 Bootloader 和 Linux 内核的源码，电路图，芯片文档。这保证你不会形而上学的在那猜测，然后似是而非的理解。这也让你有勇气指出那些前人写过的经典之作中也不免错误或者翻译不合理的存在。为什么不用 PC 或者虚拟机呢？大多数时候是可以的，但有以下几点需要注意：对于 PC 你很有可能需要多台电脑；你可能无法得到 PC 的硬件资料；由于在分析代码时可能需要不停重启，PC 重启时间过长；PC 只能让你研究 x86 等。对于虚拟机：如果要研究一些底层的原理，虚拟机不是硬件，你可能无法获取详细的配置信息；虚拟机可能让 CPU 工作在不同寻常的模式。

2. 选择你喜欢的或者工作需要的内核版本，笔者绝不推荐你选择 0.1 版本，也不推荐 2.4 版本，你应该选择 Linux2.6.11 或者更高的 2.6 版本，这基于以下理由：如果你有足够的能力让 0.1 版本在你的开发板上跑起来，你对 Linux 内核的理解似乎不是一般境界了，你让它在

PC 上跑起来都相当困难。2.4 版本已经是 10 年前的产物，这方面的书籍虽然也有很多，知名的有《Linux 内核源代码情景分析》，但是你在论坛上询问 2.4 版本的内核问题，多数时候得到的回应是沉默。最后一个理由：2.6 版本在 2.4 版本上有了天翻地覆的变化，比如强化了面向对象，统一的设备模型，名字空间的引入等等。这些功能举足轻重。为何不选择最新的 3.0 版本？本质上 3.0 在代码和功能上和高版本的 2.6 内核没有太大的变化，另外根据 3.0 编写的书籍应该还没有出现。Linux.2.6.24 或者以上的版本是一个推荐的选择，因为它们是笔者在工作中遇到的最多的内核版本，我想 BSP 提供商不会选择一个比较糟糕的内核版本作为基础来开发。

3. 从哪里下手？你这么快就准备下手了？好吧，了解内核代码的目录存放，了解内核的 Makefile 编写方式，对它们的理解均会在研究中不断深化。然后呢，你可以看看内核启动都经历了哪些阶段：Bootloader 引导阶段，解压缩阶段等等。什么时候写代码啊？如果这个时候你还不是迫不及待，那么你的热情还不是太足够。尽管内核提供了 printk 帮助打印和追踪，但是笔者还是推荐你封装一下 printk 或 vprintk，形成自己的一系列 DEBUG 函数：比如打印行号函数名等，以及添加一些醒目的颜色，放在自己的单独.c 文件中，然后把它编译到内核，另外定义一个.h 文件放在 include/linux 下并声明它们。在你当前研究的内核文件中添加该头文件，然后……

从哪个功能模块研究内核？这个问题有些仁者见仁，智者见智了。通常有两种方式：研究自己喜欢或者工作需要的特定部分；从 Bootloader 启动开始一直到 start_kernel 调用结束。本质上这是两种不同的方法，对个人的知识水平要求也有很大的不同。

第一种方法因为针对自己关注的功能模块，所以必然已经对它有所了解，所以切入进去相对简单，并且效果明显，可以迅速提升业务能力。这种方式也是在第二阶段中偶尔去研究内核功能的一种可行方法。但是这种方法有一个明显的不足：由于内核的各类功能模块并非相互独立，所以如果这个模块在某些时候引用了另一个模块的函数，对它的理解很可能陷入似是而非的假象阶段，如果你尝试不停的往下挖掘，很快你就像堆栈一样溢出了。

第二种方法也有很多人采用，但是这种方法确实不太适合新手，至少新手应该通过第一种方法研究了一些内核相对简单的实现，比如 Proc 文件系统，设备驱动模型等。之所以不太适合新手，是因为对 Bootloader 的知识可能欠缺，另外就是对内核镜像文件的构成要有所深入。当然不仅如此：对应开发板的汇编语言的掌握，对应开发板的硬件架构的了解。它的缺点是很显然的，理解周期长，一开始要分析很多汇编代码，并且它们是与硬件息息相关的。当然它的优点也毋庸置疑：整体的把握，会将大多数第一种方法的似是而非给解决掉，并且这种掌握是系统的。这有点像空中俯瞰迷宫。

几乎没有人能够完全使用第一种或者第二种方法来掌握 Linux 的大部分实现的原理(只能说是大部分，在 Linux 内核某些阴暗的角落很久都不会有人涉及)。对 Linux 的理解总是循着“循环往复，总体向上”的路线发展。对这两种方法应该合理运用，首先采用第一种方法，在恰当的时候切入第二种，这应该是比较好的路线图。

由于笔者还在路上，这里不能给出一个理想的功能模块分析的顺序，但是参考经典书籍的章节分布可能是一个不错的选择。

最后的啰嗦：内核中通常不会出现过于庞大的函数，如果是，那么基本是在做一些平行处理。你可以容易找到 switch 语句，如果不幸没有，那么你一定要细心找到其中的关键函数，一般如果不是在初始化或者注销阶段，应该不会包含太多的关键函数，然后再对关键函数一一击破，击破的过程中要尝试记录关键数据结构的状态变化。总结成一句话就是以关键函数和关键数据结构为核心。内核中最让人易于产生迷惑的地方：对位的操作，不经过一段时间你很难弄明白它们到底对应程序的哪些功能或者状态，好在当前的内核在努力避免直接对它们的操作，而是封装在一系列的宏中。内核中对很多操作总结提炼成了一类模式的操作：锁，位，链表等等，这是面向对象的一种努力，这也是研究过程中的另外一种收获，注意对它们的总结和积累。另外——还是看代码(笔者认为它包含内核文档)吧！

基于对与时俱进方法论的实践，该章节有可能被补充……

关于书

在学习 Linux 内核代码的过程中，定会参考很多书籍以及网路资源，但是并不是所有的书籍和资源都能够帮助你前进，或者说是能够将你引导向正确而高效的道路。

在学习的一些阶段可能会需要不同的书，比如在第一个阶段，一些基本的书籍都是可以参考的。这个阶段的书籍非常多，比如《鸟哥的 linux 私房菜》，它介绍了 Linux 系统的基本概念和使用。《高级 Bash 脚本编程指南》可以说是学习 Shell 的利器，翔实的示例和解释足以让你成为 Shell 专家。另外可以在 Linux 平台架设一些常用的服务器等以增加实践。

由于大部分人在大学期间都学习过 C 语言，所以在学习的第二阶段依然可以把那本经典的谭浩强读本放在手边作为参考手册。《Linux C 编程一站式学习》这本书在介绍标准 C 的同时则倾向于 Linux 平台的编程：网络编程，ELF 格式解析，链接的原理和 Makefile。此时如果不提大名鼎鼎的《UNIX 环境高级编程》和《UNIX 网络编程》这两本书，一定会遭到抗议。尽管书名包含 UNIX，但是由于 Linux 是类 UNIX 的系统，所以这两本书同样适用于 Linux 环境。

《UNIX 环境高级编程》被誉为 UNIX 编程“圣经”，并且一版再版。它的首席作者 W.Richard Stevens 是国际知名的 UNIX 和网络专家；受人尊敬的计算机图书作家；同时他还是广受欢迎的教师和顾问。Stevens 先生不幸逝于 1999 年 9 月 1 日。UNIX 操作系统的原作者 Dennis Ritchie 对该书的评价是“公认的优秀、匠心独具的名著”。再多的赞誉都不为过，因为除了参考系统函数或者 glibc 库函数的文档或者直接查看源码能够获取比该书更全面的信息外，似乎别无他法。如果你能够在某些网站或者书籍获取到比该书更详尽的描述和分析，那么把它作为参考一定不会错。非常喜欢这本书首页上的“Standing on Shoulders of Giants”——站在巨人的肩上。

《UNIX 网络编程》同为 W.Richard Stevens 的另一部名著，它弥补了《UNIX 环境高级编程》在网络编程方面叙述的不足，与该书享有同样的地位。该书的最新版本由世界著名网络专家再次修订，添加了 IPv6，SCTP 以及密匙管理相关的内容。所以值得庆幸：这两本书都没有因为 Stevens 先生的离去而失去时代的活力。

杰出人物的成就总是在让人景仰的同时，令人不由自主的惊叹。《TCP/IP 详解》（三卷本）作为了解当前盛行世界的因特网原理的圣经级著作，也同样出自其手。“Richard Stevens 以他的写作风格和作品，在 TCP/IP 编程领域竖立起一座丰碑，令其他作者高山仰止，心向往

之”。

在这里一气儿介绍了 Stevens 先生的这三本书，应该没有人会非议。如果你在一些 Linux 编程或着网络方面的资深著作中的参考书目中没有找到这三本书的名字，那确实有点不可思议。

在参考以上书籍，并在个人努力和工作经验积累的情况下，到达第三个阶段是水到渠成的事。这似乎是一个关键的时刻：要么继续留在原地，要么进入一个完全不同的领域。关于探索 Linux 内核代码的书籍这些年来已经相当繁多。如果粗略的对它们进行分类，大概如下：

一些书籍可能会大量的引用代码，但是不对该功能模块的原理和细节进行详述，最多在源码中给出一些翻译后的注释。这类书籍适合做参考，却不适合用它来系统的学习内核原理。否则很快就将进入“一鼓作气，再而衰”的境地，因为越来越多的疑问将耗尽你的激情，最后除了头脑混乱外，很可能得出学习内核太难的结论。这里对这类书做了一定的批判，但是它们依然不乏作为参考资料的价值。在最近一段时间的相关出版物中，这类书籍已经越来越少，而一批质量较高的国内书籍开始呈现出来。

有些书籍单独针对内核特定的区域或者说子系统来讲解，比如内核驱动，网络实现等。这类书籍针对性较强，对细节比较注重，能够对一些问题分析的较为透彻，另外一个优点就是这类书往往比较薄，让人容易接受。当然了这也就意味着它不能给你在另一个内核功能领域以有效的指导。这类书的典型代表有：《深入理解 LINUX 网络内幕》，《深入理解 Linux 虚拟内存管理》，《Linux 设备驱动程序》和《Embedded Linux Primer》。更确切的说后两本并不是讲解内核原理的，它们针对的是内核的应用，所以在选择时应该加以区分，但是《Essential Linux Device Drivers》这本书有取代《Linux 设备驱动程序》之势，它补充了驱动相关的内核原理部分。

还有一类书，这类书不太好对它进行分类，它们对深入理解底层硬件相关部分的基本原理帮助颇大，但毕竟描述的内核版本和当前的 2.6 内核版本相去甚远，比如对 0.1 版本的分析，如果单从这些资源来学习当前的内核版本似乎有点“舍近求远”。它们的真正价值在于分析内核的角度，也即它们没有随波逐流的去跟随已有的著作，而是独辟蹊径，从另外一个侧面揭示 Linux 的远古面纱。这种书籍的代表有《Linux 内核完全剖析》和《Linux 内核设计的艺术》，从后者的内容可以看出它应该能代表当前国内在 Linux 领域的部分研究水平。

好了，如果在这一领域没有一些重量级的选手出场一定说不过去。《深入理解 Linux 内核》和《Linux 内核设计与实现》，这两本书分别被简写为 ULK 和 LKD，它们应该算是一个重量级的。并且论述的方式非常像操作系统原理，但是是结合 Linux 的系统原理。把它们作为教材是非常之好。ULK 可以说就是 Linux 版的系统原理，从行文风格，知识点的切入方式基本是属于学术派：首先是提纲挈领的简介，然后是细分的要点详述，进而引述少量的代码。LKD 与 ULK 非常相似，但是很明显的感觉到它对代码的引用要多，实践性更强，比如它会谈到内核的代码结构，编译，调试，glibc 库以及 Linux 的代码风格等，属于实践派，但又不完全，因为它比实践的书籍多了理论的介绍，但是代码引用的程度又太少，如果完全跟随它来通过开发板实践相关的功能子系统，可以说能够摸着石头过河的时候非常少，大部分的关键代码都无法从书中找到蛛丝马迹。另外由于它页数的限制，很多内核功能的细节都被忽略掉了，虽然第三版的英文版已经发行，除了最后一个章节的内容有大的改变外，基本延续了前版叙

述的知识范围。所以它渐渐有被后来者《深入 Linux 内核架构》LKA 代替的危险。

书籍名称	作者	出版时间	内核版本
Understanding the Linux Kernel, 3rd Edition(ULK)	Daniel P. Bovet, Marco Cesati	November 2005	2.6.11
Linux Kernel Development, 3rd Edition(LKD)	Robert Love	June 2010	2.6.34
Professional Linux Kernel Architecture(LKA)	Wolfgang Mauerer	January 2008	2.6.24

Linux 内核经典书籍

ULK 基本是属于学术派的著作，两位作者都是学府内的教授，他们自然在有意无意中会更专注操作系统的原理和算法，而或多或少限制对代码实现的论述。所以它属于通用操作系统原理著作但又向 Linux 实现靠近的理论论述的一种过渡，由于 ULK 将所有篇幅放在了 Linux 对操作系统的实现上，所以它在理论层面的论述可以说鲜有挑战者。总体来看 ULK 和 LKD 之间的关系，很容易发现 LDK 的出现是用来弥补 ULK 在论述实践这一块的缺失，但是不知为什么 LKD 对实践的论述又不彻底，反而造成了一种有点儿尴尬的骑墙势。

不妨做如下的推理假设：如果在多年前，那时候内核代码的维护还基本在一个有限的圈子里，这些人都是计算机黑客或者是内核代码的狂热爱好者，他们完全有能力维护好各自负责的代码部分。但是随着功能的增加，代码量也与日俱增，而代码量的增加会以非线性的方式影响代码的复杂度，这导致对代码的把握非常不易，维护起来的难度可想而知。直至今日，内核代码的维护人员有点青黄不接。所以合理的猜测是：LKD 最初出现时是为了给内核开发的新手做一个入门式的引导，因此它花了几章的篇幅在介绍 Linux 代码结构，编译，调试以及代码风格上。

LKA 是由德国 Wolfgang Mauerer 完成的。所以它的原版是德文版，而英文版的出现已经到了 2008 年。这本书在国内推出的时间则推迟到了 2010 年的后半年。尽管笔者只看了这本书的若干章，但可以毫不扩张的说，它是在 Linux 实践领域进行相对全面的论述的集大成者。看一本书对知识点的把握深度和细度，只要看它对 Linux 内核中最复杂的内存管理机制的论述就可以了，比如它对内存区类型 ZONE_MOVABLE 的作用的叙述，在大多数的类似书籍中都难以找到对它的详细描述。另外一点值得肯定的是，它在内核代码的应用中有非常好的连贯性，使用开发板跟随它的论述进行实践验证是可行的，摸着石头过河是现实的了。由于深度细度以及代码连惯性两方面的要求，尽管这本书只有十九章，中文版的页数已经达到了一千多页。总结：这本书是用来研究内核的，它并不太注重对内核的使用，而大多数人可能是为了学习内核驱动的开发，这个 LDD3 或者 ELDD 就可以了，使用 LKA 反而是南辕北辙。但是如果你想洞悉 Linux 对操作系统的实现，那么选择它基本上没错了。

尽管 LKA 比 LDA 具有很大的优势，它也不是完美无瑕，一些地方可能翻译得不够准确，比如内存管理中对页和页帧的使用，有些混乱。另外一些句子可能翻译的不够恰当，比如 P23 页的“对于后者来说有大量的调试器可用，而对于后者来说”改为“而相对于后者来说”更为贴切。另外原版作者可能主要在 x86 上进行实验和研究，所以有些地方可能适用于 x86 架构，但并不适合所有的平台。例如 P111 中的“node_start_pfn 在 UMA 系统中总是 0”，诚然 x86 的 CPU 对物理 RAM 的地址从 0 开始分配，但是 ARM 的 CPU 就不尽然，它通常都不是从

0 开始。P137 中“减去 0Xc0000000，则可以获得对应的物理地址”，但在 ARM 上应该还要加上一个物理地址的偏移。尽管如此，瑕不掩瑜。

未知何朝何代，武林中出现了名震天下的两大流派：剑宗和气宗。它们吸取了中原各大门派武术之精华，适时有两少年各入两宗，相约十载一战。

两人各拜其师，砥砺寒暑，未敢懈怠。光阴荏苒，白驹过隙；十载春秋，弹指一挥。是日，会稽山下，只见一人仙风荡漾，一人剑气浩然。日出而战，披星而息。剑出。

廿十载，气完胜。又十载，日出而战，而三日不能息！何也？气中有见，剑中有气，阴阳相合者也。

在经过了一番“讨价还价”之后，似乎找到了学习内核原理的最佳组合了：理论与实现，ULK 和 LKA，似乎把它们缩写成 ULKA 更好，如此则阴阳相合，关于书也就说到这了！

最后的啰嗦：如何看书？是一气看完？还是分章节，一一攻破呢？显然是一一攻破要好的多，所有章节看下来的周期太长，难于把握。实际上还要根据难易程度来决定，有些章节需要非常细心的研读。另外要注意实践和理论一定要结合，比较简单的功能实现，可以先实践，也即先循着 LKA 讲的顺序看看代码，然后自我分析，接着结合 ULK 的理论，从全局把握；比较复杂的功能模块，则可以先看 ULK 理解基本的系统原理，然后再结合 LKA 进行实践。本质上需要把 90%的时间花在这两本书和在开发板的实践上，只有通过它们还不能完全理解的情况下，再去参考其他的书籍和资源，这可以保证学习的效率。

关于人

如果有好的方法，好的环境，就能让一个人取得他想要的“成功”，这世界几乎不需要“失败”这个词了。提到这个就想起来以前新东方的老罗，老罗讲过他在北京学习英语的一段趣事：一段时间要看一本励志书，然后能再坚持一段时间，不然就会被京城里的滚滚红尘给召唤去了。这说明了什么问题？人，总是想进步，但又惰性难改，难于坚持。

人之天性，既有积极的一面：求知欲和好奇心，促使人不停探索和学习；又有消极的一面：懈怠和懒惰，总是在人前进的道路上消磨意志，动摇决心。好的方法和环境在某种程度上可以放缓这种过程，比如每个阶段都能看到成果，形成良性循环；好的团队气氛，研究氛围也可以提高效率，互相鼓励，共同进步。但是在强调这些外在因素的同时，更要坚定自己的决心——这确实不易。如果你真的觉得值得把自己的一段时间用来奉献给 Linux 内核，你下定了决定，那么把它养成一个习惯吧。

“行为心理学研究表明：21 天以上的重复会形成习惯；90 天的重复会形成稳定的习惯。即同一个动作，重复 21 天就会变成习惯性的动作；同样道理，任何一个想法，重复 21 天，或者重复验证 21 次，就会变成习惯性想法。所以，一个观念如果被别人或者自己验证了 21 次以上，它一定已经变成了你的信念。”一旦养成一个习惯，那么习惯的力量将会在无形中持续，此时的坚持已经成为了自然而然，理所当然。

最初的阶段的特征是“刻意，不自然”，这是最艰难的阶段。你需要十分刻意提醒自己改变，而你也会觉得有些不自然，不舒服，甚至是痛苦。首先环境的对比会非常强烈的影响你，比

如你要花时间做一些明显不符合你平常这个时间做的事情，比如上网聊天，打游戏，或者打球，逛街，并且恰恰相反，你在做令你头昏脑胀的事情，有时还相当挫败：即便一些简单的 Linux 内核实现机制也不能保证就可以一蹴而就，一些地方很可能让你迷惑。此时你要平心静气，告诉自己“慢即是快”，当前不能解决的，先暂且放置，或者把问题发在相关的论坛，相信大家都是热心的。本来对内核的把握就是“循坏往复，总体向上”的。心态此时就起了非常大的作用。有得到必有所失去，既然选择了这件事，就要提醒自己，不要泄气。诚如老罗所说的那些所谓“精神鸦片”的成功学著作，有时候可能要看一看；笔者推荐一种方法就是将你觉得在这个领域很牛 x，你很钦佩和敬仰的人的照片放在的面前书桌前，时刻勉励自己。

有一点要提到，笔者在网上购书的时候也会看看书评，发现一个有趣的现象：无论这本书有多经典，总是会有几条评论会否定它的价值。其实应该根据自己当前所处的阶段来选择书籍，否则很可能会揠苗助长。有些国内翻译的书籍虽然颇多瑕疵，但是大部分都是可以接受的，可以对照相关的勘误表和英文原版来分析一下，是书的问题，是翻译的问题，还是自己理解的问题。千万不要，读了几页没有看懂，就断定这书很垃圾，这不但是对原版作者辛勤工作的极大不尊重，同时这很可能让你错过一本能够帮助你的好书

一旦度过最初的一个阶段，就会形成原始的积累，你会发现以前看起来一团乱麻，毫无头绪的代码并非无章可循，反而是井井有条，到处是闪光的地方：从简单的方面来讲：内核链表，哈希表，它们都可以被提取出来为我所用；从复杂方面讲：内核命名空间，它足以颠覆传统虚拟机的观念。内核代码变得相当有条理，单个子功能的代码都集中在一个或者几个.c 文件中，一个子系统基本集中在一个文件夹中。

内核是开放的，所以研究它也要保持一颗开放的心。不光要看书，或者添加简单的 DEBUG 代码，要在整个过程中，充满疑问和想法，然后验证它，比如一系列的 Kobject 是如何组织的，为什么？那么你就要专门针对它写一些相关的解析函数，以加深对它的理解。更深层次的要求是，你在这个机制上可以增加一些自己的东西，同理也可以减少一些，观察并分析内核的运行状况是否和预想的一样，其实内核代码中最原始的东西很少，基本都是在后来的过程中一行一行增加的，所以通过一些内核变更日志网站，比如 lkml，也很有可能追踪到你所迷惑的代码的根源。

一颗谦逊的心，如果是刚刚涉足这一领域，那么一颗谦逊的心可以促进你与他人交流，看清自己的水平，就像一个空杯子，它才能装更多的水。也许在这个领域已经很久了，对 Linux 内核可以如数家珍，那么不妨在网络论坛上多发表一些自己的高见，帮助他人，毕竟所有的知识都是“Standing on Shoulders of Giants”——站在巨人的肩上得来的。另外也可能精通的是某个特定的 CPU 架构，不妨看看其他的；每天更新的内核代码就有几百条，也不妨关注等等，所有这些都应成为达到更高点的动力——高山仰止，景行行止，虽不能至，心向往之。

“慢即是快”——笔者再次强调，笔者就是一个实际的例子，比如今天要分析完这一段，那么很可能你分析不来这一段，特别是刚刚开始的时候，要做好思想准备。另外，有些部分比如 Bootloader 加载到 start_kernel 阶段，你可能用了一个月，分析的很透彻了，就准备也用一个月来分析内存管理，这是完全不行，很可能需要半年时间，所以张弛有度，切忌贪多恋快。

啰嗦的太多了，写下这些，希望能给有志于探索 Linux 内核奥秘的人有所帮助，所有的观点都是笔者的愚见，因为笔者还在探索的路上，愿与所有相同志趣的人共勉：“慢即是快”；实践与理论相结合，书不在多，在于研读；在正确的方法和方向上坚持再坚持。

Netwalker

2011-11