Linux内核源代码查看及分析方法

Github资料整理员
只求星，https://github.com/0voice

Linux内核源代码查看及分析方法

1. Linux 内核源代码查看

Linux的内核源代码可以从很多途径得到。一般来讲，在安装的linux系统下，/usr/src/linux目录下的东西就是内核源代码。

对于源代码的阅读，要想比较顺利，事先最好对源代码的知识背景有一定的了解。对于linux内核源代码来讲，我认为，基本要求是:1、操作系统的基本知识;2、对C语言比较熟悉，最好要有汇编语言的知识和GNU C对标准C的扩展的知识的了解。另外在阅读之前，还应该知道Linux内核源代码的整体分布情况。我们知道现代的操作系统一般由进程管理、内存管理、文件系统、驱动程序、网络等组成。看一下Linux内核源代码就可看出，各个目录大致对应了这些方面。Linux内核源代码的组成如下(假设相对于linux目录):

arch

这个子目录包含了此核心源代码所支持的硬件体系结构相关的核心代码。如对于X86平台就是i386。

include

这个目录包括了核心的大多数include文件。另外对于每种支持的体系结构分别有一个子目录。

init

此目录包含核心启动代码。

mm

此目录包含了所有的内存管理代码。与具体硬件体系结构相关的内存管理代码位于arch/*/mm目录下，如对应于X86的就是arch/i386/mm/fault.c 。

drivers

系统中所有的设备驱动都位于此目录中。它又进一步划分成几类设备驱动，每一种也有对应的子目录，如声卡的驱动对应于drivers/sound。

Ipc

此目录包含了核心的进程间通讯代码。

modules

此目录包含已建好可动态加载的模块。

fs Linux

支持的文件系统代码。不同的文件系统有不同的子目录对应，如ext2文件系统对应的就是ext2子目录。

Kernel

主要核心代码。同时与处理器结构相关代码都放在arch/*/kernel目录下。

Net

核心的网络部分代码。里面的每个子目录对应于网络的一个方面。

Lib

此目录包含了核心的库代码。与处理器结构相关库代码被放在arch/*/lib/目录下。

Scripts

此目录包含用于配置核心的脚本文件。

Documentation

此目录是一些文档，起参考作用。

2. Linux 内核源码分析方法

2.1 内核源码之明晰

如果想透析Linux，深入操作系统的本质，阅读内核源码是最有效的途径。我们都知道，想成为优秀的程序员，需要大量的实践和代码的编写。编程固然重要，但是往往只编程的人很容易把自己局限在自己的知识领域内。如果要扩展自己知识的广度，我们需要多接触其他人编写的代码，尤其是水平比我们更高的人编写的代码。通过这种途径，我们可以跳出自己知识圈的束缚，进入他人的知识圈，了解更多甚至我们一般短期内无法了解到的信息。Linux内核由无数开源社区的“大神们”精心维护，这些人都可以称得上一顶一的代码高手。透过阅读Linux内核代码的方式，我们学习到的不光是内核相关的知识，在我看来更具价值的是学习和体会它们的编程技巧以及对计算机的理解。

我也是通过一个项目接触了Linux内核源码的分析，从源码的分析工作中，我受益颇多。除了获取相关的内核知识外，也改变了我对内核代码的过往认知：

1．内核源码的分析并非“高不可攀”。内核源码分析的难度不在于源码本身，而在于如何使用更合适的分析代码的方式和手段。内核的庞大致使我们不能按照分析一般的demo程序那样从主函数开始按部就班的分析，我们需要一种从中间介入的手段对内核源码“各个击破”。这种“按需索取”的方式使得我们可以把握源码的主线，而非过度纠结于具体的细节。

2．内核的设计是优美的。内核的地位的特殊性决定着内核的执行效率必须足够高才可以响应目前计算机应用的实时性要求，为此Linux内核使用C语言和汇编的混合编程。但是我们都知道软件执行效率和软件的可维护性很多情况下是背道而驰的。如何在保证内核高效的前提下提高内核的可维护性，这需要依赖于内核中那些“优美”的设计。

3．神奇的编程技巧。在一般的应用软件设计领域，编码的地位可能不被过度的重视，因为开发者更注重软件的良好设计，而编码仅仅是实现手段问题——就像拿斧子劈柴一样，不用太多的思考。但是这在内核中并不成立，好的编码设计带来的不光是可维护性的提高，甚至是代码性能的提升。

每个人对内核的了理解都会有所不同，随着我们对内核理解的不断加深，对其设计和实现的思想会有更多的思考和体会。因此本文更期望于引导更多徘徊在Linux内核大门之外的人进入Linux的世界，去亲自体会内核的神奇与伟大。而我也并非内核源码方面的专家，这么做也只是希望分享我自己的分析源码的经验和心得，为那些需要的人提供参考和帮助，说的“冠冕堂皇”一点，也算是为计算机这个行业，尤其是在操作系统内核方面贡献自己的一份绵薄之力。闲话少叙（已经罗嗦了很多了，囧~），下面我就来分享一下自己的Linix内核源码分析方法。

2.2 内核源码它到底难不难

从本质上讲，分析Linux内核代码和看别人的代码没有什么两样，因为摆在你面前的一般都不是你自己写出来的代码。我们先举一个简单的例子，一个陌生人随便给你一个程序，并要你看完源码后讲解一下程序的功能的设计，我想很多自我感觉编程能力还可以的人肯定觉得这没什么，只要我耐心的把他的代码从头到尾看完，肯定能找到答案，并且事实确实是如此。那么现在换一个假设，如果这个人是Linus，给你的就是Linux内核的一个模块的代码，你还会觉得依然那么轻松吗？不少人可能会有所犹豫。同样是陌生人（Linus要是认识你的话当然不算，呵呵~）给你的代码，为什么给我们的感觉大相径庭呢？我觉得有以下原因：

1．Linux内核代码在“外界”看来多少有些神秘感，而且它很庞大，猛地摆在面前可能感觉无法下手。比如可能来源于一个很细小的原因——找不到main函数。对于简单的demo程序，我们可以从头至尾的分析代码的含义，但是分析内核代码这招就彻底失效了，因为没有人能把Linux代码从头到尾看上一遍（因为确实没有必要，用到时看就可以了）。

2．不少人也接触过大型软件的代码，但多数属于应用型项目，代码的形式和含义都和自己常接触的业务逻辑相关。而内核代码不同，它处理的信息多数和计算机底层密切相关。比如操作系统、编译器、汇编、体系结构等相关的知识的欠缺，也会让阅读内核代码障碍重重。

3．分析内核代码的方法不够合理。面对大量的并且复杂的内核代码，如果不从全局的角度入手，很容易陷入代码细节的泥淖中。内核代码虽然庞大，但是它也有它的设计原则和架构，否则维护它对任何人来说都是一个噩梦！如果我们理清代码模块的整体设计思路，再去分析代码的实现，可能分析源码就是一件轻松快乐的事情了。

针对这些问题，我个人是这样理解的。如果没有接触过大型软件项目，可能分析Linux内核代码是一个很好的积累大型项目经验的机会（确实，Linux代码是我目前接触到的最大的项目了！）。如果你对计算机底层了解的不够透彻，那么我们可以选择边分析边学习的方式去积累底层的知识。可能刚开始分析代码的进度会稍显迟缓，但是随着知识的不断积累，我们对Linux内核的“业务逻辑”会逐渐明朗起来。最后一点，如何从全局的角度把握分析的源码，这也是我想与大家分享的经验。

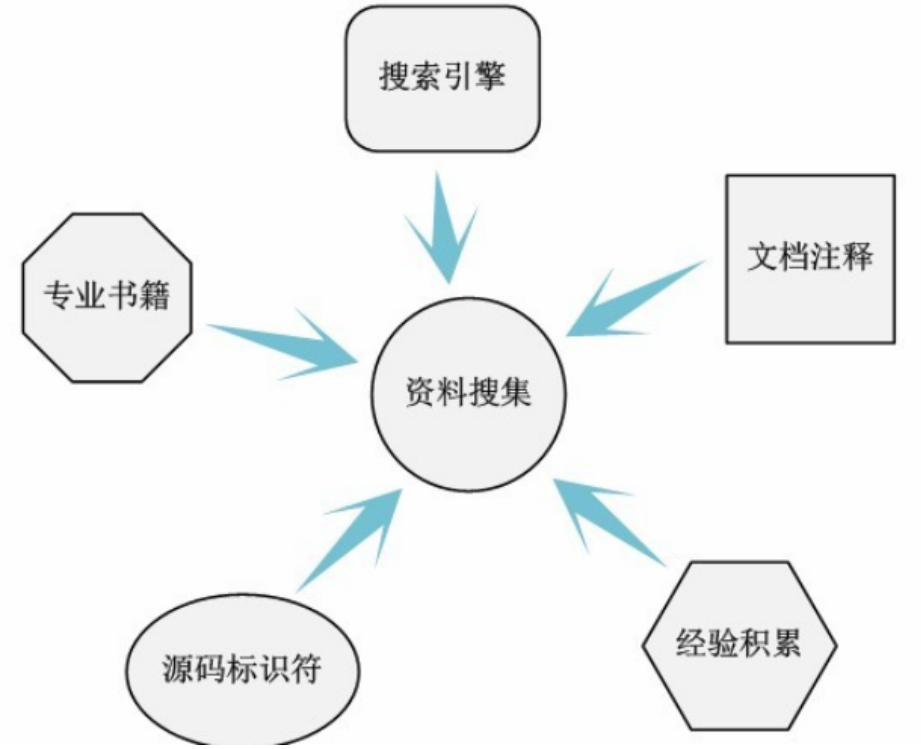
2.3 内核源码分析方法

2.3.1 资料搜集

从人认识新事物的角度来讲，在探索事物本质之前，必须有一个了解新鲜事物的过程，这个过程是的我们对新鲜事物产生一个初步的概念。比如我们想学习钢琴，那么我们需要先了解弹奏钢琴需要我们学习基本的乐理、简谱、五线谱等基础知识，然后学习钢琴弹奏的技巧和指法，最后才能真正的开始练习钢琴。

分析内核代码也是如此，首先我们需要定位要分析的代码涉及的内容。是进程同步和调度的代码，是内存管理的代码，还是设备管理的代码，还是系统启动的代码等等。内核的庞大决定着我们不能一次性将内核代码全部分析完成，因此我们需要给自己一个合理的分工。正如算法设计告诉我们的，要解决一个大问题，首先要解决它所涉及的子问题。

定位好要分析的代码范围，我们就可以动用手头的一切资源，尽可能的全面了解该部分代码的整体结构和大致功能。



这里所说的一切资源是指无论是Baidu、Google大型网络搜索引擎，还是操作系统原理教材和专业书籍，亦或是他人提供的经验和资料，甚至是Linux源码提供的文档、注释和源码标识符的名称（不要小看代码中的标识符的命名，有时它们能提供关键的信息）。总之这里的一切资源指的就是你能想到的一切可用资源。当然，我们不太可能通过这种形式的信息搜集获得所有的我们想要的信息，我们只求尽可能全面即可。因为信息搜集的越全面，之后分析代码的过程能使用的信息就更多，分析过程的困难就会越小。

这里举一个简单的例子，假定我们要分析Linux的变频机制实现的代码。目前为止我们仅仅是知道这个名词而已，透过字面含义我们可以大致猜测它应该和CPU的频率调节相关。通过信息搜集，我们应该能得到如下的相关的信息：

1．CPUFreq机制。

2．performance、powersave、userspace、ondemand、conservative调频策略。

3．/driver/cpufreq/。

4．/documention/cpufreq。

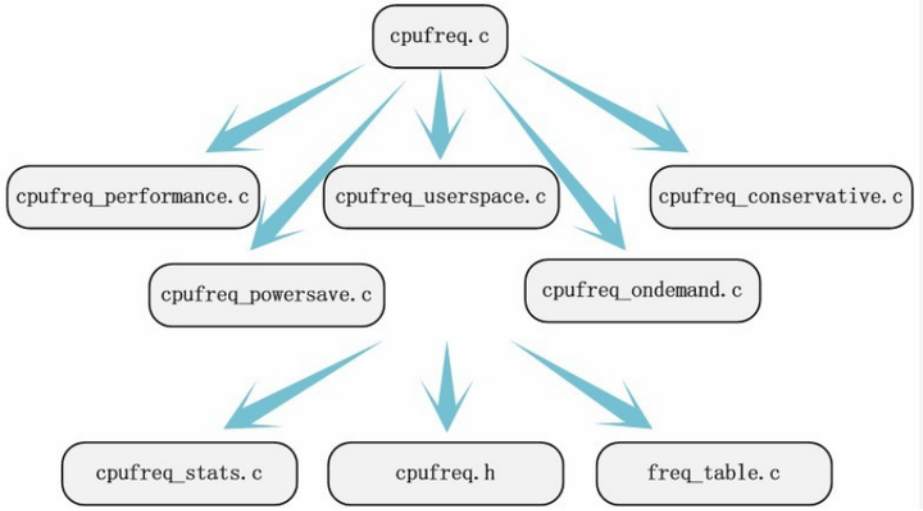
5．P state和C state。

分析Linux内核代码如果能搜集到这些信息，应该说是非常“幸运”了。毕竟有关Linux内核的资料确实不如.NET和JQuery那么丰富，不过这相比于十数年前，没有强大的搜索引擎，没有相关的研究资料的时期应该称得上是“大丰收”时代了！我们通过简单的“搜索”（可能会花费一到两天的时间吧），甚至找到了这部分代码所在的源码文件目录，不得不说这样的信息简直是“价值连城”！

2.3.2 源码定位

从资料搜集中，我们“有幸”找到了源码相关的源码目录。但是这并非意味着我们的确就是分析这个目录下的源代码。有时我们找到的目录有可能是分散的，也有时我们找到的目录下有很多和具体机器相关的代码，而我们更关心的是待分析代码的主要机制，而非与机器相关的特化代码（这样更有助于我们理解内核的本质）。因此，我们需要对资料中涉及代码文件的资料进行仔细甄选。当然，这一步也不太可能一次性完成，谁也不能保证一次就能选择出所有待分析的源码文件而且一个不漏。但是我们也不必担心，只要我们能抓住大多数模块相关的核心源文件，通过后期对代码的具体分析，就很自然的把它们全部找出来。

回到上述的例子中，我们认真的阅读/documention/cpufreq下的文档说明。目前的Linux源码会把模块相关的文档说明保存在源码目录的documention的文件夹下，如果待分析的模块没有文档说明，这多少会增加定位关键源码文件的难度，但是不会导致我们找不到我们要分析的源码。通过阅读文档说明，我们至少能关注到/driver/cpufreq/cpufreq.c这个源文件。通过这个对源文件的文档说明，结合之前搜罗到的调频策略，我们很容易关注到cpufreq_performance.c、cpufreq_powersave.c、cpufreq_userspace.c、cpufreq_ondemand、cpufreq_conservative.c这五个源文件。所有涉及的文件都找完了吗？不用担心，从它们开始分析，迟早能找到其他的源文件。如果在windows下使用sourceinsight阅读内核源码的话，我们通过函数的调用和查找符号引用等功能，结合代码的分析可以很方便的找到另外的文件freq_table.c、cpufreq_stats.c和/include/linux/cpufreq.h。



按照搜索出的信息流动方向，我们完全可以定位到需要分析的源码文件。源码定位这一步并非十分关键，因为我们不需要找出所有源码文件，我们可以把部分工作推迟到分析代码的过程中。源码定位也比较关键，找到一部分源码文件是分析源码的基础。

2.3.3 简单注释

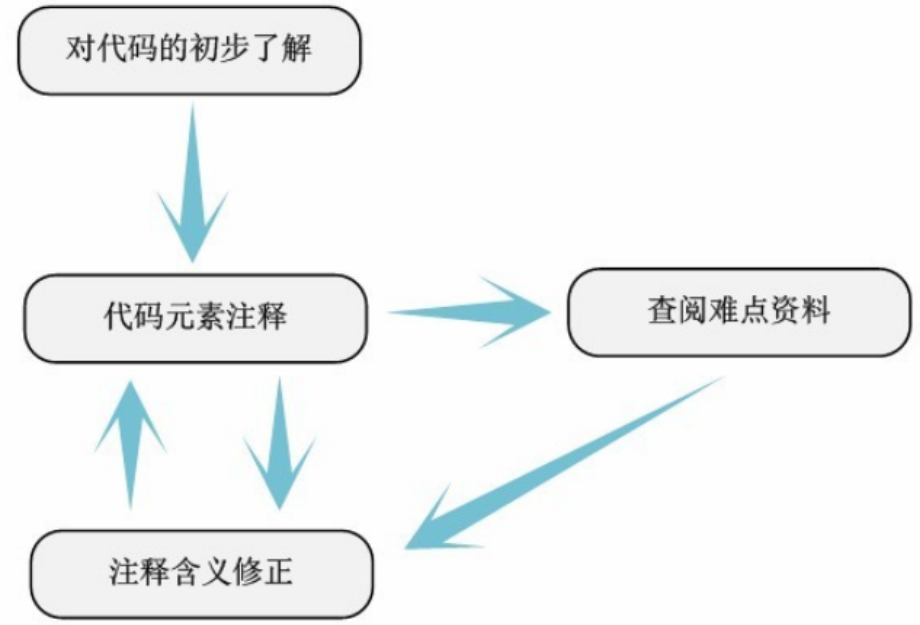
简单注释

在已定位好的源码文件中，分析每个变量、宏、函数、结构体等代码元素的大致含义和功能。之所以称此为简单注释，并非指这部分的注释工作很简单，而是指这部分的注释可以不必过分细化，只要大致描述出相关代码元素的含义即可。相反，这里的工作其实是整个分析流程中最困难的一步。因为这是第一次深入到内核代码的内部，尤其是对于首次分析内核源码的人来说，大量的生疏GNU的C语法和铺天盖地的宏定义会令人很绝望。此时只要沉下心来，弄清每个关键的难点，才能保证以后碰到类似的难点不会再被困住。而且，我们对内核相关的其他知识会不断的像树一样扩展开来。

比如在cpufreq.c文件开始就会出现“DEFINE_PER_CPU”宏的使用，我们通过查阅资料可以基本弄清这个宏的含义和功能。这里使用的手段和之前搜集资料使用的方法基本一致，另外我们也可以使用sourceinsight提供的转到定义等功能查看它的定义，或者使用LKML（Linux Kernel Mail List）查阅，实在不行我们还可以到stackoverflow.com提问寻求解答（想了解什么是LKML和stackoverflow？搜集资料吧！）。总之利用所有可能的手段，我们总能得到这个宏的含义——为每个CPU定义一个独立使用的变量。

我们也不要强求一次就能把注释描述的很准确（我们甚至都没必要弄清每个函数的具体实现流程，只要弄清大致功能含义即可），我们结合搜集到的资料和后边代码的分析不断的完善注释的含义

（源码中原有的注释和标识符命名在此很有利用价值）。通过不断的注释，不断的查阅资料，不断的修改注释的含义。



当我们把所有涉及的源码文件简单注释完毕后我们可以达到如下效果：

1．基本弄清了源码中代码元素存在的含义。

2．找出了该模块所涉及的基本上全部的关键源码文件。

结合之前搜集到的信息和资料对该待分析代码的整体或者架构描述，我们可以将分析的结果和资料对比，以确定和修正我们对代码的理解。这样，通过一遍的简单注释，我们就可以从整体上把握了源码模块的主要结构。这也达到了我们简单注释的基本目的。

2.3.4 详细注释

完成代码的简单注释后，可以认为对模块的分析工作完成了一半了，剩下的内容就是对代码的深入分析和彻底理解。简单注释总是不能将代码元素的具体含义描述的十分精确，因此详细注释是十分有必要的。这一步中，我们需要弄清以下内容：

1．变量定义在何时被使用。

2．宏定义的代码何时被使用。

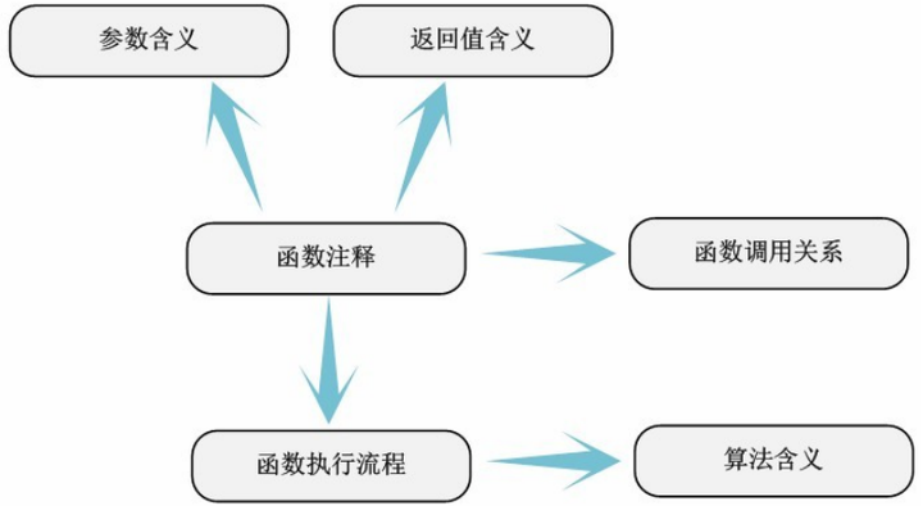
3．函数的参数和返回值的含义。

4．函数的执行流程和调用关系。

5．结构体字段的具体含义和使用条件。

我们甚至可以把这一步称为函数详细注释，因为函数之外的代码元素的含义基本上在简单注释中已经比较明确了。而函数本身的执行流程、算法等是这部分注释和分析的主要任务。

比如cpufreq_ondemand策略的实现算法（函数dbs_check_cpu中）是如何实现的。我们需要逐步分析该函数使用的变量和调用的函数等信息，弄清算法的来龙去脉。最好的结果，我们需要这些复杂函数的执行流程图和函数调用关系图，这是最直观的表达方式。

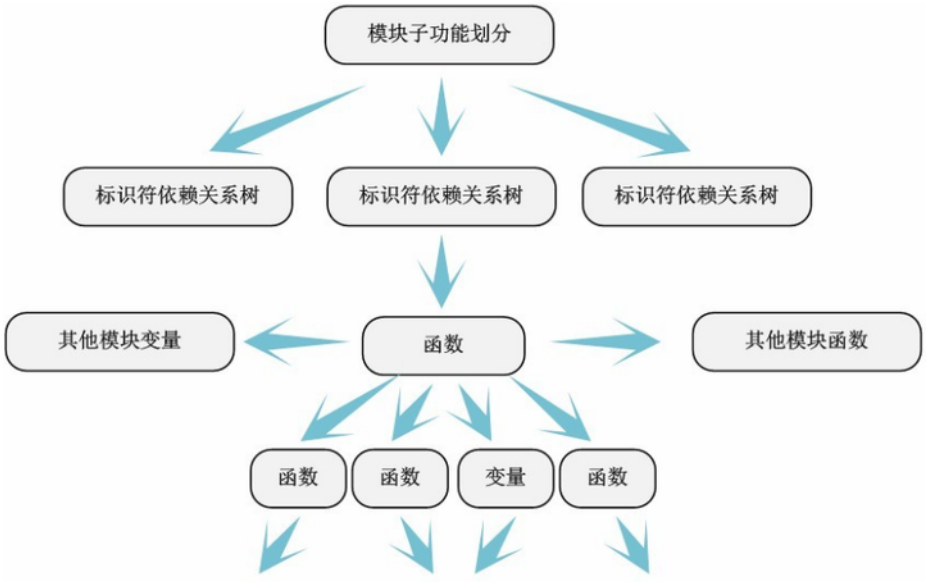


通过这一步的注释，我们基本上能完全把握待分析代码整体的实现机制了。而所有的分析工作可以认为完成了80%。这一步工作尤其关键，我们必须尽量让注释的信息足够的准确，才能更好的理解待分析代码的内部模块的划分。虽然Linux内核中使用了宏语
法“module_init”和“module_exit”声明模块文件，但是对模块内部子功能的划分是建立在充分了解模块的功能基础上的。只有正确划分好模块，我们才能弄清模块提供了哪些外部函数和变量（使用EXPORT_SYMBOL_GPL或者EXPORT_SYMBOL导出的符号）。才能继续下一步的模块内标识符依赖关系分析。

2.3.5 模块内部标识符依赖关系

通过第四步对代码模块的划分，我们就可以很“轻松”地逐个对模块进行分析。一般的，我们可以从文件底部的模块出入口函数开始（“module_init”和“module_exit”声明的函数，一般都在文件最后），根据它们调用的函数（自己定义的或者其他模块的函数）和使用的关键变量（本文件内的全局变量或者其他模块的外部变量）画出“函数-变量-函数”依赖关系图——我们称为标识符依赖关系图。

当然，模块内标识符依赖关系并非是单纯的树形结构，很多情况是错综复杂的网络关系。这时候，我们对代码的详细注释的作用就体现出来了。我们根据函数本身的含义，将模块进行子功能划分，抽取出每个子功能的标识符依赖树。

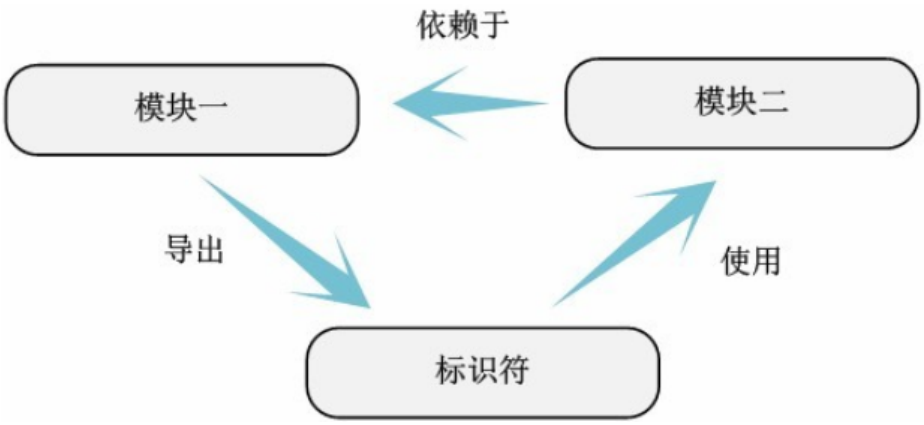


通过标识符依赖关系分析，可以很清晰的展示模块定义的函数调用了那些函数，使用了哪些变量，以及模块子功能之间的依赖关系——公用了哪些函数和变量等。

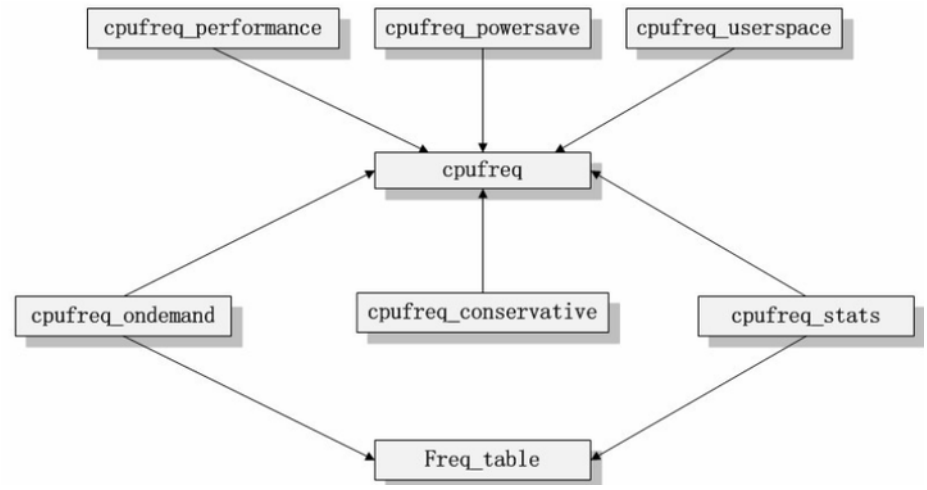
2.3.6 模块间相互依赖关系

模块间相互依赖关系

一旦将所有的模块内部标识符依赖关系图整理完毕，根据模块使用的其他模块的变量或函数，可以很容易得到模块之间的依赖关系。

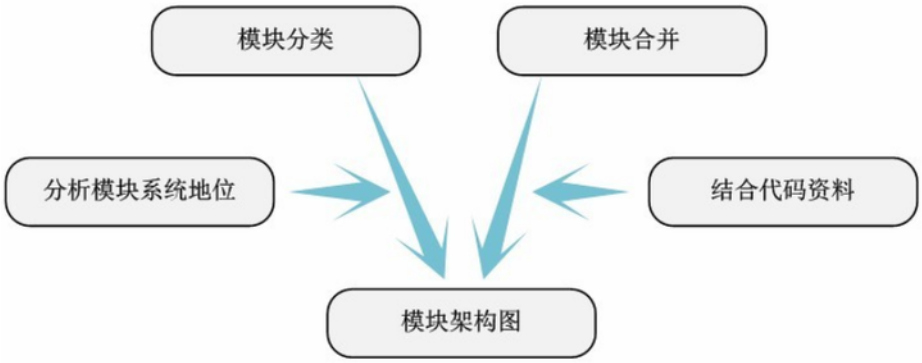


cpufreq代码的模块依赖关系可以表示为如下关系。

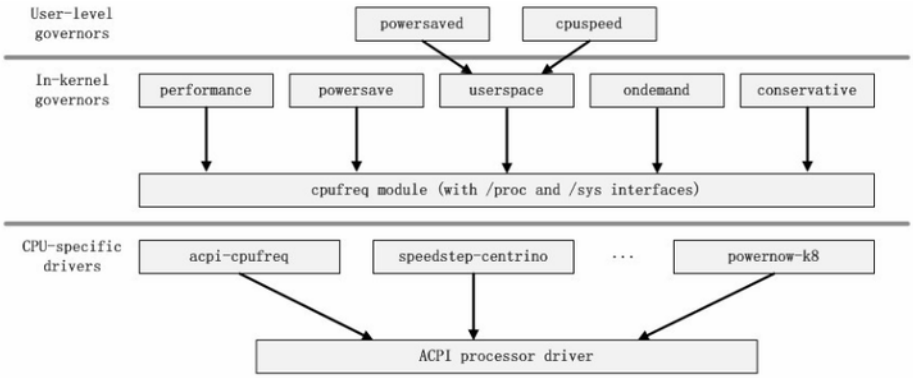


2.3.7 模块架构图

透过模块间的依赖关系图，可以很清楚的表达模块在整个待分析代码中的地位和功能。基于此，我们可以将模块分类，整理出代码的架构关系。



如cpufreq的模块依赖关系图所示，我们可以很清楚的看到所有的调频策略模块都是依赖于核心模块cpufreq、cpufreq_stats和freq_table的。如果我们把被依赖的三个模块抽象为代码的核心框架的话，这些调频策略模块都是建立在这个框架之上的，它们负责和用户层交互。而核心模块cpufreq提供了驱动等相关的接口负责与系统底层交互。因此，我们可以得到如下的模块架构图。



当然，架构图并非模块的无机拼接，我们还需要结合查阅的资料去丰富架构图的含义。因此，这里的架构图的细节会随着不同的人的理解有所偏差。但是架构图主体的含义很基本一致的。至此，我们完成了待分析的内核代码的所有分析工作。

编辑于 2021-05-29 17:37

Linux　Linux 内核　操作系统

写下你的评论...

1 条评论　默认　最新

杨宁吖
对于一个即将拥抱linux内核源码的小白来说，看到这篇文章只能说是如获至宝，果断收藏，遇到困惑拿出来参考。感谢大神精心编写～
2022-04-28　回复　喜欢

推荐阅读

有了这张图，看谁还能阻拦我研究linux内核源码！

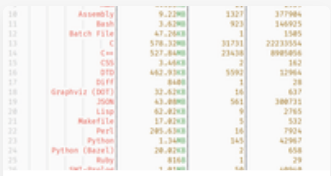
本文首发于我的个人公众号 ytcode，文章链接为 有了这张图，看谁还能阻拦我研究linux内核源码！ 最近在写一个 linux内核启动流程分析 的系列文章，主要是想从源码角度，非常细致的给大家讲…

KING.YT

如何在线阅读linux内核源代码？

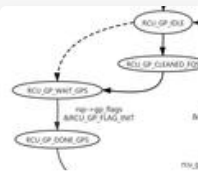
本方法适合：你临时想看linux内核的某个函数，但手头没有下载完整的linux内核源码或者还没建立源码工程。快速反查linux内核某处代码从老版本到新版本的变化，有无新增feature或者BUG修复 bo…

闪光吧Li...　发表于嵌入式Li...



Linux内核有多少行源代码？

叶芝秋



浅谈Linux内核源码

Linux嵌入式