# Linux 的启动流程

本篇的重点是讲解设备和驱动的启动流程，设备和驱动的流程是整个内核启动的核心，也是工作中最常面对的问题。出于知识点的系统性考虑，在进入主题之前我们先看下整个 Linux 在 ARM 中的启动流程如何。

## Uboot 的启动流程

ARM Linux 的启动流程大致为：Uboot → Kernel → Root filesystem。Uboot 在上电的时候就拿到 CPU 的控制权，实现了硬件的初始化。具体是怎么实现的呢？一起来看一下，CPU 的内部集成了小容量的 Sram，而 PC 指针一上电就指向 Sram 的起始地址 0x00000000，所以一上电 Uboot 代码就得到了运行。

Uboot 拿到CPU 使用权就开始做初始化工作，比如关闭看门狗、设置 CPU 运行模式、设置堆栈、初始化内存、网卡、nand flash 等，最后把 Linux 内核加载到内存中。

**初始化 RAM:** 因为内核要在 RAM 中运行，所以在调用内核之前必须初始化和设置 RAM，为调用内核做好准备。

**初始化串口:** 内核在启动过程中可以将信息通过串口输出，这样就可以清楚的知道内核启动信息。虽然串口不是 Uboot 必须要完成的工作，但是通过串口可以方便调试 Uboot 和内核的各种信息。

**检测处理器类型:** Uboot 在调用内核前需要检测系统的处理器类型，并将其保存在某个变量中提供给内核，内核在启动过程中会根据该处理器的类型调用相应的初始化程序。

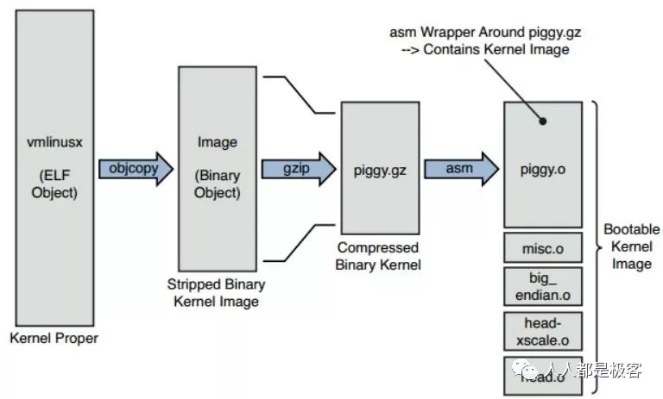
**设置内核启动参数:** 内核在启动过程中会根据该启动参数进行相应的初始化工作。

**调用内核镜像:** 值得注意的是存储 Uboot 的存储器不同，Uboot 的执行过程也并不相同，一般来讲 Flash 分为 nor Flash 和 nand Flash 两种：nor Flash 支持芯片内执行（XIP，eXecute In Place），这样代码可以在 Flash 上直接执行而不必复制到 RAM 中去执行。

但是 nand Flash 并不支持 XIP，所以要想执行 nand Flash 上的代码，必须先将其复制到 RAM 中去，然后跳到 RAM 中去执行。如果内核存放在 nor Flash 中，那么可直接跳转到内核中去执行。但通常由于在 nor Flash 中执行代码会有种种限制，而且速度也远不及 RAM 快，所以一般的嵌入式系统都是将内核复制到 RAM 中，然后跳转到 RAM 中去执行。不论哪种情况，在跳到内核执行之前 CPU 的寄存器必须满足以下条件：r0 ＝ 0，r1 ＝ 处理器类型，r2 ＝ 标记列表在 RAM 中的地址。

## Linux 内核的启动流程（设备和驱动的加载）

关于 Uboot 的启动本课程不做详细介绍，因为本课程的主要内容是内核。在讲述内核启动之前让我们先了解下内核的组成结构：



其中，

（1）vmlinusx 是ELF格式的 Object 文件，这种文件只是各个源代码经过连接以后得到的文件，并不能在 ARM 平台上运行。

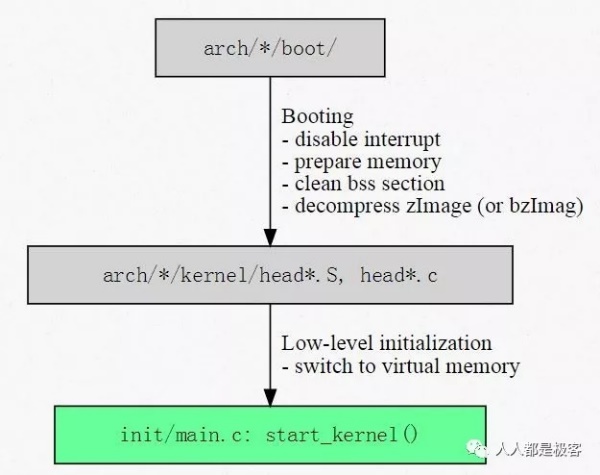
（2）经过 objcopy 这个工具转换以后，得到了二进制格式文件 Image，Image 文件相比于 vmlinusx 文件，除了格式不同以外，还被去除了许多注释和调试的信息。

（3）Image 文件经过压缩以后得到了 piggy.gz，这个文件仅仅是 Image 的压缩版，并无其他不同。

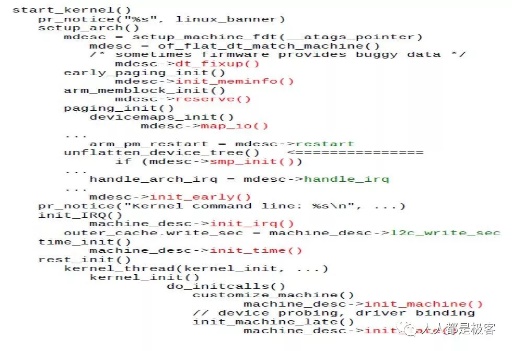
（4）接着编译生成另外几个模块文件 misc.o、big\_endian.o、head.o、head-xscale.o，这几个文件组成一个叫 Bootstrap Loader 的组件，又叫引导程序，编译生成 piggy.o 文件。

（5）最后 piggy.o 文件和 Bootstrap Loader 组成一个 Bootable Kernel Image 文件（可启动文件）。

经过上面的分析不难知道 piggy.o 就是内核镜像，而剩下的几个文件就组成了引导程序。知道了内核的组成结构，Uboot 就是按照内核的组成结构一层一层剥开然后引导内核的：



可以说 start\_kernel() 之前的所有工作都是为了将环境准备好，满足 start\_kernel() 的要求，然后由 start\_kernel() 开始进行内核的加载：



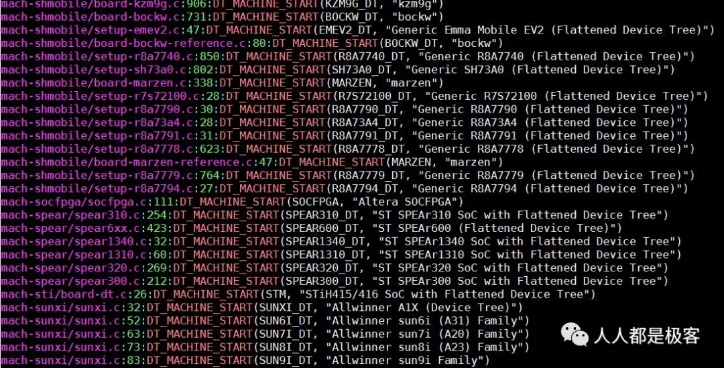
关于 start\_kernl() 函数的内容太多，可以通过红色回调函数看出，start\_kernel() 函数基本是在回调很多对应的注册函数。为了本系列课程的结构性这里就不展开所有知识点讲解，本篇内容接着前一篇设备树的内容重点讲解下设备和驱动的匹配过程。

还记得上一篇讲到的设备树三大作用吗？

* 平台标识；
* 运行时配置；
* 设备信息集合。

接下来我们就看看内核在启动的时候是如何寻找设备，驱动又如何和设备绑定的。

首先在平台目录下可以看到有很多平台描述的文件，如图：

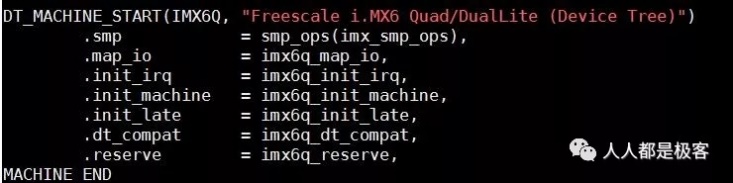


有那么多的平台，我们到底要执行哪个平台是首先要考虑的事情。这也是设备三大功能的第一个功能——平台标识。

* 设备树里有对设备根节点的 Compatible 描述，平台文件里有对 \_\_initconst 的描述，如果两个字段一致则找到了对应的板级文件，这样就通过设备树把要用的设备平台与其他平台区分开来了，如图：



* 找到平台后就可以根据回调函数的指针调用该平台的注册函数。这里以飞思卡尔 imx.6dl 平台为例，回调的时候会调用 imx6q\_init\_machine() 函数，如下：



这里补充一个知识点，细心的读者也许发现了在 Compatible 字段里用逗号分隔了两个字符串。板级匹配的时候用的是哪个字符串，另外一个字符串又是做什么用？首先后面的字段 "fsl,imx6dl" 是抽象共用平台描述符，前面的字段 "fsl,imx6dl-sabresd" 是通用平台下的具体平台描述符，可以理解为母板和子板的区别。在具体的子板文件中我们可以通过前面的字段进行设备信息的获取，如图：



* 接着是运行时配置，让内核在启动的时候根据参数设置进行不同的处理。有经验的读者清楚在 Uboot 里也有对 Bootargs 的配置，这里为什么多此一举呢，是为了在 Uboot 中更灵活的对内核启动进行配置。
* 最后的作用就是设备信息集合，这是设备和驱动匹配的核心，也是工作中面对最多的情况。出于这一作用的内容是工作中经常遇到的重点也是难点，我们专门用一篇内容来详细讲解各级设备是如何展开的，并且手把手教你如何定制一套自己的开发板全新案例。

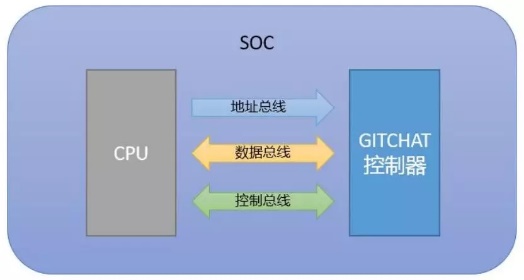
# Linux 总线、设备、驱动模型的探究

## **设备驱动模型的需求**

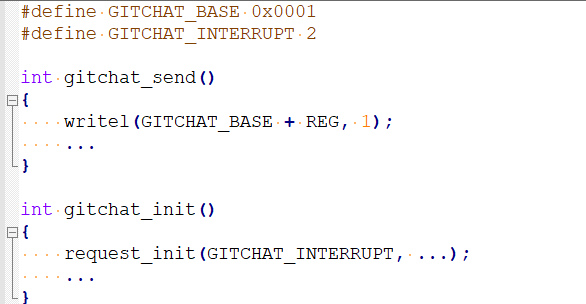
总线、设备和驱动模型，如果把它们之间的关系比喻成生活中的例子是比较容易理解的。举个例子，充电墙壁插座安静的嵌入在墙面上，无论设备是电脑还是手机，插座都能依然不动的完成它的使命——充电，没有说为了满足各种设备充电而去更换插座的。其实这就是软件工程强调的高内聚、低耦合概念。

所谓高内聚低耦合是模块内各元素联系越紧密就代表内聚性就越高，模块间联系越不紧密就代表耦合性低。所以高内聚、低耦合强调的就是内部要紧紧抱团。设备和驱动就是基于这种模型去实现彼此隔离不相干的。这里，有的读者就要问了，高内聚、低耦合的软件模型理解，可设备和驱动为什么要采用这种模型呢？没错，好问题。下面进入今天的话题——总线、设备和驱动模型的探究。

设想一个叫 GITCHAT 的网卡，它需要接在 CPU 的内部总线上，需要地址总线、数据总线和控制总线，以及中断 pin 脚等。



那么在 GITCHAT 的驱动里需要定义 GITCHAT 的基地址、中断号等信息。假设 GITCHAT 的地址为0x0001，中断号是 2，那么：



但是世界上的板子千千万，有三星、华为、飞思卡尔……每个板子的信息也都不一样，站在驱动的角度看，当每次重新换板子的时候，GITCHAT\_BASE 和 GITCHAT\_INTERRUPT 就不再一样，那驱动代码也要随之改变。这样的话一万个开发板要写一万个驱动了，这就是文章刚开始提到的高内聚、低耦合的应用场景。

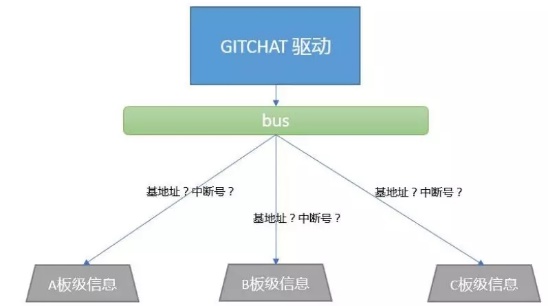
驱动想以不变应万变的姿态适配各种设备连接的话就要实现设备驱动模型。基本上我们可以认为驱动不会因为 CPU 的改变而改变，所以它应该是跨平台的。自然像 “#define GITCHAT\_BASE 0x0001，#define GITCHAT\_INTERRUPT 2” 这样描述和 CPU 相关信息的代码不应该出现在驱动里。

## **设备驱动模型的实现**

现在 CPU 板级信息和驱动分开的需求已经刻不容缓。但是基地址、中断号等板级信息始终和驱动是有一定联系的，因为驱动毕竟要取出基地址、中断号等。怎么取？有一种方法是 GITCHAT 驱动满世界去询问各个板子：请问你的基地址是多少？中断号是几？细心的读者会发现这仍然是一个耦合的情况。



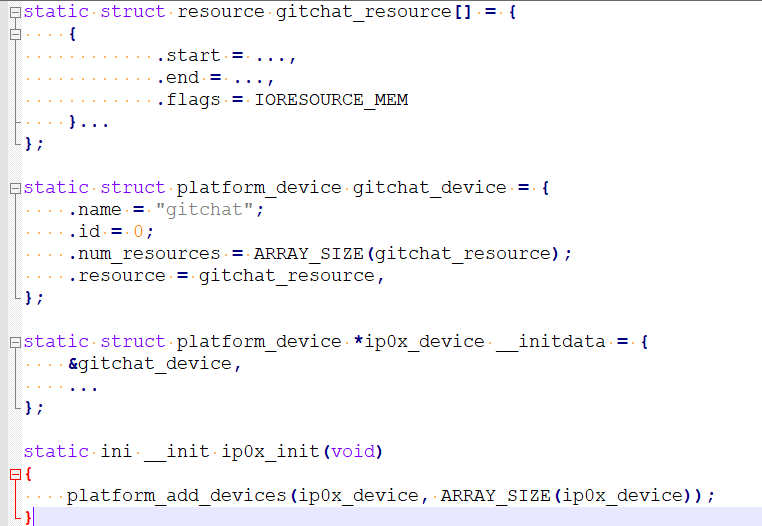
对软件工程熟悉的读者肯定立刻想到能不能设计一个类似接口适配器的类（adapter）去适配不同的板级信息，这样板子上的基地址、中断号等信息都在一个 adapter 里去维护，然后驱动通过这个 adapter 不同的 API 去获取对应的硬件信息。没错，Linux 内核里就是运用了这种设计思想去对设备和驱动进行适配隔离的，只不过在内核里我们不叫做适配层，而取名为总线，意为通过这个总线去把驱动和对应的设备绑定一起，如图：



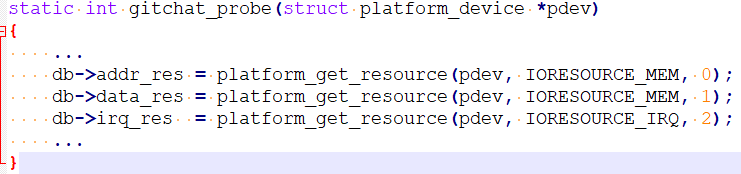
基于这种设计思想，Linux 把设备驱动分为了总线、设备和驱动三个实体，这三个实体在内核里的职责分别如下：



模型设计好后，下面来看一下具体驱动的实践，首先把板子的硬件信息填入设备端，然后让设备向总线注册，这样总线就间接的知道了设备的硬件信息。比如一个板子上有一个 GITCHAT，首先向总线注册：



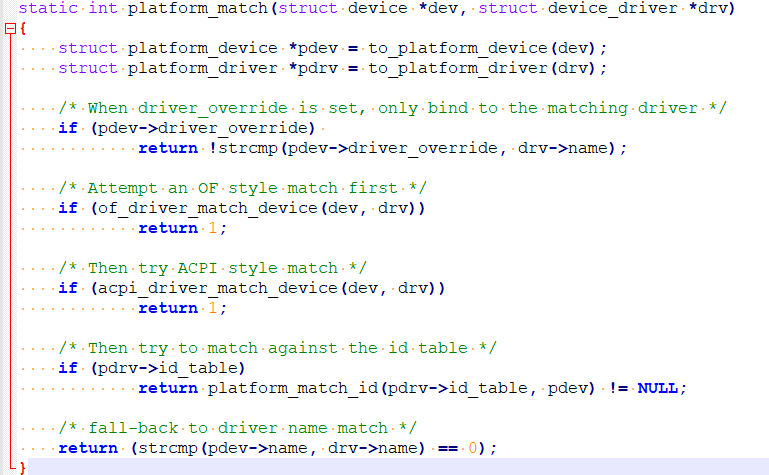
现在 platform 总线上自然知道了板子上关于 GITCHAT 设备的硬件信息，一旦 GITCHAT 的驱动也被注册的话，总线就会把驱动和设备绑定起来，从而驱动就获得了基地址、中断号等板级信息。总线存在的目的就是把设备和对应的驱动绑定起来，让内核成为该是谁的就是谁的和谐世界，有点像我们生活中红娘的角色，把有缘人通过红线牵在一起。设备注册总线的代码例子看完了，下面看下驱动注册总线的代码示例：



从代码中看到驱动是通过总线 API 接口 platform\_get\_resource 取得板级信息，这样驱动和设备之间就实现了高内聚、低耦合的设计，无论你设备怎么换，我驱动就可以岿然不动。

看到这里，可能有些喜欢探究本质的读者又要问了，设备向总线注册了板级信息，驱动也向总线注册了驱动模块，但总线是怎么做到驱动和设备匹配的呢？接下来就讲下设备和驱动是怎么通过总线进行“联姻”的。

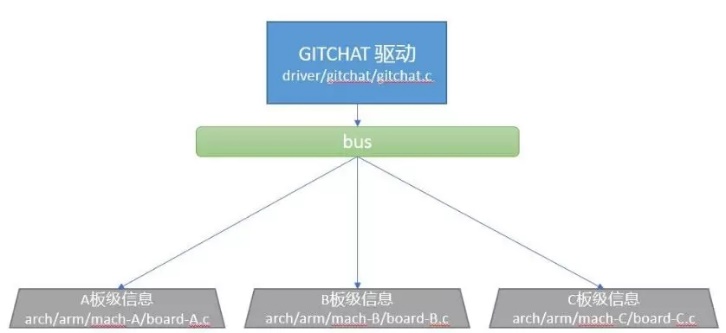
总线里有很多匹配方式，比如：



从上面可知 platform 总线下的设备和驱动是通过名字进行匹配的，先去匹配 platform\_driver 中的 id\_table 表中的各个名字与 platform\_device->name 名字是否相同，如果相同则匹配。

## **设备驱动模型的改善**

相信通过上面的学习，相信对于设备、驱动通过总线来匹配的模型已经有所了解。如果写代码的话应该是下面结构图所示：



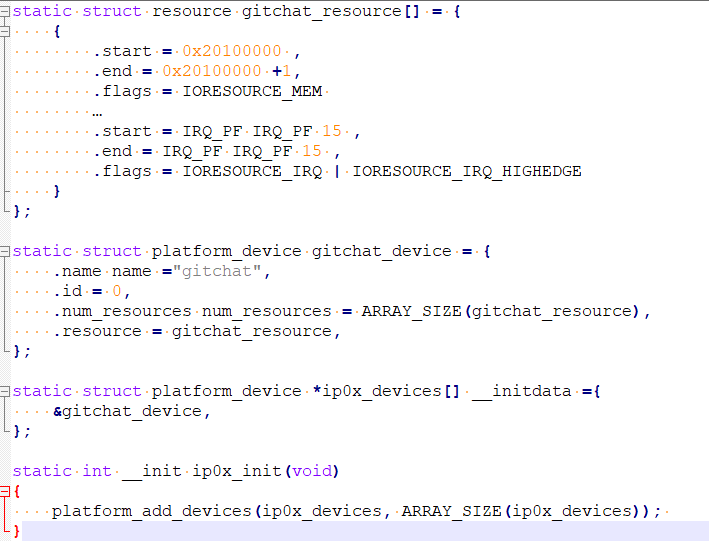
最底层是不同板子的板级文件代码，中间层是内核的总线，最上层是对应的驱动，现在描述板级的代码已经和驱动解耦了，这也是 Linux 设备驱动模型最早的实现机制，但随着时代的发展，就像是人类的贪婪促进了社会的进步一样，开发人员对这种模型有了更高的要求，虽然驱动和设备解耦了，但是天下设备千千万，每次设备的需求改动都要去修改 board-xxx.c 设备文件的话，这样下去，有太多的板级文件需要维护。完美的 Linux 怎么会允许这样的事情存在，于是乎，设备树（DTS）就登向了历史舞台，下一篇内容将探讨设备树的实现原理和用法。

# Linux 设备树（DTS）的深入理解

## **设备树的出现**

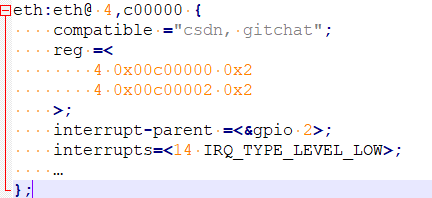
[上一节](https://mp.weixin.qq.com/s?__biz=MzIxMjE1MzU4OA==&mid=2648918733&idx=1&sn=27e9c64b03c6dab26400f9613c514769&chksm=8f5d8f56b82a06408185f6ae4f660f710899f9368e3a823870e72415389f43ee756b3746c9f5&token=142708319&lang=zh_CN&scene=21#wechat_redirect)说过设备树的出现是为了解决内核中大量的板级文件代码，通过 DTS 可以像应用程序里的 XML 语言一样很方便的对硬件信息进行配置。关于设备树的出现其实在 2005 年时候就已经在 PowerPC Linux 里出现了，由于 DTS 的方便性，慢慢地被广泛应用到 ARM、MIPS、X86 等架构上。为了理解设备树的出现的好处，先来看下在使用设备树之前是采用什么方式的。

关于硬件的描述信息之前一般放在一个个类似 arch/xxx/mach-xxx/board-xxx.c 的文件中，如：

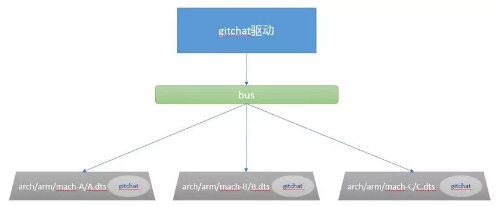


一个很少的地址获取，我们就要写大量的类似代码，当年 Linus 看到内核里有大量的类似代码，很是生气并且在 Linux 邮件列表里发了份邮件，才有了现在的设备树概念，至于设备树的出现到底带来了哪些好处，先看一下设备树的文件：

从代码中可看到对于 GITCHAT 这个网卡驱动、一些寄存器、中断号和上一层 gpio 节点都很清晰的被描述。比上一图的代码优化了很多，也容易维护了很多。这样就形成了设备在脚本，驱动在 c 文件里的关系图：

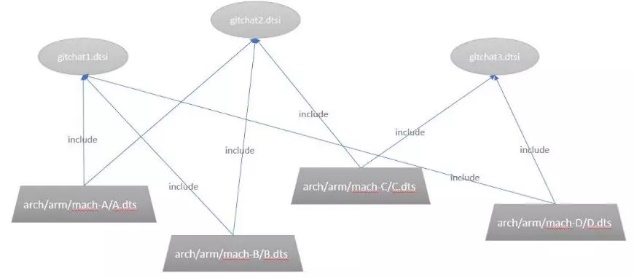


从图中可以看出 A、B、C 三个板子里都含有 GITCHAT 设备树文件，这样对于 GITCHAT 驱动写一份就可以在 A、B、C 三个板子里共用。从上幅图里不难看出，其实设备树的出现在软件模型上相对于之前并没有太大的改变，设备树的出现主要在设备维护上有了更上一层楼的提高，此外在内核编译上使内核更精简，镜像更小。



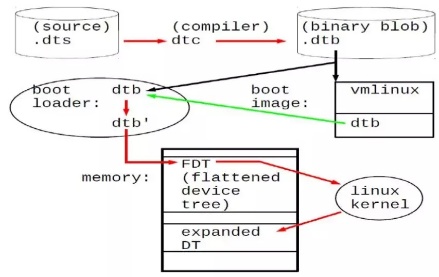
## **设备树的文件结构和剖析**

设备树和设备树之间到底是什么关系，有着哪些依赖和联系，先看下设备树之间的关系图：



除了设备树（DTS）外，还存有 dtsi 文件，就像代码里的头文件一样，是不同设备树共有的设备文件，这不难理解，但是值得注意的是如果 dts 和 dtsi 里都对某个属性进行定义的话，底层覆盖上层的属性定义。这样的好处是什么呢？假如你要做一块电路板，电路板里有很多模块是已经存在的，这样就可以直接像包含头文件一样把共性的 dtsi 文件包含进来，大大减少工作量，后期也可以对类似模块再次利用。

设备树文件的格式是 dts，包含的头文件格式是 dtsi，dts 文件是一种程序员可以看懂的格式，但是 Uboot 和 Linux 只能识别二进制文件，不能直接识别。所以就需要把 dts 文件编译成 dtb 文件。把 dts 编译成 dtb 文件的工具是 dtc，位于内核目录下 scripts/dtc，也可以手动安装：sudo apt-get install device-tree-compiler 工具。具体 dts 是如何转换成机器码并在内存里供 kernel 识别的，请看下图：



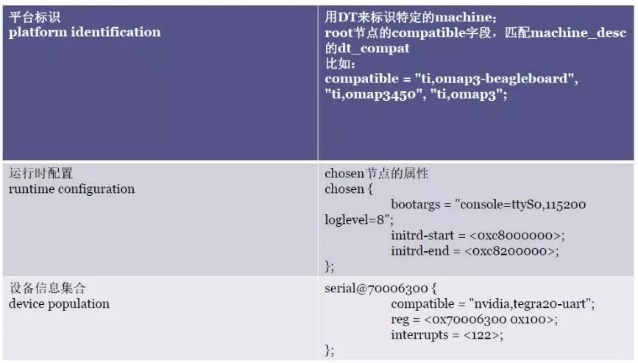
## **设备树的应用**

有了理论，在具体的工程里如何做设备树呢？这里介绍三大法宝：文档、脚本、代码。文档是对各种 node 的描述，位于内核 documentation/devicetree/bingdings/arm/ 下，脚本就是设备树 dts，代码就是你要写的设备代码，一般位于 arch/arm/ 下，以后在写设备的时候可以用这种方法，绝对的事半功倍。很多上层应用开发者没有做过内核开发的经验，对内核一直觉得很神秘，其实可以换一种思路来看内核，相信上层应用开发者最熟悉的就是各种 API，工作中可以说就是和 API 打交道，对于内核也可以想象是各种 API，只不过是内核态的 API。这里设备文件就是根据各种内核态的 API 来调用设备树里的板级信息：

|  |
| --- |
| struct device\_node \*of\_find\_node\_by\_phandle(phandle handle);  struct device\_node \*of\_get\_parent(const struct device\_node\_ \*node);  of\_get\_child\_count()  of\_property\_read\_u32\_array()  of\_property\_read\_u64()  of\_property\_read\_string()  of\_property\_read\_string\_array()  of\_property\_read\_bool() |

具体的用法这里不做进一步的解释，大家可以查询资料或者看官网解释。

这里对设备树做个总结，设备树可以总结为三大作用：一是平台标识，所谓平台标识就是板级识别，让内核知道当前使用的是哪个开发板，这里识别的方式是根据 root 节点下的 compatible 字段来匹配。二是运行时配置，就是在内核启动的时候 ramdisk 的配置，比如 bootargs 的配置，ramdisk 的起始和结束地址。三是设备信息集合，这也是最重要的信息，集合了各种设备控制器，接下来的实践课会对这一作用重点应用。这里有张图对大家理解设备树作用有一定的帮助：



【部分内容整理于宋宝华老师课程】

# Linux 设备和驱动的相遇

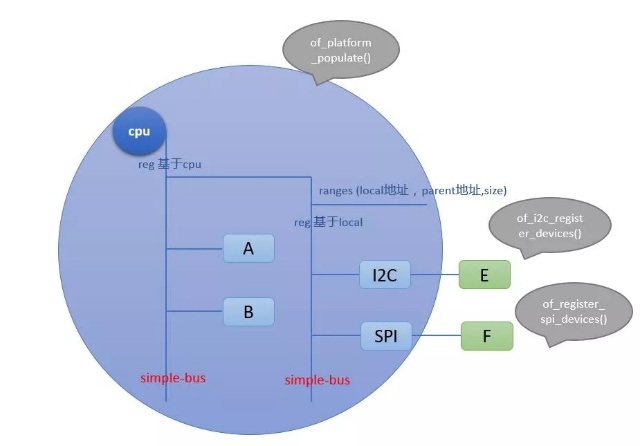
**一个开发板**

我们讲到设备树的三大作用，其最后一个作用也是最重要的作用：设备信息集合。这一节结合设备信息集合的详细讲解来认识一下设备和驱动是如何绑定的。所谓设备信息集合，就是根据不同的外设寻找各自的外设信息，我们知道一个完整的开发板有 CPU 和各种控制器（如 I2C 控制器、SPI 控制器、DMA 控制器等），CPU 和控制器可以统称为 SOC，除此之外还有各种外设 IP，如 LCD、HDMI、SD、CAMERA 等，如下图：

我们看到一个开发板有很多的设备，这些设备是如何一层一层展开的呢？设备和驱动又是如何绑定的呢？我们带着这些疑问进入本节的主题。

## **各级设备的展开**

内核启动的时候是一层一层展开地去寻找设备，设备树之所以叫设备树也是因为设备在内核中的结构就像树一样，从根部一层一层的向外展开，为了更形象的理解来看一张图：



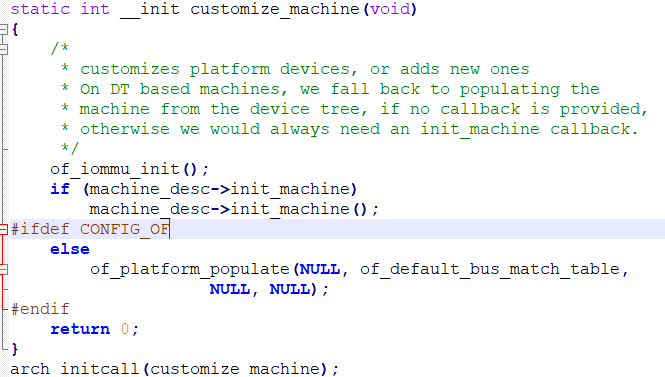
大的圆圈中就是我们常说的 soc，里面包括 CPU 和各种控制器 A、B、I2C、SPI，soc 外面接了外设 E 和 F。IP 外设有具体的总线，如 I2C 总线、SPI 总线，对应的 I2C 设备和 SPI 设备就挂在各自的总线上，但是在 soc 内部只有系统总线，是没有具体总线的。

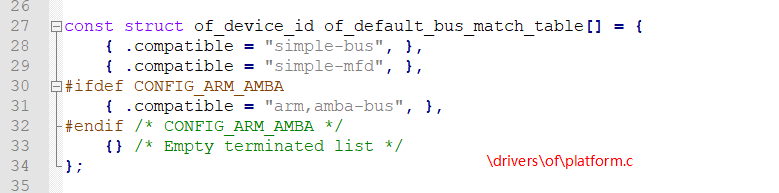
第一节中讲了总线、设备和驱动模型的原理，即任何驱动都是通过对应的总线和设备发生联系的，故虽然 soc 内部没有具体的总线，但是内核通过 platform 这条虚拟总线，把控制器一个一个找到，一样遵循了内核高内聚、低耦合的设计理念。下面我们按照 platform 设备、i2c 设备、spi 设备的顺序探究设备是如何一层一层展开的。

### 1.展开 platform 设备

上图中可以看到红色字体标注的 simple-bus，这些就是连接各类控制器的总线，在内核里即为 platform 总线，挂载的设备为 platform 设备。下面看下 platform 设备是如何展开的。

还记得上一节讲到在内核初始化的时候有一个叫做 init\_machine() 的回调函数吗？如果你在板级文件里注册了这个函数，那么在系统启动的时候这个函数会被调用，如果没有定义，则会通过调用 of\_platform\_populate() 来展开挂在“simple-bus”下的设备，如图（分别位于 kernel/arch/arm/kernel/setup.c，kernel/drivers/of/platform.c）：

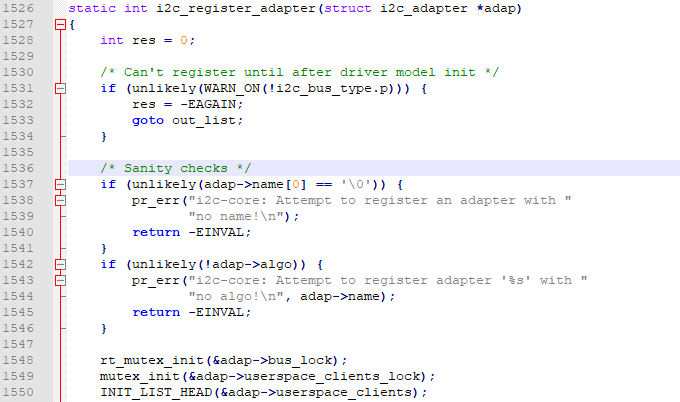


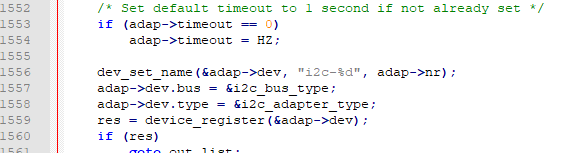


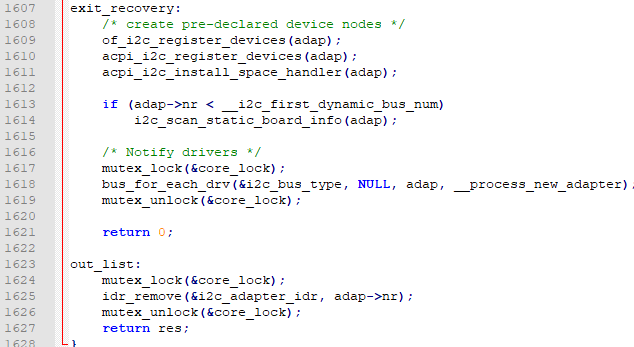
这样就把 simple-bus 下面的节点一个一个的展开为 platform 设备。

### 2.展开 i2c 设备

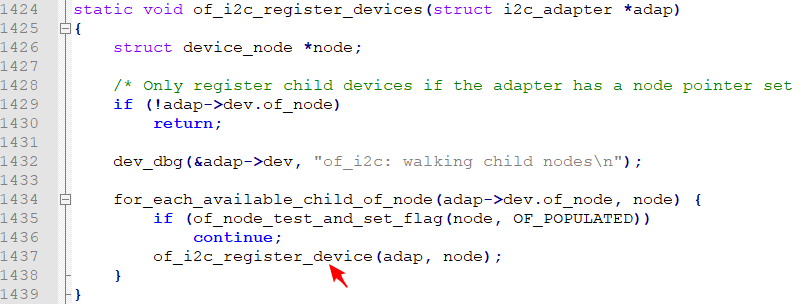
有经验的小伙伴知道在写 i2c 控制器的时候肯定会调用 i2c\_register\_adapter() 函数，该函数的实现如下（kernel/drivers/i2c/i2c-core.c）：







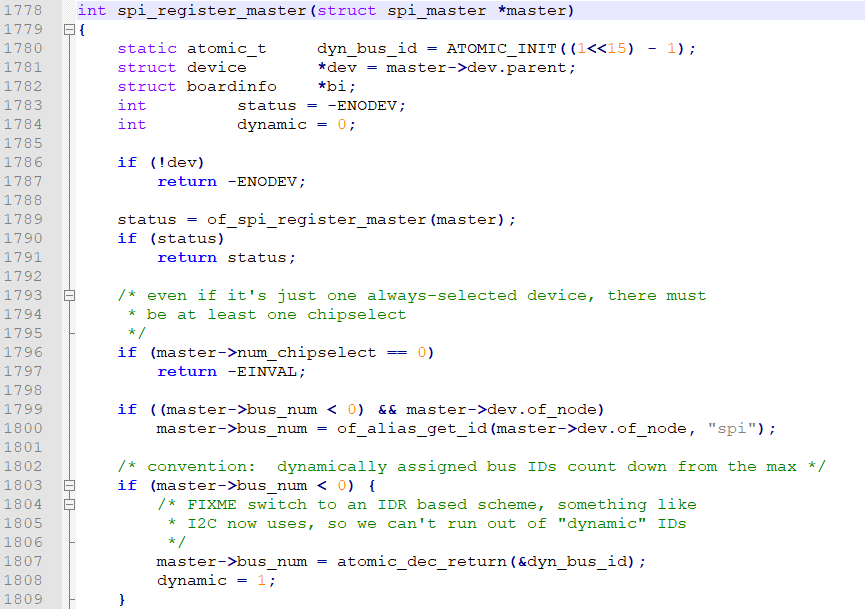
注册函数的最后有一个函数 of\_i2c\_register\_devices(adap)，实现如下：

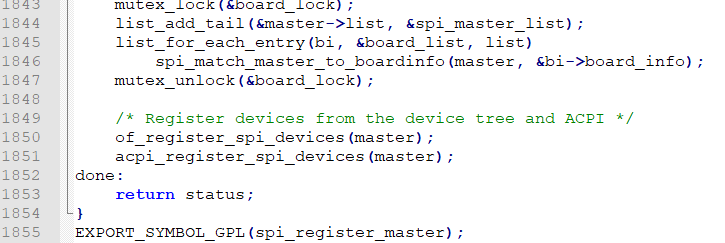


of\_i2c\_register\_devices()函数中会遍历控制器下的节点，然后通过of\_i2c\_register\_device()函数把 i2c 控制器下的设备注册进去。

### 3.展开 spi 设备

spi 设备的注册和 i2c 设备一样，在 spi 控制器下遍历 spi 节点下的设备，然后通过相应的注册函数进行注册，只是和 i2c 注册的 api 接口不一样，下面看一下具体的代码（kernel/drivers/spi/spi.c)：



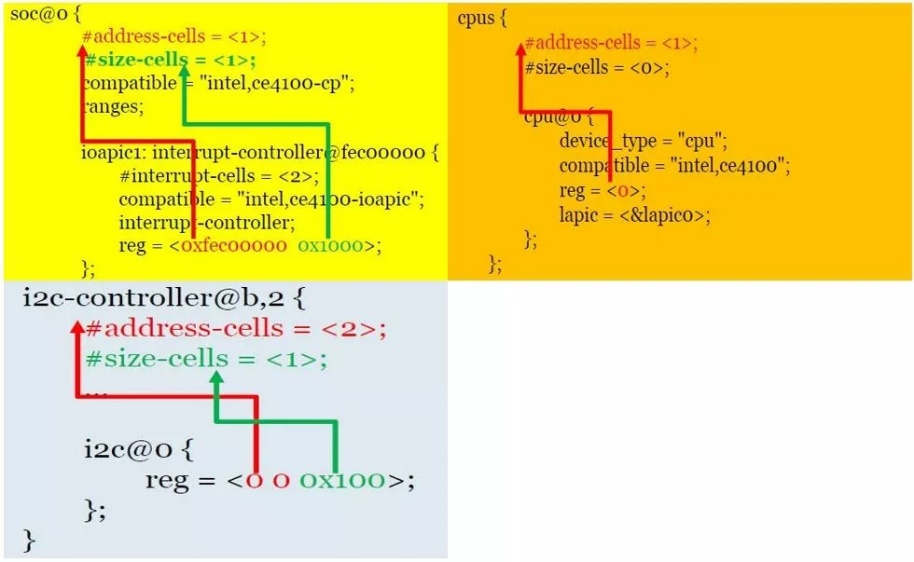


当通过 spi\_register\_master 注册 spi 控制器的时候会通过 of\_register\_spi\_devices 来遍历 spi 总线下的设备，从而注册。这样就完成了 spi 设备的注册。

## **各级设备的展开**

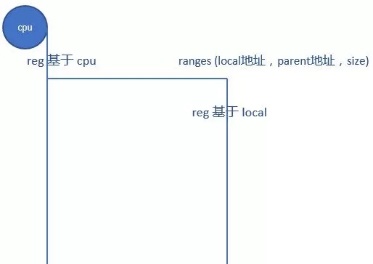
学到这里相信应该了解设备的硬件信息是从设备树里获取的，如寄存器地址、中断号、时钟等等。接下来我们一起看下这些信息在设备树里是怎么记录的，为下一节动手定制开发板做好准备。

### 1.reg 寄存器



我们先看设备树里的 soc 描述信息，红色标注的代表着寄存器地址用几个数据量来表述，绿色标注的代表着寄存器空间大小用几个数据量来表述。图中的含义是中断控制器的基地址是 0xfec00000，空间大小是 0x1000。如果 address-cells 的值是 2 的话表示需要两个数量级来表示基地址，比如寄存器是 64 位的话就需要两个数量级来表示，每个代表着 32 位的数。

### 2.ranges 取值范围



ranges 代表了 local 地址向 parent 地址的转换，如果 ranges 为空的话代表着与 cpu 是 1:1 的映射关系，如果没有 range 的话表示不是内存区域。