# 内核对设备树的处理

目录

[内核对设备树的处理 1](#_Toc536379573)

[1 内核head.S对dtb的简单处理 1](#_Toc536379574)

[2 对设备树中平台信息的处理 2](#_Toc536379575)

[2.1 数据结构简介 2](#_Toc536379576)

[2.2 machine和dts的匹配过程 2](#_Toc536379577)

[3 对设备树中运行时配置信息的处理 2](#_Toc536379578)

[4 dtb转换为device\_node(unflatten) 3](#_Toc536379579)

[5 device\_node转换为platform\_device 4](#_Toc536379580)

[5.1 哪些device\_node可以转换为platform\_device? 4](#_Toc536379581)

[5.2 转换处理过程 4](#_Toc536379582)

[5.3 I2c总线节点的处理 5](#_Toc536379583)

[6 platform\_device跟platform\_driver的匹配 5](#_Toc536379584)

[6.1 注册 platform\_driver 的过程 5](#_Toc536379585)

[6.2 注册 platform\_device 的过程 5](#_Toc536379586)

[6.3 匹配顺序 5](#_Toc536379587)

## 内核head.S对dtb的简单处理

bootloader启动内核时,会设置r0,r1,r2三个寄存器,

r0一般设置为0;

r1一般设置为machine id (在使用设备树时该参数没有被使用);

r2一般设置ATAGS或DTB的开始地址

bootloader给内核传递的参数时有2种方法：ATAGS 或 DTB

head.S对dtb的简单处理的基本流程如下：

\_\_lookup\_processor\_type @ 使用汇编指令读取CPU ID, 根据该ID找到对应的proc\_info\_list结构体(里面含有这类CPU的初始化函数、信息)

\_\_vet\_atags @ 判断是否存在可用的ATAGS或DTB

\_\_create\_page\_tables @ 创建页表, 即创建虚拟地址和物理地址的映射关系

\_\_enable\_mmu @ 使能MMU, 以后就要使用虚拟地址了

\_\_mmap\_switched @ 上述函数里将会调用\_\_mmap\_switched

把bootloader传入的r2参数, 保存到变量\_\_atags\_pointer中

调用C函数start\_kernel

以\_\_lookup\_processor\_type为例，简单分析下相关汇编代码：

\_\_lookup\_processor\_type

adr r3, \_\_lookup\_processor\_type\_data @ 将\_\_lookup\_processor\_type\_data的地址读取到r3寄存器中

@\_\_lookup\_processor\_type\_data:

@ .long .

@ .long \_\_proc\_info\_begin

@ .long \_\_proc\_info\_end

@ .size \_\_lookup\_processor\_type\_data, . - \_\_lookup\_processor\_type\_data

ldmia r3, {r4 - r6} @ 将从3寄存器指向的地址读出数据到r4~r6

sub r3, r3, r4 @ get offset between virt&phys

add r5, r5, r3 @ convert virt addresses to

add r6, r6, r3 @ physical address space

1: ldmia r5, {r3, r4} @ value, mask

and r4, r4, r9 @ mask wanted bits

teq r3, r4 @ 若匹配，则返回，此时r5寄存器中存储对应的proc\_info\_list结构体

beq 2f

add r5, r5, #PROC\_INFO\_SZ @ sizeof(proc\_info\_list)

cmp r5, r6 @ 自增，若proc\_info未遍历结束，则继续进行比较。

blo 1b

mov r5, #0 @ unknown processor

2: ret lr @ 返回，存储对应的proc\_info\_list结构体至r5寄存器中

这段代码读出\_\_lookup\_processor\_type\_data的起始和终止地址，将其转换为物理地址后，遍历这个proc\_info\_list结构体组成的链表，通过寄存器r9传递processor id，与链表中的数据进行比对，若value == mask & processor id，说明该proc\_info\_list结构体匹配成功，此时r5中存放这个结构体的地址，函数返回；若value != mask & processor id，说明processor id未匹配成功，则继续遍历该链表，直至找到对应的proc\_info\_lis结构体。若遍历结束仍未找到对应proc\_info\_lis结构体，则通过r5寄存器返回0。

由于篇幅所限，在这里不对汇编函数一一解析，在head.S中，所做的核心工作无外乎两件事：

把bootloader传来的r1值, 赋给了C变量: \_\_machine\_arch\_type

把bootloader传来的r2值, 赋给了C变量: \_\_atags\_pointer // dtb首地址

## 对设备树中平台信息的处理

### 数据结构简介

内核编译（make）之后会生成两个文件，一个Image，一个zImage，其中Image为内核映像文件，而zImage为内核的一种映像压缩文件，Image大约为4M，而zImage不到2M。 那么uImage又是什么的？它是uboot专用的映像文件，它是在zImage之前加上一个长度为64字节的“头”，说明这个内核的版本、加载位置、生成时间、大小等信息；其0x40之后与zImage没区别。

在uImage中，有一个machine\_desc属性段，用来存放一系列的machine\_desc结构体。machine\_desc中的dt\_compat成员, 它指向一个字符串数组, 里面表示该machine\_desc支持哪些单板。

下面是一个machine\_desc的例子。

**static** **const** char \***const** hi3xxx\_compat[] \_\_initconst = {

"hisilicon,hi3620-hi4511",

NULL,

};

DT\_MACHINE\_START(HI3620, "Hisilicon Hi3620 (Flattened Device Tree)")

.map\_io = hi3620\_map\_io,

.dt\_compat = hi3xxx\_compat,

MACHINE\_END

在dts根节点下的compatible属性中，描述的就是该dts所适用的一个或多个单板名称，从"最兼容"到次之。

下面是海思的dt-bindings中给的一个例子。

Hisilicon Platforms Device Tree Bindings

----------------------------------------------------

Hi4511 Board

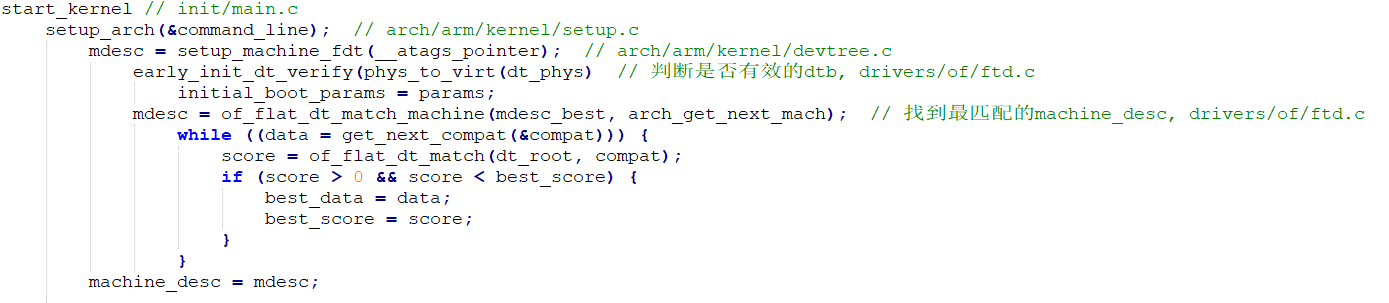
Required root node properties:

- compatible = "hisilicon,hi3620-hi4511","syscon";

### machine和dts的匹配过程

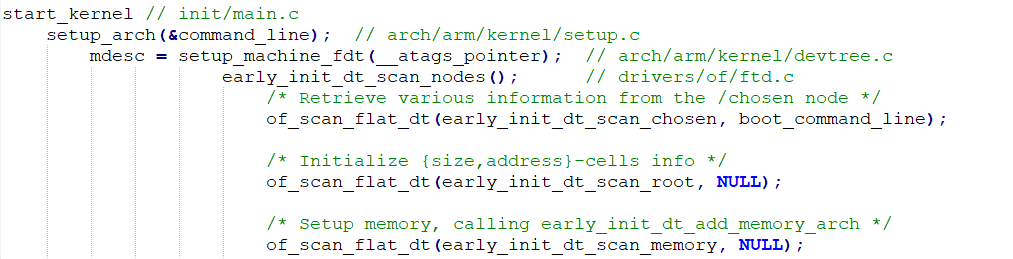
使用compatile属性的值, 跟每一个machine\_desc.dt\_compat中描述的作比较,成绩为"吻合的compatile属性值的位置",成绩越低越匹配, 对应的machine\_desc即被选中。

函数调用过程:



## 对设备树中运行时配置信息的处理

函数调用过程：



通过of\_scan\_flat\_dt函数来扫描设备树中的各个节点，这个函数的第一个参数是一个函数指针，指向针对不同节点的扫描函数。

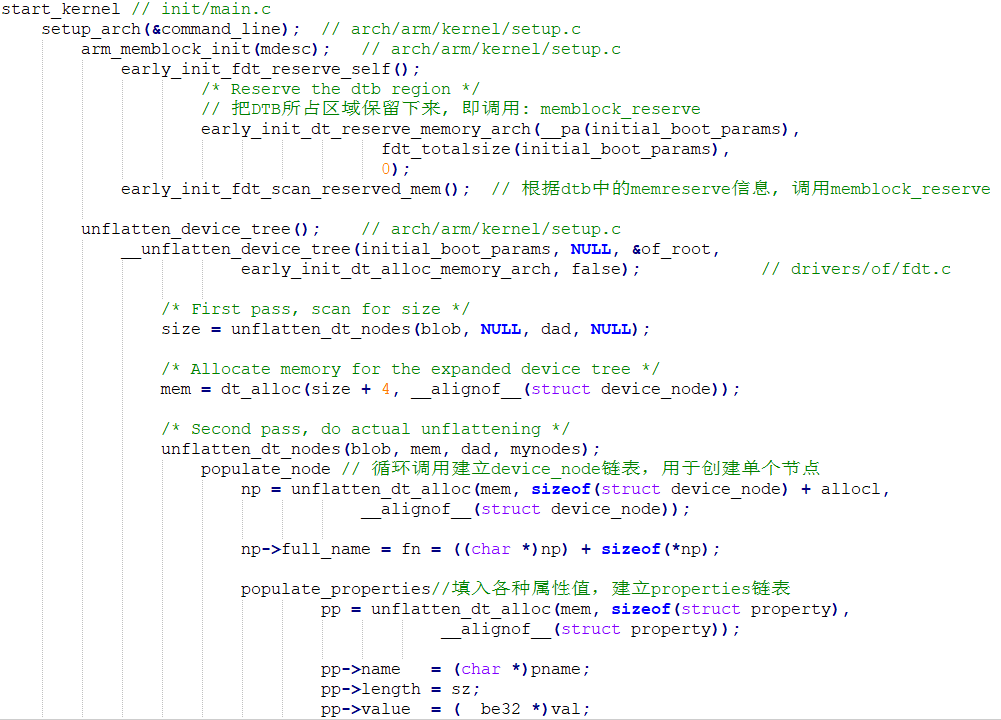
early\_init\_dt\_scan\_chosen将/chosen节点中bootargs属性的值, 存入全局变量boot\_command\_line中。

early\_init\_dt\_scan\_root确定根节点的这2个属性的值: #address-cells, #size-cells，将其存入全局变量: dt\_root\_addr\_cells, dt\_root\_size\_cells中。

early\_init\_dt\_scan\_memory解析/memory中的reg属性, 提取出"base, size", 最终调用memblock\_add(base, size)。

## dtb转换为device\_node(unflatten)

函数调用过程:



每一个节点都转换为一个device\_node结构体:

**struct** device\_node {

**const** char \*name; *// 来自节点中的name属性, 如果没有该属性, 则设为"NULL"*

**const** char \*type; *// 来自节点中的device\_type属性, 如果没有该属性, 则设为"NULL"*

phandle phandle;

**const** char \*full\_name; *// 节点的名字, node-name[@unit-address]*

**struct** fwnode\_handle fwnode;

**struct** property \*properties; *// 节点的属性*

**struct** property \*deadprops; */\* removed properties \*/*

**struct** device\_node \*parent; *// 节点的父亲*

**struct** device\_node \*child; *// 节点的孩子(子节点)*

**struct** device\_node \*sibling; *// 节点的兄弟(同级节点)*

#if defined(CONFIG\_OF\_KOBJ)

**struct** kobject kobj;

#endif

unsigned long \_flags;

void \*data;

#if defined(CONFIG\_SPARC)

**const** char \*path\_component\_name;

unsigned int unique\_id;

**struct** of\_irq\_controller \*irq\_trans;

#endif

};

device\_node结构体中有properties, 用来表示该节点的属性。每一个属性对应一个property结构体:

**struct** property {

char \*name; *// 属性名字, 指向dtb文件中的字符串*

int length; *// 属性值的长度*

void \*value; *// 属性值, 指向dtb文件中value所在位置, 数据仍以big endian存储*

**struct** property \*next;

#if defined(CONFIG\_OF\_DYNAMIC) || defined(CONFIG\_SPARC)

unsigned long \_flags;

#endif

#if defined(CONFIG\_OF\_PROMTREE)

unsigned int unique\_id;

#endif

#if defined(CONFIG\_OF\_KOBJ)

**struct** bin\_attribute attr;

#endif

};

这些device\_node最终构成一棵设备树, 根节点为: of\_root。

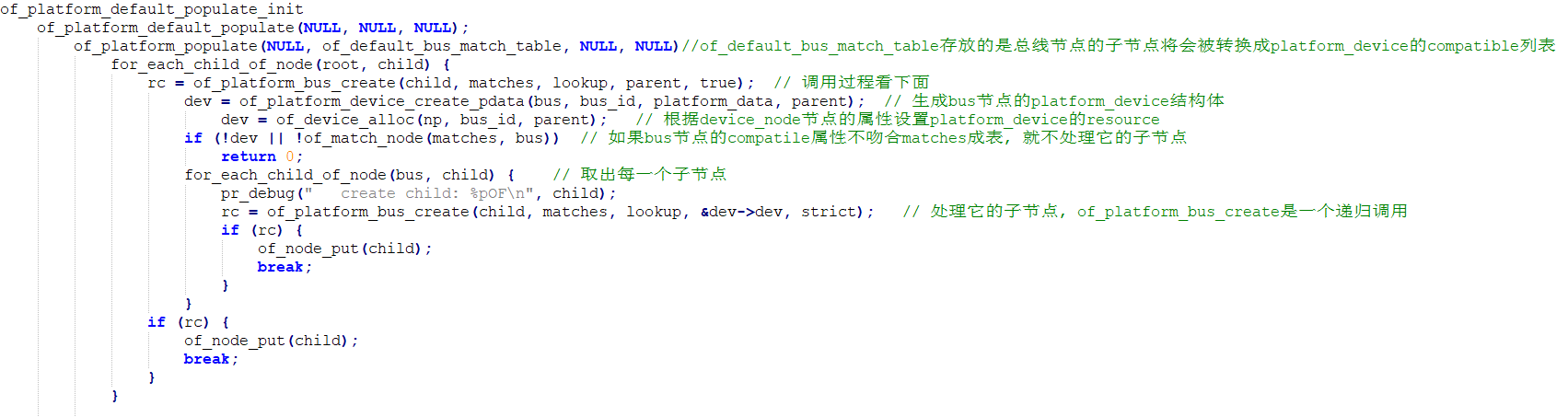
## device\_node转换为platform\_device

### 哪些device\_node可以转换为platform\_device?

1. 根节点下含有compatile属性的子节点。
2. 如果一个结点的compatile属性含有这些特殊的值("simple-bus","simple-mfd","isa","arm,amba-bus")之一, 那么它的子结点(需含compatile属性)也可以转换为platform\_device。

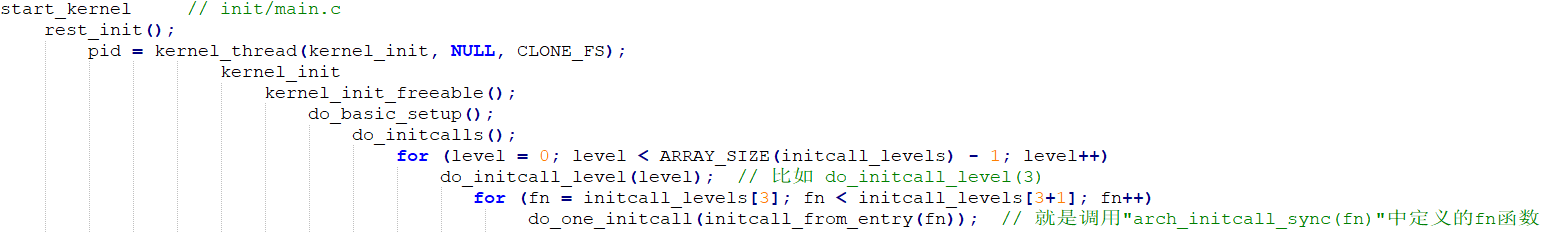
P.S. i2c, spi等总线节点下的子节点, 应该交给对应的总线驱动程序来处理, 它们不应该被转换为platform\_device。

### 转换处理过程



入口函数为of\_platform\_default\_populate\_init，此函数通过添加段属性的宏定义注册，将其放入内核初始化相应字段中。在内核启动进行初始化时，会创建一个初始化进程，在这个初始化进程中，取出初始化字段中保存的函数一一执行，正是在这个过程中生成了platform\_device节点。

内核启动后调用of\_platform\_default\_populate\_init的过程如下图所示：

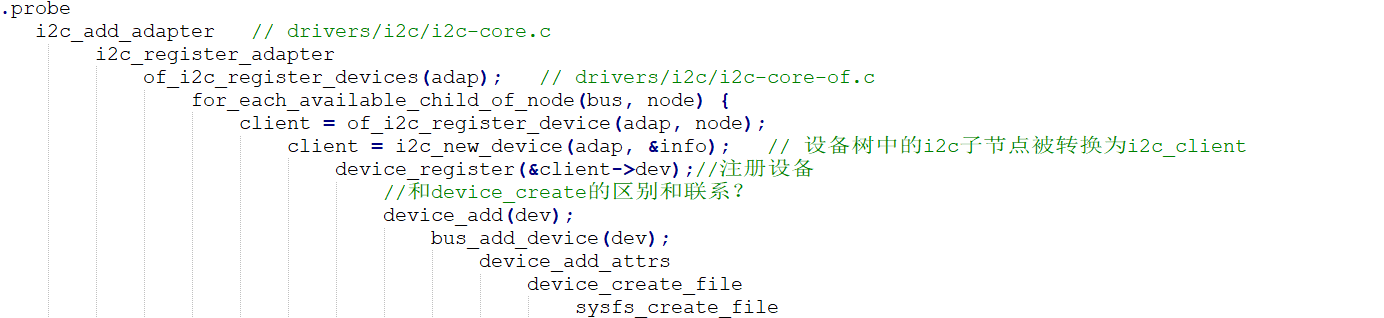


### I2c总线节点的处理

i2c节点一般表示i2c控制器, 它会被转换为platform\_device, 在内核中有对应的platform\_driver;

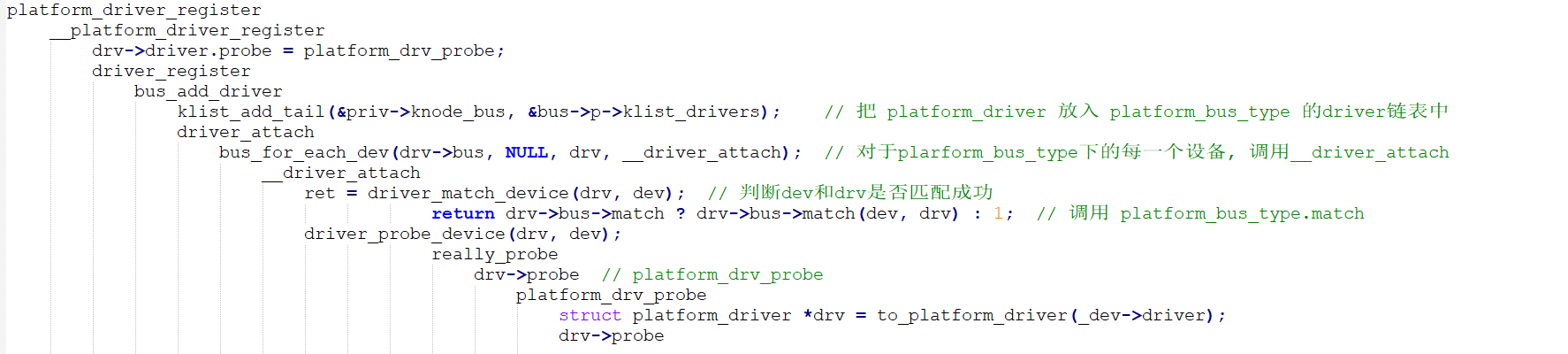
其子节点不会被转换为platform\_device, 它被如何处理完全由父节点的platform\_driver决定, 一般是被创建为一个i2c\_client。

在驱动程序调用probe函数时进行处理，具体过程如下图所示：

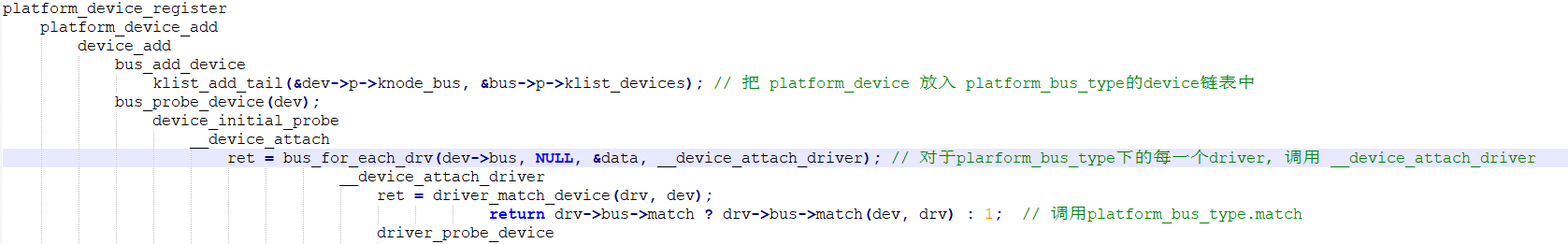


## platform\_device跟platform\_driver的匹配

### 注册 platform\_driver 的过程



### 注册 platform\_device 的过程



### 匹配顺序

匹配函数是platform\_bus\_type.match, 即platform\_match,

匹配过程按优先顺序罗列如下:

a. 比较 platform\_dev.driver\_override 和 platform\_driver.drv->name

b. 比较 platform\_dev.dev.of\_node的compatible属性 和 platform\_driver.drv->of\_match\_table

c. 比较 platform\_dev.name 和 platform\_driver.id\_table

d. 比较 platform\_dev.name 和 platform\_driver.drv->name

有一个成功, 即匹配成功