第10章 驱动进化之路：设备树的引入及简明教程

- 官方文档(可以下载到 devicetree-specification-v0.2.pdf):
  https://www.devicetree.org/specifications/
- 内核文档:
  Documentation/devicetree/booting-without-of.txt

我录制的设备树视频，它是基于 s3c2440 的，用的是 linux 4.19；需要深入研究的可以看该视频(收费)。

注意，如果只是想入门，看本文档及视频即可。

10.1 设备树的引入与作用

以 LED 驱动为例，如果你要更换 LED 所用的 GPIO 引脚，需要修改驱动程序源码、重新编译驱动、重新加载驱动。

在内核中，使用同一个芯片的板子，它们所用的外设资源不一样，比如 A 板用 GPIO A，B 板用 GPIO B。而 GPIO 的驱动程序既支持 GPIO A 也支持 GPIO B，你需要指定使用哪一个引脚，怎么指定？在 c 代码中指定。

随着 ARM 芯片的流行，内核中针对这些 ARM 板保存有大量的、没有技术含量的文件。

Linus 大发雷霆："this whole ARM thing is a f*cking pain in the ass"。

于是，Linux 内核开始引入设备树。

设备树并不是重新发明出来的，在 Linux 内核中其他平台如 PowerPC，早就使用设备树来描述硬件了。

Linus 发火之后，内核开始全面使用设备树来改造，神人就神人。

有一种错误的观点，说“新驱动都是用设备树来写了”。设备树不可能用来写驱动。

请想想，要操作硬件就需要去操作复杂的寄存器，如果设备树可以操作寄存器，那么它就是“驱动”，它就一样很复杂。

设备树只是用来给内核里的驱动程序，指定硬件的信息。比如 LED 驱动，在内核的驱动程序里去操作寄存器，但是操作哪一个引脚？这由设备树指定。

你可以事先体验一下设备树，板子启动后执行下面的命令：

```
[root@100ask:~]# ls /sys/firmware/
devicetree  fdt
```

/sys/firmware/devicetree 目录下是以目录结构程现的 dtb 文件，根节点对应 base 目录，每一个节点对应一个目录，每一个属性对应一个文件。

这些属性的值如果是字符串，可以使用 cat 命令把它打印出来；对于数值，可以用 hexdump 把它打印出来。

一个单板启动时，u-boot 先运行，它的作用是启动内核。U-boot 会把内核和设备树文件都读入内存，然后启动内核。在启动内核时会把设备树在内存

中的地址告诉内核。

10.2 设备树的语法

为什么叫“树”？

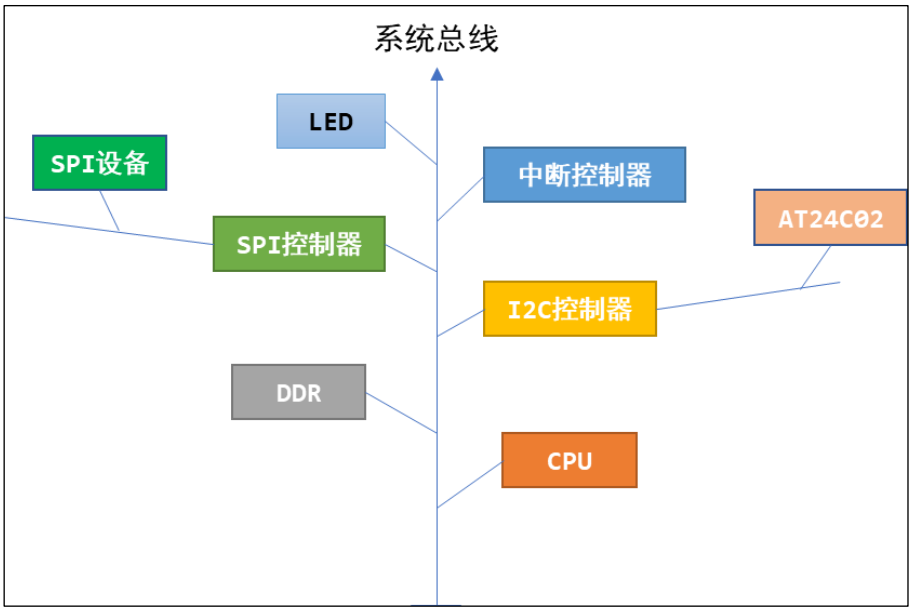

图 10.1 总线“树”

怎么描述这棵树？

我们需要编写设备树文件(dts: device tree source)，它需要编译为dtb(device tree blob)文件，内核使用的是 dtb 文件。

dts 文件是根本，它的语法很简单。

下面是一个设备树示例：

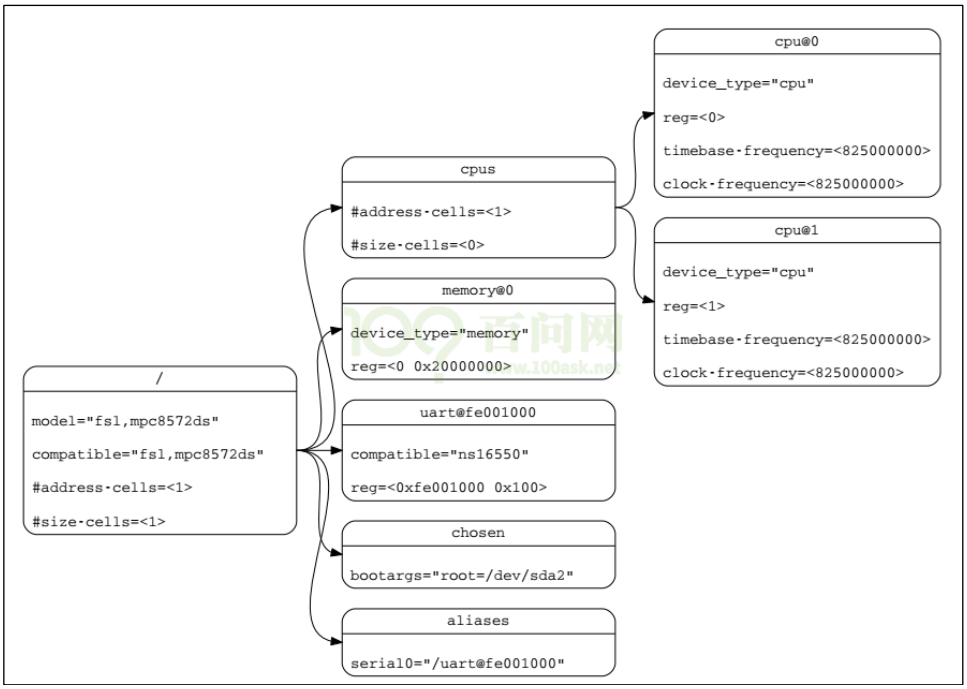

图 10.2 设备树示例

它对应的 dts 文件如下：

```
/dts-v1/;

/ {
    model="fsl,mpc8572ds"
    compatible="fsl,mpc8572ds"
    #address-cells=<1>
    #size-cells=<1>

    cpus {
        #address-cells=<1>
        #size-cells=<0>
        cpu@0 {
            device_type="cpu"
            reg=<0>
            timebase-frequency=<825000000>
            clock-frequency=<825000000>
        };

        cpu@1 {
            device_type="cpu"
            reg=<1>
            timebase-frequency=<825000000>
            clock-frequency=<825000000>
        };

    };

    memory@0 {
        device_type="memory"
        reg=<0 0x20000000>
    };

    uart@fe001000 {
        compatible="ns16550"
        reg=<0xfe001000 0x100>
    };

    chosen {
        bootargs="root=/dev/sda2";
    };

    aliases {
        serial0="/uart@fe001000"
    };
};
```

图 10.3 设备树 dts 文件

10.2.1 Devicetree 格式

1 DTS 文件的格式

DTS 文件布局(layout)：

```
/dts-v1/;                   // 表示版本
[memory reservations]       // 格式为: /memreserve/ <address> <length>;
/ {
    [property definitions]
    [child nodes]
};
```

2 node 的格式

设备树中的基本单元，被称为“node”，其格式为：

```
[label:] node-name[@unit-address] {
    [properties definitions]
    [child nodes]
};
```

label 是标号，可以省略。label 的作用是为了方便地引用 node，比如：

```
/dts-v1/;
/ {
    uart0: uart@fe001000 {
```

```
        compatible="ns16550";
        reg=<0xfe001000 0x100>;
    };
};
```

可以使用下面 2 种方法来修改 uart@fe001000 这个 node：

```
// 在根节点之外使用 label 引用 node：
&uart0 {
    status = “disabled”;
};
或在根节点之外使用全路径：
&{/uart@fe001000}  {
    status = “disabled”;
};
```

3 properties 的格式

简单地说，properties 就是“name=value”，value 有多种取值方式。

- Property 格式 1：

```
[label:] property-name = value;
```

- Property 格式 2(没有值)：

```
[label:] property-name;
```

- Property 取值只有 3 种：

```
arrays of cells(1 个或多个 32 位数据，64 位数据使用 2 个 32 位数据表示)，
string(字符串)，
bytestring(1 个或多个字节)
```

示例：

a) Arrays of cells : cell 就是一个 32 位的数据，用尖括号包围起来

```
interrupts = <17 0xc>;
```

b) 64bit 数据使用 2 个 cell 来表示，用尖括号包围起来:

```
clock-frequency = <0x00000001 0x00000000>;
```

c) A null-terminated string (有结束符的字符串)，用双引号包围起来:

```
compatible = "simple-bus";
```

d) A bytestring(字节序列)，用中括号包围起来:

```
local-mac-address = [00 00 12 34 56 78];  // 每个 byte 使用 2 个 16 进制数来表示
local-mac-address = [000012345678];       // 每个 byte 使用 2 个 16 进制数来表示
```

- 可以是各种值的组合，用逗号隔开：

```
compatible = "ns16550", "ns8250";
example = <0xf00f0000 19>, "a strange property format";
```

10.2.2 dts 文件包含 dtsi 文件

设备树文件不需要我们从零写出来，内核支持了某款芯片比如 imx6ull，在内核的 arch/arm/boot/dts 目录下就有了能用的设备树模板，一般命名为 xxxx.dtsi。“i”表示“include”，被别的文件引用的。

我们使用某款芯片制作出了自己的单板，所用资源跟 xxxx.dtsi 是大部分

相同，小部分不同，所以需要引脚 xxxx.dtsi 并修改。

dtsi 文件跟 dts 文件的语法是完全一样的。

dts 中可以包含.h 头文件，也可以包含 dtsi 文件，在.h 头文件中可以定义一些宏。

示例：

```
/dts-v1/;

#include <dt-bindings/input/input.h>
#include "stm32mp15xx-100ask.dtsi"

/ {
……
};
```

10.2.3 常用的属性

1 #address-cells、#size-cells

- cell 指一个 32 位的数值，
- address-cells：address 要用多少个 32 位数来表示；
- size-cells：size 要用多少个 32 位数来表示。

比如一段内存，怎么描述它的起始地址和大小？

下例中，address-cells 为 1，所以 reg 中用 1 个数来表示地址，即用 0x80000000 来表示地址；size-cells 为 1，所以 reg 中用 1 个数来表示大小，即用 0x20000000 表示大小：

```
/ {
#address-cells = <1>;
#size-cells = <1>;
memory {
reg = <0x80000000 0x20000000>;
    };
};
```

2 compatible

“compatible”表示“兼容”，对于某个 LED，内核中可能有 A、B、C 三个驱动都支持它，那可以这样写：

```
led {
compatible = “A”, “B”, “C”;
};
```

内核启动时，就会为这个 LED 按这样的优先顺序为它找到驱动程序：A、B、C。

根节点下也有 compatible 属性，用来选择哪一个“machine desc”：一个内核可以支持 machine A，也支持 machine B，内核启动后会根据根节点的 compatible 属性找到对应的 machine desc 结构体，执行其中的初始化函数。

compatible 的值，建议取这样的形式："manufacturer,model"，即“厂家名，模块名”。

注意：machine desc 的意思就是“机器描述”，学到内核启动流程时才涉及。

3 model

model 属性与 compatible 属性有些类似，但是有差别。

compatible 属性是一个字符串列表，表示可以你的硬件兼容 A、B、C 等驱动；

model 用来准确地定义这个硬件是什么。

比如根节点中可以这样写：

```
{
    compatible = "samsung,smdk2440", "samsung,mini2440";
    model = "jz2440_v3";
};
```

它表示这个单板，可以兼容内核中的“smdk2440”，也兼容“mini2440”。

从 compatible 属性中可以知道它兼容哪些板，但是它到底是什么板？用 model 属性来明确。

4 status

dtsi 文件中定义了很多设备，但是在你的板子上某些设备是没有的。这时你可以给这个设备节点添加一个 status 属性，设置为“disabled”：

```
&uart1 {
      status = "disabled";
};
```

Table 2.4: Values for status property

Value	Description
"okay"	Indicates the device is operational. 设备正常运行
"disabled"	Indicates that the device is not presently operational, but it might become operational in the future (for example, something is not plugged in, or switched off). 设备不可操作，但是后面可以恢复工作 Refer to the device binding for details on what disabled means for a given device.
"fail"	Indicates that the device is not operational. A serious error was detected in the device, and it is unlikely to become operational without repair. 发生了严重错误，需修复
"fail-sss"	Indicates that the device is not operational. A serious error was detected in the device and it is unlikely to become operational without repair. The *sss* portion of the value is specific to the device and indicates the error condition detected. 发生了严重错误，需修复；sss表示错误信息

图 10.4 设备属性

5 reg

reg 的本意是 register，用来表示寄存器地址。

但是在设备树里，它可以用来描述一段空间。反正对于 ARM 系统，寄存器和内存是统一编址的，即访问寄存器时用某块地址，访问内存时用某块地址，在访问方法上没有区别。

reg 属性的值，是一系列的“address size”，用多少个 32 位的数来表示 address 和 size，由其父节点的#address-cells、#size-cells 决定。

示例：

```
/dts-v1/;
/ {
        #address-cells = <1>;
        #size-cells = <1>;
        memory {
                reg = <0x80000000 0x20000000>;
        };
```

```
};
```

6 name(过时了，建议不用)

它的值是字符串，用来表示节点的名字。在跟 platform_driver 匹配时，优先级最低。

compatible 属性在匹配过程中，优先级最高。

11.2.3.7 1device_type(过时了，建议不用)

它的值是字符串，用来表示节点的类型。在跟 platform_driver 匹配时，优先级为中。

compatible 属性在匹配过程中，优先级最高。

10.2.4 常用的节点(node)

1 根节点

dts 文件中必须有一个根节点：

```
/dts-v1/;
/ {
model = "SMDK24440";
compatible = "samsung,smdk2440";

#address-cells = <1>;
#size-cells = <1>;
};
```

根节点中必须有这些属性：

```
#address-cells // 在它的子节点的 reg 属性中，使用多少个 u32 整数来描述地址(address)
#size-cells   // 在它的子节点的 reg 属性中，使用多少个 u32 整数来描述大小(size)
compatible   // 定义一系列的字符串，用来指定内核中哪个 machine_desc 可以支持本设备
             // 即这个板子兼容哪些平台
             // uImage : smdk2410 smdk2440 mini2440     ==> machine_desc
model       // 咱这个板子是什么
            // 比如有 2 款板子配置基本一致，它们的 compatible 是一样的
            // 那么就通过 model 来分辨这 2 款板子
```

2 CPU 节点

一般不需要我们设置，在 dtsi 文件中都定义好了：

```
cpus {
       #address-cells = <1>;
       #size-cells = <0>;

       cpu0: cpu@0 {
           .......
        }
};
```

3 memory 节点

芯片厂家不可能事先确定你的板子使用多大的内存，所以 memory 节点需要板厂设置，比如：

```
memory {
reg = <0x80000000 0x20000000>;
};
```

4 chosen 节点

我们可以通过设备树文件给内核传入一些参数，这要在 chosen 节点中设置 bootargs 属性：

```
chosen {
bootargs = "noinitrd root=/dev/mtdblock4 rw init=/linuxrc console=ttySAC0,115200";
};
```

10.3 编译、更换设备树

我们一般不会从零写 dts 文件，而是修改。程序员水平有高有低，改得对不对？需要编译一下。并且内核直接使用 dts 文件的话，就太低效了，它也需要使用二进制格式的 dtb 文件。

10.3.1 在内核中直接 make

设置 ARCH、CROSS_COMPILE、PATH 这三个环境变量后，进入 ubuntu 上板子内核源码的目录，执行如下命令即可编译 dtb 文件：

```
make  dtbs  V=1
```

10.3.2 手工编译

除非你对设备树比较了解，否则不建议手工使用 dtc 工具直接编译。

内核目录下 scripts/dtc/dtc 是设备树的编译工具，直接使用它的话，包含其他文件时不能使用“#include”，而必须使用“/incldue”。

编译、反编译的示例命令如下，“-I”指定输入格式，“-O”指定输出格式，“-o”指定输出文件：

```
./scripts/dtc/dtc -I dts -O dtb -o tmp.dtb arch/arm/boot/dts/xxx.dts  // 编译 dts 为 dtb
./scripts/dtc/dtc -I dtb -O dts -o tmp.dts arch/arm/boot/dts/xxx.dtb  // 反编译 dtb 为 dts
```

10.3.3 给开发板更换设备树文件

怎么给各个单板编译出设备树文件，它们的设备树文件是哪一个？

基本方法都是：设置 ARCH、CROSS_COMPILE、PATH 这三个环境变量后，在内核源码目录中执行：

```
make  dtbs
```

10.3.4 板子启动后查看设备树

板子启动后执行下面的命令：

```
[root@100ask:~]# ls /sys/firmware/
devicetree  fdt
```

/sys/firmware/devicetree 目录下是以目录结构程现的 dtb 文件，根节点对应 base 目录，每一个节点对应一个目录，每一个属性对应一个文件。

这些属性的值如果是字符串，可以使用 cat 命令把它打印出来；对于数值，可以用 hexdump 把它打印出来。

还可以看到/sys/firmware/fdt 文件，它就是 dtb 格式的设备树文件，可以把它复制出来放到 ubuntu 上，执行下面的命令反编译出来(-I dtb：输入格式是 dtb，-O dts：输出格式是 dts)：

```
cd  板子所用的内核源码目录
./scripts/dtc/dtc  -I  dtb  -O  dts   /从板子上复制出来的 fdt  -o   tmp.dts
```

10.4 内核对设备树的处理

从源代码文件 dts 文件开始，设备树的处理过程为：

图 10.5 设备树的处理过程

① dts 在 PC 机上被编译为 dtb 文件；

② u-boot 把 dtb 文件传给内核；

③ 内核解析 dtb 文件，把每一个节点都转换为 device_node 结构体；

④ 对于某些 device_node 结构体，会被转换为 platform_device 结构体。

10.4.1 dtb 中每一个节点都被转换为 device_node 结构体

```
struct device_node {
    const char *name;
    const char *type;
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct  property *properties;
    struct  property *deadprops;    /* removed properties */
    struct  device_node *parent;
    struct  device_node *child;
    struct  device_node *sibling;
    struct  kobject kobj;
    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
} « end device_node » ;
```

```
struct property {
    char    *name;
    int length;
    void    *value;
    struct property *next;
    unsigned long _flags;
    unsigned int unique_id;
    struct bin_attribute attr;
};
```

图 10.6 dtb 节点

根节点被保存在全局变量 of_root 中，从 of_root 开始可以访问到任意节点。

10.4.2 哪些设备树节点会被转换为 platform_device

① 根节点下含有 compatile 属性的子节点

② 含有特定 compatile 属性的节点的子节点

如果一个节点的 compatile 属性，它的值是这 4 者之一："simple-bus","simple-mfd","isa","arm,amba-bus"，那么它的子结点(需含 compatile 属性)也可以转换为 platform_device。

③ 总线 I2C、SPI 节点下的子节点：不转换为 platform_device

某个总线下到子节点，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。

比如以下的节点中：

- /mytest 会被转换为 platform_device，因为它兼容"simple-bus";
  它的子节点/mytest/mytest@0 也会被转换为 platform_device
- /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;
- /i2c/at24c02 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。
- 类似的也有/spi 节点，它一般也是用来表示 SPI 控制器，它会被转换为 platform_device，在内核中有对应的 platform_driver;
- /spi/flash@0 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 spi_device。

```
{
     mytest {
         compatile = "mytest", "simple-bus";
         mytest@0 {
              compatile = "mytest_0";
         };
     };

     i2c {
         compatile = "samsung,i2c";
         at24c02 {
              compatile = "at24c02";
         };
     };
     spi {
         compatile = "samsung,spi";
         flash@0 {
              compatible = "winbond,w25q32dw";
              spi-max-frequency = <25000000>;
              reg = <0>;
            };
     };
 };
```

10.4.3 怎么转换为 platform_device

内核处理设备树的函数调用过程，这里不去分析；我们只需要得到如下结论：

a) platform_device 中含有 resource 数组, 它来自 device_node 的 reg, interrupts 属性;
b) platform_device.dev.of_node 指向 device_node, 可以通过它获得其他属性

10.5 platform_device 如何与 platform_driver 配对

从设备树转换得来的 platform_device 会被注册进内核里，以后当我们每注册一个 platform_driver 时，它们就会两两确定能否配对，如果能配对成功就调用 platform_driver 的 probe 函数。

套路是一样的。我们需要将前面讲过的“匹配规则”再完善一下，先贴源码：

```
static int platform_match(struct device *dev, struct device_driver *drv)
{
        struct platform_device *pdev = to_platform_device(dev);
        struct platform_driver *pdrv = to_platform_driver(drv);

        /* When driver_override is set, only bind to the matching driver */
①       if (pdev->driver_override)
                return !strcmp(pdev->driver_override, drv->name);

        /* Attempt an OF style match first */
②       if (of_driver_match_device(dev, drv))
                return 1;

        /* Then try ACPI style match */
        if (acpi_driver_match_device(dev, drv))
                return 1;

        /* Then try to match against the id table */
③       if (pdrv->id_table)
                return platform_match_id(pdrv->id_table, pdev) != NULL;

        /* fall-back to driver name match */
④       return (strcmp(pdev->name, drv->name) == 0);
}
```

图 10.7 platform_match 源码

10.5.1 最先比较：是否强制选择某个 driver

- 比较：

platform_device.driver_override 和 platform_driver.driver.name

可以设置 platform_device 的 driver_override，强制选择某个 platform_driver。

10.5.2 然后比较：设备树信息

- 比较：

`platform_device.dev.of_node` 和 `platform_driver.driver.of_match_table`。

由设备树节点转换得来的 platform_device 中，含有一个结构体：of_node。它的类型如下：

```
struct device_node {
    const char *name; 来自节点的name属性
    const char *type; 来自节点的device_type属性
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct  property *properties;含有compatible属性
```

如果一个 `platform_driver` 支持设备树，它的 `platform_driver.driver.of_match_table` 是一个数组，类型如下：

```
struct of_device_id {
    char    name[32];
    char    type[32];
    char    compatible[128];
    const void *data;
};
```

使用设备树信息来判断 dev 和 drv 是否配对时：

① 首先，如果 of_match_table 中含有 compatible 值，就跟 dev 的 compatile 属性比较，若一致则成功，否则返回失败；

② 其次，如果 of_match_table 中含有 type 值，就跟 dev 的 device_type 属性比较，若一致则成功，否则返回失败；

③ 最后，如果 of_match_table 中含有 name 值，就跟 dev 的 name 属性比较，若一致则成功，否则返回失败。

而设备树中建议不再使用 devcie_type 和 name 属性，所以基本上只使用设备节点的 compatible 属性来寻找匹配的 platform_driver。

10.5.3 接下来比较：platform_device_id

比较 `platform_device. name` 和 `platform_driver.id_table[i].name，id_table` 中可能有多项。

`platform_driver.id_table` 是“`platform_device_id`”指针，表示该 drv 支持若干个 device，它里面列出了各个 device 的{.name, .driver_data}，其中的“name”表示该 drv 支持的设备的名字，driver_data 是些提供给该 device 的私有数据。

10.5.4 最后比较

- `platform_device.name` 和 `platform_driver.driver.name` `platform_driver.id_table` 可能为空，这时可以根据 `platform_driver.driver.name` 来寻找同名的 `platform_device`。

10.5.5 一个图概括所有的配对过程

概括出了这个图：

```
static int platform_match(struct device *dev, struct device_driver *drv)
{
        struct platform_device *pdev = to_platform_device(dev);
        struct platform_driver *pdrv = to_platform_driver(drv);

        /* When driver_override is set, only bind to the matching driver */
    ①   if (pdev->driver_override)
                return !strcmp(pdev->driver_override, drv->name);

        /* Attempt an OF style match first */
    ②   if (of_driver_match_device(dev, drv))
                return 1;

        /* Then try ACPI style match */
        if (acpi_driver_match_device(dev, drv))
                return 1;

        /* Then try to match against the id table */
    ③   if (pdrv->id_table)
                return platform_match_id(pdrv->id_table, pdev) != NULL;

        /* fall-back to driver name match */
    ④   return (strcmp(pdev->name, drv->name) == 0);
}
```

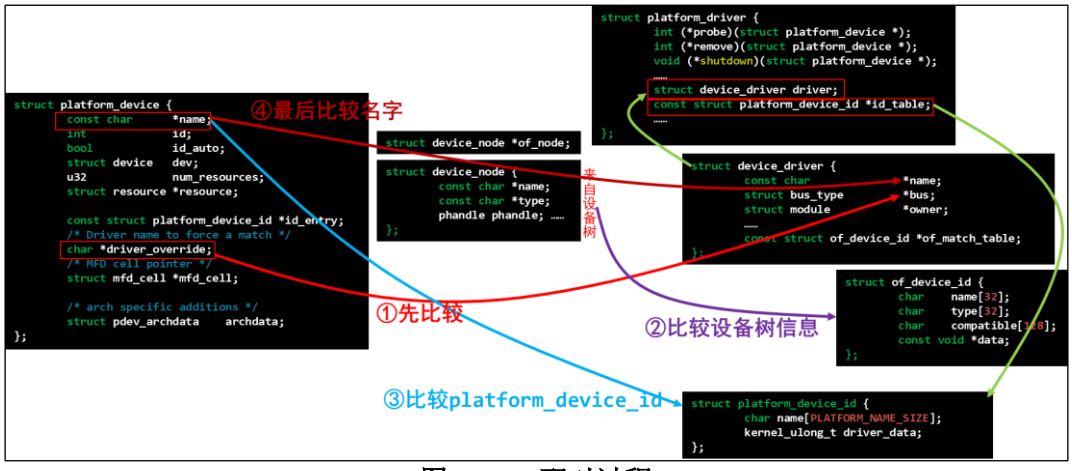

图 10.8 配对过程

10.6 没有转换为 platform_device 的节点，如何使用

任意驱动程序里，都可以直接访问设备树。

你可以使用<>>>>>>>>第五篇 10.7 内核里操作设备树的常用函数>节中介绍的函数找到节点，读出里面的值。

10.7 内核里操作设备树的常用函数

内核源码中 include/linux/目录下有很多 of 开头的头文件，of 表示“open firmware”即开放固件。

10.7.1 内核中设备树相关的头文件介绍

设备树的处理过程是：dtb -> device_node -> platform_device。

1 处理 DTB

```
of_fdt.h            // dtb 文件的相关操作函数，我们一般用不到，
// 因为 dtb 文件在内核中已经被转换为 device_node 树(它更易于使用)
```

2 处理 device_node

```
of.h                // 提供设备树的一般处理函数，
// 比如 of_property_read_u32(读取某个属性的 u32 值)，
// of_get_child_count(获取某个 device_node 的子节点数)
of_address.h        // 地址相关的函数，
// 比如 of_get_address(获得 reg 属性中的 addr，size 值)
// of_match_device (从 matches 数组中取出与当前设备最匹配的一项)
of_dma.h            // 设备树中 DMA 相关属性的函数
of_gpio.h           // GPIO 相关的函数
of_graph.h          // GPU 相关驱动中用到的函数，从设备树中获得 GPU 信息
of_iommu.h          // 很少用到
of_irq.h            // 中断相关的函数
of_mdio.h           // MDIO (Ethernet PHY) API
of_net.h            // OF helpers for network devices.
of_pci.h            // PCI 相关函数
of_pdt.h            // 很少用到
of_reserved_mem.h   // reserved_mem 的相关函数
```

3 处理 platform_device

```
of_platform.h       // 把 device_node 转换为 platform_device 时用到的函数，
                    // 比如 of_device_alloc(根据 device_node 分配设置 platform_device)，
                    // of_find_device_by_node (根据 device_node 查找到 platform_device)，
                    //   of_platform_bus_probe (处理 device_node 及它的子节点)
of_device.h         // 设备相关的函数，比如 of_match_device
```

10.7.2 platform_device 相关的函数

of_platform.h 中声明了很多函数，但是作为驱动开发者，我们只使用其中的 1、2 个。其他的都是给内核自己使用的，内核使用它们来处理设备树，转换得到 platform_device。

1 of_find_device_by_node

函数原型为：

```
extern struct platform_device *of_find_device_by_node(struct device_node *np);
```

设备树中的每一个节点，在内核里都有一个 device_node；你可以使用

device_node 去找到对应的 platform_device。

2 platform_get_resource

这个函数跟设备树没什么关系，但是设备树中的节点被转换为 platform_device 后，设备树中的 reg 属性、interrupts 属性也会被转换为“resource”。

这时，你可以使用这个函数取出这些资源。

函数原型为：

```
/**
 * platform_get_resource - get a resource for a device
 * @dev: platform device
 * @type: resource type   // 取哪类资源？IORESOURCE_MEM、IORESOURCE_REG
*                        // IORESOURCE_IRQ 等
 * @num: resource index  // 这类资源中的哪一个？
 */
struct resource *platform_get_resource(struct platform_device *dev,
                       unsigned int type, unsigned int num);
```

对于设备树节点中的 reg 属性，它对应 IORESOURCE_MEM 类型的资源；

对于设备树节点中的 interrupts 属性，它对应 IORESOURCE_IRQ 类型的资源。

10.7.3 有些节点不会生成 platform_device，怎么访问它们

内核会把 dtb 文件解析出一系列的 device_node 结构体，我们可以直接访问这些 device_node。

内核源码 incldue/linux/of.h 中声明了 device_node 和属性 property 的操作函数，device_node 和 property 的结构体定义如下：

```
struct device_node {
    const char *name;
    const char *type;
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct  property *properties;
    struct  property *deadprops;    /* removed properties */
    struct  device_node *parent;
    struct  device_node *child;
    struct  device_node *sibling;
    struct  kobject kobj;
    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
} « end device_node » ;
```

```
struct property {
    char    *name;
    int length;
    void    *value;
    struct property *next;
    unsigned long _flags;
    unsigned int unique_id;
    struct bin_attribute attr;
};
```

图 10.9 device_node 和 property

1 找到节点

① of_find_node_by_path

根据路径找到节点，比如“/”就对应根节点，“/memory”对应 memory 节点。

函数原型：

```
static inline struct device_node *of_find_node_by_path(const char *path);
```

② of_find_node_by_name

根据名字找到节点，节点如果定义了 name 属性，那我们可以根据名字找到它。

函数原型：

```
extern struct device_node *of_find_node_by_name(struct device_node *from,
                                                const char *name);
```

参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。

但是在设备树的官方规范中不建议使用“name”属性，所以这函数也不建议使用。

③ of_find_node_by_type

根据类型找到节点，节点如果定义了 device_type 属性，那我们可以根据类型找到它。

函数原型：

```
extern struct device_node *of_find_node_by_type(struct device_node *from,
                                                const char *type);
```

参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。

但是在设备树的官方规范中不建议使用“device_type”属性，所以这函数也不建议使用。

④ of_find_compatible_node

根据 compatible 找到节点，节点如果定义了 compatible 属性，那我们可以根据 compatible 属性找到它。

函数原型：

```
extern struct device_node *of_find_compatible_node(struct device_node *from,
                                                   const char *type,
                                                   const char *compat);
```

- 参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。
- 参数 compat 是一个字符串，用来指定 compatible 属性的值；
- 参数 type 是一个字符串，用来指定 device_type 属性的值，可以传入 NULL。

⑤ of_find_node_by_phandle

根据 phandle 找到节点。dts 文件被编译为 dtb 文件时，每一个节点都有一个数字 ID，这些数字 ID 彼此不同。可以使用数字 ID 来找到 device_node。这些数字 ID 就是 phandle。

函数原型：

```
extern struct device_node *of_find_node_by_phandle(phandle handle);
```

- 参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。

⑥ of_get_parent

找到 device_node 的父节点。

函数原型：

```
extern struct device_node *of_get_parent(const struct device_node *node);
```

- 参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。

⑦ of_get_next_parent

这个函数名比较奇怪，怎么可能有“next parent”？

它实际上也是找到 device_node 的父节点，跟 of_get_parent 的返回结果是一样的。

差别在于它多调用下列函数，把 node 节点的引用计数减少了 1。这意味着调用 of_get_next_parent 之后，你不再需要调用 of_node_put 释放 node 节点。

```
of_node_put(node);
```

函数原型：

```
extern struct device_node *of_get_next_parent(struct device_node *node);
```

- 参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。

⑧ of_get_next_child

取出下一个子节点。

函数原型：

```
extern struct device_node *of_get_next_child(const struct device_node *node,
                                             struct device_node *prev);
```

- 参数 node 表示父节点；
- prev 表示上一个子节点，设为 NULL 时表示想找到第 1 个子节点。

不断调用 of_get_next_child 时，不断更新 pre 参数，就可以得到所有的子节点。

⑨ of_get_next_available_child

取出下一个“可用”的子节点，有些节点的 status 是“disabled”，那就会跳过这些节点。

函数原型：

```
struct device_node *of_get_next_available_child(const struct device_node *node,
                                                struct device_node *prev);
```

- 参数 node 表示父节点；
- prev 表示上一个子节点，设为 NULL 时表示想找到第 1 个子节点。

⑩ of_get_child_by_name

根据名字取出子节点。

函数原型：

```
extern struct device_node *of_get_child_by_name(const struct device_node *node,
                                                const char *name);
```

- 参数 node 表示父节点；
- name 表示子节点的名字。

2 找到属性——of_find_property

内核源码 incldue/linux/of.h 中声明了 device_node 的操作函数，当然也包括属性的操作函数：of_find_property

找到节点中的属性。

函数原型：

```
extern struct property *of_find_property(const struct device_node *np,
                                         const char *name,
                                         int *lenp);
```

- 参数 np 表示节点，我们要在这个节点中找到名为 name 的属性。
- lenp 用来保存这个属性的长度，即它的值的长度。

在设备树中，节点大概是这样：

```
xxx_node {
    xxx_pp_name = “hello”;
};
```

上述节点中，“xxx_pp_name”就是属性的名字，值的长度是 6。

3 获取属性的值

① of_get_property

根据名字找到节点的属性，并且返回它的值。

函数原型：

```
/*
 * Find a property with a given name for a given node
 * and return the value.
 */
const void *of_get_property(const struct device_node *np,
                            const char *name,
                            int *lenp)
```

- 参数 np 表示节点，我们要在这个节点中找到名为 name 的属性，然后返回它的值。
- lenp 用来保存这个属性的长度，即它的值的长度。

② of_property_count_elems_of_size

根据名字找到节点的属性，确定它的值有多少个元素(elem)。

函数原型：

```
* of_property_count_elems_of_size - Count the number of elements in a property
 *
 * @np:    device node from which the property value is to be read.
 * @propname:  name of the property to be searched.
 * @elem_size: size of the individual element
 *
 * Search for a property in a device node and count the number of elements of
 * size elem_size in it. Returns number of elements on sucess, -EINVAL if the
 * property does not exist or its length does not match a multiple of elem_size
 * and -ENODATA if the property does not have a value.
 */
int of_property_count_elems_of_size(const struct device_node *np,
                                    const char *propname,
                                    int elem_size)
```

③ 参数 np 表示节点，我们要在这个节点中找到名为 propname 的属性，然后返回下列结果：

```
return prop->length / elem_size;
```

在设备树中，节点大概是这样：

```
xxx_node {
    xxx_pp_name = <0x50000000 1024>  <0x60000000  2048>;
};
```

- 调用 `of_property_count_elems_of_size(np, “xxx_pp_name”, 8)`时，返回值是 2；
- 调用 `of_property_count_elems_of_size(np, “xxx_pp_name”, 4)`时，返回值是 4。

④ 读整数 u32/u64

函数原型为：

```
static inline int of_property_read_u32(const struct device_node *np,
                       const char *propname,
                       u32 *out_value);

extern int of_property_read_u64(const struct device_node *np,
const char *propname,
u64 *out_value);
```

在设备树中，节点大概是这样：

```
xxx_node {
    name1 = <0x50000000>;
    name2 = <0x50000000  0x60000000>;
};
```

- 调用 `of_property_read_u32 (np, “name1”, &val)`时，val 将得到值 `0x50000000`；
- 调用 `of_property_read_u64 (np, “name2”, &val)` 时，val 将得到值 `0x6000000050000000`。

⑤ 读某个整数 u32/u64

函数原型为：

```
extern int of_property_read_u32_index(const struct device_node *np,
                                      const char *propname,
                                      u32 index,
                                      u32 *out_value);
```

在设备树中，节点大概是这样：

```
xxx_node {
    name2 = <0x50000000  0x60000000>;
};
```

- 调用 `of_property_read_u32 (np, “name2”, 1, &val)`时，val 将得到值 `0x60000000`。

⑥ 读数组

函数原型为：

```
int of_property_read_variable_u8_array(const struct device_node *np,
                                       const char *propname,
                                       u8 *out_values,
                                       size_t sz_min,
                                       size_t sz_max);
int of_property_read_variable_u16_array(const struct device_node *np,
                                        const char *propname,
                                        u16 *out_values,
                                        size_t sz_min,
```

```
                                        size_t sz_max);
int of_property_read_variable_u32_array(const struct device_node *np,
                                        const char *propname,
                                        u32 *out_values,
                                        size_t sz_min,
                                        size_t sz_max);
int of_property_read_variable_u64_array(const struct device_node *np,
                                        const char *propname,
                                        u64 *out_values,
                                        size_t sz_min,
                                        size_t sz_max);
```

在设备树中，节点大概是这样：

```
xxx_node {
    name2 = <0x50000012  0x60000034>;
};
```

上述例子中属性 name2 的值，长度为 8。

- 调用 `of_property_read_variable_u8_array (np, “name2”, out_values, 1, 10)` 时，`out_values` 中将会保存这 8 个字节：`0x12,0x00,0x00,0x50,0x34,0x00,0x00,0x60`。
- 调用 `of_property_read_variable_u16_array (np, “name2”, out_values, 1, 10)` 时，`out_values` 中将会保存这 4 个 16 位数值：`0x0012, 0x5000,0x0034,0x6000`。

总之，这些函数要么能取到全部的数值，要么一个数值都取不到：

- 如果值的长度在 sz_min 和 sz_max 之间，就返回全部的数值；
- 否则一个数值都不返回。

⑦ 读字符串

函数原型为：

```
int of_property_read_string(const struct device_node *np,
                            const char *propname,
                            const char **out_string);
```

- 返回节点 np 的属性(名为 propname)的值；
- (*out_string)指向这个值，把它当作字符串。

10.8 怎么修改设备树文件

一个写得好的驱动程序，它会尽量确定所用资源。只把不能确定的资源留给设备树，让设备树来指定。根据原理图确定"驱动程序无法确定的硬件资源"，再在设备树文件中填写对应内容。

那么，所填写内容的格式是什么？

① 使用芯片厂家提供的工具

有些芯片，厂家提供了对应的设备树生成工具，可以选择某个引脚用于某些功能，就可以自动生成设备树节点。

你再把这些节点复制到内核的设备树文件里即可。

② 看绑定文档

内核文档 Documentation/devicetree/bindings/

做得好的厂家也会提供设备树的说明文档

③ 参考同类型单板的设备树文件

④ 网上搜索

⑤ 实在没办法时，只能去研究驱动源码