链滴

# Linux 设备树

作者：zhang-ke-wei
原文链接：https://ld246.com/article/1597130240892
来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20180121.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="一-什么是设备树-">一、什么是设备树？</h2>
Linus Torvalds 在 2011 年 3 月 17 日的 ARM Linux 邮件列表宣称“this whole ARM thing is a fucking pain in the ass”，ARM Linux 社区对此作出了回应，引入设备树。
设备树(Device Tree)，起源于 OpenFirmware (OF)，在过去的 Linux 中，arch/arm/plat-xxx 和 arch/arm/mach-xxx 中充斥着大量的垃圾代码，相当多数的代码只是在描述板级细节，而这些板级节对于内核来讲，不过是垃圾，如板上的 platform 设备、resource、i2c_board_info、spi_board_in o 以及各种硬件的 platform_data。为了改变这种局面，Linux 社区的大牛们参考了 PowerPC 等体架构中使用的 Flattened Device Tree（FDT），也采用了 Device Tree 结构，许多硬件的细节可以接透过它传递给 Linux，而不再需要在 kernel 中进行大量的冗余编码。
描述设备树的文件叫做 DTS(Device Tree Source)，这个 DTS 文件采用树形结构描述板级设备也就是开发板上的设备信息，比如 CPU 数量、 内存基地址、 IIC 接口上接了哪些设备、 SPI 接口上了哪些设备等等。
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/08/image-97f548e4.png?imageView2/2/interlace/1/format/jpg"></
>
<h2 id="二-DTS--DTB-和-DTC">二、DTS、 DTB 和 DTC</h2>
DTS:设备树源文件
DTB:设备树二进制文件
DTC:设备树编译工具
那么 DTS 和 DTB 这两个文件是什么关系呢？ DTS 是设备树源码文件， DTB 是将<br>
DTS 编译以后得到的二进制文件。.c 文件编译为.o 需要用到 gcc 编译器，那么将.dts 编译为.dtb<br>

需要什么工具呢？需要用到 DTC 工具！ DTC 工具源码在 Linux 内核的 scripts/dtc 目录下。
编译设备树：
进入到 Linux 源码根目录下，然后执行如下命令：
<pre><code class="highlight-chroma"><span class="highlight
cl">make dtbs
</code></pre>
<h2 id="三-DTS-语法">三、DTS 语法</h2>
<h3 id="1-dtsi-头文件">1.dtsi 头文件</h3>
和 C 语言一样，设备树也支持头文件，设备树的头文件扩展名为.dtsi。
在.dts 设备树文件中，可以通过“#include”来引用.h、 .dtsi 和.dts 文件，在编写设备树头文的时候一般选使用.dtsi 后缀。
一般.dtsi 文件用于描述 SOC 的内部外设信息，比如 CPU 架构、主频、外设寄存器地址范围，如 UART、 IIC 等等。比如 imx6ull.dtsi 就是描述 I.MX6ULL 这颗 SOC 内部外设情况信息的，内容下：
<pre><code class="highlight-chroma"><span class="highlight
cl">#include <dt-bindings/clock/imx6ul-clock.h>
#include <dt-bi
dings/gpio/gpio.h>
#include <dt-bi
dings/interrupt-controller/arm-gic.h>
#include "imx6ull
pinfunc.h"
#include "imx6ull
pinfunc-snvs.h"
#include "skeleton
dtsi"

/ {
    aliases {
        can0 = &amp

```
flexcan1;
</span></span><span class="highlight-line"><span class="highlight-cl">        can1 = &amp
flexcan2;
</span></span><span class="highlight-line"><span class="highlight-cl">        ethernet0 =
amp;fec1;
</span></span><span class="highlight-line"><span class="highlight-cl">        ethernet1 =
amp;fec2;
</span></span><span class="highlight-line"><span class="highlight-cl">        gpio0 = &a
p;gpio1;
</span></span><span class="highlight-line"><span class="highlight-cl">.....
</span></span><span class="highlight-line"><span class="highlight-cl">    };
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    cpus {
</span></span><span class="highlight-line"><span class="highlight-cl">        #address-cell
 = &lt;1&gt;;
</span></span><span class="highlight-line"><span class="highlight-cl">        #size-cells =
&lt;0&gt;;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        cpu0: cpu@0

</span></span><span class="highlight-line"><span class="highlight-cl">            compatible
= "arm,cortex-a7";
</span></span><span class="highlight-line"><span class="highlight-cl">            device_typ
 = "cpu";
</span></span><span class="highlight-line"><span class="highlight-cl">            reg = &lt;
&gt;;
</span></span><span class="highlight-line"><span class="highlight-cl">.....
</span></span><span class="highlight-line"><span class="highlight-cl">        };
</span></span><span class="highlight-line"><span class="highlight-cl">    };
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    intc: interrupt-c
ntroller@00a01000 {
</span></span><span class="highlight-line"><span class="highlight-cl">        compatible =
"arm,cortex-a7-gic";
</span></span><span class="highlight-line"><span class="highlight-cl">        #interrupt-cel
s = &lt;3&gt;;
</span></span><span class="highlight-line"><span class="highlight-cl">        interrupt-con
troller;
</span></span><span class="highlight-line"><span class="highlight-cl">        reg = &lt;0x
0a01000 0x1000&gt;,
</span></span><span class="highlight-line"><span class="highlight-cl">            &lt;0x00
02000 0x100&gt;;
</span></span><span class="highlight-line"><span class="highlight-cl">    };
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    clocks {
</span></span><span class="highlight-line"><span class="highlight-cl">        #address-cell
 = &lt;1&gt;;
</span></span><span class="highlight-line"><span class="highlight-cl">        #size-cells =
&lt;0&gt;;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        ckil: clock@0

</span></span><span class="highlight-line"><span class="highlight-cl">            compatible
= "fixed-clock";
```

        reg = <
>;
...
    };

    soc {
        #address-cell
 = <1>;
        #size-cells =
<1>;
        compatible =
"simple-bus";
        interrupt-par
nt = <&gpc>;
        ranges;
.....
    };
};
</code></pre>
cpu0 这个设备节点信息，这个节点信息描述了 I.MX6ULL 这颗 SOC 所使用的 CPU 信息，比如构是 cortex-A7，频率支持 996MHz、 792MHz、528MHz、396MHz 和 198MHz 等等。在 imx6ul.dtsi 文件中不仅仅描述了 cpu0 这一个节点信息，I.MX6ULL 这颗 SOC 所有的外设都描述的清清楚。
<h3 id="2-设备节点">2.设备节点</h3>
设备树是采用树形结构来描述板子上的设备信息的文件，每个设备都是一个节点，叫做设备节点每个节点都通过一些属性信息来描述节点信息，属性就是键—值对。
<pre><code class="highlight-chroma"><span class="highlight
cl">/ {
        aliases {
        can0 = &amp
flexcan1;
    };

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

        cpu0: cpu@0

            compatible
= "arm,cortex-a7";
            device_typ
 = "cpu";
            reg = <
>;
        };
    };

    intc: interrupt-c
ntroller@00a01000 {
        compatible =
"arm,cortex-a7-gic";

#interrupt-cells = <3>;
interrupt-controller;
reg = <0x0a01000 0x1000>,
<0x0002000 0x100>;
};
}
</code></pre>
在设备树中节点命名格式如下：
<pre><code class="highlight-chroma">label:node-name@unit-address
</code></pre>
label:标签，可选项，引入 label 的目的就是为了方便访问节点，可以直接通过&label 来访问这个节点，比如通过&cpu0 就可以访问“cpu@0”这个节点，而不需要输入完整的节点名字。
node-name 是节点名字，为 ASCII 字符串，节点名字应该能够清晰的描述出节点的功能，比如“uart1”就表示这个节点是 UART1 外设。
“unit-address”一般表示设备的地址或寄存器首地址，如果某个节点没有地址或者寄存器的话，“unit-address”可以不要，比如“cpu@0”、“interrupt-controller@00a01000”。
<h3 id="3-数据形式">3.数据形式</h3>
每个节点都有不同属性，不同的属性又有不同的内容，属性都是键值对，值可以为空或任意的字符流。设备树源码中常用的几种数据形式如下所示：
<h4 id="--字符串">①、字符串</h4>
compatible = "arm,cortex-a7";<br>
上述代码设置 compatible 属性的值为字符串“arm,cortex-a7”
<h4 id="---32-位无符号整数">②、 32 位无符号整数</h4>
<pre><code class="highlight-chroma">reg = <0>;
</code></pre>
上述代码设置 reg 属性的值为 0， reg 的值也可以设置为一组值，比如：
<pre><code class="highlight-chroma">reg = <0 0x123456 100>;
</code></pre>
<h4 id="--字符串列表">③、字符串列表</h4>
属性值也可以为字符串列表，字符串和字符串之间采用“,”隔开，如下所示：
<pre><code class="highlight-chroma">compatible = "fsl,imx6ull-gpmi-nand", "fsl, imx6ul-gpmi-nand";
</code></pre>
上述代码设置属性 compatible 的值为“fsl,imx6ull-gpmi-nand”和“fsl, imx6ul-gpmi-nand”。
<h3 id="4-标准属性">4.标准属性</h3>
节点是由一堆的属性组成，节点都是具体的设备，不同的设备需要的属性不同，用户可以自定义属性。除了用户自定义属性，有很多属性是标准属性， Linux 下的很多外设驱动都会使用这些标准属性。
<h4 id="-1-compatible-属性">（1）compatible 属性</h4>
compatible 属性也叫做“兼容性”属性，compatible 属性的值是一个字符串列表， compatible 属性用于将设备和驱动绑定起来。字符串列表用于选择设备所要使用的驱动程序， compatible 属性值格式如下所示：
<pre><code class="highlight-chroma">"manufacturer,model"

</code></pre>
manufacturer 表示厂商， model 一般是模块对应的驱动名字。
example:
<pre><code class="highlight-chroma"><span class="highlight
cl">compatible = "fsl,imx6ul-evk-wm8960","fsl,imx-audio-wm8960";
</code></pre>
一般驱动程序文件都会有一个 OF 匹配表，此 OF 匹配表保存着一些 compatible 值，如果设备
点的 compatible 属性值和 OF 匹配表中的任何一个值相等，那么就表示设备可以使用这个驱动。</
>
<h4 id="-2-model-属性">（2）model 属性</h4>
model 属性值也是一个字符串，一般 model 属性描述设备模块信息，比如名字什么的，比如：<
p>
<pre><code class="highlight-chroma"><span class="highlight
cl">model = "wm8960-audio";
</code></pre>
<h4 id="-3-status-属性">（3）status 属性</h4>
表示设备状态， status 属性值是字符串，字符串是设备的状态信息。
<table>
<thead>
<tr>
<th>值</th>
<th>描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>“okay”</td>
<td>表明设备是可操作的。</td>
</tr>
<tr>
<td>“disabled”</td>
<td>表明设备当前是不可操作的，但是在未来可以变为可操作的，比如热插拔设备<br>插入以后。
于 disabled 的具体含义还要看设备的绑定文档。</td>
</tr>
<tr>
<td>“fail”</td>
<td>表明设备不可操作，设备检测到了一系列的错误，而且设备也不大可能变得可<br>操作。</td>
</tr>
<tr>
<td>“fail-sss”</td>
<td>含义和“fail”相同，后面的 sss 部分是检测到的错误内容。</td>
</tr>
</tbody>
</table>
<h4 id="-4--address-cells-和-size-cells-属性">（4）#address-cells 和#size-cells 属性</h4>
这两个属性的值都是无符号 32 位整形， #address-cells 和#size-cells 这两个属性可以用在任
拥有子节点的设备中，用于描述子节点的地址信息。 #address-cells 属性值决定了子节点 reg 属性
地址信息所占用的字长(32 位)， #size-cells 属性值决定了子节点 reg 属性中长度信息所占的字长(32
位)。 #address-cells 和#size-cells 表明了子节点应该如何编写 reg 属性值，一般 reg 属性都是和地
有关的内容，和地址相关的信息有两种：起始地址和地址长度， reg 属性的格式一为：
<pre><code class="highlight-chroma"><span class="highlight
cl">reg = <address1 length1 address2 length2 address3 length3.....>
</code></pre>
每个“address length”组合表示一个地址范围，其中 address 是起始地址， length 是地址长

，address-cells 表明 address 这个数据所占用的字长， size-cells 表明 length 这个数据所占用的字。
<h4 id="-5-reg-属性">（5）reg 属性</h4>
reg 属性的值一般是(address， length)对。 reg 属性一般用于描述设备地址空间资源信息，一都是某个外设的寄存器地址范围信息
<pre><code class="highlight-chroma">reg = <0x02020000 0x4000>;
</code></pre>
reg 属性中 address=0x02020000， length=0x4000。
<h4 id="-6-ranges-属性">（6）ranges 属性</h4>
ranges 属性值可以为空或者按照(child-bus-address,parent-bus-address,length)格式编写的字矩阵，ranges 是一个地址映射/转换表， ranges 属性每个项目由子地址、父地址和地址空间长度三部分组成：<br>
child-bus-address：子总线地址空间的物理地址，由父节点的#address-cells 定此物理地址所占用的字长。<br>
parent-bus-address： 父总线地址空间的物理地址，同样由父节点的#address cells 确定此物理地址所占用的字长。<br>
**length**： 子地址空间的长度，由父节点的#size-cells 确定此地址长度所占用的字长。<br>
如果 ranges 属性值为空值，说明子地址空间和父地址空间完全相同，不需要进行地址转换。
<h4 id="-7-name-属性">（7）name 属性</h4>
name 属性值为字符串， name 属性用于记录节点名字， name 属性已经被弃用，一些老的设树文件可能会使用此属性。
<h4 id="-8-device-type-属性">（8）device_type 属性</h4>
device_type 属性值为字符串， IEEE 1275 会用到此属性，用于描述设备的 FCode，但是设备没有 FCode，所以此属性也被抛弃了。此属性只能用于 cpu 节点或者 memory 节点。
<h3 id="5-根节点-compatible-属性">5.根节点 compatible 属性</h3>
根节点 compatible 属性的作用：通过根节点的 compatible 属性可以知道我们所使用的设备，般第一个值描述了所使用的硬件设备名字，第二个值描述了设备所使用的 SOC，Linux 内核会通过根点的 compoatible 属性查看是否支持此设备，如果支持的话设备就会启动 Linux 内核。
linux 如何判断是否支持某设备？
<h4 id="-1---使用设备树之前设备匹配方法">(1）使用设备树之前设备匹配方法</h4>
在没有使用设备树以前， uboot 会向 Linux 内核传递一个叫做 machine id 的值， machine id 也就是设备 ID，告诉 Linux 内核自己是个什么设备，看看 Linux 内核是否支持。Linux 内核是支持多设备的，针对每一个设备(板子)， Linux 内核都用 MACHINE_START 和 MACHINE_END 来定义个 machine_desc 结构体来描述这个设备。
比如在文件 arch/arm/mach-imx/machmx35_3ds.c 中有如下定义：
<pre><code class="highlight-chroma">
MACHINE_START(MX35_3DS, "Freescale MX35PDK")
	/* Maintainer: Freescale Semiconductor, Inc */
	.atag_offset = 0x100,
	.map_io = mx35_map_io,
	.init_early = mx35_init_early,
	.init_irq = mx35_init_irq,
	.init_time = mx35pdk_timer_init,
	.init_machine = mx35_3ds_init,
	.reserve = mx35_3ds_reserve,

	.restart	= mxc_restart,
MACHINE_END
</code></pre>
其中 MACHINE_START 和 MACHINE_END 定义在文件 arch/arm/include/asm/mach/arch.h 中
<pre><code class="highlight-chroma">#define MACHINE_START(_type, _name)		\
static const struct machine_desc __mach_desc_##_type	\
	__used		\
	__attribute__((__section__(".arch.info.init"))) = {	\
	.nr	= MACH_TYPE_##_type,	\
	.name	= _name,

#define MACHINE_END		\
};
</code></pre>
这里定义了一个 machine_desc 类型的结构体变量__mach_desc_MX35_3DS，这个变量存在“.arch.info.init”段中,MACH_TYPE_MX35_3DS 就是“Freescale MX35PDK”这个板子的 machine id。MACH_TYPE_MX35_3DS 定义在文件 include/generated/mach-types.h 中。
uboot 会给 Linux 内核传递 machine id 这个参数，Linux 内核会检查这个 machine id，其实是将 machine id 与 MACH_TYPE_XXX 宏进行对比，看看有没有相等的，如果相等的话就表示 Linux 内核支持这个设备，如果不支持的话那么这个设备就没法启动 Linux 内核。
<h4 id="-2-使用设备树以后的设备匹配方法">（2）使用设备树以后的设备匹配方法</h4>
当 Linux 内核引入设备树以后就不再使用 MACHINE_START 了，而是换为了 DT_MACHINE_START。DT_MACHINE_START 也定义在文件 arch/arm/include/asm/mach/arch.h 里面，定义如下：
<pre><code class="highlight-chroma">#define DT_MACHINE_START(_name, _namestr)		\
static const struct machine_desc __mach_desc_##_name	\
	__used		\
	__attribute__((__section__(".arch.info.init"))) = {	\
	.nr	= ~0,		\
	.name	= _namestr,

#endif
</code></pre>
DT_MACHINE_START 和 MACHINE_START 基本相同，只是.nr 的设置不同，在 DT_MACHINE_START 里面直接将.nr 设置为~0。说明引入设备树以后不会再根据 machine id 来检查 Linux 内核是支持某个设备了。
machine_desc 结构体中有个.dt_compat 成员变量，此成员变量保存着本设备兼容属性，比较节点下的 compatible 属性和.dt_compat 的值，若相等则表示 Linux 内核支持该设备。

<h2 id="四-设备树在系统中的体现">四、设备树在系统中的体现</h2>
Linux 内核启动的时候会解析设备树中各个节点的信息，并且在根文件系统的/proc/devicetree 录下根据节点名字创建不同文件夹
<h2 id="五-特殊节点">五、特殊节点</h2>
在根节点“/”中有两个特殊的子节点： aliases 和 chosen，我们接下来看一下这两个特殊的子点。
<h3 id="1-aliases-子节点">1.aliases 子节点</h3>
imx6ull.dtsi 文件， aliases 节点内容如下所示：
aliases 节点的主要功能就是定义别名，但是一般加上 label，通过&label 的形式来访问，比 aliases 用得更多。
<h3 id="2-chosen-子节点">2.chosen 子节点</h3>
chosen 并不是一个真实的设备， chosen 节点主要是为了 uboot 向 Linux 内核传递数据，重是 bootargs 参数。一般.dts 文件中 chosen 节点通常为空或者内容很少。
例如 imx6ull-alientekemmc.dts 中 chosen 节点内容：
chosen {<br>
stdout-path = &uart1;<br>
}<br>
启动 linux 后/proc/device-tree/chosen 目录下回出现一个 bootargs 文件，设备树种并没有该内容那么这些内容从什么地方来的呢？uboot 在启动的过程中会向内核传递参数，观察内核启动所打印的息会发现打印了这些信息。
在 uboot 源码中进行搜索会发现与 chosen 相关的代码，在 common/fdt_support.c 文件中有 fdt_chosen 函<br>
数，此函数内容如下所示：
<pre><code class="highlight-chroma">int fdt_chosen(void *fdt)
{
int   nodeoffset;
int   err;
char  *str;       /* u ed to set string properties */

err = fdt_check_h ader(fdt);
if (err < 0) {
    printf("fdt_chos n: %s\n", fdt_strerror(err));
    return err;
}

/* find or create "/ hosen" node. */
nodeoffset = fdt_f nd_or_add_subnode(fdt, 0, "chosen");
if (nodeoffset < 0)
    return nodeoffs t;

str = getenv("boo args");
if (str) {
    err = fdt_setpr p(fdt, nodeoffset, "bootargs", str,
            strlen(str)

原文链接：Linux 设备树

+ 1);
    if (err < 0) {
        printf("WARNING: could not set bootargs %s.\n",
                fdt_strerror(err));
        return err;
    }
}

return fdt_fixup_sdout(fdt, nodeoffset);

}
</code></pre>
fdt_find_or_add_subnode 从设备树(.dtb)中找到 chosen 节点，如果没有找到的话就会自己创一个 chosen 节点。
str = getenv("bootargs");//读取 uboot 中 bootargs 环境变量的内容。
fdt_setprop(fdt, nodeoffset, "bootargs", str, 295 strlen(str) + 1); //调用函数 fdt_setprop 向 hosen 节点添加 bootargs 属性，并且 bootargs 属性的值就是环境变量 bootargs 的内容。
bootz 命令执行流程：

<h2 id="六-内核解析-DTB-文件">六、内核解析 DTB 文件</h2>
Linux 内核在启动的时候会解析 DTB 文件，然后在/proc/device-tree 目录下生成相应的设备树点文件。
Linux 内核解析 DTB 文件的流程：

<h2 id="七-设备树常用-OF-操作函数">七、设备树常用 OF 操作函数</h2>
设备树描述了设备的详细信息，这些信息包括数字类型的、字符串类型的、数组类型的，我们在写驱动的时候需要获取到这些信息。比如设备树使用 reg 属性描述了某个外设的寄存器地址为 0X02005482，长度为 0X400，我们在编写驱动的时候需要获取到 reg 属性的 0X02005482 和 0X400 这两值，然后初始化外设。 Linux 内核给我们提供了一系列的函数来获取设备树中的节点或者属性信息，一系列的函数都有一个统一的前缀“of_”，所以在很多资料里面也被叫做 OF 函数。这些 OF 函数原都定义在 include/linux/of.h 文件中。
<h3 id="1-查找节点的-OF-函数">1.查找节点的 OF 函数</h3>
设备都是以节点的形式“挂”到设备树上的，因此要想获取这个设备的其他属性信息，必须先获到这个设备的节点。 Linux 内核使用 device_node 结构体来描述一个节点，此结构体定<br>义在文件 include/linux/of.h 中，定义如下：
<pre><code class="highlight-chroma">struct device_node {
    const char *name;
    const char *type;
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;</code></pre>

    struct    propert
 *properties;
    struct    propert
 *deadprops;    /* removed properties */
    struct    device_
ode *parent;
    struct    device_
ode *child;
    struct    device_
ode *sibling;
    struct    kobject
kobj;
    unsigned long _
lags;
    void    *data;

#if defined(CONF
G_SPARC)
    const char *pat
_component_name;
    unsigned int un
que_id;
    struct of_irq_co
troller *irq_trans;
#endif
};
</code></pre>
<h4 id="-1-of-find-node-by-name-函数">(1)of_find_node_by_name 函数</h4>
of_find_node_by_name 函数通过节点名字查找指定的节点，函数原型如下：
struct device_node *of_find_node_by_name(struct device_node *from, const char *name);
br>
from：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。<b
>
name：要查找的节点名字。<br>
返回值： 找到的节点，如果为 NULL 表示查找失败。
<h4 id="-2-of-find-node-by-type-函数">(2)of_find_node_by_type 函数</h4>
of_find_node_by_type 函数通过 device_type 属性查找指定的节点，函数原型如下：
struct device_node *of_find_node_by_type(struct device_node *from, const char *type)</
>
from：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树
<br>
type：要查找的节点对应的 type 字符串，也就是 device_type 属性值。<br>
返回值： 找到的节点，如果为 NULL 表示查找失败。
<h4 id="-3-of-find-compatible-node-函数">(3)of_find_compatible_node 函数</h4>
of_find_compatible_node 函数根据 device_type 和 compatible 这两个属性查找指定的节点，
br>
函数原型如下：
struct device_node *of_find_compatible_node(struct device_node *from, const char *type,
const char *compatible)
from：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树
<br>
type：要查找的节点对应的 type 字符串，也就是 device_type 属性值，可以为
NULL，表示忽略掉 device_type 属性。<br>

compatible： 要查找的节点所对应的 compatible 属性列表。<br>
返回值： 找到的节点，如果为 NULL 表示查找失败
<h4 id="-4-of-find-matching-node-and-match-函数">(4)of_find_matching_node_and_match 函数</h4>
of_find_matching_node_and_match 函数通过 of_device_id 匹配表来查找指定的节点，函数型如下：
struct device_node *of_find_matching_node_and_match(struct device_node *from, const struct of_device_id *matches, const struct of_device_id **match)
from：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树<br>
matches： of_device_id 匹配表，也就是在此匹配表里面查找节点。<br>
match： 找到的匹配的 of_device_id。<br>
返回值： 找到的节点，如果为 NULL 表示查找失败
<h4 id="-5-of-find-node-by-path-函数">(5)of_find_node_by_path 函数</h4>
of_find_node_by_path 函数通过路径来查找指定的节点，函数原型如下：
inline struct device_node *of_find_node_by_path(const char *path)<br>
path：带有全路径的节点名，可以使用节点的别名，比如“/backlight”就是 backlight 这个节点的全路径。<br>
返回值： 找到的节点，如果为 NULL 表示查找失败
<h3 id="2-查找父-子节点的-OF-函数">2.查找父/子节点的 OF 函数</h3>
<h4 id="-1-of-get-parent-函数">(1)of_get_parent 函数</h4>
of_get_parent 函数用于获取指定节点的父节点(如果有父节点的话)，函数原型如下：
struct device_node *of_get_parent(const struct device_node *node)<br>
node：要查找的父节点的节点。<br>
返回值： 找到的父节点。
<h4 id="-2-of-get-next-child-函数">(2)of_get_next_child 函数</h4>
of_get_next_child 函数用迭代的查找子节点，函数原型如下：
struct device_node *of_get_next_child(const struct device_node *node, struct device_node *prev)<br>
函数参数和返回值含义如下：<br>
node：父节点。<br>
prev：前一个子节点，也就是从哪一个子节点开始迭代的查找下一个子节点。以设置为 NULL，表示从第一个子节点开始。<br>
返回值： 找到的下一个子节点。
<h3 id="3--提取属性值的-OF-函数">3. 提取属性值的 OF 函数</h3>
节点的属性信息里面保存了驱动所需要的内容，因此对于属性值的提取非常重要， Linux 内<br>核中使用结构体 property 表示属性，此结构体同样定义在文件 include/linux/of.h 中，内容如下：<br>
property 结构体
<pre><code class="highlight-chroma">struct property {
    char *name; /* 属性名字 */
    int length; /* 性长度 */
    void *value; /* 性值 */
    struct property next; /* 下一个属性 */
    unsigned long _flags;
    unsigned int unique_id;
    struct bin_attri

ute attr;
};
</code></pre>
<h4 id="-1-of-find-property-函数">(1)of_find_property 函数</h4>
of_find_property 函数用于查找指定的属性，函数原型如下：
property *of_find_property(const struct device_node *np, const char *name, int *lenp)<b
>
np：设备节点。<br>
name： 属性名字。<br>
lenp：属性值的字节数<br>
返回值： 找到的属性。
<h4 id="-2-of-property-count-elems-of-size-函数">(2)of_property_count_elems_of_size 函数<
h4>
of_property_count_elems_of_size 函数用于获取属性中元素的数量，比如 reg 属性值是一个<b
>
数组，那么使用此函数可以获取到这个数组的大小，此函数原型如下：
int of_property_count_elems_of_size(const struct device_node *np, const char *propname,
int elem_size)<br>
np：设备节点。<br>
proname： 需要统计元素数量的属性名字。<br>
elem_size：元素长度。<br>
返回值： 得到的属性元素数量。
<h4 id="-3-of-property-read-u32-index-函数">(3)of_property_read_u32_index 函数</h4>
of_property_read_u32_index 函数用于从属性中获取指定标号的 u32 类型数据值(无符号 32<b
>
位)，比如某个属性有多个 u32 类型的值，那么就可以使用此函数来获取指定标号的数据值，此<br>
函数原型如下：
int of_property_read_u32_index(const struct device_node *np, const char *propname, u32
ndex, u32 *out_value)<br>
函数参数和返回值含义如下：<br>
np：设备节点。<br>
proname： 要读取的属性名字。<br>
index：要读取的值标号。<br>
out_value：读取到的值<br>
返回值： 0 读取成功，负值，读取失败， -EINVAL 表示属性不存在， -ENODA
A 表示没有要读取的数据，-EOVERFLOW 表示属性值列表太小。
<h4 id="-4-of-property-read-u8-array-函数">(4)of_property_read_u8_array 函数</h4>
of_property_read_u16_array 函数<br>
of_property_read_u32_array 函数<br>
of_property_read_u64_array 函数<br>
这 4 个函数分别是读取属性中 u8、 u16、 u32 和 u64 类型的数组数据，比如大多数的 reg 属性都
数组数据，可以使用这 4 个函数一次读取出 reg 属性中的所有数据。这四个函数的原型如下：
int of_property_read_u8_array(const struct device_node *np, const char *propname, u8 *
ut_values, size_t sz)<br>
int of_property_read_u16_array(const struct device_node *np, const char *propname, u16 *out
values, size_t sz)<br>
int of_property_read_u32_array(const struct device_node *np, const char *propname, u32 *out
values, size_t sz)<br>
int of_property_read_u64_array(const struct device_node *np, const char *propname, u64 *out
values, size_t sz)<br>
np：设备节点。<br>
proname： 要读取的属性名字。<br>
out_value：读取到的数组值，分别为 u8、 u16、 u32 和 u64。<br>
sz： 要读取的数组元素数量。<br>

返回值： 0，读取成功，负值，读取失败， -EINVAL 表示属性不存在， -ENO
ATA 表示没<br>
有要读取的数据， -EOVERFLOW 表示属性值列表太小。
<h4 id="-5-of-property-read-u8-函数">(5)of_property_read_u8 函数</h4>
of_property_read_u16 函数<br>
of_property_read_u32 函数<br>
of_property_read_u64 函数<br>
有些属性只有一个整形值，这四个函数就是用于读取这种只有一个整形值的属性，分别用于读取 u8、
u16、 u32 和 u64 类型属性值，函数原型如下：
int of_property_read_u8(const struct device_node *np, const char *propname, u8 *out_val
e)<br>
int of_property_read_u16(const struct device_node *np, const char *propname, u16 *out_value
<br>
int of_property_read_u32(const struct device_node *np, const char *propname, u32 *out_value
<br>
int of_property_read_u64(const struct device_node *np, const char *propname, u64 *out_value
<br>
np：设备节点。<br>
proname： 要读取的属性名字。<br>
out_value：读取到的数组值。<br>
返回值： 0，读取成功，负值，读取失败， -EINVAL 表示属性不存在， -ENO
ATA 表示没有要读取的数据， -EOVERFLOW 表示属性值列表太小。
<h4 id="-6-of-property-read-string-函数">(6)of_property_read_string 函数</h4>
of_property_read_string 函数用于读取属性中字符串值，函数原型如下：
int of_property_read_string(struct device_node *np, const char *propname, const char **
ut_string)<br>
np：设备节点。<br>
proname： 要读取的属性名字。<br>
out_string：读取到的字符串值。<br>
返回值： 0，读取成功，负值，读取失败。
<h4 id="-8--of-n-addr-cells-函数">(8).of_n_addr_cells 函数</h4>
of_n_addr_cells 函数用于获取#address-cells 属性值，函数原型如下：<br>
int of_n_addr_cells(struct device_node *np)<br>
np：设备节点。<br>
返回值： 获取到的#address-cells 属性值。
<h4 id="-9--of-n-size-cells-函数">(9).of_n_size_cells 函数</h4>
of_size_cells 函数用于获取#size-cells 属性值，函数原型如下：<br>
int of_n_size_cells(struct device_node *np)<br>
np：设备节点。<br>
返回值： 获取到的#size-cells 属性值。
<h3 id="4-其他常用的-OF-函数">4 其他常用的 OF 函数</h3>
<h4 id="-1-of-device-is-compatible-函数">(1)of_device_is_compatible 函数</h4>
of_device_is_compatible 函数用于查看节点的 compatible 属性是否有包含 compat 指定的字
串，也就是检查设备节点的兼容性，函数原型如下：
int of_device_is_compatible(const struct device_node *device, const char *compat)<br>
device：设备节点。<br>
compat：要查看的字符串。<br>
返回值： 0，节点的 compatible 属性中不包含 compat 指定的字符串； 正数
节点的 compatible 属性中包含 compat 指定的字符串。
<h4 id="-2-of-get-address-函数">(2)of_get_address 函数</h4>
of_get_address 函数用于获取地址相关属性，主要是“reg”或者“assigned-addresses”属
值，函数属性如下：
const __be32 *of_get_address(struct device_node *dev,int index, u64 *size, unsigned int *f
ags)<br>

dev：设备节点。<br>
index：要读取的地址标号。<br>
size：地址长度。<br>
flags：参数，比如 IORESOURCE_IO、 IORESOURCE_MEM 等<br>
返回值： 读取到的地址数据首地址，为 NULL 的话表示读取失败。
<h4 id="-3-of-translate-address-函数">(3)of_translate_address 函数</h4>
of_translate_address 函数负责将从设备树读取到的地址转换为物理地址，函数原型如下：
u64 of_translate_address(struct device_node *dev, const __be32 *in_addr)<br>
dev：设备节点。<br>
in_addr：要转换的地址。<br>
返回值： 得到的物理地址，如果为 OF_BAD_ADDR 的话表示转换失败。
<h4 id="-4-of-address-to-resource-函数">(4)of_address_to_resource 函数</h4>
IIC、 SPI、 GPIO 等这些外设都有对应的寄存器，这些寄存器其实就是一组内存空间， Linux 内使用 resource 结构体来描述一段内存空间， “resource”翻译出来就是“资源”，因此用 resource 结构体描述的都是设备资源信息， resource 结构体定义在文件 include/linux/ioport.h 中，定义如：<br>
resource 结构体<br>
18 struct resource {<br>
19 resource_size_t start;<br>
20 resource_size_t end;<br>
21 const char *name;<br>
22 unsigned long flags;<br>
23 struct resource *parent, *sibling, *child;<br>
24 };<br>
对于 32 位的 SOC 来说，resource_size_t 是 u32 类型的。其中 start 表示开始地址，end 表示结束址， name 是这个资源的名字， flags 是资源标志位，一般表示资源类型，可选的资源标志<br>
定义在文件 include/linux/ioport.h 中，如下所示：<br>
资源标志<br>
1 #define IORESOURCE_BITS 0x000000ff<br>
2 #define IORESOURCE_TYPE_BITS 0x00001f00<br>
3 #define IORESOURCE_IO 0x00000100<br>
4 #define IORESOURCE_MEM 0x00000200<br>
5 #define IORESOURCE_REG 0x00000300<br>
6 #define IORESOURCE_IRQ 0x00000400<br>
7 #define IORESOURCE_DMA 0x00000800<br>
8 #define IORESOURCE_BUS 0x00001000<br>
9 #define IORESOURCE_PREFETCH 0x00002000<br>
10 #define IORESOURCE_READONLY 0x00004000<br>
11 #define IORESOURCE_CACHEABLE 0x00008000<br>
12 #define IORESOURCE_RANGELENGTH 0x00010000<br>
13 #define IORESOURCE_SHADOWABLE 0x00020000<br>
14 #define IORESOURCE_SIZEALIGN 0x00040000<br>
15 #define IORESOURCE_STARTALIGN 0x00080000<br>
16 #define IORESOURCE_MEM_64 0x00100000<br>
17 #define IORESOURCE_WINDOW 0x00200000<br>
18 #define IORESOURCE_MUXED 0x00400000<br>
19 #define IORESOURCE_EXCLUSIVE 0x08000000<br>
20 #define IORESOURCE_DISABLED 0x10000000<br>
21 #define IORESOURCE_UNSET 0x20000000<br>
22 #define IORESOURCE_AUTO 0x40000000<br>
23 #define IORESOURCE_BUSY 0x80000000<br>
大家一般最常见的资源标志就是 IORESOURCE_MEM 、 IORESOURCE_REG 和<br>
IORESOURCE_IRQ 等。接下来我们回到 of_address_to_resource 函数，此函数看名字像是从设<br>

备树里面提取资源值，但是本质上就是将 reg 属性值，然后将其转换为 resource 结构体类型，<br>
函数原型如下所示
int of_address_to_resource(struct device_node *dev, int index, struct resource *r)<br>
函数参数和返回值含义如下：<br>
dev：设备节点。<br>
index：地址资源标号。<br>
r：得到的 resource 类型的资源值。<br>
返回值： 0，成功；负值，失败。
<h4 id="-5-of-iomap-函数">(5)of_iomap 函数</h4>
of_iomap 函数用于直接内存映射，以前我们会通过 ioremap 函数来完成物理地址到虚拟地<br>
址的映射，采用设备树以后就可以直接通过 of_iomap 函数来获取内存地址所对应的虚拟地址，<br>
不需要使用 ioremap 函数了。当然了，你也可以使用 ioremap 函数来完成物理地址到虚拟地址<br>
的内存映射，只是在采用设备树以后，大部分的驱动都使用 of_iomap 函数了。 of_iomap 函数本<b
>
质上也是将 reg 属性中地址信息转换为虚拟地址，如果 reg 属性有多段的话，可以通过 index 参<br

数指定要完成内存映射的是哪一段， of_iomap 函数原型如下：<br>
void __iomem *of_iomap(struct device_node *np, int index)<br>
函数参数和返回值含义如下：<br>
np：设备节点。<br>
index： reg 属性中要完成内存映射的段，如果 reg 属性只有一段的话 index
设置为 0。<br>
返回值： 经过内存映射后的虚拟内存首地址，如果为 NULL 的话表示内存映射
败。
关于设备树我们重点要了解一下几点内容：<br>
①、DTS、 DTB 和 DTC 之间的区别，如何将.dts 文件编译为.dtb 文件。<br>
②、设备树语法，这个是重点，因为在实际工作中我们是需要修改设备树的。<br>
③、设备树的几个特殊子节点。<br>
④、关于设备树的 OF 操作函数，也是重点，因为设备树最终是被驱动文件所使用的，而驱动文件必
要读取设备树中的属性信息，比如内存信息、 GPIO 信息、中断信息等等。要想在驱动中读取设备树
属性值，那么就必须使用 Linux 内核提供的众多的 OF 函数。