Linux设备树的用法 - green1900的专栏 - 博客频道

分类：
DTS（4）

本文通过为一个新machine写一个设备树来介绍设备树相关的概念，以及如何来描述一个machine。
关于设备树的技术细节描述，需要参考ePAPR文档，ePAPR文档中包含了大量的基础语法之外的细节，如果你需要了解更多本文之外的设备树细节，请参考ePAPR文档。
基本数据格式
设备树是一个由节点及属性组成的简单树结构。属性是基于key-value对的，节点则可以包含子节点以及属性。
如，下面这个树就是一个典型结构：

```
/ {
      node1 {
            a-string-property = "A string";
            a-string-list-property = "first string", "second string";
            a-byte-data-property = [0x01 0x23 0x34 0x56];
            child-node1 {
                  first-child-property;
                  second-child-property = <1>;
                  a-string-property = "Hello, world";
            };
            child-node2 {
            };
      };
      node2 {
            an-empty-property;
            a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
            child-node1 {
            };
      };
};
```

这颗树显然没什么实际用途，因为它没有描述任何有意义的内容，但是，我们通过它可以了解什么是属性，什么是节点。这颗树可以解读如下：

- 一个根节点： "/"
- 两个子节点： "node1" 和 "node2"
- node1下面又有两个子节点："child-node1" 和 "child-node2"
- 一堆的属性分散与整颗树的各个节点上

译者注：可以这么简单理解：节点就是树枝，属性就是树叶；树枝上可以有再长树枝也可以长树叶，而树叶上则不会再长树枝。
属性是基于key-value结构的，value可以为空或者特定格式的字符串内容。由于数据类型并不被编码到最终的数据结构中，设备树源代码中仅能支持有限的几种基本数据类型。

- text string（以null结束），以双引号括起来，如：
  - string-property = "a string"
- cells 是32位无符号整形数，以尖括号括起来，如
  - cell-property = <0xbeef 123 0xabcd1234>
- binary data 以方括号括起来，如：
  - binary-property = [0x01 0x23 0x45 0x67];
- 不同类型数据可以在同一个属性中存在，以逗号分格，如：
  - mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;
- 多个字符串组成的列表也使用逗号分格，如：
  - string-list = "red fish","blue fish";

基本概念
为了搞清楚设备树该如何使用，我们将一步一步的为一个Sample Machine建立一颗设备树。
Sample Machine
姑且想象有这么一台机器，它由"Acme"出产，名为"Coyote's Revenge"，配置如下:

- 32bit ARM CPU
- 处理器的本地内存总线连接如下设备：串口，SPI总线控制器，I2C控制器，终端控制器以及外部总线桥
- 256MB SDRAM 位于地址：0x0

- 两个串口，位于地址：0x101F1000 及 0x101F2000
- GPIO控制器位于地址：0x101F3000
- SPI控制器位于地址：0x10170000，在它下面连接了如下设备：
  - MMC卡槽，其SS脚连接了GPIO1

- 外部总线桥上接了如下设备：
  - SMC91111 以太网控制器，位于地址：0x10100000
  - I2C控制器，位于地址：0x10160000，在它下面连接了如下设备：
    - Maxim DS1338实时时钟，I2C地址为0x58

  - 64MB NOR flash 位于地址：0x30000000

初始结构
第一步是要建立一个基本结构来使得这颗设备树能描述对应的Machine

```
/ {
      compatible = "acme,coyotes-revenge";
};
```

compatible这个属性用于执行系统名，通常它是以 "厂商,型号" 这样的字符串形式存在。它能准确的描述对应的设备的特征。
CPU描述
下一部就是描述每一个CPU了。在cpus节点中，每个CPU就是一个子节点。这种情况在多核的ARM Cortex A9系统中很常见。

```
/ {
      compatible = "acme,coyotes-revenge";
      cpus {
            cpu@0 {
                  compatible = "arm,cortex-a9";
            };
            cpu@1 {
                  compatible = "arm,cortex-a9";
            };
      };
};
```

每个cpu的子节点中有个compatible属性，它描述了CPU的具体型号，形式通常也是 "厂商，型号"
当然，CPU的细节不是这么一个compatible属性能描述清楚的，后面我们会再逐步加入。
节点名
有必要为节点的命名讲几句。每个节点的名字都应该是这样的形式： <name>[@<unit-address>]
尖括号是必须的，方括号是可选的。
<name> 是一个ASCII字符串，长度不超过31个字符。通常，节点名都是与设备的类型有关联的。比如，3com的以太网卡，通常节点名就叫 ethernet 而不叫 3com509
<unit-address>被作为节点名的一部分时，用来描述设备的地址。通常，unit-address就是设备的寄存器地址，这个地址是被列举在节点的reg属性中的。在后文中我们会介绍reg属性的内容。
同级别的兄弟节点的节点名必须唯一（不可重名），但如果name一致而address不一致则是正常情况（比如，serial@101f1000 与 serial@101f2000）。
关于节点的命名规则细节，请参考ePAPR文档的2.2.1节。
设备
系统中的每个设备都对应着设备树中的一个节点。好了，下一步我们就是为每个设备都加上对应的节点。
目前，我们仅为每个设备增加一个空节点，待后面介绍了中断号及地址范围的概念后再行补充。

```
/ {
      compatible = "acme,coyotes-revenge";
      cpus {
            cpu@0 {
                  compatible = "arm,cortex-a9";
            };
            cpu@1 {
                  compatible = "arm,cortex-a9";
            };
      };
      serial@101F0000 {            compatible = "arm,pl011";      };      serial@101F2000 {            compatible = "arm,pl011";
};      gpio@101F3000 {            compatible = "arm,pl061";      };      interrupt-controller@10140000 {            compatible =
"arm,pl190";      };      spi@10115000 {            compatible = "arm,pl022";      };      external-bus {            ethernet@0,0 {
                compatible = "smc,smc91c111";            };            i2c@1,0 {                  compatible = "acme,a1234-i2c-bus";
                rtc@58 {                        compatible = "maxim,ds1338";                  };            };            flash@2,0 {
                compatible = "samsung,k8f1315ebm", "cfi-flash";            };      };
};
```

上面这棵树中，系统中的每个设备都被添加了一个节点，而且节点的结构真实反应了设备是如何挂载在系统上的。比如，ethernet，i2c等节点是external-bus的子节点，rtc设备是i2c总线下的子节点。通常，设备树的结构都是以CPU的视角来反应出来的。
上面这棵树还有几点不足，它缺少了设备的关键信息。这些信息将在后文中逐步添加上去。
关于上面这颗树，我们还需要注意：

- 每个设备节点都都一个compatible属性
- flash这个节点的compatible属性有两个字串，下面一节将介绍为什么这么写。
- 在前文中曾提到：节点名反应的是设备类型而非设备型号。请参考ePAPR的2.2.2节中列出的常用节点名。

理解compatible属性
每个设备节点都需要一个compatible属性。compatible属性是系统赖以查找对应的设备驱动程序的一个关键值，系统就是根据它的值来查找这个设备应该使用哪一个驱动的。
compatible属性的值是一个字串表。第一个字串以"厂商,型号"的形式描述了准确的设备信息。后面一个字串则表示与它兼容的其他设备。
比如，Freescale MPC8349有一个串口设备是由National Semiconductor ns16550寄存器接口来实现的。所以对MPC8349的串口设备，它的compatible属性我们就可以这样写：compatible = "fsl,mpc8349-uart", "ns16550"，对于这样的情况，fsl,mpc8349-uart准确描述了这个设备，而ns16550则说明了这个设备是与National Semiconductor ns16550寄存器接口兼容的。
注：ns16550它没有厂商名这个信息，这是由于历史原因。但所有新创建的compatible属性都应该有厂商名这个前缀。
compatible属性的这一特性，使得我们可以让新设备使用系统中已有的旧驱动。
警告：不要在compatible属性中使用通配符，如 "fsl,mpc83xx-uart" 或类似的。因为半导体厂商会不定期的更新他们的设计这会破坏你的通配符规则。选择具有更好兼容性的方案才是正途。
地址是怎么工作的
可寻址设备是通过使用以下属性来将地址信息编码到设备树中的：

- reg
- #address-cells
- #size-cells

可寻址设备通过reg属性来获取寄存器相关的地址信息列表，reg属性的形式如下：
reg = <address1 length1 [address2 length2] [address3 length3] ... >
每一组address length对应了设备所使用的一个地址区域。
address是一个list，其中包含一个或多个32位整数，我们把它叫做 cells。同理，length也是一个list，可以是多个cell或为空。
由于address和length的长度都是不固定的，所以有了#address-cells和#size-cells这两个属性。这两个属性被放到父节点中用于描述每个区域有几个cell。简单的说，就是reg属性需要配合父节点的#address-cells和#size-cells来配合使用。为了弄明白它们是怎么工作的，下面我们就来为这个设备加上地址相关的属性，先从CPU开始。
CPU地址
CPU节点的地址用法是最简单的。每个CPU被分配了唯一的ID号，而且这个没有size。

```
	cpus {
		#address-cells = <1>;			#size-cells = <0>;
		cpu@0 {
			compatible = "arm,cortex-a9";
			reg = <0>;
		};
		cpu@1 {
			compatible = "arm,cortex-a9";
			reg = <1>;
		};
	};
```

在上面的cpus节点中，#address-cells被设为了1，#size-cells则被设为了0，这就表示它的子节点中的reg属性是一个32位整数的address，而且没有size部分。
在上面的例子中，你可能注意到了节点名中的寄存器地址与reg数值一样。
约定俗成的做法是，如果一个节点有reg属性，则节点名中也需要包含unit-address这个部分，而unit-address的数值则是reg属性的第一个address值。
内存映射设备
与cpu节点中的这种单地址不同，内存映射设备所分配的是一个地址范围，而这个地址范围则是由#size-cells和节点中的reg属性的size区域来决定的。下面这个例子中，每个address是1个cell（32bit），且每个长度值也是一个cell。在32位系统中#size-cells通常就是这样设置为1的。而早64位系统中，#address-cells和#size-cells则通常设置为2。

```
/ {
	#address-cells = <1>;		#size-cells = <1>;
	...
	serial@101f0000 {
		compatible = "arm,pl011";
		 reg = <0x101f0000 0x1000 >;
	};
	serial@101f2000 {
		compatible = "arm,pl011";
```

```
            reg = <0x101f2000 0x1000 >;
        };
        gpio@101f3000 {
            compatible = "arm,pl061";
            reg = <0x101f3000 0x1000
                   0x101f4000 0x0010>;
        };
        interrupt-controller@10140000 {
            compatible = "arm,pl190";
            reg = <0x10140000 0x1000 >;
        };
        spi@10115000 {
            compatible = "arm,pl022";
            reg = <0x10115000 0x1000 >;
        };
        ...
};
```

每个设备被分配了一个基地址以及一个size。上面的例子中，gpio设备被分配了两个地址段： 0x101f3000~0x1013fff 以及 0x101f4000~0x101f4fff。

有些设备在系统总线上的地址不连续。比如，一个设备可能通过不连续的片选线连接在外部总线上。

通过在父节点设置合适的#address-cells和#size-cells，地址映射机制可以准确的描述内存映射关系。下面的代码中展示了一个不同片选信息在是如何使用的。

```
    external-bus {
        #address-cells = <2>
        #size-cells = <1>;
        ethernet@0,0 {
            compatible = "smc,smc91c111";
            reg = <0 0 0x1000>;
        };
        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            reg = <1 0 0x1000>;
            rtc@58 {
                compatible = "maxim,ds1338";
            };
        };
        flash@2,0 {
            compatible = "samsung,k8f1315ebm", "cfi-flash";
            reg = <2 0 0x4000000>;
        };
    };
```

上面的例子中，external-bus使用了两个cell来描述address；一个表示片选号，另一个表示与片选基地址间的偏移量。length区域则用了一个cell来描述。在这个例子中，每个reg节点包含3个cell，分别是：片选号，偏移量，长度

非内存映射设备

还有一些设备在总线上并不是不是内存映射型的。他们可以有地址空间，但他们没有通过CPU直接访问地址空间的能力。取而代之的是他们的父设备驱动拥有间接访问内存空间的能力。

以I2C设备（不是I2C总线哦）为例子，每个设备被分配了一个地址，没有length这个字段。看起来实际上与cpu的地址分配很相似。

```
        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <1 0 0x1000>;
            rtc@58 {
                compatible = "maxim,ds1338";
                reg = <58>;
            };
        };
```

Ranges（地址变换）

上文已经讨论过了我们如何分配地址给设备，但是这个所谓的地址仅仅是在设备节点中的本地地址。它无法描述该如何将这些本地的地址映射到CPU能访问的地址空间中。

根节点中一定有描述CPU的地址空间。子节点所用的地址空间就来自与CPU的地址空间，所以不需要进行额外的地址映射。

如：serial@101f0000 就是直接分配的地址0x101f0000。

如果一个节点它不是跟节点下的子节点，那么它就不能用CPU的地址空间。为了能将一个地址空间的地址映射到另一个地址空间，range属性被创造出来了。
下面是在一个简单的设备树中增加range属性。

```
/ {
      compatible = "acme,coyotes-revenge";
      #address-cells = <1>;
      #size-cells = <1>;
      ...
      external-bus {
            #address-cells = <2>
            #size-cells = <1>;
            ranges = <0 0    0x10100000    0x10000       // Chipselect 1, Ethernet                              1 0    0x10160000    0x10000
      // Chipselect 2, i2c controller                              2 0    0x30000000    0x1000000>; // Chipselect 3, NOR Flash
            ethernet@0,0 {
                  compatible = "smc,smc91c111";
                  reg = <0 0 0x1000>;
            };
            i2c@1,0 {
                  compatible = "acme,a1234-i2c-bus";
                  #address-cells = <1>;
                  #size-cells = <0>;
                  reg = <1 0 0x1000>;
                  rtc@58 {
                        compatible = "maxim,ds1338";
                        reg = <58>;
                  };
            };
            flash@2,0 {
                  compatible = "samsung,k8f1315ebm", "cfi-flash";
                  reg = <2 0 0x4000000>;
            };
      };
};
```

上面的例子中，ranges属性就定义了一个地址转换规格。在这个表中的每个节点表示一个地址转换关系。
ranges属性中每个字段的大小取决于当前节点的#address-cells，父节点的#address-cells以及当前节点的#size-cells。
比如上面的例子中，external-bus节点的地址长度是2，它的父节点的地址长度是1，size长度是1。所以ranges中的三个地址规则可以这样解读：

- CS0，偏移量为0的本地地址被映射到父节点地址空间的 0x10100000~0x1010ffff
- CS1，偏移量为1的本地地址被映射到父节点地址空间的 0x10160000~0x1016ffff
- CS2，偏移量为0的本地地址被映射到父节点地址空间的 0x30000000~0x31000000

更方便的是，如果父节点与子节点的地址空间完全匹配，则子节点可以只定义一个空的ranges属性。
空ranges属性所表示的意思就是子节点的地址空间与父节点地址空间是1：1的映射关系。
你可能会问：为什么上面会写1：1的映射关系？有些总线结构（如PCI总线）拥有自己独立的地址空间，但需要向操作系统开放。还有些设备有DMA引擎，这就需要知道总线上的实际地址。还有些时候几个设备因为使用同样的物理地址空间而需要组合在一起。在硬件设计以及操作系统的大量特性上决定了地址映射是不是1：1的映射关系。
也可能还注意到了上文中的i2c@1,0节点中没有ranges属性。原因是I2C总线不像外部总线，它的地址空间并没有映射到CPU的地址空间中去。实际上，CPU对rtc@58的访问是通过i2c@1,0这个设备来间接达成的。没有ranges属性正表明了这个设备是不能直接访问除父节点外的任何设备的特性。
中断是怎样工作的
不想地址空间映射表那样是遵循设备树的自然结构的（父传子），中断信号可以被machine中的任何设备产生或终止。终端信号独立与设备树将各个节点关联起来。描述一个中断需要4个属性：

- interrupt-controller 这个属性没有值，他表示这个节点是一个接收中断信号的设备
- #interrupt-cells 这个属性是一个interrupt-controller节点的属性，他说明了这个 interrupt-controller的每个中断说明符（interrupt specifier）有几个cells，类似#address-cells 与 #size-cells的作用
- interrupt-parent 这是一个设备节点的属性，用于表明当前设备的中断是属于哪一个interrupt-controller的，如果没有这个属性，则继承其父节点的interrupt-parent属性
- interrupts 这是一个设备节点的属性，他是 中断说明符 列表，每一个 中断说明符 表示此设备的一个中断信号输出。

中断说明符是由一个或多个cell数据（#interrupt-cells ）来描述一个设备是与哪一个终端信号输入设备相连接的。多数设备都只有一个中断信号输出，如下面的例子，但也有可能存在一个设备有多个终端信号输出的情况。中断说明符的含义与具体的终端控制器（ interrupt-controller）有关。每个终端控制器都可以决定它的输入信号的中断说明符有几个cell数据。
下面的代码中为我们的Coyote's Revenge添加了中断连接：

```
/ {
```

```
compatible = "acme,coyotes-revenge";
#address-cells = <1>;
#size-cells = <1>;
 interrupt-parent = <&intc>;
cpus {
      #address-cells = <1>;
      #size-cells = <0>;
      cpu@0 {
            compatible = "arm,cortex-a9";
            reg = <0>;
      };
      cpu@1 {
            compatible = "arm,cortex-a9";
            reg = <1>;
      };
};
serial@101f0000 {
      compatible = "arm,pl011";
      reg = <0x101f0000 0x1000 >;
       interrupts = < 1 0 >;
};
serial@101f2000 {
      compatible = "arm,pl011";
      reg = <0x101f2000 0x1000 >;
       interrupts = < 2 0 >;
};
gpio@101f3000 {
      compatible = "arm,pl061";
      reg = <0x101f3000 0x1000
                  0x101f4000 0x0010>;
       interrupts = < 3 0 >;
};
 intc:  interrupt-controller@10140000 {
      compatible = "arm,pl190";
      reg = <0x10140000 0x1000 >;
       interrupt-controller;
       #interrupt-cells = <2>;
};
spi@10115000 {
      compatible = "arm,pl022";
      reg = <0x10115000 0x1000 >;
       interrupts = < 4 0 >;
};
external-bus {
      #address-cells = <2>
      #size-cells = <1>;
      ranges = <0 0   0x10100000    0x10000       // Chipselect 1, Ethernet
                  1 0   0x10160000    0x10000       // Chipselect 2, i2c controller
                  2 0   0x30000000    0x1000000>; // Chipselect 3, NOR Flash
      ethernet@0,0 {
            compatible = "smc,smc91c111";
            reg = <0 0 0x1000>;
             interrupts = < 5 2 >;
      };
      i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <1 0 0x1000>;
             interrupts = < 6 2 >;
            rtc@58 {
```

```
                        compatible = "maxim,ds1338";
                        reg = <58>;
                         interrupts = < 7 3 >;
                  };
            };
            flash@2,0 {
                  compatible = "samsung,k8f1315ebm", "cfi-flash";
                  reg = <2 0 0x4000000>;
            };
      };
};
```

有些细节需要各位注意：

- 此machine仅有一个中断控制器：interrupt-controller@10140000
- 标签"intc:"被加到了中断控制器的节点上，这个标签在父节点上创建了一个phandle，这个phandle就是父节点的interrupt-parent。所以这个中断控制器就成了系统所有子节点的默认终端控制器，只有当子节点明确的声明了其interrupt-parent才会被覆盖。
- 每个设备使用interrupt属性来区分不同的终端输入线。
- #interrupt-cells（在interrupt-controller@10140000节点中）的值是2，所以，每个中断说明符由两个cell数据组成。这个例子中用的是最常见的　中断说明符　形式，第一个cell表示中断线的序号，第二个cell表示终端类型的flag（表示高有效，低有效。。。），对于不同的终端控制器，需要阅读对应的binding document来得知其　中断说明符　的格式。

设备特定数据

在常用的属性之外，我们还能为一个节点自行添加属性及子节点。只要是系统需要的任何数据都能被我们按特定的规则来添加。

首先，新设备特定的属性名需要使用　厂商前缀，这样能避免与系统已有的标准属性名称冲突。

第二，每个属性都应该在某个binding文档中有相关的说明，这样驱动作者才能知道如何使用这些属性数据。

第三，将新的binding资料在devicetree-discuss@lists.ozlabs.org中post出来，大家对binding的代码审查将能规避大部分常识性错误。

特殊节点

aliases 节点

这个特殊节点用来引用一个长路径，比如/external-bus/ethernet@0,0，它是长路径的缩写或叫别名。比如：

```
      aliases {
            ethernet0 = &eth0;
            serial0 = &serial0;
      };
```

使用别名来标识一个设备是备受操作系统欢迎的做法。

chosen节点

chosen节点并不代表一个真正的设备，而是用来在Firmware与操作系统间传递数据，如启动参数。

通常chosen节点在dts中被置空。

在我们这个例子中，被添加了如下的启动参数：

```
  chosen {
            bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";
      };
```