# Linux设备树文件结构与解析深度分析

# 1. Device Tree简介

设备树就是描述单板资源以及设备的一种文本文件。至于出现的原因，大家可以上网查询更多关于设备树的文章。本篇文章主要是更深层次的探讨设备文件的构成以及kernel解析设备树的原理。所以，本篇内容并不是针对没有任何设备树知识的读者。本篇文章主要针对已经使用过设备树或者对设备已经有所了解并想深层次的探究设备树的文件结构和kernel解析过程的读者。

# 2. Device Tree编译

Device Tree文件的格式为dts，包含的头文件格式为dtsi，dts文件是一种人可以看懂的编码格式。但是uboot和linux不能直接识别，他们只能识别二进制文件，所以需要把dts文件编译成dtb文件。dtb文件是一种可以被kernel和uboot识别的二进制文件。把dts编译成dtb文件的工具是dtc。Linux源码目录下scripts/dtc目录包含dtc工具的源码。在Linux的scripts/dtc目录下除了提供dtc工具外，也可以自己安装dtc工具，linux下执行：sudo apt-get install device-tree-compiler安装dtc工具。其中还提供了一个fdtdump的工具，可以反编译dtb文件。dts和dtb文件的转换如图1所示。

dtc工具的使用方法是：dtc –I dts –O dtb –oxxx.dtb xxx.dts，即可生成dts文件对应的dtb文件了。

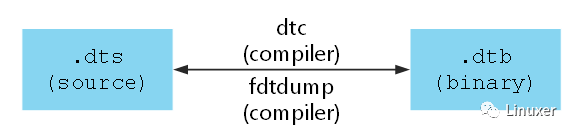


图1 dts和dtb文件转换

# 3. Device Tree头信息

fdtdump工具使用，Linux终端执行ftddump –h，输出以下信息：

fdtdump -h

Usage: fdtdump [options] <file>

Options: -[dshV]

 -d, --debug   Dump debuginformation while decoding the file

 -s, --scan    Scan for an embeddedfdt in file

 -h, --help    Print this help andexit

 -V, --version Print version and exit

本文采用s5pv21\_smc.dtb文件为例说明fdtdump工具的使用。Linux终端执行fdtdump –sd s5pv21\_smc.dtb > s5pv21\_smc.txt，打开s5pv21\_smc.txt文件，部分输出信息如下所示：

// magic:                             0xd00dfeed

// totalsize:                         0xce4 (3300)

// off\_dt\_struct:                   0x38

// off\_dt\_strings:                 0xc34

// off\_mem\_rsvmap:            0x28

// version:                           17

// last\_comp\_version:         16

// boot\_cpuid\_phys:            0x0

// size\_dt\_strings:               0xb0

// size\_dt\_struct:                 0xbfc

以上信息便是Device Tree文件头信息，存储在dtb文件的开头部分。在Linux内核中使用struct fdt\_header结构体描述。struct fdt\_header结构体定义在scripts\dtc\libfdt\fdt.h文件中。

|  |
| --- |
| struct  fdt\_header  {      fdt32\_t  magic;             /\* magic word FDT\_MAGIC \*/      fdt32\_t  totalsize;         /\* total size of DT block \*/      fdt32\_t  off\_dt\_struct;        /\* offset to  structure \*/      fdt32\_t  off\_dt\_strings;       /\* offset to  strings \*/      fdt32\_t  off\_mem\_rsvmap;       /\* offset to  memory reserve map \*/      fdt32\_t  version;               /\* format version \*/      fdt32\_t  last\_comp\_version;    /\* last compatible  version \*/        /\*  version 2 fields below \*/      fdt32\_t  boot\_cpuid\_phys;   /\* Which physical CPU  id we're booting on \*/      /\*  version 3 fields below \*/      fdt32\_t  size\_dt\_strings;   /\* size of the  strings block \*/        /\*  version 17 fields below \*/      fdt32\_t  size\_dt\_struct;       /\* size of the  structure block \*/  }; |

fdtdump工具的输出信息即是以上结构中每一个成员的值，struct fdt\_header结构体包含了Device Tree的私有信息。例如: fdt\_header.magic是fdt的魔数,固定值为0xd00dfeed，fdt\_header.totalsize是fdt文件的大小。使用二进制工具打开s5pv21\_smc.dtb验证。s5pv21\_smc.dtb二进制文件头信息如图2所示。从图2中可以得到Device Tree的文件是以大端模式储存。并且，头部信息和fdtdump的输出信息一致。

C:\Users\LongXi\AppData\Local\Temp\WeChat Files\838b77963799c36c502d6fae68798835.jpg

图2头信息

Device Tree中的节点信息举例如图3所示。

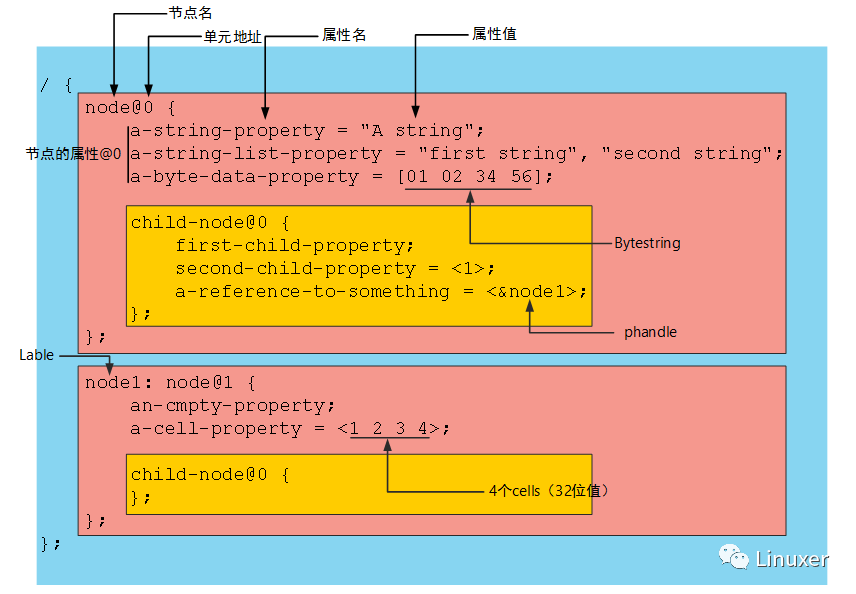


图3设备树全景试图

上述.dts文件并没有什么真实的用途，但它基本表征了一个Device Tree源文件的结构。1个root结点"/"；root结点下面含一系列子结点，本例中为"node@0"和"node@1"；结点"node@0"下又含有一系列子结点，本例中为"child-node@0"；各结点都有一系列属性。这些属性可能为空，如" an-empty-property"；可能为字符串，如"a-string-property"；可能为字符串数组，如"a-string-list-property"；可能为Cells（由u32整数组成），如"second-child-property"，可能为二进制数，如"a-byte-data-property"。Device Tree源文件的结构分为header、fill\_area、dt\_struct及dt\_string四个区域。header为头信息，fill\_area为填充区域，填充数字0，dt\_struct存储节点数值及名称相关信息，dt\_string存储属性名。例如：a-string-property就存储在dt\_string区，"A string"及node1就存储在dt\_struct区域。

我们可以给一个设备节点添加lable，之后可以通过&lable的形式访问这个lable，这种引用是通过phandle（pointer handle）进行的。例如，图3中的node1就是一个lable，node@0的子节点child-node@0通过&node1引用node@1节点。像是这种phandle的节点，在经过DTC工具编译之后，&node1会变成一个特殊的整型数字n，假设n值为1，那么在node@1节点下自动生成两个属性，属性如下：

linux,phandle = <0x00000001>;

phandle = <0x00000001>;

node@0的子节点child-node@0中的a-reference-to-something = <&node1>会变成a-reference-to-something = < 0x00000001>。此处0x00000001就是一个phandle得值，每一个phandle都有一个独一无二的整型值，在后续kernel中通过这个特殊的数字间接找到引用的节点。通过查看fdtdump输出信息以及dtb二进制文件信息，得到struct fdt\_header和文件结构之间的关系信息如所示。

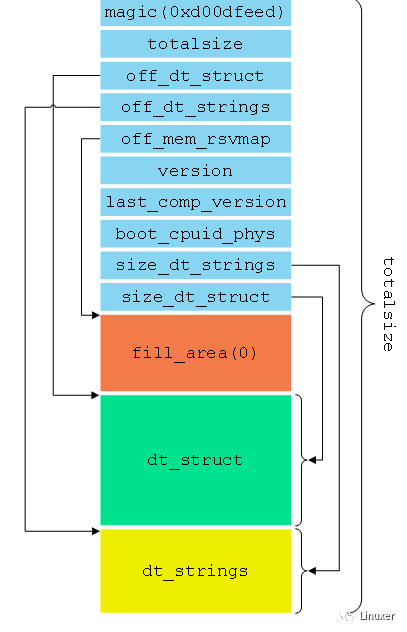


图4 struct fdt\_header和文件结构之间的关系

# 4. Device Tree文件结构

通过以上分析，可以得到Device Tree文件结构如图5所示。dtb的头部首先存放的是fdt\_header的结构体信息，接着是填充区域，填充大小为off\_dt\_struct – sizeof(struct fdt\_header)，填充的值为0。接着就是struct fdt\_property结构体的相关信息。最后是dt\_string部分。

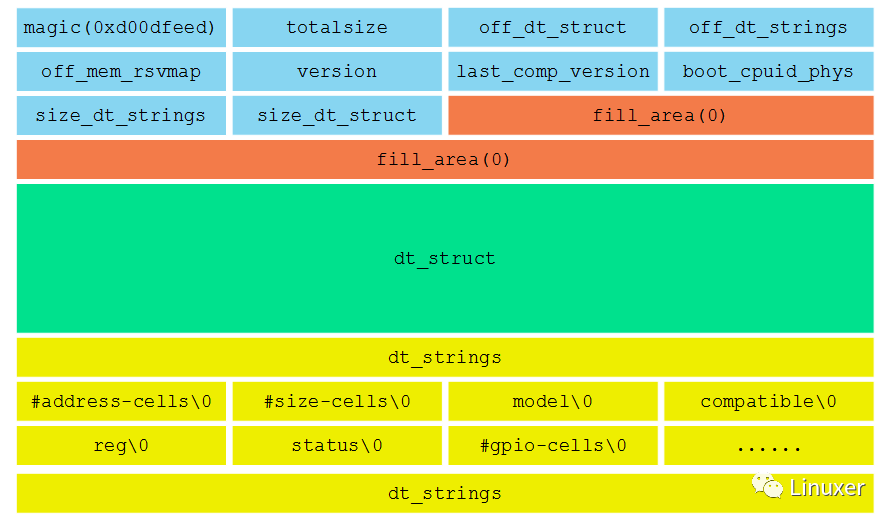


图5 Device Tree文件结构

Device Tree源文件的结构分为header、fill\_area、dt\_struct及dt\_string四个区域。fill\_area区域填充数值0。节点（node）信息使用struct fdt\_node\_header结构体描述。属性信息使用struct fdt\_property结构体描述。各个结构体信息如下:

|  |
| --- |
| struct fdt\_node\_header  {      fdt32\_t  tag;      char  name[0];  };    struct  fdt\_property  {      fdt32\_t  tag;      fdt32\_t  len;      fdt32\_t  nameoff;      char  data[0];  }; |

struct fdt\_node\_header描述节点信息，tag是标识node的起始结束等信息的标志位，name指向node名称的首地址。tag的取值如下：

|  |
| --- |
| #define FDT\_BEGIN\_NODE   0x1    /\*  Start node: full name \*/  #define FDT\_END\_NODE  0x2    /\*  End node \*/  #define FDT\_PROP          0x3    /\*  Property: name off, size, content \*/  #define FDT\_NOP            0x4    /\* nop \*/  #define FDT\_END            0x9 |

FDT\_BEGIN\_NODE和FDT\_END\_NODE标识node节点的起始和结束，FDT\_PROP标识node节点下面的属性起始符，FDT\_END标识Device Tree的结束标识符。因此，对于每个node节点的tag标识符一般为FDT\_BEGIN\_NODE，对于每个node节点下面的属性的tag标识符一般是FDT\_PROP。

描述属性采用struct fdt\_property描述，tag标识是属性，取值为FDT\_PROP；len为属性值的长度（包括‘\0’，单位：字节）；nameoff为属性名称存储位置相对于off\_dt\_strings的偏移地址。

例如：compatible ="samsung,goni", "samsung,s5pv210";compatible是属性名称，"samsung,goni", "samsung,s5pv210"是属性值。compatible属性名称字符串存放的区域是dt\_string。"samsung,goni", "samsung,s5pv210"存放的位置是fdt\_property.data后面。因此fdt\_property.data指向该属性值。fdt\_property.tag的值为属性标识，len为属性值的长度（包括‘\0’，单位：字节）,此处len = 29。nameoff为compatible字符串的位置相对于off\_dt\_strings的偏移地址，即&compatible = nameoff +off\_dt\_strings。

dt\_struct在Device Tree中的结构如图6所示。节点的嵌套也带来tag标识符的嵌套。

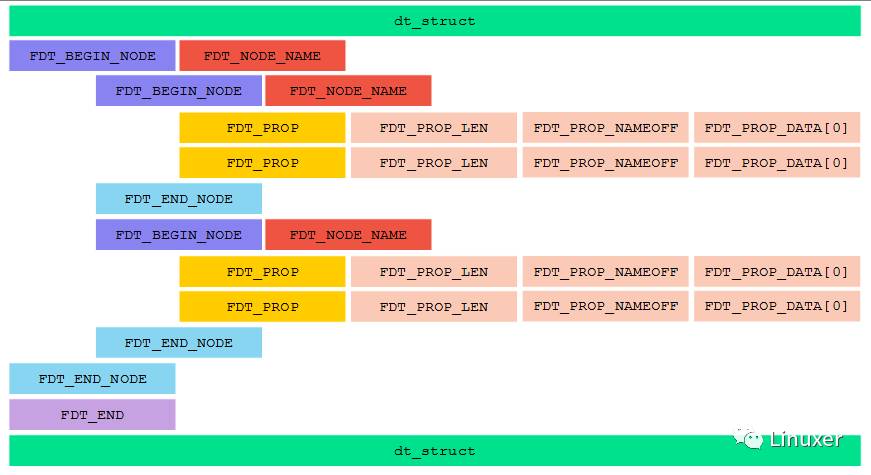


图6 dt\_struct结构图

# 5. kernel解析Device Tree

Device Tree文件结构描述就以上struct fdt\_header、struct fdt\_node\_header及struct fdt\_property三个结构体描述。kernel会根据Device Tree的结构解析出kernel能够使用的struct property结构体。kernel根据Device Tree中所有的属性解析出数据填充struct property结构体。struct property结构体描述如下：

|  |
| --- |
| struct property  {      char  \*name;                          /\* property full name \*/      int  length;                          /\* property  value length \*/      void  \*value;                         /\* property  value \*/      struct  property \*next;             /\* next property  under the same node \*/      unsigned  long \_flags;      unsigned  int unique\_id;      struct  bin\_attribute attr;        /\* 属性文件，与sysfs文件系统挂接 \*/  }; |

总的来说，kernel根据Device Tree的文件结构信息转换成struct property结构体，并将同一个node节点下面的所有属性通过property.next指针进行链接，形成一个单链表。

kernel中究竟是如何解析Device Tree的呢？下面分析函数解析过程。函数调用过程如图7所示。kernel的C语言阶段的入口函数是init/main.c/stsrt\_kernel()函数，在early\_init\_dt\_scan\_nodes()中会做以下三件事：

(1) 扫描/chosen或者/chose@0节点下面的bootargs属性值到boot\_command\_line，此外，还处理initrd相关的property，并保存在initrd\_start和initrd\_end这两个全局变量中；

(2) 扫描根节点下面，获取{size,address}-cells信息，并保存在dt\_root\_size\_cells和dt\_root\_addr\_cells全局变量中；

(3) 扫描具有device\_type = “memory”属性的/memory或者/memory@0节点下面的reg属性值，并把相关信息保存在meminfo中，全局变量meminfo保存了系统内存相关的信息。

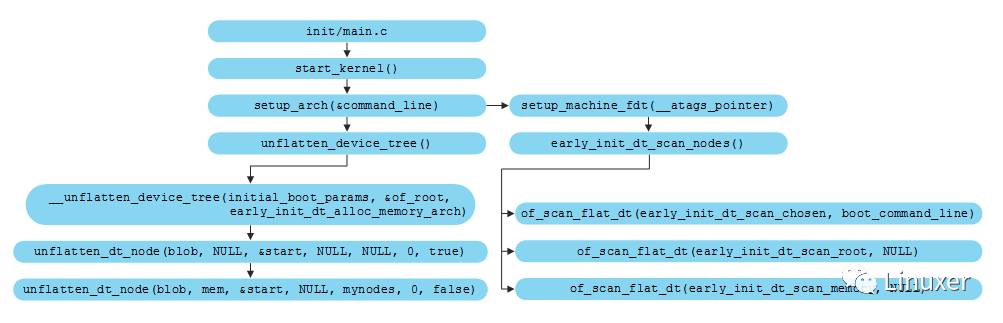


图7函数调用过程

Device Tree中的每一个node节点经过kernel处理都会生成一个struct device\_node的结构体，struct device\_node最终一般会被挂接到具体的struct device结构体。struct device\_node结构体描述如下：

|  |
| --- |
| struct device\_node  {      const  char \*name;              /\* node的名称，取最后一次“/”和“@”之间子串 \*/      const  char \*type;              /\* device\_type的属性名称，没有为<NULL> \*/      phandle  phandle;               /\* phandle属性值 \*/      const  char \*full\_name;        /\* 指向该结构体结束的位置，存放node的路径全名，例如：/chosen \*/      struct  fwnode\_handle fwnode;        struct property \*properties;  /\* 指向该节点下的第一个属性，其他属性与该属性链表相接 \*/      struct property \*deadprops;   /\* removed properties \*/      struct device\_node \*parent;   /\* 父节点 \*/      struct device\_node \*child;    /\* 子节点 \*/      struct device\_node \*sibling;  /\* 姊妹节点，与自己同等级的node \*/      struct kobject kobj;            /\* sysfs文件系统目录体现 \*/      unsigned  long \_flags;          /\* 当前node状态标志位，见/include/linux/of.h line124-127 \*/      void   \*data;  };    /\* flag descriptions (need to be  visible even when !CONFIG\_OF) \*/  #define OF\_DYNAMIC        1 /\* node and properties were  allocated via kmalloc \*/  #define OF\_DETACHED       2  /\* node has been detached from the device tree\*/  #define OF\_POPULATED      3  /\* device already created for the node \*/  #define OF\_POPULATED\_BUS  4 /\* of\_platform\_populate recursed to children of this node \*/ |

struct device\_node结构体中的每个成员作用已经备注了注释信息，下面分析以上信息是如何得来的。Device Tree的解析首先从unflatten\_device\_tree()开始，代码列出如下：

|  |
| --- |
| /\*\*   \* unflatten\_device\_tree - create tree of  device\_nodes from flat blob   \*   \* unflattens the device-tree passed by the  firmware, creating the   \* tree of struct device\_node. It also fills  the "name" and "type"   \* pointers of the nodes so the normal  device-tree walking functions   \* can be used.   \*/  void  \_\_init unflatten\_device\_tree(void)  {      \_\_unflatten\_device\_tree(initial\_boot\_params,  &of\_root,                early\_init\_dt\_alloc\_memory\_arch);        /\*  Get pointer to "/chosen" and "/aliases" nodes for use  everywhere \*/      of\_alias\_scan(early\_init\_dt\_alloc\_memory\_arch);  }    /\*\*   \* \_\_unflatten\_device\_tree - create tree of  device\_nodes from flat blob   \*   \* unflattens a device-tree, creating the   \* tree of struct device\_node. It also fills  the "name" and "type"   \* pointers of the nodes so the normal  device-tree walking functions   \* can be used.   \* @blob: The blob to expand   \* @mynodes: The device\_node tree created by  the call   \* @dt\_alloc: An allocator that provides a  virtual address to memory   \* for the resulting tree   \*/  static void \_\_unflatten\_device\_tree(const  void \*blob,                  struct device\_node \*\*mynodes,                  void \* (\*dt\_alloc)(u64 size, u64 align))  {      unsigned  long size;      int  start;      void  \*mem;         /\* 省略部分不重要部分 \*/      /\*  First pass, scan for size \*/      start  = 0;      size  = (unsigned long)unflatten\_dt\_node(blob, NULL, &start, NULL, NULL, 0,  true);      size  = ALIGN(size, 4);        /\*  Allocate memory for the expanded device tree \*/      mem  = dt\_alloc(size + 4, \_\_alignof\_\_(struct device\_node));      memset(mem,  0, size);        /\*  Second pass, do actual unflattening \*/      start  = 0;      unflatten\_dt\_node(blob,  mem, &start, NULL, mynodes, 0, false);  } |

分析以上代码，在unflatten\_device\_tree()中，调用函数\_\_unflatten\_device\_tree()，参数initial\_boot\_params指向Device Tree在内存中的首地址，of\_root在经过该函数处理之后，会指向根节点，early\_init\_dt\_alloc\_memory\_arch是一个函数指针，为struct device\_node和struct property结构体分配内存的回调函数（callback）。在\_\_unflatten\_device\_tree()函数中，两次调用unflatten\_dt\_node()函数，第一次是为了得到Device Tree转换成struct device\_node和struct property结构体需要分配的内存大小，第二次调用才是具体填充每一个struct device\_node和struct property结构体。\_\_unflatten\_device\_tree()代码列出如下：

|  |
| --- |
| /\*\*   \* unflatten\_dt\_node - Alloc and populate a  device\_node from the flat tree   \* @blob: The parent device tree blob   \* @mem: Memory chunk to use for allocating  device nodes and properties   \* @poffset: pointer to node in flat tree   \* @dad: Parent struct device\_node   \* @nodepp: The device\_node tree created by  the call   \* @fpsize: Size of the node path up at the  current depth.   \* @dryrun: If true, do not allocate device  nodes but still calculate needed   \* memory size   \*/  static void \* unflatten\_dt\_node(const  void \*blob,                void  \*mem,                int  \*poffset,                struct  device\_node \*dad,                struct  device\_node \*\*nodepp,                unsigned  long fpsize,                bool  dryrun)  {      const  \_\_be32 \*p;      struct  device\_node \*np;      struct  property \*pp, \*\*prev\_pp = NULL;      const  char \*pathp;      unsigned  int l, allocl;      static  int depth;      int  old\_depth;      int  offset;      int  has\_name = 0;      int  new\_format = 0;        /\*  获取node节点的name指针到pathp中 \*/      pathp  = fdt\_get\_name(blob, \*poffset, &l);      if  (!pathp)         return  mem;        allocl  = ++l;        /\*  version 0x10 has a more compact unit name here instead of the full       \* path. we accumulate the full path size  using "fpsize", we'll rebuild       \* it later. We detect this because the first  character of the name is       \* not '/'.       \*/      if  ((\*pathp) != '/') {         new\_format  = 1;         if  (fpsize == 0) {             /\*  root node: special case. fpsize accounts for path              \* plus terminating zero. root node only has  '/', so              \* fpsize should be 2, but we want to avoid  the first              \* level nodes to have two '/' so we use  fpsize 1 here              \*/             fpsize  = 1;             allocl  = 2;             l  = 1;             pathp  = "";         }  else {             /\*  account for '/' and path size minus terminal 0              \* already in 'l'              \*/             fpsize  += l;             allocl  = fpsize;         }      }        /\*  分配struct device\_node内存，包括路径全称大小 \*/      np  = unflatten\_dt\_alloc(&mem, sizeof(struct device\_node) + allocl,                \_\_alignof\_\_(struct  device\_node));      if  (!dryrun) {         char  \*fn;         of\_node\_init(np);           /\*  填充full\_name，full\_name指向该node节点的全路径名称字符串 \*/         np->full\_name  = fn = ((char \*)np) + sizeof(\*np);         if  (new\_format) {             /\*  rebuild full path for new format \*/             if  (dad && dad->parent) {                strcpy(fn,  dad->full\_name);                fn  += strlen(fn);             }             \*(fn++)  = '/';         }         memcpy(fn,  pathp, l);           /\*  节点挂接到相应的父节点、子节点和姊妹节点 \*/         prev\_pp  = &np->properties;         if  (dad != NULL) {             np->parent  = dad;             np->sibling  = dad->child;             dad->child  = np;         }      }      /\*  处理该node节点下面所有的property \*/      for  (offset = fdt\_first\_property\_offset(blob, \*poffset);           (offset >= 0);           (offset = fdt\_next\_property\_offset(blob,  offset))) {         const  char \*pname;         u32  sz;           if  (!(p = fdt\_getprop\_by\_offset(blob, offset, &pname, &sz))) {             offset  = -FDT\_ERR\_INTERNAL;             break;         }           if  (pname == NULL) {             pr\_info("Can't  find property name in list !\n");             break;         }         if  (strcmp(pname, "name") == 0)             has\_name  = 1;         pp  = unflatten\_dt\_alloc(&mem, sizeof(struct property),                    \_\_alignof\_\_(struct  property));         if  (!dryrun) {             /\*  We accept flattened tree phandles either in              \* ePAPR-style "phandle"  properties, or the              \* legacy "linux,phandle"  properties.  If both              \* appear and have different values, things              \* will get weird.  Don't do that. \*/               /\*  处理phandle，得到phandle值 \*/             if  ((strcmp(pname, "phandle") == 0) ||                 (strcmp(pname, "linux,phandle")  == 0)) {                if  (np->phandle == 0)                    np->phandle  = be32\_to\_cpup(p);             }             /\*  And we process the "ibm,phandle" property              \* used in pSeries dynamic device tree              \* stuff \*/             if  (strcmp(pname, "ibm,phandle") == 0)                np->phandle  = be32\_to\_cpup(p);             pp->name  = (char \*)pname;             pp->length  = sz;             pp->value  = (\_\_be32 \*)p;             \*prev\_pp  = pp;             prev\_pp  = &pp->next;         }      }      /\*  with version 0x10 we may not have the name property, recreate       \* it here from the unit name if absent       \*/      /\*  为每个node节点添加一个name的属性 \*/      if  (!has\_name) {         const  char \*p1 = pathp, \*ps = pathp, \*pa = NULL;         int  sz;           /\*  属性name的value值为node节点的名称，取“/”和“@”之间的子串 \*/         while  (\*p1) {             if  ((\*p1) == '@')                pa  = p1;             if  ((\*p1) == '/')                ps  = p1 + 1;             p1++;         }         if  (pa < ps)             pa  = p1;         sz  = (pa - ps) + 1;         pp  = unflatten\_dt\_alloc(&mem, sizeof(struct property) + sz,                    \_\_alignof\_\_(struct  property));         if  (!dryrun) {             pp->name  = "name";             pp->length  = sz;             pp->value  = pp + 1;             \*prev\_pp  = pp;             prev\_pp  = &pp->next;             memcpy(pp->value,  ps, sz - 1);             ((char  \*)pp->value)[sz - 1] = 0;         }      }      /\*  填充device\_node结构体中的name和type成员 \*/      if  (!dryrun) {         \*prev\_pp  = NULL;         np->name  = of\_get\_property(np, "name", NULL);         np->type  = of\_get\_property(np, "device\_type", NULL);           if  (!np->name)             np->name  = "<NULL>";         if  (!np->type)             np->type  = "<NULL>";      }        old\_depth  = depth;      \*poffset  = fdt\_next\_node(blob, \*poffset, &depth);      if  (depth < 0)         depth  = 0;      /\*  递归调用node节点下面的子节点 \*/      while  (\*poffset > 0 && depth > old\_depth)         mem  = unflatten\_dt\_node(blob, mem, poffset, np, NULL,                    fpsize,  dryrun);        if  (\*poffset < 0 && \*poffset != -FDT\_ERR\_NOTFOUND)         pr\_err("unflatten:  error %d processing FDT\n", \*poffset);        /\*       \* Reverse the child list. Some drivers  assumes node order matches .dts       \* node order       \*/      if  (!dryrun && np->child) {         struct  device\_node \*child = np->child;         np->child  = NULL;         while  (child) {             struct  device\_node \*next = child->sibling;             child->sibling  = np->child;             np->child  = child;             child  = next;         }      }        if  (nodepp)         \*nodepp  = np;        return  mem;  } |

通过以上函数处理就得到了所有的struct device\_node结构体，为每一个node都会自动添加一个名称为“name”的property，property.length的值为当前node的名称取最后一个“/”和“@”之间的子串（包括‘\0’）。例如：/serial@e2900800，则length = 7，property.value = device\_node.name = “serial”。

# ****6. platform\_device和device\_node绑定****

经过以上解析，DeviceTree的数据已经全部解析出具体的struct device\_node和struct property结构体，下面需要和具体的device进行绑定。首先讲解platform\_device和device\_node的绑定过程。在arch/arm/kernel/setup.c文件中，customize\_machine()函数负责填充struct platform\_device结构体。函数调用过程如图8所示。

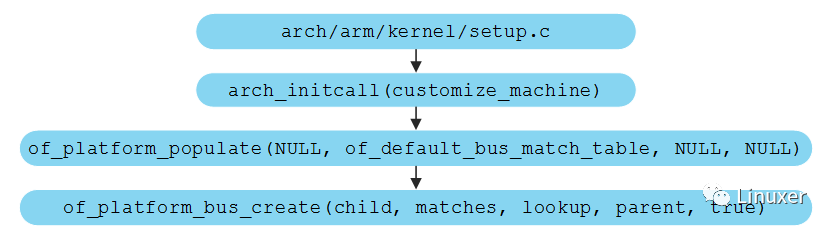


  图8 platform\_device生成流程图

代码分析如下：

|  |
| --- |
| const struct of\_device\_id  of\_default\_bus\_match\_table[] = {      {  .compatible = "simple-bus", },      {  .compatible = "simple-mfd", },  #ifdef CONFIG\_ARM\_AMBA      {  .compatible = "arm,amba-bus", },  #endif /\* CONFIG\_ARM\_AMBA \*/      {}  /\* Empty terminated list \*/  };    int of\_platform\_populate(struct  device\_node \*root,             const  struct of\_device\_id \*matches,             const  struct of\_dev\_auxdata \*lookup,             struct  device \*parent)  {      struct  device\_node \*child;      int  rc = 0;        /\*  获取根节点 \*/      root  = root ? of\_node\_get(root) : of\_find\_node\_by\_path("/");      if  (!root)         return  -EINVAL;        /\*  为根节点下面的每一个节点创建platform\_device结构体 \*/      for\_each\_child\_of\_node(root,  child) {         rc  = of\_platform\_bus\_create(child, matches, lookup, parent, true);         if  (rc) {             of\_node\_put(child);             break;         }      }      /\*  更新device\_node flag标志位 \*/      of\_node\_set\_flag(root,  OF\_POPULATED\_BUS);        of\_node\_put(root);      return  rc;  }    static int of\_platform\_bus\_create(struct  device\_node \*bus,                  const struct of\_device\_id \*matches,                  const struct of\_dev\_auxdata \*lookup,                  struct device \*parent, bool strict)  {      const  struct of\_dev\_auxdata \*auxdata;      struct  device\_node \*child;      struct  platform\_device \*dev;      const  char \*bus\_id = NULL;      void  \*platform\_data = NULL;      int  rc = 0;        /\*  只有包含"compatible"属性的node节点才会生成相应的platform\_device结构体 \*/      /\*  Make sure it has a compatible property \*/      if  (strict && (!of\_get\_property(bus, "compatible", NULL))) {         return  0;      }      /\*  省略部分代码 \*/      /\*       \* 针对节点下面得到status = "ok" 或者status = "okay"或者不存在status属性的       \* 节点分配内存并填充platform\_device结构体       \*/      dev  = of\_platform\_device\_create\_pdata(bus, bus\_id, platform\_data, parent);      if  (!dev || !of\_match\_node(matches, bus))         return  0;        /\*  递归调用节点解析函数，为子节点继续生成platform\_device结构体，前提是父节点       \* 的“compatible” = “simple-bus”，也就是匹配of\_default\_bus\_match\_table结构体中的数据       \*/      for\_each\_child\_of\_node(bus,  child) {         rc  = of\_platform\_bus\_create(child, matches, lookup, &dev->dev, strict);         if  (rc) {             of\_node\_put(child);             break;         }      }      of\_node\_set\_flag(bus,  OF\_POPULATED\_BUS);      return  rc;  } |

总的来说，当of\_platform\_populate()函数执行完毕，kernel就为DTB中所有包含compatible属性名的第一级node创建platform\_device结构体，并向平台设备总线注册设备信息。如果第一级node的compatible属性值等于“simple-bus”、“simple-mfd”或者"arm,amba-bus"的话，kernel会继续为当前node的第二级包含compatible属性的node创建platform\_device结构体，并注册设备。Linux系统下的设备大多都是挂载在平台总线下的，因此在平台总线被注册后，会根据of\_root节点的树结构，去寻找该总线的子节点，所有的子节点将被作为设备注册到该总线上。

# ****7. i2c\_client和device\_node绑定****

经过customize\_machine()函数的初始化，DTB已经转换成platform\_device结构体，这其中就包含i2c adapter设备，不同的SoC需要通过平台设备总线的方式自己实现i2c adapter设备的驱动。例如：i2c\_adapter驱动的probe函数中会调用i2c\_add\_numbered\_adapter()注册adapter驱动，函数流执行如图9所示。

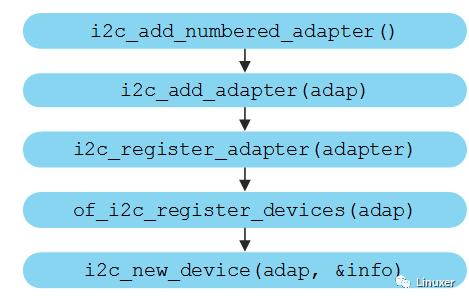


图9 i2c\_client绑定流程

在of\_i2c\_register\_devices()函数内部便利i2c节点下面的每一个子节点，并为子节点（status = “disable”的除外）创建i2c\_client结构体，并与子节点的device\_node挂接。其中i2c\_client的填充是在i2c\_new\_device()中进行的，最后device\_register()。在构建i2c\_client的时候，会对node下面的compatible属性名称的厂商名字去除作为i2c\_client的name。例如：compatible = “maxim,ds1338”,则i2c\_client->name = “ds1338”。

# ****8. Device\_Tree与sysfs****

kernel启动流程为start\_kernel()→rest\_init()→kernel\_thread():kernel\_init()→do\_basic\_setup()→driver\_init()→of\_core\_init()，在of\_core\_init()函数中在sys/firmware/devicetree/base目录下面为设备树展开成sysfs的目录和二进制属性文件，所有的node节点就是一个目录，所有的property属性就是一个二进制属性文件。