# (1条消息) [uboot] （番外篇）uboot 驱动模型_ooonebook的博客-CSDN博客_uboot 驱动模型

**[uboot] uboot**流程系列：
[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）
[project X] tiny210(s5pv210)从存储设备加载代码到DDR
[uboot]（第一章）uboot流程——概述
[uboot]（第二章）uboot流程——uboot-spl编译流程
[uboot]（第三章）uboot流程——uboot-spl代码流程
[uboot]（第四章）uboot流程——uboot编译流程
[uboot]（第五章）uboot流程——uboot启动流程
[uboot]（番外篇）global_data介绍
[uboot]（番外篇）uboot relocation介绍
[uboot]（番外篇）uboot之fdt介绍

建议先看《**[uboot]** （番外篇）**uboot**之**fdt**介绍》，了解一下**uboot**的**fdt**的功能，在驱动模型中会使用到。

==========

## 一、说明

### 1、**uboot**的驱动模型简单介绍

uboot引入了驱动模型（driver model），这种驱动模型为驱动的定义和访问接口提供了统一的方法。提高了驱动之间的兼容性以及访问的标准型。
uboot驱动模型和kernel中的设备驱动模型类似，但是又有所区别。
在后续我们将驱动模型（driver model）简称为DM，其实在uboot里面也是这样简称的。

具体细节建议参考./doc/driver-model/README.txt

### 2、如何使能**uboot**的**DM**功能

（1）配置**CONFIG_DM**
在configs/tiny210_defconfig中定义了如下：

```
CONFIG_DM=y
```

```
1
```

（2）使能相应的**uclass driver**的**config**。
DM和uclass是息息相关的，如果我们希望在某个模块引入DM，那么就需要使用相应模块的uclass driver来代替旧版的通用driver。
关于uclass我们会在后续继续说明。
以serial为例，为了在serial中引入DM，我在configs/tiny210_defconfig0中打开了CONFIG_DM_SERIAL宏，如下

```
CONFIG_DM_SERIAL=y
```

```
1
```

看driver/serial/Makefile

```
ifdef CONFIG_DM_SERIAL
obj-y += serial-uclass.o
else
obj-y += serial.o
endif
```

```
1
2
3
4
5
6
```

（3）对应设备驱动也要引入**dm**的功能
其设备驱动主要是实现和底层交互，为uclass层提供接口。后续再具体说明。

__后续都以serial-uclass进行说明

# 二、**uboot DM**整体架构

## **1**、**DM**的四个组成部分

uboot的DM主要有四个组成部分

- **udevice**
  简单就是指设备对象，可以理解为kernel中的device。
- **driver**
  udevice的驱动，可以理解为kernel中的device_driver。和底层硬件设备通信，并且为设备提供面向上层的接口。
- **uclass**

先看一下README.txt中关于uclass的说明：

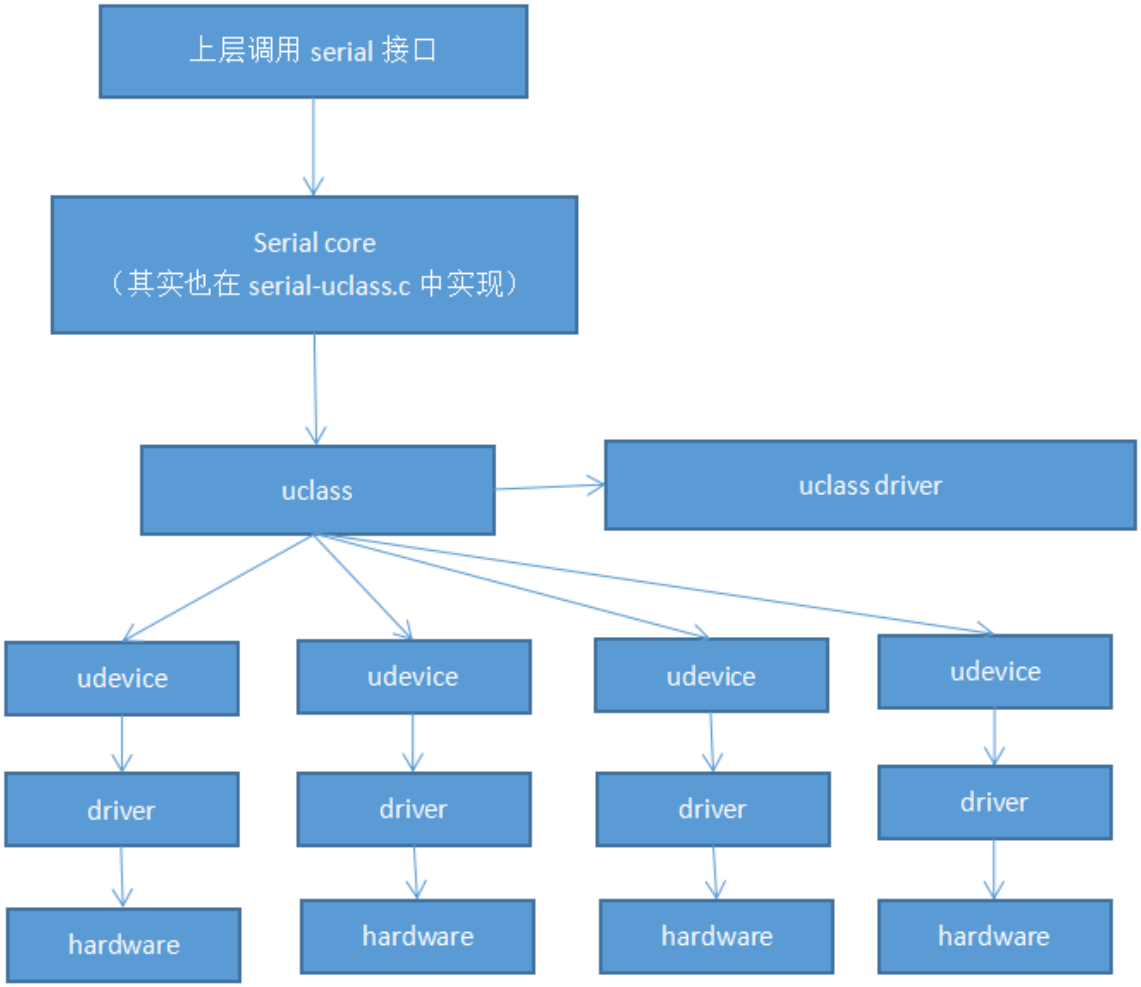
```
Uclass - a group of devices which operate in the same way. A uclass provides
        a way of accessing individual devices within the group, but always
        using the same interface. For example a GPIO uclass provides
        operations for get/set value. An I2C uclass may have 10 I2C ports,
        4 with one driver, and 6 with another.
```

- 1
- 2
- 3
- 4
- 5

uclass，使用相同方式的操作集的device的组。相当于是一种抽象。uclass为那些使用相同接口的设备提供了统一的接口。
例如，GPIO uclass提供了get/set接口。再例如，一个I2C uclass下可能有10个I2C端口，4个使用一个驱动，另外6个使用另外一个驱动。

- **uclass_driver**
  对应uclass的驱动程序。主要提供uclass操作时，如绑定udevice时的一些操作。

## **2**、调用关系框架图




### 3、相互之间的关系

结合上图来看：

- 上层接口都是和uclass的接口直接通讯。
- uclass可以理解为一些具有相同属性的udevice对外操作的接口，uclass的驱动是uclass_driver，主要为上层提供接口。

- udevice的是指具体设备的抽象，对应驱动是driver，driver主要负责和硬件通信，为uclass提供实际的操作集。
- udevice找到对应的uclass的方式主要是通过：udevice对应的driver的id和uclass对应的uclass_driver的id是否匹配。
- udevice会和uclass绑定。driver会和udevice绑定。uclass_driver会和uclass绑定。

这里先简单介绍一下：uclass和udevice都是动态生成的。在解析fdt中的设备的时候，会动态生成udevice。
然后找到udevice对应的driver，通过driver中的uclass id得到uclass_driver id。从uclass链表中查找对应的uclass是否已经生成，没有生成的话则动态生成uclass。

### **4**、**GD**中和**DM**相关的部分

```
 typedef struct global_data {
#ifdef CONFIG_DM
    struct udevice  *dm_root;

    struct udevice  *dm_root_f;

    struct list_head uclass_root;

#endif
} gd_t;
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

## 三、**DM**四个主要组成部分详细介绍

后续以数据结构、如何定义、存放位置、如何获取四个部分进行说明进行说明

### **0**、**uclass id**

每一种**uclass**都有自己对应的**ID**号。定义于其**uclass_driver**中。其附属的**udevice**的**driver**中的**uclass id**必须与其一致。
所有uclass id定义于include/dm/uclass-id.h中
列出部分id如下

```
enum uclass_id {

    UCLASS_ROOT = 0,
    UCLASS_DEMO,
    UCLASS_CLK,
    UCLASS_PINCTRL,
    UCLASS_SERIAL,
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

## 1、uclass

- （**1**）数据结构

```
struct uclass {
    void *priv;
    struct uclass_driver *uc_drv;
    struct list_head dev_head;
    struct list_head sibling_node;
};
```

- 1
- 2
- 3
- 4
- 5
- 6

- （**2**）如何定义
  uclass是uboot自动生成。并且不是所有uclass都会生成，有对应uclass driver并且有被udevice匹配到的uclass才会生成。
  具体参考后面的uboot DM初始化一节。或者参考uclass_add实现。

- （**3**）存放位置
  所有生成的uclass都会被挂载gd->uclass_root链表上。

- （**4**）如何获取、**API**
  直接遍历链表gd->uclass_root链表并且根据uclass id来获取到相应的uclass。

具体uclass_get-》uclass_find实现了这个功能。
有如下API:

```
int uclass_get(enum uclass_id key, struct uclass **ucp);
```

- 1
- 2

## 2、uclass_driver

- （**1**）数据结构
  include/dm/uclass.h

```
struct uclass_driver {
    const char *name;
    enum uclass_id id;

    int (*post_bind)(struct udevice *dev);
    int (*pre_unbind)(struct udevice *dev);
    int (*pre_probe)(struct udevice *dev);
    int (*post_probe)(struct udevice *dev);
    int (*pre_remove)(struct udevice *dev);
    int (*child_post_bind)(struct udevice *dev);
    int (*child_pre_probe)(struct udevice *dev);
    int (*init)(struct uclass *class);
    int (*destroy)(struct uclass *class);
    int priv_auto_alloc_size;
    int per_device_auto_alloc_size;
    int per_device_platdata_auto_alloc_size;
    int per_child_auto_alloc_size;
    int per_child_platdata_auto_alloc_size;
    const void *ops;
    uint32_t flags;
};
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21

- （**2**）如何定义
  通过**UCLASS_DRIVER**来定义**uclass_driver.**
  以serial-uclass为例

```
UCLASS_DRIVER(serial) = {
    .id        = UCLASS_SERIAL,
    .name        = "serial",
    .flags        = DM_UC_FLAG_SEQ_ALIAS,
    .post_probe    = serial_post_probe,
    .pre_remove    = serial_pre_remove,
    .per_device_auto_alloc_size = sizeof(struct serial_dev_priv),
};
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

UCLASS_DRIVER实现如下：

```
#define UCLASS_DRIVER(__name)                        \
    ll_entry_declare(struct uclass_driver, __name, uclass)

#define ll_entry_declare(_type, _name, _list)               \
    _type _u_boot_list_2_##_list##_2_##_name __aligned(4)       \
            __attribute__((unused,              \
            section(".u_boot_list_2_"#_list"_2_"#_name)))
```

关于ll_entry_declare我们在《[uboot] （第六章）uboot流程——命令行模式以及命令处理介绍》已经介绍过了

- 1

- 2
- 3
- 4
- 5
- 6
- 7
- 8

最终得到一个如下结构体

```
struct uclass_driver  _u_boot_list_2_uclass_2_serial = {
    .id        = UCLASS_SERIAL,
    .name        = "serial",
    .flags        = DM_UC_FLAG_SEQ_ALIAS,
    .post_probe    = serial_post_probe,
    .pre_remove    = serial_pre_remove,
    .per_device_auto_alloc_size = sizeof(struct serial_dev_priv),
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

并且存放在.u_boot_list_2_uclass_2_serial段中。

- （**3**）存放位置
  通过上述，我们知道serial的uclass_driver结构体_u_boot_list_2_uclass_2_serial被存放到.u_boot_list_2_uclass_2_serial段中。
  通过查看u-boot.map得到如下

```
.u_boot_list_2_uclass_1
                0x23e368e0        0x0 drivers/built-in.o
.u_boot_list_2_uclass_2_gpio
                0x23e368e0       0x48 drivers/gpio/built-in.o
                0x23e368e0                _u_boot_list_2_uclass_2_gpio
.u_boot_list_2_uclass_2_root
                0x23e36928       0x48 drivers/built-in.o
                0x23e36928                _u_boot_list_2_uclass_2_root
.u_boot_list_2_uclass_2_serial
                0x23e36970       0x48 drivers/serial/built-in.o
                0x23e36970                _u_boot_list_2_uclass_2_serial
.u_boot_list_2_uclass_2_simple_bus
```

```
                0x23e369b8       0x48 drivers/built-in.o
                0x23e369b8                _u_boot_list_2_uclass_2_simple_bus
.u_boot_list_2_uclass_3
                0x23e36a00        0x0 drivers/built-in.o
                0x23e36a00                . = ALIGN (0x4)
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17

最终，所有uclass driver结构体以列表的形式被放在.u_boot_list_2_uclass_1和.u_boot_list_2_uclass_3的区间中。
这个列表简称uclass_driver table。

- （**4**）如何获取、**API**
  想要获取uclass_driver需要先获取uclass_driver table。
  可以通过以下宏来获取uclass_driver table

```
struct uclass_driver *uclass =
    ll_entry_start(struct uclass_driver, uclass);


const int n_ents = ll_entry_count(struct uclass_driver, uclass);
```

- 1
- 2
- 3
- 4
- 5
- 6

接着通过遍历这个uclass_driver table，得到相应的uclass_driver。
有如下API

```
struct uclass_driver *lists_uclass_lookup(enum uclass_id id)
```

- 1
- 2

## 3、udevice

- （1）数据结构
  include/dm/device.h

```
struct udevice {
    const struct driver *driver;
    const char *name;
    void *platdata;
    void *parent_platdata;
    void *uclass_platdata;
    int of_offset;
    ulong driver_data;
    struct udevice *parent;
    void *priv;
    struct uclass *uclass;
    void *uclass_priv;
    void *parent_priv;
    struct list_head uclass_node;
    struct list_head child_head;
    struct list_head sibling_node;
    uint32_t flags;
    int req_seq;
    int seq;
#ifdef CONFIG_DEVRES
    struct list_head devres_head;
#endif
};
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12

- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23

- （**2**）如何定义
  在dtb存在的情况下，由uboot解析dtb后动态生成，后续在“uboot DM的初始化”一节中具体说明。

- （**3**）存放位置

  - 连接到对应uclass中
    也就是会连接到uclass->dev_head中
  - 连接到父设备的子设备链表中
    也就是会连接到udevice->child_head中，并且最终的根设备是gd->dm_root这个根设备。

- （**4**）如何获取、**API**

  - 从uclass中获取udevice
    遍历uclass->dev_head，获取对应的udevice。有如下API

```
#define uclass_foreach_dev(pos, uc) \
    list_for_each_entry(pos, &uc->dev_head, uclass_node)

#define uclass_foreach_dev_safe(pos, next, uc)  \
    list_for_each_entry_safe(pos, next, &uc->dev_head, uclass_node)

int uclass_get_device(enum uclass_id id, int index, struct udevice **devp);
int uclass_get_device_by_name(enum uclass_id id, const char *name,
                  struct udevice **devp);
int uclass_get_device_by_seq(enum uclass_id id, int seq, struct udevice **devp);
int uclass_get_device_by_of_offset(enum uclass_id id, int node,
                   struct udevice **devp);
int uclass_get_device_by_phandle(enum uclass_id id, struct udevice *parent,
                 const char *name, struct udevice **devp);
int uclass_first_device(enum uclass_id id, struct udevice **devp);
int uclass_first_device_err(enum uclass_id id, struct udevice **devp);
int uclass_next_device(struct udevice **devp);
int uclass_resolve_seq(struct udevice *dev);
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18

## 4、driver

和uclass_driver方式是相似的。

- （**1**）数据结构
  include/dm/device.h

```
struct driver {
    char *name;
    enum uclass_id id;
    const struct udevice_id *of_match;
    int (*bind)(struct udevice *dev);
    int (*probe)(struct udevice *dev);
    int (*remove)(struct udevice *dev);
    int (*unbind)(struct udevice *dev);
    int (*ofdata_to_platdata)(struct udevice *dev);
    int (*child_post_bind)(struct udevice *dev);
    int (*child_pre_probe)(struct udevice *dev);
    int (*child_post_remove)(struct udevice *dev);
    int priv_auto_alloc_size;
    int platdata_auto_alloc_size;
    int per_child_auto_alloc_size;
    int per_child_platdata_auto_alloc_size;
    const void *ops;
    uint32_t flags;
};
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19

- （**2**）如何定义
  通过**U_BOOT_DRIVER**来定义一个**driver**
  以s5pv210为例:
  driver/serial/serial_s5p.c

```
U_BOOT_DRIVER(serial_s5p) = {
    .name    = "serial_s5p",
    .id    = UCLASS_SERIAL,
    .of_match = s5p_serial_ids,
    .ofdata_to_platdata = s5p_serial_ofdata_to_platdata,
    .platdata_auto_alloc_size = sizeof(struct s5p_serial_platdata),
    .probe = s5p_serial_probe,
    .ops    = &s5p_serial_ops,
    .flags = DM_FLAG_PRE_RELOC,
};
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

U_BOOT_DRIVER实现如下：

```
#define U_BOOT_DRIVER(__name)                         \
    ll_entry_declare(struct driver, __name, driver)

#define ll_entry_declare(_type, _name, _list)               \
    _type _u_boot_list_2_##_list##_2_##_name __aligned(4)       \
            __attribute__((unused,              \
            section(".u_boot_list_2_"#_list"_2_"#_name)))
```

关于ll_entry_declare我们在《[uboot] （第六章）uboot流程—命令行模式以及命令处理介绍》已经介绍过了

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

最终得到如下一个结构体

```
struct driver _u_boot_list_2_driver_2_serial_s5p= {
    .name    = "serial_s5p",
    .id    = UCLASS_SERIAL,
    .of_match = s5p_serial_ids,
    .ofdata_to_platdata = s5p_serial_ofdata_to_platdata,
    .platdata_auto_alloc_size = sizeof(struct s5p_serial_platdata),
    .probe = s5p_serial_probe,
    .ops    = &s5p_serial_ops,
    .flags = DM_FLAG_PRE_RELOC,
};
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

并且存放在.u_boot_list_2_driver_2_serial_s5p段中

- （**3**）存放位置
  通过上述，我们知道serial_s5p的driver结构体_u_boot_list_2_driver_2_serial_s5p被存放在.u_boot_list_2_driver_2_serial_s5p段中。
  通过查看u-boot.map得到如下

```
 .u_boot_list_2_driver_1
                0x23e36754        0x0 drivers/built-in.o
.u_boot_list_2_driver_2_gpio_exynos
                0x23e36754       0x44 drivers/gpio/built-in.o
                0x23e36754                _u_boot_list_2_driver_2_gpio_exynos
.u_boot_list_2_driver_2_root_driver
                0x23e36798       0x44 drivers/built-in.o
                0x23e36798                _u_boot_list_2_driver_2_root_driver
.u_boot_list_2_driver_2_serial_s5p
                0x23e367dc       0x44 drivers/serial/built-in.o
                0x23e367dc                _u_boot_list_2_driver_2_serial_s5p
.u_boot_list_2_driver_2_simple_bus_drv
                0x23e36820       0x44 drivers/built-in.o
                0x23e36820                _u_boot_list_2_driver_2_simple_bus_drv
.u_boot_list_2_driver_3
                0x23e36864        0x0 drivers/built-in.o
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16

最终，所有driver结构体以列表的形式被放在.u_boot_list_2_driver_1和.u_boot_list_2_driver_3的区间中。
这个列表简称driver table。

- （**4**）如何获取、**API**
  想要获取driver需要先获取driver table。
  可以通过以下宏来获取driver table

```
struct driver *drv =
    ll_entry_start(struct driver, driver);


const int n_ents = ll_entry_count(struct driver, driver);
```

- 1
- 2
- 3
- 4
- 5
- 6

接着通过遍历这个driver table，得到相应的driver。

```
struct driver *lists_driver_lookup_name(const char *name)
```

- 1
- 2

## 四、DM的一些API整理

先看一下前面一节理解一下。

### 1、uclass相关API

```
int uclass_get(enum uclass_id key, struct uclass **ucp);
```

- 1
- 2
- 3
- 4

### 2、uclass_driver相关API

```
struct uclass_driver *lists_uclass_lookup(enum uclass_id id)
```

- 1
- 2

### 3、udevice相关API

```
#define uclass_foreach_dev(pos, uc) \
    list_for_each_entry(pos, &uc->dev_head, uclass_node)

#define uclass_foreach_dev_safe(pos, next, uc)  \
    list_for_each_entry_safe(pos, next, &uc->dev_head, uclass_node)


int device_bind(struct udevice *parent, const struct driver *drv,
        const char *name, void *platdata, int of_offset,
        struct udevice **devp)


int device_bind_by_name(struct udevice *parent, bool pre_reloc_only,
            const struct driver_info *info, struct udevice **devp)


int uclass_bind_device(struct udevice *dev)

{
    uc = dev->uclass;
    list_add_tail(&dev->uclass_node, &uc->dev_head);
}

int uclass_get_device(enum uclass_id id, int index, struct udevice **devp);
int uclass_get_device_by_name(enum uclass_id id, const char *name,
                  struct udevice **devp);
int uclass_get_device_by_seq(enum uclass_id id, int seq, struct udevice **devp);
int uclass_get_device_by_of_offset(enum uclass_id id, int node,
                   struct udevice **devp);
int uclass_get_device_by_phandle(enum uclass_id id, struct udevice *parent,
                 const char *name, struct udevice **devp);
int uclass_first_device(enum uclass_id id, struct udevice **devp);
int uclass_first_device_err(enum uclass_id id, struct udevice **devp);
int uclass_next_device(struct udevice **devp);
int uclass_resolve_seq(struct udevice *dev);
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35

### 4、driver相关API

```
struct driver *lists_driver_lookup_name(const char *name)
```

- 1
- 2

## 五、uboot 设备的表达

### 1、说明

uboot中可以通过两种方法来添加设备

- 通过直接定义平台设备（这种方式基本上不使用）
- 通过在设备树添加设备信息

注意：这里只是设备的定义，最终还是会被**uboot**解析成**udevice**结构体的。

**2**、直接定义平台设备（这种方式除了根设备外基本上不使用）

（1）通过U_BOOT_DEVICE宏来进行定义或者直接定义struct driver_info结构体
（2）U_BOOT_DEVICE宏以ns16550_serial为例

```
U_BOOT_DEVICE(overo_uart) = {
    "ns16550_serial",
    &overo_serial
};
```

U_BOOT_DEVICE实现如下：
和上述的U_BOOT_DRIVER类似，这里不详细说明了

```
#define U_BOOT_DEVICE(__name)                       \
    ll_entry_declare(struct driver_info, __name, driver_info)


#define U_BOOT_DEVICES(__name)                      \
    ll_entry_declare_list(struct driver_info, __name, driver_info)
```

（3）直接定义struct driver_info结构体，以根设备为例
uboot会创建一个根设备root，作为所有设备的祖设备
root的定义如下：

```
static const struct driver_info root_info = {
    .name       = "root_driver",
};
```

**3**、在设备树添加设备信息

在对应的dts文件中添加相应的设备节点和信息，以tiny210的serial为例：
arch/arm/dts/s5pv210-tiny210.dts

```
/dts-v1/;
#include "skeleton.dtsi"
/{
        aliases {
                console = "/serial@e2900000";
        };

        serial@e2900000 {
                compatible = "samsung,exynos4210-uart";
                reg = <0xe2900000 0x100>;
                interrupts = <0 51 0>;
                id = <0>;
        };
};
```

dts的内容这里不多说了。

## 六、**uboot DM**的初始化

关于下面可能使用到的一些FDT的API可以参考一下《[uboot] （番外篇）uboot之fdt介绍》。
关于下面可能使用到一些DM的API可以参考一下上述第四节。

### **1**、主要工作

- DM的初始化

    - 创建根设备root的udevice，存放在gd->dm_root中。
      根设备其实是一个虚拟设备，主要是为uboot的其他设备提供一个挂载点。
    - 初始化uclass链表gd->uclass_root

- DM中udevice和uclass的解析

    - udevice的创建和uclass的创建
    - udevice和uclass的绑定
    - uclass_driver和uclass的绑定
    - driver和udevice的绑定
    - 部分driver函数的调用

## **2**、入口说明

**dm**初始化的接口在**dm_init_and_scan**中。
可以发现在uboot relocate之前的initf_dm和之后的initr_dm都调用了这个函数。

```
static int initf_dm(void)
{
#if defined(CONFIG_DM) && defined(CONFIG_SYS_MALLOC_F_LEN)
    int ret;
    ret = dm_init_and_scan(true);
    if (ret)
        return ret;
#endif
    return 0;
}

#ifdef CONFIG_DM
static int initr_dm(void)
{
    int ret;

    gd->dm_root_f = gd->dm_root;
    gd->dm_root = NULL;
    ret = dm_init_and_scan(false);
    if (ret)
        return ret;
    return 0;
}
#endif
```

- 1
- 2

- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24

主要区别在于参数。
首先说明一下dts节点中的“u-boot,dm-pre-reloc”属性，当设置了这个属性时，则表示这个设备在relocate之前就需要使用。
当dm_init_and_scan的参数为true时，只会对带有“u-boot,dm-pre-reloc”属性的节点进行解析。而当参数为false的时候，则会对所有节点都进行解析。
由于“u-boot,dm-pre-reloc”的情况比较少，所以这里只学习参数为false的情况。也就是initr_dm里面的dm_init_and_scan(false);。

## 2、dm_init_and_scan说明

driver/core/root.c

```
int dm_init_and_scan(bool pre_reloc_only)
{
    int ret;

    ret = dm_init();
    if (ret) {
        debug("dm_init() failed: %d\n", ret);
        return ret;
    }
    ret = dm_scan_platdata(pre_reloc_only);
    if (ret) {
```

```
            debug("dm_scan_platdata() failed: %d\n", ret);
            return ret;
        }

        if (CONFIG_IS_ENABLED(OF_CONTROL)) {
            ret = dm_scan_fdt(gd->fdt_blob, pre_reloc_only);
            if (ret) {
                debug("dm_scan_fdt() failed: %d\n", ret);
                return ret;
            }
        }

        ret = dm_scan_other(pre_reloc_only);
        if (ret)
            return ret;

        return 0;
    }
```

- 28
- 29

### 3、DM的初始化——dm_init

对应代码如下：
driver/core/root.c

```
#define DM_ROOT_NON_CONST       (((gd_t *)gd)->dm_root)
#define DM_UCLASS_ROOT_NON_CONST    (((gd_t *)gd)->uclass_root)

int dm_init(void)
{
    int ret;

    if (gd->dm_root) {

        dm_warn("Virtual root driver already exists!\n");
        return -EINVAL;
    }

    INIT_LIST_HEAD(&DM_UCLASS_ROOT_NON_CONST);


    ret = device_bind_by_name(NULL, false, &root_info, &DM_ROOT_NON_CONST);




    if (ret)
        return ret;
#if CONFIG_IS_ENABLED(OF_CONTROL)
    DM_ROOT_NON_CONST->of_offset = 0;
#endif
    ret = device_probe(DM_ROOT_NON_CONST);



    if (ret)
        return ret;

    return 0;
}
```

- 1

```
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

这里就完成的DM的初始化了
（1）创建根设备root的udevice，存放在gd->dm_root中。
（2）初始化uclass链表gd->uclass_root

### **4**、从平台设备中解析**udevice**和**uclass——dm_scan_platdata**

跳过。

### **5**、从**dtb**中解析**udevice**和**uclass——dm_scan_fdt**

关于fdt以及一些对应API请参考《[uboot] （番外篇）uboot之fdt介绍》。
对应代码如下（后续我们忽略pre_reloc_only=true的情况）：
driver/core/root.c

```
int dm_scan_fdt(const void *blob, bool pre_reloc_only)

{
    return dm_scan_fdt_node(gd->dm_root, blob, 0, pre_reloc_only);

}

int dm_scan_fdt_node(struct udevice *parent, const void *blob, int offset,
             bool pre_reloc_only)




{
    int ret = 0, err;



    for (offset = fdt_first_subnode(blob, offset);

         offset > 0;
         offset = fdt_next_subnode(blob, offset)) {

        if (!fdtdec_get_is_enabled(blob, offset)) {

            dm_dbg("   - ignoring disabled device\n");
            continue;
        }
        err = lists_bind_fdt(parent, blob, offset, NULL);

        if (err && !ret) {
            ret = err;
            debug("%s: ret=%d\n", fdt_get_name(blob, offset, NULL),
                  ret);
        }
    }
    return ret;
}
```

- 1

- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40

**lists_bind_fdt**是从**dtb**中解析**udevice**和**uclass**的核心。
其具体实现如下:
driver/core/lists.c

```
int lists_bind_fdt(struct udevice *parent, const void *blob, int offset,
           struct udevice **devp)
```

```
{
    struct driver *driver = ll_entry_start(struct driver, driver);

    const int n_ents = ll_entry_count(struct driver, driver);

    const struct udevice_id *id;
    struct driver *entry;
    struct udevice *dev;
    bool found = false;
    const char *name;
    int result = 0;
    int ret = 0;

    dm_dbg("bind node %s\n", fdt_get_name(blob, offset, NULL));

    if (devp)
        *devp = NULL;
    for (entry = driver; entry != driver + n_ents; entry++) {

        ret = driver_check_compatible(blob, offset, entry->of_match,
                          &id);

        name = fdt_get_name(blob, offset, NULL);

        if (ret == -ENOENT) {
            continue;
        } else if (ret == -ENODEV) {
            dm_dbg("Device '%s' has no compatible string\n", name);
            break;
        } else if (ret) {
            dm_warn("Device tree error at offset %d\n", offset);
            result = ret;
            break;
        }

        dm_dbg("   - found match at '%s'\n", entry->name);
        ret = device_bind(parent, entry, name, NULL, offset, &dev);

        if (ret) {
            dm_warn("Error binding driver '%s': %d\n", entry->name,
                ret);
            return ret;
        } else {
            dev->driver_data = id->data;
            found = true;
```

```
            if (devp)
                *devp = dev;

        }
        break;
    }

    if (!found && !result && ret != -ENODEV) {
        dm_dbg("No match for node '%s'\n",
               fdt_get_name(blob, offset, NULL));
    }

    return result;
}
```

- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62

在device_bind中实现了udevice和uclass的创建和绑定以及一些初始化操作，这里专门学习一下device_bind。
device_bind的实现如下(去除部分代码)
driver/core/device.c

```
int device_bind(struct udevice *parent, const struct driver *drv,
        const char *name, void *platdata, int of_offset,
        struct udevice **devp)




{
```

```
struct udevice *dev;
struct uclass *uc;
int size, ret = 0;

ret = uclass_get(drv->id, &uc);


dev = calloc(1, sizeof(struct udevice));


dev->platdata = platdata;
dev->name = name;
dev->of_offset = of_offset;
dev->parent = parent;
dev->driver = drv;
dev->uclass = uc;

dev->seq = -1;
dev->req_seq = -1;
if (CONFIG_IS_ENABLED(OF_CONTROL) && CONFIG_IS_ENABLED(DM_SEQ_ALIAS)) {

    if (uc->uc_drv->flags & DM_UC_FLAG_SEQ_ALIAS) {
        if (uc->uc_drv->name && of_offset != -1) {
            fdtdec_get_alias_seq(gd->fdt_blob,
                    uc->uc_drv->name, of_offset,
                    &dev->req_seq);
        }

    }
}

if (!dev->platdata && drv->platdata_auto_alloc_size) {
    dev->flags |= DM_FLAG_ALLOC_PDATA;
    dev->platdata = calloc(1, drv->platdata_auto_alloc_size);

}

size = uc->uc_drv->per_device_platdata_auto_alloc_size;
if (size) {
    dev->flags |= DM_FLAG_ALLOC_UCLASS_PDATA;
    dev->uclass_platdata = calloc(1, size);

}


if (parent)
```

```
    list_add_tail(&dev->sibling_node, &parent->child_head);


ret = uclass_bind_device(dev);



if (drv->bind) {
    ret = drv->bind(dev);

}

if (parent && parent->driver->child_post_bind) {
    ret = parent->driver->child_post_bind(dev);

}
if (uc->uc_drv->post_bind) {
    ret = uc->uc_drv->post_bind(dev);
    if (ret)
        goto fail_uclass_post_bind;

}

if (devp)
    *devp = dev;


dev->flags |= DM_FLAG_BOUND;



return 0;
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13

- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59

- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79
- 80
- 81
- 82
- 83
- 84
- 85
- 86
- 87
- 88
- 89
- 90
- 91
- 92
- 93
- 94

上述就完成了dtb的解析，udevice和uclass的创建，以及各个组成部分的绑定关系。
注意，这里只是绑定，即调用了**driver**的**bind**函数，但是设备还没有真正激活，也就是还没有执行设备的**probe**函数。

## 七、**DM**工作流程

经过前面的**DM**初始化以及设备解析之后，我们只是建立了**udevice**和**uclass**之间的绑定关系。但是此时**udevice**还没有被**probe**，其对应设备还没有被激活。
激活一个设备主要是通过device_probe函数，所以在介绍DM的工作流程前，先说明device_probe函数。

## 1、device_probe

driver/core/device.c

```
int device_probe(struct udevice *dev)
{
    const struct driver *drv;
    int size = 0;
    int ret;
    int seq;

    if (dev->flags & DM_FLAG_ACTIVATED)
        return 0;


    drv = dev->driver;
    assert(drv);



    if (drv->priv_auto_alloc_size && !dev->priv) {
        dev->priv = alloc_priv(drv->priv_auto_alloc_size, drv->flags);

    }


    size = dev->uclass->uc_drv->per_device_auto_alloc_size;
    if (size && !dev->uclass_priv) {
        dev->uclass_priv = calloc(1, size);

    }



    seq = uclass_resolve_seq(dev);
    if (seq < 0) {
        ret = seq;
        goto fail;
    }
    dev->seq = seq;

    dev->flags |= DM_FLAG_ACTIVATED;


    ret = uclass_pre_probe_device(dev);
```

```
    if (drv->ofdata_to_platdata && dev->of_offset >= 0) {
        ret = drv->ofdata_to_platdata(dev);

    }

    if (drv->probe) {
        ret = drv->probe(dev);

    }

    ret = uclass_post_probe_device(dev);

    return ret;
}
```

- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58

主要工作归纳如下：

- 分配设备的私有数据
- 对父设备进行probe
- 执行probe device之前uclass需要调用的一些函数
- 调用driver的ofdata_to_platdata，将dts信息转化为设备的平台数据
- 调用driver的probe函数
- 执行probe device之后uclass需要调用的一些函数

## 2、通过**uclass**来获取一个**udevice**并且进行**probe**

通过uclass来获取一个udevice并且进行probe有如下接口
driver/core/uclass.c

```
int uclass_get_device(enum uclass_id id, int index, struct udevice **devp)
int uclass_get_device_by_name(enum uclass_id id, const char *name,
                  struct udevice **devp)
int uclass_get_device_by_seq(enum uclass_id id, int seq, struct udevice **devp)
int uclass_get_device_by_of_offset(enum uclass_id id, int node,
                   struct udevice **devp)
int uclass_get_device_by_phandle(enum uclass_id id, struct udevice *parent,
                 const char *name, struct udevice **devp)
int uclass_first_device(enum uclass_id id, struct udevice **devp)
int uclass_next_device(struct udevice **devp)
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

这些接口主要是获取设备的方法上有所区别，但是probe设备的方法都是一样的，都是通过调用uclass_get_device_tail->device_probe来probe设备的。
以uclass_get_device为例

```
int uclass_get_device(enum uclass_id id, int index, struct udevice **devp)
{
    struct udevice *dev;
    int ret;

    *devp = NULL;
    ret = uclass_find_device(id, index, &dev);
    return uclass_get_device_tail(dev, ret, devp);
}

int uclass_get_device_tail(struct udevice *dev, int ret,
                  struct udevice **devp)
{
    ret = device_probe(dev);

    if (ret)
        return ret;

    *devp = dev;
```

```
    return 0;
}
```

## 3、工作流程简单说明

serial-uclass较为简单，我们以serial-uclass为例

- （**0**）代码支持
  < 1 > serial-uclass.c中定义一个uclass_driver

```
UCLASS_DRIVER(serial) = {
    .id         = UCLASS_SERIAL,
    .name         = "serial",
    .flags         = DM_UC_FLAG_SEQ_ALIAS,
    .post_probe     = serial_post_probe,
    .pre_remove     = serial_pre_remove,
    .per_device_auto_alloc_size = sizeof(struct serial_dev_priv),
};
```

```
4
5
6
7
8
```

< 2 > 定义s5pv210的serial的dts节点

```
        serial@e2900000 {
                compatible = "samsung,exynos4210-uart";
                reg = <0xe2900000 0x100>;
                interrupts = <0 51 0>;
                id = <0>;
        };
```

```
1
2
3
4
5
6
7
```

< 3 > 定义设备驱动

```
U_BOOT_DRIVER(serial_s5p) = {
    .name    = "serial_s5p",
    .id    = UCLASS_SERIAL,
    .of_match = s5p_serial_ids,
    .ofdata_to_platdata = s5p_serial_ofdata_to_platdata,
    .platdata_auto_alloc_size = sizeof(struct s5p_serial_platdata),
    .probe = s5p_serial_probe,
    .ops    = &s5p_serial_ops,
    .flags = DM_FLAG_PRE_RELOC,
};

static const struct udevice_id s5p_serial_ids[] = {
    { .compatible = "samsung,exynos4210-uart" },
    { }
};
```

```
1
2
3
4
```

- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15

- （**1**）**udevice**和对应**uclass**的创建
  在DM初始化的过程中uboot自己创建对应的udevice和uclass。
  具体参考“六、uboot DM的初始化”

- （**2**）**udevice**和对应**uclass**的绑定
  在DM初始化的过程中uboot自己实现将udevice绑定到对应的uclass中。
  具体参考“六、uboot DM的初始化”

- （**3**）对应**udevice**的**probe**
  由模块自己实现。例如serial则需要在serial的初始化过程中，选择需要的udevice进行probe。
  serial-uclass只是操作作为console的serial，并不具有通用性，这里简单的了解下。
  代码如下，过滤掉无关代码
  driver/serial/serial-uclass.c

```
int serial_init(void)
{
    serial_find_console_or_panic();
    gd->flags |= GD_FLG_SERIAL_READY;

    return 0;
}

static void serial_find_console_or_panic(void)
{
    const void *blob = gd->fdt_blob;
    struct udevice *dev;
    int node;

    if (CONFIG_IS_ENABLED(OF_CONTROL) && blob) {
```

```
            if (!uclass_get_device_by_of_offset(UCLASS_SERIAL, node,
                                &dev)) {


                gd->cur_serial_dev = dev;


                return;
            }
        }
    }
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30

- （**4**）**uclass**的接口调用
  - 可以通过先从root_uclass链表中提取对应的uclass，然后通过uclass->uclass_driver->ops来进行接口调用，这种方法比较具有

通用性。
- 可以通过调用uclass直接expert的接口，不推荐，但是serial-uclass使用的是这种方式。
这部分应该属于serial core，但是也放在了serial-uclass.c中实现。
以serial_putc调用为例，serial-uclass使用如下：

```
void serial_putc(char ch)
{
    if (gd->cur_serial_dev)
        _serial_putc(gd->cur_serial_dev, ch);
}

static void _serial_putc(struct udevice *dev, char ch)
{
    struct dm_serial_ops *ops = serial_get_ops(dev);
    int err;

    do {
        err = ops->putc(dev, ch);
    } while (err == -EAGAIN);
}
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

到此整个流程简单介绍到这。

这里几乎都是纸上谈兵，后续会来一篇**gpio-uclass**的使用实战。