uboot 设备驱动模型

最近在移植 uboot-2015.04 的时候发现，uboot 的设备驱动也带驱动模型了，第一次见到的时候还真是愣了一下，特别是调试的时候没有以前那么方便直接了。而且设备模型和设备树捆绑在一起，又得花费一番功夫来了解了。不禁深深的感慨，搞技术的真真切切就是活到老学到老，而且这种一直学习的状态其实是外界不断强加给你的，有时候真的觉得挺累。人家改改模型，你就得重新去学习适应。其实这种学习都是高投入低产出的，真的希望国内搞技术的同志们能够加油加油加油，争取走到产业的上游，让人家去用我们的东西，让人家去跟随我们的步伐。希望有人看到这些东西，加快他们前进的步伐，能把更多的精力花在创造更多价值的事情上！！！

先讲几大要点:

要点 1： 设备模型始于 initf_dm 接口要点 2： 整个过程穿插着设备树的东西，所以最好理解设备树

1. 设备驱动入口

以前驱动入口就是一个类似 init 函数的东东，现在不一样了，都统一为一个宏 U_BOOT_DRIVER

```
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,
};
```

这里先不管这个东西，知道就行，一会儿用到的时候再解释。

2. 驱动模型

如果说设备驱动对象是一个成员，那么模型就是用来管理这些成员的规则。整个模型的入口是`initf_dm`， 其实就是调用了`dm_init_and_scan`

```
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;
}
```

`dm_init_and_scan`函数中调用了以下几个函数:

1. dm_init
2. dm_scan_platdata
3. dm_scan_fdt (在配置了 of_control 的情况下)
4. dm_scan_other

2.1 dm_init 函数。

```
/* drivers/core/root.c */
```

```
int dm_init(void)
{
    int ret;

    if (gd->dm_root) {
        dm_warn("Virtual root driver already exists!n");
        return -EINVAL;
    }
    // 初始化uclass_list
 INIT_LIST_HEAD(&DM_UCLASS_ROOT_NON_CONST);

#if defined(CONFIG_NEEDS_MANUAL_RELOC)
 fix_drivers();
    fix_uclass();
#endif

    ret = device_bind_by_name(NULL, false, &root_info, &DM_ROOT_NON_CONST);
    if (ret)
        return ret;
#ifdef CONFIG_OF_CONTROL
 DM_ROOT_NON_CONST->of_offset = 0;
#endif
 ret = device_probe(DM_ROOT_NON_CONST);
    if (ret)
        return ret;

    return 0;
}
```

其实我很讨厌分析代码，首先肯定免不了贴代码，容易照本宣科；其次语言难以发挥，很不爽。其实每个函数无非做了两件事，第一件事：焚香沐浴更衣，第二件事：真刀真枪开干。说多了就显得无聊，这里也是这样的，无非就是结构初始化，然后就调用一些接口做些事情。至于初始化为了做什么，其实都在代码中，tag 跳一下，稍微看看就能懂了。这里做实际工作的就是两个地方

```
device_bind_by_name(NULL, false, &root_info, &DM_ROOT_NON_CONST);

device_probe(DM_ROOT_NON_CONST);
```

2.1.1 device_bind_by_name

原型是这样的

```
/* drivers/core/device.c */
int device_bind_by_name(struct udevice *parent, bool pre_reloc_only, const struct driver_info *info, struct udevice **devp);

int device_probe(struct udevice *dev);
```

既然叫 device_bind_by_name，那么肯定有这么几个点是要猜到的:

1. 这个接口是要进行搜索的
2. 搜索是根据名字来的

一开始可能会觉得自己只能马后炮分析一下，但是我觉得要做到条件反射，因为计算机编程语言本质上也是语言，对语言不就应该条件反射式的反应吗？不然怎么能够理解并且构建丰富多彩的世界呢。再贴代码

```
int device_bind_by_name(struct udevice *parent, bool pre_reloc_only,
            const struct driver_info *info, struct udevice **devp)
{
    struct driver *drv;

    // 搜索driver,这里重点关注root driver
 drv = lists_driver_lookup_name(info->name);
    if (!drv)
        return -ENOENT;
    if (pre_reloc_only && !(drv->flags & DM_FLAG_PRE_RELOC))
        return -EPERM;

    // 绑定driver和device，重点关注root driver和root device
 return device_bind(parent, drv, info->name, (void *)info->platdata, -1, devp);
}
```

关于 list_driver_lookup_name 就没必要贴了，只要知道她会根据名字返回对应的通过U_BOOT_DRIVER 声明的驱动就行。这里返回的是 root_driver，这个要贴一下，还有个 uclass 也贴一下，因为马上用到，注意两者 id 是一样的。

```
/* This is the root driver - all drivers are children of this */
U_BOOT_DRIVER(root_driver) = {
    // 这个名字和info->name是不是同一个东西?哈哈
```

```
 .name = "root_driver",
    .id   = UCLASS_ROOT,
};

/* This is the root uclass */
UCLASS_DRIVER(root) = {
    .name = "root",
    .id   = UCLASS_ROOT,
};
```

好，我们得到了 root_driver，其实看这个名字也要有条件反射，既然是 root，那么其他 driver 就是 node 了，而且是可以通过 root 搜索到的。所以我们就要把这些 node 和 root 绑定起来。这是后面要做的事情，root_driver 我们还没处理，先来看 device_bind。再贴代码，靠，真他妈恶心。

```
int device_bind(struct udevice *parent, struct driver *drv, const char *name, void *platdata, int of_offset, struct udevice **devp)
{
    struct udevice *dev;
    struct uclass *uc;
    int ret = 0;

    *devp = NULL;
    if (!name)
        return -EINVAL;
    // 和前面那个搜索driver的接口一样，这里是根据id搜索对应的uclass
 ret = uclass_get(drv->id, &uc);
    if (ret)
        return ret;

    // 创建设备udevice对象
 dev = calloc(1, sizeof(struct udevice));
    if (!dev)
        return -ENOMEM;

    // 构建树形结构需要的对象，放兄弟和儿子的地方
 INIT_LIST_HEAD(&dev->sibling_node);
    INIT_LIST_HEAD(&dev->child_head);
    INIT_LIST_HEAD(&dev->uclass_node);
    // 配置设备对象
 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;
#ifdef CONFIG_OF_CONTROL
 /*
     * Some devices, such as a SPI bus, I2C bus and serial ports are
     * numbered using aliases.
     *
     * This is just a 'requested' sequence, and will be
     * resolved (and ->seq updated) when the device is probed.
     */
     /* 这里uc和uc_drv已经勾搭上了，其实她们在uclass_get的时候就好上了
      * 回顾一下文章贴的第一段代码，那个sdmmc驱动里有个flags，在这里就能用上
      */
    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);
        }
    }
#endif
 if (!dev->platdata && drv->platdata_auto_alloc_size) {
        dev->flags |= DM_FLAG_ALLOC_PDATA;
        dev->platdata = calloc(1, drv->platdata_auto_alloc_size);
        if (!dev->platdata) {
            ret = -ENOMEM;
            goto fail_alloc1;
        }
    }
    if (parent) {
        int size = parent->driver->per_child_platdata_auto_alloc_size;

        if (!size) {
            size = parent->uclass->uc_drv->
```

```
                    per_child_platdata_auto_alloc_size;
        }
        if (size) {
            dev->flags |= DM_FLAG_ALLOC_PARENT_PDATA;
            dev->parent_platdata = calloc(1, size);
            if (!dev->parent_platdata) {
                ret = -ENOMEM;
                goto fail_alloc2;
            }
        }
    }

    /* put dev into parent's successor list */
    if (parent)
        list_add_tail(&dev->sibling_node, &parent->child_head);

    ret = uclass_bind_device(dev);
    if (ret)
        goto fail_uclass_bind;

    /* if we fail to bind we remove device from successors and free it */
    if (drv->bind) {
        ret = drv->bind(dev);
        if (ret)
            goto fail_bind;
    }
    if (parent && parent->driver->child_post_bind) {
        ret = parent->driver->child_post_bind(dev);
        if (ret)
            goto fail_child_post_bind;
    }

    if (parent)
        dm_dbg("Bound device %s to %sn", dev->name, parent->name);
    *devp = dev;

    return 0;

fail_child_post_bind:
    if (drv->unbind && drv->unbind(dev)) {
        dm_warn("unbind() method failed on dev '%s' on error pathn",
            dev->name);
    }

fail_bind:
    if (uclass_unbind_device(dev)) {
        dm_warn("Failed to unbind dev '%s' on error pathn",
            dev->name);
    }
fail_uclass_bind:
    list_del(&dev->sibling_node);
    if (dev->flags & DM_FLAG_ALLOC_PARENT_PDATA) {
        free(dev->parent_platdata);
        dev->parent_platdata = NULL;
    }
fail_alloc2:
    if (dev->flags & DM_FLAG_ALLOC_PDATA) {
        free(dev->platdata);
        dev->platdata = NULL;
    }
fail_alloc1:
    free(dev);

    return ret;
}
```

root_driver 由于是混沌初开，没有什么 parent，init 这些东西，所以直接返回 0 了。

2.1.2 device_probe

对于 root driver 来讲，没有什么动作，进入下一个重要部分。

2.2 dm_scan_platdata

这里先记录两个宏，其实前面已经用到过了，放这里提示一下

```
/* Cast away any volatile pointer */
#define DM_ROOT_NON_CONST      (((gd_t *)gd)->dm_root)
#define DM_UCLASS_ROOT_NON_CONST    (((gd_t *)gd)->uclass_root)
```

这个函数会通过`ll_entry_*`宏去搜索`driver_info`字段，但是链接时并没有，所以应该直接返回 0。我有点困惑，难道目前还没用?

2.3 dm_scan_fdt

移植的这个版本中，驱动信息都是通过设备树定义的，她里面调用了

```
/* drivers/core/root.c */
dm_scan_fdt_node(gd->dm_root, blob, 0, pre_reloc_only);
```

传进去四个参数：根设备 dm_root，设备树 blob，offset=0，pre_reloc_only=True。继续贴代码吧，因为这里是*重点*了。

```
#ifdef CONFIG_OF_CONTROL
int dm_scan_fdt_node(struct udevice *parent, const void *blob, int offset,
             bool pre_reloc_only)
{
    int ret = 0, err;

    // 遍历node，至于fdt_first_subnode怎么做的，这里可以不用关心
 for (offset = fdt_first_subnode(blob, offset);
         offset > 0;
         offset = fdt_next_subnode(blob, offset)) {
         // pre_reloc_only = True
     if (pre_reloc_only &&
            !fdt_getprop(blob, offset, "u-boot,dm-pre-reloc", NULL))
            continue;
        if (!fdtdec_get_is_enabled(blob, offset)) {
            dm_dbg("   - ignoring disabled devicen");
            continue;
        }
        err = lists_bind_fdt(parent, blob, offset, NULL);
        if (err && !ret)
            ret = err;
    }

    if (ret)
        dm_warn("Some drivers failed to bindn");

    return ret;
}
```

也不知道是不是眼睛有缺陷，有时候我会把 for 和 if 看混淆。在公司里的时候也遇到过这样的问题，还叫同事帮忙看，看了好久都没发现。这次看的时候又把第一个 for 看成 if 了，然后觉得表达式里的语法不对，不过还好很快就发现了。这个函数看名字就知道是干什么的，就是用来扫描设备树，提取各个有效 node 的。这里记录一点，对函数的理解我觉得可以分为三个层次:

1. 架构层次：放在架构里，知道她是干嘛的，可以顺利阅读框架代码。
2. 模块层次：知道这个函数输入和输出，并且输出和输入的对应关系，可以调试框架。
3. 实现层次：熟悉具体是怎么实现的，就是源代码怎么写的，可以调试模块。

心里知道当前工作所处的层次，比如现在只要知道这个函数是干嘛的，那就没必要去追究实现层次的东西，人的精力终归是有限的，什么阶段做什么事情，对不对？那么这三个层次各需要怎么做呢？首先是架构层次，因为虽然我们是要知道函数是干嘛的，但是知道怎么实现的，不就知道是干嘛的了吗？其实不是这样的，首先我们肯定知道我们当前的框架是用来干嘛的，然后进一步细分就知道模块是用来干嘛的，在此基础上，结合函数位置和名字就很容易知道这个函数是用来干嘛的，这是一种自顶向下的分析方式，但是从实现开始则是自底向上的。这里又要多嘴一句了，其实相对于细节，把握顶层设计更容易把握方向，代码如此，社会也是如此。明确政策和社会导向能让你更加容易切入一个领域，少走好多弯路，归根到底人的生命是有限的啊，人这一生就是在用有限的生命最大化地最好自己想做的事情。在设备树这一块上，这篇文章还处于架构层次，至于模块层次，还要另外写一篇文章分析，尽请期待。

2.4 dm_scan_other

是个__weak，直接返回 0。

No Reply at the moment.