

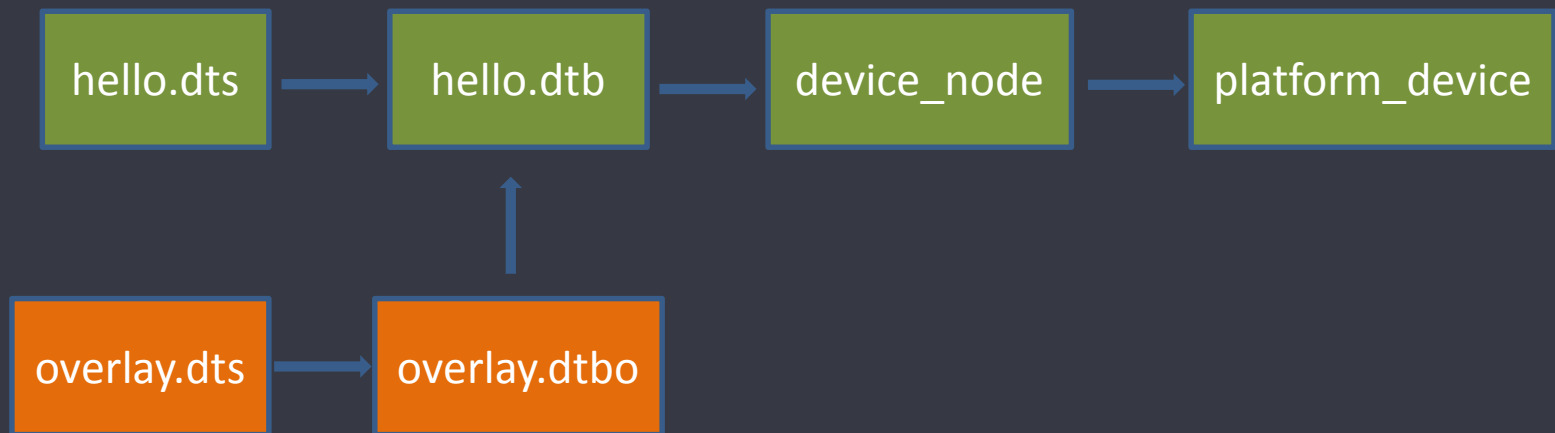
# Linux内核编程12期 设备树的overlay与ConfigFS

主讲: 王利涛

# 01 什么是设备树的overlay?

主讲: 王利涛

- 设备树的overlay功能
  - 在系统runtime期间修改设备树
  - 设备树的编译、加载和运行



- 需求与现状

- 外界插拔设备，无法在设备树中预先描述：耳机
- 树莓派 + FPGA开发板
- 基于I2C的温度传感器
- 管脚的重新配置：PIN multiplexing
- 修改bootcmd、分区
- 内核mainline还没支持，不同平台各自实现

- 本期课程的主要内容
  - 在开发板上如何实现设备树的overlay功能
  - Configfs文件系统的配置与挂载
  - Configfs编程接口
  - 如何编写设备树 overlay文件
  - 设备树 overlay的编译和运行
  - 设备树overlay运行机制分析

适合哪些人学习：

嵌入式驱动工程师

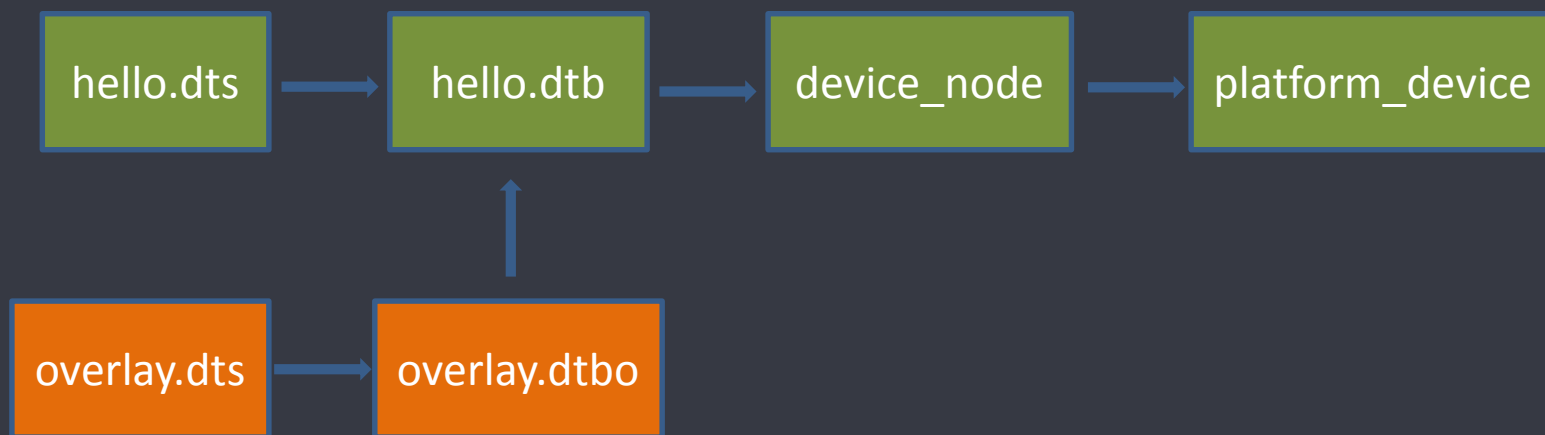
嵌入式BSP工程师

嵌入式软件工程师

想从事嵌入式软件开发的同学

## 02 设备树overlay实现原理分析

- 如何动态修改设备树？
  - 设备树的解析过程
  - 几个重要的函数
  - 如何将dtb文件加载到内核
  - 文件系统接口：proc、sysfs、configs
  - changset: 增改删



## 03 ConfigFS的编译与挂载



- ConfigFS文件系统简介
  - 基于RAM的内核对象管理器: `config_item`
  - 内核对象在用户空间的接口, 类似`proc/sysfs`
  - 亮点: 用户可以在用户空间创建内核对象
  - 源码: `linux-5.10.4/fs/configfs`

- 内核编译配置与挂载

.config - Linux/arm 5.10.4 Kernel Configuration

> File systems > Pseudo filesystems

#### Pseudo filesystems

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [\*] built-in [ ] excluded <M> module < > module capable

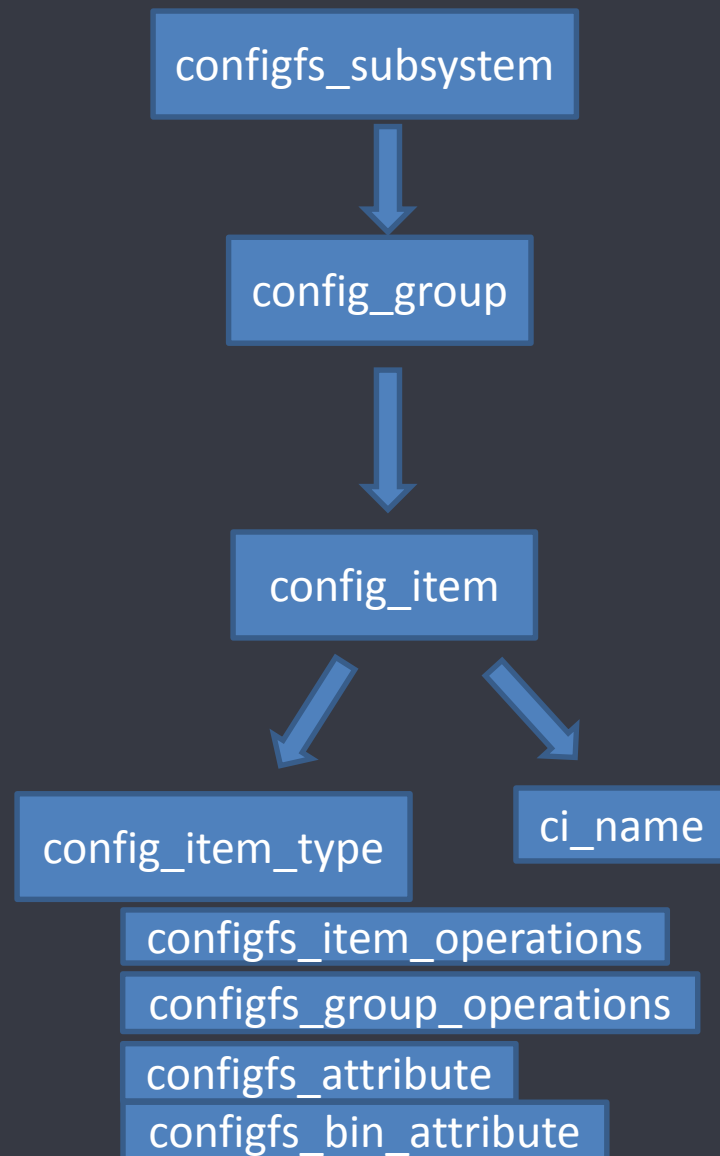
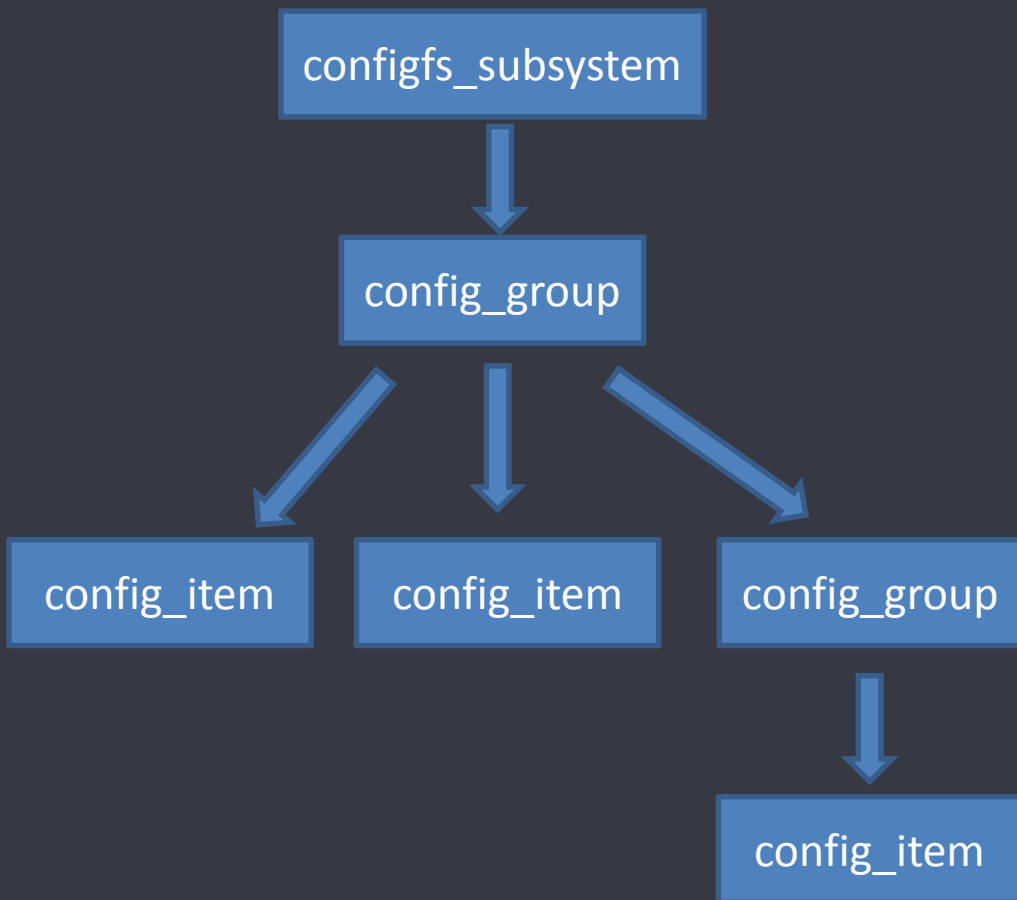
```
[ ] Include /proc/<pid>/task/<tid>/children file
[*] Tmpfs virtual memory file system support (former shm fs)
[ ]   Tmpfs POSIX Access Control Lists
[ ]   Tmpfs extended attributes
<*> Userspace-driven configuration filesystem
```

```
# mount -t configfs none /sys/kernel/config
```

	#device	mount-point	type	options	dump	fsck	order
2	proc	/proc	proc	defaults	0	0	
3	tmpfs	/tmp	tmpfs	defaults	0	0	
4	sysfs	/sys	sysfs	defaults	0	0	
5	tmpfs	/dev	tmpfs	defaults	0	0	
6	var	/dev	tmpfs	defaults	0	0	
7	ramfs	/dev	ramfs	defaults	0	0	
8	debugfs	/sys/kernel/debug	debugfs	defaults		0	0
9	configfs	/sys/kernel/config	configfs	defaults		0	0
10	tmpfs	/log	tmpfs	defaults	0	0	

# 04 ConfigFS的核心数据结构

- 几个关键的核心数据结构



- 子系统、容器和config\_item

```
struct configfs_subsystem {
    struct config_group su_group;
    struct mutex        su_mutex;
};

struct config_group {
    struct config_item    cg_item;
    struct list_head      cg_children;
    struct configfs_subsystem *cg_subsys;
    struct config_group    **default_groups;
    struct list_head      group_entry;
};

struct config_item {
    char                *ci_name;
    char                ci_namebuf[CONFIGFS_ITEM_NAME_LEN];
    struct kref          ci_kref;
    struct list_head     ci_entry;
    struct config_item    *ci_parent;
    struct config_group   *ci_group;
    struct config_item_type *ci_type;
    struct dentry         *ci_dentry;
};
```

# • 属性和方法

```
struct config_item_type {  
    struct module                *ct_owner;  
    struct configs_item_operations *ct_item_ops;  
    struct configs_group_operations *ct_group_ops;  
    struct configs_attribute      **ct_attrs;  
    struct configs_bin_attribute  **ct_bin_attrs;  
};
```

```
struct configs_attribute {  
    const char      *ca_name;    // 属性名字  
    struct module   *ca_owner;   // 属性所属模块  
    umode_t         ca_mode;     // 访问权限  
    ssize_t (*show)(struct config_item *, char *); // 属性的读写方法  
    ssize_t (*store)(struct config_item *, const char *, size_t);  
};
```

# 05 通过ConfigFS加载二进制文件

- 二进制属性和方法

```
struct config_item_type {  
    struct module                *ct_owner;  
    struct configs_item_operations *ct_item_ops;  
    struct configs_group_operations *ct_group_ops;  
    struct configs_attribute      **ct_attrs;  
    struct configs_bin_attribute  **ct_bin_attrs;  
};
```

```
struct configs_bin_attribute {  
    struct configs_attribute cb_attr;    /* std. attribute */  
    void *cb_private;                  /* for user */  
    size_t cb_max_size;                /* max core size */  
    ssize_t (*read)(struct config_item *, void *, size_t);  
    ssize_t (*write)(struct config_item *, const void *, size_t);  
};
```



# 06 创建ConfigFS子目录

## • 创建目录: make\_item

```
struct config_item {
    char                *ci_name;
    char                ci_namebuf[CONFIGFS_ITEM_NAME_LEN];
    const struct config_item_type *ci_type;
};

struct config_item_type {
    struct module                *ct_owner;
    struct configs_item_operations *ct_item_ops;
    struct configs_group_operations *ct_group_ops;
    struct configs_attribute      **ct_attrs;
    struct configs_bin_attribute  **ct_bin_attrs;
};

struct configs_item_operations {
    void (*release)(struct config_item *);
    int (*allow_link)(struct config_item *src, struct config_item *target);
    void (*drop_link)(struct config_item *src, struct config_item *target);
};

struct configs_group_operations {
    struct config_item *(*make_item)(struct config_group *group, const char *name);
    struct config_group *(*make_group)(struct config_group *group, const char *name);
    int (*commit_item)(struct config_item *item);
    void (*disconnect_notify)(struct config_group *group, struct config_item *item);
    void (*drop_item)(struct config_group *group, struct config_item *item);
};
```

# 07 创建多级ConfigFS子目录

- 创建目录: `make_group`

```
struct config_group {
    struct config_item    cg_item;
    struct list_head      cg_children;
    struct configs_subsystem *cg_subsys;
    struct config_group   **default_groups;
    struct list_head      group_entry;
};

struct config_item_type {
    struct module                *ct_owner;
    struct configs_item_operations *ct_item_ops;
    struct configs_group_operations *ct_group_ops;
    struct configs_attribute      **ct_attrs;
    struct configs_bin_attribute  **ct_bin_attrs;
};

struct configs_group_operations {
    struct config_item *(*make_item)(struct config_group *group, const char *name);
    struct config_group *(*make_group)(struct config_group *group, const char *name);
    int (*commit_item)(struct config_item *item);
    void (*disconnect_notify)(struct config_group *group, struct config_item *item);
    void (*drop_item)(struct config_group *group, struct config_item *item);
};
```

- 层次结构的构建
  - config\_group->cg\_children
  - config\_item->ci\_parent

# 08 ConfigFS mkdir过程分析

- 当我们在用户空间mkdir时...
  - 源码目录: fs/configfs/dir.c: configfs\_mkdir
  - 调用config\_item\_type实现的方法
    - » 执行make\_group分支: 创建configfs\_group
    - » 执行make\_item分支: 创建item
  - 创建目录
    - make\_group分支:
      - » configfs\_attach\_group ->
      - » configfs\_attach\_item 创建目录
    - make\_item分支:
      - » configfs\_attach\_item ->
      - » configfs\_create\_dir(item, dentry, frag) 创建目录

## • 小结

- 若一个`config_item_type`中，定义了`make_group`方法，创建目录时会调用该方法
- 没有定义`make_group`方法，创建目录时，会调用`make_item`方法
- 两者都没定义，无法创建子目录
- 若实现了`make_group`方法，会创建子`group`，每个子`config_group`对应一个目录，有各自的`config_item`
- 每个 `config_group` 下可以创建多个同级子目录，子目录对应的 `config_item` 使用链表管理
- 层级关系通过 `config_group->cg_children` 和 `config_item->ci_parent` 指针构建



# 09 实现设备树的overlay功能

- 如何实现设备的overlay?
  - 内核配置
  - 加载
  - 解析
  - 驱动源码分析
  - 编译运行

```
.config - Linux/arm 5.10.4 Kernel Configuration
> Device Drivers > Device Tree and Open Firmware support -----
                                Device Tree and Open Firmware support
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----).
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in
[ ] excluded <M> module < > module capable

-- Device Tree and Open Firmware support
[ ] Device Tree runtime unit tests
[*] Device Tree overlays
```

```
# insmod demo.ko
# cd sys/kernel/config/device-tree/overlays
```

- 设备树的overlay内核配置
  - CONFIG\_OF\_OVERLAY: 使能overlay功能
  - CONFIG\_OVERLAY\_FS

```

.config - Linux/arm 5.10.4 Kernel Configuration
# File systems

File systems
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----).
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in
[ ] excluded <M> module < > module capable

^(-)
[ ] Ext4 debugging support
[ ] JBD2 (ext4) debugging support
< > Reiserfs support
< > JFS filesystem support
< > XFS filesystem support
< > GFS2 file system support
< > OCFS2 file system support
< > Btrfs filesystem support
< > NILFS2 file system support
< > F2FS filesystem support
[ ] Enable filesystem export operations for block IO
-*- Enable POSIX file locking API
[*] Enable Mandatory file locking
[ ] FS Encryption (Per-file encryption)
[ ] FS Verity (read-only file-based authenticity protection)
[*] Dnotify support
[*] Inotify support for userspace
[ ] Filesystem wide access notification
[ ] Quota support
< > Old Kconfig name for Kernel automounter support
< > Kernel automounter support (supports v3, v4 and v5)
< > FUSE (Filesystem in Userspace) support
<M> Overlay filesystem support
-*- Overlayfs: turn on redirect directory feature by default
[*] Overlayfs: follow redirects even if redirects are turned off
[*] Overlayfs: turn on inodes index feature by default
[*] Overlayfs: turn on metadata only copy up feature by default
Caches --->
CD-ROM/DVD Filesystems --->

```

# 10 向设备树动态添加节点（上）

- 编写第一个设备树overlay文件
  - overlay语法
  - 编译
  - 加载

```
# insmod demo.ko
# cd sys/kernel/config/device-tree/overlays
# mkdir hello
# ls hello
  dtbo status
# cat /hello.dtbo > hello/dtbo
# echo 1 > hello/status
# ls /proc/device-tree
hello
```

```
# echo 1 > hello/status
# rmdir hello
# rmmod demo.ko
```

# 11 向设备树动态添加节点（下）

- 反编译设备树的overlay文件
  - 反编译后的设备树语法分析

```
/plugin/;
/{
    /* set of per-platform overlay manager properties */

    fragment@0 {
        target = <&target-label>; /* or target-path */

        __overlay__ {
            /* contents of the overlay */
        };
    };

    fragment@1 {
        /* second overlay fragment... */
    };
};
```

# 12 设备树overlay加载过程分析



- 加载过程分析
  - 顶层目录的创建
  - 用户动态创建目录
  - 加载dtb文件过程
  - 驱动源码分析

# 13 设备树overlay解析过程分析

- 内核如何解析新加载的dtb文件
  - 控制开关: status
  - 申请内存, 将dtbo展开为device\_node
  - 处理label/phandle引用, 找到真正的node
  - 创建和处理overlay changeset: 增改删
  - 内核API

of\_fdt\_unflatten\_tree: 将 dtb 展开为 device\_node  
of\_resolve\_phandles : 动态调整phandle值  
of\_overlay\_create : Creates and applies an overlay  
of\_overlay\_destroy

# 14 同时加载多个设备树overlay

- 实验
  - 同时加载多个dtbo文件
  - 动态修改某个属性