Linux 内核启动流程

----设备树的识别　曹忠明

ARM Linux 内核在 Linux-3.x 内核有了很大的变化，对一些新的平台的支持取消了传统的设备文件而用设备树取代，这里以 FS4412 设备树识别为例说明 Linux 是如何识别设备树的。

1、 设备树文件

FS4412 设备树文件(exynos4412-fs4412.dts)部分内容：

```
/*
 * Insignal's Exynos4412 based Origen board device tree source
 *
 * Copyright (c) 2012-2013 Samsung Electronics Co., Ltd.
 *      http://www.samsung.com
 *
 * Device tree source file for Insignal's Origen board which is based on
 * Samsung's Exynos4412 SoC.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
*/

/ dts - v1 /;
#include "exynos4412.dtsi"

/
{
    model = "Insignal Origen evaluation board based on Exynos4412";
    compatible = "insignal,origen4412", "samsung,exynos4412";

    memory {
        reg = <0x40000000 0x40000000>;
    };

    chosen {
        bootargs = "root=/dev/nfs nfsroot=192.168.9.120:/source/rootfs init=/linuxrc console=ttySA
    };

    firmware@0203F000 {
        compatible = "samsung,secure-firmware";
```

```
        reg = <0x0203F000 0x1000>;
    };

    srom - cs1@5000000 {
        compatible = "simple-bus";
#address-cells = <1>;
#size-cells = <1>;
        reg = <0x5000000 0x1000000>;
        ranges;

        ethernet@5000000 {
            compatible = "davicom,dm9000";
            reg = <0x5000000 0x2 0x5000004 0x2>;
            interrupt - parent = <&gpx0>;
            interrupts = <6 4>;
            davicom, no - eeprom;
            mac - address = [00 0a 2d a6 55 a2];
        };
    };



    regulators {
        compatible = "simple-bus";
#address-cells = <1>;
#size-cells = <0>;

mmc_reg:
        regulator@0 {
            compatible = "regulator-fixed";
            reg = <0>;
            regulator - name = "VMEM_VDD_2.8V";
            regulator - min - microvolt = <2800000>;
            regulator - max - microvolt = <2800000>;
            gpio = <&gpx1 1 0>;
            enable - active - high;
        };
    };

    g2d@10800000 {
        status = "okay";
    };

    sdhci@12530000 {
```

```
76          bus - width = <4>;
77          pinctrl - 0 = <&sd2_clk &sd2_cmd &sd2_bus4 &sd2_cd>;
78          pinctrl - names = "default";
79          vmmc - supply = <&mmc_reg>;
80          status = "okay";
81      };
82      ......
83  };
```

编译设备树文件，内核顶层目录下执行如下命令可以编译设备树文件：

```
$ make dtbs
```

编译后生成文件为 exynos4412-fs4412.dtb，dtb 文件是使用大端字节序存储，显示其内容如下：

```
$ hexdump exynos4412-fs4412.dtb
```

内容如下：

```
0000000 0dd0 edfe 0000 a588 0000 3800 0000 0c82
0000010 0000 2800 0000 1100 0000 1000 0000 0000
0000020 0000 9906 0000 d481 0000 0000 0000 0000
0000030 0000 0000 0000 0000 0000 0100 0000 0000
0000040 0000 0300 0000 0400 0000 0000 0000 0100
0000050 0000 0300 0000 0400 0000 0f00 0000 0100
0000060 0000 0300 0000 0400 0000 1b00 0000 0100
0000070 0000 0300 0000 2700 0000 2c00 6e69 6973
0000080 6e67 6c61 6f2c 6972 6567 346e 3134 0032
0000090 6173 736d 6e75 2c67 7865 6e79 736f 3434
.....
0008880 006e 6276 6361 2d6b 6f70 6372 0068 6676
0008890 6f72 746e 702d 726f 6863 7600 7973 636e
00088a0 6c2d 6e65 0000
00088a5
```

在 uboot 引导内核之前，会将设备树文件加载到内存中，以备 Linux 内核使用，这里就不详细说明了

2、 Linux 内核启动

Linux 内核启动分几个阶段：

1) Linux 内核自解压
2) Linux 内核初始化----汇编
3) Linux 内核初始化----C

这里从第三阶段开始说明，分析这个阶段，主要是查看函数 start_kernel，在

start_kernel 中有几个函数这里重点分析：

2.1 setup_arch

```
setup_arch(&command_line);
void __init setup_arch(char **cmdline_p)
{
        const struct machine_desc *mdesc;
        …
        mdesc = setup_machine_fdt(__atags_pointer);
        if (!mdesc)
                mdesc = setup_machine_tags(__atags__pointer, __machine_arch_type);
        …
}
```

setup_machine_fdt：

struct machine_desc *setup_machine_fdt(unsigned int dt_phys)函数是用来识别设备树，Linux 内核中是这样描述这个函数的：

```
/*
 * Machine setup when an dtb was passed to the kernel,dt_phys
 * @dt_phys: physical address of dt blob
 *
 * If a dtb was passed to the kernel in r2, then use it to choose the correct machine_desc
 * and to setup the system
 */
```

也就是说 bootloader 如果将一个设备树文件加载到内存中，其通过 r2 寄存器将设备树的物理地址传递到 Linux 内核中，Linux 内核来选择正确的机器且对其进行设置

setup_machine_tags：

这函数是在 Linux 内核不使用设备树的情况是使用，这里就不分析了。

setup_machine_fdt：

```
cosnt struct machine_desc *__init setup_machine_fdt(unsigned int dt_phys)
{
        ……
        mdesc = of_flat_dt_match_machine(mdesc_best, arch_get_next_mach);
        if (!mdesc) {
                dt_root = of_get_flat_dt_root();
                prop = of_get_flat_dt_prop(dt_root, “compatible”, &size);
        }
        ……
        return mdesc;
}
```

函数中需要重点分析的函数有:of_flat_match_machine 和 of_get_flat_dt_root

of_flat_match_machine：

```
const void * __init of_flat_dt_match_machine(const void *default_match
        const void * (*get_next_compat)(const char * const **))
{
    ……

    /*
     * 找到设备树中的根节点(开始结点)
     */
    dt_root = of_get_flat_dt_root();

    /*
     * get_next_compat = arch_get_next_mach
     * 内核中可以同时支持多个平台，这个函数用来获得下一个平台的
     * dt_compat 属性，结合上边函数克制 data 内容
     */

    while((data = get_next_compat(&compat))) {
    /*
     * of_flat_dt_match 用来匹配设备树中内容和内核所支持平台是否匹配
     */
    score = of_flat_dt_match(dt_root, compat);
    if (score > 0 && score < best_score) {
        best_data = data;
        best_score = score;
        }
    }

    if (!best_data) {
        const char *prop;
        long size;
    /*
     * 设备树中内容和内核所支持平台没有匹配则打印出错提示
     */
    pr_err("\n unrecognized device tree list: \n");
    prop = of_get_flat_dt_prop(dt_root, "compatible", &size);
    if (prop) {
        while (size > 0) {
        printk("'%s'", prop);
        size -= strlen(prop) + 1;
        prop += strlen(prop) + 1;
        }
        printk("]\n\n");
        return NULL;
```

```
        }
        }
    }
```

函数中会调用函数 of_get_flat_dt_root 和 get_next_compat，of_get_flat_dt_root 用来从设备树文件中找到根节点，get_next_compat 按照上下文这个函数等于 arch_get_next_mach，下边会重点分析这两个函数。

of_get_flat_dt_root：

补充：设备树相关宏和结构体：

结点相关宏：

```
OF_DT_HEADER            0xd00dfeed 标记
OF_DT_BEGIN_NODE        0x1   开始结点
OF_DT_END_NODE          0x2   结束结点
OF_DT_PROP              0x3   Property: name off, size, *content
                        资源
OF_DT_NOP               0x4   NOP
OF_DT_END               0x9   结束
OF_DT_VERSION           0x10  版本
```

相关结构体：

```
struct boot_param_header {
        __be32 magic;                   // OF_DT_HEADER 幻数
        __be32 totalsize;               // 设备树总体大小
        __be32 off_dt_struct;           // structure 偏移
        __be32 off_dt_strings;          // strings 偏移
        __be32 off_mem_rsvmap;          // 内存预留映射表偏移
        __be32 version;                 // 格式版本
        __be32 last_comp_version;       // 最后兼容版本
        __be32 boot_cpuid_phys;         // 我们要启动的 CPU 的 ID
        __be32 dt_strings_size;         // 设备树 strings 块大小
        __be32 dt_struct_size;          // 设备树 structure 块大小
};
```

设备树前 40 个字节就是 boot_param_header,设备树采用大端存储，所以显示内容如下，使用时 ARM 多设置为小端存储，所以需要使用 be32_to_cpu 将大端字节序转换为本地字节序，显示如下：

```
0000000 0dd0 edfe 0000 a588 0000 3800 0000 0c82
0000010 0000 2800 0000 1100 0000 1000 0000 0000
0000020 0000 9906 0000 d481 0000 0000 0000 0000
```

```
struct boot_param_header {
        __be32 magic;                   // 0x0dd0 0xedfe ==> le 0xd00d 0xfeed
        __be32 totalsize;               // 0x0000 0xa588 ==> le 0x0000 0x88a5
        __be32 off_dt_struct;           // 0x0000 0x3800 ==> le 0x0000 0x0038
```

```
    __be32 off_dt_strings;              // 0x0000 0x0c82 ==> le 0x0000 0x820c
    __be32 off_mem_rsvmap;                  // 0x0000 0x2800 ==> le 0x0000 0x0028
    __be32 version;                         // 0x0000 0x1100 ==> le 0x0000 0x0011
    __be32 last_comp_version;               // 0x0000 0x1000 ==> le 0x0000 0x0010
    __be32 boot_cpuid_phys;                 // 0x0000 0x0000 ==> le 0x0000 0x0000
    __be32 dt_strings_size;                 // 0x0000 0x9906 ==> le 0x0000 0x6099
    __be32 dt_struct_size;                  // 0x0000 0xd481 ==> le 0x0000 0x81d4
};
```

```
unsigned long __inti of_get_flat_dt_root(void)
{
    unsigned long p = ((unsigned long)initial_boot_params) +
    be32_to_cpu(initial_boot_params->off_dt_struct);
    /*
    *  结合上述内容 p =  设备树起始地址  + 0x0038
    *  设备树文件按 16 进制显示：
    * 0000030 0000 0000 0000 0000 0000 0100 0000 0000
    * 0x38 = 0x00000001
    */

    /*
     *  跳过所有无效数据
     */
    while(be32_to_cpup((__be32 *)p) == OF_DT_NOP)
    p += 4;
    /*
     * p = OF_DT_BEGIN_NODE  表示找到开始节点，也就是设备树有效数据的开始
     */
    BUG_ON(be32_to_cpup((__be32 *)p) != OF_DT_BEGIN_NODE);
    p += 4;

    /*
     *  数据对齐
     */
    return ALIGN(p + strlen(char *p) + 1, 4);
}
```

arch_get_next_mach：

```
static const void * __init arch_get_next_mach(const    char *const **match)
{
    static const struct machine_desc *mdesc = __arch_info_begin;

    if (m > __arch_info_end)
```

```
        return NULL;

        mdesc++;
        *match = m->dt_compat;

        return m;
    }
```

内核中又一个段叫.arch.info.init，内核在编译的时候会将所有支持的平台的信息链接到这个段中即 machine_desc，其中__arch_info_begin 表示这个段的开始__arch_info_end 表示这个段的结束。

连接脚本参考 arch/arm/kernel/vmlinux.lds 中有如下内容：

```
.init.arch.info: {
   __arch_info_begin = .;
   *(.arch.info.init)
   __arch_info_end = .;
}
```

machine_desc 注册参考 arch/arm/mach-exynos/mach-exynos4-dt.c 中

```
static void __init exynos4_dt_machine_init(void) {
      exynos_cpuidle_init();
      exynos_cpufreq_init();

      of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL);
}

static char const *exynos4_dt_compat[] __initdata = {
    "samsung,exynos4210",
    "samsung,exynos4212",
    "samsung,exynos4410",
    NULL
};

DT_MACHINE_START(EXYNOS4410_DT, "Samsung Exynos4 (Flattened Device Tree)")
    .dt_compat = exynos4_dt_compat,
    .init_machine = exynos4_dt_machine_init,
    ....
MACHINE_END
```

DT_MACHINE_START 的含义就是定义结构体 machine_desc 并初始化且标识链接到.arch.info.init 段中，定义如下:

```
#define DT_MACHINE_START(_name, _namestr)                \
static const struct machine_desc __mach_desc_##_name    \
__used                                                   \
```

```
__attribute__((__section__(".arch.info.init"))) = {}            \
.nr = ~0,                                                       \
.name = _namestr,
```

上述内容主要是通过读取设备树信息，检测当前使用内核是否支持本平台，如果支持则做相应的初始化，这里主要分析平台识别过程。

2.2 rest_init

reset_init 函数中创建了几个内核线程，其中 kernel_init 使我们要重点分析的。其中 kernel_init ----> kernel_init_freeable -----> do_basic_setup ----> do_initcall

```
static void __init do_initcalls(void)
{
    int level;
    for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++)
        do_initcall_level(level);
}
initcall_levels:
static initcall_t *initcall_levels[] __initdata = {
    __initcall0_start,
    __initcall1_start,
    __initcall2_start,
    __initcall3_start,
    __initcall4_start,
    __initcall5_start,
    __initcall6_start,
    __initcall7_start,
    __initcall_end,
}
```

在 Linux 内核连接文件 arch/arm/kernel/vmlinux.lds 文件中有如下内容：

```
__initcall_start = .; *(.initcallearly.init) __initcall0_start = .; *(.initcall0.init) *(.initcall0s.init) __initcall1_start = .; *(.initcall1.init) *(.initcall1s.init) __initcall2_start = .; *(.initcall2.init) *(.initcall2s.init) __initcall3_start = .; *(.initcall3.init) *(.initcall3s.init) __initcall4_start = .; *(.initcall4.init) *(.initcall4s.init) __initcall5_start = .; *(.initcall5.init) *(.initcall5s.init) __initcallrootfs_start = .; *(.initcallrootfs.init) *(.initcallrootfss.init) __initcall6_start = .; *(.initcall6.init) *(.initcall6s.init) __initcall7_start = .; *(.initcall7.init) *(.initcall7s.init) __initcall_end = .;
```

链接文件定义了若干个段，在内核编译时不同的函数会连接到不容的段中，比如 Linux 内核会创建一个函数指针指向初始化函数且链接到 initcall 段中，initcall 段中的 0~7 数字表示优先级，也就是将来这些初始化函数执行的先后。inicall 段中的函数在 Linux 内核启动过程中也就是 do_initcalls 执行的时候被调用。

```
for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++)
    do_initcall_level(level);
```

的作用就是按照先后执行所有连接到 initcall 段中的函数。那么这和设备树有什么关系

呢，接下来我们再分析一个函数。

arch/arm/kernel/setup.c

```
static init customize_machine(void)
{
    machine_desc->init_machine()
}
arch_initcall(customize_machine)
```

arch_initcall 的作用是定义函数指针变量指向 customize_machine 并将此函数指针连接到 initcall 段。

customize_machine 会在系统启动被调用

查看 customize_machine 中实际上是调用了 machine_desc->init_machine 函数也就是前文注册的 machine_desc 中的：

```
.init_machine = exynos4_dt_machine_init,
```

即：

```
static void __init exynos4_dt_machine_init(void) {
    exynos_cpuidle_init();
    exynos_cpufreq_init();

    of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL);
}
```

设备树的转换主要通过 of_platform_populate；

在内核源码中关于这个函数的描述是这样的：

of_platform_populate() - Populate platform_devices from device tree data。

也就是说 of_platform_polulate 会将相应结点转换为 platform_deivce,而且下文代码中的 of_platform_bus_create 是一个递归调用逐次便利各级子节点完成相应的转换，这里就不一一分析，先要了解整个转换过程可以跟进去看看。

原形：

```
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;

    //判断是否是根节点
    root = root ? of_node_get(root) : of_find_node_by_path("/");
    if (!root)
        return -EINVAL;

    //遍历结点
```

```
        for_each_child_of_node(root, child) {
            //转换为 platform_devie 并遍历本节点的子节点
            rc = of_platform_bus_create(child, matches, lookup, parent, true);
            if (rc)
                break;
        }
            of_node_put(root);
        return rc;
    }
```

由于技术有限，本文只是说明本人的理解，如有纰漏请见谅，并且提出宝贵意见！