

Linux 设备驱动开发

I2C 驱动框架梳理

张健宁

2022-8-26

- 1. 前言导引
- 2. Linux中I2C驱动架构总览
 - 2.1. I2C核心
 - 2.2. I2C总线驱动
 - 2.3. I2C设备驱动
- 3. Linux中I2C驱动框架代码结构总览
 - 3.1. 关键文件路径
 - 3.1.1. 核心层和总线驱动
 - 3.1.2. 设备驱动(以gt1x触摸屏为例)
 - 3.1.3. 设备树文件(以TB-RK3568X为例)
 - 3.1.4. 重要的头文件
 - 3.2. 关键数据结构
 - 3.2.1. 定义
 - 3.2.2. 关联
- 4. Linux中I2C驱动框架代码流程分析
 - 4.1. 注册I2C子系统核心层(主要是注册I2C总线)
 - 4.2. 注册i2c_adapter并将其添加到I2C总线
 - 4.2.1. 注册i2c_adapter到platform总线
 - 4.2.2. 将i2c_adapter添加到I2C总线
 - 4.2.3. 总结
 - 4.3. I2C设备驱动开发
 - 4.3.1. 将I2C控制器暴露给应用的方式
 - 4.3.2. 将I2C控制器抽象成公共驱动的方式
 - 4.4. 重点分析
 - 4.4.1. 何时调用match()函数? 何时调用probe()函数?
 - 4.4.2. 设备树匹配机制
- 5. 结语
- 6. 参考资料

1. 前言导引

I2C总线是Philips公司开发的一种简单、双向二线制同步串行总线，只需要两根线即可传送信息。I2C结合了SPI和UART的优点，可以像SPI一样将多个从机连接到单个主机，也可以使用多个主机控制一个或多个从机。

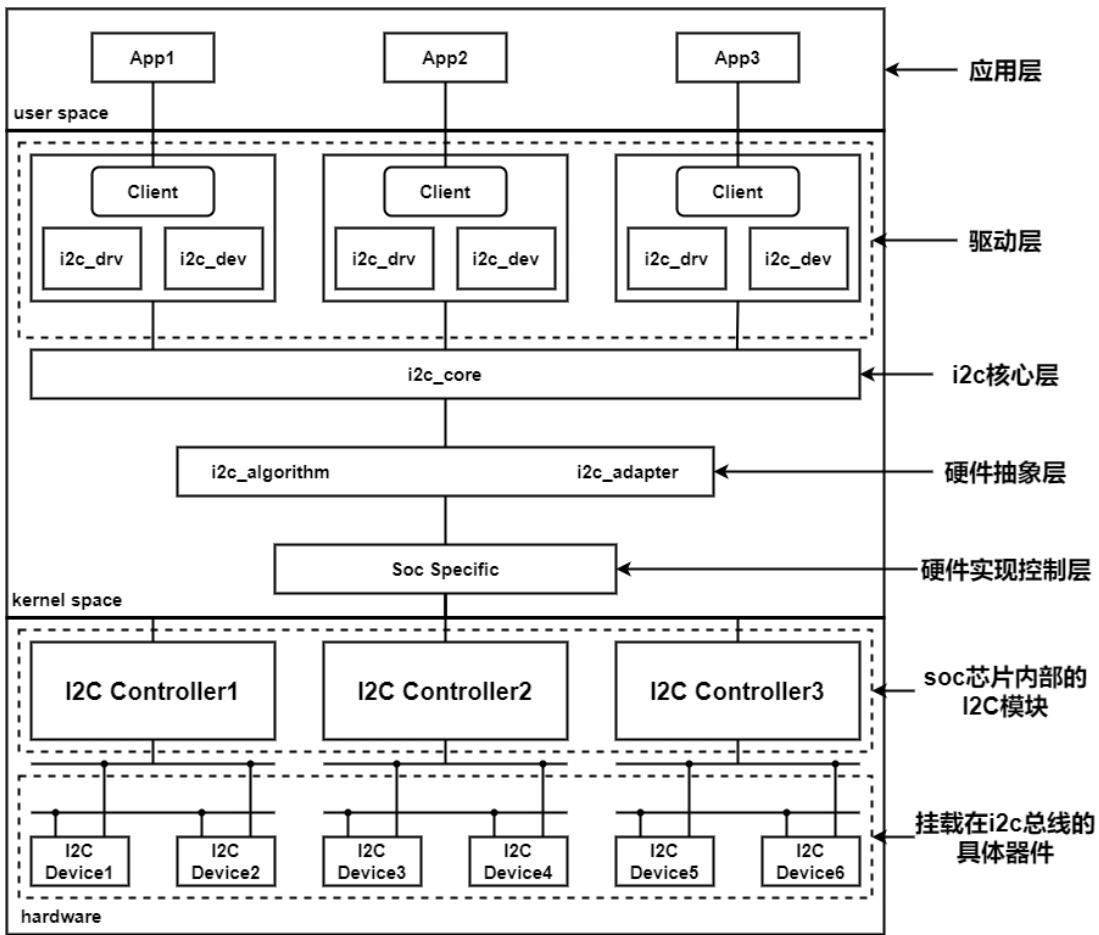
由于本文重点是梳理Linux中I2C驱动框架，所以对于I2C的通信协议、物理总线等基础内容不做过多赘述，读者可自行百度了解。

接下来，本文将从I2C驱动架构总览、I2C驱动框架代码结构总览、I2C驱动框架代码流程分析三个部分对I2C驱动框架进行梳理。

- I2C驱动架构总览主要介绍Linux中I2C驱动的整体架构，讲解各个组成部分的功能和相互联系。
- I2C驱动框架代码结构总览主要介绍Linux中I2C驱动框架的重要文件路径和关键数据结构。
- I2C驱动框架代码流程分析主要介绍Linux中I2C驱动框架的实现细节。

2. Linux中I2C驱动架构总览

下图展示了Linux中I2C驱动架构的基本架构：



对于南向开发而言，只需关注架构的内核空间部分。在《Linux设备驱动开发详解》一书第15章《Linux I2C核心、总线与设备驱动》中，将Linux内核里的I2C子系统分为核心、总线驱动和设备驱动三部分。

2.1. I2C核心

I2C核心提供了I2C总线驱动和I2C设备驱动注册和注销的方法，I2C通信方法上层的与具体适配器无关的代码，以及探测设备、检测设备地址的上层代码。I2C总线驱动和设备驱动之间依赖于I2C核心作为纽带。

2.2. I2C总线驱动

I2C总线驱动是对SoC中I2C控制器的软件实现(**i2c_algorithm**)。提供I2C控制器与从设备间完成数据通信的能力(**i2c_adapter**)。对应软件架构图中硬件抽象层部分和硬件实现控制层。

2.3. I2C设备驱动

I2C设备驱动(客户驱动)是对I2C从设备的软件实现。对应软件架构图中的驱动层。

3. Linux中I2C驱动框架代码结构总览

3.1. 关键文件路径

3.1.1. 核心层和总线驱动

```
~/Documents/OpenHarmony/out/kernel/src_tmp/linux-5.10/drivers/i2c/
├── algos           // i2c_algorithm相关, 通信算法
├── busses          // i2c_adapter相关, 已经编写好的各种向i2c核心层注册的适配器, 与I2C总线驱动相关
├── muxes
├── i2c-boardinfo.c // i2c静态声明i2c设备的文件, 设备树出现后已经不太使用。
├── i2c-core-acpi.c // 以下i2c-core-*.c对应老版本的i2c-core.c, 对应I2C核心, 由内核开发者实现的, 与硬件无关的
├── i2c-core-base.c
├── i2c-core.h
├── i2c-core-of.c
├── i2c-core-slave.c
├── i2c-core-smbus.c
├── i2c-dev.c        // 为i2c_adapter实现了设备文件功能, 只是提供了通用的read()、write()和ioctl()等接口, 提供
├── i2c-mux.c
├── i2c-slave-eeprom.c
├── i2c-slave-testunit.c
├── i2c-smbus.c      // 实现smbus协议的扩展文件
├── i2c-stub.c
└── Kconfig
└── Makefile
```

3.1.2. 设备驱动(以gt1x触摸屏为例)

```
~/Documents/OpenHarmony/out/kernel/src_tmp/linux-5.10/drivers/input/touchscreen/gt1x/
├── gt1x.c           // gt1x触摸屏设备驱动主要代码位置
├── gt1x_cfg.h
├── gt1x_extents.c
├── gt1x_firmware.h
├── gt1x_generic.c    // gt1x触摸屏设备驱动主要代码位置
├── gt1x_generic.h
├── gt1x.h
├── gt1x_tools.c
├── gt1x_update.c
└── GT5688_Config_20170713_1080_1920.cfg
└── Makefile
```

3.1.3. 设备树文件(以TB-RK3568X为例)

```
~/Documents/OpenHarmony/out/kernel/src_tmp/linux-5.10/arch/arm64/boot/dts/rockchip/
├ ...
├ rk3568-toybrick-x0-linux.dts
├ rk3568.dtsci
├ rk3568-linux.dtsci
└ rk3568-toybrick-mipi-tx0-beiqicloud.dtsci
└ ...
```

3.1.4. 重要的头文件

```
~/Documents/OpenHarmony/out/kernel/src_tmp/linux-5.10/include/linux/
├ ...
├ device.h
├ ...
└ i2c.h
├ of.h
└ of_device.h
└ ...
```

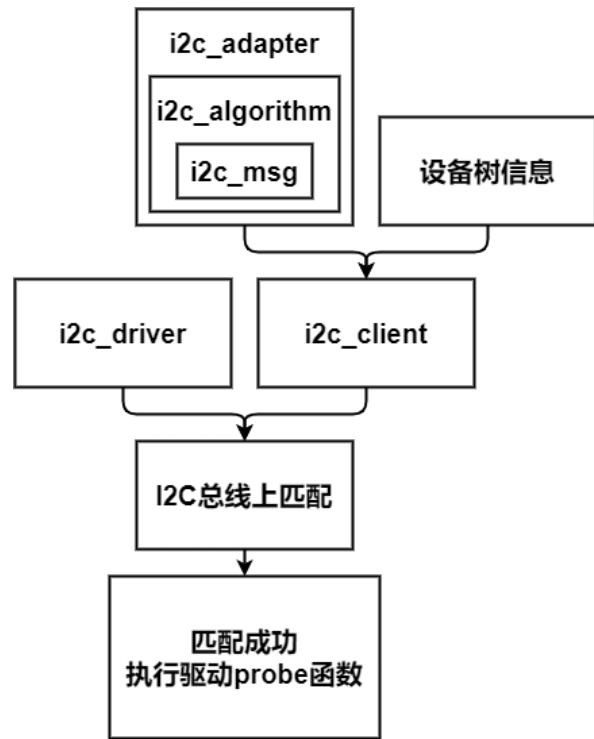
3.2. 关键数据结构

3.2.1. 定义

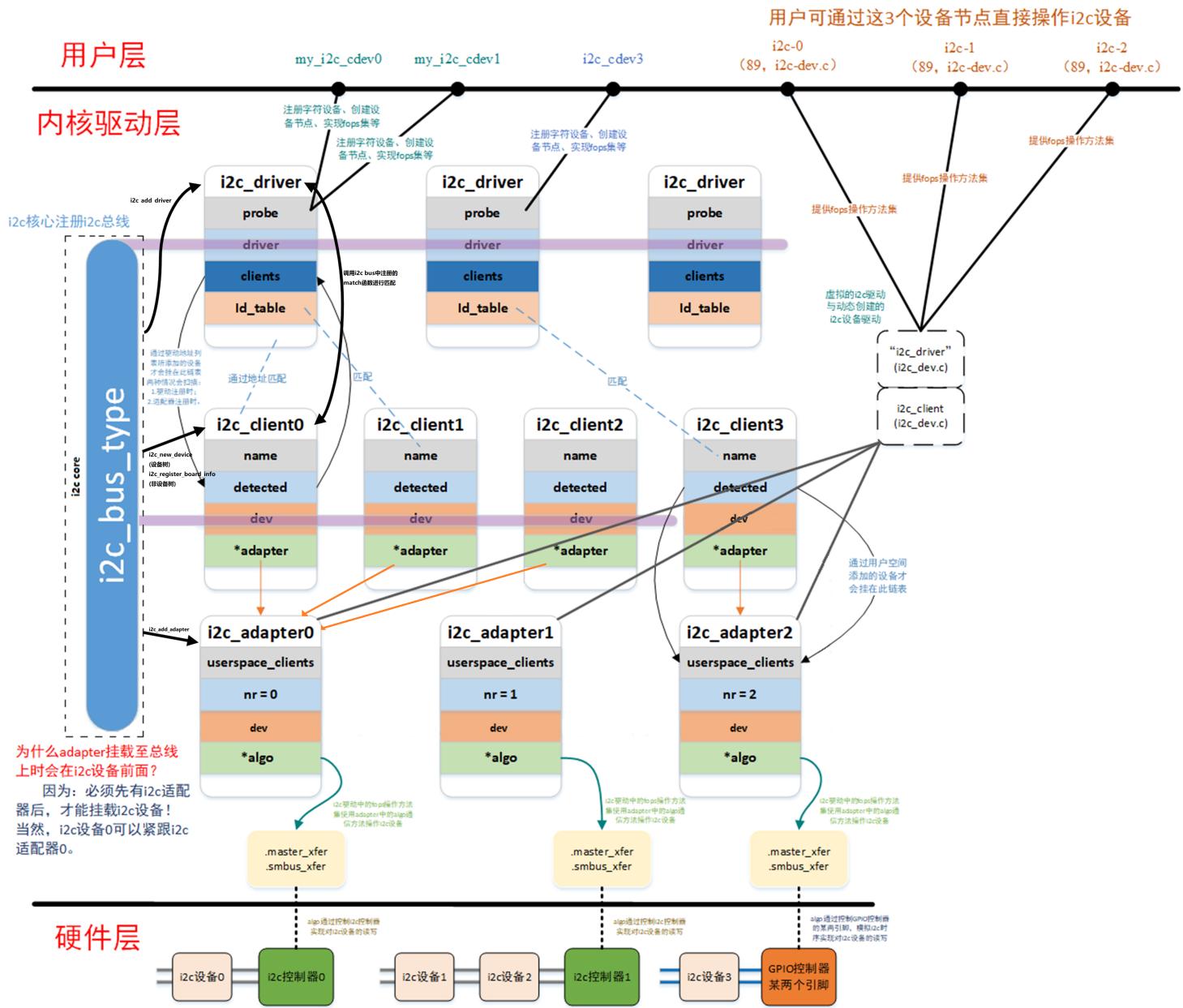
数据结构	文件路径	描述
i2c_adapter	/drivers/i2c/busses/i2c-core-base.c	用于识别物理I2Cs总线以及访问该总线所需的访问算法
i2c_algorithm	/include/linux/i2c.h	I2C通信方法
i2c_msg	/usr/include/linux/i2c.h	i2c_algorithm中通信函数的基本单位
i2c_driver	/include/linux/i2c.h	I2C设备驱动
i2c_client	/include/linux/i2c.h	I2C从机设备
i2c_bus_type	/drivers/i2c/i2c-core-base.c	I2C总线

3.2.2. 关联

下图解释了上述结构体之间的关联。在I2C设备驱动注册的过程中，会调用i2c_driver的匹配函数 `match()` 与i2c_client(在i2c_adapter注册过程中解析设备树信息生成)进行匹配，匹配成功则调用 `probe()` 函数完成驱动注册的收尾工作。设备驱动可通过i2c_adapter中提供的i2c_algorithm，构造i2c_msg与I2C设备通信，完成控制操作。



4. Linux中I2C驱动框架代码流程分析



每一个*i2c_adapter*对应一条实际的I2C总线。I2C总线上挂载着多个I2C设备实物，每个I2C设备对应一个*i2c_client*。一般来说，一个*i2c_driver*可以匹配多个*i2c_client*，而一个*i2c_client*只能匹配一个*i2c_driver*。*i2c_driver*会给每个I2C设备注册设备节点(以字符设备节点为例)，向用户层提供标准操作接口，如 `write()`/`read()`/`ioctl()`。通过调用*i2c_adapter*提供的通信方法*i2c_algorithm*，完成对I2C设备的操作。

根据上图展示的I2C驱动框架可以总结出构建流程如下：

4.1. 注册I2C子系统核心层(主要是注册I2C总线)

如果要在系统启动时便能享受到I2C总线的相关服务，就需要调用 `i2c_init()` 函数。该函数通过 `postcore_initcall(i2c_init)` 被放置在内核中的 `initcall2.init` 段处，这和驱动开发中调用 `module_init()` 类似。于是，在内核启动时，系统调用 `do_initcall()` 函数，根据指针数组 `initcall_levels[2]` 找到 `_initcall2_start` 指针。由 `vmlinux.lds.h` 可知，该指针指向 `initcall2.init` 段的起始地址，系统依次取出段中的每个函数指针并执行，从而使得系统能享受到I2C总线的相关服务。同样，编译进内核的驱动程序都通过这种方式完成启动并提供服务，具体可以查询“Linux的initcall机制”进行了解。

```
/* /drivers/i2c/i2c-core-base.c */
static int __init i2c_init(void)
{
    int retval;
    ...
    retval = bus_register(&i2c_bus_type);           // 注册I2C总线，成功返回值为0
    if (retval)           // 如果注册失败
        return retval;
    ...
    return 0;
...
}
```

`i2c_init()` 函数中，最重要的便是调用 `bus_register(&i2c_bus_type)` 函数完成I2C总线的注册。

`i2c_bus_type` 是一个 `bus_type` 结构体变量，它重载了 `bus_type` 结构体中的几个成员指针变量，其中最重要的是总线名称 `.name` 以及两个函数指针—— `match()` 和 `probe()` 。

```
/* /drivers/i2c/i2c-core-base.c */
struct bus_type i2c_bus_type = {
    .name          = "i2c",
    .match         = i2c_device_match,           // 负责总线上的device和driver匹配
    .probe         = i2c_device_probe,          // 在匹配成功后会执行以完成注册的收尾工作
    .remove        = i2c_device_remove,
    .shutdown      = i2c_device_shutdown,
};
EXPORT_SYMBOL_GPL(i2c_bus_type);           // 模块导出，可供其他模块使用
```

当任何一个`i2c_driver`或者`i2c_client`注册到I2C总线时，I2C总线都会调用 `i2c_device_match()` 函数对`i2c_driver`和`i2c_client`进行匹配。

```

/* /drivers/i2c/i2c-core-base.c */
static int i2c_device_match(struct device *dev, struct device_driver *drv)
{
    struct i2c_client      *client = i2c_verify_client(dev);
    struct i2c_driver       *driver;

    /* Attempt an OF style match */
    if (i2c_of_match_device(drv->of_match_table, client))
        return 1;

    /* Then ACPI style match */
    if (acpi_driver_match_device(dev, drv))
        return 1;

    driver = to_i2c_driver(drv);                                // 找到i2c_driver

    /* Finally an I2C match */
    if (i2c_match_id(driver->id_table, client))           // 用i2c_driver的id_table和device匹配。驱动名多个，但设备
        return 1;

    return 0;
}

```

i2c_device_match() 函数提供了三种匹配方式，它们执行顺序的先后代表了匹配优先级的高低。

- i2c_of_match_device 是设备树的匹配方式，具有最高的优先级。
- acpi_driver_match_device 是acpi(高级设置与电源管理)的匹配方式。
- i2c_match_id 则是通过注册i2c_driver结构体变量时提供的id_table进行匹配。

如今设备树匹配方式已经成为主流，后文将对其进行重点介绍。而acpi匹配方式使用较少且较复杂，所以下面只简单了解一下在设备树匹配方式产生之前，主要采用的匹配方式 i2c_match_id 。

```

const struct i2c_device_id *i2c_match_id(const struct i2c_device_id *id,
                                         const struct i2c_client *client)
{
    if (!(id && client))    // 要求设备和驱动的设备号表都非空
        return NULL;
    // 循环比对
    while (id->name[0]) {
        if (strcmp(client->name, id->name) == 0)
            return id;
        id++;
    }
    return NULL;
}

```

匹配成功后，I2C总线在后续会调用 i2c_device_probe() 函数完成相应注册的收尾工作。

除了被重载的几个成员指针变量，bus_type 结构体中还有一个关键成员变量 *p 。通过I2C驱动框架图可以知道，I2C总线上分别存储了i2c_driver链和i2c_client链来实现对驱动和设备的管理。 *p 是一个 subsys_private 结构体变量，所包含的成员 klist_devices 和 klist_drivers 分别对应了i2c_client链和i2c_driver链，在 bus_register() 中调用 klist_init() 函数进行初始化。

```

/* /include/linux/device/bus.h */
struct bus_type {
    const char *name;
    ...
    int (*match)(struct device *dev, struct device_driver *drv);
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);
    ...
    struct subsys_private *p;
    ...
};

/* /drivers/base/base.h */
struct subsys_private {
    ...
    struct klist klist_devices; // i2c_client链(对于I2C总线来说)
    struct klist klist_drivers; // i2c_driver链(对于I2C总线来说)
    ...
};

/* /include/linux/device/bus.c */
int bus_register(struct bus_type *bus)
{
    ...
    struct subsys_private *priv;
    ...
    klist_init(&priv->klist_devices, klist_devices_get, klist_devices_put); // 初始化i2c_client链
    klist_init(&priv->klist_drivers, NULL, NULL); // 初始化i2c_driver链
    ...
}

```

4.2. 注册i2c_adapter并将其添加到I2C总线

i2c_adapter 在硬件上对应SoC的I2C控制器，在内核中被认为是一个设备，而其对应驱动即是总线驱动。它向接在I2C控制器上的I2C设备提供在I2C总线上通信的基础方法，通过操作SoC的I2C控制器相关的寄存器实现数据收发。

4.2.1. 注册i2c_adapter到platform总线

i2c_adapter 一般通过两种方法注册：

- 一种做法是为I2C适配器创建一个platform设备注册到 platform_bus_type 总线上与I2C适配器的platform驱动匹配，在驱动的probe函数中向I2C总线添加 i2c_adapter 和提供 i2c_algorithm 通信方法。
- 另一种做法是I2C适配器作为pci设备注册到PCI总线上与I2C适配器的pci驱动匹配，在驱动的probe函数中向 i2c_bus_type 添加 i2c_adapter 和提供 i2c_algorithm 通信方法。

以TB-RK3568X为例，RK平台采用的是第一种方法。总线驱动的文件存储在路径 /drivers/i2c/busses 中，RK平台的总线驱动文件为 i2c-rk3x.c 。

结合4.1可知，编译进内核的驱动程序要在系统启动后为系统提供服务，则需事先将 init() 函数加入到内核的特定段中。驱动程序对应的 init() 函数是 module_init()，对应放置的位置是内核中的 initcall6.init 段(据此，也可知内核是先注册总线，后注册驱动)。

为了提高代码的重用性，消除多余的样板文件。当module_init和module_exit都不做任何特殊操作时，调用宏定义函数 module_platform_driver 替换 module_init 和 module_exit (实际还要调用一次宏定义函数 module_driver 才能完成替换)。

```
/* /include/linux/platform_device.h */
#define module_platform_driver(__platform_driver) \
    module_driver(__platform_driver, platform_driver_register, \
                  platform_driver_unregister)

/* /include/linux/device/driver.h */
/***
 * @_driver: driver name
 * @_register: register function for this driver type
 * @_unregister: unregister function for this driver type
 * @...: Additional arguments to be passed to __register and __unregister.
 */
#define module_driver(_driver, __register, __unregister, ...) \
static int __init __driver##_init(void) \
{ \
    return __register(&(_driver) , ##_VA_ARGS__); \
} \
module_init(__driver##_init); \
static void __exit __driver##_exit(void) \
{ \
    __unregister(&(_driver) , ##_VA_ARGS__); \
} \
module_exit(__driver##_exit);

/* /drivers/i2c/busses/i2c-rk3x.c */
module_platform_driver(rk3x_i2c_driver);           // 注册i2c_adapter的platform_driver
```

rk3x_i2c_driver 是一个 platform_driver 类型的结构体变量，重载了 *driver 结构体的部分变量(最重要的是匹配表 of_match_table)和两个函数指针 probe() 和 remove() 。

```
/* /drivers/i2c/busses/i2c-rk3x.c */
static struct platform_driver rk3x_i2c_driver = {
    .probe    = rk3x_i2c_probe,
    .remove   = rk3x_i2c_remove,
    .driver   = {
        .name    = "rk3x-i2c",
        .of_match_table = rk3x_i2c_match,
        .pm     = &rk3x_i2c_pm_ops,
    },
};
```

module_init() 调用 platform_driver_register(rk3x_i2c_driver) 向platform总线注册总线驱动。后续流程匹配和调用收尾工作与I2C总线类似，就不做赘述。

```

/* /drivers/i2c/busses/i2c-rk3x.c */
/**
 * @param pdev: 即i2c_adapter, 相当于挂载在platform总线上的platform_device。
 */
static int rk3x_i2c_probe(struct platform_device *pdev)
{
    struct device_node *np = pdev->dev.of_node; // i2c_adapter对应的设备节点
    const struct of_device_id *match; // 匹配表
    struct rk3x_i2c *i2c; // 声明一个rk3x_i2c的适配器结构体, 是i2c_adapter的进一步封装, 相当于面向对象中的继承
    ...
    /**
     * 采用devm_kzalloc与kzalloc相比, 优点在于不用考虑释放问题, 由内核完成内存回收工作
     * devm_kzalloc - Resource-managed kzalloc
     * @param pdev: 申请内存的目标设备
     * @param gftp: 申请内存的类型标志, 标识内存分配器将要采取的行为。其中GFP_KERNEL最常用, 五内存可用时可引起休眠。
     * @return: 成功返回首地址, 失败返回NULL
     * 为适配器结构体申请内存, 为后续实例化完成基础工作。
    */
    i2c = devm_kzalloc(&pdev->dev, sizeof(struct rk3x_i2c), GFP_KERNEL);
    if (!i2c) // 申请失败
        return -ENOMEM;
    ...
    // i2c_adapter部分成员初始化
    // 名字
    strlcpy(i2c->adap.name, "rk3x-i2c", sizeof(i2c->adap.name));
    // 拥有者
    i2c->adap.owner = THIS_MODULE;
    // 通信方法
    i2c->adap.algo = &rk3x_i2c_algorithm;
    i2c->adap.algo_data = i2c;
    i2c->adap.retries = 3;
    i2c->adap.dev.of_node = np;
    i2c->adap.dev.parent = &pdev->dev;
    i2c->dev = &pdev->dev;
    ...
    // 向I2C总线添加i2c_adapter, 重点
    ret = i2c_add_adapter(&i2c->adap);
    if (ret < 0)
        goto err_clk_notifier;
    return 0;
}

```

阅读注册收尾时被调用的 `rk3x_i2c_probe()` 函数源码可以知道，该函数负责将i2c_adapter和platform_device联系起来。并在对i2c_adapter各项参数进行配置后，将i2c_adapter添加到I2C总线。

参数配置中比较重要的是 `rk3x_i2c_algorithm`，因为每家芯片厂商SoC内部的I2C控制器是不一样的，所以 `i2c_algorithm` 中直接涉及硬件层面上的代码都是由芯片商提供。例如：对I2C控制器的寄存器操作。`i2c_algorithm` 提供的通信函数控制适配器产生特定的访问信号，虽然不同的I2C总线控制器被抽象成不同的 `i2c_adapter`，但是如果操作方式相同，则可以共享同一个 `i2c_algorithm`。

```

/* /drivers/i2c/busses/i2c-rk3x.c */
static const struct i2c_algorithm rk3x_i2c_algorithm = {
    .master_xfer          = rk3x_i2c_xfer,           // 通信方法，如果不支持I2C访问，则为NULL
    .master_xfer_atomic   = rk3x_i2c_xfer_polling,    // 通信方法，在atomic context环境下使用。比如在关机之前
    .functionality         = rk3x_i2c_func,           // 检测通信方法支持的功能或协议，设备驱动一般会调用这个
};

}

```

4.2.2. 将i2c_adapter添加到I2C总线

4.2.1末尾提到：在 rk3x_i2c_probe() 的最后，调用了I2C核心层为总线驱动开放的添加适配器至I2C总线的接口函数 i2c_add_adapter()。该函数的主要作用有两个：

- 将对应的I2C总线的id分配给i2c_adapter。因为一个Soc内部通常会有多个I2C控制器，而所有I2C控制器实际都公用同一份总线驱动代码。
- 解析由i2c_adapter控制的每一个从设备，并构建出i2c_client。设备驱动加载运行需要i2c_client才能继续。

```

/* /drivers/i2c/i2c-core-base.c */
int i2c_add_adapter(struct i2c_adapter *adapter)
{
    struct device *dev = &adapter->dev;
    int id;

    // 对于在设备树定义的i2c适配器，则通过设备树获得总线号(在rk3x_i2c_probe中赋值dev->of_node)
    if (dev->of_node) {
        // 获得总线号，因为总线驱动可以兼容多个同一平台的I2C控制器，一般会在dts里指定。
        id = of_alias_get_id(dev->of_node, "i2c");
        // 如果找到I2C总线号则直接注册
        if (id >= 0) {
            adapter->nr = id;
            return __i2c_add_numbered_adapter(adapter);      // 添加已明确总线号的I2C适配器
        }
    }
    mutex_lock(&core_lock);
    // 为i2c_adapter绑定动态总线号(从i2c_adapter_idr中申请一个可用的总线号)
    id = idr_alloc(&i2c_adapter_idr, adapter,
                   __i2c_first_dynamic_bus_num, 0, GFP_KERNEL);
    mutex_unlock(&core_lock);
    if (WARN(id < 0, "couldn't get idr"))
        return id;
    adapter->nr = id;
    return i2c_register_adapter(adapter); // 在I2C总线上注册i2c_adapter
}

```

在设备树中有一个叫做aliases的节点，在Linux内核启动的时候，会按如下流程解析这个节点，将节点内的信息加入到全局 aliases_lookup 链表中。

```

start_kernel
    --> setup_arch
        --> unflatten_device_tree
            --> of_alias_scan
                --> of_alias_add

```

`i2c_add_adapter()` 提供了两种为`i2c_adapter`分配id的方法。对于在设备树定义的`i2c_adapter`, 调用`of_alias_get_id()`遍历`aliases_lookup`链表获得明确的I2C总线号, 并调用`_i2c_add_numbered_adapter()`函数添加`i2c_adapter`。否则, 从`i2c_adapter_idr`中申请一个可用的总线号供`i2c_adapter`添加。

```
/* /drivers/of/base.c */
int of_alias_get_id(struct device_node *np, const char *stem)
{
    struct alias_prop *app;
    int id = -ENODEV;
    mutex_lock(&of_mutex);
    // 遍历链表aliases_lookup(成员为alias_prop), 逐一对比字符串stem。
    list_for_each_entry(app, &aliases_lookup, link) {
        // 过滤掉dtsi中aliases节点内的非I2C节点
        if (strcmp(app->stem, stem) != 0)
            continue;
        // 如果字符串匹配且找到对应的device_node, 则说明获得ID
        if (np == app->np) {
            id = app->id;
            break;
        }
    }
    mutex_unlock(&of_mutex);
    return id;
}
```

无论`i2c_adapter`是如何获得id的, 其最终都是调用`i2c_register_adapter`在I2C总线上注册`i2c_adapter`。

```

/* /drivers/i2c/i2c-core-base.c */
static int i2c_register_adapter(struct i2c_adapter *adap)
{
    ...
    /* 如果adapter没有name和algo算法，则无法注册 */
    if (WARN(!adap->name[0], "i2c adapter has no name"))
        goto out_list;
    if (!adap->algo) {
        pr_err("adapter '%s': no algo supplied!\n", adap->name);
        goto out_list;
    }
    ...
    // 注册到I2C总线
    dev_set_name(&adap->dev, "i2c-%d", adap->nr);
    adap->dev.bus = &i2c_bus_type;
    adap->dev.type = &i2c_adapter_type;

    res = device_register(&adap->dev);           // 将adapter设备添加到I2C总线，生成i2c_client。
    if (res) {
        pr_err("adapter '%s': can't register device (%d)\n", adap->name, res);
        goto out_list;
    }
    ...
    /* create pre-declared device nodes */
    // 构建从设备的软件抽象i2c_client，并与adapter建立联系
    of_i2c_register_devices(adap);
    ...
    if (adap->nr < __i2c_first_dynamic_bus_num)
        i2c_scan_static_board_info(adap);
    /* Notify drivers */
    mutex_lock(&core_lock);
    bus_for_each_drv(&i2c_bus_type, NULL, adap, __process_new_adapter);           // 匹配机制。遍历整个driver
    mutex_unlock(&core_lock);
    return 0;
    ...
out_list:
    ...
}

```

`i2c_register_adapter()` 函数最重要的作用是解析由`i2c_adapter`控制的每一个从设备，并构建出`i2c_client`。进而完成`i2c_driver`和`i2c_client`的匹配。前者由函数`of_i2c_register_devices()`负责，后者依靠函数`bus_for_each_drv()`调用`i2c_detect()`，寻找I2C总线支持的设备实现。

```

/* /drivers/i2c/i2c-core-of.c */
void of_i2c_register_devices(struct i2c_adapter *adap)
{
    struct device_node *bus, *node;
    // 构建i2c_client
    struct i2c_client *client;
    /* Only register child devices if the adapter has a node pointer set */
    if (!adap->dev.of_node)
        return;
    dev_dbg(&adap->dev, "of_i2c: walking child nodes\n");
    // 查找设备树节点中名称有直接描述I2C总线的节点，缩小查找范围
    bus = of_get_child_by_name(adap->dev.of_node, "i2c-bus");
    // 没找到则从头开始遍历
    if (!bus)
        bus = of_node_get(adap->dev.of_node);
    // 遍历每一个子节点，调用of_i2c_register_device解析设备树节点内容并注册为i2c_client
    for_each_available_child_of_node(bus, node) {
        if (of_node_test_and_set_flag(node, OF_POPULATED))
            continue;

        client = of_i2c_register_device(adap, node);
        if (IS_ERR(client)) {
            dev_err(&adap->dev,
                    "Failed to create I2C device for %pOF\n",
                    node);
            of_node_clear_flag(node, OF_POPULATED);
        }
    }
    of_node_put(bus);
}

/* /drivers/i2c/i2c-core-of.c */
static struct i2c_client *of_i2c_register_device(struct i2c_adapter *adap,
                                                struct device_
{
    struct i2c_client *client;
    struct i2c_board_info info;
    int ret;
    dev_dbg(&adap->dev, "of_i2c: register %pOF\n", node);
    ret = of_i2c_get_board_info(&adap->dev, node, &info);
    if (ret)
        return ERR_PTR(ret);
    client = i2c_new_client_device(adap, &info);
    if (IS_ERR(client))
        dev_err(&adap->dev, "of_i2c: Failure registering %pOF\n", node);
    return client;
}

```

在构建i2c_client的过程中，需要读取设备树信息赋值给对应的i2c_device。这个操作由 of_i2c_get_board_info() 函数完成。例如：调用 of_modalias_node() 函数获得 client->name。

```

/* /drivers/i2c/i2c-core-of.c */
int of_i2c_get_board_info(struct device *dev, struct device_node *node,
                         struct i2c_board_info *info)
{
    u32 addr;
    int ret;
    memset(info, 0, sizeof(*info));
    if (of_modalias_node(node, info->type, sizeof(info->type)) < 0) {
        dev_err(dev, "of_i2c: modalias failure on %pOF\n", node);
        return -EINVAL;
    }
    ret = of_property_read_u32(node, "reg", &addr); // 对应设备树中的<reg>标签
    if (ret) {
        dev_err(dev, "of_i2c: invalid reg on %pOF\n", node);
        return ret;
    }
    if (addr & I2C_TEN_BIT_ADDRESS) {
        addr &= ~I2C_TEN_BIT_ADDRESS;
        info->flags |= I2C_CLIENT_TEN;
    }
    if (addr & I2C_OWN_SLAVE_ADDRESS) {
        addr &= ~I2C_OWN_SLAVE_ADDRESS;
        info->flags |= I2C_CLIENT_SLAVE;
    }
    info->addr = addr;
    info->of_node = node;
    info->fwnode = of_fwnode_handle(node);
    if (of_property_read_bool(node, "host-notify"))
        info->flags |= I2C_CLIENT_HOST_NOTIFY;
    if (of_get_property(node, "wakeup-source", NULL))
        info->flags |= I2C_CLIENT_WAKE;
    return 0;
}

/* /drivers/of/base.c */
int of_modalias_node(struct device_node *node, char *modalias, int len)
{
    const char *compatible, *p;
    int cplen;

    compatible = of_get_property(node, "compatible", &cplen); // 查找"compatible"属性
    if (!compatible || strlen(compatible) > cplen)
        return -ENODEV;
    p = strchr(compatible, ','); // 定位, "manufacturer, model"
    strlcpy(modalias, p ? p + 1 : compatible, len);
    return 0;
}

```

最终由 `i2c_new_client_device()` 函数调用的 `i2c_new_client_device()` 根据获得的 `info` 完成 `i2c_client` 的构建。

```

/* /drivers/i2c/i2c-core-base.c */
struct i2c_client *i2c_new_client_device(struct i2c_adapter *adap,
                                         struct i2c_board_info const *i
{
    struct i2c_client      *client;
    int                     status;
    client = kzalloc(sizeof *client, GFP_KERNEL); // 分配内核空间
    if (!client)
        return ERR_PTR(-ENOMEM);
    // 配置client的基本信息
    client->adapter = adap;
    client->dev.platform_data = info->platform_data;
    client->flags = info->flags;
    client->addr = info->addr           // 对应设备树中的<reg>标签
    client->init_irq = info->irq;
    if (!client->init_irq)
        client->init_irq = i2c_dev_irq_from_resources(info->resources,
                                                       info-
strlcpy(client->name, info->type, sizeof(client->name));
    status = i2c_check_addr_validity(client->addr, client->flags);
    if (status) {
        dev_err(&adap->dev, "Invalid %d-bit I2C address 0x%02hx\n",
                client->flags & I2C_CLIENT_TEN ? 10 : 7, client->addr);
        goto out_err_silent;
    }
    /* Check for address business */
    status = i2c_check_addr_ex(adap, i2c_encode_flags_to_addr(client));
    if (status)
        dev_err(&adap->dev,
                "%d i2c clients have been registered at 0x%02x",
                status, client->addr);
    client->dev.parent = &client->adapter->dev;
    client->dev.bus = &i2c_bus_type;
    client->dev.type = &i2c_client_type;
    client->dev.of_node = of_node_get(info->of_node);
    client->dev.fwnode = info->fwnode;
    i2c_dev_set_name(adap, client, info, status);
    if (info->properties) {
        status = device_add_properties(&client->dev, info->properties);
        if (status) {
            dev_err(&adap->dev,
                    "Failed to add properties to client %s: %d\n",
                    client->name, status);
            goto out_err_put_of_node;
        }
    }
    status = device_register(&client->dev);
    if (status)
        goto out_free_props;
    dev_dbg(&adap->dev, "client [%s] registered with bus id %s\n",
            client->name, dev_name(&client->dev));
    return client;
out_free_props:
    ...
out_err_put_of_node:
    ...
out_err_silent:
    ...

```

}

4.2.3. 总结

整个过程调用关系如下：

```
rk3x_i2c_probe
    --> i2c_add_adapter
        --> __i2c_add_numbered_adapter
            --> i2c_register_adapter
                --> of_i2c_register_devices
                    --> of_i2c_register_device
                        --> of_i2c_get_board_info
                            --> of_modalias_node
                                --> i2c_new_client_device
                                --> bus_for_each_drv
                                    --> __process_new_adapter
                                        --> i2c_do_add_adapter
                                            --> i2c_detect
```

4.3. I2C设备驱动开发

完成I2C总线注册和i2c_adapter的注册后，就可以进行I2C设备驱动的开发了。由I2C驱动框架图可知，实现I2C设备驱动通常有两条路径：

4.3.1. 将I2C控制器暴露给应用的方式

该方式采用标准的 `file_operations` 字符设备的形式，将 `i2c_adapter` 设备化，在 `/dev` 目录下创建 `i2c-n(n=0, 1, 2...)` 设备节点。所实现的驱动可看作是一种" `i2c_driver` 成员函数 + 字符设备驱动"的虚拟驱动，需要由应用层通过 `read()`、`write()` 函数根据芯片手册直接对I2C控制器进行配置时序等操作，以实现对从设备的控制。这种方式是把对硬件的具体操作放在应用层去实现，适合用来快速测试一款I2C设备的功能，或者在 `i2c_driver` 工作不正常的时候排查具体是设备驱动工作问题还是主机驱动工作问题。并不能作为主流的开发方式。

驱动开发流程中的初始化、注册等准备流程工作和上文描述原理相同，不再赘述。

```

/* //drivers/i2c/i2c-dev.c */
//设备节点的操作方法
static const struct file_operations i2cdev_fops = {
    .owner          = THIS_MODULE,
    .llseek         = no_llseek,
    .read           = i2cdev_read,
    .write          = i2cdev_write,
    .unlocked_ioctl = i2cdev_ioctl,
    .open            = i2cdev_open,
    .release         = i2cdev_release,
};

#define I2C_MAJOR      89             /* Device major number */
static int __init i2c_dev_init(void)
{
    int res;

    printk(KERN_INFO "i2c /dev entries driver\n");
    // 申请设备号, I2C_MAJOR为89, 次设备号为0, I2C_MINORS为1<<20-1, 表示次设备号的数量。
    // 就是把这个主设备号对应的次设备号都申请了。
    res = register_chrdev_region(MKDEV(I2C_MAJOR, 0), I2C_MINORS, "i2c");
    if (res)
        goto out;
    // 创建一个同名类, 在 /sys/class中可以看到
    i2c_dev_class = class_create(THIS_MODULE, "i2c-dev");
    if (IS_ERR(i2c_dev_class)) {
        res = PTR_ERR(i2c_dev_class);
        goto out_unreg_chrdev;
    }
    i2c_dev_class->dev_groups = i2c_groups;

    /* Keep track of adapters which will be added or removed later */
    // 注册i2c总线的通知函数
    // 参数2详见下
    res = bus_register_notifier(&i2c_bus_type, &i2cdev_notifier);
    if (res)
        goto out_unreg_class;

    /* Bind to already existing adapters right away */
    // 遍历i2c总线上的所有设备, 每次都执行第二个参数对应的函数
    i2c_for_each_dev(NULL, i2cdev_attach_adapter);

    return 0;

out_unreg_class:
    class_destroy(i2c_dev_class);
out_unreg_chrdev:
    unregister_chrdev_region(MKDEV(I2C_MAJOR, 0), I2C_MINORS);
out:
    printk(KERN_ERR "%s: Driver Initialisation failed\n", __FILE__);
    return res;
}

```

设备节点是在驱动注册的过程中创建的，调用关系如下：

```
i2c_dev_init()
    i2c_for_each_dev()
        bus_for_each_dev()
            i2cdev_attach_adapter()
                device_create()
                device_create_file()
```

`i2cdev_attach_adapter()`函数调用`device_create()`函数和`device_create_file()`函数完成了设备节点的创建。

```
static int i2cdev_attach_adapter(struct i2c_adapter *adap)
{
    struct i2c_dev *i2c_dev;
    int res;

    //分配一个i2c_dev对象，并添加到i2c_dev_list链表中
    i2c_dev = get_free_i2c_dev(adap);
    if (IS_ERR(i2c_dev))
        return PTR_ERR(i2c_dev);

    /* 创建设备对象并在sysfs中注册，在/dev目录下创建设备号为MKDEV(I2C_MAJOR, adap->nr),
       名称为"i2c-%d"的字符设备节点*/
    i2c_dev->dev = device_create(i2c_dev_class, &adap->dev,
                                  MKDEV(I2C_MAJOR, adap->nr), NULL,
                                  "i2c-%d", adap->nr);
    if (IS_ERR(i2c_dev->dev)) {
        res = PTR_ERR(i2c_dev->dev);
        goto error;
    }

    //创建"/sys/class/i2c-dev/"i2c-%d"/name"文件
    res = device_create_file(i2c_dev->dev, &dev_attr_name);
    if (res)
        goto error_destroy;

    pr_debug("i2c-dev: adapter [%s] registered as minor %d\n",
            adap->name, adap->nr);
    return 0;
error_destroy:
    device_destroy(i2c_dev_class, MKDEV(I2C_MAJOR, adap->nr));
error:
    return_i2c_dev(i2c_dev);
    return res;
}
```

完成i2c设备驱动的注册后，就可以通过*i2cdev_fops*中提供的各项功能与I2C设备进行交互。

4.3.2. 将I2C控制器抽象成公共驱动的方式

该方式是把所有代码都放在驱动层实现，直接向应用层提供最终结果，即应用层甚至可以不知道I2C的存在。例如电容式触摸屏驱动直接向应用层提供`/dev/input/eventn`的操作接口，接收上报到应用层的输入事件。而不需要直到具体是怎么上报的，甚至应用层不知道触摸屏是使用I2C总线和主机进行数据交互的。

以汇顶科技的gt1x型电容式触摸屏为例，电容触摸屏通过I2C总线与SoC进行通信，利用其自带的触摸IC完成坐标计算后通过I2C将坐标信息传输给SoC，坐标的计算过程不需要SoC的参与。从这个角度上来说，电容触摸屏就是一个挂载

到SoC上的I2C从设备。触摸屏内I2C相关的功能函数 `gt1x_i2c_write()` 和 `gt1x_i2c_read()` 都被封装到 `gt1x_ts_work_func()` 中，具体可阅读位于 `/driver/input/touchscreen/gt1x/` 的源码 `gt1x.c`。

4.4. 重点分析

4.4.1. 何时调用match()函数？何时调用probe()函数？

要搞清楚 `match()` 和 `probe()` 函数是何时被何者所调用的，需对源码进行深入了解。根据文档描述，`probe()` 函数执行于 `match()` 函数之后，且匹配触发的前提是要有 `i2c_driver` 或者 `i2c_client` 注册到 I2C 总线。所以尝试从 `i2c_driver` 的注册代码中寻找答案。

```
/* /drivers/i2c/i2c-core-base.c */
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
{
    ...
    driver->driver.bus = &i2c_bus_type;
    INIT_LIST_HEAD(&driver->clients);
    ...
    /* When registration returns, the driver core
     * will have called probe() for all matching-but-unbound devices.
     */
    res = driver_register(&driver->driver);      // 实际注册位置
    if (res)
        return res;
    ...
}
```

注意到 `i2c_register_driver` 函数中实际是调用了 `driver_register()` 函数完成注册，并且根据注释描述，正是在 `driver_register()` 中完成了 `match()` 和 `probe()` 的工作。因此继续进入 `driver_register()` 函数中进行跟踪。

```
/* /drivers/base/driver.c */
int driver_register(struct device_driver *drv)
{
    ...
    ret = bus_add_driver(drv);                  // 在总线上添加传递的驱动(将驱动添加到总线的驱动链表中)
    ...
}
```

`driver_register()` 函数中调用了 `bus_add_driver()` 函数，将驱动添加到总线上，继续深入直到找到有关 `match` 和 `probe` 的函数。

```

/* /drivers/base/bus.c */
int bus_add_driver(struct device_driver *drv)
{
    ...
    if (drv->bus->p->drivers_autoprobe)
    {
        error = driver_attach(drv);           // 跟踪driver_attach();
        if (error)
            goto out_unregister;
    }
    ...
}

/* /drivers/base/dd.c */
int driver_attach(struct device_driver *drv)
{
    return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach); // 实际是在调用__driver_attach();
}

/* /drivers/base/bus.c */
int bus_for_each_dev(struct bus_type *bus, struct device *start,
                     void *data, int (*fn)(struct device *, void *))
{
    ...
    klist_iter_init_node(&bus->p->klist_devices, &i,
                         (start ? &start->p->knode_bus : NULL));      // 链表头开始遍历连接在总线上的设备链
    while (!error && (dev = next_device(&i)))
        error = fn(dev, data);
    klist_iter_exit(&i);
    return error;
}

/* /drivers/base/dd.c */
static int __driver_attach(struct device *dev, void *data)
{
    ...
    ret = driver_match_device(drv, dev);           // 驱动和设备匹配
    if (ret == 0) {
        /* no match */
        return 0;
    } else if (ret == -EPROBE_DEFER) {
        dev_dbg(dev, "Device match requests probe deferral\n");
        driver_deferred_probe_add(dev);
    } else if (ret < 0) {
        dev_dbg(dev, "Bus failed to match device: %d\n", ret);
        return ret;
    } /* ret > 0 means positive match */

    .....

    device_driver_attach(drv, dev);
    return 0;
}

```

```

/* /drivers/base/base.h */
static inline int driver_match_device(struct device_driver *drv,
                                      struct device *dev)
{
    return drv->bus->match ? drv->bus->match(dev, drv) : 1;
}

```

至此，可知是 `driver_match_device` 函数调用了I2C总线的 `i2c_device_match()` 函数完成对驱动和设备的匹配工作。而匹配完成后 `probe()` 的调用则还需跟踪进 `device_driver_attach` 函数分析。

```

/* /drivers/base/dd.c */
int device_driver_attach(struct device_driver *drv, struct device *dev)
{
    int ret = 0;
    __device_driver_lock(dev, dev->parent);
    /*
     * If device has been removed or someone has already successfully
     * bound a driver before us just skip the driver probe call.
     */
    if (!dev->p->dead && !dev->driver)
        ret = driver_probe_device(drv, dev);
    __device_driver_unlock(dev, dev->parent);
    return ret;
}

```

```

/* /drivers/base/dd.c */
int driver_probe_device(struct device_driver *drv, struct device *dev)
{
    ...
    pm_runtime_barrier(dev);
    if (initcall_debug)
        ret = really_probe_debug(dev, drv);
    else
        ret = really_probe(dev, drv);
    ...
}

```

在 `really_probe()` 函数中，调用了总线的 `probe()` 函数，以I2C为例就是 `i2c_device_probe()`。

```

/* /drivers/base/dd.c */
static int really_probe(struct device *dev, struct device_driver *drv)
{
    ...
    if (dev->bus->probe) {
        ret = dev->bus->probe(dev);
        if (ret)
            goto probe_failed;
    } else if (drv->probe) {
        ret = drv->probe(dev);
        if (ret)
            goto probe_failed;
    }
    ...
}

```

`i2c_device_probe()` 函数中，通过 `driver->probe()` 调用定义的 `i2c_driver` 的 `probe()` 函数。以 `gt1x` 型电容式触摸屏的驱动为例，则是 `gt1x_ts_probe()`。

```
/* /drivers/i2c/i2c-core-base.c */
static int i2c_device_probe(struct device *dev)
{
    ...
    /*
     * When there are no more users of probe(),
     * rename probe_new to probe.
     */
    if (driver->probe_new)
        status = driver->probe_new(client);
    else if (driver->probe)
        status = driver->probe(client,
                               i2c_match_id(driver->id_table, client));
    else
        status = -EINVAL;
    ...
}
```

整体流程如下：

```

int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
{
    res = driver_register(&driver);
    if (res)
        return res;
    i2c_for_each_dev(driver, __process_new_driver);
}

int driver_register(struct device_driver *drv)
{
    int ret;
    struct device_driver *other;
    ret = bus_add_driver(drv);
    return ret;
}

int bus_add_driver(struct device_driver *drv)
{
    error = driver_attach(drv);
}

int driver_attach(struct device_driver *drv)
{
    return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
}

int bus_for_each_dev(struct bus_type *bus, struct device *start,
                     void *data, int (*fn)(struct device *, void *))
{
    struct klist_iter i;
    struct device *dev;
    int error = 0;
    klist_iter_init_node(&bus->p->klist_devices, &i,
                         (start ? &start->p->knode_bus : NULL));
    while ((dev = next_device(&i)) && !error)
        error = fn(dev, data);
    klist_iter_exit(&i);
    return error;
}

static int __driver_attach(struct device *dev, void *data)
{
    if (!driver_match_device(drv, dev))
        return 0;

    if (!dev->driver)
        driver_probe_device(drv, dev);
}

static inline int driver_match_device(struct device_driver *drv,
                                     struct device *dev)
{
    return drv->bus->match ? drv->bus->match(dev, drv) : 1;
}

int driver_probe_device(struct device_driver *drv, struct device *dev)
{
    int ret = 0;
    pm_runtime_barrier(dev);
    ret = really_probe(dev, drv);
    pm_request_idle(dev);
    return ret;
}

static int really_probe(struct device *dev, struct device_driver *drv)
{
    if (dev->bus->probe) {
        ret = dev->bus->probe(dev);
        if (ret)
            goto ↓probe_failed;
    } else if (drv->probe) {
        ret = drv->probe(dev);
        if (ret)
            goto ↓probe_failed;
    }
}

```

调用 i2c_bus_type->match,
即 i2c_device_match()

如果匹配不成功，则
返回错误，不会调用
probe 函数

调用 i2c_bus_type->probe,
即 i2c_device_probe()

4.4.2. 设备树匹配机制

4.1节中便讨论过 `i2c_device_match()` 提供了三种匹配方式，并简单介绍了使用注册时配置的 `id_table` 进行匹配的方法。下面主要讨论设备树匹配方式。

```
/* /drivers/i2c/i2c-core-of.c */
const struct of_device_id*
i2c_of_match_device(const struct of_device_id *matches,
                    struct i2c_client *client)
{
    const struct of_device_id *match;

    if (!(client && matches))
        return NULL;

    match = of_match_device(matches, &client->dev);           // 进行匹配
    if (match)                                                 // 一次
        return match;

    return i2c_of_match_device_sysfs(matches, client);          // 与I2C MUX(I2C多路复用器有关), 内核将MUX通道抽象
}

/* /drivers/of/device.c */
const struct of_device_id *of_match_device(const struct of_device_id *matches,
                                           const struct device *dev)
{
    if ((!matches) || (!dev->of_node))
        return NULL;
    return of_match_node(matches, dev->of_node);
}

/* /drivers/of/base.c */
const struct of_device_id *of_match_node(const struct of_device_id *matches,
                                         const struct device_node *node)
{
    ...
    match = __of_match_node(matches, node);
    ...
    return match;
}
```

`__of_match_node()` 函数是设备树匹配的核心部分——通过得分机制，寻找最佳匹配。

```

/* /drivers/of/base.c */
static const struct of_device_id *_of_match_node(const struct of_device_id *matches,
                                                const struct device_node *node)
{
    const struct of_device_id *best_match = NULL;
    int score, best_score = 0;
    if (!matches)
        return NULL;
    // 根据评分来选取最佳匹配
    for (; matches->name[0] || matches->type[0] || matches->compatible[0]; matches++) {
        score = __of_device_is_compatible(node, matches->compatible,
                                           matches->type, matches->name);
        if (score > best_score) {
            best_match = matches;
            best_score = score;
        }
    }
    return best_match;
}

/* /drivers/of/base.c */
static int __of_device_is_compatible(const struct device_node *device,
                                      const char *compat, const char *type, const char *name)
{
    struct property *prop;
    const char *cp;
    int index = 0, score = 0;
    /* Compatible match has highest priority */
    if (compat && compat[0]) {
        prop = __of_find_property(device, "compatible", NULL);
        for (cp = of_prop_next_string(prop, NULL); cp;
             cp = of_prop_next_string(prop, cp), index++) {
            if (of_compat_cmp(cp, compat, strlen(compat)) == 0) {
                score = INT_MAX/2 - (index << 2);           // 评分
                break;
            }
        }
        if (!score)
            return 0;
    }
    ...
    return score;
}

```

```
/* /include/linux/of.h */
#define of_compat_cmp(s1, s2, l) strcasecmp((s1), (s2))
/* /lib/string.c */
int strcasecmp(const char *s1, const char *s2) // 忽略大小写比较字符串
{
    int c1, c2;
    do {
        c1 = tolower(*s1++);
        c2 = tolower(*s2++);
    } while (c1 == c2 && c1 != 0);
    return c1 - c2;
}
```

整体匹配流程如下：

```

static int i2c_device_match(struct device *dev, struct device_driver *drv)
{
    struct i2c_client *client = i2c_verify_client(dev);
    struct i2c_driver *driver;

    if (!client)
        return 0;

    /* Attempt an OF style match */
    if (of_driver_match_device(dev, drv)) 设备树名字比较
        return 1;

    /* Then ACPI style match */
    if (acpi_driver_match_device(dev, drv))
        return 1;

    driver = to_i2c_driver(drv);
    /* match on an id table if there is one */
    if (driver->id_table)
        return i2c_match_id(driver->id_table, client) != NULL;
没有设备树情况下比较id_table

    return 0;
} end i2c_device_match ? |

```

static const struct i2c_device_id ov5640_id[] = {
 {"ov5640", 0},
 {},
};

```

static struct i2c_driver ov5640_i2c_driver = {
    .driver = {
        .owner = THIS_MODULE,
        .name = "ov5640",
        .of_match_table = of_match_ptr(of_ov5640_id),
    },
    .probe = ov5640_probe,
    .remove = ov5640_remove,
    .id_table = ov5640_id,
};

```



```

static inline int of_driver_match_device(struct device *dev,
                                         const struct device_driver *drv)
{
    return of_match_device(drv->of_match_table, dev) != NULL;
}

const struct of_device_id *of_match_device(const struct of_device_id *matches,
                                           const struct device *dev)
{
    if ((!matches) || (!dev->of_node))
        return NULL;
    return of_match_node(matches, dev->of_node);
}

const struct of_device_id *__of_match_node(const struct of_device_id *matches,
                                          const struct device_node *node)
{
    const struct of_device_id *best_match = NULL;
    int score, best_score = 0;

    if (!matches)
        return NULL;

    for (; matches->name[0] || matches->type[0] || matches->compatible[0]; matches++) {
        score = __of_device_is_compatible(node, matches->compatible,
                                           matches->type, matches->name);
        if (score > best_score) {
            best_match = matches;
            best_score = score;
        }
    }
    return best_match;
} ? end __of_match_node ? |

```

```

static int __of_device_is_compatible(const struct device_node *device,
                                     const char *compat, const char *type, const char *name)
{
    struct property *prop;
    const char *cp;
    int index = 0, score = 0;
从设备树中查找 compatible属性

    /* Compatible match has highest priority */
    if (compat && compat[0]) {
        prop = __of_find_property(device, "compatible", NULL);
        for (cp = __of_prop_next_string(prop, NULL); cp;
             cp = __of_prop_next_string(prop, cp), index++) {
            if (of_compat_cmp(cp, compat, strlen(compat)) == 0) {
                score = INT_MAX/2 - (index << 2);
                break;
            }
        }
        if (!score)
            return 0;
    }
}

```

ov5640: ov5640@03c {
 compatible = 'ovti,ov5640';
 reg = <0x3c>;
 pinctrl-names = "default";
 pinctrl-0 = <&pinctrl_csil
};

```
#define of_compat_cmp(s1, s2, l) strcasecmp((s1), (s2))

int strcasecmp(const char *s1, const char *s2)
{
    int c1, c2;

    do {
        c1 = tolower(*s1++);
        c2 = tolower(*s2++);
    } while (c1 == c2 && c1 != 0);
    return c1 - c2;
}
```

注：除了设备树匹配所调用的函数从 `of_driver_match_device` 变成 `i2c_of_match_device`，其余地方相差不大。其他总线(如platform, SPI)都是依照上图流程进行匹配，所以就不进行修改了。

5. 结语

本文对Linux中I2C驱动框架进行了简单的介绍。通过分析整体架构和代码结构，了解各个组成部分的功能和相互联系。重点对I2C驱动框架的核心层、总线驱动、设备驱动的代码实现流程进行梳理。并深入分析了 `probe()` 和 `match()` 函数的调用时机以及设备树匹配机制。

6. 参考资料

- Linux内核源码： .../OpenHarmony/out/kernel/src_tmp/linux-5.10/
- 《Linux设备驱动开发详解》
 - 第15章 《Linux I2C核心、总线与设备驱动》
 - 第18章 《ARM Linux 设备树》
- i2c驱动移植流程
- Linux驱动分析——I2C子系统
- linux驱动之I2C子系统
- 基于RK3399的Linux驱动开发 -- I2C驱动框架
- Linux I2C驱动整理（以RK3399Pro+Kernel 4.4为例）
- I2C适配器驱动及设备驱动代码详解
- Linux驱动之I2C总线驱动开发
- linux内核I2C子系统详解——看这一篇就够了
- I2C协议和驱动框架分析（二）
- 【linux iic子系统】i2c整体框图【精髓部分】（五）
- Linux系统驱动之I2C_Adapter驱动框架讲解与编写
- Input子系统-Touch Screen
- i2c的设备树和驱动是如何匹配以及何时调用probe的
- I2C子系统之适配器的设备接口分析(i2c-dev.c文件分析)