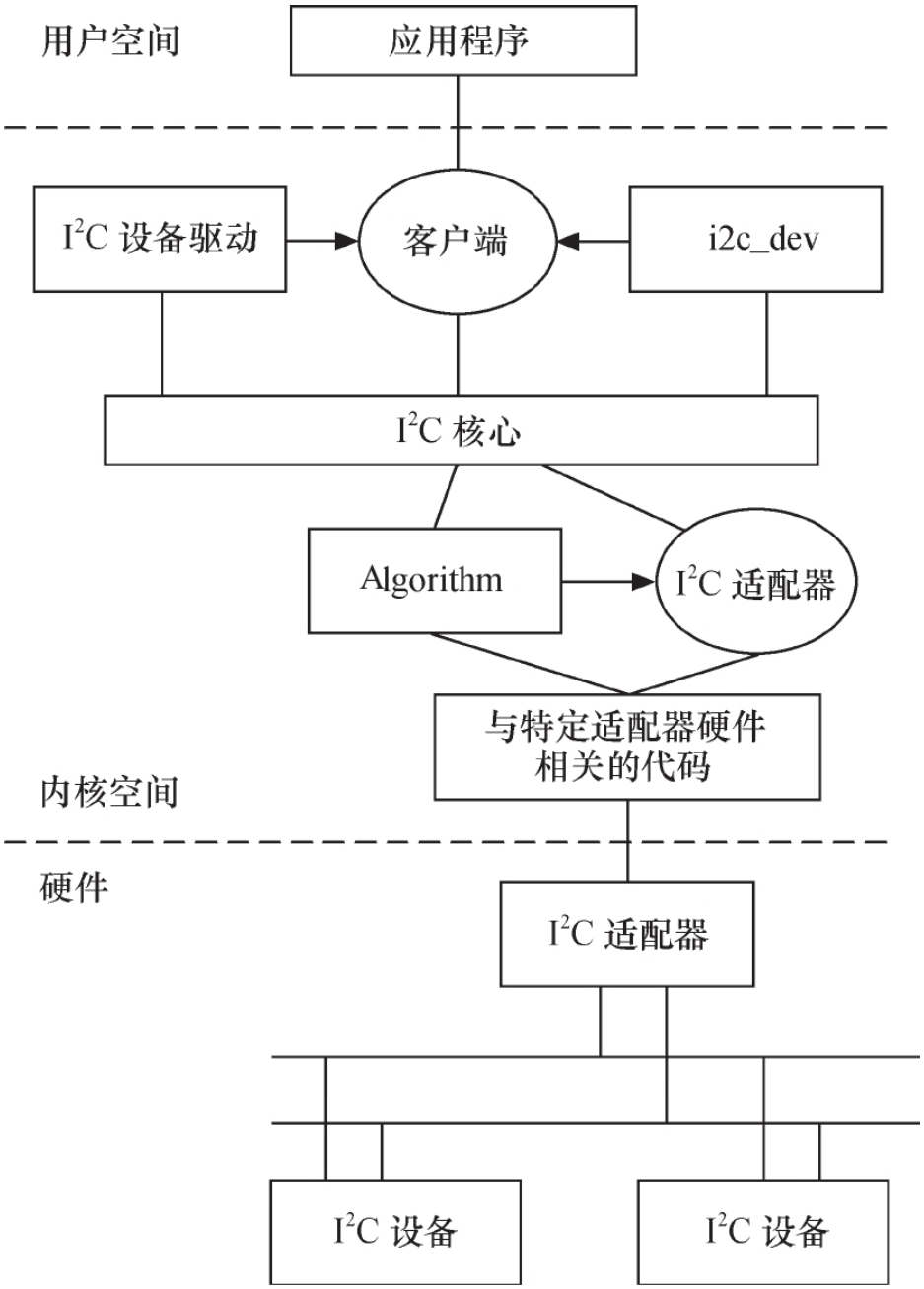
I2C体系结构



i2c体系结构分为3个组成部分：I2C核心，i2c总线驱动，i2c设备驱动

i2c核心：

i2c核心提供了i2c总线驱动和设备驱动的注册、注销方法，i2c通信方法（即Algorithm）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：

i2c总线驱动是**对I2c硬件体系中适配器端的实现**，适配器可由CPU控制，甚至可以直接集成在CPU内部

i2c总线驱动主要包括I2c**适配器数据结构i2c_adapter**、i2c适配器的**Algorithm数据结构i2c_algorithm**和控制I2C适配器产生**通信信号的函数**

经过i2c总线驱动的代码，我们可以控制i2c适配器以主控方式产生开始位、停止位、读写周期，以及以从设备方式被读写、产生ACK等

i2c设备驱动

i2c设备驱动（也称为客户驱动）是对I2C硬件体系结构中设备端的实现，设备有一半挂在受CPU控制的I2C适配器上，通过i2c适配器与CPU交换数据

i2c设备驱动主要包括数据结构I2C_driver和i2c_client，我们需要根据具体设备实现其中的成员函数

linux中i2c源码体系（linux内核源码中的driver目录下的i2c目录）

1. i2c-core.c：这个文件实现了i2c核心的功能

2. i2c-dev.c：实现了I2c适配器设备文件的功能（**总线驱动**），每一个适配器都被分配一个设备。通过适配器访问设备时的主设备号都为89，次设备号为0-255.应用程序通过“i2c-%d”文件名并使用文件操作接口来访问设备。

3. busses文件夹：这个文件包含了一些I2C主机控制器的驱动。

4. algos文件夹：实现了一些i2c总线适配器的通信方法

5. 此外在内核中 include/linux/i2c.h中，对i2c_adapter(适配器 属于总线驱动)、i2c_algorithm（通信方法，属于总线驱动）、i2c_driver和i2c_client这四个数据结构进行了定义

- i2c_adapter

```
struct i2c_adapter {
    struct module *owner;
    unsigned int class;               /* classes to allow probing for */
    const struct i2c_algorithm *algo; /* the algorithm to access the bus */
    void *algo_data;

    /* data fields that are valid for all devices   */
    const struct i2c_lock_operations *lock_ops;
    struct rt_mutex bus_lock;
    struct rt_mutex mux_lock;

    int timeout;                    /* in jiffies */
    int retries;
    struct device dev;              /* the adapter device */

    int nr;
    char name[48];
    struct completion dev_released;

    struct mutex userspace_clients_lock;
    struct list_head userspace_clients;

    struct i2c_bus_recovery_info *bus_recovery_info;
    const struct i2c_adapter_quirks *quirks;
```

```
25
26      struct irq_domain *host_notify_domain;
27  };
```

- i2c_algorithm

```
1   struct i2c_algorithm {
2      /* If an adapter algorithm can't do I2C-level access, set master_xfer
3         to NULL. If an adapter algorithm can do SMBus access, set
4         smbus_xfer. If set to NULL, the SMBus protocol is simulated
5         using common I2C messages */
6      /* master_xfer should return the number of messages successfully
7         processed, or a negative value on error */
8      int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
9                         int num);
10     int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,
11                        unsigned short flags, char read_write,
12                        u8 command, int size, union i2c_smbus_data *data);
13
14     /* To determine what the adapter supports */
15     u32 (*functionality) (struct i2c_adapter *);
16
17  #if IS_ENABLED(CONFIG_I2C_SLAVE)
18     int (*reg_slave)(struct i2c_client *client);
19     int (*unreg_slave)(struct i2c_client *client);
20  #endif
21  };
```

- i2c_driver

```
1   struct i2c_driver {
2      unsigned int class;
3
4      /* Notifies the driver that a new bus has appeared. You should avoid
5       * using this, it will be removed in a near future.
6       */
7      int (*attach_adapter)(struct i2c_adapter *) __deprecated;
8
9      /* Standard driver model interfaces */
10     int (*probe)(struct i2c_client *, const struct i2c_device_id *);
```

```
    int (*remove)(struct i2c_client *);

    /* New driver model interface to aid the seamless removal of the
     * current probe()'s, more commonly unused than used second parameter.
     */
    int (*probe_new)(struct i2c_client *);

    /* driver model interfaces that don't relate to enumeration  */
    void (*shutdown)(struct i2c_client *);

    /* Alert callback, for example for the SMBus alert protocol.
     * The format and meaning of the data value depends on the protocol.
     * For the SMBus alert protocol, there is a single bit of data passed
     * as the alert response's low bit ("event flag").
     * For the SMBus Host Notify protocol, the data corresponds to the
     * 16-bit payload data reported by the slave device acting as master.
     */
    void (*alert)(struct i2c_client *, enum i2c_alert_protocol protocol,
                  unsigned int data);

    /* a ioctl like command that can be used to perform specific functions
     * with the device.
     */
    int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);

    struct device_driver driver;
    const struct i2c_device_id *id_table;

    /* Device detection callback for automatic device creation */
    int (*detect)(struct i2c_client *, struct i2c_board_info *);
    const unsigned short *address_list;
    struct list_head clients;

    bool disable_i2c_core_irq_mapping;
};
```

- i2c_client

```
struct i2c_client {
```

```
    unsigned short flags;           /* div., see below              */
    unsigned short addr;            /* chip address - NOTE: 7bit    */
                                    /* addresses are stored in the  */
                                    /* _LOWER_ 7 bits               */
    char name[I2C_NAME_SIZE];
    struct i2c_adapter *adapter;    /* the adapter we sit on        */
    struct device dev;              /* the device structure         */
    int irq;                        /* irq issued by device         */
    struct list_head detected;
#if IS_ENABLED(CONFIG_I2C_SLAVE)
    i2c_slave_cb_t slave_cb;        /* callback for slave mode      */
#endif
};
```

下面分析i2c_adapter、i2c_algorithm、i2c_driver和i2c_client这四个数据结构之间的关系

1. i2c_adapter和i2calgorithm

i2c_adapter对应于物理上的一个适配器，i2c_algorithm对应一套通信方法。一个i2c适配器需要I2c_algorithm提供的通信函数来控制适配器产生特定的访问周期。缺少i2c_algorithm的i2c_adapter什么都做不了，因此i2c_adapter中包含i2c_algorithm的指针

i2c_algorithm中的关键函数master_xfer（）用于产生i2c访问周期的信号。以i2c_msg为单位

```
int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
                   int num);
```

i2c_msgs:其中包括了i2c地址，方向、缓冲区、缓冲长度等

```
struct i2c_msg {
    __u16 addr;     /* slave address                        */
    __u16 flags;
#define I2C_M_TEN          0x0010  /* this is a ten bit chip address */
#define I2C_M_RD           0x0001  /* read data, from slave to master */
#define I2C_M_STOP         0x8000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_NOSTART              0x4000  /* if I2C_FUNC_NOSTART */
#define I2C_M_REV_DIR_ADDR 0x2000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_IGNORE_NAK   0x1000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_NO_RD_ACK            0x0800  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_RECV_LEN             0x0400  /* length will be first received byte */
    __u16 len;              /* msg length                           */
```

```
    __u8 *buf;              /* pointer to msg data                  */
};
```

2. **i2c_driver和i2c_client**

i2c_driver对于一套驱动方法，其主要成员是probe()、remove()、suspend()、resume()等，另外，struct i2c_device_id的形式id_tabale是该驱动支持的I2c设备的列表。**i2c_client对应于真实的物理设备**，每个i2c设备都需要一个i2c_client来描述。i2c_driver跟i2c_client的关系是一对多。一个i2c_driver可以支持多个同类型的i2c_client

i2c_client的信息通常在BSP的板文件中通过i2c_board_info填充。在I2C总线驱动i2c_bus_type的match（）函数i2c_device_match（）中，会调用i2c_match_id（）函数匹配在板文件中定义的ID和i2c_driver所支持的ID表

但是在实际的驱动中描述硬件信息的一般是在dts中，然后通过i2c_driver的of_match_table字段来进行匹配

eg

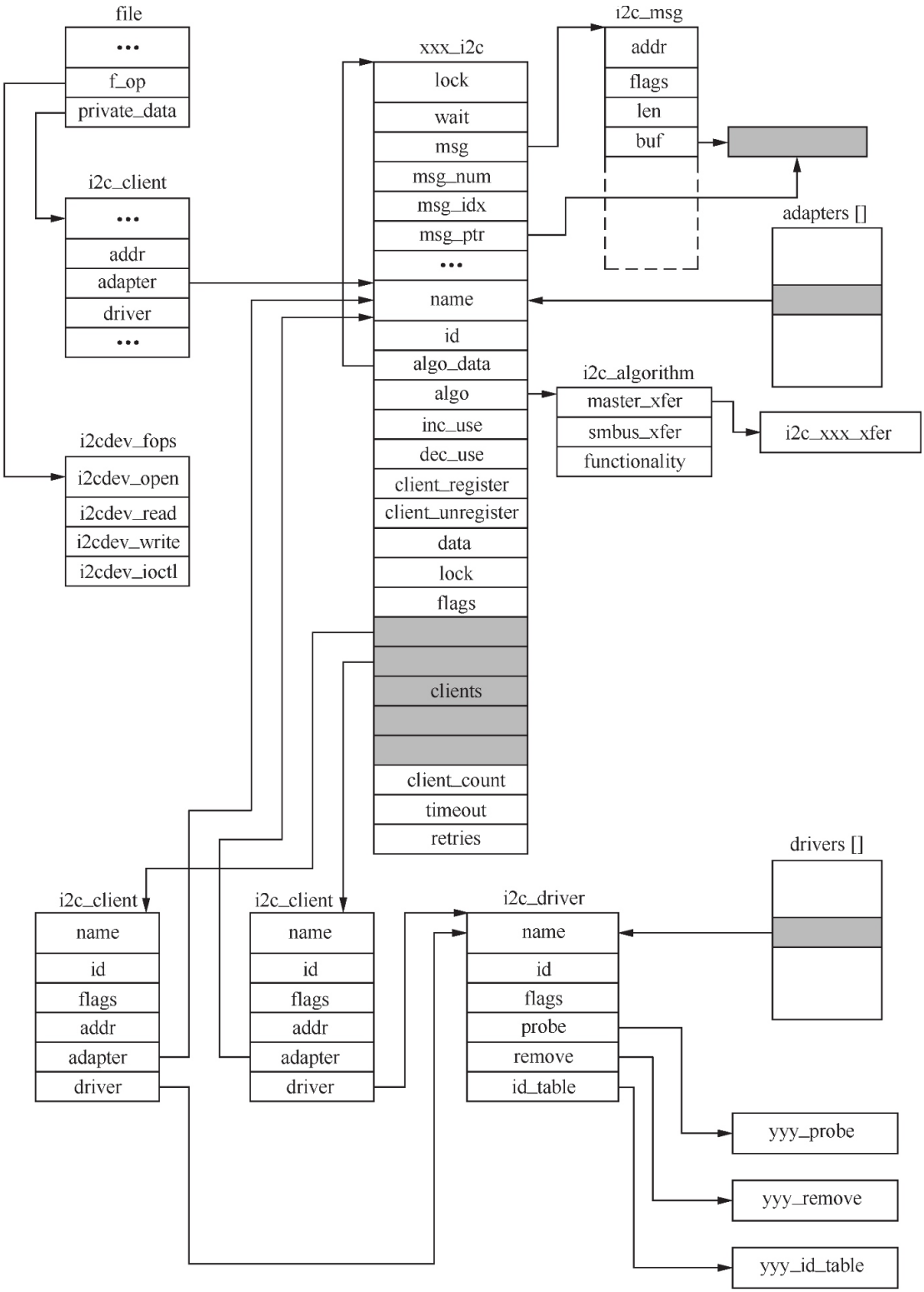
```
tatic const struct of_device_id adv7180_of_id[] = {
    { .compatible = "adi,adv7180", },
    { .compatible = "adi,adv7180cp", },
    { .compatible = "adi,adv7180st", },
    { .compatible = "adi,adv7182", },
    { .compatible = "adi,adv7280", },
    { .compatible = "adi,adv7280-m", },
    { .compatible = "adi,adv7281", },
    { .compatible = "adi,adv7281-m", },
    { .compatible = "adi,adv7281-ma", },
    { .compatible = "adi,adv7282", },
    { .compatible = "adi,adv7282-m", },
    { },
};

MODULE_DEVICE_TABLE(of, adv7180_of_id);
#endif

static struct i2c_driver adv7180_driver = {
    .driver = {
                .name = KBUILD_MODNAME,
                .pm = ADV7180_PM_OPS,
                .of_match_table = of_match_ptr(adv7180_of_id),
                },
```

```
    .probe = adv7180_probe,
    .remove = adv7180_remove,
    .id_table = adv7180_id,
};
```

3. i2c_adapter与i2c_client

i2c_adapter与i2c_client的关系与I2C硬件体系中适**配器和设备的关系一致**，即i2c_client依附于i2c_adapter。由于一个适配器可以连接多个I2C设备，所以一个i2c_adapter也可以被多个i2c_client依附，i2c_adapter中包括依附于它的i2c_client的链表

假设I2C总线适配器xxx上有两个使用相同驱动程序的yyy I2C设备，在打开该I2C总线的设备节点后，相关数据结构之间的逻辑组织关系将如图15.2所示



对于i2c子系统，工程师要实现的主要工作如下：

提供I2C适配器的硬件驱动，探测、初始化I2C适配器（如申请I2C的I/O地址和中断号）、驱动CPU控制的I2C适配器从硬件上产生各种信号以及处理I2C中断等

- 提供I2C适配器的Algorithm，用具体适配器的xxx_xfer（）函数填充i2c_algorithm的master_xfer指针，并把i2c_algorithm指针赋值给i2c_adapter的algo指针。
- 实现I2C设备驱动中的i2c_driver接口，用具体设备yyy的yyy_probe（）、yyy_remove（）、yyy_suspend（）、yyy_resume（）函数指针和i2c_device_id设备ID表赋值给i2c_driver的probe、remove、suspend、resume和id_table指针。

• 实现I2C设备所对应类型的具体驱动，i2c_driver只是实现设备与总线的挂接，而挂接在总线上的设备则千差万别。例如，如果是字符设备，就实现文件操作接口，即实现具体设备yyy的yyy_read（）、yyy_write（）和yyy_ioctl（）函数等；如果是声卡，就实现ALSA驱动

上述工作中前两个属于I2C总线驱动，后两个属于I2C设备驱动。

Linux i2c核心

I2C核心（drivers/i2c/i2c-core.c）中提供了一组不依赖于硬件平台的接口函数，这个文件一般不需要被工程师修改，但是理解其中的主要函数非常关键，**因为I2C总线驱动和设备驱动之间以I2C核心作为纽带。**I2C核心中的主要函数如下

1. 增加/删除i2c_adapter（适配器）

int i2c_add_adapter(struct i2c_adapter *adap);

2. 增加/删除i2c_driver

int i2c_register_driver(struct module *owner, struct i2c_driver *driver);

void i2c_del_driver(struct i2c_driver *driver);

#define i2c_add_driver(driver) \

i2c_register_driver(THIS_MODULE, driver)

3. I2C传输、发送和接收

```
int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg *msgs, int num);
int i2c_master_send(struct i2c_client *client,const char *buf ,int count);
int i2c_master_recv(struct i2c_client *client, char *buf ,int count);
```

i2c_transfer（）函数用于进行I2C适配器和I2C设备之间的一组消息交互，其中第2个参数是一个指向i2c_msg数组的指针，所以i2c_transfer（）一次可以传输多个i2c_msg（考虑到很多外设的读写波形比较复杂，比如读寄存器可能要先写，所以需要两个以上的消息）。而对于时序比较简单的外设，i2c_master_send（）函数和i2c_master_recv（）函数内部会调用i2c_transfer（）函数分别完成一条写消息和一条读消息

注意：i2c_transfer（）函数本身不具备驱动适配器物理硬件以完成消息交互的能力，它只是寻找到与i2c_adapter对应的i2c_algorithm，并使用i2c_algorithm的master_xfer（）函数真正驱动硬件流程

```
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
{
    int ret;

    if (adap->algo->master_xfer) {
#ifdef DEBUG
            for (ret = 0; ret < num; ret++) {
                    dev_dbg(&adap->dev,
```

```
                        "master_xfer[%d] %c, addr=0x%02x, len=%d%s\n",
                        ret, (msgs[ret].flags & I2C_M_RD) ? 'R' : 'W',
                        msgs[ret].addr, msgs[ret].len,
                        (msgs[ret].flags & I2C_M_RECV_LEN) ? "+" : "");
        }
#endif
    if (in_atomic() || irqs_disabled()) {
                ret = i2c_trylock_bus(adap, I2C_LOCK_SEGMENT);
                if (!ret)
                        /* I2C activity is ongoing. */
                        return -EAGAIN;
        } else {
                i2c_lock_bus(adap, I2C_LOCK_SEGMENT);
        }

        ret = __i2c_transfer(adap, msgs, num);
        i2c_unlock_bus(adap, I2C_LOCK_SEGMENT);

        return ret;
    } else {
        dev_dbg(&adap->dev, "I2C level transfers not supported\n");
        return -EOPNOTSUPP;
    }
}
```

linux I2C适配器驱动（i2c总线驱动）

由于I2C总线控制器通常是在内存上的，所以它本身也连接在platform总线上，要通过platform_driver和platform_device的匹配来执行。因此尽管I2C适配器给别人提供了总线，它自己也被认为是接在platform总线上的一个客户。Linux的总线、设备和驱动模型实际上是一个树形结构，每个节点虽然可能成为别人的总线控制器，但是自己也被认为是从上一级总线枚举出来的

通常我们会在I2C适配器所对应的platform_driver的probe()函数中完成两个工作

- 初始化I2C适配器所使用的的硬件资源，
- 添加适配器

通常我们会在platform_driver的remove（）函数中完成与加载函数相反的工作。

还没找到在哪个文件去添加适配器的驱动

i2总线的通信方法：

i2c设备驱动：

I2C设备驱动要使用i2c_driver和i2c_client数据结构并填充i2c_driver中的成员函数。i2c_client一般被包含在设备的私有信息结构体yyy_data中，而i2c_driver则适合被定义为全局变量并初始化。

模板：

```
1  static struct i2c_driver yyy_driver = {
2 .driver = {
3 .name = "yyy",
4 } ,
5 .probe = yyy_probe,
6 .remove = yyy_remove,
7 .id_table = yyy_id,
8  };
```

i2c设备驱动的模块加载与卸载：

1.I2C设备驱动的模块加载函数通过I2C核心的i2c_add_driver（）API函数添加i2c_driver的工作，而模块卸载函数需要做相反的工作：通过I2C核心的i2c_del_driver（）函数删除i2c_driver。

```
static int __init pm805_i2c_init(void)
{
    return i2c_add_driver(&pm805_driver);
}
subsys_initcall(pm805_i2c_init);
```

2.直接调用module_i2c_driver();

```
module_i2c_driver(adv7170_driver);
```

i2c设备驱动的数据传输

在I2C设备上读写数据的时序且数据通常通过i2c_msg数组进行组织，最后通过i2c_transfer（）函数完成