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。

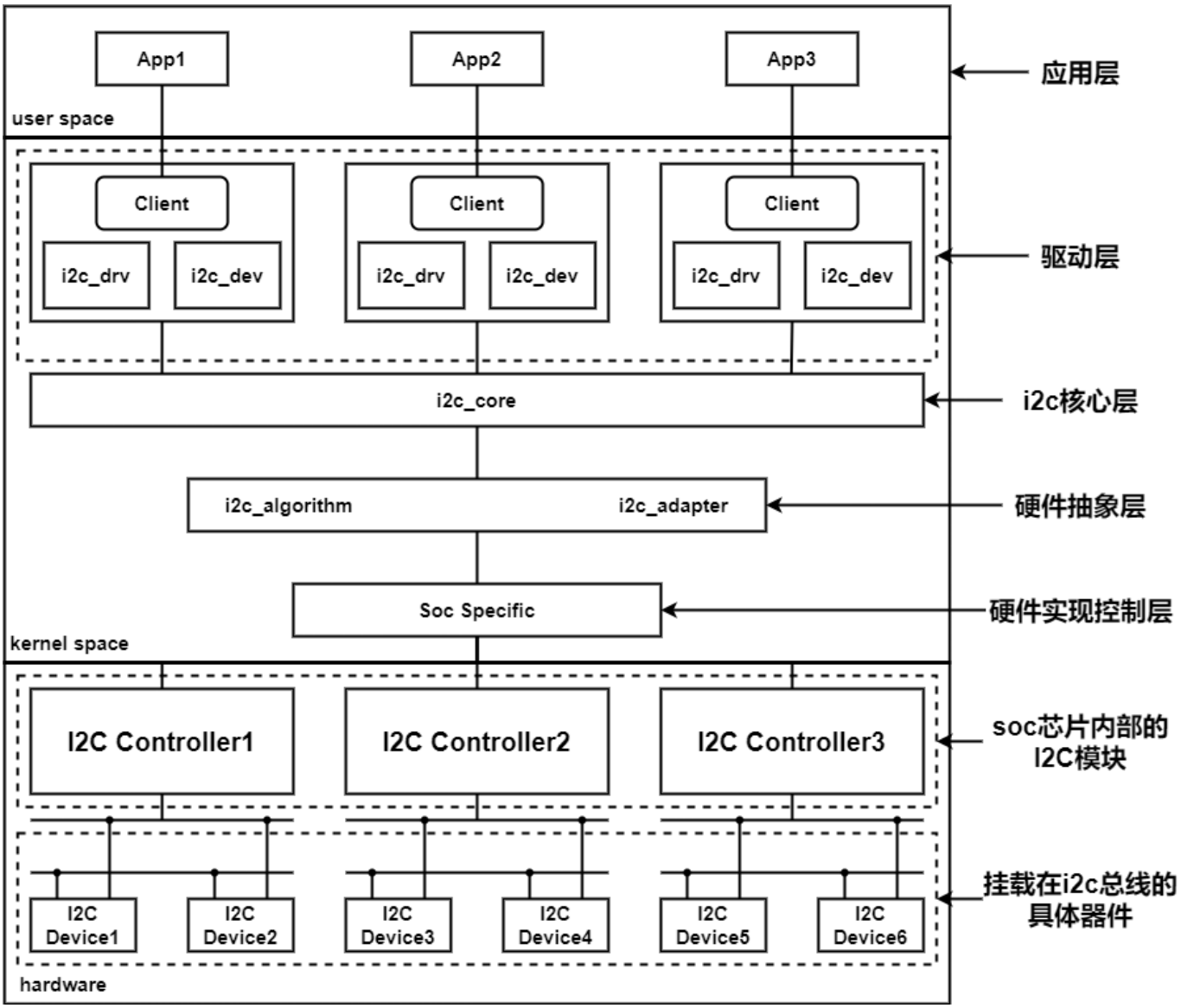
由于本文重点是梳理Linux中I2C驱动框架，所以对于I2C的通信协议、物理总线等基础内容不做过多赘述，读者可自行百度了解。

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重要数据结构。
- 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

```
/home/usr/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
├── 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，由内核开发者实现的，与硬件无关的代码。主要为其他各部分提供操作接口，在其内部通过结构体里面的函数指针调用硬件相关信息，即结构体里面函数指针的函数在设备加载的时候初始化
├── 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，供应用层直接控制I2C控制器访问I2C设备的存储空间或寄存器。
├── 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)

```
/home/usr/Documents/OpenHarmony/out/kernel/src_tmp/linux-5.10/drivers/input/touchscreen/gt1x/
├── gt1x.c          // 设备驱动主要代码位置
├── gt1x_cfg.h
├── gt1x_extents.c
├── gt1x_firmware.h
├── gt1x_generic.c
├── 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

```
/home/usr/Documents/OpenHarmony/out/kernel/src_tmp/linux-5.10/arch/arm64/boot/dts/rockchip/
├── ...
```

```
├── rk3568.dtsi
├── rk3568-linux.dtsi
├── rk3568-toybrick-mipi-tx0-beiqicloud.dtsi
├── rk3568-toybrick-x0-linux.dts
├── ...
```

3.1.4. 重要的库

```
/home/usr/Documents/OpenHarmony/out/kernel/src_tmp/linux-
5.10/include/linux/
├── ...
├── device.h
├── ...
├── i2c.h
├── of.h
├── of_device.h
├── ...
```

3.2. 关键数据结构定义

3.2.1. I2C适配器定义

```
/* /home/usr/Documents/OpenHarmony/out/kernel/src_tmp/linux-
5.10/drivers/i2c/busses/i2c-core-base.c */
/*
 * i2c_adapter is the structure used to identify a physical i2c bus along
 * with the access algorithms necessary to access it.
 */
struct i2c_adapter {
    struct module *owner;               /* 模块拥有者 */
    unsigned int class;                 /* classes to allow probing for */
    const struct i2c_algorithm *algo;   /* the algorithm to access the bus
*/
    void *algo_data;                    /* i2c_algorithm的私有数据 */

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

    int timeout;                        /* in jiffies 超过该事件无法重发*/
    int retries;                        /* I2C发送失败重发次数 */
    struct device dev;                  /* the adapter device */
    unsigned long locked_flags;         /* owned by the I2C core */
#define I2C_ALF_IS_SUSPENDED        0
#define I2C_ALF_SUSPEND_REPORTED    1

    int nr;                             /* 适配器编号，在创建i2c_client的时候会
```

```
根据编号分类，若置为-1，则代表动态分配 */
    char name[48];                      /* 适配器的名字 */
    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;

    struct irq_domain *host_notify_domain;
};
```

3.2.2. I2C通信方法定义

```
/* /home/usr/Documents/OpenHarmony/out/kernel/src_tmp/linux-
5.10/include/linux/i2c.h */
/**
 * struct i2c_algorithm - represent I2C transfer method
 * @master_xfer: Issue a set of i2c transactions to the given I2C adapter
 *   defined by the msgs array, with num messages available to transfer via
 *   the adapter specified by adap.
 * @master_xfer_atomic: same as @master_xfer. Yet, only using atomic
context
 *   so e.g. PMICs can be accessed very late before shutdown. Optional.
 * @smbus_xfer: Issue smbus transactions to the given I2C adapter. If this
 *   is not present, then the bus layer will try and convert the SMBus
calls
 *   into I2C transfers instead.
 * @smbus_xfer_atomic: same as @smbus_xfer. Yet, only using atomic context
 *   so e.g. PMICs can be accessed very late before shutdown. Optional.
 * @functionality: Return the flags that this algorithm/adapter pair
supports
 *   from the ``I2C_FUNC_*`` flags.
 * @reg_slave: Register given client to I2C slave mode of this adapter
 * @unreg_slave: Unregister given client from I2C slave mode of this
adapter
 *
 * The following structs are for those who like to implement new bus
drivers:
 * i2c_algorithm is the interface to a class of hardware solutions which
can
 * be addressed using the same bus algorithms - i.e. bit-banging or the
PCF8584
 * to name two of the most common.
 *
 * The return codes from the ``master_xfer{_atomic}`` fields should
indicate the
 * type of error code that occurred during the transfer, as documented in
the
 * Kernel Documentation file Documentation/i2c/fault-codes.rst.
```

```
 */
struct i2c_algorithm {
    /*
     * If an adapter algorithm can't do I2C-level access, set master_xfer
     * to NULL. If an adapter algorithm can do SMBus access, set
     * smbus_xfer. If set to NULL, the SMBus protocol is simulated
     * using common I2C messages.
     *
     * master_xfer should return the number of messages successfully
     * processed, or a negative value on error
     */

     // 普通I2C数据通讯协议
    int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
               int num);
    /**
     * 可选的函数，功能与master_xfer相同，不过是在atomic context环境下使用。
     * 比如在关机之前，所有中断都关闭的情况下，可用这个函数来访问电源管理芯片。
     */
    int (*master_xfer_atomic)(struct i2c_adapter *adap,
                  struct i2c_msg *msgs, int num);

    // SMBus协议，SMBus协议大部分基于I2C总线规范，并在I2C基础上扩展，在访问时序上有一
些差异
    int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,
              unsigned short flags, char read_write,
              u8 command, int size, union i2c_smbus_data *data);
    /**
     * 可选的函数，功能与smbus_xfer相同，不过是在atomic context环境下使用。
     * 比如在关机之前，所有中断都关闭的情况下，可用这个函数来访问电源管理芯片。
     */
    int (*smbus_xfer_atomic)(struct i2c_adapter *adap, u16 addr,
                 unsigned short flags, char read_write,
                 u8 command, int size, union i2c_smbus_data *data);

    /* To determine what the adapter supports */
    // 指向返回适配器支持功能的函数的指针，查看适配的能力。这些功能都是以宏定义的方式表
示，定义在include/linux/i2c.h中，以I2C_FUNC开头
    u32 (*functionality)(struct i2c_adapter *adap);

#if IS_ENABLED(CONFIG_I2C_SLAVE)
    int (*reg_slave)(struct i2c_client *client);        //  有些i2c_adapter
也可工作于Slave模式，用来实现或模拟一个I2C设备
    int (*unreg_slave)(struct i2c_client *client);        // 反注册
#endif
};
```

```
/* /usr/include/linux/i2c.h */
// i2c_algorithm中通信函数的基本单位
/**
```

```
 * struct i2c_msg - an I2C transaction segment beginning with START
 * @addr: Slave address, either seven or ten bits.  When this is a ten
 *  bit address, I2C_M_TEN must be set in @flags and the adapter
 *  must support I2C_FUNC_10BIT_ADDR.
 * @flags: I2C_M_RD is handled by all adapters.  No other flags may be
 *  provided unless the adapter exported the relevant I2C_FUNC_*
 *  flags through i2c_check_functionality().
 * @len: Number of data bytes in @buf being read from or written to the
 *  I2C slave address.  For read transactions where I2C_M_RECV_LEN
 *  is set, the caller guarantees that this buffer can hold up to
 *  32 bytes in addition to the initial length byte sent by the
 *  slave (plus, if used, the SMBus PEC); and this value will be
 *  incremented by the number of block data bytes received.
 * @buf: The buffer into which data is read, or from which it's written.
 *
 * An i2c_msg is the low level representation of one segment of an I2C
 * transaction.  It is visible to drivers in the @i2c_transfer() procedure,
 * to userspace from i2c-dev, and to I2C adapter drivers through the
 * @i2c_adapter.@master_xfer() method.
 *
 * Except when I2C "protocol mangling" is used, all I2C adapters implement
 * the standard rules for I2C transactions.  Each transaction begins with a
 * START.  That is followed by the slave address, and a bit encoding read
 * versus write.  Then follow all the data bytes, possibly including a byte
 * with SMBus PEC.  The transfer terminates with a NAK, or when all those
 * bytes have been transferred and ACKed.  If this is the last message in a
 * group, it is followed by a STOP.  Otherwise it is followed by the next
 * @i2c_msg transaction segment, beginning with a (repeated) START.
 *
 * Alternatively, when the adapter supports I2C_FUNC_PROTOCOL_MANGLING then
 * passing certain @flags may have changed those standard protocol
behaviors.
 * Those flags are only for use with broken/nonconforming slaves, and with
 * adapters which are known to support the specific mangling options they
 * need (one or more of IGNORE_NAK, NO_RD_ACK, NOSTART, and REV_DIR_ADDR).
 */
struct i2c_msg {

    __u16 addr;                 /* slave address, 从机在I2C总线上的地址  */
    __u16 flags;                        /* 消息特征标志 */

#define I2C_M_RD        0x0001  /* read data, from slave to master */

                    /* I2C_M_RD is guaranteed to be 0x0001! */

#define I2C_M_TEN       0x0010  /* this is a ten bit chip address */
#define I2C_M_DMA_SAFE      0x0200  /* the buffer of this message is DMA
safe */

                    /* makes only sense in kernelspace */
                    /* userspace buffers are copied anyway */

#define I2C_M_RECV_LEN      0x0400  /* length will be first received byte
*/
```

```
#define I2C_M_NO_RD_ACK     0x0800  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_IGNORE_NAK    0x1000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_REV_DIR_ADDR  0x2000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_NOSTART       0x4000  /* if I2C_FUNC_NOSTART */
#define I2C_M_STOP      0x8000  /* if I2C_FUNC_PROTOCOL_MANGLING */

    __u16 len;      /* msg length                   */
    __u8 *buf;      /* pointer to msg data          */

};
```

3.2.3. I2C(从机)设备驱动定义

```
/* /home/usr/Documents/OpenHarmony/out/kernel/src_tmp/linux-
5.10/include/linux/i2c.h */
/**
 * struct i2c_driver - represent an I2C device driver
 * @class: What kind of i2c device we instantiate (for detect)
 * @probe: Callback for device binding - soon to be deprecated
 * @probe_new: New callback for device binding
 * @remove: Callback for device unbinding
 * @shutdown: Callback for device shutdown
 * @alert: Alert callback, for example for the SMBus alert protocol
 * @command: Callback for bus-wide signaling (optional)
 * @driver: Device driver model driver
 * @id_table: List of I2C devices supported by this driver
 * @detect: Callback for device detection
 * @address_list: The I2C addresses to probe (for detect)
 * @clients: List of detected clients we created (for i2c-core use only)
 *
 * The driver.owner field should be set to the module owner of this driver.
 * The driver.name field should be set to the name of this driver.
 *
 * For automatic device detection, both @detect and @address_list must
 * be defined. @class should also be set, otherwise only devices forced
 * with module parameters will be created. The detect function must
 * fill at least the name field of the i2c_board_info structure it is
 * handed upon successful detection, and possibly also the flags field.
 *
 * If @detect is missing, the driver will still work fine for enumerated
 * devices. Detected devices simply won't be supported. This is expected
 * for the many I2C/SMBus devices which can't be detected reliably, and
 * the ones which can always be enumerated in practice.
 *
 * The i2c_client structure which is handed to the @detect callback is
 * not a real i2c_client. It is initialized just enough so that you can
 * call i2c_smbus_read_byte_data and friends on it. Don't do anything
 * else with it. In particular, calling dev_dbg and friends on it is
 * not allowed.
 */
```

```
struct i2c_driver {
    unsigned int class;

    /* Standard driver model interfaces */
    // probe和remove必须实现
    int (*probe)(struct i2c_client *client, const struct i2c_device_id
*id);
    int (*remove)(struct i2c_client *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 *client);

    /* driver model interfaces that don't relate to enumeration  */
    void (*shutdown)(struct i2c_client *client);    // 关机
    int (*suspend)(struct i2c_client *, pm_message_t mesg);   // 挂起
    int (*resume)(struct i2c_client *);     // 恢复
    /* 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 *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; // 在注册i2c_driver对象时，i2c_driver-
>driver的总线类型被指定为i2c_bus_type
    const struct i2c_device_id *id_table;           // 存放该驱动支持的设备列
表，驱动和设备匹配时会用到

    /* Device detection callback for automatic device creation */
    int (*detect)(struct i2c_client *client, struct i2c_board_info *info);
// 基于设备探测机制实现的 12C 设备驱动：设备探测的回调函数
    const unsigned short *address_list;     // 设备探测的地址范围
    struct list_head clients;   // 探测到的设备列表
};
#define to_i2c_driver(d) container_of(d, struct i2c_driver, driver)
```

3.2.4. I2C(从机)设备信息定义

```
/* /home/usr/Documents/OpenHarmony/out/kernel/src_tmp/linux-
5.10/include/linux/i2c.h */
```

```
/**
 * struct i2c_client - represent an I2C slave device
 * @flags: see I2C_CLIENT_* for possible flags
 * @addr: Address used on the I2C bus connected to the parent adapter.
 * @name: Indicates the type of the device, usually a chip name that's
 *  generic enough to hide second-sourcing and compatible revisions.
 * @adapter: manages the bus segment hosting this I2C device
 * @dev: Driver model device node for the slave.
 * @init_irq: IRQ that was set at initialization
 * @irq: indicates the IRQ generated by this device (if any)
 * @detected: member of an i2c_driver.clients list or i2c-core's
 *  userspace_devices list
 * @slave_cb: Callback when I2C slave mode of an adapter is used. The
adapter
 *  calls it to pass on slave events to the slave driver.
 *
 * An i2c_client identifies a single device (i.e. chip) connected to an
 * i2c bus. The behaviour exposed to Linux is defined by the driver
 * managing the device.
 */
struct i2c_client {
    unsigned short flags;       /* div., see below      */
#define I2C_CLIENT_PEC      0x04    /* Use Packet Error Checking 设备使用
SMBus包错误检查 */
#define I2C_CLIENT_TEN      0x10    /* we have a ten bit chip address 设备使
用10bit地址 */
                    /* Must equal I2C_M_TEN below */
#define I2C_CLIENT_SLAVE    0x20    /* we are the slave */
#define I2C_CLIENT_HOST_NOTIFY  0x40    /* We want to use I2C host notify
*/
#define I2C_CLIENT_WAKE     0x80    /* for board_info; true iff can wake */
#define I2C_CLIENT_SCCB     0x9000  /* Use Omnivision SCCB protocol */
                    /* Must match I2C_M_STOP|IGNORE_NAK */

    unsigned short addr;        /* chip address - NOTE: 7bit    */
                    /* addresses are stored in the  */
                    /* _LOWER_ 7 bits       */
    char name[I2C_NAME_SIZE];       // 设备的名称
    struct i2c_adapter *adapter;    /* the adapter we sit on    */
    /**
     * 内嵌的device结构体，在注册i2c_client对象时，
     * i2c_client->dev的总线类型被指定为i2c_bus_type，
     * 其type成员被指定为i2c_client_type
     */
    struct device dev;      /* the device structure     */
    int init_irq;           /* irq set at initialization    */
    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
};
#define to_i2c_client(d) container_of(d, struct i2c_client, dev)
```

3.2.5. 适配器要支持的I2C设备信息(不使用设备树时使用)

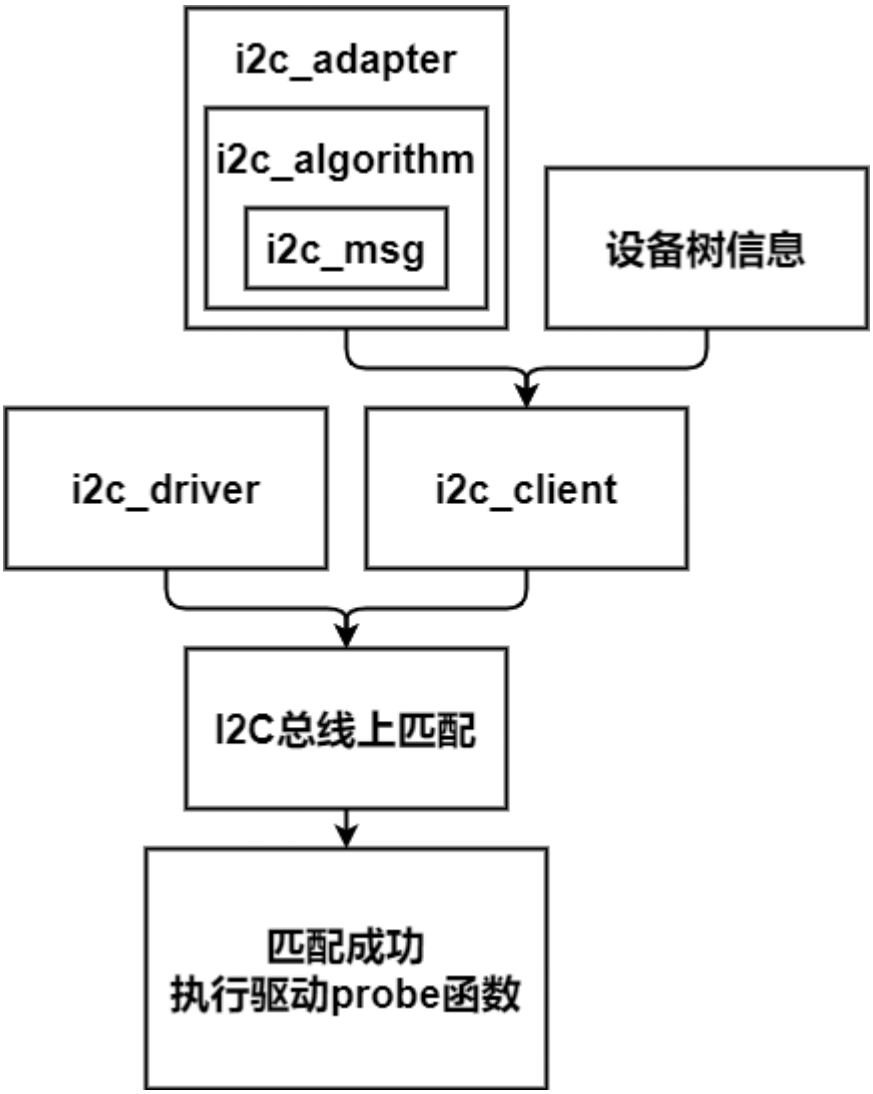
```
/*/* /home/usr/Documents/OpenHarmony/out/kernel/src_tmp/linux-
5.10/include/linux/i2c.h */
/**
 * struct i2c_board_info - template for device creation
 * @type: chip type, to initialize i2c_client.name
 * @flags: to initialize i2c_client.flags
 * @addr: stored in i2c_client.addr
 * @dev_name: Overrides the default <busnr>-<addr> dev_name if set
 * @platform_data: stored in i2c_client.dev.platform_data
 * @of_node: pointer to OpenFirmware device node
 * @fwnode: device node supplied by the platform firmware
 * @properties: additional device properties for the device
 * @resources: resources associated with the device
 * @num_resources: number of resources in the @resources array
 * @irq: stored in i2c_client.irq
 *
 * I2C doesn't actually support hardware probing, although controllers and
 * devices may be able to use I2C_SMBUS_QUICK to tell whether or not
there's
 * a device at a given address.  Drivers commonly need more information
than
 * that, such as chip type, configuration, associated IRQ, and so on.
 *
 * i2c_board_info is used to build tables of information listing I2C
devices
 * that are present.  This information is used to grow the driver model
tree.
 * For mainboards this is done statically using i2c_register_board_info();
 * bus numbers identify adapters that aren't yet available.  For add-on
boards,
 * i2c_new_client_device() does this dynamically with the adapter already
known.
 */
struct i2c_board_info {
    char        type[I2C_NAME_SIZE];
    unsigned short  flags;
    unsigned short  addr;
    const char  *dev_name;
    void        *platform_data;
    struct device_node *of_node;
    struct fwnode_handle *fwnode;
    const struct property_entry *properties;
    const struct resource *resources;
    unsigned int    num_resources;
    int     irq;
};

// 初始化结构i2c_board_info的基本字段，声明在特定板上提供的内容
```

```
/**
 * I2C_BOARD_INFO - macro used to list an i2c device and its address
 * @dev_type: identifies the device type
 * @dev_addr: the device's address on the bus.
 *
 * This macro initializes essential fields of a struct i2c_board_info,
 * declaring what has been provided on a particular board.  Optional
 * fields (such as associated irq, or device-specific platform_data)
 * are provided using conventional syntax.
 */
#define I2C_BOARD_INFO(dev_type, dev_addr) \
    .type = dev_type, .addr = (dev_addr)
```

3.2.6. 关键结构体之间的关联

下图解释了上述关键结构体之间的关联。在I2C设备驱动注册的过程中，会调用驱动的匹配函数`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，自行构造i2c_msg实现对I2C设备的控制操作。



4. Linux中I2C驱动框架代码流程分析

5. 结语

6. 参考资料