各种总线 match 匹配函数

http://blog.csdn.net/fanqipin/article/details/8153053

当向 linux 系统总线添加设备或驱动时，总是会调用各总线对应的 match 匹配函数来判断驱动和设备是否匹配，这些 match 函数之间都存在一定的差异，本文先对常用的 match 匹配函数进行讲解，以后会陆续添加新的内容。

一. 驱动和设备匹配过程常用数据结构

1. of_device_id

```
struct of_device_id
{
charname[32];
char type[32];
char compatible[128];
#ifdef __KERNEL__
void*data;
#else
kernel_ulong_t data;
#endif

};
```

2. platform_device_id

```
struct platform_device_id {

char name[PLATFORM_NAME_SIZE];

kernel_ulong_t driver_data    __attribute__((aligned(sizeof(kernel_ulong_t))));

};
```

二. 平台设备、驱动匹配 platform_match

向系统添加平台驱动或添加设备时会调用平台总线 platform_bus_type 中的 platform_match 函数来匹配平台驱动和平台设备。

```
static int platform_match(struct device *dev, struct device_driver *drv)
{

struct platform_device *pdev = to_platform_device(dev);

struct platform_driver *pdrv = to_platform_driver(drv);

/*通过驱动里定义了 of_device_id 项，则通过这一项来比对；*

if (of_driver_match_device(dev, drv))

return 1;

/*如果在平台驱动中定义了 id_table 项，则通过对比 id_table 来判断*/

if (pdrv->id_table)

return platform_match_id(pdrv->id_table, pdev) != NULL;

/*通过对比平台设备名字和平台驱动名字来判断*/

return (strcmp(pdev->name, drv->name) == 0);

}
```

由 platform_match 可以看出，驱动和设备是否匹配可以通过三种方式来进行判断，首先是通过 of_device_id 结构：

```
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;
```

```
}

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)

{

if (!matches)

return NULL;

while (matches->name[0] || matches->type[0] || matches->compatible[0]) {

int match = 1;
if (matches->name[0])
match &= node->name && !strcmp(matches->name, node->name);
if (matches->type[0])
match &= node->type && !strcmp(matches->type, node->type);
if (matches->compatible[0])
match &= of_device_is_compatible(node, matches->compatible);
if (match)
return matches;

matches++;

}
```

```
return NULL;

}
```

如果 driver 中定义了 of_device_id，则通过 **driver 中的 of_device_id 和 device 中的 device_node** 内容进行匹配判断，匹配工作由 of_match_node 来完成，该函数会遍历 of_device_id 列表，查找是否有成员与 device_node 相匹配，具体由 matches 的 name,type 和 compatioble 来进行对比，如果找到则返回相应的表项，否则返回 null.如果没有定义 of_device_id,device_node 或不能找到对应的匹配项，则通过第二种方式 platform_device_id 来进行对比匹配,通过 platform_match_id 来完成：

```
static const struct platform_device_id *platform_match_id( const struct platform_device_id *id, struct platform_device *pdev)
{

while (id->name[0]) {

if (strcmp(pdev->name, id->name) == 0) {

pdev->id_entry = id;
return id;

}

id++;

}

return NULL;

}
```

platform_match_id 函数遍历 platfrom_device_id 列表，通过比对平台设备与 id 的 name 来确定是否有匹配项，如果找到匹配的，则返回对应的 id 项，否则返回 null。如果没有定义 platform_device_id 或没有找到匹配项，则通过第三种方式进行匹配，第三种方式通过比对平台设备和平台驱动的名字，如果相等，则匹配成功，否则失败。

三. i2c 设备、驱动匹配 i2c_device_match

当向 i2c 总线添加驱动或设备时会调用 i2c_device_match 来进行匹配判断，i2c_device_match 函数定义如下所示：

```
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;

/*  通过 of_device_id 匹配  */

if (of_driver_match_device(dev, drv))

return 1;

driver = to_i2c_driver(drv);

/*如果 I2C  驱动中定义了 id_table，则通过 id_table 进行匹配；*/

if (driver->id_table)

return i2c_match_id(driver->id_table, client) != NULL;

return 0;

}
```

如 i2c_device_match 所示，i2c 通过两种方式进行匹配设备和驱动，一种是 of_device_id，另一种是 i2c_device_id，i2c_device_id 数据结构和 platform_device_id 一样。I2C 里的两种匹配方式和之前的 platform 判断方式都是一样，这里就不展开。

四. usb 设备、驱动匹配 usb_device_match

当向 usb 总线上注册驱动或添加设备时，就会调用 usb_match_device 进行驱动和设备配对，函数如下：

```
static int usb_device_match(struct device *dev, struct device_driver *drv)

{

if (is_usb_device(dev)) {

if (!is_usb_device_driver(drv))

return 0;

return 1;

} else if (is_usb_interface(dev)) {

struct usb_interface *intf;

struct usb_driver *usb_drv;

const struct usb_device_id *id;

if (is_usb_device_driver(drv))

return 0;

intf = to_usb_interface(dev);

usb_drv = to_usb_driver(drv);

id = usb_match_id(intf, usb_drv->id_table);

if (id)

return 1;
```

```
id = usb_match_dynamic_id(intf, usb_drv);

if (id)

return 1;

}

return 0;

}
```

从函数可以看出，match 分成两部分，一部分用于匹配 usb 设备，另一部分用于匹配 usb 接口，对于 usb 设备，在初始化时会设置成 usb_device_type，而 usb 接口，则会设成 usb_if_device_type。而函数中的 is_usb_device 和 is_usb_interface 就是通过这两个属性来判别的，如果为判定为设备，则进入到设备分支，否则进入到接口分支继续判断。

usb 设备驱动通过 usb_register_device_driver 接口来注册到系统，而 usb 接口驱动则通过 usb_register 来注册到系统，驱动工程师的工作基本上集中在接口驱动上，所以通常是通过 usb_register 来注册 usb 驱动的。

不管是设备驱动 usb_device_driver,还是接口驱动 usb_driver 数据结构中都包含了 struct usbdrv_wrap 项，其定义如下：

```
struct usbdrv_wrap {

struct device_driver driver;

int for_devices;

}
```

数据结构中的 for_devices 用来表示该驱动是设备驱动还是接口驱动，如果为设备驱动，则在用 usb_register_device_driver 注册时，会将该变量 for_devices 设置成 1，而接口驱动则设为 0.

usb_device_match 中的 is_usb_device_driver 函数就是通过获取上而结构中的 for_devices 来进行判断是设备还是接口驱动的，函数定义如下：

```
static inline int is_usb_device_driver(struct device_driver *drv)
```

{

return container_of(drv, struct usbdrv_wrap, driver)->for_devices;

}

当进入 is_usb_device 分支后，再通过 is_usb_device_driver 来判断是否为设备驱动，如果是则返回 1，表示匹配成功，它接受所有 usb 设备。

当进入到接口分支后，也会先用 is_usb_device_driver 来进行判断，如果不是设备驱动则继续判断，否则退出；然后再通过 usb_match_id 函数来判断设备和驱动中的 usb_device_id 是否匹配，usb_match_id 定义如下：

const struct usb_device_id *usb_match_id(struct usb_interface *interface, const struct usb_device_id *id)
{

if (id == NULL)

return NULL;

for (; id->idVendor || id->idProduct || id->bDeviceClass || id->bInterfaceClass || id->driver_info; id++) {

if (usb_match_one_id(interface, id))

return id;

}

return NULL;

}

遍历接口驱动中的 usb_device_id 列表项，只要 usb_device_id 结构中的 idVendor,idProduct,DeviceClass,binterfaceClass,driver_info 项有效就调用 usb_match_one_id 进行判断，如找到匹配项则函数返回 1，否则返回 0 。

int usb_match_one_id(struct usb_interface *interface,const struct usb_device_id *id)
{

```
struct usb_host_interface *intf;

struct usb_device *dev;

if (id == NULL)

return 0;

intf = interface->cur_altsetting;

dev = interface_to_usbdev(interface);

if (!usb_match_device(dev, id))

return 0;

if (dev->descriptor.bDeviceClass == USB_CLASS_VENDOR_SPEC &&
 !(id->match_flags & USB_DEVICE_ID_MATCH_VENDOR) &&

(id->match_flags & (USB_DEVICE_ID_MATCH_INT_CLASS
| USB_DEVICE_ID_MATCH_INT_SUBCLASS
| USB_DEVICE_ID_MATCH_INT_PROTOCOL)))

return 0;

if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_CLASS) &&

 (id->bInterfaceClass != intf->desc.bInterfaceClass))

return 0;

if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_SUBCLASS) &&

(id->bInterfaceSubClass != intf->desc.bInterfaceSubClass))

return 0;

if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_PROTOCOL) &&

 (id->bInterfaceProtocol != intf->desc.bInterfaceProtocol))
```

```
return 0;

return 1;

}
```

usb_match_one_id 和函数中的 usb_match_device 都是围绕着 usb_device_id 进行匹配的，该结构定义如下：

```
struct usb_device_id {

/* which fields to match against? */

__u16   match_flags;

/* Used for product specific matches; range is inclusive */

__u16   idVendor;

__u16   idProduct;

__u16   bcdDevice_lo;

__u16   bcdDevice_hi;

/* Used for device class matches */

__u8   bDeviceClass;

__u8   bDeviceSubClass;

__u8   bDeviceProtocol;

/* Used for interface class matches */

__u8   bInterfaceClass;

__u8   bInterfaceSubClass;

__u8   bInterfaceProtocol;
```

```
/* not matched against */

kernel_ulong_tdriver_info;

};
```

match_flags 用来规定驱动匹配时的具体项，如 match_flags 包含 USB_DEVICE_ID_MATCH_VENDOR，则是通过驱动中的 usb_device_id 和设备 dev 中的 idVendor 来判断。