一文搞懂 Linux 网络 Phy 驱动

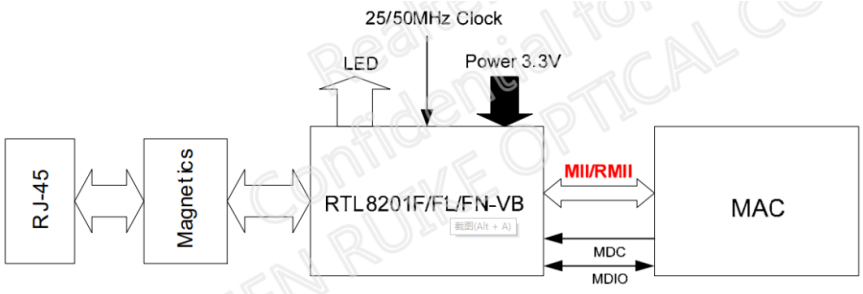
原创 BSP实战 人人极客社区 2023-11-02 08:08 发表于江苏

- 概述
- 数据结构
  - phy_device
  - phy_driver
  - mii_bus
  - net_device
- phy 设备的注册
- phy 驱动的注册
  - genphy_driver 通用驱动
  - NXP TJA 专有驱动
- 网卡 fec 和 Phy 的协作
  - 自协商配置
  - link 状态读取
  - link 状态通知

Linux BSP实战课（网络篇）：数据包的发送过程

Linux BSP实战课（网络篇）：数据包的接收过程

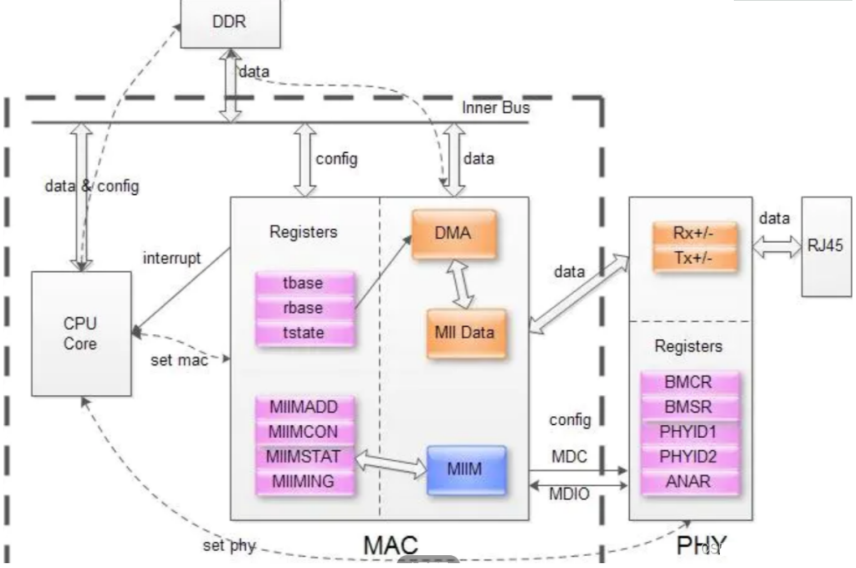
概述



上图来自 瑞昱半导体 (RealTek) 的 RTL8201F 系列网卡 PHY 芯片手册。按OSI 7层网络模型划分，网卡PHY 芯片(图中的RTL8201F)位于物理层，对应的软件层就是本文讨论的 PHY 驱动层；而 MAC 位于 数据链路层，也是通常软件上所说的网卡驱动层，它不是本文的重点，不做展开。另外，可通过 MDIO 接口对 PHY 芯片进行配置（如PHY芯片寄存器读写），而 PHY 和 MAC 通过 MII/RMII 进行数据传输。

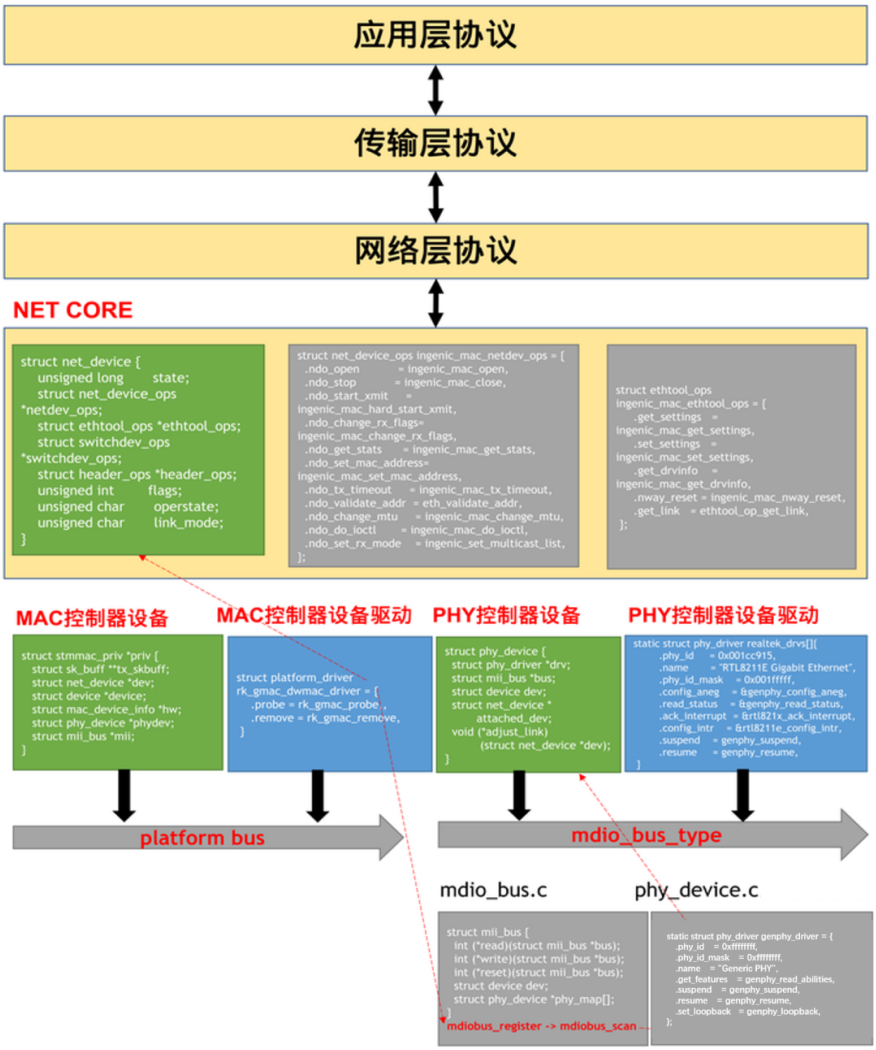
PHY芯片通过MII/GMII/RMII/SGMII/XGMII等多种媒体独立接口（介质无关接口）与数据链路层的MAC芯片相连，并通过MDIO接口实现对PHY 状态的监控、配置和管理。

PHY与MAC整体的连接框图：



数据结构

每个 phy 芯片会创建一个 struct phy_device 类型的设备，对应的有 struct phy_driver 类型的驱动，这两者实际上是挂载在 mdio_bus_type 总线上的，mac 会被注册成 struct net_device。



phy_device

```c
struct phy_device {
    struct phy_driver *drv;             // PHY设备驱动
    struct mii_bus *bus;                // 对应的MII总线
    struct device dev;                  // 设备文件
    u32 phy_id;                         // PHY ID

    struct phy_c45_device_ids c45_ids;
    bool is_c45;
```

```
    bool is_internal;
    bool has_fixups;
    bool suspended;

    enum phy_state state;                   // PHY状态
    u32 dev_flags;
    phy_interface_t interface;              // PHY接口
    int addr;                               // PHY 总线地址(0~31)

    int speed;                              // 速度
    int duplex;                             // 双工模式
    int pause;                              // 停止
    int asym_pause;
    int link;

    u32 interrupts;                         // 中断使能标志
    u32 supported;
    u32 advertising;
    u32 lp_advertising;
    int autoneg;
    int link_timeout;
    int irq;                                // 中断号

    void *priv;                             // 私有数据
    struct work_struct phy_queue;           // PHY工作队列
    struct delayed_work state_queue;        // PHY延时工作队列
    atomic_t irq_disable;

    struct mutex lock;

    struct net_device *attached_dev;        // 网络设备

    void (*adjust_link)(struct net_device *dev);
};
```

phy_driver

```
struct phy_driver {
    struct mdio_driver_common mdiodrv;
    u32 phy_id;
    char *name;
    u32 phy_id_mask;
    u32 features;
    u32 flags;
    const void *driver_data;

    int (*soft_reset)(struct phy_device *phydev);
    int (*config_init)(struct phy_device *phydev);
    int (*probe)(struct phy_device *phydev);
    int (*suspend)(struct phy_device *phydev);
    int (*resume)(struct phy_device *phydev);
    int (*config_aneg)(struct phy_device *phydev);
    int (*aneg_done)(struct phy_device *phydev);
    int (*read_status)(struct phy_device *phydev);
    int (*ack_interrupt)(struct phy_device *phydev);
    int (*config_intr)(struct phy_device *phydev);
    int (*did_interrupt)(struct phy_device *phydev);
    void (*remove)(struct phy_device *phydev);
    int (*match_phy_device)(struct phy_device *phydev);
    int (*ts_info)(struct phy_device *phydev, struct ethtool_ts_info *ti);
    int  (*hwtstamp)(struct phy_device *phydev, struct ifreq *ifr);
    bool (*rxtstamp)(struct phy_device *dev, struct sk_buff *skb, int type);
    void (*txtstamp)(struct phy_device *dev, struct sk_buff *skb, int type);
    int (*set_wol)(struct phy_device *dev, struct ethtool_wolinfo *wol);
    void (*get_wol)(struct phy_device *dev, struct ethtool_wolinfo *wol);
    void (*link_change_notify)(struct phy_device *dev);
    int (*read_mmd)(struct phy_device *dev, int devnum, u16 regnum);
```

```
    int (*write_mmd)(struct phy_device *dev, int devnum, u16 regnum,
            u16 val);
    int (*read_page)(struct phy_device *dev);
    int (*write_page)(struct phy_device *dev, int page)
    int (*module_info)(struct phy_device *dev,
              struct ethtool_modinfo *modinfo);
    int (*module_eeprom)(struct phy_device *dev,
                struct ethtool_eeprom *ee, u8 *data);
    int (*get_sset_count)(struct phy_device *dev);
    void (*get_strings)(struct phy_device *dev, u8 *data);
    void (*get_stats)(struct phy_device *dev,
             struct ethtool_stats *stats, u64 *data);
    int (*get_tunable)(struct phy_device *dev,
              struct ethtool_tunable *tuna, void *data);
    int (*set_tunable)(struct phy_device *dev,
               struct ethtool_tunable *tuna,
               const void *data);
    int (*set_loopback)(struct phy_device *dev, bool enable);
    ANDROID_KABI_RESERVE(1);
    ANDROID_KABI_RESERVE(2);
};
```

mii_bus

```
struct mii_bus {
    const char *name;             // 总线名字
    char id[MII_BUS_ID_SIZE];     // ID MII_BUS_ID_SIZE=61
    void *priv;                   // 私有数据
    int (*read)(struct mii_bus *bus, int phy_id, int regnum);              // 读方式
    int (*write)(struct mii_bus *bus, int phy_id, int regnum, u16 val);    // 写方式
    int (*reset)(struct mii_bus *bus);     // 复位

    struct mutex mdio_lock;

    struct device *parent;        // 父设备
    enum {
        MDIOBUS_ALLOCATED = 1,
        MDIOBUS_REGISTERED,
        MDIOBUS_UNREGISTERED,
        MDIOBUS_RELEASED,
    } state;                       // 总线状态
    struct device dev;             // 设备文件

    struct phy_device *phy_map[PHY_MAX_ADDR]; // PHY设备数组

    u32 phy_mask;
    int *irq;                   // 中断
};
```

net_device

```
struct net_device {
    char  name[IFNAMSIZ];         /* 用于存放网络设备的设备名称 */
    char  *ifalias;               /* 网络设备的别名 */
    int   ifindex;                /* 网络设备的接口索引值，独一无二的网络设备标识符 */
    struct hlist_node  name_hlist;  /* 这个字段用于构建网络设备名的哈希散列表，而struct net中的name_h
    struct hlist_node  index_hlist; /* 用于构建网络设备的接口索引值哈希散列表，在struct net中的index_
    struct list_head   dev_list;    /* 用于将每一个网络设备加入到一个网络命名空间中的网络设备双链表中 *
    unsigned int       flags;       /* 网络设备接口的标识符 */
    unsigned int       priv_flags;  /* 网络设备接口的标识符，但对用户空间不可见。*/
    unsigned short     type;        /* 接口硬件类型 */
    unsigned int       mtu;         /* 网络设备接口的最大传输单元 */
    unsigned short     hard_header_len;   /* 硬件接口头长度 */
    unsigned char      *dev_addr;    /* 网络设备接口的MAC地址 */
    bool           uc_promisc;       /* 网络设备接口的单播模式 */
```

```
    unsigned int       promiscuity;  /* 网络设备接口的混杂模式 */
    unsigned int       allmulti;     /* 网络设备接口的全组播模式 */
    struct netdev_hw_addr_list  uc;  /* 辅助单播MAC地址列表 */
    struct netdev_hw_addr_list  mc;  /* 主mac地址列表 */
    struct netdev_hw_addr_list  dev_addrs;  /* hw设备地址列表 */
    unsigned char      broadcast[MAX_ADDR_LEN];   /* hw广播地址 */
    struct netdev_rx_queue  *_rx;       /* 网络设备接口的数据包接收队列 */
    struct netdev_queue *_tx            /* 网络设备接口的数据包发送队列 */
    unsigned int        num_tx_queues;  /* TX队列数 */
    unsigned int        real_num_tx_queues;  /* 当前设备活动的TX队列数 */
    unsigned long      tx_queue_len;    /* 每个队列允许的最大帧 */
    unsigned long      state;           /* 网络设备接口的状态 */
    struct net_device_stats    stats;   /* 网络设备接口的统计情况 */
    possible_net_t         nd_net;      /* 用于执行网络设备所在的命名空间 */
};
```

phy 设备的注册

以网卡 Fec 为例，网卡驱动在初始化 fec_probe() 时遍历 dts 的定义，创建相应 struct phy_device 类型的设备，主要步骤为：

1. 注册网络设备 net_device
2. 申请队列和 DMA
3. 申请 MDIO 总线
4. 创建并注册 Phy 设备

```
fec_probe(struct platform_device *pdev)
    -> struct device_node *np = pdev->dev.of_node, *phy_node; // 获取设备树节点句柄，并创建一个phy的
    -> fec_enet_get_queue_num(pdev, &num_tx_qs, &num_rx_qs);  // 从设备树获取fsl,num-tx-queues和fs
    -> ndev = alloc_etherdev_mqs                              // 申请net_device
    -> netdev_priv(ndev)                                      //  获取私有数据空间首地址
-------------------------------------------------------------------------------------------------
    -> of_parse_phandle(np, "phy-handle", 0)              // 从mac的设备树节点中获取phy子节点
    -> of_get_phy_mode(pdev->dev.of_node)                 // 从设备树节点中获取phy模式，phy-mode =
    -> fec_reset_phy(pdev);                               // 复位phy
    -> fec_enet_init(ndev)                                // 申请队列和DMA，设置MAC地址
    -> of_property_read_u32(np, "fsl,wakeup_irq", &irq)   // 唤醒中断
    -> fec_enet_mii_init(pdev);                           // 注册MDIO总线、注册phy_device
        -> fep->mii_bus = mdiobus_alloc()                 //申请MDIO总线
        -> fep->mii_bus->name = "fec_enet_mii_bus";       // 总线名字
        -> fep->mii_bus->read = fec_enet_mdio_read;       // 总线的读函数
        -> fep->mii_bus->write = fec_enet_mdio_write;     // 总线的写函数
        -> snprintf(fep->mii_bus->id, MII_BUS_ID_SIZE, "%s-%x",
pdev->name, fep->dev_id + 1);                         // 总线id
        -> of_get_child_by_name(pdev->dev.of_node, "mdio");           // 获取phy节点句柄
        -> of_mdiobus_register          // 注册mii_bus设备，并通过设备树子节点创建PHY设备 drivers/of/
            -> mdio->phy_mask = ~0;    // 屏蔽所有PHY,防止自动探测。相反，设备树中列出的phy将在总线注册
            -> mdio->dev.of_node = np;
            -> mdio->reset_delay_us = DEFAULT_GPIO_RESET_DELAY;
            -> mdiobus_register(mdio)               // 注册MDIO总线设备
                -> bus->dev.parent = bus->parent;
                -> bus->dev.class = &mdio_bus_class;  // 总线设备类"/sys/bus/mdio_bus"
                /*----------------------------------------
                static struct class mdio_bus_class = {
                    .name = "mdio_bus",
                    .dev_release = mdiobus_release,
                };
                ----------------------------------------*/
                -> bus->dev.groups = NULL;
                -> dev_set_name(&bus->dev, "%s", bus->id);          //设置总线设备的名称
                -> device_register(&bus->dev);                      // 注册总线设备
                ->  if (bus->reset)      bus->reset(bus);           // 总线复位
-------------------------------------另一条分支解析(可忽略)-------------------------------------
```

```
                ->  phydev = mdiobus_scan(bus, i);                  // 扫描phy设备
                -> phydev = get_phy_device(bus, addr);          //获取创建phy设备
                ->err = phy_device_register(phydev);          //注册phy设备
-------------------------------------------------------------------------------------------
                -> for_each_available_child_of_node(np, child) {      // 遍历这个平台设备的子节点并
                -> addr = of_mdio_parse_addr(&mdio->dev, child)       // 从子节点的"reg"属性中获得P
                -> of_property_read_u32(np, "reg", &addr)
                -> if (addr < 0)     scanphys = true;      continue;  // 如果未获得子节点的"reg"属性
                -> of_mdiobus_register_phy(mdio, child, addr)     }     // 创建并注册PHY设备
                    -> is_c45 = of_device_is_compatible(child,"ethernet-phy-ieee802.3-c45") //判断
                -> if (!is_c45 && !of_get_phy_id(child, &phy_id))      //如果设备树中的PHY的属性未指
                -> phy = phy_device_create(mdio, addr, phy_id, 0, NULL);
                -> else    phy = get_phy_device(mdio, addr, is_c45);    //用这个分支
                        -> get_phy_id(bus, addr, &phy_id, is_c45, &c45_ids);//通过mdio得到PHY的ID
                            -> mdiobus_read(bus, addr, MII_PHYSID1)
                                -> __mdiobus_read(bus, addr, regnum);
                                    -> bus->read(bus, addr, regnum)
                                -> mdiobus_read(bus, addr, MII_PHYSID2)
                            -> phy_device_create(bus, addr, phy_id, is_c45, &c45_ids)    // 创建PH
                                -> struct phy_device *dev;
                                -> dev = kzalloc(sizeof(*dev), GFP_KERNEL);
                                    dev->dev.release = phy_device_release;
                                    dev->speed = 0;
                                    dev->duplex = -1;
                                    dev->pause = 0;
                                    dev->asym_pause = 0;
                                    dev->link = 1;
                                    dev->interface = PHY_INTERFACE_MODE_GMII;
                                    dev->autoneg = AUTONEG_ENABLE;      // 默认支持自协商(自动使能)
                                    dev->is_c45 = is_c45;
                                    dev->addr = addr;
                                    dev->phy_id = phy_id;
                                    if (c45_ids)
                                        dev->c45_ids = *c45_ids;
                                    dev->bus = bus;
                                    dev->dev.parent = bus->parent;
                                    dev->dev.bus = &mdio_bus_type;    //PHY设备和驱动都会挂在mdio_b
                                /*-------------------------------------------------------------------
                                    struct bus_type mdio_bus_type = {
                                        .name       = "mdio_bus",                //总线名称
                                        .match      = mdio_bus_match,            //用来匹配总线上设
                                        .pm         = MDIO_BUS_PM_OPS,
                                        .dev_groups = mdio_dev_groups,
                                    };
                                ------------------------------------------------------------------*/
                                    dev->irq = bus->irq != NULL ? bus->irq[addr] : PHY_POLL;
                                    dev_set_name(&dev->dev, PHY_ID_FMT, bus->id, addr);
                                    dev->state = PHY_DOWN;     //指示PHY设备和驱动程序尚未准备就绪,在
                                    -> INIT_DELAYED_WORK(&dev->state_queue, phy_state_machine);
                                    -> INIT_WORK(&dev->phy_queue, phy_change);   // 由phy_interru
                                    ->  request_module(MDIO_MODULE_PREFIX MDIO_ID_FMT, MDIO_ID_AR(
                                    -> device_initialize(&dev->dev);     //设备模型中的一些设备,主要
                                -> irq_of_parse_and_map(child, 0);      //将中断解析并映射到Linux v
                                -> of_node_get(child);         //将OF节点与设备结构相关联
                                -> phy->dev.of_node = child;
                                -> phy_device_register(phy)    // 注册phy设备
                                    -> if (phydev->bus->phy_map[phydev->addr])       //判断PHY是否
                                    -> phydev->bus->phy_map[phydev->addr] = phydev;  //添加PHY到总
                                    -> phy_scan_fixups(phydev);     //执行匹配的fixups
                                    -> device_add(&phydev->dev);    // 注册到Linux设备模型框架中
                        ->  if (!scanphys)   return 0;          // 如果从子节点的"reg"属性中获得
------------一般来说只要设备树种指定了PHY设备的"reg"属性，后面的流程可以自动忽略 ------------
    -> register_netdev(ndev)                // 向内核注册net_device
```

phy 驱动的注册

genphy_driver 通用驱动

内核中有 genphy_driver 通用驱动，以及专有驱动（这里以 NXP TJA 驱动为例），分别如下：

```
static struct phy_driver genphy_driver = {
 .phy_id  = 0xffffffff,
 .phy_id_mask = 0xffffffff,
 .name  = "Generic PHY",
 .get_features = genphy_read_abilities,
 .suspend = genphy_suspend,
 .resume  = genphy_resume,
 .set_loopback   = genphy_loopback,
};
```

```
static struct phy_driver tja11xx_driver[] = {
 {
  PHY_ID_MATCH_MODEL(PHY_ID_TJA1100),
  .name  = "NXP TJA1100",
  .features       = PHY_BASIC_T1_FEATURES,
  .probe  = tja11xx_probe,
  .soft_reset = tja11xx_soft_reset,
  .config_aneg = tja11xx_config_aneg,
  .config_init = tja11xx_config_init,
  .read_status = tja11xx_read_status,
  .get_sqi = tja11xx_get_sqi,
  .get_sqi_max = tja11xx_get_sqi_max,
  .suspend = genphy_suspend,
  .resume  = genphy_resume,
  .set_loopback   = genphy_loopback,
  /* Statistics */
  .get_sset_count = tja11xx_get_sset_count,
  .get_strings = tja11xx_get_strings,
  .get_stats = tja11xx_get_stats,
 }
};

module_phy_driver(tja11xx_driver);
```

genphy_driver 的 struct phy_driver 的注册过程如下：

```
phy_init
  phy_driver_register()
      driver_register(&new_driver->mdiodrv.driver)
          bus_add_driver(drv)
              driver_attach(drv)
                  bus_for_each_dev(drv->bus, NULL, drv, __driver_attach)
                      while ((dev = next_device(&i)) && !error)
                          /* 循环到注册的 PHY 设备时 */
                          fn(dev, data) = __driver_attach()
                              /* 匹配设备和驱动 */
                              driver_match_device(drv, dev)
                                  mdio_bus_match(dev, drv)
                                      phy_bus_match(dev, drv)
                                          /* 按 phy_id & phy_id_mask 匹配 */
                                          return (phydrv->phy_id & phydrv->phy_id_mask) == (phyde
                              /* 匹配到设备和驱动，加载驱动 */
                              driver_probe_device(drv, dev)
                                  really_probe(dev, drv)
                                      dev->driver = drv; /* 绑定设备的驱动 */
                                      drv->probe(dev) = phy_probe
```

其中一个关键点是 mdio driver 的 probe 函数是一个通用函数 phy_probe：

```
static int phy_probe(struct device *dev)
{
    struct phy_device *phydev = to_phy_device(dev);        // 获取PHY设备
    struct device_driver *drv = phydev->mdio.dev.driver;
    struct phy_driver *phydrv = to_phy_driver(drv);        // 获取PHY驱动
    int err = 0;
```

```
    phydev->drv = phydrv;    /* 绑定 phy_device 和 phy_driver */

     /* PHY 中断模式最终配置 */
    if (!phy_drv_supports_irq(phydrv) && phy_interrupt_is_valid(phydev))    // 设置中断方式
        phydev->irq = PHY_POLL;

    if (phydrv->flags & PHY_IS_INTERNAL)
        phydev->is_internal = true;

    /* Deassert the reset signal */
    phy_device_reset(phydev, 0);

    if (phydev->drv->probe) {                // 判断驱动有probe方式
        err = phydev->drv->probe(phydev);     /* PHY 驱动的 probe */
        if (err)
            goto out;
    }
    ......
    if (phydrv->features) {
        linkmode_copy(phydev->supported, phydrv->features);
    }
    else if (phydrv->get_features)
        err = phydrv->get_features(phydev);
    else if (phydev->is_c45)
        err = genphy_c45_pma_read_abilities(phydev);
    else
        err = genphy_read_abilities(phydev); //读取状态寄存器来确定 phy 芯片的能力

    if (err)
        goto out;

    if (!linkmode_test_bit(ETHTOOL_LINK_MODE_Autoneg_BIT,
                   phydev->supported))
        phydev->autoneg = 0;

    if (linkmode_test_bit(ETHTOOL_LINK_MODE_1000baseT_Half_BIT,
                  phydev->supported))
        phydev->is_gigabit_capable = 1;
    if (linkmode_test_bit(ETHTOOL_LINK_MODE_1000baseT_Full_BIT,
                  phydev->supported))
        phydev->is_gigabit_capable = 1;

    /* PHY 功能特性配置 */
    of_set_phy_supported(phydev);
    phy_advertise_supported(phydev);
    ......
    of_set_phy_eee_broken(phydev);
    ......
    if (!test_bit(ETHTOOL_LINK_MODE_Pause_BIT, phydev->supported) &&
        !test_bit(ETHTOOL_LINK_MODE_Asym_Pause_BIT, phydev->supported)) {
        linkmode_set_bit(ETHTOOL_LINK_MODE_Pause_BIT,
                 phydev->supported);
        linkmode_set_bit(ETHTOOL_LINK_MODE_Asym_Pause_BIT,
                 phydev->supported);
    }

    /* Set the state to READY by default */
    phydev->state = PHY_READY;    /* 标记 PHY 设备已经就绪 */

out:
    /* Re-assert the reset signal on error */
    if (err)
        phy_device_reset(phydev, 1);

    return err;
}
```

其中通用 phy 驱动会调用函数 genphy_read_abilities 来读取状态寄存器来确定 phy 芯片的能力：

```
genphy_read_abilities()
   `-| {
```

```
    | val = phy_read(phydev, MII_BMSR); // 读取 mdio 0x01 寄存器来确定 phy 的 10/100M 能力
    | linkmode_mod_bit(ETHTOOL_LINK_MODE_Autoneg_BIT, phydev->supported, val & BMSR_ANEGCAPABLE
    | linkmode_mod_bit(ETHTOOL_LINK_MODE_100baseT_Full_BIT, phydev->supported, val & BMSR_100FU
    | linkmode_mod_bit(ETHTOOL_LINK_MODE_100baseT_Half_BIT, phydev->supported, val & BMSR_100HA
    | linkmode_mod_bit(ETHTOOL_LINK_MODE_10baseT_Full_BIT, phydev->supported, val & BMSR_10FULL
    | linkmode_mod_bit(ETHTOOL_LINK_MODE_10baseT_Half_BIT, phydev->supported, val & BMSR_10HALF
    | if (val & BMSR_ESTATEN) {
    |  val = phy_read(phydev, MII_ESTATUS); // 读取 mdio 0x0f 寄存器来确定 phy 的 1000M 能力
    |  linkmode_mod_bit(ETHTOOL_LINK_MODE_1000baseT_Full_BIT, phydev->supported, val & ESTATUS_
    |  linkmode_mod_bit(ETHTOOL_LINK_MODE_1000baseT_Half_BIT, phydev->supported, val & ESTATUS_
    |  linkmode_mod_bit(ETHTOOL_LINK_MODE_1000baseX_Full_BIT, phydev->supported, val & ESTATUS_
    | }
    | }
```

NXP TJA 专有驱动

NXP TJA 驱动的 struct phy_driver 的注册过程如下:

```
#define phy_module_driver(__phy_drivers, __count)           \
static int __init phy_module_init(void)                 \
{                                   \
    return phy_drivers_register(__phy_drivers, __count, THIS_MODULE); \
}                                   \
module_init(phy_module_init);                       \
static void __exit phy_module_exit(void)                \
{                                   \
    phy_drivers_unregister(__phy_drivers, __count);         \
}                                   \
module_exit(phy_module_exit)

#define module_phy_driver(__phy_drivers)                \
    phy_module_driver(__phy_drivers, ARRAY_SIZE(__phy_drivers))
```

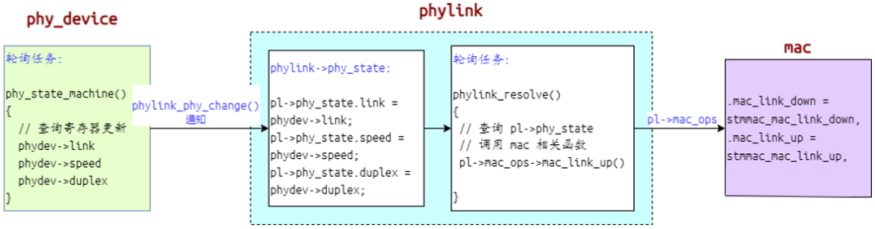
```
int phy_drivers_register(struct phy_driver *new_driver, int n,
             struct module *owner)
{
    int i, ret = 0;

    for (i = 0; i < n; i++) {
        ret = phy_driver_register(new_driver + i, owner);    // 注册数组中所有的phy驱动
        if (ret) {
            while (i-- > 0)
                phy_driver_unregister(new_driver + i);
            break;
        }
    }
    return ret;
}
EXPORT_SYMBOL(phy_drivers_register);
```

根据上面的分析，由于存在 phydev->drv->probe，所以会调用其注册的函数 tja11xx_probe。

网卡 **fec** 和 **Phy** 的协作

在 linux 内核中，以太网 mac 会被注册成 struct net_device，phy 芯片会被注册成 struct phy_device。 phy_device 的状态怎么传递给 net_device，让其在 link 状态变化时做出对应的配置改变，这个任务就落在上述的 struct phylink 中介身上。

下面就以 fec 网口驱动为例，展示一下网卡 fec 和 phy 的协作过程。整个 phy 驱动的主要调用流程如下图所示:



一个 phy 驱动的原理其实是非常简单的，一般流程如下：

1. 用轮询/中断的方式通过 mdio 总线读取 phy 芯片的状态。
2. 在 phy link 状态变化的情况下，正确配置 mac 的状态。（例如：根据 phy 自协商的速率 10/100/1000M 把 mac 配置成对应速率）

phy 芯片状态在 phy 设备注册的时候已经体现，这里详细讲下如何在 phy link 状态变化的情况下，正确配置 mac 的状态。

```
void phy_state_machine(struct work_struct *work)
{

 old_state = phydev->state;

    /* (1) 状态机主体 */
 switch (phydev->state) {
    /* (1.1) 在 PHY_DOWN/PHY_READY 状态下不动作 */
 case PHY_DOWN:
 case PHY_READY:
  break;

    /* (1.2) 在 PHY_UP 状态下，表明网口被 up 起来，需要启动自协商并且查询自协商后的 link 状态
             如果自协商结果是 link up，进入 PHY_RUNNING 状态
             如果自协商结果是 link down，进入 PHY_NOLINK 状态
     */
 case PHY_UP:
  needs_aneg = true;

  break;

    /* (1.3) 在运行的过程中定时轮询 link 状态
             如果 link up，进入 PHY_RUNNING 状态
             如果 link down，进入 PHY_NOLINK 状态
     */
 case PHY_NOLINK:
 case PHY_RUNNING:
  err = phy_check_link_status(phydev);
  break;

 }

    /* (2) 如果需要，启动自协商配置 */
 if (needs_aneg)
  err = phy_start_aneg(phydev);

    /* (3) 如果 phy link 状态有变化，通知给对应网口 netdev */
 if (old_state != phydev->state) {
  phydev_dbg(phydev, "PHY state change %s -> %s\n",
       phy_state_to_str(old_state),
       phy_state_to_str(phydev->state));
  if (phydev->drv && phydev->drv->link_change_notify)
   phydev->drv->link_change_notify(phydev);
 }

    /* (4) 重新启动 work，周期为 1s */
 if (phy_polling_mode(phydev) && phy_is_started(phydev))
  phy_queue_state_machine(phydev, PHY_STATE_TIME);
}
```

自协商配置

具体启动 phy 自协商的代码流程如下：

```
phy_state_machine()
`-| phy_start_aneg()
  `-| phy_config_aneg()
    `-| genphy_config_aneg()
      `-| __genphy_config_aneg()
        `-| genphy_setup_master_slave() // (1) 如果是千兆网口，配置其 master/slave
          `-| {
            |  phy_modify_changed(phydev, MII_CTRL1000,    // 配置 mdio 0x09 寄存器
            |     (CTL1000_ENABLE_MASTER | CTL1000_AS_MASTER | CTL1000_PREFER_MASTER), ctl);
            | }
          | genphy_config_advert() // (2) 设置本端的 advert 能力
          `-| {
            |  linkmode_and(phydev->advertising, phydev->advertising, phydev->supported);
            |  adv = linkmode_adv_to_mii_adv_t(phydev->advertising);
            |  phy_modify_changed(phydev, MII_ADVERTISE,   // 10M/100M 能力配置到 mdio 0x04 寄
            |       ADVERTISE_ALL | ADVERTISE_100BASE4 |
            |       ADVERTISE_PAUSE_CAP | ADVERTISE_PAUSE_ASYM, adv);
            |  if (!(bmsr & BMSR_ESTATEN)) return changed;
            |  adv = linkmode_adv_to_mii_ctrl1000_t(phydev->advertising);
            |  phy_modify_changed(phydev, MII_CTRL1000,    // 1000M 能力配置到 mdio 0x09 寄存
            |       ADVERTISE_1000FULL | ADVERTISE_1000HALF, adv);
            | }
          | genphy_check_and_restart_aneg()
          `-| genphy_restart_aneg() // (3) 启动 phy 自协商
            `-| {
              |  phy_modify(phydev, MII_BMCR, BMCR_ISOLATE,   // 配置 mdio 0x00 寄存器
              |       BMCR_ANENABLE | BMCR_ANRESTART);
              | }
```

link 状态读取

phy link 状态读取的代码流程如下：

```
phy_state_machine()
`-| phy_check_link_status()
  `-| phy_read_status()    // (1) 读取 Link 状态
    `-| genphy_read_status()
      `-| {
        |  genphy_update_link(phydev);   // (1.1) 更新 Link 状态
        |  if (phydev->autoneg == AUTONEG_ENABLE && old_link && phydev->link) return 0;
        |  genphy_read_master_slave(phydev); // (1.2) 如果是千兆网口，更新本端和对端的 master/sla
        |  genphy_read_lpa(phydev);  // (1.3) 更新对端(Link partner) 声明的能力
        |  if (phydev->autoneg == AUTONEG_ENABLE && phydev->autoneg_complete) {
        |    phy_resolve_aneg_linkmode(phydev);  // (1.4.1) 自协商模式，解析 Link 结果
        |  } else if (phydev->autoneg == AUTONEG_DISABLE) {
        |    genphy_read_status_fixed(phydev); // (1.4.2) 固定模式，解析 Link 结果
        |  }
        | }
    | if (phydev->link && phydev->state != PHY_RUNNING) {  // (2) Link 状态 change 事件：变成 Lin
    |   phydev->state = PHY_RUNNING;
    |   phy_link_up(phydev);   // Link up 事件，通知给 phyLink
    | } else if (!phydev->link && phydev->state != PHY_NOLINK) {  // (3) Link 状态 change 事件：变
    |   phydev->state = PHY_NOLINK;
    |   phy_link_down(phydev); // Link down 事件，通知给 phyLink
    | }
```

link 状态通知

phy 的 link 状态变化怎么通知给 netdev，并且让 mac 做出相应的配置改变，这个是通过一个中介 phylink 来实现的。

```
phy_link_up()/phy_link_down()
`-| phydev->phy_link_change(phydev, true/false);
```

```
`-| phylink_phy_change()
  `-| {
    |  pl->phy_state.speed = phydev->speed;     // (1) 把 `phy_device` 状态更新给 `phylink`
    |  pl->phy_state.duplex = phydev->duplex;
    |  pl->phy_state.interface = phydev->interface;
    |  pl->phy_state.link = up;
    |  phylink_run_resolve(pl);     // (2) 通知 `phylink` 的轮询任务启动
    | }
```

收录于合集 #网络 7

‹ 上一篇 · 关于PHY、MAC及其通信接口