清華大學
Tsinghua University

《嵌入式系统》

6-3 块设备驱动程序

6-4 网络设备驱动程序

提纲

- 块设备驱动程序设计概要
- 块设备驱动程序设计方法
- * MMC/SD卡驱动

提纲

- 块设备驱动程序设计概要
- 块设备驱动程序设计方法
- * MMC/SD卡驱动

块设备简介

- 块设备是指与系统间用块的方式传送数据的设备。这些设备通常代表可寻址设备，包括IDE硬盘、SCSI硬盘、CD-ROM等设备。
- 块设备数据存取的单位是块，块的大小通常为512B~32KB不等。块设备每次能传输一个或多个块，支持随机访问，并且采用了缓存技术。
- 块设备驱动主要针对磁盘等慢速设备，由于其支持随机访问，所以文件系统一般采用块设备作为载体。

块设备 vs 字符设备

- 块设备和字符设备类似，也是通过/dev下的文件系统节点访问。

- 块设备和字符设备的驱动主要区别在于管理数据的方式，但是这些对上层应用程序来说是透明的。

IDE

- 集成驱动电子设备（英语：Integrated Drive Electronics，简称：IDE）
- IDE是一种计算机系统接口，主要用于硬盘和CD-ROM，本意为“把控制器与盘体集成在一起的硬盘”。以前PC主机使用的硬盘，大多数都是IDE兼容的，只需用一根电缆将它们与主板或接口卡连起来就可以了，而当前主要接口为SATA接口。

Motherboard Sockets

Connector

IDE

- IDE接口实质上是存储设备与计算机连接的标准方式。而IDE并不是该接口标准的真正技术名称。它原来的名称是AT附加装置(Advanced Technology Attachment, ATA)。一般说来，ATA是一个控制器技术，而IDE是一个匹配它的磁盘驱动器技术，但是两个术语经常可以互用。
- SATA（Serial ATA）于2002年推出后，原有的ATA改名为PATA（并行高技术配置，Parallel ATA）

SATA

- 串行ATA（英语：Serial ATA，全称：Serial Advanced Technology Attachment）
- 一种计算机系统总线，负责主板和大容量存储设备（如硬盘及光盘驱动器）之间的数据传输，主要用于个人电脑
- SATA用于取代旧式PATA接口的旧式硬盘，因采用串行方式传输数据而得名

An mSATA SSD on top of a 2.5-inch SATA drive

SATA

- 在数据传输上这一方面，SATA的速度比以往更加快捷，并支持热插拔，使电脑运作时可以插上或拔除硬件。
- SATA总线在物理层使用**差分信号编码**，具备比以往更强的抗干扰和纠错能力，能对传输指令（不仅是数据）进行检查，如果发现错误会自动矫正，提高数据传输的可靠性。
- SATA和PATA最明显的分别，是使用较细的排线，有利机箱内部的空气流通，某程度上增加整个平台的稳定性。

SCSI

- 小型计算机系统接口（英语: Small Computer System Interface，简称：SCSI）
- 一种用于计算机及其周边设备之间（硬盘、软驱、光驱、打印机、扫描仪等）系统级接口的独立处理器标准。
- SCSI标准定义命令、通信协议以及实体的电气特性，最大部分的应用是在存储设备上（例如硬盘、磁带机）。

SCSI Connector

CD-ROM

- 只读光盘（Compact Disc Read-Only Memory，缩写：CD-ROM）
- 一种在电脑上使用的光盘。这种光盘只能写入数据一次，信息将永久保存在光盘上，使用时通过光盘驱动器读出信息。
- CD-ROM盘片在外观上与音乐CD完全相同，数据存取的方式也十分类似，区别仅在于它们存储数据的标准。在CD上存储数据有多种格式，它们被收集成**彩虹书***。其中包括最初音乐CD的红皮书标准，此外还有白皮书和黄皮书。

** 记录所有标准CD格式的一个规格书集合。*

CD-ROM

- CD-ROM光驱可以读取CD-ROM光盘，这种设备在个人电脑上已经普及。CD-ROM光驱可以通过IDE（ATA）、SCSI、SATA、Firewire和USB或者专门设备连接至电脑。事实上，结合适当的软件，所有的现代CD-ROM光驱都能够播放音乐CD、VCD和其它数据标准的CD。

A CD-ROM in the tray of a partially open DVD-ROM drive.

MMC

- 多媒体存储卡（英语：Multimedia Card，简称：MMC）
- 一种快闪记忆卡标准。在1997年由西门子及闪迪共同开发，技术基于东芝的NAND快闪记忆技术，较早期基于Intel NOR快闪记忆技术的存储卡（例如CF卡）更细小。
- MMC卡原本使用1bit串联接口，但较新的标准则容许同时发送4 bit或8 bits的数据。近年MMC卡技术已差不多完全被SD卡所代替；但由于MMC卡仍可被兼容SD卡的设备所读取，因此仍有其作用。
- MMC卡大小与一张邮票差不多，

  约24mm x 32mm x 1.5mm。

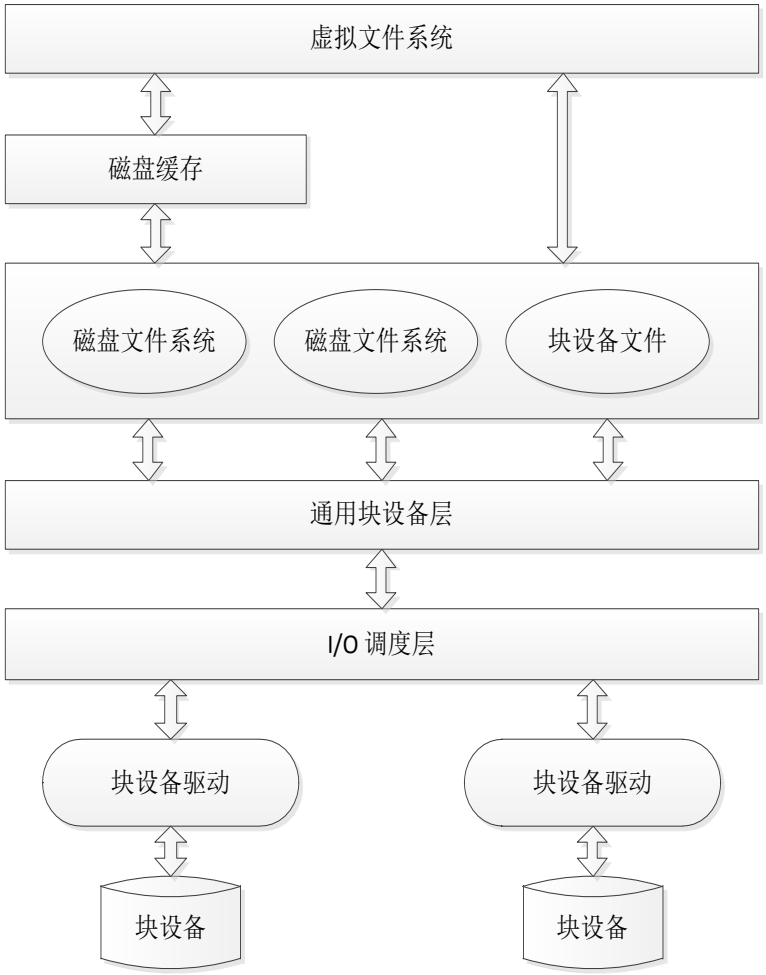
SD

- 安全数位卡（英语：Secure Digital Memory Card，简称： Secure Digital，SD）
- 一种存储卡，被广泛地于便携式设备上使用，例如数字相机、个人数码助理和多媒体播放器等。SD卡的技术建立在MultiMedia卡格式的基础上。
- SD卡有比较高的数据发送速度，而且不断更新标准。大部分SD卡的侧面设有写保护控制，以避免一些数据意外地写入，而少部分的SD卡甚至支持数字版权管理的技术。
- SD卡的大小为32mm × 24mm × 2.1mm

块设备驱动程序设计概要（1）

❑块设备的操作涉及内核中的多个组成部分。例如系统调用read()读取磁盘上的文件，内核响应的步骤：

1. 系统调用read()会触发相应的虚拟文件系统函数，**VFS**确定请求的数据是否已经在缓存中；若不在，确定如何执行读操作。
2. 假设内核必须从块设备上读取数据，内核就必须确定数据在某个物理设备上的位置(**地址**)。
3. 内核通过通用块设备层在块设备上执行读操作，**启动I/O**操作，传输请求的数据。
4. 在通用块设备层之下是**I/O调度**层（I/O Scheduler Layer），根据内核的调度策略，对等待的I/O请求排序。
5. 块设备驱动（Block Device Driver）通过向磁盘控制器发送相应的命令，执行真正的**数据传输**。



块设备驱动程序设计概要（2）

- 块设备读写请求
  - 对块设备的读写是通过**请求**实现的。
  - 对于机械硬盘这类设备来说，根据其机械特性，合理地组织请求的顺序（如电梯算法），尽量顺序地进行访问，可得到更好的性能。（为什么？）
- 块设备的I/O请求都有对应的缓冲区，并可以使用请求队列对请求进行管理

块设备驱动程序设计概要（3）

- 在 Linux中每一个块设备都有一个I/O请求队列，每个请求队列都有调度器的插口，调度器可以实现对请求队列里请求的合理组织，如合并临近请求，调整请求完成顺序等：
  - No-op I/O scheduler 实现了一个简单FIFO队列
  - Anticipatory I/O scheduler 当前内核中默认的I/O调度器，但比较庞大与复杂，在数据吞吐量非常大的数据库系统中它会变的比较缓慢
  - Deadline I/O scheduler 改善了AS的缺点
  - CFQ I/O scheduler 系统内所有任务分配相同的带宽，提供一个公平的工作环境，比较适合桌面环境

提纲

- 块设备驱动程序设计概要
- 块设备驱动程序设计方法
- * MMC/SD卡驱动

块设备的注册与注销

- 和字符设备类似，块设备驱动程序第一步通常也是向内核注册自己，实现这个任务的函数是:

```
int register_blkdev(unsigned int major, const char *name);
```

- major是块设备的主设备号，name为设备名称若 major为0，则内核将为其分派一个新的主设备号给设备，并返回此设备号给调用者。该函数不一定成功，出错则返回负值。
- 它主要完成两个工作:一是在需要的时候分配主设备号，二是在/proc/devices中创建一个entry。

块设备的注册与注销

- 与 register_blkdev对应的是注销函数 unregister_blkdev，其定义如下。

```
void unregister_blkdev(unsigned int major, const char *name);
```

块设备初始化与卸载

- 块设备驱动程序编写的第一步是编写初始化函数，在初始化过程中要完成如下几项工作
  - 注册块设备及块设备驱动程序
  - 分配、初始化、绑定请求队列（如果使用请求队列的话）
  - 分配、初始化gendisk，为相应的成员赋值并添加gendisk。
  - 其它初始化工作，如申请缓存区，设置硬件规格（不同设备，有不同处理）
- 块设备的注销动作刚好与注册相反，工作如下
  - 删除请求队列
  - 撤销对gendisk的引用并删除gendisk
  - 释放缓冲区，撤销对块设备的应用，注销块设备驱动

块设备操作

- 与字符设备类似，块设备也有一个operations结构体用于实现设备操作接口

```
struct block_device_operations {
    /* 打开与释放*/
    int (*open) (struct block_device *, fmode_t);
    int (*release) (struct gendisk *, fmode_t);
    /* I/O操作 */
    int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
    int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
    int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
    int (*direct_access) (struct block_device *, sector_t,void **, unsigned long *);
    /*介质改变*/
    int (*media_changed) (struct gendisk *);
    unsigned long long (*set_capacity) (struct gendisk *,unsigned long long);
    /* 使介质有效 */
    int (*revalidate_disk) (struct gendisk *);
    /*获取驱动器信息 */
    int (*getgeo)(struct block_device *, struct hd_geometry *);
    struct module *owner;
};
```

块设备请求处理

- 块设备不像字符设备操作，它并没有包括read和write。对块设备的读写是通过**请求函数**完成的。请求处理分为两种情况：
  - 使用请求队列: 使用请求队列对于提高机械磁盘的读写性能具有重要意义，I/O调度程序按照一定算法（如电梯算法）通过优化组织请求顺序，帮助系统获得较好的性能。
  - 不使用请求队列: 对于一些本身就支持随机寻址的设备，如SD卡、RAM盘、软件RAID组件、虚拟磁盘等设备，请求队列对其没有意义。针对这些设备的特点，块设备层提供了“无队列”的操作模式

块设备请求处理

❑请求函数

void request (request_queue_t * queue);

当内核需要对设备进行读写或者其他操作时调用该函数。

主要完成的工作：根据request、bio等结构提供的信息，完成具体的I/O传输并通知设备层。

值得注意的是：请求函数返回时，不一定完成请求的所有操作，甚至什么都没做就返回，而是将该请求添加到请求队列。

相关重要数据结构与函数

- gendisk结构：表示一个独立磁盘设备/一个分区

```
struct gendisk {
/* 只有major, first_minor 和minors是输入变量，不能直接使用，应当使用disk_devt() 和 disk_max_parts()
. */
    int major;          /* 主设备号 */
    int first_minor;
    int minors;      /* 次设备号的最大值，若为1 则该盘不能被分区*/
    char disk_name[DISK_NAME_LEN];                /*主驱动名称 */
    char *(*nodename)(struct gendisk *gd);
                    /* 磁盘分区的指针数组，使用partno进行索引. */
    struct disk_part_tbl *part_tbl;                    /*分区表*/
    struct hd_struct part0;
    struct block_device_operations *fops;
    struct request_queue *queue;                      /*请求队列*/

    void *private_data;
    int flags;
    struct device *driverfs_dev;
    struct kobject *slave_dir;
    struct timer_rand_state *random;
    atomic_t sync_io;       /* RAID */
    struct work_struct async_notify;
    int node_id;
};
```

相关重要数据结构与函数

- 其中的主要信息有: major、first_minor、minors分别表示磁盘的主设备号、次设备号，同属于同一块磁盘的分区共享主设备号，一个驱动器至少使用一个次设备号，如果驱动器可以被分区，则必须为每个可能的分区都分配一个次设备号。
- Linux提供了一组函数接口来操作gendisk结构体，如下:

```
/*分配gendisk*/ struct gendisk *alloc_disk(int minors);
/*增加gendisk*/ void add_disk(struct gendisk *disk);
/*释放gendisk*/ void del_gendisk(struct gendisk *gp);
/*引用计数*/    get_disk和put_disk
/*设置和查看磁盘容量*/ void set_capacity(struct gendisk *disk, sector_t size);
                      sector_t get_capacity(struct gendisk *disk);
```

相关重要数据结构与函数

- request结构：
  - 包含了很多成员，注释的为常用的一小部分

```
struct request {
    struct list_head queuelist;     /* 将请求连接到请求队列 */
    struct call_single_data csd;
    int cpu;
    struct request_queue *q;
    unsigned int cmd_flags;
    enum rq_cmd_type_bits cmd_type;
    unsigned long atomic_flags;
    sector_t sector;                /* 下一个传输的扇区 */
    sector_t hard_sector;           /* 下一个完成的扇区 */
    ~unsigned long nr_sectors;      /* 未提交的扇区数 */
    unsigned long hard_nr_sectors;  /* 未完成的扇区数*/
    unsigned int current_nr_sectors;/* 当前段中未提交的扇区数 */
    unsigned int hard_cur_sectors;  /* 当期段中未完成的扇区数 */
    struct bio *bio;
    struct bio *biotail;
    struct hlist_node hash;         /* 混合hash */
    void *elevator_private;
    void *elevator_private2;
    struct gendisk *rq_disk;
    unsigned long start_time;
    unsigned short nr_phys_segments;
    unsigned short ioprio;
    void *special;
    char *buffer;
    int tag;
    int errors;
    int ref_count;
    /* ... */
};
```

相关重要数据结构与函数

- request_queue队列
  - 每一个块设备都有一个请求队列，请求队列组织和跟踪该设备的所有I/O请求，并提供插入接口给I/O调度器使用。同时请求队列保存了该设备所支持的请求的类型信息，包括请求队列的最大尺寸、硬件扇区大小、同一请求中所能包含的独立段的数目、对齐要求等。

```
struct request_queue {
    struct list_head queue_head;
    struct request *last_merge;
    struct elevator_queue   *elevator;
    /* ... */
    unsigned long queue_flags;
    /* 自旋锁，不能直接应用，应使用->queue_lock访问 */
    spinlock_t __queue_lock;
    spinlock_t *queue_lock;
    struct kobject kobj;                /* 队列设置 */
    unsigned long nr_requests;         /* 最大请求数 */
    unsigned int nr_congestion_on;
    /* ... */
}
```

相关重要数据结构与函数

❑request_queue队列的操作函数

```
/* 请求队列的初始化和清除 */
request_queue *blk_init_queue_node(request_fn_proc *rfn,spinlock_t *lock, int node_id);
void blk_cleanup_queue(struct request_queue *q);
/* 提取和删除请求 */
struct request *elv_next_request(struct request_queue *q);
void blkdev_dequeue_request(struct request *req);
void elv_requeue_request(struct request_queue *q, struct request *rq);
/* 队列的参数设置 */
void blk_stop_queue(struct request_queue *q);
void blk_start_queue(struct request_queue *q);
/* 内核通告 */
void blk_queue_segment_boundary(struct request_queue *q, unsigned long mask);
```

相关重要数据结构与函数

- bio 结构
  - request实质上是一个bio结构的链表实现。bio是底层对部分块设备的I/O请求描述，其包含了驱动程序执行请求所需的全部信息。
  - 通常一个I/O请求对应一个bio。I/O调度器可将关联的bio合并成一个请求。

相关重要数据结构与函数

❑bio 结构

```
struct bio {
    sector_t bi_sector;
    struct bio *bi_next;            /*请求队列指针 */
    struct block_device *bi_bdev;
    unsigned long bi_flags;         /* 状态，命令等*/
    /*最后一位为读写标志位，* 前面的为优先级*/
    unsigned long bi_rw;
    unsigned short bi_vcnt;             /* bio_vec数*/
    unsigned short bi_idx;  /* 当前bio_vec中的索引 */
    /* 该bio的分段信息（设置了物理地址聚合有效）*/
    unsigned int bi_phys_segments;
    unsigned int bi_size;
    unsigned int bi_seg_front_size;
    unsigned int bi_seg_back_size;
    unsigned int bi_max_vecs;     /* 最大bvl_vecs数*/
    unsigned int bi_comp_cpu;
    atomic_t bi_cnt;                    /* 针脚数 */
    struct bio_vec *bi_io_vec;      /*真正的vec列表 */
    /* ... */
};
```

提纲

- 块设备驱动程序设计概要
- 块设备驱动程序设计方法
- * MMC/SD卡驱动

清華大學
Tsinghua University

MMC卡驱动

- MMC/SD卡驱动结构
- MMC卡块设备驱动分析
- HSMCI接口驱动设计分析

MMC/SD卡驱动结构

- MMC/SD驱动分为4层，如下所示

文件系统
块设备驱动(driver/mmc/card)
MMC/SD核心(driver/mmc/core)
MMC/SD接口(driver/mmc/host)

- 块设备驱动层:该层实现块设备驱动，为上层提供块设备操作的功能
- MMC/SD核心:编写MMC/SD驱动必须要遵循MMC/SD规范和协议，所有的操作必须按照协议规定进行，该层主要完成不同协议和规范的实现。
- MMC/SD接口:该层主要实现host接口的驱动，并为上层提供操作接口。

MMC卡块设备驱动分析

- 注册与注销：
  - 注册主要负责两个工作，一个是注册MMC块设备，另外一个是注册MMC驱动。
  - 使用函数mmc_blk_init。
  - MMC卡设备在注册驱动模块时并没有分配初始化请求队列以及gendisk等驱动操作所必需的数据接口，而只是注册了设备和设备驱动。
  - 因为MMC卡作为一个支持热插拔的设备，在加载驱动模块时真实的物理设备不一定已连接上，所以这些操作应放到设备的加载初始化过程中（mmc_blk_probe）完成。

MMC卡块设备驱动分析

- 注册与注销：
  - MMC设备驱动的注销工作刚好与注册相反。
  - 使用函数mmc_blk_exit。

MMC卡块设备驱动分析

- 设备加载与卸载
  - 当MMC/SD卡插入到主机，热插拔系统检测到后，系统就会调用mmc_blk_probep函数初始化设备
  - 该函数实现了对设备的初始化，包括设置设备块大小，分配和初始化设备的私有数据，添加gendisk等操作。
  - 其中最重要的是分配和初始化设备的私有数据，MMC卡驱动的运行都围绕设备的私有数据。

MMC卡块设备驱动分析

- 设备加载与卸载
  - 当用户主动卸载设备，如鼠标右键单击移除。当MMC卡被拔出时，系统会调用mmc_blk_remove函数删除相关数据结构和引用，并设置引用计数。
  - 该函数调用了mmc_cleanup_queue清除请求队列，在清除请求队列的同时终止用于处理请求的内核线程。

MMC卡块设备驱动分析

- 设备的打开与释放
  - 与字符设备类似，在用户空间程序执行fopen函数时，实际调用的是MMC卡的mmc_blk_open函数，该函数主要的功能是申请设备私有数据并检查读写方式是否正确、是否更换了物理设备。
  - 与mmc_blk_open对应的是mmc_blk_release，在释放设备时调用，用于清除设备私有数据。

MMC卡块设备驱动分析

- MMC驱动的请求处理函数
  - MMC驱动的请求处理函数主要包括三个函数，它们分别是：
  - mmc_prep_request用于请求被执行前检查请求类型是否正确。
  - mmc_requset在新的请求到来时用于唤醒执行具体请求处理任务的内核线程，当主机空闲时调用该函数查找一个等待的请求，并同时唤醒内核线程进行相应的处理。
  - mmc_blk_issue_rq是具体执行请求操作的函数。

清華大學
Tsinghua University

《嵌入式系统》

6-4 网络设备驱动程序

提纲

- 以太网基础知识
- 网络设备驱动程序设计概要
- 网络设备驱动程序设计方法

提纲

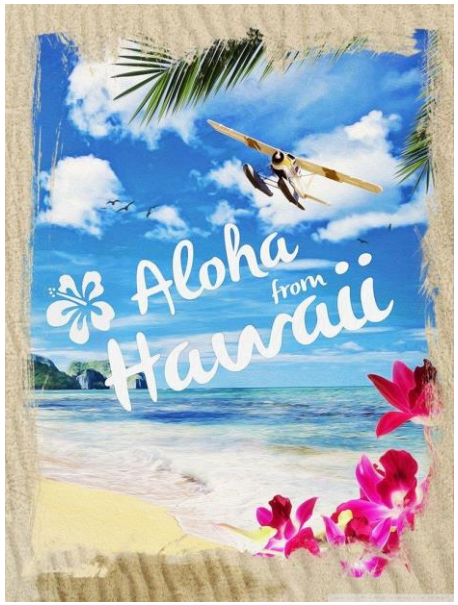
- 以太网基础知识
- 网络设备驱动程序设计概要
- 网络设备驱动程序设计方法

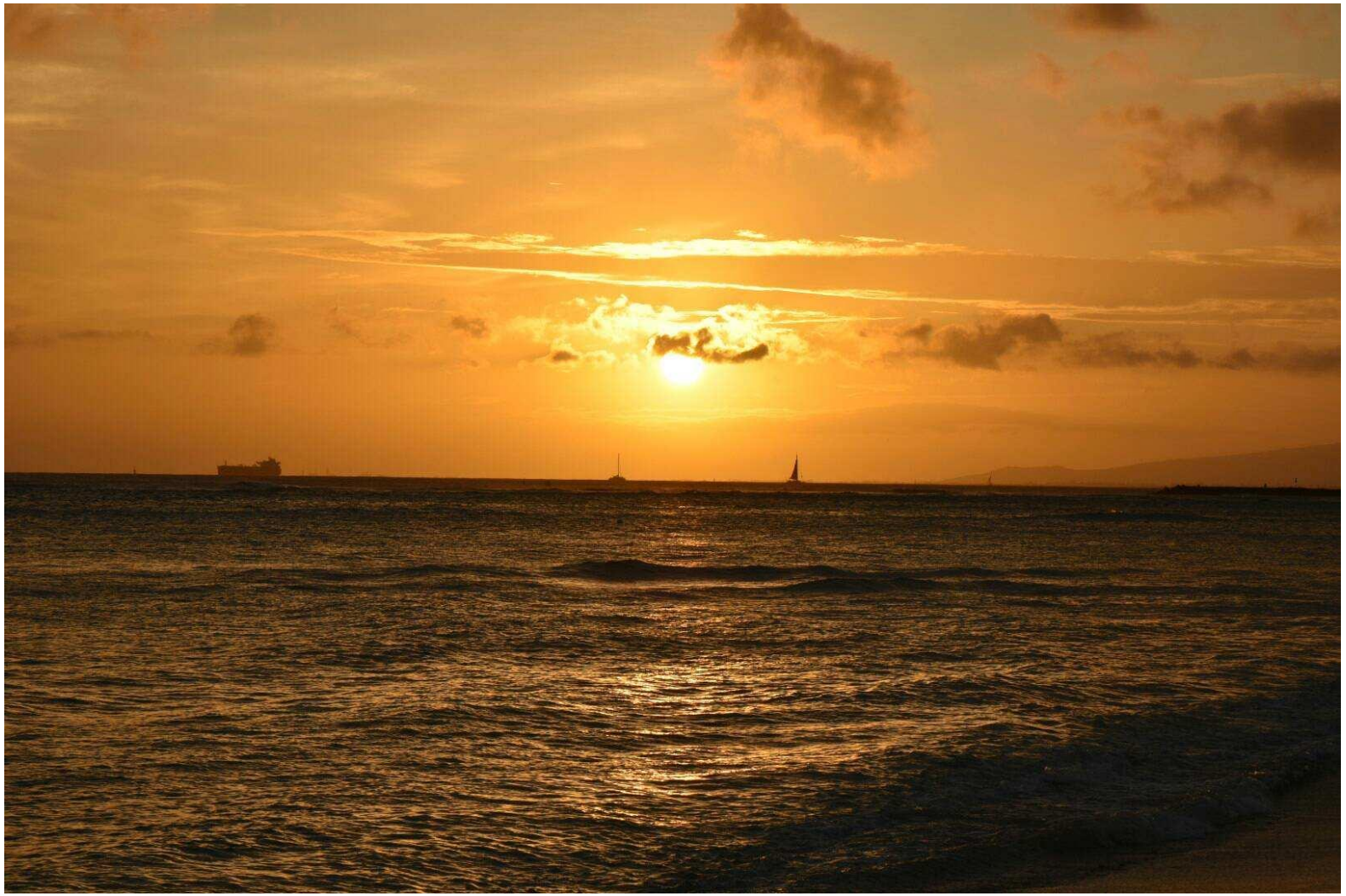
以太网基础知识

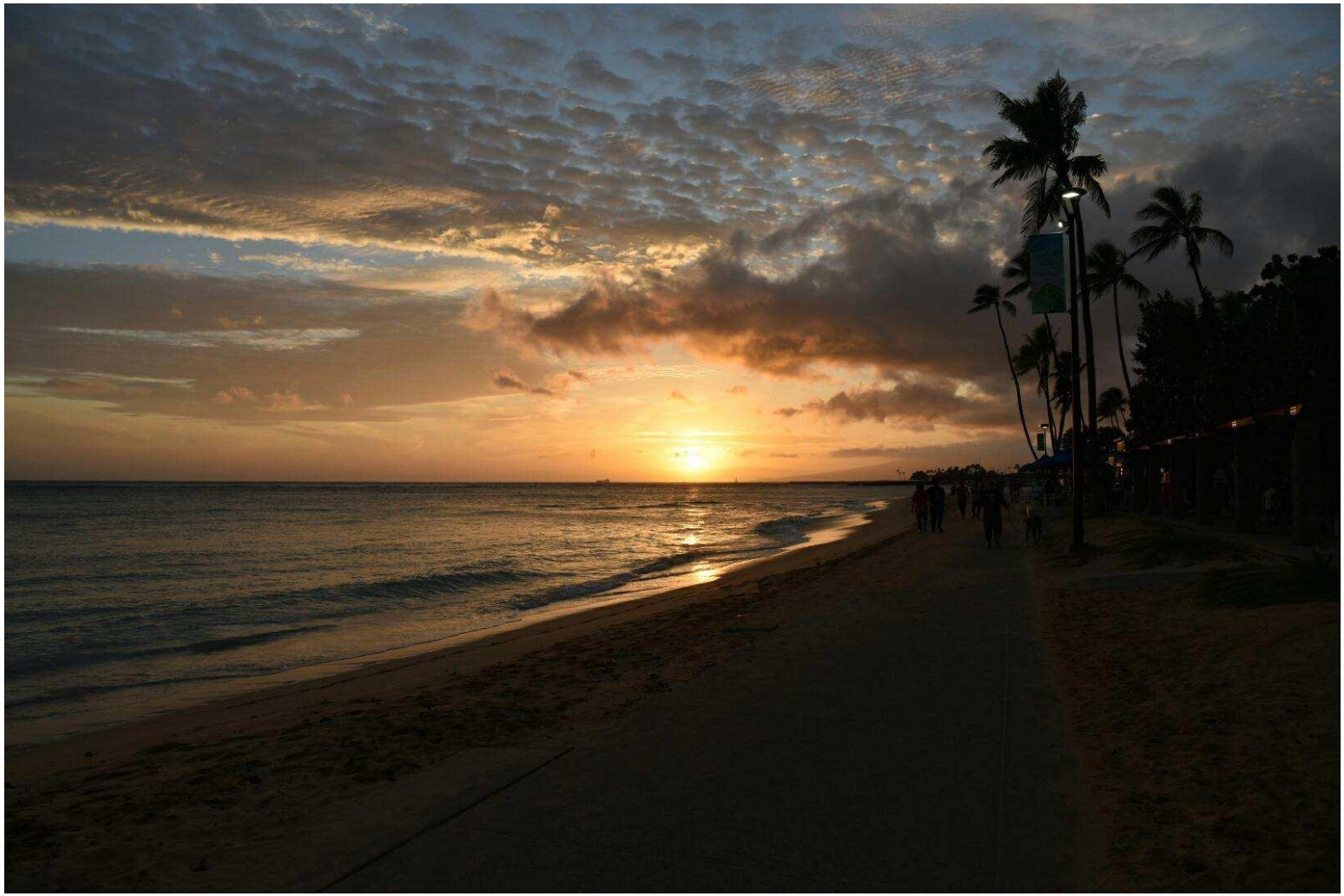
- 以太网（Ethernet）是一种广泛使用的局域网互联技术，它最初是由 Xerox公司研发，并在1980年由数据设备公司DEC（ Digial Equipment Corporation）、 Intel公司和 Xerox公司共同努力使之规范成形。后来它作为802.3标准被电气与电子工程师协会（IEEE）所采纳。IEEE制定的IEEE802.3标准给出了以太网的技术标准。它规定了包括物理层的连线、电信号和介质访问层协议的内容。以太网是当前应用最广泛的局域网技术。它很大程度上取代了其他局域网标准，如令牌环网、FDDI和 ARCNET。

CSMA协议

- 以太网实现了局域网内多用户共用一条信道的功能。为实现该功能以太网采用了载波监听多点接入的通信机制。
- 它是一种抢占型的共享介质的访问控制协议，最早起源于夏威夷大学开发的ALOHA协议，并在ALOHA的基础上通过不断改进而成。比之ALOHA协议，它有更高的介质利用率。







CSMA/CD协议

CSMA/CD为三个名字的组合，分别如下。

- 载波侦听（Carrier Sense）
  - 指任何连接到介质的设备在欲发送帧前，必须对介质进行侦听，当确认其空闲时，才可以发送。
- 多点接入（Multiple Access）
  - 指多个设备可以同时访问介质，一个设备发送的帧也可以被多个设备接收。
- 冲突检测（Collision Detect）
  - 在发送时检测冲突，并采取适当措施进行补救。

CSMA/CD协议

CSMA/CD协议的侦听发送策略有以下三种。

- 非坚持CSMA（non-persistent CSMA）
  - 当要发送帧的设备侦听到线路忙或发生冲突时，会随机等待一段时间再进行侦听；若发现不忙则立即发送。此策略可以减少冲突但会导致信道利用率降低以及较长的延迟。
- 1-坚持CSMA（1-persistent CSMA）
  - 当要发送帧的设备侦听到线路忙或发生冲突时，会持续侦听；若发现不忙则立即发送。当传播延迟较长或多个设备同时发送帧的可能性较大时，此策略会导致较多的冲突以及性能降低。

CSMA/CD协议

- p-坚持CSMA（p-persistent CSMA）
  - 当要发送帧的设备侦听到线路忙或发生冲突时，会持续侦听；若发现不忙，则根据一个事先指定的概率p来决定是发送帧还是继续侦听（以p的概率发送，1-p的概率继续侦听）。此种策略可以达到一定的平衡，但对于参数p的配置会有比较复杂的考量。

CSMA/CD协议

CSMA/CD的控制规程的核心问题:解决在公共通道上以广播方式传送数据中可能出现的问题。它主要包含以下4个处理内容。

1. 侦听

- 检测当前线路上有无其他节点在传送数据，如果线路忙则根据退避算法等待一段时间。若仍然忙，则继续延迟等待直到可以发送为止。每次延时的时间不一致，由退避算法确定延时值。

CSMA/CD协议

2. 数据发送

- 当满足条件允许发送数据时，向共享信道发送数据。数据长度最少要64B，这样便于检测冲突。

3. 冲突检测

- 数据发送后也可能发生数据碰撞。因此，设备在发送帧的同时要对信道进行侦听，以确定是否发生冲突。

CSMA/CD协议

4. 冲突处理

- 当检测到冲突后应当进行如下操作步骤。
- 发送特殊阻塞信息并立即停止发送数据。特殊阻塞信息是连续几个字节的全1信号，这样做的目的在于强化冲突，以使得其他设备能尽快检测到冲突发生。
- 在固定时间（一开始是1 contention period time）内等待随机的时间点，再次发送
- 若依旧碰撞，则采用截断二进制指数退避算法进行发送。即停止前一次“固定时间”的两倍时间内随机再发送（最多10次）；10次后，则停止前一次“固定时间”内随机再发送。尝试16次之后仍然失败则放弃传送并通知上层应用程序。

CSMA/CA协议

与CSMA/CD中相比，CSMA含义相同，CA表示：

- 冲突避免（Collision Avoidance）
  - 主动避免冲突而非被动侦测的方式来解决冲突问题。

对于无线局域网来说，直接应用CSMA/CD协议可能存在诸如下面的问题：

- CSMA/CD不断检测信道对无线设备来说开销过大；
- 即使在发送时信道空闲，但由于无线电波能够向所有的方向传播，且其传播距离受限，所以接收端仍可能发生冲突（why）；
- 无线信号强度受环境（包括障碍物）影响，动态范围非常大，使得发送端有时很难检测冲突的发生；

因此制定了更加适用于无线网络的CSMA/CA协议。

Hidden/Exposed Terminal & RTS/CTS

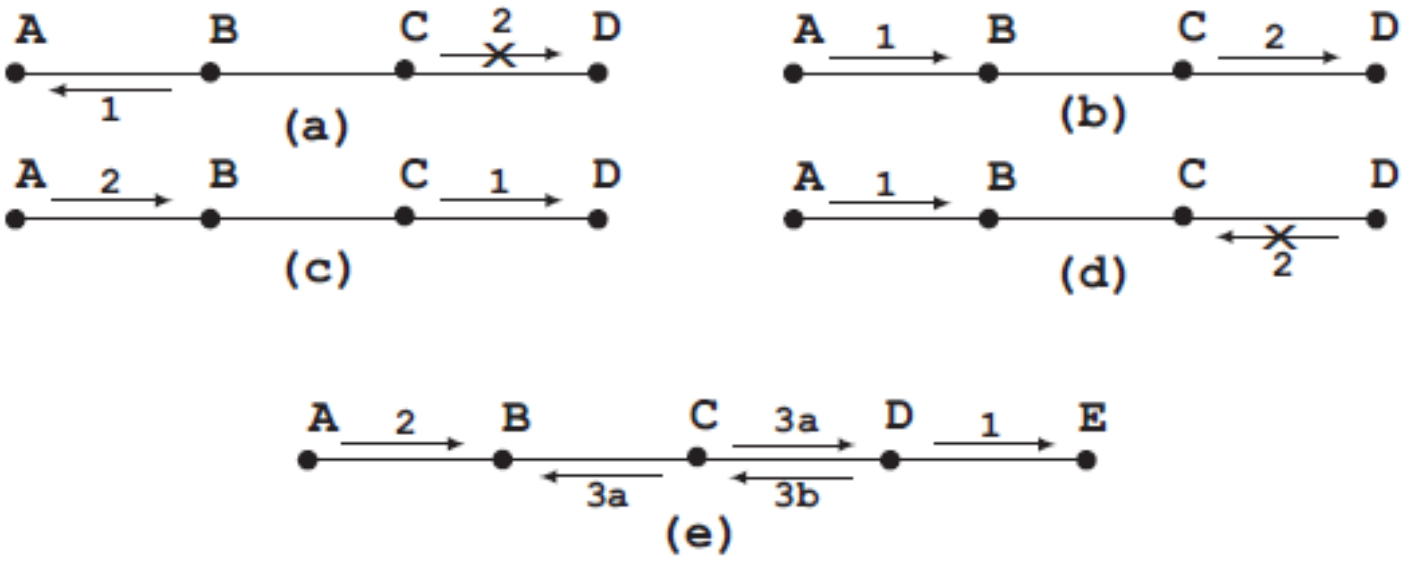
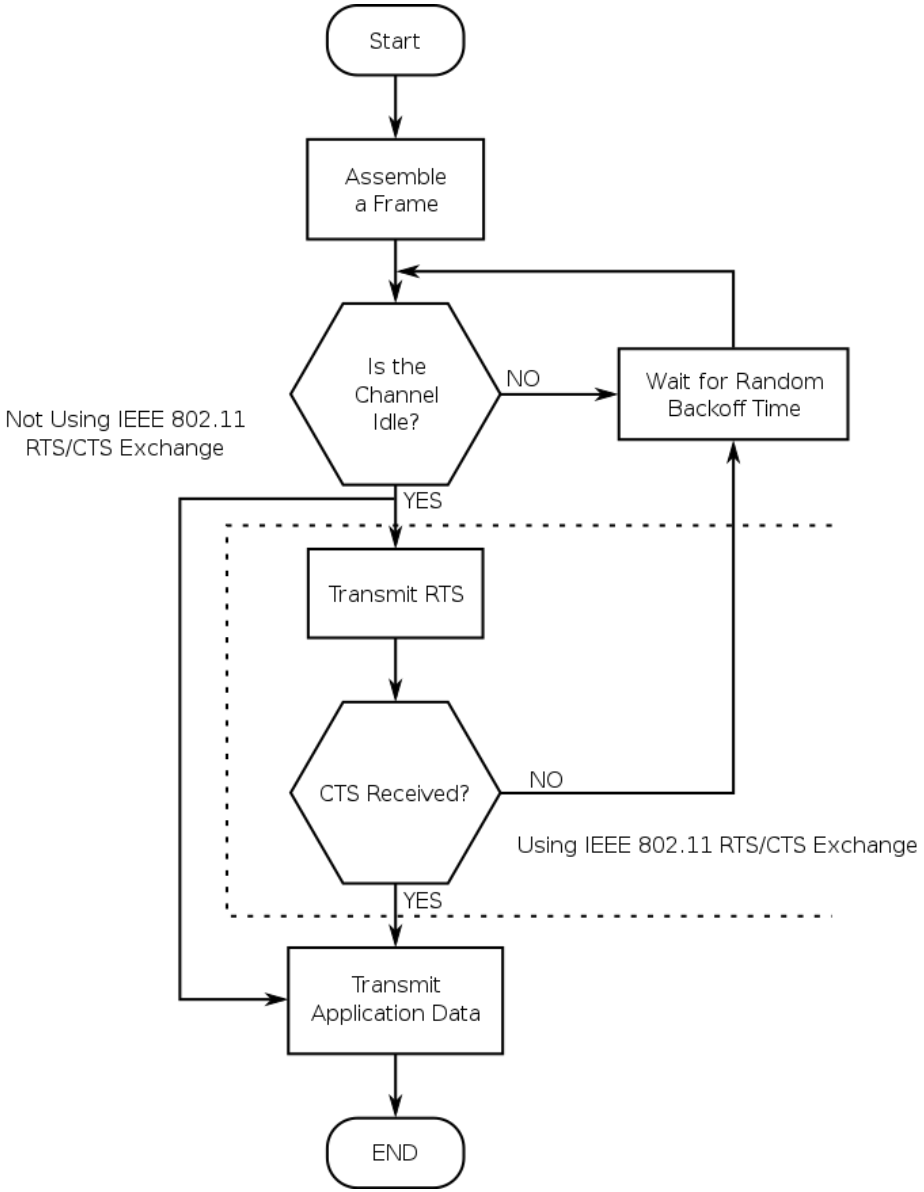


Fig. 1. Shortcomings of CSMA and RTS-CTS (the numbering of the arrows indicates the order of the transmissions).

CSMA/CA协议

- 简要的工作流程如右图。
- RTS/CTS只是理论上(why)能够解决hidden terminal problem。



CSMA/CA与CSMA/CD的区别

CSMA/CD和CSMA/CA的主要差别表现在：

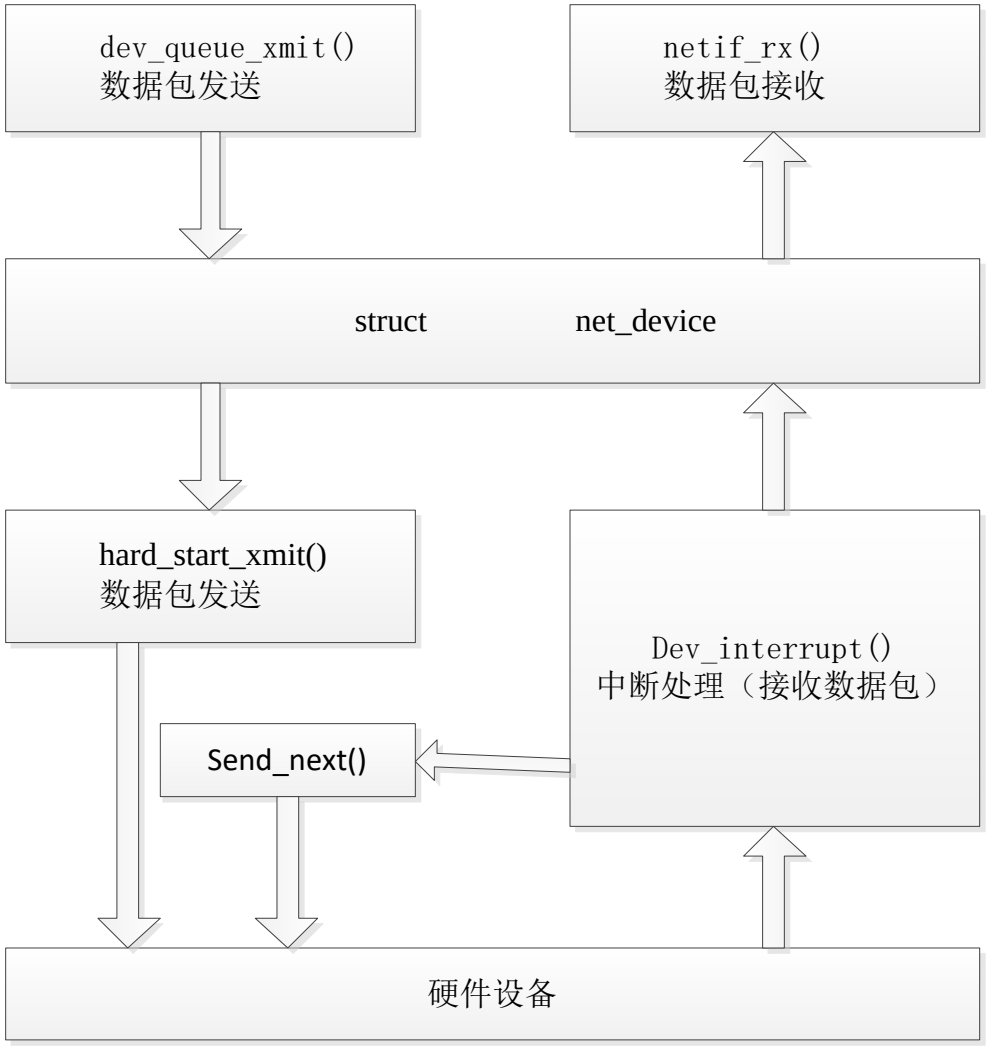
- 两者适用的传输介质不同：CSMA/CD用于总线式以太网，而CSMA/CA通常用于无线局域网802.11a/b/g/n等。
- 检测方式不同：CSMA/CD通过电缆中电压的变化来检测，当数据发生碰撞时，电缆中的电压就会随着发生变化；CSMA/CA采用能量检测（ED）、载波检测（CS）和能量载波混合检测三种检测信道空闲的方式。

提纲

- 以太网基础知识
- 网络设备驱动程序设计概要
- 网络设备驱动程序设计方法

网卡驱动程序框架

- 没有类似字符/块设备的设备文件，访问网络设备使用socket。
- 为上层协议（网络层）提供发送和接收的两个系统接口：
  - dev_queue_xmit()
  - Netif_rx()
- net_device描述具体设备的属性和操作函数
- 底层发送通过hard_start_xmit()完成
- 底层接收通过中断/轮询完成



提纲

- 以太网基础知识
- 网络设备驱动程序设计概要
- 网络设备驱动程序设计方法

基本数据结构

- net_device数据结构
  - 其本身非常庞大，我们只需了解其中一小部分

```
/*以下为全局信息 */
char name[IFNAMSIZ];                    // 网络设备的名称
int (*init)(struct net_device *dev);    // 设备初始化函数指针，通常为 NULL
/* 以下为硬件信息 */
unsigned long  mem_end;                 // 共享内存起始地址
unsigned long  mem_start;               // 共享内存终止地址
unsigned long  base_addr;               // 设备I/O基址
unsigned int   irq;                     // 设备中断号
unsigned char  if_port;                 // 指定多端口设备使用哪个端口
unsigned char  dma;                     // 分配给设备的dma通道
struct net_device_stats stats;          // 设备状态
/* 以下为接口信息 */
unsigned       mtu;                     // 最大传输单元大小
unsigned short type;                    // 接口类型
unsigned short hard_header_len;         // 硬件头长度
unsigned char  dev_addr[MAX_ADDR_LEN];  // MAC 地址
unsigned char  broadcast[MAX_ADDR_LEN]; // 广播地址
```

基本数据结构

❑接口函数

```
/* 接口函数 */
// 打开接口，注册所有的系统资源，并进行相应设置
int     (*open)(struct net_device *dev);
// 与open相反，关闭接口，以及注销资源
int     (*stop)(struct net_device *dev);
// 启动数据包发送，skb为上层需要传送的数据包
int     (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
// 设置设备的组播列表
void    (*set_multicast_list)(struct net_device *dev);
// 设置设备的MAC地址
int     (*set_mac_address)(struct net_device *dev, void *addr);
// 执行接口特有的io控制指令
int     (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
// 改变接口的配置
int     (*set_config)(struct net_device *dev,struct ifmap *map);
// 在接口MTU改变时，进行相应的设置
int     (*change_mtu)(struct net_device *dev, int new_mtu);
// 解决数据包超时问题，并重传数据
void    (*tx_timeout) (struct net_device *dev);
// 返回设备状态信息，保存到 net_device_stats结构体中
struct net_device_stats* (*get_stats)(struct net_device *dev);
// 在禁止中断的情况下，要求驱动程序检测接口下的事件
void    (*poll_controller)(struct net_device *dev);
```

基本数据结构

❑sk_buffer 数据结构

```
/* 网络协议头 */
sk_buff_data_t transport_header; // 传输层协议头
sk_buff_data_t network_header;   // 网络层协议头
sk_buff_data_t mac_header;       // 链路层协议头
/* 缓冲区指针 */
sk_buff_data_t tail;             // 当前层有效数据末尾
sk_buff_data_t end;              // 内存中缓冲区末尾（tail最大值）
unsigned char  *head, *data;     // 缓冲区起始地址，有效数据起始地址
/* 操作函数 */
// 分配一个缓存区，并初始化数据
struct sk_buff *alloc_skb(unsigned int size,gfp_t priority);
// 释放缓冲区（内核内部使用）
void kfree_skb(struct sk_buff *skb);
// 向缓冲区末尾添加数据，并更新相关指针
unsigned char *skb_put(struct sk_buff *skb, unsigned int len);
// 向缓冲区头部添加数据，并更新相关指针
unsigned char *skb_push(struct sk_buff *skb, unsigned int len);
// 从数据包中删除数据
unsigned char *skb_pull(struct sk_buff *skb, unsigned int len);
```

网络设备初始化

- 主要对net_device结构体进行初始化
- 由net_device的init函数指针指向的函数完成，当加载网络驱动模块时该函数就会被调用，初始化包括以下几个方面的任务：
  - 检测网络设备的硬件特征，检查物理设备是否存在。
  - 检测到设备存在，则进行资源配置。
  - 对net_device成员变量进行赋值。

打开和关闭接口

- 打开接口
  - 在数据包放送前，必须打开接口并初始化接口
  - 打开接口的工作由net_device的open函数指针指向的函数完成，该函数负责的工作包括请求系统资源，如申请I/O区域、DMA通道及中断等资源
  - 告知接口开始工作，调用netif_start_queue激活设备发送队列。
- 关闭接口
  - 该操作由net_device的stop函数指针指向的函数完成，该函数需要调用netif_stop_queue停止数据包传送

数据接收与发送

- 数据发送
  - 数据在实际发送的时候会调用net_device结构的hard_start_transmit函数指针指向的函数，该函数会将要发送的数据放入外发队列，并启动数据包发送
- 并发控制
  - 发送函数在指示硬件开始传送数据后就立即返回，但数据在硬件上的传送不一定完成。硬件接口的传送方式是异步的，发送函数又是可重入的，可利用net_device结构中的xmit_lock自旋锁来保护临界区资源
- 传输超时
  - 驱动程序需要处理超时带来的问题，内核会调用net_device的tx_timeout，完成超时需做的工作，并调用netif_wake_queue函数重启设备发送队列

数据接收与发送

- 数据接收
  - 中断方式：当网络设备接收到数据后触发中断，中断处理程序判断中断类型。如果是接收中断，则接收数据，并申请sk_buffer结构和数据缓冲区，根据数据的信息填写sk_buffer结构。然后将接收到的数据复制到缓冲区，最后调用netif_rx函数将sk_buffer传递给上层协议
  - 轮询方式：在轮询方式下，首个数据包到达产生中断后触发轮询过程，轮询处理程序首先关闭“接收中断”，在接收到一定数量的数据包并提交给上层协议后，再开中断等待下次轮询处理。使用netif_receive_skb函数向上层传递数据

查看状态与参数设置

- 链路状态：驱动程序可以通过查看设备的寄存器来获得链路状态信息。当链路状态改变时，驱动程序需要通知内核
  - void netif_carrier_off(struct net_device *dev);
  - void netif_carrier_on(struct net_device *dev);
- 设备状态：驱动程序的get_stats()函数向用户返回设备的状态和统计信息，保存在一个net_device_stats结构体。
- 设置MAC地址：调用ioctl并且参数为SIOCSIFHWADDR时，就会调用set_mac_address函数指针指向的函数。
- 接口参数设置：调用ioctl并且参数为SIOCSIFMAP时，就会调用set_config函数指针指向的函数，内核会给该函数传递一个ifmap的结构体。该结构体中包含了要设置的I/O地址、中断等信息

查看状态与参数设置

- 驱动程序的get_stats()函数用于向用户返回设备的状态和统计信息。这些信息保存在net_device_stats结构体中。

```
struct net_device_stats{
    unsigned long   rx_packets;         /* 收到的数据包数 */
    unsigned long   tx_packets;         /* 发送的数据包数 */
    unsigned long   rx_bytes;           /* 收到的字节数   */
    unsigned long   tx_bytes;           /* 发送的字节数   */
    unsigned long   rx_errors;          /* 收到的错误包数 */
    unsigned long   tx_errors;          /* 发送的错误包数 */
    unsigned long   rx_dropped;         /* 接收包丢包数   */
    unsigned long   tx_dropped;         /* 发送包丢包数   */
    unsigned long   multicast;          /* 收到的广播包数 */
    unsigned long   collisions;
    /* 详细接收错误信息: */
    unsigned long   rx_length_errors;   /* 接收长度错误 */
    unsigned long   rx_over_errors;     /* 溢出错误     */
    unsigned long   rx_crc_errors;      /* CRC校验错误  */
    unsigned long   rx_frame_errors;    /* 帧对齐错误   */
    unsigned long   rx_fifo_errors;     /* 接收fifo错误 */
    /* 详细发送错误信息 */
    unsigned long   tx_aborted_errors;  /* 发送中止     */
    unsigned long   tx_carrier_errors;  /* 载波错误     */
    unsigned long   tx_fifo_errors;     /* 发送fifo错误 */
    unsigned long   tx_window_errors;   /* 发送窗口错误 */
}
```

下周……

- 面向物联网的嵌入式系统前沿研究速览
  - RFID除了识别还可以当传感器用?
  - 毫米波！听起来高大上的技术，了解一下吧。
  - 你的智能手环/手表也可以当做手写板?
  - VR/AR技术当前面临的痛点是什么?
  - 无人机送外卖里的黑科技，了解一下?
  - 5G来了，真的会实现万物互联吗?
  - ……