
# 操作系统原理与安全

孙钰 系主任 博导

北京航空航天大学网络空间安全学院


# 操作系统原理与安全

崔剑 硕导

北京航空航天大学网络空间安全学院


# 目 录

CONTENTS

## 第十一章 文件系统实现

本章要点：

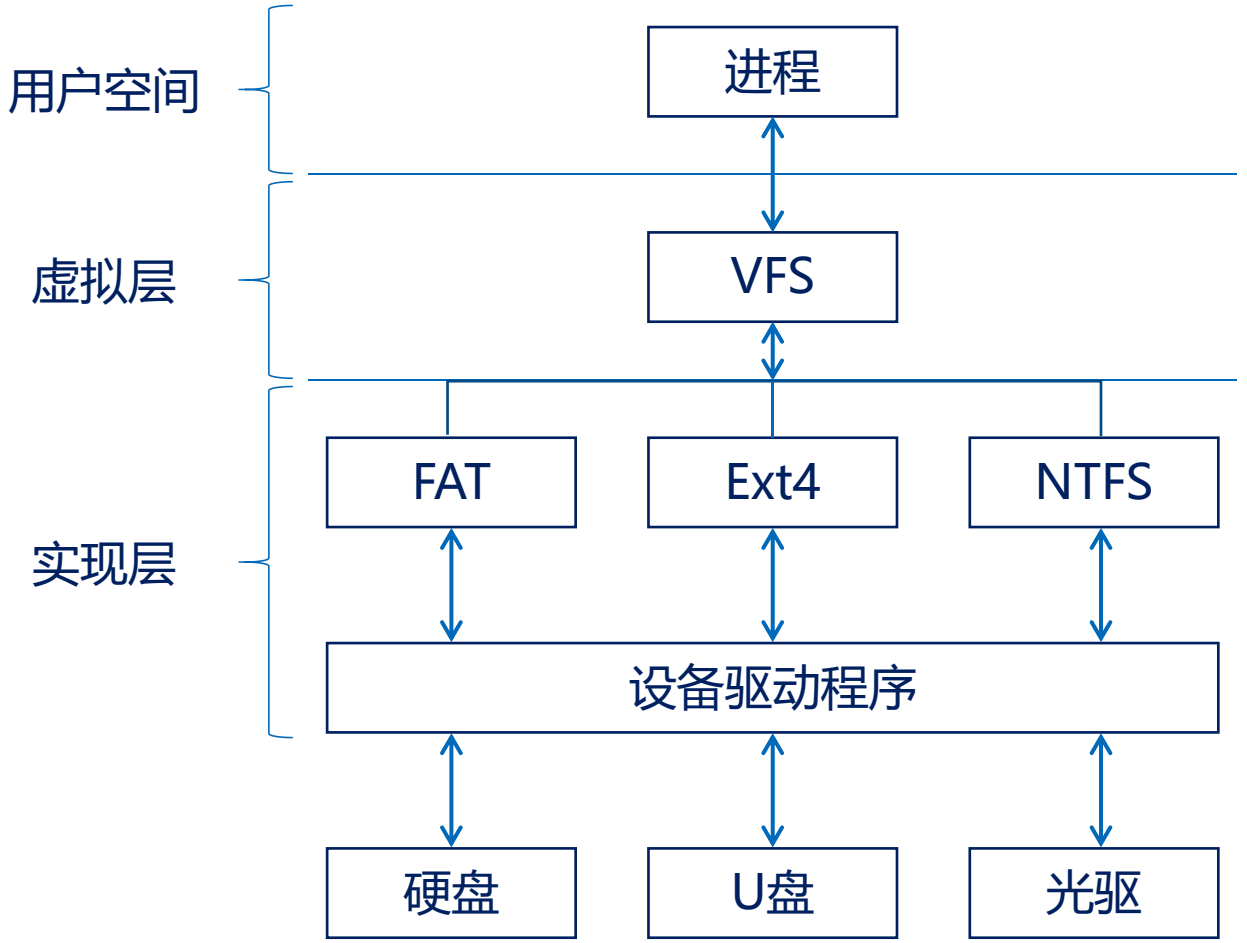
- 文件系统的整体架构（5层结构）
- FHS的层次结构
- VFS的四个对象及其层次关系
- 打开文件表的建立过程
- 以Ext4文件系统为例，理解超级块、inode块、位示图和目录项的作用
- inode索引和文件大小的计算关系
- I/O操作的常用三种方式的工作过程和特点
- 磁盘的结构、机械硬盘和固态硬盘的特点
- 磁盘和分区的关系，磁盘分区的格式


# 目 录

CONTENTS

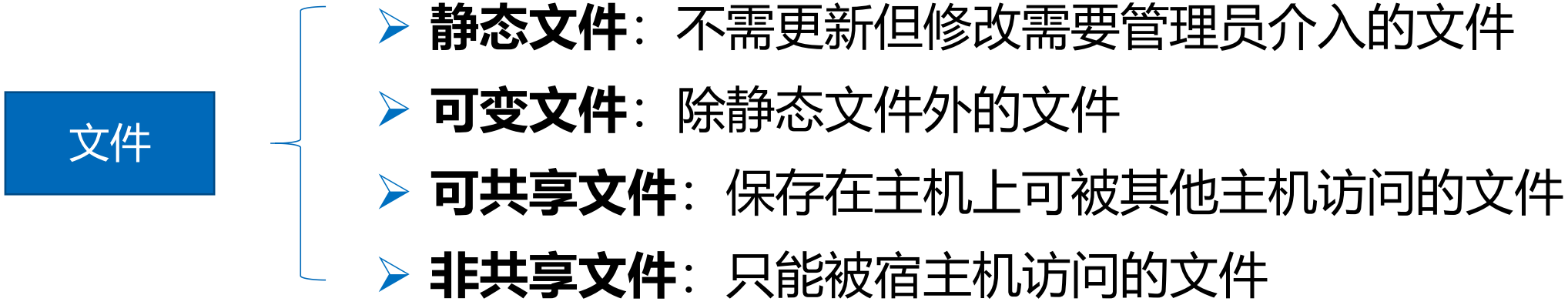
## 第十一章 文件系统实现

- 文件系统整体架构
  - ✓ 进程只和虚拟层交互
  - ✓ 进程使用统一接口访问文件系统
  - ✓ 虚拟文件系统VFS:
    - ① 物理文件系统的管理者
    - ② 向上提供统一文件系统接口
    - ③ 向下实现物理文件系统操作
    - ④ 隐藏不同文件系统的差异
    - ⑤ 实现统一的文件操作API
  - ✓ 操作系统可选择多种物理文件系统
  - ✓ 物理文件系统通过设备驱动操作设备



# 1. 文件系统概述

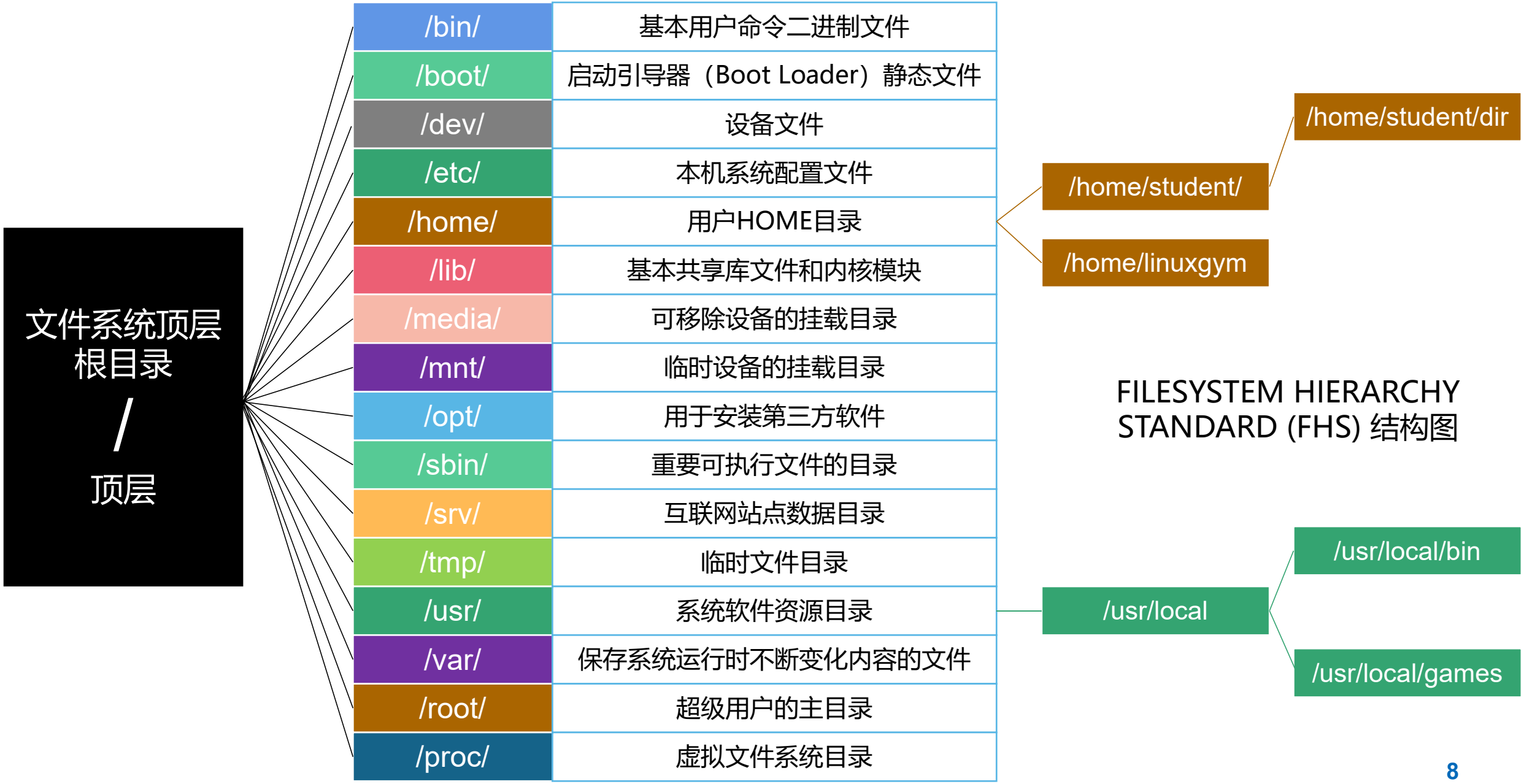
- 文件系统层次结构：
  - ✓ 大部分文件系统采用树状结构
  - ✓ FHS（Filesystem Hierarchy Standard，FHS）定义Linux的目录结构和目录内容
  - ✓ FHS将文件分为4类：




  - ✓ 按照分类组织到文件系统不同目录

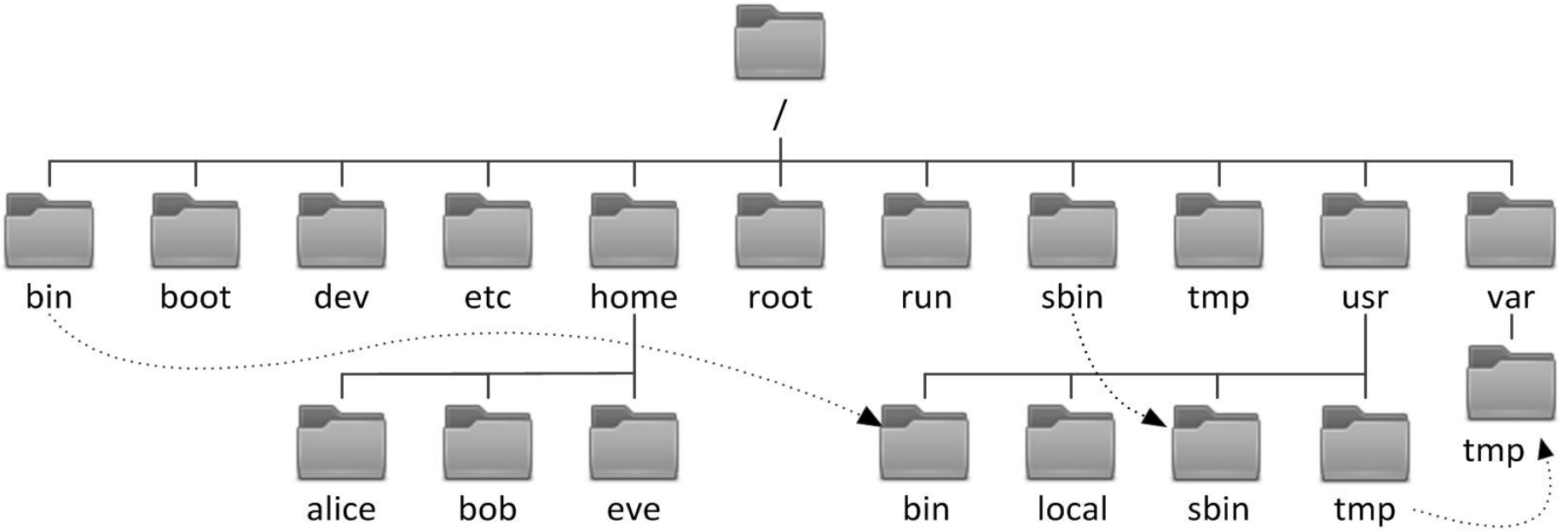
# 1. 文件系统概述-FHS






FILESYSTEM HIERARCHY STANDARD (FHS) 结构图

# FHS的树状层级结构


/
bin
boot
dev
etc
home
root
run
sbin
tmp
usr
var
alice
bob
eve
bin
local
sbin
tmp
tmp

# 1. 文件系统概述

| 目 录 | 描 述 |
| --- | --- |
| /bin | 可执行文件的目录，存放单用户维护模式下的必要命令，如cat, ls, cp等 |
| /boot | 存放引导文件的目录，存放kenel，initrd等引导文件，通常是单独的分区 |
| /dev | 存放设备文件的目录，如各种计算机的硬件设备，包括磁盘 |
| /etc | 配置文件目录 |
| /home | 用户主目录，包含保存的文件、个人设置等，每个用户单独一个子目录 |
| /lib | 系统库函数目录，包括/bin和/sbin下二进制文件需要依赖的执行库 |
| /media | 可移除设备（如光驱、U盘、移动硬盘）的挂载目录 |
| /mnt | 临时设备的挂载目录 |
| /opt | 可选软件安装目录，用于安装第三方软件 |
| /proc | 虚拟文件系统目录，用于在内存中保存数据，如uptime，network等 |
| /root | 超级用户的主目录 |
| /sbin | 重要可执行文件的目录，保存超级用户才能使用的命令 |
| /srv | 互联网站点数据目录，如FTP、WWW服务器的数据 |
| /tmp | 临时文件目录 |
| /usr | 系统软件资源目录，是UNIX Software Resource的缩写 |
| /var | 变量文件目录，保存系统运行时不断变化内容的文件 |

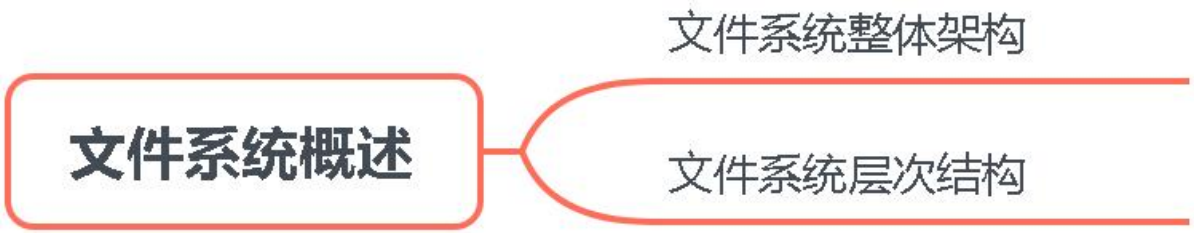
# 1. 文件系统概述

- 使用tree命令显示目录结构
- tree -L 1 -C /
  - ✓ -L指示显示几层子目录
  - ✓ -C表示使用彩色输出模式
- kali的根文件系统符合FHS
  - ✓ 部分目录采用软链接共享
  - ✓ /bin是/usr/bin的软链接
  - ✓ /lib是/usr/lib的软链接
  - ✓ /sbin是/usr/sbin的软链接

```
┌──(cuijianw㉿kali-vm-64)-[~]
└─$ tree -L 1 -C /
/
├── bin → usr/bin
├── boot
├── dev
├── etc
├── home
├── initrd.img → boot/initrd.img-6.1.0-kali5-amd64
├── initrd.img.old → boot/initrd.img-6.1.0-kali5-amd64
├── lib → usr/lib
├── lib32 → usr/lib32
├── lib64 → usr/lib64
├── libx32 → usr/libx32
├── lost+found
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin → usr/sbin
├── srv
├── sys
├── tmp
├── usr
├── var
├── vmlinuz → boot/vmlinuz-6.1.0-kali5-amd64
└── vmlinuz.old → boot/vmlinuz-6.1.0-kali5-amd64

23 directories, 4 files
```

# 1. 文件系统概述-小结



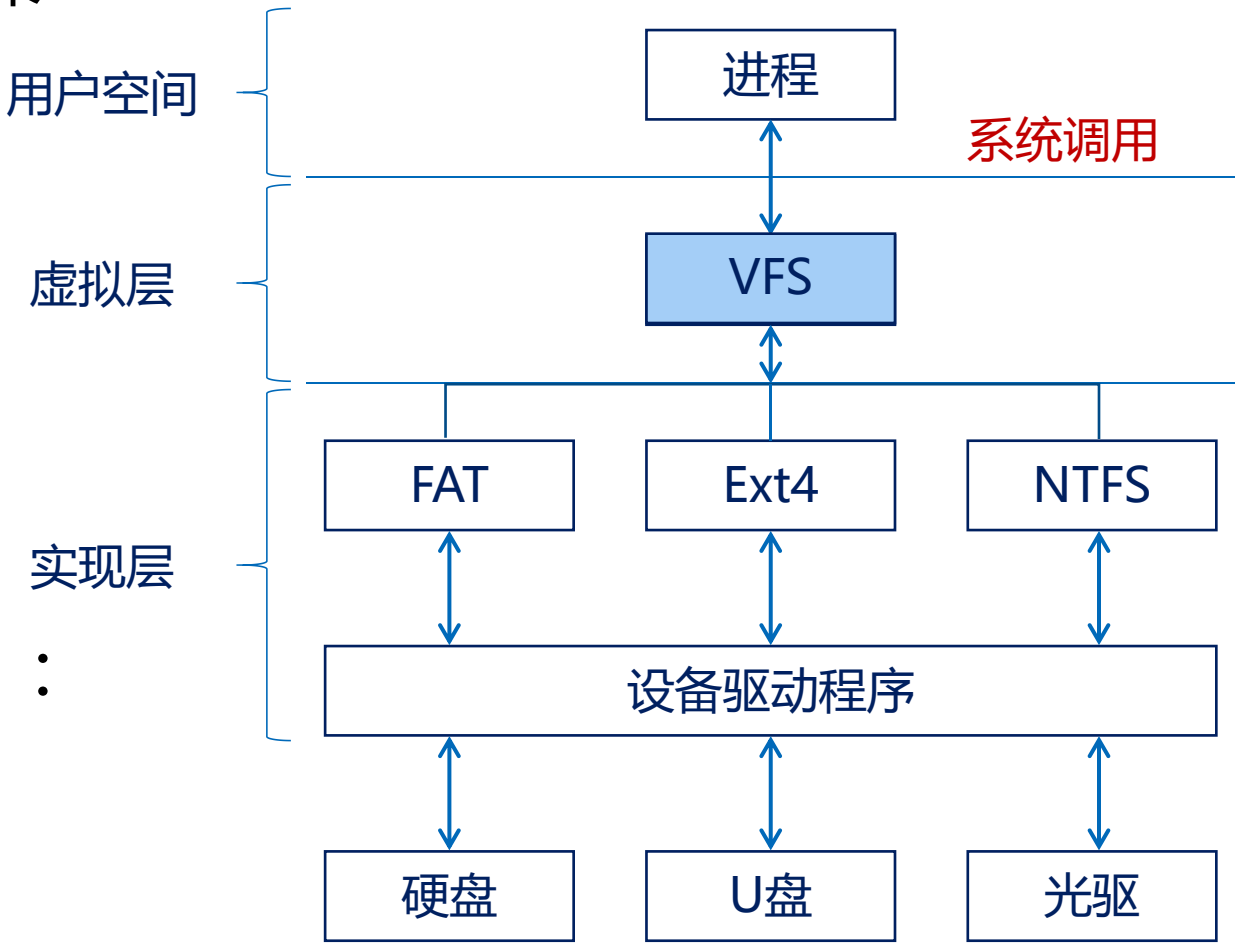

# 目 录

CONTENTS

## 第十一章 文件系统实现

# 2. 虚拟文件系统-概述

- 虚拟文件系统为用户提供文件系统操作统一接口，通过系统调用服务
- VFS支持三类底层文件系统：
  ① 基于磁盘的文件系统
  ② 基于网络的文件系统
  ③ 特殊的文件系统
- VFS包括四个操作对象（数据结构）：
  ① 超级块对象
  ② 索引节点对象
  ③ 目录项对象
  ④ 文件对象



- 超级块对象
  - ✓ 代表一个已挂载的物理文件系统，存储物理文件系统的信息

```
1.  // 源文件：include/linux/fs.h
2. struct super_block{
3.          struct list_head              s_list;          // 指向超级块链表
4.          ......
5.          unsigned long                 s_blocksize;     // 以字节为单位的块大小
6.          loff_t                        s_maxbytes;      // 文件的最大字节数
7.          struct file_system_type       * s_type;        // 文件系统类型
8.          const struct super_operations  * s_op;         // 超级块的操作函数地址表
9.          u32         s_time_gran;                       // 最后一次修改超级块的时间
10.         struct list_head              s_inodes;        // 指向inode链表
11.         void        * s_fs_info;                       // 指向缓存在内存中的物理文件系统的超级块
12.         ......
13.         struct block_device           * s_bdev;        // 指向超级块对象对应的块设备实例
14.         ......
15.   }
```

- 超级块对象

```
1.  // 源文件：include/linux/fs.h
2. struct super_block{
3.          struct list_head              s_list;          // 指向超级块链表
4.          ......
5.          unsigned long                 s_blocksize;     // 以字节为单位的块大小
6.          loff_t                        s_maxbytes;      // 文件的最大字节数
7.          struct file_system_type       * s_type;        // 文件系统类型
8.          const struct super_operations  * s_op;         // 超级块的操作函数地址表
9.          u32         s_time_gran;                       // 最后一次修改超级块的时间
10.         struct list_head              s_inodes;        // 指向inode链表
11.         void          * s_fs_info;                     // 指向缓存在内存中的物理文件系统的超级块
12.         ......
13.         struct block_device           * s_bdev;        // 指向超级块对象对应的块设备实例
14.         ......
15.   }
```

- 超级块对象

```
1.  // 源文件：include/linux/fs.h
2. struct super_block{
3.          struct list_head              s_list;          // 指向超级块链表
4.          ......
5.          unsigned long                 s_blocksize;     // 以字节为单位的块大小
6.          loff_t                        s_maxbytes;      // 文件的最大字节数
7.          struct file_system_type       * s_type;        // 文件系统类型
8.          const struct super_operations  * s_op;         // 超级块的操作函数地址表
9.          u32         s_time_gran;                       // 最后一次修改超级块的时间
10.         struct list_head              s_inodes;        // 指向inode链表
11.         void        * s_fs_info;                       // 指向缓存在内存中的物理文件系统的超级块
12.         ......
13.         struct block_device           * s_bdev;        // 指向超级块对象对应的块设备实例
14.         ......
15.   }
```

- 超级块对象

```
1.  // 源文件：include/linux/fs.h
2. struct super_block{
3.          struct list_head              s_list;          // 指向超级块链表
4.          ......
5.          unsigned long                 s_blocksize;     // 以字节为单位的块大小
6.          loff_t                        s_maxbytes;      // 文件的最大字节数
7.          struct file_system_type       * s_type;        // 文件系统类型
8.          const struct super_operations  * s_op;         // 超级块的操作函数地址表
9.          u32         s_time_gran;                       // 最后一次修改超级块的时间
10.         struct list_head              s_inodes;        // 指向inode链表
11.         void        * s_fs_info;                       // 指向缓存在内存中的物理文件系统的超级块
12.         ......
13.         struct block_device           * s_bdev;        // 指向超级块对象对应的块设备实例
14.         ......
15.   }
```

- 超级块对象
  - ✓对应于物理文件系统超级块或物理文件系统控制块
  - ✓文件系统挂载（mount）时，VFS调用函数alloc_super()读取物理文件系统超级块，并填充到内存的超级块对象中
  - ✓s_list对象指向VFS所有超级块对象的双向循环链表
  - ✓s_op指向所有超级块操作函数的地址：
    - 分配inode
    - 销毁inode
    - 读取inode
    - 写入inode
    - 文件同步

- 索引节点对象
  - ✓ 代表物理文件系统中一个文件或目录文件，包含操作该文件的全部信息，类FCB

```
1.  // 源文件：include/linux/fs.h
2. struct inode{
3.          umode_t          i_mode;          // 文件类型与访问权限
4.          kuid_t           i_uid;           // 所有者标识符
5.          unsigned long    i_ino;           // 索引节点号
6.          ……
7.          struct timespec64  i_atime;       // 上次访问文件的时间
8.          struct timespec64  i_mtime;       // 上次修改文件的时间
9.          struct timespec64  i_ctime;       // 上次修改inode的时间
10.         unsigned long       i_state;                  // 索引节点对象的状态是否为“脏”
11.         const struct file_operations  * i_fop;        // 指向索引节点对象的操作函数表
12.         struct super_block            * i_sb;         // 指向该索引节点对象所从属的超级块对象
13.         ……
14.         union {
15.                 ……
16.                 struct block_device   * i_bdev;     // 索引节点所关联块设备在内存中的实例
17.         }
18.  }
```

# 2. 虚拟文件系统-数据结构

- 索引节点对象
  - ✓索引节点对文件是唯一的
  - ✓索引节点对象会复制磁盘索引节点包含的数据
  - ✓i_state对象指示索引节点对象是否发生修改，避免数据不一致
  - ✓i_fop指向索引节点的操作接口函数
    - 创建新索引节点
    - 创建硬链接
    - 创建新目录
    - ……

- 目录项对象
  - ✓代表文件路径中的一个组成部分，即目录文件中的目录项

```
1.  // 源文件：include/linux/dcache.h
2. struct dentry{
3.          struct inode        * d_inode;        // 指向目录项所关联的索引节点对象
4.          struct qstr         d_name;           // 目录项对象的名称
5.          struct dentry       * d_parent;       // 指向父目录的目录项对象
6.          ……
7.          struct super_block * d_sb;            // 所属文件系统的超级块对象
8.          ……
9.          struct list_head    d_subdirs;        // 指向子目录项对象组成的链表
10.         ……
11.   }
```

- 目录项对象
  - ✓操作系统进行路径访问时，从根目录开始逐级查询目录项
  - ✓目录项提供文件名称和其索引节点对象的映射关系
  - ✓目录项对象在磁盘没有对应的数据结构，其是虚拟的
  - ✓执行路径名查找时，VFS会将目录项缓存
  - ✓当VFS在目录项缓存中没有命中时，其访问物理文件系统，并在VFS创建目录项对象和索引节点对象

- 文件对象

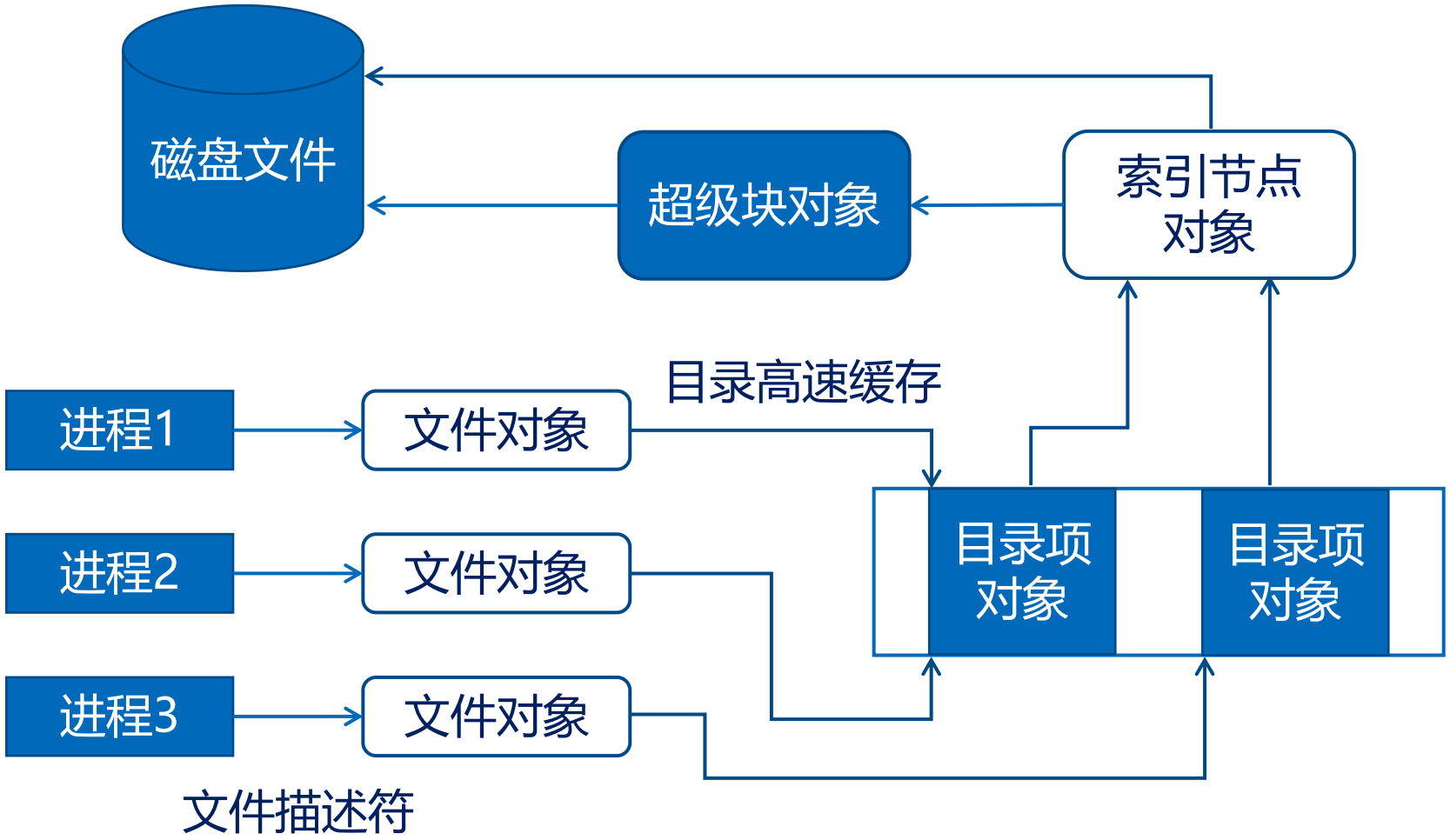
✓代表进程已打开的文件，是已打开文件在内核中的表示。

```
1.  // 源文件：include/linux/fs.h
2. struct file {
3.          struct inode                    * f_inode;      // 指向文件对应的索引节点
4.          const struct file_operations * f_op;        // 指向文件操作集合
5.          atomic_long_t                   f_count;        // 当前结构体的引用次数，用于回收
6.          fmode_t                         f_mode;         // 访问文件的模式
7.          loff_t                          f_pos;          // 文件的读写指针值
8.          struct address_space        * f_mapping;    // 指向页缓存映射的地址空间
9.          ……
10.         #define f_dentry            f_path.dentry   // 对应的目录结构
11.         struct path {                               // 保存的是文件在目录树中的位置
12.                 struct vfsmount  * mnt;             // 指向挂载描述符
13.                 struct dentry    * dentry           // 文件对应的目录项
14.         }
15.         ……
16.  }
```

- 文件对象
  - ✓ 进程使用函数open()打开文件时，VFS创建一个文件对象
  - ✓文件对象记录进程操作已打开文件的状态信息
  - ✓文件对象也记录指向物理文件系统的操作函数集合指针
  - ✓文件对象没有对应的磁盘数据
  - ✓多个进程操作同一个文件，生成多个文件对象，但索引节点对象和目录项对象是唯一的

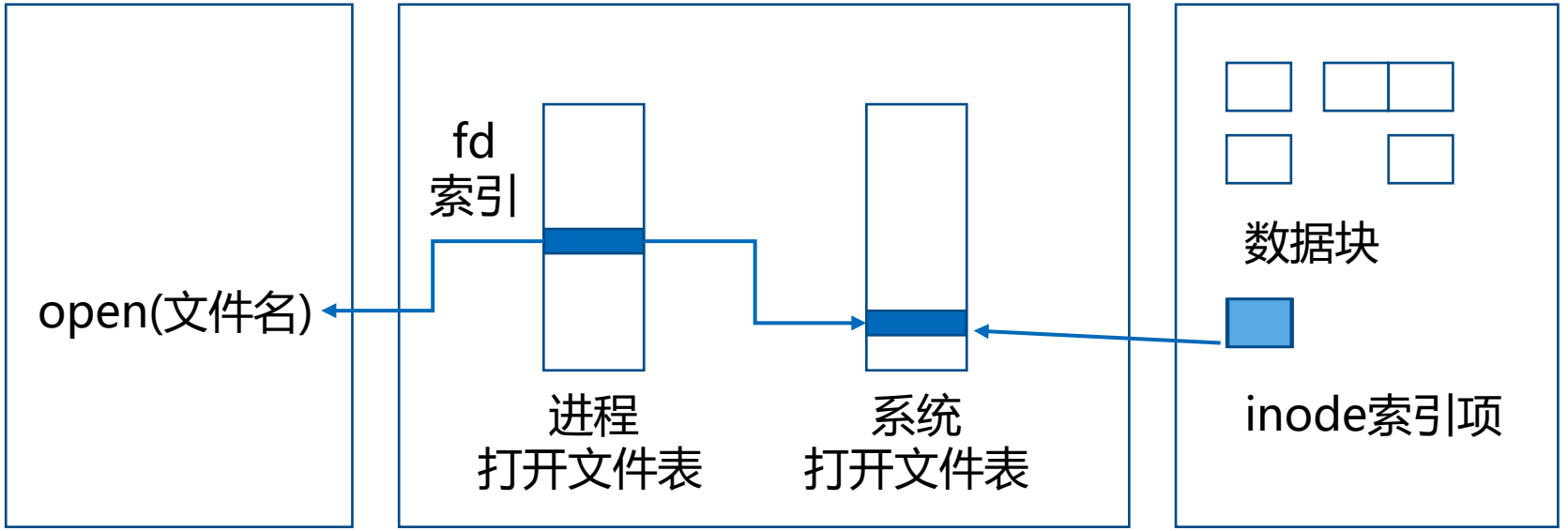
## VFS的对象交互

- ✓ 物理文件系统挂载时，在内存创建超级块对象
- ✓ 进程通过open()系统调用，分配文件描述符fd
- ✓ 解析文件路径，获取索引节点对象
- ✓ 通过索引节点对象，创建文件对象，关联二者
- ✓ 将文件对象添加到打开文件表，之后通过文件描述符访问文件



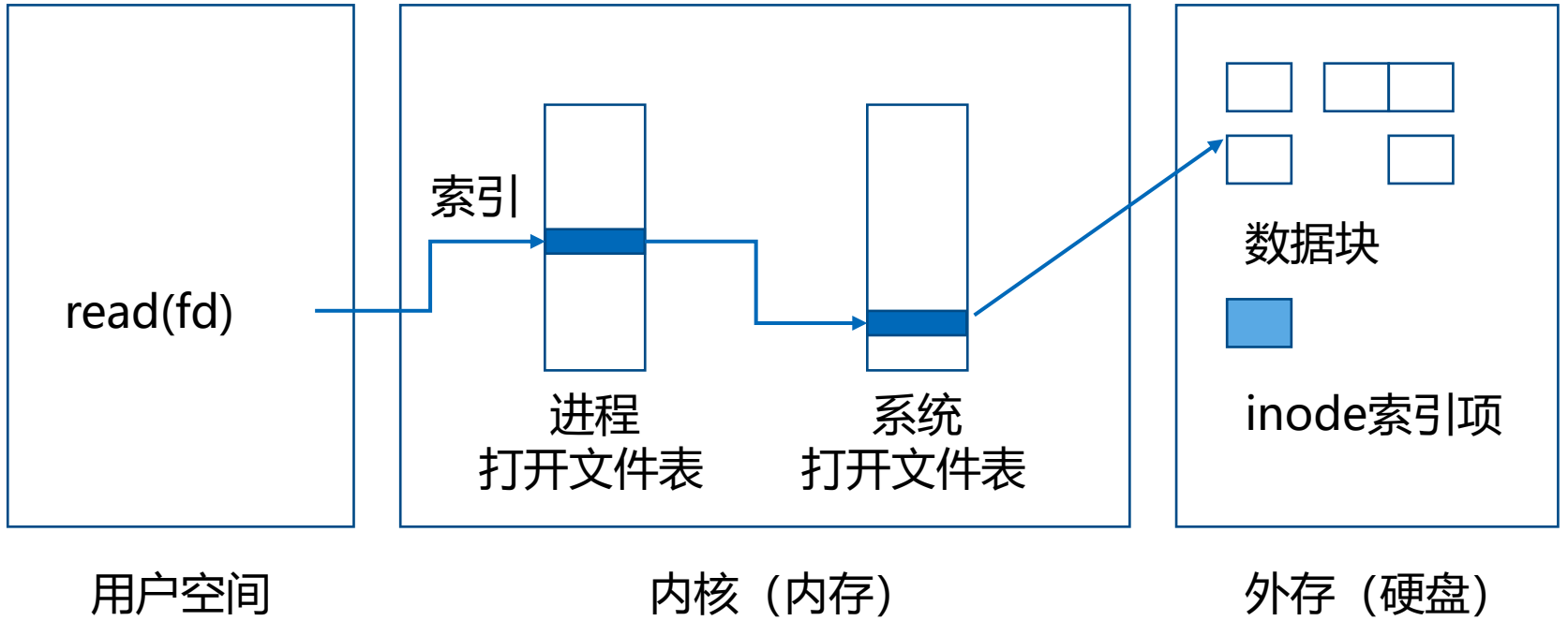
**打开文件表：**内存中分系统和文件两级文件打开表

- ✓ open()系统调用根据文件名搜索目录
- ✓ 通过目录项最终检索到文件的inode项
- ✓ 从外存的inode在内存建立文件对象，插入系统打开文件表
- ✓ 在进程打开文件添加条目，指向系统打开文件表对应项
- ✓ open()系统调用返回进程打开文件表的该项索引，即文件描述符fd

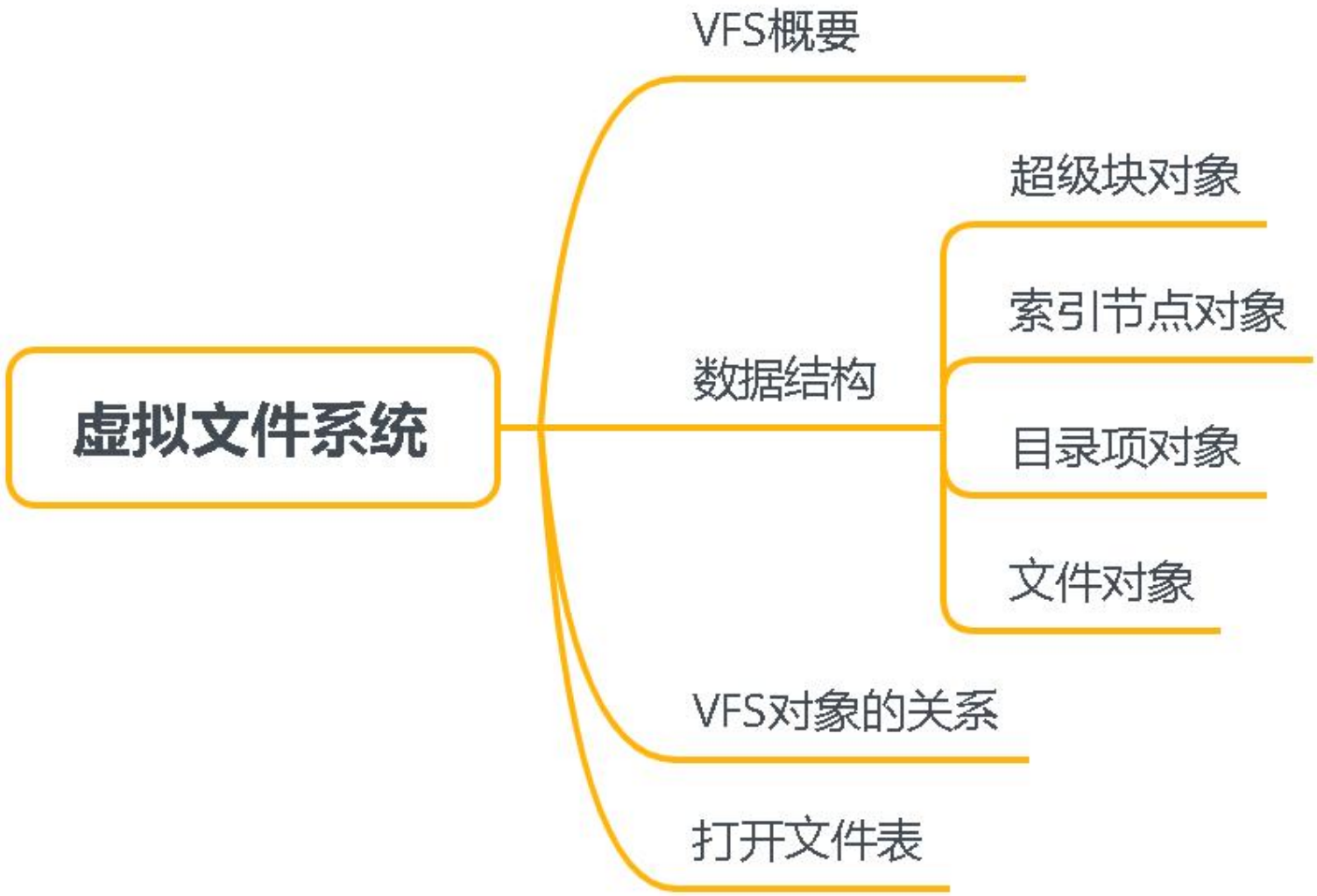


**打开文件表：**

- ✓ open()返回文件描述符后，不再依赖文件名
- ✓ 进程通过read()系统调用，利用fd找到进程打开文件表的索引项
- ✓ 进程的文件索引项指向系统打开文件表的VFS文件对象
- ✓ 通过该文件对象，即FCB，找到文件的数据块，并对其进行读取



# 2. 虚拟文件系统-小结



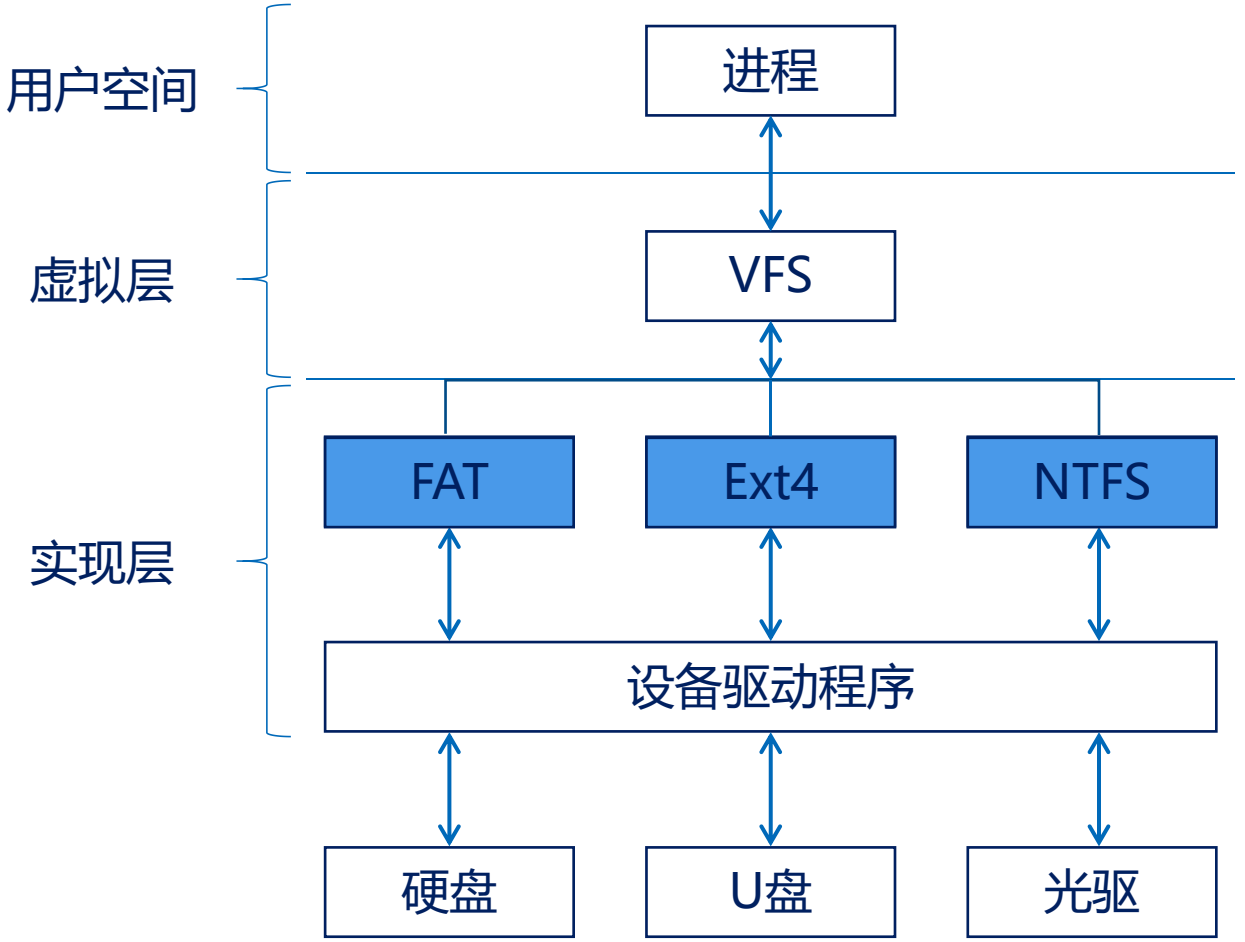

# 目 录

CONTENTS

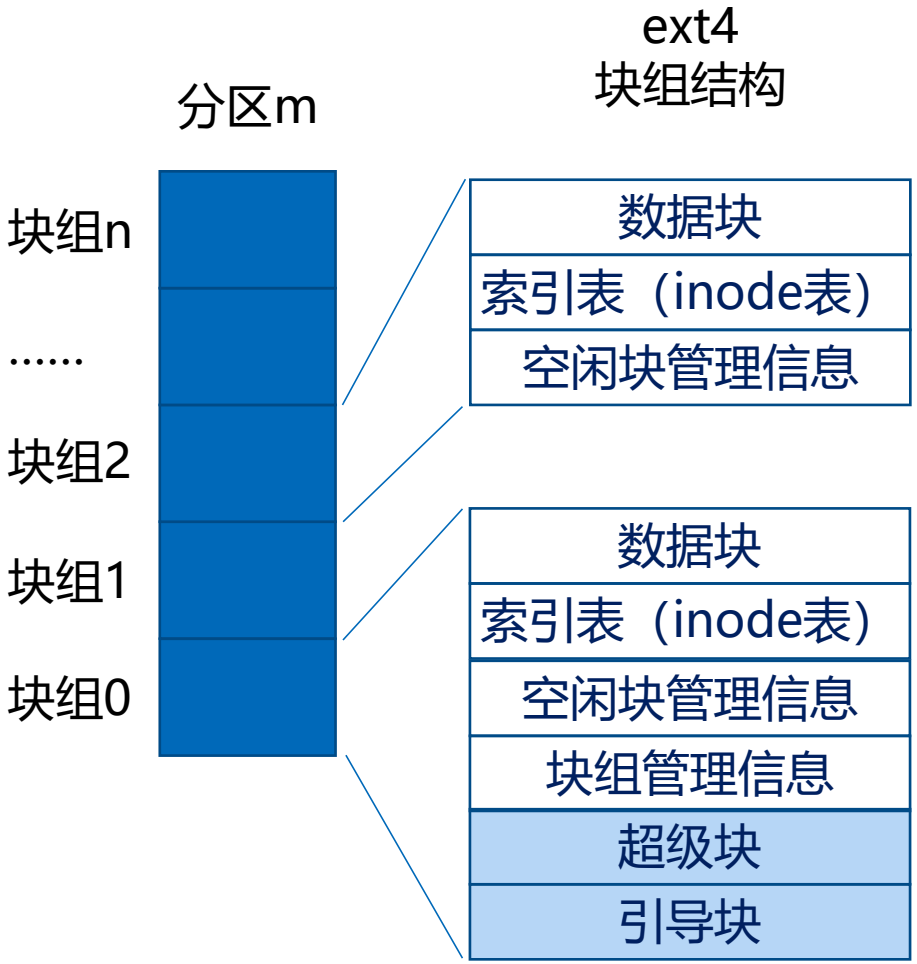
## 第十一章 文件系统实现

# 3. 文件系统实现-整体组织

- 操作系统以块为单位访问文件
- 文件系统（File System）提供高效便捷磁盘访问
  - ✓ 定义文件系统用户接口
    - ➢ 定义文件及其属性
    - ➢ 定义文件操作
    - ➢ 定义文件的目录结构
  - ✓ 映射逻辑文件系统到物理外存
    - ➢ 定义映射算法和数据结构
- 以Linux系统ext3、ext4文件系统为实例



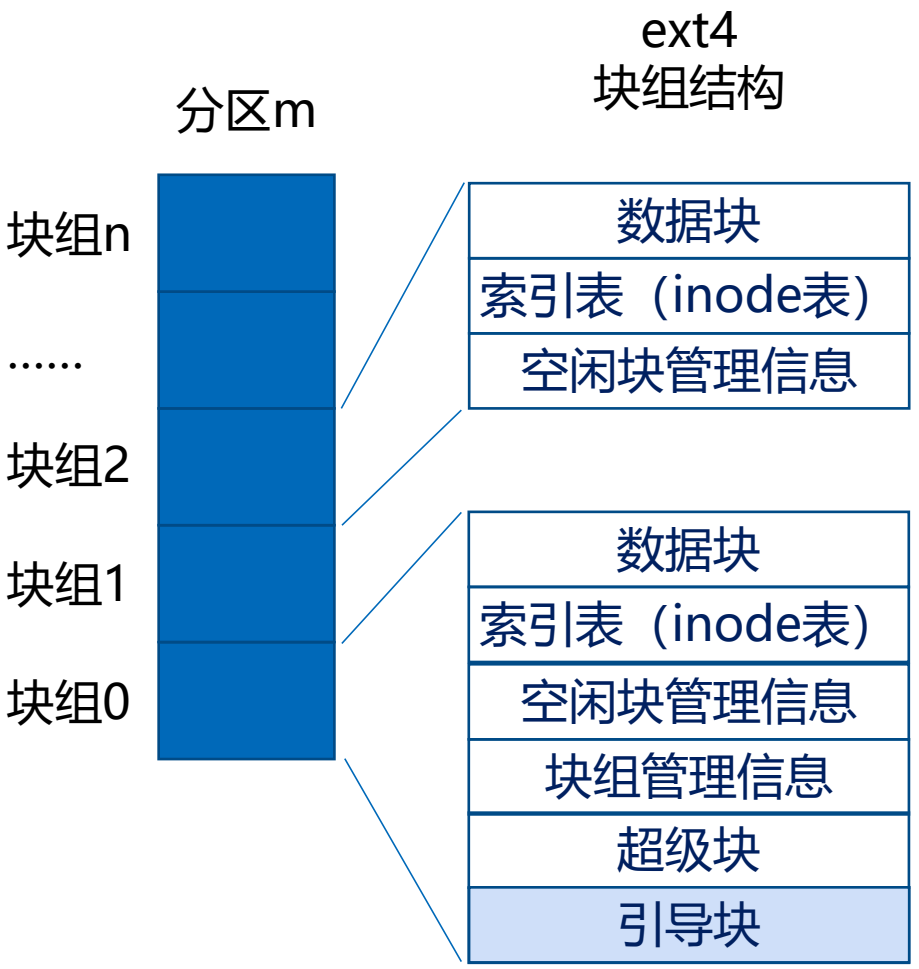
# 3. 文件系统实现-整体组织




- 引导块负责操作系统启动加载
- 超级块记录文件系统整体信息
- 块组管理信息即块组描述符GD，记录每个块组内如下信息：
  - ✓ 数据位图地址
  - ✓ inode位图地址
  - ✓ inode表地址
  - ✓ 空闲数据块数量
  - ✓ 可用inode数
- 空闲块管理，采用位图（bitmap）法标注块组内所有块是否空闲
- 索引表即inode表，记录文件的inode项
- 数据块即文件和目录保存的位置

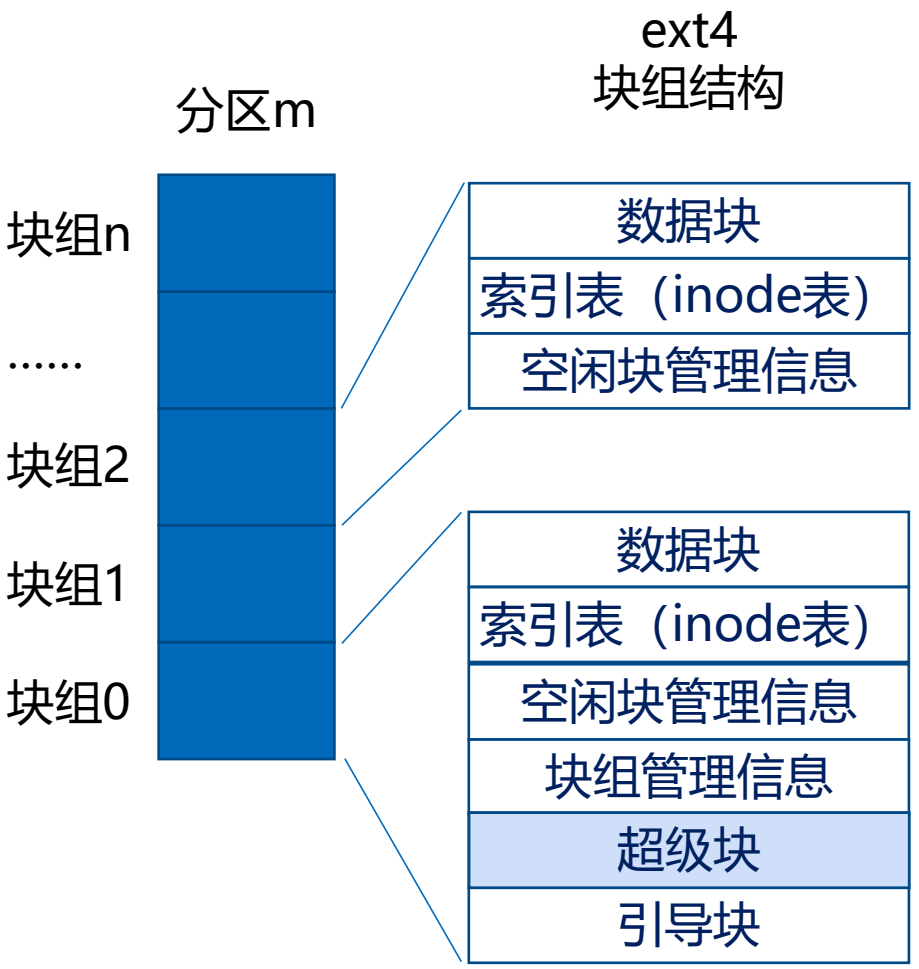
# 3. 文件系统实现-引导块

- 引导块负责该文件系统安装操作系统启动加载
- 引导块的信息不能被文件系统修改
- 引导块只存在块组0中
- 其他块组不存在引导块



# 3. 文件系统实现-超级块

- 超级块记录文件系统整体信息
  - ✓ 文件系统的inode总数
  - ✓ 文件系统大小
  - ✓ 空闲块数
  - ✓ 空闲inode数
  - ✓ 块大小
  - ✓ 文件系统类型
  - ✓ 文件系统状态
- 超级块大小固定为1KB
- 超级块存在于块组0，在部分其他块组中存有备份



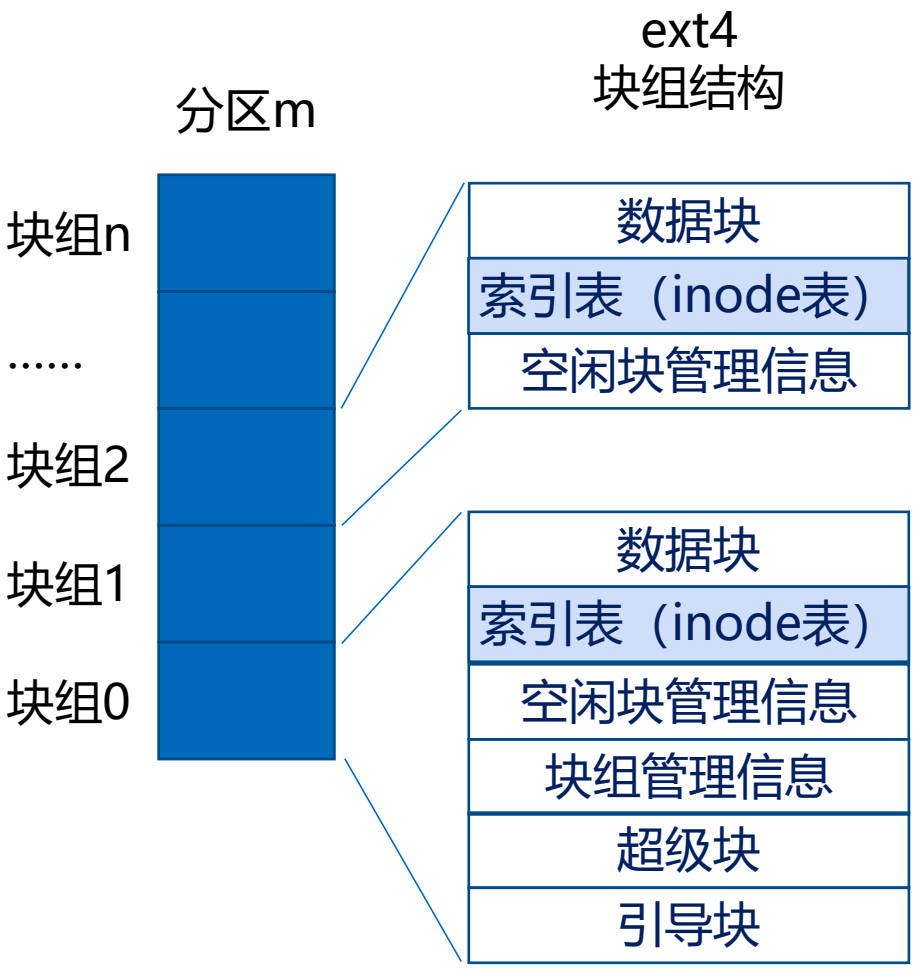
# 3. 文件系统实现-超级块

- 超级块的部分成员

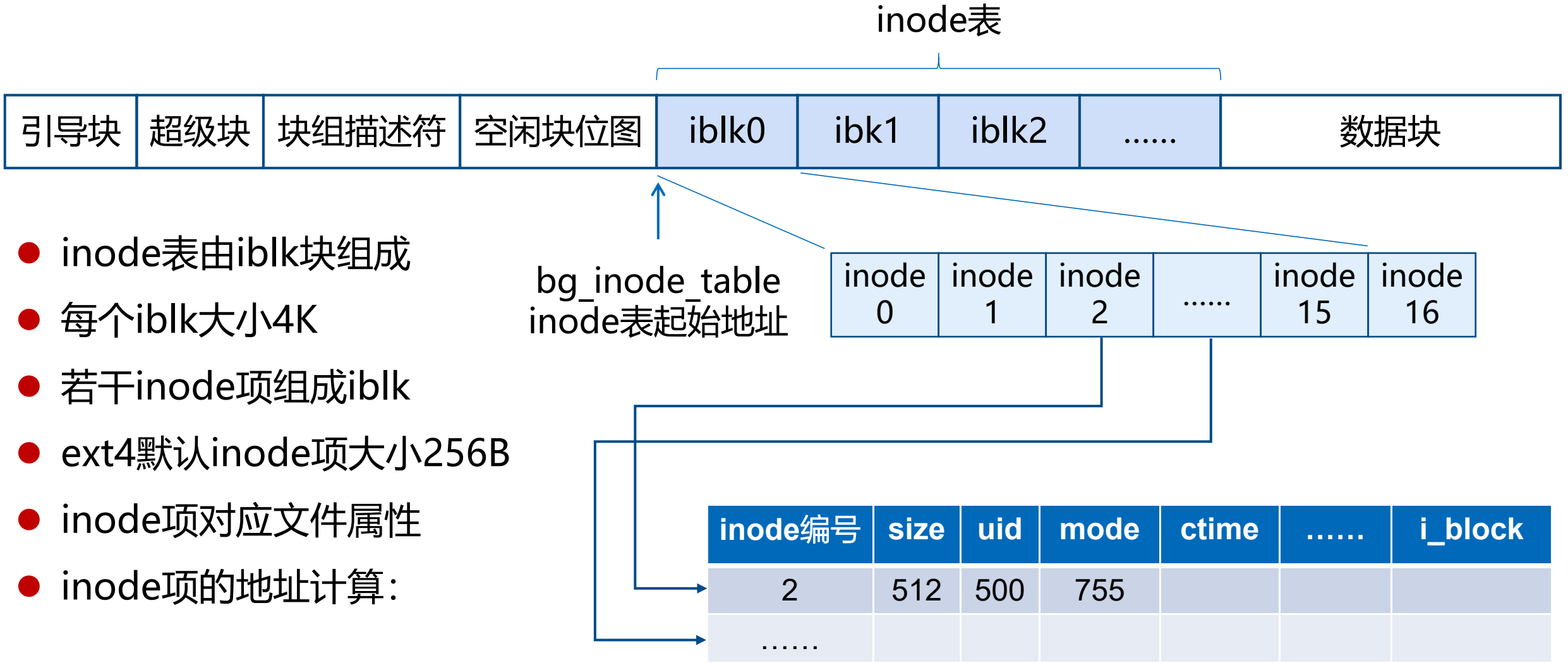
```
1.  // 源文件：fs/ext4/ext4.h
2. struct Ext4_super_block {
3.          __le32   s_inode_count;          /* inode 总数*/
4.          __le32   s_blocks_count_lo;      /* 以块为单位的文件系统的大小 */
5.          __le32   s_free_blocks_count_lo; /* 空闲块计数 */
6.          __le32   s_free_inode_count;     /* 空闲inode计数 */
7.          __le32   s_log_block_size;       /* 块的大小 */
8.          __le32   s_mtime;                /* 文件系统最后一次启动时间 */
9.          __le32   s_wtime;                /* 上一次写操作的时间 */
10.         __le32   s_creator_os;           /* 创建文件系统的操作系统 */
11.         __le16   s_magic;                /* 文件系统魔术数（幻数），代表其类型 */
12.         __le16   s_state;                /* 文件系统的状态 */
13.         ……
14.  }
```

# 3. 文件系统实现-索引表（inode表）

- 文件系统采用索引节点，即inode记录文件的信息
- 每个文件和目录都对应一个inode
- inode记录如下内容：
  - ✓ 文件大小
  - ✓ 文件所有者
  - ✓ 文件权限
  - ✓ 文件的访问时间、修改时间
  - ✓ 链接数（硬链接使用）
  - ✓ 文件数据块地址
- inode不记录文件和目录的名称
- 名称和inode的映射由**目录项**实现



# 3. 文件系统实现-inode文件组织




- inode表由iblk块组成
- 每个iblk大小4K
- 若干inode项组成iblk
- ext4默认inode项大小256B
- inode项对应文件属性
- inode项的地址计算：

inode地址 = inode号 * inode大小 + bg_inode_table

# 3. 文件系统实现-inode文件组织



| 引导块 | 超级块 | 块组描述符 | 空闲块位图 | iblock0 | iblock1 | iblock2 | …… | 数据块 |
|---|---|---|---|---|---|---|---|---|

- 每个inode对应一个文件/目录
- inode总数决定该分区最大文件数量
- 格式化分区时可指定inode数量
- 查看inode：dumpe2fs -h /dev/sda1

文件系统状态

inode总数

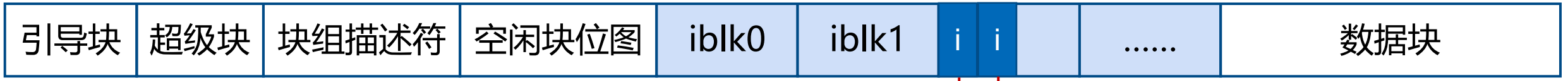
物理块数量

物理块大小

```
┌──(cuijianw㉿kali-vm-64)-[~]
└─$ sudo dumpe2fs -h /dev/sda1
[sudo] cuijianw 的密码：
dumpe2fs 1.46.6 (1-Feb-2023)
Filesystem volume name:   <none>
Last mounted on:          /
Filesystem UUID:          e99d2c79-7006-4d15-91e7-079656a3a8ca
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype
tent 64bit flex_bg sparse_super large_file huge_file dir_nlink extra_isize meta
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              1905008
Block count:              7613952
Reserved block count:     380697
Overhead clusters:        163616
Free blocks:              4256018
Free inodes:              1463109
First block:              0
Block size:               4096
Fragment size:            4096
Group descriptor size:    64
Reserved GDT blocks:      1024
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         8176
```

# 3. 文件系统实现-inode多级索引





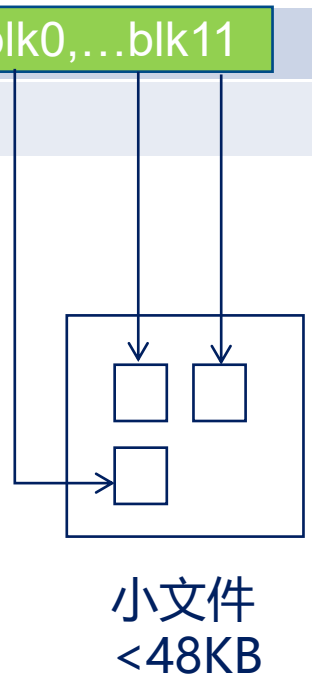

- inode中i_block记录文件的物理块
- 如用直接索引方法：
  - ✓ inode支持的最大文件：

  文件大小$_{直接索引}$=i_block数量*物理块大小
- ext4文件系统inode项总大小为：256B
- 只用直接索引无法支持大文件的索引
- 解决办法：采用直接索引+多级索引
  - ✓ 直接索引：小文件
  - ✓ 一级、二级、间接索引：较大文件
  - ✓ 三级间接索引：巨大文件

| inode编号 | size | uid | mode | ctime | ..... | i_block |
|---|---|---|---|---|---|---|
| 2 | 512 | 500 | 755 | | | bk0,bk1… |
| ..... | | | | | | |

## inode直接索引

| inode编号 | size | uid | mode | ctime | …… | i_block |
|---|---|---|---|---|---|---|
| 2 | 512 | 500 | 755 | | | blk0,…blk11 |
| …… | | | | | | |

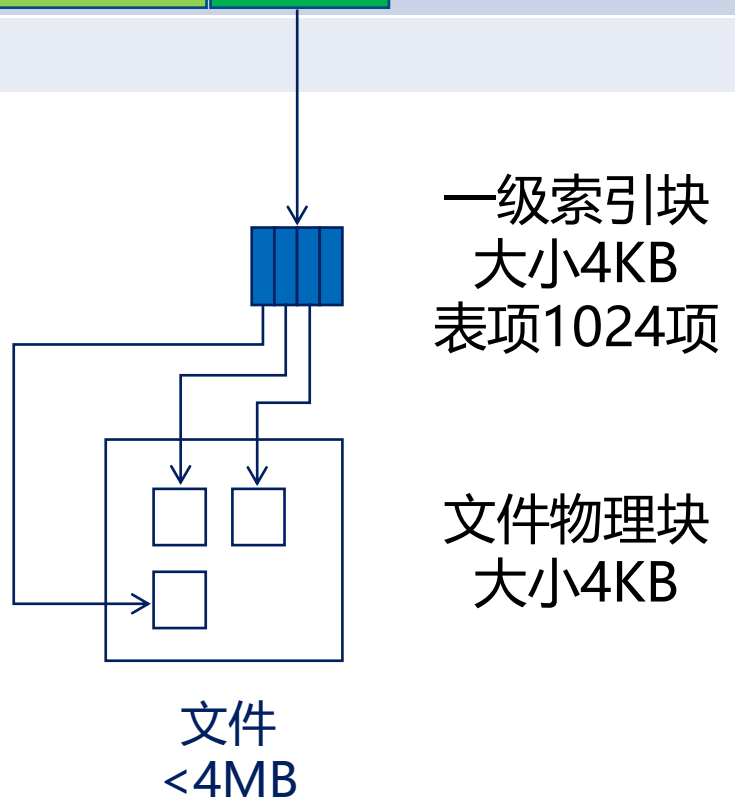



- ext4中物理块大小为4KB
- inode直接索引：
  - ✓ i_block中记录最多12个文件的物理块
  - ✓ 支持文件大小：12*4KB = 48KB
  - ✓ 小文件直接将物理块记录到i_block

## inode一级间接索引

| inode编号 | size | uid | mode | ctime | …… | i_block |
|---|---|---|---|---|---|---|
| 2 | 512 | 500 | 755 | | | blk0,…blk11 \| blk12 |
| …… | | | | | | |

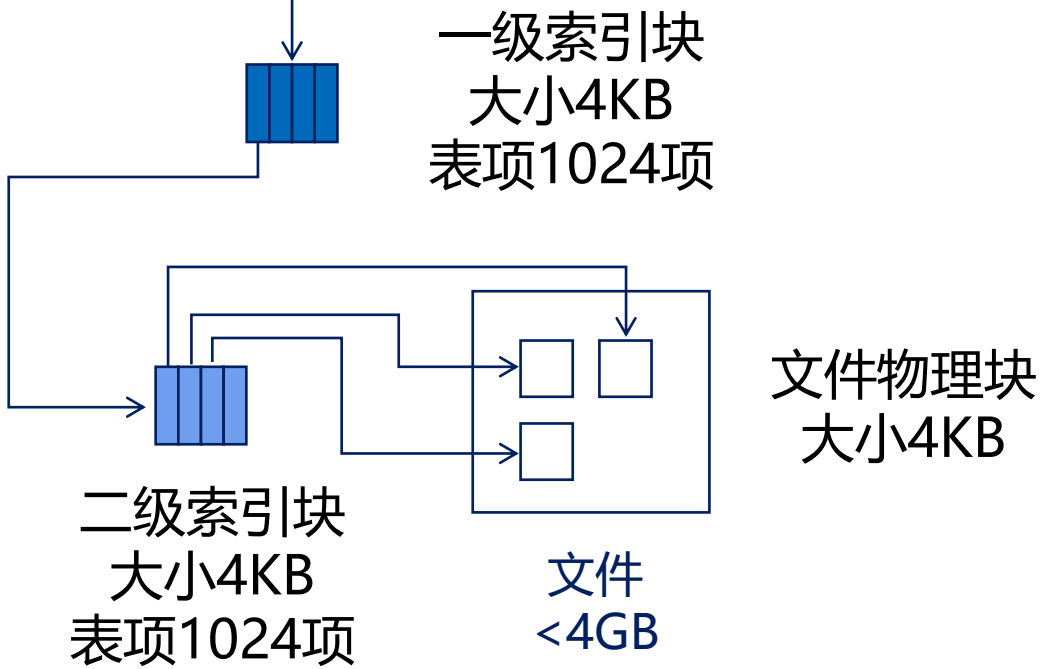
- ext4中物理块大小为4KB
- inode一级间接索引：
  - ✓ i_block中记录索引块
  - ✓ 索引块指向文件的物理块
  - ✓ i_block可记录1个一级索引块
  - ✓ 索引块中的物理块指针大小为4B
  - ✓ 支持文件大小：1*4096/4*4KB=4MB



## inode二级间接索引

| inode编号 | size | uid | mode | ctime | …… | i_block |
|---|---|---|---|---|---|---|
| 2 | 512 | 500 | 755 | | | blk0,…blk11 \| blk12 \| blk13 |
| …… | | | | | | |

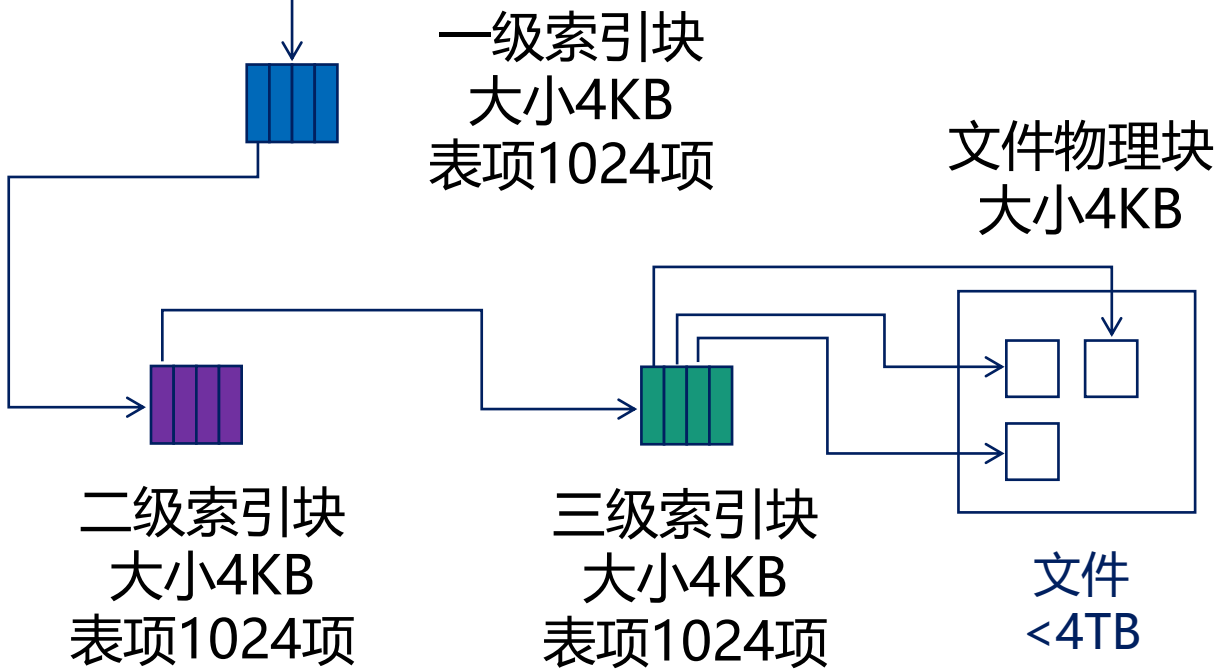
- ext4中物理块大小为4KB
- inode二级间接索引：
  - ✓ i_block中记录一级索引块
  - ✓ 一级索引块指向二级索引块
  - ✓ 二级索引块指向文件的物理块
  - ✓ i_block可记录1个二级索引块
  - ✓ 索引块中的物理块指针大小为4B
  - ✓ 支持文件大小：1*4096/4**4096/4*4KB=4GB



## inode三级间接索引

| inode编号 | size | uid | mode | ctime | …… | i_block |
|---|---|---|---|---|---|---|
| 2 | 512 | 500 | 755 | | | blk0,…blk11 \| blk12 \| blk13 \| blk14 |
| …… | | | | | | |

- inode三级间接索引：
  - ✓ i_block中记录一级索引块
  - ✓ 一级索引块指向二级索引块
  - ✓ 二级索引块指向三级索引块
  - ✓ 三级索引块指向文件的物理块
  - ✓ i_block可记录1个三级索引块
  - ✓ 索引块中的物理块指针大小为4B
  - ✓ 支持文件大小：1*4096/4**4096/4*4096/4*4KB=4TB



## inode混合索引层次关系

| inode编号 | size | uid | mode | ctime | …… | i_block |
|---|---|---|---|---|---|---|
| 2 | 512 | 500 | 755 | | | blk0,…blk11 \| blk12 \| blk13 \| blk14 |
| …… | | | | | | |

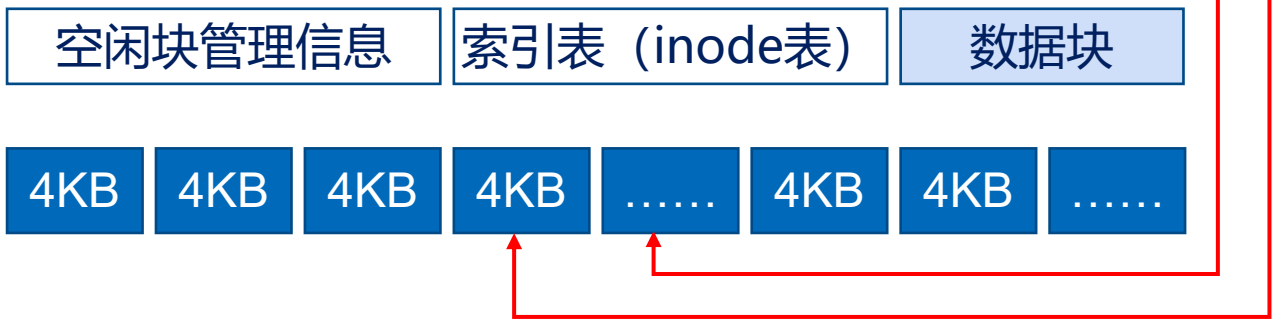
- inode混合索引层次关系为
  - ✓ 文件小于等于48KB时，首先用直接记录文件的物理块
  - ✓ 文件小于等于48KB+4MB时，先记录文件12个物理块，再用一级索引记录文件剩余物理块
  - ✓ 文件小于等于48KB+4MB+4GB时，用直接索引、一级索引和二级索引记录文件物理块
  - ✓ 文件小于等于48KB+4MB+4GB+4TB时，用直接索引、一级、二级和三级索引记录物理块
- inode扩展（Extend）索引
  - ✓ 当文件很大时，索引项过多，造成文件修改、删除代价过大
  - ✓ ext4文件系统改用扩展索引（Extend）方式，采用B+树完成大文件的索引

# 3. 文件系统实现-inode目录组织

- inode不存储文件/目录名称
- 由目录项给出文件名-inode号映射关系
- 检索文件必须读取目录文件信息
- 检索时只关心文件名，不会读入文件其他信息
- **按名存取**
  - ✓ 目录项只包含文件名和文件的inode号
  - ✓ 通过目录项检索文件名
  - ✓ 检索成功通过inode号读入文件其他信息
- 特点
  - ✓ 减小了目录文件的目录项体积
  - ✓ 提高了文件检索和访问速度

inode目录文件

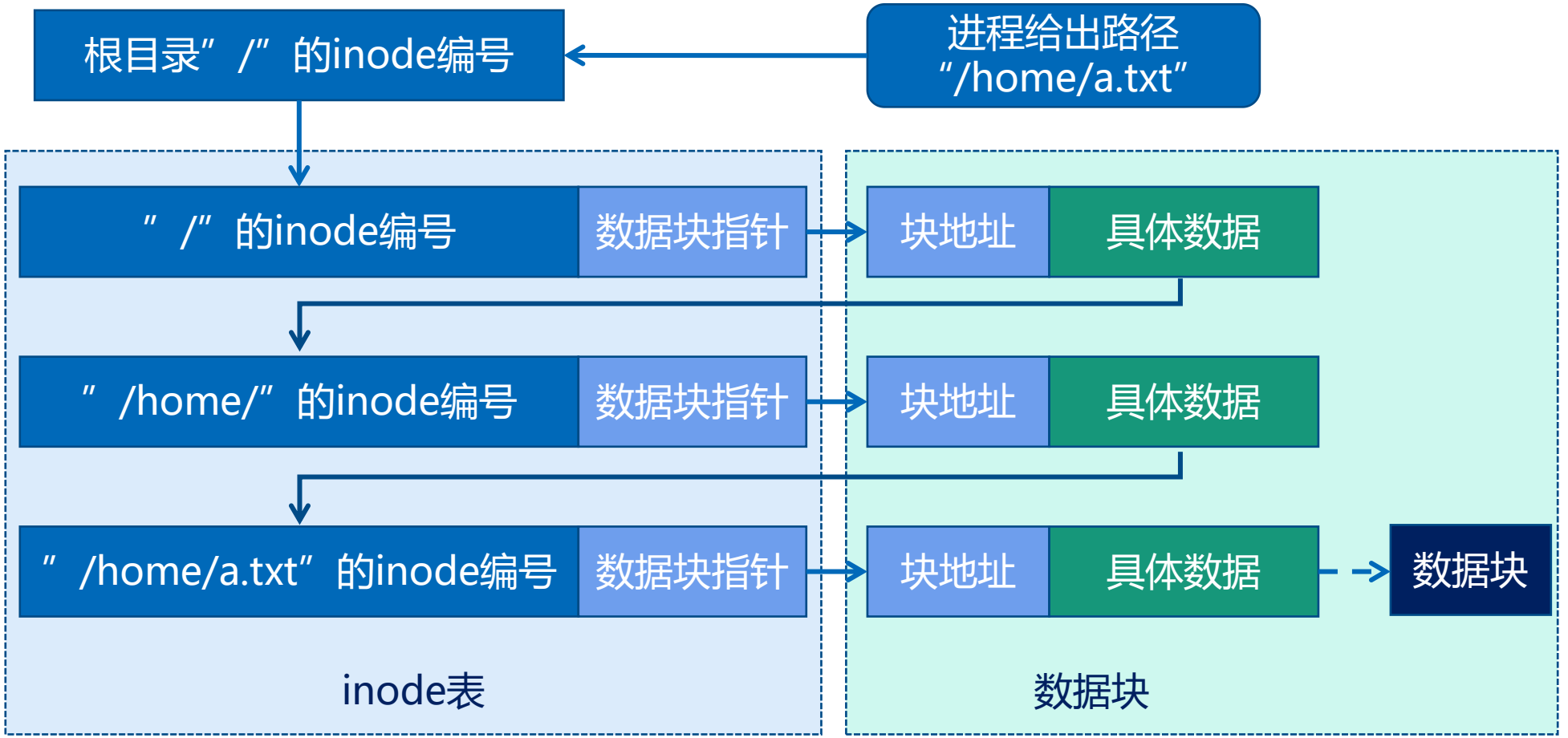
| 文件名 | inode编号 |
|---|---|
| mcu_src | 6 |
| 记账 | 512 |
| …… | …… |
| lcd插图 | 780 |
| …… | …… |



# 3. 文件系统实现-inode目录组织

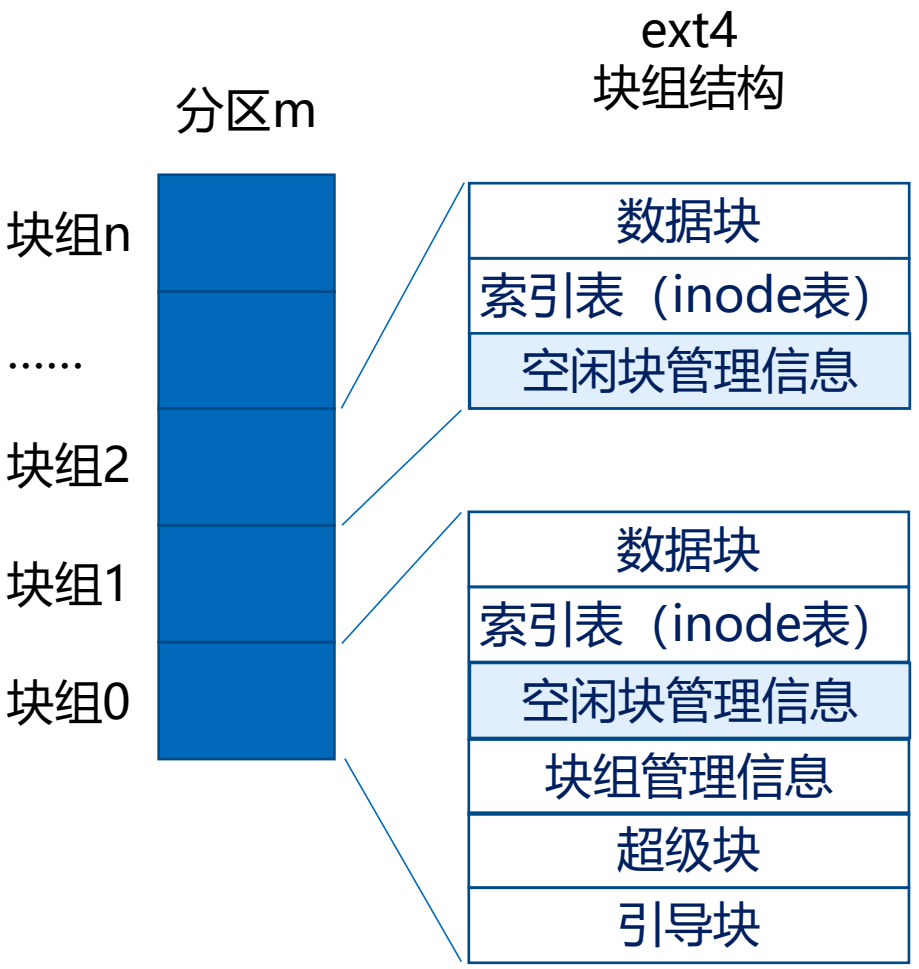
从根目录逐层解析读取目录项，检索文件的inode号，并访问文件数据块

- ✓ 读取根目录inode，根目录inode=2
- ✓ 用根目录inode找到其目录文件数据块
- ✓ 在根目录数据块找到home的inode
- ✓ 用home的inode找到其目录文件数据块
- ✓ 在home数据块找到a.txt的inode
- ✓ 用a.txt的inode找到其数据块
- ✓ 读取文件数据块



# 3. 文件系统实现-空闲空间管理

- 文件系统管理空闲空间
  - ✓ inode表空闲空间
  - ✓ 数据块空闲空间
- 空闲空间管理功能
  - ✓ 对空闲块的组织和管理
  - ✓ 空闲块的分配
  - ✓ 空闲块的回收
- 空闲空间方法
  - ✓ 空闲表法
  - ✓ 空闲链表法-早期UNIX系统
  - ✓ 位示图法-Ext4文件系统
  - ✓ 成组链接法



# 3. 文件系统实现-空闲表法

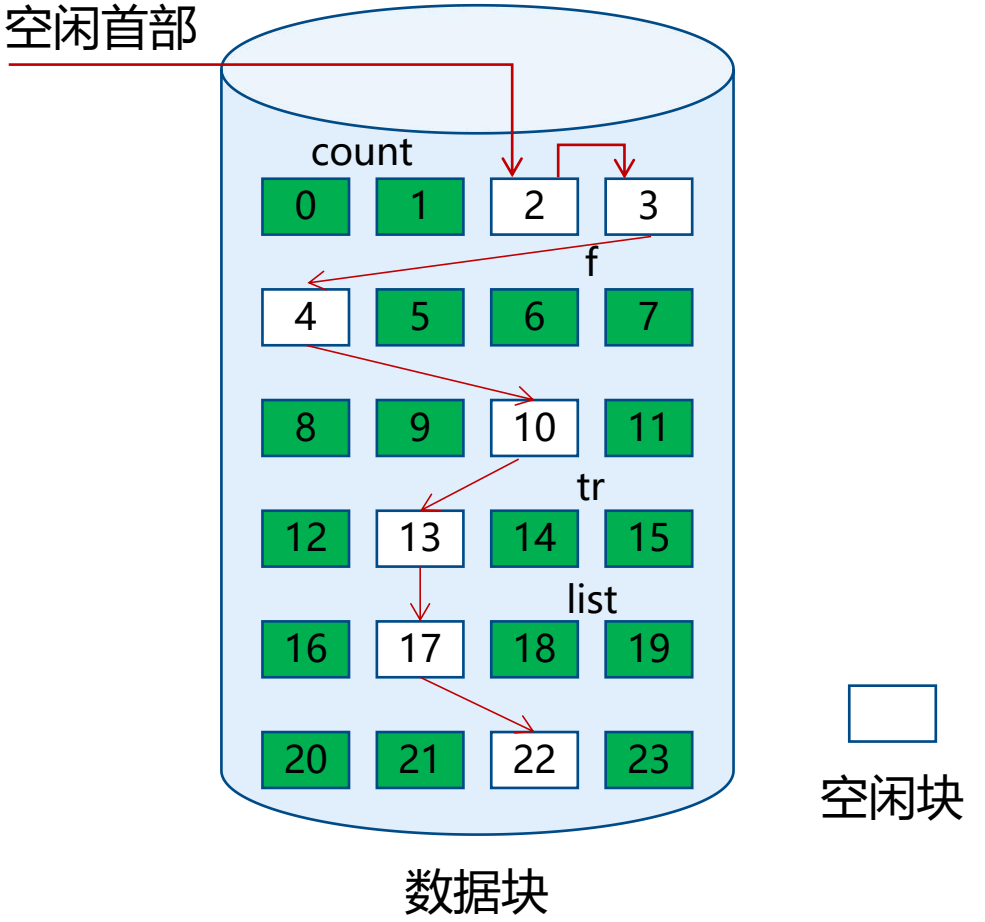
- 属于连续分配方式
  - ✓ 为文件分配一块连续存储空间
  - ✓ 为空闲区建立一张空闲表
- 空闲区分配算法
  - ✓ 首次适应算法
  - ✓ 最佳适应算法
- 空闲区回收算法
  - ✓ 增加空闲表项
  - ✓ 合并相邻空闲区
- 优点
  - ✓ 简单易实现
- 缺点
  - ✓ 需要额外空间来存放空闲表
  - ✓ 空闲表操作相当耗时

空闲表

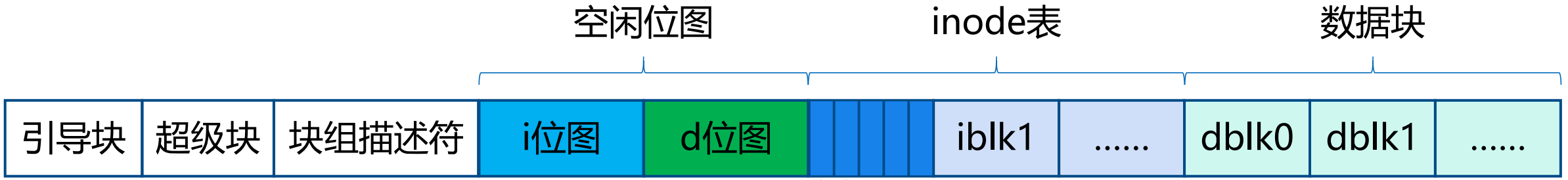
| 序号 | 起始空闲块 | 空闲区长度 |
|---|---|---|
| 1 | 2 | 4 |
| 2 | 9 | 3 |
| 3 | 15 | 5 |
| 4 | 1990 | 12 |
| 5 | …… | …… |

# 3. 文件系统实现-空闲链表法

- 将所有空闲块链接为一张链表
  - ✓ 文件系统记录空闲块首部
  - ✓ 分配
    - ➢ 从表头摘下如果空闲块给文件
  - ✓ 回收
    - ➢ 将空闲块加入链表尾部
- 优点
  - ✓ 不需专用块存放管理信息
- 缺点
  - ✓ 遍历链表，增加I/O操作
  - ✓ 难得到连续空间
  - ✓ 多次操作后，链表顺序和块号不一致



# 3. 文件系统实现-位示图法




二进制位bit

| | Byte0 | Byte1 | ..... |
|---|---|---|---|
| D0 | 0 | | |
| D1 | 0 | | |
| D2 | 0 | | |
| D3 | 1 | | |
| D4 | 0 | | |
| D5 | | | |
| D6 | | | |
| D7 | | | |

- 空闲位示图区分为两个区域
  - ✓ inode位示图-i位图
  - ✓ 数据块位示图-d位图
- inode位示图标注inode表中表项是否空闲
- i位图区每个二进制位（bit）对应inode表一个表项
  - ✓ 数据位1表示表项空闲
  - ✓ 数据位0表示表项非空闲

# 3. 文件系统实现-位示图法

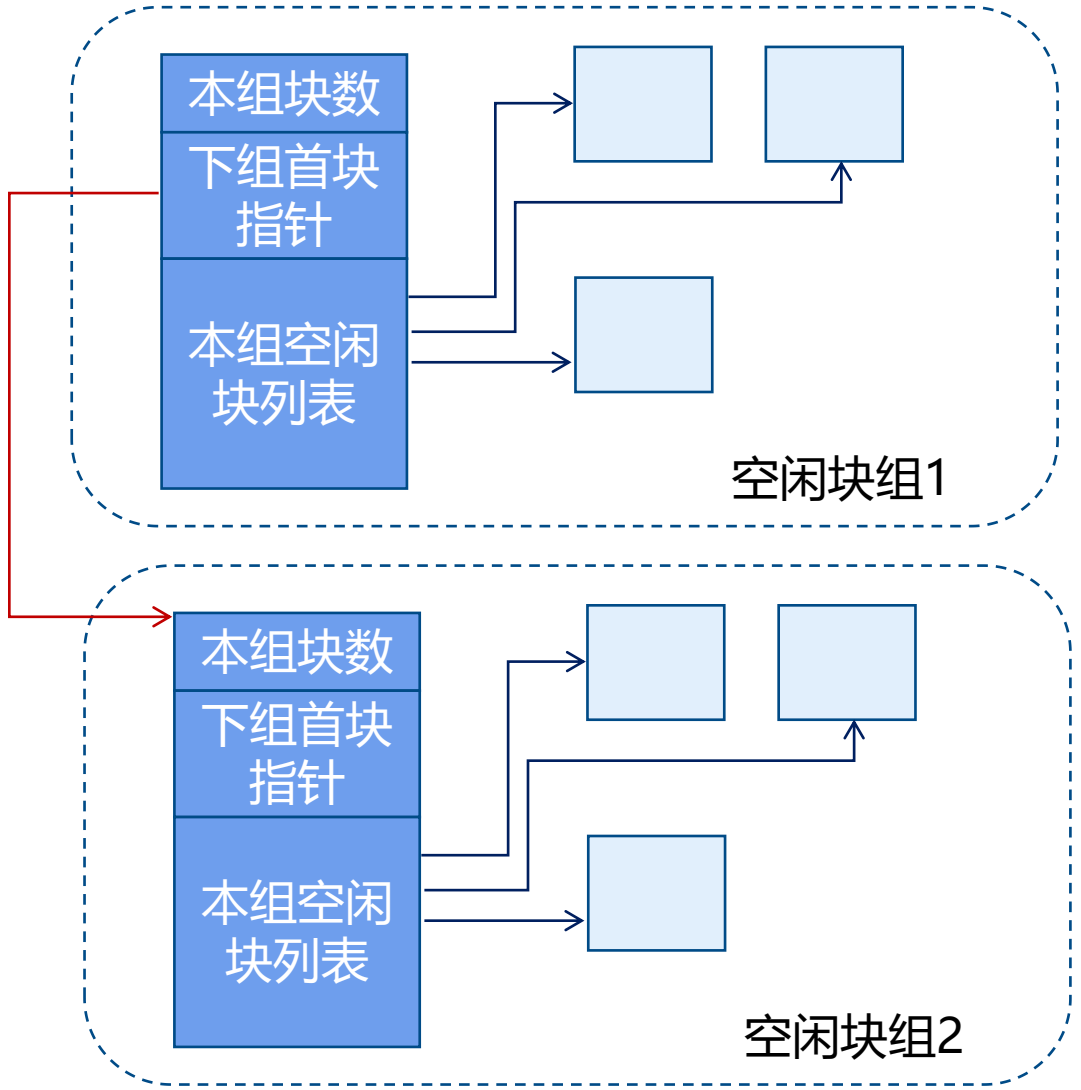

- 空闲位示图区分为两个区域
  - ✓ inode位示图-i位图
  - ✓ 数据块位示图-d位图
- 数据块位示图标注数据块是否空闲
- d位图区每个二进制位（bit）对应一个数据块的状态
  - ✓ 数据位1表示表项空闲
  - ✓ 数据位0表示表项非空闲

二进制位bit

| | Byte0 | Byte1 | ..... |
|---|---|---|---|
| D0 | 0 | | |
| D1 | 0 | | |
| D2 | ..... | | |
| D3 | | | |
| D4 | | | |
| D5 | | | |
| D6 | | | |
| D7 | | | |

# 3. 文件系统实现-成组链接法

- 成组链接法结合空闲表和空闲链表
  - ✓ 以UNIX系统为例
  - ✓ 每100个空闲块为1组
  - ✓ 每组第一个块记录本组和下一组信息
    - ➢ 本组空闲块总数
    - ➢ 下一组的首块地址
    - ➢ 本组空闲块列表
  - ✓ 若下组首块指针为0，标志最后一组
- 空闲块分配
  - ✓ 从首块组开始分配，不够继续用下组
- 空闲块回收
  - ✓ 将空闲块放置在首部，原首块组变成第二个块组，依次类推



# 3. 文件系统实现-例题讲解

某文件系统采用索引结点存放文件的属性和地址信息，块大小为4KB。每个文件索引结点占64B，有11个地址项，其中直接地址项8个，一级、二级和三级间接地址项各1个，每个地址项长度为4B。请回答下列问题。

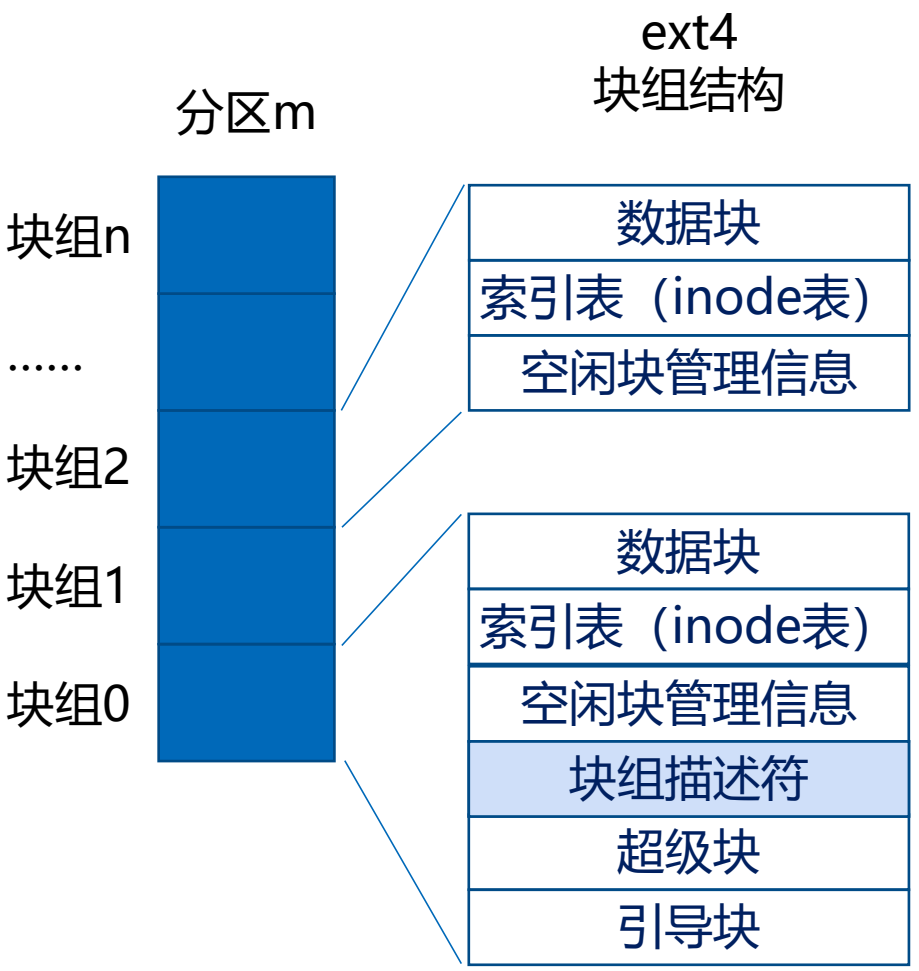
1）该文件系统能支持的最大文件长度是多少？（给出计算表达式即可）

2）文件系统用1M（1M = $2^{20}$）个块存放文件索引结点，用512M个块存放文件数据。若一个图像文件的大小为5600B，则该文件系统最多能存放多少个图像文件?

3）若文件F1的大小为6KB，文件F2的大小为40KB，打开文件后，该文件系统获取F1和F2最后一个块的块号需要的时间是否相同？为什么?

# 3. 文件系统实现-块组描述符GD

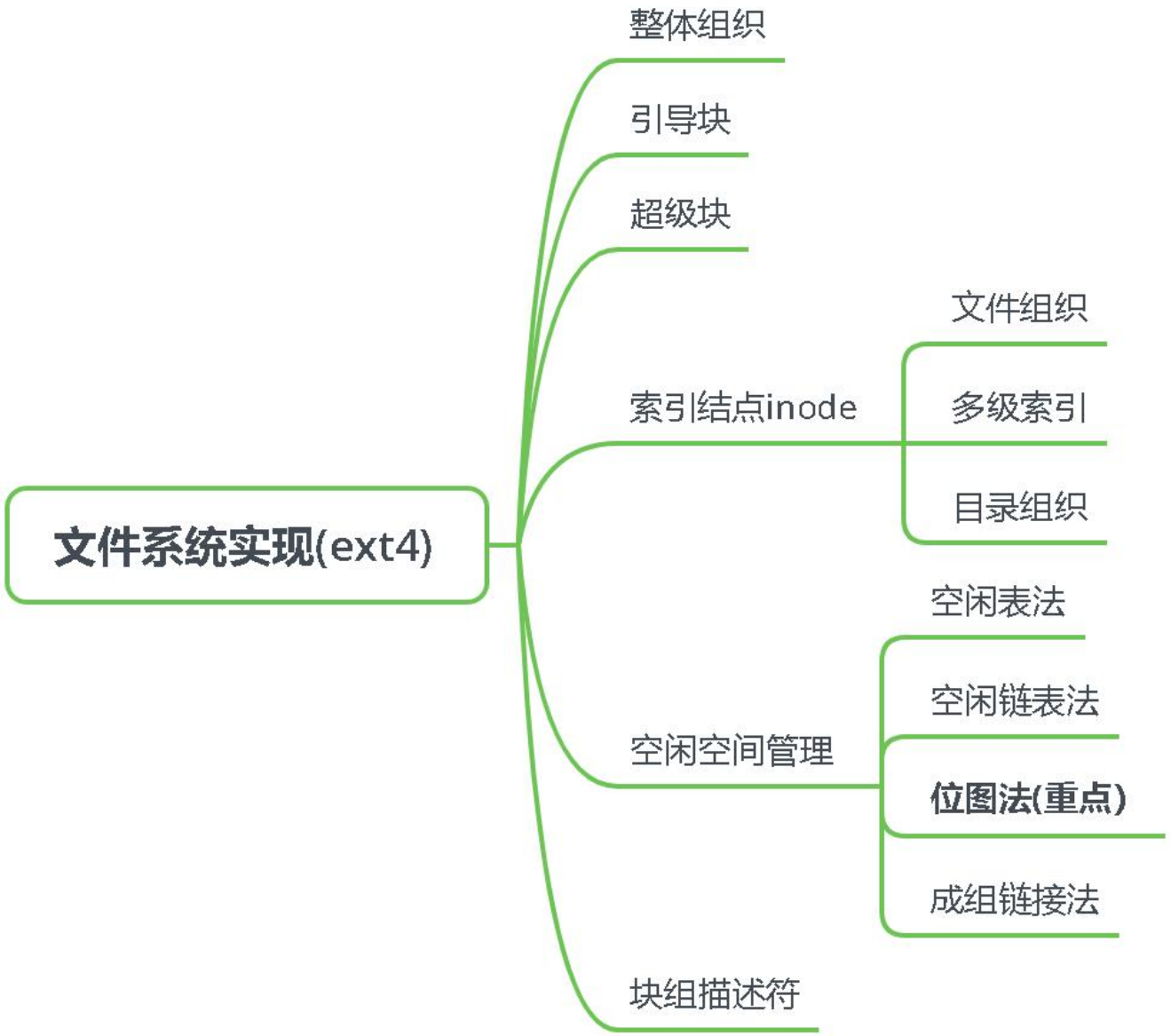
- Ext4文件系统将块分为块组管理
- 块组0的块组描述符记录块组信息
  - ✓ 块组中数据位示图（d位图）地址
  - ✓ 块组中inode位示图（i位图）地址
  - ✓ inode表地址bg_inode_table
  - ✓ 空闲块数据块数量
  - ✓ 可用inode数
  - ✓ 目录数量
  - ✓ ……



# 3. 文件系统实现-块组描述符GD

```
1.  // 源文件：fs/ext4/ext4.h
2.  /*块组描述符的定义 */
3. struct ext4_group_desc {
4.          __le32    bg_block_bitmap_lo;          /* 数据位图的地址（低32位）*/
5.          __le32    bg_inode_bitmap_lo;          /* inode位图的地址（低32位） */
6.          __le32    bg_inode_table_lo;           /* inode表的地址（低32位） */
7.          __le16    bg_free_blocks_count_lo;     /* 可用数据块数（低32位） */
8.          __le16    bg_free_inodes_count_lo;     /* 可用inode数量（低32位） */
9.          __le16    bg_used_dirs_count_lo;       /* 目录数量（低32位） */
10.         __le16    bg_flags;                    /* 块组标志 */
11.         ……
12.         __le16    bg_itable_unused_lo;         /* 未使用inode表数（低32位） */
13.         __le32    bg_block_bitmap_hi;          /* 数据位图的地址（高32位） */
14.         __le32    bg_inode_bitmap_hi;          /* inode位图的地址（高32位） */
15.         __le32    bg_inode_table_hi;           /* inode表的地址（高32位） */
16.         __le16    bg_free_blocks_count_hi;     /* 可用数据块数（高32位） */
17.         __le16    bg_free_inodes_count_hi;     /* 可用inode数量（高32位） */
18.         __le16    bg_used_dirs_count_hi;       /* 目录数量（高32位）*/
19.         __le16    bg_itable_unused_hi;         /* 未使用inode表数（高32位） */
20.  }
```

# 3. 文件系统实现-小结


文件系统实现(ext4)
整体组织
引导块
超级块
索引结点inode
文件组织
多级索引
目录组织
空闲空间管理
空闲表法
空闲链表法
位图法(重点)
成组链接法
块组描述符

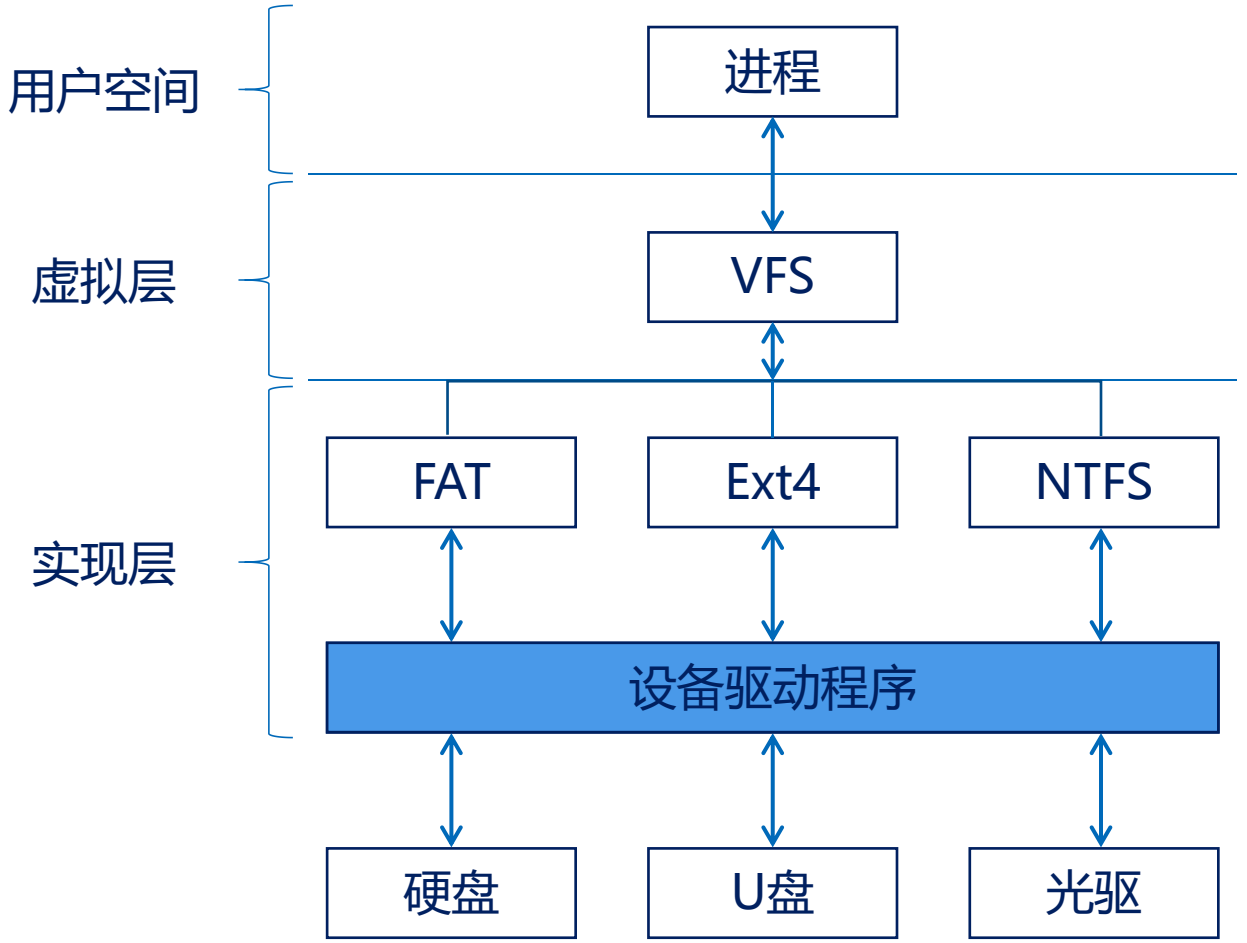

# 目 录

CONTENTS

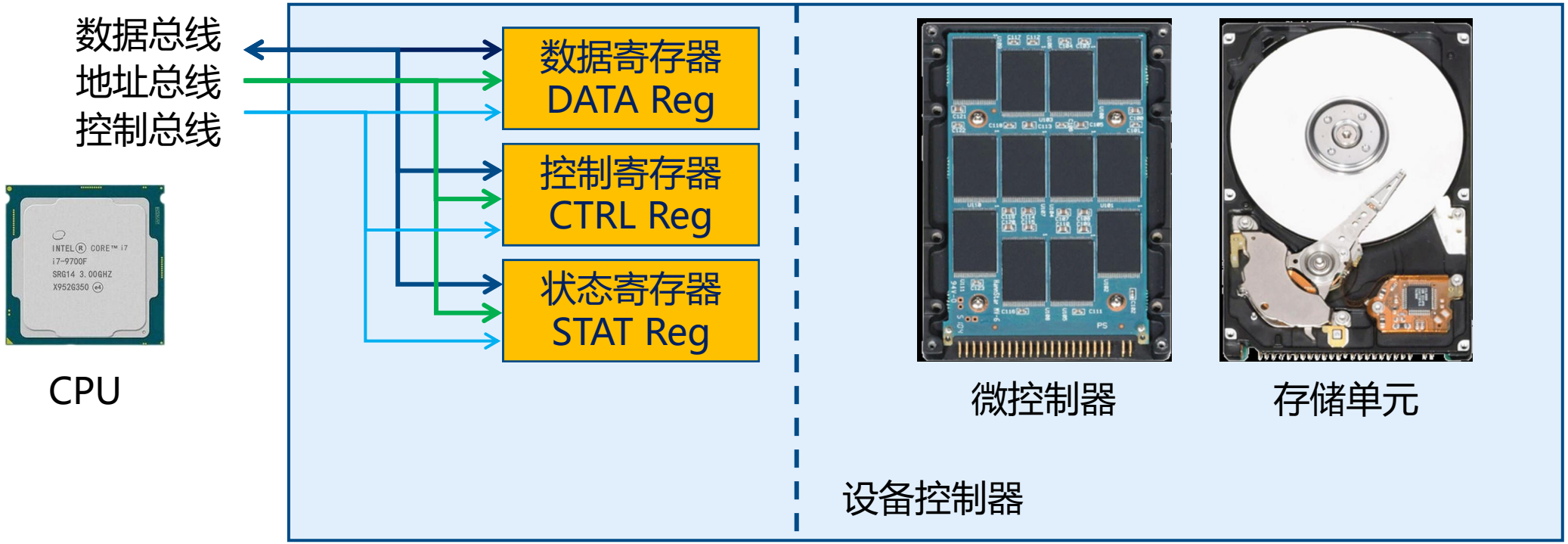
## 第十一章 文件系统实现

# 4. 存储设备访问-存储设备接口

- 设备控制器位于CPU与磁盘之间
- 设备控制器与CPU通信
  - ✓ 接收CPU发出的磁盘访问指令
  - ✓ 提供给CPU查询状态的接口
  - ✓ 接收写入磁盘的数据并缓存
  - ✓ 从缓存读取并发送从磁盘读取的数据
- 设备控制器与磁盘通信
  - ✓ 控制磁盘执行读写操作
  - ✓ 写入时，将缓存的数据发送给磁盘
  - ✓ 读取时，将磁盘的数据读入缓存



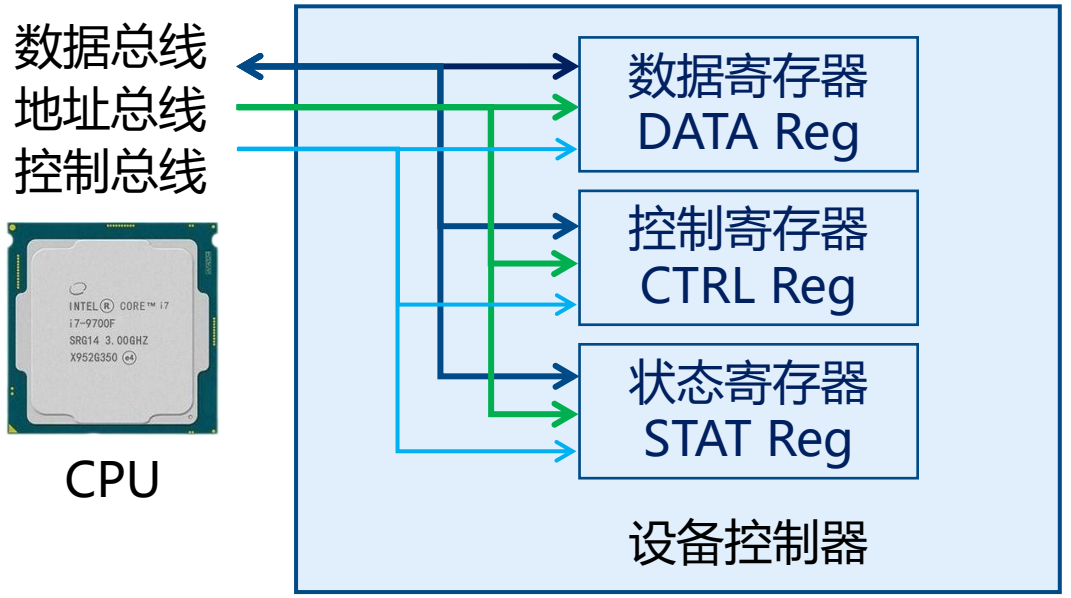
# 4. 存储设备访问-存储设备接口

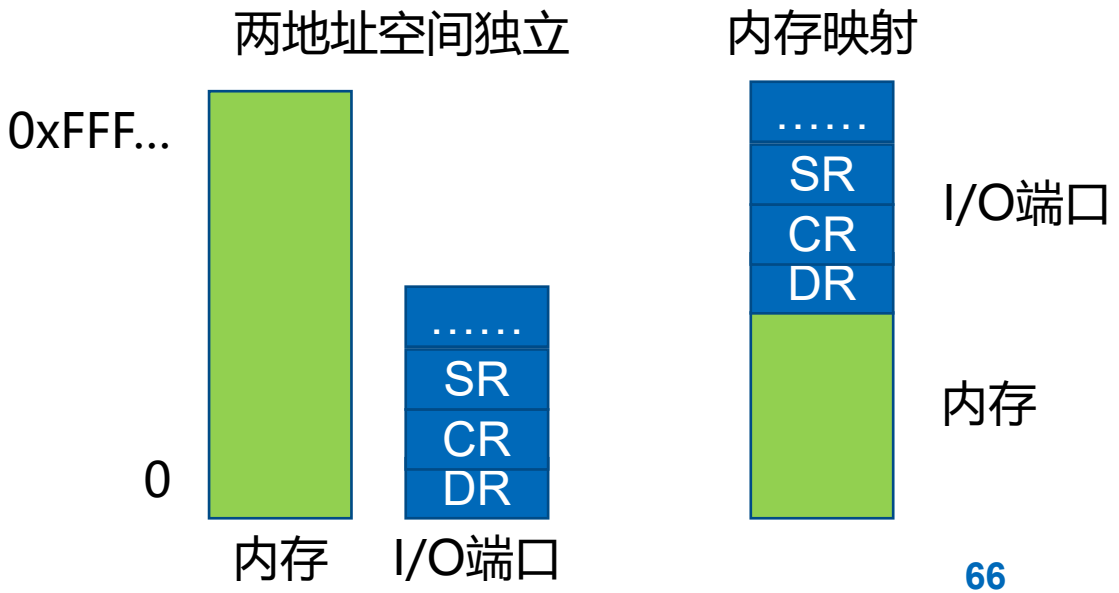



- 设备控制器对CPU提供**数据寄存器**、**控制寄存器**、**状态寄存器**三个接口
  - ✓ CPU通过数据总线、地址总线和控制总线访问三个接口寄存器
  - ✓ 数据寄存器：存放设备来的数据/CPU发送来的数据
  - ✓ 控制寄存器：存放CPU发送的控制信息，供设备读取
  - ✓ 状态寄存器：存放设备的工作状态信息，供CPU读取

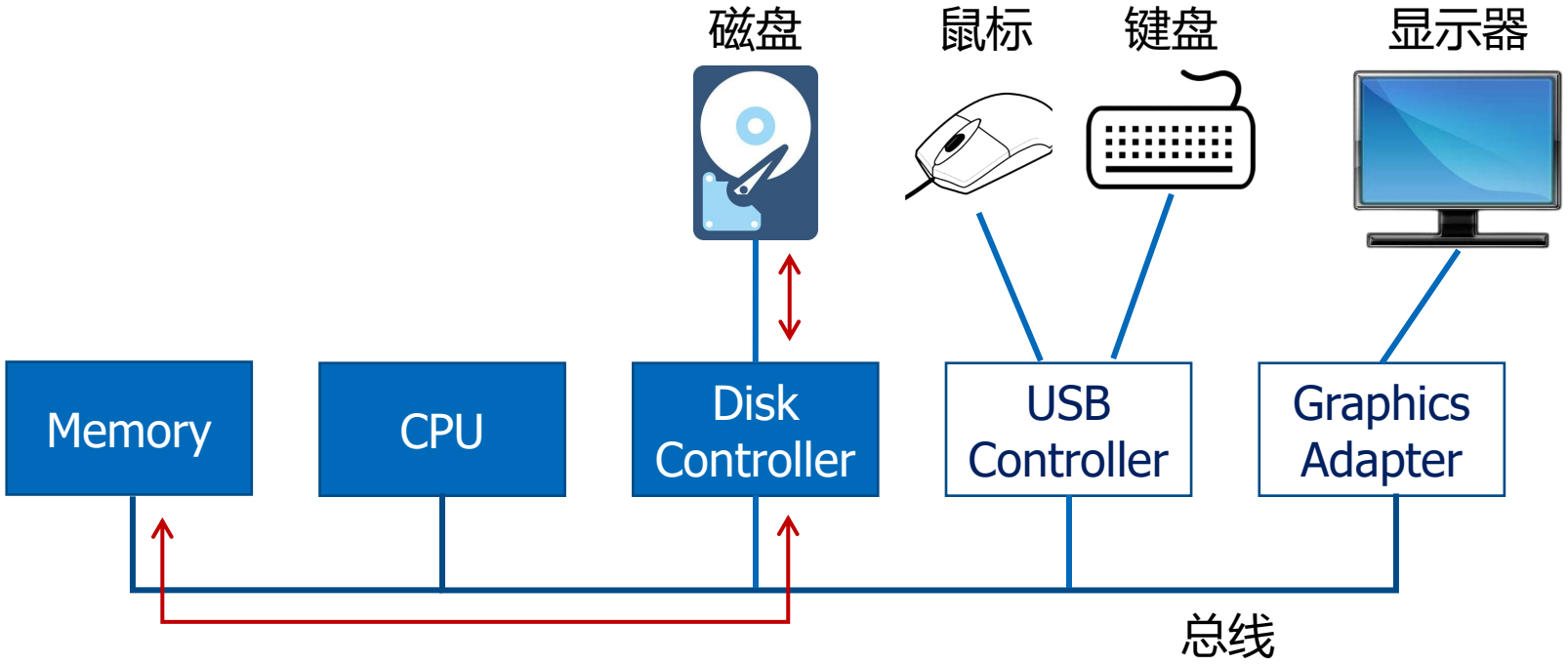
# 4. 存储设备访问-存储设备接口

- CPU通过I/O端口对3个寄存器访问
  - ✓ **数据寄存器**：实现CPU和外设数据缓冲
  - ✓ **控制寄存器**：启动命令或更改设备模式
  - ✓ **状态寄存器**：获取执行结果和设备状态信息
- CPU与I/O端口有两种通信方式：
  - ✓ 独立编址：
    - ➢ 为I/O端口分配独立端口号
    - ➢ 形成独立I/O端口空间
    - ➢ 普通用户无法访问
    - ➢ 操作系统使用特殊I/O指令访问
  - ✓ 统一编址：
    - ➢ 内存映射I/O，为I/O端口分配内存地址
    - ➢ 该地址I/O专用不会放置内存单元



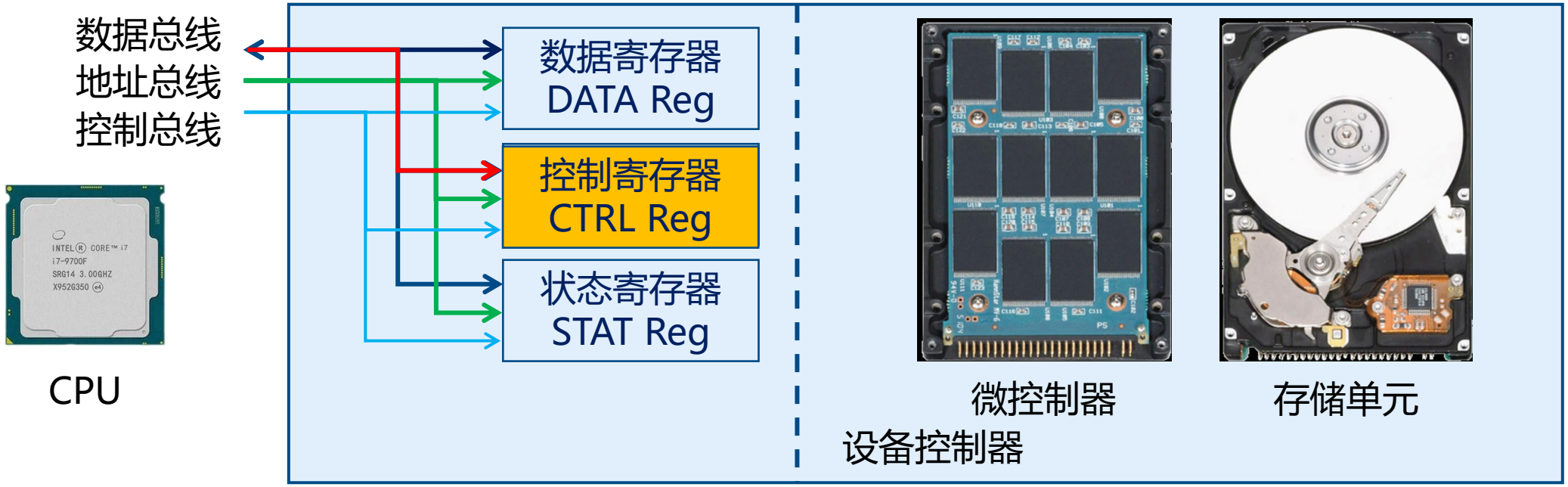



# 4. 存储设备访问-设备访问方法



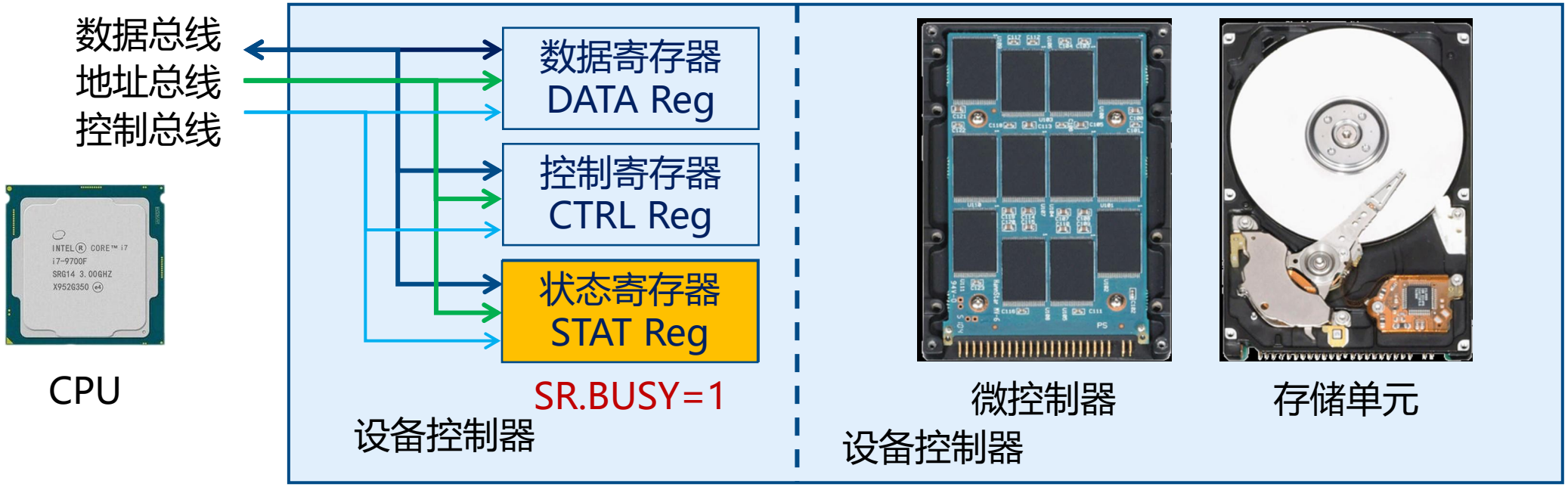

- 设备管理主要任务控制设备和内存/CPU之间的数据传输
- 设备和内存之间输入/输出控制方式有4种
  - ✓ 程序控制方式（ Programming Input/Output Model，PIO）
  - ✓ 中断驱动方式（Interrupt Driven Model）
  - ✓ 直接存储器存取（Direct Memory Access，DMA）
  - ✓ 通道控制方式

# 4. 存储设备访问-程序控制方式



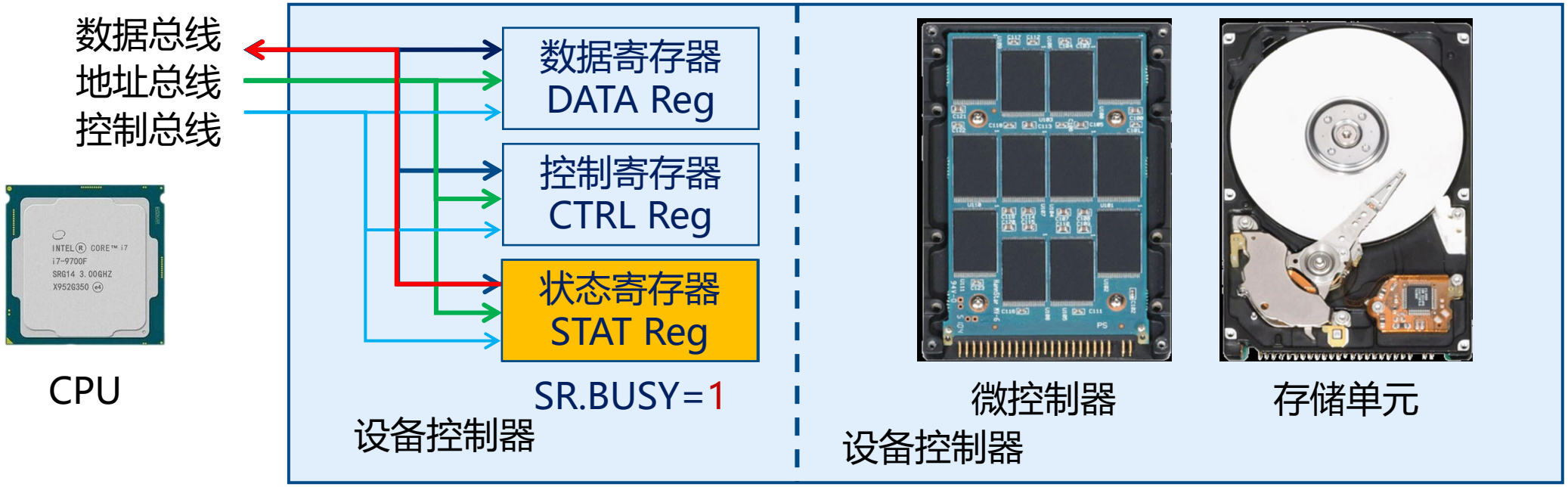

1. CPU向控制寄存器发出读指令，即写入控制指令字

# 4. 存储设备访问-程序控制方式



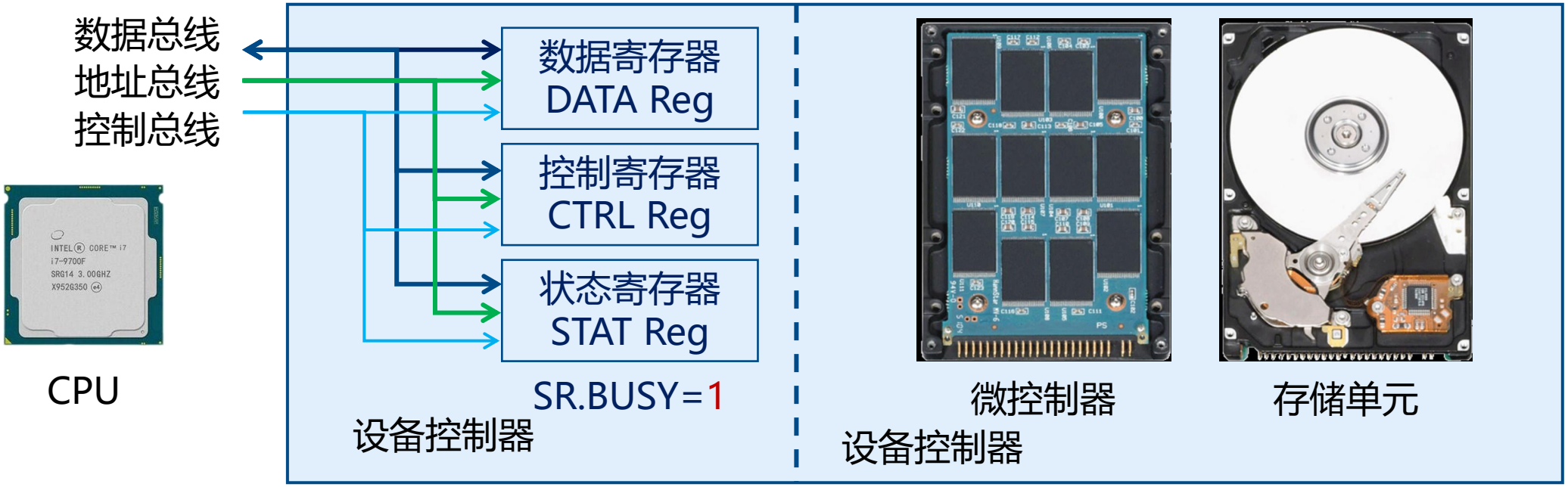

1. CPU向控制寄存器发出读指令，即写入控制指令字
2. 设备控制器修改状态寄存器忙状态，由0变成1

# 4. 存储设备访问-程序控制方式



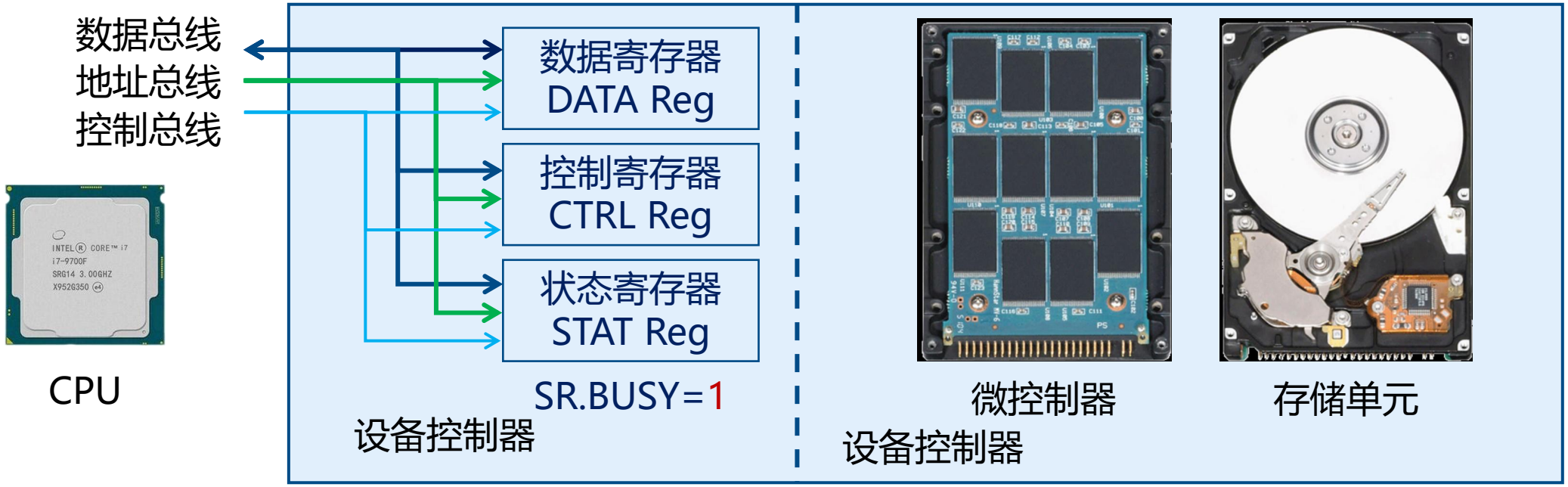

1. CPU向控制寄存器发出读指令，即写入控制指令字
2. 设备控制器修改状态寄存器忙状态，由0变成1
3. CPU通过状态寄存器读取SR.BUSY值，如果是1，则设备忙，CPU重复读取

# 4. 存储设备访问-程序控制方式



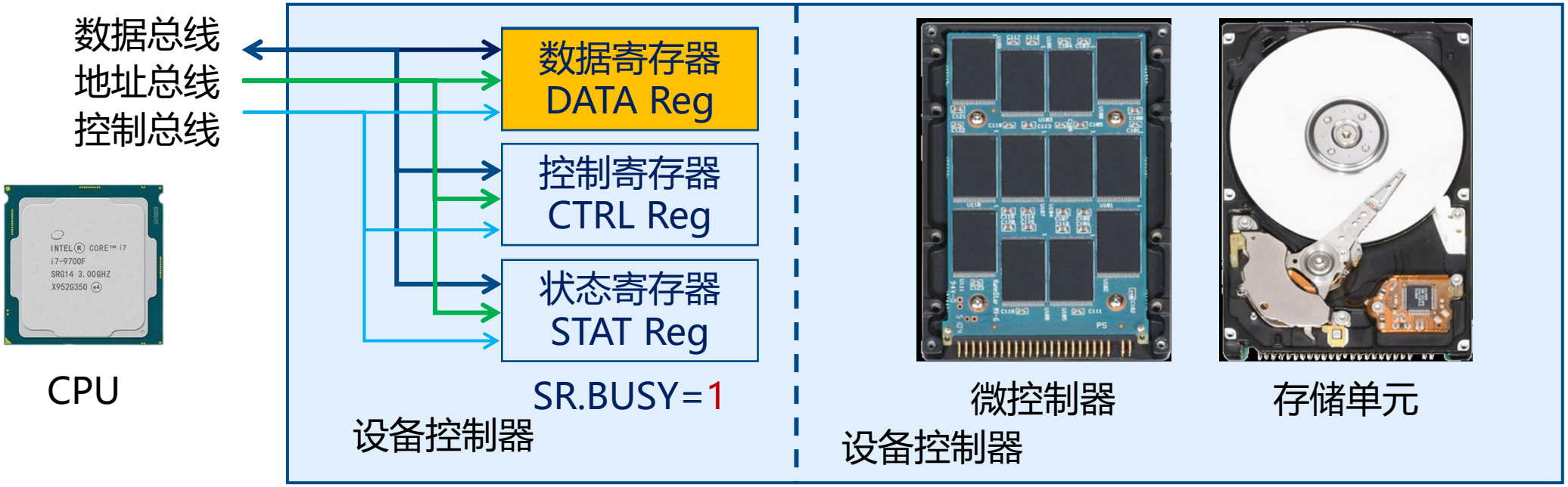

1. CPU向控制寄存器发出读指令，即写入控制指令字
2. 设备控制器修改状态寄存器忙状态，由0变成1
3. CPU通过状态寄存器读取SR.BUSY值，如果是1，则设备忙，CPU重复读取
4. 查询硬盘设备的状态

# 4. 存储设备访问-程序控制方式



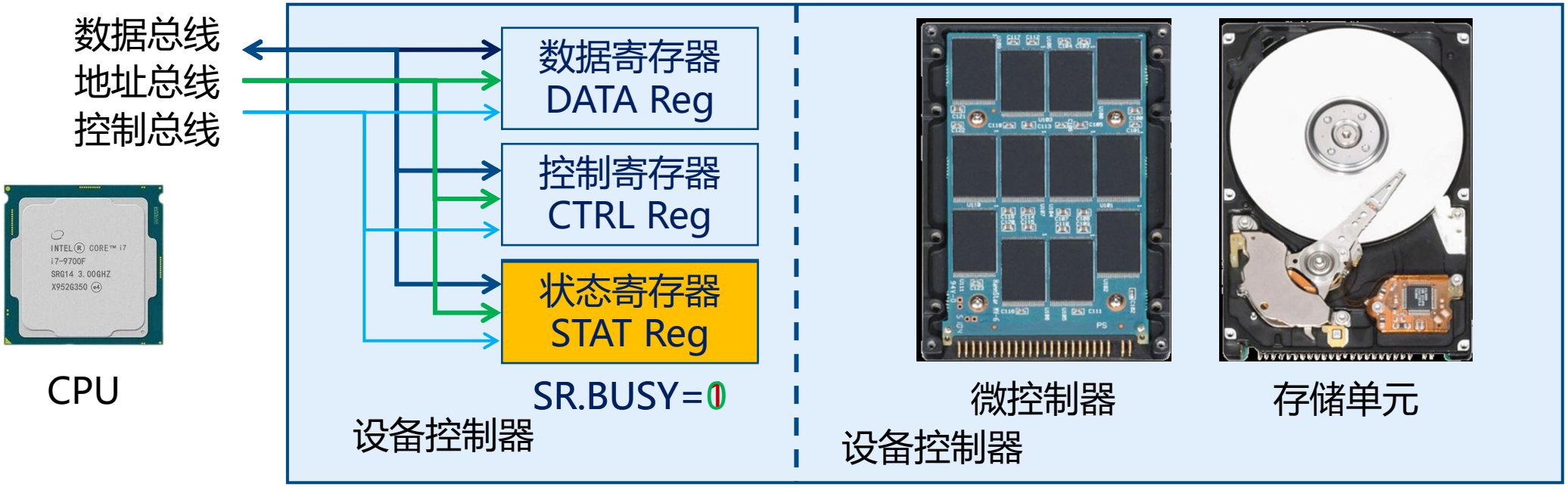

1. CPU向控制寄存器发出读指令，即写入控制指令字
2. 设备控制器修改状态寄存器忙状态，由0变成1
3. CPU通过状态寄存器读取SR.BUSY值，如果是1，则设备忙，CPU重复读取
4. 硬盘准备好数据，将数据传送给控制器IO逻辑，并报告硬盘状态

# 4. 存储设备访问-程序控制方式



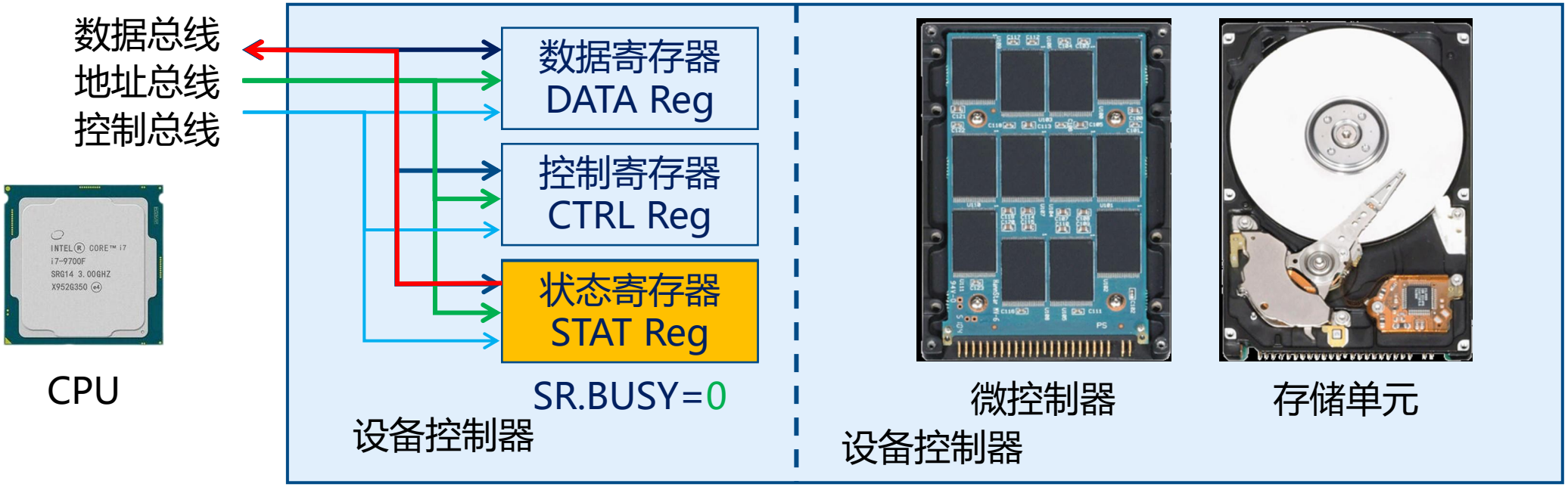

1. CPU向控制寄存器发出读指令，即写入控制指令字
2. 设备控制器修改状态寄存器忙状态，由0变成1
3. CPU通过状态寄存器读取SR.BUSY值，如果是1，则设备忙，CPU重复读取
4. 硬盘的数据放到数据寄存器

# 4. 存储设备访问-程序控制方式



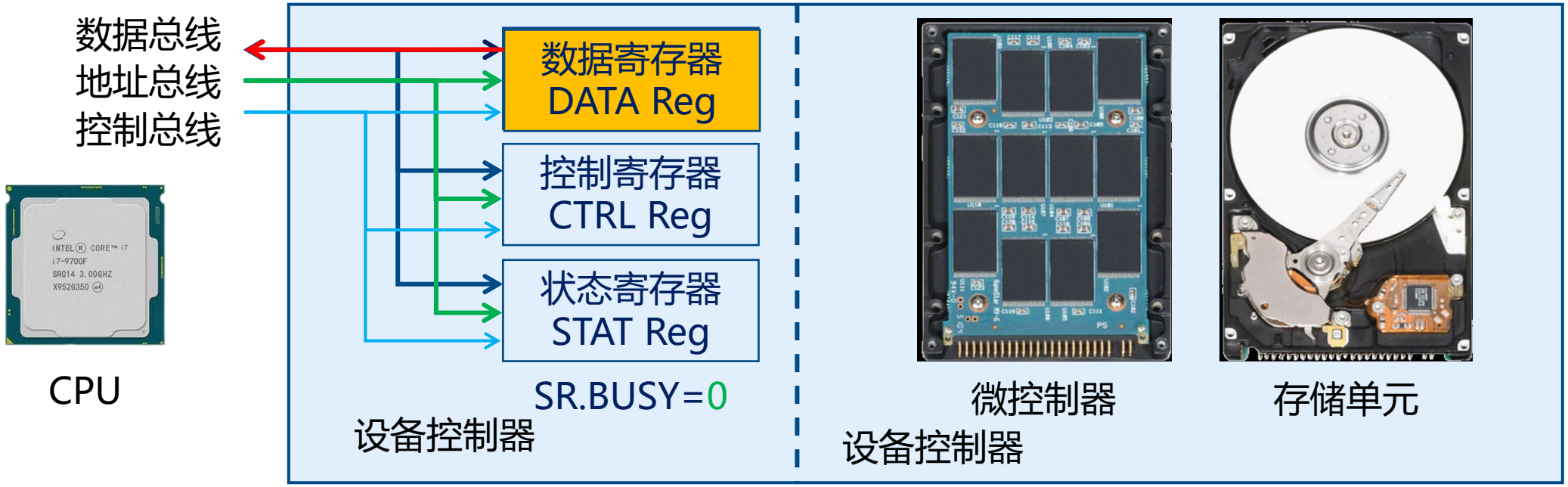

1. CPU向控制寄存器发出读指令，即写入控制指令字
2. 设备控制器修改状态寄存器忙状态，由0变成1
3. CPU通过状态寄存器读取SR.BUSY值，如果是1，则设备忙，CPU重复读取
4. 硬盘的数据放到数据寄存器
5. 修改状态寄存器的SR.BUSY值为0

# 4. 存储设备访问-程序控制方式




1. CPU向控制寄存器发出读指令，即写入控制指令字
2. 设备控制器修改状态寄存器忙状态，由0变成1
3. CPU通过状态寄存器读取SR.BUSY值，如果是1，则设备忙，CPU重复读取
4. 硬盘的数据放到数据寄存器
5. 修改状态寄存器的SR.BUSY值为0
6. CPU查询状态寄存器，发现SR.BUSY变为0的事件

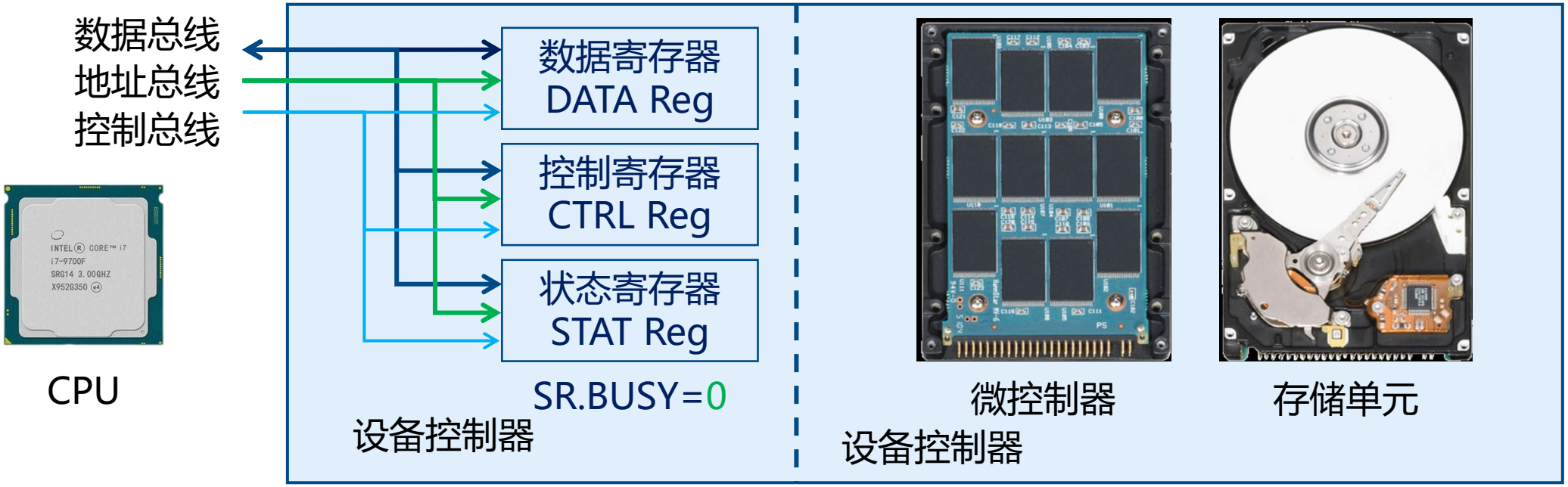
# 4. 存储设备访问-程序控制方式




6. CPU查询状态寄存器，发现SR.BUSY变为0的事件

7. CPU将数据寄存器的内容读入CPU内部的寄存器

# 4. 存储设备访问-程序控制方式




6. CPU查询状态寄存器，发现SR.BUSY变为0的事件

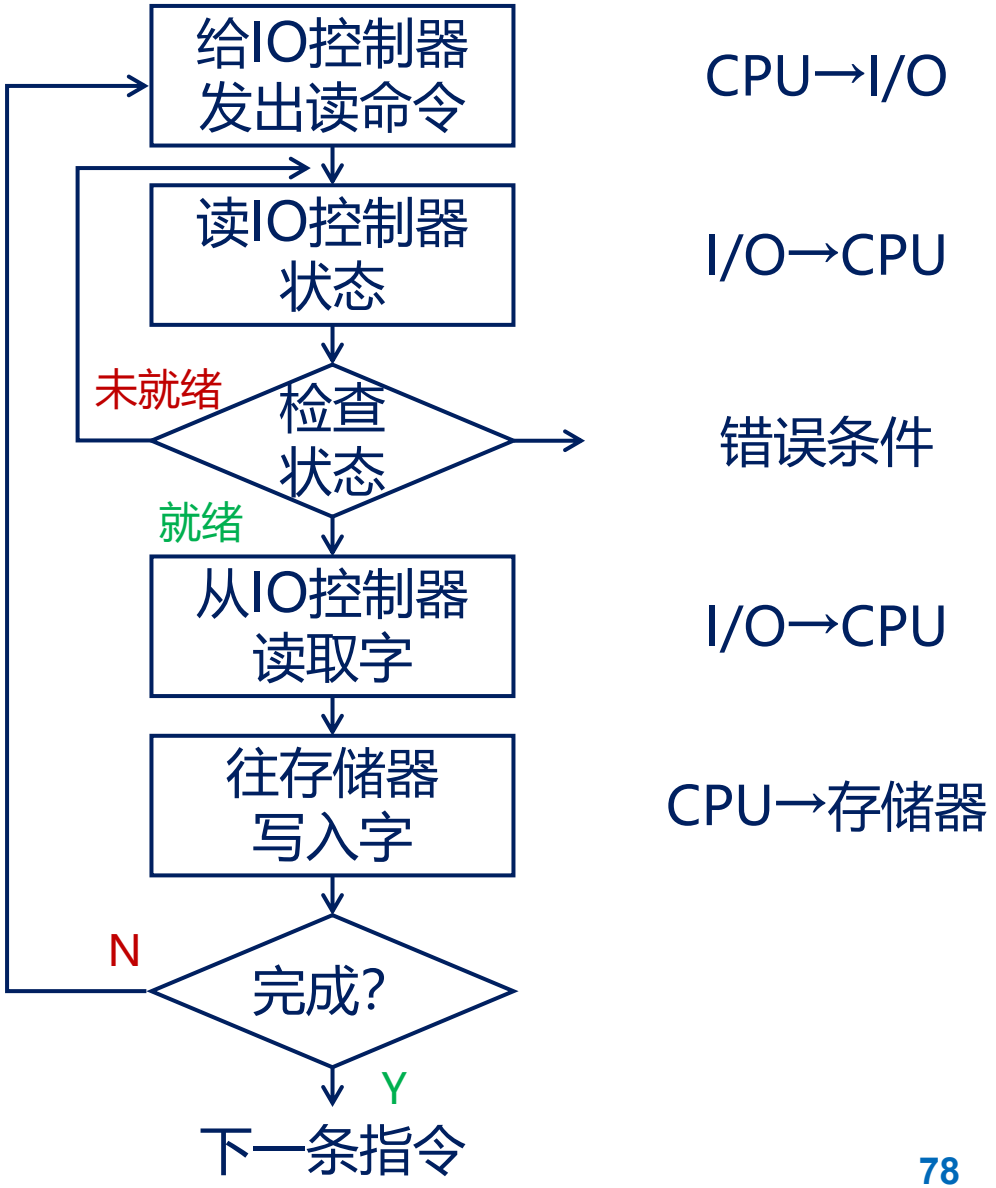
7. CPU将数据寄存器的内容读入CPU内部的寄存器

8.CPU将内部寄存器的内容写入内存

9.若CPU还要读取数据，CPU继续发出指令

# 4. 存储设备访问-程序控制方式

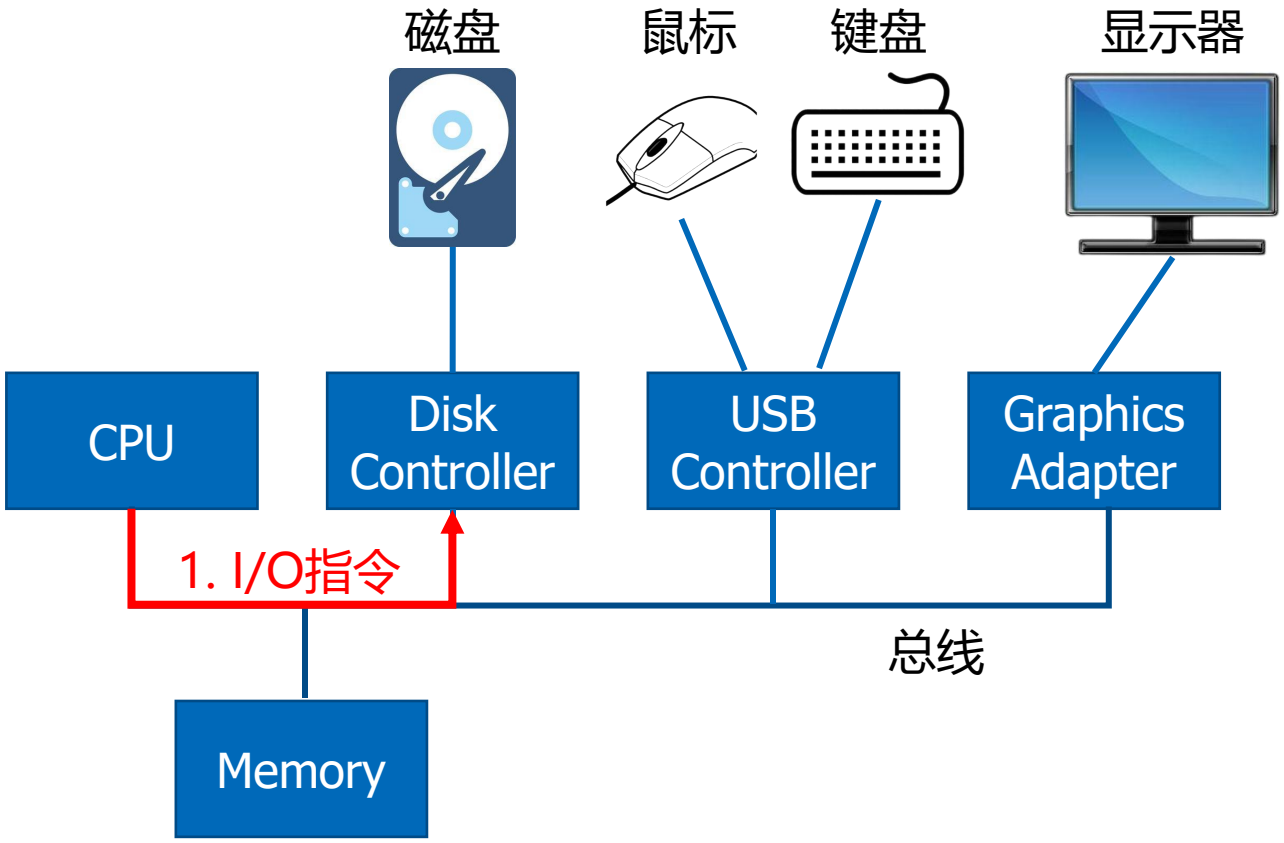
- 程序控制方式工作流程
  - ✓ CPU向控制器发出命令
  - ✓ 将I/O状态读入CPU内部寄存器
  - ✓ 检查设备状态
    - ➢ 就绪继续下一步操作
    - ➢ 未就绪返回重读I/O状态
    - ➢ 错误则进行错误处理并结束
  - ✓ I/O数据寄存器内容读入CPU内部寄存器
  - ✓ 将CPU寄存器的内容写入内存中
  - ✓ 如果完成，向后执行；若读取新内容则重复
- 优点：实现简单
- 缺点：CPU和I/O设备串行工作，CPU大量查询状态，多数处于忙等，CPU利用率低



# 4. 存储设备访问-中断驱动方式

1. CPU向磁盘控制器发出I/O指令

磁盘中断驱动方式读写过程

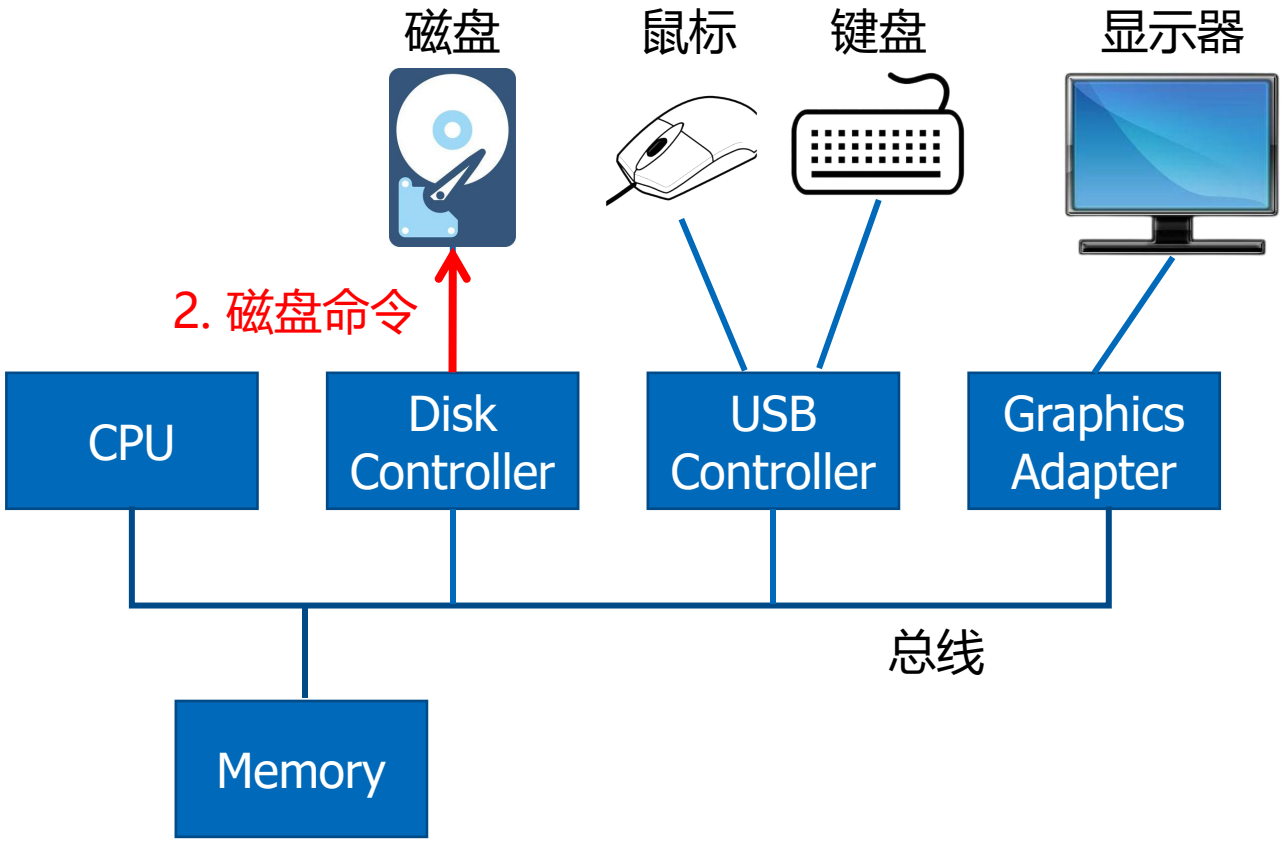


# 4. 存储设备访问-中断驱动方式

1. CPU向磁盘控制器发出I/O指令

2. 磁盘控制器向磁盘发出控制指令

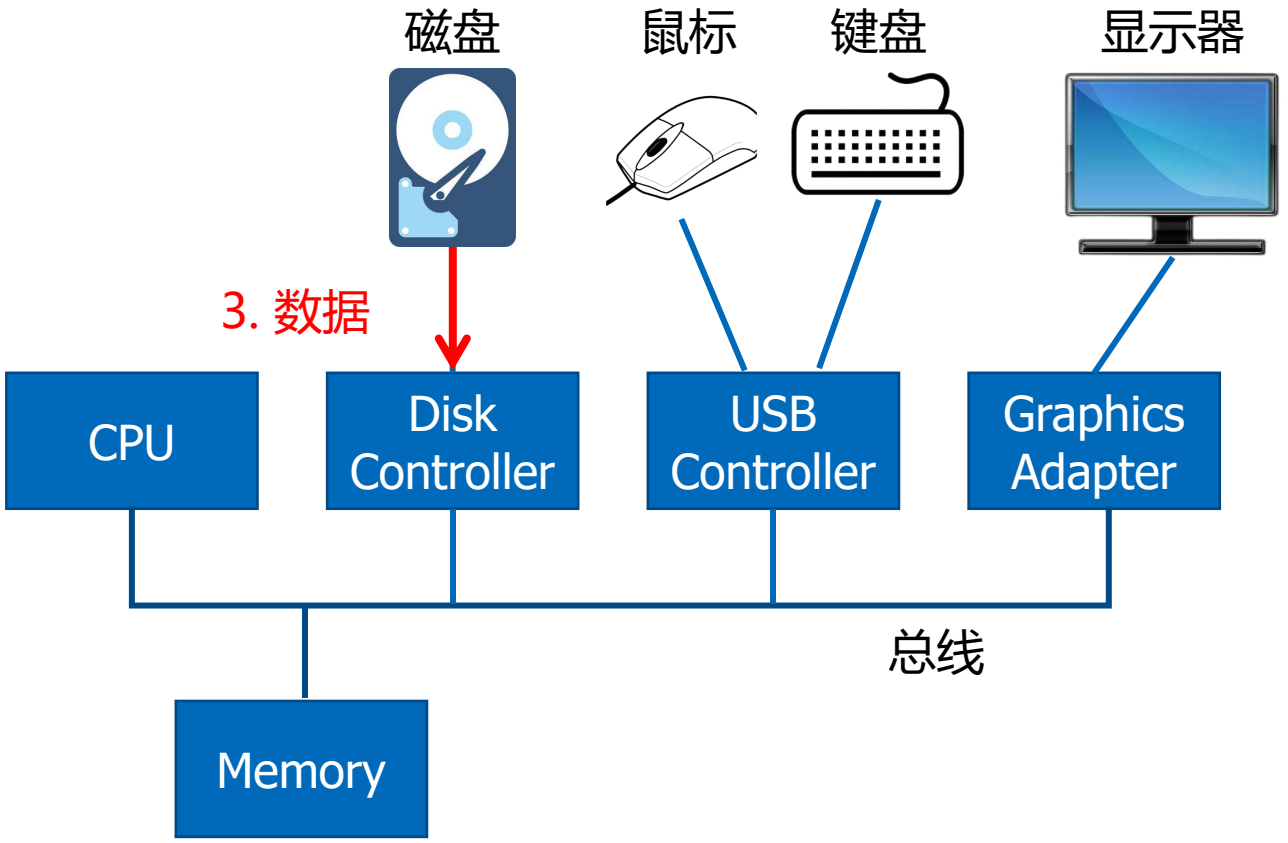
磁盘中断驱动方式读写过程



# 4. 存储设备访问-中断驱动方式

1. CPU向磁盘控制器发出I/O指令
2. 磁盘控制器向磁盘发出控制指令
3. 磁盘将数据写入磁盘控制器的缓存中

磁盘中断驱动方式读写过程

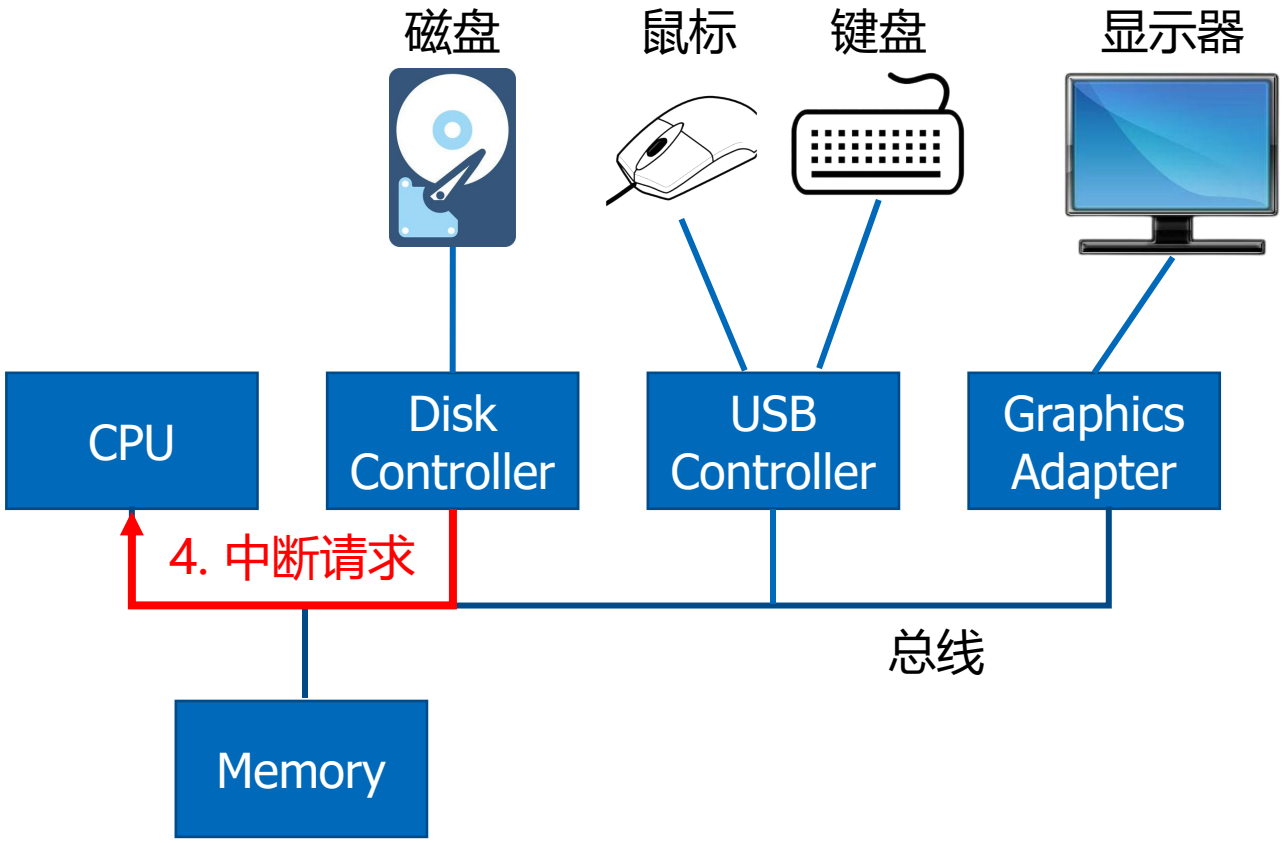


# 4. 存储设备访问-中断驱动方式



1. CPU向磁盘控制器发出I/O指令
2. 磁盘控制器向磁盘发出控制指令
3. 磁盘将数据写入磁盘控制器的缓存中
4. 磁盘控制器发出中断提醒CPU

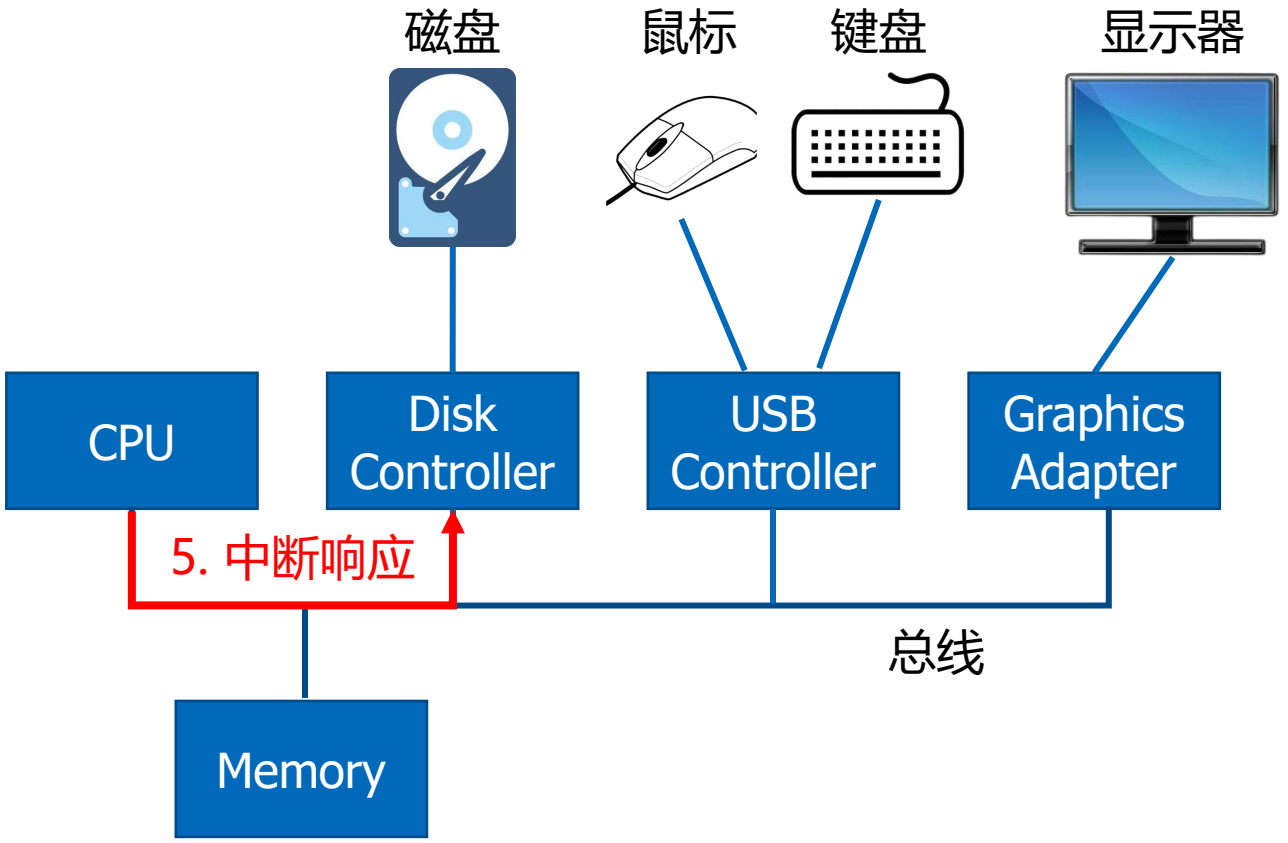
磁盘中断驱动方式读写过程



# 4. 存储设备访问-中断驱动方式

1. CPU向磁盘控制器发出I/O指令
2. 磁盘控制器向磁盘发出控制指令
3. 磁盘将数据写入磁盘控制器的缓存中
4. 磁盘控制器发出中断提醒CPU
5. CPU响应中断向硬盘控制器发出命令

磁盘中断驱动方式读写过程

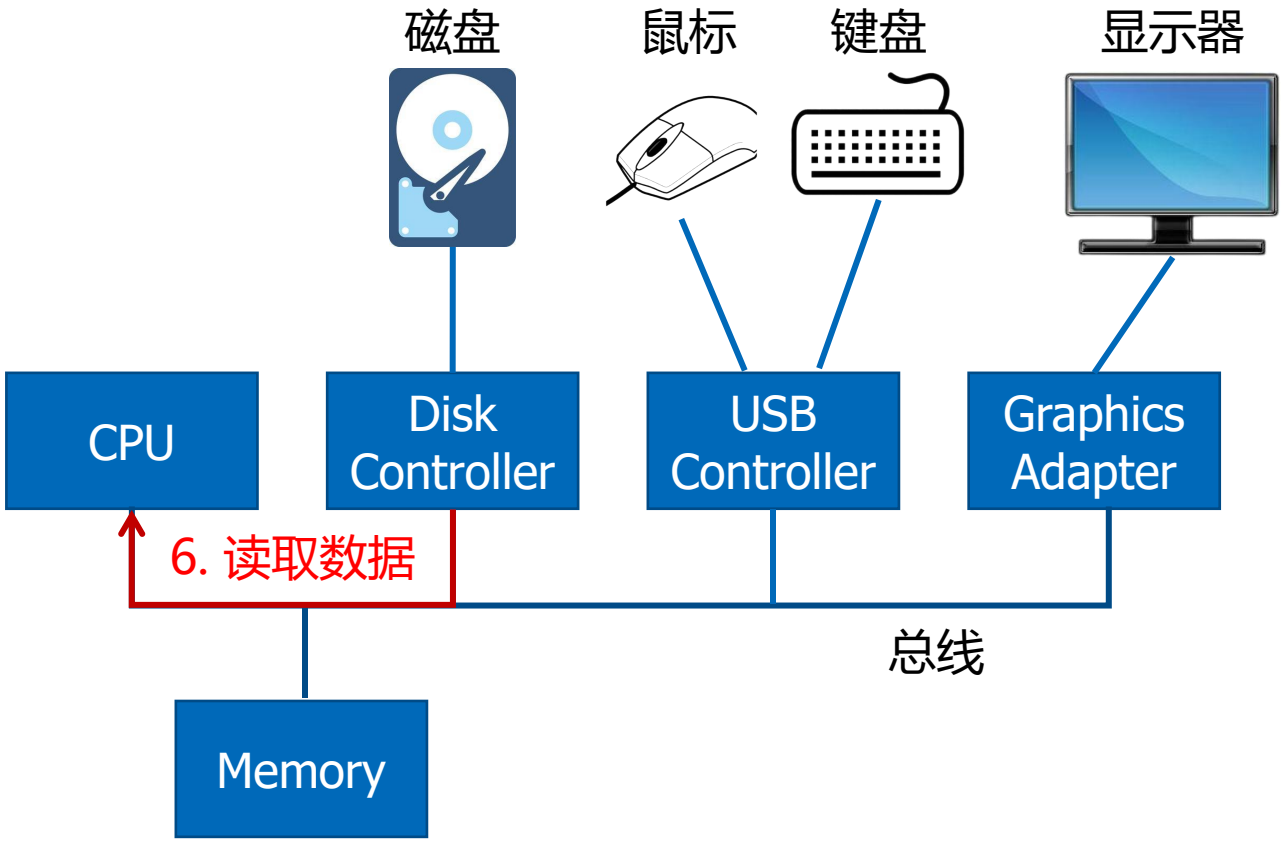


# 4. 存储设备访问-中断驱动方式



1. CPU向磁盘控制器发出I/O指令
2. 磁盘控制器向磁盘发出控制指令
3. 磁盘将数据写入磁盘控制器的缓存中
4. 磁盘控制器发出中断提醒CPU
5. CPU响应中断向硬盘控制器发出命令
6. CPU将磁盘控制器数据读入寄存器

磁盘中断驱动方式读写过程

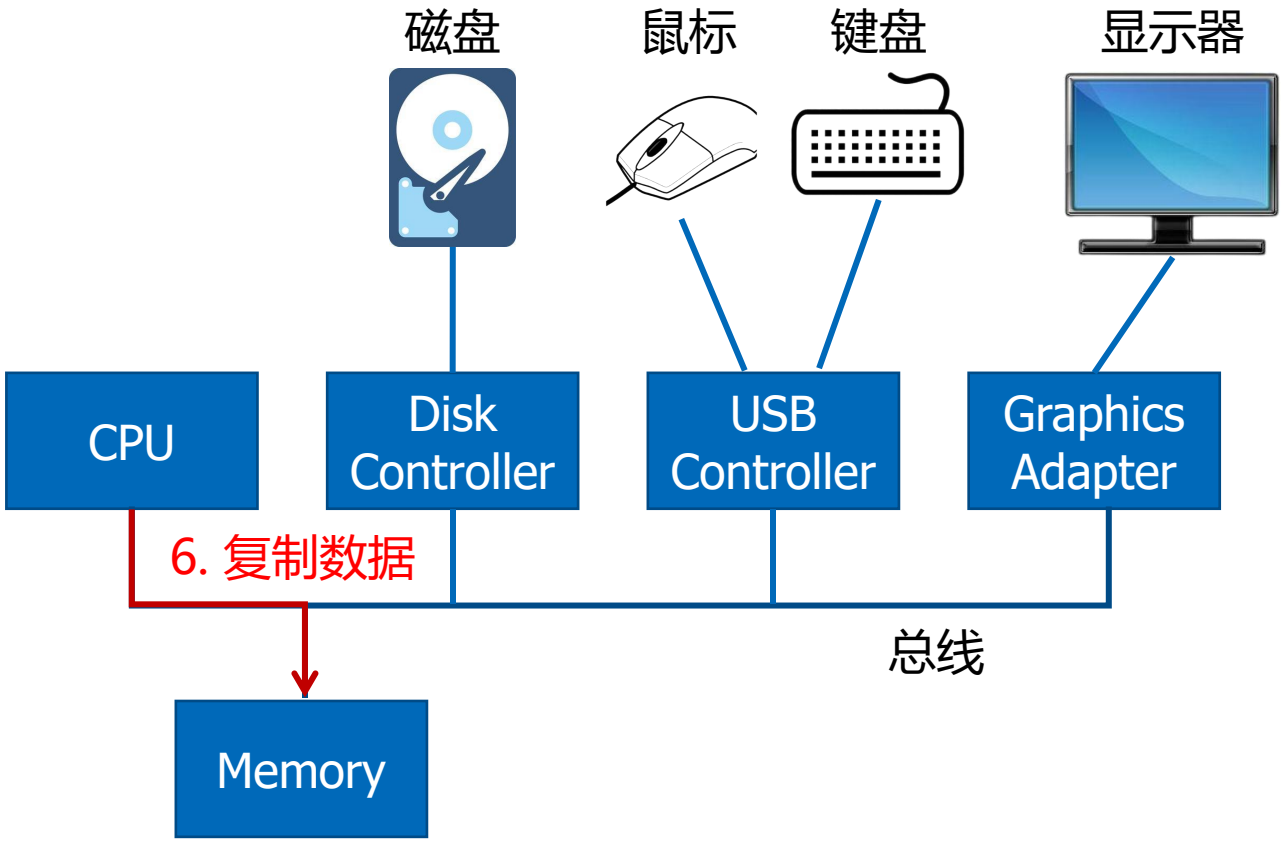


# 4. 存储设备访问-中断驱动方式



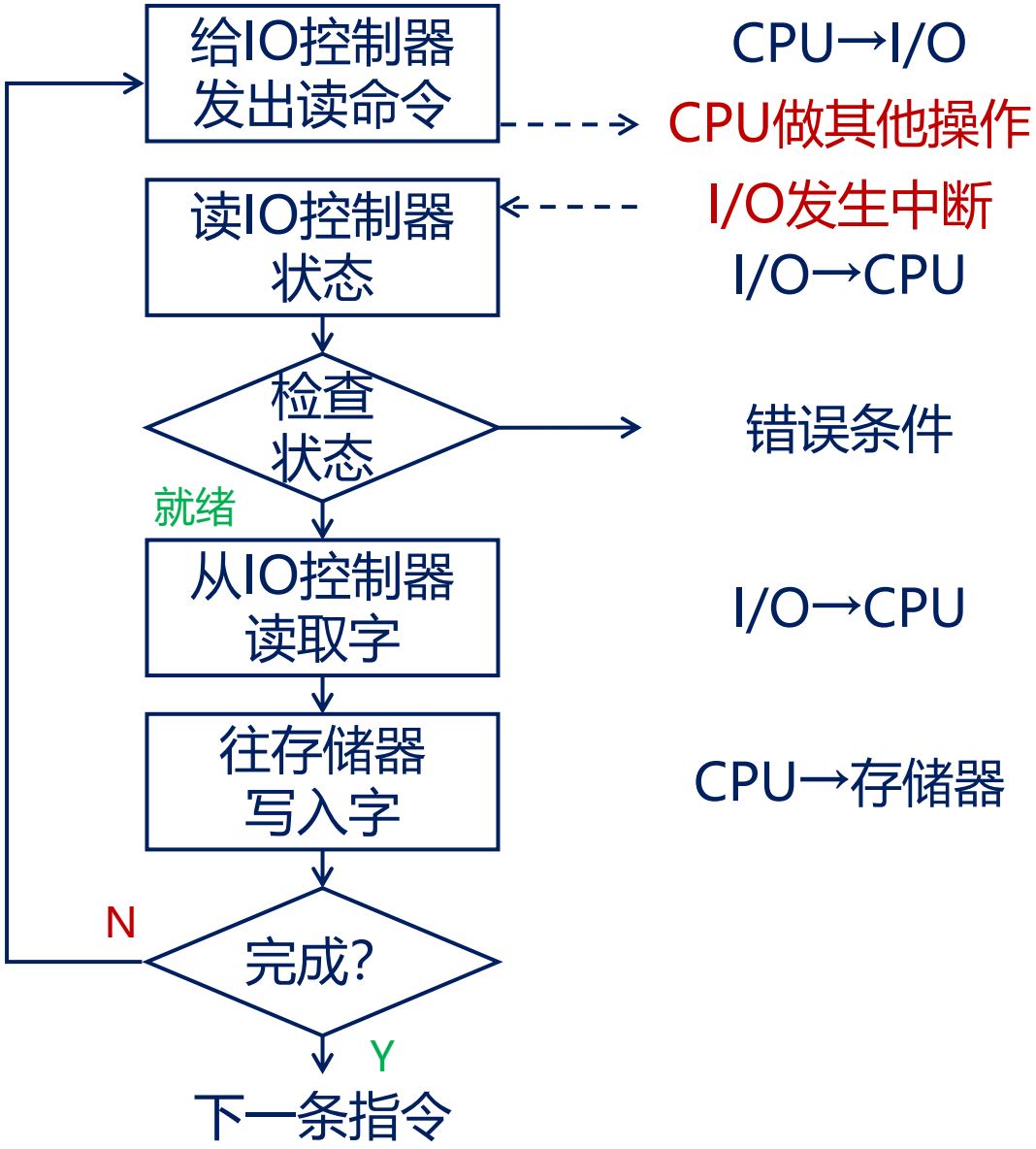
1. CPU向磁盘控制器发出I/O指令
2. 磁盘控制器向磁盘发出控制指令
3. 磁盘将数据写入磁盘控制器的缓存中
4. 磁盘控制器发出中断提醒CPU
5. CPU响应中断向硬盘控制器发出命令
6. CPU将磁盘控制器数据读入寄存器
7. CPU将寄存器内容写入主存

磁盘中断驱动方式读写过程



# 4. 存储设备访问-中断驱动方式

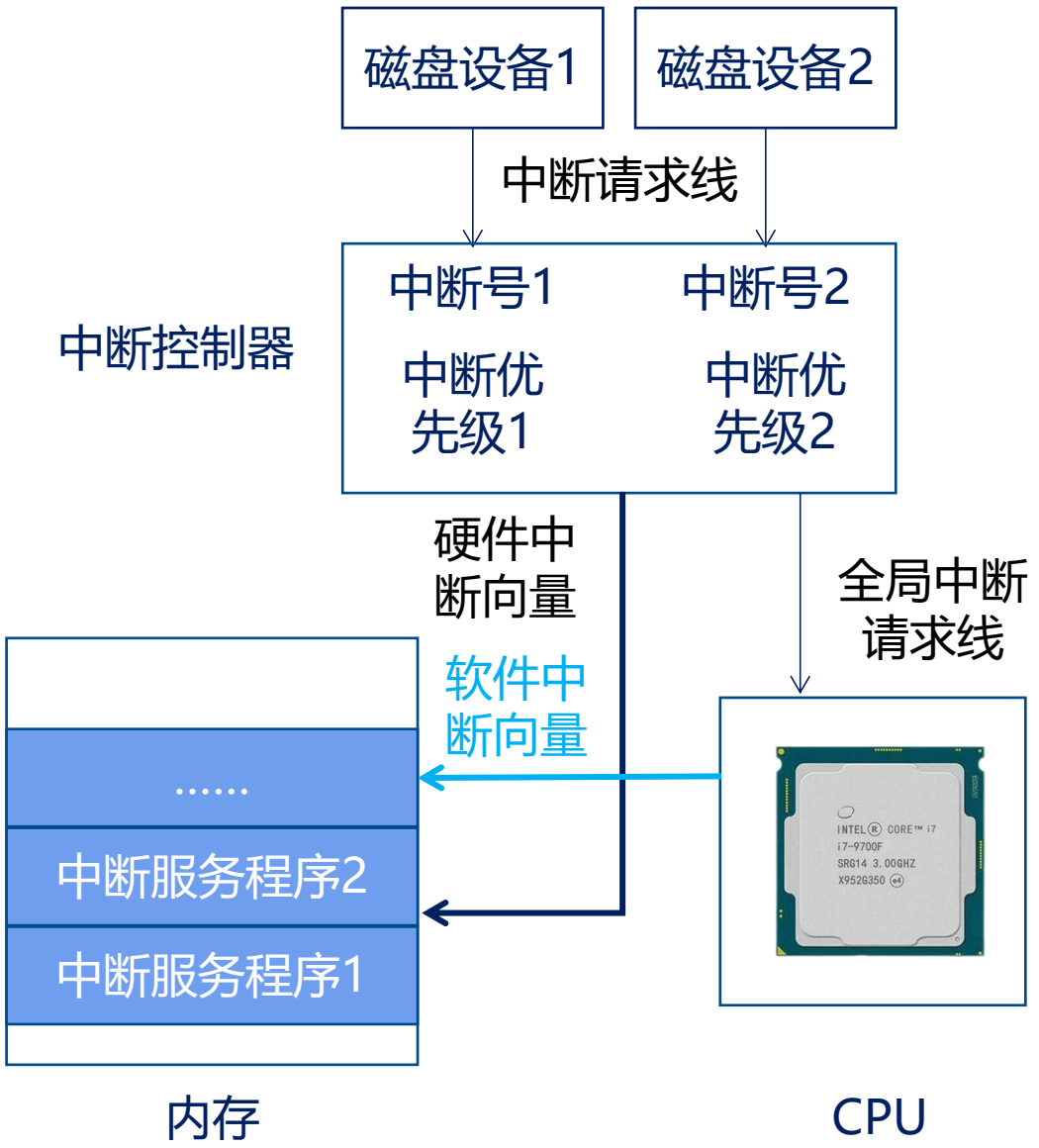
- 中断驱动方式工作流程
  - ✓ CPU向控制器发出命令，之后CPU继续其他操作
  - ✓ I/O发出中断，CPU响应中断，读取I/O状态
  - ✓ 检查设备状态
    - ➢ 就绪继续下一步操作
    - ➢ 错误则进行错误处理并结束
  - ✓ I/O数据寄存器内容读入CPU内部寄存器
  - ✓ 将CPU寄存器的内容写入内存中
  - ✓ 如果完成，向后执行；若读取新内容则重复
- 优点：CPU不需轮询，CPU和I/O设备并行工作，效率较高
- 缺点：每个数据需要经过CPU中转才能从I/O设备存储到内存，频繁中断处理消耗CPU时间



# 4. 存储设备访问-中断驱动方式



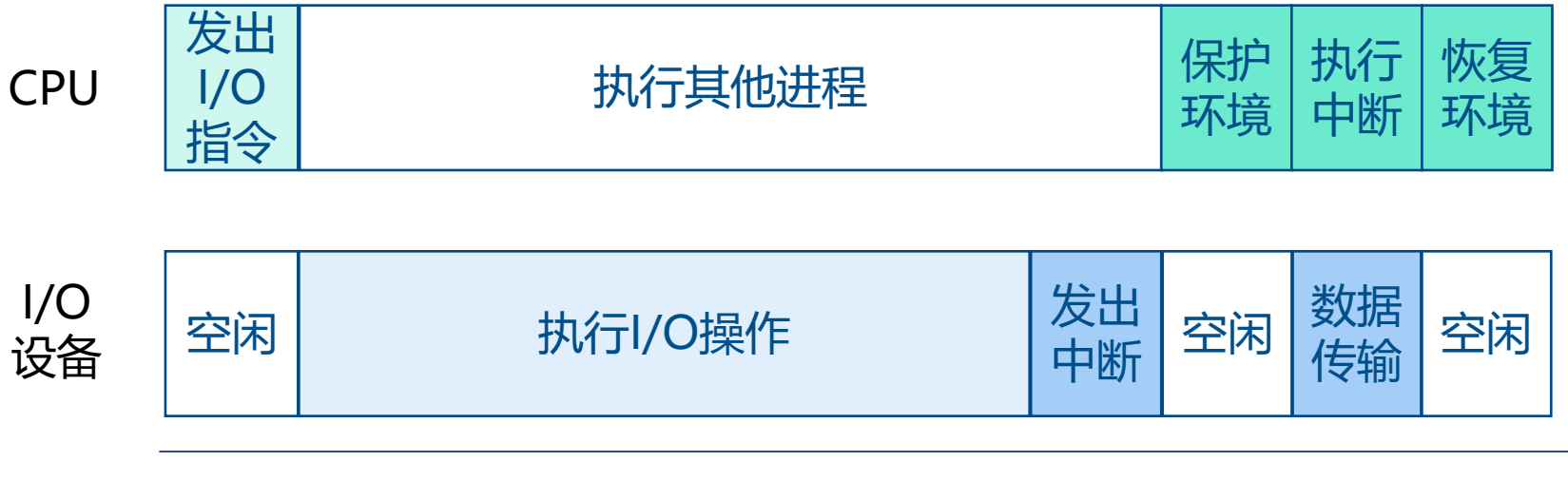
- 操作系统管理多个磁盘设备的中断
- 磁盘设备通过中断请求线发出中断请求
- 中断控制器接收设备的中断请求
- 中断控制器为每个中断请求分配资源
  - ✓ 中断号：中断的编号，0~255
  - ✓ 中断优先级：中断处理的次序
  - ✓ 中断向量：中断请求对应的处理程序首地址
- 中断控制器根据优先级选择告知CPU的中断
  - ✓ 发出全局中断请求
  - ✓ 告知CPU中断向量地址
- CPU保存当前进程执行环境，进入核心态，跳转到中断向量处执行中断服务程序（ISR）
- 中断处理结束，CPU返回到被中断的进程处继续



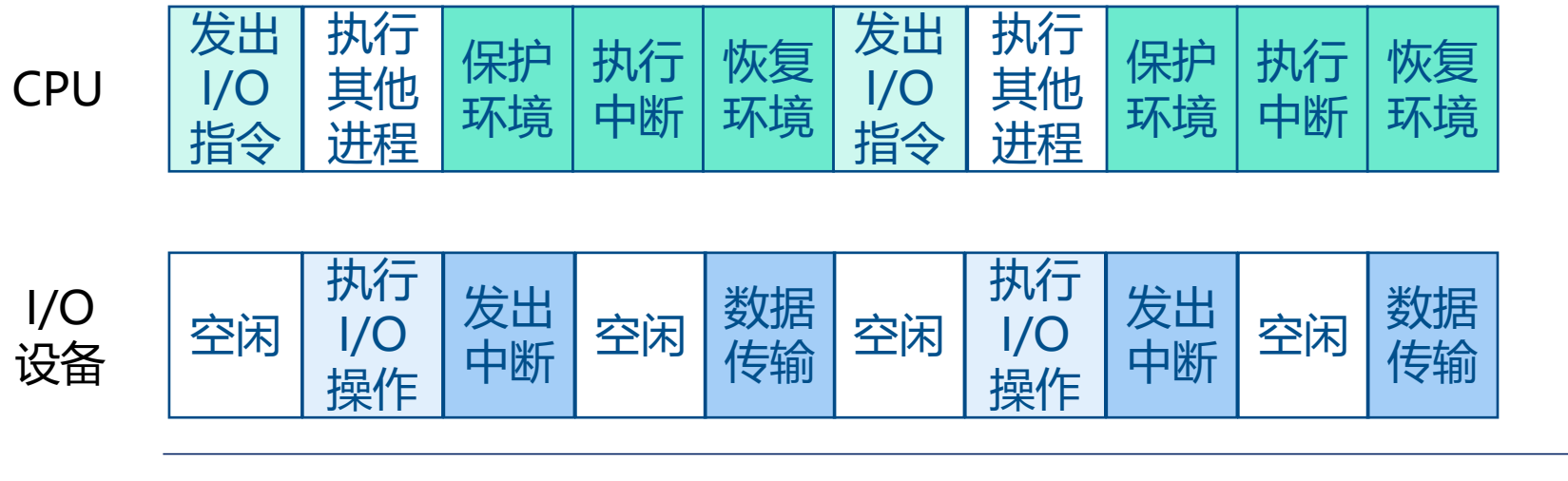
# 4. 存储设备访问-中断驱动方式


- CPU发出I/O指令
- I/O设备执行指令，CPU执行其他程序，不再访问I/O设备，
- I/O设备完成操作，发出中断
- CPU响应中断，保护当前CPU执行的进程执行环境
- CPU转去中断向量地址，执行中断服务程序，和I/O设备进行数据、状态交互
- CPU执行中断服务程序结束，返回之前的进程，恢复进程执行状态

# 4. 存储设备访问-中断驱动方式




- I/O设备操作时间长
- CPU和I/O设备并行工作



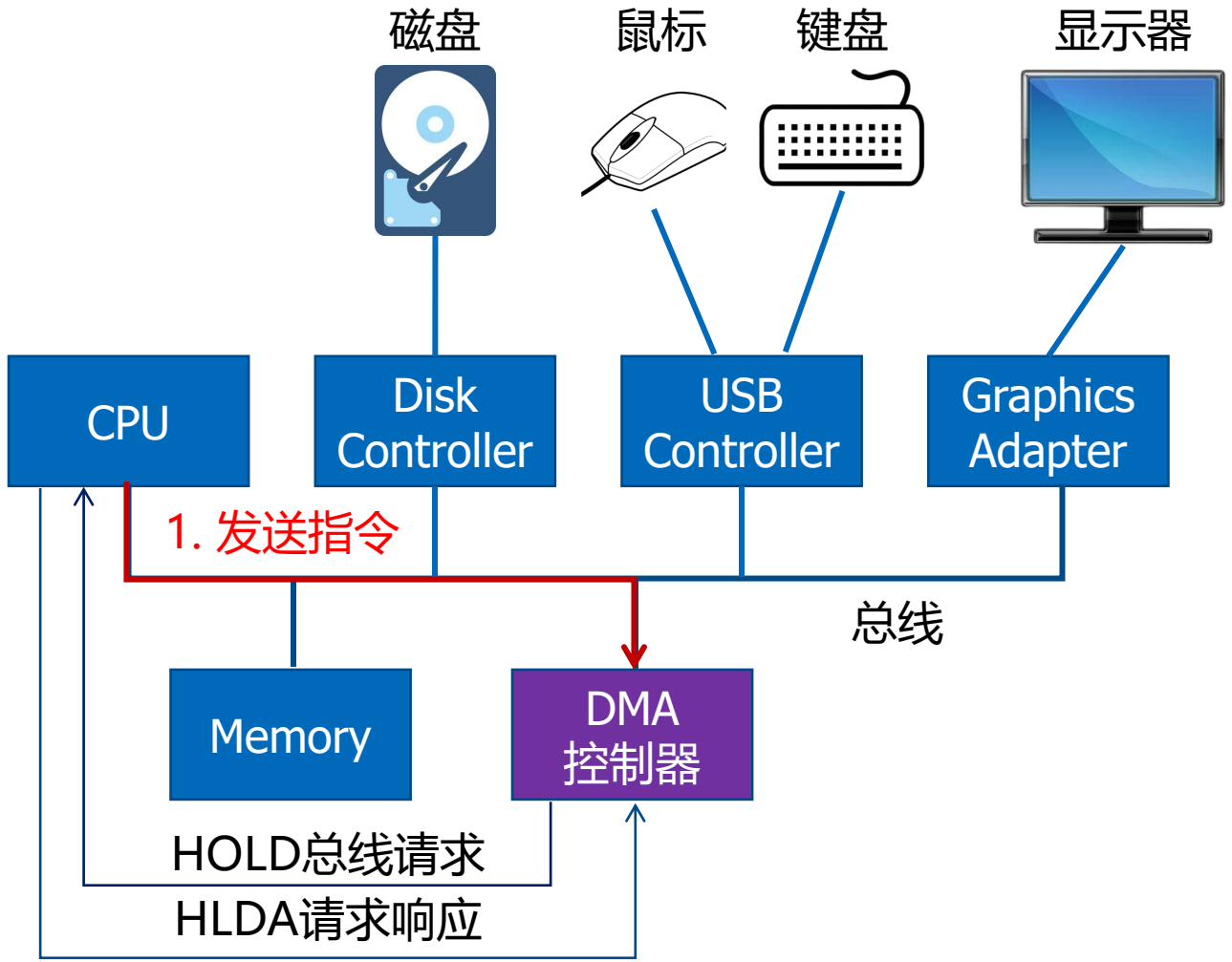

- I/O设备操作时间短
- CPU中断处理消耗过大
- CPU和I/O设备并行性差

- 当I/O设备操作时间较短，且CPU短时内大量发出中断请求，造成频繁处理中断
- 中断处理的过程占用时间过大，导致CPU过载，称为活锁（livelock）
- 活锁现象说明中断方式不是总比PIO方式优秀

# 4. 存储设备访问-直接存储器存取DMA方式

1. CPU向DMA控制器发送指令
   告知数据访问和保存的位置

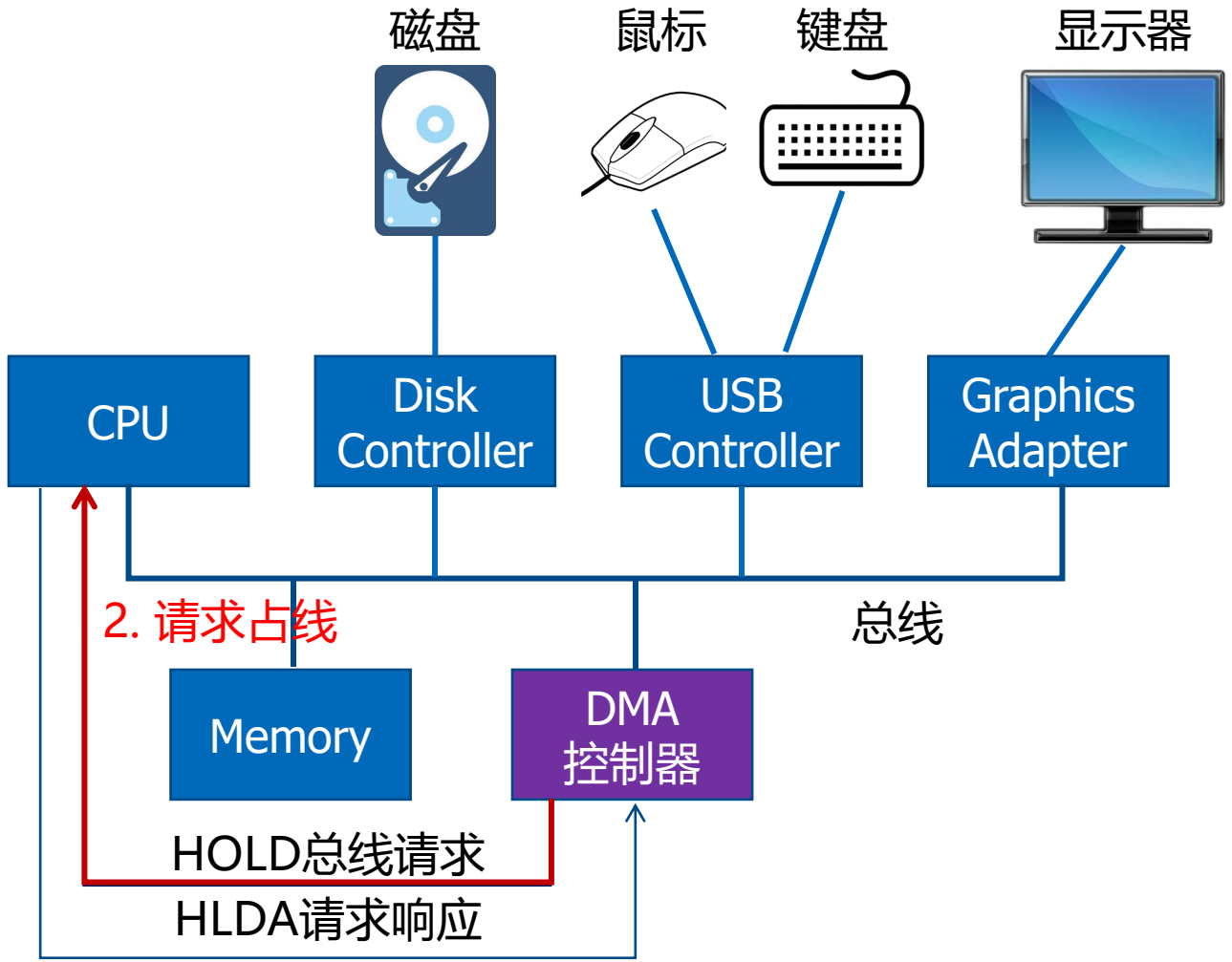
磁盘DMA方式读写过程



# 4. 存储设备访问-直接存储器存取DMA方式

1. CPU向DMA控制器发送指令
   告知数据访问和保存的位置
2. DMA控制器通过HOLD请求
   CPU放弃总线

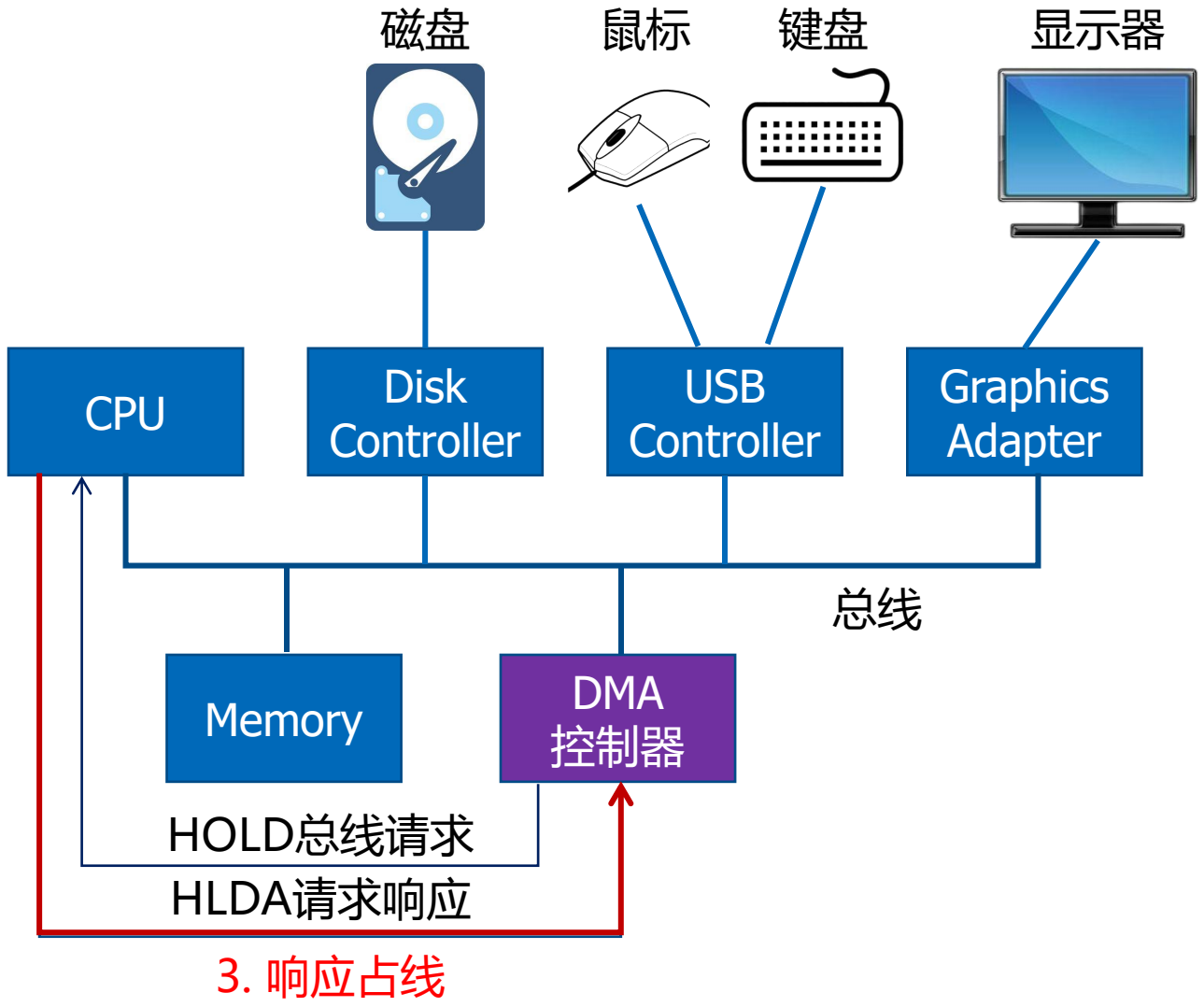
磁盘DMA方式读写过程



# 4. 存储设备访问-直接存储器存取DMA方式

1. CPU向DMA控制器发送指令告知数据访问和保存的位置
2. DMA控制器通过HOLD请求CPU放弃总线
3. CPU通过HLDA响应DMA控制器，并放弃总线操作权

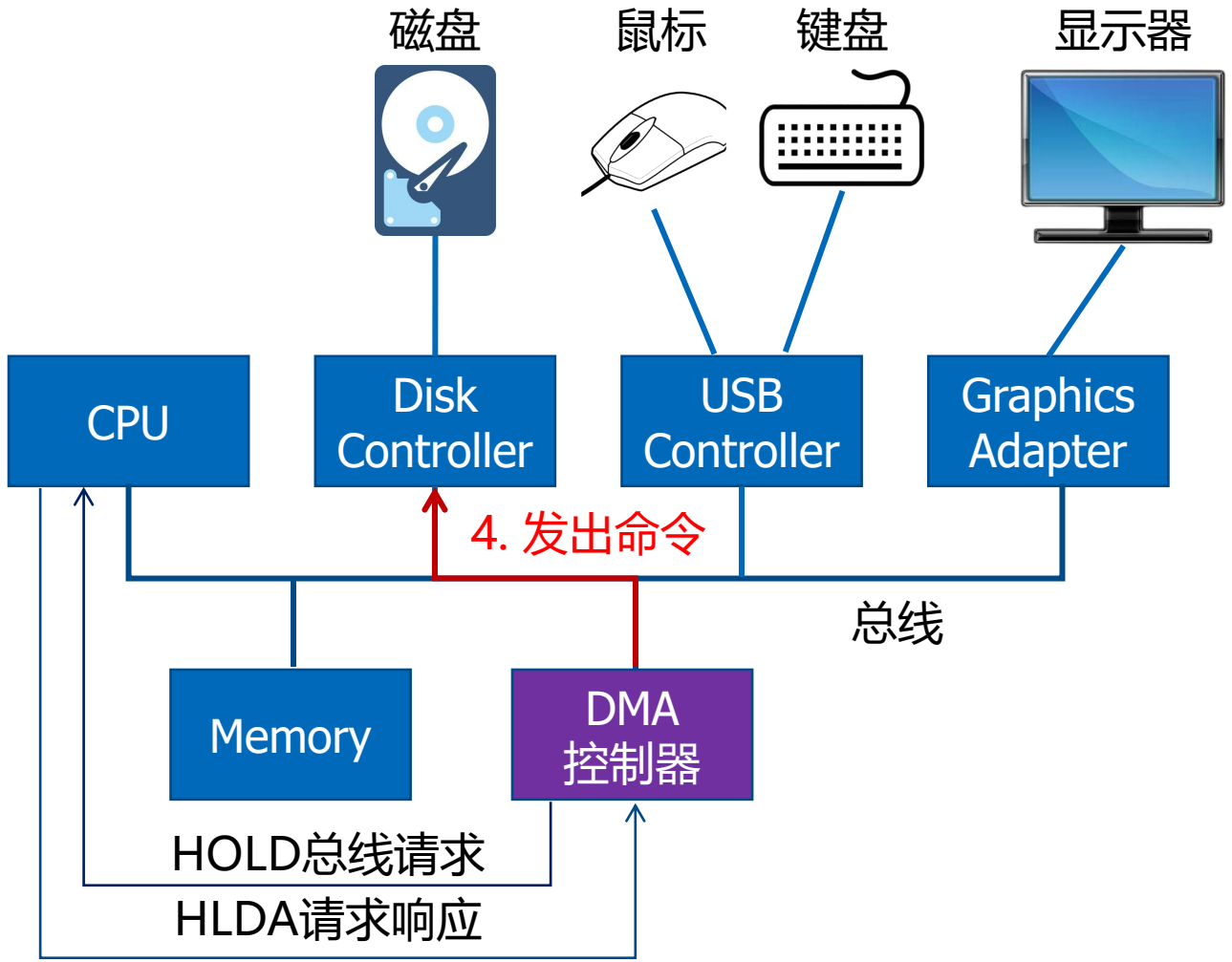
磁盘DMA方式读写过程



# 4. 存储设备访问-直接存储器存取DMA方式

1. CPU向DMA控制器发送指令告知数据访问和保存的位置
2. DMA控制器通过HOLD请求CPU放弃总线
3. CPU通过HLDA响应DMA控制器，并放弃总线操作权
4. DMA控制器向磁盘控制器发出数据访问指令，读取一个块

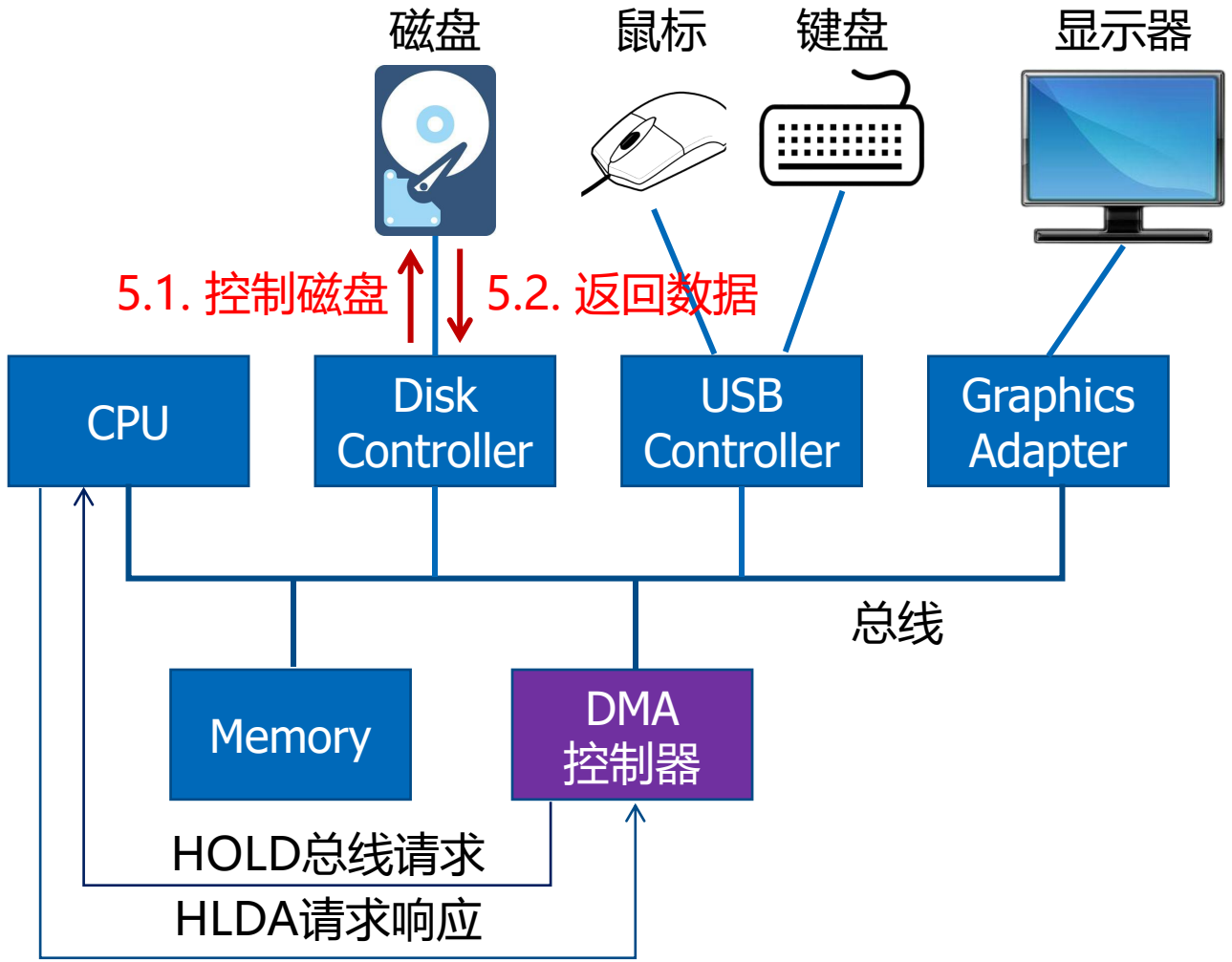
磁盘DMA方式读写过程



# 4. 存储设备访问-直接存储器存取DMA方式

1. CPU向DMA控制器发送指令告知数据访问和保存的位置
2. DMA控制器通过HOLD请求CPU放弃总线
3. CPU通过HLDA响应DMA控制器，并放弃总线操作权
4. DMA控制器向磁盘控制器发出数据访问指令，读取一个块
5. 磁盘控制器向磁盘发出数据访问指令，磁盘将数据写入控制器缓存中

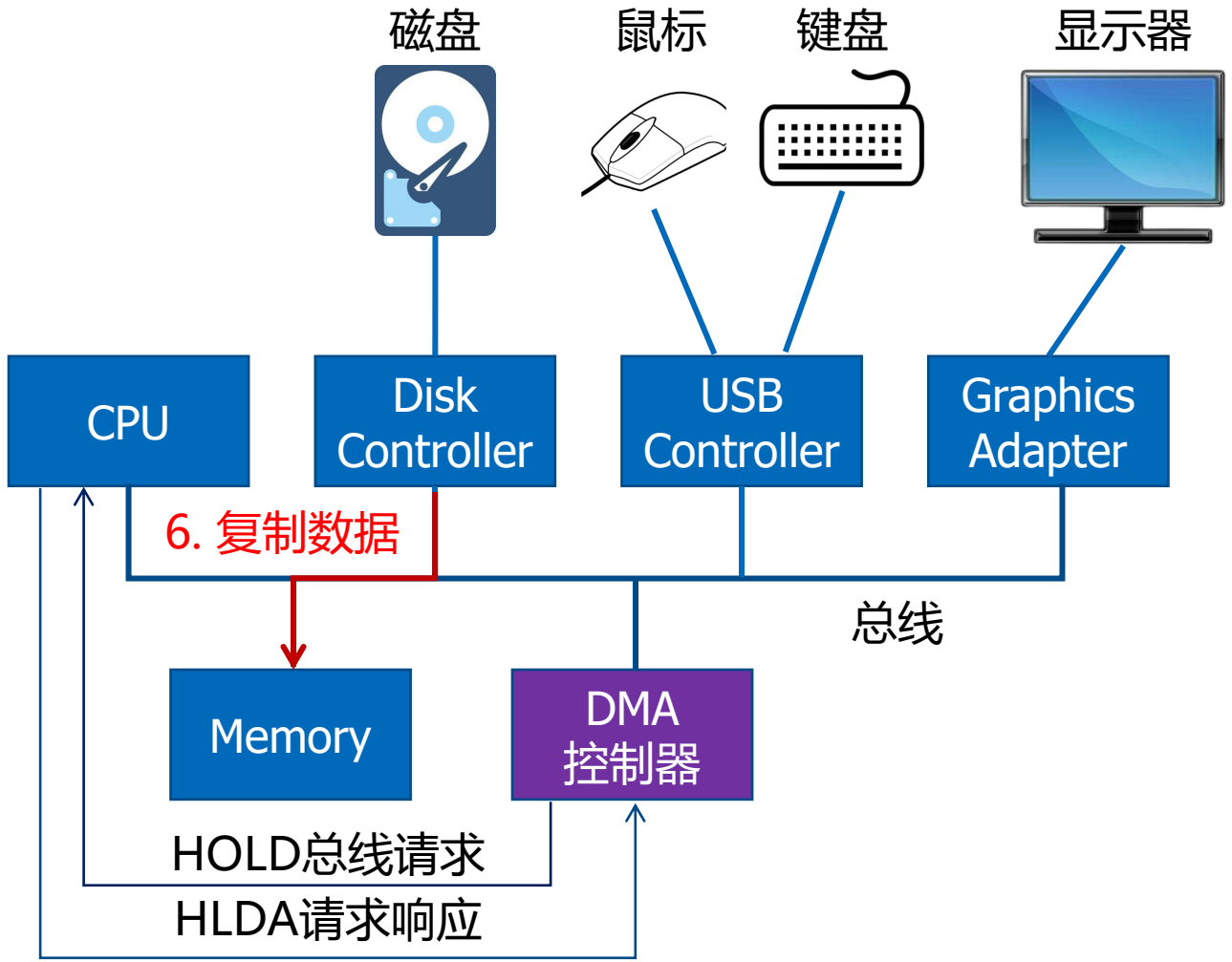
磁盘DMA方式读写过程



# 4. 存储设备访问-直接存储器存取DMA方式

1. CPU向DMA控制器发送指令告知数据访问和保存的位置
2. DMA控制器通过HOLD请求CPU放弃总线
3. CPU通过HLDA响应DMA控制器，并放弃总线操作权
4. DMA控制器向磁盘控制器发出数据访问指令，读取一个块
5. 磁盘控制器向磁盘发出数据访问指令，磁盘将数据写入控制器缓存中
6. DMA控制器读取磁盘控制器缓存内容，写入内存保存位置

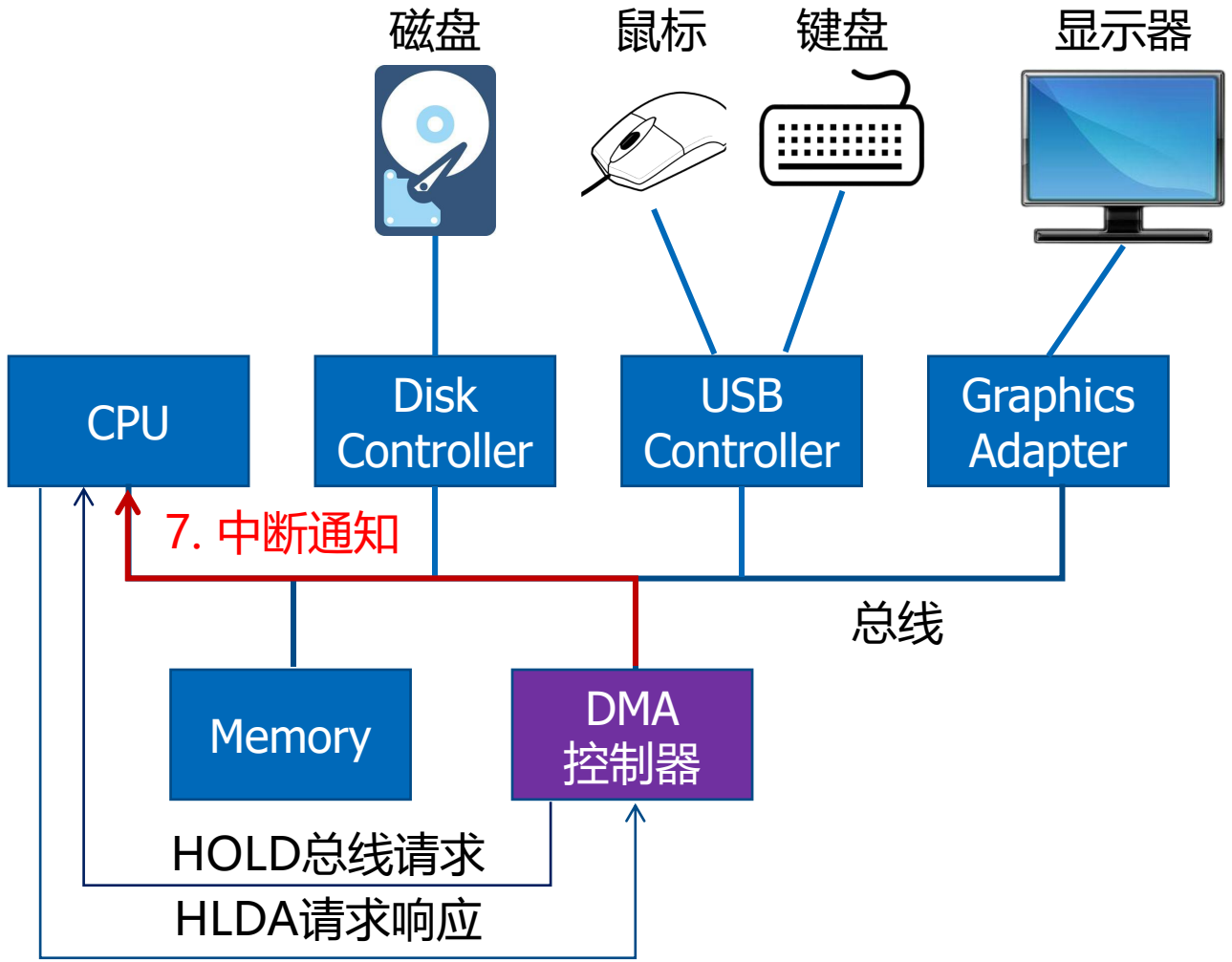
磁盘DMA方式读写过程



# 4. 存储设备访问-直接存储器存取DMA方式

1. CPU向DMA控制器发送指令告知数据访问和保存的位置
2. DMA控制器通过HOLD请求CPU放弃总线
3. CPU通过HLDA响应DMA控制器，并放弃总线操作权
4. DMA控制器向磁盘控制器发出数据访问指令，读取一个块
5. 磁盘控制器向磁盘发出数据访问指令，磁盘将数据写入控制器缓存中
6. DMA控制器读取磁盘控制器缓存内容，写入内存保存位置
7. 数据操作结束，DMA控制器通知CPU重新占用总线

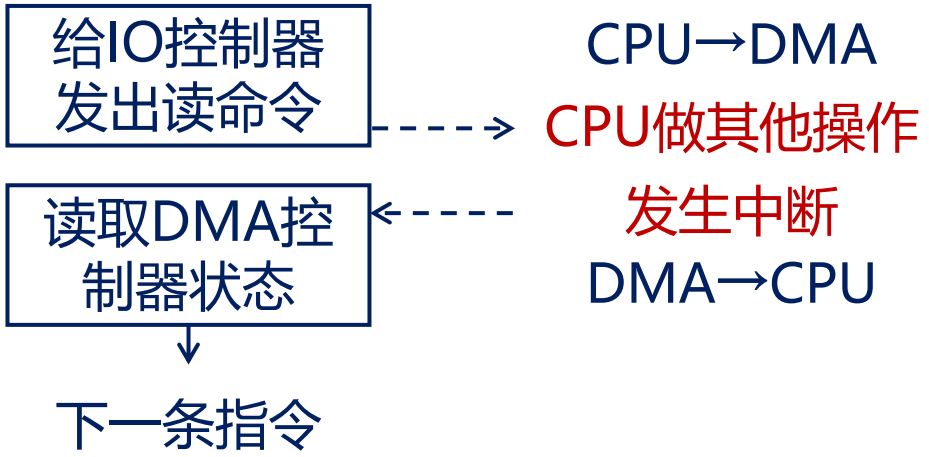
磁盘DMA方式读写过程



# 4. 存储设备访问-直接存储器存取DMA方式



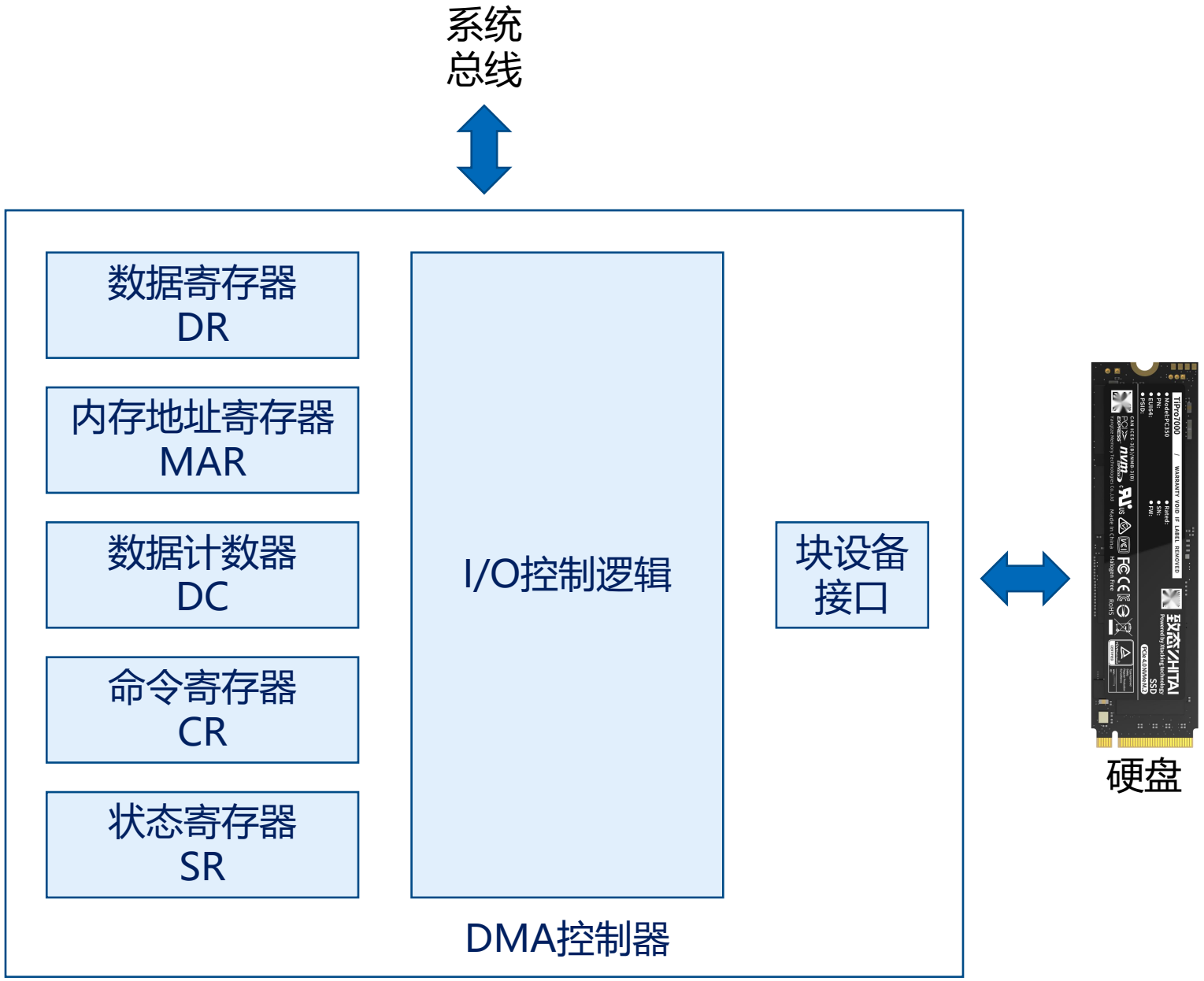
- 直接存储器存取方式工作流程
  - ✓ DMA方式CPU以块为单位进行数据传输
  - ✓ DMA的操作块大小根据设备特性变化
  - ✓ CPU向IO控制器发出读指令，告知DMA控制器传输的源位置和目的位置
  - ✓ DMA控制器接管总线，控制IO设备，将数据送入内存指定位置
  - ✓ DMA控制器传输任务完毕，以中断方式通知CPU恢复总线
- 优点：以块为单位传输，CPU介入进一步减少，CPU和I/O设备并行度更高
- 缺点：如果CPU访问的源区域或目的区域是离散的，需要多次DMA操作才能完成



# 4. 存储设备访问-直接存储器存取DMA方式



- DMA控制器的组成
  - ✓ DR：暂存从设备到内存，或从内存到设备的数据
  - ✓ MAR：表示数据应放到内存中的位置
  - ✓ DC：表示剩余要读/写的字节数
  - ✓ CR：存放CPU发来的I/O命令
  - ✓ SR：保存设备的状态信息
  - ✓ I/O控制逻辑：DMA控制器功能实现
  - ✓ 块设备接口：访问块设备

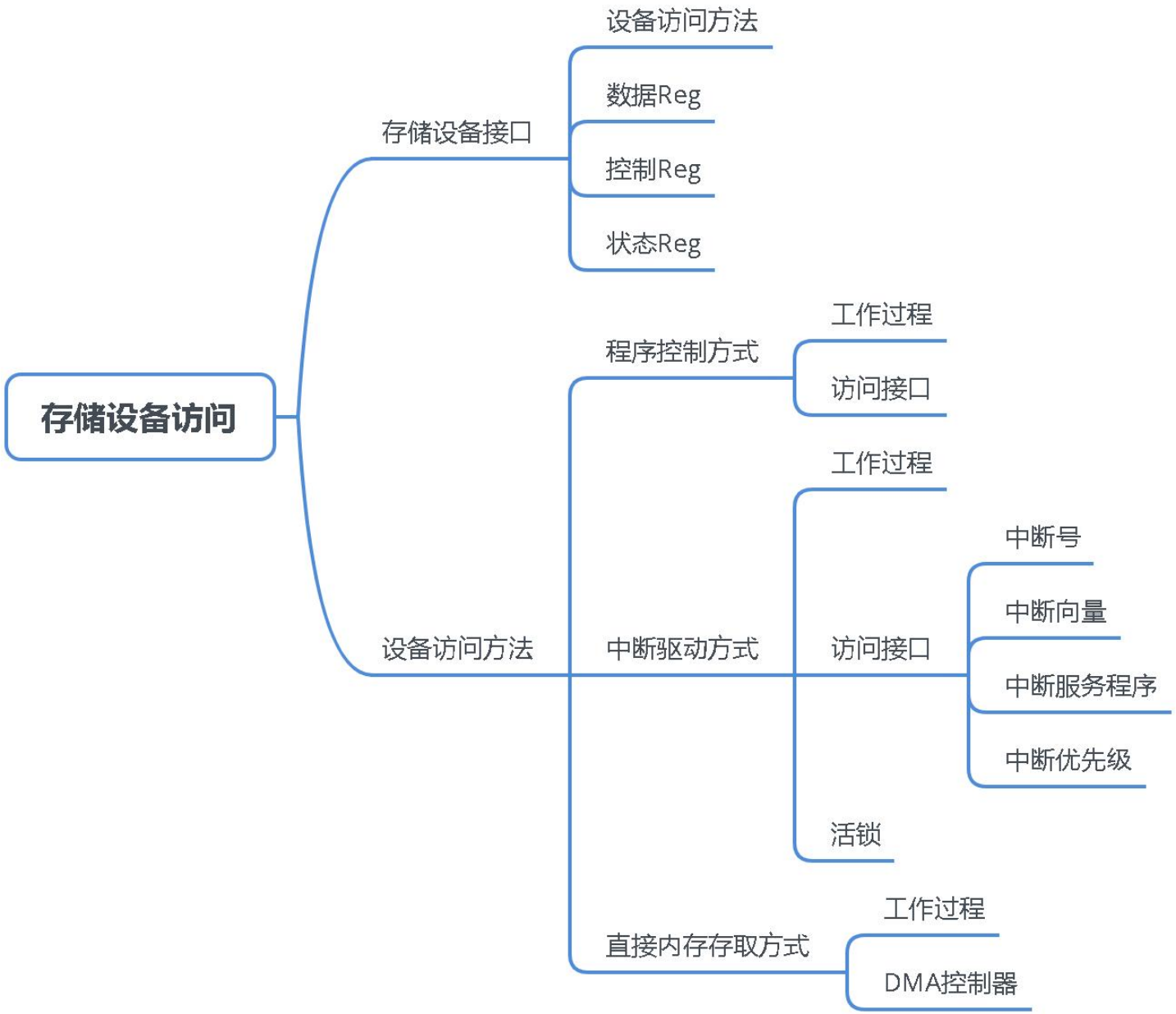


# 4. 存储设备访问-访问方式对比



| | 完成一次读/写的过程 | CPU干预频率 | 每次I/O的数据传输单位 | 数据流向<br>读/写 | 优缺点 |
|---|---|---|---|---|---|
| 程序直接控制方式 | CPU发出I/O命令后需不断轮询 | 极高 | 字 | 设备→CPU→内存<br>内存→CPU→设备 | 从上至下<br>CPU占用逐渐减小<br>CPU和设备的并行度逐渐提高 |
| 中断驱方式 | CPU发出I/O命令后可以做其他事<br>本次I/O完成设备控制器发出中断 | 高 | 字 | 设备→CPU→内存<br>内存→CPU→设备 | |
| DMA方式 | CPU发出I/O命令后可以做其他事<br>本次I/O完成DMA控制器发出中断信号 | 中 | 块 | 设备→内存<br>内存→设备 | |

# 4. 存储设备访问-小结


存储设备访问
存储设备接口
设备访问方法
数据Reg
控制Reg
状态Reg
设备访问方法
程序控制方式
工作过程
访问接口
中断驱动方式
工作过程
访问接口
中断号
中断向量
中断服务程序
中断优先级
活锁
直接内存存取方式
工作过程
DMA控制器