```
c010606f: 55                push   %ebp
c0106070: 89 e5             mov    %esp,%ebp
c0106072: 56                push   %esi
c0106073: 53                push   %ebx
c0106074: 83 ec 0c          sub    $0xc,%esp
c0106077: 68 44 a2 12 c0    push   $0xc012a24
c010607c: e8 ed 1c 00 00    call   c0107d6e <printf>
c0106081: be 00 60 10 00    mov    $0x106000,
c0106086: c1 ee 0c          shr    $0xc,%esi
c0106089: 83 c4 08          add    $0x8,%esp
c010608c: 56                push   %esi
c010608d: 68 6c a2 12 c0    push   $0xc012a26c
c0106092: e8 d7 1c 00 00    call   c0107d6e <p
c0106097: 83 c4 10          add    $0x1
c010609a: bb 00 00 00 00    mov    $0x0,
c010609f: eb 14             jmp    c01060b5 <_kernel_post_init+0x46>
c01060a1: 89 d8             mov    %ebx,%eax
c01060a3: c1 e0 0c          shl    $0xc,%eax
c01060a6: 83 ec 0c          sub    $0xc,%esp
c01060a9: 50                push   %eax
c01060aa: e8 9f 0a 00 00    call   c0106b4e <vmm_unmap_page>
c01060af: 83 c
c01060b2: 83 c
c01060b5: 39 f3
c01060b7: 72 e8
c01060b9: e8 54
c01060be: 85 c
c01060c0: 74 07
c01060c2: 8d 65 f8
c01060c5: 5b
c01060c6: 5e
c01060c7: 5d
c01060c8: c3
c01060c9: 83 ec 04          $0x4,%esp
c01060cc: 6a 40             $0x40
c01060ce: 68 af a0 12 c0    push   $0xc012a0af
```

LunaixOS

SERIAL ATA

从零开始

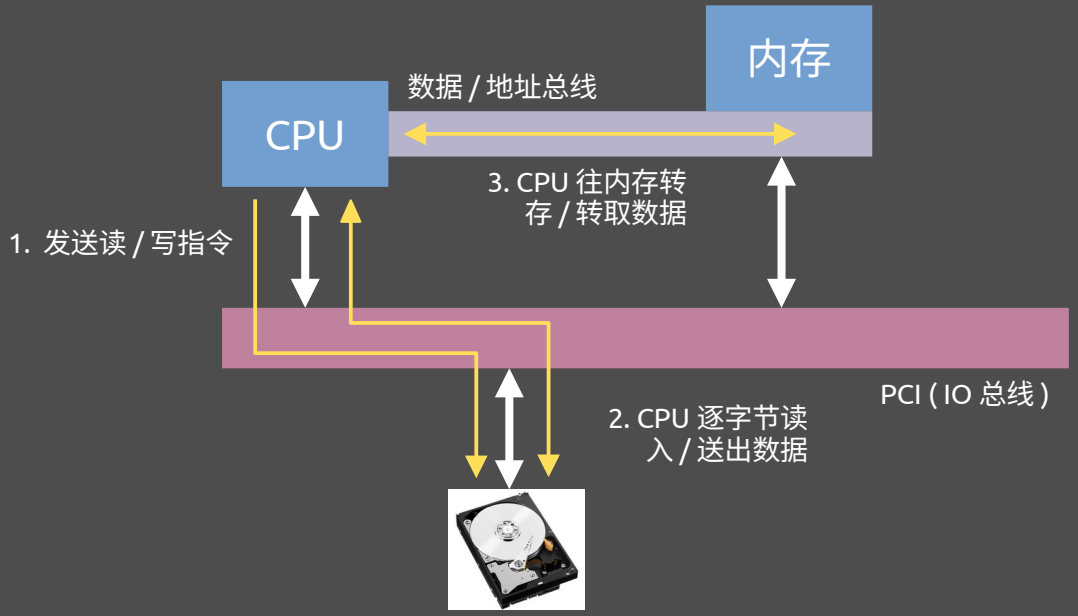
自制操作系统

磁盘与 SATA

深入浅出 SATA 协议

EP 13-1

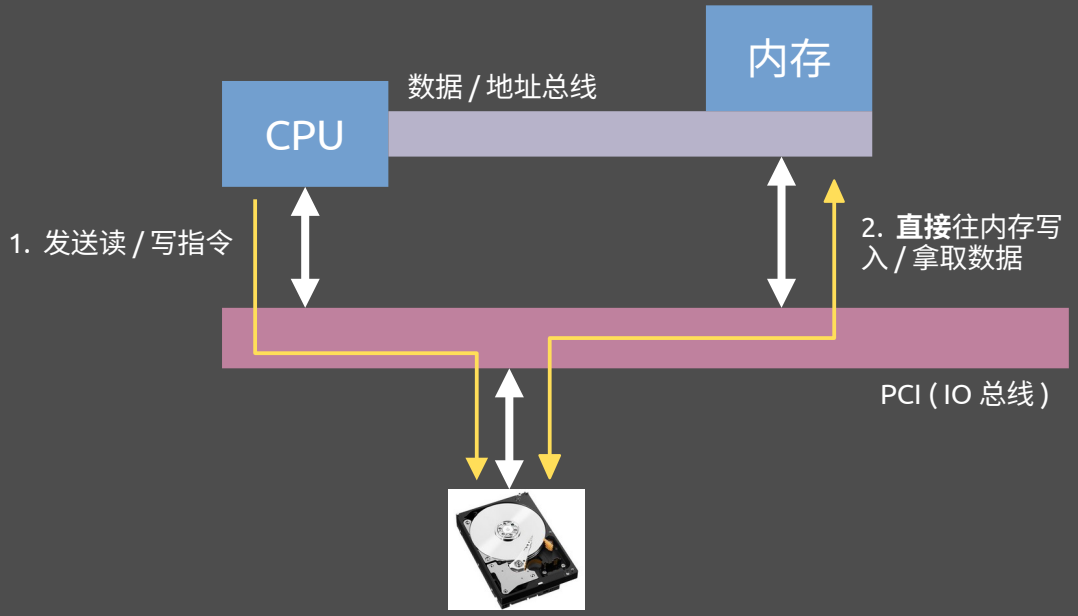
如何读写磁盘?



PIO

Programmed Input/Output

如何读写磁盘?



DMA

Direct Memory Access

读写磁盘：两种方式

PIO

1. CPU 下达指令

2. 磁盘响应指令，**通过 CPU** ，向内存拿取或写入数据。

DMA

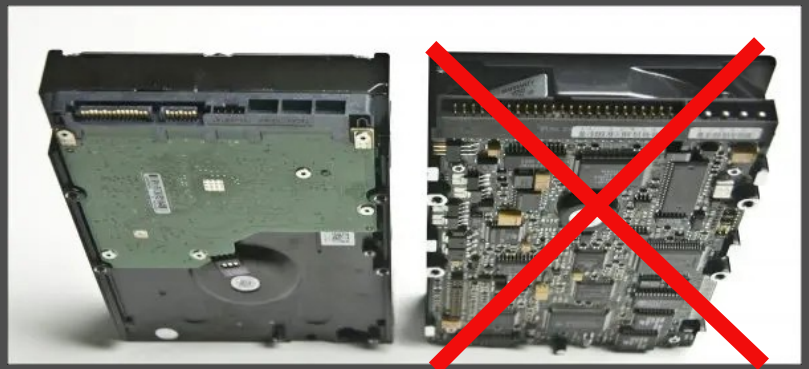
1. CPU 下达指令

2. 磁盘响应指令，根据事先同 CPU 商定好的地址，**通过 DMA 控制器**，直接向内存拿取或写入数据。

一个共同的问题：如何向磁盘发送命令？磁盘如何响应命令，协调数据的传输？

这就是 SATA 存在的目的！

SATA： 21 世纪的磁盘读写姿势



2003 SATA 1

2004 SATA 2

2009 SATA 3

2011 SATA 3.1

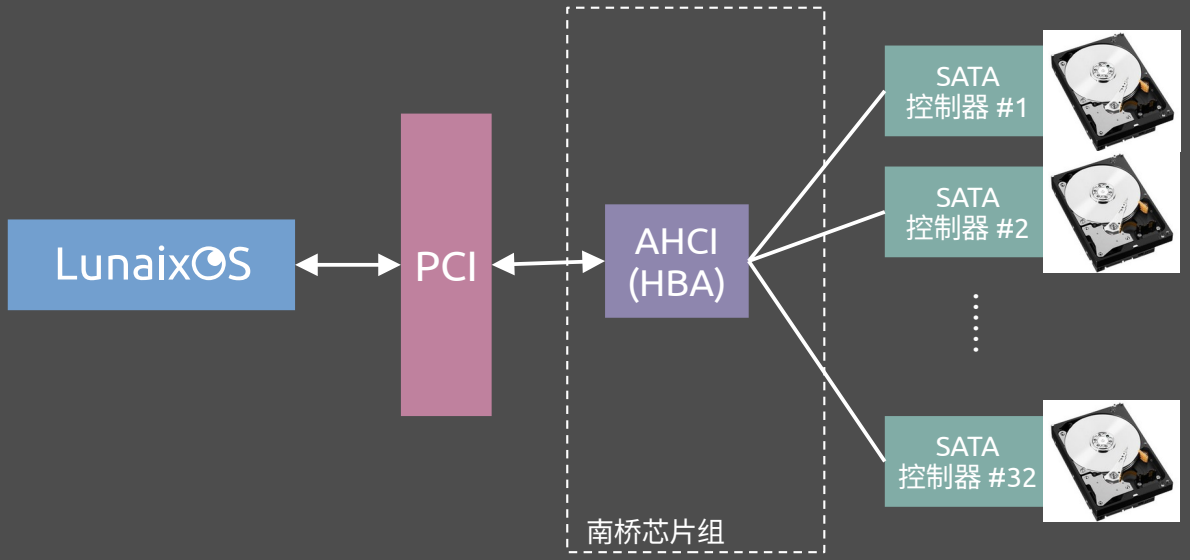
2013 SATA 3.2 ← LunaixOS

2016 SATA 3.3

2018 SATA 3.4

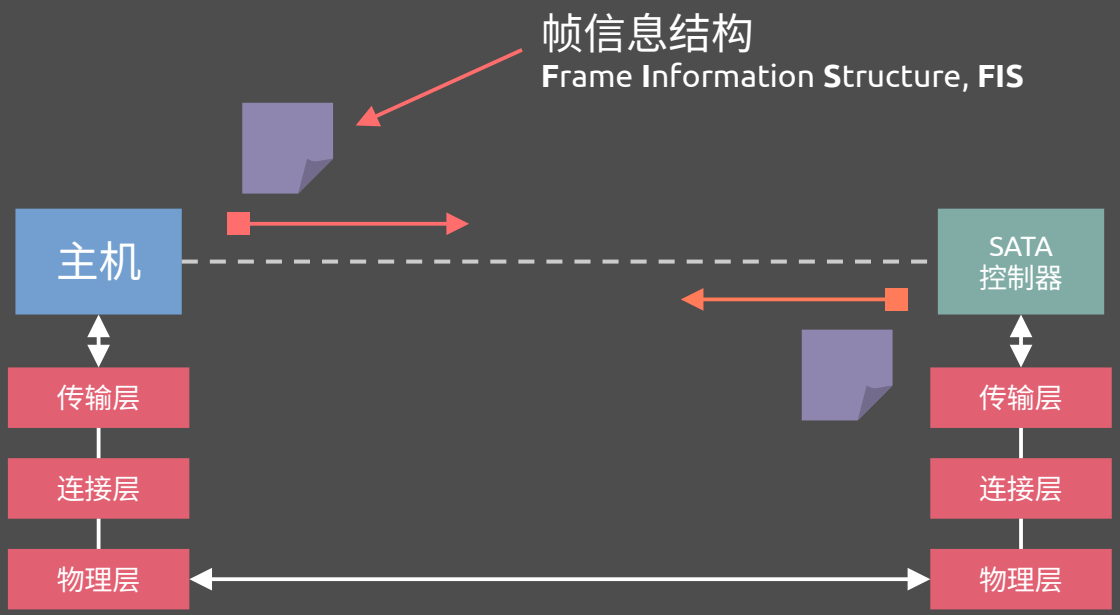
2020 SATA 3.5

操作系统的视角



SATA 协议： FIS

基于报文交换机制的 SATA：

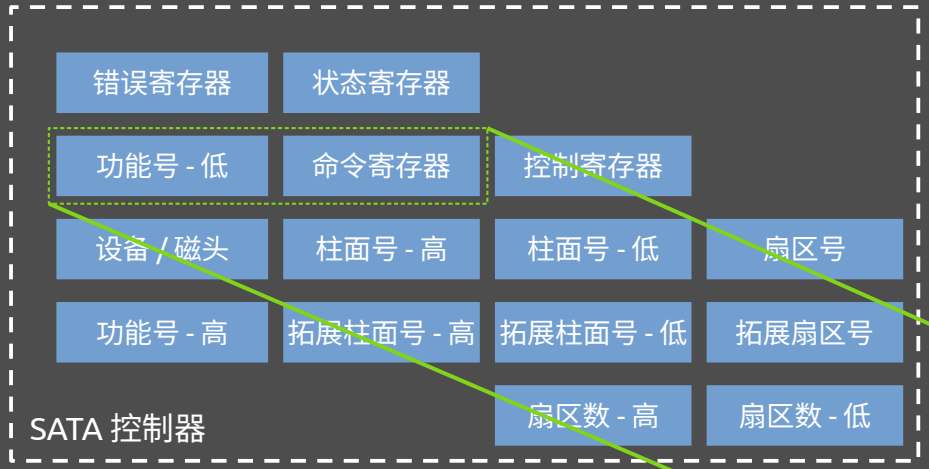


SATA 协议： FIS 的种类

SATA 标准中定义了如下的 FIS：

- Rh 寄存器 FIS （主机至设备）
- Rd 寄存器 FIS （设备至主机）
- Dev 设备设置 FIS （设备至主机）
- Ds DMA 配置 FIS （双向）
- Da DMA 激活 FIS （设备至主机）
- Ps PIO 配置 FIS （设备至主机）
- Dat 数据 FIS （双向）
- BIST 自检 FIS （双向）

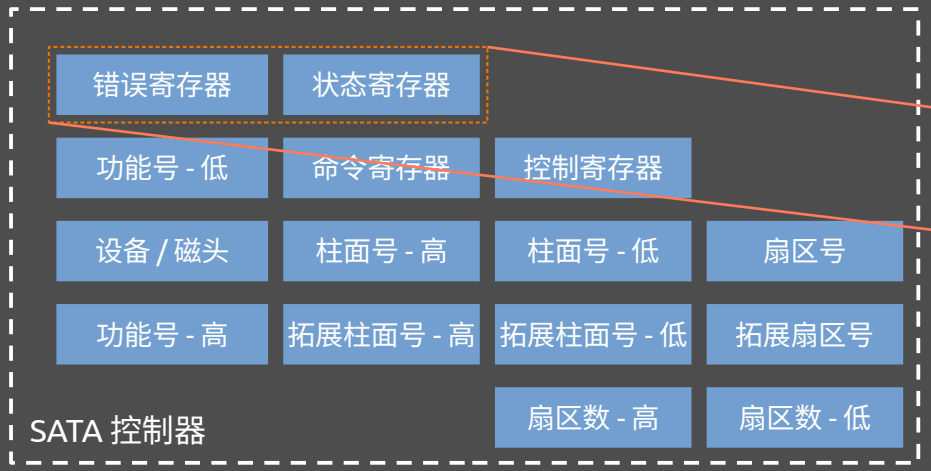
操控 SATA 设备：设备寄存器是关键



0	Features(7:0)	Command(7:0)	C	R	R	R	PM Port	FIS Type (27h)	
1	Device(7:0)	LBA(23:16)	LBA(15:8)					LBA(7:0)	
2	Features(15:8)	LBA(47:40)	LBA(39:32)					LBA(31:24)	
3	Control(7:0)	ICC(7:0)	Count(15:8)					Count(7:0)	
4	Auxiliary(31:24)	Auxiliary(23:16)	Auxiliary(15:8)					Auxiliary(7:0)	

寄存器 FIS： 主机到设备 - 向寄存器写入值

操控 SATA 设备：设备寄存器是关键



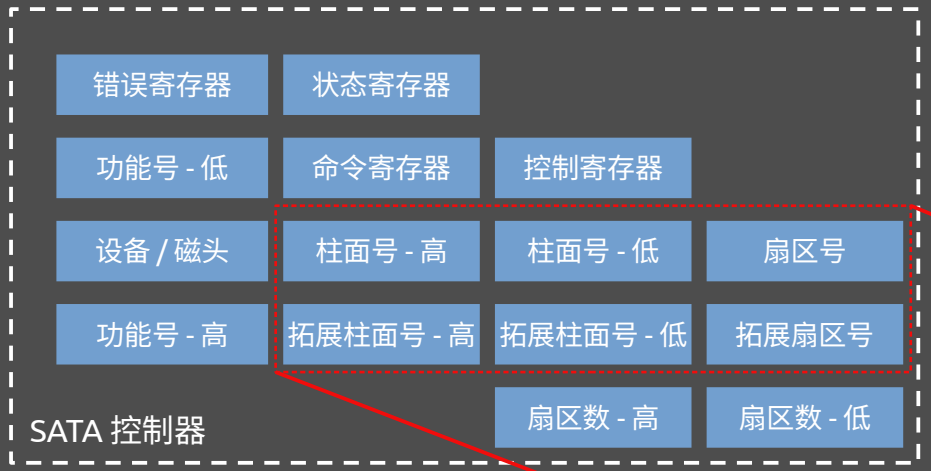
0	Error(7:0)	Status(7:0)	R	I	R	R	PM Port	FIS Type (34h)
1	Device(7:0)	LBA(23:16)	LBA(15:8)				LBA(7:0)
2	Reserved (0)	LBA(47:40)	LBA(39:32)				LBA(31:24)
3	Reserved (0)	Reserved (0)	Count(15:8)				Count(7:0)
4	Reserved (0)	Reserved (0)	Reserved (0)				Reserved (0)

寄存器 FIS： 设备到主机 - 控制器汇报寄存器值

0	Features(7:0)	Command(7:0)	C	R	R	R	PM Port	FIS Type (27h)
1	Device(7:0)	LBA(23:16)	LBA(15:8)				LBA(7:0)
2	Features(15:8)	LBA(47:40)	LBA(39:32)				LBA(31:24)
3	Control(7:0)	ICC(7:0)	Count(15:8)				Count(7:0)
4	Auxiliary(31:24)	Auxiliary(23:16)	Auxiliary(15:8)				Auxiliary(7:0)

寄存器 FIS： 主机到设备 - 向寄存器写入值

操控 SATA 设备：设备寄存器是关键



?

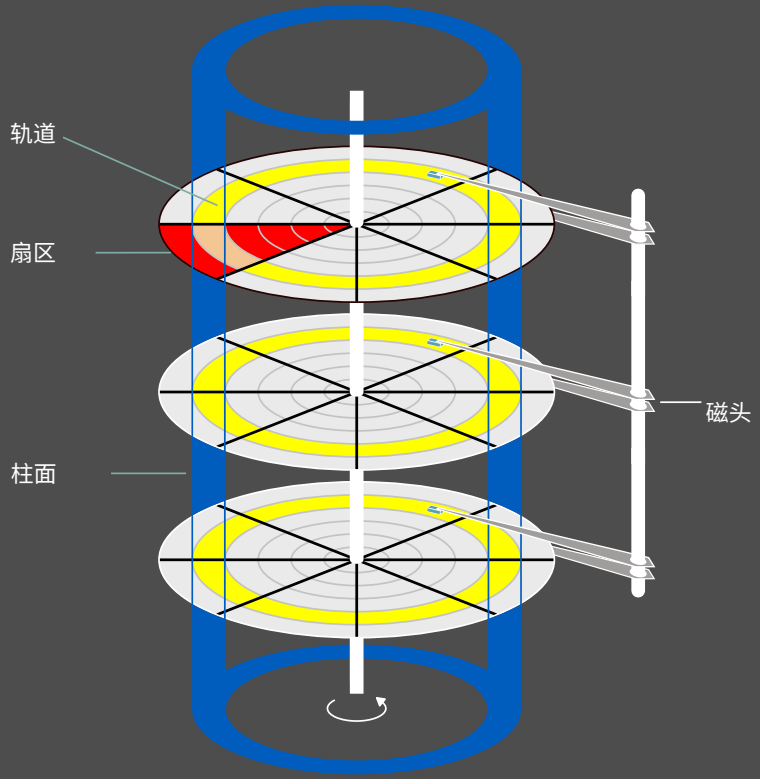
0	Error(7:0)		Status(7:0)	R	I	R	R	PM Port	FIS Type (34h)
1	Device(7:0)		LBA(23:16)		LBA(15:8)			LBA(7:0)
2	Reserved (0)		LBA(47:40)		LBA(39:32)			LBA(31:24)
3	Reserved (0)		Reserved (0)		Count(15:8)			Count(7:0)
4	Reserved (0)		Reserved (0)		Reserved (0)			Reserved (0)

寄存器 FIS： 设备到主机 - 控制器汇报寄存器值

0	Features(7:0)		Command(7:0)	C	R	R	R	PM Port	FIS Type (27h)
1	Device(7:0)		LBA(23:16)		LBA(15:8)			LBA(7:0)
2	Features(15:8)		LBA(47:40)		LBA(39:32)			LBA(31:24)
3	Control(7:0)		ICC(7:0)		Count(15:8)			Count(7:0)
4	Auxiliary(31:24)		Auxiliary(23:16)		Auxiliary(15:8)			Auxiliary(7:0)

寄存器 FIS： 主机到设备 - 向寄存器写入值

LBA 与 CHS

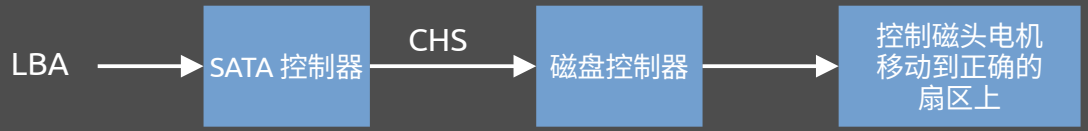


柱面号：磁头号：扇区号

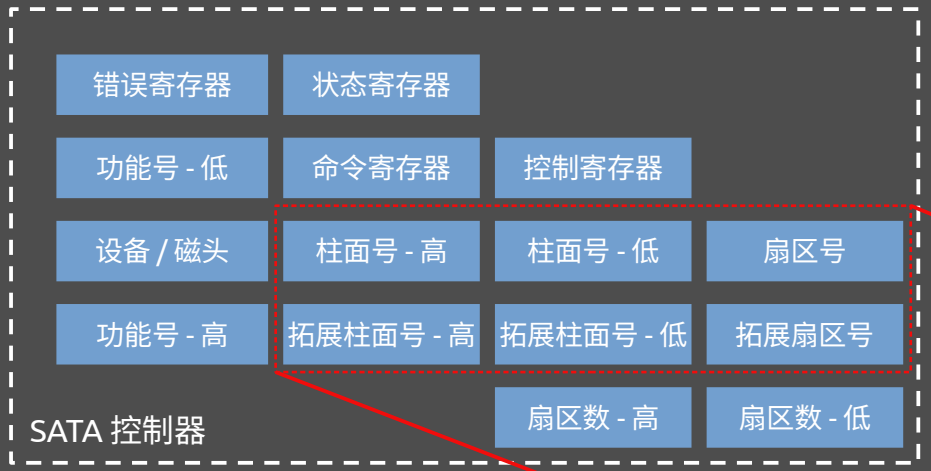
C： H： S

SATA 控制器将物理扇区映射到逻辑扇区

软件可通过逻辑扇区的编号访问扇区



操控 SATA 设备：设备寄存器是关键



0	Error(7:0)	Status(7:0)	R	I	R	R	PM Port	FIS Type (34h)
1	Device(7:0)	LBA(23:16)	LBA(15:8)					LBA(7:0)
2	Reserved (0)	LBA(47:40)	LBA(39:32)					LBA(31:24)
3	Reserved (0)	Reserved (0)	Count(15:8)					Count(7:0)
4	Reserved (0)	Reserved (0)	Reserved (0)					Reserved (0)

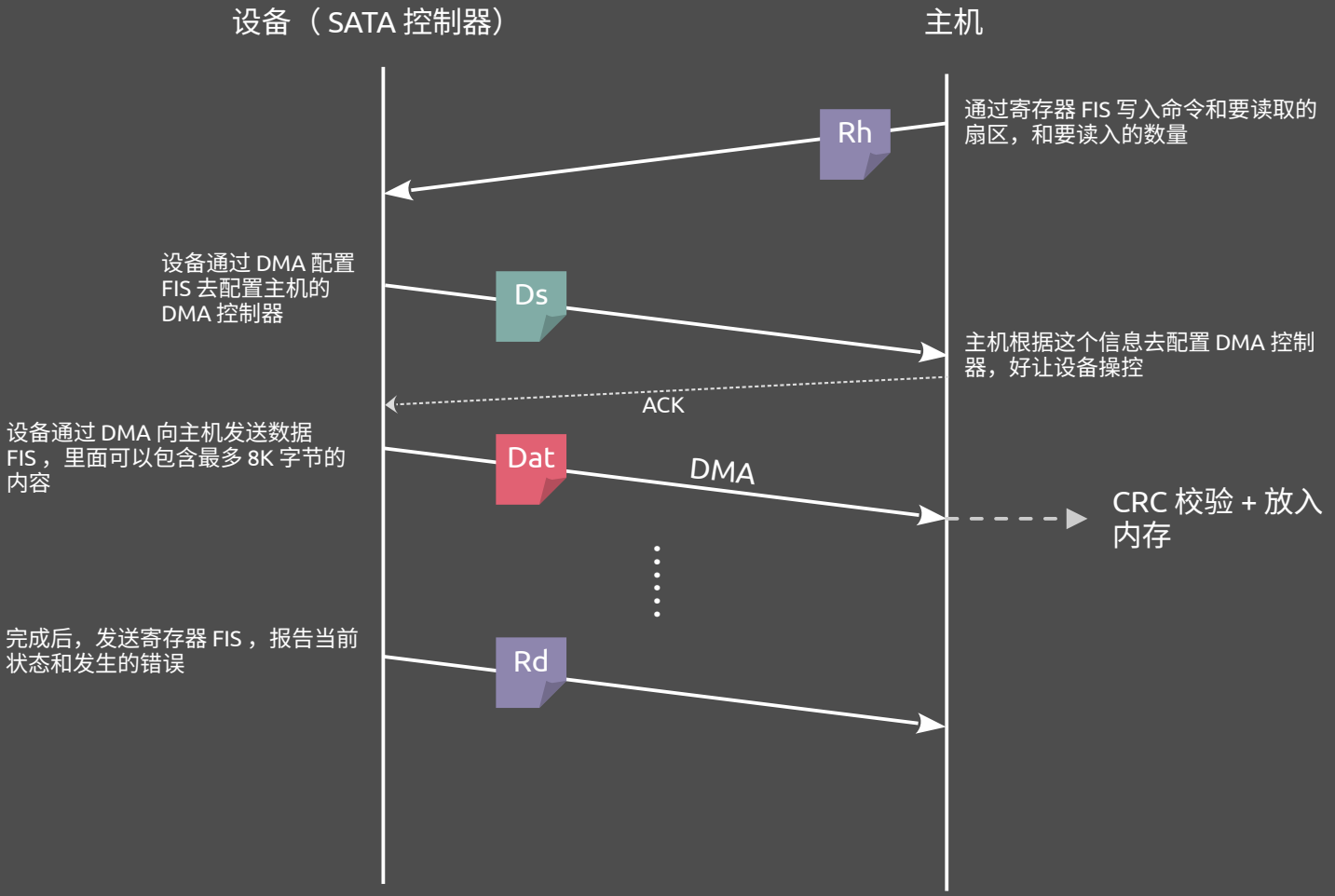
寄存器 FIS： 设备到主机 - 控制器汇报寄存器值

0	Features(7:0)	Command(7:0)	C	R	R	R	PM Port	FIS Type (27h)
1	Device(7:0)	LBA(23:16)	LBA(15:8)					LBA(7:0)
2	Features(15:8)	LBA(47:40)	LBA(39:32)					LBA(31:24)
3	Control(7:0)	ICC(7:0)	Count(15:8)					Count(7:0)
4	Auxiliary(31:24)	Auxiliary(23:16)	Auxiliary(15:8)					Auxiliary(7:0)

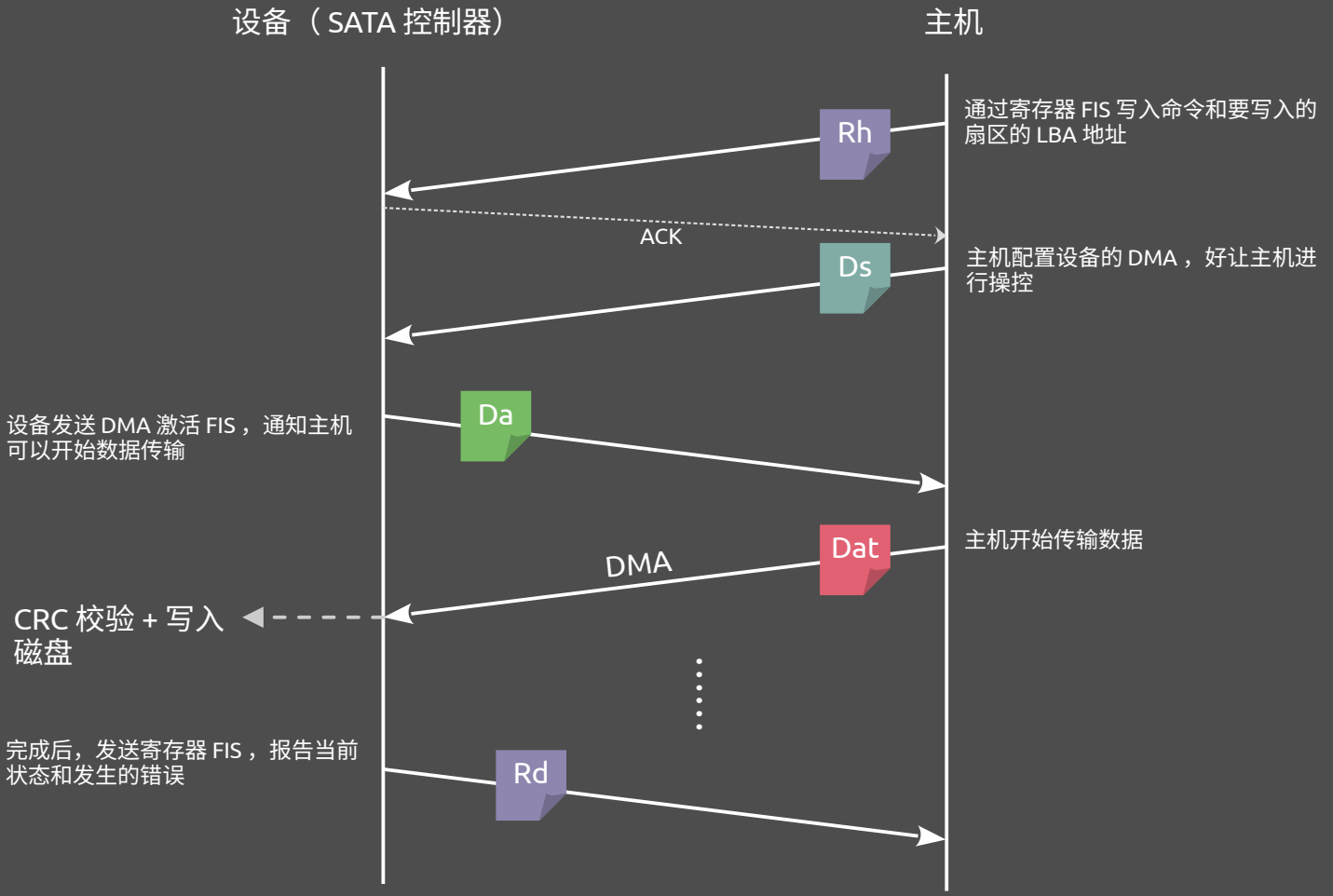
寄存器 FIS： 主机到设备 - 向寄存器写入值

如果使用 LBA 模式寻址，SATA 控制器将会将其翻译成 CHS

SATA 协议： DMA 读扇区



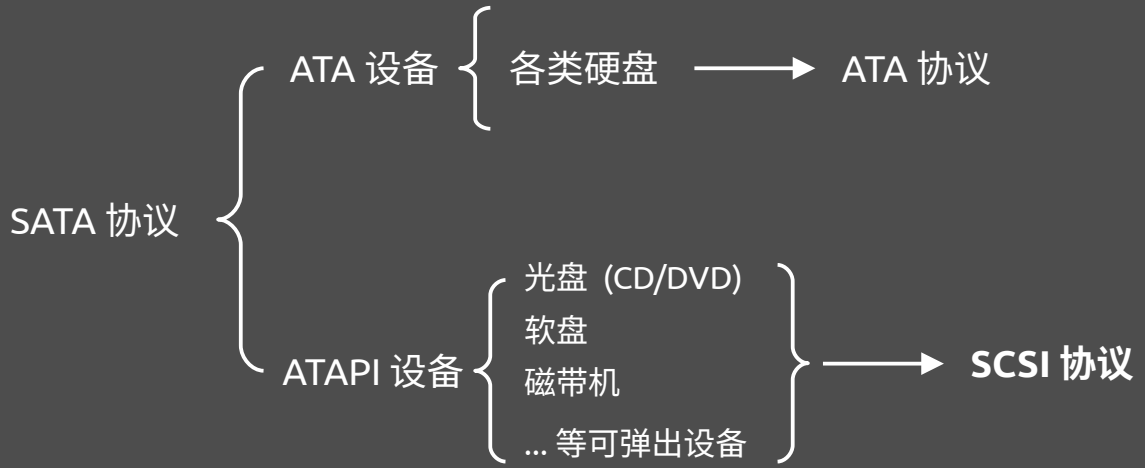
SATA 协议： DMA 写扇区



一些特殊的设备

前面的流程是对硬盘的读写（ATA 设备）

并**不适用**于软盘，光盘的操作!



SATA 协议提供了对 SCSI 协议的封装

SCSI 协议中的命令

SATA 中，命令是封装在寄存器 FIS 中的

SCSI 中，命令是封装在**命令描述块 (CDB)** 里面

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE							
1	Miscellaneous CDB information			SERVICE ACTION (if required)				
2	(MSB)							
3								
4				LOGICAL BLOCK ADDRESS (if required)				
5								(LSB)
6	(MSB)							
7								
8				TRANSFER LENGTH (If required) PARAMETER LIST LENGTH (if required) ALLOCATION LENGTH (if required)				
9								(LSB)
10	Miscellaneous CDB information							
11	CONTROL							

12 字节的 CDB

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE							
1	Miscellaneous CDB information							
2	(MSB)							
3								
4								
5								
6				LOGICAL BLOCK ADDRESS				
7								
8								
9								(LSB)
10	(MSB)							
11								
12				TRANSFER LENGTH (If required) PARAMETER LIST LENGTH (if required) ALLOCATION LENGTH (if required)				
13								(LSB)
14	Miscellaneous CDB information							
15	Control							

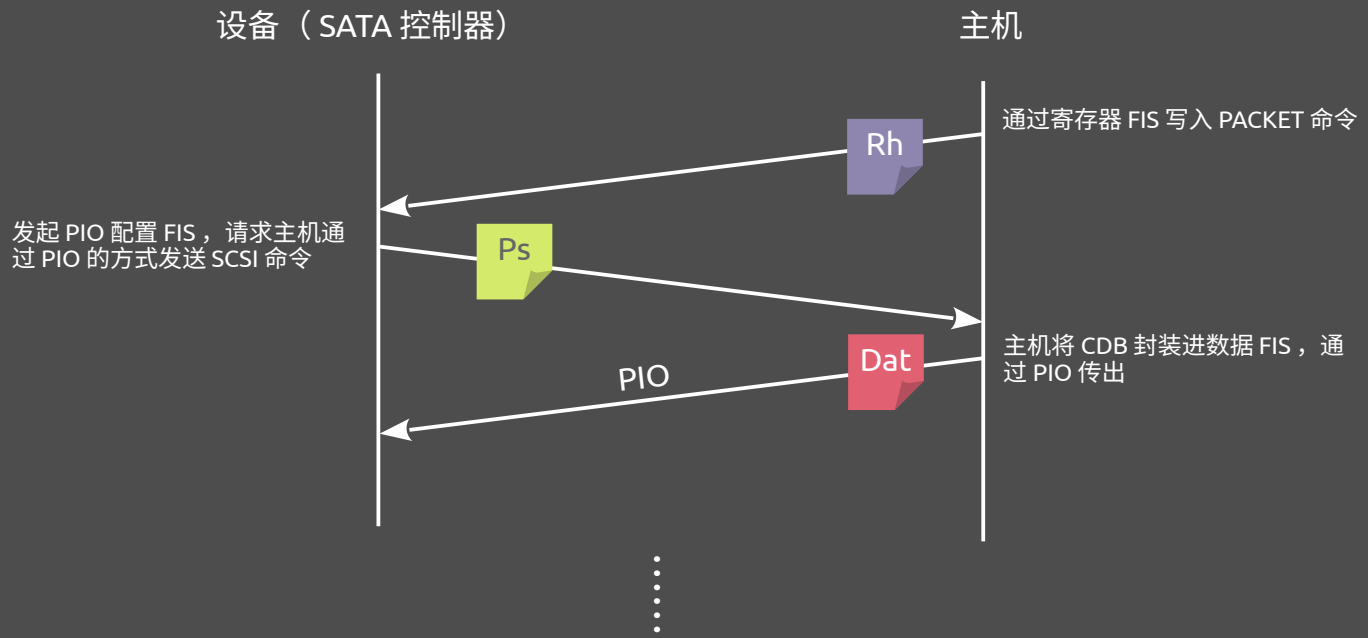
16 字节的 CDB

注意：对整数的封装， FIS 使用的是**小端序**， CDB 使用的是**大端序**

注意：两种 CDB 不可混用！不同的设备有不同的要求！

SATA 对 SCSI 的封装： PACKET 命令

SATA 协议使用 PACKET 命令，对 SCSI 命令进行封装，而后由 SATA 控制器使用 SCSI 协议进行转发。



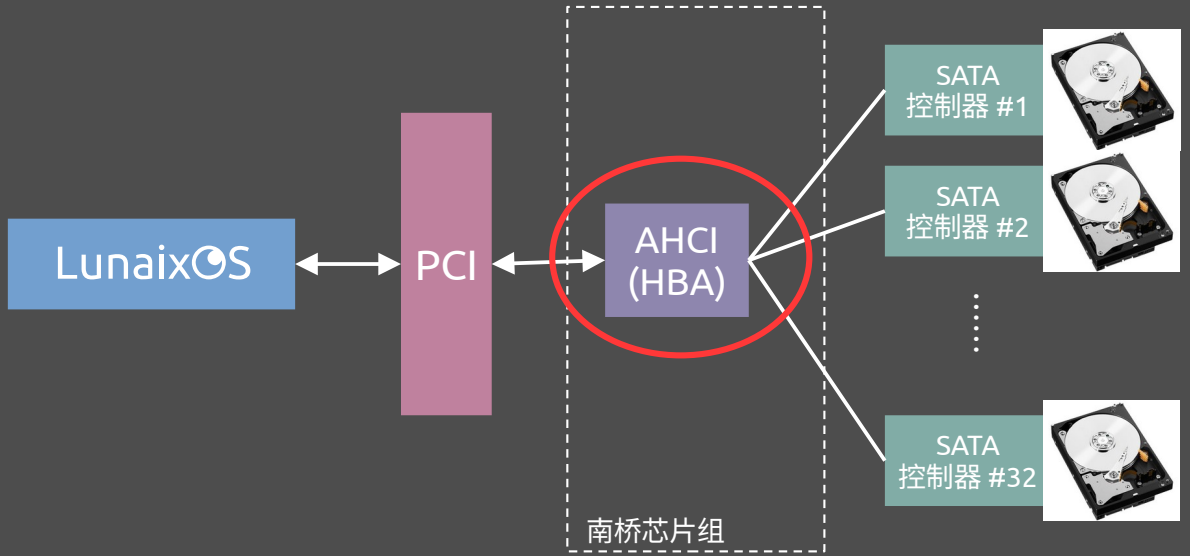
剩余的步骤取决于 SCSI 命令类型， DMA 读取或是写入，和前面一样

感觉和 SATA 控制器通讯太复杂了……

的确是这样的，但这也是早期操作系统必须要克服的困难！

也因为如此， Intel 开发了 AHCI 协议，将操作系统从底层的通讯协议中抽象了出来。允许开发者从更高级的层面去操作。

我们将会在下一期中介绍如何对 AHCI 进行编程



```
c010606f: 55                push   %ebp
c0106070: 89 e5             mov    %esp,%ebp
c0106072: 56                push   %esi
c0106073: 53                push   %ebx
c0106074: 83 ec 0c          sub    $0xc,%esp
c0106077: 68 44 a2 12 c0    push   $0xc012a24
c010607c: e8 ed 1c 00 00    call   c0107d6e <
c0106081: be 00 60 10 00    mov    $0x106000,
c0106086: c1 ee 0c          shr    $0xc,%esi
c0106089: 83 c4 08          add    $0x8,%esp
c010608c: 56                push   %esi
c010608d: 68 6c a2 12 c0    push   $0xc012a26c
c0106092: e8 d7 1c 00 00    call   c0107d6e <
c0106097: 83 c4 10          add    $0x1
c010609a: bb 00 00 00 00    mov    $0x
c010609f: eb 14             jmp    c01060b5 <_kernel_post_init+0x46>
c01060a1: 89 d8             mov    %ebx,%eax
c01060a3: c1 e0 0c          shl    $0xc,%eax
c01060a6: 83 ec 0c          sub    $0xc,%esp
c01060a9: 50                push   %eax
c01060aa: e8 9f 0a 00 00    call   c0106b4e <vmm_unmap_page>
c01060af: 83 c
c01060b2: 83 c
c01060b5: 39 f3
c01060b7: 72 e8
c01060b9: e8 54
c01060be: 85 c
c01060c0: 74 07
c01060c2: 8d 65 f8
c01060c5: 5b
c01060c6: 5e
c01060c7: 5d
c01060c8: c3
c01060c9: 83 ec 04          $0x4,%esp
c01060cc: 6a 40             $0x40
c01060ce: 68 af a0 12 c0    push   $0xc012a0af
```

LunaixOS

SERIAL ATA

从零开始

自制操作系统

磁盘与 SATA

对 AHCI 进行编程

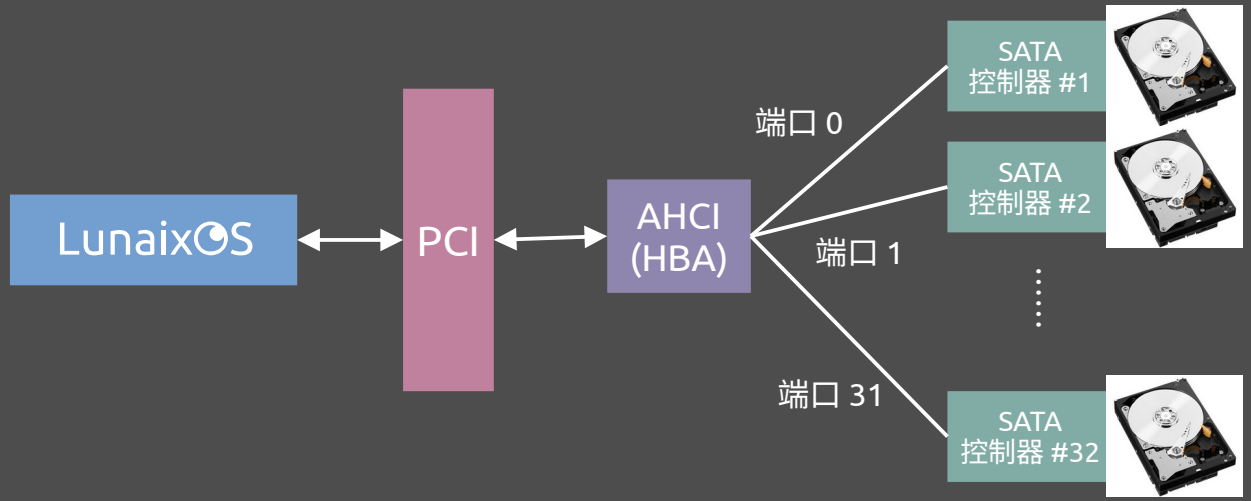
EP 13-2

什么是 AHCI (HBA)

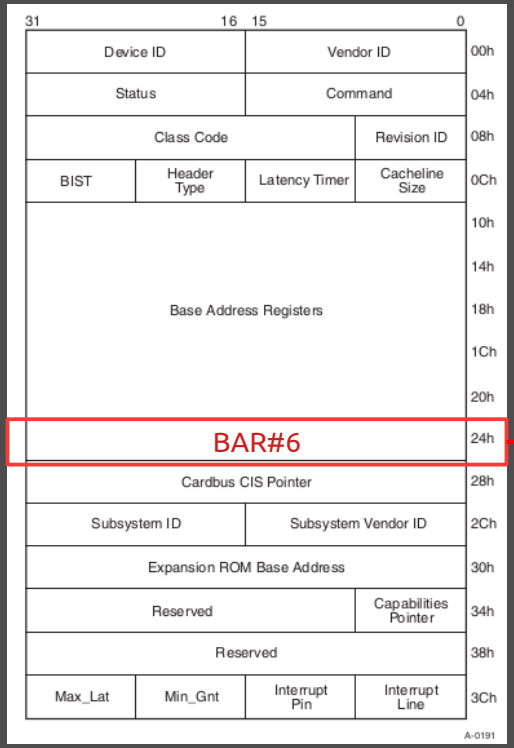
一个单独的芯片，至少可带 32 个 SATA 控制器。

是对 SATA 协议的一个更高层抽象。

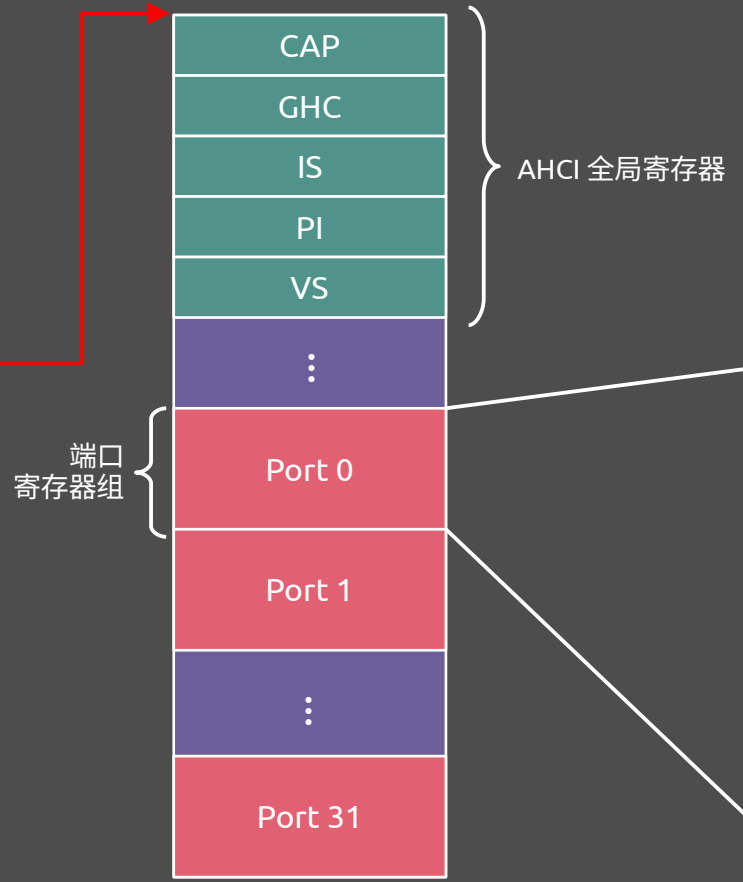
每个 SATA 控制器对应到 AHCI 的一个**端口**上。



AHCI 和他的寄存器们



PCI 配置空间



Start	End	Symbol	Description
00h	03h	CAP	Host Capabilities
04h	07h	GHC	Global Host Control
08h	0Bh	IS	Interrupt Status
0Ch	0Fh	PI	Ports Implemented
10h	13h	VS	Version
14h	17h	CCC_CTL	Command Completion Coalescing Control
18h	1Bh	CCC_PORTS	Command Completion Coalsecing Ports
1Ch	1Fh	EM_LOC	Enclosure Management Location
20h	23h	EM_CTL	Enclosure Management Control
24h	27h	CAP2	Host Capabilities Extended
28h	2Bh	BOHC	BIOS/OS Handoff Control and Status

Start	End	Symbol	Description
00h	03h	PxCLB	Port x Command List Base Address
04h	07h	PxCLBU	Port x Command List Base Address Upper 32-Bits
08h	0Bh	PxFB	Port x FIS Base Address
0Ch	0Fh	PxFBU	Port x FIS Base Address Upper 32-Bits
10h	13h	PxIS	Port x Interrupt Status
14h	17h	PxIE	Port x Interrupt Enable
18h	1Bh	PxCMD	Port x Command and Status
1Ch	1Fh	Reserved	Reserved
20h	23h	PxTFD	Port x Task File Data
24h	27h	PxSIG	Port x Signature
28h	2Bh	PxSSTS	Port x Serial ATA Status (SCR0: SStatus)
2Ch	2Fh	PxSCTL	Port x Serial ATA Control (SCR2: SControl)
30h	33h	PxSERR	Port x Serial ATA Error (SCR1: SError)
34h	37h	PxSACT	Port x Serial ATA Active (SCR3: SActive)
38h	3Bh	PxCI	Port x Command Issue
3Ch	3Fh	PxSNTF	Port x Serial ATA Notification (SCR4: SNotification)
40h	43h	PxFBS	Port x FIS-based Switching Control
44h	47h	PxDEVSLP	Port x Device Sleep
48h	6Fh	Reserved	Reserved
70h	7Fh	PxVS	Port x Vendor Specific

相对于基地址的位移
（每个寄存器都是双字大小）

在 LunaixOS 中……

```
hba.base[HBA_RGHC] |= HBA_RGHC_ACHI_ENABLE;
hba.base[HBA_RGHC] |= HBA_RGHC_INTR_ENABLE;

// As per section 3.1.1, this is 0 based value.
hba_reg_t cap = hba.base[HBA_RCAP];
hba_reg_t pmap = hba.base[HBA_RPI];
```

AHCI 的所有寄存器表示为一个 uint32 的数组

```
#define HBA_RCAP 0
#define HBA_RGHC 1
#define HBA_RIS 2
#define HBA_RPI 3
#define HBA_RVER 4

#define HBA_RPBASE (0x40)
#define HBA_RPSIZE (0x80 >> 2)
#define HBA_RPxCLB 0
#define HBA_RPxFB 2
#define HBA_RPxIS 4
#define HBA_RPxIE 5
#define HBA_RPxCMD 6
#define HBA_RPxTFD 8
#define HBA_RPxSIG 9
#define HBA_RPxSSTS 10
#define HBA_RPxSCTL 11
#define HBA_RPxSERR 12
#define HBA_RPxSACT 13
#define HBA_RPxCI 14
#define HBA_RPxSNTF 15
#define HBA_RPxFBS 16
```

第一个端口寄存器组的起始位移（双字为单位）

每个端口寄存器组的大小（双字为单位）

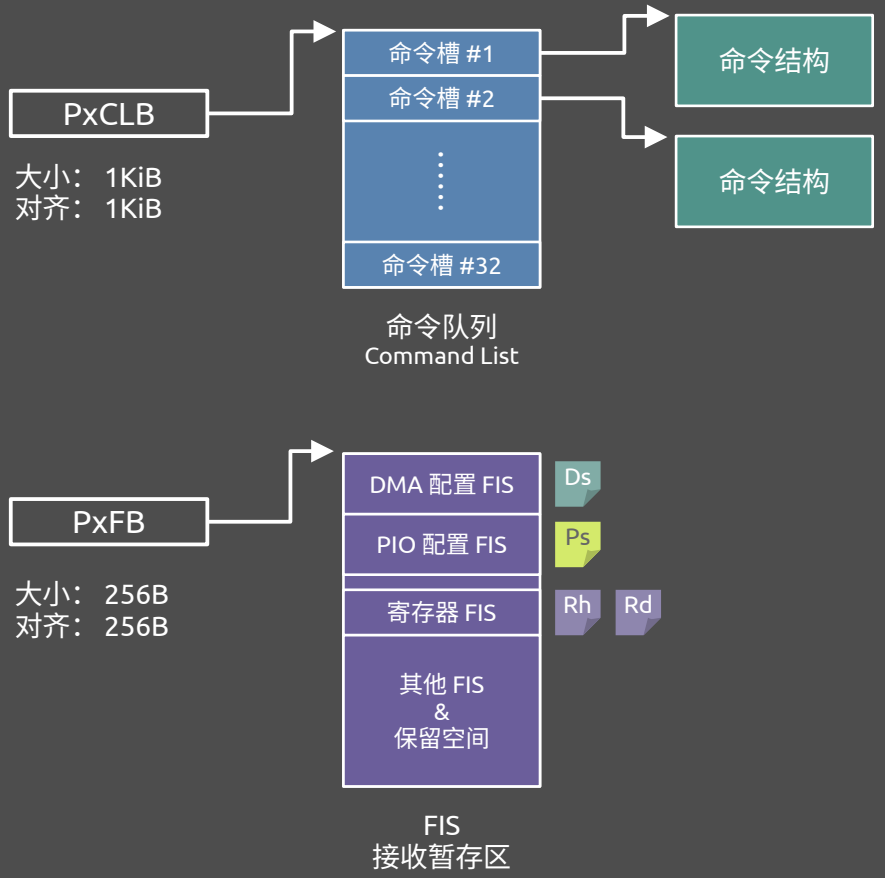
AHCI 与 SATA 的通讯

很简单，就是我们上期的全部流程！ AHCI 扮演着主机的角色

但是 AHCI 没有自己的内存，怎么办？

操作系统分配给他！

Start	End	Symbol	Description
00h	03h	PxCLB	Port x Command List Base Address
04h	07h	PxCLBU	Port x Command List Base Address Upper 32-Bits
08h	0Bh	PxFB	Port x FIS Base Address
0Ch	0Fh	PxFBU	Port x FIS Base Address Upper 32-Bits
10h	13h	PxIS	Port x Interrupt Status
14h	17h	PxIE	Port x Interrupt Enable
18h	1Bh	PxCMD	Port x Command and Status
1Ch	1Fh	Reserved	Reserved
20h	23h	PxTFD	Port x Task File Data
24h	27h	PxSIG	Port x Signature
28h	2Bh	PxSSTS	Port x Serial ATA Status (SCR0: SStatus)
2Ch	2Fh	PxSCTL	Port x Serial ATA Control (SCR2: SControl)
30h	33h	PxSERR	Port x Serial ATA Error (SCR1: SError)
34h	37h	PxSACT	Port x Serial ATA Active (SCR3: SActive)
38h	3Bh	PxCI	Port x Command Issue
3Ch	3Fh	PxSNTF	Port x Serial ATA Notification (SCR4: SNotification)
40h	43h	PxFBS	Port x FIS-based Switching Control
44h	47h	PxDEVSLP	Port x Device Sleep
48h	6Fh	Reserved	Reserved
70h	7Fh	PxVS	Port x Vendor Specific



AHCI 的初始化工作

AHCI 的标准要求软件至少执行以下几个步骤（10.1.2），以完成 AHCI 的初始化

1. GHC 寄存器的 AE 位置位，表明我们使用的是 AHCI 模式，而不是 IDE 兼容模式。

2. 读取 PI 寄存器，找出所有已开启的端口

3. 读取 CAP 寄存器的 NCS 字段，找出每个端口的命令队列可用的长度（从 0 开始算）。

对于每一个端口：

    1. 进行端口重置

    2. 分配操作空间，设置 PxCLB 和 PxFB

    3. 将 PxSERR 寄存器清空

    4. 将需要使用的中断通过 PxIE 寄存器开启

4. 最后将 GHC 的 IE 位置位，从而使得 HBA 可以向 CPU 发送中断

准备工作：映射寄存器到虚拟内存

通过读取 PCI 配置空间的 BAR#6 寄存器获取 AHCI 寄存器的物理基地址。

可是我们还需要知道这个映射的范围。

```c
size_t
pci_bar_sizing(struct pci_device* dev, uint32_t* bar_out, uint32_t bar_num)
{
    pci_reg_t bar = pci_read_cspace(dev->cspace_base, PCI_REG_BAR(bar_num));
    if (!bar) {
        *bar_out = 0;
        return 0;
    }

    pci_write_cspace(dev->cspace_base, PCI_REG_BAR(bar_num), 0xffffffff);
    pci_reg_t sized =
      pci_read_cspace(dev->cspace_base, PCI_REG_BAR(bar_num)) & ~0x1;
    if (PCI_BAR_MMIO(bar)) {
        sized = PCI_BAR_ADDR_MM(sized);
    }
    *bar_out = bar;
    pci_write_cspace(dev->cspace_base, PCI_REG_BAR(bar_num), bar);
    return ~sized + 1;
}
```

IMPLEMENTATION NOTE

Sizing a 32-bit Base Address Register Example

Decode (I/O or memory) of a register is disabled via the command register before sizing a Base Address register. Software saves the original value of the Base Address register, writes 0 FFFF FFFFh to the register, then reads it back. Size calculation can be done from the
read by first clearing encoding information bits (bit 0 for I/O, bits 0-3 for
verting all 32 bits (logical NOT), then incrementing by 1. The resultant 32-bit
memory/I/O range size decoded by the register. Note that the upper 16 bits of
ignored if the Base Address register is for I/O and bits 16-31 returned zero
The original value in the Base Address register is restored before re-enabling
e command register of the device.

PCI 标准里面告诉了我们应该如何去做

准备工作：映射寄存器到虚拟内存

然后就是在内核空间寻找到一个合适虚拟地址范围，将其映射过去

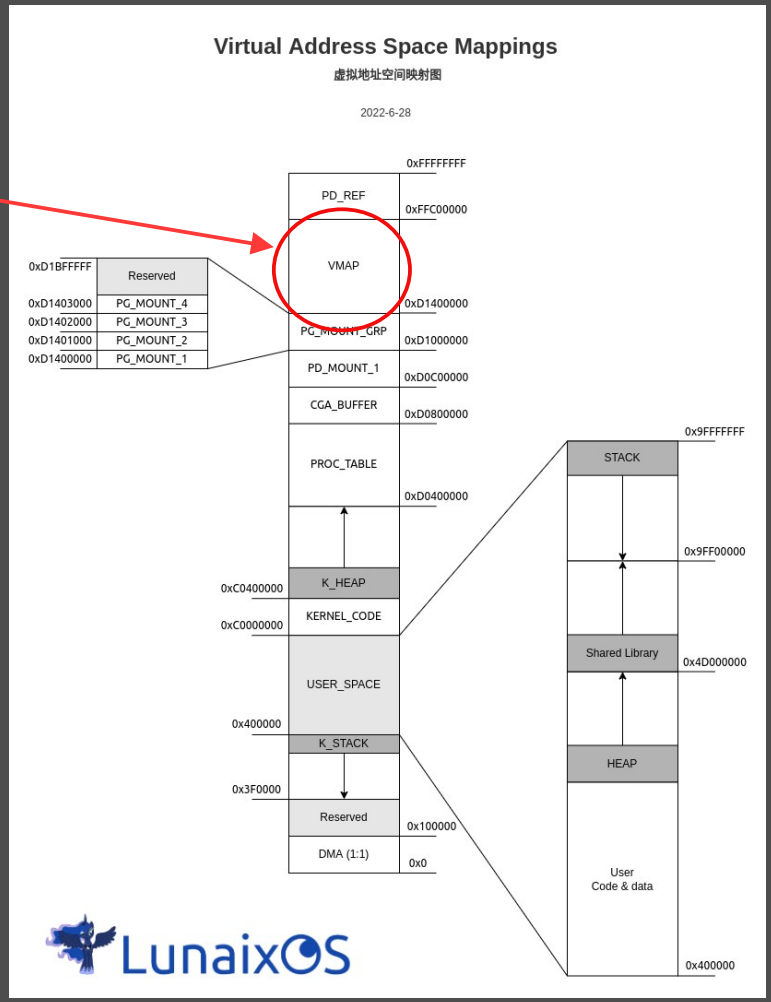
我们通过在这个范围内遍历页表，使用 Next Fit 策略找到一系列可用的页表项

我们将会在这个区域内进行寻找

```
while (!wrapped || current_addr ≥ start) {
    size_t l1inx = L1_INDEX(current_addr);
    if (!(pd->entry[l1inx])) {
        // empty 4mb region
        examed_size += MEM_4MB;
        current_addr = (current_addr & 0xffc00000) + MEM_4MB;
    } else {
        x86_page_table* ptd = (x86_page_table*)(L2_VADDR(l1inx));
        size_t i = L2_INDEX(current_addr);
        for (; i < PG_MAX_ENTRIES && examed_size < size; i++) {
            if (!ptd->entry[i]) {
                examed_size += PG_SIZE;
            } else if (examed_size) {
                // found a discontinuity, start from beginning
                examed_size = 0;
                i++;
                break;
            }
        }
        current_addr += i << 12;
    }

    if (examed_size ≥ size) {
        goto done;
    }

    if (current_addr ≥ VMAP_END) {
        wrapped = 1;
        current_addr = VMAP_START;
    }
}
```

封装进 ioremap 函数

```
void*
ioremap(uintptr_t paddr, uint32_t size)
{
    return vmm_vmap(paddr, size, PG_PREM_RW | PG_DISABLE_CACHE);
}
```



准备工作：映射寄存器到虚拟内存

0x10601

```
struct pci_device* ahci_dev = pci_get_device_by_class(AHCI_HBA_CLASS);
assert_msg(ahci_dev, "AHCI: Not found.");

uintptr_t bar6, size;
size = pci_bar_sizing(ahci_dev, &bar6, 6);
assert_msg(bar6 && PCI_BAR_MMIO(bar6), "AHCI: BAR#6 is not MMIO.");

pci_reg_t cmd = pci_read_cspace(ahci_dev->cspace_base, PCI_REG_STATUS_CMD);

// 禁用传统中断（因为我们使用MSI），启用MMIO访问，允许PCI设备间访问
cmd |= (PCI_RCMD_MM_ACCESS | PCI_RCMD_DISABLE_INTR | PCI_RCMD_BUS_MASTER);

pci_write_cspace(ahci_dev->cspace_base, PCI_REG_STATUS_CMD, cmd);

pci_setup_msi(ahci_dev, AHCI_HBA_IV);
intr_subscribe(AHCI_HBA_IV, __ahci_hba_isr);

memset(&hba, 0, sizeof(hba));

hba.base = (hba_reg_t*)ioremap(PCI_BAR_ADDR_MM(bar6), size);
```

初始化 AHCI： 全局寄存器配置

```
// 启用AHCI工作模式，启用中断
hba.base[HBA_RGHC] |= HBA_RGHC_ACHI_ENABLE;
hba.base[HBA_RGHC] |= HBA_RGHC_INTR_ENABLE;

// As per section 3.1.1, this is 0 based value.
hba_reg_t cap = hba.base[HBA_RCAP];
hba_reg_t pmap = hba.base[HBA_RPI];

hba.ports_num = (cap & 0x1f) + 1;  // CAP.PI
hba.cmd_slots = (cap >> 8) & 0x1f; // CAP.NCS
hba.version = hba.base[HBA_RVER];
hba.ports_bmp = pmap;
```

GHC

Bit	Type	Reset	Description
31	RW/RO	Impl Spec	**AHCI Enable (AE):** When set, indicates that communication to the HBA shall be via AHCI mechanisms. This can be used by an HBA that supports both legacy mechanisms (such as SFF-8038i) and AHCI to know when the HBA is running under an AHCI driver. When set, software shall only communicate with the HBA using AHCI. When cleared, software shall only communicate with the HBA using legacy mechanisms. When cleared FISes are not posted to memory and no commands are sent via AHCI mechanisms. Software shall set this bit to '1' before accessing other AHCI registers. When software clears this bit to '0' from a previous value of '1', it shall set no other bit in the GHC register as part of that operation (i.e., clearing the AE bit requires software to write 00000000h to the register). The implementation of this bit is dependent upon the value of the CAP.SAM bit. If CAP.SAM is '0', then GHC.AE shall be read-write and shall have a reset value of '0'. If CAP.SAM is '1', then AE shall be read-only and shall have a reset value of '1'.
01	RW	0	**Interrupt Enable (IE):** This global bit enables interrupts from the HBA. When cleared (reset default), all interrupt sources from all ports are disabled. When set, interrupts are enabled.

CAP

12:08	RO	Impl Spec	**Number of Command Slots (NCS):** 0's based value indicating the number of command slots per port supported by this HBA. A minimum of 1 and maximum of 32 slots per port can be supported. The same number of command slots is available on each implemented port.
04:00	RO	Impl Spec	**Number of Ports (NP):** 0's based value indicating the maximum number of ports supported by the HBA silicon. A maximum of 32 ports can be supported. A value of '0h', indicating one port, is the minimum requirement. Note that the number of ports indicated in this field may be more than the number of ports indicated in the PI register.

主板上一共有多少个 SATA 口 →

PI

31:0	RO	HwInit	**Port Implemented (PI):** This register is bit significant. If a bit is set to '1', the corresponding port is available for software to use. If a bit is cleared to '0', the port is not available for software to use. The maximum number of bits set to '1' shall not exceed CAP.NP + 1, although the number of bits set in this register may be fewer than CAP.NP + 1. At least one bit shall be set to '1'.

一个位图，指示那些 SATA 口是接着硬盘的 →

初始化 AHCI： 端口专项配置

```
        struct hba_port* port =
          (struct hba_port*)valloc(sizeof(struct hba_port));
        hba_reg_t* port_regs =
          (hba_reg_t*)(&hba.base[HBA_RPBASE + i * HBA_RPSIZE]);

#ifndef DO_HBA_FULL_RESET
        __hba_reset_port(port_regs);
#endif

        if (!clbp) {
            // 每页最多4个命令队列
            clb_pa = pmm_alloc_page(KERNEL_PID, PP_FGLOCKED);
            clb_pg_addr = ioremap(clb_pa, 0x1000);
            memset(clb_pg_addr, 0, 0x1000);
        }
        if (!fisp) {
            // 每页最多16个FIS
            fis_pa = pmm_alloc_page(KERNEL_PID, PP_FGLOCKED);
            fis_pa = ioremap(fis_pa, 0x1000);
            memset(fis_pg_addr, 0, 0x1000);
        }

        /* 重定向CLB与FIS */
        port_regs[HBA_RPxCLB] = clb_pa + clbp * HBA_CLB_SIZE;
        port_regs[HBA_RPxFB] = fis_pa + fisp * HBA_FIS_SIZE;

        *port = (struct hba_port){ .regs = port_regs,
                                   .ssts = port_regs[HBA_RPxSSTS],
                                   .cmdlst = clb_pg_addr + clbp * HBA_CLB_SIZE,
                                   .fis = fis_pg_addr + fisp * HBA_FIS_SIZE };
```

计算出端口寄存器组的起始地址

重置端口

命令和 FIS 区都分别分配专门的物理页，保证对齐 .

每个物理页最多可划分出：

4 个命令队列

或

16 个 FIS 暂存区

更新 PxCLB 和 PxFB

（注意，必须为物理地址）

初始化 AHCI： 端口专项配置

```
/* 初始化端口，并置于就绪状态 */
port_regs[HBA_RPxCI] = 0;

// 需要通过全部置位去清空这些寄存器（相当的奇怪……）
port_regs[HBA_RPxSERR] = -1;

port_regs[HBA_RPxIE] |= (HBA_PxINTR_D2HR);

hba.ports[i] = port;

if (!HBA_RPxSSTS_IF(port->ssts)) {
    continue;
}
```

```
wait_until(!(port_regs[HBA_RPxCMD] & HBA_PxCMD_CR));
port_regs[HBA_RPxCMD] |= HBA_PxCMD_FRE;
port_regs[HBA_RPxCMD] |= HBA_PxCMD_ST;
```

确保当前端口的 DMA 控制器处在关闭状态

然后再通过 FRE 和 ST 置位，开启 Port

初始化 AHCI： 端口重置

```c
void
__hba_reset_port(hba_reg_t* port_reg)
{
    // 根据：SATA-AHCI spec section 10.4.2 描述的端口重置流程
    port_reg[HBA_RPxCMD] &= ~HBA_PxCMD_ST;
    port_reg[HBA_RPxCMD] &= ~HBA_PxCMD_FRE;
    int cnt = wait_until_expire(!(port_reg[HBA_RPxCMD] & HBA_PxCMD_CR), 500000);
    if (cnt) {
        return;
    }
    // 如果port未响应，则继续执行重置
    port_reg[HBA_RPxSCTL] = (port_reg[HBA_RPxSCTL] & ~0xf) | 1;
    io_delay(100000); //等待至少一毫秒，差不多就行了
    port_reg[HBA_RPxSCTL] &= ~0xf;
}
```

10.4.2 Port Reset

If a port is not functioning properly after a software reset, software may attempt to re-initialize communication with the port via a COMRESET. It must first clear PxCMD.ST, and wait for PxCMD.CR to clear to '0' before re-initializing communication. However, if PxCMD.CR does not clear within a reasonable time (500 milliseconds), it may assume the interface is in a hung condition and may continue with issuing the port reset.

Software causes a port reset (COMRESET) by writing 1h to the PxSCTL.DET field to invoke a COMRESET on the interface and start a re-establishment of Phy layer communications. Software shall wait at least 1 millisecond before clearing PxSCTL.DET to 0h; this ensures that at least one COMRESET signal is sent over the interface. After clearing PxSCTL.DET to 0h, software should wait for communication to be re-established as indicated by PxSSTS.DET being set to 3h. Then software should write all 1s to the PxSERR register to clear any bits that were set as part of the port reset.

PxCMD

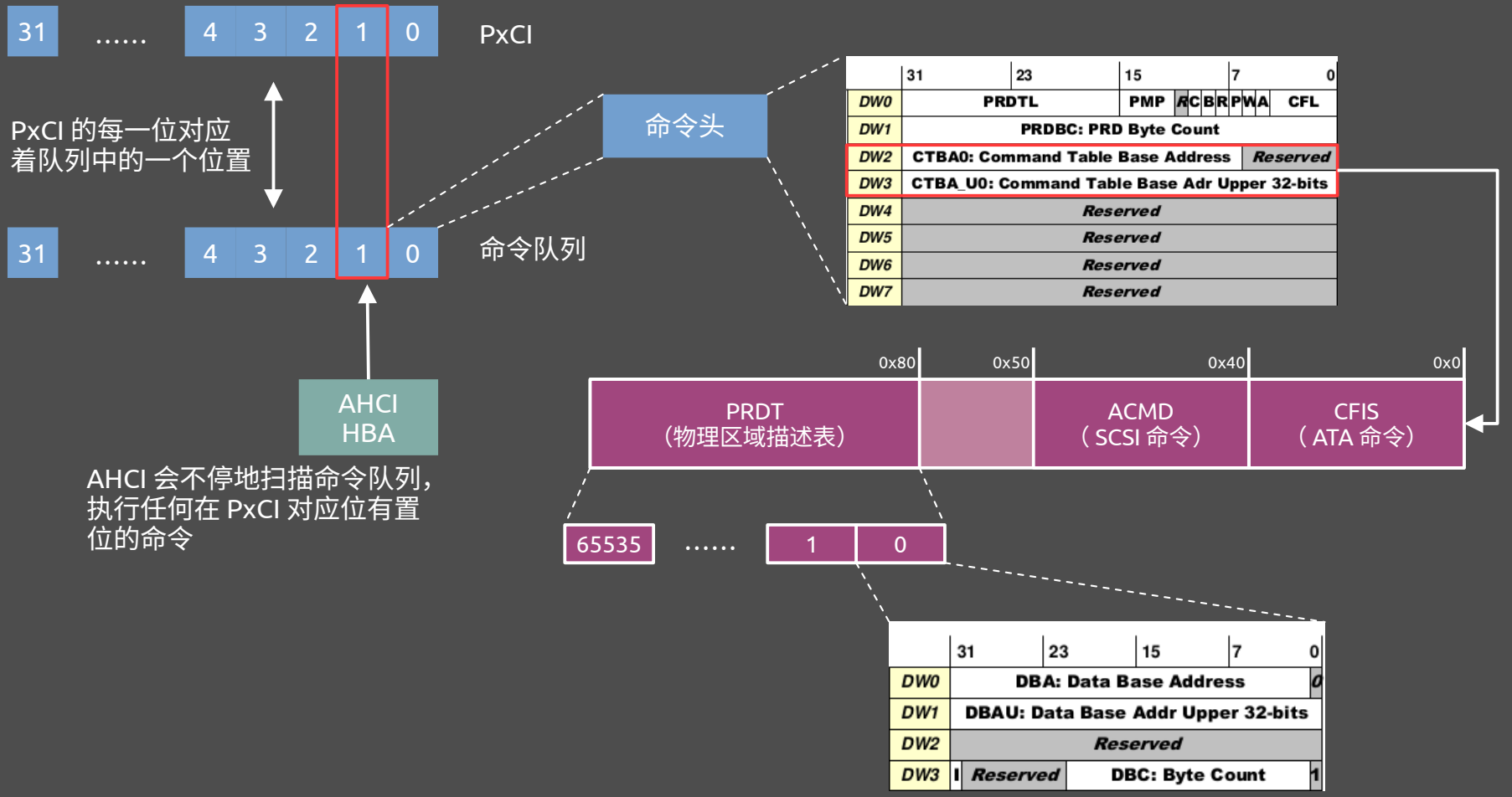
00	RW	0	**Start (ST):** When set, the HBA may process the command list. When cleared, the HBA may not process the command list. Whenever this bit is changed from a '0' to a '1', the HBA starts processing the command list at entry '0'. Whenever this bit is changed from a '1' to a '0', the PxCI register is cleared by the HBA upon the HBA putting the controller into an idle state. This bit shall only be set to '1' by software after PxCMD.FRE has been set to '1'. Refer to section 10.3.1 for important restrictions on when ST can be set to '1'.
04	RW	0	**FIS Receive Enable (FRE):** When set, the HBA may post received FISes into the FIS receive area pointed to by PxFB (and for 64-bit HBAs, PxFBU). When cleared, received FISes are not accepted by the HBA, except for the first D2H register FIS after the initialization sequence, and no FISes are posted to the FIS receive area. System software must not set this bit until PxFB (PxFBU) have been programmed with a valid pointer to the FIS receive area, and if software wishes to move the base, this bit must first be cleared, and software must wait for the FR bit in this register to be cleared. Refer to section 10.3.2 for important restrictions on when FRE can be set and cleared.

PxSCTL

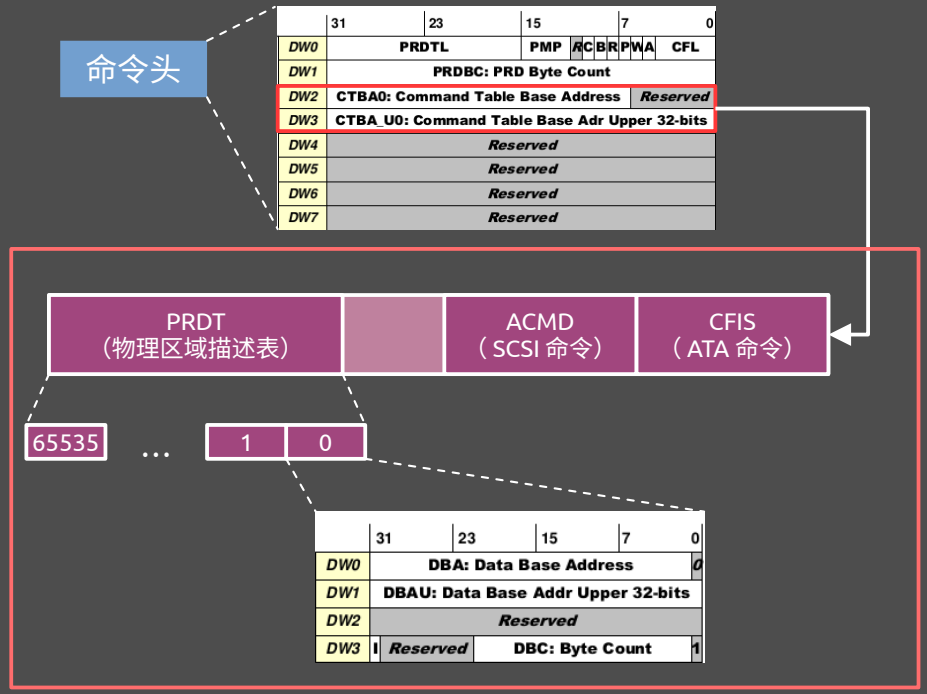
03:00	RW	0h	**Device Detection Initialization (DET):** Controls the HBA's device detection and interface initialization. 0h No device detection or initialization action requested 1h Perform interface communication initialization sequence to establish communication. This is functionally equivalent to a hard reset and results in the interface being reset and communications reinitialized. While this field is 1h, COMRESET is transmitted on the interface. Software should leave the DET field set to 1h for a minimum of 1 millisecond to ensure that a COMRESET is sent on the interface. 4h Disable the Serial ATA interface and put Phy in offline mode. All other values reserved This field may only be modified when PxCMD.ST is '0'. Changing this field while the PxCMD.ST bit is set to '1' results in undefined behavior. When PxCMD.ST is set to '1', this field should have a value of 0h. Note: It is permissible to implement any of the Serial ATA defined behaviors for transmission of COMRESET when DET=1h.

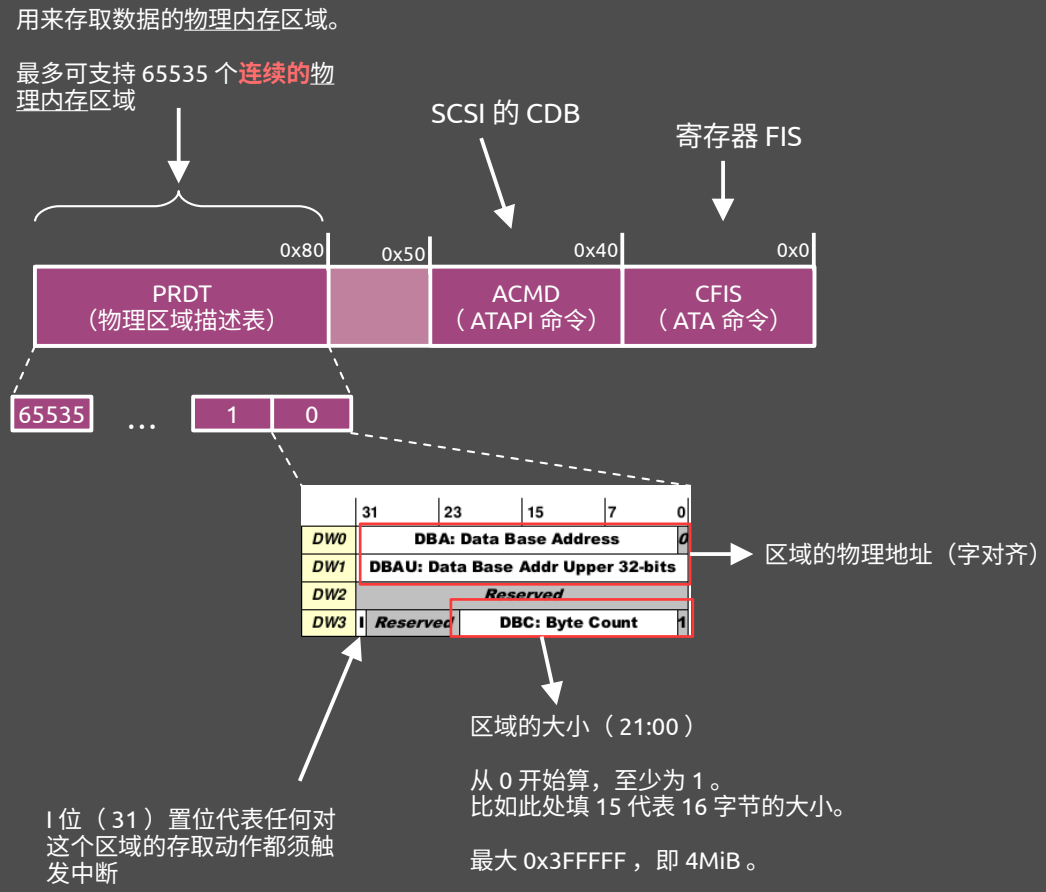
尝试向设备发送一个命令吧！

AHCI 的发令机制总览：

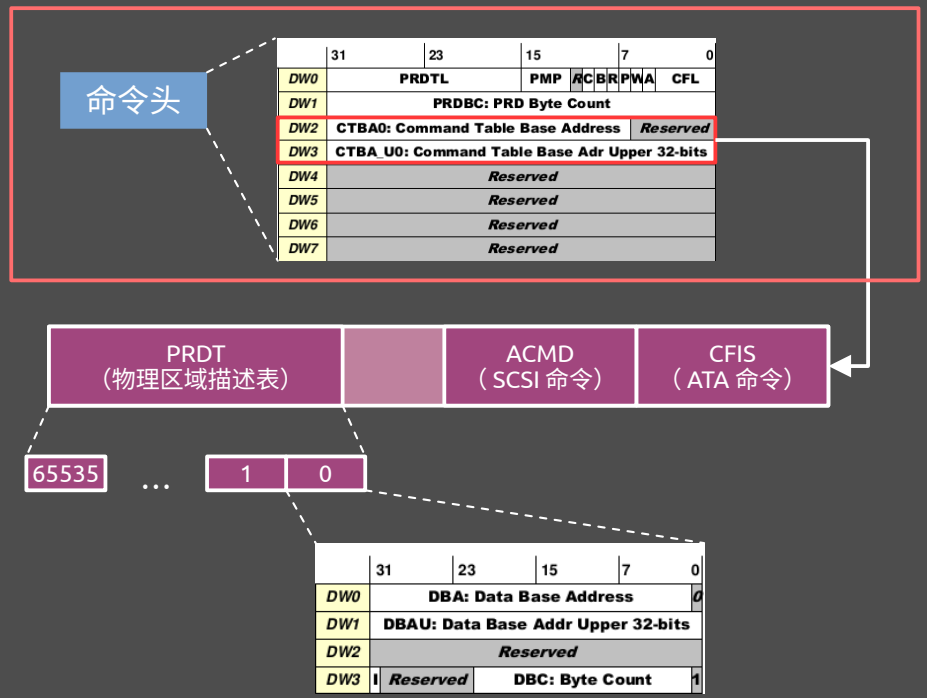


尝试向设备发送一个命令吧！



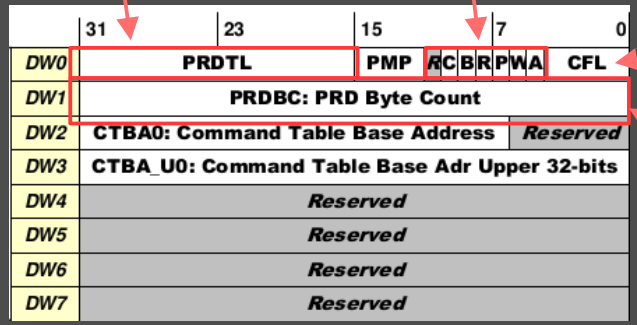


尝试向设备发送一个命令吧！



A： 该命令需要发往 ATAPI 设备
P： 物理区域可缓存
W： 该命令需要向设备写入数据
C： 命令执行完成后，需要将 PxTFD 的 BSY 位清零

描述表长度



命令 FIS 大小（双字为单位）

当前已传输的字节数量

尝试向设备发送一个命令吧!

事实上，在 AHCI 标准中（ 5.5.1 ），已经非常详细的阐明了向设备发送命令全部流程

1. 在命令队列中找到一个空闲的位置
2. 原地构建一个命令头，根据需求设置好各类字段
3. 分配一个**连续的物理空间**作为命令表。
4. 根据需求设定好命令表（设置 FIS ，设置 PRDT ）
5. 将命令表的**物理地址**挂载到命令头上
6. 将 PxCI 的对应位置位。
7. 等待，直到 PxCI 的对应位清零。这表明命令执行完成。
8. 读取 PxTFD 寄存器，获取命令执行结果，以及错误代码（如果发生错误）

动态的分配

现有的 `lxmalloc` 仅仅保证了虚拟地址的连续

我们需要一个新的内存分配器.....

```
c010606f: 55                    push   %ebp
c0106070: 89 e5                 mov    %esp,%ebp
c0106072: 56                    push   %esi
c0106073: 53                    push   %ebx
c0106074: 83 ec 0c              sub    $0xc,%esp
c0106077: 68 44 a2 12 c0        push   $0xc012a24
c010607c: e8 ed 1c 00 00        call   c0107d6e <
c0106081: be 00 60 10 00        mov    $0x106000,
c0106086: c1 ee 0c              shr    $0xc,%esi
c0106089: 83 c4 08              add    $0x8,%esp
c010608c: 56                    push   %esi
c010608d: 68 6c a2 12 c0        push   $0xc012a26c
c0106092: e8 d7 1c 00 00        call   c0107d6e <
c0106097: 83 c4 10              add    $0x1
c010609a: bb 00 00 00 00        mov    $0x
c010609f: eb 14                 jmp    c01060b5 <_kernel_post_init+0x46>
c01060a1: 89 d8                 mov    %ebx,%eax
c01060a3: c1 e0 0c              shl    $0xc,%eax
c01060a6: 83 ec 0c              sub    $0xc,%esp
c01060a9: 50                    push   %eax
c01060aa: e8 9f 0a 00 00        call   c0106b4e <vmm_unmap_page>
c01060af: 83 c
c01060b2: 83 c
c01060b5: 39 f3
c01060b7: 72 e8                                  1 <_kernel_post_init+0x32>
c01060b9: e8 54                                  <kalloc_init>
c01060be: 85 c
c01060c0: 74 07                                  nel_post_init+0x5a>
c01060c2: 8d 65 f
c01060c5: 5b
c01060c6: 5e                    p      %esi
c01060c7: 5d                           %ebp
c01060c8: c3
c01060c9: 83 ec 04              sub    $0x4,%esp
c01060cc: 6a 40                 p      $0x40
c01060ce: 68 af a0 12 c0        push   $0xc012a0af
```

LunaixOS

SERIAL ATA

从零开始

自制操作系统

磁盘与 SATA

蛋糕分配器

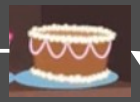
EP 13-3

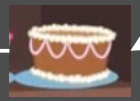
蛋……蛋糕分配器？

是的，分配蛋糕也是一门学问！

不同的小马喜欢不同的口味的蛋糕，并且要切成不同的大小

口味儿 #i





柜台 #1

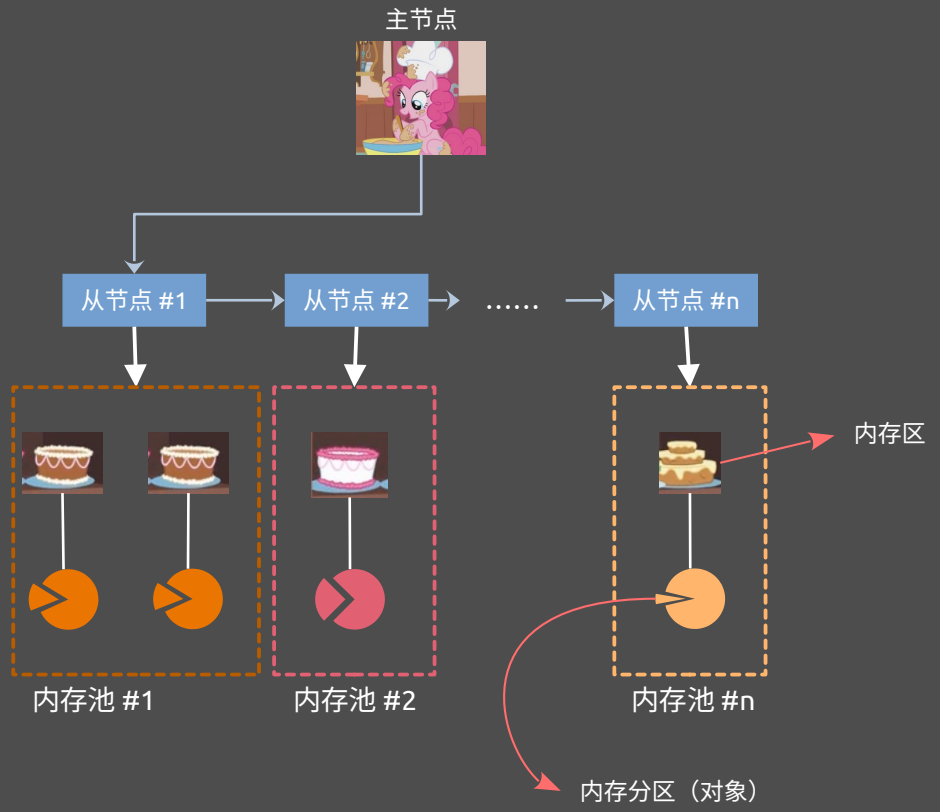
柜台 #2

口味儿 #j

柜台 #n

口味儿 #k

与内存的关系

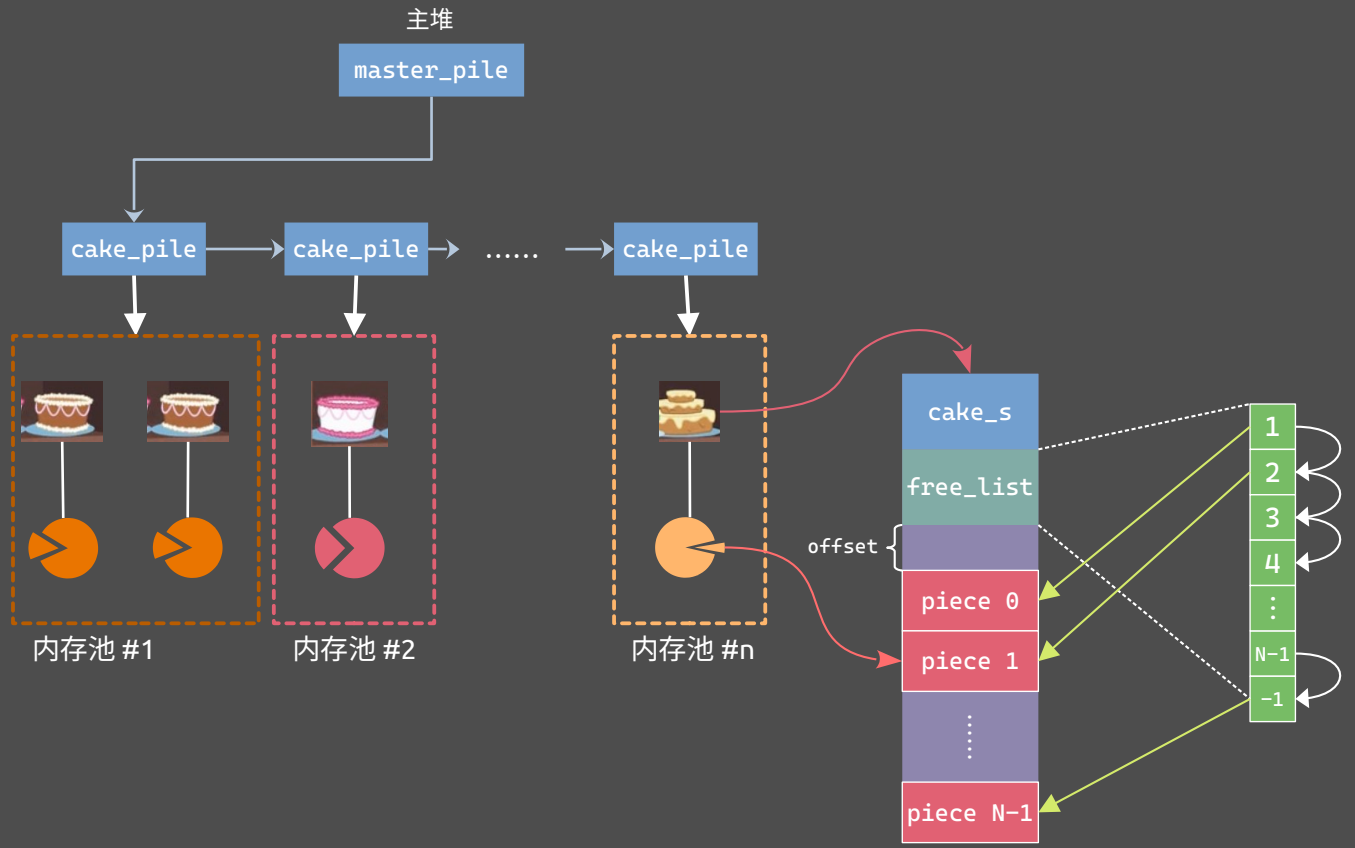


这正是大名鼎鼎的 SLAB 分配器！

在 LunaixOS 中，我们称之为蛋糕分配器。

毕竟和生硬的水泥预制板（ slab ）比起来，谁不喜欢蛋糕呢！

在 LunaixOS 中……


主堆
master_pile
cake_pile
cake_pile
.....
cake_pile
内存池 #1
内存池 #2
内存池 #n
cake_s
free_list
offset
piece 0
piece 1
piece N-1
1
2
3
4
N-1
-1

一个蛋糕的配方

每个蛋糕唯一的

第一块的指针

第一块的位移

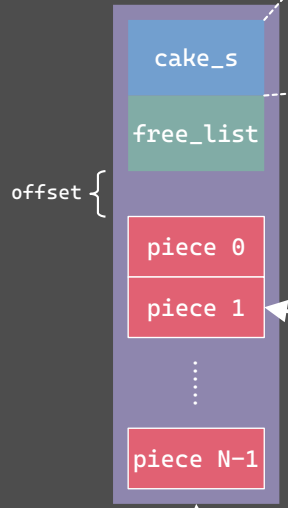
蛋糕切块的数量

蛋糕切块的大小

蛋糕占据物理页的数量

所有蛋糕共享的

```
struct cake_s
{
    struct llist_header cakes;
    void* first_piece;
    unsigned int used_pieces;

    piece_index_t free_list[0];
};
```

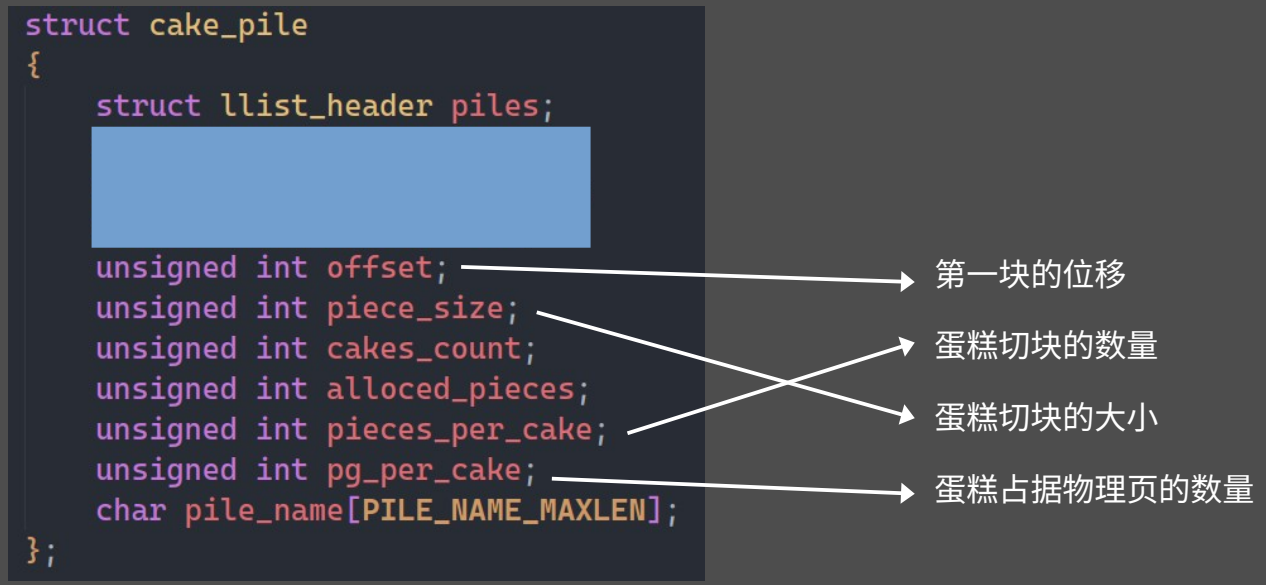


同一个蛋糕的每块儿大小等同

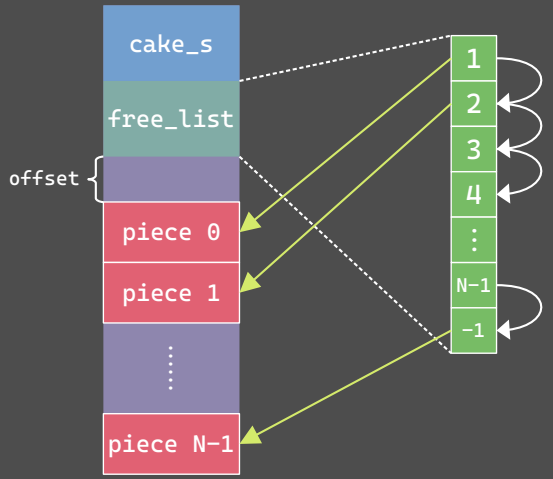
每个蛋糕占据至少一个**连续的物理页**

一堆蛋糕，统一的配方

```
struct cake_pile
{
    struct llist_header piles;

    unsigned int offset;
    unsigned int piece_size;
    unsigned int cakes_count;
    unsigned int alloced_pieces;
    unsigned int pieces_per_cake;
    unsigned int pg_per_cake;
    char pile_name[PILE_NAME_MAXLEN];
};
```



剩余的蛋糕切块在哪里？



free_list 为一个长度为 N 的数组。

每个元素的索引为蛋糕块儿的索引。

每个元素的值为另一个元素的索引。

-1 表示已到链表末尾

```
struct cake_s
{
    struct llist_header cakes;
    void* first_piece;
    unsigned int used_pieces;
    unsigned int next_free;
    piece_index_t free_list[0];
};
```

每个蛋糕会记录第一个空闲块儿的索引

→ next_free 永远指向空闲块链表的头部

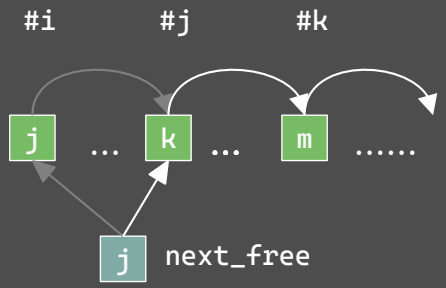
→ O(1) 的时间复杂度

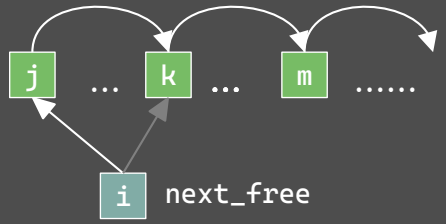
拿取一块儿蛋糕

```
piece_index_t found_index = pos->next_free;
pos->next_free = pos->free_list[found_index];
```

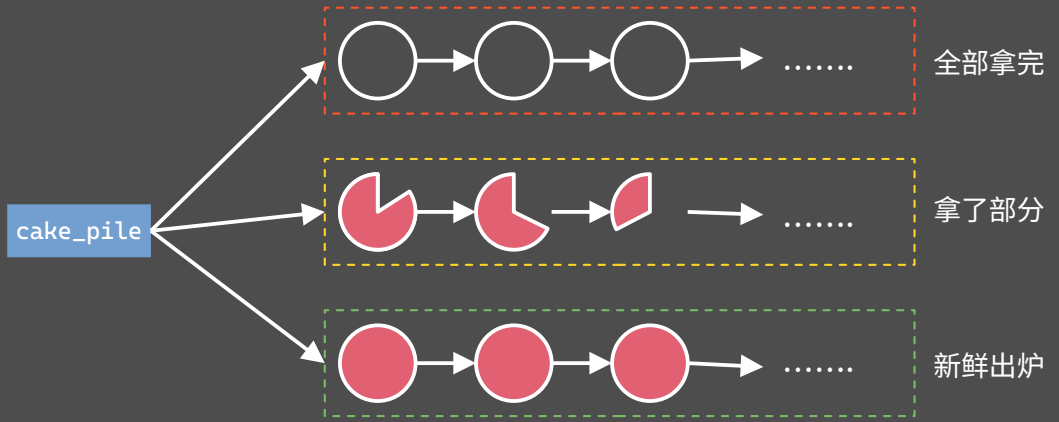
放回一块儿蛋糕

```
pos->free_list[piece_index] = pos->next_free;
pos->next_free = piece_index;
```





哪些蛋糕是有剩余的切块?




```
struct cake_pile
{
    struct llist_header piles;
    struct llist_header full;
    struct llist_header partial;
    struct llist_header free;
    unsigned int offset;
    unsigned int piece_size;
    unsigned int cakes_count;
    unsigned int alloced_pieces;
    unsigned int pieces_per_cake;
    unsigned int pg_per_cake;
    char pile_name[PILE_NAME_MAXLEN];
};
```

蛋糕块儿的拿取与归还

```
void*
cake_grab(struct cake_pile* pile)
{
    struct cake_s *pos, *n;
    if (!llist_empty(&pile->partial)) {
        pos = list_entry(pile->partial.next, typeof(*pos), cakes);
    } else if (llist_empty(&pile->free)) {
        pos = __new_cake(pile);
    } else {
        pos = list_entry(pile->free.next, typeof(*pos), cakes);
    }

    piece_index_t found_index = pos->next_free;
    pos->next_free = pos->free_list[found_index];
    pos->used_pieces++;
    pile->alloced_pieces++;

    llist_delete(&pos->cakes);
    if (pos->next_free == EO_FREE_PIECE) {
        llist_append(&pile->full, &pos->cakes);
    } else {
        llist_append(&pile->partial, &pos->cakes);
    }

    return (void*)((uintptr_t)pos->first_piece +
                   found_index * pile->piece_size);
}
```

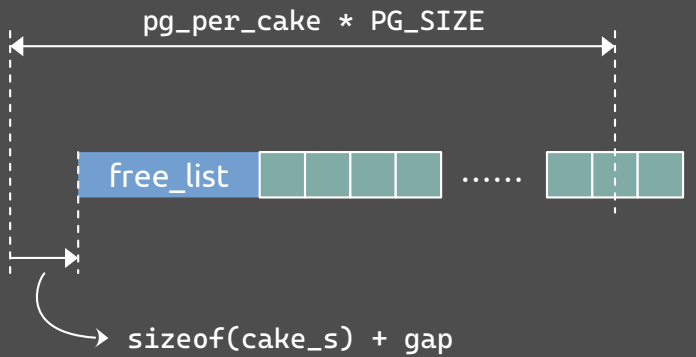
给定蛋糕堆，从中拿一块儿蛋糕
O(1) 时间！

```
int
cake_release(struct cake_pile* pile, void* area)
{
    piece_index_t piece_index;
    struct cake_s *pos, *n;
    struct llist_header* hdrs[2] = { &pile->full, &pile->partial };

    for (size_t i = 0; i < 2; i++) {
        llist_for_each(pos, n, hdrs[i], cakes)
        {
            if (pos->first_piece > area) {
                continue;
            }
            piece_index =
              (uintptr_t)(area - pos->first_piece) / pile->piece_size;
            if (piece_index < pile->pieces_per_cake) {
                goto found;
            }
        }
    }

    return 0;

found:
    pos->free_list[piece_index] = pos->next_free;
    pos->next_free = piece_index;
    pos->used_pieces--;
    pile->alloced_pieces--;

    llist_delete(pos);
    if (!pos->used_pieces) {
        llist_append(&pile->free, &pos->cakes);
    } else {
        llist_append(&pile->partial, &pos->cakes);
    }

    return 1;
}
```

将蛋糕归还到给定的蛋糕堆里
O(M+N) 时间……

让 Pinkie Pie 开始做蛋糕吧!

```
void
__init_pile(struct cake_pile* pile,
            char* name,
            unsigned int piece_size,
            unsigned int pg_per_cake,
            int options)
{
    unsigned int offset = sizeof(long);

    // 默认每块儿蛋糕对齐到地址总线宽度
    if ((options & PILE_CACHELINE)) {
        // 对齐到128字节缓存行大小，主要用于DMA
        offset = CACHE_LINE_SIZE;
    }

    piece_size = ROUNDUP(piece_size, offset);
    *pile = (struct cake_pile){ .piece_size = piece_size,
                                .cakes_count = 1,
                                .pieces_per_cake =
                                  (pg_per_cake * PG_SIZE) /
                                  (piece_size + sizeof(piece_index_t)),
                                .pg_per_cake = pg_per_cake };

    unsigned int free_list_size = pile->pieces_per_cake * sizeof(piece_index_t);

    pile->offset = ROUNDUP(sizeof(struct cake_s) + free_list_size, offset);
    pile->pieces_per_cake -= ICEIL((pile->offset - free_list_size), piece_size);

    strncpy(&pile->pile_name, name, PILE_NAME_MAXLEN);

    llist_init_head(&pile->free);
    llist_init_head(&pile->full);
    llist_init_head(&pile->partial);
    llist_append(&piles, &pile->piles);
}
```



让 Pinkie Pie 开始做蛋糕吧!

```
void*
__alloc_cake(unsigned int cake_pg)
{
    uintptr_t pa = pmm_alloc_cpage(KERNEL_PID, cake_pg, 0);
    return vmm_vmap(pa, cake_pg * PG_SIZE, PG_PREM_RW);
}

You, 2 weeks ago | 1 author (You)
struct cake_s*
__new_cake(struct cake_pile* pile)
{
    struct cake_s* cake = __alloc_cake(pile->pg_per_cake);

    int max_piece = pile->pieces_per_cake;

    cake->first_piece = (void*)((uintptr_t)cake + pile->offset);
    cake->next_free = 0;

    piece_index_t* free_list = &cake->free_list;
    for (size_t i = 0; i < max_piece - 1; i++) {
        free_list[i] = i + 1;
    }
    free_list[max_piece - 1] = EO_FREE_PIECE;

    llist_append(&pile->free, &cake->cakes);

    return cake;
}
```

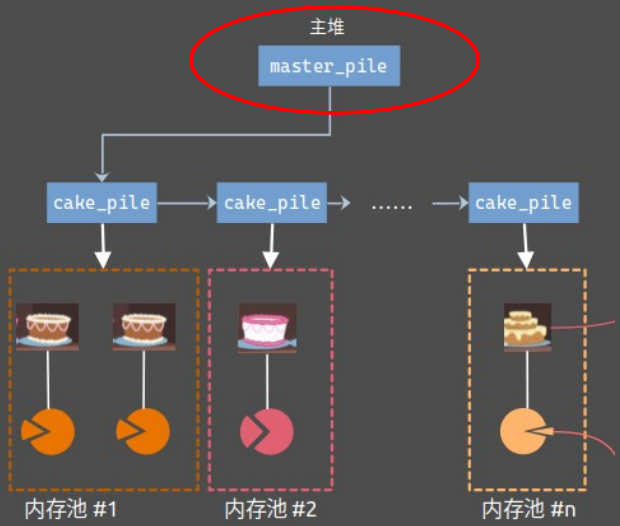
等一下，蛋糕堆们放在哪里？

做好了蛋糕，我们才能拿取一块儿作为内存。

→ 我们需要蛋糕堆来做蛋糕

→ 蛋糕堆需要一块儿内存来存放……

解决方案：我们需要有一个静态分配的**主堆**




```c
struct cake_pile master_pile;

void
cake_init()
{
    __init_pile(&master_pile, "pinkamina", sizeof(master_pile), 1, 0);
}
```

当创建新堆的时候，我们只需要直接从主堆中拿一块儿就好了！

```c
You, 2 weeks ago | 1 author (You)
struct cake_pile*
cake_new_pile(char* name,
              unsigned int piece_size,
              unsigned int pg_per_cake,
              int options)
{
    struct cake_pile* pile = (struct cake_pile*)cake_grab(&master_pile);

    __init_pile(pile, name, piece_size, pg_per_cake, options);

    return pile;
}
```

那么，蛋糕分配器有什么好处？

分配一个内存的使用常数时间（每个块儿位置固定）

内存分配现在是上下文敏感的：

1. 不同的堆可以自定义分配出的内存属性。
2. 更清晰的跟踪使用情况。

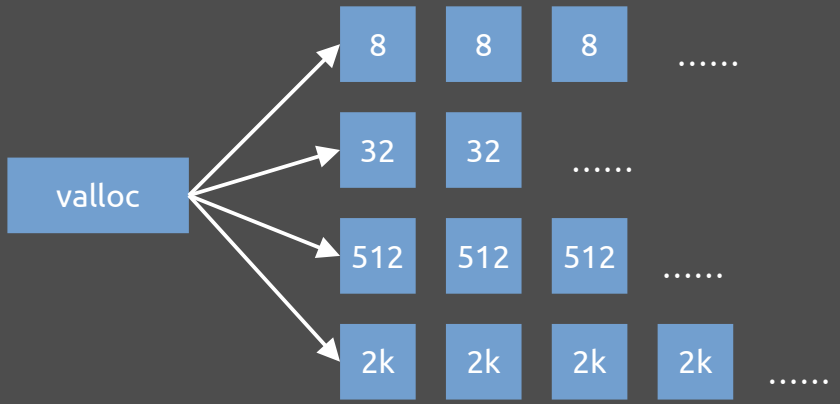
自动保证了物理与虚拟地址空间的连续性！

为实现一些更高级的内存管理技巧提供了捷径。

一个崭新的内存分配器

lxmalloc 采用传统简单的 explicit free list 方法组织空闲块。

valloc —— 采用 segregated free list 方法。



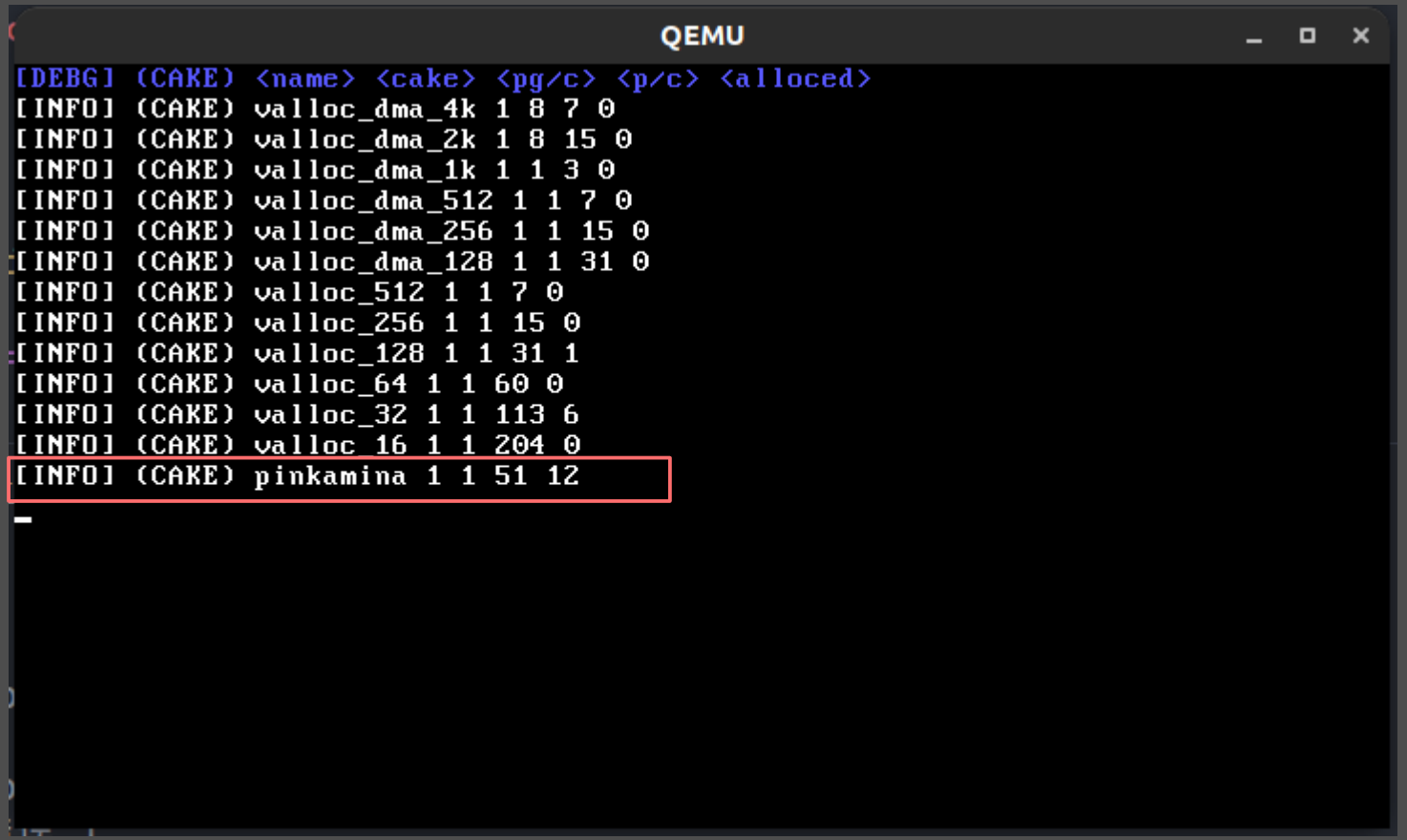

```c
static char piles_names[MAX_CLASS][PILE_NAME_MAXLEN] = {
    "valloc_16",  "valloc_32",  "valloc_64",
    "valloc_128", "valloc_256", "valloc_512"
};

static char piles_names_dma[MAX_CLASS][PILE_NAME_MAXLEN] = {
    "valloc_dma_128", "valloc_dma_256", "valloc_dma_512",
    "valloc_dma_1k",  "valloc_dma_2k",  "valloc_dma_4k"
};

static struct cake_pile* piles[MAX_CLASS];
static struct cake_pile* piles_dma[MAX_CLASS];

void
valloc_init()
{
    for (size_t i = 0; i < MAX_CLASS; i++) {
        int size = 1 << (i + 4);
        piles[i] = cake_new_pile(&piles_names[i], size, 1, 0);
    }

    // DMA 内存保证128字节对齐
    for (size_t i = 0; i < MAX_CLASS; i++) {
        int size = 1 << (i + 7);
        piles_dma[i] = cake_new_pile(
          &piles_names_dma[i], size, size > 1024 ? 8 : 1, PILE_CACHELINE);
    }
}
```

一个崭新的内存分配器

```
void*
__valloc(unsigned int size, struct cake_pile** segregate_list)
{
    size_t i = 0;
    for (; i < MAX_CLASS; i++) {
        if (segregate_list[i]→piece_size ≥ size) {
            goto found_class;
        }
    }

    return NULL;

found_class:
    return cake_grab(segregate_list[i]);
}
```

```
void
__vfree(void* ptr, struct cake_pile** segregate_list)
{
    size_t i = 0;
    for (; i < MAX_CLASS; i++) {
        if (cake_release(segregate_list[i], ptr)) {
            return;
        }
    }
}
```

运行效果


QEMU
[DEBG] (CAKE) <name> <cake> <pg/c> <p/c> <alloced>
[INFO] (CAKE) valloc_dma_4k 1 8 7 0
[INFO] (CAKE) valloc_dma_2k 1 8 15 0
[INFO] (CAKE) valloc_dma_1k 1 1 3 0
[INFO] (CAKE) valloc_dma_512 1 1 7 0
[INFO] (CAKE) valloc_dma_256 1 1 15 0
[INFO] (CAKE) valloc_dma_128 1 1 31 0
[INFO] (CAKE) valloc_512 1 1 7 0
[INFO] (CAKE) valloc_256 1 1 15 0
[INFO] (CAKE) valloc_128 1 1 31 1
[INFO] (CAKE) valloc_64 1 1 60 0
[INFO] (CAKE) valloc_32 1 1 113 6
[INFO] (CAKE) valloc_16 1 1 204 0
[INFO] (CAKE) pinkamina 1 1 51 12

我们下一步的任务……

在这一期里面，我们为 SATA 控制器专门实现了一个内存分类器。

由于其高效与灵活，我们将会在后面的开发中，将蛋糕分配器作为主力。

在下一期里，我们将会了解 ATA 设备以及 SCSI 协议中比较重要的指令，并且实现磁盘信息的识别与探测。

这将会为我们在下下期实现扇区读写，打下基础。

```
c010606f: 55                push   %ebp
c0106070: 89 e5             mov    %esp,%ebp
c0106072: 56                push   %esi
c0106073: 53                push   %ebx
c0106074: 83 ec 0c          sub    $0xc,%esp
c0106077: 68 44 a2 12 c0    push   $0xc012a24
c010607c: e8 ed 1c 00 00    call   c0107d6e <printf>
c0106081: be 00 60 10 00    mov    $0x106000,
c0106086: c1 ee 0c          shr    $0xc,%esi
c0106089: 83 c4 08          add    $0x8,%esp
c010608c: 56                push   %esi
c010608d: 68 6c a2 12 c0    push   $0xc012a26c
c0106092: e8 d7 1c 00 00    call   c0107d6e <p
c0106097: 83 c4 10          add    $0x1
c010609a: bb 00 00 00 00    mov    $0x0,
c010609f: eb 14             jmp    c01060b5 <_kernel_post_init+0x46>
c01060a1: 89 d8             mov    %ebx,%eax
c01060a3: c1 e0 0c          shl    $0xc,%eax
c01060a6: 83 ec 0c          sub    $0xc,%esp
c01060a9: 50                push   %eax
c01060aa: e8 9f 0a 00 00    call   c0106b4e <vmm_unmap_page>
c01060af: 83 c              add    $0x1,%ebx
c01060b2: 83 c              add    $0x10,%esp
c01060b5: 39 f3             cmp
c01060b7: 72 e8             
c01060b9: e8 54             
c01060be: 85 c
c01060c0: 74 07             
c01060c2: 8d 65 f
c01060c5: 5b
c01060c6: 5e                pop    %esi
c01060c7: 5d                pop    %ebp
c01060c8: c3
c01060c9: 83 ec 04          sub    $0x4,%esp
c01060cc: 6a 40             push   $0x40
c01060ce: 68 af a0 12 c0    push   $0xc012a0af
```


LunaixOS

SERIAL ATA

从零开始

自制操作系统

磁盘与 SATA

探测磁盘

EP 13-4

我们最终的目的……

不管哪一种硬盘，读写扇区是我们最终的目的。

我们必须先要知道：

- 磁盘的基础信息：生产商信息，型号信息。
- 这个磁盘的容量多少
- 这个磁盘支持那种寻址模式： 32 位还是 48 位 LBA ？
- 这个磁盘类别： ATA 还是 ATAPI ？

常用 ATA/ATAPI 设备命令一览表

	LBA 位数 / SCSI CDB 大小	ATA 硬盘（ATA 命令）	操作码	ATAPI 硬盘（SCSI 命令）	操作码
读取设备信息	28 / 12	IDENTIFY DEVICE	ECh	IDENTIFY PACKET DEVICE	A1h
	48 / 16				
读取扇区	28 / 12	READ DMA	C8h	READ	A8h
	48 / 16	READ DMA EX	25h		88h
写入扇区	28 / 12	WRITE DMA	CAh	WRITE	AAh
	48 / 16	WRITE DMA EX	35h		8Ah
容量读取	28 / 12	IDENTIFY DEVICE	ECh	READ CAPACITY	9Eh
	48 / 16				

一个非常重要的准备工作：探测磁盘信息

使用 IDENTIFY DEVICE 或 IDENTIFY PACKET DEVICE 去探测磁盘信息

两个命令均返回一个长度为 512 字节的，**结构相同的**数据块儿，包含该设备的所有信息。

每个信息以字段形式存储，每个字段的长度字对齐（2 个字节）

我们关心以下几个重要的信息：

- 磁盘生产商名称
- 磁盘型号
- 最大可寻址逻辑扇区数 *
- 逻辑扇区大小 *
- 每个逻辑扇区包含的物理扇区数 *

ATAPI 设备需要通过 READ_CAPACITY 命令获取

- 磁盘唯一编号 (WWN)
- (ATAPI) 设备要求的 CDB 大小
- 是否支持 48 位 LBA

保存这些数据

```
struct hba_device
{
    char serial_num[20];
    char model[40];
    uint32_t flags;
    uint64_t max_lba;
    uint32_t block_size;
    uint64_t wwn;
    uint8_t cbd_size;
    uint8_t last_error;
    uint8_t last_status;
    uint32_t alignment_offset;
    uint32_t block_per_sec;
    uint32_t capabilities;
    struct hba_port* port;

};
```

```
You, 4 weeks ago | 1 author (You)
struct hba_port
{
    volatile hba_reg_t* regs;
    unsigned int ssts;
    struct hba_cmdh* cmdlst;
    void* fis;
    struct hba_device* device;
};
```

磁盘信息：字符串表示

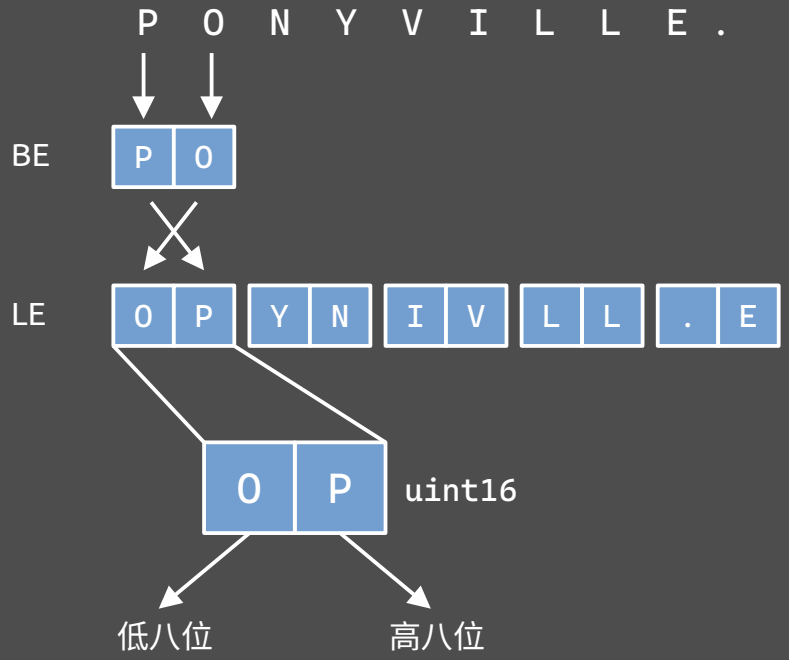
两个字符串字段：

生产商信息（20 字节定长）

产品型号（40 字节定长）

10..19	M	B	F	Serial number (see 7.12.7.10)
20..21			X	Retired
22			X	Obsolete
23..26	M	B	F	Firmware revision (see 7.12.7.13)
27..46	M	B	F	Model number (see 7.12.7.14)

位移：以字为单位




```
void
ahci_parsestr(char* str, uint16_t* reg_start, int size_word)
{
    int j = 0;
    for (int i = 0; i < size_word; i++, j += 2) {
        uint16_t reg = *(reg_start + i);
        str[j] = (char)(reg >> 8);
        str[j + 1] = (char)(reg & 0xff);
    }
    str[j - 1] = '\0';
}
```

注意：需要手动添加 0 终止符！

三个特殊的字段

Word	O M	S P	F V		Description
0	M	B			General configuration
			F	15:14	10b = ATAPI device
			F		11b = Reserved
				13	Reserved
			F	12:8	Indicates command set used by the device
			X	7	Obsolete
			F	6:5	00b = Device shall set DRQ to one within 3 ms of receiving PACKET command.
					01b = Obsolete.
					10b = Device shall set DRQ to one within 50 μs of receiving PACKET command.
					11b = Reserved
				4:3	Reserved
			V	2	Incomplete response
			F	1:0	00b = 12 byte command packet
					01b = 16 byte command packet
					1xb = Reserved

CDB 大小

106	O				Physical sector size / logical sector size (see 7.12.7.56)
		B	F	15	Shall be cleared to zero
		B	F	14	Shall be set to one
		B	F	13	Device has multiple logical sectors per physical sector.
		B	F	12	Device Logical Sector longer than 256 words
				11:4	Reserved
		B	F	3:0	2^X logical sectors per physical sector

每个物理扇区所蕴含的逻辑扇区数量

69	M				Additional Supported (see 7.12.7.30)
		N		15	Reserved for CFA
		B	F	14	Deterministic data in trimmed LBA range(s) is supported
		B	F	13	Long Physical Sector Alignment Error Reporting Control is supported
			X	12	Obsolete
		B	F	11	READ BUFFER DMA is supported
		B	F	10	WRITE BUFFER DMA is supported
			X	9	Obsolete
		B	F	8	DOWNLOAD MICROCODE DMA is supported
				7	Reserved for IEEE 1667
		B	F	6	0 = Optional ATA device 28-bit commands supported
		B	F	5	Trimmed LBA range(s) returning zeroed data is supported
		B	F	4	Device Encrypts All User Data on the device
		B	F	3	Extended Number of User Addressable Sectors is supported
		B	V	2	All write cache is non-volatile
				1:0	Reserved

是否支持 48 位 LBA

解析数据块

```
void
ahci_parse_dev_info(struct hba_device* dev_info, uint16_t* data)
{
    dev_info->max_lba = *((uint32_t*)(data + IDDEV_OFFMAXLBA));
    dev_info->block_size = *((uint32_t*)(data + IDDEV_OFFLSECSIZE));
    dev_info->cbd_size = (*data & 0x3) ? 16 : 12;
    dev_info->wwn = *(uint64_t*)(data + IDDEV_OFFWWN);
    dev_info->block_per_sec = 1 << (*(data + IDDEV_OFFLPP) & 0xf);
    dev_info->alignment_offset = *(data + IDDEV_OFFALIGN) & 0x3fff;
    dev_info->capabilities = *((uint32_t*)(data + IDDEV_OFFCAPABILITIES));

    if (!dev_info->block_size) {
        dev_info->block_size = 512;
    }

    if ((*(data + IDDEV_OFFADDSUPPORT) & 0x8)) {
        dev_info->max_lba = *((uint64_t*)(data + IDDEV_OFFMAXLBA_EXT));
        dev_info->flags |= HBA_DEV_FEXTLBA;
    }

    ahci_parsestr(&dev_info->serial_num, data + IDDEV_OFFSERIALNUM, 10);
    ahci_parsestr(&dev_info->model, data + IDDEV_OFFMODELNUM, 20);
}
```

```
#define IDDEV_OFFMAXLBA 60
#define IDDEV_OFFMAXLBA_EXT 230
#define IDDEV_OFFLSECSIZE 117
#define IDDEV_OFFWWN 108
#define IDDEV_OFFSERIALNUM 10
#define IDDEV_OFFMODELNUM 27
#define IDDEV_OFFADDSUPPORT 69
#define IDDEV_OFFALIGN 209
#define IDDEV_OFFLPP 106
#define IDDEV_OFFCAPABILITIES 49
```

更多字段以及他们的在数据块内的位移，可以参阅 ATA/ATAPI Command Set – 3 (ACS-3)，小节 7.12.7.1, 和 7.13.6.1

等等！该使用哪一个命令？

通过设备签名判断是否为 ATAPI 设备

Start	End	Symbol	Description
00h	03h	PxCLB	Port x Command List Base Address
04h	07h	PxCLBU	Port x Command List Base Address Upper 32-Bits
08h	0Bh	PxFB	Port x FIS Base Address
0Ch	0Fh	PxFBU	Port x FIS Base Address Upper 32-Bits
10h	13h	PxIS	Port x Interrupt Status
14h	17h	PxIE	Port x Interrupt Enable
18h	1Bh	PxCMD	Port x Command and Status
1Ch	1Fh	Reserved	Reserved
20h	23h	PxTFD	Port x Task File Data
24h	27h	PxSIG	Port x Signature
28h	2Bh	PxSSTS	Port x Serial ATA Status (SCR0: SStatus)
2Ch	2Fh	PxSCTL	Port x Serial ATA Control (SCR2: SControl)
30h	33h	PxSERR	Port x Serial ATA Error (SCR1: SError)
34h	37h	PxSACT	Port x Serial ATA Active (SCR3: SActive)
38h	3Bh	PxCI	Port x Command Issue
3Ch	3Fh	PxSNTF	Port x Serial ATA Notification (SCR4: SNotification)
40h	43h	PxFBS	Port x FIS-based Switching Control
44h	47h	PxDEVSLP	Port x Device Sleep
48h	6Fh	Reserved	Reserved
70h	7Fh	PxVS	Port x Vendor Specific

ATA 设备： `0x00000101`

ATAPI 设备： `0xeb140101`

Bits	ATA device [a]	ATAPI device [a]	Reserved for SATA [a]	Reserved for SATA [a]	Obsolete [a]
COUNT field (7:0)	01h	01h	01h	01h	N/A
LBA field (27:24)	Reserved	Reserved	Reserved	Reserved	Reserved
LBA field (23:16)	00h	EBh	C3h	96h	AAh
LBA field (15:8)	00h	14h	3Ch	69h	CEh
LBA field (7:0)	01h	01h	01h	01h	N/A

Bit	Type	Reset	Description
31:00	RO	FFFFFFFFh	**Signature (SIG):** Contains the signature received from a device on the first D2H Register FIS. The bit order is as follows:

Bit	Field
31:24	LBA High Register
23:16	LBA Mid Register
15:08	LBA Low Register
07:00	Sector Count Register

获取磁盘信息：发送命令

```c
/* 发送ATA命令，参考：SATA AHCI Spec Rev.1.3.1, section 5.5 */
struct hba_cmdt* cmd_table;
struct hba_cmdh* cmd_header;

// 确保端口是空闲的
wait_until(!(port->regs[HBA_RPxTFD] & (HBA_PxTFD_BSY)));

// 预备DMA接收缓存，用于存放HBA传回的数据
uint16_t* data_in = (uint16_t*)valloc_dma(512);

int slot = hba_prepare_cmd(port, &cmd_table, &cmd_header, data_in, 512);

// 清空任何待响应的中断
port->regs[HBA_RPxIS] = 0;
port->device = vzalloc(sizeof(struct hba_device));
port->device->port = port;

// 在命令表中构建命令FIS
struct sata_reg_fis* cmd_fis = (struct sata_reg_fis*)cmd_table->command_fis;

// 根据设备类型使用合适的命令
if (port->regs[HBA_RPxSIG] == HBA_DEV_SIG_ATA) {
    // ATA 一般为硬盘
    sata_create_fis(cmd_fis, ATA_IDENTIFY_DEVICE, 0, 0);
} else {
    // ATAPI 一般为光驱，软驱，或者磁带机
    port->device->flags |= HBA_DEV_FATAPI;
    sata_create_fis(cmd_fis, ATA_IDENTIFY_PAKCET_DEVICE, 0, 0);
}
```

```c
void
sata_create_fis(struct sata_reg_fis* cmd_fis,
                uint8_t command,
                uint64_t lba,
                uint16_t sector_count)
{
    cmd_fis->head.type = SATA_REG_FIS_H2D;
    cmd_fis->head.options = SATA_REG_FIS_COMMAND;
    cmd_fis->head.status_cmd = command;
    cmd_fis->dev = 0;

    cmd_fis->lba0 = SATA_LBA_COMPONENT(lba, 0);
    cmd_fis->lba8 = SATA_LBA_COMPONENT(lba, 8);
    cmd_fis->lba16 = SATA_LBA_COMPONENT(lba, 16);
    cmd_fis->lba24 = SATA_LBA_COMPONENT(lba, 24);

    cmd_fis->lba32 = SATA_LBA_COMPONENT(lba, 32);
    cmd_fis->lba40 = SATA_LBA_COMPONENT(lba, 40);

    cmd_fis->count = sector_count;
}
```

```c
// PxCI寄存器置位，告诉HBA这儿有个数据需要发送到SATA端口
port->regs[HBA_RPxCI] = (1 << slot);

wait_until(!(port->regs[HBA_RPxCI] & (1 << slot)));

if ((port->regs[HBA_RPxTFD] & HBA_PxTFD_ERR)) {
    // 有错误
    sata_read_error(port);
    goto fail;
}

/*…
ahci_parse_dev_info(port->device, data_in);

if (!(port->device->flags & HBA_DEV_FATAPI)) {
    goto done;
}
```

获取磁盘信息：发送命令

事实上，在 AHCI 标准中（5.5.1），已经非常详细的阐明了向设备发送命令全部流程

1. 在命令队列中找到一个空闲的位置
2. 原地构建一个命令头，根据需求设置好各类字段
3. 分配一个**连续的物理空间**作为命令表。
4. 根据需求设定好命令表（设置 FIS，设置 PRDT）
5. 将命令表的**物理地址**挂载到命令头上
6. 将 PxCI 的对应位置位。
7. 等待，直到 PxCI 的对应位清零。这表明命令执行完成。
8. 读取 PxTFD 寄存器，获取命令执行结果，以及错误代码（如果发生错误）

```
int
hba_prepare_cmd(struct hba_port* port,
                struct hba_cmdt** cmdt,
                struct hba_cmdh** cmdh,
                void* buffer,
                unsigned int size)
{
    int slot = __get_free_slot(port);
    assert_msg(slot ≥ 0, "HBA: No free slot");
    assert_msg(size ≤ 0x400000, "HBA: buffer too big");

    // 构建命令头（Command Header）和命令表（Command Table）
    struct hba_cmdh* cmd_header = &port->cmdlst[slot];
    struct hba_cmdt* cmd_table = vzalloc_dma(sizeof(struct hba_cmdt));

    memset(cmd_header, 0, sizeof(*cmd_header));

    // 将命令表挂到命令头上
    cmd_header->cmd_table_base = vmm_v2p(cmd_table);
    cmd_header->options =
      HBA_CMDH_FIS_LEN(sizeof(struct sata_reg_fis)) | HBA_CMDH_CLR_BUSY;

    if (buffer) {
        cmd_header->prdt_len = 1;
        cmd_table->entries[0] =
          (struct hba_prdte){ .data_base = vmm_v2p(buffer),
                              .byte_count = size - 1 };
    }

    *cmdh = cmd_header;
    *cmdt = cmd_table;

    return slot;
}
```

最后一味配方：完成对 ATAPI 设备的信息获取

Table 110. READ CAPACITY (16) command

Bit / Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (9Eh)							
1	Reserved			SERVICE ACTION (10h)				
2	(MSB)							
9	LOGICAL BLOCK ADDRESS 0							(LSB)
10	(MSB)							
13	ALLOCATION LENGTH							(LSB)
14	Reserved							PMI
15	CONTROL							

注意：

不同的命令对 CDB 的格式有着不同的定义。

已分配的接收缓冲区大小

```
wait_until(!(port->regs[HBA_RPxTFD] & (HBA_PxTFD_BSY)));

// 预备DMA接收缓存，用于存放HBA传回的数据
uint16_t* data_in = (uint16_t*)valloc_dma(512);

int slot = hba_prepare_cmd(port, &cmd_table, &cmd_header, data_in, 512);
```

最后一味配方：完成对 ATAPI 设备的信息获取

READ CAPACITY 会返回一个数据块儿，或者叫做**参数数据 (Parameter Data)**

Table 111. READ CAPACITY (16) parameter data

Bit / Byte	7	6	5	4	3	2	1	0
0	(MSB)			RETURNED LOGICAL BLOCK ADDRESS				
7								(LSB)
8	(MSB)			LOGICAL BLOCK LENGTH IN BYTES				
11								(LSB)
12	Reserved				P_TYPE			PROT_EN
13	P_I_EXPONENT				LOGICAL BLOCKS PER PHYSICAL BLOCK EXPONENT			
14	TPE	TPRZ	(MSB)	LOWEST ALIGNED LOGICAL BLOCK ADDRESS				
15								(LSB)
16	Reserved							
31								

注意：
所有整数均以大端序形式存储

发送 ATAPI 命令： READ_CAPACITY

```c
void
scsi_create_packet16(struct scsi_cdb16* cdb,
                     uint8_t opcode,
                     uint64_t lba,
                     uint32_t alloc_size)
{
    memset(cdb, 0, sizeof(*cdb));
    cdb->opcode = opcode;
    cdb->lba_be_hi = SCSI_FLIP((uint32_t)(lba >> 32));
    cdb->lba_be_lo = SCSI_FLIP((uint32_t)lba);
    cdb->length = SCSI_FLIP(alloc_size);
}
```

SCSI_FLIP： 翻转字节顺序（端序转换）

```c
#define SCSI_FLIP(val)
    ((((val)&0x000000ff) << 24) | (((val)&0x0000ff00) << 8) |
     (((val)&0x00ff0000) >> 8) | (((val)&0xff000000) >> 24))
```

```c
void
scsi_parse_capacity(struct hba_device* device, uint32_t* parameter)
{
    device->max_lba = SCSI_FLIP(*(parameter + 1));
    device->block_size = SCSI_FLIP(*(parameter + 2));
}
```

```c
struct scsi_cdb16* cdb16 = (struct scsi_cdb16*)cmd_table->atapi_cmd;

sata_create_fis(cmd_fis, ATA_PACKET, 512 << 8, 0);
scsi_create_packet16(cdb16, SCSI_READ_CAPACITY_16, 0, 512);

cdb16->misc1 = 0x10; // service action
cmd_header->transferred_size = 0;
cmd_header->options |= HBA_CMDH_ATAPI;

port->regs[HBA_RPxCI] = (1 << slot);
wait_until(!(port->regs[HBA_RPxCI] & (1 << slot)));

if ((port->regs[HBA_RPxTFD] & HBA_PxTFD_ERR)) {
    // 有错误
    sata_read_error(port);
    goto fail;
}

scsi_parse_capacity(port->device, (uint32_t*)data_in);
```

终于……我们终于可以开始读写扇区了！

我们已经知道如何探测一个硬盘，并且使用简单的命令获取到一些基础信息。

并且也注意到 ATA 与 ATAPI 设备之间的操作是需要区分的。

在下一期视频里，我们将会介绍剩下的几个读写扇区的指令。

	LBA 位数 / SCSI CDB 大小	ATA 硬盘（ATA 命令）	操作码	ATAPI 硬盘（SCSI 命令）	操作码
读取设备信息	28 / 12	IDENTIFY DEVICE	ECh	IDENTIFY PACKET DEVICE	A1h
	48 / 16				
读取扇区	28 / 12	READ DMA	C8h	READ	A8h
	48 / 16	READ DMA EX	25h		88h
写入扇区	28 / 12	WRITE DMA	CAh	WRITE	AAh
	48 / 16	WRITE DMA EX	35h		8Ah
容量读取	28 / 12	IDENTIFY DEVICE	ECh	READ CAPACITY	9Eh
	48 / 16				

```
c010606f: 55                push   %ebp
c0106070: 89 e5             mov    %esp,%ebp
c0106072: 56                push   %esi
c0106073: 53                push   %ebx
c0106074: 83 ec 0c          sub    $0xc,%esp
c0106077: 68 44 a2 12 c0    push   $0xc012a24
c010607c: e8 ed 1c 00 00    call   c0107d6e <
c0106081: be 00 60 10 00    mov    $0x106000,
c0106086: c1 ee 0c          shr    $0xc,%esi
c0106089: 83 c4 08          add    $0x8,%esp
c010608c: 56                push   %esi
c010608d: 68 6c a2 12 c0    push   $0xc012a26c
c0106092: e8 d7 1c 00 00    call   c0107d6e <
c0106097: 83 c4 10          add    $0x1
c010609a: bb 00 00 00 00    mov    $0x
c010609f: eb 14             jmp    c01060b5 <_kernel_post_init+0x46>
c01060a1: 89 d8             mov    %ebx,%eax
c01060a3: c1 e0 0c          shl    $0xc,%eax
c01060a6: 83 ec 0c          sub    $0xc,%esp
c01060a9: 50                push   %eax
c01060aa: e8 9f 0a 00 00    call   c0106b4e <vmm_unmap_page>
c01060af: 83 c
c01060b2: 83 c
c01060b5: 39 f3
c01060b7: 72 e8
c01060b9: e8 54
c01060be: 85 c
c01060c0: 74 07
c01060c2: 8d 65 f
c01060c5: 5b
c01060c6: 5e
c01060c7: 5d
c01060c8: c3
c01060c9: 83 ec 04          $0x4,%esp
c01060cc: 6a 40             $0x40
c01060ce: 68 af a0 12 c0    push   $0xc012a0af
```

LunaixOS

SERIAL ATA

从零开始

自制操作系统

磁盘与 SATA

读写扇区

EP 13-5

磁盘的读写

在此提醒：磁盘读写的最小操作单位是**扇区（逻辑块，逻辑扇区）**

	LBA 位数 / SCSI CDB 大小	ATA 硬盘（ATA 命令）	操作码	ATAPI 硬盘（SCSI 命令）	操作码
读取设备信息	28 / 12	IDENTIFY DEVICE	ECh	IDENTIFY PACKET DEVICE	A1h
	48 / 16				
读取扇区	28 / 12	READ DMA	C8h	READ	A8h
	48 / 16	READ DMA EX	25h		88h
写入扇区	28 / 12	WRITE DMA	CAh	WRITE	AAh
	48 / 16	WRITE DMA EX	35h		8Ah
容量读取	28 / 12	IDENTIFY DEVICE	ECh	READ CAPACITY	9Eh
	48 / 16				

区分 ATA 和 SCSI 的操作：面向对象的思维

不同类型的设备有不同的操作范式。

我们希望将他们抽象出来……

面向对象的接口 (interface) 概念。

```
struct hba_device
{
    char serial_num[20];
    char model[40];
    uint32_t flags;
    uint64_t max_lba;
    uint32_t block_size;
    uint64_t wwn;
    uint8_t cbd_size;
    uint8_t last_error;
    uint8_t last_status;
    uint32_t alignment_offset;
    uint32_t block_per_sec;
    uint32_t capabilities;
    struct hba_port* port;

    You, 2 weeks ago | 1 author (You)
    struct
    {
        int (*identify)(struct hba_device* dev);
        int (*read_buffer)(struct hba_device* dev,
                           uint64_t lba,
                           void* buffer,
                           uint32_t size);
        int (*write_buffer)(struct hba_device* dev,
                            uint64_t lba,
                            void* buffer,
                            uint32_t size);
    } ops;
};
```

identify： 识别这个设备（用于热插拔的情况）

read_buffer： 读取扇区到缓冲区

write_buffer： 从缓冲区写入扇区

读写 ATA 磁盘的扇区

如果磁盘支持 28 位 LBA： READ/WRITE DMA

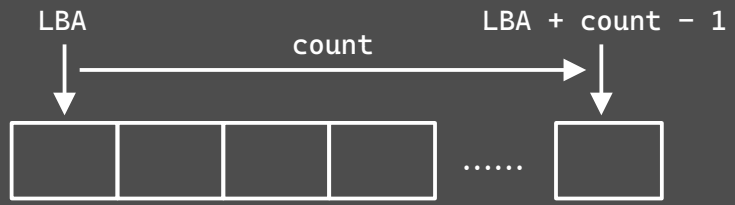
如果磁盘支持 48 位 LBA： READ/WRITE DMA (EXT) ⟶ 可向下兼容 READ/WRITE DMA

每个命令需要三个参数：

1. 起始 LBA 地址 → LBA
2. 需要读入的逻辑块数量 → count
3. 数据的存取位置 → buffer

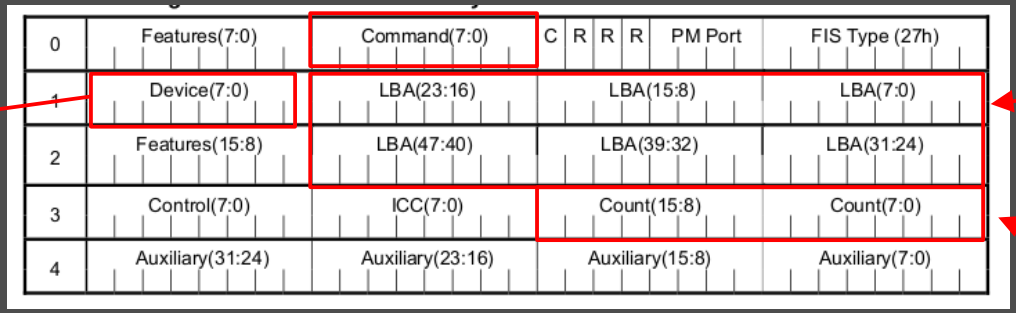
↓

buffer 需要在物理地址上连续，挂在 PRDT 里面。



读写 ATA 磁盘的扇区

```
C8h: READ DMA
25h: READ DMA EXT
CAh: WRITE DMA
35h: WRITE DMA EXT
```



注意：
Device 第 6 位需置位，表明我们使用的是 LBA 寻址模式，而非 CHS

起始 LBA

读入的逻辑块儿数

寄存器 FIS： 主机到设备 - 向寄存器写入值

读写 ATA 磁盘的扇区

和之前一样，准备发送命令头和命令结构（命令表），
并且设置 PRDT

```
struct hba_port* port = dev->port;
struct hba_cmdh* header;
struct hba_cmdt* table;
int slot = hba_prepare_cmd(port, &table, &header, buffer, size);
int bitmask = 1 << slot;
```

```
int
hba_prepare_cmd(struct hba_port* port,
                struct hba_cmdt** cmdt,
                struct hba_cmdh** cmdh,
                void* buffer,
                unsigned int size)
{
    int slot = __get_free_slot(port);
    assert_msg(slot ≥ 0, "HBA: No free slot");
    assert_msg(size ≤ 0x400000, "HBA: buffer too big");

    // 构建命令头（Command Header）和命令表（Command Table）
    struct hba_cmdh* cmd_header = &port->cmdlst[slot];
    struct hba_cmdt* cmd_table = vzalloc_dma(sizeof(struct hba_cmdt));

    memset(cmd_header, 0, sizeof(*cmd_header));

    // 将命令表挂到命令头上
    cmd_header->cmd_table_base = vmm_v2p(cmd_table);
    cmd_header->options =
      HBA_CMDH_FIS_LEN(sizeof(struct sata_reg_fis)) | HBA_CMDH_CLR_BUSY;

    if (buffer) {
        cmd_header->prdt_len = 1;
        cmd_table->entries[0] =
          (struct hba_prdte){ .data_base = vmm_v2p(buffer),
                              .byte_count = size - 1 };
    }

    *cmdh = cmd_header;
    *cmdt = cmd_table;

    return slot;
}
```

设置 PRDT

读写 ATA 磁盘的扇区

```c
if ((port->device->flags & HBA_DEV_FEXTLBA)) {
    // 如果该设备支持48位LBA寻址
    sata_create_fis(
      fis, write ? ATA_WRITE_DMA_EXT : ATA_READ_DMA_EXT, lba, count);
} else {
    sata_create_fis(fis, write ? ATA_WRITE_DMA : ATA_READ_DMA, lba, count);
}
/*
fis->dev = (1 << 6);
```

```c
void
sata_create_fis(struct sata_reg_fis* cmd_fis,
                uint8_t command,
                uint64_t lba,
                uint16_t sector_count)
{
    cmd_fis->head.type = SATA_REG_FIS_H2D;
    cmd_fis->head.options = SATA_REG_FIS_COMMAND;
    cmd_fis->head.status_cmd = command;
    cmd_fis->dev = 0;

    cmd_fis->lba0 = SATA_LBA_COMPONENT(lba, 0);
    cmd_fis->lba8 = SATA_LBA_COMPONENT(lba, 8);
    cmd_fis->lba16 = SATA_LBA_COMPONENT(lba, 16);
    cmd_fis->lba24 = SATA_LBA_COMPONENT(lba, 24);

    cmd_fis->lba32 = SATA_LBA_COMPONENT(lba, 32);
    cmd_fis->lba40 = SATA_LBA_COMPONENT(lba, 40);

    cmd_fis->count = sector_count;
}
```

读写 ATA 磁盘的扇区

封装成我们前面定义的接口形式

```
int
sata_read_buffer(struct hba_device* dev,
                 uint64_t lba,
                 void* buffer,
                 uint32_t size)
{
    return __sata_buffer_io(dev, lba, buffer, size, 0);
}

int
sata_write_buffer(struct hba_device* dev,
                  uint64_t lba,
                  void* buffer,
                  uint32_t size)
{
    return __sata_buffer_io(dev, lba, buffer, size, 1);
}
```

读写 ATAPI 设备的扇区

需要使用 SCSI 协议的命令，使用 PACKET 命令做封装

如果磁盘支持 28 位 LBA： 12 字节长度的 CDB

如果磁盘支持 48 位 LBA： 16 字节长度的 CDB

} READ/WRITE

和 ATA 设备的一样，也是需要三个参数：

1. 起始 LBA
2. 操作的逻辑块数
3. 存取缓冲区位置

ATAPI 硬盘（SCSI 命令）	操作码	
IDENTIFY PACKET DEVICE	A1h	
READ	A8h	LBA32
	88h	LBA48
WRITE	AAh	
	8Ah	
READ CAPACITY	9Eh	

读写 ATAPI 设备的扇区

READ/WRITE 命令有着一样的 CDB 格式

Table 65 — WRITE (12) command

Byte\Bit	7	6	5	4	3	2	1	0
0	OPERATION CODE (AAh)							
1	WRPROTECT			DPO	FUA	Reserved	FUA_NV	Obsolete
2	(MSB)	LOGICAL BLOCK ADDRESS						
5								(LSB)
6	(MSB)	TRANSFER LENGTH						
9								(LSB)
10	Restricted for MMC-4	Reserved		GROUP NUMBER				
11	CONTROL							

Table 31 — READ (12) command

Byte\Bit	7	6	5	4	3	2	1	0
0	OPERATION CODE (A8h)							
1	RDPROTECT			DPO	FUA	Reserved	FUA_NV	Obsolete
2	(MSB)	LOGICAL BLOCK ADDRESS						
5								(LSB)
6	(MSB)	TRANSFER LENGTH						
9								(LSB)
10	Restricted for MMC-4	Reserved		GROUP NUMBER				
11	CONTROL							

读写 ATAPI 设备的扇区

以 WRITE(12) 举例子

ATAPI 硬盘（SCSI 命令）	操作码
IDENTIFY PACKET DEVICE	A1h
READ	A8h
	88h
WRITE	AAh
	8Ah
READ CAPACITY	9Eh

Table 65 — WRITE (12) command

Byte\Bit	7	6	5	4	3	2	1	0
0	OPERATION CODE (AAh)							
1	WRPROTECT			DPO	FUA	Reserved	FUA_NV	Obsolete
2	(MSB)			LOGICAL BLOCK ADDRESS				
5								(LSB)
6	(MSB)			TRANSFER LENGTH				
9								(LSB)
10	Restricted for MMC-4		Reserved		GROUP NUMBER			
11	CONTROL							

读写 ATAPI 设备的扇区

以 WRITE(12) 举例子

Table 65 — WRITE (12) command

Byte\Bit	7	6	5	4	3	2	1	0
0	OPERATION CODE (AAh)							
1	WRPROTECT			DPO	FUA	Reserved	FUA_NV	Obsolete
2	(MSB)	LOGICAL BLOCK ADDRESS						
5								(LSB)
6	(MSB)	TRANSFER LENGTH						
9								(LSB)
10	Restricted for MMC-4	Reserved		GROUP NUMBER				
11	CONTROL							

读写 ATAPI 设备的扇区

FUA/ FUA_NV 允许用户指定在读写时缓存的参与情况

FUA： 介质单元强制访问（ Force Unit Access ）

FUA_NV： 非易失性缓存单元强制访问（ Force Unit Access on Non-Volatile cache ）

非易失性缓存（断电后数据依然驻留）

FUA	FUA_NV		命令返回行为
0	0	优先使用缓存（不论类型），介质的使用是否由磁盘驱动器决定。	命令的返回不代表数据已被写入存储介质。任何在写入存储介质时产生的错误，不会被及时递送。
0	1	总是使用非易失性缓存，如果设备不支持此类缓存，则使用普通的易失性缓存。	
1	X	总是使用介质。	命令的返回代表数据已被写入存储介质

在我们的实现中，我们使用默认情况：
FUA=0 ， FUA_NV=0

读写 ATAPI 设备的扇区

以 WRITE(12) 举例子

数据保护字段（对于读取操作则是 RDPROTECT ）。

我们目前设置为： 0x3 ，即禁用任何数据保护检查。

Table 65 — WRITE (12) command

Byte\Bit	7	6	5	4	3	2	1	0
0	OPERATION CODE (AAh)							
1	WRPROTECT			DPO	FUA	Reserved	FUA_NV	Obsolete
2	(MSB)	LOGICAL BLOCK ADDRESS						
5								(LSB)
6	(MSB)	TRANSFER LENGTH						
9								(LSB)
10	Restricted for MMC-4	Reserved		GROUP NUMBER				
11	CONTROL							

读写 ATAPI 设备的扇区

```
struct sata_reg_fis* fis = table->command_fis;
void* cdb = table->atapi_cmd;
sata_create_fis(fis, ATA_PACKET, (size << 8), 0);
fis->feature = 1 | ((!write) << 2);

if (port->device->cbd_size == 16) {
    scsi_create_packet16((struct scsi_cdb16*)cdb,
                         write ? SCSI_WRITE_BLOCKS_16 : SCSI_READ_BLOCKS_16,
                         lba,
                         count);
} else {
    scsi_create_packet12((struct scsi_cdb12*)cdb,
                         write ? SCSI_WRITE_BLOCKS_12 : SCSI_READ_BLOCKS_12,
                         lba,
                         count);
}

// field: cdb->misc1
*((uint8_t*)cdb + 1) = 3 << 5; // RPROTECT=011b 禁用保护检查
```

PACKET 命令

FEATURE

Bit	Description
7:3	Reserved
2	DMADIR bit – See 7.18.4
1	Obsolete
0	DMA bit – See 7.18.4

{ 0： DMA 写入
1： DMA 读取

是否使用 DMA

Code	Logical unit formatted with protection information	Field in protection information	Device server check	If check fails [d i], additional sense code
011b [b j]	Yes [e]	LOGICAL BLOCK GUARD	Shall not	No check performed
		LOGICAL BLOCK APPLICATION TAG	Shall not	No check performed
		LOGICAL BLOCK REFERENCE TAG	Shall not	No check performed
	No [a]	No protection information available to check		

读写 ATAPI 设备的扇区

同样，封装成我们定义的统一接口

```
void
scsi_read_buffer(struct hba_device* dev,
                 uint64_t lba,
                 void* buffer,
                 uint32_t size)
{
    __scsi_buffer_io(dev, lba, buffer, size, 0);
}

void
scsi_write_buffer(struct hba_device* dev,
                  uint64_t lba,
                  void* buffer,
                  uint32_t size)
{
    __scsi_buffer_io(dev, lba, buffer, size, 1);
}
```

接入具体实现

```
void
achi_register_ops(struct hba_port* port)
{
    port->device->ops.identify = ahci_identify_device;
    if (!(port->device->flags & HBA_DEV_FATAPI)) {
        port->device->ops.read_buffer = sata_read_buffer;
        port->device->ops.write_buffer = sata_write_buffer;
    } else {
        port->device->ops.read_buffer = scsi_read_buffer;
        port->device->ops.write_buffer = scsi_write_buffer;
    }
}
```

测试扇区读写

测试数据

```
char test_sequence[] = "Once upon a time, in a magical land of Equestria. "
                       "There were two regal sisters who ruled together "
                       "and created harmony for all the land.";

void
__test_disk_io()
{
    struct hba_port* port = ahci_get_port(0);
    char* buffer = vcalloc_dma(port->device->block_size);
    strcpy(buffer, test_sequence);
    kprintf("WRITE: %s\n", buffer);
    int result;

    // 写入第一扇区 (LBA=0)
    result =
      port->device->ops.write_buffer(port, 0, buffer, port->device->block_size);
    if (!result) {
        kprintf(KWARN "fail to write: %x\n", port->device->last_error);
    }

    memset(buffer, 0, port->device->block_size);

    // 读出我们刚刚写的内容!
    result =
      port->device->ops.read_buffer(port, 0, buffer, port->device->block_size);
    kprintf(KDEBUG "%x, %x\n", port->regs[HBA_RPxIS], port->regs[HBA_RPxTFD]);
    if (!result) {
        kprintf(KWARN "fail to read: %x\n", port->device->last_error);
    } else {
        kprint_hex(buffer, 256);
    }

    vfree_dma(buffer);
}
```

缺点儿什么吗?

SATA 驱动和磁盘 I/O 不是一个五期视频能说得完的东西……

有些东西我们没涵盖:

1、 I/O 任务队列和调度

2、 AHCI 的故障恢复机制

3、更鲁棒的命令错误的检测与处理机制

4、热插拔 SATA 设备的处理

5、 Port Multiplier 的支持

6、许多人经常问我的： SATA 协议的更深层细节

```
while (retries < MAX_RETRY) {
    port->regs[HBA_RPxCI] = bitmask;

    wait_until(!(port->regs[HBA_RPxCI] & bitmask));

    if ((port->regs[HBA_RPxTFD] & HBA_PxTFD_ERR)) {
        // 有错误
        sata_read_error(port);
        retries++;
    } else {
        vfree_dma(table);
        return 1;
    }
}
```

当 I/O 命令出错时， CI 寄存器的对应位不会清零！造成死循环！

别担心，我们最终都会涵盖的！ :)

我们的下一步

整个操作系统开发中，最重要，最复杂，也是最难啃的一个骨头：

文件系统

The File System