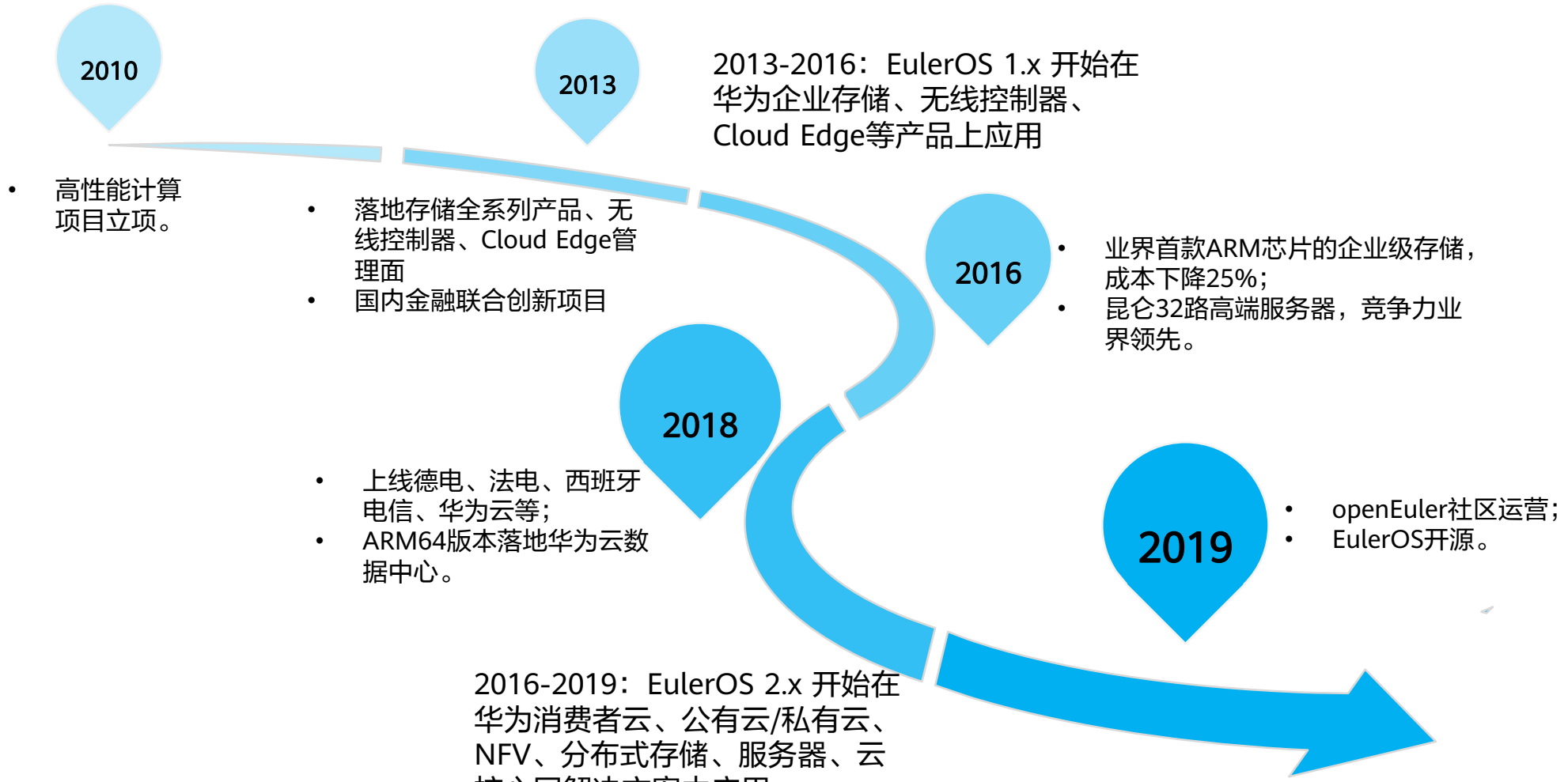
# openEuler操作系统介绍

主要讲述openEuler操作系统出现的背景、意义和适用场景、通用架构，并结合鲲鹏处理器硬件基础，介绍了openEuler相对于通用Linux操作系统的增强。

# openEuler出现的背景和意义

- 华为公司的网络设备、终端设备、存储系统等很多产品都要用到操作系统。
- 运行在通用服务器上的一款操作系统称为EulerOS。
- 随着云计算的兴起，其地位越来越重要。
- 随着鲲鹏芯片的研发，EulerOS成为与鲲鹏芯片配套的软件基础设施。
- 为推动鲲鹏生态的持续快速发展，繁荣国内和全球的计算产业，2019年底EulerOS被正式推到开源社区，成为openEuler。
- 所有个人开发者、企业、商业组织都可以使用openEuler社区版本，也可以基于openEuler社区版本发布自己二次开发的操作系统版本。

# openEuler发展历程一览



# openEuler适用场景

- openEuler是一款通用服务器操作系统；
- 支持x86和ARM等多种处理器架构；
- 适用于数据库、大数据、云计算、人工智能等各种应用场景。

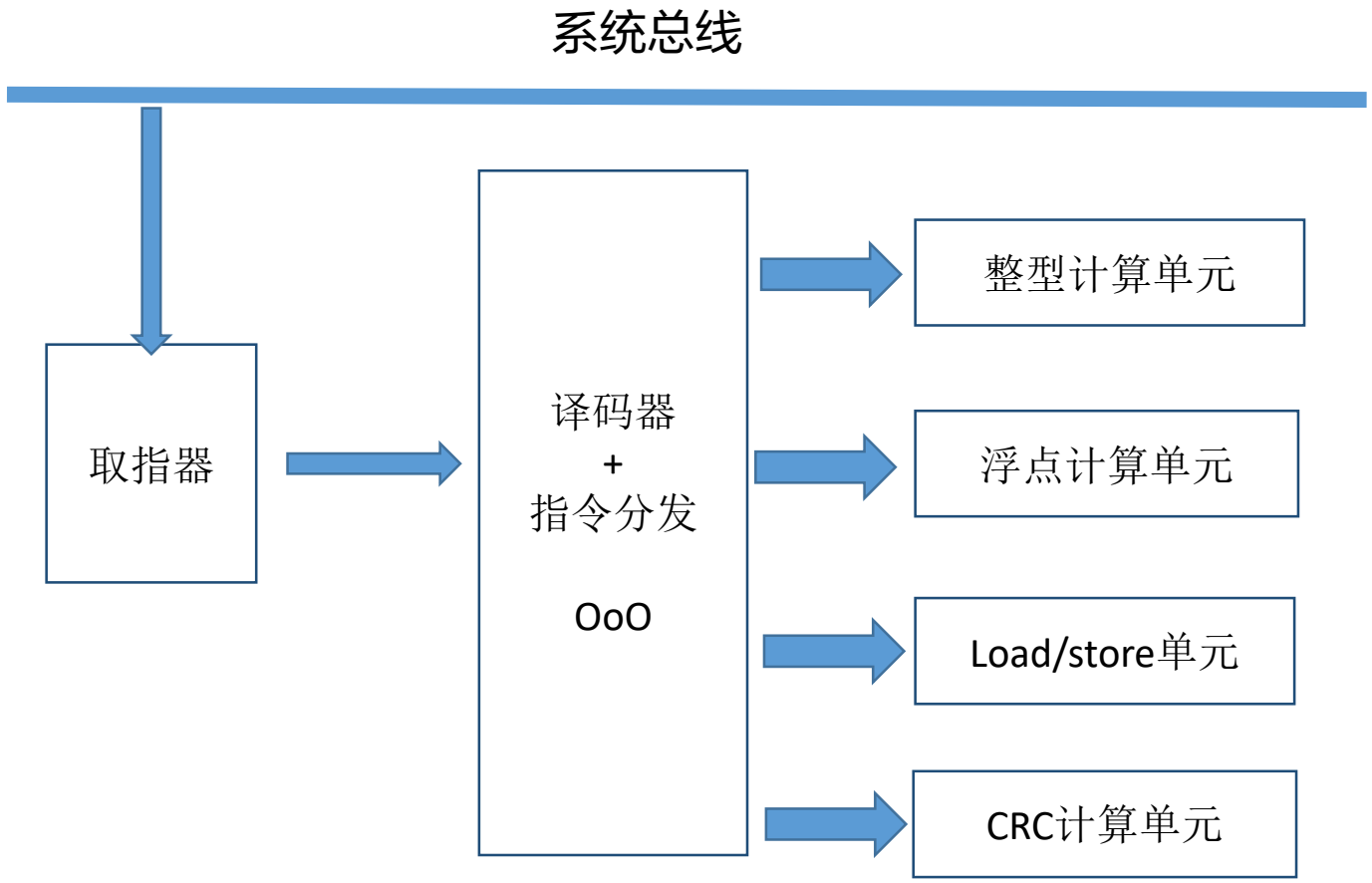
# 鲲鹏处理器简介

- 鲲鹏处理器基于ARMv8-64指令集开发的通用处理器，作为一款现代处理器，它早已经不是仅仅包含ALU的运算单元了。在其物理架构上包含SoC、Chip、DIE、cluster、core等概念。

# 鲲鹏处理器的组织

- SoC – System on chip，例如Kunpeng 920除了CPU外，还集成了RoCE 网卡、SAS控制器和南桥。
- Chip – 几块硅片可以封装在一起组成一个芯片 - DIE。
- DIE -芯片的最小物理单元。Kunpeng 920封装了3个DIE，两个用来做计算，第三个用来做IO。
- Cluster – 若干个核(core)的集合。Kunpeng 920把4个core集合成为一个cluster，而一个DIE上有8个cluster。
- Core -真正的计算单元，我们在操作系统侧看到的“核”。

# 鲲鹏920的指令执行情况




- 鲲鹏处理器的指令执行也分为取指、译码、执行等几个步骤
- 鲲鹏920还支持超标量、指令乱序执行(out-of-order)等特性
- 鲲鹏处理器中还有一部分专门的加速执行单元

# 鲲鹏处理器的内部存储结构 (2)

- 鲲鹏处理器内部设计了多层cache来缓存数据
- 鲲鹏920具有L1、L2、L3共三级cache
- L1的指令cache(L1D)和数据cache(L1I)大小都是64KB
- L2 cache不区分指令或数据，大小为512KB
- L1和L2两级cache由各个CPU core独享
- L3 cache也不区分指令和数据，但分为tag和data两部分
- 每个CPU DIE有4组DDR channel，总共支持最大2TB DDR内存空间

# 鲲鹏处理器的IO子系统

- 鲲鹏处理器和IO子系统通过IO DIE进行扩展，支持SoC片上加速器，如100G网卡、SAS控制器等。
- 鲲鹏处理器同时支持基于PCIe 4.0的设备扩展，可支持网卡、GPU等板卡。
- 为了方便软件编程，内部SoC上的高速设备也基于PCIe，且可以通过PCIe的配置空间进行配置。

# 鲲鹏处理器的虚拟化技术

- 鲲鹏处理器支持CPU Core虚拟化、内存虚拟化、中断虚拟化以及SMMU等多项虚拟化技术。
- 多个虚拟机(即VM，Virtual Machine)可以运行在一个中间层(Hypervisor)之上，共用一套硬件资源。
- 每个VM按照原有的方式运行并只看到属于自己的资源，互相不能访问对方的资源。

# 鲲鹏处理器与openEuler

- 鲲鹏处理器与openEuler操作系统有着非常紧密的联系，openEuler天然地支持鲲鹏处理器，并能够充分发挥处理器的各种特性。
- openEuler也支持x86体系架构，并有计划支持RISC-V架构。
- openEuler以及容器等基础软件平台已经贡献到开源社区。

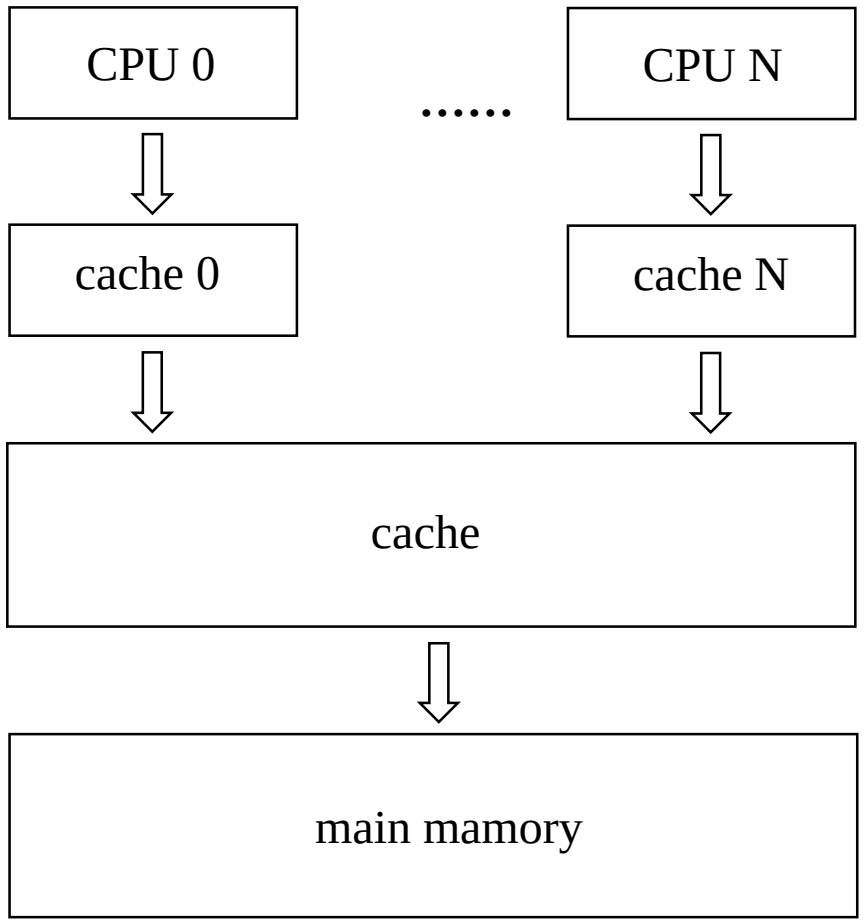
# 指令的乱序执行

- 为了提高性能，现代CPU利用流水线技术在一个时钟周期内会执行多条指令，并且不一定按照软件中规定的顺序执行，称为乱序执行。
- 这是一种指令级并行ILP(Instruction-Level Parallelism)技术。
- 程序示例：

```
1  a = 1
2  b = 2
3  c = a + b
```

在这里，第一行和第二行的乱序执行不会影响程序结果。

# 对称多处理器SMP




- 现代CPU有多个核心，每个核心有自己的cache，所有核心共享同一内存空间。我们称这种结构为SMP(Symmetric multiprocessing)

# SMP结构下的指令乱序执行

考虑如下代码：

| CPU0 | CPU1 |
| --- | --- |
| a = 1 | while (flag) { |
| b = 2 | do_something(a, b, c); |
| c = 3 | } |
| flag = true | |

乱序执行

| CPU0 | CPU1 |
| --- | --- |
| a = 1 | while (flag) { |
| b = 2 | do_something(a, b, c); |
| flag = true | } |
| c = 3 | |

本来想等a, b, c初始化完毕后CPU1 do something

现在c还未初始化CPU1就可以 do something

# 鲲鹏处理器下提供的屏障指令

- DMB(Data Memory Barrier)：保证该指令之前的内存访问结束，指令之后的内存访问必须在此之后执行，其他非内存访问指令依然可以乱序执行。
- DSB(Data Synchronization Barrier)：比DMB严格，保证DSB之前的所有指令执行完成，但是性能牺牲会比DMB更大。
- ISB(Instruction Synchronization Barrier)：保证之前的命令执行完成，ISB之后的命令需要重新从cache或memory取指。常用于发生异常，异常返回，更改系统配置寄存器的场景下。

# openEuler对内存屏障指令的封装

openEuler内核对这些指令主要进行了如下封装：

| /arch/arm64/include/asm/barrier.h (Linux内核5.7) |
| --- |
| #define mb()  dsb(sy)  //任意 |
| #define rmb()  dsb(ld)  //读-读  读-写 |
| #define wmb()  dsb(st)  //写-写 |
| #define dsb(opt) asm volatile("dsb" #opt : : : "memory") |
| |
| #define dma_rmb()  dmb(oshld)  //读-读  读-写 |
| #define dma_wmb()  dmb(oshst)  //任意 |
| #define dmb(opt) asm volatile("dmb" #opt : : : "memory") |

# 内存顺序模型

- 不同的CPU架构有不同的内存顺序模型：
  - 绝对顺序模型：禁止所有优化导致的乱序执行，所有内存操作串行排队。
  - 强内存顺序模型：以x86为代表，只允许store-load(即先执行store指令，再执行load指令)乱序执行。
  - 弱内存顺序模型：以ARM为代表，允许所有情况下的指令乱序执行。

# 内存顺序模型的一个例子

对于下面的代码：

| 初始状态：a = 0, b = 0 | |
|---|---|
| CPU0 | CPU1 |
| a = 1 | b = 1 |
| print b | print a |

四种打印结果：

| CPU0 | CPU1 | 说明 | 绝对顺序模型 | 强、弱内存模型 |
|---|---|---|---|---|
| 0 | 0 | 两个CPU都先执行打印，后执行赋值（乱序执行） | 不可能 | 可能 |
| 0 | 1 | CPU0先执行完毕，然后CPU1执行 | 可能 | 可能 |
| 1 | 0 | CPU1先执行完毕，然后CPU0执行 | 可能 | 可能 |
| 1 | 1 | 两个CPU同时执行第一条命令，然后同时执行第二条命令 | 可能 | 可能 |

- 在绝对顺序模型下，不会出现打印0，0的情况，因为此模型下指令执行顺序不会被打乱。
- 强、弱内存模型都不会保证store-load顺序，所以上面四种情况都可能会出现。

# 程序移植问题

- 由于x86和ARM架构的内存顺序模型不同(x86更为严格)，所以不同架构间移植代码的时候需要特别注意访存的问题：
  - x86硬件会保证load-load、store-store、load-store类指令的执行顺序，因此不需要软件在这些指令后添加内存屏障指令。
  - 但是，这样的代码在ARM这种弱内存模型下运行就会出问题，在没有加入内存屏障指令时，这3种指令会乱序执行。
  - 所以从x86向ARM移植代码的时候需要注意添加内存屏障解决问题。

# openEuler内存屏障实例 – kfifo队列 (1)

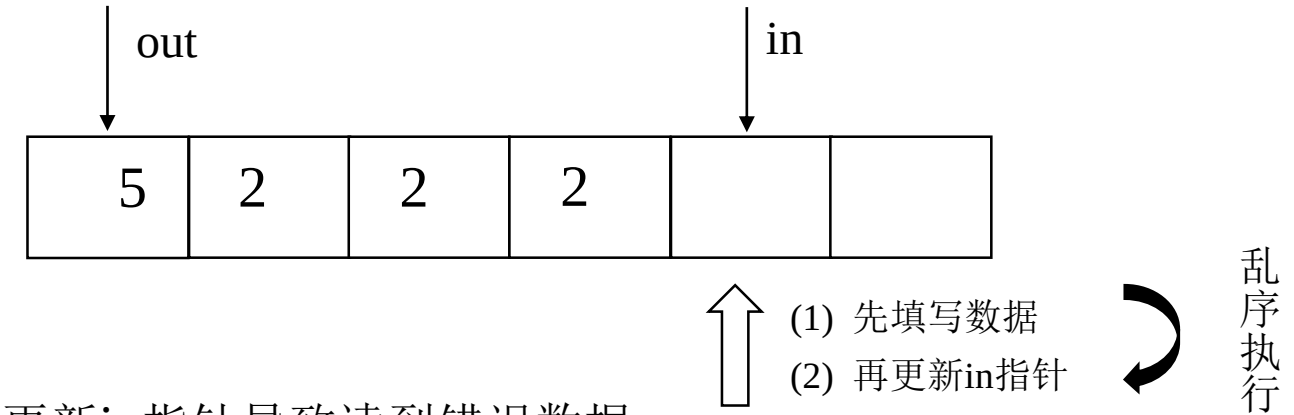
这里介绍一个openEuler OS中使用内存屏障的实例：kfifo队列。其数据结构如下：

```
struct __kfifo {
    unsigned int in;        //写指针
    unsigned int out;       //读指针
    unsigned int mask;      //用于记录队列容量
    unsigned int esize;     //元素大小
    unsigned int *data;     //队列起始位置
}
```

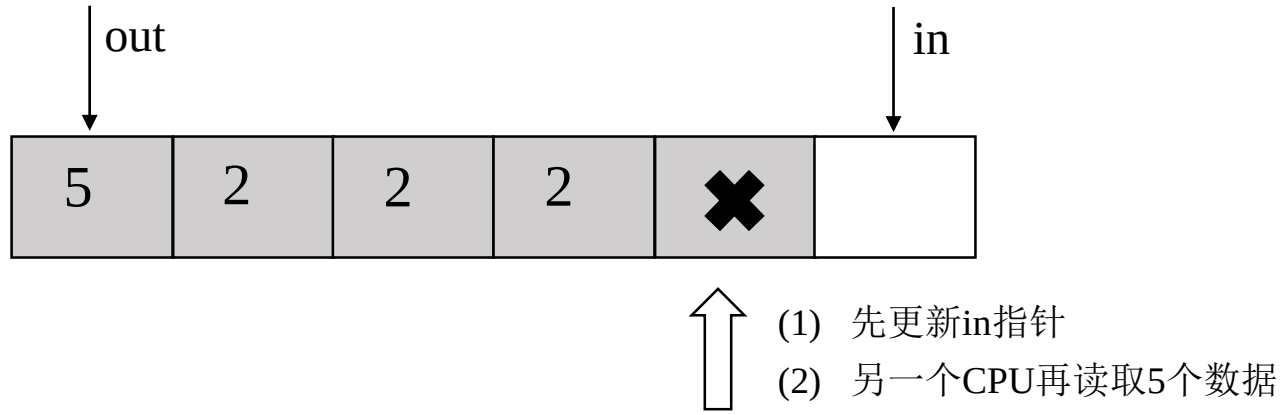
# openEuler内存屏障实例 – kfifo队列 (2)

入队操作：

写场景：




先更新in指针导致读到错误数据：

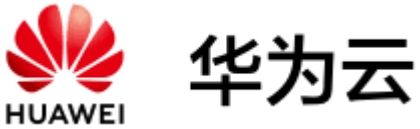


# openEuler开源社区

- https://openeuler.org/


- https://gitee.com/openeuler/


- https://www.huaweicloud.com/kunpeng/



# 思考题

1. 下列有关openEuler说法错误的是？（ ）

   A. 有近10年的技术积累，已广泛用于华为内部产品

   B. 基于Linux 4.19内核

   C. 仅支持Kunpeng处理器

   D. 是一款通用服务器操作系统

2. 下列有关openEuler说法正确的是？（ ）

   A. 提供一种Numa aware解决方案，提升了多核调度性能

   B. 提供鲲鹏加速引擎插件，使能鲲鹏硬件加速能力

   C. 提供iSulad轻量级容器全场景解决方案

   D. 提供了A-Tune智能优化引擎

# 本章总结

- 本章介绍了openEuler操作系统出现的背景、意义和适用场景，并结合鲲鹏处理器硬件基础，介绍了openEuler的架构。本章还对openEuler对通用Linux操作系统的增强进行了介绍。
- 通过本章的学习，应了解并掌握：
  - openEuler出现的背景、意义以及适用场景和通用架构
  - Kunpeng处理器体系结构。对于学有余力者还可了解ARM架构的弱内存序
  - openEuler对通用Linux操作系统的增强
  - openEuler开源社区情况

# 学习推荐

- 《 openEuler操作系统 》·清华大学出版社
- 《 openEuler内核编程技术 》·中国科学院软件研究所
- 《 openEuler应用编程技术 》·中国科学院软件研究所

Thank you.