深入理解计算机

By 公众号@极客重生

内容提要

- 经典问题
- 计算机系统大局观
- CPU
- 内存
- 磁盘
- 回答问题

经典问题

■ 为什么要学习和理解计算机系统

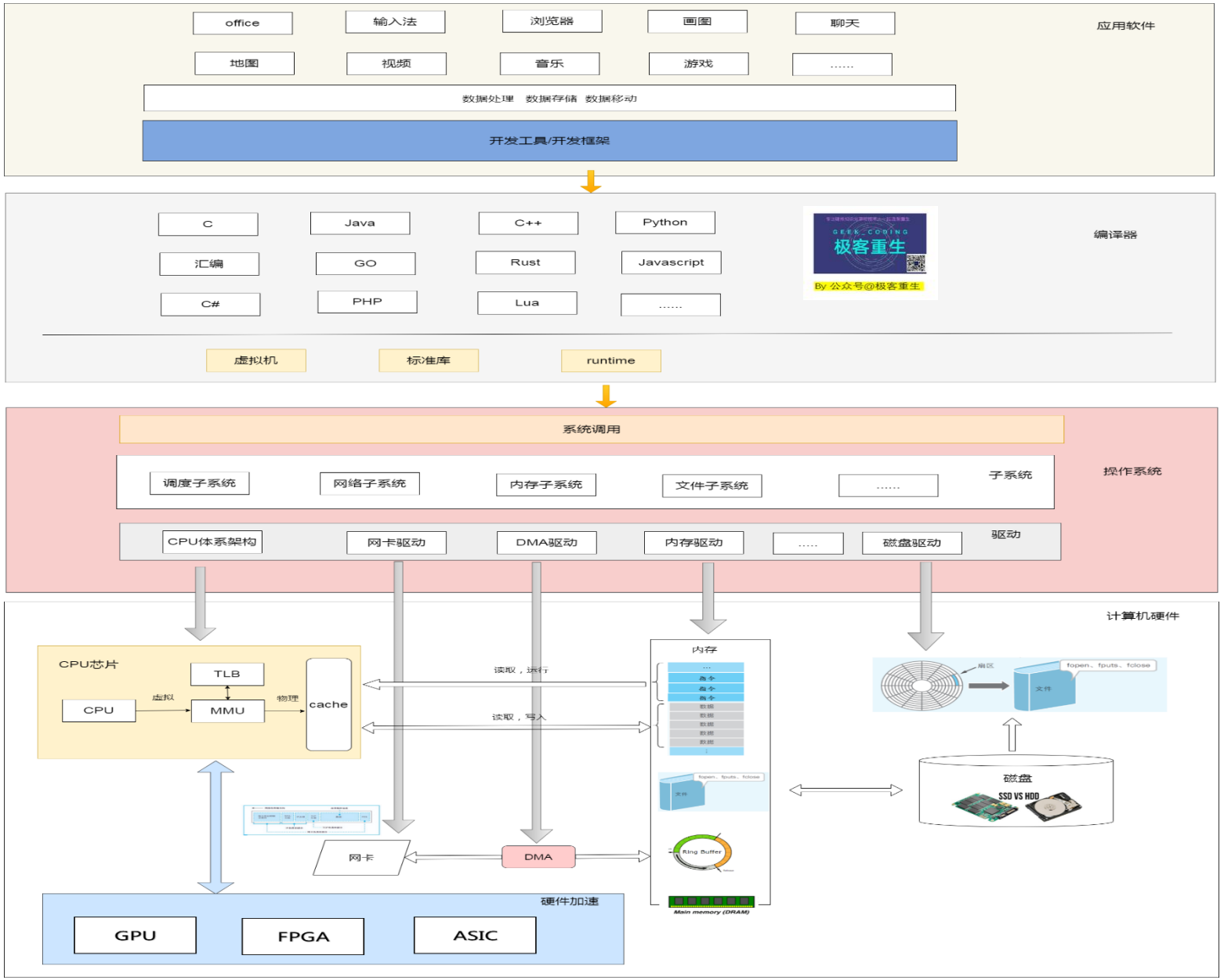
- 能够对软，硬件功能进行合理划分
- 能够对系统不同层次进行抽象和封装
- 能够对系统的整体性能进行分析和调优
- 能够对系统各层面的错误进行调试和修正
- 能够根据系统实现机理对用户程序进行准确的性能评估和优化
- 能够根据不同的应用要求合理构建系统框架等

- 对计算机系统整机概念的认识
- 对计算机系统层次结构的深刻理解
- 对高级语言程序，ISA，OS，编译器，链接器等之间关系的深入掌握
- 对指令在硬件上执行过程的理解和认识
- 对构成计算机硬件的基本电路特性和设计方法等的基本了解等，从而能够更深刻地理解时空开销和权衡，抽象和建模，分而治之，缓存和局部性，吞吐率和时延，并发和并行，远程过程调用(RPC)，权限和保护等重要的核心概念，掌握现代计算机系统中最核心的技术和实现方法.

经典问题

- 为什么要有总线？
- 为什么有缓存？
- 为什么要了解CPU并行技术
- 程序是怎么生成的？
- 程序是如何运行的 ？
- 机器语言（汇编）该这么学习？
- 函数执行原理是什么？
- Crash问题如何解决？
- 内存到底是什么？
- 为什么有虚拟内存机制？
- …

计算机系统大局观

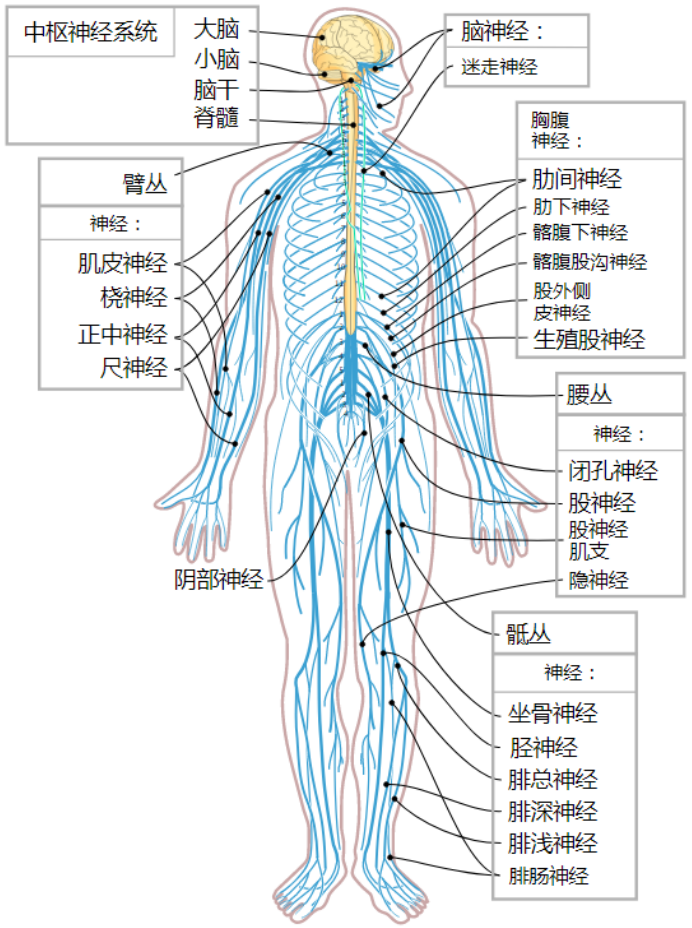


计算机发展的目标

提升处理器速度、减小计算机大小、增加内存大小以及增加 I/O 容量和速度。

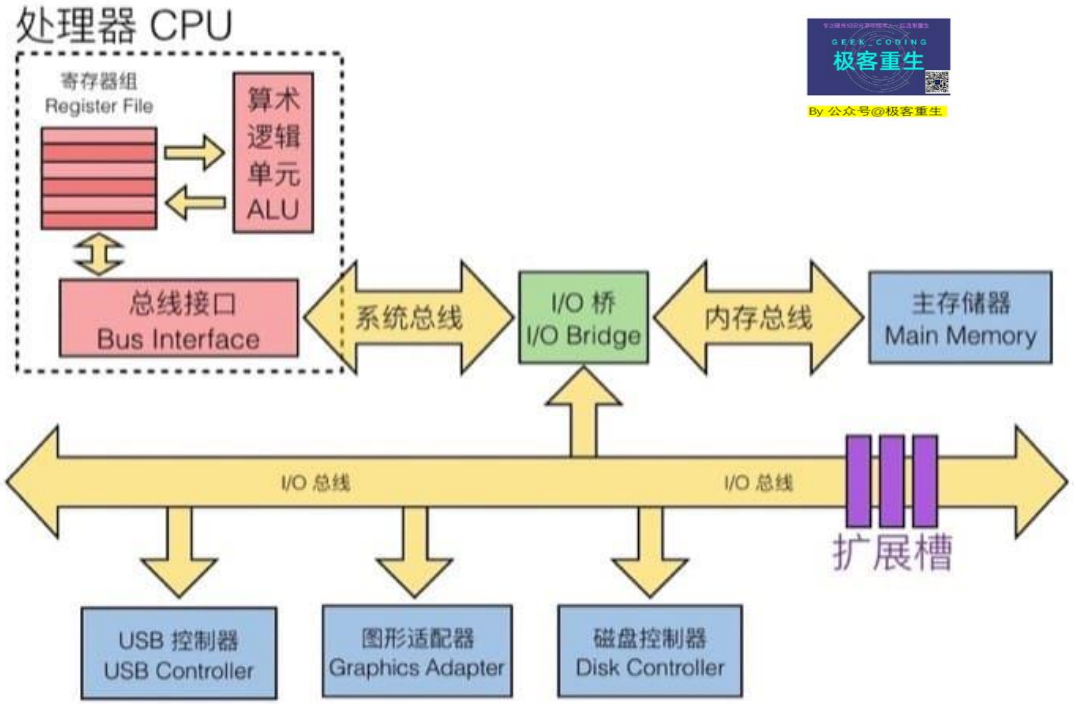
计算机本身就是工具，所以一切皆为工具，我们只需要用好这些工具。

为什么要有总线？



总线：计算机的神经系统

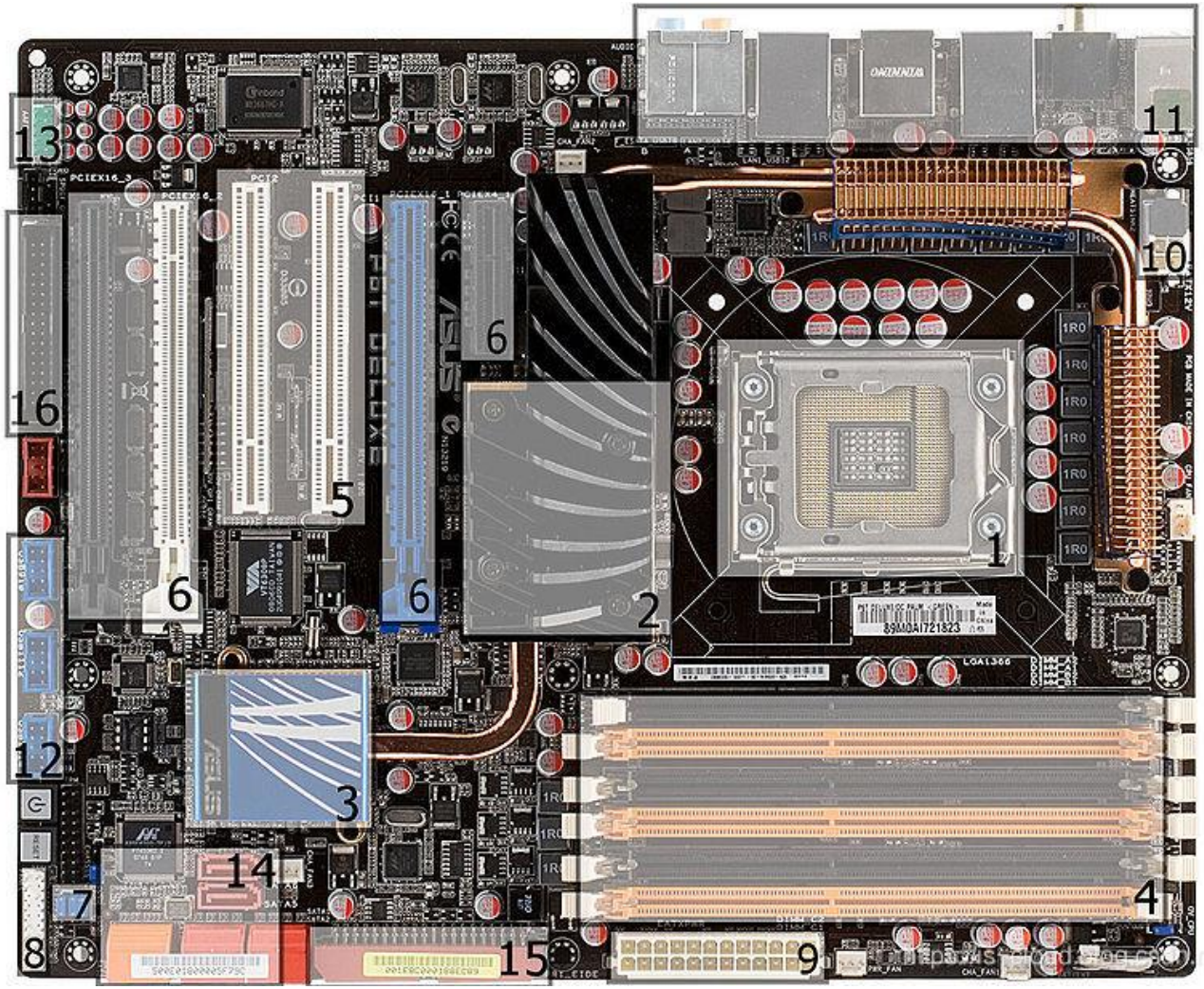
“要致富先修路”



- 总线是计算机中模块之间的通信线路。总线连接了CPU、内存、外设等计算机中的主要模块
- 计算机的总线到底是什么？
- 为什么有不同的总线？
- 软件编程需要注意什么？

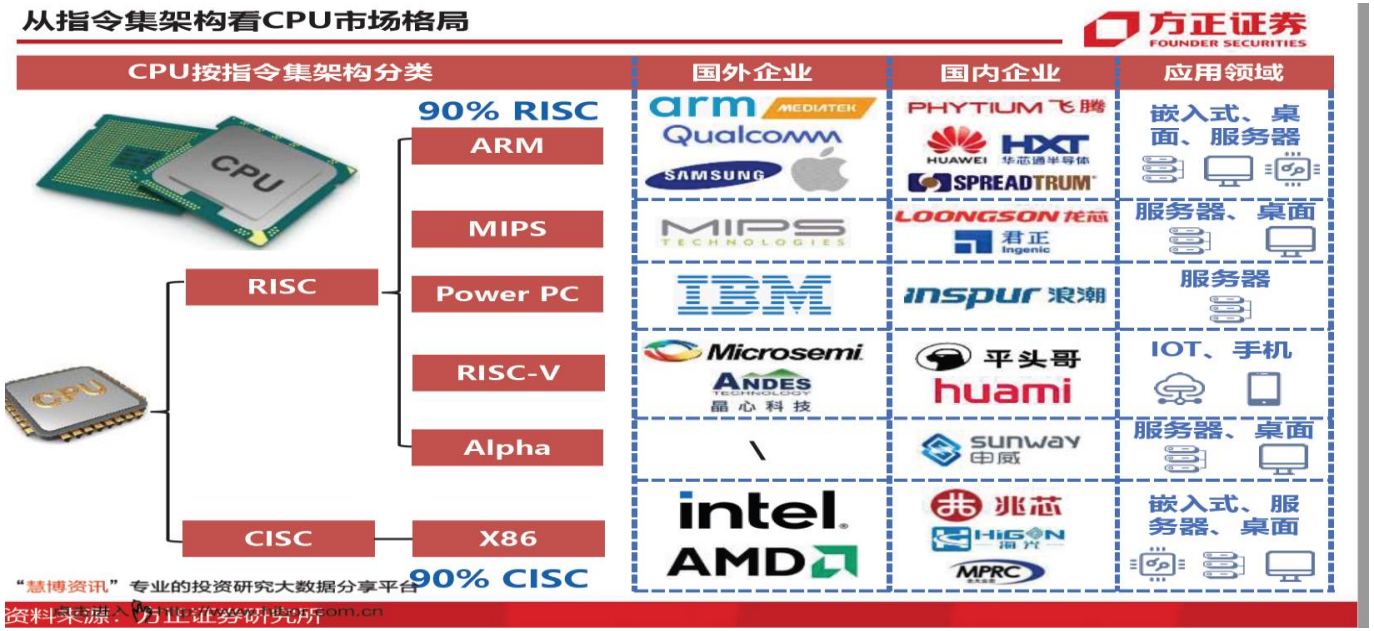
ISA总线 → PCI总线 → AGP 总线 → PCIE总线 → PCIE总线1.0->2.0->3.0->4.0->5.0->....

计算机结构（主板）



1.CPU 插槽（LGA 1366）
2.北桥（被散热片覆盖,负责与CPU通信，并且连接高速设备（内存/显卡））
3.南桥（被散热片覆盖.负责与低速设备（硬盘/USB）通信，时钟/BIOS/系统管理/旧式设备控制，并且与北桥通信）
4.记忆体插座（三通道）
5.PCI 扩充槽（网卡等）
6.PCI Express 扩充槽(独立显卡等)
7.跳线
8.控制面板（开关掣、LED 等）
9.20+4pin 主机板电源
10.4+4pin 处理器电源
11.背板 I/O
12.前置 USB 针脚
13.前置面板音效针脚
14.SATA 插座
15.ATA 插座（大部分 Intel Sandy Bridge 以后的家用主板都已舍弃 IDE 介面）
16.软碟机插座（目前绝大多数主板已舍弃软碟机介面）

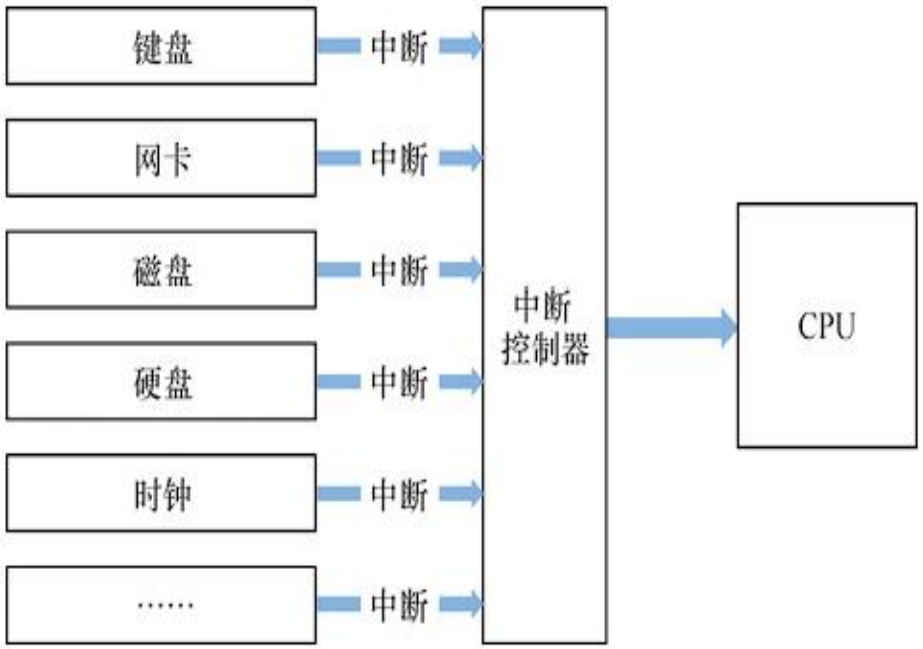
作为软件开发，我们为什么要熟悉CPU（芯片）原理



	CPU	GPU	FPGA	ASIC
特性	通用类计算、串行计算	数据依赖度低的高密度计算	无指令系统、软硬件一体化，门电路复杂冗余，可重复编程	无指令系统、软硬件一体化，门电路优化后精简，不可重复编程
编译系统	需要	需要	不需要	不需要
编译难度	低	低	高	高
开发周期	短	短	较长	长
运行主频	高	高	低	低
能耗	高	高	低	低
易用性	强	较强	较弱	弱

1. 进行性能优化，只有理解各类芯片优缺点，才能让软件跑得更快，性能最强。
2. 技术选型，成本和性能目标，选择合适的计算芯片
   - CPU Intel X86/64
     优势：单核性能强，稳定性较强，技术先进
     劣势：功耗高，价格贵
   - CPU AMD X86/64
     优势： 高性能，价格优惠，核多
     劣势： 单核性能没有intel强，稳定性比intel差
     适合云计算虚拟化场景
   - CPU ARM/64
     优势：低功耗、低成本、小体积
     劣势：同等级性能低，软件兼容性
   - GPU
     优势：高性能，并行计算强
     劣势： 通用性差，功耗高
   - FPGA
     优势：高性能，低功耗
     劣势：编程难度高
3. 了解CPU特性对程序的影响
   - 指令乱序
   - 分支预测
   - 预取(空间局部性)
   - 缓存(时间局部性)

为什么有中断？

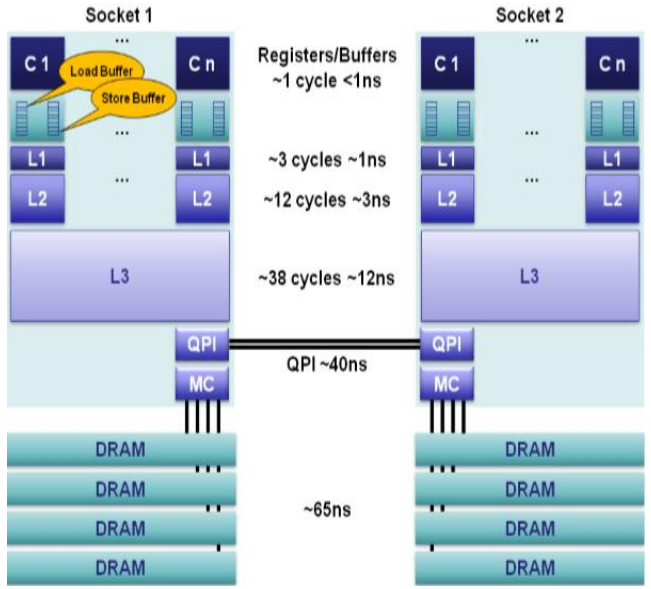


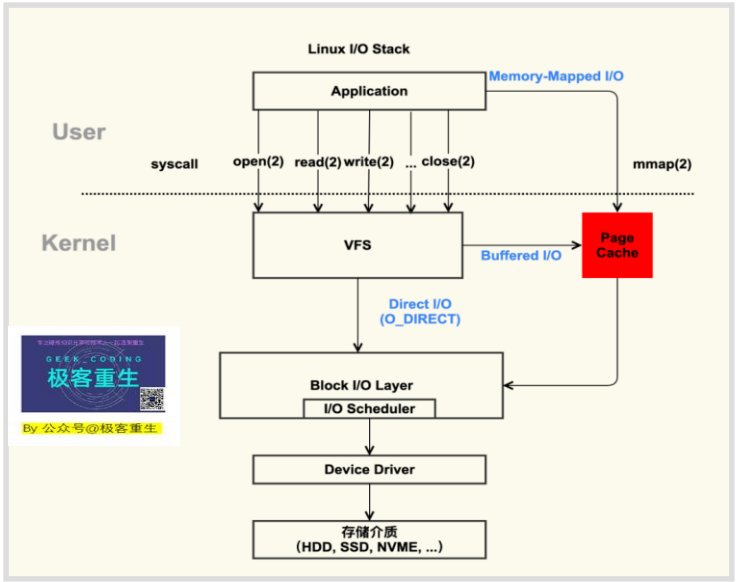
中断处理机制

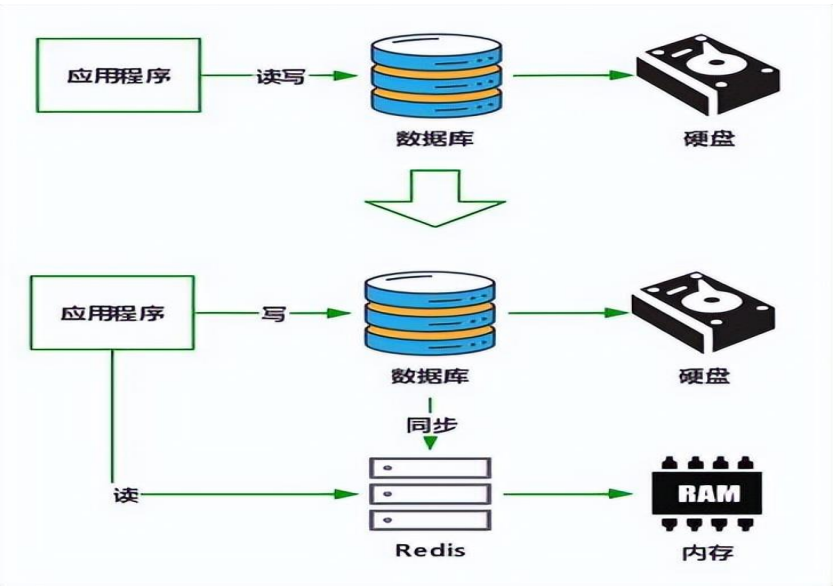
- 中断是CPU和外界的一种高效合作手段。CPU不需要时刻检查外围设备是否有状态变化，而是将绝大部分时间用于执行软件程序，只有在设备收到数据时才主动向CPU发出“通知”。在早期不提供中断支持的CPU中，CPU只能采用“轮询”（Polling）机制，定期检查外围设备的状态来判断是否有数据要处理，但这样会使CPU付出额外的不必要时间。
- 每一种CPU都在设计时规定了可以响应的中断，通常是在CPU芯片的引脚中定义用于响应的中断信号。外围设备需要向CPU发出中断时，向CPU芯片的引脚发送电平信号。CPU一般是在流水线的最后一级——提交（COMMIT）阶段检查引脚上的电平信号，用来判断是继续执行指令还是处理中断。
- CPU对中断的处理机制一般是停止流水线取指行为，CPU本身切换到操作系统特权级，调用操作系统中专门的软件模块“中断服务程序”（Interruption Service Routine，ISR）来处理中断。ISR检查是哪种外围设备发生了中断，再从设备中接收相应的数据。ISR处理完外围设备的数据后，把CPU切换回应用程序特权级，再跳回应用程序继续执行。
- 中断和异常都是使CPU停止执行当前指令的机制，两者的根本区别在于，异常是由CPU内部指令执行触发（例如除法指令的除数是0），而中断是由CPU外部信号触发。

- 硬中断和软中断
- 中断上下文
- 中断运行代码有哪些限制呢？
- 中断性能优化

为什么有缓存？







CPU缓存 → IO缓存 → 应用缓存

缓存设计的关键问题是什么？

- 数据不一致问题 → 数据同步

Cache一致性协议MESI实现了多个CPU之间的Cache同步。但是不同计算机对Cache更新通知的时序规定了不同的原则。

- **强一致性**：系统中所有更新Cache的通知要执行结束，才允许各CPU执后续的访存指令。这种方式使所有处理器核之间严格保证Cache一致性，但是会使各CPU花费大量时间等待Cache通知结束，从而降低了系统性能。
- **弱一致性**：各CPU不需要等待所有Cache通知执行结束，就可以执行访存指令。在这种情况下，CPU硬件不维护所有Cache的制一致性，某一个CPU写内存的行为可能不会及时通知到所有其他CPU，这时不同的CPU会在Cache中读取出不同数值。

- 增加系统复杂度

- 大多数CPU是什么原则，为什么？
- 这里一致性让你想到了什么？

为什么要了解CPU并行技术

流水线 → 超线程 → 向量指令 → 多核 → NUMA

- 流水线（提高CPU性能，指令乱序执行），有些并发场景需要防止乱序执行（内存屏障）

- 在CPU足够且要求高性能场景下一般会关闭超线程，硬件多线程也会争抢硬件公共资源

- 确认硬件是否支持（lscpu）支持哪些向量指令
- 需要熟悉编译器并行优化

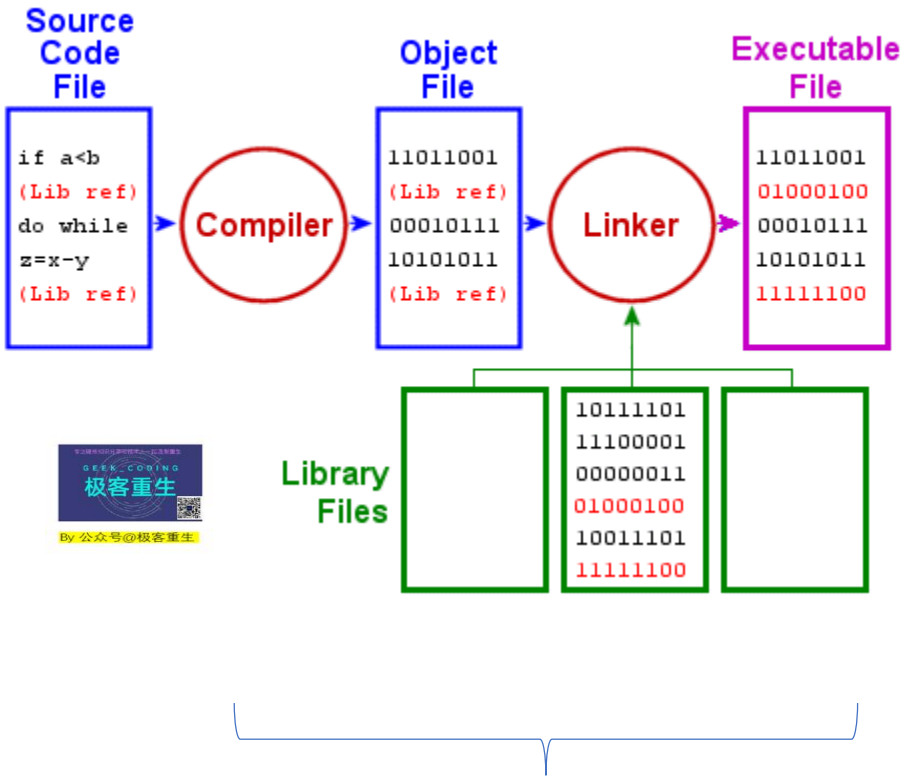
- 需要深刻理解多核并发设计
- 原子操作和锁机制

- 优化性能注意NUMA亲和和CPU访问内存时延
- 内存分配和均衡机制

软件开发需要注意什么？

程序是如何生成的？编译，连接，库

源代码（source code）→ 预处理器（preprocessor）→ 编译器（compiler）→ 汇编程序（assembler）→ 目标代码（object code）→ 链接器（linker）→ 可执行文件（executables）



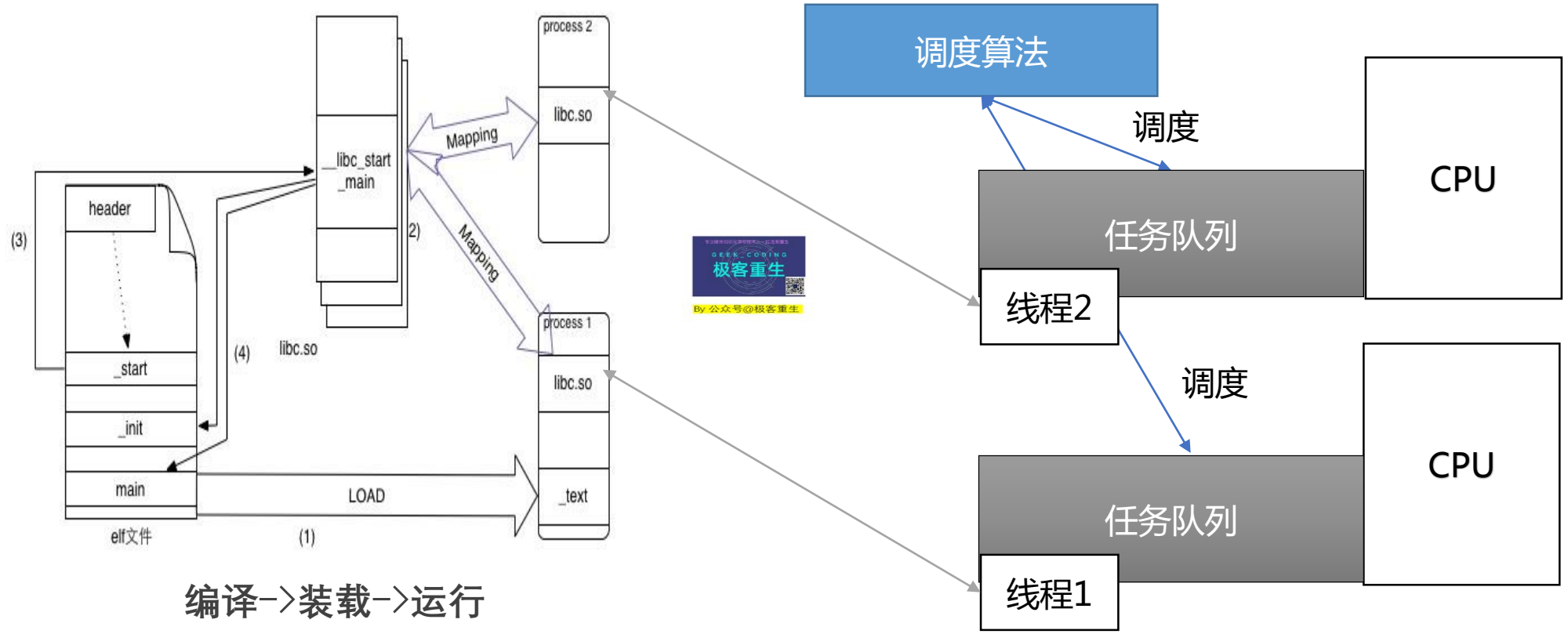
编译
- 头文件/包/引用
- 编译原理有什么用？
- 为什么需要设计DSL？
- 为什么设计芯片需要设计（修改）编译器？
- 编译优化有哪些？

连接
- 静态连接 vs 动态连接
- 符号是什么？

库
- 静态库和动态库
- ABI是什么？为什么很重要？
- 热升级

Build（Makefile，你还知道哪些编译构建工具？）

程序是如何运行的？



编译->装载->运行

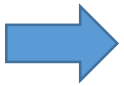
1. 首先 bash 进行 fork 系统调用，生成一个子进程，接着在子进程中运行 execve 函数指定的 elf 二进制程序（Linux中执行二进制程序最终都是通过 execve 这个库函数进行的），execve 会调用系统调用把 elf 文件 load 到内存中的代码段(_text)中。
2. 如果有依赖的动态链接库，会调用动态链接器进行库文件的地址映射，动态链接库的内存空间是被多个进程共享的。
3. 内核从 elf 文件头得到_start的地址，调度执行流从_start指向的地址开始执行，执行流在_start执行的代码段中跳转到libc中的公共初始化代码段__libc_start_main，进行程序运行前的初始化工作。
4. __libc_start_main的执行过程中，会跳转到_init中全局变量的初始化工作，随后调用我们的main函数，进入到主函数的指令流程。

机器语言（汇编）该这么学习？

类　型	功　能
数据转送指令	寄存器和内存、内存和内存、寄存器和外围设备[1]之间的数据读写操作
运算指令	用累加寄存器执行算术运算、逻辑运算、比较运算和移位运算
跳转指令	实现条件分支、循环、强制跳转等
call/return 指令	函数的调用 / 返回调用前的地址

机器语言指令的主要类型和功能


```
1  // Type your code here, or load an example.
2  int square(int num) {
3      int local_a = num + 1;
4      return local_a * local_a;
5  }
6  int main()
7  {
8      int a = 1;
9      int c;
10     c = square(a);
11     return 0;
12 }
```



```
1   square:
2           pushq   %rbp
3           movq    %rsp, %rbp
4           movl    %edi, -20(%rbp)
5           movl    -20(%rbp), %eax
6           addl    $1, %eax
7           movl    %eax, -4(%rbp)
8           movl    -4(%rbp), %eax
9           imull   %eax, %eax
10          popq    %rbp
11          ret
12  main:
13          pushq   %rbp
14          movq    %rsp, %rbp
15          subq    $16, %rsp
16          movl    $1, -4(%rbp)
17          movl    -4(%rbp), %eax
18          movl    %eax, %edi
19          call    square
20          movl    %eax, -8(%rbp)
21          movl    $0, %eax
22          leave
23          ret
```

Register	Purpose	Saved across calls
%rax	temp register; return value	No
%rbx	callee-saved	Yes
%rcx	used to pass 4th argument to functions	No
%rdx	used to pass 3rd argument to functions	No
%rsp	stack pointer	Yes
%rbp	callee-saved; base pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10-r11	temporary	No
%r12-r15	callee-saved registers	Yes

常见的64位寄存器用法

x86-64 Integer Registers (8 of 16)

63	31	15	7	0
%rax	%eax	%ax	%al	
%rbx	%ebx	%bx	%bl	
%rcx	%ecx	%cx	%cl	
%rdx	%edx	%dx	%dl	
%rsi	%esi	%si	%sil	
%rdi	%edi	%di	%dil	
%rbp	%ebp	%bpx	%bpl	
%rsp	%esp	%sp	%spl	

机器语言（汇编）该这么学习？

大师兄| 荣哥

2022-04-28 16:32

收进专栏 ···

汇编学习资料

x86-64-architecture-guide

🔗 X86-64 Architecture Guide

汇编在线显示工具：

🔗 Compiler Explorer

汇编学习网站：

🔗 【AT&T风格汇编语言】2019天津大学智算学部汇编语言程序设计-李罡_哔哩哔哩_bilibili

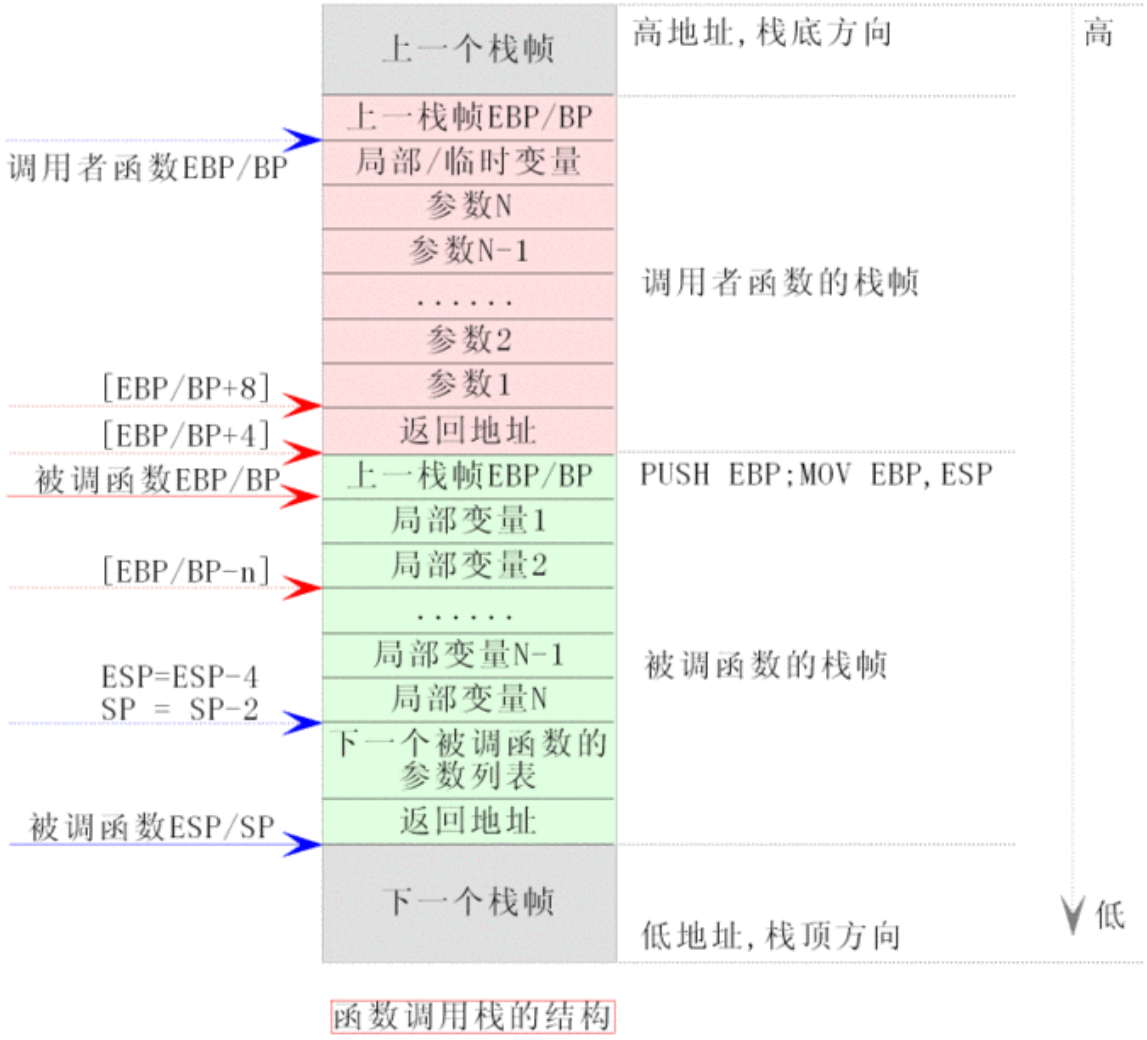
🔗 汇编语言从0开始 重制版 自学必备(配套王爽汇编语言第三版或第四版)_哔哩哔哩_bilibili#基本功# #核心技术#

X86.pdf

assembly64.pdf

基本功 核心技术

函数调用原理-堆栈原理



- push
- pop
- SP
- BP
- call
- ret

- 函数间如何转移控制（指令跳转）
- 函数间如何传递数据（参数和返回值）
- 寄存器的保存与恢复（调用者和被调用者）
- 局部变量的存储
- 栈帧的初始化与销毁

解决问题高手-Crash问题如何解决？

常见问题

- CPU-hung软锁问题（softlockup: hung tasks）
- 内存非法访问（空指针，非法页访问）
- 触发内核BUGON（内核主动crash，代码执行状态不符合预期）
- CPU指令跳变（堆栈逻辑错误）

核心思想：

大胆猜测+搜索，细心推导参数

和当前函数里面变量的值，然后

结合反汇编和异常指令等，综合

推导出代码逻辑错误或者非软件

问题等

- 了解堆栈原理, bt 查看堆栈，bt –ff/-FF 查看详细堆栈信息,bt –a 查看所有CPU堆栈信息；
- 了解x86_64汇编原理，各个寄存器作用，有些寄存器可以被刷掉，需要去推导堆栈信息；
- 参数推到：整数参数（包括指针）按顺序放在寄存器%rdi，%rsi，%rdx，%rcx，%r8和%r9中，函数的返回值存储在%eax中；
- struct 命令打印数据结构信息和字段偏移等，配合内存地址可以打印当前结构内存数据，方便对照汇编代码进行推理逻辑；
- 对照代码尝试查看全局变量或者per CPU数据得到一些系统运行信息，方便定位问题，比如时钟中断统计，watchdog信息，CPU运行队列等；

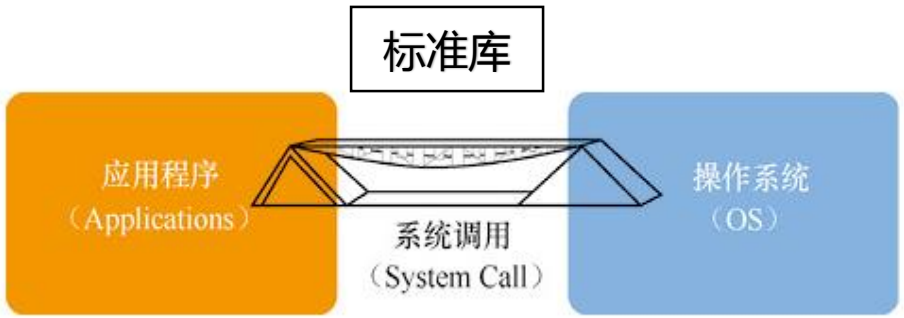
核心：深入理解堆栈原理

如何从堆栈中查找

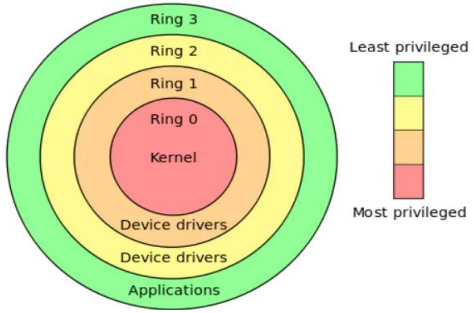
- 被压栈的寄存器参数
- 本地变量

进而找到函数调用参数，然后走一遍代码，便能找出bug

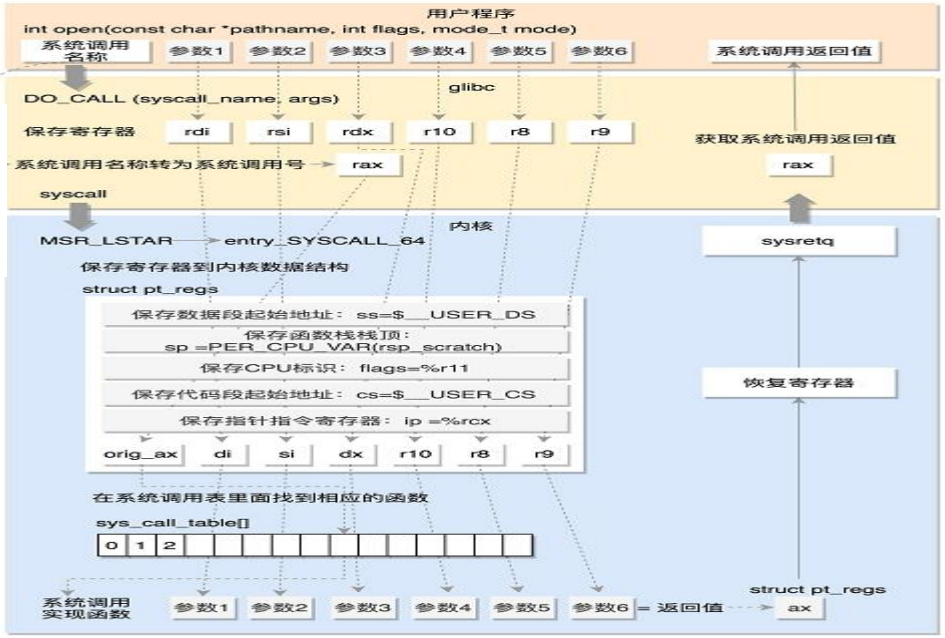
什么是系统调用？



操作系统和应用的桥梁



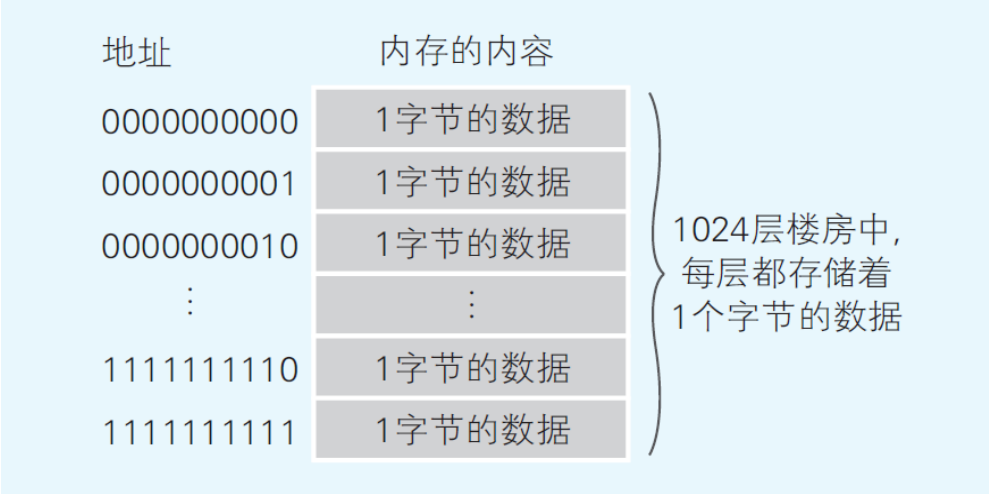
- 内核空间（Ring 0）
- 用户空间（Ring 3）

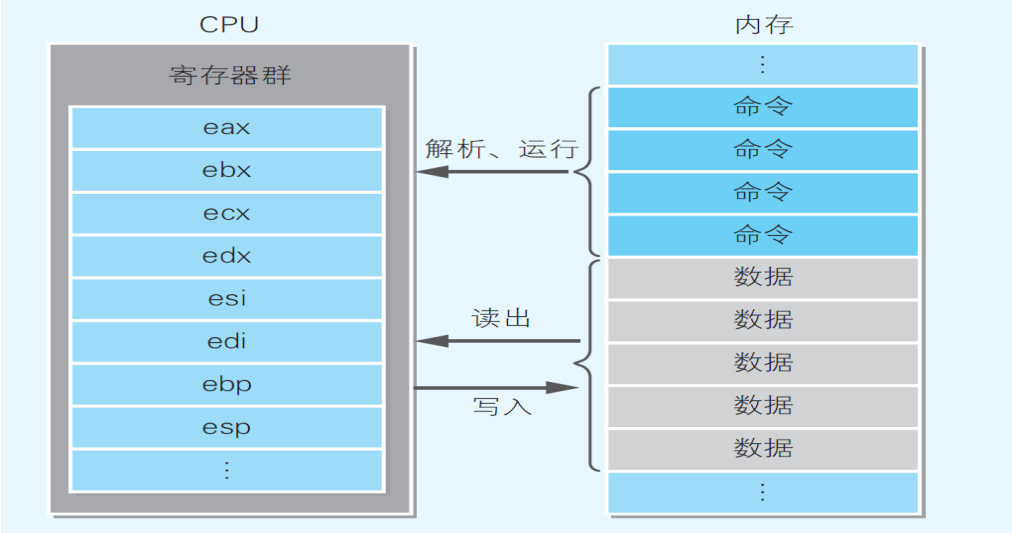


- 系统调用接口一般在哪里实现？
- 系统调用和标准库的关系？
- 如何看待系统调用机制，对我们有什么启示？

- 切换时先保存CPU寄存器中的用户态指令
- 再重新更新内核态指令位置
- 最后跳转到内核态运行内核任务
- 当系统调用结束后需要恢复原来保存的用户态

内存到底是什么



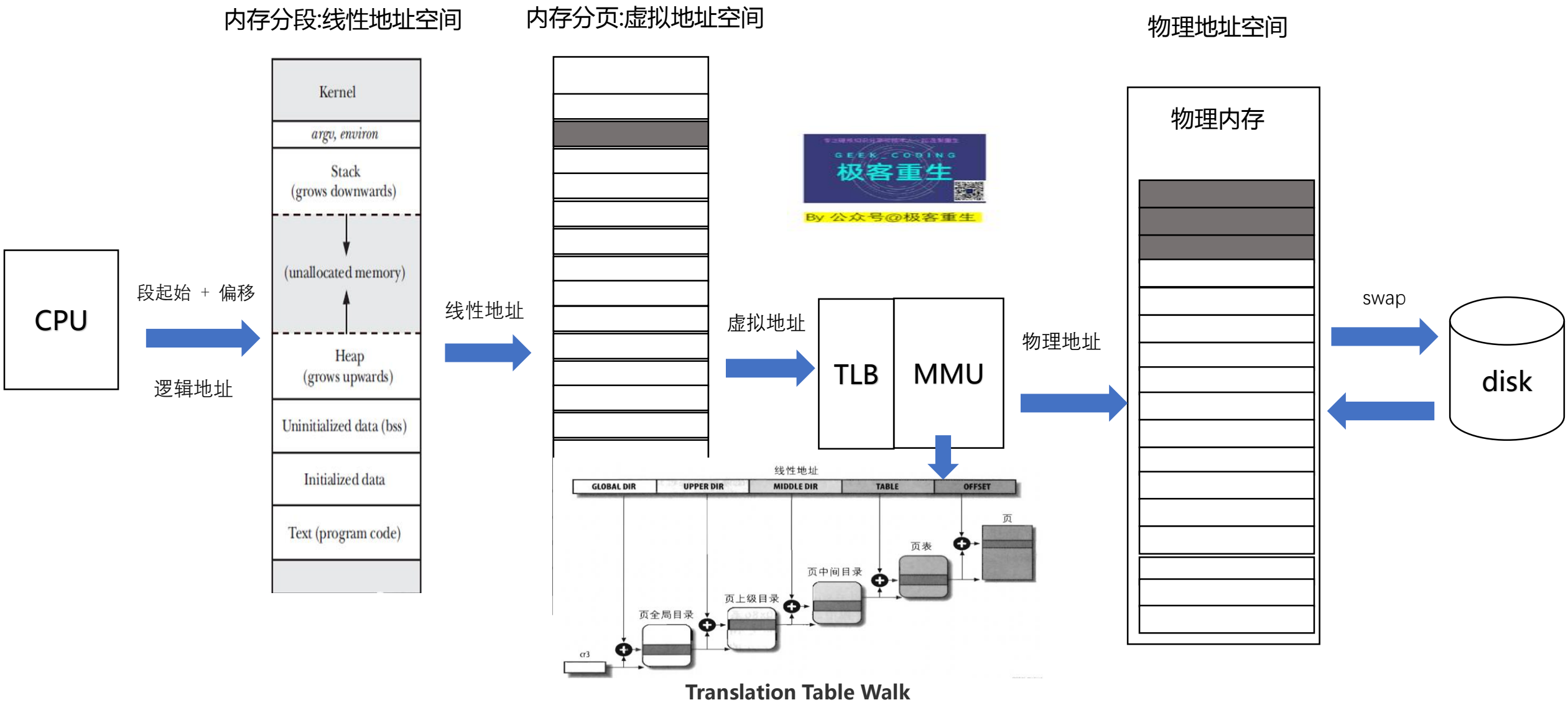


二进制数的位数一般是8 位、16 位、32 位……也就是8 的倍数，这是因为计算机所处理的信息的基本单位是8位二进制数。8位二进制数被称为一个字节。字节是最基本的信息计量单位。位是最小单位，字节是基本单位。内存和磁盘都使用字节单位来存储和读写数据，使用位单位则无法读写数据。因此，字节是信息的基本单位。

指令部分	数据部分						
10110000	10101011	00110001	00011100	00110011	00111110	01010011	11110000
输出指令	V	I	P	会		员	

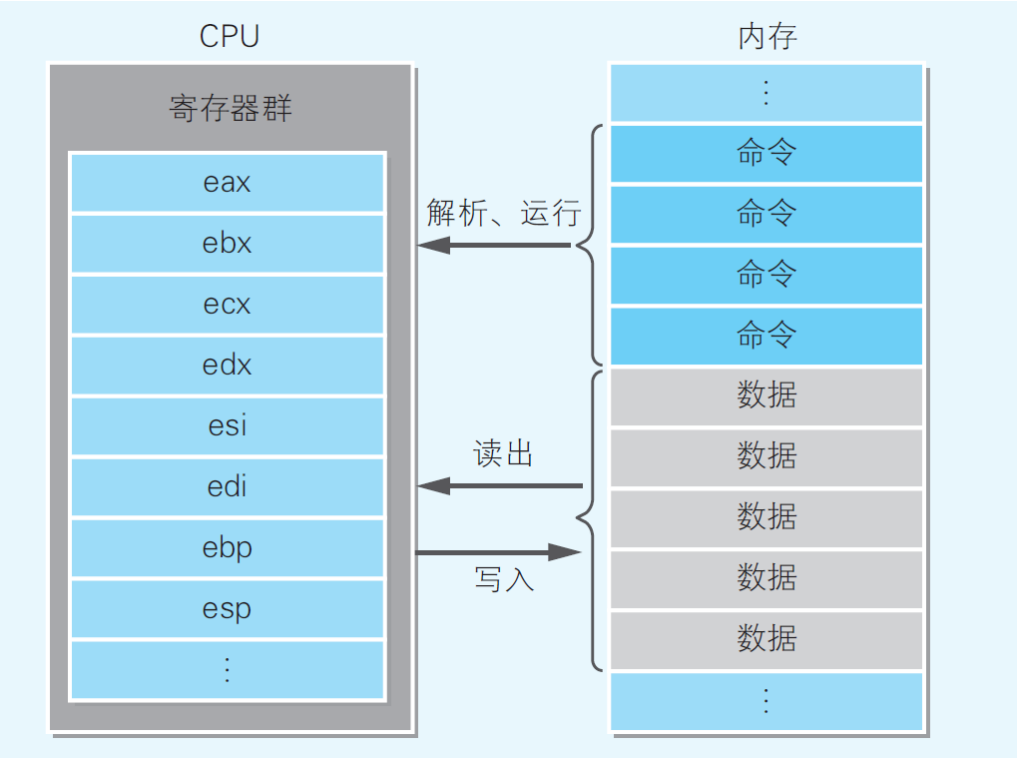
- 内存就是地址，类型就是一次读写的长度和解释
- 指针就是内存地址，指针类型就是一次读写的长度和解释
- 变量的数据类型不同，所占用的内存大小也不一样

内存管理



Translation Table Walk

为什么有虚拟内存？



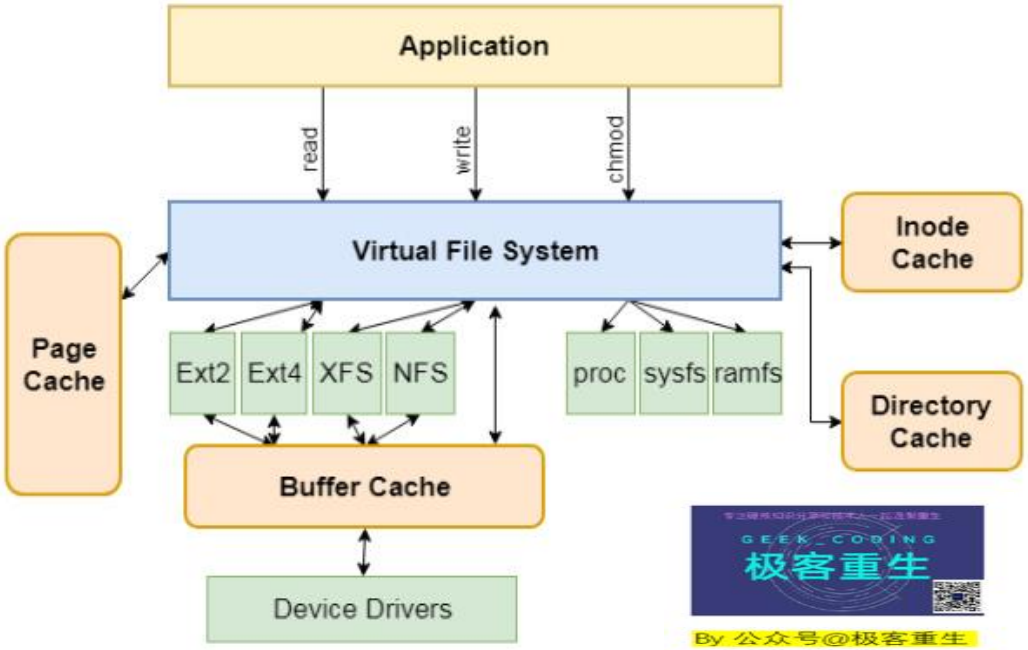
CPU和内存的关系

- 内存保护机制
- Lazy Allocation
- 扩展内存容量（磁盘等）

磁盘

应用（进程）

操作系统



磁盘

- 提高写性能
  - 数据是被整体访问，比如HDFS
  - 知道文件明确的偏移量，比如Kafka
  - 使用LSM-tree，比如level DB

追加写+数据有序+顺序IO+延迟更新+批量写入

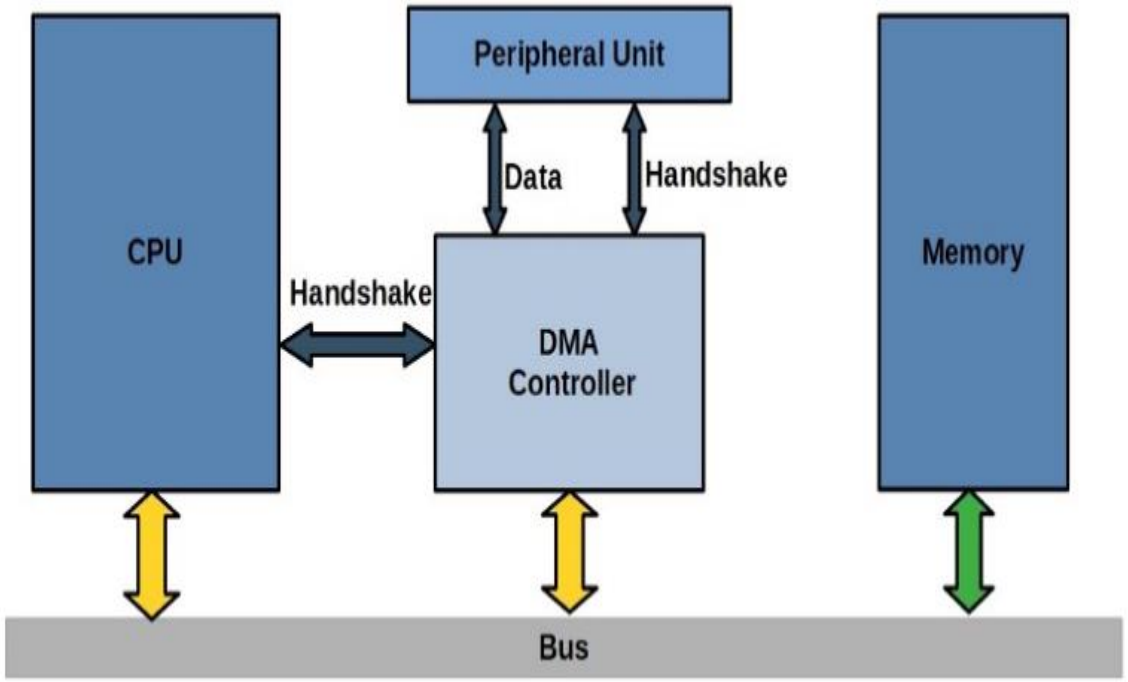
- 提高读性能
  - B+树索引-如MySQL
  - 文件数据有序保存，二分查找，如Kafka

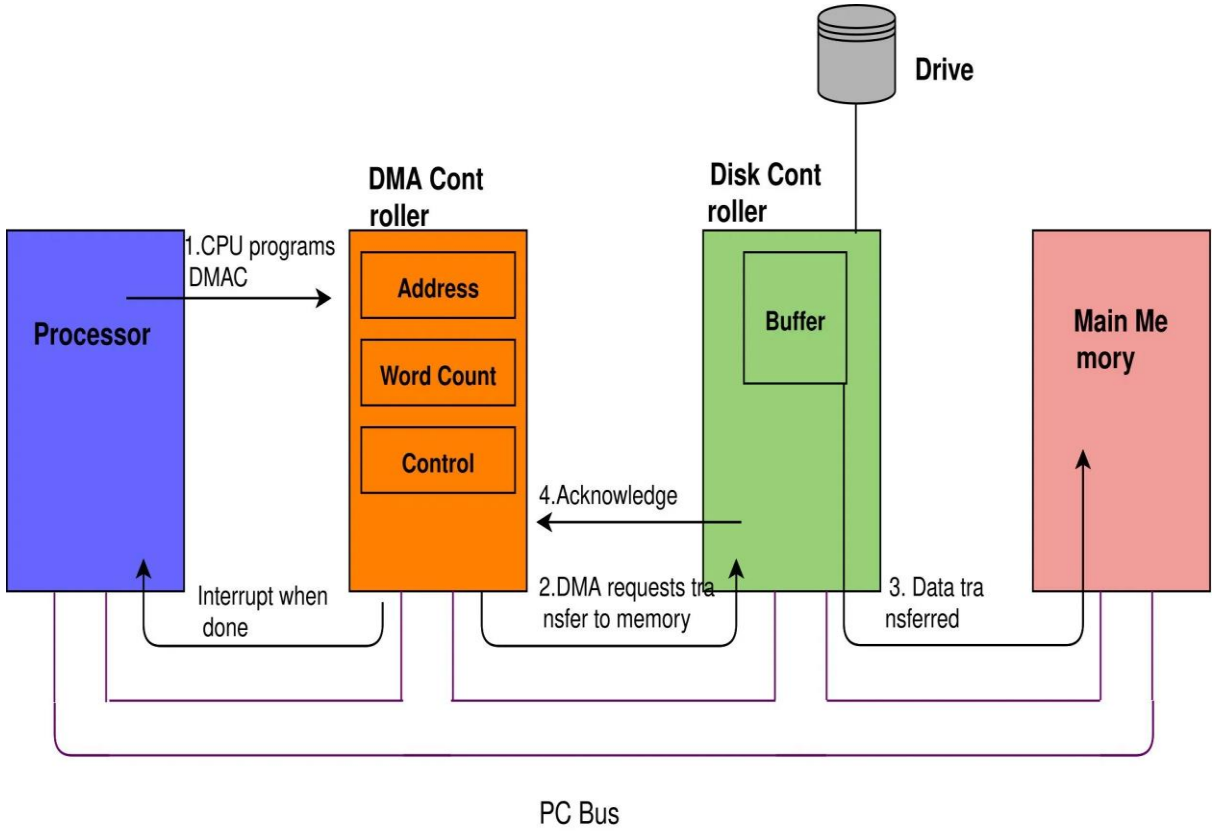
顺序读 + 缓存 + 零拷贝+ 分流（分库分表）

磁盘优化

- SSD 替代 HDD
- 使用 RAID （磁盘阵列）
- 调整 I/O 调度算法
- 多个磁盘
- 调整预读块大小/磁盘队列长度

DMA offload CPU ：设备和内存之间传输数据



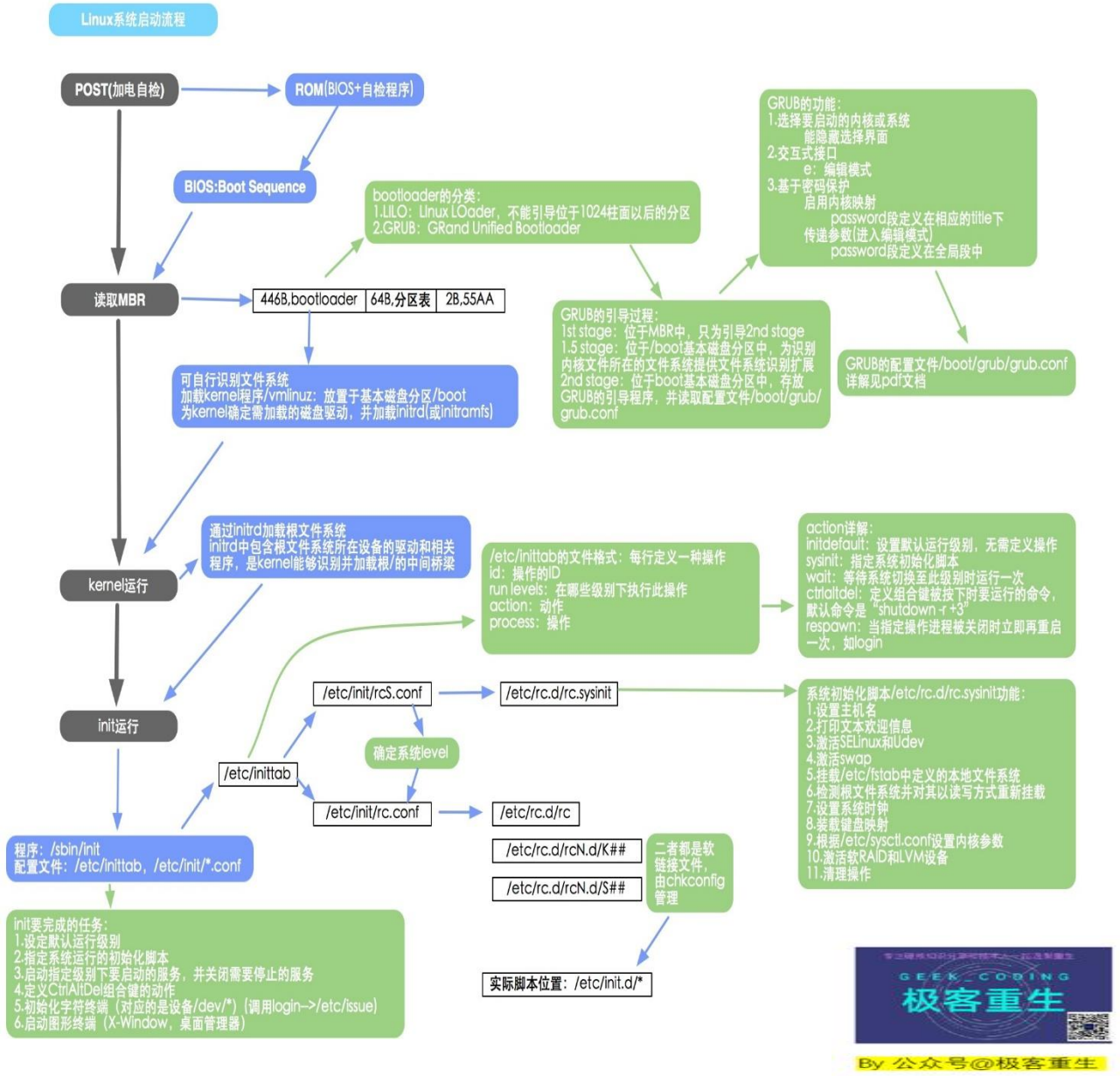


DMA (Direct Memory Access) 。 DMA是指在不通过CPU的情况下，外围设备直接和主内存进行数据传送。磁盘等都用到了这个DMA机制。

DMA offload CPU among memory and device

DMA:
Periph == Bus ==> RAM
No DMA:
Periph == Bus ==> CPU == Bus ==> RAM

操作系统是如何启动的?



电源

BIOS自检

- 检查硬件
- 执行MBR，跳转到Boot loader

系统引导：Boot loader

- 初始化硬件设备
- 加载内核镜像（内核代码），根文件系统和配置（DTS），引导内核启动

启动内核：Linux kernel

- 操作系统初始化
- 启动初始化进程

初始化系统：File system

- init初始化（systemd）
- /etc/rc.d/rc*.d

在输入框中敲下按键，计算机里发生了什么？

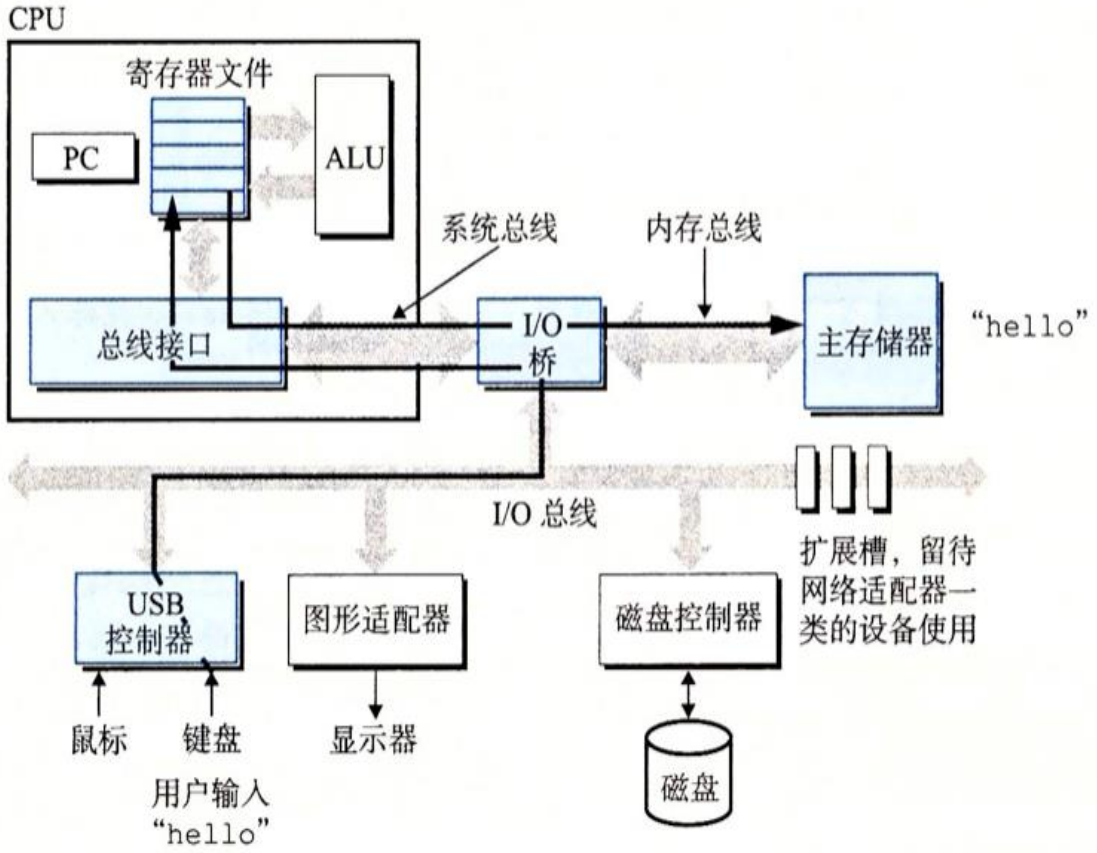
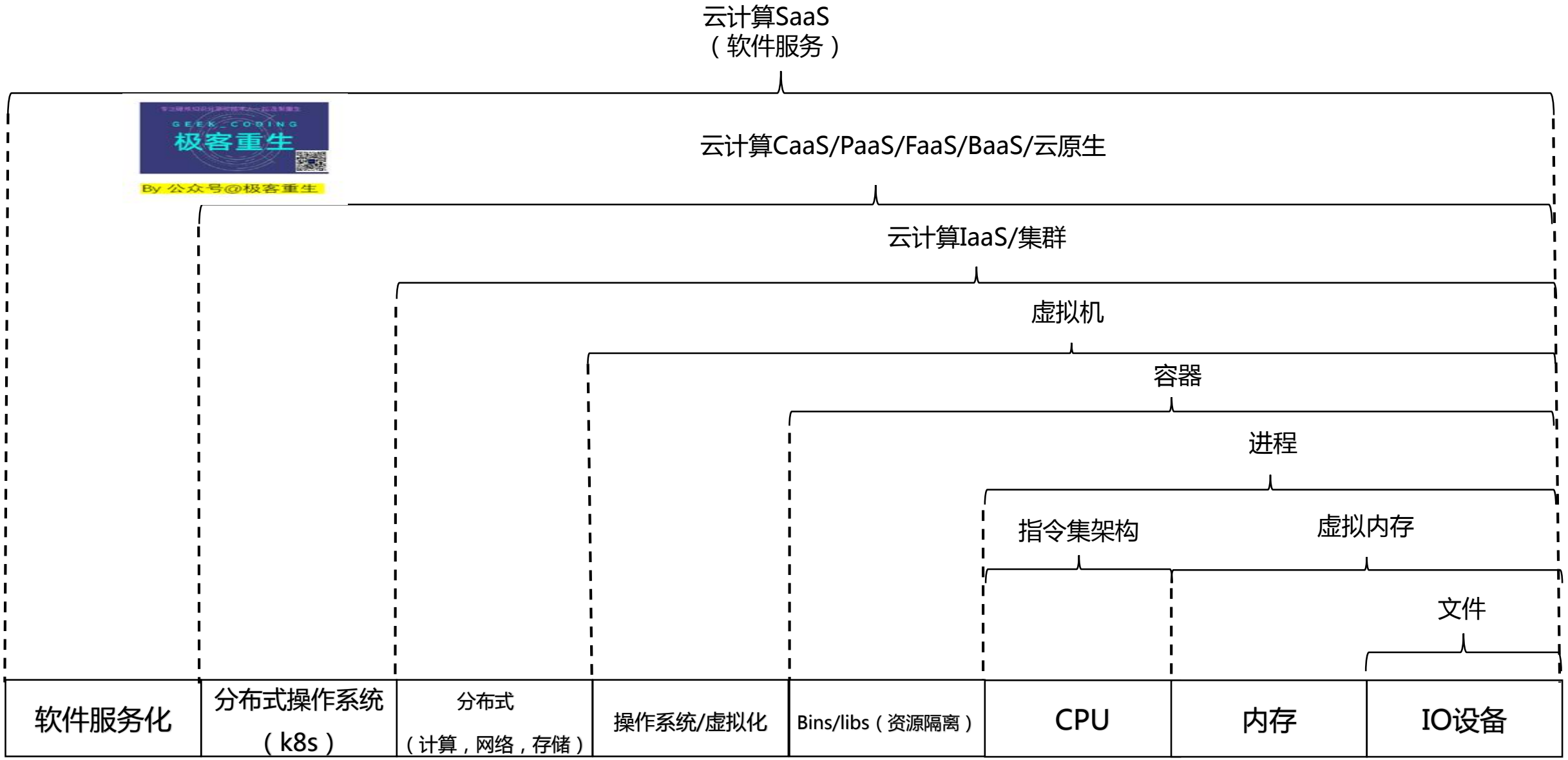
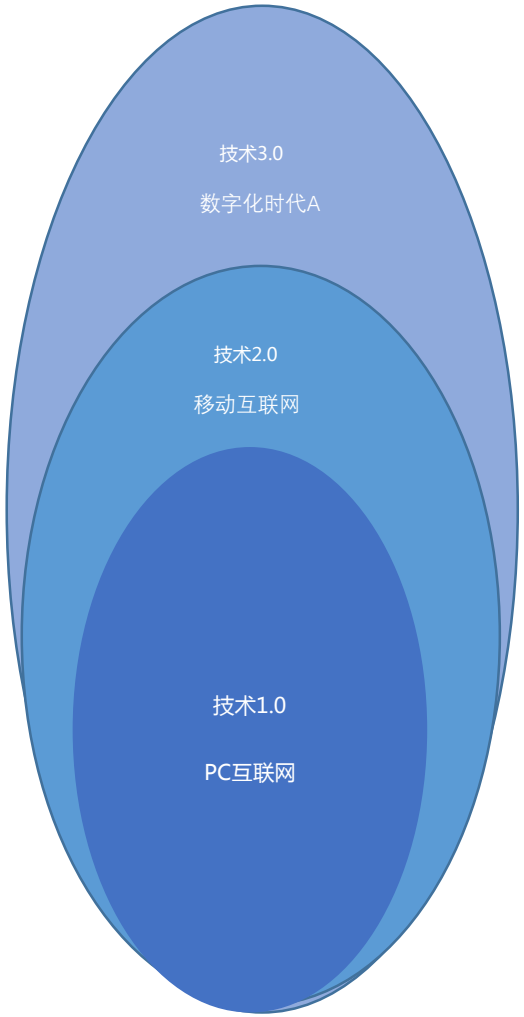


图 1-5　从键盘上读取 hello 命令

计算机系统中抽象的重要性



我们要做什么准备？ 迎接下一次IT技术革命的到来



第一次IT科技革命 PC+互联网 从1994年发展至今

核心产品：PC + 软件 + 门户网站

核心技术：
软件: window + C++（办公+游戏）， Linux + C (开源软件/工具）
网站： Linux/Apache/Mysql/PHP/JavaScript/tcp/ip

第二次IT科技革命 移动互联网 从2008年到现在

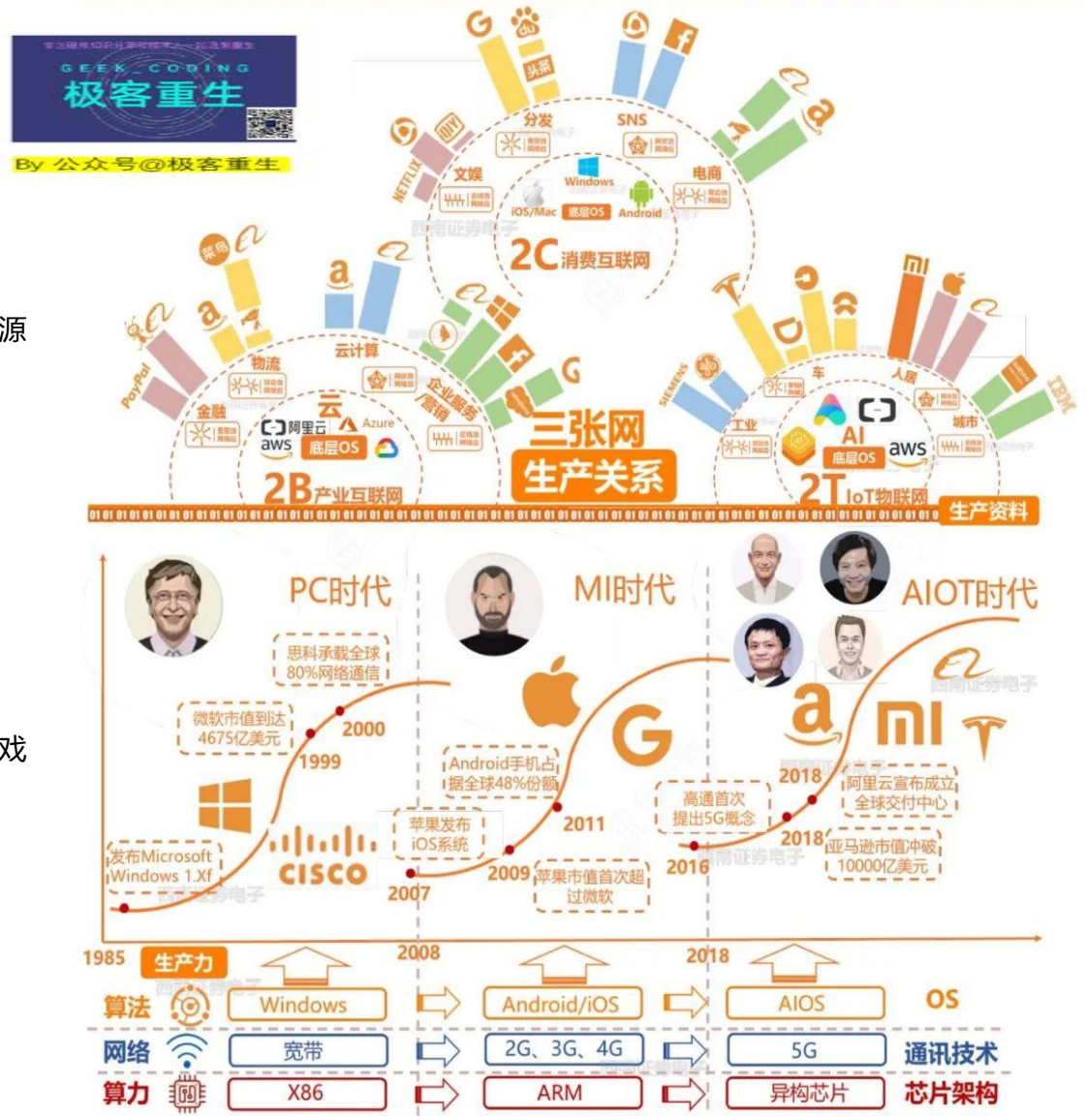
核心产品：手机 + APP

核心技术：
手机: (芯片+硬件)/Andriod/IOS/通信（3G/4G）
APP: JAVA/Linux高并发后台技术/音视频/社交/移动游戏

第三次IT科技革命 数字化时代 现在~未来？

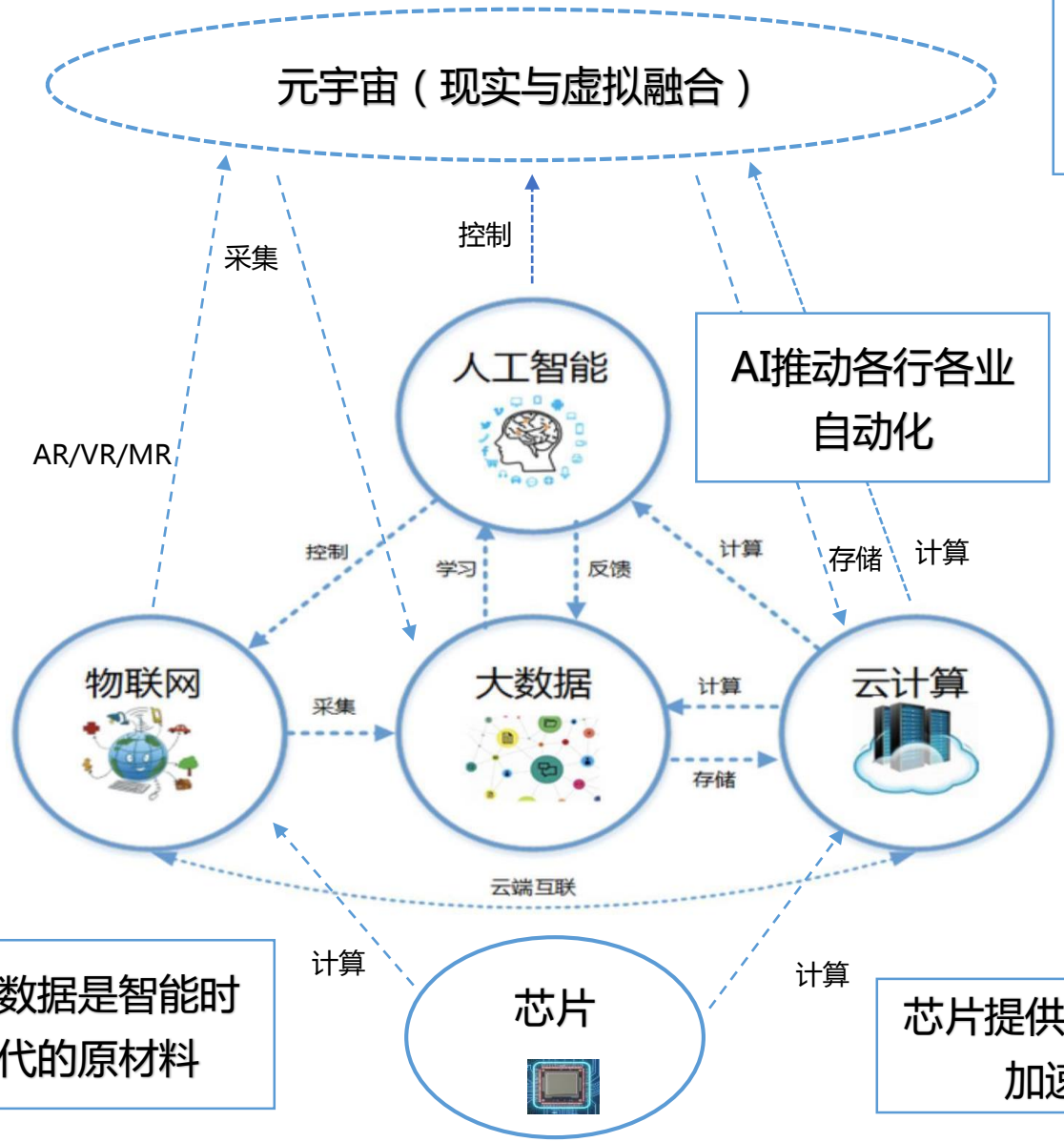
核心产品：工业互联网 + 元宇宙（未来）

核心技术： 云计算/AI/芯片/物联网/自动驾驶/5G/6G



未来核心技术

元宇宙是区块链、人工智能、5G、VR、AR、物联网、大数据、云计算、边缘计算等前沿数字技术的集成应用，元宇宙跨越了现实和虚拟现实之间的物理和数字鸿沟。



AI推动各行各业自动化

物联网是现实世界形态

云计算提供算力网络（计算,存储,网络）

大数据是智能时代的原材料

芯片提供算力和加速