# 真实面试

WXG 公众号小程序组 后台一面 2024.11.13

## 了解项目

提及：

- tencentOS
- open telementery 对标 Prometheus

## 开源相关

- PR的流程
- merge 和 rebase 的区别，画图

## 操作系统

- 软中断有哪些类型？
  在操作系统中，软中断的类型主要取决于其用途和触发来源。一般来说，软中断用于处理不需要立即响应的任务，常见的软中断类型包括以下几种：

1. 任务软中断（Tasklet）
   任务软中断是一种较轻量级的软中断机制，用于在某些任务完成后延迟执行某些操作。任务软中断通常在系统有空闲时批量执行。
   - 特点：任务软中断一般由内核模块或驱动程序发起，专注于短小、简单的任务。这些任务不需要立即响应，可以在中断上下文中延后处理。
   - 应用场景：在网络驱动程序中，数据包接收到内核缓冲区后，可以通过任务软中断来执行具体的数据处理操作，而不阻塞硬件的正常工作。
2. 工作队列（Work Queue）
   工作队列是一种基于线程的软中断机制，用于调度相对复杂、耗时的任务。它允许将任务放到系统线程中异步执行，避免阻塞内核。
   - 特点：工作队列是比任务软中断更加灵活的延迟处理机制，因为它可以在内核线程上下文中执行，因此不受中断上下文的约束，可以进行阻塞操作（如内存分配、文件操作）。
   - 应用场景：在需要较长时间的任务或对资源有依赖的操作（如设备驱动中的延时操作）时，工作队列能够提供较好的性能和可操作性。例如，在文件系统中，文件缓存的刷新任务可以通过工作队列来异步执行。
3. 定时软中断（Timer Interrupt）
   定时软中断是一种基于时间触发的软中断，用于周期性或延迟的任务执行。
   - 特点：定时软中断基于系统定时器，由内核定时触发，支持精确的时间间隔控制。它经常用于需要定期执行的任务。
   - 应用场景：例如在操作系统内核中，用于系统心跳检测、周期性检查设备状态或定期回收系统资源等。网络协议栈也常用定时软中断来处理超时重传等任务。
4. 网络软中断（Network SoftIRQ）
   网络软中断是网络协议栈中特定的软中断类型，专门用于处理网络数据包的接收和发送操作。网络软中断一般在硬中断完成后触发。
   - 特点：网络软中断主要用于数据包的解析、协议处理等非实时性的操作。通过软中断来处理，可以避免数据包传输过程中的频繁硬中断，提升网络传输效率。

- 应用场景：在网络通信中，底层硬件（如网卡）收到数据包时，首先通过硬中断通知操作系统，然后由网络软中断进行更深入的数据包解析和协议处理。

5. 块设备软中断（Block I/O SoftIRQ）
   块设备软中断是用于块设备（如硬盘）数据传输的软中断类型，用于在硬件完成数据传输后执行数据的处理。
   - 特点：块设备软中断将数据的具体处理从硬中断中解耦，避免了在数据传输完成后立即处理的延迟影响。它可以在稍后合适的时间再进行数据传输的具体处理操作。
   - 应用场景：块设备如磁盘、SSD等存储设备的读写请求通常通过块设备软中断来处理，以提高I/O处理效率。例如，当数据块加载完毕后，系统会通过软中断去执行缓冲区的读写操作。
6. 自定义软中断
   自定义软中断是操作系统或应用程序根据需求设置的专用中断类型，用于完成某些特殊的任务。自定义软中断通常用于处理一些特定场景的优化需求。
   - 特点：自定义软中断适合特定应用场景，具有灵活性，通常由开发人员设计并实现。自定义软中断的调度通常也会影响系统性能，因此在设计上要避免频繁触发或过多依赖自定义软中断。
   - 应用场景：在某些高性能计算或大规模数据处理系统中，可能会设计专用的自定义软中断，用于优化特定的计算过程或数据传输过程。例如，在图像处理应用中，可以设计一个特定的软中断去实现高频的图像帧缓存刷新。

总结
软中断的类型和机制设计旨在通过延迟和异步执行来平衡系统的响应速度和计算效率。每种软中断类型在特定场景下都有其优势和适用性：

| 软中断类型 | 特点 | 典型应用场景 |
| --- | --- | --- |
| 任务软中断 | 简单、轻量 | 网络数据的简单处理、轻量任务延迟 |
| 工作队列 | 基于线程、可阻塞 | 文件系统缓存刷新、延时操作 |
| 定时软中断 | 定时触发、支持周期性操作 | 系统心跳检测、定期资源回收 |
| 网络软中断 | 专用于网络协议栈，提升网络传输效率 | 数据包解析、协议栈处理 |
| 块设备软中断 | 针对块设备I/O操作 | 磁盘读写缓冲处理 |
| 自定义软中断 | 灵活、自定义操作 | 特定系统优化（如高性能计算的专用任务） |

通过合理使用这些软中断，操作系统可以高效地管理不同类型的任务，提高系统的响应速度和资源利用率。

- 硬中断有哪些类型?

硬中断的类型可以根据不同的触发方式、功能和优先级进行分类。硬中断通常用于处理外部设备（如键盘、网络设备、存储设备等）请求的紧急事件，操作系统对不同硬中断的管理有特定的机制。以下是常见的几种硬中断类型：

1. 外部设备中断（External Device Interrupt）
   外部设备中断是由外部硬件设备（如键盘、鼠标、网络卡、USB设备等）触发的，用于通知CPU需要处理的外部事件。
   - 特点：这些中断通常与用户交互有关，是由物理设备发出的信号。设备通过中断控制器向CPU发送请求，CPU接收到信号后会立即中断当前任务并响应。
   - 应用场景：例如，当用户按下键盘时，键盘控制器会发送中断信号，使CPU立即捕获按键事件，然后由中断处理程序负责读取按键数据。
2. 定时器中断（Timer Interrupt）
   定时器中断是由系统内部的定时器芯片（如PIT或HPET）定时触发的，用于支持操作系统的时间片轮转和系统时间管理。
   - 特点：定时器中断是周期性触发的硬中断，通常具有较高优先级。操作系统可以根据定时器中断来实现任务的时间片调度（即在多任务系统中切换任务）以及定时操作。

- 应用场景：例如，操作系统通过定时器中断实现多任务管理。定时器中断每隔一段时间触发一次，使得操作系统可以在任务之间切换，从而实现“多任务”的效果。

3. I/O中断（I/O Interrupt）
   I/O中断是由输入/输出设备（如磁盘、网络适配器等）触发的，用于通知CPU某个I/O操作已经完成或需要处理。
   - 特点：I/O中断用于管理与存储设备、网络设备、输入设备的交互过程。它能够有效减少CPU等待时间，因为在I/O操作完成后才会触发中断，使CPU能够及时处理数据。
   - 应用场景：比如在磁盘读写过程中，CPU发起数据读取请求后，可以执行其他任务。当磁盘读取完数据后，发送I/O中断，通知CPU数据准备好了，这时CPU再去处理数据，减少了等待时间。
4. 系统管理中断（System Management Interrupt, SMI）
   系统管理中断是一种特殊的硬中断，由硬件（通常是芯片组）触发，用于处理系统级管理任务，如电源管理、硬件监控等。
   - 特点：系统管理中断会让CPU进入一种特殊的“系统管理模式”（SMM），在这种模式下，系统会暂停当前的操作，转而执行系统管理任务。SMI通常具有较高优先级，且处理过程对操作系统透明，OS无法直接干预。
   - 应用场景：SMI在电源管理中比较常见，例如在电池电量低时触发SMI来切换到低功耗模式，或者在系统温度过高时进行风扇控制等。
5. 异常中断（Exception Interrupt）
   异常中断是一种特殊的硬中断，通常在CPU执行程序过程中遇到异常情况时触发。例如，除零错误、非法指令、页面错误等。
   - 特点：异常中断是由CPU内部生成的，具有较高的优先级，属于紧急中断类型，操作系统需要立即处理这些异常情况以确保系统稳定性。
   - 应用场景：常见的应用是处理非法操作，比如在程序试图访问无效内存地址时，会触发页面错误中断，操作系统则会检查地址是否合法，并做出相应的处理。
6. 中断请求级别（Interrupt Request, IRQ）中断
   在一些系统中，中断控制器会为每个中断源分配一个IRQ编号（Interrupt Request Level），按优先级管理不同的硬件中断请求。
   - 特点：IRQ编号决定了中断的优先级顺序。低编号的IRQ通常有更高的优先级，某些关键设备（如系统定时器）通常被分配较高优先级的IRQ。
   - 应用场景：例如，网络接口通常分配到较高优先级的IRQ以保证数据传输的实时性，而键盘中断则会被分配相对较低的IRQ优先级。
7. 嵌套中断（Nested Interrupt）
   嵌套中断是指在一个中断处理程序未完成时，CPU又接收到了更高优先级的中断请求，因此打断当前处理中断，转去处理新的中断。
   - 特点：嵌套中断允许系统在处理较低优先级中断时打断以处理更高优先级的中断，从而提高系统的实时性。
   - 应用场景：比如在处理网络数据包的过程中，系统接收到定时器中断（优先级更高），会暂停当前网络数据的处理中断去响应定时器，然后再继续原有的中断任务。

硬中断类型总结
不同类型的硬中断在操作系统和硬件层面的作用和处理机制各不相同。通过合理设计中断类型和优先级，操作系统能够有效地分配资源、提高响应速度和管理外部设备。

| 硬中断类型 | 触发来源 | 特点和优先级 | 典型应用场景 |
| --- | --- | --- | --- |
| 外部设备中断 | 键盘、鼠标、USB等 | 实时性高，通常响应用户交互 | 键盘按键检测、鼠标点击事件 |
| 定时器中断 | 系统定时器 | 周期性触发，<br>支持多任务系统时间片轮转 | 多任务切换、系统时间更新 |
| I/O中断 | 磁盘、网络适配器 | 数据就绪或完成后触发，<br>减少CPU等待时间 | 磁盘读写、网络数据接收 |

| 硬中断类型 | 触发来源 | 特点和优先级 | 典型应用场景 |
| --- | --- | --- | --- |
| 系统管理中断（SMI） | 芯片组、硬件事件 | 具有很高优先级，用于系统管理，OS无法干预 | 电源管理、硬件状态监控 |
| 异常中断 | CPU内部异常 | 由异常操作引发，紧急中断，需要立即处理 | 除零错误、非法指令、页面错误 |
| 中断请求级别（IRQ）中断 | 中断控制器管理的硬件中断 | 不同IRQ对应不同设备的优先级 | 网络、键盘、鼠标等中断优先级管理 |
| 嵌套中断 | 由多个硬中断请求引起 | 支持高优先级中断打断低优先级中断，增强实时性 | 定时器中断打断低优先级设备请求 |

通过这些硬中断的分类和管理，操作系统能够在资源有限的情况下实现对各种任务的高效响应。

- **硬中断和软中断的区别?**
  软中断和硬中断是操作系统中非常重要的概念，主要用来处理外部事件并提高系统响应效率。它们都属于“中断”机制，但作用场景和实现方式不同。下面我们来详细解释这两者的区别：

1. 硬中断（Hardware Interrupt）
   硬中断是由硬件设备（如键盘、鼠标、网络适配器等）直接向CPU发出的请求信号，用于通知处理器需要立即处理的事件。

类比：硬中断就像是火警警报声响起，这意味着有紧急事件（火灾）发生，必须立即打断当前工作去处理。比如一个硬盘驱动器完成了数据传输，会向CPU发送硬中断信号，告知“传输完成”，这样CPU就可以马上处理数据了。

- 工作流程：
  i. **触发中断**：当设备完成了某项操作（比如网络适配器接收到数据包）时，它会向CPU发送一个硬中断信号。
  ii. **响应中断**：CPU收到硬中断信号后，会暂停当前执行的任务，转去执行对应的中断处理程序（Interrupt Service Routine，ISR）。
  iii. **处理中断**：在ISR中，操作系统会处理该设备的请求，比如读取数据包或写入磁盘。
  iv. **恢复**：中断处理完成后，CPU会恢复到原来的任务，继续之前的操作。
- 特点：
  - **实时性强**：硬中断处理通常是及时的，否则可能会导致数据丢失或设备故障。
  - **优先级高**：硬中断一般比软中断优先级更高，且系统必须快速响应。

2. 软中断（Software Interrupt）
   软中断是由软件代码触发的中断，通常是操作系统内核或者应用程序自己请求某些操作（如网络数据的协议处理、定时任务等）时触发的。

类比：软中断就像办公室内部的一个“代办任务”提醒，比如收到通知“每小时要收集员工的工作情况”，这个任务比较重要，但不需要立即打断其他工作。可以等到手头上的工作有空再处理。软中断可以理解为一种优化手段，将不需要立即处理的任务推迟到稍后执行。

- 工作流程：
  i. **触发中断**：在操作系统或应用中，一些任务完成后（如文件写入成功或网络数据接收），会触发一个软中断请求。
  ii. **等待调度**：软中断请求进入待处理的队列，但不会立即打断CPU当前任务，而是等待系统有空闲时间时再处理。
  iii. **处理中断**：当CPU空闲时，操作系统会选择合适的时间点去处理软中断请求（可能是批量处理）。
  iv. **恢复**：处理完软中断后，系统继续执行其他任务。
- 特点：
  - **响应延迟**：因为软中断不需要立即响应，所以处理上可能会有延迟。

- **优先级低**：软中断优先级比硬中断低，通常在系统资源空闲时才处理。
- **效率高**：软中断能够减少系统开销，因为它避免了频繁打断任务，提高了系统的整体效率。

3. 软中断和硬中断的区别

| 方面 | 硬中断 | 软中断 |
| --- | --- | --- |
| 触发方式 | 由硬件设备（如键盘、网络适配器）触发 | 由软件或操作系统内部触发 |
| 优先级 | 较高，必须立即响应 | 较低，通常在系统空闲时处理 |
| 实时性 | 强，必须快速处理，否则可能导致数据丢失 | 弱，可以延迟到合适的时间处理 |
| 适用场景 | 设备请求CPU处理数据或状态 | 协议栈处理、延时任务、批量处理等非紧急任务 |

- 使用场景举例
  - **硬中断的例子**：在键盘按键按下时，系统会触发硬中断请求，让CPU中断当前任务并读取按键数据，以便迅速处理用户输入。
  - **软中断的例子**：在网络数据传输中，底层硬件会首先触发硬中断让系统感知到有新数据到达，但数据的具体处理（如解析和响应）则可能通过软中断来完成，从而减轻CPU负担。

通过软中断和硬中断的结合，操作系统能更高效、灵活地处理各种任务：硬中断负责及时响应设备请求，软中断则负责在空闲时处理批量任务，实现了性能和实时性的平衡。

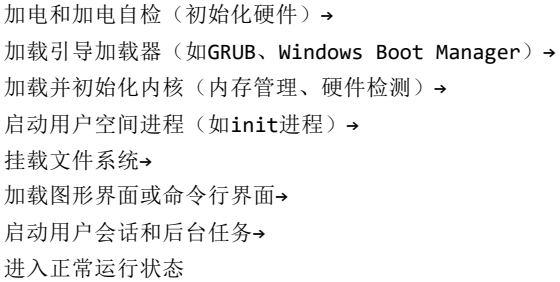
- **描述一个操作系统启动的全流程**
  操作系统启动的全过程涉及多个阶段，从系统加电到加载操作系统内核，再到最终进入用户界面或命令行。在这个过程中，硬件和软件之间紧密协作，使得系统从物理层面的设备状态逐渐进入操作系统的功能性状态。以下是操作系统启动的详细步骤：

1. 加电与初始化
   - 加电自检（Power-On Self Test, POST）：当计算机加电启动时，**BIOS**（或UEFI）会首先运行一套预编程的加电自检程序。POST的作用是检测硬件的基本完整性，比如CPU、内存、显卡、硬盘等是否存在问题。如果检测到硬件故障，POST会通过蜂鸣声或错误信息提醒用户。
   - 初始化硬件：POST结束后，BIOS或UEFI开始初始化硬件设备，确保关键设备（如CPU和内存）处于可用状态。这一步中还会识别外接设备（如键盘、鼠标、USB设备等）。
2. 加载启动引导程序
   - 引导顺序确定：在BIOS或UEFI中设定的引导顺序将决定从哪个存储设备启动操作系统，比如从硬盘、光盘、U盘或网络引导。
   - MBR或EFI分区引导：
     - 在传统的BIOS系统中，操作系统通过主引导记录（MBR, Master Boot Record）引导。MBR是硬盘上的第一个扇区（通常512字节），其中包含一个引导程序和分区表。MBR的引导程序会将控制权交给活动分区上的第二阶段引导程序。
     - 在现代的UEFI系统中，启动过程通过EFI分区（EFI System Partition, ESP）引导。UEFI会从EFI分区中加载启动管理器（Boot Manager），并从中选择要加载的操作系统文件。
3. 加载引导加载器（Bootloader）
   - 第一阶段加载：在BIOS系统中，MBR加载的第一个引导程序（通常叫做引导加载器，如GRUB或LILO）进入第一阶段，初始化一些关键的加载功能，为更复杂的引导过程做准备。
   - 第二阶段加载：引导加载器的第二阶段加载操作系统内核文件。对于复杂的引导程序（如GRUB），它会显示一个启动菜单，允许用户选择不同的内核或操作系统（比如双系统时可以选择Windows或Linux）。
   - 内核加载：引导加载器将操作系统内核从磁盘加载到内存中，将控制权交给内核，从而开始操作系统的正式启动。
4. 内核初始化

- 硬件检测和初始化：操作系统内核接管系统后，首先会检测系统中的硬件并初始化资源。内核会加载并初始化硬件驱动程序，识别和配置CPU、多核处理器、内存、网络接口、存储设备等。
- 内存管理初始化：内核会初始化内存管理模块，建立内存页表，以支持虚拟内存功能。虚拟内存能够为每个进程提供一个独立的、受保护的地址空间，使不同程序的内存互不干扰。
- 中断和异常处理初始化：内核会建立中断向量表，以便能够响应来自硬件的中断（如键盘输入、网络数据到达）。同时，内核会设置异常处理机制，确保在出现异常情况（如非法指令）时能够进行适当处理。
- 进程管理初始化：操作系统会创建第一个内核进程或系统进程（如Linux中的init进程或systemd进程），用于启动后续用户空间程序。

5. 启动用户空间进程
   - 用户空间初始化：内核初始化完成后，系统进入用户空间，加载并执行用户级的初始进程。在Linux系统中，这通常是init或systemd进程，在Windows中是Session Manager（smss.exe）。
   - 加载系统服务和守护进程：初始进程会启动系统服务和守护进程（如网络管理、日志记录、文件系统监视等），这些进程在后台运行，为操作系统和应用程序提供支持。
   - 挂载文件系统：系统会挂载根文件系统（通常是主硬盘的分区），并按需挂载其他文件系统（如USB设备、网络文件系统等），确保所有的存储资源都可供访问。
6. 加载图形界面或命令行界面
   - 进入图形界面（GUI）或命令行界面（CLI）：根据系统配置和用户偏好，操作系统会加载图形界面管理器（如X Window、Wayland、Windows Explorer等），或者进入命令行界面。在Linux系统中，图形界面通常由Xorg或Wayland启动，在Windows系统中由explorer.exe管理桌面和任务栏。
   - 登录界面：系统会启动一个登录界面，等待用户输入用户名和密码，验证成功后进入用户桌面或命令行环境。
7. 启动用户会话和应用程序
   - 用户配置加载：登录成功后，操作系统会加载用户配置文件（如桌面设置、快捷方式、启动程序等）。Linux系统通常通过/.profile、/.bashrc等文件加载配置，Windows则通过注册表和用户配置文件加载设置。
   - 启动会话管理器：会话管理器会负责管理用户的应用程序启动、退出和状态保存等。例如，用户可以在系统启动后打开浏览器、文件管理器、文本编辑器等应用。
   - 后台任务启动：根据用户配置和系统设置，一些用户级的后台任务或守护进程（如云同步、消息通知等）会自动启动，完成用户环境的全面初始化。
8. 进入正常运行状态
   - 系统进入就绪状态：操作系统完成所有初始化后，进入正常运行状态。此时用户可以自由使用系统，执行应用程序、访问文件和进行其他操作。
   - 任务管理和资源分配：系统通过任务管理器和资源管理模块来分配CPU、内存、I/O等资源，确保应用程序的正常运行。任务调度器会根据任务的优先级、资源占用情况等分配时间片，以实现多任务运行。

- 总结
  操作系统启动的全过程通过逐步初始化硬件、加载内核和用户进程，为用户提供一个稳定、可操作的计算环境。这个过程如下图所示：




通过这些阶段，操作系统逐步建立了从硬件到用户空间的完整工作环境，实现了计算机从物理设备到操作系统的转变，使用户能够与系统交互。

- 实模式和保护模式
  实模式（Real Mode）和保护模式（Protected Mode）是x86架构中CPU的两种工作模式，它们定义了处理器的内存寻址方式、指令执行权限以及系统的整体安全性和稳定性。下面我们来详细介绍这两种模式的特点、工作原理和应用场景。

1. 实模式（Real Mode）
   实模式是最早期的x86处理器工作模式，最早应用于16位的Intel 8086处理器。实模式的设计主要考虑简化内存管理和程序执行，适用于早期的操作系统，如MS-DOS。

- 特点
  - **内存寻址限制**：实模式只能使用16位寄存器进行内存寻址，因此只能访问1MB的物理内存空间。即，CPU可以访问的地址范围从0x00000到0xFFFFF。
  - **分段内存模型**：实模式采用分段寻址，将内存分成多个“段”进行管理。地址由“段基址（Segment）+偏移量（Offset）”的组合构成。
  - **缺乏内存保护**：实模式没有内存保护机制，所有程序共享同一片物理内存空间，任何程序都可以自由访问整个1MB内存。
  - **单任务模式**：实模式不支持多任务，也无法进行任务的优先级调度。
- 工作机制
  在实模式中，CPU使用16位寄存器来指定段地址和偏移地址，每个段最多64KB（2^16 字节）大小。内存地址的计算公式如下：

```
物理地址 = 段地址 × 16 + 偏移地址
```

例如，段地址为`0x1234`，偏移量为`0x5678`时，实际物理地址为：

```
物理地址 = 0x1234 × 16 + 0x5678 = 0x12340 + 0x5678 = 0x179B8
```

- 典型应用场景
  - **早期操作系统**：如MS-DOS、CP/M等，它们在实模式下工作，操作系统和应用程序都运行在无保护的内存中。
  - **启动引导**：现代操作系统的启动引导阶段仍然使用实模式。BIOS加电自检（POST）和启动加载器（Bootloader）等程序在实模式下启动，然后再切换到保护模式。

2. 保护模式（Protected Mode）
   保护模式是在80286及其后续处理器中引入的模式，目的是克服实模式的缺点，支持更强大的内存管理、权限管理和多任务处理机制。保护模式是现代操作系统（如Windows、Linux等）运行的主要模式。

- 特点
  - **扩展内存寻址**：保护模式支持32位甚至64位地址寄存器，内存寻址能力大幅提升。32位模式下可以访问4GB内存，64位模式下支持理论上高达16EB的地址空间。
  - **内存保护**：保护模式引入了内存保护机制，允许操作系统为不同进程分配独立的内存空间。进程之间的内存相互隔离，防止进程访问或修改其他进程的内存区域。
  - **分段和分页机制**：保护模式支持分段和分页两种内存管理方式。分段用于逻辑内存管理，而分页可以提供更细粒度的内存保护和虚拟内存支持。
  - **多任务支持**：保护模式支持多任务操作系统，允许CPU在多个进程之间切换，并为不同进程分配不同的权限。
  - **权限控制**：保护模式定义了四个特权级（Privilege Levels），即Ring 0到Ring 3。Ring 0具有最高权限，用于操作系统内核，Ring 3具有最低权限，用于用户应用程序。这样可以防止用户程序直接操作系统核心资源，增强系统安全性。
- 工作机制
  保护模式的内存管理基于分段机制和分页机制。
  - **分段机制**：在保护模式中，每个段可以有自己的访问权限和限制。段基地址和段界限被存储在段描述符表（GDT/LDT）中。

- **分页机制**：分页将内存划分为固定大小的页（通常为4KB），并通过页表映射实现物理内存和虚拟内存的分离。分页机制提供了精确的内存保护和内存管理，允许进程访问虚拟地址空间，而操作系统负责将虚拟地址映射到物理地址。
- 典型应用场景
  - **现代操作系统**：大部分现代操作系统（如Windows、Linux、macOS）都在保护模式下运行，以充分利用保护模式的内存保护、多任务和权限控制功能。
  - **虚拟内存和多任务处理**：保护模式下的分页机制支持虚拟内存，允许操作系统将不常用的数据交换到硬盘中，同时保证不同进程的内存隔离性。

3. 实模式与保护模式的对比

| 特性 | 实模式（Real Mode） | 保护模式（Protected Mode） |
| --- | --- | --- |
| 内存寻址能力 | 1MB | 4GB（32位），甚至16EB（64位） |
| 内存保护 | 无保护，任何程序都能访问任意内存 | 支持内存保护和进程隔离 |
| 内存管理方式 | 分段 | 分段和分页 |
| 多任务支持 | 不支持 | 支持 |
| 权限控制 | 无特权等级 | 支持特权等级（Ring 0 - Ring 3） |
| 应用场景 | 早期操作系统、启动引导阶段 | 现代操作系统运行模式（如Windows、Linux） |

4. 模式切换
   系统启动时通常从实模式进入，完成硬件自检和初始化后，操作系统会将处理器切换到保护模式以运行现代操作系统的内核。切换过程如下：

- **启用保护模式**：操作系统通过设置控制寄存器CR0的第0位来启用保护模式。
- **加载段描述符表（GDT）**：系统会加载GDT，以定义段的基址、界限和权限等信息。
- **启用分页（可选）**：在32位和64位操作系统中，分页机制通常也会被激活，通过设置CR3寄存器来加载页表。

一旦进入保护模式，系统可以利用现代内存管理和权限控制功能，并正式进入操作系统环境。

- 总结
  - 实模式是16位模式，主要用于启动和简单的低级操作系统（如DOS），没有内存保护和权限控制。
  - 保护模式是32位或64位模式，支持更大的内存空间、内存保护、多任务和权限控制，是现代操作系统的主要运行模式。

  通过从实模式切换到保护模式，CPU可以充分发挥现代操作系统的优势，实现高效、稳定和安全的系统运行。
- 内核态和用户态的资源隔离和权限控制是怎么做的?
  答了用户态通过中断的方式陷入内核执行内核的代码
  在操作系统中，内核态和用户态是两种不同的执行模式，主要用于实现资源隔离和权限控制。这种机制确保了操作系统的安全性和稳定性，避免了用户程序对系统资源的不当访问。下面我们详细说明内核态和用户态的资源隔离和权限控制是如何实现的。

1. 内核态与用户态的基本概念

- **内核态（Kernel Mode）**：内核态是系统运行权限最高的模式，操作系统的核心代码和关键资源（如内存、CPU、I/O设备等）都在内核态中运行。内核态具有对所有硬件资源的完全控制权限。
- **用户态（User Mode）**：用户态是普通应用程序运行的模式，权限有限，无法直接访问硬件资源。用户态的代码只能通过系统调用请求内核执行敏感操作。

这种模式的划分保证了系统的核心资源受到保护，用户态程序只能通过内核提供的接口间接访问硬件资源。

2. 资源隔离和权限控制的实现机制
   内核态和用户态的资源隔离和权限控制主要通过以下几个机制来实现：

- CPU特权级别（Privilege Levels）
  在x86架构中，CPU支持4个特权级别（Ring 0到Ring 3），特权级别越低权限越高。一般来说：
  - **Ring 0**：用于操作系统内核代码运行，即内核态，具有最高权限。
  - **Ring 3**：用于用户程序运行，即用户态，具有最低权限。

通常，操作系统只使用Ring 0和Ring 3来简化管理。CPU的不同权限级别通过硬件强制限制用户态程序的执行权限，防止其直接访问敏感的系统资源。

- 内存管理单元（MMU）和虚拟内存
  内存管理单元（MMU）是硬件中的一个模块，用于实现内存保护和虚拟内存功能。MMU将虚拟地址空间映射到物理内存，操作系统利用MMU实现内存隔离，使得用户态和内核态各有独立的地址空间。
- **虚拟地址空间隔离**：每个用户态程序都有独立的虚拟地址空间，彼此隔离，不能直接访问其他进程的内存空间。内核的虚拟地址空间对用户态程序不可见，即使用户程序尝试访问内核地址空间，MMU也会拒绝，导致“段错误”（Segmentation Fault）或“页面错误”（Page Fault）。
- **页表和访问权限控制**：操作系统通过页表控制每个内存页的访问权限（如可读、可写、可执行）。MMU通过访问权限检查限制用户态程序的内存访问范围，保证用户程序不能随意读取或修改内核数据。
- 系统调用接口（System Call Interface）
  用户态程序只能通过系统调用访问内核资源，系统调用是内核为用户程序提供的受控接口。系统调用机制主要通过以下方式实现隔离和控制：
  - **陷入指令**：系统调用通过特定的陷入指令（如x86架构中的int指令或syscall指令）从用户态切换到内核态。陷入指令会触发一次中断，进入内核模式后由内核的系统调用处理程序接管控制。
  - **系统调用号**：用户态程序在发起系统调用时指定一个系统调用号（用于标识具体操作），内核在查找到对应的系统调用处理程序后执行该功能。系统调用号确保了用户态程序只能执行内核提供的特定功能，而无法直接控制内核代码的执行流程。
  - **返回用户态**：系统调用执行完毕后，内核会将控制权返回给用户态程序。用户态程序始终不能直接跳转到内核态代码，必须通过系统调用实现受控访问。
- 硬件保护机制
  硬件提供的内存保护机制，如分页（Paging）和分段（Segmentation），进一步增强了内核态和用户态的隔离：
  - **分页机制**：在分页机制下，内核和用户程序的内存页面被标记不同的访问权限（如只读、只写、只执行等）。硬件会检查访问权限，阻止用户态程序直接访问内核页。
  - **内存分段机制**：在某些架构中，内核和用户程序的地址空间通过分段进行管理。用户程序的分段设置限制了其可访问的地址范围，防止其跨越边界访问内核段。

3. 资源隔离和权限控制的实现细节
   以下是资源隔离和权限控制的具体实现细节：
   - **内核数据结构隔离**：关键数据结构（如进程控制块、文件描述符表、内存管理表等）仅在内核态可见。用户态无法直接访问这些数据，必须通过系统调用获取有限的资源信息。
   - **进程隔离**：不同进程之间的内存空间通过页表隔离，防止进程之间的任意访问。即使一个进程崩溃或恶意篡改数据，也不会影响其他进程的安全。
   - **设备访问控制**：用户态程序不能直接操作硬件设备，必须通过系统调用向内核请求设备访问权限。内核会检查设备使用权限，防止未经授权的程序访问关键设备。
   - **中断处理隔离**：中断和异常处理程序只能在内核态下执行。用户态程序不能直接控制中断或异常处理逻辑，保证了系统对硬件事件的完全控制。
4. 内核态与用户态切换的代价与优化
   由于内核态和用户态的隔离是通过硬件和系统调用实现的，切换会有一定的性能代价。每次切换需要保存和恢复CPU寄存

器、切换内存页表、更新特权级别等。

现代操作系统通过以下优化手段减少切换的开销：

- **减少系统调用次数**：通过批量处理请求或提供高级API减少系统调用次数，从而减少内核态和用户态的切换。
- **用户态驱动程序**：某些非关键性设备驱动程序可以在用户态运行，减少内核态调用的需求。
- **页表缓存（TLB）优化**：通过硬件中的TLB（Translation Lookaside Buffer）缓存页表映射，加快内存访问速度，降低切换的开销。
- 总结
  内核态和用户态的隔离和权限控制是通过硬件和操作系统共同实现的，具体机制包括：
- **CPU特权级别**：将内核态和用户态运行隔离在不同的特权级别（Ring 0和Ring 3）。
- **内存管理单元（MMU）和虚拟内存**：实现内存隔离，确保用户态程序无法直接访问内核态内存。
- **系统调用机制**：提供受控的内核资源访问接口，防止用户态直接访问硬件资源。
- **硬件保护机制**：通过分页和分段保护内核地址空间和资源。

这些机制共同确保了操作系统的资源安全与稳定，防止用户程序对内核和其他进程的任意干扰，实现了计算机系统的多任务和安全性。

- **有哪些 posix 系统调用**
- **系统调用存在哪些开销**?
  系统调用虽然提供了安全性和资源隔离，但也带来了性能开销。以下是系统调用的主要开销来源和详细分析：

1. 用户态与内核态切换开销
   系统调用涉及用户态和内核态的切换，这一切换过程带来了以下开销：

- **特权级转换**：用户态和内核态在CPU的特权级别上不同（通常用户态是Ring 3，内核态是Ring 0），因此每次系统调用需要调整CPU的特权级。
- **上下文保存和恢复**：切换时需要保存用户态的寄存器状态，加载内核态的上下文；系统调用返回时，反之亦然。这一过程需要操作一系列寄存器，带来时间开销。
- **内存管理切换**：为了保证内核的安全性，用户态进程不能直接访问内核内存。因此系统调用时还可能涉及页表的切换或更新，例如从用户页表切换到内核页表，或者更新内存访问权限。

2. 地址空间切换和缓存失效
   系统调用涉及到用户态和内核态地址空间的切换，这可能导致CPU缓存和TLB（Translation Lookaside Buffer）失效：

- **缓存（Cache）失效**：在从用户态切换到内核态时，CPU可能会因为访问不同的地址空间而导致缓存失效，必须重新加载数据。这种缓存失效增加了内存访问的延迟。
- **TLB失效**：TLB缓存着虚拟地址到物理地址的映射。在切换到内核地址空间时，原本在TLB中缓存的用户态地址映射可能被清空，导致后续的内存访问需要重新加载TLB。

3. 参数传递和数据拷贝
   系统调用涉及用户态和内核态之间的数据传递，这通常需要数据的安全性检查和内存拷贝：

- **参数检查**：操作系统需要检查传递给系统调用的参数是否合法，以防止不安全操作。这包括检查指针是否指向有效的用户地址、检查字符串的长度等。这些安全性检查增加了系统调用的执行时间。
- **数据拷贝**：由于用户态和内核态的内存空间相互隔离，如果系统调用涉及到数据传递（例如读写文件、网络通信等），操作系统需要将数据从用户态内存拷贝到内核态，或反向拷贝。内存拷贝是一个相对耗时的操作，尤其是当数据量较大时，拷贝开销显著增加。

4. 系统调用上下文切换
   在多任务系统中，系统调用可能触发上下文切换，尤其是涉及I/O操作时：

- **阻塞等待**：某些系统调用（如文件读写、网络请求）涉及I/O操作。这些操作可能会导致进程阻塞，以等待数据到达或写入完成。此时操作系统会将CPU分配给其他任务，完成I/O后再恢复该进程的执行。
- **上下文切换**：当进程因系统调用阻塞时，系统会进行上下文切换，将CPU让给其他进程。上下文切换需要保存当前进程的状态、恢复新进程的状态，这一过程也会带来一定的开销。

5. 系统调用表查找
   系统调用是通过系统调用号来识别的，操作系统通过系统调用号在系统调用表中找到对应的处理函数。这一过程也存在一些开销：

- **系统调用号的查找**：系统调用号是操作系统识别特定调用的标识。操作系统会在系统调用表（系统调用向量表）中查找与系统调用号对应的处理函数。虽然查找速度较快，但仍存在一定的开销。
- **函数跳转开销**：查找到具体的系统调用处理函数后，操作系统会跳转到该函数执行。尽管跳转开销较小，但多次系统调用会累积一定的时间成本。

6. 处理器流水线的中断
   系统调用通过陷入指令（如x86架构中的int 0x80或syscall指令）从用户态切换到内核态。这一过程会中断CPU的流水线，导致流水线清空和重新填充的开销：

- **流水线中断**：处理器在用户态执行指令的过程中遇到系统调用陷入指令时，会停止流水线，清空未完成的指令。这一操作会导致一定的延迟。
- **重新填充流水线**：进入内核态后，CPU重新填充内核态的指令集。系统调用返回用户态时也会再次清空流水线，并填充用户态指令。

7. 安全性检查与权限管理
   在系统调用中，操作系统需要对用户程序的请求进行安全检查和权限管理，以防止恶意程序对系统的破坏：

- **权限验证**：操作系统会检查当前进程的权限，确保其可以执行特定的系统调用操作。比如，文件访问权限检查会验证当前进程是否具备对文件的读取或写入权限。
- **地址验证**：系统调用需要检查用户传递的指针或内存地址，以确保地址在用户空间范围内，防止用户态程序意外或恶意访问内核空间。这些安全性检查都会增加系统调用的执行时间。

8. 锁和同步机制
   多个进程或线程可能会并发调用同一系统资源（如文件系统或网络接口），操作系统需要确保这些资源的并发访问是安全的。这种同步操作也带来了开销：

- **锁操作**：为了保证数据的一致性，系统在系统调用的关键代码部分会添加锁机制。获取锁、释放锁、检测锁状态等操作都需要额外的时间。
- **等待队列**：当多个进程竞争某个资源时，操作系统可能会将一些进程加入等待队列，并在资源可用时唤醒它们。管理这些等待队列也是一种开销。
- 总结
  系统调用的开销来自多个方面，主要包括：

| 开销类型 | 说明 |
| --- | --- |
| 用户态与内核态切换开销 | CPU特权级转换、上下文保存与恢复、内存管理切换。 |
| 地址空间切换与缓存失效 | 缓存和TLB失效，导致内存访问速度下降。 |
| 参数检查和数据拷贝 | 用户态与内核态之间的数据传递和合法性检查。 |
| 系统调用上下文切换 | 阻塞调用可能引起上下文切换，增加额外的开销。 |

| 开销类型 | 说明 |
| --- | --- |
| 系统调用表查找和跳转 | 系统调用号的查找和相应函数的跳转。 |
| 处理器流水线中断 | 陷入指令会清空流水线，增加延迟。 |
| 安全性检查与权限管理 | 验证用户权限和地址范围，以保证系统安全。 |
| 锁和同步机制 | 多进程并发访问时，使用锁或等待队列进行同步控制。 |

这些开销会影响系统调用的响应速度，尤其在频繁系统调用的情况下，会对程序的整体性能产生较大影响。为此，现代操作系统通常采取减少系统调用次数、使用批处理操作、改进缓存管理等优化手段，以减小系统调用带来的开销。

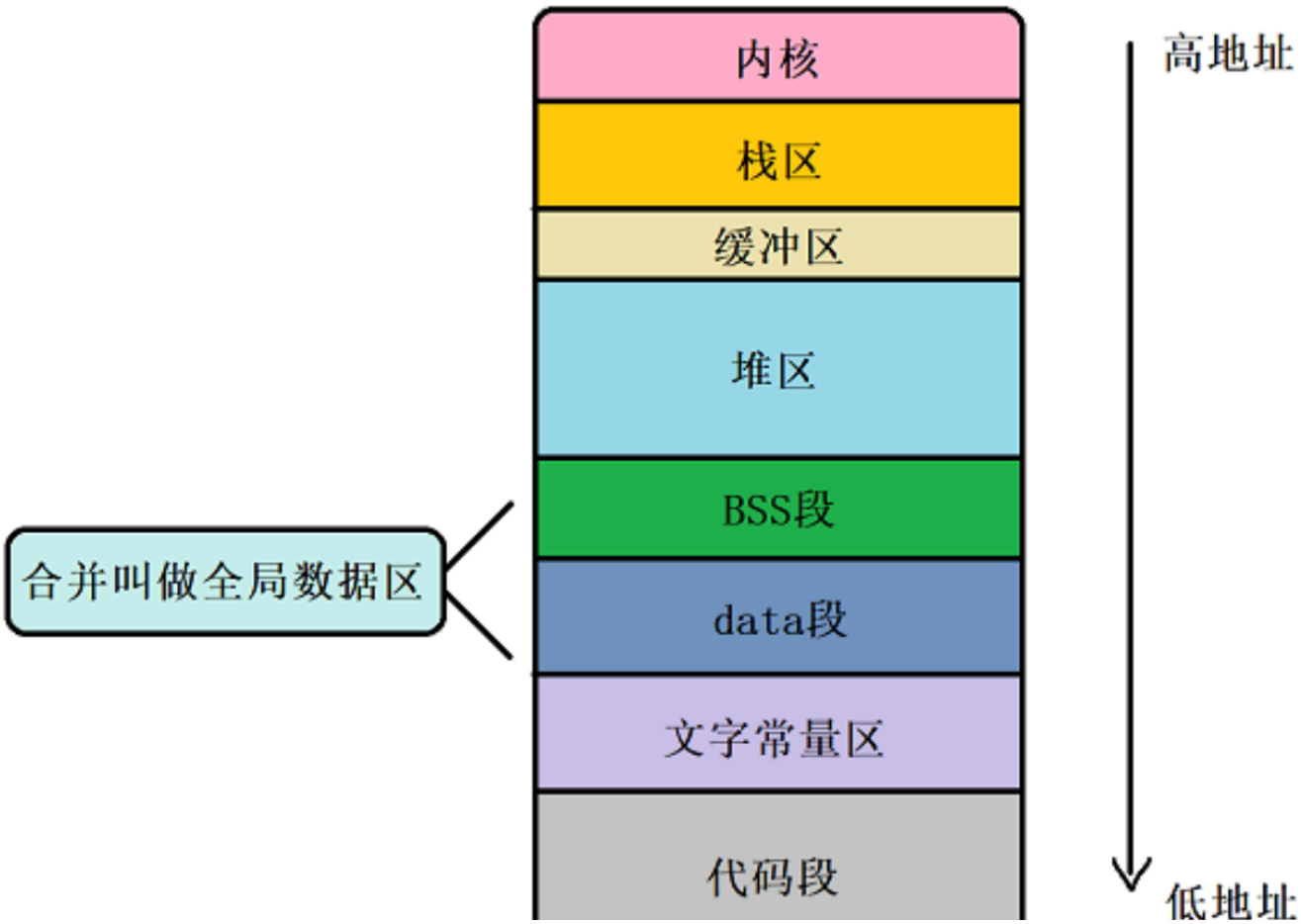
- 从用户态切换到内核态，操作系统做了什么?
  从用户态切换到内核态的过程，通常被称为“上下文切换”。当一个用户程序需要执行一个特权操作（如文件操作、网络通信、内存管理等）时，操作系统会将当前程序从用户态切换到内核态。具体步骤如下：

1. **触发系统调用或异常**：
   - 当用户程序调用系统调用（例如， `read()` 、 `write()` 、 `open()` 等），或者遇到一个硬件中断（例如，I/O请求、时钟中断等），操作系统会知道需要进入内核态来处理这些请求。
   - 系统调用通常通过 `int 0x80` 指令或 `syscall` 指令在 x86 架构中触发。硬件中断则通过中断向量表触发。
2. **保存用户态上下文**：
   - 操作系统需要保存当前执行的用户程序的状态（即上下文），这包括CPU寄存器的值、程序计数器（PC）、堆栈指针等信息。
   - 这些信息会保存在进程的进程控制块（PCB）中，以便后续可以恢复用户态时使用。
3. **切换到内核态**：
   - 通过设置特权级（通常通过修改 `CS` 寄存器中的 `DPL` 字段），处理器进入内核模式。内核可以访问所有的硬件资源并执行特权指令。
   - CPU的状态会从用户态的上下文切换到内核态的上下文，并跳转到内核中的相关代码来处理系统调用或中断。
4. **执行内核代码**：
   - 一旦进入内核态，操作系统会根据触发的事件（系统调用或中断类型）执行相应的内核服务。
   - 如果是系统调用，内核会根据调用号执行相应的内核函数；如果是中断，内核会处理中断请求，如设备驱动程序或内存管理等。
5. **恢复用户态上下文**：
   - 当内核处理完相应的任务后，会将执行结果返回给用户程序。此时，操作系统会恢复之前保存的用户程序的上下文信息（包括CPU寄存器等）。
   - 操作系统会将控制权交还给用户态程序，恢复程序计数器指向原来的位置，继续执行。
6. **返回用户态**：
   - 最终，操作系统通过特权指令（如 `iret` ）将控制权交回给用户程序，恢复到用户态继续执行。

总结来说，从用户态到内核态的切换过程涉及保存用户态的上下文、进入内核态并执行内核代码、然后恢复用户态的上下文。这个过程使得操作系统能够处理低级硬件操作和执行特权指令，而用户程序则运行在受限的用户态中。

- 不同存储介质的速度
  - 寄存器
  - L1L2L3 内存
  - 各种ROM,RAM等
  - 磁盘
- 分别讲一下不同文件系统的区别，每个都讲一下
  - xxfs

    - fat
    - ramfs，为什么要做一个内存里的文件系统?
- Linux下如何管理进程的地址空间（内存模型）？




    - 内核
    - 栈
    - 缓冲区
    - 文件映射区
    - 堆
    - bss：存储全局变量
    - data：
    - text：
    - 代码段：

## 计网

- TCP 三次握手和四次挥手
    - 画图
    - 第二、三次挥手之间为什么需要等待一段时间
    - 第四次挥手之后为什么要等待一段时间再断开连接?
        - 等待多久？几个MSL（帧最大存活时间）

# 数据库

- mysql 有几种存储引擎?
- innoDB 是怎么存储一个行数据的?
- innoDB 索引的类型
- 事务的隔离级别
- MVCC 是什么?

# C++

## 虚函数

虚函数是怎么实现的?

**虚函数相关（虚函数表，虚函数指针），虚函数的实现原理**

构造函数不能为虚函数。原因有两个：

- 一是构造函数在对象构造时调用，**对象构造时虚函数表还没有构造**，所以不能为虚函数；
- 二是**构造函数是静态的，虚函数是动态绑定的**，构造函数构造对象实例的时候，对象的类型还没有确定，无法决定是使用什么函数，静态函数不能是虚函数。

**虚函数指针和虚函数表存放在哪里?**

- **虚函数指针的位置是在对象的内存空间的最前面4个字节**，具体在内存的哪个区域取决于对象是创建在栈上还是堆上。
- **虚函数表是在代码段，是全局的，存放的是虚函数的地址。**

首先我们来说一下，C++中多态的表象是，在基类的函数前加上 **virtual** 关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数，如果是基类，就调用基类的函数。

**实际上，当一个类中包含虚函数时，编译器会为该类生成一个虚函数表，保存该类中虚函数的地址**，同样，派生类继承基类，派生类中自然一定有虚函数，所以编译器也会为派生类生成自己的虚函数表。当我们定义一个派生类对象时，编译器检测该类型有虚函数，所以为这个派生类对象生成一个虚函数指针，指向该类型的虚函数表，**这个虚函数指针的初始化是在构造函数中完成的。**

后续如果有一个基类类型的指针，指向派生类，那么当调用虚函数时，就会根据**所指真正对象的虚函数表指针去寻找虚函数的地址**，也就可以调用派生类的虚函数表中的虚函数以此实现多态。

**非 virtual 的多态**

如果基类中没有定义成 virtual，那么进行

```
Base B;
Derived D;
Base *p = D;
p->function();
```

这种情况下调用的则是 Base 中的 function()。**因为基类和派生类中都没有虚函数的定义，那么编译器就会认为不用留给动态多态的机会**，就事先进行函数地址的绑定（早绑定），详述过程就是，定义了一个派生类对象，首先要构造基类的空间，然后构造派生类的自身内容，形成一个派生类对象，那么在进行类型转换时，直接截取基类的部分的内存，编译器认为类型就是基类，那么（函数符号表［不同于虚函数表的另一个表］中）绑定的函数地址也就是基类中函数的地址，所以执行的是基类的函数。

**对于派生类来说，编译器建立虚函数表的过程其实一共是三个步骤：**

拷贝基类的虚函数表，如果是多继承，就拷贝每个有虚函数的基类的虚函数表。

当然还有一个基类的虚函数表和派生类自身的虚函数表共用了一个虚函数表，也称为某个基类为派生类的主基类。

查看派生类中是否有重写基类中的虚函数， 如果有，就替换成已经重写的虚函数地址；查看派生类是否有自己的虚函数，如果有，就追加自身的虚函数到自身的虚函数表中。

```
Derived *pd = new D(); // pd 指向 D 类的对象
B *pb = pd;            // pb 指向 D 类的对象
C *pc = pd;            // pc 指向 D 类的对象
pb->function();        // 调用 D 类对象的 B 类部分的 function() 方法
pc->function();        // 调用 D 类对象的 C 类部分的 function() 方法
```

其中 pb，pd，pc 的指针位置是不同的，要注意的是派生类的自身的内容要追加在主基类的内存块后。

**析构函数一般写成虚函数的原因**

直观的讲：是为了降低内存泄漏的可能性。

举例来说就是，一个基类的指针指向一个派生类的对象，在使用完毕准备销毁时，如果基类的析构函数没有定义成虚函数，那么编译器根据指针类型就会认为当前对象的类型是基类，调用基类的析构函数 （该对象的析构函数的函数地址早就被绑定为基类的析构函数），仅执行基类的析构，派生类的自身内容将无法被析构，造成内存泄漏。

如果基类的析构函数定义成虚函数，**那么编译器就可以根据实际对象，执行派生类的析构函数，再执行基类的析构函数，成功释放内存。**

### STL

- vector 自动扩容的原理
  - reverse
  - resize
- unordered_map 和 map 的区别

## 提升

- C++的熟悉和掌握程度（实践+八股）
- 数据库八股
- 计网八股
- 其实项目挺好的，也有开源经验
- 但是业务专注于数据库的增删改查，mysql 不熟的话就很难了