✅ Operating Systems

I. Process & Threads

- ✅ **Process vs. Thread**
  - "**Process:** An independent program in execution with its own **isolated** memory space. **Thread:** The smallest unit of execution *within* a process.

    Key Differences:

    1. **Memory:** Processes are isolated; Threads **share** the same memory (Heap, Code, Global variables) but have their own **Stack** and **Registers**.
    2. **Overhead:** Creating or switching processes is expensive (heavyweight); Threads are lightweight.
    3. **Communication:** Threads communicate easily (via shared memory); Processes need **IPC** (Inter-Process Communication)."
- ✅ **Stack vs. Heap**
  - "**Stack** is used for static memory allocation, while **Heap** is for dynamic allocation.
    1. **Management:**
       - **Stack:** Managed automatically by the CPU (LIFO order). Variables are freed when the function exits.
       - **Heap:** Managed manually by the programmer (C++) or Garbage Collector (Java).
    2. **Storage:**
       - **Stack:** Stores local variables and function calls.
       - **Heap:** Stores Objects and global variables.
- 🟡 **Context Switching：CPU/OS 具体保存什么、怎么切**
  - 👉 **context switch（上下文切换）既可以发生在进程之间，也可以发生在线程之间。**
    但在实际讨论中，**更多时候指的是线程切换。**
  - "**Context Switching** is the process of storing the state of the currently running process and restoring the state of the next process so execution can be resumed later.
  - ```
    请求 A
    ↓
    线程 1
    ↓
    被切走
    ↓
    线程 2
    ↓
    线程 3
    ```
    - **记录现场（Save State）：** 记住写到第几页、第几行（保存寄存器 PC, Stack）。

      去倒垃圾（Switch）： 做另一件事。

      恢复现场（Restore State）： 回来翻到刚才那一页，继续写。
    ```

- 🟡 **Process States 生命周期**
  - **New:** The process is being created.

    Ready: It is loaded into memory and waiting for CPU time.

    Running: The CPU is currently executing its instructions.

    Waiting (Blocked): It is waiting for an event (like I/O or user input) and cannot run.

    Terminated: Execution is finished and resources are freed.
- 🟡 **Daemon Process：是什么/怎么创建（偏 Linux）**
  - **Daemon Process 就像是商场的保洁阿姨或者空气净化器。**
    - **Background (后台):** 它们一直在默默工作，不占用前台（不占你的屏幕/Terminal）。
    - **No Interaction (无交互):** 你不需要跟它说话，它也不受你控制。
    - **Always On (常驻):** 电脑一开机它就在运行，直到关机。
- 🔴 **User Space vs. Kernel Space：CPU 如何切换/Protection rings**
  - "**User Space** is where standard applications execute (like a web browser). **Kernel Space** is reserved for the OS kernel and drivers to access hardware directly.
  - **Context switch 的核心是：**

    > CPU 从一个“执行实体”切换到另一个

    而 user → kernel：
    - ❌ 没换线程
    - ❌ 没换进程
    - ✅ 只是权限级别变了

    所以更准确的叫法是：
    👉 **mode switch（特权级切换）*

    The key difference is **privilege**:
    - **User Space** operates in **Ring 3** with restricted access to memory and hardware.
    - **Kernel Space** operates in **Ring 0** with full access to all resources.

    To switch from User to Kernel space (e.g., to read a file), the system must perform a **Mode Switch** triggered by a **System Call** or **Interrupt**."
- 🟡 **System Calls：trap/interrupt 机制（讲到 trap 就偏底层）**
  - "A **System Call** is the interface for a user program to ask the OS Kernel to perform tasks it doesn't have permission to do, like File I/O or creating a new process.

    How it works:
    1. The program executes a special instruction that triggers a **Trap** (software interrupt).
    2. This switches the CPU from **User Mode** to **Kernel Mode**.
    3. The OS executes the requested task (like `read()` or `fork()`).
    4. Finally, it returns the result and switches back to **User Mode**."
- 🟡 **User Thread vs Kernel Thread 区别**

- User Thread (**用户线程**): 是**你脑子里的分身**。
  - **线程的创建 / 切换 / 调度都在 user space 完成**

    特点
    - 内核 **不知道** 这些线程的存在
    - OS 眼里：**只有一个进程**
    - 线程调度由 **用户态 runtime / 库** 做

    优点

    ✅ **切换快**（不进内核）
    ✅ **创建 / 销毁成本低**
    ✅ 可自定义调度策略

    致命缺点

    ❌ **一个线程阻塞，整个进程阻塞**（blocking syscall）
    ❌ **无法利用多核 CPU**（内核只调度到一个执行流）
    ❌ 和系统调用 / IO 天然不友好

  Kernel Thread (**内核线程**): 是**真正雇佣的多个工人**。
  - **线程由 OS 内核直接管理和调度**

    特点
    - 内核 **完全感知**
    - 每个线程都是调度实体
    - 可以跑在 **不同 CPU 核心**

    优点

    ✅ **真正并行（多核）**
    ✅ 一个线程阻塞 **不影响其他线程**
    ✅ IO / syscall 天然支持

    缺点

    ❌ 创建 / 切换 **开销更大**
    ❌ 上下文切换涉及内核

II. Concurrency & Synchronization

- ✅ **Race Condition + 例子**
  - "A **Race Condition** occurs when multiple threads access and **modify shared data concurrently**. The final outcome depends on the **unpredictable timing** (order of execution) of the threads.
- ✅ **Critical Section**
  - **那一段会修改共享数据的代码。** 规则很简单：这个区域，一次只能有一个人进去。
- ✅ **Mutex vs. Semaphore（会问到"能不能用 binary semaphore 当 mutex"这种也算常见延伸）**
  - Mutex (**互斥锁**): 是**厕所钥匙**。
    - Ownership (**所有权**): 谁拿了钥匙，谁就必须负责开锁归还。如果你锁了门，别人不能帮你开。
    - 一次只能进一个人。

    Semaphore (**信号量**): 是**餐厅的空位计数器**。

        - **No Ownership:** 只有数字。进了人减 1，出了人加 1。
        - 厨师（Producer）做好了饭可以加 1，服务员（Consumer）端走饭减 1。（**不同线程操作**）。
- 🟡 **Spinlock vs. Mutex（busy waiting 何时值得）**
    - **Mutex:** 看到有人，你回座位**睡觉**（Sleep/Block）。那人出来了摇铃叫醒你。
        - *代价：* 睡觉起床（Context Switch）很累。

      Spinlock: 你站在门口**不停地敲门**问“好了吗？好了吗？”（Busy Waiting）。
        - *代价：* 浪费你的精力（CPU Cycles），但只要他一开门你能立刻进去。
- 🟡 **Producer-Consumer：用 semaphore / condition variable 讲清楚（偶尔会问）**
    - **核心概念 (ELI5)**
        - **场景：** 回转寿司店。厨师（Producer）放盘子，顾客（Consumer）拿盘子。传送带（Buffer）长度有限。
        - **问题：** 传送带满了，厨师要停（Wait）；传送带空了，顾客要停（Wait）。
        - **解决：** 用两个 Semaphore：
            1. `Full_Count` (盘子数)
            2. `Empty_Count` (空位数) 外加一个 Mutex 保护传送带不被两个人同时抓同一个盘子。
- ✅ **Atomic operations：为什么 i++ 不 atomic**
    - **Q: Why is `i++` not atomic?** "`i++` is actually a **Read-Modify-Write** operation (3 CPU instructions). A context switch can happen **between** these instructions. If another thread modifies `i` during that gap, the update will be lost. To make it safe, we need **locks** or **atomic instructions**."
- 🟡 **CAS：是什么/解决什么（偏并发进阶）** (Compare-And-Swap)
    - 👶 **核心概念 (ELI5)** 这是一个**乐观**的更新方式。 你不加锁，而是去更新的时候跟内存说：“**我觉得**现在的数值是 5。如果是 5，请帮我改成 6。如果现在的数值**不是** 5（说明有人悄悄改过了），那你就告诉我失败了，我重新再试。”
- 🟡 **Memory visibility / ordering（“看不见别人写入”）**
    - **Visibility (可见性):** CPU 都有自己的**小金库 (Cache)**。CPU A 改了数据，只改在自己的 Cache 里，没写回主内存。CPU B 去主内存读，读到了旧数据。CPU B “看不见” A 的修改。

      Volatile (Java): 强制要求：“别藏在 Cache 里，直接读写主内存！”保证所有threads看到的都是最新的。
- 🟡 **Livelock vs Deadlock**
    - **Deadlock:** 两人在独木桥相遇，**都不动了。** (State is blocked).

      Livelock: 两人在走廊相遇，你想往左让，他也往左让；你往右，他也往右。**两人都在疯狂动，但谁也过不去。** (State is changing, but no progress).
- ✅ **Read and write lock适用场景（读多写少）**
    - 👶 **核心概念 (ELI5) 公告栏原则。**
        - 大家都可以**同时看**（Read）。
        - 但只要有一个人要**改**（Write），其他所有人（包括看的和改的）都必须滚蛋，清场。
        - **适用：** 读多写少（比如刷微博的人多，发微博的人少）。

III. Memory Management

- ✅ **Virtual Memory：为什么需要**
  - 背诵一句话

> Virtual Memory =
> **让“每个进程都以为自己独占一整块连续的大内存”，**
> **实际上由操作系统偷偷在后面拼。**

---

1) 物理内存 physical memory 是什么?

就是你机器上真正的 RAM（内存条）。
它可以想成一排连续编号的格子（0…N），CPU 最终读写的就是这些真实格子。

但现实里：

- RAM 资源有限
- 同时跑很多进程
- 每个进程都“以为”自己有一大段连续内存

如果让每个程序直接用物理地址：

- 程序 A 很容易读到/写到程序 B 的数据（安全灾难）
- 内存碎片会让“连续大块内存”很难找（管理灾难）
- 进程搬家（内存整理）会导致地址全变，程序会崩（工程灾难）

---

2) 虚拟内存 virtual memory 是什么?

虚拟内存是“每个进程看到的一套假地址空间”。

关键点：

- 每个进程都觉得自己从地址 0 开始，有一大片连续内存
- 这套地址**不等于**真实 RAM 地址
- CPU 发出的地址先叫 **虚拟地址**，需要翻译成 **物理地址** 才能访问 RAM

你可以把它当作：

> 操作系统给每个进程发了一张“地图”，地图上的坐标（虚拟地址）不是真实世界坐标（物理地址），但 OS 保证能把地图坐标翻译到真实坐标。

---

3) 为什么要 VPN 和 PFN（以及 offset）?

因为“按每个字节做映射”太贵，所以把内存切成固定大小的块来管理。

分块的两个名字

- 虚拟地址空间里的块：**virtual page（虚拟页）**
- 物理内存里的块：**physical frame（物理页框/页帧）**

它们**大小相同**（比如都是 4KB）。

地址拆分

假设 page size = 4KB = 2^{12}，那么任意地址都能拆成：

- **offset（页内偏移）**：低 12 位（0~4095），表示在这一页里的第几个字节
- 剩下高位：页号
    - 虚拟页号叫 **VPN (Virtual Page Number)**
    - 物理页框号叫 **PFN (Physical Frame Number)**

所以：

```
虚拟地址 VA = [ VPN | offset ]
物理地址 PA = [ PFN | offset ]
```

page table 做的事就是：把 VPN 翻译成 PFN。
offset 不变，直接拼回去。

---

4) mapping 是“多大 map 到多大”?

粒度就是 1 页（page size）。

> page table 映射的是：1 个虚拟页（比如 4KB）→ 1 个物理页框（也是 4KB）

不是 1 byte→1 byte，也不是整个进程一次性映射。

举个特别直观的例子（4KB 页）

- 虚拟页 #5 表示虚拟地址范围：
  `5*4096 ~ 5*4096+4095`
- page table 说：虚拟页 #5 → 物理页框 #123

那意味着：

- 虚拟地址 `5*4096 + x`（x 在 0~4095）
  会变成
- 物理地址 `123*4096 + x`

你看：**映射的单位就是整整一页 4KB**，页内偏移 x 原样带过去。

内存长这样

```
虚拟内存: | P1 | P2 | P3 | P4 |
物理内存: | P3 | P1 | P9 | P4 |
```

👉 顺序完全可以不一样

四、那“虚拟内存 = 用硬盘当内存”吗?

不完全对，但也不完全错。

Page Fault（缺页异常）

```
访问某个虚拟页
↓
不在 RAM
↓
CPU trap
↓
OS 从磁盘 swap 读
↓
再继续执行
```

👉 这是 **虚拟内存的“扩容能力”**

但重点是：

> **虚拟内存的核心价值不是 swap，**
> **而是“隔离 + 抽象”。**

- 🔴 **Page Table：VA→PA**
  - 页表 = “地址翻译字典”：把虚拟页号 VPN 映射到物理页框号 PFN。
- 🔴 **TLB：是什么/为什么关键（偏体系结构+OS）**
  - TLB = “页表缓存”，没有它每次访存都要多次查页表，性能会崩。
- 🟡 Page Fault：
  - **A Page Fault occurs when a program accesses a page not currently in physical memory (RAM).** Steps:
    1. **Trap:** CPU raises an interrupt to the OS.
    2. **Context Switch:** OS puts the process in 'Waiting' state.
    3. **Disk I/O:** OS finds the page in Swap Space and reads it into a free frame in RAM.
    4. **Update:** OS updates the Page Table (sets Valid bit).
    5. **Retry:** The instruction is restarted."
- 🟡 Copy-on-Write（fork 为啥快）
  - fork 先“共享同一份页”，谁先写谁才复制。
- 🔴 Thrashing：成因/预防（很像考试题）
  - 系统大部分时间在换页（swap），真正计算时间很少。
- 🔴 Segmentation vs Paging

  Segmentation 是“按意义切内存”，
  Paging 是“按大小切内存”。
  - **Paging (分页):** 切方块吐司。不管你是肉还是菜，统统切成 4KB 的小方块。
    - *好处：* 整齐，好管理。
    - *坏处：* 这一块里可能只用了 1 个字节，剩下的浪费了（内部碎片）。

    Segmentation (分段): 切蛋糕。你想要多大（比如“代码段”、“数据段”），就切多大给你。
    - *好处：* 逻辑清晰，不多不少。
    - *坏处：* 剩下的边角料太碎了，塞不进新东西（外部碎片）。
- 🔴 Internal vs External Fragmentation：paging/segmentation 各自问题

    - **Internal (内部 - 占着茅坑不拉屎):** 给你分配了 4KB 的箱子，你只放了 1KB 的东西。剩下 3KB 在箱子**内部**浪费了。

      External (外部 - 缝隙太小塞不进): 仓库里空地加起来有 100KB，但都是这里 10KB、那里 20KB 的**碎缝隙**。现在来个 50KB 的大箱子，**哪里都塞不进去**。
- 🟡 Swapping / swap space Swap = Disk 和「物理内存（RAM）」之间的交换
    - 把不常用的页临时放到磁盘，给常用页腾内存。
- 🟡 OOM 什么时候发生 + OS 怎么处理
    - OOM = 系统/进程再也分不到可用内存（RAM+可用 swap/限制），只能失败或杀进程。

IV. Scheduling & I/O / File System

- 🟡 **Scheduling Algorithms（RR/FCFS/SJF/Priority）**
    - FCFS：谁先排谁先买

      SJF：买得快的人先买（让队伍整体更快）

      Priority：VIP 先买

      RR：每人买 10 秒，没买完去队尾
- 🟡 **Preemptive vs Non-preemptive**
    - Preemptive = OS 能抢走 CPU；Non-preemptive = 任务自愿让出/结束才换人。
- 🟡 **Starvation + aging（偶尔会问概念）**
    - Starvation = 某些任务长期得不到 CPU；Aging = 等得越久优先级越高来防饿死。
- 🟡 **Interrupt vs Polling（概念可能在系统设计/性能里出现）**
    - Interrupt = 设备有事主动叫 CPU；Polling = CPU 不停问设备“好了吗”。
- 🟡 **同步 IO vs 异步 IO**
    - 同步 IO：发起后要等结果（线程被阻塞或忙等）；异步 IO：发起后立刻返回，完成后再通知你。

✅ Computer Networks

I. Protocols & Layers

- 🟡 OSI 7 层（能说清层次+例子即可；背全套不常硬考）
    - Application（应用）：HTTP、DNS

      Presentation（表示）：TLS/加密、编码（UTF-8）、压缩

      Session（会话）：会话管理（现实中很多被应用/TCP吃掉了）

      Transport（传输）：TCP/UDP

      Network（网络）：IP、routing

      Data Link（数据链路）：Ethernet、Wi-Fi、MAC

      Physical（物理）：网线/光纤/无线电波
- 🟡 TCP/IP vs OSI

- **背诵一句话：** TCP/IP 是“真实世界的 4/5 层模型”，OSI 是“更细的 7 层教学模型”。

- ✅ TCP vs UDP + use cases
  - 连接：TCP 有连接（handshake），UDP 无连接

    可靠：TCP 有重传/ACK/顺序；UDP 没有（要应用自己做）

    延迟：UDP 通常更低开销（但不一定更快，取决于场景）

    “TCP 适合需要可靠、有序、不能丢的数据，比如网页内容、交易、文件；UDP 适合低延迟或允许少量丢包的场景，比如直播、会议、游戏。

- ✅ MAC vs IP
  - MAC：链路层地址（网卡地址），在同一 LAN 内用

    IP：网络层地址（逻辑地址），路由器靠它转发到不同网段

- 🟡 ARP
  - **背诵一句话：** ARP 用来在局域网里把 **IP → MAC** 映射出来。
  - 主机要发给某个 IP（同网段）

    查 ARP cache，没有就广播：`谁是这个 IP?`

    对方回：`我是，我的 MAC 是 xxx`

    缓存后再用 Ethernet 帧发过去

- 🟡 ICMP（ping/错误报告）
  - **背诵一句话：** ICMP 是 IP 的“诊断/报错消息”，ping 用它做连通性测试。
    - ping：ICMP Echo Request/Reply
    - traceroute：利用 TTL 递减触发 ICMP Time Exceeded（不同系统实现略有不同）

    面试补一句： “ICMP 不承载业务数据，主要用于网络可达性和错误反馈（比如目的不可达）。”

- ✅ DHCP（DORA）
  - “主机刚连上网没 IP，会广播 Discover；DHCP server 回复 Offer（给可用 IP、网关、DNS、租期）；主机发 Request 选择一个；server Ack 确认并下发配置。”

    追问： 为什么要 Request？→ 防止多个 server Offer 时冲突，让客户端明确选谁。

- ✅ NAT
  - **背诵一句话：** NAT 把内网私有 IP “翻译”成公网 IP（常见是端口映射 PAT），让很多设备共享一个公网地址。

- 🟡 IPv4 subnet/CIDR 计算（某些公司会考，但不是所有）
  - **背诵一句话：** CIDR `/n` 表示“前 n 位是网络号”，主机数约 `2^(32-n) - 2`（常规网段）。
    快速表（最常用）：
    - /24：256 个地址（可用 254）掩码 255.255.255.0
    - /16：65536（可用 65534）掩码 255.255.0.0
    - /8：16777216（可用 16777214）掩码 255.0.0.0

- 🟡 Routing / default gateway（讲概念即可）
  - **背诵一句话：** 路由器靠 **routing table** 决定下一跳；default gateway 是“找不到更具体路由时走的出口”。

II. TCP/UDP Deep Dive

- ✅ TCP 3-way handshake
  - “客户端发 SYN（带初始 seq x），服务端回 SYN-ACK（seq y，ack x+1），客户端再 ACK（ack y+1）。三次握手让双方确认收发能力并同步序号，为可靠传输做准备。”
- ✅ TCP 4-way termination + TIME_WAIT
  - **背诵一句话：** TCP 关闭是两边各说一次“我不发了”（FIN），所以常见 4 段；TIME_WAIT 是主动关闭方“等一会儿”确保干净结束。
    流程：
    A：FIN → B：ACK → B：FIN → A：ACK
- ✅ Sequence / ACK 作用
  - **背诵一句话：** Sequence 给字节编号，ACK 告诉对方“我已经连续收到哪里”。
    - TCP 是 **byte stream**：seq/ack 通常按“字节序号”推进
    - ACK 多为 **cumulative ACK**（累计确认）：`ACK = 下一个期望的 seq`
- 🟡 Sliding Window（flow control）
  - **背诵一句话：** Sliding Window 是“一次允许在路上飞多少数据”，提升吞吐并控制接收端别被淹没。
- 🟡 Flow control vs Congestion control
  - **背诵一句话：** Flow control 看“接收方吃不吃得下”；Congestion control 看“网络会不会堵车”。
    - Flow control：靠 **rwnd**（receiver window）
    - Congestion control：靠 **cwnd**（congestion window），慢启动/拥塞避免等算法
- 🟡 TCP 可靠传输机制（序号/ACK/重传/校验）
  - **背诵一句话：** 可靠 = **序号** + **ACK** + **重传** + **校验**（再加窗口/定时器）。
- 🟡 Head-of-Line Blocking（TCP/HTTP1.1）
  - **面试一句话：** “TCP 的有序交付导致丢包会阻塞后续数据交付，这是 HOL 的根源。”
  - **背诵一句话：** 队头阻塞 = “前面一个包丢了，后面都得等”。

III. HTTP & Web

- ✅ 输入 URL 发生什么（DNS→TCP→TLS→HTTP）
  - **面试口述（60–90秒）：**
    1. 浏览器解析 URL（scheme/host/path/query），先看 **cache**（浏览器/OS/DNS 缓存）
    2. 没缓存就做 **DNS 查询**（可能递归到权威 DNS）拿到 IP
    3. 连接：HTTP/1.1/2 通常走 **TCP 3-way handshake**；如果是 HTTPS 会继续 **TLS handshake**
    4. HTTPS：TLS 建立加密通道后发 **HTTP request**（如 GET /）
    5. 服务端返回 **HTTP response**（状态码、headers、body）
    6. 浏览器解析 HTML，遇到资源再并发请求（CSS/JS/图片）；渲染流水线（DOM/CSSOM → render）
       （加一句加分项：HTTP/2 多路复用；HTTP/3/QUIC 走 UDP。）
- ✅ HTTP methods
  - **背诵一句话：** GET 读，POST 提交，PUT 全量改，PATCH 部分改，DELETE 删，HEAD 只要头，OPTIONS

问规则。

- ✅ GET vs POST（含 idempotency））
  - **背诵一句话：** GET 用来“拿”，POST 用来“交/做事”；**GET 通常幂等**，POST 通常不幂等。
- ✅ Cookies vs Sessions
  - “Cookie 存在浏览器，会随请求带回服务器；Session 存在服务器（或集中存储如 Redis）。常见做法是：服务器创建 session，并把 session id 放到 Cookie 里。这样服务器能根据 id 找到你的登录状态、购物车等。”
- ✅ Cookie/Session/Token 区分
  - **背诵一句话：**
    - **Cookie**：浏览器存储/自动携带的小数据
    - **Session**：不是用户的所有数据，只是「当前会话需要的、临时性的、和登录状态相关的数据」。
    - **Token**：一串“凭证字符串”，常用于 API 鉴权（可无状态）（cookie里也可以存session）
  - **用户访问流程（经典）**

```
1. 从 cookie 读 session_id
2. 用 session_id 找 session
3. 从 session 拿 user_id
4. 用 user_id 查数据库（按需）
```

📌 重点：

- **cookie 不存用户数据**
- **session 不存完整用户数据**
- **数据库才是唯一真相源**

⚠️ 正确做法：

- session：只存 `user_id`
- 页面数据：**实时从数据库查**

📌 原则：

- **只对这次会话有意义**
- session 过期就没关系

7️⃣ 用户关闭浏览器 / session 过期

```
浏览器：
  cookie 失效 or 被清除

服务器：
  session 删除

数据库：
  数据还在（完全不受影响）
```

👉 结果：

- 用户被登出
- 需要重新登录
- **用户数据不丢**

8️⃣ 用户再次访问

```
cookie：
  没有 session_id

服务器：
  当成新会话

数据库：
  用户数据依旧完整
```

- **Token 就是：**
- **数据库**：用户所有永久数据（不变）
- **token（access token）**：客户端带着的“可验证凭证”（你是谁/权限/过期时间）
- **cookie（可选）**：只是 token 的存放方式之一（也可以存在内存/localStorage）
- **session（可选/弱化）**：不一定需要；但“刷新 token / 登出黑名单”等场景可能会引入服务端状态

> 关键差异：
> session **模式：服务器记住你是谁**
> token **模式：客户端随请求带“我是谁”的证明，服务器验证即可**

- 不用再去服务器查 session，
  光看这串字符串本身，服务器就能知道你是谁、有没有权限。

  1. 登录成功 → 发 **access_token(短)** + **refresh_token(长)**
  2. 请求 API → 带 access_token → 服务器验签/过期/权限
  3. access 过期 → 用 refresh 换新 access
  4. 登出 → 撤销 refresh（必要时配合版本号/黑名单）

  面试高分说法（最常用组合）：

  - 方案 A（传统 Web）：Cookie 存 session id → 服务器查 Session（有状态）
  - 方案 B（现代 API）：客户端带 Token（如 JWT）→ 服务端验证签名/权限（可无状态）
- 🟡 LocalStorage vs SessionStorage
  - 生命周期：LocalStorage 关闭浏览器仍在；SessionStorage 关 tab 就没

    作用域：LocalStorage 同源共享；SessionStorage 同源但**每个 tab 独立**

    安全：都能被 JS 访问，**遇到 XSS 会被读走** → 不适合存敏感 token（很多公司会追问这个）
- ✅ CORS（是什么/怎么解决）
  - **举个例子：**
    1. **场景：** 用户在 `www.my-blog.com` (前端) 浏览网页，网页需要调用 `api.data-`

`provider.com` (后端) 的数据。

2. **问题：** 浏览器发现源 (`my-blog`) 与数据源 (`data-provider`) 不同，判定为跨域，由于没有 CORS 许可，浏览器阻止数据读取，报错。
3. **CORS 解决：** `api.data-provider.com` 的服务器在响应头中添加：
    - `Access-Control-Allow-Origin: https://www.my-blog.com`
    - 这条声明意味着："允许 `www.my-blog.com` 跨域访问我的数据"。
4. **结果：** 浏览器检查到此响应头，允许前端读取 `api.data-provider.com` 的数据。

IV. HTTPS & Security

- 🟡 TLS handshake（讲清"非对称建通道→对称跑数据"即可）
    - **背诵一句话：** TLS 用 **Asymmetric** 安全地"协商密钥"，然后用 **Symmetric** 高速加密传数据。
- ✅ Symmetric vs Asymmetric（性能/密钥管理）
    - **背诵一句话：** Asymmetric 解决"怎么安全交换密钥"，Symmetric 解决"怎么快"。
- 🟡 DDoS 基本概念 + 基础缓解（rate limit/CDN）
    - **背诵一句话：** DDoS 是"很多机器一起把你打爆（带宽/连接/CPU）"，核心缓解是 **提前挡在边缘 + 限流。**
- ✅ XSS + 防护
    - **背诵一句话：** XSS 是"把恶意 JS 注入页面让浏览器替他执行"，防护核心是 **不让脚本注入 + 限制脚本能力。**
- ✅ CSRF + 防护
    - **背诵一句话：** CSRF 是"利用浏览器自动带 Cookie，诱导你在已登录状态下替他发请求"，防护核心是 **让请求带上'只有你网站能拿到的证明'。**

V. Advanced & System Design Related

- 🟡 DNS 解析：递归 vs 迭代 + 记录类型（A/AAAA/CNAME/MX）
    - 递归 = "你帮我查到最终答案再回来"；迭代 = "我给你下一站，你自己继续问"。
- ✅ Load Balancer：L4 vs L7（system design 常见）
    - "L4 负载均衡在传输层，基于 4 元组（src/dst IP+port）把连接转发到后端，通常更快更通用，适合任何 TCP/UDP 协议。L7 在应用层，能理解 HTTP/HTTPS，按 host/path/header/cookie 做路由、做重写、灰度发布、鉴权、限流等，但开销更大（尤其要解析 HTTP，甚至 TLS 终止）。实际架构里经常是 L4 做入口分流，L7 做应用路由。"
- ✅ Proxy vs Reverse Proxy（Nginx 之类，system design 常见）
    - Forward Proxy（正向代理）代表"客户端"；Reverse Proxy（反向代理）代表"服务器"。
- ✅ CDN 原理（边缘缓存）
    - CDN = 把内容缓存到离用户更近的边缘节点，减少延迟并抗流量峰值。
- ✅ WebSocket vs Long Polling（实时聊天）
    - **Long Polling = 客户端不断"等消息"；WebSocket = 建一条长连接，双方随时互推。**
        - Long polling：你一直打电话问"有新消息吗？没有我就先等着"，挂了又再打。
        - WebSocket：你和朋友开着电话不挂，谁想到什么随时说。