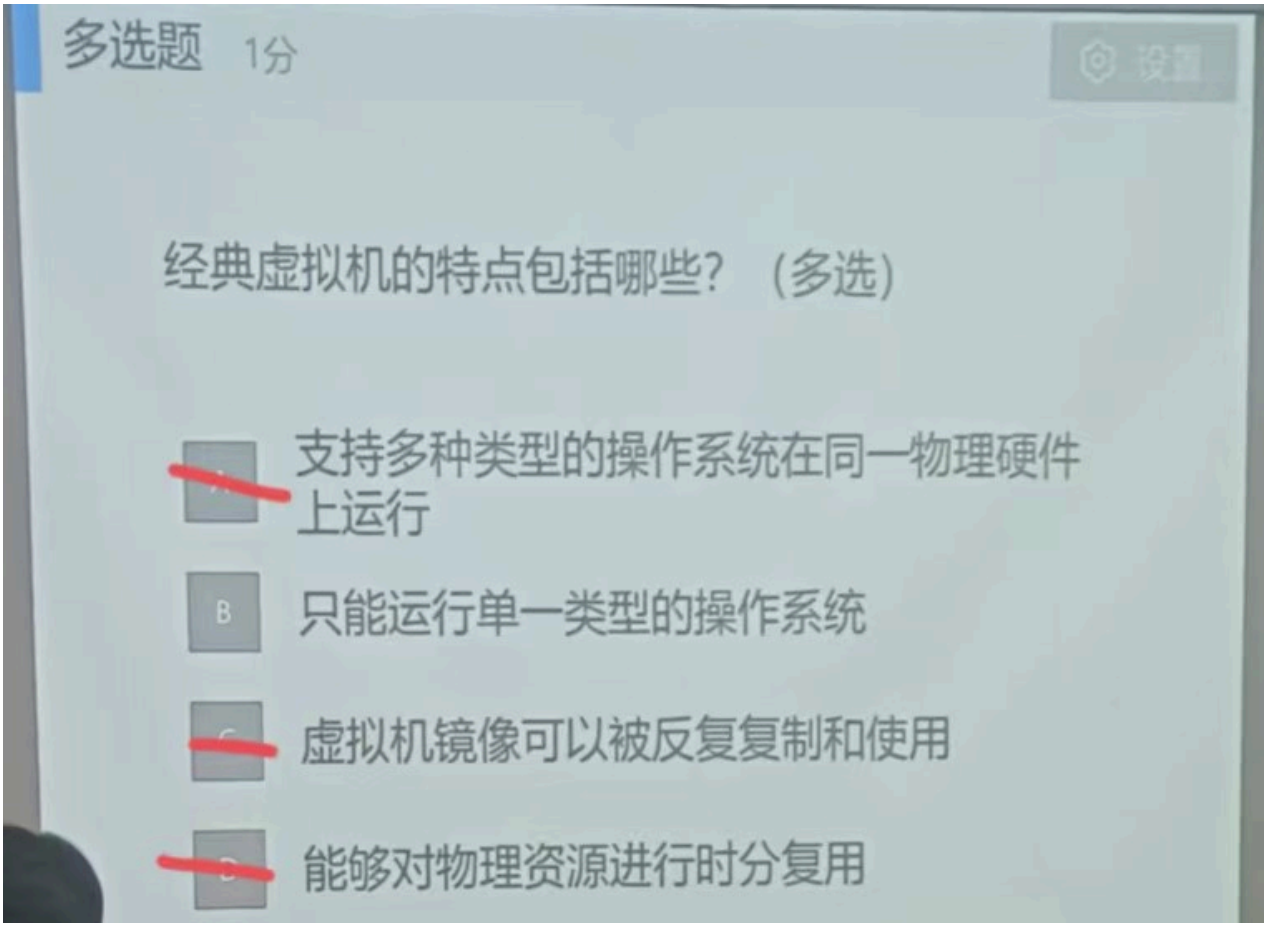
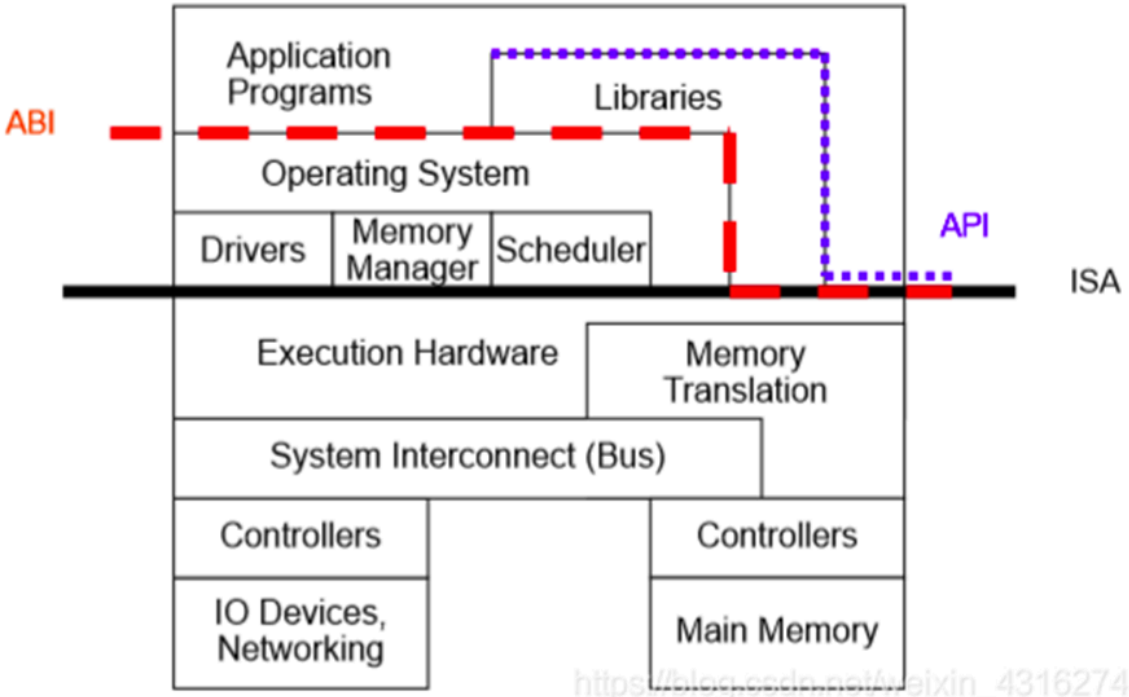
第六章 虚拟化

6.1 虚拟化技术概述

1. 定义：
    1. **虚拟化**(Virtualization)是计算机资源的**逻辑表示**，主要目的是使上层计算或应用与下层的资源或管理**解耦**，把底层资源抽象成另一种形式的资源，从而提供给上层使用和分享。
    2. **虚拟机监视器**(Virtual Machine Monitor)为软件和硬件的复制提供必要的机制。
    3. 虚拟机本质上是一套**软件系统**，它和物理机(Physical Machine)可形成对应。
    4. 对虚拟机来说，也存在虚拟处理器、虚拟存储、虚拟网络的概念，其中**处理器（计算）虚拟化**的实现最为关键。
    5. 经典虚拟机的实现方式主要采取**宿主型**(Hosted VM)，即虚拟化软件直接安装在宿主机的OS中，虚拟内存、磁盘、I/O设备的状态都**被封装成专用的文件**保存在宿主机上。
2. 经典虚拟机的**特点**：
    1. 多态(Polymorphism)：支持多种类型的OS。
    2. 重用(Manifolding)：虚拟机的镜像可以被反复复制和使用。
    3. 复用(Multiplexing)：虚拟机能够对物理资源时分复用。
3. 

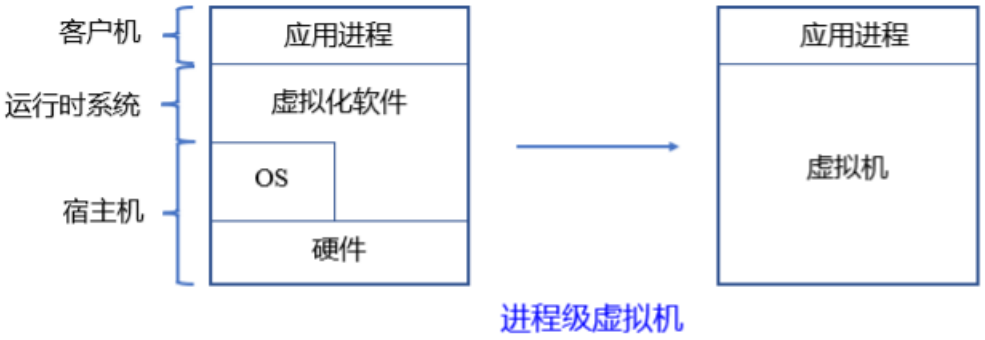
6.2 虚拟机核心原理和技术

1. **系统接口**
    1. 计算机系统的组建离不开各式各样的接口，接口将两个不同的对象环境联系起来，使之得以协调运行。
    2. 最**基本**的**接口**是**微处理器指令集架构**(Instruction Set Architecture, ISA)。
    3. 然而，应用程序一般不是借助于ISA直接运行在硬件平台之上的，多个软件应用一般也**不完全隔离运行**，它们共同运行在某个OS之上，并共享其服务。
    4. **应用程序二进制接口**(Application Binary Interface, ABI)给程序提供使用硬件资源和系统服务的接口，ABI包括**所有用户指令**和**部分系统调用接口**，应用程序和OS与硬件的交互都通过ISA接口完成，而**应用程序与关键硬件资源的交互**是**间接**的，需要OS通过系统调用方式实现。
    5. **应用程序接口**(Application Programming Interface, API)通常由某种高层次语言定义，其关键元素是一个能够由应用程序调用来触发多种服务的系统标准库。一般来说，API的功能往往涉及一个或多个ABI层面的OS调用。
    6. 从**不同角度看机器**
        1. 从ISA角度看，机器可以指代整个计算机硬件架构和设备。
        2. 从ABI接口层面看，机器可以指代整个硬件设备连同其上的OS。
        3. 从API角度看，一个完整的机器还包括了关键的库函数，需要特定程序的支持。

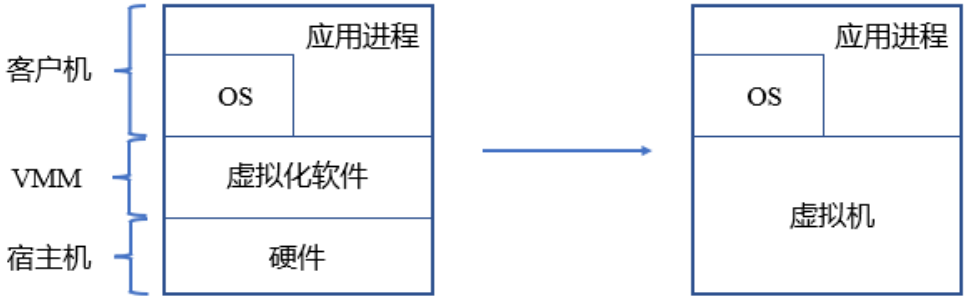
    7.
2. 运行模式
    1. 各类接口的引入对于保护多用户多程序并发运行是有极大好处的，各类应用对资源的访问是间接的，需要通过OS并借助系统调用来完成，为实现这一过程，大部分系统支持至少两种操作模式：Kernel Mode & User Mode。
    2. 与之对应，**ISA被设计为两大类**： User ISA & System ISA。
        1. User ISA：指对于应用程序完全可见的指令子集。
        2. System ISA：仅为具有较高监控权限的系统（如OS）可见.
3. 虚拟机的分类

1. 从某一接口出发，通过为底层机器配置一个耦合的软件而形成的、具有新的虚拟接口的系统，就是一个虚拟机。在这个虚拟机之上，原始上层软件可以无缝的运行。
2. **进程级虚拟机**(Process VM)：是基于**OS**的VM，是应用程序二进制接口**ABI层面**设立的虚拟机，虚拟化软件需要为用户程序模拟所需要的**主机硬件功能**。



进程级虚拟机

3. **系统级虚拟机**(System VM)：是基于**硬件**的VM，是**ISA层面**设立的虚拟机，虚拟化软件设置在**硬件设备和软件系统**之间。



系统级虚拟机

288

以下不属于系统接口的是：

A ISA

B ABI

C API

D UI

4.

以下对机器描述错误的是：

5.

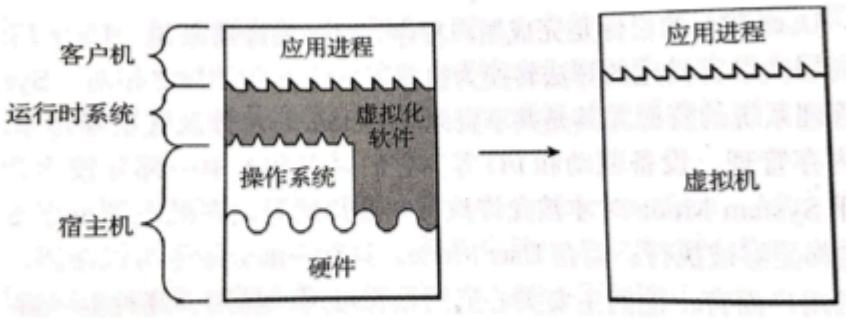
- A 机器是指为安装任何软件的物理机器。
- B 从ISA角度看，机器可以指代整个计算机硬件架构和设备。
- C 从ABI接口层面看，机器可以指代整个硬件设备连同其上的OS。
- D 从API角度看，一个完整的机器还包括了关键的库函数，需要特定程序的支持。

微处理器指令集架构包括：

6.

- A Kernel Mode
- B Kernel ISA
- C User ISA
- D System ISA

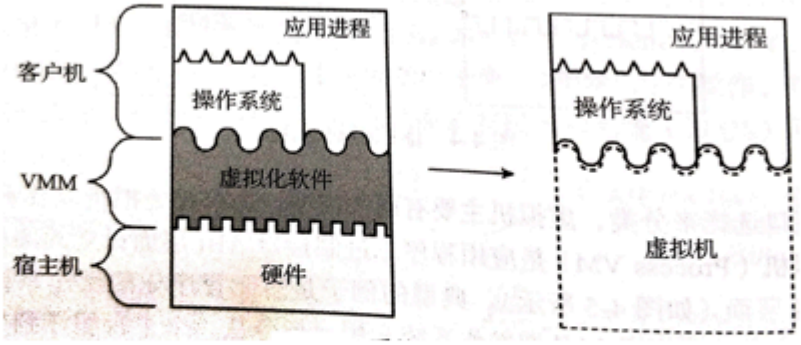
7. 下图的虚拟机属于：



A 进程级虚拟机

B 系统级虚拟机

8. 下图的虚拟机属于：



A 进程级虚拟机

B 系统级虚拟机

6.3 虚拟化的实现

1. Xen虚拟机技术
    1. 概述：
        1. Xen是由剑桥大学计算机实验室开发的一个开源项目。是一个直接运行在计算机硬件之上的用以替代操作系统的软件层，它能够在计算机硬件上并发的运行多个客户操作系统（Guest OS）。
        2. **硬件支持**：x86、x86-64、安腾(Itanium)、Power PC和ARM多种处理器。
        3. **支持的客户操作系统**：Linux、NetBSD、FreeBSD、Windows、Solaris和其他常用的操作系统。

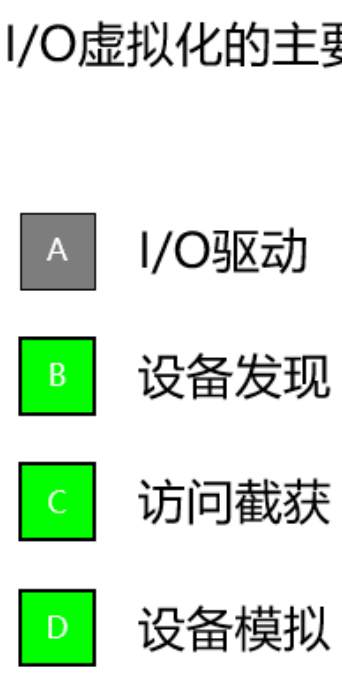
4. **资源占用少**：采用微内核设计，体积小（约1MB），接口精简，运行效率高，资源占用极少。

2. Xen的基本组件
    1. Xen Hypervisor
        1. Xen虚拟机技术创造了一个**基于原生态硬件的Hypervisor**，它把虚拟化软件直接安装在不加任何修饰的服务器**原生态硬件**上。
        2. Type-1（裸金属）Hypervisor
    2. **Domain0**
        1. 对Xen来说，Hypervisor上一般运行多个客户作业系统，有个特殊的虚拟机会成为管理其它客户作业系统的**高权限管理VM**，Xen称它为Domain0。
        2. Domain0能够和Hypervisor交互，其上一般安装用于操纵底层硬件的驱动软件，可以**启动或者停止虚拟机**。
        3. 核心功能：
            1. 硬件访问
            2. 资源分配
            3. 虚拟机管理
    3. **DomainU**
        1. 与Domain0相对应，其他客户使用的是在该管理VM旁运行的客户VM，称为**DomainU**。
        2. 这种对虚拟机功能的划分也应用在Microsoft的**Hyper-V**虚拟化技术中，**管理VM的一般称为Parent Partition**，而**客户VM一般称为Child Partition**。
3. **全虚拟化**(Full Virtualization)
    1. 不需要修改客户操作系统的虚拟化技术。虚拟机监视器通过**完全模拟**底层硬件环境，让客户操作系统**以为自己运行在真实的物理硬件**上。
    2. 软件辅助的全虚拟化
        1. 挑战：x86架构的指令集中包含许多**敏感指令**（Sensitive Instructions）和**特权指令**（Privileged Instructions），这些指令在虚拟环境中可能导致异常或安全问题。
        2. **优先级压缩**：
            1. 当客户OS的内核尝试执行特权指令（如修改CPU状态），指令会被捕获（Trap）到Hypervisor，模拟指令效果并返回结果。
        3. 二进制代码翻译
            1. Hypervisor扫描客户OS的指令流，识别敏感指令。
            2. 将敏感指令**替换为等效的Hypervisor调用**（Hypercall）**或安全指令。**
            3. 翻译后的代码直接在Hypervisor控制下运行，避免了指令捕获的开销。
        4. 兼容性强：无需修改客户OS，支持运行**任何操作系统**（如Windows、Linux）。
        5. 性能开销大：优先级压缩和二进制代码翻译**增加了指令处理开销**，性能不如半虚拟化。
        6. 复杂度高：Hypervisor需要模拟所有硬件行为，设计和维护成本较高。

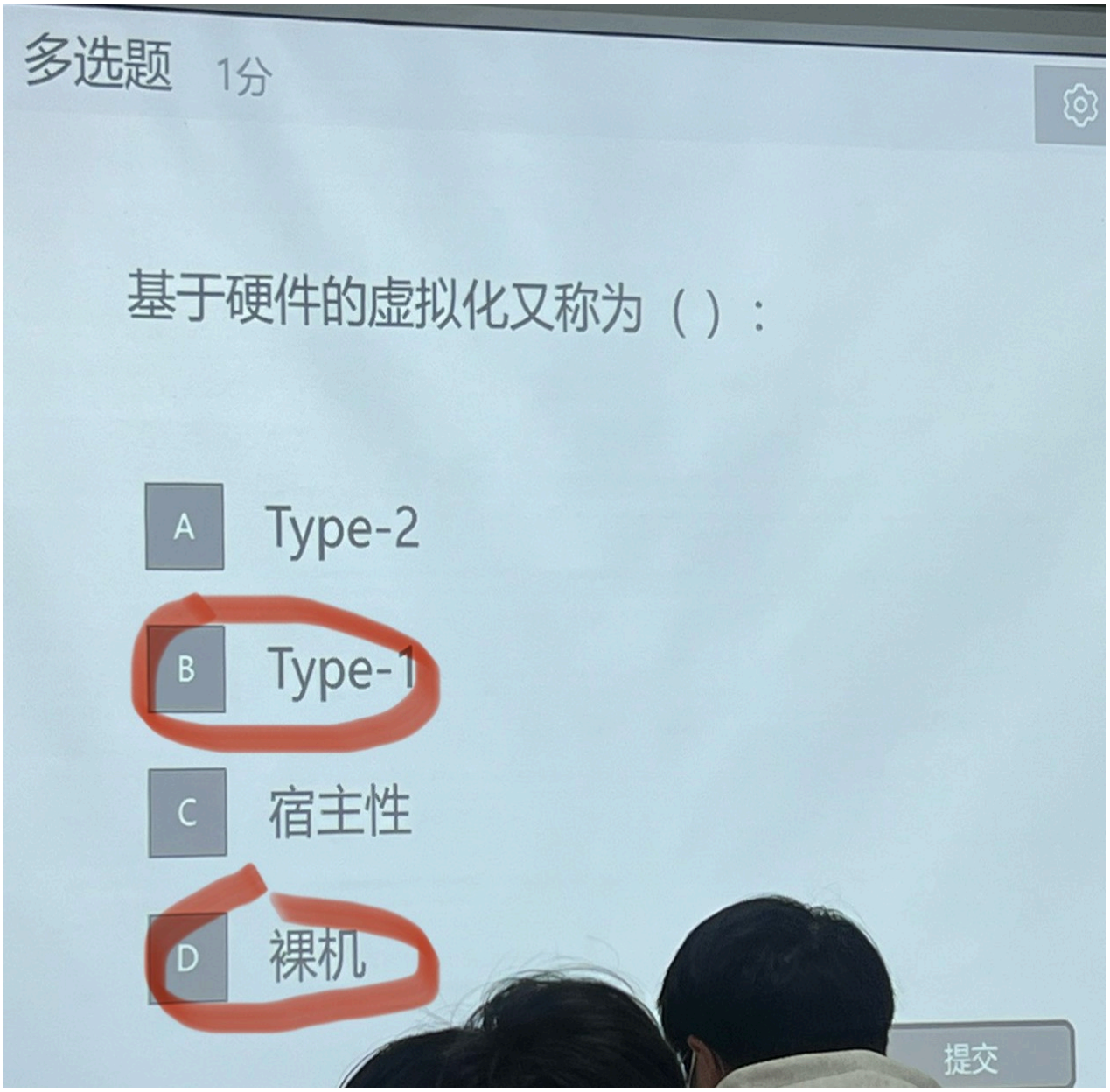
3. 硬件辅助的全虚拟化
    1. VT-x：VT-x通过硬件支持**直接执行**客户操作系统的特权指令和敏感指令，减少Hypervisor的干预，提高虚拟化效率。
4. Para-Virtualization技术
    1. 为减轻虚拟化的复杂度，Xen采用了**Para-Virtualization方法（半虚拟化）**，该方法并**不会**让客户OS认为自己运行在真实独立的硬件资源上，而是**要求对客户OS进行改进**，使它们了解自己运行在虚拟环境下，从而采取不同的接口调用方式，因此适合于开源OS。
    2. **Para-Virtualization**在Hypervisor里**不对设备驱动器进行仿真**，仅对CPU和内存做虚拟，所以又称**半虚拟化或准虚拟化**。
    3. Para-Virtualization的**代表性应用**包括Xen和Microsoft的Hyper-V。
    4. 除了OS协助的虚拟化外，还可以采用**硬件协助的虚拟化技术**(Hardware-Assisted-Virtualization)来实现“低特权态下发出的敏感指令”的处理，该技术以Intel VT-x与AMD-V为代表。
5. Xen调度器
    1. 每个Xen的Domain上可以拥有**一个或以上的虚拟CPU**(vCPU)，，如果没有特别指定，Xen尽可能的把vCPU**均匀的分布**在物理CPU上。
    2. 借助相关指令，用户可以控制某一个Domain拥有的vCPU个数，或者把vCPU固定在**某一个物理CPU(PCPU)**上面。
    3. **尽量保证公平的分配**，并**尽可能避免CPU时钟的浪费**，Domain0也受其制约。
    4. Xen设计了**基于Credit的调度器**，
        1. 资源足够
            1. 如果目前有更多的CPU资源，**所有**DomainU都能获得所需的资源；
        2. 资源不足时，调度器决定各个vCPU获得的PCPU份额。
            1. Xen调度器给每个Domain分配一个**权重**(Weight)，比如权重200的Domain可以获得两倍于权重100的Domain的CPU份额。
            2. Xen调度器还给Domain施加了一个**封顶阈值**(Cap)，即可以使用的最高绝对份额，用百分比表示。
            3. vCPU**运行时消耗Credit**，当一个vCPU用光了自身的Credit，那么只有在比它优先级更高的vCPU运行结束后才能使用物理处理器。

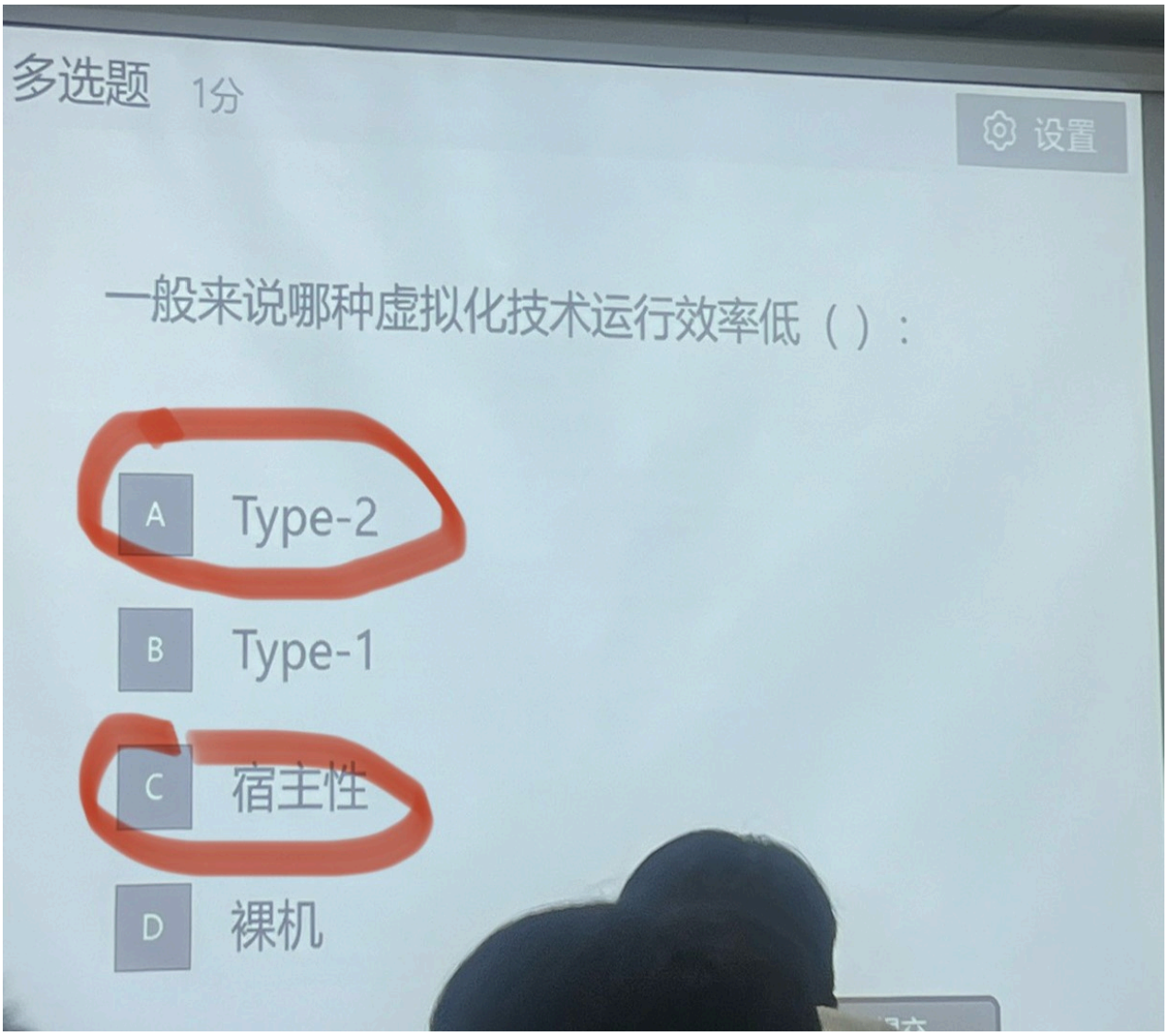
2. KVM虚拟机技术
    1. KVM是Kernel-based Virtual Machine的简称，KVM是一种基于Linux内核的开源虚拟化技术，于2007年被集成到Linux内核主线（2.6.20版本）。
    3. 一般来说，Linux的进程包括两种执行模式：**内核态和用户态**。用户态是应用程序运行的常见默认模式，当需要特权服务时（如硬盘写入）则进入内核态。
    4. 客户机是作为一个**用户态**进程存在的
    5. 优点
        1. 深度整合Linux
        2. 支持硬件虚拟化：硬件虚拟化技术（如VT-x、AMD-V），性能接近原生硬件
        3. 灵活性强：持多种客户操作系统（如Linux、Windows、BSD），广泛应用于x86、ARM等架构

4. 开源生态

3. 其它资源虚拟化技术
    1. I/O虚拟化
        1. 处理器是通过I/O设备来访问外设资源的，**I/O设备的相关虚拟化**称为I/O虚拟化。
        2. I/O虚拟化包括虚拟的芯片组、虚拟的PCI总线、虚拟的系统设备、虚拟的基本I/O设备（如网卡硬盘等）。
        3. I/O虚拟化主要通过VMM截获客户OS对设备的访问请求，然后以**软件模拟**的方式实现真实设备的效果。主要过程包括三个方面：**设备发现、访问截获、设备模拟**。
            1. **设备发现**：可以帮助客户OS**识别虚拟化设备**，从而加载相关驱动程序，客户OS中的驱动会按照接口定义访问这个虚拟设备。
            2. **访问截获**：I/O设备的访问本身是**特权指令**，处于低特权的客户机访问端口I/O会抛出异常，从而陷入到VMM中，
            3. **设备模拟**：交给设备模拟器进行模拟。
        4. 一些I/O虚拟化的例子包括VLAN、VT-d（Intel为I/O直接访问设计的虚拟化技术）等。
        5. 

    2. GPU虚拟化
        1. 通用的GPU虚拟化技术目前有不同的实施方法，如显卡直通(GPU pass-through)和显卡虚拟化(Shared vGPU)。
        2. **显卡直通**：绕过虚拟机管理系统，为每个虚拟机**分配专用的GPU设备**，只有该虚拟机拥有使用GPU的权限。
            1. 优点：虚拟机独占设备**保留了GPU的完整性和独立性**，在性能方面与非虚拟化条件下接近，且可以用来进行通用计算。
            2. 缺点：**不支持实时迁移**等虚拟机高级特性，虚拟机整合率取决于物理GPU的数目。

3. **显卡虚拟化**：将GPU资源进行细粒度切分，并将这些资源**时间片段分配**给不同的虚拟机使用，这其实是采用了半虚拟化I/O的方法，通过vGPU管理模块分时共享一个物理GPU。
    1. 优点：使用灵活，虚拟机整合率高。
    2. 缺点：性能不如显卡直通技术。
4. 

5. 

6. 对Xen来说，管理其它客户作业系统的高权限管理VM称为：

A Domain0

B DomainU

C Parent Partition

D Child Partition

7. 对Hyper-V来说，管理其它客户作业系统的高权限管理VM称为：

A Domain0

B DomainU

C Parent Partition

D Child Partition

8. Xen采用的虚拟化技术属于：

A 全虚拟化技术

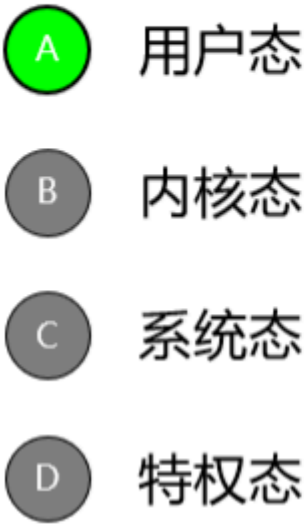
B 半虚拟化技术

C CPU虚拟化技术

D 硬件虚拟化技术

9. Xen调度器进行调度管理的要素包括：

10. KVM技术中，客户机是作为(　)进程运行的。



6.4 虚拟机的管理和调度

1. **虚拟机热迁移**(Live Migration)：又叫**动态迁移**或**实时迁移**，通常是将整个虚拟机的运行状态完全保存下来，同时可以快速恢复到**原有**硬件平台甚至是**不同**硬件平台上。
    1. 这种方式能极大**降低服务宕机时间**和**整体迁移时间**，服务恢复以后，虚拟机仍旧平滑运行，用户不会察觉到任何差异。
    2. 虚拟机迁移的三个步骤：
        1. 传输虚拟机的**基本配置和设备信息**。
        2. 传输虚拟机**内存和关键状态**。

    3. **恢复**目标虚拟机运行。
3. 一般来说，**传输时间**和虚拟机的**工作内存**大小成**正比**，传输过程中原系统服务有可能出现短暂停机。
4. 虚拟机迁移的主要方式：
    1. **预拷贝**(Pre-Copy)：需要经过初始复制、迭代复制、停机复制三个过程。
        1. **初始复制**：将虚拟机**全部内存页面**复制到目标主机。
        2. 迭代复制：针对上一轮过程中源虚拟机**被修改的内存文件**进行复制。
        3. 停机复制：将虚拟机剩余的**少量没有同步内存页面和信息**复制到目标主机。
        4. 理想的Pre-Copy性能应该等同于内存没有被改写的停机拷贝(Stop-and-Copy)。
        5. 最糟糕的情况下，可能存在较长时间的内存脏页面的迭代传送过程，引起极大的系统开销。
    2. **后拷贝**(Post-Copy)：
        1. 源主机先传输包括处理器状态在内的、能够在目标主机上运行的最小数据集。
        2. 然后停止源虚拟机的运行，目的主机借助最小数据集恢复虚拟主机运行。
        3. 最后通过网络从源虚拟主机获取内存页面。
        4. “按需”思想
2. 检查点和热备份
    1. 作为虚拟化的一个重要机制，**检查点**可以对虚拟机进行系统级的状态保存。该机制提供了将虚拟机**恢复到过去状态**的快速而简单的方法，并确保恢复后的虚拟机可以继续运行。
    2. 可以通过**周期性的制造检查点**来捕获虚拟机信息，然后动态的把虚拟机信息复制到另一个备份主机上，即使原始物理机出现故障，系统也可以**借助虚拟机检查点来恢复到最近状态。**
    3. 需要注意的是，对于虚拟机层面的检查点设置会**占用大量的检查点空间**，可能达到数G字节。
        1. 这样大量的数据写入硬盘会不可避免的**降低虚拟机的磁盘性能**，或者造成和宿主机上共存应用对**其他资源**(CPU，I/O带宽等)的**竞争**。
        2. 不建议在虚拟机提供**时间敏感型服务**时使用检查点，也不建议在**存储空间的可用性至关重要时**使用检查点。
3. 虚拟化资源整合
    1. 根据VMware的报告，在传统的“**一虚拟机一负载**”供给模式下，大多数服务器的利用率很低，这导致了虚拟机**资源过度供给**和**资源利用率不高**。
    2. **服务器整合**(Server Consolidation)是借助虚拟化来实现多应用对物理服务器的共享，提升资源利用率。
        1. 每个子系统被封装在虚拟机内，并集中部署。
        2. 服务器整合：**寻找适合的候选虚拟机**，根据应用程序**推断整合方案，跟踪整合后的应用程序性能**，并**调整资源利用方式**。
    3. **服务器整合比**(Consolidation Ratio)指一台物理机上部署的虚拟机的**数量**

    1. 根据IBM公司2010年的调查显示，整合比一般在10:1。
    2. 针对微软公司应用的理想整合比可达18:1。
    3. **整合比不是越高越好**，把过量的虚拟机安置在资源有限的物理机上，会导致系统性能的极大下降，增加运维难度，还会增大能源消耗。
4. 虚拟机蔓延管控
    1. 随着虚拟机数量飞速增长，**回收虚拟化的计算资源**或**清理非正常虚拟机**的工作变得越来越困难，这种不受控制的虚拟机繁殖被称为**虚拟机蔓延**(VM Sprawl)。
    2. 表现形式：
        1. 虚胖虚拟机：虚拟机被**过度配置**，如规划了过高的CPU、内存和存储容量等。
        2. 幽灵虚拟机：虚拟机的创建没有经过符合规程的审核和验证，造成不必要的系统配置，或者由于业务需求需要保留一定数量的冗余虚拟机。这些虚拟机弃用后，导致人们**不知道这些虚拟机创建的原因**，不敢删除回收，任其消耗资源。
        3. 僵尸虚拟机文件：许多虚拟机虽然被停机了，但由于生命周期管理流程的缺陷，**相关虚拟机镜像文件依然保留在硬盘**上。有时出于可靠性的考虑，文件可能还存在许多副本，这些僵尸资源占据着大量服务器存储资源。
    3. 解决虚拟机蔓延的关键在于**优化对虚拟机全生命周期的管控**，可概括为3R：
        1. 降低(Reduce)：减少不必要的虚拟机供给，包括减少缺乏认证的过度供给的虚拟机资源，降低虚拟机资源的浪费，这能够极大的提升投资回报率。
        2. 重用(Reuse)：意味着不止一次的使用虚拟机资源，无论每次使用是否做同样的工作。对于虚拟化基础设施来说，资源的重用能极大提升利用率。
        3. 回收(Recycle)：回收过程包括收集、处理、重生产和重使用。对于回收过程来说，如何准确的确定不再活跃的虚拟机是个非常有挑战的问题。
5. 弹性伸缩
    1. **弹性伸缩**(Elastic Compute Cloud, EC2)是云计算平台的一个关键特性，指根据用户的业务需求和发展策略，系统能**自动调整其所需的计算资源**的能力。
    2. 有了弹性伸缩技术，用户可以在**业务需求高峰时无缝的增加**ECS(Elastic Compute Service)实例，并在业务需求下降时自动减少ECS实例以节约成本
        1. 利用垂直扩展，可以实时升级CPU和内存，实时升级带宽。
        2. 利用水平扩展，可以实时创建新的实例，完成任务后，可以立刻销毁这些实例。

将整个虚拟机的运行状态完全保存下来，以便快速恢复到原有硬件平台甚至是不同硬件平台上的技术，称为：

A 虚拟机复制

B 运行时拷贝

C 虚拟机热迁移

D 平台转移

6.

一般来说，虚拟机热迁移的传输时间和虚拟机的（　）大小成正比

A 工作内存

B 硬盘容量

C 存储数据的大小

D 应用数量

7.

8. 虚拟机热迁移的主要方式包括：

A 全拷贝

B 部分拷贝

C 预拷贝

D 后拷贝

9. 为了能将虚拟机恢复到之前运行的一个状态，应当设置：

A 热迁移备份

B 检查点

C 虚拟机备份

D 物理机备份

服务器整合比是指：

A 可以整合的服务器在所有服务器中的占比。

B 将多台服务器整合提供计算资源。

C 一台物理机上部署多台虚拟机。

D 一台物理机上部署的资源共享的虚拟机的数量。

10.

虚拟机蔓延的表现形式包括：

A 虚胖虚拟机

B 垃圾虚拟机

C 幽灵虚拟机

D 僵尸虚拟机文件

11.

12. 规划了过高的CPU、内存和存储容量的虚拟机称为：

A 虚胖虚拟机

B 垃圾虚拟机

C 幽灵虚拟机

D 僵尸虚拟机文件

13. 有一台虚拟机，连续多天观测发现其一直未被使用，目前该虚拟机创建的原因已经丢失，这种虚拟机可能是：

A 虚胖虚拟机

B 正常虚拟机

C 幽灵虚拟机

D 僵尸虚拟机文件

14. 已经弃用的虚拟机，其对应的保存在硬盘上的、未能及时发现和删除的虚拟机文件称为：

A 虚胖虚拟机

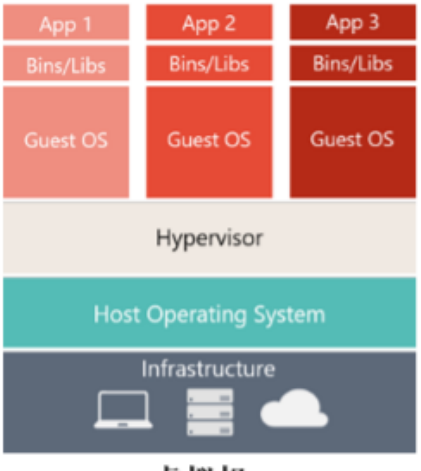
B 垃圾虚拟机

C 幽灵虚拟机

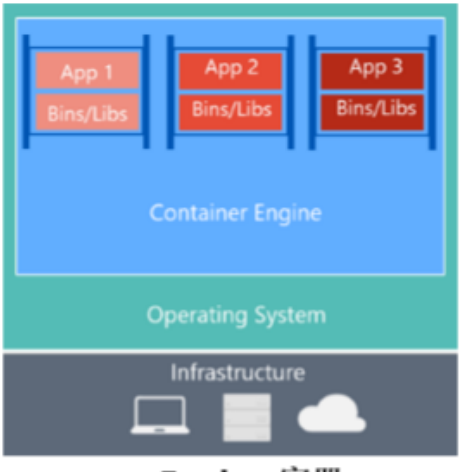
D 僵尸虚拟机文件

6.5 容器技术

1. 操作系统级虚拟化
    1. **虚拟机的缺点**
        1. 尽管实现了隔离，但是每个应用对应一个Guest OS，因此IT资源占用高，导致资源浪费。
        2. 启动和部署速度慢
    2. **操作系统级虚拟化**(Operating System Level Virtualization)：是一种虚拟技术，能够实现多个**相互隔离**的实例**共享一个操作系统内核**。
    3. **容器**(Container)：包含相应应用程序组件的服务实例。
        1. 在一个容器中运行的程序无法看到容器外的程序进程，包括那些直接运行在宿主机(host)上的应用和其它容器中的应用。



虚拟机



Docker 容器

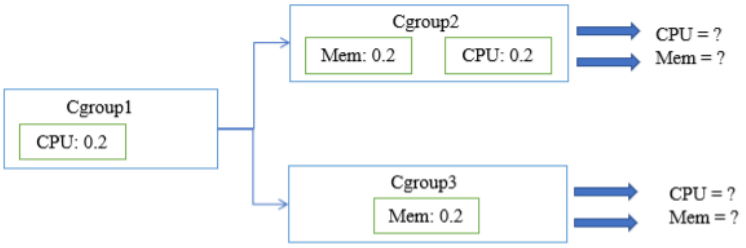
        2. 如何实现容器之间的隔离和限制：1）Namespace；2）Cgroups
            1. 命名空间**Namespace**：（资源的隔离）

1. 使用多个命名空间来隔离容器之间的**进程、网络、文件系统**等资源。每个容器都有自己的命名空间，使得不同容器之间的资源相互**隔离**。两个不同namespace下的进程可以拥有相同的PID。
2. 命名空间可以存在**嵌套关系**。父namespace可以查看所有后代namespace的进程信息，但子namespace无法看到父namespace或兄弟namespace的进程信息
3. 缺点
    1. **隔离不彻底**：在某些情况下，一个进程可以通过文件描述符访问另一个容器的文件系统。
    2. **有些资源和对象不能被Namespace化**，如：容器中的程序调用 settimeofday() 修改了时间，整个宿主机的时间都会被修改
    3. **安全问题**：因为共享宿主机内核，容器中的应用暴露出来的攻击面很大。

2. 控制组Cgroups：（资源的限制）
    1. Linux内核中是为进程设置资源限制的重要手段
    2. 把多个进程放到组里面，对组设置资源使用的权限，实现对进程的控制。
    3. **具体功能**：
        1. **资源控制**，Resource Limiting ： cgroup通过进程组对资源总额进行限制。如：程序使用内存时，要为程序设定可以使用主机的多少内存，也叫作限额。
        2. **优先级分配**，Prioritization：使用硬件的权重值。如：当两个程序都需要进程读取CPU，哪个先哪个后，通过优先级来进行控制。
        3. **资源统计**，Accounting：可以统计硬件资源的用量，如：CPU、内存…使用了多长时间。
        4. **进程控制**，Control：可以对进程组实现挂起/恢复的操作。
    4. 子系统（可以限制哪些资源）
        1. cpu 子系统，主要限制进程的 cpu 使用率。
        2. cpuacct 子系统，可以统计 cgroups 中的进程的 cpu 使用情况。
        3. cpuset 子系统，可以为 cgroups 中的进程分配单独的 cpu 核或者内存。
        4. memory 子系统，可以限制进程的 memory 使用量。
        5. blkio 子系统，可以限制进程对存储设备的读取。
        6. devices 子系统，可以控制进程能够访问某些设备。
    5. 步骤：
        1. 创建cgroup：目录 /sys/fs/cgroup
        2. 设置cgroup配额（将cpu/内存等各种子系统添加到该cgroup中）
        3. 将进程添加为cgroup的任务
    6. cgroup ： Cgroups中的资源控制都以cgroup为单位实现的。cgroup表示按照某种资源控制标准划分而成的任务组，**包含一个或多个子系**

统。

7. **层级关系**（hierarchy）：hierarchy由一系列cgroup以一个**树状结构**排列而成，每个hierarchy通过绑定对应的subsystem进行资源调度。
    1. 目的是简化配置
        1. 一群Cgroup组成的树状结构
        2. 每个Cgroup可以看成一个节点，子节点可以继承父节点属性



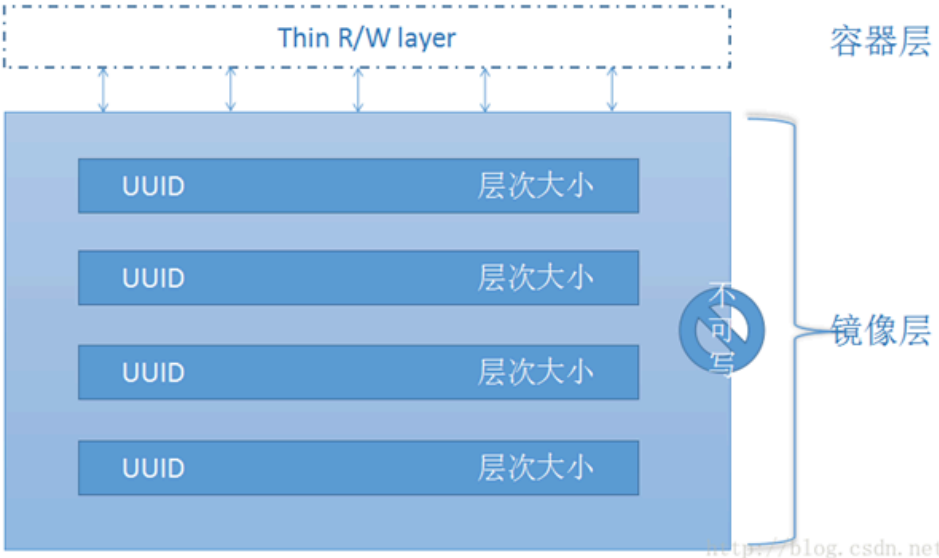
        3.
    2. 规则：
        1. 同一个hierarchy能够附加一个或多个subsystem。
        2. 一个subsystem只能附加到一个hierarchy上。
        3. 一个task不能存在于同一个hierarchy的不同cgroup中，可以存在于不同Hierarchy的多个Cgroup中。
        4. 任何一个task(Linux中的进程)fork自己创建一个子task(子进程)时，子task会自动的继承父task cgroup的关系。

3. 开发环境和部署环境不一致的问题；对于一些应用来说，环境配置十分麻烦；无法跨平台使用：Windows -- Linux
    1. 解决方案：将**应用程序**和**所有依赖**项、库、配置文件和运行时环境合二为一，打包成镜像。
    2. 容器往往容纳了该程序运行所需要的全部文件，它可能包含自己的库、自己的/boot目录、/usr目录、/home目录等。然而，如果需要的话，运行中的容器甚至可能仅包含一个文件，比如运行一个不依赖任何文件的二进制程序。

4. 一些重要的操作系统级虚拟化技术有：
    1. OpenVZ：基于Linux内核和操作系统，允许物理服务器运行多个相互隔离的操作系统实例（容器），以实现更高的服务器利用率，并防止应用间的冲突。这些容器又称为专用虚拟服务器(Virtual Private Server, VPS)或虚拟环境(Virtual Environment, VE)。
    2. FreeBSD Jail：可以把一个基于FreeBSD的系统分割成若干独立的子系统，称为Jail。每一个Jail都是一个虚拟化计算环境，运行在宿主机上并拥有文件、进程、用户和超级用户账户。
    3. Solaris Container：为x86和SPARC系统设计的操作系统层面虚拟化技术，Solaris Zone是单一操作系统实例中的一个完全隔离的虚拟服务器，一个Solaris容器则由Solaris Zone提供的**边界隔离**以及相应的**系统资源控制**组成。
    4. AIX Workload Partition(WPAR)：是构建于IBM公司的AIX操作系统上的虚拟化计算环境，WPAR提供应用环境的隔离以最小化WPAR之间的应用干扰。尽管

WPAR都使用相同的操作系统，但它们之间的资源交互是有限的。

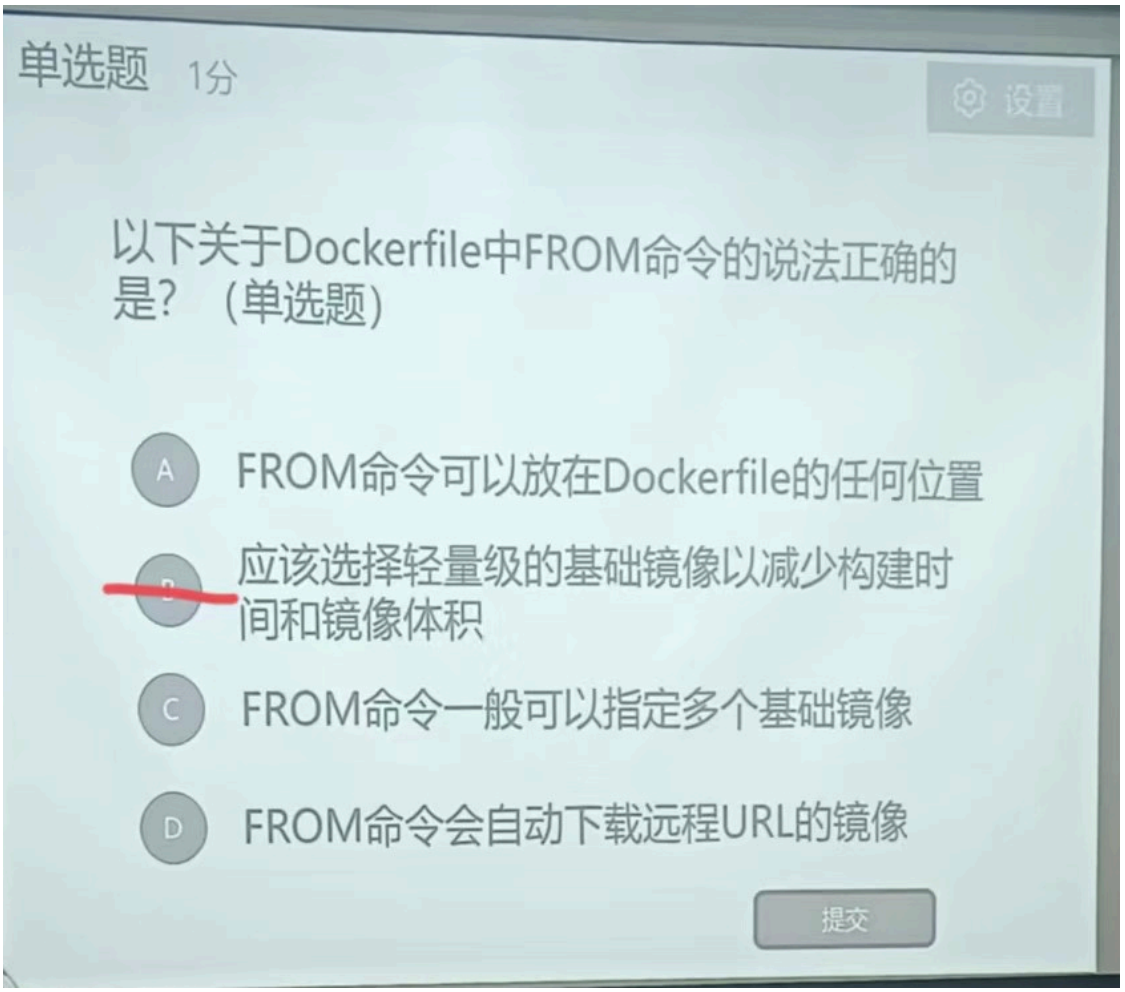
2. Docker
    1. 容器
        1. Docker本身是一个开源项目，2013年诞生于dotCloud公司。
        2. 该项目加入了Linux基金会，并遵从Apache 2.0协议。
        3. 用户操作Docker的容器就像管理一个无比快速且轻量级的虚拟机一样简单。
        4. Docker希望将用户的大型应用拆分成若干不同的微服务，每个微服务提供原子的功能，互相连接；每个微服务有相同镜像相同配置的一组容器组成，一个或多个服务组成了一个完整的容器“应用”。
        5. Docker是一个工具，能帮助我们快速的创建、分享和运行软件。
        6. Docker允许我们将一个软件和其依赖(运行环境)打包成一个单独的库，更利于移植可运行的软件。
    2. 镜像
        1. **Docker镜像**是一个构建容器的**只读模板**，提供了容器应用打包的标准格式，容器即为一个通过Docker镜像创建的运行时实例。
        2. Docker 镜像包含Dockerfile、依赖、程序代码和重要配置参数（环境变量、网络配置）。
        3. 一个镜像可以同时生成多个运行的容器实例。
        4. 尽量使用被人制作好的镜像文件，而不是从头开始制作。
        5. 每个Docker镜像的文件系统由一系列**只读层**(read-only layer)组成，只读层自底向上逐层堆叠，构成了容器所需要的基础文件系统。

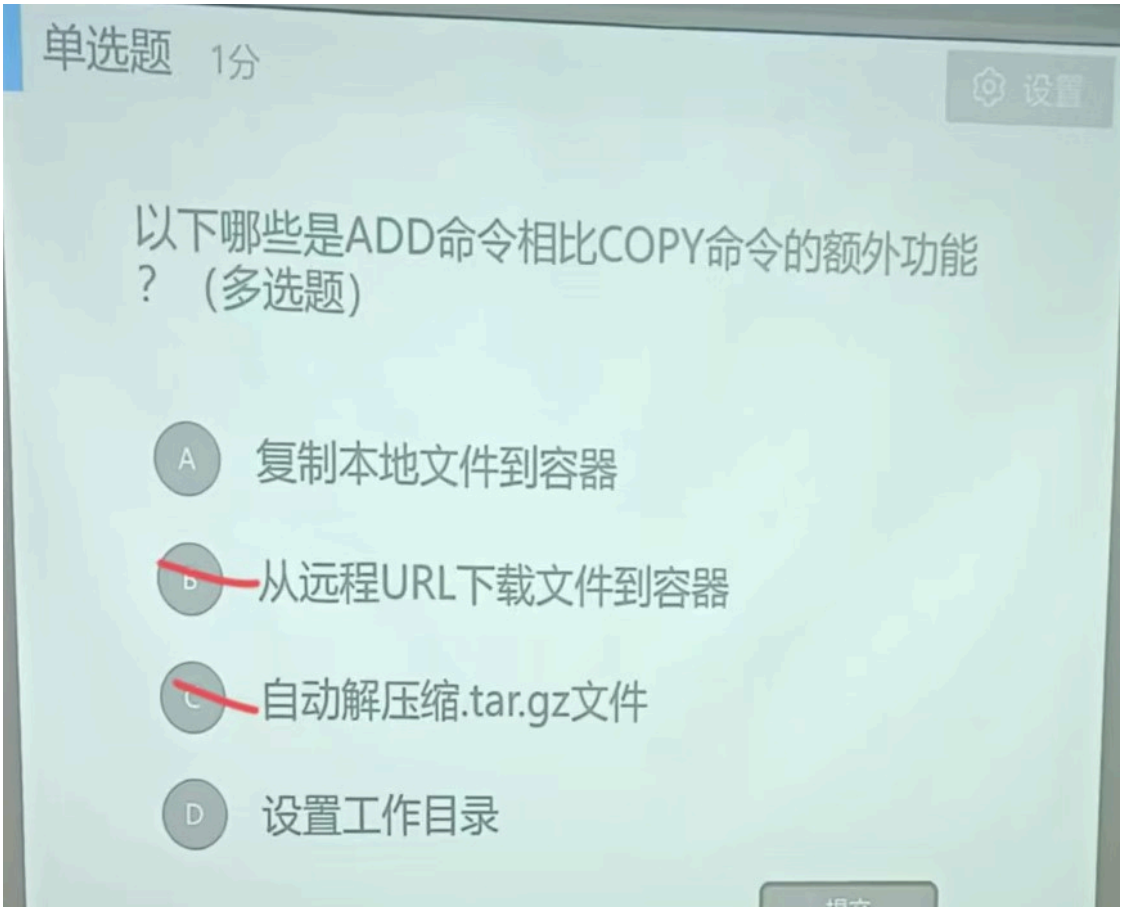


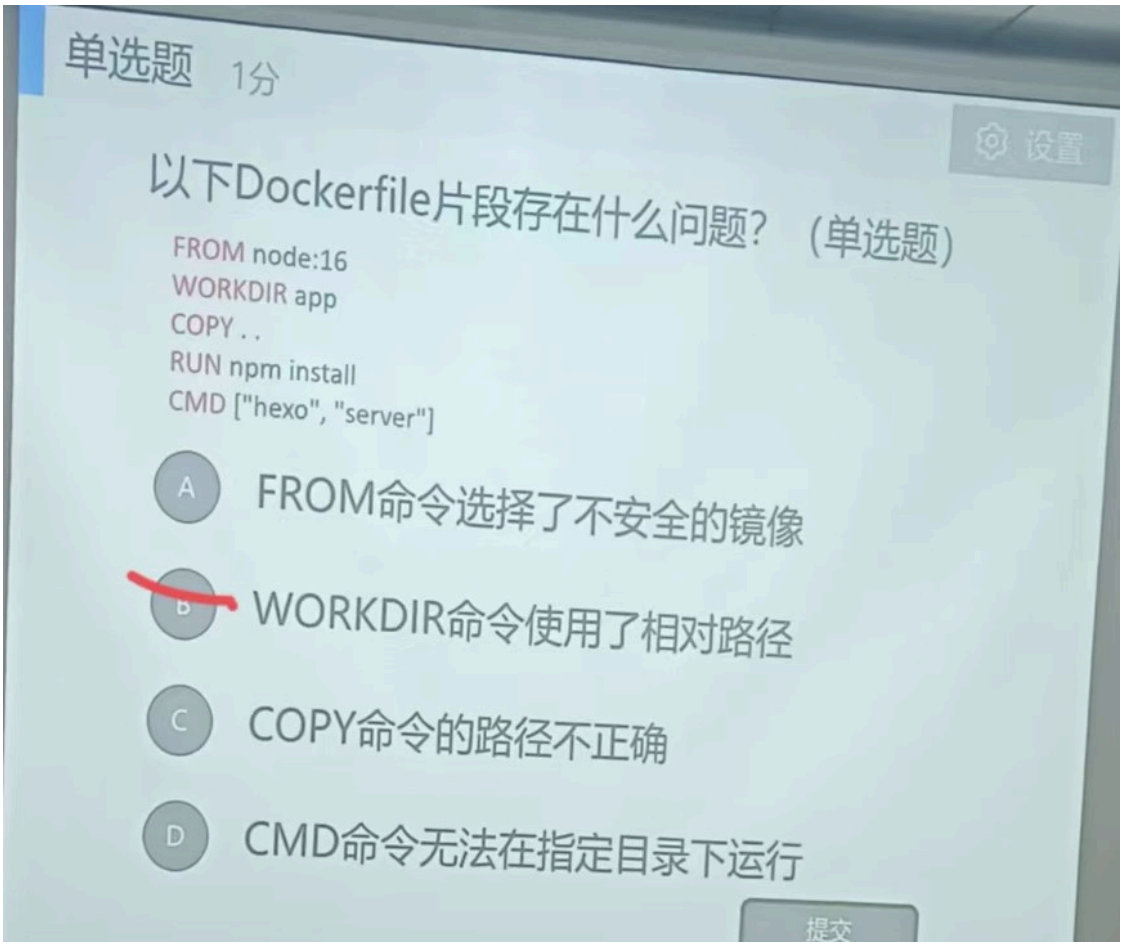
        6. 优点：使得镜像的复用、定制变的更为容易实现。
    3. 哈希函数：
        1. 如果两个输入串的hash函数的值一样，则称这**两个串是一个碰撞**(Collision)。既然是把任意长度的字符串变成固定长度的字符串，所以必有一个输出串对应无穷多个输入串，碰撞是**必然存在的**。
    4. 官方镜像： Hello World
        1. 镜像结构
            1. 保存镜像文件：docker save hello-world > hello.tar

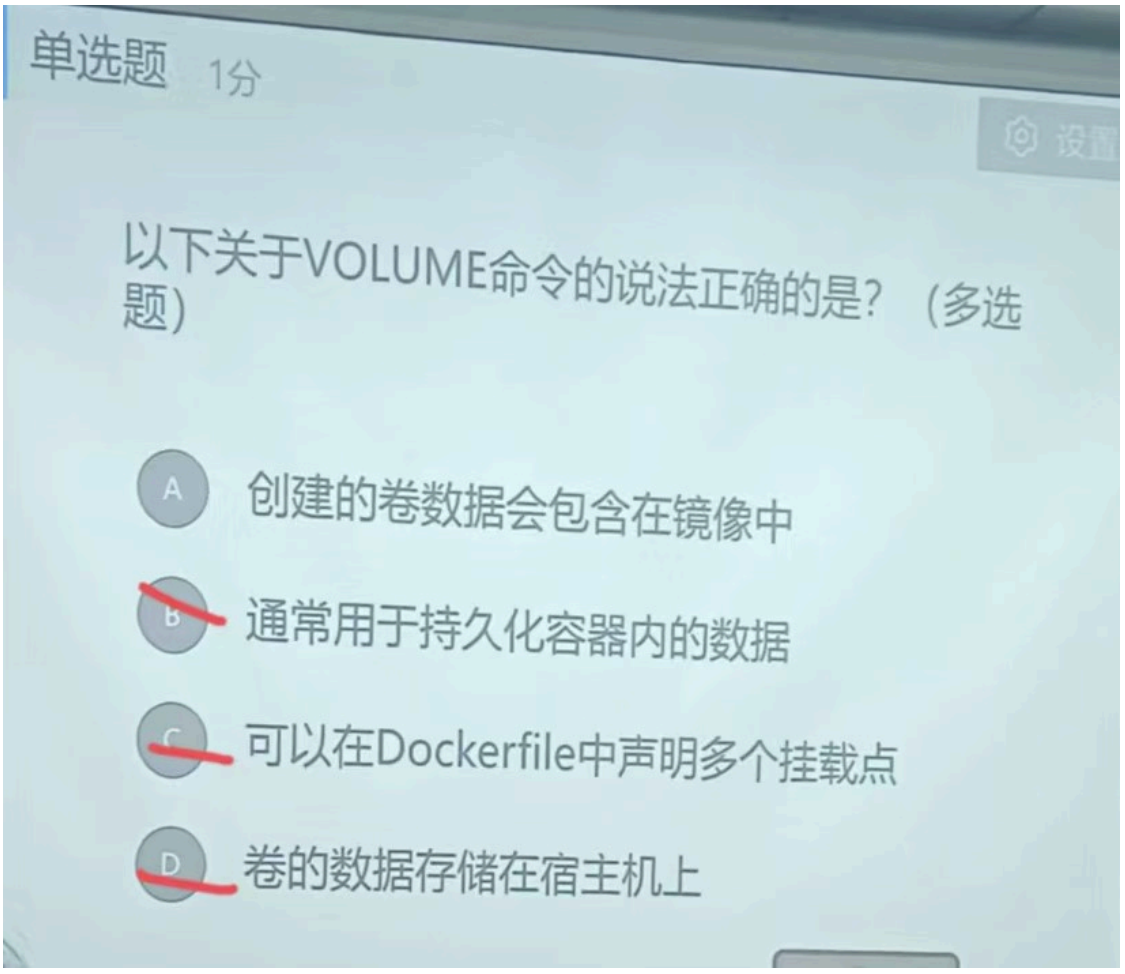
    2. manifest.json：包含了关于镜像层和配置的元数据
        1. Config: 指向包含镜像配置的xxx.JSON文件，包括创建容器时需要的环境变量、运行命令、暴漏的端口等信息
        2. RepoTags: 表示镜像的名称和标签
        3. Layers: xxx文件夹，是镜像层文件
    3. xxx.json
    4. xxx文件夹
    5. Repositories：是 Docker 镜像仓库的元数据文件之一，用于存储镜像的标签和对应的哈希值。在 Docker 镜像中，每个镜像可以有多个标签，而这些标签与具体的哈希值相对应。
5. Dockerfile中包含一系列的指令用来创建Docker Image
6. 建立自己的镜像和容器
    1. 在Docker中使用Ubuntu镜像和Apache镜像构建新的镜像
        1. 一个包含创建Docker Image指令的文件：Dockerfile
        2. 建新镜像：`docker build -t myubuntu_apache .`
        3. 运行镜像：`docker run -d --name mycontainer -p 80:80 myubuntu_apache`
    2. 在Docker中使用Flask镜像和自己编写的应用程序
        1. 一个包含创建Docker Image指令的文件: Dockerfile
        2. 创建自己的应用程序：`app.py`
        3. 建新镜像：`docker build -t my-flask-image .`
        4. 运行镜像：`docker run -d --name my-flask-container -p 88:88 my-flask-image`
7. Dockerfile 命令
    1. FROM：
        1. 指定基础镜像。
        2. 注意事项：应该尽可能选择轻量级的基础镜像，并确保基础镜像的可靠性和安全性。一般只能指定一次基础镜像。
    2. WORKDIR：
        1. 设置工作目录。
        2. 注意事项：确保使用**绝对路径**，并避免在一个 Dockerfile 中频繁地更改工作目录。
    3. COPY 和 ADD：
        1. 将本地的文件或者文件夹复制到容器指定的路径中。
        2. 注意事项：
            1. COPY 只能复制**本地**文件或目录到容器中，而 ADD 还支持从**远程** URL 下载内容并复制文件到容器的文件系统，还可以压缩包解压后复制到指定位置。
            2. 使用ADD构建的镜像比COPY**体积要大**，尽量使用COPY，除非需要自动解压缩或远程文件复制功能。

3. 总结区别：
    1. COPY：简单复制，仅限本地文件/目录。
    2. ADD：支持本地复制 + 远程URL下载 + 自动解压缩。
4. EXPOSE：
    1. 声明容器运行时**监听的端口**。
    2. 注意事项：EXPOSE 命令仅用于声明端口，实际上并不会启动监听。运行时需通过 -p 或 -P 参数来映射端口。例如：`docker run -d -p 80:80 nginx-web`
5. VOLUME：
    1. 说明：声明容器内的挂载点。
    2. 注意事项：VOLUME 命令创建一个具有指定名称的卷，并且通常用于持久化容器内的数据。
6. RUN：
    1. 在创建镜像时执行命令，用于安装软件包、运行脚本等操作，先于CMD、ENTRYPOINT，只在创建镜像时执行一次。
    2. 注意事项：尽量将**多个命令合并为单个 RUN 指令**，使用 && 运算符来将多个命令连接在一起，避免创建不必要的镜像层。
    3.
```
FROM ubuntu

RUN apt-get update && apt-get install -y apache2

COPY ./index.html /var/www/html/

EXPOSE 80

CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```
7. CMD 和 ENTRYPOINT：
    1. 定义了容器启动后执行的命令或程序。
    2. CMD 用于指定默认的命令和参数，Dockerfile只允许使用一次CMD命令；如果使用多个，只有最后一个生效；一般作为文件中最后一个命令。**Docker run命令可以覆盖CMD指定的默认命令**。
    3. ENTRYPOINT 用于指定容器启动时执行的可执行文件。docker run 时把容器名之后的所有内容当成参数传递给其指定的命令（不会对命令覆盖）。

8. 
单选题 1分
设置
以下关于Dockerfile中FROM命令的说法正确的是？（单选题）
A FROM命令可以放在Dockerfile的任何位置
B 应该选择轻量级的基础镜像以减少构建时间和镜像体积
C FROM命令一般可以指定多个基础镜像
D FROM命令会自动下载远程URL的镜像
提交

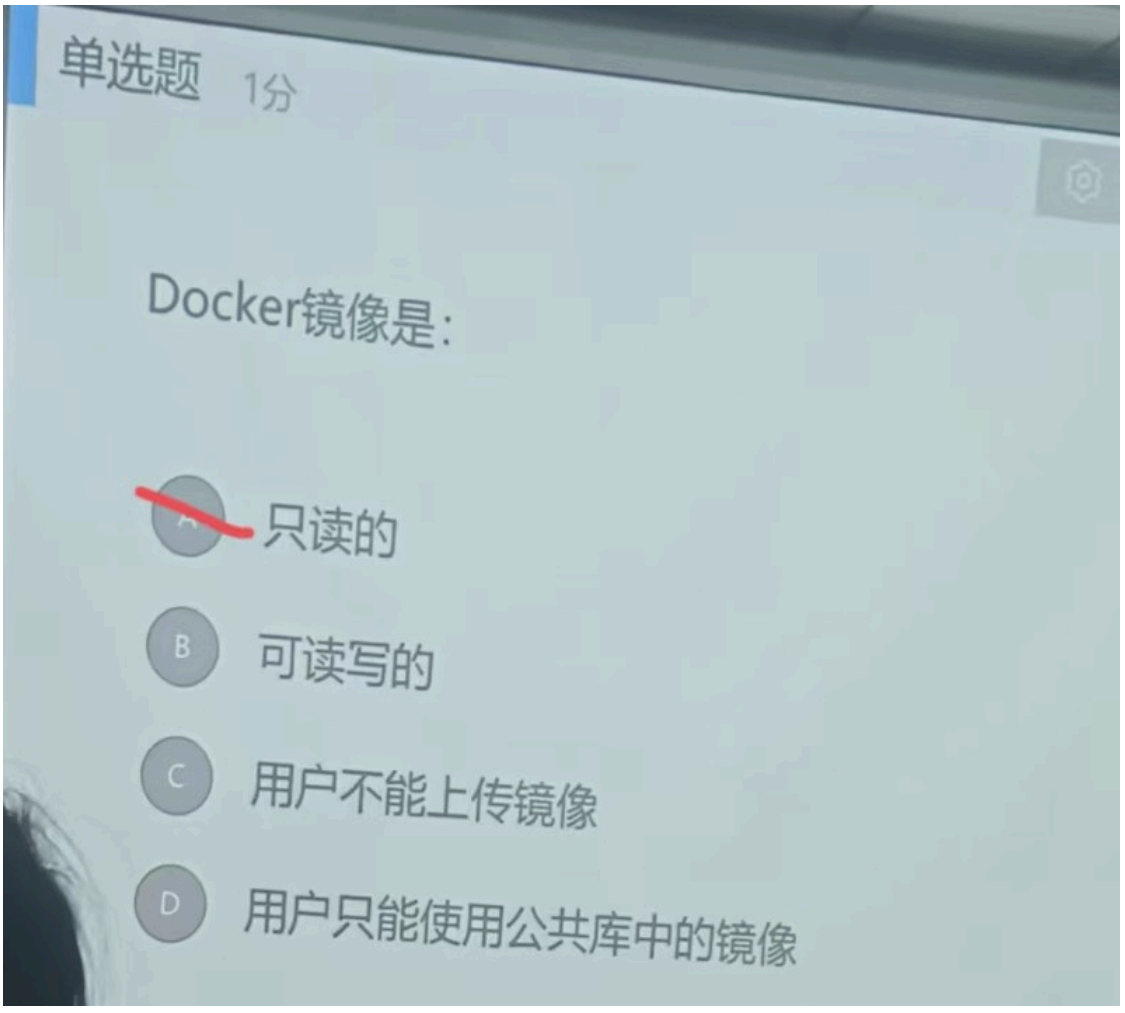
9. 
单选题 1分
设置
以下哪些是ADD命令相比COPY命令的额外功能？（多选题）
A 复制本地文件到容器
B 从远程URL下载文件到容器
C 自动解压缩.tar.gz文件
D 设置工作目录

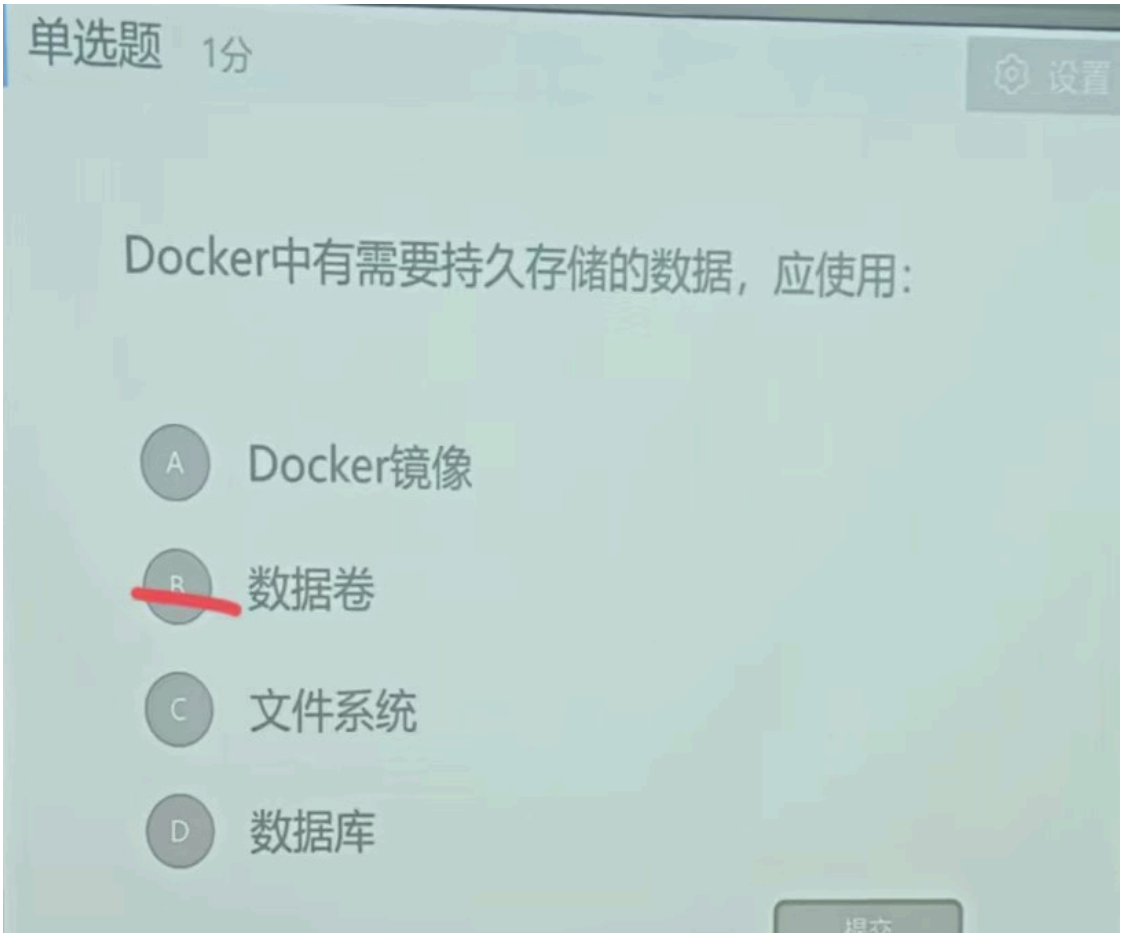
10. 

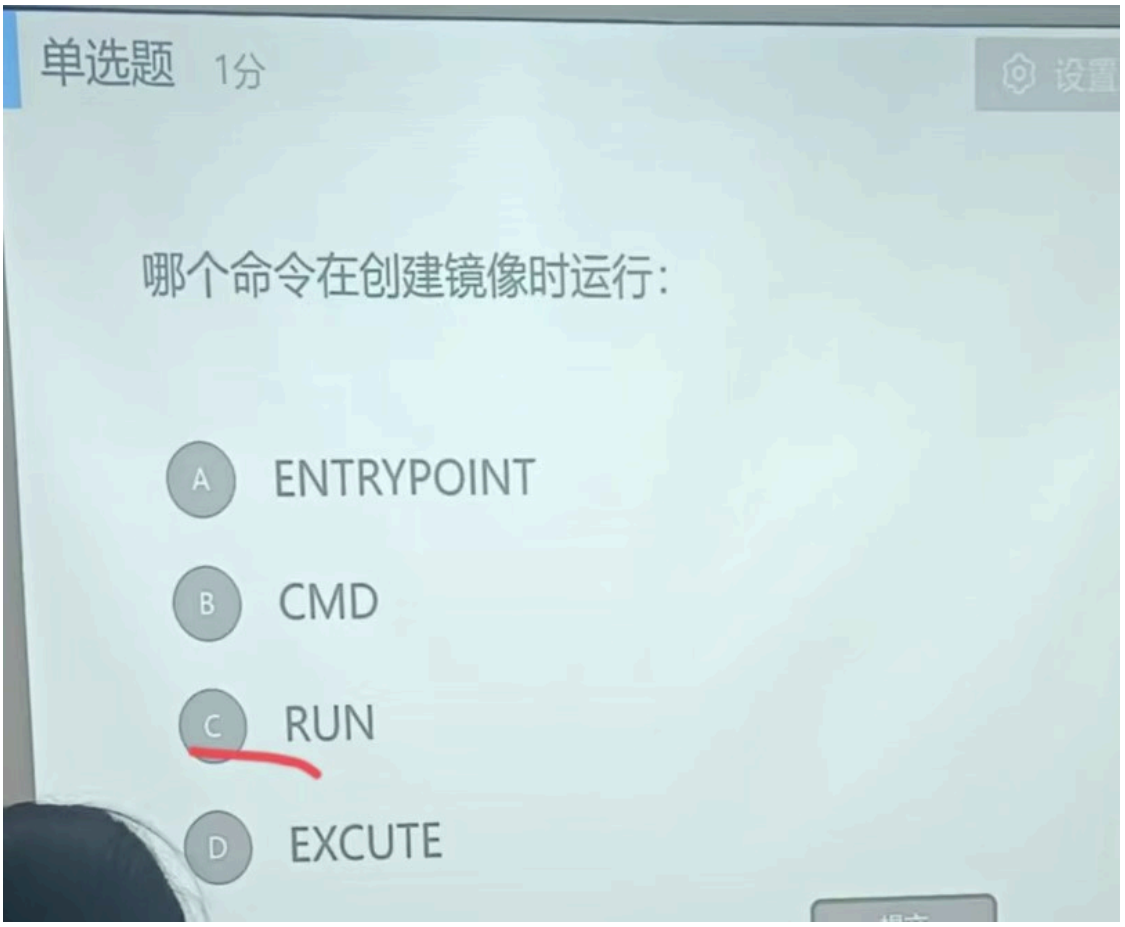
11. 

8. 镜像
    1. 使用 docker images 来列出本地主机上的镜像。
        1. REPOSITORY：表示镜像的仓库源

    2. TAG：镜像的标签
    3. IMAGE ID：镜像ID
    4. CREATED：镜像创建时间
    5. SIZE：镜像大小
2. 获取一个新的镜像
    1. 在本地主机上使用一个不存在的镜像时 Docker 就会自动下载这个镜像。如果想预先下载这个镜像，可以使用 docker pull 命令来下载它。
    2. 在docker Desktop里获取新镜像
3. 查找镜像
    1. 从 Docker Hub 网站来搜索镜像，Docker Hub 网址为：`https://hub.docker.com/`
    2. 也可以使用 docker search 命令来搜索镜像。比如需要一个 httpd 的镜像来作为 web 服务。可以通过 docker search 命令搜索 httpd 来寻找适合的镜像。
        1. NAME: 镜像仓库源的名称
        2. DESCRIPTION: 镜像的描述
        3. OFFICIAL: 是否 docker 官方发布
        4. AUTOMATED: 自动构建。
4. 删除镜像
    1. 镜像删除使用 docker rmi 命令
5. 更新镜像
    1. 更新镜像之前，使用镜像来创建一个容器。
    2. 在运行的容器内使用 apt-get update 命令进行更新。
    3. 在完成操作之后，输入 exit 命令来退出这个容器。
    4. 此时 ID 为 `e218edb10161` 的容器，是按需求更改的容器。通过命令 docker commit 来提交容器副本。
    5.
```
runoob@runoob:~$ docker commit -m="has update" -a="runoob" e218edb10161 runoob/ubuntu:v2
sha256:70bf1840fd7c0d2d8ef0a42a817eb29f854c1af8f7c59fc03ac7bdee9545aff8
```
    6. `e218edb10161`：容器 ID
    7. `runoob/ubuntu:v2`：指定要创建的目标镜像名

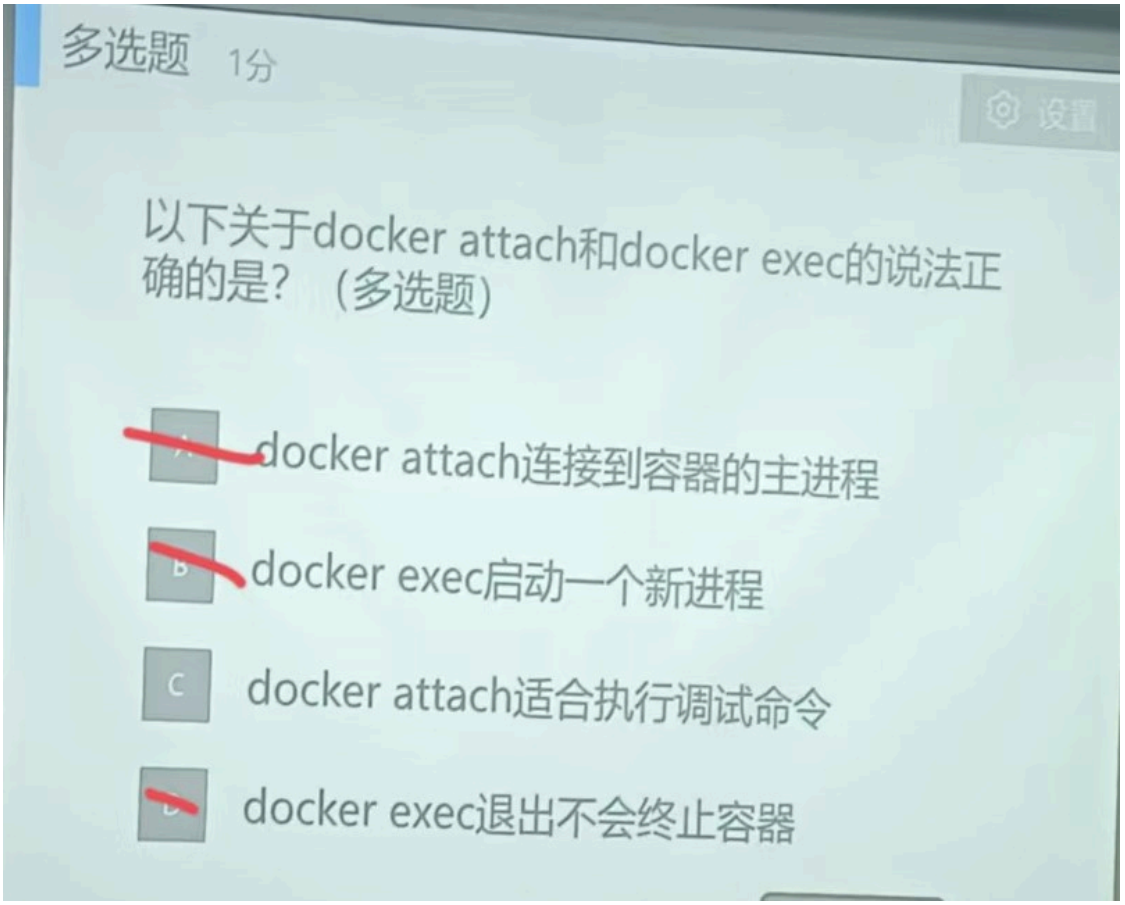
6. 
单选题 1分
Docker镜像是:
A 只读的
B 可读写的
C 用户不能上传镜像
D 用户只能使用公共库中的镜像

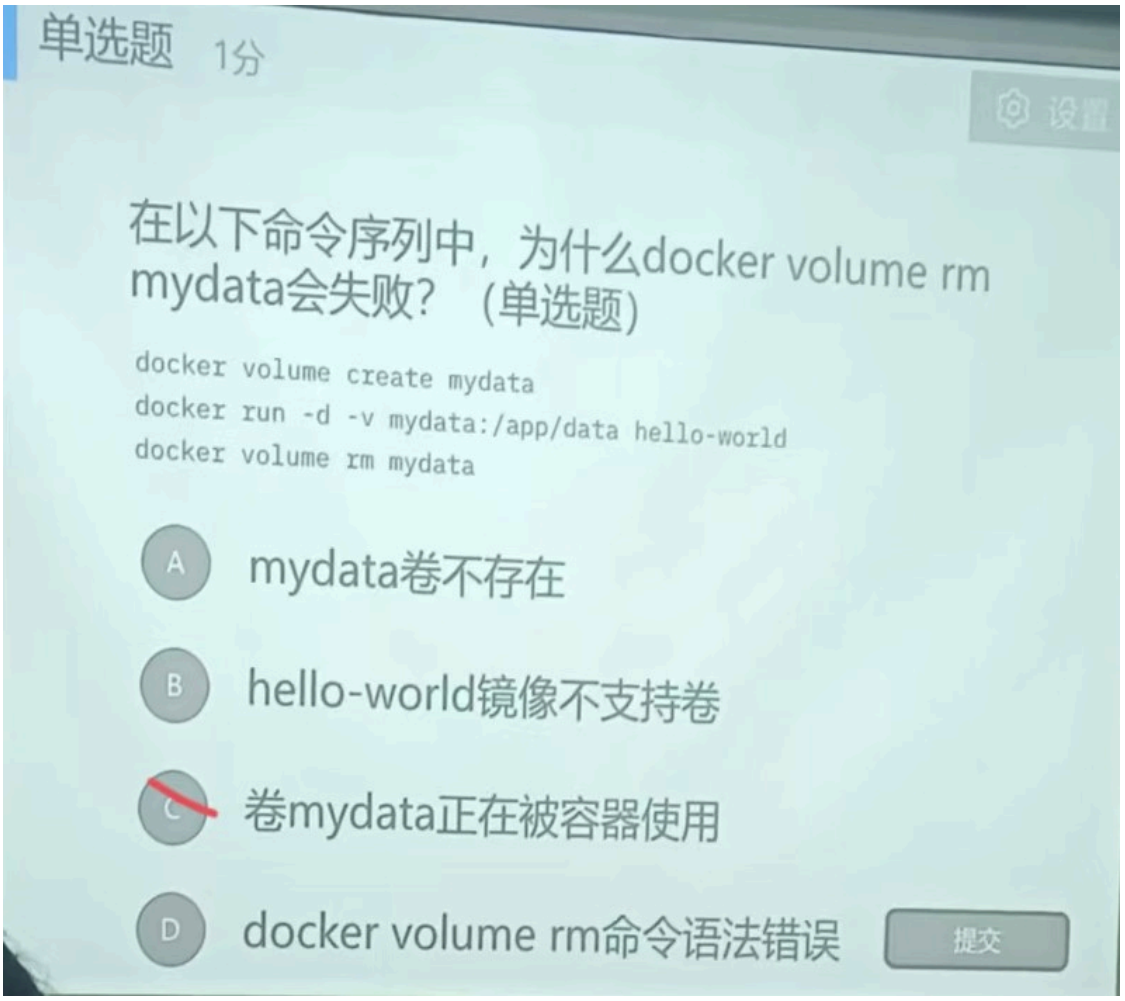
7. 
单选题 1分
设置
Docker中有需要持久存储的数据，应使用:
A Docker镜像
B 数据卷
C 文件系统
D 数据库

8. 

9. 

9. 容器
    1. 容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等 。

2. 当启动一个容器时，Docker会在镜像栈的顶部增加一个新的、薄的读写层，这一层即“容器层”。当前运行容器的所有操作（比如写新文件、修改现有文件、删除文件）都写到这一读写层中。当这一容器被删除时，其读写层也被删除，而底层的镜像保持原状，而重新利用该镜像创建的应用也不保留此前的更改。这种只读层结合顶部读写层的组合被称为Union File System。在这样的架构下，多个容器可以安全的共享一个底层镜像。
3. 启动已停止运行的容器
    1. 使用 `docker ps -a` 查看所有的容器
    2. 使用 `docker start` 启动一个已停止的容器
4. 后台运行
    1. 在大部分的场景下， docker 的服务是在后台运行的，可以过 -d 使容器在后台运行。
5. 进入容器
    1. 在使用 -d 参数时，容器启动后会进入后台。此时想要进入容器，可以通过以下指令进入： `docker attach` 和 `docker exec`
    2. `docker attach` 连接的是容器的主进程，可以查看输出或与进程交互。
    3. exit、Ctrl+C会终止容器，推荐使用 `docker exec` 命令，因为此命令会退出容器终端但不会导致容器的停止。
6. 导出和导入容器
    1. 使用 `docker export` 命令导出本地某个容器
    2. 可以使用 `docker import` 从容器快照文件中再导入为镜像
7. 删除容器
    1. 使用docker rm 命令删除容器
8. 

10. 数据卷
    1. 数据卷是宿主机中的一个目录或文件
    2. 特性：
        1. 独立生存周期
        2. 实时同步：单个容器挂载多个数据卷当容器目录和数据卷目录绑定后，对方的修改会立即同步
        3. 多容器共享：一个数据卷可以被多个容器同时挂载
        4. 容器挂载多个数据卷：一个容器也可以被挂载多个数据卷
        5. 被使用时无法被删除
    3. 数据卷作用
        1. 容器数据持久化
        2. 外部机器和容器间接通信
        3. 容器之间数据交换
    4. 创建名为 `mydata` 的数据卷：`docker volume create mydata`
    5. 运行 `CentOS` 容器，并将 `mydata` 数据卷挂载到容器的 /data 目录：`docker run -d -v mydata:/data --name mycontainer centos`
    6. 

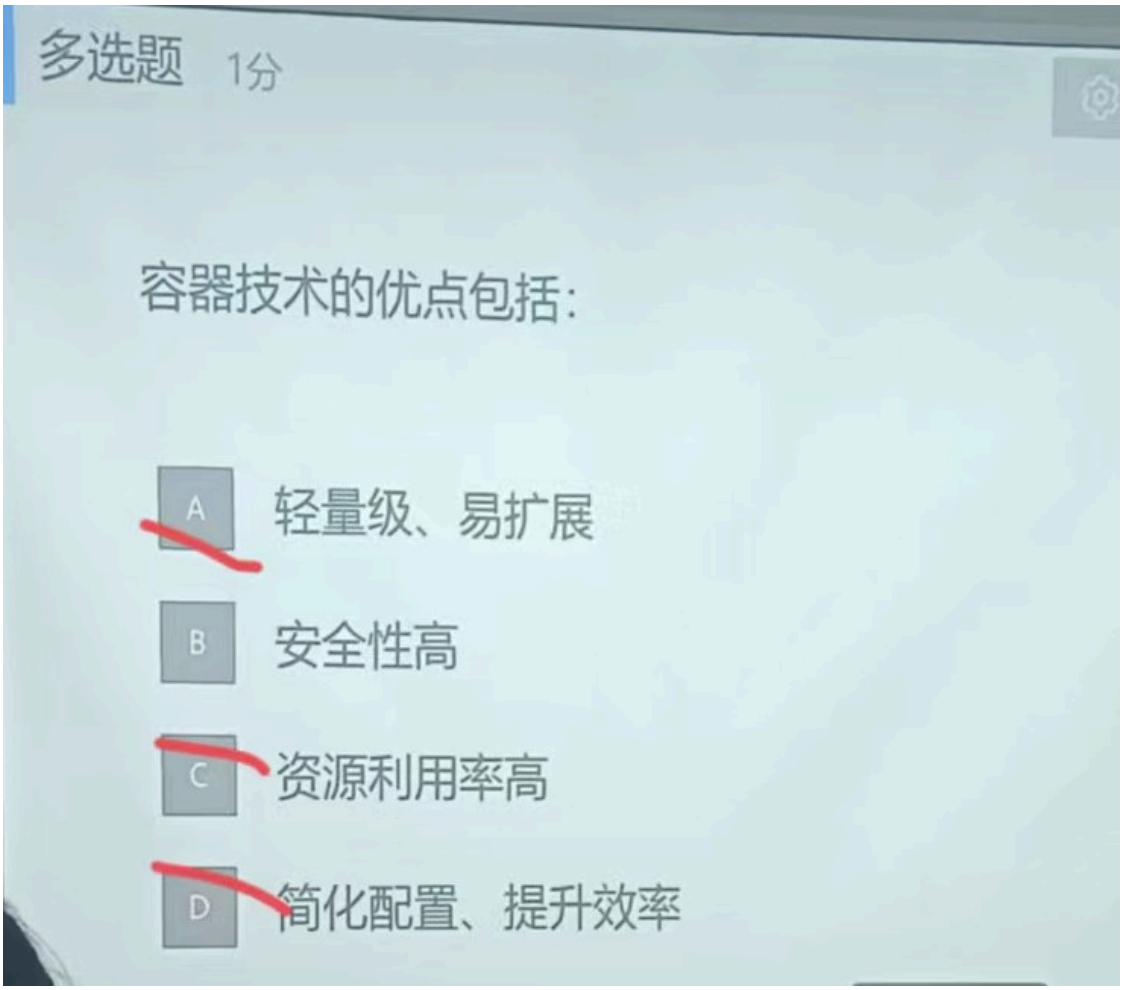
11. Docker Registry
    1. 仓库（Repository）是集中存放镜像文件的场所。
    2. 仓库(Repository)和仓库注册服务器（Registry）是有区别的。仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。

3. 仓库分为公开仓库（Public）和私有仓库（Private）两种形式。
4. 最大的公开仓库是DockerHub(https://hub.docker.com/)，存放了数量庞大的镜像供用户下载。国内的公开仓库包括阿里云、网易云等

12. 容器与传统虚拟机
    1. 容器技术不是虚拟化的替代方案，它还不能取代传统的服务器虚拟化技术
    2.

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

    3. 容器与虚拟机（VM）两者是可以共存的
        1. 全虚拟化技术的多数应用场景是面向高度复杂服务的云基础设施，为其提供计算、存储、迁移等服务。
        2. 隔离用户，隔离应用
    4. 优点
        1. 轻量级、易扩展：虚拟机自身是一个完备系统，拥有虚拟化的硬件和特定资源，如果每个VM有2 GB容量，则10个虚拟机就需要20 GB；若采用容器，因为共享其操作系统内核，因此并不会占据20 GB空间。
        2. 资源利用率高：虚拟机需要借助虚拟化软件层模拟硬件行为，而容器则直接运行在主机操作系统上。其启动时间也短。
        3. 简化配置、提升效率：降低了硬件资源和应用环境的耦合度，并且可以给开发者提供理想的开发环境，提升开发效率。
    5. 缺点
        1. 安全性：容器极度依赖其主机操作系统，所以任何针对主机操作系统的攻击都会造成其安全问题。同时，主机操作系统能够看到容器中运行的一切资源。
        2. 隔离型相比虚拟机差：如果某个应用运行时导致内核崩溃，所有的容器都会崩溃
    6. Docker的轻量级特性使其成为未来云计算的重要拓展方向之一——边缘计算(Edge Computing)的重要使能技术
        1. 基于**地理分布**的边缘设备的计算方式，需要一种灵活可扩展的平台来实现应用和服务的部署。
        2. 该平台应当能够部署适合小型边缘设备的**轻量级可重用的服务**，并且**不依赖于异构硬件架构**。
        3. 该平台还应当能够有效的**与云端进行交互**，从而实现分布式大数据的边缘计算和云端存储。
        4. 这一平台还应**易于安装、配置、管理、升级、以及调试**。

7. 


```
$ docker image pull hello-world
$ docker image ls
$ docker container run hello-world

Hello from Docker!
This message shows that your installation appears to be working
correctly. ......
Docker run hello-world
```