**云计算知识点**

1. 云计算的特征
   1. 弹性伸缩
   2. 快速部署
   3. 资源抽象
   4. 按用量收费
   5. 宽带访问
2. 云计算按部署模式分类：公有云/私有云/社区云/混合云
3. 云计算按服务类型分类：IaaS/PaaS/SaaS
4. 基础设施即服务IaaS
5. 平台即服务PaaS
6. 软件即服务SaaS
7. CAP理论

一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。

* 1. 一致性：即更新操作成功并返回客户端后，所有节点在同一时间的数据完全一致。
  2. 可用性：即服务一直可用，而且是在正常的响应时间内。
  3. 分区容错性：指分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。

1. BASE理论

即使无法做到强一致性（CAP的一致性就是强一致性），但应用可以采用适合的方式达到最终一致性。

BASE是指基本可用（Basically Available）、软状态（Soft State）、最终一致性（Eventual Consistency）。

* 1. 基本可用：指分布式系统在出现故障时，允许损失部分可用性，即保证核心可用。
  2. 软状态：指允许系统存在中间状态，而该中间状态不会影响系统整体可用性。
  3. 最终一致性：指系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。

1. 最终一致性
   1. 强一致性（即时一致性）
   2. 弱一致性：最终一致性是弱一致性的一种特例。
2. 虚拟机技术基本概念：Host、Guest、VMM
   1. Host：主机、宿主。多机系统中起主要作用和控制作用的计算机。它还为别的计算机准备程序，并对其他系统所用的程序进行编译、编辑、连接和测试。
   2. Guest：指让给客人访问电脑系统的账户。
   3. VMM：虚拟化管理器，对系统资源有完全的控制能力和管理权限，包括资源的分配，监控和回收。
3. 虚拟机技术分类：全虚拟化/半虚拟化/硬件辅助虚拟化
   1. 全虚拟化：全虚拟化提供底层物理系统的全部抽象化，且创建一个新的虚拟系统，客户机操作系统可以在里面运行。不需要对客户机操作系统或者应用程序进行修改（客户机操作系统或者应用程序像往常一样运行，意识不到虚拟环境的存在）。
   2. 半虚拟化：半虚拟化需要对运行在虚拟机上的客户机操作系统进行修改（这些客户机操作系统会意识到它们运行在虚拟环境里）并提供相近的性能。但半虚拟化的性能要比全虚拟化更优越，半虚拟化系统性能可以接近在裸机上的性能。
   3. 硬件辅助虚拟化：不是独立的虚拟化技术，结合到全/半虚拟化技术中。通过对部分全虚拟化和半虚拟化使用到的软件技术进行硬件化来提高性能。
   4. Xen支持半虚拟化和完全虚拟化。
   5. VMware、KVM支持完全虚拟化。
4. Linux虚拟化技术栈：KVM/QEMU/libvirt
   1. KVM：KVM（Kernel Virtual Machine，内核虚拟机）的运行需要主机是x86架构且硬件支持虚拟化技术（Intel VT或AMD-V），还需要一个经过修改的QEMU软件（qemu-kvm）作为虚拟机上层控制和界面。KVM能在不改变Linux或Windows镜像的情况下同时运行多个虚拟机，并为每一个虚拟机配置个性化硬件环境。
   2. KVM介绍：高性能、成本可控、需要CPU硬件支持。
   3. KVM主要功能：虚拟主机管理、共享存储管理、网络管理、快照与备份管理、迁移管理。
   4. QEMU：全称为Quick Emulator，原来是独立的全虚拟化软件，性能较差。现在一般与KVM结合（qemu-kvm），由qemu充当外层管理控制，底层内核依赖于KVM。采用pass through半虚拟化设备以提高性能。
   5. libvirt：是目前使用最广泛的对KVM虚拟机进行管理的工具和API。libvirtd是一个daemon进程，可以被本地或远超virsh调用。Libvirtd调用qemu-kvm操纵虚拟机，qemu-kvm调用KVM等内核模块实现虚拟化。
5. qemu-kvm使用：理解命令行参数含义
   1. qemu-img命令：用于管理虚拟机磁盘映像。

+check用于检查磁盘镜像

+create用于创建磁盘镜像

+convert用于执行磁盘镜像的格式转换，甚至可以执行异构虚拟主机之间的磁盘镜像转换

+info 用于查看磁盘镜像信息

+snapshot用于磁盘镜像快照的创建、删除与应用

+rebase用于磁盘镜像的派生与调整

* 1. qemu-img使用示例

创建一个磁盘映像

qemu-img create –f raw windows 4G

* 1. qemu-kvm命令：用于运行与安装KVM虚拟机

qemu-kvm [options] [disk\_image]

* 1. qemu-kvm重点参数

-m、-cpu：用于设定内存大小与CPU的个数

-had：指定IDE磁盘，后面可以跟磁盘镜像文件

-cdrom：CD驱动器，后面跟ISO安装程序

-boot[a|c|d]：有软盘（a）、硬盘（c）或CD-ROM（d）启动，默认由硬盘启动

-vnc：指定VNC管理地址

-net：用于配置网络信息，如配置IP地址、VLAN信息、MAC地址信息等

* 1. qemu-kvm使用示例

通过qemu-kvm命令将虚拟主机与镜像文件相关联，并在镜像文件中启动和安装虚拟主机

qemu-kvm -vnc :3 -cdrom win98.iso -m 256 -boot d windows

1. virsh使用：理解虚拟机定义XML文件

<domain type='kvm'> #用kvm创建的虚拟机称为domain，type定义使用哪个虚拟机管理程序，值可以是：xen、kvm、qemu、lxc、kqemu。

<name>fedora-14</name>#name参数为虚拟机定义了一个简短的名字，必须唯一。

<uuid>c7a5fdbd-cdaf-9455-926a-d65c16db1809</uuid> #uuid，全球唯一，linux下可以用uuidgen生成

<memory unit=’KiB’>219200</memory> #可以分配到的最大内存，内存单位有unit定义，单位可以是：K、KiB、M、MiB、G、GiB、T、TiB。默认是KiB。

<currentMemory>219200</currentMemory> #实际分给客户端的内存小于memory的定义，如果没有定义，值和memory一致。

<vcpu>2</vcpu> #vcpu的内容是为虚拟机最多分配几个cpu。

<os>

<type arch='i686' machine='pc'>hvm</type> #arch指定虚拟机的cpu架构，machine指定机器的类型。hvm表明该os被设计为直接运行在裸金属上面，需要全虚拟化。

<boot dev='cdrom'/> #dev属性的值可以是：fd、hd、cdrom、network，boot的元素可以被设置多个用来建立一个启动优先规则。

</os>

<devices> #所有的设备都是一个名为devices元素的子设备。

<emulator>/usr/libexec/qemu-kvm</emulator> #emulator元素指定模拟设备二进制文件的全路径。

<disk type='file' device='cdrom'>

<source file='/home/user/boot.iso'/> #source元素，在disk的type是file时，file属性指定一个合格的全路径文件映像作为客户端的磁盘。

<target dev='hdc'/>#dev属性表明本地磁盘在客户端上的实际名称，因为实际设备的名称指定并不能保证映射到客户端os上的设备。

<readonly/> #readonly元素，指定客户端不能修改设备。当一个disk含有type=cdrom，readonly则是默认值。

</disk>

<disk type='file' device='disk'>

<source file='/home/user/fedora.img'/>

<target dev='hda'/>

</disk>

<interface type='network'>

<source network='default'/>

</interface>

<graphics type='vnc' port='5900'/> #配置vnc的配置文件，port是指定端口号。

</devices>

</domain>

1. 分布式存储的定义和特点
   1. 分布式存储系统的定义：分布式存储系统是将为数众多的普通计算机或服务器通过网络进行连接，同时对外提供一个整体的存储服务。
   2. 特点：高性能、可扩展、低成本、易用性。
2. 根据数据类型的存储分类：非结构化数据/结构化数据/半结构化数据
   1. 非结构化数据：一般指无法结构化的数据，例如图片，文件，视频。
   2. 结构化数据：就是具有模式的数据，结构就是模式，例如数据库。
   3. 半结构化数据：它的数据是有结构的，但却不方便模式化。
3. 存储解决方案：SAN/NAS/Ceph
   1. SAN（Storage Area Network）：通过将磁盘存储系统和服务器直接相连的方式提供一个易扩展、高可靠的存储环境，高可靠的光纤通道交换机和光纤通道网络协议保证各个设备间链接的可靠性和高效性。设备间的连接接口主要是采用FC或者SCSI。
   2. NAS（Network Attached Storage）：通过基于TCP/IP的各种上层应用在各工作站和服务器之间进行文件访问，直接在工作站客户端和NAS文件共享设备之间建立连接，NAS隐藏了文件系统的底层实现，注重上层的文件服务实现，具有良好的扩展性。
   3. Ceph：Ceph是一个统一的、分布式的存储系统。
      * 1. 统一：意味着Ceph可以以一套存储系统同时提供“对象存储”“块存储”和“文件系统”三种功能，以满足不同应用的需求。
        2. 分布式：意味着无中心结构和系统规模的无限扩展。
   4. Ceph的设计思路：充分发挥存储设备自身的计算能力、去除所有的中心点。
4. 云计算网络相关概念：覆盖网络、SDN、NFV
   1. 覆盖网络（Overlay Network）：是一种在原有网络基础上构建网络连接抽象及管理的技术。覆盖网络中的节点可以被认为是通过虚拟或逻辑链接相连，其中每个链接对应一条路径（Path）。覆盖网络通常的实现方法是在原有网络的基础上构建隧道。目前常用于构建隧道的网络协议有GRE、VLAN、VXLAN、NVGRE、IPSEC。
   2. 灵活控制：软件定义网络（SDN）：是网络虚拟化的一种实现方式，其核心技术OpenFlow通过将网络设备的控制面与数据面分离开来，从而实现了网络流量的灵活控制，使网络作为管道变得更加智能，为核心网络及应用的创新提供了良好的平台。
   3. 快速部署：网络功能虚拟化（NFV）：一种对于网络架构的概念，利用虚拟化技术，将网络节点阶层的功能，分割成几个功能区块，分别以软件方式实现，不再局限于硬件架构。
5. OpenStack组件/项目：Keystone/Horizon/Nova/Neutron/Swift/Cinder /Glance（详见第6讲）
   1. Keystone-身份（Identity）
   2. Horizon-仪表板（Dashboard）
   3. Nova-计算（Compute）
   4. Neutron-网络（Network）
   5. Swift-对象存储（Object Storage）
   6. Cinder-块存储（Block Storage）
   7. Glance-镜像（Image）
6. OpenStack组件之间的通信关系

组件之间的通信主要是通过相互调用API来实现的。

OpenStack组件之间的通信分为四类：

* 1. 基于HTTP协议进行通信：通过各项目的API建立通信关系，基本上都属于这一类，这些API都是RESTful Web API，最常见的就是通过Horizon或者命令行接口对各组件操作的时候产生的这种通信，然后就是各组件通过Keystone对用户身份进行校验，进行验证的时候使用这种通信。还有比如说Nova Compute在获取镜像的时候和Glance之间，对Glance API的调用；还有比方说Swift数据的读写，也是通过这个HTTP协议的RESTful Web API来进行的。
  2. 基于高级消息队列协议：基于AMQP协议进行的通信，主要是每个项目内部各个组件之间的通信，比方说Nova的Nova Compute和Scheduler之间，然后Cinder的Scheduler和Cinder Volume之间。需要说明的是，Cinder是从Nova Volume演化出来的，所以Cinder和Nova之间也有通过AMQP协议的通信关系，由于AMQP协议进行通信也属于面向服务的架构，虽然大部分通过AMQP协议进行通信的组件属于同一个项目，但是并不要求它们安装在同一个节点上，给系统的横向扩展带来了很大的好处，可以对其中的各个组件分别按照他们负载的情况进行横向扩展，因为他们不在一个节点上，分别用不同数量的节点去承载它们的这些服务。
     + 1. AMQP是一种协议，OpenStack没有规定它是用什么实现，我们经常使用的是Private MQ，实际上用户也可以根据自身的情况选择其它的消息中间件。
  3. 基于SQL通信：通过数据库连接实现通信，这些通信大多也属于各个项目内部，也不要求数据库和项目其它组件安装在同一个节点上，它也可以分开安装，还可以专门部署数据库服务器，把数据库服务放到上面，之间通过基于SQL的这些连接来进行通信。OpenStack没有规定必须使用哪种数据库，通常使用MySQL。
  4. 通过Native API实现通信：出现在OpenStack各组件和第三方的软硬件之间，比如说，Cinder和存储后端之间的通信，Neutron的agent或者说插件和网络设备之间的通信，这些通信都需要调用第三方的设备或第三方软件的API，我们称为它们为Native API，那么这个就是我们前面说的基于第三方API的通信。

1. OpenStack部署：控制节点/计算节点
   1. 控制节点：一般只需要1个网络端口用于通信和管理各个节点。
   2. 计算节点：需要2个网络端口，一个与控制节点进行通信，受控制节点统一调配；另一个与网络节点和存储节点进行通信。
   3. 存储节点：需要2个网络端口，一个与控制节点进行通信，受控制节点统一调配；另一个与计算节点和网络节点进行通信。
   4. 网络节点：需要3个网络端口，分别用于与控制节点进行通信、与除控制节点之外的计算和存储节点之间的通信、外部的虚拟机与相应网络之间的通信。
2. OpenStack网络：提供者网络/自服务网络



1. OpenStack API形式：RESTful API
   1. RESTful架构的核心是资源和资源的操作。
   2. OpenStack各个项目都提供了RESTful架构的API作为对外提供的接口。OpenStack定义了很多的资源，并实现了针对这些资源的各种操作函数。其API服务进程接收到客户端的HTTP请求时，一个所谓的“路由”模块就会将请求的URL转化成相应的资源，并路由到合适的操作函数上。
2. REST资源CRUD操作与HTTP请求类型的对应

|  |  |
| --- | --- |
| REST资源CRUD操作 | HTTP请求类型 |
| Create | GET |
| Retrieve | POST |
| Update | PUT |
| Delete | DELETE |

1. OpenStack API访问方式

从底层看，访问OpenStack提供的服务也就是访问OpenStack提供的API，通常有四种方式可以用来发送API请求：cURL、OpenStack命令行客户端、REST客户端和python SDK。

* 1. cURL：一种用来发送http请求和接受http相应的linux命令行工具。
  2. OpenStack命令行客户端：OpenStack提供封装了用户容易使用的命令的python客户端来访问底层的API，如python-novaclient,python-glanceclient等，都可以用来访问OpenStack的API。
  3. REST客户端：一种基于浏览器的图形界面，用于和OpenStack底层的API进行http请求的发送和相应的接收。
  4. python SDK：python软件开发工具包，使用该工具包编译python自动化脚本程序，创建和管理OpenStack上的资源。该工具包实现了与python之间的绑定，允许用户使用python调用而不是REST调用执行相应的任务，所有的OpenStack命令行工具的底层均是由python SDK实现。

1. OpenStack认证与API请求流程
   1. 请求认证token时，发送认证信息。
   2. 如果认证成功，会获得认证token。
   3. 在发送的API请求中将认证token填入X-Auth-Token字段。可以一直使用这个认证token发送API请求，直到任务完成或出现401非认证错误。
   4. 如果出现401非认证错误，可以重新请求一个认证token。
2. curl访问API：理解命令行参数含义

curl -X POST -H "Content-Type: application/json" -d '{"sentence":"hahaha"}' [http://:8080/sentiment](NULL)

* 1. 通过 -X 来指定请求时的HTTP方法，通常使用POST方法，GET方法省略。
  2. 通过 -H 来添加Header信息，后面跟键值对，不需要大括号。
  3. 使用POST方法通常是为了往服务器传递数据，可以利用 -d 来完成。后面需要用带大括号键值对的方式去传递。

1. 云原生应用的技术：微服务、DevOps、容器技术、CI/CD
   1. 微服务技术：将应用程序开发为一系列小型服务的结构，每个服务实现独立的业务功能，运行在自己的独立环境中，并通过RESTful API或消息传递进行通信。每个微服务可以独立于应用程序中的其他服务进行部署、升级、扩展和重新启动，通常可以自动化进行，能够在不影响用户使用的情况下频繁更新应用程序。
   2. DevOps技术：通过自动化软件交付和架构变更的流程，使得构建、测试、发布软件能够更加地快捷、频繁和可靠。可以把DevOps看作开发（软件工程）、技术运营和质量保障（QA）三者的交集。
   3. 容器技术：容器技术与虚拟机技术相比，拥有更高的资源使用效率，因为它并不需要为每个应用分配单独的操作系统，所以实例规模更小、创建和迁移速度也更快。相对于虚拟机，单个操作系统能够承载更多的容器。
      * 1. 好处：容器化最大的好处是保持运行环境的一致性，只要应用可以打包成容器镜像（通常使用Docker容器），就可以一次编程后，在各处运行。
        2. 容器也可以作为应用运行的最小组件来部署，且更适合作为无状态应用运行。结合容器编排工具（如Kubernetes）将大大增加系统的扩展性和自愈能力，轻松应对大流量下的高并发场景，加快业务的迭代速度。
   4. CI/CD：持续集成与持续交付技术
      * 1. 持续集成是一种软件开发的实践方法，它要求团队成员经常整合他们的工作成果（通常是程序代码）。通常情况下，团队成员中的每人每天至少提交一次自己的代码到代码仓库做集成构建，这样对于整个项目而言，每天就会有多次集成构建。
        2. 持续交付是一种以可持续的方式安全快速地将所有类型的软件变更（包括新功能开发、配置更改、Bug修复等）转化为生产环节下的工作产品交付给用户直接使用的软件过程控制方法，它的最终目标是将变更直接部署到生产环境。
2. 云原生应用12要素：对任一要素可以解释说明（详见第9讲）
   1. 一份代码库与多份部署
   2. 显示声明依赖关系
   3. 在环境中存储配置
   4. 把后端服务当作附加资源
   5. 严格分离构建和运行
   6. 以一个或多个无状态进程运行应用
   7. 通过端口绑定提供服务
   8. 通过进程模型进行扩展
   9. 快速启动和优雅终止可最大化健壮性
   10. 尽可能保持开发与预发布线上环境相同
   11. 把日志当作事件流
   12. 后台管理任务当作一次性进程运行
3. 微服务架构相关：SOA/MSA
   1. SOA：面向服务架构（SOA，Service Oriented Architecture）。面向服务的架构是一个组件模型，它将应用程序的不同功能单元（称为服务）通过这些服务之间定义良好的接口和契约联系起来。SOA实际是对软件构件之间的进一步解耦。
   2. MSA：微服务架构（MSA，MicroService Architecture）。是一种软件架构风格，以专注于单一责任与功能的小型构建块为基础，组合出复杂的大型应用程序，使用与编程语言无关的API集相互通信。
      * 1. 微服务：依业务功能设计的小服务，拥有自己的进程，使用轻量化机制与其他服务通信（通常是HTTP API），以全自动的方式部署，使用最小规模的集中管理能力，服务可以用不同的编程语言与数据库等组件实现。
   3. 微服务的优势：微服务架构相对于传统SOA应用，一般是单体架构。
      * 1. 单体架构应用的问题：每一个业务功能是不可分割的。若要进行扩展则必须将整个应用都放到新的运算资源内（如虚拟机），会有大量的资源浪费。
        2. 微服务架构：以业务功能或流程设计进行分割，将各个业务功能都独立实现成一个能自主运行的个体服务，再组合起来形成一个应用程序。若需要针对特定业务功能进行扩展时，只要对该业务功能的服务进行扩展。
4. Web Service相关：SOAP方式/RESTful方式
   1. SOAP：简单对象访问协议。简单对象访问协议是交换数据的一种协议规范，是一种轻量级、简单的、基于XML（标准通用标记语言下的一个子集）的协议，它被设计成在WEB上交换结构化的和固化的信息。
   2. RESTful：是一种网络应用程序的设计风格和开发方式，基于HTTP，可以使用XML格式定义或者JSON格式定义。RESTful适用于移动互联网厂商作为业务使能接口的场景，实现第三方OTT调用移动网络资源的功能，动作类型为新增、变更、删除所调用资源。
5. 容器与虚拟机技术的对比
   1. 容器：是“轻量级”的操作系统虚拟化，镜像大小在100M量级，可以在秒级启动，可以在一个资源隔离的进程中运行应用及其依赖项。运行应用程序所必需的组件都将打包成一个镜像并可以复用，运行在一个隔离环境中，并且不会共享宿主机的内存、CPU以及磁盘，这就保证了容器内进程不能监控容器外的任何进程。
   2. 虚拟机：虚拟机是“重量级”的，通常包含整个操作系统及其应用程序，同时也需要运行一个hypervisor来控制虚拟机，镜像大小在10G量级，部署和启动需要较长时间。
   3. 容器技术和虚拟机技术并不互斥，云计算环境中容器一般运行在虚拟机中。
6. 容器技术的优势
   1. 简单部署：应用打包成单一地址访问、Registry存储、通过单一命令就可以部署完成的组件。
   2. 快速启动：容器技术对操作系统的资源进行抽象而非对硬件资源进行虚拟化，打包好的服务可以快速启动。
   3. 服务组合：采用容器的方式进行部署，整个系统会变得易于组合。
   4. 易于迁移：允许用户对工作负载进行迁移，避免局限于单一的平台提供商。
7. Docker核心技术：chroot、CGroups、namespace、UnionFS
   1. chroot：通过改变系统根目录，能够限制用户的权利，在新的根目录下并不能够访问旧系统根目录的结构和文件。
   2. CGroups：隔离和限制宿主机器上的物理资源，例如CPU、内存、磁盘I/O和网络带宽。
   3. namespace：是Linux提供的用于分离进程树、网络接口、挂载点以及进程间通信等资源的方法。
   4. UnionFS：支持分层文件系统构造镜像。
8. Docker基本概念：镜像、仓库（docker.io）、容器
   1. 镜像（image）：就是一堆只读层（read-only layer）的统一视角。Docker提供了一个很简单的机制来创建镜像或者更新现有的镜像，用户甚至可以直接从其他人那里下载一个已经做好的镜像来直接使用。
   2. 仓库（repository）：是集中存放镜像文件的场所，官方仓库docker.io。用户创建了自己的镜像之后就可以使用push命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像的时候，只需要从仓库上pull下来。
   3. 容器（container）：容器是从镜像创建的运行实例，它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台，可以把容器看做是一个独立的Linux环境和运行在其中的应用程序。一个镜像可以创建很多容器。
9. Docker基本命令：pull、push、run、build、ps、rm、rmi、login、
   1. docker pull：从仓库获取所需要的镜像。
      * 1. 使用示例：docker pull centos:centos6

含义：从docker.io获取centos仓库的centos6镜像。

* 1. docker build：利用Dockerfile来创建镜像。Dockerfile中每条指令创建镜像的一层，其中的每层可以复用。
  2. docker push：上传镜像。
  3. docker run <image-id>：创建容器。是docker create和docker start两个命令的组合。
  4. docker ps：查看容器列表。
  5. docker rm <container-id>：删除容器。
  6. docker rmi <image-id>：删除镜像。
  7. docker login：用于登录Docker Registry。

1. Docker镜像构建：Dockerfile解读

FROM python:3.6.6-alpine #必须放在Dockerfile的第一行，表示从哪个基础镜像开始构建。

COPY sa /app #拷贝文件或目录到镜像中，不会自动解压文件，也不能访问网络资源。

WORKDIR /app #指定在创建容器后终端默认登录进来的工作目录，一个落脚点。

RUN pip3 install -r requirements.txt && \ python3 -m textblob.download\_corpora #容器构建时需要运行的命令。

EXPOSE 5000 #当前容器对外暴露出的端口。

ENTRYPOINT ["python3"] #配置容器，使其可执行化。

CMD ["sentiment\_analysis.py"] #指定容器启动时要运行的命令。

1. 应用扩展方式：垂直扩展scale up/水平扩展 scale out
   1. 垂直扩展（Scale up）：在同一个逻辑单位中添加资源以扩充处理能力，例如增加服务器的CPU数量、内存数量、硬盘数量等，受制于硬件限制，会有无法突破的上限。
   2. 水平扩展（Scale out）：增加更多的逻辑单位并使其成为一个整体协同工作，以获得更强处理能力，理论上可以无限扩展。水平扩展需要应用架构和基础设施两方面的支持。
2. 云资源调度策略分类

根据不同的优化目标，资源调度策略及算法可以划分成三种类型：基于性能的资源调度、基于成本的资源调度、基于性能和成本的资源调度。

* 1. 本地性感知任务调度主要是为了提高任务的执行效率，可靠性感知任务调度主要是为了提高云系统的可靠性，都属于基于性能的资源调度。
  2. 能耗感知资源调度是为了减小运营成本，属于基于成本的资源调度。
  3. 工作流调度同时优化了时间和成本，属于基于性能和成本的资源调度。
  4. 云计算资源调度的四个热点问题：
     + 1. 本地性感知任务调度问题：如何在云资源调度中增强数据本地性来提高执行效率以节约网络带宽。
       2. 可靠性感知调度问题：如何减少云计算资源调度中任务的失效率来提高云系统的可靠性和执行效率。
       3. 能耗感知资源调度问题：如何通过降低数据中心的能源消耗来减少云提供商的运营成本。
       4. 工作流调度问题：如何优化工作流调度来权衡完成时间与成本。

1. 容器编排的任务
   1. 容器编排：现代应用（云应用）一般由容器化的单独组件（通常称为微服务）组成，且须按调用关系在网络级别进行组织，以使其能够按照设计结构运行。以这种方法对多个容器进行组织的流程即称为容器编排。
   2. 主要任务：
      * 1. 让用户控制容器启动和停止的时间、将其组合到集群中，以及协调应用组合的流程。
        2. 允许用户指导容器部署与自动更新、运行状况监控以及故障转移等。
2. Kubenetes基本概念：Pod、Service、Deployment、Label
   1. Pod：Kubernetes的最小管理单位，用于控制创建、重启、伸缩一组功能相近、共享磁盘的Docker容器。
   2. Service：可用作服务发现，类似于LoadBalance，通过Selectors为一组Pod提供对外的接口。
   3. Deployment：用于部署应用。它是Kubernetes中最常用的一个对象，它为ReplicaSet和Pod的创建提供了一种声明式的定义方法，从而无需手动创建ReplicaSet和Pod对象。
      * 1. 通过使用Deployment对象，可以轻松实现：创建ReplicaSet和Pod、滚动升级（不停止旧服务的状态下升级）和回滚应用（将应用回滚到之前的版本）、平滑地扩容和缩容、暂停和继续Deployment。
   4. Label：K/V键值对，用来标记Kubernetes组件的类别（例如标记一组Pod是frontService，另一组是backService）。Label对于Kubernetes的伸缩调度非常重要。
3. Kubenetes使用：解读YAML文件

sa-frontend-pod.yaml

apiVersion: v1

kind: Pod # 1 kind:指定想创建的Kubernetes资源的类型，这里是Pod。

metadata:

name: sa-frontend # 2 name：定义了该资源的名字，在这里命名为sa-frontend。

labels:

app: sa-frontend

spec: # 3 spec：定义了资源应有的状态，Pod Spec中最重要的属性是容器的数组。

containers:

- image: xieqian02/sa-frontend # 4 image：希望在本Pod中启动的容器的映像。

name: sa-frontend # 5 name：Pod容器的名字。

ports: - containerPort: 80 # 6 containerPort：是指容器监听的端口号，只是为了提供文档信息（即便没有这个端口也不会影响访问）。