1.1微服务与单体应用区别和优劣势

单体应用：

那么单体应用本身带来的问题主要有哪些？  
  
1.系统复杂：内部多个模块紧耦合，关联依赖复杂，牵一发而动全身。  
2.运维困难：变更或升级的影响分析困难，任何一个小修改都可能导致单体应用整体运行出现故障。  
3.无法扩展：无法拆分部署，出现性能瓶颈后往往只能够增加服务器或增加集群节点，但是DB问题难解决

微服务优点：

微服务架构强调的第一个重点就是业务系统需要彻底的组件化和服务化，原有的单个业务系统会拆分为多个可以独立开发，设计，运行和运维的小应用。这些小应用之间通过服务完成交互和集成。每个小应用从前端web ui，到控制层，逻辑层，数据库访问，数据库都完全是独立的一套。在这里我们不用组件而用小应用这个词更加合适，每个小应用除了完成自身本身的业务功能外，重点就是还需要消费外部其它应用暴露的服务，同时自身也将自身的能力朝外部发布为服务。

微服务缺点

1.开发者需要应对创建分布式系统所产生的额外的复杂因素

l  目前的IDE主要面对的是单体工程程序，无法显示支持分布式应用的开发

l  [测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "http://blog.csdn.net/sunhuiliang85/article/details/_blank)工作更加困难

l  需要采用服务间的通讯机制

l  很难在不采用分布式事务的情况下跨服务实现功能

l  跨服务实现要求功能要求团队之间的紧密协作

2.部署复杂

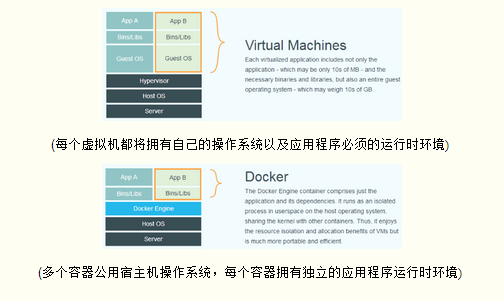
3.内存占用量更高

1.2Docker简介

它和经典的虚机的虚拟化方式不一样，经典的虚机会引入操作系统，库函数和程序

Docker只打包与别人不一样的程序

Docker：标准化的构建，交付和运维手段



## **1.3 Docker的优势**

　　Docker 容器的启动可以在秒级实现，这相比传统的虚拟机方式要快得多。 其次，Docker 对系统资源的利用率很高，一台主机上可以同时运行数千个 Docker 容器。

　　容器除了运行其中应用外，基本不消耗额外的系统资源，使得应用的性能很高，同时系统的开销尽量小。传统虚拟机方式运行 10 个不同的应用就要起 10 个虚拟机，而Docker 只需要启动 10 个隔离的应用即可。

　　具体说来，Docker 在如下几个方面具有较大的优势。

* **更快速的交付和部署**

　　对开发和运维（devop）人员来说，最希望的就是一次创建或配置，可以在任意地方正常运行。

开发者可以使用一个标准的镜像来构建一套开发容器，开发完成之后，运维人员可以直接使用这个容器来部署代码。 Docker 容器很轻很快！容器的启动时间是秒级的，大量地节约开发、测试、部署的时间。

* **更高效的虚拟化**

　　Docker 容器的运行不需要额外的 hypervisor(虚拟化管理程序)支持，它是内核级的虚拟化，因此可以实现更高的性能和效率。

* **更轻松的迁移和扩展**

　　Docker 容器几乎可以在任意的平台上运行，包括物理机、虚拟机、公有云、私有云、个人电脑、服务器等。 这种兼容性可以让用户把一个应用程序从一个平台直接迁移到另外一个。

|  |  |  |
| --- | --- | --- |
| 特性 | 容器(Docker) | 虚拟机(VM) |
| 启动 | 秒级 | 分钟级 |
| 硬盘使用 | MB级 | GB级 |
| 性能 | 接近原生 | 弱于 |
| 系统支持量 | 单机支持上千个容器 | 单机支持几十个 |

## **1.4 Docker的局限**

Docker并不是全能的，设计之初也不是KVM之类虚拟化手段的替代品，简单总结几点：

* Docker是基于Linux 64bit的，无法在windows/unix或32bit的linux环境下使用LXC是基于linux kernel的，因此container的guest系统只能是linux base的隔离性比KVM虚拟化方案还是有些欠缺，所有container公用一部分的运行库网络管理相对简单，主要是基于namespace隔离cgroup的cpu功能相比KVM的等虚拟化方案相比难以度量docker对disk的管理比较有限container随着用户进程的停止而销毁，container中的log等用户数据不便收集

　　1-2，有windows base应用的是不能使用Docker的

　　3-5主要是看用户的需求，到底是需要一个container还是一个VM, 同时也决定了docker作为 IaaS 不太可行

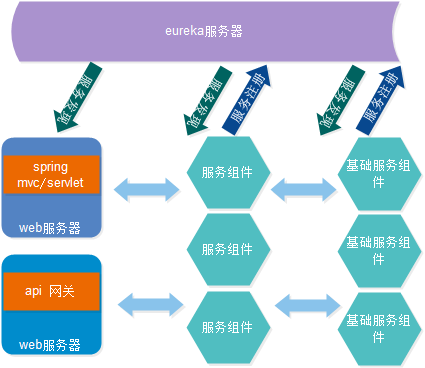
6-7虽然是docker本身不支持的功能，但是可以通过其他手段解决(disk quota, mount --bind)。总之，选用container还是vm, 就是在隔离性和资源复用性上做权衡

1.5springcloud

1.5.1springcloud是spring4.0之后出现的将一些经过实战考研的，在分布式环境中通过组件组装起来，通过注解的方式包装成框架，让你非常方便的使用。所有的服务和微服务都是springboot项目，springboot把tomcat等外部环境一块打包，是的你的环境中只要有jar包，就可以跑你的应用。

1.5.2服务注册（基于eureka服务器）

服务注册管理器原理如下图所示：



所有的服务端及访问服务的客户端都需要连接到注册管理器（eureka服务器）。服务在启动时会自动注册自己到eureka服务器，每一个服务都有一个名字，这个名字会被注册到eureka服务器。使用服务的一方只需要使用该名字加上方法名就可以调用到服务。

　　Spring cloud的服务注册及发现，不仅仅只有eureka，还支持Zookeeper和Consul。默认情况下是eureka，spring 封装了eureka，使其非常简单易用，只需要比传统应用增加一行代码就可以使用了，这一行代码就是一个注解。

## 为什么说eureka比zookeeper好？

## Zookeeper保证CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

## Eureka保证AP

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：   
1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务   
2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)   
3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。