Kubernetes设计理念与分布式系统

分析和理解Kubernetes的设计理念可以使我们更深入地了解Kubernetes系统，更好地利用它管理分布式部署的云原生应用，另一方面也可以让我们借鉴其在分布式系统设计方面的经验。

API设计原则

对于云计算系统，系统API实际上处于系统设计的统领地位，正如本文前面所说，Kubernetes集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的API对象，支持对该功能的管理操作，理解掌握的API，就好比抓住了Kubernetes系统的牛鼻子。Kubernetes系统API的设计有以下几条原则：

1. 所有API应该是声明式的。正如前文所说，声明式的操作，相对于命令式操作，对于重复操作的效果是稳定的，这对于容易出现数据丢失或重复的分布式环境来说是很重要的。另外，声明式操作更容易被用户使用，可以使系统向用户隐藏实现的细节，隐藏实现的细节的同时，也就保留了系统未来持续优化的可能性。此外，声明式的API，同时隐含了所有的API对象都是名词性质的，例如Service、Volume这些API都是名词，这些名词描述了用户所期望得到的一个目标分布式对象。

2. API对象是彼此互补而且可组合的。这里面实际是鼓励API对象尽量实现面向对象设计时的要求，即“高内聚，松耦合”，对业务相关的概念有一个合适的分解，提高分解出来的对象的可重用性。事实上，Kubernetes这种分布式系统管理平台，也是一种业务系统，只不过它的业务就是调度和管理容器服务。

3. 高层API以操作意图为基础设计。如何能够设计好API，跟如何能用面向对象的方法设计好应用系统有相通的地方，高层设计一定是从业务出发，而不是过早的从技术实现出发。因此，针对Kubernetes的高层API设计，一定是以Kubernetes的业务为基础出发，也就是以系统调度管理容器的操作意图为基础设计。

4. 低层API根据高层API的控制需要设计。设计实现低层API的目的，是为了被高层API使用，考虑减少冗余、提高重用性的目的，低层API的设计也要以需求为基础，要尽量抵抗受技术实现影响的诱惑。

5. 尽量避免简单封装，不要有在外部API无法显式知道的内部隐藏的机制。简单的封装，实际没有提供新的功能，反而增加了对所封装API的依赖性。内部隐藏的机制也是非常不利于系统维护的设计方式，例如StatefulSet和ReplicaSet，本来就是两种Pod集合，那么Kubernetes就用不同API对象来定义它们，而不会说只用同一个ReplicaSet，内部通过特殊的算法再来区分这个ReplicaSet是有状态的还是无状态。

6. API操作复杂度与对象数量成正比。这一条主要是从系统性能角度考虑，要保证整个系统随着系统规模的扩大，性能不会迅速变慢到无法使用，那么最低的限定就是API的操作复杂度不能超过O(N)，N是对象的数量，否则系统就不具备水平伸缩性了。

7. API对象状态不能依赖于网络连接状态。由于众所周知，在分布式环境下，网络连接断开是经常发生的事情，因此要保证API对象状态能应对网络的不稳定，API对象的状态就不能依赖于网络连接状态。

8. 尽量避免让操作机制依赖于全局状态，因为在分布式系统中要保证全局状态的同步是非常困难的。

控制机制设计原则

- **控制逻辑应该只依赖于当前状态。**这是为了保证分布式系统的稳定可靠，对于经常出现局部错误的分布式系统，如果控制逻辑只依赖当前状态，那么就非常容易将一个暂时出现故障的系统恢复到正常状态，因为你只要将该系统重置到某个稳定状态，就可以自信的知道系统的所有控制逻辑会开始按照正常方式运行。
- **假设任何错误的可能，并做容错处理。**在一个分布式系统中出现局部和临时错误是大概率事件。错误可能来自于物理系统故障，外部系统故障也可能来自于系统自身的代码错误，依靠自己实现的代码不会出错来保证系统稳定其实也是难以实现的，因此要设计对任何可能错误的容错处理。

- **尽量避免复杂状态机，控制逻辑不要依赖无法监控的内部状态。**因为分布式系统各个子系统都是不能严格通过程序内部保持同步的，所以如果两个子系统的控制逻辑如果互相有影响，那么子系统就一定要能互相访问到影响控制逻辑的状态，否则，就等同于系统里存在不确定的控制逻辑。
- **假设任何操作都可能被任何操作对象拒绝，甚至被错误解析。**由于分布式系统的复杂性以及各子系统的相对独立性，不同子系统经常来自不同的开发团队，所以不能奢望任何操作被另一个子系统以正确的方式处理，要保证出现错误的时候，操作级别的错误不会影响到系统稳定性。
- **每个模块都可以在出错后自动恢复。**由于分布式系统中无法保证系统各个模块是始终连接的，因此每个模块要有自我修复的能力，保证不会因为连接不到其他模块而自我崩溃。
- **每个模块都可以在必要时优雅地降级服务。**所谓优雅地降级服务，是对系统鲁棒性的要求，即要求在设计实现模块时划分清楚基本功能和高级功能，保证基本功能不会依赖高级功能，这样同时就保证了不会因为高级功能出现故障而导致整个模块崩溃。根据这种理念实现的系统，也更容易快速地增加新的高级功能，因为不必担心引入高级功能影响原有的基本功能。

Kubernetes的核心技术概念和API对象

API对象是Kubernetes集群中的管理操作单元。Kubernetes集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的API对象，支持对该功能的管理操作。例如副本集Replica Set对应的API对象是RS。

每个API对象都有3大类属性：元数据metadata、规范spec和状态status。元数据是用来标识API对象的，每个对象都至少有3个元数据：namespace，name和uid；除此以外还有各种各样的标签labels用来标识和匹配不同的对象，例如用户可以用标签env来标识区分不同的服务部署环境，分别用env=dev、env=testing、env=production来标识开发、测试、生产的不同服务。规范描述了

用户期望Kubernetes集群中的分布式系统达到的理想状态（Desired State），例如用户可以通过复制控制器Replication Controller设置期望的Pod副本数为3；status描述了系统实际当前达到的状态（Status），例如系统当前实际的Pod副本数为2；那么复制控制器当前的程序逻辑就是自动启动新的Pod，争取达到副本数为3。

Kubernetes中所有的配置都是通过API对象的spec去设置的，也就是用户通过配置系统的理想状态来改变系统，这是Kubernetes重要设计理念之一，即所有的操作都是声明式（Declarative）的而不是命令式（Imperative）的。声明式操作在分布式系统中的好处是稳定，不怕丢操作或运行多次，例如设置副本数为3的操作运行多次也还是一个结果，而给副本数加1的操作就不是声明式的，运行多次结果就错了。

Pod

Kubernetes有很多技术概念，同时对应很多API对象，最重要的也是最基础的是Pod。Pod是在Kubernetes集群中运行部署应用或服务的最小单元，它是可以支持多容器的。Pod的设计理念是支持多个容器在一个Pod中共享网络地址和文件系统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。Pod对多容器的支持是K8s最基础的设计理念。比如你运行一个操作系统发行版的软件仓库，一个Nginx容器用来发布软件，另一个容器专门用来从源仓库做同步，这两个容器的镜像不太可能是一个团队开发的，但是他们一块儿工作才能提供一个微服务；这种情况下，不同的团队各自开发构建自己的容器镜像，在部署的时候组合成一个微服务对外提供服务。

Pod是Kubernetes集群中所有业务类型的基础，可以看作运行在K8集群中的小机器人，不同类型的业务就需要不同类型的小机器人去执行。目前Kubernetes中的业务主要可以分为长期伺服型（long-running）、批处理型（batch）、节点后台支撑型（node-daemon）和有状态应用型（stateful application）；分别对应的小机器人控制器为Deployment、Job、DaemonSet和StatefulSet，本文后面会一一介绍。

副本控制器（Replication Controller，RC）

RC是Kubernetes集群中最早的保证Pod高可用的API对象。通过监控运行中的Pod来保证集群中运行指定数目的Pod副本。指定的数目可以是多个也可以是1个；少于指定数目，RC就会启动运行新的Pod副本；多于指定数目，RC就会杀死多余的Pod副本。即使在指定数目为1的情况下，通过RC运行Pod也比直接运行Pod更明智，因为RC也可以发挥它高可用的能力，保证永远有1个Pod在运行。RC是Kubernetes较早期的技术概念，只适用于长期伺服型的业务类型，比如控制小机器人提供高可用的Web服务。

副本集（Replica Set，RS）

RS是新一代RC，提供同样的高可用能力，区别主要在于RS后来居上，能支持更多种类的匹配模式。副本集对象一般不单独使用，而是作为Deployment的理想状态参数使用。

部署（Deployment）

部署表示用户对Kubernetes集群的一次更新操作。部署是一个比RS应用模式更广的API对象，可以是创建一个新的服务，更新一个新的服务，也可以是滚动升级一个服务。滚动升级一个服务，实际是创建一个新的RS，然后逐渐将新RS中副本数增加到理想状态，将旧RS中的副本数减小到0的复合操作；这样一个复合操作用一个RS是不太好描述的，所以用一个更通用的Deployment来描述。以Kubernetes的发展方向，未来对所有长期伺服型的的业务的管理，都会通过Deployment来管理。

服务（Service）

RC、RS和Deployment只是保证了支撑服务的微服务Pod的数量，但是没有解决如何访问这些服务的问题。一个Pod只是一个运行服务的实例，随时可能在一个节点上停止，在另一个节点以一个新的IP启动一个新的Pod，因此不能以确定的IP和端口号提供服务。要稳定地提供服务需要服务发现和负载均衡能力。服务发

现完成的工作，是针对客户端访问的服务，找到对应的的后端服务实例。在K8s集群中，客户端需要访问的服务就是Service对象。每个Service会对应一个集群内部有效的虚拟IP，集群内部通过虚拟IP访问一个服务。在Kubernetes集群中微服务的负载均衡是由Kube-proxy实现的。Kube-proxy是Kubernetes集群内部的负载均衡器。它是一个分布式代理服务器，在Kubernetes的每个节点上都有一个；这一设计体现了它的伸缩性优势，需要访问服务的节点越多，提供负载均衡能力的Kube-proxy就越多，高可用节点也随之增多。与之相比，我们平时在服务器端做个反向代理做负载均衡，还要进一步解决反向代理的负载均衡和高可用问题。

任务（Job）

Job是Kubernetes用来控制批处理型任务的API对象。批处理业务与长期伺服业务的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出了。成功完成的标志根据不同的spec.completions策略而不同：单Pod型任务有一个Pod成功就标志完成；定数成功型任务保证有N个任务全部成功；工作队列型任务根据应用确认的全局成功而标志成功。

后台支撑服务集（DaemonSet）

长期伺服型和批处理型服务的核心在业务应用，可能有些节点运行多个同类业务的Pod，有些节点上又没有这类Pod运行；而后台支撑型服务的核心关注点在Kubernetes集群中的节点（物理机或虚拟机），要保证每个节点上都有一个此类Pod运行。节点可能是所有集群节点也可能是通过nodeSelector选定的一些特定节点。典型的后台支撑型服务包括，存储，日志和监控等在每个节点上支持Kubernetes集群运行的服务。

有状态服务集（StatefulSet）

Kubernetes在1.3版本里发布了Alpha版的PetSet功能，在1.5版本里将PetSet功能升级到了Beta版本，并重新命名为StatefulSet，最终在1.9版本里成为正式GA版本。在云原生应用的体系里，有下面两组近义词；第一组是无状态（stateless）、牲畜（cattle）、无名（nameless）、可丢弃（disposable）；

第二组是有状态（stateful）、宠物（pet）、有名（having name）、不可丢弃（non-disposable）。RC和RS主要是控制提供无状态服务的，其所控制的Pod的名字是随机设置的，一个Pod出故障了就被丢弃掉，在另一个地方重启一个新的Pod，名字变了、名字和启动在哪儿都不重要，重要的只是Pod总数；而StatefulSet是用来控制有状态服务，StatefulSet中的每个Pod的名字都是事先确定的，不能更改。StatefulSet中Pod的名字的作用，并不是《千与千寻》的人性原因，而是关联与该Pod对应的状态。

对于RC和RS中的Pod，一般不挂载存储或者挂载共享存储，保存的是所有Pod共享的状态，Pod像牲畜一样没有分别（这似乎也确实意味着失去了人性特征）；对于StatefulSet中的Pod，每个Pod挂载自己独立的存储，如果一个Pod出现故障，从其他节点启动一个同样名字的Pod，要挂载上原来Pod的存储继续以它的状态提供服务。

适合于StatefulSet的业务包括数据库服务MySQL和PostgreSQL，集群化管理服务ZooKeeper、etcd等有状态服务。StatefulSet的另一种典型应用场景是作为一种比普通容器更稳定可靠的模拟虚拟机的机制。传统的虚拟机正是一种有状态的宠物，运维人员需要不断地维护它，容器刚开始流行时，我们用容器来模拟虚拟机使用，所有状态都保存在容器里，而这已被证明是非常不安全、不可靠的。使用StatefulSet，Pod仍然可以通过漂移到不同节点提供高可用，而存储也可以通过外挂的存储来提供高可靠性，StatefulSet做的只是将确定的Pod与确定的存储关联起来保证状态的连续性。

集群联邦（Federation）

Kubernetes在1.3版本里发布了beta版的Federation功能。在云计算环境中，服务的作用距离范围从近到远一般可以有：同主机（Host，Node）、跨主机同可用区（Available Zone）、跨可用区同地区（Region）、跨地区同服务商（Cloud Service Provider）、跨云平台。Kubernetes的设计定位是单一集群在同一个地域内，因为同一个地区的网络性能才能满足Kubernetes的调度和计算存储连接要求。而联合集群服务就是为提供跨Region跨服务商Kubernetes集群服务而设计的。

每个Kubernetes Federation有自己的分布式存储、API Server和Controller Manager。用户可以通过Federation的API Server注册该Federation的成员Kubernetes Cluster。当用户通过Federation的API Server创建、更改API对象时，Federation API Server会在自己所有注册的子Kubernetes Cluster都创建一份对应的API对象。在提供业务请求服务时，Kubernetes Federation会先在自己的各个子Cluster之间做负载均衡，而对于发送到某个具体Kubernetes Cluster的业务请求，会依照这个Kubernetes Cluster独立提供服务时一样的调度模式去做Kubernetes Cluster内部的负载均衡。而Cluster之间的负载均衡是通过域名服务的负载均衡来实现的。

Federation V1的设计是尽量不影响Kubernetes Cluster现有的工作机制，这样对于每个子Kubernetes集群来说，并不需要更外层的有一个Kubernetes Federation，也就是意味着所有现有的Kubernetes代码和机制不需要因为Federation功能有任何变化。

目前正在开发的Federation V2，在保留现有Kubernetes API的同时，会开发新的Federation专用的API接口，详细内容可以在这里找到。

存储卷（Volume）

Kubernetes集群中的存储卷跟Docker的存储卷有些类似，只不过Docker的存储卷作用范围为一个容器，而Kubernetes的存储卷的生命周期和作用范围是一个Pod。每个Pod中声明的存储卷由Pod中的所有容器共享。Kubernetes支持非常多的存储卷类型，特别的，支持多种公有云平台的存储，包括AWS，Google和Azure云；支持多种分布式存储包括GlusterFS和Ceph；也支持较容易使用的主机本地目录emptyDir, hostPath和NFS。Kubernetes还支持使用Persistent Volume Claim即PVC这种逻辑存储，使用这种存储，使得存储的使用者可以忽略后台的实际存储技术（例如AWS，Google或GlusterFS和Ceph），而将有关存储实际技术的配置交给存储管理员通过Persistent Volume来配置。

持久存储卷（Persistent Volume，PV）和持久存储卷声明（Persistent Volume

Claim，PVC）

PV和PVC使得Kubernetes集群具备了存储的逻辑抽象能力，使得在配置Pod的逻辑里可以忽略对实际后台存储技术的配置，而把这项配置的工作交给PV的配置者，即集群的管理者。存储的PV和PVC的这种关系，跟计算的Node和Pod的关系是非常类似的；PV和Node是资源的提供者，根据集群的基础设施变化而变化，由Kubernetes集群管理员配置；而PVC和Pod是资源的使用者，根据业务服务的需求变化而变化，有Kubernetes集群的使用者即服务的管理员来配置。

节点（Node）

Kubernetes集群中的计算能力由Node提供，最初Node称为服务节点Minion，后来改名为Node。Kubernetes集群中的Node也就等同于Mesos集群中的Slave节点，是所有Pod运行所在的工作主机，可以是物理机也可以是虚拟机。不论是物理机还是虚拟机，工作主机的统一特征是上面要运行kubelet管理节点上运行的容器。

密钥对象（Secret）

Secret是用来保存和传递密码、密钥、认证凭证这些敏感信息的对象。使用Secret的好处是可以避免把敏感信息明文写在配置文件里。在Kubernetes集群中配置和使用服务不可避免的要用到各种敏感信息实现登录、认证等功能，例如访问AWS存储的用户名密码。为了避免将类似的敏感信息明文写在所有需要使用的配置文件中，可以将这些信息存入一个Secret对象，而在配置文件中通过Secret对象引用这些敏感信息。这种方式的好处包括：意图明确，避免重复，减少暴漏机会。

用户帐户（User Account）和服务帐户（Service Account）

顾名思义，用户帐户为人提供账户标识，而服务账户为计算机进程和Kubernetes集群中运行的Pod提供账户标识。用户帐户和服务帐户的一个区别是作用范围；用户帐户对应的是人的身份，人的身份与服务的namespace无关，

所以用户账户是跨namespace的；而服务帐户对应的是一个运行中程序的身份，与特定namespace是相关的。

命名空间（Namespace）

命名空间为Kubernetes集群提供虚拟的隔离作用，Kubernetes集群初始有两个命名空间，分别是默认命名空间default和系统命名空间kube-system，除此以外，管理员可以可以创建新的命名空间满足需要。

RBAC访问授权

Kubernetes在1.3版本中发布了alpha版的基于角色的访问控制（Role-based Access Control，RBAC）的授权模式。相对于基于属性的访问控制（Attribute-based Access Control，ABAC），RBAC主要是引入了角色（Role）和角色绑定（RoleBinding）的抽象概念。在ABAC中，Kubernetes集群中的访问策略只能跟用户直接关联；而在RBAC中，访问策略可以跟某个角色关联，具体的用户在跟一个或多个角色相关联。显然，RBAC像其他新功能一样，每次引入新功能，都会引入新的API对象，从而引入新的概念抽象，而这一新的概念抽象一定会使集群服务管理和使用更容易扩展和重用。

总结

从Kubernetes的系统架构、技术概念和设计理念，我们可以看到Kubernetes系统最核心的两个设计理念：一个是**容错性**，一个是**易扩展性**。容错性实际是保证Kubernetes系统稳定性和安全性的基础，易扩展性是保证Kubernetes对变更友好，可以快速迭代增加新功能的基础。

按照分布式系统一致性算法Paxos发明人计算机科学家Leslie Lamport的理念，一个分布式系统有两类特性：安全性Safety和活性Liveness。安全性保证系统的稳定，保证系统不会崩溃，不会出现业务错误，不会做坏事，是严格约束的；活性使得系统可以提供功能，提高性能，增加易用性，让系统可以在用户“看到的时间内”做些好事，是尽力而为的。Kubernetes系统的设计理念正好与Lamport安全性与活性的理念不谋而合，也正是因为Kubernetes在引入功能和技术的时

候，非常好地划分了安全性和活性，才可以让Kubernetes能有这么快版本迭代，快速引入像RBAC、Federation和PetSet这种新功能。