9Kubernetes系列（十一）10 种 Deployment 的反模式（不恰当的配置）

Kubernetes系列（十一）10 种 Deployment 的反模式（不恰当的配置）

随着容器技术采用和市场占有率的不断增加，Kubernetes (K8s)已经成为容器编排的领先平台。它是一个开源项目，有来自超过315家公司的数万名贡献者，他们的目的是保持可扩展性以及不依赖任何的云平台，它是大多数主流云供应商的基础设施。

当您在生产环境中运行容器时，您希望您的生产环境尽可能地稳定和具有弹性，以避免灾难(想想每一次黑色星期五的在线购物体验)。当一个容器挂掉时，另一个需要运行起来取代它，不管它是白天的什么时候，或者是深夜的凌晨。Kubernetes提供了一个框架，用于灵活地运行分布式系统，从扩展到故障转移到负载平衡等等。还有许多与Kubernetes集成的工具可以帮助满足您的需求。

最佳实践随着时间的推移而发展，所以不断地研究和试验Kubernetes开发的更好方法总是好的。由于它仍然是一项年轻的技术，我们一直在寻求提高我们对它的理解和使用。

在本文中，我们将研究Kubernetes部署中的10种常见实践，它们在较高的层次上具有更好的解决方案。我不会深入探讨最佳实践，因为自定义实现可能因用户而异。

下面是 10项错误的用法：

1. 把配置文件放在 Docker 镜像的内部
2. 不使用 Helm 或其他类型的模板工具
3. 以特定的顺序部署程序。(应用程序不应该因为依赖项没有准备好而崩溃。)
4. 部署 PODS 时没有设置 cpu 以及内存的限制
5. 在生产环境的容器中使用 `latest` 镜像标签
6. 通过杀死 pod 来部署新的更新/修复版本，以便它们在重启过程中拉取新的 Docker 镜像
7. 在同一个集群中混合部署生产和非生产环境。
8. 在核心 deployment 中不使用蓝绿或金丝雀部署方案。(Kubernetes的默认滚动更新并不总是满足要求的。)
9. 没有适当的指标来监控部署是否成功。(您的健康检查需要应用程序支持。)
10. 云供应商锁定:将 自己锁定在IaaS供应商的Kubernetes或 serverless 计算服务上

10项 Kubernetes 反例

1. 将配置文件放在 Docker 镜像里面

容器为开发人员提供了一种使用单个镜像的方法，贯穿从开发/QA到 staging 再到生产的整个软件生命周期。

然而，一种常见的做法是为生命周期中的每个阶段都构建其自己的镜像，每个阶段都使用特定于其环境(QA、staging 或生产)的不同工件构建。

不要在构建时进行硬编码配置。

这里的最佳实践是将通用配置外化到ConfigMaps中，而敏感信息(如API keys 和 secrets) 可以存储在Secrets 资源中(该资源具有Base64编码，但在其他方面与ConfigMaps相同)。ConfigMaps可以作为卷挂载，也可以作为环境变量传入，但是 Secrets 应该作为卷挂载。我提到 ConfigMaps和Secrets是因为它们是Kubernetes的原生资源，不需要集成，但是它们可能会受到限制。

还有其他的解决方案，如由HashiCorp 维护的ZooKeeper和Consul ConfigMap，或由HashiCorp 提供的 Vault，Keywhiz，confidant等，可能更适合您的需要。

当你将配置与应用程序解耦后，当你需要更新配置时，你不再需要重新编译应用程序——它可以在应用程序运行时更新。您的应用程序在运行时获取配置而不是在构建期间。更重要的是，您在软件生命周期的所有阶段都使用相同的源代码。

2. 不使用 Helm 或其他类型的模板工具

您可以通过直接更新 YAML 来管理Kubernetes部署。当推出一个新版本的代码时，你可能需要更新以下一个或多个:

• Docker 镜像名称

• Docker 镜像标签

• 运行副本数

• Service 标签

• pods

• Configmaps 等

如果要管理多个集群，并在开发、Staging 和生产环境中应用相同的更新，这可能会很乏味。在所有部署中，您基本上都是在修改相同的文件，并进行少量修改。这需要进行大量的复制和粘贴，或者搜索和替换，同时还需要了解部署YAML所要用于的环境。在这个过程中有很多犯错的机会:

• 拼写错误(版本号错误，镜像名称拼写错误等)

• 使用错误的更新修改YAML(例如，连接到错误的数据库)

• 遗漏更新某个资源等

在YAML中可能需要更改许多东西，如果不仔细注意，一个YAML很容易被误认为是另一个部署的YAML。

模板帮助简化Kubernetes应用程序的安装和管理。由于Kubernetes没有提供原生模板机制，我们必须在其他地方寻找这种类型的管理。

Helm 是第一个可用的包管理器(2015年)。它被宣称为"Kubernetes的自制程序"，并逐步发展到包括模板功能。Helm 通过 charts 将其资源打包，其中 charts 是描述Kubernetes 资源相关文件集合。在chart 仓库（ Artifact Hub ）中有1400多个公开可用的 Chart(你也可以使用 `[helm search hub](https://helm.sh/docs/helm/helm_search_hub/) [keyword] [flags]` Cli 命令进行搜索)，基本上是 Kubernetes上安装、升级和卸载应用的可重用的实现。通过Helm charts，您可以修改 `[values.yaml](https://helm.sh/docs/chart_template_guide/values_files/)` 文件来设置Kubernetes 部署程序所需的修改，并且您可以为每个环境拥有不同的 Helm Chart。因此，如果您有一个QA、Staging和生产环境，您只需要管理三个 Helm Chart，而不是在每个环境中的每个部署中修改每个YAML。

我们使用 Helm 的另一个优势是，如果出现问题，使用 Helm 很容易回滚到之前的版本:

`helm rollback <RELEASE> [REVISION] [flags]` .

如果你想回滚到之前的版本，你可以使用:

`helm rollback <RELEASE> 0` .

所以我们会看到这样的情况:

```
$ helm upgrade – install – wait – timeout 20 demo demo/
$ helm upgrade – install – wait – timeout 20 – set
readinessPath=/fail demo demo/
$ helm rollback – wait – timeout 20 demo 1
```

并且Helm chart 记录了历史的版本:

```
$ helm history demo
REVISION STATUS DESCRIPTION
1 SUPERSEDED Install complete
2 SUPERSEDED Upgrade “demo” failed: timed out waiting for the condition
3 DEPLOYED Rollback to 1
```

另外谷歌的 Kustomize（ `https://kustomize.io/` ） 工具，是除了 Helm 之外的比较好的选择。

3. 以特定的顺序部署程序

应用程序不应该因为依赖项没有准备好而崩溃。在传统开发中，当启动应用程序时，启动和停止任务有一个特定的顺序。重要的是，不要将这种思维引入容器编排。在Kubernetes、Docker等平台上，这些组件可以同时启动，因此无法定义启动顺序。即使当应用程序启动并运行时，它的依赖项也可能失败或被迁移，从而导致进一步的问题。Kubernetes的实际情况也充满了无数潜在的通信失败点，在这些地方无法达到依赖关系，在此期间，PODS 可能会崩溃或服务可能不可用。网络延迟，比如微弱的信号或中断的网络连接，是导致通信失败的常见原因。

为了简单起见，让我们研究一个假设的购物应用程序，它有两个服务:一个库存数据库和一个店面UI。在应用程序可以启动之前，后端服务必须启动，满足其所有检查并开始运行。然后，前端服务可以启动，满足其检查并开始运行。

假设我们使用 `[kubectl wait](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands)` 命令强制执行部署顺序，如下所示:

```
kubectl wait – for=condition=Ready pod/serviceA
```

但是，当这个条件永远不满足时，下一个部署就无法进行，流程就会中断。

以下是部署按照顺序运行的一个简单流程:

这个过程不能继续，直到上一步完成。

因为Kubernetes是自愈的。标准的方法是让应用程序中的所有服务并发启动，让容器崩溃并重新启动，直到它们全部启动并运行为止。我让服务A和B独立启动(一个解耦的、无状态的云本机应用应该这样)，但为了用户体验，也许我可以告诉UI(服务B)显示一个漂亮的加载消息，直到服务A准备好，但服务B的实际启动不应该受到服务A的影响。

现在，当pod崩溃时，Kubernetes重新启动服务，直到一切都启动并运行。如果您陷入CrashLoopBackOff，那么就有必要检查您的代码、配置或资源争用。

当然，我们需要做的不仅仅是依靠自我修复。我们需要实施应对失败的解决方案，这些失败是不可避免的，也会发生。我们应该预料到它们会发生，并制定框架以帮助我们避免停机或数据丢失的方式进行响应。

在我假设的购物应用程序中，我的店面UI(服务B)需要库存(服务A)才能给用户一个完整的体验。因此，当出现部分故障时，比如服务 A 短时间内不可用或崩溃等，系统应该仍然能够从问题中恢复。

像这样的短暂故障总是存在的，所以为了最小化它们的影响，我们可以实现一个 Retry模式 。Retry 模式可以通过以下策略帮助提高应用程序的稳定性:

- **取消** 如果故障不是暂时的或如果过程不可能成功的重复尝试，那么应用程序应该取消操作和报告一个异常-例如，认证失败。无效凭证永远不能工作!
- **重试** 如果故障是不寻常的或罕见的，它可能是由于不寻常的情况(例如，网络数据包损坏)。应用程序应该立即重试请求，因为相同的失败不太可能再次发生。
- **延迟后重试** 如果故障是由常见的连接或繁忙故障引起的，最好在再次尝试之前让任何工作积压或流量清除。应用程序在重试请求之前应该等待。
- 你也可以使用 指数后退 来实现你的重试模式(指数增加等待时间和设置最大重试次数)。

在创建弹性微服务应用程序时，实现断路模式也是一个重要的策略。就像您家里的断路器会自动切换以保护您免受过量电流或短路造成的广泛损害一样，断路器模式为您提供了一种编写应用程序的方法，同时限制了可能需要更长时间来修复的意外故障的影响，如部分连接丢失或服务完全故障。在重试不起作用的情况下，应用程序应该能够接受已经发生的失败并作出相应的响应。

4. 部署 PODS 时没有设置 cpu 以及内存的限制

资源分配随服务的不同而不同，如果不测试实现，很难预测容器可能需要哪些资源来实现最佳性能。一个服务可能需要固定的CPU和内存消耗配置，而另一个服务的消耗配置可能是动态的。

如果部署 pods 时没有仔细考虑内存和CPU限制，这可能会导致资源争用和环境不稳定的情况。如果容器没有内存或CPU限制，那么调度器将其内存利用率(和CPU利用率)视为零，因此可以在任何节点上调度无限数量的pods。这可能会导致资源的过度使用和可能的节点和kubelet崩溃。

当没有为容器指定内存限制时，有几个场景可以适用(这些场景也适用于CPU):

1. 容器可以使用的内存量没有上限。因此，容器可以使用其节点上的所有可用内存，可能会调用OOM(内存不足)Killer。对于没有资源限制的容器来说，发生OOM Kill情况的可能性更大。

2. 命名空间(容器在其中运行)的默认内存限制被分配给容器。集群管理员可以使用LimitRange来指定内存限制的默认值。

声明集群中容器的内存和CPU限制允许您有效地利用集群节点上的可用资源。这有助于kube-scheduler确定pod应该驻留在哪个节点上，以获得最有效的硬件利用率。

在为容器设置内存和CPU限制时，应该注意不要将 request 设置超过 limit。对于拥有多个容器的pod，聚合的资源请求不能超过设置的限制,否则，pod将永远不会被调度。

资源 request 不能超过 limit

设置内存和CPU requests 低于它们的 limits 可以保证两件事:

1. 当内存/CPU可用时，pod可以充分利用它。

2. 在使用率变高时，pod被限制在合理的内存/CPU数量。

最佳实践是将CPU请求保持在一个核心或更低的位置，然后使用ReplicaSets向外扩展，这为系统提供了灵活性和可靠性。

在同一个集群中部署容器时，如果有不同的团队争夺资源，会发生什么情况?如果进程超过了内存限制，那么它将被终止，而如果它超过了CPU限制，进程将被限制(导致更糟糕的性能)。

您可以通过名称空间设置中的 resource quotas 和 LimitRange 控制资源限制。这些设置有助于解决容器部署没有限制或资源请求高的问题。

5. 在生产环境的容器中使用 `latest` 镜像标签

使用 `latest` 标签被认为是不好的做法，尤其是在生产中。Pods会因为各种原因意外地崩溃，所以它们可以在任何时候拉取图像。不幸的是， `latest` 标签在确定构建何时崩溃时并不是很有描述性。运行的镜像版本是什么?上次有用是什么时候?这在生产环境中尤其糟糕，因为您需要能够在最小的停机时间内恢复和运行。

默认情况下， `imagePullPolicy` 被设置为 `Always` ，并且在重新启动时总是拉取图像。如果不指定标签，Kubernetes 将默认使用 `latest` 。但是，只有在发生崩溃(当pod重启时拉取镜像)或部署pod的模板(.spec.template)发生更改时，才会更新部署。

即使您已经将 `imagePullPolicy` 更改为另一个值而不是 `Always` ，您的pod仍然会在需要重启时拉取一个图像(无论是由于崩溃还是故意重启)。如果您使用版本控制并使用有意义的标记(如v1.4.0)设置 `imagePullPolicy` ，那么您可以回滚到最近的稳定版本，并且更容易地排除代码中出现错误的时间和位置。

除了使用特定的、有意义的Docker 标签之外，你还应该记住容器是无状态和不可变的。它们也是短暂的(您应该将容器外的任何数据存储在持久存储中)。一旦你启动了一个容器，你就不应该修改它:没有补丁，没有更新，没有配置改变。当您需要更新配置时，您应该使用更新后的配置部署一个新容器。

这种不可变性允许更安全和可重复的部署。如果需要重新部署旧镜像，还可以更容易地回滚。通过保持Docker镜像和容器不可变，可以在每个环境中部署相同的容器镜像。

在进行故障排除时，我们可以回滚到以前的稳定版本。

6. 通过杀死 pod 来部署新的更新/修复版本，以便它们在重启过程中拉取新的 Docker 镜像

就像依赖于 latest 的标签去获取更新一样，依赖于杀死 pod 去推出新的更新也是一种糟糕的做法，因为你并没有对你的代码进行版本控制。如果你想在生产中通过杀死 pod 来获取更新的Docker 镜像，不要这样做。一旦一个版本在生产中发布，它就不应该被重写。如果某些东西损坏了，那么当您在进行故障排除时需要回滚代码时，您将不知道哪里或什么时候出了问题，以及需要回滚到什么时候。

另一个问题是，重新启动容器以拉取一个新的Docker镜像并不总是有效。“当且仅当Deployment的Pod模板(即' .spec.template ')被更改时，例如模板的标签或容器镜像被更新时，部署的rollout被触发。其他更新，例如扩展部署数量，不会触发rollout。”

你必须修改 `.spec.template` 来触发部署。

更新你的pod以获得新的Docker镜像的正确方法是通过 Version控制，然后修改部署规范，以反映一个有意义的标签(不是' latest '，参见反模式5)，以进一步讨论，但如v1.4.0的新版本或v1.4.1的补丁)Kubernetes将在零停机时间触发升级。

7. 在同一个集群中混合部署生产和非生产环境

Kubernetes支持 命名空间 特性，允许用户在同一个物理集群中管理不同的环境(虚拟集群)。可以将命名空间视为在单个物理集群上管理不同环境的一种经济有效的方法。例如，您可以在同一个集群中运行 Staging 环境和生产环境，从而节省资源和资金。然而，在开发中运行Kubernetes和在生产中运行Kubernetes之间有很大的差距。

在同一个集群上混合使用生产和非生产工作负载时，需要考虑很多因素。首先，您必须考虑资源限制，以确保生产环境的性能不受

影响(常见的做法可能是不对生产名称空间设置配额，而对任何非生产名称空间设置配额)。

你还需要考虑隔离。开发人员需要比生产环境更多的访问和权限，您可能希望尽可能地锁定这些权限。虽然命名空间相互隐藏，但默认情况下它们并不是完全隔离的。这意味着您在开发命名空间中的应用程序可以调用测试、Staging 或 生产中的应用程序(或反之亦然)，这被认为不是良好的实践。当然，您可以使用NetworkPolicies来设置隔离命名空间的规则。

但是，全面测试资源限制、性能、安全性和可靠性非常耗时，因此不建议将生产工作负载与非生产工作负载运行在同一个集群中。与其将生产工作负载和非生产工作负载混合在同一个集群中，不如使用单独的集群进行开发/测试/生产—这样可以获得更好的隔离和安全性。您还应该将CI/CD 推广尽可能多地自动化，以减少人为错误的机会。您的生产环境需要尽可能稳定。

8. 在核心 deployment 中不使用蓝绿或金丝雀部署方案

许多现代应用程序都有频繁的部署，从一个月内的多次更改到一天内的多次部署。这在微服务架构中当然是可以实现的，因为不同的组件可以在不同的周期中开发、管理和发布，只要它们一起工作，无缝执行。当然，在推出更新时，保持应用24*7运行显然很重要。

Kubernetes的默认滚动更新并不总是可用的。执行更新的一个常见策略是使用默认的Kubernetes 滚动更新 特性:

```
.spec.strategy.type==RollingUpdate
```

可以设置maxUnavailable(不可用的pods的百分比或数量)和maxSurge字段(可选)来控制滚动更新过程。当适当地实现时，滚动更新允许逐步更新，而无需停机，因为pods是增量更新的。

但是，一旦您将部署更新到下一个版本，就不太容易返回。您应该有一个计划，以便在生产中发生故障时回滚它。当pod更新到下一个版本时，部署将创建一个新的ReplicaSet。而Kubernetes将存储以前的ReplicaSets(默认情况下，是10个，但你可以通过 `spec.revisionHistoryLimit` 更改)。副本集以随机顺序保存在 `app6ff34b8374` 等名称下，并且您不会在部署应用程序YAML中找到对副本集的引用。你可以用以下语句找到它:

```
ReplicaSet.metatada.annotation
```

并通过下面的命令查看 revision:

```
kubectl get replicaset app-6ff88c4474 -o yaml
```

查找版本号。这变得复杂，因为rollout历史不会保留日志，除非你在YAML资源中留下一个注释(你可以用 `– record` 标志做:

当有数十、数百甚至数千个部署同时进行更新时，很难同时跟踪它们。如果您存储的修订都包含相同的回归，那么您的生产环境将不会处于良好的状态.

其他一些问题是:

• 并非所有应用程序都能同时运行多个版本。

• 您的集群可能在更新过程中耗尽资源，这可能会中断整个进程。

在生产环境中，这些都是非常令人沮丧和有压力的问题。

更可靠的更新部署的替代方法包括:

• *蓝绿(红/黑)部署***

使用蓝绿时，旧实例和新实例的完整集合同时存在。蓝色是活动版本，新版本部署到绿色副本。当绿色环境通过测试和验证后，负载均衡器将流量切换到绿色环境，绿色环境变成蓝色环境，旧版本变成绿色版本。因为我们维护了两个完整的版本，所以执行回滚

很简单—您所需要做的就是切换回负载均衡器。

额外的优势包括:

• 因为我们从来没有直接部署到生产中，所以当我们把绿色变成蓝色的时候，压力是相当小的。

• 流量重定向立即发生，所以没有停机。

• 在切换前可以进行大量的测试以反映实际生产情况。(如前所述，开发环境与生产环境非常不同。)

Kubernetes没有将蓝绿部署作为其原生工具之一。您可以阅读更多关于如何在CI/CD自动化中实现蓝绿的内容（https://codefresh.io/kubernetes-tutorial/fully-automated-blue-green-deployments-kubernetes-codefresh/）。

金丝雀版本允许我们在影响整个生产系统/用户基础之前测试潜在的问题并满足关键指标。我们通过直接部署到生产环境来"在生产中测试"，但只部署到一小部分用户。您可以选择基于百分比的路由，也可以根据地区/用户位置、客户端类型和计费属性来驱动路由。即使在部署到一个小子集时，仔细监视应用程序的性能和测量错误也是很重要的—这些指标定义了质量阈值。如果应用程序的行为符合预期，我们将开始传输更多的新版本实例以支持更多的流量。

其他优点包括:

• 可观测性

• 能够在生产流量上进行测试(在开发中获得真正的生产体验是困难的)

• 能够将一个版本发布给一小部分用户，并在更大的版本发布之前获得真实的反馈

• 快速失败。由于我们直接部署到生产环境中，所以如果它崩溃了，我们可以很快失败(即立即恢复)，而且它只影响一个子集，而不是整个社区。

9. 没有适当的指标来监控部署是否成功

您的健康检查需要应用程序支持。

你可以利用Kubernetes来完成容器编排中的许多任务:

• 控制应用或团队的资源消耗(命名空间、CPU/mem、限制)，防止应用消耗过多的资源

• 实现不同应用实例的负载均衡，在资源短缺或主机死亡时，将应用实例从一台主机转移到另一台主机上

• 自修复-重新启动容器，如果他们崩溃

• 如果集群中增加了新的主机，则自动利用额外的资源

所以有时候我们很容易忘记指标和监控。然而，一个成功的部署并不是您的行动工作的结束。最好主动一点，为意想不到的惊喜做好准备。仍然有更多的层面需要监视，而且K8s的动态特性使故障排除变得困难。例如，如果您没有密切关注可用的资源，pods的自动重新调度可能会导致容量问题，您的应用程序可能会崩溃或永远无法部署。这在生产中尤其不幸，因为除非有人提交bug报告，或者您碰巧检查了它，否则您不会知道。

监测本身也面临一系列挑战:在Kubernetes上运行的应用程序遇到障碍时，需要调查许多日志、数据和组件，特别是当涉及到多个微服务时，而在传统的单体架构中，所有内容都输出到几个日志中。

对应用程序行为(如应用程序如何执行)的洞察有助于不断改进。您还需要一个容器、POD、服务和集群的整体视图。如果您能够确定应用程序是如何使用其资源的，那么您就可以使用Kubernetes来更好地检测和消除瓶颈。为了获得应用程序的完整视图，你需要使用应用程序性能监控解决方案，如 普罗米修斯， Grafana，等等。

无论您是否决定使用监控解决方案，以下是Kubernetes文档建议您密切跟踪的关键指标:

• 运行 POD 及其 deployment

• 资源指标CPU、内存占用率、磁盘I/O

• Container-native指标

• 应用程序指标

10. 云供应商锁定:将 自己锁定在IaaS供应商的Kubernetes或 serverless 计算服务上

供应商锁定否定了部署到云上的主要价值:容器灵活性。的确，选择合适的云提供商不是一个容易的决定。每个提供者都有自己的接口、开放api以及专有规范和标准。此外，一个提供商可能比其他提供商更适合您的需求，只适合您的业务需要发生意外变化。

幸运的是，容器是平台无关的和可移植的，所有主要的提供商都有一个Kubernetes基础，它是云无关的。当您需要在云之间移动工作负载时，您不需要重新架构或重写应用程序代码，因此您不应该因为无法"提升和转移"而将自己锁定在云提供商中。

以下是您应该考虑的事项列表，以确保您能够灵活地防止或最小化供应商锁定。

谈判进入和退出策略。许多供应商都很容易上手，让你上瘾。这可能包括免费试用或积分等激励措施，但随着规模的扩大，这些成本可能会迅速增加。

检查诸如自动续订、提前终止费用等事项，以及在迁移到另一个供应商时，提供商是否会帮助解决版本问题，以及与退出相关的sla。

如果您已经在为云进行开发并使用本地云原则，那么很可能您的应用程序代码应该很容易提升和转移。正是这些围绕在代码周围的东西可能将您锁定在云供应商中。例如，你可以:

• 检查你的应用程序使用的服务和功能(如数据库，api等)是可移植的。

• 检查您的部署和供应脚本是否特定于云。一些云有自己的本地或推荐的自动化工具，可能不容易转换到其他提供商。有很多工具可以用来辅助云基础设施自动化，并与许多主要的云提供商兼容，比如 Puppet 、 Ansible 和 Chef 。

• 检查你的DevOps环境(通常包括Git和CI/CD)是否可以运行在任何云环境中。例如，许多云都有自己特定的CI/CD工具，如 IBM Cloud ContinuousDelivery 、 Azure CI/CD 或 AWS Pipelines ，这可能需要额外的工作来移植到另一个云供应商。相反，你可以使用像 Codefresh 这样的完整的CI/CD解决方案，它对Docker和Kubernetes有很大的支持，并集成了许多其他流行的工具。还有无数其他的解决方案，一些CI或CD，或两者兼有，如 GitLab ， Bamboo ， Jenkins ， Travis 等。

• 检查你的测试过程是否需要在不同的供应商之间改变。

您也可以选择遵循 多云策略

使用多云策略，您可以从不同的云提供商中挑选最适合您希望交付的应用程序类型的服务。在计划多云部署时，应该仔细考虑互操作性。

总结

Kubernetes非常受欢迎，但是开始比较难，而且在传统开发中有很多实践无法转化为云本地开发。

在本文中，我们研究了:

1. 外部化你的配置数据：你可以使用ConfigMaps和Secrets或者其他类似的东西

2. 不使用Helm或其他类型的模板: 使用Helm或Kustomize来简化你的容器编排和减少人为错误

3. 按照特定的顺序部署:应用程序不应该因为依赖项没有准备好而崩溃。利用Kubernetes的自愈机制实现重试和断路器

4. 部署pod时不设置内存和/或CPU限制: **您应该考虑设置内存和CPU限制以减少资源争用的风险，特别是在与他人共享集群时。**

5. 在生产的容器中拉取' latest '标签: **永远不要使用** ' latest '。总是使用一些有意义的东西，如v1.4.0/根据 语义版本规范 ，并使用不可变的Docker镜像

6. 通过杀死pod来部署新的更新/修复，这样它们就会在重启过程拉取新的Docker镜像: **版本化你的代码，这样你就可以更好地管理你的发布**

7. 在同一个集群中混合使用生产和非生产工作负载: **如果可能的话，在单独的集群中运行生产和非生产工作负载。这减少了生产环境中资源争用和意外的环境交叉带来的风险**

8. 在关键任务部署中不使用蓝绿或金丝雀版本(Kubernetes的默认滚动更新并不总是足够): **你应该考虑蓝绿部署或金丝雀版本，以减少生产中的压力和更有意义的生产结果**

9. 没有适当的指标来了解部署是否成功(您的健康检查需要应用程序支持): **您应该确保监视您的部署，以避免任何意外。你可以使用像Prometheus, Grafana, New Relic或Cisco AppDynamics这样的工具来帮助你更好地了解你的部署**

10. 云供应商锁定:将自己锁定在IaaS供应商的Kubernetes或无服务器计算服务中: **您的业务需求随时可能改变** 。你不应该无意中把自己锁定在云提供商上，因为你可以很容易地提升和转移云本地应用

感谢你的阅读!

欢迎关注我的公众号“ **云原生拓展** ”，原创技术文章第一时间推送。