Kubernetes 部署失败的 10 个最普遍原因（二）

上篇文章我写了 Kubernetes 部署失败的前5个最普遍原因。本文是剩下的几个，其中有几个特别令人沮丧。

6. 资源配额

和资源限额类似，Kubernetes 也允许管理员给每个 namespace 设置资源配额。这些配额可以在 Pods，Deployments，PersistentVolumes，CPU，内存等资源上设置软性或者硬性限制。

让我们看看超出资源配额后会发生什么。以下是我们的 deployment 例子：

```
# test-quota.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
name: gateway-quota
spec:
template:
spec:
  containers:
    - name: test-container
      image: nginx
```

我们可用 kubectl create -f test-quota.yaml 创建，然后观察我们的 Pods：

```
$ kubectl get pods
NAME                    READY     STATUS    RESTARTS   AGE
```

```
gateway-quota-551394438-pix5d   1/1       Running   0          16s
```

看起来很好，现在让我们扩展到 3 个副本：kubectl scale deploy/gateway-quota --replicas=3，然后再次观察 Pods：

```
$ kubectl get pods
NAME                             READY     STATUS    RESTARTS   AGE
gateway-quota-551394438-pix5d   1/1       Running   0          9m
```

啊，我们的pod去哪了？让我们观察一下 deployment：

```
$ kubectl describe deploy/gateway-quota
Name:           gateway-quota
Namespace:      fail
CreationTimestamp:    Sat, 11 Feb 2017 16:33:16 -0500
Labels:         app=gateway
Selector:       app=gateway
Replicas:       1 updated | 3 total | 1 available | 2 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:    1 max unavailable, 1 max surge
OldReplicaSets:
NewReplicaSet:      gateway-quota-551394438 (1/3 replicas created)
Events:
FirstSeen     LastSeen     Count    From                    SubObjectPath    Type
 Reason         Message
---------    --------    -----   ----            -------------   --------    ------        ---
----
9m            9m          1     {deployment-controller }                  Normal
 ScalingReplicaSet   Scaled up replica set gateway-quota-551394438 to 1
```

```
5m 5m 1 {deployment-controller } Normal
ScalingReplicaSet Scaled up replica set gateway-quota-551394438 to 3
```

在最后一行，我们可以看到 ReplicaSet 被告知扩展到 3 。我们用 describe 来观察一下这个 ReplicaSet 以了解更多信息：

```
kubectl describe replicaset gateway-quota-551394438
Name: gateway-quota-551394438
Namespace: fail
Image(s): nginx
Selector: app=gateway,pod-template-hash=551394438
Labels: app=gateway
  pod-template-hash=551394438
Replicas: 1 current / 3 desired
Pods Status: 1 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
FirstSeen LastSeen Count From SubObjectPath Type
Reason Message
--------- -------- ----- ---- ------------- -------- ------ ---
----
11m 11m 1 {replicaset-controller } Normal SuccessfulCreate
  Created pod: gateway-quota-551394438-pix5d
11m 30s 33 {replicaset-controller } Warning FailedCreate
E
```

哦！我们的 ReplicaSet 无法创建更多的 pods 了，因为配额限制了：exceeded quota: compute-resources, requested: pods=1, used: pods=1, limited: pods=1。

和资源限额类似，我们也有 3 个选项：

1. 要求集群管理员提升该 namespace 的配额

2. 删除或者收缩该 namespace 下其它的 deployment

3. 直接编辑配额

7. 集群资源不足

除非你的集群开通了集群自动伸缩功能，否则总有一天你的集群中 CPU 和内存资源会耗尽。

这不是说 CPU 和内存被完全使用了，而是指它们被 Kubernetes 调度器完全使用了。如同我们在第 5 点看到的，集群管理员可以限制开发者能够申请分配给 pod 或者容器的 CPU 或者内存的数量。聪明的管理员也会设置一个默认的 CPU/内存 申请数量，在开发者未提供申请额度时使用。

如果你所有的工作都在 default 这个 namespace 下工作，你很可能有个默认值 100m 的容器 CP U申请额度，对此你甚至可能都不清楚。运行 kubectl describe ns default 检查一下是否如此。

我们假定你的 Kubernetes 集群只有一个包含 CPU 的节点。你的 Kubernetes 集群有 1000m 的可调度 CPU。

当前忽略其它的系统 pods（kubectl -n kube-system get pods），你的单节点集群能部署 10 个 pod（每个 pod 都只有一个包含 100m 的容器）。

10 Pods * (1 Container * 100m) = 1000m == Cluster CPUs

当你扩大到 11 个的时候，会发生什么?

下面是一个申请 1CPU（1000m）的 deployment 例子：

```
# cpu-scale.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
name: cpu-scale
spec:
template:
metadata:
  labels:
    app: cpu-scale
spec:
  containers:
    - name: test-container
      image: nginx
      resources:
        requests:
          cpu: 1
```

我把这个应用部署到有 2 个可用 CPU 的集群。除了我的 cpu-scale 应用，Kubernetes 内部服务也在消耗 CPU 和内存。

我们可以用 kubectl create -f cpu-scale.yaml 部署这个应用，并观察 pods：

```
$ kubectl get pods
NAME                         READY     STATUS    RESTARTS   AGE
cpu-scale-908056305-xstti    1/1       Running   0          5m
```

第一个 pod 被调度并运行了。我们看看扩展一个会发生什么：

```
$ kubectl scale deploy/cpu-scale --replicas=2
deployment "cpu-scale" scaled
$ kubectl get pods
NAME                         READY     STATUS    RESTARTS   AGE
cpu-scale-908056305-phb4j    0/1       Pending   0          4m
cpu-scale-908056305-xstti    1/1       Running   0          5m
```

我们的第二个pod一直处于 Pending，被阻塞了。我们可以 describe 这第二个 pod 查看更多的信息：

```
$ kubectl describe pod cpu-scale-908056305-phb4j
Name:        cpu-scale-908056305-phb4j
Namespace:    fail
Node:        gke-ctm-1-sysdig2-35e99c16-qwds/10.128.0.4
Start Time:    Sun, 12 Feb 2017 08:57:51 -0500
Labels:        app=cpu-scale
    pod-template-hash=908056305
Status:        Pending
IP:
Controllers:    ReplicaSet/cpu-scale-908056305
[...]
Events:
FirstSeen    LastSeen    Count    From            SubObjectPath    Type        Reason
        Message
---------    --------    -----    ----            -------------    --------    ------        ------
-
3m        3m        1    {default-scheduler }            Warning     FailedScheduling
 pod (cpu-scale-908056305-phb4j) failed to fit in any node
```

```
fit failure on node (gke-ctm-1-sysdig2-35e99c16-wx0s): Insufficient cpu
fit failure on node (gke-ctm-1-sysdig2-35e99c16-tgfm): Insufficient cpu
fit failure on node (gke-ctm-1-sysdig2-35e99c16-qwds): Insufficient cpu
```

好吧，Events 模块告诉我们 Kubernetes 调度器（default-scheduler）无法调度这个 pod 因为它无法匹配任何节点。它甚至告诉我们每个节点哪个扩展点失败了（Insufficient cpu）。

那么我们如何解决这个问题？如果你太渴望你申请的 CPU/内存 的大小，你可以减少申请的大小并重新部署。当然，你也可以请求你的集群管理员扩展这个集群（因为很可能你不是唯一一个碰到这个问题的人）。

现在你可能会想：我们的 Kubernetes 节点是在我们的云提供商的自动伸缩群组里，为什么他们没有生效呢?

原因是，你的云提供商没有深入理解 Kubernetes 调度器是做啥的。利用 Kubernetes 的集群自动伸缩能力允许你的集群根据调度器的需求自动伸缩它自身。如果你在使用 GCE，集群伸缩能力是一个 beta 特性。

8. 持久化卷挂载失败

另一个常见错误是创建了一个引用不存在的持久化卷（PersistentVolumes）的 deployment。不论你是使用 PersistentVolumeClaims（你应该使用这个！），还是直接访问持久化磁盘，最终结果都是类似的。

下面是我们的测试 deployment，它想使用一个名为 my-data-disk 的 GCE 持久化卷：

```
# volume-test.yaml
apiVersion: extensions/v1beta1
```

```
kind: Deployment
metadata:
name: volume-test
spec:
template:
metadata:
  labels:
    app: volume-test
spec:
containers:
    - name: test-container
      image: nginx
      volumeMounts:
      - mountPath: /test
        name: test-volume
  volumes:
  - name: test-volume
    # This GCE PD must already exist (oops!)
    gcePersistentDisk:
      pdName: my-data-disk
      fsType: ext4
```

让我们创建这个 deployment：kubectl create -f volume-test.yaml，过几分钟后查看 pod：

```
kubectl get pods
NAME                           READY     STATUS              RESTARTS   AGE
volume-test-3922807804-33nux   0/1       ContainerCreating   0          3m
```

3 分钟的等待容器创建时间是很长了。让我们用 describe 来查看这个 pod，看看到底发生了什么：

```
$ kubectl describe pod volume-test-3922807804-33nux
Name:        volume-test-3922807804-33nux
Namespace:    fail
Node:        gke-ctm-1-sysdig2-35e99c16-qwds/10.128.0.4
Start Time:    Sun, 12 Feb 2017 09:24:50 -0500
Labels:        app=volume-test
    pod-template-hash=3922807804
Status:        Pending
IP:
Controllers:    ReplicaSet/volume-test-3922807804
[...]
Volumes:
test-volume:
Type:      GCEPersistentDisk (a Persistent Disk resource in Google Compute Engine)
PDName:    my-data-disk
FSType:    ext4
Partition:    0
ReadOnly:    false
[...]
Events:
FirstSeen    LastSeen    Count    From                    SubObjectPath    Type
 Reason      Message
---------    --------    -----    ----                    -------------    --------    ------
 -------
```

4m 4m 1 {default-scheduler } Normal Scheduled Successfully assigned volume-test-3922807804-33nux to gke-ctm-1-sysdig2-35e99c16-qwds

1m 1m 1 {kubelet gke-ctm-1-sysdig2-35e99c16-qwds} Warning FailedMount Unable to mount volumes for pod "volume-test-3922807804-33nux_fail(e2180d94-f12e-11e6-bd01-42010af0012c)": timeout expired waiting for volumes to attach/mount for pod "volume-test-3922807804-33nux"/"fail". list of unattached/unmounted volumes=[test-volume]

1m 1m 1 {kubelet gke-ctm-1-sysdig2-35e99c16-qwds} Warning FailedSync Error syncing pod, skipping: timeout expired waiting for volumes to attach/mount for pod "volume-test-3922807804-33nux"/"fail". list of unattached/unmounted volumes=[test-volume]

3m 50s 3 {controller-manager } Warning FailedMount Failed to attach volume "test-volume" on node "gke-ctm-1-sysdig2-35e99c16-qwds" with: GCE persistent disk not found: diskName="my-data-disk" zone="us-central1-a"

很神奇！ Events 模块留有我们一直在寻找的线索。我们的 pod 被正确调度到了一个节点（Successfully assigned volume-test-3922807804-33nux to gke-ctm-1-sysdig2-35e99c16-qwds），但是那个节点上的 kubelet 无法挂载期望的卷 test-volume。那个卷本应该在持久化磁盘被关联到这个节点的时候就被创建了，但是，正如我们看到的，controller-manager 失败了：Failed to attach volume "test-volume" on node "gke-ctm-1-sysdig2-35e99c16-qwds" with: GCE persistent disk not found: diskName="my-data-disk" zone="us-central1-a"。

最后一条信息相当清楚了：为了解决这个问题，我们需要在 GKE 的 us-central1-a 区中创建一个名为 my-data-disk 的持久化卷。一旦这个磁盘创建完成，controller-manager 将挂载这块磁盘，并启动容器创建过程。

9. 校验错误

看着整个 build-test-deploy 任务到了 deploy 步骤却失败了，原因竟是 Kubernetes 对象不合法。还有什么比这更让人沮丧的！

你可能之前也碰到过这种错误：

```
$ kubectl create -f test-application.deploy.yaml
error: error validating "test-application.deploy.yaml": error validating data: found in
```

在这个例子中，我尝试创建以下 deployment：

```
# test-application.deploy.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
name: test-app
spec:
template:
metadata:
  labels:
    app: test-app
spec:
  containers:
  - image: nginx
    name: nginx
  resources:
    limits:
```

```
      cpu: 100m
      memory: 200Mi
    requests:
      cpu: 100m
      memory: 100Mi
```

一眼望去，这个 YAML 文件是正确的，但错误消息会证明是有用的。错误说的是 found invalid field resources for v1.PodSpec，再仔细看一下 v1.PodSpec， 我们可以看到 resource 对象变成了 v1.PodSpec 的一个子对象。事实上它应该是 v1.Container 的子对象。在把 resource 对象缩进一层后，这个 deployment 对象就可以正常工作了。

除了查找缩进错误，另一个常见的错误是写错了对象名（比如 peristentVolumeClaim 写成了 persistentVolumeClaim）。这个错误曾经在我们时间很赶的时候绊住了我和另一位高级工程师。

为了能在早期就发现这些错误，我推荐在 pre-commit 钩子或者构建的测试阶段添加一些校验步骤。

例如，你可以：

1. 用 python -c 'import yaml,sys;yaml.safe_load(sys.stdin)' < test-application.deployment.yaml 验证 YAML 格式

2. 使用标识 --dry-run 来验证 Kubernetes API 对象，比如这样：kubectl create -f test-application.deploy.yaml --dry-run --validate=true

重要提醒：校验 Kubernetes 对象的机制是在服务端的校验，这意味着 kubectl 必须有一个在工作的 Kubernetes 集群与之通信。不幸的是，当前 kubectl 还没有客户端的校验选项，但是已经有 issue（kubernetes/kubernetes #29410 和 kubernetes/kubernetes #11488）在跟踪这个缺失的特性了。

10. 容器镜像没有更新

我了解的在使用 Kubernetes 的大多数人都碰到过这个问题，它也确实是一个难题。

这个场景就像下面这样：

1. 使用一个镜像 tag（比如：rosskulinski/myapplication:v1） 创建一个 deployment

2. 注意到 myapplication 镜像中存在一个 bug

3. 构建了一个新的镜像，并推送到了相同的 tag（rosskukulinski/myapplication:v1）

4. 删除了所有 myapplication 的 pods，新的实例被 deployment 创建出了

5. 发现 bug 仍然存在

6. 重复 3-5 步直到你抓狂为止

这个问题关系到 Kubernetes 在启动 pod 内的容器时是如何决策是否做 docker pull 动作的。

在 v1.Container（http://suo.im/5b56d） 说明中，有一个选项 ImagePullPolicy：

> Image pull policy. One of Always, Never, IfNotPresent. Defaults to Always if :latest tag is specified, or IfNotPresent otherwise.

因为我们把我们的镜像 tag 标记为 :v1，默认的镜像拉取策略是 IfNotPresent。Kubelet 在本地已经有一份 rosskukulinski/myapplication:v1 的拷贝了，因此它就不会在做 docker pull 动作了。当新的 pod 出现的时候，它仍然使用了老的有问题的镜像。

有三个方法来解决这个问题：

1. 切成 :latest tag（千万不要这么做！http://suo.im/46ikcS）

2. deployment 中指定 ImagePullPolicy: Always

3. 使用唯一的 tag（比如基于你的代码版本控制器的 commit id）

在开发阶段或者要快速验证原型的时候，我会指定 ImagePullPolicy: Always 这样我可以使用相同的 tag 来构建和推送。

然而，在我的产品部署阶段，我使用基于 Git SHA-1 的唯一 tag。这样很容易查到产品部署的应用使用的源代码。

总结

哇哦，我们有这么多地方要当心。到目前为止，你应该已经成为一个能定位，识别和修复失败的 Kubernetes 部署的专家了。

一般来说，大部分常见的部署失败都可以用下面的命令定位出来：

- kubectl describe deployment/<deployname>
- kubectl describe replicaset/<rsname>
- kubectl get pods
- kubectl describe pod/<podname>
- kubectl logs <podname> --previous

在追求自动化，把我从繁琐的定位工作中解放出来的过程中，我写了一个 bash 脚本，它在 CI/CD 的部署过程中任何失败的时候，都可以跑。在 Jenkins/CircleCI 等的构建输出中，将显示有用的 Kubernetes 信息，帮助开发者快速找到任何明显的问题。

希望你能喜欢这两篇文章！