30 | 应用健康：如何迅速判断业务状态和可用性？

2023-02-15 王炜 来自北京

讲述：王炜

时长 10:53 大小 9.94M

你好，我是王炜。

从这节课开始，我们正式进入云原生架构的全新领域：应用可观测性。

在生产环境下，我们经常会被问到这些问题：业务现在健康吗？线上部署的是什么版本，是在什么时候部署的？它的整体可用性怎么样？是否有报错信息等等。在单体应用架构下，这些问题可能非常容易回答，但是在分布式的微服务架构下，要迅速回答这些问题并不简单。

从概念的定义上来说，可观测性包括三个方面，也就是我们常说的指标、日志和链路。指标主要用于衡量程序性能，通过指标的度量值我们可以判断系统的表现情况。最常见的指标有CPU、内存、磁盘和网络等。日志主要用来统一收集业务输出的日志，包括各种级别的提示警告和错误日志信息等，结合查询系统我们便能够快速定位错误。链路一般指的是分布式追踪，它可以评估一个完整的请求链上微服务的性能情况。

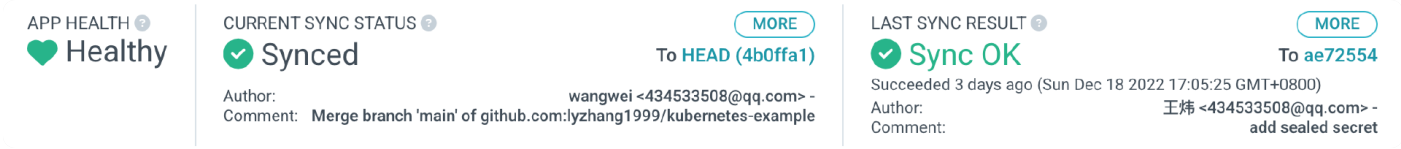
不过，可观测性更多的是深入到微服务内部监控性能和指标，而在实际的业务场景中，当业务出现异常时，我们一般会从外到内排查问题。也就是先检查应用整体的状态和可用性，再借助可观测性工具对应用内每一个微服务进行深入排查。

所以，在正式进入到可观测性的学习之前，这节课，我会先从排查问题的第一原则出发，也就是从最外层的业务应用出发，介绍在 GitOps 场景下判断业务状态、查找故障的几种方法。当你在实践中遇到生产故障时，完全可以把这节课的内容作为故障排查手册来使用。

应用健康状态

对于 Kubernetes 应用而言，它往往会包含不同的 Kubernetes 对象和工作负载，要判断应用是否处于健康状态，等同于判断应用所有的工作负载是否都处于健康状态，这很繁琐并且也不直观。

我之前之所以推荐使用 ArgoCD 来部署应用，其中一个很重要的原因就是它内置了应用级的**“健康状态”**，如下图所示。



ArgoCD 为几种标准的 Kubernetes 对象提供了健康状态的算法，当应用内所有资源都处于健康状态时，就认为应用也处于健康状态，这非常符合我们判断应用状态的标准。

对于 Kubernetes 的工作负载，例如 Deployment、StatefulSet 和 DaemonSet 等，有下面三个判断健康状态的条件。

1. 工作负载处于运行状态。
2. 部署的版本符合期望版本。
3. 实际的副本数符合期望的副本数。

对于 Service 来说，ArgoCD 会检查类型是否为 LoadBalancer ，并判断 status.loadBalancer.ingress 字段是否为空，从而判断是否成功创建了负载均衡器。而对于 Ingress ，则判断 status.loadBalancer.ingress 字段是否为空。

最后，对于持久卷（PVC），ArgoCD 会判断 status.phase 字段是否为 Bound 状态。

所以，要检查业务应用的健康状态，你应该首先查看 ArgoCD 的应用健康状态。如果状态是 Heathy，说明应用运行正常，如果状态是 Degraded，说明应用处于异常状态。

此外，ArgoCD 的 “CURRENT SYNC STATUS” 还能够帮助我们进一步判断集群内所有资源是否已经和仓库进行了同步。Synced 状态表示已经同步完成，Out-Of-Sync 代表集群资源和 Git 仓库有差异，这时候就需要留意产生差异的原因，例如是否手动修改了集群内的资源或者直接在集群内创建了新的资源等。

Pod 健康状态

如果 ArgoCD 的应用状态为 Degraded，代表应用状态异常，这时候就需要进一步排查，并从 Kubernetes 对象开始着手了。

在之前的课程中，我们提到 Pod 是 Kubernetes 调度的最小单位。所以，当工作负载出现故障时，我们首先应该查看 Pod 的状态。

Pod 的状态涉及到启动状态和运行状态，排查问题也相对复杂。接下来，我们通过一个例子来说明一下排查方法。

首先，我们创建用来实验的 Pod，将下面的内容保存为 pod-status.yaml 文件。

复制代码

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: running
  labels:
    app: nginx
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: TCP

---
apiVersion: v1
```

```
kind: Pod
metadata:
  name: backoff
spec:
  containers:
    - name: web
      image: nginx:not-exist

---
apiVersion: v1
kind: Pod
metadata:
  name: error
spec:
  containers:
    - name: web
      image: nginx
      command: ["sleep", "a"]
```

然后，使用 kubectl apply 命令将它应用到集群内。

复制代码

```
$ kubectl apply -f pod-status.yaml
pod/running created
pod/backoff created
pod/error created
```

接下来，使用 kubectl get pods 查看刚才创建的 3 个 Pod。

复制代码

```
NAME                       READY   STATUS             RESTARTS        AGE
backoff                    0/1     ImagePullBackOff   0               50s
error                      0/1     CrashLoopBackOff   1               4s
running                    1/1     Running            0               50s
```

在这个例子中，我们一共创建了 3 个 Pod，它们的状态包含 ImagePullBackOff、CrashLoopBackOff 和 Running。

Running 状态表示运行中，除此之外，剩下两种异常状态分别代表**启动和运行阶段的错误**，它们也是生产环境出现频率最高的错误，接下来我会对它们继续深入分析。

ImagePullBackOff

ImagePullBackOff 是一个典型的**容器启动错误**，除它之外，你可能还会看到下面这几个容器启动阶段的错误：

- ErrImagePull
- ImageInspectError
- ErrImageNeverPull
- RegistryUnavailable
- InvalidImageName

这几种错误出现的概率比较低，并且单纯从字面上也就能大概理解它的含义所以这里我主要介绍 ImagePullBackOff 错误。它可能是下面这两个原因导致的。

1. 镜像名称或者版本错误，在我们刚才创建的 backoff Pod 中，我们指定的镜像为 nginx:not-exist，但是实际上镜像版本 not-exist 并不存在，自然也就会抛出错误。
2. 指定了私有镜像，但又没有提供拉取凭据。

有的时候，你可能看到了错误，但很难立刻找到具体的原因，这里我为你总结一个方法。对于启动阶段的错误，我们可以使用 kubectl describe 命令来查看错误的详情。

```
$ kubectl describe pod backoff
Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  10m                  default-scheduler  Successfully assi
  Normal   Pulling    8m43s (x4 over 10m)  kubelet            Pulling image "ng
  Warning  Failed     8m40s (x4 over 10m)  kubelet            Failed to pull im
  Warning  Failed     8m40s (x4 over 10m)  kubelet            Error: ErrImagePu
  Warning  Failed     8m11s (x6 over 10m)  kubelet            Error: ImagePullB
  Normal   BackOff    4s (x42 over 10m)    kubelet            Back-off pulling
```

从返回结果里 Event 事件中的第三行我们可以发现，集群抛出了 nginx:not-exist: not found 的异常，这样我们也就定位到了具体的错误。

CrashLoopBackOff

CrashLoopBackOff 是一种典型的容器**运行阶段的错误**。除此之外，你可能还会看到类似的 RunContainerError 错误。

出现这个错误的原因主要有下面两个。

1. 容器内的应用程序在启动时出现了错误，例如配置读取失败导致无法启动。
2. 配置出错，例如配置了错误的容器启动命令。

在上面创建的 error Pod 的例子中，我故意错误地配置了容器的启动命令，这样我们也就看到了 CrashLoopBackOff 异常。

对于运行阶段的错误，大部分错误都来源于业务本身的启动阶段，所以，我们只需要查看 Pod 的日志一般就能够找到问题所在。比如，我们尝试来查看 error Pod 的日志。

```
$ kubectl logs error
sleep: invalid time interval 'a'
Try 'sleep --help' for more information.
```

从返回的日志来看，sleep 命令抛出了一个异常，也就是参数错误。在生产环境下，我们一般会用 Deoloyment 工作负载来管理 Pod，在 Pod 出现运行阶段异常的情况下，Pod 名称会随着重新启动而出现变化，这时候你可以在查看日志时增加 --previous 参数，以此查看之前的 Pod 日志。

```
$ kubectl logs pod-name --previous
```

Pending

有时候，你可能不会看到启动和运行的错误状态，但查看状态时，会看到 Pod 处于 Pending 状态。你可以尝试将下面的内容保存为 pending-pod.yaml 文件，并通过 kubectl apppy -f 将这个例子部署到集群内。

复制代码

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pending
spec:
  containers:
    - name: web
      image: nginx
      resources:
        requests:
          cpu: 32
          memory: 64Gi
```

接下来，尝试查看 Pod 的状态。

复制代码

```
$ kubectl get pods
NAME                                READY   STATUS              RESTARTS          AG
pending                             0/1     Pending             0                 15
```

从返回结果我们会发现，Pod 没有抛出任何异常，但它的状态处于 Pending，同时 READY 0/1 表示 Pod 没有准备好接收外部流量。

出现 Pending 状态主要的原因可能有下面三种。

1. 集群资源不足以调度 Pod。
2. Pod 正在等待 PVC 持久化存储卷。
3. Pod 资源用量超过了命名空间的资源配额。

在上面的例子中，我们为 Pod 配置了 32 核 64G 的资源请求配额，这显然超出了集群资源。此时 Pod 会处于 Pending 状态，并且 Kubernetes 会一直尝试调度，一旦加入了新的节点并满足资源要求，Pod 就会被重新启动。

Pending 状态其实也算是容器启动异常的一种情况，但它并不能算是错误，只是暂时无法调度。要查明 Pending 状态的具体原因，你可以参考寻找容器启动错误的方法，通过 kubectl describe 命令来查看。

```
$ kubectl describe pod pending
Events:
  Type     Reason            Age    From               Message
  ----     ------            ----   ----               -------
  Warning  FailedScheduling  11m    default-scheduler  0/1 nodes are available:
  Warning  FailedScheduling  6m45s  default-scheduler  0/1 nodes are available:
```

从返回结果的 Event 事件中我们可以得出结论，Pending 出现的原因是没有符合 CPU 资源条件的 Node 节点，Kubernetes 尝试调度了两次，异常情况相同。

Service 连接状态

有时候，即便是 Pod 处于运行且处于就绪状态，我们也无法从外部请求到业务服务。这时候就要关注 Service 的连接状态了。

Service 是 Kubernetes 的核心组件，正常情况下它都是可用的。在生产环境下，流量的流向一般是从 Ingress 到 Service 再到 Pod。所以，当无法在外部访问到 Pod 的业务服务时，我们可以先从最内层也就是 Pod 开始检查，最简单的方式就是直连 Pod 并发起请求，查看 Pod 是否能够正常工作。

要在本地访问 Pod，我们可以使用 kubectl port-forward 进行端口转发，以我们刚才创建的 Nginx Pod 为例。

```
$ kubectl port-forward pod/running 8081:80
```

如果在本地访问 8081 端口请求能够成功，则代表 Pod 和业务层面是正常的。

接下来，我们进一步检查 Service 的连接状态。同样地，最简单的方式也是通过端口转发直连 Service 发起请求。

```
$ kubectl port-forward service/<service-name> local_port:service_pod
```

如果请求 Service 能够正确返回内容，说明 Service 这一层也是正常的。如果无法返回内容，这时候通常可能有两个原因。

1. Service Selector 选择器没有正确匹配到 Pod。
2. Service 的 Port 和 TargetPort 配置错误。

通过修复这两项配置，你应该就能修复 Service 到 Pod 的连接问题了。

Ingress 连接状态

到这里，如果仍然无法从 Ingress 访问业务服务，那么就需要继续排查 Ingress 了。

首先，确认 Ingress 控制器的 Pod 是否处于运行状态。

```
$ kubectl get pods -n ingress-nginx
NAME                                        READY   STATUS      RESTARTS
ingress-nginx-controller-8544b85788-c9m2g   1/1     Running     6 (4h35m ago)
```

在确认 Ingress 控制器并无异常之后，基本上可以确认是 Ingress 策略配置错误导致的故障了。

你可以通过 kubectl describe ingress 命令来查看 Ingress 策略。

```
$ kubectl describe ingress ingress_name
Name:             ingress_name
Namespace:        default
Rules:
  Host        Path  Backends
  ----        ----  --------
              /     running-service:80 (<error: endpoints "running-service" not
```

如果结果中出现 not found，那么修正配置即可修复 Ingress 访问的问题了。

总结

这节课，我向你简单介绍了可观测性，它们包括指标、日志和链路。不过，在带你实践可观测性之前，我们需要先了解应用状态的判定标准以及排除常见故障的方法。

在判断应用健康状态方面，我们可以借助 ArgoCD 控制台的**“应用健康”**来快速了解业务的可用性。业务应用往往是由很多不同的 Kubernetes 对象组成的，ArgoCD 会对它们进行综合判断得到应用健康状态，这使我们不再需要手动查看每一个资源的状态。

当应用处于不健康的状态时，我们就需要进一步深入到 Kubernetes 对象中查找具体的原因了。

在这么多的工作负载类型中，Pod 的健康状态是我们首先要查看的。它可能在**启动和运行阶段**产生故障，这两个故障类型的典型的代表是 ImagePullBackOff 和 CrashLoopBackOff。根据我的生产经验，这两种故障大部分情况下是镜像名和版本错误，或者业务进程自身启动异常导致的。要查找具体的原因，你只需要记住两条命令（kubectl logs 和 kubectl describe pod ）就可以满足大部分的场景了。

Pod 的 Pending 状态则相对特殊，它并不是错误，而是调度层面的限制导致的。例如资源配额不足或者等待 PVC 绑定等。

最后，如果 Pod 处于运行和就绪状态，但是仍然无法从集群外部访问业务系统，我们就需要进一步查看 Service 和 Ingress 了，这一般是由**配置错误**导致的。通常，你可以先将 Pod 进行端口转发，并尝试在本地直接访问 Pod，如果访问正常，那么就可以进一步排查 Service 和 Ingress 配置，例如查看 Service 选择器和端口配置，以及 Ingress 指向后端 Service 的配置。

在下一节课，我将会介绍可观测性的日志方向，并带你从零搭建一个实时的日志系统。

思考题

最后，给你留一道思考题吧。

如何在不借助 Ingress 外网 IP 的情况下，调试完整的请求链路（从域名 ->Ingress->Service->Pod 完整的链路）？

欢迎你给我留言交流讨论，你也可以把这节课分享给更多的朋友一起阅读。我们下节课见。

分享给需要的人，Ta购买本课程，你将得 18 元

生成海报并分享

赞 1　提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　29丨安全提升：如何解决 GitOps 的秘钥存储问题？

下一篇　31丨日志：如何搭建轻量云原生业务日志系统？

更多课程推荐

精选留言 (1)

写留言

郑海成
2023-02-20 来自北京

我理解问题可以认为和下面的说法一致。如何在本地机房通过域名暴露服务?
1.service和pod配置方法和有外网IP是一样的，就不重复说了
2.Ingress在云厂商环境下会提供外网的LB-VIP，在本地机房环境则没有，需要解决一下。方法想到两个：方法一，Ingress-controller的service由loadbalancer改为nodeport，这样就可以使用集群node的IP替代VIP，缺点是nodeport是一个大端口还可能会变；方法二，使用metall

b等其他lb方案为Ingress-controller service提供VIP这样在大二层网络通的情况下就可以访问了

3.域名解析在云厂商环境是通过其提供的DNS解析到LB-VIP实现，本地机房可以使用基础网络里的自建DNS实现或者host文件实现

作者回复: 两种方法都可以实现暴露对外暴露，metallb 的方案可能会更好一些。