生产环境中的容器运行指南

很多人认为容器是一个热门的新技术，然而事实并非如此，容器可以说是一个古老的技术，其起源可以追溯到 1979 年推出的 chroot 系统调用。多年来，人们一直尝试实现容器架构，直到 2013 年，Docker 最终推出了一套标准化的容器格式。从那时起，由于强大的 Docker 生态和 Kubernetes 的普及，容器才得到广泛应用。

虽然，当下已经有许多人开始接受使用容器，但在将代码库和系统迁移到新的生态系统中时却有诸多顾虑，尤其是无法确保生产环境中容器的安全运行。

关于本报告

本报告主要探讨了在生产环境中运行容器的效益和部署容器化应用的挑战，概述了容器的生态体系以及与之相配套的开发流程和结构。本报告的主要内容如下：

- 构建和管理容器环境
- 创建 CI/CD
- 监控容器
- 确保容器安全

目录

第 1 章：容器和编排工具.....3
1.1 了解微服务和容器.....3
1.2 编排和管理工具.....3
第 2 章：构建和部署容器.....4
2.1 构建镜像.....5
2.2 分发镜像.....5
2.3 实施 CI/CD/CS.....5
2.3.1 持续集成.....5
2.3.2 持续交付.....6
2.3.3 持续部署.....6
2.3.4 持续安全.....6
第 3 章：容器的持续监控.....6
3.1 制定监控流程.....7
3.1.1 确定指标.....7
3.1.2 追踪记录.....7
3.1.3 实时告警.....7
3.1.4 故障排查.....8
3.2 收集数据方式.....8
3.3 数据汇总和分类.....8
第 4 章：确保容器安全性.....9
总结.....12

第 1 章：容器和编排工具

Docker 在 2013 年首次亮相其容器平台，其技术已经非常成熟，完全可以运行生产工作负载相关的应用。Docker 通过打包依赖关系、运行工作负载所需的软件和库的方法，让一个容器化的应用程序能够在不同的操作系统上运行。这为开发人员提供了一种应用程序的编码方法，可以轻松地将它们从开发的笔记本电脑转移到测试环境，最后进入生产环境。

那么容器是做什么的呢？容器是一种操作系统的虚拟化技术，本质上是一个轻量级的虚拟机，通过隔离 CPU、内存、磁盘或网络等资源确保多个隔离的工作负载可以在同一主机上运行，容器比传统的虚拟化方法更加灵活，能够实现更高密度的工作负载。

1.1 了解微服务和容器

在容器出现之前，大多数企业应用程序由一个庞大的代码库组成，包含了应用程序所需的所有功能，迁移起来非常困难。随着业务需求的不断扩张，只能通过不停的扩展硬件资源来满足工作负载的硬件环境要求。

在这样的背景之下，微服务应运而生。在微服务的世界里，应用程序不再是吞噬硬件资源的怪物。相反，应用程序被分解成低耦合、独立部署的各种应用服务，而这些应用服务通常使用标准的协议（如 HTTP 或 GRPC）进行通信。企业不再需要对一个庞大的应用程序进行处理，只需要处理一堆微小的应用程序。

那么，这种微服务与容器到底有什么关系呢？容器恰好非常适合这些微服务，容器非常容易实现持续扩展和持续部署。

1.2 编排和管理工具

在一个集群中，跨多个主机进行自动操作数百个甚至数千个容器，就需要一个编排和管理工具。顾名思义，编排工具就是指通过管理和协调构成环境所需的所有服务，承担着乐团指挥的角色。这意味着管理主机、容器和服务的创建、启动、停止、升级、连接和可用的所有行为。

目前，有许多编排管理解决方案，有商业化方案，也有开源方案，可选择范围非常广泛。下文将介绍五种流行的容器编排和管理工具。

- **Kubernetes**

Kubernetes 简称 K8s 或 Kube，最初由谷歌创建，在 2015 年被捐赠给云原生计算基金会，现在作为一个开源项目，用于管理云平台中多个主机上的容器化应用。它将容器组织成 pod 组，以简化工作负载的调度难度。通过 K8S，可以实现自动部署、自动重启和自动复制，让应用程序能够自我修复，确保它们控制的服务正常运行时间。Kubernetes 能够对整个集群进行滚动升级，而不会出现应用或服务停机。

截至目前，Kubernetes 没有展示出发展放缓的迹象，凭借开源社区庞大的项目贡献者，

Kubernetes 生态系统正在快速增长。

- **OpenShift**

OpenShift 不是一个独立的编排平台，它是围绕着 Kubernetes 建立的，并扩展了该平台的功能，是 Redhat 旗下的企业容器应用平台。OpenShift 继承了 Kubernetes 的所有上游功能，但也通过增加功能来增强企业用户体验，以实现快速的应用开发、轻松部署和生命周期维护。

- **Docker Swarm**

Docker Swarm 是 Docker 本地主机集群和容器调度工具。Docker 将 Swarm 包含在 Docker Engine 中，与其他解决方案不同的是，它不需要额外的组件来操作。

鉴于其简单特点，Docker Swarm 在小型环境中的效果非常好，当然也拥有多达 30,000 个容器的大型用户。有了 Docker 企业版，可以在同一平台内同时使用 Docker Swarm 和 Kubernetes 来部署应用。

- **ECS、Fargate、EKS**

AWS 提供了几种编排容器化应用程序的服务。ECS 允许在 AWS EC2 实例上运行和扩展基于 Docker 的应用程序。ECS 还支持 Fargate，它允许运行容器而不必管理服务器。通过 ECS 内置的调度器，可以用来根据资源的可用性和需求来触发容器的部署。

亚马逊还提供了一项针对 Kubernetes 的服务，即 Kubernetes 的弹性容器服务（EKS），它可以在 AWS 上运行 Kubernetes，而不需要安装和管理集群。这种解决方案最适合那些想使用 Kubernetes 功能但又喜欢管理服务的人。

- **Apache Mesos、Mesosphere DC/OS 以及 Marathon**

Apache Mesos 既是一个集群管理工具，也是一个主机操作系统。Mesos 有自己的容器格式，也支持 Docker 容器。Mesosphere 在 Mesos 之上开发了 DC/OS（数据中心操作系统），作为一个开源项目，为需要额外功能和支持的企业提供商业服务。DC/OS 提供了一个具有管理界面、调度、网络等即用型平台。

Marathon 是一个用于容器和服务协调的框架，通常与 Mesos 和 DC/OS 一起使用。Kubernetes 也可用于在 DC/OS 之上进行协调。Mesos 和 DC/OS 通常用于节点数超过 1 万的大规模集群，集群中运行数十万个容器。对于需要大规模分布式系统规模并使用 Hadoop、Spark 或 Cassandra 等大数据应用的组织来说，这是一个不错的选择。

第 2 章：构建和部署容器

每一个采用 DevOps 的组织都希望拥有一个能够持续开发、持续打包并使应用程序能够更快地投入生产的供应链。今天，容器终于帮助用户实现了这个目标，简化服务的构建、打包、分发、部署和运维。

2.1 构建镜像

镜像是一切的核心，在镜像的创建过程中，需要注意以下四点：

- 使用可信的基础镜像。基于标准的基础镜像构建容器，确保对镜像的更新方式和时间有更好的了解。
- 限制库和依赖项。容器经常被用来解耦应用程序的组件，使得应用可以通过微服务的架构方式运行。尽可能减少镜像功能，确保安全性和操作的简易性。
- 限制访问。鉴于安全的重要性，在镜像构建过程中需要制定有效的安全方法。例如，避免以 root 权限运行进程，限制对存储器和容器中其他资源的访问。
- 镜像扫描。使用镜像扫描工具来识别镜像漏洞。

一旦确定好流程，就可以将很多工具集成到流程中去。例如 Jenkins、CircleCI、Bamboo、CodeFresh 和 GoCD，自动完成构建、打包和测试软件的任务。

2.2 分发镜像

镜像构建完成后，会推送到镜像仓库，如 DockerHub、Quay、Google、Amazon、Azure Container Registry，或私有镜像仓库中，在 Docker Registry、Portus 或 Harbor 等工具上运行。

可信的镜像仓库必须经过认证，只有通过了扫描，且已准备好被部署的镜像才能被拉取使用。这个过程中，需要为镜像添加数字签名，验证镜像的完整性和发布来源。

2.3 实施 CI/CD/CS

一个完整的容器供应链流程通常包括持续集成(Continuous Integration)、持续交付(Continuous Delivery)、持续部署(Continuous Deployment)、持续安全(Continuous Security)，这是一个持续不断的过程。

2.3.1 持续集成

持续集成是一个过程，每次代码修改提交到版本控制时，都会自动对代码进行测试。通过一系列活动来实施持续集成，包括：

- 为每个功能编写自动化测试，防止 bug 流传到后面环节中去。
- 创建一个自定义 CI 流程。CI 服务器可以监控代码库的变化，当提交新代码时会自动触发测试。
- 建立良好的测试文化。测试应该被看作是开发过程的核心部分，包括单元测试、功能测试等。

2.3.2 持续交付

持续交付是一个概念，在实施持续交付的过程中，需要注意几个重要考虑因素：

- 镜像应该由 CI 服务器自动推送到镜像仓库。
- 静态测试。对基础设施、服务配置和安全的修改与代码修改的管理过程相同。
- 运行时测试。在进入生产环境之前，在测试环境中即对运行时功能进行测试，确保代码被推送到生产环境之前没有任何问题。

2.3.3 持续部署

持续交付和持续部署存在细微的差别。持续交付有一个最终的结果，即一个特定的更新，可以用来安全部署。但是该更新并没有自动部署到生产环境中。持续部署的特点如下：

- 实施自动部署。部署可以是完全自动化的，或者至少是由人所触发的一个步骤，与编排平台集成相配合。
- 回滚机制。确保了如果在进入生产环境后后发现问题，企业可以返回到以前的软件版本。
- 集成功能标志。集成功能标志是一个可以启用或禁用某个特定功能的开关。这可以让开发人员开始执行新功能的代码，即使这些新功能还没有准备好，也可以在特殊时段使用，确保了用户不会受到不完整功能的影响。

2.3.4 持续安全

随着 DevSecOps 文化流行，安全被加入到开发运维管道中去。除了在构建和部署容器时应考虑的安全最佳实践外，还需要在容器生命周期的生产阶段实施新的安全管控。第 4 章对此有更详细的介绍。

第 3 章：容器的持续监控

虽然，容器提供了灵活性和可移植性，但它们也让监控和故障排除变得非常复杂。作为孤立的"黑盒子"，容器让传统的工具难以穿透它们的外壳，更别谈实现可视化管控。容器监控，并不是简单地获得进程的可见性，而是需要通过最小代价获得整体可见性，包括对容器、服务和基础设施等监控。

随着微服务和容器的发展，面临新的基础设施，许多传统监控系统不再有意义，企业迫切需要能够适应容器和微服务环境的安全解决方案。

在生产环境中运行容器化应用程序，必须不断监测其可用性和响应时间，这需要通过多种多样的方法收集大量的监测数据。

3.1 制定监控流程

3.1.1 确定指标

通过将收集的容器指标与基础设施的编排数据进行关联，做到实时监控容器化应用程序的性能和状态，最大限度地提高可用性。例如，青藤蜂巢 Agent 能够监测各类容器指标，包括主机和容器性能指标、自定义指标、状态指标、应用程序指标等，并用元数据和标签来标记所有指标，以支持探索、聚合、细分和深入研究。所有的系统调用活动都可以被记录下来，用于容器的故障排查。

3.1.2 追踪记录

追踪的目的是记录和追踪事件流。追踪对性能有很大的影响，所以它通常被限制在故障排除方面。通常情况下，开发人员希望得到关于他们的应用程序性能的具体数据和运行时的行为数据。自定义指标框架，如 JMX、StatsD 和 Prometheus，提供了所需的信息，却存在性能上的缺陷。常见的两个追踪工具如下：

- **应用性能监控**。应用性能管理（APM）工具通常被程序员用来在具有请求级可见性的存储环境中进行调试，对容器化应用和传统应用起着同样的作用。

- **OpenTracing**。OpenTracing 是一个新的、开放的应用程序分布式追踪标准。它允许应用程序的开发者使用代码进行事务追踪，而不需要绑定到任何特定的追踪供应商。

3.1.3 实时告警

容器环境中的告警是一项关键活动，收集并向管理员发出所发生事件信号的过程，包括镜像拉取、容器启动、杀死或内存不足的事件。这些告警信息，管理员希望在事件发生时能得到通知。

与传统环境不同，进程死亡或容器被杀死并不一定意味着有问题。编排工具即是为处理这些

情况而设计的，其通常可以自我修复，使事件恢复到工作状态，而无需管理员干预。但是，如果一个问题会影响到平台或用户，管理员可能需要得到通知。这就是为什么必须实施警报及生成警报报告，是为了来让管理员了解整个环境的运行情况。

3.1.4 故障排查

排查容器故障的一个关键挑战是，当问题发生后，容器就不再存在。这是由容器本身的原理决定的，因为容器只有在被需要时才会持续运行。当容器的单一任务完成后，就会被停止、销毁或以其他方式被处理。

编排工具会在环境发生变化时，对容器进行重新调度和安排。全方位的容器监控解决方案包含了自动记录系统事件发生的所有活动的能力。捕获信息，如命令、进程细节、网络活动和文件系统活动，并可以在容器消失和生产环境之外进行事后调查，例如，通过溯源分析来确定应用程序崩溃的原因。

3.2 收集数据方式

容器监控，需要广泛的收集指标和事件数据，以反映真实状态下的服务响应时间、资源利用率和安全状况。常见的监控和收集数据的方法包括：

- 内嵌模式。将监控软件添加到容器中，以收集和输出指标，缺点是会使容器进程变得复杂、臃肿。

- Sidecar 模式。将一个监控代理容器附加到每个部署的应用容器上。监控代理作为一个独立的进程运行会使容器数量成倍增长。

- Agent-per-pod。将一个 Agent 附加到一组容器上，如 Kubernetes pods，共享一个命名空间的容器。这种方法很容易设置，但每个 Agent 的资源消耗量大。

- Agent-per-node。利用每个主机的 Agent，通过观察操作系统内核的系统调用来收集指标。这种方法支持收集容器、进程、编排工具和底层基础设施的深入数据。

3.3 数据汇总和分类

为了解微服务和容器的实际性能，数据收集的重点在于收集分析逻辑服务而不是物理基础设施的指标。例如，在 Kubernetes 中，按照服务、部署、pods 和容器的这些层划分指标，对于逻辑故障排查是至关重要的，这样可以确定问题是来自于应用程序、微服务、pod、容器还是主机。

- **基础设施层**

基础设施层面的监控包括从主机资源到存储和网络服务，通过监控这些信息有助于确定容器

发生问题的根本原因。例如，了解哪些容器使用了最多的计算资源时，主机 CPU 指标是一个重要的因素。

- **服务**

服务指标涵盖了构成应用程序的每个服务的概况。透过这些数据可以看到每个微服务的相对行为表现，更有助于找出问题所在。

- **应用**

应用指标，如连接数、当前的响应时间和报告错误等数据。

- **编排**

编排工具包含了微服务大量的信息，主要包括：

- **常规指标**。主要指性能数据，如 CPU、内存、磁盘和响应时间。当对逻辑或应用实体（如命名空间或 Kubernetes 中的 pod）进行汇总时，这些指标有助于更好地了解应用程序行为。

- **状态指标**。例如 Kubernetes kube-state-metrics 提供的那些指标，是关于编排对象的状态或计数。状态指标可以提供关于容器数量与特定服务的信息，以及容器在循环中重启等信息，以便可以采取适当的行动。

- **内置服务**。编排工具是由多个组件和服务构建的。为确保其安全性，需要对其进行持续的监控。以 Kubernetes 为例，需要监控服务包括 etcd、API 服务器和 kubelet 等。

- **自定义指标**

自定义指标是指那些在应用程序中唯一定义的指标，或由开发人员为追踪特定信息而定义的指标。自定义指标通常具有很高的价值，用于揭示关于应用程序行为和事件的重要细节。例如，JMX 最常见的场景是监控 Java 程序的基本信息和运行情况，任何 Java 程序都可以开启 JMX，然后使用 JConsole 或 Visual VM 进行预览。

第 4 章：确保容器安全性

容器的安全运营实践是一场永无止境的循环战役。即使是实施全新的安全措施和安装补丁之后，也有可能立马遇到一个新的漏洞。下文描述了在容器环境中需要考虑的一些安全问题及安全防护措施。

- **实施容器限制**

由于容器轻量级和单进程的性质，其数量通常远远超过虚拟机，有可能会产生大量的容器。

一旦存在软件错误、设计失误或恶意软件攻击很容易造成拒绝服务。通过实施容器限制，可以防止容器消耗太多的资源。如果容器没有限制，它们可以很容易影响主机，导致无法为其它工作负载提供关键资源。因此，需要对容器 CPU 和内存使用设置限制，并且应该实施监控，将这些限制与实际消耗进行比较。

- **过期的容器镜像安全**

如果软件长时间不更新，就很容易会在生产中运行旧的、有漏洞的软件，最终被攻击者利用的可能性就越大。可以采取几个步骤来避免这种情况：首先，不断地监控容器在生产环境中运行的时间，并确保它们保持最新状态。其次，尽量避免在生产环境中运行同一镜像的不同版本，避免导致混乱和不一致的情况。最后，确保所使用的漏洞扫描器拥有最新的漏洞库，能够发现任何潜在安全问题。

- **密钥管理**

任何运行的软件可能都包含敏感信息，例如用户密码哈希值、服务器端证书、加密密钥等。这些敏感信息应该区别于应用程序代码、容器镜像和配置的管理，需要存储在一个安全的地方。

例如，Kubernetes secret 是 K8S 中的一个资源对象，主要用于保存数据库用户名和密码、令牌、认证密钥等轻量的敏感信息。Secret 可以通过 pod 配置进行使用。例如，那些有权限从 API 服务器中检索 Secret 的攻击者（例如，通过使用 pod 服务账户）就可以访问 Secret 中的敏感信息，其中可能包括各种服务的凭证。

- **容器镜像的真实性**

互联网上有大量的 Docker 镜像和资源库，如果在没有使用任何真实性机制验证的情况下提取镜像，并在自己的系统上运行软件，风险会很大。在使用镜像之前可以思考一下下述的几个问题：

- 镜像是从何而来
- 镜像的创建者是否可信
- 该镜像最后一次更新是什么时候
- 对镜像采取了哪些安全策略
- 如何确认镜像是否有被人篡改过

青藤蜂巢的镜像检查能力已经覆盖到开发、测试等多个环节中，可快速发现镜像中存在的漏洞、病毒木马、webshell 等镜像风险。

在开发环节，确保构建的镜像是符合安全规定的。在这个阶段，青藤蜂巢可以对镜像的相关

构建文件进行深度检查，包括 Dockerfile 文件、Yaml 文件等。

在测试环节，确保镜像运行起来后是安全的，发现那些在静态测试环节无法发现的安全问题。青藤蜂巢的动态检测会对运行时最底层的组件风险、应用风险、微服务风险进行全面的检查，且整个动态检测过程完全自动化，无需手动操作。这也是目前国内首个针对运行态的容器进行风险检查的安全方案。

针对镜像扫描的检测，包括了静态检测和动态检测，在静态检测过程中，凡是不合格镜像都“不准入”测试仓库，在动态检测过程中，需对生产仓库和节点的镜像进行持续的安全检查，凡是不合格镜像都“不准出”生产仓库进行部署运行。

青藤蜂巢的镜像扫描又快又准，每千个镜像扫描时间少于 2 分钟，CPU 消耗少于 10%，内存小于 100M。截至目前，青藤蜂巢拥有 5 万+的安全漏洞补丁库，覆盖了 python、nodejs、ruby、php 等组件，通过多引擎发现镜像中的病毒、挖矿、web 后门，以及深入发现镜像中 ssh-key、用户密码等敏感信息。

- **共享内核架构的安全性**

主机系统是一个共享组件，一旦被攻击，就会导致容器底层被攻击。所以，采取适当的措施，限制对容器和主机操作系统服务进行不必要的内核功能访问，降低默认的容器权限是很重要的。用户应该只访问他们需要的服务，去除非必要的功能；当添加服务时，避免以特权（root）用户、特权容器和敏感挂载点的方式运行容器；确保强制执行访问控制，以防止不必要的操作，尤其是在主机内核级别。

- **运行时监控**

即便采取了所有预防措施，镜像仍然可能在运行期间存在入侵的风险。比如，内部应用程序有一个未知的漏洞，或者发现攻击者正在使用一个包含的 0day 漏洞的镜像时该怎么办？，这就需要采取安全措施来检测各种攻击活动。

面对众多的容器运行时入侵行为，青藤蜂巢采用多锚点的分析方法，实时检测容器中的已知威胁、恶意行为、异常事件。

首先，青藤蜂巢能对容器内的文件、代码、脚本等进行已知特征的检测。以 Webshell 检测为例，青藤雷火引擎不依赖正则匹配，根据 AI 推理发现 Webshell 中存在的可疑内容。即便是在实战化对抗环境下，Webshell 检测率高达 99.54%。

其次，青藤蜂巢基于对恶意行为模式的定义，可对容器及编排工具内的黑客攻击行为进行实时检测。例如，通过比对攻击链路上的关键攻击路径，针对黑客的每一步探测，系统均会进行持续性的检测。

最后，由于容器不可变基础设施的属性，其运行时行为模式相对固定。青藤蜂巢，通过对其进程行为、网络行为、文件行为进行监控和学习，建立稳定的容器模型。只要对异常偏离的行为进行分析，就能发现未知的入侵威胁，包括 0day 等高级攻击。

- **安全合规**

安全问题大部分都是由于未遵循安全配置和合规性实践导致的。幸运的是，容器有许多默认的运行时安全功能，例如，CIS 已经发布了针对 Docker 和 Kubernetes 的一般安全建议，可以作为安全基线重要组成部分。

- **K8S 安全**

Kubernetes 的安全功能主要包括：

- 基于角色的访问控制（RBAC），规定了授权和访问控制规范，定义了 Kubernetes 实体上允许的行动。
- Pod 安全策略，例如，配置资源、权限和敏感配置项。
- 网络策略，pod 群组之间以及与其他网络端间的通信规范。

总结

在过去的几年中，容器技术得到快速发展，企业机构尝试运用容器技术提高软件开发效率，但是容器存在着诸多挑战。要充分实现容器化应用程序的安全性、高可用性和数据持久性，需要确保生产环境中容器的安全运行，这是保护企业容器安全的“必修课”。