第 42 卷第 3 期 遥 测 遥 控 Vol. 42, No. 3
2021 年 5 月 Journal of Telemetry, Tracking and Command May 2021

Email:ycyk704@163.com TEL:010-68382327 010-68382557

基于 Kubernetes 的航天地面应用软件架构设计与实现

钟伟宏[1]，孙甲琦[2]，朱宏涛[1]，冯 炜[1]
（1 北京遥测技术研究所 北京 100094
2 中国航天电子技术研究院 北京 100094）

摘要：为了满足航天地面应用软件功能组成日益复杂、资源动态重组、系统可用性要求越来越高等新需求，采用微服务架构思想并结合容器轻量级的特点，设计并实现了一套基于 Kubernetes 微服务管理的航天地面应用软件架构，详细介绍了微服务划分和部署、服务发现和负载均衡、服务动态更新以及服务高可用等关键功能并进行了关键性能测试。测试结果表明，架构有效提高了复杂化的航天地面应用软件的可用性、可复用性和资源利用率，极大地降低了软件开发和运维成本。

关键词：航天地面应用软件；微服务；容器；Kubernetes
中图分类号：TP311.5 **文献标识码：**A **文章编号：**CN11-1780(2021)03-0039-09

Design and implementation of aerospace ground application software architecture based on Kubernetes

ZHONG Weihong[1], SUN Jiaqi[2], ZHU Hongtao[1], FENG Wei[1]
（1 Beijing Research Institute of Telemetry, Beijing 100094, China;
2 China Academy of Aerospace Electronics Technology, Beijing 100094, China）

Abstract: In order to meet the new requirements of increasingly complex aerospace ground application software structure, dynamic reorganization of resources and high system availability, this paper adopts the idea of microservice architecture and combines the lightweight characteristics of containers to design and implement a set of aerospace ground application software architecture based on Kubernetes. It introduces the division and deployment of services, service discovery and load balancing, service dynamic update and service high availability in the software in detail and the above key performance has been tested. The test results show that the software design and implementation plan effectively improves the availability, reusability and resource utilization of the complicated aerospace ground application software, and greatly reduces the cost of enterprise software development and maintenance.

Key words: Aerospace ground application software; Microservice; Container; Kubernetes

引 言

随着航天技术的快速发展，我国载人航天、深空探测等任务深入推进，航天发射任务日益密集，多频段、新体制、新型号任务不断涌现，在轨航天器数量大幅增加[1]。航天地面应用软件需要快速适应任务资源重组要求，软件需求变化快、研制周期短、状态固化晚、质量要求高、结构也变得越来越复杂。在传统模块化的航天地面应用软件架构中，模块之间耦合度较高，一个小模块的异常就有可能导致整个系统软件崩溃，极大地降低了系统的稳定性和容错性。同时，由于系统的功能之间关联度高，一个功能模块的开发还依赖于另一个功能模块，降低了软件开发效率，延长了软件开发周期。因此，航天地面应用软件需要一种新架构来降低软件的复杂性并将系统解耦合，提高系统的容错性，而微服务架构恰恰满足这种新需求[2]。微服务架构的理念是将一个庞大的应用分成一组小而专的服务[3]，每个服务可以独立运行在不同的运行环境中，服务之间通过轻量级的通信机制进行交互，并且服务间可以独立部署，互不影响。在航天地面应用软件微服务化的过程中，划分出来的微服务变得越来越多，如何对应用软件中的

收稿日期：2021-01-15 收修改稿日期：2021-03-22

微服务进行管理也变得越来越重要。

本文针对当前航天地面应用软件结构复杂化、可用性要求高和需求变化快等特点，提出一种基于 Kubernetes 微服务管理的航天地面应用软件架构，详细介绍了基于 Kubernetes 来实现航天地面应用软件中微服务的划分和部署、服务发现和负载均衡、服务动态更新与服务高可用等关键功能，完成对服务的管理。通过该方案可以实现航天地面应用软件的快速构建和分布式部署，极大地提高了航天地面应用软件的可用性、可重用性和开发效率，降低了开发和维护成本。

1　航天地面应用软件架构设计

1.1　Kubernetes 介绍

Kubernetes 是目前最流行的容器管理平台[4]，采用 Kubernetes 可实现应用的部署、调度以及监控等一系列的基本功能，并且还支持服务的发现、自动重启、自动伸缩等诸多功能。

Kubernetes 是 Google 开源的容器集群管理平台，该平台提供了一整套完整易用的 RESTful API（Representational State Transfer Application Programing Interface）对外服务接口。Kubernetes 管理平台的核心理念是为运行在其上的容器应用提供一套高可用的集群自我愈合机制，从而使运行在其上的应用一直处于用户所期待的状态中[5,6]。

Kubernetes 的架构如图 1 所示，主要可以分为控制层和工作节点两个部分[7]。控制层主要包括 API 服务器、控制管理器、调度器和 etcd 键值数据库等组件，实现了对集群做出全局决策（比如调度）以及检测和响应集群事件等控制功能；而工作节点则是微服务实际运行的物理服务器，每个节点运行着多个 Pod，每个 Pod 可运行多个容器，多个同种类型的 Pod 共同组成一个微服务。工作节点主要包括 kubelet 组件和 kube-proxy 组件，这两个组件可以维护节点上运行的 Pod 并为每个节点提供运行环境。

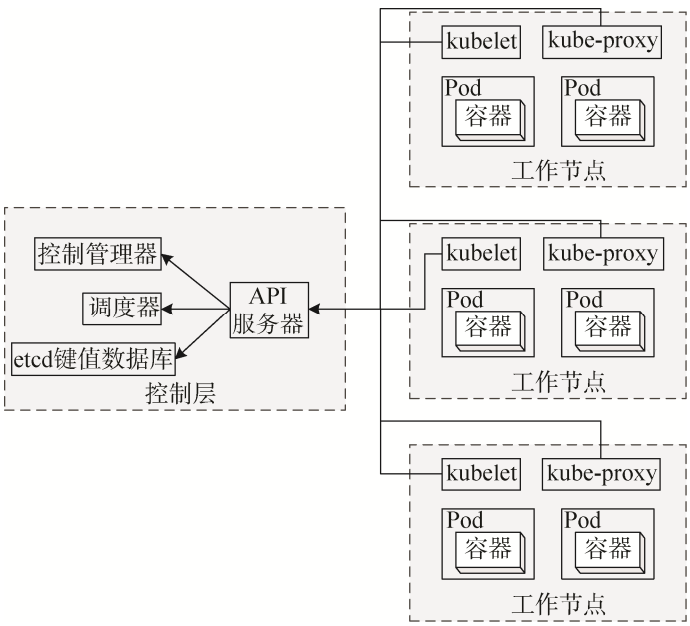

图 1　Kubernetes 架构图

Fig. 1　Kubernetes architecture diagram

1.2　基于 Kubernetes 的航天地面应用软件架构设计

常规应用软件的架构一般分为两种类型：传统物理机部署架构和虚拟机部署架构。传统物理机部署架构一般把应用部署在物理服务器上，但由于无法为一台物理服务器中的应用程序定义资源边界，因此，可能会导致应用间资源抢占的问题。一种解决方案是在不同的物理机上部署不同的应用程序，每台物理机只运行一个应用程序，但这会导致每台计算机的资源利用率太低。虚拟机部署架构允许在一台物理服务器上运行多个虚拟机（Virtual Machine），允许应用程序运行在虚拟机之间并进行资源隔离和进程隔离，能够更好地利用物理服务器上的资源。容器技术类似于虚拟机，也提供一定程度的隔离性，具备自己的文件系统、CPU、内存、进程空间等。但虚拟机相当于虚拟化出来一台完整的计算机，在虚拟化硬件之上运行所有组件（包括其自己的操作系统），而容器则更加轻量化，只在应用程序之间共享操作系统[8]。

相比前两种架构部署方式，由于容器轻量级的特性，它创建起来比较简便和快捷，启动快速，构建简单，能支持应用的快速构建、部署和更新，因此，在将航天地面应用软件划分成微服务后，本文选择微服务容器化这种架构来进行应用的部署[9]。基于 Kubernetes 微服务管理的航天地面应用软件总体架构如图 2 所示，包括物理层、Kubernetes 平台层和应用层三层，其中，应用层根据当前航天地面应用软件

特点采用 C/S 架构设计。三个基础管理功能包括分布式存储管理、镜像仓库管理和服务间通信组件管理支撑本架构正常运行。

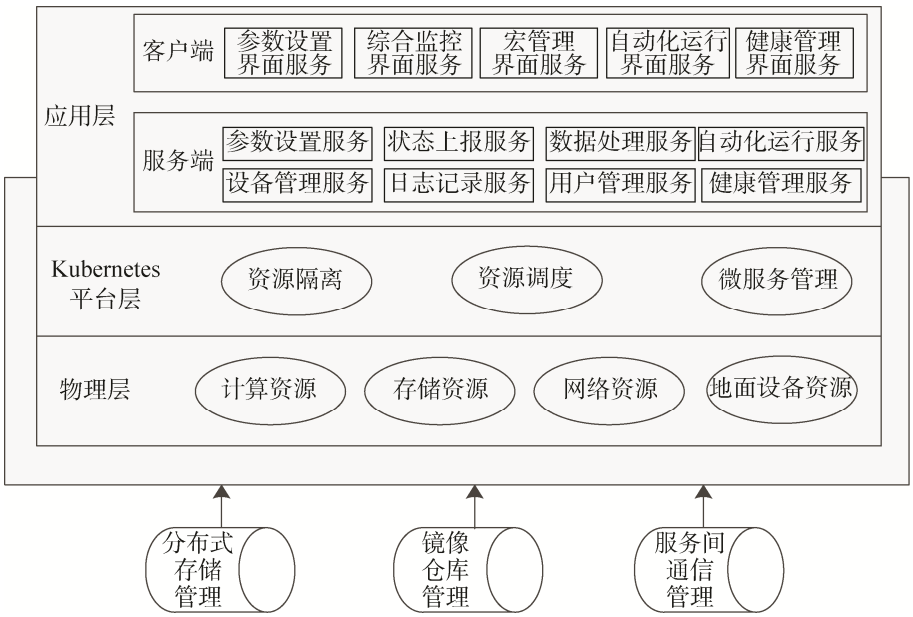

图 2　航天地面应用软件架构图

Fig. 2　Aerospace ground application software architecture diagram

物理层主要是物理机器、虚拟机、云平台、地面设备等物理设备抽象出来的一些资源，包括 CPU、内存、磁盘、网卡等资源在内的计算资源、存储资源和网络资源，这些资源可被上一层的 Kubernetes 调度并分配给不同微服务。

Kubernetes 平台层主要是由 Kubernetes 构建的微服务管理平台，它能够对应用层所有微服务进行监控和管理，应用的生命周期、资源分配等都由这一层统一管理控制。

应用层主要是采用微服务架构的思想将航天地面应用软件抽象成一个个微服务并容器化地部署到 Kubernetes 平台中[10]。微服务架构在实现系统的低耦合和高内聚的过程中起到了极其重要的作用，使得软件的开发更加适合小而自治的团队开发，提高了多部门协作的效率。同时，结合容器技术，能够实现应用的自动化部署、扩展和监控，大大地提高了开发效率。在应用层，将航天地面应用软件中具有代表性的功能划分成客户端服务和服务端服务，每个服务功能划分单一、服务之间接口明确，通过各个服务之间的组合来完成航天地面应用软件对设备（包括天线、基带以及各种硬件板卡等）和数据（包括常见的遥测接收数据和遥控指令等）等资源进行管理并完成相应的功能。各个微服务分别以容器的方式运行在各自的运行环境中，微服务间独立部署、互不干扰，相互之间只通过特定的接口来进行交互，降低了软件系统的耦合度。

分布式存储管理的主要作用是将物理集群中的存储资源都集中起来统一进行管理，为系统提供统一的存储服务，提高了存储资源的利用率，简化了分布式环境中微服务的数据存储复杂度，保证了数据的一致性。镜像仓库管理主要是用来存储航天地面应用软件微服务的镜像文件，对微服务功能代码的更新首先需要先将镜像文件更新，再通过 Kubernetes 从镜像仓库中拉取相应的镜像并创建新的容器。服务间通信组件管理主要是通过消息机制来传递微服务间的数据，并降低微服务间通信的耦合性。当系统中某一个微服务需要跟其它微服务通信时，只需要发送相应的消息即可，另一个微服务只需要订阅与它相关的消息就能收到其它微服务发来的数据。

2　关键功能设计与实现

本文针对航天地面应用软件架构中微服务划分与部署、服务发现与负载均衡、服务动态更新和服务

高可用等关键功能进行详细设计与实现。与常规软件相比，航天地面应用软件功能繁多、复杂程度差别大，如任务参数控制软件功能简单，自动化运行软件功能复杂、消耗资源多，通过负载均衡可以把复杂软件均衡分配到物理机上。航天地面应用中用户的应用方式与细节跟任务紧密相关，有些具体需求在任务联试过程中才逐步明确，人机交互功能变动大、需要频繁升级，服务动态更新能解决服务不停机升级问题而不影响其它服务。另外，由于航天系统对软件的可靠性要求高，原来单机部署时需要使用双机备份机制提高可靠性，采用本文中的微服务架构实现服务高可用不仅能保证系统的高可靠性，同时还极大地提高了软件的资源利用率。

2.1　微服务划分与部署

航天地面应用软件覆盖面特别广泛，本文针对具有代表性的航天地面测控系统软件来描述微服务划分与部署。依据服务职责单一、高内聚低耦合、服务粒度适中等划分原则，可以将航天地面测控系统应用软件进行抽象并划分为参数设置界面服务、综合监控界面服务、宏管理界面服务、自动化运行界面服务、健康管理界面服务等客户端微服务部分和参数设置服务、状态上报服务、数据处理服务、自动化运行服务、设备管理服务、日志记录服务、用户管理服务、健康管理服务等服务端微服务部分，服务之间通过约定好的接口进行交互。通过这种细粒度的服务划分，不同项目软件服务可重复使用，提高了软件开发效率。

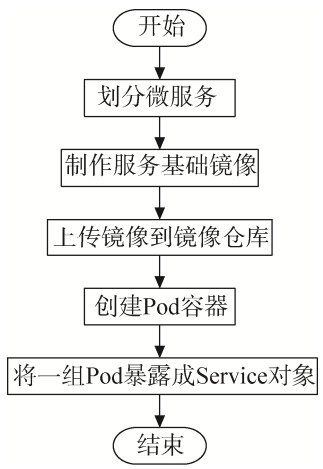

图 3　微服务部署流程图

Fig. 3　Microservice deployment flowchart

微服务部署的流程图如图 3 所示，首先，将其制作成容器运行所需要的基础镜像，再将基础镜像上传到集群镜像仓库中；然后，通过编写 YAML（YAML Ain't a Markup Language）文件由 Kubernetes 从镜像仓库中调度相应微服务镜像，在合适的工作节点创建相应的 Pod；最后，通过创建 Service 对象将一组微服务 Pod 的 IP 地址和端口暴露出来，并提供负载均衡功能。

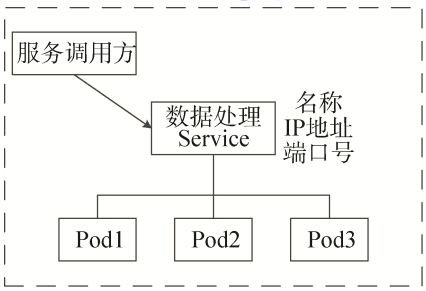

图 4　数据处理 Service 示意图

Fig. 4　Data process Service schematic diagram

如图 4 所示，Service 可以看作是一组提供相同服务的 Pods 对外提供的接口，对外提供了固定和统一的访问地址，服务调用方只需要向该 Service 对象发起请求即可调用服务，而不需要关心服务实例具体运行在何处。每个 Service 对象将不可变的名称、IP、端口抽象成一个前端，然后通过标签选择器选择一组符合条件的服务 Pods 作为后端。后端是高度动态的，这些 Pods 随时都可能因为动态伸缩或者自动更新而增加新的 Pod 或者删除原有的 Pod，若没有一个统一的、不变的前端 Service，每个应用服务都需要对每个 Pod 进行监控并维护一个当前健康、可用的 Pod 列表，这将大大增加应用的工作量和降低程序的运行效率。Service 对象将应用服务抽象出来，极大地提高了应用的可扩展性和伸缩性。

2.2　服务发现与负载均衡

航天地面应用软件微服务之间需要能互相感知并且能相互调用，才能构成一个功能完整的软件系统。以一个用户界面服务和一个数据处理服务为例，我们将服务之间的注册、发现和调用设计为如图 5 所示。在数据处理服务创建的时候，首先要进行服务的注册，使其能被其他服务所感知，本文使用 Kubernetes 中支持集群的 DNS（Domain Name System）服务器来作为服务的注册中心。服务创建时向 API 服务器以 dataProcessService 为名称提交一个新的数据处理服务定义，API 服务器为该服务分配一个虚拟 IP 地址 Cluster IP，而 DNS 服务器一直在监听 API 服务器中是否有新服务的提交。当新服务创建的

时候，它会为每个服务都创建一组 DNS 记录，记录服务名称和相应的虚拟 IP 地址。当其它微服务需要调用数据处理服务时，只需要向 DNS 服务器发起对域名（服务名）为 dataProcessService 的查询即可，然后就能在 DNS 记录里找到域名对应的服务的访问地址 Cluster IP，至此就完成了对数据处理服务的发现。最后，只需对该 Cluster IP 发起请求即可对数据处理服务进行调用。

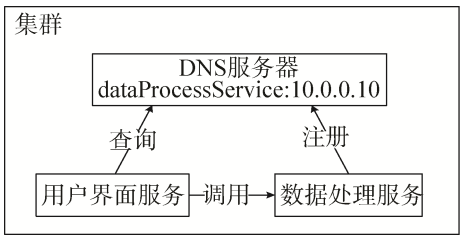

图 5　服务发现示意图

Fig. 5　Service discovery schematic diagram

数据处理服务的发现只是获取到服务的 Cluster IP，但该地址只是虚拟 IP 地址，如图 4 所示。服务后端还可能对应着多个具体的 Pod 实例，还需要将指向 Service 对象的流量转发到具体的 Pod 实例中去。本文主要通过 Kubernetes 集群中每个工作节点上运行着的 kube-proxy 进程来实现将服务调用方的流量均衡到后端的 Pods 上。kube-proxy 组件主要通过设置 IPVS（Internet Protocol Virtual Server）规则，按一定的策略将发给 Cluster IP 的请求转发到相应的 Pod，其默认负载均衡策略为轮询调度策略。轮询调度策略是最简单的负载均衡算法，它假设所有后端 Pod 的处理能力是一样的，调度器会将所有请求按依次循环的方式平均地分配到每个不同的 Pod 上，该算法一般足以满足日常的负载均衡需求。除此之外，IPVS 还支持加权轮询、最小连接数、加权最小连接数、目标地址哈希等负载均衡算法，可以根据航天地面应用业务实际需求选择最合适的算法。

2.3　服务动态更新

航天地面应用软件具有长期管理需求，需要 7×24 小时不间断运行。如果某些服务功能在正常运行过程中进行更新和扩展，就需要考虑对服务进行动态更新，实现应用的不停机升级、回滚等操作。

本文采用 Kubernetes 中的 Deployment 控制器来维护服务的后端副本集 ReplicaSets 和 Pods 状态。在创建服务的时候，首先指定 Pods 的期望副本个数以及用来创建容器的镜像等模板来创建 ReplicaSet 对象，然后使用 ReplicaSet 对象实时监控当前集群中 Pod 的状态和数量，通过增加或者删除 Pod 来使得副本数达到定义时的期望值，确保任何时间都有指定数量的 Pod 副本在运行。

服务动态更新过程如图 6 所示，在对 Pod 副本数量为 3 的服务进行更新的时候，首先将其模板进行更新，然后根据新模板会创建一个副本数量为 0 的新 ReplicaSet，并保持原有的旧 ReplicaSet 不变以不间歇提供服务。在更新过程中逐渐将新的 ReplicaSet 副本数扩展到 3，把旧的 ReplicaSet 副本数缩减到 0，并维持服务所有 ReplicaSet 的副本数保持在期望的 3 个左右。因此，在服务的更新过程中能保持有一定数量的 Pod 来提供服务，实现了应用的不停机动态更新功能。

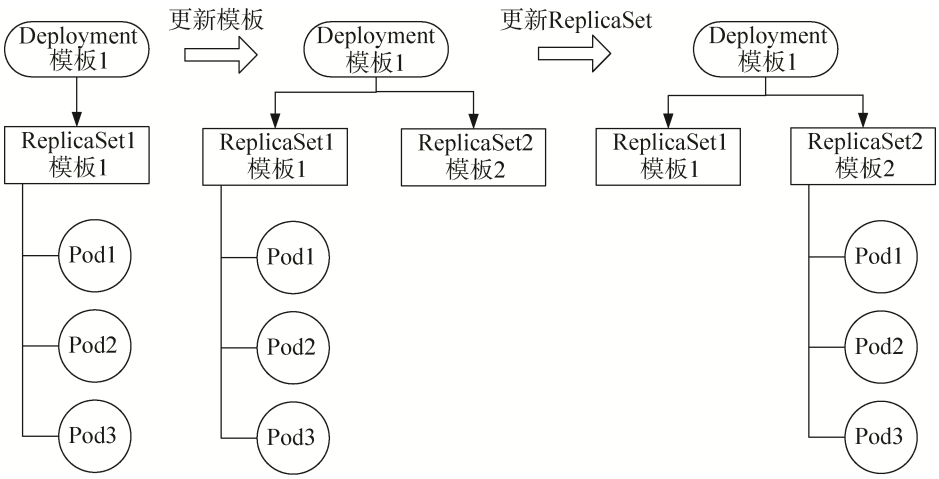

图 6　服务更新过程

Fig. 6　Service update process

在对服务进行升级后，会将每次的修订记录保存在系统中，若发现当前版本功能不够完善时，可以

查看历史的修订记录，回退到以前的稳定版。由于之前升级时保留了旧版本的 ReplicaSet 没有删除，因此，只需将现有的 ReplicaSet 副本数缩减到 0，再将之前稳定版本的 ReplicaSet 副本数扩展到期望的数量，即可十分方便地对服务进行回滚。

2.4　服务高可用

高可用设计是分布式航天地面应用软件架构设计中必须考虑的因素之一，它通常是指通过设计减少系统不能提供服务的时间。

在微服务架构中，航天地面应用软件以服务的方式运行在集群中，每个服务后端对应着多个 Pod，这些 Pod 互为主备，相互冗余，即使其中一个 Pod 出现问题，其它 Pod 也能继续提供服务。仅仅设计 Pod 的冗余策略还不能完全保证服务的高可用，当服务后端某个 Pod 宕机后，只通过人为的方式进行异常恢复明显是不可行的。本文采用 Kubernetes 的容器探针来周期性检查容器中的进程和服务状态，从而根据预设的重启策略来决定是否要自动重启容器。

容器探针分为三种类型：启动探针、就绪探针和存活探针。这三种类型分别能检查不同类型的服务异常状况。启动探针主要用来探测服务是否启动完成；就绪探针主要用来探测容器启动完成后是否准备好能提供服务，若未就绪，则不将该容器的 IP 和端口对外暴露以提供服务；而存活探针则主要探测服务在运行过程中是否能正常对外提供服务，感知服务是否存活，若探测失败，则重启服务的容器。

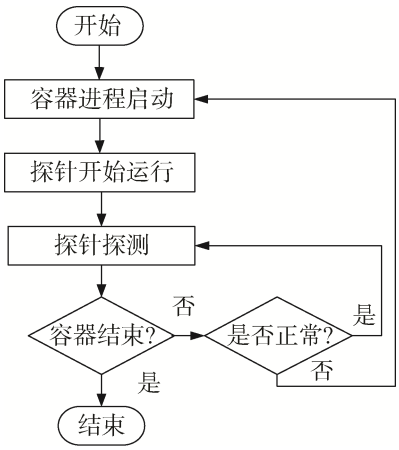

图 7　容器探针探测流程图

Fig. 7　Container probe detection flowchart

容器探针探测的流程图如图 7 所示，在服务后端相应的容器进程启动时，探针开始运行，每隔一定时间都会检查一遍容器是否就绪、能否正常提供服务，若探测出现异常，则根据重启策略来重启容器，若探测正常，则再隔一段时间继续进行探测，直至容器被销毁。探针探测贯穿容器的整个生命周期，为服务的正常运行提供健康检查，并自动重启异常服务，极大地提高了服务的可用性。

3　关键性能测试

本章主要对微服务的负载均衡、微服务动态更新和服务高可用等关键性能进行测试。系统的运行环境是包含 3 个节点的服务器集群，每个节点的硬件配置为 24 核 CPU、32G 内存，操作系统采用 Centos7，集群中部署了 Kubernetes 管理平台来管理微服务。

3.1　服务负载均衡测试

服务负载均衡测试主要是测试服务调用方向服务发送请求时，服务是否将请求均匀地分发到后端的每个 Pod 中去。负载均衡测试用例如表 1 所示。

表 1　服务负载均衡测试用例

Table 1　Serveice load balance test case

测试编号	01
测试目的	测试服务的负载均衡功能是否正常
测试步骤	① 创建后端 Pod 数量为 3 的数据处理服务，该服务会返回当前 Pod 的名称和数据处理结果； ② 同时模拟向服务发起 100 个数据处理请求； ③ 统计返回的 Pod 名称的数量并以百分比表示，同时观察每个 Pod 的 CPU、内存等资源占用情况。
预期结果	请求均匀地分发到服务的不同 Pod 上，返回的每个 Pod 名称的数量大概在 33%左右，每个 Pod 的 CPU 和内存等资源占用大体相当。
测试结果	测试结果与预期相符

图 8 所示为返回的 Pod 名称百分比，可以看到发送到每个 Pod 的请求数量大致是一样的，表明服务的轮询负载均衡策略成功生效，并把发给前端服务的流量均匀地转发到了后端 Pod。

3.2　服务动态更新测试

服务动态更新测试主要是对服务进行滚动升级测试。对服务的模板进行更新，把原来数据处理服务镜像版本从 1.00 版升级到 1.01 版，测试是否会始终保持有可用的 Pod 来提供服务，测试用例如表 2 所示。

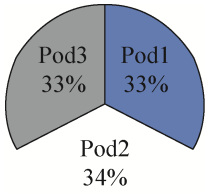

图 8　Pod 数量占比

Fig. 8　The proportion of Pods

如图 9 所示，查看服务 Deployment 控制器的事件可以看到，首先新建了一个 ReplicaSet 并增加其控制的 Pod 副本数，然后慢慢减少旧 ReplicaSet 的 Pod 副本数，最终完成服务的升级。相比传统的软件升级必须重启才能生效的做法，服务滚动升级始终保持可用的 Pod 副本数在 3 个左右，保证了服务的不停机更新，提高了应用的可用性。

表 2　服务滚动升级测试用例

Table 2　Serveice rolling update test case

测试编号	02
测试目的	测试服务的滚动升级功能是否正常
测试步骤	① 选择镜像仓库中版本为 1.00 的镜像创建数据处理服务，并设置 Pod 副本数为 3； ② 选择要升级的服务，将服务的镜像版本从 1.00 改为 1.01 进行升级； ③ 观察服务后端的 ReplicaSet 和相应的 Pod 数量。
预期结果	在对服务的镜像版本进行升级后，服务后端会新增一个 ReplicaSet，它控制的 Pod 数量在慢慢增加，原来旧的 ReplicaSet 控制的 Pod 数量在慢慢减少，最终服务升级完成后新 ReplicaSet 的 Pod 副本数维持在 3 个，旧 ReplicaSet 的 Pod 副本数减少到 0。
测试结果	测试结果与预期相符

```
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  24m    deployment-controller  Scaled up replica set data-process-5dd6877b95 to 3
  Normal  ScalingReplicaSet  5m38s  deployment-controller  Scaled up replica set data-process-7c74bcf768 to 1
  Normal  ScalingReplicaSet  5m36s  deployment-controller  Scaled down replica set data-process-5dd6877b95 to 2
  Normal  ScalingReplicaSet  5m36s  deployment-controller  Scaled up replica set data-process-7c74bcf768 to 2
  Normal  ScalingReplicaSet  5m34s  deployment-controller  Scaled down replica set data-process-5dd6877b95 to 1
  Normal  ScalingReplicaSet  5m34s  deployment-controller  Scaled up replica set data-process-7c74bcf768 to 3
  Normal  ScalingReplicaSet  5m31s  deployment-controller  Scaled down replica set data-process-5dd6877b95 to 0
```

图 9　服务滚动升级测试结果

Fig. 9　Service rolling update test result

3.3　服务高可用测试

服务高可用测试主要有两个测试项目，一个是测试服务中 Pod 异常退出后是否会重新创建新的 Pod 来提供服务，一个是使用存活探针探测 Pod 是否存活，并验证探测失败后是否会自动重启容器，这两个测试的测试用例如表 3 和表 4 所示。

如图 10 所示，通过查看服务中 ReplicaSet 控制器的事件可以看到，在服务的其中一个 Pod 宕机后，ReplicaSet 又重新创建了一个新的 Pod 来提供服务。不管是由于断电等物理原因还是程序异常等软件原因，导致 Pod 进程异常退出后 Pod 都会自动进行重启，这在一定程度保证了服务的可用性。

表 3　Pod 自动重启测试用例

Table 3　Pod automatic restart test case

测试编号	03
测试目的	测试服务中的 Pod 自动重启功能是否正常
测试步骤	① 创建副本数量为 3 的数据处理服务，使其均匀地分布在集群中 3 台工作节点上； ② 手动关闭其中一台服务器，模拟该服务器由于异常而导致宕机； ③ 观察该服务器上的 Pod 是否在其它服务器上重启。
预期结果	在手动关闭其中一台服务器后，Kubernetes 检测到服务的某个 Pod 异常退出且服务器节点少了一个，则会自动在存活的另两台服务器上选择一台，将退出的 Pod 在合适的服务器重新创建，使服务的后端 Pod 副本数维持在 3 个。
测试结果	测试结果与预期相符

表 4　存活探针功能测试用例

Table 4　Liveness probe function test case

测试编号	04
测试目的	测试存活探针对容器的诊断功能是否正常
测试步骤	① 在创建 Pod 时定义存活探针的命令，通过 Exec 的方式调用命令 cat /tmp/liveness，探测 liveness 文件是否存在判断容器是否存活； ② 在 Pod 运行一段时间后，手动将容器中/tmp 目录下的 liveness 文件删除； ③ 观察容器是否自动重启。
预期结果	Pod 创建后一直保持运行状态，在将 liveness 文件删除后，存活探针执行 cat /tmp/liveness 命令后返回码不为 0，因此探针探测存活失败，然后会自动将该容器重启。
测试结果	测试结果与预期相符

```
Events:
  Type    Reason            Age    From                   Message
  ----    ------            ----   ----                   -------
  Normal  SuccessfulCreate  6m31s  replicaset-controller  Created pod: data-process-5dd6877b95-8rkrw
  Normal  SuccessfulCreate  6m30s  replicaset-controller  Created pod: data-process-5dd6877b95-khp6n
  Normal  SuccessfulCreate  6m30s  replicaset-controller  Created pod: data-process-5dd6877b95-svzr4
  Normal  SuccessfulCreate  61s    replicaset-controller  Created pod: data-process-5dd6877b95-vg944
```

图 10　Pod 自动重启测试结果

Fig. 10　Pod automatic restart test result

如图 11 所示，在删除容器中的 liveness 文件后，通过查看 Pod 的事件可以看到，在存活探针运行命令 cat /tmp/liveness 失败后，系统检测到该容器处于不健康的状态，因此，会将该容器杀死并重启该容器，以便该容器能正常提供服务。这种探针探测方法相比于传统的进程退出即重启的方法，为我们提供了另外的健康检查机制检测应用正常与否，避免了由于进程阻塞等其它异常问题而导致的服务不可用的情况，极大地提高了服务的可用性。

```
  Warning  Unhealthy  3m52s (x3 over 4m2s)  kubelet, node01    Liveness probe failed: cat: can't open '/tmp/liveness': No su
ch file or directory
  Normal   Killing    3m52s                kubelet, node01    Container data-process failed liveness probe, will be restart
ed
```

图 11　存活探针测试结果

Fig. 11　Liveness probe test result

3.4　测试结果分析

经过对系统进行关键性能测试，有效验证了微服务负载均衡、动态更新和服务高可用功能。表 5 所示为传统架构和本文架构的对比。本文中微服务架构相比传统物理机部署架构，从一个功能只由一个软件实现的单实例模式升级到能将请求均衡地发送到服务的多个实例中，极大地提高了软件的处理能力；在软件更新时，传统架构需要重启软件才能使更新生效，本文架构能支持服务的动态更新，使软件能 7×24 小时不间断运行；在高可用方面，传统架构采用双机备份方式提高系统的可靠性，这样不仅没有充分利用主机资源，且如果运行软件的两个物理机都出现问题时软件也无法正常运行，本文架构通过服务自动重启和容器探针来支持多个服务的高可用，即使集群中多个主机宕机，服务也能在其它可用节点重启，能更加有效地提高航天地面应用软件的可靠性和资源利用率。

表 5　传统架构和本文架构的对比

Table 5　Comparison of traditional architecture and this paper's architecture

功能/架构	传统物理机部署架构	本文架构
负载均衡	一个功能只由一个软件实现	可以将请求均衡发送到多个实例
软件更新	需重启软件才能生效	支持不停机更新
高可用	采用双机备份提升可靠性	支持多个服务的高可用

4　总　结

本文针对目前航天地面应用软件越来越复杂、无法有效利用物理资源、系统容错性较低等问题，提

出了一种基于 Kubernetes 微服务管理的航天地面应用软件架构设计与实现思路，将航天地面应用软件划分成若干个微服务运行在服务器集群上，并基于 Kubernetes 平台对微服务进行管理，实现了应用中服务的负载均衡、动态更新和高可用，通过关键性能测试表明这种架构能有效提高航天地面应用软件的资源利用率和容错性，降低开发和运维成本，为航天地面应用软件架构创新设计提供了一种有益的尝试。

参考文献

[1] 雷厉, 朱勤专. 飞行器测控通信技术发展趋势与建议[J]. 飞行器测控学报, 2014, 33(6): 463–468.
LEI Li, ZHU Qinzhuan. Analysis of the trend of development of spacecraft TT&C and communication technologies and suggestions[J]. Journal of Spacecraft TT&C Technology, 2014, 33(6): 463–468.

[2] 辛园园, 钮俊, 谢志军, 等. 微服务体系结构实现框架综述[J]. 计算机工程与应用, 2018, 54(19): 10–17.
XIN Yuanyuan, NIU Jun, XIE Zhijun, et al. Survey of implementation framework for microservices architecture[J]. Computer Engineering and Applications, 2018, 54(19): 10–17.

[3] FOWLER M, JAMES L. Microservices a definition of this new architectural term[EB/OL]. https://martinfowler.com/articles/microservices.html.

[4] BURNS B, GRANT B, OPPENHEIMER D, et al. Borg, Omega, and Kubernertes[J]. Communications of the ACM, 2016, 59(5): 50–57.

[5] 陈金光. 基于阿里云的 Kubernetes 容器云平台设计与实现[D]. 杭州: 浙江大学, 2018.
CHEN Jinguang. Design and implementation of Kubernetes container cloud platform based on Alicloud[D]. Hangzhou: Zhejiang University, 2018.

[6] 翁湜元, 单杏花, 阎志远, 等. 基于 Kubernetes 的容器云平台设计与实践[J]. 铁路计算机应用, 2019, 28(12): 48–53.
WENG Shengyuan, Shan Xinghua, Yan Zhiyuan, et al. Design and practice of container cloud platform based on Kubernetes[J]. Railway Computer Application, 2019, 28(12): 48–53.

[7] KUBERNETES COMPONENTS[EB/OL]. https://kubernetes.io/docs/concepts/overview/components.

[8] JOY A M. Performance comparison between Linux containers and virtual machines[C]//2015 IEEE International Conference on Advances in Computer Engineering and Applications, 2015: 342–346.

[9] 于泽萍. 面向微服务架构的容器云平台设计与实现[D]. 哈尔滨: 哈尔滨工业大学, 2018.
YU Zeping. Design and implementation of container cloud platform based on microservice architecture[D]. Harbin : Harbin Institute of Technology, 2018.

[10] BERNSTEIN D. Containers and Cloud: From LXC to Docker to Kubernetes[J]. IEEE Cloud Computing, 2014, 1(3): 81–84.

[作者简介]

钟伟宏　1997 年生，在读硕士研究生，研究方向为航天测控系统软件设计。

孙甲琦　1976 年生，博士，研究员，主要从事航天测控系统总体设计工作。

朱宏涛　1977 年生，硕士，研究员，主要从事航天测控系统软件总体设计工作。

冯　炜　1982 年生，硕士，高级工程师，主要从事航天测控系统软件设计工作。