面向微服务系统的运行时部署优化

徐琛杰 周 翔 彭 鑫 赵文耘
(复旦大学软件学院 上海 201203)
(上海市数据科学重点实验室(复旦大学) 上海 201203)

摘 要 传统云计算环境下的自动伸缩工具根据预先设定的条件为所有资源过少的微服务增加实例,所有资源过多的微服务减少实例,从而调节微服务占有的资源量。当集群资源不足时,这些工具可以动态地向集群中添加新服务器。但在集群资源固定且有限的场景下,这些工具无法向集群中添加新服务器,进而当集群资源不足时,将无法如期调节微服务实例数量。此外,这些工具需要为每个微服务单独配置伸缩方案,因此不适用于集群资源固定且有限的场景。针对以上问题,提出一个基于 MAPE(Monitor,Analyze,Plan,Execute)环路的自适应部署优化方法。在集群资源固定且有限的场景下,根据各个微服务实例的资源使用情况、微服务调用链及集群资源使用情况,选出一组微服务进行动态伸缩,以达到降低平均响应时间的效果。实现一个面向微服务系统的运行时监控及部署优化工具,并进行仿真实验。实验验证了在集群资源固定且有限的场景下部署优化方法的有效性。

关键词 微服务 可视化 监控 自动伸缩 自适应

中图分类号 TP311 **文献标识码** A DOI:10.3969/j.issn.1000-386x.2018.10.016

MICROSERVICE SYSTEM ORIENTED RUNTIME DEPLOYMENT OPTIMIZATION

Xu Chenjie Zhou Xiang Peng Xin Zhao Wenyun
(*Software School*, *Fudan University*, *Shanghai 201203*, *China*)
(*Shanghai Key Laboratory of Data Science*, *Fudan University*, *Shanghai 201203*, *China*)

Abstract The auto-scaling tools in the traditional cloud computing environment add instances for all the microservices with too few resources and reduce instances for all the micro services with too many resources according to the preset conditions so as to adjust the amount of resources occupied by the micro services. When cluster's resources are insufficient, these tools can dynamically add new servers to the cluster. However, when the cluster's resources are fixed and limited, these tools cannot add new servers to the cluster. The number of microservice instances cannot be adjusted as expected while the cluster's resources are insufficient. And these tools require a separate scaling scheme for each microservice. So these tools are not suitable for the scenario where the cluster's resources are fixed and limited. To solve the above problem, we proposed a self-adaptive deployment optimization method based on MAPE(Monitor, Analyze, Plan, Execute) loop. In the scene where the cluster's resources were fixed and limited, a set of microservices were selected and dynamically scaled according to the resource usage of each microservice instance, the microservice call chains and the cluster's resource usage, so as to reduce the average response time. A runtime monitoring and deployment optimization tool for microservice system was implemented, and simulation experiment was carried out. The results verify the effectiveness of the deployment optimization method under the condition that the cluster's resources are fixed and limited.

Keywords Microservice Visualization Monitoring Auto-scalability Self-adaptive

---

收稿日期:2018-04-08。国家重点研发计划项目(VGH2301001)。**徐琛杰**,硕士生,主研领域:微服务系统部署优化。**周翔**,博士生。**彭鑫**,教授。**赵文耘**,教授。

0 引 言

近年来,越来越多的公司如谷歌、亚马逊[1]等开始采用微服务架构。文献[2]将微服务定义为可独立开发、部署、测试和伸缩的应用程序。基于微服务架构的应用系统(下文简称“微服务系统”)相较于单体应用有易于开发、部署、维护和伸缩等优点[3]。

微服务系统在线运行时,不同微服务拥有的实例数量不一,而微服务占有的资源量与实例数量呈正相关关系。微服务实例越多,该微服务占有的资源就越多,反之亦然[4]。进而当请求数量较多时,若某个微服务实例数量过少,其占有的资源也会较少。这将导致该微服务出现平均响应时间急剧变长的现象,严重影响用户的使用。同时,微服务系统线上运行时,其负载一直动态变化,导致特定的部署方案无法有效降低系统的整体响应时间。特别是在集群资源固定且有限的场景下,需要根据负载的变化动态地调节各个微服务的实例数量,从而降低微服务系统的整体响应时间。

Kubernetes[5]、Marathon[6]等当下流行的微服务管理工具提供了微服务的自动伸缩[7]功能。这些工具根据预先设定的条件为所有资源过少的微服务添加实例、为所有资源过多的微服务减少实例,从而调节微服务占有的资源量。但这些工具均假定在云计算环境下运行,当集群资源不足时,可以动态地向集群中添加新服务器。在集群资源固定且有限的场景下,当集群资源不足时,因无法添加新服务器,这些工具将无法如期调节微服务实例数量。此外,这些工具需要人工为每个微服务配置伸缩方案,配置工作相对繁重。

针对以上问题,本文提出了一个基于 MAPE(Monitor、Analyze、Plan、Execute)环路[8]的自适应部署优化方法。在集群资源固定且有限的场景下,每轮 MAPE 循环时,通过分析微服务调用链及各个微服务实例的资源使用情况,根据调用链上缺少资源的微服务数量,挑选出数量最多的一条调用链。当集群资源充足时,为挑选出的调用链上缺少资源的微服务各添加一个实例。当集群资源不足时,先为所有存在空闲实例的微服务各减少一个实例,再根据减少实例后集群资源空闲情况,为挑选出的调用链上部分或全部缺少资源的微服务各添加一个实例。最终达到降低微服务系统整体响应时间的效果。在开源系统 Spring Cloud Sleuth 和 Zipkin 的基础上,本文实现了一个面向微服务系统的运行时可视化监控工具,基于该工具实现了一个面向微服务系统的运行时部署优化工具,并通过实验验证了部署优化方法的有效性。Train_ticket[9]是一个开源的、基于微服务架构的火车票订票系统,拥有超过 40 个微服务,本文基于该系统举例说明相关方法和技术,同时进行实验验证。

1 相关工作

谷歌开发的 Dapper[10]是一个面向分布式系统的追踪工具,通过向通信模块、流程控制模块和线程库等被绝大多数应用所使用的模块中插入探针,使 Dapper 能在收集追踪信息时保持对应用层透明。Spring Cloud Sleuth 是 Spring Cloud 提供的一个分布式追踪解决方案。Zipkin 是一个基于 Dapper 开发的分布式追踪系统,可以用来收集 Spring Cloud Sleuth 指导分布式系统产生的运行时数据并可视化。

文献[11]实现了一组监控和管理微服务系统的静态页面,对单个微服务的平均响应时间、吞吐量等进行了可视化。本文不仅实现了单个微服务的平均响应时间、吞吐量等的可视化,还实现了微服务调用链及线上部署情况的可视化,并结合收集到的数据进行了可视化展示。

Kubernetes 可以将单个微服务部署为一个 Pod,通过自动伸缩 Pod 的方式,实现微服务的自动伸缩。但 Kubernetes 的自动伸缩功能要求微服务系统运行在云计算环境下,从而可以动态地向集群中添加新服务器。Kubernetes 会为所有资源利用率超过预先设定值的微服务启动新实例,但如果运行的实例总数量过大,集群中的资源将不能启动全部新实例,Kubernetes 就会添加新服务器到集群中,但是这种做法不适合集群资源固定且有限的场景。此外,Kubernetes 还需人工为每个微服务单独配置伸缩方案,如微服务实例数量的上下限、期望的 CPU 平均利用率等限制条件,配置过程相对繁琐。文献[12]将微服务调用链定义为请求到达微服务系统后,微服务之间的相互调用所形成的有向无环图。本文提出的基于 MAPE 环路的自适应部署优化方法是在集群资源固定且有限的场景下,根据调用链来调节各个微服务的实例数量从而缩短微服务系统整体的平均响应时间,不需要人工为每个微服务配置伸缩方案,降低了配置的复杂度。

文献[13]提出了一个基于 MAPE 环路的自动伸缩方法,但该方法仅关注了 CPU 资源利用率,且每轮调节只调节一个节点,这导致在请求数量快速变化的场景下其调节速度较慢。本文提出的部署优化方法除关注 CPU 资源利用率外,还关注内存的使用情况,且每轮调节会根据调用链选择出一组微服务进行调节。

当请求数量快速变化时,本文提出的部署优化方法能快速调节微服务实例数量。

2　背景知识

微服务系统属于分布式系统,因此可将基于 Dapper 开发的面向分布式系统的开源工具 Spring Cloud Sleuth 和 Zipkin,用于研究微服务系统的行为。其中 Spring Cloud Sleuth 负责指导微服务产生 trace 信息,Zipkin 负责收集 trace 信息并展示。

用户对微服务系统的一次请求通常需要大量的微服务调用进行处理。Dapper 将针对一个特定请求的全部后台调用定义为一个 trace,将微服务实例间的一次远程过程调用定义为一个 span。Dapper 为每个 trace 分配一个唯一的 ID,称为 trace id,从而区分开了来自前台的不同请求;Dapper 为每个 span 分配一个唯一的 ID,称为 span id,从而区分开了不同微服务实例之间的相互调用。除最外层 span,每个 span 均有一个 span 作为自己的父亲,并将父亲 span 的 span id 作为 parent id 记录在 span 中。图 1 展示了一个简单的微服务系统请求处理流程(图中,六边形节点代表了微服务系统中的不同微服务实例),rpc1 对应的 span1,是 rpc3 对应的 span3 的父亲,也是 rpc4 对应的 span4 的父亲;而微服务实例 C 没有再调用其他微服务实例,因此 rpc2 对应的 span2 没有孩子。通过 span 之间的父子关系,同一 trace 下的众多 span 构成了一颗树,记录了微服务系统不同实例之间的调用关系。

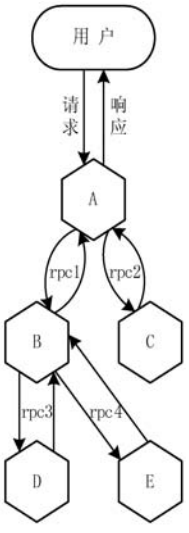

图 1　微服务系统请求处理路径图

Span 除记录 trace id、span id 和 parent id 外,还记录了 span 的 name(通常为被调用的接口)、远程过程调用的时间节点、被调用方的接口、IP 等众多信息,为研究微服务系统的行为奠定了基础。

3　可视化监控

为实现本文提出的基于 MAPE 环路的自适应部署优化方法,本文在开源系统 Spring Cloud Sleuth 和 Zipkin 的基础上,实现了一个面向微服务系统的运行时可视化监控工具。该工具从微服务可用性、微服务调用链及部署情况三个维度监控微服务系统的运行状态,并将结果可视化。

通过分析 trace 信息可以得到微服务可用性信息,微服务可用性信息包括每秒钟请求数量 QPS(Query Per Second)、平均响应时间、最短响应时间、最长响应时间和错误率(请求错误数量与请求总数量的比值)等信息。微服务系统的大量微服务在短时间内可产生庞大的 trace 数据。例如,将 Train_ticket 部署在 1 台服务器上,每个微服务拥有 1 个实例,以每秒 50 个请求的速度发送查票请求,持续一小时可以收集 32.6 GB 的 trace 数据,数据量巨大。当用户查询微服务可用性信息时,如果直接从数据库中取得数据进行统计,由于数据量巨大,往往需要很长的时间分析数据。为了提升系统性能,本文采用 MapReduce 编程模型[14]对 trace 数据进行预处理。

MapReduce 是一种用于处理大规模数据集的编程模型,该模型首先对数据进行映射(Map),然后对映射后的数据进行归约(Reduce),最终获得需要的结果。本文通过定时任务,每分钟统计一次微服务可用性信息。统计微服务可用性信息时先将 span 映射为统计微服务可用性的 avaCount,再将具有相同微服务名称、微服务实例 IP 地址和接口名称的 avaCount 归约,最终将归约后得到的 avaCount 转换为微服务可用性信息(即中间结果)。avaCount 中包含如下信息:微服务名称、微服务实例 IP 地址、接口名称、函数名称、查询数量、总响应时间、错误数量、最短响应时间和最长响应时间。将预处理获得的中间结果存储在数据库中。当用户查询微服务可用性信息时,只需分析数据库中的中间结果即可获得微服务可用性信息。

映射过程如图 2 所示。将 span 与 avaCount 中边框类型相同的项目进行映射。将 span 的名称映射为 avaCount 中的接口名称。将 span 的持续时间分别映射为 avaCount 的最短响应时间、最长响应时间、总响应时间。Span 的类型有 CLIENT、SERVER、PRODUCER、CONSUMER 四种。其中 CLIENT 对应同步或异步调用中的调用方,SERVER 对应同步或者异步调用中的被调用方;PRODUCER 对应消息队列中的生产方,CONSUMER 对应消息队列中的消费方。当 span 的类型为

SERVER 或者 CONSUMER 时，span 的 localEndpoint 中记录了被调用微服务的信息，serviceName 为微服务名称，Ipv4 为微服务实例 IP 地址，将以上两项分别映射为 avaCount 中的微服务名称和微服务实例 IP 地址。Span 中的 tags 记录了一些额外信息。如图 2 中 mvc. controller. method 为微服务被调用的函数名称，将其映射为 avaCount 中的函数名称。当被调用的微服务响应出现错误时，tags 中将包含 error，此时 avaCount 中的错误数量应置为 1。因 1 个 span 对应 1 个请求，avaCount 中的查询数量应置为 1。

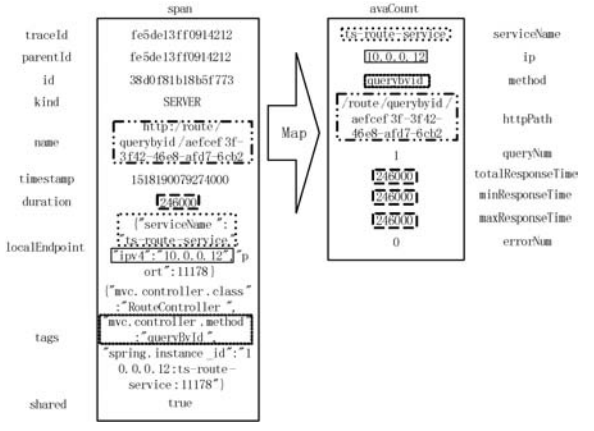

图 2 span 映射为 avaCount 示意图

归约过程如图 3 所示。avaCount1 和 avaCount2 的微服务名称均为 ts-route-service、微服务实例 IP 地址均为 10.0.0.12、函数名称均为 querybyid，因此可以将 avaCount1 和 avaCount2 归约为 reduced avaCount。接口名称中可能包含参数，因此取 httpPath1 和 httpPath2 的相同部分作为归约后的 httpPath。查询数量、总响应时间和错误数量均采用相加的方式归约。最小响应时间取 minResponseTime1 和 minResponseTime2 中较小者。最大响应时间取 maxResponseTime1 和 maxResponseTime2 中较大者。

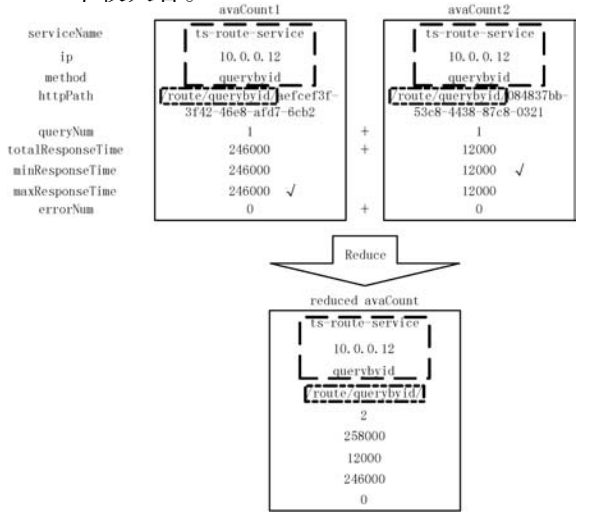

图 3 avaCount 归约示意图

图 4 给出了一个基于 Train_ticket 收集的微服务可用性信息表格，表格共八列，从左到右依次为：微服务名称、微服务实例 ID（通常为 IP 地址）、调用接口名称、查询时间内的 QPS、查询时间内的平均响应时间、查询时间内最短响应时间、查询时间内最长响应时间、查询时间内的错误率。表格中第 1 ~ 6 行分别为微服务实例 10.0.0.39 的 6 个不同接口的可用性信息，该实例属于微服务 ts-order-service。

Microservice Name	Instance ID	Http Path	QPS	Avarage Response Time	Min Response Time	Max Response Time	Error Rate
ts-order-service	10.0.0.39	/order/create/	0.016 7	203.000 0 ms	203ms	203ms	0.000 0
ts-order-service	10.0.0.39	/order/calculate	0.016 7	299.500 0 ms	28ms	571ms	0.000 0
ts-order-service	10.0.0.39	/order/gettbyid	0.050 0	11.666 0 ms	10ms	13ms	0.000 0
ts-order-service	10.0.0.39	/order/getticketlistbydateandtripid	0.041 7	82.600 0 ms	10ms	351ms	0.000 0
ts-order-service	10.0.0.39	/getorderinfoforsecurity	0.016 7	26.000 0 ms	26ms	26ms	0.000 0
ts-order-service	10.0.0.39	/order/modifyorderstatus	0.050 0	21.666 0 ms	10ms	41ms	0.000 0
ts-station-service	10.0.0.17	/station/exist	0.050 0	42.000 0 ms	5ms	198ms	0.000 0
ts-station-service	10.0.0.17	/station/queryforid	0.216 7	24.884 6 ms	5ms	414ms	0.000 0
ts-price-service	10.0.0.73	/price/query	0.025 0	278.333 3 ms	11ms	470ms	0.000 0
ts-seat-service	10.0.0.61	/seat/getseat	0.016 7	97.000 0 ms	97ms	97ms	0.000 0
ts-seat-service	10.0.0.61	/seat/getleftticketofinterval	0.033 3	962.500 0 ms	204ms	2 322ms	0.000 0
ts-order-other-service	10.0.0.7	/getorderotherinfoforsecurity	0.016 7	797.000 0 ms	797ms	797ms	0.000 0
ts-food-service	10.0.0.70	/food/getfood	0.016 7	5 356.119 0 ms	5 356ms	5 356ms	0.000 0
ts-preserve-service	10.0.0.81	/preserve	0.016 7	10 249.546 0 ms	10 249ms	10 249ms	0.000 0

图 4 微服务可用性

图 5 给出了一个 Train_ticket 中查票微服务的 QPS 随时间动态变化图。横轴为时间，纵轴为对应时间一分钟内的平均 QPS。图 6 给出了一个 Train_ticket 中查票微服务的平均响应时间随时间变化图。横轴为时间，纵轴为对应时间一分钟内的平均响应时间。

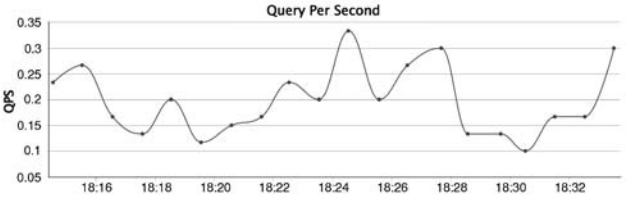

图 5 QPS 随时间动态变化图

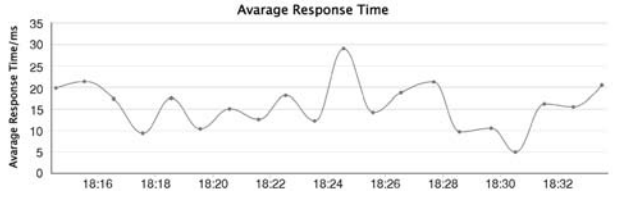

图 6 平均响应时间随时间动态变化图

本文基于 Zipkin 实现了微服务调用链的可视化。微服务调用链可视化展示了微服务间的调用关系，线条粗细表示了微服务间调用次数的多少，线条上标注的数字表示了各微服务调用其余微服务的可能性。图 7 给出了一个基于 Train_ticket 收集的微服务调用链的可视化界面，每个节点代表一个微服务，节点上的字母代表微服务的名称。例如：节点 ts-travel-service 代表微服务 ts-travel-service，节点 ts-ticketinfo-service 代表微服务 ts-ticketinfo-service。节点间的有向线段表示微服务间的远程调用关系，位于有向线段起点的微服务

调用位于有向线段终点的微服务。有向线段的粗细表示调用次数的多少,越粗调用次数越多。线段上标注了“数字 1:数字 2”,其中“数字 1”代表查询时间内远程调用的次数,“数字 2”代表有向线段起点的微服务调用线段终点的微服务的概率。如 ts-travel-service 调用 ts-ticketinfo-service 共 12 次, ts-travel-service 有 0.400 0的概率调用 ts-ticketinfo-service,且通过线段的粗细可以看出:ts-travel-service 调用 ts-ticketinfo-service 和 ts-route-service 的次数较多、调用 ts-seat-service 和 ts-train-service 的次数较少。

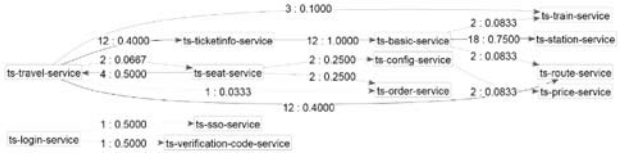

图 7　微服务调用链可视化

本文实现的可视化监控工具通过调用外部系统接口获取微服务系统线上部署情况。Kubernetes 和 Docker Swarm 等微服务管理工具通常提供了获取微服务实例部署情况的接口。为使开发和运维人员能更直观地了解微服务系统的线上部署情况,可视化监控工具通过 3D 图形来表示微服务系统的在线部署情况。图 8 给出了一个基于 Train_ticket 收集的部署情况的可视化界面,图中每个小球代表一个微服务实例,颜色相同的小球属于同一个微服务。单击小球可看到对应微服务的名称,如图中的 ts-food-map-service。可以明显看到小球聚集成两堆,聚集在一起的小球表示对应微服务实例部署在同一台服务器上,横坐标 server2 和 server1 则代表着对应服务器的名称。

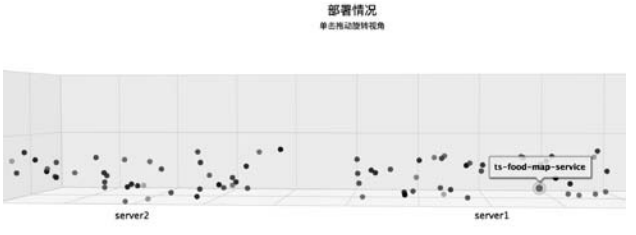

图 8　微服务系统线上部署可视化

4　基于 MAPE 环路的自适应部署优化

本文提出的基于 MAPE 环路的自适应部署优化方法,由监控模块、分析模块、规划模块和执行模块四部分组成。监控模块负责监控微服务系统中各个微服务的运行情况及服务器的资源使用情况;分析模块根据监控模块收集到的信息,分析各个微服务实例是否过载或空闲;规划模块根据微服务调用链、微服务实例资源使用情况、集群资源使用情况等信息,从分析模块找出的过载和空闲的微服务实例中挑选出需要进行动态伸缩的一组微服务,并生成最终的自动伸缩方案;执行模块根据规划模块生成的自动伸缩方案,通过调用外部系统接口实现动态伸缩微服务。

基于所实现的面向微服务系统的运行时可视化监控工具,本文实现了一个运行时部署优化工具。本文将微服务部署在 Docker 容器中,并由 Docker Swarm 管理微服务集群。Docker Swarm 提供了 spread、binpack、random 三种容器放置策略[15]。Spread 策略将容器尽可能平均地置于集群中的服务器上;binpack 策略将容器尽可能地置于已放置了容器的服务器上,以便将空余服务器用于放置未来较大的容器;random 策略随机选择服务器放置容器上。本文采用 spread 策略,使容器尽可能平均地分布在服务器上。

4.1　监　控

监控模块主要监控以下三项信息:trace 信息、服务器上 CPU 的平均利用率和内存的平均使用情况、各个微服务实例使用服务器 CPU 百分比。

本文实现的部署优化工具中,监控模块通过可视化监控工具收集对应时间内的 trace 信息;通过定时采样的方式,收集对应时间内服务器上 CPU 平均利用率和内存平均使用情况;通过 Docker Swarm 提供的 Docker Stats 指令[16],以定时采样的方式收集对应时间内各个微服务实例使用服务器 CPU 百分比。

4.2　分　析

分析模块通过分析监控模块收集到的各个微服务实例使用服务器 CPU 百分比,判断各个微服务实例的运行状态,进而判断各个微服务是否获得了足够的资源。

将各个微服务实例使用服务器 CPU 百分比超过上限的次数称为 upper、低于下限的次数称为 lower。若某个微服务实例的 upper 高于阈值,则认为该微服务实例过载;若某个微服务实例的 lower 高于阈值,则认为该微服务实例空闲。若某个微服务的全部实例过载,则认为该微服务处于资源过少的状态;若某个微服务的部分或者全部实例空闲,则认为该微服务处于资源过多的状态。

在判断微服务运行状态时,本文只通过微服务实例使用服务器 CPU 百分比来判断,而没有考虑请求数量。因为请求数量会直接表现在 CPU 利用率上,通过观察 CPU 利用率便可知请求数量的多少。如果 CPU 利用率过高,说明该微服务实例正在处理数量较多的请求;如果 CPU 利用率过低,则说明该微服务实例正

在处理数量较少的请求。

4.3 规 划

本文根据调用链和微服务实例资源使用情况,每次挑选一条调用链上需要添加实例的微服务及全部微服务中资源利用率过低需要减少实例的微服务进行动态伸缩。在挑选需要减少实例的微服务时,本文采取的方法类似于 Kubernetes,将该轮 MAPE 循环中资源利用率过低的微服务全部减少一个实例,而不是从一条调用链上挑选。因为通过实验发现,在集群资源固定且有限的场景下,能更快地减少实例就意味着能更快地为需要资源的微服务空出启动新实例所需的资源。

通过分析监控模块收集到的 trace 信息,可以获得对应的调用链信息。用户对微服务系统的一次请求,通常需要大量的微服务调用来进行处理。例如,网站 amazon. com 构建每个页面平均需要调用 100 ~ 150 个微服务[17]。被请求直接调用的最外层接口,我们称之为调用链的名称。调用链中包括了调用链的名称及所有被调用微服务的名称组成的列表。

获得了调用链信息后,就可以根据调用链信息和分析模块获得的微服务实例资源使用率超过上限的次数,判断该为哪条调用链上缺少资源的微服务添加实例;根据调用链信息和微服务实例资源使用率低于下限的次数,判断该为哪些微服务减少实例。算法 1 给出了基于调用链的微服务选取算法。

算法 1 基于调用链的微服务选取算法

```
01:输入:监控模块采样次数 n,微服务实例超过资源使用率上限的次数 microserviceId2Upper,微服务实例资源使用率低于下限的次数 microserviceId2Lower,微服务实例 id 列表 microserviceIdList
02:输出:需要添加实例的微服务列表 increaseList,需要减少实例的微服务列表 decreaseList
03:threshold ← n/2,微服务实例资源使用率超过上限次数大于 threshold 即为过载,微服务的全部实例均过载则该微服务过载
04:overloadList 初始化,过载微服务组成的列表
05:increaseList 初始化
06:decreaseList 初始化
07:for id in microserviceId2Upper. keySet( )
08:   if microserviceId2Upper. get(id) > threshold then
09:      name ← toServiceName(id)
10:      find ← false
11:      for id2 in microserviceId2Upper. keySet( )
12:         if ! id2. equals(id) &&
            name. equals(toServiceName(id2)) &&
            microserviceId2Upper. get(id2) < threshold then
13:            find ← true
14:         end if
15:      end for
16:   if ! find then
17:         overloadList. add(toServiceName(id))
18:      end if
19:   end if
20:end for
21:callChainOverload ← 挑选出包含缺少资源的微服务数量最多的一条调用链
22:   for microservice in callChainOverload
23:      if overloadList. contains(microservice)
24:         increaseList. add(microservice)
25:      end if
26:end for
27:for id in microserviceId2Lower. keySet( )
28:   if microserviceId2Lower. get(id) > threshold then
29:      name ← toServiceName(id)
30:      find ← false
31:      for id2 in microserviceId2Lower. keySet( )
32:         if ! id2. equals(id) &&
            name. equals(toServiceName(id2)) then
33:         find ← true
34:      end if
35:   end for
36:   if find then
37:      decreaseList. add(name)
38:   end if
39:end if
40:end for
```

算法 1 第 3 行,计算 threshold。第 4 ~ 6 行负责初始化。第 7 ~ 20 行,针对每个微服务的实例判断所属微服务是否过载。第 8 行,判断该实例是否过载。第 9 行,获取该实例对应的微服务名称。第 10 ~ 18 行,查找是否存在属于同一微服务的未过载的不同实例。若存在,说明这个实例过载是负载不均衡导致的,不应该通过加实例的方式解决该实例过载的问题;若不存在,说明该微服务全部实例过载,则此微服务应该增加实例,将其添加到过载微服务列表 overloadList 中。第 21 行,针对调用链缓存中的每条调用链,分别计算该调用链上需要添加实例的微服务数量,选择数量最多的一条调用链。第 22 ~ 26 行,将该调用链上需要添加实例的微服务添加到 increaseList 中。第 27 ~ 40 行,针对每个微服务实例判断是否应该将对应的微服务加入 decreaseList 中。第 28 行,判断该微服务实例是否空闲。第 29 行,根据微服务实例 id 获得微服务名称。30 ~ 35 行,判断是否存在属于同一微服务的不同实

例,若存在 find 为 true,不存在 find 为 false。第 36 ~ 38 行,将实例数量大于 1 的微服务加入 decreaseList 中。

获得了需要增加实例的微服务列表和需要减少实例的微服务列表后,根据集群资源使用情况,综合判断生成最终的部署调节方案。算法 2 给出了部署调节方案生成算法。

算法 2　部署调节方案生成算法

```
01:输入:集群资源利用情况 serverResources,服务器 CPU 利用上限 cpu_threshold,服务器内存利用上限 memory_threshold、需要添加实例的微服务列表 increaseList、需要减少实例的微服务列表 decreaseList
02:输出:最终的部署调节方案 changes
03:enough ← false
04:for serverResource in serverResources
05:     if serverResource. CPU  <  cpu_threshold &&
          serverResource. memeory  <  memory_threshold then
06:     enough ← true
07:end if
08:end for
09:if enough then
10:   if increaseList ！ = null then
11:      changes. add( increaseList)
12:   end if
13:else
14:   if decreaseList ！ = null then
15:      changes. add( decreaseList)
16:      if increaseList ！ = null then
17:       if increaseList. size( )  >  decreaseList. size( ) then
18:            list ← 从 increaseList 中随机选择 m 个微服务,其中 m = decreaseList. size( )
19:      changes. add( list)
20:   end if
21: else
22:      changes. add( increaseList)
23:   end if
24:end if
25:end if
```

算法 2 第 3 行负责初始化。第 4 行,针对每台服务器的资源使用情况进行判断。第 5 行,如服务器 CPU 和内存的使用率分别低于 cpu_threshold 和 memory_threshold,则认为该服务器资源充足。只要存在一台服务器资源充足,即认为整个集群资源充足。第 9 ~ 12行,若集群资源充足,则仅仅为需要添加实例的微服务各添加一个实例。原因在于,在集群资源固定且有空余时,不需要通过减少空闲微服务实例的方式来降低资源消耗。此外,这种做法还可以应对请求突增的情况。第 13 ~ 25 行,对应集群资源不足的情况。第 14 ~ 24 行,对应 decreaseList 不为空的情况,即存在微服务可以减少实例。第 15 行,将这些微服务加入 changes 中,为这些微服务各减少一个实例。第 16 行,若 increaseList 不为空,即存在微服务可以增加实例。第 17 ~ 20 行,若 increaseList 中微服务个数大于 decreaseList 中微服务个数,则从 increaseList 中随机选择 m 个微服务加入 changes 中,为这些微服务各添加一个实例,其中 m 为 decreaseList 中微服务的个数。因此时集群资源不足,当减少 m 个实例后,粗略认为可以增加 m 个实例到集群中。第 21 ~ 23 行,若 increaseList 中微服务个数小于等于 decreaseList 中微服务个数,则将 increaseList 中微服务全部加入 changes 中,为这些微服务各添加一个实例。但如果集群资源不足,且没有微服务可以减少实例时,不添加也不减少任何实例。

4.4　执　行

执行模块根据规划模块生成的自动伸缩方案,通过调用外部系统提供的接口来增加或减少某些微服务的实例数量,从而实现动态地伸缩微服务。Kubernetes、Docker Swarm 等当下流行的微服务管理工具均提供了动态调节微服务实例数量的功能。本文实现的部署优化工具中,执行模块通过远程调用 Docker Swarm 提供的伸缩微服务的 Docker update 指令[18]来添加或删除微服务实例。

5　实　验

为了验证本文所提出的基于 MAPE 环路的自适应部署优化工具的有效性,本文进行了仿真实验。本文使用 Train_ticket 作为微服务系统,将其和本文开发的部署优化工具共同部署,从而进行仿真实验。

5.1　实验设置

服务器为四台虚拟机(操作系统 Centos 7、8 * Intel Xeon E5649@2.53 GHz CPU、24 GB 内存、50 GB 硬盘、10 M/s 带宽)。微服务系统和部署优化工具部署在两台服务器上,另外两台服务器部署压力测试工具 Jmeter。微服务系统部署在 Docker Swarm 集群中,每个微服务部署两个实例,每个实例部署在一个 Docker 容器中,且一个 Docker 容器中只部署一个实例。

使用 Jmeter 模拟 400 个用户的并发访问,请求分别为登录和查票。如表 1 所示,登录对应调用链包含 3 个微服务,查票对应调用链包含 10 个微服务,且两条调用链包含的微服务没有交集。请求到达速度如图 9 所示,持续时间为 35 分钟。

表 1 登录和查票对应调用链包含微服务列表

登录	查票	
ts-login-service	ts-travel-service	ts-route-service
ts-sso-service	ts-order-service	ts-config-service
ts-verification-code-service	ts-ticketinfo-service	ts-basic-service
	ts-price-service	ts-station-service
	ts-seat-service	ts-train-service

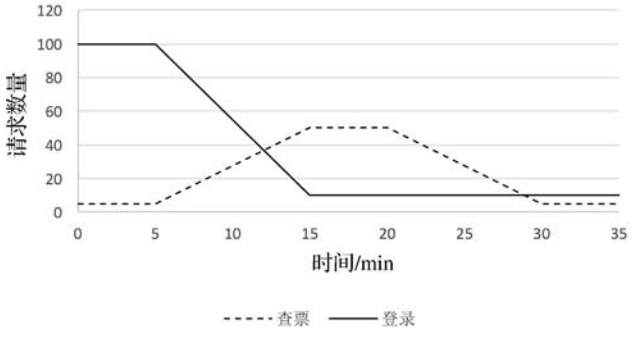

图 9 请求到达速度图

5.2 实验结果与分析

登录和查票的平均响应时间随时间变化图像如图 10 所示。前 2 分钟登录和查票的平均响应时间均较长,这与订票系统 Train_ticket 本身相关。在使用 Train_ticket 过程中发现,系统初启动时明显慢于启动一段时间之后。查看部署优化工具的日志发现,第 4 分钟由于集群资源不足,触发了扩缩容操作,共有 31 个微服务缩容、4 个微服务扩容。登录相关的 3 个微服务保持不变,查票相关的 4 个微服务执行了扩容操作、5 个微服务执行了缩容操作、1 个微服务保持不变,使得查票的平均响应时间在第 4 ~ 7 分钟变短。

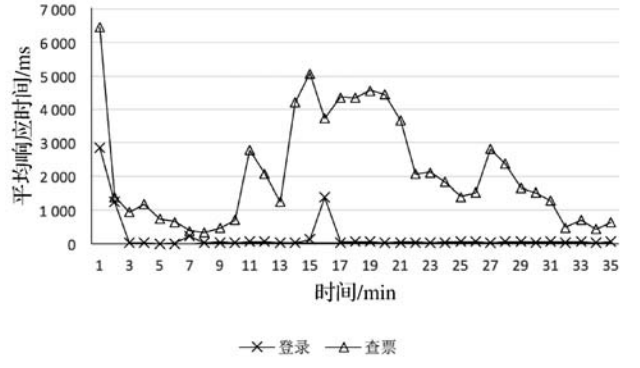

图 10 平均响应时间变化图

从第 5 分钟开始,查票请求数量逐步增加,登录请求数量逐步减少。第 7 分钟处,查票相关的 2 个微服务执行了扩容操作,使查票平均响应时间在第 7 ~ 8 分钟略微降低。

第 8 ~ 15 分钟,查票平均响应时间随着请求数量递增,在整体上呈变长的趋势。但在第 11 分钟处,查票相关的 7 个微服务执行了扩容操作,使得平均响应时间明显缩短。

第 16 分钟处,集群资源不足导致同时触发了缩容和扩容操作。登录相关的 3 个微服务缩容,查票相关的微服务中 5 个需要添加实例。但由于此时集群资源不足,且需要执行扩容操作的微服务数量多于执行缩容操作的微服务数量。因此从需要扩容的 5 个微服务中随机挑选了 3 个进行扩容,分别为 ts-seat-service、ts-route-service 和 ts-station-service。扩容操作在带来查票平均响应时间缩短的同时,缩容操作也使得登录平均响应时间在短时间内急剧变长。随后登录平均响应时间降至正常水平,这与登录的请求数量减少有关。登录相关的 3 个微服务初始部署时各有 2 个实例,可正常响应 100 req/s(request/s,每秒钟请求数量)的请求到达速度而不触发扩容操作。16 分钟后,虽然登录相关的 3 个微服务实例数量均变成了 1 个,但登录的请求到达速度已变为 10 req/s。

查票的请求到达速度在 15 ~ 20 分钟一直维持在 50 req/s。从 21 分钟开始,查票的请求到达速度开始下降。而在 21 分钟时,恰好因为 20 分钟时较高请求到达速度执行了扩容操作,使得第 21 和 22 分钟平均响应时间在扩容和请求数量变少的双重作用下显著缩短。

第 27 分钟处,查票相关的 6 个微服务触发了缩容操作,导致平均响应时间短暂变长,而后又随请求数量的降低而缩短。

综合以上分析,本文提出的基于 MAPE 环路的自适应部署优化工具,在集群资源固定且有限的场景下,能有效地调节微服务的实例数量、降低系统整体响应时间。但同时也发现了一些问题,第 15 ~ 20 分钟,查票请求到达速度维持在 50 req/s,此时为相应微服务执行扩容操作,虽能够降低平均响应时间,但仍然无法将平均响应时间缩短至第 5 ~ 9 分钟的水平。经分析 Train_ticket 发现,存在单个微服务的多个实例使用同一数据库服务的情况,如 ts-station-service 的四个实例均使用 ts-station-mongo。这使得系统的瓶颈转移至数据库处,此时再为 ts-station-service 增加实例,不能有效降低平均响应时间。另外,从第 10 ~ 13 分钟可以看出,本文提出的基于 MAPE 环路的自适应部署优化方法有一定滞后性,平均响应时间随请求数量增长一小段时间后,才会触发自适应调节操作。

6 结 语

本文提出一个基于 MAPE 环路的自适应部署优化

方法,在开源系统 Spring Cloud Sleuth 和 Zipkin 的基础上,实现了面向微服务系统的运行时监控和部署优化工具,并通过实验验证了部署优化方法的有效性。在集群资源固定且有限的场景下,部署优化方法能有效调节微服务实例数量、降低系统整体响应时间。

本文提出的基于 MAPE 环路的自适应部署优化方法还存在一些不足。不同微服务实例对服务器资源需求不同,当集群资源不足时需要先减少一些实例再启动相同数量的实例,但这是建立在不同微服务实例需要资源相当的基础上的。在一定程度上,为微服务添加实例可降低平均响应时间,但当实例数量到达一定程度时,系统的性能瓶颈可能转移到数据库等部分,此时继续添加实例无法有效降低平均响应时间。通过实验发现,自适应调节仍有一定的滞后性。

参 考 文 献

[1] Singleton A. The Economics of Microservices[J]. IEEE Cloud Computing, 2016, 3(5):16-20.

[2] Thönes J. Microservices[J]. IEEE Software, 2015, 32(1): 116.

[3] Dragoni N, Lanese I, Larsen S T, et al. Microservices: How To Make Your Application Scale[M]//Perspectives of System Informatics. 2018.

[4] Kang H, Le M, Tao S. Container and Microservice Driven Design for Cloud Infrastructure DevOps[C]//IEEE International Conference on Cloud Engineering. IEEE, 2016:202-211.

[5] Horizontal Pod Autoscaler[EB/OL]. Last Accessed: Feb 2018. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[6] Autoscaling with Marathon[EB/OL]. Last Accessed: Feb 2018. https://docs.mesosphere.com/1.7/usage/tutorials/autoscaling/.

[7] Hasselbring W, Steinacker G. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce[C]//IEEE International Conference on Software Architecture Workshops. IEEE, 2017:243-246.

[8] Kephart J O, Chess D M. The Vision of Autonomic Computing[J]. Computer, 2003,36(1):41-50.

[9] Train_ticket[EB/OL]. Last Accessed: Feb 2018. https://github.com/microcosmx/train_ticket.

[10] Sigelman B H, Barroso L A, Burrows M, et al. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure[R]. Google, Inc, 2010.

[11] Mayer B, Weinreich R. A Dashboard for Microservice Monitoring and Management[C]//IEEE International Conference on Software Architecture Workshops. IEEE, 2017:66-69.

[12] Leitner P. Modelling and Managing Deployment Costs of Microservice-Based Cloud Applications[C]//IEEE/ACM, International Conference on Utility and Cloud Computing. IEEE, 2017:165-174.

[13] Barna C, Khazaei H, Fokaefs M, et al. Delivering Elastic Containerized Cloud Applications to Enable DevOps[C]//International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE, 2017:65-75.

[14] Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters[C]//Conference on Symposium on Opearting Systems Design & Implementation. USENIX Association, 2004:10-10.

[15] Docker Swarm Strategy[EB/OL]. Last Accessed: Feb 2018. https://github.com/docker/docker.github.io/blob/master/swarm/scheduler/strategy.md.

[16] Docker Stats[EB/OL]. Last Accessed: Feb 2018. https://docs.docker.com/engine/reference/commandline/stats/.

[17] Mirjana S. Application Delivery Service Challenges in Microservices-based Applications[EB/OL]. Last Accessed: Feb 2018. http://www.thefabricnet.com/application-delivery-service-challenges-in-microservices-based-applications/.

[18] Docker Service Scale[EB/OL]. Last Accessed: Feb 2018. https://docs.docker.com/engine/reference/commandline/service_scale/.

(上接第 67 页)

[7] Giri B C, Chakraborty A, Maiti T. Pricing and return product collection decisions in a closed-loop supply chain with dual-channel in both forward and reverse logistics[J]. Journal of Manufacturing System, 2017,42:104-123.

[8] 倪明,莫露骅. 两种回收模式下废旧电子产品再制造闭环供应链模型比较研究[J]. 中国软科学,2013(8):170-175.

[9] 姚锋敏,滕春贤. 公平关切下零售商主导的闭环供应链决策模型[J]. 控制与决策,2017,32(1):117-123.

[10] 李明芳,薛景梅. 不同渠道权力结构下制造商回收闭环供应链绩效分析[J]. 控制与决策,2016,31(11):2095-2100.

[11] 赵晓敏,林英晖,苏承明. 不同渠道权力结构下的 S-M 两级闭环供应链绩效分析[J]. 中国管理科学,2012,20(2):78-86.

[12] 谷泽昊,郭志芳,王文利. 基于顾客渠道偏好改进的 Hoteling 模型的双渠道供应链零售商博弈[J]. 工业工程,2015,18(1):77-83.

[13] 张涛,顾峰,徐璐君,等. 基于 Hoteling 模型的回收企业竞争策略[J]. 系统管理学报,2015,24(5):537-544.

[14] 任方旭,章仁俊. 零售商主导型供应链强弱零售商间价格博弈研究[J]. 工业技术经济,2009,28(5):56-59.