学号 2019302110173

密级

武汉大学本科毕业论文

FaaSMaT: 基于独立内存分配的 FaaS 系统吞吐量优化策略

院（系）名　称：计算机学院

专　业　名　称：软件工程

学　生　姓　名：闫凯

指　导　教　师：胡创　研究员

二〇二三年四月

郑 重 声 明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名：闫凯　　日期：2023.5.13

摘　　要

函数即服务（FaaS）是一种相对新兴的云计算服务形式，它允许开发人员以函数的形式来构建、计算、运行和管理程序，无需关心基础设施的维护。然而，FaaS 系统中不同函数具有不同的调用频次、执行时间以及内存占用等特征。如果不依靠函数的特点对其进行合理的资源调度分配，FaaS 系统就难以达到最优的吞吐效率，在处理负载过程中会存在大量无意义的冷启动。FaaS 系统的计算资源会被浪费。系统使用者无法获得高质量服务，系统提供商的硬件资源无法得到高效利用。

针对以上问题，本文提出了优化策略 FaaSMaT。它可以通过历史记录定位到 FaaS 系统负载中具有较高调用频率的函数。之后依靠流量计算以及时间序列预测技术，为这些函数独立分配可动态变化的内存资源。FaaSM 借此优化 FaaS 系统的吞吐量，提高内存资源的利用率。

通过实验测试，对于存在多个高调用频次函数的 FaaS 负载，在采取 FaaSMaT 的优化措施后，FaaS 系统能够以原先十分之一级别的内存占用处理十倍级别的调用请求。FaaS 系统的吞吐效率和内存有效利用率都得到了有效的提升。

关键词：无服务；函数即服务；调用流量；时间序列预测；资源调度优化

ABSTRACT

Function as a Service (FaaS) is a relatively emerging form of cloud computing services that allows developers to build, calculate, run, and manage programs in the form of functions without caring about infrastructure maintenance. However, different functions in a FaaS system have different characteristics such as call frequency, execution time, and memory usage. If we do not rely on the characteristics of functions to schedule and allocate resources reasonably, it is difficult for FaaS systems to achieve optimal throughput efficiency. There will be a large number of meaningless cold starts in the process of handling function workload, and the computing resources of the FaaS system will be wasted. Users of FaaS systems cannot obtain high-quality services, and hardware resources of FaaS system providers cannot be efficiently utilized.

Aiming at the above problems, this paper proposes the FaaSMaT. It can locate functions with higher call frequency in the FaaS system workload through historical records. After that, it relies on traffic calculation and time series prediction techniques to independently allocate dynamically memory resources for these functions. The FaaSM system takes this opportunity to optimize the throughput of the FaaS system and improve the utilization of memory resources.

Through experimental testing, for FaaS workloads with multiple high call frequency functions, after taking FaaSM optimization measures, the FaaS system can handle ten times the level of call requests with the original memory footprint of one tenth. The throughput efficiency and effective memory utilization of the FaaS system can be effectively improved.

Key words: Serverless Computing; Function as a Service; Invoke Traffic; Time Series Prediction; Resource Scheduling Optimization

目　　录

图片索引

表格索引

1 绪论

本章主要介绍云服务的发展现状，简要描述了 FaaS 的基本特征。简要提出提出 FaaS 系统中，内存空间调度存在的问题以及优化方向。之后介绍了国内外与之相关的各方面研究。最后总结了本文的研究内容与主要贡献。

1.1 研究背景与意义

随着互联网技术的发展，云计算的地位与作用日益显著。云计算提供各种可访问的计算资源如应用、服务器、数据存储和开发工具等。用户可以直接使用云服务商提供的设施与功能，直接在云上部署自己的服务。相比传统的部署方式和开发流程，云服务具有更高的灵活性和可扩展性。对于难以维护庞大机房的中小企业来讲，云服务能够降低其 IT 成本，提高其开发部署的敏捷性并缩短其价值实现时间。云计算的服务形式多种多样，包括基础设施即服务 (IaaS)、平台即服务 (PaaS)、软件即服务 (SaaS) 等。随着云服务架构的不断发展与演化，在云计算落地已有十余年时间的今日，一种在相对意义上能够实现无限性能的架构已经成为了现实。业界称之为无服务架构（Serverless）。

在工业界，2012 年，Iron.io 公司率先提出了“无服务”[1]（Serverless，应该翻译为“无服务器”更合适，但现在称“无服务”已形成习惯）的概念，2014 年开始，亚马逊发布了名为 Lambda[2] 的商业化无服务应用，并在后续的几年里逐步得到开发者认可，发展成目前世界上最大的无服务的运行平台；到了 2018 年，中国的阿里云[3]、腾讯云[4] 等厂商也开始跟进，发布了旗下的无服务的产品。在学术界，2009 年，云计算概念刚提出的早期，UC Berkeley 大学曾发表的论文《Above the Clouds: A Berkeley View of Cloud Computing》[5]，文中预言的云计算的价值、演进和普及在过去的十余年里一一得到验证。十年之后的 2019 年，UC Berkeley 的第二篇有着相同命名风格的论文《Cloud Programming Simplified: A Berkeley View on Serverless Computing》[6] 发表，再次预言未来“无服务将会发展成为未来云计算的主要形式”，由此来看，“无服务”也同样是被主流学术界所认可的发展方向之一。

Serverless 架构目前没有特别权威的官方定义，但是业界一般认为其主要由“后端即服务”（Backend as a Service，BaaS）和“函数即服务”（Functions as a Service，FaaS）两种服务组成。FaaS 是业界研究的主流，也是本文研究的方向。

FaaS 作为一种云计算服务模式，具有很多优点。它提供了一种基于事件驱动的编程模型，程序员只需要编写业务逻辑代码，不需要考虑服务器的管理和扩展，更加方便快捷。同时，FaaS 的计费方式是按照执行时间收费，对于一些短时间内需要执行大量任务的场景非常适用。FaaS 平台的架构与其他云计算服务模式有所不同。FaaS 平台通常采用无服务器架构，即后端服务和管理平台由云服务提供商维护，开发者可以直接部署业务逻辑代码，无需考虑服务器的运维和扩展。由此使开发者脱离繁琐的服务器运维工作。

然而，FaaS 也存在一些问题。其中对 FaaS 平台服务质量影响较大的是冷启动[7–9]。FaaS 平台在接收到调用请求后会在虚拟机或容器（下文中以容器统称）中构建业务代码的执行环境，之后方能执行业务代码[8]。空闲的容器依照某种策略在保持存活一段时间后会被删除。此外，如果部署 FaaS 系统的机器可用内存有限，为了生成新的容器，即使未到达设定的存活时间，空闲的容器也会被 FaaS 系统采取某种策略驱逐[10, 11]。无论出于何种原因，在函数的容器被删除后，当此函数的调用再次到达时，就必须重复进行容器初始化以及代码运行环境初始化的工作。需要进行这些初始化的启动即为冷启动。相比直接在已完成初始化的容器中执行调用（热启动[7–9]），冷启动过程中有大量的时间被用于初始化工作。这会导致调用时间过长，系统整体的性能明显降低。在需要及时响应的情境下是难以忍受的[8, 9]。

针对通用情况下 FaaS 系统的冷启动优化问题，学术界已经有了许多研究成果[11–17]，各大云服务商也提供了其 FaaS 平台数据集以便于研究者进行分析调研[18, 19]。在对云服务商提供的函数调用记录进行分析后，我们发现 FaaS 系统中存在着许多调用频率很高的函数（例如：每分钟调用 1000 次以上的函数）。现有 FaaS 平台如果不能对这些调用频率较高的函数进行适当的处理，它们会对内存空间资源进行争用，进而导致系统整体吞吐效率的下滑和冷启动次数的上升。针对此类情景进行相应的优化是提高 FaaS 系统整体性能的有效途径，本文提出了 FaaSMaT 策略用于对此类情景进行优化。

总之，FaaS 作为云计算服务的一种新兴形式，具有很多优势和特点，但也面临一些挑战和问题。为了提高 FaaS 服务的质量和性能，需要不断优化其架构和服务质量，提升其可靠性和可用性，以满足用户的需求。本文在此背景下提出了基于调用流量和时间序列预测的吞吐量优化方法：FaaSMaT，。

1.2 国内外研究现状

1.2.1 Serverless 平台以及 FaaS 系统冷启动优化的相关研究

Serverless 和 FaaS 的概念自提出至今，已经有了长远的发展和演化。而 FaaS 系统的重要研究和优化方向就是对冷启动的优化。Guy 和 Assaf[8] 等人通过对大型云服务提供商的实际 FaaS 工作负载进行分析，提出了一些针对冷启动优化的策略，包括使用缓存、基于预测的方法和针对内存使用的优化等。Fuerst[11] 的 FaaSCache 通过将 FaaS 系统的容器池与缓存进行类比，实现了将 FaaS 容器池视作缓存池，利用缓存技术减少冷启动概率的 FaaS 缓存系统，能够有效的减少函数负载冷启动的次数。Catalyzer[20] 实现了一种基于新的检查点机制以及恢复应用程序和沙箱状态的方法，降低了在基于 gvisor 的沙箱环境中冷启动的初始化成本。Bermach[17] 提出了一种基于已知的 Application 信息来减少 FaaS 冷启动的方法。Jeganna 等[21] 实现了一种基于时间序列预测以减少冷启动时间的策略。减少冷启动时间或冷启动次数能够降低响应时间与延迟，同时降低成本，是提升资源利用率的有效途径。Wang[19] 提出了一种可以快速且可扩展地提供 FaaS 系统 runtime 地策略。Abad[22] 等人尝试将需要相同包的函数分配给相同的工作节点，提出了包感知调度算法。提高了包缓存的命中率，降低了云函数的延迟；同时考虑了工作节点所承受的负载，避免超出可配置的阈值。Oakes[23] 等试图通过将函数所需包缓存在工作节点来减少云函数的启动时间，并提出了共享包缓存，大大降低了响应时间与延迟。然而，这些方面忽略了函数调用的周期性和规律性，直接缓存依赖包或者启动容器将增加硬件资源的压力。

以上研究在不同方向上提出了行之有效的 FaaS 平台冷启动优化方法。但是他们忽略了函数调用的周期性和规律性，并没有考虑到具有独特调用特征的函数会给系统带来的资源压力。

1.2.2 时序预测的相关研究

时间序列预测技术已经有了长远的发展和十分丰富的研究成果。Sepp Hochreiter 和 Jürgen Schmidhuber 的经典论文[24] 提出了一种称为 LSTM 的递归神经网络结构，可以有效地处理时序数据，并在许多任务中实现了最先进的性能。Salinas 和 Flunkert 等人提出了一种新的递归神经网络结构，称为 DeepAR[25]，可以用于时间序列预测和生成，并通过有效的概率建模来提高预测准确性。[26] 等提出了一种用于时间序列建模的新型卷积神经网络结构，称为 TCN，可以有效地处理长期时间

依赖关系，并在多个时序预测任务中实现了最先进的性能。

在 FaaS 系统中使用时序预测是一种较为常见的优化策略[21]。时序预测负载变化可以提前对特殊情况做出反应，考证服务的可靠性并优化用户体验。[8, 27] 。徐[28]在设计自身的 FaaS 平台时依靠时间序列预测技术实现了一个函数冷启动加速器，能够相对有效的减少函数冷启动次数。Kesidis[29] 建议使用对函数资源需求的预测，使提供者能够在容器上超额预订功能。樊[30] 基于二次指数平滑法对过去的访问请求进行处理优化 Knative 框架取得了不错的效果。然而，这些预测方法忽略了趋势，季节与特殊节假日等多种因素，比如基于指数平滑的方法尽管能捉了每周的季节性，但忽略了长期的季节性。

1.2.3 资源调度分配的相关研究

由于现有平台的调度机制不能满足 Serverless 应用程序的独特特性，如突发、可变、无状态等，当前主要的方法是通过动态优化分配资源来提高 Serverless 平台的资源利用情况[31–33]。周[34] 等人对 FaaS 平台的资源调度做了深入调查和分析，其文章阐述了对于资源利用、响应时间以及多目标优化的 FaaS 平台资源调度的技术原理和相关研究现状。在此基础上，他们总结了 FaaS 平台资源调度未来的主要研究方向。Kaffes[31] 等通过维护集群资源的全局视图来提高调度器的资源利用情况。他们使用集中的方式消除了队列不平衡，采用以核心为粒度的方式减少了干扰，并提出了一种集中式的内核粒度调度器。这种设计的好处在于，把准备执行的函数直接分配给空闲的工作内核，避免不必要的排队，消除队列不平衡，进而提高平台的资源利用率。Jiang 等人[33] 运用混合作业分派方法提高了工作流的资源利用情况。他们同时考虑了工作流不同执行阶段的资源消耗模式，提出了一个 Serverless 的工作流执行系统。在大规模测试中，与传统的集群执行模型相比，大幅度简化了在公共云上执行大规模科学工作流所需的工作。王[32] 结合系统内存资源的使用情况，主动对容器池的容量进行伸缩，在匹配请求速率的前提下尽可能降低容器池的内存占用，翡[35] 提出了基于蚁群优化的虚拟机部署，实现了对启发式因子的优化、消耗状态优化以及信息素浓度更新优化。Li[36] 等人提出了一种对 FaaS 系统资源进行动态适应性伸缩的资源分配策略。

这些方法通常对单一资源目标进行建模，缺乏不同目标的联合考虑。

1.3 研究内容与创新点

FaaS 系统能够在用户无需主动管理和关心服务器资源的情况下为用户提供计算资源。对于用户来说，他们希望能够尽可能快速稳定地获得调用结果，这要求云服务商采取相应的策略来减少冷启动的发生频次。对云服务商来说，在减少用户调用冷启动发生率的同时，还要考虑到如何尽可能高效的利用其硬件资源。目前为止，已经有很多研究指出了内存分配大小对 FaaS 平台函数执行的影响[8]，也有了大量冷启动优化方向的研究。但是缺少一种针对特定的函数特征，对函数容器占用的内存资源进行调度的 FaaS 平台优化策略。

本文提出了 FaaSMaT：一种基于函数调用流量以及时间序列预测技术，进行独立内存空间分配的 FaaS 平台优化策略。FaaSMaT 的命名有以下含义：FaaS（函数即服务）；Seperated Memory（独立内存空间）；Time Series Prediction（时间序列预测）；Through Output（吞吐量）。FaasMaT 的命名将以上概念糅合为一体。

本文主要研究的内容和创新点有以下两点：

1. FaaS 计算平台模拟器的设计与实现。

   目前多数云服务商都提供 FaaS 的相关服务，如 Azure Function[37]，AWS Lambda[2]。也存在开源的 FaaS 平台如 Apache OpenWhisk[9]。但是这些平台具有较高的复杂度与耦合度，对其进行部署、调试、测试是一个漫长而又易出错的过程。如果想要对一个较大的数据集进行模拟测试，将其部署到相应平台上并进行测试更是一项及其繁杂的工作。基于以上痛点，我们使用 Java 语言实现了一个的 FaaS 平台模拟器[38]。基于 Apache OpenWhisk 的架构，我们模拟了 FaaS 系统的以下能力：读取 Azure 数据集模拟函数调用请求，基于消息队列进行消息生产与消费，模拟 Docker 的容器内业务执行，基于 LRU 机制进行容器驱逐，模拟容器复用与预热等功能。模拟器可以读取云服务商提供的 FaaS 平台调用数据，模拟调用的发生；同时记录下调用过程中产生的各项数据，便于进行研究分析。针对 FaaS 平台的优化设计可以直接实现在模拟器上。之后利用模拟器进行测试，能够快捷准确地观察到优化的效果。

2. FaaSMaT：基于调用流量和时间序列预测进行独立内存空间分配的 FaaS 系统优化策略。

   冷启动发生的原因在于函数的执行容器会由于不活跃或者平台的移除策略被移除。在具有较高负载的 FaaS 系统中，每一秒都会有大量的函数调用到达，他们会占用大量的内存空间用于生成和保存自己的容器。在需要生成新

的容器但内存空间不足时，平台会采取某种策略（OpenWhisk 采用 LRU）[11] 驱逐未使用的但也未超出存活时间的容器。但这样会导致新的冷启动，降低系统的整体效率的同时产生大量内存浪费。针对以上问题，本文提出了一种依靠吞吐流量获取高频高占用函数，然后为其分配独立内存空间的优化策略。

我们选用了数个 FaaS 系统函数集，使用其调用负载在模拟器上对优化效果进行了测试。经验证，采取 FaaSMaT 策略后，即使 FaaS 系统容器可用内存空间较小（例如：4G），平台整体相比未经优化地初始状态也能获得数倍的热启动率和函数处理率。与此同时，FaaS 平台的内存空间的使用效率也得到了有效提升。

1.4 论文结构

本文共分为五章，各章节内容如下：

第一章：绪论。这一章主要介绍了研究的背景和意义。首先介绍了云计算以及 FaaS 平台的发展现状、所面临的挑战。之后阐述了国内外对于 FaaS 平台冷启动和资源调度相关问题的研究现状。最后引出本文的主要研究内容与贡献。

第二章：相关技术与研究动机。这一章主要介绍了函数即服务（FaaS）的基本架构，函数负载的动态特征，并分析了无服务器计算环境下的内存资源占用的问题。首先介绍无服务器计算的基本概念与模型，包括数据路径，控制路径，调度及资源分配策略等，随后结合公开的数据集分析无服务器计算的负载特征，最后结合特征数据分析了调用超时问题产生原因及优化方向。

第三章：系统设计。这一章在前文的观察与讨论的基础上对要解决的问题进行逐个分析，并提出了各个问题的解决方案；简要展示了采用 FaaSMaT 改进后 FaaS 系统的结构总览；之后介绍了 FaaSMaT 各个模块的详细设计方案。在最后一部分，简要的说明了 FaaS 系统模拟器的设计与实现。

第四章：实验与分析。这一章基于前文对 FaaSMaT 的设计与实现，在模拟器上对独立空间分配策略的优化效果进行了实验验证。此章节首先介绍了实验所采用的函数负载与设置；之后展示并分析了实验的统计结果；最后总结了 FaaSMaT 优化的效果。

第五章：讨论与总结。这一章对全文内容进行了总结讨论，提出了进一步优化改进的思路与可能。最后对全文内容进行了总结。

2 相关技术与研究动机

这一章主要介绍了函数及服务系统（FaaS）的基本架构，函数负载的动态特征，并分析了无服务器计算环境下的内存资源占用的问题。首先介绍无服务器计算的基本概念与模型，包括数据路径，控制路径，调度及资源分配策略等，随后结合公开的数据集分析无服务器计算的负载特征，最后结合特征数据分析了调用超时问题产生原因及优化方向。

2.1 FaaS 技术介绍

2.1.1 FaaS 基本概念

无服务器架构是当前云计算领域的热门趋势。自亚马逊发布 AWS Functions[2] 服务以来，随着需求不断更新，出现了许多无服务器架构的服务提供商。Serverless 架构主要由“后端即服务”（Backend as a Service，BaaS）和“函数即服务”（Functions as a Service，FaaS）两种服务组成。

1. 后端即服务（BaaS）是由服务提供商为客户提供整合好的程序功能或 API 以提供服务的服务。包括文件存储、数据库、推送服务、身份验证等。例如 AWS 的 S3[39] 存储桶和 RDS 数据库[40] 都属于这种服务类型。
2. 函数即服务（FaaS）是一种事件驱动的、由消息触发的服务。目前最广泛应用的是 AWS 提供的 AWS Lambda[2]。服务提供商一般会集成各种同步和异步的事件源，通过订阅这些事件源，可以突发或者定期的触发函数运行。大部分服务提供商所提供的 FaaS 功能是，提供一个平台，允许客户上传应用程序代码并对其进行管理，同时通过服务商所提供的触发器来触发客户的应用代码，从而免除了客户配置基础架构的步骤。按照此模型构建应用是实现无服务器架构的一种方式。

与传统互联网应用架构不同，无服务器架构是当今 IT 行业中最有力的工具之一。开发者在构建应用的过程中无需关心计算资源的获取和运维，由服务提供商按需分配计算资源并保证应用执行。2018 年，Google 发布了 Knative[41] 为 Serverless 提供了标准化的解决方案，备受各界关注。基于事件驱动的无服务器计算已经有了许多典型的应用场景，这些场景包括物联网、移动应用和聊天机器人。无服务

器计算中的事件主要有以下来源：传感器、人的动作（提交代码、发送请求）和上一个数据流或事件。对于已经构建完毕的应用来说，将其变更成无服务器架构是有代价的。如果需要保证可靠性或需要长时间运行的应用，这样的应用就不适合变更成无服务器架构的应用；但并不意味着这样的应用无法实现无服务器化，只需要对应用进行重新设计。目前无服务器服务提供商主要提供 FaaS 服务，因此在介绍无服务器服务时，一般指的是各大服务提供商所提供的 FaaS 服务。

容器[42] 技术是 FaaS 架构实现的另一个基础，容器和 FaaS 在技术上有相同的地方，如结合容器的特点和 Kubernetes[43] 这种容器编排平台，用户可以实现对容器应用的自动弹性伸缩。Kubernetes 对底层的主机资源进行抽象，在一般场景下，用户也不关注容器应用具体运行在什么主机上。容器平台的最小运行单元是容器，虽然目前容器内一般运行的是一个完整的应用，但是容器内运行的对象变成函数并无技术困难。Kubernetes 上默认没有事件触发的支持，无法做到按需部署容器应用。但是通过 Kubernetes 叠加上一些 FaaS 框架，运行包含函数逻辑的容器，用户很容易使 Kubernetes 具备 FaaS 服务的能力。因此，容器是构建 FaaS 能力的一个重要实现基础。当前许多 PaaS 平台也开始支持容器，或是以容器为技术架构的基础，如 OpenShift[44] 就是一个以 Docker 和 Kubernetes 为基础的开源 PaaS 平台。

相比于传统云计算，FaaS 平台具有着运维自动化、按需加载、弹性伸缩、强隔离性、敏捷开发部署等技术特点，带来了降低人力成本、降低风险、降低基础设施成本、降低交付时间等核心优势。自其问世至今，各大云服务商都为用户提供了 FaaS 功能，包括但不限于：亚马逊的 AWS Lambda[2]，谷歌云平台的 Google Cloud Functions[45]，微软 Azure 的 Microsoft Azure Functions[37]，IBM 的 IBM Cloud Functions[46] 以及阿里巴巴的 Alibaba Function Compute[3]。这些平台都提供了强大的功能和扩展性，使得开发人员可以更轻松地构建和部署服务器端应用程序，从而更专注于应用程序的逻辑和功能。

2.1.2 FaaS 系统架构

FaaS 平台的架构通常由以下几个组件构成[8]：

1. FaaS 引擎：负责运行函数代码的核心组件，通常采用容器技术来隔离函数执行环境，确保函数之间互不干扰。
2. API 网关：用于接收外部请求并将其转发到相应的函数执行器，还可以提供访问控制、负载均衡和监控等功能。

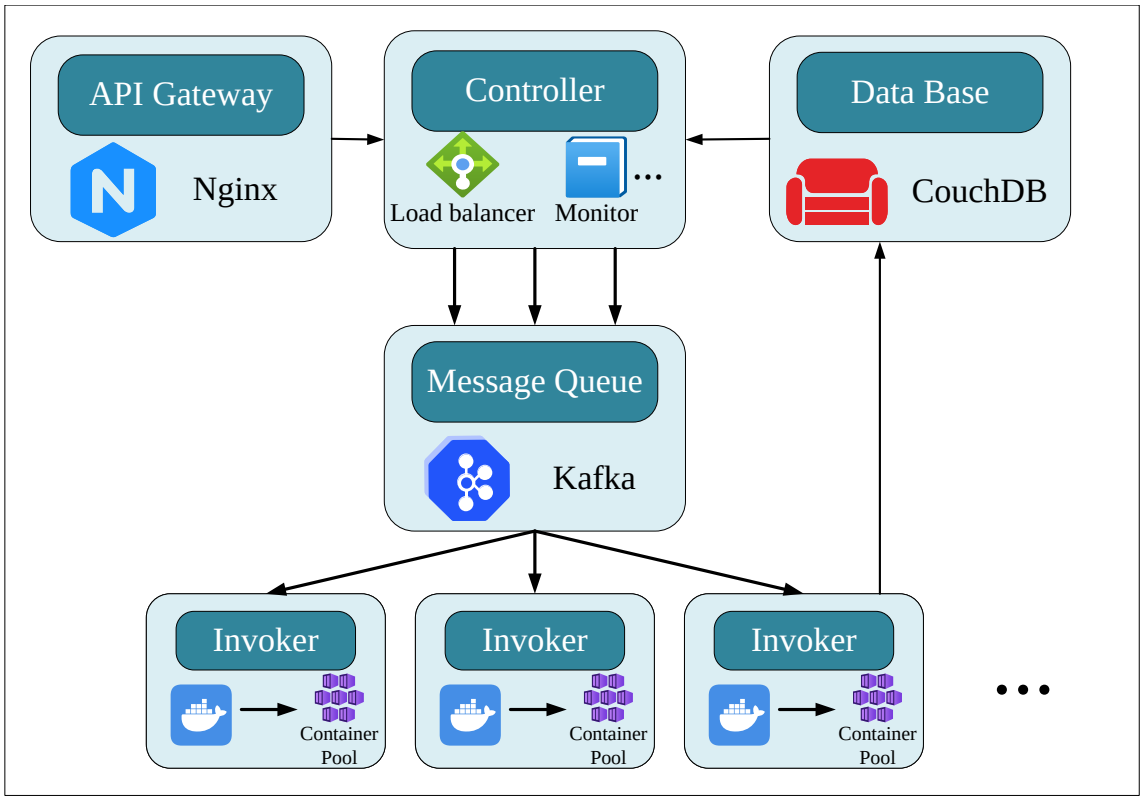


图 2.1　OpenWhisk 架构

3. 系统控制器：控制器负责协调和管理 FaaS 平台的各种资源和组件，履行调度、扩展等职能。
4. 函数执行器：运行函数代码的实例，负责处理请求、调用其他函数、访问外部服务等操作。
5. 存储服务：用于存储函数代码、运行时状态和其他相关数据。通常包括对象存储、数据库等组件。
6. 监控服务：用于收集和展示平台和函数的运行指标、日志和报警等信息。

不同的 FaaS 平台可能在这些组件的具体实现和架构方面存在差异，但基本的功能和原理是相似的。以学术界常用的开源 FaaS 平台 OpenWhisk[9] 为例：OpenWhisk 使用 Docker[42] 作为容器引擎、Nginx[47] 作为 API 网关、CouchDB[48] 提供存储服务以及 Kafka[49] 作为消息队列。配合上其使用 Scala[50] 语言编写的 Controller 与 Invoker，完整地提供了一个成熟 FaaS 系统所需的全部组件与功能。OpenWhisk 的系统架构如图 2.1所示。

在上述架构下，FaaS 计算中的常规函数执行工作流程往往包括数个步骤，如图 2.2所示。

用户在客户端通过 Http 或命令行传入一个自定义的函数调用请求，请求信息被传递给 Controller（此过程中往往会有反向代理）。然后，Controller 从数据库中

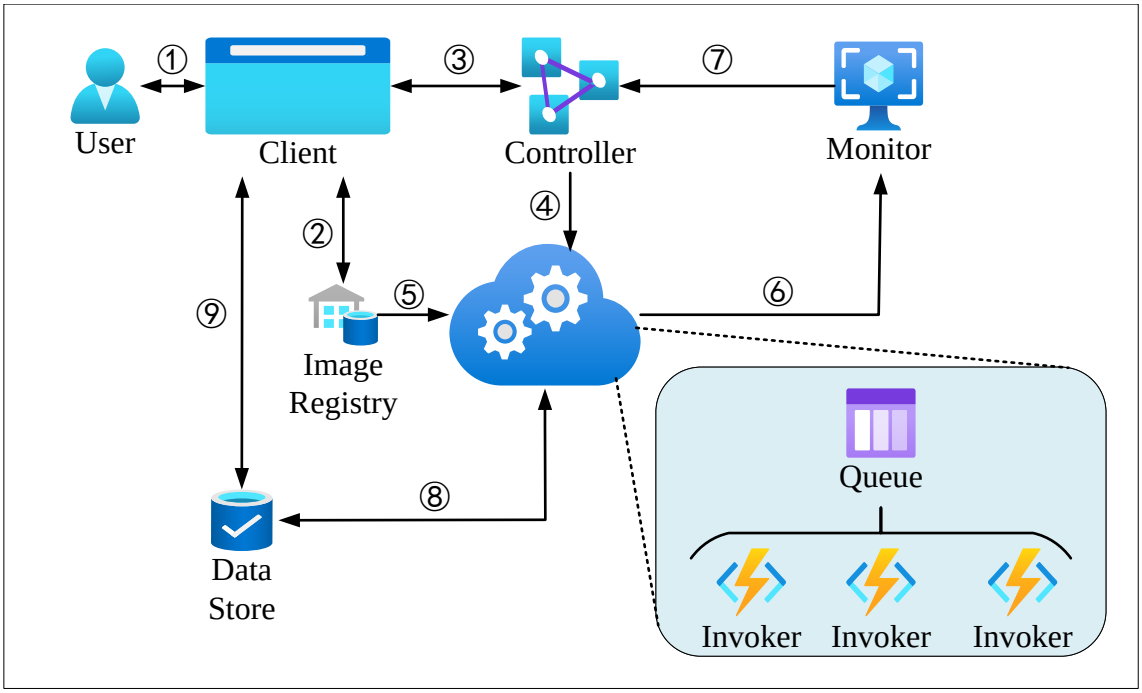


图 2.2　FaaS 系统调用流程

获取并验证用户的身份和请求的有效性，并通过 Load Balancer 分配一个合适的运行时环境（在本模型中是 Docker 容器），将请求分派到实际执行环境中。RunTime 从 Registry 调取镜像，启动并初始化容器（若 Registry 中没有镜像则进行镜像拉取）。在执行过程中，Monitor 检查资源使用情况，并将其发送给 Controller 进行负载均衡控制。

当执行完成后，结果被存储在数据库中。用户可以通过客户端查询结果。在执行环境内部，消息队列作为一个缓冲存储请求，以避免高并发导致的冲突。实际的运行时 RunTime 通常为 Invoker，每个 Invoker 都有各自的容器池来运行函数。[9]

2.1.3　热启动缓存与容器复用

出于提高资源利用率的考量，FaaS 系统往往都会采用热启动缓存和容器复用策略[8, 9]。热启动缓存通常是基于某种预热机制实现的，例如周期性地预热容器实例，或者将容器实例缓存在内存中，以便快速响应后续请求。热启动缓存的目的是为了提高函数的响应速度和性能，以实现更好的用户体验。容器复用是指在函数执行完毕后，将容器实例保留一定时间，称之为 keepAlive，等待后续的请求使用，从而避免重新实例化容器和加载资源的时间和成本，提高系统的响应速度和资源利用率。容器复用的目的是为了尽可能减少资源的浪费，提高系统的资源利用率。

不同的 FaaS 平台会采用不同的热启动缓存机制和容器复用策略，而针对这方

面的研究是 FaaS 平台冷启动优化的热门方向[8, 11, 20, 21, 27]。需要了解的是，OpenWhisk 的采用的容器复用策略是固定 10 分钟的 keepAlive[9] 时间。**意为一个 OpenWhisk 的函数容器在函数完成执行后，如果 10 分钟内没有新的、同一函数的调用到达，那么此容器将被销毁**。

除空闲 10 分钟的销毁机制外，OpenWhisk 中的还有另一种空闲容器销毁机制。**不考虑多个 Invoker 进行负载均衡的前提下**，单个 Invoker 想要从从消息队列中取出一个函数调用请求去处理，如果此时容器池中没有此属于此函数的、处于 keepAlive 状态的容器，且容器池空间不足以生成新的容器，那么 OpenWhisk 会根据 LRU（最近最久未使用）原则销毁掉 keepAlive 时间最久的容器，为新容器的生成腾出空间[9–11]。（例如：在已经等待 9min，8min，5min 的三个 keepAlive 容器中销毁掉已经等待 9min 的容器。）销毁后空间依旧不足则继续以 LRU 机制销毁空闲容器，直到空间充足。

Invoker 在执行上述操作前会提前检查能否通过销毁空闲容器腾出足够空间。若所有容器都在执行函数，不存在可销毁的空闲容器，那么调用请求会继续存放在消息队列中。调用请求若存放在消息队列太久中未被处理，视作请求过期，系统将不再尝试处理该请求。此种情景本文中称为 **drop**。

一次 drop 意味着一次调用请求由于在消息队列中等待超时而未能成功执行。

请注意，OpenWhisk 所采取的 LRU 销毁策略，以及由此引发的 drop 结果，是本文进行吞吐量优化的重要动机与方向。实际生产环境中，云服务商往往能提供理论上“无限”的内存资源，因此 drop 的情景并不常见。本文的研究建立在内存资源受限的情境下，这样能更好观察系统吞吐率提升的效果。

2.2 研究动机与研究目的

2.2.1 FaaS 系统负载模拟与统计

近年来，许多云服务商对外开源了其所采集的 FaaS 系统调用记录，以便于研究者进行研究与分析。这些调用数据对实际的 Function 进行了哈希匿名，但提供了其他详细的信息。如函数调用时刻，调用次数，执行耗时，所占内存等数据。Azure 提供的 Azure Functions 数据集 Azure Functions Data Set 2019[18] 是公开数据集中较为详细的一个，它完整提供了 2019 年 7 月的 12 天中在 Azure Functions 上运行的部分应用程序的以下信息：

1. 每分钟每个（匿名化）函数被调用的次数及其相应的触发器组。

2. 函数被分组为（匿名化）应用程序的方式，以及应用程序被（匿名化）所有者分组的方式。
3. 每个函数的执行时间分布。
4. 每个应用程序的内存使用分布（容器占用内存）。

大致信息如表 2.1所示。

表 2.1　Azure 数据集信息概览

名称信息	内存信息	执行时间信息	调用信息
哈希函数名	极值	极值	每分钟调用次数
哈希 App 名	均值	均值	
触发器	百分比分布	百分比分布	

我们分析了 Azure Functions Data Set 2019 提供的完整 12 天数据中第一天的数据集（以下简称数据集），并在模拟器中对其中一部分函数的调用进行了模拟。数据集中共有 36000+ 函数，我们对其进行了采样。数据集中 3 万余函数被按照调用频率四等分，每份中随机采样 100 个函数，共采样 400 个函数。这 400 个函数组成了一个具有代表性的函数集合，既包含一天内偶尔调用的函数（例如，一天调用五次以下），也包含一天内调用极频繁的函数（例如：每分钟调用在 1000 次以上）。数据集中函数调用请求频次的统计如表 2.2所示。

表 2.2　被采样函数调用请求频次统计

一天内调用频度	函数数量
< 5	129
>= 5 && < 10	23
>= 10 && < 100	98
>= 100 && < 1440	96
>= 1440 && < 144,000	38
>= 144,000 && < 1,440,000	13
>= 1,440,000	3

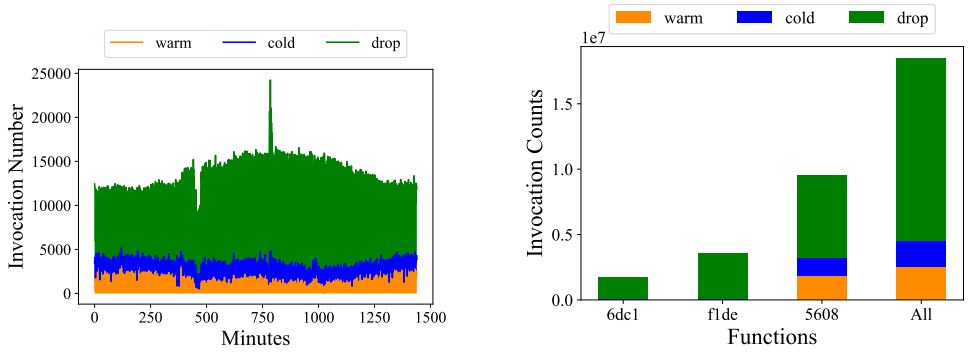
随后我们生成了这 400 个函数在一天内被调用的过程，并在模拟器中模拟了 FaaS 系统对这 400 个函数一天 1440 分钟内共计 18,452,673 次的调用。调用结果分为热启动（warm），冷启动（cold）以及超时未能处理（drop）三种。在模拟中，系统设定的条件为：系统消息队列中调用过期时间为 **10ms**，可用的容器池内存大小为 **6G**。（实际生产环境中 FaaS 系统部署在集群上，可以提供理论上“无限”的内存资源，出于研究优化的目的，我们对内存空间设置了上限。）容器执行的并行度

为 **1**。（单个容器同一时刻只能处理一个调用）

为了准确描述 FaaS 系统对函数调用请求的处理效果，**本文定义：系统对于函数负载的吞吐率为: 系统成功调用次数（热启动次数 + 冷启动次数）与总调用请求次数之比**。

$$\textbf{吞吐率} = \textbf{（热启动次数 + 冷启动次数）/总请求次数} \tag{2.1}$$

调用模拟的结果如图 2.3a所示。



(a) 1440 分钟数据集整体调用结果统计

(b) 三个高调用频次函数与数据集整体的调用结果对比

图 2.3　调用结果统计总览。函数 6dc1,f1de,5608 有着很高的 drop 频次，导致系统对于此函数集的整体吞吐率很低。

观察调用结果图，我们发现：在系统的容器池可用内存为 16G 时，系统整体的吞吐率 (2.1) 很低。有大量的请求因为在消息队列中等待超时而未能被成功处理（drop），而且系统还有着相当高的冷启动次数。

为了了解为何系统的吞吐效率会如此低，我们统计了被模拟的每个函数的调用结果记录，包含每个函数的热启动，冷启动以及超时的次数。观察发现，400 个函数中有 3 个调用请求频次极高的函数。这 3 个函数有着每分钟 1000 次以上乃至每分钟 7000 次以上的调用请求。这些调用在热启动情况下所耗费的执行时间并不是特别长，平均只有几十毫秒到几百毫秒，详见表 2.3。但是这些函数的调用请求有相当大一部分因为超时而未能被成功处理。也正是因为他们请求频次极高（一天可达数百万乃至千万次），且超时的频次也极高，进而导致系统总体的吞吐率显得很低, 如图 2.3b所示。

我们尝试进一步分析这些调用请求频次高的函数处理率低的原因。考虑到容器池内存相对受限条件的影响，我们统计了这些高请求频次函数的容器在 1440 分钟内每分钟的内存占用，结果如图 2.4所示。可以发现，即使这些高频函数单个容

表 2.3　高请求函数信息概览

名称	平均内存占用	热启动执行时间	冷启动执行时间	平均每分钟调用次数
6dc1	72M	94ms	18,435ms	1,216
f1de	89M	440ms	36,904ms	2,466
5608	30M	30ms	10,232ms	6,608

表 2.4　高请求函数调用结果一览

名称	热启动次数	冷启动次数	超时次数
6dc1	10,388	76,387	1,665,578
f1de	20,624	57,516	3,473,657
5608	1,852,207	1,345,451	6,317,965

器所占内存并不大 (表 2.3)，但是他们依然会占用大量的容器池内存用于存放自己的容器。单个容器内存可能只有几十 M，但总体却能够占用数 G 空间来存放上百个容器。**这意味这些函数在同一时刻需要大量的容器来并行处理自己的调用请求**。

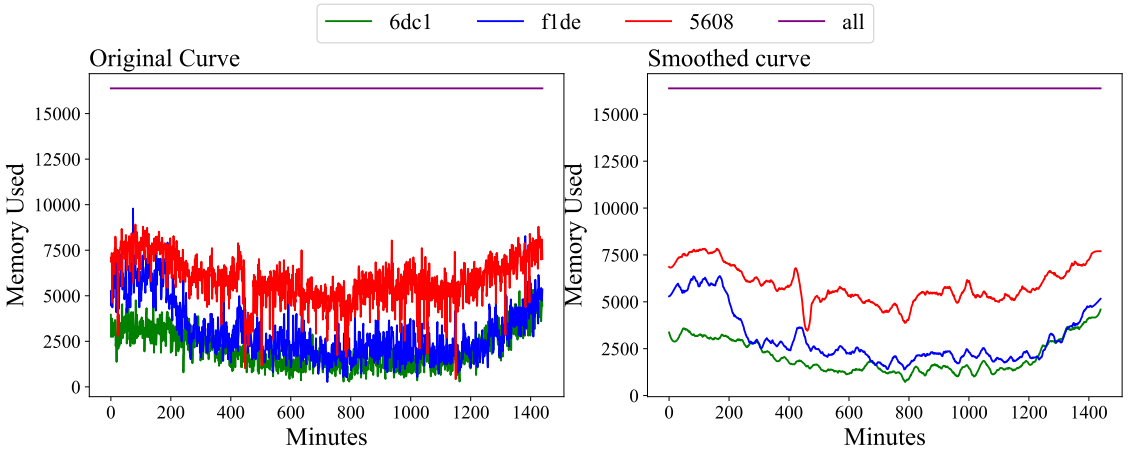


图 2.4　三个高请求频次函数的容器在 1440 分钟内所占内存变化。左图未作处理，右图将曲线平滑处理。

但是理论上来讲，由于这些函数单次执行时间并不长，都在毫秒级，在每分钟内调用请求到达相对均匀的情况下，即使每分钟有数千次调用要执行，数个或者数十个并行存在的同类型容器也应当足以满足调用需求。而且由于他们的调用频次高，容器会持续处于活跃状态，理论上应该有很高的热启动次数。

理想状态下的情景如图 2.5所示。

然而正如图 2.4所示，这三个函数分别申请了数 G 的空间用于存放其容器，实际占用的内存空间大小远远超出了其理论上的实际需求。此外，图 2.4的左图显示，这三个函数的容器占用的空间并不稳定，在不同分钟间剧烈上下波动，这代表其

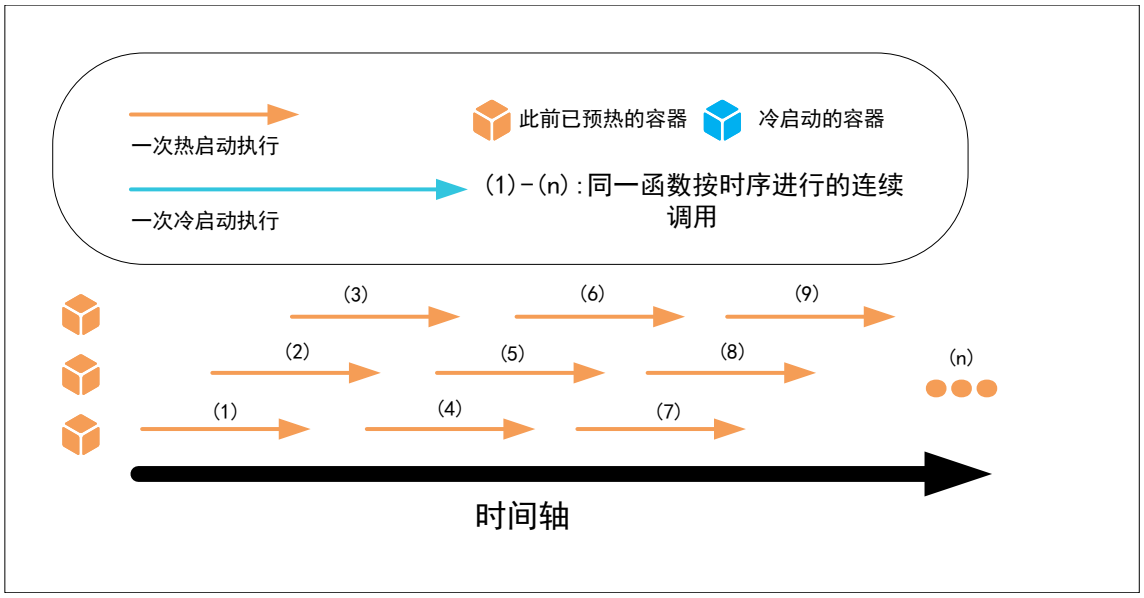


图 2.5　理想情境下，数个容器即可并行处理同一函数高频次到达的调用，且由于容器持续活跃，这些请求处理皆为热启动。

容器数量会剧烈波动，和图 2.5中数个并行容器稳定存在的设想去甚远。

为了探求其根本原因，我们综合分析了这 3 个高请求频次函数在 1-1440 分钟每分钟热启动，冷启动和超时的次数，以及他们各项指标的影响。数据如表 2.3和表 2.4所示。结果发现，这些高频函数并没有出现理论中较为高频的热启动次数。与之相反的是，每分钟这些函数都会经历相对热启动次数而言很高的的冷启动次数。此外，这些函数的冷启动时间可以是热启动时间数十倍乃至百倍。

2.2.2　系统低吞吐率原因

综合以上信息与分析，我们意识到，模拟结果中极低的吞吐率以及异常的冷启动率可能是由两个原因造成的：

1. 冷启动时的所需的容器数量暴增。
2. 容器可用内存空间不足导致的空间资源争用。

冷启动时的容器数量暴增的情景如图2.6所示，由于冷启动时间远大于热启动时间，因此在发生冷启动时如果有新的调用到达，冷启动的执行尚未完成，容器不空闲。此时需要生成新的容器来并行处理调用。冷启动时间相比调用间隔时间越大，所需新生成的容器数量就越多。因此，对高频函数而言，在冷启动发生时会出现同一函数容器数量暴增的情况。

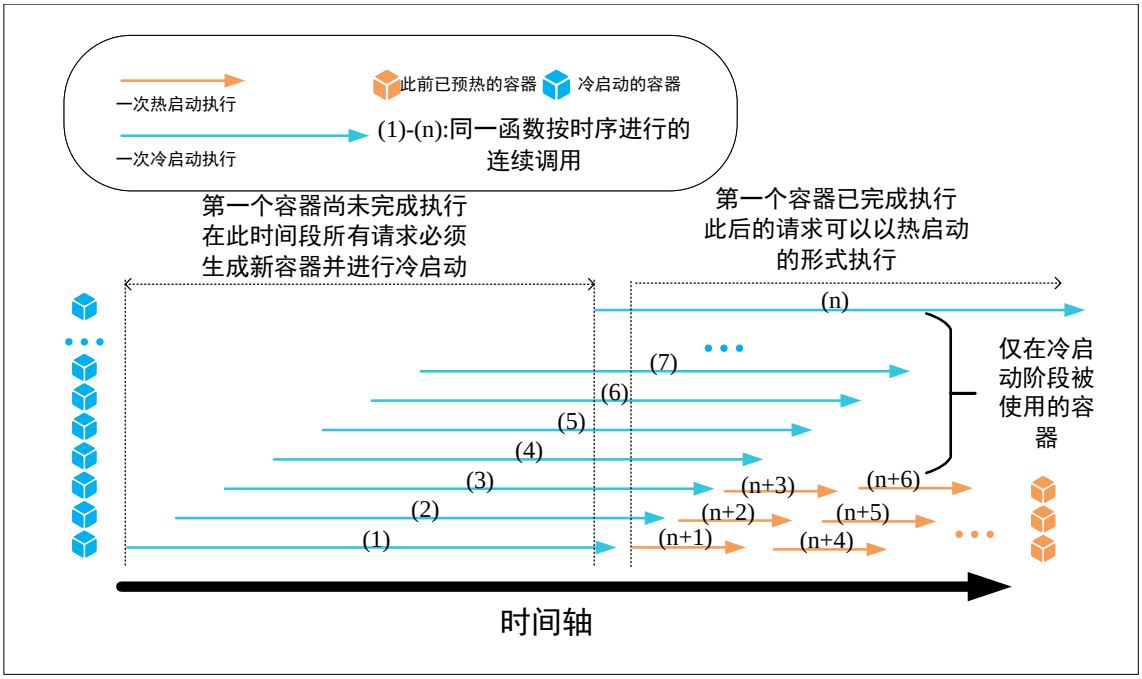


图 2.6　冷启动导致的容器数量暴增。

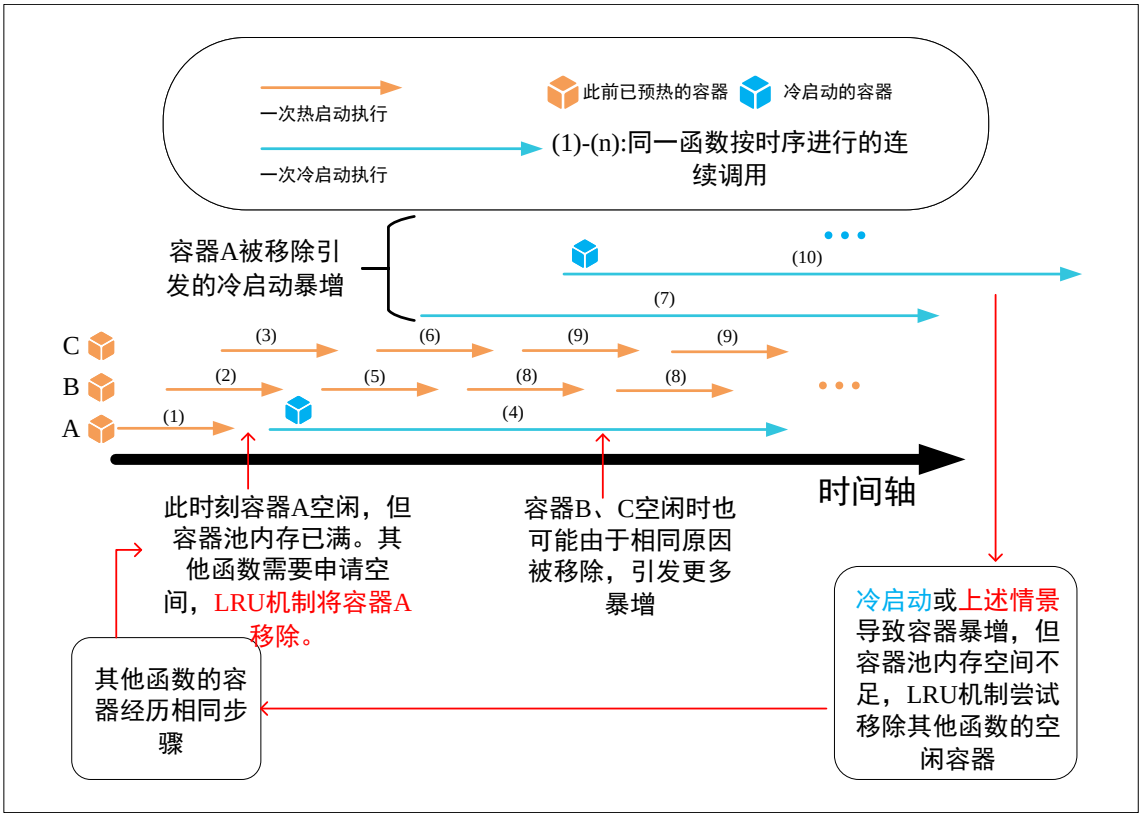


图 2.7　死循环：高调用频次函数间的内存空间争用导致不断发生冷启动，冷启动导致的容器数量暴增又不断导致空间争用。

内存争用的情景如图 2.7所示，多个高频函数共同存在于系统中，且我们限定容器池可用空间并非无限。由于函数的调用频次在一天内会有所波动。当一个高频函数调用随时间波动升高时，为了生成新的容器执行调用，必然要经历冷启动

暴增的阶段。但由于容器池内存不足，LRU[11] 机制会删除其他函数的容器来腾出空间。删除的容器一旦属于另一个高频请求函数，那么当被删除容器的高频函数新的调用到达时，又会重复进行上述的步骤。

由于频繁发生冷启动暴增的情景，容器池中往往挤满了属于高频请求函数的容器 (图 2.4)，导致被删除容器属于另一个高请求频次函数的几率很高。如此，**多个高频函数会不断导致彼此的容器被 LRU 机制删除，争用内存资源，结果又不断导致彼此的容器数量不足，进而引发彼此的冷启动暴增。由此陷入了一个死循环。**结果，所有高频函数都因为容器数量不足，有了极高的超时次数和冷启动次数，正如图 2.3b所示的统计结果。

如果容器池可用内存足够大，给冷启动时的容器暴增提供充足的空间，理论上高频函数之间的空间争用便不易出现。我们模拟并统计了容器池可用空间从 4G 增长到 48G 过程中总体的超时次数，结果如图 2.8所示，上述观点得到了佐证。

在有多个 Invoker 的集群系统中，理论上可以通过 LoadBalancer 将高频函数放在不同的容器池中执行来打破高频函数对空间的争用，进而阻止争用的出现。

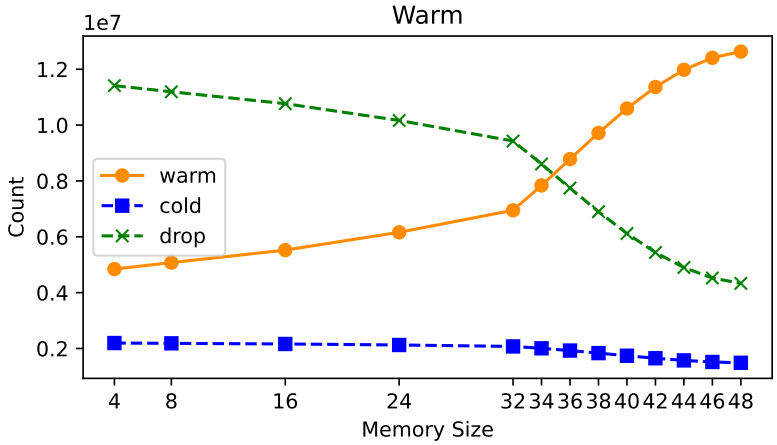
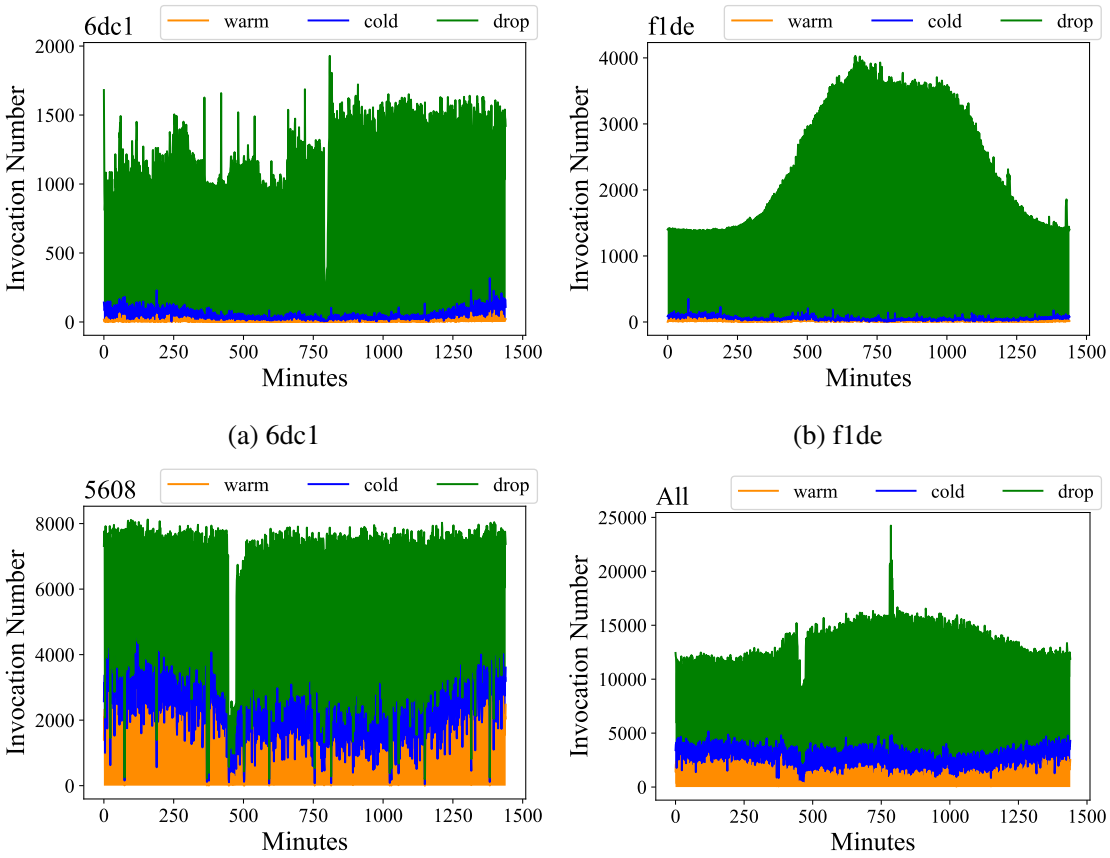


图 2.8　容器池可用内存从 4G 增长到 48G 过程中，数据集总体 warm，cold 与 drop 次数的变化

那么，在单个 Invoker，单容器池且容器池可用空间相对受限的情境下，面对高频函数争用空间资源的情景对 FaaS 系统进行优化改进，进而提高 FaaS 系统的吞吐效率和调用处理效率，以更少的内存资源处理更多的调用请求。就是本文研究解决的问题。受多 Invoker 系统的启发，我们想到的解决方案是：

1. 定位到调用频次高且占用空间资源较多的函数，并计算其所需的空间。
2. 在容器池中为这些函数独立分配空间资源，相互隔离以避免争用。



(a) 6dc1　(b) f1de

(c) 5608　(d) All

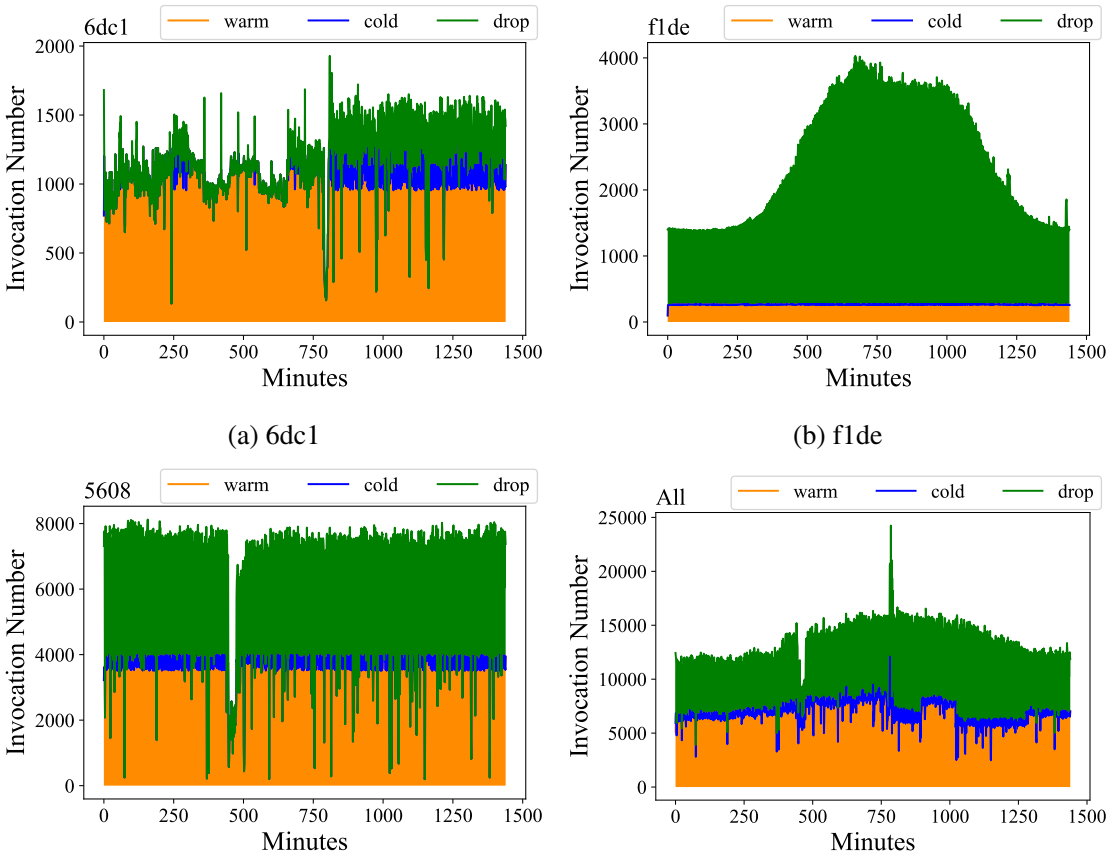
图 2.9　原生 LRU 策略时 3 个高请求频次函数与数据集总体在 1440 分钟内调用结果

为了初步验证我们的猜想，我们尝试简单验证**高请求函数容器空间彼此隔离**的效果。我们在模拟器中为这三个函数分别独立分配了其所需内存两倍的空间，并重新模拟了 FaaS 系统的调用。此时系统中有四块彼此隔离的空间，三块独立空间由三个高请求频次函数使用；一块主空间由其他函数使用，遵循 LRU 机制。调用模拟结果如图 2.10所示。

图 2.9展示的是在原生的 LRU 策略下三个高请求频次函数以及负载整体的调用结果统计。通过图 2.9与图 2.10对比，**可见：即使在进行简单的独立空间分配后，3 个请求高频次函数以及系统整体的吞吐效率都有了十分可观的提高**。

这成功地验证了我们的分析与设想，证明了可以通过独立空间分配策略提升系统地整体吞吐率。

但很显然，简单的分配两倍空间并不是最优的分配方式，高频高占用函数也不能靠人眼识别。我们需要对 FaaS 系统进行改进，为其增添高占用函数定位的能力以及智能化的空间分配能力。



(a) 6dc1　(b) f1de

(c) 5608　(d) All

图 2.10　高频函数独立分配两倍所需空间时，3 个高请求频次函数与数据集总体在 1440 分钟内调用结果

具体的设计方案详见下章。

3 系统设计

本章在前文的观察与讨论的基础上对要解决的问题进行逐个分析，并提出了各个问题的解决方案；之后简要展示了采用 FaaSMaT 优化后的系统结构总览；然后介绍了各个模块的详细设计方案。在最后一部分，本章简要介绍了 FaaS 系统模拟器的设计与实现。

3.1 需求分析

综合上文的观察与分析，想要提高 FaaS 系统的整体吞吐量，降低超时的次数，其关键在与打破高频函数之间对空间资源的争用。对此我们提供的思路是：

1. 寻找调用频次高且占用空间资源较多的函数，并计算其所需空间。
2. 在 Invoker 中为这些函数独立分配空间资源，以达到彼此隔离，避免争用的效果。

3.1.1 函数定位

首先我们需要能够找到请求频次高且资源占用多的函数。在实验环境中，我们能从数据集中获取函数的调用请求时间、执行时间、内存占用等各种数据；在实际生产环境中，各个 FaaS 平台都提供了函数调用监控与记录的能力。因此获取函数调用的各项信息并非难事。在获取各项数据后，即可依靠相关算法在函数集中定位到高频高占用函数。

3.1.2 空间大小计算策略

在定位到需要独立分配空间的函数后，下一个问题是应该给找到的函数分配多少独立空间。此问题类似于流量问题。例如，在不考虑冷启动的情况下，假设 10ms 时间段内有 5 次调用请求，每次请求的热启动执行时间也是 10ms，那么为了使 10ms 内所有调用请求都能被执行，我们需要单个容器大小为 5*10 / 10 = 5 倍空间，用于生成并存放 5 个容器；如果执行时间为 20ms，那么就需要 5*20 / 10 = 10 倍的空间；其余同理。考虑到我们能够获取函数一天内的总调用请求次数、函数的热启动时间、函数的平均内存占用，那么稍加计算就能得到所需分配的空间大小。

但是上述分配依照平均值做分配，是静态的分配方式。但如图 2.9所示，高频

函数每天每分钟的调用请求次数会随着时间变化上下波动。较为极端的情况下，高频函数可能在一天内大多数时间内并无调用，但在某一时间段内会有大量调用到达。因此为高频函数分配的内存空间不应是固定不变的，而应随着调用频次的宏观变化做出动态改变。数据集提供了函数每天每分钟的调用次数，依照这些数据，我们可以设定分配空间的大小每分钟动态改变。如此，既能为高频函数提供足够的空间，也能避免分配过多导致资源的浪费，从而达到最优的系统吞吐量优化和最高效率的内存空间使用。

Azure Functions Data Set 2019[18] 提供了完整的 12 天的函数调用记录，其中包含每分钟的调用次数，因此在实验时我们可以依靠这 12 天的数据进行时序预测。在实际生产环境中，获取某一函数数周或数月内每分钟的调用频次记录也并非难事。有了这些具有一定周期规律的数据，我们可以依靠时间序列预测技术预测来实现空间分配的动态调整。

综合上述分析，我们可以实现一个较为完善且智能的空间分配计算策略。

3.1.3 其他

为了保证非高频函数的处理效率，还需要对高频函数可分配的独立空间大小设置上限。否则在总可用空间相对较小的情境下，高频函数的独立空间如果占据了大多数资源，就会导致某些调用频率很低但空间占用较大的函数因为空间资源不足调用失败。

3.2 系统总览

综合上一节的分析，在通用 FaaS 系统的基础上，我们可以对 FaaS 系统进行改进优化，所有改进本文统称为文中我们的改进名为 FaaSMaT。进行 FaaSMaT 优化后的系统架构如图3.1所示。

本相比原先的 FaaS 系统 (图 2.1)，FaaSMaT 增添了两个模块与一处修改。其中：函数定位模块（Function Locator）负责从 Controller 中获取函数的各项信息，并由此定位到需要进行独立空间分配的函数。时序预测模块（Predict）通过往期数据预测接下来一天内这些函数每分钟的调用次数。之后依照上述数据进行计算，得到应分配的空间大小。预测的结果以及计算后的空间大小存放在数据库中。Invoker 模块的在修改后，单个 Invoker 不再只持有一个容器池，而是除主容器池外额外持有数个独立容器池。每个独立容器池用于存放某一高频高占用函数的容器。且这

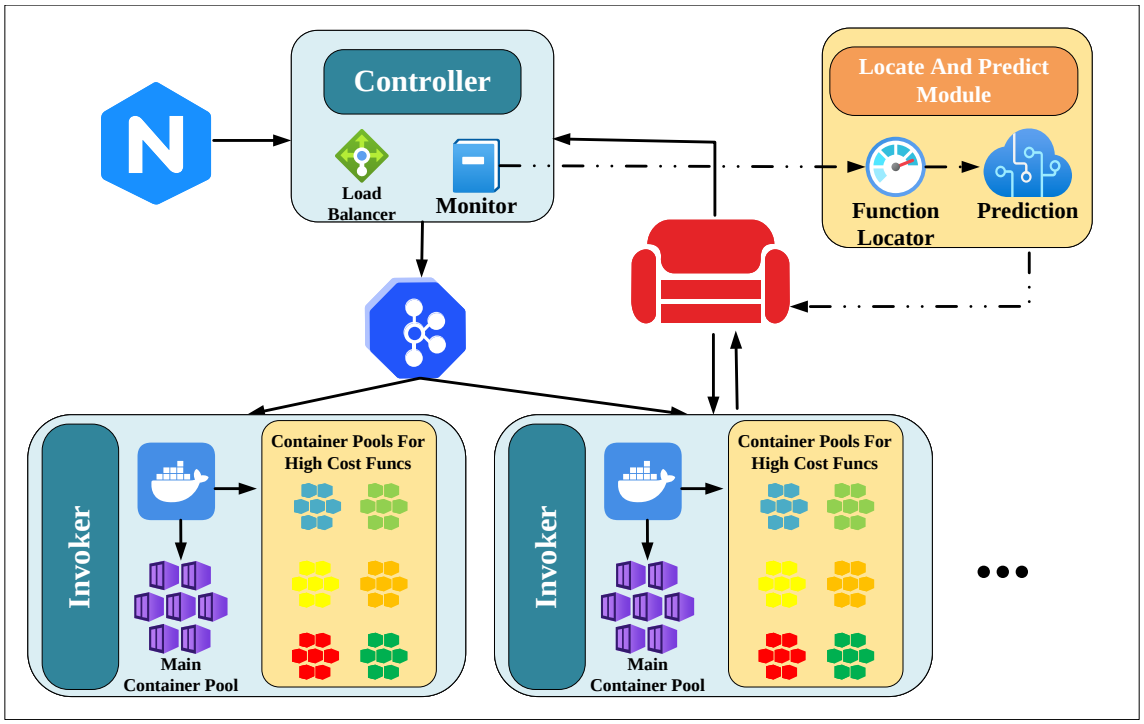


图 3.1　FaaSMaT 优化后的系统架构

些容器池的可用空间大小能够动态伸缩。

在 FaaSMaT 优化后的系统中，函数执行工作流程依旧如图 2.2所示，不过系统会执行一些额外的工作：

每天零点，系统会对此前 24 小时中收集到的数据进行汇总。系统依照这些数据定位到高频高占用函数，然后使用这些函数此前一周的调用记录预测其接下来 1440 分钟每分钟的调用次数，之后计算应分配的空间大小。预测与计算的结果被存储到数据库中。系统的容器池空间会进行分块，主空间负责处理非高频高占用函数的调用。其余独立的分块空间则属于各个高频的函数。每分钟初，这些空间的大小还会根据预测结果进行动态伸缩。属于高频高占用函数的调用请求到达后，会在独立空间寻找或生成容器，然后进行函数执行操作。

3.3　详细设计

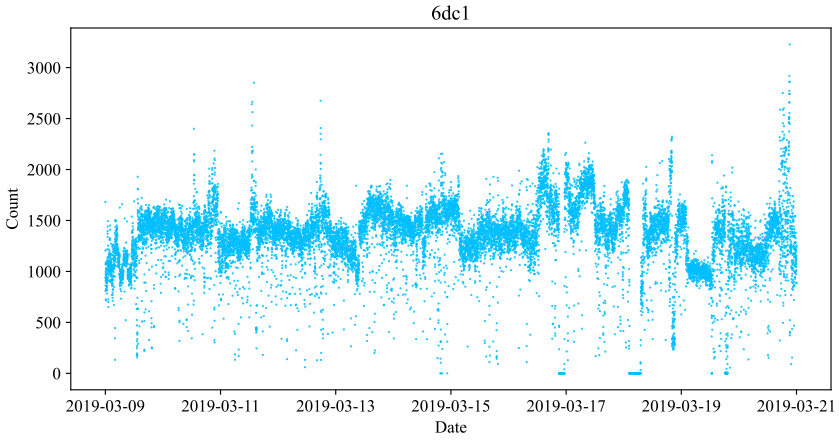
本节介绍了 FaaSMaT 优化策略中各个模块的具体设计，简要说明了具体的实现思路。

3.3.1　函数定位

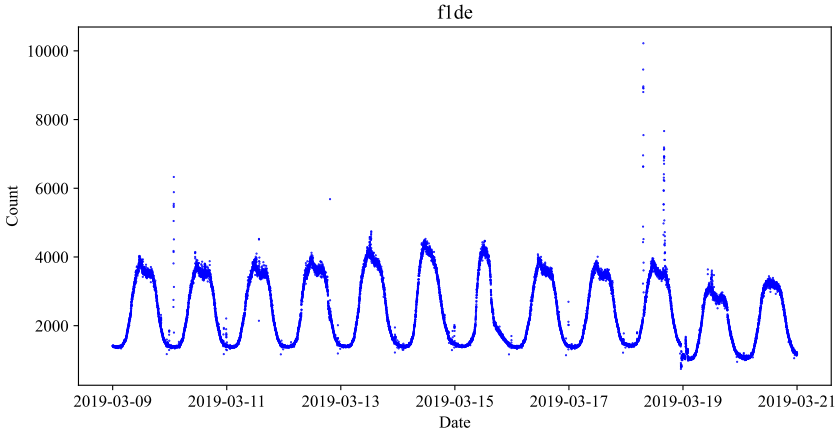
定位模块负责收集并存储函数的调用信息，并依据收集到的信息周期性的进行计算，得到应对分配空间的高频高占用函数。之后存储这些函数在本周期内的

各项数据，以供此后分配模块进行空间分配大小的计算。

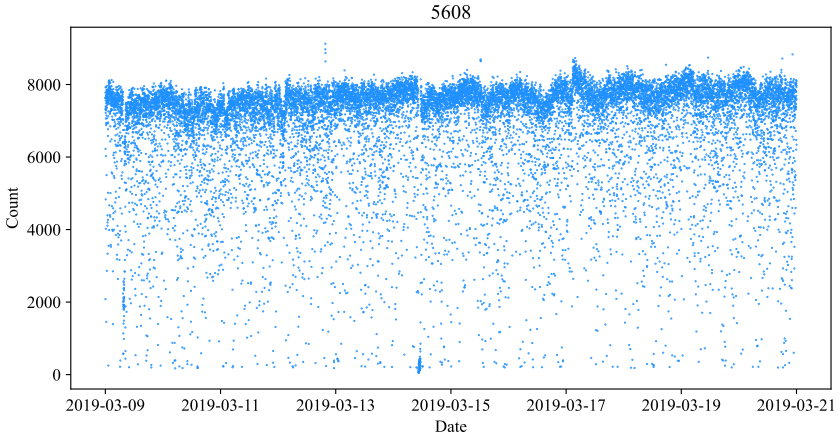
Azure Functions Data Set 2019 提供了完整的 12 天调用记录，我们对上文中出现的 3 个高请求频次函数十二天中的调用记录进行了统计，如图 3.2所示。可见调用频率较高的函数，其请求具有一定的每日周期性。因此我们以天为单位进行周期性监控统计，在每天 24:00 收集此周期的调用记录信息。



(a) 6dc1 12 天走势



(b) f1de 12 天走势



(c) 5608 12 天走势

图 3.2　3 个高频函数 12 天内每分钟的调用请求次数分布

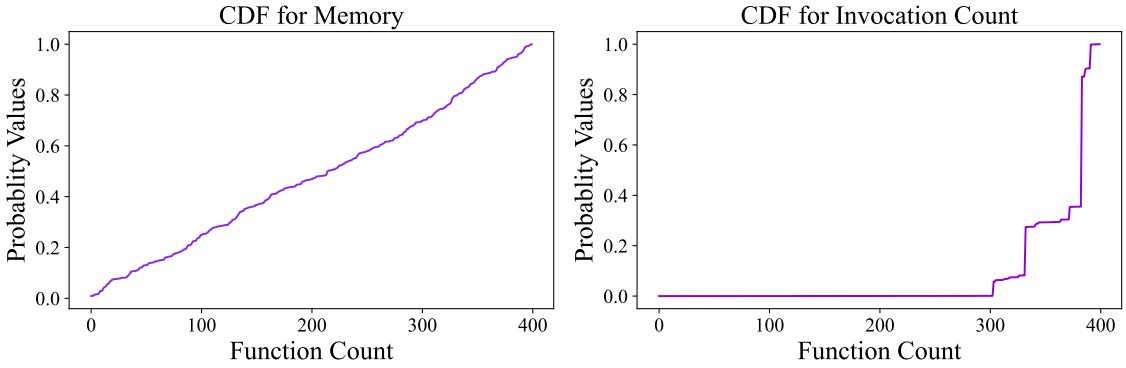


图 3.3　400 个函数内存占用以及一天内调用次数的 CDF 图

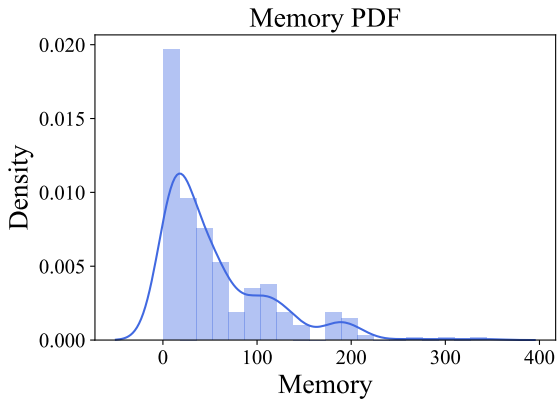


图 3.4　400 个函数内存占用的 PDF 图

图 3.3展示的是 400 个被采样函数中函数的平均内存占用以及一天内调用次数的 CDF 图。观察可知，400 个函数的内存占用分布相对均匀，但就调用频次而言，大部分函数的调用频次并不高，少数函数的调用频次极高。由此我们采取以下策略从函数中定位高频高占用函数：

1. 为每个函数进行得分计算。设函数的平均内存占用为 m（MB），每分钟平均调用次数为 c，得分为 score。

$$socre = m * c \tag{3.1}$$

当且仅当 score > 10000 时函数被视作候选函数。

2. 为候选函数进行并行度计算。设候选函数的平均热启动执行时间为 t（ms），其并行度为 p。

$$p = (c * t)/60000 \tag{3.2}$$

当且仅当 p > 1 时，候选函数被视作高频高占用函数。

上述策略同时考虑到了函数的内存占用、调用频次、执行时间三项特征。其中得分 score 筛选出了调用请求频次与内存占用两项特征相乘较高的函数。由图 3.4的

统计结果可知，函数的内存占用普遍处于 0-150 的区间。调用频次较高的函数往往有每分钟 1000 以上的调用请求次数。因此以 10,000 作为限制点可以筛选出调用频次在 1000 次级别且空间占用在 10MB 级别以上的函数。并行度计算则在候选函数中筛选出单个容器无法满足执行需求的函数。**并行度大于一，意味着此函数平均一分钟内需要多份容器并行存在才能完成执行需求**。他们往往是函数空间争用出现的主要推手。

基于热启动时间进行并行度计算，相当于无视了冷启动暴增时带来的容器需求，由此设定的空间上限避免了独立空间占用内存过多。虽然牺牲了冷启动时的调用请求处理率，但是保证了稳定的热容器并行。

经过以上步骤，高占用函数的定位需求便可完成。在 OpenWhisk 中，函数调用信息的获取可以依靠 OpenWhisk 自带的日志系统实现，获取函数的内存占用可以通过 Docker 监控完成。只需在 OpenWhisk 外部添加一个模块。在每天 24 时周期性的获取 OpenWhisk 的 Invocation 日志和 Docker 监控信息，并加以统计计算，然后将定位到的函数名（OpenWhisk 中以包名/函数名的形式存在）存储到数据库中即可。

3.3.2 空间分配与时序预测

空间大小的分配类似于流量计算，即上一节的并行度计算。设在时间段 T 内某函数执行了 n 次，每次热启动执行时间为 t。那么为了使 T 内所有调用请求都能被执行，应分配的容器数量 c 为：

$$c = (n * t)/T \tag{3.3}$$

多个容器并行处理的情景正是我们前文中指出的理想情境, 如图 2.5所示。当 T 为 60,000ms（60s）时，所需的容器数量 c 即为此分钟函数的并行度，用 c 乘以函数的平均空间占用 m 即为应为函数分配的独立空间大小。

$$memtoallocate = c * m \tag{3.4}$$

在使用静态分配策略时，我们可以统计一天 1440 分钟内函数调用的总次数，进而得到平均每分钟的调用次数，使用平均值计算出的应分配空间即为函数所需空间。静态的计算方式好处是无需太大的额外开销，只需在每天零点时进行计算与分配即可。

但是如图 3.2所示，高频高占用函数一天的调用请求是有一定波动的。使用均

值计算空间大小只会导致调用频次较低时函数所分配的空间过多，调用频次较高时函数分配的空间不足，这既浪费了空间又无法达到最优的吞吐率优化。因此应该对分配空间的大小进行动态伸缩。考虑到时间粒度与系统消耗，我们在每分钟初进行空间的伸缩。伸缩改变的大小采用时间序列预测技术获取。

时间序列预测技术只是本文采用的优化方法，而非研究内容，因此在此只做简单介绍。

我们选择采用 Facebook 的 Prophet[51] 进行时序预测的工作。Prophet 是一种用于时间序列预测的开源工具，它基于统计学方法和机器学习技术，可以帮助用户快速、准确地预测时间序列数据的趋势和季节性变化。

Prophet 的特点包括：

1. 简单易用：Prophet 使用简单的 API 和参数设置，可以快速进行时间序列预测，即使没有深入的统计学或机器学习知识，用户也可以轻松使用。
2. 稳健性强：Prophet 对缺失值、异常值和离群点等常见问题具有较强的鲁棒性，可以处理多种类型的时间序列数据。
3. 可解释性好：Prophet 提供了丰富的可视化和诊断工具，可以帮助用户更好地理解模型预测的结果，并优化模型的性能。

Prophet 的基本原理是将时间序列数据分解为趋势、季节性和节假日等多个部分，并使用时间序列分析和机器学习技术对每个部分进行建模和预测。具体来说，Prophet 使用以下三个主要组件：自回归模型（AR），季节性调整因子，节假日效应：Prophet 还可以自动检测和建模节假日和其他重要事件对时间序列的影响，以提高模型的预测准确性。

Prophet 可以用于多种时间序列预测应用，例如销售预测、股票价格预测、天气预测等。Prophet 的源代码托管在 GitHub 上，可以免费下载和使用。同时 Prophet 有着简单的 Python API，对数据进行整理后可以直接调用。

在具体实现时，我们可以对收集到的 OpenWhisk 日志信息以及 Docker 信息进行整理调整后，调用 Prophet 的 API 进行时序预测。预测信息存放在临时文件中或数据库中，以供每分钟内存空间伸缩时使用。

至于独立内存空间分配的实际实现，由于 OpenWhisk 底层采用 Docker[42]，因此独立空间分配相当于使用 Docker 分配独立容器池。实现途径是在 OpenWhisk 的 Controller 以及 Invoker 中对高频函数进行特殊标记处理。然后使用 Linux 内核的 cgroup 机制[52] 为 Docker 的不同容器可占用的内存空间进行分块，从而实现为属

于高频高占用函数的容器独立分配不同的内存资源的需求。具体细节在此不表。

需要注意的是，相比于静态分配策略一天只需一次的分配模式，动态分配策略每分钟都需要对 cgroup 进行内存分配和控制。相对来讲系统的吞吐率和内存利用效率有所优化，但是操作系统的额外开销有着一定增加。

3.3.3 其他细节

关于独立空间上限的设置，我们的设计中默认设置独立空间最多可占用内存池总内存空间的 70%。当要进行新的空间申请时，如果已达到申请上限，则不再申请直接放弃。

3.4 FaaS 系统模拟器设计细节

本节详细介绍前文 1.3小节中提到的 FaaS 模拟器的具体设计与实现。本文的实验测试基于此模拟器完成，阅读此节便于理解下一章节的实验过程，但不阅读此节也无大碍。读者可自由选择是否跳过。

FaaS 系统的基本架构与调用处理流程如前文图 2.1和图 2.2所示。模拟器的设计也遵循这种设计模式与请求处理流程：

1. 获得函数调用请求；
2. 将调用请求加入消息队列；
3. Invoker 从消息队列中取出消息进行处理；
4. 从数据库中获取请求对应的函数;
5. Invoker 在容器池中寻找或生成容器，执行函数；
6. 完成执行后返回调用结果；

模拟器无法真正地执行函数，因此执行过程中所需内存、时间等数据需要从云服务商提供的数据集中获取。Azure Data Set[18] 等数据集提供了函数的内存占用、执行时间、每分钟调用次数等数据。模拟器会将调用请求均匀分散到一分钟内，并使用数据集提供的执行时间和内存大小用于模拟。

模拟器并不模拟 FaaS 系统的 API 网关能力，而是通过读取文件模拟调用请求的到达。模拟器接受两项输入：函数的调用请求文件以及函数数据文件。调用请求文件以毫秒为粒度，按照时间顺序逐个记录函数调用请求。模拟器通过逐行读取此文件，模拟一天内函数请求的到达。函数数据文件记录了函数的各种信息，如内存占用、执行时间等。模拟器会读取此文件，将信息记录到内存中。执行请求时，

模拟器会根据请求的函数名中获取函数的相关信息。

真实的 FaaS 系统中往往会部署在集群中，集群会有多个 Invoker。FaaS 系统的 Controller 会根据 Invoker 的负载状况进行负载均衡。但本文的实验与模拟不考虑集群部署，因此模拟器中只有一个 Invoker，函数执行时不进行负载均衡。FaaS 的函数需要在容器中执行，模拟器也模拟了这一情景。容器的寻找、生成和删除策略与前文实际的 FaaS 系统保持一致 (2.1.3)，在此不多赘述。

模拟器的核心是一个毫秒循环 (图 3.5)，模拟 FaaS 系统一天 1440 × 60,000 毫秒的调用处理。每一毫秒，模拟器会读取调用请求文件，判断是否有此毫秒的调用。若存在，调用会进入消息队列。消息队列每毫秒从队首取出调用，超时的请求会被丢弃；未超时，则进行容器的寻找、生成、驱逐以及函数的执行等操作。模拟器每毫秒初还会检查各个容器，完成函数执行的容器会被置为空闲状态，空闲等待满 10 分钟的容器会被删除。

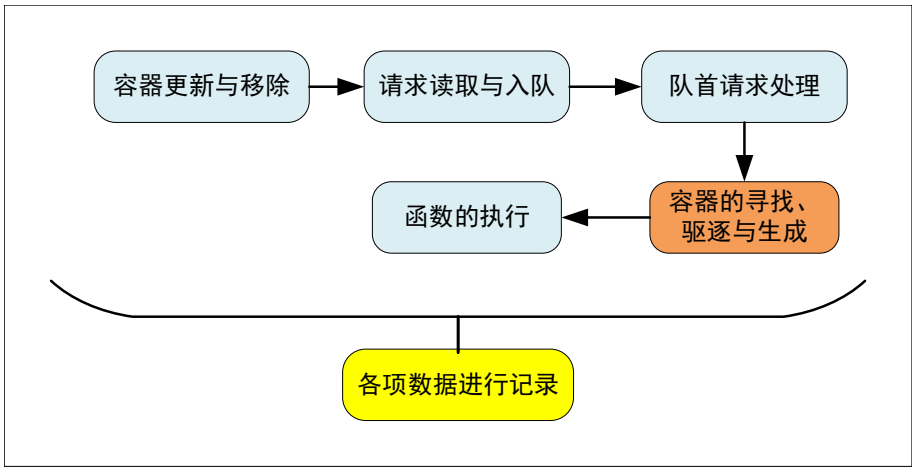


图 3.5　模拟器主循环示意图

模拟器没有真正的执行函数，因此模拟器并没有函数执行的输入输出。默认情境下，模拟器会在 1440 分钟的模拟完成后打印总体热启动、冷启动和超时的次数。可以通过对模拟器进行简单的修改实现各种数据的采样。可以自由的添加记录类，记录内存、调用结果、容器状态等诸多数据。我们在模拟器中实现了数个记录类，他们能够记录特定函数的调用结果、每分钟的调用数据、每分钟的内存占用等。记录结果被写入 csv 文件中，便于统计与绘图。读者可以阅读源码，增加或修改记录类，以获取自己想要的数据。

模拟器使用 Java 语言编写，能够快速修改与编译。除了原生的 LRU 机制外，我们在模拟器中也实现了本文的 FaaSMaT 机制，并进行了相关的实验与测试。读者可在 GitHub[38] 上找到模拟器的源代码。

4 实验与分析

本章基于前文的设计与实现，在模拟器上对独立空间分配策略的效果进行了验证与分析。首先介绍了我们所采用的函数负载与实验设置；之后展示并分析了实验的结果统计；最后总结了优化的效果。

4.1 实验目的与实验环境

4.1.1 实验目的

综合前文的设计与分析，我们在 FaaS 模拟器上集成了独立空间分配策略，以此进行实验。

本实验的目的是：在对高占用函数采取独立空间分配策略后，FaaS 系统处理函数负载，检测 FaaS 系统整体吞吐率的变化以及内存资源利用率的变化。在实验过程中对比多种空间分配策略以及数据集，量化以及可视化独立空间分配策略的效果。

4.1.2 实验环境

考虑到 FaaS 系统的高耦合与复杂性，为了快速观察到系统优化的效果，我们使用 Java 语言编写实现了 FaaS 系统的模拟器。它能够从预处理后的数据集中读取调用记录，模拟 FaaS 系统对调用请求处理的全过程，并记录下模拟全过程中的各项数据。能够满足简单修改、快速测试、benchmark 等实验需求。源码已在 Github[38] 上公开。

在使用模拟器进行实验时，我们的实验环境如下。

1. 操作系统：Ubuntu 18.04 LTS。
2. 数据集：Azure Function DataSet 2019
3. 实验：FaaS 系统模拟与分析
4. 硬件：32G 内存，16 cores CPU
5. 硬盘空间需求：10GB
6. 完成所有实验大致所需时间：6h

4.2 负载选择与数据预处理

4.2.1 数据集采样

我们的目标是测试独立空间分配针对系统整体吞吐率优化的影响，优化的前提是负载函数集中存在着高频高占用函数。为了观察整体的效果，我们在 Azure Function Trace 中采样了三份函数集。三份采样采取不同的采样策略，使其中高频函数的数量各不相同，这便于我们观察独立空间分配策略在不同负载下的表现，更好的检验此优化方式的效果，我们使用了如下三份函数集作为负载:（细节详见表 4.1）

1. **INFREQUENT**: 此函数集包含了 1000 个调用极不频繁的函数，他们普遍在一天内调用次数不超过 10 次且分布较为分散。因此，它们往往以冷启动的方式被执行，容器往往因为满足 10min 的 keepAlive 时间被自动销毁。FaaS 系统处理此负载时，几乎不可能遇到 LRU 驱逐容器的情形。
2. **REPRESENTATIVE**: 此函数集与我们前文进行分析的函数集一致。我们将 Azure Functions Data Set 2019 第一天的 36000+ 个函数按照调用频次从小到大排序，将其等分为四份，每份进行采样，取出 100 个函数，共取得 400 个函数。此函数集具有一定的代表性，其中既有低频，高执行时间的函数；也有高频但执行相对较快的函数。
3. **FREQUENT**: 此函数集中包含了调用请求频次相对而言十分频繁的函数。我们按照 REPRESENTATIVE 中的方式将第一天的 36000+ 个函数 100 等分。此次在最后一份即调用频次最高的一份中随机采样了 10 个函数。这样就有更大的概率使我们能采样到更多的高频函数。当他们的容器同时存在于同一容器池时，不同高频函数之间的空间资源争用现象会更加明显，更方便我们观察优化的效果。

我们的分析主要基于 representative 负载，另外两个负载只进行大概的阐述。

值得一提的是，由于函数的采样具有随机性，represennative 负载中包含了一个调用频次极高的函数（哈希为 5608 见图 2.3b），所以 representative 负载的总调用次数比 frequent 负载还要高。不过这并不影响我们观测优化的效果。

4.2.2 数据集预处理

原始的 Azure Functions DataSet 2019 的格式[18] 需要进行一些额外的预处理才能生成工作负载。完整数据集包含完整的 12 天的函数调用记录。其中有数十亿个

表 4.1　选用的三个数据集概览

数据集	函数个数	总调用次数	调用频率	平均调用间隔
representative	400	18,452,673	214 次/sec	4.67ms
rare	1000	5911	4 次/min	15000ms
frequent	10	14276398	165 次/sec	6.01ms

单独的调用请求。我们使用第一天的数据作为主要数据，其余 11 天的数据只用于时间序列预测。我们并不考虑从未重复使用的函数（即，一天内少于两次调用的函数）。原始数据集提供了应用程序级别（Application）的内存消耗，应用程序由多个函数组成。因此，我们将内存占用均匀分配给应用程序中的所有函数。数据集提供了每天每分钟为单位的调用次数。参照常见的做法[11]，在生成工作负载时，如果一分钟中只有一次调用，则视为此调用在此分钟初进行。对于一分钟多次调用的函数，它们的调用在整个分钟中等间隔分布。

每个函数的冷启动时间估计为最大执行时间，并使用数据集提供的执行时间进行计算。数据集未考虑某些重要的冷启动开销来源，例如执行环境的创建（例如，Docker）。这不幸地低估了冷启动开销。然而，由于它适用于所有函数，因此它保留了不同保持活动策略的相对性能。函数的热启动时间以其平均执行时间替代。

对于每份函数负载，他们被预处理后会生成以下两份文件：

1. functions.csv: 被采样函数的数据总文件，详细地记录了被采样函数地各项数据。如内存占用、执行时间、每分钟的调用次数列表等，用于提供宏观的函数数据记录。
2. invokes.csv: 第一天内所有函数的调用请求表。按照时间顺序记录了从 0 到 1440×60000 毫秒每一毫秒到达的函数调用请求（因此文件较大）。实验时会逐个读取这些记录，模拟函数调用请求的依次到达。

此外，除第一天的数据被用于生成以上文件外，其余十一天的数据也有用处。在定位到高频函数后，Facebook Prophet 会使用前 12 天的数据作为输入进行时间序列预测，预测高频函数第 13 天中每分钟函数调用到达的次数。如果 12 天中某个函数的数据存在缺失，那么默认以第一天的数据作为此日的数据。在测试环节，我们会假定第一天的数据为第 13 天的真实数据，以此判断和检验时序预测的效果。

在实验过程中，我们默认 CPU 的处理能力很强，能够顺利的处理函数调用并执行相应的操作，Invoker 容器池可用总空间设为 16GB。因此模拟过程中所有的超时（drop）都是因为内存空间不足造成的。

在实验过程中我们采取了三种空间分配策略。

1. LRU：OpenWhisk 原生的容器驱逐策略。以下简称原生策略。
2. SSMP：static separated memory pool 静态独立容器池策略。采取此策略时，高频高占用函数的容器会被分配到其独属的空间中。但空间大小是按照每分钟平均调用请求次数计算，不会动态伸缩。下文简称静态策略。
3. SMP：dynamic separated memory pool 动态独立容器池策略。相比 SSMP，此策略中独立空间的大小会依照时序预测数据进行动态伸缩。下文简称之为动态策略。

下文中，我们将采用 SSMP 和 DSMP 进行优化简称为**采取优化措施**。考虑我们关于**系统吞吐率** [2.1] 的定义，针对单个函数，我们以**处理率**代指函数的（热启动次数 + 冷启动次数）/总调用次数。

函数处理率 = （函数热启动次数 + 函数冷启动次数）/函数总调用次数　(4.1)

4.3 实验结果

4.3.1 representative 负载

我们首先对 representative 负载采用以上三种策略进行了实验，采取三种策略时系统整体每分钟的热启动，冷启动和超时次数对比如图 4.1所示。可以观察到，在简单采取静态策略后，负载的超时次数就有了显著的下降。系统对于此负载的吞吐率有了明显的提升。在采取动态预测对空间大小进行动态伸缩后，系统对于负载的吞吐率也有了一些提升，不过提升并不大。

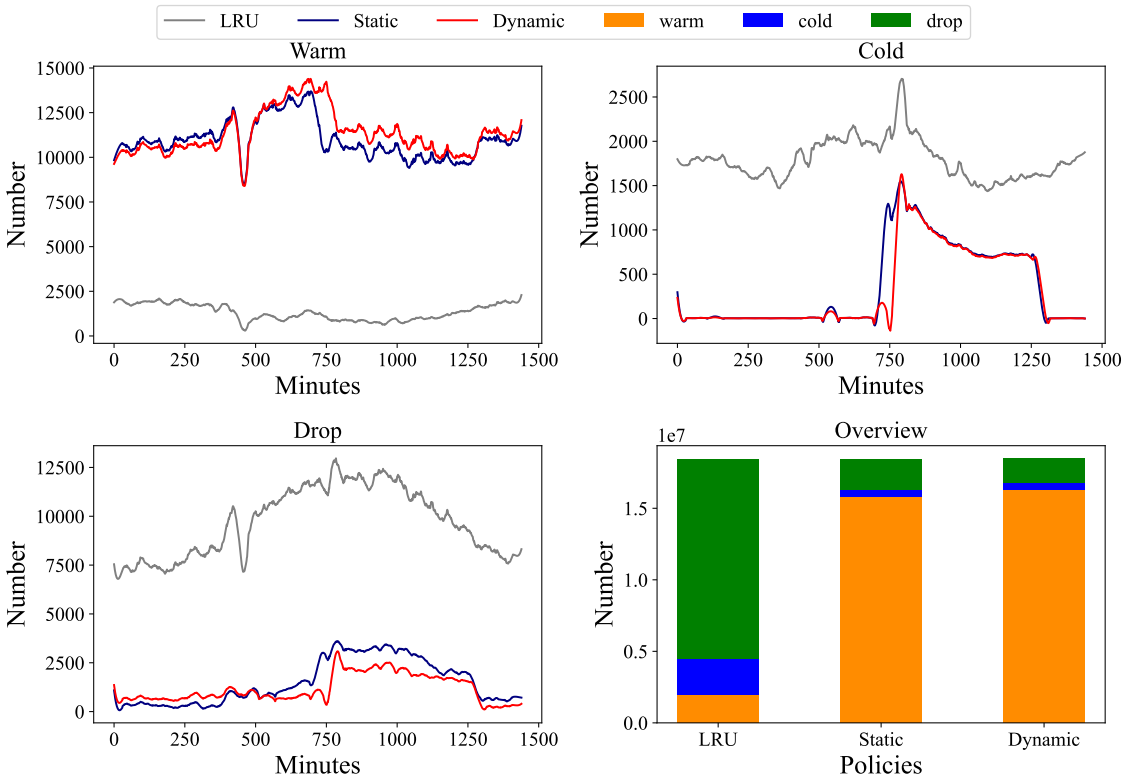


图 4.1　representative 负载在采取三种策略时每分钟的调用结果记录。以及三种策略结果对比。

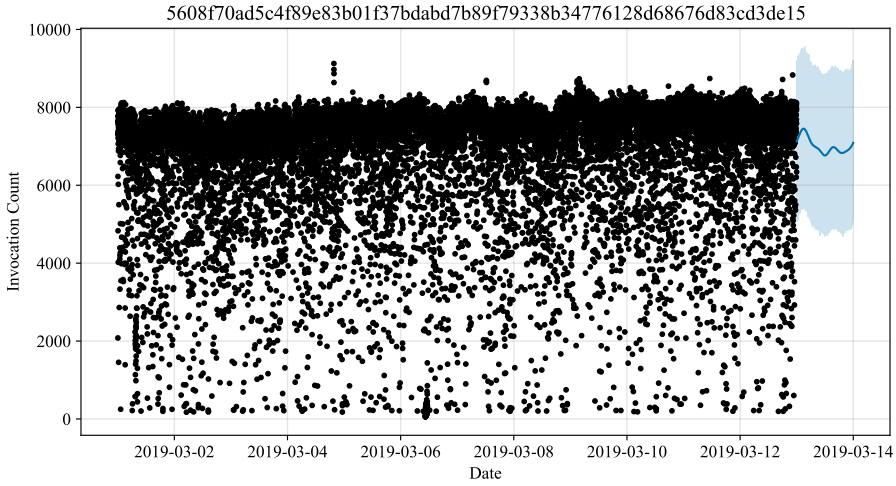


图 4.2　函数 5608 时序预测结果

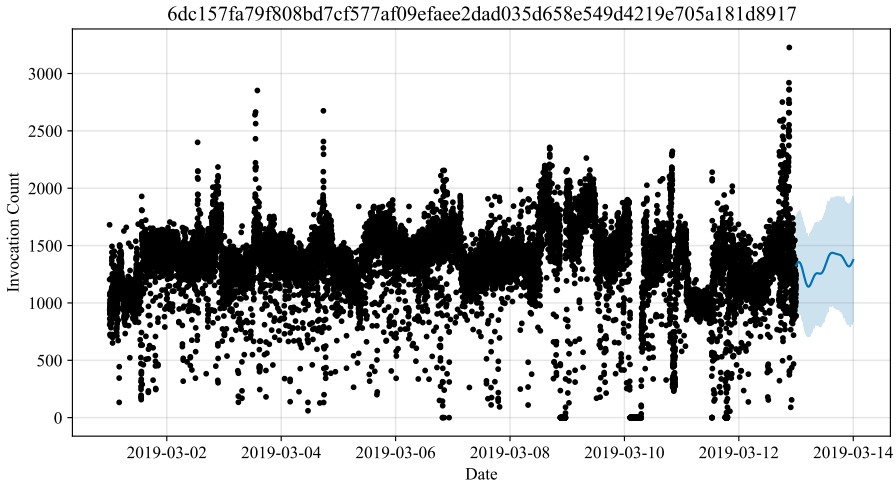


图 4.3　函数 6dc1 时序预测结果

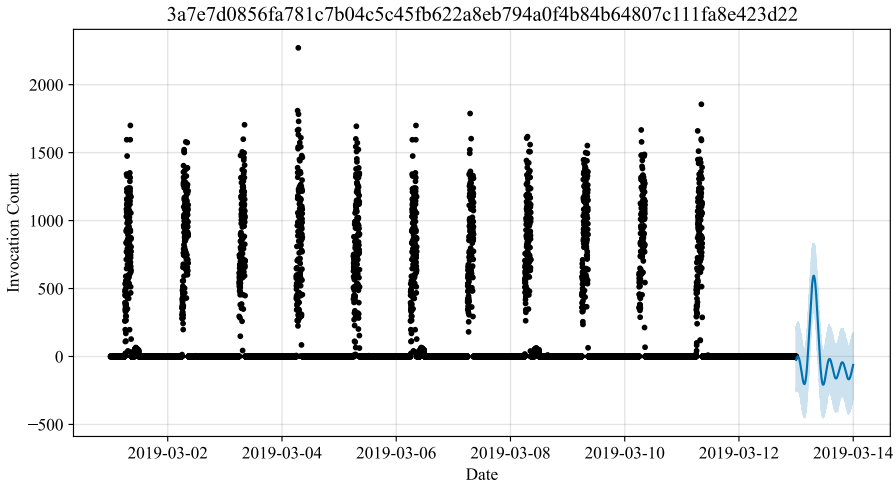


图 4.4　函数 3a7e 时序预测结果

在 representative 数据集中，高占用函数定位算法共定位到了五个高频高占用函数。图 4.2-图 4.6是在采取动态策略时五个被定位到的函数的时序预测结果。对比预测的数据和 12 天的准确数据可知，时序预测相对准确的预知了第 13 日每分钟调用次数的大致走向，但对于突发的流量洪峰无能为力（图 4.3， 4.5）。不过其对于一天内存在调用洪峰的函数也能做到预测 (图4.4)。

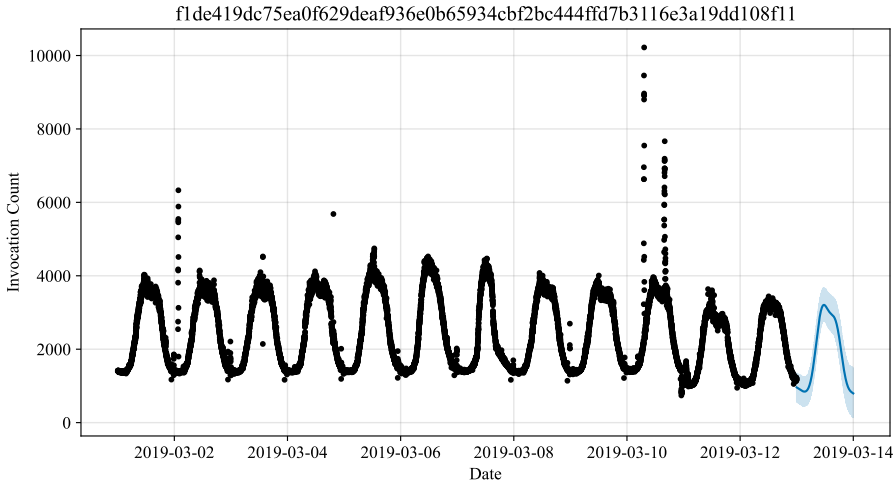


图 4.5　函数 f1de 时序预测结果

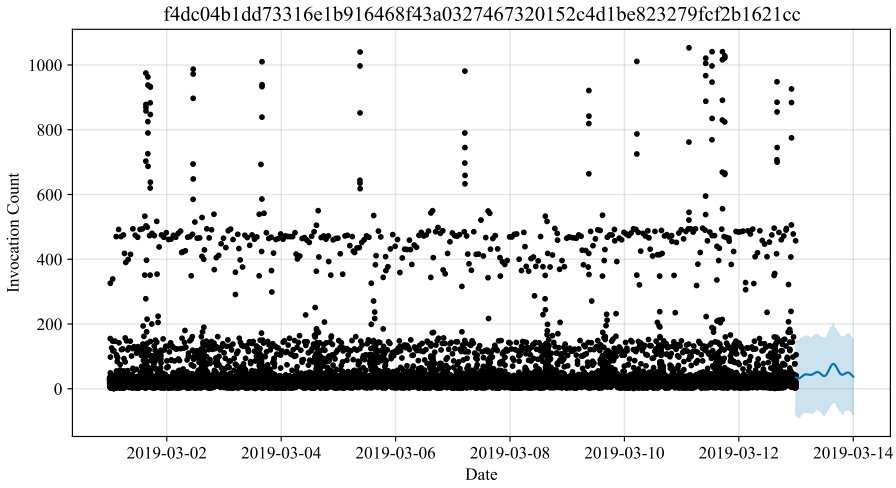


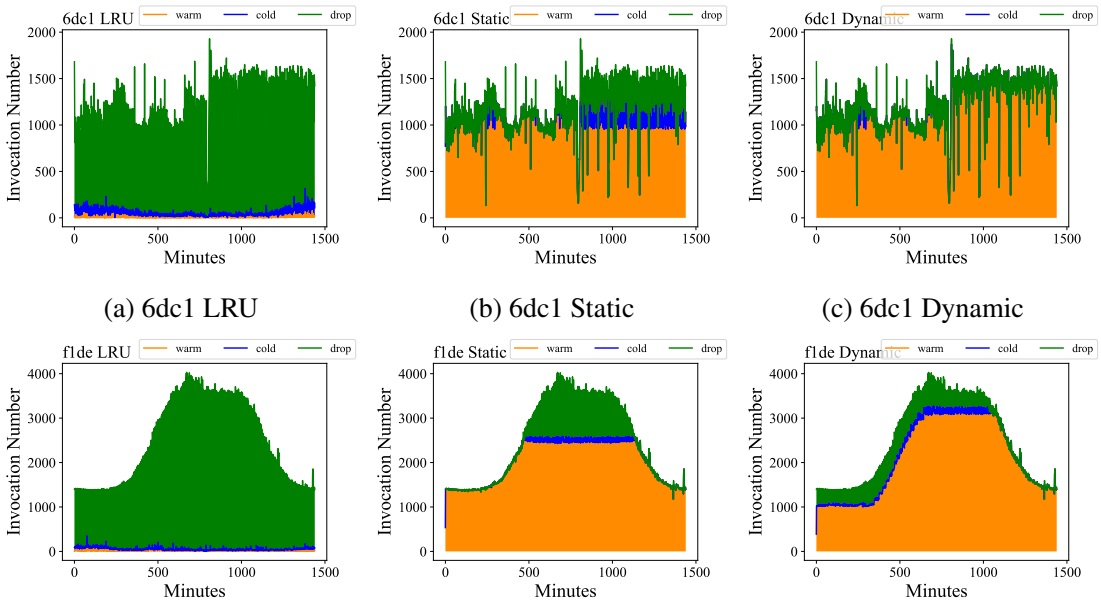
图 4.6　函数 f4dc 时序预测结果

对于这五个函数，他们在采取三种策略时总体的热启动、冷启动和超时次数详细数据如表 4.2所示。能够明显看到采取优化策略后这些高频函数的被处理率有了显著提升。但采取动态策略相比静态策略的提升并不大。

表 4.2　五个函数在三种策略下调用结果总和统计

策略	Warm	Cold	Drop
LRU	1,883,333	1,483,186	11,647,054
SSMP	13,787,860	157	1,225,556
DSMP	14,204,537	147	808,889

接下来我们深入观察动态策略的效果，总体看来，动态策略相比静态策略的提升并很小（图 4.1）。但我们以两个具有代表性的函数做对比，函数 6dc1 一天内波动幅度并不大（图 4.3），而函数 f1de 具有非常明显的波动周期（图 4.5）。图 4.7绘



(a) 6dc1 LRU (b) 6dc1 Static (c) 6dc1 Dynamic

(d) f1de LRU (e) f1de Static (f) f1de Dynamic

图 4.7　函数 f6dc1 和函数 f1de 在 0-1440 分钟，采取三种策略时调用结果走势

制了在采取 LRU、静态策略和动态策略时这两个函数 1440 分钟内的整体的 warm、cold 和 drop 次数走势图。可见，由于函数 6dc1 的波动性不大，因此动态策略相比静态策略的提升并不明显。但由于函数 f1de 的波动性，对其而言采用动态策略相比静态策略能有更好的处理率。在调用频次处于波峰时，动态策略能够增加独立空间地大小，提升函数的处理率；在调用频次处于波谷时，动态策略可以通过收缩独立空间的大小，有效减少空间的浪费。

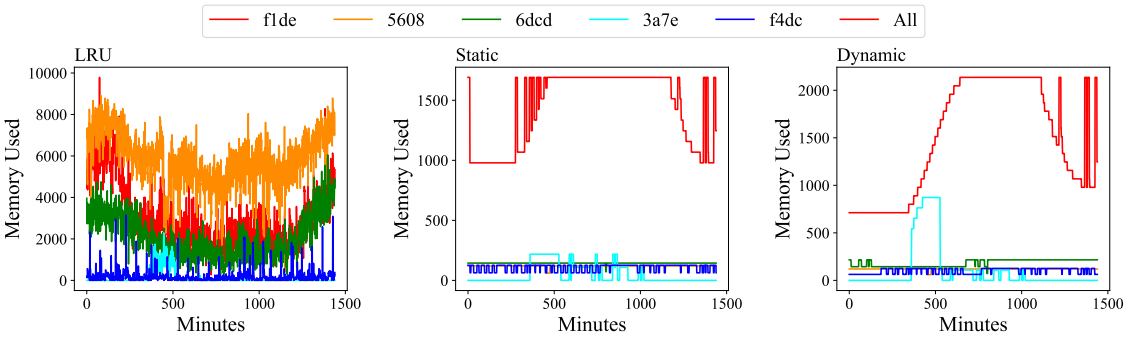
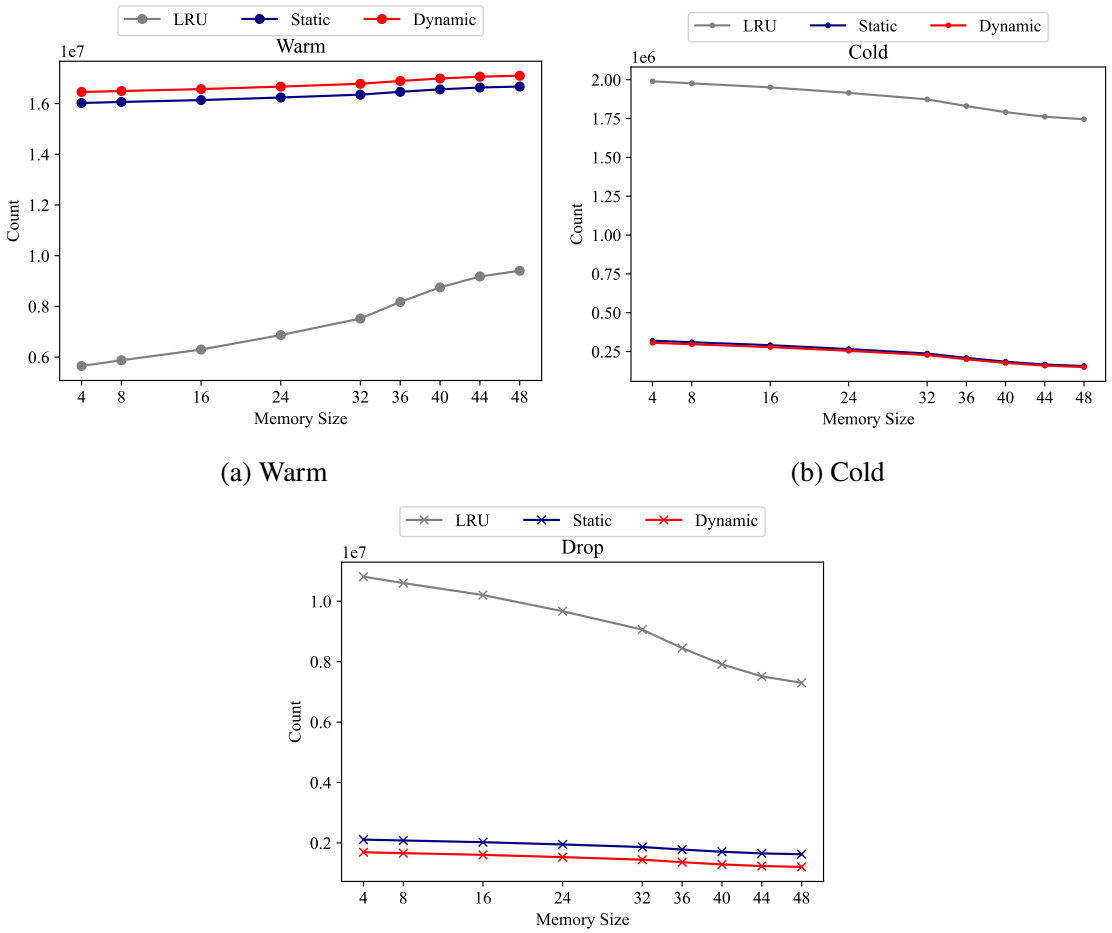


图 4.8　五个高占用函数在三种策略下内存占用走势

接下来我们观察内存占用的情况。前文我们已经观察到，在不采用独立空间策略时，高占用函数因为冷启动暴增会占用大量内存，且由于彼此之间的争用，占用空间的大小会有剧烈的波动（图 2.4）。图 4.8统计了在采用三个策略时，这五个高频函数的容器每分钟的最大内存占用。可以清晰的看到，在采取静态和动态策略时，由于设置的独立的存在空间上限，抑制了容器暴增的出现。所以他们的容器

占用的内存并不大，相比未采取优化策略时有着数量级的差距。但是由于相互隔离避免了前文中中提到的空间资源争用情形，没有了 LRU 时剧烈波动的情景，这些为数不多的容器能够持久的存在于独立空间中处理调用请求。所以系统以更少的空间消耗反而处理了更多的调用请求。

图 4.9展示了我们将容器池可用空间从 4G 增长到 48G 的过程中 representative 负载的调用结果。可见采取优化措施后，系统使用 4G 容器池空间时就有了远胜于 48G 空间的吞吐量。



(a) Warm

(b) Cold

(c) Drop

图 4.9　内存从 4G 增长到 48G 时三种策略调用结果统计

4.3.2　rare 与 frequent 负载

表格 4.3展示了在 16G 空间大小时 rare 数据集 frequent 数据集的调用结果统计。由于 rare 数据集中的函数都是调用次数极低的函数，因此 rare 数据集的实验结果中并未出现一次超时。也不存在可被定位到的高占用函数，因此在采取三种策略时并无差异。

表 4.3　rare 和 frequent 负载调用结果统计

负载	策略	Warm	Cold	Drop
rare	LRU,Static,Dynamic	2482	3429	0
frequent	LRU	331,710	1,565,840	12,382,371
frequent	Static	10,058,162	390,640	3,827,596
frequent	Dynamic	10,453,296	335,534	3,487,568

frequent 负载中只有 10 个函数，但定位到了 8 个高频函数，因此在采取优化措施后 frequent 数据集的效果有着明显的提升。不过有意思的一点是，由于函数的采样具有随机性，representative 数据集中函数的调用频次极高，结果是 representative 数据集的总调用次数相比 frequent 数据集还要高。而且由于 frequent 数据集中 8 个定位到的函数只有一个函数具有较强的周期性，所以相比静态策略，在采用动态策略后 frequenct 数据集的优化很不明显（只增加了不到 40 万次热启动）。

frequent 负载中各个定位到的函数的具体数据，以及系统整体的内存变化数据与 representative 数据集的模式大致相同，因此不再做展示与分析。

4.4　实验结论

综合上文的实验结果，我们可以得出结论：对于存在高占用函数的单个 FaaS invoker，在采取独立空间分配策略后，系统能够以原先 1/10 数量级的内存消耗处理 10 倍数量级的调用请求。系统整体的吞吐率、高占用函数的处理率都会有着显著提升。对于存在明显周期性的高占用函数，采取动态策略后函数的处理率会有更大的提升。

5 讨论与总结

本章讨论了在采取独立内存空间分配优化后可以进一步采取的优化方案，简要提及了面对多 Invoker 集群时独立空间分配策略存在的不足以及可采取的改进措施，最后对全文进行了总结。

5.1 未来展望

在采取独立空间分配策略后，高频高占用函数之间相互隔离，基于流量的大小的计算策略限制了单个高频函数可占用的内存空间上限。如此，通过限制冷启动暴增以及避免高频函数之间的空间争用实现了系统吞吐量优化，以更小的空间处理了更多的请求。但是如果单个独立空间中调用请求的频次随时间变化而增加，独立空间中依然会发生冷启动，(图 4.7 f1de Dynamic 子图)。冷启动的发生对系统本身以及独立空间中的函数请求来讲都不友好。

关于这个问题，我们认为可以在独立空间中引入容器预热机制。在独立空间完成分配后或者空间大小随时间时序预测增加后及时在空间中对容器进行预热，为接下来到达的请求增加做准备，尽可能减少冷启动发生的频次。与此同时，也可以移除独立空间中容器的 10min 移除机制，只要空间充足，那么独立空间中的容器便可以永久存在。当需要根据时序预测结果收缩空间大小时，再按需清理空闲的容器。采取以上两项优化，能够最大限度的减少独立空间中的冷启动频次，进一步提升系统的吞吐率。

5.2 集群优化

我们的研究与实验都是基于单个 Invoker，且 Invoker 的可用内存空间受限的前提。研究以及优化的目的是使用更少的资源处理更多的请求。但在实际生产环境中，云服务商往往使用集群[9, 44] 部署 FaaS 系统。服务器集群具有庞大的数量以及充足的硬件资源，这也是理论上 FaaS 集群能够提供“无限”的 CPU 内存等硬件资源的原因。因此即便集群中出现了大量的冷启动暴增，系统也有足够的内存空间用于存放容器，进而避免不同函数之间的空间资源争用。

前文中图 4.9也显示了这一特点。在可用内存大小增长的过程中，静态策略和动态策略限制了高占用函数可用的空间大小。因此他们在 4G 大小获得了很高的吞

吐率的同时，整体的效果并未随着总可用空间大小的变化有太多改进。但在 LRU 策略下，负载的热启动次数和超时次数却能随着内存的增加有着正向的优化。设想之，如果可用内存空间继续增大，原始的 LRU 机制下负载的超时次数终究会降低为 0。这也是 FaaS 系统“无限计算资源”的体现。

与之相对，采取两种策略后，高频函数可利用的空间存在上限。那么无论主容器池可用的空间有多大，都与他们无关。关于这个问题，我们提出的改进措施是：**空间借用**。

> 当主容器池可用空间充裕时，独立空间若发生空间不足的情形，可以在主空间中额外划分新空间资源，作为独立空间的一部分。借用的空间按照独立空间对待，使用后再将此空间归还给主空间。

这样，就能在维持空间独立性的基础上解决流量洪峰带来的独立空间大小不足的问题。

本文的研究目的是使用更少的资源处理更多的请求，提升 FaaS 系统的吞吐率，实验部分已经验证了优化策略的效果。讨论部分的内容不再做实验与对比。

5.3 总结

函数即服务（Function as a Service，FaaS）在当下是一种十分受欢迎的云计算服务模式。它提供了一种基于事件驱动的云计算模式，开发者可以以函数为基本单位进行开发、部署和运行，无需在意繁杂的服务器管理与维护等工作。云服务商可以提供理论上无限的计算资源。相较于传统的基于虚拟机的云计算模式，FaaS 可以更好地满足云计算场景下快速、低成本的计算需求。

然而，FaaS 系统存在着一些值得优化的方向，包括但不限于冷启动加速、资源调度优化等，学术界以及工业界都有了许多这些方面的研究。我们注意到，在 FaaS 系统中，请求频次较高进而有较高占用的函数会对系统的吞吐量产生不良影响，他们会大量占用 FaaS 服务器的空间资源，从而影响系统的性能表现。因此，如何对此类情景提高 FaaS 系统的吞吐量是一个值得研究的问题。

针对以上情景，我们提出了 FaaSMaT，一种基于调用流量与时序预测对函数进行独立空间分配的优化策略。该优化方法通过定位到高占用的函数，对其进行流量计算以及时间序列预测，并根据计算以及预测的结果动态进行容器池空间分配，从而提高系统的吞吐量。具体来说，FaaSMaT 通过对 FaaS 系统中的历史记录进行分析，定位到请求频次较高进而具有较高占用的函数。利用用其请求流量以

及历史数据，FaaSMaT 计算这些函数所需的容器池内存大小。然后为他们分配独立的容器池空间，将高占用的函数与其他函数隔离，高占用函数之间也彼此隔离。如此，FaaSMaT 打破高负载环境下高请求频次函数对内存空间的争用，避免了因为内存空间争用导致的容器驱逐以及冷启动。以此实现了 FaaS 系统整体吞吐率的优化。

在实验中，我们将 FaasMaT 优化策略应用于真实的 FaaS 函数负载中，并与原生策略进行对比。结果显示，我们的优化方法可以显著提高 FaaS 系统的吞吐率。在存在高请求高占用函数的负载中，单个 FaaS Invoker 最优时能使用原先十分之一级别的内存占用处理十倍级别的调用请求。

参考文献

[1] iron.io[EB/OL]. https://www.iron.io/.

[2] Aws Lambda[EB/OL]. https://aws.amazon.com/cn/lambda/.

[3] Alibaba Cloud Function Compute[EB/OL]. https://www.alibabacloud.com/zh/product/function-compute.

[4] Tencent Cloud Function Compute[EB/OL]. https://cloud.tencent.com/product/scf.

[5] FOX A, GRIFFITH R, JOSEPH A, et al. Above the clouds: A berkeley view of cloud computing[J]. Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS, 2009, 28(13): 2009.

[6] JONAS E, SCHLEIER-SMITH J, SREEKANTI V, et al. Cloud programming simplified: A berkeley view on serverless computing[J]. arXiv preprint arXiv:1902.03383, 2019.

[7] MANNER J, ENDRESS M, HECKEL T, et al. Cold start influencing factors in function as a service[C]. 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). IEEE, 2018: 181-188.

[8] SHAHRAD M, FONSECA R, GOIRI I, et al. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider[J]. arXiv preprint arXiv:2003.03423, 2020.

[9] Apache Openwhisk[EB/OL]. https://openwhisk.apache.org/.

[10] WU C P. Container management for serverless edge computing offerings[D]. 2019.

[11] FUERST A, SHARMA P. Faascache: keeping serverless computing alive with greedy-dual caching[C]. Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021: 386-400.

[12] BALDINI I, CASTRO P, CHANG K, et al. Serverless computing: Current trends and open problems[J]. Research advances in cloud computing, 2017: 1-20.

[13] MCGRATH G, BRENNER P R. Serverless computing: Design, implementation, and performance[C]. 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2017: 405-410.

[14] HENDRICKSON S, STURDEVANT S, HARTER T, et al. Serverless computation with openlambda[C]. 8th {USENIX} workshop on hot topics in cloud computing (HotCloud 16). 2016.

[15] COTE R J, ROSEN P, LESSER M, et al. Prediction of early relapse in patients with operable breast cancer by detection of occult bone marrow micrometastases. [J]. Journal of Clinical Oncology, 1991, 9(10): 1749-1756.

[16] LIN P M, GLIKSON A. Mitigating cold starts in serverless platforms: A pool-based approach[J]. arXiv preprint arXiv:1903.12221, 2019.

[17] BERMBACH D, KARAKAYA A S, BUCHHOLZ S. Using application knowledge to reduce cold starts in faas services[C]. Proceedings of the 35th annual ACM symposium on applied computing. 2020: 134-143.

[18] Azure functions data set 2019[EB/OL]. https://github.com/Azure/AzurePublicDataset.

[19] WANG A, CHANG S, TIAN H, et al. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute[C]. 2021 USENIX Annual Technical Conference (USENIX ATC 21). 2021.

[20] DU D, YU T, XIA Y, et al. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting[C]. Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020: 467-481.

[21] JEGANNATHAN A P, SAHA R, ADDYA S K. A time series forecasting approach to minimize cold start time in cloud-serverless platform[C]. 2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom). IEEE, 2022: 325-330.

[22] VAN EYK E, IOSUP A, ABAD C L, et al. A spec rg cloud group's vision on the performance challenges of faas cloud architectures[C]. Companion of the 2018 acm/spec international conference on performance engineering. 2018: 21-24.

[23] OAKES E, YANG L, HOUCK K, et al. Pipsqueak: Lean lambdas with large libraries [C]. 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2017: 395-400.

[24] HOCHREITER S, SCHMIDHUBER J. Long short-term memory[J]. Neural com-

putation, 1997, 9(8): 1735-1780.

[25] SALINAS D, FLUNKERT V, GASTHAUS J, et al. Deepar: Probabilistic forecasting with autoregressive recurrent networks[J]. International Journal of Forecasting, 2020, 36(3): 1181-1191.

[26] BAI S, KOLTER J Z, KOLTUN V. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling[J]. arXiv preprint arXiv:1803.01271, 2018.

[27] HOSEINYFARAHABADY M, TAHERI J, TARI Z, et al. A dynamic resource controller for a lambda architecture[C]. 2017 46th International Conference on Parallel Processing (ICPP). IEEE, 2017: 332-341.

[28] 徐政钧. 无服务器云计算平台中函数启动加速器的设计与实现[D]. 北京邮电大学, 2020.

[29] KESIDIS G. Temporal overbooking of lambda functions in the cloud[J]. arXiv preprint arXiv:1901.09842, 2019.

[30] 樊大勇. Serverless 架构调度策略研究[D]. 北京工业大学, 2020.

[31] KAFFES K, YADWADKAR N J, KOZYRAKIS C. Centralized core-granular scheduling for serverless functions[C]. Proceedings of the ACM symposium on cloud computing. 2019: 158-164.

[32] 王子轲. 基于自适应容器池的无服务器计算冷启动优化研究[D]. 华中科技大学, 2021.

[33] JIANG Q, LEE Y C, ZOMAYA A Y. Serverless execution of scientific workflows [C]. Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings. Springer, 2017: 706-721.

[34] 马泽华林伟伟; 李加伟. 无服务器平台资源调度综述[J]. 计算机科学, 2021 (261-267).

[35] 裴云曼. 基于无服务器架构的云资源管理技术研究[D]. 华北电力大学 (北京), 2020.

[36] LI X, KANG P, MOLONE J, et al. Kneescale: Efficient resource scaling for serverless computing at the edge[C]. 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). IEEE, 2022: 180-189.

[37] https://azure.microsoft.com/zh-cn/products/functions/[EB/OL]. https://www.iron.

io/.

[38] Faas system simulator[EB/OL]. https://github.com/Yorklandian/GraduationDesign.

[39] Aws S3[EB/OL]. https://aws.amazon.com/cn/s3/.

[40] Aws RDS[EB/OL]. https://aws.amazon.com/cn/rds/.

[41] Kubeless[EB/OL]. https://www.serverless.com/framework/docs/providers/kubeless/guide/intro/.

[42] Docker[EB/OL]. https://www.docker.com/.

[43] Kubernetes[EB/OL]. https://kubernetes.io/zh-cn/.

[44] Redhat openshift[EB/OL]. https://www.redhat.com/zh/technologies/cloud-computing/openshift.

[45] Google Functions[EB/OL]. https://www.iron.io/.

[46] Ibm Cloud Functions[EB/OL]. https://www.ibm.com/cn-zh/cloud/functions.

[47] Nginx[EB/OL]. https://nginx.org/en/.

[48] Couchdb[EB/OL]. https://couchdb.apache.org/.

[49] Kafka[EB/OL]. https://kafka.apache.org/.

[50] Scala[EB/OL]. https://scala-lang.org/.

[51] Facebook Prophet[EB/OL]. https://facebook.github.io/prophet/.

[52] Linux c group[EB/OL]. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.

致谢

经过几个月的努力，本论文在胡创老师的指导下完成。从开始论文选题到系统的实现，再到实验的进行和论文文章的实现，每一步都是对我的考验与提高。从一步步的探索到完成的论文，胡老师对我的论文严谨认真负责地进行指导，提供科学合理的建议。同时，我也要感谢在完成论文的过程中，蔡昕F学长给予我的帮助。他的帮助、经验和指导是我在实验和写作过程中不可或缺的助力。我此前并没有系统严谨的论文写作经验，因此在实验和写作过程中遇到了不少的阻碍，胡老师和蔡学长的指导极大的提高了我的学术研究能力和论文写作水平，在此表示衷心的感谢。

本科四年的生活转瞬即逝。很庆幸，在校期间遇到的良师益友，无论在学习上，工作上还是生活上，都给予了我无私的帮助，陪我一起品尝求学的艰辛欢乐，受益匪浅。感谢我的三名室友，我在这四年里和你们一起度过了十分难忘的时光，愿我们的友谊长存。感谢我的家人，在我的学习上对我不懈支持，给我提供很大的帮助。我会继续努力，不管结局是好是坏，依然义无反顾地去做，并且不管发生什么都坚持到底。

最后，由于我的学术水平有限，所写论文难免有不足之处，望各位老师和同学批评指正。