优化无服务器计算中的冷启动问题的缓存与快照方案

李思涵

(华中科技大学 计算机科学与技术学院, 武汉市 430074)

摘　要　近年来，无服务器计算的模式因其便利性受到了广泛的欢迎，在云计算中扮演着越来越重要的角色，也对平台方的设备和技术水平提出了更高的要求。其中函数冷启动问题是影响用户体验的关键因素，引起了企业和学界的广泛关注。本文介绍了无服务器计算的基本概念和冷启动问题的成因，梳理近年的相关研究论文，着重探讨了基于缓存和快照的启动加速方案的设计思路。

关键词　无服务器计算；云计算；冷启动；函数即服务；负载预测；虚拟机快照

Optimizing cold start in serverless computing by caching and snapshot

Li Si-Han

[1)](Department of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China)

Abstract　In recent years, the serverless computing has gained widespread popularity due to its convenience. It plays an increasingly significant role in cloud computing, imposing higher demands on the equipment and technological capabilities of platforms. Cold start has emerged as a critical factor influencing user experience. This issue has drawn extensive attention from both enterprises and academia. This paper introduces the fundamental concepts of serverless computing and delves into the underlying causes of the cold start problem. Moreover, the paper reviews recent research papers in this domain. It emphasizes the exploration of design strategies for acceleration solutions, particularly focusing on approaches based on caching and snapshot mechanisms.

Key words　serverless computing; cloud computing; cold start; FaaS; workload prediction; VM snapshot

1　引言

无服务器计算（Serverless Computing）是一种新兴的云计算范式，它极大减少了开发者的负担、降低了使用门槛，自第一个无服务器计算平台 AWS Lambda 推出以来，这一概念就受到了市场的欢迎和学界的广泛关注。无服务器计算的冷启动问题是影响整体性能和用户体验的关键因素，本文旨在介绍并分析冷启动问题的成因，并着重梳理近年文献中基于缓存和快照的解决方案。

2　无服务器计算与冷启动问题

“无服务器”并非是指真的不存在服务器，而是指软件开发者不需要过多关心与服务器相关的底层细节，资源调配、运行维护等均由平台负责。从开发者角度来看，他们只需要提供函数及其触发条件，平台方会在满足触发条件时自动地运行函数，这就是“事件驱动（event-driven）”和“函数即服务（Function as a Service，FaaS）”的含义，它们与无服务器平台的弹性缩放（auto-scaling）和按量付费（pay-as-you-go）特性有紧密的关联。平台方要在

服务器上提供沙盒环境来实现函数实例间的隔离，保障运行环境的可用性和安全性。无服务器平台常用的沙盒技术可以分成容器和虚拟机两类。除了提供函数运行的服务器外，还需要相应的后端支持，如分布式存储、数据库等。

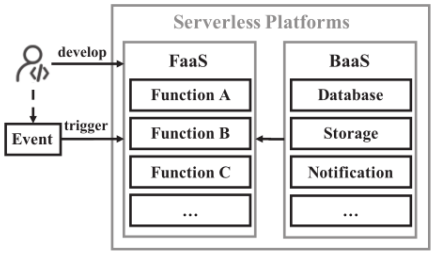

图 1 开发者视角的无服务器平台架构

无服务器计算的重要特性之一是弹性缩放，它包含横向和纵向两个方面的含义。横向的缩放是指分配新函数启动所需的资源，或回收执行完毕的函数占有的资源。纵向的缩放则是为正在运行的函数实例追加或回收资源。资源回收对于弹性缩放来说是必要的，用户自然不希望为函数执行以外的时间和资源付费，平台方也要尽量提高资源的利用率。但长期没有对应请求到来时，为函数分配的资源会下降至零，后续的请求将会面临冷启动（cold start）问题，也就是必须从零开始建立运行时环境。而相当一部分的 serverless 函数执行时间都很短，冷启动的开销就成了不可忽略的问题。在应对突发的负载高峰时，往往需要在多台服务器上进行冷启动，更是会严重影响整体服务质量。

以 OpenWhisk（使用容器隔离不同函数实例的运行环境）上运行的机器学习推断应用为例，其冷启动过程如图所示。首先 OpenWhisk 检查是否有可用的已初始化完毕的容器，若没有则启动一个容器。启动容器的过程包括拉取所需镜像、设置 cgroup 和 namespace 以完成容器化，以及运行时的初始化。最后一步往往耗时较长，需要准备编程语言的运行时，并根据函数中显式的要求导入依赖库，整个启动过程与函数本身的运行时间相当。

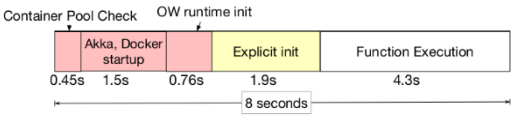

图 2 无服务器平台上函数的冷启动过程

优化启动开销的主要思路之一是利用已经初始化完毕的环境启动函数实例。一系列初始化操作的最大目的是构建好函数执行所需的内存状态，若能将这种状态保存下来，有请求到达时再按某种预定的方式进行恢复，就可以大大减少启动开销。本文将这类策略划分为基于缓存和快照两类，前者将状态维持在内存中，后者则将其保存为文件进行持久化存储，将在后面两部分进行详细阐述。

3 基于缓存的方法

为了解决冷启动问题，无服务器云计算平台广泛采取的策略包括预配置并发（provisioned concurrency）和保活（keep-alive）等。前者指在函数调用前即预先配置好函数的运行环境（如虚拟机、容器），亦可称其为预热（pre-warm）；后者则是函数执行完毕后的一段时间内（如几十分钟），仍保留其运行环境以减少冷启动开销，并应对可能的负载高峰。基于缓存的策略在单个函数的启动和执行速度上表现极佳，基本等同于最优解，但往往会大量消耗服务器上的计算资源尤其是内存资源，降低系统资源的总体利用率，所以必须要在性能表现和资源消耗之间找到平衡点，这也是对缓存策略研究的重点。

目前的研究主要有如下几种方向。一些研究专注于缓存的准入与淘汰策略，尽可能提高缓存的利用率并降低资源消耗，如下面介绍的 FaaSCache 和 Icebreaker 就分别着眼于保活和预取策略，这类方案具有比较好的普适性，对于虚拟机、容器乃至更小的缓存粒度都适用。还有一些研究通过探讨缓存和复用的粒度，尽可能地减少重复缓存，如 SOCK 就通过树型结构分析不同函数依赖间的包含关系避免重复，其中提出的用于加速容器化过程的方案——lean container 也在后续研究中得到了广泛的使用。此外还有偏重于调度策略和体系结构的研究等，实现不同粒度上的去重与复用。

3.1 FaaSCache：传统缓存策略的迁移运用

这篇文章首先将无服务器计算平台的保活策略与传统的 CPU-主存层的 cache 进行类比，保存一个函数执行环境相当于缓存一个对象，从保活的执行环境启动函数则对应缓存命中，相应地，终止这个执行环境就是将其从缓存中淘汰。在此基础上，已被充分研究过的缓存策略就可以迁移到无服务器计算平台上。

$$Priority = Clock + \frac{Freq \times Cost}{Size} \tag{3.1}$$

文中的策略以 Greedy-Dual 框架为基础，为每个函数单独计算优先级（统一函数的所有容器共享这一数值），优先级计算方式见公式 3.1。

这一公式综合了 LRU、LFU 的思想，并将函数的资源消耗与冷启动代价纳入考量。其中 Clock 是一个逻辑时钟，用以表征函数的 recency 信息，近期没有执行过的函数对应较小的 Clock 值。Freq 是函数执行的频率信息，记录全局范围内函数执行的总次数，一个函数对应的所有容器都终止时设 Freq 值为 0。Cost 等同于函数的总初始化时间，表示将其从保活状态淘汰需要付出的冷启动代价。Size 则表示函数所对应容器的资源消耗，与优先级负相关，保活消耗资源越多的越容易被淘汰。

为了实现弹性缩放还需要确定分配给各函数的缓存空间大小，一般情况下，找到 cache 命中率曲线的拐点即可。本文基于重用距离（reuse distance）对命中率作出了定义（式 3.2）

$$Hit-ratio\ (c) = \sum_{x=0}^{c} P\ (Reuse-distance = x) \tag{3.2}$$

其中重用距离表示对同一函数相邻的两次调用之间出现的所有不同函数的大小之和。用累积概率函数表示命中率与缓存大小的关系如图 3，可以看出缓存大小的增加有明显的边际效应。理论曲线（蓝色）与实测的曲线（红色）在缓存容量较小时存在一定偏差，原因可能在于 cache 与保活的机制并不能完美地一一对应。除了缓存大小不均以外，缓存对象的使用方式也有所不同。计算命中率的单位是函数，但每个函数可能拥有多个缓存中的容器实例。如果一个函数的所有缓存实例都已被占用，新到来的请求就只能经历冷启动过程，也就是“假命中”。在实际应用中计算准确的命中率曲线开销相当大，作者使用采样的方法，并结合历史命中信息对命中率曲线进行修正，达成了在线调整缓存大小的目标。

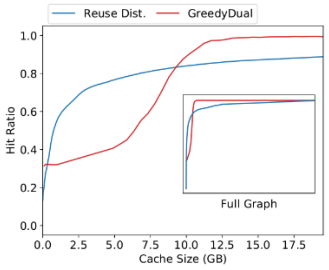

图 3 由重用距离定义的命中率曲线

3.2 IceBreaker：基于异构服务器的预热策略

IceBreaker 的主要贡献有两点，其一是引入了异构的服务器，其二是利用快速傅里叶变换（FTT）对负载预测机制作出了优化，并基于预测结果进行调度。

首先，对于完全采用同构服务器的情况，预热与保活显然会挤占大量计算资源；其次，在函数缓存期间，同样的函数被再次调用的概率并不是均等分布的，对于已不在较高调用概率区间的函数，仍让其长期占用资源也不够合理。类比分级 cache 的思想，作者提出使用异构的服务器，即性能较高、容量较小的高端服务器，以及性能较低、内存容量更大的低端服务器。理想情况下，将被调用可能性较低的函数放在低端服务器上预热、保活，就可以有效避免高端服务器的计算资源被浪费，在同等预算下达到更好的性能表现。

采用异构服务器对预热和保活策略提出了更高的要求，也需要对负载情况作出更准确的预测。作者观察发现，对函数的并发调用往往存在周期性，将并发度视为振幅，就可以用傅里叶变换对函数调用的冷热状态进行拟合和预测。而长期来看，调用函数的行为可能存在某种非周期性的变化趋势，作者的解决方案是将这一趋势用二次多项式拟合，剩余部分用 FTT 处理，最终使用的公式如式 3.3。在 IceBreaker 的函数调用预测器（FIP）的设计中，只考虑在本地时间窗口（如 12 小时）内的函数并发调用以减少预测开销。

$$FFT(f(t) - a \times t^2 + b \times t + c) = \sum_{i=1}^{n} cos(2\pi f_i t + \theta_i) \tag{3.3}$$

若 FIP 对一个函数给出了正的预测值，就可能需要为它进行预热，更详细的决策由部署决策器（PDM）作出。PDM 综合考虑了 FIP 的预测值、历史准确性（包括假阳性和假阴性）以及函数自身特性（如内存开销），计算得分后将其与高端服务器和低端服务器的阈值（可以动态调整）相比较，决定将函数放入高端或低端服务器预热，或暂不进行预热。

3.3 MITOSIS：通过远程 fork 减少所需缓存实例

前述的缓存方式一般默认是一对一的，即一个缓存的对象只能启动一个函数实例（如使用 docker unpause）。如果能让一个缓存实例负责多个函数的启动，就可以大大减少内存消耗。考虑到容器即为

一组进程，用于复制进程的 fork 系统调用就有了发挥空间。MITOSIS 则将 fork 的语义进行扩展，利用 RDMA 技术设计完成了远程 fork 系统调用，仅需常数级的缓存就可以在集群的所有服务器中启动对应的函数实例。

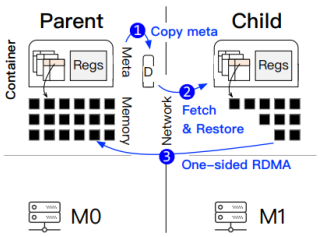

图 4　MITOSIS 的远程 fork 机制

MITOSIS 将 fork 的过程分为准备阶段和恢复阶段两部分。准备工作由父容器一方完成，将页表、寄存器内容、cgroup 和 namespace 等信息保存在一个描述符中。之后子容器通过 RPC 获取描述符的地址，利用 RDMA 读取父容器的页表等信息并完成 cgroup 和 namespace 的设置后就可以开始执行调用的函数。执行过程中采取 on-demand 的策略，仅在需要时才触发缺页错误读取对应的页。为此需要对内核中的缺页处理进行修改，令其能够处理所需页在其他服务器上的情况。

除了对启动的优化，MITOSIS 还能起到加速状态传递的作用。serverless 函数在执行完毕后，所用的资源会被回收，是无状态的。但函数间可能存在一些协作关系，组织成一个可以用有向无环图表示的工作流。规模较小的数据可以使用平台的消息传递机制，但对于大量数据，序列化和反序列化的开销就无法简单忽略。MITOSIS 基于 RDMA 的 fork 机制可以把上游函数内存中的数据直接传递给下游，免去序列化开销的同时也绕过了内核，提高了工作流的执行效率。

MITOSIS 虽然也是利用内存中缓存下来的实例，但涉及到单个实例的远程多次复用，实质上更加类似于下文所述的快照方法。

4　基于快照的方法

对于单个被调用的函数而言，基于缓存的方式可以达到非常理想的执行速度。但单独使用这种方式往往会造成内存资源浪费，间接地降低整体的资源利用效率和性能表现。相比之下，快照可以作为一种比较折中的选择。

快照机制由来已久，serverless 概念出现之前就已被用于虚拟机的迁移。应用于 serverless 平台时，快照捕获虚拟机的状态（包括虚拟机管理器 VMM、客户机的物理内存内容）并保存为文件；有对应的函数调用时，就可以从快照中迅速恢复函数实例，从而避免冷启动开销。恢复过程中，记录了客户机物理内存文件会被映射到虚拟内存页，多数解决方案在运行过程中采取 on-demand 策略，也就是为了缩短恢复过程的时间，只有在开始执行并触发缺页错误时才去读取对应的页，但缺页的处理开销很大，拖慢了运行时间。因此对快照方法的研究往往致力于降低缺页的发生率并缩短处理时间。

4.1　REAP：利用历史经验加速启动

作者经过实验观察发现，使用快照机制启动的函数执行耗时比热启动长 95%，对于热启动情况下执行时间很短的函数，甚至有 2-3 个数量级的差距，其中大部分的时间用于处理缺页错误。而由于运行过程中产生的缺页错误缺乏局部性，导致宿主 OS 的 read-ahead 预取机制难以发挥作用，只能单独处理大量离散的缺页错误，从硬盘中读取不连续的页，导致了严重的延迟。

另一方面，经过实验观察，采取 on-demand 策略时往往并不需要读取整个快照文件，因为实际运行函数时只会用到少量内存页，与整个启动过程的内存消耗相比少得多（如图 5，运行过程中读取的内存页只占总量的 3%-39%，平均 9%）。REAP（REcord-And-Prefetch）的设计就从此处着手，以内存页为单位记录函数的工作集，在后续的函数调用中使用记录下来的工作集主动进行预取，有效减少执行过程中缺页错误的触发次数。

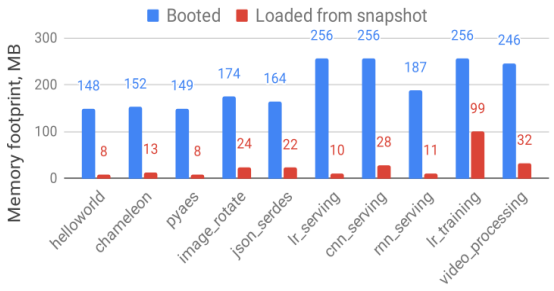

图 5　一次调用后函数的内存页访问行为

REAP 主要包括两种操作，分别是记录阶段和预取阶段。在 REAP 的设计中，使用 monitor 线程来完成记录或预取工作，当调用的函数没有可用的缓存实例时就会启动。monitor 完全工作在用户态，利用 Linux 的 userfaultfd 调用，在用户态处理缺页

错误。

记录阶段，monitor 通过用户错误文件描述符获取缺页错误的详细信息，在处理缺页错误的同时，记录这些页在 guest 内存中的偏移，待被调用的函数进程完成后，把对应的页保存在一个单独的工作集（WS）文件中，人为地制造了局部性。预取阶段，monitor 通过单次 read 系统调用读取 WS 文件，主动将这些页安装到页表中，唤醒目标函数实例进入执行阶段。这样对于在 WS 中的页，缺页处理时间会大大缩短。只要同一函数所有实例的访存行为都相似，这种方式就可以发挥作用。monitor 还绕过了 host 操作系统的页缓存机制，充分利用了存储的带宽，读取 WS 文件时可以达到接近 SSD 峰值的吞吐量。

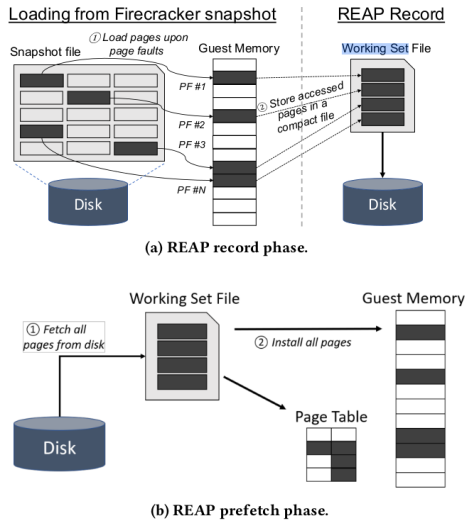

图 6　REAP 在记录和预取阶段的操作

4.2 FaaSnap：并发的页预取

与 REAP 的方法类似，FaaSnap 也使用了预取工作集的机制，但并不会在加载整个工作集后才开始函数执行阶段。FaaSnap 守护进程收到调用请求时就会启动 loader 线程开始预取工作，VMM 完成虚拟机的设置后就立刻启动虚拟机。如果一个页由 loader 首先获取，这页就会被放进页缓存区，之后虚拟机触发缺页时就可以快速处理。基于这种“并发”的预取机制，优先读取最先被需要的页是很有必要的。但为了兼顾局部性，FaaSnap 把工作集进行了分割。首先按照上次调用中页的使用顺序分组，保证优先读取最早的一部分页；在各组内部则按照偏移量进行排序，以充分利用提前读。

本文的作者还从虚拟机的缺页错误处理过程本身入手，对比了缓存和一般的快照恢复方式。缓存方式中，guest 内存区被映射为 host 的匿名内存，缺页错误的处理较快；而快照机制中，guest 内存区需要从快照文件恢复，所有的内存页都被映射为 file-backed 类型。注意到 guest 内新分配的匿名页都会初始化为全 0,那么从 host 读取页这个行为本身就是没有意义的。FaaSnap 采取了简化的方式，只把非零页映射到 guest 内存文件中，全 0 页（无论是否为 guest 内部的匿名页）则直接映射为匿名 host 内存。在实际的恢复过程中，就只需要加载工作集中的非零页。对于不同调用期间工作集差距较大的函数，FaaSnap 得益于其并发的预取机制，在函数执行时间上的表现超出 REAP。

5 总结

加速函数的启动是无服务器计算领域的热点研究方向之一，对用户和平台方都有重要意义。本文梳理了近期文献中基于缓存和快照的加速方案，分析其主要研究思路。这类方案未来的优化方向可能包括如下几点：对工作负载更加准确的预测方法，并在此基础上设计预热策略；针对无服务器计算缓存的独特特点设计保活策略，并完善类似 cache 的评价机制；优化函数调度和负载均衡机制，最大化缓存的效果；保证隔离性和安全性的基础上，进一步共享函数间公有的依赖库，减少计算资源的浪费等。

参 考 文 献

[1] Ustiugov D , Petrov P , Kogias M ,et al. Benchmarking, analysis, and optimization of serverless function snapshots. Architectural Support for Programming Languages and Operating Systems. 2021.

[2] Oakes E., Yang L., Zhou D., et al. SOCK: Rapid task provisioning with Serverless-Optimized containers. In Proceedings of 2018 USENIX annual technical conference. 2018.

[3] Fuerst A, and Prateek S, et al. FaasCache: keeping serverless computing alive with greedy-dual caching. Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021.

[4] Roy, Rohan Basu, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022.

[5] Wen Jinfeng, Chen Zhenpeng, et al. Rise of the planet of serverless computing: A systematic review. ACM Transactions on Software Engineering and Methodology (2023).

[6] Ao Lixiang, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In Proceedings of the Seventeenth European Conference on Computer Systems. 2022.

[7] Wei Xingda, et al. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation. 2023.

附录：汇报记录

问题：（MITOSIS）的主要贡献在于什么地方？文中用到的技术都比较成熟。

补充回答：结合RDMA设计了远程fork系统调用，可以用于远程迁移进程、容器等等，取得了远高于用途相近的CRIU的性能，因而可以应用于无服务器计算平台。从原理上讲，作者所做的更像是把从快照恢复容器或虚拟机状态的技术迁移到了缓存上。远程fork的思路过于顺理成章，以至于让人怀疑为什么没有前人涉足，猜测其主要难点在于工程实现；在内核空间使用RDMA也引入了额外的安全性问题。作者在会议上的展示偏重于领域的发展历程和两个实验案例，缺少对实现方案细节的讲解。