针对容器与函数的云计算加速方案

干捷 1)

1)(华中科技大学武汉光电国家研究中心，湖北省武汉市 435400)

摘 要 云计算是一种新兴的计算模式。在预定执行之前，先对更新功能进行预估，以便为用户提供更快捷、更具成本效益的服务。不幸的是，预热函数可能是不准确的，并且在预热期间会产生非常昂贵的成本(即维持成本)。云计算模式的兴起导致了容器镜像的广泛扩散。相关的存储性能和容量需求给存储和服务映像的容器注册中心的基础设施带来了很大的压力。利用实际容器映像中的高文件冗余是一种很有前途的方法，可以大幅降低日益增长的注册中心的苛刻存储需求。但是，现有的重复数据删除技术会因为较高的恢复开销而显著降低注册表的性能。本文讨论了针对云计算的优化方法，主要包含针对函数的优化与针对容器镜像的优化，来提高云计算时的效率，减少云计算成本。

关键词 云计算；容器；函数加速；服务器异构；文件系统

Cloud computing acceleration solution for containers and functions

Jie Gan[1)]

1)(Wuhan National Laboratory for Optoelectronic,Huazhong University Of Science and Technology,Wuhan,Hubei)

Abstract Cloud computing is a new computing model. Prior to scheduled execution, update features are estimated to provide faster and more cost effective service to users. Unfortunately, warm-up functions can be inaccurate and incur very expensive costs (i.e. maintenance costs) during warm-up. The rise of cloud computing has led to the proliferation of container images. The associated storage performance and capacity requirements put a lot of strain on the container registry infrastructure for storing and serving images. Leveraging high file redundancy in actual container images is a promising way to dramatically reduce the demanding storage requirements of growing registries. However, existing deduplication techniques can significantly reduce registry performance due to high recovery overhead. This paper discusses optimization methods for cloud computing, including optimization for function and optimization for container image, to improve the efficiency of cloud computing and reduce the cost of cloud computing.

Key words Cloud computing; Container; Function acceleration; Server heterogeneity; File system

1. 介绍

serverless 范式在多个领域变得越来越普遍，包括 web 应用程序、微服务、延迟关键型工作负载、数据处理和机器学习。这是因为 serverless 降低了终端用户的进入门槛，使他们能够以弹性伸缩的按次计费模式使用云计算资源，而无需担心计算资源的管理和供应。serverless 函数是在 docker 实例上生成的。它们会遇到冷启动的问题，即设置实例和将应用程序逻辑加载到内存的开销(也称为冷启动时间)。

这种开销占 serverless 函数执行时间的很大一部分，因为由于 serverless 实例的资源有限，serverless 主要吸引短时间运行的任务(以秒为单位)。

避免冷启动最直观的解决方案是在内存中保持无服务器函数实例的设置和加载，这样当函数被调用时，就可以避免冷启动开销。这个使函数在内存中保持活跃的过程称为预热。但是，保持函数活跃会占用内存，并为无服务器提供商带来保持活跃的成本，从而增加资本预算。保持活动成本定义为

三个术语的乘积:保留服务器的单位内存和单位时间的成本，容纳函数实例所需的内存，以及函数保持活动的时间。如果函数在内存中保持活动状态时被调用，它将经历一个热启动，因此执行时不会引起冷启动开销。

随着无服务器工作负载随着不同的调用模式而变得多样化，决定函数保持存活多长时间以及何时对其进行预热以避免冷启动变得越来越困难。

无服务器提供商通常在函数调用后遵循 10 分钟的保持活动策略，但这种固定策略并不能确保针对不同的调用特征缓解冷启动。

为了减少冷启动的影响，以前的工作提出了超预定服务器实例和预缓存应用程序的不同部分等策略。然而，这些技术增加了资金成本。其他预测函数到达时间的策略也会增加维持生存的成本。尽管完整的服务器是为创建无服务器计算平台而保留的，但降低维持生存成本非常重要，因为它直接将内存浪费降到最低。这有助于服务器容纳更多的功能。

针对函数冷热的问题，Rohan 等人提出了 icebreaker 模型，利用傅里叶变换的方法，将函数放在异构服务器中执行，动态调整其运行模式，取得了较好的效果。Mohammad 等人提出，需要综合考虑 FaaS 的工作负载以及各函数间的触发器情况，设计了一套通过函数冷热来减少冷启动的模型。

此外，在云计算中，容器镜像的启动也是不可忽视的一部分。容器映像是容器化应用程序的核心。一个应用程序的容器镜像包含了可执行的文件及其依赖（即应用程序所需的其他可执行文件、库以及配置和数据文件）。

图像是分层结构的。当使用在 Docker 中，每个执行的命令(如 apt install)都会在之前的层之上创建一个新层，其中包含命令修改或添加的文件。Docker 利用联合文件系统在启动容器时有效地将层合并到单个文件系统树中。容器可以在不同的映像中共享相同的层。

为了存储和分发容器镜像，Docker 依赖于镜像注册表(例如，Docker Hub)。Docker 客户端可以根据需要将图像推送到注册表中或从注册表中拉出。在注册表端，每一层都存储为压缩的 tarball，并由基于内容的地址标识。Docker 注册表支持各种存储后端来保存和检索层。例如，典型的大规模设置将每个层作为对象存储在对象存储中。

容器映像的存储需求由于映像中大量的重复数据而恶化。由于 Docker 映像在定义上必须是自包含的，不同的映像经常包含相同的、公共的依赖项(例如库)。因此，不同的映像容易包含大量重复的文件，因为共享组件存在于多个映像中。针对该问题，Nannan 等人提出了 DupHunter 模型在 docker registry 中对容器镜像进行去重操作，提高了容器镜像的使用效率。

Huiba 等人注意到容器镜像格式的优化可能性，提出了使用块级的存储来对镜像进行优化，实现了 DADI（Data Acceleration for Disaggregated Infrastructure）模型。DADI 映像服务的核心是一个名为叠加块设备(OverlayBD)，它提供了基于块的层序列的合并视图。从概念上讲，它可以被视为目前通常用于合并容器映像的联合文件系统的对应物。它比联合文件系统更简单，这种简单性使得优化成为可能，包括层的扁平化，以避免具有许多层的容器的性能下降。

为了减少内存浪费和最小化函数响应时间，Zijun 等人建议通过容器共享来缓解冷容器启动，而不是预热容器。例如，如果图 1 中的函数 A 可以从其空闲的暖容器中为函数 B“fork 或出借”运行时检查点，那么就消除了 B 的冷启动。通过这种方式，我们可以利用一个函数在回收之前的空闲热容器来帮助其他倾向于体验冷容器启动的函数。

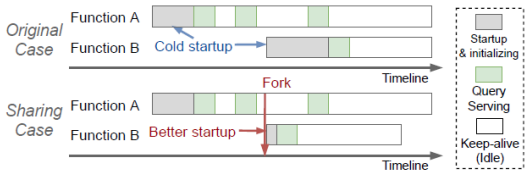

图 1 分支函数 A 的空闲容器缓解函数 B 的冷启动

他们观察到有以下三个挑战：

（1） 函数调用负载不稳定[51]。很难判断函数 A 的温容器是否实际闲置。如果一直用 A 的温容器来帮忙 B, A 的调用可能无法获得一个热容器，甚至可能会违反服务质量(QoS)。

（2） 功能依赖不同的软件包。当函数 A 为函数 B“分支”一个暖容器时，这个暖容器必须安装额外的包。安装和导入这些包可能需要比冷容器启动更长的时间。更糟糕的是，贪婪的重新包装过多的功能也会导致巨大的图像大小或引起包版本矛盾。

（3） 从其他函数中“分叉”空闲容器可能会引入安全漏洞。虽然函数的代码或数据

是私有存储的，但与其他函数共享容器是有风险的

他们在 Help Rather Than Recycle 中提出温容器共享的概念，有效的避免了函数的冷启动问题。

2. 基于函数的优化方案

在该节中，我们主要介绍在云计算加速方案中，针对函数冷热特性实现的加速方案模型，主要包含对函数冷热的预测，以及对 FaaS 负载的研究。

2.1 IceBreaker 模型

2.1.1 IceBreaker 介绍

Rohan 等人提出的 icebreaker 模型主要基于服务器的异构模式以及傅里叶变换预测函数冷热。主要贡献下：

IceBreaker 是一种针对无服务器函数的新型函数预热和保活方案，它利用服务器异构性来降低保活成本和缩短服务时间。IceBreaker 是第一个提出使用异构服务器(高端和低端)混合的模型。IceBreaker 适当地决定在哪里加热无服务器功能，以获得最大的效益，同时降低维持生存的成本。

IceBreaker 的异构思想的有效实现需要一个有效的函数调用预测机制。IceBreaker 设计并实现了一种新的基于快速傅里叶变换的预测机制，该机制消除了现有工作中的这些限制，并且实验表明，对于最近的生产级无服务器功能跟踪，即使在同构集群上，其性能也优于现有策略，最高可达 27%。

IceBreaker 为每个函数分配了一个基于几个重要因素的积分，这些因素包括到达概率、下一次到达的预测和预期的并发度、不同服务器类型的函数在执行时间上的相对收益，以及维持生存成本。IceBreaker 将其预测机制和效用估计结合起来，以确定加热函数的适当位置和其存续时间。

2.1.2 IceBreaker 设计

如图 2 所示，为 IceBreaker 的总体设计。其核心部分有函数调用预测器（FIP，Function Invocation Predictor）与布局决策器（PDM，Placement Decision Maker)。FIP 主要负责预测用户将要调用的函数及并发度，动态调整模型。PDM 主要负责决定在何处何时预热函数，或根本不预热。

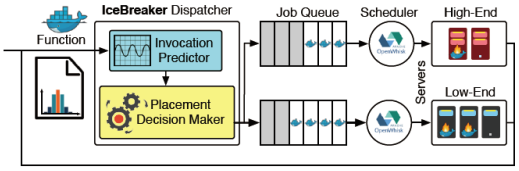

图 2　IceBreaker 总体设计

IceBreaker 的 FIP 使用傅里叶变换来预测函数调用的冷热状态，使用下式拟合：

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

并使用下式预测：

$$f(t_k+1)=a(t_k+1)^2+b(t_k+1)+c+\sum_{i=1}^{n}cos(2\pi f_i(t_k+1)+\theta_i)$$

这样就得到了函数调用频率的预测值。

IceBreaker的PDM使用了四个指标来判断该函数放在何种位置，分别是：

True negative prediction rate（Tn）

False positive prediction rate（Fp）

Inter-server speedup（Is）

Memory Footprint（Mr）

并按照算法 1 中所示，给函数执行位置进行分类。

```
Initialize the high and low-end servers, f(t) ← Prediction
concurrency, S_u ← Utility score
for Every time interval t do
    for Every time function do
        I_s ← Inter-server speedup, M_r ← Memory
        f(t) ← at^2 + bt + c + Σ_{i=1}^{n} cos(2πf_i t + θ_i)
        if f(t) > 0 then
            Update F_p, T_n, H_E, L_E
            S_u ← [T_n + (1 − F_p) + (1 − I_s) + (1 − M_r)]/4
            if S_u > H_E then Warm up on high-end server
            else if S_u < L_E then Do not warm up
            else Warm up on low-end server
    if A function is invoked and server is available then
        Execute it with warm or cold start
```

算法 1　PDM 执行算法

IceBreaker 使用 Apache OpenWhisk[40]，这是一种广泛使用的开源无服务器部署，用于在预留服务器上创建无服务器平台。IceBreaker 使用两种类型的服务器(低端和高端)，带有一个主节点来控制功能布局。IceBreaker 使用两级设计堆栈实现:(1)服务器间分派器，(2)OpenWhisk 控制器。

2.2 基于函数负载冷热的优化

FaaS 以一种在云端将计算任务部署到 serverless 服务后端的方式变的越来越流行，这种方式改变了开发者向云服务提供商申请资源的复杂性，而这种方式也与云原生时代所追求 ″充分

利用云的能力以最短路径释放云的价值” 的理念相吻合。当前各大云服务提供商都有对应的 FaaS 服务，如国外的 AWS Lambda、Microsoft Azure Functions 、 Google Cloud Functions 、 IBM OpenWhisk，以及国内的阿里云函数计算、腾讯云函数等。

对于用户，希望 FaaS 服务以极致弹性的方式提供持续可用的资源。而对于云提供商，则希望以尽可能少的资源满足用户的需求。Mohammad 等人以 Microsoft Azure Functions 生产环境数据为基础，通过对 FaaS 用户工作负载的分析与研究，提出一种自适应策略方式，提高 FaaS 服务的资源利用率并降低冷启动率。

2.2.1 FaaS 工作负载研究

图 3 主要描述了每个 Application 的 Function 数目，可以看到，54%的 App 仅仅包含一个 functions，95%的 App 拥有最多不超过 10 个 functions，而只有大约 0.04%的 App 拥有超过 100 个 functions。

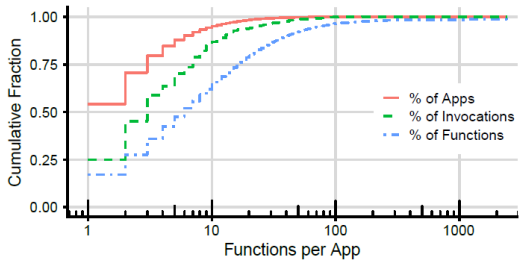

图 3 每个应用函数数量分布

另外两条曲线表示的是 Functions（函数）和 Invocations（调用）主要出现或发生在那类 App 中的比例，比如：50%的调用来自拥有 function 数目不超过 3 个的 App;50%的函数来自拥有 function 数目不超过 6 个的 App。

图 4 展示了 Function 和 Invocation 的触发器类型的占比。可见 HTTP 是最受欢迎的触发器，55%的 function 和 35.9%的 Invocations 都是由 HTTP 触发。这里需要注意，Queue 和 Event 类型虽然在 Function 中占比较小，但是在 Invocation 中占比很大，这是因为这些触发器是自动化地，频繁地被触发。而 Timer 则相反，虽然在 Function 中占比较大，但是在 Invocation 中占比很小，说明定时器的定时时长都很高。

Trigger	%Functions	%Invocations
HTTP	55.0	35.9
Queue	15.2	33.5
Event	2.2	24.7
Orchestration	6.9	2.3
Timer	15.6	2.0
Storage	2.8	0.7
Others	2.2	1.0

图 4 函数与调用的触发器类型

图 5 表示，一天当中 function 的调用频率。App 的调用频率肯定是大于 Function 的，因此 App 的曲线更低。上横坐标表示每间隔多久调用一次，因为该图是基于一天的当中调用频率，因此 1d 对应的调用次数为 10^0 次；下横坐标则是一天中总计的调用次数。

首先我们可以发现 function 的调用频次横跨了 8 个数量级，展示此 serverless 调用的高度差异性(highly variable). 其次可以观察到大多数的 function 或者 application 的调用频率都非常低，图上不同的颜色区域分别表示了>1h, >1min 以及>0 调用间隔，可以看到有 45%的 application 的调用间隔在 1 分钟以上，80%的 application 的调用间隔在 1min 以上。

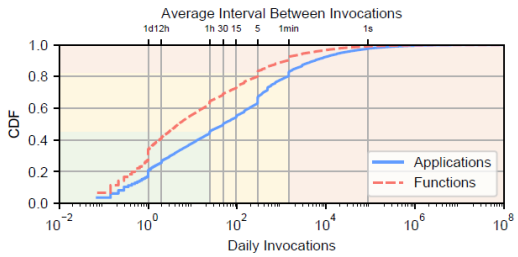

图 5 调用平均间隔

可以发现 function 的调用频次横跨了 8 个数量级，展示此 serverless 调用的高度差异性(highly variable). 其次可以观察到大多数的 function 或者 application 的调用频率都非常低，图上不同的颜色区域分别表示了>1h, >1min 以及>0 调用间隔，可以看到有 45%的 application 的调用间隔在 1 分钟以上，80%的 application 的调用间隔在 1min 以上。

基于这些观察，我们现在重申在 Introduction 中提出的三点：

首先，绝大多数函数的执行时间大约为几秒—其中 75% 的最大执行时间为 10 秒—因此执行时间与启动函数所需的时间相同。因此，减少冷启动次数或大幅加快冷启动速度至关重要。消除冷启动与使其启动无限快相同。

其次，绝大多数应用程序很少被调用——其中 81% 的应用程序平均每分钟最多调用一次。同时，不到 20% 的应用程序负责 99.6% 的所有调用。因此，就内存占用而言，始终保持接收不频繁调用的应用程序是昂贵的。

第三，许多应用程序在其 IAT 中表现出广泛的可变性—其中 40% 的 IAT 的 CV 高于 1—因此预测下一次调用的任务可能具有挑战性，尤其是对于不经常调用的应用程序。

2.2.2　冷启动的消除

提出两个窗口：

Pre-warming window.

预热窗口，即 function 执行结束后会被立即释放，经过预热窗口大小的时间后，会再次启动 function 实例。预热窗口的值可以为 0，即表示，function 执行结束后不会立即释放，而是继续运行。

Keep-alive window

保活窗口，当实例被预热后持续运行等待的时间。

如图 6 所示，Top：预热窗口为 0，这样函数执行结束后会立即进入保活状态，在保活窗口结束之前收到了一个请求，此时即为暖启。当这个请求执行结束后，将重新进入保活窗口。

Middle：预热窗口不为 0，函数执行结束后，经过一个预热窗口后，会启动一个实例然后进行保活，保活窗口内收到了请求，此时也是暖启动，执行结束后，进入下一个预热窗口；

Bottom：预热窗口不为 0，函数执行结束后，在预热窗口结束之前，接收到了请求，此时为冷启动，执行结束后，进入下一个预热窗口，经过这个预热窗口后，会启动一个实例然后进行保活，但是在保活窗口中没收到请求，于是结束保活，进入预热窗口，此时收到了请求，那么又要进行冷启动。

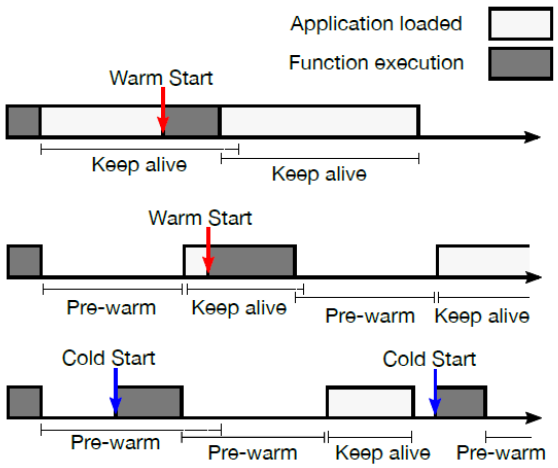

图 6 冷启动的消除策略

3. 基于容器的优化方案

在本节中，我们主要介绍云计算加速方案中针对容器优化，这些优化包括：在 docker registry 中对 docker images 进行去重来提高冷启动效率、将 docker image 格式转化为块格式提高效率与安全性、在函数运行时将空闲温容器分支给冷启动函数服务。

3.1 DupHunter 模型

观察发现：

1.容器镜像拥有大量冗余，而现有的重复数据删除技术不适用于 Docker registry，会导致极高的 layer 恢复时间开销。

2.大多数 layer 只会被同一用户获取一次；许多用户只 pull 一个 layer 一次；用户对于 layer 一般要么只 pull 一次，要么总是 repull，所以可以根据一个用户的 pull 历史来预测他要 pull 哪些 layer。

3.Docker 客户机从 registry 中提取镜像时，首先检索镜像 manifest，其中包括对镜像 layer 的引用。GET manifest 和 GET layer 请求之间往往存在秒级的间隔。在此间隔内预测用户需要哪些 layer 并对 layer 进行预构造能显著地降低 layer 恢复的时间开销。

3.1.1 DupHunter 架构

如图 7 所示，DupHunter 主要由两部分组成：一个分布式元数据数据库和一个存储服务器集群。

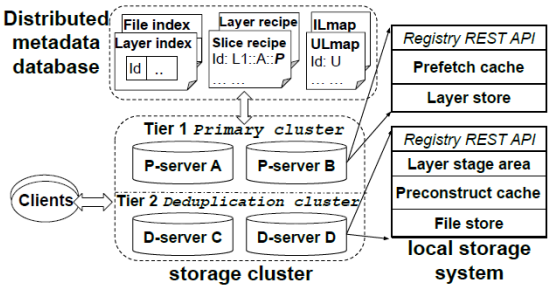

图 7 DupHunter 架构

在上传或下载 layer 时，Docker 客户端可以使用 registry API 与任何 DupHunter 服务器进行通信。集群中的每个服务器都包含一个 API 服务和一个后端存储系统。 后端存储系统存储 layer 并执行重复数据删除，最后将重复数据删除元数据保存在数据库中。

服务器集群中又分为由 P-server 组成的原始集群服务器(Primary cluster)和由 D-sever 组成的重删集群服务器(Deduplication cluster)。P-sever 负责存储无需进行重复数据删除的完整 layer 压缩文件副本和 manifest 副本；D-sever 负责对 layer 压缩文件进行重复数据删除并存储和复制 unique 文件。

每个 D-server 的本地存储分为三个部分：layer stage 区域、预构建缓存和文件存储。layer stage 区域会临时存储新添加的 layer 完整副本。在对副本进行重复数据删除后，生成的 unique 文件存储在内容可寻址的文件存储中，并复制到对等服务器以提供冗余。存储了所有文件副本后就会从 layer stage 区域删除 layer 副本。

3.1.2 DupHunter 核心技术

1.Replica deduplication modes：

①basic deduplication mode n (B-mode n)：代表 DupHunter 只保持 n 个完整的 layer 副本，并对剩余的 R-n 个副本进行重复数据删除(R 为 layer 复制级别)。在极端情况下，B-mode R 代表不进行重复数据删除，提供了最好的恢复性能但是没有数据减少；B-mode 0 代表对所有 layer 副本进行重复数据删除，提供了最高的重复数据删除率，但增加了恢复开销。

②selective deduplication mode (S-mode)：根据 layer 被 pull 的频率（即热度）来决定对多少个副本进行重删。在 S-mode 下，layer 完整副本的数量与其热度成正比，这意味着热 layer 有更多完整的副本，因此可以更快地提供服务，而冷 layer 则可以更积极地进行重复数据删除。

为了适应不同的模式，DupHunter 如前面所说将服务器分为原始集群服务器和重删集群服务器两种，在默认情况下，原始集群服务于所有传入的 GET 请求。如果不能从原始集群提供服务(例如，节点故障，或 DupHunter 在 B-mode 0 或 S-mode 下运行)，请求将被转发到重复数据删除集群，并将重建请求地 layer。

2.Parallel layer reconstruction：

DupHunter 通过并行性加快了 layer 重建的速度。DupHunter 将 layer 的 unique 文件分发到几个服务器上。一个服务器上属于同一 layer 的所有文件被称为切片。切片有相应的切片 recipe,layer recipe 定义重建该 layer 所需的切片。这些信息存储在 DupHunter 的元数据数据库中。多个 D-server 存储着一个 layer 的不同切片使得 D-sever 可以并行地重建 layer 切片，从而提高重建性能。DupHunter 在元数据数据库中维护 layer 和文件的指纹索引。

3.Predictive cache prefetch and preconstruction：

为了降低 layer 的访问延迟，DupHunter 分别在原始集群和重复数据删除集群中使用了一个缓存层。每个 P-server 都有一个 in-memory 的基于用户行为的预取缓存，以减少磁盘 I/Os。当收到来自用户的 GET manifest 请求时，DupHunter 预测镜像中实际需要提取哪些 layer，并在缓存中预取它们。此外，为了减少 layer 恢复开销，每个 D-server 都有一个 on-disk 的基于用户行为的预构造缓存。与预取缓存一样，当接收到 GET manifest 请求时，DupHunter 预测镜像中的哪些 layer 将被提取，预构造这些 layer，并将它们加载到预构造缓存中。为了准确地预测要预取哪些 layer，DupHunter 维护了两个映射：ILmap 和 ULmap。ILmap 存储镜像和 layer 之间的映射，而 ULmap 跟踪用户的访问历史，即用户 pull 了哪些 layer 以及 pull 它们的次数。

3.2 DADI 镜像格式

DADI 借鉴了 overlayfs 和早期联合文件系统（union filesystem）的思想，但提出了一种全新的基于块设备的分层堆叠技术，称之为 overlaybd，它为容器镜像提供了一系列基于块的合并数据视图。overlaybd 的实现十分简单，因此很多之前想做而不能做的事都可以成为现实；而实现一个完全 POSIX 兼容的文件系统接口则充满挑

战，并可能存在 bug，这点从各个主流文件系统的发展历史上就可以看出。

如图 8 所示，分别介绍各个部件：

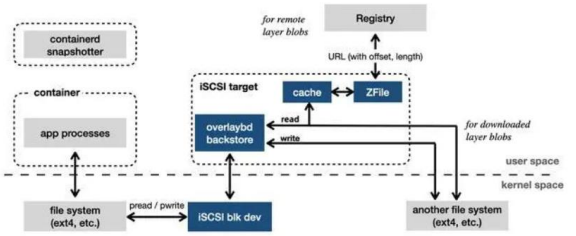

图 8 DADI 架构

1. containerd snapshotter

snapshotter 的核心功能是实现抽象的服务接口，用于容器 rootfs 的挂载和卸载等操作。它的设计替代了在 Docker 早期版本称之为 graphdriver 的模块，使得存储驱动更加简化，同时兼容了块设备快照与 overlayfs。

DADI 提供的 overlaybd-snapshotter 一方面能让容器引擎支持新的 overlaybd 格式的镜像，即将虚拟块设备挂载到对应的目录，另一方面也兼容传统 OCI tar 格式镜像，让用户继续以 overlayfs 运行普通容器。

2. iSCSI target、

iSCSI 是一种被广泛支持的远程块设备协议，稳定成熟性能高，遇到故障可恢复。overlaybd 模块作为 iSCSI 协议的后端存储，即使程序意外 crash，重新拉起即可恢复。而基于文件系统的镜像加速方案，例如 stargz，则无法恢复。

iSCSI target 是 overlaybd 的运行时载体。在本项目中，我们实现了两种 target 模块：第一种是基于开源项目 tgt，由于其拥有 backing store 机制，可以将代码编译成动态链接库以便运行时加载；第二种是基于 Linux 内核的 LIO SCSI target（又称为 TCMU），整个 target 运行在内核态，可以比较方便地输出虚拟块设备。

3. ZFile

ZFile 是一种支持在线解压的数据压缩格式。它将源文件按固定大小的 block size 切分，各数据块进行单独压缩，同时维护一个 jump table，记录了各数据块在 ZFile 中的物理偏移位置。如需从 ZFile 中读数据，只要查找索引找到对应位置，并仅解压缩相关的 data block 即可。

ZFile 支持各种有效的压缩算法，包括 lz4，zstd 等，它解压速度极快，开销低，可以有效节省存储空间和数据传输量。实验数据表明，按需解压远程的 ZFile 数据，性能高于加载非压缩数据，这是因为传输节省的时间，大于解压的额外开销。

overlaybd 支持将 layer 文件导出成 ZFile 格式。

4. cache

layer 文件保存在 Registry 上，容器对块设备的读 I/O 会映射到对 Registry 的请求上（这里利用到了 Registry 对 HTTP Partial Content 的支持）。但是由于 cache 机制的存在，这种情形不会一直存在。cache 会在容器启动后的一段时间后自动开始下载 layer 文件，并持久化到本地文件系统。如果 cache 命中，则读 I/O 就不会再发给 Registry，而是读本地。

3.3 共享温容器

Zijun 等人提出一套缓解 Serverless 冷启动的机制。其核心是将空闲的待回收(Recycle)的容器转化为可以帮助(Help)其他 function 缓解冷启动的 zygote 容器

zygote 是 Android 系统中的一种优化机制，由 SOCK 首次将 Zygote 容器的概念引入到 Serverless 的冷启动优化中，并将基于 Zygote 容器来 fork 新容器的方式作为其核心贡献；此后 Catalyzer 和 Faasm 中也使用到了类似的 zygote 思想。与 SOCK 的 Zygote Tree 相比，这种方式（这个观点是我凭借对此论文以及 SOCK 的理解做出的判断，论文并未详细论述其与 SOCK 的区别）：

不需要维护一个 zygote 容器缓存树(之所以是树，是因为 SOCK 是基于 Python 包通过 fork 一层层构建的，所有包含相同 package 的 function 公用一个 zygote 容器)。

SOCK 的 zygote 只能 fork 与包与其完全相同的 function 的容器，而本文的 zygote 容器更加的通用，即每个 zygote 可以为多个 function fork 容器实例，即使他们的包不相同；并且由于 zygote 容器不止一个，因此更能适应高并发的场景，即我可以多个模板一起 fork。

SOCK 的 fork，是在 python 执行了 import 命令之后的，也就是说，这些包已经被全部或者部分的加载到了内存中；而本文的 fork，据我看是没有 SOCK 共享的这么彻底，仅仅是将应用可能用到的包匿名挂载到了容器中，仅仅节省了从网络中下载这一过程。

如图 9 所示，每个 function 都拥有三类容器

池：

Private 容器池，专用于自己的容器，处理完 Invocation 之后会成为 Idle 容器，被识别后就会被替换为 Zygote 容器

Zygote 容器池，主要的作用是通过 fork 帮助其他 function 冷启动

Helper 容器池，通过其他 function 的 Zygote 容器 fork 而来。

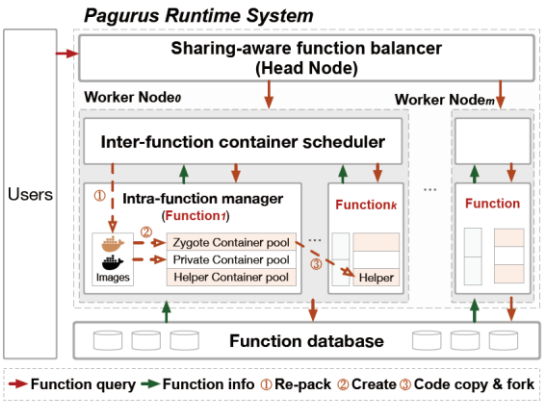

图 9 共享容器架构

4. 实验对比总结

与最先进的模型相比，IceBreaker 模型降低了 45%的维持成本和 27%的使用时间。IceBreaker 可以部署在任何具有无服务器工作负载管理器的集群中，对最终用户和服务提供商都有好处。

Mohammad 等人使用模拟和真实的实现以及真实的工作负载跟踪来评估策略。结果表明，该策略可以以较低的资源成本实现相同的冷启动次数，或者保持相同的资源成本但显著减少冷启动次数。

DupHunter 引入了两层存储层次结构，并基于用户访问模式引入了一种新颖的层预取/预构造缓存算法。与现有技术相比，DupHunter 的预取缓存可以将 GET 延迟提高 2.8 倍，而预构造缓存可以将恢复开销降低 20.9 倍。

DADI 基于块的层的相对简单性进一步促进了优化以提高敏捷性。这些功能包括远程图像的细粒度按需数据传输、使用高效编解码器的在线解压、基于跟踪的预取、处理突发工作负载的点对点传输、与容器生态系统的轻松集成。

Pagurus 显著地缓解了冷容器启动。如果 Pagurus 加以缓解，冷启动延迟将从数百毫秒减少到 16 毫秒。

参 考 文 献

[1] Roy R B, Patel T, Tiwari D. IceBreaker: warming serverless functions better with heterogeneity[C]//Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022: 753-767.

[2] Shahrad M, Fonseca R, Goiri Í, et al. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider[C]//2020 USENIX Annual Technical Conference (USENIX ATC 20). 2020: 205-218.

[3] Zhao N, Albahar H, Abraham S, et al. {DupHunter}: Flexible {High-Performance} Deduplication for Docker Registries[C]//2020 USENIX Annual Technical Conference (USENIX ATC 20). 2020: 769-783.

[4] DADI: Block-Level Image Service for Agile and Elastic Application Deployment

[5] Li Z, Guo L, Chen Q, et al. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through {Inter-Function} Container Sharing[C]//2022 USENIX Annual Technical Conference (USENIX ATC 22). 2022: 69-84.