基于 Pagurus 的热启动策略研究

摘要

在无服务器计算中，应用程序会被拆解为一个或多个细粒度的函数，当函数收到调用时，将被分配到一个容器（或虚拟机）中执行。而函数调用常常会遭遇冷启动，现有的许多相关工作针对缓解冷启动进行了深入研究，如通过维护一个固定大小的专属预热容器池或基于模板的共享预热容器池减少冷启动的影响。但上述方法或可扩展性差，或耗费大量的系统资源。为此，论文《Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing》[8] 中提出了名为 Pagurus 的基于共享的容器调度管理策略，让空闲的热容器可以被其他函数使用，而不是到期就立即回收，从而缓解无服务器计算中的冷启动问题，其实验结果表明，与其他策略相比，Pagurus 对缓解不同函数的冷启动、启动延迟和端到端延迟都有较好的效果。本文在该文章的基础上对 AWS 和 Microsoft Azure Functions 的数据上进行了复现实验，同时改进 SF-WRS 算法，缓解了 Pagurus 高内存占用率的问题。

关键词：无服务器计算；容器共享；冷启动

1 引言

无服务器计算 (Serverless Computing) 是一种新兴和流行的云计算模型。其中应用程序通过用户定义的“函数”执行应用程序代码来使用云资源。与传统的云计算服务相比，用户不用显式地提供或管理云资源，只需将函数的代码传到云端，当事件被“触发”或者“调用”时，函数就会被执行。无服务器计算模型使得开发者专注于应用程序代码的编写，而不需要关注服务器的配置和管理工作。随后，无服务器平台将自动处理所有服务器的配置，部署，扩展和维护任务。逐渐成熟的云计算市场和不断增长的用户微服务需求，使得无服务器计算研究在国内外工业界和学术界以及公共云发展中逐渐得到了广泛的关注和应用。很多知名的云计算服务提供商已经推出了相应的服务，如微软 Azure 的 Functions，谷歌云、亚马逊 AWS 的 Lambda、阿里云，腾讯云以及华为云等云服务提供商相继开展了无服务器计算服务的开发与推广。无服务器计算在移动应用，事件驱动应用，数据处理和分析，物联网以及微服务等众多领域都有着广泛的应用场景。无服务器计算可以用于搭建 Web 应用程序，如网站、电子商务平台等。而在事件驱动的应用程序中，无服务器计算可以用于搭建事件驱动的应用程序，如消息队列，日志处理等。此外，无服务器计算可以用于搭建数据处理和分析平台，如数据挖掘，机器学习等。最后，在微服务方向，无服务器计算可以用于搭建微服务架构，如 API 网关，服务发现等。

无服务器计算主要由事件源，函数实例，FaaS 控制器以及平台服务等元素组成，具体的核心组件图如下图1所示。

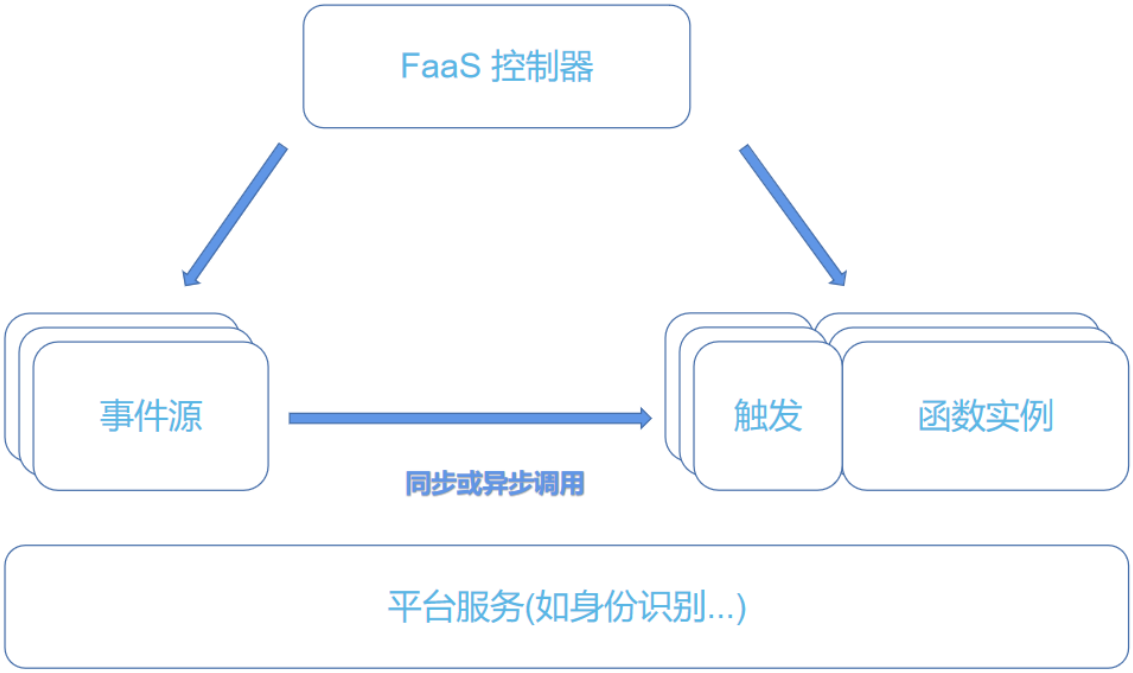

图 1. 无服务器计算组件图

- 事件源（Event sources）：将事件触发或流式传输到一个或多个函数实例中
- 函数实例（Function instances）：一个可以随需扩展的单一的函数或微服务
- FaaS 控制器（FaaS Controller）：部署，监控以及控制函数来源及其实例
- 平台服务（Platform services）：FaaS 平台使用的通用集群或云服务

FaaS 控制器监控控制着事件源和函数实例，事件源可以通过 API/SDK 同步或者异步调用函数实例。平台服务可以为用户提供数据和身份验证等功能。其中函数实例（也称用户函数）是开发人员事先编写的代码，能够在事件触发时自动运行。用户函数可以执行多种任务如调用其他服务，处理数据以及生成响应等。用户函数的具体工作流程如下：开发人员首先编写用户函数代码，将代码上传至无服务器计算平台；用户函数随时等待事件触发；当外界有请求触发事件时，用户函数自动被调用，同时平台会自动分配资源给函数；用户函数执行相应的任务并生成结果数据；无服务器计算平台将相应数据返回给请求方，函数执行完毕，平台自动释放资源。

由于无服务器计算的运行环境采用按需启动原则并且会根据流量的变化自动缩放和扩展，如果没有外界请求时，则会处于休眠状态，需要一定的时间来启动。因此当一个函数被触发时，如果其应用程序尚未被加载到内存中，此时就会遭遇冷启动。冷启动是指一个新的函数实例的过程，需要启动和初始化函数的运行环境，加载运行所需要的库和调度资源等，存在着一定的延迟和性能损失，延迟通常是数秒甚至数十秒，这直接降低了项目的性能，不仅影响用户的体验且对一些时延敏感的项目而言是致命的。FaaS 中的一些应用程序的冷启动开销与程序运行时间对比如下表1所示。

可以看到，冷启动开销可以高达总运行时间的 80%。降低冷启动开销是无服务器计算中的一个重要挑战，为降低冷启动带来的时延影响，减少冷启动次数，可以将一些热启动策略

表 1. FaaS 应用程序冷启动开销和程序运行时间 [4]

Application	Mem size	Run time	Init. time
ML Inference (CNN)	512 MB	6.5 s	4.5 s
Video Encoding	500 MB	56 s	3 s
Matrix Multiply	256 MB	2.5 s	2.2 s
Disk-bench (dd)	256 MB	2.2 s	1.8 s
Web-serving	64 MB	2.4 s	2 s
Floating Point	128 MB	2 s	1.7 s

应用于无服务计算系统中，如使用预热函数提前预热容器，使用缓存，延迟关闭容器，使用快速启动技术以及在闲置时间内执行定期预热等。

而 Pagurus 中则提出一种容器共享的思想，容器共享可以理解为：函数 A 在完成调用后，不立即销毁其使用的容器，而是可以在另一个函数 B 的调用到达时出借给函数 B，即容器不再专用于某一函数，不同函数之间可使用同一个容器来缓解冷启动。通过这种方法，可以在闲置的容器被系统回收之前，利用它来帮助那些可能会经历冷启动的函数。

2 相关工作

如果第一次调用某个函数或者没有为其提供活动（或热）容器，则无服务器系统会启动一个新容器来封装其函数运行时、初始化软件环境、加载特定于应用程序的代码并运行该函数。所有这些步骤构成了冷启动，甚至可能需要几秒钟。冷启动显著增加了查询的端到端延迟 [2]。当函数调用很短（例如数百毫秒）时，这个问题就十分尖锐。

预热启动会生成已使用软件环境初始化的模板容器。尽管它跳过容器启动并且用户只需要执行特定于应用程序的代码初始化 [5]，但其预加载的包可能会使镜像太大，或者导致预热容器消耗更多内存。之前已经进行了许多研究来减少容器启动延迟。[1], [3]，然而，现有的工作主要集中在寻求轻量级虚拟化技术以追求更低的开销 [7] 或优化预热策略以获得更准确的预测模型和更少的初始化成本，常见的优化是在空闲时暂停容器，以节省功能代码和包消耗的资源，然后在调用时重新加载以供重用 [6]。

SAND [1] 允许一个应用程序的功能通过不同进程在同一容器中运行，使用应用程序级沙箱来防止应用程序中后续函数调用的冷启动延迟。Shahrad et al. [10] 提出根据时间序列预测动态改变回收和配置实例的实例寿命。FaasCache [4] 将对象的缓存模型引入到无服务器上下文中，并实现了贪婪双保活缓存机制，以减少资源需求并保持容器温暖。SOCK [9] 建议通过对库集的智能缓存和对函数使用轻量级隔离机制来优化 OpenLambda 中 Python 函数的加载，引入了树缓存，并使用收益成本模型动态更新预热容器中的包。而对于 Pagurus 来说，zygote 设计在通过缓存树维护包时会消耗更多内存。此外，如果函数需要包版本冲突了，缓存树将不起作用。一些研究人员使用 C/R（检查点和恢复）从检查点恢复容器镜像以加速冷启动。例如，Catalyzer [3] 利用 C/R 来实现按需恢复。然而，与热启动相比，它仍然会产生较长的延迟。以上这些技术是与复现论文中的内容是相关的，Pagurus 可以与它们结合以进一步减少冷启动延迟。

3 本文方法

3.1 本文方法概述

Pagurus 的设计核心是在确保函数自身容器需求得到基本满足的同时，将函数后续大概率不再用到的热容器所占内存让出，用共享容器替代空闲热容器并安全地共享给其他函数，以缓解其他函数冷启动。基于共享概念，Pagurus 把一个函数的容器分为三种类型。第一种为 private 容器，仅可以被该函数自身使用，并且在使用时已经加载好函数运行所需的环境。第二种为 zygote 容器，包含了所有待帮助函数所需的共享包以及对应的私有包信息，是可以被其他函数经过复制后安装所需的私有包及导入代码所使用的新容器，也可以通过安装自己所需的剩余包和导入代码供自身使用。第三种则为 helper 容器，它经其他函数的 zygote 容器复制并导入该函数自身所需要的剩余私有包等信息转变而来，相当于是其他函数共享帮助自己的容器。helper 容器中已经加载完毕了该函数的代码和运行时所需的所有包，可以直接用于将来的函数调用。三种容器的示意图如下图2所示。

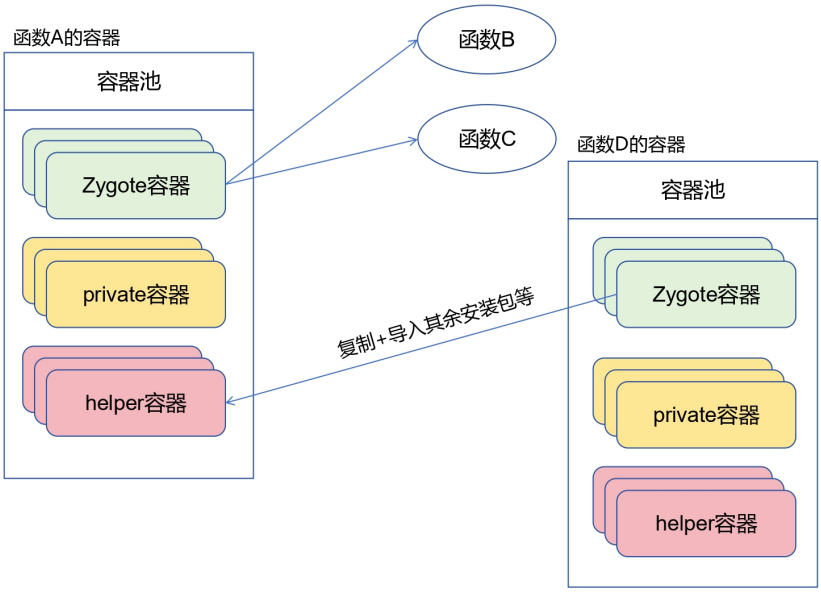

图 2. Pagurus 的三种容器

3.2 Paguru 整体设计

为了管理函数内部容器和函数间的容器共享，Pagurus 为每个函数提供了一个函数内部容器管理器来管理函数自身的容器，同时在节点上提供一个函数间的容器调度器来管理不同函数之间的容器共享操作。对于一个函数，其函数内部管理器负责当有函数调用请求时，选择合适的容器进行函数调用，监控容器池中每个容器的状态并识别出空闲的热容器，同时用 zygote 容器替代空闲热容器。而函数间的容器调度器主要负责通过相似性过滤加权随机抽样（SF-WRS）算法来确定每个函数合适的待帮助函数集合。Pagurus 的整体设计图如图3

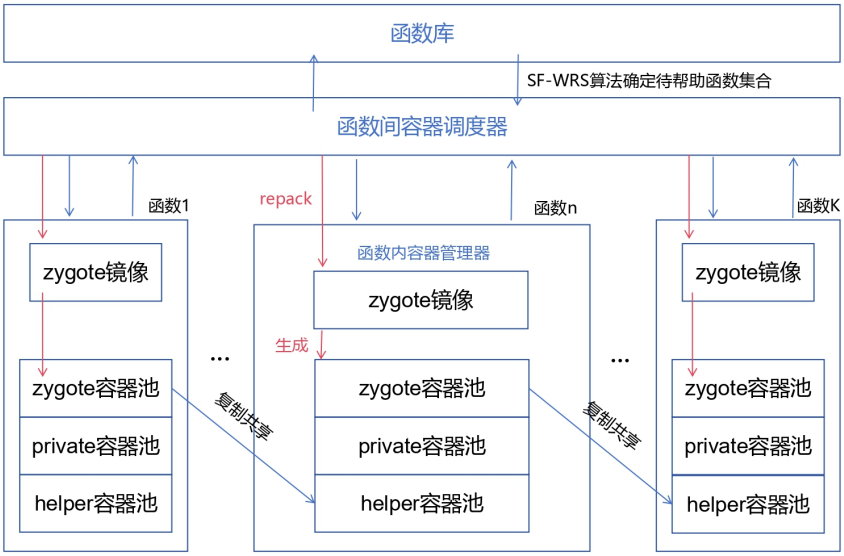

图 3. Pagurus 整体设计

一个函数调用到达时，可以使用自己的 private 容器，helper 容器，自己的 zygote 容器以及从可以其他可帮助它的函数获得 zygote 容器复制而来的容器。由于 private 容器仅供函数自身使用且已经加载好函数运行所需的环境可以直接使用，因此应该作为首要选择。然后，由于 helper 容器是从其他函数的 zygote 容器复制而来，也已经具备了完整的函数运行环境，可以作为第二选择。进一步的，如果自己的 zygote 容器池中有容器，可以使用该容器并安装自己的私有包和导入代码，加载后供承载调用。最后，可以向其他函数寻求帮助，借助可以帮助自己的函数的 zygote 容器来完成此次调用。基于上述的容器划分方式，在 Pagurus 的容器策略下，当一个函数 f 的调用到达时，其将通过如下四个步骤获取一个容器来执行此次调用。

- 函数首先判断是否可以直接从自己的 private 容器池中获得一个空闲的暖容器。如果有空闲的热容器，则直接使用容器，无需经历冷启动过程。如果没有空闲热容器或私有容器为空则检查自己的 helper 容器池是否有可用的热容器。
- 如果 helper 容器池中获取到空闲热容器，则可直接使用，同样无需经历冷启动过程。否则会继续检查 zygote 容器池中是否有 zygote 容器。如果有 zygote 容器，则可以通过安装私有包和导入代码来执行此次调用。
- 如果 zygote 容器池是空的，Pagurus 系统将通过函数间调度器去可以为函数 f 提供帮助的函数处复制一个 zygote 容器，并导入函数 f 的私有包和代码，作为函数 f 的 helper 容器使用，使用完毕后放回函数 f 的 helper 容器池中。
- 若上述步骤无法获取容器时，此时必须经历冷启动获取一个新容器。

3.3 容器内函数管理器

函数内容器管理器的作用是为函数调用选择合适的容器，识别出函数内空闲的热容器并用 zygote 容器替代它，同时确保 zygote 容器共享时的安全性和隐私性。

3.3.1 识别空闲热容器

如果一个容器长时间不执行函数调用，即定义其为空闲容器。通过为每个容器设置一个计时器测量其空闲的时间，来判断一个容器是否为空闲容器。如果某热容器的计时器超过设定的阈值 $T_{idle(f)}$，即可视其处于空闲状态。一旦函数调用到达，容器被重新唤醒则会重置其计时器。由于不同函数具有不同的调用频率和调用模式，应该设置对应不同的时间阈值。同时由于函数在不同时间段内的调用频率可能也会发生变化，因此需要定期的更新时间阈值。一个程序运行时，由于函数的调用存在时间局部性，因此下一次是否有可能被调用会受之前调用的影响。对于调用较为频繁的函数，可以选择先前某一次较大的调用时间间隔作为本次的阈值。而对于调用较少的函数，可以使阈值稍大，使得几乎所有的函数调用都可以获得热容器。统计前一个周期内该函数的所有 m 次调用，相邻调用之间的时间间隔（按升序排序）依次记录为 $T_1, T_2, ..., T_m$。$T_{idle(f)}$ 的计算方式如 (1):

$$T_{idle(f)} = \begin{cases} T_{\lceil 0.95m \rceil} & m \leq 30 \\ T_{default} & \text{m} > 30 \end{cases} \tag{1}$$

其中 m 为此刻的前一段时间内该函数被调用的总次数，$T_{\lceil 0.95m \rceil}$ 指 m 个时间间隔升序排序后的第 0.95m 个向上取整的间隔，$T_{default}$ 设置为 60s。依据公式 (1)，对于频繁调用的函数(这里将调用总次数的情况视为频繁调用)，由于大约超过 95% 的函数调用间隔没有超过设定的空闲阈值，热容器还未被 zygote 容器取代共享给其他函数，因此有超过 95% 的函数调用能够获得热容器。对于较少调用的函数，将阈值 $T_{idle(f)}$ 设置为固定值，使得几乎每次调用都可以获得热容器，无需经历冷启动过程。

3.3.2 zygote 容器的安全性和隐私性管理

当识别出某函数存在空闲的热容器时，该空闲热容器中可能仍然驻留着该函数的数据和代码，处于数据安全性的考虑，一个函数的空闲容器不能被其他函数直接使用。为此，Pagurus 创建了一种不包括其任何代码或数据的 zygote 容器。zygote 容器只包括待帮助函数的共享包以及各函数对应私有包的信息，使用该 zygote 容器替代识别出的空闲热容器，为帮助其他函数提供可能。然而将容器共享给其他函数存在着一定的安全隐患，不同待帮助函数使用共享容器，对应的私有包信息也需要合理的管理。为了确保容器共享时的安全性，通过在操作系统中使用特权控制，待帮助函数的私有包信息安装在该容器所在的目录下，且每个待帮助函数对应自己的私有包信息目录，借助 zygote 容器的函数不能获得其他函数的任何包信息、代码。基于 Linux 操作系统的特权控制，每个函数只能访问自己的软件包信息。具体的避免 zygote 容器中函数包信息泄露的方法如下图4所示。

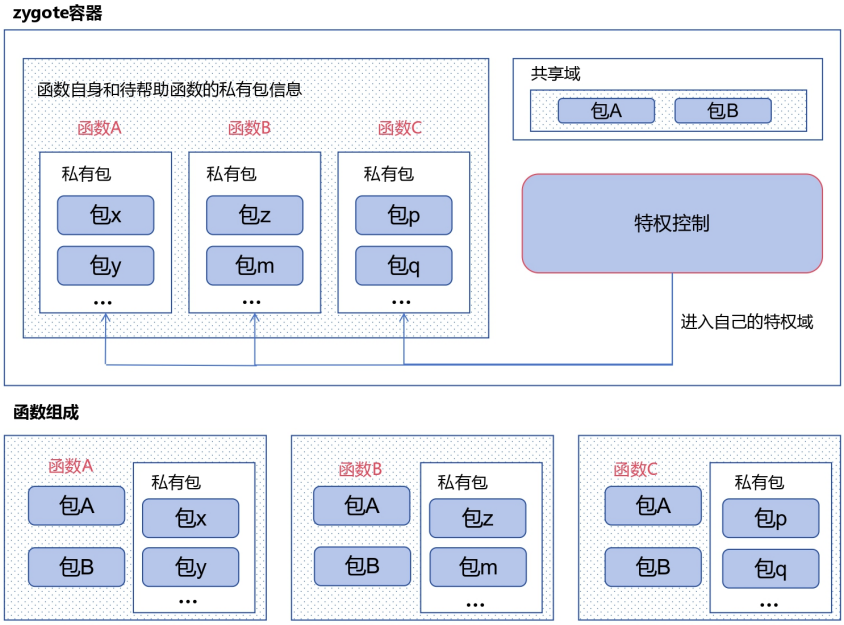

图 4. zygote 容器安全管理

函数 B 和函数 C 是函数 A 的待帮助函数，函数 A 的 zygote 容器共享域中则包含了函数 A，B，C 三者的共享包包 A, 包 B。待帮助函数的私有包信息等缓存在 zygote 容器中的不同目录下，所有函数都作为非根用户运行，确保函数无法访问其他函数的目录内容。每个可能使用 zygote 容器的函数都拥有自己的特权域并且只被允许进入其对应的包目录，当函数 A 获得 zygote 容器时，只能进入自己的特权域获取自己所需的包信息（包 x 和包 y 等私有包），然后通过安装自己的私有包和导入自己的代码加载环境运行。由于特权域的存在，一个 zygote 容器可被视作为安全的。

3.4 函数间容器调度器

函数间容器调度器的功能主要是确定函数内合理的容器共享机制，为每个函数的 zygote 容器确定待帮助函数集合，重新打包 zygote 镜像并返回给函数内容器管理器用于生成 zygote 容器，同时将函数的 zygote 容器复制到待帮助函数的 helper 容器池进行管理。

3.4.1 SF-WRS 算法确定待帮助函数

用函数的 zygote 容器帮助其他函数时，最重要的就是确定函数该帮助的函数有哪些。在确定待帮助函数时，最简单的方法就是将所有其他函数都作为待帮助函数，那么就需要将所有函数所需的所有包都安装到 zygote 容器中，但是这需要花费极大的内存，zygote 容器也会变得异常大，这不可取的。在注重时间和资源消耗的同时，从 Azure 函数追踪中可以观察到一些函数往往比其他函数有更多的冷启动次数，如果选择的待帮助函数是很少被调用的，那么极有可能使得共享机制引起的冷启动开销降低几乎没有效果。考虑包的相似性，在选择待帮助函数时，如果有函数和自身运行时所需的包有很多重复的，那么就可以在 zygote 容器中安装共享包，同时待帮助函数使用 zygote 这个共享容器时也只需经历短暂的加载剩余包就可以使用。论文提出了一个 SF-WRS 算法，该算法主要由基于相似性的过滤器和 A-ExpJ 算法组成。SF-WRS 算法的核心工作就是在基于相似性的过滤器为函数 f 确定相似度达标且不含

包冲突的函数集合上，根据集合中每个函数的冷启动频率利用 A-ExpJ 算法选择具体的 K 个待帮助函数并确定为函数 f 的最终待帮助函数集合。

Similarity-based Filter, 将函数 f 视为其所有包的集合，即 f=$\{pkg_1, pkg_2,...\}$。其余函数集合为 F=$\{f_1, f_2,...f_k\}$ 。对于函数 f 和 f_i ($f_i \in$ F) 的所有包并集中的某一个包 pkg, 将 pkg 与两个函数的包含关系表示如式 (2)：

$$con(pkg, f) = \begin{cases} 1 & \textbf{if} pkg \in f \\ 0 & \textbf{others} \end{cases} \forall pkg \in f \cup f_i \tag{2}$$

定义函数 f 和 f_i ($f_i \in$ F) 与并集中所有包的关系向量分别为

$$\vec{f} = con(pkg, f) \forall pkg \in f \cup f_i \tag{3}$$

$$\vec{f_i} = con(pkg, f_i) \forall pkg \in f \cup f_i \tag{4}$$

函数 f 和 f_i ($f_i \in$ F) 的相似度可以用以上两个向量的余弦距离表示，如式5

$$con(\vec{f}, \vec{f_i}) = \begin{cases} \frac{\vec{f} \cdot \vec{f_i}}{\|\vec{f}\| \cdot \|\vec{f_i}\|} & \|\vec{f}\| \cdot \|\vec{f_i}\| \neq 0 \\ 1 & \|\vec{f}\| \cdot \|\vec{f_i}\| = 0 \end{cases} \forall f_i \in F \tag{5}$$

其中，$\| \vec{f} \|$ 和 $\| \vec{f_i} \|$ 分别是向量 $\vec{f}$ 和 $\vec{f_i}$ 的模长，$\vec{f} \cdot \vec{f_i}$ 是两个向量的乘积。

然而，函数之间的包可能有不同的版本，上面得到的初始待帮助函数集合仍存在着包冲突的可能性。具体如下图5所示。

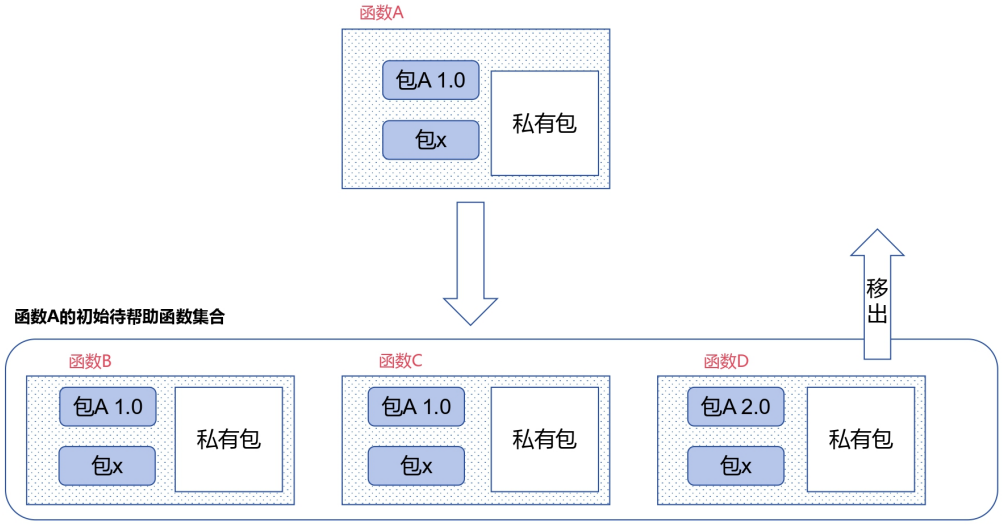

图 5. 包冲突

函数 A 确定的初始待帮助函数集合为函数 B，C 和 D，四者的函数共享包为包 A 和包 X。但是由于函数 D 与其他三者的包 A 版本并不相同，因此是无法共存的，存在包冲突矛盾。因此在上述确定的初始待帮助函数集合中还需要根据包的版本移除部分含有冲突包的函数。

用 $conflict(f_1, f_2)$ 表示两个函数之间是否存在包冲突。具体来说，定义如果两个函数 f_1 和 f_2 之间没有版本冲突的包，则两个函数相互兼容，即 $conflict(f_1, f_2)$=1；反之则 $conflict(f_1, f_2)$=0。对于函数 f 和其初始待帮助函数集合中的所有函数，需要确保所有函数任意两两之间都不存在包冲突问题，令除去包冲突函数后的函数 f 的待帮助函数集合为 C_f，则该集合应该满足如下条件：

$$conflict(f_i, f_j) = 0, \forall f_i, f_j \in C_f \cup f \tag{6}$$

为防止存在部分函数的待帮助函数过多，使得 zygote 映像过大或 zygote 重包装时间过长，需进一步限制每个函数的待帮助函数在阈值 K 个以内，并采用 WRS 随机抽样方法进行选择。

3.5 重打包 zygote 映像

zygote 镜像中包含待帮助函数的共享包及对应的私有包信息。zygote 容器就是依据 zygote 镜像生成的。确定函数的待帮助函数集合后，就需要将待帮助函数的共享包及各函数对应的私有包信息装入 zygote 镜像中，便于后续变成 zygote 容器帮助其他函数。这一过程称之为重新打包 zygote 镜像。zygote 映像重新打包后，需要将其返回给函数内容器管理器，再根据镜像内容生成 zygote 容器。其流程图如6所示

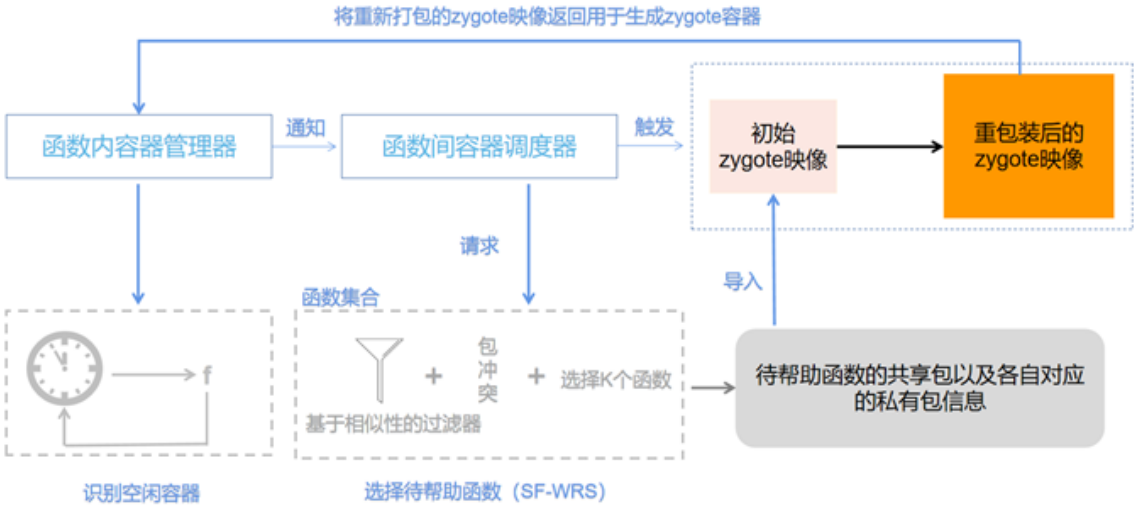

图 6. 重打包镜像

3.6 复制 zygote 容器到待帮助函数的 helper 容器池

在复制函数 f 的 zygote 容器到待帮助函数 p 的 helper 容器池中时，首先经历 zygote 容器的复制。复制的具体过程为，从原函数 f 的 zygote 池中复制得到一个容器 Con，然后卸载容器 Con 中除了函数 p 以外的函数包目录，并将控制访问转移到 p 对应的特权域。成功复制 zygote 容器后，将函数 p 的代码复制到容器中，最后再安装 p 所需的私有包，供函数 p 承载此次调用。调用结束后，再将容器放入到函数 p 的 helper 容器池中用于后续使用，如图7

表 2. 实验配置

CPU	AMD Ryzen 5800H with Radeon Graphics
Cores	8
Disk	100GB
Memory	16GB
Operating system	Ubuntu 20.04.4 LTS
Docker	23.0.1
Python	Python 3.11.4

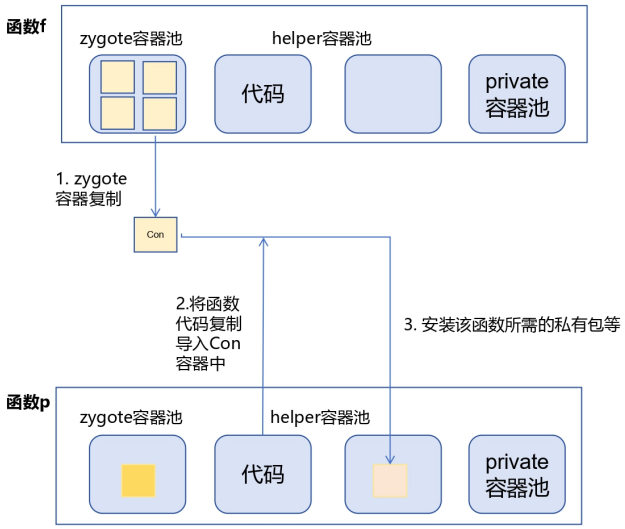

图 7. 复制 zygote 容器

4 复现细节

4.1 与已有开源代码对比

参考了开源代码。在原有代码的基础上增加了运行时内存数据检测部分，并对文中 SF-WRS 中集合相似性度量方法进行了修改，并与原有代码的运行结果进行对比。

4.2 实验环境搭建

实验需要用到 8 个 cpu 核和 16GB 内存的配置，由于这个配置的云服务器价格高昂，因此在本地主机安装了 linux 系统进行测试，具体配置见表2。

衡量指标包括冷启动次数、启动延迟和端到端延迟。

4.3 创新点

补充了无服务器计算容器调度策略的重要测试指标：内存占用。动态监测了两个数据集的在执行过程中的内存使用情况。另外，对论文中 SF-WRS 算法的函数依赖包集合的相似性度量的方法进行修改，并将修改的方案得到的结果与原来方案的结果进行对比，发现性能得到了提升。

5 实验结果分析

5.1 AWS 小规模数据测试

实验数据采用 AWS 样本数据中具有最多 github stars 的 10 个应用程序的真实数据，同时为避免偶然性，选择 18 组不同的样本进行测试，并与启用预热的 OpenWhisk、禁用预热的 OpenWhisk、SOCK 等策略进行对比。

冷启动次数, 如图8，与禁用预热的 OpenWhisk 相比，Pagurus 缓解了大约 59.5% 的冷启动，SOCK 与启用预热的 OpenWhisk 分别缓解了大约 25.4% 和 22.8% 的冷启动。

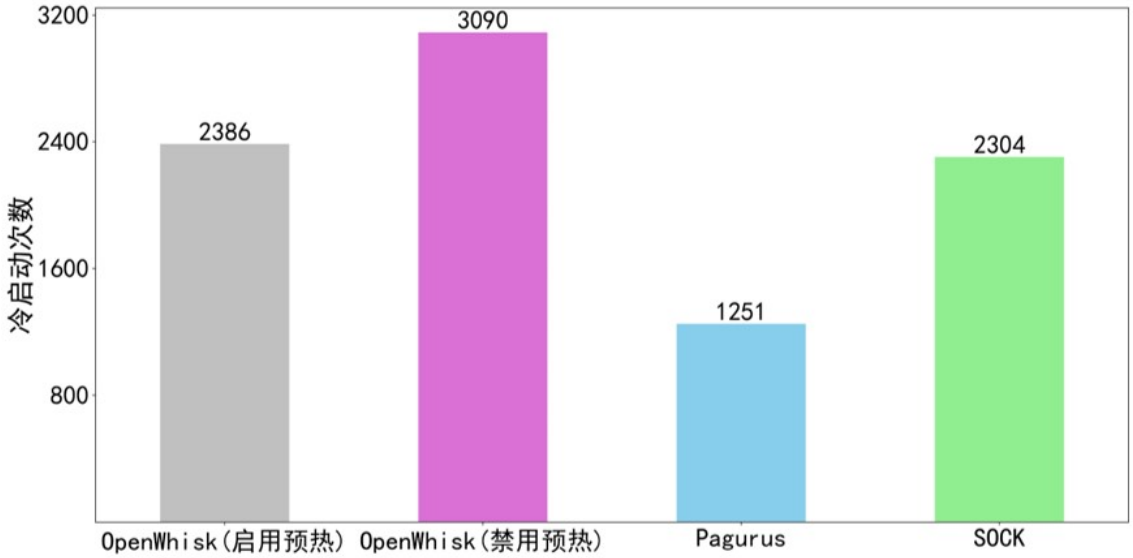

图 8. 冷启动次数

如图9, 对于大多函数，SOCK 和启用预热的 OpenWhisk 只能减少较少比例的冷启动，而 Pagurus 有比较显著的冷启动缓解效果。对于少量如 eco 应用程序中的某些函数，Pagurus 缓解冷启动效果比 SOCK 差些，这是由于这些函数本身具有比较低的冷启动频率，而我们的 SF-WRS 策略倾向于将冷启动频率较高的函数作为待帮助函数以缓解它们的冷启动，因此效果偏差.

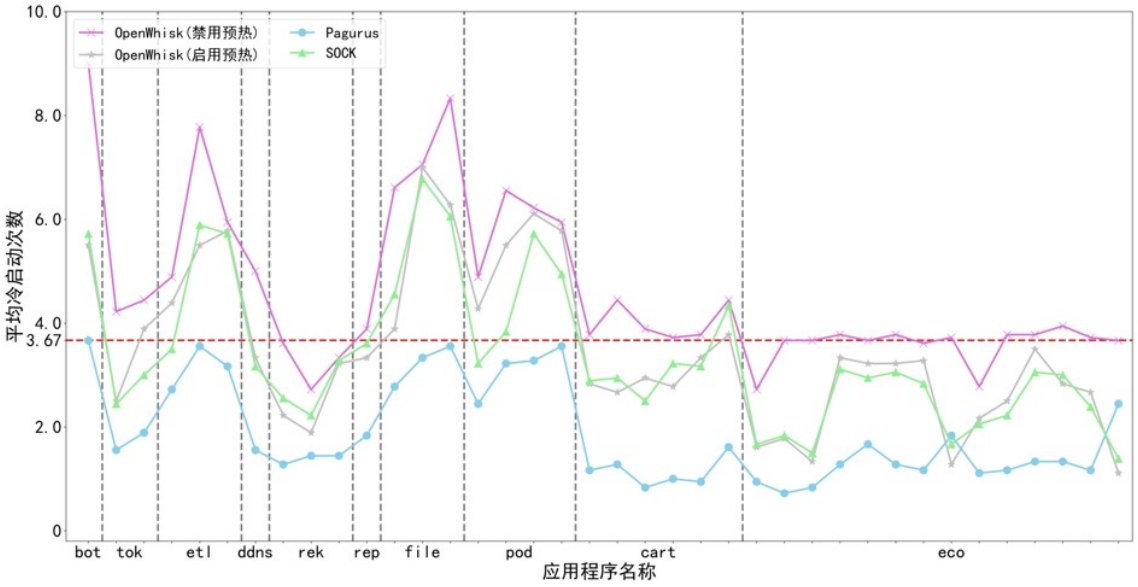

图 9. 单个函数的冷启动次数

启动延迟, 如图10对于不同函数的调用, Pagurus 的启动延迟普遍是偏低的。由于 Pagurus 系统中，如果函数调用使用的容器是自身的 private 或者 helper 容器，由于包和函数代码已经导入，可以直接获得一个容器运行。而禁用以及启用预热的 OpenWhisk 还需要导入函数的所有包，SOCK 对于大多数函数也需要导入部分包，因此三者启动延迟偏高。

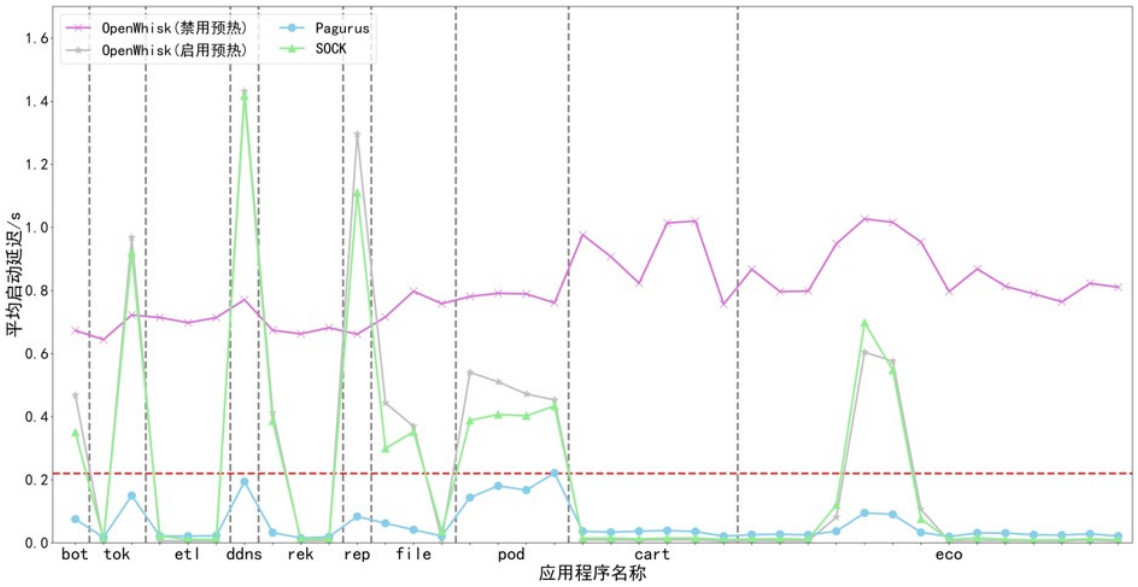

图 10. 单个函数的启动延迟

端到端延迟, 如图11，Pagurus 系统的平均端到端延迟是最低的，与禁用预热的 OpenWhisk 相比，Pagurus 系统大约降低了 79.6%。

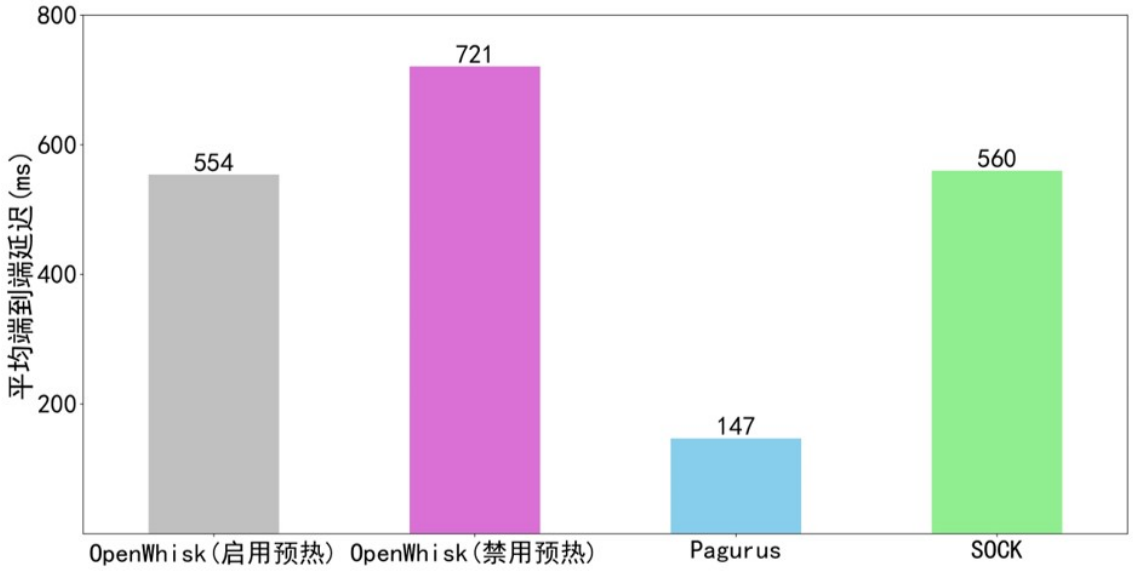

图 11. 端到端延迟

对于单个函数，如图12，Pagurus 的端到端延迟对于不同函数都是最低的，几乎所有函数的端到端延迟大小都在 0.68s 以下，对于大多函数的函数调用都有较为理想的响应速度.

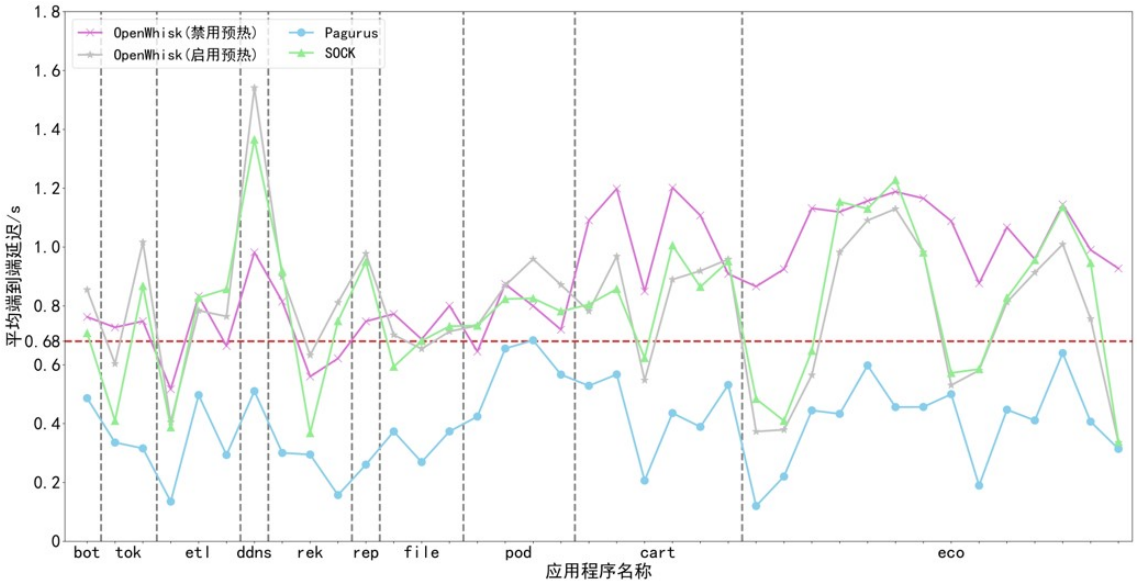

图 12. 单个函数的端到端延迟

5.2 Azure 大规模数据测试

随机选取了微软 Azure Trace 中某一天的数据中约 3000 个函数，并将其随机分成 8 个小规模的数据子集分别进行测试，其中每个子集的测试时间约为 16h，最后将各子集运行结果汇总。

冷启动次数，如图13，基于测试的数据集，Pagurus 和 OpenWhisk 的冷启动总次数为 1596 和 9289，相比 OpenWhisk，Pagurus 将冷启动次数降低了约 82.8%。禁用预热的 OpenWhisk 导致了调用频率适中的函数频繁出现冷启动。

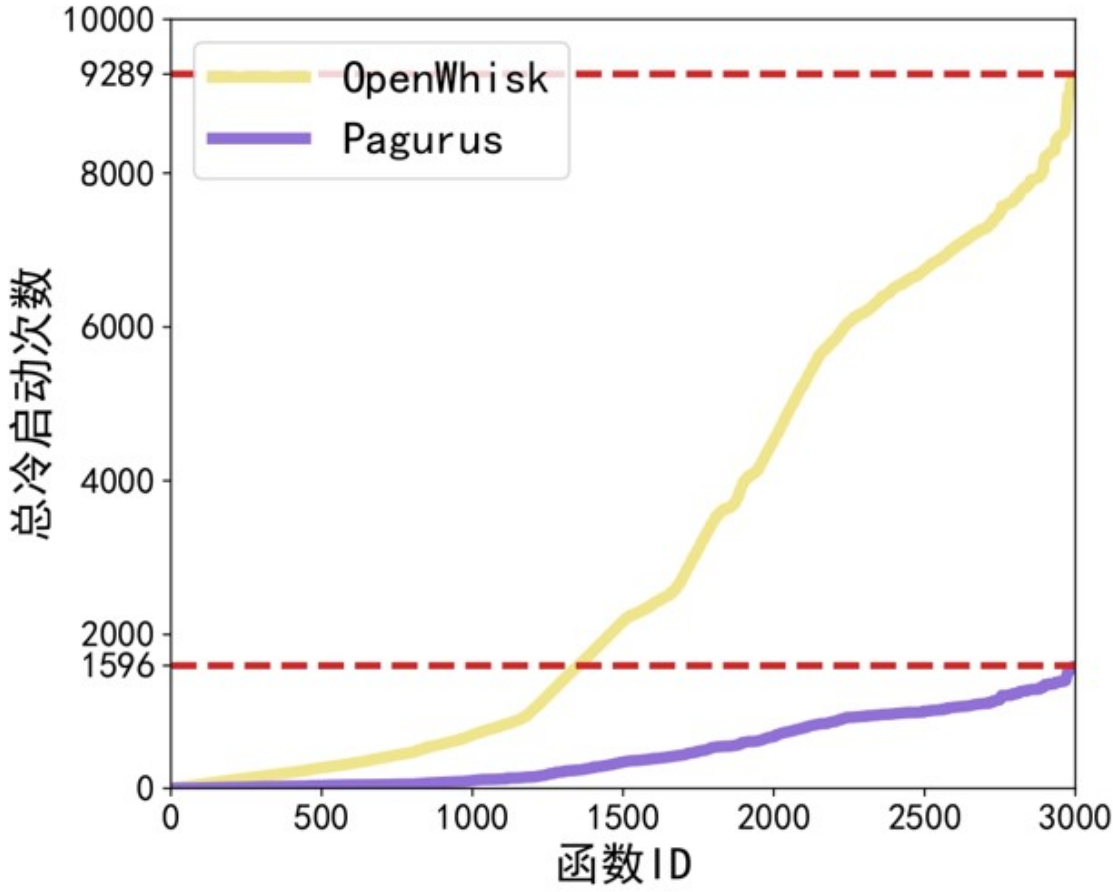

图 13. 冷起动次数

端到端 95% 尾延迟, 如图14，3000 个函数中绝大多数函数的 95%-尾延迟在 6 秒以下，只有少数高于 6 秒。OpenWhisk 中大多数函数的 95%-尾延迟比 Pagurus 的要长.

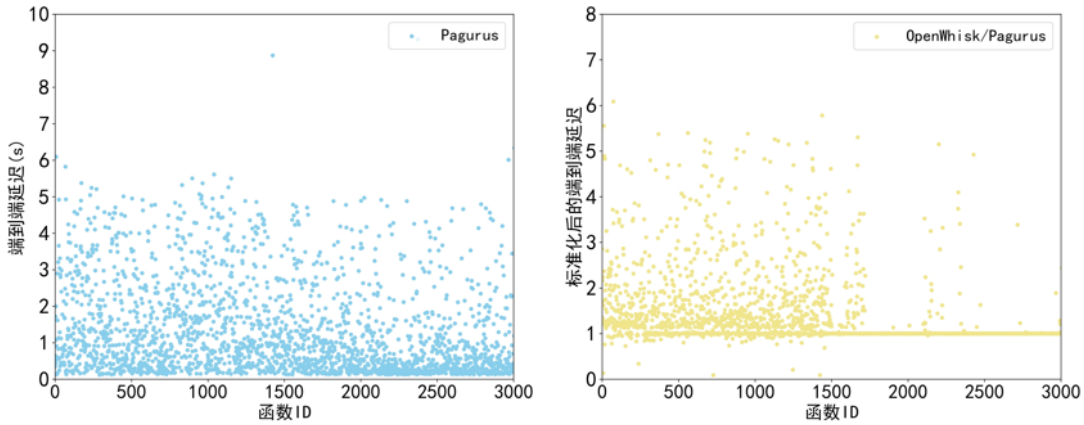

图 14. 尾延迟

5.3 Pagurus 策略的内存问题

对 Pagurus 和 OpenWhisk 运行时的内存使用率进行动态实时检测, 得到如图15和图16的结果对于 AWS 数据集，在运行的两个小时内，Pagurus 的内存占用率总体上在 32% 左右，而 OpenWhisk 则在 18% 左右，Pagurus 的内存使用率比 OpenWhisk 高约 14% 。对于 Azure 数

据集，总体上 Pagurus 的内存使用率在 56% 到 70% 之间波动，而 OpenWhisk 的内存使用率则在 40% 到 54% 之间波动。Pagurus 的内存使用率比 OpenWhisk 高约 16%。

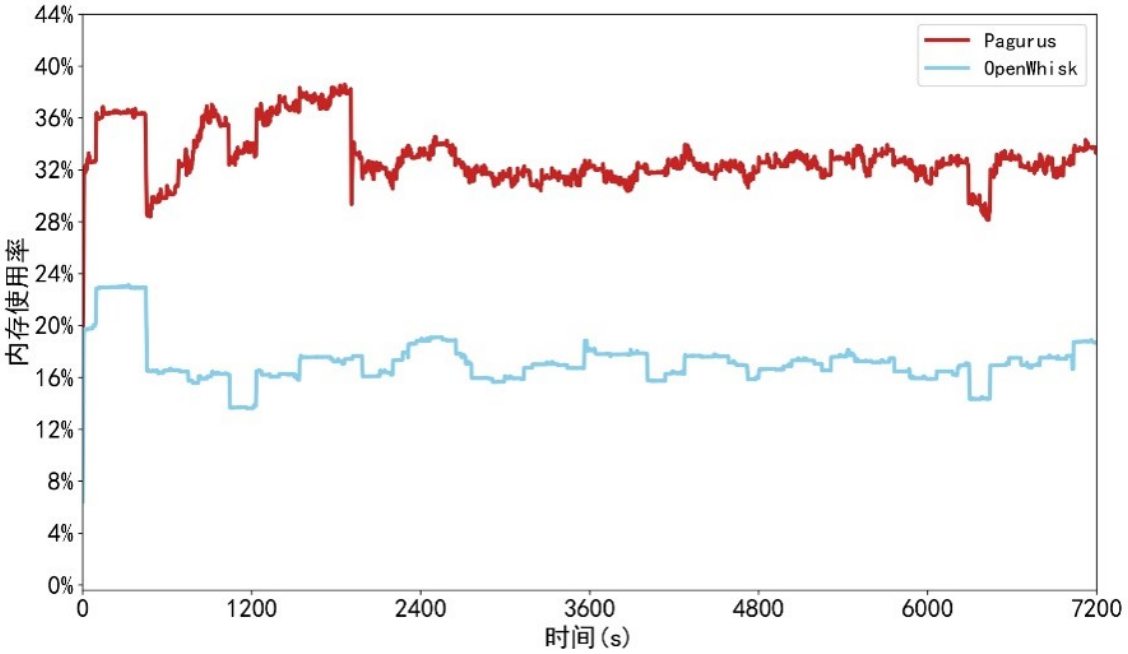

图 15. AWS 内存使用率

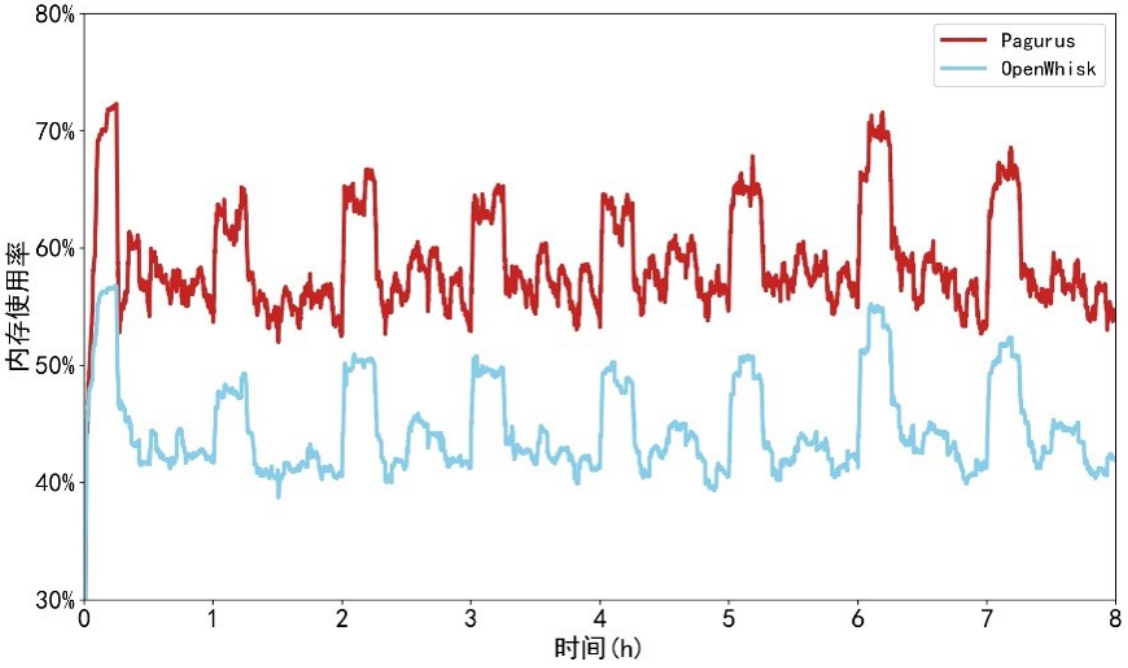

图 16. Azure 内存使用率

5.4 SF 相似度的改进

论文中 Similarity-based Filter 求解不同函数依赖包的相似度采用了余弦相似度的方法，这种方法并不适用于大小差异很大的集合，且其时间复杂度比较高。而求解集合相似性的方法还有 Jaccard 相似度、Dice 系数等。Jaccard 相似度：计算简单，但对小集合不够敏感，函

数 f 和 f_i ($f_i \in$ F) 的相似度可以用式7表示：

$$Jaccard(f, f_i) = \frac{\|f \cap f_i\|}{\|f \cup f_i\|}, \forall f_i \in F \tag{7}$$

Dice 系数：类似于 Jaccard，计算简单，对小集合更敏感，函数 f 和 f_i ($f_i \in$ F) 的相似度可以用式8表示:

$$Dice(f, f_i) = \frac{2 \cdot \|f \cap f_i\|}{\|f\| + \|f_i\|}, \forall f_i \in F \tag{8}$$

分别使用 Jaccard 相似度、Dice 系数应用于 Similarity-based Filter 中的相似度求解，并在 AWS 数据集上进行测试，将得到的结果与论文中使用到的 Cosine 相似度进行对比，如图17，黄色、蓝色、红色曲线分别代表使用 Jaccard 相似度、Dice 系数和 Cosine 相似度来计算函数相似度的在运行过程中的内存使用率曲线。相比之下，Jaccard 和 Dice 的整体内存使用率要低于 Cosine，约低 5% 左右，Jaccard 的效果最好。

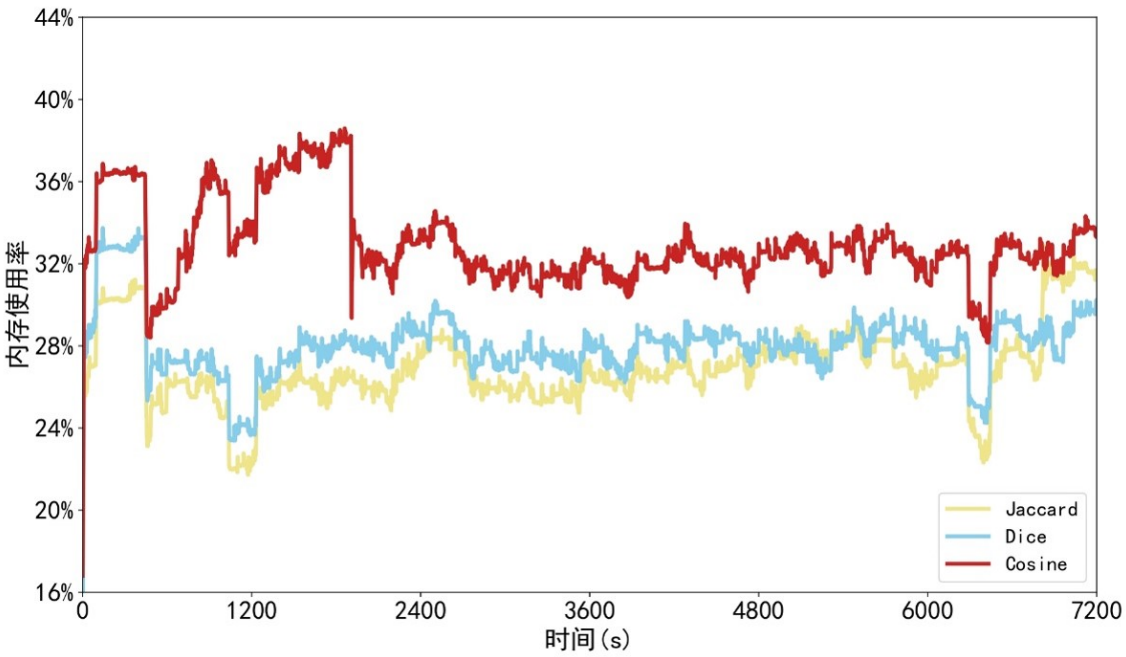

图 17. 不同相似性度量策略对比

6 总结与展望

本次实验，学习了无服务器计算相关知识如无服务器计算概念，模型以及冷启动热启动原理等，对无服务器计算领域有了初步的理解。在了解了常见的无服务器计算平台如 OpenWhisk 以及热启动策略后，通过阅读文献学习了 Pagurus 缓解冷启动的策略核心——容器共享，并对整个实验任务有了大致的掌握。进一步的，在清晰整个系统的架构下，依次掌握了系统内的三种容器的划分依据和用处，函数内容器调度器识别空闲容器并用 zygote 容器替换的实现步骤，函数间调度器如何通过 SF-WRS 算法计算每个函数的待帮助函数集合并实现 zygote 容器共享。同时结合系统源码进行分析，在亚马逊 AWS 样本以及微软的 Azure Functions 数据集对 Pagurus 系统进行了数据测试。实验结果表明，与禁用和启用预热的 OpenWhisk 系

统以及 SOCK 相比，Pagurus 系统对缓解不同函数的冷启动次数都有着较好的效果，在亚马逊 AWS 样本上，Pagurus 系统缓解了大约 92.6% 的冷启动。而在 Azure Functions 数据集上，Pagurus 将冷启动次数降低了约 82.8%。同时，Pagurus 系统的启动延迟，端到端延迟普遍比其他系统偏低，具有较为理想的响应速度和系统性能。然而在经过运行时动态内存监测发现，Pagurus 的内存开销较大，分析可能是由于 Pagurus 将函数容器分为三种进行管理，维护每个函数的三个容器池的内存开销较大。同时由于 zygote 容器是由 zygote 映像创建的，zygote 容器中存储了公共包，而 private 容器和 helper 容器也事先存储了所能承载的对应函数所有包，zygote 映像以及大量提前导入的包也将导致较大的内存占用。因此我调整了 SF-WRS 算法中相似性度量的策略，更换为 Jaccard 和 Dice 进行测试，结果表明 Jaccard 和 Dice 所得到的效果比原本使用 Cosine 的效果要好些。

在未来的工作中，需要寻找更好的缓解冷启动延迟与降低内存占用率之间的权衡方法，在保证内存使用率较低的情况下更好地缓解冷启动。

参考文献

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[2] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

[3] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Alexander Fuerst and Prateek Sharma. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, February 2016. USENIX Association.

[6] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with

OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[7] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, Carlsbad, CA, July 2022. USENIX Association.

[8] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

[9] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[10] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.