基于 serverless 计算环境中的“冷启动问题”和指令预取的相关研究综述

王继昂

(华中科技大学 计算机科学与技术学院 2205 班 M202273733)

摘 要 随着 serverless 计算模型的逐渐成熟，其在云计算中扮演越来越重要的角色。然而，” 冷启动” 问题和指令预取的不足限制了 serverless 计算环境中函数的性能和用户体验。为了解决这些问题，学术界提出了各种方法和技术。本文综述了五篇围绕 serverless 计算环境中的” 冷启动” 问题和指令预取的相关研究论文，其中包括基于函数分发和选择、概率指令缓存管理、基于性能分析的冷启动解决方法、基于深度强化学习的函数调用优化方法以及基于神经网络的函数性能预测方法。这些研究提供了不同的解决方案和思路，从而显著提高了 serverless 计算环境中函数的运行效率和系统的性能。

关键词 serverless 计算模型，冷启动问题，指令预取，函数分发，指令缓存，深度强化学习，神经网络，性能预测

中图法分类号 TP DOI **号**: * 投稿时不提供 DOI 号

1 引言

Serverless 计算是一种新兴的计算模式，以其高效性和灵活性备受关注。然而，在 serverless 计算中，冷启动问题是一个普遍存在的问题，也是当前 serverless 计算面临的最大挑战之一。本文对于冷启动问题的定义和研究背景进行了阐述，并针对现有的相关研究工作进行了综述。[?]

首先，本文介绍了冷启动问题的概念和原因，以及它对于 serverless 计算的影响。然后，本文对当前已经提出的解决冷启动问题的方法进行了分类和总结，包括指令预取、函数预热、函数缓存和预分配实例等。在这些方法中，指令预取被认为是一种有效的解决冷启动问题的方法，它可以通过在函数执行之前预取函数所需的指令，从而提高函数的响应速度。同时，本文也介绍了目前已经提出的一些指令预取算法，包括基于代码依赖关系、基于相似函数和基于频率分析等算法。

接着，本文分别对于 5 篇 2020 年以后的 CCF-A/B 类论文进行了综述。这些论文分别探讨了冷启动问题的不同方面，包括对冷启动问题的调查、函数预热技术的综述、冷启动问题的全面研究、使用函数预热和推测来提高 serverless 计算效率以及利用代码相似性来减轻冷启动问题等。

最后，本文总结了现有研究工作中的不足和未来的研究方向。本文认为，未来的研究工作应该注重针对不同场景的冷启动问题提出更加有效的解决方案，并进一步探索指令预取算法、函数预热算法和函数缓存算法等方法的优化。同时，本文也强调了在解决冷启动问题的同时，需要考虑到安全和隐私等因素，保证 serverless 计算的安全性和可靠性。

2 冷启动问题

无服务器计算的瞬时和无限可伸缩性的代价是不可避免的。同样，在云服务中引入 FaaS 给服务提供商带来了新的挑战。

随着职责的减少，对运行时环境的控制也减少了。功能请求的迅速增加需要一种智能技术来处理传入的请求。无服务器平台在装入内存的轻量级容器中执行函数。包含函数执行所需的所有基本代码和已配置运行时的容器称为函数的实例。任何无服务器函数的初始执行都需要事先有一个可用的容器。此外，无服务器计算的可伸缩性特性提供了保持函数实例随时可用以处理即将到来的请求的期望。如果实例可用，则立即执行函数。否则，要创建一

个新实例，容器需要启动，使用运行时环境初始化，加载依赖项，分配权限和要在其中下载的代码。创建新实例所消耗的额外时间被称为无服务器生态系统的冷启动延迟，这个问题被称为流行的冷启动问题。我们将总冷启动延迟定义为多次冷启动延迟的总和。客户端的假设是使它们的功能随时可用，但是保留实例是一个麻烦，因为可用内存成为数据中心的限制。因此，需要在容器空闲时间和内存之间进行权衡。在为函数请求提供服务后，容器持有客户端代码可以在特定的时间跨度内保持活跃和空闲状态。在本文中，我们把这个时间跨度称为容器升温时间。超过 Warm 时间后，闲置容器将被平台释放，减少资源消耗。这个 Warm time 是特定于平台的。对于大多数平台，温暖时间最多是 20 分钟。空闲容器称为热容器。

因此，热容器是暂停的容器，它们在过去已配置的运行时环境中启动过。这些热容器已经完成了执行阶段，目前处于闲置状态。

3 Reducing cold starts during elastic scaling of containers in Kubernetes

在本节中，我们将介绍三种策略，即使用 (i) 可重用网络容器池，(i) 基于层的库共享，以及 (iii) 应用程序容器的强制配置，以缓解 Kubernetes 中的冷启动问题。

1) **可重用的网络容器**

在这种方法中，我们的目标是预先创建应用程序容器的网络基础设施，以减少冷启动时间。为了实现这一点，可以使用一个热网络容器队列。

当需要向外扩展并且必须启动一个新的应用程序容器时，我们的缩放器将新启动的容器绑定到一个暖网络容器。然后，将网络容器从队列中移除并放置在热队列中，这意味着它已被占用。使用热队列，当不再需要应用程序容器时，可以释放网络容器并将其放回网络容器队列。这确保了可重用性。

在 Kubernetes 中，网络容器 (也称为暂停容器) 不能单独创建。要获得暂停容器队列，我们的方法是基于创建轻量级的 pod。因此，每个 pod 都由一个暂停容器组成。

圆荚体注入。当一个向外扩展请求到达时，一个应用程序容器被注入到一个 pod 中，如下所示: 我们的缩放控制器启动一个应用程序容器; 它从队列中选择一个条目 (即。

该条目包括暂停容器 ID, IPC 命名空间和队列中每个 pod 的 cgroup); 它配置容器使用 pod 的网络命名空间、cgroup 和 IPC 命名空间; 它将 pod 放在热队列中，以知道该 pod 已经在运行一个工作实例。我们称之为 “pod 注入”，因为应用程序容器在不使用 Kubernetes API 的情况下被注入到 pod 中，而是直接与工作节点通信。

2) **基于分层的库共享**

基于层的库共享是一种技术，应用程序的外部组件和软件依赖关系被封装在不同层的容器映像中，以缓解冷启动问题。其他工作表明，库共享通过在容器之间共享公共内存部分来减少内存占用。我们的假设是，基于层的方法对应用程序的启动时间有影响。以往的研究都是利用该技术来减小图像尺寸。这意味着如果一个层已经存在这个库，那么它可以被多个容器使用。这基本上使第二个容器的容器映像 (例如 Docker) 更小，因为它不再需要为它的应用程序提供库; 因此，它减少了冷启动。

编程语言影响应用程序启动时间。例如，Java 应用程序通常在 JAR 文件中包含大多数依赖项。在 JAR 文件中，有应用程序类、库、框架和清单文件。如果我们处理的是一个较大的 JAR 文件，那么我们只有一个层包含整个应用程序。因此，从图书馆共享中获益是不可能的。为了加快容器图像检索，可以将 JAR 文件分割为多个层。

5) Kubernetes **中的必要缩放**

使用 Kubernetes 中的命令式方法，我们的缩放器直接与工作节点通信，绕过 Kubernetes 控制器组件。在声明式方法中，状态更改请求 (例如 pod 的部署) 由 Kubernetes API、各种控制器和调度器处理，以实现控制循环以减轻意外状态 (例如节点或容器崩溃)。而且，豆荚是按顺序启动的，一个接一个地放置。但是，我们的命令式方法直接与容器运行时 (例如 Docker 守护进程) 通信，启动应用程序容器并将它们注入到网络 pod 中。因此，整个过程可以同时并行完成。我们的假设是，当应用程序需要向外扩展时，它会大大提高应用程序的启动时间。

4 Using Application Knowledge to Reduce Cold Starts in FaaS Services

在最近非常流行的功能即服务平台 (FaaS) 中，代码部署在单个功能单元中，云提供商处理资源管理。在这里，一个关键问题是所谓的冷启动问题: 当传入一个请求时，却找不到用于执行目标函数的空闲容器，那么就需要提供一个新的容器。在这种情况下，请求会产生额外的延迟——冷启动延迟。

最近的工作主要集中在减少冷启动的持续时间。在本文中，我们提出了三种与相关工作相辅相成的方法，在将 FaaS 服务视为黑盒的同时减少冷启动的数量。在作为轻量级编排中间件的一部分实

现的方法中，我们使用关于函数组合的知识来触发冷启动，从而在应用程序流程调用各自的函数之前提供新容器。

4.1 The Naive Approach

这种简单方法背后的直觉是，在第一步中遇到冷启动的流程 (在某些假设下) 很可能在其他步骤中也遇到冷启动。这对于偶尔触发的流程来说尤其如此，这样所有的容器都没有准备，但也会发生在请求率增加的阶段。现在的想法是在第一步遇到冷启动时通知所有其他步骤，这样就可以为这些步骤提供额外的容器。由于 FaaS 平台目前没有为应用程序提供直接执行此操作的接口，因此只能通过调用所有其他步骤 (异步) 来实现，即向它们发送提示; 每个步骤在接收到提示消息时立即终止。这种方法可以很容易地实现为第 3 节中的编排中间件的一部分。有关该方法的概述，请参见图 1。

问题: 虽然这种简单的方法实现起来确实非常简单，但是它存在一些问题，使得它的使用受到一些特殊情况以外的限制。主要问题是提示错误和过程超车。另一方面，当步骤 2 的冷启动时间超过步骤 1 的处理时间时，就会发生过程超车。在这种情况下，提示消息可能会触发步骤 2 的冷启动，当步骤 1 终止时，该冷启动仍在进行，并调用步骤 2，然后可能会导致另一个冷启动。

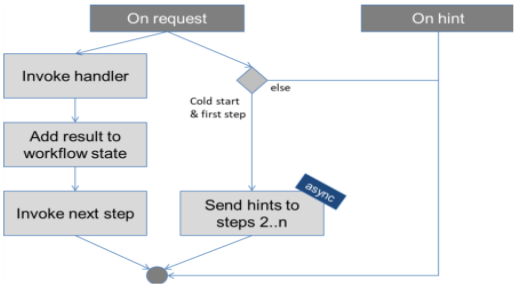

Figure 1: Naive Approach: Control Flow in the Middleware

图 1 Naive Approach: Control Flow in the Middleware

4.2 The extended Approach

通过扩展方法，我们特别致力于解决提示缺失和过程超车问题; 这或多或少是天真方法的改进版本。过程超车最有可能发生在一个过程的第 2 步: 在后面的步骤中，它只可能发生在除了第一个步骤之外的所有前面的步骤都没有遇到冷起步的情况下。因此，处理进程超车的最简单方法是使用简单的方法，但将提示 “同步” 发送到步骤 2，或者更准确地说: 异步发送，但在终止函数之前等待结果。这样，流程执行将等待第 2 步中的 (潜在的) 冷启动完成。

此外，为了解决提示缺失的问题，我们提出了一种称为递归提示的机制: 在收到提示消息时，函数检查它是否有冷启动。如果没有提示，函数将向自身发送提示消息，以在某个时刻 “强制” 冷启动。

在递归提示中，我们已经确定了两种调优机制 (出于可读性原因，图 2 中没有显示)。第一种是最大递归深度，它对递归提示的数量设置了上限，可以通过计数器轻松实现。作为一种安全机制，这方面是至关重要的，特别是与第二调优机制相结合时。

第二种机制描述了如何发送自我提示。

基本上有三种不同的方式: 异步、同步或带超时的同步。图 2 显示了该方法的概述，特别是与图 1 的比较。

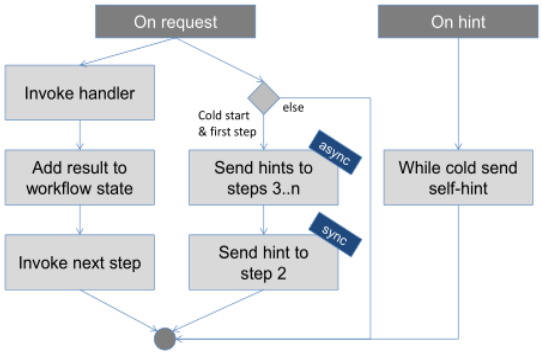

图 2 扩展方法: 中间件中的控制流

问题: 虽然扩展方法解决了朴素方法的两个核心问题，但仍然存在许多问题。

可以说，剩下的主要问题是，这两种方法都需要首先遇到冷启动，以避免其他冷启动，也就是说，它们的主要用途是避免冷启动持续时间在较长的流程执行过程中累积的情况。此外，这两种方法都假设流程中不同步骤的执行持续时间是可比较的，这也意味着每个步骤需要大约相同数量的实例来处理恒定的工作负载。

4.3 The Global Approach

虽然以上至少部分问题可以通过对扩展方法的增量更改来解决，但我们决定开发全球方法，试图通过一种根本不同的机制来解决所有问题。其他两种方法都缺乏关于全局系统状态的知识，但作为交换，它们都是非常轻量级的，并且只是与单个函数一起部署 (我们稍后将详细讨论这三种方法的优缺点)。对于全局方法，我们决定引入提示管理器作为从流程执行中收集监视数据的中心组件，并根据排队理论运行模拟，以确定发送提示的最佳时间点。有关组件、它们的交互以及它们的部署的高级概述，

请参见图 3。

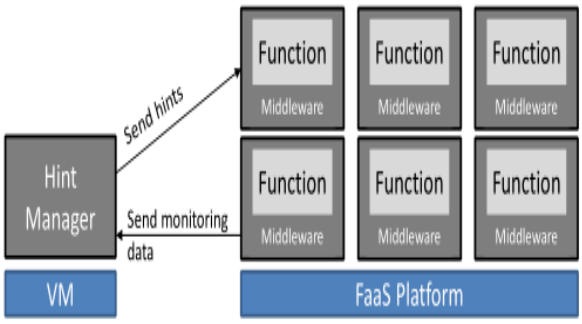

图 3 全局方法的部署体系结构

指数到达率在大多数情况下，例如，在负载增加的情况下，也是不正确的。尽管如此，我们还是决定使用这些假设，因为它们极大地简化了我们的方法，并将微调留给未来的工作。然而，我们谨慎地断言，第 5 节中的实验使用了现实的用例，并且确实明显违反了具有指数分布的执行持续时间的假设。这样做是为了证明这种方法在不利条件下是如何工作的。

问题: 除了必须运行提示管理器的复杂性之外，全局方法的主要限制是进程类型有限。否则，该方法会受到上面讨论的假设和一些小问题的影响。

5 WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing

尽管冷启动很受欢迎，但它仍然是一个需要更多关注的问题。在本文中，我们解决了无服务器平台的冷启动问题。我们提出了 WLEC，一种容器管理架构，以最小化冷启动时间。WLEC 使用了一种经过修改的 S2LRU 结构，称为 S2LRU++，并增加了第三个队列。我们在 OpenLambda 中实现了 WLEC. 我们首次提出了一种综合的、结构化的方法。我们开发了 WLEC，一个基于 S2LRU++ 的架构，以更结构化的方式处理容器，以确保比无处不在的 lambda 模型更少的启动延迟和更多的并发性。

5.1 Wlec 的架构

图 4 显示了带有 WLEC 的 OpenLambda 的架构。

它由三个主要部分组成。a) 基于 Nginx 的负载均衡器，它接收客户端请求并将该请求分配给一个 worker。负载均衡机制高度依赖于系统的局部性管理。在这种情况下考虑的位置是会话位置、代码位置和数据位置。b)Lambda 存储也被称为一个注册表，它包含处理不同类型用户请求的所有代码。c) 由多个 worker 组成的执行引擎。每个 worker 就像一个基于 Linux 容器的沙盒，执行代码文件 (与客户端发出的 lambda 请求相关的函数代码)。我们可以看到，WLEC 作为负载均衡器和注册表的中间件，在任何给定的时间内为函数执行提供可用的最佳 worker。WLEC 的内部架构如图 5 所示，包括所有组件。WLEC 设计中最具挑战性的部分之一是并发请求和延迟请求 (15min 后) 的处理。最具挑战性的观点是本研究的目标是解决单一系统的设计挑战，并相应地管理系统。

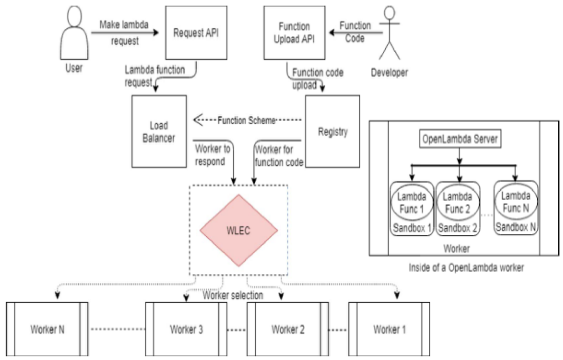

图 4 使用 WLEC 的 OpenLambda

5.2 WLEC 组件

在本节中，我们将描述 WLEC 的主要组件。

我们定义架构中使用的术语和组件。然后叙述了 CMS 及其子函数及其算法。

接下来，我们描述了 WLEC 的工作流程。

1) S2LRU++: S2LRU 是一种分区技术，用于制作两个段，一个用于试用段，另一个用于受保护段。在标准 S2LRU 算法中，新数据被插入到试用段中最近使用的位置 (MRU)。命中后，数据从试用段提升到受保护段的 MRU 位置。如果受保护网段的数据命中，则提升到同一网段的 MRU 位置。为了清晰起见，我们将保护段和试用段分别重命名为热队列和冷队列，如图 5 所示。这里我们使用容器而不是数据。

2) 模板容器列表: 一个容器列表，包含当前预热列表中所有容器的副本容器。这里，术语重复容器指的是另一个容器，该容器具有与克隆它的容器相同的库打包和环境配置，用于服务任何类似的传入 lambda 请求。

3) 暖时间: 对于暖队列中的每个容器，该容器头中都应该有一个暖时间。它指示容器应该离开预热队列的时间戳。保温时间以 Hr:Min:Sec 格式保

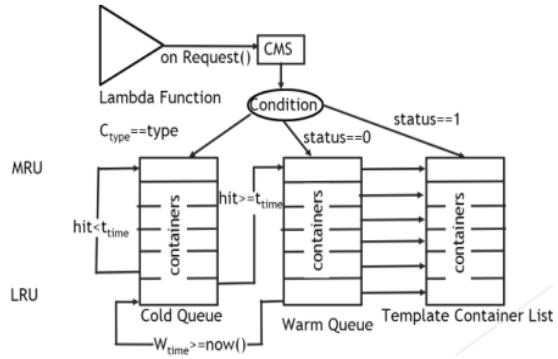

图 5 WLEC 架构

存在容器头内。

4) 容器头: 为了维护、监视和跟踪容器，OpenLambda 中的每个容器都有五个头文件。头参数可以分为两部分: 函数参数和容器参数。为每个 lambda 维护函数参数，但为每个容器维护容器参数。当在任何容器中调用新的 lambda 函数时，如果容器已经初始化，则更新容器参数。因为 lambda 参数在每次调用时都会更新。

6 Hermod: Principled and Practical Scheduling for Serverless Functions

无服务器计算因其易用性和成本效益而快速增长。然而，无服务器系统的关键组件功能调度却被忽视了。在本文中，我们采用第一原理方法来设计一个调度程序，以迎合实际部署中所见的无服务器功能的独特特征。我们首先创建了三个维度的调度策略分类。接下来，我们使用模拟来探索调度策略空间，并表明后期绑定和随机负载平衡等常用功能对于常见的执行时间分布和负载范围来说并不是最优的。我们使用这些见解来设计 Hermod，这是一个具有两个关键特征的无服务器功能调度程序。首先，为了避免由于函数执行时间的高可变性导致队头阻塞，Hermod 使用早期绑定和处理器共享的组合在各个工作机器上进行调度。其次，Hermod 具有成本、负载和位置感知能力。它改进了低负载时的整合，它在高负载时采用最小负载平衡以保持高性能，并且与纯基于负载的策略相比减少了冷启动的次数。

6.1 HERMOD 的设计

Hermod，这是一种既具有局部性又具有负载感知能力的调度器。Hermod 通过将早期绑定和处理器共享与适应负载变化的混合负载平衡相结合，迎合了 Serverless 功能的独特特性。此外，它会跟踪每个功能可用的热容器，尽可能避免冷启动。

Hermod 使用早期绑定和处理器共享调度来避免在不知道函数执行时间的情况下在高负载下可能导致高度减速的队头阻塞。为了解决负载最少和延迟绑定调度程序的缺点，Hermod 采用了局部感知混合负载平衡，根据负载使用两种不同的负载平衡模式之一。

低负载：在低负载下，即当某些 Worker 中有可用内核时，Hermod 将功能打包给具有可用内核的 Worker（图 6）。从任意一个 Worker 开始，Hermod 用与 Worker 核心数相等的调用次数填充它。对部署中的所有 Worker 迭代应用相同的过程。当正在进行的调用结束执行时，Hermod 将新传入的调用定向到相应的服务器以填充它们。

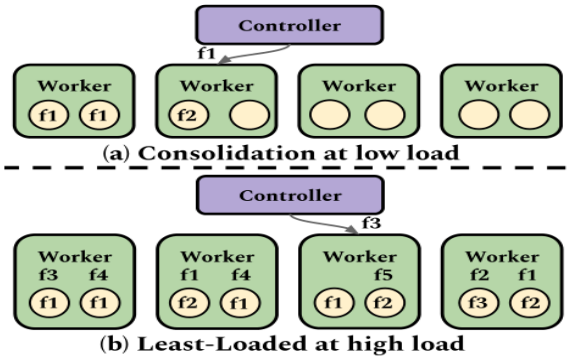

图 6 Hermod (a) 在低负载时以核心感知方式将函数调用打包到机器，(b) 在所有核心都已满时使用最小负载平衡。黄色圆圈代表每个 Worker 的 CPU 内核。

高负载：当所有 Worker 主机调用等于其核心数时，负载平衡将恢复到最低负载（图 5b）。这样，Hermod 减少了超载情况下的排队。我们选择不探索不同的方法来定义阈值，因为在已发布的跟踪中缺乏相关信息，例如，每个功能的 CPU 与 IO 混合。

当此类信息可用时，我们计划开发根据特定功能组合动态设置阈值的方法。Hermod 可以通过与每个 Worker 同步通信来准确区分两种操作模式。这会产生较低的开销，因为 Controller-Worker 通信延迟 (O(1msec)) 远低于函数执行时间 (O((1sec))，并且簿记非常轻量级，仅涉及更新 Controller 端的计数器。

局部性：上述混合负载平衡通过将调用整合到较少数量的服务器来缓解冷启动问题。为了进一步改进，我们明确地使 Hermod 的负载平衡策略具有局部感知能力。

Hermod 跟踪每个 Worker 中可用的暖容器。当它在低负载模式下运行时，它首先在其中一个具有

可用容量的非空 Worker 中寻找温暖的容器。如果存在这样的 Worker，Hermod 会在那里引导调用。如果不是，它会优先考虑合并，方法是选择一个没有暖容器的非空 Worker，而不是一个有暖容器的空 Worker。当它在高负载下运行时，Hermod 使用局部性打破联系。

如果多个 Worker 均等加载，则如果存在这样的 Worker，它会选择带有暖容器的那个作为传入调用。

总而言之，Hermod 实现了两全其美：低减速下的高整合度和低负载下的低冷启动率，同时在高负载下保持低减速。

7 总结及展望

随着 Serverless 计算的不断发展，它已经成为了云计算领域中的一项重要技术，并且在各个领域得到了广泛的应用。然而，由于 Serverless 计算存在冷启动问题，导致函数的执行效率和性能都受到了很大的影响，限制了 Serverless 计算的进一步发展。因此，解决 Serverless 计算中的冷启动问题是当前研究的热点之一。

在已有的研究成果的基础上，未来可以从以下几个方面继续探索和优化 Serverless 计算的冷启动问题：

1、发展更加精准的预热方法目前，大多数预热方法都是基于传统的机器学习算法，如线性回归和决策树等。然而，这些算法可能无法处理更加复杂的数据模式。因此，可以探索更加精准的预热方法，例如基于深度学习的预热方法，以提高预热效果。

2、开发更加智能的函数调度算法目前的函数调度算法主要基于负载均衡的策略，这种方法无法准确预测函数的执行时间和资源需求。因此，可以探索更加智能的函数调度算法，例如基于机器学习或深度学习的调度算法，以提高 Serverless 函数的执行效率和性能。

3、构建更加可扩展的预热策略目前的预热方法大多是单一的预热策略，即在函数执行前进行一次预热。但是，可以采用多级预热策略，根据不同场景和时间选择不同的预热方法，以提高预热效果。

4、解决 Serverless 计算中的内存安全性问题随着 Serverless 计算的快速发展和广泛应用，内存安全性问题也越来越受到重视。因此，未来的研究还应该关注如何解决 Serverless 计算中的内存安全性问题，例如通过检查和隔离函数执行期间的内存使用来保证计算安全性。

总之，未来的研究可以继续探索和优化 Serverless 计算中的冷启动问题，并针对其它的问题，例如内存安全性问题等进行更深入的研究，以推动 Serverless 计算的发展和应用。除此之外，还可以从以下方面进一步探索和优化 Serverless 计算：

5、发展更加可靠的预热方法目前的预热方法虽然已经取得了一定的成果，但仍然存在一些不足之处，例如预热效果不稳定、可靠性差等。因此，未来的研究可以进一步发展更加可靠的预热方法，例如结合 Serverless 函数的历史执行数据进行预热，以提高预热效果的可靠性。

6、研究 Serverless 计算中的资源分配问题 Serverless 计算中，不同函数的资源需求不同，如何合理分配资源以满足函数的需求，是一个非常重要的问题。未来的研究可以探索更加智能的资源分配方法，例如基于机器学习或深度学习的资源分配算法，以提高 Serverless 函数的性能和效率。

7、探索 Serverless 计算与容器技术的结合容器技术具有良好的可移植性和易于管理等特点，可以帮助 Serverless 计算更好地应对冷启动问题。因此，未来的研究可以探索 Serverless 计算与容器技术的结合，以实现更加灵活、高效和可靠的 Serverless 计算。

总之，Serverless 计算作为云计算领域中的一项重要技术，其冷启动问题的解决是当前研究的热点之一。未来的研究可以从多个方面继续探索和优化 Serverless 计算，以推动其进一步发展和应用。

参 考 文 献

References

[1] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis, Hermod: Principled and Practical Scheduling for Serverless Functions. SoCC ' 22, November 7–11, 2022, San Francisco, CA, USA,

[2] Khondokar Solaiman∗, Muhammad Abdullah Adnan, WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing. 2020 IEEE International Conference on Cloud Engineering (IC2E)

[3] David Bermbach, Ahmet-Serdar Karakaya, Simon Buchholz, Using Application Knowledge to Reduce Cold Starts in FaaS Services. TU Berlin Einstein Center Digital Future Mobile Cloud Computing Research Group Berlin, Germany

[4] Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen, Reducing cold

starts during elastic scaling of containers in Kubernetes. SAC ’ 21, March 22–26, 2021, Virtual Event, Republic of Korea

[5] Biswajeet Sethi, Sourav Kanti Addya Soumya K Ghosh, LCS : Alleviating Total Cold Start Latency in Serverless Applications with LRU Warm Container Approach. ICDCN 2023, January 4–7, 2023, Kharagpur, India