Computer Science and Application 计算机科学与应用, 2020, 10(11), 1993-2005
Published Online November 2020 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2020.1011211

Hans汉斯

一种面向云函数的超轻量运行时环境构建方法

吴绍岭[1]，田春岐[1]，万　兴[1]，王　伟[2]

[1]同济大学计算机科学与技术系，上海
[2]华东师范大学数据科学与工程学院，上海
Email: wsl_go@163.com, tianchunqi@tongji.edu.cn, 625660134@qq.com

收稿日期：2020年11月3日；录用日期：2020年11月18日；发布日期：2020年11月25日

摘　要

随着虚拟化技术和软件体系结构的不断发展，云函数(也称无服务计算)逐渐成为一种新的计算范式，该范式允许用户直接部署函数代码，而不必关心基础架构，同时能够提供极高的弹性和快速生灭能力。然而，现有的云函数通常运行在功能较为完备的容器中，存在启动延迟高、资源共享度低、镜像构建和分发时间长等问题。本文分析并验证了影响Linux容器启动时延和资源共享的关键因素，提出了面向云函数的超轻量运行时环境构建方法，该方法从资源隔离、资源限制、文件系统、网络通信等方面对运行时环境进行定义，基于操作系统虚拟化技术，通过进一步抽取共享层来提高资源共享度，通过降低网络隔离和控制组操作的性能瓶颈来加快启动速度。根据上述方法实现了超轻量运行时环境引擎FRE (Function Runtime Environment)。实验中，从顺序与并发启动时间、内存资源利用率、镜像大小等方面对FRE与当前主流的Docker容器引擎进行对比实验，验证了FRE在云函数场景下的有效性。

关键词

云函数，无服务器计算，虚拟化，容器，资源共享

A Lightweight Runtime Environment Construction Method for Cloud Functions

Shaoling Wu[1], Chunqi Tian[1], Xing Wan[1], Wei Wang[2]

[1]Department of Computer Science and Engineering, Tongji University, Shanghai
[2]School of Data Science & Engineering, East China Normal University, Shanghai
Email: wsl_go@163.com, tianchunqi@tongji.edu.cn, 625660134@qq.com

Received: Nov. 3rd, 2020; accepted: Nov. 18th, 2020; published: Nov. 25th, 2020

Abstract

With the development of virtualization technology and software architecture, cloud function (also known as serverless computing) has gradually become a new computing paradigm, which allows

文章引用: 吴绍岭, 田春岐, 万兴, 王伟. 一种面向云函数的超轻量运行时环境构建方法[J]. 计算机科学与应用, 2020, 10(11): 1993-2005. DOI: 10.12677/csa.2020.1011211

users to directly deploy function codes without having to consider the infrastructure, while providing high flexibility and rapid birth and destroy ability. However, existing cloud functions usually run in fully functional containers, with problems such as high startup latency, low resource sharing, and long image build and distribution time. This paper analyzes and verifies the key factors affecting Linux container startup latency and resource sharing, and presents a cloud function-oriented lightweight runtime environment construction method, which defines the runtime environment from four aspects: resource isolation, resource restriction, file system, and network communication. Based on the operating system virtualization technology, the resource sharing is improved by further extracting the shared layer, and the startup speed is accelerated by reducing the performance bottleneck of network isolation and control group operation. Based on these designs, we implement FRE (Function Runtime Environment), a lightweight runtime environment engine. Finally, we carry out a series of comparative experiments on sequence and concurrent startup time, memory resource utilization, image volume etc. between FRE and Docker, and verify the effectiveness of the proposed method in the cloud function scenario.

Keywords

Cloud Function, Serverless Computing, Virtualization, Container, Resource Sharing

1. 引言

传统云计算自上而下可以分为应用层(SaaS)、平台层(PaaS)、基础层(IaaS) [1]。云函数，也称无服务器计算(Serverless Computing)或函数即服务(FaaS)，是一种新兴的云计算范式，其计算的基本单位是独立的函数。为了保证应用之间的安全性，函数通常在具有受限资源(例如 CPU 时间和内存)的容器或其他类型的沙箱中执行。与传统的基础架构即服务(IaaS)平台中的虚拟机不同，函数实例仅在调用时启动，并在处理请求后立即进入睡眠状态，租户按服务调用次数收费，无需支付未使用和空闲的资源[2]。云函数平台对租户完全隐藏了服务器管理，提供自动弹性伸缩能力以适应负载，这样开发人员就不必关注底层机器的数量和配置，这也为云提供商带来了提高其计算资源效率的机会[3]。

相比于传统服务，云函数具有许多新的特性：其计算的基本单位是函数，粒度更细，单主机实例密度更高；函数的运行一般靠事件驱动，运行环境的创建和销毁更加频繁。这些特性使得云函数运行环境面临新的挑战：隔离的细粒度的计算单位降低了资源共享度，尤其是内存资源；频繁的冷启动会给请求的处理带来额外的时间开销[4]。因此，云函数的运行环境需要具备快速启动能力和较高的资源利用率。

近年来，容器技术一直是学术界和工业界研究的热点，并广泛用于云函数[5] [6]、微服务、软件开发运维(DevOps)等领域。容器使用操作系统虚拟化技术，利用内核功能来打包和隔离进程，可以作为常规系统进程运行[7]。相比于虚拟机，容器具有无可比拟的轻量级优势，其共享主机的内核，镜像文件的体积更小，创建和销毁速度更快。容器通常比传统 VM 效率更高[8]。

尽管容器具有轻量级优势，但将现有以 Docker [9]为代表的容器作为云函数的运行环境仍然存在一些问题，具体而言：现有容器使用 Linux 命名空间(Namespace) [10]机制来实现多方面的隔离，使用控制组(Cgroup) [11]来限制进程可使用的资源，这会对容器的启动速度带来一定的影响；其次，现有容器通常封

装了较为完整的操作系统发行版的功能，这会导致在运行多个容器时相同依赖文件被重复加载进内存，造成内存冗余，同时这些功能在运行简短的云函数代码时并不会全部使用到，造成镜像体积大，构建和分发时间长等问题。

容器启动速度和内存资源利用率对租户和云函数提供商都产生很大的影响，尤其是在开源协作自动化平台领域，这类平台允许用户自定义事件处理函数，为了保证安全性，处理函数通常运行在容器中。当收到事件时，系统启动容器并执行用户的函数脚本，执行完成之后立即销毁函数实例，以取得性能和成本的平衡。运行环境具备更快的启动速度和更高的资源利用率将有效提高该类平台的服务能力，同时能够有效节省成本。

本文的主要贡献有 3 个方面：

1) 通过分析和实验验证了命名空间与控制组对容器启动速度的影响，以及文件隔离对资源共享带来的影响。

2) 提出了一种面向云函数的超轻量运行时环境构建方法，通过降低网络隔离和控制组操作的性能瓶颈来加快启动速度，通过进一步抽象共享层来提高资源共享度，降低镜像体积和构建、分发时间。

3) 基于以上方法实现了面向云函数的超轻量运行时环境引擎 FRE，在顺序和并发启动时间、内存资源利用率、镜像大小等方面进行对比实验，验证了所提方法的有效性。

本文第 2 节介绍容器优化相关的工作，第 3 节介绍影响 Linux 容器启动速度和资源共享的关键因素，第 4 节介绍面向云函数的超轻量运行时环境整体设计，第 5 节介绍 FRE 原型实现，第 6 节介绍实验和结果分析，第 7 节总结和未来展望。

2. 相关工作

类似于从汇编语言到高级编程语言的发展，云计算的发展经历了从裸机，到虚拟机，到容器，再到无服务器计算的变革。无服务器计算代表了云计算发展的下一个前沿，对运行环境的启动速度和资源利用率要求更高，学术界也在不断提出许多优化方案。

一些学者通过精简容器镜像的体积和优化镜像加载方式来提高容器的启动速度。Tyler Harter 等设计了能够快速启动容器的新 Docker 存储驱动程序 Slacker [12]。Slacker 基于集中存储，通过优先下载与容器启动相关的文件并延迟加载其它文件来减少容器的部署周期，加快启动速度。CNTR [13]将传统容器镜像分为“胖(包含工具)”、“瘦(包含主要应用程序)”两部分，通过优先部署“瘦”镜像并动态加载“胖”镜像来提高启动速度。TotalCOW [14]使用写时复制与缓存共享技术来最大程度地减少内存占用和避免不必要的 I/O 操作。以上方式在一定程度上精简了镜像的体积，加快了容器的部署周期，但并未涉及到命名空间和控制组带来的影响，且内存资源的共享度并未得到提升。

Pipsqueak [15]及其后续工作 SOCK [16]通过创建预热的 Python 解释器的缓存来提高容器的启动速度，作者首先验证了程序依赖包的加载过程会降低容器的启动速度，然后分析了当前主流的 Python 包的依赖情况和使用频率，最后通过预先加载常用的函数依赖库到内存中来减少容器启动延迟，同时设计了基于 Zygotes 的三层缓存策略来进一步加快启动速度。但是这种方式只针对具有相同解释器的函数代码生效，且需要一些特殊机制来选择最合适的解释器和缓存策略，不具有通用性。

SAND [17]引入了两级隔离模型来提高无服务器计算的性能。其两级隔离模型的关键思想是不同应用程序之间使用容器来实现较强的隔离机制，同一应用程序的多个副本之间基于进程模型进行隔离，这种方式使得 SAND 可以快速分配和回收资源，从而降低了函数的启动延迟。在多租户的云函数场景中，安全性至关重要，SAND 基于进程的隔离模型无法保证函数实例的强隔离性，且无法对每个进程实例做资源限制，仅适用于弱安全需求的场景中。

尽管基于操作系统虚拟化的容器技术已经广泛应用在包括云函数在内的各个领域中，但一些学者仍在探索一些新的虚拟化实现方式，Unikernal [18]便是一种解决方案。Unikernel 是一种使用库操作系统(library operating system)构建的专用单一地址空间机器映像[19]，仅由一个应用程序及其所依赖的最小的操作系统内核组成，通常可以在裸机或虚拟机管理程序上运行。Filipe Manco 等[20]提出了一种基于 Xen 虚拟化的 Unikernel 实现方案 LightVM，性能评估表明，LightVM 的启动速度和并发度都优于传统的容器。尽管定制的 Unikernel 在镜像大小和启动速度上具有一定的优势，但是 Unikernel 从静态密封的整体式设备中剥离了进程抽象，从而牺牲了灵活性、效率和适用性。

除此之外，Shillaker [21]评估了 OpenWhisk 在不同级别的吞吐量和并发功能上的响应延迟，确定了通过在运行时中用新的隔离机制替换容器来缩短无服务器框架中启动时间的研究方向。Wang 等[2]对主要的无服务器平台做出的许多设计决策进行了逆向分析，为无服务器平台的优化提供了参考依据。McGrath 等[22]还研究了现有无服务器框架中的延迟，这些问题对于无服务器运行环境的设计非常重要。

3. 容器性能影响因素分析

3.1. 容器启动速度影响因素分析

Linux 容器通常使用命名空间(Namespace)机制来实现资源隔离，使用控制组(Cgroup)来限制进程可使用的资源。本节将分别讨论命名空间和控制组对容器启动速度的影响。

3.1.1. 命名空间(Namespace)

命名空间可实现进程资源的隔离，进程只能感知当前所在的命名空间下的资源，无法直接感知其它命名空间下的资源。常用的命名空间包括：IPC (进程间通信)、Network (网络)、Mount (文件系统挂载点)、PID (进程 ID)、User (用户和组)、UTS (主机名和 NIS 域名)。在创建新进程时，可以通过指定资源对应的参数来创建新的命名空间，这样就实现了子进程与父进程资源隔离的效果。

我们首先测试了不启用命名空间和分别启用单个命名空间对进程并发创建时间的影响，结果表明，Network 命名空间对进程的并发创建影响较大，而其它 5 种命名空间则几乎无影响。进一步，我们测试了同时启用 6 种命名空间和同时启用除 Network 之外的 5 种命名空间，然后和不启用命名空间时进程创建时间进行对比，结果如图 1 所示。其中 a 为 512 并发情况，b 为 1024 并发的情况，纵轴为创建进程所需的时间(图中的值为实际值开方后的结果)。可以观察到，同时启用除 Network 之外的 5 种命名空间时，和不启用命名空间的性能基本一致。Network 命名空间对并发创建进程的影响较大，这是由于 Network 命名空间之间共享单个全局锁[23]，所以在创建新的 Network 命名空间时，将会因争抢全局锁而导致进程创建延迟非常高。

3.1.2. 控制组(Cgroup)

控制组可限制进程可使用的资源量。控制组把系统中的所有进程组织成一棵棵独立的树，每棵树都包含系统的所有进程，且和一个或者多个 subsystem 关联，而 subsystem 被用来限制每个进程组可使用的资源(CPU、内存、网络接口、磁盘 IO 等)。当创建新的进程时，可以为进程创建新的 Cgroup 并把进程加入进去，也可以将进程直接加入已有的 Cgroup，从而实现进程的资源限制。

为了测试 Cgroup 对容器启动速度的影响，我们测试了在不同并发度下创建新的 Cgroup(create)和重用已有的空闲的 Cgroup(reuse)的性能，结果如图 2 所示。图中横轴为并发操作数，纵轴为操作所需的时间，可以看出，reuse 的方式消耗时间较少(<1 ms)，且几乎不受并发度的影响，而 create 的方式则受并发度影响较大，随着并发创建数量的增多，创建时间会延长。

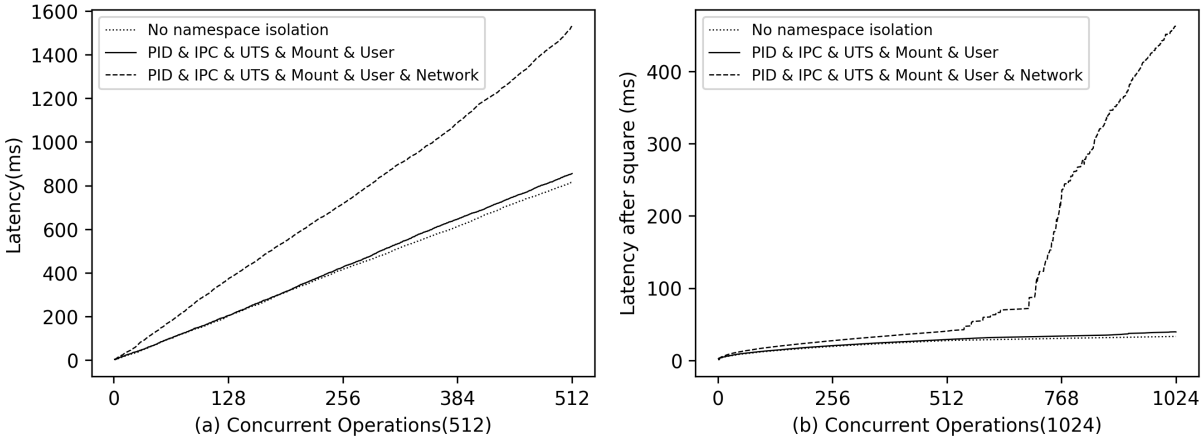

Figure 1. Comparison of concurrent namespaces creation time
图 1. 命名空间并发创建时间对比

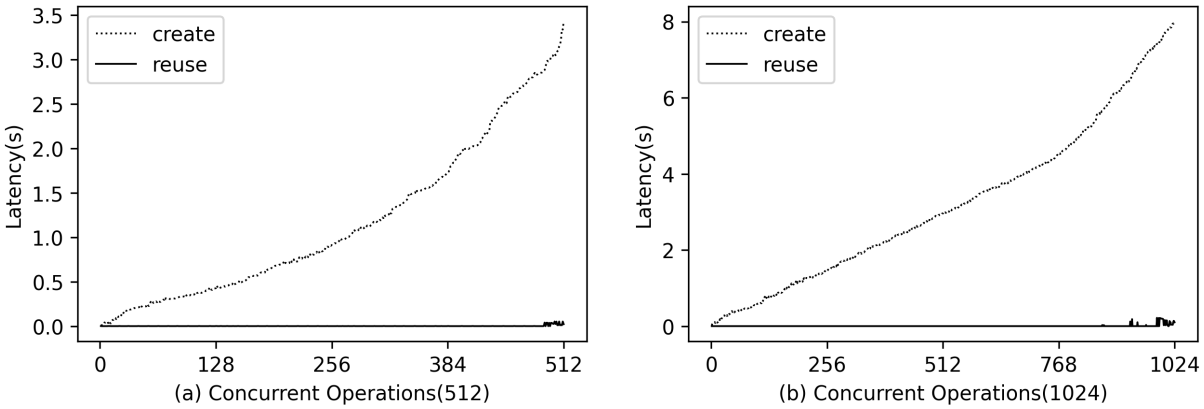

Figure 2. Comparison of Cgroup concurrent operation performance
图 2. Cgroup 并发操作性能对比

3.2. 容器内存资源利用率影响因素分析

容器基于镜像来启动，镜像打包了运行应用所需的所有内容，包括代码、运行时、库、环境变量和配置等。当前容器镜像通常封装了较为完备的文件，这样既可以保证应用运行环境的完备性，也能很好地控制依赖文件的版本，但这种方法却可能导致不同容器依赖的相同文件重复加载进内存，造成内存资源浪费。Docker 使用联合文件系统机制在一定程度上提高了内存共享度，但对于其他存储驱动或非同源镜像启动的容器仍然存在内存冗余问题[12]。

共享库是实现内存共享的一种关键方式，Linux 系统使用虚地址空间机制实现了不同应用所依赖的同一个共享库文件在内存中也只有一份，从而能够有效消除内存冗余，提高资源利用率。Ferreira 等人[24]的工作中提到，运行时依赖文件在内存中的冗余程度是影响内存利用率的重要因素。Zhang [25]等人通过建立数学模型，分析并量化了共享库与内存资源利用率之间的关系，从理论上证明了容器间即使仅共享一小部分内容，就可以显著提高内存的利用率。

3.3. 小结

通过以上分析可知，网络命名空间和控制组会对容器的创建性能带来一定的损耗，而镜像文件的隔离也造成了内存资源的冗余。网络隔离在传统容器设计中能带来安全性和灵活性，但是在云函数场景中，

云函数平台通常具有请求网关，所有请求事件会统一路由到网关，网关预处理后再交给应用函数来处理[8] [9]，这种方式使得网络命名空间的价值很小；在创建容器时，为容器创建新的控制组能简化处理逻辑，但也增加了延迟，而控制组的复用能显著降低容器的创建延迟，因此通过维护控制组资源池的方式能够有效提高容器的创建速度；相比于传统服务，云函数服务仅包含必要的代码，这种特性使得云函数运行环境可以做到非常精简，这也为进一步抽象共享层，提高资源共享度提供了契机。

4. 超轻量运行时环境整体设计

上节的分析和结论给云函数运行环境的设计带来很多启发，本节根据前面的结论提出了一种全新的面向云函数的超轻量运行时环境构建方法。

4.1. 超轻量运行时环境

云函数超轻量运行时环境基于两个主要设计目标：1) 降低命名空间和控制组对容器创建带来的时间损耗；2) 提高容器间内存资源的共享度。其设计涉及到资源隔离、资源限制、文件系统、网络通信四个方面。

4.1.1. 资源隔离

为了保证安全性，资源隔离是必要的，我们采用 Linux Namespace 机制对进程的 IPC、Mount、PID、User、UTS 资源进行隔离，而根据上一节的分析结果，Network 命名空间会对容器的并发创建产生非常大的时间损耗，且网络隔离给云函数带来的价值并不大，因此不进行 Network 命名空间的隔离。

4.1.2. 资源限制

同样的，为了保证安全性，需要对容器可使用的资源量进行限制，我们基于 Linux 控制组机制来实现：当创建容器时，会将容器进程加入 CPU、内存、设备等资源对应的控制组中，从而将容器最多可使用的资源量限制在指定的阈值内。在上一节的分析中我们发现控制组的并发创建会给容器创建速度带来很大的影响，因此这里采用控制组缓存池机制，如图 4 所示，容器引擎会预先创建空闲控制组缓存池(Cgroup Pool)，当创建容器时，先从缓存池获取可用的控制组，获取不到时才会执行创建操作。当容器退出时，其绑定的控制组不会被直接删除，而是加入缓存池中以备复用。

4.1.3. 文件系统

从上一节的分析可知，容器之间通过共享库文件可以有效提高内存资源的利用率，为了进一步提高内存资源的共享度，我们重新组织了容器文件系统，进一步抽取出共享层，同一台主机中函数运行所依赖的二进制文件、库文件、依赖包文件和代码及配置文件在磁盘中仅有一份，且在内存中也只加载一份，这种方式有效降低了磁盘和内存冗余。其与 Docker 容器结构的主要差别如图 3 所示。

图 4 展示了容器的文件系统，根据文件是否可以被共享，可以将文件分成 2 类：只读文件和可读写文件。只读文件具体包括二进制文件(bins)、动态链接库文件(libs)、代码和配置文件(codes)、函数依赖包文件(packages)，同时为了保证容器内进程的正确运行，还只读挂载载了主机的/sys，/proc 等目录(图中未画出)，由于这类文件可被多个容器共享，因此通过只读的方式防止进程恶意操作是有意义且必要的。另一类是可读写文件，具体包括 Unix Domain Socket 文件和一个可读写的目录 workDir。Unix Domain Socket 文件用来实现容器进程和容器引擎之间的通信。考虑到函数运行过程中产生需要持久化存储的数据，因此文件系统还包括一个可读写的目录 workDir，该目录是租户级别隔离的，同一租户下的所有函数实例均能以读写的方式访问该目录，并且该目录中的文件在主机中持久化存储，不会随容器的删除而删除。

容器所需的文件通过硬链接的方式组织到单独的目录中，并把这个目录作为容器文件系统的根目录挂载点。不同容器中依赖的只读文件会统一映射到主机文件，所以容器间相同的依赖文件具有相同的索引节点号，因此这些文件在磁盘中仅有一份，在内存中也仅有一份。

4.1.4. 网络通信

我们使用 Unix Domain Socket 机制来实现容器进程与容器引擎之间的通信，如图 3 所示，Unix Domain Socket 文件由容器引擎生成，并被挂载到容器中，容器进程可通过此文件与容器引擎之间建立双向通信通道，后续所有的数据、控制指令等消息均可通过此通道来进行通信。

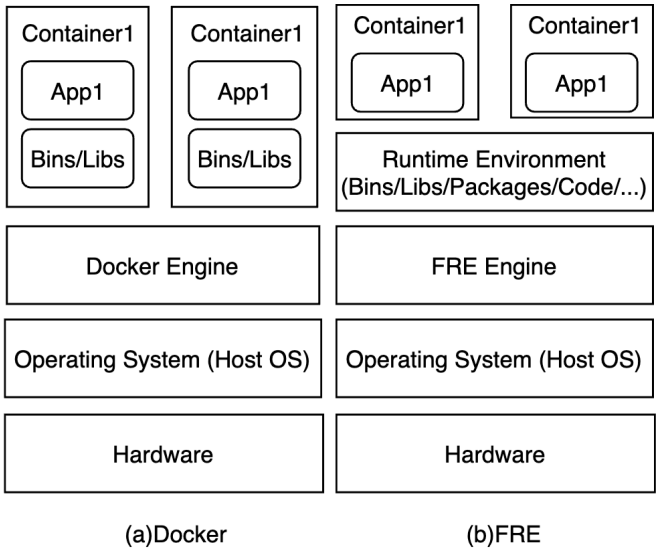

Figure 3. Comparison of Docker and FRE
图 3. 2 种容器的架构对比

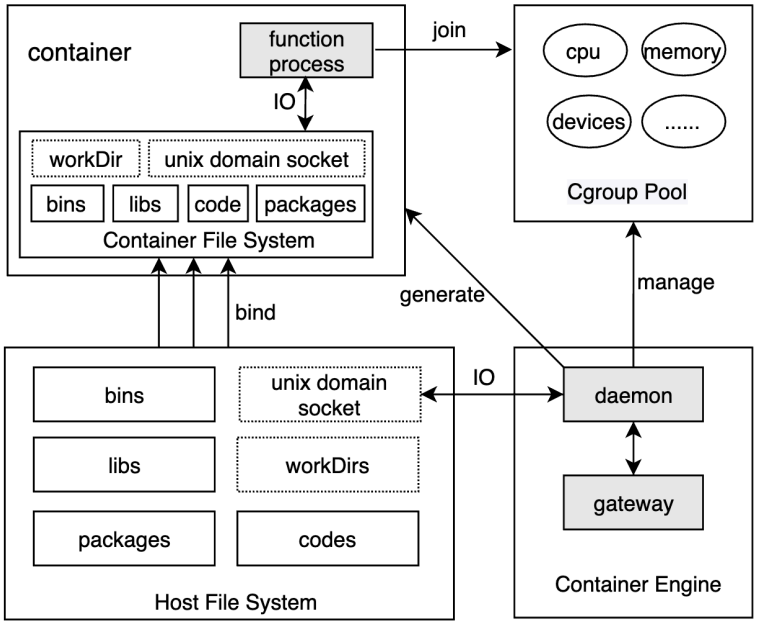

Figure 4. System architecture of lightweight runtime environment
图 4. 超轻量运行时环境系统架构

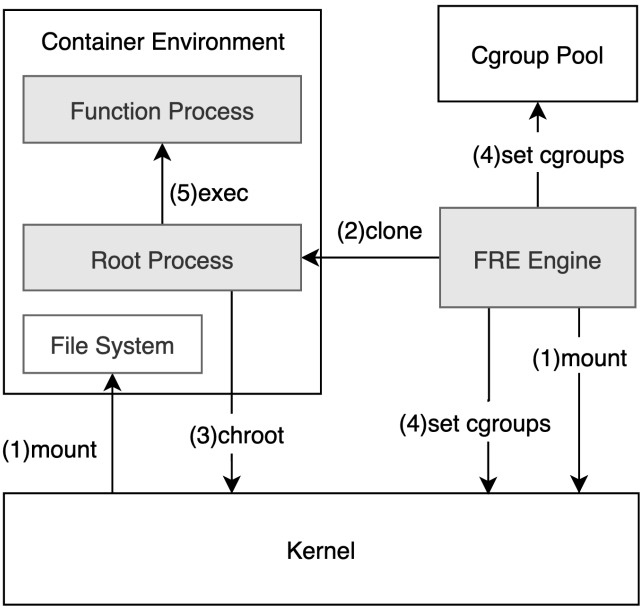

Figure 5. Runtime environment creation process
图 5. 运行时环境创建流程

4.2. 运行时环境创建流程

图 5 展示了容器化的运行时环境的创建过程，具体来说：1) 容器引擎首先调用内核的 mount 系统调用，为容器挂载文件系统；2) 容器引擎调用 clone 系统调用创建容器的根进程，同时指定生成新的命名空间，这里不包括网络命名空间；3) 创建出来的子进程作为容器的根进程，然后根进程执行 chroot 操作，更改根目录；4) 容器引擎从缓存池中获取或者创建新的 Cgroup，然后调用内核函数将容器进程加入 Cgroup 中，以限制进程的可用资源量；5) 容器内的根进程准备函数所需要的上下文信息，然后调用 exec 系统调用创建函数进程，随后开始执行函数。

4.3. 函数镜像与仓库

Docker 容器镜像通常封装了较为完整的功能，保证了容器运行环境的一致性和可移植性，Docker 采用分层的方式来组织镜像，并实现了容器镜像的配置化，用户使用 Dockerfile 可以完整构造出一个镜像。基于云函数所需运行环境精简的特点，我们对云函数的镜像做了超轻量处理，其仅包含函数运行所依赖的文件，同时提出了运行时环境描述模版，通过模版可以生成函数环境镜像。运行时环境描述模版指明了函数运行所需的基础运行环境(如 Python、Java、Nodejs 等)、函数依赖包、函数代码和配置等文件。云函数远程镜像仓库包含了各类语言的基础运行环境。当构造函数镜像时，会根据模版从云函数远程镜像仓库下载相应语言的基础运行环境，函数依赖包文件从当前主流的包管理平台(如 PyPI、NPMjs、MvnRepository 等)下载，为了保证用户的代码和配置的安全性，代码和配置文件会从指定的私有代码托管平台下载，下载完成后，再根据环境描述模版将这些文件的硬链接组织到指定的目录中，函数镜像即生成完成。

5. FRE 原型实现

本节基于上一节提到的方法实现了面向云函数的超轻量运行时环境原型 FRE (Function Runtime Environment)。FRE 的整体架构如图 6 所示，其主要包含 LRE Engine、FRE Local Registry、Remote Registry 3 个部分。

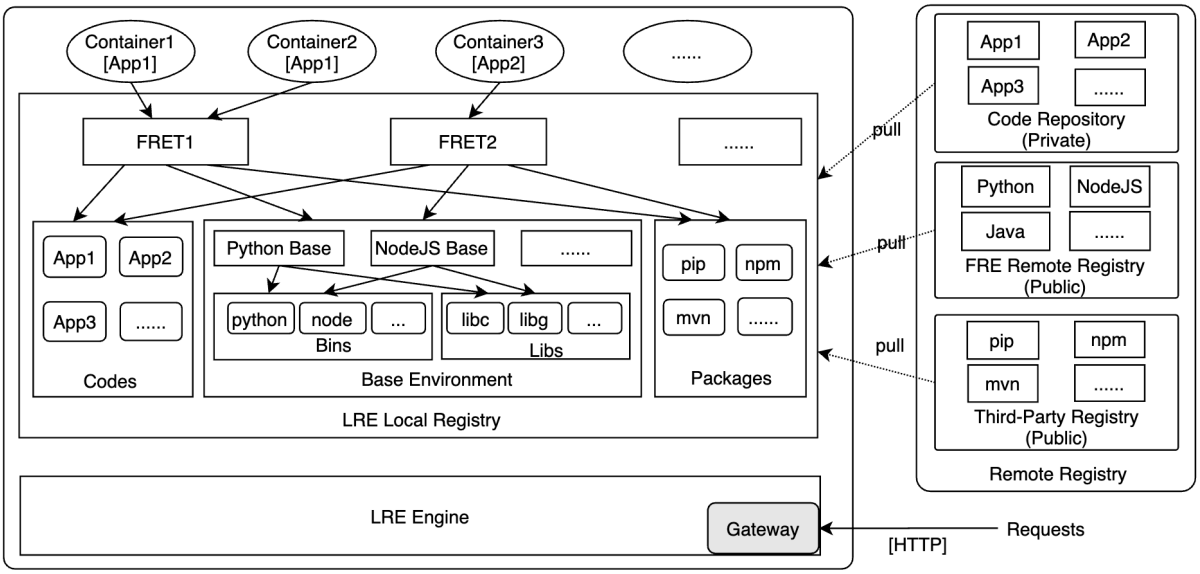

Figure 6. FRE lightweight runtime environment engine architecture
图 6. FRE 超轻量运行时环境引擎架构图

FRE Engine 是事件处理和超轻量运行时环境的管理核心，我们基于 Golang 编程语言实现。Engine 中的 Gateway 以 REST API 的形式对外暴露服务，负责接收并解析所有外部的 HTTP 请求，然后再路由到具体的处理模块；Engine 负责云函数运行时环境全生命周期的管理，具体包括运行时环境的创建、启动、删除等操作，这些操作均使用 Linux 内核提供的原生接口；Engine 还负责运行时环境镜像的构建工作，包括解析运行时环境模版、从远程仓库拉取文件、生成运行时环境的文件系统等。

FRE Local Registry 是主机本地函数依赖环境仓库，包含了运行时环境所依赖的二进制文件、共享库文件、依赖包文件和函数代码等文件，这些文件被所有的运行时环境所共享。FRET (Function Runtime Environment Template)类似于 Docker 镜像，根据运行时环境描述模版生成，其内部包含了函数运行所依赖的文件，可作为函数运行时环境的文件系统。当构建 FRET 时，Engine 根据环境描述模版首先从本地仓库中查找文件，查找不到时再从远程仓库拉取到本地。运行时环境描述模版采用 YAML 格式，定义了 FRET 唯一标识、版本、基础环境、函数代码、附加共享库、依赖包、资源限制等基本属性。

Remote Registry 作为远程仓库，负责依赖文件的存储和分发。其中 Code Repository 为租户私有代码仓库，存放函数代码和配置；FRE Remote Registry 存放各类语言的基础环境，主要包含二进制文件和共享库文件；Third-Party Registry 为各类第三方公共服务，例如 pip、npm、mvn 等仓库，公共仓库提供了完善的存储和分发能力，使用第三方公共仓库服务能够利用已有资源，显著降低系统的复杂度。

6. 实验结果与分析

本节我们将 FRE 引擎与 Docker 引擎进行性能对比评估，主要从容器顺序和并发启动时间、容器内存占用情况、容器镜像文件大小四个方面进行测试，实验环境配置为：8 核 Intel Xeon E5 2.20 GHz 处理器；16 GB LPDDR3 内存；128GB PCIeNVMe 高速固态硬盘；操作系统为 Centos 7.6.1810 64 位，内核版本为 3.10.0-957.el7.x86_64；Docker Server 为官方 Community 19.03.1 版本；FRE 为 0.1 版本，基于 go1.15.2 构建。根据上述配置，我们准备了 3 台相同配置的主机用于对比实验。

6.1. 启动时间

本节对 FRE 和 Docker 容器引擎顺序和并发启动实例的时间进行了测试。我们使用 Golang 语言构建

了专门的测试程序，该程序启动时输出当前纳秒级的时间戳，然后进入睡眠状态等待终止信号。通过记录容器启动时刻的时间戳和测试程序输出的时间戳可以计算出容器启动花费的时间。我们基于该测试程序分别构建了 FRE 和 Docker 镜像，作为容器启动时间基准测试镜像。

顺序启动：我们在相同的系统环境下分别使用 FRE 和 Docker 进行连续启动容器实例测试，容器基于前面构建的基准测试镜像启动，每组测试仅连续启动的实例数量不同，结果如图 7 所示。Docker 容器的平均启动时间大于 400 ms，且随着连续启动数量的增加，平均启动时间也有所上升(>10 ms)。FRE 容器的平均启动时间在 20 ms 左右，且随着连续启动数量的增加，平均启动时间增幅较小(≤2 ms)。相比于 Docker，FRE 在连续启动不同数量的容器时，启动时间均下降了 95%以上，启动速度得到大幅提升。

并发启动：云函数具有快速弹性伸缩的特性，因此容器的并发启动性能至关重要。我们对 FRE 和 Docker 的并发启动性能进行了测试，这里复用了顺序启动中的测试方法和测试环境，在创建容器时，将顺序启动改为并发启动。图 8 展示了 FRE 和 Docker 在不同并发量下实例启动时间的累积分布函数(CDF)图，纵坐标为实际值取以 2 为底的对数后的值。从图中可以看出，在不同的并发量下 FRE 实例的启动时间都远低于 Docker 容器实例的启动时间。在 256 并发下，Docker 容器的最长启动时间达到了 32,820 ms，而 FRE 实例仅为 776ms。相比于 Docker，FRE 在不同并发数量下的最长启动时间均下降了 97%以上。

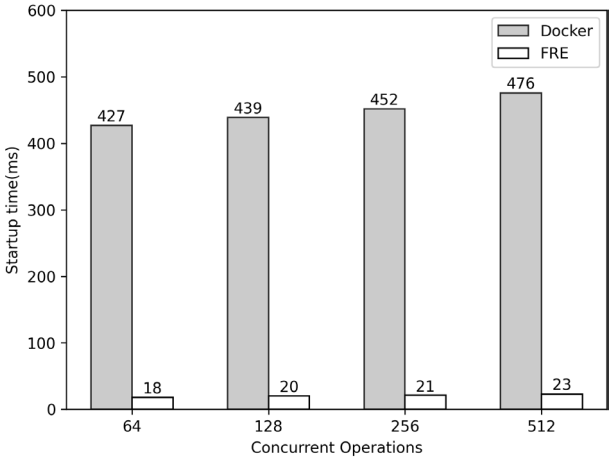

Figure 7. Comparison of container serial startup time
图 7. 容器顺序启动时间对比

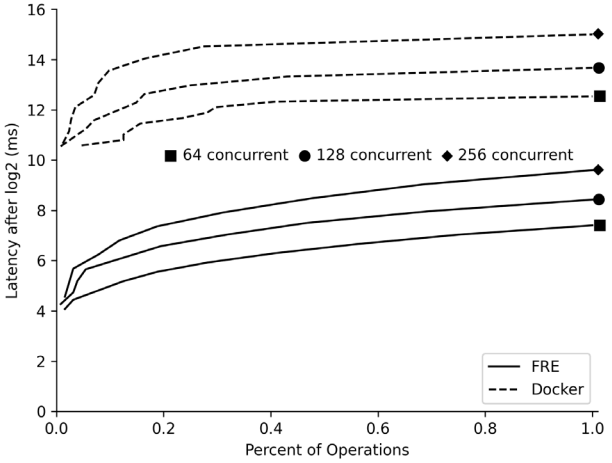

Figure 8. Comparison of container concurrent startup time
图 8. 容器并发启动时间对比

6.2. 内存利用率

云函数计算粒度更小，有效的内存利用将显著提高单机函数实例的密度，本节针对 FRE 和 Docker 容器实例进行内存使用测试。对于 FRE，我们构建了 Nodejs8、Python2.7、Python3、Java8 和 PHP5.6 这 5 种语言的基础运行环境；对于 Docker，我们使用 DockerHub 提供的对应版本的镜像。同时针对每种语言的运行环境构建了 web 应用，然后将 web 应用放到对应的镜像中，最终每种语言运行环境的 Docker 镜像和 FRE 镜像中均包含相同的 web 应用。

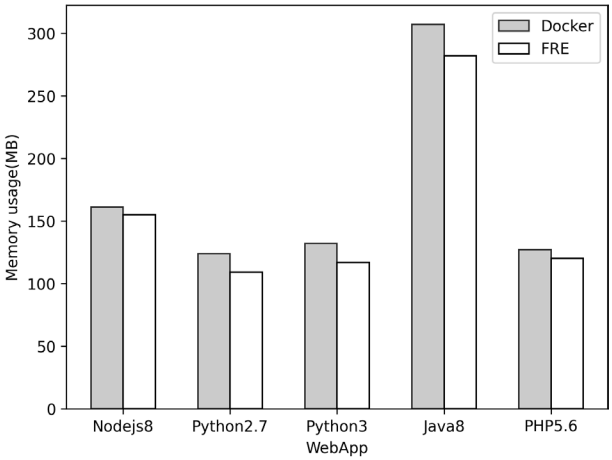

Figure 9. Comparison of single container memory usage
图 9. 单类型容器内存占用情况对比

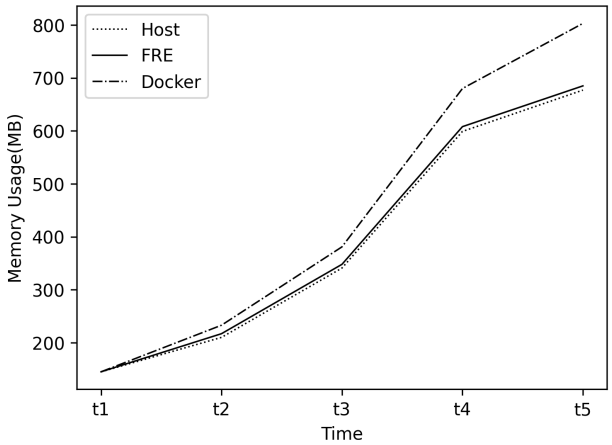

Figure 10. Comparison of multiple container memory usage
图 10. 多类型容器内存占用情况对比

针对以上 5 种类型的 web 应用，分别启动相同数量的 FRE 容器实例和 Docker 容器实例，然后记录内存使用情况。图 9 显示了每种类型的容器在单独运行时的平均内存占用情况。由于基于同源镜像的 Docker 容器可以共享二进制文件和库文件，因此 FRE 和 Docker 在单类型的容器内存占用方面差别不大。

为了验证多类型容器间内存共享的情况，我们在不同的时刻启动同类型的 Docker 和 FRE 容器实例，并保持容器实例运行，然后记录内存占用情况。同时为了能更全面的测量 FRE 容器的内存共享情况，我们还在相同配置的主机中直接启动上述 5 种应用来进行对比测试，结果如图 10 所示。从图中可以看出，由于非同源 Docker 容器间无法共享库文件和二进制文件，因此占用的内存与单独启动容器时的内存基本

相同。而 FRE 实现了容器间共享库文件、二进制文件和其他依赖包文件，所以实际占用内存要低，和物理机直接运行的内存占用基本一致。基于 FRE 启动的容器总内存占用量比基于 Docker 的方式共减少了约 15%，比直接主机启动的方式仅多占用了 1%的内存。

6.3. 镜像大小

较小的镜像体积将显著减少镜像构建和分发的时间，也能有效节约磁盘空间。FRE 运行环境仅包含应用函数所需的文件，所以具有更轻量的优势，我们将前面构建的 5 种类型的 FRE 镜像和 Docker 镜像进行文件大小对比，结果如图 11 所示。从图中可以看出，由于 FRE 运行环境仅包含应用函数所需的文件，所以镜像体积更小。相比于 Docker 镜像，FRE 镜像体积平均减少了 60%。

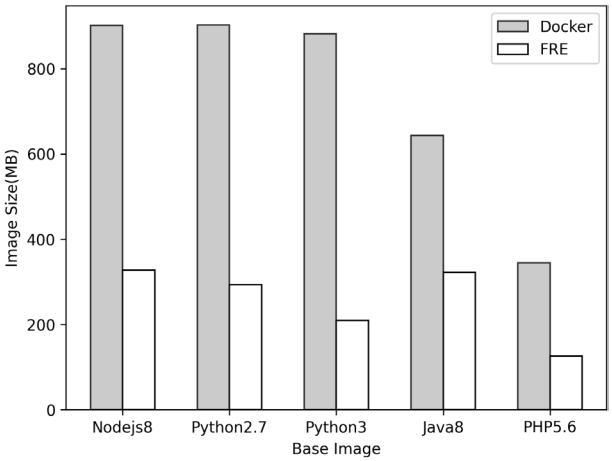

Figure 11. Comparison of basic image volume
图 11. 基础镜像体积对比

综上所述，本文提出的面向云函数的超轻量运行时环境在本地和实际应用测试中，相比于 Docker，其顺序和并发启动速度、内存利用率、镜像大小等方面均有显著的提升；相比于直接使用物理机的方式，性能损耗较低，验证了本文所提方法的有效性。

7. 总结与未来展望

本文提出了一种面向云函数的超轻量运行时环境构建方法，该方法通过进一步抽取共享层来提高资源共享度，通过降低网络隔离和控制组操作的性能瓶颈来加快启动速度。基于上述方法构建了超轻量运行时环境引擎 FRE，最后通过实验验证了 FRE 相较于主流的 Docker 容器引擎在顺序和并发启动速度、内存利用率、镜像体积等方面都有明显的提升。

本文实验未涉及到 FRE 引擎在网络、磁盘 I/O、CPU 利用率等方面的性能表现，这部分工作将在未来进行补充，同时，函数间通信、镜像安全性等问题也是将来的研究重点。

参考文献

[1] Hayes, B. (2008) Cloud Computing. *Communications of the ACM*, **51**, 9-11. https://doi.org/10.1145/1364782.1364786

[2] Wang, L., *et al*. (2018) Peeking behind the Curtains of Serverless Platforms. *USENIX ATC'*18 *Proceedings of the* 2018 *USENIX Conference on Usenix Annual Technical Conference*, Boston, MA, 11-13 July 2018, 133-145.

[3] Jonas, E., *et al*. (2019) Cloud Programming Simplified: A Berkeley View on Serverless Computing.

[4] Lloyd, W., *et al*. (2018) Serverless Computing: An Investigation of Factors Influencing Microservice Performance. 2018 *IEEE International Conference on Cloud Engineering* (*IC2E*), Orlando, 17-20 April 2018, 159-169.

https://doi.org/10.1109/IC2E.2018.00039

[5] OpenWhisk. https://openwhisk.apache.org

[6] Kubeless. https://kubeless.io

[7] 吴松, 王坤, 金海. 操作系统虚拟化的研究现状与展望[J]. 计算机研究与发展, 2019, 56(1): 58-68.

[8] Seo, K.T., Hwang, H.S., Moon, I.Y., *et al*. (2014) Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud. *Advanced Science and Technology Letters*, **66**, 105-111. https://doi.org/10.14257/astl.2014.66.25

[9] Merkel, D. (2014) Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, **2014**, Article No. 2.

[10] Namespace-Linux Namespace. https://man7.org/linux/man-pages/man7/namespaces.7.html

[11] Cgroups-Linux Control Groups. http://man7.org/linux/man-pages/man7/cgroups.7.html

[12] Harter, T., Salmon, B., Liu, R., *et al*. (2016) Slacker: Fast Distribution with Lazy Docker Containers. 14*th USENIX Conference on File and Storage Technologies*, Santa Clara, CA, 181-195.

[13] Thalheim, J., *et al*. (2018) Cntr: Lightweight OS Containers. 2018 *USENIX Annual Technical Conference*, Boston, MA, 11-13 July 2018.

[14] Wu, X.B., Wang, W.G. and Song, J. (2015) Totalcow: Unleash the Power of Copy-on-Write for Thin-Provisioned Containers. *Proceedings of the* 6*th Asia-Pacific Workshop on Systems*, Tokyo, July 2015. https://doi.org/10.1145/2797022.2797024

[15] Oakes, E., Yang, L., Houck, K., *et al*. (2017) Pipsqueak: Lean Lambdas with Large Libraries. 2017 *IEEE* 37*th International Conference on Distributed Computing Systems Workshops* (*ICDCSW*), Atlanta, 5-8 June 2017, 395-400. https://doi.org/10.1109/ICDCSW.2017.32

[16] Oakes, E., Yang, L., Zhou, D., *et al*. (2018) SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. 2018 *USENIX Annual Technical Conference*, Boston, MA, 11-13 July 2018, 57-70.

[17] Akkus, I.E., Chen, R., Rimac, I., *et al*. (2018) SAND: Towards High-Performance Serverless Computing. 2018 *USENIX Annual Technical Conference*, Boston, MA, 11-13 July 2018, 923-935.

[18] Engler, D.R., *et al*. (1995) Exokernel: An Operating System Architecture for Application-Level Resource Management. *ACM SIGOPS Operating Systems Review*, **29**, 251-266. https://doi.org/10.1145/224057.224076

[19] Madhavapeddy, A., *et al*. (2013) Unikernels: Library Operating Systems for the Cloud. *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 41, 461-472. https://doi.org/10.1145/2451116.2451167

[20] Manco, F., Lupu, C., Schmidt, F., *et al*. (2017) My VM Is Lighter (and Safer) than your Container. *Proceedings of the* 26*th Symposium on Operating Systems Principles*, Shanghai, October 2017, 218-233. https://doi.org/10.1145/3132747.3132763

[21] Shillaker, S. (2018) A Provider-Friendly Serverless Framework for Latency-Critical Applications. 12*th Eurosys Doctoral Workshop*, Porto, 23 April 2018.

[22] McGrath, G. and Brenner, P.R. (2017) Serverless Computing: Design, Implementation, and Performance. 2017 *IEEE* 37*th International Conference on Distributed Computing Systems Workshops* (*ICDCSW*), Atlanta, 5-8 June 2017, 405-410. https://doi.org/10.1109/ICDCSW.2017.36

[23] Dumazet, E. (2020) Re: Net: Cleanup Net Is Slow. https://lkml.org/lkml/2017/4/21/533

[24] Ferreira, J.B., Cello, M. and Iglesias, J.O. (2017) More Sharing, More Benefits? A Study of Library Sharing in Container-Based Infrastructures. In: *LNCS* 10417: *Proc. of the* 23*rd European Conf on Parallel Processing*, Springer, Cham, 358-371. https://doi.org/10.1007/978-3-319-64203-1_26

[25] Zhang, L.Q., *et al*. (2019) An Ultra Lightweight Container That Maximizes Memory Sharing and Minimizes the Runtime Environment. *Journal of Computer Research and Development*, **56**, 1545.