Computer Science and Application 计算机科学与应用, **2021, 11(12), 3136-3147**
Published Online December 2021 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2021.1112317

Hans汉斯

基于Init-Less模式的Serverless冷启动优化方法研究

刘　畅[1]，田春岐[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

收稿日期：2021年11月27日；录用日期：2021年12月23日；发布日期：2021年12月30日

摘　要

无服务器计算(Serverless)的兴起正为开发人员提供了更加有效的成本节约和弹性计算能力，该模式极大地提高了计算资源分配的灵活性，同时为真正按需租赁计算资源提供了可能。使得用户不必参与资源调度，自动实现无任何计算需求时不占用算力，同时在计算量攀升时及时扩容保证响应的及时性。然而目前基于容器虚拟化技术的无服务计算额外带来了冷启动问题，容器启动以及用户代码初始化会造成数秒的响应延迟，云计算厂商通常建议通过预留实例缓解此问题，但是这并不能消除容器扩容时的冷启动。同时，预留实例的方案也减弱了无服务计算相较于微服务节省成本的优势。现阶段多数研究目标集中于通过改造底层容器技术以尽量减少冷启动耗时，但是相对于广泛采用的Docker方案，难以在实际应用场景中对其进行取代。另一方面，仅仅对底层容器进行改造，无法减少用户代码初始化的额外耗时。本文分析了在无服务器计算场景下常规Docker容器应用的开发模式，提出了基于init-less和lazy-restore策略的Docker容器冷启动优化方法。该方法对用户代码的网络、文件、内存等计算资源的使用模式进行约定，基于CRIU技术捕获已初始化完毕的Docker容器的快照，并通过两阶段lazy-restore对常规Docker容器的启动流程进行替代。根据上述方法实现了docker-initless，绕过了容器应用的启动时间瓶颈，极大降低了Docker容器的冷启动延迟。实验从内存、文件资源等方面对比了Docker和docker-initless，验证了docker-initless在不额外预留计算资源的条件下对容器冷启动优化的有效性，同时可以保证对现有Serverless方案的兼容性。

关键词

无服务器计算，容器，快照还原，热迁移

Research on Serverless Cold Start Optimization Methods Based on Init-Less Mode

Chang Liu[1], Chunqi Tian[1], Wei Wang[2]

文章引用: 刘畅, 田春岐, 王伟. 基于 Init-Less 模式的 Serverless 冷启动优化方法研究[J]. 计算机科学与应用, 2021, 11(12): 3136-3147. DOI: 10.12677/csa.2021.1112317

[1]Department of Computer Science and Engineering, Tongji University, Shanghai
[2]School of Data Science & Engineering, East China Normal University, Shanghai

Received: Nov. 27th, 2021; accepted: Dec. 23rd, 2021; published: Dec. 30th, 2021

Abstract

The rise of Serverless computing (Serverless) is providing developers with more effective cost savings and flexible computing capabilities. This model greatly improves the flexibility of computing resource allocation and at the same time makes it possible to rent computing resources on demand. So that users do not need to participate in resource scheduling, automatically realize that no computing power is occupied when there is no computing demand, and at the same time, the capacity in time is expanded to ensure the timeliness of the response when the computing volume rises. However, the current Serverless computing based on container virtualization technology has additionally brought about cold start problems. Container startup and user code initialization will cause a response delay of several seconds. Cloud computing vendors usually recommend using reserved instances to alleviate this problem, but cold start cannot be eliminated during container expansion. At the same time, the reserved instance solution also reduces the cost-saving advantage of Serverless compared to microservice. At this stage, most research goals are focused on reducing the time-consuming on cold start by modifying the underlying container technology. But compared to the widely used Docker solution, it is difficult to replace it in actual application scenarios. On the other hand, merely modifying the underlying container cannot reduce the additional time-consuming initialize of user code. This article analyzes the development mode of conventional Docker container applications in Serverless computing scenarios, and proposes a cold start optimization method for Docker containers based on init-less and lazy-restore strategies. This method stipulates the usage patterns of computing resources such as network, files, and memory of user code, captures a snapshot of the initialized Docker container based on CRIU technology, and replaces the startup process of the conventional Docker container through a two-stage lazy-restore. According to the above method, docker-initless is realized, which bypasses the bottleneck of the startup time of container applications, and greatly reduces the cold start time of Docker containers. The experiment compares Docker and docker-initless in terms of memory, file resources, etc., and verifies the effectiveness of docker-initless in optimizing the cold start of containers without additional computing resources, while ensuring compatibility with existing Serverless solutions.

Keywords

Serverless Computing, Container, Snapshot Restore, Live Migration

Copyright © 2021 by author(s) and Hans Publishers Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

1. 引言

以 Docker 为代表的容器虚拟化技术极大地促进了云计算的发展，同时改变了传统上人们对于计算机软件技术的认知和使用习惯。近年来，由 Docker 技术演化而来的 Serverless 无服务器计算正越来越受到广泛的关注和应用，开发人员只需编写代码并将其部署在无服务器平台上，平台负责代码存储、执行、

网络和容错能力。目前，无服务器计算已应用于云计算、网络边缘的数据分析、科学计算和移动计算等方面。

容器作为一种新型的虚拟化技术，其在保证了应用级隔离性的基础上，实现了秒级的启动时间。由于与宿主机操作系统共享内核，因此容器在内存占用等指标上要比虚拟机技术所需要的硬件资源要小得多[1] [2]。此外受益于容器的快速启动等特性的支持，相较于预先启动的虚拟机，无服务器计算(Serverless [3] [4] [5])可以将计算资源的占用延迟到请求到来时进行，在资源使用的灵活度上拥有很大的优势[6] [7] [8] [9] [10]。Edwin 等人的成本分析表明，Serverless 可以显著节省成本：Serverless 版服务运行的成本可以达到服务进程常驻状态下成本的 0.85% [11]。但是对服务的响应时间有较苛刻需求的场景，例如应对大量并发请求的数据库应用，受限于启动时间对性能的过大影响，Serverless 尚不能良好地解决此类问题 [12] [13] [14] [15]。

通常情况下，Serverless 的响应时间包含了容器创建的固定开销以及容器内部应用程序的启动延迟。当容器所需要启动的程序较多或者运行时耗费的计算资源较高时，Serverless 服务的延迟将会变得不可预计，冷启动时间成为了 Serverless 大规模应用的一个瓶颈。Serverless 应用的启动时间优化，在学术上是一个很有研究价值的问题。

本文目标为解决单机 Docker 容器启动的性能优化问题，结合多种容器启动优化方法，探究一个新的容器启动流程优化模式，最终实现一个融合多种加速技术的容器启动性能优化系统。主要内容如下：

1) 在典型 Serverless 应用场景下，通过实验数据分析 Docker 容器启动整个流程在各阶段的耗时，验证了容器 namespace 创建和应用初始化是服务冷启动响应延迟的关键因素。

2) 提出了一种基于 init-less 策略的用户代码编程模式，通过将应用代码改写为 init-less 模式，使用 CRIU 技术捕获容器内存快照，以两阶段 lazy-restore 替代常规 Docker 容器的启动流程，最终降低 FaaS 冷启动响应的总耗时。

3) 基于以上研究实现了 docker-initless，在多个 Serverless 应用场景对冷启动情况下任务请求响应延迟进行对比实验，验证了此方法的有效性。

本文第 2 节介绍容器冷启动优化相关领域的工作，第 3 节介绍 init-less 策略用户代码编程模式的设计，第 4 节介绍 Docker 容器冷启动流程中影响请求响应时间的关键因素，第 5 节介绍 docker-initless 的原型实现，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

云计算厂商正尝试以提供编程 api 接口的形式提升 Serverless 服务的响应速度，编程开发人员可以将应用中负责进行初始化的代码放到容器预创建阶段执行[16]，以尽量减少在容器运行阶段的时间开销[17]。容器预创建是一个类似于 linux 中断上下两部分的优化思路，Docker 容器的生命周期包括了 create、start、run、stop 等阶段[18]。在 create 阶段，Docker Daemon 将会注册一个容器实例，并为其分配文件层。

通常情况下，Serverless 的任务具有可并行重复性，因此可以以相同的运行参数预先进行容器的创建[19]，形成一个“容器池”，此阶段并未进行造成实际的内存占用，在经济上是节约的。在 start 阶段，Docker 将完成对容器进程组的 cgroup 和 namespace 以及网络环境的设置[20]，在设置完成之后，将会最终执行容器进程组的启动。此阶段将会耗费至少数百毫秒的固定开销[21]，这在秒级的启动时间中有较大的优化空间。

CRIU (Checkpoint/Restore in Userspace)是 Linux 系统的一个软件工具，其功能是在用户空间实现 Checkpoint/Restore 功能。使用此技术可以冻结一个正在运行的程序，并且将其状态保存为一系列的文件，然后使用这些文件就可以在任何主机上重新恢复程序到被冻结的状态[22]。BLCR [23] (Berkeley Lab

Checkpoint/Restart)提供了用户态的libcr库和内核模块来完成相关的Checkpoint/Restore工作。DMTCP [24] (Distributed MultiThreaded Checkpointing)则以 library 库的形式实现对一个进程的 Checkpoint/Restore。gVisitor 也支持以 Checkpoint/Restore 的形式进行容器状态的保存和恢复[25]。

此外，Ranjan 等使用了改进的 VAS-CRIU [26]技术，来解决当 Docker 容器占用内存较高时 CRIU 对内存的捕获和还原性能开销过大的问题。CRIU 使用了 VFS 作为快照存储和读取的地址，但是这造成了读写文件系统的额外性能开销[27]。VAS-CRIU 使用了 COW [28] (Copy On Write)技术，直接以父容器内存为基准创建子容器[29]，将内存页的拷贝过程延迟到子容器执行内存写操作之时进行。由于避免了向文件系统进行内存页的序列化、反序列化操作，VAS-CRIU 的时间开销并不会随着容器所使用的内存量变大而增大。Dong Du 等提出的 Catalyzer [30]使用了 on-demand 的 restore 策略，将内存恢复和 io 重建延迟到启动流程的关键路径以后进行，使得容器应用不必等待暂未使用到的内存页和文件描述符恢复完毕即可就绪，可极大减少 restore 阶段的不必要耗时。

3. Init-Less 模式介绍

在常规的 Serverless 服务中，延迟敏感型任务是本文研究的重点。云计算厂商提供的 FaaS 是目前 Serverless 的工程实现，而 FaaS 服务常以 Docker 作为底层容器支持。FaaS 提出了触发器这一概念作为对任务事件的抽象，典型的触发器类型包括定时任务、对象存储、http 请求等，其中 http 型触发器对响应延迟的敏感性要显著高于其余类型的任务。

http 型 Serverless 服务的目标是保证在无任何请求处理时，无需维持 Docker 容器实例运行，在请求到来时才进行容器实例的创建。一个典型的 http 型 Serverless 服务包含这些项目：

1) Bootstrap 启动脚本。Serverless 引擎通过在容器内部执行 Bootstrap 脚本来启动用户进程。

2) 文件目录挂载。用户进程以此来读写保存在外部文件系统中的二进制码、依赖库、数据文件等。

3) 网络端口映射。通过约定一个 tcp 端口如 9000，使得 Docker Daemon 可以映射任意宿主端口到容器内。

4) 内存规格限制。Docker Daemon 基于 Cgroup 提供计算资源隔离性，使得容器内用户进程不会消耗过多宿主机内存。

3.1. FaaS 模式

云服务厂商提出了可在较小编程改动的条件下将现有微服务改造为 Serverless 服务的方法，称为 FaaS 模式，FaaS 模式演化自基于 state-less 模式的微服务。

state-less 模式的原则是从源头上避免问题的产生。一个微服务应用实例在处理请求时不应使用本机计算资源来缓存计算时产生的上下文数据，而是由统一的有状态服务来进行存储，以避免不同实例间状态数据不一致的问题。

FaaS 模式相对于 state-less 模式额外限制了进程的驻留时间，使得 Serverless 引擎可以以毫秒级时间片为粒度进行计算资源的分配。通常约定容器实例在不处理请求时为冻结状态，不占用任何计算资源，即不可使用后台线程在处理请求之外的任何时间运行代码。

微服务常由支持长连接的 rpc 协议对外暴露 tcp 端口，而 FaaS 模式通常仅支持编写短连接的 http 服务，在使用数据库等外部服务时也需要被调用方针对短时间内多次建立短连接进行配置调优。

3.2. FaaS 的 AOT 优化

现有研究表明，在 FaaS 服务中，适用于 python、c、go 等轻型 runtime 语言的冷启动优化方法难以

对重型 runtime 语言如 java、dotnet 等产生有效作用，一个最基本的 Spring Boot 项目初始化可能会耗时长达 2 s。基于 init-less 策略的 AOT (Ahead of Time)优化是解决此类问题的关键，这也是虚拟机冷启动时间优化的成功经验。

AOT 优化的核心思想是收集容器已初始化完毕后对计算资源的占用信息，将进程元数据如寄存器、内存页、文件描述符等序列化为二进制文件保存。将捕获的快照以高吞吐量的磁盘顺序读的方式写入内存，避免在运行时进行初始化所带来的耗时操作，以达到冷启动优化的效果。

不同于虚拟机与宿主机计算资源的高度隔离性，容器本质上为宿主机内的一组进程。CRIU 技术对于进程 checkpoint 和 restore 前后运行环境的一致性较为严格，在 restore 时主要困难为：

1) 对于进程已打开的文件，需要进行 redo-open 操作。如果此时有任何一个文件已被其它进程修改，那么 restore 会失败。

2) 进程使用的 ip 端口不可被其他进程占用。

3) 进程使用的 pid 不可被其他进程占用。

4) 进程使用的 tty 终端必须可被重连。

5) 进程建立的 socket 连接的对等端 ip 必须可被重新访问。

而 Docker 容器对于 ip 端口和 pid 等使用了 namespace 机制，因此 CRIU 在对容器进程进行 restore 时可以比非容器进程获得更好的兼容性。同时，由于容器进程在启动时并不与任何 tty 终端绑定，也无需关注此类冲突问题。此外，Docker 容器使用了 overlayfs 进行文件目录和挂载点的隔离，这也使得单个进程快照在 restore 为多个实例时所带来的文件冲突问题得到了很好的解决。但是对于容器进程打开的文件和建立的 socket 连接，需要有一种新的编程模式进行约定。

3.3. Init-Less 模式

为了使得容器可被 checkpoint，以及在 restore 后能够响应请求，init-less 模式在 FaaS 模式的基础之上额外增加了这些限制：

1) 对于在进程初始化完毕后打开的文件，若为多实例公共读模式则应始终保持只读属性，不可为公共写模式。或者进程只在处理请求时才打开文件进行 io 读写操作，并在初始化过程中不打开任何文件。

2) 对于在进程初始化完毕后主动建立的 tcp 长连接，在连接不可用时有重试逻辑。或者进程只在处理请求时才使用短 tcp 连接访问外部服务，并在初始化过程中不建立任何 tcp 连接。

3) 容器在初始化完毕后，需要向 stdout 打印初始化完毕短语，以表示容器此时可被 checkpoint，而不可使用常规的 http 健康检查方法探测服务是否已就绪。

init-less 模式的提出解决了何种类型的 FaaS 服务可被改进以适应 checkpoint 和 restore 的问题，也同时给出了改造服务的具体思路。基于 init-less 模式，我们可以编写出能够被 docker-initless 良好支持的 FaaS 服务。

3.4. 小结

相对于有状态计算模式，state-less 模式消除了对实例内存数据一致性的依赖，FaaS 模式额外消除了实例闲置时对计算资源的依赖，init-less 模式进一步消除了对实例网络资源、外部文件不可变性的依赖。

4. 容器冷启动耗时分析

由于 Docker 使用 runc 进行容器的创建，本文通过在 runc 源代码中插入计时代码片段，对 Docker 容器启动的各个阶段进行了耗时统计。

4.1. 容器创建阶段

在容器创建阶段，runc 会进行 init 进程初始化设置，其中创建网络 namespace 耗时平均为 181.492 ms，创建 Cgroup 为 0.381 ms。此阶段主要为 Docker 容器的网络、内存、CPU 等计算资源进行额度分配，并不会产生真正的资源占用。

由于 FaaS 通常为单服务多 Docker 容器实例模式，因此可以使用容器预加载机制，形成一个容器池，在有任务需要处理时从容器池中取出 pre-start 的实例进行派发，可以在整个请求－响应流程中节省 300 ms 的固定耗时。

4.2. 请求响应阶段

此阶段完成 Docker 容器计算资源的额度分配和隔离，其流程为：init 进程执行 Bootstrap 创建用户进程，等待用户进程初始化完毕后，将 Docker 宿主端口收到的 http 请求转发到容器内部 9000 端口，再将响应结果返回。我们分为常规场景和基于 init-less 模式的 lazy-restore 场景分别分析。

4.2.1. 常规场景

以 java 的空 Spring Boot 项目为例：在用户代码执行之前需要进行 JVM 初始化和依赖库文件加载，此过程可长达 1.89 s。此后才会进行 http 请求的处理，响应平均耗时为 99.351 ms。在 FaaS 容器冷启动场景下，处理 http 请求耗时可能会达到微服务场景下的 22.7 倍。

4.2.2. Lazy-Restore 场景

Docker 已集成了 CRIU 作为容器 C/R 的底层支持。但是由于容器在代码编程模式没有进行限制时难以被正常 C/R，因此这个特性长期处于 experimental 状态没有被默认开启。此外，Docker 只使用了 CRIU 的一阶段 restore。本文基于 CRIU 的 lazy-pages 特性对 Docker 容器进行了两阶段的 lazy-restore，使得进程并不需要内存页完全恢复完毕即可提前运行。

lazy-pages 基于 Linux 的 userfaultfd 技术，对于进程预分配的内存，在产生访问并发生缺页中断时可由进程自行处理。此时 CRIU 将访问 page-server 并通过 rpc 读取内存页，实现按需恢复的特性。

Table 1. Resulting data of Flask experiment
表 1. Flask 实验结果数据

快照内存规格(GB)	容器创建耗时(s)	CRIU lazy-restore 耗时(s)	lazy-restore 场景 http 请求耗时(s)	CRIU restore 耗时(s)
1.3	0.349	0.356	0.704	1.402
2.5	0.348	0.362	0.715	2.151
5	0.350	0.368	0.723	4.003
9.8	0.371	0.400	0.785	7.135

如表 1 所示，以一个 python 的 Flask 项目为例，本文通过在进程初始化时创建一个包含上千万个 int 键值对的字典对象，并使用 lazy-restore 方法进行测试。实验分别模拟了 1x、2x、4x、8x 的大内存规格容器，并在每次请求时按一个随机的 key 访问不同地址的内存页。

lazy-restore 时请求总耗时与具体的容器任务类型有关，按需 restore 的策略在容器快照体积较大且每次处理请求不需完全访存时可以取得较为恒定的响应时间，且总是优于非 lazy-restore 场景。

5. Docker-Initless 设计

docker-initless 是由 Docker 修改得到的，专用于优化基于 init-less 模式开发的 http 容器应用冷启动时

间的 FaaS 系统。docker-initless 的架构如图所示，其主要分为 Core Controler、Checkpoint Manager、Docker Daemon 共 3 个部分。

5.1. Docker Daemon

Docker Daemon 基于 Docker CE 17.03 版本，是 docker-initless 的底层容器支持，保持了对原生 Docker 的兼容。docker-initless 修改了此版本所使用的 runc 和 containerd，并调整了 dockerd 的配置文件。

5.1.1. Dockerd

docker-initless 开启了 experimental 特性，以使用基于 CRIU 的 checkpoint 功能。此外将容器底层文件系统由 overlayfs 更改为 btrfs，基于 inode 级别的 COW 特性，可极大降低容器文件的写时复制耗时，同时减少冗余文件的空间占用。

5.1.2. Runc

不同于常规情况下 runc 启动容器的流程，在 docker-initless 中 runc 使用了 CRIU 的 swrk 模式来进行容器快照的还原。CRIU 将进程组的 restore 共分为 network-unlock、setup-namespaces、post-setup-namespaces、post-restore 等 4 个阶段。在 swrk 模式中，由 runc 开启一个 socket 并创建 CRIU 子进程，runc 以 rpc 的形式对 CRIU 进行调用，CRIU 在 restore 各个阶段完成时返回响应，runc 在此时进行 hook 函数的执行。

针对 lazy-restore 策略，docker-initless 额外增加了 rpc 请求中对 lazy-pages 的支持，使得 CRIU 可以使用懒加载的模式进行进程 restore，在内存页恢复完毕之前即可进入 post-restore 阶段，并使 Core Controler 能够尽快进行 http 请求的分配。

针对 pre-start 策略，在 post-setup-namespaces 阶段，runc 将通过容器 id 在 redis 中获取 Core Controler 预设的端口号，并开启一个 tcp server 在 localhost 上对此端口进行监听。一旦收到来自 Core Controler 的 tcp 连接请求，runc 将终止等待状态，发送 rpc 请求调用 CRIU 进入 restore 的下一阶段。

5.1.3. Containerd

将 containerd 的容器启动超时时间由默认的 2 分钟增加到 30 分钟，这个调整是针对 pre-start 策略的优化。pre-start 策略要求 docker-initless 维护一个处于 pre-start 状态的容器池，由于在 rpc 响应超时机制中超过 2 分钟 containerd 则放弃等待并返回超时错误，因此过短的超时时间将会导致容器的频繁新建和销毁，我们需要对其进行调整。

5.2. Checkpoint Manager

Checkpoint Manager 是一个用于创建容器快照的 Python 脚本，容器快照的制作流程可由约定格式的 Dockerfile 限定，主要规则为：

1) Dockerfile 中必须通过 ENV 指定 CHECKPOINT_MSG。即容器内应用初始化完毕后向 stdout 打印的字符串短语。

2) Dockerfile 中必须通过 ENV 指定 RW_DIRS，且必须为 VOLUME 的子集，即容器挂载的私有读写目录。这些目录将在容器快照捕获时由 Checkpoint Manager 进行复制备份，但不包括容器挂载的只读目录。

由于 Dockerfile 只用于制作镜像，用户需要提供一个 docker create 命令用于在宿主机上进行容器实例创建。容器快照的制作流程为：

1) 使用 build 命令进行容器基础镜像的构建。

2) 使用用户提供的 create 命令和已构建好的基础镜像进行容器创建。

3) 使用 start 命令开启容器，并执行用户提供的 Bootstrap 脚本，进行用户代码的初始化。

4) 循环使用 info 命令捕获容器的 stdout 信息，判断是否已打印 CHECKPOINT_MSG。如果 CHECKPOINT_MSG 已被打印，则进行下一步。

5) 对所有 RW_DIRS 进行文件复制备份，在复制的同时保留文件的全部属性。所有的文件将在 restore 时复制出一份新的版本以进行挂载。

6) 使用 commit 命令将此时的容器保存为新的容器镜像。这一步是为了保持容器应用内临时文件的一致性，而不是直接使用第 1 步中构建的镜像。

7) 使用 checkpint create 命令，获得 CRIU 捕获的此时容器运行时快照。包含内存页的二进制快照将会在 restore 时由 page-server 读取。

5.3. Core Controler

Core Controler 是由 Go 编写的 FaaS 引擎，也是 docker-initless 的主控模块。每个 FaaS 服务都由一个容器池对 http 请求进行处理，Core Controler 负责了该容器池的创建、维护和请求分发。在具体实现上，对于每个已预制作容器快照的服务，Core Controler 会使用 pre-start 策略进行容器预创建，并将预定的端口号保存在 redis 中。在容器创建时会为每个实例单独开启一个 CRIU 的 page-server，以便配合 CRIU 的 lazy-pages 特性使用。并为该实例单独复制一份私有目录文件，基于 btrfs 的 COW 特性，只需在复制时设置 reflink 属性为 always 即可。pre-start 的容器将会处于 post-setup-namespaces 阶段最多 30 分钟，如果期间一直没有 http 请求需要处理，将会被 Core Controler 销毁并重新创建。

Core Controler 会对外暴露一个 tcp 端口以供 http 请求的访问。当容器池内并无已完全启动的实例时，一旦有 http 请求到来，便从容器池内选择一个实例进行 CRIU 的下一阶段 restore。基于 lazy-restore 策略，CRIU 会在容器快照内存页申请完毕并设置为 userfaultfd 后立即进入 post-restore 阶段，由 page-server 配合完成后续缺页中断的处理。此时 Core Controler 可将 http 请求转发给容器实例，并等待响应结果返回，最终完成整个 http 请求－响应流程。

已冷启动完毕的容器实例将会被 Core Controler 优先分配 http 请求，直到在一定时间内没有 http 请求需要处理最终被超时销毁为止。若单个容器实例达到请求并发能力上限，将会触发 Core Controler 进行扩容冷启动。

6. 实验结果与分析

实验环境为：8 核 AMD Ryzen R7 3700X 4.20 GHz 处理器；64 GB DDR4 内存；512 GB 固态硬盘；操作系统为 Ubuntu 18.04，内核版本为 5.10.60.1-microsoft-standard-WSL2；Docker 为 Community 17.03.2 版本。

实验结果包含了两个阶段的时间：Boot 阶段为 Docker Daemon 收到 http 请求开始创建容器，到容器内应用可处理 http 请求为止；Execution 阶段为容器内应用开始处理 http 请求，到 Docker Daemon 将 http 响应结果返回为止。

实验包括了三个典型的 FaaS 服务子实验，其源代码均以 init-less 模式编写，并且提供了 Dockerfile 以便 docker-initless 测试。实验的流程为：

1) 使用 Checkpoint Manager 制作 FaaS 服务的容器快照和镜像。

2) 使用原版 Docker 创建容器，统计两阶段耗时。

3) 关闭 pre-start 和 lazy-pages 特性，使用 docker-initless 基于容器快照创建实例，统计两阶段耗时。

4) 开启 pre-start 和 lazy-pages 特性，使用 docker-initless 基于容器快照创建实例，统计两阶段耗时。

5) 将第 1 步制作的容器快照放在使用高速内存创建的 tmpfs 文件系统中，开启 pre-start 和 lazy-pages 特性，使用 docker-initless 基于容器快照创建实例，统计两阶段耗时。

6) 重复第 2~5 步各 1000 次，对耗时取平均值。

6.1. Flask

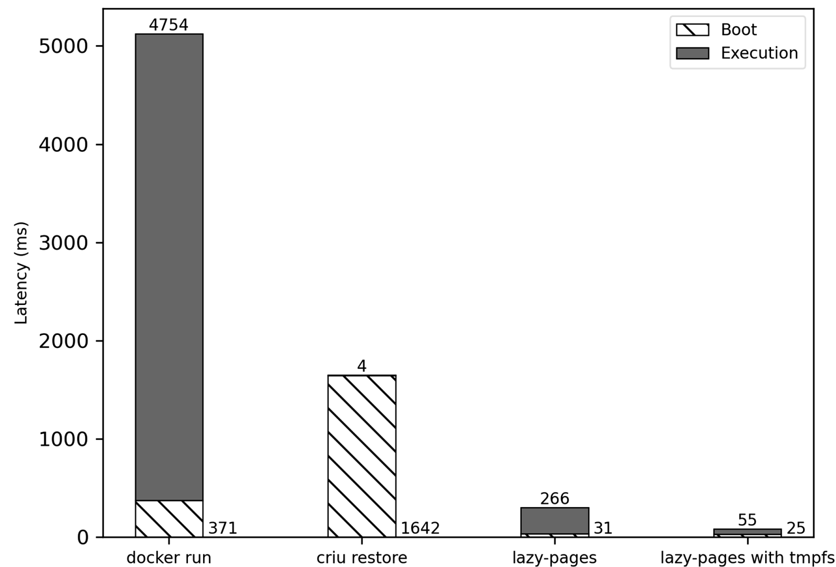

Figure 1. Resulting data of Flask experiment
图 1. Flask 实验结果数据

如图 1 所示，本实验场景为一个内存消耗约 1.3 GB 的 Flask 应用，在其初始化的过程中需要创建一个包含一千万个 int 键值对的字典对象。实验发送的 http 请求会随机访问其中的一个键值对，以测试按需 restore 的优化效果。

根据实验结果，在不额外使用优化策略的情况下，init-less 模式相较于普通 FaaS 模式在总响应时间上取得了 3.11x 的优化效果。主要收益来源于对 Python 大内存对象初始化的 AOT 优化，通过以 init-less 策略取代 init 过程来节省耗时。

在使用了 lazy-pages 的对比实验中，pre-start 策略消除了 Boot 阶段约 92%的耗时，主要收益来源于提前进行了 namespace 和 Cgroup 等容器环境的初始化；lazy-restore 策略也在 Execution 阶段取得了 17.82x 的优化效果。同时在基于高速内存的 tmpfs 的加速下，总的响应时间最终得到了 64.06x 的优化。

6.2. Spring Boot

如图 2 所示，本实验场景为一个输出 Hello World 的 Spring Boot 项目，在初始化的过程中需要启动 JVM 并加载依赖库文件，最终产生约 300 MB 的内存占用。本实验模拟了带有重型 runtime 的 FaaS 服务的一般情况。

与 Flask 实验类型，在不额外使用优化策略的情况下，init-less 模式相较于 FaaS 模式在总响应时间上取得了 3.17x 的优化效果。由 docker run 场景可知，Docker 创建空容器耗时约为 364 ms；由 criu restore 场景可知，Spring Boot 在完全初始化完毕时，其 Execution 阶段固定耗时约为 99 ms；因此可认为在 criu restore 场景中 CRIU 恢复快照耗时约为 611 − 364 = 247 ms，而在 docker run 场景中 JVM 和 Spring Boot 库初始化耗时约为 1888 − 99 = 1789 ms，init-less 模式相较于 init 流程可取得 7.24x 的加速比。

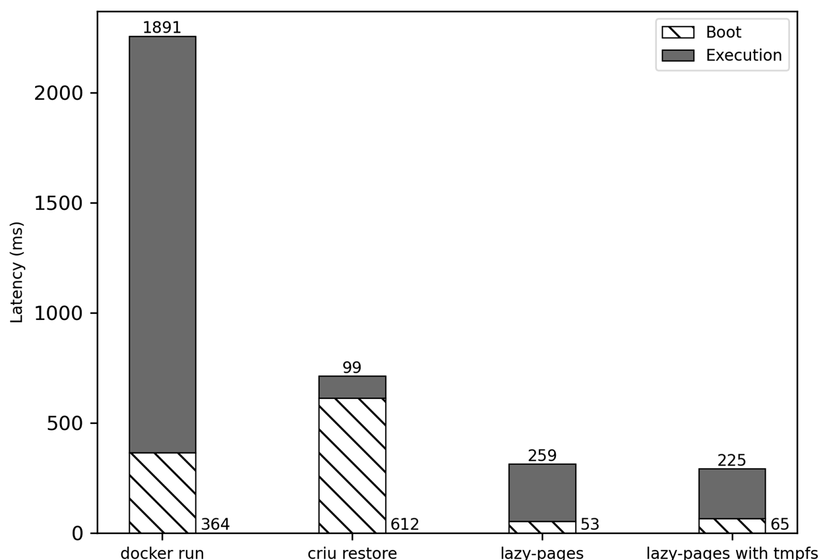

Figure 2. Resulting data of Spring Boot experiment
图 2. Spring Boot 结果数据

由 lazy-pages 场景可知，在使用了 pre-start 策略后，在 Boot 阶段可取得平均约 364 − 53 = 311 ms 的耗时优化。而额外的 611 + 99 − (257 + 53) − 311 = 89 ms，是 lazy-restore 策略在 Execution 阶段取得的耗时优化，这表明快照中由于懒加载并未在 Execution 阶段产生缺页中断的内存页占比约为 89/247 = 36.03%。

由于 Spring Boot 实验的内存快照体积相比于 Flask 较小，CRIU 在 restore 阶段并未产生过多磁盘 io，因此将文件系统改为 tmpfs 并没有取得显著优化效果，使用高速内存前后的加速比分别为 7.26x 和 7.93x。

6.3. Rembg

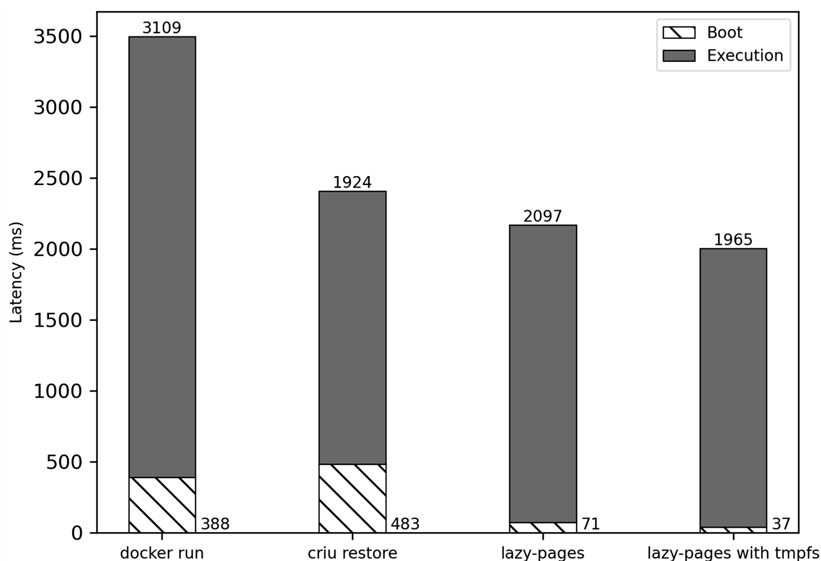

Figure 3. Resulting data of Rembg experiment
图 3. Rembg 结果数据

如图 3 所示，实验场景为一个为图片人像去除背景的 Pytorch 应用，在初始化的过程中需要约 160 MB 的模型文件并在计算时加载约 1 GB 的依赖库。本实验模拟了 CPU 密集型 FaaS 服务的一般情况，根据阿姆达尔定律，此类计算加速比的提升效果取决于固定耗时的占比。

即使 Rembg 在初始化完毕的情况下，完成一次请求−响应耗时仍需约 2 s，而使用 init-less 模式并不能减少这一部分的耗时。因此在本实验中毫秒级的耗时优化对加速比提升的贡献较小，但最终仍然取得了最高约 1.75x 的优化效果。

6.4. 小结

在多种 FaaS 服务场景下，docker-initless 均可将容器冷启动阶段耗时由数秒优化至数百毫秒级别，并且验证了 docker-initless 在使用计算资源方面的经济性，同时还可以保证对现有 Serverless 方案的兼容性。

7. 总结与未来展望

本文提出了一种面向 Serverless 服务冷启动优化的 init-less 编程模式，并基于该模式实现了一个集成 pre-start 和 lazy-restore 策略的 FaaS 系统 docker-initless。该系统在多个典型实验场景下对 FaaS 服务的冷启动响应时间有了显著改善效果，同时在成本上保持经济性。

本文实验并未涉及在恢复内存快照时使用写时复制技术，这部分的工作将会是未来的研究重点。

参考文献

[1] Zhao, N., Tarasov, V., Anwar, A., *et al*. (2019) Slimmer: Weight Loss Secrets for Docker Registries. 2019 *IEEE* 12*th International Conference on Cloud Computing* (*CLOUD*), Milan, 8-13 July 2019, 517-519. https://doi.org/10.1109/CLOUD.2019.00096

[2] 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

[3] Amazon. AWS Lambda-Serverless Compute. https://aws.amazon.com/lambda

[4] Microsoft. Azure Functions Serverless Architecture. https://azure.microsoft.com/en-us/services/functions

[5] Google. Google Cloud Function. https://cloud.google.com/functions

[6] Jonas, E., Pu, Q., Venkataraman, S., *et al*. (2017) Occupy the Cloud: Distributed Computing for the 99%. *ACM Proceedings of the* 2017 *Symposium on Cloud Computing*, Santa Clara, 24-27 September 2017, 445-451. https://doi.org/10.1145/3127479.3128601

[7] Lynn, T., Rosati, P., Lejeune, A., *et al*. (2017) A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms. 2017 *IEEE International Conference on Cloud Computing Technology and Science* (*CloudCom*), Hong Kong, 11-14 December 2017, 162-169. https://doi.org/10.1109/CloudCom.2017.15

[8] Wang, L., Li, M.Y., Zhang, Y.Q., Ristenpart, T. and Swift, M. (2018) Peeking behind the Curtains of Serverless Platforms. 2018 *USENIX Annual Technical Conference* (*USENIX ATC* 18), Boston, 11-13 July 2018, 133-146.

[9] Brewer, E. (2015) Kubernetes and the Path to Cloud Native. *Proceedings of the Sixth ACM Symposium on Cloud Computing*, Kohala Coast, 27-29 August 2015, 167-167. https://doi.org/10.1145/2806777.2809955

[10] Lloyd, W., Ramesh, S., Chinthalapati, S., *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

[11] Boza, E.F. andrade, X., Cedeno, J., *et al*. (2020) On Implementing Autonomic Systems with a Serverless Computing Approach: The Case of Self-Partitioning Cloud Caches. *Computers*, **9**, 14. https://doi.org/10.3390/computers9010014

[12] Jonas, E., Pu, Q.F., Venkataraman, S., Stoica, I. and Recht, B. (2017) Occupy the Cloud: Distributed Computing for the 99%. *Proceedings of the* 2017 *Symposium on Cloud Computing*, Santa Clara, 24-27 September 2017, 445-451. https://doi.org/10.1145/3127479.3128601

[13] Hellerstein, J.M., Stonebraker, M., Hamilton, J., *et al*. (2007) Architecture of a Database System. *Foundations and Trends R in Databases*, **1**, 141-259. https://doi.org/10.1561/1900000002

[14] Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., *et al*. (2013) Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems* (*TOCS*), **31**, 8. https://doi.org/10.1145/2518037.2491245

[15] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernandez Moctezuma, R.J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., *et al*. (2015) The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment*, **8**, 1792-1803. https://doi.org/10.14778/2824032.2824076

[16] Knative: Kubernetes-Based Platform to Build, Deploy, and Manage Modern Serverless Workloads. https://cloud.google.com/knative

[17] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. and Arpaci-Dusseau, R. (2018) SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. 2018 *USENIX Annual Technical Conference* (*USENIX ATC* 18), Boston, 11-13 July 2018, 57-70.

[18] Hellerstein, J.M., Faleiro, J., Gonzalez, J.E., *et al.* (2019) Serverless Computing: One Step Forward, Two Steps Back. arXiv preprint arXiv:1812.03651

[19] Wagner, T.A. (2018) Acquisition and Maintenance of Compute Capacity, September 4. US Patent 10067801B1.

[20] Docker, M.D. (2014) Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, **2014**, 2.

[21] Baldini, I., Castro, P., Chang, K., *et al*. (2017) Serverless Computing: Current Trends and Open Problems. In: *Research Advances in Cloud Computing*, Springer, Singapore, 1-20. https://doi.org/10.1007/978-981-10-5026-8_1

[22] CRIU Community (2019) Checkpoint/Restart in Userspace (CRIU). https://criu.org

[23] Hargrove, P.H. and Duell, J.C. (2006) Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. *Journal of Physics Conference Series*, **46**, 494. https://doi.org/10.1088/1742-6596/46/1/067

[24] Ansel, J., Arya, K. and Cooperman, G. (2007) DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop.

[25] (n.d.) Checkpoint/Restore in gVisor. https://gvisor.dev/docs/user_guide/checkpoint_restore

[26] Venkatesh, R.S., Smejkal, T., Miloji, D.S. and Gavrilovska, A. (2019) Fast In-Memory CRIU for Docker Containers. *ACM Proceedings of the International Symposium on Memory Systems*, Washington DC, 30 September-3 October 2019, 53-65. https://doi.org/10.1145/3357526.3357542

[27] Gioiosa, R., Sancho, J.C., Jiang, S., Petrini, F. and Davis, K. (2005) Transparent, Incremental Checkpointing at Kernel Level: A Foundation for Fault Tolerance for Parallel Computers. *Proceedings of the* 2005 *ACM/IEEE Conference on Supercomputing*, Seattle, 12-18 November 2005, 9.

[28] Li, Y.W. and Lan, Z.L. (2011) FREM: A Fast Restart Mechanism for General Checkpoint/Restart. *IEEE Transactions on Computers*, **60**, 639-652. https://doi.org/10.1109/TC.2010.129

[29] Plank, J.S., Beck, M., Kingsley, G. and Li, K. (1994) Libckpt: Transparent Checkpointing under Unix. Computer Science Department.

[30] Du, D., Yu, T.Y., Xia, Y.B., Zang, B.Y., Yan, G.L., Qin, C.G., Wu, Q.X. and Chen, H.B. (2020) Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne, 16-20 March 2020, 467-481.