AWS的“炮仗”与Serverless

2019-07-12 阅读 254

背景

Serverless Computing，即”无服务器计算”，这一概念在刚刚提出的时候并没有获得太多的关注，直到2014年AWS Lambda这一里程碑式的产品出现。通过将无服务器计算的概念嵌入到整个云计算服务的整体产品框架中，无服务器计算正式走进了云计算的舞台。2017年，AWS发布了Fargate产品以充实自己的无服务器计算产品线。

今年5月，Google在KubeCon+CloudNative 2018期间开源了gVisor容器沙箱运行时并分享了它的设计理念和原则。随后，今年7月，Google在旧金山举办了2018年度Google Next大会，在这次大会上，Google推出了自己的 Google Serverless Platform。针对App Engine，最重要的更新就是低层的沙箱技术采用了gVisor。当然，我们有足够的理由相信Google指的是gVisor的内部实现版本。

今年的re:Invent 2018上，AWS点（kai）燃（yuan）了Firecracker —— AWS容器安全沙箱的基础组件，用于函数计算服务AWS Lambda和托管的容器服务AWS Fargate[1] [7]。

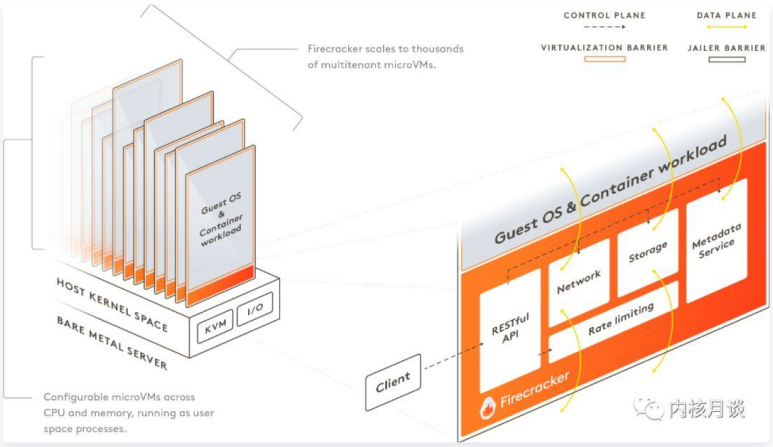


图1 Firecracker microVM

Firecracker利用了Linux KVM来构建专门用于容器的微虚拟机，即Firecracker microVM。并力图提供一种针对容器的，同时满足了安全隔离、性能稳定、高资源利用率的方案。AWS的首席“传教士”Jeff Barr称它：即提供了传统虚拟机对业务负载的安全与隔离特性，也带来了像使用容器一样高效的资源利用率。

Firecracker派生自Crosvm[2] —— 用Rust编写的、开源的、用于Chromium OS的Virtual Machine Monitor。基于Crosvm，AWS于2017年10月开始了Firecracker的研发。但与Crosvm的目标不同，Firecracker聚焦于Serverless，即：专为无服务器计算场景提供安全高效的运行时。近些年，系统安全越发受到重视，Rust语言也变的越来越流行。Firecracker可能也是Rust语言在生产环境中部署的，规模最大的系统软件。

Firecracker目前还没有实现与Docker及Kubernetes对接。但是AWS同时开源了一个对接containerd的原型[9]，并表示未来一定会和Kubernetes兼容。

根据AWS的说法，Firecracker微虚机可以在每个主机上以每秒150个实例的速率，在125ms内启动。并宣称VMM组件的内存开销小于5MiB（注：不包括客户内存，vCPU线程占用的内存，和控制平面上API Server线程占用的内存）。因此，可以在一台服务器上部署成百上千个微虚机。

2. AWS Lambda的演进与Firecracker的诞生

作者介绍

Linux阅码场

关注 专栏

文章	阅读量	获赞	作者排名
213	41.8K	491	1148

目录

Firecracker目前已经用在AWS无服务器计算业务中，包括AWS Lambda和AWS Fargate。AWS认为，使用无服务器计算服务的用户负载的典型特点是“生命周期短”，而Firecracker专为这种场景打造。让我们看一下，Firecracker是如何支撑AWS Lambda的。

Firecracker诞生的内因是AWS Lambda的演进，而要了解Lambda的演进，就需要看一下Lambda对用户请求的执行过程和执行环境。如下图所示，用户请求通过“ALB”转发给“Front End”，“Front End”请求“Worker Manager”，“Worker Manager”初始化“Worker”，“Worker”准备函数沙箱执行环境，完成后，将状态原路返回给“Front End”，然后由“Front End”触发函数执行。

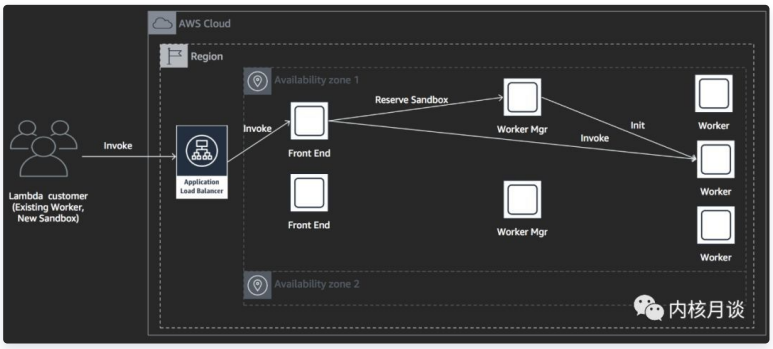


图2 AWS Lambda 执行过程

用户函数运行在“Lambda Runtime”中，在其之下是沙箱。与Linux中跑容器时常用的套路一样，使用了cgroups，namespaces，seccomp，iptables，和chroot等一些列工具以实现操作系统层级上的虚拟化（也称为“容器化”）[11]。再往下一层，是实现安全隔离的重点，即虚拟化技术与设备模拟。全栈如下图所示：

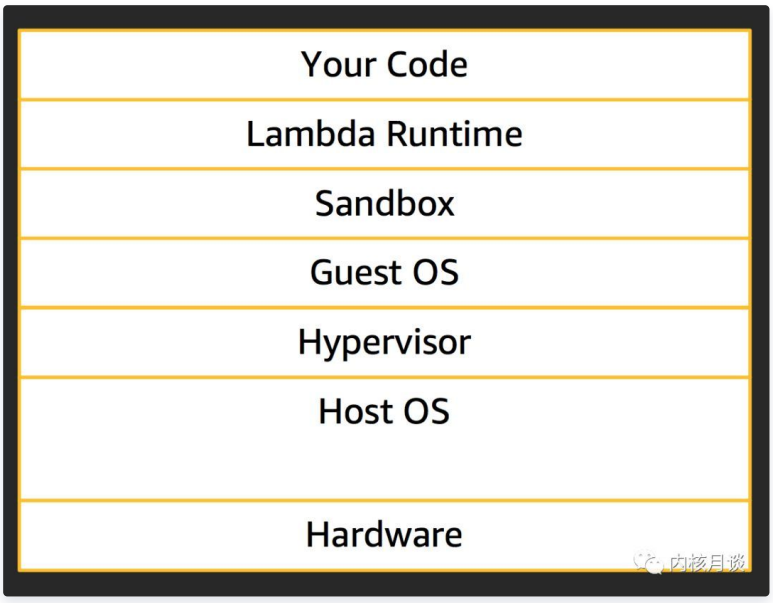


图3 AWS Lambda 执行环境

当AWS刚开始打造Lambda服务时，它始于在一个EC2实例中构建每一个“Worker”。原因很直接：

- 很好的安全边界；
- 快速构建好整个系统使业务上线；

这种方式今天依然在使用，并且运行在Nitro平台上面。

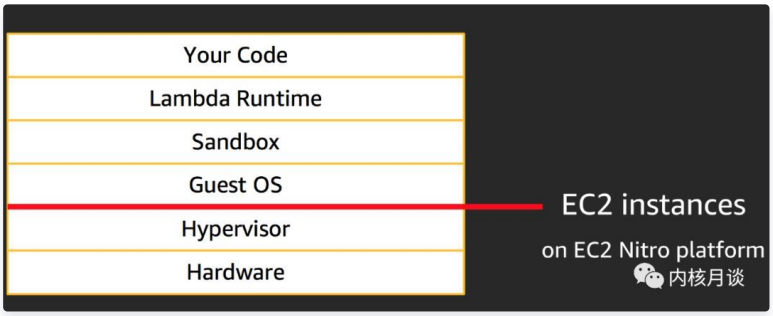


图4 基于EC2实例的AWS Lambda

通过AWS Lambda长期以来的生产实践和客户的需求反馈，AWS意识到，基于EC2实例的Lambda并不适合今天的无服务器计算场景。并总结出无服务器计算的典型特征应该是：**“启动快，密度高，水平扩展”**。但要达到以上这三个点，不能损失一点安全性。基于这些因素，AWS决定对Lambda进行改进，并在此过程中开发了Firecracker微虚机。由此，AWS Lambda有了另一种跑在微虚机中的“Worker”。

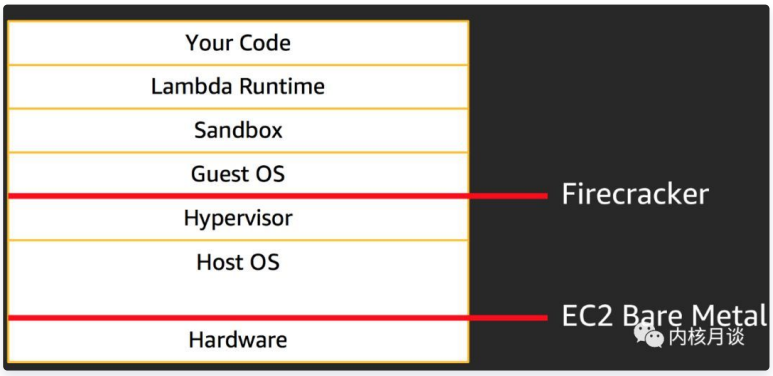


图5 基于Firecracker的AWS Lambda

为了进一步加固安全隔离，AWS在微虚机外面又套了一层沙箱（使用运行容器时常用的工具）。由此可见，安全隔离是对外提供服务的基本前提。

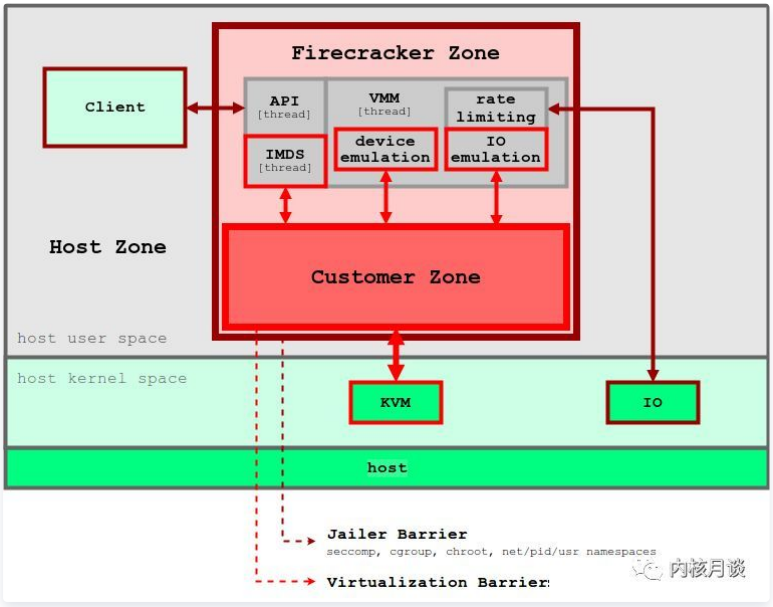
当启动变快，内存开销变低时，实例部署密度也自然有了更大的提升空间。但实际上，实例部署密度不仅与CPU、内存相关，还涉及到与业务相关的一整套资源，比如：ENI网卡，IP地址资源等。随着部署密度从一百提升到一千甚至更高的时候，相关资源的供给及使用的问题随之而来。

当Lambda创建和启动一个函数服务时，它需要经历在用户VPC网络中创建EC2 ENI网卡，并将该网卡添加给“Worker”。这个添加网卡的过程比较费时，并且每个ENI网卡需要在用户子网中消耗一个IP地址。有些情况下，这种模型还不错，简单并且支持VPC的所有特性。但最大的弊端，也是特别被某些用户所诟病的，就是等待VPC启动所耗费的时间过长。因此，AWS将ENI从“Worker”中移出，在“Worker”与ENI之间做了NAT，在多个不同的“Worker”间复用同一个ENI。本质上，这意味着在多个租户间复用数量有限的ENI网卡。这样改进后，带来的直接好就是可预期的VPC启动延时，快速的水平伸缩，低服务延时，和高易用性。

3. Firecracker的设计

3.1 内部架构

Firecracker微虚机的创建用到两个组件，Jailer和Firecracker，前者负责利用Linux提供的seccomp、cgroup、chroot、net/pid/user namespaces来创建沙箱环境，然后在其创建的沙箱环境中启动后者。后者利用Linux KVM创建设备模型极度精简的微虚拟机。结构如下：



6 firecracker结构框图

一个Firecracker进程就是一个微虚拟机，其内部主要有三个组件：

API Server

API Server以Unix domain socket的方式对主机提供了一个API endpoint，接口采用RESTful API
格式，详见接口规范[10]。

通过这个API Endpoint，可以对微虚机进行管理和控制，包括：

- 规格配置：比如vCPU个数，用户内存大小；
- 网络配置：添加一个或多个网卡；
- 存储配置：
  - 添加“只读”或“读写”虚拟盘，每个虚拟盘盘是一个基于文件的块设备；
  - 运行时触发“re-scan”；
  - 更换后端文件；
- QoS：通过带宽限制和iops限制进行流控；
- 日志与遥测配置；
- 启动配置：内核及其参数，根文件系统；
- 关闭微虚机；

Firecracker以一个单独的线程运行API Server。

Virtual Machine Monitor

VMM负责构建Firecracker定制的虚拟机模型。其中包括：

- 最小化的老式设备模型；
- 微虚机元数据服务（microVM metadata service/MMDS）；
- VirtIO虚拟网络设备和块设备；
- QoS流控；
- 串口控制台和半功能键盘；

VMM采用单线程事件驱动模型，对各种I/O请求进行服务。

vCPU Threads

根据规格配置，通过KVM接口创建vCPU结构，为每个vCPU启动一个线程，执行vCPU事件循环，并执行同步I/O和基于内存映射I/O的操作。

3.2 微虚机模型

Firecracker利用了硬件辅助虚拟化，同时使用一个极简的设备模型。从系统虚拟化角度看，可分解为如下几个方面：

- CPU/Memory: 利用VT-x进行CPU虚拟化和内存虚拟化
- 系统总线：移除PCI系统总线模

- 设备模拟：
  - virtio-net
  - virtio-block
  - console
  - keyboard
  - irqchip
  - clock source
  - KVM in kernel devices
  - in VMM

3.3 社区及路线图

在Firecracker代码库中的文档里面公布的路线图上[8]可以看出，目前它主要部署在Intel的平台，计划还会支持AMD、ARM平台，及存储加密等特性。

Firecracker的开发者与社区的互动还是比较积极的。由此看来，他们希望借助社区的力量以实现与k8s很好的集成。在它的版本库上，还提供了一个与containerd对接的原型“firecrack-containerd”。Firecracker的维护者Anthony Liguori（前QEMU社区维护者）也表示出与Kata Containers社区合作的意愿。

4. 总结

注意到许多关于Firecracker的评论中，不少人对“容器运行时”与Firecracker之间的差别存在误解，在此强调下：**Firecracker** 是一个 **virtual machine manager**，QEMU也是一个virtual machine manager。Kata Containers使用QEMU。因此，Firecracker是AWS用于构建无服务器计算场景下的“容器运行时(Runtime)”（也叫“容器安全沙箱”）所用到的一个组件，**作用是替换掉QEMU**。当然，更谈不上是新型虚拟化技术，它依然使用Intel VT-x，依然需要机器模型和设备模型，只不过，它做的很精简（*当然，为什么不呢？*）。

为什么要替换QEMU？ 原因有很多，比如：庞大的代码体积；近年来高发的漏洞数量[12]；对基本上用不到的传统设备、总线、机器模型的模拟。虽然某些情况下，对各种硬件协议的真实模拟还是不错的，但是，针对无服务器计算（Serverless）这样的场景，需要业务**启动快，密度高，可快速水平扩展**，这种方式显然就不适合了，需要一种更敏捷的容器运行环境。

除了Firecracker，Kata Containers和gVisor也致力于提供安全可靠的容器运行环境。它们之间存在哪些差异呢？

Firecracker与Kata Containers

首先，Kata Containers使用QEMU作为VMM，使用Linux作为Guest OS，通过配置QEMU的编译选项来裁剪掉一些不用的功能，通过配置Linux的编译选项裁剪掉不用的设备驱动、子系统和一些功能。但是，QEMU中的传统机器模型始终存在，还有一些“设备模拟”的功能没有编译选项，因此无法被裁剪掉；而Linux的子系统，如SMP，调度，内存管理，ACPI，PCI总线等也都依然假定活在真实物理机上。对于无服务器计算场景，这些都是没有意义的，因为在这种场景下，Guest OS完全由我们来提供。不需要考虑其他情况，如Windows或其他老的Linux版本。但凡对业务运行没有用的设备都不需要，甚至是设备模型和机器模型。Firecracker走的方向与我们正在走的设计方向很相似，即：**极简的机器模型，拿掉PCI总线，替换掉QEMU**。我们也曾考虑用Rust语言构建容器沙箱，但AWS动手更早，并已经大规模部署了。回头来考虑我们的容器实例场景，Aliyun ECI，试想下我们用Firecracker替换了QEMU，并且对Guest OS做进一步的优化，比如页表预分配，vCPU直接64bit分页模式启动等，沙箱的启动可以更快。

Firecracker与gVisor

*对比Firecracker与gVisor的设计，不难发现一个很有意思的话题：**“虚拟化的界面”***，即：对Guest而言，它与Hypvervisor之间的接口是什么？_与“虚拟机”模型不同，gVisor采用了与Dune[13]类似的“进程虚拟化”模型，将虚拟化的界面画在了“系统调用/syscall”这个边界上。因此，彻底去掉了机器模型和设备模型。这不仅意味着减轻了虚拟化的“开销”，还意味着可以更加灵活高效的利用主机上的系统资源。gVisor就通过host-guest（vmx-root/nonroot）镜像内核地址空间的内存布局设计，使得它可以既作为host上的hypervisor，又作为guest中的supervisor，因此可以在vCPU调度上内外打通，使得vCPU“协程”可以按需增减。此外，gVisor自然的享受了Go Runtime中的concurrent garbage collector带来的好处，比如当执行完“用户负载/函数”时，或当

Guest中的“工作集”缩小时，Go的GC的会立即把多余的内存回收并还给主机系统。这就使得gVisor在vCPU和内存资源的使用上都很有“弹性”。

但是，在系统调用这个边界上提供虚拟化意味着：为Guest提供大量的POSIX接口支持。从安全隔离的角度，这开出了很大的口子，因此，出于安全和性能的考虑，gVisor不得不将一些系统调用的实现放在它的内核里面，并在整个进程外面套一层沙箱环境（cgroups，namespaces，seccomp）。一直以来，我们也在讨论这个话题，这个“界面”越往上，虚拟化的开销越低，但同时，接口数量也变得越大，含义越丰富，严谨性越弱，即：“攻击面”越
大。_那么，将“界面”画在哪里才是合理的呢？_可能没有一种完美的设计可以满足所有用户场景。但在针对无服务器计算这个场景，AWS给出的选择是**接口数量小、含义确定的“虚拟机”模型**，不同的是采用极简的机器模型和设备模型来降低开销。当然，这也就是说，无论在vCPU还是内存方面，firecracker都跟普通虚拟机一样，没有gVisor那样的“弹性”。这也说明，当在安全隔离和其他因素之间做取舍时，AWS首选前者。

此外，Go runtime并不是“免税”的，它带来“弹性”的同时也引入了一些不利的影响，Cody Cutler[14]在他paper中对用Go语言编写的内核进行了详细分析，在此不展开了。最后，我们也看到，自Google开源gVisor以来，已经存在几个漏洞，如[15][16]。可见开发一个稳定的内核很不容易，需要严谨的设计和长时间的打磨。

Firecracker的核心设计准则

无服务计算（Serverless）到底需要什么样的平台呢？ 根据前面的分析，其实不难看出，AWS已经给出了它的答案。

一、对外服务的前提是安全隔离，而硬件辅助虚拟化是在多租户间进行安全隔离的最低标准。**在安全和性能面前， 安全第一**。

二、无服务器计算场景下，典型的业务特征是生命周期短，因此需要它的平台提供：

- **启动快**：极简设备模型，没有BIOS，没有PCI，甚至不需要设备直通；
- **密度高**：内存开销低；
- **水平扩展**：因为容器的生命周期短；甚至不需要热迁移；

三、在提高服务器资源利用率方面，AWS也给出了答案，即：**基于统计数据搞混部**。例如AWS Lambda，它将不同用户、不同函数运行在同一组硬件资源上，利用用户负载的波峰波谷互补（*与我们搞混部的思路也是一致的*）。

参考文献

1. Secure and fast microVMs for serverless computing
2. Chrome OS Virtual Machine Monitor
3. Firecracker Design
4. Getting Started with Firecracker
5. A Serverless Journey:Under the Hood of AWS Lambda
6. It doesn’t work with Docker, K8s right now, but everyone’s going nuts anyway for AWS’s Firecracker microVMs
7. Announcing the Firecracker Open Source Technology: Secure and Fast microVM for Serverless Computing
8. Firecracker roadmap
9. firecracker-containerd
10. API Spec
11. Operating-system-level virtualization
12. Qemu Vulnerability Statistics
13. Dune: Safe User-level Access to Privileged CPU Features
14. The benefits and costs of writing a POSIX kernel in a high-level language
15. CVE-2018-16359
16. CVE-2018-19333

点赞 4 分享 举报

0 条评论 我来说两句

登录 后参与评论

分享

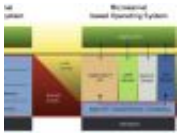
相关文章

有哪些工具可以让嵌入式开发事半功倍？详…

嵌入式开发就是指在嵌入式操作系统下进行开发，一般常用的系统有μcos，vxworks，linux，android等。当然，对于…

Linux阅码场

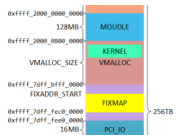
有关微内核OS史上最透彻一篇 - 写于华为鸿…



华为鸿蒙OS发布已经一周了，在这一周中发生了很多事情，有人对华为路转粉，也有人对华为粉转黑，在时下，只要…

Linux阅码场

ARM64 Kernel Image Mapping的变化



随着linux的代码更新，阅读linux-4.15代码，从中发现很多与众不同的地方。之所以与众不同，就是因为和我之前从网…

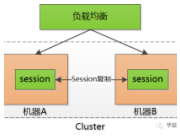
Linux阅码场

小犀牛鸟迎新礼包背后有故事

【前言】 2019年是腾讯犀牛鸟精英人才培养计划成立的第三年，三年来，项目吸引了大量海内外优秀学生的关注及报…

腾讯高校合作

一文带你彻底理解 Cookie、Session、Token



1、很久很久以前，Web 基本上就是文档的浏览而已， 既然是浏览，作为服务器， 不需要记录谁在某一段时间里都浏…

乔戈里

北京邮电大学石川：人生路上的幸与不幸

精彩内容编者按：2013年，CCF联合腾讯发起“犀牛鸟”基金，旨在为青年学者搭建“让伟大的梦想变成现实的影响”的…

腾讯高校合作

南国寒潮中的一股暖流： 犀牛鸟项目“探”主…

犀牛鸟项目“探”主题年度线下沙龙　　相比于阳光和煦的往常月份，12月初的深圳显得有些特别。冷风过境带来的丝…

腾讯高校合作

「SDL第八篇」支持倍速与慢放的YUV视频播放器

今天向大家介绍一下如何通过 SDL 实现一个YUV视频播放器。它与上次介绍的音频播放器一样，也是一个简单的不能再简单的播放器了。只不过一个是播放的音频PCM…

音视频_李超

面向对象编程的思想

天天_哥

一文带您彻底理解Cookie、Session、Token

1、很久很久以前，Web 基本上就是文档的浏览而已， 既然是浏览，作为服务器， 不需要记录谁在某一段时间里都浏览了什么文档，每次请求都是一个新的HTTP协议， ...

掌上编程

更多文章 >