阿里云开发者社区 | 云原生应用平台

ALIBABA CLOUD DEVELOPER COMMUNITY | CLOUD NATIVE APPLICATION PLATFORM

Serverless 开发速查手册

让开发者像使用手机一样玩转 Serverless

基础概念 | 设计哲学 | 工具使用 | 实战案例

Serverless 技术入门工具书

作者：刘宇（江昱）

关注阿里云官方 Serverless 微信公众号

阿里云开发者“藏经阁”

前言

随着云计算的不断发展，Serverless 架构逐渐成为了近些年来备受关注的焦点，作为被认为将会引领云计算下一个十年的技术架构，Serverless 架构一方面继承云计算优点的同时，另一方面也为业务带来了极致弹性、按量付费、低运维等优势；在 Serverless 架构逐渐成为诸多业务的技术选型之时，其也暴露出了包括调试困难，厂商锁定严重，资源琐碎管理难度大等在内一些列的问题，Serverless Devs 作为开源建设的 Serverless 工具链生态，一方面从开源开放的角度出发，为开发者们提供更安全、可靠、自主的工具；另一方面也通过 Serverless 应用全生命周期角度出发，为开发者提供一款可以“像玩手机一样玩转 Serverless 架构”的应用生态。

为什么写作本书

随着时间的发展，Serverless 架构在最近几年变得越发的火热，其凭借着极致弹性、按量付费、低运维等特性，在很多领域发挥着越来越重要的作用；而伴随着 Serverless 架构越发火热，Serverless 架构在工具链层面的匮乏逐渐显露；以提高研发效能为核心优势之一的 Serverless 架构，因为没有完善的工具链体系，导致开发困难、调试困难、管理困难、运维困难；此时 Serverless Devs 应运而生，在 Serverless 架构持续赋能传统应用整体效能提升的同时，Serverless Devs 提供了 Serverless 应用全生命周期管理工具，助力 Serverless 应用综合效能的提升，让应用开发效率更上一层楼。本书希望通过更简单明了的语言，真实的案例，以及开放的源代码，为读者介绍 Serverless 架构工具链 Serverless Devs 相关的基础知识，希望读者可以通过本书真正意义上理解 Serverless Devs 的重要价值，以及可以顺利在 Serverless 架构下开发，并上线 Serverless 应用项目，更加直接地获得云计算带来的时代技术红利。

关于本书作者

刘宇：国防科技大学电子信息专业博士；阿里云 Serverless 产品经理，开源项目 Serverless Devs 发起人；《Serverless 架构：从原理、设计到项目实战》、《Serverless 工程实践：从入门到进阶》、《架构师特刊：人人都能学会的 Serverless 实践》等热门图书作者；阿里云麒麟布道师。

本书主要内容

本书是一本关于 Serverless 开发者工具助力 Serverless 应用研发效能提升的书，本书一共有 5 个章节，通过这 5 个章节读者可以从 Serverless 架构以及 Serverless Devs 开发者工具的基础入门，通过快速入门案例，在 AI 领域下的应用，在 CI/CD 中的实战案例，等多个维度进行学习和了解 Serverless Devs 开发者工具。

第一章：介绍什么是 Serverless 以及 Serverless Devs，并且为读者分享 Serverless 工具链的必然性和重要性；

第二章：帮助读者快速入门 Serverless 以及 Serverless Devs 开发者工具；包括不限于工具的下载，部署，调试，以及相关规范等内容；

第三章：通过一些常见的，有趣的 AI 案例，帮助读者可以更为深入的学习和了解 Serverless Devs 开发者工具；

第四章：通过在 CI/CD 工具/平台下的实战，为读者介绍 Serverless Devs 的集成性与被集成性；

第五章节：是 Serverless 应用的开发经验和常见的应用思路，包括传统框架的迁移案例，应用调试调优的方法等；

如何阅读本书

在阅读此书前，读者应当具有一定的编程技术，例如掌握 Node.js、Python 等语言的编程能力或具备这方面的知识，同时也需要对云计算有初步的了解，有使用过阿里云的经验，另外也需要读者对 Serverless 架构相关的基础有一定的了解。这本书将会通过循序渐进的方法，从云计算的发展历史，什么是 Serverless 架构，Serverless 架构下的工具链体系重要性说起，通过快速上手案例，不同领域的实战案例，以及对 Serverless Devs 的开发者模型的介绍等几个部分为读者介绍 Serverless Devs 助力 Serverless 架构带来的研发效能变革，我建议读者分几遍阅读本书。

第一遍先将书通读，主要弄清楚概念，再完成 Serverless Devs 的基础入门，并对基于 Serverless Devs 哲学思想，使用流程等有一个较为基础的了解；

第二遍通过对本书在不同领域的实践案例，进行进一步的学习和探索，了解 Serverless Devs 如何助力 Serverless 架构在不同行业的发力，助力业务综合效能的提升；

第三遍可以针对本书的开发经验分享等部分，对 Serverless 架构有更为深入的了解，帮助开发者可以更好的了解 Serverless 架构，上手 Serverless 应用开发；

阅读是一个反复又枯燥的过程，读者只有在反复的研读中，才能使自己在 Serverless 架构的学习不断深入，通过 Serverless Devs 工具链的加持，实现更多的应用案例，本书的案例代码可能会存在失效的情况，但是笔者会尽快在每个案例的代码仓库中更新代码，希望读者可以切实利用好每个案例提供的仓库地址，帮助自己更加高效的了解、学习 Serverless 架构下如何更简单、更方便、更科学地开发、部署、运维机器学习项目。

致谢

本书是一本 Serverless 工具链相关的书籍，在进行本书编写的时候，曾遇到过很多的困难，在此特别感谢阿里云云原生团队给予的大力支持。

感谢杨秋弟（曼红）、杨浩然（不瞋）等前辈，是你们在这本书从开始到结束的过程中，不断给予鼓励和支持，才得以让本书在预期的时间内顺利完成；感谢阿里云云原生开发者运营姜曦（筱姜），在本书对外宣传过程中给予的帮助和支持；感谢、罗松（西流）、张千风（千风）等人在本书编写过程中，指导部分代码的完成，功能、案例的实现。同时，也要感谢我的家人对我工作的鼓励和支持，对我每走一步的信任和鼓舞；感谢身边的小伙伴，对我的关心和帮助；感谢身边每一个人，谢谢你们。

由于作者水平有限，书中不足及错误之处在所难免，敬请专家和读者给予批评指正。

江昱（刘宇）

2022 年 2 月

目录

前言 3

为什么写作本书 3

关于本书作者 3

本书主要内容 4

如何阅读本书 4

致谢 5

1. 引言 10

1.1 从云计算到 Serverless 10

1.2 Serverless 架构简介 18

1.3 Serverless 架构应用场景 21

1.3.1 Web 应用/移动应用后端 22

1.3.2 实时文件/数据处理 23

1.3.3 离线数据处理 24

1.3.4 人工智能领域 24

1.3.5 IoT 等物联网领域 25

1.3.6 监控与自动化运维 25

1.4 Serverless Devs 简介 27

1.4.1 工具链建设的必要性 27

1.4.2 六大优势 28

1.4.5 设计哲学 29

1.5 阿里云函数计算（FC）组件简介 33

1.5.1 Serverless Devs 和 FC 组件的关系 33

1.5.2 五大亮点 34

1.5.3 功能支持 34

2. 快速入门与功能案例 36

2.1 Serverless Devs 基础入门 36

2.1.1 工具安装 36

2.1.2 密钥配置 36

2.1.3 Yaml 的使用规范 42

2.1.4 多种操作模式下的工具体系 51

2.2 案例与体验 55

2.2.1 项目上手体验 55

2.2.2 功能上手体验 63

2.3 让发布更安全的：线上异动感知 71

2.3.1 部署前检测 72

2.3.2 部署异动感知 74

2.3.3 在自动化流程中的策略 74

2.4 性能心中有数：一键压测函数计算 75

2.4.1 实现原理 75

2.4.2 操作案例 75

2.5 从工具看函数资源评估 78

2.5.1 内存探测模式 78

2.5.2 并发度探测模式 80

2.6 Serverless Devs 与 Funcraft/Fcli 的对比 82

2.6.1 形式对比 82

2.6.2 功能对比 82

2.6.3 场景对比 83

2.6.4 迁移案例 84

3. Serverless 架构下 AI 最佳实践 87

3.1 PaddleOCR 项目开发与部署 87

3.1.1 本地开发 87

3.1.2 部署到 Serverless 架构 90

3.1.3 项目优化 93

3.2 Tensorflow 与目标检测系统 95

3.2.1 本地开发 95

3.2.2 部署到 Serverless 架构 97

3.2.3 项目优化 103

4. Serverless Devs 与 CI/CD 实战 104

4.1 与 CI/CD 平台/工具集成 104

4.1.1 与 Github Action 的集成 105

4.1.2 与 Gitee Go 的集成 107

4.1.3 与 Jenkins 的集成 110

4.1.4 与云效的集成 113

4.2 Custom Container 的 CI/CD 最佳实践案例 115

4.2.1 实验准备 115

4.2.2 流程 116

4.2.3 项目测试 118

4.2.4 案例总结与注意事项 119

4.3 基于 Gitee Go 的函数代码更新与版本发布 123

4.3.1 实验准备 123

4.3.2 基于 Gitee 的 CD 能力建设 125

4.3.3 案例总结 132

5. Serverless 应用开发经验/常见思路 **133**

5.1 如何上传文件 133

5.2 文件读写与持久化方法 135

5.3 慎用部分 Web 框架的特性 136

5.3.1 异步 136

5.3.2 定时任务 136

5.4 要注意应用组成结构 137

5.4.1 静态资源与业务逻辑 137

5.4.2 业务逻辑的进一步拆分 137

5.5 善于利用函数的厂商特性 139

5.6 传统框架迁移方案与策略 141

5.6.1 传统框架迁移案例 141

5.6.2 通过开发者工具快速迁移/部署 143

5.7 可观测性 149

5.7.1 通过工具进行 Metrics 查看 151

5.7.2 通过工具进行 Logs 查看 152

5.8 如何对应用进行调试 154

5.8.1 在线调试 154

5.8.2 端云联调 155

5.8.3 本地调试 162

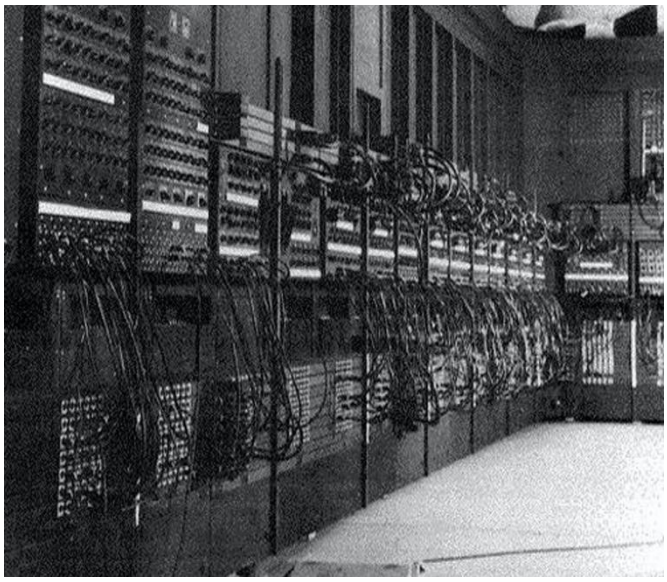
5.8.4 其他调试方案 164

附录 **167**

1. 引言

1.1 从云计算到 Serverless

自世界第一台通用计算机 ENIAC（Electronic Numerical Integrator And Computer）诞生以来，计算机科学与技术的发展就从未停止过前进的脚步，尤其是近些年计算机的发展更是日新月异，有不断突破和进化的人工智能领域，有 5G 带来更多机会的物联网领域，还有“可信”的区块链技术，当然也有不断更新、不断迭代，不断走进“寻常百姓家”的云计算。

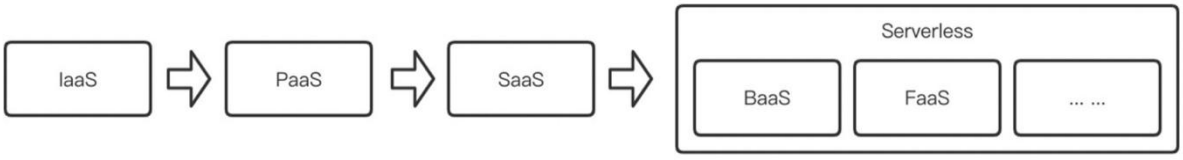


说起云计算，不得不说“Utility Computing”，在 1961 年麻省理工学院百周年纪念典礼上，约翰·麦卡锡（1971 年图灵奖获得者）第一次提出了“Utility Computing”的概念，这个概念可以认为是云计算的一个“最初的”，“超前的”遐想模型，它翻译成现今的大意就是：计算机在未来，将变成一种公共资源，会像生活中的水、电、煤气一样，被每一个人使用。1984 年，SUN 公司联合创始人 John Gage（约翰·盖奇）提出了“网络就是计算机（The Network is the Computer）”的重要猜想，用于描述分布式计算技术带来的新世界，而再过 12 年，即 1996 年，康柏（Compaq）公司的一群技术主管在讨论计算业务的发展时，首次使用了 Cloud Computing 这个词，并认为商业计算会向 Cloud Computing 的方向转移。这也是“云计算”从雏形到正式被提出的基本过程。

自“云计算”被提出之后，其可谓是如同雨后春笋般，蓬勃发展： 2003 年到 2006 年间，谷歌发表了 The Google File System、MapReduce: Simplified Data Processing on Large Clusters 、Bigtable: A Distributed Storage System for Structured Data 等文章，这些文章指明了 HDFS（分布式文件系统），MapReduce（并行计算）和 Hbase（分布式数据库）的技术基础以及未来机会，至此奠定了云计算的发展方向；2006 年，Google 首席执行官埃里克·施密特（Eric Schmidt）在搜索引擎大会（SESSanJose2006）首次公开正式地提出“云计算”（Cloud Computing）的概念，同年亚马逊第一次将其弹性计算能力作为云服务进行售卖，这也标志着云计算这种新的商业模式正式诞生。两年后，即 2008 年，微软发布云计算战略和平台 Windows Azure Platform，尝试将技术和服务托管化、线上化。到了 2009 年，UC Berkeley 发表了：Above the Clouds: A Berkeley View of Cloud Computing，在该文章中，明确指出：云计算是一个即将实现的古老梦想，是计算作为基础设施这一长久以来梦想的新称谓，它在最近正快速变为商业现实。在该文章中，明确的为云计算做了定义：云计算包含互联网上的应用服务及在数据中心提供这些服务的软硬件设施。同时在该文章中，也提出了云计算所面临的挑战和机遇：

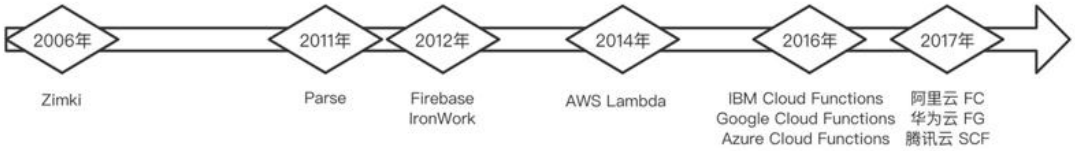
问题	**机会**
服务的可用性	选用多个云计算提供商；利用弹性来防范 DDOS 攻击
数据丢失	标准化的 API；使用兼容的软硬件以进行波动计算
数据安全性和可审计性	采用加密技术，VLANs 和防火墙；跨地域的数据存储
数据传输瓶颈	快递硬盘；数据备份/获取；更加低的广域网路由开销；更高带宽的 LAN 交换机
性能不可预知性	改进虚拟机支持；闪存；支持 HPC 应用的虚拟集群
可伸缩的存储	发明可伸缩的存储
大规模分布式系统中的错误	发明基于分布式虚拟机的调试工具
快速伸缩	基于机器学习的计算自动伸缩；使用快照以节约资源
声誉和法律危机	采用特定的服务进行保护
软件许可	使用即付许可；批量销售

在 UC Berkeley 的这篇文章中，作者不仅仅对云计算进行了一个比较明确和细腻的定义，提出了面临的挑战和存在的机遇，更对云计算的未来发展方向等进行大胆预测。同年，阿里软件在江苏南京建立首个“电子商务云计算中心”（即现在的阿里云）。至此，云计算进入到了更加快速的发展阶段。

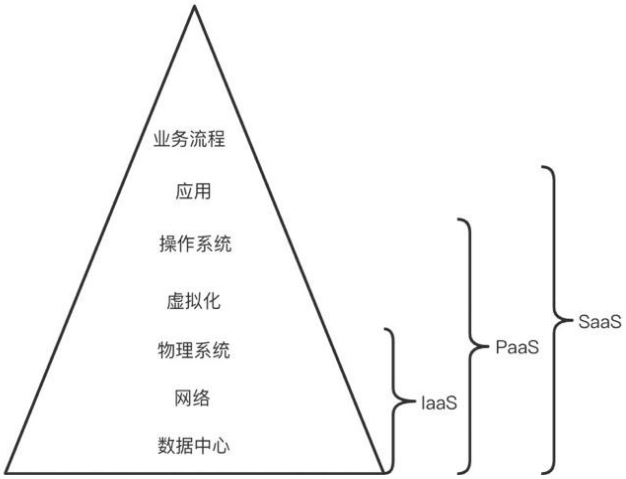


如上图所示，云计算飞速发展的阶段，云计算的形态也在不断的演进，从 IaaS 到 PaaS，再到 SaaS，云计算逐渐地“找到了正确的发展方向”，2012 年由 Iron.io 的副总裁 Ken Form 所写的一篇名为 Why The Future of Software and Apps is Serverless 的文章中，提出了一个新的观点： 即使云计算已经逐渐的兴起，但是大家仍然在围绕着服务器转。 不过，这不会持续太久， 云应用正在朝着无服务器方向发展，这将对应用程序的创建和分发产生重大影响。并首次将“Serverless”这个词带进了大众的视野。2014 年 Amazon 发布了 AWS Lambda 让“Serverless”这一范式提高到一个全新的层面，为云中运行的应用程序提供了一种全新的系统体系结构，至此再也不需要在服务器上持续运行进程以等待 HTTP 请求或 API 调用，而是可以通过某种事件机制触发代码执行，通常这只需要在 AWS 的某台服务器上配置一个简单的功能。此后 Ant Stanley 在 2015 年 7 月名为 Server are Dead**…的文章中更是围绕着 AWS Lambda 及刚刚发布的 AWS API Gateway 这两个服务解释了他心目中的 Serverless，并说 Servers are dead … they just don't know it yet.

到了 2015 年，在 AWS 的 re:Invent 大会上，Serverless 的这个概念更是反复地出现，其中包括了 The Serverless Company Using AWS Lambda 和 JAWS：The Monstrously Scalable Serverless Framework 的这些演讲。随着 Serverless 这个概念的进一步发酵，2016 年 10 月在伦敦举办了第一届的 ServerlessConf，在两天时间里面，来自全世界 40 多位演讲嘉宾为开发者分享了关于这个领域进展，并且对未来进行了展望，提出了 Serverless 的发展机会以及所面临的挑战，这场大会是针对 Serverless 领域的第一场具有较大规模的会议，在 Serverless 的发展史上具有里程碑的意义。



如上图所示，而截至 2017 年，各大云厂商基本上都已经在 Serverless 进行了基础的布局，尤其是国内的几大云厂商，也都先后在这一年迈入“Serverless 时代”。从 IaaS 到 PaaS 再到 SaaS 的过程中，如图所示，云计算所表现出的去服务器化越来越明显，那么 Ken Form 所提出来的 Serverless 又是什么，它在云计算发展的过程中，又在扮演什么角色呢，它的去服务器化又到了什么程度呢?



Serverless 翻译成中文是无服务器，所谓的无服务器并非是说不需要依靠服务器等资源，而是说开发者再也不用过多考虑服务器的问题，可以更专注在产品代码上，同时计算资源也开始作为服务出现，而不是作为服务器的概念出现，Serverless 是一种构建和管理基于微服务架构的完整流程，允许用户在服务部署级别而不是服务器部署级别来管理用户的应用部署。与传统架构的不同之处在于，它完全由第三方管理，由事件触发，存在于无状态（Stateless），暂存（可能只存在于一次调用的过程中）在计算容器内，Serverless 部署应用无需涉及更多的基础设施建设，就可以基本实现自动构建、部署和启动服务。

近些年来，微服务（Micro Service）是软件架构领域另一个热门的话题，如果说微服务是以专注于单一责任与功能的小型功能块为基础，利用模组化的方式组合出复杂的大型应用程序，那么可以进一步认为 Serverless 架构可以提供一种更加“代码碎片化”的软件架构范式，而这一部分称之为 Function as a Services（FaaS）。而所谓的“函数”提供的是相比微服务更加细

小的程序单元。例如，可以通过微服务代表为某个客户执行所有 CRUD 操作所需的代码，而 FaaS 中的函数可以代表客户所要执行的每个操作：创建、读取、更新以及删除。当触发“创建账户”事件后，将通过函数的方式执行相应的“函数”。单就这一层意思来说，可以简单地将 Serverless 架构与 FaaS 概念等同起来。但是就具体的概念深刻探索的话，Serverless 和 FaaS 还是不同的，Serverless 和 FaaS 被广为接受的关系是：

$$Serverless = FaaS + BaaS \quad (+ \ .....)$$

在这个关系中，可以看到 Serverless 的组成除了 FaaS 和 BaaS 之外，还有一系列的省略号，其实这是 Serverless 给予给大家的遐想空间，给予这个时代的一些期待。

随着各大厂商逐渐布局 Serverless 领域，Serverless 也逐渐地从启蒙阶段，市场教育阶段向更深一步生产应用，最佳实践阶段前进。

同样在 2017 年，CNCF Serverless WG 成立了，并且开始以社区的力量推动 Serverless 快速前行，包括 CNCF Serverless Whitepaper、CloudEvents 等相关的立项研究与探索，2017 年年末，eWEEK 的 Chris J. Preimesberger 发表文章 Predictions 2018: Why Serverless Processing May Be Wave of the Future 来表达在“全新的阶段下”大家对 Serverless 的看法和期盼，在这篇文章中有来自知名团队以及公司的相关负责人对 Serverless 表达了自己的想法：

- Sumo Logic 产品营销副总裁 Kalyan Ramanathan：Serverless 可能是继容器之后的未来。
- Avere Systems 技术总监 Dan Nydick：我们将看到更多 Serverless 技术和托管服务。
- Atlassian 平台负责人 Steve Deasy：2018 年将改变软件的构建方式。
- Evident.io 的首席执行官 Tim Prendergast 和客户解决方案副总裁 John Martinez：容器和 Serverless 正在兴起，但它们会带来安全问题。
- Contino 美国总裁 Jason McDonald：Serverless 将继续增加其影响力。
- 美国 OVH 的现场布道者兼原理系统工程师 Paul Stephenson：（2018 年）将会更清楚地认识到 Serverless 可以解决哪些问题。
- 数据探险首席执行官 Seth Noble：Serverless 将在 2018 年与其他技术进行集成。
- Platform9 首席执行官 Sirish Raghuram：Kubernetes 将在 Serverless 产品－AWS Lambda 部署中发挥更大的影响力。

- Accelerite 首席执行官 Nara Rajagopalan：Serverless 将改变开发模式。

而到了 2018 年，Serverless 的发展速度要比想象中的更加快速，在这一年，Google 发布了 Knative， 一个基于 Kubernetes 的开源 Serverless 框架，具备构建容器、流量调配、弹性伸缩、零实例、函数事件等能力。AWS 发布了 Firecracker，一个开源的虚拟化技术，面向基于函数的服务，创建和管控安全的、多租户的容器。Firecracker 的目标是把传统虚拟机安全性和隔离性，和容器的诉求，资源效率结合起来。在这一年，CNCF 也正式发布了 Serverless 领域的白皮书：CNCF Serverless Whitepaper V1.0，阐明 Serverless 技术概况、生态系统状态，为 CNCF 的下一步动作做指导，同时 CloudEvnent 规范，进入 CNCF Sandbox；在这一年，UC Berkeley 发文 Serverless Computing: One Step Forward, Two Steps Back，表达了对 Serverless 的担忧和挑战，在这篇文章中，作者认为 Serverless 会对开源服务创新有所阻塞，作者认为 Serverless 是：One Step Forward，Two Steps Back，关于这个部分内容的描述如下：

One Step Forward

通过提供自动缩放功能，今天的 FaaS 产品在云编程方面迈出了一大步，它提供了一种实际上可管理的，看似无限的计算平台。

Two Steps Back

首先，他们忽略了高效数据处理的重要性；其次，它们阻碍了分布式系统的开发。

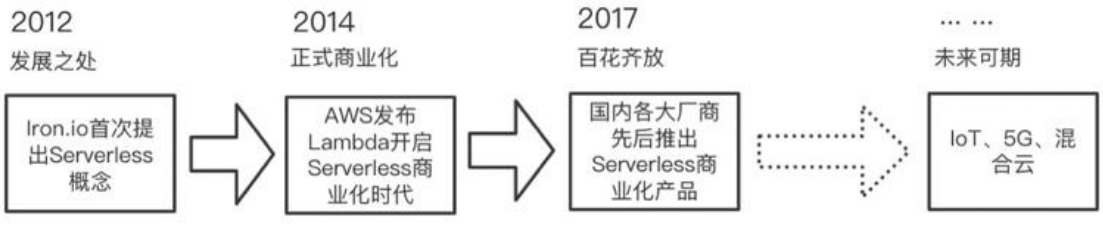
任何一个新的技术、概念出现都会遇到一定的挑战和担忧，就如同当年云计算出现时，也被一些人（如 Oracle 公司总裁 Larry Ellison、GNU 发起人 Richard Stallman）认为只是又一个商业炒作的概念，毫无新意，甚至蠢不可及。当然，事实也证明，任何一个新的事物，都只有在经历各种挑战和质疑之后，才能更茁壮地成长，Serverless 也不例外，从 2019 年开始，Serverless 进入到了一个真正意义上的生产应用，最佳实践快速发展阶段而 2019 年对 Serverless 而言是非常关键的一年，也是 Serverless 具有里程碑式发展的一年，被很多人定义为“Serverless 正式发展的元年”。在这一年不仅有 KubeCon 在中国上海的 CloudNativeCon 中关于 Serverless 的“海量主题演讲”，这些演讲包括来自 Captial One 银行的 Kevin Hoffman

的 WebAssembly、无服务器和云，IBM 的 Doug Davis 的 CNCF CloudEvents 项目：迈向无服务器互操作的一步等；在这一年，更有 UC Berkeley 最新的文章，也被称之为 Serverless 领域的 UC Berkeley 断言：Cloud Programming Simplified: A Berkeley View on Serverless Computing，在这篇文章中，经历了一年的发展，作者也从 UC Berkeley 的学者们也从 1 年前的质疑，悲观转变成了一种自信与期待，这篇文章中，作者犀利断言 Serverless 将会在接下来的十年被采用，将会得到飞速的发展，并且说出了一些观点：

- 新的 BaaS 存储服务会被发明，以扩展在 Serverless 计算上能够运行更加适配的应用程序类型。这样的存储能够与本地块存储的性能相匹配，而且具有临时和持久可供选择。
- 比现有的 x86 微处理器更多的异构计算机。
- 更加安全、易用的编程，不仅具有高级语言的抽象能力，还有很好的细粒度的隔离性。
- 基于 Serverless 计算的价格将低于 Serverful 计算，至少不会高于 Serverful 计算。
- Serverless 将会接入更多的后台支撑服务，如 OLTP 数据库、消息队列服务等。
- Serverless 计算一旦取得技术上的突破，将会导致 Serverful 服务的下滑。
- Serverless 将会成为云时代默认的计算范式，将会取代 Serverful 计算，因此也意味着服务器 - 客户端模式的终结。

除了这些断言，伯克利也对什么是“Serverless”给出了自己的想法：Put simply, Serverless computing = FaaS + BaaS. In our definition, for a service to be considered serverless, it must scale automatically with no need for explicit provisioning, and be billed based on usage.

在学术界，不仅仅 UC Berkeley 对 Serverless 发表过多篇论文，其实很多国内外高校，都已经在 Serverless 领域，投入足够的精力进行科研探索，就目前来看， Serverless 已经成为学术界的研究热点，从 2017 年开始，每年相关论文数呈 2 倍速增长。



如上图所示，从 2012 年，Serverless 概念被正式提出之后，2014 年 AWS 带领 Lambda 开启了 Serverless 的商业化，再到 2017 年各大厂商纷纷布局 Serverless 领域，再到 2019 年，Serverless 成为热点议题在 KubeCon 中被众多人参与探讨，UC Berkeley 发文断言 Serverless 将引领云计算的下一个十年，Serverless 随着时间的不断推进，各种技术的不断进步，正在逐渐地朝着更完整，更清晰的方向发展，随着 5G 时代的到来，Serverless 将会在更多领域发挥至关重要的作用。

从 IaaS 到 FaaS 再到 SaaS，再到如今的 Serverless，云计算的发展在近十余年中发生了翻天覆地的变化，从虚拟空间到云主机，从自建数据库等业务，到云数据库等服务，云计算的发展是迅速地，未来的方向和形态却是模糊的，没人知道云计算的终态是什么。诚然，现在有人说 Serverless 实现了当初了云计算目标，Serverless 才是真正的云计算，但是没人可以肯定地说，Serverless 就是云计算的终态表现，或许，Serverless 也仅仅是一个过渡的产物，但是这就要交个时间去验证了，目前就不得而知了。

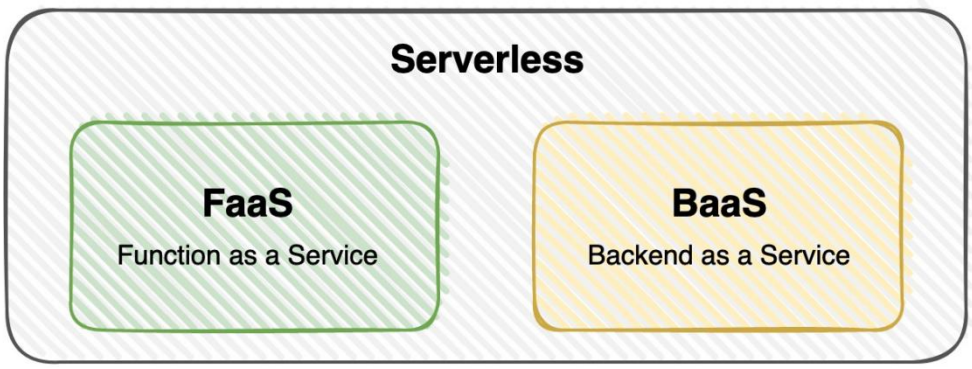
1.2 Serverless 架构简介

随着云服务的发展，计算资源被高度抽象化，从物理机到云服务器，再到容器服务，计算资源逐渐变得更加细腻化。

2012 年，Iron.io 的副总裁 Ken Form 在文章 Why The Future of Software and Apps is Serverless 中首次提出了无服务器的概念，并指出“即使云计算的已经逐渐的兴起，但是大家仍然在围绕着服务器转。不过，这不会持续太久，云应用正在朝着无服务器方向发展，这将对应用程序的创建和分发产生重大影响”；2019 年，UC Berkeley 发表论文 Cloud Programming Simplified: A Berkeley View on Serverless Computing，在文章中，作者犀利断言“新的 BaaS 存储服务会被发明，以扩展在 Serverless 计算上能够运行更加适配的应用程序类型。这样的存储能够与本地块存储的性能相匹配，而且具有临时和持久可供选择。基于 Serverless 计算的价格将低于 Serverful 计算，至少不会高于 Serverful 计算。Serverless 计算一旦取得技术上的突破，将会导致 Serverful 服务的下滑。Serverless 将会成为云时代默认的计算范式，将会取代 Serverful 计算，因此也意味着服务器 —— 客户端模式的终结。”。

Serverless 架构从 2012 年的首次走进大众视野到 2019 年成为 UC Berkeley 对云计算领域犀利断言的主角，Serverless 架构利用了 7 年的时间，完成了从一个“新的观点”向“万众瞩目的架构”转身，在这 7 年之间，Serverless 架构从鲜为人知，到被商业化应用，再到头部云厂商纷纷布局 Serverless 架构作为云计算战略，Serverless 架构逐渐的成为了人尽皆知的新的技术范式。当然，在这 7 年之间，Serverless 不仅仅技术架构在逐渐的升级和完善，其概念也是越发的明确，其目标和方向也是逐渐的清晰明朗起来。

关于 Serverless 的定义，Martin Fowler 在 Serverless Architectures 一文中认为 Serverless 实际上是 BaaS 与 FaaS 的组合。这个简单明了的定义，也逐渐的奠定了 Serverless 组成结构的基础。



如上图所示，Martin Fowler 认为，在 Serverless 架构中，应用的一部分服务端逻辑依然由开发者完成，但是和传统架构不同，它运行在一个无状态的计算容器中，由事件驱动、生命周期很短（甚至只有一次调用）、完全由第三方管理，这种情况称为 Functions as a service / FaaS。AWS Lambda 是目前的热门 FaaS 实现之一；除此之外，Serverless 架构还要有部分依赖于第三方（云端）应用或服务来管理服务器端逻辑和状态的应用，这些应用通常是富客户端应用（单页应用或者移动端 App），建立在云服务生态之上，包括数据库（Parse、Firebase）、账号系统（Auth0、AWS Cognito）等，而这些服务最早被称为 “(Mobile) Backend as a Service”，即所认为“BaaS”部分。

在同样认为 Serverless 有 FaaS 与 BaaS 结合而成的同时，CNCF 在对 Serverless 架构的定义进行了进一步完善描述：Serverless 是指构建和运行不需要服务器管理的应用程序概念。它描述了一种更细粒度的部署模型，其中将应用程序打包为一个或多个功能，上传到平台，然后执行、扩展和计费，以响应当时确切的需求。与此同时，在 2019 年 UC Berkeley 的文章《Cloud Programming Simplified: A Berkeley View on Serverless Computing》中也同样从 Serverless 架构特性的角度，对什么是 Serverless 也进行了补充描述和定义：简单地说，Serverless = FaaS + BaaS。在对于被认为是 Serverless 的服务，它必须具备弹性伸缩和按量付费的特点；在信通院云原生产业联盟所发布的《云原生发展白皮书（2020 年）》中对 Serverless 的概念也有相关的描述：无服务器（即 Serverless）是一种架构理念，其核心思想是将提供服务资源的基础设施抽象成各种服务，以 API 接口的方式供给用户按需调用，真正做到按需伸缩、按使用收费。这种架构体系结构消除了对传统的海量持续在线服务器组件的需求，降低了开发和运维的复杂性，降低运营成本并缩短了业务系统的交付周期，使得用户能够专注在价值密度更高的业务逻辑的开发上。至此，Serverless 架构从结构上，行为上以及特性上的定义，就可以总结成下图的形式：

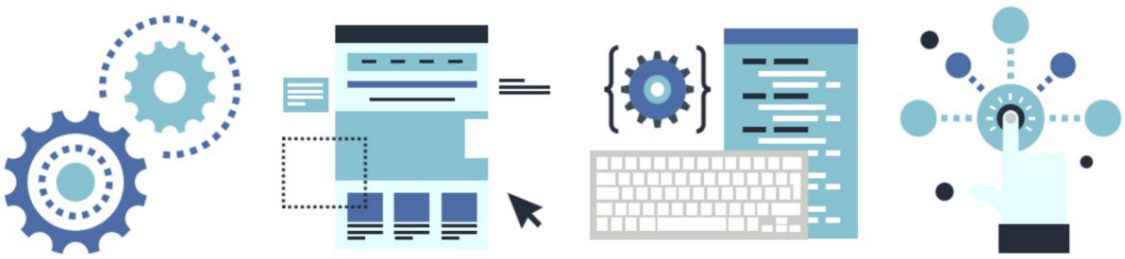
Serverless
从组成定义
Serverless = FaaS + BaaS
从行为定义
一种更细粒度的部署模型，其中将应用程序打包为一个或多个功能，上传到平台，然后执行、扩展和计费，以响应当时确切的需求
从特性定义
必须具备弹性伸缩和按量付费的特点

1.3 Serverless 架构应用场景

UC Berkeley 曾犀利断言“Serverless 架构将成为云时代的默认计算范例，主要取代服务器计算，从而为客户端 - 服务器时代带来封闭”，在信通院云原生产业联盟所发布的《云原生发展白皮书（2020 年）》中，也对云原生发展趋势进行了大胆的预测“应用服务 Serverless 化，更加聚焦业务的核心价值。Serverless 架构作为云原生技术未来的演进方向，无服务器架构技术（Serverless）也从观望逐渐落地，据 Gartner 的过往预测数据显示：到 2020 年全球将有 20% 的企业部署无服务器架构。Serverless 将进一步释放云计算的能力，将安全、可靠、可伸缩等需求交由基础设施实现，使用户仅需关注业务逻辑而无需关注具体部署和运行，极大地提高应用开发效率。同时这个方式促进了社会分工协作，云厂商可以进一步通过规模化、集约化实现计算成本大幅优化。“

由此可见，Serverless 架构将会成为未来云计算领域中重要的技术架构，将会被更多的业务所采纳，成为其技术选型。那么再进一步的深究，Serverless 在什么场景下可以有非常优秀的表现，在什么类型的场景下可能表现得并不是很理想呢？或者说，有哪些场景更适合 Serverless 架构呢？

Serverless 架构的应用场景，通常是由其特性决定方向，所支持的触发器决定具体场景。



如上图所示，在 CNCF Serverless Whitepaper v1.0 关于 Serverless 架构所适合的用户场景包括：

- 异步的并发，组件可独立部署和扩展的场景
- 应对突发或服务使用量不可预测的场景
- 短暂、无状态的应用，对冷启动时间不敏感的场景

- 需要快速开发迭代的业务，因为无需提前申请资源，因此可以加快业务上线速度

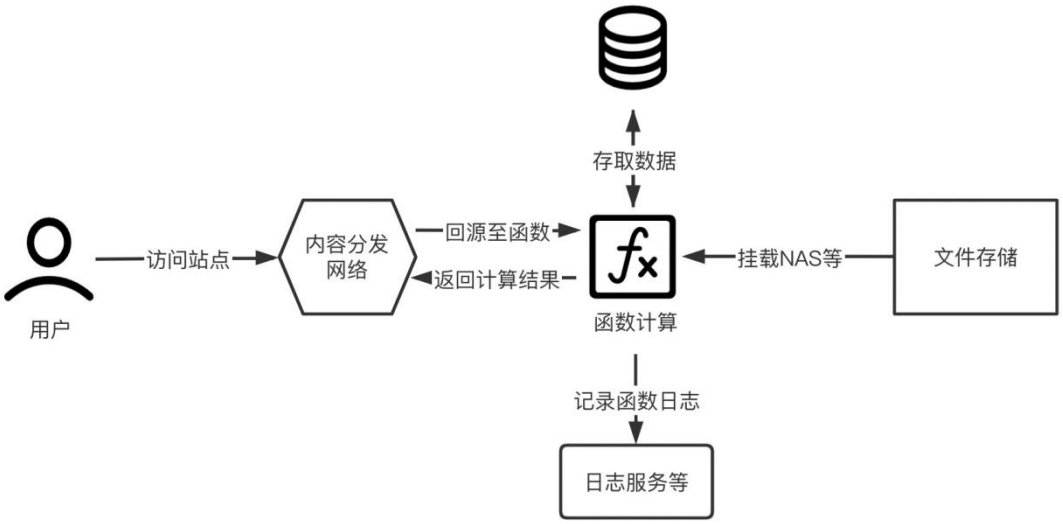
CNCF 基于 Serverless 架构的特性给出了四个用户场景之外，还结合常见的触发器提供了详细的例子：

- 响应数据库更改（插入、更新、触发、删除）的执行逻辑
- 对物联网传感器输入消息（如 MQTT 消息）进行分析
- 处理流处理（分析或修改动态数据）
- 管理单次提取、转换和存储需要在短时间内进行大量处理（ETL）
- 通过聊天机器人界面提供认知计算（异步）
- 调度短时间内执行的任务，例如 CRON 或批处理的调用
- 机器学习和人工智能模型
- 持续集成管道，按需为构建作业提供资源，而不是保持一个构建从主机池等待作业分派的任务

CNCF Serverless Whitepaper v1.0 在描述 Serverless 架构的特点时，站在理论角度描述了 Serverless 架构适合的场景或业务，云厂商将会从自身的业务角度出发，整体来描述 Serverless 架构的典型场景。通常情况下，对象存储为触发器在 Serverless 架构的典型应用场景包括视频处理、数据 ETL 处理等；API 网关更多会为用户赋能对外的访问链接以及相关联的功能等，当以 API 网关作为 Serverless 相关产品的触发器时，常见的应用场景就是后端服务，包括 App 的后端服务，网站的后端服务甚至是微信小程序等相关产品的后端服务，同时像一些智能音箱也会开放相关的接口，这个接口也是可以通过 API 网关出发云函数，获得相应的服务等；除了对象存储触发以及 API 网关触发，常见的触发器还有消息队列触发器，Kafka 触发器，日志触发器等。

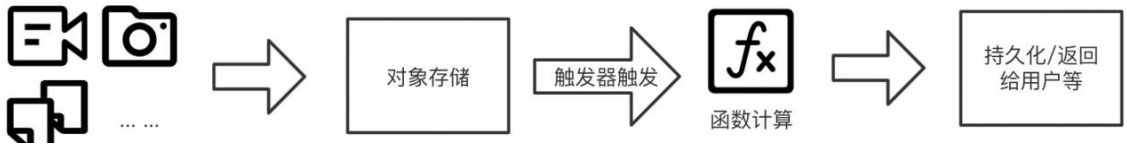
1.3.1 Web 应用/移动应用后端

Serverless 架构和云厂商所提供的其他云产品进行结合，开发者能够构建可弹性扩展的移动或 Web 应用程序 – 轻松创建丰富的无服务器后端，而且这些程序可在多个数据中心高可用运行，无须在可扩展性、备份冗余方面执行任何管理工作。例如 Web 应用处理示例：

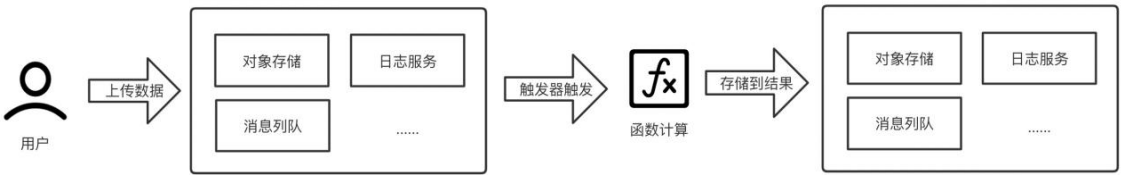


1.3.2 实时文件/数据处理

视频应用、社交应用等场景下，用户上传的图片、音视频往往总量大、频率高，对处理系统的实时性和并发能力都有较高的要求。例如：对于用户上传的图片，可以使用多个函数对其分别处理，包括图片的压缩、格式转换、鉴黄鉴恐等，以满足不同场景下的需求。例如：

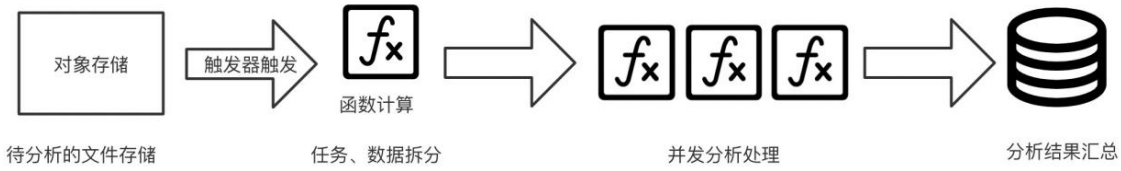


通过 Serverless 架构所支持的丰富的事件源，通过事件触发机制，可以通过几行代码和简单的配置对数据进行实时处理，例如：对对象存储压缩包进行解压、对日志或数据库中的数据进行清洗、对 MNS 消息进行自定义消费等：



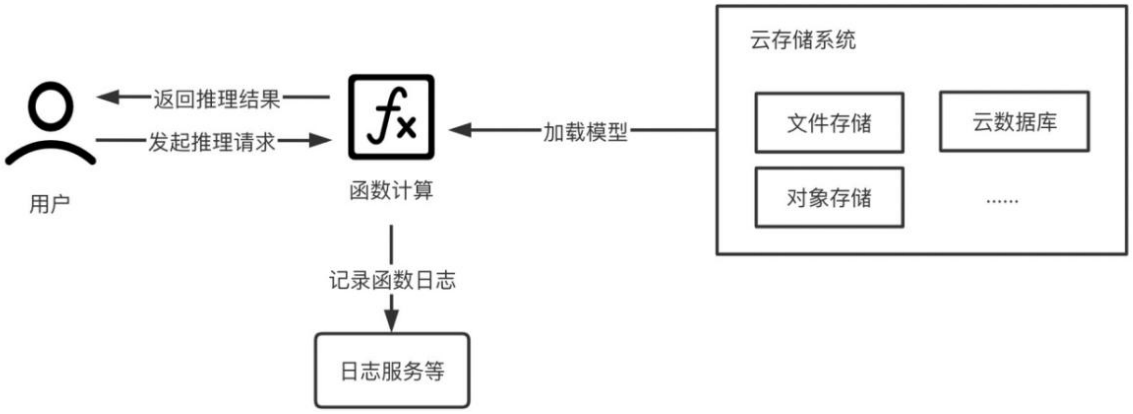
1.3.3 离线数据处理

通常要对大数据进行处理，需要搭建 Hadoop 或者 Spark 等相关大数据的框架，同时要有一个处理数据的集群。通过 Serverless 技术，只需要将获得到的数据不断的存储到对象存储，并且通过对象存储相关触发器触发数据拆分函数进行相关数据或者任务的拆分，然后再调用相关处理函数，处理完成之后，存储到云数据库中。例如：某证券公司每 12 小时统计一次该时段的交易情况并整理出该时段交易量 top 5；每天处理一遍秒杀网站的交易流日志获取因售罄而导致的错误从而分析商品热度和趋势等。函数计算近乎无限扩容的能力可以使用户轻松地进行大容量数据的计算。利用 Serverless 架构可以对源数据并发执行多个 mapper 和 reducer 函数，在短时间内完成工作；相比传统的工作方式，使用 Serverless 架构更能避免资源的闲置浪费从而节省成本，整个流程可以简化为：



1.3.4 人工智能领域

在 AI 模型完成训练后，对外提供推理服务时，可以使用 Serverless 架构，通过将数据模型包装在调用函数中，在实际用户请求到达时再运行代码。相对于传统的推理预测，这样做的好处是无论是函数模块还是后端的 GPU 服务器，以及对接的其他相关的机器学习服务，都是可以进行按量付费以及自动伸缩，从而保证性能的同时也确保了服务的稳定：



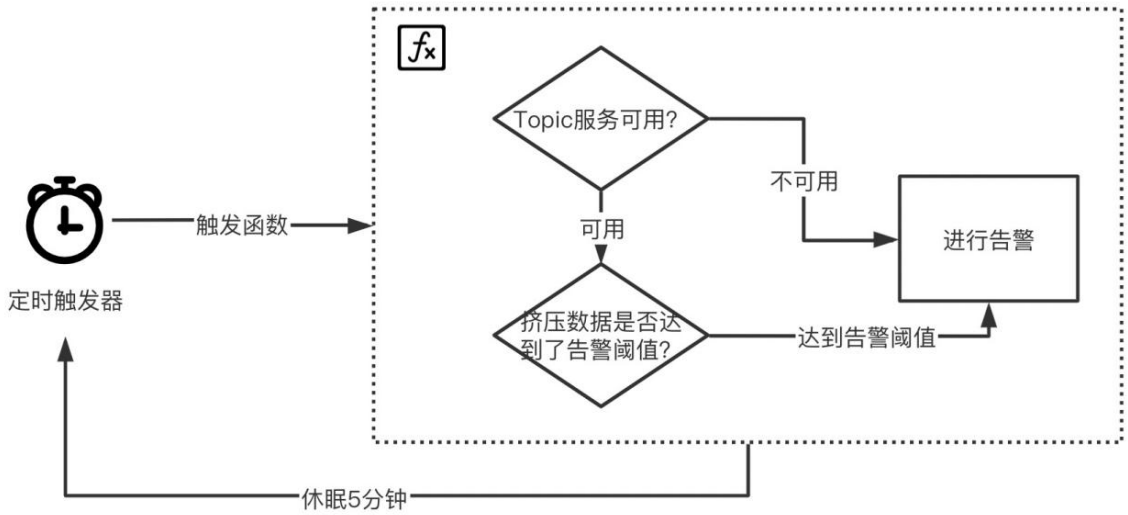
1.3.5 IoT 等物联网领域

目前很多厂商都在推出自己的智能音箱产品，用户可以对智能音箱说出一句话，智能音箱可以通过互联网将这句话传递给后端服务，然后获得到反馈结果，再返回给用户。通过 Serverless 架构，可以将 API 网关、云函数以及数据库产品进行结合来替代传统的服务器或者是虚拟机等，通过这样的一个架构，一方面，可以确保资源的按量付费，只有在使用的时候，函数部分才会计费，另一方面当用户量增加之后，通过 Serverless 实现的智能音箱系统的后端，也会进行弹性伸缩，可以保证用户侧的服务稳定，当要对其中某个功能进行维护，相当于对单个函数进行维护，并不会对主流程产生额外风险。相对来说会更加安全、稳定等：

1.3.6 监控与自动化运维

在实际生产中，经常需要做一些监控脚本来监控网站服务或者 API 服务是否健康，包括是否可用、响应速度等。传统的方法是通过一些网站监控平台（例如 DNSPod 监控、360 网站服务监控，以及阿里云监控等）来进行监控和告警服务，这些监控平台的原理是通过用户自己设置要监控的网址和预期的时间阈值，由监控平台部署在各地区的服务器定期发起请求对网站或服务的可用性进行判断。当然，这些服务很多都是大众化的，虽然说通用性很强，但是实际上并不一定适合。例如，现在需要监控某网站状态码，不同区域的延时，并且设置一个延时阈值，当网站状态异常或者延时过大时，通过邮件等进行通知告警，针对这样一个定制化需求，目前来说大部分的监控平台很难直接实现，所以定制开发一个网站状态监控工具就显得尤为重要。除此之外，在实际的生产运维中，还非常有必要对所使用的云服务进行监控和告警，例如在使用 Hadoop、Spark 的时候要对节点的健康进行监控，在使用 K8S 的时候要对 API Server、ETCD 等多维度的指标进行监控，在使用 Kafka 的时候，也要对数据积压量，以及 Topic、Consumer 等指标进行监控；这些服务的监控，往往不能通过简单的 URL 以及某些状态来进行判断，在传统的运维中，通常会在额外的机器上设置一个定时任务，对相关的服务进行旁路监控。

Serverless 架构的很重要的一个应用场景就是运维、监控与告警，通过与定时触发器进行结合使用，可以非常简单地实现某些资源健康状态监控与感知，并进行一些告警功能建设、自动化运维能力建设：



1.4 Serverless Devs 简介

1.4.1 工具链建设的必要性

Serverless 正在改变未来软件开发的模式和流程，并被预测将引领云计算的下一个十年，但尽管如此，开发者在选择使用 Serverless 时仍有诸多担忧，这其中最受关注的无疑就是工具链体系的匮乏。

所谓的工具链匮乏，一方面表现在市面上工具链不完善，这导致开发和部署难度大，进而增加成本；另一方面表现在，缺乏相关的工具链在体验层将 Serverless 体验进一步规范，优质工具链的匮乏导致本来就担心被厂商绑定的 Serverless 开发者变得更难与厂商解绑。2020 年 10 月，中国信息通信研究院发布国内首个《云原生用户调查报告》明确指出在使用 Serverless 架构之前，49% 的用户考虑部署成本，26% 的用户考虑厂商绑定情况，24% 的用户考虑相关工具集完善程度，这些数据背后透露的实际上是：开发者对于完善工具链的强烈需求。尽管，有一些开发者认为入门 Serverless 架构，通过白屏化的操作相对来说会更容易入门，在一定程度上通过各个云厂商的控制台进行函数的创建、更新也会更为方便。但是不可否定的是，Serverless 开发者工具在一定程度上却有着更为重要的价值和作用：

- 通过脚手架，快速创建 Serverless 架构的应用；
- 在开发过程中，通过开发者工具进行应用的调试等；
- 在开发完成之后，通过开发者工具将应用（可能包括多个函数以及相对应的 BaaS 类产品）一键部署到线上；
- 项目运维阶段，通过开发者工具进行项目的可观测以及问题定位等；
- 若需要实现科学部署，通过某些 CI/CD 平台/工具发布 Serverless 架构的应用，通常是离不开开发者工具的;
- 就目前来看，Serverless 领域的开发者工具复杂多样，且功能不完善；
- 并没有绝对统一/一致的 Serverless 开发者工具，每个厂商都有自己的开发者工具，而且使用形式，行为表现并不相同，这就导致了开发者在开发前的调研、开发中的调试、部署后的运维等多个层面面临很严峻的挑战；

- 绝大部分的 Serverless 开发者工具更多是资源编排、部署工具，并不能真正意义上称之为开发工具、运维工具，尤其在开发态的调试如何保证线上线下环境的一致性；如何在运维态可以快速的对业务进行调试；如何更简单的排查错误，定位问题.....等方面并没有统一的、完整的方案，这就导致 Serverless 架构的学习成本、使用成本对开发者来说会变的非常高；

综上所述，作为可以提升 Serverless 应用开发效能，降低 Serverless 架构使用难度的 Serverless 工具链体系建设，显得尤为重要，也正是因为此，Serverless Devs 应运而生。

Serverless Devs 是一个开源开放的 Serverless 开发者平台，致力于为开发者提供强大的工具链体系。通过该平台，开发者不仅可以一键体验多云 Serverless 产品，极速部署 Serverless 项目，还可以在 Serverless 应用全生命周期进行项目的管理，并且非常简单快速的将 Serverless Devs 与其他工具/平台进行结合，进一步提升研发、运维效能。

1.4.2 六大优势

- **无厂商锁定：**得益于功能的可插拔特性，可以非常简单的支持不同云厂商的项目部署，或者一键部署到不同云平台。目前 Serverless Devs 已经支持了阿里云函数计算、AWS Lambda 、百度智能云函数计算 、华为云函数工作流 、腾讯云云函数等多云的 FaaS 产品；

- **开源形式建设：**项目通过开源代码，开放生态进行建设，开发者可以随时查看和参与 Serverless Devs 开发者工具的贡献，也可以随时随地进行相关组件和应用的贡献。当然，除了这种开源开放的形态，也鼓励一些企业级团队，通过 Serverless Registry Model 建设自己的私有 Registry 以定制化某些不便公开的自定义组件；
- **功能灵活可插拔：**Serverless Devs 开发者工具本身，不具备任何业务能力，所有的业务能力均是通过组件化的形式，进行可插拔式使用，并且每个组件可以根据需要，自定义相对应的命令和功能；开发者可以在一个应用中，选择不同的组件完成对应的业务能力，以满足对不同模块的诉求；
- **简单快速上手：**通过开放 Serverless Registry 的模型/规范，该项目可以通过应用的模式，为开发者提供多种形式，多种领域以及多种场景的上手案例，帮助开发者快速了解、学习、深入、上手 Serverless 架构，例如新手引导中的 Serverless：Hello World；人工智能：目标检测；传统框架：基于 Django 的博客项目等项目；
- **应用全生命周期管理：**通过组件化的支持，Serverless Devs 可以在应用的全生命周期发挥重要作用，以阿里云函数计算的 FC 组件为例，开发者可以在项目创建、项目的开发、调试、可观测性等多个层面进行项目的建设和管理；
- **良好的集成与被集成性：**项目具有非常好的集成性与被集成性，可以通过组件化的支持，非常简单的与传统的生态进行有机结合。同时，Serverless Devs 开发者工具也可以非常简单的被集成到海量的自动化流程中，例如 CI/CD 文档中，就举例了与 Github Action 的集成、与 Gitee Go 的集成、与 Jenkins 的集成等平台集成的案例；

1.4.5 设计哲学

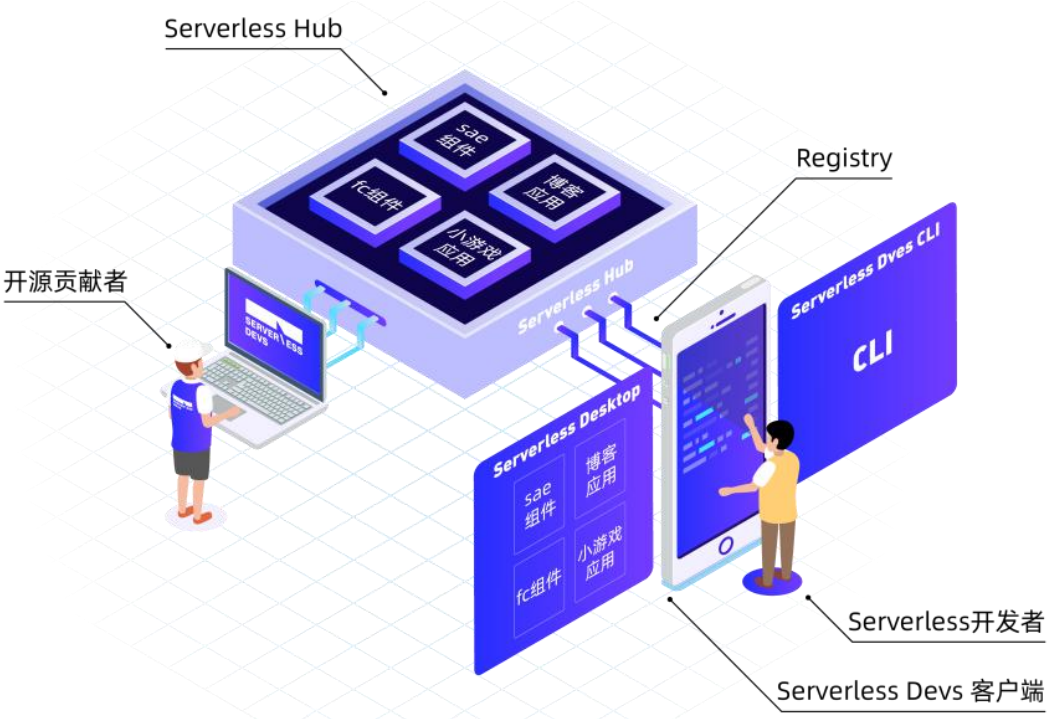
Serverless Devs 是一个开源开放的 Serverless 领域的工具链项目，它不仅仅表示单纯的某个命令行工具，在一定程度上指的是一个完整的工具链体系。

在 Serverless Devs 中，拥有两个角色：

- **开源贡献者：**开源贡献者将按照 Serverless Package Model 进行组件/应用的开发，并将内容发布到 Serverless Hub 中，既可以被更多人所使用；
- **Serverless 开发者：**通过开发者工具（包括命令行工具以及桌面端等工具），进行应用的初始化，以及组件的使用；通过开发者工具，将业务按照预期部署到线上。

在这样一个 Serverless Devs 的应用框架上，不难发现可以和其他任何一种模式/生态，具有相似的命名以及模块：

- **Serverless Hub**：类似于一种组件、应用、案例中心；类似于 Docker Hub 等；
- **Serverless Registry**：类似于一种组件、应用的管理工具或者规范模型；类似于 Python 生态中的 Pypi，类似于 Node.js 生态中的 NPM；



当然，在 Serverless Hub 中，有两种形态的 Package（组件和应用）,同时通过上图也可以看到两个比较明显的词汇：Component 和 Application：

- **Component**：指的是组件；是由 Package developer 开发并发布的符合 Serverless Package Model 规范的一段代码，通常这段代码会在应用中被引用，并在 Serverless Devs 开发者工具被加载，并按照预定的规则进行执行某些动作。例如，将用户的代码部署到 Serverless 平台；将 Serverless 应用进行构建和打包；对 Serverless 应用进行调试等；
- **Application**：指的是应用，可以由 Package developer 公开发布到 Registry，以供更多人学习和使用，例如某位贡献者贡献了一个猫狗识别的案例到 Registry；也可以由 Serverless developer 开发，例如某人开发了一个识别的应用；通常情况下一个应用可以引用一个或者

多个组件，并通过 Serverless Devs 开发者工具部署到 Serverless 平台，例如我开发了一个猫狗识别的应用，在这个应用中引用了 Lambda 组件帮助我将部分业务逻辑部署到 FaaS 平台，同时我也引用了 Website 组件帮助我把前端业务代码部署到对象存储中。

Serverless 工具链体系，让开发者更专注于业务逻辑，提升 Serverless 应用开发、部署、运维效率，通过该模型，开发者可以通过一种更灵活、更通用的方法使用不同云厂商以及开源的 Serverless 产品，进而更高效、更简洁、更便利的实现 Serverless 应用管理。

成长历史

如果说 Serverless 提升了传统应用的开发效能，那么 Serverless Devs 开发者工具就是提升了 Serverless 应用开发的效能。随着时间的发展，Serverless Devs 也从 1.0 版本 到了 2.0 版本，更是从简单的效能提升，变成了更加规范、更加科学的效能提升。希望可以通过 Serverless Devs 的工具链模式和思路，为应用的开发，传统项目上 Serverless 架构提供巨大的便利和更科学的管理。

- 2020 年 10 月 23 日，Serverless 开发者平台 Serverless Devs 正式开源
- 2020 年 11 月，Serverless Devs 被 CNCF Landsacpe 收录，成为国内首个进驻的 Serverless 工具
- 2020 年 11 月，Serverless Developer Meetup 首召开，成 Serverless 开发者技术新渠道
- 2020 年 11 月，入围 InfoQ 评选 2020 年度十大开源新锐项目

- 2020 年 12 月，Serverless Devs 下载次数突破 5000，组件&应用累计使用次数突破 10000
- 2021 年 4 月，Serverless Developer Meetup 在上海召开，并正式发布 Serverless Devs 2.0 版本
- 2021 年 6 月，Serverless Devs 下载次数突破 10000
- 2021 年 7 月，Serverless Developer Meetup 在杭州召开，阿里云函数计算团队在会上正式发布端云联调、桌面客户端等功能
- 2021 年 8 月，Serverless Devs 下载次数突破 20000
- 2021 年 10 月，在 2021 OpenInfra Days China 会议上，Serverless Devs 带来了《Serverless Devs：Serverless 全生命周期的工具链建设》的主题演讲
- 2021 年 12 月，Serverless Developer Meetup 在深圳召开，并尝试性的对外展示了 Serverless Devs Model

1.5 阿里云函数计算（FC）组件简介

阿里云函数计算（FC）组件是一款符合 Serverless Devs 规范的组件，通过在 Serverless Devs 开发者工具中，使用该组件，可以快速帮助开发者使用阿里云函数计算产品。

1.5.1 Serverless Devs 和 FC 组件的关系

1、Serverless Devs 是一个无厂商锁定 Serverless 的工具框架，本身不具任何能力，用户可以通过引入不同的组件使用不同的功能，例如：

2、而 FC 组件则是这个工具框架的一个组件，主要是对阿里云函数计算进行操作的，例如创建函数，删除函数、发布版本、业务构建、在线调试等；

如果需要进行比喻：

- Serverless Devs 是小时候玩的红白机，而 fc 组件，oss 组件，nas 组件等都是游戏卡，游戏机本身不具备啥功能，根据插入的游戏卡实现不同的功能；
- Serverless Devs 就相当于使用 VSCode 工具，本身不具备太多的能力，但是可以安装不同的插件，来丰富 VSCode 的能力，而这些插件对应到 Serverless Devs 生态中，就是不同的组件，例如 FC 组件，nas 组件，oss 组件等；

1.5.2 五大亮点

- **全生命周期管理：**组件拥有项目的创建、开发、调试、部署、运维全生命周期管理能力；
- **安全发布：**通过其他形式对函数进行变更，组件可以感知并安全更新；
- **快速集成：**借助于 Serverless Devs 的集成性和被集成性，可以与常见的 CI/CD 平台工具等集成；
- **可观测性：**拥有完善的可观测性，在客户端可以通过指标查询 metrics 以及日志查询 logs 等命令，进行业务的数据指标、执行日志等多重维度观测；
- **多模调试：**提出了多模调试方案，可以同时满足开发态、运维态的不同调试需求；包括本地运行、在线运行、云端调试、端云联调等功能；

1.5.3 功能支持

阿里云函数计算（FC）组件可参与到项目的创建、开发、调试、部署与运维的全流程中：

- 在项目的创建阶段，可通过命令行工具或者应用中心进行项目的最初创建；
- 在项目开发过程中，可以通过本地开发、调试等能力来验证本地开发的正确性；
- 在项目调试的环节，可以通过本地调试与远程调用、日志查询等能力，来进行项目的最终调试；
- 在部署环节，可以先通过依赖安装、项目构建等流程构建出完整的部署包，再进行项目的部署；
- 在后期运维环节，可以通过指标查询来进行项目健康度检查，通过日志查询等来进行问题定位，通过项目发布等能力进行版本发布，别名发布以及灰度发布等；

目前为止，阿里云函数计算（FC）组件已经支持的能力包括：

构建&部署	**可观测性**	**调用&调试**	**发布&配置**	**其他功能**
部署 deploy	指标查询 metrics	本地调用 local	版本 version	硬盘挂载 nas
构建 build	日志查询 logs	函数触发 invoke	别名 alias	Fun 项目迁移 fun2s
移除 remove		端云联调 proxied	预留 provision	查看函数 info
计划变更 plan		实例登陆 instance	按量资源 ondemand	资源同步 sync
		内存&并发度探测 eval	层 layer	压测 stress
				API 调用 api

2. 快速入门与功能案例

2.1 Serverless Devs 基础入门

2.1.1 工具安装

- 第一步：安装 Node.js(>=10.8.0) 与 NPM 包管理工具；
- 第二步：安装 Serverless Devs 开发者工具；

```
$ npm install @serverless-devs/s -g
```

- 第三步：可以通过 s -v 判断工具是否安装成功，如果安装成功可以看到相对应的版本信息，例如：

```
@serverless-devs/s: 2.0.89, @serverless-devs/core: 0.1.7, arwin-x64, node-v12.15.0
```

2.1.2 密钥配置

2.1.2.1 获取密钥

配置 Serverless Devs 的阿里云密钥，一般需要密钥信息，获取页面为：
https://usercenter.console.aliyun.com/#/manage/ak

- AccessKeyID：用户的 AK 信息
- AccessKeySecret：用户的 SK 信息

关于密钥信息的获取流程如下：打开获取密钥信息

2.1.2.2 引导式密钥配置

通过引导式进行密钥配置，可以通过 s config add 命令，进行引导式创建：

- 执行 s config add，并选择 Alibaba Cloud (alibaba)：

```
$ s config add
? Please select a template: Alibaba Cloud (alibaba)
    Refer to the document for alibaba key:  http://config.devsapp.net/account/alibaba
? AccountID ()
```

- 此时，可以按照引导，进行密钥的配置：

```
? Please select a template: Alibaba Cloud (alibaba)
    Refer to the document for alibaba key:  http://config.devsapp.net/account/alibaba
? AccessKeyID 此处填写 AccessKeyID
? AccessKeySecret 此处填写 AccessKeySecret
? Please create alias for key pair. If not, please enter to skip alibaba-access

    Alias: alibaba-access
    AccessKeyID: 此处填写 AccessKeyID
    AccessKeySecret: 此处填写 AccessKeySecret
Configuration successful
```

- 为了验证密钥是否正确配置，可以通过 s config get -a alibaba-access 进行指定密钥的查看：

```
$ s config get -a alibaba-access
alibaba-access:
  AccountID: 此处填*******tID
  AccessKeyID: 此处填*********yID
  AccessKeySecret: 此处填*************ret
```

致此，通过引导式，完成了密钥配置信息。

2.1.2.3 命令式密钥配置

为了在一些自动化流程中可以更好的使用 Serverless Devs，所以除了通过引导式进行密钥的配置， Serverless Devs 还支持通过命令行非交互式进行密钥的配置。

同样以阿里云密钥配置为例，可以直接通过参数将密钥信息传入：

```
$ s config add --AccessKeyID myAccessKeyID --AccessKeySecret myAccessKeySecret -a
 demo

Alias: demo
      AccountID: myAccountID
      AccessKeyID: myAccessKeyID
      AccessKeySecret: myAccessKeySecret

Configuration successful
```

如果需要配置更多格式的密钥信息，可以通过自定义 Key-Value 实现，例如：

```
$ s config add --AccessKeyID ****** -kl key1,key2,key3 -il info1,info2,info3 -a demo-2

    Alias: demo-2
    key1: info1
    key2: info2
    key3: info3
    AccessKeyID: code

Configuration successful
```

2.1.2.4 通过环境变量配置

有相当一部分的开发者会将密钥信息放在环境变量中，这样在使用工具的时候，就需要从环境变量中读取密钥信息，此时，通过环境变量配置密钥的方法就显得尤为重要，为此，Serverless Devs 提供了两种通过环境变量配置密钥的方法：

方法 1：直接通过 config add 进行配置

这种方法很简单，基本和上面所描述的命令式密钥配置是类似的，只不过传入的不是固定值，而是环境变量，例如在环境变量中有 ALIBABA_CLOUD_ACCESS_KEY_ID、ALIBABA_CLOUD_ACCESS_KEY_SECRET 等相关内容，此时可以通过 s config add 命令进行添加：

```
$ s config add -a default-aliyun -kl AccessKeyID,AccessKeySecret -il ${ALIBABA_CLOUD_ACCESS_KEY_ID},${ALIBABA_CLOUD_ACCESS_KEY_SECRET}
```

方法 2：通过指定名称使用环境变量密钥

通过指定环境变量的名字进行配置：例如当前有阿里云密钥对：

```
AccountID: temp_accountid
AccessKeyID: temp_accesskeyid
```

```
AccessKeySecret: temp_accesskeysecret
```

此时可以在环境变量中可以命名 key 为*********_serverless_devs_access，例如 default_serverless_devs_access，value 为 JSON 字符串，例如：

```
Key：default_serverless_devs_access
Value：{\"AccessKeyID\":\"temp_accesskeyid\",\"AccessKeySecret\":\"temp_accesskeysecret\"}
```

2.1.2.5 密钥使用注意事项

2.1.2.5.1 安全相关

云账号 AccessKey 是您访问阿里云 API 的密钥，具有该账户完全的权限，请您务必妥善保管！不要通过任何方式（e.g. Github）将 AccessKey 公开到外部渠道，以避免被他人利用而造成威胁 。强烈建议开发者遵循阿里云安全最佳实践 ，使用 RAM 子用户 AccessKey 来进行 API 调用。

2.1.2.5.2 关于密钥配置中 Alias 的设计思路

在 Serverless Devs 中，除了配置云厂商所提供的密钥信息之外，还需要额外进行 Alias 的设置，所谓的 Alias 是指对密钥进行的别名设置。由于 Serverless Devs 支持多密钥的配置和管理，所以一般情况下，一个别名对应一个密钥对。相关的最佳实践可以是：

- 有两个账号，分别是阿里云账号和腾讯云账号，那么配置密钥的时候就可以设置别名 alibaba、tencent，在使用的时候，通过引用不同别名使用不同的密钥，以防止每次切换密钥的时候，进行密钥重新配置；
- 自己拥有两个环境的密钥，一个是测试环境密钥 test，一个是线上环境密钥 release，当开发完成之后需要把业务部署到不同的环境下，可以通过指定密钥的形式，直接进行部署，而无需因为密钥的切换反复进行密钥的重新配置；

2.1.2.5.3 密钥使用方法

在 Serverless Devs 中，密钥的使用主要在两个层面：

- 命令行层面：在命令行中使用的时候，可以直接通过-a/--access 参数进行使用，例如在部署某业务的时候，可以通过 s deploy -a demo 指定使用 demo 密钥对；
- Yaml 配置文件层面：可以通过在 Yaml 中进行密钥对的指定，例如：

```yaml
edition: 1.0.0
name: hello-world-app
access: demo

vars: # 全局变量
  region: cn-hangzhou
  service:
    name: hello-world-service
    description: 'hello world by serverless devs'

services:
  helloworld: # 业务名称/模块名称
    component: fc
#   access: test
#   actions: # 自定义执行逻辑
#     pre-deploy: # 在deploy之前运行
#       - run: s build --use-docker  # 要运行的命令行
#         path: ./ # 命令行运行的路径
#     post-deploy: # 在deploy之后运行
```

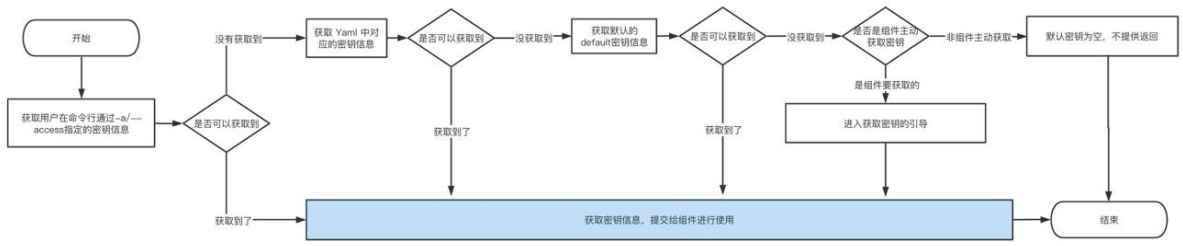
在 Yaml 的中直接指定 access，表示整个应用都通过当前密钥对进行部署，也可以在某个模块/业务下指定当前模块/业务使用某指定的密钥对进行部署。

2.1.2.5.4 密钥使用顺序相关

密钥支持多种形式的使用，也就会出现密钥的使用的顺序问题：

- 通过 -a/--access 参数指定的密钥信息
- 使用已经配置的 default 密钥信息
- 使用通过环境变量配置的``default_serverless_devs_access`密钥信息
- 不使用密钥信息 / 进入密钥信息配置引导

具体的流程图为：



2.1.2.5.5 密钥的其他相关操作

在 Serverless Devs 中，除了配置密钥之外还包括密钥的修改、删除和查看，此时可以通过 s config -h 进行相关功能的查看。

- 密钥的查看，可以通过 s config get -h 查看帮助；
- 密钥的修改，可以重新进行指定别名的密钥的创建，并通过-f 参数，强行覆盖；
- 密钥的删除，可以通过 s config delete-h 查看帮助；

2.1.3 Yaml 的使用规范

Serverless Devs 可以通过指定格式的 Yaml 对 Serverless 应用进行描述，在 Serverless Devs 的规定中，一个 Yaml 可以被认为是一个 Serverless 应用。

Yaml 的格式需要按照 Serverless Devs 的规范，提供相对应的资源/行为描述文件，且该文件还需要符合以下条件：

- 拓展名可以是.yaml 或.yml
- 格式必须符合 Yaml 规范

对于需要通过描述文件进行环境隔离的项目，建议将文件命名为 s-${ENV}.yaml 或 s-${ENV}.yml 格式。 例如：s-prod.yaml。

在默认情况下，Serverless Devs 开发者工具会默认该描述文件的名称为 s.yaml 或 s.yml，且 s.yaml 的优先级大于 s.yml， 即在一个 Serverless 应用下，同时出现 s.yaml 与 s.yml 时，系统

会优先识别和使用 s.yaml。

当然，开发者也可以通过-t, --template [templatePath]进行指定，例如，在某应用在生产环境下的描述文件名为 s-prod.yml，则可以在执行相关命令时，增加参数-t s-prod.yml 或者--template s-prod.yml。

2.1.3.1 描述文件格式/规范

关于 Serverless Devs 所支持的资源/行为描述文件基本格式为：

```
edition: 1.0.0              #  命令行 YAML 规范版本，遵循语义化版本（Semantic Versioning）规范
name: applicationName     #  应用名称
access: xxx-account1      #  秘钥别名

vars: # [全局变量，提供给各个服务使用]
  Key: Value

Service: # 可以包括多个服务
  ServiceName: # 服务名称
    access: xxx-account1        #  秘钥别名，如果和项目的 access 相同，可省略
    component: componentName   #  组件名称
    props: serviceProp          #  组件的属性值
    actions: serviceActions     #  自定义执行逻辑
```

例如，一个相对完整的 Yaml 案例可以是：

```
edition: 1.0.0              #  命令行 YAML 规范版本，遵循语义化版本（Semantic Versioning）规范
name: FullStack           #  项目名称
access: xxx-account1   #  秘钥别名
```

```
vars: # [全局变量，提供给各个服务使用]
  logo: https://image.aliyun.com/xxxx.png

services:
  nextjs-portal: #  服务名称
    access: xxx-account1  #  秘钥别名，如果和项目的 access 相同，可省略
    component: vue-component  # 组件名称
    props: #  组件的属性值
      src: ./frontend_src
      url: url
    actions: # 自定义执行逻辑
      pre-deploy: # 在 deploy 之前运行
        - run: s exec -- publish  # 要运行的命令行
          path: ./backend_src # 命令行运行的路径
        - run: s build  # 要运行的命令行
          path: ./backend_src # 命令行运行的路径
      post-deploy: # 在 deploy 之后运行
        - run: s clean
          path: ./frontend_src

  assets:
    component: static
    props:
      cache-control: "public, max-age=604800, immutable"
      www: "./public"

  express-blog:
    component: express
    props:
      app: ./express-blog
```

```
    url: ${vars.domain}
  actions:
    pre-deploy:
      - run: npm run build
        path: ./express-blog

gateway:
  component: serverless-gateway # 路由组件：HTTP URL 和服务之间的映射规则
  props:
    routes:
      - route: /~assets
        value: ${assets.output.url}
      - route: /
        value: ${nextjs-portal.output.url}
        index: index.html
      - route: /~portal
        value: ${nextjs-portal.output.url}
        inex: index.html
      - route: /~blog
        value: ${express-blog.output.url}
```

2.1.3.2 元数据相关描述

在该格式中：

参数名	代表含义
edition	命令行 YAML 规范版本，遵循语义化版本（Semantic Versioning）规范
name	应用名称
access	秘钥别名，可以使用通过 config 命令配置的密钥信息，以及配置到环境变量的密钥信息
vars	全局变量，提供给各个服务使用，是一个 Key-Value 的形式
Service	应用所包含的服务，是一个 Key-Value 的形式

关于 Service 参数：

参数名	代表含义
access	秘钥别名，如果和项目的 access 相同，可省略
component	组件名称
actions	自定义执行逻辑
props	组件的属性值

2.1.3.3 变量赋值

Serverless Devs 的 Yaml 文件支持多种变量格式：

- 获取当前机器中的环境变量：${env(环境变量)}，例如${env(secretId)}
- 获取外部文档的变量：${file(路径)}，例如${file(./path)}
- 获取全局变量：${vars.*}
- 获取其他项目的变量：${projectName.props.*}
- 获取 Yaml 中其他项目的结果变量：${projectName.output.*}

2.1.3.4 服务顺序

如果一个 Serverless Application 模型对应的 Yaml 文件中有过多的服务，系统会默认分析部署顺序，该部署顺序分为两个步骤：

- 分析项目中的依赖关系
- 有依赖关系的按照依赖关系从前到后部署，无依赖关系的按 Yaml 配置的从上到下部署

2.1.3.5 行为描述

在 Serverless Application 模型对应的 Yaml 文件中，可以针对服务，提供对应的行为操作，其基本格式是：

```
actions: # 自定义执行逻辑
  pre-命令: # 在命令之前运行
    - run: command  # 要运行的操作
      path: ./path # 运行操作的路径
  post-命令: # 在命令之后运行
    - run: command  # 要运行的操作
      path: ./path # 运行操作的路径
```

例如：

```
actions: # 自定义执行逻辑
  pre-deploy: # 在 deploy 之前运行
    - run: s exec -- publish  # 要运行的命令行
      path: ./backend_src # 命令行运行的路径
    - run: s build  # 要运行的命令行
      path: ./backend_src # 命令行运行的路径
  post-deploy: # 在 deploy 之后运行
    - run: s clean
      path: ./frontend_src
```

当 Serverless Devs 开发者工具执行到该服务时，会在进行相关的命令之行之前，优先按照顺序执行 pre-命令的操作，所有内容完成执行之后，再执行 post-命令的操作。

以下面的 Yaml 为例：

```
edition: 1.0.0          #  命令行 YAML 规范版本，遵循语义化版本（Semantic Versioning）规范
name: FullStack         #  项目名称

services:
  nextjs-portal: #  服务名称
    access: xxx-account1  #  秘钥别名，如果和项目的 access 相同，可省略
    component: vue-component  # 组件名称
    props: #  组件的属性值
      src: ./frontend_src
      url: url
    actions: # 自定义执行逻辑
      pre-deploy: # 在 deploy 之前运行
        - run: s exec -- publish  # 要运行的命令行
          path: ./backend_src # 命令行运行的路径
        - run: s build  # 要运行的命令行
          path: ./backend_src # 命令行运行的路径
      post-deploy: # 在 deploy 之后运行
        - run: s clean
          path: ./frontend_src
```

当开发者在当前应用下执行了 deploy 命令，系统将会按照以下顺序进行操作：

- 在./backend_src 目录下执行 s exec -- publish
- 在./backend_src 目录下执行 s build

- 调用组件 vue-component 的 deploy 方法，并将 props 和项目的基本信息传入到组件 vue-component 的 deploy 方法中
- 在./frontend_src 目录下执行 s clean

以上顺序仅适用于整个流程没有出错的前提下，如果流程出现错误，系统将会进行报错，并终止后续流程的执行。

2.1.3.6 通过命令操作应用

所谓的自定义命令指的是由组件决定的命令。由于 Serverless Devs 开发者工具，本身并不具备任何业务相关的能力（值得包括不限于函数的部署、应用的构建、项目的测试等），所以，这些能力都将会由组件提供，通过 Serverless Devs 开发者工具进行透出。

例如，某应用的资源/行为描述文件如下：

```
edition: 1.0.0          #  命令行 YAML 规范版本，遵循语义化版本（Semantic Versioning）规范
name: FullStack         #  项目名称
access: xxx-account1

services:
  backend: #  服务名称
    component: django-component  # 组件名称
    props: #  组件的属性值
      src: ./backend_src
      url: url
  user—frontend: #  服务名称
    component: vue-component  # 组件名称
    props: #  组件的属性值
      src: ./frontend_src_user
      url: url
```

```
admin-frontend:  #   服务名称
  component:  vue-component   # 组件名称
  props:  #   组件的属性值
    src:  ./frontend_src_admin
    url:  url
```

通过该 Yaml 文件可以看出以下信息：

- 该应用的名字是 FullStack，将会使用密钥 xxx-account1;
- 该应用拥有三个服务：
  - backend 服务：使用了 django-component 组件
  - user—frontend 服务：使用了 vue-component 组件
  - admin-frontend 服务：使用了 vue-component 组件

如果此时 django-component 组件和 vue-component 组件支持的自定义命令为：

	django-component	**vue-component**
deploy	支持	支持
remove	支持	支持
test	支持	不支持

则可以通过自定义命令实现应用级操作和服务级操作。

2.1.3.6.1 应用级操作

在当前项目下，可以执行 s [自定义命令]实现应用纬度的操作。

- 执行 s deploy 或者 s remove 时，由于 backend、user—frontend、admin-frontend 三个服务对应的组件，均支持 deploy 和 remove 方法，所以此时系统会按照服务的部署顺序，进行三个服务分别对应的组件的 deploy 或 remove 操作；此时，系统的 exit code 为 0；

- 执行 s test 时，由于 user—frontend、admin-frontend 两个服务对应的组件并不支持 test 方法，所以此时系统会执行 backend 对应组件（django-component）的 test 操作；此时，系统会对 user—frontend、admin-frontend 两个服务进行警告，但是并不会报错，最终的 exit code 为 0；
- 如果在执行相关的命令时，backend、user—frontend、admin-frontend 三个服务任何一个服务在执行过程中出现了错误，系统则会报错，并终止下一步的操作，此时，系统的 exit code 为 101；

2.1.3.6.2 服务级操作

在当前项目下，可以执行 s [服务名] [自定义命令]实现服务级操作。

- 执行 s backend deploy 等，可以针对服务 backend 进行 deploy 相关的操作，**如果顺利完成与其操作，系统的 exit code 为 0；否则，出现错误，系统的 exit code 为 101**；
- 执行 s admin-frontend test 是，由于服务 admin-frontend 对应的 test 方法是不存在的，**此时系统将会认为是未找到组件方法，系统的 exit code 为 100；**

2.1.4 多种操作模式下的工具体系

众所周之，目前大部分的 Serverless 开发者工具均是通过 Yaml 等进行资源描述，少部分的 Serverless 开发者工具是通过命令行直接操作，无论是通过 Yaml 进行资源描述，还是通过纯命令行的操作，可以说两者各有各的好处。而在 Serverless Devs 开发者工具中，这两者均得以有效的支持。

Serverless Devs 开发者工具从根本上提供了两种使用方法。

- Yaml 模式：需要依赖资源描述文档进行操作的模式；
- Cli 模式：可以在任何目录下直接执行，而不需要依赖资源描述文档；

这两者的核心区别是：

- 如果想要使用 Yaml 模式，在当前目录下，必须要有 s.yaml/s.yml 文件，或通过-t/--template 指定的资源部描述文件；
- 如果想要试用 Cli 模式，则必须是 s cli 组件名 方法 参数的格式进行，此时不需要 Yaml 文件；

举一个非常简单的例子，如果有一个应用的资源描述文件 s.yaml 如下：

```
name: myApp
edition: 1.0.0
access: "myaccess"

services:
  website-starter:
    component: devsapp/website
    props:
      bucket: testbucket
  backend-starter:
    component: devsapp/demo
    props:
      service:
        name: serviceName
      function:
        name: functionName
      region: cn-hangzhou
```

此时，可以执行 s deploy 进行 myApp 应用部署，如果执行 s backend-starter deploy 则可以进行 myApp 应用下的 backend-starter 项目/服务部署。

此时，部署过程中，所需要的相关参数，可以通过该 Yaml 文件进行读取。

但是，在某些情况下，并不方便直接使用 Serverless Devs 规范的 Yaml 文件（例如，将线上资源同步到本地，或者要将 Funcraft 的 Yaml 转换成为 Serverless Devs 的 Yaml），此时可以选择纯命令行形式，即 s cli 模式。

在 s cli 模式下，由于不会读取 Yaml 等资源描述文件，所以很多参数都需要自行填写，这时的填写方法有两种：

- 通过 s cli 天然支持的 -p/--prop 参数，进行相关 Yaml 参数的赋值，例如上述案例的 s backend-starter deploy，此时可以改写成：

```
$ s cli devsapp/demo -p "{\"service\":{\"name\":\"serviceName\"},\"function\":{\"name\":\"functionName\"},\"region\":\"cn-hangzhou\"}"
```

- 通过 demo 组件本身所支持的一些参数，例如通过 s cli devsapp/demo -h，可以得到帮助信息，部分内容如下：

```
--region [region]                [C-Required] Specify the fc region, value: cn-hangzhou/cn-beijing/cn-beijing/cn-hangzhou/cn-shanghai/cn-qingdao/cn-zhangjiakou/cn-huhehaote/cn-shenzhen/cn-chengdu/cn-hongkong/ap-southeast-1/ap-southeast-2/ap-southeast-3/ap-southeast-5/ap-northeast-1/eu-central-1/eu-west-1/us-west-1/us-east-1/ap-south-1
--service-name [serviceName]     [C-Required] Specify the fc service name
--function-name [functionName]   [Optional] Specify the fc function name
```

此时，就可与通过下面的命令实现上述功能：

```
$ s cli devsapp/demo --region cn-hangzhou --service-name serviceName --function-name functionName
```

2.1.4.1 特点对比

模式	使用方法	优势	劣势	适用场景
Yaml 模式	在具有符合 Serverless Devs 规范，且存在资源/行为描述的 Yaml 文件的应用目录下，执行组件对应的命令，即可直接使用，例如 s deploy，s service name build 等	可以一键部署一个完整的应用（例如，某个应用中规定了多个 Service，可以通过该命令一键部署）；同时，通过资源/行为描述文档，可以更佳简单，清晰的对应用进行描述；	需要学习 Yaml 的规范，且在某些时候与一些自动化流程进行结合，会比较复杂；	部署、运维等操作，尤其是批量操作时更为合适；
纯 Cli 模式	在任何目录下，通过子命令 cli 进行触发，同样适用全部组件，例如 s cli deploy -p "{/"function/": /"function-name/"}"，s cli fc-api listFunctions --service-name my-service	相对来说可以更加简单，快速上手工具，并且可以非常简单的与自动化流程进行结合，降低了 Yaml 格式/规范的学习难度	对于一些复杂项目而言，需要在命令行中写过多的参数，出错的概率会比较高；	更适合项目的管理，源自化操作

2.1.4.2 设计思路

为什么要同时存在 Yaml 模式和 Cli 模式?

因为在长期的实践过程中，发现通过 Yaml 进行资源描述会相对来说更简单和方便，例如 K8S 等也都是通过 Yaml 进行资源描述的；但是，在某些情况下，Yaml 文件也可能成为一种负担，例如想要查看某个服务下的函数列表，查看某个地区下的服务列表，因为这样一个简单的事情要额外的去完成一个 Yaml 文件，就显得过于臃肿，所以，在 Serverless Devs 项目中，同时保留了两种使用方法。

2.2 案例与体验

2.2.1 项目上手体验

2.2.1.1 Serverless：Hello World

- 执行 s 命令：

```
$ s
? Serverless Devs project is not detected. Do you want to create a new project? (Y/n)
```

- 填写 y，并按回车，可以进入到创建引导部分：

```
  Serverless Awesome: https://github.com/Serverless-Devs/package-awesome

? Hello Serverless for Cloud Vendors (Use arrow keys or type to search)
❯ Alibaba Cloud Serverless
  AWS Cloud Serverless
  Baidu Cloud Serverless
  Huawei Cloud Serverless
  Tencent Cloud Serverless
  Dev Template for Serverless Devs
```

- 此时只需要选择对应的选项，按照引导进行操作，即可。例如选择 Alibaba Cloud Serverless，就可以看到阿里云 Serverless 产品下的应用模板分类:

```
? Please select an Serverless-Devs Application (Use arrow keys or type to search)
❯ fc-runtime-starter - 快速部署一个 FC 函数
  fc-custom-container-stater - 快速部署一个 custom-container 应用
  web-framework-stater - 快速部署一个 Web 框架
  static-website-stater - 快速部署一个静态网站
  serverless-best-practice - 快速体验 Serverless 最佳实践
```

- 此时可以继续选择某分类下的具体应用进行初始化，例如选择 fc-runtime-starter 之后，可以看到该分类下的具体模板应用：

```
? Please select an templete (Use arrow keys or type to search)
❯ fc-http-nodejs - 快速部署一个 nodejs12 http 函数
  fc-http-python - 快速部署一个 python3 http 函数
  fc-http-php - 快速部署一个 php http 函数
  fc-http-java - 快速部署一个 java8 http 函数
  fc-event-nodejs - 快速部署一个 nodejs12 event 函数
  fc-event-python - 快速部署一个 python3 event 函数
  fc-event-php - 快速部署一个 php event 函数
```

选择 fc-http-nodejs 即可完成创建，在引导的过程中，可能会出现填写项目名称以及选择密钥的过程：

  - 项目名称可以是：start-fc-http-nodejs12
  - 密钥可以选择上文中创建过的：alibaba-access

例如：

```
  Serverless Awesome: https://github.com/Serverless-Devs/package-awesome

? Hello Serverless for Cloud Vendors Alibaba Cloud Serverless
```

```
? Please select an Serverless-Devs Application fc-runtime-starter - 快速部署一个 FC 函数
? Please select an templete fc-http-nodejs - 快速部署一个 nodejs12 http 函数

    Create application command: [s init devsapp/start-fc-http-nodejs12]

? Please input your project name (init dir) start-fc-http-nodejs12
✔ file decompression completed
? please select credential alias alibaba-access

... ...

    Thanks for using Serverless-Devs
    You could [cd /Users/jiangyu/start-application/start-fc-http-nodejs12] and enjoy your serverless journey!
    If you need help for this example, you can use [s -h] after you enter folder.
    Document ❤ Star：https://github.com/Serverless-Devs/Serverless-Devs

? 是否立即部署该项目？ (Y/n)
```

可以看到，系统在最后有一个提醒，是否要部署该项目，此时可以输入 y，直接进行项目的部署，稍等片刻，可以看到部署结果：

```
fc-deploy-test:
  region:     cn-hangzhou
  service:
    name:  fc-deploy-service
  function:
    name:           http-trigger-nodejs12
    runtime:       nodejs12
    handler:       index.handler
```

```
    memorySize: 128
    timeout:    60
  url:
    system_url:     https://1583208943291465.cn-hangzhou.fc.aliyuncs.com/2016-08-15/proxy/fc-deploy-service/http-trigger-nodejs12/
    custom_domain:
      -
        domain: http://http-trigger-nodejs12.fc-deploy-service.1583208943291465.cn-hangzhou.fc.devsapp.net
  triggers:
    -
      type: http
      name: httpTrigger
```

此时可以打开 domain 返回的域名，进行测试。

2.2.1.2 人工智能：目标检测

- 初始化一个已有的人工智能目标检测项目：s init devsapp/image-prediction-app，初始化过程中可能会出现填写项目名称以及选择密钥的过程：
  - 项目名称可以是：image-prediction-app
  - 密钥可以选择上文中创建过的：alibaba-access

例如：

```
$ s init devsapp/image-prediction-app

   Serverless Awesome: https://github.com/Serverless-Devs/package-awesome

? Please input your project name (init dir) image-prediction-app
✔ file decompression completed
```

```
? please select credential alias alibaba-access


      ___     __     __   ________   ________   ________
     |    | |  |_|   ||    _    ||          ||          |
     |    | |         ||   |_|   ||      ____||      ____|
     |    | |         ||         ||     | __ |     |____
     |    | |         ||         ||     ||  ||      ____|
     |    | | ||_|| ||    _    ||    |_| ||     |____
     |____| |_|    |_||__| |__||________||________|


      Welcome to the image-prediction-app application
       This application requires to open these services:
            FC : https://fc.console.aliyun.com/
       This application can help you quickly deploy the image-prediction-app project.
       The application uses FC component: https://github.com/devsapp/fc
       The application homepage: https://github.com/devsapp/image-prediction-app


    Thanks for using Serverless-Devs
    You could [cd /Users/jiangyu/start-application/image-prediction-app] and enjoy you
r serverless journey!
    If you need help for this example, you can use [s -h] after you enter folder.
    Document ❤ Star: https://github.com/Serverless-Devs/Serverless-Devs
```

- 进入项目目录：cd image-prediction-app
- 通过 deploy 命令进行项目的部署：

```
Tips for next step
======================
* Display information of the deployed resource: s info
* Display metrics: s metrics
* Display logs: s logs
* Invoke remote function: s invoke
* Remove Service: s remove service
* Remove Function: s remove function
* Remove Trigger: s remove trigger
* Remove CustomDomain: s remove domain

imageAi:
  region: cn-hangzhou
  url:
    custom_domain:
      -
        domain: http://server.ai-cv-image-prediction.1583208943291465.cn-hangzhou.fc.devsapp.net
```

- 此时可以打开系统分配的测试域名，并上传一张图片进行测试：

2.2.1.3 传统框架：基于 Django 的博客项目

- 初始化一个已有的基于 Django 的博客项目：s init django-blog，初始化过程中可能会出现填写项目名称以及选择密钥的过程：
  - 项目名称可以是：django-blog
  - 密钥可以选择上文中创建过的：alibaba-access

例如：

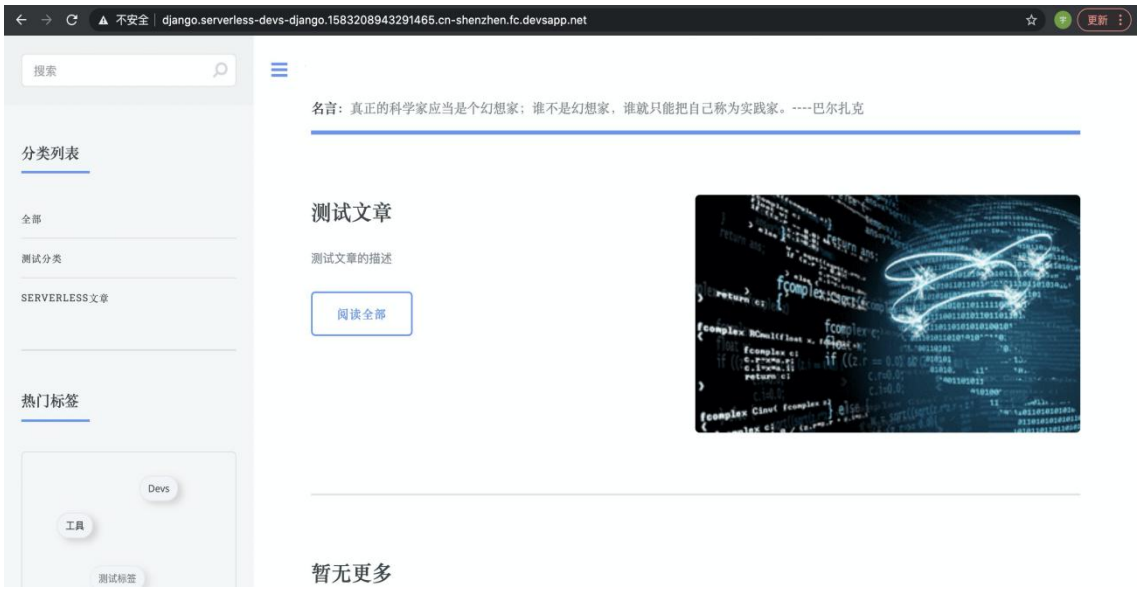
```
$ s init django-blog

    Serverless Awesome: https://github.com/Serverless-Devs/package-awesome

? Please input your project name (init dir) django-blog
✔ file decompression completed
? please select credential alias alibaba-access

      Welcome to the django-blog application
       This application requires to open these services:
            FC : https://fc.console.aliyun.com/
       This application can help you quickly deploy the django-blog project.
       The application uses Django component：https://github.com/devsapp/django
       The application homepage: https://github.com/devsapp/django-blog

       * Python 3.7 is recommended;
       * If the version is greater than Python 3.7:
           * Operation error: ImportError: cannot import name 'metadata' from 'importlib', you can refer to: https://stackoverflow.com/questions/59216175/importerror-cannot-import-name-metadata-from-importlib
       * Default information:
           * Admin：/admin
```

```
        * Default Admin Username: blog
        * Default Admin Password: myblog12345!

    Thanks for using Serverless-Devs
    You could [cd /Users/jiangyu/django-blog] and enjoy your serverless journey!
    If you need help for this example, you can use [s -h] after you enter folder.
    Document ❤ Star：https://github.com/Serverless-Devs/Serverless-Devs
```

- 进入项目目录：cd django-blog
- 通过 deploy 命令进行项目的部署：

```
Tips for next step
======================
* Invoke remote function: s invoke
✔ Try container acceleration
djangoBlog:
  region:             cn-shenzhen
  serviceName:     serverless-devs-django
  functionName:    django
  customDomains:
    - http://django.serverless-devs-django.1583208943291465.cn-shenzhen.fc.devsapp.net
```

- 此时可以打开系统分配的测试域名，并上传一张图片进行测试：



2.2.2 功能上手体验

2.2.2.1 测试项目创建

通过 s init 命令创建一个 Python 语言的 Hello World 项目，在引导的过程中，可能会出现填写项目名称以及选择密钥的过程：

- 项目名称可以是：start-fc-http-python3
- 密钥可以选择上文中创建过的：alibaba-access ，例如：

```
$ s init devsapp/start-fc-http-python3

?      Serverless Awesome: https://github.com/Serverless-Devs/package-awesome

? Please input your project name (init dir) start-fc-http-python3
✔ file decompression completed
? please select credential alias alibaba-access

...
```

```
    Thanks for using Serverless-Devs
    You could [cd /Users/jiangyu/demo/test/start-fc-http-python3] and enjoy your serve
rless journey!
    If you need help for this example, you can use [s -h] after you enter folder.
    Document ❤ Star：https://github.com/Serverless-Devs/Serverless-Devs

? 是否立即部署该项目？ (Y/n)
```

此时，可以先不进行项目的部署，即选择 n 即可。接下来，可以通过 cd 等命令进入项目（例如：cd start-fc-http-python3）。

2.2.2.2 部署 deploy

为了便于后续的体验，可以对默认的 s.yaml 文件进行修改，增加自动化日志配置的能力：logConfig: auto，完整的项目 Yaml 如下：

```
edition: 1.0.0              #  命令行 YAML 规范版本，遵循语义化版本（Semantic Versioning）
规范
name: fcDeployApp           #  项目名称
access: "default"           #  秘钥别名

services:
  fc-deploy-test:           #  服务名称
    component: devsapp/fc   #  组件名称
    props:                  #  组件的属性值
      region: cn-hangzhou
      service:
        name: fc-deploy-service
        description: 'demo for fc-deploy component'
        logConfig: auto
```

```
    function:
      name: http-trigger-py36
      description: this is a test
      runtime: python3
      codeUri: ./code
      handler: index.handler
      memorySize: 128
      timeout: 60
    triggers:
      - name: httpTrigger
        type: http
        config:
          authType: anonymous
          methods:
            - GET
    customDomains:
      - domainName: auto
        protocol: HTTP
        routeConfigs:
          - path: /*
            methods:
              - GET
```

保存并退出编辑之后，可以执行 s deploy 直接进行项目的部署，稍等片刻，即可看到部署结果：

```
fc-deploy-test:
  region:    cn-hangzhou
  service:
    name: fc-deploy-service
  function:
    name:          http-trigger-py36
```

```
    runtime:      python3
    handler:      index.handler
    memorySize:  128
    timeout:      60
  url:
    system_url:      https://1583208943291465.cn-hangzhou.fc.aliyuncs.com/2016-08-15/proxy/fc-deploy-service/http-trigger-py36/
    custom_domain:
      -
        domain:  http://http-trigger-py36.fc-deploy-service.1583208943291465.cn-hangzhou.fc.devsapp.net
  triggers:
    -
      type:  http
      name:  httpTrigger
```

2.2.2.3 调用相关

本地调用

由于该项目是一个 HTTP 函数，所以可以使用 s local start 进行本地调用的测试（如果是其他 Event 函数，可以考虑用 s local invoke）。

```
url:  http://localhost:7665/2016-08-15/proxy/fc-deploy-service/http-trigger-py36/
    methods:  GET
    authType:  anonymous

    Tips for more action:
        Start with customDomain method: [s local start auto]
        Debug with customDomain method: [s local start -d 3000 auto]
```

此时，可以根据系统返回的 url 参数，在浏览器中打开 http://localhost:7665/2016-08-15/proxy/fc-deploy-service/http-trigger-py36/，进行 HTTP 函数的本地测试。

远程调用

在当前项目下，直接使用 s invoke 即可实现线上函数的调用/触发：

```
Request url: https://1583208943291465.cn-hangzhou.fc.aliyuncs.com/2016-08-15/proxy/fc-deploy-service/http-trigger-py36/

========= FC invoke Logs begin =========
FC Invoke Start RequestId: eb9cf022-297e-4a27-b3bf-ad304f6e04c9
FC Invoke End RequestId: eb9cf022-297e-4a27-b3bf-ad304f6e04c9

Duration: 2.64 ms, Billed Duration: 3 ms, Memory Size: 128 MB, Max Memory Used: 10.77 MB
========= FC invoke Logs end =========

FC Invoke Result[code: ${resp.code}]:
Hello world!
```

2.2.2.4 可观测性

2.2.2.4.1 日志查看

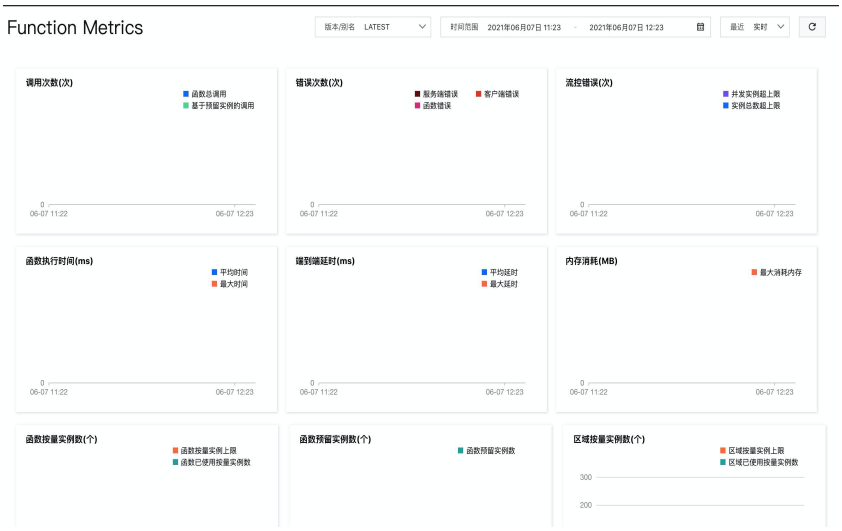
在当前项目下，直接使用 s logs 命令，可以进行日志查看，也可以通过 s logs -t 进入到 tail 模式：

```
FunctionCompute python3 runtime inited.



FC Invoke Start RequestId: eb9cf022-297e-4a27-b3bf-ad304f6e04c9
FC Invoke End RequestId: eb9cf022-297e-4a27-b3bf-ad304f6e04c9
```

2.2.2.4.2 指标查看

在当前项目，直接执行 s metrics 命令，可以进行指标的查看：

```
[2021-06-07T12:20:06.661] [INFO ] [FC-METRICS] - 请用浏览器访问 Uri 地址进行查看: http://localhost:3000
```



2.2.2.5 其他功能

阿里云函数计算组件，除了上面所快速体验的功能之外，还有以下诸多功能：

- 构建与部署相关：
  - 部署 deploy ： https://github.com/devsapp/fc/blob/main/docs/zh/command/deploy.md

- 构建 build：https://github.com/devsapp/fc/blob/main/docs/zh/command/build.md
- 移除 remove：https://github.com/devsapp/fc/blob/main/docs/zh/command/remove.md
- 计划变更 plan：https://github.com/devsapp/fc/blob/main/docs/zh/command/plan.md

- 可观测性相关：
  - 指标查询 metrics：https://github.com/devsapp/fc/blob/main/docs/zh/command/metrics.md
  - 日志查询 logs：https://github.com/devsapp/fc/blob/main/docs/zh/command/logs.md

- 调用与调试相关：
  - 本地调用 local：https://github.com/devsapp/fc/blob/main/docs/zh/command/local.md
  - 函数触发 invoke：https://github.com/devsapp/fc/blob/main/docs/zh/command/invoke.md
  - 端云联调 proxied：https://github.com/devsapp/fc/blob/main/docs/zh/command/proxied.md
  - 远程调试 instance：https://github.com/devsapp/fc/blob/main/docs/zh/command/instance.md
  - 内存&并发度探测 eval：https://github.com/devsapp/fc/blob/main/docs/zh/command/eval.md

- 发布与配置相关：
  - 版本 version：https://github.com/devsapp/fc/blob/main/docs/zh/command/version.md
  - 别名 alias：https://github.com/devsapp/fc/blob/main/docs/zh/command/alias.md
  - 预留 provision：https://github.com/devsapp/fc/blob/main/docs/zh/command/provision.md
  - 按量资源 ondemand：https://github.com/devsapp/fc/blob/main/docs/zh/command/ondemands.md
  - 层 layer：https://github.com/devsapp/fc/blob/main/docs/zh/command/layer.md

- 其他功能：
  - 硬盘挂载 nas：https://github.com/devsapp/fc/blob/main/docs/zh/command/nas.md
  - Fun 项目迁移 fun2s：https://github.com/devsapp/fc/blob/main/docs/zh/command/fun2s.md
  - 查看函数 info：https://github.com/devsapp/fc/blob/main/docs/zh/command/info.md
  - 资源同步 sync：https://github.com/devsapp/fc/blob/main/docs/zh/command/sync.md
  - 压测 stress：https://github.com/devsapp/fc/blob/main/docs/zh/command/stress.md
  - API 调用 api：https://github.com/devsapp/fc/blob/main/docs/zh/command/api.

2.3 让发布更安全的：线上异动感知

使用 Serverless 工具时，很多人都会关注：如何保证 Serverless 业务部署更新的一致性。

所谓的一致性在这里指的是：通过工具在本地进行项目部署，此时再有其他人通过其他途径（例如控制台等），对项目进行过更新等操作，此时再在本地进行项目部署，是不是会直接覆盖？

例如，当用户 A 在本地更新了业务，因为一些特殊情况，导致出现了一个线上异常 x，此时用户 B 重新更新，将这个内容修复了，但是 B 没有及时同步给 A 这个事情，A 又更新了新的功能，直接覆盖了 B 的内容，这个时候之前的异常 x 又出现了，如果此时在 A 更新的时候，可以感知到线上资源已经变动，那么这种事情就不会再次发生。

目前基于 Serverless Devs 的阿里云函数计算组件，已经支持了线上“异动”的感知能力，包括了以下几个情况：

线上服务	线上函数	本地记录的服务	本地记录的函数	本地要部署服务	本地要部署函数	直接部署	use-local	use-remote
存在	存在	与线上一致	与线上一致	-	-	无交互	线上服务与函数将会被更新	线上服务与函数不做任何操作
存在	不存在	与线上一致	-	-	-	无交互	线上服务与函数将会被更新	线上服务不做任何操作，函数将会被创建
不存在	不存在	-	-	-	-	无交互	线上服务与函数将会被更新	线上服务与函数将会被创建

线上服务	线上函数	本地记录的服务	本地记录的函数	本地要部署服务	本地要部署函数	直接部署	use-local	use-remote
存在	存在	与线上不一致	与线上不一致	与线上一致	与线上一致	无交互	线上服务与函数将会被更新	线上服务与函数不做任何操作
存在	存在	与线上不一致	与线上不一致	与线上不一致	与线上不一致	有交互	线上服务与函数将会被更新	线上服务与函数不做任何操作

2.3.1 部署前检测

当通过 Serverless Devs 开发者工具进行项目部署完成：

```
framework:
  region:   cn-beijing
  service:
    name: web-framework
  function:
    name:       bottle
    runtime:    python3
    handler:    index.app
    memorySize: 128
    timeout:    60
  url:
    system_url:    https://1583208943291465.cn-beijing.fc.aliyuncs.com/2016-08-15/proxy/web-framework/bottle/
    custom_domain:
      -
        domain: http://bottle.web-framework.1583208943291465.cn-beijing.fc.devsapp.net
  triggers:
    -
      type: http
      name: httpTrigger
```

再模拟其他用户在控制台对配置信息进行更新：

完成之后，通过 plan 功能可以进行部署前的资源检测：

```
[2021-12-27 10:49:44] [INFO] [S-CLI] - Start ...


region: cn-beijing
service:
  description: Serverless Devs Web Framework Service
  name: web-framework
function:
  description: Serverless Devs Web Framework Bottle Function
  runtime: python3
  handler: index.app
  ~ timeout: 40 => 60
  memorySize: 128
  instanceConcurrency: 1
  instanceType: e1
  name: bottle
triggers:
  name: httpTrigger
  type: http
  config:
    methods:
      - GET
    authType: anonymous
customDomains:
  domainName: bottle.web-framework.1583208943291465.cn-beijing.fc.devsapp.net
  protocol: HTTP
  routeConfig:
      path: /*
      serviceName: web-framework
      functionName: bottle
```

可以看到，系统已经成功的感知到了变更。

2.3.2 部署异动感知

继续上面的例子，如果在部署的时候直接执行 s deploy，系统同样会在部署的时候，告知代码内容发生了变化：

```
function:
  description: Serverless Devs Web Framework Bottle Function
  runtime: python3
  handler: index.app
  ~ timeout: 40 => 60
  memorySize: 128
  instanceConcurrency: 1
  instanceType: e1
```

并且让使用者主动选择 use-remote 或者 use-local，进行继续部署。

2.3.3 在自动化流程中的策略

代码在其他场景被更新，需要在当前得到感知，这个事情其实是非常重要的，和代码的安全发布密不可少。而此时，通过 Serverless Devs 是可以做到的。

那么问题来了，如果已经有了一个项目，想集成到 cd 流程，不想出现交互式操作，应该如何处理呢？此时提供一个--use-local/--use-remote 参数，用来强行指定优先使用的配置，通过这样的指令就可以实现无交互的部署了。

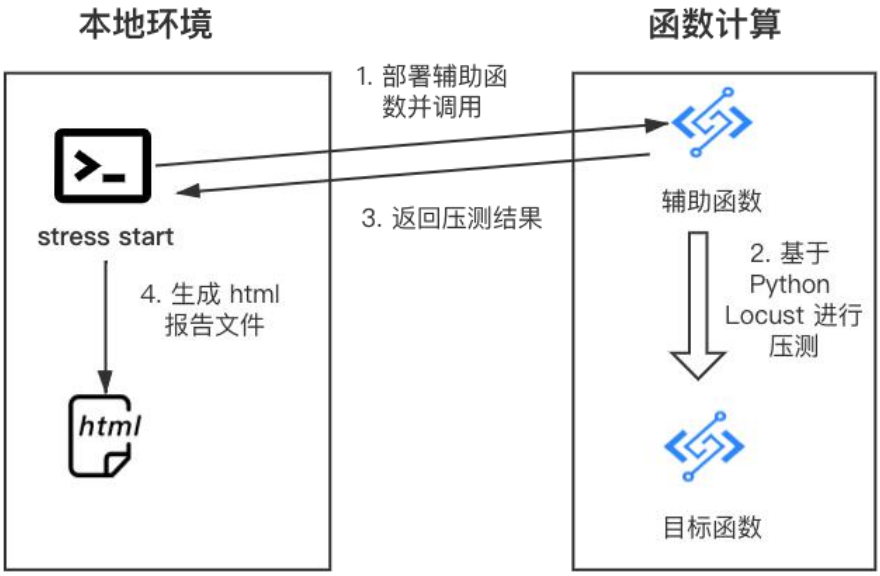
例如：

- 优先使用本地配置：s deploy --use-local
- 优先使用线上配置：s deploy --use-remote

2.4 性能心中有数：一键压测函数计算

2.4.1 实现原理

stress 命令的原理是通过创建辅助函数，对目标函数进行压测，架构简图如下所示：



- stress start 指令会根据组件内置配置，创建辅助函数（辅助函数的服务名和函数名均为：_DEFAULT_FC_STRESS_COMPONENT_SERVICE 和）；
- 辅助函数创建完成后，将会调用目标函数，压测参数放置在调用负载中；辅助函数被调用后就会基于 Python Locust 对目标函数发起压测试；
- 完成测试之后，将压测结果返回给本地；
- 本地收到结果后，会展示结果并并生成 html 报告文件；

2.4.2 操作案例

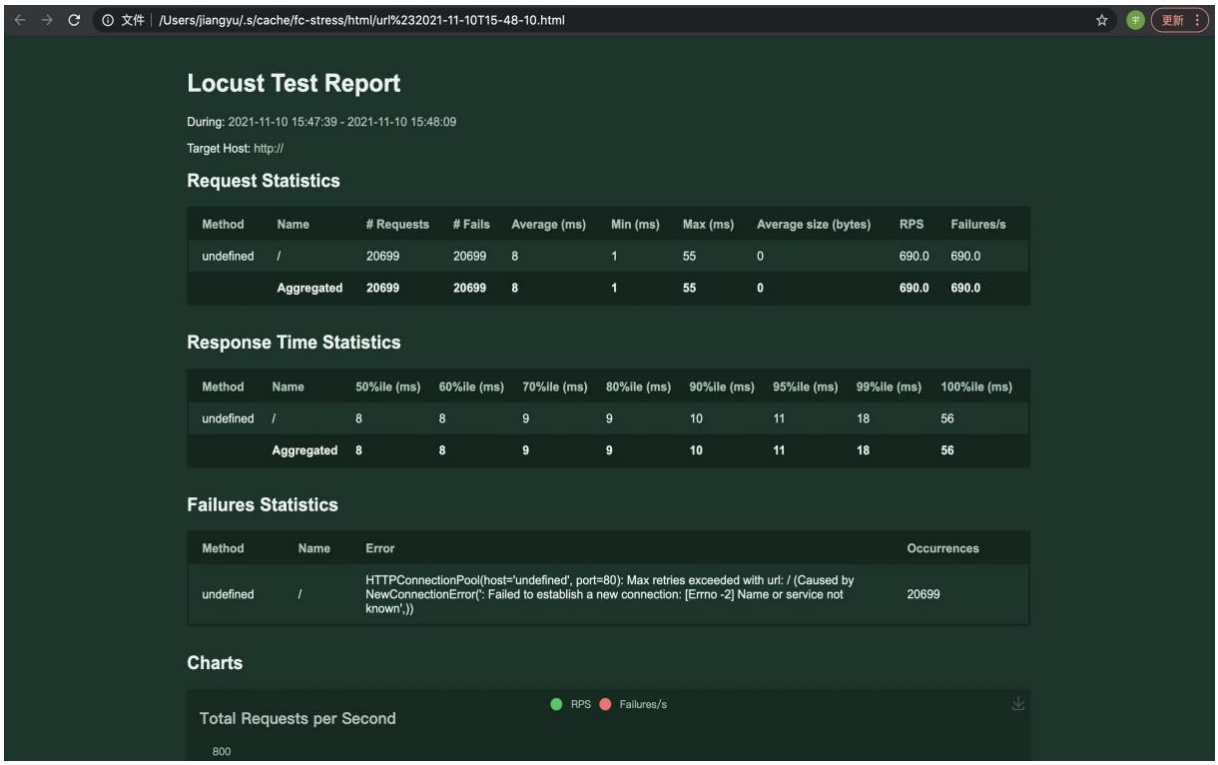
- **有资源描述文件（Yaml）时**，可以直接执行 s stress start 开始目标函数压测；
- **纯命令行形式（在没有资源描述 Yaml 文件时）**，需要指定服务所在地区以及服务名称，函数名等，例如：

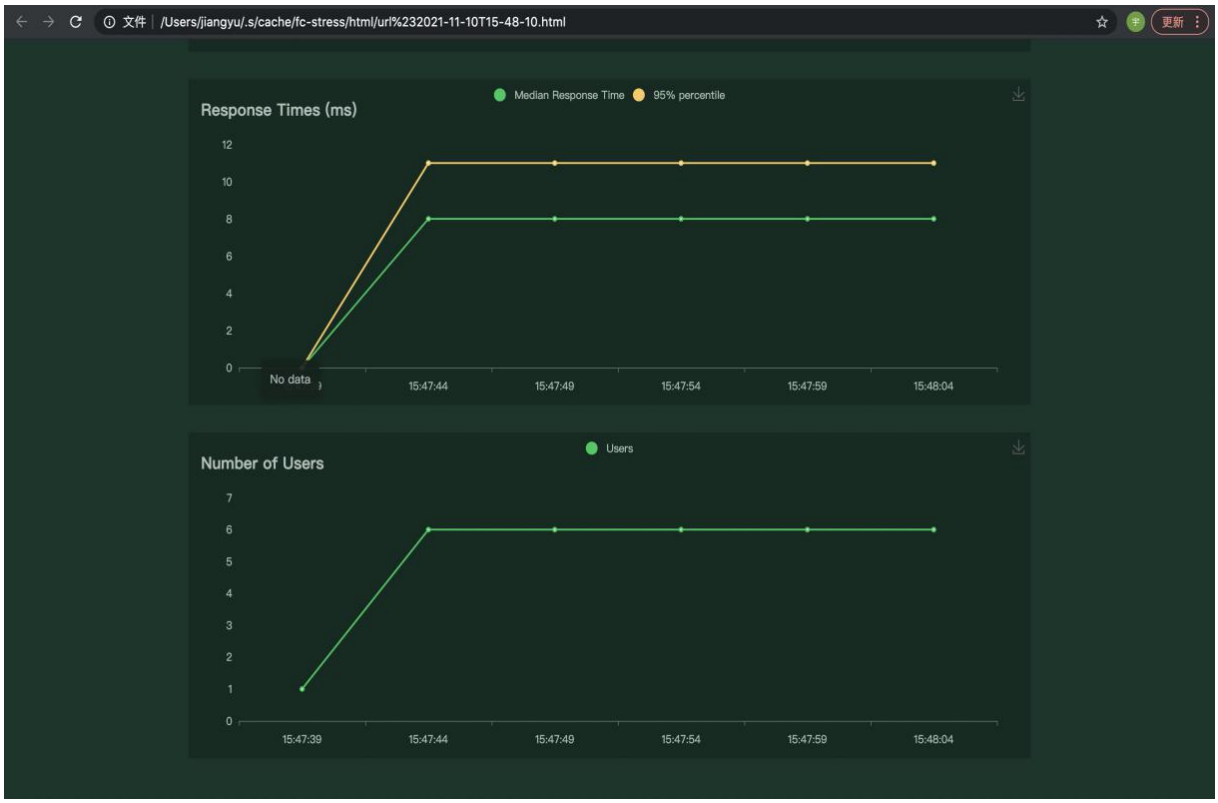
```
$ s cli fc stress start --region cn-hangzhou --access myAccess --service-name fc-deploy-service --function-name http-trigger-py36 --function-type event
```

上述命令的执行结果示例：

```
Html report flie: /Users/jiangyu/.s/cache/fc-stress/html/url#2021-11-10T15-48-10.html
Execute 'open /Users/jiangyu/.s/cache/fc-stress/html/url#2021-11-10T15-48-10.html' on macos for html report with browser.
fc-deploy-test:
  Average:        8
  Error:          HTTPConnectionPool(host='undefined', port=80): Max retries exceeded with url: / (Caused by NewConnectionError(': Failed to establish a new connection: [Errno -2] Name or service not known',))
  Fails:          20699
  Failures/s:     690
  Max:            55
  Method:         undefined
  Min:            1
  Name:           /
  Occurrences:    20699
  RPS:            690
  Requests:       20699
  p50:            8
  p60:            8
  p70:            9
  p90:            10
  p95:            11
  p99:            18
```

根据返回信息（例如：` Execute 'open /Users/jiangyu/.s/cache/fc-stress/html/url#2021-11-10T15-48-10.html' on macos for html report with browser.）可打开相对应的压测报告：


文件 | /Users/jiangyu/.s/cache/fc-stress/html/url%232021-11-10T15-48-10.html
更新
Locust Test Report
During: 2021-11-10 15:47:39 - 2021-11-10 15:48:09
Target Host: http://
Request Statistics
Method Name # Requests # Fails Average (ms) Min (ms) Max (ms) Average size (bytes) RPS Failures/s
undefined / 20699 20699 8 1 55 0 690.0 690.0
Aggregated 20699 20699 8 1 55 0 690.0 690.0
Response Time Statistics
Method Name 50%ile (ms) 60%ile (ms) 70%ile (ms) 80%ile (ms) 90%ile (ms) 95%ile (ms) 99%ile (ms) 100%ile (ms)
undefined / 8 8 9 9 10 11 18 56
Aggregated 8 8 9 9 10 11 18 56
Failures Statistics
Method Name Error Occurrences
undefined / HTTPConnectionPool(host='undefined', port=80): Max retries exceeded with url: / (Caused by NewConnectionError(': Failed to establish a new connection: [Errno -2] Name or service not known',)) 20699
Charts
Total Requests per Second
RPS
Failures/s
800


文件 | /Users/jiangyu/.s/cache/fc-stress/html/url%232021-11-10T15-48-10.html
更新
Response Times (ms)
Median Response Time
95% percentile
12
10
8
6
4
2
0
No data
15:47:44
15:47:49
15:47:54
15:47:59
15:48:04
Number of Users
Users
7
6
5
4
3
2
1
0
15:47:39
15:47:44
15:47:49
15:47:54
15:47:59
15:48:04

2.5 从工具看函数资源评估

在 Serverless 领域内，通常会出现以下两种使用场景：

- CPU 密集型场景

  对于 CPU 密集型场景，例如音视频处理、AI 推理或图片处理等，您一般会选择使用单实例单并发。由于该类场景的函数内存大小和 CPU 能力成正比，因此您需要根据函数是成本敏感型还是延迟敏感型选择合适的内存规格。

- I/O 密集型场景

  对于 I/O 密集型场景，您一般会选择使用单实例多并发。该类场景下由于函数内存和 CPU 能力成正比，建议您将函数内存规格设置足够大，但可能会出现浪费资源的现象，很难选择合适的单实例并发值。

针对在以上两种使用场景无法设置合适的参数规格的情况，Serverless Devs 为您提供了探测功能，可以实现内存探测和并发度探测，获取满足您需求的参数配置信息。eval 命令是对函数进行探测的命令；通过 eval 指令，可以对函数探测内存(单实例单并发)或者探测并发度(单实例多并发)。例如给 CPU 密集型场景的函数设置合适的内存，给 I/O 密集型场景的函数设置合适的并发值，根据探测结果，获取满足需求的最佳内存大小或最佳并发度值。

注意： 这个命令只是针对开发上线前阶段的函数，不要对生产函数执行探测操作。

2.5.1 内存探测模式

本示例以探测 CPU 密集型场景事件函数为例，介绍如何实现内存模式的探测。请执行以下命令，获取最佳的内存规格：

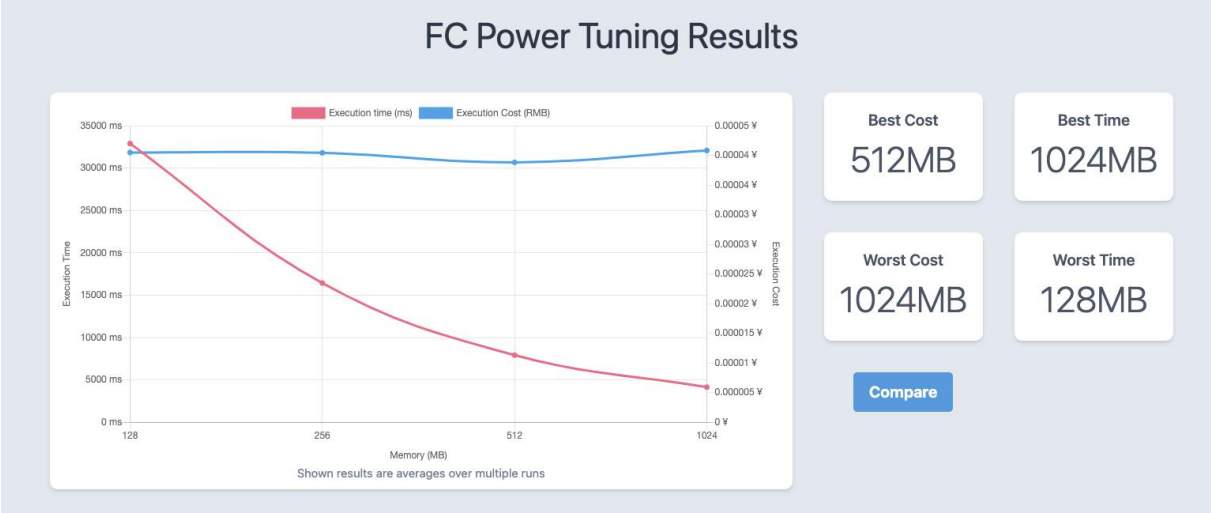
```
s cli fc eval start --region cn-hangzhou --function-name cpu-test --service-name Service --eval-type memory --run-count 10 --payload '{"key":"val"}' --memory-size 128,256,512,1024 --access default
```

本示例表示被探测的目标函数在 128 MB、256 MB、512 MB 和 1024 MB 四个不同的内存模式下分别被调用十次后取平均值，例如目标函数在 128 MB 内存格式下被调用十次后取平均值，最终对目标函数在这四个不同内存模式下分别被调用十次后的平均值进行对比，获取探测信息。

输出示例：

```
...
>-
http://memory-tuning.devsapp.cn/#gAAAAQACAAQ=;AIAARwBYgEYAoPdFAHiBRQ==;37w+OH+BPjiqxzc4/SxAOA==
```

此时，可以通过浏览器打开返回的地址，查看相关探测信息：



关于该结果的解析如下：

- 图中红色的线表示执行时间与内存大小的关系；蓝色的线表示所消耗的成本与内存之间的关系；
- 右侧四个数字分别是：
  - Best Cost：消费最少的时候，内存是 512MB；
  - Best Time：执行耗时最短的时候，内存是 1024MB；

- Worst Cost：消费最多的时候，内存是 1024MB；
- Worst Time：执行耗时最长的时候，内存是 128MB；

此时可以根据这个数据，当前函数资源进行内存设定，可以综合曲线和实际业务需求，例如，在当前时刻，延迟要求在 20000ms 以内，256M 是成本最佳， 如果要求延迟在 5000ms 以内， 这个时候 1024M 是最佳选择。

2.5.2 并发度探测模式

本示例以 I/O 密集型场景的 HTTP 函数为例，介绍如何对函数实现并发度的探测。请执行以下命令，获取最佳的并发度规格：

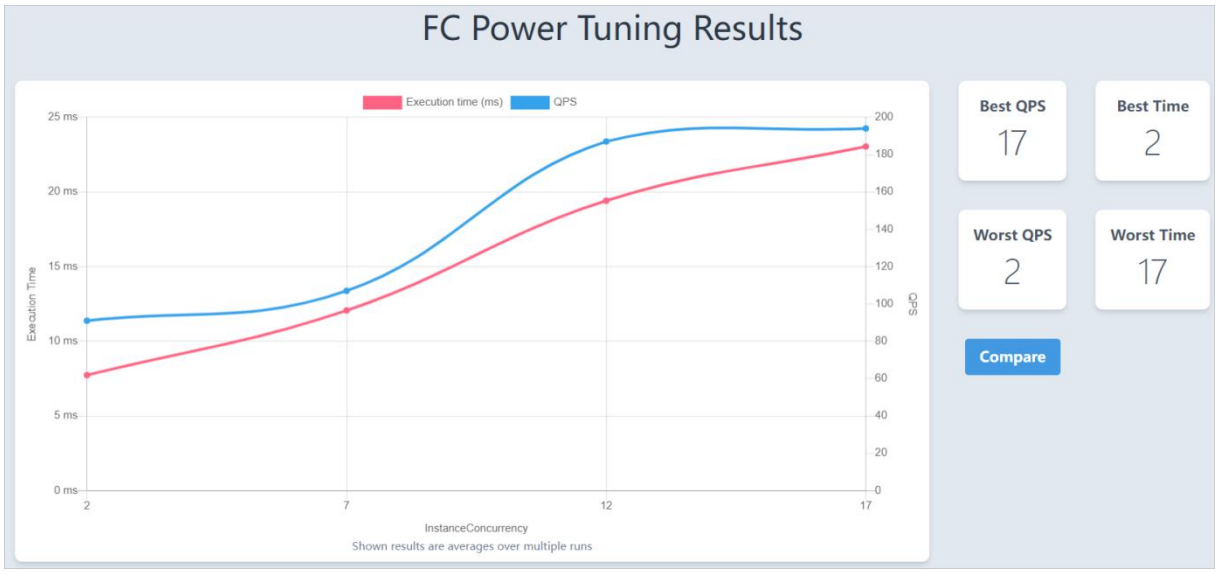
```
s cli fc eval start --region cn-hangzhou --function-name demo --service-name Service --eval-type concurrency --memory 1536 --concurrency-args 2,20,5 --rt 200 --method=get --path /login  --query 'a=1&b=2'  --headers='{"header_a":"v"}' --access default
```

本示例表示将被探测的目标函数的内存规格设置为 1.5 GB，并发度范围[2,20]，步长为 5，即被探测的目标函数分别在 2，7，12 和 17 不同并发值模式下实现探测，获取探测信息。

输出示例：

```
...
>-
http://concurrency-tuning.devsapp.cn/#AgAHAAwAEQA=;Ilj3QCL+QEFCR5tBGFO4QQ==;AAC2QgAA1kIAADtDAABCQw==
```

此时，可以通过浏览器打开返回的地址，查看相关探测信息：



关于该结果的解析如下：

- 图中红色的线表示执行时间与单实例并发度的关系；蓝色的线表示有效的 QPS 与单实例并发度之间的关系；
- 右侧四个数字分别是：
  - Best QPS：有效 QPS 最高的时候，单实例并发度是 17；
  - Best Time：执行耗时最短的时候，单实例并发度是 2；
  - Worst QPS：有效 QPS 最低的时候，单实例并发度是 2；
  - Worst Time：执行耗时最短的时候，单实例并发度是 17；

此时可以根据这个数据，可以对这个函数的单实例多并发的并发度值进行设置， 如果需要函数的每次执行时间都小于 20ms，那么设置并发度 12 是一个好的选择。

2.6 Serverless Devs 与 Funcraft/Fcli 的对比

Serverless Devs 与 Funcraft/Fcli 等工具无论是在使用形式上还是在功能对比上，以及应用场景上都有有比较明显的区别。

2.6.1 形式对比

	fc 组件	funcraft	fcli
依赖资源描述文件（Yaml）	✓	✓	
纯命令行模式（不依赖 Yaml）	✓		✓

2.6.2 功能对比

	fc 组件	funcraft	fcli
应用部署	✓	✓	✓（需要执行多条命令）
应用移除	✓		✓（需要执行多条命令）
构建	✓	✓	
远程调用	✓	✓	
本地调用	✓	✓	
查看日志	✓		
查看指标	✓		
NAS 操作	✓	✓	
同步操作	✓		
版本	✓		

	fc 组件	funcraft	fcli
别名	✓		
预留	✓		
按量资源	✓		
层	✓		
端云联调	✓		
一键压测	✓		
内存和并发度探测	✓		
远程调试	✓		
函数异动感知	✓		
端到端部署	✓		
多账号管理	✓		
API 操作	✓		

2.6.3 场景对比

	fc 组件	funcraft	fcli
用户可能同时有测试账号和线上账号，或者个人账号和公司账号，需要进行进行不同账号的切换。	✓		
用户需要在一个项目的执行前后，进行其他相关的行为定义，例如部署前需要进行 build，部署后需要进行版本的发布，相关文件的上传，灰度的设置等。	✓		
用户需要一键部署端到端的项目，例如将前端代码上传到对象存储，后端代码上传到函数计算，同时部署 API 网关、CDN 等相关业务。	✓		

	fc 组件	funcraft	fcli
用户需要在本地进行调试，但是有一些网络环境时线上的 VPC，此时需要在本地连接到线上的 VPC 环境，进行代码的调试等。	✓		
在进行项目部署时，Yaml 需要从环境变量获取一些敏感信息，或者从其他的文件获取信息，也或者从已经部署完成的项目获得返回值作为入参，进行项目的部署。	✓		
不依赖 Yaml 进行相关的原子性的操作，例如查看函数列表，服务列表，删除某个函数、服务，查看版本列表等；	✓		✓

2.6.4 迁移案例

2.6.4.1 从 Funcraft 迁移到 Serverless Devs 的方法

【推荐】Yaml 格式切换：这种方法是将 Funcarft 规范的资源描述文档（例如 template.yaml 文件），转换成符合 Serverless Devs 规范且使用 FC 组件的资源描述文档（例如 s.yaml 文件）。可以在 Funcraft 项目目录下，通过 fun2s 命令，实现 Yaml 规范转换，例如：

```
s cli fc fun2s --region cn-shenzhen --target ./s.yml

[2021-11-02T19:51:25.418] [INFO ] [FC-TRANSFORM] - Using funcraft yaml: /Users/jiangyu/demo/test/start-fc-http-python3/template.yml
[2021-11-02T19:51:25.429] [INFO ] [FC-TRANSFORM] - Reminder serverless devs yaml path: /Users/jiangyu/demo/test/start-fc-http-python3/s.yml

Tips for next step
======================
* Deploy Function: s deploy -t ./s.yml
* Invoke Event Function: s local invoke -t ./s.yml
* Invoke HTTP Function: s local start -t ./s.yml
```

此时，就可以将原有的 Funcraft 规范的 template.yaml 转换成支持 Serverless Devs 规范的 s.yaml。

转换前（template.yaml）：

```
ROSTemplateFormatVersion: '2015-09-01'
Transform: 'Aliyun::Serverless-2018-04-03'
Resources:
  start-fc-http-python3:
    Type: 'Aliyun::Serverless::Service'
    Properties:
      Description: 'helloworld'
    start-fc-http-python3:
      Type: 'Aliyun::Serverless::Function'
      Properties:
        Handler: index.handler
        Runtime: nodejs10
        CodeUri: './'
```

转换后（s.yaml）：

```
edition: 1.0.0
name: transform_fun
access: default
vars:
  region: cn-shenzhen
services:
  fc-start-fc-http-python3-start-fc-http-python3:
    component: devsapp/fc
    props:
      region: ${vars.region}
```

```
    service:
      name: start-fc-http-python3
      description: helloworld
      internetAccess: true
    function:
      name: start-fc-http-python3
      handler: index.handler
      runtime: nodejs10
      codeUri: ./
```

资源信息重新同步：这种方法是将线上的函数资源，直接同步到本地，包括线上函数的代码和相关的配置（此时的配置是符合 Serverless Devs 规范且使用 FC 组件的资源描述文档，例如 s.yaml）。指定好对应的服务名、函数名等，例如：

```
s cli fc sync --region cn-hanghzou --service-name fc-deploy-service --function-name http-trigger-py36;
```

上述命令的执行结果示例：

```
fc-deploy-test:
  codeFiles:
    http-trigger-py36: /Users/jiangyu/demo/test/start-fc-http-python3/1583208943291465_cn-hangzhou_fc-deploy_service_http-trigger-py36
  configYmlPath: /Users/jiangyu/demo/test/start-fc-http-python3/s.cn-hangzhou-fc-deploy-service.sync.yaml
```

2.6.4.2 从 Fcli 迁移到 Serverless Devs 的方法

从 Fcli 迁移到 Serverless Devs 的项目，通常是进行函数管理或者是与自动化脚本集成的需求，此时可以考虑使用 API 直接进行迁移，FC 组件的 API 能力是直接操作函数计算 API 的功能。

3. Serverless 架构下 AI 最佳实践

3.1 PaddleOCR 项目开发与部署

PaddlePaddle 团队首次开源文字识别模型套件 PaddleOCR，目标是打造丰富、领先、实用的文本识别模型/工具库。该模型套件是一个实用的超轻量 OCR 系统。主要由 DB 文本检测、检测框矫正和 CRNN 文本识别三部分组成。该系统从骨干网络选择和调整、预测头部的设计、数据增强、学习率变换策略、正则化参数选择、预训练模型使用以及模型自动裁剪量化 8 个方面，采用 19 个有效策略，对各个模块的模型进行效果调优和瘦身，最终得到整体大小为 3.5M 的超轻量中英文 OCR 和 2.8M 的英文数字 OCR。

3.1.1 本地开发

根据 PaddleOCR 的项目案例，采用轻量级 Python Web 框架 Bottle 框架进行项目开发：

```
# index.py
import base64
import bottle
import random
from paddleocr import PaddleOCR

ocr = PaddleOCR(use_gpu=False)

@bottle.route('/ocr', method='POST')
def login():
    filePath = './temp/' + (''.join(random.sample('zyxwvutsrqponmlkjihgfedcba', 5)))
    with open(filePath, 'wb') as f:
        f.write(base64.b64decode(bottle.request.body.read().decode("utf-8").split(',')[1]))
```

```
    ocrResult = ocr.ocr(filePath, cls=False)
    return {'result': [[line[1][0], float(line[1][1])] for line in ocrResult]}

bottle.run(host='0.0.0.0', port=8080)
```

开发完成之后，运行该项目：

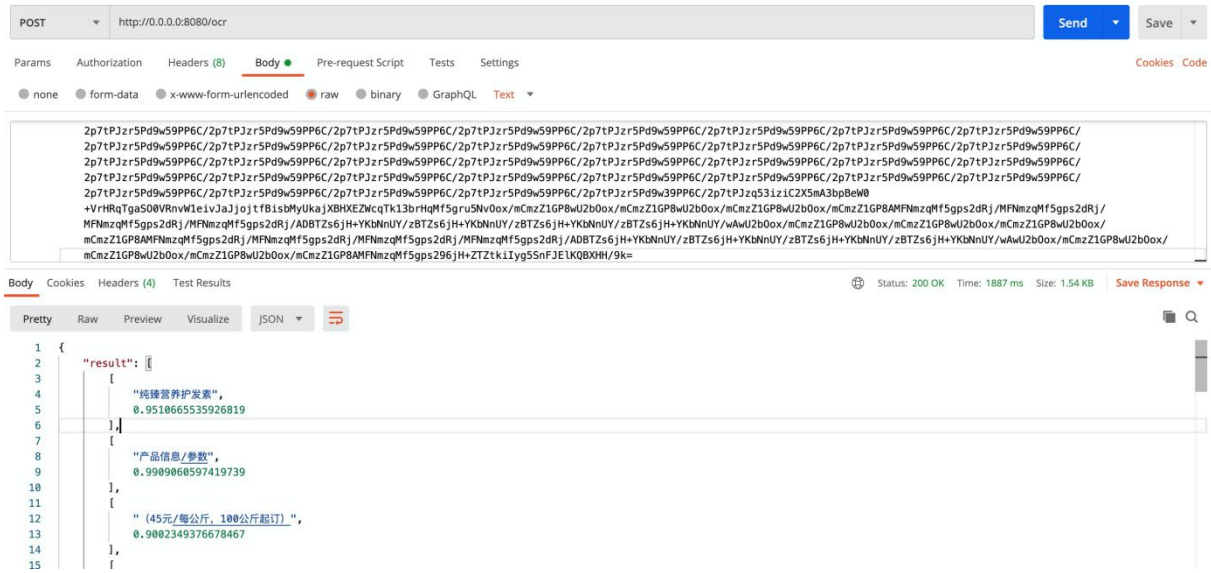
```
python index.py
```

可以看到服务已经启动：

```
--- Running IR pass [runtime_context_cache_pass]
--- Running analysis [ir_params_sync_among_devices_pass]
--- Running analysis [adjust_cudnn_workspace_size_pass]
--- Running analysis [inference_op_replace_pass]
--- Running analysis [memory_optimize_pass]
I0909 13:16:37.899991 152645056 memory_optimize_pass.cc:199] Cluster name : fill_constant_batch_size_like_1.tmp_0  size: 1024
I0909 13:16:37.900034 152645056 memory_optimize_pass.cc:199] Cluster name : lstm_1.tmp_3  size: 1
I0909 13:16:37.900044 152645056 memory_optimize_pass.cc:199] Cluster name : lstm_1.tmp_1  size: 1024
I0909 13:16:37.900051 152645056 memory_optimize_pass.cc:199] Cluster name : x  size: 38400
I0909 13:16:37.900058 152645056 memory_optimize_pass.cc:199] Cluster name : lstm_1._generated_var_0  size: 1
I0909 13:16:37.900063 152645056 memory_optimize_pass.cc:199] Cluster name : hardsigmoid_1.tmp_0  size: 2048
I0909 13:16:37.900070 152645056 memory_optimize_pass.cc:199] Cluster name : depthwise_conv2d_12.tmp_0  size: 204800
I0909 13:16:37.900076 152645056 memory_optimize_pass.cc:199] Cluster name : student_ctc_head_2.tmp_1  size: 662500
I0909 13:16:37.900082 152645056 memory_optimize_pass.cc:199] Cluster name : softmax_0.tmp_0  size: 662500
--- Running analysis [ir_graph_to_program_pass]
I0909 13:16:37.955715 152645056 analysis_predictor.cc:636] ======= optimize end =======
I0909 13:16:37.955866 152645056 naive_executor.cc:98] ---  skip [feed], feed -> x
I0909 13:16:37.959389 152645056 naive_executor.cc:98] ---  skip [student_ctc_head_2.tmp_1], fetch -> fetch
Bottle v0.12.19 server starting up (using WSGIRefServer())...
Listening on http://0.0.0.0:8080/
Hit Ctrl-C to quit.
```

然后通过 Postman 工具进行测试，首先准备一张图片（此处以 PaddleOCR 项目内置的测试图片为例）：

通过将图片转换为 Base64 编码，并以 POST 方法请求刚刚启动的 Web 服务，可以看到 PaddleOCR 的执行结果：



3.1.2 部署到 Serverless 架构

3.1.2.1 部署前准备

首先需要完成 Dockerfile 文件：

```
FROM python:3.7-slim
RUN apt update && apt install gcc libglib2.0-dev libgl1-mesa-glx libsm6 libxrender1 -y
 && pip install paddlepaddle bottle scikit-build paddleocr
# Create app directory
WORKDIR /usr/src/app
# Bundle app source
COPY . .
```

编写符合 Serverless Devs 规范的 Yaml 文档：

```
edition: 1.0.0
name: paddle-ocr
access: default

services:
  paddle-ocr:
    component: fc
    props:
      region: cn-shanghai
      service:
        name: paddle-ocr
        description: paddle-ocr service
      function:
        name: paddle-ocr-function
```

```
    runtime: custom-container
    caPort: 8080
    codeUri: ./
    timeout: 60
    customContainerConfig:
      image: 'registry.cn-shanghai.aliyuncs.com/custom-container/paddle-ocr:0.0.1'
      command: '["python"]'
      args: '["index.py"]'
  triggers:
    - name: httpTrigger
      type: http
      config:
        authType: anonymous
        methods:
          - GET
          - POST
  customDomains:
    - domainName: auto
      protocol: HTTP
      routeConfigs:
        - path: /*
```

3.1.2.2 项目部署

首先构建镜像，此处可以通过 Serverless Devs 进行构建：

```
s build --use-docker
```

```
Sending build context to Docker daemon  5.632kB
Step 1/4 : FROM python:3.7-slim
 ---> 959d1a6868be
Step 2/4 : RUN apt update && apt install gcc -y && pip install paddlepaddle bottle scikit-build paddleocr
 ---> Using cache
 ---> b9e5ebab6ee1
Step 3/4 : WORKDIR /usr/src/app
 ---> Using cache
 ---> 1d2f6ebed41a
Step 4/4 : COPY . .
 ---> bd8f1acd9784
Successfully built bd8f1acd9784
Successfully tagged registry.cn-shanghai.aliyuncs.com/custom-container/paddle-ocr:0.0.1
Build image(registry.cn-shanghai.aliyuncs.com/custom-container/paddle-ocr:0.0.1) successfully
[2021-09-09T13:28:07.339] [INFO ] [FC-BUILD] - Build artifact successfully.

Tips for next step
======================
* Invoke Event Function: s local invoke
* Invoke Http Function: s local start
* Deploy Resources: s deploy
End of method: build
```

构建完成之后，可以通过工具直接进行部署：

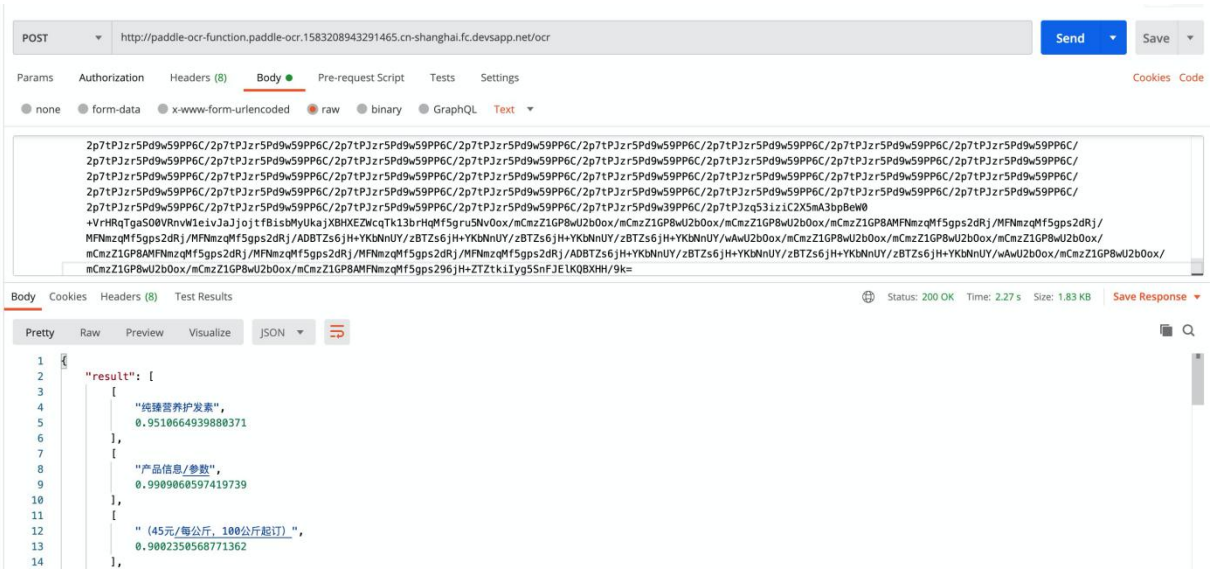
```
s deploy --use-local -y
```

部署完成，可以看到系统返回的测试地址：

```
paddle-ocr:
  region: cn-shanghai
  service:
    name: paddle-ocr
  function:
    name: paddle-ocr-function
    runtime: custom-container
    handler: not-used
    memorySize: 1024
    timeout: 60
  url:
    system_url: >-
      https://1583208943291465.cn-shanghai.fc.aliyuncs.com/2016-08-15/proxy/paddle-ocr/paddle-ocr-function/
    custom_domain:
      - domain: >-
          http://paddle-ocr-function.paddle-ocr.1583208943291465.cn-shanghai.fc.devsapp.net
  triggers:
    - type: http
      name: httpTrigger
```

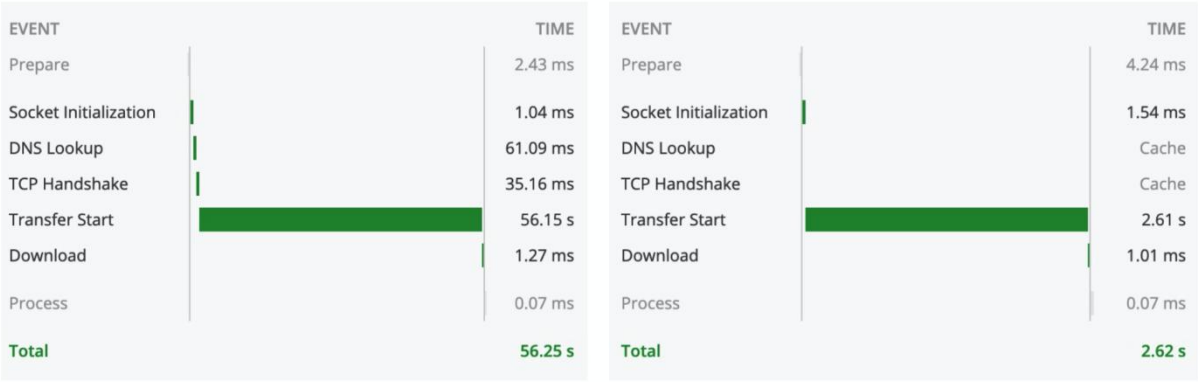
3.1.2.3 项目测试

此时，可以通过该测试地址进行测试，同样得到了预期效果：



3.1.3 项目优化

通过对部署在 Serverless 架构上的项目进行请求，可以看到冷启动和热启动的时间消耗：



冷启动　　热启动

通过冷启动与热启动的对比，可以发现，在热启动时，整个系统的性能是相对优秀的。但是遇到冷启动整个项目的响应时常是不可控的，此时可以考虑一下途径进行优化：

- 缩减容器镜像的体积，减少不必要的依赖、文件等，清理掉安装依赖时留下的缓存等；因为函数计算的冷启动包括镜像拉取时间；
- 部分流程进行优化，例如在 PaddleOCR 项目中有明确说明：“paddleocr 会自动下载 ppocr 轻量级模型作为默认模型”，所以这就意味着该项目在 Serverless 架构的冷启动过程中，相对比热启动还增加了一个模型下载和解压的流程，所以这一部分在必要时是可以打入到容器镜像中，进而减少冷启动带来的影响；
- 开启镜像加速，可以有效降低容器镜像的冷启动，在阿里云函数计算官方文档中有相关镜像加速的性能测试描述：“开启函数计算的镜像加速后，可提速 2~5 倍，将分钟级的镜像拉取缩短至秒级”；
- 实例预留，最大程度上降低冷启动率。通过实例预留，可以通过多种算法/策略进行实例的预热和预启动，可以最大程度上降低 Serverless 架构冷启动带来的影响；

3.2 Tensorflow 与目标检测系统

ImageAI 是一个 Python 库，旨在使开发人员能够使用简单的几行代码构建具有包含深度学习和计算机视觉功能的应用程序和系统。ImageAI 本着简洁的原则，支持最先进的机器学习算法，用于图像预测，自定义图像预测，物体检测，视频检测，视频对象跟踪和图像预测训练。该库目前支持使用在 ImageNet-1000 数据集上训练的 4 种不同机器学习算法进行图像预测和训练，除此之外还支持使用在 COCO 数据集上训练的 RetinaNet 进行对象检测，视频检测和对象跟踪等。

ImageAI 项目是一个依赖 Tensorflow 1.4.0（及更高版本）的项目，通过对 Tesorflow 项目的进一步封装，ImageAI 项目提供用于图像预测的 4 种算法包括 **SqueezeNet**，**ResNet**，**InceptionV3** 和 **DenseNet**，可以通过非常简单的方法实现图像预测、目标检测等任务。

3.2.1 本地开发

首先明确要进行目标检测的图像：

然后可以根据 ImageAI 提供的开发文档，实现以下代码：

```
# index.py
from imageai.Prediction import ImagePrediction

# 模型加载
prediction = ImagePrediction()
prediction.setModelTypeAsResNet()
prediction.setModelPath("resnet50_weights_tf_dim_ordering_tf_kernels.h5")
prediction.loadModel()

predictions, probabilities = prediction.predictImage("./picture.jpg", result_count=5 )
for eachPrediction, eachProbability in zip(predictions, probabilities):
    print(str(eachPrediction) + " : " + str(eachProbability))
```

完成代码开发之后，可以通过执行该文件进行效果测试：

```
python index.py
```

可以得到结果：

```
laptop : 71.43893241882324
notebook : 16.265612840652466
modem : 4.899394512176514
hard_disc : 4.007557779550552
mouse : 1.2981942854821682
```

如果在使用过程中觉得模型 resnet50_weights_tf_dim_ordering_tf_kernels.h5 过大，耗时过长，可以按需求选择模型：

- SqueezeNet（文件大小：4.82 MB，预测时间最短，精准度适中）

- ResNet50 by Microsoft Research （文件大小：98 MB，预测时间较快，精准度高）
- InceptionV3 by Google Brain team （文件大小：91.6 MB，预测时间慢，精度更高）
- DenseNet121 by Facebook AI Research （文件大小：31.6 MB，预测时间较慢，精度最高）

3.2.2 部署到 Serverless 架构

3.2.2.1 部署前准备

在本地完成 ImageAI 的基本测试运行之后，可以将项目部署到 Serverless 架构上。在 Serverless 架构上，该项目的运行流程基本是：

所以这里涉及到几个部分：

- 获取本地上传的图片
- 图片缓存到/tmp/目录下
- 通过 AI 进行预测
- 返回数据

按照以上流程，以阿里云函数计算和轻量级 Python Web 框架 Bottle 为例，可以分别实现页面代码以及逻辑代码。其中逻辑代码主要是将上文在本地开发的案例进行稍作修改，与 Bottle 框架进行结合：

```
# -*- coding: utf-8 -*-
from imageai.Prediction import ImagePrediction
import base64
import bottle
import random
import json

# 随机字符串
randomStr = lambda num=5: "".join(random.sample('abcdefghijklmnopqrstuvwxyz', num))

# 模型加载
prediction = ImagePrediction()
prediction.setModelTypeAsResNet()
prediction.setModelPath("/mnt/auto/model/resnet50_weights_tf_dim_ordering_tf_kernels.h5")
prediction.loadModel()

@bottle.route('/image_prediction', method='POST')
def getNextLine():
    postData = json.loads(bottle.request.body.read().decode("utf-8"))
    image = postData.get("image", None)
    image = image.split("base64,")[1]
    # 图片获取
    imagePath = "/tmp/%s" % randomStr(10)
    with open(imagePath, 'wb') as f:
        f.write(base64.b64decode(image))
    # 内容预测
    result = {}
    predictions, probabilities = prediction.predictImage(imagePath, result_count=5)
    for eachPrediction, eachProbability in zip(predictions, probabilities):
```

```
            result[str(eachPrediction)] = str(eachProbability)
    return result

@bottle.route('/', method='GET')
def getNextLine():
    return bottle.template('./html/index.html')

app = bottle.default_app()
if __name__ == "__main__":
    bottle.run(host='localhost', port=8080)
```

而页面代码，则主要是通过 HTML 与 Javascript 实现一个便于测试的前端页面：

除此之外，还需明确该项目所必须的依赖：

```
tensorflow==1.13.1
numpy==1.19.4
scipy==1.5.4
opencv-python==4.4.0.46
pillow==8.0.1
matplotlib==3.3.3
h5py==3.1.0
keras==2.4.3
imageai==2.1.5
bottle==0.12.19
```

为了更快速的在与阿里云 FaaS 平台相同的环境下进行依赖安装，以及项目部署，此时可以通过 Serverless Devs 开发者工具进行部署，在部署之前需要编写资源描述文档 s.yaml：

```
edition: 1.0.0
name: imageAi              # 项目名称
access: 'default'          #  秘钥别名

services:
  imageAi: #  服务名称
    component: devsapp/fc
    actions: # 自定义执行逻辑
      pre-deploy: # 在 deploy 之前运行
        - run: s build --use-docker  # 要运行的命令行
          path: ./ # 命令行运行的路径
      post-deploy: # 在 deploy 之后运行
        - run: s nas command mkdir /mnt/auto/.s
          path: ./ # 命令行运行的路径
        - run: s nas upload -r -n ./.s/build/artifacts/ai-cv-image-prediction/server/.s/python /mnt/auto/.s/python    # 要运行的命令行
          path: ./ # 命令行运行的路径
        - run: s nas upload -r -n ./src/model  /mnt/auto/model    # 要运行的命令行
          path: ./ # 命令行运行的路径
    props: #  组件的属性值
      region: cn-hangzhou
      service:
        name: ai-cv-image-prediction
        description: 图片目标检测服务
        nasConfig: auto
        vpcConfig: auto
        logConfig: auto
      function:
```

```
    name: server
    description: 图片目标检测
    runtime: python3
    codeUri: ./src
    handler: index.app
    memorySize: 3072
    timeout: 60
    environmentVariables:
      PYTHONUSERBASE: /mnt/auto/.s/python
  triggers:
    - name: httpTrigger
      type: http
      config:
        authType: anonymous
        methods:
          - GET
          - POST
          - PUT
  customDomains:
    - domainName: auto
      protocol: HTTP
      routeConfigs:
        - path: /*
```

3.2.2.2 项目部署

通过 Serverless Devs 开发者工具，拉取 Docker 镜像，进行依赖安装：

```
s build --use-docker
```

```
[2021-09-09T15:20:30.451] [INFO ] [S-CLI] - Start ...
[2021-09-09T15:20:30.708] [INFO ] [FC-BUILD] - Build artifact start...
[2021-09-09T15:20:30.720] [INFO ] [FC-BUILD] - Use docker for building.
[2021-09-09T15:20:32.236] [INFO ] [FC-BUILD] - Build function using image: registry.cn-beijing.aliyuncs.com/aliyunfc/runtime-python3.6:build-1.9.19
[2021-09-09T15:20:32.367] [INFO ] [FC-BUILD] - skip pulling image registry.cn-beijing.aliyuncs.com/aliyunfc/runtime-python3.6:build-1.9.19...
builder begin to buildundefined
[2021-09-09T15:25:56.013] [INFO ] [FC-BUILD] - Build artifact successfully.

Tips for next step
======================
* Invoke Event Function: s local invoke
* Invoke Http Function: s local start
* Deploy Resources: s deploy
```

依赖安装完成之后，可以通过 s deploy 进行项目的部署。部署完成，可以看到系统返回的测试地址：

```
imageAi:
  region: cn-hangzhou
  service:
    name: ai-cv-image-prediction
  function:
    name: server
    runtime: python3
    handler: index.app
    memorySize: 3072
    timeout: 60
  url:
    system_url: >-
      https://1583208943291465.cn-hangzhou.fc.aliyuncs.com/2016-08-15/proxy/ai-cv-image-prediction/server/
    custom_domain:
      - domain: >-
          http://server.ai-cv-image-prediction.1583208943291465.cn-hangzhou.fc.devsapp.net
  triggers:
    - type: http
      name: httpTrigger
```

3.2.2.3 项目测试

通过测试地址，可以上传一张图片，并进行预测：

可以看到，预测结果和前文在本地执行的结果是一致的。至此，已经成功的将本地的 Tensorflow 项目部署到了 Serverless 架构上。

3.2.3 项目优化

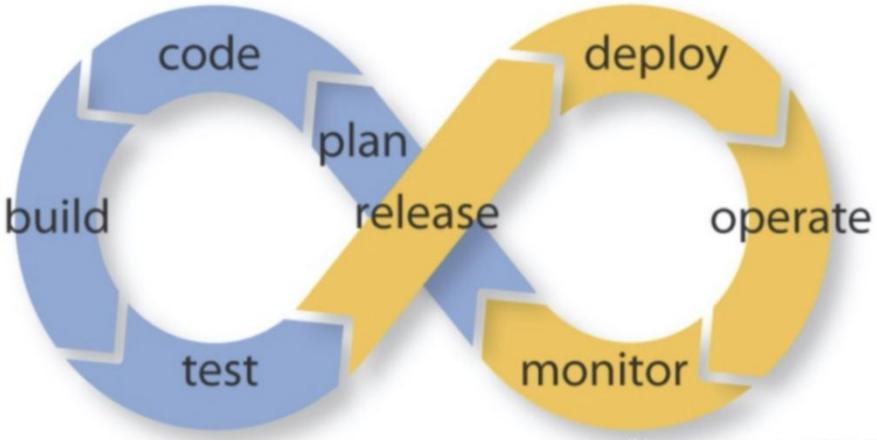
本项目采用的是阿里云函数计算的 Python 运行时。相对来说，该运行时与人工智能项目相结合存在一定的复杂度：

- 由于 Tensorflow 等依赖比较大，而通常情况下，函数计算可以上传的代码包大小在 100M 以下，这就导致模型与依赖很难同步部署到 Serverless 架构下；虽然本项目采用了 NAS 作为挂在盘，完成了大依赖包以及模型文件的上传和加载，但是在开发和部署环节相对比较复杂，比较难上手；可以通过容器镜像等运行时降低这一部分的复杂度；
- 由于 FaaS 平台的环境在很多情况下都会和开发者的本地环境有一定的差异，所以在安装一些需要编译的依赖时会存在一定的无法跨平台运行，所以此时就需要准备和 FaaS 平台一致的环境进行以来的安装和项目的打包，虽然有类似 Serverless Devs 的开发者工具协助依赖安装和项目打包，但是在一定程度上还是相对复杂的，此时同样可以通过容器镜像等运行时降低这一部分的复杂度；

4. Serverless Devs 与 CI/CD 实战

4.1 与 CI/CD 平台/工具集成

CI/CD 是一种通过在应用开发阶段引入自动化来频繁向客户交付应用的方法。CI/CD 的核心概念是持续集成、持续交付和持续部署。作为一个面向开发和运营团队的解决方案，CI/CD 主要针对在集成新代码时所引发的问题。具体而言，CI/CD 可让持续自动化和持续监控贯穿于应用的整个生命周期（从集成和测试阶段，到交付和部署）。这些关联的事务通常被统称为“CI/CD 管道”，由开发和运维团队以敏捷方式协同支持。



在 Serverless 架构下，通常会有很多函数构成一个完整的功能或者服务，Serverless 架构下，这种比较细粒度的功能开发往往会给后期的项目维护带来了极大的不便，包括不限于函数管理的不便、项目的构建、发布层面的不便等。此时在 Serverless 架构中，CI/CD 就显得尤为重要。更加科学、安全的持续集成和部署过程，不仅仅会让整体的业务流程更加规范，也会在一定程度上，降低人为操作，手工集成部署所产生错误的概率，同时也会大规模降低运维人员的工作负担。

如果想要通过 CI/CD 平台或者工具，科学且方便的进行 Serverless 应用的持续集成与交付部署，那么通常情况下是需要借助相对应的开发者工具，例如著名的开源项目 Serverless Framework，Serverless Devs 等。整体的流程通常是：

4.1.1 与 Github Action 的集成

在 Github Action 的 Yaml 文件中，可以增加 Serverless Devs 的相关下载、配置以及命令执行相关能力。

例如，在仓库中可以创建该文件.github/workflows/publish.yml，文件内容：

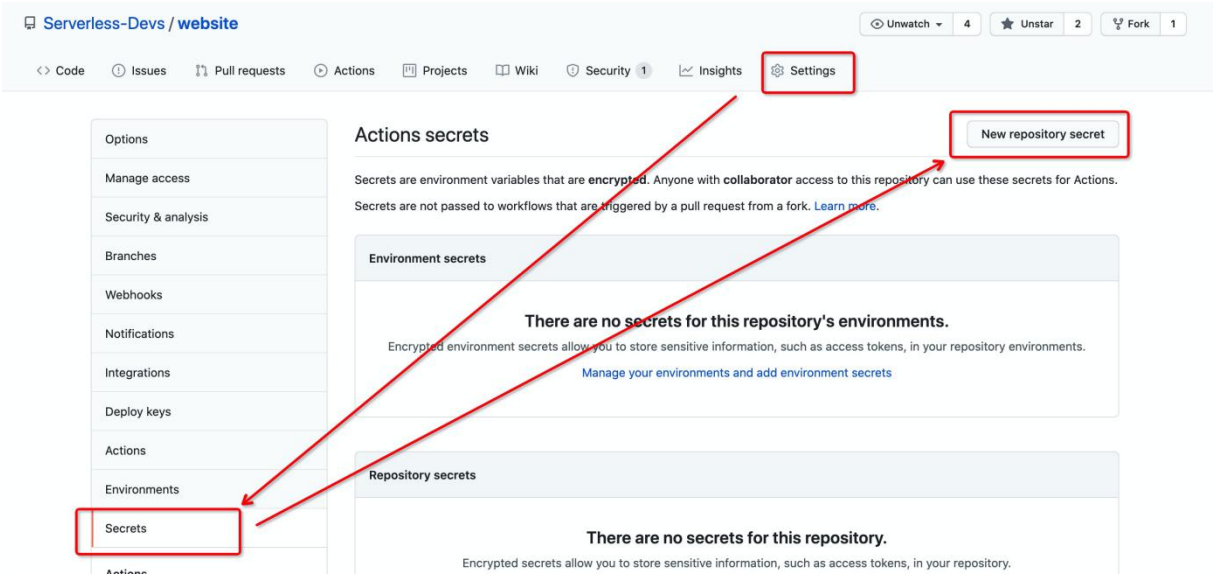
```
name: Serverless Devs Project CI/CD

on:
  push:
    branches: [ master ]

jobs:
  serverless-devs-cd:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v2
        with:
          node-version: 12
          registry-url: https://registry.npmjs.org/
      - run: npm install
      - run: npm install -g @serverless-devs/s
      - run: s config add --AccessKeyID ${{secrets.AccessKeyID}} --AccessKeySecret ${{secrets.AccessKeySecret}} -a default
      - run: s deploy
```

主要包括几个部分的内容：

- run: npm install -g @serverless-devs/s:
  通过 NPM 安装最新版本的 Serverless Devs 开发者工具；
- run: s config add --AccessKeyID ${{secrets.AccessKeyID}} --AccessKeySecret ${{secrets.AccessKeySecret}} -a default
  通过 config 命令进行密钥等信息的配置；
- run: s deploy
  执行某些命令，例如通过 deploy 进行项目的部署，或者 build 等命令进行构建等；

关于密钥的配置：密钥信息的获取是通过${{secrets.*}}进行获取的，所以此时，需要将所需要的密钥和对应的 Key 配置到 Github Secrets 中，例如在上面的案例中，需要 AccessKeyID, AccessKeySecret 等密钥的 Key ，我们就可以配置相关的内容：

1、将密钥信息配置到 Github Secrets 中



2、我们创建多对密钥信息：

例如，我此处配置了三对密钥：

4.1.2 与 Gitee Go 的集成

在开启 Gitee Go 的服务之后，在流水线的 Yaml 文件中，可以增加 Serverless Devs 的相关下载、配置以及命令执行相关能力。

例如，在仓库中可以创建流水线文件，文件内容：

```
name: serverless-devs
displayName: 'Serverless Devs Project CI/CD'
triggers:                                    # 流水线触发器配置
  push:                                      # 设置 master 分支 在产生代码 push 时
精确触发（PRECISE）构建
    - matchType: PRECISE
      branch: master
commitMessage: ''                            # 通过匹配当前提交的 CommitMessage 决
定是否执行流水线
stages:                                      # 构建阶段配置
  - stage:                                   # 定义一个 ID 标识为 deploy-stage ,名为「
Deploy Stage 」的阶段
      name: deploy-stage
      displayName: 'Deploy Stage'
      failFast: false                        # 允许快速失败，即当 Stage 中有任务失败时，
直接结束整个 Stage

      steps:                                 # 构建步骤配置
        - step: npmbuild@1                   # 采用 npm 编译环境
          name: deploy-step                  # 定义一个 ID 标识为 deploy-step ,名为
「 Deploy Step 」的阶段
          displayName: 'Deploy Step'
          inputs:                            # 构建输入参数设定
            nodeVersion: 14.15               # 指定 node 环境版本为 14.15
            goals: |                         # 安装依赖，配置相关主题、部署参数并发布
部署
              node -v
              npm -v
              npm install -g @serverless-devs/s
              s config add --AccessKeyID $ACCESSKEYID --AccessKeySecret $ACCESSK
EYSECRET -a default
              s deploy
```

主要包括几个部分的内容：

- npm install -g @serverless-devs/s:
  通过 NPM 安装最新版本的 Serverless Devs 开发者工具；
- s config add --AccessKeyID $ACCESSKEYID --AccessKeySecret $ACCESSKEYSECRET -a default
  通过 config 命令进行密钥等信息的配置；
- s deploy
  执行某些命令，例如通过 deploy 进行项目的部署，或者 build 等命令进行构建等；

关于密钥的配置：密钥信息的获取是通过$*进行获取的，所以此时，需要将所需要的密钥和对应的 Key 配置到 Gitee 的环境变量管理即可，例如在上面的案例中，需要 ACCESSKEYID, ACCESSKEYSECRET 等密钥的 Key ，我们就可以配置相关的内容：

1、找到 Gitee 的环境变量管理

2、我们创建多对密钥信息：

例如，此处配置了三对密钥：

4.1.3 与 Jenkins 的集成

在准备将 Serverless Devs 集成到 Jenkins 之前，需要先基于 Jenkins 官网 安装并运行 Jenkins。

本地启动 Jenkins 后，通过浏览器进入链接 http://localhost:8080 配置完成基础设置后，需要新增 Credentials 设置，如下图所示：

此时可以根据需要，增加密钥信息，以阿里云为例，新增两个全局凭据：

jenkins-alicloud-access-key-id ：阿里云 accessKeyId
jenkins-alicloud-access-key-secret ：阿里云 accessKeySecret

此时，可以对自身的 Serverless Devs 项目进行完善：

- 创建文件 Jenkinsfile

```
pipeline {
    agent {
        docker {
            image 'maven:3.3-jdk-8'
        }
    }

    environment {
        ALICLOUD_ACCESS = 'default'
        ALICLOUD_ACCESS_KEY_ID       = credentials('jenkins-alicloud-access-key-id')
        ALICLOUD_ACCESS_KEY_SECRET       = credentials('jenkins-alicloud-access-key-
secret')
    }

    stages {
        stage('Setup') {
```

```
            steps {
                sh 'scripts/setup.sh'
            }
        }
    }
}
```

主要的内容包括两个部分：

1、environment 部分，主要是根据上面步骤配置的密钥信息，进行密钥的处理；
2、stages 部分，这里面会有一个部分是 sh 'scripts/setup.sh'，即运行 scripts/setup.sh 文件，进行相关内容的准备和配置；

- 准备 scripts/setup.sh 文件，只需要在项目下，创建该文件即可：

```
#!/usr/bin/env bash

echo $(pwd)
curl -o- -L http://cli.so/install.sh | bash

source ~/.bashrc

echo $ALICLOUD_ACCOUNT_ID
s config add --AccessKeyID $ALICLOUD_ACCESS_KEY_ID --AccessKeySecret $ALICLOUD_
ACCESS_KEY_SECRET -a $ALICLOUD_ACCESS

(cd code && mvn package && echo $(pwd))

s deploy -y --use-local --access $ALICLOUD_ACCESS
```

在该文件中，主要包括了几个部分：

- curl -o- -L http://cli.so/install.sh | bash
  下载并安装 Serverless Devs 开发者工具；
- s config add --AccessKeyID $ALICLOUD_ACCESS_KEY_ID --AccessKeySecret $ALICLOUD_ACCESS_KEY_SECRET -a $ALICLOUD_ACCESS
  配置密钥信息等内容；
- s deploy -y --use-local --access $ALICLOUD_ACCESS
  执行某些命令，例如通过 deploy 进行项目的部署，或者 build 等命令进行构建等；

完成密钥配置之后，可以创建一个 Jenkins 流水线，该流水线的源是目标 github 地址。接下来，就可以开始运行 Jenkins 流水线，运行结束后，就可以得到相关的内容结果。

4.1.4 与云效的集成

在云效中，可以直接选择 Serverless Devs 开发者工具，并在自定义命令中，输入以下内容即可：

```
# input your command here
npm install -g @serverless-devs/s
s config add ---AccessKeyID ${ACCESSKEYID} --AccessKeySecret ${ACCESSKEYSECRET} -a default
s deploy
```

这里主要包括三个部分：

- npm install -g @serverless-devs/s:
  通过 NPM 安装最新版本的 Serverless Devs 开发者工具（虽然云效中已经拥有了相关版本的 Serverless Devs，但是实际上，这个版本可能比较老旧，所以可以通过该命令安装最新版本）；
- s config add --AccessKeyID ${ACCESSKEYID} --AccessKeySecret ${ACCESSKEYSECRET} -a default
  通过 config 命令进行密钥等信息的配置；

- s deploy

  执行某些命令，例如通过 deploy 进行项目的部署，或者 build 等命令进行构建等；

效果如下：

由于在命令中，引用了两个重要的环境变量 ACCESSKEYID，ACCESSKEYSECRET，所以还需要在环境变量中，增加类似的内容：

4.2 Custom Container 的 CI/CD 最佳实践案例

在 Serverless 应用的开发过程中，或者真实业务实际生产过程中，往往会遇到这样一个通用的项目持续发布的流程：

Git Clone -> Docker Build -> Docker Push -> Deploy Function

虽然这是一个简单的流程，却在很多 Serverless 架构的工具中不易实现，但是 Serverless Devs 开发者工具天然对自动化流程有着比较亲密的支持，所以用 Serverless Devs 可以非常简单的实现上面的流程。

4.2.1 实验准备

这个仓库包括了以下的内容：

- 用户的业务代码；
- 构建镜像所需要的 Dockerfile；
- 部署所需要的资源描述文件 s.yaml；
- 流水线描述配置 workflow 相关的 yaml；

以仓库 anycodes/CustomContainerDemo 为例，可以看到这是一个 Node.js 的项目，其中：

- 用户的代码
  - server.js
  - package.json
- 构建镜像所需要的 Dockerfile
  - Dockerfile
- 部署所需要的资源描述文件
  - s.yaml

- 一些流程脚本
  - setup.sh
- 其他文件
  - Github Action 文件
  - version（描述景象 tag 的文件）

4.2.2 流程

在整个项目中，包括两个流程：

- Github Action 的流程
- 自定义 setup.sh 流程

4.2.2.1 Github Action 的流程

这个流程主要是一些环境的初始化等：

```
name: Publish

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v2
        with:
```

```
            node-version: 12
            registry-url: https://registry.npmjs.org/
        - run: npm install -g @serverless-devs/s
        - run: s config add --AccountID ${{secrets.AccountID}} --AccessKeyID ${{secrets.A
ccessKeyID}} --AccessKeySecret ${{secrets.AccessKeySecret}} -a publish_access
        - run: chmod +x ./setup.sh
        - run: ./setup.sh
```

整个过程为确定 nodejs 环境，安装 Serverless Devs，配置密钥信息。

完成上述的初始化和密钥配置之后，可以直接执行我们的流程./setup.sh

4.2.2.2 自定义 Setup.sh 流程

该流程主要做了几个事情：

- 明确镜像 registry 地址和 tag（此处 tag 是从 version 文件读取的）
- 通过 serverless devs fc 组件提供的 build 能力，进行构建操作
- 通过 deploy 方法进行项目部署

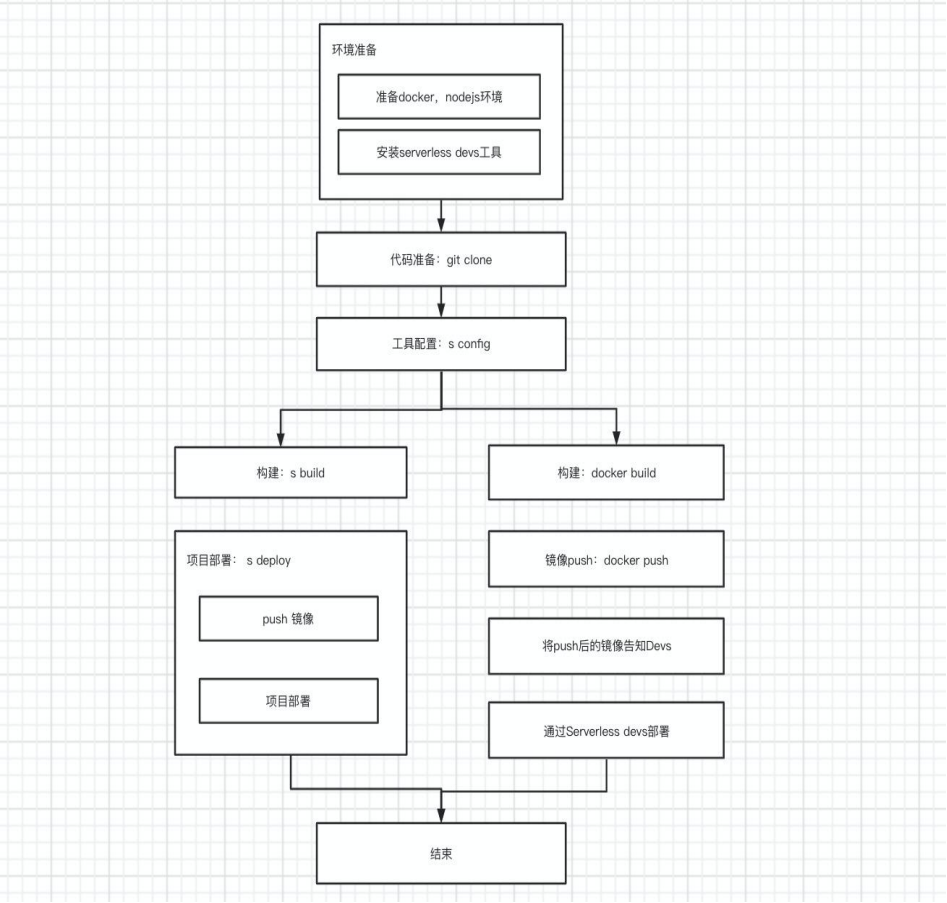
```
#!/usr/bin/env bash

# git clone && cd repo

version=$(cat version)
registry='registry.cn-shanghai.aliyuncs.com/custom-container/test:'

export image=$registry$version
s build --use-docker
s deploy --push-registry acr-internet --use-local -y
```

这里有一个问题：谁进行的 docker build 以及谁进行的 docker push?

- 在本例子中 docker build 行为是由 Serverless Devs 帮做的，但是此出也可以不选择 s build，可以选择更为原生的 docker build；
- 在制定了相关的镜像信息之后，Serverless Devs 开发者工具会帮助开发者自动 push 镜像；

4.2.2.2 自定义 Setup.sh 流程

项目的整体流程通过图像化的表达，可以加单表述为：

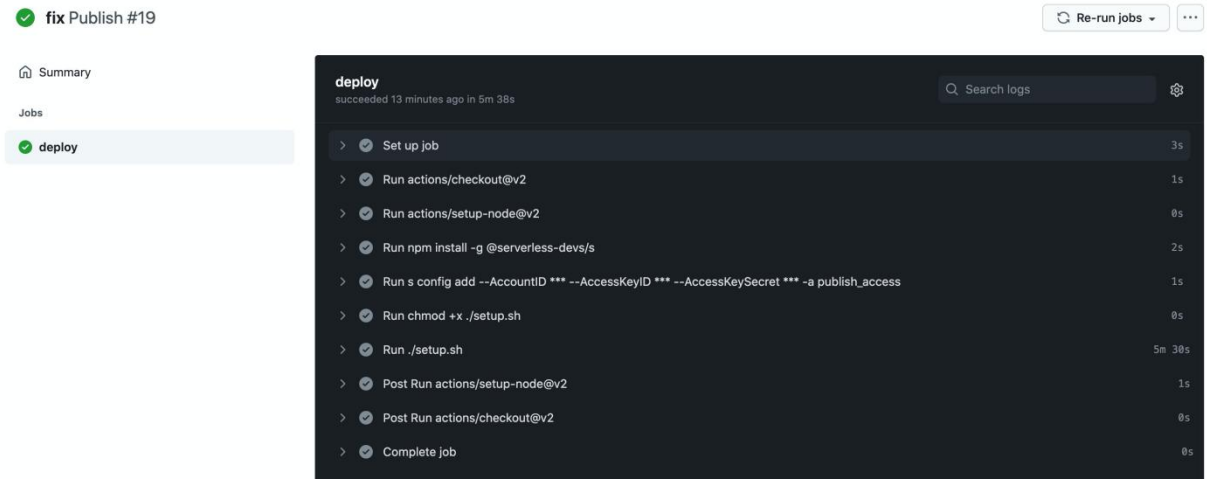


4.2.3 项目测试

由于在 Github Action 中声明的是：

```
on:
  push:
    branches: [ main ]
```

所以，此时只需要 push 代码，即可触发发布流程：



部署后的地址效果：

4.2.4 案例总结与注意事项

在上面的步骤中，进行了密钥的配置：

```
s config add --AccessKeyID ${{secrets.AccessKeyID}} --AccessKeySecret ${{secrets.AccessKeySecret}} -a publish_access
```

这里面其实最后有一个参数是-a publish_access，它的含义是为当前密钥指定一个别名，因为 Serverless Devs 支持多密钥的，所以为当前密钥配置一个别名，在以后的使用过程中可以指定，例如在当前的 Yaml 中，第三行有：

```
access: publish_access
```

用来指定使用该密钥，测试的 Yaml 配置如下：

```
edition: 1.0.0
name: fcDeployApp
access: publish_access

services:
  HelloWorld:
    component: fc
    props:
      region: cn-shanghai
      service:
        name: custom-container-test
        description: demo for custom-container-test
      function:
        name: custom-container-function
        runtime: custom-container
        caPort: 8080
        codeUri: ./
        timeout: 60
        customContainerConfig:
          image: ${env(image)}
          command: '["node"]'
          args: '["server.js"]'
      triggers:
        - name: httpTrigger
          type: http
          config:
            authType: anonymous
```

```
            methods:
              - GET
              - POST
      customDomains:
        - domainName: auto
          protocol: HTTP
          routeConfigs:
            - path: /*
```

完整的 Yaml 配置可以参考：
https://github.com/devsapp/fc/blob/main/docs/Others/yaml.md

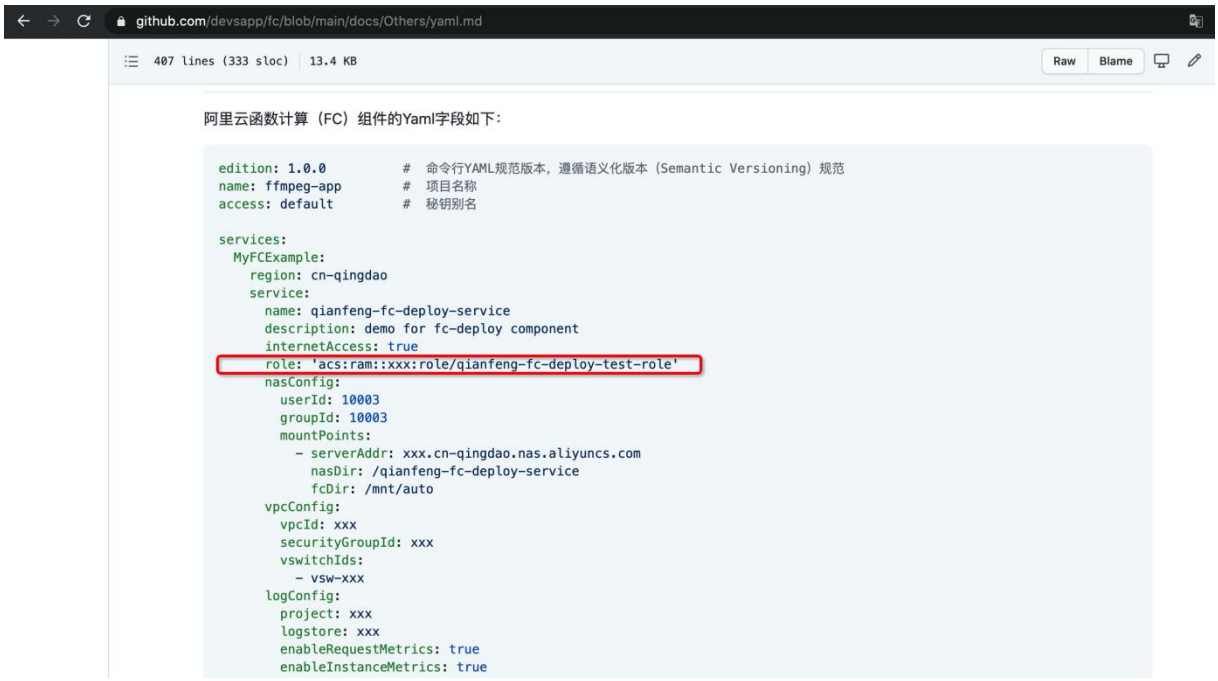
在上面的 Yaml 中，其实可以看到 image: ${env(image)}，其实 Serverless Devs 的 Yaml 支持多种形式的变量：

- 获取当前机器中的环境变量：${env(环境变量)}，例如${env(secretId)}
- 获取外部文档的变量：${file(路径)}，例如${file(./path)}
- 获取全局变量：${vars.*}
- 获取其他项目的变量：${projectName.props.*}
- 获取 Yaml 中其他项目的结果变量：${projectName.output.*}

实战举例，例如当我需要访问数据库等，此时我并不想把密钥明文配置到 Yaml 中，便可以考虑，将密钥配置到环境变量中，进行直接使用。

关于 **构建** 问题：
如果使用 s build --use-docker 构建镜像，则需要确保 s.yml 中的 codeUri 字段指向的目录中包含 Dockerfile。

关于 **权限** 问题：
如果配置的密钥权限不够（例如是子账号），则可能会导致用户无法创建某些权限，进而导致部署不成功，这个时候可以考虑让主账号创建好相关的 Role，并且在此处指定：



关于密钥最小权限：

- AliyunFCFullAccess
- AliyunContainerRegistryFullAccess

关于所绑定的 Role 的最小权限：

- AliyunContainerRegistryReadOnlyAccess

4.3 基于 Gitee Go 的函数代码更新与版本发布

前文中，介绍了如何通过 Github + Github Action 进行函数计算的业务发布能力，下面将会分享如何通过 Gitee + Gitee Go 实现：

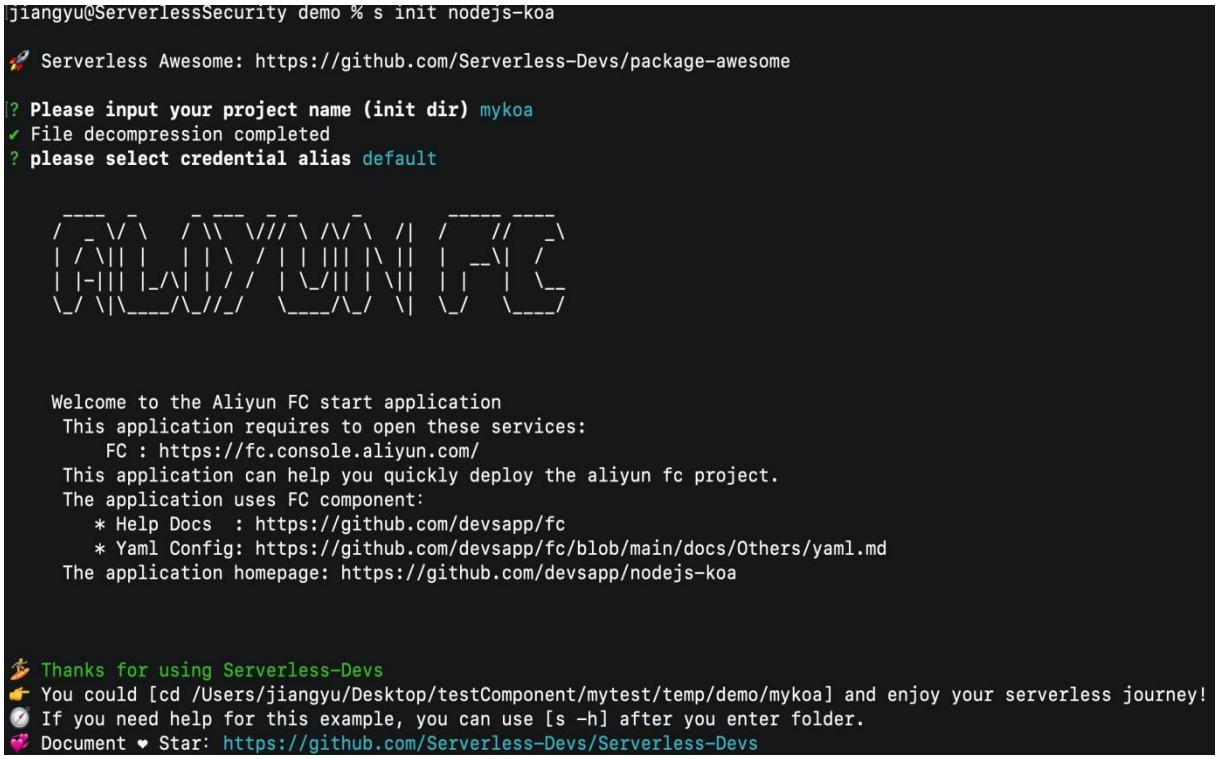
- 单纯更新函数代码
- 进行版本发布

4.3.1 实验准备

首先需要在本地初始化一个基于 Node.js 运行时的 Koa.js 项目

```
s init nodejs-koa
```

在进行初始化的时候会交互式填写相关内容，例如项目目录，选择密钥等：



配置好密钥与目录之后，进入到项目目录，并进行部署操作：

```
jiangyu@ServerlessSecurity demo % cd mykoa
jiangyu@ServerlessSecurity mykoa % s deploy
[2021-07-07T15:08:42.127] [INFO ] [S-CLI] - Start ...
[2021-07-07T15:08:42.139] [INFO ] [S-CLI] - Start the pre-action
[2021-07-07T15:08:42.139] [INFO ] [S-CLI] - Action: npm install --registry=https://registry.npm.taobao.org
npm WARN deprecated querystring@0.2.0: The querystring API is considered Legacy. new code should use the URLSearchParams API inste
ad
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN start-koa@1.0.0 No description
npm WARN start-koa@1.0.0 No repository field.
```

稍等片刻，便可看到项目已经完成部署：

```
    config:
      authType: anonymous
      methods:
        - GET
  customDomains:
    - domainName: >-
        http://http-trigger-function.koademo.1583208943291465.cn-hangzhou.fc.devsapp.net
      protocol: HTTP
      routeConfigs:
        - path: /*
          methods:
            - GET
          serviceName: koademo
          functionName: http-trigger-function
```

此时，打开系统返回的地址，可以看到测试页面：

至此，完成了实验前的业务代码的准备。

4.3.2 基于 Gitee 的 CD 能力建设

在这一步，需要做几个事情：

- 准备一个 Gitee 仓库；
- 在仓库中 push 刚刚准备妥当的代码；
- 配置环境变量；
- 开启 Gitee Go；
- 尝试对代码进行更新以触发 CD 流程；

创建一个 Gitee 仓库：

push 代码到仓库：

```
jiangyu@ServerlessSecurity mykoa % git push origin master
Enumerating objects: 483, done.
Counting objects: 100% (483/483), done.
Delta compression using up to 8 threads
Compressing objects: 100% (457/457), done.
Writing objects: 100% (483/483), 371.19 KiB | 5.54 MiB/s, done.
Total 483 (delta 90), reused 0 (delta 0)
remote: Resolving deltas: 100% (90/90), done.
remote: Powered by GITEE.COM [GNK-5.0]
To https://gitee.com/anycodes_admin/gitee-serverless-devs-koa.git
 * [new branch]      master -> master
```

推送后：

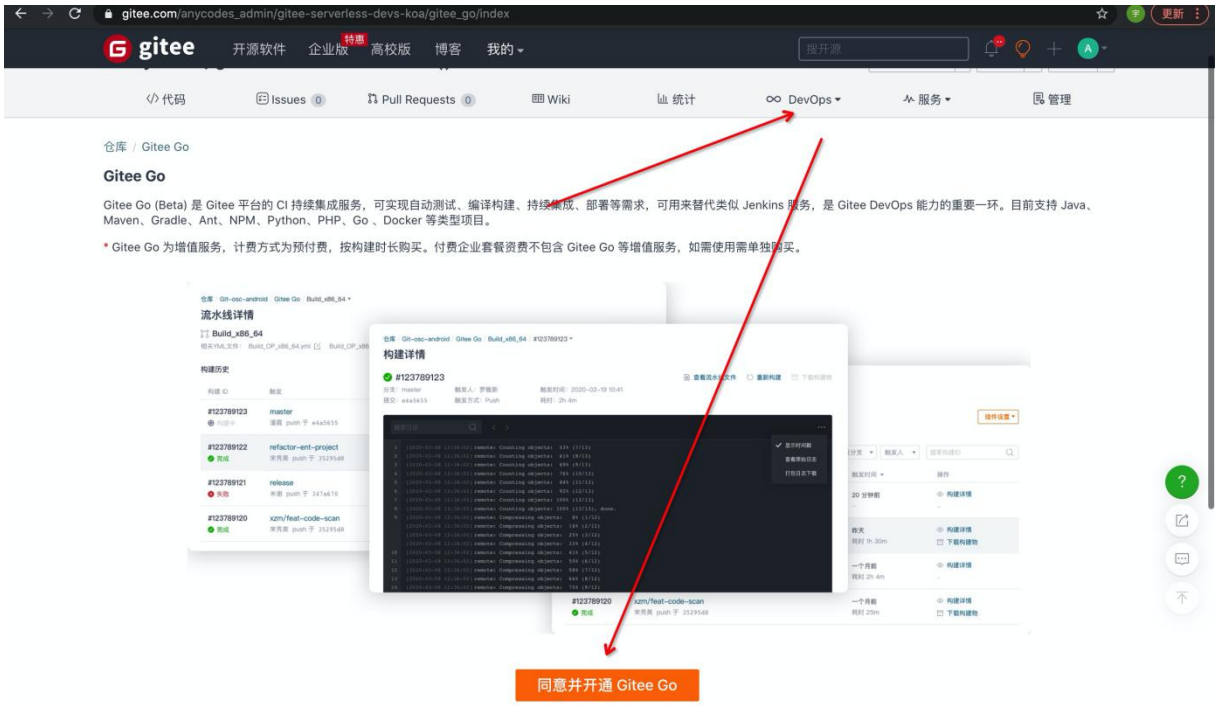
配置环境变量

此时，将阿里云的密钥等信息配置到环境变量：

例如配置：

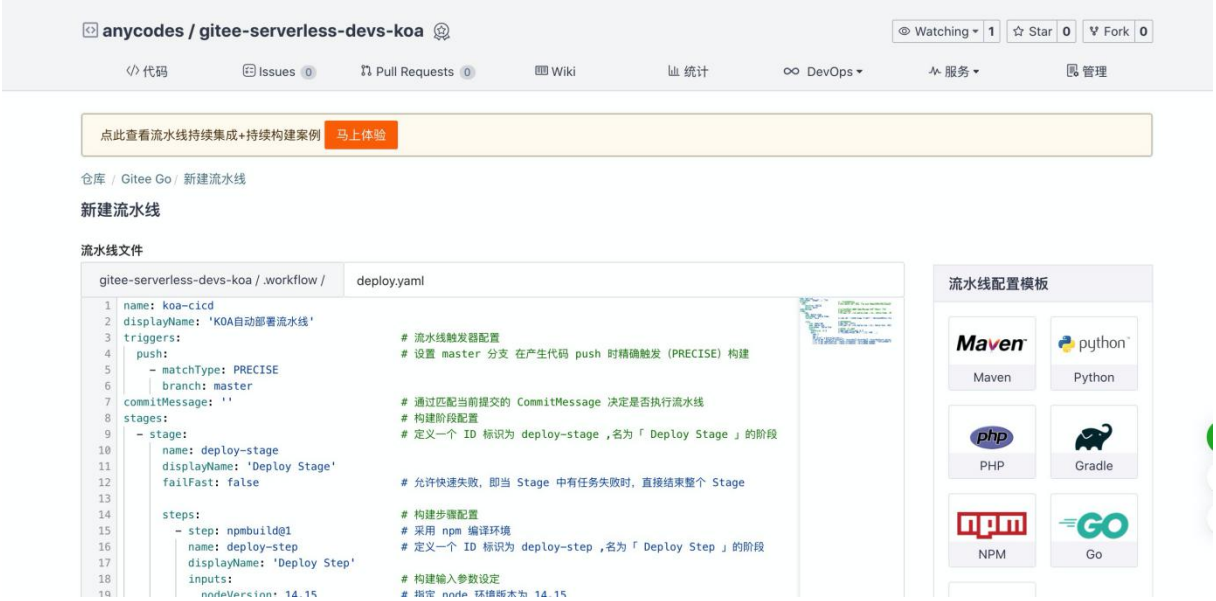
配置后的效果：

此时开启 Gitee Go：



然后：

点击创建流水线，并输入流水线内容：



流水线文件名：deploy.yml

流水线配置：

```
name: koa-cicd
displayName: 'KOA 自动部署流水线'
triggers:                                    # 流水线触发器配置
  push:                                      # 设置 master 分支 在产生代码 push 时精确触发（PRECISE）构建
    - matchType: PRECISE
      branch: master
commitMessage: ''                            # 通过匹配当前提交的 CommitMessage 决定是否执行流水线
stages:                                      # 构建阶段配置
  - stage:                                   # 定义一个 ID 标识为 deploy-stage ,名为「 Deploy Stage 」的阶段
      name: deploy-stage
      displayName: 'Deploy Stage'
      failFast: false                        # 允许快速失败，即当 Stage 中有任务失败时，直接结束整个 Stage
```

```
    steps:                                  # 构建步骤配置
      - step: npmbuild@1                    # 采用 npm 编译环境
        name: deploy-step                   # 定义一个 ID 标识为 deploy-step ,名为「 Deploy Step 」的阶段
        displayName: 'Deploy Step'
        inputs:                             # 构建输入参数设定
          nodeVersion: 14.15                # 指定 node 环境版本为 14.15
          goals: |                          # 安装依赖，配置相关主题、部署参数并发布部署
            node -v
            npm -v
            npm install -g @serverless-devs/s
            s config add --AccountID $ACCOUNTID --AccessKeyID $ACCESSKEYID --AccessKeySecret $ACCESSKEYSECRET -a default
            cd src && npm install
            s cli fc-api updateFunction --region cn-hangzhou --serviceName koademo --functionName http-trigger-function --code '{"zipFile":"./"}'
            s cli fc-api publishVersion --region cn-hangzhou --serviceName koademo
```

其实核心部分只有 5 句话：

```
npm install -g @serverless-devs/s
s config add --AccessKeyID $ACCESSKEYID --AccessKeySecret $ACCESSKEYSECRET -a default
cd src && npm install
s cli fc-api updateFunction --region cn-hangzhou --serviceName koademo --functionName http-trigger-function --code '{"zipFile":"./"}'
s cli fc-api publishVersion --region cn-hangzhou --serviceName koademo
```

- npm install -g @serverless-devs/s: 安装 Serverless Devs 工具
- s config add --AccessKeyID $ACCESSKEYID --AccessKeySecret $ACCESSKEYSECRET -a default: 根据刚才配置的环境变量，取环境变量内容配置密钥
- cd src && npm install: 进入 src 目录，并安装依赖
- s cli fc-api updateFunction --region cn-hangzhou --serviceName koademo --functionName http-trigger-function --code '{"zipFile":"./src/"}': 更新函数代码
- s cli fc-api publishVersion --region cn-hangzhou --serviceName koademo: 发布函数版本

此时，可以对 index.js 内容进行更改：

然后保存，稍等片刻，可以在流水线中看到这个发布流程：

此时，可以查看流程中详情信息：

稍等片刻，可以看到 CD 流程完成：

完成之后，可以点击查看线上的代码：

4.3.3 案例总结

基于 Serverless 架构进行项目开发，与 CI/CD 的集成，搞定自动化发布，是必不可少的“课程”，希望通过本节，读者可以对相关的流程有进一步的思路，能够应用到自己的项目中。

5. Serverless 应用开发经验/常见思路

由于 Serverless 架构下应用开发，在一定程度上相对于传统架构的应用开发具有比较大的区别点，例如天然分布式架构会让很多框架丧失一定的"便利性"，无状态的特点又让很多"传统架构下看起来再正常不过的操作"变得异常风险。所以这一小节将会针对 Serverless 架构下，常见的应用开发注意事项进一步探究。

5.1 如何上传文件

在传统 Web 框架中，上传文件是非常简单和便捷的，例如 Python 的 Flask 框架：

```
f = request.files['file']
f.save('my_file_path')
```

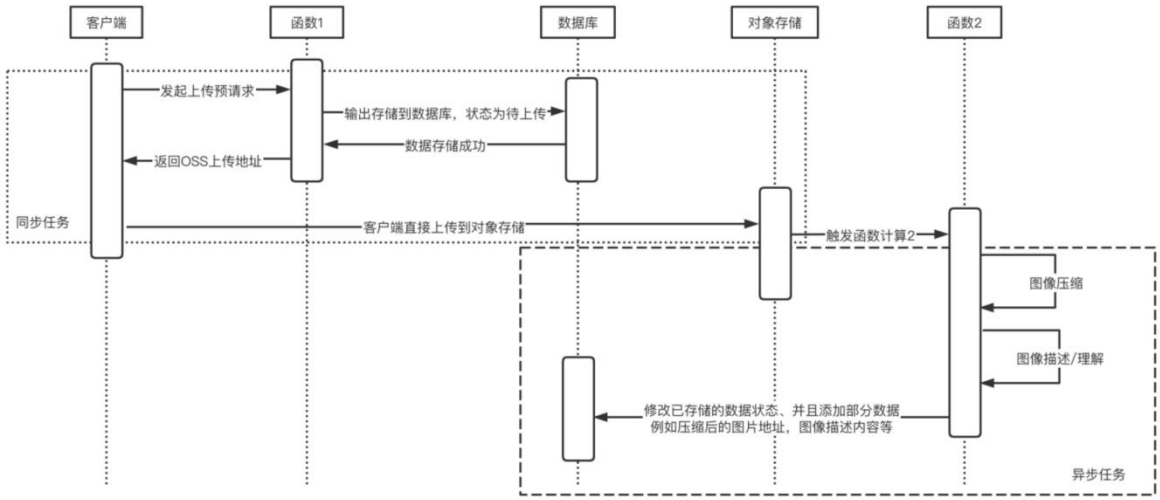
但是在 Serverless 架构下，却不能直接上传文件，因为：

- 一般情况下，一些云平台的 API 网关触发器会将二进制文件转换成字符串；不便直接获取和存储；
- 一般情况下，API 网关与 FaaS 平台之间传递的数据包有大小限制，很多平台被限制在 6M；
- FaaS 平台大都是无状态的，即使存储到当前实例中，也会随着实例释放而导致文件丢失；

所以，传统框架中常用的上传方案是不太适合在 Serverless 架构中直接使用的，在 Serverless 架构上传文件的方法通常有两种：

- 一种是 BASE64 后上传，持久化到对象存储或者是 NAS 中，这种做法可能会触及到 API 网关与 FaaS 平台之间传递的数据包有大小限制，所以一般使用这种上传方法的通常是上传头像等小文件的业务场景；

- 第二种上传方法是通过对象存储等平台来上传，因为客户端直接通过密钥等信息来将文件直传到对象存储是有一定风险的，所以通常情况是客户端发起上传请求，函数计算根据请求内容进行预签名操作，并将预签名地址返回给客户端，客户端再使用指定的方法进行上传，上传完成之后，可以通过对象存储触发器等来对上传结果进行更新等：



5.2 文件读写与持久化方法

应用在执行过程中，可能会涉及到文件的读写操作，或者是一些文件的持久化操作。在传统的云主机模式下，通常情况下是可以直接读写文件，或者将文件持久化某个目录下，但是在 Serverless 架构下并不是这样的。

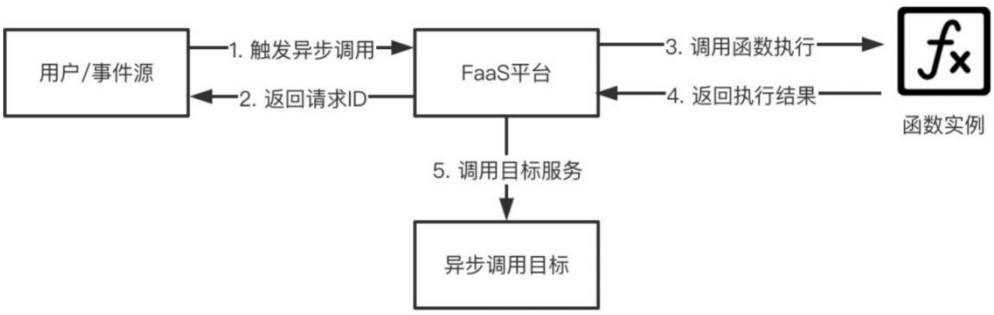
由于 FaaS 平台是无状态的，并且用过之后会被销毁，所以文件如果需要持久化并不能直接持久化在实例中，可以选择持久化到其他的服务中，例如对象存储、NAS 等。

同时，在不配置 NAS 的情况下，FaaS 平台通常情况下之后/tmp 目录具有可写权限，所以部分临时文件可以缓存在/tmp 文件夹下。

5.3 慎用部分 Web 框架的特性

5.3.1 异步

函数计算是请求级别的隔离，所以可以认为这个请求结束了，实例就有可能进入到一个“静默”的状态了，而在函数计算中，API 网关触发器通常是同步调用（以阿里云函数计算为例，通常只在定时触发器、OSS 事件触发器、MNS 主题触发器和 IoT 触发器等几种情况下是异步触发），这就意味着当 API 网关将结果返回给客户端的时候，整个函数就会进入“静默”状态，或者被销毁，而不是会继续执行完异步方法，所以通常情况下像 Tornado 等框架就很难在 Serverless 架构下发挥其异步的作用。当然，如果使用者需要异步能力，可以参考云厂商所提供异步方法，以阿里云函数计算为例，阿里云函数计算为用户提供了一种异步调用能力，当函数的异步调用被触发后，函数计算会将触发事件放入内部队列中，并返回请求 ID，而具体的调用情况及函数执行状态将不会返回。如果用户希望获得异步调用的结果，则可以通过配置异步调用目标来实现：



5.3.2 定时任务

Serverless 架构下，应用一旦完成当前请求，就会进入到“静默”状态，甚至实例会被销毁，所以这就导致一些自带定时任务的框架没有办法正常执行定时任务。因为函数计算通常是由事件触发，不会自主定时启动，例如 Egg 项目中设定了一个定时任务，但是在实际的函数计算中如果没有通过触发器触发该函数，那么该函数是不会被触发，函数也不会从内部自动启动来执行定时任务，所以此时可以使用各个云厂商为其 FaaS 平台提供的定时触发器，通过定时触发器触发指定方法，来替代定时任务。

5.4 要注意应用组成结构

5.4.1 静态资源与业务逻辑

在 Serverless 架构下，静态资源更应该在对象存储与 CDN 的加持下对外提供服务，否则所有的资源都在函数中，并通过函数计算对外暴露，不仅仅会让函数的真实业务逻辑并发度降低，也会造成更多的成本支出。尤其是将一些已有的程序迁移到 Serverless 架构上，例如 Wordpress 等，更是要注意将静态资源与业务逻辑进行拆分，否则在高并发的情况下，性能与成本都将会受到比较严格的考验。

5.4.2 业务逻辑的进一步拆分

在众多云厂商中，函数的收费标准都是依靠运行时间和配置的内存，以及产生的了流量进行收费的。如果一个函数的内存设置不合理，会导致成本成倍增加。想要保证内存设置合理，更要保证业务逻辑结构的可靠性。

以阿里云函数计算为例，当一个应用，有两个对外的接口，其中有一个接口的内存消耗在 128 MB 以下，另一个接口的内存消耗稳定在 3000MB 左右。这两个接口，平均每天会被触发 1000 0 次，并且时间消耗均在 100ms。如果两个接口写到一个函数中，那么这个函数可能需要将内存设置在 3072MB，同时在用户请求内存消耗较少的接口时，可能在冷启动的情况下难以得到较好的性能表现；如果两个接口分别写到两个函数中，则两个函数内存分别设置成 128MB 以及 3072MB 即可：

	函数 1	函数 2 内存	日消费	月消费
写到一个函数中	3072MB 20000 次/日	-	66.38 元	1991.4 元
写到两个函数中	3072MB 10000 次/日	128MB 10000 次/日	34.59 元	1037.7 元
费用差	-	-	31.79 元	953.7 元

通过上表，可以明确看出，当合理的、适当地把业务进行拆分之后会在一定程度上更节约成本，以上面的例子来看，成本节约近 50%。

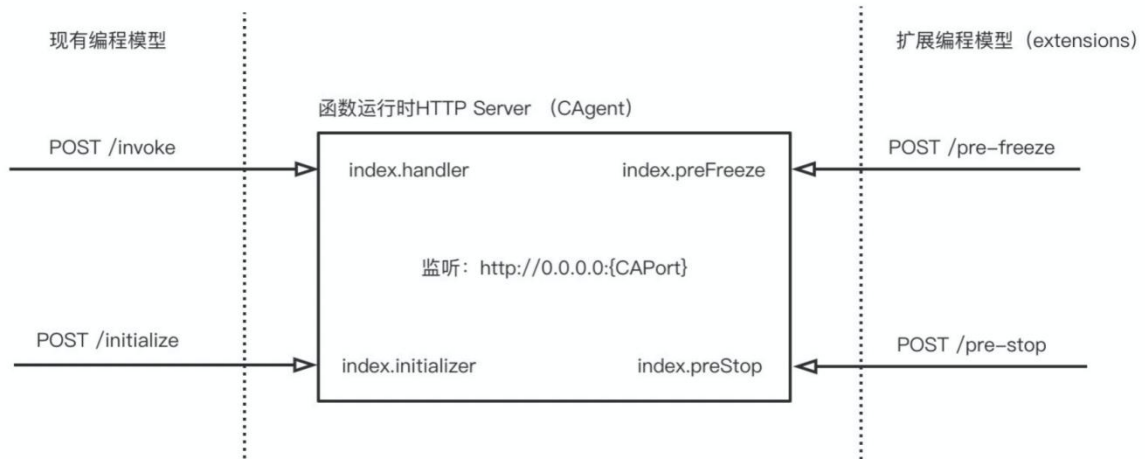
5.5 善于利用函数的厂商特性

各个云厂商的 FaaS 平台都有一些“平台特性”，所谓的平台特性就是说这些功能可能并不是《CNCF WG-Serverless Whitepaper v1.0》中规定的能力，或者描述的能力，仅仅是作为云平台根据自身业务发展和诉求，从用户角度出发挖掘出来，并且实现的功能，可能只在某个云平台或者某几个云平台所拥有的功能，这类功能一般情况下如果利用得当会让业务性能等有质的提升。例如阿里云函数计算提 Pre-freeze & Pre-stop

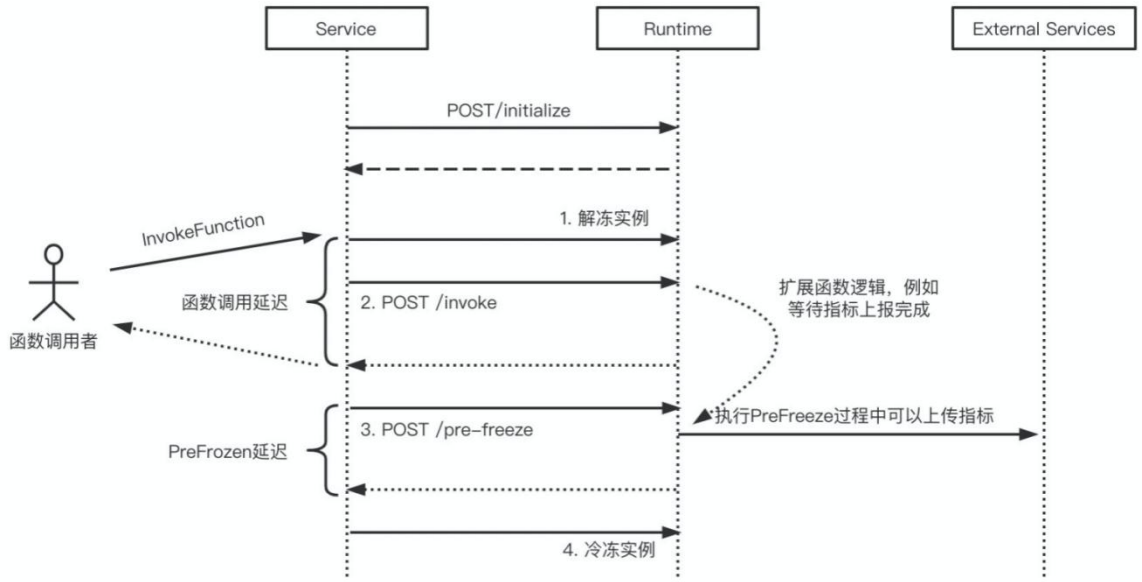
以阿里云函数计算为例，在平台发展过程中，挖掘出用户痛点（尤其是阻碍传统应用平滑迁移至 Serverless 架构）：

- 异步背景指标数据延迟或丢失：如果在请求期间没有发送成功，则可能被延迟至下一次请求，或者数据点被丢弃。
- 同步发送指标增加延迟：如果在每个请求结束后都调用类似 Flush 接口，不仅增加了每个请求的延迟，对于后端服务也产生了不必要的压力。
- 函数优雅下线：实例关闭时应用有清理连接，关闭进程，上报状态等需求。在函数计算中实例下线时机开发者无法掌握，也缺少 Webhook 通知函数实例下线事件。

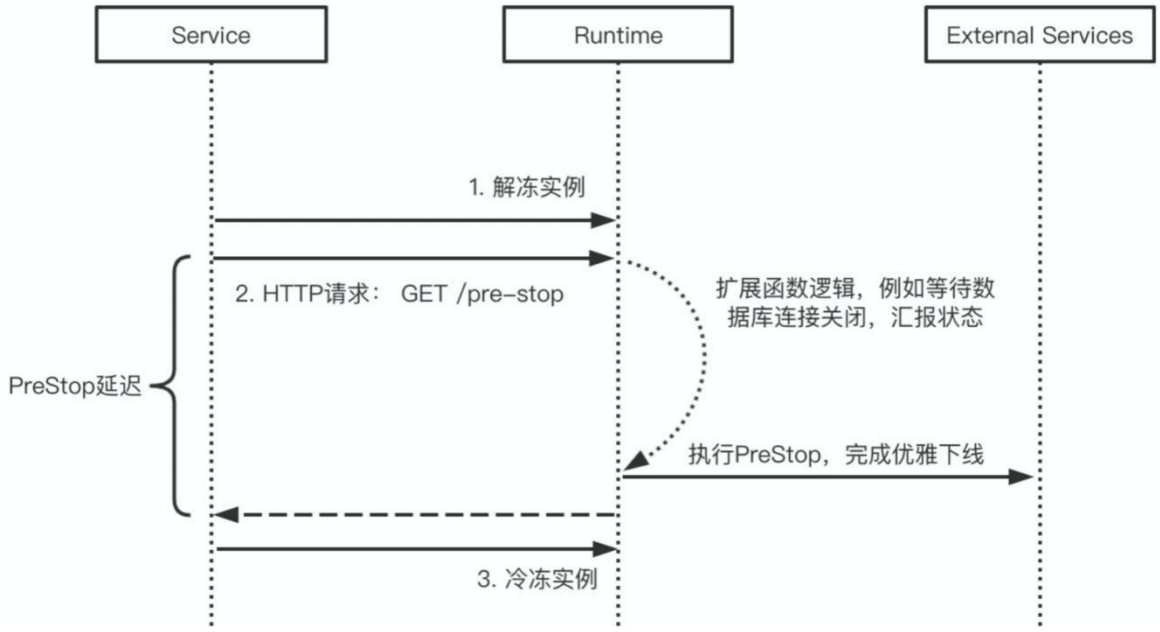
根据这些痛点发布了运行时扩展（runtime extensions）功能。该功能在现有的 HTTP 服务编程模型上扩展，在已有的 HTTP 服务器的模型中增加了 PreFreeze 和 PreStop webhooks。扩展开发者实现 HTTP handler，监听函数实例生命周期事件，如图所示：



PreFreeze：在每次函数计算服务决定冷冻当前函数实例前，函数计算服务会调用 HTTP GET /pre-freeze 路径，扩展开发者负责实现相应逻辑以确保完成实例冷冻前的必要操作，例如等待指标发送成功等。函数调用 InvokeFunction 的时间不包 PreFreeze hook 的执行时间。



PreStop：在每次函数计算决定停止当前函数实例前，函数计算服务会调用 HTTP GET /pre-stop 路径，扩展开发者负责实现相应逻辑以确保完成实例释放前的必要操作，如关闭数据库链接，以及上报、更新状态等。



5.6 传统框架迁移方案与策略

与其说 Serverless 架构是一个新的概念/架构，倒不如说他是一个全新的思路，一种新的编程范式，在这种新的架构下，或者说新的编程范式下，使用全新的思路来做 Serverless 应用是再好不过的了，但是实际上并不是这样的，原生的 Serverless 开发框架是非常少的，以 Web 框架为例，目前的主流的 Web 框架“均不支持 Serverless 模式部署”，一方面是要尝试接触 Serverless，一方面又没办法完全放弃传统框架，所以将传统框架如何更简单、更快速、更科学地部署到 Serverless 架构上就是一个值得探讨的问题。

5.6.1 传统框架迁移案例

请求集成方案实际上就是把真实的 API 网关请求，直接透传给 FaaS 平台，而不在中途增加任何转换逻辑，以阿里云函数计算的 HTTP 函数为例，当想要把传统框架（例如 Django，Flask，Express，Next.js 等）部署到阿里云函数计算平台上，并且体验 Serverless 带来的按量付费，弹性伸缩等红利时，得益于阿里云函数计算的 HTTP 函数和 HTTP 触发器，使用者不仅可以快速、简单地将框架部署到阿里云函数计算，更可以保持和传统开发一样的体验。以 Python 的 Bottle 框架为例，当开发一个 Bottle 项目之后：

```
# index.pyp
import bottle

@bottle.route('/hello/<name>')
def index(name):
    return "Hello world"

if __name__ == '__main__':
    bottle.run(host='localhost', port=8080, debug=True)
```

可以直接在本地进行调试。当想要把该项目部署到阿里云函数计算上，只需要增加一个 default_app 的对象即可：

```
app = bottle.default_app()
```

整个项目：

```
# index.py
import bottle

@bottle.route('/hello/<name>')
def index(name):
    return "Hello world"

app = bottle.default_app()

if __name__ == '__main__':
    bottle.run(host='localhost', port=8080, debug=True)
```

此时，在阿里云函数计算平台，创建函数时，将函数入口设置为：index.app 即可。除了 Bottle 之外，其他的 Web 框架的操作方法是类似的，再以 Flask 为例：

```
# index.py
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run(
        host="0.0.0.0",
        port=int("8001")
)
```

在配置函数的时候写上入口函数为：index.app 即可，就可以保证该 Flask 项目运行在函数计算平台上。

当然，除了使用已有的语言化的 Runtime，还可以考虑使用 Custom Runtime 和 Custom Container 来实现，例如，一个 Web 项目完成之后，可以编写一个 Bootstrap 文件（在 Bootstrap 文件中写一些启动命令即可），例如我要启动一个 Express 的项目，我把我的 Express 项目准备完成之后，可以直接通过 Bootstrap 为：

```
#!/usr/bin/env bash
export PORT=9000
npm run star
```

5.6.2 通过开发者工具快速迁移/部署

如果通过开发者工具进行传统框架的支持，可以直接通过 Serverless Devs 的命令，进行案例的项目的创建。目前 Serverless Devs 已经支持以下框架：

Nodejs	**Python**	**PHP**	**Java**	**Others**
Express	Flask	Think PHP	SpringBoot	React
Egg	Tornado	laravel		Vue
Nextjs	Bottle	Discuz		Hexo
Nuxtjs	Web.py	WordPress		Docusaurus
Hapi	Django	Zblog		Vuepress
Koa	FastAPI	Ecshop		
Nest	Pyramid	Metinfo		
Connect		Whatsns		
Think.js		Typecho		

详情可以参考：https://github.com/devsapp/start-web-framework

以 Express 项目为例，可以在命令行工具执行：

```
s init start-express
```

即可进行项目的初始化：

```
      _____
     |  ___|
     | |____   ___ __   _ __ ___   ___ ___
     |  __\ \/ / '_ \| '__/ _ \ __/ __|
     | |___>  <| |_) | | |  __/\__ \__ \
     \____/_/\_\ .__/|_|  \___||___/___/
               | |
               |_|

? please select credential alias default

     Welcome to the start-express application
      This application requires to open these services:
           FC : https://fc.console.aliyun.com/
      Express development docs: https://www.expressjs.com.cn/4x/api.html

      * 额外说明：s.yaml 中声明了 actions：
         部署前执行：npm install --production
        如果遇到 npm 命令找不到等问题，可以适当进行手动项目构建，并根据需要取消 acti
ons 内容
      * 项目初始化完成，您可以直接进入项目目录下，并使用 s deploy 进行项目部署
```

```
    Thanks for using Serverless-Devs
    You could [cd /Users/jiangyu/Desktop/fc-custom-lua-event/image-prediction-app/start-express] and enjoy your serverless journey!
    If you need help for this example, you can use [s -h] after you enter folder.
    Document ❤ Star：https://github.com/Serverless-Devs/Serverless-Devs
```

完成之后，可以进入项目进行项目的部署：

```
$ s deploy


framework:
  region:     cn-beijing
  service:
    name:  web-framework
  function:
    name:           express
    runtime:       custom
    handler:       index.handler
    memorySize:  128
    timeout:       60
  url:
    system_url:       https://1583208943291465.cn-beijing.fc.aliyuncs.com/2016-08-15/proxy/web-framework/express/
    custom_domain:
      -
        domain:  http://express.web-framework.1583208943291465.cn-beijing.fc.devsapp.net
  triggers:
    -
      type:  http
      name:  httpTrigger
```

此时可以通过浏览器打开页面：

此时，可以根据案例提供的 bootstrap 以及 s.yaml 进行参考，并将自身的项目部署/迁移到阿里云 Serverless 架构。其中 bootstrap 为：

```
#!/bin/bash

node  index.js
```

其中 s.yaml 为：

```
# ------------------------------------
#    欢迎您使用阿里云函数计算 FC 组件进行项目开发
#    组件仓库地址/帮助文档：https://github.com/devsapp/fc
#    Yaml 参考文档：https://github.com/devsapp/fc/blob/jiangyu-docs/docs/zh/yaml.md
#    关于：
#        - Serverless Devs 和 FC 组件的关系、如何声明/部署多个函数、超过 50M 的代码包如何部署
#        - 关于.fcignore 使用方法、工具中.s 目录是做什么、函数进行 build 操作之后如何处理 build 的产物
```

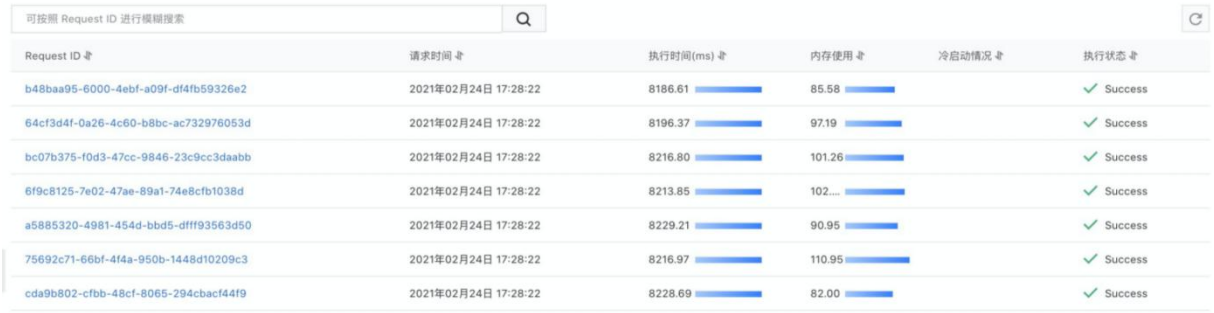
```
#   等问题，可以参考文档：https://github.com/devsapp/fc/blob/jiangyu-docs/docs/zh/tips.md
#   关于如何做 CICD 等问题，可以参考：https://github.com/Serverless-Devs/Serverless-Devs/blob/master/docs/zh/cicd.md
#   有问题快来钉钉群问一下吧：33947367
# ----------------------------------
edition: 1.0.0              #  命令行 YAML 规范版本，遵循语义化版本（Semantic Versioning）规范
name: framework             #  项目名称
access: "default"           #  秘钥别名

services:
  framework: # 业务名称/模块名称
    component: fc  # 组件名称
    actions:
      pre-deploy: # 在 deploy 之前运行
        - run: npm install --production  # 要运行的命令行
          path: ./code # 命令行运行的路径
    props: # 组件的属性值
      region: cn-beijing
      service:
        name: web-framework
        description: 'Serverless Devs Web Framework Service'
      function:
        name: express
        description: 'Serverless Devs Web Framework Express Function'
        codeUri: './code'
        runtime: custom
        timeout: 60
        caPort: 9000
```

```
    triggers:
      - name: httpTrigger
        type: http
        config:
          authType: anonymous
          methods:
            - GET
    customDomains:
      - domainName: auto
        protocol: HTTP
        routeConfigs:
          - path: '/*'
```

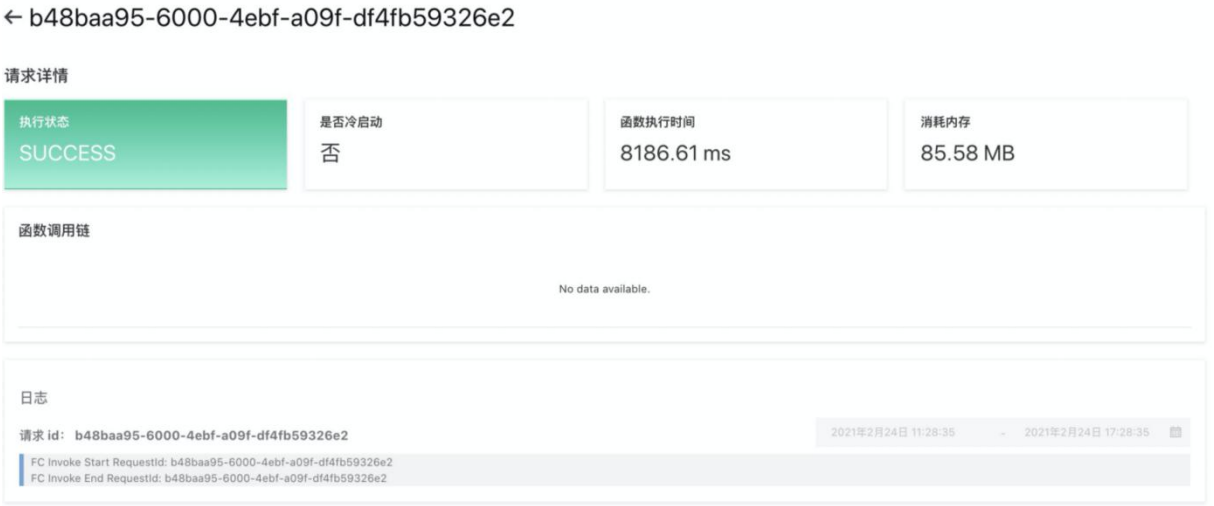
5.7 可观测性

Serverless 应用的可观测性是被很多用户所关注的。可观测性是通过外部表现判断系统内部状态的衡量方式，在应用开发中，可观测性帮助判断系统内部的健康状况。在系统出现问题时，帮助定位问题、排查问题、分析问题；在系统平稳运行时，帮助评估风险，预测可能出现的问题。在 Serverless 应用开发中，如果观察到函数的并发度持续升高，很可能是业务推广团队的努力工作导致业务规模迅速扩张，为了避免达到并发度限制触发流控，开发者就需要提前提升并发度，以阿里云函数计算为例，阿里云函数计算在可观测性层面提供了多种纬度，包括 Logging、Metrics 以及 Tracing 等内容。

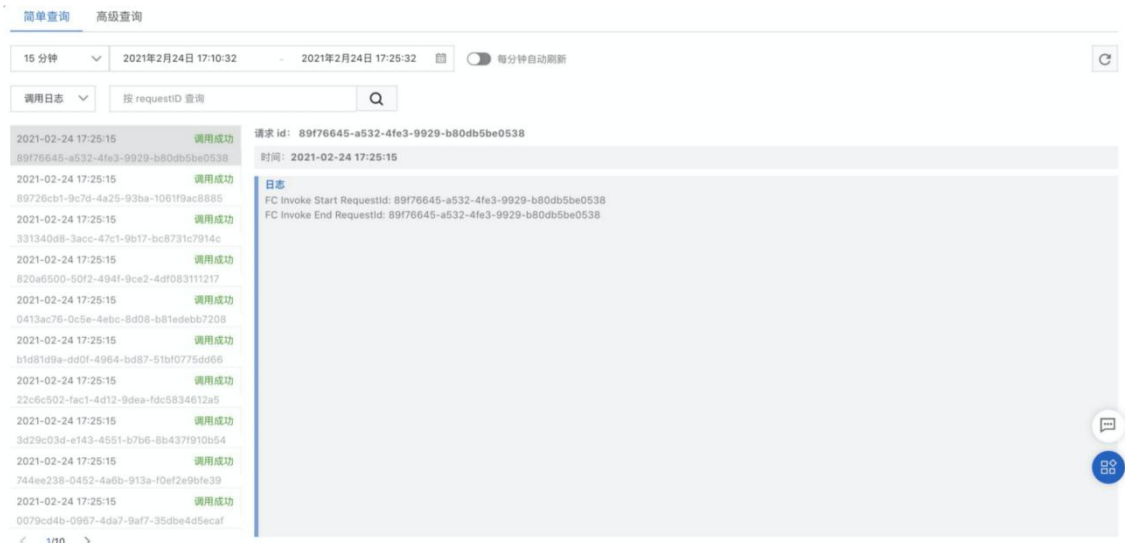
如图，在控制台监控中心，可以查看到整体的 Metrics，服务级 Metrics 以及每个函数的 Metrics。除此之外，还可以看到当前函数的请求记录：



根据不同的请求记录，可以查看到函数的详细信息：



除了在控制台的监控中心处可以查看到函数的日志等信息，在函数详情页面，也可以看到函数的详细日志信息：



以及 Tracing 相关信息：

Span名称	时间轴（ms）	应用名称	开始时间	IP地址	状态
InvokeFunction	1787.132ms	FC:TracingService/TracingFunction	2021-01-25 11:31:42.773	21.0.3.1	
ColdStart	519.764ms	FC:TracingService/TracingFunction	2021-01-25 11:31:43.975	21.0.3.1	
PrepareCode	17.494ms	FC:TracingService/TracingFunction	2021-01-25 11:31:43.975	21.0.3.1	
RuntimeInitialization	502.224ms	FC:TracingService/TracingFunction	2021-01-25 11:31:43.993	21.0.3.1	
Initialization	26.893ms	FC:TracingService/TracingFunction	2021-01-25 11:31:44.496	21.0.3.1	
Invocation	11.624ms	FC:TracingService/TracingFunction	2021-01-25 11:31:44.549	21.0.3.1	

当然，通过 Serverless Devs 开发者工具，以及函数计算组件也可以进行观测相关操作：

5.7.1 通过工具进行 Metrics 查看

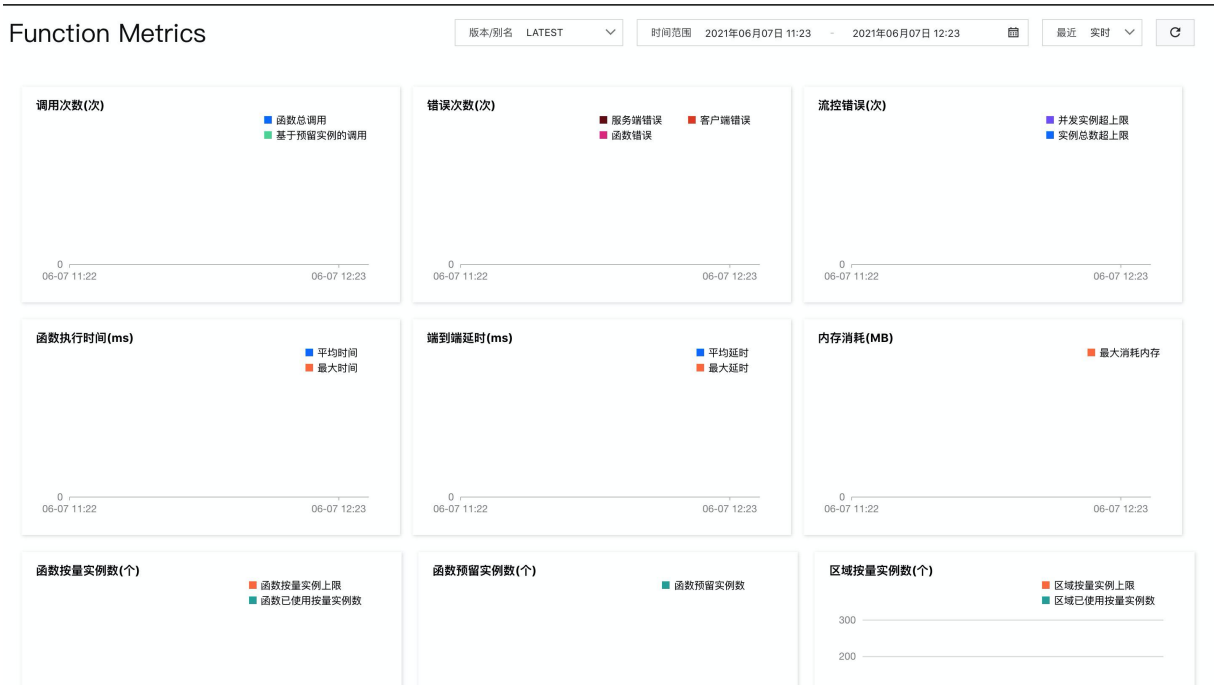
详情参考：https://github.com/devsapp/fc/blob/main/docs/zh/command/metrics.md

- **有资源描述文件（Yaml）时**，可以直接执行 s metrics 查看函数的指标信息；
- **纯命令行形式（在没有资源描述 Yaml 文件时）**，需要指定服务所在地区以及服务名称，函数名等，例如 s cli fc metrics --region ch-hangzhou --service-name myService --function-name myFunction；

上述命令的执行结果示例：

```
[2021-06-07T12:20:06.661] [INFO ] [FC-METRICS] - 请用浏览器访问 Uri 地址进行查看: http://localhost:3000
```

此时，通过浏览器打开地址，可以看到函数指标信息：



需要开启请求级别指标，才能查看函数指标信息，否则图表不展示数据。

如何开通请求级别指标:

1、https://fcnext.console.aliyun.com/

2、服务及函数中-找到自己 region-对应的服务名称-在操作栏点击**配置开启**请求级别指标

5.7.2 通过工具进行 Logs 查看

详情参考：https://github.com/devsapp/fc/blob/main/docs/zh/command/logs.md

- **有资源描述文件（Yaml）时**，可以直接执行 s logs 进行线上函数的日志查询；
- **纯命令行形式（在没有资源描述 Yaml 文件时）**，需要指定服务所在地区以及服务名称，函数名等，例如 s cli fc logs --region cn-hangzhou --service-name fc-deploy-service --function-name http-trigger-py36

上述命令的执行结果示例:

```
FunctionCompute python3 runtime inited.

FC Invoke Start RequestId: 84d6ae81-02ff-4011-b3ca-45e65b210cc3
FC Invoke End RequestId: 84d6ae81-02ff-4011-b3ca-45e65b210cc3

FC Invoke Start RequestId: de4812be-9137-4a33-9869-370cb61ac427
FC Invoke End RequestId: de4812be-9137-4a33-9869-370cb61ac427
```

如果需要以 tail 模式进行日志的查询，可以增加--tail 参数，例如 s logs --tail；

查询指定时间段的日志，可以通过增加--start-time 和--end-time 参数实现，例如 s logs -s 2021-11-04T15:40:00 -e 2021-11-04T15:45:00；

5.8 如何对应用进行调试

在应用开发过程中，或者应用开发完成，当所执行结果不符合预期时，通常要进行一定的调试工作。但是在 Serverless 加架构下，调试往往会受到极大的环境因素限制，通常会出现的情况是，所开发的应用在本地是可以比较健康的、符合预期的运行，但是在 FaaS 平台上，则会出现一些不可预测的问题；或者是在一些特殊的环境下，本地没有办法模拟线上环境，难以进行项目的开发和调试。

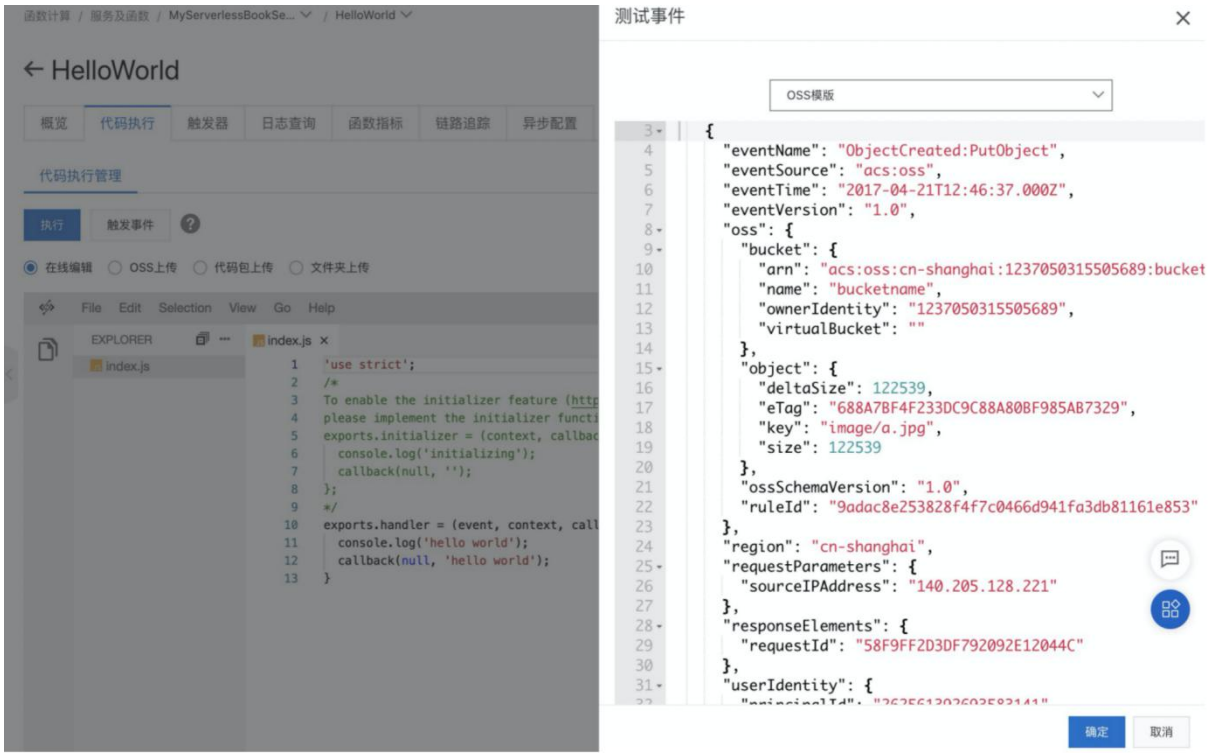
Serverless 应用的调试，一直都是备受诟病的，但是各个云厂商并没有因此放弃在调试方向的不断深入探索。以阿里云函数计算为例，其提供了在线调试、本地调试等多种调试方案。

5.8.1 在线调试

5.8.1.1 简单调试

所谓的简单地调试，就是在控制台来进行调试，以阿里云函数计算为例，可以在控制台通过执行按钮，进行基本的调试：

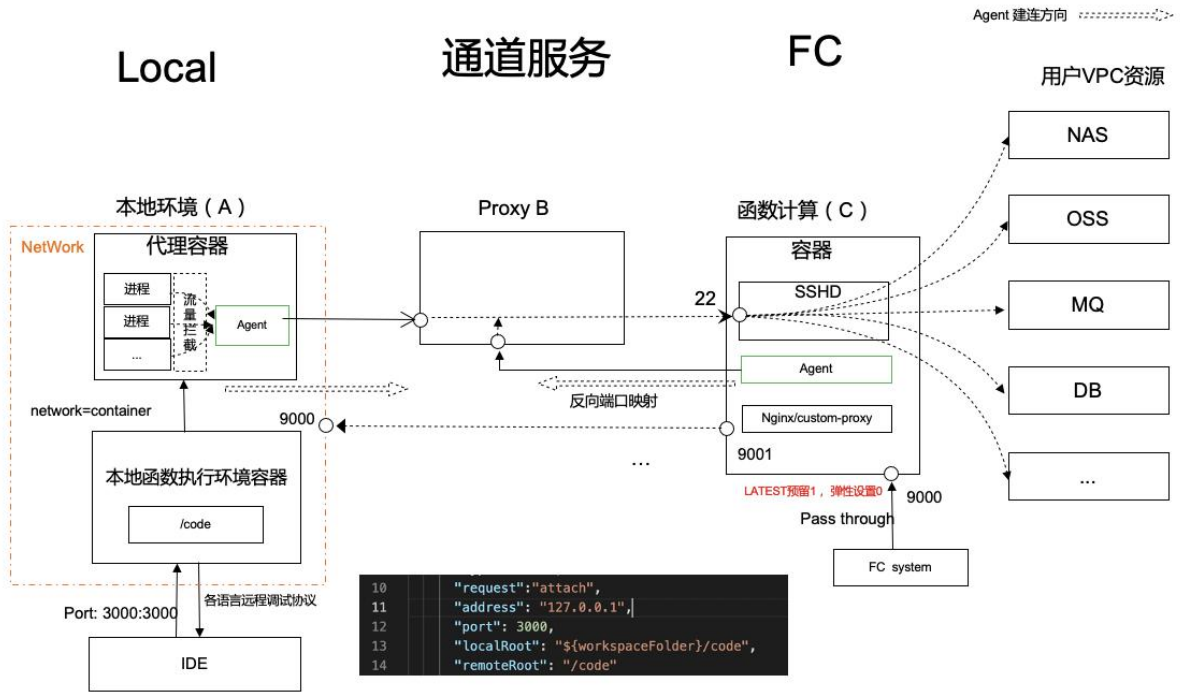
必要时候，也可以通过设置 Event 来模拟一些事件：



在线进行调试的好处是，可以使用线上的一些环境进行代码的测试。否则当线上环境拥有 VPC 等资源时，在本地环境是很难进行调试，例如数据库需要通过 VPC 来进行访问时，或者有对象存储触发器的业务逻辑等。

5.8.2 端云联调

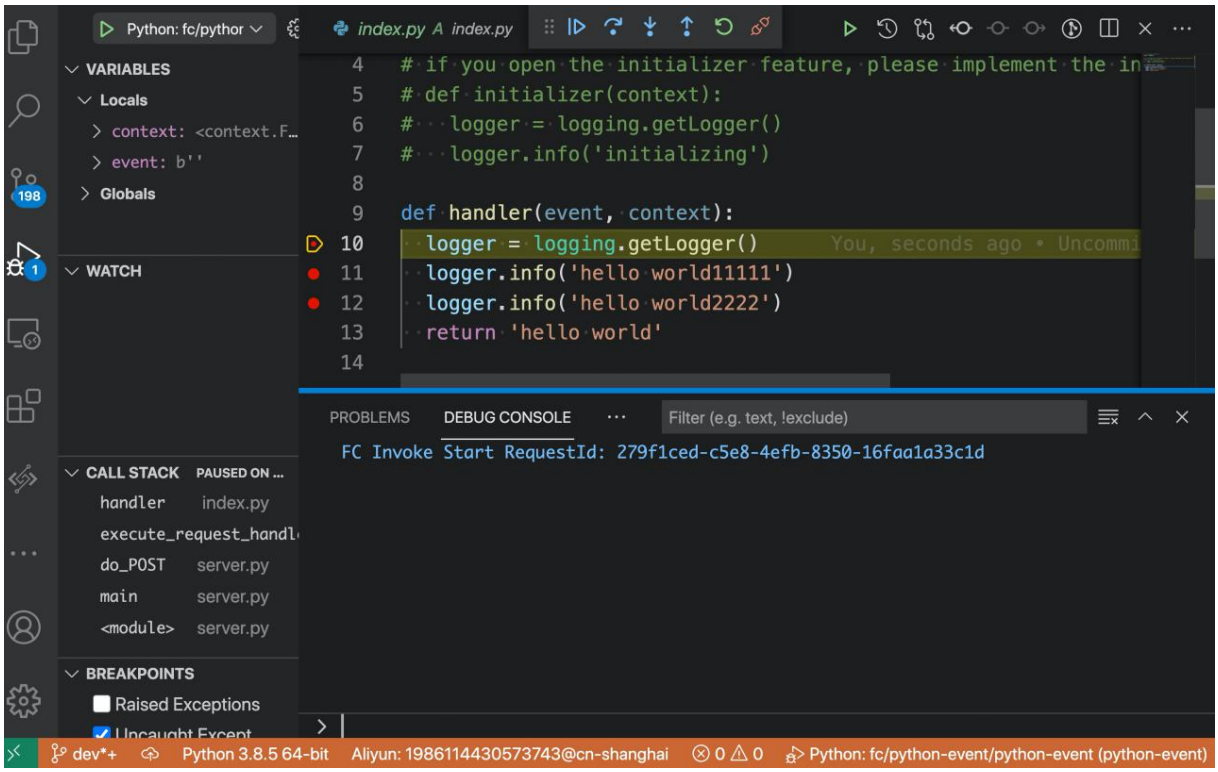
所谓的端云联调指的是在本地进行 Serverless 应用开发时，往往会涉及到一些线上资源，例如通过对象存储触发器触发函数执行，通过 VPC 访问数据库等，此时由于线上线下环境的不一致性，会让线下的开发、调试面临极大的挑战。Serverless Devs 团队通过搭建 Proxy 辅助函数的方法，将线上线下资源打通，可以快速帮助开发者在本地进行函数的开发与调试。



如图所示， Serverless Devs 开发者工具会根据开发者的函数的 yaml 文件配置，创建辅助服务和辅助函数（这个辅助服务和函数的配置跟您的服务和函数是相同的），并通过辅助服务和辅助函数实现线上线下的网络环境打通，以及完整的端云联调能力：

- 调用这个辅助函数，流量会打回到本地的调试实例， 这个时候本地实例接受到 event 和 context 是真实来自线上的；
- 本地调试的实例运行函数逻辑，能直接利用辅助函数运行的那个容器， 可以直接访问 vpc 内网以及一些云服务的内网地址；
- 具体的使用流程是：
- 执行 s proxied setup 来准备端云联调所需的辅助资源以及本地环境；
- 对于无触发器的普通事件函数或者 http 触发器函数，准备工作完成后，启动另一个新的终端，切换到该项目路径下，执行 s proxied invoke 来调用本地函数；
- 完成调试任务之后，可以执行 s proxied cleanup 清理端云联调所需的辅助资源以及本地环境；

除了通过命令行使用端云联调能力，也可以在 VSCode 等开发者工具中使用，例如：



5.8.2.1 最佳实践案例

三步完成端云联调

端云联调可以通过三个非常简单的步骤快速实现：

- 步骤 1: 在已有的项目下，创建端云联调的辅助资源，开启端云联调模式：s proxied setup；
- 步骤 2: 在完成端云联调模式开启动作之后，通过 s proxied invoke 或者线上的事件进行函数的触发，调试；
- 步骤 3: 完成端云联调之后，通过 s proxied clean 命令，对对因端云联调而产生的辅助资源进行清理；

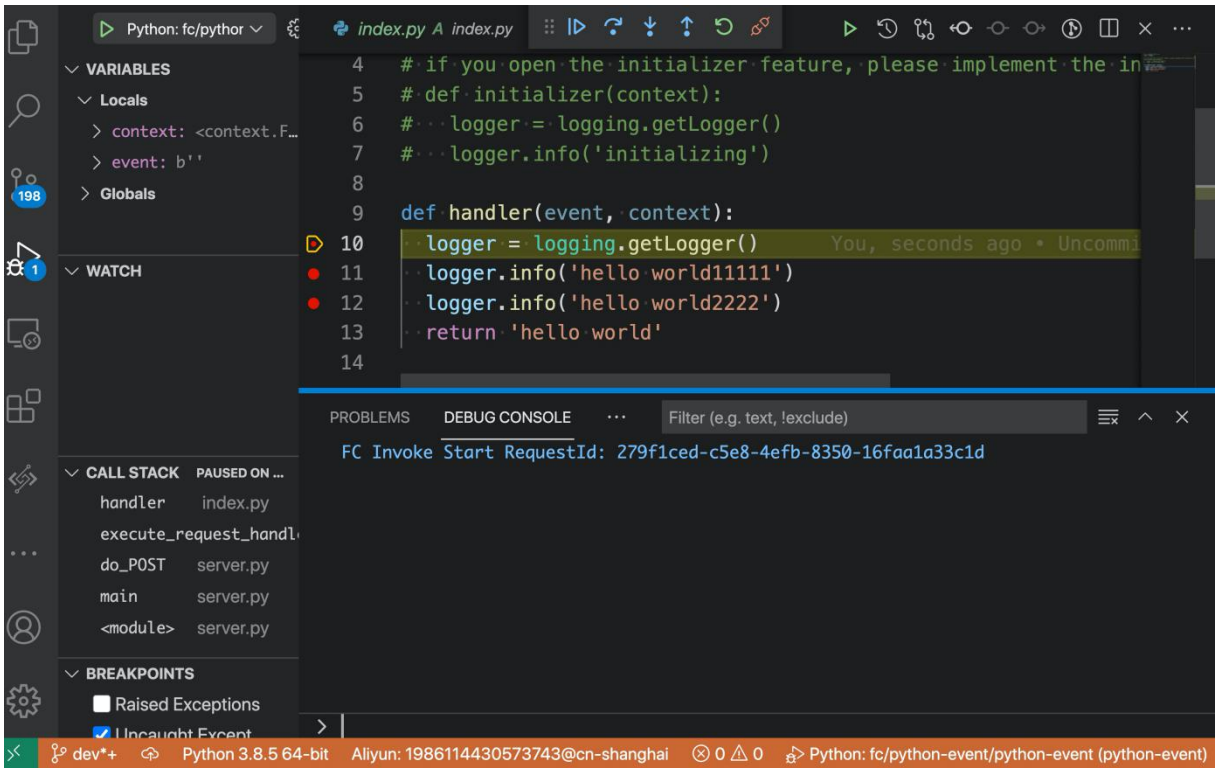
断点调试

通过与常见的 IDE 进行结合，可以在常见的 IDE 上实现端云联调的断点调试。

VSCode 断点调试案例

- 步骤 1: 在已有的项目下，开启调试模式的端云联调能力：$ s proxied setup --config vscode --debug-port 3000，命令执行完成功后， 本地的函数计算执行环境会阻塞等待调用(执行环境本质是一个 HTTP Server)；
  此时若要进行断点调试，需要进行以下的操作在 VSCode 上进行相关的配置：Serverless Devs 开发者工具自动在工程目录下面生成 .vscode/launch.json 文件，通过下图完成调试配置：

- 步骤 2: 打开一个新的终端，通过 proxied invoke 进行触发（例如 s proxied invoke，如果是事件函数也可以通过线上触发器进行触发，此时要注意将触发器临时指向辅助函数，详情参考 proxied invoke 命令操作过程），回到 VSCode 界面，既可以进行断点调试了：



调试完成后返回结果。

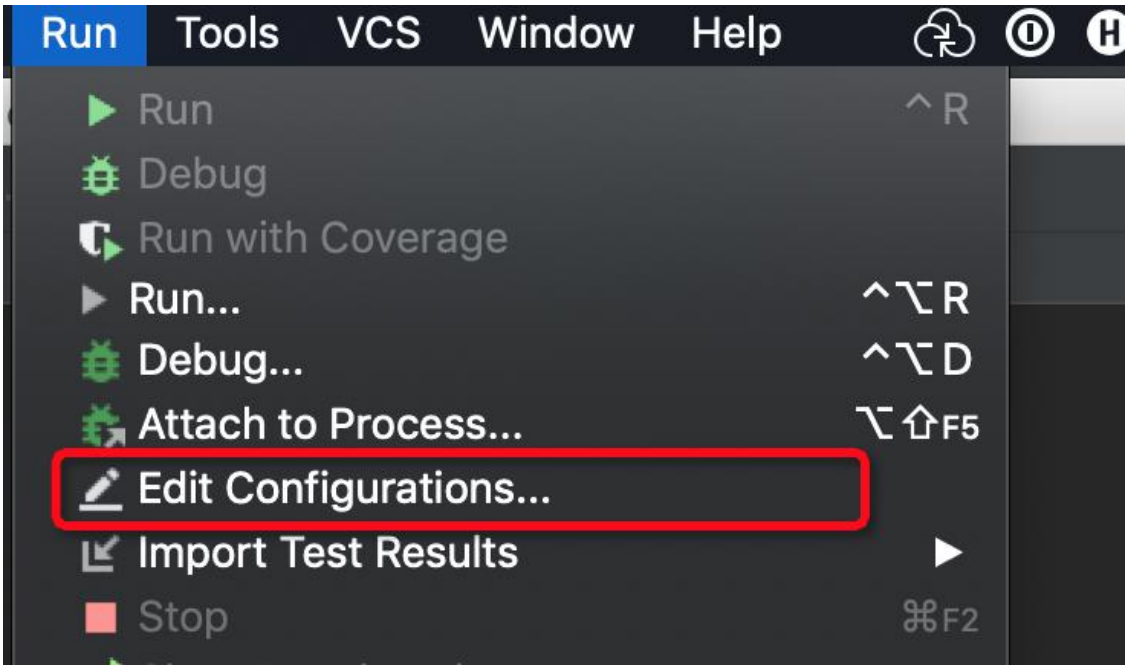
若要在调用的时候制定传入的 event 参数，可以使用 --event，例如 s proxied invoke -h

- 步骤 3: 完成端云联调之后，通过 s proxied clean 命令，对对因端云联调而产生的辅助资源进行清理；

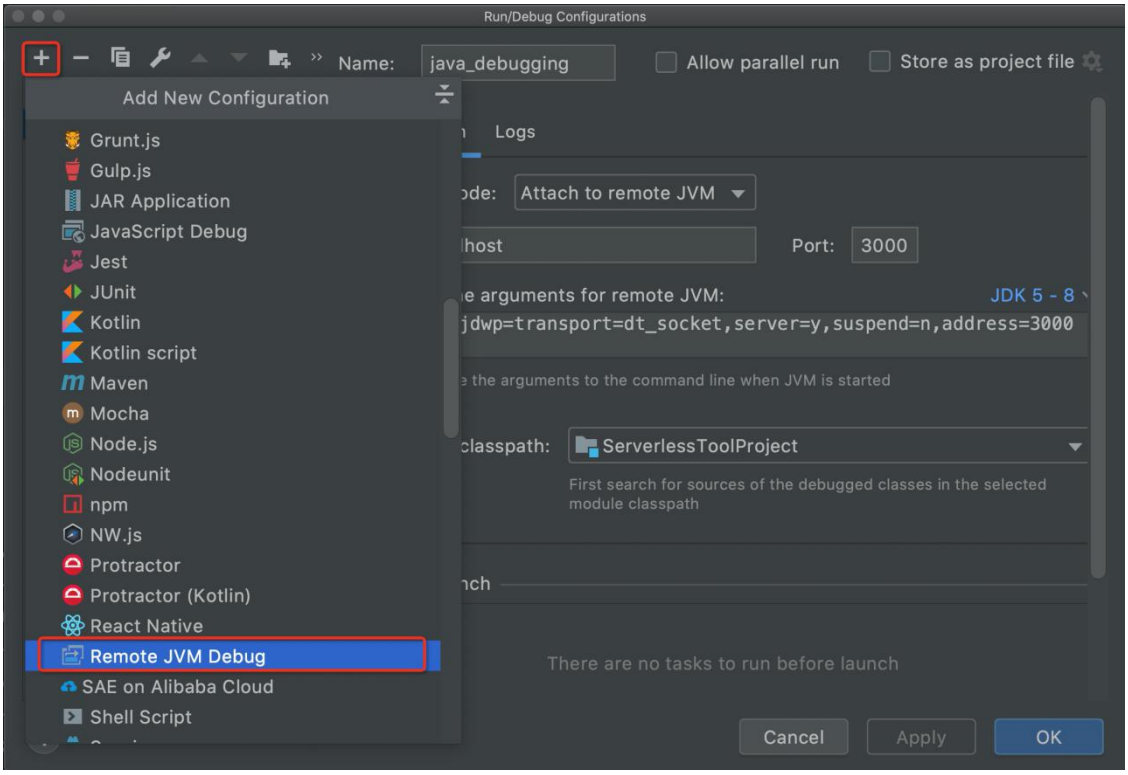
Intelli 断点调试案例

- 步骤 1: 例如需要在 IDEA 下进行调试，可以在已有的项目下，开启调试模式的端云联调能力：$ s proxied setup --config intellij --debug-port 3000，命令执行完成功后，本地的函数计算执行环境会阻塞等待调用(执行环境本质是一个 HTTP Server)；
  此时若要进行断点调试，需要进行以下的操作在 IDEA 上进行相关的配置：

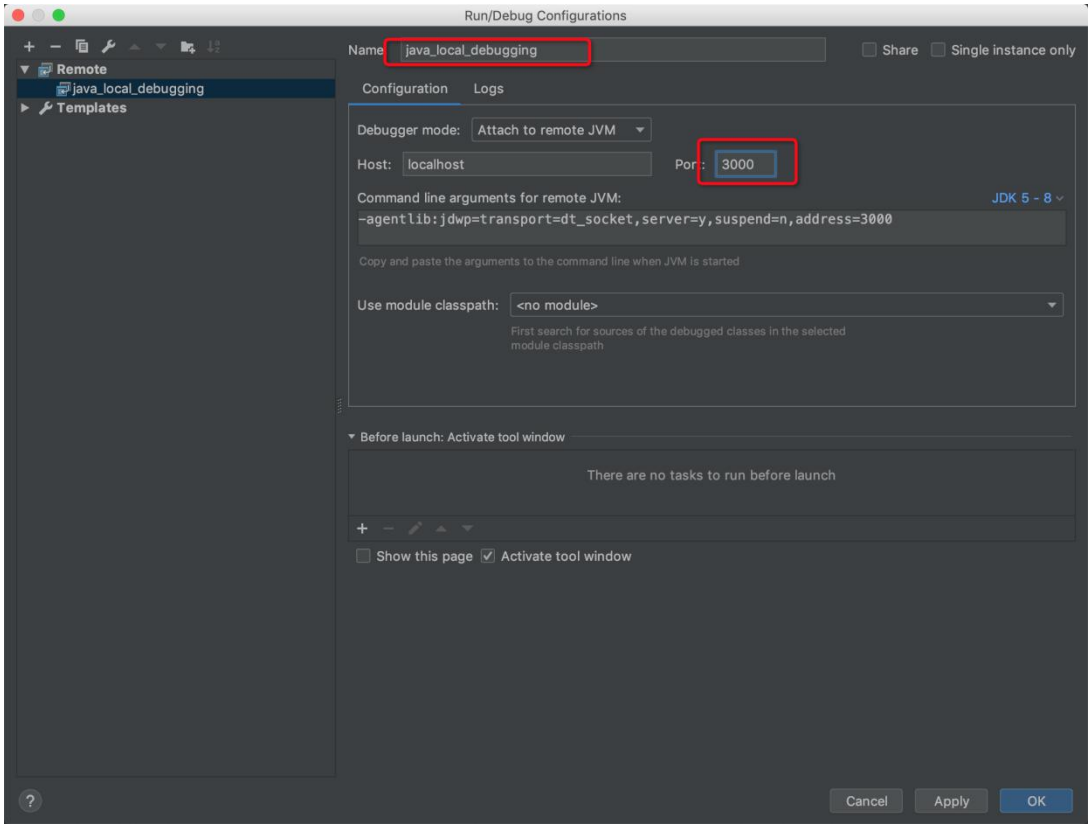
1、在菜单栏选择 Run… > Edit Configurations 。



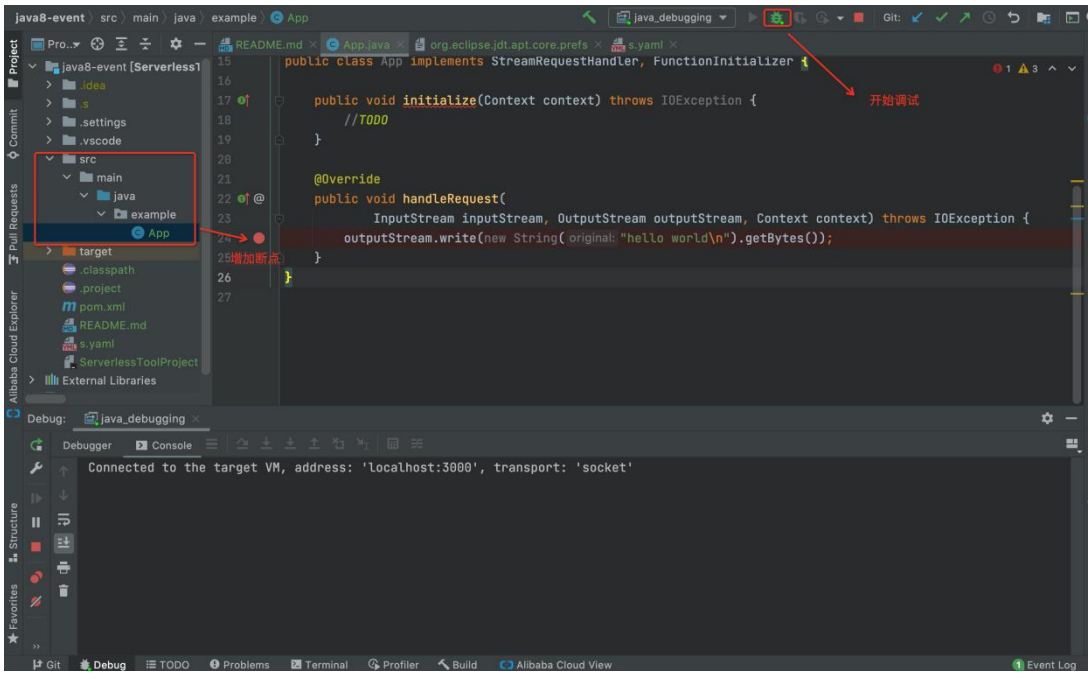
2、新建一个 Remote Debugging 。



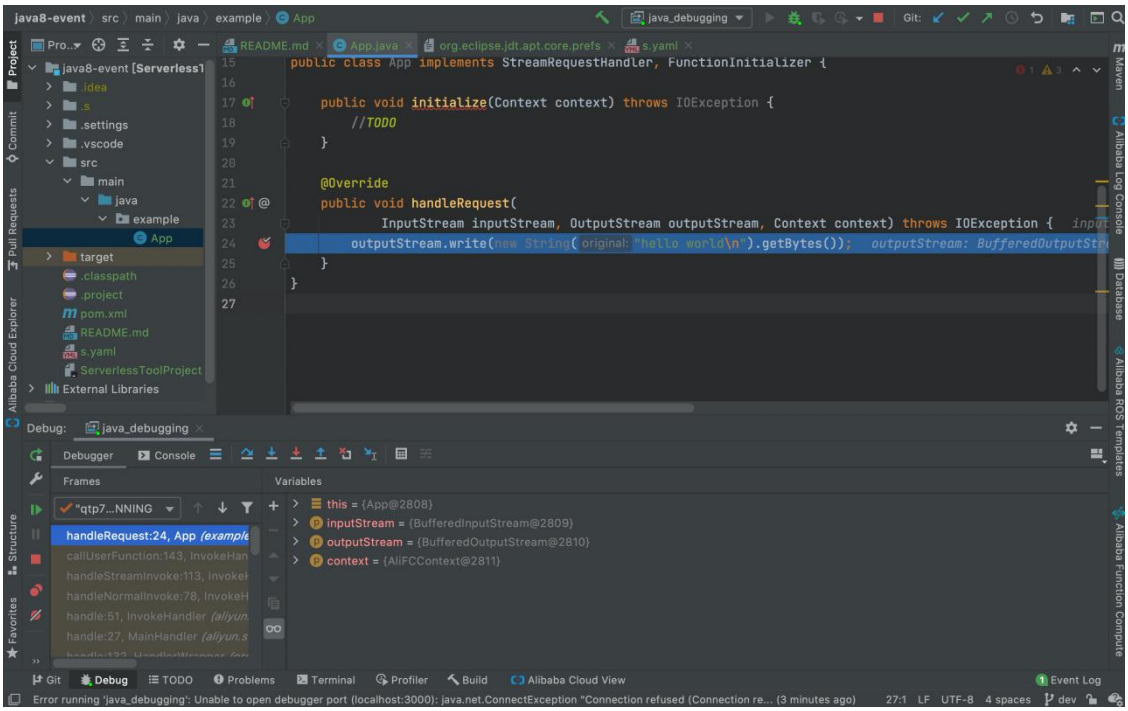
3、自定义调试器名称，并将端口配置为 3000 。



4、上述配置完成后，在 IDEA 编辑器侧边栏为函数代码增加断点，点击"开始调试"按钮。



- 步骤 2: 打开一个新的终端，通过 proxied invoke 进行触发（例如 s proxied invoke，如果是事件函数也可以通过线上触发器进行触发，此时要注意将触发器临时指向辅助函数，详情参考 proxied invoke 命令操作过程），回到 IDEA 界面，既可以进行断点调试了：



调试完成后返回结果。

若要在调用的时候制定传入的 event 参数，可以使用 --event，例如 s proxied invoke -h

- 步骤 3: 完成端云联调之后，通过 s proxied clean 命令，对对因端云联调而产生的辅助资源进行清理；

5.8.3 本地调试

5.8.3.1 命令行工具

就目前来看，大部分 FaaS 平台都会为用户提供相对完备的命令行工具，包括 AWS 的 SAM CLI，阿里云的 Funcraft，同时也有一些开源项目例如 Serverless Framework、Serverless Devs 等

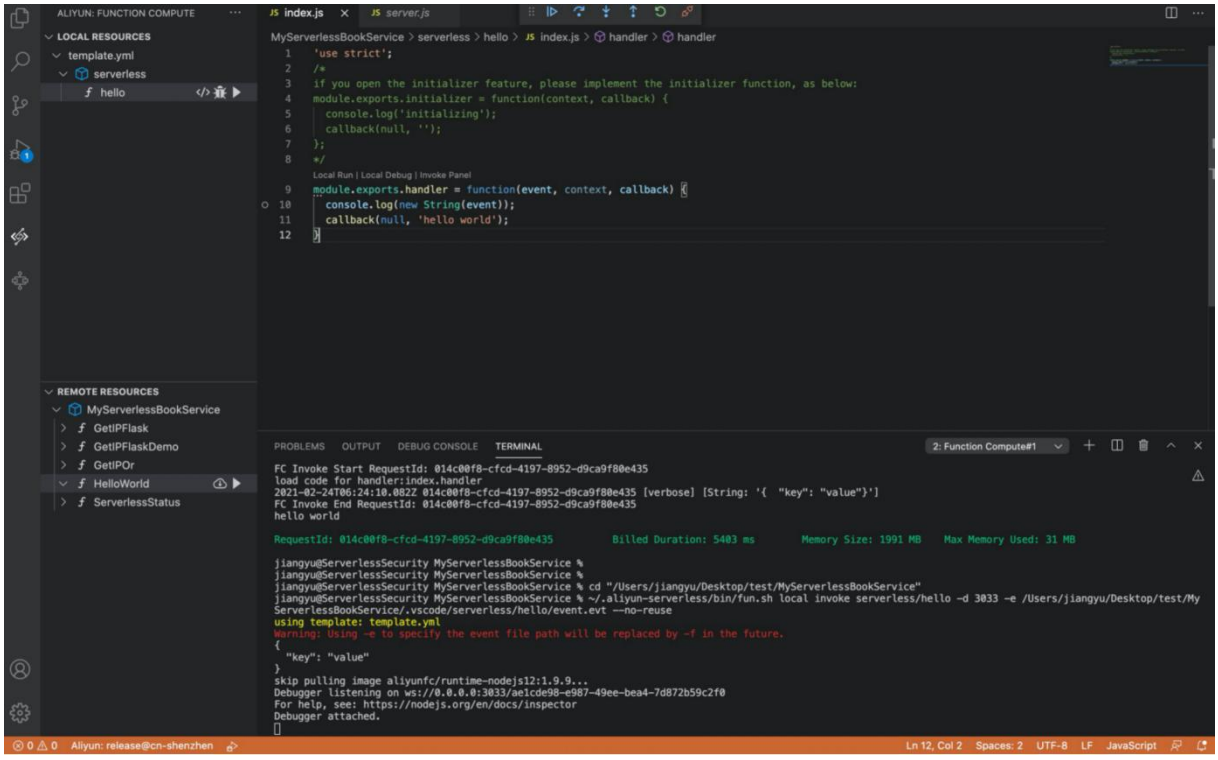
对多云厂商的支持等。通过命令行工具进行代码调试的方法很简单，以 Serverless Devs 为例，本地调试阿里云函数计算：

首先确保，本地拥有一个函数计算的项目，然后在项目下执行调试指令，例如在 Docker 中进行调试：

```
jiangyu@ServerlessSecurity python3 % s invoke docker

Start ...
It is detected that your project has the following project/projects < MyFunctionDemo > to be execute
Start executing project MyFunctionDemo
Skip pulling image aliyunfc/runtime-python3.6:1.9.6...
FunctionCompute python3 runtime inited.
FC Invoke Start RequestId: 3e744085-d0b1-47a4-b826-b984afab170a
2021-02-24T06:10:16.379Z 3e744085-d0b1-47a4-b826-b984afab170a [INFO] hello world
FC Invoke End RequestId: 3e744085-d0b1-47a4-b826-b984afab170a
hello world

RequestId: 3e744085-d0b1-47a4-b826-b984afab170a          Billed Duration: 89 ms          Memory Size: 1991 MB    Max Memory Used:
41 MB

Project MyFunctionDemo successfully to execute

End of method: invoke
```

5.8.4.2 编辑器插件

以 VScode 插件为例，当下载好阿里云函数计算的 VSCode 插件，并且配置好账号信息之后，可以在本地新建函数，并且在进行打点，之后可以进行断点调试：



当函数调试完成之后，可以进行部署等操作。

5.8.4 其他调试方案

5.8.4.1 Web 框架的本地调试

在阿里云 FaaS 平台开发传统 Web 框架，以 Python 语言 Bottle 框架为例，可以增加：

```
app = bottle.default_app()
```

并且对 run()方法进行条件限制(if __name__ == '__main__')：

```
if __name__ == '__main__':
    bottle.run(host='localhost', port=8080, debug=True)
```

例如：

```
# index.py
import bottle

@bottle.route('/hello/<name>')
def index(name):
    return "Hello world"

app = bottle.default_app()

if __name__ == '__main__':
    bottle.run(host='localhost', port=8080, debug=True)
```

这样，可以在本地开发的同时，和传统开发思路一样，在本地进行调试。当部署到线上时，只需要在入口方法处填写 index.app，即可实现平滑的部署。

5.8.4.2 本地模拟事件调试

针对非 Web 框架，可以在本地构建一个方法，例如要调试一下对象存储触发器，可以是：

```
import json

def handler(event, context):
    print(event)

def test():
    event = {
        "events": [
            {
                "eventName": "ObjectCreated:PutObject",
                "eventSource": "acs:oss",
                "eventTime": "2017-04-21T12:46:37.000Z",
                "eventVersion": "1.0",
                "oss": {
                    "bucket": {
                        "arn": "acs:oss:cn-shanghai:123456789:bucketname",
                        "name": "testbucket",
                        "ownerIdentity": "123456789",
                        "virtualBucket": ""
                    },
                    "object": {
                        "deltaSize": 122539,
                        "eTag": "688A7BF4F233DC9C88A80BF985AB7329",
                        "key": "image/a.jpg",
```

```
                        "size": 122539
                    },
                    "ossSchemaVersion": "1.0",
                    "ruleId": "9adac8e253828f4f7c0466d941fa3db81161****"
                },
                "region": "cn-shanghai",
                "requestParameters": {
                    "sourceIPAddress": "140.205.***.***"
                },
                "responseElements": {
                    "requestId": "58F9FF2D3DF792092E12044C"
                },
                "userIdentity": {
                    "principalId": "123456789"
                }
            }
        ]
    }
    handler(json.dumps(event), None)

if __name__ == "__main__":
    print(test())
```

这样通过构造一个 event 对象，即可实现模拟事件触发。

附录

- Serverless Devs：
  https://github.com/Serverless-Devs
- Serverless Devs 阿里云密钥：
  http://www.serverless-devs.com/zh-cn/docs/provider-config/alibabacloud.html
- Serverless Devs 安装文档：
  http://serverless-devs.com/zhcn/docs/installed/cliinstall.html
- ServerlessDesktop：
  http://www.serverless-devs.com/zh-cn/desktop/index.html
- Serverless Devs Cli：
  http://www.serverless-devs.com/zh-cn/cli/index.html
- Serverless Hub：
  https://serverlesshub.resume.net.cn/#/hubs/specialview
- 项目仓库：
  https://github.com/Serverless-Devs/Serverless-Devs
- Serverless 应用开发者套件：
  http://serverlessdk.oss.devsapp.net/docs/tutorial-dk/intro/react
- 云解析 DNS：
  https://wanwang.aliyun.com/domain/dns
- 函数计算：
  https://www.aliyun.com/product/fc
- API 网关：
  https://www.aliyun.com/product/apigateway
- 对象存储 OSS：
  https://www.aliyun.com/product/oss
- 阿里云日志服务：
  https://help.aliyun.com/product/28958.html
- Serverless 应用引擎：
  https://www.aliyun.com/product/sae

- SDK 列表：
  https://help.aliyun.com/document_detail/53277.html
- Serverless Devs 端云联调：
  https://github.com/devsapp/fc/blob/main/docs/zh/command/proxied.md
- Serverless Dves FC：
  https://github.com/devsapp/fc
- 构建与部署相关：
  - 部署 deploy ：https://github.com/devsapp/fc/blob/main/docs/zh/command/deploy.md
  - 构建 build：https://github.com/devsapp/fc/blob/main/docs/zh/command/build.md
  - 移除 remove: https://github.com/devsapp/fc/blob/main/docs/zh/command/remove.md
  - 计划变更 plan：https://github.com/devsapp/fc/blob/main/docs/zh/command/plan.md
- 可观测性相关：
  - 指标查询 metrics：https://github.com/devsapp/fc/blob/main/docs/zh/command/metrics.md
  - 日志查询 logs：https://github.com/devsapp/fc/blob/main/docs/zh/command/logs.md
- 调用与调试相关：
  - 本地调用 local：https://github.com/devsapp/fc/blob/main/docs/zh/command/local.md
  - 函数触发 invoke：https://github.com/devsapp/fc/blob/main/docs/zh/command/invoke.md
  - 端云联调 proxied：https://github.com/devsapp/fc/blob/main/docs/zh/command/proxied.md
  - 远程调试 instance：https://github.com/devsapp/fc/blob/main/docs/zh/command/instance.md
  - 内存&并发度探测 eval：https://github.com/devsapp/fc/blob/main/docs/zh/command/eval.md
- 发布与配置相关：
  - 版本 version：https://github.com/devsapp/fc/blob/main/docs/zh/command/version.md

  - 别名 alias：https://github.com/devsapp/fc/blob/main/docs/zh/command/alias.md
  - 预留 provision：https://github.com/devsapp/fc/blob/main/docs/zh/command/provision.md
  - 按量资源 ondemand：https://github.com/devsapp/fc/blob/main/docs/zh/command/ondemands.md
  - 层 layer：https://github.com/devsapp/fc/blob/main/docs/zh/command/layer.md
- 其他功能：
  - 硬盘挂载 nas：https://github.com/devsapp/fc/blob/main/docs/zh/command/nas.md
  - Fun 项目迁移 fun2s：https://github.com/devsapp/fc/blob/main/docs/zh/command/fun2s.md
  - 查看函数 info：https://github.com/devsapp/fc/blob/main/docs/zh/command/info.md
  - 资源同步 sync：https://github.com/devsapp/fc/blob/main/docs/zh/command/sync.md
  - 压测 stress：https://github.com/devsapp/fc/blob/main/docs/zh/command/stress.md
  - API 调用 api：https://github.com/devsapp/fc/blob/main/docs/zh/command/api.

拓展链接：

- 通过 Serverless 架构实现监控告警能力：
  https://github.com/Serverless-Book-Anycodes/Website-Monitor
- 钉钉/ 企业微信机器人：“GitHub 触发器”与“Issue 机器人”
  https://github.com/Serverless-Book-Anycodes/Github-Issue-Robot
- 触发器和函数赋能自动化运维：
  https://github.com/Serverless-Book-Anycodes/DevOps
- Serverless CI/CD 实践案例：
  https://github.com/Serverless-Book-Anycodes/Serverless-CICD
- Serverless 架构下的图片压缩与水印：
  https://github.com/Serverless-Book-Anycodes/Image-Compress-WaterMark
- Serverless 架构下的音视频处理：
  https://github.com/Serverless-Book-Anycodes/fc-oss-fImpeg
  https://github.com/Serverless-Book-Anycodes/fc-fnf-video-processing

- Serverless：让图像合成更简单
  https://github.com/Serverless-Book-Anycodes/Image-Synthesis
  https://github.com/Serverless-Book-Anycodes/Add-Christmas-Hat
- ImageAI 与图像识别：
  https://github.com/Serverless-Book-Anycodes/Image-Prediction
- Serverless 与 NLP：让我们的博客更有趣
  https://github.com/Serverless-Book-Anycodes/NLP-Text-Summary
  https://github.com/Serverless-Book-Anycodes/NLP-Poem
- 基于 Serverless 架构的验证码识别功能：
  https://github.com/Serverless-Book-Anycodes/Image-Captcha
- 两数计算与对象存储实现 WordCount：
  https://github.com/Serverless-Book-Anycodes/BigData-MapReduce
- Serverless 与 Websocket 的聊天工具：
  https://github.com/Serverless-Book-Anycodes/Websocket-ChatApp
- Serverless 与 IOT：为智能音箱赋能
  https://github.com/Serverless-Book-Anycodes/IoT-SmartSpeaker
- 用手机写代码：基于 Serverless 的在线编程能力探索
  https://github.com/Serverless-Book-Anycodes/Online-Programming
- 基于 Serverless 架构的博客系统：
  https://github.com/Serverless-Book-Anycodes/Serverless-Blog-Django
  https://github.com/Serverless-Book-Anycodes/Serverless-Blog-Zblog
- 基于 Serverless 架构的人工智能相册小程序：
  https://github.com/Serverless-Book-Anycodes/AIAlbum

关注阿里云官方
Serverless微信公众号

Serverless开发者技术
学习钉钉群