结束语-带你整体回顾我们的Serverless案例

你好，我是秦粤。在经过了11节课的学习后，相信此刻，你对Serverless一定有了一些新的认识。那到了尾声，今天这节课我们就结合“待办任务”Web服务的演进过程，带你整体回顾一下本专栏的内容，希望能对你自身沉淀知识有所助益。

一路认真学习并动手实践课后作业的同学其实很容易发现，这个专栏并不是教大家写代码的，而是一堂服务端技术架构课。我们的实践内容和作业，主要也是让你通过部署项目代码体验一下运维的工作，更深刻地理解**“Serverless是对服务端运维的极端抽象”**这句话。

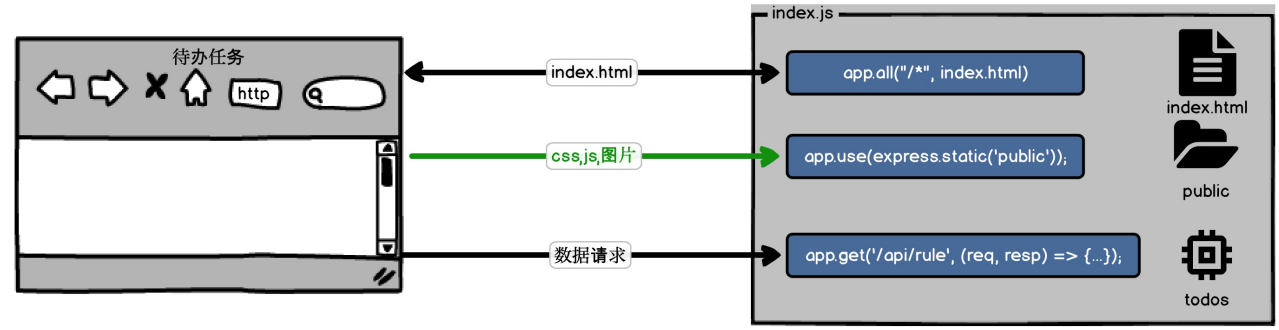
下面我们就分几个阶段去回顾“待办任务”Web服务这个大案例。

“待办任务”Web服务

我们的代码都在GitHub上，我建议你一定要跟着我的节奏run一下。

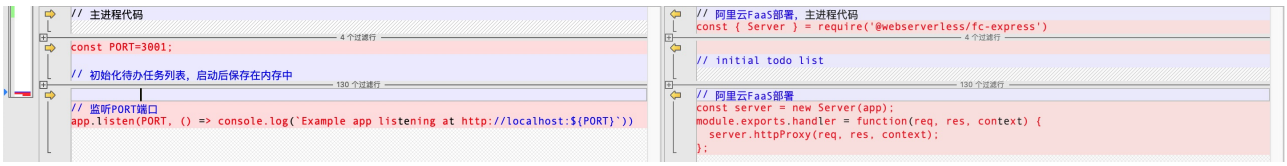
All-in-one

第一个版本master分支，以下是这个版本的示意图。



你可以看到这个master分支的版本，采用的是Express.js框架，这是一个典型的MVC架构。而且所有的请求，无论index.html、数据API请求，还是静态资源，都放在了一个文件index.js中处理。

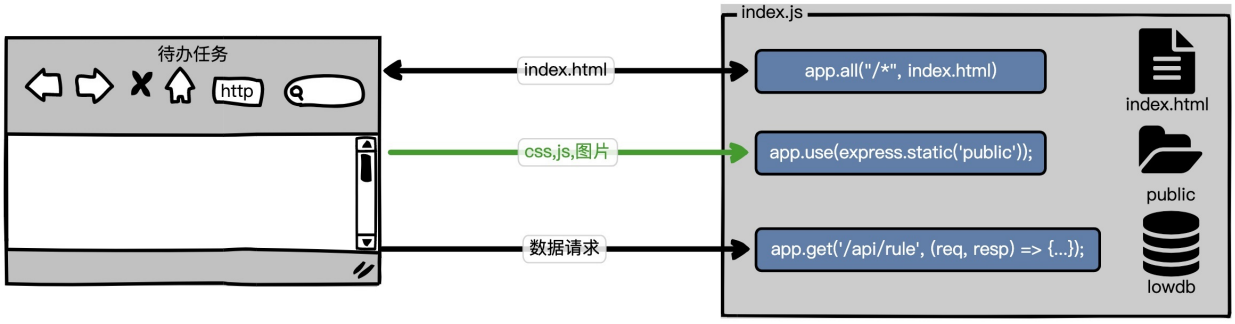
这里我特意给出了2个文件：index.js和index-faas.js。index.js是用于本地开发和调试的，而index-faas.js是用于部署到阿里云函数服务的。我们可以对比一下，其实不难发现这2个文件只有细微的差别。



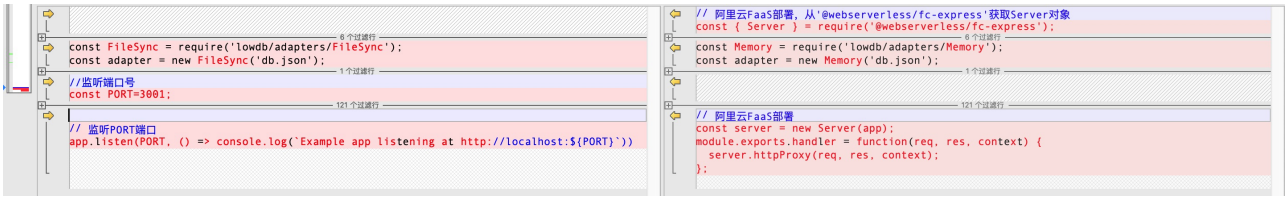
index.js因为我们在本地调试，所以它需要在我们本地IP上监听一个端口号：3001。通过端口号将用户的HTTP请求转入到我们上面注册的Express函数中。

index-faas.js因为部署在云上的函数服务中，它是通过事件触发的，因此它从函数服务提供的Runtime中获取了fc-express库的Server对象，这个Server对象其实是一个适配器，它将函数服务HTTP事件对象适配成Express的request、response和context对象，这样我们的其他代码就可以复用index.js了。

“待办任务”Web服务，第二个版本是lesson04-homework分支，以下是这个版本的示意图。



这个版本也是index.js接收所有的请求，不同于上一个版本的是，待办任务的数据我们存储在了lowdb仓库中。我们对比一下index.js和index-faas.js。

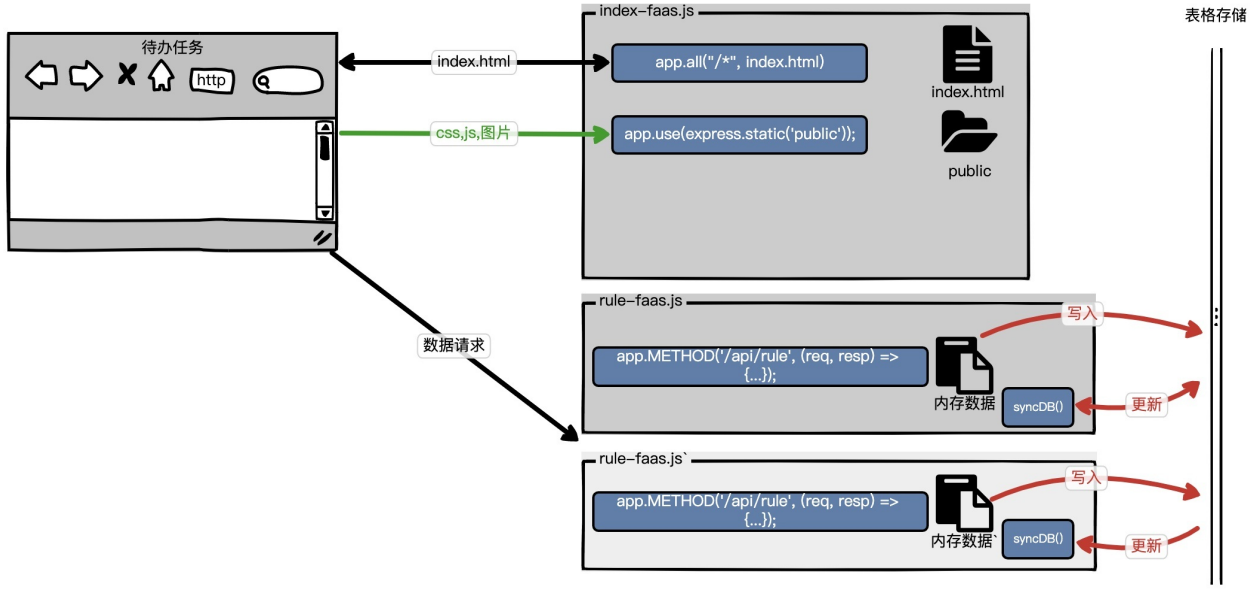


```
const FileSync = require('lowdb/adapters/FileSync');
const adapter = new FileSync('db.json');

//监听端口号
const PORT=3001;

// 监听PORT端口
app.listen(PORT, () => console.log(`Example app listening at http://localhost:${PORT}`))
```

```
// 阿里云FaaS部署，从'@webserverless/fc-express'获取Server对象
const { Server } = require('@webserverless/fc-express');
const Memory = require('lowdb/adapters/Memory');
const adapter = new Memory('db.json');

// 阿里云FaaS部署
const server = new Server(app);
module.exports.handler = function(req, res, context) {
  server.httpProxy(req, res, context);
};
```

可以看出我们在本地能利用机器的硬盘持久化Todos数据，但是在函数服务上却不行，就像我们专栏中介绍的FaaS实例要求我们无状态Stateless，因为我们的函数实例每次启动都是一个全新的机器，然后加载我们的代码。你如果尝试在函数服务上使用index.js的写文件的方式，将数据放入db.json，你就会发现我们的函数实例的机器硬盘是只读的。我需要指明一下，即使我们可以读写/tmp目录，但每次启动的函数实例都是无法保存状态的。

“待办任务”Web服务，第三个版本是lesson05分支，以下是这个版本的示意图。

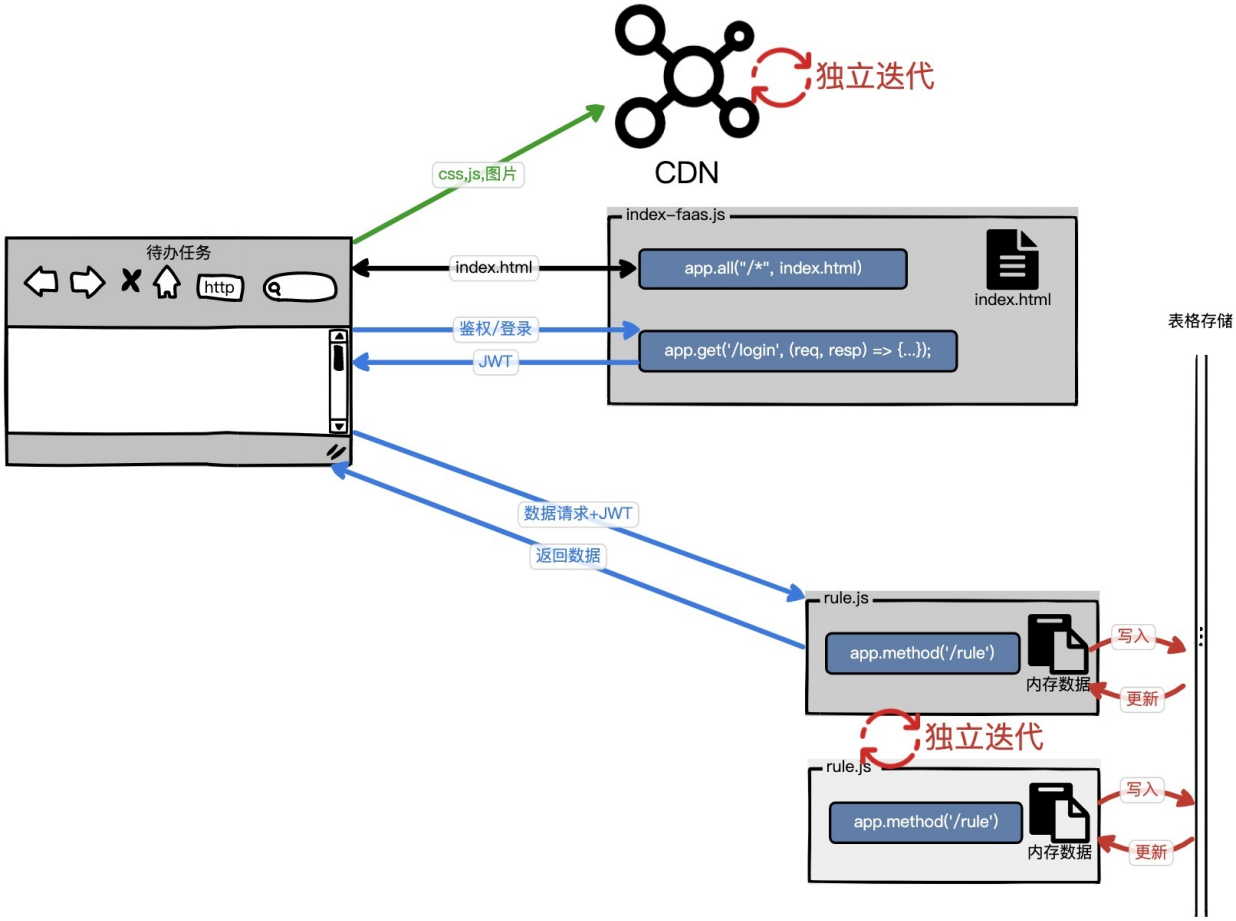


为了解决前面lesson04中的函数服务的数据持久化问题，我们引入了消息队列。当然我实际代码中是采用了表格存储OTS，而且我们将数据库操作的rule抽离出来，放在了一个独立的文件rule-faas中，rule-faas提供RESTful的HTTP API给客户端访问。这样做是为了index-faas和rule-faas独立部署，避免它们的逻辑耦合在一起，当触发扩缩容时造成资源浪费。例如用户的前端资源请求量大时，也会触发实例扩容，如果rule-faas没有独立部署，就会导致额外的数据库对象创建。

这种All in one的做法最简单直接，所有的内容都放在一起，部署和调试都很方便。而且我们本地开发和云端的差异很小，方便我们去验证、测试函数服务的一些功能。

静态资源分离

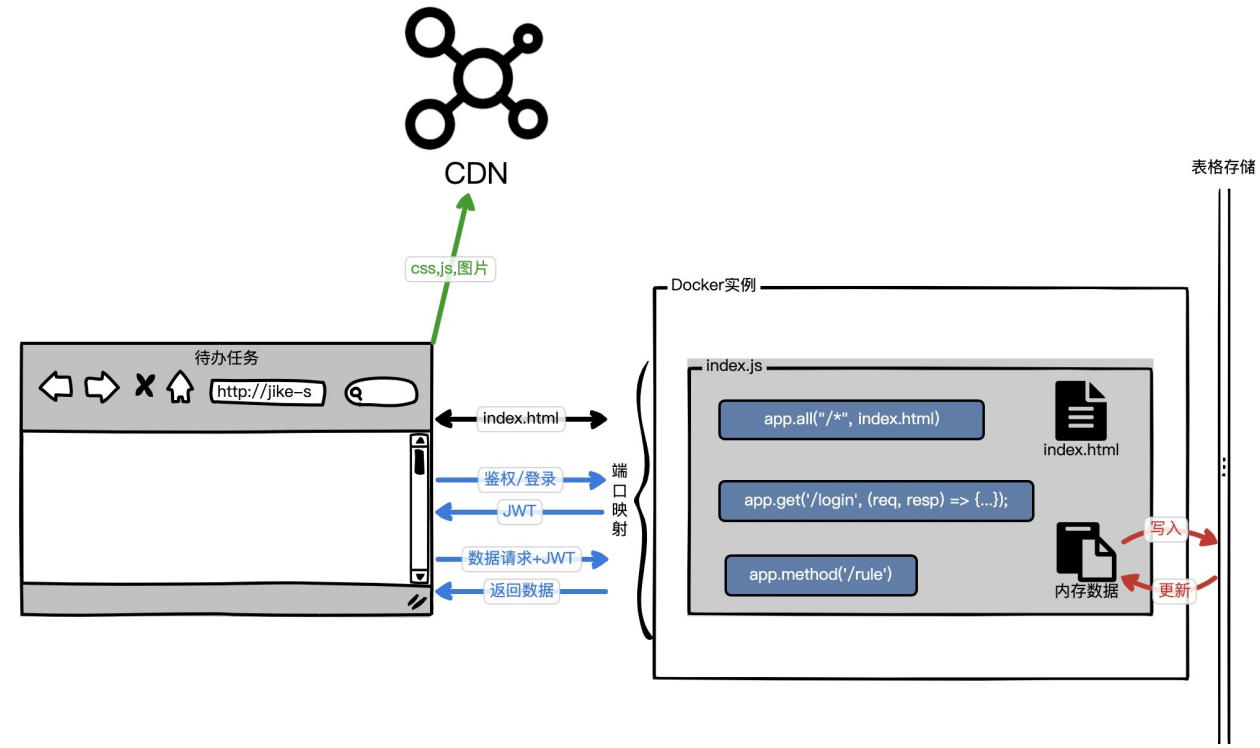
“待办任务”Web服务，第四个版本是lesson06分支，以下是这个版本的示意图。



这个版本和上个版本相比，最大的改变是，我们将静态资源从public中移出，部署到CDN上了。另外为了增加安全性，避免我们的rule服务直接被HTTP请求篡改，我们引入了微服务概念中的JWT。用户需要先登录(/api/currentUser)，Cookie获取到JWT，才能顺利通过rule.js的JWT token校验。

Docker容器

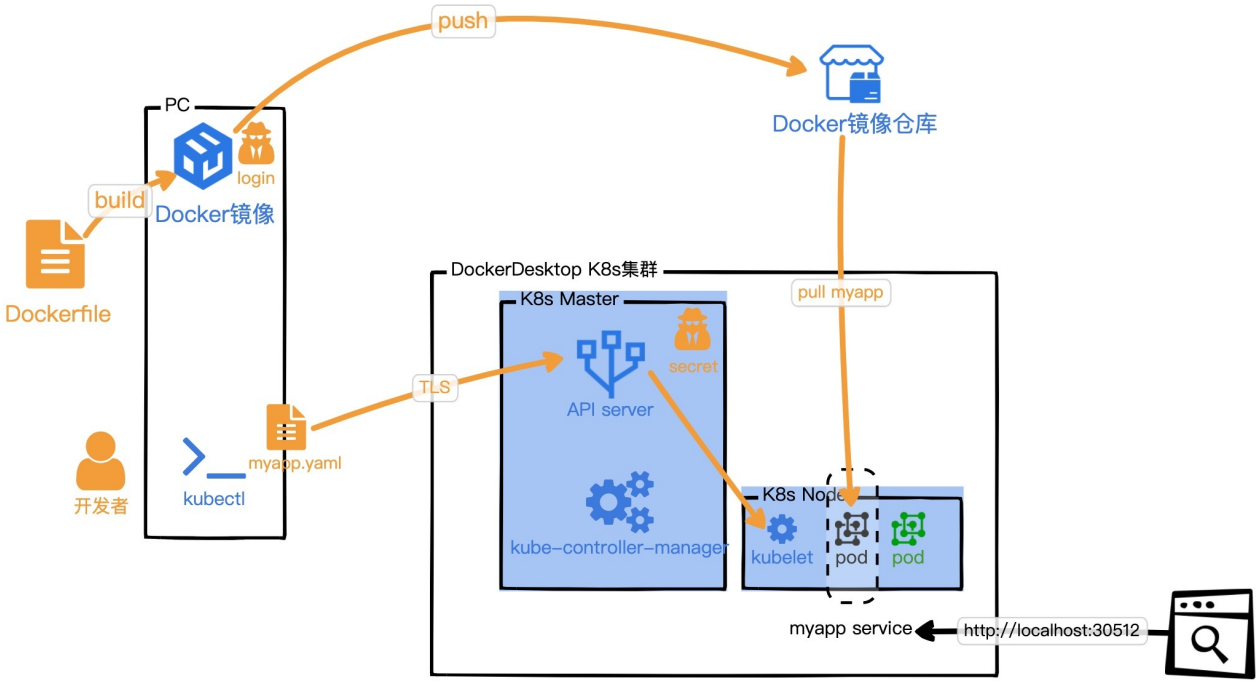
“待办任务”Web服务，第五个版本是lesson07分支，以下是这个版本的示意图。



这个版本和上一个版本相比，区别就是将所有的内容都放回到index.js，并且运行在容器中了。然后我们在代码中引入Dockerfile，构建我们的第一个容器镜像。这里我啰嗦一下，为了简化我们这节课的体验，我将rule.js合并到index.js中了。这样做其实也是动态化网络的思想，我们没必要为了划分微服务而去划分微服务。在实际工作中，合并、拆解节点应该是常见的操作，具体根据我们的业务需求来就行。

Kubernetes

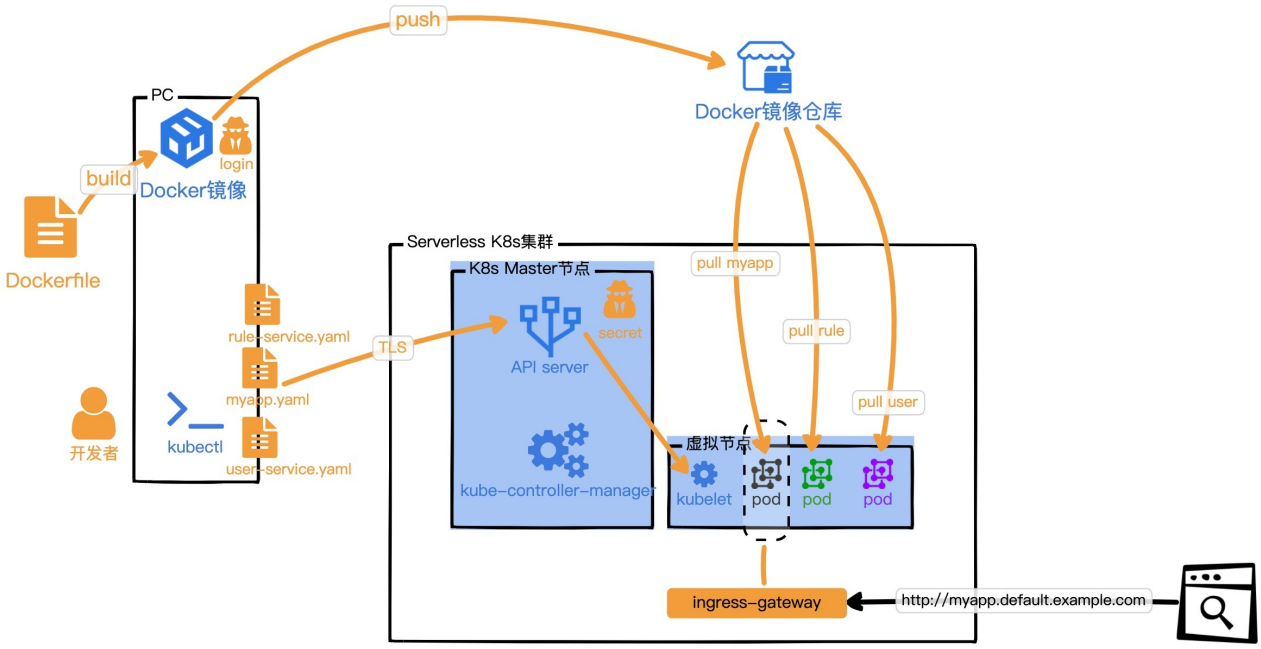
“待办任务”Web服务，第六个版本是lesson08分支，以下是这个版本的示意图。



为了更好地管理Docker容器，我们这个版本引入了K8s。通过上面的示意图我们可以看出，K8s通过策略配置或指令将我们的“待办任务”Web服务的生命周期管理了起来。我们应用开发者除了关心Dockerfile，不用再关心我们的容器重启、奔溃、扩缩容等等问题了，但我们应用在K8s集群的运维状态，还是需要运维人员手动维护的。

Knative

“待办任务”Web服务，第七个版本是lesson09分支，以下是这个版本的示意图。

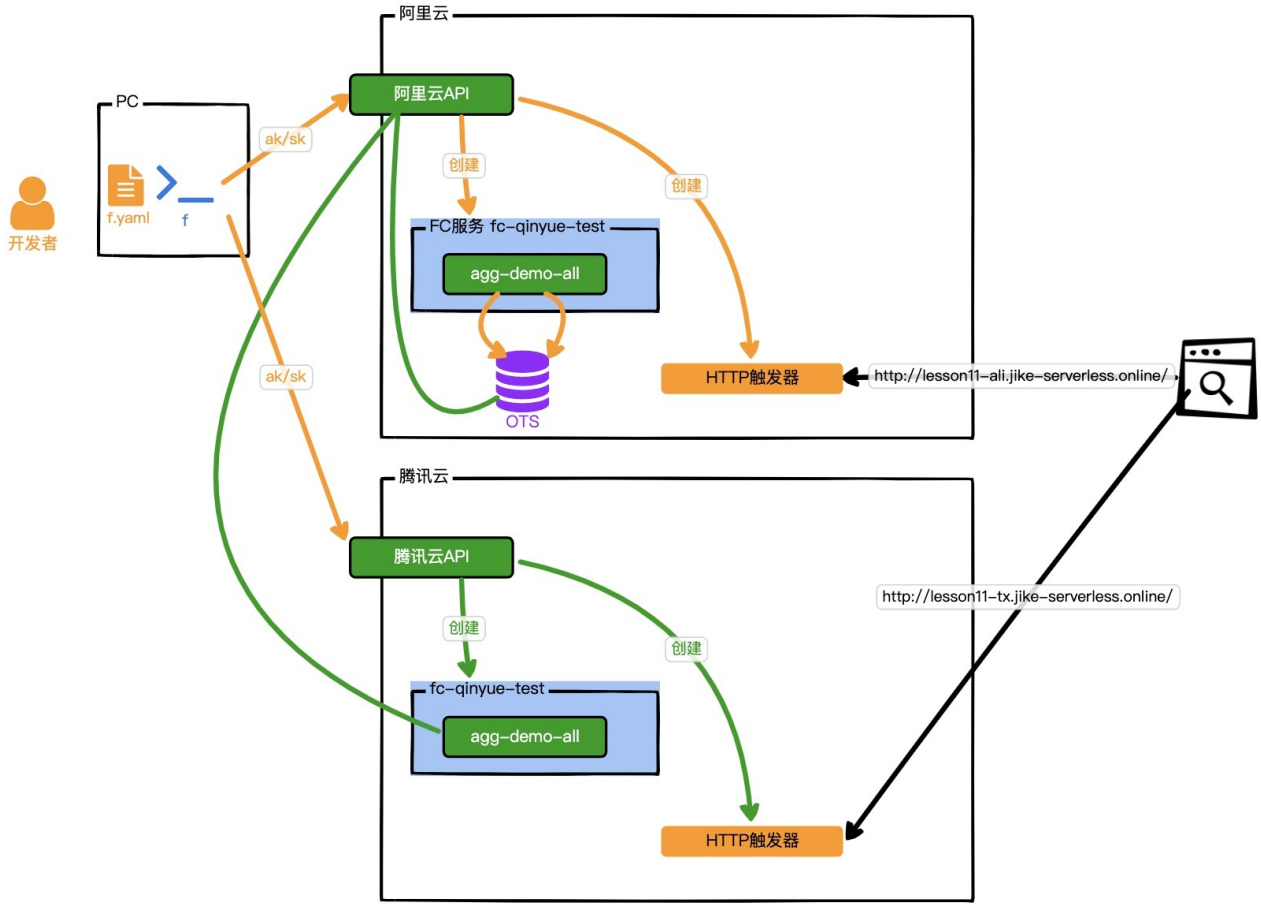


这个版本和上个版本相比，我们在K8s集群中安装了Istio组件和Knative组件。这2个组件都会给我们应用的Pod注入伴生容器Sidecar，就像一个监护人，监视着我们应用容器的真实状态，利用控制面板和数据面板的配合，自动化运维我们的应用在K8s集群中的状态。

研发人员还是只关心Dockerfile，所以对于研发人员来说是DevOps。运维人员只需要在K8s集群中安装好Knative组件，并维护Knative组件就可以了。应用的状态由Knative自动维护，所以我们也称之为Container Serverless。要注意这里的应用可以是单个函数，也可以是微服务，也可以是数据库，具体内容完全由你的Dockerfile去编排。所以我们可以看出，FaaS和BaaS的底层就是由容器服务实现的，但具体的容器方案，可能是Docker，也可能是虚拟机。每个云服务商都有自己的策略，不过我们通过Knative可以了解到Container Serverless的工作原理。

Serverless应用

“待办任务”Web服务，第八个版本是lesson11分支，以下是这个版本的示意图。



这个版本是个重大的重构，因为跟我们前面的写法都不一样了。我们的“待办任务”Web服务采用了Midway-FaaS框架，依托Midway-FaaS去适配云服务商，支持我们的应用可以自由选择部署在阿里云或腾讯云的FaaS上。而且这个版本，我们的代码写成的TypeScript，逻辑也更加清晰了。这个也是现在FaaS部署Serverless应用的最佳体验，大厂将自己的成功的Serverless业务，做成方案沉淀，作为框架输出。我们依赖这些Serverless框架，可以快速开发迭代我们自己的Serverless应用，享受FaaS的红利。

以上就是贯穿咱们专栏的案例——“待办任务”Web服务的整个演进过程了。

结语

我搜集了部分关注Serverless技术的同学的提问，在这里我也想统一回答一下，你也可以看看自己是否也有同样的疑问。

就像我[第1课]中所讲的，Serverless是对现代互联网服务端运维体系的极端抽象，对开发者的变革较大。降低了服务端运维的门槛，就意味着即使服务端运维经验是零，也可以将自己开发的应用快速部署上云，这点对前端工程师是很大的利好。

对于后端工程师和运维工程师，掌握FaaS和服务编排，无疑也是一大利器。FaaS的低成本、高可用，还有事件响应机制，都可以在现有的后端微服务或者应用架构中发挥出巨大的优势。

对于云服务商，FaaS还可以利用碎片化的物理机计算资源，提升资源利用率，而且还可以帮助云服务商提升云服务的利用率。

有同学让我讲一下大厂的成功案例，其实大家看到的很多FaaS创建的模板，都是来自于大厂案例的沉淀。FaaS诞生的过程，其实是大厂或云服务厂商将自己的应用运维能力逐渐下沉的一个过程，并不是先有了

FaaS，大家再去思考什么场景适合FaaS。这也是为什么我要用“待办任务”Web服务这个案例，一步步升级运维体验，向你讲解整个Serverless的发展史。

最后，我想说，我并不想用我们的最佳实践来束缚你的思想。我一直觉得，Serverless虽然是在大厂运维能力的基础上诞生并成长来的，但是利用Serverless，再结合我们的想象力，是可以创造出更多的可能的。总的来说，一句话，不要让它束缚我们的想象力，Serverless+AI、Serverless+IoT、Serverless+游戏等等，才应该是我们下一步要探索的方向。

未来，加油！

精选留言：

- 我来也 2020-05-13 11:28:10
  "这个专栏是一堂服务端技术架构课"
  这个描述一点也不夸张.

  从一个简单项目的多次变迁,可以看清架构是如何演变的.
  新引入的技术解决了原有中的什么问题.

  其实在平常的工作中,很难有这种完整的经历.
  特别是在业务比较平稳的企业,或业务规模不大的小企业中,原有的架构可能并不会遇到瓶颈.
  领导可能并没有优化架构的意愿.

  也许以后的人,都是直接基于云原生云平台来开发了.
  但理清了历史的变迁过程,才能更好的用好当下,和展望未来.

  看了老师的答疑,我有了新的认识.

虽然现在的Serverless大多都是Node.js或TypeScript的案例,但并不代表就只适合这个.
后面还有很大的想象空间,我们可以基于自己熟悉的语言,熟悉的场景,来用好Serverless.
为以后的人提供一些经验和参考.

感谢老师在此期间的辛苦付出!
[2赞]

作者回复2020-05-14 11:09:11
感谢你一路的支持，坚持做课后作业。我的课后作业也都画了很多心思设计的，以后我还想做成一个更完整的例子，不过只会更新github仓库了。
因为目前Serverless应用，我碰到好多前端同学学习，他们中间的知识跨度太大，所以才有了这门课的想法。使用Serverless不难，难的是怎么在实际工作中使用Serverless，目前也是百家齐鸣，这里无论是创业，就业，还是提升自我影响力，机会都很多。

- 许童童 2020-05-13 14:03:04
江湖再见

作者回复2020-05-14 11:04:58
来阿里巴巴可以见到我~

- Bora.Don 2020-05-13 10:23:47
谢谢老师的课程，虽然后半段有很多没看懂的地方。。。
很赞同最后的预测，Serverless不是只服务于网页前端的服务，IoT一样可以直接调用Serverless服务

作者回复2020-05-14 11:10:24
慢慢消化吸收一下，这里后半段面信息量比较大。
如果有问题，可以在留言区或者github上面可以和我互动。

- 文蔺 2020-05-13 00:56:58
安装knative时 总是遇到gcr.io镜像拉取失败的问题，请教老师有没有比较好用的解决办法

作者回复2020-05-14 12:46:47
我的文章和仓库中提供的docker-k8s-prefetch.sh，就是提前拉取镜像的。
镜像下载可以通过阿里云的镜像仓库的加速服务。