

# Serverless

副标题：Serverless：云计算的下一个十年

嗨，你好，欢迎来到 Serverless 课程，最近几年的技术圈，对于 Serverless 技术的讨论异常火热，在业内也有了很多成熟的案例，国外发展较早，比较有代表性的就是亚马逊和谷歌，而在国内，腾讯和阿里两位巨头，都将 Serverless 定义为集团战略型项目，不遗余力的推广和研发自己的 Serverless 技术；

那么，Serverless 到底是什么？为什么说 Serverless 是云计算的下一个十年？

## 第 1 章 认识 Serverless

首先我们先来认识一下 Serverless，Serverless 是一种后端架构技术，更准确的说，它应该是一种后端架构的概念或者思维，你可能会问，既然是后端架构的内容，那和我前端开发者有什么关系呢？没错，Serverless 本身和前端半毛钱关系都没有，但是它却是前端程序员最希望落地应用的技术，众多前端开发者，望穿秋水一般的期待它的成熟落地，而很多后端程序员，漠不关心甚至排斥；为什么会出现这种一方冷淡一方火热的奇怪现象？要搞清楚这一点，我们需要从后端架构的演进历史说起……

### 1-1 后端架构的演进

每一个 B/S 架构的互联网应用，都是由最基础的客户端和服务端构成的，客户端要呈现内容，就需要服务端提供服务，最初，搭建一个服务器是非常繁琐的，我们需要购买一台电脑主机，然后找一个机房对机器进行托管，要将外观拍照，将各项硬件参数提交到备案中心进行备案，链接电源和网线，安装好操作系统，然后搭建好代码的运行环境，部署程序代码后，再将申请的域名和静态IP做好解析，就算是上线部署完毕了，这个过程需要肉身行动，减肥效果非常不错；

这样的服务器架构是单机版的单体架构，数据库、应用代码、HTTP 服务器等服务全都在一台你自己管理的服务器上运行，因为我们要接触物理机，所以，我们也把它成为 物理机时代；

在 物理机时代 不仅仅是上线部署非常繁琐的问题，在整个应用的运行中，还有各种各样的问题出现，比如磁道磨损的硬盘，机房的意外停电，老鼠咬断的网线；

随着技术的不断发展，我们终于摆脱了物理机时代，跨入虚拟机时代。其中一个重要节点之一就是 2001 年 VMWare 带来的针对 x86 服务器的虚拟化产品，通过虚拟化技术，可以把一台物理机分割成多台虚拟机提供给用户使用，充分利用硬件资源，而对于硬件设备的管理，统一由云厂商负责，对于开发者来说，就不用再买硬件了，直接在云平台买虚拟机，比如 AWS 的 EC2、阿里云 ECS、腾讯云 CVM；



云服务器也真正的进入了大众的视野，开发者再也不用担心断电断网和硬件故障了，不过业务量的不断增长，用户越来越多，数据库每天都有几千万条数据写入，数据库性能很快就会达到瓶颈；除此之外，每天也有上百万图片存到磁盘，磁盘也快要耗尽了。为了降低服务器负载，我们把数据库迁移到了云厂商提供的云数据库上，把图片存储迁移到对象存储：

云数据库有专门的服务器，并且还提供了备份容灾，比自己在服务器上安装数据库更稳性能更强。

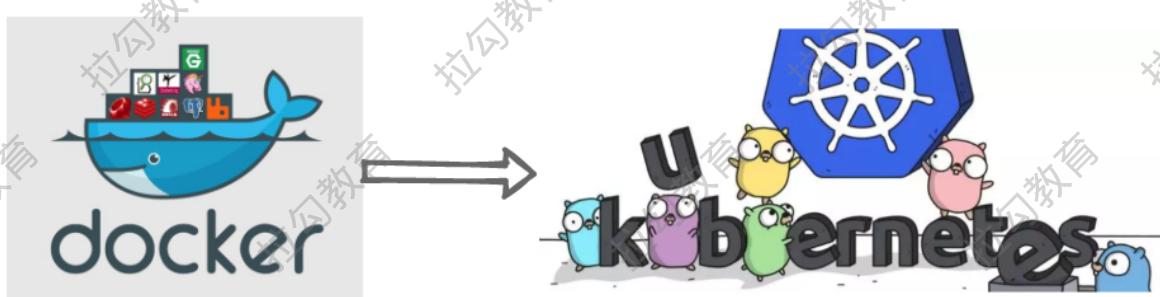
对象存储能无限扩容，不用担心磁盘不够了。

这样一来，服务器就只负责处理用户的请求，把计算和存储分离开来，既降低了系统负载，也提升了数据安全性。并且单机应用升级为了集群应用，通过负载均衡，会把用户流量均匀分配到每台服务器上。

不过在服务器扩容的过程中，你还是会遇到一些麻烦。比如购买服务器后，都需要在上面初始化软件环境和配置，还需要保证所有服务器运行环境一致，这是个非常复杂还容易出错的工作。这对运维工程师也是一个非常大的挑战；

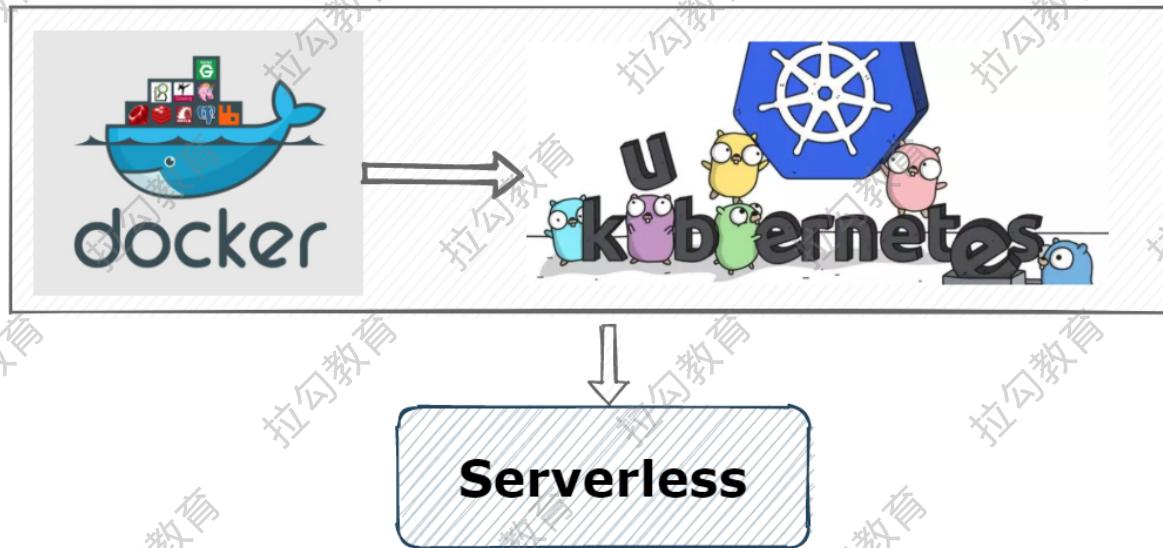
总的来说，虚拟机可以让你不用关心底层硬件，但是我们依然要为服务器集群的管理工作付出高昂的成本，如果有一项技术，能够在每次的服务器扩容时，让服务器的运行环境保持一致，那就太好了，于是，容器技术应运而生。

2013 年 Docker 的发布，代表着容器技术替代了虚拟化技术，云计算进入容器时代。容器技术就是在虚拟化技术的基础上，把代码和运行环境打包在一起，这样，不论服务器的配置怎样，代码和运行环境均能保持一致性。有了容器技术，你在服务器上部署的就不再是应用了，而是容器。当容器多了的时候，如何管理就成了一个问题，于是出现了容器编排技术，比如 2014 年 Google 开源的 Kubernetes，就是俗称的 K8S。



在目前的后端架构中，容器技术依然是主流的服务器架构技术，但是随着互联网应用的普及，我们需要面对各种各样的应用场景，举个栗子，每当大型购物节来临，我们的在线商城会面临巨大的流量洪峰，而在平时，流量显然要小很多，为了迎接购物节的流量洪峰，我们需要大量扩充服务器，服务器的扩充和大量容器的编排工作，也是一个不小的难题，如果我们对流量的压力预估不到位，还会有服务器宕机的风险，而在平时，如此多的服务器运行显然是在浪费资源增加成本；

那么，有没有一种技术，让我们只关心业务代码的功能实现，脱离服务器的管理呢？让我们不再为运行环境劳心伤神，不再为服务器的扩缩容半夜惊醒，当流量洪峰来临时，自动调配更多的服务器资源支撑，当流量低谷时，自动释放服务器资源节约成本；这样美好的时代，正在想你走来，它叫 serverless



简单总结一下，纵观后端架构的发展史，其实就是 Serverless 的兴起史，每一个时代，都是对前一个时代基础架构的抽象，从物理机时代跨越到虚拟机时代，让我们不在关心硬件设备的管理，将一台真实的计算机抽象为相互隔离的多个虚拟机，从虚拟机时代到容器时代，让我们摆脱了运行环境和集群管理的繁杂工作，将虚拟机的运行环境抽象为容器，而serverless 的到来，让我们不在关心运行环境和服务器资源的调配；

## 1-2 Serverless 的基本概念

上一节，我们通过简单的梳理后端架构的发展，引出了 Serverless，你可能已经迫不及待的想要尝试 Serverless 了，但是Serverless中有很多全新概念的引入，相比于具体的应用，Serverless 的相关理念更值得我们探讨和学习，那么 Serverless 到底是什么呢？技术圈对 Serverless 的定义也在不断的调整和变化中，所以导致有不少刚接触 Serverless 的同学会认为 FaaS 就是 Serverless，也有同学认为 PaaS 也是 Serverless，还有同学说使用 Serverless 就没有服务器了。总的来说，很多同学对 Serverless 到底是什么并没有一个很清晰的认知，概念还比较模糊，那么，接下来，我们尝试从广义和狭义两个角度入手，解释 Serverless 的架构理念；

广义上来说，Serverless 是一种后端架构理念，或者说是一种思想、概念，直接翻译过来叫做“无服务”，但是不要被字面意思误导，这并不代表着应用运行不需要服务器，在 Serverless 时代之前，我们可以将传统架构统称为 Serverful 时代，意思就是关于服务器的一切，我们都需要人工干预，而 Serverless，更准确的说，应该是开发者不用关心服务器的意思，是将服务器相关的工作交给云平台来做，对于开发者来说，与服务器运维有关的所有工作都不再关心，Server（服务器）是不可能真正消失的；

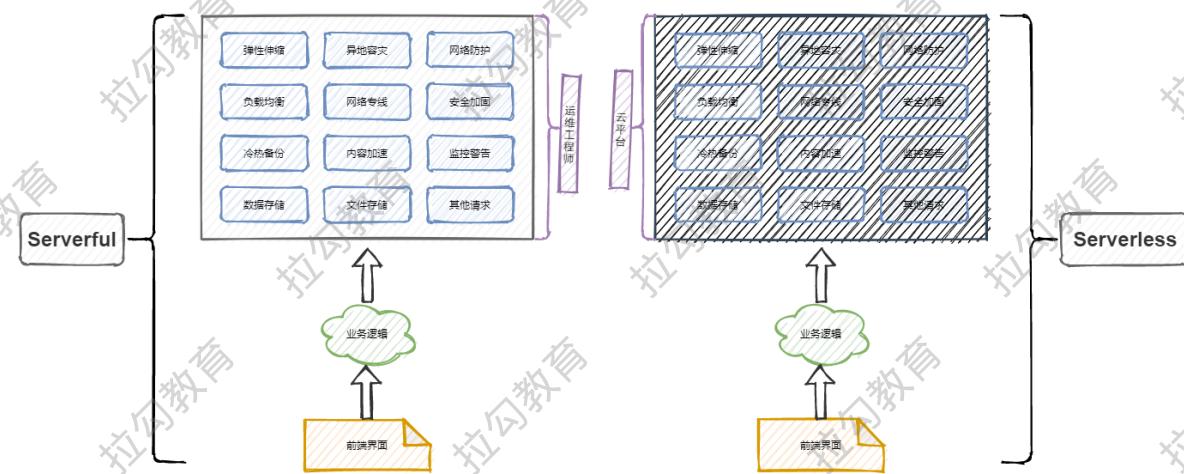
2019 年 2 月，UC伯克利大学发表了一篇标题为《Cloud Programming Simplified: A Berkeley View on Serverless Computing》的论文，论文中有这样一段对Serverless 的描述：

在云的上下文中，Serverful 的计算就像使用低级的汇编语言编程，而 Serverless 的计算就像使用 Python 这样的高级语言进行编程。例如  $c=a+b$  这样简单的表达式，如果用汇编描述，就必须先选择几个寄存器，把值加载到寄存器，进行数学计算，再存储结果。

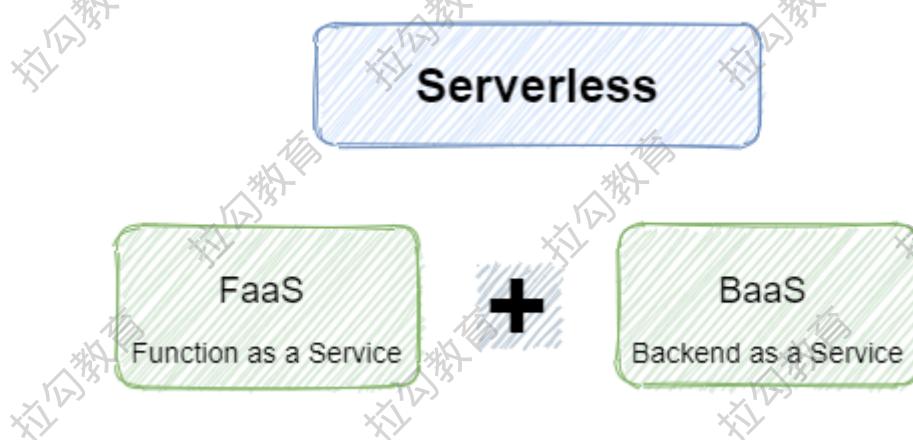
这就好比今天在云环境下 Serverful 的计算，开发首先需要分配或找到可用的资源，然后加载代码和数据，再执行计算，将计算的结果存储起来，最后还需要管理资源的释放。

如果你不是计算机专业，可能感知并不是很强烈，但从计算机专业的角度来讲，这样的比喻非常清晰，Serverful 是我们今天主流的使用云的方式，但不应该是未来我们使用云的方式，Serverless 所希望的是开发者用代码去支撑业务逻辑，而对于资源的管理交给工具和云；

在 Serverful 的架构下，我们需要关心的问题非常多，比如：根据业务流量大小等指标，响应式地调整服务规模，实现自动弹性伸缩。再比如异地容灾、负载均衡、日志监控、文件存储等等，解决这些复杂的问题需要投入大量的人力、物力，而在 Serverless 架构下，开发者只专注于开发业务逻辑，所有的这些与业务无关的基础设施，全部交给云平台负责，由云平台统一调度、运维。



在这样的理念指导下，各家云平台厂商，都有不同实现方案，每家云平台提供的 Serverless 服务，都或多或少的存在差异，但是，按照 CNCF（云原生计算基金会）对 Serverless 计算的定义，Serverless 架构应该是采用 FaaS（函数即服务）和 BaaS（后端即服务）服务来解决问题的一种设计。这样的定义从应用落地的角度来说，更加的具体可行，也让我们对 Serverless 的理解更加的清晰明了；因此，从应用落地的角度，狭义的 Serverless 就是 FaaS+BaaS 的组合；



那么 FaaS 和 BaaS 分别是什么呢？

FaaS 是 Function as a Service 的缩写，翻译过来的意思是“函数即服务”；

BaaS 是 Backend as a Service 的缩写，意思也很简单，翻译过来就是“后端即服务”

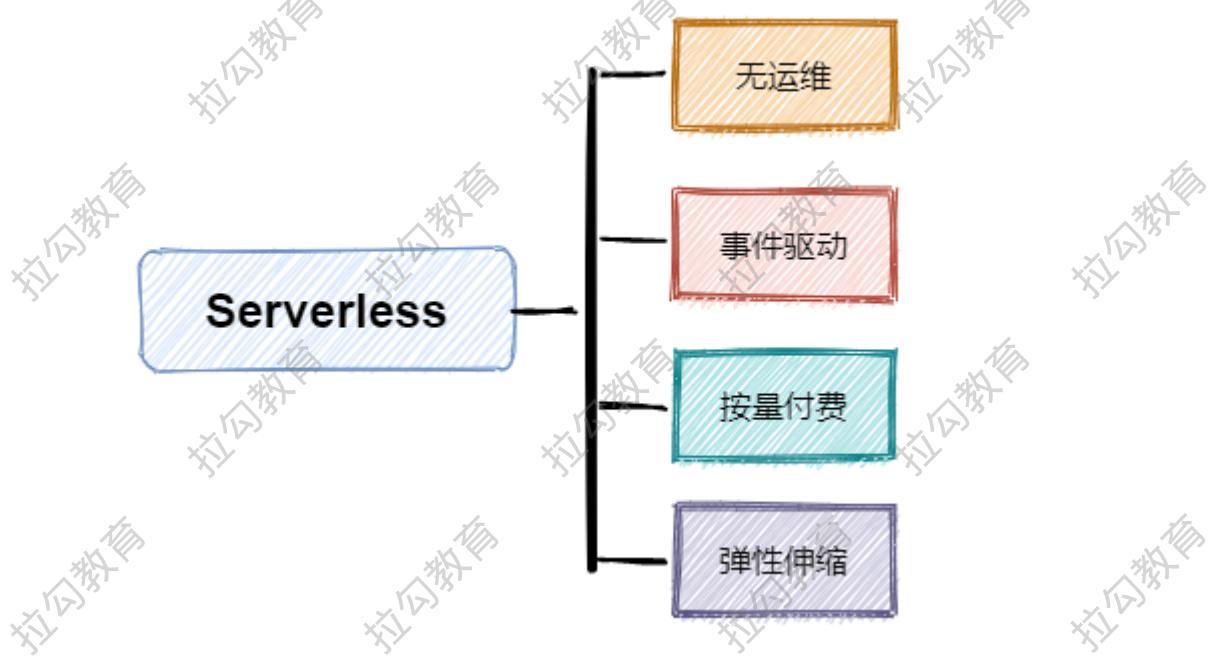
具体什么意思呢？我们先来看 BaaS，前面说，Serverless 把后端架构的工作全部包揽下来，硬件的维护，集群的管理，运行环境的搭建，全部由云平台完成，除此之外，像缓存、数据库、文件存储、消息中间件等，也全部由云平台帮我们做好，封装起来，以接口的形式提供服务，这就是 BaaS，所谓后端即服务，对于开发者，BaaS 就是一个黑盒，你不用知道我怎么做，更不需要关心我如何做，你需要什么过来拿就行了；

但是，我需要向数据库存一条数据，用户上传的照片我需要裁剪以后存到文件存储中，这是需要我们编写业务逻辑代码完成的功能，假设我现在已经把这些逻辑代码写好，用的是 Node.js，前面说所有的服务器及运行环境都放在了 BaaS 这个黑盒子中，我怎么让这些代码运行呢？换句话说就是，我现在写的逻辑代码，是需要 Node.js 这个运行环境的，怎么办？我们只需要将写好的代码，交个 Serverless 就行了，Serverless 中有专门运行我们的逻辑代码的地方，这个地方就是 FaaS，FaaS 是以函数的方式运行我们的代码的，本质上 FaaS 就是一个函数运行平台，大多数的 Serverless 云平台提供的 FaaS，都支持 Node.js、Python、Java、PHP 等编程语言，你可以选择你喜欢的编程语言编写函数并运行。对于开发者来说，使用 FaaS 几乎就是使用 Serverless 的一切了，在 FaaS 中，我们能够体会到 Serverless 全部的特性；

首先 FaaS 函数运行时，开发者对底层的服务器是无感知的，FaaS 产品会负责服务器资源的调度和运维，这些就是我们前面说的 BaaS，这也是 Serverless 最大的特点，无运维；

其次，FaaS 中的函数也不是持续运行的，而是通过一定的条件进行触发，比如 HTTP 事件、消息事件、定时器事件等，产生事件的源头叫触发器，FaaS 平台会集成这些触发器，我们直接用就行，这是 FaaS 的第二个特点，事件驱动。

再者就是 Serverless 的付费方式了，与其他云产品不同的是，Serverless 的付费方式是按量付费，是按照 FaaS 函数执行次数和执行时消耗的 CPU、内存等资源进行计费的，用多少付多少，不用不付费，同时，FaaS 会根据并发量自动生成多个函数实例，BaaS 会根据函数运行所需要的资源量自动调配服务器资源，理论上的资源调用量没有上限，这也就实现了不同访问量的弹性伸缩了，而且是实时的弹性伸缩；



基于 FaaS 和 BaaS 的架构，是一种计算和存储分离的架构。计算由 FaaS 负责，存储由 BaaS 负责，计算和存储也被分开部署和收费。这使应用的存储不再是应用本身的一部分，而是演变成了独立的云服务，降低了数据丢失的风险。而应用本身也变成了无状态的应用，更容易进行调度和扩缩容。

基于 FaaS 和 BaaS，你的应用就实现了自动弹性伸缩、按量付费、不用关心服务器，这正是 Serverless 架构的必要因素。所以说狭义的 Serverless 是 FaaS 和 BaaS 的组合。

## 1-3 Serverless 的优缺点

我们前面介绍 Serverless 是什么，说了很多 Serverless 的特点，实际上这些也都是它的优点，这里我们也简单总结一下，Serverless 可以不用运维、实现自动的弹性伸缩、按量付费节省成本、更高的安全性、易于迭代和部署。

Serverless 就是十全十美的吗，不，它一样也存在很多缺点。了解它的缺点，可以让你今后更好地进行技术选型，决定是否用 Serverless 进行应用开发。那么它都有什么缺点呢？

### 依赖第三方服务

Serverless 的能力是云厂商打包提供的，所以 Serverless 产品一定是和云厂商绑定的，又因为 Serverless 理念和具体实现之间并没有统一的标准，比如 A 厂商认为 Serverless 的数据库必须使用标准 SQL 规范，而 B 厂商则认为数据库可以使用 SQL 规范也可以使用 JSON 文件的存储方案，这就出现了不同的云厂商实现了不同的 FaaS 接口，我们的同一套代码，是无法在不同的 Serverless 产品上运行的，要想从一个云平台迁移到另一个云平台，成本非常高；

### 开发调试困难

Serverless 应用依赖的云服务，难以在本地环境搭建，要想在本地开发调试非常复杂。同时，Serverless 架构正处于飞速发展的阶段，其开发、调试、部署工具链并不完善；

### 底层硬件的多样性

目前 Serverless 的技术实现是 FaaS 和 BaaS。我们的应用代码在 FaaS 上运行，但 BaaS 是个黑盒，其底层的硬件资源是不确定的，某些场景下，代码必须运行在某种类型的 CPU 或 GPU 上，目前云厂商并没有提供针对底层硬件的可选项；

## 第2章 基本应用

前面我们介绍了 Serverless 的由来，了解了 Serverless 架构的基本理念和设计思想，了解了它的诸多优点和缺点，这就结束了吗？当然不是，Serverless 还有很多特性我们并没有讲到，包括它其他的优缺点，运行机制，开发方式等等，当然，你也可能对前面的讲解依然处于似懂非懂的状态，没关系，接下来，我们会在具体的应用实践中，深入感受 Serverless 的架构理念，补充对 Serverless 其他特性的讲解，关于 Serverless 的一切，我们会在接下来的案例中，一一呈现；

前面说，Serverless 是与云厂商绑定的，所以，我们需要选择一家云平台服务商来进行实战演练，我用过阿里云、AWS 以及腾讯云提供的 Serverless，相比于其他厂商，我个人非常推荐腾讯云，并不是其他厂商不好，只是针对初学者，腾讯云提供了非常好的手册及教程，而如果你使用过小程序的云开发，那么对于腾讯云，你会感到非常熟悉；腾讯云开发文档中心：<https://docs.cloudbase.net/>

### 2-1 注册并登录腾讯云

前往 [腾讯云官网](#)，注册腾讯云账号，然后登录账号。如有账号，可以直接登录。注意，一定要先完成实名认证；

## 2-2 创建云函数

登录之后，申请开通 [云开发 CloudBase](https://cloud.tencent.com/product/tcb)：<https://cloud.tencent.com/product/tcb>，进入控制台，创建应用：

创建环境

1 应用模板 > 2 环境信息 > 3 应用配置

地域：广州 上海

计费方式：按量计费 包年包月 模式对比

环境名称：mycloudes

免费资源：开启免费资源

费用：按实际使用的资源量收费。查看计费详情 同意《计费规则》

上一步 下一步 取消

创建成功后，点击 云函数 --> 新建云函数：

新建函数

1 基础信息 > 2 函数配置

函数名称：hello

支持大小写字母、数字、-和\_，但必须以字母开头、以字母和数字结尾，最长45个字符

创建方式：空白函数 使用helloworld示例创建空白函数

运行环境：Nodejs 10.15

函数内存：256MB

下一步 关闭

将函数代码修改如下：

The screenshot shows the Tencent Cloud Function Editor interface for a function named 'hello'. The '函数代码' tab is selected. A red arrow points from the top-left towards the code editor area. Inside the editor, the 'index.js' file is open, displaying the following code:

```
exports.main = async (event, context) => {
  return "hello tcb & xiling"
};
```

A red box highlights the return statement 'return "hello tcb & xiling"'. Another red arrow points from the bottom of the code editor towards the bottom buttons. The '保存' (Save) button is highlighted in blue, while the '保存并安装依赖' (Save and install dependencies) button is in a grey box.

```
exports.main = async (event, context) => {
  return "hello tcb & xiling"
};
```

简单解释一下：

`index.js` 是入口文件，其中的 `main` 函数是入口函数，函数接受两个参数 `event` 对象和 `context` 对象。

`event` 对象指的是**触发云函数的事件**。例如：

- 在小程序端调用时，`event` 是小程序端调用云函数时传入的参数；
- 在使用 HTTP 请求的形式调用时，`event` 是**集成请求体**。

`context` 对象：包含了此调用的**调用信息和函数的运行状态**，可以使用 `context` 了解服务运行的情况。

## 2-3 创建触发器

保存函数代码后，如何执行和访问呢？前面说过，FaaS 产品的一大特性是事件驱动，想要让函数代码执行，我需要创建一个事件的触发器，我们比较熟悉的就是 HTTP 触发器了，在腾讯云它叫“HTTP访问服务”，点击新建：

### 新建触发路径

- 触发时优先匹配云函数中的触发路径
- 变更触发路径后，需同步修改资源服务内的路由

域名 i

\*

触发路径

/hello

关联资源

云函数

hello

鉴权开关 i



确认

取消

等待触发器创建成功后，我们可以使用默认域名，访问应用函数：



## 2-4 本地环境及开发工具

CloudBase CLI 是云开发 (Tencent CloudBase, TCB) 开源的命令行界面交互工具，用于帮助用户快速、方便的部署项目，管理云开发资源。

CloudBase CLI 是基于 Node.js 开发的工具，因此，我们需要保证本地的 Node.js 版本在 8.6 以上

执行 `npm i -g @cloudbase/cli` 或 `yarn global add @cloudbase/cli` 进行全局安装，安装成功后，使用 `tcb -v` 查看版本号，使用 `tcb -h` 查看相关帮助，显示所有可用命令；

在使用之前，我们需要先进行授权登录，命令行执行 `tcb login`，会自动弹出浏览器的授权界面，确认授权后，命令行的控制台会打印登录成功的相关信息：

The screenshot shows a terminal window and a modal dialog. The terminal displays the command \$ tcb login and its output, which includes CloudBase CLI version 1.6.4 and Framework version 1.7.1, followed by a success message and usage instructions. A red arrow points from the terminal to the modal. The modal is titled 'CLI 授权' (CLI Authorization) and contains a sub-section '① CLI 授权说明' (CLI Authorization Description). It states: '您将授权 CloudBase CLI 工具以下权限' (You will grant the CloudBase CLI tool the following permissions) and lists: '操作 CloudBase 资源的全部权限, 包括但不限于 COS, SCF, 数据库等' (Full permissions to operate CloudBase resources, including but not limited to COS, SCF, databases, etc.). At the bottom are '取消' (Cancel) and '确认授权' (Confirm Authorization) buttons.

```
administrator@xiling-Y9000x MINGW64 /d/Codes/cloudbase
$ tcb login
CloudBase CLI 1.6.4
CloudBase Framework 1.7.1
✓ 登录成功!
可使用下面命令继续操作:
- 创建免费环境
  $ tcb env create envName
- 初始化云开发项目
  $ tcb new
- 部署云函数
  $ tcb fn deploy
- 查看命令使用介绍
  $ tcb -h
Tips: 可以使用简写命令 tcb 代替 cloudbase
```

授权成功后，我们就可以在本地创建云函数了，在命令行执行 `tcb new` 命令行的选项中，有一项需要注意，本地创建是只能创建函数而不是应用，也就是说，我们最好是先在控制台创建好应用后，再创建本地函数，准备好之后，我们就可以看到可选的应用的了，选择函数所属的应用后，在选择对应的函数模板；

这里我们选择与之前一样的 Node 云函数示例，输入项目名称后，云函数会下载相关代码道本地，注意，此时，在控制台中，我们是看不到这个函数的，也就是说，这个函数并没有上线运行，其实命令行中也给了我们提示：“执行命令 `tcb 一键部署`”，想要上线运行，还需要进到项目目录中，执行 `tcb` 命令；

The screenshot shows a terminal window displaying the deployment log for a Node.js application. The log starts with the framework's version information and configuration validation. It then details the deployment process, including plugin installation, hooks execution ('preDeploy', 'postCompile'), and function compilation. A progress bar indicates the deployment is at 100% completion. Finally, it shows the successful deployment of the function and the entire application ('node-app').

```
CloudBase Framework  info  Version v1.7.1
CloudBase Framework  info  Github: https://github.com/Tencent/cloudbase-framework

CloudBase Framework  info  EnvId lagou01-3gcch2haf8740e8f
CloudBase Framework  info  Validate config file success.
CloudBase Framework  info AppName node-starter
CloudBase Framework  info  ◆install plugins
CloudBase Framework  info  callHooks 'preDeploy'
CloudBase Framework  info  ◆init: function...
CloudBase Framework  info  ◆build: function...
CloudBase Framework  info  ◆compile: function...
CloudBase Framework  info  callHooks 'postCompile'
正在部署[██████████] 100% 5.0 s
CloudBase Framework  info  ◆deploy: function...
CloudBase Framework  info  ◆[node-app] 云函数部署成功
CloudBase Framework  info  ◆云函数部署成功
CloudBase Framework  info  callHooks 'postDeploy'
CloudBase Framework  info  ✨ done
```

函数的运行是需要触发器的，所以，函数部署成功后，执行 `tcb service create` 命令，创建 http 触发器，创建成功后，会返回访问地址，但是，与在控制台创建触发器一样，需要等待几分钟的时间后，才能看到访问结果；

```
Administrator@xiling-Y9000x MINGW64 /d/Codes/cloudbase/sh-node
$ tcb service create
CloudBase CLI 1.6.4
CloudBase Framework 1.7.1
✓ 请选择创建HTTP 访问服务的云函数 . node-app
✓ 请输入HTTP 访问服务路径 . /nodes
✓ HTTP 访问服务创建成功!
点击访问> https://lagou01-3gcch2haf8740e8f-1253229081.ap-shanghai.app.tcloudbase.com/nodes

Administrator@xiling-Y9000x MINGW64 /d/Codes/cloudbase/sh-node
```

接着，我们修改函数返回值的内容，重新刷新后并没有显示，这需要我们更新函数代码后才能生效，可以使用 `tcb fn code update xxx函数名`，命令只会更新函数的代码以及执行入口，不会修改函数的其他配置；

## 2-5 本地测试工具

虽然代码放在了本地编写，但是，本地是没有触发器的，当然我本地有Node环境，但是与云环境还是有很大差别的，我不能每次写完代码，都需要上传在测试啊，这中体验根本就没法用啊，为了解决这个问题，腾讯开发了开源的 `scf-cli : https://github.com/TencentCloud/scf-node-debug` 工具，帮助我们在本地快速调试，基本原理就是在我本地开启一个服务器模拟云环境，让我们可以在本地进行调试；

```
$ scf
Usage: init|i [options]

本地测试运行云函数的小工具

Options:
  -V, --version          output the version number
  -h, --help              output usage information

Commands:
  init|i [options] [host][port][debug]  Init The TestCli For Your App
```

执行 `npm install scf-cli -g` 安装，安装成功后，在项目目录下执行 `scf init` 或者 `scf i` 开启本地测试环境，命令提示符会让我们输入入口文件的路径，以及入口函数名，设置好超时时间后，选择测试方式；

```
$ scf init
? 请输入入口文件地址(相对路径) ./functions/node-app
? 请输入入口执行方法名称 main
? 请输入超时时间限制(单位: s) 5
? 请选择测试模版 http
[Weapp CLI] [2021-03-29T19:47:21+08:00] Server has listened [IP]:localhost [PORT]:3000. http://localhost:3000
```

如果选择的是 `http`，那么默认的会开启3000端口的服务器；此时，我们就可以愉快的使用 Serverless 云原开发了；

## 2-6 Express 与云函数

经过前面一些列的配置，我们终于可以在本地开发调试了，但是，使用纯原生的 Node.js 开发，效率是非常低，如果能在云函数中引入一款我们熟悉的后端开发框架，那么我们的开发，才算坐上了高铁；这里我们选择一款比较大众化的框架，Express，如何将 Express 结合到云函数中呢？

先在本地安装好 Express，按照以往的开发经验，我们在本地开启 HTTP 服务。

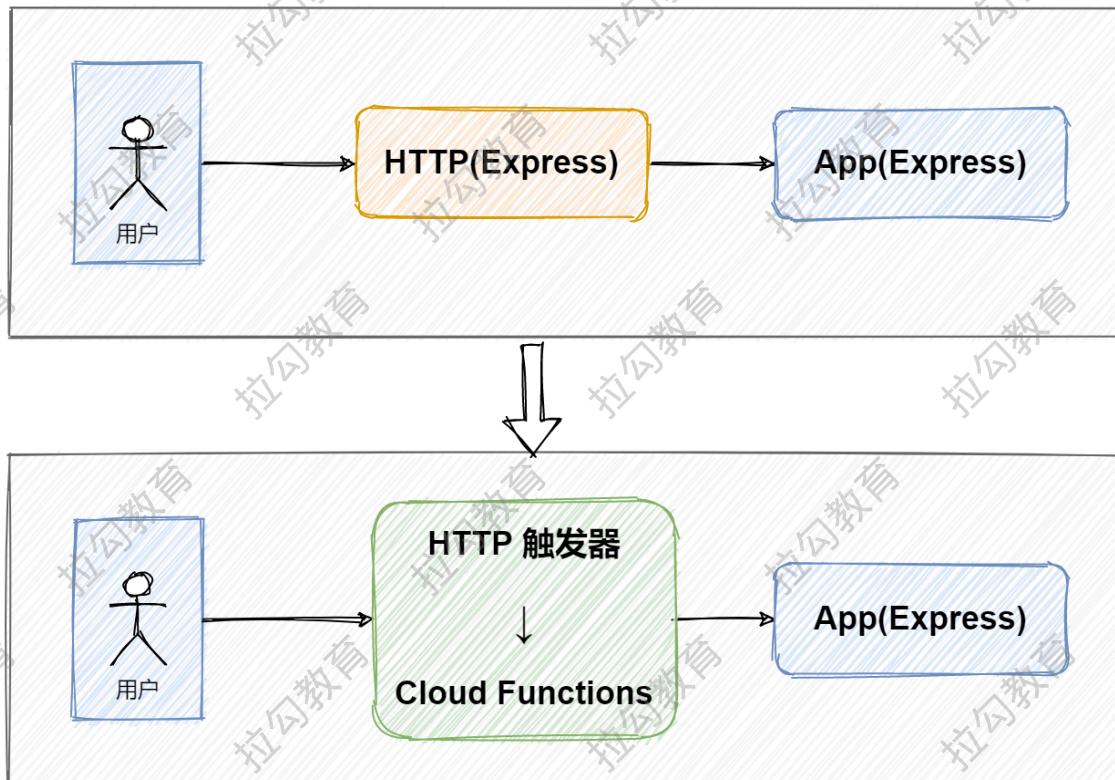
```
const express = require("express");
const app = express();

app.use('/users', (req, res) =>{
    res.send('lagou')
});

module.exports = app;
```

```
const app = require('../app');
app.listen(3000, () =>{
    console.log('本地测试: http://127.0.0.1:3000')
});
```

而在云函数中，我们知道，代码是在入口函数运行的，而且云环境中有 HTTP 的触发器，是不需要我们创建服务器的



有一款开源工具，专门用作对框架代码进行包装，[serverless-http](#):  
<https://github.com/dougmoscrop/serverless-http>，<https://www.npmjs.com/package/serverless-http>是专门用于在 Serverless 环境下，包装接口的模块，不需要服务器，也不需要监听端口；  
使用方式也很简单，命令行安装 `npm install serverless-http`

```
const serverless = require('serverless-http');
let app = require('./app.js');
const handler = serverless(app)
module.exports.main = async (event, context) => {
  const result = await handler(event, context)
  return result;
};
```

写好之后将代码部署到云平台，然后进行请求测试，在本地开发测试，我们开启本地服务器就可以了；

## 第3章 TodoList API 接口案例

前面将云函数与Express 进行整合后，接下来我们通过案例的方式继续学习云函数的使用开发；这里我们选择实现一个 TodoList 案例的后端 API 接口，这个案例具备最基础的增删改查等基础功能；

### 3-1 添加业务路由

首先，我们将上面的代码进行修改，将 TodoList 的业务处理分发到不同的路由请求；

```
const express = require("express");
const app = express();

app.use(express.json());
app.use(express.urlencoded({ extended: false }));

const indexRouter = require("./routers/index");
const todoRouter = require("./routers/todo");

app.use("/", indexRouter);
app.use("/todo", todoRouter);

module.exports = app;
```

routers\index.js

```
const express = require('express')
const router = express.Router();

router.get('/', (req, res)=>{
  res.send('index router @西岭老湿')
})

module.exports = router
```

node-app\routers\todo.js

```
const express = require('express');
const router = express.Router();
const db = require('../db')

router.get('/', async function(req, res, next) {
  // 获取全部数据
});

module.exports = router;
```

代码实现之后，在本地请求 / 根路径 和 todo 路径，测试完成后，部署云函数，然后再进行对应的测试；

基础的业务路由配置好之后，我们回到业务代码的编写中，再 TodoList 案例中，我先来实现 增删改查 的相关操作。

```
const express = require('express')
const router = express.Router();

// 获取任务
router.get('/',(req,res)=>{
  res.send('get todo router')
})

// 添加任务
router.post('/',(req,res)=>{
  res.send('add todo router')
})

// 修改任务
router.put('/',(req,res)=>{
  res.send('change todo router')
})

// 删除任务
router.delete('/',(req,res)=>{
  res.send(' del todo router')
})

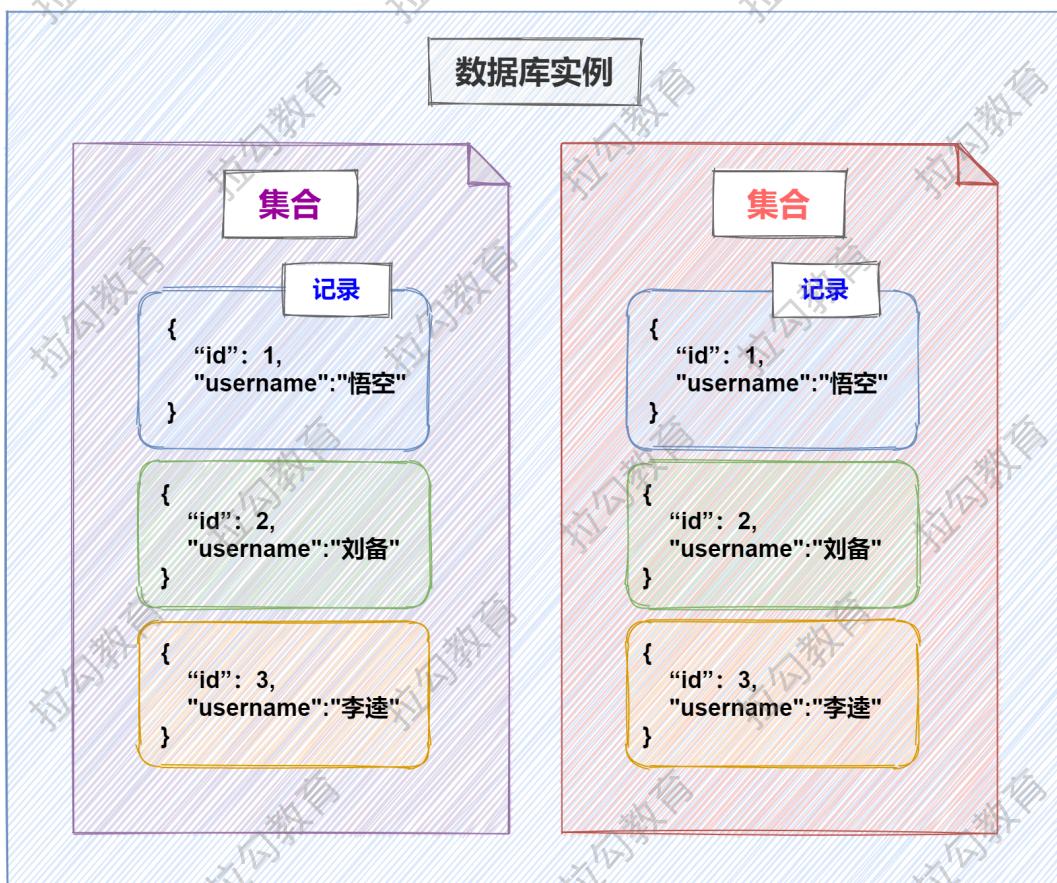
module.exports = router
```

使用 postman 进行本地及云函数的测试；

根据我们 Todo 的业务逻辑，肯定是先需要往里面添加数据的，有了数据，才能进行后面的 增删改，问题在于我们添加的数据存放在什么地方？当然是存数据库了，巧了，cloudbase 提供了云数据库，我们直接用就行了，那么具体怎么用呢？

## 3-2 云数据连接

在使用云数据库之前，我们需要先理清楚它的一些基本概念；腾讯云给我们提供的云数据库是一种**文档型数据库**，提供基础读写、聚合搜索、数据库事务、实时推送等功能，数据库中有数据库实例、集合、记录这三个基本概念，每个云开发环境下有且只有一个数据库实例，数据库实例中，可以创建多个集合，你可以将集合理解为一个文本文件，每个文件中可以存放多个类似 JSON 格式的对象，这样的对象就被称为记录；



那么如何使用呢？我们打开官方手册看一下：<https://docs.cloudbase.net/database/introduce.html>

需要注意的是，我们需要下载安装 node-sdk, `npm install cloudbase/node-sdk`，通过服务端调用时，需要在 SDK 初始化参数中，填入应用的envid，同时需要填入腾讯云密钥（SecretID 和 SecretKey），手册上并没有说，但是一定注意，除了腾讯云密钥还需要 env，也就是云环境 ID；

```
const nodesdk = require("@cloudbase/node-sdk");

const cloudDb = nodesdk.init({
  env: 'lagou01-3gcch2hafxxxxxxxxx',
  secretId: 'AxxxxxxxxxxxxKpww6zbXXXXXXXXX',
  secretKey: 'kFxdo0xxxxxp22Awixxxxxxxxxxx'
})

// 获取数据库引用(数据实例对象)
var db = cloudDb.database();
```

配置好基本信息之后，我们就可以连接数据库了，执行对应操作了，但是，数据库的操作以集合为单位的，所以，在操作之前需要先创建集合，用 `db.collection` 获取集合引用后，再执行对用操作就可以了；

```
// 添加任务
router.post('/',async (req,res)=>{
  var addtodoData = {
    content:'学习拉勾前端课',
    done:false
  }
  // 数据库的操作以集合为单位的，db.collection 获取集合引用
  const backDb = await db.collection('todo').add(addtodoData);
  console.log(backDb);
  res.send('add todo router---'+backDb)
})
```

最后，我们将数据库的连接进行封装，创建 config/db.js

```
const nodeSdk = require("@clouddbase/node-sdk");

const cloudDb = nodeSdk.init({
  env: 'lagou01-3gcch2haxxxxxxxxxx',
  secretId: 'AxxxxxxxxxxxxKpww6zbxxxxxxxxx',
  secretKey: 'kFXd00xxxxxp22Awixxxxxxxxxx'
})

// 获取数据库引用(数据实例对象)
var db = cloudDb.database();

module.exports = db
```

封装好之后，我们只需要在 todo.js 中，引入db文件即可；

### 3-3 增删改查操作

#### 添加任务

```
// 添加任务
router.post('/', async (req, res) => {

  // 判断数据长度
  var v = Object.values(req.body);
  if (JSON.stringify(v).length > 500) {
    return res.send('数据太多了')
  }

  // 配置相关参数
  req.body.userid = 1
  req.body.createtime = new Date().getTime().toString()
  req.body.remindtime = ''
  req.body.done = false

  const backDb = await db.collection('todo').add(req.body)

  // 响应数据
  res.send('添加成功')
})
```

```
    req.body._id = backDb.id
    backdata = {status:200,msg:'添加任务成功',data:req.body}
    res.send(backdata)
})
```

## 获取任务

```
// 获取任务
router.get('/', async (req, res) => {
  const backDb = await db.collection('todo').get()

  backdata = {status:200,msg:'获取全部任务成功',data:backDb.data}
  res.send(backdata)
})
```

## 修改任务

```
router.put('/', async (req, res) => {
  if (req.query.id === undefined) {
    res.send('必须传入ID')
    return
  }

  var d = {
    title: req.body.title,
    describe: req.body.describe,
    remindtime: req.body.remindtime ? req.body.remindtime : '',
    done: false,
    files: []
  }

  // 任务状态
  if (req.body.done === '1') {
    d.done = true
  } else if (req.body.done === '0') {
    d.done = false
  } else if (req.body.done === undefined) {
    delete d.done
  } else {
    delete d.done
  }

  // 任务描述
  if (d.describe === undefined) {
    delete d.describe
  }

  const backDb = await db.collection('todo')
    .doc(req.query.id)
    .update(d)
  // res.send(backDb)
```

```
    if (backDb.updated >= 1) {
      backdata = { status: 201, msg: '修改任务成功', data: [] }
    }else{
      backdata = { status: 404, msg: '修改任务失败', data: [] }
    }
    res.send(backdata)
  })
}
```

## 删除任务

```
// 删除任务
router.delete('/', async (req, res) => {
  if (req.query.id === undefined) {
    res.send({ status: 404, msg: '任务 ID 不能为空', data: [] })
    return
  }

  const backDb = await db.collection('todo')
    .doc(req.query.id)
    .remove()

  if(backDb.deleted>=1){
    backdata = { status: 204, msg: '删除任务成功', data: [] }
  }else{
    backdata = { status: 404, msg: '删除任务失败', data: [] }
  }
  res.send(backdata)
})
```

## 3-4 客户端接口调用

这里我们选择使用普通的 Vue 框架作为客户端，按照传统的方式创建，安装好 Element-ui 及 Axios 请求库，就可以直接向云函数发送请求获取数据了；

我这里简单的写了一个请求的示例，发送请求后，渲染到页面中，后续的工作也很简单，就是实现其他的接口功能就行了，一切看起来都是那么美好，但问题就在于它竟然能正常显示，你就没有那么一丝丝的不安吗？当我查看完云服务器的响应头之后，我才放心下来，你知道发生什么了吗？

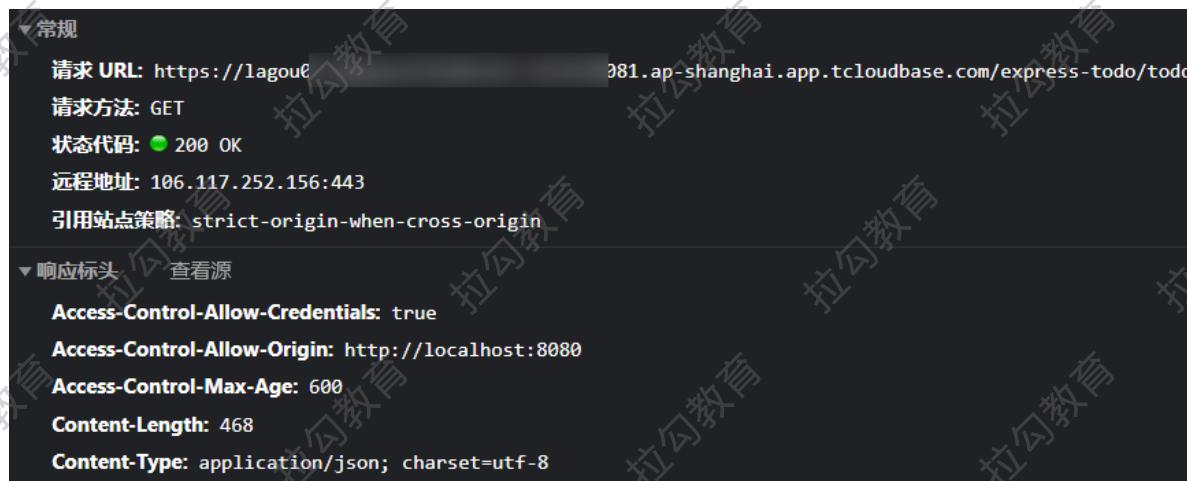
```
<template>
<div>
  <el-card class="box-card">
    <div slot="header" class="clearfix">
      <span>Todo List</span>
    </div>
    <div v-for="(todo, key) in cloudData" :key="key" class="text item">
      <el-checkbox> {{ todo.title }} </el-checkbox>
      <el-divider></el-divider>
    </div>
  </el-card>
</div>
</template>
```

```

<script>
import Axios from "axios";
export default {
  data() {
    return {
      cloudData: [],
    };
  },
  methods: {
    async getData() {
      var backdata = await Axios({
        url:
          "https://lagou02-8gsxxxxx5-111119081.ap-shanghai.app.tcloudbase.com/express-todo/todo",
      });
      console.log(backdata);
      this.cloudData = backdata.data.data;
    },
  },
  mounted() {
    this.getData();
  }
};
</script>

```

没错就是因为跨域了，一开始我以为页面是不可能拿到的数据的，但是，我看到了下面的响应头，还是有些意外的：



这只是一个小小插曲，我相信你也能看的明白，我这里就不细说了，但是代码，我却不想继续写了，因为，这样的开发方式，非常不 Serverless；

前面我们在应用中创建了一个云函数，并将云函数与 Express 进行整合，配合云数据库写好了增删改查的接口，但是这样的开发方式并不是 Serverless 的最佳食用方法，在我们的代码中，是将整个后端应用的全部业务能力，写进了一个函数中，这样做的好处就是方便管理，毕竟在一个应用下只有一个云函数，但是单个云函数的并发是有限的，并行的函数实例个数，由云厂商决定，而超过限制后，事件队列就需要等待其他函数实例执行完毕后，再生成新的函数实例。那可能就有人问了，不是说 Serverless 是弹性伸缩的吗？不是说会根据业务处理的需求自动调配资源嘛？为什么还会有函数的并发限制呢？

要搞清楚这一点，我们需要了解 FaaS 的运行机制；

# 第4章 云开发高阶应用

## 4-1 FaaS 的运行机制

在 FaaS 平台中，函数默认是不运行的，也不会分配任何资源。甚至 FaaS 中都不会保存函数代码。只有当 FaaS 接收到触发器的事件后，才会启动并运行函数。前面我们就是使用 HTTP 的触发器来执行函数代码的，整个函数的运行过程实际上可以分为四个阶段：

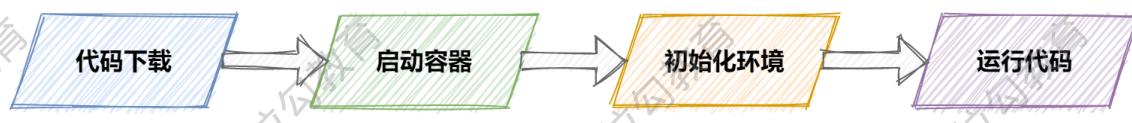
代码下载：FaaS 平台本身不会存储代码，而是将代码放在对象存储中，需要执行函数的时候，再从对象存储中将函数代码下载下来并解压，因此 FaaS 平台一般都会对代码包的大小进行限制，通常代码包不能超过 50MB。

启动容器：代码下载完成后，FaaS 会根据函数的配置，启动对应容器，FaaS 使用容器进行资源隔离。

初始化运行环境：分析代码依赖、执行用户初始化逻辑、初始化入口函数之外的代码等。

运行代码：调用入口函数执行代码。

当函数第一次执行时，会经过完整的四个步骤，前三个过程统称为“冷启动”，最后一步称为“热启动”。



整个冷启动流程耗时可能达到百毫秒级别。函数运行完毕后，运行环境会保留一段时间，这个时间在几分钟到几十分钟不等，这和具体云厂商有关。如果这段时间内函数需要再次执行，则 FaaS 平台就会使用上一次的运行环境，这就是“执行上下文重用”，也叫做“实例复用”，函数的这个启动过程也叫“热启动”。“热启动”的耗时就完全是启动函数的耗时了。当一段时间内没有请求时，函数运行环境就会被释放，直到下一次事件到来，再重新从冷启动开始初始化。

考虑下面这个云函数：

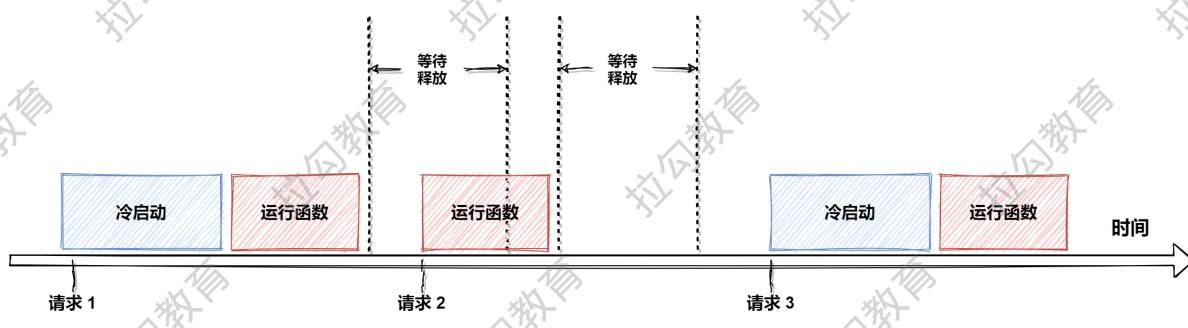
```
let i = 0;
exports.main = async (event = {}) => {
  i++;
  console.log(i);
  return i;
};
```

在第一次调用该云函数的时候函数返回值为 1，这是符合预期的。

但如果连续调用这个云函数，**其返回值有可能是从 2 递增，也有可能变成 1**，这便是实例复用的结果：

- 当**热启动**时，执行函数的 Node.js 进程被复用，进程的上下文也得到了保留，所以**变量 i 自增**。
- 当**冷启动**时，Node.js 进程是全新的，代码会从头完整的执行一遍，**此时返回 1**。

下面是一个函数的请求示意图，其中“请求1”“请求3”是冷启动，“请求2”是热启动。

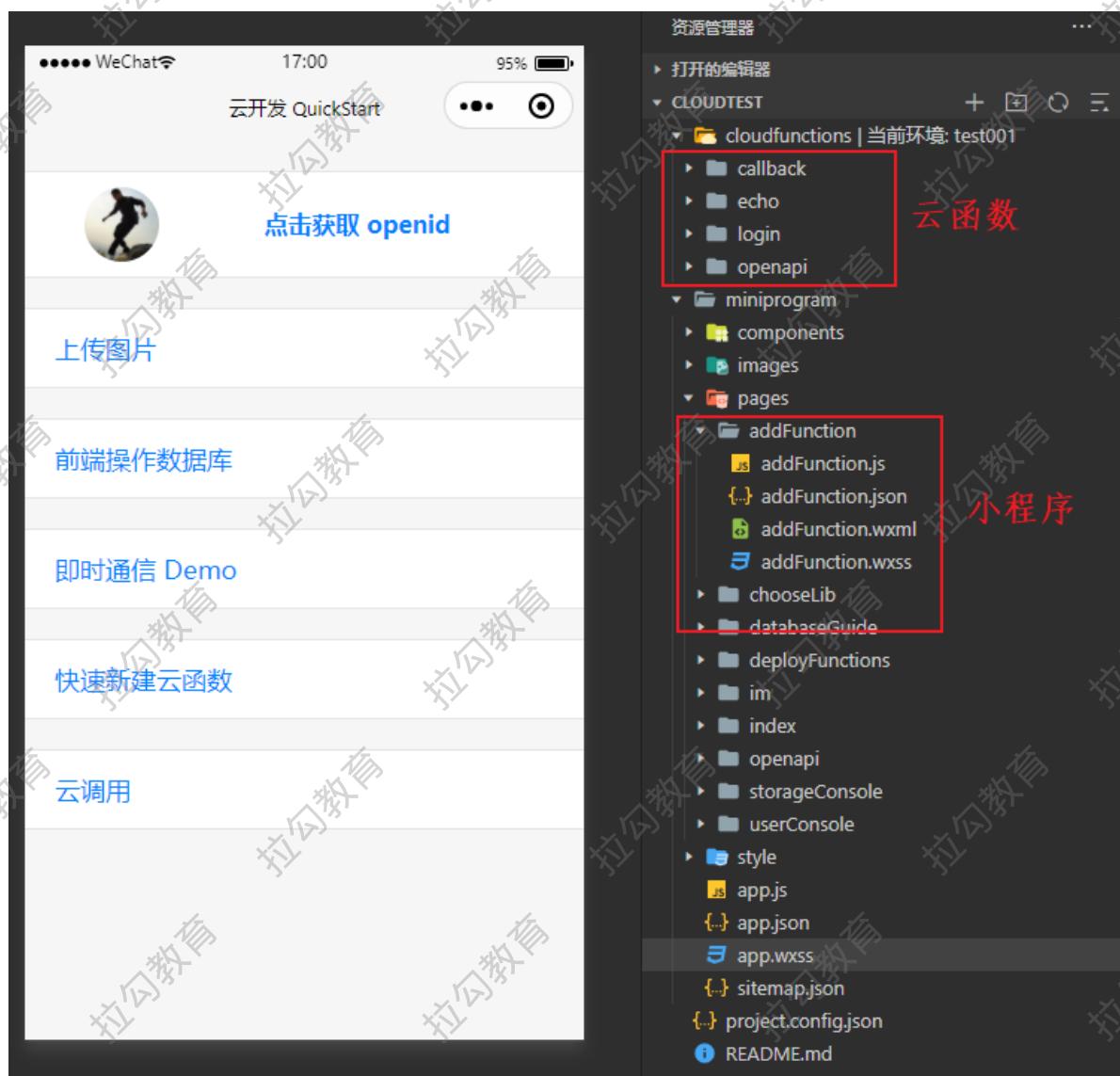


函数执行完毕后销毁运行环境，虽然对首次函数执行的性能有损耗，但极大提高了资源利用效率，只有需要执行代码的时候才初始化环境、消耗硬件资源。并且如果你的应用请求量比较大，则大部分时候函数的执行可能都是热启动。

从函数运行的生命周期中你可以发现，如果函数每分钟都执行，则函数几乎都是热启动的，也就是会重复使用之前的执行上下文。执行上下文就包括函数的容器环境、入口函数之外的代码。

云平台会根据当前负载情况，自动控制云函数实例的数量，并且均衡地分发请求。连续的多次请求有可能由同一实例处理，也可能不是。这就是我们在上面的代码中看到的，`i` 的值非常放肆，根本就找不到规律；所以，我们在编写云函数时，应注意保证云函数是无状态的、幂等的，即当次云函数的执行不依赖上一次云函数执行过程中在运行环境中残留的信息。

再次回到我们的 Todo 案例中，因为我们将全部的业务逻辑放到了一个云函数中，因此，处理的并发量，会受到极大的限制，当并发量达到一定的程度时，无法创建更多的函数实例，也就无法分配更多的服务器资源；更好的方式，是对我们的业务逻辑进行拆分，一个云函数就对应一个独立的业务逻辑处理；这在小程序的云开发中就有体现，默认给我们的小程序云开发模板中，就是一个小程序应用对应对个云函数的处理方式；

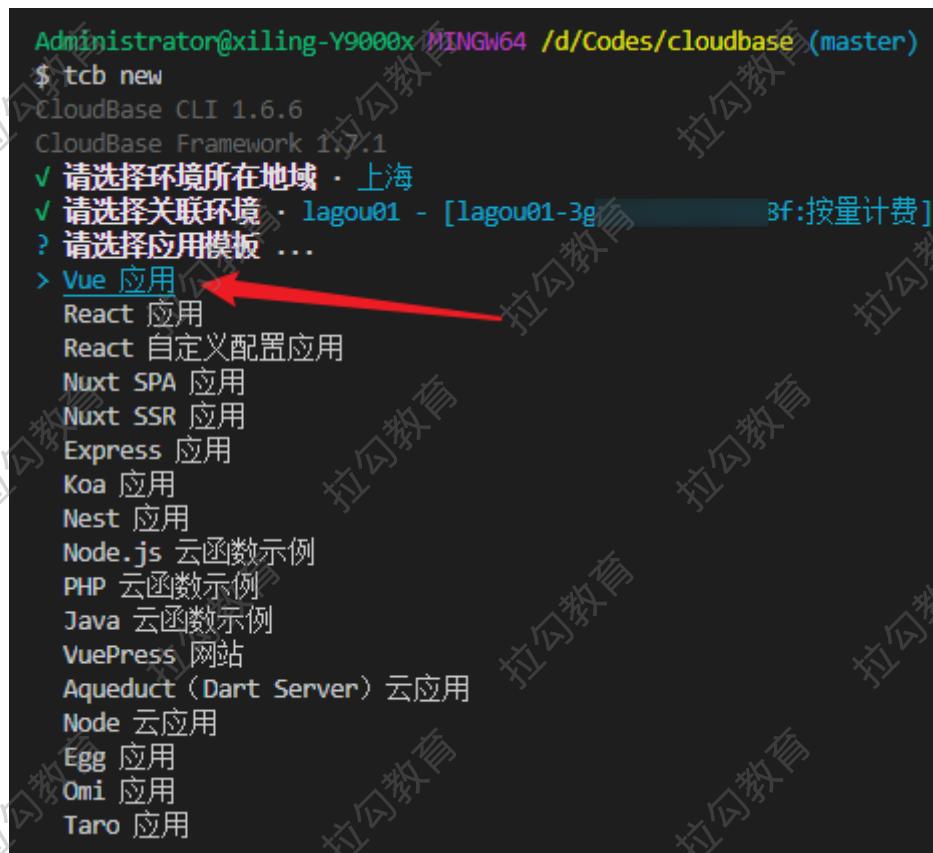


## 4-2 使用 CloudBase Framework

腾讯官方提供的 [cloudbase-framework](#) 工具则给我们提供了一种方式，我们前面使用的 CloudBase CLI 命令行工具，就是使用 cloudbase-framework 的对外接口工具，也就是说，我们使用的命令行，实际就是调用了 cloudbase-framework 提供的功能，前面我们已经使用过一些了，比如：

`tcb new` 创建应用、`tcb` 应用部署、`tcb service create` 创建 http 触发器 增量更新代码；除了这些部署代码相关的命令，framework 还给我提供了一站式管理云平台资源的能力；

接下来，我们按照 Serverless 的开发模式，对 Todo 案例进行重构，在腾讯云开发 CloudBase 下，已经给我们创建好了各种各样的开发的模板，使用 `tcb new` 这个命令就可以看到，在选择应用模板时，选择 Vue 应用，就可以创建一个 Vue 云开发的项目；



```
Administrator@xiling-Y9000x MINGW64 /d/Codes/cloudbase (master)
$ tcb new
CloudBase CLI 1.6.6
CloudBase Framework 1.7.1
✓ 请选择环境所在地域 · 上海
✓ 请选择关联环境 · lagou01 - [lagou01-3g] bf:按量计费]
? 请选择应用模板 ...
> Vue 应用 ← Red arrow pointing to this option
React 应用
React 自定义配置应用
Nuxt SPA 应用
Nuxt SSR 应用
Express 应用
Koa 应用
Nest 应用
Node.js 云函数示例
PHP 云函数示例
Java 云函数示例
VuePress 网站
Aqueduct (Dart Server) 云应用
Node 云应用
Egg 应用
Omi 应用
Taro 应用
```

项目创建后之后，我们能看都项目路径下有 `cloudfunctions` 目录，这就是存放云函数的地方，一个函数就是一个文件夹，那么怎么管理这些函数呢？在项目的根路径下，有一个 `cloudbaserc.json` 的文件，它就是整个应用的 framework 的配置文件，我们可以通过这个配文件来管理我们的项目应用，所以在开始之前，我们吆先来认识一下这个配置文件中，各个配置项的含义：

```
{
  "version": "2.0", // framework 版本，开启配置文件支持动态变量的特性
  "envId": "{{env.ENV_ID}}", // 应用 ID
  "$schema": "https://framework-xxxxxxxxxx.xxxxapp.com/schema/latest.json", // 配置模板的描述信息
  "framework": {
    "name": "techo-show",
    "plugins": []
  },
  "region": "ap-shanghai", // 应用所在地区
}
```

`version` 字段： CLI 0.9.1+ 版本引入了 2.0 新版本配置文件，支持了动态变量的特性。在 `cloudbaserc.json` 中声明 "version": "2.0" 即可启用新的特性，新版配置文件只支持 JSON 格式。动态变量特性允许在 `cloudbaserc.json` 配置文件中使用动态变量，从环境变量或其他数据源获取动态的数据。使用 `{}{}` 包围的值定义为动态变量，可以引用数据源中的值。

`envId` 字段：应用ID

`$schema`：配置模板的描述信息

`region`：应用所在地区

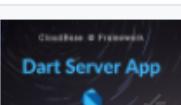
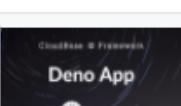
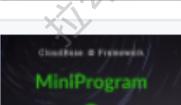
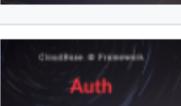
`framework`：配置文件的主要配置项

## **framework** 字段

`name` 属性是应用名字

### **framework.plugins**

这是我们管理应用的重点，[Framework](#) 是支持插件机制的，提供了多种应用框架和云资源的插件。应用依赖哪些插件，都在 `plugins` 参属下配置，`framework` 会根据 `plugins` 的配置来管理应用，处理应用中的构建、部署、开发、调试等流程，一个应用可以使用多个插件，使用不同的自定义属性名字进行管理；官方提供的插件有很多，具体可以查看：<https://docs.cloudbase.net/framework/plugins/>

插件链接	拉勾教育 插件	最新版本	插件介绍
	<a href="#">@cloudbase/framework-plugin-website</a>	npm v1.7.1 [ ]	一键部署网站应用
	<a href="#">@cloudbase/framework-plugin-node</a>	npm v1.7.1 [ ]	一键部署 Node 应用 (支持底层部署为函数或者云托管)
	<a href="#">@cloudbase/framework-plugin-nuxt</a>	npm v1.7.1 [ ]	一键部署 Nuxt SSR 应用
	<a href="#">@cloudbase/framework-plugin-function</a>	npm v1.7.1 [ ]	一键部署函数资源
	<a href="#">@cloudbase/framework-plugin-container</a>	npm v1.7.1 [ ]	一键部署云托管容器服务
	<a href="#">@cloudbase/framework-plugin-dart</a>	npm v1.7.1 [ ]	一键部署 Dart 应用
	<a href="#">@cloudbase/framework-plugin-database</a>	npm v1.7.1 [ ]	一键声明式部署云开发 NoSQL 云数据库
	<a href="#">@cloudbase/framework-plugin-deno</a>	npm v1.7.1 [ ]	一键部署 Deno 应用
	<a href="#">@cloudbase/framework-plugin-next</a>	npm v1.7.1 [ ]	一键部署 Next SSR 应用
	<a href="#">@cloudbase/framework-plugin-mp</a>	npm v1.7.1 [ ]	一键部署微信小程序应用
	<a href="#">@cloudbase/framework-plugin-auth</a>	npm v1.7.1 [ ]	一键设置登录配置

## 云函数插件

首先我们对之前写好的云函数进行插件方式的修改：

```
"server": { // 自定义插件名字
  "use": "@cloudbase/framework-plugin-function", // 引入使用的插件
  "inputs": { //对于当前插件的配置信息
    "functionRoot": "./functions", //函数代码的本地路径
    "functions": [ //云函数的配置信息
      {
        "name": "hello",
        "config": {
          "handler": "index.main",
          "trigger": "cloud_function"
        }
      }
    ]
  }
}
```

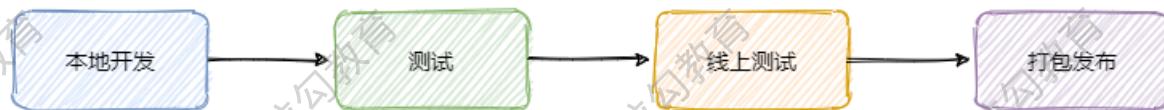
```
{  
    "name": "express-todo", // 云函数的名字  
    "timeout": 5, // 运行超时时间  
    "envVariables": {}, // 环境变量的键值对对象  
    "runtime": "Nodejs10.15", // 运行时环境(编程语言)  
    "memorySize": 128, // 运行最大分配内存 M 单位  
    "handler": "index.main" // 函数入口  
}  
}  
},
```

配置完成后，修改代码，然后进行部署测试；

```
CloudBase Framework info EnvId lagou Validate config file success.  
CloudBase Framework info AppName node-starter  
CloudBase Framework info ◆ install plugins  
CloudBase Framework info callHooks 'preDeploy'  
CloudBase Framework info ◆ init: server...  
CloudBase Framework info ◆ build: server...  
CloudBase Framework info ◆ compile: server...  
CloudBase Framework info callHooks 'postCompile'  
正在部署 [██████████] 100% 0.0 s  
CloudBase Framework info ◆ deploy: server...  
CloudBase Framework info ◆ [express-todo] 云函数部署成功  
CloudBase Framework info ◆ 云函数部署成功  
CloudBase Framework info callHooks 'postDeploy'  
CloudBase Framework info ✨ done
```

## 静态网站插件

云函数配置好之后，回到我们的客户端代码中，正常的开发部署流程是：



在云开发中，有一个静态站点托管的服务，我们可以借助 静态网站插件，一键完成打包上线部署的全部工作，不用再手动完成了；

配置也很简单：

```
"client": {  
  "use": "@cloudbase/framework-plugin-website", // 引入使用的插件  
  "inputs": {  
    "buildCommand": "npm run build", // 本地执行的打包命令  
    "outputPath": "dist", // 静态文件地址  
    "cloudPath": "/", // 静态网站托管的代码路径  
    "envVariables": {  
      "VUE_APP_ENV_ID": "{{env.ENV_ID}}"  
    }  
  }  
}
```

## 4-3 云函数 Todo 重构

搞清楚各个配置的含义之后，我们就可以按照之前的思路，实现 Todo 应用了；

先配置一个添加任务的函数，

```
"functionRootPath": "cloudfunctions",  
  "functions": [  
    {  
      "name": "addtodo",  
      "timeout": 3,  
      "envVariables": {},  
      "runtime": "Nodejs10.15",  
      "memory": 128,  
      "aclRule": {  
        "invoke": true  
      }  
    }  
  ]
```

注意，配置文件不能帮我们在本地创建文件及文件夹，不具备小程序的能力，所以，写好配置文件，我们需要自己创建对应的代码目录及文件；

/cloudfunctions/add/index.js

```
const cloud = require("./clouddb");  
  
async function addTodo(event){  
  var req = event;  
  if(req.title == undefined){  
    return ;  
  }  
  var todo = {  
    title:req.title,  
    createTime:Date.now(),  
    done:false  
  }  
  var backdata = await cloud.collection('todo').add(todo);  
  return backdata
```

```
}

exports.main = async (event, context) => {
    return addTodo(event)
};

}
```

/cloudfunctions/add/cloud.js

```
const nodesdk = require('@cloudbase/node-sdk')
const cloudDb = nodesdk.init({
    env: 'guanyingguanzhu'xilinglaoshi',
    secretId: 'AKIDAakFYgxilinglaoshixbnimLOKVR',
    secretKey: 'vf9MLxilinglaoshioHxilinglaoshiy'
})

const app = cloudDb.database();
module.exports = app
```

代码写好之后，如何进行本地测试呢？还记得之前我们使用的 SCF 这个工具吗？没错，就用它就行了；

```
Administrator@xiling-Y9000x MINGW64 /d/Codes/cloudbase/todos (master)
$ scf init
? 请输入入口文件地址(相对路径) ./cloudfunctions/add
? 请输入入口执行方法名称 main
? 请输入超时时间限制(单位: s) 3
? 请选择测试模版 http
[Weapp CLI][2021-04-13T20:23:44+08:00] Server has listened [IP]:localhost [PORT]:3000, http://localhost:3000
✓ [Weapp CLI][2021-04-13T20:24:16+08:00] SCF运行状态: resolved
[Weapp CLI][2021-04-13T20:24:16+08:00] SCF运行结束
[Weapp CLI][2021-04-13T20:24:16+08:00] 运行错误: null
[Weapp CLI][2021-04-13T20:24:16+08:00] 运行结果: {"id": "17453ede60758d7001781bae3a75896d", "requestId": "178cb307de8_1"}
[Weapp CLI][2021-04-13T20:24:16+08:00] 进程返回码: 0
[Weapp CLI][2021-04-13T20:24:16+08:00] 日志内容: 没有日志输出
```

然后再使用 Postman 发个请求试试：

POST http://localhost:3000/

Params Auth Headers (10) Body Body Tests Settings Cookies

x-www-form-urlencoded

KEY	VALUE	DESCRIPTION	Bulk Edit
<input checked="" type="checkbox"/> title	学习PHP		
Key	Value	Description	

Body

Pretty Raw Preview Visualize JSON

1 {  
2 "id": "17453ede60758d7001781bae3a75896d",  
3 "requestId": "178cb307de8\_1"  
4 }

200 OK 1260 ms 239 B Save Response

测试完成后，我们可以使用 tcb 命令进行全量部署，注意，全量部署时，vue 也会跟随打包并部署到静态站点中，如果只想部署单个云函数，

```
tcb fn code update xxx函数名
```

可以使用命令 `tcb fn deploy add` 对 add 这个函数单独部署：

```
Administrator@xiling-Y9000x MINGW64 /d/Codes/cloudbase/todos (master)
$ tcb fn deploy add
CloudBase CLI 1.6.7
CloudBase Framework 1.7.3
  i 未找到函数发布配置，使用默认配置 -> 运行时：Nodejs10.15/在线安装依赖
  ✓ [add] 云函数部署成功！

  i 控制台查看函数详情或创建HTTP 访问服务链接 🔍 : https://console.cloud.tencent.com/tcb/scf?envId=todos
  i 使用 cloudbase functions:list 命令查看已部署云函数
```

部署完成后可以登录云控制台查看，也可以在本地使用 `tcb fn list` 查看已部署的函数列表；

```
Administrator@xiling-Y9000x MINGW64 /d/Codes/cloudbase/todos (master)
$ tcb fn list
CloudBase CLI 1.6.7
CloudBase Framework 1.7.3
```

函数 Id	函数名称	运行时	创建时间	修改时间	状态
lam-f2poa7fm	add	Nodejs10.15	2021-04-13 20:40:46	2021-04-13 20:40:46	部署完成

获取数据

```
"functionRootPath": "cloudfunctions",
"functions": [
  {
    "name": "gettodo",
    "timeout": 3,
    "envVariables": {},
    "runtime": "Nodejs10.15",
    "memory": 128,
    "aclRule": {
      "invoke": true
    }
  },
  {
    "name": "addtodo",
    "timeout": 3,
    "envVariables": {},
    "runtime": "Nodejs10.15",
    "memory": 128,
    "aclRule": {
      "invoke": true
    }
  }
]
```

```
const app = require('./clouddb')

async function showTodo(){
  const backdb = await app.collection('todo').get();
  return backdb
}

exports.main = async (event, context) => {
  return showTodo();
};
```

## 4-4 Vue 客户端调用

在 Vue 中调用云函数，与传统方式不一样的是，我们不需要自己发送 HTTP 请求，腾讯官方封装了 Vue 插件：vue-provider：<https://github.com/TencentCloudBase/cloudbase-vue>，在我们构建的项目中，已经引入了，在 main.js 中修改我们的环境参数，就可以使用了；

```
import Vue from "vue";
import App from "./App.vue";
import Cloudbase from "@cloudbase/vue-provider";

import ElementUI from 'element-ui';
import 'element-ui/lib/theme-chalk/index.css';

Vue.use(ElementUI);
window._tcbEnv = window._tcbEnv || {};

export const envId = window._tcbEnv.TCB_ENV_ID;
export const region = window._tcbEnv.TCB_REGION;

Vue.config.productionTip = false;

Vue.use(Cloudbase, {
  env: 'lagoxilingtaoshizhenshui',
  region: 'ap-shanghai'
});

new Vue({
  render: h => h(App)
}).$mount("#app");
```

同时，在index.html中，还会默认加载一个静态的配置文件 `_init_tcb-env.js`，其实就是环境的配置参数，因为我们已经在 main.js 配置了环境参数，因此，直接屏蔽这个文件即可；

完成这些配置之后，我们在 Vue 中完成添加任务的功能，进行测试，但是，这里有个坑，腾讯在这个地方，小学生收割机，收割了我1个小时的时光，我才搞清楚，callFunction 调用的传参与 HTTP 触发器调用的 event 入参是不一样的，一定注意(手册中没有写) <https://docs.cloudbase.net/cloud-function/how-works.html>，callFunction 调用的云函数 event 的入参就是传入 callFunction 的 data，没有请求信息数据，所以，云函数的代码一定记得修改；

```
<div class="box-card">
  <el-card shadow="always">
    <el-input v-model="addData.title" placeholder="请输入内容">
      <el-button slot="append" @click="addTodo">添加任务</el-button>
    </el-input>
  </el-card>
</div>
```

```
async addTodo() {
  var todoData = this.addData;
  var back = await this.$cloudbase.callFunction({
    name: "addtodo",
    data: todoData,
  });
  console.log(back);
},
```

此时，我们会收到一个没有权限的报错，这是因为，调用云函数必须要进行登录鉴权。我们暂时先使用匿名登录的方式，调通接口的数据通信，后面我们再详细接受 Cloudbase 的用户管理服务器，但是，就算是使用匿名登陆也是个坑，控制台中登录鉴权的实例代码是错误的，正确的代码示例在文档中心哪里 <https://docs.cloudbase.net/authentication/anonymous.html>

```
mounted() {
  // 使用匿名登录
  var auth = this.$cloudbase.auth();
  auth.anonymousAuthProvider().signIn();
},
```

当然，光有代码还不够，你还需要到控制台中，开启应用允许匿名登录的选项才行，不过一般都是默认就开通的，这里就不再细说了，通过登录验证后，我们终于可以去骂骂腾讯写的垃圾文档了；

继续完善获取数据的功能：

```
<el-card class="box-card">
  <div v-for="(todo, key) in todos" :key="key" class="text item">
    <template v-if="todo.done == false">
      <div class="lists">
        <el-checkbox v-model="todo.done" @change="todoDone(key)">
          {{ todo.title }}
        </el-checkbox>
        <div>
          <i class="el-icon-notebook-2" @click="showTodo(key)"></i> |
          <i class="el-icon-delete" @click="deleteTodo(key)"></i>
        </div>
      </div>
      <el-divider></el-divider>
    </template>
  </div>
</el-card>

<el-drawer title="任务详情" :visible.sync="drawer">
```

```
<div class="dra">
  <p>任务: {{ drawerTodo.title }}</p>
  <p>附件:<br />
    
  </p>
  <p>
    <input type="file" ref="fil" />
    <el-button @click="upFile(drawerTodo._id)"> 点击上传 </el-button>
  </p>
</div>
</el-drawer>
```

```
// 获取数据
async getData() {
  var back = await this.$cloudbase.callFunction({
    name: "gettodo",
  });
  this.todos = back.result.data;
  console.log(this.todos);
},

// 展示详情
async showTodo(key) {
  var todoInfo = this.todos[key];

  console.log(todoInfo.fileId);
  var back = await this.$cloudbase.getTempFileURL({
    fileList: [todoInfo.fileId],
  });

  this.drawerTodo = todoInfo;
  this.drawerTodo.tempfile = back.fileList[0].tempFileURL;
  this.drawer = true;
},
```

## 4-5 文件上传 & 云对象存储

```
async upFile(id) {
  var s = this.$refs.fil.files[0];
  // uploadFile 就是云对象存储提供的接口，集成到了vue 云开发插件中
  var back = await this.$cloudbase.uploadFile({
    // 云存储的路径
    cloudPath: "todo/" + Date.now() + s.name,
    // 需要上传的文件，File 类型
    filePath: this.$refs.fil.files[0],
  });
  console.log(id, back);
  //
```

```
var changeData = {
  _id: id,
  fileId: back.fileID,
};

var changeBack = await this.$cloudbase.callFunction({
  name: "puttodo",
  data: changeData,
});

console.log(changeBack);
},
```

## 4-6 用户管理及登录授权服务

### 4-6-1 基础架构代码

开始之前，先完成基础的代码和功能搭建，引入路由，完成注册登录的页面和对应的表单；

```
$ npm install vue-router
```

添加路由文件，\src\router\index.js

```
import Vue from 'vue'
import VueRouter from 'vue-router'

Vue.use(VueRouter)

const routes = [
  {
    path: '/',
    name: 'Index',
    component: () => import('../components/Index.vue')
  },
  {
    path: '/register',
    name: 'Register',
    component: () => import('../components/Register.vue')
  },
  {
    path: '/login',
    name: 'Login',
    component: () => import('../components/Login.vue')
  }
]

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

export default router
```

在入口文件main.js 中引入并注册路由：

```
import router from './router'

.....
new Vue({
  router,
  render: h => h(App)
}).$mount("#app");
```

完成对应的单文件组件代码：

注册组件 src\components\Register.vue

```
<template>
<div class="box-card">
  <el-row>
    <el-col>
      <el-card shadow="always">
        <el-form label-position="left" label-width="80px" :model="user">
          <el-form-item label="手机号">
            <el-input v-model="user.phone">
              <template slot="append">
                <el-button size="mini" :disabled="disa" @click="sendCode">
                  {{sendCodeMsg}}
                </el-button>
              </template>
            </el-input>
          </el-form-item>
          <el-form-item label="验证码">
            <el-input v-model="user.code"></el-input>
          </el-form-item>
          <el-form-item label="密码">
            <el-input v-model="user.pwd"></el-input>
          </el-form-item>
          <el-form-item>
            <el-button size="primary" @click="sendRegister">注册</el-button>
          </el-form-item>
        </el-form>
      </el-card>
    </el-col>
  </el-row>
</div>
</template>

<style>
.box-card {
  margin: 0 auto;
  width: 580px;
  margin-bottom: 20px;
}
</style>
```

登录组件：\src\components\Login.vue

```

<template>
<div class="box-card">
  <el-row>
    <el-col>
      <el-card shadow="always">
        <el-form label-position="left" label-width="80px" :model="user">
          <el-form-item label="手机号">
            <el-input v-model="user.phoneNumber"></el-input>
          </el-form-item>
          <el-form-item label="密码">
            <el-input v-model="user.password"></el-input>
          </el-form-item>
          <el-form-item>
            <el-button size="primary" @click="Login">登录</el-button>
          </el-form-item>
        </el-form>
      </el-card>
    </el-col>
  </el-row>
</div>
</template>

<style>
.box-card {
  margin: 0 auto;
  width: 580px;
  margin-bottom: 20px;
}
</style>

```

## 4-6-2 注册逻辑

在控制台开启“短信验证码登录”的选项

登录方式	启用状态	描述
微信公众号	<input type="checkbox"/>	
微信开放平台	<input type="checkbox"/>	
匿名登录	<input checked="" type="checkbox"/>	启动匿名登录后，用户将不需要登录即可访问应用。详细参考 <a href="#">匿名登录方式</a> 。
未登录	<input checked="" type="checkbox"/>	允许未登录后，用户将不需要登录即可访问应用。
邮箱登录	<input checked="" type="checkbox"/>	email:
自定义登录	<input type="checkbox"/>	若您未开启MFA，下载私钥操作将不受保护。 <a href="#">绑定MFA</a>
用户名密码登录	<input type="checkbox"/>	启动用户名密码登录
短信验证码登录	<input checked="" type="checkbox"/>	可通过短信验证码快速登录

短信验证使用的是 js-sdk，手册在这里 [登录认证 | 云开发 CloudBase - 一站式后端云服务](#)，所以先安装：`npm install @cloudbase/js-sdk`

因为我们需要在多个地方使用，因此先进行封装，这里选在使用 Vue 插件的方式：

\src\assets\auth.js

```
import cloudbase from "@cloudbase/js-sdk";

// 自定义插件对象
var auths = {};
auths.install = function (vue) {
  const app = cloudbase.init({
    env: "lagou01-5gxilinglaoshi72a1",
  });
  const myauth = app.auth()
  vue.prototype.$auths = myauth;
}
// 导出插件
export default auths;
```

不要忘记在入口文件中导入: \src\main.js

```
import router from './router'

import auths from './assets/auth.js'
Vue.use(auths) // 注册使用 auth 插件

import ElementUI from 'element-ui';
import 'element-ui/lib/theme-chalk/index.css';
```

首先完成手机验证码的发送

```
// 发送注册验证码
async sendCode() {
  const res = await this.$auths.sendPhoneCode(this.user.phone);
  if (res) {
    // 验证码发送成功
    this.disabled = true
    this.sendCodeMsg = "有效期5分钟"
  } else {
    console.log("短信发送失败");
  }
},
```

用户输入验证码及密码，进行验证码及手机号的验证；

```
// 验证码密码注册
async sendRegister(){
  const res = await
this.$auths.signupWithPhoneCode(this.user.phone, this.user.code, this.user.pwd)
  if(res){
    console.log('注册成功')
    this.$router.push({path: '/login'})
  }
}
```

验证注册成功后，跳转到登录界面

### 4-6-3 登录逻辑

登录验证是非常简单的，那么，我们是如何保持登录状态的呢？

```
export default {
  data() {
    return {
      user: {
        phoneNumber: '',
        password: '',
      },
    };
  },
  methods: {
    async Login() {
      const res = await this.$auths.signInWithPhoneCodeOrPassword(this.user);
      if (res) {
        this.$router.push({path: "/"})
      }
    },
  },
};
```

登录状态的保持有三种不同的方式，[登录认证 | 云开发 CloudBase - 一站式后端云服务](#)

local：在显式退出登录之前的 30 天内保留身份验证状态

session：在窗口关闭时清除身份验证状态

none：在页面重新加载时清除身份验证状态

在初始化调用auth方法时，传入：`\src\assets\auth.js`

```
import cloudbase from "@cloudbase/js-sdk";

// 自定义插件对象
var auths = {};
auths.install = function (vue) {
  const app = cloudbase.init({
    env: "lagou01-5gxilinglaohsi872a1",
  });

  // persistence: 身份认证状态如何持久保留
  // local: 在显式退出登录之前的 30 天内保留身份验证状态
  // session: 在窗口关闭时清除身份验证状态
  // none: 在页面重新加载时清除身份验证状态
  const myauth = app.auth({
    persistence: "local" // 用户显式退出或更改密码之前的30天一直有效
  })
  vue.prototype.$auths = myauth;
}

// 导出插件
export default auths;
```

不同的登录状态都可以在浏览器的控制台的“Application”中查看，那么在不同的组件中，如何获取登录状态和登录的数据呢？

auth 对象中，有 getLoginState 方法，看名字也知道，是获取登录状态的，我们在首页中使用挂载的生命周期函数进行验证，\src\components\Index.vue，

```
async mounted() {
  var hls = await this.$auths.getLoginstate();
  if(hls == null){
    this.$router.push({path:"/login"})
  }
  this.getdata();
},
```

当然，你也可以使用 Vue-router 提供的 导航守卫 进行全局的登录状态验证；

## 第5章 上线部署



申请SSL证书，添加 CNAME 记录执行服务器地址即可；