

云原生时代的微服务架构实践

前言

写作本书的目的

微服务架构已经火了很多年了，如：Dubbo、Spring Cloud，再到后来的 Spring Cloud Alibaba，但都是仅限于 Java 语言的瓶颈，如何让各种语言之间的微服务更加有效、快速的通讯，这是当前很多企业需要面临的问题，因为一个企业中，不只是基于单纯的某一种语言开发，这就涉及到多语言服务之间的访问。本书的创作重点，则是在于讲述在巨多语言的情况下，该如何设计微服务架构，以及云原生时代的微服务的高可用、自动化等等。

如何阅读本书

由于本书介绍的一些知识都是比较流行的，最近几年很火的，而且本身其涉及的面还是很广的：K8s，大家可以通过其官网(<https://kubernetes.io/>)系统的学习其相关技术与实践，以便可以更好的将其发挥于微服务架构设计中。另外，本书中介绍的知识，大家都可以一边实践、一边阅读，以便更深刻的理解其原理。

扫码关注作者

- 作者毕业于三流院校，笔名：Damon，拥有七年工作经验，技术爱好者，长期从事 Java 开发、Spring Cloud、Golang 的微服务架构设计，以及结合 Docker、K8s 做微服务容器化，自动化部署等一站式项目落地。目前主要从事基于 K8s 云原生架构研发的工作。Golang 语言开发，长期研究边缘计算框架 KubeEdge、调度框架 Volcano 等。公众号：**程序猿Damon**，个站：www.damon8.cn。
- 欢迎扫码回复「入群」加入技术交流群



目录

第一部分 基础知识

第一章 什么是微服务架构

1.1 微服务到底是啥

1.2 微服务的发展史

1.3 微服务的革命性与重要性

第二章 微服务的拆分

2.1 微服务的设计原则

2.2 微服务划分的粒度

2.3 不同场景的微服务

第三章 容器化技术

3.1 什么是容器

3.2 容器的发展进程

3.3 Docker 与 Kubernetes

3.4 K8s 容器化应用

第四章 为何借助容器助力微服务

4.1 微服务的多语言性

4.2 微服务的高可用

4.3 微服务的复杂性

第二部分 原理与应用

第五章 Kubernetes 介绍

5.1 Kubernetes 的基本概念与特性

5.2 部署 Kubernetes 集群

5.3 Kubernetes 的组件及及负载均衡

第六章 为什么选择 Kubernetes

6.1 Kubernetes 与微服务的天生绝配

6.2 基于 Kubernetes 集群的服务治理

6.3 基于 Kubernetes 的服务无缝迁移

第七章 第一个基于 K8s 的多语言微服务架构

7.1 基于 K8s 的 Java 微服务

7.2 第一个Golang微服务

7.3 部署微服务应用

正文

第一部分 基础知识

第一章 什么是微服务架构

1.1 微服务的发展史

在微服务到来之前，单体应用程序所暴露的缺点主要有：

- 复杂性高
- 团队协作开发成本高
- 扩展性差
- 部署效率低下
- 系统很差的高可用性

复杂性，体现在：随着业务的不断迭代，项目的代码量急剧的增多，项目模块也会随着而增加，整个项目就会变成的非常复杂。

开发成本高，体现在：团队开发几十个人在修改代码，然后一起合并到同一地址分支，打包部署，测试阶段只要有一小块功能有问题，就得重新编译打包部署，重新测试，所有相关的开发人员都得参与其中，效率低下，开发成本极高。

扩展性差，体现在：在新增功能业务的时候，代码层面会考虑在不影响现有的业务基础上编写代码，提高了代码的复杂性。

部署效率低，体现在：当单体应用的代码越来越多，依赖的资源越来越多时，应用编译打包、部署测试一次，需要花费的时间越来越多，导致部署效率低下。

高可用差，体现在：由于所有的业务功能最后都部署到同一个文件，一旦某一功能涉及的代码或者资源有问题，那就会影响到整个文件包部署的功能。举个特别鲜明的示例：上世纪八、九十年代，很多的黄页以及延伸到后来的网站中，很多的展示页面与获取数据的后端都是在一个服务模块中。这就造成一个很不好的影响：如果只是修改极小部分的页面展示或图片展示，则需要把整个服务模块进行打包部署，这样会导致时间的严重浪费以及成本的增加。更加糟糕的是，给用户带来非常不好的体验，用户无法理解的是：只是换个网站的某块微小的展示区，导致了整个网站在那一时刻无法正常的访问。当然，也许，对于那个时候互联网的不发达，人们对于这样的体验，已经算是一种幸福的享受了。

由于单体应用具有以上的种种缺点，导致了一个新名词、新概念的诞生：微服务。

1.2 微服务到底是啥

其实，从早年间的单体应用，到 2014 年起，得益于以 Docker 为代表的容器化技术的成熟以及 DevOps 文化的兴起，服务化的思想进一步演化，演变为今天我们所熟知的微服务。那么，微服务到底是啥？

微服务，英文名：microservice，百度百科上将其定义为：SOA 架构的一种变体。微服务（或微服务架构）是一种将应用程序构造为一组低耦合的服务。

微服务有着一些鲜明的特点：

- 功能单一
- 服务粒度小
- 服务间独立性强
- 服务间依赖性弱
- 服务独立维护
- 服务独立部署

对于每一个微服务来说，其提供的功能应该是单一的；其粒度很小的；它只会提供某一业务功能涉及到的相关接口。如：电商系统中的订单系统、支付系统、产品系统等，每一个系统服务都只是做该系统独立的功能，不会涉及到不属于它的功能逻辑。

微服务之间的依赖性应该是尽量弱的，这样带来的好处是：不会因为单一系统服务的宕机，而导致其它系统无法正常运行，从而影响用户的体验。同样以电商系统为例：用户将商品加入购物车后，提交订单，这时候去支付，发现无法支付，此时，可以将订单进入待支付状态，从而防止订单的丢失和用户体验的不友好。如果订单系统与支付系统的强依赖性，会导致订单系统一直在等待支付系统的回应，这样会导致用户的界面始终处于加载状态，从而导致用户无法进行任何操作。

当出现某个微服务的功能需要升级，或某个功能需要修复 bug 时，只需要把当前的服务进行编译、部署即可，不需要一个个打包整个产品业务功能的巨多服务，独立维护、独立部署。

上面描述的微服务，其实突出其鲜明特性：**高内聚、低耦合**，问题来了。什么是高内聚，什么是低耦合呢？所谓高内聚：就是说每个服务处于同一个网络或网域下，而且相对于外部，整个的是一个封闭的、安全的盒子。盒子对外的接口是不变的，盒子内部各模块之间的接口也是不变的，但是各模块内部的内容可以更改。模块只对外暴露最小限度的接口，避免强依赖关系。增删一个模块，应该只会影响有依赖关系的相关模块，无关的不应该受影响。

所谓低耦合：从小的角度来看，就是要每个 Java 类之间的耦合性降低，多用接口，利用 Java 面向对象编程思想的封装、继承、多态，隐藏实现细节。从模块之间来讲，就是要每个模块之间的关系降低，减少冗余、重复、交叉的复杂度，模块功能划分尽可能单一。

1.3 微服务的革命性与重要性

上一小节讲述了什么是微服务，微服务的鲜明特性。其实，从单体应用看微服务，就能看出微服务的重要性，它是彻底改革了应用程序的惯性，它的设计理念的出现：让开发人员减少大量的开发成本以及修复成本；让产品的使用者拥有一种舒适的体验感。它解决了单体应用程序的很多难以解决的问题，更具有创新性。它让我们的系统尽可能快地响应变化。

微服务将原来耦合在一起的复杂业务拆分为单个服务，规避了原本复杂度无止境的积累，每一个微服务专注于单一功能，并通过定义良好的接口清晰表述服务边界。

由于微服务具备独立的运行进程，所以每个微服务可以独立部署。当业务迭代时只需要发布相关服务的迭代即可，降低了测试的工作量同时也降低了服务发布的风险。

在微服务架构下，当某一组件发生故障时，故障会被隔离在单个服务中。如通过限流、熔断等方式降低错误导致的危害，保障核心业务的正常运行。

第二章 微服务的拆分

2.1 微服务的设计原则

- 高内聚、低耦合

紧密关联的事应该放在一起，每个服务是针对一个单一职责的业务能力的封装，需要专注于做好一件事情。这样避免内容耦合，降低代码的冗余，提高代码的可复用性。

避免服务之间的数据库的共享。这样既可以减少数据库的并发操作，又可以避免死锁的出现。通常微服务不会直接共用一个数据库，可以通过主、从表的方式来进行，结合读、写分离，实现数据的共享。

微服务之间应该是轻量级的通信方式，主要是为了解耦，降低业务的复杂性，以及服务的负载。一个服务调用另一个服务应该是不受到后者的牵制，如果后者发生宕机，前者应该照样继续运行下去，这才是微服务设计时的合理安排。为了降低前者的不受牵制，这就需要轻量级的通信，比如：异步调用、重试策略等。

- 以业务为中心

每个服务代表了特定的业务逻辑，不应该掺着其他业务的逻辑，或微不足道的、公共的逻辑。

围绕业务开展，主要就当前业务进行扩展。

能快速的响应业务的变化，需要做到接口的兼容性，兼容业务场景的变化，这样减少变动太大带来的风险。

隔离实现细节，让业务领域可以被重用，需要以封装接口形式来实现业务逻辑，这就涉及到代码的复用性。引用 Java 的原理：封装、继承、多态。

- 弹性容错设计

设计可容错的系统：拥抱失败，为已知的错误而设计，这主要是为了增强交互。

可防御的系统：服务降级、服务隔离、请求限制、防止级联错误等，这也是为了增强交互的友好。

- 自治和高可用

独立开发和业务扩展，这就涉及到服务随着业务的发展而进行兼容、合理的扩展后的高度自治。

独立部署、运行和高可用，避免单点的孤注一掷。

- 日志与监控

聚合系统日志、数据，从而当遇到问题时，可以深入分析原因。

当需要重现问题时，可以根据日志以及监控来复盘。

监控主要包括服务状态、请求流量、调用链、API 错误计数，结构化的日志、服务依赖关系可视化等内容，以便发现问题及时修复，实时调整系统负载，必要时进行服务降级，过载保护等等，从而让系统和环境提供高效高质量的服务。

- 自动化

降低部署和发布的难度，如：在持续集成和持续交付中，自动化编译，测试，安全扫描，打包，集成测试，部署。随着服务越来越多，在发布过程中，需要进一步自动化金丝雀部署。

利用 K8s 等进行服务自动弹性伸缩等。

2.2 微服务划分的粒度

服务的划分，可以从水平的功能划分，也可从垂直的业务划分，粒度的大小，可以根据当前的产品需求来定位，最关键的是要做到：**高内聚、低耦合**。

如电商系统为例，如下图：



电商中涉及到业务很可能是最多的，商品、库存、订单、促销、支付、会员、购物车、发票、店铺等等，这个是根据业务的不同来进行模块的划分。微服务划分的粒度一定是要有明确性的，不能因为含糊而新增一个服务模块，这样会导致功能接口的可复用性差。一个好的架构设计，肯定是可复用性很强的结构模式。我喜欢这样的一句话：**微服务的边界(粒度)是“决策”，而不是个“标准答案”。**即应该将各微服务划分的方式，深度思考，周全的考量各方面的因素下，所作出的一个“最适合”的架构决策，而不是一个人芸芸的“标准答案”。

2.3 不同场景的微服务

微服务的应用场景也是很多的，电商场景是比较常见的，比如阿里的体系：淘宝、支付宝、钉钉、饿了么、咸鱼、口碑等。电商场景的微服务实相对比较复杂的，所以需要更好的做好微服务的拆分以及扩展。

金融系统中也存在微服务的场景，比如：银行系统、证券系统、金融公司、机构。其需要考虑的重点是微服务的安全性、可靠性。

第三章 容器化技术

3.1 什么是容器

什么是容器呢？自然界的解释：容器是指用以容纳物料并以壳体为主的基本装置。但今天讲的容器也是一个容纳物质的载体。那计算机所指的容器(Container)到底是什么呢？容器是镜像 (Image) 的运行时实例。正如从虚拟机模板上启动 VM 一样，用户也同样可以从单个镜像上启动一个或多个容器。虚拟机和容器最大的区别是容器更快并且更轻量级，与虚拟机运行在完整的操作系统之上相比，容器会共享其所在主机的操作系统/内核。

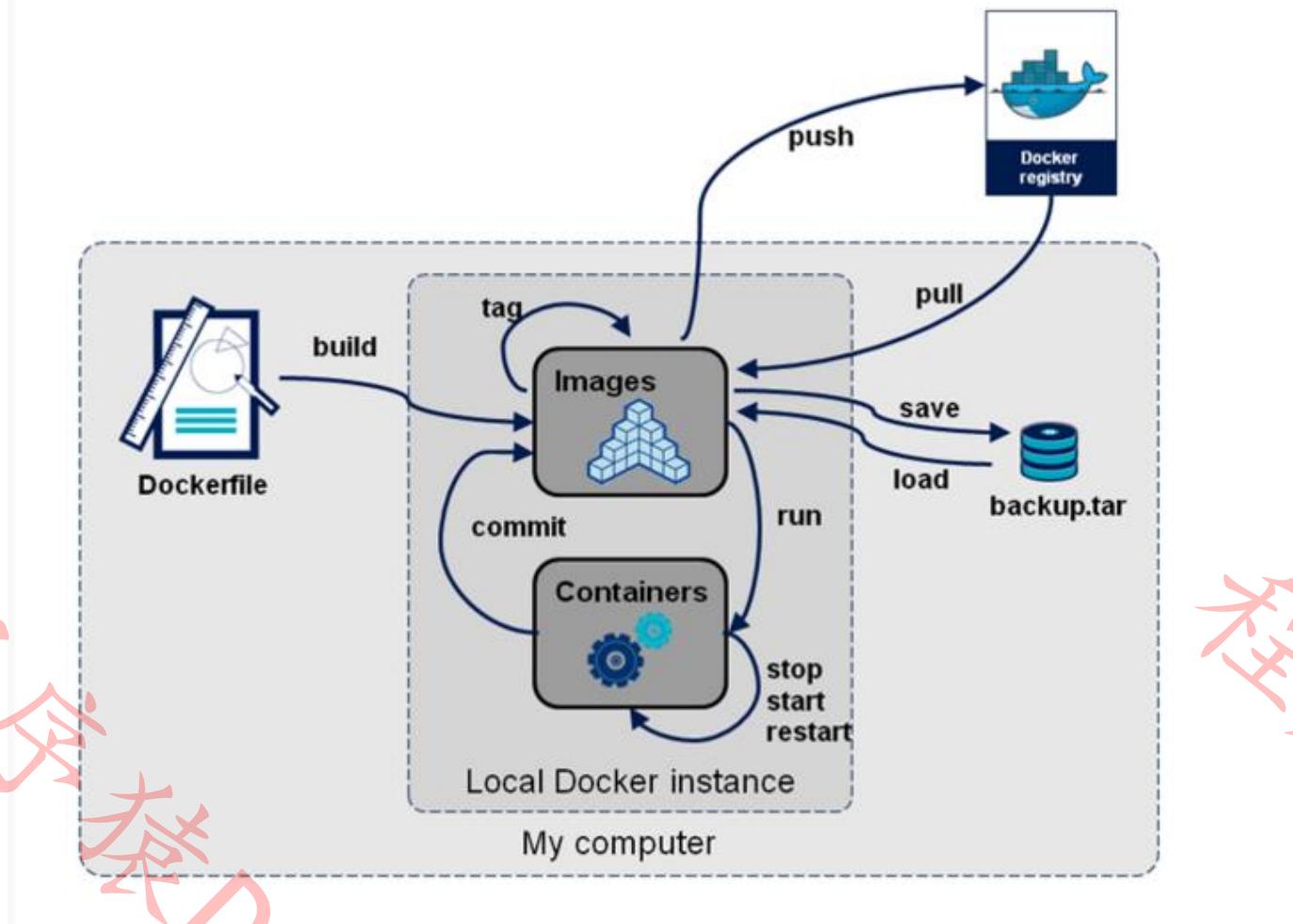
为什么要用容器呢？假设你在使用一台电脑开发一个应用，而且开发环境具有特定的配置。其他开发人员身处的环境配置可能稍有不同。你正在开发的应用不止依赖于您当前的配置，还需要某些特定的库、依赖项和文件。与此同时，你的企业还拥有标准化的开发和生产环境，有着自己的配置和一系列支持文件。你希望尽可能多在本地模拟这些环境，而不产生重新创建服务器环境的开销。这时候，就会需要容器来模拟这些环境。

我们常见的容器启动方式是 Docker，Docker 是一个开源的应用容器引擎，基于 Go 语言 并遵从 Apache2.0 协议开源。Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何 Linux 机器上，也可以实现虚拟化。



3.2 容器的发展进程

2010 年，几个年轻小伙在旧金山成立了一家做 PaaS 平台的公司，起名为"dotCloud"，该公司主要是基于 PaaS 平台为开发者或开发商提供技术服务。他们提供了对多种运行环境支持。但随着市场接受度、规模、加上科技巨头等影响，有一天 dotCloud 的创始人 Solomon Hykes 就召集了公司核心开发人员，商量准备开源 Docker 技术。因此，在 2013 年 3 月，Docker 正式以开源软件形式在 pycon 网站(见下图)首次发布了。正式由于这次开源，让容器领域焕发了第二春。后来在美国，几乎所有的云计算厂商都在拥抱 Docker 这个生态圈。很快 Docker 技术风靡全球，于是，dotCloud 决定改名为 Docker Inc(下面简称"Docker")，全身心投入到 Docker 的开发中。更名后的 Docker 并于 2014 年 8 月，Docker 宣布把平台即服务的业务 dotCloud 出售给位于德国柏林的平台即服务提供商 cloudControl，自此 dotCloud 和 Docker 分道扬镳。



3.3 Docker 与 Kubernetes

Google 多年来一直使用容器作为交付应用程序的一种重要方式，且运行有一款名为 Borg 的编排工具。Google、RedHat 等公司为了对抗以 Docker 公司为核心的容器商业生态，他们一起成立了 CNCF(Cloud Native Computing Foundation)。当谷歌于 2014 年 3 月开始开发 Kubernetes 时，很明智的选择当时最流行的容器，没错，就是 Docker。Kubernetes 对 Docker 容器运行时的支持，迎来了大量的使用用户。Kubernetes 于 2014 年 6 月 6 日首次发布。这便有了容器编排工具 Kubernetes 的诞生。另外，CNCF 的目的是以开源的 K8S 为基础，使得 K8S 能够在容器编排方面能够覆盖更多的场景，提供更强的能力。K8S 必须面临 Swarm 和 Mesos 的挑战。Swarm 的强项是和 Docker 生态的天然无缝集成，Mesos 的强项是大规模集群的管理和调度。K8S 是 Google 基于公司已经使用了十多年的 Borg 项目进行了沉淀和升华才提出的一套框架。它的优点就是有一套完整的新颖的设计理念，同时有 Google 的背书，而且在设计上有很强的扩展性，所以，最终 K8S 赢得了胜利，成为了容器生态的行业标准。

3.4 K8s 容器化应用

前面说了，K8s 是一种编排容器管理容器工具，那么如何通过 K8s 来将服务容器化呢？

首先，我们来看看 K8s 如何使用？第一条就是编写配置文件，因为配置文件可以是 YAML 或者 JSON 格式的。为方便阅读与理解，在后面的讲解中，我会统一使用 YAML 文件来指代它们。Kubernetes 跟 Docker 等很多项目最大的不同，就在于它不推荐你使用命令行的方式直接运行容器（虽然 Kubernetes 项目也支持这种方式，比如：kubectl run），而是希望你用 YAML 文件的方

式，即：把容器的定义、参数、配置，统统记录在一个 YAML 文件中，然后用这样一句指令把它运行起来：

```
kubectl create -f xxx.yaml
```

这样做最直接的一个好处是：你会有一个文件能记录下 K8s 到底 run 了哪些东西。比如下面这个例子：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tomcat-deployment
spec:
  selector:
    matchLabels:
      app: tomcat
  replicas: 2
  template:
    metadata:
      labels:
        app: tomcat
    spec:
      containers:
        - name: tomcat
          image: tomcat:10.0.5
          ports:
            - containerPort: 80
```

像这样的一个 YAML 文件，对应到 kubernetes 中，就是一个 API Object (API 对象)。当你为这个对象的各个字段填好值并提交给 Kubernetes 之后，Kubernetes 就会负责创建出这些对象所定义的容器或者其他类型的 API 资源。可以看到，这个 YAML 文件中的 Kind 字段，指定了这个 API 对象的类型 (Type)，是一个 Deployment。Deployment 是一个定义多副本应用 (即多个副本 Pod) 的对象。此外，Deployment 还负责在 Pod 定义发生变化时，对每个副本进行滚动更新 (Rolling+Update)。

在上面这个 Yaml 文件中，我给它定义的 Pod 副本个数 (spec.replicas) 是：2。但，这些 Pod 副本长啥样子呢？为此，我们定义了一个 Pod 模版 (spec.template)，这个模版描述了我要创建的 Pod 的细节。在上面的例子里，这个 Pod 里只有一个容器，这个容器的镜像 (spec.containers.image) 是 tomcat=10.0.5，这个容器监听端口 (containerPort) 是 80。

需要注意的是，像这种，使用一种 API 对象 (Deployment) 管理另一种 API 对象 (Pod) 的方法，在 Kubernetes 中，叫作“控制器”模式 (controller pattern)。在我们的这个 demo 中，Deployment 扮演的正是 Pod 的控制器的角色。而 Pod 是 Kubernetes 世界里的应用；而一个应用，可以由多个容器 (container) 组成。为了让我们这个 tomcat 服务容器化运行起来，我们只需要执行：

```
tom@PK001:~/damon$ kubectl create -f tomcat-deployment.yaml
deployment.apps/tomcat-deployment created
```

执行完上面的命令后，你就可以看容器运行情况，此时，只需要执行：

```
tom@PK001:~/damon$ kubectl get pod -l app=tomcat
NAME                               READY   STATUS    RESTARTS   AGE
tomcat-deployment-799f46f546-7nxrj  1/1    Running   0          77s
tomcat-deployment-799f46f546-hp874  0/1    Running   0          77s
```

'kubectl get' 指令的作用，就是从 Kubernetes 里面获取 (GET) 指定的 API 对象。可以看到，在这里我还加上了一个 -l 参数，即获取所有匹配 app=nginx 标签的 Pod。需要注意的是，在命令行中，所有 key-value 格式的参数，都使用 “=” 而非 “:” 表示。从这条指令返回的结果中，我们可以看到现在有两个 Pod 处于 Running 状态，也就意味着我们这个 Deployment 所管理的 Pod 都处于预期的状态。

此外，你还可以使用 kubectl describe 命令，查看一个 API 对象的细节，比如：

```
tom@PK001:~/damon$ kubectl describe pod tomcat-deployment-799f46f546-7nxrj
Name:           tomcat-deployment-799f46f546-7nxrj
Namespace:      default
Priority:       0
Node:           ca005/10.10.2.5
Start Time:     Thu, 08 Apr 2021 10:41:08 +0800
Labels:         app=tomcat
Annotations:    pod-template-hash=799f46f546
                cni.projectcalico.org/podIP: 20.162.35.234/32
Status:         Running
IP:             20.162.35.234
Controlled By: ReplicaSet/tomcat-deployment-799f46f546
Containers:
  tomcat:
    Container ID: docker://5a734248525617e950b7ce03ad7a19acd4ffbd71c67aacd9e3ec829d051b46d3
    Image:         tomcat:10.0.5
    Image ID:      docker-
    pullable:     sha256:2637c2c75e488fb3480492ff9b3d1948415151ea9c503a496c243ceb1800cbe4
    Port:          80/TCP
    Host Port:    0/TCP
    State:        Running
    Started:      Thu, 08 Apr 2021 10:41:58 +0800
    Ready:         True
    Restart Count: 0
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-2ww52 (ro)
Conditions:
  Type  Status
  Initialized  True
  Ready  True
  ContainersReady  True
  PodScheduled  True
Volumes:
  default-token-2ww52:
    Type:      Secret (a volume populated by a Secret)
    SecretName: default-token-2ww52
```

```

Optional:  false
QoS Class:  BestEffort
Node-Selectors:  <none>
Tolerations:  node.kubernetes.io/not-ready:NoExecute for 300s
               node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age   From           Message
  ----  -----     --   ----           -----
  Normal  Scheduled  4m17s  default-scheduler  Successfully assigned default/tomcat-
  deployment-799f46f546-7nxrj to ca005
  Normal  Pulling    4m16s  kubelet, ca005  Pulling image "tomcat:10.0.5"
  Normal  Pulled     3m27s  kubelet, ca005  Successfully pulled image "tomcat:10.0.5"
  Normal  Created    3m27s  kubelet, ca005  Created container tomcat
  Normal  Started    3m27s  kubelet, ca005  Started container tomcat

```

在 kubectl describe 命令返回的结果中，可以清楚地看到这个 Pod 的详细信息，比如它的 IP 地址等等。其中，有一个部分值得你特别关注，它就是 Events（事件）。

在 Kubernetes 执行的过程中，对 API 对象的所有重要操作，都会被记录在这个对象的 Events 里，并且显示在 kubectl describe 指令返回的结果中。这些 Events 中的信息很重要，可以排查容器是否运行、正常运行的原因。

如果你希望升级 tomcat 的版本，那可以直接修改 Yaml 文件：

```

spec:
  containers:
  - name: tomcat
    image: tomcat:latest
    ports:
    - containerPort: 80

```

修改完 Yaml 文件后，执行：

```
kubectl apply -f tomcat-deployment.yaml
```

这样的操作方法，是 Kubernetes “声明式 API” 所推荐的使用方法。也就是说，作为用户，你不必关心当前的操作是创建，还是更新，你执行的命令始终是 kubectl apply，而 Kubernetes 则会根据 YAML 文件的内容变化，自动进行具体的处理。

同时，可以查看容器内的服务的日志情况：

```

tom@PK001:~/damon$ kubectl logs -f tomcat-deployment-799f46f546-7nxrj
NOTE: Picked up JDK_JAVA_OPTIONS: --add-opens=java.base/java.lang=ALL-UNNAMED --add-
       opens=java.base/java.io=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED --add-
       opens=java.base/java.util.concurrent=ALL-UNNAMED --add-
       opens=java.rmi/sun.rmi.transport=ALL-UNNAMED
08-Apr-2021 02:41:59.037 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Server version name: Apache
Tomcat/10.0.5
08-Apr-2021 02:41:59.040 INFO [main]

```

org.apache.catalina.startup.VersionLoggerListener.log Server built: Mar 30 2021
08:19:50 UTC
08-Apr-2021 02:41:59.040 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Server version number: 10.0.5.0
08-Apr-2021 02:41:59.040 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log OS Name: Linux
08-Apr-2021 02:41:59.040 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log OS Version: 4.4.0-116-generic
08-Apr-2021 02:41:59.040 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Architecture: amd64
08-Apr-2021 02:41:59.040 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Java Home: /usr/local/openjdk-11
08-Apr-2021 02:41:59.040 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log JVM Version: 11.0.10+9
08-Apr-2021 02:41:59.040 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log JVM Vendor: Oracle
Corporation
08-Apr-2021 02:41:59.040 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log CATALINA_BASE: /usr/local/tomcat
08-Apr-2021 02:41:59.041 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log CATALINA_HOME: /usr/local/tomcat
08-Apr-2021 02:41:59.051 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-
opens=java.base/java.lang=ALL-UNNAMED
08-Apr-2021 02:41:59.051 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-
opens=java.base/java.io=ALL-UNNAMED
08-Apr-2021 02:41:59.051 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-
opens=java.base/java.util=ALL-UNNAMED
08-Apr-2021 02:41:59.051 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-
opens=java.base/java.util.concurrent=ALL-UNNAMED
08-Apr-2021 02:41:59.052 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-
opens=java.rmi/sun.rmi.transport=ALL-UNNAMED
08-Apr-2021 02:41:59.052 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -
Djava.util.logging.config.file=/usr/local/tomcat/conf/logging.properties
08-Apr-2021 02:41:59.052 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -
Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
08-Apr-2021 02:41:59.052 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -
Djdk.tls.ephemeralDHKeySize=2048
08-Apr-2021 02:41:59.052 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -
Djava.protocol.handler.pkgs=org.apache.catalina.webresources
08-Apr-2021 02:41:59.052 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -
Dorg.apache.catalina.security.SecurityListener.UMASK=0027
08-Apr-2021 02:41:59.052 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -
Dignore.endorsed.dirs=

```
08-Apr-2021 02:41:59.052 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -
Dcatalina.base=/usr/local/tomcat
08-Apr-2021 02:41:59.052 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -
Dcatalina.home=/usr/local/tomcat
08-Apr-2021 02:41:59.052 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -
Djava.io.tmpdir=/usr/local/tomcat/temp
08-Apr-2021 02:41:59.056 INFO [main]
org.apache.catalina.core.AprLifecycleListener.lifecycleEvent Loaded Apache Tomcat Native
library [1.2.27] using APR version [1.6.5].
08-Apr-2021 02:41:59.056 INFO [main]
org.apache.catalina.core.AprLifecycleListener.lifecycleEvent APR capabilities: IPv6
[true], sendfile [true], accept filters [false], random [true], UDS [true].
08-Apr-2021 02:41:59.059 INFO [main]
org.apache.catalina.core.AprLifecycleListener.initializeSSL OpenSSL successfully
initialized [OpenSSL 1.1.1d 10 Sep 2019]
08-Apr-2021 02:41:59.312 INFO [main] org.apache.coyote.AbstractProtocol.init Initializing
ProtocolHandler ["http-nio-8080"]
08-Apr-2021 02:41:59.331 INFO [main] org.apache.catalina.startup.Catalina.load Server
initialization in [441] milliseconds
08-Apr-2021 02:41:59.369 INFO [main]
org.apache.catalina.core.StandardService.startInternal Starting service [Catalina]
08-Apr-2021 02:41:59.370 INFO [main]
org.apache.catalina.core.StandardEngine.startInternal Starting Servlet engine: [Apache
Tomcat/10.0.5]
08-Apr-2021 02:41:59.377 INFO [main] org.apache.coyote.AbstractProtocol.start Starting
ProtocolHandler ["http-nio-8080"]
08-Apr-2021 02:41:59.392 INFO [main] org.apache.catalina.startup.Catalina.start Server
startup in [61] milliseconds
```

第四章 为何借助容器助力微服务

4.1 微服务的多语言性

微服务可以说是越来越火了：那么对于语言场景，从 Java 中的 Spring MVC，到后面的 SOA、Dubbo、Spring Cloud、Spring cloud Alibaba 等。这是单纯 Java 语言的微服务的发展史，那么对于 Python、Go 呢？其实也是有其微服务设计理念的，比如 Golang 的 beego、gin 等。

这么多的微服务框架，都是独立、限制于自己的体系，撇开其它技术不说，Java 可以自成生态体系，Golang 亦是。其实微服务不应该受某种语言的限制，各种语言的服务之间应该可以互通：这在云原生时代设计中，它们都被称为 Service。后面会介绍对于不同语言的服务（Service）如何在云原生中互通。

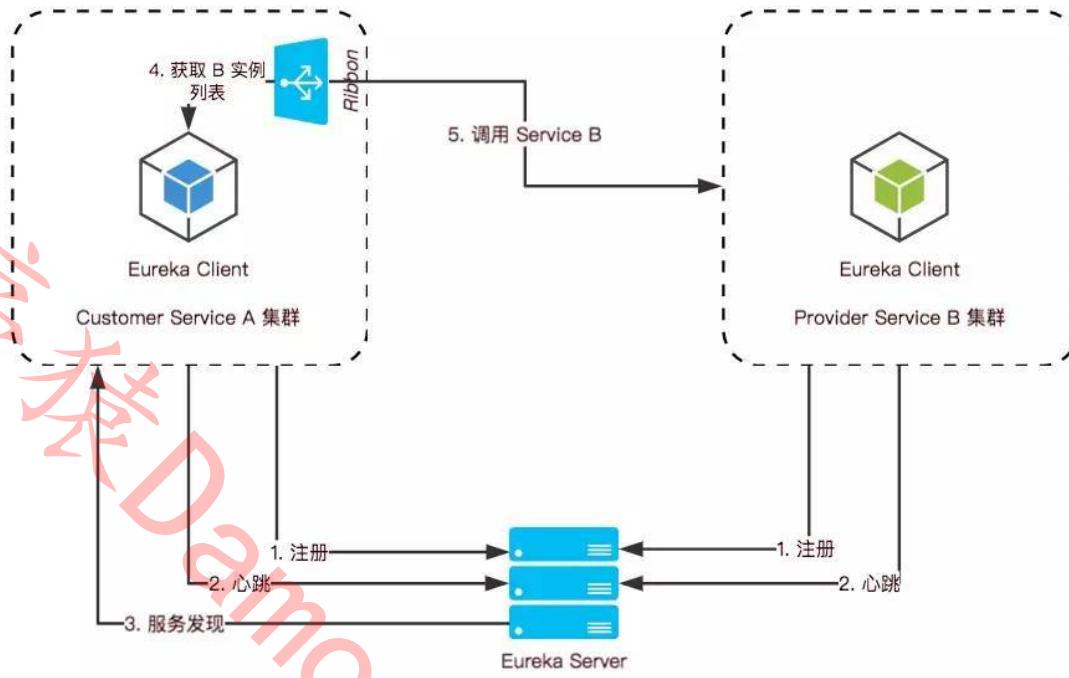
4.2 微服务的高可用

微服务出现后，同样面临着一个重要的话题：高可用。所谓高可用：英文缩写 HA(High Availability)，是指当某个服务或服务所在节点出现故障时，其对外的功能可以转移到该服务其他的副本或该服务在其他节点的副本，从而在减少停机时间的前提下，满足业务的持续性，这两个或多个服务构成了服务高可用。同时，这种高可用需要考虑到服务的性能压力，即服务的负载均衡。

我们知道对于服务的高可用，或者说服务的负载来说，有很多方式来解决这些问题。比如：

- 主从方式，其工作原理是：主机处于工作状态，备机处于监控、准备状态，当主机出现宕机的情况下，备机可以接管主机的一切工作，等到主机恢复正常后，将会手动或自动的方式将服务切换到主机上运行，数据的一致性通过共享存储来实现。
- 集群方式，其工作原理是：多台机器同时运行一个或几个服务，当其中的某个节点出现宕机时，这时该节点的服务将无法提供业务功能，可以选择根据一定的机制，将服务请求转移到该服务所在的其他节点上，这样可以让逻辑持续的执行下去，即消除软件单点故障。这其实就涉及到负载均衡策略。

对于微服务的高可用，涉及到的其中一个就是其服务的负载均衡。在微服务中，负载均衡的前提是，同一个服务需要被发现多个，或者说多个副本，这样才能实现负载均衡以及服务的高可用。那么，怎么让服务被发现呢？我们来看一张图：



在上图中，我们可以看到：

- 1.各微服务先往注册中心 Eureka 注册服务。
- 2.各微服务保持与注册中心心跳。
- 3.注册中心发现各微服务。
- 4.注册中心根据配置规则定期获取心跳，超时即认为节点无效。
- 5.根据规则来定期清理无效节点。

这是基于注册中心 Eureka 的服务注册与发现，同样，基于其他的注册中心实现原理基本类似。如：Eureka、Zookeeper、Consul、etcd、nacos 等。其实，在云原生 K8s 中，也存在着服务的注册与发现，这在后面章节中会讲解。

服务发现后，其实面临的是一个主要的问题就是应该访问哪一个？因为发现了某个服务的多个实例，最终只会访问其中某一个，这就涉及到服务的负载均衡了。

负载均衡在微服务中是一个很常见的话题，实现负载均衡的插件也越来越多。netflix 开源的 Zuul、Gateway 等等，其实 K8s 中也存在着负载均衡器：kube-dns、kube-proxy。

在实现服务注册与发现、负载均衡后，其实高可用还涉及很多：高并发、缓存等。先讲讲高并发：

- 幂等性
- 接口代码的规范性
- 操作 DB 的性能
- 读写分离操作
- 服务的横向扩展
- 服务的健壮性（缓存、限流、熔断）

其中，幂等性：就是说一次和多次请求某一个资源时对于资源本身应该具有同样的结果（网络超时等问题除外）。也就是说，其任意次执行所产生的效果和返回的结果都是一样的。这种场景是一个很有效的实现高并发的情景，设想，用户充值某个会员，在并发情况下，用户由于误操作，或者由于网络、时间等问题导致重试机制的发生时，可能会触发触发多次交易的扣费，这样给用户一个很不好的体验。此时，就需要接口幂等性来解决这类问题。

幂等性解决方案有以下几种：

- token 机制
- 接口逻辑实现幂等性
- 数据库层处理实现幂等性

token 机制：数据提交时携带 token，token 放到 redis，token 有效时间，提交后台后校验 token，同时删除 token，生成新的 token 并返回。

接口的幂等性：常见的接口幂等性，是定义接口时，加上参数序列号、来源等，序列号与请求来源联合唯一索引，这样可以有效判断本次请求方与请求的序列号，防止重复的请求。

数据库处理：DB 层处理有多种方式：1. 悲观锁，2. 乐观锁，3. 唯一索引、组合唯一索引，4. 分布式锁。

悲观锁：所谓悲观锁，是指存在危机意识，事先(查询时)加锁处理，防止事情发生。如：

```
select * from xxx where id= 1 for update
```

乐观锁：是指存在乐观心理，只在更新时加锁，乐观锁通常用 version 版本号来控制如：

```
update xxx set name=#name#,version=version+1 where version=#version#
```

也可以通过条件限制，这里就使用了组合唯一索引来处理，如：

```
update xxx set name=#name#,version=version+1 where id=#id# and version=#version#
```

分布式锁：通过 redis、zookeeper 来设置分布式锁，当插入或更新数据时，获取分布式锁，然后做操作，之后释放锁。

接口的规范性：接口的性能如何，最终还是跟接口的实现逻辑有关，比如代码规范，逻辑实现等，尤其是业务逻辑复杂的情况下，这点需要注意的。

操作 DB：对于业务的持久层，用的比较多的就是 mybatis、hibernate，还有可能是 JPA，无论是哪个，最终都是通过工厂类注入 bean，最后执行 SQL 来操作 DB。所以这里尤为重要的事 SQL 的写法，SQL 的优化决定着操作 DB 的时间以及效果，如果写得不好的话，则会导致死循环，或死锁，或内存溢出。另外测试时，使用真实、规范的数据进行测试，并在测试时不要局限于相同的数据，最后就是并发压测了。

读写分离：当服务足够多，数据足够多时，有可能读与写的占比为：10:1，此时读写应该分离，这样可以有效减少因为读的频繁操作导致的写的性能下降。常见的读写分离的方法有：采用 mycat 中间件方式、amoeba 直接实现读写分离、手动修改 mysql 操作类直接实现读写分离和随机实现的负载均衡，权限独立分配、mysql-proxy（还是测试版本，时间消耗有点高）。

服务的横向扩展：对于服务的请求越来越多时，此时需要对服务进行多节点部署，这样减少单机带来的服务负载压力。

服务的健壮性：服务的健壮性包括缓存、限流、熔断。

对于缓存，大家都知道，有常见的许多中间件如：redis、kafka、RabbitMq、zookeeper。对于一些 session 等常用 redis 来缓存、共享。对于一些大一点的数据如果嫌弃加载慢，也可以采用缓存机制来解决。

什么叫限流呢？很好理解，就是限制节点的流量，限制服务的请求数。那么如何做到限流呢？常用的限流算法比如有计数器算法、令牌桶算法、漏桶算法。有几种方式：利用 springcloud 组件 zuul 来对请求进行限流，主要是通过谷歌提供的 RateLimiter 结合一些限流算法来限流比较常用。利用 redis 同样可以做限流算法的，甚至可以利用 nginx 直接作计数限流，可以对请求速率进行限制、对每个 ip 连接数量进行限制、对每个服务的连接数量进行限制。如：

```
#对请求速率进行限制
limit_req_zone $binary_remote_addr zone=req_one:20m rate=12r/s;
limit_conn_zone $binary_remote_addr zone=addr:10m;
#对每个ip连接数量进行限制
limit_conn_zone $server_name zone=perserver:20m;
#对每个服务的连接数量进行限制
server{
    listen 80;
    location / {
        proxy_pass http://ip:port;
        limit_req zone=req_one burst= 80 nodelay;
        limit_conn addr 20;
    }
}
```

其实在 Springboot2.x 中，推出自己的 Spring-Cloud-Gateway 来作网关，同时 Spring-Cloud-Gateway 中提供了基于 Redis 的实现来达到限流的目的。对于其它框架，都有加入限流的插件功能。

对于熔断，或者说熔断，这个在实际的业务当中是很有必要的。比如：用户在某一商城秒杀某一件物品，或在某米商城上抢购某一部手机，在准点抢购时，发现人很多，请求很多，这时，主要是需

要有限流机制，同时也需要有熔断（熔断），给用户留下一个很好的体验的感觉。当用户在点击抢购按钮后，如果当前的请求数很多，需要用户等待，这是需要给一个友好的界面让用户去等待，而不是直接给用户提示请求失败，或者报异常，这样的红色抛出是一个非常不好的事情，用户可能会骂街的，下次也不会逛了。

Spring-Cloud-Gateway 作网关时，过滤器时使用 HystrixGatewayFilterFactory 来创建一个 Filter 实现基于 Route 级别的熔断功能。

对于缓存问题，随着系统用户的越来愈多，所有的服务压力也会指数型递增，这时候缓存是一个很好的减轻服务压力的方式。这样可以有效缓冲请求对服务的负载压力。常见的缓存可能是 Redis、MQ(RabbitMQ、RocketMQ)、Kafka、ZooKeeper 等。

Redis 一般主要做 session 或用户信息的缓存，实现多机中 session 的共享。也会用来作分布式锁，在分布式高并发下实现锁的功能，例如实现秒杀、抢单等功能。还会被用作一些订单信息的缓存，防止大量的订单信息被积压而导致服务器的负载很高。总之，Redis 常被用来作为一种缓冲剂使用。

MQ 常见的有 RabbitMQ、RocketMQ、ActiveMQ、Kafka 等，以下是各种之间的对比：

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，吞吐量比RocketMQ和Kafka要低了一个数量级	万级，吞吐量比RocketMQ和Kafka要低了一个数量级	10万级，RocketMQ也是可以支撑高吞吐的一种MQ	10万级别，这是kafka最大的优点，就是吞吐量高。一般配合大数据类的系统来进行实时数据计算、日志采集等场景
时效性	ms级	微秒级	ms级	延迟在ms级以内
可用性	高，基于主从架构实现高可用性	高，基于主从架构实现高可用性	非常高，分布式架构	非常高，kafka是分布式的，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据		经过参数优化配置，可以做到0丢失	经过参数优化配置，消息可以做到0丢失
功能支持	MQ领域的功能极其完备	基于erlang开发，所以并发能力很强，性能极好，延时很低	MQ功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的MQ功能，在大数据领域的实时计算以及日志采集被大规模使用，是事实上的标准

ZooKeeper 也是经常会存储海量数据，例如 Hadoop 中，在使用 YARN 作资源调度时，采用 ZooKeeper 来存储海量的状态机状态以及任务的信息（包括历史信息）。

4.3 微服务的复杂性

说起微服务，使用 DDD 划分微服务的好处的时候，经常会说 DDD 能够让相关的业务逻辑更加内聚，并且降低服务之间的耦合性，从而最终实现达到降解系统的复杂性。但是在这里，不论是高内聚，低耦合，甚至我们经常说的系统复杂性，我们有没有一个客观的可以量化的指标来衡量这些概念。由于无法量化，所以就没法度量，这样当团队在讨论一个系统是否复杂，有多复杂这些问题的时候，就很容易陷入各种主观直觉的争论中。

互联网时代，这些巨多的系统在细节上不一样，但是如果从抽象层面来看有很多共性，一个微服务系统由很多个微服务组成，这些微服务的行为产生出复杂的行为模式。整个微服务系统会通过 API 利用内部或外部的信号，同时在系统内部也是通过服务之间的接口进行信息传递。一个微服务系统并不是静态的，而是会不断适应业务变化，改变系统的 API 或者系统内部的组织方式来增加生存的机会。

微服务的复杂，不仅仅在于其系统本身，还需要考虑的是：高可用、服务自治、服务并发、服务限流、熔断等。

服务的高可用在上一节中已经说明了，服务自治，目的其实是在对其修改时对其他部分造成尽可能小的影响；自治服务运营时也不会对其他服务的功能造成影响。服务几乎总是要依赖其他服务提供的数据。例如，网上商城都有一个购物车微服务，一些其他服务必须能向购物车添加商品，还必须能访问购物车内的商品并下单和配送。现在问题是，如何在保持服务尽可能自治的前提下实现对接。那需要遵循一定的模式：交互、信息传递。

交互模式：Request-Reply 还是 Publish-Subscribe

- Request-Reply（请求-应答）意味着一个服务处理信息的特定请求或者执行一些动作并返回一个应答。发起调用的服务需要知道去哪儿请求以及请求些什么？这种模式仍然可以被实现为异步执行，并且你还可以做一些抽象使服务调用方不需要知道被调用服务的物理地址，不能逃避的一点是服务必须明确的要求一个特定的信息和功能（或者执行动作）并等待应答。
- Publish-Subscribe（发布-订阅）这种模式下的服务将自己注册为对特定的信息感兴趣，或者能够处理特定的请求，相关的信息和请求将被交付给它，并且它可以决定怎么处理这些信息和请求。本文假定有一些中间件能够处理交付或者发布消息给订阅这些消息服务。

信息传递：Events 还是 Queries/Commands

- Events（事件）是没有争议的事实，比如订单号 123 的订单已经创建，事件只陈述发生了什么事，不描述这样一个事件会导致什么事情发生。
- Queries/Commands（查询/命令）两者都传达了什么事情会发生，查询是对信息的特定请求，命令是要求一个服务执行一些动作的特定请求。

上面的四种即可作为微服务间对接的四种方式：**REQUEST-REPLY WITH EVENTS**、**REQUEST-REPLY WITH COMMANDS/QUERIES**、**PUBLISH-SUBSCRIBE WITH EVENTS**、**PUBLISH-SUBSCRIBE WITH COMMANDS/QUERIES**。

- REQUEST-REPLY WITH EVENTS，在这种模式下，一个服务请求另一个导致事件发生的服务，这意味着这两种服务之间有很强的依赖。配送服务必须知道要连接那个服务来获得订单相关的事件，这也导致了运行时依赖，因为配送服务只有在订单服务可用的时候才能配送新订单。既然配送服务只接收事件，它基于事件里的信息自己决

定何时一个订单可以被配送，订单服务不需要知道配送服务的任何信息，它只是简单的提供事件表明当其他服务请求时订单进行怎样的处理，把响应事件的职责完全交给请求事件的服务。

- REQUEST-REPLY WITH COMMANDS/QUERIES，如：订单服务将请求配送服务来配送一个订单，这意味着强烈的依赖，因为订单服务明确的请求一个特定的服务来处理配送，现在订单服务必须决定何时一个订单准备好配送，它意识到配送服务的存在，甚至知道怎样与配送服务交互，在订单配送前需要考虑是否有其他因素关联到订单（比如客户信用卡状态），订单服务在请求配送服务来配送订单前也需要考虑这一点。现在业务处理被混到了架构里，因此架构不能被简单的修改。这也是运行时依赖，因为订单服务必须确保配送请求成功交付给了配送服务。
- PUBLISH-SUBSCRIBE WITH EVENTS，配送服务注册自己对订单相关的事件感兴趣，注册后，配送服务会收到订单的所有事件而不需要关心订单事件的来源，这是对订单事件来源的松散耦合，配送服务需要保留接收到事件的副本，这样就可以决定何时订单准备好配送。订单服务需要对配送无关，如果多个服务提供包含配送服务需要的相关数据的订单相关事件，配送服务应该不可识别，如果一个提供订单事件的服务宕机，配送服务也应该不知道，只是收到的事件变少了，配送服务不会因此阻塞。
- PUBLISH-SUBSCRIBE WITH COMMANDS/QUERIES，配送服务自己注册为能够配送货物的服务，接受所有想要配送货物的命令，配送服务不需要意识到配送命令的来源，同样订单服务业不知道那些服务将处理配送，在这个意义上说，他们是松散耦合的，不过，订单服务知道既然发送了配送命令，订单必须被配送的事实，这确实让耦合更强了。

两种 Request-Reply 模式都意味着两个服务的运行时耦合和强耦合，两种 Command/Queries 模式意味着一个服务知道另一个服务应该做的事，这也意味着强耦合，但是这一次在功能级别。留下了一个选项：PUBLISH-SUBSCRIBE WITH EVENTS，这种情况下，两种服务从运行时和功能的角度都没有意识到彼此的存在。

但是，我们需要考虑更多的因素，一直使用这种方式交互是有代价的，例如，数据被复制、事件丢失、事件驱动的架构增加更多基础设施的需求、额外的延迟。

第二部分 原理与应用

第五章 Kubernetes 介绍

5.1 Kubernetes 的基本概念与特性

在前面的章节中介绍了 Kubernetes 的由来，那么 Kubernetes 到底是干嘛的呢？这就涉及到 Kubernetes 的定义以及其特性，本节来描述下 Kubernetes 的特性以及应用场景。

Kubernetes，简称 K8s，是把 8 代替了 8 个字符 “ubernete” 而成的缩写。K8s 是一个一个开源的，用于管理云平台中多个主机上的容器化的应用编排工具。它的目标是让部署容器化的应用简单且高效，Kubernetes 提供了应用部署，规划，更新，维护的一种机制。

Kubernetes 是 Google 开源的一个容器编排引擎，支持自动化部署、大规模可伸缩、应用容器化管理。在生产环境中部署一个应用程序时，通常要部署该应用的多个实例以便对应用请求进行负载

均衡。在 Kubernetes 中，我们可以创建多个容器，每个容器里面运行一个应用实例，然后通过内置的负载均衡策略，实现对这一组应用实例的管理、发现、访问，而这些细节都不需要运维人员去进行复杂的手工配置和处理。

特性：

- 可移植：支持公有云，私有云，混合云
- 可扩展：模块化，插件化，可挂载，可组合
- 自动化：自动部署，自动重启，自动复制，自动弹性伸缩

可移植，意味着可以穿梭任何系统，不受系统的限制，也不受任何语言的限制，支持任何的其他的服务形式。

可扩展，是指 K8s 的各个模块之间是解耦合的，可以增加插件来丰富其功能，也可以替换其组件来达到想要的效果。

自动部署和回滚，K8s 采用滚动更新策略更新应用，一次更新一个 Pod，而不是同时删除所有 Pod，如果更新过程中出现问题，将回滚更改，确保升级不受影响业务。

弹性伸缩，使用命令、UI 或者基于 CPU 使用情况自动快速扩容和缩容应用程序实例，保证应用业务高峰并发时的高可用性；业务低峰时回收资源，以最小成本运行服务。

自动重启，是说在集群节点宕机、机器重启后，其 K8s 集群具有自动重启的功能，同时，会拉起集群中所有的应用服务，这就是其编排能力的一个体现。

自动复制，是指所有相关的数据，可以被备份到 etcd 或其它插件中，以便 K8s 可以通过 controllers、scheduler 来很好的编排。

可挂载，是指存储编排，挂载外部存储系统，无论是来自本地存储，公有云（如 AWS），还是网络存储（如 NFS、GlusterFS、Ceph）都作为集群资源的一部分使用，极大提高存储使用灵活性。

自我修复，在节点故障时重新启动失败的容器，替换和重新部署，保证预期的副本数量；杀死健康检查失败的容器，并且在未准备好之前不会处理客户端请求，确保线上服务不中断。

5.2 部署 Kubernetes 集群

如果想要了解 K8s 的一些特性，并且将其运用的很好，那就需要动手部署一个 K8s 集群。下面讲解下 K8s 集群部署流程。

单机版 K8s

环境：

- Ubuntu 16.04
- GPU 驱动 418.56
- Docker 18.06
- K8s 1.13.5

以上的环境，针对的是高版本的 K8s，而且 Docker 的版本必须要注意。另外 GPU 驱动的话，如果大家是非 GPU 机器的话，可以考虑不用。如果含有 GPU 机器的话，需要安装驱动，并且驱动版本不能过低喔。

设置环境

在配置环境前，首先备份一下源配置：

```
cp /etc/apt/sources.list /etc/apt/sources.list.cp
```

然后我们重新写一份配置，编辑内容，加上阿里源：

```
vim /etc/apt/sources.list

deb-src http://archive.ubuntu.com/ubuntu xenial main restricted
deb http://mirrors.aliyun.com/ubuntu/ xenial main restricted
deb-src http://mirrors.aliyun.com/ubuntu/ xenial main restricted multiverse universe
deb http://mirrors.aliyun.com/ubuntu/ xenial-updates main restricted
deb-src http://mirrors.aliyun.com/ubuntu/ xenial-updates main restricted multiverse
universe
deb http://mirrors.aliyun.com/ubuntu/ xenial universe
deb http://mirrors.aliyun.com/ubuntu/ xenial-updates universe
deb http://mirrors.aliyun.com/ubuntu/ xenial multiverse
deb http://mirrors.aliyun.com/ubuntu/ xenial-updates multiverse
deb http://mirrors.aliyun.com/ubuntu/ xenial-backports main restricted universe
multiverse
deb-src http://mirrors.aliyun.com/ubuntu/xenial-backports main restricted universe
multiverse
deb http://archive.canonical.com/ubuntu xenial partner
deb-src http://archive.canonical.com/ubuntu xenial partner
deb http://mirrors.aliyun.com/ubuntu/ xenial-security main restricted
deb-src http://mirrors.aliyun.com/ubuntu/ xenial-security main restricted multiverse
universe
deb http://mirrors.aliyun.com/ubuntu/ xenial-security universe
deb http://mirrors.aliyun.com/ubuntu/ xenial-security multiverse
```

添加好后，可以执行如下命令，更新一下源：

```
apt-get update
```

如果出现问题，可以执行如下命令来自动修复安装出现 broken 的 package：

```
apt --fix-broken install
```

执行升级命令时，注意：对于 GPU 机器可不执行，否则可能升级 GPU 驱动导致问题。

```
apt-get upgrade
```

由于 K8s 安装要求，需要关闭防火墙：

```
ufw disable
```

安装 SELinux：

```
apt install selinux-utils
```

SELinux 防火墙配置：

```
setenforce 0  
vim /etc/selinux/config  
SELINUX=disabled
```

设置网络，将桥接的 IPV4 流量传递到 iptables 的链：

```
tee /etc/sysctl.d/k8s.conf << 'EOF'  
net.bridge.bridge-nf-call-ip6tables = 1  
net.bridge.bridge-nf-call-iptables = 1  
net.ipv4.ip_forward = 1  
EOF
```

为了防止下面执行的会报错，可以先执行一下：

```
modprobe br_netfilter
```

最后，查看 IPV4 与 v6 配置是否生效：

```
sysctl --system
```

配置 iptables：

```
iptables -P FORWARD ACCEPT  
vim /etc/rc.local  
/usr/sbin/iptables -P FORWARD ACCEPT
```

需要永久关闭 swap 分区：

```
sed -i 's/.*swap.*/#&/' /etc/fstab
```

以上为 K8s 系统的环境配置，这个条件是硬性的，必须要处理。接下来就是安装需要的配套工具了。

安装 Docker

设置环境，在 Docker 安装前设置：

```
apt-get install apt-transport-https ca-certificates curl software-properties-common  
curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | apt-key add -  
add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu  
$(lsb_release -cs) stable" apt-get update
```

删除已经存在的低版本 Docker：

```
apt-get purge docker-ce docker docker-engine docker.io && rm -rf /var/lib/docker  
apt-get autoremove docker-ce docker docker-engine docker.io
```

安装指定版本的 Docker：

```
apt-get install -y docker-ce=18.06.3~ce~3-0~ubuntu
```

启动 Docker 并设置开机自重启：

```
systemctl enable docker && systemctl start docker
```

安装好 Docker 后，需要配置一下 Docker 以让其生效：

```
vim /etc/docker/daemon.json  
{  
  "log-driver": "json-file",  
  "log-opt": {  
    "max-size": "100m",  
    "max-file": "10"  
  },  
  "insecure-registries": ["http://k8s.gcr.io"],  
  "data-root": "",  
  "default-runtime": "nvidia",  
  "runtimes": {  
    "nvidia": {  
      "path": "/usr/bin/nvidia-container-runtime",  
      "runtimeArgs": []  
    }  
  }  
}
```

```
}
```

上面是含 GPU 的配置，如果你的机器不含 GPU，可按照下面来配置：

```
{
  "registry-mirrors": [
    "https://registry.docker-cn.com"
  ],
  "storage-driver": "overlay2",
  "log-driver": "json-file",
  "log-opt": {
    "max-size": "100m"
  },
  "exec-opt": [
    "native.cgroupdriver=systemd"
  ],
  "insecure-registries": ["http://k8s.gcr.io"],
  "live-restore": true
}
```

最后，我们重启 Docker 服务并设置开机自动重启，重启后可以看到 Docker 的相关信息：

```
systemctl daemon-reload && systemctl restart docker && docker info
```

安装 K8s

在安装 K8s 之前，我们需要设置一下环境，以便很快的拉取相关的镜像，这里选择了阿里源：

```
apt-get update && apt-get install -y apt-transport-https curl
curl -s https://mirrors.aliyun.com/kubernetes/apt/doc/apt-key.gpg | apt-key add -
tee /etc/apt/sources.list.d/kubernetes.list <<- 'EOF'
deb https://mirrors.aliyun.com/kubernetes/apt kubernetes-xenial main
EOF
```

配置完阿里源后，我们更新一下资源：

```
apt-get update
```

更新完后，先摒弃不可用或损坏的 K8s 组件 kubectl、kubeadm、kubelet 的残留：

```
apt-get purge kubelet=1.13.5-00 kubeadm=1.13.5-00 kubectl=1.13.5-00
apt-get autoremove kubelet=1.13.5-00 kubeadm=1.13.5-00 kubectl=1.13.5-00
```

重新更新、安装一份指定版本的 K8s 组件：

```
apt-get install -y kubelet=1.13.5-00 kubeadm=1.13.5-00 kubectl=1.13.5-00
apt-mark hold kubelet=1.13.5-00 kubeadm=1.13.5-00 kubectl=1.13.5-00
```

这里的三大组件 kubectl、kubeadm、kubelet，都是比较重要的，下面简单介绍下：

- kubectl 是 K8s 集群的命令行工具，通过 kubectl 能够对集群本身进行管理，并能够在集群上进行容器化应用的安装部署。
- kubeadm 是部署、安装 K8s 的一种命令工具。它提供了 `kubeadm init` 以及 `kubeadm join` 这两个命令作为快速创建 Kubernetes 集群的最佳实践。
- 关于 kubelet，在 K8s 集群中，在每个 Node 上都会启动一个 kubelet 服务的进程。该进程用于处理 Master 下发到本节点的任务，管理 Pod 及 Pod 中的容器。每个 kubelet 进程都会在 API Server 上注册节点自身的信息，定期向 Master 汇报节点资源的使用情况，并通过 cAdvisor 监控容器和节点资源。

接下来启动服务并设置开机自动重启：

```
systemctl enable kubelet && sudo systemctl start kubelet
```

组件安装好了，接下来安装 K8s 相关镜像，由于 gcr.io 网络访问不了，从 registry.cn-hangzhou.aliyuncs.com 镜像地址下载：

```
docker pull registry.cn-hangzhou.aliyuncs.com/gg-gcr-io/kube-apiserver:v1.13.5
docker pull registry.cn-hangzhou.aliyuncs.com/gg-gcr-io/kube-controller-manager:v1.13.5
docker pull registry.cn-hangzhou.aliyuncs.com/gg-gcr-io/kube-scheduler:v1.13.5
docker pull registry.cn-hangzhou.aliyuncs.com/gg-gcr-io/kube-proxy:v1.13.5
docker pull registry.cn-hangzhou.aliyuncs.com/kuberimages/pause:3.1
docker pull registry.cn-hangzhou.aliyuncs.com/kuberimages/etcd:3.2.24
docker pull registry.cn-hangzhou.aliyuncs.com/kuberimages/coredns:1.2.6
```

注意：每个镜像的版本要注意，而且对应的 CoreDNS、Etcd 版本也要对应，每个版本的 K8s 对应的不一样的。否则，可能会出问题的。

拉取镜像后，我们需要打标签，因为 `kubeadm init` 的时候，标签是固定的，具体如下：

```
docker tag registry.cn-hangzhou.aliyuncs.com/gg-gcr-io/kube-apiserver:v1.13.5
k8s.gcr.io/kube-apiserver:v1.13.5

docker tag registry.cn-hangzhou.aliyuncs.com/gg-gcr-io/kube-controller-manager:v1.13.5
k8s.gcr.io/kube-controller-manager:v1.13.5
```

```
docker tag registry.cn-hangzhou.aliyuncs.com/gg-gcr-io/kube-scheduler:v1.13.5  
k8s.gcr.io/kube-scheduler:v1.13.5

docker tag registry.cn-hangzhou.aliyuncs.com/gg-gcr-io/kube-proxy:v1.13.5  
k8s.gcr.io/kube-proxy:v1.13.5

docker tag registry.cn-hangzhou.aliyuncs.com/kuberimages/pause:3.1 k8s.gcr.io/pause:3.1

docker tag registry.cn-hangzhou.aliyuncs.com/kuberimages/etcd:3.2.24  
k8s.gcr.io/etcd:3.2.24

docker tag registry.cn-hangzhou.aliyuncs.com/kuberimages/coredns:1.2.6  
k8s.gcr.io/coredns:1.2.6
```

kubeadm 初始化

上面的操作步骤走完后，接下来就是利用 kubeadm 初始化 K8s，其中主机 IP 根据自己的实际情况输入，通过 `kubeadm init [flags]` 形式可以启动一个 master 节点：

```
kubeadm init --kubernetes-version=v1.13.5 --pod-network-cidr=10.244.0.0/16 --service-cidr=10.16.0.0/16 --apiserver-advertise-address=${masterIp} | tee kubeadm-init.log
```

此时，如果未知主机 IP，也可利用 yaml 文件动态初始化，我们通过 hosts 来进行动态加载：

```
vi /etc/hosts  
10.10.5.100 k8s.api.server

vi kube-init.yaml

apiVersion: kubeadm.k8s.io/v1beta1
kind: ClusterConfiguration
kubernetesVersion: v1.13.5
imageRepository: registry.aliyuncs.com/google_containers
apiServer:
  certSANs:
    - "k8s.api.server"
controlPlaneEndpoint: "k8s.api.server:6443"
networking:
  serviceSubnet: "10.1.0.0/16"
  podSubnet: "10.244.0.0/16"
```

设置 Etcd HA 版本：

```
apiVersion: kubeadm.k8s.io/v1beta1
kind: ClusterConfiguration
kubernetesVersion: v1.13.5
imageRepository: registry.aliyuncs.com/google_containers
apiServer:
  certSANs:
    - "api.k8s.com"
```

```
controlPlaneEndpoint: "api.k8s.com:6443"
etcd:
  external:
    endpoints:
      - https://ETCD_0_IP:2379
      - https://ETCD_1_IP:2379
      - https://ETCD_2_IP:2379
networking:
  serviceSubnet: 10.1.0.0/16
  podSubnet: 10.244.0.0/16
```

注意：apiVersion 中用 kubeadm，因为需要用 kubeadm 来初始化，最后执行下面来初始化。

```
kubeadm init --config=kube-init.yaml
```

请耐心等几分钟直到结束。

出现问题，解决后，执行：

```
kubeadm reset
```

如果需要更多，可执行下面来查看：

```
kubeadm --help
```

部署如果没问题，查看当前的版本：

```
kubelet --version
```

部署出现问题

先删除 node 节点（集群版）：

```
kubectl drain <node name> --delete-local-data --force --ignore-daemonsets
kubectl delete node <node name>
```

清空 init 配置在需要删除的节点上执行（注意，当执行 init 或者 join 后出现任何错误，都可以使用此命令返回）：

```
kubeadm reset
```

查问题

初始化后出现问题，可以通过以下命令先查看其容器状态以及网络情况：

```
sudo docker ps -a | grep kube | grep -v pause  
sudo docker logs CONTAINERID  
sudo docker images && systemctl status -l kubelet  
netstat -nlpt  
kubectl describe ep kubernetes  
kubectl describe svc kubernetes  
kubectl get svc kubernetes  
kubectl get ep  
netstat -nlpt | grep apiser  
vi /var/log/syslog
```

给当前用户配置 K8s apiserver 访问公钥

```
sudo mkdir -p $HOME/.kube  
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config  
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

网络插件

在上面的步骤后，如果查看节点情况：

```
kubectl get nodes
```

查看 nodes 状态信息，看到 node 节点的状态为 NotReady，这是因为缺少容器网络的配置。

接下来需要部署网络插件：

```
kubectl apply -f https://docs.projectcalico.org/v3.3/getting-started/kubernetes/installation/hosted/rbac-kdd.yaml  
  
wget https://docs.projectcalico.org/v3.3/getting-started/kubernetes/installation/hosted/kubernetes-datastore/calico-networking/1.7/calico.yaml  
  
vi calico.yaml
```

```
- name: CALICO_IPV4POOL_IPIP
  value: "off"
- name: CALICO_IPV4POOL_CIDR
  value: "10.244.0.0/16"
```

```
kubectl apply -f calico.yaml
```

单机下允许 master 节点部署 pod 命令如下：

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

禁止 master 部署 pod：

```
kubectl taint nodes k8s node-role.kubernetes.io/master=true:NoSchedule
```

以上单机版部署结束，如果你的项目中，交付的是软硬件结合的一体机，那么到此就结束了。记得单机下要允许 master 节点部署哟！

K8s 集群版实战

以上面部署的机器为例，作为 master 节点，我们备份一些配置到节点机器，执行：

```
scp /etc/kubernetes/admin.conf $nodeUser@$nodeIp:/home/$nodeUser
scp /etc/kubernetes/pki/etcd/* $nodeUser@$nodeIp:/home/$nodeUser/etcd
kubeadm token generate
kubeadm token create $token_name --print-join-command --ttl=0
kubeadm join $masterIP:6443 --token $token_name --discovery-token-ca-cert-hash $hash
```

注意，这个 token 24 小时后会失效，如果后面有其他节点要加入的话，处理方法：

```
kubeadm token generate
kubeadm token list
openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der
2>/dev/null | openssl dgst -sha256 -hex | sed 's/^.* //'
```

然后拿到 token 和一个 sha256 密钥后执行下面即可加入集群：

```
kubeadm join $masterIP:6443 --token $token_name --discovery-token-ca-cert-hash $hash
```

Node 机器执行时，如果需要 Cuda，可以参考以下资料：

- <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#ubuntu-installation>
- <https://blog.csdn.net/u012235003/article/details/54575758>
- https://blog.csdn.net/qq_39670011/article/details/90404111

正式执行：

```
vim /etc/modprobe.d/blacklist-nouveau.conf

blacklist nouveau
options nouveau modeset=0
update-initramfs -u
```

重启 Ubuntu 查看是否禁用成功：

```
lsmod | grep nouveau

apt-get remove --purge nvidia*

https://developer.nvidia.com/cuda-downloads

sudo apt-get install freeglut3-dev build-essential libx11-dev libxmu-dev libxi-dev
libgl1-mesa-glx libglu1-mesa libglu1-mesa-dev
```

安装 Cuda：

```
accept

select "Install" / Enter

select "Yes"

sh cuda_10.1.168_418.67_linux.run

echo 'export PATH=/usr/local/cuda-10.1/bin:$PATH' >> ~/.bashrc

echo 'export PATH=/usr/local/cuda-10.1/NsightCompute-2019.3:$PATH' >> ~/.bashrc

echo 'export LD_LIBRARY_PATH=/usr/local/cuda-10.1/lib64:$LD_LIBRARY_PATH' >> ~/.bashrc

source ~/.bashrc
```

重启机器，检查 Cuda 是否安装成功。

查看是否有 nvidia* 的设备：

```
cd /dev && ls -al
```

如果没有，创建一个 nv.sh：

```
vi nv.sh
#!/bin/bash /sbin/modprobe nvidia
if [ "$?" -eq 0 ];
then
NVDEVS=`lspci |
grep -i NVIDIA
` 

N3D=`
echo
"$NVDEVS"
| grep "3D controller" |
wc -l
` 

NVGA=`
echo
"$NVDEVS"
| grep "VGA compatible controller" |
wc -l
` 

N=
expr $N3D + $NVGA -
1
` 

for i in `seq
0
$N
` ; do
    mknod -m 666 /dev/nvidia$i c 195 $i
done
    mknod -m 666 /dev/nvidiactl c 195 255
else
    exit 1
fi

chmod +x nv.sh && bash nv.sh
```

再次重启机器查看 Cuda 版本：

```
nvcc -V
```

编译：

```
cd /usr/local/cuda-10.1/samples && make

cd /usr/local/cuda-10.1/samples/bin/x86_64/linux/release ./deviceQuery
```

以上如果输出 “Result = PASS” ，代表 Cuda 安装成功。

安装 nvdocker:

```
vim /etc/docker/daemon.json
{
  "runtimes": {
    "nvidia": {
      "path": "nvidia-container-runtime",
      "runtimeArgs": []
    }
  },
  "registry-mirrors": ["https://registry.docker-cn.com"],
  "storage-driver": "overlay2",
  "default-runtime": "nvidia",
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "exec-opts": ["native.cgroupdriver=systemd"],
  "insecure-registries": [$harborRegistry],
  "live-restore": true
}
```

重启 Docker:

```
sudo systemctl daemon-reload && sudo systemctl restart docker && docker info
```

检查 nvidia-docker 安装是否成功:

```
docker run --runtime=nvidia --rm nvidia/cuda:9.0-base nvidia-smi
```

在节点机器进入 su 模式:

```
su $nodeUser
```

给当前节点用户配置 K8s apiserver 访问公钥:

```
mkdir -p $HOME/.kube
cp -i admin.conf $HOME/.kube/config
chown $(id -u):$(id -g) $HOME/.kube/config
mkdir -p $HOME/etc
sudo rm -rf /etc/kubernetes
sudo mkdir -p /etc/kubernetes/pki/etc
sudo cp /home/$nodeUser/etc/* /etc/kubernetes/pki/etc
```

```
sudo kubeadm join $masterIP:6443 --token $token_name --discovery-token-ca-cert-hash  
$hash
```

如：

```
sudo kubeadm join 192.168.8.116:6443 --token vyi4ga.foyxqr2iz9i391q3 --discovery-token-  
ca-cert-hash sha256:929143bcdaa3e23c6faf20bc51ef6a57df02edf9df86cedf200320a9b4d3220a
```

检查 node 是否加入 master：

```
kubectl get node
```

到此，K8s 单机、集群版部署流程就结束了，后面我们会将 K8s 与微服务结合起来分析 K8s 的组件的特性。

5.3 Kubernetes 的组件及负载均衡

上面介绍了 K8s 的由来、概念以及特性，接下来，我们看看 K8s 的组件。K8s 的组件分为 Master 组件、Node 组件。

Master 组件

1、kube-apiserver

Kubernetes API 服务器的主要实现是 kube-apiserver。kube-apiserver 设计上考虑了水平伸缩，也就是说，它可通过部署多个实例进行伸缩。你可以运行 kube-apiserver 的多个实例，并在这些实例之间平衡流量。

2、etcd

etcd 是兼具一致性和高可用性的键值数据库，可以作为保存 Kubernetes 所有集群数据的后台数据库。通常，集群的 etcd 数据库通常需要有个备份计划。

3、kube-controller-manager

在主节点上运行控制器的组件。

4、cloud-controller-manager

cloud-controller-manager 仅运行特定于云平台的控制回路。如果你在自己的环境中运行 Kubernetes，或者在本地计算机中运行学习环境，所部署的环境中不需要云控制器管理器。

5、kube-scheduler

控制平面组件，负责监视新创建的、未指定运行节点（node）的 Pods，选择节点让 Pod 在上面运行。调度决策考虑的因素包括单个 Pod 和 Pod 集合的资源需求、硬件/软件/策略约束、亲和性和反亲和性规范、数据位置、工作负载间的干扰和最后时限。

Node 组件

节点组件在每个节点上运行，维护运行的 Pod 并提供 Kubernetes 运行环境。如果 Master 也被设置允许为工作节点，则节点组件同样运行在 Master 上。

1、kubelet

一个在集群中每个节点（node）上运行的代理。它保证容器（containers）都运行在 Pod 中。

2、kube-proxy

kube-proxy 是集群中每个节点上运行的网络代理，实现 Kubernetes 服务（Service）概念的一部分。kube-proxy 维护节点上的网络规则。这些网络规则允许从集群内部或外部的网络会话与 Pod 进行网络通信。

- 1.kube-proxy 主要是处理集群外部通过 nodePort 访问集群内服务，通过 iptables 规则，解析 clusterIP 到 PodIP 的过程，并提供服务的负载均衡能力。
- 2.kube-proxy 还可以提供集群内部服务间通过 clusterIP 访问，也会经过 kube-proxy 负责转发。
- 3.kube-dns 主要在 Pod 内通过 serviceName 访问其他服务，找到服务对应的关系，和一些基本的域名解析功能。
- 4.kube-dns 是和 kube-proxy 协同工作的，前者通过 servicename 找到指定 clusterIP，后者完成通过 clusterIP 到 PodIP 的过程。
- 这里，K8s 通过虚拟出一个集群 IP，利用 kube-proxy 为 service 提供 cluster 内的服务发现和负载均衡。

3、Container Runtime

容器运行环境是负责运行容器的软件。Kubernetes 支持多个容器运行环境：Docker、containerd、CRI-O 以及任何实现 Kubernetes CRI（容器运行环境接口）。

4、插件 Addons

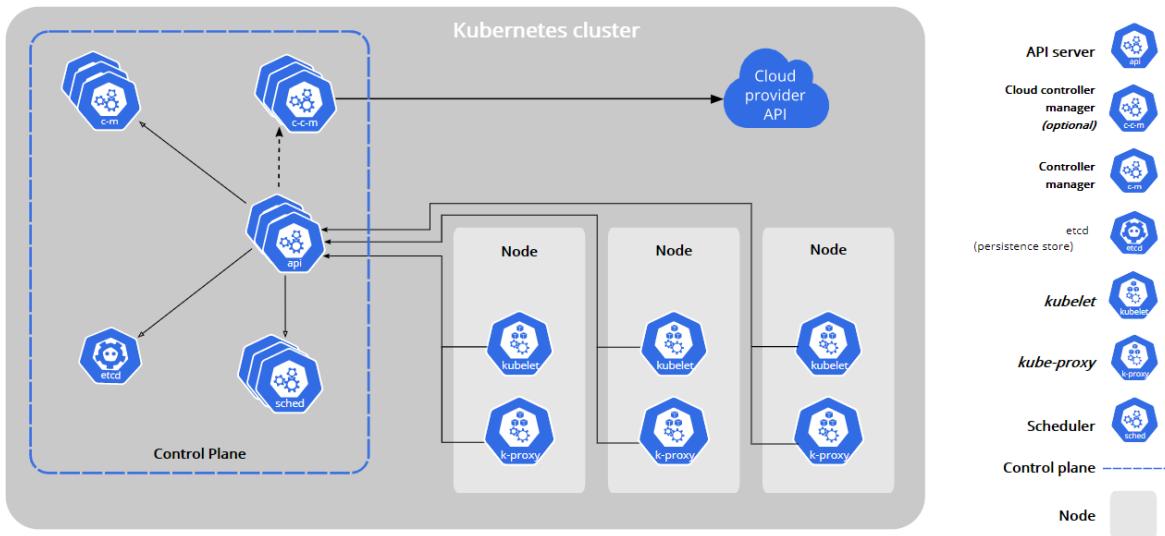
插件使用 Kubernetes 资源（DaemonSet、Deployment 等）实现集群功能。因为这些插件提供集群级别的功能，插件中命名空间域的资源属于 kube-system 命名空间。

5、DNS

尽管其他插件都并非严格意义上的必需组件，但几乎所有 Kubernetes 集群都应该有集群 DNS，因为很多示例都需要 DNS 服务。

6、Dashboard

Dashboard 是 Kubernetes 集群的通用的、基于 Web 的用户界面。它使用户可以管理集群中运行的应用程序以及集群本身并进行故障排除。



K8s如何实现服务注册与发现

上面介绍了K8s的各种组件，接下来，我们看看K8s是如何实现服务的注册与发现，然后如何做到服务的转发、实现负载均衡的能力。

服务在K8s中，也定义了一种资源：Service，Service，顾名思义是一个服务，什么样的服务呢？它是定义了一个服务的多种 pod 的逻辑合集以及一种访问 pod 的策略。

service 的类型有四种：

- **ExternalName:** 创建一个 DNS 别名指向 service name，这样可以防止 service name 发生变化，但需要配合 DNS 插件使用。
- **ClusterIP:** 默认的类型，用于为集群内 Pod 访问时，提供的固定访问地址，默认是自动分配地址，可使用 ClusterIP 关键字指定固定 IP。
- **NodePort:** 基于 ClusterIP，用于为集群外部访问 Service 后面 Pod 提供访问接入端口。
- **LoadBalancer:** 它是基于 NodePort。

从上面讲的 Service，我们可以看到一种场景：所有的微服务在一个局域网内，或者说在一个 K8s 集群下，那么可以通过 Service 用于集群内 Pod 的访问，这就是 Service 默认的一种类型 ClusterIP，ClusterIP 这种的默认会自动分配地址。

那么问题来了，既然可以通过上面的 ClusterIP 来实现集群内部的服务访问，那么如何注册服务呢？其实 K8s 并没有引入任何的注册中心，使用的就是 K8s 的 kube-dns 组件。然后 K8s 将 Service 的名称当做域名注册到 kube-dns 中，每一个 Service 在 kube-dns 中都有一条 DNS 记录，同时，如果有服务的 IP 更换，kube-dns 自动会同步，对服务来说是不需要改动的。通过 Service 的名称就可以访问其提供的服务。那么问题又来了，如果一个服务的 pod 对应有多个，那么如何实现 LB？其实，最终通过 kube-proxy，实现负载均衡。也就是说 kube-dns 通过 servicename 找到指定 clusterIP，kube-proxy 完成通过 clusterIP 到 PodIP 的过程。

说到这，我们来看下 Service 的服务发现与负载均衡的策略，Service 负载分发策略有两种：

- **RoundRobin:** 轮询模式，即轮询将请求转发到后端的各个 pod 上，其为默认模式。

- SessionAffinity: 基于客户端 IP 地址进行会话保持的模式, 类似 IP Hash 的方式, 来实现服务的负载均衡。

下面写一个很简单的例子:

```
apiVersion: v1
kind: Service
metadata:
  name: cas-server-service
  namespace: default
spec:
  ports:
    - name: cas-server01
      port: 2000
      targetPort: cas-server01
  selector:
    app: cas-server
```

可以看到执行 `kubectl apply -f service.yaml` 后:

```
root@ubuntu:~$ kubectl get svc
NAME          TYPE        CLUSTER-IP   EXTERNAL-IP  PORT(S)
AGE
admin-web-service   ClusterIP  10.16.129.24 <none>       2001/TCP
84d
cas-server-service   ClusterIP  10.16.230.167 <none>       2000/TCP
67d
cloud-admin-service-service ClusterIP  10.16.25.178 <none>       1001/TCP
190d
```

这样, 我们可以看到默认的类型是 ClusterIP, 用于为集群内 Pod 访问时, 可以先通过域名来解析到多个服务地址信息, 然后再通过 LB 策略来选择其中一个作为请求的对象。

K8s 如何处理微服务中常用的配置

接下来我们看看微服务中场景的居多配置该如何来利用K8s实现统一管理。其实, 在K8s中, 定义了一种资源: ConfigMap, 我们来看看这种资源。

ConfigMap, 看到这个名字可以理解: 它是用于保存配置信息的键值对, 可以用来保存单个属性, 也可以保存配置文件。对于一些非敏感的信息, 比如应用的配置信息, 则可以使用 ConfigMap。

创建一个 ConfigMap 有多种方式如下。

1. key-value 字符串创建

```
kubectl create configmap test-config --from-literal=baseDir=/usr
```

上面的命令创建了一个名为 test-config，拥有一条 key 为 baseDir，value 为 "/usr" 的键值对数据。

2. 根据 yml 描述文件创建

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: test-config
data:
  baseDir: /usr
```

也可以这样，创建一个 yml 文件，选择不同的环境配置不同的信息：

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: cas-server
data:
  application.yaml: |-  
    greeting:  
      message: Say Hello to the World  
  
    spring:  
      profiles: dev  
      greeting:  
        message: Say Hello to the Dev  
    spring:  
      profiles: test  
      greeting:  
        message: Say Hello to the Test  
    spring:  
      profiles: prod  
      greeting:  
        message: Say Hello to the Prod
```

注意点：

- ConfigMap 必须在 Pod 使用其之前创建。
- Pod 只能使用同一个命名空间的 ConfigMap。

当然，还有其他更多用途，具体可以参考官网
(<https://kubernetes.io/zh/docs/concepts/configuration/configmap/>)。

前面讲述了几种创建ConfigMap的方式，其中有一种在 Java 中常常用到：通过创建 yml 文件来实现配置管理。比如：

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: cas-server
```

```
data:  
  application.yaml: |-  
    greeting:  
      message: Say Hello to the World  
    ---  
    spring:  
      profiles: dev  
    greeting:  
      message: Say Hello to the Dev  
    spring:  
      profiles: test  
    greeting:  
      message: Say Hello to the Test  
    spring:  
      profiles: prod  
    greeting:  
      message: Say Hello to the Prod
```

这样，当我们启动容器时，通过 `--spring.profiles.active=dev` 来指定当前容器的活跃环境，即可获取 ConfigMap 中对应的配置。是不是感觉跟 Java 中的 Config 配置多个环境的配置有点类似呢？但是，我们不用那么复杂，这些统统可以交给 K8s 来处理。只需要你启动这一命令即可，是不是很简单？

第六章 为什么选择 Kubernetes

6.1 Kubernetes 与微服务的天生绝配

其实，为什么我们需要 K8s，它到底能做什么呢？

容器是打包和运行应用程序的好方式。在生产环境中，你需要管理运行应用程序的容器，并确保不会停机。例如，如果一个容器发生故障，则需要启动另一个容器。如果系统处理此行为，会不会更容易？这就是 Kubernetes 来解决这些问题的方法！Kubernetes 为你提供了一个可弹性运行分布式系统的框架。Kubernetes 会满足你的扩展要求、故障转移、部署模式等。例如，Kubernetes 可以轻松管理系统的 Canary 部署。

Kubernetes 会提供：

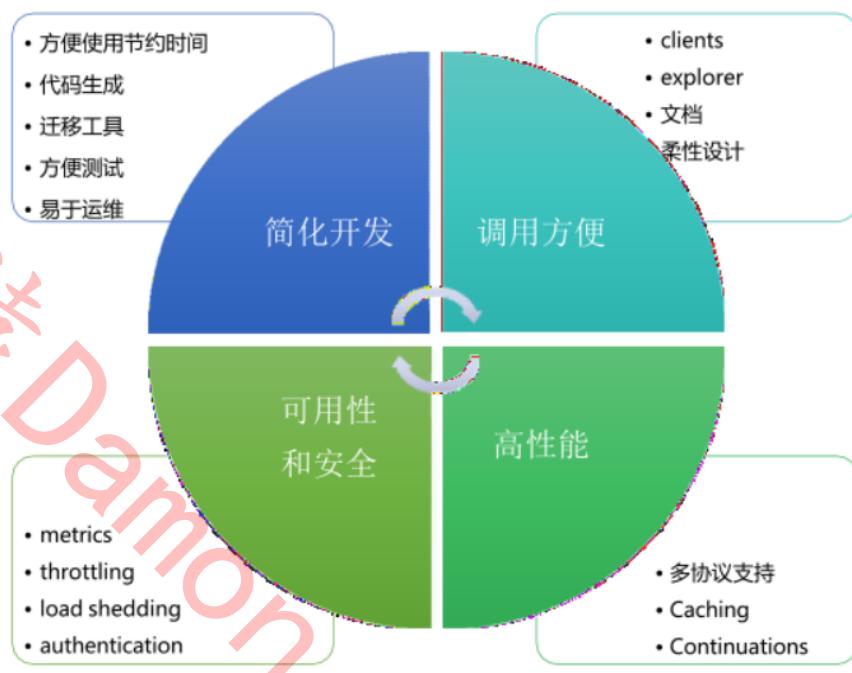
- 服务发现和负载均衡，Kubernetes 可以使用 DNS 名称或自己的 IP 地址公开容器，如果进入容器的流量很大，Kubernetes 可以负载均衡并分配网络流量，从而使部署稳定。
- 自动部署和回滚，你可以使用 Kubernetes 描述已部署容器的所需状态，它可以以受控的速率将实际状态更改为期望状态。例如，你可以自动化 Kubernetes 来为你的部署创建新容器，删除现有容器并将它们的所有资源用于新容器。
- 自我修复，Kubernetes 重新启动失败的容器、替换容器、杀死不响应用户定义的运行状况检查的容器，并且在准备好服务之前不将其通告给客户端。
- 存储编排，Kubernetes 允许你自动挂载你选择的存储系统，例如本地存储、公共云提供商等。
- 自动完成装箱计算，Kubernetes 允许你指定每个容器所需 CPU 和内存（RAM）。当容器指定了资源请求时，Kubernetes 可以做出更好的决策来管理容器的资源。

- 密钥与配置管理，Kubernetes 允许你存储和管理敏感信息，例如密码、OAuth 令牌和 ssh 密钥。你可以在不重建容器镜像的情况下部署和更新密钥和应用程序配置，也无需在堆栈配置中暴露密钥。

那么对于 K8s 提供的这些功能，其实对于微服务来讲，都是很好的一个平台提供。换句话说，对于微服务来说，如果使用 K8s 的话，可以不用考虑语言上的限制，更不用考虑各种开发语言的框架的限制；对于各种语言来说，在前面也介绍过，都有很多不同的框架，那如果运用这些框架时，就需要考虑不同服务之间如果属于不同的语言，那么该如何来实现微服务的架构呢？从这一角度来分析，微服务与 K8s 属于天作之合，它们的结合可以说是天衣无缝、完美至极。在后面章节中，将会介绍它们的天衣无缝：自治与无缝迁移。

6.2 基于 Kubernetes 集群的服务治理

其实，做微服务架构设计，我们希望得到什么呢？看下图：



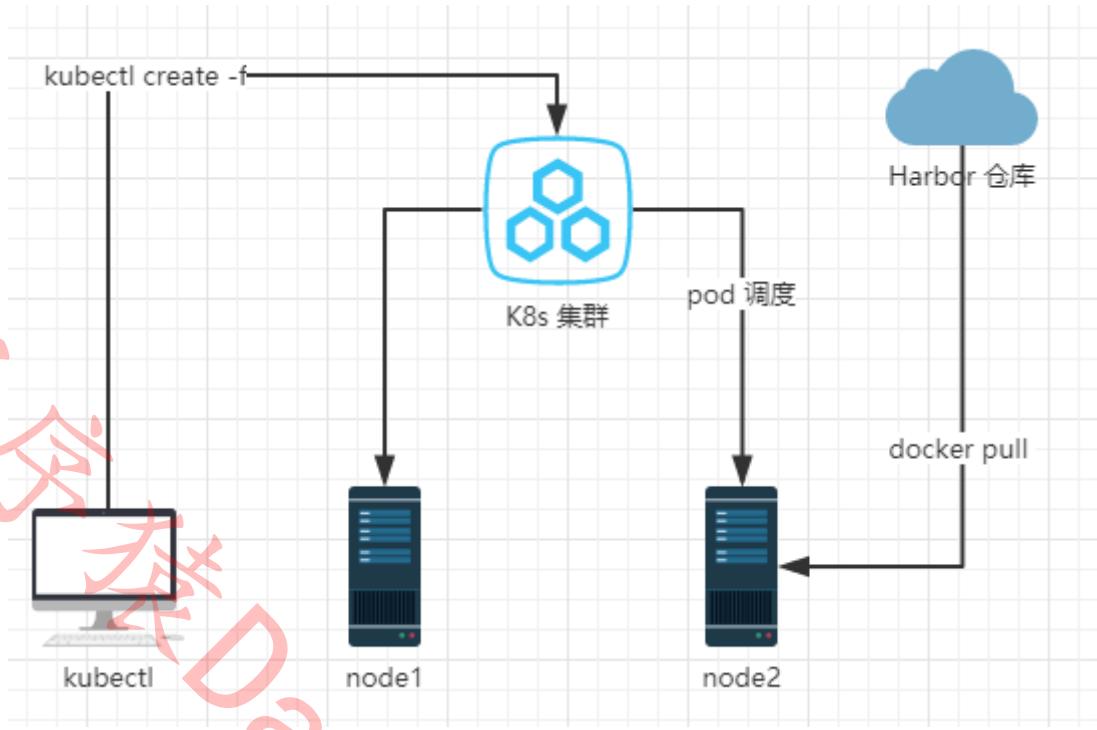
从上面这张图中可以看到，微服务的解耦、封装，从而简化开发人员的开发。调用方便，主要体现在sdk或者说client的提供者很容易被调用，这就体现了K8s的服务注册与发现。安全性考虑，基于K8s集群的保障，可以让微服务们处于一个堡垒中，这样避免外部的干扰。同时，服务之间直接走内部网络，可以大大提升性能。

说到这些，其实服务的自动化才是一个重点，自治能力体现了系统的健壮性。在前面章节中说到了K8s具有自动修复的能力，可以将失败或出现问题的容器进行重新编排、启动。容器本身的健康检查会被监视，当不响应用户定义的运行检查的容器就会被杀死。

同时，K8s提供自动部署能力，可以通过简单的命令来执行即可发挥K8s的作用。同时，K8s会根据用户的节点选择，将pod分配到对应的节点，这样对于运维人员来说，即使出现节点宕机，pod可以迅速的在其他节点被启动。

6.3 基于 Kubernetes 的服务无缝迁移

在K8s集群中的服务，如果想要被迁移到其他的机器或其他集群。在传统的实现中，可能需要考虑到很多点：服务包的转移、共享，配置的转移，数据库的转移等等。但对于容器化来说，这些都被打包成image，而这些image可以被上传到一个仓库Habor，当需要迁移环境的时候，这些服务的镜像其实都可以不动，运维人员将要做的是：将开发人员编写的基于K8s的yaml文件在对应的集群中进行部署即可。运维人员无需关心任何其他事情，只需要在部署前，将需要的相关配置处理好即可。这将大大减少开发人员、运维的成本，让他们专注于部署，而不用关心其他的琐碎的事情。因为K8s是可以跨平台、跨系统的。主要存在K8s集群，服务都可以无缝的进行迁移过去。这也是微服务基于K8s的一个优势。下图为K8s基于Habor的架构图。



- 通过kubectl命令工具发起资源创建kubectl create -f xxx.yaml
- k8s处理相关请求后kube-scheduler服务为pod寻找一个合适的“家”node2并创建pod。
- node2上的kubelet处理相关资源，使用docker拉取相关镜像并运行。

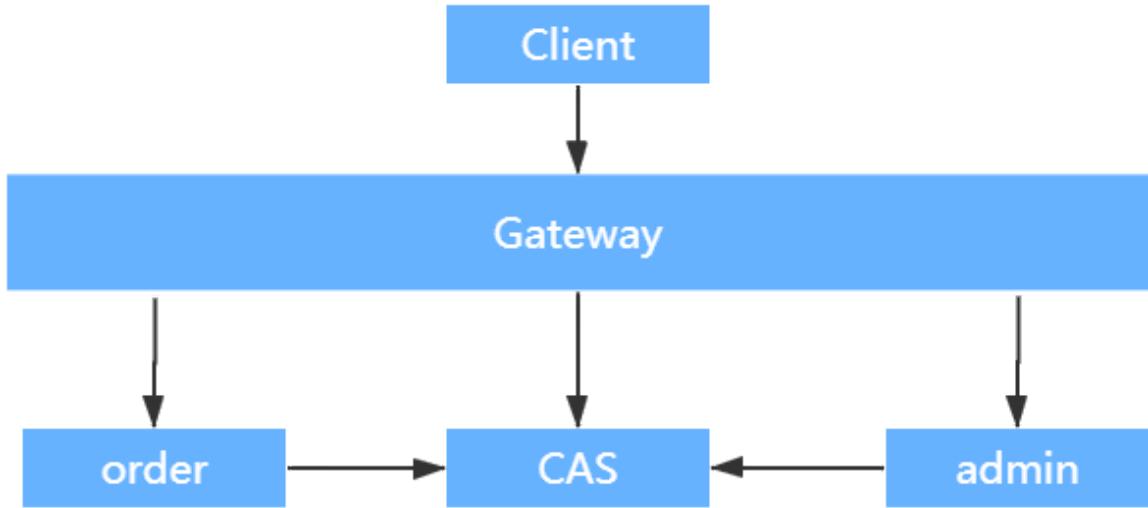
从上面的流程操作来看，将微服务从一个K8s集群，迁移到另一个集群，其操作是可以无缝对接的，可以同配置、同环境参数的无缝的迁移，这就是云原生下K8s带来的优势。也是需要企业现在一直推荐属性docker、K8s等技术的一个关键性要素。

第七章 第一个基于 K8s 的多语言微服务架构

7.1 基于 K8s 的 Java 微服务

前面很多的都是在说K8s为什么可以实现配置化，为什么可以提供负载均衡能力，接下来，我们举个电商系统来实现体验下K8s带来的效果，手写Java代码。

如下图，我们简单的画了一个系统图，从客户端到网关，再到订单服务、后台管理系统以及鉴权中心的流程。



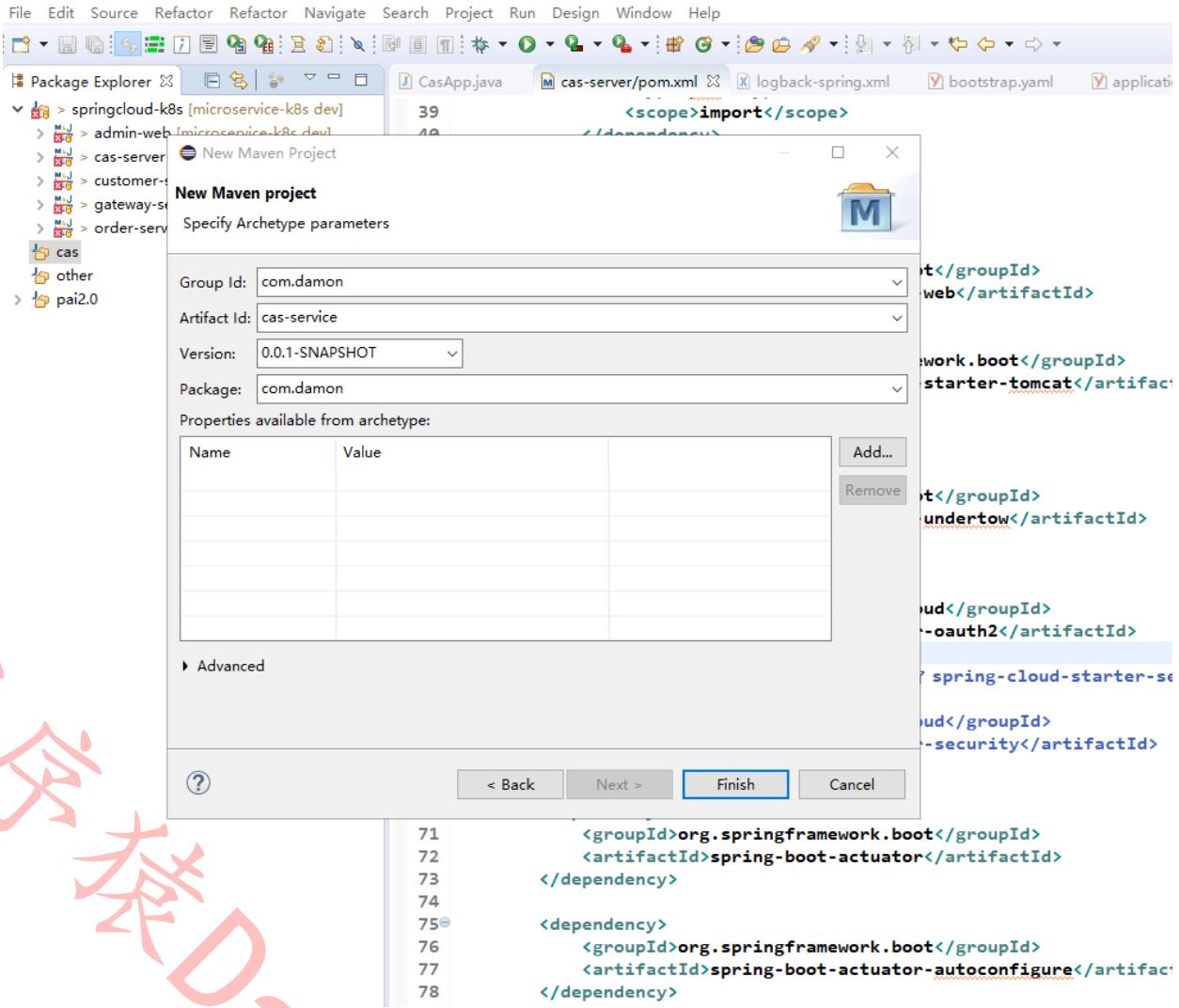
认证中心

基于Java现在许多比较流行的框架，作者选择了SpringCloud，因为很多基础特性SpringCloud都具备了，接下来我们看如何实现鉴权。

环境：

- Ubuntu 16.04
- Docker 18.06
- K8s 1.13.5
- springboot 2.3.8.RELEASE
- springcloud Hoxton.SR9

首先新建一个Java项目：



引入依赖配置：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-actuator</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-actuator-autoconfigure</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-kubernetes-config</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.0</version>
</dependency>
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
    <version>4.6.3</version>
</dependency>

<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>19.0</version>
</dependency>

<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
</dependency>

<dependency>
    <groupId>commons-collections</groupId>
    <artifactId>commons-collections</artifactId>
    <version>3.2.2</version>
</dependency>

<!-- mybatis -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.1.1</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>${mysql.version}</version>
</dependency>

<!-- datasource pool-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.3</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

新建服务启动类：

```
1 package com.damon;
2
3 import org.springframework.boot.SpringApplication;
4
5 /**
6  * 配置最多的就是认证服务端，验证账号、密码，存储 token，检查 token，刷新 token 等都是认证服务端的工作
7  * @author Damon
8  * @date 2020年1月13日下午2:29:42
9  */
10
11 @Configuration
12 @EnableAutoConfiguration
13 @ComponentScan(basePackages = {"com.damon"})
14 //@SpringBootApplication(scanBasePackages = { "com.damon" })
15 @EnableConfigurationProperties(EnvConfig.class)
16 public class CasApp {
17     public static void main(String[] args) {
18         SpringApplication.run(CasApp.class, args);
19     }
20 }
```

常见的配置文件bootstrap.yml、application.yml，主要是配置服务启动时，需要加载的参数、配置：

```
management:
  endpoint:
    restart:
      enabled: true
  health:
    enabled: true
  info:
    enabled: true

spring:
  application:
    name: cas-server
  cloud:
    kubernetes:
      config:
        sources:
          - name: ${spring.application.name}
            namespace: system-server
    discovery:
      all-namespaces: true
  reload:
    #自动更新配置的开关设置为打开
    enabled: true
    #更新配置信息的模式: polling是主动拉取, event是事件通知
    mode: polling
    #主动拉取的间隔时间是500毫秒
```

```
period: 500

redis: #redis相关配置
  database: 8
  host: 10.10.3.15 #Localhost
  port: 6379
  password: xxxxx #有密码时设置
jedis:
  pool:
    max-active: 8
    max-idle: 8
    min-idle: 0
  timeout: 10000ms

http:
  encoding:
    charset: UTF-8
    enabled: true
    force: true
  mvc:
    throw-exception-if-no-handler-found: true
  main:
    allow-bean-definition-overriding: true

logging:
  path: /data/${spring.application.name}/logs
```

第一个配置中，介绍了该服务的信息，以及springboot2结合K8s的一个特性：自动刷新、加载配置，该模式有两种：

- 主动拉取：polling，每隔一定时间拉取，自定义设置。
- 事件通知，event

这里主要用主动拉取模式。后面就是配置一些插件redis、日志配置。

第二个配置文件可以设置一些环境、mybatis配置以及设置http协议的超时：

```
spring:
  profiles:
    active: dev

server:
  port: 2000
  undertow:
    accesslog:
      enabled: false
      pattern: combined
  servlet:
    session:
      timeout: PT120M

client:
  http:
    request:
```

```
connectTimeout: 8000
readTimeout: 30000

mybatis:
  mapperLocations: classpath:mapper/*.xml
  typeAliasesPackage: com.damon.*.model
```

由于我们可以设置**spring.profiles.active=dev**, 所以可以设置几个不同环境的文件来设置日志的级别。当然, 你也可以直接在启动服务时, 设置参数:

```
-- spring.profiles.active=dev --logging.level.org.springframework.web=INFO --
logging.level.com.damon=INFO
```

到目前为止, 一些基础配置都已经写完了, 接下来, 我们看看鉴权的核心逻辑了。首先我们来写一个认证服务器配置:

```
package com.damon.config;

import java.util.ArrayList;
import java.util.List;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.oauth2.config.annotation.builders.ClientDetailsServiceBuilder;
import
org.springframework.security.oauth2.config.annotation.builders.InMemoryClientDetailsServiceBuilder;
import
org.springframework.security.oauth2.config.annotation.configurers.ClientDetailsServiceConfigurer;
import
org.springframework.security.oauth2.config.annotation.web.configuration.AuthorizationServerConfigurerAdapter;
import
org.springframework.security.oauth2.config.annotation.web.configuration.EnableAuthorizationOnServer;
import
org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerEndpointsConfigurer;
import
org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerSecurityConfigurer;
import org.springframework.security.oauth2.provider.error.WebResponseExceptionTranslator;
import org.springframework.security.oauth2.provider.token.TokenEnhancer;
import org.springframework.security.oauth2.provider.token.TokenEnhancerChain;
```

```
import org.springframework.security.oauth2.provider.token.TokenStore;
import org.springframework.security.oauth2.provider.token.store.JwtAccessTokenConverter;

import com.damon.component.JwtTokenEnhancer;
import com.damon.login.service.LoginService;

/**
 *
 * 认证服务器配置
 * @author Damon
 * @date 2020年1月13日 下午3:03:30
 *
 */
@Configuration
@EnableAuthorizationServer
public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private LoginService loginService;

    @Autowired
    @Qualifier("jwtTokenStore")
    @Qualifier("redisTokenStore")
    private TokenStore tokenStore;
    /*@Autowired
    private JwtAccessTokenConverter jwtAccessTokenConverter;
    @Autowired
    private JwtTokenEnhancer jwtTokenEnhancer;*/

    @Autowired
    private Environment env;

    @Autowired
    private DataSource dataSource;

    @Autowired
    private WebResponseExceptionTranslator userOAuth2WebResponseExceptionTranslator;

    /* @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) {
        TokenEnhancerChain enhancerChain = new TokenEnhancerChain();
        List<TokenEnhancer> delegates = new ArrayList<>();
        delegates.add(jwtTokenEnhancer); //配置JWT的内容增强器
        delegates.add(jwtAccessTokenConverter);
        enhancerChain.setTokenEnhancers(delegates);
        endpoints.authenticationManager(authenticationManager)//支持 password 模式
            .userDetailsService(loginService)
            .tokenStore(tokenStore) //配置令牌存储策略
            .accessTokenConverter(jwtAccessTokenConverter)
            .tokenEnhancer(enhancerChain);
    }*/
```

```
    }*/

    /**
     * redis token 方式
     */
    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws
Exception {
    //验证时发生的情况处理
    endpoints.authenticationManager(authenticationManager) //支持 password 模式
        .exceptionTranslator(userOAuth2WebResponseExceptionTranslator)//自定义异常
    处理类添加到认证服务器配置
        .userDetailsService(loginService)
        .tokenStore(tokenStore);

}

/**
 * 客户端配置 (给谁发令牌)
 * 不同客户端配置不同
 *
 * authorizedGrantTypes 可以包括如下几种设置中的一种或多种:
    authorization_code: 授权码类型。需要redirect_uri
    implicit: 隐式授权类型。需要redirect_uri
    password: 资源所有者 (即用户) 密码类型。
    client_credentials: 客户端凭据 (客户端ID以及Key) 类型。
    refresh_token: 通过以上授权获得的刷新令牌来获取新的令牌。
    accessTokenValiditySeconds: token 的有效期
    scopes: 用来限制客户端访问的权限, 在换取的 token 的时候会带上 scope 参数, 只有在 scopes
    定义内的, 才可以正常换取 token。
    * @param clients
    * @throws Exception
    * @author Damon
    * @date 2020年1月13日
    *
    */
    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient("admin-web")
            .secret(passwordEncoder.encode("admin-web-123"))
            .accessTokenValiditySeconds(3600)
            .refreshTokenValiditySeconds(864000)//配置刷新token的有效期
            .autoApprove(true) //自动授权配置
            .scopes("all")//配置申请的权限范围
            .authorizedGrantTypes("password", "authorization_code",
"client_credentials", "refresh_token")//配置授权模式
            .redirectUris("http://localhost:2001/login")//授权码模式开启后必须指定

            .and()
            .withClient("order-service")
            .secret(passwordEncoder.encode("order-service-123"))
            .accessTokenValiditySeconds(3600)
            .refreshTokenValiditySeconds(864000)//配置刷新token的有效期
            .autoApprove(true) //自动授权配置
```

```

        .scopes("all")
        .authorizedGrantTypes("password", "authorization_code",
"client_credentials", "refresh_token")//配置授权模式
        .redirectUris("http://localhost:2003/login")//授权码模式开启后必须指定

        .and()
        .withClient("customer-service")
        .secret(passwordEncoder.encode("customer-service-123"))
        .accessTokenValiditySeconds(3600)
        .refreshTokenValiditySeconds(864000)//配置刷新token的有效期
        .autoApprove(true) //自动授权配置
        .scopes("all")
        .authorizedGrantTypes("password", "authorization_code",
"client_credentials", "refresh_token")//配置授权模式
        .redirectUris("http://localhost:6000/login")//授权码模式开启后必须指定
        ;
    }

@Override
public void configure(AuthorizationServerSecurityConfigurer security) {
    security.allowFormAuthenticationForClients();//是允许客户端访问 OAuth2 授权接口，否则请求 token 会返回 401
    security.checkTokenAccess("isAuthenticated()");//是允许已授权用户访问 checkToken 接口
    security.tokenKeyAccess("isAuthenticated()"); //
    security.tokenKeyAccess("permitAll()");获取密钥需要身份认证，使用单点登录时必须配置，是允许已授权用户获取 token 接口
}
}

```

在这个配置中，我们redis来记录token，自定义了登录认证的逻辑LoginService，自定义异常处理类添加到认证服务器配置。同时函数configure加载了居多需要认证的客户端服务，配置了授权模式：“password”、“authorization_code”、“client_credentials”、“refresh_token”，配置刷新token的有效期。

我们先来看看自定义的认证逻辑，先来看看接口类：

```

package com.damon.login.service;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;

import com.damon.commons.Response;

/**
 * @author Damon
 * @date 2018年11月15日 上午11:59:24
 */

```

```

public interface LoginService extends UserDetailsService {

    /**
     *
     * Spring Security默认函数
     * @param username
     * @return
     * @throws UsernameNotFoundException
     * @author Damon
     * @date 2020年1月13日
     *
     */
    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

    //以下自定义

    Response<Object> login(String username, String password);

    /**
     *
     * 校验token合法性
     * @param request
     * @param token
     * @return
     * @author Damon
     * @date 2019年8月15日
     *
     */
    Response<Object> verify(HttpServletRequest request, String token);

    Response<Object> updatePwd(HttpServletRequest req, String username, String oldPwd,
String newPwd);

    //Response<Object> logout(HttpServletRequest req, HttpServletResponse res);

}

```

在Spring Security中，有默认的函数**loadUserByUsername**来实现鉴权逻辑：

```

/**
 * Auth
 * 登录认证
 * 实际中从数据库获取信息
 * 这里为了做演示，把用户名、密码和所属角色都写在代码里了，正式环境中，这里应该是从数据库或者其他地方根据用户名将加密后的密码及所属角色查出来的。账号 damon ,
 * 密码123456，稍后在换取 token 的时候会用到。并且给这个用户设置 "ROLE_ADMIN" 角色。
 *
 */
@Override
public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
    logger.info("clientIp is: {} ,username: {}", IpUtil.getClientIp(req), username);
    logger.info("serverIp is: {}", IpUtil.getCurrentIp());
    // 查询数据库操作
}

```

```

SysUser user = userMapper.getUserByUsername(username);
if (user == null) {
    logger.error("user not exist");
    throw new UsernameNotFoundException("username is not exist");
}
else {
    // 用户角色也应在数据库中获取，这里简化
    String role = "";
    if(user.getisAdmin() == 1) {
        role = "admin";
    }
    List<SimpleGrantedAuthority> authorities = Lists.newArrayList();
    authorities.add(new SimpleGrantedAuthority(role));
    //String password = passwordEncoder.encode("123456");// 123456是密码
    //return new User(username, password, authorities);
    // 线上环境应该通过用户名查询数据库获取加密后的密码
    return new User(username, user.getPassword(), authorities);
}
}

```

可以看到通过用户信息获取用户的权限，然后记录用户信息。这里使用redis来记录用户信息以及token信息：

```

package com.damon.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.security.oauth2.provider.token.TokenStore;
import org.springframework.security.oauth2.provider.token.store.redis.RedisTokenStore;

/**
 * 使用redis存储token的配置
 * @author Damon
 * @date 2020年1月13日 下午3:03:19
 *
 */
@Configuration
public class RedisTokenStoreConfig {

    @Autowired
    private RedisConnectionFactory redisConnectionFactory;

    @Bean
    public TokenStore redisTokenStore (){
        //return new RedisTokenStore(redisConnectionFactory);
        return new MyRedisTokenStore(redisConnectionFactory);
    }
}

```

同时，我们自定义了异常处理类，统一处理异常：

```
package com.damon.config;

import java.io.IOException;

import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.oauth2.common.DefaultThrowableAnalyzer;
import org.springframework.security.oauth2.common.exceptions.InsufficientScopeException;
import org.springframework.security.oauth2.common.exceptions.OAuth2Exception;
import org.springframework.security.oauth2.provider.error.WebResponseExceptionTranslator;
import org.springframework.security.web.util.ThrowableAnalyzer;
import org.springframework.stereotype.Component;
import org.springframework.web.HttpRequestMethodNotSupportedException;

import com.damon.exception.UserOAuth2Exception;

/**
 *
 * 自定义异常转换类
 * @author Damon
 * @date 2020年2月27日 上午10:28:19
 */
@Component("userOAuth2WebResponseExceptionTranslator")
public class UserOAuth2WebResponseExceptionTranslator implements
WebResponseExceptionTranslator {
    private ThrowableAnalyzer throwableAnalyzer = new DefaultThrowableAnalyzer();

    @Override
    public ResponseEntity<OAuth2Exception> translate(Exception e) throws Exception {
        Throwable[] causeChain = this.throwableAnalyzer.determineCauseChain(e);
        Exception ase =
(OAuth2Exception)this.throwableAnalyzer.getFirstThrowableOfType(OAuth2Exception.class,
causeChain);
        //异常链中有OAuth2Exception异常
        if (ase != null) {
            return this.handleOAuth2Exception((OAuth2Exception)ase);
        }
        //身份验证相关异常
        ase =
(AuthenticationException)this.throwableAnalyzer.getFirstThrowableOfType(AuthenticationException.class,
causeChain);
        if (ase != null) {
            return this.handleOAuth2Exception(new
UserOAuth2WebResponseExceptionTranslator.UnauthorizedException(e.getMessage(), e));
        }
        //异常链中包含拒绝访问异常
        ase =
(AccessDeniedException)this.throwableAnalyzer.getFirstThrowableOfType(AccessDeniedException.class,
causeChain);
        if (ase instanceof AccessDeniedException) {
            return this.handleOAuth2Exception(new
```

```
UserOAuth2WebResponseExceptionTranslator.ForbiddenException(ase.getMessage(), ase));
    }
    //异常链中包含Http方法请求异常
    ase =
(HttpServletRequestMethodNotSupportedException)this.throwableAnalyzer.getFirstThrowableOfType(HttpServletRequestMethodNotSupportedException.class, causeChain);
    if(ase instanceof HttpServletRequestMethodNotSupportedException){
        return this.handleOAuth2Exception(new
UserOAuth2WebResponseExceptionTranslator.MethodNotAllowed(ase.getMessage(), ase));
    }
    return this.handleOAuth2Exception(new
UserOAuth2WebResponseExceptionTranslator.ServerErrorException(HttpStatus.INTERNAL_SERVER_ERROR.getReasonPhrase(), e));
}

private ResponseEntity<OAuth2Exception> handleOAuth2Exception(OAuth2Exception e)
throws IOException {
    int status = e.getHttpErrorCode();
    HttpHeaders headers = new HttpHeaders();
    headers.set("Cache-Control", "no-store");
    headers.set("Pragma", "no-cache");
    if (status == HttpStatus.UNAUTHORIZED.value() || e instanceof
InsufficientScopeException) {
        headers.set("WWW-Authenticate", String.format("%s %s", "Bearer",
e.getSummary()));
    }
    OAuth2Exception exception = new OAuth2Exception(e.getMessage(),e);
    ResponseEntity<OAuth2Exception> response = new ResponseEntity(exception, headers,
HttpStatus.valueOf(status));
    return response;
}

private static class MethodNotAllowed extends OAuth2Exception {
    public MethodNotAllowed(String msg, Throwable t) {
        super(msg, t);
    }
    @Override
    public String getOAuth2ErrorCode() {
        return "method_not_allowed";
    }
    @Override
    public int getHttpErrorCode() {
        return 405;
    }
}

private static class UnauthorizedException extends OAuth2Exception {
    public UnauthorizedException(String msg, Throwable t) {
        super(msg, t);
    }
    @Override
    public String getOAuth2ErrorCode() {
        return "unauthorized";
    }
    @Override
    public int getHttpErrorCode() {
        return 401;
    }
}
```

```

    }

}

private static class ServerErrorException extends OAuth2Exception {
    public ServerErrorException(String msg, Throwable t) {
        super(msg, t);
    }
    @Override
    public String getOAuth2ErrorCode() {
        return "server_error";
    }
    @Override
    public int getHttpErrorCode() {
        return 500;
    }
}

private static class ForbiddenException extends OAuth2Exception {
    public ForbiddenException(String msg, Throwable t) {
        super(msg, t);
    }
    @Override
    public String getOAuth2ErrorCode() {
        return "access_denied";
    }
    @Override
    public int getHttpErrorCode() {
        return 403;
    }
}
}

```

前面是对认证服务进行配置的详解，接下来，我们看看对于资源服务的配置：

```

package com.damon.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceSer
ver;
import
org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerCon
figurerAdapter;

/**
 *
 * 资源服务器配置
 * @author Damon
 * @date 2020年1月13日 下午3:03:48
 *
 */
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {

```

```
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()

            .exceptionHandling()
            .authenticationEntryPoint(new AuthenticationEntryPointHandle())
            // .authenticationEntryPoint((request, response, authException) ->
        response.sendError(HttpServletRequest.SC_UNAUTHORIZED))
            .and()

            .requestMatchers().antMatchers("/api/**")
            .and()
            .authorizeRequests()
            .antMatchers("/api/**").authenticated()
            .and()
            .httpBasic();
    }
}
```

这里也配置了对于资源拦截的统一处理：

```
package com.damon.config;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.http.HttpStatus;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;

import com.alibaba.fastjson.JSON;
import com.damon.commons.Response;

/**
 *
 * 统一结果处理
 *
 * @author Damon
 * @date 2020年1月16日 上午11:11:44
 *
 */
public class AuthenticationEntryPointHandle implements AuthenticationEntryPoint {
    /**
     *
     * @author Damon
     * @date 2020年1月16日
     *
     */
    @Override
```

```

public void commence(HttpServletRequest request, HttpServletResponse response,
        AuthenticationException authException) throws IOException, ServletException {

    //response.setStatus(HttpServletRequest.SC_FORBIDDEN);
    //response.setStatus(HttpStatus.OK.value());

    //response.setHeader("Access-Control-Allow-Origin", "*"); //gateway已加，无需再加
    //response.setHeader("Access-Control-Allow-Headers", "token");
    //解决低危漏洞点击劫持 X-Frame-Options Header未配置
    response.setHeader("X-Frame-Options", "SAMEORIGIN");
    response.setCharacterEncoding("UTF-8");
    response.setContentType("application/json; charset=utf-8");

    response.getWriter()
        .write(JSON.toJSONString(Response.ok(response.getStatus(), -2,
        authException.getMessage(), null)));
    /*response.getWriter()
        .write(JSON.toJSONString(Response.ok(200, -2, "Internal Server Error",
        authException.getMessage())));*/
    }
}

```

以上就是认证中心的核心代码了，这里有一些需要注意的地方：有2个拦截器，鉴权服务配置、资源权限配置，鉴权时可以通过几种模式来进行。授权码模式交互比较多，密码模式比较常见应用。

接下来，我们新建一个电商系统的订单服务，作为鉴权的客户端，我们来看看代码。

订单服务客户端

订单系统我们考虑到系统的高可用，以及系统的TPS，我们采用负载均衡手段，加上一些中间件来处理，使得服务更加具有代表性。

订单系统的环境这就不介绍了，这里主要还是看依赖K8s的组件：

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-actuator</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-actuator-autoconfigure</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-kubernetes-config</artifactId>
</dependency>

<!-- springCloud-k8s-discovery -->

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-commons</artifactId>

```

```
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-kubernetes-core</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-kubernetes-discovery</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-kubernetes-ribbon</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

上面的依赖是K8s对于Java的sdk-client，接下来配置一些环境使其生效：

```
spring:
  application:
    name: order-service
    #redis: #redis相关配置
    #password: 123456 #有密码时设置
  cloud:
    kubernetes:
      config:
        sources:
          - name: ${spring.application.name}
            namespace: system-server
      discovery:
        all-namespaces: true
      reload:
        #自动更新配置的开关设置为打开
        enabled: true
        #更新配置信息的模式: polling是主动拉取, event是事件通知
        mode: polling
        #主动拉取的间隔时间是500毫秒
        period: 500
    http:
      encoding:
        charset: UTF-8
        enabled: true
        force: true
  mvc:
```

```

throw-exception-if-no-handler-found: true
main:
  allow-bean-definition-overriding: true # 当遇到同样名称时，是否允许覆盖注册

logging:
  path: /data/${spring.application.name}/logs

cas-server-url: http://cas-server-service #http://localhost:2000#设置可以访问的地址

security:
  oauth2: #与cas-server对应的配置
    client:
      client-id: order-service
      client-secret: order-service-123
      user-authorization-uri: ${cas-server-url}/oauth/authorize #是授权码认证方式需要的
      access-token-uri: ${cas-server-url}/oauth/token #是密码模式需要用到的获取 token 的接
□
  resource:
    loadBalanced: true
    #jwt: #jwt存储token时开启
    #key-uri: ${cas-server-url}/oauth/token_key
    #key-value: test_jwt_sign_key
    id: order-service
    #指定用户信息地址
    user-info-uri: ${cas-server-url}/api/user #指定user info的URI, 原生地址后缀
    为/auth/user
    prefer-token-info: false
    #token-info-uri:
    authorization:
      check-token-access: ${cas-server-url}/oauth/check_token #当此web服务端接收到来自UI客
      户端的请求后, 需要拿着请求中的 token 到认证服务端做 token 验证, 就是请求的这个接口

```

最上面是配置了加载服务的环境变量，采用K8s的ConfigMap实现，不再使用复杂的各种大厂提供的插件。下面是订单服务接入鉴权时需要的鉴权配置。这里为了鉴权的健壮性，采用了分布式部署：

```
cas-server-url: http://cas-server-service
```

通过K8s的Service来进行及安全认证，实现服务的高可用。接下来是订单服务的各种模式的支持，这里主要是授权码模式、密码模式。同时，开启了订单服务的负载均衡策略：

```

package com.damon;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.SpringBootConfiguration;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.security.oauth2.client.EnableOAuth2Sso;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.netflix.hystrix.EnableHystrix;
import org.springframework.context.annotation.ComponentScan;

```

```

import org.springframework.context.annotation.Configuration;

import com.damon.config.EnvConfig;

/**
 * @author Damon
 * @date 2020年1月13日 下午3:23:06
 *
 */

@EnableOAuth2Sso
@Configuration//@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(basePackages = {"com.damon"})
@EnableConfigurationProperties(EnvConfig.class)
@EnableDiscoveryClient
@EnableCircuitBreaker//@EnableHystrix

//@RibbonClients针对多个服务源进行策略的指定，这里注意这种方式时，RibbonConfiguration类不能被
//包含在@ComponentScan的扫描包中
/*@RibbonClients(value = {
    @RibbonClient(name="cas-server-service", configuration =
RibbonConfiguration.class),
    @RibbonClient(name="admin-web-service", configuration =
RibbonConfiguration.class)
})*/
public class OrderApp {

    public static void main(String[] args) {
        SpringApplication.run(OrderApp.class, args);
    }
}

```

接下来，我们看下客户端的资源配置：

```

package com.damon.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceSer
ver;
import
org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerCon
figurerAdapter;

import javax.servlet.http.HttpServletResponse;

/**
 *
*
* @author Damon
* @date 2020年1月16日 下午6:28:35
*

```

```
/*
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()

            .exceptionHandling()
            .authenticationEntryPoint(new AuthenticationEntryPointHandle())
            // .authenticationEntryPoint((request, response, authException) ->
            response.sendError(HttpStatus.SC_UNAUTHORIZED))
            .and()

            .requestMatchers().antMatchers("/api/**")
            .and()
            .authorizeRequests()
            .antMatchers("/api/**").authenticated()
            .and()
            .httpBasic();
    }
}
```

这里加了一个统一结果处理类：

```
package com.damon.config;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;

import com.alibaba.fastjson.JSON;
import com.damon.commons.Response;

/**
 *
 * 统一结果处理
 *
 * @author Damon
 * @date 2020年1月16日 上午11:11:44
 *
 */

public class AuthenticationEntryPointHandle implements AuthenticationEntryPoint {
```

```

    */
    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
                         AuthenticationException authException) throws IOException, ServletException {

        //response.setStatus(HttpServletRequest.SC_FORBIDDEN);
        //response.setStatus(HttpStatus.OK.value());

        //response.setHeader("Access-Control-Allow-Origin", "*"); //gateway已加，无需再加
        //response.setHeader("Access-Control-Allow-Headers", "token");
        //解决低危漏洞点击劫持 X-Frame-Options Header未配置
        response.setHeader("X-Frame-Options", "SAMEORIGIN");
        response.setCharacterEncoding("UTF-8");
        response.setContentType("application/json; charset=utf-8");

        response.getWriter()
            .write(JSON.toJSONString(Response.ok(response.getStatus(), -2,
                authException.getMessage(), null)));
        /*response.getWriter()
            .write(JSON.toJSONString(Response.ok(200, -2, "Internal Server Error",
                authException.getMessage())));*/
    }

    /*@Override
    public void commence(HttpServletRequest request,
                          HttpServletResponse response,
                          AuthenticationException authException) throws IOException, ServletException {

        response.setCharacterEncoding("UTF-8");
        response.setContentType("application/json; charset=utf-8");
        response.setStatus(HttpServletRequest.SC_FORBIDDEN);
        response.getWriter()
            .write(JSON.toJSONString(Response.ok(200, -2, "Internal Server Error",
                authException.getMessage())));
    }*/
}

```

前面说增加了负载均衡策略，此处我们引入的是Ribbon作为负载均衡器：

```

server:
  port: 2003
  undertow:
    accesslog:
      enabled: false
      pattern: combined
  servlet:
    session:
      timeout: PT120M
      cookie:
        name: ORDER-SERVICE-SESSIONID #防止Cookie冲突，冲突会导致登录验证不通过

client:
  http:
    request:

```

```

connectTimeout: 8000
readTimeout: 30000

backend:
  ribbon:
    eureka:
      enabled: false
    client:
      enabled: true
    ServerListRefreshInterval: 5000

ribbon:
  ConnectTimeout: 2000
  ReadTimeout: 3000
  eager-load:
    enabled: true
    clients: cas-server-service,admin-web-service
  MaxAutoRetries: 1 #对第一次请求的服务的重试次数
  MaxAutoRetriesNextServer: 1 #要重试的下一个服务的最大数量 (不包括第一个服务)
  #ListOfServers: localhost:5556,localhost:5557
  #ServerListRefreshInterval: 2000
  OkToRetryOnAllOperations: true
  NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule
  #com.damon.config.RibbonConfiguration #自定义策略类

hystrix:
  command:
    BackendCall: #default or commandKey的值
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 5000
  threadpool:
    BackendCallThread:
      coreSize: 5

```

同时，这里采用的是默认的轮训策略，当然可以自定义一种策略：

```

package com.damon.config;

import org.springframework.context.annotation.Bean;

import com.netflix.client.config.IClientConfig;
import com.netflix.loadbalancer.IPing;
import com.netflix.loadbalancer.IRule;
import com.netflix.loadbalancer.PingUrl;
import com.netflix.loadbalancer.RoundRobinRule;

/**
 * @author Damon
 * @date 2019年10月30日 下午5:03:54
 */

```

```
public class RibbonConfiguration {  
  
    /**  
     * 检查服务是否可用的实例，  
     * 此地址返回的响应的返回码如果是200表示服务可用  
     * @param config  
     * @return  
     */  
    @Bean  
    public IPing ribbonPing(IClientConfig config){  
        return new PingUrl();  
    }  
  
    /**  
     * 轮询规则  
     * @param config  
     * @return  
     */  
    @Bean  
    public IRule ribbonRule(IClientConfig config){  
        //return new AvailabilityFilteringRule();  
        return new RoundRobinRule(); //轮询  
        //return new RetryRule(); //重试  
        //return new RandomRule(); //这里配置策略，和配置文件对应  
        //return new WeightedResponseTimeRule(); //这里配置策略，和配置文件对应  
        //return new BestAvailableRule(); //选择一个最小的并发请求的server  
        //return new MyProbabilityRandomRule(); //自定义  
    }  
}
```

在上面的策略函数**ribbonRule**中，实现自定义策略：

```
package com.damon.config;  
  
import java.util.List;  
import java.util.Random;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
import com.netflix.loadbalancer.BaseLoadBalancer;  
import com.netflix.loadbalancer.ILoadBalancer;  
import com.netflix.loadbalancer.IRule;  
import com.netflix.loadbalancer.Server;  
  
/**  
 * 实现自定义负载均衡策略  
 * @author Damon  
 * @date 2019年10月30日 上午9:08:49  
 */  
  
public class MyProbabilityRandomRule implements IRule {
```

```

Logger log = LoggerFactory.getLogger(MyProbabilityRandomRule.class);

ILoadBalancer balancer = new BaseLoadBalancer();

@Override
public Server choose(Object key) {
    List<Server> allServers = balancer.getAllServers();
    Random random = new Random();
    final int number = random.nextInt(10);
    if (number < 7) {
        return findServer(allServers,8091);
    }
    return findServer(allServers,8092);
}

private Server findServer(List<Server> allServers, int port) {
    for (Server server : allServers) {
        if (server.getPort() == port) {
            return server;
        }
    }
    log.info("NULL port="+port);
    return null;
}

@Override
public void setLoadBalancer(ILoadBalancer lb) {
    this.balancer = lb;
}

@Override
public ILoadBalancer getLoadBalancer() {
    return this.balancer;
}

}

```

到此，关于客户端的配置全部解决完了，接下来简单写一个API：

```

@GetMapping("/getCurrentOrderUser")
@PreAuthorize("hasAuthority('admin')")
public Object getCurrentOrderUser(Authentication authentication) {
    logger.info("test password");
    return authentication;
}

//@PreAuthorize("hasAuthority('admin')")
@PreAuthorize("hasAuthority('admin1')")
@GetMapping("/auth/admin")
public Object adminAuth() {
    logger.info("aAaqsqwsw");
    return "Has admin auth!";
}

```

我们简单的实现了一个客户端的代码，以及鉴权认证。

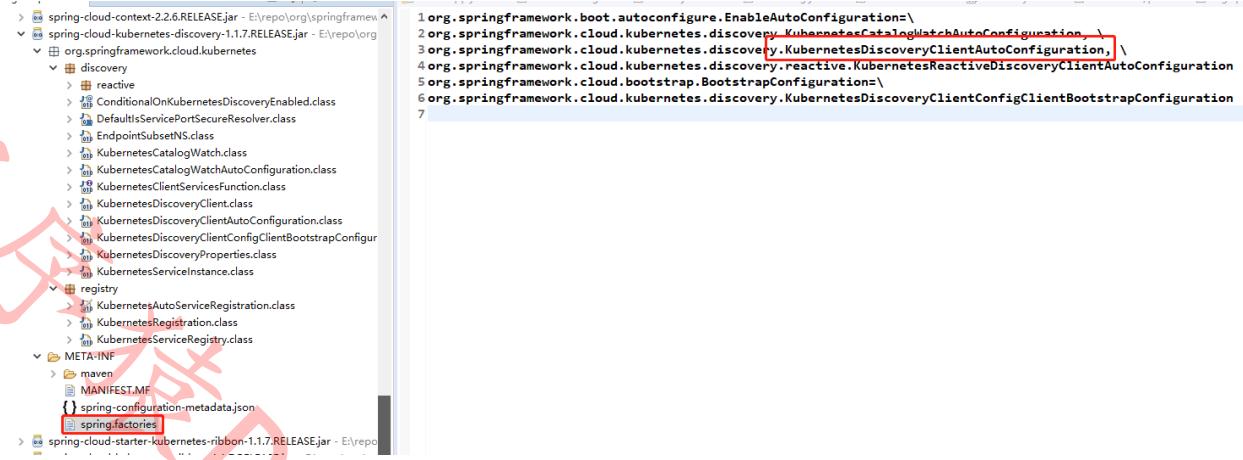
基于Java阐述K8s的服务发现

前面从K8s组件以及资源的角度，来分析了K8s的服务注册与发现的缘由。接下来，我们看看基于Java，K8s如何提供该项功能。

在pom.xml中，有对spring-cloud-kubernetes框架的依赖配置：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-kubernetes-discovery</artifactId>
</dependency>
```

我们看下这个依赖的源码：

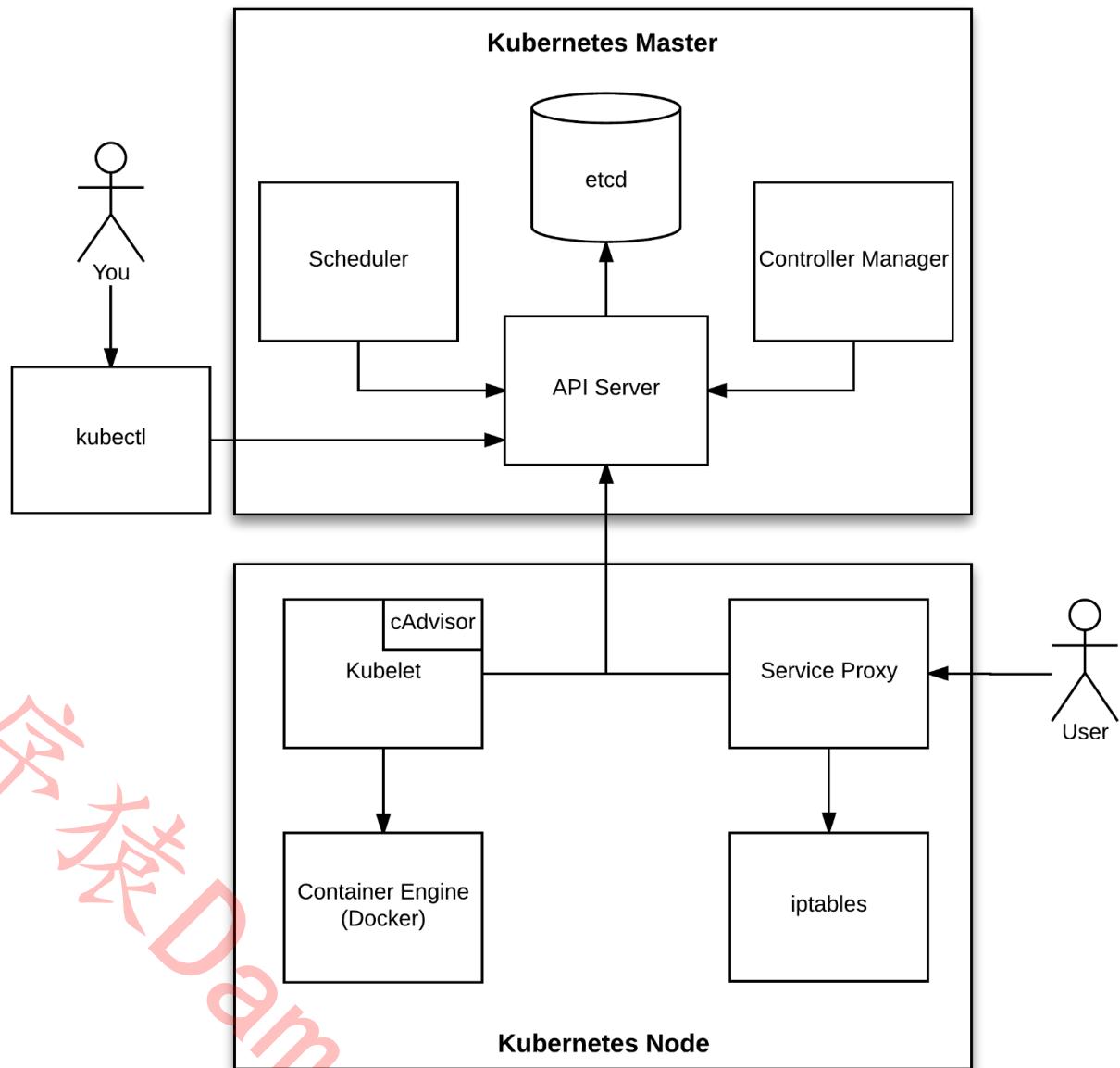


spring容器启动时，会找到classpath下的spring.factories文件，spring.factories文件中有两个类：KubernetesDiscoveryClientAutoConfiguration和KubernetesDiscoveryClientConfigClientBootstrapConfiguration都会被实例化；

再看KubernetesDiscoveryClientAutoConfiguration源码，注意**kubernetesDiscoveryClient**方法，这里面实例化了DiscoveryController所需的DiscoveryClient接口实现：

```
@Bean
@ConditionalOnMissingBean
@ConditionalOnProperty(name = "spring.cloud.kubernetes.discovery.enabled", matchIfMissing = true)
public KubernetesDiscoveryClient kubernetesDiscoveryClient(KubernetesClient client,
    KubernetesDiscoveryProperties properties,
    KubernetesClientServicesFunction kubernetesClientServicesFunction,
    DefaultIsServicePortSecureResolver isServicePortSecureResolver) {
    return new KubernetesDiscoveryClient(client, properties,
        kubernetesClientServicesFunction,
        isServicePortSecureResolver);
}
```

其实最终是向kubernetes的API Server发起http请求，获取service资源的数据列表；在K8s的API Server收到请求后，**由API Server从etcd中取得service的数据返回**。



可见 spring-cloud-kubernetes 的 DiscoveryClient 服务将 kubernetes 中的"service"资源与 SpringCloud 中的服务对应起来了，有了这个 DiscoveryClient，我们在 kubernetes 环境就不需要 eureka 来做注册发现了，而是直接使用 kubernetes 的服务机制，此时不得不感慨 SpringCloud 的对 DiscoveryClient 的设计是如此的精妙。

7.2 手写第一个Golang微服务

现在有不少的Golang框架：Beego、Gin等，今天为了简单起见，我们直接使用Beego。

环境

- GoLand 2020.3.2
- Golang 1.12+

第一个Golang的main函数：

```
package main
```

```

import (
    "github.com/astaxie/beego"
    "github.com/astaxie/beego/logs"
    "github.com/spf13/pflag"
    _ "pay-service/routers"
    "time"
)

func init() {
    logs.SetLogFuncCall(true)
    logs.SetLogFuncCallDepth(3)
    logs.SetLogger(logs.AdapterFile, `{"filename":"/data/pay-service/log/pay-
service.log","level":7,"daily":true,"maxdays":7,"color":true}`)
}

var logFlushFreq = pflag.Duration("log-flush-frequency", 5*time.Second, "Maximum number
of seconds between log flushes")

func main() {
    //先在先执行，后者不再执行

    /*http.HandleFunc("/health", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, fmt.Sprintf("%v", true))
    })
    http.ListenAndServe(":38080", nil)*/
}

beego.Run()

//cache health
//var configFile = flag.String("config", "conf/config-dev.yaml", "the config file of
pay service")
/*flag.Parse()

go wait.Until(glog.Flush, *logFlushFreq, wait.NeverStop)
defer glog.Flush()

config, err := config.ParseFromYaml(*configFile)
if config.ProcessNum > 0 {
    runtime.GOMAXPROCS(config.ProcessNum)
} else {
    runtime.GOMAXPROCS(runtime.NumCPU())
}
runtime.NumCPU()

cache := cache.New(&cache.Config{})
done := make(chan struct{})
cache.Run(done)
<-done*/
}

```

这里启动时，很简单，就是beego的线程启动，当然，如果你需要添加一些其他的，可以在**init**函数中添加逻辑。

接下来，我们新建一个routers目录，新建router文件来设置路由：

```

// @APIVersion V1.0.0
// @Title 支付服务 API
// @Description 提供支付服务相关的API集合.
package routers

import (
    "github.com/astaxie/beego"
    "pay-service/api"
    "common-core/common/check"
)

func init() {
    ns := beego.NewNamespace("/api/v1",
        beego.NSRouter("/healthz", &api.ProbeHealthAPI{}),
        beego.NSRouter("/pay/:orderId", &api.PayDetailAPI{}),
    )

    beego.InsertFilter("/api/v1/*", beego.BeforeRouter, check.CheckAuth)
    beego.AddNamespace(ns)
}

```

上面的路由包括两个接口，一个是服务自身的健康检测healthz，还有一个基于订单号查询支付服务信息。

```

package api

import (
    base_api "common-core/common/api"
    "strings"
)

type PayDetailAPI struct {
    base_api.BaseAPI
}

func (pd *PayDetailAPI) Get() {
    orderId := strings.TrimSpace(pd.GetString(":orderId"))
    pd.RespSucc("success: " + orderId)
}

```

这里由于API需要集成Beego的控制类，所以新建一个common目录，提供基础功能，包括api:

```

package api

import (
    "common-core/common/handler"
    "common-core/common/res"
    "encoding/json"
    "github.com/astaxie/beego"
    "github.com/astaxie/beego/logs"
)

```

```

        "io/ioutil"
        "net/http"
    )

type BaseAPI struct {
    beego.Controller
}

func (ba *BaseAPI) GetBody(body interface{}) error {
    bodyByte, err := ioutil.ReadAll(ba.Ctx.Request.Body)
    if err != nil {
        logs.Error(err.Error())
        return handler.NewBadReqError(handler.ComErrorCodeInvalidPara, err.Error())
    }

    if err := json.Unmarshal(bodyByte, body); err != nil {
        return handler.NewBadReqError(handler.ComErrorCodeInvalidPara, err.Error())
    }

    return nil
}

func (ba *BaseAPI) RespSucc(body interface{}) {
    ba.Resp(http.StatusOK, (&res.DefaultResponse{}).Succ(body))
}

func (ba *BaseAPI) RespError(err error) {
    if handler.IsBadRequestError(err) {
        ba.Resp(http.StatusBadRequest, (&res.DefaultResponse{}).Fail(err))
    } else {
        ba.Resp(http.StatusInternalServerError, (&res.DefaultResponse{}).Fail(err))
    }
}

func (ba *BaseAPI) RespSuccError(err error) {
    if handler.IsBadRequestError(err) {
        ba.Resp(http.StatusOK, (&res.DefaultResponse{}).Fail(err))
    } else {
        ba.Resp(http.StatusOK, (&res.DefaultResponse{}).Fail(err))
    }
}

func (ba *BaseAPI) Resp(httpStatus int, res res.Response) {
    ba.Data["json"] = res
    ba.Ctx.Output.SetStatus(httpStatus)
    ba.ServeJSON()
}

```

同时，提供了一些异常处理：

```

package handler

import (

```

```
"fmt"
)

const (
    //公共错误码 0-100
    ComErrorCodeSuccess      = 0
    ComErrorCodeNoVistAuth   = 1
    ComErrorCodeInvalidPara  = 2
    ComErrorCodeUnknown       = 3
)

const (
    badRequestType = "[bad request]"
    innerErrorType = "[inner error]"
    successType    = "[success]"
)

type ExceptionHandler struct {
    Code      int      `json:"code"`
    ErrorType string  `json:"-"`
    Msg       string  `json:"msg"`
}

func (ec ExceptionHandler) Error() string {
    return ec.Msg
}

func NewBadReqError(code int, format string, paras ...interface{}) error {
    return newError(code, badRequestType, format, paras...)
}

func NewInnerError(code int, format string, paras ...interface{}) error {
    return newError(code, innerErrorType, format, paras...)
}

func NewSuccess(format string, paras ...interface{}) error {
    return newError(ComErrorCodeSuccess, successType, format, paras...)
}

func NewUnkownError(format string, paras ...interface{}) error {
    return newError(ComErrorCodeUnknown, innerErrorType, format, paras...)
}

func IsInnerError(err error) bool {
    if errCustom, ok := err.(ExceptionHandler); ok {
        return errCustom.ErrorType == innerErrorType
    }
    return false
}

func IsBadRequestError(err error) bool {
    if errCustom, ok := err.(ExceptionHandler); ok {
        return errCustom.ErrorType == badRequestType
    }
    return false
}

func IsSuccess(err error) bool {
```

```

if err == nil {
    return true
}

if errCustom, ok := err.(*ExceptionHandler); ok {
    return errCustom.ErrorType == successType
}
return false
}

func NewError(code int, format string, paras ...interface{}) *ExceptionHandler {
    return &ExceptionHandler{Code: code, Msg: fmt.Sprintf(format, paras...)}
}

func newError(code int, errorType string, format string, paras ...interface{}) error {
    return &ExceptionHandler{Code: code, ErrorType: errorType, Msg: fmt.Sprintf(format,
paras...)}
}

```

返回响应体：

```

package res

import (
    // http client driver
    "github.com/astaxie/beego/logs"
    "common-core/common/handler"
)

type Response interface {
    Succ(body interface{}) Response
    Fail(err error) Response
}

type DefaultResponse struct {
    Status handler.ExceptionHandler `json:"status"`
    Data   interface{}              `json:"data"`
}

func (dr *DefaultResponse) Succ(body interface{}) Response {
    success, _ := handler.NewSuccess("success").(*handler.ExceptionHandler)
    dr.Status = *success
    dr.Data = body
    return dr
}

func (dr *DefaultResponse) Fail(err error) Response {
    errCustom, ok := err.(*handler.ExceptionHandler)
    if !ok {
        logs.Error("error type invalid, please use custom error")
        unkownError, _ := handler.NewUnkownError(err.Error()).(*handler.ExceptionHandler)
        dr.Status = *unkownError
    } else {
        dr.Status = *errCustom
    }
}

```

```
    return dr  
}
```

到此，一个简单的微服务就写完了，如果需要借助一些缓存，或插件来实现一些功能，可以自行添加。

启动微服务，可以运行**main**函数，输出：

```
2021/04/13 16:51:38.785 [I] [app.go:214] http server Running on http://:8080
```

通过在浏览器输入请求地址：

```
http://localhost:8080/api/v1/healthz
```

```
http://localhost:8080/api/v1/pay/232
```

如果增加鉴权认证，则会返回：

```
{"status": {"code": 1, "msg": "no token"}, "data": null}
```

7.3 部署微服务应用

部署服务

这里引入了部署框架，具体内容将在后续公开，现在先来看看如何部署cas-server：

安装教程

- git clone https://gitee.com/damon_one/microservice-k8s.git
- kubectl create namespace system-server
- 在microservice-k8s目录下
 1. sh install_requirement.sh
 2. cd build
 3. 修改对应的配置
- vi/deployment/quick-start/quick-start-AIO-example.yaml

```
default:  
  cluster_id: singlebox  
  gpu_version: 440.31  
  mysql_password: ssswsw  
  cluster_server:
```

```

default:
  ssh-username: damon #机器用户名
  ssh-password: wwwww #机器密码
  gpu:
    type: debug
    count: 2
    cpu: 24
    mem: 187
master:
- ip: 10.12.3.17
  hostname: damon
  username: damon
  password: wwwww
  gpu:
    # required if
    type: debug
    count: 2
# compute:
# - ip: <compute node ip>
#   hostname: <compute node hostname>
#   gpu:
#     # the type and count must be configured at same time
#     type: <gpu type that is different from default>
#     count: <gpu type that is different from default>

registry_info:
# the registry address of docker,format [ip:port] eg. 10.10.8.100:5000
domain: 10.10.8.100:5000
# the registry's username
username: admin
# the registry's password
password: wdwddwddw
# the namespace of kubernetes
k8s_namespace: singlebox_google_containers
# the namespace of services
leinao_namespace: hub

```

4. sudo python/controller.py config generate -i /home/damon/microservice-k8s/deployment/quick-start/quick-start-AIO-example.yaml -o /home/damon/deployment/output_config

- 按照config部署k8s(可不操作， 默认已经部署k8s)

1. sudo python/controller.py cluster k8s-clean -p /home/damon/deployment/output_config

2. sudo python/controller.py cluster k8s-bootup -p /home/damon/deployment/output_config

- 把config配置push到远程,k8s重新部署时需要执行:

1. sudo python/controller.py config push -p /home/damon/deployment/output_config one

- 编译镜像:

1. sudo python pai_build.py build -c /home/damon/deployment/output_config -s cas-server
2. sudo python pai_build.py push -c /home/damon/deployment/output_config -i cas-server

- 部署服务: one

1. sudo python ..controller.py service delete -n cas-server or mysql
2. sudo python ..controller.py service start -n cas-server

测试

部署后，我们可以执行K8s命令查看服务pod:

```
tom@PK001:~/damon$ kubectl get po -n system-server
NAME                               READY   STATUS    RESTARTS   AGE
cas-server-deployment-6bdw56dwde-5pdwf   1/1     Running   0          3d
gateway-service-deployment-6b7856bc99-5pk56   1/1     Running   0          5d3h
order-service-76f57dbd7c-rpfsls           1/1     Running   0          10d
pay-service-68bcc4db8-v42gm              1/1     Running   0          10d
```

不同 NS 之间服务的互通:

kube-system ---> system-server

```
tom@PK001:~/damon$ kubectl exec -it order-service-76f57dbd7c-rpfsls sh -n kube-system
#
#
# curl http://pay-service-svc.system-server.svc.cluster.local:8090/api/v1/healthz
{
  "status": {
    "code": 0,
    "msg": "success"
  },
  "data": "success"
}#
#
#
```

system-server ---> kube-system

```
tom@PK001:~/damon$ kubectl exec -it pay-service-768bcc4db8-v42gm sh -n system-server
#
#
# curl http://order-service-svc.kube-system.svc.cluster.local:9300/api/v1/orders
{"status":{"code":1,"msg":"no token"},"data":null}#
#
```

```
#  
#
```

当出现鉴权认证时，需要获取鉴权令牌Token：

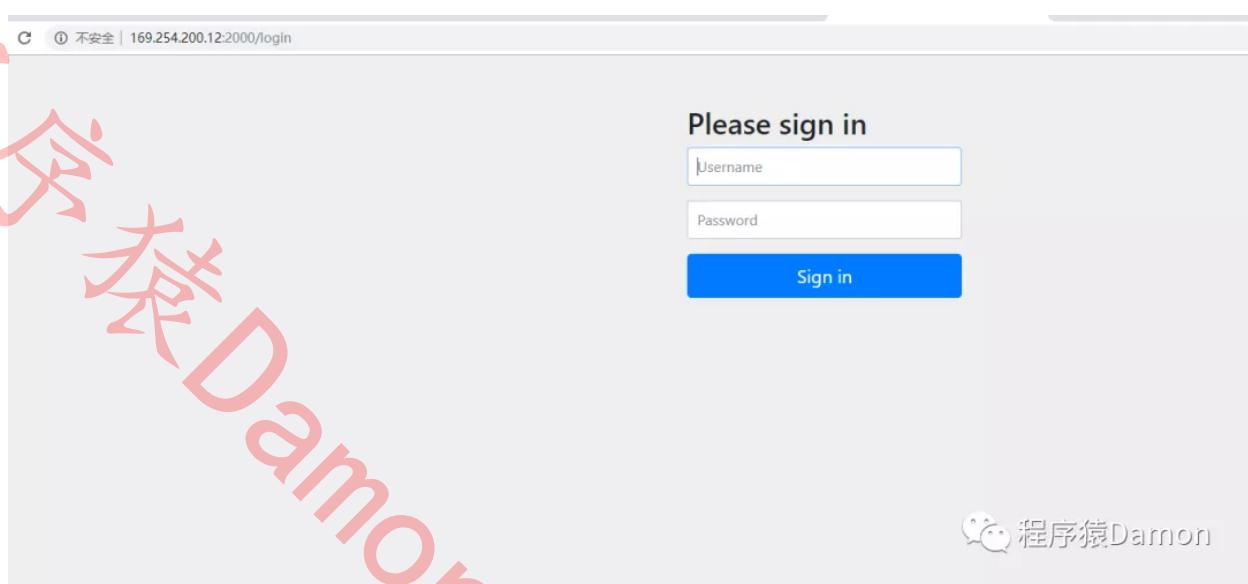
授权码模式

1. 获取授权码

页面打开url：

```
localhost:2000/oauth/authorize?response_type=code&client_id=order-service&redirect_uri=http://order-service/Login&scope=all
```

此时，页面被拦截到登录页：



在点击 Approve、Authorize 后，输入用户名、密码，跳转到上面的重定向地址，并带有 code 属性参数：

```
http://order-service/Login?code=GW00J7
```

2. 根据 code 获取 access_token

```
curl -i -X POST -d "grant_type=authorization_code&code=GW00J7&client_id=order-service&client_secret=order-service-123&redirect_uri=http://order-service/login"  
http://localhost:2000/oauth/token
```

返回信息：

```
{"access_token": "a2af3f0b-27da-41b8-90c0-3bd2a1ed0421", "token_type": "bearer", "refresh_token": "91c22287-aa24-4305-95cf-
```

```
38f7903865f3","expires_in":3283,"scope":"all"}
```

3. 拿到 token 获取用户信息

- 头部携带

```
curl -i -H "Accept: application/json" -H "Authorization:bearer a2af3f0b-27da-41b8-90c0-3bd2a1ed0421" -X GET http://localhost:2003/api/order/getCurrentOrderUser
```

- 直接 get 方式传 token

```
http://localhost:2003/api/order/getCurrentOrderUser?access_token=a2af3f0b-27da-41b8-90c0-3bd2a1ed0421
```

4. 刷新token

通过上面的 "refresh_token" 来刷新获取新 token:

```
curl -i -X POST -d "grant_type=refresh_token&refresh_token=91c22287-aa24-4305-95cf-38f7903865f3&client_id=provider-service&client_secret=provider-service-123" http://localhost:2000/oauth/token
```

5. 退出

```
curl -i -H "Accept: application/json" -H "Authorization:bearer fafcc3c8-28f5-4ce6-87df-e7f929fa6a34" -X DELETE http://localhost:2000/api/logout
```

密码模式

密码模式由于都是通过直接调用服务来获取token，所以，咱们可以直接在容器内运行，看看效果：

```
curl -i -X POST -d "username=admin&password=123456&grant_type=password&client_id=order-service&client_secret=order-service-123" http://cas-server-service.system-server.svc.cluster.local:2000/oauth/token
```

认证成功后，会返回如下结果：

```
{"access_token":"d2066f68-665b-4038-9dbe-5dd1035e75a0","token_type":"bearer","refresh_token":"44009836-731c-4e6a-9cc3-274ce3af8c6b","expires_in":3599,"scope":"all"}
```

接下来，我们通过 token 来访问接口：

```
curl -i -H "Accept: application/json" -H "Authorization:bearer d2066f68-665b-4038-9dbe-5dd1035e75a0" -X GET http://order-service-service.system-server.svc.cluster.local:2003/api/order/auth/admin
```

成功会返回结果:

```
Has admin auth!
```

token 如果失效，会返回:

```
{"error": "invalid_token", "error_description": "d2066f68-665b-4038-9dbe-5dd1035e75a01"}
```

这里主要通过Service来进行服务访问，实现基于K8s的负载均衡：

```
tom@PK001:~$ kubectl get svc -n system-server
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)   AGE
gateway-service-service  ClusterIP  20.16.1.5    <none>       5556/TCP  74d
pay-service-svc     ClusterIP  20.16.63.79  <none>       8090/TCP  10d
admin-web-service   ClusterIP  10.16.129.24  <none>       2001/TCP  84d
cas-server-service  ClusterIP  10.16.230.167  <none>       2000/TCP  67d
cloud-admin-service-service ClusterIP  10.16.25.178  <none>       1001/TCP  190d
```

结束语

云原生技术与微服务架构的天衣无缝

云原生的微服务架构是云原生技术和微服务架构的完美结合。微服务作为一种架构风格，所解决的问题是交纵复杂的软件系统的架构与设计；云原生技术乃一种实现方式，所解决的问题是软件系统的运行、维护和治理。微服务架构可以选择不同的实现方式，如 Java 中的 Dubbo、Spring Cloud、Spring Cloud Alibaba，Golang 中的 Beego，Python 中的 Flask 等。但这些不同语言的服务之间的访问与运行可能存在一定得困难性与复杂性。但，云原生和微服务架构的结合，使得它们相得益彰。这其中的原因在于：云原生技术可以有效地弥补微服务架构所带来的实现上的复杂度；微服务架构难以落地的一个重要原因是它过于复杂，对开发团队的组织管理、技术水平和运维能力都提出了极高的要求。因此，一直以来只有少数技术实力雄厚的大企业会采用微服务架构。随着云原生技术的流行，在弥补了微服务架构的这一个短板之后，极大地降低了微服务架构实现的复杂度，使得广大的中小企业有能力在实践中应用微服务架构。云原生技术促进了微服务架构的推广，也是微服务架构落地的最佳搭配。

云原生时代的微服务的未来

云原生的第一个发展趋势：标准化和规范化，该技术的基础是容器化和容器编排技术，最经常会用到的技术是 Kubernetes 和 Docker 等。随着云原生技术的发展，云原生技术的标准化和规范化工作正在不断推进，其目的是促进技术的发展和避免供应商锁定的问题，这对于整个云原生技术的生态系统是至关重要的。

云原生的第二个发展趋势：平台化，以服务网格技术为代表，这一趋势的出发点是增强云平台的能力，从而降低运维的复杂度。流量控制、身份认证和访问控制、性能指标数据收集、分布式服务追踪和集中式日志管理等功能，都可以由底层平台来提供，这就极大地降低了中小企业在运行和维护云原生应用时的复杂度，服务网格以 Istio 和 Linkerd 为开源代表。

云原生的第三个发展趋势：应用管理技术的进步，如在 Kubernetes 平台上部署和更新应用一直以来都比较复杂，传统的基于资源声明 YAML 文件的做法，已经逐步被 Helm 所替代。操作员模式在 Helm 的基础上更进一步，以更高效、自动化和可扩展的方式对应用部署进行管理。

开源项目

实践项目代码开源：https://gitee.com/damon_one/microservice-k8s

欢迎大家star、fork，欢迎联系我，一起学习。

程序员 Damon