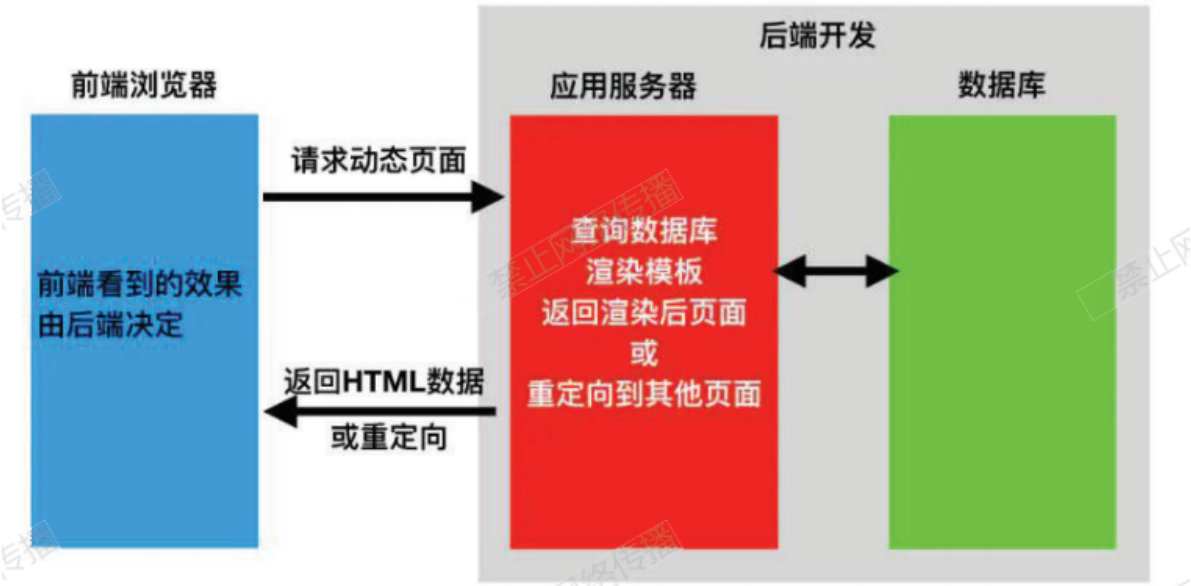
服务端开发复习

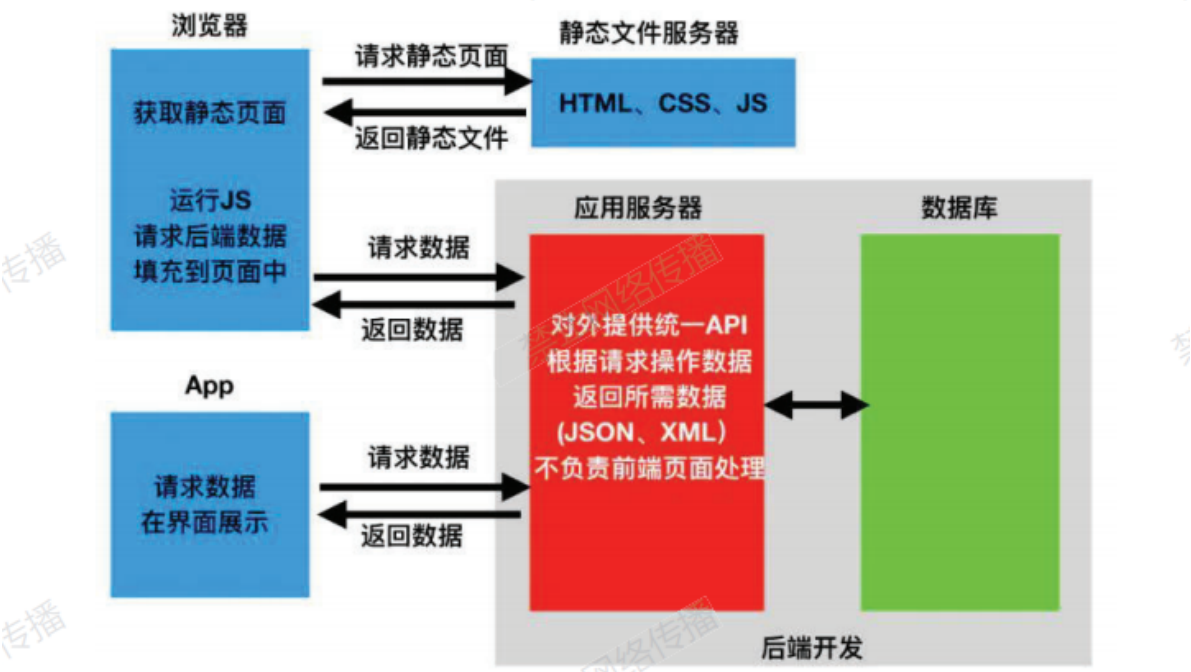
> 加粗的为考试重点

01-创建一个SpringBoot应用程序

前后端不分离的开发模式



前后端分离的开发模式



微服务架构

微服务架构是将应用拆成小业务单元开发和部署，使用轻量级协议通信，通过协同工作实现应用逻辑的架构模式。

特性：

1. 小，且职责单一
2. 独立的进程
3. 轻量级通信机制
4. 独立部署

开发框架

开发框架是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法，另一种定义认为框架是被应用开发者定制的应用骨架。

前端常用：Vue，React

Java后端常用：Spring

Spring框架的特点

- 轻量级
- 非侵入性
- 容器
- 控制反转 Inversion of Control (IoC)
- 依赖注入 Dependency Injected (DI)
- 面向切面编程 Aspect-Oriented Programming (AOP)
- 持久层（JDBC，事务，ORM）
- Web框架（SpringMVC）
- 其他企业服务的封装

Spring的衍生

- Spring：核心，基础框架
- Spring Boot：简化基于Spring的开发，自动配置
- Spring Cloud：基于云的，分布式系统开发，相关技术：容器，微服务，DevOps

Spring Boot开发框架

特征：

1. 可以创建独立的Spring应用程序，并且基于maven或gradle插件，可以创建可执行的JARs和WARs
2. 内嵌Tomcat或Jetty等Servlet容器
3. 提供自动配置的starter，简化maven配置
4. 尽可能自动配置Spring容器
5. 提供准备好的特性，如指标、健康检查和外部化配置
6. 绝对没有代码生成，不需要XML

开发阶段工具：Spring Boot Devtools 热部署

- 代码更改后会自动重启
- 当面向浏览器的资源（如模板、JS、CSS）发生变化时，会自动刷新浏览器
    - 需要安装VSCode插件：LiveReload

    - 需要安装浏览器扩展：LiveReload
- 自动禁用模板缓存
- 如果使用了H2数据库，则自带了H2控制台
    - 访问 `http://localhost:8080/h2-console`
- **注意是在开发阶段使用，是一个runtime的依赖**
- **不应该在生产环境中用**

Git工具的使用

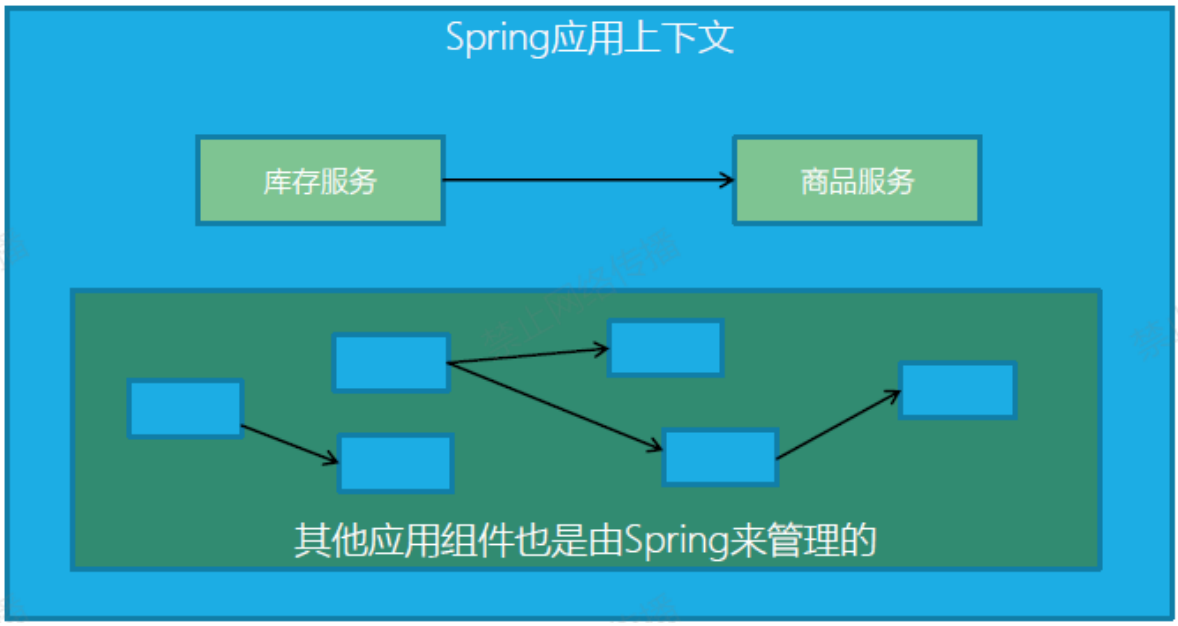
- **配置文件，脚本等也应该纳入仓库进行版本管理**

02-依赖注入

Spring核心技术

- **DI：保留抽象接口，让组件依赖于抽象接口，当组件要与其他实际的对象发生依赖关系时，由抽象接口来注入依赖的实际对象。**
- **AOP：通过预编译方式和运行期间动态代理实现程序功能的统一维护的一种技术。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发效率。**
- **IoC：控制反转是软件工程中的一个原则，它将对程序的对象或部分的控制转移到容器或框架中。我们最常在面向对象编程的上下文中使用它。**

Spring的核心是一个容器



Spring配置方案

1、自动化配置

```java
@Component
public class CDPlayer implements MediaPlayer {
    private CompactDisc cd;

    @Autowired
    public CDPlayer(CompactDisc cd) {
        this.cd = cd;
    }

    public void play() {
        cd.play();
    }
```

2、 JavaConfig

```java
@Configuration
public class CDPlayerConfig {

    @Bean
    public CompactDisc compactDisc() {
        return new SgtPeppers();
    }

    @Bean
    public CDPlayer cdPlayer(CompactDisc cd) {
        return new CDPlayer(cd);
    }
```

3、XML配置

```xml
<bean id="compactDisc" class="soundsystem.SgtPeppers" />

<bean id="cdPlayer" class="soundsystem.CDPlayer">
  <constructor-arg ref="compactDisc" />
</bean>
```

院

组件扫描

- @Configuration
- @ComponentScan

自动装配

@Autowired

- **用在构造器**
- **用在setter方法**
- **用在属性上（私有属性也可以）**
- required=false（加上这个属性，如果没有找到对应的bean不会报错）

JavaConfig

- @Configuration
- @Bean(name="...")

XML装配

- 不能类型检查
- 构造器注入
- 属性注入

混合配置

- JavaConfig中的导入
  - @Import(配置类.class)
  - @ImportResource(xml文件)
- XML中的导入
  - `<import resource="xml文件"/>`
  - `<bean class="配置类"/>`

@Profile

- @Profile('dev') 开发环境
- @Profile('prod') 生产环境
- @ActiveProfiles("dev") 激活

@Conditional

- @Conditional(**.class)
- 仅当对应的类被创建了才实例化当前类

自动装配的歧义性

如果一个类有多个实例化的bean，用@Qualifier("...")进行修饰。

Bean的作用域

@Scope可以与Component和@Bean一起使用，指定作用域。

- **Singleton 单例：在整个应用中只创建一个实例**
- **Prototype 原型：每次注入或通过应用上下文获取的时候，都会创建一个bean**
- **Session 会话：在Web应用中，在每个会话创建一个bean实例**
- **Request 请求：在Web应用中，为每个请求创建一个bean实例**

03-面向切面编程

软件编程方法的发展

- 面向过程编程 POP
- 面向对象编程 OOP
- 面向切面编程 AOP
- 函数式编程 FP
- 反应式编程 Rx

横切关注点

- 日志
- 安全
- 事务
- 缓存
- 继承（可选）
- 委托（可选）

AOP术语

- **通知 Advice：切面做什么以及何时做**
- **切点 PointCut：何处**
- **切面 Aspect：Advice和Pointcut的结合**
- **连接点 Join Point：方法、字段修改、构造方法**
- **引入 Introduction：引入新的行为和状态**
- **织入 Weaving：切面应用到目标对象的过程**

通知（Advice）类型

- @Before
- @After
- @AfterReturning
- @AfterThrowing
- @Around

织入时机

- 编译期，需要特殊的编译器
- 类加载期，需要类加载器的处理
- 运行期：Spring所采纳的方式，使用代理对象，只支持方法级别的连接点

Spring AOP

- @AspectJ 注解驱动的切面
- @EnableAspectJAutoProxy 开启自动代理

定义切面

@Aspect

- 例子

```
@Pointcut(
        "execution(* soundsystem.CompactDisc.playTrack( int )) " +
            "&& args(trackNumber)")  //获取参数
&& within(soundsystem.*)    //限定包路径
&& bean(sgtPeppers)   //限定bean名称，或者：  && !bean(sgtPeppers)
```

- 另一个例子

```
@Around("@annotation(innerAuth)")   //限定注解
 public Object innerAround(ProceedingJoinPoint point, InnerAuth innerAuth) { ... }

@InnerAuth
public R<Boolean> register(@RequestBody SysUser sysUser) { ... }
```

注意：@Aspect注解不包含@Component注解，Spring不会扫描到这个类并实例化

引入接口

- @DeclareParents
- 可以增加功能

哪些注解包含@Component注解的功能?

1. **@Controller**
2. **@Service**
3. **@Repository**

04-Web开发框架

lombok

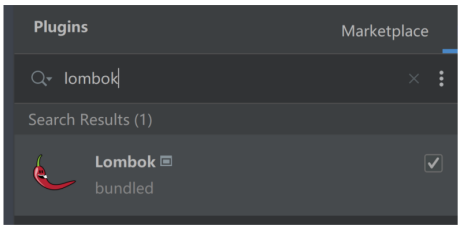
- 依赖

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

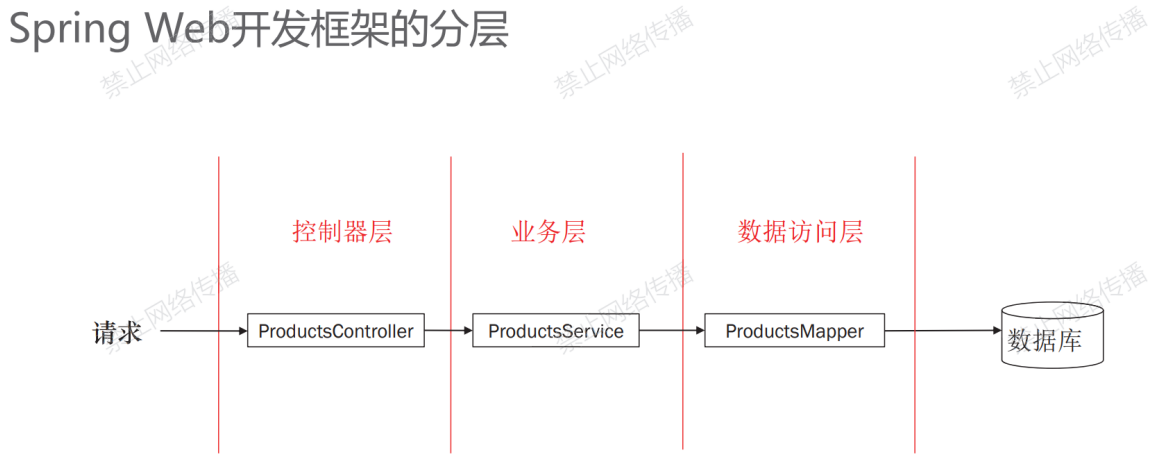
- 编译期后就不需要了，要排除

```
<groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
      <excludes>
        <exclude>
          <groupId>org.projectlombok</groupId>
          <artifactId>lombok</artifactId>
```

另外需要安装IntelliJ IDEA插件



Spring Web开发框架的分层



Spring MVC的请求映射注解

- **@RequestMapping**
  - **注意@RequestMapping既可以加在方法上，也可以加在类上**
- **@GetMapping**
- **@PostMapping**
- **@PutMapping**
- **@DeleteMapping**
- **@PatchMapping**

视图依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

校验表单输入

- JavaBean Validation API
- spring-boot-starter-validation
- 领域类上添加校验规则
- 控制器中声明校验：@Valid

视图控制器

简单的从URL到视图：

```
registry.addViewController("/").setViewName("home");
```

05-Spring Data JDBC, JPA

使用JDBC的依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Spring Data项目

- Spring Data JDBC
- Spring Data JPA
- Spring Data MongoDB
- Spring Data Neo4j
- Spring Data Redis
- Spring Data Cassndra

用spring boot编译出支持不同版本的jdk

```
<properties>
    <java.version>11</java.version>
</properties>
```

如果当前jdk是17，则编译出的版本支持1.8，11，17。

定义持久化接口

```
import org.springframework.data.repository.CrudRepository;
public interface IngredientRepository
    extends CrudRepository<Ingredient, String>
```

领域类注解

- @Table
- @id
- @Coloum

提高程序健壮性

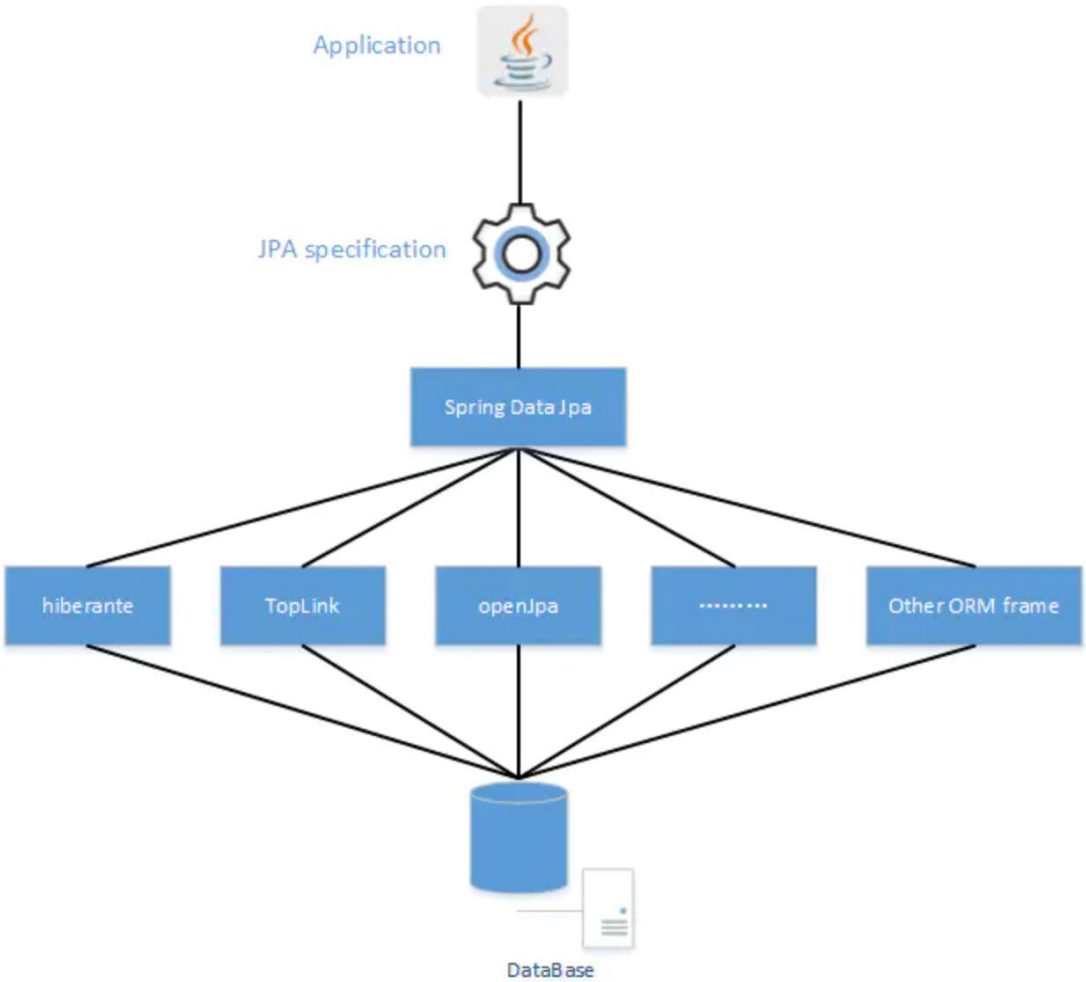
- JVM assert
- 开启debug模式
- 使用spring boot的Assert类

Spring Data JPA

- JPA：Java Persistence API
- JPA的宗旨是为POJO提供持久化标准规范
- 依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Jpa,Hibernate,Spring Data三者关系



@Entity

- 加在实体类上，会被JPA识别为实体

自定义查询方法

- 定义查询方法，无需实现
  - 使用领域特定语言DSL，spring data的命名约定
  - 查询动词 + 主题 + 断言
  - 查询动词：get、read、find、count
  - 例子：

```
List<TacoOrder> findByDeliveryZip( String deliveryZip );
```

06-Spring Security

Spring Security依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

两种配置

- 纯Java配置类

```
@Configuration
public class SecurityConfig{}
```

- 继承自WebSecurityConfigurerAdapter

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {}
```

用户信息存储

三种方式

1. 内存用户存储
2. JDBC用户存储
3. LDAP用户存储

保护Web请求

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
```

权限分类

- Authority，权限
- Role，角色，---> 权限，加上前缀ROLE_

创建自定义登录页

- 当需要认证时转向登录页：`.loginPage("/login")`
- 视图控制器，定义login请求对应的视图：`registry.addViewController("/login")`
- 登录的post请求由Spring Security自动处理，名称默认为username和password，可重新配置

实现方法级别的安全

- 在配置类上加注解@Configuration，@EnableGlobalMethodSecurity
- 在方法上加注解@PreAuthorize("hasRole('ADMIN')")

获取当前登录的用户

1. DesignTacoController

   参数：Principal principal

   ```
   String username = principal.getName();
   ```

2. OrderController

   ```
   @AuthenticationPrincipal User user
   ```

3. 安全上下文获取

   ```
   Authentication authentication =
   SecurityContextHolder.getContext().getAuthentication(); User user = (User)
   authentication.getPrincipal();
   ```

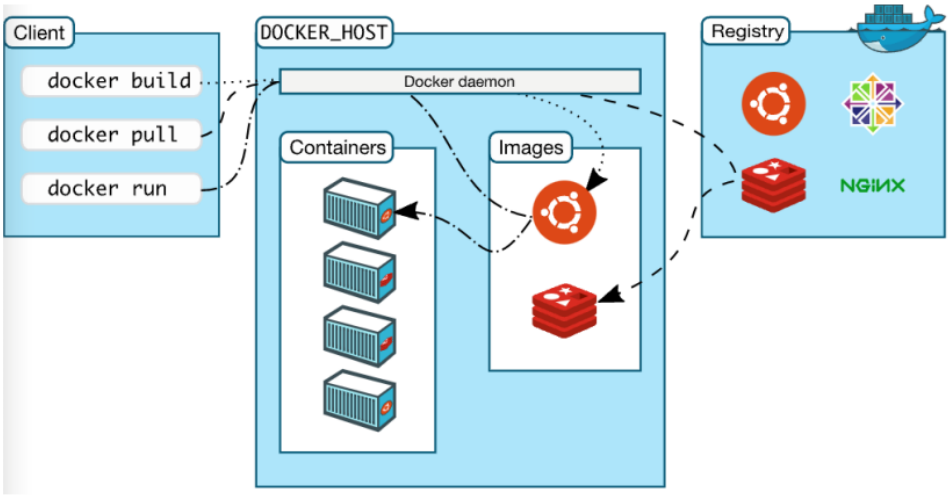
07-Docker使用

容器

- 容器是另外一种轻量级的虚拟化，容器是共用主机内核，利用内核的虚拟化技术隔离出一个独立的运行环境，拥有独立的一个文件系统，网络空间，进程空间视图等。
- 容器是在Linux内核实现的轻量级资源隔离机制
- 虚拟机是操作系统级别的资源隔离，本质上是进程级的资源隔离

Docker的三部分

Docker的三部分



Docker基本命令

- docker
- docker container --help
- docker --version
- docker version
- docker info
- docker image ls
- docker pull
- docker search xxx
- docker stop，start

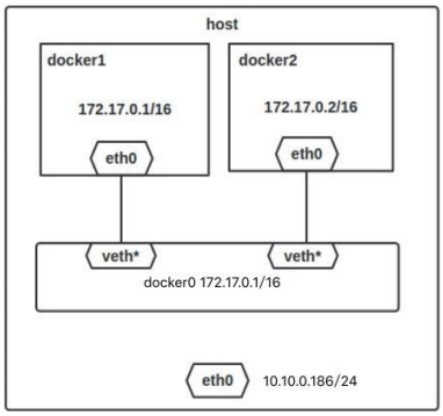
docker run命令

- **-p 端口号映射，格式为：主机端口:容器端口**
- **--rm 退出时自动删除容器**
- **-d 后台运行，并返回容器ID**
- **-e 设置环境变量**

- docker run hello-world
- -d: 后台运行容器，并返回容器ID
- -i: 以交互模式运行容器，通常与 -t 同时使用
- -t: 为容器重新分配一个伪输入终端，通常与 -i 同时使用
- -p: 指定（发布）端口映射，格式为：主机(宿主)端口:容器端口
- -P: 随机端口映射，容器内部端口随机映射到主机的高端口
- --name="nginx-lb": 为容器指定一个名称
- -e username="ritchie": 设置环境变量
- --env-file=c:/temp1/t1.txt: 从指定文件读入环境变量
- --expose=2000-2002: 开放（暴露）一个端口或一组端口；
- --link my-mysql:taozs : 添加链接到另一个容器
- -v c:/temp1:/data: 绑定一个卷(volume)
- --rm 退出时自动删除容器

容器网络

- none网络，--net=none
- host网络，--net=host
- bridge网络，--net=bridge ， docker0 的 linux bridge
- container模式，--net=container:NAME_or_ID



- **容器运行后，可以cat /etc/hosts 查看容器ip地址**

docker管理命令

- **docker volume**
- **docker network**
- **docker container**
- **docker image**
- **docker stop 停止正在运行的容器**
- **docker restart 重启容器**

08-容器镜像构建与编排

Dockfile文件指令（会考）

> **注意ADD和COPY的区别：ADD可以自动解压，COPY只是拷贝**
>
> **注意CMD和ENTRYPOINT的区别：不可以互相替代**

- FROM：指定基础镜像，必须为第一个命令
- **RUN：构建镜像时执行的命令**
- **ADD：将本地文件添加到容器中，tar类型会自动解压**
- **COPY：功能类似ADD，但是不会自动解压文件**
- **CMD：构建容器后调用，也就是在容器启动时才进行调用**
- **ENTRYPOINT：配置容器，使其可执行化。配合CMD可省去"application"，只使用参数，用于docker run时根据不同参数执行不同功能**
- LABEL：用于为镜像添加元数据
- ENV：设置环境变量
- EXPOSE：指定与外界交互的端口，容器内的端口号，docker run时加-P则会映射一个随机号
- VOLUME：用于指定持久化目录，docker run时如果没有指定挂载目录，会创建一个volume
- WORKDIR：工作目录，类似于cd命令

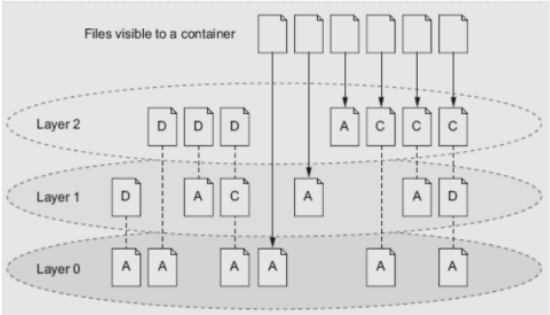
Docker build

- docker build [options] PATH | URL | -
- **如何编写最佳的Dockerfile**
  - **.dockerignore文件（排除不需要的文件）**

- **容器只运行单个应用（因为容器本身就是轻量级的）**
- **将多个RUN指令合并为一个（减少镜像的层级）**
- **基础镜像的标签不要用latest**
- **每个RUN指令后删除多余文件**
- **选择合适的基础镜像**
- **设置WORKDIR和CMD**

镜像分层

- 写时复制(COW, Copy-On- Write)
- docker history <image name> 查看镜像的层



服务编排工具，docker-compose

- Compose项目是Docker官方的开源项目，负责实现对Docker容器集群的快速编排
- 一个单独的**docker-compose.yml**模板文件来定义一组相关联的应用容器为一个项目(project)
- Compose的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷性生命周期管理。
- Compose中两个重要的概念
  - **服务(service)：一个应用的容器（可能会有多个容器），实际上可以包含若干运行相同镜像的容器实例**
  - **项目(project)：由一组关联的容器组成的一个完整业务单元，在docker-compose.yml文件中定义**
  - **服务和项目的关系：一个project当中可包含多个service，每个service中定义了容器运行的镜像，参数，依赖。一个服务当中可包括多个容器实例。**
- 使用微服务架构的系统一般包含若干个微服务，每个微服务一般部署多个实例。如果每个业务都要手动暂停，那么效率低，维护量大

YAML文件

- 使用**缩进表示层级关系**，**不允许使用Tab键**，只允许使用空格
- **# 表示注释**，从这个字符一直到行尾，都会被解析器忽略
- **对象，键值对**，使用冒号结构表示
  - animal: pets
  - hash: {name: Steve, foo: bar}
- 数字，一组连词线开头的行，构成一个数组
  - - Cat
  - - Dog
  - - Goldfish
  - 行内表示法：animal: {Cat, Dog}

docker-compose常用命令（了解）

- docker-compose --help
- **docker-compose up -d 部署容器**
  - 该命令十分强大，它将尝试自动完成包括构建镜像，创建服务，启动服务，并关联服务相关容器的一系列操作
- **docker-compose ps 只呈现当前目录下docker-compose.yml文件所部署的容器**
- docker-compose ps --services 只呈现当前目录下docker-compose.yml文件所部署的容器
- **docker-compose images 只呈现当前目录下docker-compose.yml文件所部署的镜像**
- docker-compose stop
  - 终止这个服务集合
- docker-compose stop nginx
  - 终止指定的服务
  - 启动的时候会先启动depend_on中的容器，关闭的时候不会影响到depend_on中的
- **docker-compose logs -f [services...] 查看容器的输出日志**
  - **对应于kubectl logs -f [pods...]，也可以查看日志**
- docker-compose build [services...]
- docker-compose rm nginx
  - 移除指定的容器
- docker-compose up -d --scale flask=3 organizationservice=2
  - 设置指定服务运行的容器个数

09-k8s使用

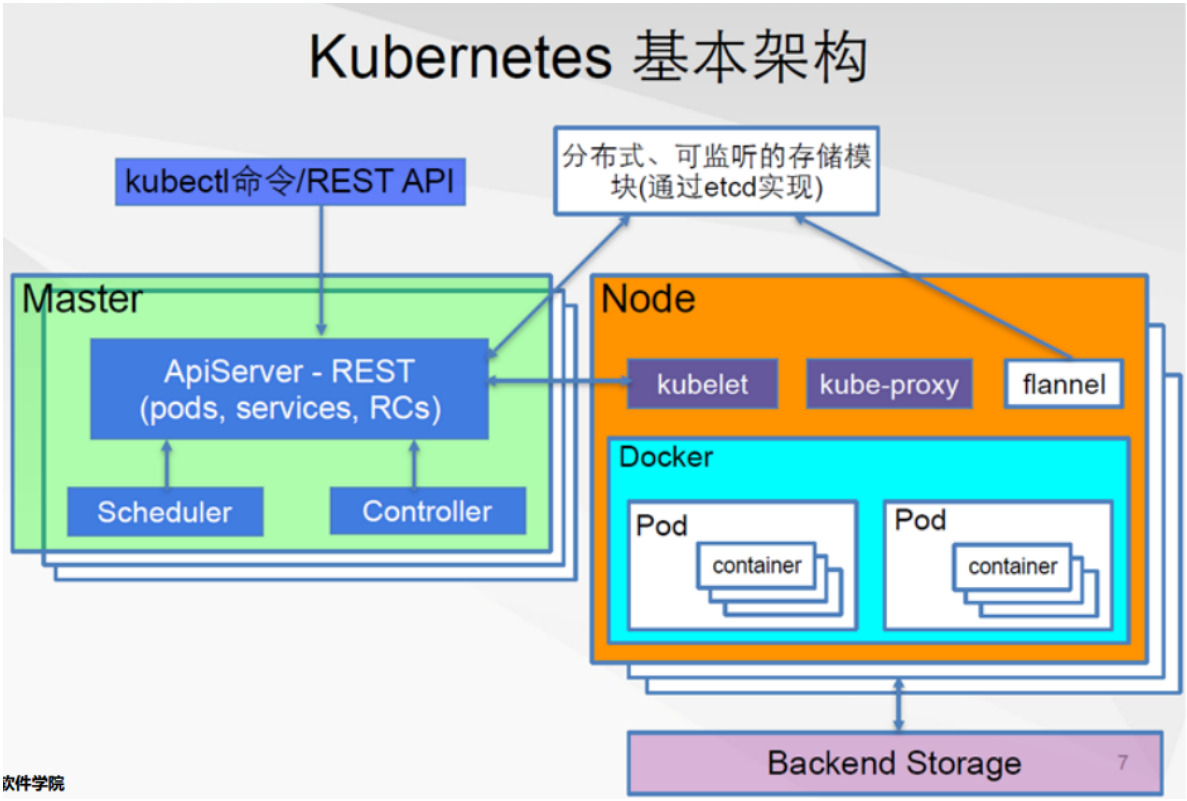
k8s中的资源

- namespace
- **Pods**
- ReplicaSet
- **Deployment**
- **Service**
- **Ingress**
- ConfigMap
- secrets
- serviceaccounts
- DaemonSet

验证Kubernetes集群状态

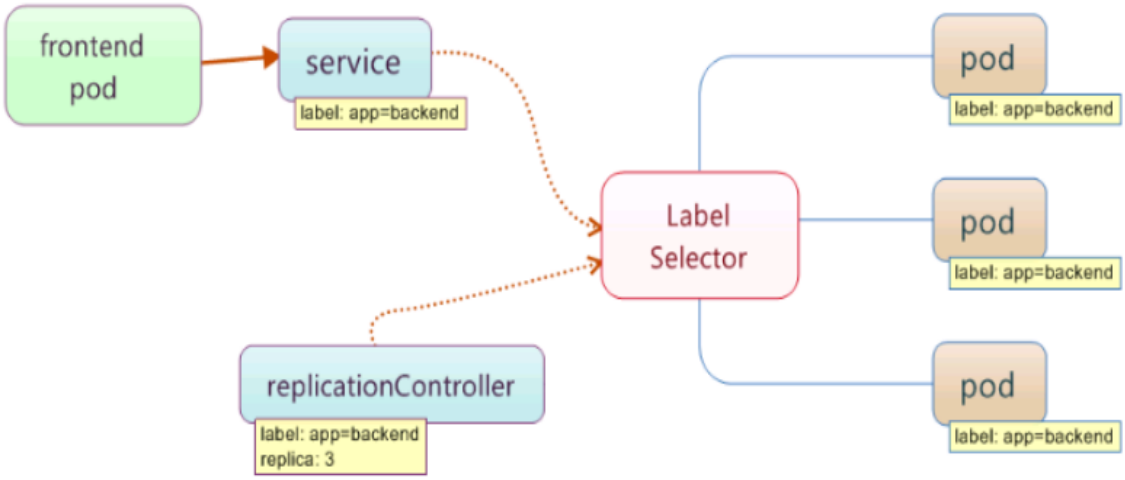
- kubectl cluster-info
- kubectl get nodes
- kubectl get nodes --show-labels
- 给节点打标签：kubectl label node docker-desktop disktype=ssd

Kubernetes基本架构



Label

- **可以给pod打标签**
- **一个service对应多个pod**
- **每个服务有一个集群IP地址，可以通过集群IP访问**

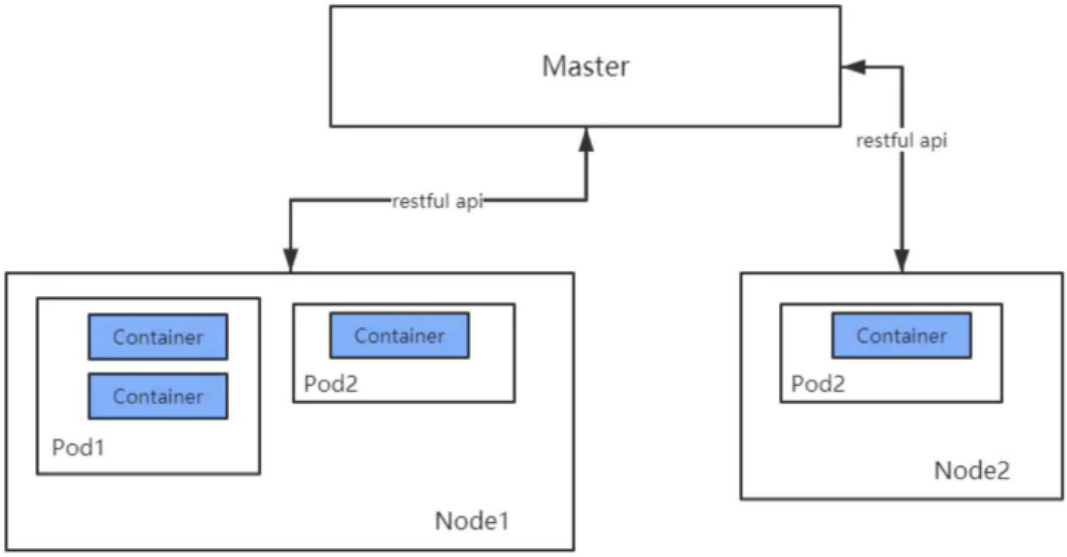


Pod

- **Pod是Kubernetes调度的最小单元**
- **Pod可以共享哪些资源**
  - **PID命名空间**：Pod中不同的应用程序可以看到其他应用程序的进程ID
  - **network命名空间**：Pod中多个容器处于同一个网络命名空间，因此能够访问的IP和端口号都是相同的。也可以通过localhost相互访问

    - **IPC命名空间**：Pod中的多个容器共享Inner-process Communication命名空间，因此可以通过SystemV IPC或POSIX进行进程间通信
    - **UTS命名空间**：Pod中的多个容器共享一个主机名
    - **Volumes**：Pod中各个容器可以共享在Pod中定义分存储卷(Volume)
- restartPolicy字段
    - Always：只要退出就重启
    - OnFailure：失败退出时(exit code不为0)重启
    - Never：永远不重启

Pod、Container与Node之间的关系



如何访问k8s中的服务

方法1：port-forward 映射端口

- **kubectl port-forward pod/myspittr 8081:8080**
    - **[本机端口] : [容器端口]**
- **kubectl port-forward service/demo 8081:80**

方法2：创建Ingress

- **要访问服务还可以创建Ingress，然后在hosts文件添加域名，通过路由访问服务**
    - kubectl create ingress myspittr --class=nginx --rule=www.demo.com/*=myspittr:8080
        - 访问：http://www.demo.com/spittr/
    - kubectl delete ingress myspittr

方法3：在k8s用curl借助http访问

- **还有一种方式，通过curl访问部署的服务**
    - kubectl run -it --rm=true mycurl --image=curlimages/curl:latest --restart=Never ==command --sh

k8s常用命令

- kubectl get secrets/pods/all [-n namespace]
- kubectl get secret mysecret -o yaml
- kubectl delete pod pod_name [-n namespace]
- kubectl apply -f [json或yaml文件]
- kubectl delete -f [json或yaml文件]
- kubectl describe secret mysecret
- kubectl logs secret1 -pod

Deployment

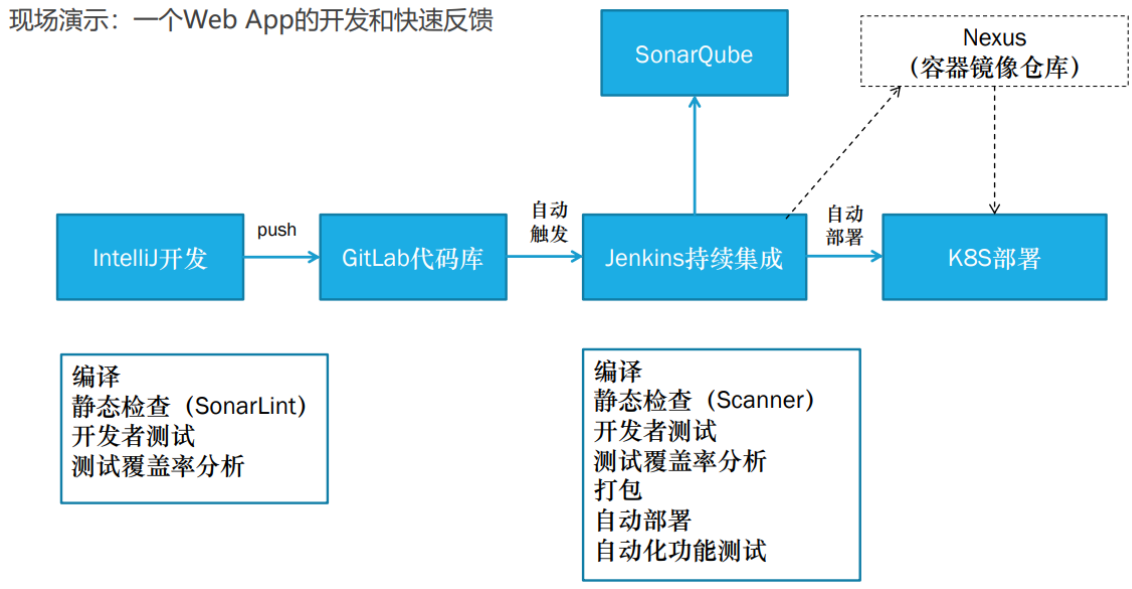
- **自动伸缩，可以根据CPU使用率伸缩**
  - kubectl **autoscale** deployment spittr --min=10 --max=15 --cpu-percent=80

如何理解k8s的service和nacos中的service的异同点?

- **共同点：**
  - **通过服务名来访问多个服务，一个服务背后有多个服务的实例，**
  - **k8s通过pod来实现，pod会消亡，pod个数会增加也会减少，nacos中服务的实例也会增加和减少。**
- **不同点：**
  - **k8s的是pod层级的，和其他资源配合使用；nacos是服务层级的，和Spring Cloud框架相关。**
  - **服务的维护还是需要k8s，并且k8s也支持服务注册与发现；nacos不能维护服务，只维护元数据**
- 两个可以二选一

10-REST服务、微服务开发与部署

WebApp的开发和快速反馈



Spring Boot

- 简化Spring Web开发
- Spring Boot Starter
  - 自动管理依赖、版本号
- 自动配置
  - 根据类路径加载的类自动创建需要的Bean
  - 如：DataSource、JdbcTemplate、视图解析器等
- Acuator
  - /autoconfig，使用了哪些自动配置(positiveMatches)
  - /beans，包含bean依赖关系

单体应用程序

- 数据库对所有模块可见
- 一个人的修改整个应用都要重新构建、测试、部署
- 整体复制分布式部署，不能拆分按需部署

微服务架构模式的特征（重要）

- **应用程序分解为具有明确定义了职责范围的细粒度组件**
- **完全独立部署，独立测试，并可复用**
- **使用轻量级通信协议，HTTP和JSON，松耦合**
- **服务实现可使用多种编程语言和技术**
- **将大型团队划分成多个小型开发团队，每个团队只负责他们各自的服务**

Spring Boot和Spring Cloud

- Spring Boot提供了基于Java的、面向REST的微服务框架
- Spring Cloud使实施和部署微服务到私有云变得更加简单

REST原则

- **Re**presentational **S**tate **T**ransfer，表现层状态转移
- 资源（Resources），就是网络上的一个实体，标识：URI
- 表现层（Representation）：json、xml、html、pdf、excel
- 状态转移（State Transfer）：服务端--客户端
- HTTP协议的四个操作方式的动词：GET、POST、PUT、DELETE
  - CRUD：Create、Read、Update、Delete
- 如果一个架构符合REST原则，就称它为RESTful架构。

HTTP状态码

- **1xx：表示服务器已接收了客户端的请求，客户端可以继续发送请求**
- **2xx：表示服务器已成功接收到请求并进行处理**
- **3xx：表示服务器要求客户端重定向**
- **4xx：表示客户端的请求有非法内容**
- **5xx：标识服务器未能正常处理客户端的请求而出现意外错误**

客户端表述的两种方式

1. 内容协商
2. 消息转换器
    - **客户端发来的JSON转Java对象：方法上加@RequestBody**
    - **返回的Java对象转JSON格式：方法上加@ResponseBody或类级@RestController**

例子代码

- **@SpringBootApplication**
    - 配置类@Configuration
    - @ComponentScan
- @RestController
    - @Controller
    - 请求响应，JSON编解码（序列化）
- **健康检查**：**localhost:8080/acuator/health （需要添加acuator依赖）**

现实和挑战

- 程序规模越来越大，越来越复杂
- 客户期望快速频繁交付
- 性能和可伸缩性
- 弹性，应用程序中某个部分的故障或问题不应该导致整个应用程序崩溃
- 小型的、简单的和解耦的服务 = 可伸缩的、有弹性的和灵活的应用程序

云计算平台

- 基础设施即服务(Infrastructrue as a Service, Iaas)
- 平台即服务(Platform as a service, Paas)
- 软件即服务(Software as a service, Saas)
- 函数即服务(Functions as a service, Faas)，将代码块以”无服务器“的形式部署，无需管理任何服务器基础设施
- 容器即服务(Container as a service, CaaS)，如亚马逊ECS

微服务开发要考虑的问题

- 微服务划分，服务粒度、通信协议、接口设计、配置管理、使用实践解耦微服务
- 服务注册、发现和路由
- 弹性，负载均衡，断路器模式（熔断），容错
- 可伸缩
- 日志记录和跟踪
- 安全
- 构造和部署，基础设施即代码

Spring Cloud的工具集成

- spring cloud alibaba
    - 数据配置，与Nacos集成
    - 服务注册与发现，与Nacos集成
        - Spring Cloud Loadbalancer
        - Spring Cloud openfeign

    - 限流、熔断，与Sentinel集成
- Spring Cloud gateway，网关服务
- Spring Cloud Stream，与RabbitMQ、Kafka集成
- Spring Cloud Sleuth，与日志聚合工具Papertrail、跟踪工具Zipkin集成
- Spring Cloud Security，与OAuth2集成

微服务划分

- 可以从数据模型入手，每个域的服务只能访问自己的表
- 刚开始粒度可以大一点，不要太细，由粗粒度重构到细粒度是比较容易的
- 设计是逐步演化的

接口设计

- 使用标准HTTP动词：GET、PUT、POST、DELETE，映射到CRUD
- 使用URI来传达意图
- 请求和响应使用JSON
- 使用HTTP状态码来传达结果

运维实践

- **所有功能代码、测试代码、脚本都在源代码库中（配置数据一般不放在库中）**
- **指定JAR依赖的版本号**
- **配置与源代码分开放**
- **已构建的服务应该是不可变的，不能再被修改**
- **微服务应该是无状态的**
- **并发，通过启动更多的微服务实例横向扩展， 多线程是纵向拓展**

11-基于NACOS的数据配置

如何基于NACOS的配置管理做微服务开发

1. **在pom.xml中加依赖：spring-cloud-starter-alibaba-nacos-config**
2. **在bootstrap.yml中定义nacos访问地址、文件后缀、服务名**
3. **在代码中加注解@Value（从配置文件获取值）、@RefreshScope（nacos刷新）**

将服务配置信息与代码分开

- 配置信息硬编码到代码中
- 分离的外部属性文件
- 与物理部署分离，如外部数据库
- k8s-configmap
- 配置数据作为单独的服务提供

微服务的配置数据来源

- Spring Cloud Config ： 文件系统、Git、Eureka、Consul
- nacos：mysql、h2、derby

nacos

- Nacos /nɑːkəʊs/ 是 Dynamic Naming and Configuration Service的首字母简称，一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台
- 主要功能：动态配置服务、服务发现及管理

curl工具

- curl 是常用的命令行工具，用来请求 Web 服务器。它的名字就是客户端（client）的 URL 工具的意思。它的功能非常强大，命令行参数多达几十种。如果熟练的话，完全可以取代 Postman 这一类的图形界面工具
- -i 打印响应的标头
- -L 让 HTTP 请求跟随服务器的重定向。curl 默认不跟随重定向。
- -v 输出通信的整个过程，用于调试
- -u 设置认证的用户名密码（下面两个等价）
    - curl http://tzs919:123456@spittr:8080/spittr/spittles
    - curl -u tzs919:123456 http://spittr:8080/spittr/spittles

dataId的完整格式（重要）

- **dataId是nacos管理的最小单位**
- ${prefix}-${spring.profiles.active}.${file-extension}
- prefix默认为**spring.application.name**的值，也可以通过配置项spring.cloud.nacos.config.prefix来配置
- spring.profiles.active即为**当前环境对应的profile**
- file-exetension为**配置内容的数据格式**，可以通过配置项spring.cloud.nacos.config.file-extension来配置。 目前只支持 properties 和 yaml类型

12-基于NACOS的服务注册与发现

nacos练习

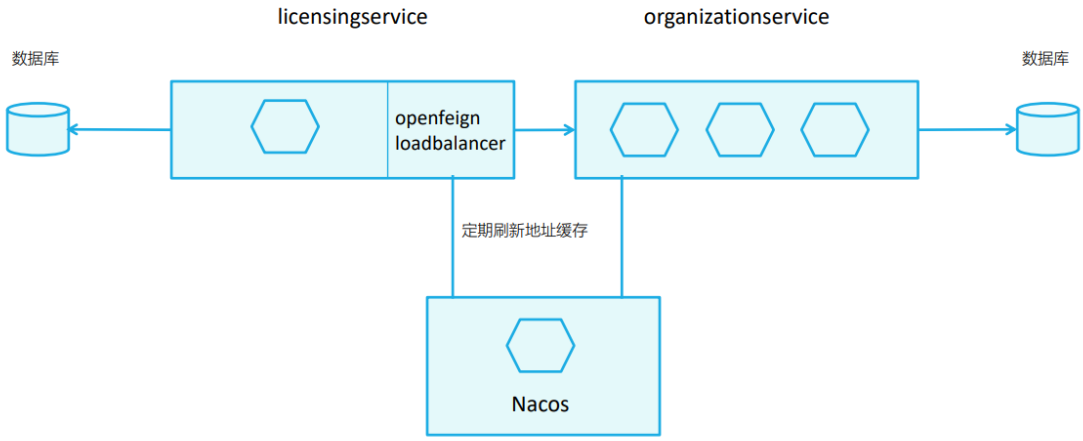
- Nacos中临时实例基于心跳上报方式维持活性，基本的健康检查流程基本如下：Nacos客户端会维护一个定时任务，每隔5秒 发送一次心跳请求，以确保自己处于活跃状态。Nacos服务端在15秒内如果没收到客户端的心跳请求，会将该实例设置为不健 康，在30秒内没收到心跳，会将这个临时实例摘除
- 这个心跳间隔、心跳超时可以自定义

服务发现的好处

- **快速水平伸缩，而不是垂直伸缩。不影响客户端**
- **提高应用程序的弹性**

服务部署

- 注意，服务之间不是借助nacos通信，而是直接通过IP地址通信



Spring Cloud家族

	Spring Cloud Netflix	Spring Cloud 官方	Spring Cloud Zookeeper	Spring Cloud Consul	Spring Cloud Kubernetes	Spring Cloud Alibaba
分布式配置	Archaius	Spring Cloud Config	Zookeeper	Consul	ConfigMap	Nacos
服务注册/发现	Eureka	–	Zookeeper	Consul	Api Server	Nacos
服务熔断	Hystrix	–	–	–	–	Sentinel
服务调用	Feign	OpenFeign RestTemplate	–	–	–	Dubbo RPC
服务路由	Zuul	Spring Cloud Gateway	–	–	–	Dubbo PROXY
分布式消息	–	SCS RabbitMQ	–	–	–	SCS RocketMQ
负载均衡	Ribbon	–	–	–	–	Dubbo LB
分布式事务	–	–	–	–	–	–

Spring Cloud Alibaba

- Spring Cloud Alibaba是Spring Cloud的子项目，符合Spring Cloud的标准，致力于提供微服务开发的一站式解决方案
- **包含nacos，sentinel**

使用到的starter依赖

- 服务配置：com.alibaba.cloud，**spring-cloud-starter-alibaba-nacos-config**
- 服务注册：com.alibaba.cloud，**spring-cloud-starter-alibaba-nacos-discovery**
- 客户端负载均衡：org.springframework.cloud，**spring-cloud-starter-loadbalancer**
- 简化客户端调用：org.springframework.cloud，**spring-cloud-starter-openfeign**

使用的注解

- @EnableDiscoveryClient
- @EnableFeignClients
- @LoadBalanced

调用服务的三种方式

- **不建议用第一种，一般用第二、三，因为可以做负载均衡，更多用第三种**
- **Spring DiscoveryClient**
- **使用支持LoadBalanced的RestTemplate**
- **使用OpenFeign（@FeignClient）**
  - OpenFeign是一款声明式、模板化的HTTP客户端， Feign可以帮助我们更快捷、优雅地调用HTTP API

有用的命令

- 查看日志：kubectl logs -f -l app=organizationservice --all-containers=true
- 重部署：kubectl rollout restart deployment organizationservice
- 扩容：kubectl scale deployment organizationservice --replicas 5

健康检查

- 临时实例的客户端主动上报机制，**临时实例每隔 5s 发送一个心跳包给 Nacos 服务器端**
  - 学习参考（源码剖析）
  - 客户端调用如何应对服务状态不能及时更新，请参考学习
- 永久实例的服务端反向探测机制，永久实例支持 3 种探测协议，TCP、HTTP 和 MySQL，默认探测协议为 TCP， 也就是通过不断 ping 的方式来判断实例是否健康。
- **两种健康检查机制的学习参考：**
  - **客户端主动上报机制**
  - **服务器端反向探测机制**

负载均衡策略

- roundLoadBalancer
- randomLoadBalancer
- @LoadBalancerClient(name = "organizationservice", configuration = Application.class)
- 对第二、三种调用都有效

使用nacos进行服务注册与发现的步骤

1. **pom.xml加依赖：nacos-discovery, nacos-config, load-balancer, openfeign**
2. **bootstrap.yml定义nacos访问地址**
3. **在启动类加@EnableDiscoveryClient，@EnableFeignClients**
4. **定义接口，加上@FeignClient("【服务名】")**
5. **在其他类用@Autowired注入这个接口，接口不用实现**

13-基于Sentinel的流控与熔断

Sentinel定义资源

- 资源是 Sentinel 的关键概念。它可以是 Java 应 用程序中的任何内容，例如，由应用程序提供的 服务，或由应用程序调用的其它应用提供的服务， 甚至可以是一段代码
- 只要通过 Sentinel API 定义的代码，就是资源， 能够被 Sentinel 保护起来。大部分情况下，可以使用方法签名，URL，甚至服务名称作为资源名 来标示资源

定义资源的三种方式

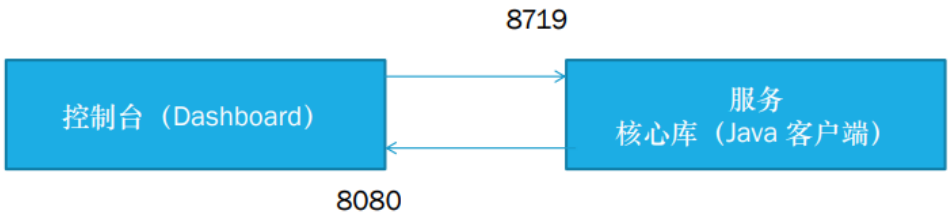
1. **代码直接定义**
2. **使用注解定义**
3. **SpringCloud自动定义**
4. 配置文件只能定义规则，不能定义资源

规则的种类

- **流量控制规则 FlowException属于BlockException的子类**
- **熔断降级规则 DegradeException属于BlockException的子类**
  - **针对耗时长**
  - **针对业务本身抛出异常**
  - **熔断策略：慢调用比例、异常的比例、异常数目**
- 系统保护规则
- 来源访问控制规则
- 热点参数规则

Sentinel组成

- 核心库（Java 客户端）：不依赖任何框架/库，能够运行于 Java 8 及以上的版本的运行时环境，同时对 Dubbo / Spring Cloud 等框架也有较好的支持
- 控制台（Dashboard）：Dashboard 主要负责管理推送规则、监控、管理机器信息等，**控制台不会保存规则，服务结束规则消失**



查看效果

- **QPS**一般指每秒查询率。 每秒查询率（QPS，Queries-per-second）是对一个特定的查询服务器在规定时间内 所处理流量多少的衡量标准