1、单体架构和分布式架构

单体架构：将业务的所有功能集中在一个项目中开发,打成一个包部署。

分布式架构：根据业务功能对系统做拆分，**每个业务功能模块**作为独立项目开发，称为一个服务。（降低服务耦合）

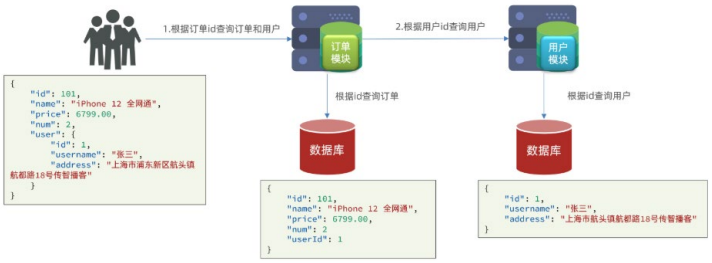
2、服务拆分原则

1）不同微服务，**不要重复开发相同业务**；

2）微服务**数据独立**，不要访问其它微服务的数据库；

3）微服务可以**将自己的业务暴露为接口**，供其它微服务调用。

3、使用 RestTemplate 实现远程调用（服务与服务之间**根据 url 调用**）



1）**注册一个 RestTemplate 的实例**到 Spring 容器；

```
@MapperScan("cn.itcast.order.mapper")
@SpringBootApplication
public class OrderApplication {

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

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

2）修改 order-service 服务中的 OrderService 类中的 queryOrderById 方法，加上远程查询 user：根据 Order 对象中的 userId **构造 url 地址**，然后使用 **getForObject(url,User.class)**查询到 User 对象；

3）将查询的 User 填充到 Order 对象，一起返回。

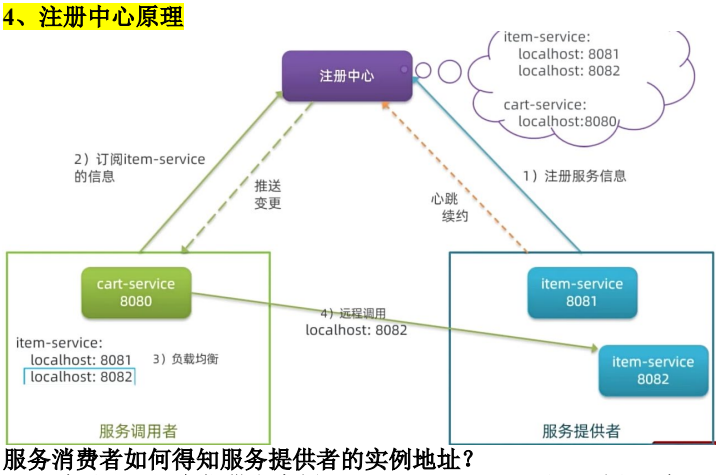
```
@Service
public class OrderService {

    @Autowired
    private OrderMapper orderMapper;

    @Autowired
    private RestTemplate restTemplate;

    public Order queryOrderById(Long orderId) {
        // 1.查询订单
        Order order = orderMapper.findById(orderId);
        // 2.远程查询user
        // 2.1.url地址
        String url = "http://localhost:8081/user/" + order.getUserId();
        // 2.2.发起调用
        User user = restTemplate.getForObject(url, User.class);
        // 3.存入order
        order.setUser(user);
        // 4.返回
        return order;
    }
}
```

4、注册中心原理



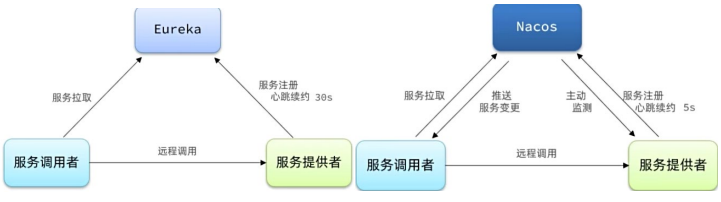
服务消费者如何得知服务提供者的实例地址？

1）服务注册：服务提供者实例启动后将自己的信息**注册到注册中心服务端**；（并且通过**心跳续约**判断服务提供者是否宕机，异常则剔除）

2）注册中心服务端：保存**服务名称**到**服务实例地址列表**的**映射关系**；

3）服务发现：服务消费者根据**服务名称**拉取**实例地址列表**，（并且采用**负载均衡算法**选中一个地址进行远程调用）。

【Eureka 和 Nacos 的区别】



同：都支持**服务注册**、**服务拉取**、**心跳续约**；

异：1）Nacos 支持服务端**主动检测提供者状态**；

2）**临时**实例用**心跳**（异常则**剔除**），**永久**实例用**主动检测**（异常**不剔除**）；并且 Nacos 心跳 **5 秒**，Eureka 心跳 **30 秒**；

3）Nacos 支持**服务列表变更的消息推送**，服务列表更新更及时；

4）Nacos 集群**默认 AP 方式**，支持 **CP 方式**；Eureka 集群是 **AP 方式**。

5、服务注册到 nacos

1）引入依赖

在cloud-demo父工程的pom文件中的<dependencyManagement>中引入SpringCloudAlibaba的依赖：

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-alibaba-dependencies</artifactId>
    <version>2.2.6.RELEASE</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

然后在user-service和order-service中的pom文件中引入nacos-discovery依赖：

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

2）配置 nacos 地址（spring.cloud.nacos.**server-addr:**）

在user-service和order-service的application.yml中添加nacos地址：

```
spring:
  cloud:
    nacos:
      server-addr: localhost:8848
```

6、nacos 服务分级存储模型

一个服务**可以有多个实例**（不同的地址），多个实例还可能分布在不同的机房，故将**同一机房内的实例**划分为一个**集群**。微服务互相访问时，应该**尽可能访问同集群实例**，因为本地访问速度更快。当本集群内不可用时，才访问其它集群。

1）配置集群（spring.cloud.nacos.**discovery.cluster-name:**）

```
spring:
  cloud:
    nacos:
      server-addr: localhost:8848
      discovery:
        cluster-name: HZ # 集群名称
```

2）同集群优先的负载均衡

默认的 **ZoneAvoidanceRule** 并不能实现根据同集群优先来实现负载均衡。因此 Nacos 中提供了一个 **NacosRule** 的实现，可以**优先从同集群中挑选实例**。

2）修改负载均衡规则

修改order-service的application.yml文件，修改负载均衡规则：

```
userservice:
  ribbon:
    NFLoadBalancerRuleClassName: com.alibaba.cloud.nacos.ribbon.NacosRule # 负载均衡规则
```

3）权重配置

默认情况下 **NacosRule** 是同集群内**随机挑选**，不会考虑机器的性能问题。因此，Nacos 提供了**权重配置来控制访问频率**，权重越大则访问频率越高。

4）环境隔离

Nacos 提供了 namespace 来实现环境隔离功能。nacos 中可以有多个 namespace，**不同 namespace 之间相互隔离**，例如不同 namespace 的服务互相不可见。

给微服务配置 namespace：（spring.cloud.nacos.**discovery.namespace:**）

```
spring:
  cloud:
    nacos:
      server-addr: localhost:8848
      discovery:
        cluster-name: HZ
        namespace: 492a7d5d-237b-46a1-a99a-fa8e98e4b0f9 # 命名空间，填ID
```

7、Feign 远程调用替代 RestTemplate

1）在**服务消费者的 pom 文件**中引入依赖

我们在order-service服务的pom文件中引入feign的依赖：

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

2）在服务消费者的**启动类上添加注解@EnableFeignClients**

```
@EnableFeignClients
@MapperScan("cn.itcast.order.mapper")
@SpringBootApplication
public class OrderApplication {

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

3）在**服务消费者中新建一个接口，编写 Feign 的客户端**

客户端主要是基于 SpringMVC 的注解来声明远程调用的信息：

1>**@FeignClient**("需要调用的服务生产者的**服务名称**")；

2>**接口方法上**写**请求方式**和具体的**请求路径**；

3>**接口方法前**有**返回值类型**，**接口方法里**有**请求参数**。

这样 Feign 就可以帮助我们发送 http 请求获取对应实例了

```
@FeignClient("userservice")
public interface UserClient {
    @GetMapping("/user/{id}")
    User findById(@PathVariable("id") Long id);
}
```

4）在服务消费者中 service 类的具体方法中**直接调用此方法**即可

修改order-service中的OrderService类中的queryOrderById方法，使用Feign客户端代替RestTemplate：

```
@Autowired
private UserClient userClient;

public Order queryOrderById(Long orderId) {
    // 1.查询订单
    Order order = orderMapper.findById(orderId);
    // 2.利用Feign发起http请求，查询用户
    User user = userClient.findById(order.getUserId());
    // 3.封装user到Order
    order.setUser(user);
    // 4.返回
    return order;
}
```

8、Feign 自定义配置

类型	作用	说明
feign.Logger.Level	修改日志级别	包含四种不同的级别：NONE、BASIC、HEADERS、FULL
feign.codec.Decoder	响应结果的解析器	http远程调用的结果做解析，例如解析json字符串为java对象
feign.codec.Encoder	请求参数编码	将请求参数编码，便于通过http请求发送
feign. Contract	支持的注解格式	默认是SpringMVC的注解
feign. Retryer	失败重试机制	请求失败的重试机制，默认是没有，不过会使用Ribbon的重试

1）配置文件方式（feign.client.config.**userservice/default.配置：**）

基于配置文件修改feign的日志级别可以针对单个服务：

```
feign:
  client:
    config:
      userservice: # 针对某个微服务的配置
        loggerLevel: FULL #  日志级别
```

也可以针对所有服务：

```
feign:
  client:
    config:
      default: # 这里用default就是全局配置，对所有要访问的微服务都生效
        loggerLevel: FULL #  日志级别
```

2）Java 代码配置类方式

全局生效：将其放到启动类的@EnableFeignClients 这个注解中；

局部生效：则把它放到对应的@FeignClient 这个注解中。

```
public class DefaultFeignConfiguration  {
    @Bean
    public Logger.Level feignLogLevel(){
        return Logger.Level.BASIC; // 日志级别为BASIC
    }
}
```

如果要**全局生效**，将其放到启动类的@EnableFeignClients这个注解中：

```
@EnableFeignClients(defaultConfiguration = DefaultFeignConfiguration .class)
```

如果是**局部生效**，则把它放到对应的@FeignClient这个注解中：

```
@FeignClient(value = "userservice", configuration = DefaultFeignConfiguration .class)
```

9、Feign 使用优化

Feign 底层发起 http 请求，依赖于其它的框架。其底层客户端实现**默认为 URLConnection（不支持连接池）**，所以可以通过配置**改用 HttpClient 的 feign 去支持连接池**操作；

1）引入 HttpClient 依赖：

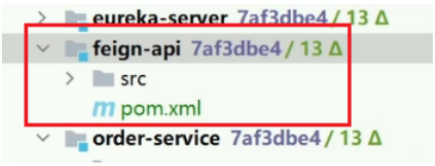
```
<!--httpClient的依赖 -->
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-httpclient</artifactId>
</dependency>
```

2）配置连接池

```
feign:
  client:
    config:
      default: # default全局的配置
        loggerLevel: BASIC # 日志级别，BASIC就是基本的请求和响应信息
  httpclient:
    enabled: true # 开启feign对HttpClient的支持
    max-connections: 200 # 最大的连接数
    max-connections-per-route: 50 # 每个路径的最大连接数
```

10、通过抽取方式将 Feign 的 Client 抽取为独立模块，供所有消费者

1）首先创建一个 module，命名为 feign-api，并在 pom.xml 中引入依赖：



在feign-api中然后引入feign的starter依赖

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

2）在服务消费者模块 pom.xml 中引入 feign-api 的依赖：

在order-service的pom文件中中引入feign-api的依赖：

```
<dependency>
    <groupId>cn.itcast.demo</groupId>
    <artifactId>feign-api</artifactId>
    <version>1.0</version>
</dependency>
```

3）扫描包问题：（feign-api 包不在服务消费者启动类扫描包范围内）

方式一：在**服务消费者启动类**上指定 **Feign 应该扫描的包**：（全扫）

方式二：在**服务消费者启动类**上指定**需要加载的 Client 接口**：（单扫）

方式一：

指定Feign应该扫描的包：

```
@EnableFeignClients(basePackages = "cn.itcast.feign.clients")
```

方式二：

指定需要加载的Client接口：

```
@EnableFeignClients(clients = {UserClient.class})
```

11、为什么需要网关？（网关的作用）

1）权限控制：网关作为**微服务入口**，需要**校验用户是是否有请求资格**，如果没有则进行拦截。

2）路由和负载均衡：一切请求都必须先经过 gateway，但**网关不处理业务**，而是根据某种规则，把**请求转发到某个微服务**，这个过程叫做**路由**。当然路由的**目标服务有多个时**，还需要做**负载均衡**。

3）限流：当请求流量过高时，在网关中**按照下流的微服务能够接受的速度**来放行请求，**避免服务压力过大**。

2、Spring Cloud Gateway 的工作流程？

路由判断 → 请求过滤 → 服务处理 → 响应过滤 → 响应返回

客户端的请求先**通过匹配规则找到合适的路由**，就能**映射到具体的服务**。然后请求经过**前置过滤器**处理后**转发给具体的服务**，服务处理后，再次经过**后置过滤器**处理，最后**返回给客户端**。

3、Spring Cloud Gateway 的断言是什么？(predicates)

在 Gateway 中，如果客户端发送的请求满足了断言的条件，则映射到指定的路由器，就能转发到指定的服务上进行处理。

名称	说明	示例
After	是某个时间点后的请求	- After=2037-01-20T17:42:47.789-07:00[America/Denver]
Before	是某个时间点之前的请求	- Before=2031-04-13T15:14:47.433+08:00[Asia/Shanghai]
Between	是某两个时间点之前的请求	- Between=2037-01-20T17:42:47.789-07:00[America/Denver], 2037-01-21T17:42:47.789-07:00[America/Denver]
Cookie	请求必须包含某些cookie	- Cookie=chocolate, ch.p
Header	请求必须包含某些header	- Header=X-Request-Id, \d+
Host	请求必须是访问某个host（域名）	- Host=**.somehost.org**,.anotherhost.org
Method	请求方式必须是指定方式	- Method=GET,POST
Path	请求路径必须符合指定规则	- Path=/red/{segment},/blue/**
Query	请求参数必须包含指定参数	- Query=name, Jack或者- Query=name
RemoteAddr	请求者的ip必须是指定范围	- RemoteAddr=192.168.1.1/24
Weight	权重处理	

4、Spring Cloud Gateway 的路由和断言是什么关系？

1）**一个路由**规则包含**多个断言**，则需要**同时满足**才能匹配。如路由 Route 配置了两个断言，客户端发送的请求必须同时满足这两个断言，才能匹配路由 Route。

2）**一个请求**可以匹配**多个路由**，则映射**第一个匹配成功**的路由。如客户端发送的请求满足 Route1 和 Route2 的断言，但是 Route1 的配置在**配置文件中靠前**，所以只会匹配 Route1。

5、Spring Cloud Gateway 如何实现动态路由？

【需求】Spring Cloud Gateway 作为**微服务的入口**，需要**尽量避免重启**，而现在配置更改需要重启服务不能满足实际生产过程中的动态刷新、实时变更的业务需求，所以我们需要在 Spring Cloud Gateway **运行时动态配置网关**。

【解决】**基于 Nacos 注册中心**来做动态路由。Spring Cloud Gateway 可以**从注册中心获取服务的元数据**（例如服务名称、路径等），然后根据这些信息**自动生成路由规则**。这样，当你添加、移除或更新服务实例时，网关**会自动感知**并相应地**调整路由规则**，无需手动维护路由配置。

6、过滤器（spring.cloud.gateway.default-filters(/routes.filters)）

1）默认过滤器：通过配置定义，应用在所有路由上的过滤器。

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: user-service
        uri: lb://userservice
        predicates:
        - Path=/user/**
      default-filters: # 默认过滤项
      - AddRequestHeader=Truth, Itcast is freaking awesome!
```

2）当前路由过滤器：通过配置定义，应用在当前路由上的过滤器。

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: user-service
        uri: lb://userservice
        predicates:
        - Path=/user/**
        filters: # 过滤器
        - AddRequestHeader=Truth, Itcast is freaking awesome! # 添加请求头
```

名称	说明
AddRequestHeader	给当前请求添加一个请求头
RemoveRequestHeader	移除请求中的一个请求头
AddResponseHeader	给响应结果中添加一个响应头
RemoveResponseHeader	从响应结果中移除有一个响应头
RequestRateLimiter	限制请求的流量

3）全局过滤器接口：自定义代码逻辑，应用在所有路由上的过滤器。

```java
     *  处理当前请求，有必要的话通过{@link GatewayFilterChain}将请求交给下一个过滤器处理
     *
     * @param exchange 请求上下文，里面可以获取Request、Response等信息
     * @param chain 用来把请求委托给下一个过滤器
     * @return {@code Mono<Void>} 返回标示当前过滤器业务结束
     */
@Order(-1)
@Component
public class AuthorizeFilter implements GlobalFilter {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        // 1.获取请求参数
        MultiValueMap<String, String> params = exchange.getRequest().getQueryParams();
        // 2.获取authorization参数
        String auth = params.getFirst("authorization");
        // 3.校验
        if ("admin".equals(auth)) {
            // 放行
            return chain.filter(exchange);
        }
        // 4.拦截
        // 4.1.禁止访问，设置状态码
        exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
        // 4.2.结束处理
        return exchange.getResponse().setComplete();
    }
}
```

7、过滤器执行顺序

1）每一个过滤器都必须指定一个 int 类型的 order 值，**order 值越小，优先级越高，执行顺序越靠前**；

2）**GlobalFilter** 通过实现 Ordered 接口，或者添加**@Order 注解来指定** order 值；

3）**路由过滤器**和**默认过滤器**的 order 由 Spring 指定，默认是**按照声明顺序**从 1 递增。

4）当过滤器的 **order 值一样**时，会按照 defaultFilter > 路由过滤器 > GlobalFilter 的顺序执行。

8、Spring Cloud Gateway 支持限流吗？

Spring Cloud Gateway 自带了**限流过滤器**，对应的**接口是 RateLimiter**，RateLimiter 接口只有一个**实现类 RedisRateLimiter**（基于 Redis + Lua 实现的限流），提供的**限流功能**比较**简易**且**不易使用**。

Spring Cloud Gateway 可以结合 **Sentinel** 提供的**两种资源维度的限流**（**route 维度**和**自定义 API 维度**）实现更强大的网关流量控制。

9、什么是跨域问题？

跨域：**域名不一致**，包括：1）**域名不同**；2）域名相同，**端口不同**。

跨域问题：**浏览器禁止**请求的发起者与服务端发生**跨域 ajax 请求**，**请求被浏览器拦截**的问题。

【解决】

1）方式 1：controller 类上添加@CrossOrigin 标签；

2）方式 2：在网关中添加配置类 CorsConfig.java 解决跨域：

```java
@Configuration
public class CorsConfig {

    @Bean
    public CorsWebFilter corsFilter() {
        CorsConfiguration config = new CorsConfiguration();
        //跨域的请求方法
        config.addAllowedMethod("*");
        //跨域的请求地址
        config.addAllowedOrigin("*");
        //跨域的请求头
        config.addAllowedHeader("*");
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource(new
PathPatternParser());
        //对接口配置跨域设置
        source.registerCorsConfiguration("/**", config);
        return new CorsWebFilter(source);
    }
}
```

在 gateway 服务的 application.yml 文件中，添加下面的配置：

```yaml
spring:
  cloud:
    gateway:
      globalcors: # 全局的跨域处理
        add-to-simple-url-handler-mapping: true # 解决options请求被拦截问题
        corsConfigurations:
          '[/**]':
            allowedOrigins: # 允许哪些网站的跨域请求
              - "http://localhost:8090"
            allowedMethods: # 允许的跨域ajax的请求方式
              - "GET"
              - "POST"
              - "DELETE"
              - "PUT"
              - "OPTIONS"
            allowedHeaders: "*" # 允许在请求中携带的头信息
            allowCredentials: true # 是否允许携带cookie
            maxAge: 360000 # 这次跨域检测的有效期
```