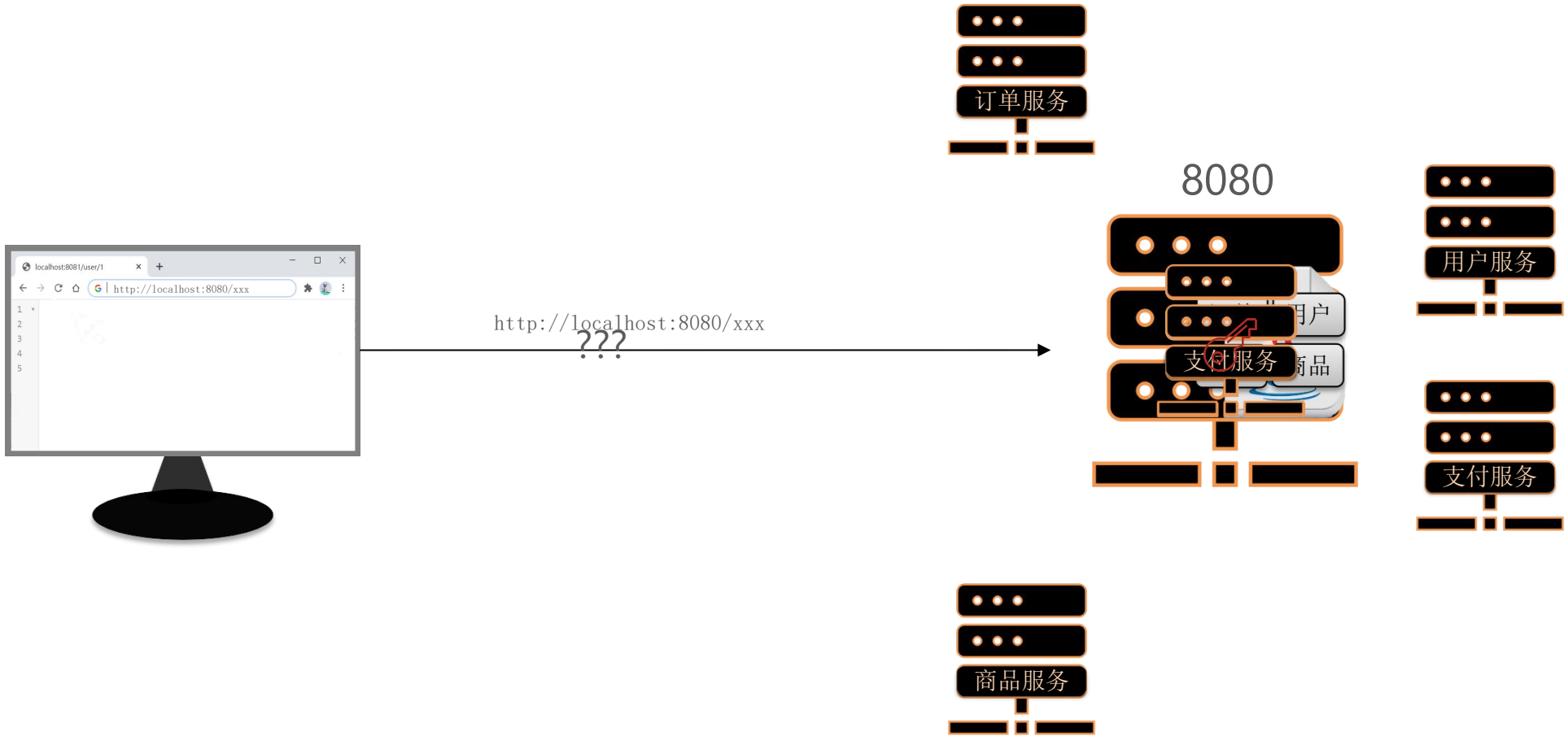
微服务

网关及配置管理

黑马程序员
www.itheima.com

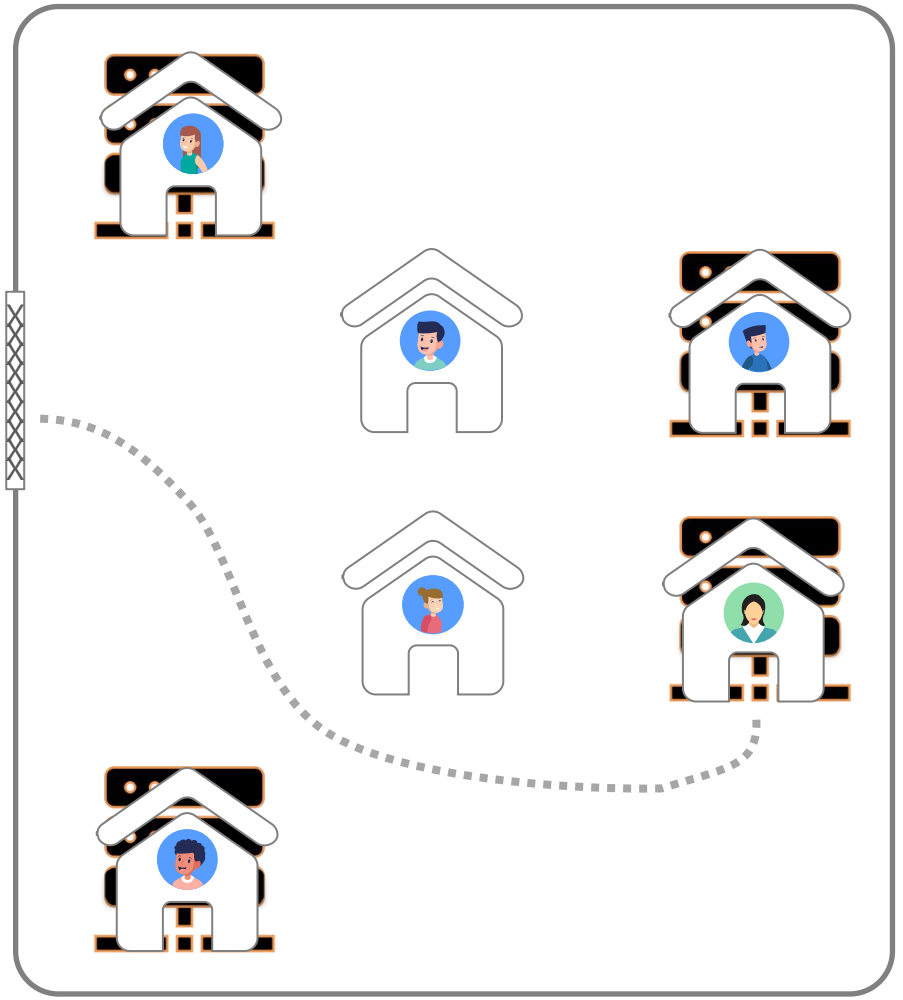
传智教育旗下
高端IT教育品牌

网关


订单服务
8080
用户服务
支付服务
支付服务
商品服务
http://localhost:8080/xxx
???
localhost:8081/user/1
http://localhost:8080/xxx

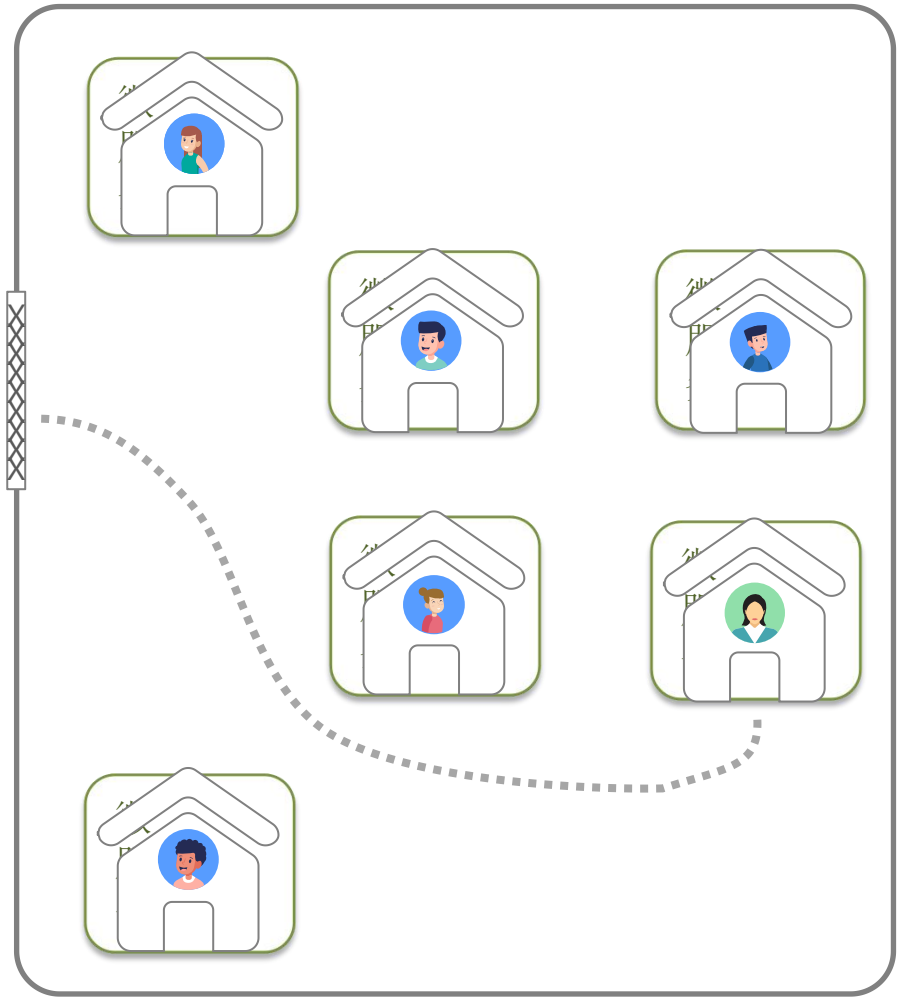
网关

网关：就是网络的关口，负责请求的路由、转发、身份校验。



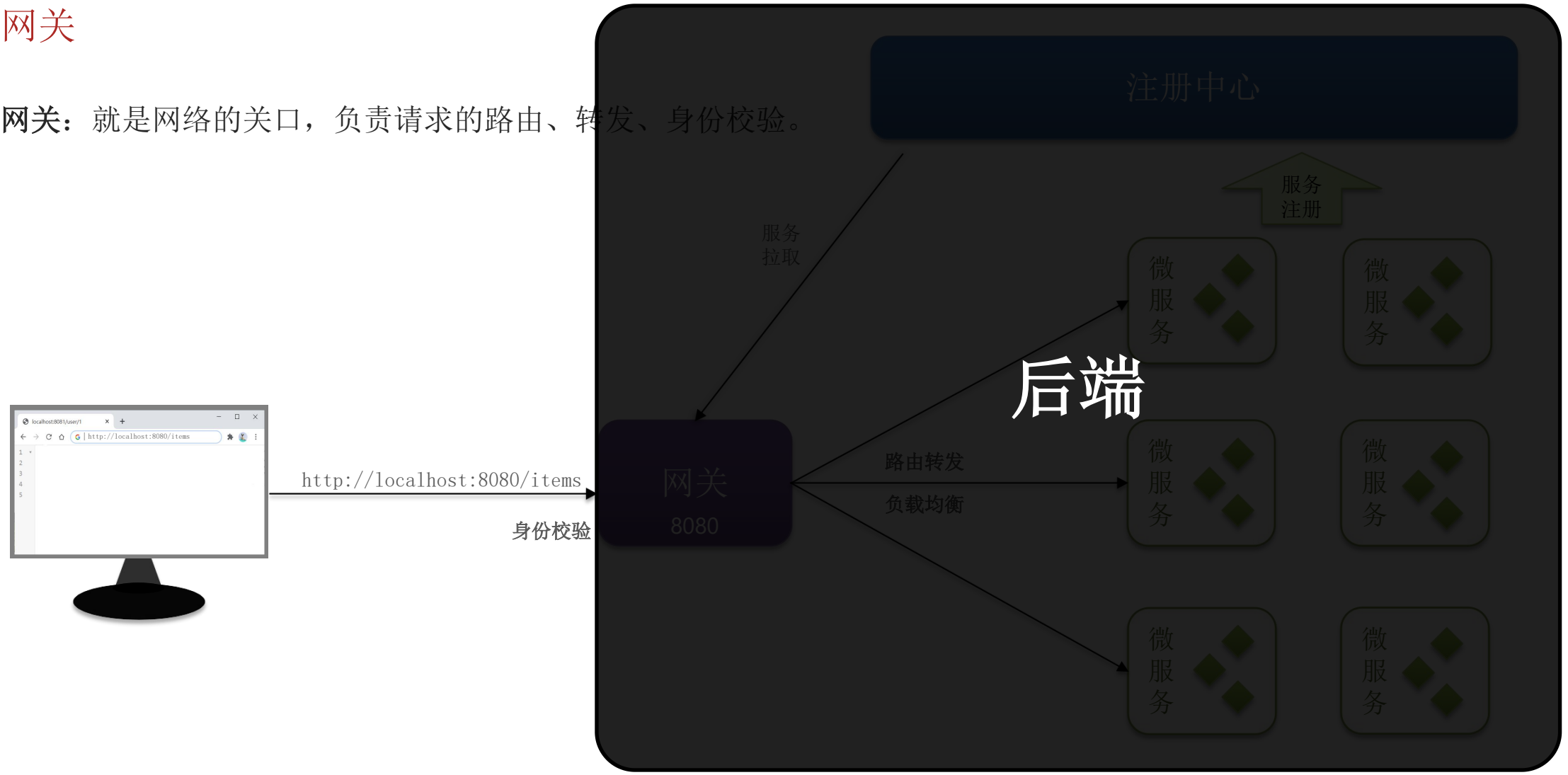
网关

网关：就是网络的关口，负责请求的路由、转发、身份校验。



网关

网关：就是网络的关口，负责请求的路由、转发、身份校验。



网关

在SpringCloud中网关的实现包括两种：

Spring Cloud Gateway

- Spring官方出品
- 基于WebFlux响应式编程
- 无需调优即可获得优异性能

Netfilx Zuul

- Netflix出品
- 基于Servlet的阻塞式编程
- 需要调优才能获得与SpringCloudGateway类似的性能

目录

Contents

- 网关路由
- 身份校验
- 配置管理

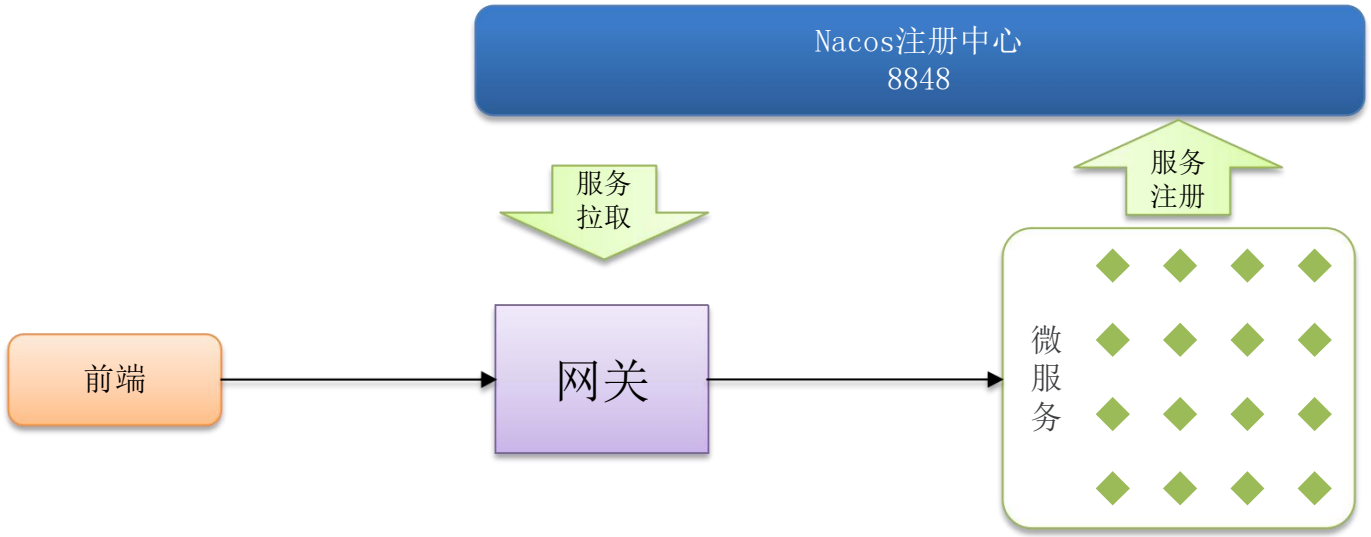
网关路由

目录

Contents

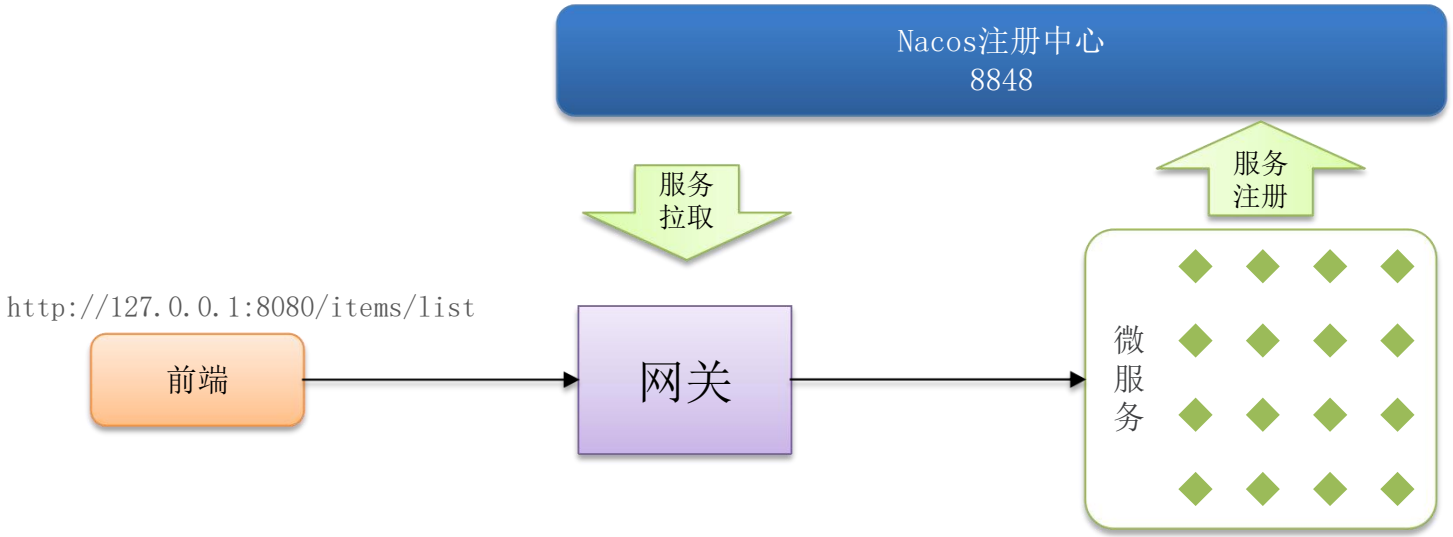
快速入门

1. 创建新模块
2. 引入网关依赖
3. 编写启动类
4. 配置路由规则



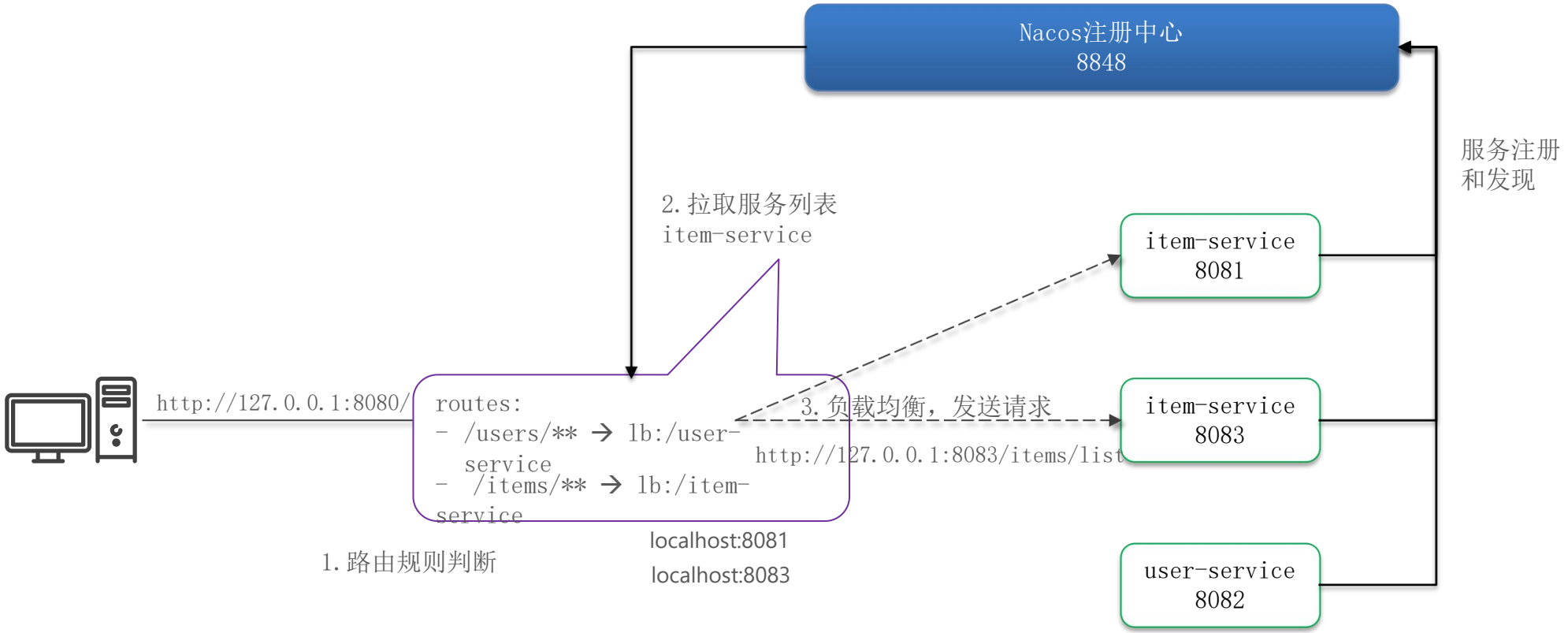
快速入门

1. 创建新模块
2. 引入网关依赖
3. 编写启动类
4. **配置路由规则**




```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: item # 路由规则id，自定义，唯一
          uri: lb://item-service # 路由目标微服务，lb代表负载均衡
          predicates: # 路由断言，判断请求是否符合规则，符合则路由到目标
            - Path=/items/** # 以请求路径做判断，以/items开头则符合
        - id: xx
          uri: lb://xx-service
          predicates:
            - Path=/xx/**
```

快速入门


Nacos注册中心
8848
服务注册
和发现
2.拉取服务列表
item-service
item-service
8081
item-service
8083
user-service
8082
http://127.0.0.1:8080/
routes:
- /users/** → lb:/user-service
- /items/** → lb:/item-service
3.负载均衡，发送请求
http://127.0.0.1:8083/items/list
1.路由规则判断
localhost:8081
localhost:8083

快速入门

1. 创建新模块
2. 引入网关依赖
3. 编写启动类
4. **配置路由规则**

```
<!--网关-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

```
spring:
  cloud:
      gateway:
    routes:
      - id: item # 路由规则id，自定义，唯一
        uri: lb://item-service # 路由目标微服务，lb代表负载均衡
        predicates: # 路由断言，判断请求是否符合规则，符合则路由到目标
          - Path=/items/** # 以请求路径做判断，以/items开头则符合
      - id: xx
        uri: lb://xx-service
        predicates:
          - Path=/xx/**
```

目录

Contents

路由属性

网关路由对应的Java类型是RouteDefinition，其中常见的属性有：

- id：路由唯一标示
- uri：路由目标地址
- predicates：路由断言，判断请求是否符合当前路由。
- filters：路由过滤器，对请求或响应做特殊处理。

路由断言

Spring提供了12种基本的RoutePredicateFactory实现：

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	- Weight=group1, 2
XForwarded Remote Addr	基于请求的来源IP做判断	- XForwardedRemoteAddr=192.168.1.1/24

路由过滤器

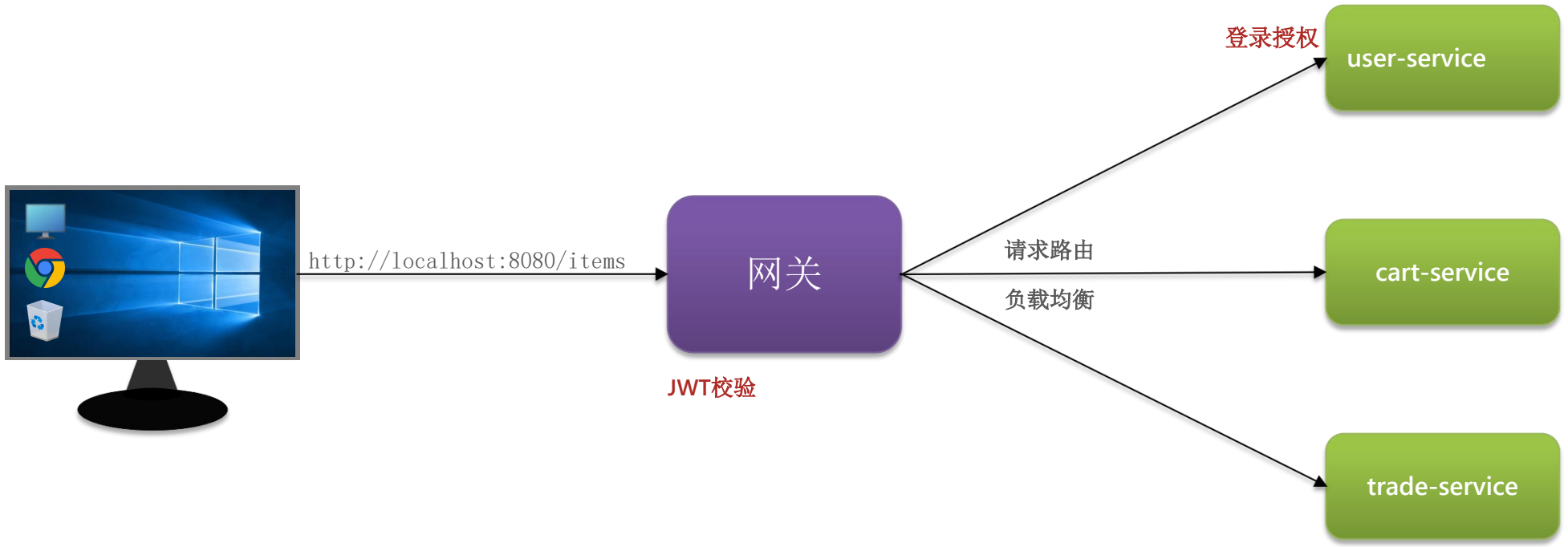
网关中提供了33种路由过滤器，每种过滤器都有独特的作用。

名称	说明	示例
AddRequestHeader	给当前请求添加一个请求头	AddrequestHeader=headerName,headerValue
RemoveRequestHeader	移除请求中的一个请求头	RemoveRequestHeader=headerName
AddResponseHeader	给响应结果中添加一个响应头	AddResponseHeader=headerName,headerValue
RemoveResponseHeader	从响应结果中移除有一个响应头	RemoveResponseHeader=headerName
RewritePath	请求路径重写	RewritePath=/red/?(?.*), /$¥{segment}
StripPrefix	去除请求路径中的N段前缀	StripPrefix=1，则路径/a/b转发时只保留/b
...		

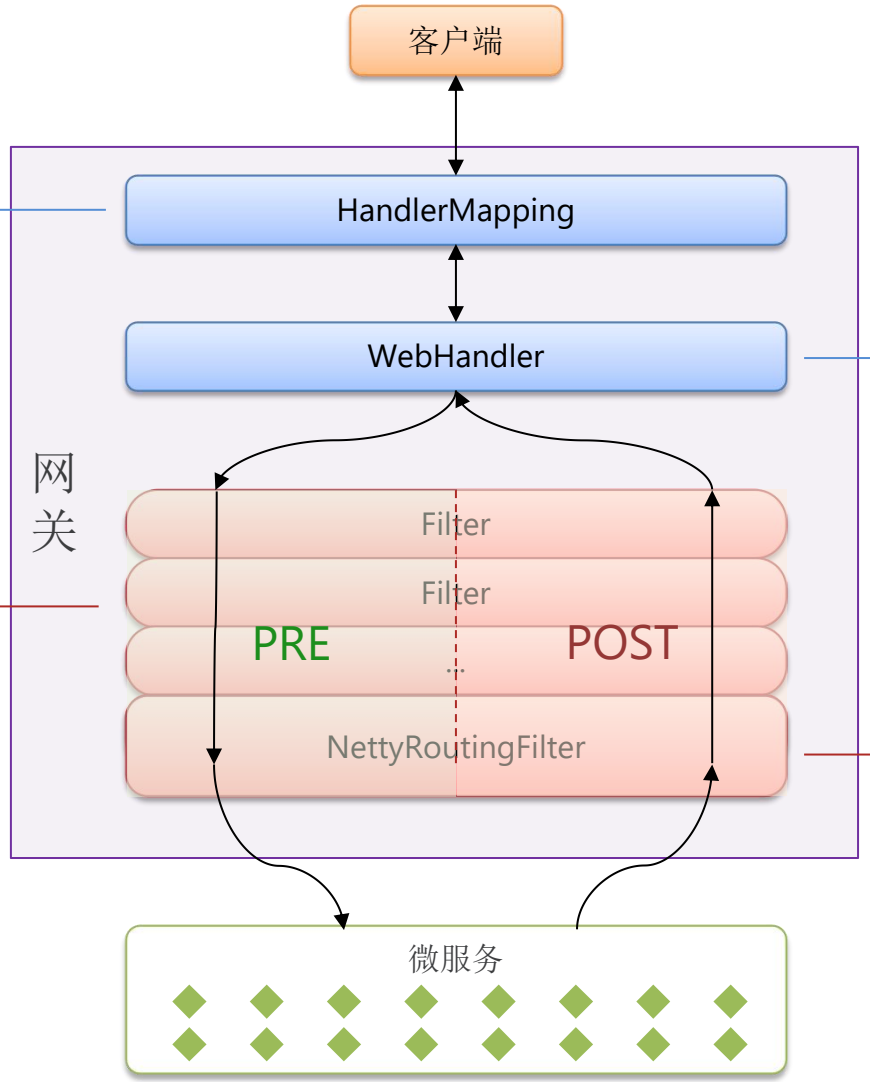
网关登录校验

网关登录校验

- 如何在网关转发之前做登录校验？



网关请求处理流程



路由映射器

- HandlerMapping的默认实现是RoutePredicateHandlerMapping.
- HandlerMapping根据请求找到匹配的路由并存入上下文，然后把请求交给WebHandler处理。

请求处理器

默认实现是FilteringWebHandler，顾名思义是一个过滤器处理器。它会加载网关中配置的多个过滤器，放入集合并排序，形成**过滤器链**。然后依次执行这些过滤器。

过滤器

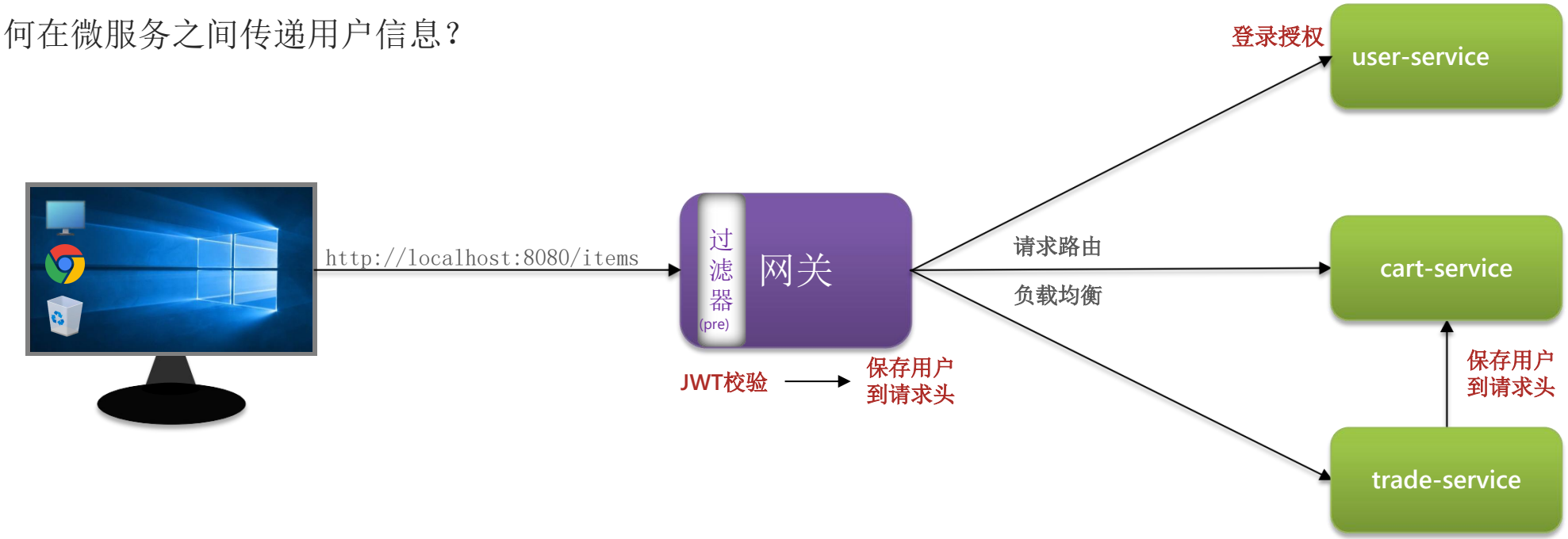
- 过滤器内部可以包含两部分逻辑，分别是**pre**和**post**，分别会在请求路由到微服务**之前**和**之后**执行。
- 当所有Filter的pre逻辑都依次**顺序**执行通过后，请求才会被路由到微服务，否则会被拦截，后续过滤器不再执行。
- 微服务返回结果后，再**倒序**执行Filter的post逻辑

Netty路由过滤器

负责将请求转发到微服务，当微服务返回结果后存入上下文

网关登录校验

- 如何在网关转发之前做登录校验？
- 网关如何将用户信息传递给微服务？
- 如何在微服务之间传递用户信息？



目录

Contents

自定义过滤器

网关过滤器有两种，分别是：

- GatewayFilter：路由过滤器，作用于任意指定的路由；默认不生效，要配置到路由后生效。
- GlobalFilter：全局过滤器，作用范围是所有路由；声明后自动生效。

两种过滤器的**过滤方法**签名完全一致：

```java
public interface GatewayFilter extends ShortcutConfigurable {

    String NAME_KEY = "name";
    String VALUE_KEY = "value";

    /**
     * Process the Web request and (optionally) delegate to the next
     * {@code WebFilter} through the given {@link GatewayFilterChain}.
     * @param exchange the current server exchange
     * @param chain provides a way to delegate to the next filter
     * @return Mono<Void> to indicate when request processing is complete
     */
    Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain);

}
```

```java
public interface GlobalFilter {

    /**
     * Process the Web request and (optionally) delegate to the
     * next {@code WebFilter} through the given GatewayFilterChain.
     * @param exchange the current server exchange
     * @param chain provides a way to delegate to the next filter
     * @return Mono<Void> to indicate when request processing is complete
     */
    Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain);

}
```

自定义过滤器

网关过滤器有两种，分别是：

- GatewayFilter：路由过滤器，作用于任意指定的路由；默认不生效，要配置到路由后生效。
- GlobalFilter：全局过滤器，作用范围是所有路由；声明后自动生效。

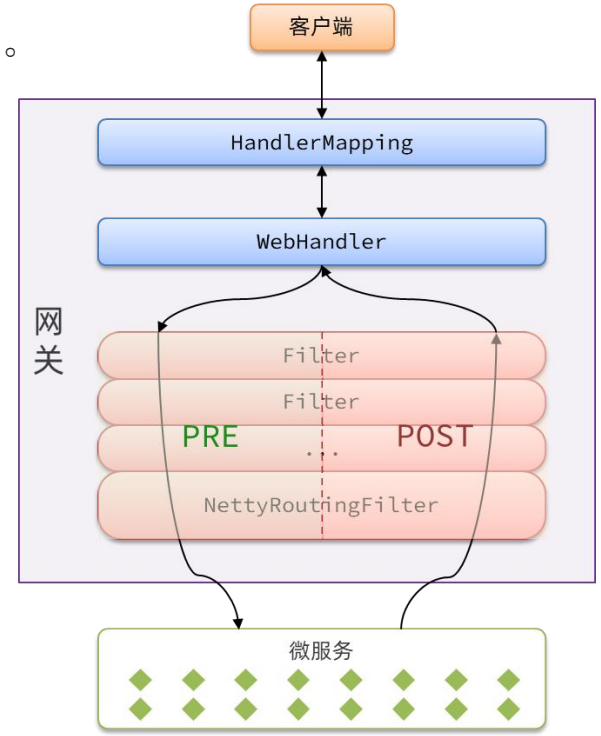
```
public interface GlobalFilter {

    /**
     * Process the Web request and (optionally) delegate to the
     * next {@code WebFilter} through the given GatewayFilterChain.
     * @param exchange the current server exchange
     * @param chain provides a way to delegate to the next filter
     * @return Mono<Void> to indicate when request processing is complete
     */
    Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain);

}
```

请求上下文，包含整个过滤器链内共享数据，例如request、response等

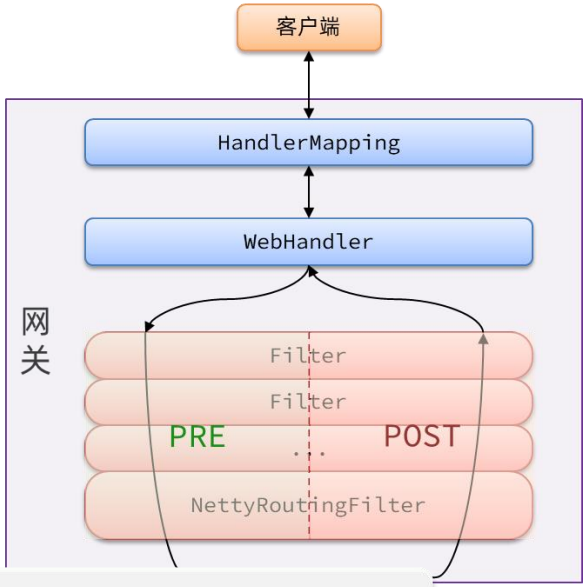
过滤器链。
当前过滤器执行完后，要调用过滤器链中的下一个过滤器



自定义过滤器GlobalFilter

自定义GlobalFilter比较简单，直接实现GlobalFilter接口即可：

```
@Component
public class MyGlobalFilter implements                GlobalFilter      , Ordered
{
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        // 1.获取请求
        ServerHttpRequest request = exchange.getRequest();
        // 2.过滤器业务处理
        System.out.println("GlobalFilter pre阶段 执行了。");
        // 3.放行
        return chain.filter(exchange);
    }
    @Override
    public int getOrder() {
        // 过滤器执行顺序，值越小，优先级越高
                return 0;
    }
}
```



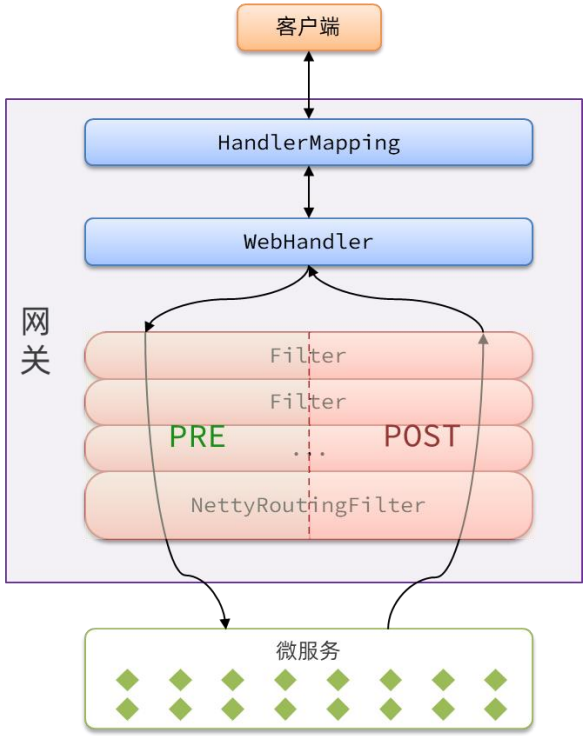

```
spring:
  cloud:
    gateway:
      default-filters:
        - AddRequestHeader=a,b
```

自定义过滤器

自定义GlobalFilter比较简单，直接实现GlobalFilter接口即可：

```
@Component
public class MyGlobalFilter implements                GlobalFilter      , Ordered
{
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        System.out.println("pre阶段 执行了。");
        return chain.filter(exchange)                          ;
                                .then(Mono.fromRunnable(() -> {
                                        System.out.println("post阶段 执行了。");
                                    }))
    }
    @Override
    public int getOrder() {
        //过滤器执行顺序，值越小，优先级越高
                return 0;
    }

}
```



自定义过滤器GatewayFilter

自定义GatewayFilter不是直接实现GatewayFilter，而是实现AbstractGatewayFilterFactory，示例如下：

固定的类名称后缀，方便配置使用

```
@Component
public class PrintAnyGatewayFilterFactory extends AbstractGatewayFilterFactory<      > {                    Object
    @Override
    public GatewayFilter apply(      config) {Object
        return new GatewayFilter() {
            @Override
            public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
                //编写过滤器逻辑
                        System.out.println("PrintAny filter 执行了");
                //放行
                        return chain.filter(exchange);
            }
        };
    }

}
```

```
spring:
  cloud:
    gateway:
      default-filters:
        - Ad  - PrintAny      =1,2,3
```


```java
@Component
public class PrintAnyGatewayFilterFactory extends AbstractGatewayFilterFactory<      > {
    @Override
    public GatewayFilter apply(      config) {
        return new GatewayFilter() {
            @Override
            public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
                //编写过滤器逻辑
                        System.out.println("PrintAny filter 执行了");
                //放行
                        return chain.filter(exchange);

            }
        };
    }
}    //自定义配置属性，成员变量名称很重要，下面会用到
    @Data
    public static class Config{
        private String a;
        private String b;
        private String c;
    }

    //将变量名称依次返回，顺序很重要，将来读取参数时需要按顺序获取
    @Override
    public List<String> shortcutFieldOrder() {
        return List.of("a", "b", "c");
    }

    //将Config字节码传递给父类，父类负责帮我们读取yaml配置
    public PrintAnyGatewayFilterFactory() {
        super(Config.class);
    }
```

```yaml
spring:
  cloud:
    gateway:
      default-filters:
        - Ad  - PrintAny      =1,2,3
```

自定义过滤器GlobalFilter

自定义GlobalFilter比较简单，直接实现GlobalFilter接口即可：

```
@Component
public class MyGlobalFilter implements                GlobalFilter      , Ordered
{
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        // 1.获取请求
        ServerHttpRequest request = exchange.getRequest();
        // 2.过滤器业务处理
        System.out.println("GlobalFilter pre阶段 执行了。");
        // 3.放行
        return chain.filter(exchange);
    }
    @Override
    public int getOrder() {
        // 过滤器执行顺序，值越小，优先级越高
                return 0;
    }
}
```

目录

Contents

案例

实现登录校验

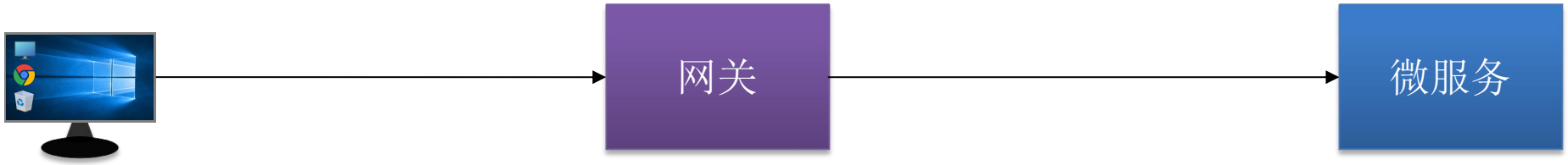
需求：在网关中基于过滤器实现登录校验功能

提示：黑马商城是基于JWT实现的登录校验，目前相关功能在hm-service模块。我们可以将其中的JWT工具拷贝到gateway模块，然后基于GlobalFilter来实现登录校验。

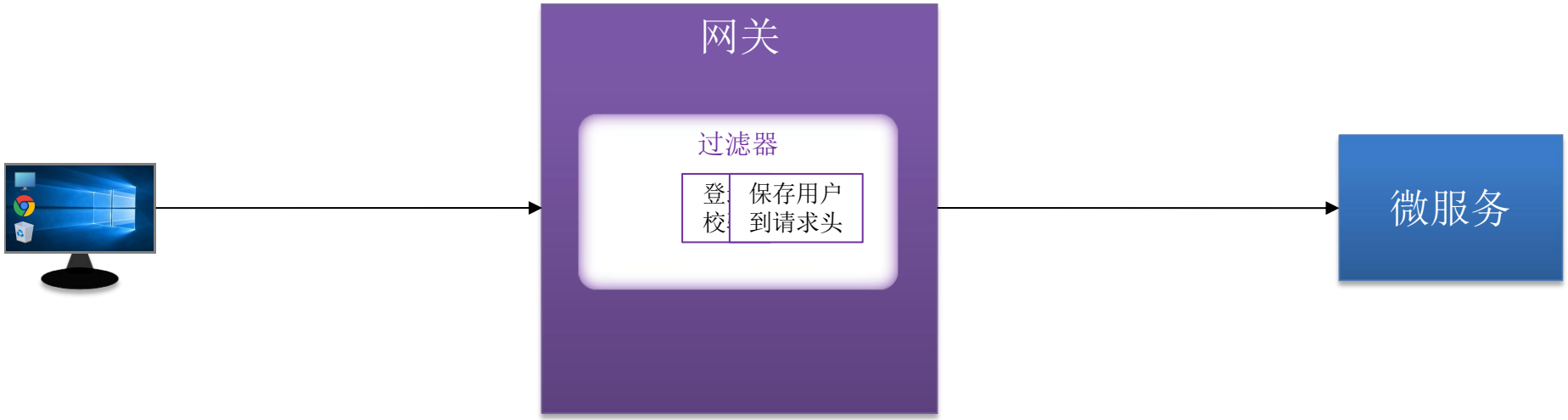
目录

Contents

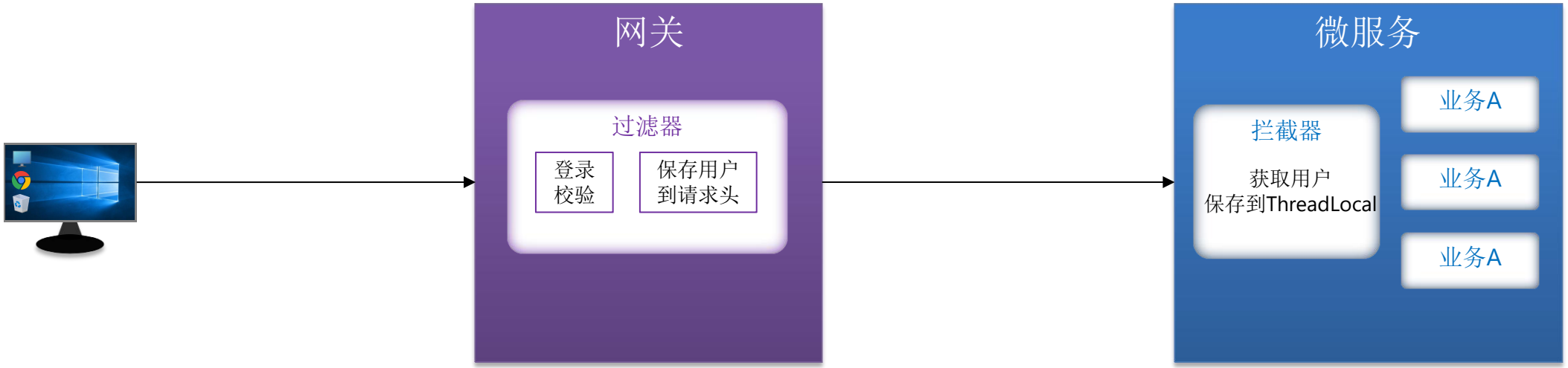
网关传递用户



网关传递用户



网关传递用户


网关
过滤器
登录
校验
保存用户
到请求头
微服务
拦截器
获取用户
保存到ThreadLocal
业务A
业务A
业务A

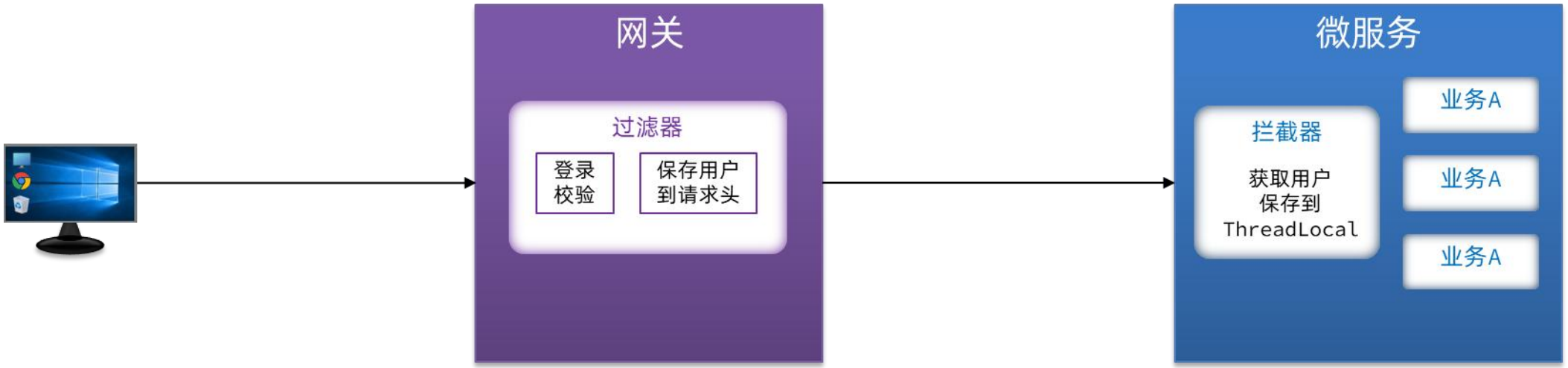
步骤

一、在网关的登录校验过滤器中，把获取到的用户写入请求头

需求：修改gateway模块中的登录校验拦截器，在校验成功后保存用户到下游请求的请求头中。

提示：要修改转发到微服务的请求，需要用到ServerWebExchange类提供的API，示例如下：

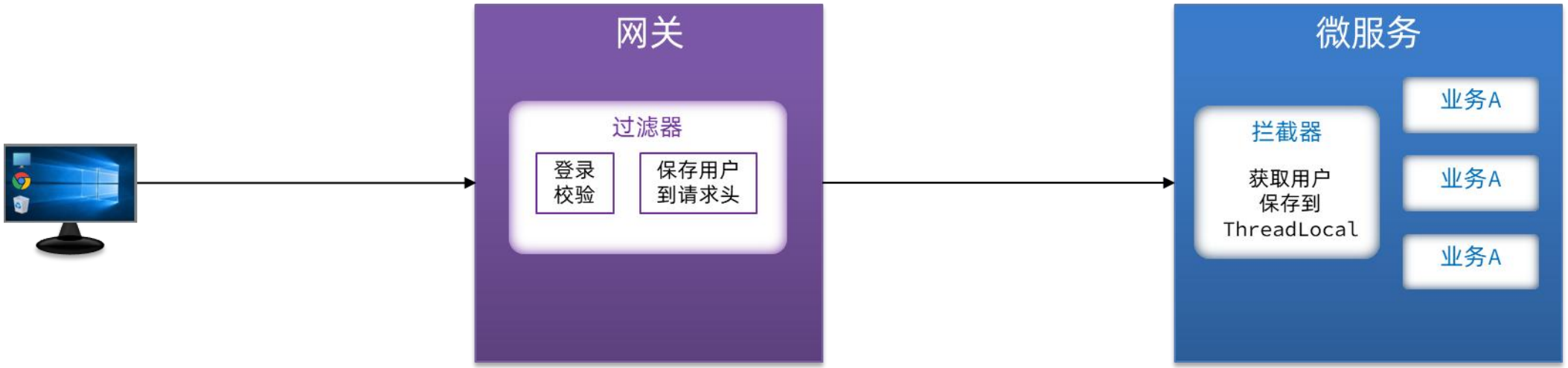
```
exchange.mutate() // mutate就是对下游请求做更改
        .request(builder -> builder.header("user-info", userInfo))
    .build();
```



步骤

二、在hm-common中编写SpringMVC拦截器，获取登录用户

需求：由于每个微服务都可能有获取登录用户的需求，因此我们直接在hm-common模块定义拦截器，这样微服务只需要引入依赖即可生效，无需重复编写。

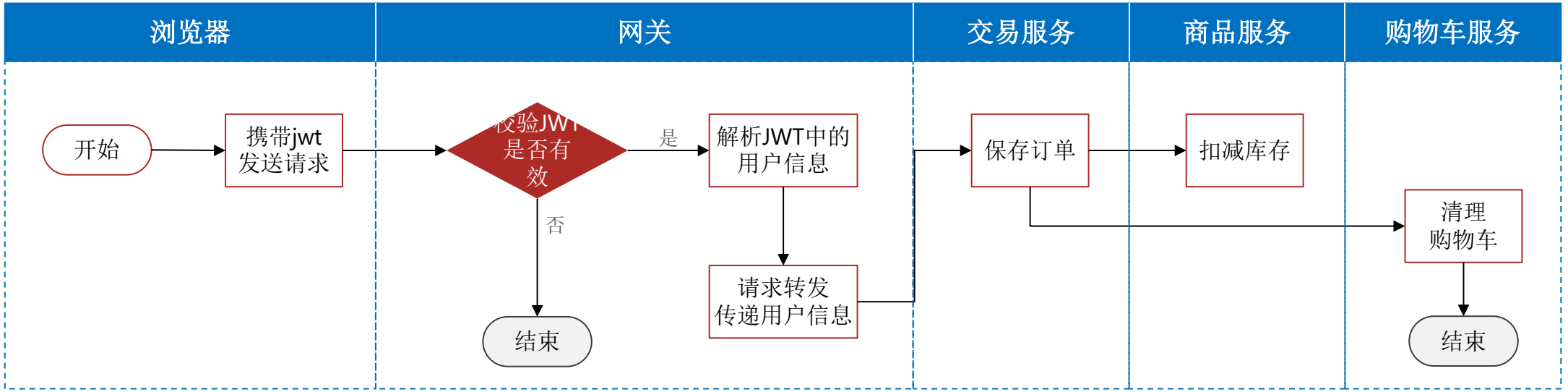


目录

Contents

OpenFeign传递用户

微服务项目中的很多业务要多个微服务共同合作完成，而这个过程中也需要传递登录用户信息，例如：



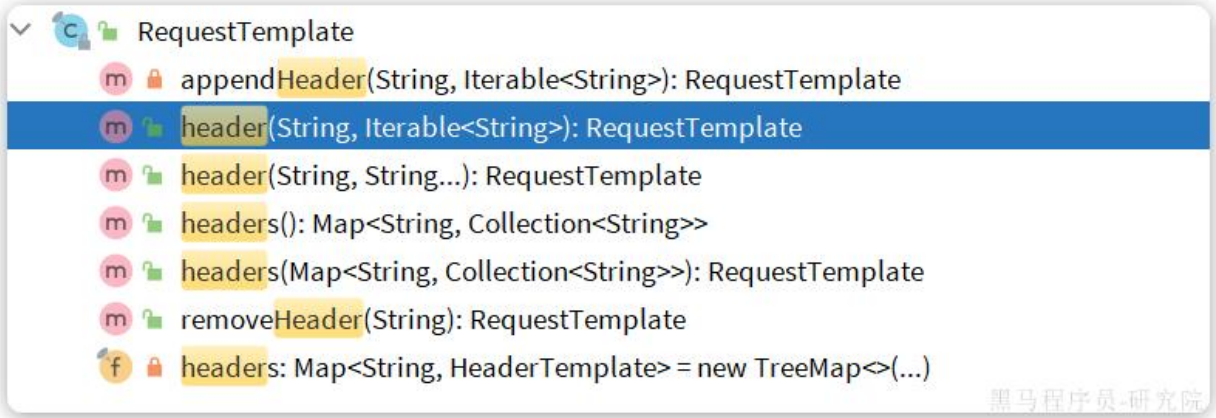
OpenFeign传递用户

OpenFeign中提供了一个拦截器接口，所有由OpenFeign发起的请求都会先调用拦截器处理请求：

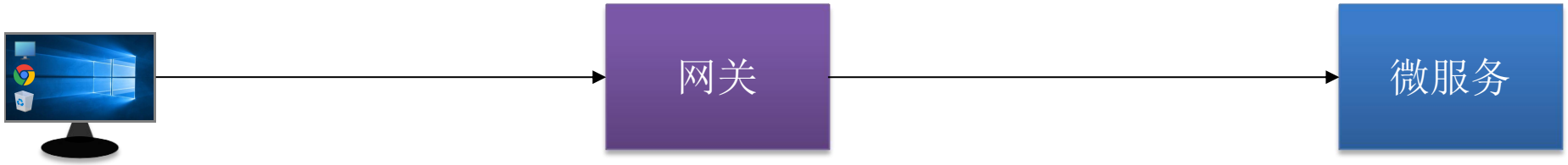
```
public interface RequestInterceptor {

  /**
   * Called for every request. Add data using methods on the supplied {@link RequestTemplate}.
   */
  void apply(RequestTemplate template);
}
```

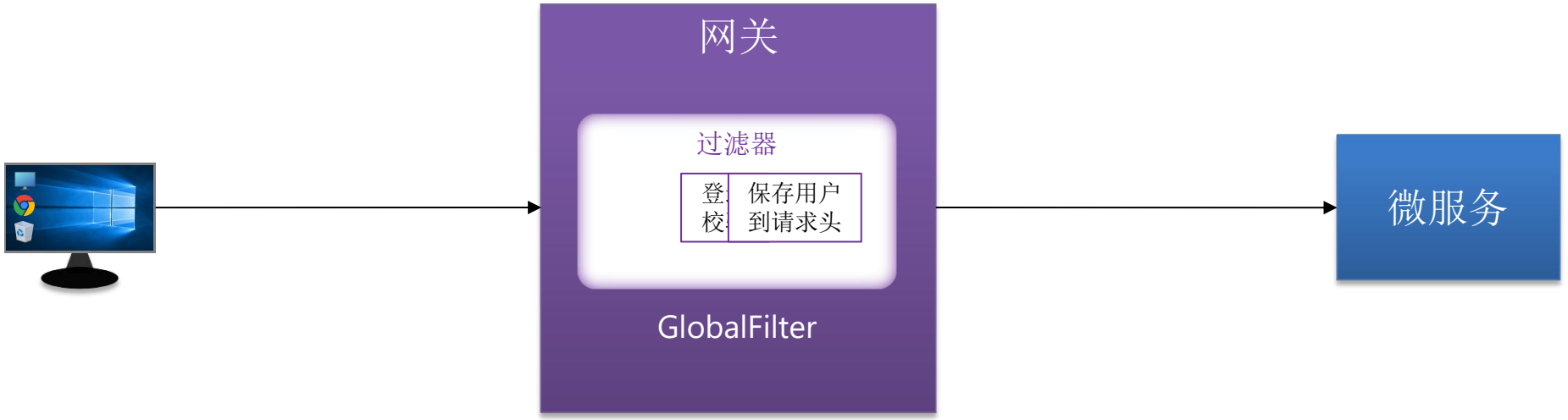
其中的RequestTemplate类中提供了一些方法可以让我们修改请求头：



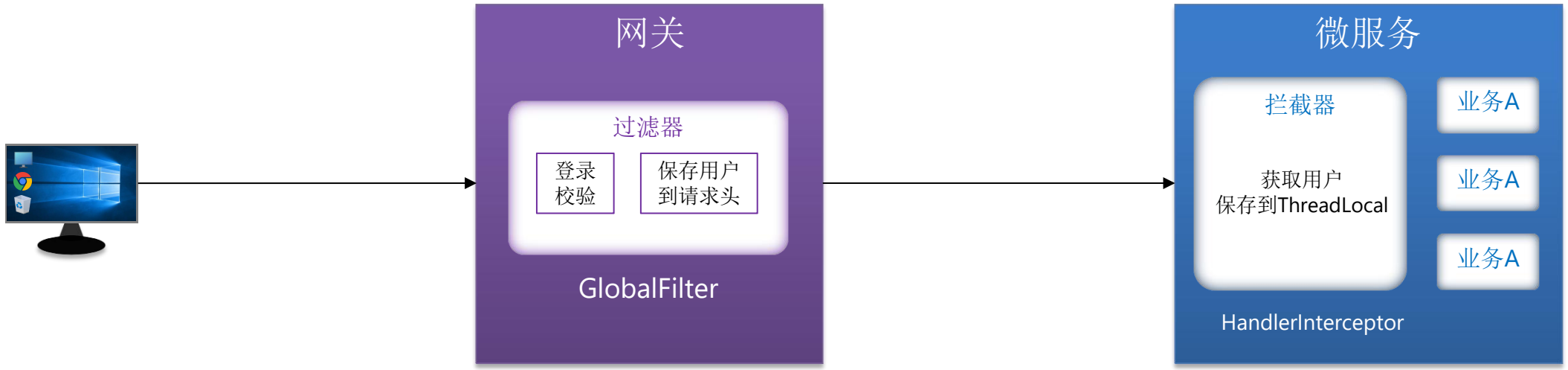
微服务登录解决方案



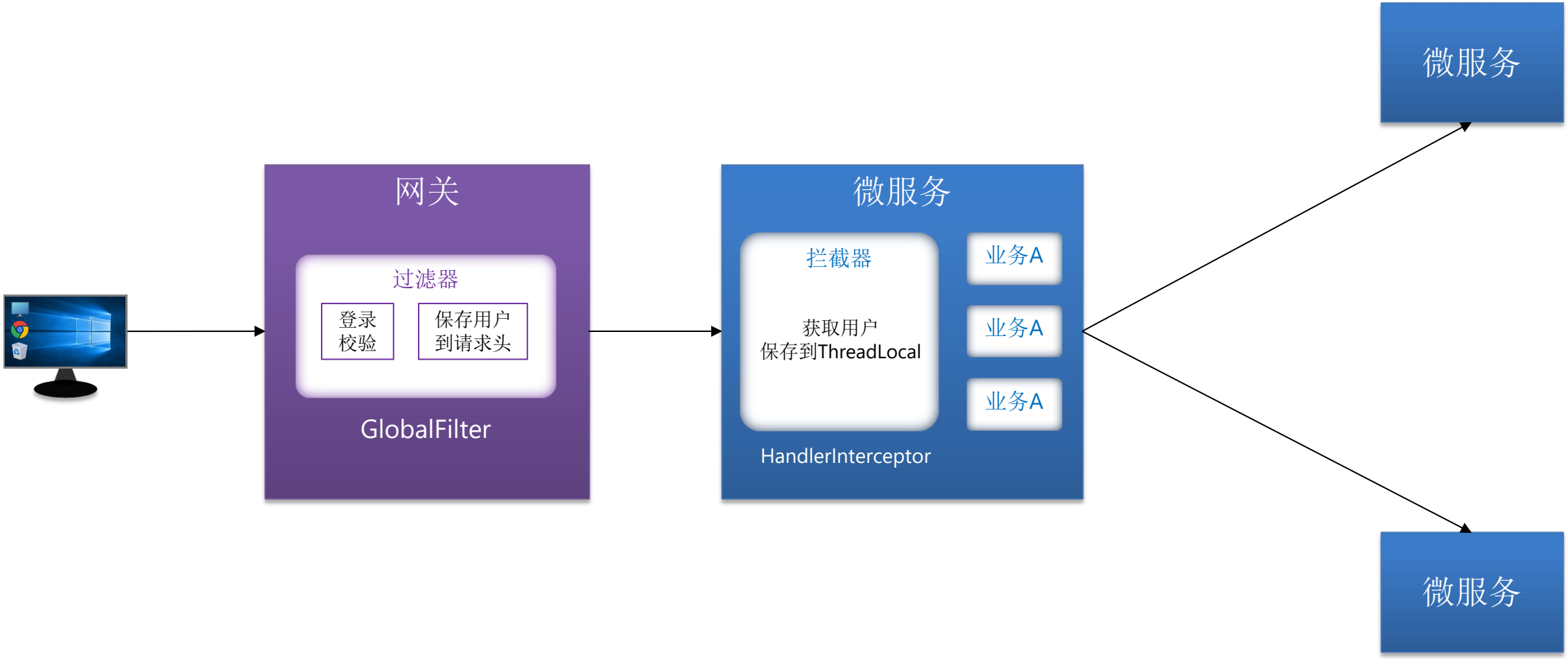
微服务登录解决方案



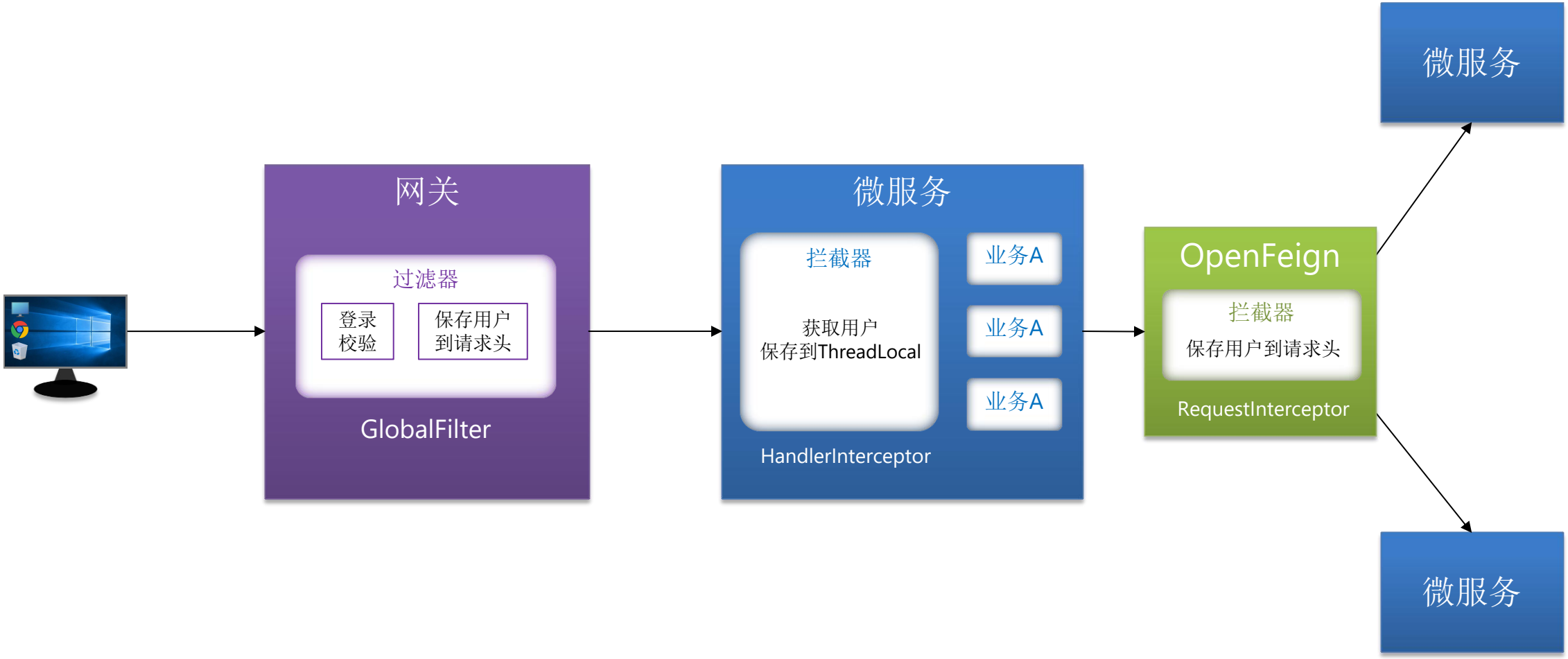
微服务登录解决方案


网关
过滤器
登录
校验
保存用户
到请求头
GlobalFilter
微服务
拦截器
获取用户
保存到ThreadLocal
业务A
业务A
业务A
HandlerInterceptor

微服务登录解决方案


微服务
网关
过滤器
登录
校验
保存用户
到请求头
GlobalFilter
微服务
拦截器
获取用户
保存到ThreadLocal
业务A
业务A
业务A
HandlerInterceptor
微服务

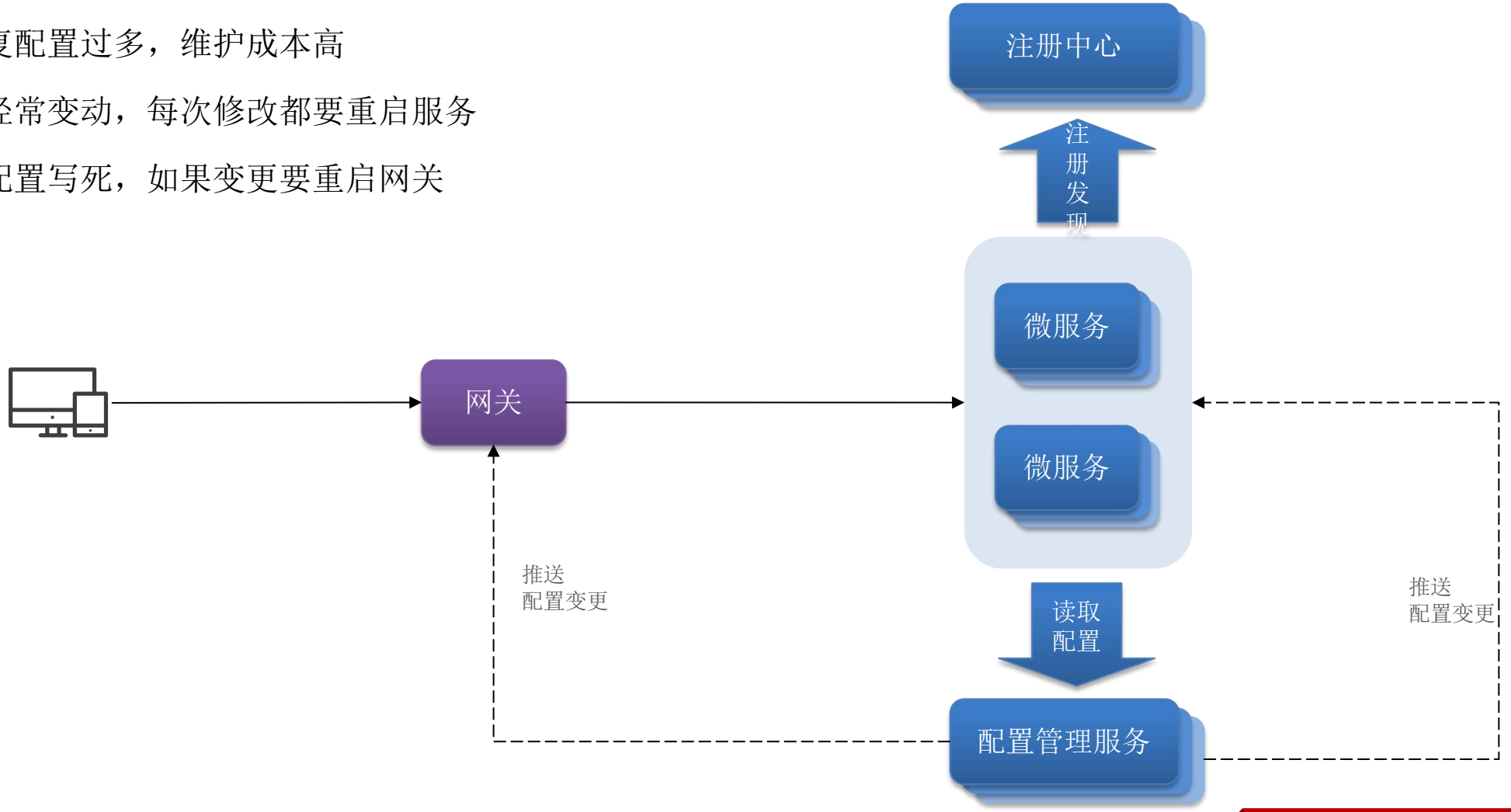
微服务登录解决方案


网关
过滤器
登录
校验
保存用户
到请求头
GlobalFilter
微服务
拦截器
获取用户
保存到ThreadLocal
HandlerInterceptor
业务A
业务A
业务A
OpenFeign
拦截器
保存用户到请求头
RequestInterceptor
微服务
微服务

03 配置管理

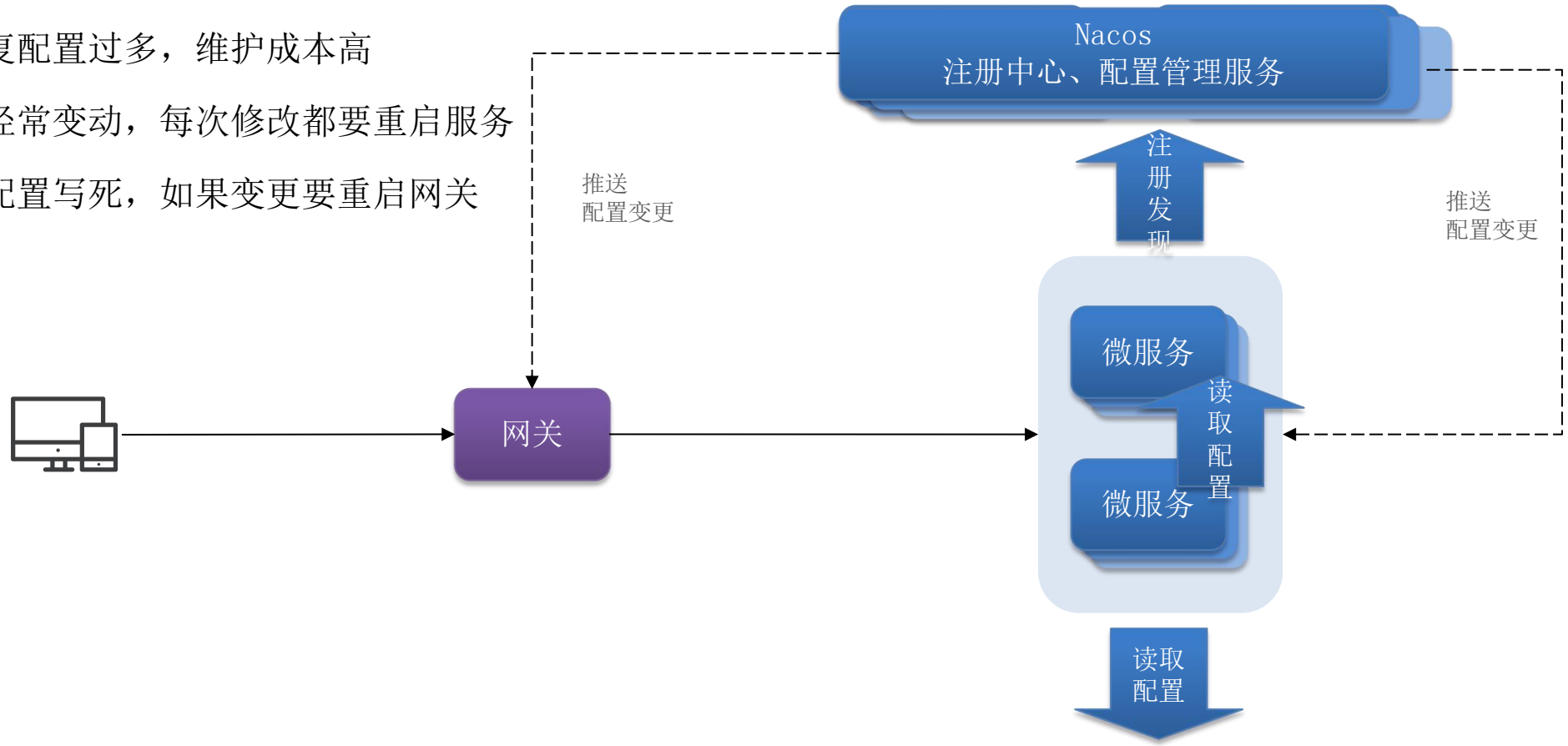
配置管理

- 微服务重复配置过多，维护成本高
- 业务配置经常变动，每次修改都要重启服务
- 网关路由配置写死，如果变更要重启网关



配置管理

- 微服务重复配置过多，维护成本高
- 业务配置经常变动，每次修改都要重启服务
- 网关路由配置写死，如果变更要重启网关



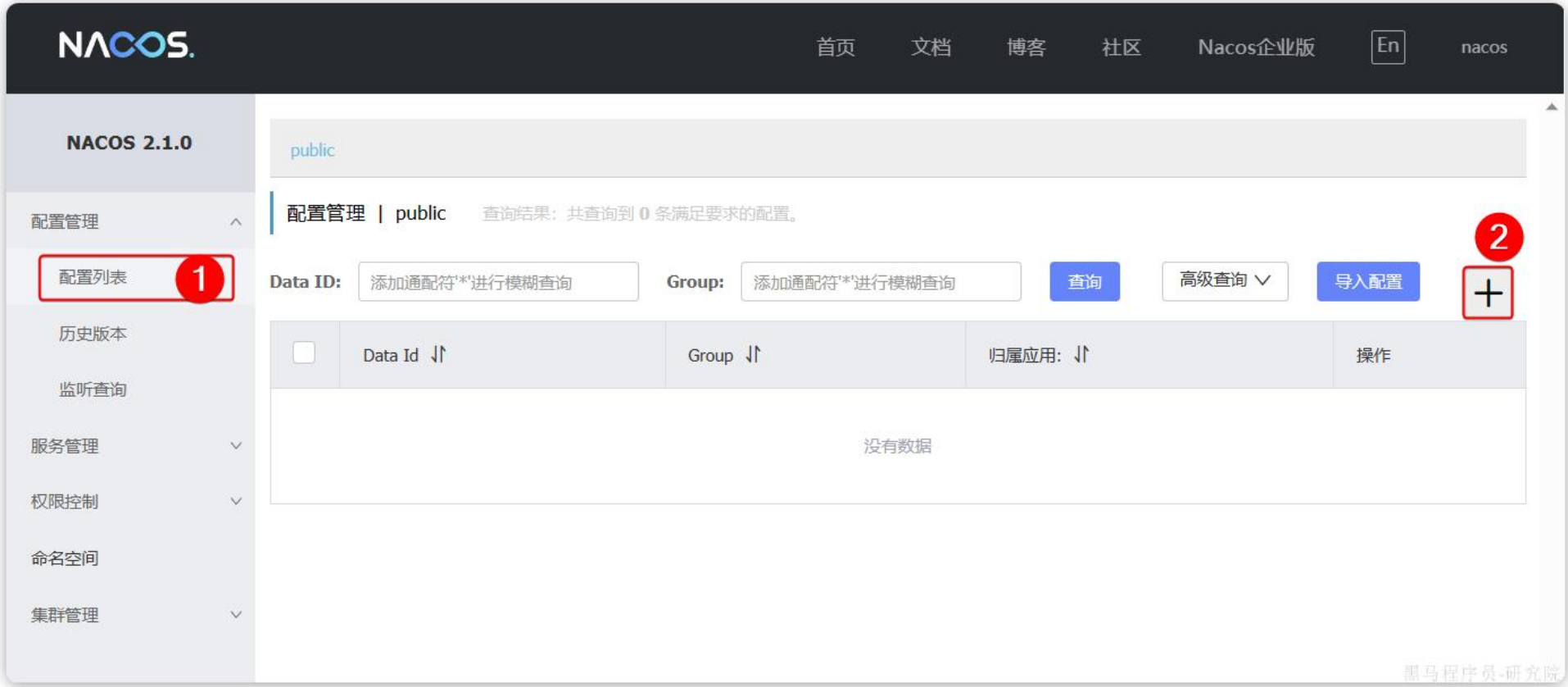
目录

Contents

步骤

一．添加配置到Nacos

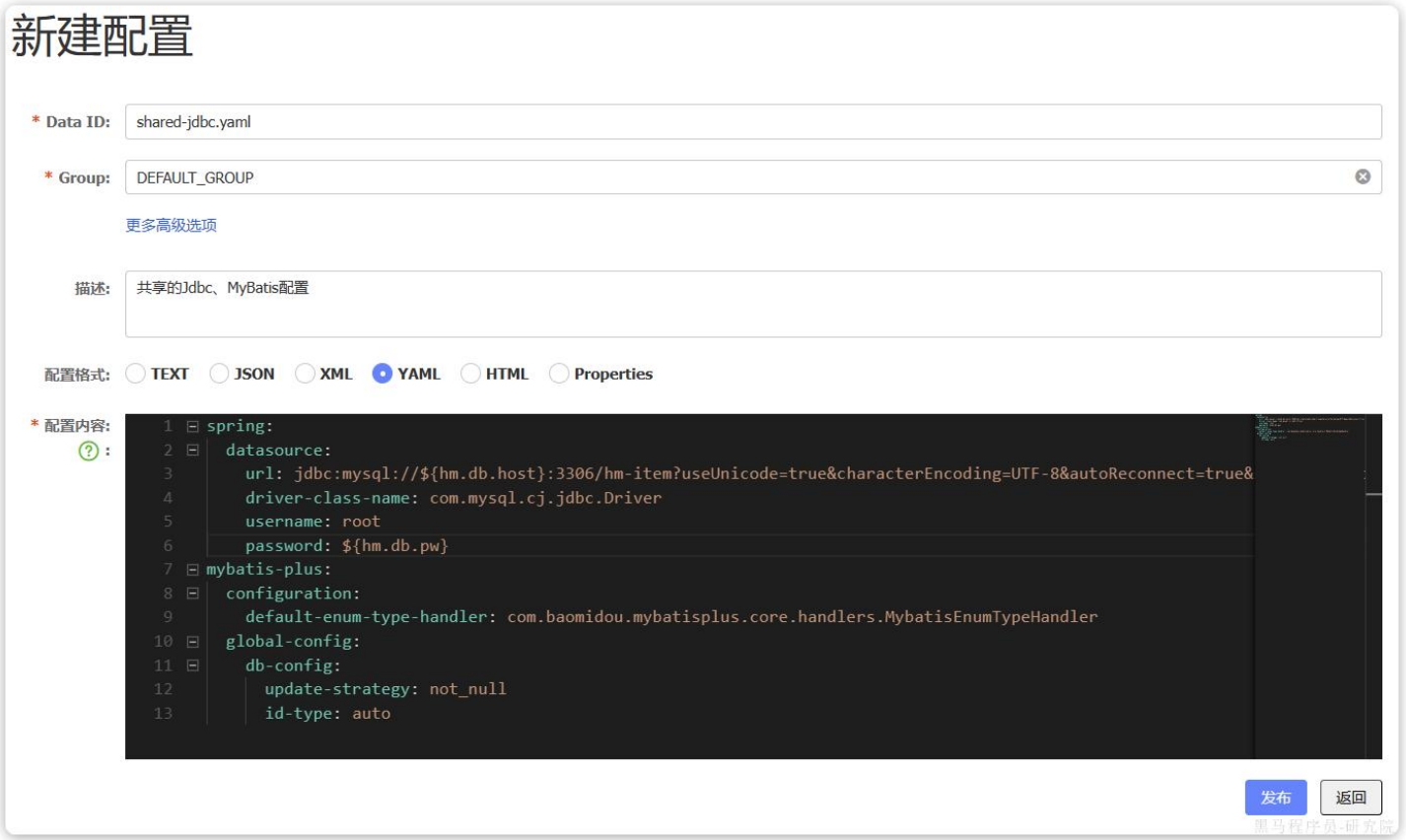
添加一些共享配置到Nacos中，包括：Jdbc、MybatisPlus、日志、Swagger、OpenFeign等配置



步骤

一. 添加配置到Nacos

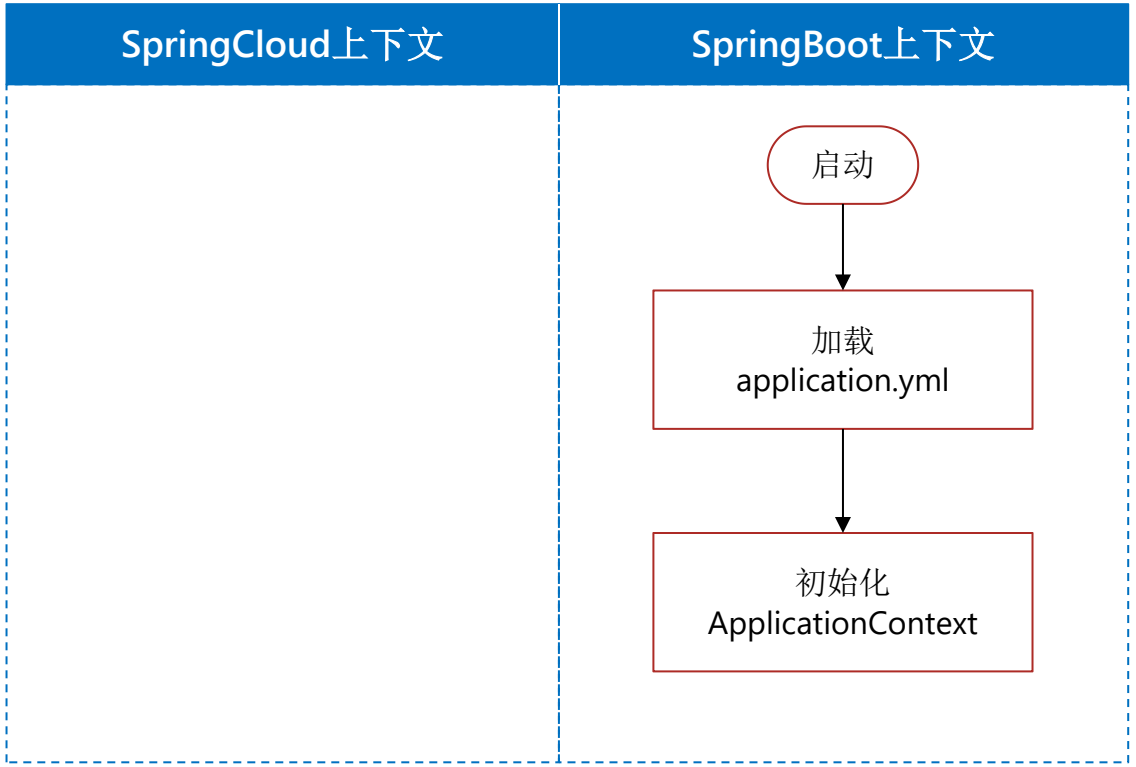
添加一些共享配置到Nacos中，包括：Jdbc、MybatisPlus、日志、Swagger、OpenFeign等配置



步骤

二．拉取共享配置

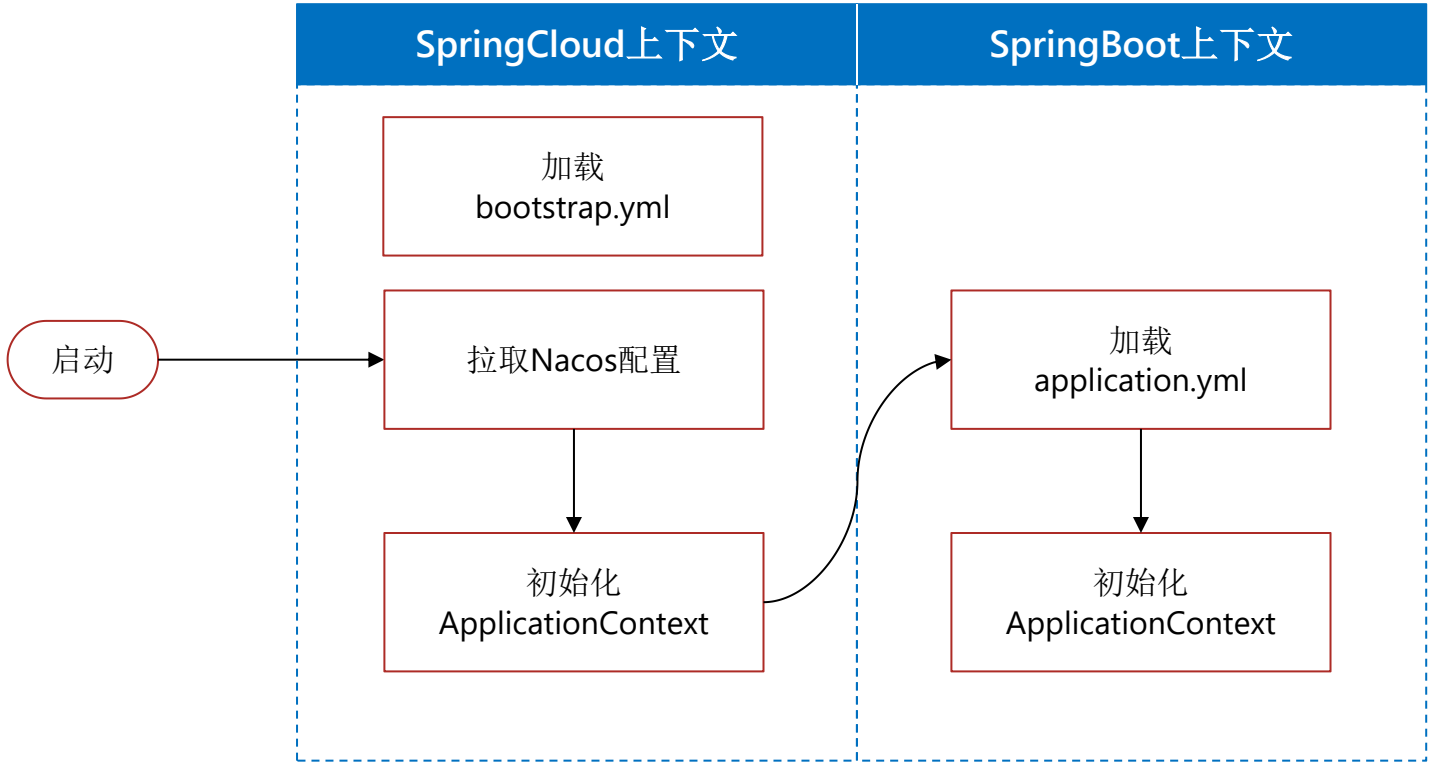
基于NacosConfig拉取共享配置代替微服务的本地配置。



步骤

二．拉取共享配置

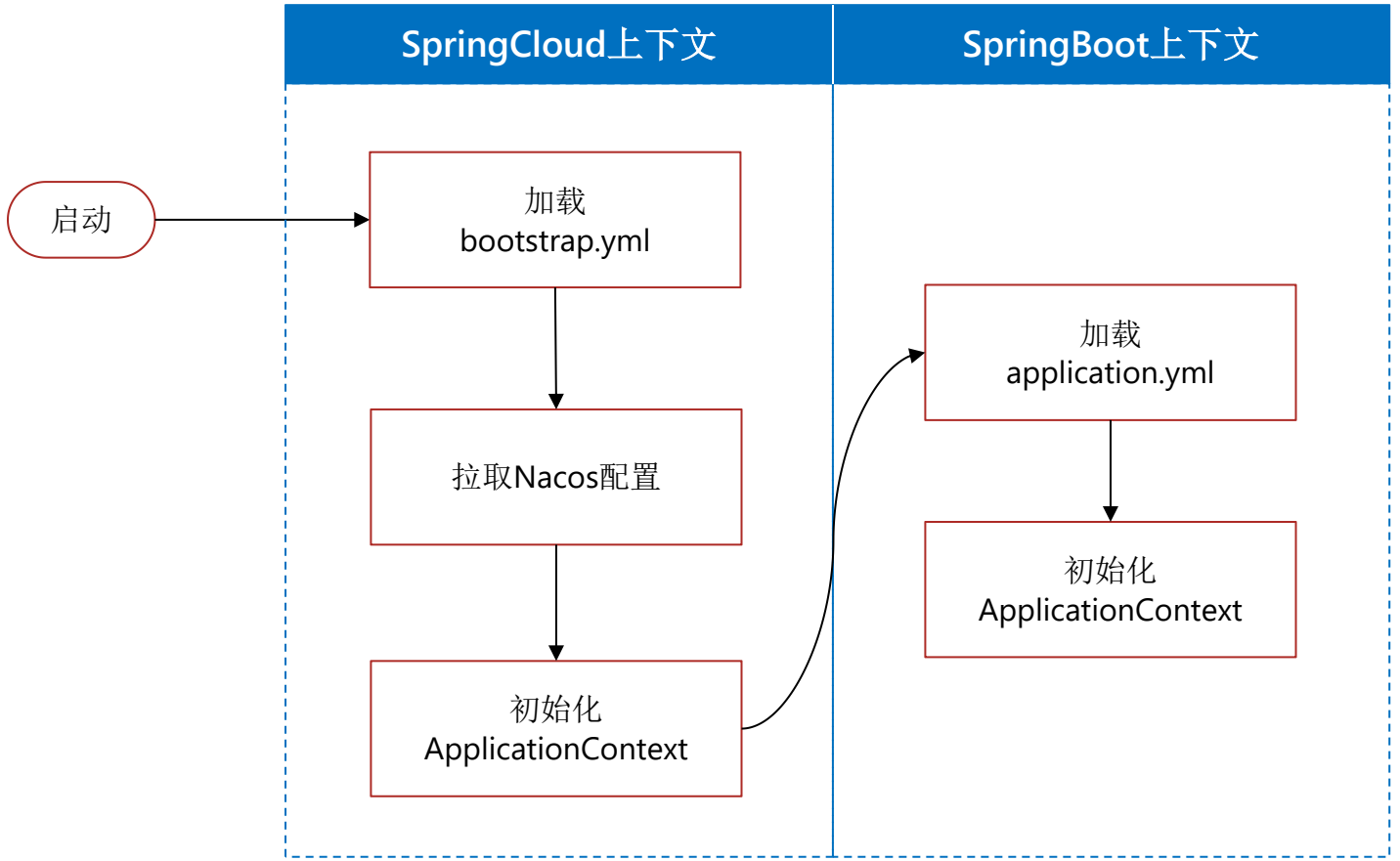
基于NacosConfig拉取共享配置代替微服务的本地配置。



步骤

二．拉取共享配置

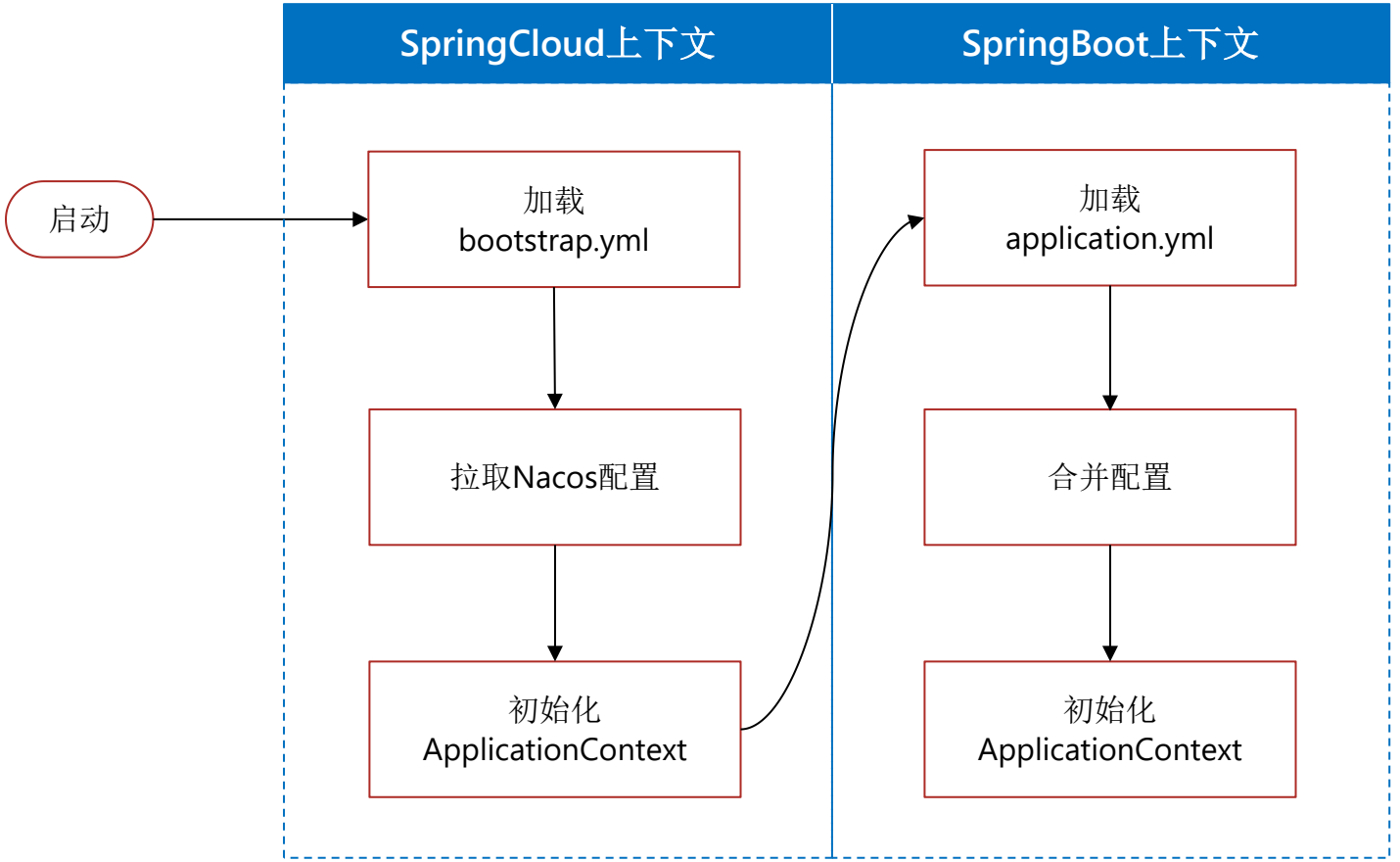
基于NacosConfig拉取共享配置代替微服务的本地配置。



步骤

二．拉取共享配置

基于NacosConfig拉取共享配置代替微服务的本地配置。



步骤

二．拉取共享配置

基于NacosConfig拉取共享配置代替微服务的本地配置。

① 引入依赖

```
<!--nacos配置管理-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
<!--读取bootstrap文件-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bootstrap</artifactId>
</dependency>
```

步骤

二．拉取共享配置

基于NacosConfig拉取共享配置代替微服务的本地配置。

② 新建bootstrap.yaml

```yaml
spring:
  application:
    name: cart-service # 服务名称
    profiles:
    active: dev
  cloud:
    nacos:
      server-addr: 192.168.150.101:8848 # nacos地址
            config:
        file-extension: yaml # 文件后缀名
                shared-configs: # 共享配置
                    - dataId: shared-jdbc.yaml # 共享mybatis配置
                    - dataId: shared-log.yaml # 共享日志配置
                    - dataId: shared-swagger.yaml # 共享日志配置
```

目录

Contents

配置热更新

配置热更新：当修改配置文件中的配置时，微服务无需重启即可使配置生效。

配置热更新

配置热更新：当修改配置文件中的配置时，微服务无需重启即可使配置生效。

前提条件：

① nacos中要有一个与微服务名有关的配置文件。

② 微服务中要以特定方式读取需要热更新的配置属性

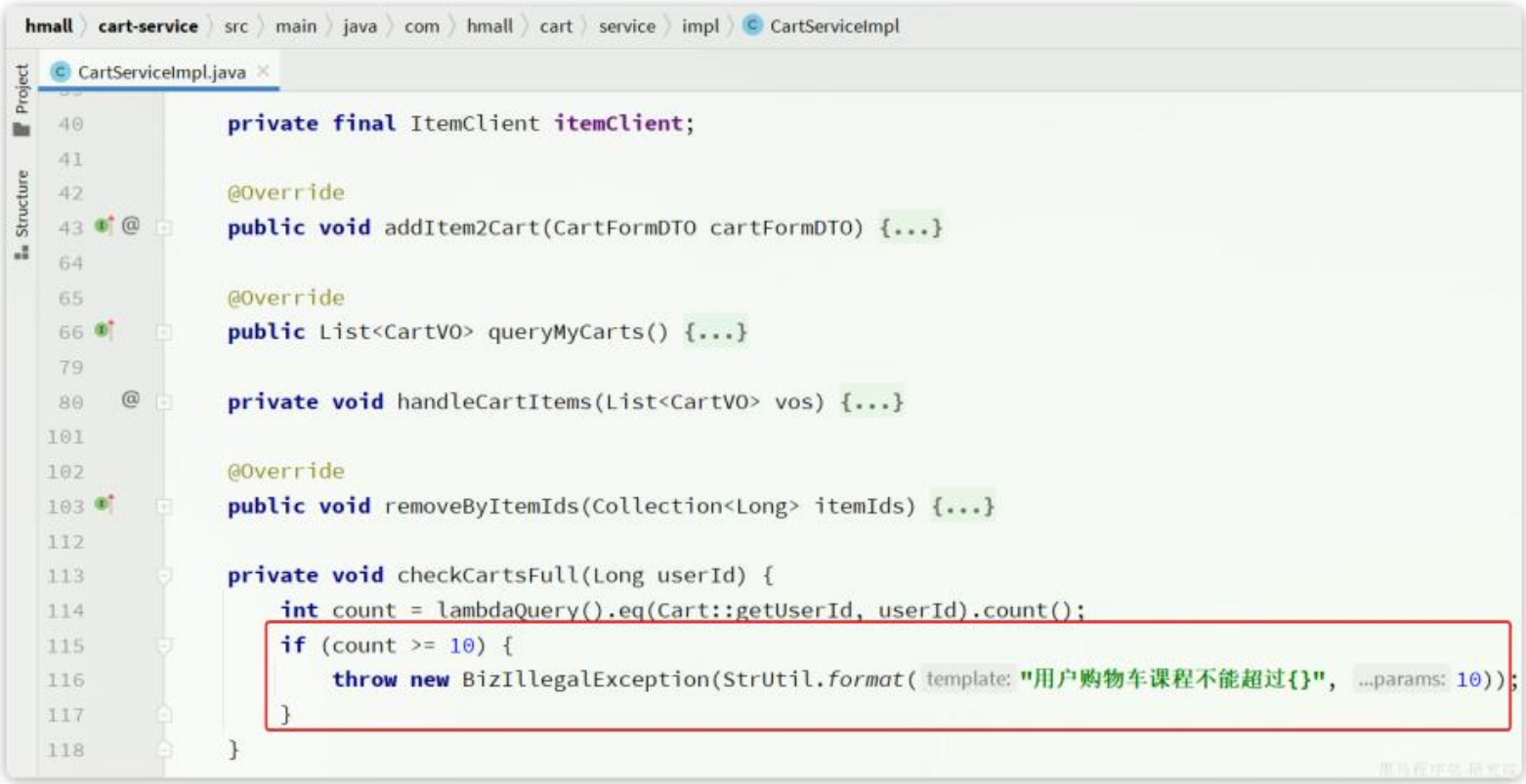
```java
@Data
@ConfigurationProperties(prefix = "hm.cart")
public class CartProperties {
    private int maxItems;
}
```

```java
@Data
@RefreshScope
public class CartProperties {
    @Value("${hm.cart.maxItems}")
    private int maxItems;
}
```

案例

实现购物车添加商品上限的配置热更新

需求：购物车的限定数量目前是写死在业务中的，将其改为读取配置文件属性，并将配置交给Nacos管理，实现热更新。



配置热更新

配置热更新：当修改配置文件中的配置时，微服务无需重启即可使配置生效。

前提条件：

① nacos中要有一个与微服务名有关的配置文件，dataId的格式如下：

② 微服务中要以特定方式读取需要热更新的配置属性

```java
@Data
@ConfigurationProperties(prefix = "hm.cart")
public class CartProperties {
    private int maxItems;
}
```

```java
@Data
@RefreshScope
public class CartProperties {
    @Value("${hm.cart.maxItems}")
    private int maxItems;
}
```

目录

Contents

动态路由

要实现**动态路由**首先要将路由配置保存到Nacos，当Nacos中的路由配置变更时，推送最新配置到网关，实时更新网关中的路由信息。

我们需要完成两件事情：

① 监听Nacos配置变更的消息

② 当配置变更时，将最新的路由信息更新到网关路由表

动态路由 - 监听Nacos配置

监听Nacos配置变更可以参考官方文档： https://nacos.io/zh-cn/docs/sdk.html

```
private final NacosConfigManager nacosConfigManager;

public void initRouteConfigListener() throws NacosException {
    // 1.注册监听器并首次拉取配置
        String configInfo = nacosConfigManager.getConfigService()
            .getConfigAndSignListener(dataId, group, 5000, new Listener() {
                @Override
                public Executor getExecutor() {
                    return null;
                }
                @Override
                public void receiveConfigInfo(String configInfo) {
                    // TODO 监听到配置变更，更新一次配置

                }
            });
    // TODO 2.首次启动时，更新一次配置
}
```

动态路由 - 更新路由表

监听到路由信息后，可以利用**RouteDefinitionWriter**来更新路由表：

```
/**
 * @author Spencer Gibb
 */
public interface RouteDefinitionWriter {
    /**
     * 更新路由到路由表，如果路由id重复，则会覆盖旧的路由
         */
    Mono<Void> save(Mono<RouteDefinition> route);
    /**
     * 根据路由id删除某个路由
         */
    Mono<Void> delete(Mono<String> routeId);

}
```

动态路由 - 路由配置语法

为了方便解析从Nacos读取到的路由配置，推荐使用json格式的路由配置，模板如下：

```
{
    "id": "item",
    "uri": "lb://item-service",
    "predicates": [{
        "name": "Path",
        "args": {
            "_genkey_0":"/items/**",
            "_genkey_1":"/search/**"
        }
    }],
    "filters": []
}
```

```
spring:
  cloud:
    gateway:
      routes:
        - id: item
          uri: lb://item-service
          predicates:
            - Path=/items/**,/search/**
```