Astro 企业应用

开发指南

文档版本 01

发布日期 2025-01-09

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 特性开发指南

1.1 扩展拦截开发指南

1.1.1 devspore-horizon 介绍

扩展拦截功能依赖devspore-horizon组件实现。devspore-horizon是DevSpore开发框架提供的一种插件机制，用户可以在请求被处理前（Entry）和应答发送前（Exit）调用DevSpore预置的插件，也可以使用自己开发的插件，用于实现统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等功能。

常用概念

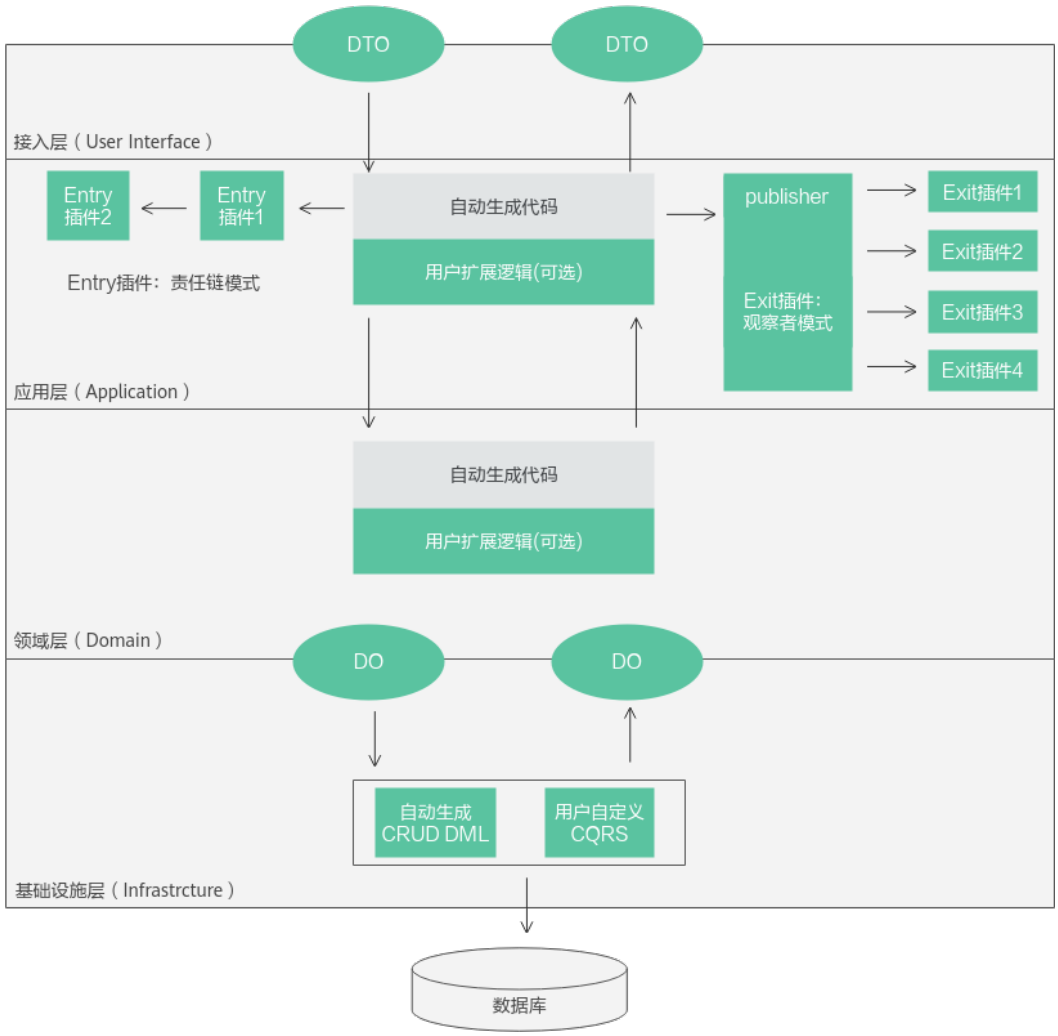
DTO：是一种数据传输对象，主要用于在不同层之间传输数据。

DO：是一种领域对象，用于表示业务领域中的实体或概念。

horizon 整体架构

devspore-horizon架构图如下：

图 1-1 架构图

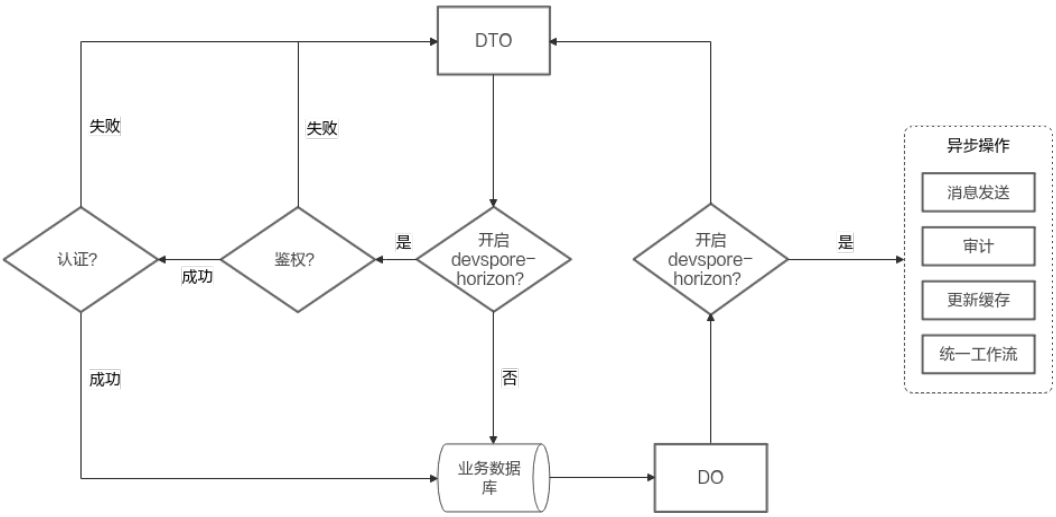


- Entry类插件采用的是责任链模式（**chain of resposibility**），所有的插件按照被注册的顺序依次调用，其中有一个插件抛出异常则整个请求返回失败。
- Exit类插件采用的是观察者模式（**Observer**），所有的插件作为观察者注册，DevSpore运行时库里的Publisher模块同时调用所有的观察者插件（插件可能被并发执行且执行的顺序随机），插件执行的结果不能影响请求返回。

实现流程

devspore-horizon整体流程概括如下：

图 1-2 实现流程

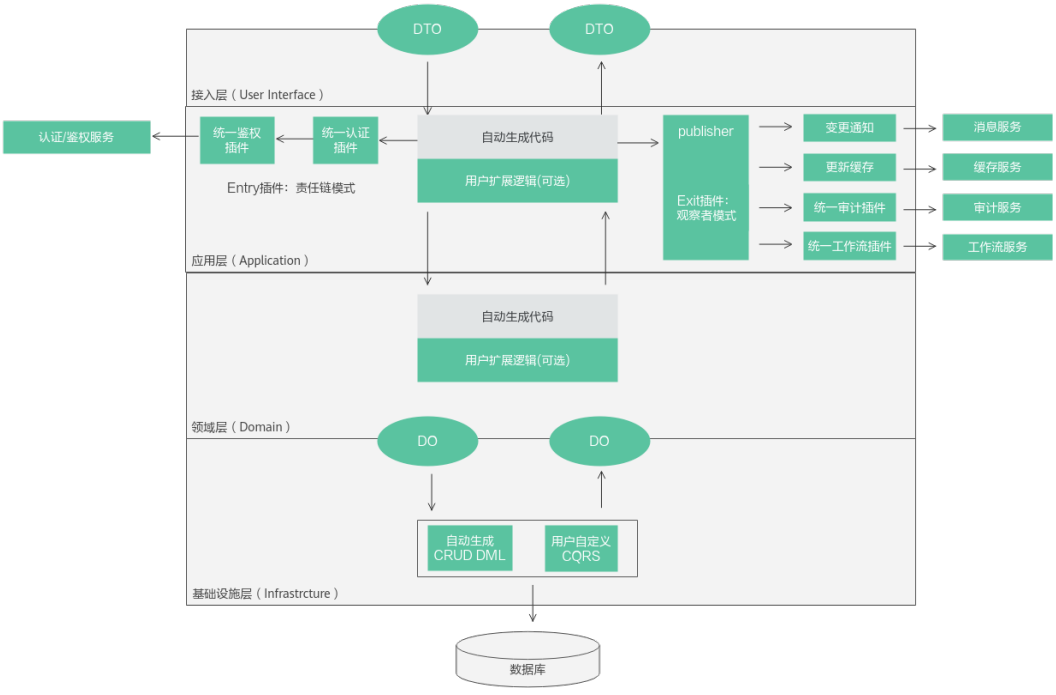


典型案例

devspore-horizon允许用户用内置的鉴权和消息发送插件，也支持用户自定义插件来自主实现鉴权（认证等）和消息发送（审计等）。

下图为一个典型的使用案例：

图 1-3 典型案例



如何实现 horizon 功能

只要实现了相应接口，填写配置信息即可实现horizon功能。

如图1-1中描述，horizon开放了两种插件**入口拦截Entry插件**和**出口拦截Exit插件**。其中统一认证和统一鉴权需要实现Entry插件；而变更通知、更新缓存、统一审计、统一工作流需要实现Exit插件。

1.1.2 拦截插件

入口插件

步骤1 实现Entry插件。

devspore-horizon提供了抽象类**Processor**，用户需要继承这个抽象类，实现抽象方法**doProcess(DataEvent event)**，把认证和鉴权的处理逻辑写到此方法体内。此方法返回值为**boolean型**。如果认证或者鉴权的逻辑失败，则返回**false**，责任链后续节点不再执行，整个请求返回不再执行；如果认证或者鉴权成功，则返回为**true**，继续执行后续节点。**Processor**代码如下：

```
public abstract class Processor {
    protected MetaDocument metaDoc;

    private String name;

    private Processor next;

    public Processor getNext() {
        return next;
    }

    public void setNext(Processor next) {
        this.next = next;
    }

    public boolean process(DataEvent dataEvent) throws PluginException {
        if (!doProcess(dataEvent)) {
            return false;
        }
        if (null != next) {
            return next.process(dataEvent);
        }
        return true;
    }

    public abstract boolean doProcess(DataEvent dataEvent) throws PluginException;

    public String getName() {
        return name;
    }
}
```

步骤2 填写配置信息。

- 完成接口实现之后，还需要填写配置信息。有关horizon的入口拦截配置信息如下：
  ```
  devspore.horizon.approvers
  ```
  实现的Entry插件，插件必须注入到spring IOC容器，并提供bean id，赋值给devspore.horizon.processors配置项。**devspore.horizon.processors**配置项是个数组。horizon会读取数组中每个bean id，执行已实现的鉴权、认证等逻辑。
- 多个插件以","分隔，多个插件按配置顺序执行。
  ```
  devspore:
    horizon:
      processors: bean1,bean2
  ```

----结束

出口插件

步骤1 实现exit插件。

devspore-horizon提供了接口Observer，用户需要实现这个接口的**observe(DataEvent dataEvent)**方法，把变更通知、更新缓存、统一审计、统一工作流的处理逻辑写到此方法体内即可。

Observer代码如下：

```
public interface Observer {
    // receive the BO changes
    void observe(DataEvent dataEvent) throws PluginException;
}
```

步骤2 填写配置信息。

- 完成接口实现之后，还需要填写配置信息。有关horizon的出口拦截配置信息如下：
  ```
  devspore.horizon.observers
  ```
  devspore.horizon.observers配置项也是个数组。horizon会读取数组中每个bean id，执行已实现的消息发送、审计等逻辑。
- 多个插件以","分隔，多个插件按配置顺序执行。
  ```
  devspore:
    horizon:
      observers: bean1,bean2
  ```

----结束

插件参数 DataEvent

DataEvent提供了插件的参数上下文，其主要包括以下信息：参数取值示例见**操作示例**

表 1-1 DataEvent 参数说明

参数名称	数据类型	描述
metaDocument	MetaDocument	服务的元数据对象。
operation	Operation	资源操作类型，包含增删改查、批量增删改查，自定义等操作。 **@Extension**注解operation参数值。
customMethod	String	扩展API中定义的method对象，仅在operation为CUSTOM时候有意义。 **@Extension**注解customMethod参数值。
sourceName	String	操作的资源名称。**@Extension**注解name参数值。
argsMap	Map	操作方法参数与对应实际入参的集合。
id	Object	操作的资源id。 • add：创建资源对象中的资源id。 • update：修改的资源id。 • delete：删除的资源id。

参数名称	数据类型	描述
originDo	Object	api被调用前do对象（一般从数据库取出）。 ● add：为null。 ● update：修改前数据库中的原始对象。 ● delete：删除前数据库中的原始对象。
targetDo	Object	api想要设置的do对象的克隆对象（对其进行修改不会持久化）。 ● add：需要创建的对象。 ● update：需要修改到的对象。 ● delete：为null。
methodName	String	@Extension注解所在原方法名。
className	String	@Extension注解所在的类名。
contextMap	Map<String, Object>	缓存上下文对象，由请求插件设置，并由响应插件读取，通常key是插件名称。

内置插件

开启horizon后，会自动生成两个内置插件，用户在插件中实现统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等逻辑即可：

- Entry插件

```
/**
 * default request plugin
 */
@Component
@javax.annotation.Generated(value = "com.huaweicloud.devspore.codegen, xxx-RELEASE")
public class DefaultRequestPlugin extends Processor {
    @Override
    public boolean doProcess(DataEvent dataEvent) throws PluginException {
        // do your code
        return true;
    }
}
```

- Exit插件

```
/**
 * default response plugin
 */
@Component
@javax.annotation.Generated(value = "com.huaweicloud.devspore.codegen, xxx-RELEASE")
public class DefaultResponsePlugin implements Observer {
    @Override
    public void observe(DataEvent dataEvent) throws PluginException {
        // do your code
    }
}
```

- 配置文件

```
devspore:
  horizon:
    processors: defaultRequestPlugin
    observers: defaultResponsePlugin
```

1.1.3 用户自定义拦截插件使用

- **Entry插件**

  继承抽象类Processor，实现doProcess接口，可定义多个插件。

  插件均需注册为bean。

```
import com.huawei.devspore.horizon.DataEvent;
import com.huawei.devspore.horizon.exception.PluginException;
import com.huawei.devspore.horizon.processor.Processor;

import org.springframework.stereotype.Component;

/**
 * custom request plugin
 */
@Component
public class CustomRequestPlugin extends Processor {
    @Override
    public boolean doProcess(DataEvent dataEvent) throws PluginException {
        // do your code
        ........
        return true;
    }
}
```

  配置文件中多个插件以","分隔，多个插件按配置顺序执行。

```
evspore:
  horizon:
    processors: customRequestPlugin
```

- **Exit插件**

  实现Observer接口中observer接口，可定义多个插件。

  插件均需注册为bean。

```
import com.huawei.devspore.horizon.DataEvent;
import com.huawei.devspore.horizon.exception.PluginException;
import com.huawei.devspore.horizon.subscribe.Observer;

import org.springframework.stereotype.Component;

/**
 * custom response plugin
 */
@Component
public class CustomResponsePlugin implements Observer {
    @Override
    public void observe(DataEvent dataEvent) throws PluginException {
        // do your code
        ......
    }
}
```

  配置文件中多个插件以","分隔，多个插件按配置顺序执行。

```
devspore:
  horizon:
    observers: customResponsePlugin
```

1.1.4 @Extension 介绍

devspore-horizon是devspore提供的对metaBO（以及对应的RO）操作前鉴权、认证，metaBO变动后消息发送、审计等功能的组件。配置horizon后，generator生成的项目的repository中相关操作上会带有@Extension注解。

```
// 普通增删改查接口
@Extension(operation = Operation.CREATE, name = "Project")
```

```
// 用户自定义扩展接口
@Extension(operation = Operation.CUSTOM, customMethod = "methodName", name = "Project")
```

表 1-2 属性说明

参数名称	数据类型	描述
operation	Operation	操作类型。此值会传递到**DataEvent**对应operation字段。 • CREATE：新增操作。 • DELETE：删除操作。 • UPDATE：更新操作。 • VIEW：查询操作。 • BATCH_CREATE：批量新增。 • BATCH_UPDATE：批量更新。 • BATCH_VIEW：批量查询。 • BATCH_DELETE：批量删除。 • CUSTOM：用户自定义操作。
customMethod	String	扩展API中定义的method对象，仅在operation为CUSTOM时有意义；此值会传递到**DataEvent**对应customMethod字段。
name	String	表示本次操作的metaBO或者RO（关系类型）的名称。此值会传递到**DataEvent**对应sourceName字段。

操作示例

- 新增操作：此示例表示可对名为Order的metaBO新增操作做统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等。

```
/**
* AddOrder Method
*
* @param order order
* @return Order
*/
@Extension(operation = Operation.CREATE, name = "Order")
@Override
public Order addOrder(Order order) {
return orderRepository.saveSelective(order);
}
```

新增操作**插件参数DataEvent**取值示例：

表 1-3 新增操作 DataEvent 取值示例

参数名称	取值
metaDocument	服务的元数据对象，自动读取服务元数据并反序列化为metaDocument。
operation	CREATE。

参数名称	取值
customMethod	""。
sourceName	Order。
argsMap	order：创建的order对象。
id	order对象中Id值。
originDo	null。
targetDo	order的克隆对象。
methodName	addOrder。
className	该方法所在类全限定名。
contextMap	null。

- 批量新增操作：此示例表示可对名为Order的metaBO批量新增操作做统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等。

```
/**
 * AddOrders Method
 *
 * @param orderList orderList
 * @return List<Order>
 */
@Extension(operation = Operation.BATCH_CREATE, name = "Order")
@Override
public List<Order> addOrders(List<Order> orderList) {
    return orderRepository.saveSelectiveAll(orderList);
}
```

批量新增操作**插件参数DataEvent**取值示例：

表 1-4 批量新增操作 DataEvent 取值示例

参数名称	取值
metaDocument	服务的元数据对象，自动读取服务元数据并反序列化为metaDocument。
operation	BATCH_CREATE。
customMethod	""。
sourceName	Order。
argsMap	orderList：创建的order对象集合。
id	null。
originDo	null。
targetDo	null。
methodName	addOrders。
className	该方法所在类全限定名。

参数名称	取值
contextMap	null。

- 删除操作：此示例表示可对名为Order的metaBO删除操作做统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等。

```
/**
 * DeleteOrderById Method
 *
 * @param orderId orderId
 * @return Integer
 */
@Extension(operation = Operation.DELETE, name = "Order")
@Override
public Integer deleteOrderById(String orderId) {
    return orderRepository.deleteById(orderId);
}
```

删除操作**插件参数DataEvent**取值示例：

表 1-5 删除操作 DataEvent 取值示例

参数名称	取值
metaDocument	服务的元数据对象，自动读取服务元数据并反序列化为metaDocument。
operation	DELETE。
customMethod	""。
sourceName	Order。
argsMap	orderId：待删除的orderId。
id	orderId。
originDo	删除前数据库中orderId对应的原始对象。
targetDo	null。
methodName	deleteOrderById。
className	该方法所在类全限定名。
contextMap	null。

- 批量删除操作：此示例表示可对名为Order的metaBO批量删除操作做统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等。

```
/**
 * DeleteOrderByIds Method
 *
 * @param orderIds orderIds
 * @return Integer
 */
@Extension(operation = Operation.BATCH_DELETE, name = "Order")
@Override
public Integer deleteOrderByIds(List<String> orderIds) {
    return orderRepository.deleteAllById(orderIds);
}
```

批量删除操作**插件参数DataEvent**取值示例：

表 1-6 批量删除操作 DataEvent 取值示例

参数名称	取值
metaDocument	服务的元数据对象，自动读取服务元数据并反序列化为metaDocument。
operation	BATCH_DELETE。
customMethod	""。
sourceName	Order。
argsMap	orderIds：待删除的orderId集合。
id	orderId。
originDo	null。
targetDo	null。
methodName	deleteOrderByIds。
className	该方法所在类全限定名。
contextMap	null。

- 更新操作：此示例表示可对名为Order的metaBO更新操作做统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等。

```
/**
 * UpdateOrderById Method
 *
 * @param order order
 * @param orderId orderId
 * @return Order
 */
@Extension(operation = Operation.UPDATE, name = "Order")
@Override
public Order updateOrderById(Order order, String orderId) {
    return orderRepository.updateById(order);
}
```

更新操作**插件参数DataEvent**取值示例：

表 1-7 更新操作 DataEvent 取值示例

参数名称	取值
metaDocument	服务的元数据对象，自动读取服务元数据并反序列化为metaDocument。
operation	UPDATE。
customMethod	""。
sourceName	Order。

参数名称	取值
argsMap	• order：order对象。 • orderId：order对象Id。
id	orderId。
originDo	更新前数据库中orderId对应的原始对象。
targetDo	order的克隆对象。
methodName	updateOrderById。
className	该方法所在类全限定名。
contextMap	null。

- 批量更新操作：此示例表示可对名为Order的metaBO批量更新操作做统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等。

```
/**
* UpdateOrderByIds Method
*
* @param orderList orderList
* @return List<Order>
*/
@Extension(operation = Operation.BATCH_UPDATE, name = "Order")
@Override
public List<Order> updateOrderByIds(List<Order> orderList) {
    return orderRepository.updateByIds(orderList);
}
```

批量更新操作**插件参数DataEvent**取值示例：

表 1-8 批量更新操作 DataEvent 取值示例

参数名称	取值
metaDocument	服务的元数据对象，自动读取服务元数据并反序列化为metaDocument。
operation	BATCH_UPDATE。
customMethod	""。
sourceName	Order。
argsMap	orderList：待更新的order对象集合。
id	null。
originDo	null。
targetDo	null。
methodName	updateOrderByIds。
className	该方法所在类全限定名。
contextMap	null。

- 查询操作：此示例表示可对名为Order的metaBO查询操作做统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等。

```
/**
 * GetOrderById Method
 *
 * @param orderId orderId
 * @return Order
 */
@Extension(operation = Operation.VIEW, name = "Order")
@Override
public Order getOrderById(String orderId) {
    return orderRepository.findOrderById(orderId);
}
```

查询操作**插件参数DataEvent**取值示例：

表 1-9 查询操作 DataEvent 取值示例

参数名称	取值
metaDocument	服务的元数据对象，自动读取服务元数据并反序列化为metaDocument。
operation	VIEW。
customMethod	""。
sourceName	Order。
argsMap	orderId：order对象Id。
id	null。
originDo	null。
targetDo	null。
methodName	getOrderById。
className	该方法所在类全限定名。
contextMap	null。

- 批量查询操作：此示例表示可对名为Order的metaBO批量查询操作做统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等。

```
/**
 * GetOrders Method
 *
 * @param orderQo orderQo
 * @return PageInfo<Order>
 */
@Extension(operation = Operation.BATCH_VIEW, name = "Order")
@Override
public PageInfo<Order> getOrders(OrderQo orderQo) {
    return orderRepository.findOrders(orderQo);
}
```

批量查询操作**插件参数DataEvent**取值示例：

表 1-10 批量查询操作 DataEvent 取值示例

参数名称	取值
metaDocument	服务的元数据对象，自动读取服务元数据并反序列化为metaDocument。
operation	BATCH_VIEW。
customMethod	""。
sourceName	Order。
argsMap	orderQo：orderQo对象。
id	null。
originDo	null。
targetDo	null。
methodName	getOrders。
className	该方法所在类全限定名。
contextMap	null。

- 自定义扩展API操作：此示例表示可对名为Order的metaBO中自定义扩展API“pay”操作做统一的操作前认证、鉴权，操作后消息发送、缓存更新、审计等。

```
/**
 * pay
 *
 * @param payment payment
 * @param orderId orderId
 * @return OrderOrderDetailNested
 */
@Extension(operation = Operation.CUSTOM, customMethod = "pay", name = "Order")
@Override
public OrderOrderDetailNested pay(Payment payment, String accountId, String orderId) {
    OrderOrderDetailNested orderOrderDetailNested =
orderRepository.findOrderOrderDetailNested(accountId, orderId);

    ......

    return orderOrderDetailNested;
}
```

自定义扩展API操作**插件参数DataEvent**取值示例：

表 1-11 自定义扩展 API 操作 DataEvent 取值示例

参数名称	取值
metaDocument	服务的元数据对象，自动读取服务元数据并反序列化为metaDocument。
operation	CUSTOM。
customMethod	pay。
sourceName	Order。

参数名称	取值
argsMap	• payment：payment对象。 • accountId：accountId值。 • orderId：orderId值。
id	null。
originDo	null。
targetDo	null。
methodName	getOrders。
className	该方法所在类全限定名。
contextMap	null。

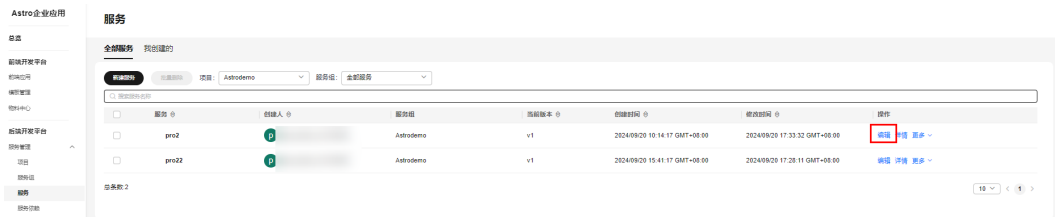
1.1.5 开启 horizon

步骤1 登录AstroPro控制台，单击“进入首页”，进入AstroPro实例。

步骤2 在左侧导航栏中，选择“后端开发平台 > 服务管理 > 服务”。

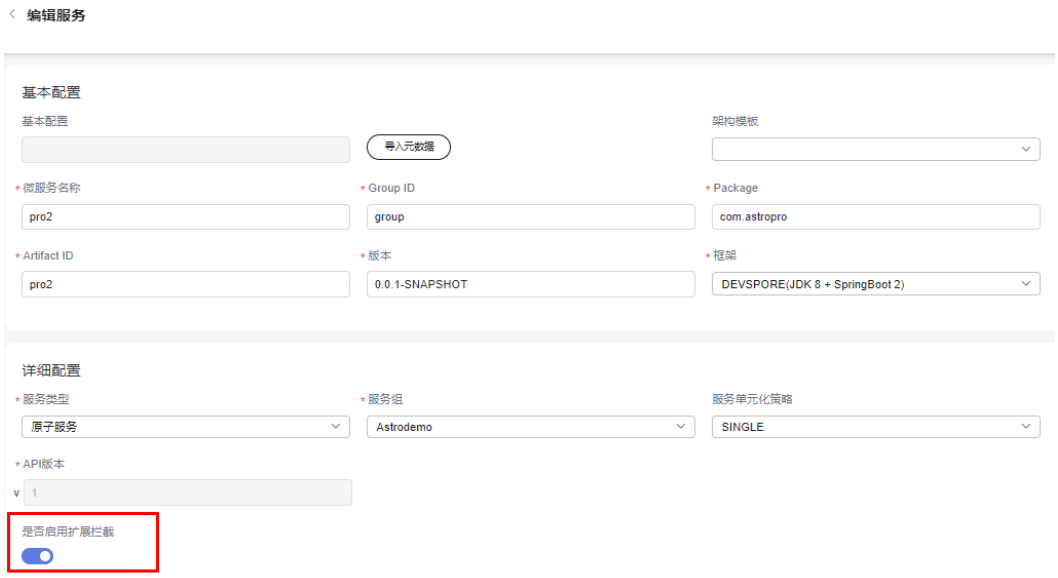
步骤3 在服务列表中，单击待操作服务“操作”列的“编辑”。

图 1-4 编辑服务



步骤4 在基本配置中，开启“是否启用扩展拦截”。

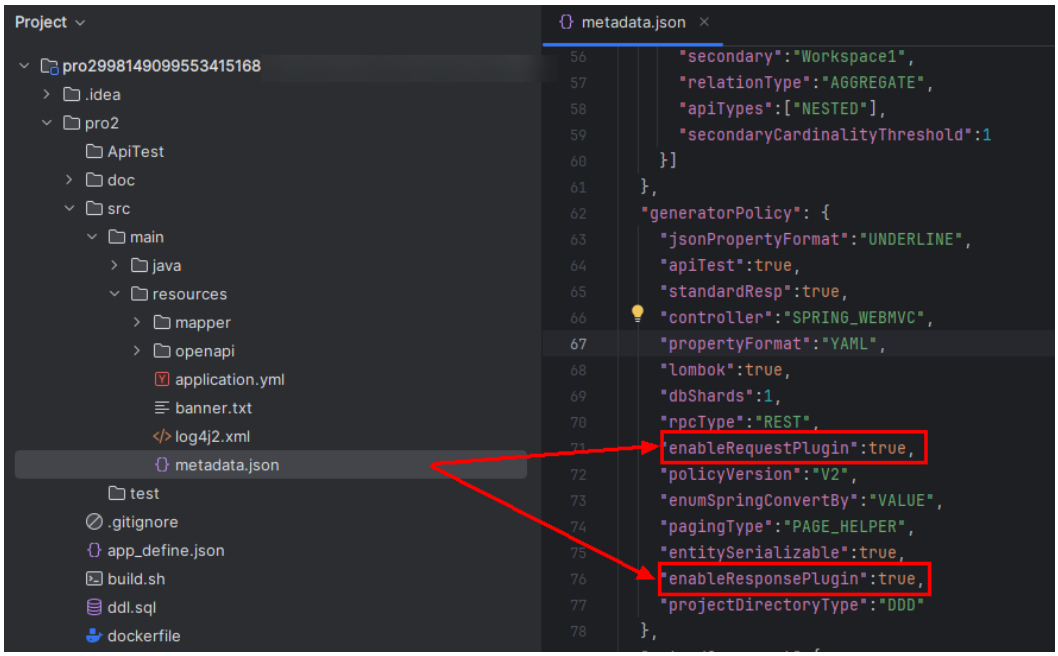
图 1-5 启用扩展拦截



步骤5 “框架配置”、“生成策略”、“业务设计”和“服务依赖”按需配置。更多可参考**后端开发快速入门**。

步骤6 **生成服务代码**后，打开本地压缩包，在“src/main/resources”路径下，查看在metadata.json文件，其中，generatorPolicy.enableRequestPlugin和generatorPolicy.enableResponsePlugin的属性值为true，表示已开启horizon功能。

图 1-6 查看代码



----结束

1.2 自定义认证开发指南

1.2.1 开启自定义认证功能

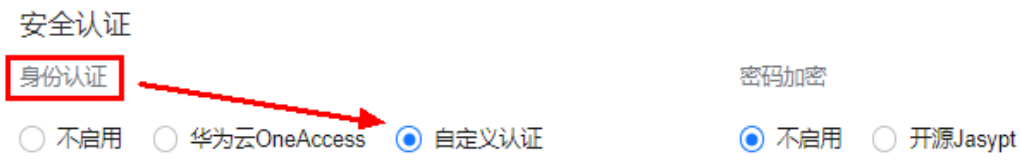
步骤1 登录AstroPro控制台，单击“进入首页”，进入AstroPro实例。

步骤2 在左侧导航栏中，选择“后端开发平台 > 服务管理 > 服务”。

步骤3 在服务列表中，选择对应服务，单击“操作”列“编辑”，进入服务编辑页面。

步骤4 服务编辑页面中，在“框架配置”页面，“安全认证”模块中，“身份认证”选择“自定义认证”。

图 1-7 开启自定义认证



----结束

1.2.2 用户自定义认证使用

工作原理

- 引入devspore-auth-starter依赖后，会在SDK内自动注册认证Filter：AuthAuthenticationFilter。
- 调用业务接口时会自动执行已注册的认证Filter，AuthAuthenticationFilter.doFilterInternal中将通过SPI方式先调用TokenParser的实现类（DefaultTokenParser，可自定义）完成token解析。
- AuthAuthenticationFilter.doFilterInternal在token解析成功后将解析出的用户信息通过SPI方式调用UserStore的实现类（DefaultUserStore，用户可自定义）存储用户信息供后续业务代码使用。

约束与限制

TokenParser实现类与UserStore实现类，均需注册成bean，并完成bean扫描，确保Spring容器中可获取到对应bean。

TokenParser 接口介绍

```
package com.huaweicloud.devspore.auth.commons.spi;

import com.huawei.devspore.plugin.spi.authentication.UserInfo;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public interface TokenParser {

    /**
     * 根据token解析用户信息
     *
     * @param token
     * @param response
```

```
     * @return 用户信息 UserInfo
     */
    UserInfo parserToken(String token, HttpServletResponse response);

    /**
     * 根据HttpRequest解析用户信息
     *
     * @param httpServletRequest
     * @param response
     * @return 用户信息 UserInfo
     */
    UserInfo parserToken(HttpServletRequest httpServletRequest, HttpServletResponse response);
}
```

两个方法，用户选其一实现即可；AuthAuthenticationFilter中通过devspore.auth.in-header配置值（默认true，配置说明详见下文“**配置说明**”章节）自动调用相应方法。

- 若token定义在header中，需实现parserToken（String token, HttpServletResponse response）接口，token获取根据配置由devspore-auth-starter获取。
- 若token未定义在header中，devspore.auth.in-header配置为false，并实现parserToken（HttpServletRequest httpServletRequest, HttpServletResponse response）接口。

UserInfo 用户信息类

```
package com.huawei.devspore.plugin.spi.authentication;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import lombok.Data;
import java.io.Serializable;
import java.util.List;
/**
 * 用户信息
 */
@Data
public class UserInfo {
    /**
     * 用户Id
     */
    private String userId;

    /**
     * 用户名
     */
    private String userName;

    /**
     * 租户Id
     */
    private String tenantId;

    /**
     * 租户名
     */
    private String tenantName;

    /**
     * 请求token
     */
    private String token;

    /**
     * 用户角色
     */
```

```
    private List<String> roles;

    /**
     * JWTToken信息，仅JWTToken使用
     */
    private JwtInfo jwtInfo;

    @JsonIgnoreProperties(ignoreUnknown = true)
    @Data
    public static final class JwtInfo implements Serializable {
        private static final long serialVersionUID = -2464405170204203810L;

        /**
         * 签发者
         */
        private String iss;

        /**
         * 主题
         */
        private String sub;

        /**
         * 签收者
         */
        private List<String> aud;

        /**
         * 过期时间
         */
        private String exp;

        /**
         * 生效时间
         */
        private String nbf;

        /**
         * 签发时间
         */
        private String iat;

        /**
         * Jwt唯一ID
         */
        private String jti;
    }
}
```

实现 TokenParser 接口示例

```
import com.huawei.devspore.plugin.spi.authentication.UserInfo;

import com.huaweicloud.devspore.auth.commons.spi.TokenParser;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Component;

@Component
public class CustomTokenParser implements TokenParser {

    /**
     * 若token定义在header中，可使用该接口，token获取根据配置由devspore-auth获取
     *
     * @param token
     * @param response
     * @return UserInfo
```

```
     */
    @Override
    public UserInfo parserToken(String token, HttpServletResponse response) {

        // 自定义token解析
        ......

        // 判断token是否过期
        ......

        UserInfo userInfo = new UserInfo();

        // 根据解析后的token信息填充UserInfo对象
        ......

        return userInfo;
    }

    /**
     * 若token未定义在header中，使用该接口
     *
     * @param request
     * @param response
     * @return UserInfo
     */
    @Override
    public UserInfo parserToken(HttpServletRequest request, HttpServletResponse response) {
        // 从httpServletRequest获取token
        ......

        // 自定义token解析
        ......

        // 判断token是否过期
        ......

        UserInfo userInfo = new UserInfo();

        // 根据解析后的token信息填充UserInfo对象
        ......

        return userInfo;
    }

}
```

- token解析成功返回用户信息UserInfo，UserInfo中userId为空则默认解析失败。
- token解析失败，返回UserInfo为空，可抛出com.huaweicloud.devspore.auth.commons.exception.DevsporeAuthException，也可使用用户自定义异常。

UserStore 接口介绍

```
package com.huawei.devspore.plugin.spi.authentication;

/**
 * 用户信息接口
 */
public interface UserStore {

    /**
     * 设置用户信息
     * @param userInfo
     */
    void setUserInfo(UserInfo userInfo);

    /**
     * 获取用户信息
```

```
     * @return UserInfo
     */
    UserInfo getUserInfo();
}
```

- AuthAuthenticationFilter.doFilterInternal会在token解析成功后调用setUserInfo（UserInfo userInfo）接口存储用户信息。
- 后续业务接口鉴权或业务逻辑需要时调用getUserInfo()接口获取当前用户信息。

实现 UserStore 接口

```
import com.huawei.devspore.plugin.spi.authentication.UserInfo;
import com.huawei.devspore.plugin.spi.authentication.UserStore;

import org.springframework.stereotype.Component;

@Component
public class CustomUserStore implements UserStore {

    @Override
    public void setUserInfo(UserInfo userInfo) {
        // 自定义存储用户信息
        ......
    }

    @Override
    public UserInfo getUserInfo() {
        UserInfo userInfo = new UserInfo();

        // 自定义用户信息获取
        ......

        return userInfo;
    }

}
```

认证配置

一般服务配置devspore.auth.exclude-paths与devspore.auth.auth-patterns其中之一即可。

在配置文件中加入devspore相关配置，如下：

示例1：/healthz;/error;/swagger-ui.html;/swagger-ui/**;/oneaccess-docs/**;/api-docs/**;等API无需认证，除此之外所有API均进行认证。

```
devspore:
  auth:
    exclude-paths: /healthz;/error;/swagger-ui.html;/swagger-ui/**;/oneaccess-docs/**;/api-docs/**; # 不需要认证的路径，多个规则用;分隔
    in-header: true  # token是否在header中，默认true
    token-name: X-Auth-Token  # token在header中的参数名，inHeader设置为true时生效，默认X-Auth-Token
```

示例2：以“/v1/”开头的所有API均进行认证。

```
devspore:
  auth:
    auth-patterns: /v1/**  # 需要认证的路径，多个规则用;分隔
    in-header: true  # token是否在header中，默认true
    token-name: X-Auth-Token  # token在header中的参数名，inHeader设置为true时生效，默认X-Auth-Token
```

配置说明

表 1-12 配置说明

参数名	是否必须	功能
devspore.auth.exclude-paths	false	不需要认证的路径，多个规则用“;”分隔，优先级大于devspore.auth.auth-patterns。
devspore.auth.auth-patterns	false	需要认证的路径，多个规则用“;”分隔。
devspore.auth.in-header	false	token是否在header中，默认true。
devspore.auth.token-name	false	token在header中的参数名，inHeader设置为true时生效，默认“X-Auth-Token”。

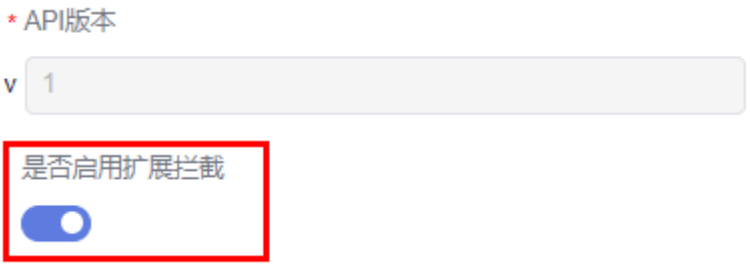
1.3 自定义插件服务鉴权开发指南

1.3.1 开启自定义鉴权功能

自定义鉴权功能在微服务架构、API管理、企业内部系统集成等方面发挥着重要作用，它提供了灵活性和可扩展性，以满足不同业务场景下的安全需求。

开启自定义鉴权功能前，请确保依赖服务的“是否启用扩展拦截”按钮已启用。

图 1-8 开启“是否启用扩展拦截”配置



步骤1 登录AstroPro控制台，单击“进入首页”，进入AstroPro实例。

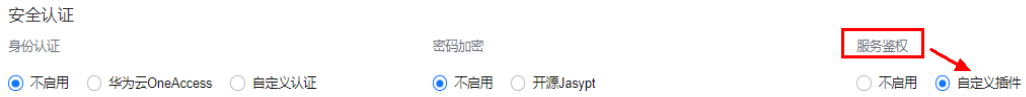
步骤2 在左侧导航栏中，选择“后端开发平台 > 服务管理 > 服务”。

步骤3 在服务列表中，选择对应服务，单击“操作”列“编辑”，进入服务编辑页面。

步骤4 在服务编辑页面中，单击流程导图中的“框架配置”，进入框架配置页面。

步骤5 在“安全认证”模块中，设置“服务鉴权”为“自定义插件”。

图 1-9 开启自定义鉴权



----结束

1.3.2 用户自定义鉴权使用

devspore提供鉴权插件接口，用户只需实现对应鉴权接口即可。

自定义鉴权插件

- 继承抽象类Processor，实现doProcess接口，可定义多个插件。
- 插件均需注册为bean。

```
import com.huawei.devspore.plugin.spi.authorization.AuthObject;
import com.huawei.devspore.plugin.spi.authorization.AuthSubject;
import com.huawei.devspore.plugin.spi.authorization.Authorizer;

import org.springframework.stereotype.Component;

/**
 * custom Authorizer
 */
@Component
public class CustomAuthorizer implements Authorizer {
    /**
     * 执行鉴权
     *
     * @param subject 鉴权主体
     * @param resource 鉴权客体，被鉴权对象
     * @param operation 被鉴权的动作，传入方法自动注解的operation，为MetaBo对象上的对应的
Operation:
     *        CREATE,DELETE,UPDATE,VIEW...或者用户自定义扩展API中的action
     * @param properties 鉴权动作的其它context
     * @return 鉴权通过返回true,否则返回false
     */
    @Override
    public boolean authorize(@NonNull final AuthSubject subject, @NonNull final AuthObject resource,
        @NonNull final String operation, @Nullable final Map<String, String> properties) {
        // do your authorizer code
        .......
        return true;
    }
}
```

- 配置文件。

```
devspore:
  horizon:
    processors: authorizerProcessor
```

AuthSubject 对象

AuthSubject对象即鉴权主体。

表 1-13 AuthSubject 对象参数说明

参数名称	数据类型	描述
tenantId	String	根据用户所在的租户传值： ● 对于非多租应用（MetaService中tenantModel为空），允许为空。 ● 如果为多租应用（MetaService中tenantModel值为TENANT或者TENANT_PROJECT），则传入tenantId。
uid	String	鉴权主体的user id，不允许为空。

AuthObject 对象

AuthObject对象即鉴权客体，被鉴权对象。

表 1-14 AuthObject 对象参数说明

参数名称	数据类型	描述
projectId	String	被鉴权资源所在的项目id，对于不属于项目的资源，允许为空。 ● 对于非多项目应用（MetaService中tenantModel为null或者TENANT），允许为空。 ● 如果为多项目应用（MetaService中tenantModel值为TENANT_PROJECT），则传入projectId。
resource	String	被鉴权资源，不允许为空。 根据MetaBOAuthorizeType不同的值，传入对应值。 ● ROOT：传入BO对象所在的Root BO的id。 ● TYPE_LEVEL：传入BO name。 ● INSTANCE_LEVEL：传入BO对象id。

horizon 插件与鉴权插件配合使用

开启自定义鉴权插件后，devspore为用户自动生成内置**入口插件**，并在内置**入口插件**中实现向鉴权接口的参数传递；用户只需实现Authorizer.authorize鉴权接口即可。

- 内置**入口插件**。

```
package com.huawei.devspore.horizon.authorization;

import com.huawei.devspore.horizon.DataEvent;
import com.huawei.devspore.horizon.Operation;
import com.huawei.devspore.horizon.exception.PluginException;
import com.huawei.devspore.horizon.factory.Plugins;
import com.huawei.devspore.horizon.processor.Processor;
```

```
import com.huawei.devspore.metadata.v1.model.MetaBO;
import com.huawei.devspore.plugin.spi.authentication.UserStore;
import com.huawei.devspore.plugin.spi.authorization.AuthObject;
import com.huawei.devspore.plugin.spi.authorization.AuthSubject;
import com.huawei.devspore.plugin.spi.authorization.Authorizer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.lang.NonNull;
import org.springframework.lang.Nullable;
import org.springframework.stereotype.Component;

import java.util.Map;

@Component
public class AuthorizerProcessor extends Processor {

    @Autowired(required = false)
    private UserStore userStore;

    @Autowired(required = false)
    private Authorizer authorizer;

    @Override
    public boolean doProcess(@NonNull final DataEvent dataEvent) throws PluginException {
        if (authorizer == null) {
            throw new PluginException("No Authorizer SPI implementation defined");
        }
        MetaBO metaBO = Plugins.getMetaBO(dataEvent);
        AuthObject resource = new AuthObject();
        switch (metaBO.getAtzType()) {
            case TYPE_LEVEL:
                resource.setResource(metaBO.getName());
                return authorize(getAuthSubject(), resource, getOperation(dataEvent), null);
            case INSTANCE_LEVEL:
                resource.setResource(dataEvent.getId().toString());
                return authorize(getAuthSubject(), resource, getOperation(dataEvent), null);
            default:
                return true;
        }
    }

    private AuthSubject getAuthSubject() {
        AuthSubject authSubject = new AuthSubject();
        authSubject.setTenantId(userStore.getUserInfo().getTenantId());
        authSubject.setUid(userStore.getUserInfo().getUserId());
        return authSubject;
    }

    /**
     * @return operation for authorization, audit and other purposes
     */
    public String getOperation(DataEvent dataEvent) {
        if (dataEvent.getOperation() == Operation.CUSTOM) {
            return dataEvent.getCustomMethod();
        } else {
            return dataEvent.getOperation().toString();
        }
    }

    /**
     * 执行鉴权
     *
     * @param subject 鉴权主体
     * @param resource 鉴权客体，被鉴权对象
     * @param operation 被鉴权的动作，传入方法自动注解的operation，为aggregate root对象上的对应
的Operation:
     *        CREATE,DELETE,UPDATE,VIEW,VIEW_SECRET或者MetaExtAction中的action
     * @param properties 鉴权动作的其它context
     * @return 鉴权通过返回true, 否则返回false
```

```
     */
    public boolean authorize(@NonNull final AuthSubject subject, @NonNull final AuthObject resource,
        @NonNull final String operation, @Nullable final Map<String, String> properties) {
        return authorizer.authorize(subject, resource, operation, properties);
    }

}
```

- UserStore接口介绍：**UserStore接口介绍**。
- UserInfo用户信息类介绍：**UserInfo用户信息类**。

若内置**入口插件**无法满足用户业务需求，可在自动生成的DefaultRequestPlugin中完成插件逻辑，也可自定义插件，详见**1.1.2 拦截插件**。

2 DevSpore-SDK 使用指南

2.1 devspore-auth 使用指南

2.1.1 devspore-auth-oneaccess 使用指南

2.1.1.1 devspore-auth-oneaccess 使用概述

OneAccess是华为云提供的应用身份管理服务，具备集中式的身份管理、认证和授权能力，保证企业用户根据权限访问受信任的云端和本地应用系统，并对异常访问行为进行有效防范。

devspore-auth-oneaccess模块用于对OneAccess服务颁发的token进行认证。

2.1.1.2 使用指南

基础开发环境准备

安装的工具包括JDK、Maven、Eclipse和IDEA ，配置对应的环境变量，确保本地开发环境可用。

添加依赖

代码中添加spring-boot-starter-huawei-devspore-auth-oneaccess依赖。

```
<dependency>
   <groupId>com.huaweicloud.devspore</groupId>
   <artifactId>spring-boot-starter-huawei-devspore-auth-oneaccess</artifactId>
   <version>${project.version}</version>
   <exclusions>
      <exclusion>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-logging</artifactId>
      </exclusion>
      <exclusion>
         <artifactId>log4j-to-slf4j</artifactId>
         <groupId>org.apache.logging.log4j</groupId>
      </exclusion>
   </exclusions>
</dependency>
```


认证配置

在配置文件中加入devspore相关配置，如下：

```
devspore:
  auth:
    exclude-paths: GET:/test # 不需要认证的路径
    in-header: true  #token是否在header中，默认true
    token-name: X-Auth-Token  #token在header中的参数名，inHeader设置为true时生效，默认X-Auth-Token
    oneaccess:
      ignore-ssl: true  #是否校验oneaccess证书
      jwt:
        jwk-set-uri: https://arvymzmajv.huaweicloudoneaccess.com/api/v1/oauth2/keys #对应的oneaccessjwk地
址
        issuer-uri: https://arvymzmajv.huaweicloudoneaccess.com/api/v1/oauth2 #签发机构
```

arvymzmajv.huaweicloudoneaccess.com为本文示例中使用的域名，实际应用中请换成自己对应的域名。

添加完后启动项目即可开启对token的认证。

2.1.1.3 配置说明

表 2-1 参数配置说明

参数名称	是否必选	功能
devspore.auth.in-header	false	token是否在header中，默认true。
devspore.auth.token-name	false	token在header中的参数名，inHeader设置为true时生效，默认"X-Auth-Token"。
devspore.auth.oneaccess.ignore-ssl	true	是否校验oneaccess证书。
devspore.auth.oneaccess.auth-patterns	false	需要认证的路径。
devspore.auth.oneaccess.jwt.jwk-set-uri	true	对应的oneaccessjwk地址。
devspore.auth.oneaccess.jwt.issuer-uri	true	签发机构。
devspore.auth.oneaccess.jwt.jws-algorithm	false	签名算法，默认RS256。

2.1.2 devspore-auth 自定义 token 解析认证

模块依赖设计图

图 2-1 devspore-auth 插件依赖

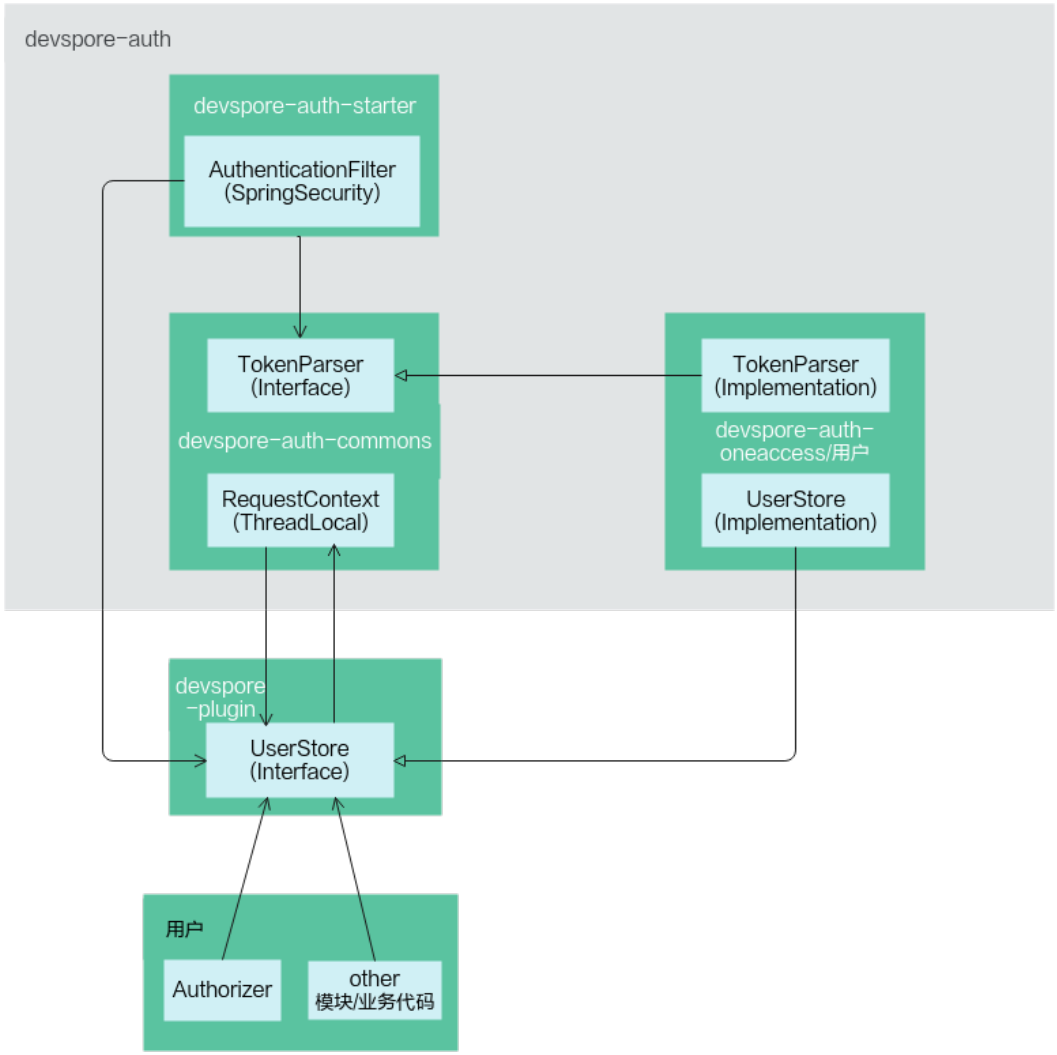
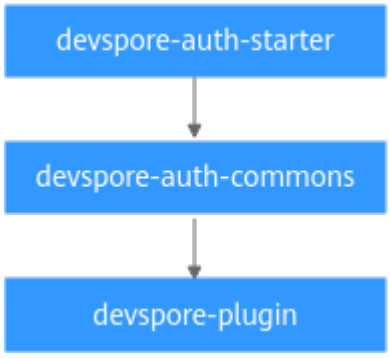


图 2-2 模块依赖图



开发环境准备

需提前安装的工具，包括JDK、Maven、Eclipse和IDEA，配置对应的环境变量，确保本地开发环境可用。

工作原理

- 引入devspore-auth-starter依赖后，会在SDK内自动注册认证Filter：AuthAuthenticationFilter。
- 调用业务接口时会自动执行已注册的认证Filter，AuthAuthenticationFilter.doFilterInternal中将通过SPI方式先调用TokenParser的实现类（DefaultTokenParser，用户可自定义）完成token解析。
- AuthAuthenticationFilter.doFilterInternal在token解析成功后将解析出的用户信息通过SPI方式调用UserStore的实现类（DefaultUserStore，用户可自定义）存储用户信息供后续业务代码使用。
- TokenParser接口介绍详见：**TokenParser接口介绍**
- UserInfo用户信息类介绍详见：**UserInfo用户信息类**
- UserStore接口介绍详见：**UserStore接口介绍**

认证配置

认证配置详见：**认证配置**

配置说明

配置说明详见：**配置说明**

使用 devspore-auth 通用认证能力

步骤1 添加依赖。

代码中添加devspore-auth-starter依赖。

```
<dependency>
   <groupId>com.huaweicloud.devspore</groupId>
   <artifactId>devspore-auth-starter</artifactId>
   <version>${devspore-auth-starter.version}</version>
</dependency>
```

须知

TokenParser实现类与UserStore实现类，均需注册成bean，并完成bean扫描，确保Spring容器中可获取到对应bean。

步骤2 实现TokenParser接口：详见**实现TokenParser接口示例**

步骤3 实现UserStore接口：详见**实现UserStore接口**

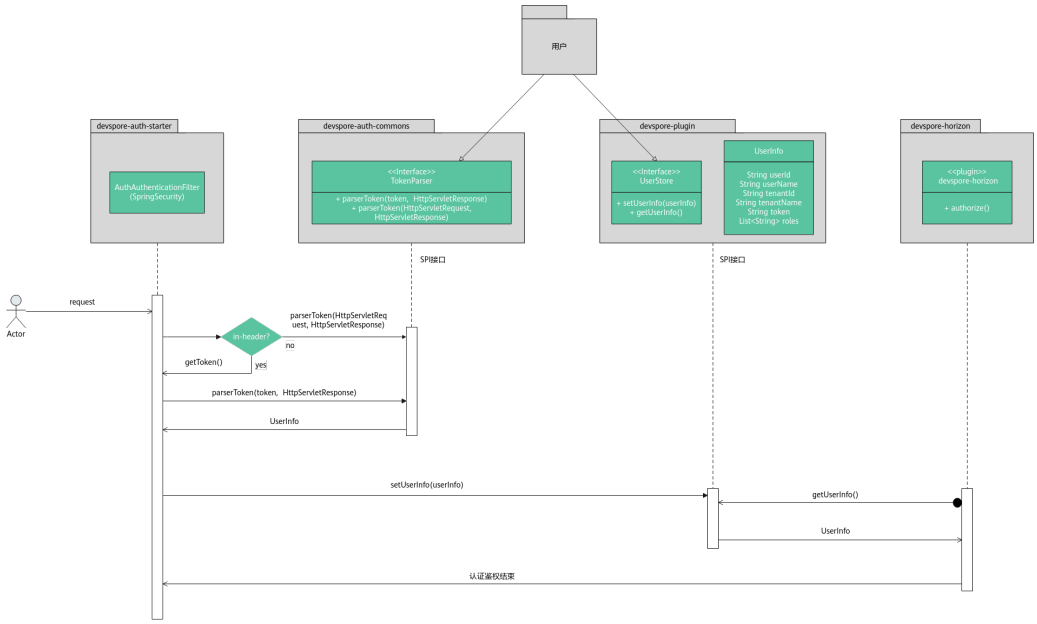
步骤4 添加认证配置：在配置文件中加入devspore相关配置，详见**认证配置**

----结束

时序图

使用devspore完成认证鉴权模块调用时序图，如图2-3所示。

图 2-3 模块调用时序图



2.2 devspore-security 使用指南

2.2.1 devspore-security 使用概述

devspore-security安全组件内置了明文加密、参数校验等安全功能，相比原有组件自动化程度更高，集成方式更简单。

表 2-2 安全功能

名称	描述
配置自动加解密	自动对配置文件的字段加解密，通过(NeedEnc)(NeedDec)标注。
参数校验	内置标准参数校验规则。

2.2.2 使用指南

基础开发环境准备

安装的工具包括JDK、Maven、Eclipse和IDEA ，配置对应的环境变量，确保本地开发环境可用。

使用说明

- 配置文件里的密钥，建议自行加密存储并实现解密逻辑。
- 输入的密钥，建议按全随机数生成，长度32字节以上。

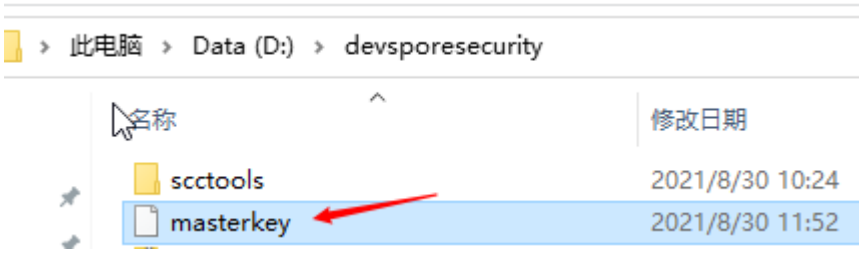
敏感数据加密

DevSporeSecurity集成Jasypt加解密组件，实现配置文件敏感数据自动加解密。

步骤1 pom文件中添加如下依赖。

```
<dependency>
    <groupId>com.huaweicloud.devspore</groupId>
    <artifactId>devspore-security</artifactId>
    <version>${version}</version>
</dependency>
<!--或者-->
<dependency>
    <groupId>com.huaweicloud.devspore</groupId>
    <artifactId>spring-boot-starter-huawei-devspore-security</artifactId>
    <version>${revision}</version>
</dependency>
```

步骤2 配置masterkey，内容自行定义。



步骤3 配置masterkey路径和需要加密的属性。

```
#在需要加密的属性前添加(NeedEnc)前缀
test=(NeedEnc)123
devspore.security.provider.jasypt.masterkey-path=D:/devsporesecurity
```

步骤4 配置masterkeypath的路径。

masterkeypath的路径需要满足如下要求：

- 路径必须为如下路径：
  ```
  ${devspore.security.provider.jasypt.masterkey.path}/masterkey
  ```
- masterKey属于敏感数据，需服务自行控制好权限。

  如果在windows上运行，则在第一次启动时会自动将配置文件加密，加密后的内容如下所示。
  ```
  test=(NeedDec)EBl1iK4irrKDl4lj0g+pPg==
  devspore.security.provider.jasypt.masterkey-path=D:/devsporesecurity
  ```
  可以看到test对应的属性已变为加密数据。

----结束

hibernate-validator 参数校验

步骤1 pom引入依赖。

```
<dependency>
    <groupId>com.huaweicloud.devspore</groupId>
    <artifactId>devspore-security</artifactId>
    <version>${version}</version>
```

```
</dependency>
<!--或者-->
<dependency>
    <groupId>com.huaweicloud.devspore</groupId>
    <artifactId>spring-boot-starter-huawei-devspore-security</artifactId>
    <version>${com.huaweicloud.version}</version>
</dependency>
```

步骤2 校验使用。

```
@Data
public class User {

  @NotBlank(message = "username is blank")
  private String username;

  @Pattern(regexp = "^1([34578])\\d{9}$", message = "incorrect mobile number format ")
  private String mobile;

  @NotNull(message = "age is blank")
  @Range(min = 5, max = 90, message = "age between 5 and 90")
  private Integer age;
}
@RestController
@Validated
public class DemoController {
    @RequestMapping("/demo1")
    public String testHandler1(@Past(message = "必须是过去的时间") Date date) {
        return "pass demo1";
    }

    @PostMapping("/demo2")
    public String testHandler2(@RequestBody @Valid User user) {
        return "pass demo2";
    }
}
```

----结束

常用约束

表 2-3 常用约束说明

注解	约束说明
@Null	被注释的元素必须为null。
@NotNull	被注释的元素必须不为null。
@AssertTrue	被注释的元素必须为true。
@AssertFalse	被注释的元素必须为false。
@Min(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值。
@Max(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值。
@DecimalMin(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值。
@DecimalMax(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值。

注解	约束说明
@Size(max=, min=)	被注释的元素的大小必须在指定的范围内。
@Digits (integer, fraction)	被注释的元素必须是一个数字，其值必须在可接受的范围内。
@Past	被注释的元素必须是一个过去的日期。
@Future	被注释的元素必须是一个将来的日期。
@Pattern(regex=,flag=)	被注释的元素必须符合指定的正则表达式。
@NotBlank(message =)	验证字符串非null，且长度必须大于0。
@Email	被注释的元素必须是电子邮箱地址。
@Length(min=,max=)	被注释的字符串的大小必须在指定的范围内。
@NotEmpty	被注释的字符串的必须非空。
@Range(min=,max=,message =)	被注释的元素必须在合适的范围内。

2.2.3 配置说明

表 2-4 公共参数说明

参数名称	是否必选	默认值	类型	说明
devspore.security.debug	否	true	String	是否为debug模式，debug模式会打印加解密调试信息。
devspore.security.provider	否	com.huaweicloud.devspore.security.crypto.DevSporeSecurityExtFactory	String	自动配置时使用的加解密提供方具体工程完整类路径。

表 2-5 jasypt 参数

参数名称	是否必选	默认值	类型	说明
devspore.security.provider.jasypt.masterkey.path	是	/devsporesecurity	String	jasypt masterkey路径。

2.3 devspore-probe 使用指南

2.3.1 devspore-probe 使用概述

devspore-probe主要是用来对微服务进行监控指标暴露的功能组件，devspore-probe对Micrometer做二次封装，借助自动配置，默认提供多种中间件的兼容，同时支持指标扩展，业务可根据自身需要扩展指标。

devspore-probe除了原有的Micrometer的功能外，还提供了以下两种功能：

- 监控数据自动上报AOM平台
- 支持注解形式自定义监控指标

2.3.2 使用指南

基础开发环境准备

安装的工具包括JDK、Maven、Eclipse和IDEA ，配置对应的环境变量，确保本地开发环境可用。

如何使用 devspore-probe

步骤1 引入Maven依赖，在项目的pom.xml中添加下面的依赖，version应使用最新版本。

```
<dependency>
   <groupId>com.huaweicloud.devspore</groupId>
   <artifactId>devspore-probe</artifactId>
   <version>lastest</version>
</dependency>
```

步骤2 项目配置文件application.yaml或者application.properties中加入devspore-probe配置项。

须知

- 下文中配置项的格式为yaml文件，如果配置文件为properties格式，需自行修改格式。
- 配置项的具体值含义参考**2.3.4 配置说明**

devspore-probe的配置样例：

```
devspore:
   probe:
      metrics:
         tags:
            component: demo-application
```

----结束

2.3.3 功能介绍

2.3.3.1 支持注解形式自定义监控指标

使用场景

为了方便用户更加快速、便捷的使用micrometer定义自己的监控指标，devspore-probe提供了注解的方式去快速的定义监控指标，避免了定义复杂的监控类。

如何使用

步骤1 项目中引入devspore-probe依赖，在项目的pom.xml中添加下面的配置。

```
<dependency>
   <groupId>com.huaweicloud.devspore</groupId>
   <artifactId>devspore-probe</artifactId>
   <version>lastest</version>
</dependency>
```

步骤2 根据业务需要，添加自定义监控指标，devspore-probe根据Prometheus的监控数据类型，提供了Counter、Gauge、Summary、Histogram四种类型的自定义指标类型，使用说明、配置、示例、效果参照**表2-6**。

表 2-6 自定义监控指标

监控类型	注解参数	示例

Counter（计数器）
每次调用被注解函数，监控指标值自动加1。
	方法注解@Counter。
	• name：String类型，自定义指标名称。
	• 方法类型：不限。
	• 效果：每次执行该方法，对应指标值+1。
	参数注解@Label。
	• name：String类型，自定义指标Label Key值。
	• 被注解的参数：String类型，自定义指标Label Value值。
		@Counter(name = "application_name")
		public Object index(@Label(name = "name") String userName, @Label(name = "age") String age) {
		// 执行业务代码
		}
		prometheus效果：
		application_name_total{age="xxx",application="devspore-application",name="xxxxxx",} 1.0
		application_name_total{age="yyy",application="devspore-application",name="yyyyyy",} 1.0

监控类型	注解参数	示例

Gauge(测量)
是表示单个数值，可以任意地上升和下降的度量。
	方法注解@Guage。
	• name：String类型，自定义指标名称。
	• 方法类型：Number。
	• 效果：每次执行该方法，对应指标值设置为方法的返回值。
	参数注解@Label。
	• name：String类型，自定义指标Label Key值。
	• 被注解的参数：String类型，自定义指标Label Value值 。
		@Gauge(name = "application_cpu_usage")
		public double gauge(@Label(name = "name") String userName, @Label(name = "age") String age) {
		// 执行业务代码
		}
		prometheus效果：
		application_cpu_usage{age="xxx",application="devspore-application2",name="xxxx",}
		0.013787109469182268
		application_cpu_usage{age="yyy",application="devspore-application2",name="yyyy",}
		0.4807864361390415

Summary(概要)
提供数值的百分比分布区间，百分比包括0.50，0.75，0.90，0.99。
	方法注解@Summary。
	• name：String类型，自定义指标名称。
	• 方法类型：Number。
	• 效果：每次执行该方法，对应指标添加该方法的返回值作为统计数据。
	参数注解@Label。
	• name：String类型，自定义指标Label Key值。
	• 被注解的参数：String类型，自定义指标Label Value值。
		@Summary(name = "request_time_summary")
		public double summary(@Label(name = "name") String userName, @Label(name = "age") String age) {
		// 执行业务代码
		}
		prometheus效果：
		request_time_summary{age="xxx",application="devspore-application2",name="xxxx",quantile="0.5",}
		0.52734375
		request_time_summary{age="xxx",application="devspore-application2",name="xxxx",quantile="0.75",}
		0.74609375
		request_time_summary{age="xxx",application="devspore-application2",name="xxxx",quantile="0.9",}
		0.87109375
		request_time_summary{age="xxx",application="devspore-application2",name="xxxx",quantile="0.99",}
		0.99609375
		request_time_summary_count{age="xxx",application="devspore-application2",name="xxxx",} 100.0
		request_time_summary_sum{age="xxx",application="devspore-application2",name="xxxx",}
		51.06987916008812

监控类型	注解参数	示例

Histogram（直方图）
不同区间内样本的个数。
	方法注解@Histogram。
	• name：String类型，自定义指标名称。
	• maxValue：double类型，数据采样的最大值。
	• 方法类型：Number。
	• 效果：每次执行该方法，对应指标添加该方法的返回值作为统计数据。
	参数注解@Label。
	• name：String类型，自定义指标Label Key值。
	• 被注解的参数：String类型，自定义指标Label Value值。
		@Histogram(name = "request_time_histogram", maxValue = 10)
		public double histogram(@Label(name = "name") String userName, @Label(name = "age") String age) {
		// 执行业务代码
		}
		prometheus效果：
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="1.0",} 5.0
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="2.0",} 12.0
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="3.0",} 17.0
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="4.0",} 23.0
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="5.0",} 28.0
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="6.0",} 34.0
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="7.0",} 39.0
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="8.0",} 43.0
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="9.0",} 47.0
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="10.0",} 52.0
		request_time_histogram_bucket{age="xxx",application="devspore-application2",name="xxxx",le="+Inf",} 52.0
		request_time_histogram_count{age="xxx",application="devspore-application2",name="xxxx",} 52.0
		request_time_histogram_sum{age="xxx",application="dev

----结束

2.3.3.2 监控数据自动上报 AOM 平台

使用场景

devspore-probe支持将微服务中的监控指标及数据自动上报到AOM平台，用户可在AOM平台上通过指标名称、标签等过滤并分析监控数据。

相关概念

应用运维管理（Application Operations Management），简称AOM，是云上应用的一站式立体化运维管理平台，实时监控应用及云资源，采集各项指标、日志及事件等

数据分析应用健康状态，提供告警及数据可视化功能，帮助用户及时发现故障，全面掌握应用、资源及业务的实时运行状况。

通过 AK/SK 认证将监控数据上报 AOM

步骤1 获取用户AK/SK，具体可参考**获取AK/SK**。

步骤2 获取AOM监控上报地址，具体可参考**添加监控数据**中URL。

步骤3 项目中引入devspore-probe依赖，在项目的pom.xml中添加下面的配置。

```
<dependency>
   <groupId>com.huaweicloud.devspore</groupId>
   <artifactId>devspore-probe</artifactId>
   <version>lastest</version>
</dependency>
```

步骤4 项目配置文件application.yaml或者application.properties添加配置参数，支持通过AS/SK认证的方式将监控数据上报到AOM。

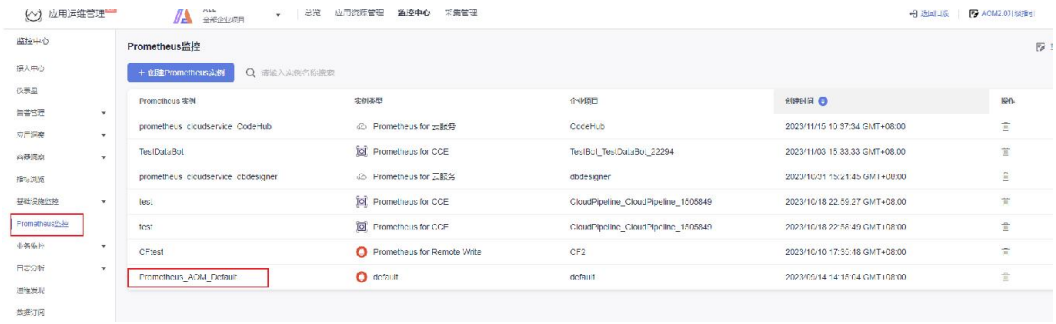
```
devspore:
  probe:
    metrics:
      tags:
        component: probe-test
    aom:
      authentication: AccessKey
      url:
      ak:
      sk:
```

----结束

通过 AccessCode 认证将监控数据上报 AOM

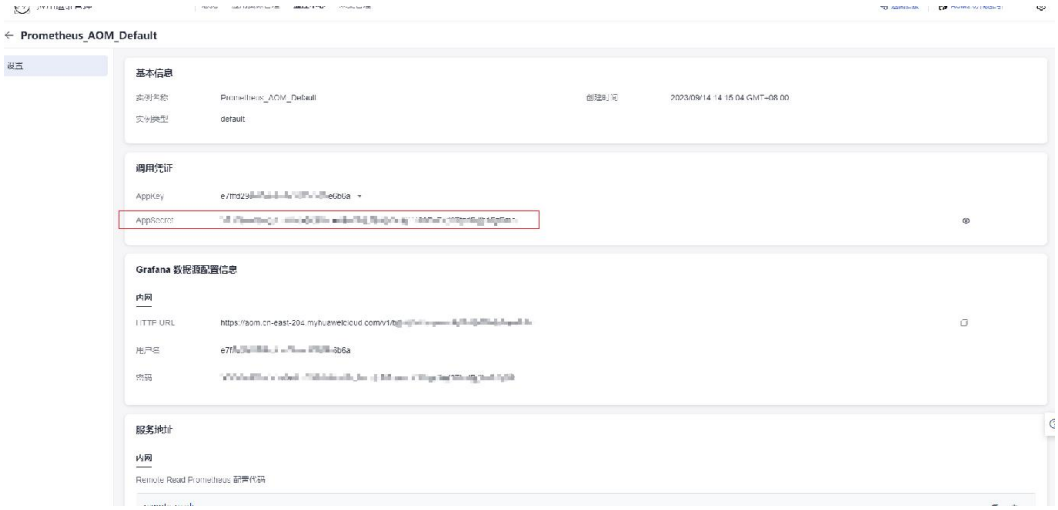
步骤1 登录AOM2.0，单击“Prometheus监控”，进入一个Prometheus实例。

图 2-4 进入实例



步骤2 在调用凭证中获取AccessCode。

图 2-5 获取 AccessCode



步骤3 获取AOM监控上报地址。

图 2-6 上报地址

步骤4 项目中引入devspore-probe依赖，在项目的pom.xml中添加下面的配置。

```
<dependency>
    <groupId>com.huaweicloud.devspore</groupId>
    <artifactId>devspore-probe</artifactId>
    <version>lastest</version>
</dependency>
```

步骤5 项目配置文件application.yaml或者application.properties添加配置参数，支持通过AS/SK认证的方式将监控数据上报到AOM。

```
devspore:
  probe:
    metrics:
      tags:
        component: probe-test
    aom:
      authentication: AccessCode
      url:
      access-code:
```

----结束

2.3.4 配置说明

devspore-probe 常用的配置参数如下所示

```
devspore:
  probe:
    metrics:
      tags:
        component: devspore-application          # 应用名称，使监控指标带上应用标签
      api-description-tags: false                #是否开启http_server_request指标添加description标签功能
    aom:
      enable: false                            # 是否开启监控数据上报AOM平台功能
      authentication：AccessCode                 # 监控数据上报AOM的鉴权方式，可选值：AccessCode，
AccessKey
      ak: ${aom_ak}                            # 用户AK
      sk: ${aom_sk}                            # 用户SK
      url: ${aom_url}                          # 监控数据上报AOM平台接口URL
      access-code：${aom_accessCode}             # 选择AccessCode配置
      push-interval-seconds: 60                # 监控数据上报AOM平台的采集间隔，单位：秒
      pattern:                                 # 符合该正则表达式的监控项将被上报到AOM，默认上报所有的监
控项
      external-env-info: false                 # 是否在监控数据中加入serviceStage环境变量
      batch-size：60                           # 每批次上传AOM平台的Body包包含的监控项数量，默认值60
```

参数说明

表 2-7 参数说明

参数名称	默认值	类型	被监控组件或应用名称
devspore.probe.metrics.tags.component	-	String	被监控组件或应用名称。
devspore.probe.metrics.api-description-tags	false	boolean	是否开启http_server_request指标添加description标签功能，如开启，会自动添加@ApiOperation注解的value字段或@Operation注解的summary字段作为description值。
devspore.probe.aom.enable	false	boolean	是否开启监控数据上报AOM平台功能。
devspore.probe.aom.authentication	-	String	监控数据上报AOM的鉴权方式，可选值：AccessCode，AccessKey。当选择AccessCode时，url、access-code必须配置；当选择AccessKey时，url、ak、sk必须配置。
devspore.probe.aom.ak	-	String	用户AK。
devspore.probe.aom.sk	-	String	用户SK。
devspore.probe.aom.access-code	-	String	用户AccessCode。

参数名称	默认值	类型	被监控组件或应用名称
devspore.probe.aom.apig-app-code	-	String	被授权服务的appCode、非必须使用，非内部用户使用AccessCode鉴权方式的时候需要配置此配置。
devspore.probe.aom.url	-	String	监控数据上报AOM平台接口URL。
devspore.probe.aom.push-interval-seconds	60	int	监控数据上报AOM平台的采集间隔，单位：秒。
devspore.probe.aom.pattern	/w*	String	符合该正则表达式的监控项将被上报到AOM，默认上报所有的监控项。
devspore.probe.aom.external-env-info	false	boolean	是否在监控数据中加入serviceStage环境变量。
devspore.probe.aom.batch-size	false	int	每批次上传AOM平台的Body包包含的监控项数量，默认值60。

2.3.5 常见问题

问题描述

使用devspore-probe并开启上报监控数据到AOM后频繁打印日志：post metrics to aom success。

解决方案

在项目中添加配置参数：logging.level.com.huawei.devspore.probe=error，使得devspore-probe仅打印error日志。

2.4 devspore-clientcontrol 使用指南

2.4.1 devspore-clientcontrol 使用概述

本文主要描述如何通过devspore-clientcontrol在代码中使用超时、重试、熔断和降级的功能。

开发能力要求

您需具备以下开发能力：

- 熟悉Java语言，并有Java程序开发经验。
- 熟悉Maven。

基础开发环境准备

安装的工具包括JDK、Maven、Eclipse和IDEA ，配置对应的环境变量，确保本地开发环境可用。

常用概念

- 超时：超过指定时间还未返回指定结果的任务。
- 重试：对于失败的任务，在一定规则内重复执行。
- 熔断：当任务失败比例达到一定要求后，会触发熔断，对于后续调用请求，不再继续调用目标任务，直接返回，快速释放资源，等到该目标任务情况好转再恢复调用。
- 降级：当任务失败后，执行的一种补偿任务。
- 性能缓存：存入缓存的时间到获取缓存的时间差值小于设定阈值的缓存。

开发流程

1. 通过Maven引入需要的依赖。

```
<dependency>
   <groupId>com.huaweicloud.devspore</groupId>
   <artifactId>devspore-clientcontrol</artifactId>
   <version>${devspore-clientcontrol.version}</version> #版本号，版本号可参考2.6 AstroPro-SDK版本变更与下载。
</dependency>
```

2. 根据业务需求编写配置文件。

```
devspore:
  client-control:
    biz-pool: # 具体可参见 2.4.4 参数配置说明
    caches:  # 具体可参见 2.4.4 参数配置说明
    rules: # 具体可参见 2.4.4 参数配置说明
```

3. 在业务代码中引入@ClientControl注解。

```
@ClientControl(rule = "timeLimitTest")
public String testTimelimit(long sleepTime) {
   return "demo";
}
```

2.4.2 接入指南

1. pom文件添加依赖。

```
<dependency>
   <groupId>com.huaweicloud.devspore</groupId>
   <artifactId>devspore-clientcontrol</artifactId>
   <version>${devspore-clientcontrol.version}</version> #版本号，版本号可参考2.6 AstroPro-SDK版本变更与下载。
</dependency>
```

2. 配置文件示例。

```
devspore:
  client-control:
    biz-pool: #（线程池配置，当使用超时功能时需要配置此参数，多个超时规则共用一个线程池）
      prioritized: false # 线程池队列是否采用优先队列，当消息有优先级时需要设置: true: 优先队列 false: 非优先队列
      core-pool-size: 20 # 线程池核心线程数
      maximum-pool-size: 32 # 线程池所能容纳的最大线程数
      keep-alive-time-ms: 60000 # 线程池任务队列大小
      work-queue-size: 5000 # 非核心线程闲置时的超时时长
      rejectedExecutionHandlerClassName: java.util.concurrent.ThreadPoolExecutor$AbortPolicy # 线程池的拒绝策略（类的全路径限定名）
    caches:
      demo-caches: # 自定义的缓存策略，可定义多个
      ttl: 60000 # 缓存时长，默认6小时
```

```
    performance-ttl: 30000 # 性能缓存时长，默认10分钟
    type: caffeine # 缓存类型，可选值：redis/caffeine
    redis-connection-factory-bean-name: redisConnectionFactory # 缓存类型为redis时可通过配置该选项指定RedisConnectionFactory的bean名称来选择所用数据源 (type为redis时使用)
    maximum-size: 60000 # caffeine缓存最大数量，默认60000（ type为caffeine时使用 ）
    cache-consecutive-exception-times: 3 # 缓存连续异常次数，默认3次 (达到指定次数后关闭缓存的功能)
    cache-interruption-duration: 60 # 缓存功能关闭时长，单位s，默认60 (达到指定时间后恢复缓存的功能)
  rules:
    demo-rules: # 自定义的流控规则，可定义多个
      time-limit:
        enable: false # 超时功能开关, 默认开启
        timeout-duration: 1000 # 限时时长 默认1000ms
    cancel-running-future: true # 超时是否结束当前任务 默认true
      retry:
        enable: false # 重试开关，默认开启
    enable-log: true  # 重试日志开关，默认开启
    log-correlation-property: demo # 可取出header或attributes中相应字段值打印到重试日志中(无需要可配置为空字符串 "")
    max-attempts: 3 # 最大重试次数，默认3
    wait-duration-ms: 500 # 重试间隔，默认500（ ms ）
    policy: Randomized # 可选值Exponential/Randomized（ 重试策略，Exponential指数级/Randomized固定间隔（ 默认Randomized ））
    randomized-wait-factor: 0.5 # 随机策略随机因子，默认0.5
    exponential-backoff-multiplier: 2 # 指数退避算法的乘数，默认2.0
    ignore-exceptions: # 不触发重试的异常，此项配置在超时启用时失效
      - java.awt.AWTException
      - javax.security.auth.login.AccountExpiredException
    retry-exceptions: # 不触发重试的异常，此项配置在超时启用时失效
      - java.awt.AWTException
      - javax.security.auth.login.AccountExpiredException
      fallback:
        enable: true # 熔断开关, 默认开启
    enable-log：true # 熔断状态切换日志开关
        slow-call-duration-threshold: 1 # 计入慢调用的时间，单位 s
        slow-call-rate-threshold: 20 # 慢调用百分比(100为关闭慢调用熔断功能)（ 慢调用计算窗口和失败的窗口是独立计算的两个窗口 ）
        failure-rate-threshold: 50 # 滑动窗口内失败百分比，默认50
        sliding-window-type: COUNT_BASED #     滑动窗口类型（ COUNT_BASED次数/TIME_BASED时间 ）
        sliding-window-size: 5 # 滑动窗口大小（ 次/秒 ）
        minimum-number-of-calls: 5 # 滑动窗口内最小请求数（ 滑动窗口内满足最小请求数才会触发熔断 ）
        wait-duration-in-open-state: 10000 # 进入半开所需时间（ 断路器打开以后进入半开状态等待时间 ）
        permitted-number-of-calls-in-half-open-state: 5 #半开状态允许通过的请求数量，默认10个请求（ 失败比例达到设置的百分比，断路器继续打开，再次等待进入半开 ）注：不大于滑动窗口内最小请求数，相对较小的配置优先起作用，所以如果大于滑动窗口最小请求，起作用的就是滑动窗口最小请求数了
    ignore-exceptions: # 不触发熔断的异常，此项配置在超时启用时失效
      - java.awt.AWTException
      - javax.security.auth.login.AccountExpiredException
    retry-exceptions: # 指定触发熔断的异常，此项配置在超时启用时失效
      - java.awt.AWTException
      - javax.security.auth.login.AccountExpiredException
```

3. 代码中添加@Clientcontrol注解。

```
@ClientControl(rule = "demo-rules")
public String testTimelimit(long sleepTime) {
  return "test";
}
```

2.4.3 使用场景

2.4.3.1 使用超时功能

clientcontrol提供了超时功能，当目标业务方法执行超过执行时间后，会终止代码的执行，具体使用方式如下：

1. **pom文件添加依赖**。
2. 配置文件，参考下面配置样例。

```
devspore:
  client-control:
    biz-pool:
      prioritized: false
      core-pool-size: 20
      maximum-pool-size: 32
      keep-alive-time-ms: 60000
      work-queue-size: 5000
      rejectedExecutionHandlerClassName: java.util.concurrent.ThreadPoolExecutor$AbortPolicy #此处配置的线程池的拒绝策略
    rules:
      timeLimitTest:  # 此处配置的是一个别名，用户可自定义，具体使用地方是在注解上
        match: ^.*$
        module: RetryFirst
        time-limit:
          enable: true
          timeout-duration: 3000  # 此处配置的是超时时间
          cancel-running-future: true
        retry:
          enable: false  # 重试功能默认开启，当只使用超时功能时，此处手动关闭
        fallback:
          enable: false  # 熔断功能默认开启，当只使用超时功能时，此处手动关闭
```

3. 在目标方法上添加@ClientControl注解，且rule属性指定用户在配置文件中自定义的rules名称（本示例使用timeLimitTest）。

```
// 在想要使用超时功能的方法上添加 @ClientControl注解， 属性值rule 填写配置文件中自定义的名称
@ClientControl(rule = "timeLimitTest")
public String testTimelimit(long sleepTime) {
    try {
        Thread.sleep(sleepTime);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    return "测试超时";
}
```

注意事项

- 当使用超时功能时，方法会运行在异步线程中，如在方法中使用了线程变量，会产生无法读取变量的问题。
- 使用超时功能时必须配置线程池devspore:client-control:biz-pool，具体参数配置请参考**表2-8**。

2.4.3.2 使用重试功能

clientcontrol提供了重试功能，当目标业务方法执行报错时，会按照用户配置的策略重试目标业务方法，具体使用方式如下：

步骤1 **pom文件添加依赖**。

步骤2 配置文件，参考下面配置样例。

```
devspore:
  client-control:
    rules:
      retryTest:  # 此处配置的是一个别名，用户可自定义，具体使用地方是在注解上
        match: ^.*$
```

```
      module: RetryFirst
      time-limit:  # 超时功能默认开启，当只使用重试功能时，需要手动关闭超时功能
        enable: false
      retry:
        enable: true
        max-attempts: 3 # 重试次数，算上第一次执行的时候
        wait-duration-ms: 500 # 每次重试的间隔时间
        policy: Randomized  #此处配置的是重试间隔策略，可选值（Exponential/Randomized）
        randomized-wait-factor: 0.5  #随机策略随机因子(选择随机策略生效：基础间隔500，每次间隔为：
500-500*0.5~500+500*0.5)
        exponential-backoff-multiplier: 2 #指数退避算法的乘数(选择指数策略生效：基础间隔500，每次间隔
为：500，1000，2000.....)
        #不触发重试异常
        ignore-exceptions:
          - com.huaweicloud.devspore.clientcontrol.exception.TestException
      fallback: # 熔断功能默认开启，当只使用重试功能时，需要手动关闭超时功能
        enable: false
```

步骤3 在目标方法上添加@ClientControl注解，且rule属性指定用户在配置文件中自定义的rules名称（本示例使用retryTest）。

```
// 在想要使用超时功能的方法上添加 @ClientControl注解， 属性值rule 填写配置文件中自定义的名称
@ClientControl(rule = "retryTest")
public User testRetry(Integer id) {
    int i = 1 / id;
    return new User(id, "vn", 12);
}
```

----结束

注意事项

- 当使用超时功能时，当前方法不能运行在异步线程中，异步会导致超时功能失效。
- 触发重试的机制是方法抛出异常，不管是什么异常，只要抛出了异常方法就会重试，如果抛出某些异常的时候不想重试，则可以参考配置中的ignore-exceptions，没有的时候此参数可以不配置。
- 配置中的policy中在提供的两种方式中选择一个即可，当使用Randomized的时候，配置randomized-wait-factor。 当使用Exponential的时候配置exponential-backoff-multiplier。

2.4.3.3 使用熔断降级功能

2.4.3.3.1 缓存优先

clientcontrol提供了一种名为缓存优先的功能。当使用缓存优先功能时，在执行目标业务代码时，优先会查询缓存，如果缓存存在则返回缓存结果，如果缓存不存在则执行目标业务代码。

当使用缓存优先时clientcontrol的处理逻辑如下：

- 读取缓存，判断是否是性能缓存。
  - 如果是性能缓存，直接将性能缓存当结果返回。
  - 如果不是性能缓存，执行原方法。
- 如果原方法执行成功，更新缓存，返回结果。 如果原方法执行失败，返回获取的缓存。
- 如果一开始没有获取到缓存，会直接将原方法执行失败的异常返回。

性能缓存是clientcontrol自带的一个概念，对应配置参数中的performance-ttl。例如一个缓存的有效时长是10s，那性能缓存可以配置为3s，代表一个逻辑概念，标识缓存的结果离更新缓存的时间更近，不会对实际的缓存产生影响。

具体使用方式如下：

步骤1 **pom文件添加依赖**。

步骤2 配置文件，参考下面配置样例。

```
devspore:
  client-control:
    caches:
      test:
        ttl: 60000  #此处配置的是缓存的有效时长
        performance-ttl: 30000  #此处配置的是一个性能缓存，时长一般低于ttl， 当缓存的时间小于性能缓存时，clientcontrol会直接把性能缓存作为方法返回值返回
        type: redis  此处配置的是 缓存类型， 支持 redis/caffine 指定具体缓存类型后，用户需要手动引入相关的依赖
        maximum-size: 60000
        redis-connection-factory-bean-name: redisConnectionFactory  # 当使用redis的时候，需要将redis的连接工厂的bean的名称配置在这里
    rules:
      fallbackTest: # 此处配置的是一个别名，用户可自定义，具体使用地方是在注解上
        time-limit:
          enable: false
        retry:
          enable: false
        fallback:
          # 默认开启
          enable: true
          # 慢调用时间（超过即为慢调用，单位s，默认60S）
          slow-call-duration-threshold: 30
          # 慢调用熔断比例（慢调用数量达到比例则熔断，默认100等于关闭状态）
          slow-call-rate-threshold: 100
          #失败百分比（触发断路器，默认50%）
          failure-rate-threshold: 50
          #滑动窗口类型（COUNT_BASED/TIME_BASED,数量/时间，默认时间）
          sliding-window-type: COUNT_BASED
          #滑动窗口大小(默认100，数量：次/时间：秒)
          sliding-window-size: 5
          #滑动窗口内最小请求数(默认100)  必须满足这个要求，才会触发断路器  不满足，不管失败率多少都不会触发
          minimum-number-of-calls: 5
          #进入半开所需时间(默认60s，单位ms)
          wait-duration-in-open-state: 10000
          #半开状态允许通过的请求数量，默认10个请求（失败比例达到设置的百分比，断路器继续打开，再次等待进入半开）注：不大于滑动窗口内最小请求数，相对较小的配置优先起作用，所以如果大于滑动窗口最小请求，起作用的就是滑动窗口最小请求数了
          permitted-number-of-calls-in-half-open-state: 5
```

步骤3 目标方法上添加@ClientControl注解，且policy属性设置为CacheOrder.CACHEFIRST，rule属性选择配置文件中自己定义的rules名称（本示例中使用fallbackTest），cacheManagerName属性选择配置文件中自己定义的caches的名称（本示例使用test）。

```
@ClientControl(rule = "fallbackTest", policy = CacheOrder.CACHEFIRST, cacheManagerName = "test")
public String testClientControlJiangji(Integer id) {
    int i = 1 / id;
    return new User(id, "vn", 12).toString();
}
```

----结束

2.4.3.3.2 服务优先

clientcontrol提供了一种名为服务优先的功能。当使用服务优先功能时，在执行目标业务代码时，优先执行业务代码并返回，如果业务代码报错则去查询缓存并返回。具体使用方式如下：

步骤1 **pom文件添加依赖**。

步骤2 配置文件，参考下面配置样例。

```
devspore:
  client-control:
    caches:
      test:
        ttl: 60000  #此处配置的是缓存的有效时长
        performance-ttl: 30000  #此处配置的是一个性能缓存，时长一般低于ttl， 当缓存的时间小于性能缓存时，clientcontrol会直接把性能缓存作为方法返回值返回
        type: redis  此处配置的是 缓存类型， 支持 redis/caffine 指定具体缓存类型后，用户需要手动引入相关的依赖
        maximum-size: 60000
        redis-connection-factory-bean-name: redisConnectionFactory  # 当使用redis的时候，需要将redis的连接工厂的bean的名称配置在这里
    rules:
      fallbackTest: # 此处配置的是一个别名，用户可自定义，具体使用地方是在注解上
        time-limit:
          enable: false
        retry:
          enable: false
        fallback:
          # 默认开启
          enable: true
          # 慢调用时间（超过即为慢调用，单位s，默认60S）
          slow-call-duration-threshold: 30
          # 慢调用熔断比例（慢调用数量达到比例则熔断，默认100等于关闭状态）
          slow-call-rate-threshold: 100
          #失败百分比（触发断路器，默认50%）
          failure-rate-threshold: 50
          #滑动窗口类型（COUNT_BASED/TIME_BASED,数量/时间，默认时间）
          sliding-window-type: COUNT_BASED
          #滑动窗口大小(默认100，数量：次/时间：秒)
          sliding-window-size: 5
          #滑动窗口内最小请求数(默认100)  必须满足这个要求，才会触发断路器  不满足，不管失败率多少都不会触发
          minimum-number-of-calls: 5
          #进入半开所需时间(默认60s，单位ms)
          wait-duration-in-open-state: 10000
          #半开状态允许通过的请求数量，默认10个请求（失败比例达到设置的百分比，断路器继续打开，再次等待进入半开）注：不大于滑动窗口内最小请求数，相对较小的配置优先起作用，所以如果大于滑动窗口最小请求，起作用的就是滑动窗口最小请求数了
          permitted-number-of-calls-in-half-open-state: 5
```

步骤3 目标方法上添加@ClientControl注解，且policy属性设置为CacheOrder.SERVICECALLFIRST，rule属性选择配置文件中自己定义的rules名称（本示例中使用fallbackTest），cacheManagerName属性选择配置文件中自己定义的caches的名称（本示例使用test）。

```
@ClientControl(rule = "fallbackTest", policy = CacheOrder.SERVICECALLFIRST, cacheManagerName = "test")
public String testClientControlJiangji(Integer id) {
    int i = 1 / id;
    return new User(id, "vn", 12).toString();
}
```

当使用服务优先时clientcontrol的处理逻辑如下：

- 执行原方法，如果执行成功，更新缓存，返回结果。如果执行方法失败，获取缓存。

- 如果成功拿到缓存，则将缓存结果返回。如果没有拿到缓存则将原方法的执行失败异常返回。

----结束

2.4.3.3.3 自定义降级

clientcontrol提供了自定义降级的功能。当目标业务代码报错时，如果用户想要执行自定义的降级逻辑可以参考以下配置。具体使用方式如下：

步骤1 **pom文件添加依赖**。

步骤2 配置文件，参考下面配置样例。

```
devspore:
  client-control:
    caches:
      test:
        ttl: 60000  #此处配置的是缓存的有效时长
        performance-ttl: 30000  #此处配置的是一个性能缓存，时长一般低于ttl， 当缓存的时间小于性能缓存时，clientcontrol会直接把性能缓存作为方法返回值返回
        type: redis  此处配置的是 缓存类型， 支持 redis/caffine 指定具体缓存类型后，用户需要手动引入相关的依赖
        maximum-size: 60000
        redis-connection-factory-bean-name: redisConnectionFactory  # 当使用redis的时候，需要将redis的连接工厂的bean的名称配置在这里
    rules:
      fallbackTest: # 此处配置的是一个别名，用户可自定义，具体使用地方是在注解上
        time-limit:
          enable: false
        retry:
          enable: false
        fallback:
          # 默认开启
          enable: true
          # 慢调用时间（超过即为慢调用，单位s，默认60S）
          slow-call-duration-threshold: 30
          # 慢调用熔断比例（慢调用数量达到比例则熔断，默认100等于关闭状态）
          slow-call-rate-threshold: 100
          #失败百分比（触发断路器，默认50%）
          failure-rate-threshold: 50
          #滑动窗口类型（COUNT_BASED/TIME_BASED,数量/时间，默认时间）
          sliding-window-type: COUNT_BASED
          #滑动窗口大小(默认100，数量：次/时间：秒)
          sliding-window-size: 5
          #滑动窗口内最小请求数(默认100)  必须满足这个要求，才会触发断路器  不满足，不管失败率多少都不会触发
          minimum-number-of-calls: 5
          #进入半开所需时间(默认60s，单位ms)
          wait-duration-in-open-state: 10000
          #半开状态允许通过的请求数量，默认10个请求（失败比例达到设置的百分比，断路器继续打开，再次等待进入半开）注：不大于滑动窗口内最小请求数，相对较小的配置优先起作用，所以如果大于滑动窗口最小请求，起作用的就是滑动窗口最小请求数了
          permitted-number-of-calls-in-half-open-state: 5
```

步骤3 目标方法上添加@ClientControl注解，fallback属性设置为自定义降级方法的名称。

```
@ClientControl(fallback = "customFallback")
public String testCustomDowngrade(Integer id) {
    int i = 1 / id;
    return new User(id, "test", 12).toString();
}

// 自定义降级方法需要和原方法返回值相同，参数列表在原方法的基础上多添加一个 Throwable throwable（固定写法），降级方法要和原方法在同一个类中。
public String customFallback(Integer id, Throwable throwable) {
    log.info("------------ 执行自定义降级方法 customFallback ----------------");
```

```
    return "自定义降级";
}
```

----结束

注意事项

- 当只需要使用自定义降级方法的时候，无需关注配置。
- 当结合clientcontrol的其他功能使用时，只需要在注解中添加fallback属性就可以了。

```
@ClientControl(fallback = "customFallback")
public String testCustomDowngrade(Integer id) {
    int i = 1 / id;
    return new User(id, "test", 12).toString();
}
```

2.4.4 参数配置说明

表 2-8 参数配置说明

参数名称	是否必选	参数类型	取值范围	描述
bizPool	否	ThreadPoolConfig	参考表2-9。	线程池配置。
caches	否	Map of CacheConfig	key为userdefine1，userdefine2..... 单个维度请参考表2-10。	多个缓存配置。
rules	否	Map of ClientProperties	key为userdefine1，userdefine2..... 单个维度请参考表2-11。	多个熔断和重试。

表 2-9 ThreadPoolConfig 数据结构说明

参数名称	是否必选	参数类型	默认值	取值范围	描述
prioritized	否	boolean	false	true/false	线程池队列是否采用优先队列，当消息有优先级时需要设置： • true: 优先队列。 • false：非优先队列。
core-pool-size	否	Interger	20	-	线程池核心线程数。
maximum-pool-size	否	Interger	32	-	线程池所能容纳的最大线程数。

参数名称	是否必选	参数类型	默认值	取值范围	描述
work-queue-size	否	Interger	60000L	-	线程池任务队列大小。
keep-alive-time-ms	否	Long	5000	-	非核心线程闲置时的超时时长。
rejected-execution-handler-class-name	否	String	无	java.util.concurrent.ThreadPoolExecutor$CallerRunsPolicy	线程池的拒绝策略。

表 2-10 CacheConfig 数据结构说明

参数名称	是否必选	参数类型	取值范围	默认值	描述
ttl	否	Long	-	21600	缓存时长（单位：秒），默认6小时。
performance-ttl	否	Long	-	600	性能缓存时长（单位：秒），默认10分钟。
type	否	CacheType	redis/caffeine	nocache	缓存类型。
redis-connection-factory-bean-name	否	String	-	clientcontrolDcs	缓存类型为redis时可通过配置该选项指定。RedisConnectionFactory的bean名称来选择所用数据源，默认使用dcs的默认配置。
maximum-size	否	Long	-	60000L	caffeine缓存最大数量，默认60000。
cache-consecutive-exception-times	否	Integer	-	3	缓存连续异常次数，默认3次。
cache-interruption-duration	否	Long	-	60	缓存功能关闭时长，单位（秒），默认60。

参数名称	是否必选	参数类型	取值范围	默认值	描述
custom-cache-manager-bean-name	否	String	-	customCacheManager	当缓存类型为custom时需要指定该值。

表 2-11 ClientProperties 数据结构说明

参数名称	是否必选	参数类型	取值范围	描述
match	否	String	-	正则表达式，用于与requestMapping中的url匹配。
module	否	FallbackOrder	RetryFirst/CircuitBreakerFisrt二选一的枚举类。	RetryFirst重试次数不计入断路器/CircuitBreakerFisrt重试次数也计入断路器（默认CircuitBreakerFisrt）。
time-limit	否	TimeLimit	参考**表2-12**。	限时配置。
retry	否	Retry	参考**表2-13**。	重试的配置。
fallback	否	Fallback	参考**表2-14**。	断路器配置。

表 2-12 TimeLimit 数据结构说明

参数名称	是否必选	参数类型	取值范围	默认值	描述
enable	否	boolean	true/false	true	超时开关。
timeout-duration	否	Duration	-	1000	限时时长（单位：毫秒）默认1000ms。
cancel-running-future	否	boolean	true/false	true	超时是否结束当前任务，默认true。

表 2-13 Retry 数据结构说明

参数名称	是否必选	参数类型	取值范围	默认值	描述
enable	否	boolean	true/false	true	重试开关，默认开启。
enable-log	否	boolean	true/false	true	重试日志开关，默认开启。
log-correlation-property	否	String	-	Client Control	可取出header或attributes中相应字段值打印到重试日志中。
max-attempts	否	Long	-	3	最大重试次数，默认为3次。
wait-duration-ms	否	Long	-	500	重试间隔，单位（毫秒），默认500（ms）。
policy	否	String	Exponential/Randomized	Randomized	重试策略，Exponential指数级/Randomized固定间隔（默认Randomized）。
randomized-wait-factor	否	double	0.0-1.0	0.5	随机策略随机因子，默认0.5。
exponential-backoff-multiplier	否	double	-	2	指数退避算法的乘数，默认2.0。
ignore-exceptions	否	Class<? extends Throwable>	-	-	不触发重试的异常，此项配置在超时启用时失效。
retry-exceptions	否	Class<? extends Throwable>	-	-	指定触发重试的异常。

表 2-14 Fallback 数据结构说明

参数名称	是否必选	参数类型	取值范围	默认值	描述
enable	否	boolean	true/false	true	熔断开关。

参数名称	是否必选	参数类型	取值范围	默认值	描述
enable-log	否	boolean	true/false	true	熔断状态切换日志开关。
slow-call-duration-threshold	否	Long	-	60（单位 s）	计入慢调用的时间（单位：秒）。
slow-call-rate-threshold	否	Long	-	100	慢调用百分比(100为关闭慢调用熔断功能)（慢调用计算窗口和失败的窗口是独立计算的两个窗口）。
failure-rate-threshold	否	Long	0-100	50	滑动窗口内失败百分比，默认50。
sliding-window-type	否	String	COUNT_BASED/ TIME_BASED	TIME_BASED	滑动窗口类型（COUNT_BASED次数/TIME_BASED时间）。
sliding-window-size	否	Long	-	30	滑动窗口大小（次/秒）。
minimum-number-of-calls	否	Long	-	100	滑动窗口内最小请求数（滑动窗口内满足最小请求数才会触发熔断）。
wait-duration-in-open-state	否	Duration	-	60000（单位 ms）	进入半开所需时间（断路器打开以后进入半开状态等待时间）。
permitted-number-of-calls-in-half-open-state	否	Long	-	10	半开状态允许通过的请求数量（如果失败比例达到设置的百分比，则断路器继续进入打开状态，再次等待进入半开）。
ignore-exceptions	否	Class<? extends Throwable>	-	-	不触发熔断的异常，此项配置在超时启用时失效。
record-exceptions	否	Class<? extends Throwable>	-	-	指定触发熔断的异常类。

2.4.5 常见问题

2.4.5.1 clientcontrol 中 ttl 和 performanceTtl 的具体含义是什么

Cache配置有两个ttl，一个ttl（可靠性缓存）和一个performanceTtl（性能缓存）。

存在误解这两个为是两个缓存，其实缓存只有一个。

实际给缓存设置的缓存过期时间为ttl配置，缓存存入的时候，连同当前时间一同存入了，取出的时候就可以取出缓存存入时间，通过计算与performanceTtl进行比较得出是否是性能缓存。

2.4.5.2 无法获取本地线程变量

问题描述

ClientControl的超时功能是搭配线程池使用的，所以当使用本地线程变量的时候会导致无法获取线程变量的问题。

解决方案

- 这种情况一般推荐使用远程调用自带的超时控制，将ClientControl的超时功能关闭。
- 提前获取需要的信息，传入加了@ClientControl注解的方法中。

2.4.5.3 redis 缓存不存在的属性反序列化

问题描述

redis在存入的时候，如果对象中存在is开头的方法，但是又没有具体的属性。那么在存入的时候，会将auth转换为一个属性存入缓存当中，在反序列化的时候就会报反序列化失败。

```
public boolean isAuth(){
    return false;
}
```

解决方案

在is方法上面添加@JsonIgnore注解。

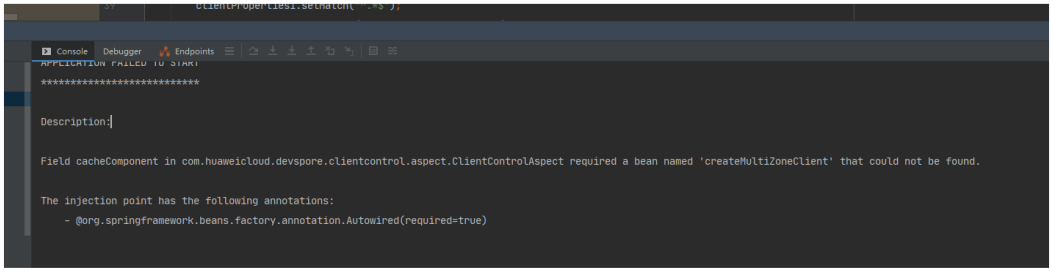
```
@JsonIgnore
public boolean isAuth(){
    return false;
}
```

2.4.5.4 找不到 createMultiZonClient 的 bean

问题描述

当使用缓存功能，并且缓存组件使用的redis时，报如图所示的异常。

图 2-7 报错异常



解决方案

- 使用devspore-dcs连接redis。

  devspore-dcs会默认创建一个bean名称为createMultiZoneClient的redisConnectionFactory，如果用户没有手动修改这个bean名称的话可以不配置devspore.client-control.caches.xxx.redis-connection-factory-bean-name属性，clientcontrol会自动去寻找createMultiZoneClient的bean。

- 没有使用devspore-dcs。

  此时用户需要手动设置devspore.client-control.caches.xxx.redis-connection-factory-bean-name属性，如果不设置，clientcontrol会自动去寻找createMultiZoneClient的bean，因为没有使用devspore-dcs，所以并不存在bean名称为createMultiZoneClient的redisConnectionFactory，此时就会报如上的图的异常。

2.4.5.5 自定义缓存 key 读取失败

问题描述

运行时报错**com.huaweicloud.devspore.clientcontrol.core.ClientControlCacheComponent] [ERROR] - [Expression [xxxxx] @xx: ELxxxx]**。

解决方案

此报错是因为使用了自定义缓存key的功能，注解中配置了@clientcontrol(rules='xxxx', key='xxxxxx')此处key的解析使用的是标准的spring的spel表达式的解析，如果报了如上错误，需要自行排查el表达式的写法是否正确。

如果实在排查不出来el表达式的问题，用户也可以选择使用自定义类的方式去生成指定的缓存key值@ClientControl(rule = "retryTest", policy = CacheOrder.CACHEFIRST, cacheManagerName = "cacheCaffine",keyGenerator = "myKeyGenerator")，其中，keyGenerator属性配置的就是自定义缓存key生成器的bean名称。

如果用户没有必须自定义缓存key的需求，可以不配置这两个属性，clientcontrol会自动生成缓存key的，生成的规则是：全路径限定名+方法名+参数的sha256，clientcontrol提供了一个公共方法来获取这个key值。ClientControlCommonHandle.getCacheKey(final String className, final String methodName, final Object[] args), 此处是伪代码，方法中传入的是全路径限定类名、方法名和参数集合。

2.4.5.6 访问熔断状态的方法时报错异常

问题描述

运行时报错**Request processing failed; nested exception is io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'com.huaweicloud.testclientcontrol.serivce.impl.UserServiceImpl#testFallBack' is OPEN and does not permit further calls**。

解决方案

1. 当程序报出这个错误时，说明使用了熔断功能，且此时方法处于熔断状态（熔断器状态为OPEN）。当访问熔断状态的方法时就会抛出这个异常。
2. 熔断器的状态变化时，clientcontrol会打印出如下的日志：
   ```
   [com.huaweicloud.devspore.clientcontrol.core.ClientControlRegistryComponent] - [ClientControlCircuitBreakerSignal:2024-06-14T10:12:59.268797400+08:00[Asia/Shanghai]: CircuitBreaker 'com.huaweicloud.testclientcontrol.serivce.impl.UserServiceImpl#testFallBack' changed state from CLOSED to OPEN]
   ```
   熔断器有三个状态：
   - CLOSED：这个状态下，方法可以正常访问。
   - HALF_OPEN：这个状态下，方法可以正常访问。
   - OPEN：这个状态下，方法不可以访问。

2.4.5.7 clientcontrol 注解不生效

问题描述

clientcontrol注解不生效。

解决方案

clientcontorl是依托于spring的aop来实现的。当注解不生效时，用户需要排查以下两个问题：

- 版本使用是否正确，如果用户使用的是jdk17、clientcontrol必须使用jdk17的版本，jdk17的版本在版本号中会有JDK17的标识，如 2.1.6.JDK17-RELEASE，不带JDK17的均为jdk8的版本。
- 是否在同一个类中调用了使用@clientcontrol注解的方法， 如果存在这种情况，spring的aop是不会生效的，所以clientcontrol功能不生效。此时用户需要修改代码，将调用到@clientcontrol注解的方法放到其他类中。

2.4.5.8 自定义降级方法不生效

问题描述

自定义降级方法不生效。

解决方案

请排查自定义降级方法的定义的限制。

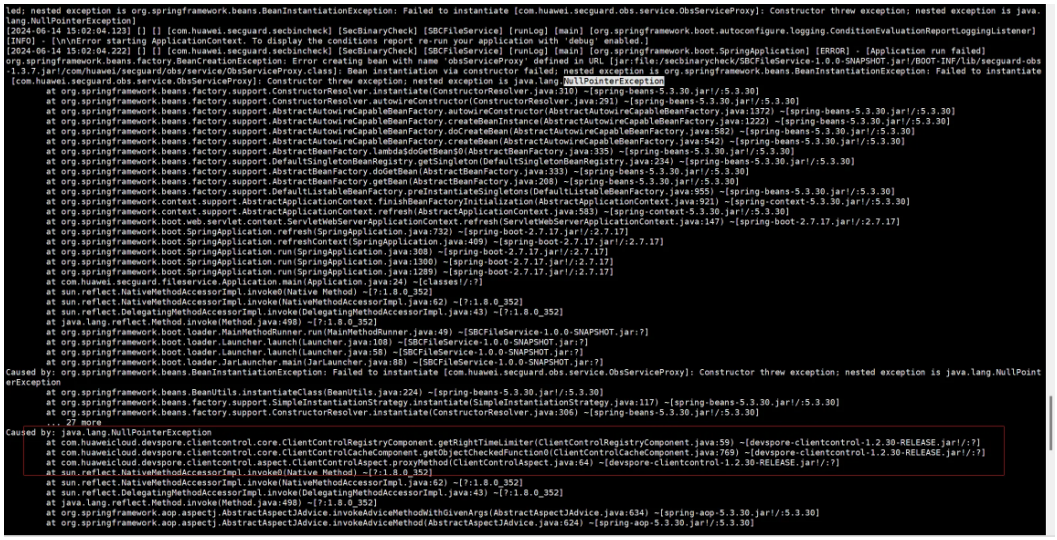
- 方法列表需要和原方法一致，此外要额外添加一个Throwable的参数，放在参数列表的最后一个。
- 方法的返回值要和原方法保持一致。
- 方法的修饰类型要大于原方法。如：原方法是public的，那自定义降级的方法不能小于public。

2.4.5.9 在项目启动时报 NPE 问题

问题描述

在项目启动时，报如图2-8所示问题。

图 2-8 NPE 报错



解决方案

用户需要排查是否在项目启动时使用到了clientcontrol的功能，如在启动时使用了clientcontrol的功能，需要保证clientcontrol要优先初始化。可以在使用到clientcontrol功能的bean上加上@DependsOn("defaultClientControlRules")。

2.4.5.10 熔断器打开，导致方法调用失败

问题描述

熔断器打开，导致方法调用失败。

解决方案

- 在没有使用自定义降级方法和缓存的时候，原方法如果调用失败会抛出方法原始出的错误。
- 如果使用了自定义降级方法。
  - 熔断前：自定义降级方法中添加的最后一个异常的参数封装的就是原方法的异常。

- 熔断后：最后一个异常参数会固定抛出“io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'xxxxx#xxxxx' is OPEN and does not permit further calls”异常。

  可以通过这个异常参数找到熔断的原因，如果日志过大，找不到熔断前的日志，可以对异常参数类型做个判断，将CallNotPermittedException异常和其他异常区别打印，这样更方便定位问题。如下所示：

  ```
  public String myFallBack(long sleepTime, Throwable throwable) {
      if (throwable instanceof CallNotPermittedException) {
          log.info("clientcontrol call not permited exception ", throwable);
      } else {
          log.info("service exception " , throwable);
      }
      return "xxxx";
  }
  ```

  熔断器的状态变化会打印相应的日志，日志中关键信息如下：

  - 关闭到打开（此时处于熔断状态，方法不可访问）：changed state from CLOSED to OPEN。
  - 打开到半开（方法可以访问）：changed state from OPEN to HALF_OPEN。
  - 半开到打开（此时处于熔断状态，方法不可访问）：changed state from HALF_OPEN to OPEN。
  - 半开到关闭（方法恢复正常，可以访问）：changed state from HALF_OPEN to CLOSED。

  如果日志中找不到原方法的报错信息，只能找到“io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'xxxxx#xxxxx' is OPEN and does not permit further calls”这种日志，需要排查clientcontrol的配置文件，看是否开启了慢调用熔断，如果开启了慢调用熔断，会出现原方法没有报出异常，也被熔断的现象。慢调用的配置参数如下：

  ```
  devspore:
    client-control:
      rules:
        xxx:
          fallback:
            #慢调用时间（超过即为慢调用，单位s，默认60S）
            slow-call-duration-threshold: 60
            #慢调用熔断比例（慢调用数量达到比例则熔断，默认100等于关闭状态）
            slow-call-rate-threshold: 100
  ```

2.4.5.11 redis 故障后 clientcontrol 的处理逻辑是什么

当clientcontrol的缓存类型配置为redis时，此时clientcontrol相当于是redis的使用方，redis自身的配置（超时时间，重试等）还是服务自己设置的，跟clientcontrol没有关系。

clientcontrol中有两个配置参数**cache-consecutive-exception-times**和**cache-interruption-duration**，具体查看：**4.5.5CacheConfig数据结构说明**。

- cache-consecutive-exception-times：记录缓存连续失败的次数上限（此上限是针对配置的一个缓存策略，devspore.client-control.caches.xxx）（读失败，写失败都会累加这个次数，当调用读成功或写成功一次后，会将这个参数置0），当达到这个上限后clientcontrol会关闭缓存的功能，既不读也不写。

- cache-interruption-duration：此参数是控制关闭缓存的时间，当达到缓存连续失败的次数上限后，clientcontrol会关闭缓存功能这个参数配置的时间，时间到了以后会开启缓存功能。

2.4.5.12 2.1.7.JDK17-RELEASE 之前版本开启重试功能概率性报错

问题描述

2.1.7.JDK17-RELEASE之前版本开启重试功能时，有概率会报“The reuqest object has been recycled and is no longer associated with this facade”错误。

解决方案

- 升级版本到2.1.7.JDK17-RELEASE。
- 将重试参数log-correlation-property设置为空字符串（“”）。

2.5 spring-boot-huawei 使用指南

2.5.1 spring-boot-huawei 使用概述

开发简介

本SDK旨在简化基于spring-boot的项目开发时，对于开发环境依赖jar包的管理问题。

开发能力要求

您需具备以下开发能力：

- 熟悉Java语言，并有Java程序开发经验。
- 熟悉Maven。

基础开发环境准备

安装的工具包括JDK、Maven、Eclipse和IDEA ，配置对应的环境变量，确保本地开发环境可用。

2.5.2 使用指南

2.5.2.1 使用 spring-boot-huawei 组件

spring-boot-huawei是一个管理jar包的SDK，主要帮助用户管理一些常用sdk的依赖版本。

使用方式

当前spring-boot-huawei各功能模块以jar包的方式提供起始依赖，自动加入实现该模块功能的jar依赖，使用时直接在项目的pom中添加对应版本的功能模块即可，以核心模块为例，使用如下：

```
<properties>
    ...
    <com.huawei.version>2.1.9-RELEASE</com.huawei.version>
    ...
<properties>
<dependencies>
    <dependency>
        <groupId>com.huawei.devspore</groupId>
        <artifactId>spring-boot-starter-huawei</artifactId>
        <version>${com.huawei.version}</version>
    </dependency>
    ...
</dependencies>
```

其他模块参见各组件的文档。

源码基本结构说明

spring-boot-huawei内部模块及简介:

```
spring-boot-huawei
 |
 |-spring-boot-huawei-dependencies # 该模块为内部模块的依赖模块，主要功能是提供依赖包的版本管理，其他模块的父模块均直接或间接使用此模块，如无特殊，spring-boot-huawei使用统一版本的jar包。
 |
 |-spring-boot-huawei-parent # 该模块以dependencies模块为父模块，使用其提供的依赖版本管理,本身提供一些插件定义及插件版本管理，作为本项目中其他模块的父模块使用
 |
 |-spring-boot-starter-huawei # 核心组件,该组件提供一个基于spring-boot的web项目最基本的起步依赖
```

2.5.2.2 使用 spring-boot-starter-huawei 组件

spring-boot-starter-huawei组件具备开发一个基本的基于spring-boot的web项目的基本依赖，使用时只需将该组件的坐标添加到工程的pom文件中即可。

使用方式

直接作为dependency依赖引入（此步骤必须）。

```
<properties>
    ...
    <com.huawei.version>2.1.9RELEASE</com.huawei.version>
    ...
<properties>

<dependencies>
    <dependency>
        <groupId>com.huawei.devspore</groupId>
        <artifactId>spring-boot-starter-huawei</artifactId>
        <version>${com.huaweicloud.version}</version>
    </dependency>
</dependencies>
```

版本管理

步骤1 将spring-boot-huawei-parent设置为项目的parent。

```
<parent>
    <groupId>com.huawei.devspore</groupId>
    <artifactId>spring-boot-huawei-parent</artifactId>
    <version>2.1.9-RELEASE</version>
</parent>
```

- spring-boot-huawei本身带有一个模块spring-boot-huawei-dependencies用来管理整个项目的依赖管理。

- spring-boot-huawei项目下其他模块均通过spring-boot-huawei-parent作为父模块间接使用其版本管理功能。
- spring-boot-starter-huawei依赖各个包的版本都受dependencies模块管理，其本身并未对依赖的包的版本显式声明。
- 当spring-boot-starter-huawei作为一个jar包依赖到所需工程时，若该工程中的父模块或当前模块也声明了对相同包的依赖版本，则spring-boot-starter-huawei中声明的版本可能会被覆盖而失效。此时，若需要使用spring-boot-starter-huawei中的版本声明，建议将spring-boot-huawei-dependencies模块作为依赖管理添加至当前pom（单模块工程）或根项目模块（多模块工程）的dependencyManagement中；或者将spring-boot-huawei-parent设置为当前项目的parent（单模块工程）或根项目模块的parent（多模块工程）。

步骤2 将spring-boot-huawei-dependencies加入dependencyManagement。

```
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>com.huawei.devspore</groupId>
         <artifactId>spring-boot-huawei-dependencies</artifactId>
         <version>${com.huawei.version}</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
   ...
</dependencyManagement>
```

同时将父模块或本模块的其他依赖声明删除。

spring-boot-huawei-dependencies中已经声明了对spring-boot-dependencies的依赖管理，故受该依赖管理的包版本也受spring-boot-huawei-dependencies管理。若服务为已有服务，已经声明了spring-boot-dependencies的依赖，则spring-boot-starter-huawei中使用的依赖版本可能会是spring-boot-dependencies中传递的版本。

建议将本项目的spring-boot-dependencies的版本声明为spring-boot-huawei一致，以保证不受spring管理的优选或可选的其他依赖不会有版本或功能冲突。

须知

设置项目中pom的parent为spring-boot-huawei-parent和在项目pom中的dependencyManagement中添加spring-boot-huawei-dependencies这两种方式根据项目实际情况选择一种方式即可，无需同时使用，推荐使用b方式。

----结束

组件说明

为支持及简化项目依赖，spring-boot-starter-huawei中配置了基于springboot的web项目常用的基本功能的依赖支持，目前包含以下常用依赖：

- 基本的spring环境依赖：使用spring-boot-starter完成对基本的spring环境集成，主要包含spring-core，spring-beans，spring-expression，spring-aop，spring-context，spring-aspects，spring-boot，spring-boot-autoconfigure等。
- web项目的依赖：包含web开发所需的spring-web，spring-webmvc模块，处理常用格式json数据所需的jackson-databind等。

- 服务在线文档生成工具：swagger(v2,v3)，springdoc-openapi-ui(jdk8)，springdoc-openapi-starter-webmvc-ui(jdk17)，有了这些工具后可以为服务接口字段生成openapi在线文档。
- yaml文件解析用到的snakeyaml，注解：jakarta.annotation-api。
- 常用工具类的依赖：lombok，commons-lang，guava。

spring-boot-starter-huawei中默认使用log4j2日志框架，需要根据实际情况确定是否使用该日志框架。

2.5.2.3 组件纳管的依赖包和版本

组件纳管的依赖包

- spring-boot-starter-huawei包含的依赖包（代码中pom可不引入相关依赖）。

表 2-15 spring-boot-starter-huawei 包含的依赖包

包名groupId:artifactId	备注
org.springframework:spring-core	-
org.springframework:spring-beans	-
org.springframework:spring-expression	-
org.springframework:spring-aop	-
org.springframework:spring-context	-
org.springframework:spring-web	-
org.springframework:spring-webmvc	-
org.springframework:spring-aspects	-
org.springframework:spring-boot	-
org.springframework:spring-boot-autoconfigure	-
jakarta.annotation-api	-
org.yaml:snakeyaml	-
com.fasterxml.jackson.core:jackson-databind	-
org.projectlombok:lombok	-
org.apache.commons:commons-lang3	-
com.google.guava:guava	-
org.slf4j:slf4j-api	-
org.apache.logging.log4j:log4j-api	-
org.apache.logging.log4j:log4j-core	-
org.apache.logging.log4j:log4j-slf4j-impl	-

包名groupId:artifactId	备注
org.apache.httpcomponents:httpclient	jdk8版本存在
org.springdoc:springdoc-openapi-ui	jdk8版本存在
org.springdoc:springdoc-openapi-starter-webmvc-ui	jdk17版本存在
org.apache.httpcomponents.client5:httpclient5	jdk17版本存在
io.swagger.core.v3:swagger-core	-
io.swagger:swagger-core	-
org.springframework:spring-test	-
org.springframework.boot:spring-boot-test	-

- spring-boot-huawei-dependencies已管理的包。

  spring-boot-starter-huawei未对以下版本做实际依赖，仅在spring-boot-huawei-dependencies中做版本管理，实际使用时需在pom中显式依赖，例如：

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

  需要注意以下事项：

  – 前提是已经在本模块（或父模块）的dependencyManagement中加入了spring-boot-huawei-dependencies或本模块（或父模块）的parent设置为spring-boot-huawei-parent。
  – 由于spring-boot-huawei-dependencies中继承了spring-boot-dependencies，故未在spring-boot-huawei-dependencies中声明版本的依赖可能来自spring-boot-dependencies， 此时若有依赖的版本不符合版本检查条件，则需要在本地pom中指定所依赖的版本。

  表 2-16 spring-boot-huawei-dependencies 已管理的包

包名groupId:artifactId	备注
org.springframework:spring-framework-bom	-
org.springframework.security:spring-security-bom	-
org.springframework.boot:spring-boot-dependencies	-
io.netty:netty-bom	-
io.swagger:swagger-core	-
io.swagger:swagger-models	-
io.swagger:swagger-annotations	-
io.swagger.core.v3:swagger-core	-

包名groupId:artifactId	备注
org.springdoc:springdoc-openapi-ui	jdk8版本存在
org.springdoc:springdoc-openapi-starter-webmvc-ui	jdk17版本存在
org.springdoc:springdoc-openapi-starter-webflux-ui	jdk17版本存在
org.yaml:snakeyaml	-
org.projectlombok:lombok	-
org.apache.commons:commons-lang3	-
org.apache.commons:commons-pool2	-
org.apache.commons:commons-text	-
com.google.guava:guava	-
org.apache.httpcomponents:httpclient	jdk8版本存在
org.apache.httpcomponents.client5:httpclient5	jdk17版本存在
org.apache.httpcomponents:httpmime	jdk8版本存在
org.apache.logging.log4j:log4j-api	-
org.apache.logging.log4j:log4j-core	-
org.apache.logging.log4j:log4j-slf4j-impl	-
org.apache.logging.log4j:log4j-to-slf4j	-
io.micrometer:micrometer-registry-prometheus	-
org.json:json	-
com.google.code.gson:gson	-
com.fasterxml.jackson.core:jackson-core	-
com.fasterxml.jackson.core:jackson-annotations	-
com.fasterxml.jackson.core:jackson-databind	-
com.fasterxml.jackson.dataformat:jackson-dataformat-yaml	-
com.fasterxml.jackson.datatype:jackson-datatype-jsr310	-
com.fasterxml.jackson.datatype:jackson-datatype-jdk8	-
com.fasterxml.jackson.module:jackson-module-afterburner	-
com.fasterxml.jackson.module:jackson-module-parameter-names	-
com.fasterxml.jackson.dataformat:jackson-dataformat-csv	-
com.fasterxml.jackson.dataformat:jackson-dataformat-smile	-

包名groupId:artifactId	备注
com.fasterxml.jackson.dataformat:jackson-dataformat-xml	-
com.fasterxml.jackson.dataformat:jackson-dataformat-cbor	-
com.fasterxml.jackson.module:jackson-module-jaxb-annotations	-
net.bytebuddy:byte-buddy	-
net.bytebuddy:byte-buddy-agent	-
com.alibaba:fastjson	-
com.squareup.okhttp3:okhttp	-
com.squareup.okio:okio	-
org.apache.zookeeper:zookeeper	-
org.mariadb.jdbc:mariadb-java-client	-
com.alibaba:druid	-
com.alibaba:druid-spring-boot-starter	-
jaxen:jaxen	-
org.mybatis:mybatis-spring	-
org.mybatis.spring.boot:mybatis-spring-boot-starter	-
org.bouncycastle:bcpkix-jdk18on	-
com.github.ben-manes.caffeine:caffeine	-
com.huaweicloud:esdk-obs-java	-
io.projectreactor:reactor-core	-
org.antlr:antlr4-runtime	-
jakarta.annotation:jakarta.annotation-api	-
io.prometheus:simpleclient_common	-
io.prometheus:simpleclient	-
io.prometheus:simpleclient_tracer_otel	-
io.prometheus:simpleclient_tracer_otel_agent	-

组件纳管的依赖版本

- spring-boot-starter-huawei和spring-boot-huawei-dependencies(jdk8优选本版本)：2.1.10.JDK8-RELEASE。
- spring-boot-starter-huawei和spring-boot-huawei-dependencies(jdk17优选版本)：2.1.14.JDK17-RELEASE。

优选版本依赖详情如下：

- spring-boot-starter-huawei

表 2-17 spring-boot-starter-huawei 版本依赖

包名groupId:artifactId	jdk8版本	jdk17版本	备注
org.springframework:spring-core	5.3.31	6.0.19	-
org.springframework:spring-beans	5.3.31	6.0.19	-
org.springframework:spring-expression	5.3.31	6.0.19	-
org.springframework:spring-aop	5.3.31	6.0.19	-
org.springframework:spring-context	5.3.31	6.0.19	-
org.springframework:spring-web	5.3.31	6.0.19	-
org.springframework:spring-webmvc	5.3.31	6.0.19	-
org.springframework:spring-aspects	5.3.31	6.0.19	-
org.springframework:spring-boot	5.3.31	6.0.19	-
org.springframework:spring-boot-autoconfigure	5.3.31	6.0.19	-
jakarta.annotation:jakarta.annotation-api	2.1.1	2.1.1	-
org.yaml:snakeyaml	2.2	2.0	-
com.fasterxml.jackson.core:jackson-databind	2.15.2	2.15.2	-
org.projectlombok:lombok	1.18.26	1.18.28	-
org.apache.commons:commons-lang3	3.13.0	3.13.0	-
com.google.guava:guava	32.1.2-jre	32.1.2-jre	-
org.slf4j:slf4j-api	2.18.0	2.20.0	-
org.apache.logging.log4j:log4j-api	2.18.0	2.20.0	-

包名groupId:artifactId	jdk8版本	jdk17版本	备注
org.apache.logging.log4j:log4j-core	2.18.0	2.20.0	-
org.apache.logging.log4j:log4j-slf4j-impl	2.18.0	2.20.0	-
org.apache.httpcomponents:httpclient	4.5.14	-	jdk8版本存在
org.springdoc:springdoc-openapi-ui	1.6.9	-	jdk8版本存在
org.springdoc:springdoc-openapi-starter-webmvc-ui	-	2.2.0	jdk17版本存在
org.apache.httpcomponents.client5:httpclient5	-	5.2.1	jdk17版本存在
io.swagger.core.v3:swagger-core	2.2.8	2.2.8	-
io.swagger:swagger-core	1.6.6	1.6.6	-
org.springframework:spring-test	5.3.31	6.0.19	-
org.springframework.boot:spring-boot-test	2.7.18	3.1.6	-

- spring-boot-huawei-dependencies

表 2-18 spring-boot-huawei-dependencies 版本依赖

包名groupId:artifactId	jdk8版本	jdk17版本	备注
org.springframework:spring-framework-bom	5.3.31	6.0.19	-
org.springframework.security:spring-security-bom	5.8.8	6.2.3	-
org.springframework.boot:spring-boot-dependencies	2.7.18	3.1.6	-
io.netty:netty-bom	4.1.109.Final	4.1.109.Final	-
io.swagger:swagger-core	1.6.6	1.6.6	-
io.swagger:swagger-models	1.6.6	1.6.6	-
io.swagger:swagger-annotations	1.6.6	1.6.6	-

包名groupId:artifactId	jdk8版本	jdk17版本	备注
io.swagger.core.v3:swagger-core	2.2.8	2.2.8	-
org.springdoc:springdoc-openapi-ui	1.6.9	-	jdk8版本存在
org.springdoc:springdoc-openapi-starter-webmvc-ui	-	2.2.0	jdk17版本存在
org.springdoc:springdoc-openapi-starter-webflux-ui	-	2.2.0	jdk17版本存在
org.yaml:snakeyaml	2.2	2.0	-
org.projectlombok:lombok	1.18.26	1.18.28	-
org.apache.commons:commons-lang3	3.13.0	3.13.0	-
org.apache.commons:commons-pool2	2.11.1	2.11.1	-
org.apache.commons:commons-text	1.10.0	1.10.0	-
com.google.guava:guava	32.1.2-jre	32.1.2-jre	-
org.apache.httpcomponents:httpclient	4.5.14	-	jdk8版本存在
org.apache.httpcomponents.client5:httpclient5	-	5.2.1	jdk17版本存在
org.apache.httpcomponents:httpmime	4.5.14	-	jdk8版本存在
org.apache.logging.log4j:log4j-api	2.18.0	2.20.0	-
org.apache.logging.log4j:log4j-core	2.18.0	2.20.0	-
org.apache.logging.log4j:log4j-slf4j-impl	2.18.0	2.20.0	-
org.apache.logging.log4j:log4j-to-slf4j	2.18.0	2.20.0	-
io.micrometer:micrometer-registry-prometheus	1.10.1	1.10.1	-
org.json:json	20231013	20231013	-
com.google.code.gson:gson	2.9.1	2.10.1	-
com.fasterxml.jackson.core:jackson-core	2.15.2	2.15.2	-

包名groupId:artifactId	jdk8版本	jdk17版本	备注
com.fasterxml.jackson.core:jackson-annotations	2.15.2	2.15.2	-
com.fasterxml.jackson.core:jackson-databind	2.15.2	2.15.2	-
com.fasterxml.jackson.dataformat:jackson-dataformat-yaml	2.15.2	2.15.2	-
com.fasterxml.jackson.datatype:jackson-datatype-jsr310	2.15.2	2.15.2	-
com.fasterxml.jackson.datatype:jackson-datatype-jdk8	2.15.2	2.15.2	-
com.fasterxml.jackson.module:jackson-module-afterburner	2.15.2	2.15.2	-
com.fasterxml.jackson.module:jackson-module-parameter-names-	2.15.2	2.15.2	-
com.fasterxml.jackson.dataformat:jackson-dataformat-csv	2.15.2	2.15.2	-
com.fasterxml.jackson.dataformat:jackson-dataformat-smile	2.15.2	2.15.2	-
com.fasterxml.jackson.dataformat:jackson-dataformat-xml	2.15.2	2.15.2	-
com.fasterxml.jackson.dataformat:jackson-dataformat-cbor	2.15.2	2.15.2	-
com.fasterxml.jackson.module:jackson-module-jaxb-annotations	2.15.2	2.15.2	-
net.bytebuddy:byte-buddy	1.12.10	1.14.6	-
net.bytebuddy:byte-buddy-agent	1.12.10	1.14.6	-
com.alibaba:fastjson	1.2.83	2.0.39	-
com.squareup.okhttp3:okhttp	4.11.0	4.11.0	-
com.squareup.okio:okio	3.4.0	3.4.0	-
org.apache.zookeeper:zookeeper	3.7.1	3.9.2	-
org.mariadb.jdbc:mariadb-java-client	3.0.5	3.0.5	-
com.alibaba:druid	1.2.10	1.2.18	-

包名groupId:artifactId	jdk8版本	jdk17版本	备注
com.alibaba:druid-spring-boot-starter	1.2.9	1.2.15	-
jaxen:jaxen	2.0.0	2.0.0	-
org.mybatis:mybatis-spring	2.0.7	2.1.0	-
org.mybatis.spring.boot:mybatis-spring-boot-starter	2.2.2	3.0.1	-
org.bouncycastle:bcpkix-jdk18on	1.75	1.75	-
com.github.ben-manes.caffeine:caffeine	2.9.3	3.1.8	-
com.huaweicloud:esdk-obs-java	3.23.3	3.23.3	-
io.projectreactor:reactor-core	3.4.18	3.5.11	-
org.antlr:antlr4-runtime	4.9.2	4.13.1	-
jakarta.annotation:jakarta.annotation-api	2.1.1	2.1.1	-
io.prometheus:simpleclient_common	0.16.0	0.16.0	-
io.prometheus:simpleclient	0.16.0	0.16.0	-
io.prometheus:simpleclient_tracer_otel	0.16.0	0.16.0	-
io.prometheus:simpleclient_tracer_otel_agent	0.16.0	0.16.0	-

2.5.3 配置说明

2.5.3.1 日志配置

spring-boot-starter-huawei中默认使用slf4j作为日志门面，log4j2作为日志实现框架。

当项目中的依赖可能存在有关logback日志框架的时候需要将其排除，否则可能会有spring-boot-starter-huawei中配置的slf4j（使用log4j-slf4j-impl桥接slf4j的日志）的日志框架相冲突。

org.springframework.boot:spring-boot-starter-logging使用的也是logback的日志框架，所以当项目的依赖中存在对spring-boot-starter-logging的依赖时需要将其排除，例如：

```
<dependency>
   <groupId>org.mybatis.spring.boot</groupId>
```

```
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.2.0</version>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

对于开发时简单的日志配置，直接在application.yml中使用yml方式进行配置，该配置简单易于理解；但也只能进行简单的配置，滚动策略无法配置。

- **在application.yml中配置简单的日志。**

```
#日志配置
logging:
  charset:
    file: UTF-8
    console: UTF-8
  file:
    name: /opt/cloud/log/{服务名}/{服务名}.log #日志文件的位置及名称
  pattern:
    file: "[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%-5level] [%thread] [%X{apm-traceid}] [%X{apm-
gtraceid}] [%logger{15}] - [%msg]%n" #日志记录格式
    console: "[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%-5level] [%thread] [%X{apm-traceid}] [%X{apm-
gtraceid}] [%logger{15}] - [%msg]%n" #日志记录格式
  level:
    root: INFO
    org.mybatis: DEBUG
    org.springfrmework: INFO
    com.example.demo.controller: INFO
```

以上日志配置适合开发时本地配置，较为简单。当以上配置无法满足要求的时候，可以使用单独的日志配置文件来进行日志配置。

- **使用yml文件进行日志配置**。

```
#日志配置
#配置log4j2日志文件时使用yml方式配置,在项目资源路径下添加log4j2.yml文件(或在application.yml中使
用logging.config=log4j2-xxx.yml指定位置)配置内容如下示例:

Configuration:
  status: warn
  Properties: # 变量定义
    Property: #
      - name: microservice        ### DONE: 增加配置
        value: default
      - name: log.path #日志路径,推荐配置:此路径
        value: /opt/cloud/log/${microservice}
      - name: log.pattern #日志格式,推荐此格式
        value: "[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%-5level] [%thread] [%X{apm-traceid}] [%X{apm-
gtraceid}] [%logger{15}] - [%msg]%n"    ###DONE LC:paas规范 刷新减掉 timestamp , 与apm集成
  Appenders:
    Console:  # 输出到控制台
      name: CONSOLE
      target: SYSTEM_OUT
      ThresholdFilter: #过滤条件
        level: info        ###DONE: LC:Console只输出info级别就即可， 详细debug日志记录到文件
        onMatch: ACCEPT
        onMismatch: DENY
      PatternLayout:
        pattern: "${log.pattern}" # 控制台输出格式
    RollingFile: # 输出到文件
      - name: ROLLING_FILE
        ignoreExceptions: false
        fileName: "${log.path}/${microservice}.log" #日志文件名称        ###DONE LC:  ${{}} demo?  应用
名?
        filePattern: "${log.path}/${microservice}-%d{yyyy-MM-dd}.%i.zip" #压缩后日志文件名称
###DONE 看规范是否要更新建议
        append: true
        PatternLayout:
```

```
        pattern: "${log.pattern}" # 文件输出格式
      Policies:
#       TimeBasedTriggeringPolicy:  #按天分类
#         modulate: true
#         interval: 1
        SizeBasedTriggeringPolicy: #按大小分类
          size: 100MB #大小,超过此大小后压缩打包日志文件,单位:KB,MB,GB          ###DONE -> 刷新规范是50MB
      DefaultRolloverStrategy: #最大保存文件个数
        max: 100        ###DONE -> LC: 刷新规范是40 -> 100
        Delete: # 删除策略
          basePath: ${log.path} # 删除基本路径
          maxDepth: 2 # 目录检索深度,当前${log.path}指定的目录深度为0
          IfFileName: # 匹配文件名称删除
          glob: "*${microservice}-*zip" # 文件名称              ###DONE LC:  demo被hardcode了
            IfLastModified: # 删除文件条件定义
              age: 7d # 7天或以上
              ifAny: # 附件的一些条件,可不定义
                IfAccumulatedFileSize: # 文件大小,当前满足age设定的条件时,并且达到该条件则删除(保留10G过期文件)
                  exceeds: 10GB
                IfAccumulatedFileCount: # 文件个数,当前满足age设定的条件时,并且达到该条件则删除(保留100个过期文件)
                  exceeds: 100
  Loggers:
    Root:
      level: info
      AppenderRef:
        - ref: CONSOLE
        - ref: ROLLING_FILE
    Logger:
      - name: com.huaweicloud.sample.core.controller
        level: debug
        additivity: false # 是否向父logger传递,否:false
        AppenderRef:
          - ref: CONSOLE
          - ref: ROLLING_FILE
```

- **使用xml文件进行日志配置。**

```
<?xml version="1.0" encoding="UTF-8"?>
<!--配置log4j2日志文件时使用xml方式配置,在项目资源路径下添加log4j2.xml文件(或在application.yml中使用logging.config=log4j2-xxx.xml指定位置)配置内容如下示例:-->
<Configuration status="debug" strict="true" name="XMLConfigTest"
packages="org.apache.logging.log4j.test">
  <Properties>
    <Property name="microservice">default</Property>
    <Property name="log.path">/opt/cloud/log/${microservice}</Property>
    <Property name="log.pattern">[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%-5level] [%thread] [%X{apm-traceid}] [%X{apm-gtraceid}] [%logger{15}] - [%msg]%n</Property>
  </Properties>

  <Appenders>
    <Appender type="Console" name="CONSOLE">
      <Layout type="PatternLayout" pattern="${log.pattern}" />
      <Filters>
        <Filter type="ThresholdFilter" level="info" onMatch="ACCEPT" onMismatch="DENY" />
      </Filters>
    </Appender>
    <RollingFile name="ROLLING_FILE">
      <FileName>${log.path}/${microservice}.log</FileName>
      <FilePattern>${log.path}/${microservice}-%d{yyyy-MM-dd}.%i.zip</FilePattern>
      <Layout type="PatternLayout">
        <Pattern>${log.pattern}</Pattern>
      </Layout>
      <Policies>
        <SizeBasedTriggeringPolicy size="100MB" />
      </Policies>
      <DefaultRolloverStrategy max="100">
        <Delete basePath="${log.path}" maxDepth="2">
      <IfFileName glob="*${microservice}-*zip">
```

```
      <IfLastModified age="7d">
          <IfAny>
            <IfAccumulatedFileSize exceeds="10GB" />
            <IfAccumulatedFileCount exceeds="100" />
          </IfAny>
        </IfLastModified>
      </IfFileName>
    </Delete>
  </DefaultRolloverStrategy>
 </RollingFile>
</Appenders>

<Loggers>
 <Root level="info">
  <AppenderRef ref="CONSOLE" />
  <AppenderRef ref="ROLLING_FILE" />
 </Root>
 <Logger name="org.mybatis" level="debug" additivity="false">
  <AppenderRef ref="CONSOLE" />
  <AppenderRef ref="ROLLING_FILE" />
 </Logger>
 <Logger name="com.example.demo.controller" level="info" additivity="false">
  <AppenderRef ref="CONSOLE" />
  <AppenderRef ref="ROLLING_FILE" />
 </Logger>
</Loggers>

</Configuration>
```

更多配置，请参考log4j2官方文档配置。中文文档参考： **https://www.docs4dev.com/docs/zh/log4j2/2.x/all/manual-configuration.html**。

2.5.3.2 使用 swagger 进行在线 API 文档配置

文档配置

spring-boot-starter-huawei添加了swagger2和swagger3包的依赖，API在线文档使用springdoc-openapi-ui。springdoc基本配置较为简单，如下所示：

```
springdoc:
 api-docs:
  enabled: true
  path: /api-docs
  resolve-schema-properties: true
 swagger-ui:
  path: /swagger-ui.html #在线文档访问路径,实际访问路径为:http://{host}:{port}/{context-path}/swagger-ui.html
  showCommonExtensions: true
  disable-swagger-default-url: false
```

若原有项目中使用的是springfox+swagger，则启动时可能报错：

```
org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'swaggerWelcome' defined in class path resource [org/springdoc/webmvc/ui/SwaggerConfig.class]: Post-processing of merged bean definition failed; nested exception is java.lang.IllegalStateException: Failed to introspect Class [org.springdoc.webmvc.ui.SwaggerWelcomeWebMvc] from ClassLoader [sun.misc.Launcher$AppClassLoader@18b4aac2]
```

此时，若后续仍想使用springfox+swagger，则将spring-boot-starter-huawei中的springdoc排除即可：

```
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-boot-starter-huawei</artifactId>
  <version>${spring.boot.starter.huawei}</version>
  <exclusions>
    <exclusion>
```

```
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-ui</artifactId>
      </exclusion>
    </exclusions>
</dependency>
```

若后续使用springdoc生成在线api文档，则：

1、将原来的sprignfox相关的依赖删除。

2、将配置"SwaggerConfig"（根据实际情况查找配置类名称）相关的配置删除。

3、配置文件中添加springdoc的配置并删除springfox的配置。

swagger 支持 https 和认证的方式

- **支持https**

  在启动类中加入注解：

```
@OpenAPIDefinition(
        servers = {
            @Server(
                    url = "https://coralgeneratorsvr.{Environment}-szv-kunpeng-camp.tools.huawei.com/
coralgeneratorsvr", #替换成自己的服务地址
                    variables = {
                        @ServerVariable(
                                name = "Environment",# 参数定义,可使用{param}取其中的值
                                allowableValues = {"alpha", "beta", "gamma", "prod"},#参数取值范围
                                defaultValue = "alpha"),
                    },
                    description = "CoralGeneratorSvr Environmental Urls"),
            @Server(url = "http://localhost:8080/coralgeneratorsvr", description = "Local Dev Url")
        },
        info = # 服务描述信息
                @Info(
                        title = "CoralGeneratorSvr",
                        version = "v1",
                        description = "Coral Generator Server",
                        contact = @Contact(name = "x00464738", email = "xiaoweimin@huawei.com")))
@SpringBootApplication
@ComponentScan(value = {"com.huawei.coral", "com.huawei.coral.coralgeneratorsvr",
"com.huawei.clouddragon.devuc.sdk.*", "com.huawei.clouddragon.apigateway_client"})
@MapperScan("com.huawei.coral.coralgeneratorsvr.mapper")
@Slf4j
public class Application extends SpringBootServletInitializer {
    ...
}
```

- **支持服务接口认证**

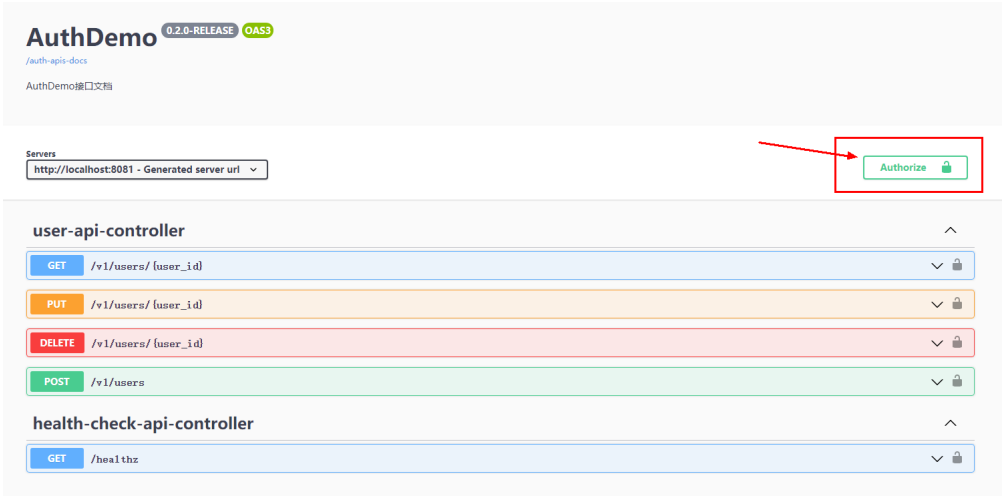
  新增一个配置类，在配置类中加入以下配置：

```
/**
 * 在线文档接口认证配置
 *
 * @since 2021-08-12
 */
@Configuration(proxyBeanMethods = false)
public class SwaggerAuthConfig {
    @Bean
    public OpenAPI springShopOpenAPI() {
        return new OpenAPI()
                .components(new Components()
                        .addSecuritySchemes("xAuthToken",
                                new SecurityScheme()
                                        .type(SecurityScheme.Type.APIKEY) //请求认证类型
                                        .name("x-auth-token") //参数名称
                                        .description("sso proxy token: x-auth-token") //API key描述
                                        .in(SecurityScheme.In.HEADER))) //设置API key的存放位置（发送请求时请
```

```
求头中会带上x-auth-token )
                .security(Collections.singletonList(new SecurityRequirement().addList("xAuthToken"))) //
SecurityRequirement中配置的名称需要与SecuritySchema的名称匹配
                .info(new Info().title("AuthDemo")
                        .description("AuthDemo接口文档")
                        .version("0.2.0-RELEASE"));
    }
}
```

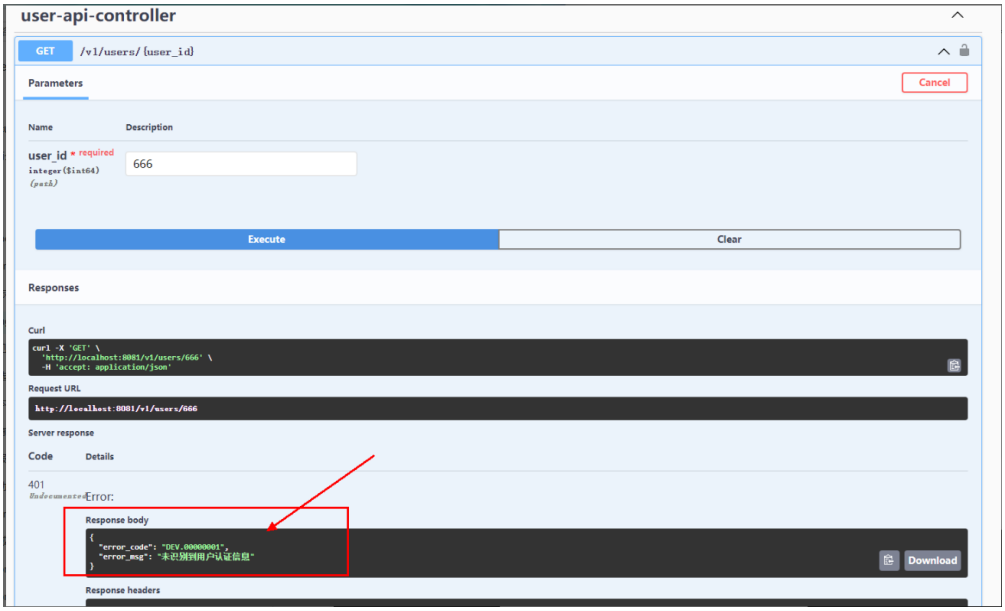
完成以上配置后，在线文档中会多出一个认证按钮：

图 2-9 认证



在未输入认证凭证时，直接访问需要认证的接口会返回需要认证的信息：

图 2-10 返回认证信息



输入认证凭证：

图 2-11 输入认证凭证

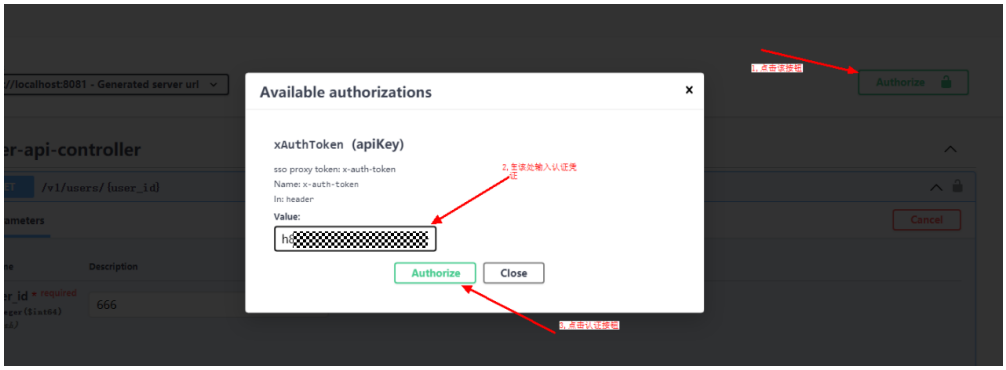
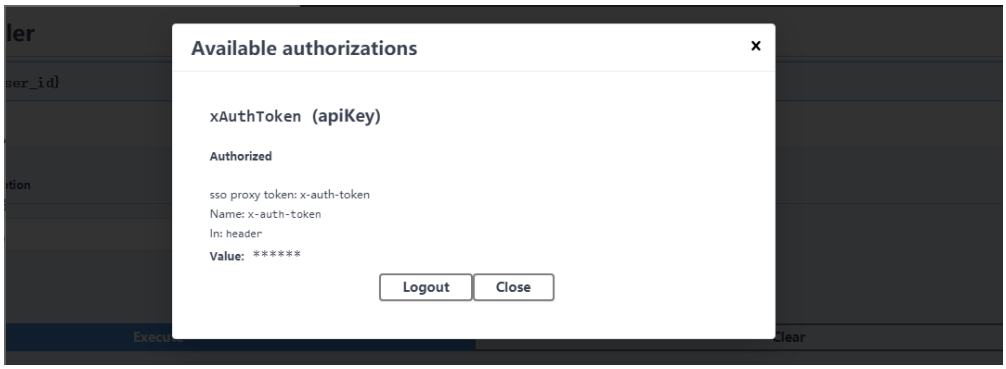
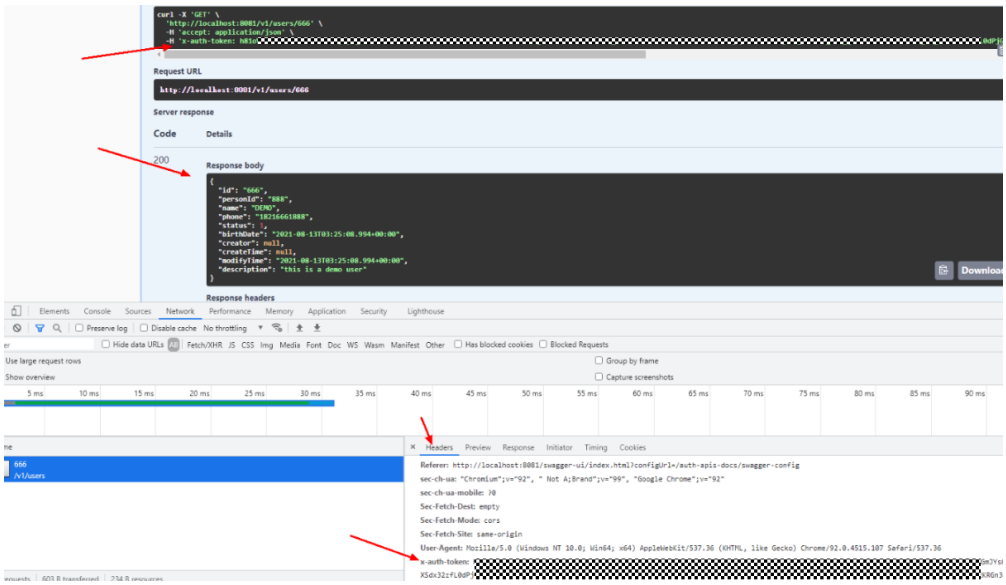


图 2-12 认证完成



输入认证凭证后，再次调用接口时会将认证信息以配置的方式同请求内容一同发送。

图 2-13 发送认证信息



2.6 AstroPro-SDK 版本变更与下载

AstroPro-SDK版本变更记录及下载地址如下。您可以按需下载所需的AstroPro-SDK，也可以直接通过表2-19中链接，一次性下载所有的AstroPro-SDK。

如果您能访问外网并下线maven依赖，则可通过修改maven settings.xml文件来配置华为开源镜像仓库进行依赖管理，具体配置如下：

步骤1 在profiles节点中添加如下内容：

```
<profile>
   <id>MyProfile</id>
   <repositories>
      <repository>
         <id>HuaweiCloudSDK</id>
         <url>https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/</url>
         <releases>
            <enabled>true</enabled>
         </releases>
         <snapshots>
            <enabled>false</enabled>
         </snapshots>
      </repository>
   </repositories>
   <pluginRepositories>
      <pluginRepository>
         <id>HuaweiCloudSDK</id>
         <url>https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/</url>
         <releases>
            <enabled>true</enabled>
         </releases>
         <snapshots>
            <enabled>false</enabled>
         </snapshots>
      </pluginRepository>
   </pluginRepositories>
</profile>
```

步骤2 在mirrors节点中增加：

```
<mirror>
   <id>huaweicloud</id>
   <mirrorOf>*,!HuaweiCloudSDK</mirrorOf>
   <url>https://repo.huaweicloud.com/repository/maven/</url>
</mirror>
```

步骤3 增加activeProfiles标签激活配置：

```
<activeProfiles>
   <activeProfile>MyProfile</activeProfile>
</activeProfiles>
```

----结束

表 2-19 SDK 全量下载

开发语言	包含的模块	版本及下载地址	SHA256值	变更描述
Java	所有	obs-20240525	e6aec815e76ee0b12624a5238234309e7173210a528949f2b65503802c51981d	2024年05月25日发布版本。

开发语言	包含的模块	版本及下载地址	SHA256值	变更描述
Java	所有	obs-20240330	b5724214b6b56645d2517fa21bfec8eb8328b762999ee4bc1ee358b060d109e1	2024年03月30日发布版本。

表 2-20 devspore-auth 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	devspore-auth-oneaccess	3.10.9.JDK17-RELEASE 3.10.8.JDK8-RELEASE	修复一些问题。
Java		3.10.7.JDK17-RELEASE 3.10.7.JDK8-RELEASE	修复一些问题。
Java		3.10.6.JDK17-RELEASE 3.10.6.JDK8-RELEASE	修复一些问题。
Java		3.10.5.JDK17 3.10.5.JDK8-RELEASE	• 新增JDK17。 • 修复一些问题。
Java		3.10.3.OBT.JDK8-RELEASE	初次发布。

表 2-21 devspore-security 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	• devspore-security • devspore-security-commons • devspore-security-parent	2.1.9.JDK17-RELEASE 2.1.5.JDK8-RELEASE	修复一些问题。
Java		2.1.4.JDK17-RELEASE 2.1.4.JDK8-RELEASE	修复一些问题。

开发语言	包含的模块	版本及下载地址	变更描述
Java		2.1.3.JDK17-RELEASE 2.1.3.JDK8-RELEASE	修复一些问题。
Java		2.1.2.JDK17-RELEASE 2.1.2.JDK8-RELEASE	• 新增JDK17。 • 修复一些问题。
Java		2.0.3.OBT.JDK8-RELEASE	初次发布。

表 2-22 devspore-probe 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	devspore-probe	2.1.8.JDK17-RELEASE 2.1.7.JDK8-RELEASE	修复一些问题。
Java		2.1.4.JDK17-RELEASE 2.1.4.JDK8-RELEASE	修复一些问题。
Java		2.1.3.JDK17-RELEASE 2.1.3.JDK8-RELEASE	修复一些问题。
Java		2.1.2.JDK17-RELEASE 2.1.2.JDK8-RELEASE	• 新增JDK17。 • 修复一些问题。
Java		2.0.2.JDK8-RELEASE	修复一些问题。
Java		2.0.1.JDK8-RELEASE	初次发布。

表 2-23 devspore-clientcontrol 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	devspore-clientcontrol	2.1.4.JDK17-RELEASE 2.1.4.JDK8-RELEASE	修复一些问题。
Java		2.1.3.JDK17-RELEASE 2.1.3.JDK8-RELEASE	• 新增JDK17。 • 修复一些问题。
Java		2.0.3.JDK8-RELEASE	初次发布。

表 2-24 devspore-http-log 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	devspore-http-log	2.1.4.JDK17-RELEASE	修复一些问题。此次只更新了JDK17。
Java		2.1.3.JDK17-RELEASE 2.1.3.JDK8-RELEASE	修复一些问题。
Java		2.1.2.JDK17-RELEASE 2.1.2.JDK8-RELEASE	• 新增JDK17。 • 修复一些问题。
Java		2.0.3.JDK8-RELEASE	修复一些问题。
Java		2.0.1.JDK8-RELEASE	初次发布。

表 2-25 devspore-mas-common 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	devspore-mas-common	2.1.3.JDK17-RELEASE 2.1.3.JDK8-RELEASE	修复一些问题。
Java		2.1.2.JDK17-RELEASE 2.1.2.JDK8-RELEASE	• 新增JDK17。 • 修复一些问题。
Java		2.0.1.JDK8-RELEASE	初次发布。

表 2-26 devspore-dds 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	• devspore-dds • devspore-dds-parent • spring-cloud-starter-huawei-devspore-dds	2.1.2.JDK17-RELEASE 2.1.2.JDK8-RELEASE	• 新增JDK17。 • 修复一些问题。
Java		2.0.2.JDK8-RELEASE	初次发布。

表 2-27 devspore-css 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	devspore-css	2.0.1.JDK8-RELEASE	初次发布。

表 2-28 devspore-datasource 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	• devspore-datasource • devspore-datasource-parent • spring-cloud-starter-huawei-devspore-datasource	2.1.2.JDK17-RELEASE 2.1.2.JDK8-RELEASE	• 新增JDK17。 • 修复一些问题。
Java		2.0.2.JDK8-RELEASE	初次发布。

表 2-29 devspore-dcs 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	• devspore-dcs • devspore-dcs-parent • spring-cloud-starter-huawei-devspore-dcs	2.1.3.JDK17-RELEASE 2.1.3.JDK8-RELEASE	修复一些问题。
Java		2.1.2.JDK17-RELEASE 2.1.2.JDK8-RELEASE	• 新增JDK17。 • 修复一些问题。
Java		2.0.2.JDK8-RELEASE	初次发布。

表 2-30 spring-boot-huawei 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	• spring-boot-huawei • spring-boot-huawei-dependencies • spring-boot-huawei-parent • spring-boot-starter-huawei	2.1.14.JDK17-RELEASE 2.1.10.JDK8-RELEASE	修复一些问题。
Java		2.1.8.JDK17-RELEASE 2.1.8.JDK8-RELEASE	修复一些问题。
Java		2.1.6.JDK17-RELEASE 2.1.6.JDK8-RELEASE	修复一些问题。
Java		2.1.4.JDK17-RELEASE 2.1.4.JDK8-RELEASE	• 新增JDK17。 • 修复一些问题。
Java		2.0.3.JDK8-RELEASE	初次发布。

表 2-31 devspore-generator 版本变更

开发语言	包含的模块	版本及下载地址	变更描述
Java	• devspore-core • devspore-generator • devspore-horizon • devspore-metadata • devspore-parent • devspore-plugin • devspore-util	2.1.15-RELEASE	修复一些问题。
Java		2.1.11-RELEASE	修复一些问题。
Java		2.1.8-RELEASE	修复一些问题。
Java		2.0.2.JDK8-RELEASE	初次发布。

SDK 压缩包完整性校验

- linux下验证

步骤1 在表格的“版本及下载地址”中获取SDK包下载路径。

步骤2 下载SDK包到本地。

步骤3 输入如下命令。

```
sha256sum {压缩包名}
```

步骤4 对比压缩包.sha256的SHA256值和下载后的SDK包的SHA256值。

- 一致，则表示压缩包完整，下载过程不存在篡改和丢包。

- 不一致，说明SDK压缩包被篡改，需要重新获取。

----结束

- windows验证

步骤1 在表格的“版本及下载地址”中获取SDK包下载路径。

步骤2 下载SDK包到本地。

步骤3 打开本地命令提示符框，输入如下命令。

```
certutil -hashfile {压缩包名} SHA256
```

步骤4 对比压缩包.sha256的SHA256值和下载后的SDK包的SHA256值。

- 一致，则表示压缩包完整，下载过程不存在篡改和丢包。
- 不一致，说明SDK压缩包被篡改，需要重新获取。

----结束