后端Serverless接入指北

包依赖

```
<!--SDK坐标-->
<dependency>
    <groupId>com.ac.apaas</groupId>
    <artifactId>serverless-sdk</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>
<!--注意事项:
    当前发布仓库地址：172.22.5.11:8082
    如有其它仓库环境需发布，请联系lex
-->
```

注解启用开关

```
import com.ac.apaas.serverless.sdk.annotation.EnableServerless;

@SpringBootApplication(scanBasePackages = {"xxx.xxx"})
@EnableServerless
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

nacos配置

```
#在应用的bootstrap.yml文件中添加一个扩展配置
spring:
  cloud:
    nacos:
      config:
        extension-configs:
```

```
        - data-id: shared-serverless-config.${spring.cloud.nacos.config.file-extension:properties}
          group: ${spring.cloud.nacos.config.group}
          refresh: true
```

以下为shared-serverless-config.properties配置项

```
# 是否启用serverless的开关
apaas.serverless.enable=true
# 租户名称isagent islot pachinko
apaas.serverless.namespace=islot
# serverless后端服务地址
## test:https://apaas-serverless-api-v2.dmrd-test.com
## uat: https://apaas-serverless-api.isuat.net
##prod: https://apaas-serverless-api.islotprd.com
openapi.client.app.serverless.host=https://apaas-serverless-api-v2.dmrd-test.com
# serverless api版本
openapi.client.app.serverless.version=api/v1
```

1. Bean用法

1.1 管理后台创建新的类

- 针对在托管服务上执行openfeign调用，需要在feignClient属性注入上添加如下注解用于将feign调用转为泛化调用。该注解的作用只作为代码翻译，在编译时会丢弃

```
@org.springframework.beans.factory.annotation.Resource
@com.ac.apaas.serverless.sdk.annotation.GenericFeignClientMappers(name = "xxxService",
        value = {

@com.ac.apaas.serverless.sdk.annotation.GenericFeignClientMethodMapper(sourceMethodName = "test", httpMethod = "post", targetMethodUrl = "/xxx/xxx"),
        })
private ServiceFeignClient serviceFeignClient;
```

2. Function用法

2.1 方法上添加注解@FunctionInjection

在java运行时，方法的实现将由脚本代码动态替换执行

2.2 注解参数详解

- name：函数别名，必填参数，同一租户下须唯一。与控制台函数管理中创建的函数名一对一，建议采用packagename.classname.methodname使其唯一
- isAsync：是否异步执行，默认否
- traceId：自定义traceId，不填则以namespace:appname:timestamp组成traceId
- clientIp：当前应用的ip，可空

2.3 function脚本规范约定

1. 必须以类的形式包装代码逻辑
2. 必须包括5个保留函数：
   a. accept：该函数用于自定义此函数是否执行的前提条件
   b. onBefore：action方法执行前的处理
   c. action：真正执行的函数逻辑。为了提升稳定性，须指定方法执行的超时时间，通过@TimedInterrupt注解即可指定，值为秒。action的返回值类型必须与原方法的返回值类型保持一致
   d. onException：action方法执行异常时的处理
   e. onAfter：action方法执行后的处理
3. 参数列表 Map<String, Object> map：

框架会自动将原方法的参数列表合并到map结构，比如原方法为void print(int a, BigDecimal b, ApiResponse c)，则map的key会有a,b,c。同时会将租户下该应用的环境变量也装载进来。环境变量在控制台设置和管理。

```
import com.slid.live.slot.component.api.ApiResponse
import groovy.transform.TimedInterrupt

class GroovyFunctionTest {
    /**
     * 匹配条件，如果为false，则后续的方法不会执行
     * @param map 请求参数，原方法的参数列表合并到此map结构
     * @return
     */
    Boolean accept(Map<String, Object> map) {
        return true;
    }

    /**
```

```
     * action方法执行前的处理
     * @param map 请求参数，原方法的参数列表合并到此map结构
     */
    void onBefore(Map<String, Object> map) {
    }

    /**
     * 真正执行的函数逻辑
     * @param map 请求参数，原方法的参数列表合并到此map结构。比如原方法为:
        ApiResponse<String> testFunction(@RequestBody Request request);
        @Data
        public static class Request {
            private String key;
            private Integer num;
        }
     * @return 必须与原方法的返回值类型保持一致
     */
    @TimedInterrupt(value = 60L)
    ApiResponse<String> action(Map<String, Object> map) {
        println("action")
        int num = (int)
Optional.ofNullable(map.get('request').getAt('num')).orElse(3)
        while (num-- > 0) {

println(Optional.ofNullable(map.get('request').getAt('key')).orElse("no key"))
        }
        String data =
Optional.ofNullable(map.get('request').getAt('key')).orElse("") + ":updated"
        return ApiResponse.create("200", null, data)
    }

    /**
     * action方法执行异常时的处理
     * @param map 请求参数，原方法的参数列表合并为此map结构
     */
    void onException(Map<String, Object> map) {
    }

    /**
     * action方法执行后的处理
     * @param map 请求参数，原方法的参数列表合并为此map结构
     */
    void onAfter(Map<String, Object> map) {
    }
}
```

2.4 版本化管理

每次在控制台保存一次函数，都会以最新的版本号追加新记录不会更新老版本。默认以最新的版本号（也就是版本号最大的）执行

2.5 版本回滚

在控制台上找到须回滚的某一版本，点击启用按钮，即可让sdk获取该版本的代码执行

3. Hotfix用法

3.1 控制台创建hotfix任务

- namespace：租户名称，必填
- appname：租户下的应用名，须与${spring.application.name}配置项一致，必填
- classpath：要修复的类路径，必填
- methodName：要修复的类方法名，必填
- returnType：方法返回类型的类路径，必填
- code：groovy代码，必填
- argNames：修复代码的参数名称数组
- argTypes：修复代码的参数类型classpath数组
- argValues：修复代码的参数值数组
- description：描述

3.2 hotfix脚本规范约定

1. 必须以类的形式包装代码逻辑
2. 必须以**hotfix方法名**命名热更新方法：
3. hotfix方法返回值类型必须与原方法的返回值类型保持一致
4. hotfix的入参可以与原方法不一致，如果在控制台指定了修复代码的入参，则须以Map<String, Object>结构引用自定义参数，此map结构也会包含应用维度的环境变量。
5. 代码示例：
   a. 原代码

```
1 @Service
2 public class TestService {
3
4     public ResponseEntity<String> printForGroovyTest(String msg) {
```

```
        System.out.println("hello world");
        return ResponseEntity.ok(msg);
    }
}
```

b. 要热修复的groovy脚本

```
import groovy.transform.TimedInterrupt

class GroovyHotfixTest {
    @TimedInterrupt(value = 10L)
    @groovy.transform.ThreadInterrupt
    org.springframework.http.ResponseEntity<String> hotfix() {
        println('热更新');
        return org.springframework.http.ResponseEntity.ok('热更新');
    }
}
```

3.3 发布

点击发布按钮发布hotfix