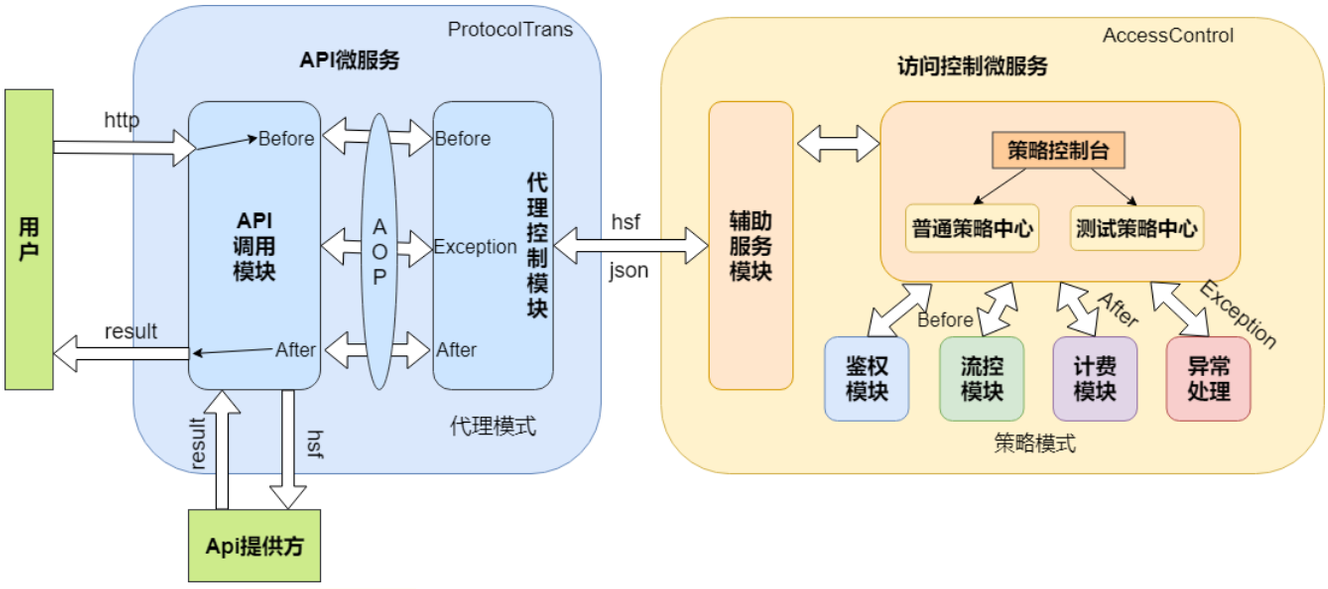
API 访问和控制模块

一、总体架构



1. **微服务设计（祝大哥来补充）**

   Api 调用模块和访问控制模块含有拆分成两个微服务的需要。

   从架构拆分原理上看：

   （1） 两个高内聚、低耦合的程序。

   （2） 独立进程、独立部署

   从拓展性上看：

   （1） 较强的 X 轴拓展的需要。压力情况下，某单个服务的负载过大，可通过给该服务增加节点的方式来解决。

   （2） 拓展业务接入支持。如果有因新业务产生的 Api 访问相关程序，可以共同调用同一个访问控制服务，避免重新在新业务中加入冗余的控制代码。（Y 轴拓展的可能）

2. **微服务平台**

   采用 Alibaba 的企业级分布式应用 EDAS。EDAS 是一个围绕应用和微服务的 PaaS 平台，提供多样的应用发布和轻量级微服务解决方案，帮助用户解决在应用和服务管理过程中监控、诊断和高可用运维问题。

3. **微服务通信**

   微服务通信采用 HSF 通信。是一种 Alibaba 支持的 RPC 调用方式。这两个微服务之间的调用关系是，访问控制服务为服务提供者，Api 服务为服务消费者。（消费者与提供者的关系并不是永恒不变的，不同的调用过程对应着有不同的角色分配。对于用户来说，Api 微服务是它的服务提供者。对于 Api 提供方来说，这个 Api 微服务又是服务消费者。）

二、模块设计

1. 代理模式　　包括远程代理、虚拟代理、保护代理、智能代理

意图：为其他对象提供一种代理以控制对这个对象的访问。

主要解决：在直接访问对象时带来的问题，比如说：**要访问的对象在远程的机器上**。在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者**某些操作需要安全控制**，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。

如何解决：增加中间层。

优点： 1、职责清晰。 2、高扩展性。

缺点：由于在客户端和真实主题之间增加了代理对象，因此有可能会比较慢。

实际上我们的微服务架构隐含有两层代理。一层是在用户调用 Api 时，被 ProtocolTranslation 服务中的代理控制服务拦截进行处理。一层是代理控制服务远程调用实际的控制服务，也就是我们的 AccessControl 微服务来进行处理。

具体应用：Spring AOP（留给祝大哥）

2. 策略模式

意图：定义一系列的算法，把它们一个个封装起来，并且使它们可**相互替换**。

主要解决：在有**多种算法相似**的情况下，使用 if...else 所带来的复杂和难以维护。

如何解决：将这些算法封装成一个一个的类，任意地替换。

关键代码：实现同一个接口。

（1）AccessControl 服务中，所有的控制策略都实现同一个接口。

```
public interface AccessStrategyI {
    public void before(String args);

    /**
     * 调用前访问控制
     * @param object 控制参数
     * @throws ControlException 拒绝访问异常 {code：错误码，msg：错误信息}
     */
    public void around(JSONObject object) throws ControlException;

    /**
     * 调用后控制
     * @param object 控制参数
     * @throws ControlException 拒绝访问异常 {code：错误码，msg：错误信息}
     */
    public void after(JSONObject object) throws ControlException;

    public void afterReturning(String args);

    public void afterThrowing(String args);
}
```

（2）**不同的策略中心，配置不同的策略。**例如正常调用过程的普通策略中心，分别包含了鉴权策略，账户费用策略，流量控制策略，资源控制策略，计量策略。而测试策略中心则只包含鉴权策略、资源策略和测试计量策略。

```
<!-- 根据具体业务逻辑，注入真正接口调用访问控制策略 -->
<bean id="strategyContext" class="com.kd.openplatform.control.StrategyContext">
    <property name="strategies">
        <list>
            <ref bean="accessInterceptor"/>
            <ref bean="chargeInterceptor"/>
            <ref bean="flowInterceptor"/>
            <ref bean="resourceInterceptor"/>
            <ref bean="measureInterceptor"/>
        </list>
    </property>
</bean>
```

从面向对象的角度来看，每个策略都是一个封装好的对象，可以支持增加新的策略、替换原有的策略，每个策略之间相对独立，不会影响其它策略。

从面向切面的角度来看，访问控制模块保留了对 AOP 切到的各个过程的方法处理，每个策略都可以有（但是不必须有）自己对于被控制方法对应阶段（Before, Around, After）的处理策略。

出于访问控制的需要，任何一个优先级高的策略都可以通过抛出异常阻断之后策略的执行，这里好比一个关卡，一个关卡不过，直接拦截返回原因。

4. 工厂模式

意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

主要解决：主要解决**接口选择的问题**。

何时使用：我们明确地计划**不同条件下创建不同实例**时。

如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。

关键代码：创建过程在其子类执行。

优点： 1、一个调用者想创建一个对象，只要知道其名称就可以了。 2、扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。 3、屏蔽产品的具体实现，调用者只关心产品的接口。

我们的账户费用策略中，包含了在访问前对账户的订阅情况检查和余额检查，以及在访问后的计费和扣费。对于三种不同的计费方式（包年包月、按次计费、按流量计费）来说，**逻辑上属于同一个策略**，但是具体实施上，它们实现了不同的检查方法和计费方法。

此处用工厂模式符合其具备的两个优点：

（1）**策略中心只关心是否通过账户检查、是否成功计费，但并不关心实现的是哪一种方法**。就好比用户买一辆车，可以直接从工厂里提货，但是并不需要知道从哪个生产线产生出来的，由工厂来自己做具体的选择和实现。

（2）**扩展性高。**日后，如果业务拓展产生了其它的计费策略，可以通过实现同样的抽象接口来扩展。不需要修改已有的代码。

此处的工厂类调用通过反射来实现。