

# 蚂蚁科技

## 移动网关 使用指南

文档版本：20240808



# 法律声明

蚂蚁集团版权所有 © 2022，并保留一切权利。

未经蚂蚁集团事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

## 商标声明

 蚂蚁集团 ANT GROUP 及其他蚂蚁集团相关的商标均为蚂蚁集团所有。本文档涉及的第三方的注册商标，依法由权利人所有。

## 免责声明

由于产品版本升级、调整或其他原因，本文档内容有可能变更。蚂蚁集团保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在蚂蚁集团授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过蚂蚁集团授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

# 通用约定

格式	说明	样例
 危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	 危险 重置操作将丢失用户配置数据。
 警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	 警告 重启操作将导致业务中断，恢复业务时间约十分钟。
 注意	用于警示信息、补充说明等，是用户必须了解的内容。	 注意 权重设置为0，该服务器不会再接受新请求。
 说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	 说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击设置 > 网络 > 设置网络类型。
<b>粗体</b>	表示按键、菜单、页面名称等UI元素。	在结果确认页面，单击确定。
Courier字体	命令或代码。	执行 <code>cd /d C:/window</code> 命令，进入Windows系统文件夹。
<i>斜体</i>	表示参数、变量。	<code>bae log list --instanceid</code> <code>Instance_ID</code>
[ ] 或者 [a b]	表示可选项，至多选择一个。	<code>ipconfig [-all -t]</code>
{ } 或者 {a b}	表示必选项，至多选择一个。	<code>switch {active stand}</code>

# 目录

1.变更记录	07
2.移动网关简介	08
3.基本概念	09
4.快速开始	10
4.1. HTTP API	10
4.2. Dubbo API	10
4.3. HRPC API (仅专有云可用)	13
4.4. Dubbo API	16
4.5. TR API	19
5.接入客户端	23
5.1. 接入 Android	23
5.1.1. 快速开始	23
5.1.2. 进阶指南	24
5.2. 接入 iOS	25
5.2.1. 添加 SDK	25
5.2.2. 使用 SDK	26
5.3. 接入 HarmonyOS NEXT (beta)	29
5.3.1. 添加 SDK	29
5.3.2. 使用 SDK	30
5.4. H5 JS 编程	34
6.接入服务端	36
6.1. 后端签名校验说明	36
6.2. 服务定义与开发	37
6.3. 网关辅助类使用说明	40
7.使用控制台	44
7.1. 使用须知	44

---

7.2. 路由规则	44
7.3. API 分组	45
7.4. API 管理	48
7.4.1. 注册 API	48
7.4.2. 配置 API	49
7.4.2.1. 操作步骤	49
7.4.2.2. 基础信息配置	49
7.4.2.3. 高级配置	50
7.4.2.4. header 设置	50
7.4.2.5. 限流配置	50
7.4.2.6. 熔断配置	50
7.4.2.7. 缓存配置	51
7.4.2.8. 参数设置	51
7.4.3. API 授权	51
7.4.4. API 限流	52
7.4.5. API 缓存	53
7.4.6. API 模拟	54
7.4.7. 同步 API	55
7.4.8. 导出及导入 API	55
7.5. API 调用	55
7.5.1. API 测试	55
7.5.2. 生成代码	56
7.5.3. HTTP API 请求格式	57
7.6. 网关管理	57
7.6.1. 网关管理功能介绍	57
7.6.2. 数据加密	58
7.6.3. 跨域资源共享	59
7.7. 数据模型	61

---

7.8. API 分析	61
7.9. HarmonyOS NEXT 控制台使用 (beta)	61
7.10. 管理员操作	61
8. 网关异常排查	63
9. 常见问题	64
10. 参考	65
10.1. 网关结果码说明	65
10.2. 无线保镖结果码说明	67
10.3. 网关日志说明	69
10.3.1. 网关服务端日志	69
10.3.2. 网关 SPI 日志	71
10.4. 业务接口定义规范	72
10.5. 密钥生成方法	74
10.6. 网关签名机制	74

# 1. 变更记录

文档版本	变更内容
V20240116	<ul style="list-style-type: none"><li>新增 <a href="#">操作步骤</a> 章节。</li><li>新增 <a href="#">基础信息配置</a> 章节。</li><li>新增 <a href="#">高级配置</a> 章节。</li><li>新增 <a href="#">header 设置</a> 章节。</li><li>新增 <a href="#">限流配置</a> 章节。</li><li>新增 <a href="#">熔断配置</a> 章节。</li><li>新增 <a href="#">缓存配置</a> 章节。</li><li>新增 <a href="#">参数设置</a> 章节。</li><li>将 <a href="#">配置 API</a> 章节拆分。</li></ul>
V20221118	<ul style="list-style-type: none"><li>新增 <a href="#">HRPC API (仅专有云可用)</a> 章节。</li></ul>
V20220218	<ul style="list-style-type: none"><li>在 <a href="#">服务定义与开发</a> 章节中，新增 HRPC 基础依赖。</li><li>在 <a href="#">API 分组</a> 章节中，新增创建 HRPC 分组和配置 HRPC 分组。</li><li>在 <a href="#">注册 API</a> 章节中，新增添加 HRPC API。</li><li>在 <a href="#">配置 API</a> 章节中，配置信息的基础信息中添加 HRPC API 的基本信息。</li></ul>
V20211105	<ul style="list-style-type: none"><li>在 <a href="#">移动网关简介</a> 章节中，新增熔断能力和动态路由能力说明。</li><li>在 <a href="#">API 分组</a> 章节中，更新 Dubbo API 分组创建操作，补充多中心、注册中心鉴权信息。</li><li>在 <a href="#">配置 API</a> 章节中，新增熔断机制配置说明，更新参数设置说明。</li><li>新增 <a href="#">网关管理功能介绍</a> 章节中，新增对熔断机制的说明。</li><li>新增 <a href="#">路由规则</a> 章节，介绍如何配置、管理路由规则。</li></ul>
V20210630	<ul style="list-style-type: none"><li>在 <a href="#">服务定义与开发</a> 章节中，更新 Maven 依赖版本。</li><li>在 <a href="#">后端签名校验说明</a> 章节中，新增国密算法 (SM2/SM3) 验签代码示例。</li></ul>

## 2. 移动网关简介

移动网关服务 (Mobile Gateway Service, MGS) 是移动开发平台 (mPaaS) 提供的连接移动客户端与服务端的组件产品。该组件简化了移动端与服务端的数据协议和通讯协议, 能够显著提升开发效率和网络通讯效率。

### 功能特点

移动网关是连接移动客户端跟服务端的桥梁, 移动客户端通过网关来访问后台服务接口。移动网关能够:

- 自动生成客户端的 RPC 调用代码, 用户不需要关心网络通信、协议以及使用的数据格式。
- 将服务端返回的数据自动反解生成 Objective-C 对象, 无需额外编码。
- 提供数据压缩、缓存等增强服务。
- 统一进行异常处理, 如弹出对话框、Toast 提示框等。
- 支持 RPC 拦截器, 实现定制化的请求与处理。
- 实行统一的安全加密机制和防篡改的请求签名验证机制。
- 限流管控, 保护后台服务器。
- 提供熔断能力, 在后端系统出现异常时对后端进行保护。
- 具备动态路由能力, 支持动态配置路由规则。

### 价值优势

移动网关服务的优势在于:

- 简单配置即可适配多种终端, 连接异构的后端服务。
- 自动生成移动端 SDK, 实现前后端分离, 提升开发效率。
- 支持服务注册、发现与管控, 实现服务聚合与集成, 降低管理成本和安全风险。
- 提供优化后的数据协议与通讯协议, 提高网络通讯质量和效率。

### 应用场景

移动网关服务的应用场景如下:

- 开放移动服务能力  
随着移动互联网、普惠金融的迅猛发展, 企业越来越迫切地希望将现有成熟的后端服务开放出去。接入移动网关服务, 无需额外工作, 即可形成移动服务能力。
- 一套服务, 多端输出  
移动互联网时代, 服务需要支持多样化的终端设备, 这往往极大地增加了系统复杂性。企业只需在移动网关中定义服务, 便能支持多种终端接入。
- 异构服务, 建立标准统一的对外服务接口  
企业往往存在多种语言和结构的后端服务, 只需遵循一定的标准接入移动网关, 就可以对外开放标准统一的服务接口。

## 3. 基本概念

### API 分组

API 归属的分组 (API group) ，可以是具体的系统名、模块名或者抽象的标识。

### API 服务标识

API 服务标识 (OperationType) 是 API 服务的唯一标识，即创建 API 时输入的 OperationType。

### HRPC

HRPC 是基于 HTTP 实现的 RPC 方案。

### 工作空间标识

移动平台工作空间的标识 (WorkSpaceId) ，用于隔离不同的环境。可在控制台的下载配置文件页面中查看。

### MPC

MPC 为 mpaaschannel 的缩写，是 mPaaS 实现的一套 RPC 方案。

### 移动 App 标识

移动应用标识 (AppId) 是创建 mPaaS 应用时生成的标识。可在控制台的下载配置文件页面中查看。

### 移动网关服务

移动网关服务 (Mobile Gateway Service, MGS) 是提供网关 API 服务的组件名称。

## 4. 快速开始

### 4.1. HTTP API

快速开始指导您快速注册并发布一个供移动端调用的 HTTP 类型的 API 服务。整体过程分为五步：

1. [注册 API 分组](#)
2. [创建 API](#)
3. [配置 API 服务](#)
4. [测试 API 服务](#)
5. [生成客户端 SDK](#)

#### 准备

1. 登录控制台，在 [产品与服务](#) 中选择 [移动开发平台 mPaaS](#) 进入移动开发平台主页。
2. 切换至正确的工作空间后，单击需要接入 API 服务的 App 名称。
3. 在左侧导航栏选择 [移动网关](#)，进入移动网关配置页面。

#### 注册 API 分组

只有先注册了 API 分组，在后续创建完 API 并对其进行配置时，才可在 [接入系统](#) 中选择已创建的 API 分组，详见 [配置 API 服务](#) 中的截图。

1. 选择 [API 分组](#) 标签，进入 API 分组列表页。
2. 点击 [创建 API 分组](#) 按钮，在弹出的对话框中填写 API 分组信息。
  - **分组类型**：选择 HTTP。MGS 支持 HTTP、DUBBO、TR 三种类型。
  - **API 分组**：必填，提供服务的业务系统的名称，由字母或下划线开头，只支持字母、数字、下划线以及连字符。API 分组名要与注册的 API 服务应用名保持一致。
  - **服务地址**：必填，业务系统的 HTTP/HTTPS URL。
  - **超时时间**：选填，发送请求至业务系统时的超时时间，单位毫秒；默认值为 3000 ms。
3. 完成 API 分组信息配置后，点击 [确定](#) 完成分组创建。

如需进一步完善 [API 分组](#) 相关配置，请参考 [API 分组](#)。

#### 创建 API

1. 选择 [API 管理](#) 选项卡进入 API 列表页，单击 [创建 API](#) 按钮。
2. 在弹出的对话框中填写 API 信息。
  - **API 类型**：默认为 HTTP。
  - **添加方式**：目前只支持手动方式注册 HTTP API。
  - **operationType**：必填，当前环境和应用下 API 服务唯一标识。命名规则为 `组织.产品域.产品.子产品.操作`。
3. 单击 [确定](#) 按钮提交。

#### 配置 API 服务

1. 在 [API 管理](#) 选项卡中，单击 API 列表操作列中的 [配置](#)，进入 API 配置页面。
2. 在 API 配置区域，单击 [修改](#) 按钮进行相应参数的编辑；修改完成后，单击 [保存](#) 按钮。

#### 重要

- 为了快速入门，您可以先关闭 [高级配置](#) 中的 [签名校验](#) 开关。
- 关于签名校验的详细信息，请参见 [后端签名校验说明](#)。
- 关于 API 配置的详细信息，请参见 [配置 API](#)。

3. 打开右上方开关，使 API 服务处于 [开通](#) 状态。只有处于开通状态的 API 服务才能被调用。

#### 测试 API 服务

相关信息请参见 [API 测试](#)。

#### 生成客户端 SDK

相关信息请参见 [生成代码](#)。

#### 结果

完成上述几步操作，API 服务即可供客户端调用。有关客户端开发的更多信息，参见下列 [客户端开发指南](#)：

- [Android](#)
- [iOS](#)
- [H5 JS](#)

### 4.2. Dubbo API

Dubbo 服务仅适用于专有云。快速开始指导您注册并发布一个供移动端调用的 Dubbo 类型的 API 服务。整体过程分为六步：

1. [服务端开发](#)
2. [注册 API 分组](#)
3. [创建 API](#)
4. [配置 API 服务](#)
5. [测试 API 服务](#)
6. [生成客户端 SDK](#)

#### 服务端开发

#### 引入网关二方包

在项目的 `pom.xml` 文件中引入如下二方包（如原工程已经有依赖，请忽略）。其中 `mobilegw-unify` 系列依赖请使用最新版本，当前最新版本为 `1.0.5.20201010`。

```
<!-- mobilegw unify dependency-->
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-spi-dubbo</artifactId>
  <version>${the-latest-version}</version>
</dependency>
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-spi-adapter</artifactId>
  <version>${the-latest-version}</version>
</dependency>
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-log</artifactId>
  <version>${the-latest-version}</version>
</dependency>
<dependency>
  <groupId>com.alipay.hybirdpb</groupId>
  <artifactId>classparser</artifactId>
  <version>1.2.2</version>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.5</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.72_noneautotype</version>
</dependency>

<!-- 如果使用了pb，请加入如下依赖-->
<dependency>
  <groupId>com.google.protobuf</groupId>
  <artifactId>protobuf-java</artifactId>
  <version>2.6.1</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-core</artifactId>
  <version>1.3.8.20160722</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-runtime</artifactId>
  <version>1.3.8.20160722</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-api</artifactId>
  <version>1.3.8.20160722</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-collectionschema</artifactId>
  <version>1.3.8.20160722</version>
</dependency>

<!-- dubbo 使用 apache 最新版本-->
<dependency>
  <groupId>org.apache.dubbo</groupId>
  <artifactId>dubbo</artifactId>
  <version>2.7.8</version>
</dependency>
```

## 定义服务接口并实现接口

- 按照业务需求，定义服务接口：`com.alipay.xxx.MockRpc`。

建议将方法定义中的入参定义为 VO，这样后期可以直接在 VO 中添加参数，而不改变方法的声明格式。服务接口定义的相关规范，请参见 [业务接口定义规范](#)。

- 提供该接口的实现 `com.alipay.xxx.MockRpcImpl`。

## 定义 OperationType

在服务接口的方法上添加 `@OperationType` 注解，定义发布服务的接口名称。`@OperationType` 有三个参数成员，为便于维护，请填写完整：

- value**：接口唯一标识，在网关全局唯一。定义规则：`组织.产品域.产品.子产品.操作`。定义该参数值时应尽量详细，否则一旦与其他业务方的 value 值重复，会导致无法注册服务。
- name**：接口名称。
- desc**：接口描述。

示例如下：

```
public interface MockRpc {

    @OperationType(value="com.alipay.mock", name="DUBBO mock 接口", desc="复杂 mock 接口")
    Resp mock(Req s);

    @OperationType(value="com.alipay.mock2", name="xxx", desc="xxx")
    String mock2(String s);
}

public static class Resp {
    private String msg;
    private int code;

    // ignore getter & setter
}

public static class Req {
    private String name;
    private int age;

    // ignore getter & setter
}
```

## 声明 API 服务

该步骤目的是将定义好的 RPC 服务，通过网关提供的 `SPI` 包，声明为对外提供服务的 API。需要如下 2 个参数：

- **registryUrl**：注册中心的地址。
- **appName**：业务方的应用名。

您可以通过 `Spring` 或 `Spring Boot` 方式声明 API 服务。

### Spring 声明方式

1. 在对应 bundle 的 Spring 配置文件中，声明上述服务的 Spring Bean。示例如下：

```
<bean id="mockRpc" class="com.alipay.gateway.spi.dubbo.test.MockRpcImpl"/>
```

2. 在对应 bundle 的 Spring 配置文件中，声明 `com.alipay.gateway.spi.dubbo.DubboServiceStarter` 类型的 Spring Bean。`DubboServiceStarter` 会将所有带有 `@OperationType` 的 bean 通过 Dubbo 协议注册到指定的注册中心。示例如下：

```
<bean id="dubboServiceStarter" class="com.alipay.gateway.spi.dubbo.DubboServiceStarter">
  <property name="registryUrl" value="${registry_url}"/>
  <property name="appName" value="${app_name}"/>
</bean>
```

### Spring Boot 声明方式

1. 以注解的方式声明上述服务的 Spring Bean。示例如下：

```
@Service
public class MockRpcImpl implements MockRpc{
}
```

2. 以注解的方式声明 `com.alipay.gateway.spi.dubbo.DubboServiceStarter` 类型的 Spring Bean。`DubboServiceStarter` 会将所有带有 `@OperationType` 的 bean 通过 Dubbo 协议注册到指定的注册中心。示例如下：

```
@Configuration
public class DubboDemo {
    @Bean(name="dubboServiceStarter")
    public DubboServiceStarter dubboServiceStarter(){
        DubboServiceStarter dubboServiceStarter = new DubboServiceStarter();
        dubboServiceStarter.setAppName("${app_name}");
        dubboServiceStarter.setRegistryUrl("${registry_url}");
        return dubboServiceStarter;
    }
}
```

## 注册 API 分组

1. 进入移动网关管理页面。
  - i. 登录控制台，在 **产品与服务** 中选择 **移动开发平台 mPaaS** 进入移动开发平台主页。
  - ii. 切换至正确的工作空间后，单击需要接入 API 服务的 APP 名称。
  - iii. 在左侧导航栏选择 **后台服务管理 > 移动网关**，进入移动网关管理页面。
2. 选择 **API 分组** 选项卡进入 API 分组列表页，单击 **创建 API 分组** 按钮。
3. 在弹出的对话框中填写表单信息。
  - **分组类型**：此处选择 DUBBO。
  - **API 分组**：必填，提供服务的业务系统的英文名称。API 分组名称要与注册的 API 服务应用名保持一致。
  - **注册中心**：必填，注册中心地址。
  - **超时时间**：选填，发送请求至业务系统时的超时时间，单位毫秒，默认值为 3000 ms。
  - **注册中心**：填写注册中心地址，支持 ZooKeeper 集群或者直连。
  - **注册中心鉴权**：支持对注册中心进行权限管控，即只有通过鉴权的用户才可以访问注册中心。打开注册中心鉴权开关后，需要设置相应的用户名和密码。
4. 单击 **确定** 按钮提交。如需进一步完善 API 分组相关配置，请阅读 [配置分组](#)。

### 创建 API

1. 选择 **API 管理** 选项卡进入 API 列表页，单击 **创建 API** 按钮。
2. 在弹出的对话框中，**API 类型** 选择 **DUBBO**，选择 **API 分组**，在拉取到的 `operationType` 列表中勾选需要的服务，单击 **确认** 按钮。

### 配置 API 服务

1. 在 **API 管理** 选项卡中，单击 API 列表操作列中的 **配置**，进入 API 配置页面。
2. 在 API 配置区域，单击 **修改** 按钮进行相应参数的编辑；修改完成后，单击 **保存** 按钮。

#### ⓘ 重要

- 为了快速入门，您可以先关闭 **高级配置** 中的 **签名校验** 开关。关于签名校验的详细信息，请参见 [后端签名校验说明](#)。
- 关于 API 配置的详细信息，请参见 [配置 API](#)。

3. 检查右上方开关，保证 API 服务处于 **开通** 状态。只有处于开通状态的 API 服务才能被调用。

### 测试 API 服务

相关信息请参见 [API 测试](#)。

### 生成客户端 SDK

相关信息请参见 [生成代码](#)。

### 结果

完成上述几步操作，API 服务即可供客户端调用。有关客户端开发的更多信息，参见下列客户端开发指南：

- [Android](#)
- [iOS](#)
- [H5 JS](#)

## 4.3. HRPC API（仅专有云可用）

HRPC 服务仅适用于 **专有云**。快速开始指导您注册并发布一个供移动端调用的 HRPC 类型的 API 服务。整体过程分为六步：

1. [服务端开发](#)
2. [注册 API 分组](#)
3. [注册 API 服务](#)
4. [配置 API 服务](#)
5. [测试 API 服务](#)
6. [生成客户端 SDK](#)

### 服务端开发

#### 引入网关二方包

在项目的主 `pom.xml` 文件中引入如下二方包（如原工程已经有依赖，请忽略）。`mobilegw-unify` 系列依赖请使用最新版本，当前最新版本为 1.0.5.20200930。

```
<!-- mobilegw unify dependency-->
<dependency>
<groupId>com.alipay.gateway</groupId>
<artifactId>mobilegw-unify-spi-hrpc</artifactId>
<version>${the-latest-version}</version>
</dependency>
<dependency>
<groupId>com.alipay.gateway</groupId>
<artifactId>mobilegw-unify-spi-adapter</artifactId>
<version>${the-latest-version}</version>
</dependency>
<dependency>
V20210317/移动网关
第 5 页
按照业务需求，定义服务接口：com.alipay.xxxx.MockRpc。
<groupId>com.alipay.gateway</groupId>
<artifactId>mobilegw-unify-log</artifactId>
<version>${the-latest-version}</version>
</dependency>
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>hessian</artifactId>
<version>3.3.6</version>
</dependency>
<dependency>
<groupId>com.alipay.hybirdpb</groupId>
<artifactId>classparser</artifactId>
<version>1.2.2</version>
</dependency>
<dependency>
<groupId>org.apache.commons</groupId>
<artifactId>commons-lang3</artifactId>
<version>3.5</version>
</dependency>
<dependency>
<groupId>com.alibaba</groupId>
<artifactId>fastjson</artifactId>
<version>1.2.69_noneautotype</version>
</dependency>
<!-- 如果使用了pb，请加入如下依赖-->
<dependency>
<groupId>com.google.protobuf</groupId>
<artifactId>protobuf-java</artifactId>
<version>2.6.1</version>
</dependency>
<dependency>
<groupId>io.protostuff</groupId>
<artifactId>protostuff-core</artifactId>
<version>1.3.8.20160722</version>
</dependency>
<dependency>
<groupId>io.protostuff</groupId>
<artifactId>protostuff-runtime</artifactId>
<version>1.3.8.20160722</version>
</dependency>
<dependency>
<groupId>io.protostuff</groupId>
<artifactId>protostuff-api</artifactId>
<version>1.3.8.20160722</version>
</dependency>
<dependency>
<groupId>io.protostuff</groupId>
<artifactId>protostuff-collectionschema</artifactId>
<version>1.3.8.20160722</version>
</dependency>
```

## 定义服务接口并实现接口

- 按照业务需求，定义服务接口 `com.alipay.xxxx.MockRpc`。

### ② 说明

- 建议将方法定义中的入参定义为 VO，这样后期可以直接在 VO 中添加参数，而不改变方法的声明格式。
- 服务接口定义的相关规范，请参见 [业务接口定义规范](#)。

- 该接口的实现为 `com.alipay.xxxx.MockRpcImpl`。

## 定义 OperationType

在服务接口的方法上添加 `@OperationType` 注解，定义发布服务的接口名称。`@OperationType` 有三个参数成员，为便于维护，请填写完整：

- value**：接口唯一标识，在网关全局唯一。定义规则：`组织.产品域.产品.子产品.操作`。定义该参数值时应尽量详细，否则一旦与其他业务方的 value 值重复，会导致无法注册服务。
- name**：接口名称。
- desc**：接口描述。

示例如下：

```
public interface MockRpc {
    @OperationType(value="com.alipay.mock", name="HRPC mock 接口", desc="复杂 mock 接口")
    Resp mock(Req s);
    @OperationType(value="com.alipay.mock2",name="xxx", desc="xxx")
    String mock2(String s);
}

public static class Resp {
    private String msg;
    private int code;
    // ignore getter & setter
}

public static class Req {
    private String name;
    private int age;
    // ignore getter & setter
}
```

## 声明 API 服务

该步骤目的是将定义好的 RPC 服务，通过网关提供的 SPI 包，声明为对外提供服务的 API。需要如下三个参数：

- **registryUrl**：必填，注册中心的地址。
- **appName**：必填，业务方的应用名。
- **serverPort**：选填，HRPC 服务监听端口，默认为 7079。

您可以通过 `Spring` 或 `Spring Boot` 方式声明 API 服务。

### Spring 声明方式

1. 在对应 bundle 的 Spring 配置文件中，声明上述服务的 Spring Bean。示例如下：

```
<bean id="mockRpc" class="com.alipay.gateway.spi.hrpc.test.MockRpcImpl"/>
```

2. 在对应 bundle 的 Spring 配置文件中，声明 `com.alipay.gateway.spi.hrpc.HRpcServiceStarter` 类型的 Spring Bean。`HRpcServiceStarter` 会将所有带有 `@OperationType` 的 bean 通过 HRPC 协议注册到指定的注册中心。示例如下：

```
<bean id="hrpcServiceStarter" class="com.alipay.gateway.spi.hrpc.HRpcServiceStarter">
  <property name="registryUrl" value="${registry_url}"/>
  <property name="appName" value="${app_name}"/>
  <property name="serverPort" value="${serverPort可选, 默认为 7079}"/>
</bean>
```

### Spring Boot 声明方式

1. 以注解的方式声明上述服务的 Spring Bean。示例如下：

```
@Service
public class MockRpcImpl implements MockRpc{
}
```

2. 以注解的方式声明 `com.alipay.gateway.spi.hrpc.HRpcServiceStarter` 类型的 Spring Bean。`HRpcServiceStarter` 会将所有带有 `@OperationType` 的 bean 通过 HRPC 协议注册到指定的注册中心。示例如下：

```
@Configuration
public class HRpcDemo {
    @Bean(name="hrpcServiceStarter")
    public HRpcServiceStarter hrpcServiceStarter(){
        HRpcServiceStarter hrpcServiceStarter = new HRpcServiceStarter();
        hrpcServiceStarter.setAppName("${app_name}");
        hrpcServiceStarter.setRegistryUrl("${registry_url}");
        hrpcServiceStarter.setServerPort("${serverPort可选}");
        return hrpcServiceStarter;
    }
}
```

## 注册 API 分组

只有先注册了 API 分组，在后续创建完 API 并对其进行配置时，才可在接入系统中选择已创建的 API 分组。

1. 进入移动网关管理页面。
  - i. 登录控制台，在 **产品与服务** 中选择 **移动开发平台 mPaaS** 进入移动开发平台主页。
  - ii. 切换至正确的工作空间后，单击需要接入 API 服务的 App 名称。
  - iii. 在左侧导航栏选择 **后台服务管理 > 移动网关**，进入移动网关管理页面。
2. 选择 **API 分组** 选项卡进入 API 分组列表页，单击 **创建 API 分组** 按钮。
3. 在弹出的对话框中填写表单信息。
  - **分组类型**：此处选择 HRPC。
  - **API 分组**：必填，提供服务的业务系统的名称，由字母或下划线开头，只支持字母、数字、下划线以及连字符。
  - **注册中心**：必填，注册中心 URL。
  - **超时时间**：选填，发送请求至业务系统时的超时时间，单位毫秒，默认值为 3000 ms。
4. 单击 **确定** 按钮提交。如需进一步完善 API 分组相关配置，请阅读 [配置分组](#)。

### 注册 API 服务

1. 选择 **API 管理** 选项卡进入 API 列表页，单击 **创建 API** 按钮。
2. 在弹出的对话框中，**API 类型** 选择 **HRPC**，选择 API 分组，在拉取到的 **operationType** 列表中勾选需要的服务，单击 **确认** 按钮。

### 配置 API 服务

1. 单击 API 列表操作列中的 **配置**，进入 API 配置页面。
2. 在 API 配置区域，单击 **修改** 按钮进行相应参数的编辑；修改完成后，单击 **保存** 按钮。

#### ② 说明

- 为了快速入门，您可以先将 **高级配置** 中的 **签名校验** 关闭。关于签名校验的详细信息，请参见 [后端签名校验说明](#)。
- 关于 API 配置的详细信息，请参见 [配置 API](#)。

3. 检查右上方开关，保证 API 服务处于 **开通** 状态。只有处于开通状态的 API 服务才能被调用。

### 测试 API 服务

相关信息请参见 [API 测试](#)。

### 生成客户端 SDK

相关信息请参见 [生成代码](#)。

### 结果

完成上述几步操作，API 服务即可供客户端调用。有关客户端开发的更多信息，参见下列客户端开发指南：

- [Android](#)
- [iOS](#)
- [H5 JS](#)

## 4.4. Dubbo API

Dubbo 服务仅适用于专有云。快速开始指导您注册并发布一个供移动端调用的 Dubbo 类型的 API 服务。整体过程分为六步：

1. [服务端开发](#)
2. [注册 API 分组](#)
3. [创建 API](#)
4. [配置 API 服务](#)
5. [测试 API 服务](#)
6. [生成客户端 SDK](#)

### 服务端开发

#### 引入网关二方包

在项目的主 `pom.xml` 文件中引入如下二方包（如原工程已经有依赖，请忽略）。其中 `moblegw-unify` 系列依赖请使用最新版本，当前最新版本为 `1.0.5.20201010`。

```
<!-- mobilegw unify dependency-->
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-spi-dubbo</artifactId>
  <version>${the-latest-version}</version>
</dependency>
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-spi-adapter</artifactId>
  <version>${the-latest-version}</version>
</dependency>
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-log</artifactId>
  <version>${the-latest-version}</version>
</dependency>
<dependency>
  <groupId>com.alipay.hybirdpb</groupId>
  <artifactId>classparser</artifactId>
  <version>1.2.2</version>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.5</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.72_noneautotype</version>
</dependency>

<!-- 如果使用了pb，请加入如下依赖-->
<dependency>
  <groupId>com.google.protobuf</groupId>
  <artifactId>protobuf-java</artifactId>
  <version>2.6.1</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-core</artifactId>
  <version>1.3.8.20160722</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-runtime</artifactId>
  <version>1.3.8.20160722</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-api</artifactId>
  <version>1.3.8.20160722</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-collectionschema</artifactId>
  <version>1.3.8.20160722</version>
</dependency>

<!-- dubbo 使用 apache 最新版本-->
<dependency>
  <groupId>org.apache.dubbo</groupId>
  <artifactId>dubbo</artifactId>
  <version>2.7.8</version>
</dependency>
```

## 定义服务接口并实现接口

1. 按照业务需求，定义服务接口：`com.alipay.xxxx.MockRpc`。

建议将方法定义中的入参定义为 VO，这样后期可以直接在 VO 中添加参数，而不改变方法的声明格式。服务接口定义的相关规范，请参见 [业务接口定义规范](#)。

2. 提供该接口的实现 `com.alipay.xxxx.MockRpcImpl`。

## 定义 OperationType

在服务接口的方法上添加 `@OperationType` 注解，定义发布服务的接口名称。`@OperationType` 有三个参数成员，为便于维护，请填写完整：

- **value**：接口唯一标识，在网关全局唯一。定义规则：`组织.产品域.产品.子产品.操作`。定义该参数值时应尽量详细，否则一旦与其他业务方的 value 值重复，会导致无法注册服务。
- **name**：接口名称。
- **desc**：接口描述。

示例如下：

```
public interface MockRpc {

    @OperationType(value="com.alipay.mock", name="DUBBO mock 接口", desc="复杂 mock 接口")
    Resp mock(Req s);

    @OperationType(value="com.alipay.mock2", name="xxx", desc="xxx")
    String mock2(String s);
}

public static class Resp {
    private String msg;
    private int code;

    // ignore getter & setter
}

public static class Req {
    private String name;
    private int age;

    // ignore getter & setter
}
```

## 声明 API 服务

该步骤目的是将定义好的 RPC 服务，通过网关提供的 `SPI` 包，声明为对外提供服务的 API。需要如下 2 个参数：

- **registryUrl**：注册中心的地址。
- **appName**：业务方的应用名。

您可以通过 `Spring` 或 `Spring Boot` 方式声明 API 服务。

### Spring 声明方式

1. 在对应 bundle 的 Spring 配置文件中，声明上述服务的 Spring Bean。示例如下：

```
<bean id="mockRpc" class="com.alipay.gateway.spi.dubbo.test.MockRpcImpl"/>
```

2. 在对应 bundle 的 Spring 配置文件中，声明 `com.alipay.gateway.spi.dubbo.DubboServiceStarter` 类型的 Spring Bean。`DubboServiceStarter` 会将所有带有 `@OperationType` 的 bean 通过 Dubbo 协议注册到指定的注册中心。示例如下：

```
<bean id="dubboServiceStarter" class="com.alipay.gateway.spi.dubbo.DubboServiceStarter">
  <property name="registryUrl" value="${registry_url}"/>
  <property name="appName" value="${app_name}"/>
</bean>
```

### Spring Boot 声明方式

1. 以注解的方式声明上述服务的 Spring Bean。示例如下：

```
@Service
public class MockRpcImpl implements MockRpc{
}
```

2. 以注解的方式声明 `com.alipay.gateway.spi.dubbo.DubboServiceStarter` 类型的 Spring Bean。`DubboServiceStarter` 会将所有带有 `@OperationType` 的 bean 通过 Dubbo 协议注册到指定的注册中心。示例如下：

```
@Configuration
public class DubboDemo {
    @Bean(name="dubboServiceStarter")
    public DubboServiceStarter dubboServiceStarter(){
        DubboServiceStarter dubboServiceStarter = new DubboServiceStarter();
        dubboServiceStarter.setAppName("${app_name}");
        dubboServiceStarter.setRegistryUrl("${registry_url}");
        return dubboServiceStarter;
    }
}
```

## 注册 API 分组

1. 进入移动网关管理页面。
  - i. 登录控制台，在 **产品与服务** 中选择 **移动开发平台 mPaaS** 进入移动开发平台主页。
  - ii. 切换至正确的工作空间后，单击需要接入 API 服务的 APP 名称。
  - iii. 在左侧导航栏选择 **后台服务管理 > 移动网关**，进入移动网关管理页面。
2. 选择 **API 分组** 选项卡进入 API 分组列表页，单击 **创建 API 分组** 按钮。
3. 在弹出的对话框中填写表单信息。
  - **分组类型**：此处选择 DUBBO。
  - **API 分组**：必填，提供服务的业务系统的英文名称。API 分组名称要与注册的 API 服务应用名保持一致。
  - **注册中心**：必填，注册中心地址。
  - **超时时间**：选填，发送请求至业务系统时的超时时间，单位毫秒，默认值为 3000 ms。
  - **注册中心**：填写注册中心地址，支持 ZooKeeper 集群或者直连。
  - **注册中心鉴权**：支持对注册中心进行权限管控，即只有通过鉴权的用户才可以访问注册中心。打开注册中心鉴权开关后，需要设置相应的用户名和密码。
4. 单击 **确定** 按钮提交。如需进一步完善 API 分组相关配置，请阅读 [配置分组](#)。

### 创建 API

1. 选择 **API 管理** 选项卡进入 API 列表页，单击 **创建 API** 按钮。
2. 在弹出的对话框中，**API 类型** 选择 **DUBBO**，选择 **API 分组**，在拉取到的 `operationType` 列表中勾选需要的服务，单击 **确认** 按钮。

### 配置 API 服务

1. 在 **API 管理** 选项卡中，单击 API 列表操作列中的 **配置**，进入 API 配置页面。
2. 在 API 配置区域，单击 **修改** 按钮进行相应参数的编辑；修改完成后，单击 **保存** 按钮。

#### ⓘ 重要

- 为了快速入门，您可以先关闭 **高级配置** 中的 **签名校验** 开关。关于签名校验的详细信息，请参见 [后端签名校验说明](#)。
- 关于 API 配置的详细信息，请参见 [配置 API](#)。

3. 检查右上方开关，保证 API 服务处于 **开通** 状态。只有处于开通状态的 API 服务才能被调用。

### 测试 API 服务

相关信息请参见 [API 测试](#)。

### 生成客户端 SDK

相关信息请参见 [生成代码](#)。

### 结果

完成上述几步操作，API 服务即可供客户端调用。有关客户端开发的更多信息，参见下列客户端开发指南：

- [Android](#)
- [iOS](#)
- [H5 JS](#)

## 4.5. TR API

TR 是蚂蚁金服 RPC 框架，仅适用于专有云。快速开始指导您注册并发布一个供移动端调用的 TR 类型的 API 服务。整体过程分为六步：

1. [服务端开发](#)
2. [注册 API 分组](#)
3. [创建 API](#)
4. [配置 API 服务](#)
5. [测试 API 服务](#)
6. [生成客户端 SDK](#)

### 服务端开发

#### 引入网关二方包

在项目的主 `pom.xml` 文件中引入如下二方包（如原工程已经有依赖，请忽略）。`mobilegw-unify` 系列依赖请使用最新版本，当前最新版本为 `1.0.5.20201010`。

```
<!-- mobilegw unify dependency-->
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-spi-sofa</artifactId>
  <version>${the-latest-version}</version>
  <exclusions>
    <exclusion>
      <groupId>hessian</groupId>
      <artifactId>hessian</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-spi-adapter</artifactId>
  <version>${the-latest-version}</version>
</dependency>
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-log</artifactId>
  <version>${the-latest-version}</version>
</dependency>
<dependency>
  <groupId>com.alipay.hybirdpb</groupId>
  <artifactId>classparser</artifactId>
  <version>1.2.2</version>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.5</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.72_noneautotype</version>
</dependency>
<dependency>
  <groupId>com.alipay.sofa</groupId>
  <artifactId>hessian</artifactId>
  <version>3.3.6</version>
</dependency>

<!-- 如果使用了pb，请加入如下依赖-->
<dependency>
  <groupId>com.google.protobuf</groupId>
  <artifactId>protobuf-java</artifactId>
  <version>2.6.1</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-core</artifactId>
  <version>1.3.8.20160722</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-runtime</artifactId>
  <version>1.3.8.20160722</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-api</artifactId>
  <version>1.3.8.20160722</version>
</dependency>
<dependency>
  <groupId>io.protostuff</groupId>
  <artifactId>protostuff-collectionschema</artifactId>
  <version>1.3.8.20160722</version>
</dependency>
```

## 定义服务接口并实现接口

1. 按照业务需求，定义服务接口：`com.alipay.xxx.MockRpc`。

建议将方法定义中的入参定义为 VO，这样后期可以直接在 VO 中添加参数，而不改变方法的声明格式。服务接口定义的相关规范，请参见 [业务接口定义规范](#)。

2. 提供该接口的实现 `com.alipay.xxx.MockRpcImpl`。

## 定义 OperationType

在服务接口的方法上添加 `@OperationType` 注解，定义发布服务的接口名称。`@OperationType` 有 3 个参数成员，为便于维护，请填写完整：

- **value**：接口唯一标识，在网关全局唯一。定义规则：`组织.产品域.产品.子产品.操作`。定义该参数值时应尽量详细，否则一旦与其他业务方的 value 值重复，会导致无法注册服务。
- **name**：接口中文名称。
- **desc**：接口描述。

示例如下：

```
public interface MockRpc {

    @OperationType(value="com.alipay.mock", name="MPC mock 接口", desc="复杂 mock 接口")
    Resp mock(Req s);

    @OperationType(value="com.alipay.mock2", name="xxx", desc="xxx")
    String mock2(String s);
}

public static class Resp {
    private String msg;
    private int code;

    // ignore getter & setter
}

public static class Req {
    private String name;
    private int age;

    // ignore getter & setter
}
```

## 声明 API 服务

该步骤目的是将定义好的 RPC 服务，通过网关提供的 `SPI` 包，声明为对外提供服务的 API。需要填写参数 `appName`，参数值为业务方的应用名。

您可以通过 `Spring` 或 `Spring Boot` 方式声明 API 服务。

### Spring 声明方式

1. 在对应 bundle 的 Spring 配置文件中，声明上述服务的 Spring Bean。示例如下：

```
<bean id="mockRpc" class="com.alipay.gateway.spi.mpc.test.MockRpcImpl"/>
```

2. 在对应 bundle 的 Spring 配置文件中，声明 `com.alipay.gateway.spi.sofa.SofaServiceStarter` 类型的 Spring Bean。`SofaServiceStarter` 会将所有带有 `@OperationType` 的 bean 通过 TR 协议暴露给网关调用。示例如下：

```
<bean id="sofaServiceStarter" class="com.alipay.gateway.spi.sofa.SofaServiceStarter">
  <property name="appName" value="${app_name}"/>
</bean>
```

### Spring Boot 声明方式

1. 以注解的方式声明上述服务的 Spring Bean。示例如下：

```
@Service
public class MockRpcImpl implements MockRpc{
}
```

2. 以注解的方式声明 `com.alipay.gateway.spi.sofa.SofaServiceStarter` 类型的 Spring Bean。`SofaServiceStarter` 会将所有带有 `@OperationType` 的 bean 通过 TR 协议暴露给网关调用。示例如下：

```
@Configuration
public class TRDemo {
    @Bean(name="sofaServiceStarter")
    public SofaServiceStarter sofaServiceStarter(){
        SofaServiceStarter sofaServiceStarter = new SofaServiceStarter();
        sofaServiceStarter.setAppName("${app_name}");
        return sofaServiceStarter;
    }
}
```

## 注册 API 分组

1. 进入移动网关管理页面。
  - i. 登录控制台，在 **产品与服务** 中选择 **移动开发平台 mPaaS** 进入移动开发平台主页。
  - ii. 切换至正确的工作空间后，点击需要接入 API 服务的 APP 名称。
  - iii. 在左侧导航栏选择 **后台服务管理 > 移动网关**，进入移动网关管理页面。
2. 选择 **API 分组** 选项卡进入 API 分组列表页，点击 **添加 API 分组** 按钮。
3. 在弹出的对话框中填写表单信息。
  - **分组类型**：此处选择 **TR**。
  - **API 分组**：必填，提供服务的业务系统的英文名称。API 分组名称要与注册的 API 服务应用名保持一致。
  - **直连地址**：选填，需要直连时填写。由 IP 和端口组成，端口不指定时默认为 `12200`。
  - **超时时间**：选填，发送请求至业务系统时的超时时间，单位毫秒，默认值为 `3000 ms`。
4. 点击 **确定** 按钮提交。如需进一步完善 API 分组相关配置，请阅读 [配置分组](#)。

### 创建 API

1. 选择 **API 管理** 选项卡进入 API 列表页，点击 **添加 API** 按钮。
2. 在弹出的对话框中，**API 类型** 选择 **TR**，选择 **API 分组**，在拉取到的 operationType 列表中勾选需要的服务，点击 **确认** 按钮。

### 配置 API 服务

1. 点击 API 列表操作列中的 **配置**，进入 API 配置页面。
2. 在 API 配置区域，点击 **修改** 按钮进行相应参数的编辑；修改完成后，点击 **保存** 按钮。

① 重要

- 为了快速入门，您可以先关闭 高级配置 中的 签名校验 开关。关于签名校验的详细信息，请参见 [后端签名校验说明](#)。
- 关于 API 配置的详细信息，请参见 [配置 API](#)。

3. 检查右上方开关，保证 API 服务处于 开通 状态。只有处于开通状态的 API 服务才能被调用。

### 测试 API 服务

相关信息请参见 [API 测试](#)。

### 生成客户端 SDK

相关信息请参见 [生成代码](#)。

### 结果

完成上述几步操作，API 服务即可供客户端调用。有关客户端开发的更多信息，参见下列客户端开发指南：

- [Android](#)
- [iOS](#)
- [H5 JS](#)

## 5. 接入客户端

### 5.1. 接入 Android

#### 5.1.1. 快速开始

网关是连接客户端与服务端的桥梁，客户端通过网关来访问后台服务接口。

通过使用网关，您可以实现以下目的：

- 通过动态代理的方式，封装客户端和服务端之间的通讯。
- 如果服务端和客户端定义了一致的接口，可由服务端自动生成代码并导出给客户端使用。
- 对 `RpcException` 进行统一的异常处理，弹对话框、toast 消息框等。

移动网关支持 原生 AAR 接入 和 组件化 (Portal&Bundle) 接入 两种接入方式。

##### 前置条件

- 若采用原生 AAR 方式接入，需要先。
- 若采用组件化方式接入，需要先完成。

##### 添加 SDK

###### 原生 AAR 方式

参考 [管理组件依赖 \(原生 AAR\)](#)，通过 [组件管理 \(AAR\)](#) 在工程中安装 移动网关 组件。

###### 组件化 (Portal&Bundle) 方式

在 Portal 和 Bundle 工程中通过 [组件管理](#) 安装 移动网关 组件。更多信息，请参考 [组件化方式接入流程](#)。

##### 初始化 mPaaS

如果您使用 原生 AAR 接入，您需要初始化 mPaaS。

```
public class MyApplication extends Application {  
  
    @Override  
    public void onCreate() {  
        super.onCreate();  
        // mPaaS 初始化  
        MP.init(this);  
    }  
}
```

详情请参考：[初始化 mPaaS](#)。

##### 生成 RPC 代码

当 App 在移动网关控制台接入后台服务后，进入 mPaaS 控制台，从左侧导航栏选择 移动网关 > API 管理 > 生成代码，下载客户端的 RPC 代码。详细说明参见 [注册 API](#) 相关文档。

下载的 RPC 代码结构如下，包括 RPC 配置、request 模型和 response 模型。



##### 调用 RPC

客户端发起 RPC 调用。代码示例如下：

```
// 获取 client 实例  
RpcDemoClient client = MPRpc.getRpcProxy(RpcDemoClient.class);  
// 设置请求  
GetIdGetReq req = new GetIdGetReq();  
req.id = "123";  
req.age = 14;  
req.isMale = true;  
// 发起 RPC 请求  
try {  
    String response = client.getIdGet(req);  
} catch (RpcException e) {  
    // 处理 RPC 请求异常  
    Log.i("RpcException", "code: " + e.getCode() + " msg: " + e.getMsg());  
}
```

RPC 调用异常会统一通过 `RpcException` 抛出来，可根据 `RpcException` 的 `code` 结果码做相应处理，错误码详情参见 [网关结果码说明](#)。

##### 相关链接

- [代码示例](#)
- [网关结果码说明](#)
- [密钥生成方法](#)

## 5.1.2. 进阶指南

本文对移动网关 RPC 拦截器、RPC 请求头、RPC Cookie、RPC 签名的设置进行说明。

### 🔔 重要

在 10.2.3 基线中新增设置 RPC 签名内容。

### RPC 拦截

在业务开发中，如果在某些情况下需要控制客户端的网络请求（例如拦截网络请求，禁止访问某些接口，或者限流），可以通过 RPC 拦截器实现。

#### 创建全局拦截器

```
public class CommonInterceptor implements RpcInterceptor {

    /**
     * 前置拦截：发送 RPC 之前回调。
     * @param proxy RPC 代理对象。
     * @param clazz rpcface 模型类，通过 clazz 参数可以判断当前调用的是哪个 RPC 模型类
     * @param method 当前 RPC 调用的方法。
     * @throws RpcException
     * @return true 表示继续向下执行，false 表示中断当前请求，抛出 RpcException，错误码：9。
     */
    @Override
    public boolean preHandle(Object proxy,
                            ThreadLocal<Object> retValue,
                            byte[] retRawValue,
                            Class<?> clazz,
                            Method method,
                            Object[] args,
                            Annotation annotation,
                            ThreadLocal<Map<String, Object>> extParams)
        throws RpcException {

        //Do something...
        return true;
    }

    /**后置拦截：发起 RPC 成功之后回调。
     * @return true 表示继续向下执行，false 表示中断当前请求，抛出 RpcException，错误码：9。
     */
    @Override
    public boolean postHandle(Object proxy,
                              ThreadLocal<Object> retValue,
                              byte[] retRawValue,
                              Class<?> clazz,
                              Method method,
                              Object[] args,
                              Annotation annotation) throws RpcException {

        //Do something...
        return true;
    }

    /**
     * 异常拦截：发起 RPC 失败之后回调。
     * @param exception 表示当前 RPC 出错异常。
     * @return true 表示将当前异常继续向上抛出，false 表示不要抛出异常，正常返回，没有特殊需求，切勿返回 false。
     */
    @Override
    public boolean exceptionHandle(Object proxy,
                                   ThreadLocal<Object> retValue,
                                   byte[] retRawValue,
                                   Class<?> clazz,
                                   Method method,
                                   Object[] args,
                                   RpcException exception,
                                   Annotation annotation) throws RpcException {

        //Do something...
        return true;
    }
}
```

#### 注册拦截器

在框架启动过程中，初始化 `RpcService` 时，将拦截器注册上去，例如：

```
public class MockLauncherApplicationAgent extends LauncherApplicationAgent {

    public MockLauncherApplicationAgent(Application context, Object bundleContext) {
        super(context, bundleContext);
    }

    @Override
    public void preInit() {
        super.preInit();
    }

    @Override
    public void postInit() {
        super.postInit();
        RpcService rpcService = getMicroApplicationContext().findServiceByInterface(RpcService.class.getName());
        rpcService.addRpcInterceptor(OperationType.class, new CommonInterceptor());
    }
}
```

## 设置 RPC 请求头

在 `MainActivity` 类的 `initRpcConfig` 方法中，设置 RPC 请求头。具体参考 [代码示例](#)。

```
private void initRpcConfig(RpcService rpcService) {
    //设置请求头
    Map<String, String> headerMap = new HashMap<>();
    headerMap.put("key1", "val1");
    headerMap.put("key2", "val2");
    rpcInvokeContext.setRequestHeaders(headerMap);
}
```

## 设置 RPC cookie

### 设置 cookie

通过调用以下接口来进行 RPC cookie 设置。其中，`Your domain` 的规则是网关 URL 的第一个 `.` 以及其后第一个 `/` 之前的所有内容。例如，网关 URL 为 `http://test-cn-hangzhou-mgs-gw.cloud.alipay.com/mgw.htm`，那么 `Your domain` 则是 `.cloud.alipay.com`。

```
GwCookieCacheHelper.setCookies(Your domain, cookiesMap);
```

### 移除 cookie

通过调用以下接口即可移除设置的 cookie。

```
GwCookieCacheHelper.removeAllCookie();
```

## 设置 SM3 验签

在 RPC 初始化之后，可以通过 `MpRpc` 类的 `setGlobalSignType` 方法，来指定全局验签方式为 sm3 类型。

```
MpRpc.setGlobalSignType(TransportConstants.SIGN_TYPE_SM3);
```

## 设置 RPC 签名

### 接口

```
class TransportConstants {
    public static final int SIGN_TYPE_DEFAULT = 0; // 默认签名方式，即 md5
    public static final int SIGN_TYPE_MD5 = 1; // md5
    public static final int SIGN_TYPE_HMACSHA256 = 3; // hmacsha256
    public static final int SIGN_TYPE_SHA256 = 4; // sha256
    public static final int SIGN_TYPE_SM3 = 5; // sm3
}

// 全局 rpc 签名算法设置
MpRpc.setGlobalSignType(int signType);
```

### 示例

设置全局 RPC signType，且对所有 RPC 生效。

```
// 设置签名方式为SM3
MpRpc.setGlobalSignType(TransportConstants.SIGN_TYPE_SM3);
```

## 5.2. 接入 iOS

### 5.2.1. 添加 SDK

本文介绍如何快速将移动网关组件接入到 iOS 客户端。移动网关支持基于 mPaaS 框架接入、基于已有工程且使用 mPaaS 插件接入以及基于已有工程且使用 CocoaPods 接入三种接入方式。您可以参考 [接入方式介绍](#)，根据实际情况选择合适的接入方式。

#### 前置条件

添加移动网关 SDK 前，确保已完成以下操作：

- 接入工程到 mPaaS：
  - [基于 mPaaS 框架接入](#)
  - [基于已有工程且使用 mPaaS 插件接入](#)
  - [基于已有工程且使用 CocoaPods 接入](#)
- 引入用于 [请求加签](#) 的无线保镭图片 `yw_1222.jpg`：
  - 公有云：完成第 1 步后，工程中会自动生成无线保镭图片，您无需额外操作。
  - 专有云：参考 [生成无线保镭图片](#) 生成专有云无线保镭图片。

### 添加 SDK

根据您采用的接入方式，请选择相应的添加方式。

#### 使用 mPaaS Xcode Extension 插件

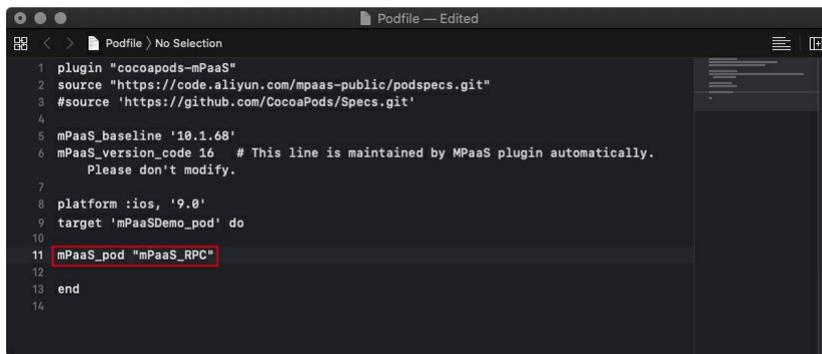
此方式适用于 [基于 mPaaS 框架接入](#) 或 [基于已有工程且使用 mPaaS 插件接入](#) 的接入方式。

- 点击 Xcode 菜单项 **Editor > mPaaS > 编辑工程**，打开编辑工程页面。
- 选择 **移动网关**，保存后点击 **开始编辑**，即可完成添加。

#### 使用 cocoapods-mPaaS 插件

此方式适用于 [基于已有工程且使用 CocoaPods 接入](#) 的接入方式。

- 在 Podfile 文件中，使用 `mPaaS_pod "mPaaS_RPC"` 添加移动网关组件依赖。



- 在命令行中执行 `pod install` 即可完成接入。

### 后续步骤

[使用 SDK](#)

## 5.2.2. 使用 SDK

RPC 相关模块为 `APMobileNetwork.framework`、`MPMgsAdapter`，推荐使用 `MPMgsAdapter` 中的接口。

本文引导您通过以下步骤使用移动网关 SDK：

- [初始化网关服务](#)
- [生成 RPC 代码](#)
- [发送请求](#)
- [请求自定义配置](#)
- [自定义 RPC 拦截器](#)
- [数据加密](#)
- [数据签名](#)

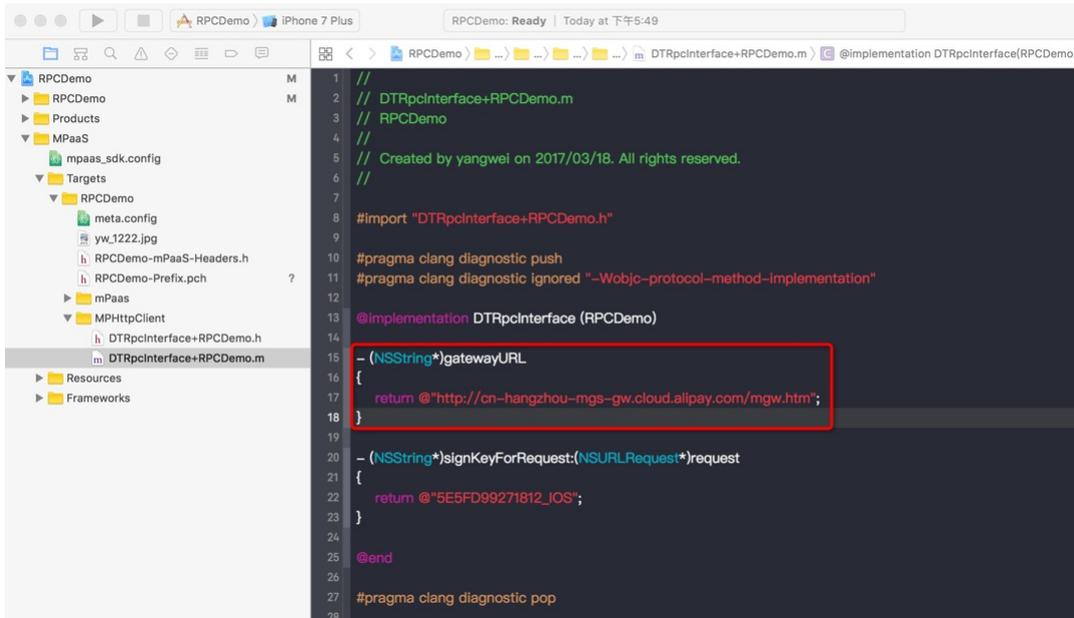
### 初始化网关服务

调用以下方法初始化网关服务：

```
[MPRpcInterface initRpc];
```

### 旧版本升级注意事项

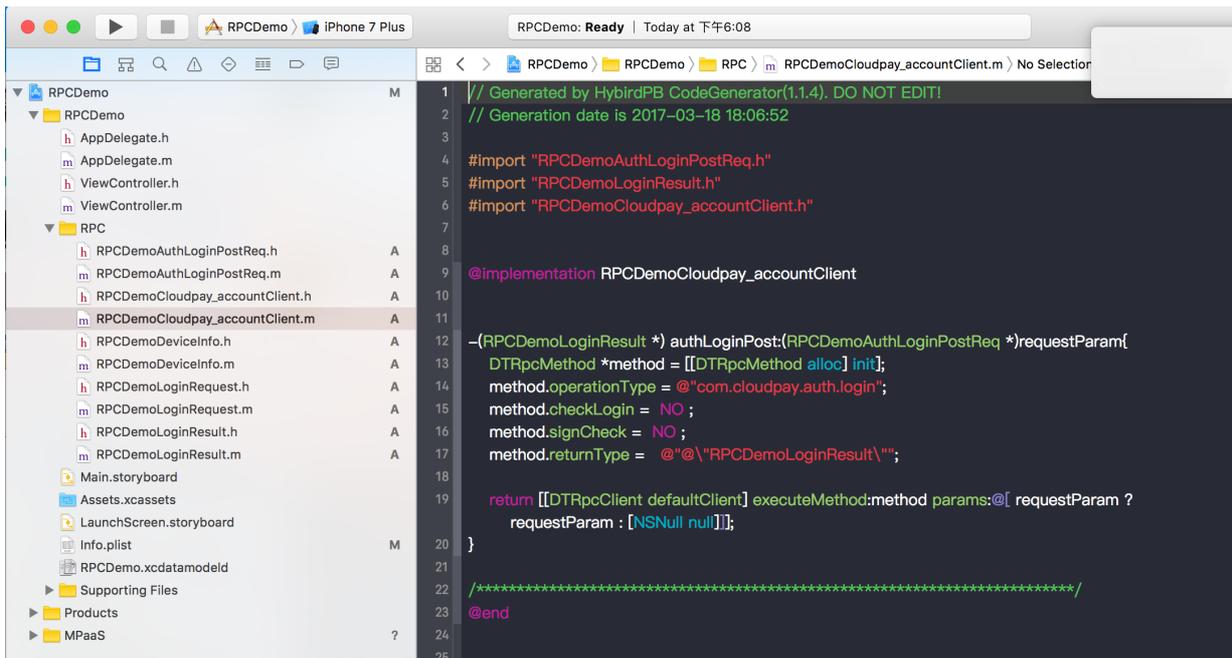
10.1.32 版本之后不再需要添加 `DTRpcInterface` 类的 `Category` 文件，中间层会实现包装从 `meta.config` 中读取，升级版本后请检查工程中是否存在旧版本配置，如果有请移除。下面为新版本应移除的 `DTRpcInterface` 类的 `Category` 文件。



### 生成 RPC 代码

当 App 在移动网关控制台接入后台服务后，即可下载客户端的 RPC 代码。更多信息请参考 [生成代码](#)。

下载的 RPC 代码结构如下：



其中：

- RPCDemoCloudpay\_accountClient 为 RPC 配置。
- RPCDemoAuthLoginPostReq 为 request 模型。
- RPCDemoLoginResult 为 response 模型。

### 发送请求

RPC 请求必须在子线程调用，可使用中间层中 `MFRpcInterface` 封装的子线程调用接口，回调方法默认为主线程。示例代码如下：

```
- (void)sendRpc
{
    __block RPCDemoLoginResult *result = nil;
    [MPRpcInterface callAsyncBlock:^(
        @try
        {
            RPCDemoLoginRequest *req = [[RPCDemoLoginRequest alloc] init];
            req.loginId = @"alipayAdmin";
            req.loginPassword = @"123456";
            RPCDemoAuthLoginPostReq *loginPostReq = [[RPCDemoAuthLoginPostReq alloc] init];
            loginPostReq._requestBody = req;
            RPCDemoCloudpay_accountClient *service = [[RPCDemoCloudpay_accountClient alloc] init];
            result = [service authLoginPost:loginPostReq];
        }
        @catch (NSEException *exception) {
            NSLog(@"%@", exception);
            NSError *error = [userInfo objectForKey:@"kDTRpcErrorCauseError"]; // 获取异常详细信息
            NSInteger code = error.code; // 获取异常详细信息错误码
        }
    ] completion:^(
        NSString *str = @"";
        if (result && result.success) {
            str = @"登录成功";
        } else {
            str = @"登录失败";
        }

        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:str message:nil delegate:nil
            cancelButtonTitle:nil otherButtonTitles:@"ok", nil];
        [alert show];
    )];
}
```

#### ② 说明

要使用 `try catch` 捕获异常；当网关异常时会抛出，根据 [网关结果码说明](#) 查询原因。

### 请求自定义配置

`DTRpcMethod` 为 RPC 请求方法描述，记录 RPC 请求的方法名、参数、返回类型等信息。

- 如果发送请求时，不需要加签，可以将 `DTRpcMethod` 的 `signCheck` 属性设置为 NO。

```
-(MPDemoUserInfo *) dataPostSetTimeout:(MPDemoPostPostReq *)requestParam
{
    DTRpcMethod *method = [[DTRpcMethod alloc] init];
    method.operationType = @"com.antcloud.request.post";
    method.checkLogin = NO ;
    method.signCheck = NO ;
    method.returnType = @"@"MPDemoUserInfo@"";

    return [[DTRpcClient defaultClient] executeMethod:method params:[ ]];
}
```

- 如果需要设置超时时间，可以配置 `DTRpcMethod` 的 `timeoutInterval` 属性。

```
-(MPDemoUserInfo *) dataPostSetTimeout:(MPDemoPostPostReq *)requestParam
{
    DTRpcMethod *method = [[DTRpcMethod alloc] init];
    method.operationType = @"com.antcloud.request.post";
    method.checkLogin = NO ;
    method.signCheck = YES ;
    method.timeoutInterval = 1; // 这个超时时间是客户端收到网关返回的时间，服务端配置的超时时间是后端业务系统的返回时间；默认 20s，设置小于 1 时无效即为默认值
    method.returnType = @"@"MPDemoUserInfo@"";

    return [[DTRpcClient defaultClient] executeMethod:method params:[ ]];
}
```

- 如果需要为接口添加 Header，可以使用下面 `DTRpcClient` 的扩展方法。

```
-(MPDemoUserInfo *) dataPostAddHeader:(MPDemoPostPostReq *)requestParam
{
    DTRpcMethod *method = [[DTRpcMethod alloc] init];
    method.operationType = @"com.antcloud.request.postAddHeader";
    method.checkLogin = NO ;
    method.signCheck = YES ;
    method.returnType = @"@"MPDemoUserInfo@"";

    // 针对接口添加 header
    NSDictionary *customHeader = @{@"testKey": @"testValue"};
    return [[DTRpcClient defaultClient] executeMethod:method params:[ ] requestHeaderField:customHeader responseHeaderFields:nil];
}
```

- 如果需要为所有接口添加 Header，可以参考下方拦截器的使用，采用拦截器的方式实现。具体实现方法请参考移动网关 [代码示例](#)。



## 前置条件

添加移动网关 SDK 前，请您确保已经将工程接入到 mPaaS。更多信息请参见开发指南手册中基于已有工程使用 ohpmrc 接入。

## 引入依赖

在项目的 `.ohpmrc` 中添加如下仓库：

```
@mpaas:registry=https://mpaas-ohpm.oss-cn-hangzhou.aliyuncs.com/meta
```

## 添加 SDK

在 `oh-package.json5` 中配置所需依赖，其中，当前版本号为 0.0.2。

```
{
  "license": "",
  "devDependencies": {},
  "author": "",
  "name": "entry",
  "description": "Please describe the basic information.",
  "main": "",
  "version": "1.0.0",
  "dependencies": {
    "@mpaas/rpc": "0.0.2"
  }
}
```

## 配置权限

在 `module.json5` 中配置所需权限。

```
"requestPermissions": [
  {
    "name": "ohos.permission.GET_NETWORK_INFO",
  },
  {
    "name": "ohos.permission.SET_NETWORK_INFO",
  },
  {
    "name": "ohos.permission.INTERNET",
  }
]
```

## 5.3.2. 使用 SDK

### 初始化框架

在使用 RPC 之前，需先初始化 mPaaS 框架：

```
export default class EntryAbilityStage extends AbilityStage {
  async onCreate() {
    const app = this.context;
    MPFramework.create(app);

    const instance: MPFramework = MPFramework.instance;
    const ctx: Context = instance.context
  }
}
```

### 生成 RPC 代码

以下代码由控制台自动生成，若尚未升级控制台，可以手动按以下格式输入模型，如：`VipInfo.ts`。

```
interface VipInfo {
  expireTime: number,
  level: number
}
```

模型中可以包含模型，如：`userInfo.ts`。

```
interface UserInfo {
  vip: VipInfo,
  name: string,
  age: number
}
```

包装参数由控制台自动生成，如：`LoginPostReq.ts`。

```
interface LoginPostReq {
  _mPaaSCustomBody: UserInfo
}
```

自动生成接口，如尚未升级控制台，可以先手动配置。以 `Client.ets` 为例。

```
import {MPRPC} from '@mpaas/rpc'

class Client{<>中为返回值类型, needsign, ispb为必传字段
  async loginPost(req:LoginPostReq):Promise<string>{<>返回类型为string
    return MPRpc.executeRpc<string>(this, {
      operationType:"com.antcloud.request.post",
      needSign:false,
      isPb:false
    }, req);
  }
}
```

## 调用 RPC 接口

RPC 请求必须在子线程调用, 可使用中间层中 `MPRPCInterface` 封装的子线程调用接口, 回调方法默认为主线程。示例代码如下:

```
//前提条件初始化rpc
MPRPC.init();
function testRpc(){
  let cl = new Client();
  let account:VipInfo = {
    expireTime:1,
    level:1
  };
  let user:UserInfo = {
    vip:account,
    name:"handsome",
    age:14
  }
  let req:LoginPostReq = {
    _mPaaSCustomBody:user
  }
  cl.loginPost(req).then((result)=>{
    console.log("result")
  }).catch((e:Error)=>{
    console.log(e.message) //通过e.message可以拿到具体的报错原因
  });
}
```

### ② 说明

要使用 `try catch` 捕获异常, 当网关异常时就会抛出, 您可以根据 [网关结果码说明](#) 查询原因。

## 请求自定义配置

### 设置超时时间

添加 `timeout` 配置, 代码如下:

```
import {MPRPC} from '@mpaas/rpc'

class Client{<>中为返回值类型, needsign, ispb为必传字段
  async loginPost(req:UserInfo):Promise<string>{
    return MPRpc.executeRpc<string>(this, {
      operationType:"com.antcloud.request.post",
      needSign:false,
      isPb:false,
      timeout:1000000//设置超时时间, 默认为1分钟
    }, req);
  }
}
```

### 设置请求 URL

添加 URL 配置, 代码如下:

```
import {MPRPC} from '@mpaas/rpc'

class Client{<>中为返回值类型, needsign, ispb为必传字段
  async loginPost(req:UserInfo):Promise<string>{
    return MPRpc.executeRpc<string>(this, {
      operationType:"com.antcloud.request.post",
      needSign:false,
      isPb:false,
      url:"https://xxxx.xx/xx/"//设置请求 URL。默认为网关地址
    }, req);
  }
}
```

### 设置请求 Header

添加 Header 配置, 代码如下:

```
import {MPRPC} from '@mpaas/rpc'

//初始化header
let headers:Map<string,string> = new Map();
headers.set("key1","value1");
headers.set("key2","value2");

class Client{<!--中为返回值类型, needsign, ispb为必传字段
async loginPost(req:UserInfo):Promise<string>{
  return MPRPC.executeRpc<string>(this,{
    operationType:"com.antcloud.request.post",
    needSign:false,
    isPb:false,
    header:headers //设置headers
  },req);
}
```

## 自定义 RPC 拦截器

### 1. 初始化拦截器。

- `preHandle` 在请求前执行，参数返回 `RpcInvokeContext` 对象，可以进行 Header，URL，timeout 等操作，`return false` 表示进行拦截，rpc 接口可以 catch 异常，请求将终止。
- `postHandle` 在请求后执行，参数返回 `RpcInvokeContext` 对象，可以获取 `response header` 等操作，`return false` 表示进行拦截，rpc 接口可以 catch 异常，请求将终止。
- `handleException`，表示遇到异常时进行拦截。

```
import {RpcInterceptor,RpcInvokeContext} from "@mpaas/rpc";

class interceptor implements RpcInterceptor{
  preHandle(rpcInvokeContext: RpcInvokeContext): boolean {
    return true;
  }
  postHandle(rpcInvokeContext: RpcInvokeContext): boolean {
    return true;
  }
  handleException(rpcInvokeContext: RpcInvokeContext, exception: RPCException)
  {
    throw exception;
  }
}
```

### 2. 配置拦截器。

- 拦截指定 RPC。

在自动生成的代码中添加 `interceptor` 配置：

```
import {MPRPC} from '@mpaas/rpc'
import {RpcInterceptor,RpcInvokeContext} from "@mpaas/rpc";

class interceptor implements RpcInterceptor{
  preHandle(rpcInvokeContext: RpcInvokeContext): boolean {
    return true;
  }
  postHandle(rpcInvokeContext: RpcInvokeContext): boolean {
    return true;
  }
  handleException(rpcInvokeContext: RpcInvokeContext, exception: RPCException)
  {
    throw exception;
  }
}

class Client{<!--中为返回值类型, needsign, ispb为必传字段
async loginPost(req:UserInfo):Promise<string>{
  return MPRPC.executeRpc<string>(this,{
    operationType:"com.antcloud.request.post",
    needSign:false,
    isPb:false,
    interceptor:new interceptor()
  },req);
}
```

- 设置全局拦截。

在 `MPRPC` 接口中设置：

```
import {RpcInterceptor,RpcInvokeContext} from "@mpaas/rpc";
import {MPRpc} from '@mpaas/rpc'

class interceptor implements RpcInterceptor{
  preHandle(rpcInvokeContext: RpcInvokeContext): boolean {
    return true;
  }
  postHandle(rpcInvokeContext: RpcInvokeContext): boolean {
    return true;
  }
  handleException(rpcInvokeContext: RpcInvokeContext, exception: RPCException)
  {
    throw exception;
  }
}

MPRpc.addGlobalInterceptor(new interceptor());
```

## 数据加密

在 `rawfile` 目录下添加 `mpaasnetconfig.json` 文件。其中：

- `type`：加密类型，支持 ECC/RSA/SM2。
- `crypt`：是否开启加密。
- `gw`：使用蚂蚁自研 gzip，必须设置为 `true`。
- `pubKey`：公钥，需与加密类型匹配。
- `gwList`：支持加密的 URL，多个 URL 可以用 `,` 分割，只有命中 URL 的才会参与加密。

```
{
  "type": "ECC",//支持ECC/RSA/SM2
  "crypt": false,//是否开启加密
  "gw": true,//使用蚂蚁自研 gzip
  "pubKey": "-----BEGIN PUBLIC KEY-----
\nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEz796auKwF42leWWX/cwvfrvz/M2\nd6f4ovv4G7wmu45Ed+5WBDfp7vKHB0P3il4SxmvK6be6m1MhL2kkY8Kj0Q==\n-----END PUBLIC KEY-----",
  "gwList": "https://mgw.mpaas.cn-hangzhou.aliyuncs.com/mgw.htm,http://11.164.247.80/mgw.htm"
}
```

## 数据签名（临时方案）

在接口文件中进行配置，`needSign` 设置为 `true`，`sec` 中设置 `appsecret`，`signType` 中设置签名类型，目前支持：MD5 (type 为 0)，SHA256 (type 为 4)，SHA1 (type 为 1)。

```
import {MPRpc} from '@mpaas/rpc'

class Client{<<<中为返回值类型，needsign, ispb为必传字段
  async loginPost(req:UserInfo):Promise<string>{
    return MPRpc.executeRpc<string>(this,{
      operationType:"com.antcloud.request.post",
      needSign:true, (默认为false)
      isPb:false,
      sec:"sss",
      signType:0 //如果不填，默认为0 md5 (默认不填为md5)
    },req);
  }
}
```

## 数据签名（安全图片）

在接口文件中进行配置，`needSign` 设置为 `true`，`signType` 中设置签名类型，目前支持：MD5 (type 为 0)，SHA256 (type 为 4)，SM3 (type 为 5)。

### 生成安全图片

当前版本尚未提供自动化工具，需客户提供相关信息，由 mPaaS 方帮助生成。需要提供的信息包括 `appId`、`workspaceId`、应用包名、`appsecret` 和 应用签名 `fingerPrint`。

其中 `fingerPrint` 可以通过鸿蒙官方接口获取，建议接入方使用统一签名，否则针对不同签名需要生成不同的图片。

获取 `fingerPrint` 的接口如下：

```
import bundleManager from '@ohos.bundle.bundleManager';

let info = bundleManager.getBundleInfoForSelfSync(bundleManager.BundleFlag.GET_BUNDLE_INFO_WITH_SIGNATURE_INFO);
let finger = info.signatureInfo.fingerprint;
```

获得安全图片后，将其放在 `rawfile` 目录下即可。

### 针对全局

- 可以设置全局 `secret`。

```
MPFramework.instance.appSecret = "xxxxxxx"
```

- 可以利用全局拦截器进行设置。

```
import {RpcInterceptor,RpcInvokeContext} from "@mpaas/rpc";
import {MPRpc} from '@mpaas/rpc'

class interceptor implements RpcInterceptor{
  preHandle(rpcInvokeContext: RpcInvokeContext): boolean {
    rpcInvokeContext.setNeedSign(true);
    rpcInvokeContext.setSignType(0);
    rpcInvokeContext.setSecret("xxxx");
    return true;
  }
  postHandle(rpcInvokeContext: RpcInvokeContext): boolean {
    return true;
  }
  handleException(rpcInvokeContext: RpcInvokeContext, exception: RPCException)
  {
    throw exception;
  }
}

MPRpc.addGlobalInterceptor(new interceptor());
```

## 5.4. H5 JS 编程

目前，很多移动 App 前端都采用了 JavaScript (JS) 语言进行编码。mPaaS 也提供了移动端 Web 解决方案 —— [H5 容器简介](#)。H5 承载于 Android 和 iOS 之上，需要进行客户端接入。

在客户端接入 H5 容器后，前端可以很方便地使用网关：

- 通过动态代理的方式，封装客户端和服务端之间的通讯。
- 如果服务端和客户端定义了一致的接口，可由服务端自动生成代码并导出给客户端使用。
- 对 `RpcException` 进行统一的异常处理，弹对话框、Toast 消息框等。

### 前置条件

进行 H5 JS 编程之前，需确保 Android/iOS 客户端已经接入 H5 容器。客户端接入方法参见 [接入 Android](#) 和 [接入 iOS](#)。

### 生成 JS 代码

在移动网关控制台接入 App 后台服务后，即可通过控制台生成 RPC 的 JS SDK 供客户端调用，详细说明参见 [生成代码](#)。

客户端代码生成 ×

\* API 分组:

Platform:  Android  iOS  JS

PackageName:

目前针对每个 API，根据约定的接口参数，都会生成如下模板代码：

```
var params = [{
  "_requestBody":{"userName":"","userId":0}
}]
var operationType = 'alipay.mobile.ic.dispatch'

AlipayJSBridge.call('rpc', {
  operationType: operationType,
  requestData: params,
  headers:{}
}, function (result) {
  console.log(result);
});
```

前端需要使用到 RPC 时，会直接使用上面的模板，填入调用的请求参数。

### 调用 RPC 接口

JS 调用 RPC 的方法如下：

```
AlipayJSBridge.call('rpc', {
  operationType: 'alipay.client.xxxx',
  requestData: [],
  headers:{}
}, function (result) {
  console.log(result);
});
```

### 参数说明

参数	类型	是否必需	默认值	描述
operationType	string	Y		RPC 服务名称
requestData	array	Y		RPC 请求的参数，需要开发者根据具体 RPC 接口进行构造。
headers	dictionary	Y	{}	RPC 请求设置的 headers。
gateway	string	Y	alipay 网关	网关地址
compress	boolean	Y	true	是否支持 request gzip 压缩。
disableLimitView	boolean	Y	false	RPC 网关被限流时是否禁止自动弹出统一限流弹窗。

**请求结果**

结果	类型	描述
result	dictionary	RPC 响应的结果。非字典结构的字符串值会被放入一个字典结构，key 为 <code>resData</code> 。

**错误码**

错误码	描述
10	网络错误。
11	请求超时。
其他	由 mobilegw 网关定义。

## 6. 接入服务端

### 6.1. 后端签名校验说明

移动网关提供服务端 HTTP 服务签名验证功能，提高从网关到服务端的数据安全性。

- 在网关控制台开启某一 API 分组的签名校验后，移动网关会对该分组里面的每一个 API 请求创建签名信息，签名使用的公私钥可在网关控制台创建。
- 服务端读取签名字符串后，对收到的请求进行本地签名计算，比对与收到的签名是否一致，以此来判断请求是否合法。

#### 读取签名

移动网关计算的签名保存在 Request 的 Header 中，Header Key 为 `X-Mgs-Proxy-Signature`。

API 分组中配置的密钥 Key 可用于区分和获取不同的密钥值对应的 Key，Header Key 为 `X-Mgs-Proxy-Signature-Secret-Key`。

#### 验签方法

##### 组织加签数据

```
String stringToSign =
    HTTPMethod + "\n" +
    Content-MD5 + "\n" +
    Url
```

- `HTTPMethod`：全大写的 HTTPMethod，如 `PUT` 或 `POST` 等。
- `Content-MD5`：请求 Body 的 MD5 值。计算方法如下：
  - 若 `HTTPMethod` 不是 `PUT` 或 `POST` 之一，则 MD5 为空字符串 `""`；否则执行第二步。
  - 若请求有 Body 且 Body 为 Form 表单，则 MD5 为空字符串 `""`；否则执行第三步。
  - 使用以下方式计算 MD5。其中，当请求无 Body 时，`bodyStream` 为字符串 `"null"`。

```
String content-MD5 = Base64.encodeBase64(MD5(bodyStream.getBytes("UTF-8")));
```

#### ① 重要

即使 `content-MD5` 为空字符串 `""`，加签方法中 `content-MD5` 后面的换行符 `"\n"` 也不能省略，即此时签名中会有连续两个 `"\n"`。

- `Url`：由 Path、Query 以及 Body 中的 Form 参数组装而成。假设请求格式为 `http://ip:port/test/testSign?c=3&a=1` 且 Form 中的参数为 `b=2&d=4`，组装步骤如下：
  - 获取 Path：Path 是 `ip:port` 之后、`?` 之前的部分。此例中即为 `/test/testSign`。
  - 若请求 Query 和 Form 参数均为空，则 `Url` 即为 Path；否则进行下一步。
  - 拼接参数。将 Query 和 Form 中的参数根据 Key 按照字典序排序，然后拼接为 `Key1=Value1&Key2=Value2&...&KeyN=ValueN`。此例中即为 `a=1&b=2&c=3&d=4`。

#### ② 说明

Query 或 Form 参数的 Value 可能有多个，只取第一个 `Value` 即可。

- 拼接 URL。URL 为 `Path?Key1=Value1&Key2=Value2&...&KeyN=ValueN`。此例中即为 `/test/testSign?a=1&b=2&c=3&d=4`。

#### 验证签名

- 采用 MD5 算法验签

```
String sign = "xxxxxxx"; //移动网关传过来的签名
String salt = "xxx"; //MD5 Salt

MessageDigest digest = MessageDigest.getInstance("MD5");
String toSignedContent = stringToSign + salt;
byte[] content = digest.digest(toSignedContent.getBytes("UTF-8"));
String computedSign = new String(Hex.encodeHexString(content));

boolean isSignLegal = sign.equals(computedSign) ? true : false;
```

- 采用 RSA 算法验签

```
String sign = "xxxxxxx"; //移动网关传过来的签名
String publicKey = "xxx"; //移动网关的 RSA 公钥

PublicKey pubKey = KeyReader.getPublicKeyFromX509("RSA", new ByteArrayInputStream(publicKey.getBytes()));
java.security.Signature signature = java.security.Signature.getInstance("SHA1WithRSA");
signature.initVerify(pubKey);
signature.update(stringToSign.getBytes("UTF-8"));

boolean isSignLegal = signature.verify(Base64.decodeBase64(sign.getBytes("UTF-8")));
```

- 采用国密算法验签

SM3 算法：

```
String sign = xxxx;
String salt = xxxxxx;
String toSignedContent = stringToSign + salt;
byte[] srcData = toSignedContent.getBytes("UTF-8");
SM3Digest digest = new SM3Digest();
digest.update(srcData, 0, srcData.length);
byte[] resultHash = new byte[digest.getDigestSize()];
digest.doFinal(resultHash, 0);
String computedSign = Hex.encodeHexString(resultHash);
boolean isSignLegal = sign.equals(computedSign);
```

#### SM2 算法：

```
String sign = xxxx;
String pubKey = xxxxx;
```

```
Signature signature = Signature.getInstance("SM3withSM2", "LOCCSBC");
KeyPair keyPair = getSmKeyPair(pubKey);
```

```
PublicKey publicKey = keyPair.getPublic();
signature.initVerify(publicKey);
signature.update(stringToSign.getBytes("UTF-8"));
return signature.verify(Hex.decodeHex(sign));
```

```
public static KeyPair getSmKeyPair(String keyPairContent) throws IOException, InvalidKeySpecException, NoSuchProviderException,
NoSuchAlgorithmException {
    KeyFactory keyFactory = KeyFactory.getInstance("SM2", "LOCCSBC");
```

```
byte[] keyBytes = getPKCS1fromPEMString(keyPairContent);
SM2ParameterSpec spec = SM2NamedCurveTable.getParameterSpec("sm2p256v1", "1234567812345678");
```

```
SM2PrivateKeySpec sm2PrivateKeySpec = new SM2PrivateKeySpec(BigInteger.fromUnsignedByteArray(keyBytes), spec);
SM2PrivateKey prvkey = (SM2PrivateKey) keyFactory.generatePrivate(sm2PrivateKeySpec);
```

```
SM2PublicKeySpec sm2PublicKeySpec = new SM2PublicKeySpec(prvkey.getQ(), spec);
SM2PublicKey pubKey = (SM2PublicKey) keyFactory.generatePublic(sm2PublicKeySpec);
```

```
return new KeyPair(pubKey, prvkey);
}
```

```
public static byte[] getPKCS1fromPEMString(String pemStr) {
    // pkcs1
    try {
        PEMParser reader = new PEMParser(new StringReader(pemStr));
        PemObject pemObject = reader.readPemObject();
        reader.close();

        ASN1InputStream asn1In = new ASN1InputStream(pemObject.getContent());
        ASN1Sequence derSequence = (ASN1Sequence) asn1In.readObject();
        asn1In.close();

        // openssl format
        Object obj = derSequence.getObjectAt(1);
        if (obj instanceof ASN1OctetString) {
            return ((ASN1OctetString) obj).getOctets();
        }

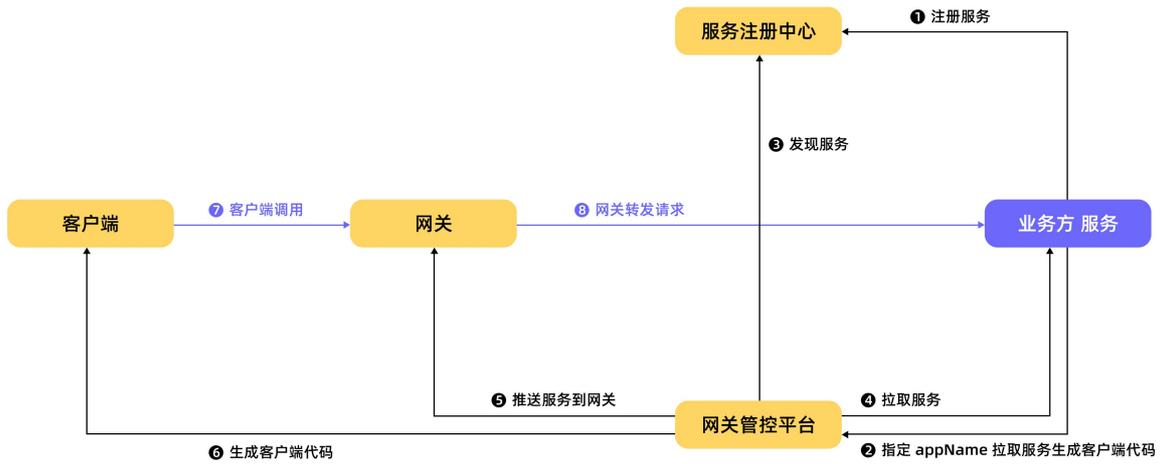
        // gmssl format
        DEROctetString derOctetString = (DEROctetString) derSequence.getObjectAt(2);
        ASN1Sequence sequence = (ASN1Sequence) ASN1Sequence.fromByteArray(derOctetString.getOctets());
        return ((ASN1OctetString) sequence.getObjectAt(1)).getOctets();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

#### 代码示例

更多详情，请参考 [HttpSignUtil.java](#)。

## 6.2. 服务定义与开发

此文档仅针对集成了网关 SPI 的系统，如对外暴露 mpaaschannel 或 Dubbo 类型 API 服务的业务系统。对于使用 HTTP API 的业务系统无需查看此文档。



### 引入网关二方包

在项目的主 `pom.xml` 文件中引入如下二方包（若原工程已经有依赖，请忽略）。  
基础依赖都需要引用，请根据实际需要集成的 API 类型，选择 Dubbo 或 TR 依赖。

② 说明

引入依赖前，请确认您完成了 Maven 的配置。详情参见 [配置 Maven](#)。

### 基础依赖

```

<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-spi-adapter</artifactId>
  <version>1.0.5.20201010</version>
</dependency>
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-log</artifactId>
  <version>1.0.5.20201010</version>
</dependency>
<dependency>
  <groupId>com.alipay.hybirdpb</groupId>
  <artifactId>classparser</artifactId>
  <version>1.2.2</version>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.5</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.72_noneautotype</version>
</dependency>
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-common</artifactId>
  <version>1.0.5.20201010</version>
</dependency>
    
```

### Dubbo 依赖

```

<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-spi-dubbo</artifactId>
  <version>1.0.5.20201010</version>
</dependency>
<dependency>
  <groupId>org.apache.dubbo</groupId>
  <artifactId>dubbo</artifactId>
  <version>2.7.8</version>
</dependency>
    
```

② 说明

Dubbo 请使用原生版本，不要使用 dubbox（不兼容）。

## HRPC 依赖

```
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-spi-hrpc</artifactId>
  <version>1.0.5.20201010</version>
</dependency>
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-registry</artifactId>
  <version>1.0.5.20201010</version>
</dependency>
<dependency>
  <groupId>hessian</groupId>
  <artifactId>hessian</artifactId>
  <version>3.3.6</version>
</dependency>
```

## TR 依赖

```
<dependency>
  <groupId>com.alipay.gateway</groupId>
  <artifactId>mobilegw-unify-spi-sofa</artifactId>
  <version>1.0.5.20201010</version>
</dependency>
```

## 定义服务接口并实现

按照业务需求，定义服务接口：`com.alipay.xxxx.MockRpc`；并提供该接口的实现 `com.alipay.xxxx.MockRpcImpl`。

### ② 说明

- 方法定义中的入参尽量定义为 VO，后期添加参数，就可以在 VO 中添加参数，而不改变方法的声明格式。
- 服务接口定义的相关规范，请参见 [业务接口定义规范](#)。

## 定义 operationType

在服务接口的方法上添加 `@OperationType` 注解，定义发布服务的接口名称。`@OperationType` 有三个参数成员：

- `value`：RPC 服务的唯一标识，定义规则为 `组织.产品域.产品.子产品.操作`。
- `name`：接口中文名称。
- `desc`：接口描述。

### ② 说明

- `value` 在网关为全局唯一，尽量定义详细，否则可能会和其他业务方的 `value` 值一样，导致无法注册服务。
- 为便于维护，请务必填写完整 `@OperationType` 的三个字段。

样例：

```
public interface MockRpc {

    @OperationType("com.alipay.mock")
    Resp mock(Req s);

    @OperationType("com.alipay.mock2")
    String mock2(String s);
}

public static class Resp {
    private String msg;
    private int code;

    // ignore getter & setter
}

public static class Req {
    private String name;
    private int age;

    // ignore getter & setter
}
```

然后，通过网关提供的 SPI 包，将定义好的 API 服务注册到指定的注册中心。

## 注册 Dubbo API 服务

注册 Dubbo API 服务需要如下参数：

- `registryUrl`：该值为注册中心的地址。
- `appName`：该值为业务方的应用名，与 API 分组名相同。

## spring 方式

- 在对应 bundle 的 spring 配置文件中，声明定义好的服务的 spring bean：

```
<bean id="mockRpc" class="com.alipay.gateway.spi.mpc.test.MockRpcImpl"/>
```

2. 在对应 bundle 的 spring 配置文件中，声明暴露服务的 starter bean — `DubboServiceStarter`，该接口会将所有带有 `OperationType` 的 bean 通过 Dubbo 协议注册到指定的注册中心。

```
<bean id="dubboServiceStarter" class="com.alipay.gateway.spi.dubbo.DubboServiceStarter">
  <property name="registryUrl" value="${registry_url}"/>
  <property name="appName" value="${app_name}"/>
</bean>
```

### spring-boot 方式

spring-boot 和 spring 本质上一样，只是注册的方式改为注解的方式，不用配置 xml 文件。

1. 通过注解的方式，将定义的服务注册成 bean：

```
@Service
public class MockRpcImpl implements MockRpc{
}
```

2. 以注解的方式，定义暴露服务的 starter：

```
@Configuration
public class DubboDemo {
  @Bean(name="dubboServiceStarter")
  public DubboServiceStarter dubboServiceStarter(){
    DubboServiceStarter dubboServiceStarter = new DubboServiceStarter();
    dubboServiceStarter.setAppName("${app_name}");
    dubboServiceStarter.setRegistryUrl("${registry_url}");
    return dubboServiceStarter;
  }
}
```

### 注册 TR API 服务

注册 TR API 服务需要 `appName` 这一个参数，该参数值为业务方的应用名，必填。

### spring 方式

1. 在对应 bundle 的 spring 配置文件中，声明定义好的服务的 spring bean：

```
<bean id="mockRpc" class="com.alipay.gateway.spi.mpc.test.MockRpcImpl"/>
```

2. 在对应 bundle 的 spring 配置文件中，声明暴露服务的 starter bean。

接口 `SofaServiceStarter` 会将所有带有 `@OperationType` 的 bean 通过 TR 协议暴露给网关调用。

```
<bean id="SofaServiceStarter" class="com.alipay.gateway.spi.sofa.SofaServiceStarter">
  <property name="appName" value="${app_name}"/>
</bean>
```

### spring-boot 方式

spring-boot 和 spring 本质上一样，只是注册的方式改为注解的方式，不用配置 xml 文件。

1. 通过注解的方式，将定义的服务注册成 bean：

```
@Service
public class MockRpcImpl implements MockRpc{
}
```

2. 以注解的方式，定义暴露服务的 starter：

```
@Configuration
public class TRDemo {
  @Bean(name="sofaServiceStarter")
  public SofaServiceStarter sofaServiceStarter(){
    SofaServiceStarter sofaServiceStarter = new SofaServiceStarter();
    sofaServiceStarter.setAppName("${app_name}");
    return sofaServiceStarter;
  }
}
```

### 结果

完成上述步骤后，就可以通过在网关进行一系列的操作，将定义的 API 服务对客户端进行暴露。具体请参见 [注册 API](#)。

## 6.3. 网关辅助类使用说明

本文对网关中用到的相关辅助类，包括拦截器类、MobileRpcHolder，以及网关错误码的使用进行说明。

### 实现拦截器功能

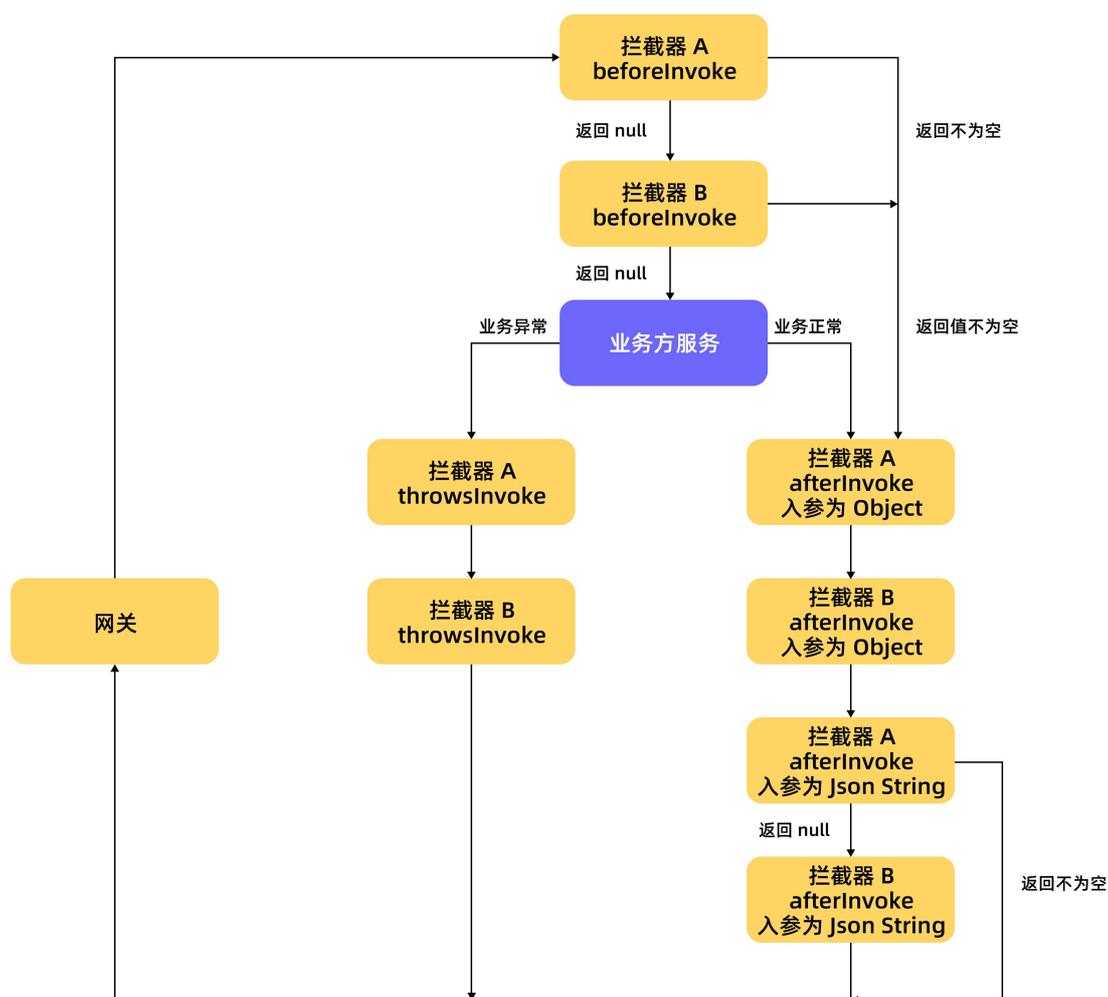
拦截器只适用于非 HTTP 类型服务。

`mobilegw-unify-spi-adapter.jar` 实际上是通过 Java 的反射调用业务方法，即 `OperationType` 所指定的方法。在方法调用的过程中，业务方可以实现 SPI 包中定义的拦截器，从而实现扩展。

网关的 SPI 包定义了两个拦截器：`AbstractMobileServiceInterceptor` 抽象类和 `MobileServiceInterceptor` 接口。

### AbstractMobileServiceInterceptor

`MobileServiceInterceptor` 主要提供了四个方法，分别是 `beforeInvoke`、`afterInvoke`（分为两种：一种入参为业务返回的 Object；另一种入参为 Object 转成的 JSON string）、`throwsInvoke` 和 `getOrder`。



如上图所示，拦截器主要在以下三种情况下进行拦截：

- 方法调用前：即 `beforeInvoke` 方法，该方法有返回值。一旦该方法的返回值不为空，那么网关认定拦截成功，将会跳过剩余拦截器的 `beforeInvoke` 方法，同时跳过调用业务方的方法，直接进入拦截器的 `afterInvoke` 方法。
- 方法调用后：即 `afterInvoke` 方法。`afterInvoke` 有两种，一种入参为 `Object`，即业务方返回的对象，该方法没有返回值，所有的拦截器的该方法都会执行；另一种入参为 `Object` 转成的 `JSON string`，该方法可以改变传入的 `JSON` 数据并返回。一旦返回值不为空，那么网关认定拦截成功，后续的拦截器将被忽略。
- 方法出现异常：即 `throwsInvoke` 方法。该方法没有返回值，所有拦截器的该方法都会被执行。在业务方出现异常时会被调用。

### MobileServiceInterceptor

`MobileServiceInterceptor` 继承了框架的 `Ordered` 接口，因此，业务方实现的拦截器还可以通过实现 `getOrder` 方法指定执行顺序，设置的数值越小，执行的优先级越高；设置的数值越大，执行的优先级越低。

#### 使用示例

1. 编码自己的拦截器类，继承 `AbstractMobileServiceInterceptor` 类，或者实现 `MobileServiceInterceptor` 接口。

```
public class MyInterceptor implements MobileServiceInterceptor {

    /*
    参数说明
    method：即业务方的方法（@OperationType 定义的方法）
    args： 一个对象数组，即业务方方法的传入参数，传入参数个数即等于数组大小。
        使用时业务方根据需要进行类型转换。
    bean： 即业务方的接口实例。
    返回值说明：
    Object：可以在拦截器中返回数据，一旦返回值不为空，则网关认为已被拦截，就不会再调用业务方法
        同时，直接跳过其他拦截器的 beforeInvoke 方法，执行拦截器中的 afterInvoke 方法。
    */

    @Override
    public Object beforeInvoke(Method method, Object[] args, Object target) {
        //Do Something
        return null;
    }

    /*
    *参数说明
    *returnValue：业务方法返回的对象
    * 其它参数同上
    */
    @Override
    public void afterInvoke(Object returnValue, Method method, Object[] args, Object target) {
        //注意：这里入参是业务方返回的 Object
    }

    @Override
    public String afterInvoke(String returnJsonValue, Method method, Object[] args, Object target) {
        //注意：这里入参是由业务方返回的 object，转换而成的 JSON 格式的 string
        //可以返回新的 JSON 数据
        return null;
    }

    @Override
    public void throwsInvoke(Throwable t, Method method, Object[] args, Object target) {
    }

    @Override
    public int getOrder() {
        //最高级（数值最小）和最低级（数值最大）。
        return 0;
    }
}
```

2. 发布实现的类 `MyInterceptor`，成为 Bean。

- Spring Boot：直接在该类加注解 `@service`。

```
@service
public class MyInterceptor implements MobileServiceInterceptor{
```

- Spring：在配置的 `xml` 文件声明。

```
<bean id="myInterceptor" class="com.xxx.xxx.MyInterceptor"/>
```

### MobileRpcHolder 辅助类

`MobileRpcHolder` 是 `mobilegw-unify-spi-adapter.jar` 中提供的一个静态辅助类，该类定义了一个请求过程中的相关信息，最主要的定义如下：

<code>Map&lt;String, String&gt; session</code>	保存请求的 <code>session</code>
<code>Map&lt;String, String&gt; header</code>	保存请求的头部相关信息
<code>Map&lt;String, String&gt; context</code>	保存网关调用的上下文信息
<code>String operationType</code>	保存此次请求的 <code>operationType</code>

在业务方的服务（即 `OperationType`）被调用之前，SPI 服务会根据网关转发的请求 `MobileRpcRequest` 去设置 `MobileRpcHolder` 中的信息。在调用业务方服务之后这些信息会被清除。

`MobileRpcHolder` 的生命周期为整个服务的调用过程，调用后清除。

业务方也可以根据需要进行设置这些信息，这些信息会在调用业务的服务过程中一直存在，在调用过程中业务服务可以获取这些信息。具体的设置可以通过拦截器，在方法调用前后动态地修改 `MobileRpcHolder` 中保存的信息。

以下通过例子说明 `MobileRpcHolder` 如何使用。

#### 使用示例

这里以修改和获取 `session` 为例。

1. 修改 `session`。创建拦截器，具体过程参考上面的拦截器示例。下面以在方法调用前拦截为例：

```
@Override
public Object beforeInvoke(Method method, Object[] args, Object target) {
    Map<String, String> session = MobileRpcHolder.getSession();
    session.put("key_test", "value_test");
    MobileRpcHolder.setSession(session);
}
```

这样就能修改 `MobileRpcHolder` 中的 session 信息。

2. 获取 session。业务方在自己定义的服务中可以获取 session 信息。

```
@OperationType("com.alipay.account.query")
public String mock2(String s) {
    Map<String, String> session = MobileRpcHolder.getSession();
}
```

其他信息（如 header、context）的修改和获取方式同上。

```
// 获取 header 所有信息
Map<String,String> headers = MobileRpcHolder.getHeaders();
// 这里上下文信息指的是请求中的上下文信息
Map<String,String> context = MobileRpcHolder.getRequestCtx();
// 获取 OperationType
String opt = MobileRpcHolder.getOperationType();
```

## 使用网关错误码

移动网关有自己的一套错误码规范，详见 [网关结果码说明](#)。

需要注意 `BizException 6666`，此错误是业务出现异常后，网关会抛出的异常。

如果想要在出现具体错误时，返回其他错误码，可以通过抛出 `RpcException(ResultEnum resultCode)` 来控制 RPC 层的错误，比如 `resultCode=1001`，会返回给客户端“没有权限访问”。

## 代码示例

```
@Override
public String mock2(String s) throws RpcException {
    try{
        test();
    }catch (Exception e){
        throw new RpcException(IllegalArgumentException);
    }
    return "11111111";
}
```

## 自定义错误码

如果想使用自定义错误码，那么在调用业务方法时不能往外抛异常。

业务方法只要出现异常，就会返回状态码 6666。同时客户端收到该状态码，即认为服务出错，不会去解析业务返回的数据。客户端只有在接收到 1000 状态码时，才会去解析返回的数据。

具体做法是，服务端和客户端约定好具体的错误码，然后在调用业务方法时捕获（catch）所有的异常，将自定义错误码放在返回的数据中。这样即使业务出现异常，网关也会返回 1000 成功。同时客户端去解析返回的数据，提取自定义错误码。

## 7. 使用控制台

### 7.1. 使用须知

在专有云环境中进行控制台操作时，请勿对“MPAAS内置应用”下的任何 MGS 配置进行修改，否则会导致 mPaaS 组件的 RPC 无法正常访问。MPAAS 内置应用用于进行组件 RPC 配置。

### 7.2. 路由规则

移动网关支持动态路由功能。网关在接收到请求后，会通过路由规则将请求路由到后端系统。本文介绍如何创建路由规则，并对创建的路由规则进行管理，包括查看、修改、删除规则。

#### 创建路由规则

登录 mPaaS 控制台，选择目标应用后，完成以下步骤，创建路由规则：

1. 从左侧导航栏进入 **后台服务管理** > **移动网关** > **路由规则** 页面。
2. 单击 **创建路由规则** 按钮，在规则创建面板中，完成路由规则配置：
  - **规则名称**：用于标识路由规则。
  - **后端协议类型**：目前仅支持 HTTP 协议类型。
  - **路由方式**：目前支持权重路由和根据 header 路由两种路由方式。
  - **规则详情**：当前路由规则的规则详情，可单击 + 添加 来设置多条规则。最多可添加 100 条规则。
    - **权重路由规则详情**
      - **流量占比 (%)**：分配对应 API 分组的流量权重。输入 0 ~ 100 之间的整数，数字越大，权重越高。

#### ② 说明

当前路由规则下，所有 API 分组的流量比例之和必须为 100%。

- **服务地址**：输入对应的服务地址。

#### ▪ 根据 header 路由规则详情

- **参数名**：header 的 key。
- **匹配方式**：可选完全匹配或 IN 匹配。
  - 完全匹配为精准匹配，header 的 value 值完全匹配上，才可转发到对应后端服务地址。

#### ② 说明

完全匹配只能匹配一个参数值。

- IN 匹配只要 header 值在该参考列表内，都可转发到对应的后端服务地址。

#### ② 说明

IN 匹配支持输入多个值，输入单个值后需要使用 ; 分隔。

- **参数值**：header 的 value。
  - **服务地址**：匹配后的目标服务地址。
- **路由状态探活**：支持心跳检测，用户可手动选择是否开启，一旦开启后即可动态监测服务端路由状态。

#### ① 重要

该状态默认为关闭，关闭后 **路径 path** 和 **超时时间** 不显示。

- **路径 path**：路由状态探活的路径。
  - **超时时间**：路由状态探活的超时时间。
3. 单击 **确定**，即可完成路由规则的创建。新建的路由规则将展示在列表中。

#### 修改路由规则

完成以下步骤，修改路由规则：

1. 在路由规则列表中，单击路由规则右侧 **操作** 列中的 **编辑** 打开路由规则编辑界面。
2. 打开的 **编辑路由规则** 界面与创建路由规则界面内容一致，您可修改 **规则名称**、**路由方式** 以及 **规则详情**。

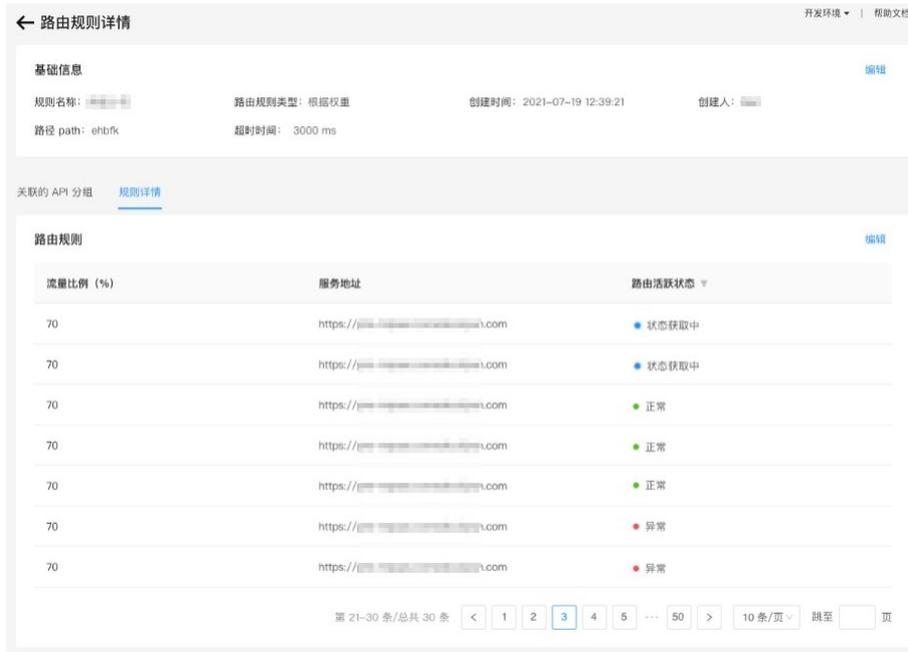
#### ② 说明

后端协议类型一旦创建则无法更改。

3. 单击 **确定**，即可完成路由规则的修改。

#### 查看路由规则详情

在路由规则列表页，单击 **路由规则名称** 可进入路由规则详情页。路由规则详情分为基础信息、关联的 API 分组和规则详情三部分。



### 基础信息

展示当前路由规则的名称、路由方式、创建时间、创建人信息、路径 path 以及超时时间。

### 关联的 API 分组

在路由规则详情页的 **关联的 API 分组** 标签页中，可查看当前路由规则已绑定的 API 分组。

在绑定的 API 分组列表中，单击 API 分组名称即可跳转至该 API 分组的详情页，您可查看该 API 分组的 API 信息、验签配置、超时时间等信息。

### 规则详情

在路由规则详情页的 **规则详情** 标签页中，可查看当前路由规则下的所有规则信息。

- 权重路由规则详情包括各个服务地址、对应的流量权重以及路由活跃状态。
- 根据 header 路由规则详情包括各个服务地址、参数名称、匹配方式、参数值以及路由活跃状态。

### 删除路由规则

#### 🔔 重要

如果有 API 分组引用了当前路由规则，则该路由规则无法删除。需先将 API 分组与路由规则解绑后，然后才能删除。

在路由规则列表页，单击路由规则右侧 **操作** 列中的 **删除** 后，确认删除即可。

## 7.3. API 分组

API 分组即 API 归属的分组，可以是具体的系统名、模块名或者抽象的标识。目前，移动网关支持创建以下四种类型的 API 分组：

- HTTP：符合 RESTful 风格的 HTTP 服务，支持跨专有网络 VPC 调用。
- DUBBO：基于 Apache Dubbo RPC 分布式服务框架。
- TR：基于蚂蚁金服 SOFARPC 服务框架。
- HRPC：mPaaS 网关独有的 RPC 框架。

#### 🔔 重要

控制台默认不打开 HRPC 选项，如需选择 HRPC，请联系技术支持人员开启 HRPC 选项。

### 创建分组

创建 Dubbo API 分组或 TR API 分组前，确保已将定义好的 API 服务注册到指定的注册中心，具体参见 [注册 Dubbo API 服务](#) 和 [注册 TR API 服务](#)。

登录 mPaaS 控制台后，从左侧导航栏进入移动网关页面。根据业务需要，创建不同类型的 API 分组。

- [创建 HTTP API 分组](#)
- [创建 Dubbo API 分组](#)
- [创建 TR API 分组](#)
- [创建 HRPC 分组](#)

### 创建 HTTP API 分组

完成以下步骤，创建 HTTP API 分组：

- 选择 **API 分组** 标签，进入 API 分组列表页。
- 单击 **创建 API 分组** 按钮，在弹出的对话框中填写 API 分组信息。



- **分组类型**：必填，选择 HTTP。
- **API 分组**：必填，填写服务的业务系统的英文名称。
- **多中心**：选填，提供多中心支持能力，方便在集群出现故障时能够快速容灾。开启多中心后，需要填写各数据中心的地址。目前仅支持配置两个数据中心。

**重要**

多中心功能与服务地址路由功能不兼容，只能二选一。

- **服务选择**：选填，支持将服务请求发送至服务地址，或者按照路由规则将请求路由到后端系统。若开启了多中心，则该配置项不可见。
  - **服务地址**：输入业务系统的 HTTP/HTTPS URL。
  - **路由规则**：选择已配置的路由规则，即当网关接收到请求后使用的路由策略。

**说明**

目前仅支持根据权重选择路由。

- **超时时间**：选填，发送请求至业务系统时的超时时间，单位毫秒，默认值：3000 ms。

3. 完成 API 分组信息配置后，单击 **确定** 完成分组创建。

### 创建 Dubbo API 分组

完成以下步骤，创建 Dubbo API 分组：

1. 选择 **API 分组** 标签，进入 API 分组列表页。
2. 单击 **创建 API 分组** 按钮，在弹出的对话框中填写 API 分组信息。



- **分组类型**：选择 DUBBO。
  - **API 分组**：必填，填写服务的业务系统的英文名称。API 分组名要与注册的 API 服务应用名保持一致。
  - **超时时间**：选填，发送请求至业务系统时的超时时间，单位毫秒，默认值：3000 ms。
  - **多中心**：选填，提供多中心支持能力，方便在集群出现故障时能够快速容灾。开启多中心后，需要填写各数据中心的地址。目前仅支持配置两个数据中心。
  - **注册中心**：填写注册中心地址，支持 ZooKeeper 集群或者直连。若开启了多中心，则该配置项不可见。
  - **注册中心鉴权**：选填，支持对注册中心进行权限管控，即只有通过鉴权的用户才可以访问注册中心。打开注册中心鉴权开关后，需要设置相应的用户名和密码。
3. 完成 API 分组信息配置后，单击 **确定** 完成分组创建。

### 创建 TR API 分组

完成以下步骤，创建 TR API 分组：

1. 选择 **API 分组** 标签，进入 API 分组列表页。
2. 单击 **创建 API 分组** 按钮，在弹出的对话框中填写 API 分组信息。

- **分组类型**：选择 TR。
  - **API 分组**：必填，填写服务的业务系统的英文名称。API 分组名要与注册的 API 服务应用名保持一致。
  - **超时时间**：选填，发送请求至业务系统时的超时时间，单位毫秒，默认值：3000 ms。
  - **开启直连**：选填，开启直连后，所有的流量都会请求直连地址。若要用于生产环境，请评估风险后再使用。若要开启直连，需配置直连地址。
3. 完成 API 分组信息配置后，单击 **确定** 完成分组创建。

### 创建 HRPC 分组

完成以下步骤，创建 TR API 分组：

1. 选择 **API 分组** 标签，进入 API 分组列表页。
2. 单击 **创建 API 分组** 按钮，在弹出的对话框中填写 API 分组信息。

- **分组类型**：选择 HRPC。
- **API 分组**：必填，填写服务的业务系统的英文名称。API 分组名要与注册的 API 服务应用名保持一致。
- **超时时间**：选填，发送请求至业务系统时的超时时间，单位毫秒，默认值：3000 ms。
- **注册中心**：必填，HRPC 目前支持两种对后端的路由方式。通过 Zookeeper 注册中心进行服务发现，协议为 `zookeeper://`；直连业务服务器方式，协议为 `direct://`。

#### ② 说明

使用直连业务服务器方式方式后，网关将会把所有流量路由到指定的 Direct IP。目前 Direct 方式只支持设置 **单 IP + 端口**，例如：`direct://x.x.x.x:8989`。

3. 完成 API 分组信息配置后，单击 **确定** 完成分组创建。

### 配置分组

根据不同的 API 分组类型，完成相应的 API 分组配置。

#### 配置 HTTP API 分组

在分组列表中，找到类型为 **HTTP** 的分组，在其右侧的 **操作** 列中单击 **详情**，进入 HTTP API 分组详情页面。单击右上方的 **修改** 可配置分组。HTTP 分组的配置项如下：

基础信息 保存 取消

服务地址:

超时时间:  ms

是否验签:  开启

加签算法:  MD5  RSA  SM2  SM3

密钥Key:

密钥内容:

- **多中心**：开启或关闭多中心。
- **服务地址**：HTTP 服务的 URL 地址。
- **超时时间**：单位毫秒，默认 3000 ms。
- **是否验签**：业务系统如需验证调用者的身份，请开启该项。有关如何验证，参见 [后端签名校验说明](#)。开启验签开关后，需完成以下配置：
  - **加签算法**：生成签名的算法。移动网关除支持 MD5、RSA、SM2、SM3 算法外，还支持 MOBILEGW 的签名算法。
  - **密钥 Key**：后端签名使用的密钥 Key，可以自定义。
  - **密钥内容**：后端签名使用的密钥 Value。
    - 当加签算法是 MD5 时，可以自定义。
    - 当加签算法是 RSA 时，为移动网关的公钥。
    - 当加签算法是国密 SM2 或 SM3 时，可以自定义。
    - 当加签算法是 MOBILEGW 时，请向域内网关维护人员申请。关于如何生成密钥，参见 [密钥生成方法](#)。

### 配置 Dubbo API 分组

在分组列表中，找到类型为 **Dubbo** 的分组，在其右侧的 **操作** 列中单击 **详情**，进入 Dubbo API 分组详情页面。单击右上方的 **修改** 可配置分组。Dubbo 分组的配置项如下：

- **多中心**：开启或关闭多中心。
- **注册中心**：修改 Dubbo 服务的配置中心地址。
- **超时时间**：单位毫秒，默认 3000 ms。
- **注册中心鉴权**：打开或关闭注册中心鉴权。若打开，则设置相应的用户名和密码。

### 配置 TR API 分组

在分组列表中，找到类型为 **TR** 的分组，在其右侧的 **操作** 列中单击 **详情**，进入 TR API 分组详情页面。单击右上方的 **修改** 可配置分组。TR 分组的配置项如下：

- **直连地址**：TR 服务的直连地址，由 IP 和端口组成，端口不指定时默认为 12200。
- **超时时间**：单位毫秒，默认 3000 ms。

### 配置 HRPC 分组

在分组列表中，找到类型为 **HRPC** 的分组，在其右侧的 **操作** 列中单击 **详情**，进入 **HRPC API 分组** 详情页面。单击右上方的 **修改** 可配置分组。HRPC 分组的配置项如下：

- **超时时间**：单位毫秒，默认 3000 ms。
- **注册中心**：HRPC 服务的配置中心地址。

## 7.4. API 管理

### 7.4.1. 注册 API

移动网关支持多种不同类型的 API 服务。本文介绍如何通过控制台添加 API 来完成 API 注册。

针对不同类型的 API 服务，需要进行的操作有所不同，下面分别予以介绍。

添加 Dubbo 和 TR API 前，确保已创建好相应的 Dubbo 和 TR API 分组。

- [HTTP API](#)
- [Dubbo API](#)
- [TR API](#)
- [HRPC API](#)

#### 添加 HTTP API

登录 mPaaS 控制台，完成以下步骤注册 HTTP API：

1. 在左侧导航栏，点击 **后台服务管理** > **移动网关** 菜单。
2. 在 **API 管理** 标签页，点击 **创建 API**。
3. 在弹出的对话框中，勾选 **HTTP API** 类型。
4. 在 **operationType** 栏，输入值后，点击 **确定** 完成注册。
  - **operationType** 为当前环境和 APP 下 API 服务的唯一标识。
  - **operationType** 定义规则：`组织.产品域.产品.子产品.操作`。

#### 添加 Dubbo API

完成以下步骤添加 Dubbo API：

1. 在左侧导航栏，点击 **后台服务管理** > **移动网关**。
2. 在 **API 管理** 标签页，点击 **创建 API**。
3. 在弹出的对话框中，勾选 **Dubbo** API 类型。
4. 选择对应的 API 分组。在 API 分组中，从获取到的 API 列表中选择需要注册的 API 服务。
5. 点击 **确定** 提交。

### 添加 TR API

完成以下步骤自动拉取 TR API：

1. 在左侧导航栏，点击 **后台服务管理** > **移动网关**。
2. 在 **API 管理** 标签页，点击 **创建 API**。
3. 在弹出的对话框中，勾选 **TR** 的 API 类型。
4. 选择对应的 API 分组。在 API 分组中，从获取到的 API 列表中选择需要注册的 API 服务。
5. 点击 **确定** 提交。

### 添加 HRPC API

登录 mPaaS 控制台，完成以下步骤自动拉取 HRPC API：

1. 在左侧导航栏，单击 **后台服务管理** > **移动网关**。
2. 在 **API 管理** 标签页，单击 **创建 API**。
3. 勾选 **HRPC** 的 API 类型。
4. 选择对应的 API 分组。在 API 分组中，从获取到的 API 列表中选择需要注册的 API 服务。
5. 点击 **确定** 提交。

## 7.4.2. 配置 API

### 7.4.2.1. 操作步骤

注册 API 服务后，您需要进行相关配置才能使用 API 服务，尤其是 HTTP 类型的 API 服务。只有 API 服务状态为 **开通** 才能被调用。您需要手动开通 HTTP 类型的 API。

#### 关于此任务

API 包括以下类型的参数配置：

- **基础信息配置**：API 名称、接口描述、接入系统等，不同类型的 API 服务还具备不同的属性。
- **高级配置**：签名校验、ETag 缓存、超时时间。
- **header 设置**：网关支持为所有请求添加或者删除 header。
- **限流配置**：对 API 调用进行限流配置。
- **熔断配置**：对 API 调用进行熔断配置。
- **缓存配置**：缓存 API 的响应，减轻业务系统压力。
- **参数设置**：请求参数设置、响应设置等，此项配置为 HTTP 类型的 API 服务特有。

欲了解详细的参数介绍及配置规则，单击以上配置类型名称。

要配置 API，完成以下步骤：

1. 登录 mPaaS 控制台，在左侧导航栏，单击 **移动网关**。
2. 在 API 列表中，选择要配置的 API 名称，单击操作列中的 **配置**，进入 API 详情页面。
3. 切换详情页右上角的 API 开关，以开通或关闭当前 API。
4. 在以下 API 不同的配置区域，单击 **修改** 进行相应参数的编辑。具体的配置方法，参见具体的配置信息。
5. 单击 **保存** 完成配置。

### 7.4.2.2. 基础信息配置

根据不同类型的 API 服务，编辑相应的参数值：

- HTTP API
  - **API 名称**：必填，该 API 的接口名称，方便后续维护。
  - **接口描述**：选填，更详细的 API 描述。
  - **接入系统**：必填，API 所属的业务系统。
  - **请求 Path**：必填，URL Path，可使用 `{}` 包含 path 参数，比如：`/pets/{id}`。
  - **请求方式**：必填，支持 GET、POST、PUT、DELETE 和 HEAD。
  - **报文编码**：必填，UTF-8 或 GBK 格式。
- Dubbo API
  - **API 名称**：必填，该 API 的接口名称，方便后续维护。
  - **接口描述**：选填，更详细的 API 描述。
  - **接入系统**：API 所属的业务系统。
  - **接口方法**：API 服务端方法。
  - **接口名称**：API 服务端接口。
- TR API
  - **API 名称**：必填，该 API 的接口名称，方便后续维护。
  - **接口描述**：选填，更详细的 API 描述。
  - **接入系统**：API 所属的业务系统。
  - **接口方法**：API 服务端方法。

- 接口名称：API 服务端接口。
- HRPC API
  - API 名称：必填，该 API 的接口名称，方便后续维护。
  - 接口描述：选填，更详细的 API 描述。
  - 接入系统：API 所属的业务系统。
  - 接口方法：API 服务端方法。
  - 接口名称：API 服务端接口。

### 7.4.2.3. 高级配置

- 签名校验：选择是否开启签名校验。若开启，将对客户端请求的签名进行校验。
- 超时时间：设置服务超时时间，单位毫秒（ms）。  
超时优先级：接口超时设置 > 系统超时设置 > 默认 3000 ms。
- 开放 JSONP：设置是否支持跨域 HTTP 请求。JSONP 允许快速跨域使用 API。

### 7.4.2.4. header 设置

网关支持为所有请求添加或者删除 header。在 **header 设置** 区域，单击 **修改** 打开编辑模式后，即可通过单击 **Add** 添加一条规则。每条规则包含 **位置**、**类型**、**headerKey**、**value** 和 **操作** 五个属性。

- **位置**：可选 request header 或 response header。
  - 添加 request header 会自动在 request 中增加 header，业务可以通过 MobileRpcHolder 获取该 header。
  - 添加 response header 会自动在 response 中增加 header，客户端可以从 response 中获取该 header。
- **类型**：可选增加（add）或删除（delete）。
  - 增加（add）：添加一个新的 header，如果原请求已有 header，则会被新增的 header 覆盖。
  - 删除（delete）：删除一个 header。删除 header 可以配置 value 也可以不配置。如果配置 value，那么只有在 value 匹配时才会删除。
- **headerKey**：headerKey 可以为符合 RFC 定义的任意字符串，除 HTTP 协议中的特有 header 之外，比如 host、content-Type 等；也不可以是 mpaasgw 中特有 header，比如 operation-Type 等。
- **value**：可以为任意字符串。
- **操作**：删除当前 header 规则。

#### ② 说明

定义 HTTP header 时，请不要使用下划线“\_”。

### 7.4.2.5. 限流配置

限流配置包括限流模式、限流值、限流响应：

- **限流模式**
  - 关闭：不限制 API 调用。
  - 拦截：当调用频次超过限流值，拦截请求。
- **限流值** 根据业务需求设置合理的限流阈值（单位：秒）。限流模式为拦截且超过此值时，请求会被限流。
- **限流响应** 限流默认的响应为：`{"resultStatus":1002,"tips":"顾客太多，客官请稍候"}` 如需定制限流响应，使用如下格式：

```
{
  "result": "==此处为定制响应内容,请填写==",
  "tips": "ok",
  "resultStatus": 1000
}
```

其中：

- `result` 为定制响应数据，JSON 格式。只有 `resultStatus` 为 1000 时，客户端才会取此字段处理。
- `tips` 为定制的限流提示。若 `resultStatus` 为 1002，会取此字段提示用户。
- `resultStatus` 为限流返回的结果码，具体含义请参见 [网关结果码说明](#)。

### 7.4.2.6. 熔断配置

#### 熔断配置

仅当 mPaaS 控制台 **网关管理** 页面中的熔断开关打开时，才可以进行熔断配置。

配置熔断机制后，当在指定的时间窗口内当前 API 调用失败次数达到配置的阈值时，将自动触发服务熔断，移动网关将不再转发该 API 的请求至后端服务，而是返回设定的响应内容。

熔断机制默认处于关闭状态，即不会对当前 API 做熔断处理，无需进行任何配置。

在 **熔断配置** 区域，单击 **修改**，打开 **熔断机制** 开关，进入编辑模式，设置熔断机制，及熔断的触发条件以及熔断响应：

- **API 调用失败阈值**：自定义，即当 API 调用失败次数达到设定的阈值后将触发熔断。
- **失败阈值对应的单位时间**：自定义，指 API 调用失败阈值对应的时间窗。在该时间窗口内，API 调用失败次数达到该阈值后将触发熔断。例如，在 3 分钟内失败 10 次就触发熔断。
- **熔断后恢复的时间间隔**：自定义，在该时间窗口内，网关将不再转发当前 API 的请求至后端服务。熔断恢复时间间隔过后，网关重新尝试对该 API 请求进行转发。
- **熔断响应**：自定义，服务熔断时返回的响应内容。如需定制响应内容，使用如下格式：
  - `result`：定制的熔断响应数据，JSON 格式。只有 `resultStatus` 为 1000 时，客户端才会取此字段处理。
  - `tips`：定制的熔断提示。一旦触发熔断，即会取该字段提示用户。

- `resultStatus` : 熔断返回的结果码, 由用户自定义。

### 7.4.2.7. 缓存配置

缓存 API 的响应, 减轻业务系统压力。具体配置参见 [API 缓存](#)。

### 7.4.2.8. 参数设置

参数设置适用于 HTTP 类型的 API 服务。通过自动导入方式创建的 API 无需配置参数。

- **请求参数** : 设置 Path 中的动态参数和 URL Query 参数, 参数名称保证唯一。
  - **参数名称** : 必填, 参数的名称。

② 说明

如果为 Path 参数, 请保证名称与请求 Path 中的一致。例如, 请求 Path 为 `/pets/{id}`, 此时参数名称应为 `id`。

- **参数位置** : 必填, 参数位于 Path 或者 Query String 中。
- **类型** : 必填, 可选择的类型有 String、Int、Long、Float、Double、Boolean。
- **默认值** : 选填, 参数的默认值。
- **描述** : 选填, 参数描述。
- **请求 Body** : 指定请求 Body 的数据模型及 Content-Type。

② 说明

仅当 API 的调用方式为 POST 时, 才会展示请求 Body 配置。

- **请求 Body 类型** : 支持的 Body 类型有 String、Int、Long、Float、Double、Boolean、List、Map、Object。
- **报文类型** : 支持的报文类型有 application/json、application/x-www-form-urlencoded、application/protobuf。
- **响应结果** : 指定响应结果的类型。
  - **响应结果类型** : 指基础类型或者自定义的数据模型。

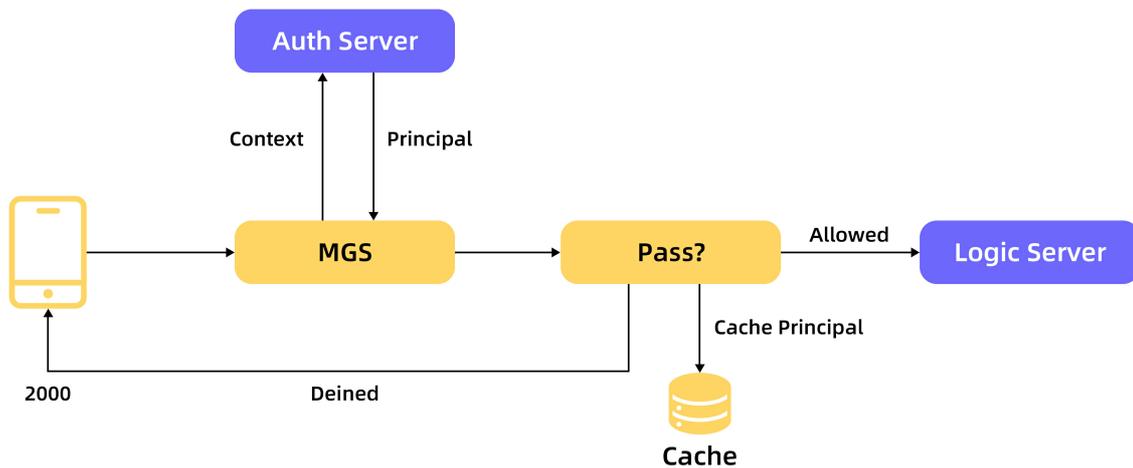
## 7.4.3. API 授权

了解 API 授权的使用场景。根据您的业务需求, 开启 API 授权、配置授权规则, 定义授权方接口, 并将授权规则应用到 API。

### 功能介绍

API 授权功能允许业务在 MGS 上定义通用的 API 访问授权规则:

1. 创建授权 API A 并在网关管理中配置, 然后到业务 API B 配置中做关联。
2. 客户端发起对后业务 API B 的请求时, MGS 会根据 API 授权配置从该请求 Header 或 Cookie 中取出授权参数放到 Context 里然后调用业务 API B 关联的授权 API A, 授权 API A Server 需要根据 Context 中的参数做业务权限校验。
3. 如校验合法, MGS 会将校验结果 Principal 添加在请求 Header 中, 传递给后端业务 API B。如需缓存, MGS 会缓存校验结果 Principal, 提高授权的性能。



### 使用场景

#### 场景一

客户有分布式 session, 登录后会产生会话 ID。授权过程如下:

1. 用户 A 请求登录接口, 登录成功后, 产生会话 ID 和会话信息保存到分布式缓存中, `sessionId: {username:A, age:18, ...}`, 并且下发 sessionId 到客户端。
2. 用户 A 请求一个需要登录授权的接口, 网关从请求 Header 中获取 sessionId, 发送给授权系统, 授权系统根据 sessionId 从分布式缓存中获取到用户信息, 并且将 `{username:A, age:18, ...}` 返回给网关。
3. 网关判断登录成功, 将 `{username:A, age:18, ...}` 添加在 Header 中, 转发请求到后端的业务 Server。

#### 场景二

客户端基于 HMAC 的授权方案, 授权过程如下:

1. 用户 A 登录成功后，下发一个 token 到客户端，`token=hmac(username+password)`。
2. 用户 A 请求一个需要登录授权的接口，网关从 Header 中获取 token，发送给授权系统，授权系统根据再算一遍 HMAC，如果符合则返回用户信息 `{username:A, age:18,...}` 给网关。
3. 网关判断登录成功，将 `{username:A, age:18,...}` 添加到请求 Header 中，转发请求到后端的业务 Server。

### 操作步骤

#### 配置授权规则

1. 登录 mPaaS 控制台。在左侧导航栏，点击 **后台服务管理** > **移动网关** 菜单。
2. 点击 **网关管理** 标签，在 **API 授权** 下方，点击 **创建授权方** 或点击已存在的授权规则记录列表中操作列中的 **详情**，进入授权规则配置页面：
  - **授权方名称**：必填，授权规则的名称。
  - **授权方接口**：必填，用于验证请求授权情况的接口。
  - **授权缓存**：是否缓存授权的验证结果。
  - **缓存TTL**：验证结果的缓存存活时间。
  - **身份来源**：如果点击 **添加来源字段**，填写用于授权的请求参数，表明请求身份的信息，由以下字段组成：
    - **位置**：参数所处位置，`header` 或者 `cookie`。
    - **字段**：参数名称。

#### 说明

如果实际 API 请求时的身份来源字段缺失，授权验证无法通过。

#### 定义授权方接口

#### 说明

如果后端系统提供的授权接口为 HTTP 类型，需要将授权 API 配置为 POST 方法。

在添加授权关系前，业务系统需要提前开发一个 `Auth API`。当 API 需要验证授权关系时，会调用 `Auth API` 进行授权校验。`Auth API` 的定义（请求和响应）遵循以下的标准：

#### AuthRequest

```
public class AuthRequest {
    private Map<String,String> context;
}
```

#### AuthResponse

```
public class AuthResponse {
    private boolean success;
    private Map<String,String> principal;
}
```

#### 接口示例

```
@PostMapping("/testAuth")
public AuthResponse testAuth(@RequestBody AuthRequest authRequest) {
    String sid = authRequest.getContext().get("sid");
    Map<String, String> principal = new HashMap<>();
    principal.put("uid", sid + "_uid");
    AuthResponse authResponse = new AuthResponse();
    authResponse.setSuccess(true);
    authResponse.setPrincipal(principal);
    return authResponse;
}
```

- 当验证授权的响应中 `success` 字段值为 `true` 时，网关会根据缓存策略缓存 `principal` 信息，然后将 `principal` 信息放入这次请求的 `header` 中，透传到后端业务系统中。没有 `principal` 也需要传个空 Map。
- 当验证授权的响应中 `success` 字段值为 `false` 时，网关会返回 2000 错误码，客户端需要根据 2000 做相应的操作，例如弹出登录框。

#### 使用授权规则

当授权规则配置后，可以在 API 配置页面中的 **高级配置** > **API 授权** 中选择对应的规则，为该 API 启用授权功能。

要使用 API 授权，确保已在 **网关管理** 页面中开启 **API 授权** 功能。开启步骤如下：

1. 登录 mPaaS 控制台。在左侧导航栏，点击 **后台服务管理** > **移动网关** 菜单。
2. 点击 **网关管理** 标签，确保 **API 授权** 按钮开启。

该 API 在请求后端系统前，会进行授权验证。通过则接受请求，网关将请求路由到后端系统。否则，请求会被拒绝，调用方将收到授权失败的错误响应。

## 7.4.4. API 限流

API 限流不仅支持对单个 API 进行限流设置，还支持对 API 设置限流默认值以及设置应用级别的限流总值，避免高峰期间后台服务器被压垮。如果同时设置了 API 限流默认值和 App 限流总值，则按照限流值的大小依次处理，限流值较小的优先生效。

本文仅对如何配置 API 限流默认值和 App 限流总值进行说明。如需对单个 API 进行限流设置，请在 API 详情页面的限流配置模块中进行设置，具体参见 [配置 API](#)。

#### 前置条件

要使用限流配置，确保已开启限流功能。登录 mPaaS 控制台，从左侧导航栏进入 **移动网关** > **网关管理** 页面，打开 **限流** 开关即可。

#### API 限流默认值

对 API 设置限流默认值，作用于当前应用下的所有 API。限流默认配置生效规则如下：

- 若之前已针对单个 API 配置过限流值，则该 API 的限流值以之前配置的为准。
- 单个 API 的限流配置会覆盖 API 默认限流配置。
- 修改后的限流默认配置对之前已使用限流默认配置的 API 生效。

配置步骤：

1. 打开 **API 限流默认配置** 开关。
2. 在默认限流值配置框中，单击 **编辑**，配置限流信息。
  - **默认限流值**：根据业务需求设置合理的限流阈值。超过此值时，请求会被限流。

② 说明

限流阈值指一秒内的请求上限。

- **限流响应**：限流默认响应为：`{"resultStatus":1002,"tips":"顾客太多，客官请稍候"}`。如需定制限流响应，请使用如下格式：

```
{
  "result": "==此处为定制响应内容，请填写==",
  "tips": "ok",
  "resultStatus": 1000,
}
```

其中：

- `result` 为定制响应数据，`JSON` 格式。只有 `resultStatus` 为 1000 时，客户端才会取此字段处理。
- `tips` 为定制的限流提示。若 `resultStatus` 为 1002，会取此字段提示用户。
- `resultStatus` 为限流返回的结果码，具体含义请参见 [网关结果码说明](#)。

## App 限流总值

对当前应用下所有的 API 设置限流的总值。一旦超出应用的限流总值，则当前应用下所有 API 的请求都将被限流。

配置步骤：

1. 打开 **App 限流总值** 开关。
2. 在 App 限流总值配置框中，单击 **编辑**，配置限流信息。
  - **限流总值**：根据业务需求设置合理的限流阈值（单位：秒）。超过此值时，请求会被限流。
  - **限流响应**：限流默认响应为：`{"resultStatus":1002,"tips":"顾客太多，客官请稍候"}`。如需定制限流响应，使用如下格式：

```
{
  "result": "==此处为定制响应内容，请填写==",
  "tips": "ok",
  "resultStatus": 1000,
}
```

其中：

- `result` 为定制响应数据，`JSON` 格式。只有 `resultStatus` 为 1000 时，客户端才会取此字段处理。
- `resultStatus` 为限流返回的结果码，具体含义请参见 [网关结果码说明](#)。
- `tips` 为定制的限流提示。若 `resultStatus` 为 1002，会取此字段提示用户。

## 7.4.5. API 缓存

配置 API 缓存信息，缓存 API 的响应，减轻业务系统压力。

### 关于此任务

API 缓存包含整个后端请求的响应，包括响应头和响应体，所以要求被缓存的 API 的响应头中排除状态类信息，比如 `cookie` 中的用户态数据。该缓存功能只适合缓存无状态数据。

后端业务系统可以通过在响应头中添加 `Pragma: no-cache` 告知网关不缓存该次响应。

### 操作步骤

1. 登录 mPaaS 控制台，在左侧导航栏，点击 **后台服务管理** > **移动网关** 菜单。
2. 在 API 列表中，选择要配置的 API 名称，点击操作列中的 **配置**，进入 API 详情页面。
3. 点击 **缓存配置** 区域的 **修改** 按钮，配置如下规则：
  - **结果缓存**：是否开启缓存。
  - **缓存时间**：缓存的存活时间，单位：秒。
  - **缓存键值**：用于缓存的键值表达式。点击 **修改** 定义缓存键值。在弹出的模态框中，填写缓存所需的主键，可以拖曳进行排序。关于键值语法的信息，参见下文的键值语法。

### 键值语法

#### 支持的语法

当 API 请求到达网关后，网关会根据键值配置获取相应的数据，作为缓存的键值，其语法如下：

语法	描述
\$	根对象，例如：\$.bar
[num]	数组访问，其中 num 是数字。例如：\$[0].bar.foos[1].name

.	属性访问，例如：\$.bar
['key']	属性访问，例如：\$['bar']
\$.header	API 请求头对象，用于获取请求头中的字段，例如：\$.header.remote_addr
\$.cookie	API 请求 cookie 对象，用于获取 cookie 中的值，例如：\$.cookie.session_id
\$.http_body	后端为 HTTP 类型的请求体对象，用于获取请求体中的字段，例如：\$.http_body.name
\$.http_qs	后端为 HTTP 类型的请求参数对象，用于获取请求参数，例如：\$.http_qs.name

### 代码示例

以如下请求数据为例，从请求报文中获取对象：

```
URL:/json.htm?tenantId=boo

Header:
Content-Type:application/json
opt:com.mobile.info.get
workspaceId:default
appId:B2D553102
cookie:JSESSIONID=abcd;traceId=trace1000

Body:
[
  {
    "key": "1234",
    "locations": [
      "beijing",
      "shanghai"
    ],
    "language": "zh-Hans",
    "unit": "c"
  },
  {
    "demo": {
      "name": "nick"
    }
  }
]
```

如下显示表达式的示例：

```
$.header.appId = B2D553102
$.cookie.traceId = trace1000
$.http_qs.tenantId = boo
$[0].key = 1234
$[0].locations[1] = shanghai
$[1].demo.name = nick
```

## 7.4.6. API 模拟

API 模拟是对某一 API 的返回值进行模拟 (Mock)，以提供特定的响应结果。要使用 API 模拟功能，需先前往 **网关管理 > 功能开关** 页面打开 API Mock 开关。

### 操作步骤

完成以下操作步骤配置 API Mock：

1. 选择 **API 管理** 选项卡 > API 列表操作列的 **更多 > API Mock**。
2. 在 **Mock 配置** 页面，完成以下参数配置：
  - **API Mock**：开启 API Mock 开关。
  - **命中规则**：当前支持 **百分比** 规则。
  - **规则配置**：具体的百分比数值，取值在 0 ~ 100 之间。
  - **Mock 数据**：Mock 的 API 响应数据。Mock 数据格式如下：

```
{
  "resultStatus": 1000,
  "tips": "ok",
  "result": "==此处为Mock的业务数据,请填写=="
}
```

其中：

- `resultStatus` 为响应结果码，具体含义请参见 [网关结果码说明](#)。
- `tips` 为响应提示。
- `result` 为定制的响应数据，JSON 格式。

3. 点击 **提交**。

## 7.4.7. 同步 API

同步 API 可将当前环境中的 API 同步至其他环境中。

为保证系统稳定性，只同步目标环境不存在的 API 配置。同步后，会自动加载配置使其生效。

### 操作步骤

1. 登录 mPaaS 控制台，在左侧导航栏，点击 **后台服务管理** > **移动网关** 菜单。
2. 在 **API 管理** 标签页中，点击 **更多** > **同步 API**。



3. 在 **基础信息** 栏，选择要从当前环境同步 API 配置同步的 **目标环境**。
4. 选择 **同步后状态**：
  - **保持原样**：跟当前环境的 API 配置保持不变。
  - **全部关闭**：同步后的 API 状态置为关闭。
5. 在 **所选 API** 栏，选择需要同步的 API。
6. 点击 **确定** 按钮开始同步 API。

### 结果

同步成功后，当前页面上方会出现提示，显示同步的结果。

## 7.4.8. 导出及导入 API

为便于将当前 API 配置应用于其他环境或其他应用，移动网关服务支持将当前应用的 API 以 .txt 文件形式导出。同时，您也可以将 API 配置导入并应用到当前环境。下面是对 API 导出和导入操作的具体介绍。

### 导出 API

按需选择要导出的 API，导出操作会将 API 关联的分组、数据模型一并导出。

**说明**  
仅支持导出 HTTP 类型的 API。

操作步骤如下：

1. 登录 mPaaS 控制台，在左侧导航栏，点击 **后台服务管理** > **移动网关** 菜单。
2. 在 **API 管理** 标签页中，点击 **更多** > **导出 API**。
3. 在 **所选 API** 栏，选择需要导出的 API。
4. 点击 **确定** 按钮开始导出 API。导出的 API 保存在一个 .txt 文件中。

### 导入 API

为保证系统稳定性，在 **保留** 或 **覆盖** 中，建议选择 **保留** 作为导入策略。

操作步骤如下：

1. 在 **API 管理** 标签页中，点击 **更多** > **导入 API**。
2. 确认 AppID 和当前环境是否正确。
3. 选择 **导入策略**：当导入配置与已有配置存在冲突时所采用的策略。
  - **保留**：当导入的配置与已有配置冲突时，将保留已有配置、放弃导入配置。
  - **覆盖**：当导入的配置与已有配置冲突时，将采用导入配置、替换已有配置。
4. 点击 **文件上传**，选择要导入的 API 文件。
5. 点击 **确定** 按钮开始导入 API。导入成功后，会显示导入的结果。

## 7.5. API 调用

### 7.5.1. API 测试

在 API 测试中可测试当前 API 的功能性是否完好。要进行 **API 测试 (Test)**，在 **API Test** 页面完成相关配置。

**重要**

com.antcloud.session.validate 的 operationType 为 MSS 组件的专用接口，如果配置了 com.antcloud.session.validate 的 operationType 且开启网关加密，在 API Test 中测试该 operationType 时会报 6004 错误。

**操作步骤**

1. 选择 **API 管理** 选项卡 > API 列表操作列 **更多** > **API Test**，打开如下页面：

< API Test: com.mpaas.getArticleList

请求参数设置

\* 请求路径:

\* 请求参数:

```
{
  "pageNo": "请求的第几页"
}
```

+ 增加 Header

测试

响应 body

{}

响应 header

{}

2. 在 **API Test** 页面中，完成以下配置：

  - **请求路径**：应用所在环境的网关地址。例如，mPaaS 移动网关的公有云地址为

`https://cn-hangzhou-mgs-gw.cloud.alipay.com/mgw.htm`

**重要**

地址必须带有 `/mgw.htm`，否则请求会失败。

- **请求参数**：默认会模拟符合请求参数格式的数据，具体请按照业务含义调整。
  - **Header**：您可以按需增加 Request Header。
3. 点击 **测试** 按钮，右侧会得到请求执行的如下结果：
    - **响应 body**：业务返回的响应数据。
    - **响应 header**：包括网关约定以及业务返回的 Response Header。
      - Result-Status：具体参见 [网关结果码说明](#)。
      - Mgw-TraceId：该请求的 TraceId，可以据此进行链路分析。

## 7.5.2. 生成代码

移动网关提供生成 API 的客户端 SDK 功能。

**关于此任务**

只有 HTTP 类型的 API 支持单个代码生成。

**操作步骤**

1. 通过以下的操作方式打开 **客户端代码生成** 窗口：
  - 方式一：在 **API 分组** 标签页中，点击系统列表操作列中的 **生成代码** 按钮。
  - 方式二：在 **API 管理** 标签页中，点击列表上方的 **生成代码** 按钮。
  - 方式三：在 **API 管理** 标签页中，点击列表操作列中的 **更多** > **生成代码** 按钮。该方式用于生成单个 API 代码（非 API 分组），仅适用于 HTTP 类型。

客户端代码生成

\* API分组:

Platform:  Android  iOS  JS

PackageName:

取消 提交

2. 在 **客户端代码生成** 窗口，完成以下信息配置：

  - **API 分组**：选择需要生成 SDK 的 API 分组。
  - **Platform**：选择 Android、iOS 或 JS。

- 若选择 **Android**，在 **PackageName** 中，填写客户端代码的包名；若不填，默认为 `com.client.service`。
- 若选择 **iOS**，在 **Prefix** 中，填写唯一前缀；若不填，默认不加前缀。

3. 点击 **提交** 生成 API SDK 供客户端调用。

### 7.5.3. HTTP API 请求格式

mPaaS 客户端到移动网关的 MRPC 协议是基于 HTTP 的 RPC 协议。当通过移动网关的 API 测试页面或直接通过 Postman 等非 mPaaS 客户端调用 API 时，可以参考如下格式手动构造对应的 RPC 请求。

#### API 测试页面

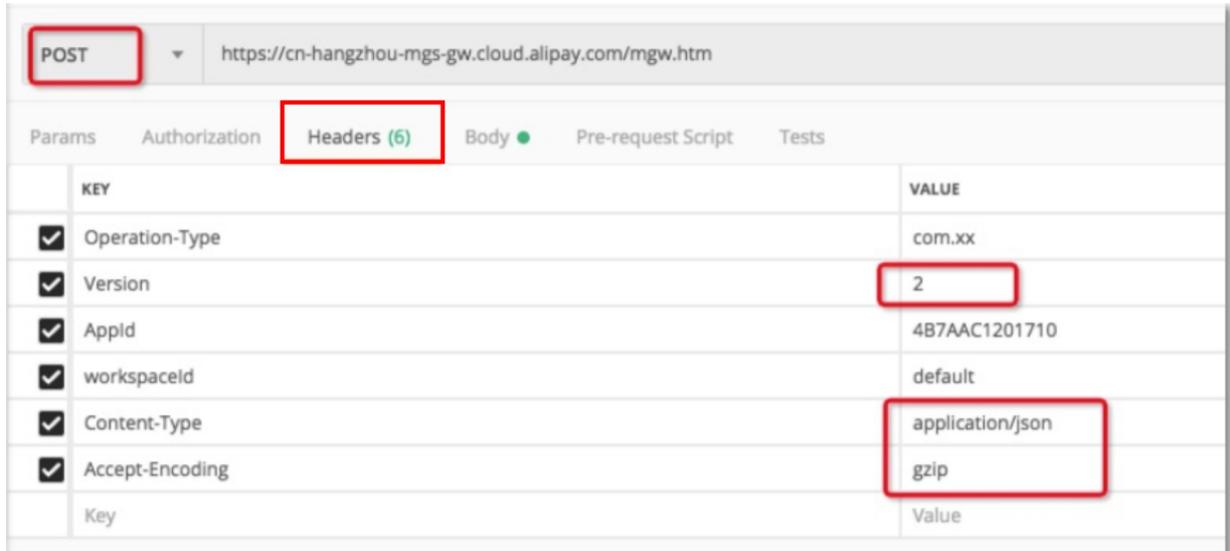
API 测试页面中请求参数格式为 `{{}}`。当该 API 后端为 HTTP 服务时，到后端 HTTP 服务 `get` 请求的 query 参数在 `{{}}` 里平铺；到后端 HTTP 服务 `post` 请求的 body 参数放在 key 为 `_requestBody` 的 value 中。

#### Postman 等非 mPaaS 客户端

##### 🔔 重要

直接通过 Postman 等非 mPaaS 客户端调用 API 时，需要在移动网关控制台的 **网关管理** 页面中关闭 **签名校验** 和 **数据加密** 功能开关，否则会提示请求失败。

下图中，红框标示的参数为固定参数，其他参数可换成具体 API 的参数。注意，如果后端 HTTP 服务是 `get` 类型的服务，这里的 `post` 依然固定不能改变，只需在移动网关控制台的 **API 详情** 页面 > **基础信息** 区域中将调用方式改成 `get` 即可。



请求中 Body 参数的格式为 `{{}}`。当 API 后端为 HTTP 服务时，到后端 HTTP 服务 `get` 请求的 query 参数在 `{{}}` 里平铺；到后端 HTTP 服务 `post` 请求的 body 参数放在 key 为 `_requestBody` 的 value 中。

## 7.6. 网关管理

### 7.6.1. 网关管理功能介绍

网关管理页面提供功能开关设置、结果码定制、操作记录查询和链路分析功能。

#### 功能开关

功能开关是全局的，可以根据需要暂时地开启或者关闭所有的 API 相关功能。

#### 签名校验

对客户端到移动网关的请求进行验证，以验证调用者身份保证安全，默认 **打开**。

#### 验签时间戳

在开启签名校验功能的情况下，支持自定义配置验签时间戳的有效时长，避免因用户调整手机时间（例如将手机时间调慢），造成设备获取的时间戳不一致，进而导致 API 验签时间戳超时的问题。

目前，支持配置的验签时间戳有效期上限为 10 年 (5,256,000 分钟)。

#### API 限流

API 限流不仅支持对单个 API 进行限流设置，还支持对 API 设置限流默认值以及设置应用级别的限流总值，避免高峰期后台服务器被压垮。如果同时设置了 API 限流默认值和 App 限流总值，则按照限流值的大小依次处理，限流值较小的优先生效。

- [单个 API 限流配置](#)
- [API 限流默认值和 App 限流总值配置](#)

#### 熔断机制

熔断机制是指在 API 服务出现异常的情况下，在接下来的时间窗口内，自动切断对当前 API 的服务调用，返回设定值或者抛出异常，以避免资源浪费，防止级联故障。

支持对单个 API 进行熔断配置。配置熔断机制后，当在指定的时间窗口内 API 调用失败次数达到配置的阈值时，将自动触发熔断。在配置的熔断恢复时间间隔内，网关将不再转发该 API 的请求至后端服务。熔断恢复时间间隔过后，网关重新尝试对该 API 请求进行转发，如后端服务成功返回，将取消熔断状态；否则将继续熔断，下一次熔断恢复时间间隔后再进行检查。

#### API Mock

对某一 API 的返回值进行 Mock，以提供特定的响应结果，默认 **关闭**。

### API 授权

在网关将客户端请求路由到后端业务系统之前，校验该次请求的合法性，验证通过才予以放行，默认 **关闭**。

具体配置参见 [API 授权](#)。

### 数据加密

对客户端到移动网关的请求进行加密，确保数据在传输过程中的安全性，默认 **关闭**；目前支持的加密算法有 ECC 和 RSA，该功能需要与客户端配合使用。如果客户端数据加密方式与此处设置的方式不一致，网关将无法解析客户端请求。

具体配置参见 [数据加密](#)。

### CORS

跨域资源共享，根据规则进行跨域访问控制。如果需要支持跨域请求，请配置该规则。

具体配置参见 [跨域资源共享](#)。

### 结果码定制

网关结果码都有默认的提示文案，您也可以根据实际需求定制结果码提示。

选择 **网关管理** 选项卡，点击右侧 **结果码定制** 进入定制页面。

### 操作记录

**操作记录** 对配置人员在网关上的相关动作进行记录和展示，方便进行追溯。

### 常用工具

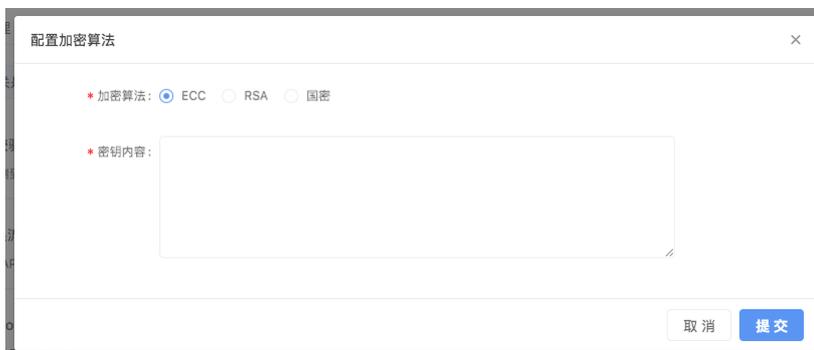
**链路分析** 可以对 `TraceId` 进行分析，解析出对应的时间和网关服务器。

## 7.6.2. 数据加密

要进行数据加密，在服务端，您需要进行相关配置生成密钥；在客户端，根据不同的操作平台，完成相应的配置。

### 服务端配置

1. 登录 mPaaS 控制台，在左侧导航栏，点击 **后台服务管理 > 移动网关** 菜单。
2. 选择 **网关管理** 选项卡，点击右侧的 **功能开关** 标签页。
3. 将 **数据加密** 状态切换至 **开**。
4. 在弹出的 **配置加密算法** 窗口，完成以下配置：
  - 加密算法：支持 ECC、RSA 和国密（SM2）。
  - 密钥内容：
    - 当加密算法为 ECC 或国密时，填写私钥内容。
    - 当加密算法为 RSA 时，分别填写公私钥内容。



有关加密算法的密钥生成方法，参见 [密钥生成方法](#)。

### 客户端配置

#### Android 配置

在 `assets` 目录下新建 `mpaas_netconfig.properties` 文件，用于存放网络相关全局配置。

```
mpaas_netconfig.properties x
1 Crypt=true
2 RSA/ECC/SM2=SM2
3 PublicKey=-----BEGIN PUBLIC KEY-----\nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEUR+80YGaGkcEDqNR73UlwIGGqM18W\ntbfZiUfHqebEMPOSdd
4 GWhitelList=http://21.96.92.106:80/mgw.htm
```

- `Crypt`：表示是否使用自加密，`true` 表示使用，`false` 表示关闭自加密功能。
- `RSA/ECC/SM2`：表示要使用的非对称加密算法，其值只能填充 `RSA` 或 `ECC` 或 `SM2`。
- `PublicKey`：表示选择的非对称加密算法的公钥。

说明

由于 Android 中 `properties` 文件的 `value` 值需要在同一行，因此填充公钥时需注意使用换行符 `\n` 将 `Pubkey` 转为为一行。

- `GWWhiteList`：需要进行加密的网关，即当前环境的网关地址（在 mPaaS 控制台获取的配置文件中的 `rpcGW` 字段）。如果没有该 `key`，则所有的请求都不会加密。

### iOS 配置

iOS 端加密配置是从 `info.plist` 里面读取，如下图所示：

▼ mPaaSCrypt	Dictionary	(4 items)
Crypt	Boolean	YES
▼ GWWhiteList	Array	(2 items)
Item 0	String	http://192.168.1.1:8080/mgw.htm
Item 1	String	http://192.168.1.1:9080/mgw.htm
PubKey	String	-----BEGIN PUBLIC KEY-----
RSA/ECC/SM2	String	RSA

- `mPaaSCrypt`：加密配置的主 `key`，`value` 是 `Dictionary` 类型，里面包含了客户端加密所需设置的相关信息。
  - `Crypt`：是否进行加密，`value` 是 `Boolean` 类型，`YES` 代表加密，`NO` 代表不加密。
    - `Crypt` 设置为 `NO` 时，RPC 不进行加密，`RSA/ECC/SM2` 和 `PubKey` 的设置会忽略。
    - `Crypt` 设置为 `YES` 时，`RSA/ECC/SM2` 和 `PubKey` 必须设置且不能为空字符，否则 `Debug` 时会中断言，程序直接退出。
  - `GWWhiteList`：需要进行加密的网关，即当前环境的网关地址（在 mPaaS 控制台获取的配置文件中的 `rpcGW` 字段）。如果没有该 `key`，则所有的请求都不会加密。
  - `RSA/ECC/SM2`：非对称加密算法选择，`value` 是 `String` 类型，只能填 `RSA` 或 `ECC` 或 `SM2`。
    - `RSA/ECC/SM2` 和 `PubKey` 的设置必须一一对应。
    - 选择 `RSA` 算法，`PubKey` 填 `RSA` 对应的公钥。
    - 选择 `ECC` 算法，`PubKey` 填 `ECC` 对应的公钥。
    - 选择 `SM2` 算法，`PubKey` 填 `SM2` 对应的公钥。
  - `PubKey`：非对称加密公钥。`value` 是 `String` 类型，与选择的非对称加密算法保持一致。
- `PubKey` 格式必须携带 `-----BEGIN PUBLIC KEY-----` 及 `-----END PUBLIC KEY-----`，格式如下：

```
-----BEGIN PUBLIC KEY-----
*****
*****
*****
*****
-----END PUBLIC KEY-----
```

## 7.6.3. 跨域资源共享

跨域访问是指请求一个与自身资源不同源（不同的域名、协议或端口）的资源。不同源可以是不同的域名、协议或端口。

### CORS

跨域访问是指请求一个与自身资源不同源（不同的域名、协议或端口）的资源。不同源可以是不同的域名、协议或端口。

浏览器出于安全考虑设置了同源策略，限制了从脚本内发起跨域请求。但在实际应用中，经常会发生跨域访问。为此，W3C 提供了一个标准的跨域解决方案：[跨域资源共享 \(Cross-Origin Resource Sharing, CORS\)](#)，支持安全的跨域请求和数据传输。

浏览器将 CORS 请求分为以下两类：

- 简单请求
- 预检请求：防止资源被本来没有权限的请求修改的保护机制。浏览器会在发送实际请求之前使用 `OPTIONS` 方法发送一个预检请求，从而获知服务端是否允许该跨域请求。服务端确认后，才会发起实际的 HTTP 请求。

### 简单请求

请求满足如下所有条件，为简单请求：

- 请求方法是如下之一：
  - HEAD
  - GET
  - POST
- HTTP 头信息不超过以下几种字段：
  - Cache-Control
  - Content-Language
  - Content-Type
  - Expires
  - Last-Modified
  - Pragma
  - DPR
  - Downlink

- Save-Data
- Viewport-Width
- Width
- Content-Type 的值仅限下列几种：
  - text/plain
  - multipart/form-data
  - application/x-www-form-urlencoded

### 预检请求

不符合简单请求条件的请求，会在正式通信之前触发一个 OPTIONS 请求进行预检。这类请求为预检请求。

预检请求会在请求头中附带一些正式请求的信息给服务端，主要有：

- Origin：请求源信息。
- Access-Control-Request-Method：接下来的请求类型，如 POST、GET 等。
- Access-Control-Request-Headers：接下来的请求中包含的用户显式设置的 Header 列表。

服务端收到预检请求后，会根据上述附带的信息判断是否允许跨域，通过响应头返回对应的信息：

- Access-Control-Allow-Origin：允许跨域的 Origin 列表。
- Access-Control-Allow-Methods：允许跨域的方法列表。
- Access-Control-Allow-Headers：允许跨域的 Header 列表。
- Access-Control-Expose-Headers：允许暴露的 Header 列表。
- Access-Control-Max-Age：最大的浏览器缓存时间，单位：秒。
- Access-Control-Allow-Credentials：是否允许发送 Cookie。

浏览器会根据返回的 CORS 信息判断是否继续发送真实的请求。以上行为都是浏览器自动完成的，服务端只需要配置特定的 CORS 规则。

### 网关对 CORS 的支持

网关提供了配置 CORS 规则的功能，让业务方决定是否允许特定的跨域请求。该规则以 appId + workspaceId 维度配置。

#### 配置 CORS

登录 mPaaS 控制台，完成以下步骤：

1. 在左侧导航栏，点击 后台服务管理 > 移动网关 菜单。
2. 选择 网关管理 标签页，点击右侧的 功能开关 标签页，进行 CORS 配置。

开启 CORS 后，app 在该 workspace 下的所有 API 服务都将支持符合以下配置的跨域请求：

- 允许来源：Access-Control-Allow-Origin，可以设置多个，逗号分割，允许“\*”通配符。
- 允许方法：Access-Control-Allow-Methods，可以选择多个。
- 允许标头：Access-Control-Allow-Headers，可以设置多个，逗号分割，允许“\*”通配符。
- 公开标头：Access-Control-Expose-Headers：可以设置多个，逗号分割，不允许“\*”通配符。
- 有效期：Access-Control-Max-Age，最大的浏览器缓存时间，单位：秒。
- 允许凭证：Access-Control-Allow-Credentials，是否允许发送 Cookie。

#### 跨域请求

跨域的 API 请求要添加 X-CORS- $\{\text{appId}\}-\{\text{workspaceId}\}$  请求头。当预检请求到达网关后，网关解析 Access-Control-Request-Headers 中的 X-CORS- $\{\text{appId}\}-\{\text{workspaceId}\}$  获取 appId 和 workspaceId，再进一步获取对应的 CORS 配置。网关跨域请求的请求头中要包含如下内容：

- X-CORS- $\{\text{AppId}\}-\{\text{WorkspaceId}\}$ ：一定要有此请求头，并用实际的 AppId 和 WorkspaceId 替换占位符内容；
- Operation-Type
- WorkspaceId
- AppId
- Content-Type
- Version

```
$ajax({
  url: 'http:// $\{\text{mpaasgw\_host}\}$ /mgw.htm', // 请填写网关地址
  headers: {
    'X-CORS- $\{\text{appId}\}-\{\text{workspaceId}\}$ ': '1' // 一定要设置这个请求头
    'Operation-Type':  $\{\text{operationType}\}$ , // 请填写 operationType
    'AppId':  $\{\text{appId}\}$ , // 请填写 appId
    'WorkspaceId':  $\{\text{workspaceId}\}$ , // 请填写 workspaceId
    'Content-Type': 'application/json',
    'Version': '2.0',
  },
  type: 'POST',
  dataType: 'json',
  data: JSON.stringify(reqData),
  success: function(data){}
});
```

#### ② 说明

CORS 配置中的 允许标头 配置，根据实际情况添加或者设置“\*”。

## 7.7. 数据模型

作为业务人员，您可以将 API 服务的请求与响应定义成数据模型，通过模型复用减少繁琐的参数设置。该功能用在定义 HTTP 类型的 API 服务的参数，其他类型的 API 服务无需手动定义。

### 关于此任务

目前支持以下数据模型定义方式：

- 可视化编辑：逐条添加模型参数。
- 样本数据编辑：从样本数据解析出数据模型，推荐使用这种方式。

### 操作步骤

完成以下步骤配置数据模型：

1. 在移动网关主页，选择 **数据模型** 选项卡，进入数据模型列表页。
2. 点击 **添加数据模型** 按钮或者列表中的 **详情** 链接，添加或编辑数据模型定义：
  - **模型名称**：模型名称，以字母或下划线开头，由字母、下划线、数字组成。
  - **模型描述**：模型的描述信息。
  - **模型参数**：
    - **名称**：必填，模型中参数的名称。
    - **类型**：必填，可选择的类型有 `String`、`Int`、`Long`、`Float`、`Double`、`Boolean`、`List` 及已定义的数据模型。
    - **默认值**：选填，参数的默认值。
    - **描述**：选填，参数描述。

② 说明  
目前尚不支持 Map 类型。

3. 点击 **提交** 保存修改内容。

## 7.8. API 分析

在移动网关主页，选择 **API 分析** 选项卡进入 API 分析界面。API 分析界面提供以下几种 API 的统计数据：

- API 调用量：统计某一 API 分钟级的调用量。
- API 报错量：统计某一 API 分钟级的报错量。
- 平均调用耗时：统计某一 API 分钟级的平均调用耗时。
- API 调用同比：统计 API 当前时刻调用量相对昨天的同比。
- API 报错同比：统计 API 当前时刻报错量相对昨天的同比。
- API 耗时同比：统计 API 当前时刻平均耗时相对昨天的同比。

## 7.9. HarmonyOS NEXT 控制台使用 (beta)

本文介绍了如何在控制台生成鸿蒙客户端代码。

### 关于此任务

只有 HTTP 类型的 API 支持单个鸿蒙代码生成。

### 操作步骤

1. 通过以下的操作方式打开 **客户端代码生成** 窗口：
  - 方式一：在 **API 分组** 标签页中，单击系统列表操作列中的 **生成代码** 按钮。
  - 方式二：在 **API 管理** 标签页中，单击列表上方的 **生成代码** 按钮。
  - 方式三：在 **API 管理** 标签页中，单击列表操作列中的 **更多 > 生成代码** 按钮。该方式用于生成单个 API 代码（非 API 分组），仅适用于 HTTP 类型。

客户端代码生成 ×

API类型:  Native API  H5 API

Platform:  Android  iOS  Harmony  JS

PackageName:

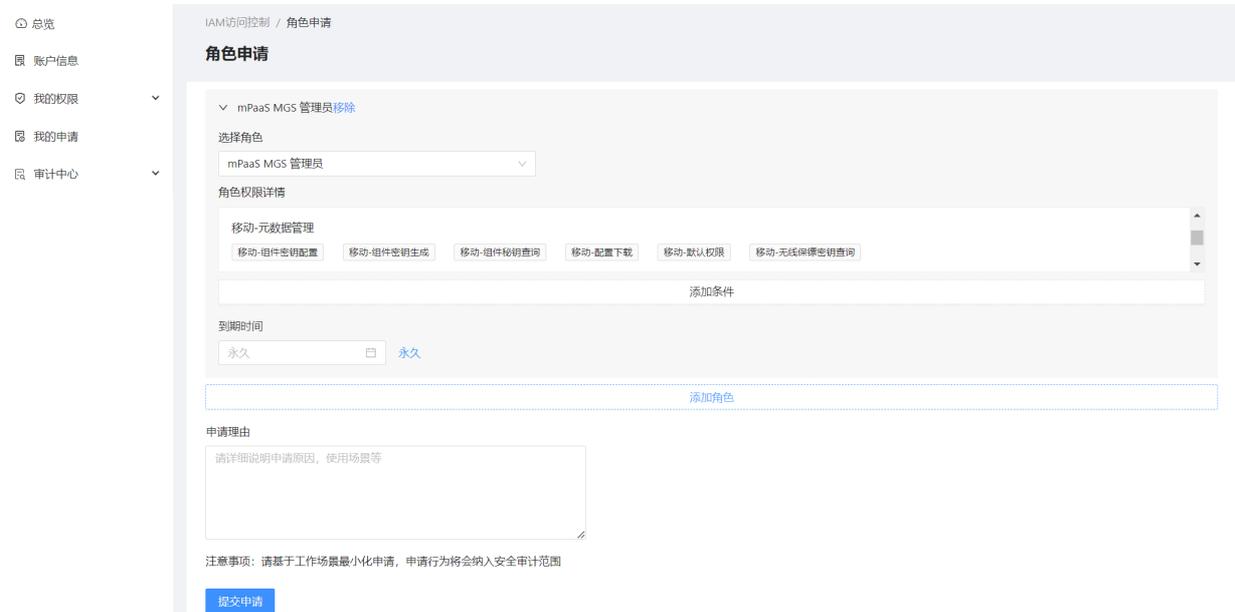
2. 在 **客户端代码生成** 窗口，完成以下信息配置：
  - **API 分组**：选择需要生成 SDK 的 API 分组。
  - **Platform**：选择 Harmony。
  - **PackageName**：填写客户端代码的包名；若不填，默认为 `com.client.service`。
3. 单击 **提交** 生成 API SDK 供鸿蒙客户端调用。

## 7.10. 管理员操作

管理员与普通用户主要是通过角色和权限来区分。系统部署初始有可以配置系统管理，系统管理员登录IAM系统，可以配置相关产品角色和操作权限。普通用户也可以通过主动申请，审批后获得相关权限。

### 1、管理员角色创建

填写角色名称》勾选角色权限，



角色权限请根据实际需要勾选：



### 2、普通用户申请角色

普通用户从mPaaS产品管控台进入IAM，进行角色申请，申请审批同意后，将会获得相关权限，包含功能权限或者数据权限。如下图观察员角色，就是只能查看，无法操作的权限。



## 8. 网关异常排查

### 单请求问题排查

#### 1. 客户端请求抓包

客户端抓包一般采用 Charles (推荐) 或 Fiddler 工具, 通过抓包工具, 可以看到 RPC 请求的一些关键数据。

下面的是一个抓包的案例:

- 请求 Header 样例:

```
POST /mgs/mgw.htm HTTP/1.1
Host:
AppId: 910143 AppId
Cookie: _SESSIONID=0A01E89E58541077C1710E980F07D2D974E6548800;_NRF=6CC07DC9C28B1B66AAB87
DId: W5a2rRADEtoDAJgw5zOU8Uq 设备 Id
User-Agent: 7 CFNetwork/893.14.2 Darwin/17.3.0
T: Mtu7GR 签名时间戳
tk: SrwJ5yE0mKz2CIEke9Hb7TjyJ19Kpt2wTrREK5pC3g451210
nbappid: 60000003
UniformGateway: https:// /mgs/mgw.htm
Content-Length: 285
WorkspaceId: product WorkspaceId
Sign: 4c49624c8fb776ec7fa7e51c49891a46 签名
Platform: IOS 平台
Operation-Type: com.queryOrder RPC 接口名
Connection: keep-alive
Accept-Language: zh-cn
x-mgs-encryption: 1 RPC 数据自加密标识
Accept: */*
Content-Type: application/json RPC 数据序列化格式
Accept-Encoding: br, gzip, deflate
nbversion: 1.1.1.0
```

- 响应 Header 样例:

```
HTTP/1.1 200 OK
Date: Thu, 21 Dec 2017 08:10:15 GMT
Server: openresty/1.11.2.1
Content-Type: text/plain; charset=UTF-8
Mgw-TraceId: 0a017714151384381574937071794 MGS traceId
Cache-Control: no-cache
Tips: %E6%93%8D%E4%BD%9C%E6%88%90%E5%8A%9F%E3%80%B2 RPC 调用结果文案
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Encoding: gzip
Result-Status: 1000 RPC 调用结果码
X-Powered-By: Servlet/2.5; JBoss-5.0./JBossWeb-2.1
X-Via: 1.1 dxin41:6 (Cdn Cache Server V2.0)
X-Cdn-Src-Port: 60857
Transfer-Encoding: chunked
Connection: Keep-alive
```

#### 2. 根据 TraceId 查询 MGS 日志 (仅适用于专有云)

- 从响应 Header 中获取 Mgw-TraceId。
- 进入 mPaaS 控制台的 移动网关 > 网关管理 > 常用工具 > 链路分析 页面, 输入 TraceId 可以解析出处理该请求所在的 MGS 服务器 IP 和处理时间。
- 通过 SSH 到 MGS 的服务器, 根据 TraceId 查询请求相关的日志。

```
ssh -p2022 account@IP account/password
cd /home/admin/logs/gateway
grep #traceid# *.log
```

- 根据 网关日志说明 和 网关结果码说明 分析日志。

#### 集群 GREP 问题排查 (仅适用于专有云)

有些时候, 您可能需要在 MGS 集群搜索某个日志。这时候, 可以使用开源的 pssh 工具。

- 下载 pssh 工具。
- 从 Gamma 平台导出 MGS 所有服务器 IP 列表到 mgs\_host.txt 文件中, 如下:

```
log@10.2.216.33:2022
log@10.2.216.26:2022
log@10.2.216.25:2022
```

- 运行以下执行命令:

```
pssh -i -h mgs_host.txt -A -P 'grep "xxxx" /home/admin/logs/gateway/xxx.log'
```

## 9. 常见问题

本文列举了移动网关使用过程中常见的一些问题以及问题排查方法。

### 调用失败，如何排查？

参见 [网关异常排查](#)。

### API 返回的错误码是什么意思？

参见 [网关结果码说明](#)。

### 如果引用了 okhttp，存在 okio 和 mPaaS 的冲突该怎么解决？

您需要完成以下步骤以解决该冲突：

1. 注释掉 mPaaS 的 wire 组件。

```
mpaascomponents{
    excludeDependencies=['com.alipay.android.phone.thirdparty:wire-build']
}
```

2. 使用公网提供的 wire 组件。

```
implementation 'com.squareup.wire:wire-lite-runtime:1.5.3.4@jar'
```

### 通过 JSAPI 调用 MGS RPC 接口向后端发送 POST 请求时，如何把参数放到 POST BODY 中？

MGS 正确配置好 POST BODY 及对应的数据模型后，通过 JSAPI 发送请求时需要把 POST BODY 的内容作为 `_requestBody` 的值放在 `requestData` 参数中，参见下面的样例：

```
window.onload = function() {
    ready(function() {
        window.AlipayJSBridge.call('rpc', {
            operationType: 'MYAPI',
            requestData: {
                ["_requestBody"]: '{"key1\\":\\"value1\\",\\"key2\\":\\"value2\\"}',
                headers: {},
                getResponse: true
            }, function(data) {
                alert(JSON.stringify(data));
            });
        });
    });
}
```

## 10. 参考

### 10.1. 网关结果码说明

本文对使用移动网关过程中出现的结果码进行说明，方便您进行问题排查。

#### 网关侧结果码

- 1000 为 API 调用成功，其他都是失败的错误码。
- 1001-5999、7XXX 为网关错误。
  - 其中，7XXX 表示无线保鉴验签或解密报错，具体参见 [无线保鉴结果码说明](#) 进行排查。
  - 除结果码外，您还可以查看响应 Header 中的 `Memo` 和 `tips` 字段，以了解更多错误信息。
  - 专有云用户还可以通过网关服务器上的 `~/logs/gateway/gateway-error.log` 日志查看详细错误信息。
- 当发生异常时，可以尝试通过网关异常排查进行排查。欲了解具体信息，参见 [网关异常排查](#)。

结果码	描述	解释
1000	处理成功	网关 API 调用处理成功。
1001	拒绝访问	Mock 格式错误，缺少 <code>resultStatus</code> ，WAF 校验失败，或者鉴权接口用户无权访问。
1002	调用次数超过限额	开启 <a href="#">限流配置</a> 后，当触发限流时会导致该异常。
1005	未授权	开启 <a href="#">API 授权</a> 后，API 调用时授权校验失败。
2000	登录超时	开启授权校验功能，非登录状态会触发该异常。
3000	RPC 接口不存在或关闭	在当前 <code>workspaceId</code> 对应的环境下， <code>appId</code> 对应的移动应用没有配置该 <code>operationType</code> 的 API 服务，或者该 API 服务不处于 <a href="#">开放</a> 状态。
3001	请求数据为空	客户端请求数据中的 <code>requestData</code> 为空。请检查客户端 RPC 是否正常，iOS 端需确认已 <a href="#">初始化网关服务</a> 。
3002	数据格式有误	RPC 请求格式有问题。专有云用户可以在服务端日志 <code>gateway-error.log</code> 中查看详细信息。
3003	数据解密失败	数据解密失败。
4001	服务请求超时	MGS 调用业务系统服务超时。后端业务系统负载过高导致，需排查后端系统的运行情况。若超时设置不合理，可以适当调整。注：默认超时时间为 3s。
4002	远程调用业务系统异常	MGS 调用业务系统服务出现异常。专有云用户可以在服务端日志 <code>gateway-error.log</code> 中查看详细信息。
4003	API 分组 HOST 异常	MGS 调用 HTTP 业务系统服务出现 <code>UnknownHostException</code> 异常。请检查 API 分组配置的域名是否存在。
5000	未知异常	其他严重错误。专有云用户可以在服务端日志 <code>gateway-error.log</code> 中查看详细信息。
7000	没有设置公钥	移动 APP 中无线保鉴中无 <code>appId</code> 对应的密钥或者网关无法获取 <code>appId</code> 对应的签名密钥。
7001	验签的参数不够	网关服务端验证签名不通过。
7002	验签失败	网关服务端验证签名不通过。
7003	验签-时效性失败	API 请求入参 <code>ts</code> 时间戳超过系统设置的时间有效性。需要检查客户端时间是否为系统时间。
7007	验签-缺少 <code>ts</code> 参数	API 请求缺少验签 <code>ts</code> 参数。
7014	验签-缺少 <code>sign</code> 参数	API 请求缺少验签 <code>sign</code> 参数。一般情况下是客户端签名数据失败，导致缺失 <code>sign</code> 参数。请检查客户端无线保鉴图片是否正确。
8002	跨域预检请求 (CORS preflight)	跨域预检请求。

#### 业务侧结果码

以下结果码可在业务系统服务器内部查看错误信息。

通过查看各个业务系统上的 `~/logs/mobileservice/monitor.log` 日志可确定异常具体信息。

结果码	适用协议	描述	解释
6000	MPC、DUBBO	RPC-目标服务找不到	发布的服务 (service) 无法找到，服务器无法访问或者服务已迁移。
6001	MPC、DUBBO	RPC-目标方法找不到	发布的该 service 内的方法无法找到。
6002	MPC、DUBBO	RPC-参数数目不正确	传入的参数个数，与声明的参数个数不相等。
6003	MPC、DUBBO	RPC-目标方法不可访问	目标方法不能被调用。
6004	HTTP、MPC、DUBBO	RPC-JSON 解析异常	HTTP：将 RPC 参数转换为后端 HTTP 请求参数时发生异常。MPC/DUBBO：将 RPC JSON 数据反序列化化为业务参数对象时失败。
6005	MPC、DUBBO	RPC-调用目标方法时参数不合法	反射调用时，参数不合法。
6007	MPC、DUBBO	RPC-验证登录服务不可用	没有实现 SPI 包中验证登录接口或者验证登录接口配置出错。
6666	HTTP、MPC、DUBBO	RPC-业务抛出异常	HTTP：后端系统返回 HTTP status code 不等于 200。MPC/DUBBO：业务方抛出的异常。RPC 无法处理，统一为业务异常。

### Android 客户端结果码

结果码	描述	提示文案
0	未知错误	未知错误，请稍后再试
1	客户端找不到通讯对象，没有设置 Transport	网络出错，请稍后再试
2	客户端没有网络，如用户关闭了网络或者禁止了应用的网络权限	网络无法连接
3	SSL相关错误，包括 SSL 握手错误，SSL 证书错误	客户端证书有误，请检查手机的时间设置是否准确。
4	客户端网络连接超时，TCP 建连超时，目前超时时间为 10s	网络欠佳
5	客户端网络速度过慢，数据读写超时，socketTimeout 的场景	网络欠佳
6	客户端请求服务端无响应，NoHttpResponseException	网络出错，请稍后再试
7	客户端网络 IO 错误，对应 IOException	网络出错，请稍后再试
8	客户端网络请求调度错误，执行线程中断异常	网络出错，请稍后再试
9	客户端处理错误，包括序列化错误、注解处理错误、线程执行错误	网络出错，请稍后再试
10	客户端数据反序列化错误，服务端数据格式有误	网络出错，请稍后再试
13	请求中断错误，例如线程中断时网络请求会被中断	网络出错，请稍后再试
15	客户端网络授权错误，HttpHostConnectException，Connection to xxx refused，无网络或者对应服务器拒绝连接	网络无法连接
16	DNS 解析错误	网络无法连接，请稍后再试
18	网络限流，客户端限流，当客户端请求流量超过阈值后会被限制网络请求	网络限流，请稍后再试
code ≥ 400 和 code < 500	HTTP 响应码为 4xx	网络无法连接
400 > code ≥ 100 和 500 < code < 600	HTTP 非成功的响应码	网络无法连接，请稍后再试

## 10.2. 无线保镖结果码说明

本文所列的无线保镖错误码同时适用于 Android 及 iOS 操作系统。

根据不同错误类型，下面将错误码分为以下三类：

- [一般错误码](#)
- [静态数据加解密错误码](#)
- [安全签名接口错误码](#)

如果发生错误，Xcode 的 console 中会打印“SG ERROR: xxxx”格式的错误码，具体含义如下所示。

### 一般错误码

错误码	含义
101	参数不正确，请检查输入的参数。
102	主插件初始化失败。
103	有依赖的插件没有引入。本错误码打印的同时会提示缺失的插件名，请按提示进行操作。
104	引入了插件，但加载失败。一般是因为 other linker flags 中没有添加 <code>-all_load</code> 或 <code>-ObjC</code> 所致，添加后可以解决。
105	引入了对应插件，但该插件的依赖插件没有引入。本错误码打印的同时会提示缺失的插件名，请按提示进行操作。
106	引入了对应插件，但该插件的依赖插件版本不符合要求。本错误码打印的同时会提示依赖的版本号，请按提示进行操作。
107	引入了对应插件，但该插件的版本不符合要求。
108	引入了对应插件，但该插件的依赖资源没有被引入。
109	引入了对应插件，但该插件的依赖资源版本不符合要求。
121	图片文件有问题。一般是生成图片文件时的 bundle id 和当前应用的 bundle id 不一致。
122	没有找到图片文件，请确保图片文件在项目目录下。
123	图片文件格式有问题，请重新生成图片文件。
124	当前图片的版本太低。
125	init with authcode 初始化错误。
199	未知错误，请重试。
201	参数不正确，请检查输入的参数。
202	图片文件有问题。一般是生成图片文件时的 bundle id 和当前应用的 bundle id 不一致。
203	没有找到图片文件，请确保图片文件在项目目录下。
204	图片文件格式有问题，请重新生成图片文件。
205	图片文件的内容不正确，请重新生成图片文件。
206	参数中的 key 在图片文件中找不到，请确认图片文件中有这个 key。
207	输入的 key 非法。
208	内存不足，请重试。
209	不存在指定索引的 key。
212	请升级新版本图片，当前图片的版本太低。

299	未知错误，请重试。
-----	-----------

### 静态数据加解密错误码

错误码	含义
301	参数不正确，请检查输入的参数。
302	图片文件有问题。一般是获取图片文件时的 apk 签名和当前程序的 apk 签名不一致。请使用当前程序的 apk 重新生成图片。
303	没有找到图片文件，请确保图片文件在 <code>res\drawable</code> 目录下。
304	图片文件格式有问题，请重新生成图片文件。一种常见场景就是二方和三方图片混用。二方和三方的图片不兼容，需要各自生成。
305	图片文件内的内容不正确，请重新生成图片文件。
306	参数中的 key 在图片文件中找不到，请确认图片文件中有这个 key。
307	输入的 key 非法。
308	内存不足，请重试。
309	不存在指定索引的 key。
310	待解密数据不是可解密数据。
311	待解密数据与密钥不匹配。
312	当前图片的版本太低，请升级新版本的图片。
399	未知错误，请重试。
401	参数不正确，请检查输入的参数。
402	内存不足，请重试。
403	获取系统属性失败，请确认是否有软件拦截，获取系统参数。
404	获取图片文件的密钥失败，请确认图片文件的格式和内容是否正确。
405	获取动态加密密钥失败，请重试。
406	待解密数据格式不符合解密要求。
407	待解密数据不符合解密要求，请确认该数据是本设备上保镳动态加密产生。
499	未知错误，请重试。
501	参数不正确，请检查输入的参数。
502	内存不足，请重试。
503	获取系统属性失败，请确认是否有软件拦截，获取系统参数。
504	获取图片文件的密钥失败，请确认图片文件的格式和内容是否正确。
505	获取动态加密密钥失败，请重试。
506	待解密数据不是可解密数据。
507	待解密数据与密钥不匹配，请重试。

508	传入 key 对应的 value 不存在。
599	未知错误，请重试。

### 安全签名接口错误码

错误码	含义
601	参数不正确，请检查输入的参数。
602	内存不足，请重试。
606	使用带 seedkey 的 top 签名时，没有找到 seedkey 对应的 seedsecret。
607	<code>yw_1222.jpg</code> 图片文件有问题。一般是生成图片时的 bundle id 和应用的 bundle id 不匹配。
608	没有找到 <code>yw_1222.jpg</code> 图片文件，请确保图片文件在项目目录下。若工程中已存在此图片，请确认工程 <code>meta.config</code> 文件中 <code>base64Code</code> 字段不为空。若为空，请参考 <a href="#">无线保鉴</a> 重新手动生成 <code>yw_1222.jpg</code> 图片。
609	<code>yw_1222.jpg</code> 图片文件格式有问题，请重新生成图片文件。一种常见场景就是二方和三方图片混用。二方和三方的图片不兼容，需要各自生成。
610	<code>yw_1222.jpg</code> 图片文件内的内容不正确，请重新生成图片文件。
611	参数中的 key 在图片文件中找不到，请确认图片文件中有这个 key。
615	当前图片的版本太低，请升级新版本的图片。
699	未知错误，请重试。

## 10.3. 网关日志说明

### 10.3.1. 网关服务端日志

本文包含对各类服务端日志的说明。

#### 🔔 重要

只有专有云用户才有权限查看服务端日志。

#### API 摘要日志

日志路径：`~/logs/gateway/gateway-page-digest.log`

- 日志打印时间
- 请求地址
- 响应
- 结果 (Y/N)
- 耗时：单位 ms
- operationType
- 系统名
- appId
- workspaceId
- 结果码
- 客户端 productId
- 客户端 productVersion
- 渠道
- 用户 ID
- 设备 ID
- UUID
- 客户端 trackId
- 客户端 IP
- 网络协议：HTTP 或 HTTP2
- 数据协议：JSON 或 PB
- 请求数据大小：字节
- 响应数据大小：字节

- 压测标识
- TraceId：请求的唯一标识，可以将所有的摘要日志、详细日志或者异常日志串起来
- cpt 标
- 客户端系统类型
- 后端系统耗时
- clientIp 类型：4 或 6
- RPC 协议版本：1.0 或 2.0

格式：

```
时间 - (请求地址, 响应, 结果 (Y/N), 耗时, operationType, 系统名, appId, workspaceId, 结果码, 客户端productId, 客户端productVersion, 渠道, 用户ID, 设备ID, UUID, 客户端trackId, 客户端IP, 网络协议, 数据协议, 请求数据大小, 响应数据大小, 是否压测, TraceId, 是否组件API, 客户端系统类型, 后端系统耗时, IP协议版本, RPC协议版本)
```

样例：

```
2020-06-03 14:14:08,001 - (/mgw.htm, response, Y, 61ms, alipay.mcdp.space.initSpaceInfo, -, 84EFA9A281942, default, 1000, -, -, -, Wz4Zak5peDgDAGRnW5rFFGhT, Wz4Zak5peDgDAGRnW5rFFGhTn9uqCLa, Wz4Zak5peDgDAGRnW5rFFGhTn9uqCLa, 223.104.XXX.XXX, HTTP, JSON, 2, 2406, F, 0aid766715911648479408
```

## API 详细日志

日志路径：`~/logs/gateway/gateway-page-detail.log`

详细日志分为以下类别：

- 请求日志：`[request]`
- 响应日志：`[response]`

### 请求日志

- 日志打印时间
- 客户端 IP
- TraceId
- 日志级别
- 日志类型：request
- operationType
- appld
- workspaceId
- requestData
- sessionId
- did：设备 ID
- contentType
- mmtp：T 或者 F，是否使用 MMTP 协议
- async：是否异步调用，T 或者 F

### 响应日志

- 日志打印时间
- 客户端 IP
- TraceId
- 日志级别
- 日志类型：response
- operationType
- appld
- workspaceId
- responseData
- resultStatus：结果码
- contentType
- sessionId
- did：设备 ID
- mmtp：T 或者 F，是否使用 MMTP 协议
- async：是否异步调用，T 或者 F

样例：

```
2017-12-21 15:37:10,208 [100.97.90.113] [79c731d51513841830208829314258] INFO -
[request]operationType=com.alipay.gateway.test,appId=2A9ADA1045,workspaceId=antcloud,requestData=***,sessionId=
,did=WjtkmWeIuHsDADl7BEleyK2L,contentType=JSON,mmtp=F,async=T

2017-12-21 15:37:10,229 [] [79c731d51513841830208829314258] INFO -
[response]operationType=com.alipay.gateway.test,appId=2A9ADA1045,workspaceId=antcloud,responseData=***,resultStatus=1000,contentType=JSON,session
,did=WjtkmWeIuHsDADl7BEleyK2L,mmtp=F,async=T
```

## API 统计日志

日志路径：`~/logs/gateway/gateway-page-stat-s.log`

- 日志打印时间
- operationType
- appld

- workspaceId
- 结果：Y/N
- 结果码
- 压测标识
- 请求总量
- 请求总耗时：ms

格式：

```
时间 - operationType,appId,workspaceId,结果 (Y/N),结果码,压测标识 (T/F),请求总量,请求总耗时 (ms)
```

样例：

```
2017-12-21 15:34:58,419 - com.alipay.gateway.test,2A9ADA1045,antcloud,Y,1000,F,1,3
```

### 网关线程统计日志

日志路径：`~/logs/gateway/gateway-threadpool.log`

- 日志打印时间
- 线程名
- 活动线程数
- 当前线程池的线程数
- 创建过的最大线程数量
- 核心线程数
- 最大线程数
- 任务队列容量
- 剩余队列容量

格式：

```
时间 [线程名,ActiveCount,PoolSize,LargestPoolSize,CorePoolSize,MaximumPoolSize,QueueSize,QueueRemainingCapacity]
```

样例：

```
2017-12-21 16:33:32,617 [gateway-executor,0,80,80,80,400,0,1000]
```

### 网关配置日志

日志路径：`~/logs/gateway/gateway-config.log`

此日志记录网关配置变更的相关通知。

### 网关默认日志

日志路径：`~/logs/gateway/gateway-default.log`

网关默认日志，未指定特定日志的埋点都会打到此日志。

### 网关错误日志

日志路径：`~/logs/gateway/gateway-error.log`

此日志记录错误和异常堆栈。

## 10.3.2. 网关 SPI 日志

此部分日志说明，只针对集成了 mpaasgw-spi-mpc 或 mpaasgw-spi-dubbo 的业务系统。

### API 摘要日志

日志路径：`~/logs/mobileservice/page-digest.log`

- 日志打印时间
- operationType
- 客户端 productId
- 客户端 productVersion
- 耗时：单位 ms
- 结果 (Y/N)
- 结果码
- uniqueId

格式：

```
时间 - (operationType,productId,productVersion,耗时,结果 (Y/N),结果码,uniqueId)
```

样例：

```
2017-09-12 11:15:57,700 - (com.alipay.gateway.test,ANT_CLOUD_APP,3.0.0.20171214,36ms,Y,1000,79c731d5150518615768657974443)
```

### SPI 启动日志

日志路径：`~/logs/mobileservice/boot.log`

启动日志记录业务系统 mobileservice 的注册和启动情况，分为如下六个阶段：

- Start-To-Register-Service：开始解析 API 服务接口。
- Start-To-Analyze-Method：开始解析 API 服务接口中的方法。
- Analyze-Method-Parameter：解析方法参数。
- Method-Info：方法信息。
- Registered-OperationType：完成单个 API operationType 注册。
- Register-Service-Success：完成此接口中所有的 API operationType 注册。

此日志可以帮助排查 operationType 是否注册成功。

样例：

```
2017-12-20 11:25:59,746 [Start-To-Register-Service] target: com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpcImpl@5b490d5e, interface: interface com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpc

2017-12-20 11:25:59,771 [Start-To-Analyze-Method] method=mock

2017-12-20 11:25:59,780 [Analyze-Method-Parameter] parameters=["s"]

2017-12-20 11:25:59,839 [Method-Info] MethodInfo[paramCount=1,paramType={class com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpc$Req},paramNames={s},returnType=class com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpc$Resp,target=com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpcImpl@5b490d5e,method=public abstract com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpc$Resp com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpc.mock(com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpc$Req),interfaceClass=interface com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpc]

2017-12-20 11:25:59,839 [Registered-OperationType] operationType=com.alipay.sofa.mock

2017-12-20 11:25:59,840 [Register-Service-Success] target=com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpcImpl@5b490d5e, interface=interface com.alibaba.mpaasgw.biz.shared.rpcTest.MockRpc
```

### API 监控日志

日志路径：`~/logs/mobileservice/monitor.log`

记录 API 请求的详细日志，包括 API 请求的 debug 日志以及出错情况下的异常堆栈。

### SPI 默认日志

日志路径：`~/logs/mobileservice/common-default.log`

SPI 默认日志，未指定特定日志的埋点都会打到此日志。

### SPI 错误日志

日志路径：`~/logs/mobileservice/common-error.log`

记录错误和异常堆栈。

## 10.4. 业务接口定义规范

考虑到手机端开发环境的限制（尤其是 iOS 系统），以及保持接口定义的简单，服务端在定义移动服务接口时，不能使用 Java 语法的全集。

接口定义规范涉及三类定义：

- **内部支持数据类规范**：已支持的 Java 原生类和包装类。
- **用户接口类规范**：用户定义的 interface，包含 API 调用的 method 声明。
- **用户定义实体类规范**：用户定义的实体 class（包含 field 声明），接口类中 method 参数或返回值、其它用户定义实体类将会引用到。

### 内部支持数据类规范

#### 不支持的数据类型

- 容器类型不能多层嵌套。
- List 或 Map 必须有泛型信息。
- List 或 Map 的泛型信息不能是 array 类型
- 不支持单字节（字节数据 byte [] 是支持的）
- 不支持对象数组，请用 list 代替。
- 属性名不能是 data 和 description，会与 iOS 的属性冲突。
- Map 类型的 key 必须是 String 类型。
- 类型不能是抽象类。
- 类型不能是接口类。

错误的写法：

```
public class Req {
    private Map<String,List<Person>> map; //容器类型不能多层嵌套。
    private List<Map<Person>> list; //容器类型不能多层嵌套。
    private List list1; //List 或 Map 必须有泛型信息。
    private Map map1; //List 或 Map 必须有泛型信息。
    private List<Person[]> listArray; //List 或 Map 的泛型信息不能是 Array 类型。
    private byte b; //不能为单字节
    private Person[] personArray; //不支持对象数组，请用 List 代替
    private String description; //属性名不能为 description
}
```

### 支持的数据类型

```
boolean, char, double, float, int, long, short
java.lang.Boolean
java.lang.Character
java.lang.Double
java.lang.Float
java.lang.Integer
java.lang.Long
java.lang.Short
java.lang.String
java.util.List, 但：必须使用类型参数；不能使用其具体子类 以下简称 List
java.util.Map, 但：必须使用类型参数；不能使用其具体子类；key 类型必须是 String 以下简称 Map
Enum
byte[]
```

#### 正确的写法：

```
public class Req {
    private String s = "ss";
    private int i;
    private double d;
    private Long l;
    private long ll;
    private boolean b;
    private List<String> stringList;
    private List<Person> personList;
    private Map<String,Person> map;
    private byte[] bytes;
    private EnumType type;
}

public class Person {
    private String name;
    private int age;
}
```

## 用户接口类规范

### method 的参数

#### 不可以引用：

- 枚举类型
- 除上文提到的 Map、List、Set 之外的泛型
- 抽象类
- 接口类
- 原生类型的数组

#### 可以引用：

- 具体的实体类，要求引用类型与实际的对象类型保持一致；不可使用父类引用类型指向子类对象。
- 内部支持数据类，但数组、Map、List、Set 这些集合类型不可以嵌套。

#### 如下是错误示例：

```
Map<String,String[]>
Map<String,List<Person>>(Person为一个具体的实体类)
List<Map<String,Person>>
List<Persion[]>
```

### method 的返回值

#### 不可以引用：

- 枚举类型
- 除上文提到的 Map、List、Set 之外的泛型
- 抽象类
- 接口类
- 原生类型的数组

#### 可以引用：

- 具体的数据类，要求引用类型与实际的对象类型保持一致；不可使用父类引用类型指向子类对象，例如，不能用 Object 引用指向其它对象。

#### 📌 重要

如果父类为具体类，生成工具将检查不出此类错误。

- 内部支持的数据类见文章开头定义。数组、Map、List、Set 集合类型不可以嵌套，参见上文相关示例。

### method 的定义

- 使用 `@OperationType` 注解，未加此注解的方法将被生成工具忽略。
- method 不可 overloading。

### 代码生成工具限制

- 允许接口类定义的继承关系，但会合并层次关系。
- 允许但忽略接口类中定义变量。

- 允许但忽略接口类中的方法声明抛出异常。
- 一个源文件中只能包含一个接口类的定义，不能包含其它类（内部类、匿名类等）的定义。
- 接口类定义本身和其引用到的类型必须为内部支持数据类或可以从源码获得定义。

### 用户定义实体类规范

#### field 定义

不可以引用：

- 枚举类型
- 除上文提到的 Map、List、Set 之外的泛型
- 抽象类
- 接口类
- 原生类型的数组

可以引用：

- 具体的实体类，要求引用类型与实际的对象类型保持一致；不可使用父类引用类型指向子类对象。
- 内部支持数据类。数组、Map、List、Set 集合类型不可以嵌套，参见上文相关示例。
- 修饰符包括 transient 的属性将会被忽略。
- final static int 定义的常量（其它不符合该要求定义的常量或静态变量将被忽略）。

② 说明

不推荐 `is` 开头的成员变量定义。

#### 类的定义

- 可以继承自其它实体类。
- 忽略其方法声明，生成工具将会自动根据实体类字段生成 setter/getter 方法。

#### 代码生成工具限制

- 属性的声明必须一行一个。
- 允许但忽略用户定义实体类实现的接口。
- 一个源文件中只能包含一个用户定义实体类的定义，不能包含其它类（内部类、匿名类等）的定义。
- 接口类定义本身和其引用到的类型必须为内部支持数据类或可以从源码获得定义。

## 10.5. 密钥生成方法

根据您的业务需求，查看生成密钥的方法。密钥包括 RSA 密钥、ECC 密钥、国密密钥。

### 前置条件

您已通过 [OpenSSL 官网](#) 下载并安装 OpenSSL 工具（1.1.1 或以上版本）。

### 生成 RSA 密钥

1. 打开 OpenSSL 工具，使用以下命令行生成 RSA 私钥。您可以选择生成 1024 或 2048 位的私钥。

```
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt rsa_keygen_bits:2048
```

2. 根据 RSA 私钥生成 RSA 公钥。

```
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

### 生成 ECC 密钥

1. 打开 OpenSSL 工具，使用以下命令行生成 ECC 的密钥对。您必须选择 secp256k1 椭圆曲线算法。

```
openssl ecparam -name secp256k1 -genkey -noout -out secp256k1-key.pem
```

2. 根据 `secp256k1-key.pem` 密钥对生成 ECC 公钥。

```
openssl ec -in secp256k1-key.pem -pubout -out ecpubkey.pem
```

### 生成国密密钥

1. 打开 OpenSSL 工具，使用以下命令行生成国密 SM2 私钥 `sm2-key.pem`。

```
openssl ecparam -name SM2 -genkey -noout -out sm2-key.pem
```

2. 根据 `sm2-key.pem` 密钥对生成国密 SM2 公钥 `sm2pubkey.pem`。

```
openssl ec -in sm2-key.pem -pubout -out sm2pubkey.pem
```

## 10.6. 网关签名机制

为保证客户端请求不被篡改和伪造，RPC 请求有签名机制，RPC 模块会自动实现加签功能。

基本的加签、验签过程如下：

1. 将 `requestBody` 中的内容转换为字符串。
2. 使用无线保镱安全模块，通过保存在加密图片（即无线保镱图片）中的加密密钥，对转化的字符串进行加签。
3. 将加密后的签名放在请求中发给网关。
4. 网关使用相同方式签名，校验两个签名是否相等。