ALIBABA CLOUD

阿里云

SOFAStack微服务

SOFARPC

文档版本：20220606

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，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。

3. 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。阿里云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，并在阿里云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阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

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**。
粗体	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 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.概述

SOFARPC 提供应用之间的点对点服务调用功能。

产品特性

- **高可用**

  SOFARPC 提供软件负载均衡的能力，它是对等服务调用的调度器，会帮助服务消费方在这些对等的服务提供方中合理地选择一个来执行相关的业务逻辑。

- **高容错**

  一切服务调用的容错机制均由软负载和配置中心控制，这样可以在应用系统无感知的情况下，帮助服务消费方正确选择健康的服务提供方，保障全站的稳定性。

基本功能

主要为用户提供下述功能：

- **多种服务路由方式**：包括软负载、硬负载、直连等。
- **负载均衡**：支持随机策略、考虑长连接、权重等负载均衡策略。
- **多种调用方式**：支持同步、单向、回调、泛化等多种调用方式。
- **多种编程界面**：支持 XML、动态客户端、Standalone 模式等多种编程界面。
- **流量转发**：支持应用之间的流量转发。
- **链路追踪**：支持网格外部应用调用网格内部应用并形成一个完整的链路追踪信息
- **链路数据透传**：支持应用调用上下文中存放数据，达到整个链路上的应用都可以操作该数据。
- **故障剔除**：目前支持 bolt 协议。它会自动监控 RPC 调用的情况。

协议支持

SOFARPC 支持不同的通讯协议，目前主要包括：

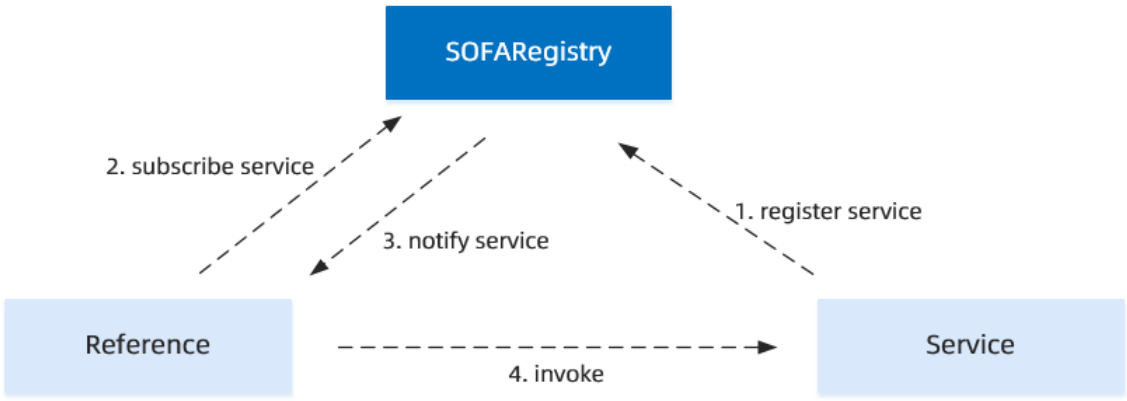
- **BOLT**：是蚂蚁金融科技服务集团开放的，基于 Netty 开发的网络通信框架。
- **RESTful**：基于 HTTP 一种设计框架。
- **Dubbo**：开源分布式服务框架
- **H2C**：开放的网络通信框架。

实现原理

SOFARPC 中的远程调用是通过服务模型来定义服务调用双方的。服务分为：

- **服务消费方**：对应 RPC 的调用端，可以理解为调用客户端，即 “引用 (Reference)” 。
- **服务提供方**：对应 RPC 的被调用端，可以理解为调用服务端。即 “服务 (Service)” 。

SOFARPC 实现原理示意图



上述原理说明如下：

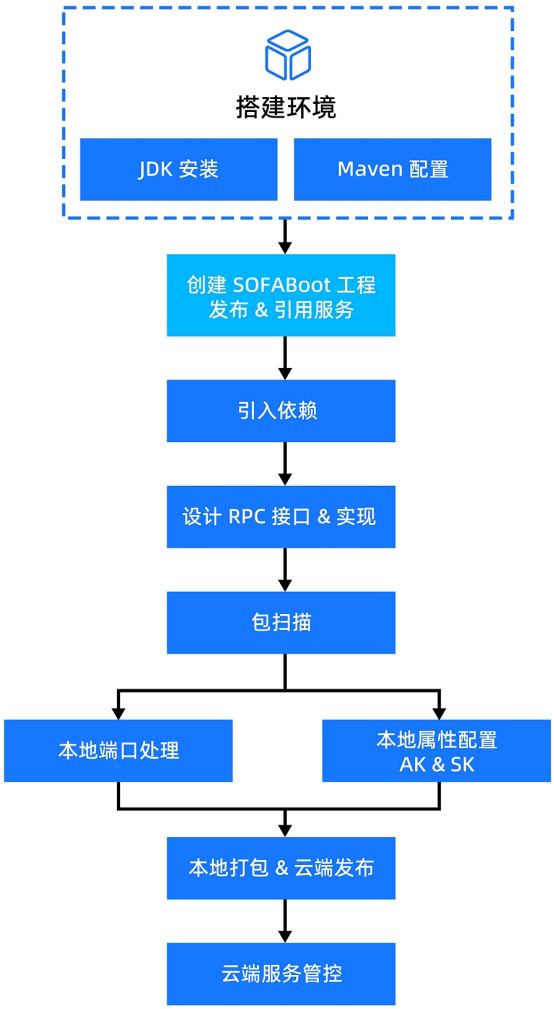
1. **register service**：当一个 SOFARPC 的应用启动的时候，如果发现当前应用需要发布 RPC 服务的话，那么 SOFARPC 会将这些服务注册到服务注册中心上，如上图中 Service 指向 Registry。
2. **subscribe service**：当引用这个服务的 SOFARPC 应用启动时，会从服务注册中心订阅到相应服务的元数据信息。
3. **notify address**：服务注册中心收到订阅请求后，会将发布方的元数据列表实时推送给服务引用方，如上图中 Registry 指向 Reference。
4. **invoke**：当服务引用方拿到地址以后，就可以从中选取地址发起调用了，如上图中 Reference 指向 Service。

2.SOFARPC 快速入门

微服务（SOFAStack MicroService）主要是通过 SOFARPC 来实现服务的发布和引用，微服务中的其它模块也都围绕 SOFARPC 展开。本文以微服务本地开发到云端发布的整体流程为框架，让您了解 SOFARPC 如何在本地实现、如何发布到云端、如何在云端进行服务管控。在云端发布前，示例工程也支持直连的方式，让您在本地体验 SOFARPC 的服务发布和服务引用。

本地工程开发

SOFARPC 工程开发流程图



SOFARPC 示例代码演示视频

准备工作

- 搭建 SOFABoot 环境。具体操作，请参见 搭建环境。

> 注意
>
> 暂时不支持非 SpringBoot、SOFABoot 应用接入。

- 通过以下任一方式生成 2 个 SOFABoot Web 工程，分别作为服务发布方和引用方。

- 创建 2 个 SOFABoot Web 工程。具体操作，请参见 创建工程。
- 直接下载 示例工程。下载后，请参考 版本说明 将工程根目录下 `pom.xml` 文件中的版本号修改为最新版本号。

本地开发流程

1. 引入依赖。

   按步骤创建的工程默认已经引入该依赖，请忽略此步骤；直接下载的示例工程需在 2 个本地 SOFABoot 工程 Web 模块 `pom.xml` 中引入 SOFARPC 的 Maven 依赖。

```
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>rpc-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

2. 编写业务逻辑。

   主要为服务发布和服务引用。本示例使用注解的方式配置 Bean，实现服务发布和引用。更多详情，请参见 使用注解方式。其它配置方式，请参见 使用 XML 配置 及 使用编程 API。

   - 服务发布的业务逻辑
     - 设计服务接口类

       本示例类名称为 `SampleService.java` ，接口路径为 `com.alipay.samples.rpc.SampleService` 。

```
public interface SampleService{
   public String hello();
}
```

     - 编写服务实现类

       本示例类名称为 `SampleServiceImpl.java` ，接口实现路径为 `com.alipay.samples.rpc.impl.SampleServiceImpl` 。

```
@Service
@SofaService(interfaceType =SampleService.class,bindings =@SofaServiceBinding(bindi
ngType ="bolt"))
public class SampleServiceImpl implements SampleService{
   private int count =0;

   @Override
   public String hello(){
      return "Hello SOFARpc! times = "+ count++;
   }
}
```

   - 服务引用的业务逻辑

- 引用对象注入

本示例采用 `ReferenceHolder` 类对引用对象进行统一管理，示例如下：

```
@Component
public class ReferenceHolder {
    @SofaReference(interfaceType = SampleService.class, binding = @SofaReferenceBin
ding(bindingType = "bolt", directUrl = "127.0.0.1:12201"))
    private SampleService sampleService;

    public SampleService getSampleService() {
        return sampleService;
    }

    public void setSampleService(SampleService sampleService) {
        this.sampleService = sampleService;
    }
}
```

> **说明**
>
> `directUrl="127.0.0.1:12201"` ：注意端口号 12201 需要和 `myserver-app` Web 模块 `src/main/resources/config/application.properties` 中的配置 `rpc.tr.port=12201` 对应。

- 引用对象

为了方便直观使用，本示例将引用服务逻辑放在 `myclient-app` 工程 Web 模块的 `com.alipay.mytestsofa.SOFABootWebSpringApplication` 中，示例如下：

```
@SpringBootApplication
public class SOFABootWebSpringApplication{
    private static final Logger logger =LoggerFactory.getLogger(SOFABootWebSpringAp
plication.class);

    public static void main(String[] args){
        //*************** 注意 ******************//
        //本地同时启动 myserver-app 和 myclient-app 时，由于 tomcat 端口冲突问题，需要修
改 myclient-app 的 端口号为 8084。
        //将 myserver-app 和 myclient-app 发布到云上环境时，由于默认健康检查端口是 8080，
所以需要注释掉该行代码。
        System.setProperty("server.port","8084");
        //由于本地启动没有注册中心，所以使用本地直连的方式访问本地启动的 myserver-app，发布
到线上的时候需要注释掉该行代码。
        System.setProperty("run.mode","TEST");
        //*******************************//

        SpringApplication springApplication =new SpringApplication(SOFABootWebSprin
gApplication.class);
        ApplicationContext applicationContext = springApplication.run(args);
```

```
        if(logger.isInfoEnabled()){
            printMsg("SofaRpc Application (myclient-app) started on 8084 port.");
        }

        //调用 SOFARPC 服务。
        ReferenceHolder referenceHolder = applicationContext.getBean(ReferenceHolde
r.class);
        final SampleService sampleService = referenceHolder.getSampleService();

        new Thread(newRunnable(){
            @Override
            public void run(){
                while(true){
                    try{
                        String response = sampleService.hello();
                        printMsg("Response from myserver-app: "+ response);
                    }catch(Exception e){
                        e.printStackTrace();
                    }finally{
                        try{
                            TimeUnit.SECONDS.sleep(3);
                        }catch(InterruptedException e){
                        }
                    }

                }

            }
        }).start();
    }
    private static void printMsg(String msg){
        System.out.println(msg);
        if(logger.isInfoEnabled()){
            logger.info(msg);
        }
    }
}
```

3. 配置包扫描。

```
@SpringBootApplication(scanBasePackages ={"com.alipay.mytestsofa","com.alipay.samples.r
pc"})
public class SOFABootWebSpringApplication{
...
}
```

4. 配置本地运行的端口。

   i. 在 `myserver-app` Web 模块 `application.properties` 文件中配置 `rpc.tr.port=12201` 。

   `rpc.tr.port` 是 TR 端口号，默认为 12200； `TR = TaobaoRemoting` 是 RPC 使用的底层通信框架。云端发布时不需要该项配置。

ii. 运行 Web 子模块中的 `SOFABootWebSpringApplication` 。

运行后，框架会自动进行服务的发布。为了避免端口冲突，需要在该类中指定端口。

```
System.setProperty("server.port","8083");
```

> **注意**
>
> 将 `myserver-app` 发布至云上环境前，必须注释掉代码中对 8083 端口的配置。

iii. 在 `myclient-app` Web 模块 `application.properties` 文件中配置 `rpc.tr.port=12202` 。

`rpc.tr.port` 是 TR 端口号，默认为 12200。云端发布时不需要该项配置。

5. 配置 `application.properties` 。

i. 登录 SOFAStack 控制台。

ii. 在左侧导航栏选择 **研发效能 > 脚手架**，然后在 **Step 2** 中获取如下信息：

- **实例标识**：应用实例在工作空间中的唯一标识。

  在 `application.properties` 中对应的 key 为 `com.alipay.instanceid` 。

- **AntVIP**：区域的唯一标识，每个区域一个地址。您可以在 SOFAStack 控制台左侧导航栏，单击 **中间件总览**，然后在 **元数据信息** 区域获取该信息。

  在 `application.properties` 中对应的 key 为 `com.antcloud.antvip.endpoint` 。

- **Access Key ID**：用于标识用户。单击 **获取 AK** 可前往 RAM 控制台获取。具体操作，请参见 创建 AccessKey。

  在 `application.properties` 中对应的 key 为 `com.antcloud.mw.access` 。

- **Access Key Secret**：用于验证用户。单击 **获取 SK** 可前往 RAM 控制台获取。具体操作，请参见 创建 AccessKey。

  在 `application.properties` 中对应的 key 为 `com.antcloud.mw.secret` 。

iii. 配置运行模式和运行环境，示例如下：

```
run.mode=NORMAL
com.alipay.env=shared
```

iv. 将上述步骤获取的参数配置在 `application.properties` 文件中。

> **说明**
>
> 更多信息，请参见 引入 SOFA 中间件。

示例工程

下文以示例工程为例，对 SOFARPC 的实现原理进行说明。

示例概述

通过 IDEA 或 Eclipse 分别打开 rpc-demo 中的 `myserver-app` 和 `myclient-app` 工程。

示例工程关键信息

- groupId：工程组织的唯一标识，示例工程为 `com.alipay.mytestsofa` 。
- artifactId：工程的构件标识符，示例工程为 `myserver-app` 或 `myclient-app` 。
- version：版本号，默认为 `1.0-SNAPSHOT` 。
- package：应用包名，默认等同于 groupId，工程示例为 `com.alipay.mytestsofa` 。

示例工程 Bean 配置

Bean 的配置分为服务发布和服务引用 2 个类型：

- 服务发布示例

```
@SofaService(interfaceType =SampleService.class,bindings =@SofaServiceBinding(bindingType
="bolt"))
```

- 服务引用示例

```
@SofaReference(interfaceType =SampleService.class,binding =@SofaReferenceBinding(bindingT
ype ="bolt",directUrl ="127.0.0.1:12201"))
privateSampleService sampleService;
```

> **说明**
>
> 示例中，为了便于对所有待引用实例进行统一管理，创建了 `ReferenceHolder` 类，进行引用管理。

示例工程原理

- 在 2 个工程的 endpoint 模块中相同位置，提供相同的服务接口和实现，并通过注解发现服务。2 个工程通过相同接口实现关联。一个客户端，一个服务端，如果是本地工程，在引用时，通过配置 directUrl，以直连方式发现服务；如果是服务器上部署测试，则通过 DSR（Direct Server Return） 底座发现服务。
- 启动 `myserver-app` Web 模块的 `SOFABootWebSpringApplication` 可发布服务。
- 启动 `myclient-app` Web 模块的 `SOFABootWebSpringApplication` 可引用服务。

示例工程服务验证

1. 启动 `myserver-app` Web 模块的 `SOFABootWebSpringApplication` 发布服务。
2. 启动 `myclient-app` Web 模块的 `SOFABootWebSpringApplication` 引用服务。

   引用成功后， `myclient-app` 控制台将输出：

   ```
   Response from myserver-app:HelloSOFARpc! times = xx
   ```

   您也可通过日志来查看服务引用结果。在 Web 子模块路径 `src/main/resources/config/application.properties` 中，可设置日志路径。默认在 `logs/myweb-app/common-default.log` 中查看服务引用结果。

应用打包和云端发布

1. 打包本地应用。

   操作步骤，请参见 编译运行。
2. 发布应用。
   - 应用整体发布流程，请参见 技术栈与应用发布流程。
   - 应用的详细发布步骤，建议根据发布方式，参考下述文档：
     - 经典应用服务，请参见 经典应用服务快速入门。
     - 容器应用服务，请参见 容器应用服务快速入门。

服务管控

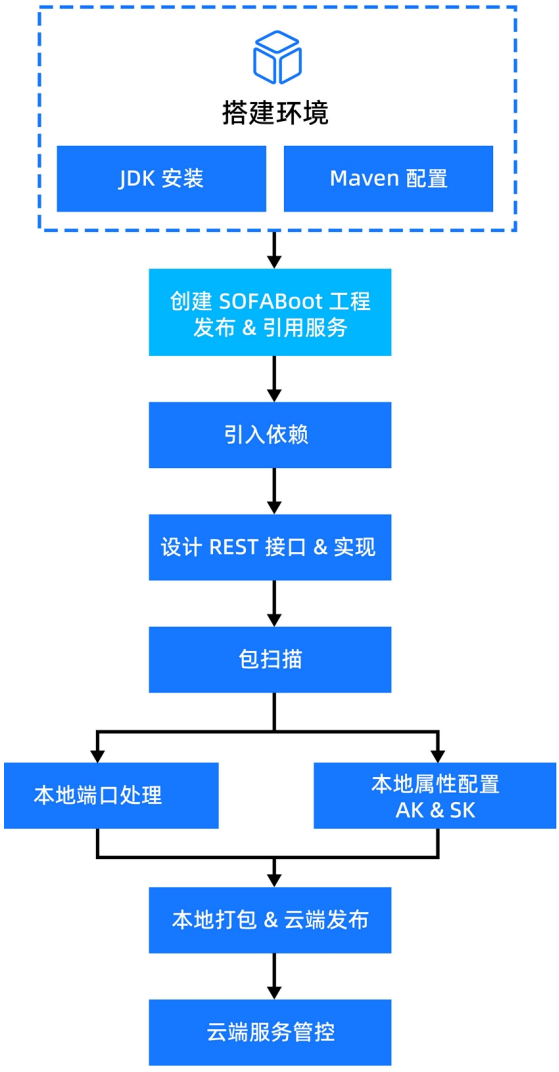
在 服务管控 页面查询并管控发布的 RPC 服务。更多详情，请参见 服务查看及管理。

3.REST 服务快速入门

微服务（SOFAStack MicroService）主要是通过 SOFARPC 来实现服务的发布和引用，而 SOFARPC 支持 REST 协议。本文以微服务本地开发到云端发布的整体流程为框架，让您了解如何在本地实现 SOFAREST 功能，以及如何将应用发布到云端，并在云端进行服务管控。本文主要讲述 SOFAREST 的实现原理，示例工程也支持直连的方式，让您在本地体验 SOFAREST 的服务发布和服务引用。

本地工程开发

SOFAREST 工程开发流程图



SOFAREST 示例代码演示视频

准备工作

- 搭建 SOFABoot 环境。具体操作，请参见 搭建环境。
- 通过以下任一方式生成 2 个 SOFABoot Web 工程，分别作为服务发布方和引用方。
  - 创建 2 个 SOFABoot Web 工程。具体操作，请参见 创建工程。

- 直接下载 示例工程。下载后，请参考 版本说明 将工程根目录下 `pom.xml` 文件中的版本号修改为最新版本号。

本地开发流程

1. 引入依赖。

   按步骤创建的工程默认已经引入该依赖，请忽略此步骤；直接下载的示例工程需在 2 个本地 SOFABoot 工程 Web 模块 `pom.xml` 中引入 SOFARPC 的 Maven 依赖。

```
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>rpc-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

2. 编写业务逻辑。

   主要为服务发布和服务引用。本示例使用注解的方式配置 Bean，实现服务发布和引用。更多详情，请参见 使用注解方式。其它配置方式，请参见 使用 XML 配置 及 使用编程 API。

   - 服务发布的业务逻辑
     - 设计服务接口类

       本示例类名称为 `SampleRestFacade.java` ，接口路径为 `com.alipay.samples.rpc.SampleRestFacade` 。

```
@Path("/sofarest")  //注意该注解的继承性问题，实现类或方法中，该注解的缺失，可能会造成 SOFA REST 调用报 404 错误。
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
@Produces(MediaType.APPLICATION_JSON +";charset=UTF-8")
public interface SampleRestFacade{
    /**
    * http://localhost:8341/sofarest/hello
    */
    @GET
    @Path("/hello")
    public String hello();
}
```

- 编写服务实现类

  本示例类名称为 `SampleRestFacadeImpl.java`，接口实现路径为 `com.alipay.samples.rpc.impl.SampleRestFacadeImpl`。

```
@Service
@SofaService(interfaceType =SampleRestFacade.class,bindings =@SofaServiceBinding(bi
ndingType ="rest"))
public class SampleRestFacadeImpl implements SampleRestFacade{
      private int count =0;

      public SampleRestFacadeImpl(){
          System.out.println("print start");
      }

     @Override
     public String hello(){
          return "Hello SOFARest! times = "+ count++;
     }
}
```

- 服务引用逻辑
  - 引用对象注入

    本示例采用 `ReferenceHolder` 类对引用对象进行统一管理，示例如下：

```
@Component
public class ReferenceHolder{
      @SofaReference(interfaceType =SampleRestFacade.class, binding =@SofaReference
Binding(bindingType ="rest",directUrl ="127.0.0.1:8341"))
      private SampleRestFacade sampleRestFacade;
      public SampleRestFacade getSampleRestFacade(){
          return sampleRestFacade;
      }
     public void setSampleRestFacade(SampleRestFacade sampleRestFacade){
         this.sampleRestFacade = sampleRestFacade;
     }
}
```

> **说明**
>
> bindingType 为 rest 协议，直连 URL 端口为 8341，即 `directUrl="127.0.0.1:8341"`。

  - 引用对象

    为了方便直观使用，本示例将引用服务逻辑放在了 `myclient-app` 工程 Web 模块的 `com.alipay.mytestsofa.SOFABootWebSpringApplication` 中，示例如下：

```
@SpringBootApplication
public class SOFABootWebSpringApplication{
```

```
    private static final Logger logger =LoggerFactory.getLogger(SOFABootWebSpringA
pplication.class);

    public static void main(String[] args){
          //*************** 注意 ******************//
         //本地同时启动 myserver-app 和 myclient-app 时，由于 tomcat 端口冲突问题，需要
修改 myclient-app 的 端口号为 8084。
         //将 myserver-app 和 myclient-app 发布到云上环境时，由于默认健康检查端口是 808
0，所以需要注释掉该行代码。
        System.setProperty("server.port","8084");
        //由于本地启动没有注册中心，所以使用本地直连的方式访问本地启动的 myserver-app，发
布到线上的时候需要注释掉该行代码。
        System.setProperty("run.mode","TEST");
        //*******************************//

        SpringApplication springApplication =newSpringApplication(SOFABootWebSprin
gApplication.class);
        ApplicationContext applicationContext = springApplication.run(args);

         if(logger.isInfoEnabled()){
               printMsg("SofaRpc Application (myclient-app) started on 8084 port."
);
         }

         ReferenceHolder referenceHolder = applicationContext.getBean(ReferenceHol
der.class);
         //调用 SOFAREST 服务。
        final SampleRestFacade sampleRestFacade = referenceHolder.getSampleRestFac
ade();

         new Thread(newRunnable(){
            @Override
            public void run(){
                 while(true){
                        try{
                                String response = sampleRestFacade.hello();
                                printMsg("Response from myserver-app.rest: "+ re
sponse);
                        }catch(Exception e){
                               e.printStackTrace();
                        }finally{
                               try{
                                       TimeUnit.SECONDS.sleep(3);
                                }catch(InterruptedException e){
                                    //ignore
                               }
                          }
                    }
                }
         }).start();
     }
    private static void printMsg(String msg){
         System.out.println(msg);
          if(logger.isInfoEnabled()){
```

```
                    logger.info(msg);
                }
            }
        }
```

3. 配置包扫描。

```
@SpringBootApplication(scanBasePackages ={"com.alipay.mytestsofa","com.alipay.samples.rpc"})
public class SOFABootWebSpringApplication{
...
}
```

4. 配置本地运行的端口。

   i. 在 `myserver-app` Web 模块 `application.properties` 文件中配置 `rpc.tr.port=12201` 。

   `rpc.tr.port` 是 TR 端口号，默认为 12200； `TR = TaobaoRemoting` 是 RPC 使用的底层通信框架。云端发布时不需要该项配置。

   ii. 运行 Web 子模块中的 `SOFABootWebSpringApplication` 。

   运行后，框架会自动进行服务的发布。为了避免端口冲突，需要在该类中指定端口。

   > **注意**
   >
   > 将 `myserver-app` 发布至云上环境前，必须注释掉代码中对 8083 端口的配置。

   iii. 在 `myclient-app` Web 模块 `application.properties` 文件中配置 `rpc.tr.port=12202` 。

   `rpc.tr.port` 是 TR 端口号，默认为 12200。云端发布时不需要该项配置。

5. 配置 `application.properties` 。

   i. 登录 SOFAStack 控制台。

ii. 在左侧导航栏选择 **研发效能 > 脚手架**，然后在 Step 2 中获取如下信息：

- **实例标识**：应用实例在工作空间中的唯一标识。

  在 `application.properties` 中对应的 key 为 `com.alipay.instanceid` 。

- **AntVIP**：区域的唯一标识，每个区域一个地址。应用通过 AntVIP 寻找所在区域环境的地址。不同环境的 AntVIP 地址值如下：

  - 杭州金区：`cn-hangzhou-fin-middleware-acvip-prod.cloud.alipaycs.net`
  - 上海非金：`cn-shanghai-middleware-acvip-prod.cloud.alipaycs.net`
  - 上海金区：`cn-shanghai-fin-sofastack-middleware-acvip-prod.cloud.alipaycs.net`
  - 杭州非金：`cn-hangzhou-middleware-acvip-prod.cloud.alipaycs.net`

  在 `application.properties` 中对应的 key 为 `com.antcloud.antvip.endpoint` 。

- **Access Key ID**：用于标识用户。单击 **获取 AK** 可前往 RAM 控制台获取。具体操作，请参见 创建 AccessKey。

  在 `application.properties` 中对应的 key 为 `com.antcloud.mw.access` 。

- **Access Key Secret**：用于验证用户。单击 **获取 SK** 可前往 RAM 控制台获取。具体操作，请参见 创建 AccessKey。

  在 `application.properties` 中对应的 key 为 `com.antcloud.mw.secret` 。

iii. 配置运行模式和运行环境，示例如下：

```
run.mode=NORMAL
com.alipay.env=shared
```

iv. 将上述步骤获取的参数配置在 `application.properties` 文件中。

> 说明
>
> 更多信息，请参见 引入 SOFA 中间件。

示例工程

下文以示例工程为例，对 SOFARPC 的实现原理进行说明。

示例概述

通过 IDEA 或 Eclipse 分别打开 rpc-demo 中的 `myserver-app` 和 `myclient-app` 工程。

示例工程关键信息

- groupId: 工程组织的唯一标识，示例工程为 `com.alipay.mytestsofa` 。
- artifactId：工程的构件标识符，示例工程为 `myserver-app` 或 `myclient-app` 。
- version：版本号，默认为 `1.0-SNAPSHOT` 。
- package：应用包名，默认等同于 groupId，工程示例为 `com.alipay.mytestsofa` 。

示例工程 Bean 配置

Bean 的配置分为服务发布和服务引用 2 个类型：

- 服务发布的示例

```
@SofaService(interfaceType =SampleRestFacade.class,bindings =@SofaServiceBinding(bindingT
ype ="rest"))
```

- 服务引用的示例

```
@SofaReference(interfaceType =SampleRestFacade.class, binding =@SofaReferenceBinding(bind
ingType ="rest",directUrl ="127.0.0.1:8341"))
private SampleRestFacade sampleRestFacade;
```

> 说明
>
> 示例中，为了便于对所有待引用实例进行统一管理，创建了 `ReferenceHolder` 类，进行引用管理。

示例工程的 REST 实现

SOFAREST 的实现基于 SOFARPC，SOFARPC 的实现原理说明如下：

- 在 2 个工程的 endpoint 模块中相同位置，提供相同的服务接口和实现，并通过注解发现服务。2 个工程通过相同接口实现关联。一个客户端，一个服务端，如果是本地工程，在引用时，通过配置 directUrl，以直连方式发现服务；如果是服务器上部署测试，则通过 DSR（Direct Server Return） 底座发现服务。
- 启动 `myserver-app` Web 模块的 `SOFABootWebSpringApplication` 可发布服务。

- 启动 `myclient-app` Web 模块的 `SOFABootWebSpringApplication` 可引用服务。

示例工程服务验证

1. 启动 `myserver-app` Web 模块的 `SOFABootWebSpringApplication` 发布服务。
2. 启动 `myclient-app` Web 模块的 `SOFABootWebSpringApplication` 引用服务。

   引用成功后， `myclient-app` 控制台将输出：

   ```
   Response from myserver-app.rest:HelloSOFARest! times = xx
   ```

   本地浏览器访问 `http://localhost:8341/sofarest/hello` ，将输出访问 URL 链接那一刻，客户端的调用次数。

   ```
   HelloSOFARest! times = xx
   ```

云端发布后，可以在客户端命令行中输入 `curl http://{服务器 IP 地址}:8341/sofarest/hello` 命令进行验证。

您也可以通过日志来查看服务引用结果：默认在 `/logs/myclient-app/common-default.log` 中查看服务引用结果。您可以在 Web 子模块路径 `src/main/resources/config/application.properties` 中修改日志路径。

应用打包和云端发布

1. 打包本地应用。

   操作步骤，请参见 编译运行。
2. 发布应用。
   - 应用整体发布流程，请参见 技术栈与应用发布流程。
   - 应用的详细发布步骤，建议根据发布方式，参考下述文档：
     - 经典应用服务，请参见 经典应用服务快速入门。
     - 容器应用服务，请参见 容器应用服务快速入门。

服务管控

在 服务管控 页面查询并管控发布的 RPC 服务。更多详情，请参见 服务管控。

4.服务发布与引用

4.1. 发布 SOFARPC 服务

RPC 是日常开发中最常用的中间件，本文主要说明如何发布一个 RPC 服务。

前提条件

- 已完成环境搭建。具体步骤，请参见 搭建环境。
- 已下载 示例工程。
- 已将工程导入 IDE 工具。具体操作，请参见 将 Web 工程导入 IDE。

发布 SOFARPC 服务

要发布一个 RPC 服务，主要步骤说明如下：

1. 在 `app/endpoint/` 下设计接口，示例如下：

```
// com.alipay.APPNAME.facade.SampleService
public interface SampleService{
    String message();
}
```

2. 提供一个接口的实现类，示例如下：

```
// com.alipay.APPNAME.service.SampleServiceImpl
public class SampleServiceImpl implements SampleService{
   @Override
   public String message(){
        return "Hello, Service SOFABootRPC!";
   }
}
```

3. 在接口所在模块中，通过 `resources/META-INF` 下的 xml 文件，将接口实现类配置为一个 Java bean。

```
<bean id="sampleServiceBean" class="com.alipay.APPNAME.service.SampleServiceImpl"/>
```

4. 在接口所在模块中，通过 `resources/META-INF` 下的 xml 文件，发布 RPC 服务。根据接口实现类的个数，可以分为下述 2 种：
   - 单接口单实现，示例如下：

```
<!-- 服务 -->
<sofa:service ref="sampleServiceBean" interface="com.alipay.APPNAME.facade.SampleServ
ice">
<sofa:binding.bolt/>
</sofa:service>
```

   - 单接口多实现，示例如下：

```
<!-- 服务一 -->
<sofa:service ref="sampleServiceBean1" interface="com.alipay.APPNAME.facade.SampleSer
vice"unique-id="service1">
<sofa:binding.bolt/>
</sofa:service>

<!-- 服务二 -->
<sofa:service ref="sampleServiceBean2" interface="com.alipay.APPNAME.facade.SampleSer
vice"unique-id="service2">
<sofa:binding.bolt/>
</sofa:service>
```

说明

- 服务提供方定义服务 `<sofa:service>`，并进行服务发布；服务消费方定义服务引用 `<sofa:reference>`，并进行服务引用。任何一个 SOFA 框架应用节点都可以同时发布服务和引用其它节点的服务。
- RPC 服务提供方通过服务 `<sofa:service>` 来定义，主要属性有 `interface`、`unique-id` 和 `ref`。
  - `interface`：该属性用于确定一个服务，属性值为：命名空间包名 + Java 接口名。
  - `unique-id`：如果同一个接口有两个不同的实现，而这两个不同的实现都需要发布成 SOFA 的 RPC 服务，则可以在发布服务的时候加上一个 `unique-id` 属性来进行区分。
  - `ref`：该属性用于指定服务实现所对应的 Spring Bean，通过 Bean ID 和服务实现类进行关联。

本地运行

1. 在工程的根目录下执行 `mvn clean install` 命令，在 `target` 目录下生成一个可执行的 `Fat JAR` 文件，例如：`APPNAME-service-1.0-SNAPSHOT-executable.jar`。
2. 通过以下任一方法执行 JAR 文件，如果没有错误日志输出，则表示 `sofaboot-rpc-server` 工程启动成功。
   - 在命令行中执行 `java -jar APPNAME-service-1.0-SNAPSHOT-executable.jar` 命令。
   - 在本地 IDE 中直接运行 `main` 函数。

云端运行

SOFARPC 在本地只能通过直连方式进行体验。只有在云端发布成功后，通过 SOFAStack 控制台，才能进行服务管控和治理。具体云端发布步骤，请参考下述信息：

- 对于应用的整体发布流程，请参见 技术栈与应用发布流程。
- 应用的详细发布步骤，建议根据发布方式，参考下述文档：
  - 经典应用服务，请参见 经典应用服务快速入门。
  - 容器应用服务，请参见 容器应用服务快速入门。

> 说明
>
> 云端发布时，您必须在 `application.properties` 中配置以下属性：
>
> - `run.mode=NORMAL`
> - `com.alipay.env=shared`
> - `com.alipay.instanceid=`
> - `com.antcloud.antvip.endpoint=`
> - `com.antcloud.mw.access=`
> - `com.antcloud.mw.secret=`
>
> 运行模式和运行环境的值为默认固定值。参数的具体配置，请参见 properties 配置项。

日志查看

1. 查看本地工程 RPC 发布服务启动日志 `logs/rpc/common-default.log` 。

   如出现类似以下内容，说明 RPC 服务端成功启动：

   ```
   2016-12-1715:16:44,466 INFO  main                                - sofa rpc run.mode = DEV
   2016-12-1715:16:49,479 INFO  main                                - PID:42843 sofa rpc star
   ting!
   ```

2. 查看日志 `logs/rpc/rpc-registry.log` ，内容参考如下：

   ```
   2016-12-1715:17:07,764 INFO  main                                RPC-REGISTRY -发布 RPC 服
   务：服务名[com.alipay.APPNAME.facade.SampleService:1.0@DEFAULT]
   ```

3. 查看错误日志 `logs/rpc/common-error.log` 。

   如果没有任何错误日志输出且应用启动正常，说明成功发布了一个 RPC 服务。

关于日志的更多详情，请参见 日志说明。

4.2. 引用 SOFARPC 服务

RPC 是日常开发中最常用的中间件，本文主要说明如何引用一个 RPC 服务。

前提条件

- 已完成环境搭建。具体步骤，请参见 搭建环境。
- 已下载 示例工程。
- 已将工程导入 IDE 工具。具体操作，请参见 将 Web 工程导入 IDE。

引入接口定义依赖

要引用一个 RPC 服务，您需要知道 RPC 服务的提供方所发布的接口是什么（如果发布的服务有 `unique-id` ，您还需要知道 `unique-id` ），这就要求服务提供方将发布的接口所在的 JAR 包及依赖信息传到 Maven 仓库，以便服务引用方能够引用服务提供方所发布的 RPC 服务。

- 如果是本地运行，需要在 `sofaboot-rpc-server` 工程目录运行 `mvn clean install` ，将接口依赖的 JAR 安装到本地仓库。
- 若非本地运行，需要将接口依赖 JAR 上传到对应的 Maven 仓库。

获得 RPC 服务的发布接口后，在 `sofaboot-rpc-client` 工程下的主 pom 中添加所引用的 RPC 服务的接口依赖信息，示例如下：

```
<dependency>
    <groupId>com.alipay.APPNAME</groupId>
    <artifactId>APPNAME-facade</artifactId>
     <version>1.0-SNAPSHOT</version>
</dependency>
```

> **说明**
>
> 此处客户端和服务端的应用名称均命名为 APPNAME，仅供学习使用。在实际环境中，两个应用名称不能完全一样。

引用 RPC 服务

在配置文件 `META-INF/APPNAME/APPNAME-web.xml` 中，根据接口配置引用一个 RPC 服务：

```
<sofa:reference id="sampleRpcService" interface="com.alipay.APPNAME.facade.SampleService">
<sofa:binding.bolt/>
</sofa:reference>
```

此处的 RPC 引用也是一个 bean，其 bean id 为 `sampleRpcService` 。

将引用的 RPC 服务注入 Controller

> **注意**
>
> 这一步是为了演示方便，实现用户通过浏览器或者其他方式访问一个 rest 接口，触发调用所引用的服务，并调用到服务端。实际开发中，并不需要注入 Controller 这一步。

本教程将这个 RPC 服务注入到了 `com.alipay.APPNAME.web.springrest.RpcTestController` 中，示例如下：

```
@RestController
@RequestMapping("/rpc")
public class RpcTestController{
    @Autowired
     private SampleService sampleRpcService;
     @RequestMapping("/hello")
     String rpcUniqueAndTimeout(){
          String rpcResult =this.sampleRpcService.message();
          return rpcResult;
     }
}
```

本地编译

`sofaboot-rpc-client` 是一个 SOFABoot Web 工程。依次执行以下命令以进行本地编译并启动 RPC Client：

1. 将客户端和服务端 `config/application.properties` 中的 `run.mode` 均配置为 `DEV`，即 `run.mode=DEV`。
2. 在工程根目录下执行 `mvn clean install` 命令，生成可执行文件 `target/APPNAME-web-1.0-SNAPSHOT-executable.jar`。
3. 在工程根目录下执行 `java -jar ./target/APPNAME-web-1.0-SNAPSHOT-executable.jar` 命令。
   - 如果控制台输出如下信息，则表示 Web 容器启动成功。

     ```
     16:11:13.625 INFO  org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedSer
     vletContainer-Tomcat started on port(s):8080(http)
     ```

   - 如果输出错误信息，请在解决问题后重试以上步骤。

测试 RPC 服务

1. 启动 RPC 服务端 `sofaboot-rpc-server` 及客户端 `sofaboot-rpc-client`。
2. 在浏览器中访问 `http://localhost:8080/rpc/hello` 来测试引用的 RPC 服务。

   当浏览器输出如下信息时，表示 RPC 服务发布和引用均成功。

   ```
   Hello,ServiceSOFABoot
   ```

云端运行

SOFARPC 在本地只能通过直连方式进行体验。只有在云端发布成功后，通过 SOFAStack 控制台，才能进行服务管控和治理。具体云端发布步骤，请参考下述信息：

- 对于应用的整体发布流程，请参见 技术栈与应用发布流程。
- 应用的详细发布步骤，建议根据发布方式，参考下述文档：
  - 经典应用服务，请参见 经典应用服务快速入门。

- 容器应用服务，请参见 容器应用服务快速入门。

> **说明**
>
> 云端发布时，您必须在 `application.properties` 中配置以下属性：
>
> - `run.mode=NORMAL`
> - `com.alipay.env=shared`
> - `com.alipay.instanceid=`
> - `com.antcloud.antvip.endpoint=`
> - `com.antcloud.mw.access=`
> - `com.antcloud.mw.secret=`
>
> 运行模式和运行环境的值为默认固定值。其他参数的具体配置，请参见 properties 配置项。

日志查看

查看 `sofaboot-rpc-client` 工程引用 RPC 服务启动日志 `logs/rpc/rpc-registry.log`，出现类似如下日志时，说明 RPC 服务引用成功。

```
2016-12-1715:45:50,340 INFO  main                                RPC-REGISTRY -订阅 RPC 服务：服务名[com.alipay.APPNAME.facade.SampleService:1.0@DEFAULT]
```

如果发现引用 RPC 服务失败，请重点关注日志目录 `logs` 下的所有 `common-error.log`。

有关日志的详细信息，请参考 日志说明。

5.配置说明

5.1. 配置方式

SOFARPC 的服务发布和引用方式包括使用注解方式、使用 XML 配置方式和使用编程 API 方式。

使用注解方式

SOFABoot 环境支持使用注解方式，包括以下两种：

- 单协议注解：`@SofaService` 和 `@SofaReference` .
- 多协议注解：增加注解 `@SofaServiceBinding` 和 `@SofaReferenceBinding` 。

服务发布

如果要发布一个 RPC 服务，只需要在 Bean 上面打上 `@SofaService` 注解，指定接口和协议类型即可。

```
@SofaService(interfaceType =AnnotationService.class, bindings ={@SofaServiceBinding(binding
Type ="bolt")})
@Component
public class AnnotationServiceImpl implements AnnotationService{
     @Override
     public String sayAnnotation(String helloAnno){
          return helloAnno;
     }
}
```

服务引用

对于需要引用远程服务的 Bean，只需要在属性或者方法上打上 `Reference` 的注解即可，支持 Bolt、Dubbo、REST 协议。

```
@Component
public class AnnotationClientImpl{

     @SofaReference(interfaceType =AnnotationService.class, binding =@SofaReferenceBinding(
bindingType ="bolt"))
     private AnnotationService annotationService;

      public String sayClientAnnotation(String clientAnno){

             String result = annotationService.sayAnnotation(clientAnno);

             return result;
       }
}
```

使用 XML 配置

XML 配置中主要标签含义如下：

- `sofa:service` ：表示发布服务。
- `sofa:reference` ：表示引用服务。
- `sofa:binding` ：表示服务发布或引用的协议。

> **说明**
>
> XML 配置完整示例，请参见 示例工程。

服务发布示例

- 单协议发布

```
<bean id="personServiceImpl" class="com.alipay.sofa.boot.examples.demo.rpc.bean.PersonSer
viceImpl"/>
<sofa:service ref="personServiceImpl" interface="com.alipay.sofa.boot.examples.demo.rpc.b
ean.PersonService">
   <sofa:binding.bolt/>
</sofa:service>
```

- 多协议发布

```
<sofa:service ref="personServiceImpl" interface="com.alipay.sofa.boot.examples.demo.rpc.b
ean.PersonService">
    <sofa:binding.bolt/>
    <sofa:binding.rest/>
    <sofa:binding.dubbo/>
</sofa:service>
```

服务引用示例

- Bolt 协议引用

```
<sofa:reference id="personReferenceBolt" interface="com.alipay.sofa.boot.examples.demo.rp
c.bean.PersonService">
     <sofa:binding.bolt/>
</sofa:reference>
```

- REST 协议引用

```
<sofa:reference id="personReferenceRest" interface="com.alipay.sofa.boot.examples.demo.rp
c.bean.PersonService">
    <sofa:binding.rest/>
</sofa:reference>
```

使用编程 API

SOFA 提供一套机制去存放各种组件的编程 API，并提供一套统一的方法，让您可以获取到这些 API。组件编程 API 的存放与获取均通过 SOFA 的 ClientFactory 类进行，通过这个 ClientFactory 类，可以获取到对应组件的编程 API。

前提条件

使用 SOFA 组件编程相关的 API，请确保使用的模块里面已经添加了如下的依赖：

```
<dependency>
     <groupId>com.alipay.sofa</groupId>
     <artifactId>rpc-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

配置方式

SOFA 提供两种方式获取 `ClientFactory` ：

- 实现 ClientFactoryAware 接口

  代码示例如下：

  ```
  public class ClientFactoryBean implements ClientFactoryAware{
        private ClientFactory clientFactory;

        @Override
        public void setClientFactory(ClientFactory clientFactory){
            this.clientFactory = clientFactory;
        }

       public ClientFactory getClientFactory(){
            return clientFactory;
       }
  }
  ```

  然后，将上面的 `ClientFactoryBean` 配置成一个 Spring Bean。

  ```
  <bean id="clientFactoryBean" class="com.alipay.test.ClientFactoryBean"/>
  ```

  完成后， `ClientFactoryBean` 就可以获取到 `clientFactory` 对象来使用了。

- 使用 @SofaClientFactory 注解

  代码示例如下：

  ```
  public class ClientAnnotatedBean{
        @SofaClientFactory
        private ClientFactory clientFactory;

        public ClientFactory getClientFactory(){
             return clientFactory;
        }
  }
  ```

  您需要在 `ClientFactory` 字段上加上 `@SofaClientFactory` 的注解，然后将 `ClientAnnotatedBean` 配置成一个 Spring Bean 。

```
<bean id="clientAnnotatedBean" class="com.alipay.test.ClientAnnotatedBean"/>
```

完成后，SOFA 框架就会自动将 `ClientFactory` 的实例注入到被 `@SofaClientFactory` 注解的字段上。

以上操作只是获取到 `ClientFactory` 这个对象，如果要获取特定的客户端（如 `ServiceFactory`），您还需调用 `ClientFactory` 的 `getClient` 方法。SOFA 对 `@SofaClientFactory` 的注解进行了增强，可以直接通过 `@SofaClientFactory` 来获取具体的 Client，代码示例如下：

```
public class ClientAnnotatedBean{
    @SofaClientFactory
    private ServiceClient serviceClient;

    public ServiceClient getServiceClient(){
        return serviceClient;
    }
}
```

当 `@SofaClientFactory` 直接使用在具体的 Client 对象上时，此注解可以直接将对应的 Client 对象注入到被注解的字段上。在上述例子中，`@SofaClientFactory` 是直接注解在类型为 `ServiceClient` 的字段上，SOFA 框架会直接将 `ServiceClient` 对象注入到这个字段上。所有在 `ClientFactory` 中存在的对象，都可以通过此种方式来获得。

编程 API 示例

服务的发布与订阅不仅可以通过在 XML 中配置 Spring Bean 的方式在应用启动期静态加载，也可以采用编程 API 的方式在应用运行期动态执行，用法如下：

Bolt 服务发布

```
ServiceClient serviceClient = clientFactory.getClient(ServiceClient.class);

ServiceParam serviceParam = new ServiceParam();
serviceParam.setInstance(sampleService);
serviceParam.setInterfaceType(SampleService.class);
serviceParam.setUniqueId(uniqueId);

TrBindingParam trBinding = new TrBindingParam();
// 对应 global-attrs 标签。
trBinding.setClientTimeout(5000);

serviceParam.addBindingParam(trBinding);

serviceClient.service(serviceParam);
```

Bolt 服务引用

```
ReferenceClient referenceClient = clientFactory.getClient(ReferenceClient.class);
ReferenceParam<SampleService> referenceParam = new ReferenceParam<SampleService>();
referenceParam.setInterfaceType(SampleService.class);
referenceParam.setUniqueId(uniqueId);

TrBindingParam trBinding = new TrBindingParam();
// 对应 global-attrs 标签。
trBinding.setClientTimeout(8000);

// 对应 method 标签。
TrBindingMethodInfo trMethodInfo = new TrBindingMethodInfo();
trMethodInfo.setName("helloMethod");
trMethodInfo.setType("callback");
// 对象必须实现 com.alipay.sofa.rpc.api.callback.SofaResponseCallback 接口。
trMethodInfo.setCallbackHandler(callbackHandler);
trBinding.addMethodInfo(trMethodInfo);

referenceParam.setBindingParam(trBinding);

SampleService proxy = referenceClient.reference(referenceParam);
```

警告

通过动态客户端创建 SOFA Reference 返回的对象是一个非常重要的对象，在使用的时候不要频繁创建，自行做好缓存，否则可能存在内存溢出的风险。 因为编程 API 的类不能轻易变化，类名为了兼容以前的用法，保持 TR 写法，但实际其中走的是 Bolt 协议。

5.2. 应用维度配置

本文介绍 SOFARPC 可用的配置项和常见配置。

基础配置项说明

您可以根据自身环境的需求，在 `application.properties` 文件中添加以下配置项：

配置项	类型	说明	默认值
`sofa_runtime_local_mode`	BOOLEAN	本地优先调用开关。	false
`run_mode`	STRING	RPC 运行模式。	空
`rpc_tr_port`	INTEGER	TR 端口号。	12200

配置项	类型	说明	默认值
rpc_bind_network_interface	STRING	服务器绑定固定网卡。	空
rpc_enabled_ip_range	STRING	服务器绑定本地 IP 范围。	空
rpc_min_pool_size_tr	INTEGER	TR 服务器线程池最小线程数。	20
rpc_max_pool_size_tr	INTEGER	TR 服务器线程池最大线程数。	200
rpc_pool_queue_size_tr	INTEGER	TR 服务器线程池队列大小。	0
com.alipay.sofa.rpc.bolt.port	INTEGER	Bolt 端口号。	12200
com.alipay.sofa.rpc.bolt.thread.pool.core.size	INTEGER	Bolt 服务器线程池最小线程数。	20
com.alipay.sofa.rpc.bolt.thread.pool.max.size	INTEGER	Bolt 服务器线程池最大线程数。	200
com.alipay.sofa.rpc.bolt.thread.pool.queue.size	INTEGER	Bolt 服务器线程池队列大小。	0
com.alipay.sofa.rpc.rest.port	INTEGER	SOFAREST 端口号。	8341
rpc_transmit_url	STRING	预热与权重配置。	空

配置项	类型	说明	默认值
rpc_transmit_url_timeout_tr	INTEGER	预热调用超时时间，单位 ms。	0
rpc_profile_threshold_tr	INTEGER	RPC 服务处理性能日志打印阈值，单位 ms。	300

本地优先调用模式

当本地启动多个 SOFA 应用时，要使这几个应用能优先相互调用，而不需要经过软负载过程，只需要在 `application.properties` 中加入 `sofa_runtime_local_mode=true` 即可。

但是 `sofa_runtime_local_mode` 这个配置依然需要依赖于配置中心推送下来的地址。拿到服务提供方地址列表后，服务消费方会优先选择本地的 IP 地址进行服务调用。如果开发者所处的工作空间没有配置中心，则需要指定服务提供方地址进行调用，具体参见 路由与配置中心。

```
application.properties:

run_mode=TEST

<!--服务应用方配置-->
<sofa:reference ...>
     <sofa:binding.bolt>
         <global-attrs test-url="localhost:12200"/>
     </sofa:binding.bolt>
</sofa:reference>
```

IP 或网卡绑定

SOFARPC 发布服务地址的时候，只会选取本地的第一张网卡的 IP 发布到配置中心，如果有多张网卡（如在 SOFAStack 平台上，有内网 IP 和外网 IP），则需要设置 IP 选择策略。

SOFARPC 提供了两种方式选择 IP：

- `rpc_bind_network_interface`

  指定具体的网卡名进行选择，如：`rpc_bind_network_interface=eth0` 。

- `rpc_enabled_ip_range`

  指定 IP 范围进行绑定，格式：`IP_RANGE1:IP_RANGE2,IP_RANGE` 。例如，`rpc_enabled_ip_range=10.1:10.2,11` 表示 IP 范围在 `10.1.0.0~10.2.255.255` 和 `11.0.0.0~11.255.255.255` 内的才会选择。

> 说明
>
> SOFAStack 平台的内网地址均绑定在 eth0 网卡上，推荐直接使用 `rpc_bind_network_interface=eth0` 配置。如果应用运行在其它非 SOFAStack 平台上，请查看运行机器的内网地址自行修改。查看机器地址的命令如下：
>
> - Windows 系统：`ipconfig`
> - Mac 和 Linux 系统：`ifconfig`

TR 线程池配置

在 `application.properties` 文件中使用以下选项配置 TR 线程池信息：

- `com.alipay.sofa.rpc.bolt.thread.pool.core.size`：最小线程数，默认 20。
- `com.alipay.sofa.rpc.bolt.thread.pool.max.size`：最大线程数，默认 200。
- `com.alipay.sofa.rpc.bolt.thread.pool.queue.size`：队列大小，默认 0。

TR 采用了 JDK 中的线程池 `ThreadPoolExecutor`。当核心线程池扩张时，先涨到最小线程数大小。当并发请求达到最小线程数后，请求被放入线程池队列中。队列满了之后，线程池会扩张到最大线程数指定的大小。如果超过最大线程数则会抛出 `RejectionException` 异常。

5.3. 应用维度配置扩展

在 SOFABoot 的使用场景下，RPC 框架在应用层面提供一些配置参数，如端口、线程池等信息。

应用参数都是通过 `Spring Boot@ConfigurationProperties` 进行的绑定，绑定属性类为 `com.alipay.sofa.rpc.boot.config.SofaBootRpcProperties`，配置前缀如下：

```
static final String PREFIX ="com.alipay.sofa.rpc";
```

您可以根据需要，在 `application.properties` 文件中增加以下配置项：

> 说明
>
> 您也可以根据自己的编码习惯按照 Spring Boot 的规范，使用驼峰、中划线等进行书写。

```
# 单机故障剔除
com.alipay.sofa.rpc.aft.regulation.effective # 是否开启单机故障剔除功能。
com.alipay.sofa.rpc.aft.degrade.effective  # 是否开启降级。
com.alipay.sofa.rpc.aft.time.window # 时间窗口。
com.alipay.sofa.rpc.aft.least.window.count # 最小调用次数。
com.alipay.sofa.rpc.aft.least.window.exception.rate.multiple # 最小异常率。
com.alipay.sofa.rpc.aft.weight.degrade.rate # 降级速率。
```

```
com.alipay.sofa.rpc.aft.weight.recover.rate # 恢复速率。
com.alipay.sofa.rpc.aft.degrade.least.weight #降级最小权重。
com.alipay.sofa.rpc.aft.degrade.max.ip.count # 最大降级 IP。

# bolt
com.alipay.sofa.rpc.bolt.port # bolt 端口。
com.alipay.sofa.rpc.bolt.thread.pool.core.size # bolt 核心线程数。
com.alipay.sofa.rpc.bolt.thread.pool.max.size # bolt 最大线程数。
com.alipay.sofa.rpc.bolt.thread.pool.queue.size # bolt 线程池队列。
com.alipay.sofa.rpc.bolt.accepts.size # 服务端允许客户端建立的连接数。

# rest
com.alipay.sofa.rpc.rest.hostname # rest hostname。
com.alipay.sofa.rpc.rest.port # rest port。
com.alipay.sofa.rpc.rest.io.thread.size # rest io 线程数。
com.alipay.sofa.rpc.rest.context.path # rest context path。
com.alipay.sofa.rpc.rest.thread.pool.core.size # rest 核心线程数。
com.alipay.sofa.rpc.rest.thread.pool.max.size # rest 最大线程数。
com.alipay.sofa.rpc.rest.max.request.size # rest 最大请求大小。
com.alipay.sofa.rpc.rest.telnet # 是否允许 rest telnet
com.alipay.sofa.rpc.rest.daemon # 是否hold住端口，true的话随主线程退出而退出。

# dubbo
com.alipay.sofa.rpc.dubbo.port # dubbo port。
com.alipay.sofa.rpc.dubbo.io.thread.size # dubbo io 线程大小。
com.alipay.sofa.rpc.dubbo.thread.pool.max.size # dubbo 业务线程最大数。
com.alipay.sofa.rpc.dubbo.accepts.size # dubbo 服务端允许客户端建立的连接数。
com.alipay.sofa.rpc.dubbo.thread.pool.core.size #dubbo 核心线程数。
com.alipay.sofa.rpc.dubbo.thread.pool.queue.size #dubbo 最大线程数。

# registry
com.alipay.sofa.rpc.registry.address # 注册中心地址。
com.alipay.sofa.rpc.virtual.host # virtual host。
com.alipay.sofa.rpc.bound.host # 绑定 host。
com.alipay.sofa.rpc.virtual.port # virtual 端口。
com.alipay.sofa.rpc.enabled.ip.range # 多网卡 IP 范围。
com.alipay.sofa.rpc.bind.network.interface # 绑定网卡。

# h2c
com.alipay.sofa.rpc.h2c.port # h2c 端口。
com.alipay.sofa.rpc.h2c.thread.pool.core.size # h2c 核心线程数。
com.alipay.sofa.rpc.h2c.thread.pool.max.size # h2c 最大线程数。
com.alipay.sofa.rpc.h2c.thread.pool.queue.size # h2c 队列大小。
com.alipay.sofa.rpc.h2c.accepts.size # 服务端允许客户端建立的连接数。

# 扩展
com.alipay.sofa.rpc.lookout.collect.disable # 是否关闭 lookout。

# 代理
com.alipay.sofa.rpc.consumer.repeated.reference.limit # 允许客户端对同一个服务生成的引用代理数量，默认为3。

# 注册中心
<sofa:binding.bolt>
```

```
<sofa:global-attrs registry="mesh" />  # 指定服务的注册中心为 mesh。
</sofa:binding.bolt>
```

> **说明**
>
> 服务注册中心默认为 DSR，若您有特殊需求需要变更，请参见 注册中心路由。

5.4. 服务维度配置

本文介绍在 SOFABoot 环境下完整的 SOFARPC 服务发布与引用说明。

发布服务

```
<bean id="helloSyncServiceImpl" class="com.alipay.sofa.rpc.samples.invoke.HelloSyncServiceImpl"/>
<sofa:service ref="helloSyncServiceImpl" interface="com.alipay.sofa.rpc.samples.invoke.HelloSyncService"unique-id="">
<sofa:binding.bolt>
<sofa:global-attrs registry="" serialize-type="" filter="" timeout="3000" thread-pool-ref=""
warm-up-time="60000"
warm-up-weight="10" weight="100"/>
</sofa:binding.bolt>
<sofa:binding.rest>
</sofa:binding.rest>
</sofa:service>
```

属性说明：

属性	默认值	说明
id	Bean 名	ID
class	-	类
ref	-	服务接口实现类
interface	-	服务接口（唯一标识元素）
unique-id	-	服务标签（唯一标识元素）
filter	-	过滤器配置别名

属性	默认值	说明
registry	DSR	服务端注册中心。多个注册中心需使用逗号分隔。 支持的注册中心请参见 注册中心路由。
timeout	3000	服务端执行超时时间 单位：毫秒。
serialize-type	hessian2	序列化协议，取值为：hessian2、protobuf。
thread-pool-ref	-	自定义业务线程池，需要指向一个 `com.alipay.sofa.rpc.core.context.UserThreadPool` 对象。
weight	100	服务静态权重
warm-up-weight	10	服务提供者的预热权重，当客户端刚建立连接时，在指定的预热时间内会使用此权重。
warm-up-time	0	服务提供者的预热时间，当客户端刚建立连接时，在该时间内会使用预热权重。 单位：毫秒。

引用服务

```
<sofa:reference jvm-first="false" id="helloSyncServiceReference"
interface="com.alipay.sofa.rpc.samples.invoke.HelloSyncService" unique-id="">
<sofa:binding.bolt>
<sofa:global-attrs type="sync" timeout="3000" callback-ref="" callback-class="" address-wai
t-time="1000"
connect.num="1" check="false" connect.timeout="1000" filter="" generic-interface=""
idle.timeout="1000"
idle.timeout.read="1000" lazy="false" loadBalancer="" registry="" retries="1"
serialize-type=""/>
<sofa:route target-url="xxx:12200"/>
<sofa:method name="hello" callback-class="" callback-ref="" timeout="3000" type="sync"/>
</sofa:binding.bolt>
</sofa:reference>
```

属性说明：

属性	默认值	说明
id	自动生成	ID
jvm-first	true	是否优先使用本地 JVM 配置，取值：true、false。
interface	-	服务接口（唯一标识元素） 无论是普通调用还是返回调用，都必须设置实际的接口类。
unique-id	-	服务标签（唯一标识元素）
type	sync	调用方式。取值为：sync、oneway、callback、future。
filter	-	过滤器配置别名
registry	DSR	服务端注册中心。多个注册中心需使用逗号分隔。 支持的注册中心请参见 注册中心路由。

属性	默认值	说明
method	-	方法级配置，`method` 配置的属性优先级高于 `global-attrs` 中的配置。没有配置时，默认跟随全局配置。
serialize-type	hessian2	序列化协议，取值为：hessian2、protobuf。
target-url	-	使用直连调用时配置的直连地址。 配置 `target-url` 后，软负载阶段将会失效。详情请参见 直连调用。
generic-interface	-	泛化接口
connect.timeout	1000	消费方连接超时时间 单位：毫秒。
connect.num	-1	消费方连接数 -1 表示不设置（默认为每个目标地址建立一个连接）。
idle.timeout	-1	消费方最大空闲时间 -1 表示使用底层默认值（底层默认值为 0，表示永远不会读 idle）。该配置也是心跳的时间间隔，当请求 idle 时间超过配置时间后，发送心跳到服务方。

属性	默认值	说明
idle.timeout.read	-1	消费方最大读空闲时间 -1 表示使用底层默认值（底层默认值为 30）。如果 `idle.timeout > 0`，在 `idle.timeout` + `idle.timeout.read` 时间内，如果没有请求发生，将自动断开连接。
loadBalancer	random	负载均衡算法
lazy	false	是否延迟建立长连接，取值：true、false。
address-wait-time	0	reference 生成时，等待配置中心将地址推送到消费方的时间。 单位：毫秒。 最大值为 30000 毫秒，超过这个值时会被调整为 30000 毫秒。
timeout	3000(cover 5000)	调用超时时间
retries	0	当出现非业务异常时，框架层面自动重试的次数。需要服务提供者自己保证方法的幂等性。
callback-class	-	callback 回调类 callback 时才可用。
callback-ref	-	callback 回调类 callback 时才可用。

配置的优先级

SOFARPC 里面的某些配置，例如调用超时 `timeout` 属性可以在服务提供方设置，也可以在服务调用方设置。这些配置的优先级从高到低排列如下：

1. 线程调用级别设置。
2. 服务调用方。
   - 方法级别设置。
   - Reference 级别设置。
3. 服务提供方。
   - 方法级别设置。
   - Service 级别设置。

6.服务路由

6.1. 服务路由介绍

RPC 最重要的功能是获取对端地址。根据使用环境的不同，服务路由策略主要有软负载和直连两种。

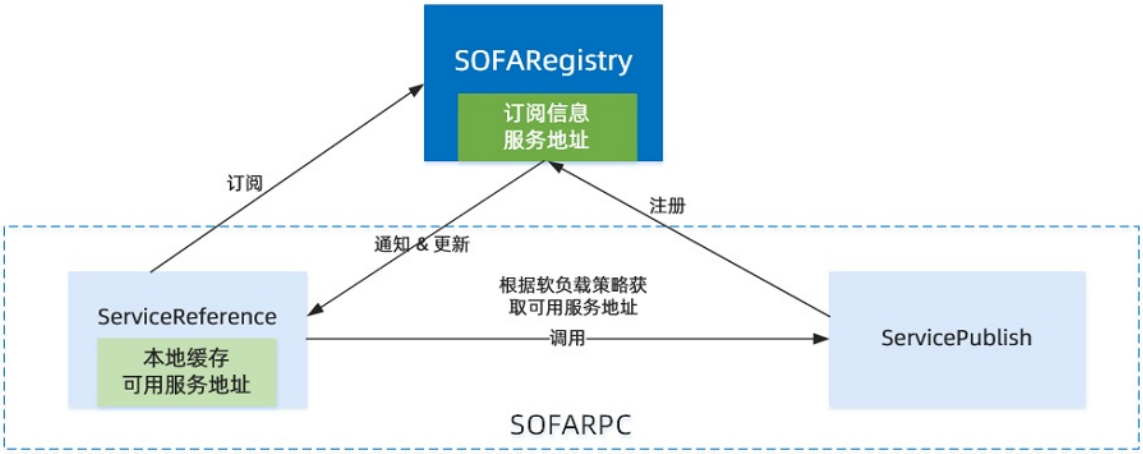
软负载

软负载即软件负载。当需要调用服务时，消费方根据软负载策略，从服务注册中心（SOFARegistry）推送到本地缓存的列表里选择一个地址，再调用该地址所提供的服务。

SOFARPC 采用服务发布（ServicePublish）和引用（ServiceReference）模型，通过服务注册中心（SOFARegistry）动态感知服务发布，并将服务地址列表推送给已经引用该服务的消费方，更新消费方本地缓存中的可用服务列表，最后通过软负载策略，为消费方选择可用地址进行远程通信。

通过路由策略，您在使用 SOFARPC 的时候，就不用将服务地址硬编码在服务注册中心的代码中。详情请参见 负载均衡。

软负载示意图



直连

在开发及测试环境下，开发者经常需要绕过注册中心，只测试指定服务提供方，这时候可能需要点对点直连来进行测试。该功能可以通过 SOFARPC 的 `test-url` 或 `target-url` 配置来实现。详情请参见 直连调用。

6.2. 负载均衡

本文介绍 SOFARPC 支持的负载均衡算法以及如何设置负载均衡。

负载均衡算法

SOFARPC 目前支持以下五种算法：

类型	名称	描述
random	随机算法	默认负载均衡算法。
localPref	本地优先算法	优先发现是否本机发布了该服务，如果没有再采用随机算法。
roundRobin	轮询算法	方法级别的轮询，各个方法间各自轮询，互不影响。
consistentHash	一致性 Hash 算法	服务请求与处理请求的服务器之间可以一一对应。
weightRoundRobin	按权重负载均衡轮询算法	按照权重对节点进行轮询。性能较差，不推荐使用。

设置负载均衡

要使用某种特定的负载均衡算法，可以按照以下的方式进行设置：

- **XML 方式**

  如果使用 XML 的方式引用服务，可以通过设置 `sofa:global-attrs` 标签的 `loadBalancer` 属性来设置负载均衡。以下示例以设置负载均衡算法为 roundRobin 为例：

```
<sofa:reference interface="com.example.demo.SampleService" id="sampleService">
    <sofa:binding.bolt>
        <sofa:global-attrs loadBalancer="roundRobin"/>
    </sofa:binding.bolt>
</sofa:reference>
```

- **在 Spring 环境下 API 方式**

  如果在 Spring 或者 Spring Boot 的环境下使用 API，可以通过调用 `BoltBindingParam` 的 `setLoadBalancer` 方法来设置。以下示例以设置负载均衡算法为 roundRobin 为例：

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setLoadBalancer("roundRobin");
```

- **非 Spring 环境下 API 方式**

  如果在非 Spring 环境下直接使用 SOFARPC 提供的裸 API，可以通过调用 `ConsumerConfig` 的 `setLoadBalancer` 方法来设置。以下示例以设置负载均衡算法为 random 为例：

```
ConsumerConfig consumerConfig =new ConsumerConfig();
consumerConfig.setLoadbalancer("random");
```

6.3. 直连调用

SOFARPC 支持指定地址进行调用的场景，此方式允许用户指定服务端地址。本文介绍直连调用的应用场景和配置方式。

注意事项

- 当直连配置生效时，注册中心软负载将会失效。
- 直连地址允许配置多个地址，多个地址之间使用英文逗号（,）或英文分号（;）分隔，多个地址间支持负载均衡。

应用场景

具体使用场景分为以下 2 种：

- 线上场景

  在 XML 文件中配置服务引用时，在标签 `sofa:binding.bolt` 里面加上一个 `route` 标签。`route` 标签中放入一个 `target-url` 属性，属性值设置为需要调用的地址。

```
<sofa:reference id="sampelService" interface="com.alipay.test.SampleService">
     <sofa:binding.bolt>
         <sofa:route target-url="${targetUrl}:port"/>
     </sofa:binding.bolt>
</sofa:reference>
```

  `${targetUrl}:port` 需修改为实际调用的地址和端口。更多配置方式，请参见 配置方式。

- 测试场景
  - 在 `application.properties` 中配置 `run_mode=TEST` 。
  - 在 XML 文件中配置服务引用时，在标签 `sofa:binding.bolt` 里面加上一个 `global-attrs` 标签，里面放入一个 `test-url` 的属性，属性值设置为需要调用的地址。

```
<sofa:reference id="sampleService" interface="com.alipay.test.SampleService">
     <sofa:binding.bolt>
         <sofa:global-attrs test-url="${targetUrl}:port"/>
     </sofa:binding.bolt>
</sofa:reference>
```

> 注意
>
> - `test-url` 仅适用于 XML 的配置方式。
> - `test-url` 必须配合 `run_mode=TEST` 才会生效，主要用于测试阶段。

配置方式

- API 方式

  通过 API 方式配置直连调用，地址格式为： `协议://IP:端口` 。配置示例如下：

```
ConsumerConfig<HelloService> consumer =new ConsumerConfig<HelloService>()
.setInterfaceId(HelloService.class.getName())
.setRegistry(registryConfig)
.setDirectUrl("bolt://127.0.0.1:12201");
```

- Annotation 方式

  通过 Annotation 方式配置直连调用，地址格式为： `IP:端口` 。配置示例如下：

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt", directUrl ="127.0.0.1:
12220"))
private SampleService sampleService;
```

- XML 方式

  通过 XML 方式配置直连调用，地址格式为： `IP:端口` 。配置示例如下：

```
<sofa:reference id="sampleService" interface="com.alipay.test.SampleService">
     <sofa:binding.bolt>
         <sofa:global-attrs target-url="127.0.0.1:12200"/>
     </sofa:binding.bolt>
</sofa:reference>
```

6.4. 注册中心路由

软负载的情况下，消费方会从注册中心做服务发现，默认选择蚂蚁的注册中心 DSR。您也可以手动指定服务的注册中心。

注册中心

目前支持的注册中心如下：

注册中心 alias		注册中心名称
普通注册中心	dsr	蚂蚁注册中心（企业版）
	sofa	蚂蚁注册中心（开源版）
	consul	开源注册中心 Consul
	zookeeper	开源注册中心 ZooKeeper

注册中心 alias		注册中心名称
	nacos	开源注册中心 Nacos
特殊注册中心	mesh	Mesh 模式，使用蚂蚁 MOSN
	gateway	网关模式，使用网关服务发现
	local	本地模式，本地调试使用
	multicast	广播模式

配置方式

- 为全局服务指定默认注册中心

  您可以在 `application.properties` 或启动参数中增加如下参数，为全局服务指定注册中心。

  - 默认使用 DSR 注册中心

    ```
    run.mode=normal
    ```

  - 默认使用本地模式

    ```
    run.mode=dev
    ```

  - 使用 Mesh 模式

    ```
    MOSN_ENABLE=true
    ```

- 为单个服务指定注册中心

  您可以在发布和订阅服务的代码中加入 `<sofa:global-attrs registry="<注册中心 alias>" />` 字段指定注册中心，示例如下：

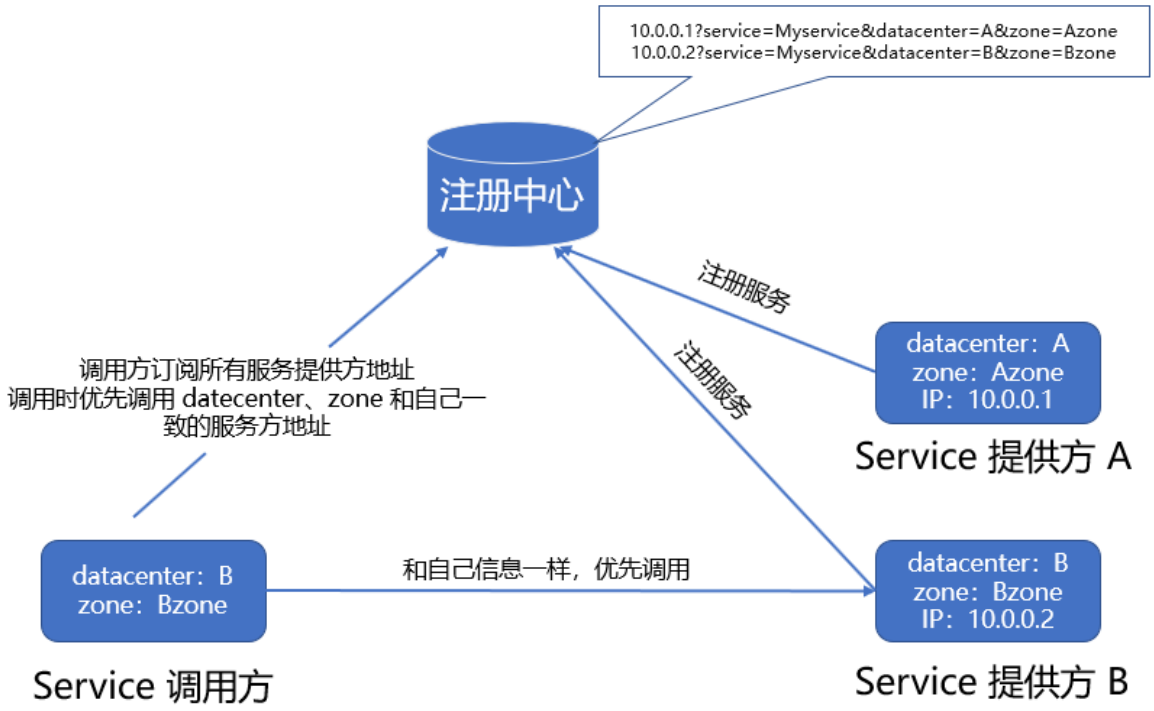
```
<bean id="helloSyncServiceImpl" class="com.alipay.sofa.rpc.samples.invoke.HelloSyncServic
eImpl"/>
<sofa:service ref="helloSyncServiceImpl" interface="com.alipay.sofa.rpc.samples.invoke.He
lloSyncService"unique-id="">
<sofa:binding.bolt>
<sofa:global-attrs registry="mesh" />
</sofa:binding.bolt>

</sofa:service>
```

6.5. 同机房路由收敛

当存在多个机房时，RPC 请求可能会跨机房调用。受限于网络等因素，跨机房调用增加调用耗时。为保证业务质量，建议您配置同机房路由收敛，让 RPC 请求优先调用同机房服务器提供的服务。

调用流程



> **说明**
>
> 以上数据结构为讲解原理进行了简化，并非真实的结构。

- datacenter：服务器所在的物理机房。
- zone：物理机房为了方便管理划分出来的逻辑单元，datacenter + zone 共同标识服务器位置信息。

调用流程如下：

1. 服务提供方 A 和 B 将自己的服务注册到注册中心。
2. 服务调用方会订阅注册中心所有服务提供方的地址信息。
3. 当调用方调用服务时，会优先调用 datacenter、zone 和自身配置一致的服务方地址。
4. 如果当前 zone 没有提供服务，则调用其他 zone 的服务。

配置方式

在服务启动参数中增加服务的位置信息：

```
-Dcom.alipay.ldc.zone=xxx
-Dcom.alipay.ldc.datacenter=xxx
```

同机房路由收敛默认开启，如果没有配置 datacenter、zone，则系统使用默认值，即所有服务的 datacenter、zone 一样，同机房路由收敛不会生效。

注意事项

- 同机房路由收敛会优先调本 zone 的服务，所以在部署时要平衡各个 zone 的服务数量，避免出现某个 zone 流量大，服务少的情况。
- datacenter + zone 标识唯一位置信息，但目前某些版本对 datacenter 不感知，所以不同 datacenter 内也要保持 zone 名不同。
- datacenter 和 zone 在阿里云上对应 datacenter 和 cell，为了方便维护，尽量不要自己命名。

6.6. 单元化配置

本文将引导您如何在本地客户端配置单元化，发布与引用 RPC 服务。

> **说明**
>
> 该功能仅适用于开启了单元化能力的用户。

升级依赖

您需要将 SOFABoot 版本升级到 3.3.0 或以上，详见 SOFABoot 版本说明。

```
<parent>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>sofaboot-enterprise-dependencies</artifactId>
    <version>3.3.0</version>
</parent>
```

应用参数配置

在本地项目的全局配置文件 `application.properties` 中，配置以下参数：

- 配置 AntVIP、instanceid、AccessKey ID、AccessKey Secret。配置说明，请参见 配置项说明。
- 配置 Zone 信息，示例如下：

```
com.alipay.ldc.zone=LCD-Test-A-1
com.alipay.ldc.datacenter=LCD-Test-A
//开启严格模式的 LDC。
com.alipay.ldc.strictmode=true
zmode=true
//需要确保您配置的 zonemng-pool 能 ping 通。
zonemng_zone_url=http://zonemng-pool
```

服务发布

服务发布，如果您没有特殊要求，跟随部署的 Zone 进行发布即可。如在 RZone 部署，则服务就发布在 RZone；如在 CZone 部署，服务就发布在 CZone；如果两个 Zone 都部署，那么都发布。

默认情况下，服务全 Zone 发布，如果您需要指定在某些 Zone 发布，则配置如下 DRM 动态配置即可。

- 资源：`com.alipay.sofa.rpc.ldc.route.drm.config`
- 字段：`publishConfig`
- 内容：`[{"serviceName":"interface:1.0:uniqueId","cell":"RZ,CZ"}]`

比如，您当前的 Zone 是 RZ00A，则通过上面的 DRM 提前配置，您的服务会在 RZ00A 发布出来，而如果您的 Zone 名是 TZ00A，则不会发布。

配置好之后，进行应用重启，即可看到效果。目前暂不支持运行中动态变更该信息。一旦修改，必须重启生效。

服务引用

目前，一旦配置了 `com.alipay.ldc.strictmode=true` （严格模式），则路由的时候，要求入参的第一个参数必须为 `userId` 路由信息。RPC 框架计算出倒数第二、三位进行路由。如果不是，则会认为没有 LDC 逻辑，走本 Zone 优先。

如果在某些情况下，您认为以上的默认规则不足，也支持更高级的用法（支持方法维度）：

- 资源：`com.alipay.sofa.rpc.ldc.route.drm.config`
- 字段：`routeConfig`

如下所示，先匹配方法，然后匹配接口。会根据第一个参数的第二、三位进行计算，例如对于 `123` ，则计算为 `23` 。

```
[
    {
        "serviceName":"interface:1.0:uniqueId",
        "rule":"string.substring(args[0], 1, 3)"
    },
    {
        "serviceName":"interface:1.0:uniqueId:methodA",
        "rule":"string.substring(args[0], 1, 3)"
    }
]
```

如果是复杂对象，需要使用如下配置：

```
[
    {
        "serviceName":"interface:1.0:uniqueId",
        "rule":"string.substring(#args.[0].uid, 1, 3)"
    }
]
```

如是简单对象，也可以使用如下配置：

```
[
    {
        "serviceName":"interface:1.0:uniqueId",
        "rule":"string.substring(#args.[0], 1, 3)"
    }
]
```

7.路由容错

7.1. 调用重试

SOFARPC 支持进行框架层面的重试策略，前提是集群模式为 FailOver（SOFARPC 默认为 FailOver 模式）。重试只有在服务端的框架层面异常或超时异常时才会发起，不会对业务抛出的异常进行重试。默认情况下 SOFARPC 不进行任何重试。

> 注意
>
> 超时异常虽然可以重试，但是需要服务端保证业务的幂等性，否则可能会有风险。

XML 方式

如果使用 XML 方式订阅服务，可以设置 `sofa:global-attrs` 的 `retries` 参数来设置重试次数：

```
<sofa:reference jvm-first="false" id="retriesServiceReferenceBolt" interface="com.alipay.sofa.rpc.samples.retries.RetriesService">
    <sofa:binding.bolt>
        <sofa:global-attrs retries="2"/>
    </sofa:binding.bolt>
</sofa:reference>
```

Annotation 方式

如果是使用 Annotation 的方式，可以通过设置 `@SofaReferenceBinding` 注解的 `retries` 属性来设置重试次数：

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt", retries =2))
private SampleService sampleService;
```

Spring 环境下 API 方式

如果是在 Spring 环境下用 API 的方式，可以调用 `BoltBindingParam` 的 `setRetries` 方法来设置重试次数：

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setRetries(2);
```

非 Spring 环境下 API 方式

如果是在非 Spring 环境下直接使用 SOFARPC 的裸 API 的方式，可以通过调用 `ConsumerConfig` 的 `setRetries` 方法来设置重试次数：

```
ConsumerConfig<RetriesService> consumerConfig =new ConsumerConfig<RetriesService>();
consumerConfig.setRetries(2);
```

7.2. 预热转发

集群中某台机器刚启动的一段时间（称之为“预热期”）内，如果收到的请求过多，可能会影响机器性能和正常业务。RPC 框架可通过预热转发功能将处于预热期机器的请求转发给集群内其它机器，当机器过了预热期后再恢复正常。

您可以手动指定预热期内请求转发的比例，比如 80% 的请求转发出去，20% 自身系统处理。

与 RPC 压测转发不同的是，RPC 预热转发仅适用于应用启动后的一段时间，而压测转发则是长期生效。

注意事项

要使用预热转发功能时，您需要注意以下要求：

- 服务端在同一台主机上启动时，预热转发不生效。
- 先启动的服务还在预热中时，预热转发不生效。
- 先启动的服务没有配置预热转发时，后启动的服务不会转发到没配置预热转发的机器。
- 所有预热转发的机器，BOLT 端口必须一致。

RPC 转发配置

配置的方式有两种：

- 直接转发到 IP

  在 `application.properties` 增加如下参数：

  ```
  rpc.transmit.url = 10.**.**.2
  ```

  数据无论何时都直接转发到指定 IP，和 `rpc.transmit.url=address:10.**.**.2` 效果同样。

- 配置预热与权重

  在 `application.properties` 增加如下参数：

  ```
  rpc.transmit.url=weightStarting:0.3,during:60,weightStarted:0.2,address:10.**.**.2,uniqueId:core_unique
  ```

  - `weightStarting`：预热期内的转发权重或概率。RPC 框架内部会在集群中随机找一台机器以此权重转出或接收。
  - `during`：预热期的时间长度。单位为秒。
  - `weightStarted`：预热期过后的转发权重。将会一直生效。
  - `address`：预热期过后的转发地址。将会一直生效。
  - `uniqueId`：同 appName 多集群部署的情况下，要区别不同集群时可配置此项。指定一个自定义的系统变量，保证集群唯一即可。`core_unique` 是一个 `application.properties` 的配置，可以动态替换。

> 说明
>
> 在 `application.properties` 中可以配置转发请求超时时间，例如 `rpc_transmit_url_timeout_tr=8000` 。单位为 ms，默认为 10000 ms。

7.3. 容灾恢复

集群中通常一个服务有多个服务提供者，其中部分服务提供者可能由于网络、配置、长时间 fullgc、线程池满、硬件故障等导致长连接还存活但是程序已经无法正常响应。单机故障剔除功能会将这部分异常的服务提供者进行降级，使客户端的请求更多地指向健康节点。当异常节点的表现正常后，单机故障剔除功能会对该节点进行恢复。解决了服务故障持续影响业务的问题，避免了雪崩效应，提高系统可用率。

功能原理

单机故障剔除会统计一个时间窗口内的调用次数和异常次数，并计算每个服务对应 IP 的异常率和该服务的平均异常率。当 IP 的异常率大于服务平均异常率，且达到一定比例时，单机故障剔除会对该服务 + IP 的维度进行权重降级。如果该服务 + IP 维度的权重并没有降为 0，那么当该服务 + IP 维度的调用情况正常时，则会对其进行权重恢复。整个计算和调控过程异步进行，不会阻塞调用。

使用方式

配置示例如下：

```
FaultToleranceConfig faultToleranceConfig =new FaultToleranceConfig();
faultToleranceConfig.setRegulationEffective(true);
faultToleranceConfig.setDegradeEffective(true);
faultToleranceConfig.setTimeWindow(20);
faultToleranceConfig.setWeightDegradeRate(0.5);

FaultToleranceConfigManager.putAppConfig("appName", faultToleranceConfig);
```

以上配置完成后，目标应用会在每 20s 的时间窗口进行一次异常情况计算，如果某个服务 + IP 的调用维度被判定为故障节点，则会将该服务 + IP 的权重降级为 0.5。更多信息，请参见 自动故障剔除。

7.4. 自动故障剔除

自动故障剔除功能会自动监控 RPC 调用的情况，当某个节点出现故障时，可对故障节点进行权重降级，并在节点恢复健康时进行权重恢复。目前支持 Bolt 协议。

配置方式

将自动故障剔除的参数配置到 SOFABoot 中的 `application.properties` 即可。参数说明如下：

参数	默认值	说明
com.alipay.sofa.rpc.aft.time.window	10s	时间窗口：用于设定统计信息计算的周期。

参数	默认值	说明
com.alipay.sofa.rpc.aft.least.window.count	10 次	时间窗口内最少调用数：只有在时间窗口内达到了该最低值的数据才会被加入到计算和调控中。
com.alipay.sofa.rpc.aft.least.window.exception.rate.multiple	6 倍	时间窗口内异常率与服务平均异常率的降级比值：在对统计信息进行计算的时候，会计算出该服务所有有效调用 IP 的平均异常率，如果某个 IP 的异常率大于等于了这个最低比值，则会被降级。
com.alipay.sofa.rpc.aft.weight.degrade.rate	1/20	降级比率：地址在进行权重降级时的降级比率。
com.alipay.sofa.rpc.aft.weight.recover.rate	2 倍	恢复比率：地址在进行权重恢复时的恢复比率。
com.alipay.sofa.rpc.aft.degrade.effective	false	降级开关：如果应用打开了这个开关，则会对符合降级的地址进行降级，否则只会进行日志打印。
com.alipay.sofa.rpc.aft.degrade.least.weight	1	降级最小权重：地址权重被降级后的值如果小于这个最小权重，则会以该最小权重作为降级后的值。
com.alipay.sofa.rpc.aft.degrade.max.ip.count	2	降级的最大 IP 数：同一个服务被降级的 IP 数不能超过该值。
com.alipay.sofa.rpc.aft.regulation.effective	false	全局开关：如果应用打开了这个开关，则会开启整个单点故障自动剔除功能，否则该功能不启用。

说明

- 每个参数都有默认值，您可以根据需要自行修改参数值。
- `com.alipay.sofa.rpc.aft.regulation.effective` 是该功能的全局开关，如果关闭则该功能不会运行，其他参数也都不生效。

配置示例

```
com.alipay.sofa.rpc.aft.time.window=20
com.alipay.sofa.rpc.aft.least.window.count=30
com.alipay.sofa.rpc.aft.least.window.exception.rate.multiple=1.4
com.alipay.sofa.rpc.aft.weight.degrade.rate=0.5
com.alipay.sofa.rpc.aft.weight.recover.rate=1.2
com.alipay.sofa.rpc.aft.degrade.effective=true
com.alipay.sofa.rpc.aft.degrade.least.weight=1
com.alipay.sofa.rpc.aft.degrade.max.ip.count=2
com.alipay.sofa.rpc.aft.regulation.effective=true
```

对示例配置，说明如下：

- 已打开自动故障剔除功能和降级开关，当节点出现故障时会被进行权重降级，在恢复时会被进行权重恢复。
- 每隔 20s 进行一次节点健康状态的度量，20s 内调用次数超过 30 次的节点才被作为计算数据。
- 如果单个节点的异常率超过了所有节点的平均异常率的 1.4 倍，则对该节点进行权重降级，降级的比率为 0.5。
- 如果单个节点的异常率低于平均异常率的 1.4 倍，则对该节点进行权重恢复，恢复的比率为 1.2。
- 权重最小降级到 1。
- 单个服务最多降级 2 个 IP。

8.通信协议

8.1. 多协议发布

在 SOFARPC 中，同一个服务可以多协议发布，并按不同协议被客户端调用。主要调用方式有下述几种：

- Java API 方式

  可以按照如下的代码构建多个 `ServerConfig`，不同的 `ServerConfig` 设置不同的协议，然后将这些 `ServerConfig` 设置给 `ProviderConfig`。

```
List<ServerConfig> serverConfigs =new ArrayList<ServerConfig>();
serverConfigs.add(serverConfigA);
serverConfigs.add(serverConfigB);
providerConfig.setServer(serverConfigs);
```

- XML 方式

  直接在 `<sofa:service>` 标签中增加多个 `binding` 即可：

```
<sofa:service ref="sampleFacadeImpl" interface="com.alipay.sofa.rpc.bean.SampleFacade">
    <sofa:binding.bolt/>
    <sofa:binding.rest/>
    <sofa:binding.dubbo/>
</sofa:service>
```

- Annotation 方式

  在 `@SofaService` 中增加多个 `binding` 即可：

> **说明**
>
> 如果使用 rest 协议，接口需要要加 path。

```
@SofaService(
      interfaceType =SampleService.class,
      bindings ={
@SofaServiceBinding(bindingType ="rest"),//使用 rest 协议，接口需要要加 path。
@SofaServiceBinding(bindingType ="bolt")
}
)
public class SampleServiceImpl implements SampleService{
// ...
}
```

8.2. Bolt 协议

8.2.1. Bolt 协议的基本用法

本文介绍如何发布和引用 Bolt 协议服务。

发布服务

使用 SOFARPC 发布一个 Bolt 协议的服务，只需要增加名称为 Bolt 的 Binding 即可，不同的使用方式添加 Bolt Binding 的方式如下：

- XML

  使用 XML 发布一个 Bolt 协议只需在 `<sofa:service>` 标签下增加 `<sofa:binding.bolt>` 标签。

```
<sofa:service ref="sampleService" interface="com.alipay.sofa.rpc.sample.SampleService">
    <sofa:binding.bolt/>
</sofa:service>
```

- Annotation

  使用 Annotation 发布一个 Bolt 协议的服务只需设置 `@SofaServiceBinding` 的 `bindingType` 为 `bolt`。

```
@Service
@SofaService(bindings ={@SofaServiceBinding(bindingType ="bolt")})
public class SampleServiceImpl implements SampleService{
}
```

- Spring 环境下 API 方式

  在 Spring 或者 Spring Boot 环境下发布一个 Bolt 协议的服务只需往 `ServiceParam` 里面增加一个 `BoltBindingParam`。

```
ServiceParam serviceParam =new ServiceParam();
serviceParam.setInterfaceType(SampleService.class);// 设置服务接口。
serviceParam.setInstance(new SampleServiceImpl());// 设置服务接口的实现。

List<BindingParam> params =new ArrayList<BindingParam>();
BindingParam serviceBindingParam =new BoltBindingParam();
params.add(serviceBindingParam);
serviceParam.setBindingParams(params);
```

- 非 Spring 环境下的 API 方式

  在非 Spring 环境下使用 SOFARPC 的裸 API 提供 Bolt 协议的服务，只需要将 Protocol 为 Bolt 的 `ServerConfig` 设置给对应的 `ProviderConfig`。

```
RegistryConfig registryConfig =new RegistryConfig()
.setProtocol("zookeeper")
.setAddress("127.0.0.1:2181");
// 新建一个协议为 Bolt 的 ServerConfig。
ServerConfig serverConfig =new ServerConfig()
.setPort(8803)
.setProtocol("bolt");
ProviderConfig<SampleService> providerConfig =new ProviderConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRef(new SampleServiceImpl())
// 将 ServerConfig 设置给 ProviderConfig，表示这个服务发布的协议为 Bolt。
.setServer(serverConfig)
.setRegistry(registryConfig);
providerConfig.export();
```

引用服务

使用 SOFARPC 引用一个 Bolt 服务，只需要增加名称为 Bolt 的 Binding 即可。不同发布方式中添加 Bolt Binding 的方法如下：

- XML

  使用 XML 引用一个 Bolt 协议的服务只需在 `<sofa:reference>` 标签下增加 `<sofa:binding.bolt>` 标签。

```
<sofa:reference id="sampleService" interface="com.alipay.sofa.rpc.sample.SampleService">
<sofa:binding.bolt/>
</sofa:reference>
```

- Annotation

  使用 Annotation 引用一个 Bolt 协议的服务只需设置 `@SofaReferenceBinding` 的 `bindingType` 为 `bolt` 。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt"))
private SampleService sampleService;
```

- Spring 环境下 API 方式

  在 Spring 或者 Spring Boot 环境下引用一个 Bolt 协议的服务，只需往 `ReferenceParam` 里面增加一个 `BoltBindingParam` 。

```
ReferenceClient referenceClient = clientFactory.getClient(ReferenceClient.class);
ReferenceParam<SampleService> referenceParam =new ReferenceParam<SampleService>();
referenceParam.setInterfaceType(SampleService.class);

BindingParam refBindingParam =new BoltBindingParam();
referenceParam.setBindingParam(refBindingParam);
```

- 非 Spring 环境下的 API 方式

在非 Spring 环境下使用 SOFARPC 的裸 API 引用一个 Bolt 协议的服务，只需设置 `ConsumerConfig` 的 Protocol 为 `bolt` 。

```
ConsumerConfig<SampleService> consumerConfig =new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRegistry(registryConfig)
.setProtocol("bolt");
SampleService sampleService = consumerConfig.refer();
```

8.2.2. Bolt 协议的调用方式

SOFARPC 在 BOLT 协议下提供了多种调用方式满足不同的场景。

同步

在同步调用方式下，客户端发起调用后会等待服务端返回结果再进行后续的操作。这是 SOFARPC 的默认调用方式，无需进行任何设置。

异步

设置调用方式

异步调用的方式下，客户端发起调用后不会等到服务端的结果，而是继续执行后面的业务逻辑。服务端返回的结果会被 SOFARPC 缓存，当客户端需要结果的时候，再主动调用 API 获取。如果您需要将一个服务设置为异步的调用方式，可以在对应的使用方式下设置 `type` 属性。

- XML 方式

  在 XML 方式下，设置 `<sofa:global-attrs>` 标签的 `type` 属性为 `future` 。

```
<sofa:reference interface="com.example.demo.SampleService" id="sampleService">
     <sofa:binding.bolt>
         <sofa:global-attrs type="future"/>
     </sofa:binding.bolt>
</sofa:reference>
```

- Annotation 方式

  在 Annotation 方式下，设置 `@SofaReferenceBinding` 的 `invokeType` 属性为 `future` 。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt", invokeType ="future"))
private SampleService sampleService;
```

- Spring 环境下 API 方式

  在 Spring 环境下使用 API，设置 `BoltBindingParam` 的 `type` 属性为 `future` 。

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setType("future");
```

- **在非 Spring 环境下 API 方式**

  在非 Spring 环境下使用 SOFARPC 的裸 API，设置 `ConsumerConfig` 的 `invokeType` 属性为 `future`。

```
ConsumerConfig<SampleService> consumerConfig =new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRegistry(registryConfig)
.setProtocol("bolt")
.setInvokeType("future");
```

获取调用结果

目前提供两种方式来获取异步调用的结果：

- **直接获取结果**

  您可以通过以下的方式来直接获取异步调用的结果：

```
String result =(String)SofaResponseFuture.getResponse(0,true);
```

  其中，`0` 表示获取结果的超时时间，您可以根据需要修改；`true` 表示清除线程上下文中的结果。若您不希望清除结果，可以修改为 `false`。

- **获取 JDK 原生 Future**

  您可以通过以下的方式来获取 JDK 的原生的 Future 对象，再从任意地方去调用这个 Future 对象来获取结果。

```
Future future =SofaResponseFuture.getFuture(true);
```

  `true` 表示清除线程上下文中的结果。若您不希望清除，可以修改为 `false`。

回调

SOFARPC Bolt 协议的回调方式可以让 SOFARPC 在发起调用后不等待结果，在客户端收到服务端返回的结果后，自动回调用户实现的一个回调接口。

使用 SOFARPC Bolt 协议的回调方式，首先需要实现一个回调接口，并且在对应的配置中设置回调接口，再将调用方式设置为 `callback`。

实现回调接口

SOFARPC 提供了一个回调的接口 `com.alipay.sofa.rpc.core.invoke.SofaResponseCallback`。使用 SOFARPC Bolt 协议的回调方式时，需要先实现这个接口，该接口提供以下三个方法：

- `onAppResponse`：当客户端接收到服务端的正常返回的时候，SOFARPC 会回调这个方法。

- `onAppException`：当客户端接收到服务端的异常响应的时候，SOFARPC 会回调这个方法。
- `onSofaException`：当 SOFARPC 本身出现一些错误时（例如路由错误），SOFARPC 会回调这个方法。

设置回调接口

实现回调接口之后，您需要将实现类设置到对应的服务引用配置中，并且将调用方式设置为 callback。SOFARPC 为设置回调接口提供了 Callback Class 和 Callback Ref 两种方式：

- Callback Class：直接设置回调的类名，SOFARPC 会通过调用回调类的默认构造函数的方式生成回调类的实例。
- Callback Ref：为用户直接提供回调类的实例。

通过不同使用方式设置回调接口的代码如下：

- **XML 方式**

  如果通过 XML 的方式引用服务，将 `<sofa:global-attrs>` 标签的 `type` 属性设置为 `callback`，并且设置 `callback-ref` 或者 `callback-class` 属性。以 `callback-ref` 为例，代码如下：

```
<bean id="sampleCallback" class="com.example.demo.SampleCallback"/>
<sofa:reference interface="com.example.demo.SampleService" id="sampleService">
    <sofa:binding.bolt>
        <sofa:global-attrs type="callback" callback-ref="sampleCallback"/>
    </sofa:binding.bolt>
</sofa:reference>
```

  在 XML 的方式下，`callback-ref` 的值需要是回调类的 Bean 名称。

- **Annotation 方式**

  如果通过 Annotation 的方式引用服务，设置 `@SofaReferenceBinding` 注解的 `invokeType` 属性为 `callback`，并且设置 `callbackClass` 或者 `callbackRef` 属性。以 `callback-ref` 为例，代码如下：

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt",
            invokeType ="callback",
            callbackRef ="sampleCallback"))
private SampleService sampleService;
```

  在 Annotation 的方式下，`callbackRef` 属性的值需要是回调类的 Bean 名称。

- **Spring 环境 API 方式**

  如果在 Spring 或者 Spring Boot 的环境下使用 API 的方式，设置 `BoltBindingParam` 的 `type` 属性为 `callback`，并且设置 `callbackClass` 或者 `callbackRef` 属性即可。以 `callbackClass` 为例，代码如下：

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setType("callback");
boltBindingParam.setCallbackClass("com.example.demo.SampleCallback");
```

- **非 Spring 环境下 API 方式**

  如果在非 Spring 环境下使用 SOFARPC 的裸 API，需将 `setInvokeType` 类型设置成 `callback`，并且调用 `setOnReturn` 设置回调类。

```
ConsumerConfig<SampleService> consumerConfig =new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRegistry(registryConfig)
.setProtocol("bolt")
.setInvokeType("callback")
.setOnReturn(newSampleCallback());
```

在调用级别设置回调接口

除了在服务级别设置回调接口之外，您还可以在调用级别设置回调接口，方式如下：

```
RpcInvokeContext.getContext().setResponseCallback(newSampleCallback());
```

单向

当客户端发送请求后不关心服务端返回的结果时，您可以使用单向的调用方式。这种方式会在发起调用后立即返回 null，并且忽略服务端的返回结果。

使用单向的方式只需将调用方式设置为 `oneway` 即可，设置方式和将调用方式设置为 `future` 或者 `callback` 一样，您可以参考上面的文档中提供的设置方式。需要特别注意的是，由于单向的调用方式会立即返回，所有的超时设置在单向的情况下都是无效的。

8.2.3. 配置 Bolt 服务

Bolt 服务的名称源自 RPC 使用的底层通信框架 Bolt。相对于传统的 WebService，Bolt 支持更加复杂的对象，序列化后的对象更小，且提供了更为丰富的调用方式（sync、oneway、callback、future 等），支持更广泛的应用场景。

在 SOFA 中，Bolt 服务提供方使用的端口是 12200，详情请参见 配置说明。

发布及引用 Bolt 服务

SOFA 中的 RPC 通过 Binding 模型定义不同的通信协议，每接入一种新的协议，就会增加一种 Binding 模型。要在 SOFA 中添加 RPC 的 Bolt 协议的实现，就需要在 Binding 模型中增加一个 `<sofa:binding.bolt/>`。

- 发布一个 Bolt 的服务需在 `<sofa:service>` 中添加 `<sofa:binding.bolt/>`，示例如下：

```
<!-- 发布 Bolt 服务 -->
<sofa:service interface="com.alipay.test.SampleService" ref="sampleService" unique-id="se
rvice1">
     <sofa:binding.bolt/>
</sofa:service>
```

- 引用一个 Bolt 的服务需在 `<sofa:reference>` 中添加 `<sofa:binding.bolt/>`，示例如下：

```
<!-- 引用 Bolt 服务 -->
<sofa:reference interface="com.alipay.test.SampleService" id="sampleService">
     <sofa:binding.bolt/>
</sofa:reference>
```

Bolt 服务完整配置

Bolt 配置标签主要有 `global-attrs` 和 `method` 两种，同时配置时，以 `method` 中的配置为准。可配置的参数如下：

- `type`：调用方式。取值为 sync、oneway、callback、future，默认为 sync。
- `timeout`：超时时间。默认为 3000 ms。
- `callback-class`：调用方式为 callback 时的回调对象类。
- `callback-ref`：调用方式为 callback 时的回调 Spring Bean 引用。

Bolt 服务发布完整配置

```
<sofa:service interface="com.alipay.test.SampleService" ref="sampleService" unique-id="serv
ice1">
    <sofa:binding.bolt>
        <sofa:global-attrs timeout="5000"/>
        <!-- 此处方法名 service 为示例，需要替换成实际方法名。 -->
        <sofa:method name="service" timeout="3000"/>
    </sofa:binding.bolt>
</sofa:service>
```

Bolt 服务引用完整配置

```
<sofa:reference id="sampleService" interface="com.alipay.test.SampleService" unique-id="ser
vice1">
    <sofa:binding.bolt>
        <sofa:global-attrs timeout="5000" test-url="localhost:12200" address-wait-time="100
0"
            connect.timeout="1000" connect.num="-1" idle.timeout="-1" idle.timeout.read="-1
"/>
        <!-- method 配置的属性优先级高于 global-attrs 中的配置。 -->
        <!-- 此处方法名 futureMethod 和 callbackMethod 为示例，需要替换成实际方法名。 -->
        <sofa:method name="futureMethod" type="future" timeout="5000"/>
        <sofa:method name="callbackMethod" type="callback" timeout="3000"
            callback-class="com.alipay.test.TestCallback"/>
    </sofa:binding.bolt>
</sofa:reference>
```

Bolt 服务提供方配置

Bolt 的底层服务提供方是一个 Java NIO Server (Non-blocking I/O Server)，SOFA 框架提供几个选项来调整 Bolt Server 的一些属性。详情请参见 配置说明。

Bolt 服务消费方配置

Bolt 引用调用方式

Bolt 提供多种调用方式，以满足各种业务场景的需求。目前，Bolt 提供的调用方式有以下几种：

调用方式	类型	说明
sync	同步	Bolt 默认的调用方式。
oneway	异步	消费方发送请求后直接返回，忽略提供方的处理结果。
callback	异步	消费方提供一个回调接口，当提供方返回后，SOFA 框架会执行回调接口。
future	异步	消费方发起调用后马上返回，当需要结果时，消费方需要主动去获取数据。

- sync 调用

  Bolt 默认的调用方式，支持在服务提供方和消费方配置超时时间：

- 服务提供方

  XML 配置如下：

```
<!-- 服务方超时配置 -->
<sofa:service interface="com.alipay.test.SampleService2" ref="sampleService2">
    <sofa:binding.bolt>
    <!-- 此处方法名 service 为示例，需要替换成实际方法名。 -->
    <sofa:method name="service" timeout="5000"/>
    </sofa:binding.bolt>
</sofa:service>
```

- 服务消费方

  XML 配置如下：

```
<!-- 消费方超时配置 -->
<sofa:reference interface="com.alipay.test.SampleService2" id="sampleServiceSync">
    <sofa:binding.bolt>
         <!-- 超时全局配置 -->
          <sofa:global-attrs timeout="8000" test-url="127.0.0.1:12200"/>
          <!-- 如果配置了 method ，优先使用 method 配置。 -->
          <!-- 此处方法名 service 为示例，需要替换成实际方法名。 -->
          <sofa:method name="service" type="sync" timeout="10000"/>
    </sofa:binding.bolt>
</sofa:reference>
```

  `type` 的默认取值为 `sync` 。如果您有多个方法需要配置超时时间，并且超时时间都相同，也可以设置全局的超时时间。XML 配置如下：

```
<!-- 消费方批量配置超时时间 -->
<sofa:reference interface="com.alipay.test.SampleService2" id="sampleServiceSync">
     <sofa:binding.bolt>
          <sofa:global-attrs timeout="5000" test-url="127.0.0.1:12200"/>
     </sofa:binding.bolt>
</sofa:reference>
```

  > **说明**
  >
  > 如果在消费方配置了超时时间，那么将以消费方的超时时间为准，提供方的超时时间将被覆盖；如果消费方没有配置，则以提供方的超时时间为准。超时时间优先级：`reference method > reference global-attrs > service method > service global-attrs` 。

- oneway 调用

  如果是 oneway 调用方式，消费方不关心结果，在发起调用后直接返回，框架会忽略提供方的处理结果。在 Bolt 中，在 XML 中针对 `method` 的配置如下：

```
<!-- 配置 oneway -->
<sofa:reference interface="com.alipay.test.SampleService2" id="sampleService2">
    <sofa:binding.bolt>
        <sofa:global-attrs test-url="127.0.0.1:12200"/>
        <!-- 此处方法名 service 为示例，需要替换成实际方法名。 -->
        <sofa:method name="service" type="oneway"/>
    </sofa:binding.bolt>
</sofa:reference>
```

> 说明
>
> 由于消费方是直接返回，不关心处理结果，所以在 oneway 方式下配置超时属性是无效的。

- callback 调用

  callback 是一种异步回调方式，消费方需要提供回调接口。在调用结束后，回调接口会被框架调用。callback 接口支持通过配置 `class` 或引用 Bean 的方式实现 callback:

  - 配置 class

    XML 配置如下：

    ```
    <!-- callback 调用配置 -->
    <sofa:reference interface="com.alipay.test.SampleService2" id="sampleService">
        <sofa:binding.bolt>
            <!-- 此处方法名 testCallback 为示例，需要替换成实际方法名。 -->
            <sofa:method name="testCallback" type="callback" callback-class="com.alipay.tes
    t.binding.tr.MyCallBackHandler"/>
        </sofa:binding.bolt>
    </sofa:reference>
    ```

  - 引用 Bean

    XML 配置如下：

    ```
    <!-- 使用 Bean 来实现 callback 接口 -->
    <bean id="myCallBackHandlerBean"class="com.alipay.test.binding.tr.MyCallBackHandler"/>
    <sofa:reference interface="com.alipay.test.SampleService2" id="sampleService">
        <sofa:binding.bolt>
            <!-- 此处方法名 testCallback 为示例，需要替换成实际方法名。 -->
            <sofa:method name="testCallback" type="callback" callback-ref="myCallBackHandle
    rBean"/>
        </sofa:binding.bolt>
    </sofa:reference>
    ```

需要注意，使用 callback 调用方式时，callback 接口必须实现 `com.alipay.sofa.rpc.api.callback.SofaResponseCallback` 。这个接口包含三个方法，说明如下：

```
public interface SofaResponseCallback {

  /**
   * 当服务提供方业务层正常返回结果，sofa-remoting 层将回调该方法。
   * @param appResponse response object。
   * @param methodName 调用服务对应的方法名。
   * @param request callback 对应的 request。
   `*/
  public void onAppResponse(Object appResponse, String methodName, RequestBase request);

  /**
   * 当服务提供方业务层抛出异常，sofa-remoting 层将回调该方法。
   * @param t 服务方业务层抛出的异常。
   * @param methodName 调用服务对应的方法名。
   * @param request callback 对应的 request。
   `*/
  public void onAppException(Throwable t, String methodName, RequestBase request);

  /**
   * 当 sofa-remoting 层出现异常时，回调该方法。
   * @param sofaException sofa-remoting 层异常。
   * @param methodName 调用服务对应的方法名。
   * @param request callback 对应的 request。
   `*/
  public void onSofaException(SofaRpcException sofaException, String methodName,
                              RequestBase request);

}
```

- future 调用

  future 也是一种异步的调用方式。消费方发起调用后，马上返回。当需要结果时，消费方需要主动去获取数据。使用 future 的方式调用，配置和其他方式类似，只需要在 `method` 层面设置 `type` 为 future 即可。XML 配置如下：

  ```
  <!-- Future 调用配置 -->
  <sofa:reference interface="com.alipay.test.SampleService" id="sampleServiceFuture">
      <sofa:binding.bolt>
          <sofa:global-attrs test-url="127.0.0.1:12200"/>
          <!-- 此处方法名 service 为示例，需要替换成实际方法名。 -->
          <sofa:method name="service" type="future"/>
      </sofa:binding.bolt>
  </sofa:reference>
  ```

  使用 future 调用，返回的结果保存在一个 `ThreadLocal` 线程变量里面，可以通过如下方式获取这个线程变量的值：

```
// Future 获取调用结果。
public void testFuture() throws SofaException, InterruptedException {
  sampleServiceFuture.service();
  Object result = SofaResponseFuture.getResponse(1000, true);

  Assert.assertEquals("Hello, world!", result);
}
```

要拿到 future 调用后的结果，只需要调用 `SofaResponseFuture` 的 `getResponse` 方法即可。

`getResponse` 方法的两个参数说明如下：

- 第一个参数是超时时间，含义是调用线程等待的最长时间，负数表示无等待时间限制，零表示立即返回，单位是毫秒。
- 第二个参数代表是否清除 `ThreadLocal` 变量的值，如果设为 true，则返回 response 之前，先清除 `ThreadLocal` 的值，避免内存泄漏。

Bolt 引用详细配置

除了各种调用方式之外，Bolt 还在消费方提供了各种配置选项，这些选项不太常用，列举如下：

配置项	类型	默认值	说明
`connect.timeout`	INTEGER	1000	消费方连接超时时间。 单位：ms
`connect.num`	INTEGER	-1	消费方连接数。 -1 代表不设置（默认为每个目标地址建立一个连接）。
`idle.timeout`	INTEGER	-1	消费方最大空闲时间。 -1 表示使用底层默认值（底层默认值为 0，表示永远不会有读 idle）。该配置也是心跳的时间间隔，当请求 idle 时间超过配置时间后，发送心跳到服务方。
`idle.timeout.read`	INTEGER	-1	消费方最大读空闲时间。 -1 表示使用底层默认值（底层默认值为 30）。如果 `idle.timeout` > 0，在 `idle.timeout` + `idle.timeout.read` 时间内，如果没有请求发生，那么该连接将会自动断开。

配置项	类型	默认值	说明
address-wait-time	INTEGER	0	reference 生成时，等待服务注册中心将地址推送到消费方的时间。 单位：ms 取值范围：[0, 30000] 取值超过 30000 时，默认调整为 30000 。

8.2.4. 超时控制

使用 BOLT 协议进行通信的时，SOFARPC 的超时时间默认为 3000 毫秒，您可以在引用服务时设置超时时间，也可以在服务以及方法的维度设置超时时间。SOFARPC 的超时时间的设置的单位都为毫秒。

服务维度

如果您需要在发布服务时，在服务维度设置超时时间，可以设置 `timeout` 参数。

XML 方式

如果使用 XML 的方式引用服务，设置 `<sofa:binding.bolt>` 标签下的 `<sofa:global-attrs>` 标签的 `timeout` 属性的值即可。

```
<sofa:reference interface="com.example.demo.SampleService" id="sampleService">
    <sofa:binding.bolt>
        <sofa:global-attrs timeout="2000"/>
    </sofa:binding.bolt>
</sofa:reference>
```

Annotation 方式

如果使用 Annotation 引用服务，设置 `@SofaReferenceBinding` 的 `timeout` 属性的值即可。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt", timeout =2000))
private SampleService sampleService;
```

Spring 环境 API 方式

如果在 Spring 或者 Spring Boot 的环境下引用服务，设置 `BoltBindingParam` 的 `timeout` 属性的值即可。

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setTimeout(2000)
```

非 Spring 环境下 API 方式

如果在非 Spring 环境下直接使用 SOFARPC 的裸 API 引用服务，设置 `ConsumerConfig` 的 `timeout` 属性即可。

```
ConsumerConfig<SampleService> consumerConfig =new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRegistry(registryConfig)
.setProtocol("bolt")
.setTimeout(2000);
```

方法维度

如果您希望单独调整一个服务中某一个方法的超时时间，可以在方法维度上设置超时时间。对于某一个方法，优先生效方法维度的超时时间，如果没有设置，则使用服务维度的超时时间。

XML 方式

如果使用 XML 的方式引用一个服务，设置对应的 `<sofa:method>` 标签的 `timeout` 属性即可。

```
<sofa:reference interface="com.example.demo.SampleService" id="sampleService">
    <sofa:binding.bolt>
        <sofa:method name="hello" timeout="2000"/>
    </sofa:binding.bolt>
</sofa:reference>
```

Annotation 方式

目前暂未提供通过 Annotation 的方式来设置方法级别的超时时间。

Spring 环境 API 方式

如果在 Spring 或者 Spring Boot 的环境下引用服务，设置 `RpcBindingMethodInfo` 的 `timeout` 属性的值即可。

```
BoltBindingParam boltBindingParam =new BoltBindingParam();

RpcBindingMethodInfo rpcBindingMethodInfo =new RpcBindingMethodInfo();
rpcBindingMethodInfo.setName("hello");
rpcBindingMethodInfo.setTimeout(2000);

List<RpcBindingMethodInfo> rpcBindingMethodInfos =new ArrayList<>();
rpcBindingMethodInfos.add(rpcBindingMethodInfo);

boltBindingParam.setMethodInfos(rpcBindingMethodInfos);
```

非 Spring 环境 API 方式

如果在非 Spring 环境下使用 SOFARPC 的裸 API 引用服务，设置 `MethodConfig` 的 `timeout` 属性即可。

```
MethodConfig methodConfig =new MethodConfig();
methodConfig.setName("hello");
methodConfig.setTimeout(2000);

List<MethodConfig> methodConfigs =new ArrayList<MethodConfig>();
methodConfigs.add(methodConfig);

ConsumerConfig<SampleService> consumerConfig =new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRegistry(registryConfig)
.setProtocol("bolt")
.setMethods(methodConfigs);
```

8.2.5. 泛化调用

在进行 RPC 调用时，应用无需依赖服务提供方的 JAR 包，只需要知道服务的接口名、方法名即可调用 RPC 服务。

泛化接口

```
public interface GenericService{

/**
   * 泛化调用仅支持方法参数为基本数据类型，或者方法参数类型在当前应用的 ClassLoader 中存在的情况。
   *
   * @param methodName 调用方法名。
   * @param args 调用参数列表。
   * @return 调用结果。
   * @throws com.alipay.sofa.rpc.core.exception.GenericException 调用异常。
   */
Object $invoke(String methodName,String[] argTypes,Object[] args) throws GenericException;

/**
  * 支持参数类型无法在类加载器加载情况的泛化调用，对于非 JDK 类会序列化为 GenericObject。
  *
  * @param methodName  调用方法名。
  * @param argTypes  参数类型。
  * @param args 方法参数，参数类型支持 GenericObject。
  * @return result GenericObject 类型。
  * @throws com.alipay.sofa.rpc.core.exception.GenericException
  */
Object $genericInvoke(String methodName,String[] argTypes,Object[] args)
throws GenericException;

/**
     * 支持参数类型无法在类加载器加载情况的泛化调用。
     *
     * @param methodName 调用方法名。
     * @param argTypes   参数类型。
     * @param args       方法参数,参数类型支持 GenericObject。
     * @param context    GenericContext
     * @return result GenericObject 类型。
     * @throws com.alipay.sofa.rpc.core.exception.GenericException
```

```
     */
Object $genericInvoke(String methodName,String[] argTypes,Object[] args,
GenericContext context)throws GenericException;

/**
  * 支持参数类型无法在类加载器加载情况的泛化调用，返回结果类型为 T。
  *
  * @param methodName  调用方法名。
  * @param argTypes  参数类型。
  * @param args 方法参数，参数类型支持 GenericObject。
  * @return result T 类型。
  * @throws com.alipay.sofa.rpc.core.exception.GenericException
  */
<T> T $genericInvoke(String methodName,String[] argTypes,Object[] args,Class<T> clazz) thro
ws GenericException;

/**
     * 支持参数类型无法在类加载器加载情况的泛化调用。
     *
     * @param methodName 调用方法名。
     * @param argTypes   参数类型。
     * @param args       方法参数，参数类型支持 GenericObject。
     * @param clazz      返回类型。
     * @param context    GenericContext
     * @return result T 类型。
     * @throws com.alipay.sofa.rpc.core.exception.GenericException
     */
<T> T $genericInvoke(String methodName,String[] argTypes,Object[] args,Class<T> clazz,Gener
icContext context)throwsGenericException;

}
```

`$invoke` 仅支持方法参数类型在当前应用的 `ClassLoader` 中存在的情况；`$genericInvoke` 支持方法参数类型在当前应用的 `ClassLoader` 中不存在的情况。

使用示例

- 使用 Spring XML 创建泛化调用代理对象

```
<!-- 引用 TR 服务 -->
<sofa:reference interface="com.alipay.sofa.rpc.api.GenericService" id="genericService">
    <sofa:binding.tr>
        <sofa:global-attrs generic-interface="com.alipay.test.SampleService"/>
    </sofa:binding.tr>
</sofa:reference>
```

Java 代码：

```
/**
    * Java Bean
    */
   public class People{
       private String name;
       private int    age;

       // getters and setters
   }


   /**
    * 服务方提供的接口。
    */
   interface SampleService{
       String hello(String arg);
       People hello(People people);
   }
```

```
 /**
     * 消费方测试类。
     */
    public class ConsumerClass{
        GenericService genericService;

        public void do(){
            // 1. $invoke 仅支持方法参数类型在当前应用的 ClassLoader 中存在的情况。
            genericService.$invoke("hello",new String[]{String.class.getName()},new Objec
t[]{"I'm an arg"});

            // 2. $genericInvoke 支持方法参数类型在当前应用的 ClassLoader 中不存在的情况。
            // 2.1 构造参数
            GenericObject genericObject =new GenericObject("com.alipay.sofa.rpc.test.gene
ric.bean.People");// 构造函数中指定全路径类名。
            genericObject.putField("name","Lilei");// 调用 putField，指定field值。
            genericObject.putField("age",15);

            // 2.2 进行调用，不指定返回类型，返回结果类型为 GenericObject。
            Object obj = genericService.$genericInvoke("hello",new String[]{"com.alipay.s
ofa.rpc.test.generic.bean.People"},new Object[]{ genericObject });
            Assert.assertTrue(obj.getClass()==GenericObject.class);

            // 2.3 进行调用，指定返回类型。
            People people = genericService.$genericInvoke("hello",new String[]{"com.alipa
y.sofa.rpc.test.generic.bean.People"},new Object[]{ genericObject },People.class);

            // 3. LDC 架构下的泛化调用使用。
            // 3.1 构造 GenericContext 对象。
            AlipayGenericContext genericContext =new AlipayGenericContext();
            genericContext.setUid("33");

            // 3.2 进行调用。
            People people = genericService.$genericInvoke("hello",new String[]{"com.alipa
y.sofa.rpc.test.generic.bean.People"},new Object[]{ genericObject },People.class, generic
Context);
        }
```

- 使用 API 创建泛化调用代理对象

  在非必要情况下，并不推荐项目中使用 API 方式创建泛化调用代理对象，建议您优先通过 Spring XML 进行创建。如果您无法使用 Spring XML 方式进行初始化，可考虑使用 API 方式进行创建。

  使用 API 方式创建泛化调用代理时，需要注意以下事项：

  - 泛化调用代理是比较重的对象，不可放在交易中反复创建。

    RPC 框架会检查是否有重复创建的情况，当相同配置的服务引用超过 3 次时，会产生异常，影响正常使用。

  - 在应用启动过程中进行对象创建，通过异常中止、健康检查等方式确保项目启动时，代理对象已创建成功。
  - 在应用退出前销毁泛化调用代理，实现优雅停止。

  Java 代码：

```
@Component
public class GenericServiceDemo implements InitializingBean, DisposableBean {
    private ConsumerConfig<GenericService> consumerConfig;
    private GenericService genericService;

    public GenericService getGenericService() {
        return genericService;
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        this.consumerConfig = new ConsumerConfig<>();
        RegistryConfig registryConfig = new RegistryConfig()
                .setProtocol("dsr");
        consumerConfig.setGeneric(true)
                .setInterfaceId("com.alipay.test.SampleService")
                .setRegistry(registryConfig)
                .setProtocol("bolt")     // 引用 Bolt 服务。
                .setTimeout(5000);       // 设置超时时间。
        this.genericService = consumerConfig.refer();
    }

    @Override
    public void destroy() throws Exception {
        if(this.consumerConfig != null) {
            consumerConfig.unRefer();   // 注销引用。
        }
    }
}
```

特殊说明

调用 `$genericInvoke(String methodName, String[] argTypes, Object[] args)` 接口，会将除以下包以外的其他类序列化为 `GenericObject` 。

```
"com.sun",
"java",
"javax",
"org.ietf",
"org.ogm",
"org.w3c",
"org.xml",
"sunw.io",
"sunw.util"
```

8.2.6. 序列化协议

SOFARPC 在使用 Bolt 通信协议的情况下，可以选择不同的序列化协议。目前支持 hessian2 和 protobuf。

默认情况下，SOFARPC 使用 hessian2 作为序列化协议。如果您需要将序列化协议设置成 protobuf，需要在发布服务时进行如下设置：

在 `<sofa:binding.bolt>` 标签内增加 `<sofa:global-attrs>` 标签，并且设置 `serialize-type` 属性为 `protobuf` 。

```
<sofa:service ref="sampleService" interface="com.alipay.sofarpc.demo.SampleService">
    <sofa:binding.bolt>
        <sofa:global-attrs serialize-type="protobuf"/>
    </sofa:binding.bolt>
</sofa:service>
```

在引用服务时，您也需要将序列化协议改成 `protobuf` ，设置方式和发布服务的时候类似。

```
<sofa:reference interface="com.alipay.sofarpc.demo.SampleService" id="sampleServiceRef" jvm
-first="false">
    <sofa:binding.bolt>
        <sofa:global-attrs serialize-type="protobuf"/>
    </sofa:binding.bolt>
</sofa:reference>
```

目前，使用注解的方式尚不能支持设置序列化协议，这个将在后续的版本中支持。

8.2.7. 自定义线程池

SOFARPC 支持自定义业务线程池，可以为指定服务设置一个与 SOFARPC 业务线程池隔离的独立业务线程池。多个服务可以共用一个独立的线程池。

> **说明**
>
> SOFARPC 要求自定义线程池的类型必须是 `com.alipay.sofa.rpc.server.UserThreadPool` 。

XML 方式

如果采用 XML 的方式发布服务，您可以先设定一个 class 为 `com.alipay.sofa.rpc.server.UserThreadPool` 的线程池的 Bean，然后设置到 `<sofa:global-attrs>` 标签的 `thread-pool-ref` 属性中。

```
<bean id="helloService" class="com.alipay.sofa.rpc.quickstart.HelloService"/>

<!-- 自定义一个线程池 -->
<bean id="customExecutor" class="com.alipay.sofa.rpc.server.UserThreadPool" init-method="in
it">
    <property name="corePoolSize" value="10"/>
    <property name="maximumPoolSize" value="10"/>
    <property name="queueSize" value="0"/>
</bean>

<sofa:service ref="helloService" interface="XXXService">
    <sofa:binding.bolt>
       <!-- 将线程池设置给一个 Service -->
       <sofa:global-attrs thread-pool-ref="customExecutor"/>
    </sofa:binding.bolt>
</sofa:service>
```

Annotation 方式

如果是采用 Annotation 的方式发布服务，您可以通过设置 `@SofaServiceBinding` 的 `userThreadPool` 属性来设置自定义线程池的 Bean。

```
@SofaService(bindings ={@SofaServiceBinding(bindingType ="bolt", userThreadPool ="customThr
eadPool")})
public class SampleServiceImpl implements SampleService{
}
```

在 Spring 环境使用 API 方式

如果是在 Spring 环境下使用 API 的方式发布服务，您可以通过调用 `BoltBindingParam` 的 `setUserThreadPool` 方法来设置自定义线程池。

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setUserThreadPool(new UserThreadPool());
```

在非 Spring 环境下使用 API 方式

如果是在非 Spring 环境下使用 API 的方式，您可以通过如下的方式来设置自定义线程池。

```
UserThreadPool threadPool =new UserThreadPool();
threadPool.setCorePoolSize(10);
threadPool.setMaximumPoolSize(100);
threadPool.setKeepAliveTime(200);
threadPool.setPrestartAllCoreThreads(false);
threadPool.setAllowCoreThreadTimeOut(false);
threadPool.setQueueSize(200);

UserThreadPoolManager.registerUserThread(ConfigUniqueNameGenerator.getUniqueName(providerCo
nfig), threadPool);
```

8.3. RESTful 协议

8.3.1. RESTful 协议的基本用法

在 SOFARPC 中，使用不同的通信协议即使用不同的 Binding。如果需要使用 RESTful 协议，只要将 Binding 设置为 REST 即可。

发布服务

1. 定义服务接口。

   在定义 RESTful 服务接口时，需要采用 JAXRS 标准的注解在接口上加上元信息，示例如下：

   ```
   @Path("sample")
       public interface SampleService{
           @GET
           @Path("hello")
           String hello();
       }
   ```

   > **说明**
   >
   > JAXRS 的标准的注解的使用方式可以参考 RESTEasy 的文档。

2. 发布服务。

   定义好接口之后，你可以将接口的实现发布成一个服务。以 Annotation 方式发布服务为例：

   ```
   @Service
   @SofaService(bindings ={@SofaServiceBinding(bindingType ="rest")})
   public class RestfulSampleServiceImpl implements SampleService{
       @Override
       public String hello(){
           return "Hello";
       }
   }
   ```

如果您要通过其他方式发布服务，请参考 BOLT 协议基本使用。

通过浏览器访问服务

服务发布后，您可以通过浏览器直接访问服务。以上文发布的服务为例，SOFARPC 的 RESTful 服务的默认端口为 8341，访问地址如下：

```
http://localhost:8341/sample/hello
```

引用服务

您也可以通过 SOFARPC 标准的服务引用的方式来引用服务。以 Annotation 方式引用服务为例：

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="rest"))
private SampleService sampleService;
```

如果您要通过其他方式引用服务，请参考 BOLT 协议基本使用。

8.3.2. REST 跨域

对于 REST，SOFARPC 内置了一个跨域 Filter，您可以以此实现跨域访问。

SOFARPC API 方式

对于使用 SOFARPC API 的用户，可以在 ServerConfig 中添加如下参数，实现跨域访问。

```
Map<String,String> parameters=new HashMap<String,String>()
parameters.put(RpcConstants.ALLOWED_ORIGINS,"abc.com,cdf.com");
serverConfig.setParameters(parameters);
```

XML 方式

对于 XML 方式，直接通过如下配置即可。

```
com.alipay.sofa.rpc.rest.allowed.origins=a.com,b.com
```

8.3.3. REST 自定义 Filter

对于 REST，SOFAStack 支持 JAXRSProviderManager 管理器类，您可以根据这个管理器类自定义 Filter。

JAXRSProviderManager 管理器类在服务端生效，生效时间为服务启动时。接口如下：

```
com.alipay.sofa.rpc.server.rest.RestServer#registerProvider
```

对于您自定义的 Filter 类，可以在初始化完成后，调用如下类进行注册。

```
com.alipay.sofa.rpc.config.JAXRSProviderManager#registerCustomProviderInstance
```

其中，自定义的 Filter 遵循 REST 的规范，需要实现如下接口：

```
javax.ws.rs.container.ContainerResponseFilter
或者
javax.ws.rs.container.ContainerRequestFilter
```

REST server 启动之后，对于裸 SOFARPC 的使用，您需要先注册，再启动服务。对于 SOFABoot 环境下的使用，也是类似的过程，具体的写法可以参考如下示例：

```
com.alipay.sofa.rpc.server.rest.TraceRequestFilter
com.alipay.sofa.rpc.server.rest.TraceResponseFilter
```

8.3.4. 集成 SOFARPC RESTful 服务和 Swagger

本文介绍如何集成SOFARPC RESTful 服务和 Swagger。

使用 rpc-sofa-boot-starter 6.0.1 以上版本

从 `rpc-sofa-boot-starter` 6.0.1 版本开始，SOFARPC 提供了 RESTful 服务和 Swagger 的一键集成能力。操作步骤如下：

1. 在 `pom.xml` 中增加 Swagger 的依赖。

```
<dependency>
    <groupId>io.swagger.core.v3</groupId>
    <artifactId>swagger-jaxrs2</artifactId>
    <version>2.0.0</version>
</dependency>
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>20.0</version>
</dependency>
```

2. 在 `application.properties` 里面增加 `com.alipay.sofa.rpc.restSwagger=true` 。

3. 访问 `http://localhost:8341/swagger/openapi` 即可获取 SOFARPC 的 RESTful 的 Swagger OpenAPI 内容。

不使用 rpc-sofa-boot-starter 或版本低于 6.0.1

如果没有使用 `rpc-sofa-boot-starter` 或者 `rpc-sofa-boot-starter` 的版本低于 6.0.1，可以采用如下的方式集成 Swagger:

1. 在应用中引入 Swagger 相关的依赖。

   由于 SOFARPC 的 RESTful 协议使用的是 JAXRS 标准，因此只需引入 Swagger 的 JAXRS 依赖。依赖如下：

```
<dependency>
      <groupId>io.swagger.core.v3</groupId>
      <artifactId>swagger-jaxrs2</artifactId>
      <version>2.0.0</version>
</dependency>
<dependency>
     <groupId>com.google.guava</groupId>
     <artifactId>guava</artifactId>
     <version>20.0</version>
</dependency>
```

   > **说明**
   >
   > 引入 Guava 的 20.0 的版本是为了解决 Guava 的版本冲突。

2. 发布一个 Swagger 的 RESTful 服务。

   为了让 Swagger 能够将 SOFARPC 的 RESTful 的服务通过 Swagger OpenAPI 暴露出去，可以反向使用 SOFARPC 的 RESTful 的服务提供 Swagger 的 OpenAPI 服务。

i. 新建一个接口。

```
@Path("swagger")
public interface OpenApiService{
    @GET
    @Path("openapi")
    @Produces("application/json")
    String openApi();
}
```

ii. 提供一个实现类，并发布成 SOFARPC 的 RESTful 的服务。

```
@Service
@SofaService(bindings ={@SofaServiceBinding(bindingType ="rest")}, interfaceType =O
penApiService.class)
public class OpenApiServiceImpl implements OpenApiService,InitializingBean{
     private OpenAPI openAPI;

    @Override
    public String openApi(){
         return Json.pretty(openAPI);
    }

    @Override
    public void afterPropertiesSet(){
    List<Package> resources =new ArrayList<>();
    resources.add(this.getClass().getPackage());// 扫描当前类所在的 Package，也可以扫描
其他的 SOFARPC RESTful 服务接口所在的 Package
    if(!resources.isEmpty()){
         // init context
        try{
            SwaggerConfiguration oasConfig =new SwaggerConfiguration()
            .resourcePackages(resources.stream().map(Package::getName).collect(Coll
ectors.toSet()));

            OpenApiContext oac =new JaxrsOpenApiContextBuilder()
            .openApiConfiguration(oasConfig)
            .buildContext(true);
            openAPI = oac.read();
         }catch(OpenApiConfigurationException e){
            throw new RuntimeException(e.getMessage(), e);
       }
    }
  }
}
```

3. 应用启动后，访问 `http://localhost:8341/swagger/openapi` 即可得到当前的应用发布的所有的 RESTful 的服务的信息。

解决跨域问题

如果您在另外一个端口中启动了一个 Swagger UI，并且希望通过 Swagger UI 访问 `http://localhost:8341/swagger/openapi` 查看 API 定义，发起调用，可能需要解决访问跨域的问题。您可以在应用启动前增加如下代码：

```
import org.jboss.resteasy.plugins.interceptors.CorsFilter;

public static void main(String[] args){
    CorsFilter corsFilter =newCorsFilter();
    corsFilter.getAllowedOrigins().add("*");
    JAXRSProviderManager.registerCustomProviderInstance(corsFilter);
    SpringApplication.run(DemoApplication.class, args);
}
```

8.4. Dubbo 协议

8.4.1. Dubbo 协议的基本使用

在 SOFARPC 中，使用不同的通信协议只要设置使用不同的 Binding 即可。如果需要使用 Dubbo 协议，只要将 Binding 设置为 Dubbo 即可。

本文以注解的使用方式为例，其他使用方式可以参考 BOLT 协议基本使用。

发布服务

发布一个 Dubbo 的服务，只需要将 `@SofaServiceBinding` 的 `bindingType` 设置为 `dubbo` 。

```
@Service
@SofaService(bindings ={@SofaServiceBinding(bindingType ="dubbo")})
public class SampleServiceImpl implements SampleService{
}
```

引用服务

引用一个 Dubbo 的服务，只需要将 `@SofaReferenceBinding` 的 `bindingType` 设置为 `dubbo` 。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="dubbo"), jvmFirst =false)
private SampleService sampleService;
```

设置 Dubbo 服务的 Group

在 SOFARPC 的模型中，没有直接的字段叫做 Group，但 SOFARPC 有一个 `uniqueId` 模型，可以直接映射到 Dubbo 的模型中的 Group。比如下面的代码，就是发布了一个 Group 为 `groupDemo` 的服务。

```
@Service
@SofaService(bindings ={@SofaServiceBinding(bindingType ="dubbo")}, uniqueId ="groupDemo")
public class SampleServiceImpl implements SampleService{
}
```

如下代码引用了一个 Group 为 `groupDemo` 的服务。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="dubbo"), uniqueId ="groupDemo",
jvmFirst =false)
private SampleService sampleService;
```

> **注意**
>
> 目前 Dubbo 协议只支持 Zookeeper 作为服务注册中心。

8.5. H2C 协议

8.5.1. H2C 协议的基本使用

在 SOFARPC 中，使用不同的通信协议只要设置使用不同的 Binding 即可。如果需要使用 H2C 协议，只要将 Binding 设置为 H2C。

本文以注解的使用方式为例，其他使用方式可以参考 BOLT 协议基本使用。

发布服务

发布一个 H2C 的服务，只需要将 `@SofaServiceBinding` 的 `bindingType` 设置为 `h2c` 。

```
@Service
@SofaService(bindings ={@SofaServiceBinding(bindingType ="h2c")})
public class SampleServiceImpl implements SampleService{
}
```

引用服务

引用一个 H2C 的服务，只需要将 `@SofaReferenceBinding` 的 `bindingType` 设置为 `h2c` 。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="h2c"), jvmFirst =false)
private SampleService sampleService;
```

8.6. HTTP 协议

8.6.1. HTTP 协议的基本使用

在 SOFARPC（非 SOFABoot 环境）中使用 HTTP 作为服务端协议时，支持 JSON 以序列化方式作为一些基础的测试方式使用。

发布服务

```
// 只有1个线程执行。
ServerConfig serverConfig = new ServerConfig()
.setStopTimeout(60000)
.setPort(12300)
.setProtocol(RpcConstants.PROTOCOL_TYPE_HTTP)
.setDaemon(true);

// 发布一个服务，每个请求要执行 1 秒。
ProviderConfig<HttpService> providerConfig = new ProviderConfig<HttpService>()
.setInterfaceId(HttpService.class.getName())
.setRef(new HttpServiceImpl())
.setApplication(new ApplicationConfig().setAppName("serverApp"))
.setServer(serverConfig)
.setUniqueId("uuu")
.setRegister(false);
providerConfig.export();
```

引用服务

因为是 HTTP + JSON，所以引用方可以直接通过 HttpClient 进行调用，以下为一段测试代码。

```
private ObjectMapper mapper =new ObjectMapper();
HttpClient httpclient =HttpClientBuilder.create().build();
// POST 正常请求。
String url ="http://127.0.0.1:12300/com.alipay.sofa.rpc.server.http.HttpService:uuu/object"
;
HttpPost httpPost =new HttpPost(url);
httpPost.setHeader(RemotingConstants.HEAD_SERIALIZE_TYPE,"json");
ExampleObj obj =new ExampleObj();
obj.setId(1);
obj.setName("xxx");
byte[] bytes = mapper.writeValueAsBytes(obj);
ByteArrayEntity entity =new ByteArrayEntity(bytes,
ContentType.create("application/json"));

httpPost.setEntity(entity);

HttpResponse httpResponse = httpclient.execute(httpPost);
Assert.assertEquals(200, httpResponse.getStatusLine().getStatusCode());
byte[] data =EntityUtils.toByteArray(httpResponse.getEntity());

ExampleObj result = mapper.readValue(data,ExampleObj.class);

Assert.assertEquals("xxxxx", result.getName());
```

9.链路说明

9.1. 链路追踪

SOFARPC 集成了 SOFATracer 的功能，可以输出链路中的数据信息。默认开启。

默认为 `JSON` 数据格式，具体的字段含义解释如下：

RPC 客户端摘要日志（ rpc-client-digest.log）

日志示例如下：

```
{"timestamp":"2018-05-20 17:03:20.708","tracerId":"1e27326d152680700049810018559 7","spanId"
:"0","span.kind":"client","local.app":"SOFATracerRPC","protocol":"bolt","service":"com.alip
ay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","current.thr
ead.name":"main","invoke.type":"sync","router.record":"DIRECT","remote.ip":"127.0.0.1:12200
","local.client.ip":"127.0.0.1","result.code":"00","req.serialize.time":"33","resp.deserial
ize.time":"39","resp.size":"170","req.size":"582","client.conn.time":"0","client.elapse.tim
e":"155","local.client.port":"59774","baggage":""}
```

参数	说明
timestamp	日志打印时间
tracerId	请求的 TracerId。
spanId	请求的 SpanId。
span.kind	Span 类型。
local.app	当前 App 名称。
protocol	协议类型。
service	服务接口信息。
method	方法名。
current.thread.name	当前线程名。
invoke.type	调用类型。

参数	说明
router.record	路由记录
remote.ip	目标 IP。
local.client.ip	本机 IP。
result.code	返回码。
req.serialize.time	请求序列化时间。
resp.deserialize.time	响应反序列化时间。
resp.size	响应大小，单位Byte。
req.size	请求大小，单位 Byte。
client.conn.time	客户端连接耗时。
client.elapse.time	客户端调用总耗时。
local.client.port	本地客户端端口。
baggage	透传的 baggage 数据 （KV 格式）。

RPC 服务端摘要日志（ rpc-server-digest.log）

日志示例如下：

{"timestamp":"2018-05-20 17:00:53.312","tracerId":"1e27326d1526806853032100185011","spanId":"0","span.kind":"server","service":"com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"SOFATracerRPC","protocol":"bolt","local.app":"SOFATracerRPC","current.thread.name":"SOFA-BOLT-BIZ-12200-5-T1","result.code":"00","server.pool.wait.time":"3","biz.impl.time":"0","resp.serialize.time":"4","req.deserialize.time":"38","resp.size":"170","req.size":"582","baggage":"",{"timestamp":"2018-05-20 17:03:05.646","tracerId":"1e27326d1526806985394100185589","spanId":"0","span.kind":"server","service":"com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"SOFATracerRPC","protocol":"bolt","local.app":"SOFATracerRPC","current.thread.name":"SOFA-BOLT-BIZ-12200-5-T1","result.code":"00","server.pool.wait.time":"2","biz.impl.time":"1","resp.serialize.time":"1","req.deserialize.time":"6","resp.size":"170","req.size":"582","baggage":"",{"timestamp":"2018-05-20 17:03:20.701","tracerId":"1e27326d1526807000498100185597","spanId":"0","span.kind":"server","service":"com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"SOFATracerRPC","protocol":"bolt","local.app":"SOFATracerRPC","current.thread.name":"SOFA-BOLT-BIZ-12200-5-T1","result.code":"00","server.pool.wait.time":"2","biz.impl.time":"0","resp.serialize.time":"1","req.deserialize.time":"4","resp.size":"170","req.size":"582","baggage":"",{"timestamp":"2018-05-20 17:04:19.966","tracerId":"1e27326d1526807046606100185635","spanId":"0","span.kind":"server","service":"com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"SOFATracerRPC","protocol":"bolt","local.app":"SOFATracerRPC","current.thread.name":"SOFA-BOLT-BIZ-12200-5-T1","result.code":"00","server.pool.wait.time":"2","biz.impl.time":"0","resp.serialize.time":"1","req.deserialize.time":"4","resp.size":"170","req.size":"582","baggage":""}

参数	说明
timestamp	日志打印时间。
tracerId	请求的 TracerId。
spanId	请求的 SpanId。
span.kind	Span 类型。
server	服务接口信息。
method	方法名。
remote.ip	来源 IP。
remote.app	来源 App 名称。

参数	说明
protocol	协议类型。
local.app	本地 App 名称。
current.thread.name	当前线程名。
result.code	返回码。
server.pool.wait.time	服务端线程池等待时间。
biz.impl.time	业务处理耗时。
resp.serialize.time	响应序列化时间。
req.deserialize.time	请求反序列化时间。
resp.size	响应大小，单位 Byte。
req.size	请求大小，单位 Byte。
baggage	透传的 baggage 数据（KV 格式）。

RPC 客户端统计日志（ rpc-client-stat.log）

日志示例如下：

```
{"time":"2018-05-18 07:02:19.717","stat.key":{"method":"method","local.app":"client","servi
ce":"app.service:1.0"},"count":10,"total.cost.milliseconds":17,"success":"Y"}
```

参数	说明
time	日志打印时间。
stat.key	日志关键 key。

参数	说明
method	方法信息。
local.app	客户端 App 名称。
service	服务接口信息。
count	调用次数。
total.cost.milliseconds	总耗时。
success	调用结果。

RPC 服务端统计日志（ rpc-server-stat.log）

日志示例如下：

```
{"time":"2018-05-18 07:02:19.717","stat.key":{"method":"method","local.app":"client","service":"app.service:1.0"},"count":10,"total.cost.milliseconds":17,"success":"Y"}
```

参数	说明
time	日志打印时间。
stat.key	日志关键 key。
method	方法信息。
local.app	客户端 App 名称。
service	服务接口信息。
count	调用次数。
total.cost.milliseconds	总耗时。

参数	说明
success	调用结果。

9.2. 链路数据透传

链路数据透传功能支持应用向调用上下文中存放数据，使得整个链路上的应用都可以操作该数据。

您可以分别向链路的 request 和 response 中放入数据进行透传，并可获取到链路中相应的数据。使用方式如下：

```
RpcInvokeContext.getContext().putRequestBaggage("key_request","value_request");
RpcInvokeContext.getContext().putResponseBaggage("key_response","value_response");

String requestValue=RpcInvokeContext.getContext().getRequestBaggage("key_request");
String responseValue=RpcInvokeContext.getContext().getResponseBaggage("key_response");
```

使用示例

例如数据从 A 到 B，再到 C 的场景中，将 A 设置的请求隐式传参数据传递给 B 和 C。在返回的时候，将 C 和 B 的响应隐式传参数据传递给 A。

A 请求方设置：

```
// 调用前设置请求透传的值。
RpcInvokeContext context =RpcInvokeContext.getContext();
context.putRequestBaggage("reqBaggageB","a2bbb");
// 调用。
String result = service.hello();
// 拿到结果透传的值。
context.getResponseBaggage("respBaggageB");
```

B 业务代码：

```
public String hello(){
    // 拿到请求透传的值。
    RpcInvokeContext context =RpcInvokeContext.getContext();
    String reqBaggage = context.getRequestBaggage("reqBaggageB");
    doSomthing();
    // 结果透传一个值。
    context.putResponseBaggage("respBaggageB","b2aaa");
    return result;
}
```

如果中途自己启动了子线程，则需要设置子线程的上下文：

```
CountDownLatch latch =new CountDownLatch(1);
finalRpcInvokeContext parentContext =RpcInvokeContext.peekContext();
Thread thread =new Thread(new Runnable(){
    public void run(){
          try{
                RpcInvokeContext.setContext(parentContext);
                // 调一个远程服务。
                xxxService.sayHello();
                latch.countDown();
          }finally{
               RpcInvokeContext.removeContext();
          }
    }
},"new-thread");
thread.start();

// 此时拿不到返回值透传的数据的。
latch.await();//等待
// 此时返回结束，能拿到返回透传的值。
```

和 SOFATracer 的比较

SOFATracer 是蚂蚁开源的一个分布式链路追踪系统，RPC 目前已经集成 Tracer，默认开启。

RPC 和 Tracer 进行数据传递不同点如下：

- RPC 的数据透传更偏向业务使用，而且可以在全链路中进行双向传递。调用方可以传给服务方，服务方也可以传递信息给调用方。SOFATracer 更加偏向于中间件和业务无感知的数据的传递，只能进行单向传递。
- RPC 的透传可以选择性地不在全链路中透传，而 Tracer 中如果传递大量信息，会在整个链路中传递，可能对下游业务会有影响。

所以整体来看，两者各有利弊，在有一些和业务相关的透传数据的情况下，可以选择 RPC 的透传。

9.3. 调用上下文

RPC 上下文中存放了当前调用过程中的一些其他信息，如服务提供方应用名、IP。应用开发人员可以获取这些信息做一些业务上的操作。

RPC 提供获取单次调用上下文的工具类 `com.alipay.sofa.rpc.api.context.RpcContextManager`，您可以通过该类获得最后一次 Reference 以及当次 Service 的相关信息。

> **注意**
>
> RPC 上下文是存放在 `ThreadLocal` 中的临时数据，切换线程或者清空 `ThreadLocal` 后数据都将丢失。

使用方式

```
// Reference Context
SampleService sampleService;
public void do(){
   sampleService.hello();
   // 参数为 true 代表清空上下文信息。
   RpcReferenceContext referenceContext =RpcContextManager.lastReferenceContext(true);
  // do something on referenceContext
}
```

```
// Service Context
public void doService(){
   // do sth
   ...
  // 参数为 true 代表清空上下文信息。
  RpcServiceContext serviceContext =RpcContextManager.currentServiceContext(true);
  // do something on serviceContext
  ...
}
```

上下文内容

RPC 上下文的信息均是从 Tracer 中获得，包含如下内容：

Reference

参数	说明
traceId	Trace ID 名称。
rpcId	RPC ID 名称。
interfaceName	服务接口。
methodName	服务方法。
uniqueId	服务的唯一标识。
serviceName	唯一的服务名。
isGeneric	是否为泛化调用。
targetAppName	服务提供方的应用名。

参数	说明
targetUrl	服务提供方的地址。
protocol	调用协议，例如 TR。
invokeType	调用类型，例如 sync、oneway 等。
routeRecord	路由寻址链路，例如 `TURL>CFS>RDM`，表示路由寻址路径是从 `test-url` 到软负载到随机寻址。如果上次请求的路由策略是 `test-url`，则 `routeRecord` 等于 `TURL>RDM`。 如要判断 `test-url` 或者软负载是否生效，请使用 `RpcReferenceContext.isTestUrlValid` 或者 `RpcReferenceContext.isConfigServerValid` 方法。详细的路由规则，请参见 RPC 路由。
costTime	调用耗时，单位为 ms。
resultCode	**结果码，取值如下：** • **00：表示成功。** • **01：表示业务异常。** • **02：表示 RPC 框架错误。** • **03：表示出现超时失败错误。** • **04：表示路由失败。**

Service

参数	说明
traceId	Trace ID 名称。
rpcId	RPC ID 名称。

参数	说明
methodName	服务方法。
serviceName	唯一的服务名。
callerAppName	服务消费方的应用名。
callerUrl	服务消费方的地址。

更多信息，请参见 SOFARPC 日志。

10.高级功能

10.1. Node 跨语言调用

本文介绍如何通过 NodeJs 调用 SOFARPC服务。

安装 SOFARPC Nodejs

执行以下命令安装 SOFARPC Nodejs：

```
$ npm install sofa-rpc-node --save
```

更多信息请参见 GitHub。

代码示例

暴露 RPC 服务，并发布到注册中心

```
'use strict';

const{RpcServer}= require('sofa-rpc-node').server;
const{ZookeeperRegistry}= require('sofa-rpc-node').registry;
const logger = console;

// 创建 zk 注册中心客户端。
const registry =newZookeeperRegistry({
  logger,
  address:'127.0.0.1:2181',// 需要本地启动一个 zkServer
});

// 创建 RPC Server 实例。
const server =newRpcServer({
  logger,
  // 传入注册中心客户端。
  registry,
  port:12200,
});

// 添加服务。
server.addService({
  interfaceName:'com.nodejs.test.TestService',
},{
  async plus(a, b){
return a + b;
},
});

// 启动 Server 并发布服务。
server.start()
.then(()=>{
    server.publish();
});
```

调用 RPC 服务（从注册中心获取服务列表）

```
'use strict';

const{RpcClient}= require('sofa-rpc-node').client;
const{ZookeeperRegistry}= require('sofa-rpc-node').registry;
const logger = console;

// 创建 zk 注册中心客户端。
const registry =new ZookeeperRegistry({
  logger,
  address:'127.0.0.1:2181',
});

async function invoke(){
// 创建 RPC Client 实例。
const client =new RpcClient({
    logger,
    registry,
});
// 创建服务的 consumer。
const consumer = client.createConsumer({
    interfaceName:'com.nodejs.test.TestService',
});
// 等待 consumer ready（从注册中心订阅服务列表...）。
  await consumer.ready();

// 执行泛化调用。
const result = await consumer.invoke('plus',[1,2],{ responseTimeout:3000});
  console.log('1 + 2 = '+ result);
}

invoke().catch(console.error);
```

调用 RPC 服务（直连模式）

```
'use strict';

const{RpcClient}= require('sofa-rpc-node').client;
const logger = console;

async function invoke(){
// 不需要传入 registry 实例了。
const client =newRpcClient({
    logger,
});
const consumer = client.createConsumer({
    interfaceName:'com.nodejs.test.TestService',
    // 直接指定服务地址。
    serverHost:'127.0.0.1:12200',
});
  await consumer.ready();

const result = await consumer.invoke('plus',[1,2],{ responseTimeout:3000});
  console.log('1 + 2 = '+ result);
}

invoke().catch(console.error);
```

暴露和调用 protobuf 接口

接口定义

通过 `*.proto` 定义接口。

```
syntax ="proto3";

package com.alipay.sofa.rpc.test;

// 可选
option java_multiple_files =false;

service ProtoService{
  rpc echoObj (EchoRequest) returns (EchoResponse){}
}

message EchoRequest{
   string name =1;
   Groupgroup=2;
}

message EchoResponse{
  int32 code =1;
  string message =2;
}

enumGroup{
  A =0;
  B =1;
}
```

服务端代码

```
'use strict';

const antpb = require('antpb');
const protocol = require('sofa-bolt-node');
const{RpcServer}= require('sofa-rpc-node').server;
const{ZookeeperRegistry}= require('sofa-rpc-node').registry;
const logger = console;

// 传入 *.proto 文件存放的目录，加载接口定义。
const proto = antpb.loadAll('/path/proto');
// 将 proto 设置到协议中。
protocol.setOptions({ proto });

const registry =new ZookeeperRegistry({
  logger,
  address:'127.0.0.1:2181',
});

const server =new RpcServer({
  logger,
  // 覆盖协议。
  protocol,
  registry,
  // 设置默认的序列化方式为 protobuf。
  codecType:'protobuf',
  port:12200,
});

server.addService({
  interfaceName:'com.alipay.sofa.rpc.test.ProtoService',
},{
  async echoObj(req){
    req = req.toObject({ enums:String});
return{
      code:200,
      message:'hello '+ req.name +', you are in '+ req.group,
};
},
});
server.start()
.then(()=>{
    server.publish();
});
```

客户端代码

```
'use strict';

const antpb = require('antpb');
const protocol = require('sofa-bolt-node');
const{RpcClient}= require('sofa-rpc-node').client;
const{ZookeeperRegistry}= require('sofa-rpc-node').registry;
const logger = console;

// 传入 *.proto 文件存放的目录，加载接口定义。
const proto = antpb.loadAll('/path/proto');
// 将 proto 设置到协议中。
protocol.setOptions({ proto });

const registry =new ZookeeperRegistry({
  logger,
  address:'127.0.0.1:2181',
});

async function invoke(){
const client =new RpcClient({
    logger,
    protocol,
    registry,
});
const consumer = client.createConsumer({
    interfaceName:'com.alipay.sofa.rpc.test.ProtoService',
});
  await consumer.ready();

const result = await consumer.invoke('echoObj',[{
    name:'gxcsoccer',
    group:'B',
}],{ responseTimeout:3000});
  console.log(result);
}

invoke().catch(console.error);
```

10.2. 多注册中心注册

同一个服务可以注册多个注册中心，您可以通过以下方式实现：

XML 方式

1. 在 `application.properties` 中配置注册中心源信息。

   配置格式为：`com.alipay.sofa.rpc.registries.<自定义别名>=协议://地址`，支持的协议为：dsr、sofa、zookeeper、consul、gateway、mesh、multicast、local、nacos。

   配置示例如下：

```
com.alipay.sofa.rpc.registries.registryName1=dsr://10.0.0.1:9600
com.alipay.sofa.rpc.registries.registryName2=dsr://10.0.0.2:9600
```

2. 配置需要多注册中心的服务。

```
<sofa:service interface="com.alipay.sofa.facade.SampleService" ref="sampleService">
        <sofa:binding.bolt>
            <sofa:global-attrs registry="registryName1,registryName2"/>
        </sofa:binding.bolt>

    </sofa:service>
```

API 方式

1. 构建多个 `RegistryConfig`，并设置给 `ProviderConfig`。

   示例如下：

```
List<RegistryConfig> registryConfigs =new ArrayList<RegistryConfig>();
registryConfigs.add(registryA);
registryConfigs.add(registryB);
providerConfig.setRegistry(registryConfigs);
```

2. 调用 `MethodConfig` 对象相应的 `set` 方法设置对应的参数。

   示例如下：

```
MethodConfig methodConfigA =new MethodConfig();
MethodConfig methodConfigB =new MethodConfig();
List<MethodConfig> methodConfigs =new ArrayList<MethodConfig>();
methodConfigs.add(methodConfigA);
methodConfigs.add(methodConfigB);
providerConfig.setMethods(methodConfigs);     //服务端设置
consumerConfig.setMethods(methodConfigs);  //客户端设置
```

10.3. 使用 Nacos 作为注册中心

SOFARPC 已支持使用 Nacos 作为服务注册中心，本文介绍如何配置。

前提条件

已在本地部署好 Nacos Server，服务发现的端口默认设置为 `8848`。部署 Nacos Server 操作，请参见 Nacos 快速开始。

配置方法

在 SOFARPC 中使用 Nacos 作为服务注册中心时，只需要在 `application.properties` 中加入如下配置即可：

```
com.alipay.sofa.rpc.registry.address=nacos://127.0.0.1:8848
```

如果您直接使用了 SOFARPC，而不是 SOFABoot，需要手动添加 Nacos 的依赖：

```
<dependency>
    <groupId>com.alibaba.nacos</groupId>
    <artifactId>nacos-client</artifactId>
    <version>${version}</version>
</dependency>
```

其中 `${version}` 为您想使用的 Nacos 版本。当前支持 Nacos 的版本如下：

- SOFARPC 5.5.0 支持 Nacos 服务端版本为 0.6.0、SOFABoot 版本为 2.5.3。
- SOFARPC 5.6.0 支持 Nacos 服务端版本为 1.0.0。

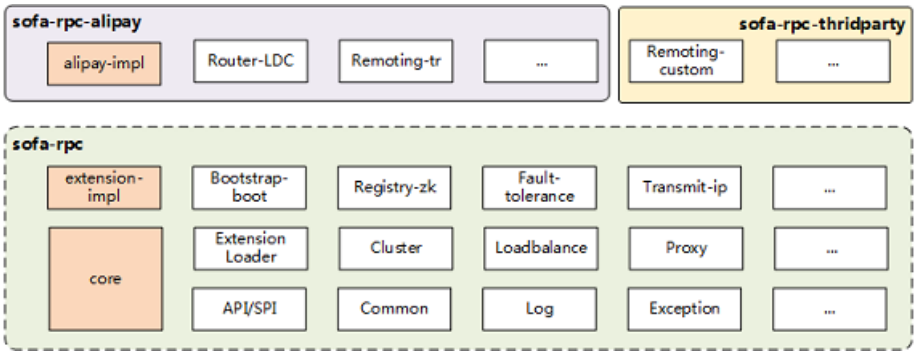
11.自定义扩展

11.1. 架构模块介绍

本文介绍 SOFARPC 的架构模块。

工程架构

SOFARPC 架构如下所示：



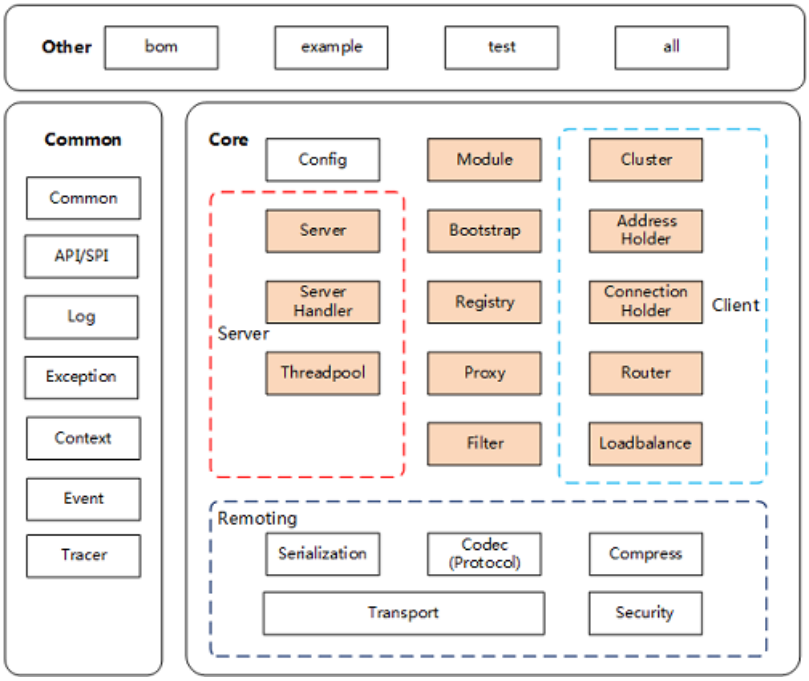
SOFARPC 从下到上分为两层：

- 核心层：包含了 RPC 的核心组件（例如各种接口、API、公共包）以及一些通用的实现（例如随机等负载均衡算法）。
- 功能实现层：所有的功能实现层的用户都是平等的，都是基于扩展机制实现的。

模块划分

SOFARPC 各个模块的实现类都只在自己模块中出现，一般不交叉依赖。需要交叉依赖的实现类已经全部抽象到 core 或者 common 模块中。

模块划分如下：

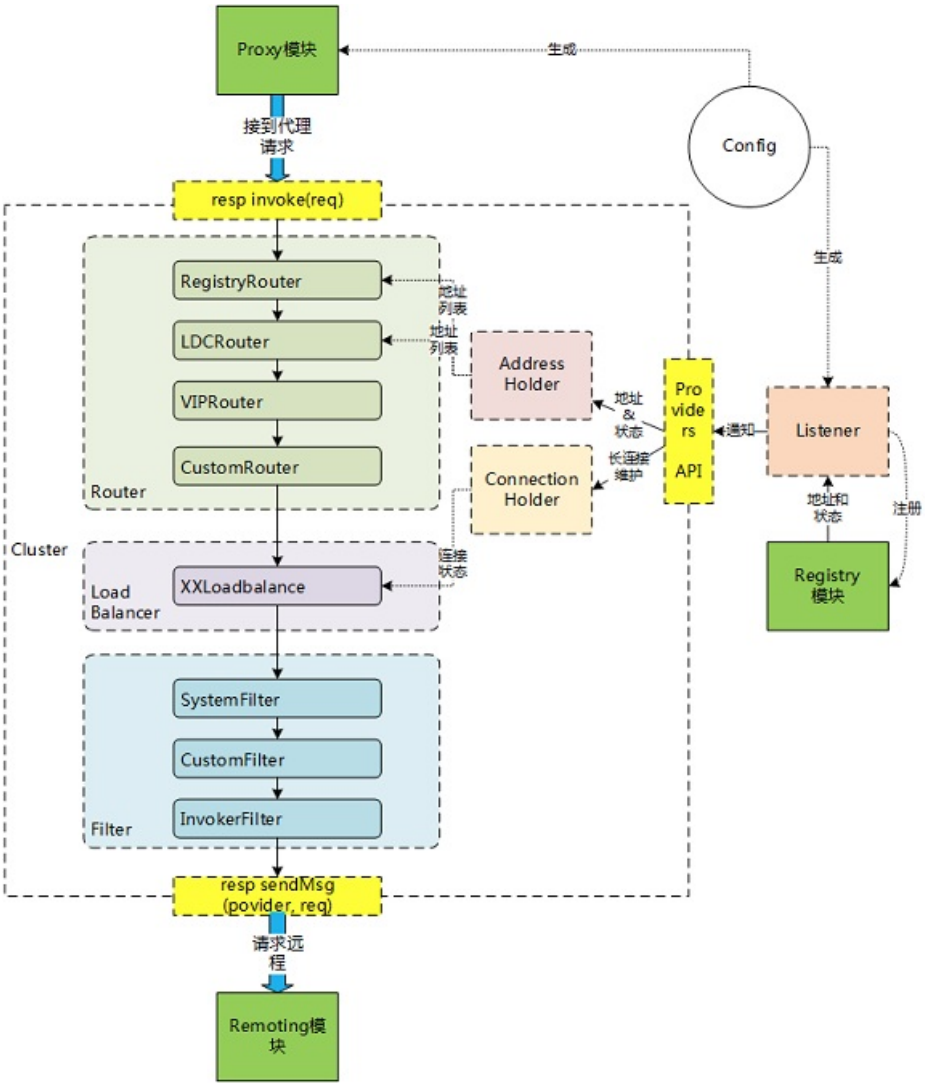


模块名	子模块名	中文名	说明	依赖
all	-	发布打包模块	需要打包的全部模块。	all
bom	-	依赖管控模块	依赖版本管控。	无
example	-	示例模块	-	all
test	-	测试模块	包含集成测试	all
core	api	API模块	各种基本流程接口、消息、上下文、扩展接口等。	common
core	common	公共模块	utils、数据结构。	exception
core	exception	异常模块	各种异常接口等。	common
bootstrap	-	启动实现模块	启动类，发布或者引用服务逻辑以及 registry 的操作。	core
proxy	-	代理实现模块	接口实现代理生成。	core
client	-	客户端实现模块	发送请求、接收响应、连接维护、路由、负载均衡、同步异步等。	core
server	-	服务端实现模块	启动监听、接收请求，发送响应、业务线程分发等。	core
filter	-	拦截器实现模块	服务端和客户端的各种拦截器实现。	core
codec	-	编解码实现模块	例如压缩，序列化等。	core

模块名	子模块名	中文名	说明	依赖
protocol	-	协议实现模块	协议的包装处理、协商。	core
transport	-	网络传输实现模块	TCP 连接的建立、数据分包粘包处理、请求响应对象分发等。	core
registry	-	注册中心实现模块	实现注册中心，例如 zk。	core

11.2. 客户端调用流程

客户端模块是一个较复杂的模块，这里包含了集群管理、路由、地址管理器、连接管理器、负载均衡器，还与代理、注册中心等模块交互。



11.3. 关键类介绍

本文介绍在使用 SOFARPC 进行开发时会涉及的关键类。

消息

内部消息全部使用 `SofaRequest` 和 `SofaResponse` 进行传递。如果您需要转换为其它协议，请在真正调用和收到请求的时候，转换为实际要传输的对象。

可以对 `SofaRequest` 和 `SofaResponse` 进行写入操作的模块如下：

- Invoker
- Filter
- ServerHandler
- Serialization

对消息体是只读的模块如下：

- Cluster

- Router
- LoadBalance

日志

日志的初始化也是基于扩展机制，但由于日志先加载，所以在 `rpc-config.json` 里有一个单独的 Key：

```
{
//日志实现。日志早于配置加载，所以不能适应Extension机制。
"logger.impl":"com.alipay.sofa.rpc.log.MiddlewareLoggerImpl"
}
```

配置项

使用者的 RPC 配置

用户的配置。例如端口配置（虽然已经开放对象中设置端口的字段，但是 SOFA 默认是从配置文件里读取）、线程池大小配置等。

- 通过 `SofaConfigs` 加载配置，调用 `ExternalConfigLoader` 读取外部属性。
- 通过 `SofaConfigs` 提供的 API 进行获取。
- 所有内部配置的 Key 都在 `SofaOptions` 类。
- 优先级：`System.property` > `sofa-config.properties`（每个应用一个）> `rpc-config.properties`。

RPC 框架配置

框架自身的配置。例如默认序列化、默认超时等。

- 通过 `RpcConfigs` 加载配置文件。
- 通过 `RpcConfigs` 其提供的 API 进行获取和监听数据变化。
- 所有内部配置的 Key 都在 `RpcOptions` 类。
- 优先级：`System.property` > `custom rpc-config.json`（可能存在多个自定义，会排序）> `rpc-config-default.json`。

常量

- 全局的基本常量在 `RpcConstants` 中，例如：
  - 调用方式：sync、oneway。
  - 协议：bolt、grpc。
  - 序列化：hessian、java、protobuf。
  - 上下文的 Key。

- 如果扩展实现自身的常量，请自行维护。

  例如 BOLT 协议的常量：

  - `SERIALIZE_CODE_HESSIAN = 1`
  - `PROTOCOL_TR = 13`

  例如 DSR 配置中心相关的常量 `_WEIGHT`、`_CONNECTTIMEOUT`，这种配置中心特有的 key。

地址

地址信息放到 `ProviderInfo` 类中。`ProviderInfo` 的值主要分为三部分：

- 字段：一般是一些必须项目，例如 IP、端口、状态等。
- 静态字段：例如应用名。
- 动态字段：例如预热权重等。

字段枚举维护在 `ProviderInfoAttrs` 类中。

11.4. 关键配置类介绍

本文介绍 RPC 的关键配置类，您可以基于这些配置类进行开发。

ProviderConfig

属性	默认值	说明
id	自动生成	ID
application	空 ApplicationConfig	设置应用对象。
interfaceId	-	设置服务接口（唯一标识元素）。 无论是普通调用还是返回调用，这里都设置实际的接口类。
uniqueId	-	设置服务标签（唯一标识元素）。
filterRef	-	过滤器配置实例。 List
filter	-	过滤器配置别名。 List，多个别名使用英文逗号（,）分隔。

属性	默认值	说明
registry	-	设置服务端注册中心。 List
methods	-	方法级配置，配置格式： `Map<String, MethodConfig>` 。
serialization	hessian2	设置序列化协议。
register	true	是否注册服务。 取决于实现，可能不生效。
subscribe	true	是否订阅服务。 取决于实现，可能不生效。
proxy	javassist	设置代理类型，支持 javassist 代理和 JDK 动态代理。
ref	-	服务接口实现类
server	-	服务端 List，可以一次发到多个服务端。
delay	0	设置服务延迟发布时间。 设置为 -1 代表 spring 加载完毕（通过 spring 才生效）。
weight	-	设置服务静态权重。
include	-	发布的方法列表
exclude	-	不发布的方法列表。
dynamic	true	是否为动态注册。 配置为 false 表示不主动发布，您需要到管理端进行上线操作。

属性	默认值	说明
priority	-	设置服务优先级。 数值越大，优先级越高。
bootstrap	bolt	设置服务发布启动器。
executor	-	设置自定义线程池。
timeout	-	设置服务端执行超时时间。 单位：毫秒 超时后不会打断执行线程，只是打印警告。
concurrents	-	设置接口下每个方法的最大可并行执行请求数，取值如下： • -1：关闭并发过滤器。 • 0：开启过滤，但不限制并发执行数量。
cacheRef	-	结果缓存实现类
mockRef	-	Mock 实现类
mock	-	是否开启 Mock。
validation	-	是否开启参数验证（基于 JSR303）。
compress	false	是否启动压缩。
cache	false	是否启用结果缓存。
parameters	-	额外属性，格式：`Map<String, String>`

ConsumerConfig

属性	默认值	说明
id	自动生成	ID
application	空 ApplicationConfig	设置应用对象。
interfaceId	-	设置服务接口（唯一标识元素）。 无论是普通调用还是返回调用，这里都设置实际的接口类。
uniqueId	-	设置服务标签（唯一标识元素）。
filterRef	-	设置过滤器配置实例。 List
filter	-	设置过滤器配置别名。 List，多个别名使用英文逗号（,）分隔。
registry	-	设置服务端注册中心。 List
methods	-	方法级配置，格式： `Map<String, MethodConfig>`
serialization	hessian2	设置序列化协议。
register	true	是否注册服务。 取决于实现，可能不生效。
subscribe	true	是否订阅服务。 取决于实现，可能不生效。
proxy	javassist	设置代理类型，支持 javassist 代理和 JDK 动态代理。
protocol	bolt	设置调用的协议，目前支持 bolt、rest、dubbo。

属性	默认值	说明
directUrl		设置直连地址，用于直连后 register。
generic	false	是否为泛化调用。
connectTimeout	3000(cover 5000)	设置建立连接超时时间。 单位：毫秒
disconnectTimeout	5000(cover 10000)	设置断开连接等待超时时间。 单位：毫秒
cluster	failover	设置集群模式。
connectionHolder	all	设置连接管理器实现。
loadBalancer	random	设置负载均衡算法。
lazy	false	是否延迟建立长连接。 第一次调用时创建。
sticky	false	是否使用粘性连接。 取值为 true 时，跳过负载均衡算法，使用上一个地址（一个连接断开后，才会选下一个地址）。
injVM	true	是否转为 JVM 调用。 配置为 true 时，由 JVM 发现服务提供者，使用本地配置。
check	false	设置是否检查强依赖。 配置为 true 时，若无可用服务端，会导致启动失败。
heartbeat	30000	设置客户端给服务端发送的心跳间隔。 取决于实现，可能不生效。
reconnect	10000	设置客户端重建端口长连接的间隔。 取决于实现，可能不生效。

属性	默认值	说明
router	-	设置路由器配置别名。 List
routerRef	-	设置路由器配置实例。 List
bootstrap	bolt	设置服务引用启动器。
addressWait	-1	设置等待地址获取时间。 取决于实现，可能不生效。
timeout	3000(cover 5000)	设置调用超时时间。 单位：毫秒
retries	0	设置失败后重试次数。 跟集群模式有关，failover 会读取此参数。
invokeType	sync	设置调用类型，取值如下： • sync：同步调用，Bolt 默认的调用方式。 • oneway：异步调用，消费方发送请求后直接返回，忽略提供方的处理结果。 • callback：异步调用，消费方提供一个回调接口，当提供方返回后，SOFA 框架会执行回调接口。 • future：异步调用，消费方发起调用后马上返回，当需要结果时，消费方需要主动去获取数据。
onReturn	-	接口下每个方法的最大可并行执行请求数，取值如下： • -1：关闭并发过滤器。 • 0：开启过滤，但不限制最大请求数。
cacheRef	-	结果缓存实现类。
mockRef	-	Mock 实现类。
cache	false	是否启用结果缓存。

属性	默认值	说明
mock	-	是否开启 Mock。
validation	-	是否开启参数验证（基于 JSR303）。
compress	false	是否启动压缩。
parameters	-	额外属性，格式：`Map<String, String>`

MethodConfig

属性	默认值	备注
name	-	设置方法名。 不能用于重载方法。
timeout	-	设置调用超时时间。 单位：毫秒
retries	-	设置失败后重试次数。
invokeType	-	设置调用类型，取值如下： • sync：同步调用，Bolt 默认的调用方式。 • oneway：异步调用，消费方发送请求后直接返回，忽略提供方的处理结果。 • callback：异步调用，消费方提供一个回调接口，当提供方返回后，SOFA 框架会执行回调接口。 • future：异步调用，消费方发起调用后马上返回，当需要结果时，消费方需要主动去获取数据。
validation	-	是否开启参数验证（基于JSR303）。
onReturn	-	返回时调用的 SofaResponseCallback，用于实现 Callback 等。

属性	默认值	备注
concurrent	-	接口下每个方法的最大可并行执行请求数，取值如下： • -1：关闭并发过滤器。 • 0：开启过滤，但不限制最大请求数。
validation	-	是否开启参数验证。
compress	-	是否启动压缩。
parameters	-	额外属性，格式：`Map<String, String>`。

ServerConfig

属性	默认值	备注
protocol	bolt	调用的协议，目前支持 bolt、rest、dubbo。
host	0.0.0.0	实际监听 IP，与网卡地址对应。
port	12200	协议的端口，默认端口如下： • bolt：12200 • rest：8341 • h2c：12300 • dubbo：20880
contextPath	-	设置上下文路径。
ioThreads	0	设置 IO 线程池数。 取决于实现，可能不生效。例如 Bolt 默认的默认线程数为 cpu 数量*2。0 表示自动计算。
threadPoolType	cached	设置业务线程池类型。
coreThreads	80(override 20)	设置业务线程池核心大小。

属性	默认值	备注
maxThreads	400(override 200)	设置业务线程池最大值。
telnet	true	是否允许 telnet。 取决于实现，可能不生效。例如 Bolt 不支持 telnet。
queueType	normal	设置业务线程池类型。 可用于实现优先级队列等。
queues	1000(override 0)	设置业务线程池队列。
aliveTime	300000(override 60000)	设置业务线程池存活时间。 单位：毫秒
preStartCore	-	是否初始化核心线程。
accepts	100000	设置最大长连接数量。 取决于实现，可能不生效。
serialization	hesisan2	设置序列化协议。
virtualHost	-	设置虚拟主机地址，注册到注册中心时优先使用该地址。
virtualPort	-	设置虚拟主机端口，注册到注册中心时优先使用该端口。
epoll	false	是否启动 epoll。 取决于实现，可能不生效。
daemon	true	是否守护端口，取值如下： • true：随主线程退出而退出。 • false：需要主动退出。
adaptivePort	false	是否调整端口。 设置为 true 时，若端口被占用，则会自动将端口号 +1 进行适应。

属性	默认值	备注
transport	bolt (cover netty4)	传输层实现 取决于实现，可能不生效。
autoStart	true	是否自动启动端口。
stopTimeout	10000(override 20000)	设置优雅关闭超时时间。 单位：毫秒
boundHost	-	绑定的地址，默认取 host 值。
parameters	-	额外属性，格式：`Map<String, String>`。

RegistryConfig

属性	默认值	备注
protocol	zookeeper	服务协议，目前支持 zookeeper 和 local。
address	-	指定注册中心地址。 address 和 index 属性必须配置一个。
index	-	指定注册中心寻址服务的地址。 address 和 index 属性必须配置一个。
register	true	是否注册服务。
subscribe	true	是否订阅服务。
timeout	10	调用注册中心超时时间 单位：秒
connectTimeout	20	连接注册中心超时时间 单位：秒

属性	默认值	备注
file	$HOME	local 协议时使用的本地文件位置，默认在 `$HOME` 下。

11.5. 自定义 Registry

> **说明**
>
> 自定义注册中心的实现类可参考 `com.alipay.sofa.rpc.registry.zk.ZookeeperRegistry` ，实现的具体信息请参见 Registry-ZK。

自定义注册中心的流程如下：

实现自定义注册中心类

1. 自定义注册中心类。

   自定义注册中心类时，需继承如下虚拟类：

   ```
   package com.alipay.sofa.rpc.registry;

   @Extensible(singleton =false)
   public abstract class Registry implements Initializable,Destroyable{
       public abstract boolean start();  //启动客户端
       public abstract void register(ProviderConfig config);  //注册服务
       public abstract void unRegister(ProviderConfig config);  //取消注册，优雅关闭使用
       public abstract void batchUnRegister(List<ProviderConfig> configs); //批量取消注册
       public abstract List<ProviderGroup> subscribe(ConsumerConfig config); //订阅服务
       public abstract void unSubscribe(ConsumerConfig config); //取消订阅，优雅关闭使用
       public abstract void batchUnSubscribe(List<ConsumerConfig> configs); //批量取消订阅
   }
   ```

2. 添加注解。

   示例如下：

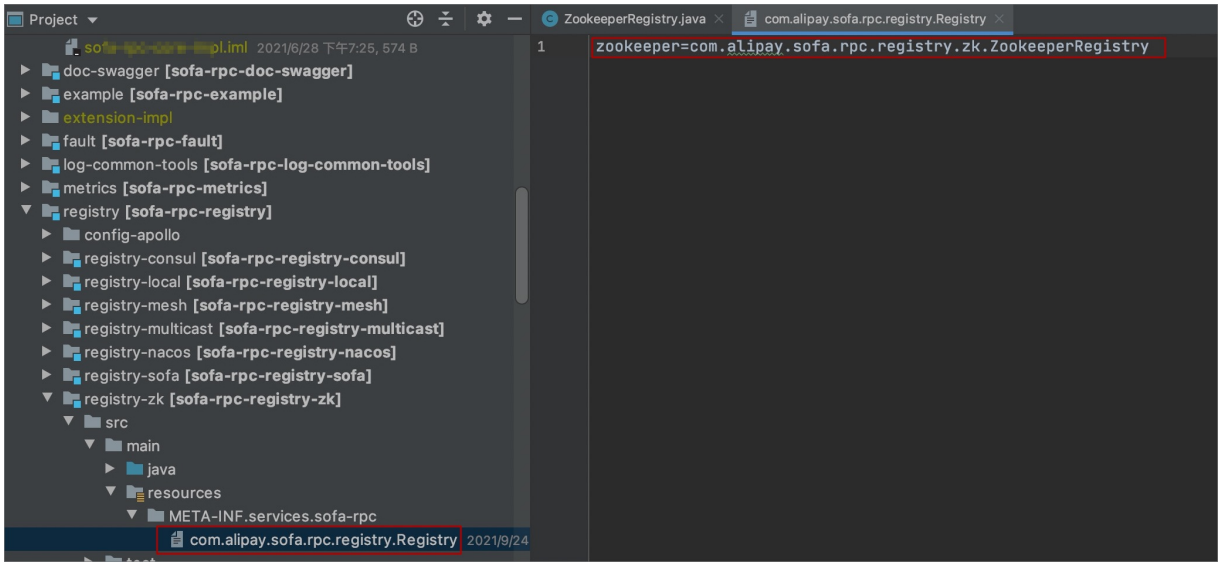
   ```
   @Extension("zookeeper")
   public class ZookeeperRegistry extends Registry {
   ```

   `@Extension("")` 用于指定注册中心别名，可自定义。您可以在业务指定注册中心时使用该别名。

将自定义注册中心配置到 RPC 架构

基于 SOFARPC 的 SPI 机制，您需要新建扩展文件 `META-INF/services/sofa-rpc/com.alipay.sofa.rpc.registry.Registry` ，将自定义的注册中心配置到 RPC 架构。内容示例如下：

```
zookeeper=com.alipay.sofa.rpc.registry.zk.ZookeeperRegistry
```



在业务中使用自定义注册中心

1. 在 `application.properties` 中配置注册中心源信息。

   配置格式为：`com.alipay.sofa.rpc.registries.<注册中心用例名称>=协议://地址`，协议名称为上面自定义的注册中心名称，例如 `zookeeper`。示例如下：

   ```
   com.alipay.sofa.rpc.registries.myRegistry=zookeeper://10.0.0.1:9600  //配置注册中心地址。
   ```

2. 配置需要注册中心的服务。

   注册中心名称为上面自定义的注册中心用例名。示例如下：

   ```
   <sofa:service interface="com.alipay.sofa.facade.SampleService" ref="sampleService">
           <sofa:binding.bolt>
               <sofa:global-attrs registry="myRegistry"/>
           </sofa:binding.bolt>

       </sofa:service>
   ```

11.6. 自定义 Filter

SOFARPC 提供了一套良好的可扩展性机制，为各个模块提供 SPI（Service Provider Interface） 的能力。

SOFARPC 对请求与响应的过滤链（FilterChain）处理方式为：

- 通过多个过滤器 Filter 来进行具体的拦截处理。
- 允许用户自定义 Filter 扩展，且自定义 Filter 的执行顺序在内置 Filter 之后。

下文将就自定义 Filter 类及其生效进行说明：

自定义 Filter 类

自定义 Filter 类需继承 `com.alipay.sofa.rpc.filter.Filter` 类。示例如下：

▶ Details

自定义 Filter 类生效方式

自定义 Filter 类支持如下生效方式：

- API 方式

  该方式可以指定生效的 provider 或 consumer 。示例如下：

  ▶ Details

- @Extension 注解 + 扩展文件 + 编码注入

  处理步骤如下：

  i. 处理自定义 Filter 类。

  ▶ Details

  ii. 创建扩展文件。

  - 文件名：以 Filter 为后缀。例如：`com.alipay.sofa.rpc.filter.Filter` 。
  - 文件路径：为固定路径。位于 `META-INF/services/sofa-rpc/` 下。

  完整示例：`META-INF/services/sofa-rpc/com.alipay.sofa.rpc.filter.Filter` 。扩展文件内容如下：

  ```
  customer=com.alipay.sofa.rpc.custom.CustomFilter
  ```

  iii. 注入编码。

  ▶ Details

- @Extension 注解 + 扩展文件 + @AutoActive 注解

  处理步骤如下：

  i. 处理自定义 Filter 类。

  ▶ Details

  > **说明**
  >
  > 使用 2 个注解 `@Extension` 和 `@AutoActive` 处理 Filter 类可以免去编码注入。作用域为所有 provider 或 consumer。其中，参数 `providerSide` 表示是否生效于服务端；参数 `consumerSide` 表示是否生效于客户端。

  ii. 创建扩展文件。

  - 文件名：以 Filter 为后缀。例如 `com.alipay.sofa.rpc.filter.Filter` 。
  - 文件路径：固定路径，位于 `META-INF/services/sofa-rpc/` 下。

完整示例：`META-INF/services/sofa-rpc/com.alipay.sofa.rpc.filter.Filter`。扩展文件内容如下：

```
customer=com.alipay.sofa.rpc.custom.CustomFilter
```

11.7. 自定义 Router

SOFARPC 中对服务地址的选择也抽象为了一条处理链，由每一个 Router 进行处理。同 Filter 一样，SOFARPC 对 Router 提供了同样的扩展能力。本文介绍如何自定义 Router 类。

自定义 Router 类的流程如下：

实现自定义 Router 类

自定义 Router 类时，需继承 `com.alipay.sofa.rpc.client.Router` 类：

```
@Extension(value ="customerRouter")
@AutoActive(consumerSide =true)
public class CustomerRouter extends Router {

    @Override
    public void init(ConsumerBootstrap consumerBootstrap) {

    }
    @Override
    public boolean needToLoad(ConsumerBootstrap consumerBootstrap) {
        return true;
    }
    @Override
    public List<ProviderInfo> route(SofaRequest request, List<ProviderInfo> providerInfos)
{
        return providerInfos;
    }
}
```

RPC 框架支持通过 `@Extension` 注解来扩展 Router，并且可以覆盖重名的 Router。如需覆盖原有 Router，需将 `@Extension` 中 `override` 置为 `true`，并确保 `order` 属性值大于要覆盖的重名 Router。未配置 `order` 属性时，默认为 0。

如上自定义了一个 `CustomerRouter`，生效于所有消费者。其中：

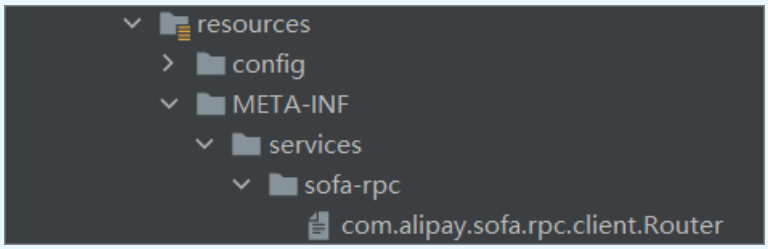
- `init` 参数 `ConsumerBootstrap` 是引用服务的包装类，能够拿到 `ConsumerConfig`、代理类、服务地址池等对象。
- `needToLoad` 表示是否生效该 Router。`route` 方法即筛选地址的方法。

将自定义 Router 类配置到 RPC 架构

基于 SOFARPC 的 SPI 机制，您需要在类路径中新建扩展文件 `META-INF/services/sofa-rpc/com.alipay.sofa.rpc.client.Router`，将自定义的 Router 类配置到 RPC 架构。内容示例如下：

```
customerRouter=com.alipay.sofa.rpc.custom.CustomRouter
```

> 注意
>
> `resource` 内的目录和文件名需要遵守如下固定规则：



11.8. 自定义 ShutdownHook

优雅关闭涉及两方面，一个是 RPC 框架作为客户端，一个是 RPC 框架作为服务端。

作为服务端

RPC 框架作为服务端，在关闭时，不应该直接暴力关闭。关闭方式如下：

1. 查看 RPC 框架：

   ```
   com.alipay.sofa.rpc.context.RpcRuntimeContext
   ```

2. 在静态初始化块中，添加一个 `ShutdownHook` 。

   ```
       // 增加 JVM 关闭事件。
       if(RpcConfigs.getOrDefaultValue(RpcOptions.JVM_SHUTDOWN_HOOK,true)){
           Runtime.getRuntime().addShutdownHook(newThread(newRunnable(){
               @Override
               public void run(){
                   if(LOGGER.isWarnEnabled()){
                       LOGGER.warn("SOFA RPC Framework catch JVM shutdown event, Run shutd
   own hook now.");
                   }
                   destroy(false);
               }
           },"SOFA-RPC-ShutdownHook"));
       }
   ```

   `ShutdownHook` 的作用是当发布平台或用户执行 `kill pid` 时，会先执行 `ShutdownHook` 中的逻辑。在销毁操作中，RPC 框架会先执行向注册中心取消服务注册、关闭服务端口等动作。示例如下：

```
private static void destroy(boolean active){
        RpcRunningState.setShuttingDown(true);
        for(Destroyable.DestroyHook destroyHook : DESTROY_HOOKS){
            destroyHook.preDestroy();
        }
        List<ProviderConfig> providerConfigs = new ArrayList<ProviderConfig>();
        for(ProviderBootstrap bootstrap : EXPORTED_PROVIDER_CONFIGS){
            providerConfigs.add(bootstrap.getProviderConfig());
        }
        // 先反注册服务端。
        List<Registry> registries =RegistryFactory.getRegistries();
        if(CommonUtils.isNotEmpty(registries) && CommonUtils.isNotEmpty(providerConfigs
)){
            for(Registry registry : registries){
                registry.batchUnRegister(providerConfigs);
            }
        }
        // 关闭启动的端口。
        ServerFactory.destroyAll();
        // 关闭发布的服务。
        for(ProviderBootstrap bootstrap : EXPORTED_PROVIDER_CONFIGS){
            bootstrap.unExport();
        }
        // 关闭调用的服务。
        for(ConsumerBootstrap bootstrap : REFERRED_CONSUMER_CONFIGS){
            ConsumerConfig config = bootstrap.getConsumerConfig();
                if(!CommonUtils.isFalse(config.getParameter(RpcConstants.HIDDEN_KEY_DES
TROY))){
                // 除非不让主动 unrefer。
                bootstrap.unRefer();
            }
        }
        // 关闭注册中心。
        RegistryFactory.destroyAll();
        // 关闭客户端的一些公共资源。
        ClientTransportFactory.closeAll();
        // 卸载模块。
        if(!RpcRunningState.isUnitTestMode()){
            ModuleFactory.uninstallModules();
        }
        // 卸载钩子。
        for(Destroyable.DestroyHook destroyHook : DESTROY_HOOKS){
            destroyHook.postDestroy();
        }
        // 清理缓存。
        RpcCacheManager.clearAll();
        RpcRunningState.setShuttingDown(false);
        if(LOGGER.isWarnEnabled()){
            LOGGER.warn("SOFA RPC Framework has been release all resources {}...",
                    active ?"actively ":"");
        }
 }
```

其中以 Bolt 为例，关闭端口并不是一个立刻执行的动作，而是会判断当前服务端上面的连接和队列的任务，先处理完队列中的任务再缓慢关闭。

```
    @Override
    public void destroy(){
        if(!started){
            return;
        }
        int stopTimeout = serverConfig.getStopTimeout();
        if(stopTimeout >0){// 需要等待结束时间
            AtomicInteger count = boltServerProcessor.processingCount;
            // 有正在执行的请求 或者 队列里有请求
            if(count.get()>0|| bizThreadPool.getQueue().size()>0){
                long start =RpcRuntimeContext.now();
                if(LOGGER.isInfoEnabled()){
                    LOGGER.info("There are {} call in processing and {} call in queue,
wait {} ms to end",
                            count, bizThreadPool.getQueue().size(), stopTimeout);
                }
                while((count.get()>0|| bizThreadPool.getQueue().size()>0)
                        && RpcRuntimeContext.now()- start < stopTimeout){
                    // 等待返回结果
                    try{
                        Thread.sleep(10);
                    }catch(InterruptedException ignore){
                    }
                }
            }
            // 关闭前检查已有请求?
        }
        // 关闭线程池
        bizThreadPool.shutdown();
        stop();
    }
```

作为客户端

RPC 框架作为客户端时，实际上就是 Cluster 的关闭。关闭调用的服务这一步，可以查看 `com.alipay.sofa.rpc.client.AbstractCluster` 。

```
    /**
     * 优雅关闭的钩子。
     */
    protected class GracefulDestroyHook implements DestroyHook{
        @Override
        public void preDestroy(){
            // 准备关闭连接。
            int count = countOfInvoke.get();
            final int timeout = consumerConfig.getDisconnectTimeout();// 等待结果超时时间。
            if(count >0){// 有正在调用的请求。
                long start =RpcRuntimeContext.now();
                if(LOGGER.isWarnEnabled()){
                    LOGGER.warn("There are {} outstanding call in client, will close transp
orts util return",
                                count);
                }
                while(countOfInvoke.get()>0 && RpcRuntimeContext.now()- start < timeout){//
等待返回结果。
                    try{
                        Thread.sleep(10);
                    }catch(InterruptedException ignore){
                    }
                }
            }
        }

        @Override
        public void postDestroy(){
        }
    }
```

客户端会逐步将正在调用的请求处理完成才会下线。

> **说明**
>
> 优雅关闭是需要和发布平台联动的。如果强制 kill，那么任何优雅关闭的方案都不会生效。后续会考虑在 SOFABoot 层面提供一个统一的 API，来给发布平台调用，而不是依赖 hook 的逻辑。

11.9. 单元测试与性能测试

本文主要介绍在进行 SOFARPC 开发时如何进行单元测试与性能测试。

单元测试

将单元测试的示例放到您自己开发的模块下。如果依赖了第三方服务端（例如 Zookeeper），请手动加入 profile，并放到 `test-intergrated-3rd` 模块中。参考 `registry-zookeeper` 模块代码。如果依赖了其它模块要集成测试，请放到 `test/test-intergrated` 模块中。

性能测试

1. 在启动参数中修改以下参数，以关闭相关项目。

```
-Dcontext.attachment.enable=false
-Dserialize.blacklist.enable=false
-Ddefault.tracer= false
-Dlogger.impl=com.alipay.sofa.rpc.log.SLF4JLoggerImpl
-Dmultiple.classloader.enable=false
-Devent.bus.enable=false
```

2. 对 `bolt+hessian` 进行压测。

   压测环境如下：

   - 服务端：4C8G 虚拟机、千兆网络、JDK1.8.0_111。
   - 客户端：50 个客户端并发请求。

   压测结果：

协议	请求	响应	服务端	TPS	平均 RT(ms)
bolt+hessian	1000 String	1000 String	直接返回	10000	1.93
bolt+hessian	1000 String	1000 String	直接返回	20000	4.13
bolt+hessian	1000 String	1000 String	直接返回	30000	7.32
bolt+hessian	1000 String	1000 String	直接返回	40000	15.78
bolt+hessian	1000 String	1000 String	直接返回	50000（接近极限，错误率 0.3%）	26.51

12. 日志说明

当您使用 SOFARPC 启动应用程序以后，默认情况下，RPC 会创建 logs 目录，并生成相关日志文件。

日志列表

日志名称	说明
rpc/rpc-registry.log	服务地址订阅与接收日志。
rpc/tr-threadpool	服务连接池日志（SOFABoot 支持该日志）。
rpc/rpc-default.log	SOFARPC INFO、WARN 日志，无标准格式。
rpc/common-error.log	SOFARPC 错误日志，无标准格式。
rpc/rpc-remoting.log	网络层日志，无标准格式。
rpc/sofa-router.log	SOFARouter 相关日志，无标准格式。
rpc/rpc-remoting-serialization.log	网络层序列化日志，无标准格式。
tracelog/rpc-client-digest.log	SOFARPC 调用客户端摘要日志。
tracelog/rpc-server-digest.log	SOFARPC 调用服务端摘要日志。
tracelog/rpc-profile.log	SOFARPC 处理性能日志。
confreg/config.client.log	服务注册中心客户端日志。

SOFATracer 日志

SOFATracer 集成在 SOFARPC（5.4.0 及之后的版本）后，还会输出如下链路数据日志。

> **说明**
>
> - 开源版的日志默认为 JSON 格式，企业版默认以逗号分隔。
> - 日志会不定期新增部分字段，新增字段会从日志尾部添加，不会影响原日志字段。若您实际打印的日志与本文中日志字段数不一致，请按顺序进行对比，新增字段可咨询售后技术支持。

RPC 客户端摘要日志

`rpc-client-digest.log` 是 RPC 客户端摘要日志，日志样例如下：

```
2021-09-27 16:43:59.096,myserver-app,1ecee1741632732******410896596,0,com.alipay.samples.rpc.SampleService:1.0,hello,bolt,,127.0.0.1,myclient-app,,,1ms,0ms,SOFA-SEV-BOLT-BIZ-12201-9-T20,00,,,0ms,,
```

对应 key 的说明如下：

key	说明
timestamp	日志打印时间
local.app	客户端应用名称
tracerId	TraceId
spanId	SpanId
service	服务接口信息
method	调用方法
span.kind	Span 类型
protocol	协议类型。取值：bolt、rest。
invoke.type	调用类型。取值如下： • sync：同步调用，Bolt 默认的调用方式。 • oneway：异步调用，消费方发送请求后直接返回，忽略提供方的处理结果。 • callback：异步调用，消费方提供一个回调接口，当提供方返回后，SOFA 框架会执行回调接口。 • future：异步调用，消费方发起调用后马上返回，当需要结果时，消费方需要主动去获取数据。
remote.ip	目标 IP
remote.app	服务端应用名称

key	说明
remote.zone	目标 zone
remote.idc	目标 IDC
remote.city	目标城市
user.id	用户 ID
result.code	结果码。取值如下： • 00：请求成功。 • 01：业务异常。 • 02：RPC 逻辑错误。 • 03：请求超时。 • 04：路由失败。
req.size	请求数据大小
resp.size	响应数据大小
client.elapse.time	调用总耗时，单位：ms。
client.conn.time	客户端连接耗时，单位：ms。
req.serialize.time	请求序列化耗时，单位：ms。
outtime	超时参考耗时，单位：ms。
current.thread.name	当前线程名
route.record	路由记录，路由选择的过程记录。
elastic.id	弹性数据位

key	说明
be.elastic	本次调用的服务是否需要弹性： • T：表示需要弹性。 • F：表示不需要。
elastic.service.name	转发调用的方法名。
local.client.ip	源 IP
local.client.port	本地客户端端口
local.zone	本地 zone
target.ip.in.one.physical	目标 IP 是否在当前物理机： • T：表示在同一物理机。 • F：表示不在同一物理机。
sys.baggage	系统透传的 baggage 数据
bus.baggage	业务透传的 baggage 数据
send.time	RPC 请求耗时
phase.time	各阶段耗时，单位：ms。
special.time.mark	特殊时间点标记
router.forward	路由转发详情

RPC 服务端摘要日志

`rpc-server-digest.log` 是 RPC 服务端摘要日志，日志样例如下：

```
2014-06-19 17:14:35.006,client,0ad1348f140****2750021003,0.1,com.alipay.cloudenginetest.services.SofaApiWebReferenceLocalFalseTrService:1.0,service_method,TR,,10.**.**.143,client,,,12ms,0ms,HSFBizProcessor-4-thread-2,00,,F,1000ms,zue:l;ztg:r,mark=T&uid=a2&
```

对应 key 的说明如下：

key	说明
timestamp	日志打印时间
local.app	客户端应用名称
tracerId	TraceId
spanId	SpanId
service	服务接口信息
method	调用方法
protocol	协议类型。取值：bolt、rest。
remote.ip	目标 IP
remote.app	服务端应用名称
invoke.type	调用类型。取值如下： • sync：同步调用，Bolt 默认的调用方式。 • oneway：异步调用，消费方发送请求后直接返回，忽略提供方的处理结果。 • callback：异步调用，消费方提供一个回调接口，当提供方返回后，SOFA 框架会执行回调接口。 • future：异步调用，消费方发起调用后马上返回，当需要结果时，消费方需要主动去获取数据。
remote.ip	调用方 IP
remote.app	调用方应用名
remote.zone	调用方 zone
remote.idc	调用方 IDC

key	说明
biz.impl.time	请求处理耗时，单位：ms。 不包含服务端响应序列化耗时和反序列化耗时。
resp.serialize.time	响应序列化耗时，单位：ms。
current.thread.name	当前线程名
result.code	返回码，取值如下： • 00：请求成功。 • 01：业务异常。 • 02：RPC 逻辑错误。
elastic.service.name	表明当前是转发调用，包含转发的服务名称和方法值。
be.elasticc	本次服务是否被转发： • T：表示已被转发。 • F：表示未被转发。
server.pool.wait.time	服务端线程池等待时间，单位：ms。
sys.baggage	系统透传的 baggage 数据
bus.baggage	业务透传的 baggage 数据
server.send.time	RPC 请求转发耗时
req.size	请求数据大小
resp.size	响应数据大小
phase.time	各阶段耗时明细
special.time	特殊时间点标记

key	说明
router.forward	路由转发详情

RPC 客户端统计日志

`rpc-client-stat.log` 是 RPC 客户端统计日志。其中，`stat.key` 即本段时间内的统计关键字集合。统计关键字集合唯一确定一组统计数据，包含 `local.app`、`remote.app`、`service.name` 和 `method.name` 字段。日志样例如下：

```
2014-06-19 17:14:02.186,client,client,com.alipay.cloudenginetest.services.SofaApiWebReferenceLocalFalseTrService:1.0,service_method,1,79,Y,T,RZ00B
```

对应 key 的说明如下：

key		说明
time		日志打印时间
stat.key	local.app	客户端应用名称
	remote.app	服务端应用名称
	service.name	服务名
	method.name	方法名
count		调用次数
total.cost.milliseconds		请求总耗时
success		调用结果： • Y：调用成功。 • N：调用失败。

key	说明
load.test.mark	判断当前是否为全链路压测： • T：表示当前为全链路压测。当前线程中能获获取到日志上下文，且上下文中有压测信息。 • F：表示当前非全链路压测。当前线程中不能获取到日志上下文，或上下文中没有压测信息。
remote.zone	目标 zone

RPC 服务端统计日志

`rpc-server-stat.log` 是 RPC 服务端统计日志，以 JSON 格式输出的数据。其中，`stat.key` 即本段时间内的统计关键字集合。统计关键字集合唯一确定一组统计数据，包含 `local.app`、`remote.app`、`service.name` 和 `method.name` 字段。日志样例如下：

```
2014-06-19 17:14:02.186,client,client,com.alipay.cloudenginetest.services.SofaApiWebReferenceLocalFalseTrService:1.0,service_method,1,7,Y,T,GZ00B
```

对应 key 的说明如下：

key		说明
time		日志打印时间
stat.key	local.app	客户端应用名称
	remote.app	服务端应用名称
	service.name	服务名
	method	方法名
count		调用次数
total.cost.milliseconds		请求总耗时

key	说明
success	调用结果： • Y：调用成功。 • N：调用失败。
load.test.mark	判断当前是否为全链路压测： • T：表示当前为全链路压测。当前线程中能获获取到日志上下文，且上下文中有压测信息。 • F：表示当前非全链路压测。当前线程中不能获取到日志上下文，或上下文中没有压测信息。
remote.zone	目标 zone