博学谷
www.boxuegu.com

第十章 Spring Cloud Alibaba Dubbo 高性能服务通信

一样的在线教育，不一样的教学品质

目录 Contents

整合 Nacos 服务发现

小节导学

既然服务通信要使用 Dubbo RPC 的方式，那我们最关心的肯定是 如何实现？与 HTTP 调用方式有何不同？

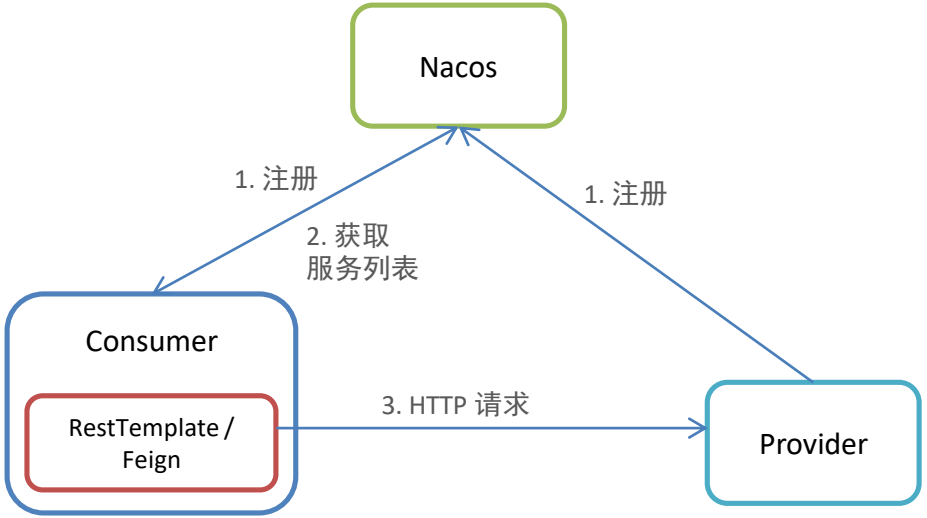
本节我们要弄明白使用 Dubbo 的思路，以及具体的开发方法。

- Dubbo 服务调用流程
- Dubbo 整合 Nacos 服务调用实践

1. Dubbo 服务调用流程

RestTemplate 和 Feign 的服务调用方式

1. Consumer、Provider 注册到服务注册中心
2. Consumer 从注册中心拿到 Provider 地址
3. Consumer 内部发起 HTTP 请求



1. Dubbo 服务调用流程

Dubbo 服务调用方式

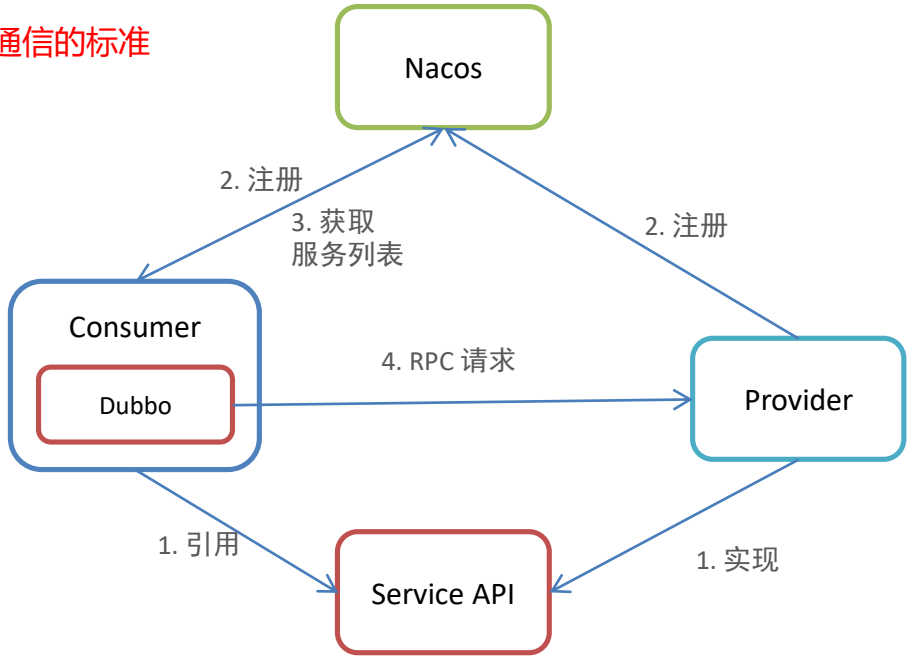
1. 定义一个服务接口

Consumer、Provider 都是使用此接口，作为通信的标准

2. Consumer、Provider 注册到服务注册中心

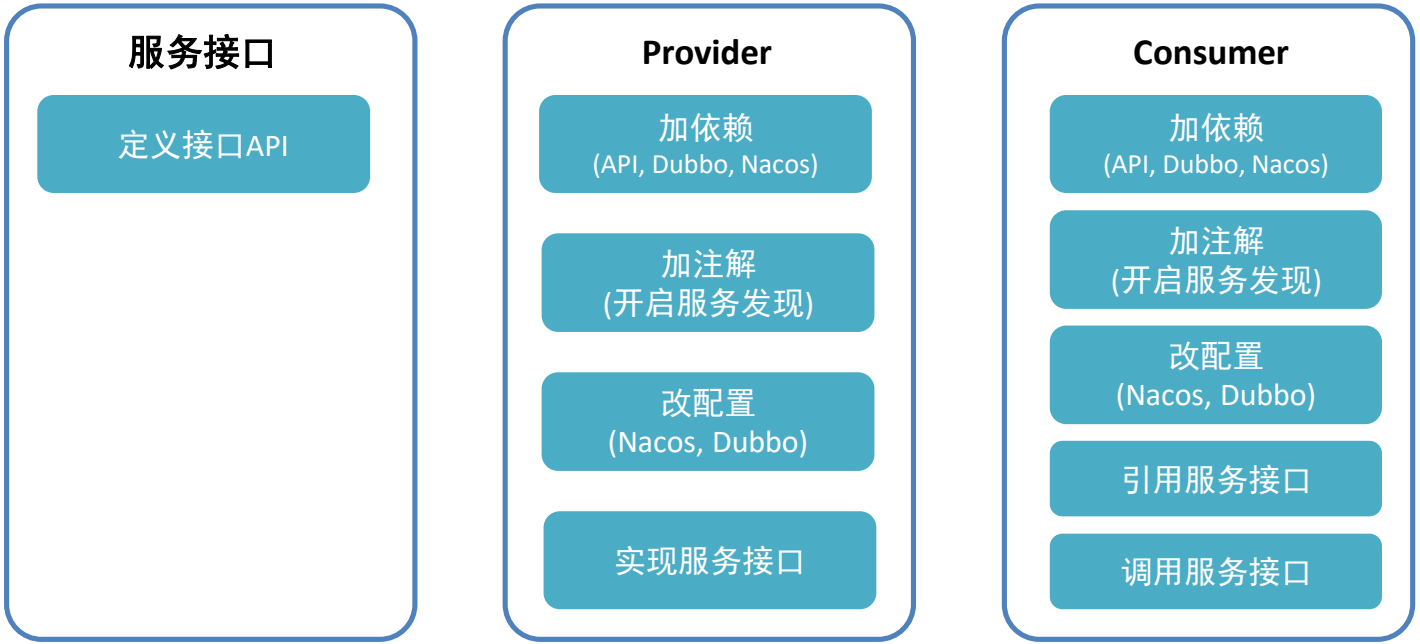
3. Consumer 从注册中心拿到 Provider 地址

4. Consumer 内部发起 RPC 请求



2. Dubbo 整合 Nacos 服务调用实践

开发流程

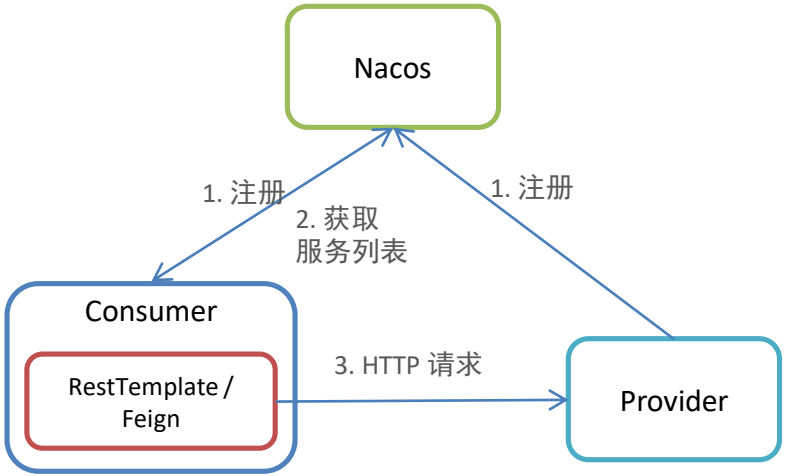


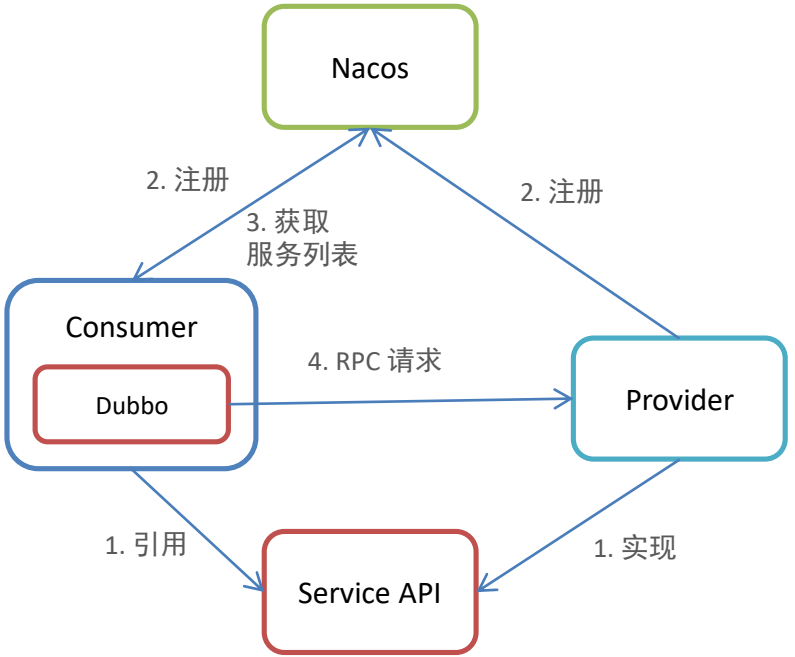
整合 Nacos 服务发现-总结

重难点

1. Dubbo 调用流程与 RestTemplate/Feign 的异同
2. Dubbo 整合 Nacos 实现服务调用的开发流程

Dubbo 调用流程与 RestTemplate/Feign 的异同



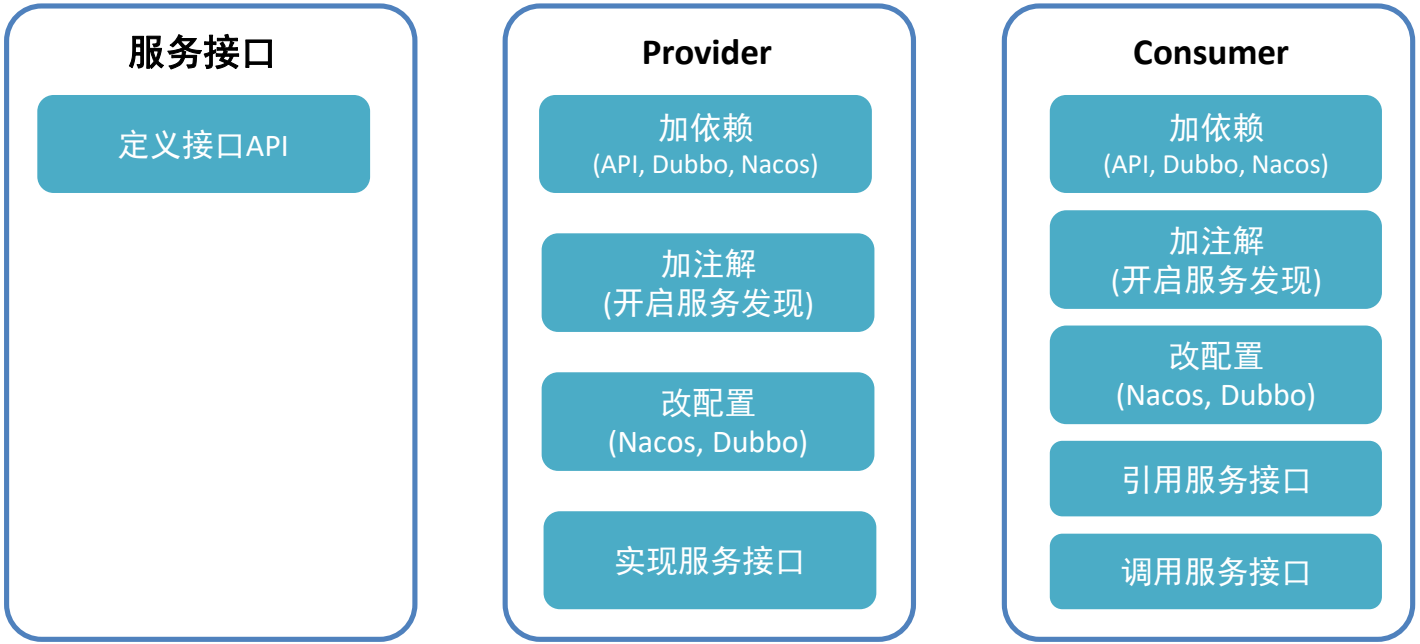


整合 Nacos 服务发现-总结

重难点

1. Dubbo 调用流程与 RestTemplate/Feign 的异同
2. Dubbo 整合 Nacos 实现服务调用的开发流程

Dubbo 整合 Nacos 实现服务调用的开发流程



整合 Nacos 服务发现-总结

重难点

1. Dubbo 调用流程与 RestTemplate/Feign 的异同

2. Dubbo 整合 Nacos 实现服务调用的开发流程

下节

Dubbo 如何与 Nacos Config 整合?

目录 Contents

整合 Nacos 外部配置

小节导学

我们已经完成了 Dubbo 与 Nacos 服务发现的整合，Dubbo 的配置文件是写在配置文件里的。

如果把配置文件放到配置中心，Dubbo 支持外部配置吗？如果支持，对 Nacos Config 是否兼容呢？

本节我们就实践验证一下。

整合 Nacos 外部配置

步骤

1. 添加 Nacos Config 依赖

```
<dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

整合 Nacos 外部配置

步骤

2. 改用 bootstrap.yaml

```
spring:
  application:
    name: hello-dubbo-provider
  cloud:
    nacos:
      config:
        server-addr: localhost:8848
        file-extension: yaml
```

整合 Nacos 外部配置

步骤

3. Nacos Config 添加配置 hello-dubbo-provider.yaml

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
dubbo:
  registry:
    address: spring-cloud://localhost
  scan:
    base-packages: com.example.demo.service
  protocol:
    name: dubbo
    port: -1
server:
  port: 8082
```

整合 Nacos 外部配置

结论

OK

整合 Nacos 外部配置-总结

重难点

1. Dubbo 整合 Nacos Config 的流程

整合 Nacos 外部配置-总结

重难点

1. Dubbo 整合 Nacos Config 的流程

下节

为什么要做 Dubbo 序列化的优化？怎么做？

目录 Contents

高性能序列化

小节导学

Java 自带序列化功能，直接用不就行了，为什么要单独讲?

主要原因:

1. 序列化与反序列化是服务调用过程中的关键性能点，需要得到重视
2. Java 自带的序列化性能较弱，需要改进

本节我们就分析一下序列化这个问题，并实践 Dubbo 如何 使用 Kyro 替代默认的序列化方式。

- 序列化介绍
- Dubbo 应用 Kyro

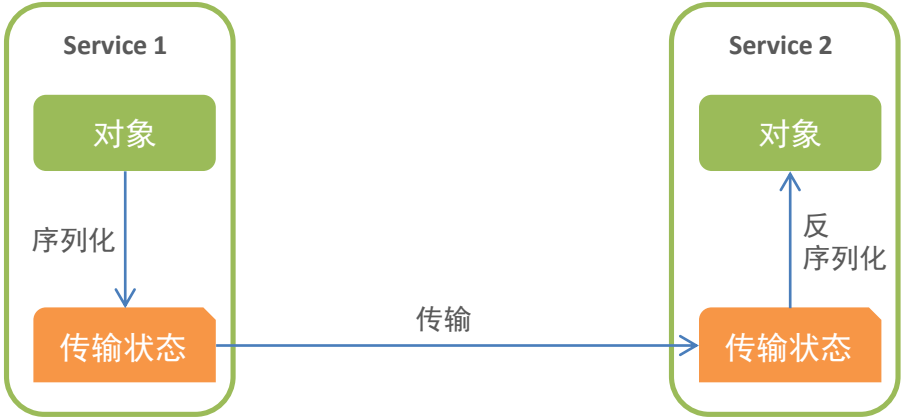
1. 序列化介绍

什么是序列化?

序列化 (Serialization)是将对象状态信息转换为可传输形式的过程。

例如把对象转为字符串，就可以传输。

有“序列化”就同样需要“**反序列化**”，把传输形式转换为对象形式。



1. 序列化介绍

序列化为什么是性能关键点?

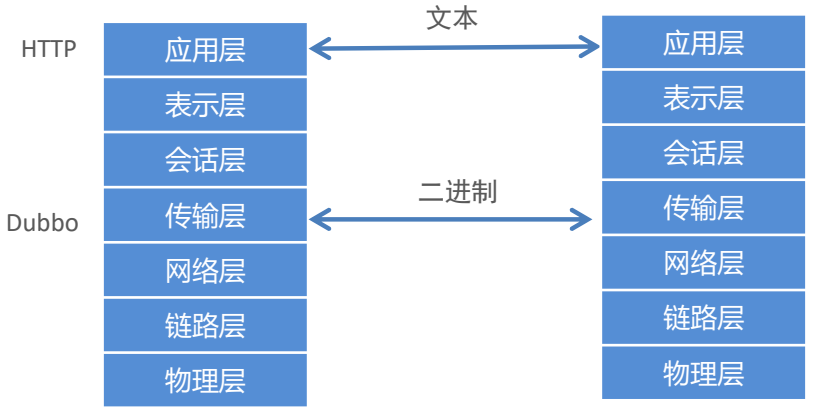
Dubbo RPC 的通信方式在以下几点优于 HTTP 通信方式:

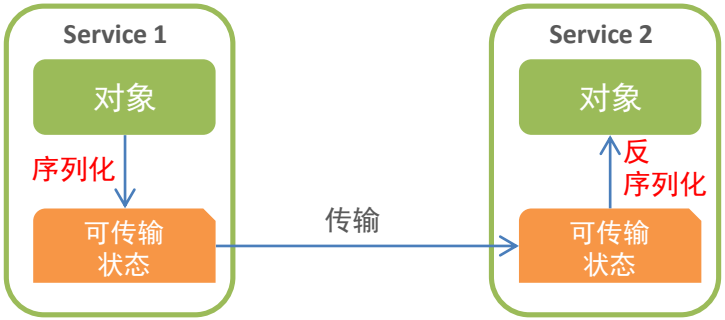
1. 通信协议层次优化，传输层比应用层更高效
2. 二进制数据比文本数据体积更小，传输更快
3. TCP/IP 的长连接比 HTTP 的短连接更高效

这些都是聚焦在“传输”这个过程

“序列化/反序列化”是每次“传输”前、后的操作

其性能至关重要。

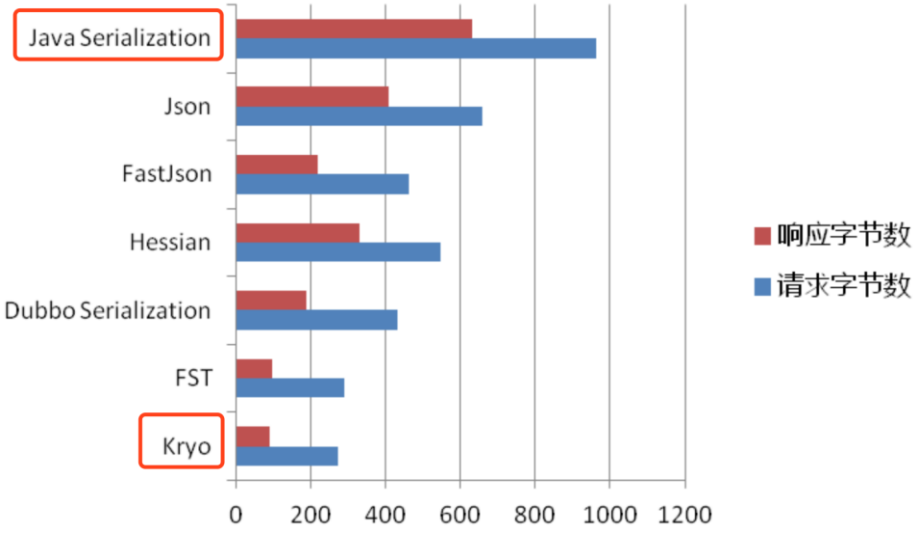




1. 序列化介绍

为什么选择 Kyro?

1. 性能极佳
2. 成熟，Hive、Storm 等知名开源项目在用
3. 专门针对 Java 优化



2. Dubbo 应用 Kyro

步骤

1. Provider、Consumer 中添加 Kyro 依赖

```
<dependency>
            <groupId>org.apache.dubbo</groupId>
            <artifactId>dubbo-serialization-kryo</artifactId>
            <version>2.7.1</version>
</dependency>
```

2. 配置文件中指定使用 Kyro 序列化方式

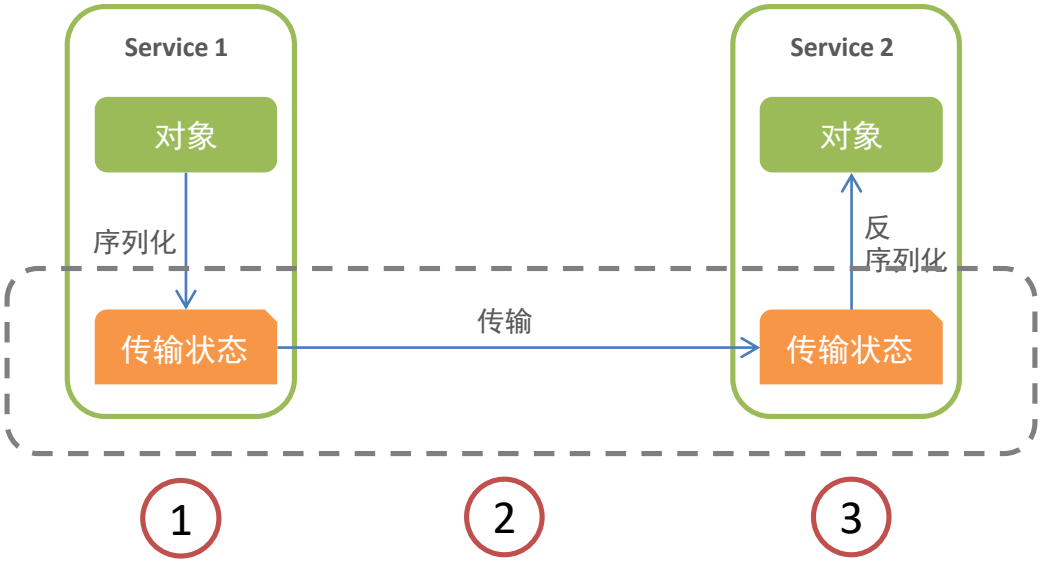
```
dubbo.protocol.serialization=kryo
```

高性能序列化-总结

重难点

1. 序列化对于性能优化的作用
2. Dubbo 使用 Kryo 的方法

序列化的重要性



高性能序列化-总结

重难点

1. 序列化对于性能优化的作用
2. Dubbo 使用 Kryo 的方法

高性能序列化-总结

重难点

1. 序列化对于性能优化的作用
2. Dubbo 使用 Kyro 的方法

下节

Dubbo 如何实现**负载均衡**?

目录 Contents

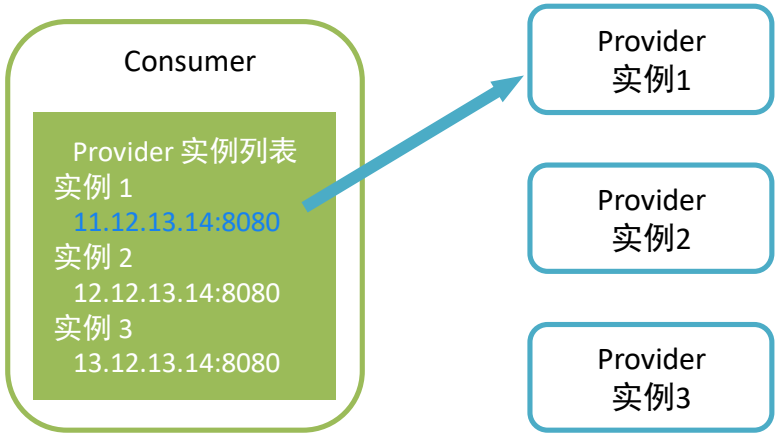
- 整合 Nacos 服务发现
- 整合 Nacos 外部配置
- 高性能序列化
- 负载均衡
- 整合 Sentinel 系统防护
- 整合 SkyWalking 链路跟踪
- Mock 本地伪装
- Stub 本地存根

负载均衡

小节导学

微服务架构中，Service Provider 有多个实例，Service Consumer 需要自己做负载均衡处理。RestTemplate、Feign 都可以实现客户端负载均衡，那么 Dubbo 中如何做负载均衡呢？支持哪些策略呢？本节介绍 Dubbo 支持的负载均衡策略，以及如何配置。

- Dubbo 负载均衡策略
- Dubbo 负载均衡配置



1. Dubbo 负载均衡策略

负载均衡策略 - 随机 (random)

随机选择，就像大转盘游戏，转到哪个就选哪个。

可以设置权重，按权重设置随机概率，就像给游戏机作弊，偏重某些结果。

1. Dubbo 负载均衡策略

负载均衡策略 - 轮询 (roundrobin)

按顺序来，一个接一个的选择。

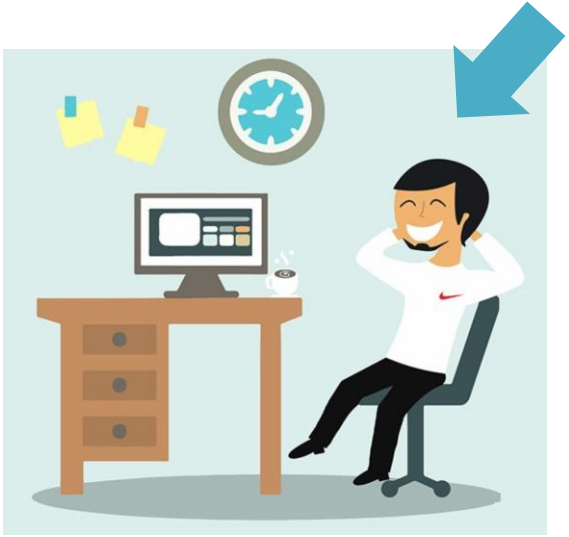
可以设置权重，按权重设置轮询比率。

1. Dubbo 负载均衡策略

负载均衡策略 - 最少活跃调用数 (leastactive)

选择当前压力小的服务实例，谁正在处理的请求少，就选谁。

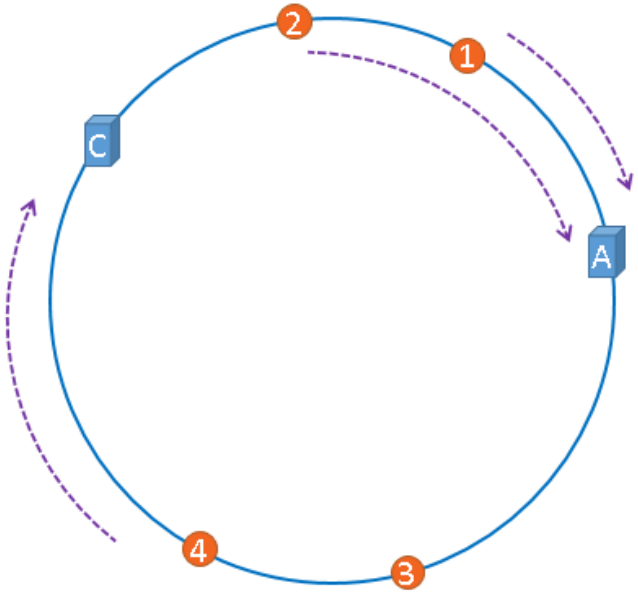
选他



1. Dubbo 负载均衡策略

负载均衡策略 - 一致性hash (consistenthash)

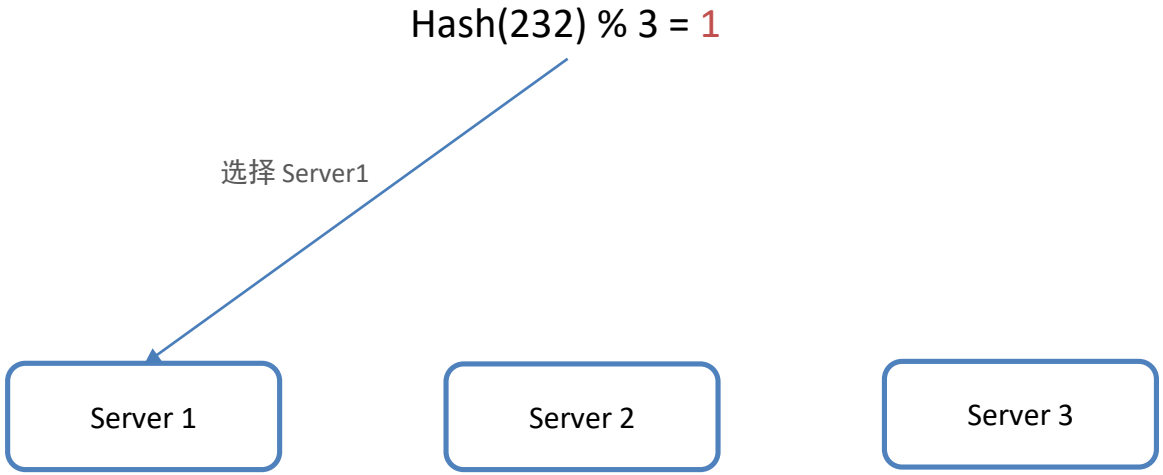
相同参数的请求总是发到同一个provider，当provider宕机后，原本发送给此provider的请求，会转到其他provider，不会引起剧烈变动。



补充：hash 与 一致性hash

普通 hash 方式

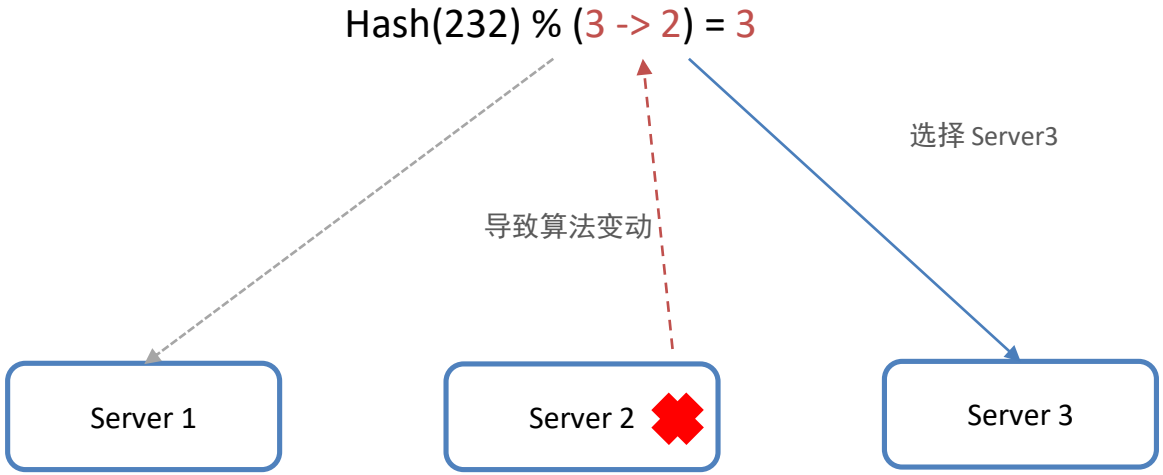
Hash（用户ID）% 服务器数量



补充：hash 与 一致性hash

普通 hash 方式 – 伸缩性差

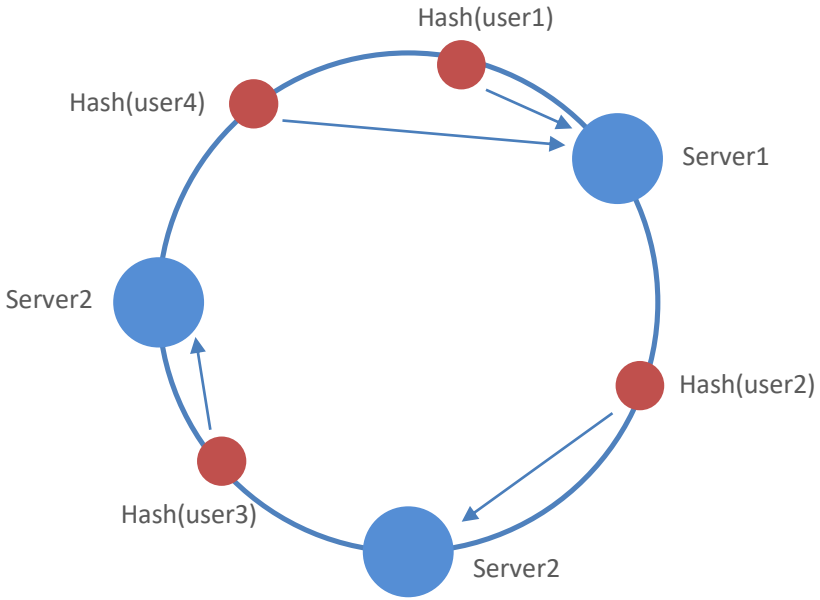
服务器数量变动后，请求转发就重新洗牌



补充：hash 与 一致性hash

一致性 hash 方式

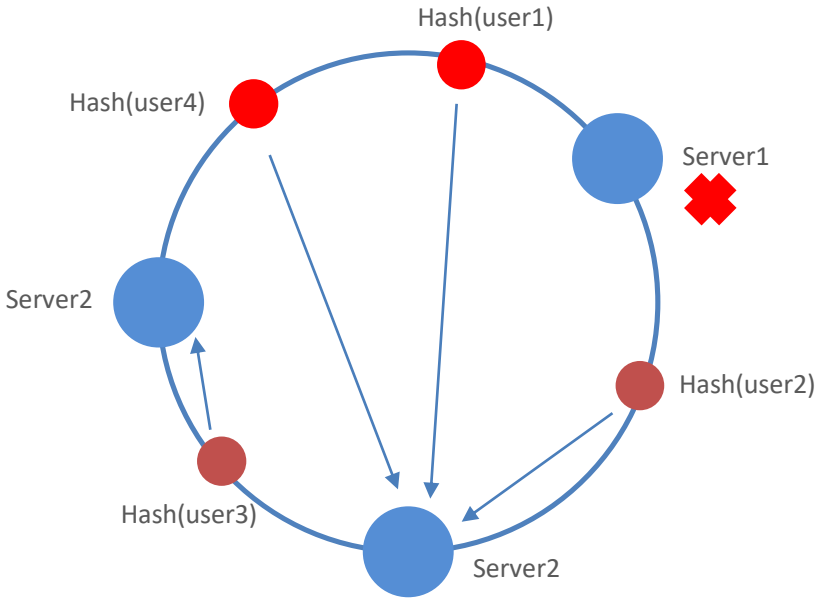
形成一个 hash 圆环，服务器分散在环上，为每个请求选择顺时针方向上最近的那个服务器。



补充：hash 与 一致性hash

一致性 hash 方式 – 伸缩性好

服务器增减后只影响一部分请求。



2. Dubbo 负载均衡配置

属性配置

```
dubbo:
  provider:
    loadbalance: roundrobin
```

可选择：

1. random
2. roundrobin
3. leastactive
4. consistenthash

负载均衡-总结

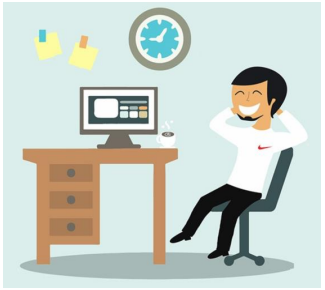
重难点

1. Dubbo 支持的负载均衡策略
2. Dubbo 配置负载均衡策略的方法

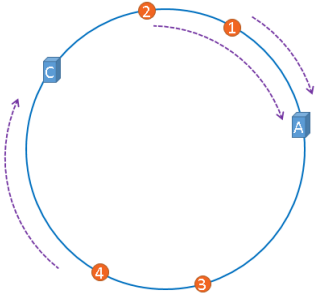
负载均衡策略

随机

轮询



最少活跃调用数



一致性hash

负载均衡-总结

重难点

1. Dubbo 支持的负载均衡策略
2. Dubbo 配置负载均衡策略的方法

   （dubbo.provider.loadbalance）

负载均衡-总结

重难点

1. Dubbo 支持的负载均衡策略
2. Dubbo 配置负载均衡策略的方法

下节

Sentinel 是否**兼容** Dubbo?

目录 Contents

整合 Sentinel 系统防护

小节导学

Sentinel 对系统中的资源进行保护，例如：

- RestTemplate 调用是个资源
- Feign 的服务接口是个资源
- Gateway 的 Route 是个资源

Sentinel 对其进行了适配，都可以进行防护。Dubbo 的 Provider 理论上也是资源，Sentinel 如何保护呢？

Sentinel 开发了 sentinel-dubbo-adapter，用于适配 Dubbo 的资源，本节我们就实践 Dubbo 如何与 Sentinel 进行整合。

- Dubbo 整合 Sentinel
- 自定义异常处理

1. Dubbo 整合 Sentinel

步骤

1. 添加 Sentinel 相关依赖

```
<groupId>com.alibaba.cloud</groupId>
<artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>

<groupId>com.alibaba.csp</groupId>
<artifactId>sentinel-apache-dubbo-adapter</artifactId>

<groupId>com.alibaba.csp</groupId>
<artifactId>sentinel-transport-simple-http</artifactId>
```

1. Dubbo 整合 Sentinel

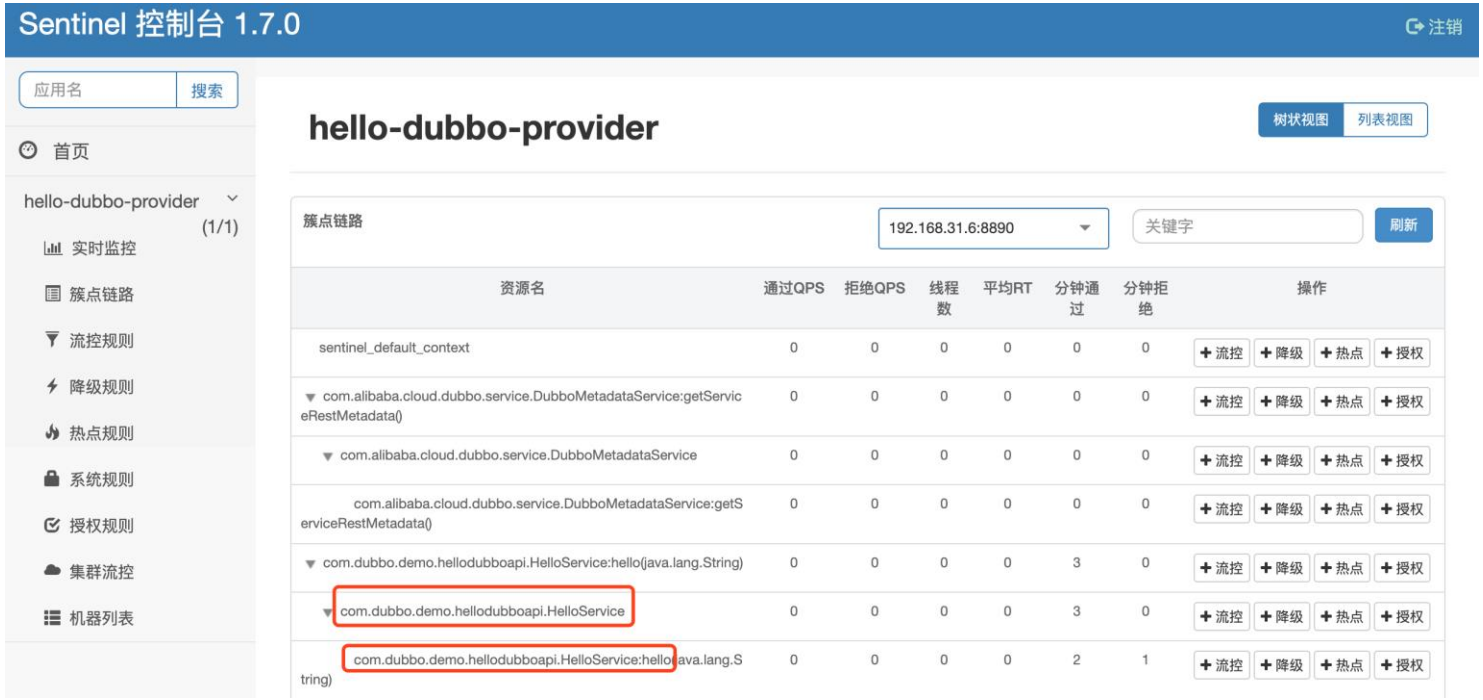
步骤

2. 属性配置

```
spring:
  application:
    name: hello-dubbo-consumer
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
    sentinel:
      transport:
        dashboard: localhost:8080
        port: 8890
```

1. Dubbo 整合 Sentinel

测试



2. 自定义异常处理

步骤

1. 自定义 DubboFallback

```
public class MyDubboFallback implements DubboFallback {
    public MyDubboFallback() {
    }
    @Override
    public Result handle(Invoker<?> invoker, Invocation invocation, BlockException e) {
        Result result = invoker.invoke(invocation);
        result.setValue("limit");
        return result;
    }
}
```

2. 自定义异常处理

步骤

2. 注册 DubboFallback

```
@EnableDiscoveryClient
@SpringBootApplication
public class HellodubboproviderApplication {
            public static void main(String[] args) {
                        SpringApplication.run(HellodubboproviderApplication.class, args);
            }
            @PostConstruct
            public void setFallback(){
                        DubboFallbackRegistry.setProviderFallback(new MyDubboFallback());
            }
}
```

整合 Sentinel 系统防护-总结

重难点

1. Dubbo 整合 Sentinel 的流程
2. 自定义 Sentinel Fallback 的方法

Dubbo 整合 Sentinel

1. 添加 Sentinel 相关依赖

```
<groupId>com.alibaba.cloud</groupId>
<artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>

<groupId>com.alibaba.csp</groupId>
<artifactId>sentinel-apache-dubbo-adapter</artifactId>
```

2. 属性配置

```
spring:
  cloud:
    sentinel:
      transport:
        dashboard: localhost:8080
        port: 8890
```

整合 Sentinel 系统防护-总结

重难点

1. Dubbo 整合 Sentinel 的流程
2. 自定义 Sentinel Fallback 的方法

自定义 Sentinel Fallback 的方法

1. 自定义 DubboFallback

```
public class MyDubboFallback implements DubboFallback {
    @Override
    public Result handle(Invoker<?> invoker, Invocation invocation, BlockException e) {
        ……
    }
}
```

2. 注册

```
DubboFallbackRegistry.setProviderFallback(new MyDubboFallback());
```

整合 Sentinel 系统防护-总结

重难点

1. Dubbo 整合 Sentinel 的流程
2. 自定义 Sentinel Fallback 的方法

下节

Dubbo 是否可以**顺利整合** SkyWalking?

目录 Contents

整合 SkyWalking 链路跟踪

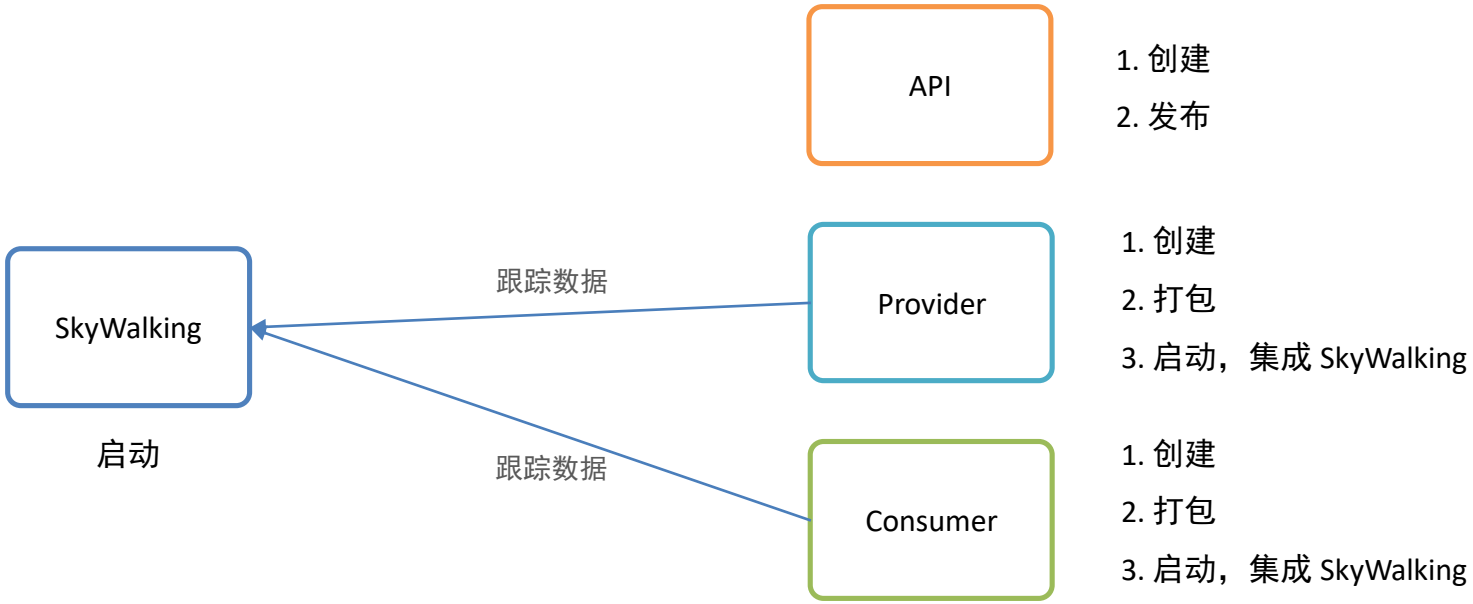
小节导学

调用链跟踪对于 HTTP 的方式肯定没问题，但换成 Dubbo RPC 的调用方式之后，还能正常跟踪吗?

本节我们实践 SkyWalking 对 Dubbo 服务调用跟踪，验证是否正常工作。

整合 SkyWalking 链路跟踪

验证流程



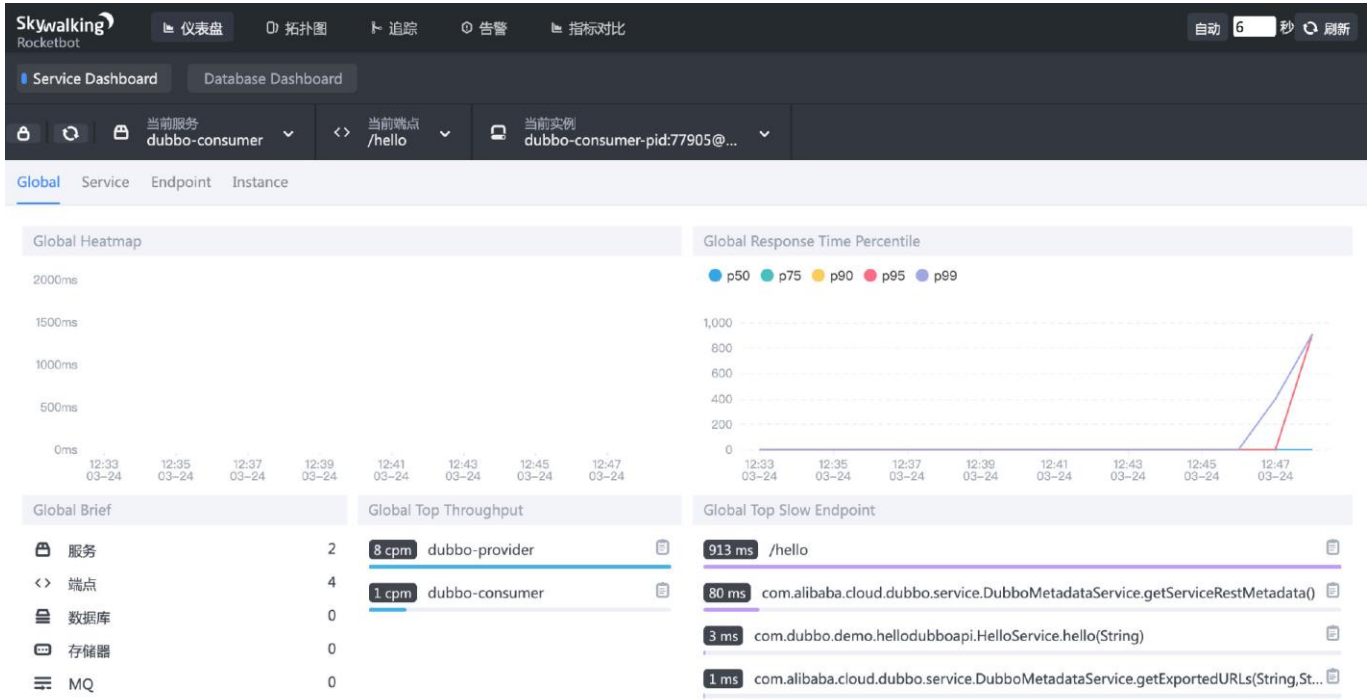
整合 SkyWalking 链路跟踪

整合 SkyWalking Agent

```
java \
  -javaagent:apache-skywalking-apm-bin-6.6/agent/skywalking-agent.jar \
  -Dskywalking.agent.service_name=dubbo-provider \
  -Dskywalking.collector.servers=localhost:11800 \
  -jar hello-dubbo-provider-0.0.1-SNAPSHOT.jar

java \
  -javaagent:apache-skywalking-apm-bin-6.6/agent/skywalking-agent.jar \
  -Dskywalking.agent.service_name=dubbo-consumer \
  -Dskywalking.collector.servers=localhost:11800 \
  -jar hello-dubbo-consumer-0.0.1-SNAPSHOT.jar
```

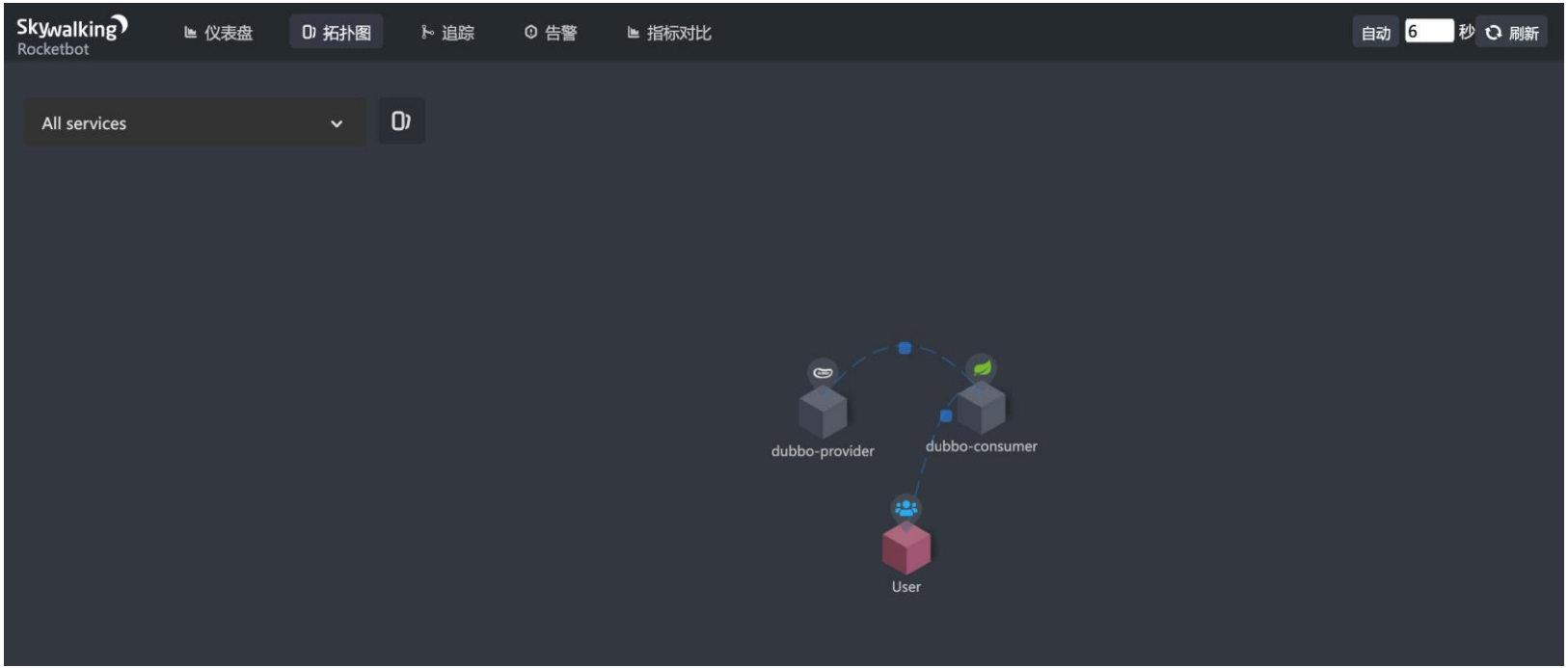
整合 SkyWalking 链路跟踪

效果



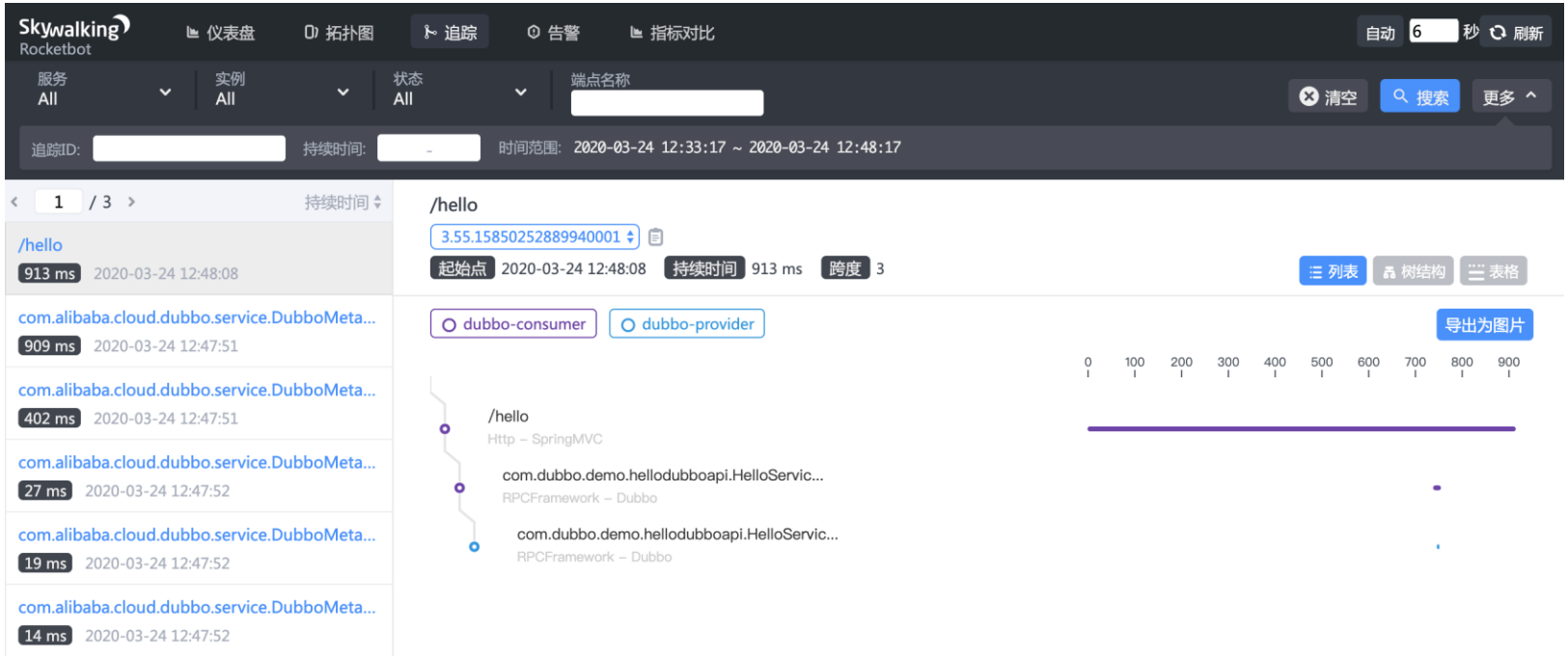
整合 SkyWalking 链路跟踪

效果



整合 SkyWalking 链路跟踪

效果



整合 SkyWalking 链路跟踪

结论

OK

整合 SkyWalking 链路跟踪-总结

重难点

1. Dubbo 整合 SkyWalking 的流程

整合 SkyWalking 链路跟踪-总结

重难点

1. Dubbo 整合 SkyWalking 的流程

下节

Mock，这是做什么的？怎么用？

目录 Contents

Mock 本地伪装

小节导学

在调用 Provider 的时候，难免会出现异常。

这时如果 Consumer 希望提供一个默认结果数据，如何处理?

Dubbo 提供了 Mock 伪装 的方法。

本节我们就分析一下 Mock 的作用与用法。

Mock 本地伪装

Mock 伪装实现思路

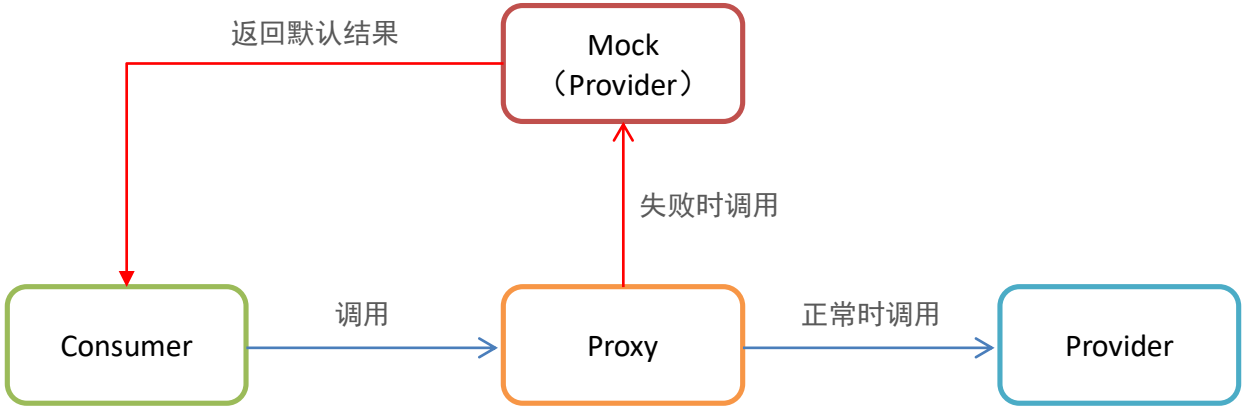
Consumer 调用 Provider，正常情况没问题，但如果 Provider 故障了，Consumer就拿不到结果数据了。

Consumer 希望在 Provider 异常的时候使用自己的默认数据。

Mock 本地伪装

Mock 伪装实现思路

可以通过一个代理 Proxy + Mock 实现。



Mock 本地伪装

Mock 伪装实现思路

Mock 本地伪装

实现步骤

远程接口

```
package com.dubbo.demo.hellodubboapi;

public interface HelloService {
    String hello(String name);
}
```

1. Consumer 本地创建 Mock 伪装

```
package com.dubbo.demo.hellodubboapi;
public class HelloServiceMock implements
HelloService {
    public String hello(String name){
        return "mock hello " + name;
    }
}
```

2. 声明使用 Mock

```
@Reference(mock = "true")
HelloService helloService;
```

Mock 本地伪装

Mock 命名规则

1. 本地伪装的包名需要和服务接口的包名一致
2. 类名是在服务接口名后加上“Mock”后缀。

例如：

```
org.apache.dubbo.samples.stub.api.DemoService
org.apache.dubbo.samples.stub.api.DemoServiceMock
```

如果不希望使用默认的命名规则，也可以通过“mock”属性来指定全类名

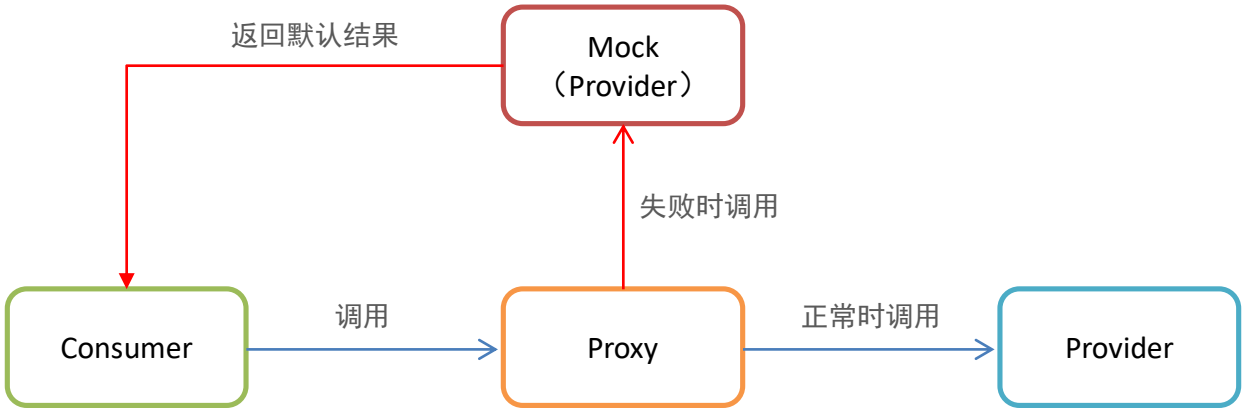
Mock 本地伪装-总结

重难点

1. Dubbo Mock 的作用与实现思路
2. Mock 的开发流程

Mock 作用与实现思路

Mock 用于真实的 Provider 异常后提供默认结果。



Mock 本地伪装-总结

重难点

1. Dubbo Mock 的作用
2. Mock 的开发流程

Mock 开发流程

远程接口

```
package com.dubbo.demo.hellodubboapi;

public interface HelloService {
    String hello(String name);
}
```

1. Consumer 本地创建 Mock 伪装

```
package com.dubbo.demo.hellodubboapi;
public class HelloServiceMock implements
HelloService {
    public String hello(String name){
        return "mock hello " + name;
    }
}
```

2. 声明使用 Mock

```
@Reference(mock = "true")
HelloService helloService;
```

Mock 本地伪装-总结

重难点

1. Dubbo Mock 的作用
2. Mock 的开发流程

下节

Stub，这是做什么的？怎么用？

目录 Contents

- 整合 Nacos 服务发现
- 整合 Nacos 外部配置
- 高性能序列化
- 负载均衡
- 整合 Sentinel 系统防护
- 整合 SkyWalking 链路跟踪
- Mock 本地伪装
- Stub 本地存根

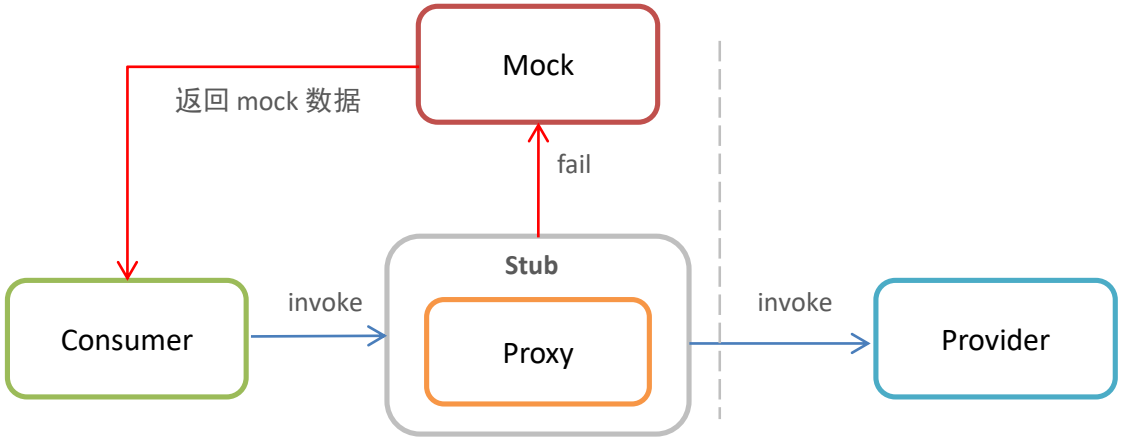
Stub 本地存根

小节导学

在调用 Provider 的时候，调用之前有可能希望做一些处理，例如验证一下参数的正确性。

如果调用之后也可能希望做一些处理，例如缓存返回结果。

Dubbo 提供了 Stub 存根 的方法。



Stub 本地存根

实现步骤

远程接口

```
package com.dubbo.demo.hellodubboapi;

public interface HelloService {
    String hello(String name);
}
```

1. Consumer 本地创建 Stub 存根

```
package com.dubbo.demo.hellodubboapi;
public class HelloServiceStub
            implements HelloService {
    ……
}
```

2. 声明使用 Stub

```
@Reference(stub = "true")
HelloService helloService;
```

Stub 本地存根 – 实现分析


```
package com.example.demo.stub.api;

public class HelloServiceStub implements HelloService {



    @Override
    public String hello(String name) {
        String result = "";









        return result;
    }
}
```

实现接口
复写接口中的方法

命名规则：
1. 包名与服务接口一致
2. 类名 = 接口名 + Stub

Stub 本地存根 – 实现分析


```
package com.example.demo.stub.api;

public class HelloServiceStub implements HelloService {

    private HelloService helloService;
    public HelloServiceStub(HelloService helloService) {
        this.helloService = helloService;
    }

    @Override
    public String hello(String name) {
        String result = "";









        return result;
    }
}
```

创建构造方法
注入实际的服务接口

Stub 本地存根 – 实现分析


```
package com.example.demo.stub.api;

public class HelloServiceStub implements HelloService {

    private HelloService helloService;
    public HelloServiceStub(HelloService helloService) {
        this.helloService = helloService;
    }

    @Override
    public String hello(String name) {
        String result = "";

        System.out.println("调用 hello() 之前的处理逻辑 ...");









        return result;
    }
}
```

发起调用之前的逻辑

Stub 本地存根 – 实现分析


```
package com.example.demo.stub.api;

public class HelloServiceStub implements HelloService {

    private HelloService helloService;
    public HelloServiceStub(HelloService helloService) {
        this.helloService = helloService;
    }

    @Override
    public String hello(String name) {
        String result = "";

        System.out.println("调用 hello() 之前的处理逻辑 ...");

        result = helloService.hello(name);
        System.out.println("调用 hello() 的结果: " + result);



        return result;
    }
}
```

发起调用

Stub 本地存根 – 实现分析


```
package com.example.demo.stub.api;

public class HelloServiceStub implements HelloService {

    private HelloService helloService;
    public HelloServiceStub(HelloService helloService) {
        this.helloService = helloService;
    }

    @Override
    public String hello(String name) {
        String result = "";

        System.out.println("调用 hello() 之前的处理逻辑 ...");

        result = helloService.hello(name);
        System.out.println("调用 hello() 的结果: " + result);

        System.out.println("调用 hello() 之前的处理逻辑 ...");

        return result;
    }
}
```

调用之后的逻辑

Stub 本地存根 – 实现分析

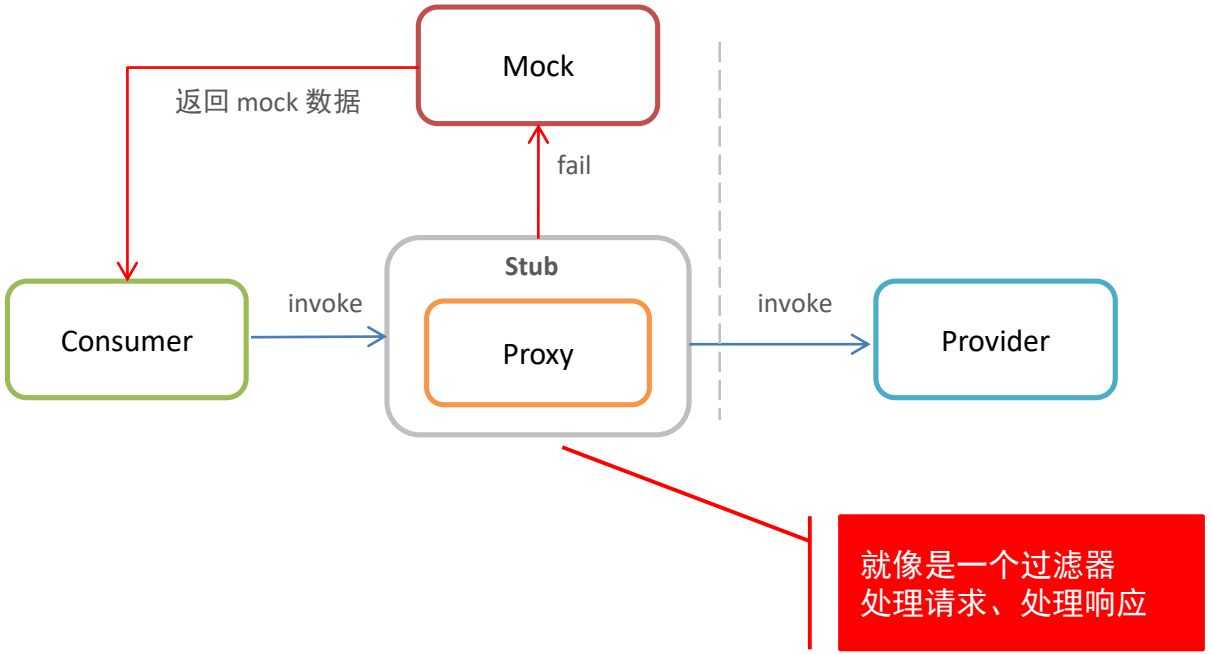

```
package com.example.demo.stub.api;

public class HelloServiceStub implements HelloService {

    private HelloService helloService;
    public HelloServiceStub(HelloService helloService) {
        this.helloService = helloService;
    }

    @Override
    public String hello(String name) {
        String result = "";

        System.out.println("调用 hello() 之前的处理逻辑 ...");

        result = helloService.hello(name);
        System.out.println("调用 hello() 的结果: " + result);

        System.out.println("调用 hello() 之前的处理逻辑 ...");

        return result;
    }
}
```

Stub 本地存根-总结

重难点

1. Dubbo Stub 的作用
2. Stub 的开发流程

Stub 本地存根


Mock
返回 mock 数据
fail
Consumer
invoke
Stub
Proxy
invoke
Provider
就像是一个过滤器
处理请求、处理响应

Stub 本地存根-总结

重难点

1. Dubbo Stub 的作用
2. Stub 的开发流程

Stub 本地存根

实现步骤

远程接口

```
package com.dubbo.demo.hellodubboapi;

public interface HelloService {
    String hello(String name);
}
```

1. Consumer 本地创建 Stub 存根

```
package com.dubbo.demo.hellodubboapi;
public class HelloServiceStub
            implements HelloService {
    ……
}
```

2. 声明使用 Stub

```
@Reference(stub = "true")
HelloService helloService;
```

Stub 本地存根 – 实现分析


```
package com.example.demo.stub.api;

public class HelloServiceStub implements HelloService {

    private HelloService helloService;
    public HelloServiceStub(HelloService helloService) {
        this.helloService = helloService;
    }

    @Override
    public String hello(String name) {
        String result = "";

        System.out.println("调用 hello() 之前的处理逻辑 ...");

        result = helloService.hello(name);
        System.out.println("调用 hello() 的结果: " + result);

        System.out.println("调用 hello() 之前的处理逻辑 ...");

        return result;
    }
}
```

Stub 本地存根-总结

重难点

1. Dubbo Stub 的作用
2. Stub 的开发流程

下节

本章内容**总结**

一样的在线教育，不一样的教学品质