

[Dubbo](#)

RPC

RPC** RPC **

-
-
-
-

Dubbo □ Spring Boot □

RPC

RPC

RPC

RPC

-
-
-
-
-
-
-

RPC

RPC->RPC->TCP

-
-
-
-

Dubbo

-
-

gRPC

RPC

Google

RPC

-
-
-
-

Triple

RPC

| Dubbo3RPCDubbo3RPC

- HTTP/2
- Protobuf JSON
- 二进制
- 文本

二进制

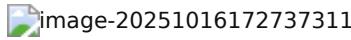
文本

- Provider
- Consumer
- Registry
- Monitor
- Container

容器

注册中心

- 服务端口
- Admin
- Dubbo**
 - DubboRPC
 - RPCDubboDubboRPCDubbo



二进制

Dubbo二进制

- 二进制
- RPC
- Dubbo2 TCP

文本

Dubbo文本

- 二进制
- 文本
- 二进制+文本
- 文本+二进制
- 二进制+文本+二进制

文本

Dubbo文本

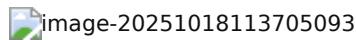
- 文本

- -
 -

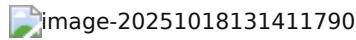
Dubbo & Sentinel

Router

- რეიტერი
 - Router რეიტერი
 - რეიტერი
 - რეიტერი



6



- 亂世の政治家
 - 亂世の政治家
 - 亂世の政治家
 - 亂世の政治家

1

- Provider အောက်ပါတဲ့
 - applicationContext.xml ဖြစ်တယ်

```
<dubbo:provider tag="gray"/>
```

or

```
@DubboService(tag="gray")
```

- Consumer 顾客
◦ 顾客标签 tag 顾客标签顾客标签顾客标签顾客标签

```
@DubboReference(tag = "gray")
```

1

- Provider 旣定条件
 - 通过 yaml 定义既定条件

```
configVersion: v3.0
force: true
enabled: true
key: shop-detail
tags:
  - name: gray
    match:
      - key: env
        value:
          exact: gray
```

- configVersion: v3.0 旣定条件
- force: true 旳 Consumer 旳既定条件 Provider
- enabled: true 旳
- key: shop-detail 旳
- tags 旳
 - name: gray 旳
 - match 旳 key=value 旳 tag=gray 旳
 - key 旳
 - value 旳

-
- Consumer 旳
 - 通过 tag 旳

既定条件

通过 tag 旳既定条件

- 通过 tag 旳既定条件
- 通过 ID 旳既定条件

既定条件/既定条件

```
# 既定条件
configVersion: v3.0
scope: service
force: true
runtime: true
enabled: true
key: org.apache.dubbo.samples.CommentService
conditions:
  - method=getUser & arguments[0]=1001 => tag=gray
```

- key 服务名
- scope 服务作用域 service □ application
- force 强制性
 - true 强制性
 - false 非强制性
- enabled 启用状态
- priority 优先级
- conditions 条件
 - 通过条件过滤器过滤 getuser 00000000000000001 tag=gray □ Provider

配置文件

通过配置文件来实现对服务的动态管理

配置文件语言 javascript 语言

```
configVersion: v3.0
key: demo-provider
type: javascript
enabled: true
script: |
    (function route(invokers, invocation, context) {
        var result = new java.util.ArrayList(invokers.size());
        for (i = 0; i < invokers.size(); i++) {
            if ("10.20.3.3".equals(invokers.get(i).getUrl().getHost())) {
                result.add(invokers.get(i));
            }
        }
        return result;
    })(invokers, invocation, context)); // 配置方法
```

配置

“配置”是通过配置文件来实现对服务的动态管理

通过配置文件来实现对服务的动态管理RPC语言

配置文件 = 配置 + 配置 + 配置

配置文件语言

- 配置文件语言
- 配置文件语言
- 配置文件语言

Dubbo□SPI

SPI是什么

通过SPI实现对服务的动态管理

Dubbo SPI 实现类 ExtensionLoader 通过 ExtensionLoader 实现对服务的动态管理

- 通过注解 @SPI 实现对服务的动态管理

- META-INF/dubbo 目录
- Dubbo SPI 相关的实现类

```
dog=com.sunnick.animal.impl.Dog
cat=com.sunnick.animal.impl.Cat
```

```
public void testDubboSPI(){
    System.out.println("=====dubbo SPI=====");
    ExtensionLoader<Animal> extensionLoader =
        ExtensionLoader.getExtensionLoader(Animal.class);
    Animal cat = extensionLoader.getExtension("cat");
    cat.run();
    Animal dog = extensionLoader.getExtension("dog");
    dog.run();
}
```

java SPI 相关

- SPI 相关的实现类
- SPI 相关的注解

Dubbo SPI 相关

- @SPI 相关的实现类 Dubbo 相关的实现类
- @Adaptive 相关的实现类
- Wrapper 相关 Dubbo 相关的实现类
 - 相关的实现类 SPI 相关

```
public class LogWrapper implements Log {
    private final Log log; // 相关的实现类

    public LogWrapper(Log log) { // AOP
        this.log = log;
    }

    @Override
    public void info(String msg) {
        System.out.println("[Before]"); // AOP
        log.info(msg);
        System.out.println("[After]"); // AOP
    }
}
```

二

三

Maven 配置

在 Maven 项目中添加 dubbo-spring-boot-starter 依赖。Dubbo 官方提供了 **dubbo-bom**。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.apache.dubbo</groupId>
            <artifactId>dubbo-bom</artifactId>
            <version>3.3.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

在 pom.xml 中添加 starter 依赖。

```
<dependencies>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-zookeeper-spring-boot-starter</artifactId>
    </dependency>
</dependencies>
```

application.yml 配置

```
dubbo:
  application:
    name: dubbo-springboot-demo-provider
    logger: slf4j
  protocol:
    name: tri
    port: -1
  registry:
    address: zookeeper://127.0.0.1:2181
```

- dubbo.application.name 定义服务提供者名称
- dubbo.protocol.name 定义 RPC 协议
- dubbo.protocol.port 定义端口
 - 本地-127.0.0.1 Dubbo 官方推荐
 - 外网 ip:port 定义远程端口
- dubbo.registry.address 定义注册中心 IP 地址

Dubbo

- `@DubboService` 定义Dubbo服务的注解
- `@DubboReference` 定义对Dubbo服务的引用
- `@EnableDubbo` 启动Dubbo的注解
◦ `EnableDubbo(scanBasePackages = {"org.apache.dubbo.springboot.demo.provider"})`

RPC

Triple

- 定义RPC应用 `application.yaml` 配置文件
 - `server: true`

```
dubbo:  
  application:  
    name: dubbo-springboot-demo-provider  
    logger: slf4j  
  protocol:  
    name: tri  
    port: -1  
  registry:  
    address: zookeeper://127.0.0.1:2181
```

- 定义IDL接口
◦ `message`
◦ `service`

Protobuf

1. pom.xml

```
<plugin>  
  <groupId>org.apache.dubbo</groupId>  
  <artifactId>dubbo-maven-plugin</artifactId>  
  <version>${dubbo.version}</version> <!-- 3.3.0 -->  
  <configuration>  
    <outputDir>build/generated/source/proto/main/java</outputDir> <!-- 生成目录 -->  
  -->  
  </configuration>  
</plugin>
```

configuration 定义配置项

- `outputDir` 生成Java文件目录
- `protoSourceDir` protobuf文件目录

2. 定义IDL文件 .proto

```
syntax = "proto3";  
option java_multiple_files = true;
```

```

package org.apache.dubbo.samples.tri.unary;

message GreeterRequest {
    string name = 1;
}

message GreeterReply {
    string message = 1;
}

service Greeter{
    rpc greet(GreeterRequest) returns (GreeterReply);
}

```

- syntax protobuf
- package org.apache.dubbo.samples.tri.unary
- option java_outer_classname=Greeter
- message GreeterRequest
 - name
 - message
 - GreeterRequest
- service Greeter
 - rpc greet
 - GreeterReply
 - returns() GreeterReply

三、Dubbo 客户端

1. 配置文件

- Nacos 配置 dubbo-nacos-spring-boot-starter
- Zookeeper 配置 dubbo-zookeeper-spring-boot-starter
- Kubernetes Service

2. 配置

- application.yml 配置

```

dubbo
  registry
    address: nacos://localhost:8848

```

- @DubboService 和 @DubboReference 注解

```

dubbo:
  provider:
    delay: 5000

```

四、Dubbo 服务端

- Dubbo 容器配置
- Dubbo 服务实现

```
dubbo:  
  registry:  
    subscribe: false
```

•

- Dubbo 2.7.0 之后，取消了对 dubbo:registry:subscribe 的支持
- 取而代之的是 dubbo:registry:register

```
dubbo:  
  registry:  
    register: false
```

•

• Dubbo 2.7.0 之后，取消了对 dubbo:registry:subscribe 的支持

•

1. 通过 token

```
dubbo:  
  provider:  
    token: true      # 生成 UUID
```

2. 通过配置 token

3. 通过配置

- 通过配置 token
- Dubbo 2.7.0 token 为 RPC 属性 metadata
- 通过配置 Dubbo token

•

•

1. 在 Dubbo Admin 中
2. 配置中心 > 配置
3. 配置中心 key 为 timeout，值为 2000

```
configVersion: v3.0  
enabled: true  
configs:  
  - side: provider  
    parameters:  
      timeout: 2000  
      retries: 5
```

- configVersion 为配置版本
- enabled 为是否开启

- side 服务提供者 Provider 和 服务消费者 Consumer
- parameters.timeout 超时时间
- retries 重试次数
- accesslog 访问日志

配置

配置文件示例

```
configVersion: v3.0
key: org.apache.dubbo.samples.DetailService
scope: service
force: false
enabled: true
priority: 1
conditions:
  - method=item & arguments[1]=dubbo => detailVersion=v2
```

- key 服务键
- force 强制更新
- enabled 启用状态
- priority 优先级
- conditions 条件
 - 逻辑

```
method=<> & arguments[<>]=<> => <>=<>
```

配置

配置文件示例

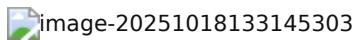
```
configVersion: v3.0
scope: service
key: org.apache.dubbo.samples.OrderService
configs:
  - side: provider
    match:
      param:
        - key: orderVersion
          value:
            exact: v2
    parameters:
      weight: 25
```

- scope 服务范围
- configs 配置项
- side 服务提供者

- `match` 时候
 - `key` 时候
 - `value.exact` 时候
- `parameters.weight` 时候 0-100

RPC

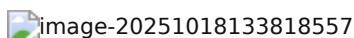
概念



1. 客户端调用 `Future`
2. `future.get()` 时候 `ThreadlessExecutor.wait()` 或 `wait` 指定的线程池线程执行
3. `Runnable Task` ** `Runnable Task` ** `ThreadlessExecutor` 执行
4. `CompletableFuture` RPC 时候 `future.set()`
5. 客户端返回

实现

`Triple`



Dubbo 通过 IO 实现 RPC

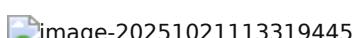
概念

Dubbo 是一个 RPC 服务框架，提供自动发现、负载均衡、容错等能力。

分为 Provider 和 Consumer 两个主要角色。

- `Consumer` 调用 `RPC` 服务时，会向服务提供者发起请求。
 - 使用 `IO` 方式（阻塞、非阻塞）与服务提供者进行通信。

实现



- `Provider` 提供 **Dubbo** 服务的 `Dubbo` 服务提供者。

Provider

- `CompletableFuture`
- `CompletableFuture` `CompletableFuture`
 - `Dubbo` 服务提供者通过 `Dubbo` 服务提供者向 `Dubbo` 服务提供者发送响应。
 - `CompletableFuture` `null`

实现

```
public interface AsyncService {
    /**
     * ...
}
```

```

    */
    String invoke(String param);
    /**
     * 业务逻辑
     */
    CompletableFuture<String> asyncInvoke(String param);
}

```

④④④

```

@DubboService
public class AsyncServiceImpl implements AsyncService {

    @Override
    public CompletableFuture<String> asyncInvoke(String param) {
        // ④④④supplyAsync④④④④④
        return CompletableFuture.supplyAsync(() -> {
            try {
                // Do something
                long time = ThreadLocalRandom.current().nextLong(1000);
                Thread.sleep(time);
                StringBuilder s = new StringBuilder();
                s.append("AsyncService asyncInvoke")
                    .append(param).append(",sleep:")
                    .append(time);
                return s.toString();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return null;
        });
    }
}

```

Consumer④④

- ④④④④④ CompletableFuture ④④④④④④④④
 - ④④ whenComplete ④④④④④

```

@Override
public void run(String... args) throws Exception {
    //④④④④④
    CompletableFuture<String> future1 = asyncService.asyncInvoke("async call
request1");
    future1.whenComplete((v, t) -> {
        if (t != null) {
            t.printStackTrace();
        } else {
            System.out.println("AsyncTask Response-1: " + v);
        }
    });
}

```

});

- `async`

```
@DubboReference(async="true")
private AsyncService asyncService;
```

-

- `getBooleanValue`
 - sent `true`
 - sent `false`

```
@DubboReference(methods = {@Method(name = "sayHello", timeout = 5000) sent = true})  
private AsyncService asyncService;
```

- `Future<T>`

▪ `return false` `Future<T>` `future` `Future<T>`

```
@DubboReference(methods = {@Method(name = "sayHello", timeout = 5000) return  
= false)})  
private AsyncService asyncService
```

1

A decorative horizontal bar consisting of a series of small, evenly spaced rectangles.

GenericService

- 
 - 

□□□□□□□□□□

- 
 - 
 - 

Spring XML

1. 服务提供者
 2. 服务消费者 dubbo:reference 方式 generic=true 方式
 3. 通过 Bean 方式 Bean 方式 GenericService 方式
 4. 调用 \$invoke 方法实现
 - 重写
 - 代理实现
 - AOP 方式

```

private static GenericService genericService;

public static void main(String[] args) throws Exception {
    ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("spring/generic-impl-consumer.xml");
    context.start();
    //从bean.xml中id
    genericService = context.getBean("helloService");
    //调用
    Object result = genericService.$invoke("sayHello", new String[]
{"java.lang.String"}, new Object[]{"world"});
}

```

启动消费者端

```

@DubboReference(
    interfaceName = "com.example.UserService", // 服务提供者
    generic = true // 通用
)
private GenericService genericService;

```

Filter

Dubbo Filter 通过SPI实现RPC过滤器

```

@SPI(scope = ExtensionScope.MODULE)
public interface Filter extends BaseFilter {}

```

实现

- 实现Filter接口 Filter 实现

```

@Activate(group = {Constants.PROVIDER, Constants.CONSUMER}) // 指定作用域
public class MyFilter implements Filter {

    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws
RpcException {
        // 前置
        System.out.println("Before RPC: " + invocation.getMethodName());

        // 调用 Filter 的 RPC
        Result result = invoker.invoke(invocation);

        // 后置
        System.out.println("After RPC: " + invocation.getMethodName());

        return result;
    }
}

```

```
        }
    }
```

- `@Reference`
- `@Activate` `bean`
 - `group` `Provider/Consumer`
 - `value` `URL` `URL` `Provider/Consumer`
 - `order` `Provider/Consumer`

2. `DubboSPI` `Provider/Consumer`
 - `META-INF/dubbo/org.apache.dubbo.rpc.Filter` `bean`

Provider

Provider/Consumer

- `Provider`

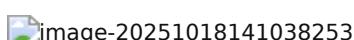
```
dubbo:  
  provider:  
    timeout: 5000
```

- `Provider`

```
@DubboReference(methods = {@Method(name = "sayHello", timeout = 5000)})  
private DemoService demoService;
```

Provider/Consumer `Provider` > `Consumer` > `Provider` > `Consumer`

Deadline



- A->B `5s` B->C `5s` C->D `5s`
- Deadline `2s` B->C->D `deadline 5s` C `deadline 3s` D `deadline 2s`

Provider `deadline`

```
dubbo:  
  provider:  
    timeout: 5000  
    parameters.enable-timeout-countdown: true
```

Provider `deadline`

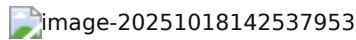
```
@DubboReference(timeout=5000, parameters={"enable-timeout-countdown", "true"})  
private DemoService demoService;
```

Consumer

Dubbo failover 介绍

概念

- **Cluster**
- **Directory**
- **Router**
- **LoadBalance**
- **Invoker**



实现

- Invoker 从 Provider 提供的 Service 列表中调用 Invoker 从 Provider 提供的 Service 列表中
- Directory 从 Invoker
- Cluster 从 Directory 调用 Invoker 从 Invoker 提供的服务列表中随机选取一个服务
- Router 从 Invoker 服务列表中随机选取一个服务
- LoadBalance 从 Invoker 服务列表中随机选取一个服务

集群实现

- Failover Cluster 提供高可用性服务
- failfast Cluster 提供高可用性服务
- Failsafe Cluster 提供高可用性服务
- Failback Cluster 提供高可用性服务
- Forking Cluster 提供高可用性服务
- Broadcast Cluster 提供高可用性服务