

Dubbo

xbZhong

2025-10-18

[本页PDF](#)

[Dubbo中文文档](#)

介绍

是一款基于 **RPC** 的服务开发框架，可以帮助解决如下微服务实践问题：

- 微服务**编程范式**和**工具**
- 高性能的**RPC通信**
- 微服务**监控与治理**

Dubbo 对 Spring Boot 微服务框架都做了很好的适配

RPC

一种**通信模型**或**设计思想**

RPC 自身**不规定具体实现方式**，只是思想

核心：远程调用本地化

- 开发者无需显式处理**网络通信**的细节
- 适合分布式系统服务间的**内部通信**
- 像调用本地方法一样，调用**远程服务**
- 使用**TCP**进行数据传输
- 基于**Netty**使用**TCP长连接**进行连接复用

RPC的请求流程

桩文件->序列化->TCP

- 定义**IDL文件（接口描述文件）**，编译工具根据 **pb** 文件生成 **stub** 桩文件，调用者根据 **stub** 文件调用服务
 - 解决**函数映射**的问题
- 网络里传输的数据是**二进制传输**，需要对请求参数，返回结果进行encode和decode
- 根据RPC协议约定数据头、元数据、消息体等，保证有ID能使**请求和返回结果做到一一映射**

Dubbo支持的通信协议和序列化协议

- **通信协议**：**Triple**、**dubbo**、**http**、**webservice** 等
- **序列化协议**：**hessian2**（dubbo默认协议）、**json**、**protobuf** 等

gRPC

是RPC通信模型的**具体实现**

Google开发的高性能RPC框架

- 基于HTTP/2，二进制序列化，具有**多路复用**，**头部压缩（HPACK）**，**二进制传输**等优点
- 二进制序列化协议采用 **Protobuf**
- 支持多种语言
- **云原生友好**，可与 **Kubernetes、Istio、Envoy** 等集成

Triple

是RPC通信模型的**具体实现**

阿里在Dubbo3推出的新一代RPC协议，是Dubbo3的默认RPC协议之一

- 协议层基于HTTP/2
- 支持 **Protobuf**、**JSON** 等**序列化协议**
- 支持多种语言
- **云原生友好**

架构

传统架构

- **Provider**：服务提供者
- **Consumer**：服务消费者
- **Registry**：注册中心，服务注册与发现
- **Monitor**：监控中心，统计调用次数，耗时等
- **Container**：服务运行容器

云原生架构

架构分为两层：

- **服务治理抽象控制面**
 - 包含注册中心、流量管控策略、Admin控制台等
- **Dubbo数据面**
 - 代表集群部署的所有Dubbo进程，进程间通过**RPC协议进行数据交换**
 - 有**服务消费者**（发起业务调用或RPC通信的Dubbo进程）和**服务提供者**（接收业务调用或RPC通信的Dubbo进程）

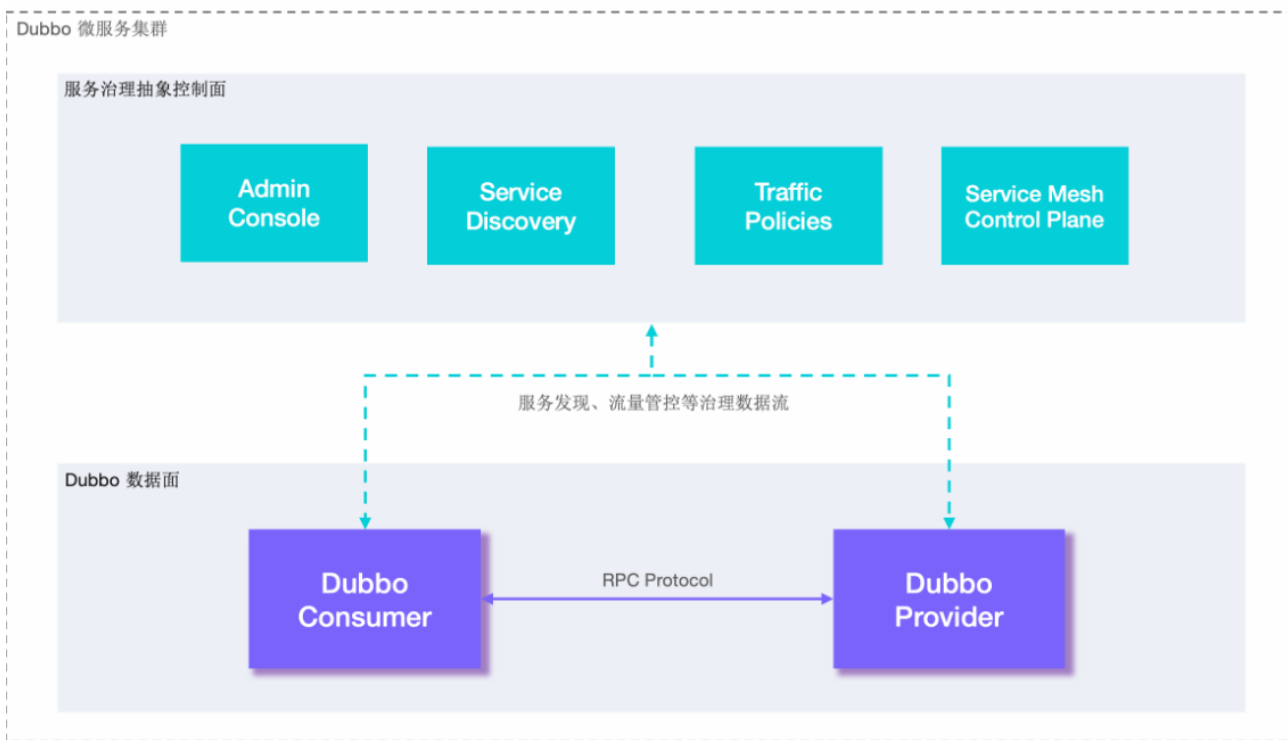


image-20251016172737311

通信协议

Dubbo支持几乎所有主流的**通信协议**

- 支持多协议暴露，可以在**单个端口暴露多个协议**
- 可以将一个RBC服务发布在**不同的端口**
- 内置的 Dubbo2 协议是在 TCP 传输层协议之上设计的**二进制通信协议**

负载均衡

内置多种**负载均衡算法**：

- 加权随机
- 加权轮询
- 最少活跃优先+加权随机
- 最短响应优先+加权随机
- 一致性哈希，相同参数的请求总是发到同一提供者

流量管控

有以下几种**不同的路由规则**：

- 条件路由规则
- 标签路由规则
- 脚本路由规则
- 动态配置规则

Dubbo 还支持限流&熔断，并且支持集成第三方工具如 Sentinel 实现

Router

主要作用：服务实例过滤器，是实现服务治理的关键环节

- 消费者要调用一个服务时，会先到注册中心获取**该服务实例**
- Router 根据**请求上下文和路由规则**，把不合适服务实例剔除掉，只保留合适的那部分
- 一个服务的路由过程可以由**多个路由组成**，叫做**路由链**
- 留下来的实例交给负载均衡算法进行选取

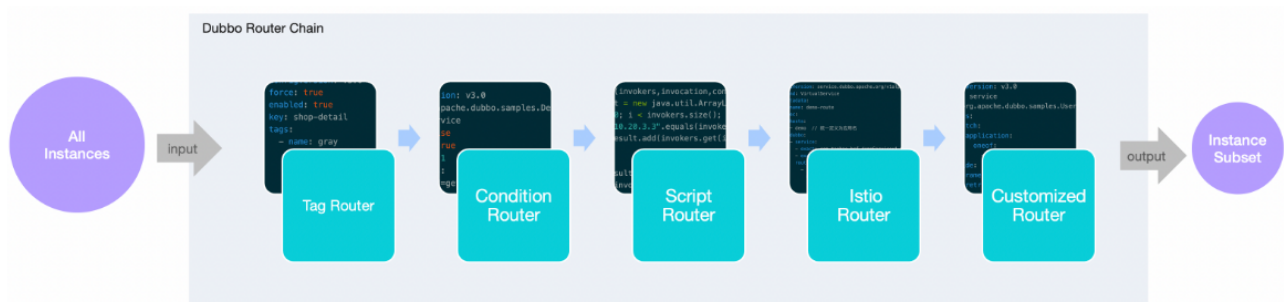


image-20251018113705093

标签路由规则

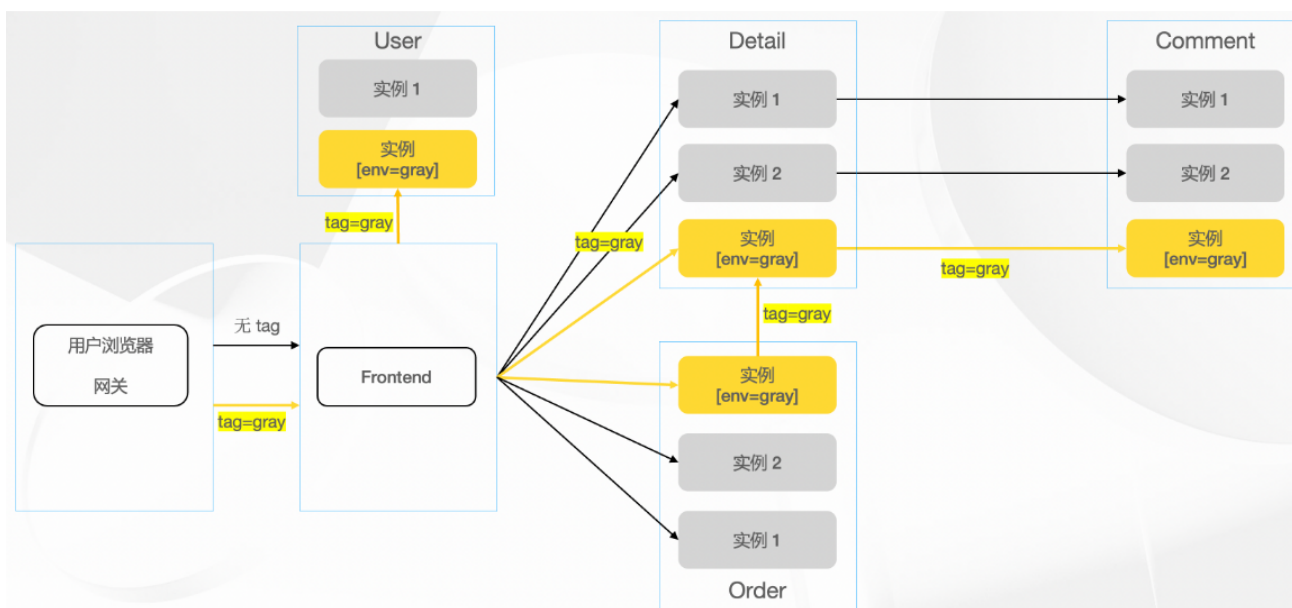


image-20251018131411790

非此即彼的流量隔离方案，匹配 标签 的请求会**100%**转发到有相同 标签 的实例，没有匹配 标签 的请求会**100%**转发到其余未匹配的实例

- 主要用于**灰度发布、版本隔离**
- 标签是主要针对**服务实例**的分组，分为**静态规则打标**和**动态规则打标**
- **静态规则打标**：直接在**服务配置文件**或者**注册中心**注册时指定标签，修改后需要**重启实例**才能生效

- **动态规则打标**：运行时通过规则**动态生成标签**，根据指定的匹配条件将服务实例**动态划分**到不同的流量分组中，是**热更新**

静态打标

- **Provider**：服务提供者
 - 在 `applicationContext.xml` 文件定义或者在**注解**上进行指定

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

or

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

- **Consumer**：服务调用者
 - 每次请求前通过 `tag` 设置**流量标签**，确保流量被调度到带有**同样标签**的服务提供方

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

动态打标

- **Provider**：服务提供者
 - 在服务提供者的 `yaml` 文件进行**动态规则**的配置

```
configVersion: v3.0
force: true
enabled: true
shop-detail

name: gray
match:
  - key: env
    value:
      exact: gray
```

- `configVersion: v3.0`：这是规则的版本号
- `force: true`：强制路由，意思是 **Consumer** 只能调用**匹配标签**的 **Provider**
- `enabled: true`：规则生效

- `key:shop-detail`：服务名
- `tags`：标签列表
 - `name:gray`：给服务实例打的标签名
 - `match`：匹配条件，表示服务调用者请求上下文里 `key=value` 的请求才会路由到 `tag=gray` 实例
 - `key`：键名
 - `value`：值名

- `Consumer`：服务调用者
 - 每次请求前通过 `tag` 设置流量标签

条件路由规则

根据请求上下文的任意属性动态决定路由，更加灵活

- 可实现流量控制、灰度发布等
- 属性包括用户ID、区域、版本、环境等

在服务端/注册中心进行配置

条件路由

```

gVersion: v3.0
: service
: true
ne: true
ed: true
org.apache.dubbo.samples.CommentService
tions:
method=getUser & arguments[0]=1001 => tag=gray
  
```

- `key`：服务名
- `scope`：规则作用范围，可以是 `service` 或 `application`
- `force`：是否强制路由
 - `true`：必须匹配到实例，否则失败
 - `false`：未匹配到实例会随机路由其他可用实例
- `enabled`：规则是否生效
- `priority`：规则优先级
- `conditions`：条件路由表达式列表
 - 例子中的意思是只匹配服务端接口方法名是 `getUser` 的调用，且第0个参数是1001，则路由到带 `tag=gray` 的 Provider

脚本路由规则

可以为某个微服务定义一条脚本规则，则后续所有请求都会执行一遍这个脚本，脚本过滤出来的地址即为请求允许发送到的、有效的地址集合

下面是个例子，可使用 `javascript` 进行脚本编写

```
gVersion: v3.0
demo-provider
javascript
ed: true
t: |
action 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)); // 表示立即执行方法
```

动态配置

是“动态化”的总称，包含标签路由、条件路由和服务参数动态修改

无需重启应用的情况下，实现**热部署**，**动态调整**RPC调用行为

动态配置路由 = 动态打标 + 动态条件路由 + 动态服务参数修改

有以下关键信息值得注意：

- 设置规则生效过滤条件
- 设置规则生效范围
- 选择规则管理粒度

Dubbo的SPI

SPI是什么

是一种**服务发现机制**，它可以用来实现**接口与实现解耦**，让系统在运行时可以动态加载某个接口的实现类，而不用硬编码接口的实现类

Dubbo的SPI的相关逻辑被封装在了 `ExtensionLoader` 类中，通过 `ExtensionLoader`，我们可以加载指定的实现类

- 在接口上加上 `@SPI` 注解，标记接口为可扩展点
- 所需的配置文件需放置在 `META-INF/dubbo` 路径下
- `Dubbo SPI` 是通过键值对的方式进行配置，这样就可以按需加载指定的实现类

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

```

1 void testDubboSPI(){
2     System.out.println("=====dubbo SPI=====");
3     ExtensionLoader<Animal> extensionLoader =
4         ExtensionLoader.getExtensionLoader(Animal.class);
5     Animal cat = extensionLoader.getExtension("cat");
6     cat.run();
7     Animal dog = extensionLoader.getExtension("dog");
8     dog.run();
9 }

```

java原生SPI有以下几个缺点：

- 需要遍历所有的实现并实例化，无法只加载某个指定的实现类，加载机制不够灵活
- 配置文件中没有给实现类命名，无法在程序中准确的引用它们

Dubbo的SPI解决了以上痛点，具有以下几个重要机制

- `@SPI` 注解：在接口上注解，标识接口是一个 Dubbo 扩展点，可以指定一个默认实现名
- `@Adaptive` 注解：在实现类上注解，用于生成一个自适应扩展类，会根据运行时参数自动加载所需的实现类
- Wrapper 机制：Dubbo 在创建某个扩展点的实例时，自动用包装类套一层，从而增强功能
 - 需要创建一个包装类和实现类，并对这两个类进行SPI的配置

```

1 public class LogWrapper implements Log {
2     private final Log log; // 注意：构造函数参数是接口类型
3
4     public LogWrapper(Log log) { // 这一点非常关键！！
5         this.log = log;
6     }
7
8     @Override
9     public void info(String msg) {
10        System.out.println("[Before]"); // AOP增强
11        log.info(msg);
12        System.out.println("[After]"); // AOP增强
13    }
14 }

```

使用

创建应用

Maven配置文件

在 Maven 文件通过引入 `dubbo-spring-boot-starter` 实现引入Dubbo核心依赖，自动扫描dubbo相关配置与注解

```
<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>
```

在相应模块的pom中增加必要的 `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` 配置文件

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

- `dubbo.application.name`：服务名称，唯一，根据此名称在注册中心进行注册
- `dubbo.protocol.name`：使用的RPC协议
- `dubbo.protocol.port`：服务暴露端口
 - 可设置为-1，那么Dubbo将会自动分配空闲窗口
 - 设置的 `ip:port` 将会在注册中心进行服务注册
- `dubbo.registry.address`：注册中心的IP地址

Dubbo注解

- `@DubboService`：实现Dubbo的服务暴露，且可以在这个注解上设置常见的服务参数
- `@DubboReference`：自动引入Dubbo服务实例，直接在代码中进行使用
- `@EnableDubbo`：必须配置，否则无法加载Dubbo注解定义的服务，一般加在启动类上
 - 默认只会扫描启动类所在的包，如果服务在其他包，需要增加配置，如 `EnableDubbo(scanBasePackages = {"org.apache.dubbo.springboot.demo.provider"})`

RPC协议配置

以Triple 协议为例

- 如上，已对RPC协议在 `application.yaml` 文件进行配置
 - 同一服务也可以提供多种协议访问方式

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

- 接着需要进行进行IDL文件进行服务定义

- 支持多语言，但学习成本较高
- 需要使用Dubbo提供的 `protoc` 编译插件将IDL文件编译成stub桩文件

使用Protobuf开发Triple通信服务

1. 引入pom依赖

```

<!-- 引入Dubbo Maven Plugin -->
<groupId>org.apache.dubbo</groupId>
<artifactId>dubbo-maven-plugin</artifactId>
<version>${dubbo.version}</version> <!-- 3.3.0及以上版本 -->
<configuration>
    <outputDir>build/generated/source/proto/main/java</outputDir> <!-- 参考下文可配置参数 -->
</configuration>
</plugin>

```

`configuration` 可配置参数，介绍常用的

- `outputDir`：生成的Java文件存放目录
- `protoSourceDir`：proto文件存放目录

2. 编写IDL文件，后缀为 `.proto`

```

syntax = "proto3";
package org.apache.dubbo.samples.tri.unary;

// GreeterRequest {
//     string name = 1;
// }

// GreeterReply {
//     string message = 1;
// }

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

```

- `syntax`：指定语法版本，必须写在文件第一行
- `package`：设置存放生成的Java代码的目录
- `option`：可选项，控制代码生成的细节
- `message`：定义数据结构，相当于Java的类，每个字段都有
 - 类型

- 名称
 - 唯一编号**：就是字段后面的数字，是二进制序列化的关键
- service：定义服务接口
 - rpc 后面是方法名
 - 括号里第一个是请求类型
 - returns() 中是返回类型

注册中心

常用注册中心

- Nacos：引入 dubbo-nacos-spring-boot-starter
- Zookeeper：引入 dubbo-zookeeper-spring-boot-starter
- Kubernetes Service

配置

- 在 application.yml 文件进行配置

```

registry:
  address: nacos://localhost:8848

```

- 全局延时注册（单位为毫秒），也支持部分服务延时注册（需要在 @DubboService 或者 @DubboReference 进行配置）

```

registry:
  provider:
    delay: 5000

```

只注册

- 使用场景：服务提供者只注册自己的服务，但不关心其它服务，不依赖其它服务
- 当前服务提供者**不订阅注册中心的服务**，只注册自己

```

registry:
  subscribe: false

```

只订阅

- 使用场景：服务提供者正在进行开发，暂时不想被消费者调用，且需要对其它服务进行订阅
- 当前服务提供者**不会注册到注册中心**

```
:
istry:
egister: false
```

权限控制

通过令牌验证在注册中心控制权限，以决定要不要下发令牌给消费者，可以防止消费者绕过注册中心访问提供者

步骤

1. 提供者配置 token

```
:
vider:
oken: true      # 自动生成 UUID
```

2. 注册中心记录token

3. 消费者订阅服务

- 消费者会自动从注册中心获取token
- Dubbo内部会把token放到RPC调用的 metadata 中
- 调用提供者时，Dubbo会验证token

流量管控

操作步骤

1. 打开Dubbo Admin控制台
2. 进入服务治理 > 动态配置
3. 点击创建，配置规则key、服务名和timeout值（是热更新），并进行保存

```
gVersion: v3.0
ed: true
gs:
ide: provider
arameters:
timeout: 2000
retries: 5
```

- configVersion：动态配置规则版本
- enabled：是否启用规则

- `side`：配置作用对象，可以为 **Provider**，也可以为 **Consumer**
- `parameters.timeout`：超时时间
- `retries`：服务重试次数
- `accesslog`：是否开启访问日志

参数路由

对条件、标签路由进行动态配置

```

detailVersion: v3.0
org.apache.dubbo.samples.DetailService
: service
: false
ad: true
ity: 1
tions:
method=getItem & arguments[1]=dubbo => detailVersion=v2

```

- `key`：规则标识
- `force`：强制路由，如果匹配不到目标实例，则随机访问其他可用实例
- `enabled`：是否生效
- `priority`：规则优先级
- `conditions`：条件路由规则
 - 条件语法

```

method=<方法名> & arguments[<索引>]=<值> => <标签>=<值>

```

权重分流

可以通过路由配置进行流量分流

```
gVersion: v3.0
: service
org.apache.dubbo.samples.OrderService
gs:
ide: provider
atch:
param:
  - key: orderVersion
    value:
      exact: v2
arameters:
weight: 25
```

- scope: 规则作用范围
- configs: 配置列表
- side: 规则作用对象
- match: 匹配条件
 - key: 实例标签名
 - value.exact: 标签值
- parameters.weight: 权重值 (0-100)

RPC框架

消费端线程模型

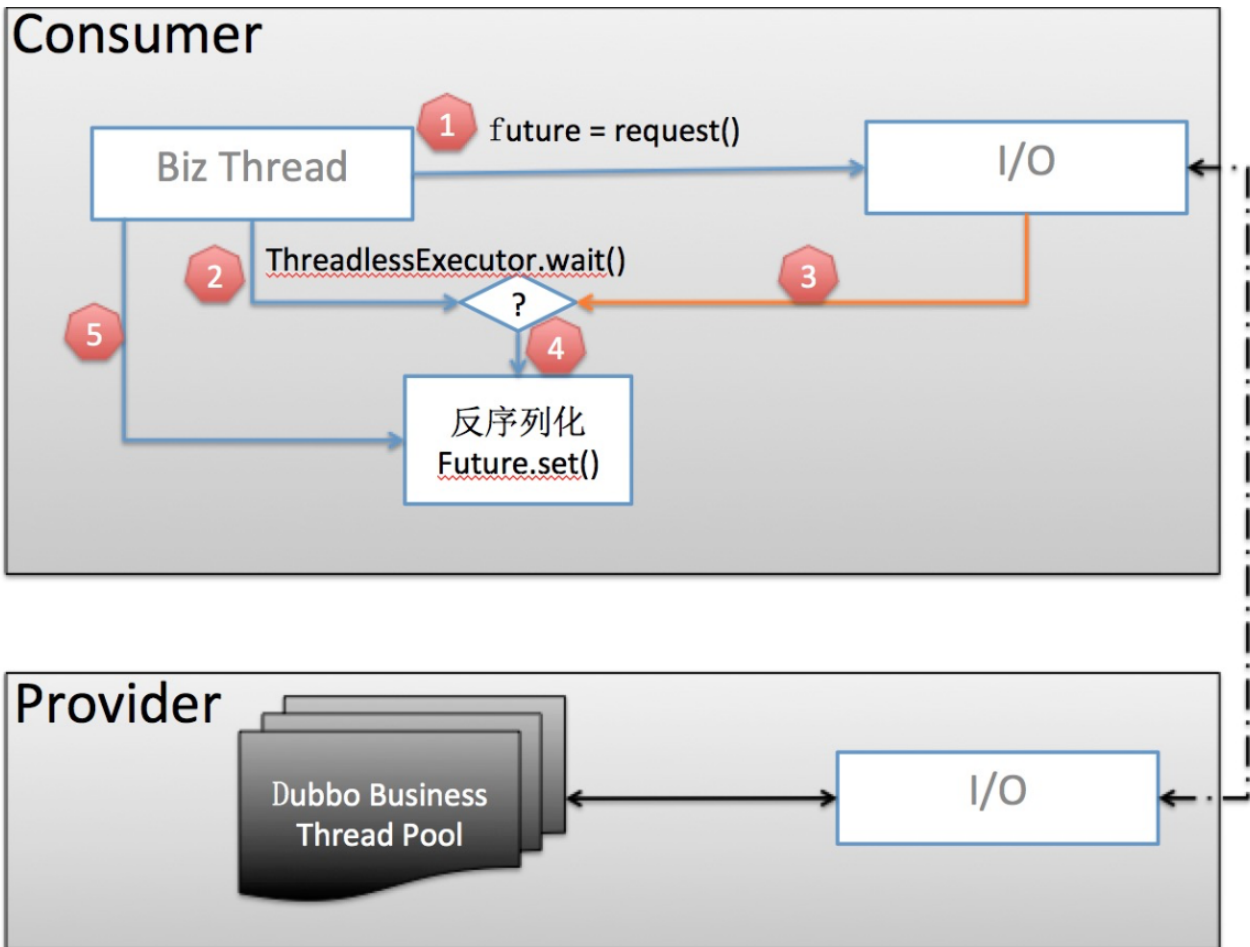


image-20251018133145303

1. 业务端发起请求，拿到一个Future实例
2. 调用 `future.get()` 之前，先调用 `ThreadlessExecutor.wait()`，`wait` 会使业务线程在一个阻塞队列上等待，直到队列中被加入元素
3. 业务数据返回一个 **Runnable Task**，放入到 `ThreadlessExecutor` 队列
4. 业务现场从队列拿出数据，进行RPC反序列化并进行 `future.set()`
5. 业务线程拿到结果进行返回

提供端线程模型

以Triple为例

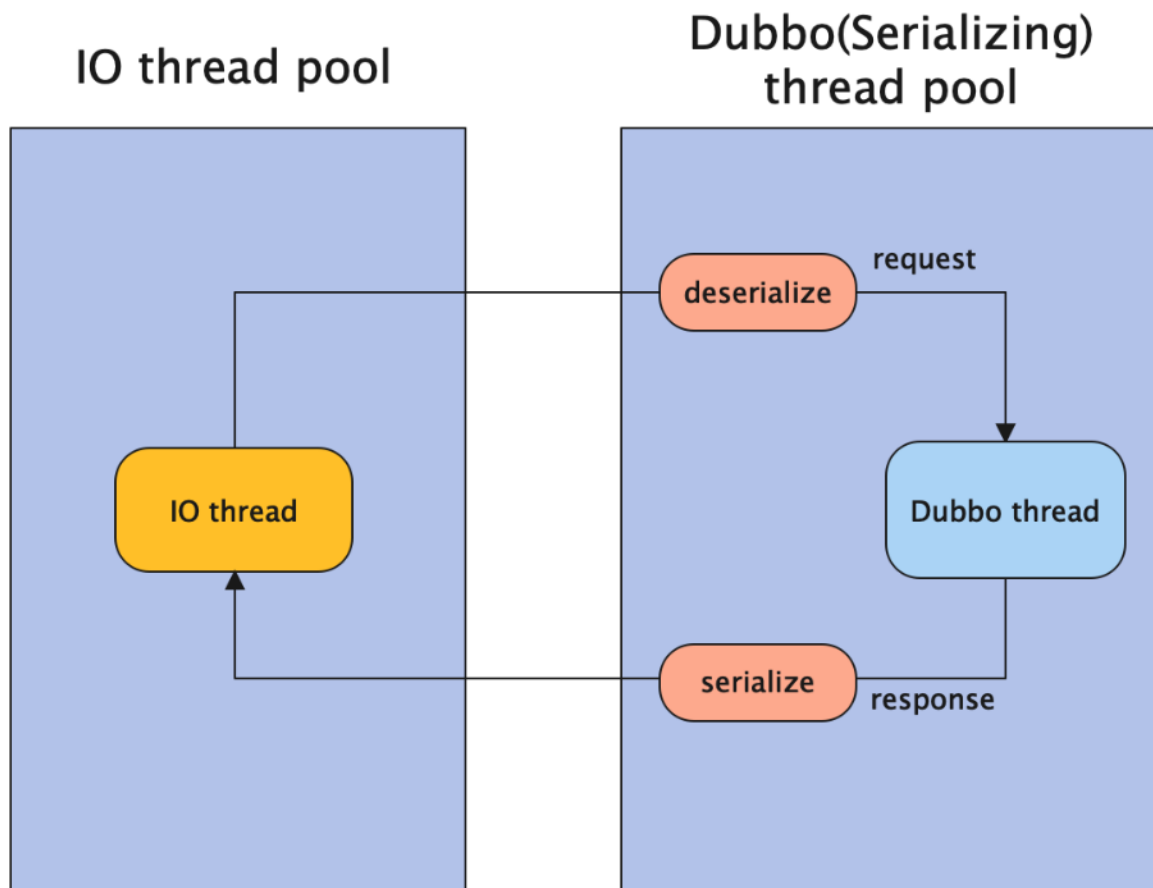


image-20251018133818557

序列化和反序列化操作都在Dubbo线程上工作，而IO线程并没有承载这些工作

异步调用

Dubbo线程池：处理RPC网络通信相关的任务，包括请求接收、响应发送、序列化/反序列化等

可分为 **Provider** 异步调用和 **Consumer** 异步调用两种模式，二者**相互独立，可进行任意正交组合**

- Consumer** 异步调用指的是发起RPC调用后**立即返回**，调用线程执行其他任务，当响应结果返回后通过**回调函数**通知消费端结果
 - 让IO线程进行请求参数的序列化，请求的发送等工作，使网络IO和业务线程解耦

消费端异步调用工作示例

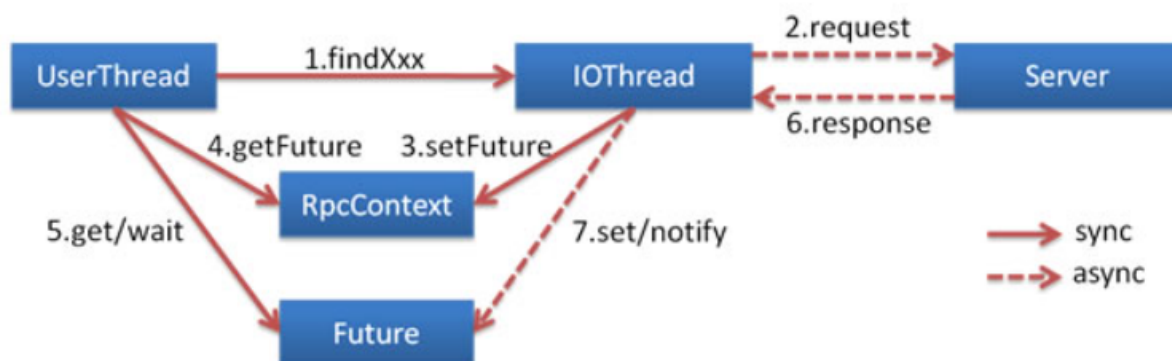


image-20251021113319445

- **Provider** 端异步执行将阻塞的业务从 **Dubbo 内部线程池**切换到业务自定义线程，避免Dubbo线程池的过度占用

Provider异步

- 接口定义使用 **CompletableFuture**
- 服务实现需要使用 **CompletableFuture** 设置回调函数
 - 使用这个类可以将线程从Dubbo线程切换到业务线程，防止对Dubbo线程池的阻塞
 - 当结果未响应时，返回 **null**

接口定义

```
public interface AsyncService {  
    /**  
     * 同步调用方法  
     */  
    String invoke(String param);  
    /**  
     * 异步调用方法  
     */  
    CompletableFuture<String> asyncInvoke(String param);  
}
```

接口实现

```

Service
class AsyncServiceImpl implements AsyncService {

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

```

Consumer异步

- 方法一：使用 `CompletableFuture` 声明服务方法返回值
 - 使用 `whenComplete` 方法写业务逻辑

```

@Override
public void run(String... args) throws Exception {
    //调用异步接口
    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);
        }
    });
}

```

- 方法二：在注解中配置 `async` 参数

```
Reference(async="true")
te AsyncService asyncService;
```

- 其它配置
 - 可以设置客户端**是否等待消息成功发出**
 - `sent` 为true客户端则会等待消息成功发出，否则抛出异常
 - `sent` 为false客户端则不等待消息发出

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

- 可以选择是否忽略返回值
 - `return` 为false则表示忽略返回值，不创建 `future` 对象，减少资源开销

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

泛化调用

核心：调用端不依赖具体接口类型，不需要在编译期知道服务提供方的接口定义

可以通过一个通用的 `GenericService` 接口对所有服务发起请求，使用场景如下：

- 网关服务
- 测试平台

泛化调用中调用端需要知道

- 服务接口名
- 方法名
- 参数值和参数类型

Spring调用方式（以XML为例）

1. 生产者端无需改动
2. 消费者端原有的 `dubbo:reference` 标签加上 `generic=true` 的属性
3. 获取到 Bean 容器，通过 Bean 容器拿到 `GenericService` 实例。
4. 调用 `$invoke` 方法获取结果
 - 方法名
 - 参数类型数组
 - 参数值数组

```

    te static GenericService genericService;

    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("spring/generic-impl-consumer.xml");
        context.start();
        //服务对应bean的名字由xml标签的id决定
        genericService = context.getBean("helloService");
        //获得结果
        Object result = genericService.$invoke("sayHello", new String[]{"java.lang.String"}, new Object[]{"world"});
    }

```

当然也可以通过注解的方式进行服务接口名的声明

```

    @Reference(
        interfaceName = "com.example.UserService", // 必须指定接口名字字符串
        generic = true                               // 启用泛化调用
    )

    te GenericService genericService;

```

Filter拦截器

Dubbo Filter 是基于SPI的拦截器，用于在RPC调用链路上增强功能

```

scope = ExtensionScope.MODULE)

public interface Filter extends BaseFilter {}

```

使用步骤

1. 定义自己的Filter，实现 Filter 接口

```

@Activate(group = {Constants.PROVIDER, Constants.CONSUMER}) // 可选，自动激活
public class MyFilter implements Filter {

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

        // 调用下一个 Filter 或 RPC 实现
        Result result = invoker.invoke(invocation);

        // 请求后逻辑
        System.out.println("After RPC: " + invocation.getMethodName());

        return result;
    }
}

```

- 可实现自动激活
- @Activate 常用属性
 - group：指定作用端（Provider/Consumer）
 - value：指定 URL 参数名，当 URL 中存在这些参数时才激活
 - order：控制执行顺序，越小越先执行
- 2. 进行SPI（服务发现机制）注册
 - 在 META-INF/dubbo/org.apache.dubbo.rpc.Filter 进行注册

超时时间

上文简单提了下，现在进行详细阐述

- 可以进行全局超时时间配置

```

:
Provider:
timeout: 5000

```

- 消费端对特定方法进行超时时间配置，提供端同理

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

配置形式的优先级从高到低依次为：方法级别配置 > 服务级别配置 > 全局配置 > 默认值

Deadline机制

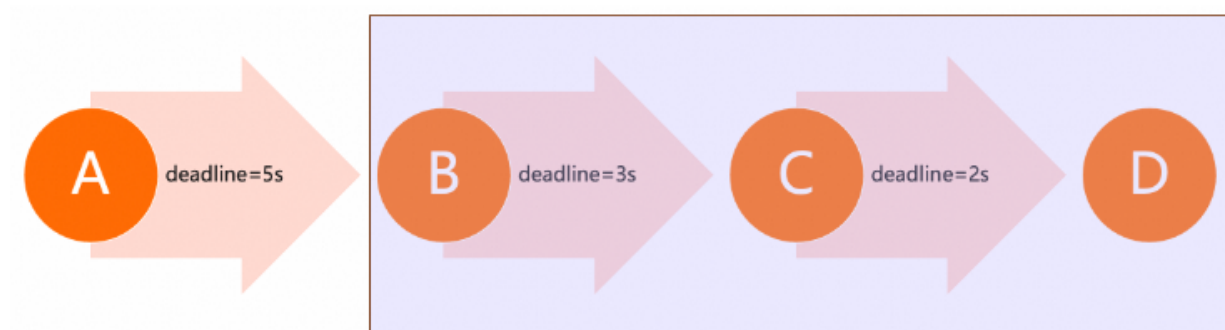


image-20251018141038253

- A调用B设置了超时时间为5s，因此 B->C->D 总计处理时间应该不超过5s
- Deadline机制是把 B->C->D 看成一个整体，随着时间流逝 deadline 会从 5s 逐步扣减，比如 C 收到请求时已经过去了 3s，则 C->D 的超时时间只剩下 2s

全局开启 deadline 机制

```
:
vider:
imeout: 5000
arameters.enable-timeout-countdown: true
```

指定某个服务开启 deadline 机制

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

集群容错

当集群调用失败时，Dubbo提供多种容错方案，默认为 failover 重试

调用链路

- 集群容错机制 (Cluster)

- 目录服务 (Directory)
- 路由 (Router)
- 负载均衡 (LoadBalance)
- 最终调用者 (Invoker)

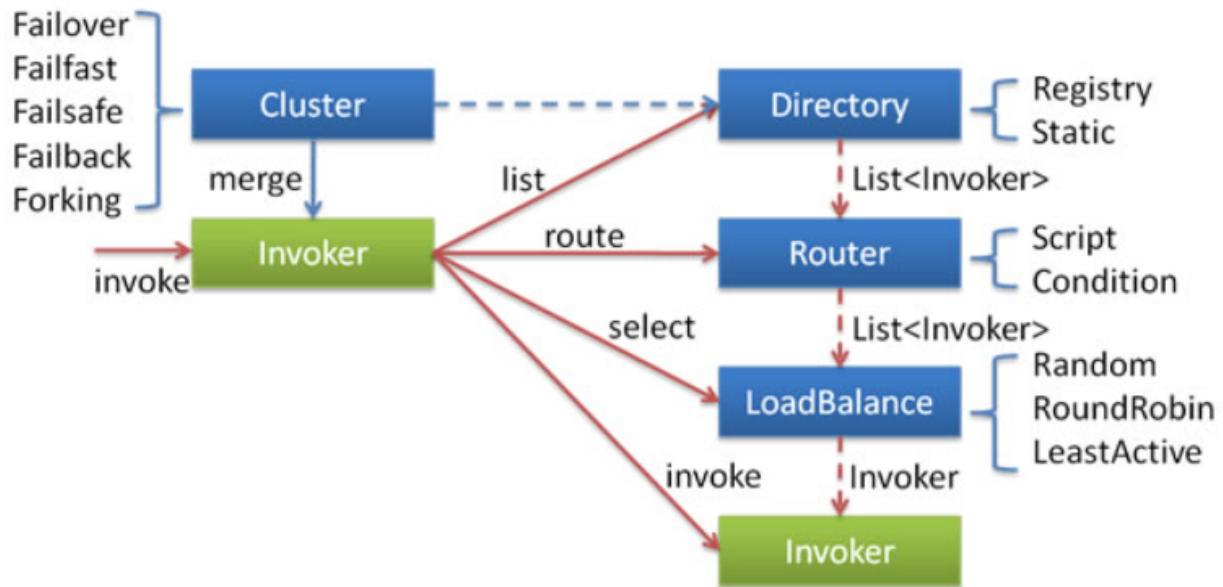


image-20251018142537953

各节点关系：

- Invoker 是 Provider 的一个可调用的 Service 的抽象，Invoker 封装了 Provider 地址以及 Service 的接口信息
- Directory 代表多个 Invoker
- Cluster 将 Directory 中的多个 Invoker 伪装成一个 Invoker，对上层透明，伪装过程包含了容错逻辑，调用失败后，重试另一个
- Router 负责从多个 Invoker 中按路由规则选出子集
- LoadBalance 负责从多个 Invoker 中选出具体的一个用于本次调用

集群容错模式

- Failover Cluster：失败自动切换，当出现失败，重试其它服务器
- failfast Cluster：快速失败，只发起一次调用，失败立即报错
- Failsafe Cluster：失败安全，出现失败时，直接忽略
- Failback Cluster：失败自动恢复，后台记录失败请求，定时重发
- Forking Cluster：并行调用多个服务器，只要一个成功即返回
- Broadcast Cluster：广播调用所有提供者，逐个调用，任意一台报错则报错