

[Dubbo入门](#)

简介

所谓**RPC**即远程过程调用

- 分布式系统
- 所谓**RPC**即
- 远程过程调用

Dubbo 与 Spring Boot 整合

RPC

所谓RPC即

RPC 即远程过程调用

所谓RPC即

- 分布式系统
- 所谓RPC即
- 远程过程调用
- 所谓**TCP**即
- 所谓**Netty**即**TCP**即

RPC入门

所谓->所谓->**TCP**

- 所谓**IDL**即所谓所谓所谓 pb 所谓 stub 所谓所谓 stub 所谓所谓
 - 所谓所谓
- 所谓所谓所谓所谓所谓所谓所谓encode decode
- 所谓RPC即所谓所谓所谓所谓**ID**即所谓所谓所谓所谓

Dubbo入门

- 所谓 Triple 与 dubbo 与 http 与 webservice 与
- 所谓 hessian2 与 dubbo 与 json 与 protobuf 与

gRPC

所谓RPC即

Google即所谓**RPC**即

- 所谓HTTP/2即所谓所谓所谓所谓**HPACK**即所谓所谓
- 所谓所谓 Protobuf
- 所谓所谓
- 所谓所谓 Kubernetes Istio Envoy 与

Triple

所谓RPC即

Dubbo3RPCDubbo3RPC

- HTTP/2
- Protobuf JSON
-
-

- Provider
- Consumer
- Registry
- Monitor
- Container

- - Admin
- Dubbo
 - DubboRPC
 - RPCDubboRPCDubbo

image-20251016172737311

Dubbo

-
- RBC
- Dubbo2 TCP

-
-
- +
- +
-

-

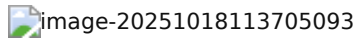
- 注册中心
- 负载均衡
- 熔断降级

Dubbo 注册中心&负载均衡器 Sentinel 熔断

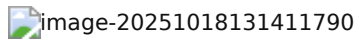
Router

负载均衡器

- 负载均衡器
- Router 负载均衡器
- 负载均衡器
- 负载均衡器



负载均衡器



负载均衡器 100% 负载均衡器 100% 负载均衡器

- 负载均衡器
- 负载均衡器
- 负载均衡器
- 负载均衡器

负载均衡器

- Provider 负载均衡器
 - applicationContext.xml 负载均衡器

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

or

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

- Consumer 负载均衡器
 - tag 负载均衡器

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

负载均衡器

- Provider 配置
 - 配置 yaml 格式

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

- configVersion: v3.0 配置
- force: true 强制 Consumer 使用 Provider
- enabled:true 是否启用
- key:shop-detail 键
- tags 标签
 - name:gray 名称
 - match 匹配规则 key=value 匹配 tag=gray
 - key 键
 - value 值

-
- Consumer 配置
 - 配置 tag 配置

配置

配置格式

- 配置
- 配置ID

配置/配置

```
# 配置
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 名称
- scope 作用域 service 或 application
- force 强制
 - true 强制注册
 - false 不强制注册
- enabled 是否启用
- priority 优先级
- conditions 条件
 - 通过配置项 getUser 指定 0 或 1001 指定 tag=gray 或 Provider

配置项

配置项的取值范围及默认值

配置项的取值范围 javascript 配置项

```
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)); // 配置项
```

配置

配置项的取值范围

配置项的取值范围 RPC 配置

配置项 = 配置 + 配置 + 配置

配置项的取值范围

- 配置项
- 配置项
- 配置项

Dubbo SPI

SPI 配置

配置项的取值范围

Dubbo SPI 配置项 ExtensionLoader 配置 ExtensionLoader 配置

- 配置项 @SPI 配置

- 通过META-INF/dubbo 文件
- Dubbo SPI 通过Dubbo SPI 文件

```
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 文件

- 通过META-INF/dubbo 文件
- 通过Dubbo SPI 文件

Dubbo SPI 文件

- @SPI 通过Dubbo SPI 文件
- @Adaptive 通过Dubbo SPI 文件
- Wrapper 通过Dubbo SPI 文件
 - 通过Dubbo SPI 文件

```
public class LogWrapper implements Log {
    private final Log log; // 通过Dubbo SPI 文件

    public LogWrapper(Log log) { // 通过Dubbo SPI 文件
        this.log = log;
    }

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

通过

通过

Maven配置

在 Maven 的 pom.xml 中添加 dubbo-spring-boot-starter 依赖，引入 Dubbo 和 Spring Boot 的集成包 **dubbo-spring-boot-starter**。

```
<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 应用名称
- dubbo.protocol.name 协议名称，默认为 RPC
- dubbo.protocol.port 协议端口
 - 默认为 -1，表示 Dubbo 会自动选择一个端口
 - 指定 ip:port 格式，表示指定 IP 和端口
- dubbo.registry.address 注册中心 IP 地址

Dubbo

- `@DubboService` 标注Dubbo提供者
- `@DubboReference` 标注Dubbo消费者
- `@EnableDubbo` 开启Dubbo支持
 - `EnableDubbo(scanBasePackages = {"org.apache.dubbo.springboot.demo.provider"})`

RPC

Triple

- 配置RPC `application.yaml`
 - 配置Dubbo

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

- 配置IDL
 - 配置Dubbo
 - 配置Dubbo `protoc` 生成IDL stub

Protobuf Triple

1. pom

```
<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> <!-- -->
  </configuration>
</plugin>
```

configuration

- `outputDir` 生成Java
- `protoSourceDir` 生成proto

2. IDL .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 描述语言
- package 包名
- option 选项
- message 消息
 - 请求
 - 响应
 - 消息体
- service 服务
 - rpc 远程过程调用
 - 接口
 - returns() 返回

配置

配置项

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

配置

- 在 application.yml 中配置

```

dubbo
  registry
    address: nacos://localhost:8848

```

- 在代码中配置 `@DubboService` 和 `@DubboReference`

```

dubbo:
  provider:
    delay: 5000

```

配置

- 在代码中配置 `@DubboService` 和 `@DubboReference`
- 在代码中配置 `@DubboService` 和 `@DubboReference`

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

####

- 注册中心地址
- 是否订阅注册中心

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

####

注册中心地址

####

1. 注册中心 **token**

```
dubbo:
  provider:
    token: true      # 注册中心 UUID
```

2. 注册中心**token**

3. 注册中心

- 注册中心token
- Dubbo注册中心tokenRPC调用 metadata
- 注册中心Dubbo注册中心token

####

####

1. Dubbo Admin
2. 注册中心 > 注册中心
3. 注册中心**key**注册中心**timeout**注册中心

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

- configVersion 注册中心
- enabled 注册中心

- side 指定 Provider 还是 Consumer
- parameters.timeout 超时
- retries 重试
- accesslog 访问日志

配置

配置示例

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

- key 唯一
- force 强制覆盖
- enabled 是否启用
- priority 优先级
- conditions 条件
 - 表达式

```
method=<方法> & arguments[<索引>]=<值> => <条件>=<值>
```

配置

配置示例


```
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 作用域
- configs 配置
- side 侧

- `match` 匹配
 - `key` 匹配
 - `value.exact` 精确
- `parameters.weight` 权重 0-100

RPC


分布式

 image-20251018133145303

1. 创建 `Future`
2. 调用 `future.get()` 阻塞 `ThreadlessExecutor.wait()` 或 `wait` 阻塞
3. 实现 `Runnable Task` 接口 `ThreadlessExecutor`
4. 实现 `RPC` 接口 `future.set()`
5. 实现 `Runnable`

分布式

`Triple`

 image-20251018133818557

分布式 `Dubbo` 接口 `IO`


接口

Dubbo 接口 `RPC` 接口 `IO`

接口 `Provider` 接口 `Consumer` 接口

- `Consumer` 接口 `RPC` 接口 `IO`
- `IO` 接口 `IO`

接口

 image-20251021113319445

- `Provider` 接口 **Dubbo** 接口 `Dubbo` 接口

Provider

- 接口 `CompletableFuture`
- 接口 `CompletableFuture` 接口
 - 接口 `Dubbo` 接口 `Dubbo` 接口
 - 接口 `null`

接口

```
public interface AsyncService {
    /**
     * 接口
```

```

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

```

测试

```

@DubboService
public class AsyncServiceImpl implements AsyncService {

    @Override
    public CompletableFuture<String> asyncInvoke(String param) {
        // 异步supplyAsync
        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

- 使用 CompletableFuture 测试
 - 使用 whenComplete 测试

```

@Override
public void run(String... args) throws Exception {
    //测试
    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);
        }
    })
}

```

```
});  
}
```

- 通过 `async` 设置

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

- 通过 `sent` 设置
 - 通过 `sent` 设置
 - `sent = true` 表示同步调用
 - `sent = false` 表示异步调用

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

- 通过 `return` 设置
 - `return = false` 表示同步调用，`future` 表示异步调用

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

通过

```
通过 GenericService 接口
```

通过 `GenericService` 接口

- 通过
- 通过

通过

- 通过
- 通过
- 通过

Spring 通过 XML 配置

1. 通过
2. 通过 `dubbo:reference` 设置 `generic=true` 属性
3. 通过 `Bean` 设置 `GenericService` 接口
4. 通过 `$invoke` 设置
 - 通过
 - 通过
 - 通过

```

private static GenericService genericService;

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

```

测试成功，输出如下：

```

@DubboReference(
    interfaceName = "com.example.UserService", // 接口名称
    generic = true // 是否支持泛型
)
private GenericService genericService;

```

Filter

Dubbo Filter 是 SPI 定义的 RPC 过滤器。

```

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

```

实现

1. 实现 Filter 接口

```

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

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

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

        // 打印日志
        System.out.println("After RPC: " + invocation.getMethodName());

        return result;
    }
}

```

```
}  
}
```

- 配置信息
- `@Activate` 注解
 - `group` 指定 Provider/Consumer
 - `value` 指定 URL 地址 URL 地址
 - `order` 指定优先级

2. SPI 接口

- `META-INF/dubbo/org.apache.dubbo.rpc.Filter` 接口

配置

配置信息

- 配置信息

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

- 配置信息

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

配置信息 > 配置信息 > 配置信息 > 配置

Deadline

image-20251018141038253

- A 向 B 发送消息 5s 后 B 向 C 发送消息 5s
- Deadline 配置 B 向 C 发送消息 deadline 为 5s 后 C 向 D 发送消息 3s 后 C 向 D 发送消息 2s

配置 deadline 配置

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

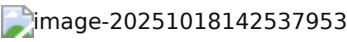
配置信息 deadline 配置

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

配置

Dubbo failover

- Cluster
- Directory
- Router
- LoadBalance
- Invoker



- Invoker Provider Service Invoker Provider Service
- Directory Invoker
- Cluster Directory Invoker Invoker
- Router Invoker
- LoadBalance Invoker

- Failover Cluster
- failfast Cluster
- Failsafe Cluster
- Failback Cluster
- Forking Cluster
- Broadcast Cluster