01-dubbo入门

目标

1. 掌握dubbo是什么
2. 掌握dubbo能做什么
3. 掌握dubbo的架构
4. 掌握dubbo的特性
5. 掌握dubbo使用

0 为什么要学Dubbo?

1 dubbo是什么

一款高性能的Java RPC框架

一款简单、易用的Java RPC框架

一款优秀的RPC服务治理框架

由阿里贡献的开源RPC框架。

官网地址：http://dubbo.apache.org/zh-cn/index.html

官方学习文档：http://dubbo.apache.org/zh-cn/docs/user/quick-start.html

github：https://github.com/apache/incubator-dubbo

2 dubbo能做什么

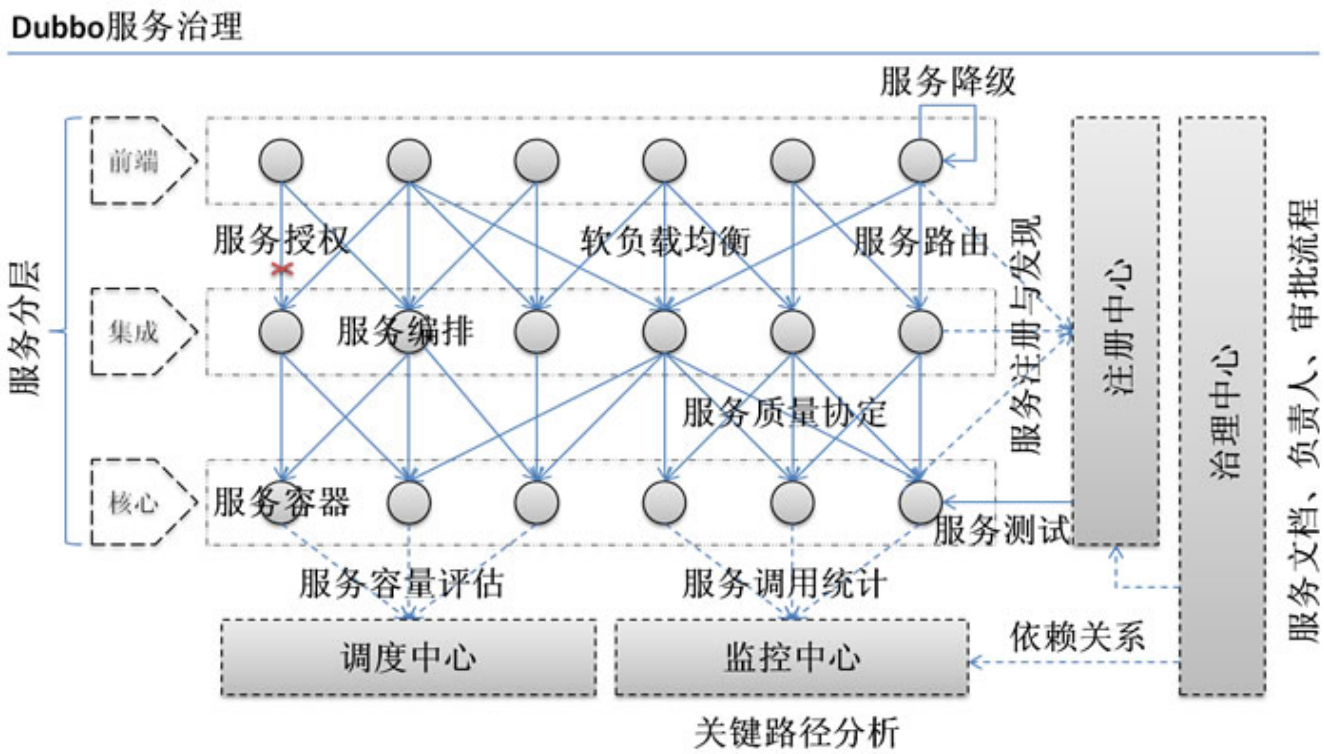
服务开发（rpc应用开发）

服务软负载均衡

服务依赖管理

服务监控

服务治理



参考：http://dubbo.apache.org/zh-cn/docs/user/preface/requirements.html

在大规模服务化之前，应用可能只是通过 RMI 或 Hessian 等工具，简单的暴露和引用远程服务，通过配置服务的URL地址进行调用，通过 F5 等硬件进行负载均衡。

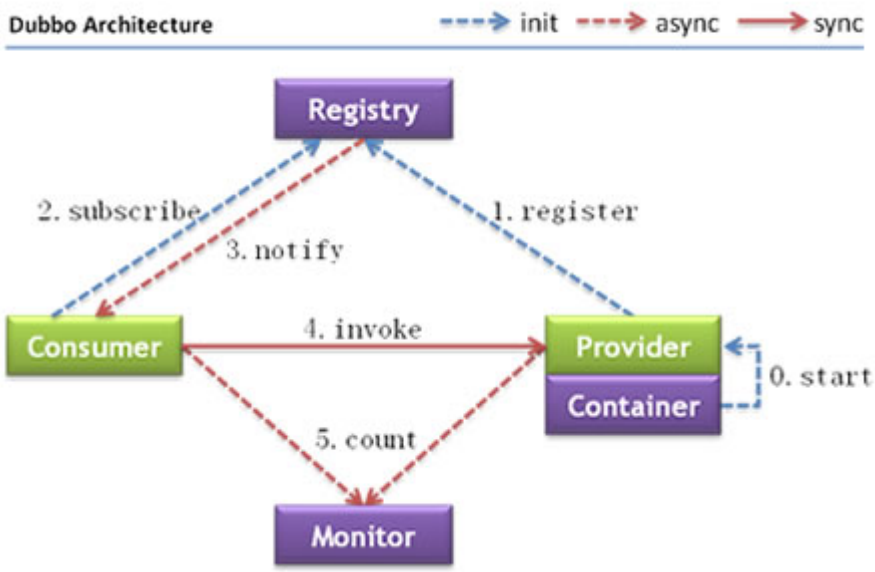
当服务越来越多时，服务 URL 配置管理变得非常困难，F5 硬件负载均衡器的单点压力也越来越大。 此时需要一个服务注册中心，动态地注册和发现服务，使服务的位置透明。并通过在消费方获取服务提供方地址列表，实现软负载均衡和 Failover，降低对 F5 硬件负载均衡器的依赖，也能减少部分成本。

当进一步发展，服务间依赖关系变得错踪复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。 这时，需要自动画出应用间的依赖关系图，以帮助架构师理清理关系。

接着，服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要多少机器支撑？什么时候该加机器？ 为了解决这些问题，第一步，要将服务现在每天的调用量，响应时间，都统计出来，作为容量规划的参考指标。其次，要可以动态调整权重，在线上，将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阈值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

以上是 Dubbo 最基本的几个需求。

3 dubbo架构



注意途中的构成部分、顺序。

节点角色说明

节点	角色说明
`Provider`	暴露服务的服务提供方
`Consumer`	调用远程服务的服务消费方
`Registry`	服务注册与发现的注册中心
`Monitor`	统计服务的调用次数和调用时间的监控中心
`Container`	服务运行容器

调用关系说明

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

Dubbo 架构具有以下几个特点，分别是连通性、健壮性、伸缩性、以及向未来架构的升级性。

连通性

- 注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小
- 监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示
- 服务提供者向注册中心注册其提供的服务，并汇报调用时间到监控中心，此时间不包含网络开销
- 服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，同时汇报调用时间到监控中心，此时间包含网络开销
- 注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外
- 注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者
- 注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表

- 注册中心和监控中心都是可选的，服务消费者可以直连服务提供者

健壮性

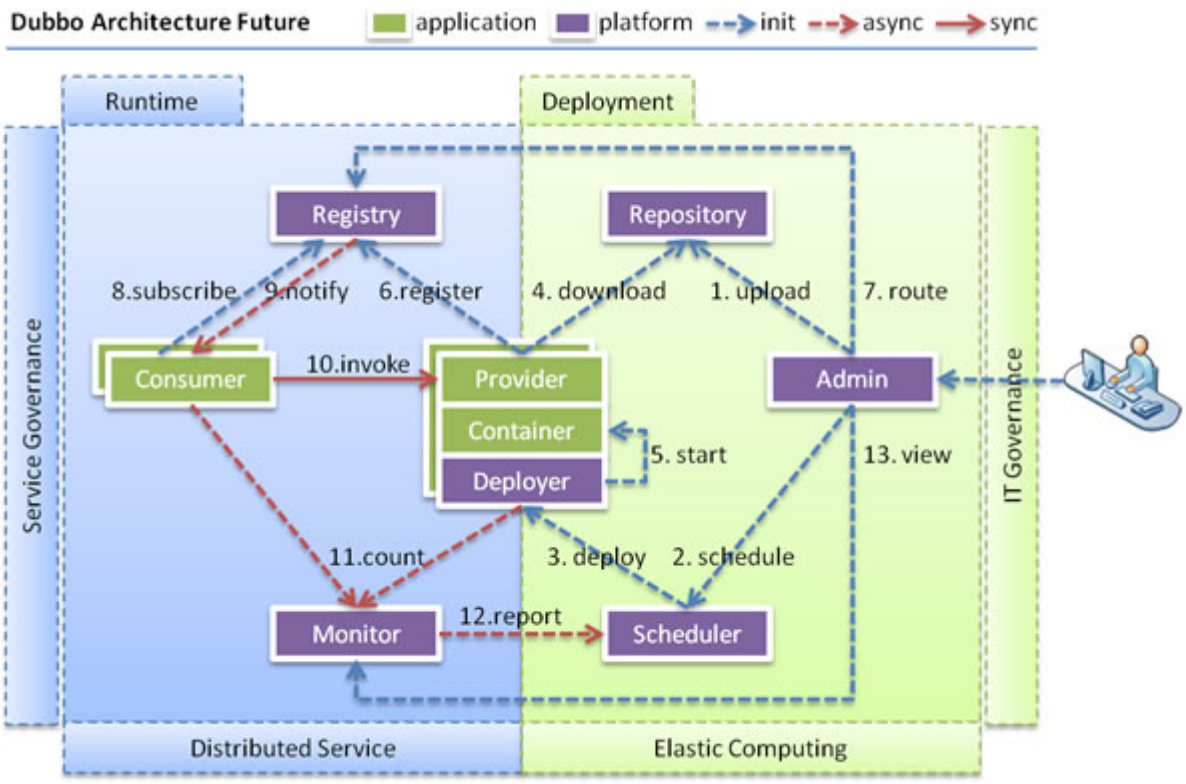
- 监控中心宕掉不影响使用，只是丢失部分采样数据
- 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务
- 注册中心对等集群，任意一台宕掉后，将自动切换到另一台
- 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯
- 服务提供者无状态，任意一台宕掉后，不影响使用
- 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

伸缩性

- 注册中心为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的注册中心
- 服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者

升级性

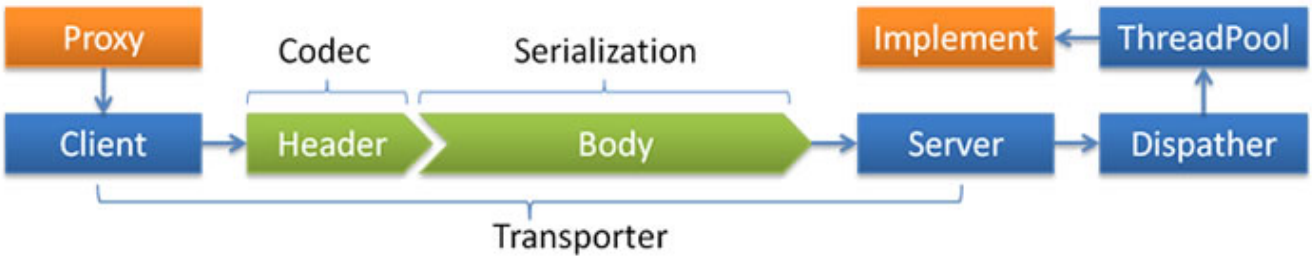
当服务集群规模进一步扩大，带动IT治理结构进一步升级，需要实现动态部署，进行流动计算，现有分布式服务架构不会带来阻力。下图是未来可能的一种架构：



节点角色说明

节点	角色说明
`Deployer`	自动部署服务的本地代理
`Repository`	仓库用于存储服务应用发布包
`Scheduler`	调度中心基于访问压力自动增减服务提供者
`Admin`	统一管理控制台
`Registry`	服务注册与发现的注册中心
`Monitor`	统计服务的调用次数和调用时间的监控中心

服务调用工作流程



4 dubbo使用

4.0 依赖说明

学习：http://dubbo.apache.org/zh-cn/docs/user/dependencies.html

4.1 可以如何使用dubbo

服务提供端：

1. 独立的服务（以普通的java程序形式）
2. 集成在应用中（在应用中增加远程服务能力）

消费端：

1. 在应用中调用远程服务。
2. 也可是在服务提供者中调用远程服务。

4.2 dubbo的使用步骤

1. 引入dubobo相关依赖
2. 配置dubbo框架（提供了3中配置方式）
3. 开发服务
4. 配置服务
5. 启动、调用

4.2.1 引入dubbo相关依赖

```
<dependencies>
  <dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>dubbo</artifactId>
      <version>2.6.6</version>
  </dependency>
  <!-- 这里我们使用netty -->
  <dependency>
      <groupId>io.netty</groupId>
      <artifactId>netty-all</artifactId>
      <version>4.1.32.Final</version>
  </dependency>
</dependencies>
```

4.2.2 配置dubbo框架

Dubbo 采用全 Spring 配置方式，透明化接入应用，对应用没有任何 API 侵入，只需用 Spring 加载 Dubbo 的配置即可，Dubbo 基于 Spring 的 Schema 扩展 进行加载。

如果不想使用 Spring 配置，可以通过 API 的方式 进行调用。还可以在spring中基于注解的方式进行配置。

3种配置方式：

- spring schema xml 方式 适用于spring应用
- 注解方式 适用于spring应用， 需要 `2.6.3` 及以上版本
- API方式 API方式使用范围说明：API 仅用于 OpenAPI, ESB, Test, Mock 等系统集成。普通服务提供方或消费方，请采用XML 配置方式使用 Dubbo

4.2.2.1 spring Schema XML 方式

服务提供者

示例服务：

服务接口定义（该接口需单独打包，在服务提供方和消费方共享）：

DemoService.java

```
public interface DemoService {
    String sayHello(String name);
}
```

在服务提供方实现接口

DemoServiceImpl.java

```
public class DemoServiceImpl implements DemoService {
    public String sayHello(String name) {
        return "Hello " + name;
    }
}
```

用 Spring 配置声明暴露服务

provider.xml（放在类目录下）：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
        http://www.springframework.org/schema/beans/spring-beans-4.3.xsd

        http://dubbo.apache.org/schema/dubbo
        http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

    <!-- 提供方应用信息，用于计算依赖关系 -->
    <dubbo:application name="hello-world-app"  />

    <!-- 使用multicast广播注册中心暴露服务地址 -->
    <dubbo:registry address="multicast://224.5.6.7:1234" />

    <!-- 用dubbo协议在20880端口暴露服务 -->
    <dubbo:protocol name="dubbo" port="20880" />

    <!-- 声明需要暴露的服务接口 -->
    <dubbo:service interface="com.study.mike.dubbo.DemoService"
ref="demoService" />

    <!-- 和本地bean一样实现服务 -->
    <bean id="demoService"
class="com.study.mike.dubbo.provider.DemoServiceImpl" />
</beans>
```

注意dubbo命名空间的指定，以及配置了哪些项。

启动服务程序（这里是作为独立的java程序启动）

Provider.java：

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Provider {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("provider.xml");
        context.start();
        System.in.read(); // 按任意键退出
    }
}
```

服务消费者

通过 Spring 配置引用远程服务

consumer.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-
4.3.xsd
            http://dubbo.apache.org/schema/dubbo
            http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

    <!-- 消费方应用名，用于计算依赖关系，不是匹配条件，不要与提供方一样 -->
    <dubbo:application name="consumer-of-helloworld-app"  />

    <!-- 使用multicast广播注册中心暴露发现服务地址 -->
    <dubbo:registry address="multicast://224.5.6.7:1234" />

    <!-- 生成远程服务代理，可以和本地bean一样使用demoService -->
    <dubbo:reference id="demoService"
interface="com.study.mike.dubbo.DemoService" />
</beans>
```

加载Spring配置，并调用远程服务

Consumer.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.apache.dubbo.demo.DemoService;

public class Consumer {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("consumer.xml");
        context.start();
        DemoService demoService = (DemoService)
context.getBean("demoService"); // 获取远程服务代理
        String hello = demoService.sayHello("world"); // 执行远程方法
        System.out.println(hello); // 显示调用结果
        context.close();
    }
}
```

4.2.2.2 注解方式

需要 `2.6.3` 及以上版本

服务提供方

`@Service`注解暴露服务，注意是com.alibaba.dubbo.config.annotation.Service

```
import com.alibaba.dubbo.config.annotation.Service;
import com.study.mike.rpc.demo.DemoService;

@Service
public class DemoServiceImpl implements DemoService {
    public String sayHello(String name) {
        return "Hello " + name;
    }
}
```

增加应用共享配置：classpath:/dubbo/dubbo-provider.properties

```
# dubbo-provider.properties
dubbo.application.name=annotation-provider
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.protocol.name=dubbo
dubbo.protocol.port=20880
```

指定Spring扫描路径，启动服务

```
@Configuration
@EnableDubbo(scanBasePackages = "com.study.mike.dubbo.provider ")
@PropertySource("classpath:/dubbo/dubbo-provider.properties")
public class AnnotationProviderConfiguration {
    public static void main(String[] args) throws Exception {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(
                AnnotationProviderConfiguration.class);
        context.start();
        System.in.read(); // 按任意键退出
        context.close();
    }
}
```

服务消费方

`Reference`注解引用服务

```
@Component
public class AnnotationDemoAction {

    @Reference
    private DemoService demoService;

    public String doSayHello(String name) {
        return demoService.sayHello(name);
    }
}
```

增加应用共享配置：classpath:/dubbo/dubbo-consumer.properties

```
# dubbo-consumer.properties
dubbo.application.name=annotation-consumer
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.consumer.timeout=3000
```

指定Spring扫描路径，调用服务

```
@Configuration
@EnableDubbo(scanBasePackages = "com.study.mike.dubbo.consumer")
@PropertySource("classpath:/dubbo/dubbo-consumer.properties")
@ComponentScan(value = { "com.study.mike.dubbo.consumer" })
public class AnnotationConsumerConfiguration {

    public static void main(String[] args) throws Exception {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(
                AnnotationConsumerConfiguration.class);
        context.start();
        final AnnotationDemoAction annotationAction =
context.getBean(AnnotationDemoAction.class);
        String hello = annotationAction.doSayHello("world");
        System.out.println(hello);
        context.close();
    }
}
```

注意：示例中使用了zookeeper来做注册中心，要引入zookeeper相关依赖

```
    <!-- 默认使用的是第三方zookeeper客户端 curator -->
    <dependency>
        <groupId>org.apache.curator</groupId>
        <artifactId>curator-recipes</artifactId>
        <version>4.2.0</version>
        <!-- 如果你使用的zookeeper服务版本不是3.5的，请排除自动依赖，再单独引入
zookeeper依赖 -->
        <exclusions>
            <exclusion>
                <groupId>org.apache.zookeeper</groupId>
                <artifactId>zookeeper</artifactId>
            </exclusion>
        </exclusions>
```

```
    </dependency>
    <!-- 引入zookeeper服务对应版本的zookeeper jar -->
    <dependency>
        <groupId>org.apache.zookeeper</groupId>
        <artifactId>zookeeper</artifactId>
        <version>3.4.11</version>
    </dependency>
```

4.2.2.3 API方式

API方式使用范围说明：API 仅用于 OpenAPI, ESB, Test, Mock 等系统集成。普通服务提供方或消费方，请采用XML 配置方式使用 Dubbo

服务提供者

```
import com.alibaba.dubbo.config.ApplicationConfig;
import com.alibaba.dubbo.config.ProtocolConfig;
import com.alibaba.dubbo.config.RegistryConfig;
import com.alibaba.dubbo.config.ServiceConfig;
import com.study.mike.rpc.demo.DemoService;

public class ApiProviderConfiguration {

    public static void main(String[] args) throws Exception {
        // 服务实现
        DemoService demoService = new DemoServiceImpl();

        // 当前应用配置。 请学习ApplicationConfig的API
        ApplicationConfig application = new ApplicationConfig();
        application.setName("hello-world-app");

        // 连接注册中心配置。 请学习RegistryConfig的API
        RegistryConfig registry = new RegistryConfig("224.5.6.7:1234",
"multicast");

        // 服务提供者协议配置
        ProtocolConfig protocol = new ProtocolConfig();
        protocol.setName("dubbo");
        protocol.setPort(12345);
        protocol.setThreads(200);
```

```
        // 注意：ServiceConfig为重对象，内部封装了与注册中心的连接，以及开启服务端口
        // 服务提供者暴露服务配置。请学习ServiceConfig的API
        // 此实例很重，封装了与注册中心的连接，请自行缓存，否则可能造成内存和连接泄漏
        ServiceConfig<DemoService> service = new ServiceConfig<DemoService>
();
        service.setApplication(application);
        service.setRegistry(registry); // 多个注册中心可以用setRegistries()
        service.setProtocol(protocol); // 多个协议可以用setProtocols()
        service.setInterface(DemoService.class);
        service.setRef(demoService);
        service.setVersion("1.0.0");

        // 暴露及注册服务
        service.export();

        System.in.read(); // 按任意键退出
    }
}
```

服务消费者

```
import com.alibaba.dubbo.config.ApplicationConfig;
import com.alibaba.dubbo.config.ReferenceConfig;
import com.alibaba.dubbo.config.RegistryConfig;
import com.study.mike.rpc.demo.DemoService;

public class ApiConsumerConfiguration {

    public static void main(String[] args) {
        // 当前应用配置
        ApplicationConfig application = new ApplicationConfig();
        application.setName("consumer-of-helloworld-app");

        // 连接注册中心配置
        RegistryConfig registry = new RegistryConfig("224.5.6.7:1234",
"multicast");

        // 注意：ReferenceConfig为重对象，内部封装了与注册中心的连接，以及与服务提供
方的连接
        // 引用远程服务
        // 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造
```

```
成内存和连接泄漏
        ReferenceConfig<DemoService> reference = new
ReferenceConfig<DemoService>();
        reference.setApplication(application);
        reference.setRegistry(registry); // 多个注册中心可以用setRegistries()
        reference.setInterface(DemoService.class);
        reference.setVersion("1.0.0");

        // 和本地bean一样使用demoService
        DemoService demoService = reference.get(); // 注意：此代理对象内部封装
了所有通讯细节，对象较重，请缓存复用
        String hello = demoService.sayHello("API demo");
        System.out.println(hello);
    }
}
```

特殊场景

下面只列出不同的地方，其它参见上面的写法

方法级设置

```
...

// 方法级配置
List<MethodConfig> methods = new ArrayList<MethodConfig>();
MethodConfig method = new MethodConfig();
method.setName("createXxx");
method.setTimeout(10000);
method.setRetries(0);
methods.add(method);

// 引用远程服务
ReferenceConfig<XxxService> reference = new ReferenceConfig<XxxService>();
// 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连
接泄漏
...
reference.setMethods(methods); // 设置方法级配置

...
```

点对点直连

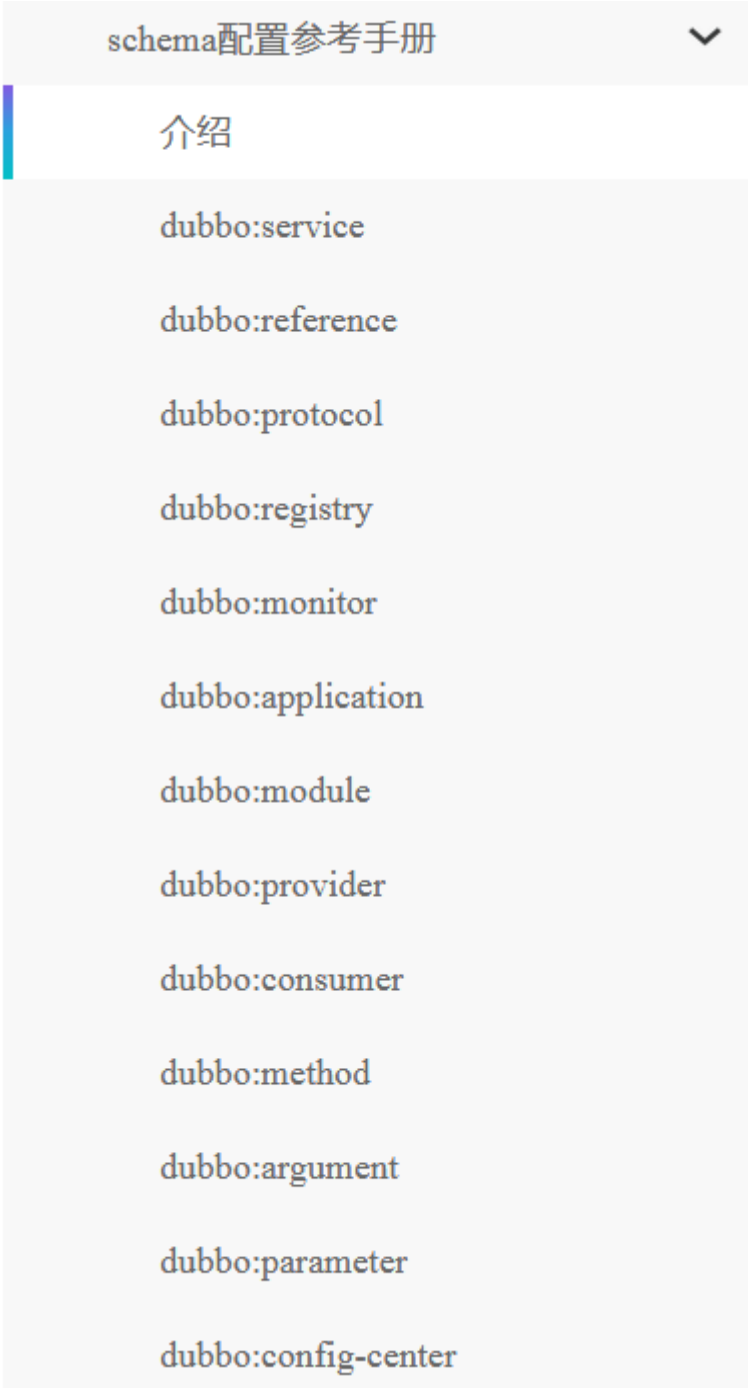
```
...

ReferenceConfig<XxxService> reference = new ReferenceConfig<XxxService>();
// 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏
// 如果点对点直连，可以用reference.setUrl()指定目标地址，设置url后将绕过注册中心，
// 其中，协议对应provider.setProtocol()的值，端口对应provider.setPort()的值，
// 路径对应service.setPath()的值，如果未设置path，缺省path为接口名
reference.setUrl("dubbo://10.20.130.230:20880/com.xxx.XxxService");

...
```

4.3 配置项学习

http://dubbo.apache.org/zh-cn/docs/user/references/xml/introduction.html



一定要了解各配置元素可配置属性。

4.4 spring boot中集成

方式一：@EnableDubbo 注解

0、引入对应的jar

```
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>dubbo</artifactId>
            <version>2.6.6</version>
        </dependency>
        <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>4.1.32.Final</version>
        </dependency>

        <!-- 默认使用的是第三方zookeeper客户端 curator -->
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-recipes</artifactId>
            <version>4.2.0</version>
            <!-- 如果你使用的zookeeper服务版本不是3.5的，请排除自动依赖，再单独引入
zookeeper依赖 -->
            <exclusions>
                <exclusion>
                    <groupId>org.apache.zookeeper</groupId>
                    <artifactId>zookeeper</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <!-- 引入zookeeper服务对应版本的zookeeper jar -->
        <dependency>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
            <version>3.4.11</version>
        </dependency>
```

1、在springboot 的启动类上加 @EnableDubbo 注解开启dubbo（服务提供者、消费者的是一样的，扫描的包可能不一样）

```
@SpringBootApplication
@EnableDubbo(scanBasePackages = "com.study.mike.dubbo.provider")
public class SpringBootDubboApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootDubboApplication.class, args);
    }
}
```

2、然后在application.yml中配置dubbo:

服务端示例：

```
#服务提供者 application.yml
spring:
  main:
    allow-bean-definition-overriding: true

dubbo:
  application:
    name: service-app1
  registry:
    address: zookeeper://127.0.0.1:2181
  protocol:
    name: dubbo
    port: 20880
```

消费者示例：

```
# 消费者 application.yml
spring:
  main:
    allow-bean-definition-overriding: true

server.port: 9000   #因在同一机器上跑spring-boot web，所以改下端口

dubbo:
```

```
  application:
    name: consumer-service-app1
  registry:
    address: zookeeper://127.0.0.1:2181
  consumer:
    timeout: 3000
```

在消费者提供Controller，测试一下

AnnotationDemoAction.java

```
@RestController
public class AnnotationDemoAction {

    @Reference
    private DemoService demoService;

    @RequestMapping("/hello")
    public String doSayHello(String name) {
        return demoService.sayHello(name);
    }
}
```

方式二：dubbo-spring-boot-starter方式

1、引入dubbo-spring-boot-starter 及对应的dubbo jar

```
        <dependency>
            <groupId>com.alibaba.boot</groupId>
            <artifactId>dubbo-spring-boot-starter</artifactId>
            <version>0.2.1.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>dubbo</artifactId>
            <version>2.6.6</version>
        </dependency>
        <dependency>
            <groupId>io.netty</groupId>
```

```
            <artifactId>netty-all</artifactId>
            <version>4.1.32.Final</version>
        </dependency>

        <!-- 默认使用的是第三方zookeeper客户端 curator -->
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-recipes</artifactId>
            <version>4.2.0</version>
            <!-- 如果你使用的zookeeper服务版本不是3.5的，请排除自动依赖，再单独引入
zookeeper依赖 -->
            <exclusions>
                <exclusion>
                    <groupId>org.apache.zookeeper</groupId>
                    <artifactId>zookeeper</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <!-- 引入zookeeper服务对应版本的zookeeper jar -->
        <dependency>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
            <version>3.4.11</version>
        </dependency>
```

2、配置

在application.yml完成和方式一相同的配置

在application.yml中通过dubbo.scan.base-packages参数指定dubbo扫描的包（服务提供者、消费者设置方式一样）

```
# application.yml
spring:
  main:
    allow-bean-definition-overriding: true

dubbo:
  application:
    name: service-app1
  registry:
    address: zookeeper://127.0.0.1:2181
```

```
  protocol:
    name: dubbo
    port: 20880
  scan:
    base-packages: com.study.mike.dubbo.provider
```

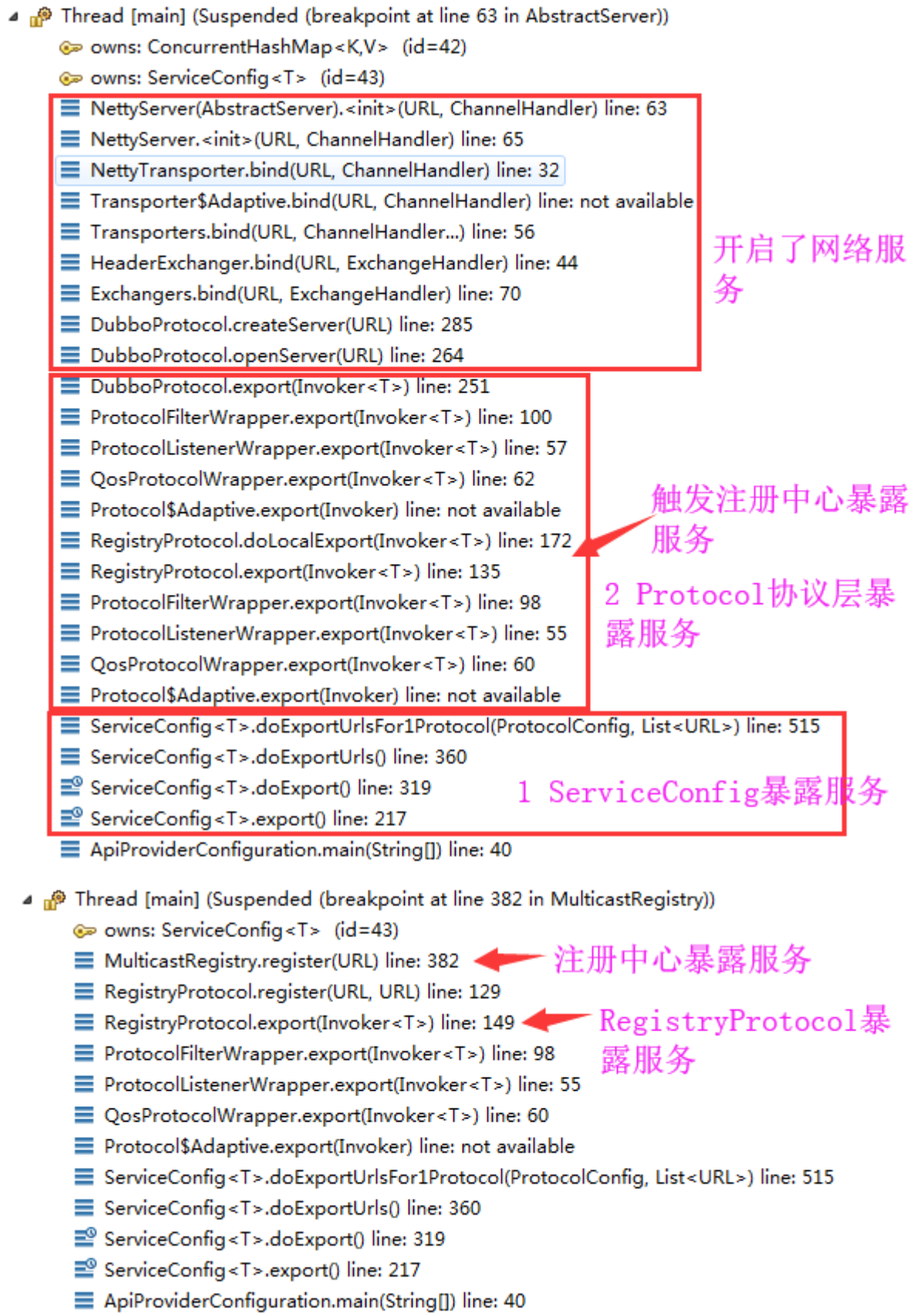
5 源码导读

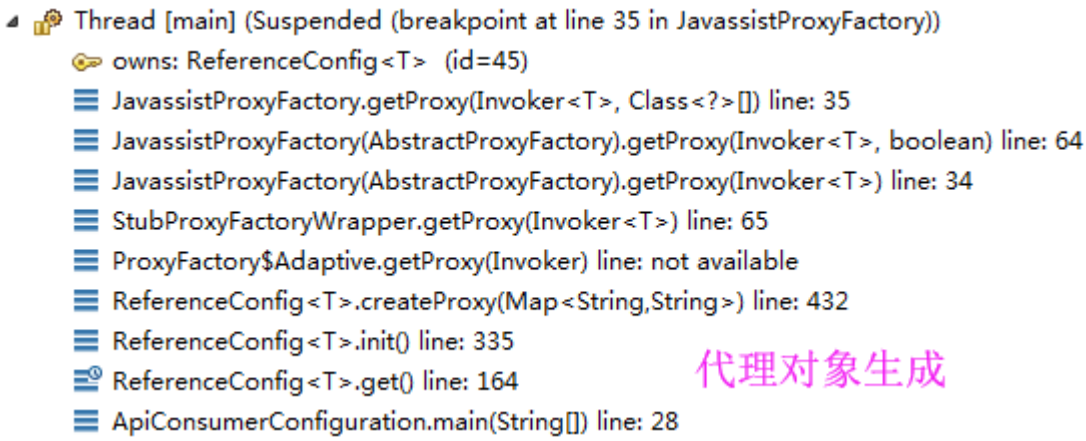
5.1 API方式工作过程解读

ServiceConfig

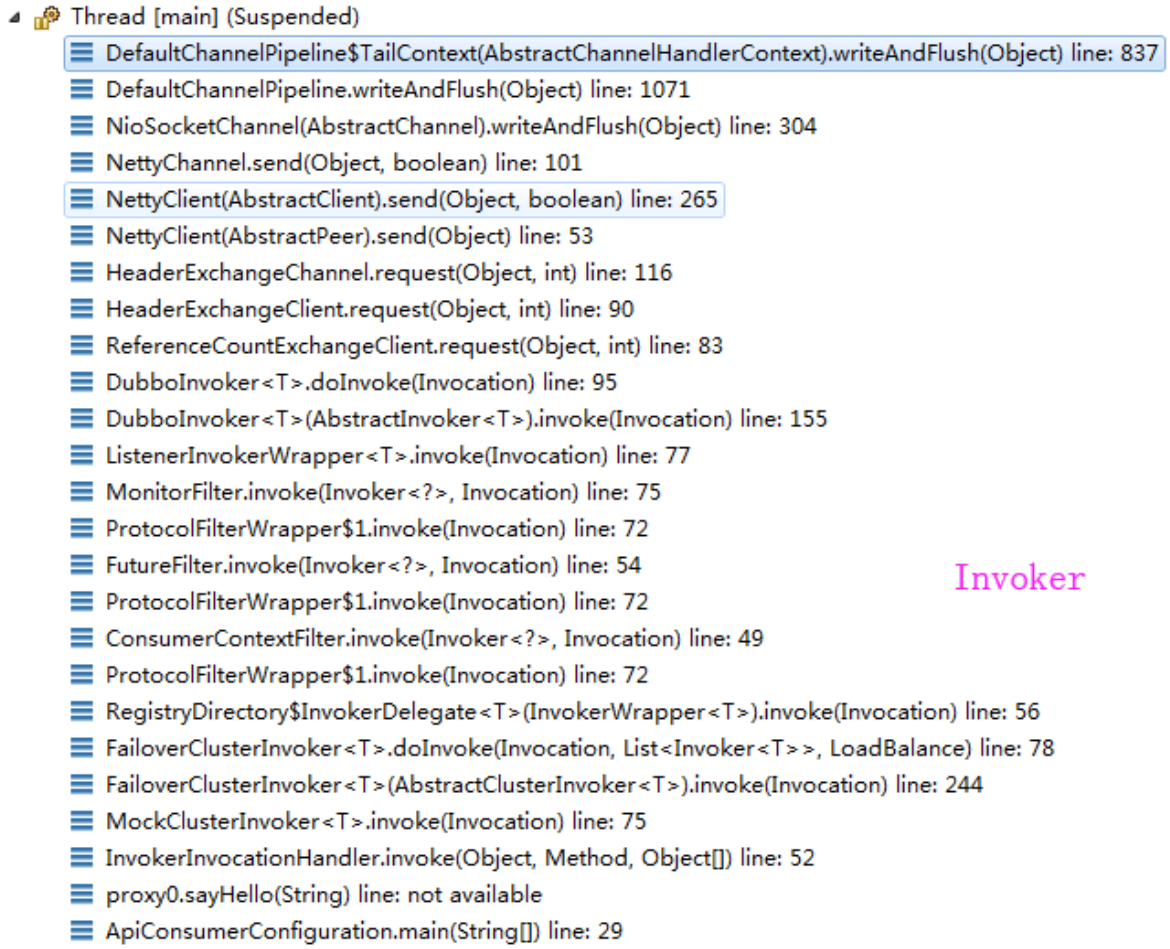
Invoker

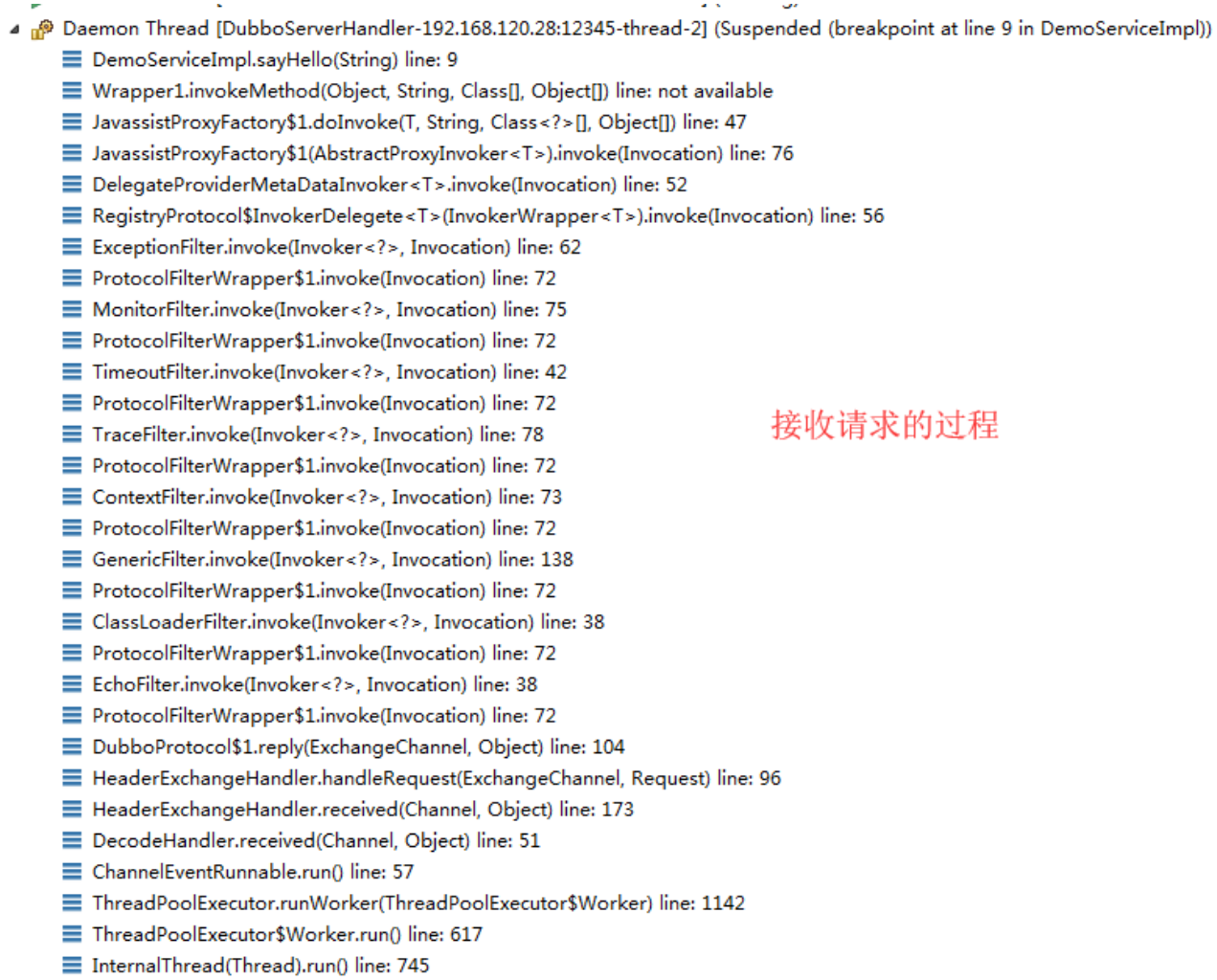
Protocol

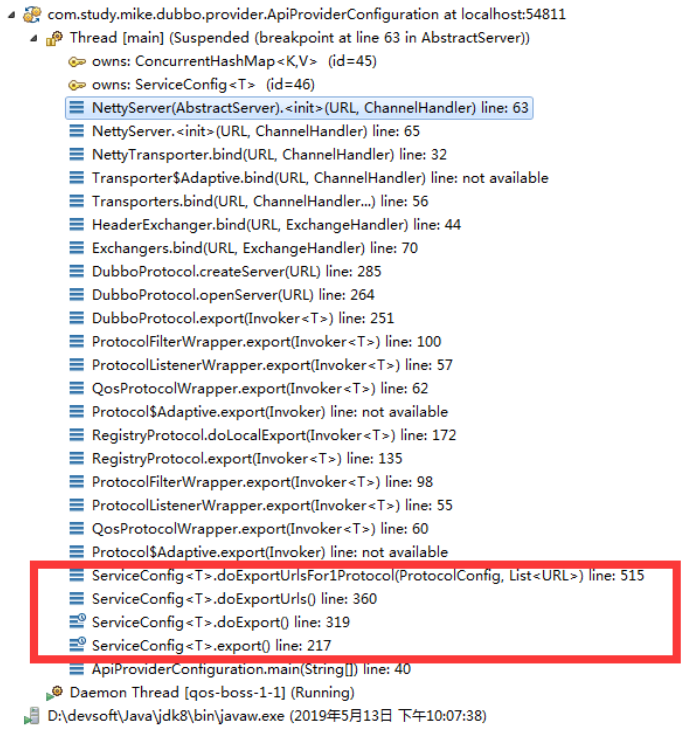

Thread [main] (Suspended (breakpoint at line 63 in AbstractServer))
owns: ConcurrentHashMap<K,V> (id=42)
owns: ServiceConfig<T> (id=43)
NettyServer(AbstractServer).<init>(URL, ChannelHandler) line: 63
NettyServer.<init>(URL, ChannelHandler) line: 65
NettyTransporter.bind(URL, ChannelHandler) line: 32
Transporter$Adaptive.bind(URL, ChannelHandler) line: not available
Transporters.bind(URL, ChannelHandler...) line: 56
HeaderExchanger.bind(URL, ExchangeHandler) line: 44
Exchangers.bind(URL, ExchangeHandler) line: 70
DubboProtocol.createServer(URL) line: 285
DubboProtocol.openServer(URL) line: 264
开启了网络服务
DubboProtocol.export(Invoker<T>) line: 251
ProtocolFilterWrapper.export(Invoker<T>) line: 100
ProtocolListenerWrapper.export(Invoker<T>) line: 57
QosProtocolWrapper.export(Invoker<T>) line: 62
Protocol$Adaptive.export(Invoker) line: not available
RegistryProtocol.doLocalExport(Invoker<T>) line: 172
RegistryProtocol.export(Invoker<T>) line: 135
ProtocolFilterWrapper.export(Invoker<T>) line: 98
ProtocolListenerWrapper.export(Invoker<T>) line: 55
QosProtocolWrapper.export(Invoker<T>) line: 60
Protocol$Adaptive.export(Invoker) line: not available
触发注册中心暴露服务
2 Protocol协议层暴露服务
ServiceConfig<T>.doExportUrlsFor1Protocol(ProtocolConfig, List<URL>) line: 515
ServiceConfig<T>.doExportUrls() line: 360
ServiceConfig<T>.doExport() line: 319
ServiceConfig<T>.export() line: 217
1 ServiceConfig暴露服务
ApiProviderConfiguration.main(String[]) line: 40
Thread [main] (Suspended (breakpoint at line 382 in MulticastRegistry))
owns: ServiceConfig<T> (id=43)
MulticastRegistry.register(URL) line: 382
注册中心暴露服务
RegistryProtocol.register(URL, URL) line: 129
RegistryProtocol.export(Invoker<T>) line: 149
RegistryProtocol暴露服务
ProtocolFilterWrapper.export(Invoker<T>) line: 98
ProtocolListenerWrapper.export(Invoker<T>) line: 55
QosProtocolWrapper.export(Invoker<T>) line: 60
Protocol$Adaptive.export(Invoker) line: not available
ServiceConfig<T>.doExportUrlsFor1Protocol(ProtocolConfig, List<URL>) line: 515
ServiceConfig<T>.doExportUrls() line: 360
ServiceConfig<T>.doExport() line: 319
ServiceConfig<T>.export() line: 217
ApiProviderConfiguration.main(String[]) line: 40

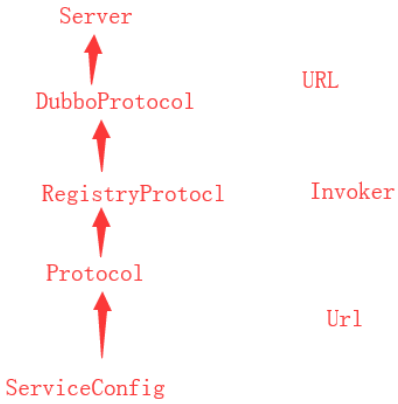

Thread [main] (Suspended (breakpoint at line 35 in JavassistProxyFactory))
owns: ReferenceConfig<T> (id=45)
JavassistProxyFactory.getProxy(Invoker<T>, Class<?>[]) line: 35
JavassistProxyFactory(AbstractProxyFactory).getProxy(Invoker<T>, boolean) line: 64
JavassistProxyFactory(AbstractProxyFactory).getProxy(Invoker<T>) line: 34
StubProxyFactoryWrapper.getProxy(Invoker<T>) line: 65
ProxyFactory$Adaptive.getProxy(Invoker) line: not available
ReferenceConfig<T>.createProxy(Map<String,String>) line: 432
ReferenceConfig<T>.init() line: 335
ReferenceConfig<T>.get() line: 164
ApiConsumerConfiguration.main(String[]) line: 28
代理对象生成


Thread [main] (Suspended (breakpoint at line 272 in MulticastRegistry))
owns: ReferenceConfig<T> (id=45)
MulticastRegistry.doSubscribe(URL, NotifyListener) line: 272
MulticastRegistry(FailbackRegistry).subscribe(URL, NotifyListener) line: 196
MulticastRegistry.subscribe(URL, NotifyListener) line: 394
RegistryDirectory<T>.subscribe(URL) line: 161
RegistryProtocol.doRefer(Cluster, Registry, Class<T>, URL) line: 310
RegistryProtocol.refer(Class<T>, URL) line: 290
ProtocolListenerWrapper.refer(Class<T>, URL) line: 65
QosProtocolWrapper.refer(Class<T>, URL) line: 69
ProtocolFilterWrapper.refer(Class<T>, URL) line: 106
Protocol$Adaptive.refer(Class, URL) line: not available
ReferenceConfig<T>.createProxy(Map<String,String>) line: 396
ReferenceConfig<T>.init() line: 335
ReferenceConfig<T>.get() line: 164
ApiConsumerConfiguration.main(String[]) line: 28
从注册中心获取服务信息


Thread [main] (Suspended)
DefaultChannelPipeline$TailContext(AbstractChannelHandlerContext).writeAndFlush(Object) line: 837
DefaultChannelPipeline.writeAndFlush(Object) line: 1071
NioSocketChannel(AbstractChannel).writeAndFlush(Object) line: 304
NettyChannel.send(Object, boolean) line: 101
NettyClient(AbstractClient).send(Object, boolean) line: 265
NettyClient(AbstractPeer).send(Object) line: 53
HeaderExchangeChannel.request(Object, int) line: 116
HeaderExchangeClient.request(Object, int) line: 90
ReferenceCountExchangeClient.request(Object, int) line: 83
DubboInvoker<T>.doInvoke(Invocation) line: 95
DubboInvoker<T>(AbstractInvoker<T>).invoke(Invocation) line: 155
ListenerInvokerWrapper<T>.invoke(Invocation) line: 77
MonitorFilter.invoke(Invoker<?>, Invocation) line: 75
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
FutureFilter.invoke(Invoker<?>, Invocation) line: 54
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
ConsumerContextFilter.invoke(Invoker<?>, Invocation) line: 49
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
RegistryDirectory$InvokerDelegate<T>(InvokerWrapper<T>).invoke(Invocation) line: 56
FailoverClusterInvoker<T>.doInvoke(Invocation, List<Invoker<T>>, LoadBalance) line: 78
FailoverClusterInvoker<T>(AbstractClusterInvoker<T>).invoke(Invocation) line: 244
MockClusterInvoker<T>.invoke(Invocation) line: 75
InvokerInvocationHandler.invoke(Object, Method, Object[]) line: 52
proxy0.sayHello(String) line: not available
ApiConsumerConfiguration.main(String[]) line: 29
Invoker


Daemon Thread [DubboServerHandler-192.168.120.28:12345-thread-2] (Suspended (breakpoint at line 9 in DemoServiceImpl))
DemoServiceImpl.sayHello(String) line: 9
Wrapper1.invokeMethod(Object, String, Class[], Object[]) line: not available
JavassistProxyFactory$1.doInvoke(T, String, Class<?>[], Object[]) line: 47
JavassistProxyFactory$1(AbstractProxyInvoker<T>).invoke(Invocation) line: 76
DelegateProviderMetaDataInvoker<T>.invoke(Invocation) line: 52
RegistryProtocol$InvokerDelegete<T>(InvokerWrapper<T>).invoke(Invocation) line: 56
ExceptionFilter.invoke(Invoker<?>, Invocation) line: 62
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
MonitorFilter.invoke(Invoker<?>, Invocation) line: 75
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
TimeoutFilter.invoke(Invoker<?>, Invocation) line: 42
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
TraceFilter.invoke(Invoker<?>, Invocation) line: 78
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
ContextFilter.invoke(Invoker<?>, Invocation) line: 73
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
GenericFilter.invoke(Invoker<?>, Invocation) line: 138
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
ClassLoaderFilter.invoke(Invoker<?>, Invocation) line: 38
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
EchoFilter.invoke(Invoker<?>, Invocation) line: 38
ProtocolFilterWrapper$1.invoke(Invocation) line: 72
DubboProtocol$1.reply(ExchangeChannel, Object) line: 104
HeaderExchangeHandler.handleRequest(ExchangeChannel, Request) line: 96
HeaderExchangeHandler.received(Channel, Object) line: 173
DecodeHandler.received(Channel, Object) line: 51
ChannelEventRunnable.run() line: 57
ThreadPoolExecutor.runWorker(ThreadPoolExecutor$Worker) line: 1142
ThreadPoolExecutor$Worker.run() line: 617
InternalThread(Thread).run() line: 745
接收请求的过程





5.2 xml标签的解析

5.3 注解方式的生效过程