

Dubbo

一,Dubbo概述

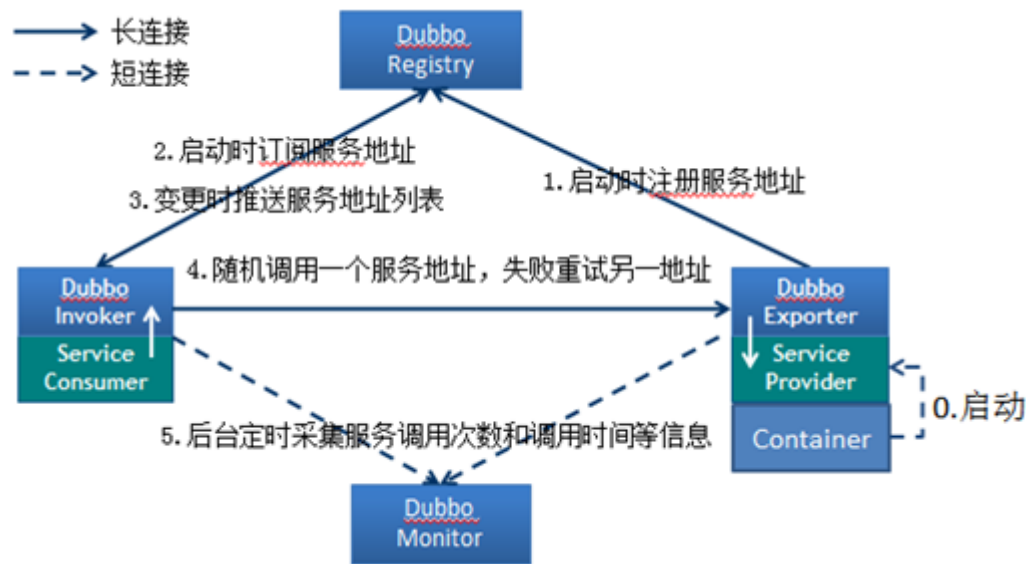
1.什么是Dubbo

Dubbo是 阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和 Spring框架无缝集成。

Dubbo是一款高性能、轻量级的开源Java RPC框架，它提供了三大核心能力：面向接口的远程方法调用，智能容错和负载均衡，以及服务自动注册和发现。

官网: <http://dubbo.apache.org/zh-cn/>

2.Dubbo架构



2.1节点角色

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

2.2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。


二,Dubbo的使用和配置

1.环境的搭建


1.1window环境的搭建

1.1.1安装zookeeper

- 下载zookeeper 网址 <https://archive.apache.org/dist/zookeeper/zookeeper-3.4.13/>

 zookeeper-3.4.11.tar.gz


- 解压zookeeper
- 修改zoo.cfg配置文件,将conf下的zoo_sample.cfg复制一份改名为zoo.cfg,需要配置:
 - dataDir=./ 临时数据存储的目录（可写相对路径）
 - clientPort=2181 zookeeper的端口号
- 进入bin目录,运行zkServer.cmd

 zkServer.cmd

1.1.2安装dubbo-admin管理控制台

为了让用户更好的管理监控众多的dubbo服务，官方提供了一个可视化的监控程序，不过这个监控即使不装也不影响使用。

- 下载dubbo-admin 网址: <https://github.com/apache/incubator-dubbo-ops>

 incubator-dubbo-ops-master.zip

- 解压,修改dubbo-admin配置. 修改 src\main\resources\application.properties 指定zookeeper地址(默认就是本地的)

```
24 spring.root.password=root
25 spring.guest.password=guest
26
27 dubbo.registry.address=zookeeper://127.0.0.1:2181
```

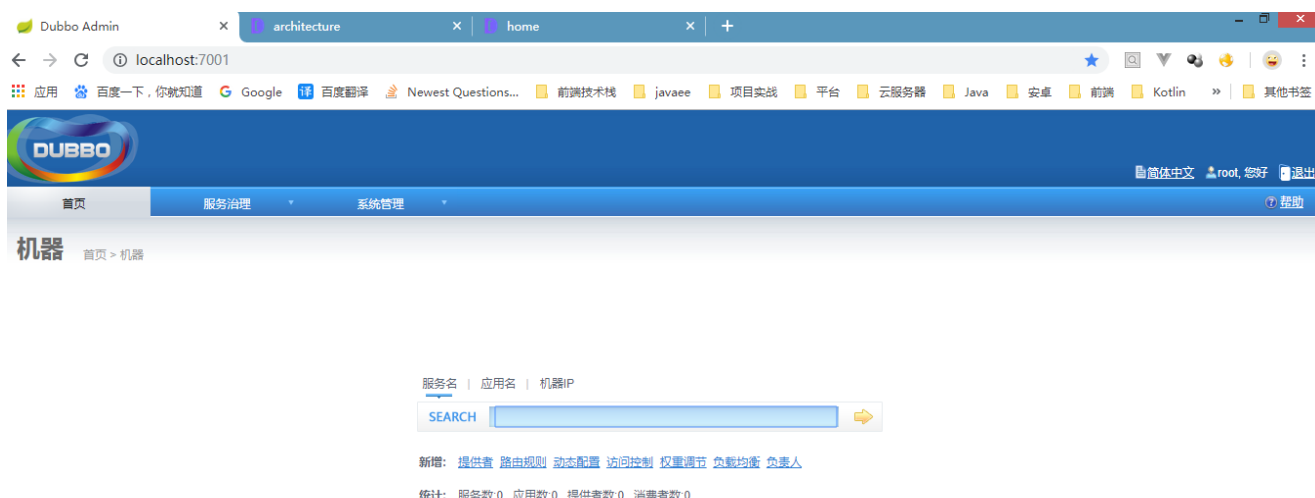
- 通过Maven打包, 进入dubbo-admin目录执行

```
mvn clean package -Dmaven.test.skip=true
```

- 运行dubbo-admin

```
java -jar dubbo-admin-0.0.1-SNAPSHOT.jar
```

- 通过浏览器访问 <http://localhost:7001/> 注意：【有可能控制台看着启动了，但是网页打不开，需要在控制台按下ctrl+c即可】，默认使用root/root 登陆



1.2Linux环境的搭建

1.2.1安装zookeeper

- 安装jdk（此步省略，我给大家提供的镜像已经安装好JDK）
- 把 zookeeper 的压缩包（资源\配套软件\dubbox\zookeeper-3.4.6.tar.gz）上传到 linux 系统
Alt+P 进入SFTP，输入put d:\zookeeper-3.4.6.tar.gz 上传
- 解压缩压缩包

```
tar -zxvf zookeeper-3.4.6.tar.gz
```

- 进入 zookeeper-3.4.6 目录，创建 data 文件夹

```
mkdir data
```

- 进入conf目录，把 zoo_sample.cfg 改名为 zoo.cfg

```
cd conf
mv zoo_sample.cfg zoo.cfg
```

- 打开zoo.cfg，修改 data 属性：dataDir=/root/zookeeper-3.4.6/data
- 进入bin目录，启动服务输入命令

```
./zkServer.sh start
```

```
JMX enabled by default
Using config: /root/zookeeper-3.4.6/bin/../conf/zoo.cfg
starting zookeeper ... STARTED
```

1.2.2安装dubbo-admin管理控制台

- 和window版本类似

Spring整合Dubbo的使用	功能	描述
------------------	----	----

2.1 需求

模块	功能
订单模块 (war)	创建订单等
用户模块 (war)	查询用户地址等

下订单时候,需要获得用户的地址. 订单模块和用户模块发布在不同的服务器里面

工程	描述
dubbo-pojo(jar)	实体类
dubbo-interface(jar)	接口
dubbo-user(war) 生产者	用户模块
dubbo-order(war) 消费者	订单模块

2.2 工程的创建

2.2.1 dubbo-pojo(jar)

- Address.java

```
public class Address implements Serializable {  
  
    private Integer id;//地址Id  
    private String address; //用户地址  
    private String userId; //用户id  
    private String consignee; //收货人  
    private String phone; //电话号码  
  
}
```

2.2.2 dubbo-interface(jar)

- pom.xml

```
<dependencies>  
  <dependency>  
    <groupId>com.itheima</groupId>  
    <artifactId>dubbo-pojo</artifactId>  
    <version>1.0-SNAPSHOT</version>  
  </dependency>  
</dependencies>
```

- OrderService.java

```
public interface OrderService {  
    /**  
     * 下订单  
     */  
    void saveOrder();  
}
```

- UserService.java

```
public interface UserService {  
  
    /**  
     * 按照用户id返回所有的收货地址  
     * @param userId  
     * @return  
     */  
    List<Address> findUserAddressList(String userId);  
  
}
```

2.2.3dubbo-user(war) 提供者

- pom.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itheima</groupId>
    <artifactId>dubbo-user</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>com.itheima</groupId>
            <artifactId>dubbo-interface</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>

        <!-- 引入dubbo2.6.2已经导入了Spring-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>dubbo</artifactId>
            <version>2.6.2</version>
        </dependency>
        <!-- 由于我们使用zookeeper作为注册中心，所以需要操作zookeeper
            dubbo 2.6以前的版本引入zkclient操作zookeeper
            dubbo 2.6及以后的版本引入curator操作zookeeper
        -->
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-framework</artifactId>
            <version>2.12.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>4.3.16.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.3.16.RELEASE</version>
        </dependency>
    </dependencies>
</project>

```

- UserServiceImpl.java

```
public class UserServiceImpl implements UserService {  
    /**  
     * 按照用户id返回所有的收货地址  
     * @param userId  
     * @return  
     */  
    public List<Address> findUserAddressList(String userId) {  
        //模拟dao查询数据库  
        List<Address> list = new ArrayList<Address>();  
        list.add(new Address(1,"北京","1","张三","12306"));  
        list.add(new Address(2,"武汉","2","李四","18170"));  
        return list;  
    }  
}
```

- 创建Spring的配置文件 provider.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"  
    xmlns:context="http://www.springframework.org/schema/context"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
http://www.springframework.org/schema/beans/spring-beans.xsd http://dubbo.apache.org/schema/dubbo  
http://dubbo.apache.org/schema/dubbo/dubbo.xsd http://www.springframework.org/schema/context  
http://www.springframework.org/schema/context/spring-context.xsd">  
  
    <bean id="userService" class="com.itheima.user.service.impl.UserServiceImpl"></bean>  
  
    <!--应用名-->  
    <dubbo:application name="dubbo-user"></dubbo:application>  
    <!--注册中心-->  
    <dubbo:registry address="zookeeper://127.0.0.1:2181" />  
    <!--使用dubbo协议, 将服务暴露在20880端口 -->  
    <dubbo:protocol name="dubbo" port="20880" />  
    <!-- 【xml版本】 -->  
    <!-- 指定需要暴露的服务 -->  
    <dubbo:service interface="com.itheima.service.UserService" ref="userService" />  
  
    <!-- 【注解版本】 -->  
    <!--扫描服务,在业务类上面添加@Service(dubbo的)-->  
    <!--<dubbo:annotation package="com.itheima.user.service.impl"/>-->  
  
</beans>
```

- web.xml配置监听器,加载配置文件

```
<!-- 加载spring容器 -->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:provider.xml</param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

2.2.4dubbo-order(war) 消费者

- pom.xml


```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itheima</groupId>
    <artifactId>dubbo-order</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>

        <dependency>
            <groupId>com.itheima</groupId>
            <artifactId>dubbo-interface</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>

        <!-- 引入dubbo2.6.2已经导入了Spring-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>dubbo</artifactId>
            <version>2.6.2</version>
        </dependency>
        <!-- 由于我们使用zookeeper作为注册中心，所以需要操作zookeeper
            dubbo 2.6以前的版本引入zkclient操作zookeeper
            dubbo 2.6及以后的版本引入curator操作zookeeper
        -->
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-framework</artifactId>
            <version>2.12.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>4.3.16.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.3.16.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

```

    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.0</version>
      <scope>provided</scope>
    </dependency>

  </dependencies>
</project>

```

- OrderController

```

@Controller
@RequestMapping("/order")
public class OrderController {
    @Autowired
    private UserService userService;

    @RequestMapping("/save")
    public String save(){
        List<Address> list = userService.findUserAddressList("1");
        System.out.println(list);
        //调用业务...
        return "success";
    }
}

```

- 创建spring的配置文件 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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd http://dubbo.apache.org/schema/dubbo
http://dubbo.apache.org/schema/dubbo/dubbo.xsd http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>

    <context:component-scan base-package="com.itheima.order"></context:component-scan>
    <!--应用名-->
    <dubbo:application name="dubbo-order"></dubbo:application>
    <!--注册中心-->
    <dubbo:registry address="zookeeper://127.0.0.1:2181"/>

    <!--【xml版本】-->
    <!-- 生成远程服务代理,使用AutoWired注入 -->
    <dubbo:reference id="userService" interface="com.itheima.service.UserService">
</dubbo:reference>

    <!--【注解版本】-->
    <!--扫描服务,使用@Reference注入-->
    <!--<dubbo:annotation package="com.itheima.order.controller"/>-->
</beans>

```

- web.xml配置

```

<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:consumer.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

```

2.3常见的配置

2.3.1重试次数

- 失败自动切换，当出现失败，重试其它服务器，但重试会带来更长延迟。可通过 `retries="2"` 来设置重试次数 (不含第一次)。

```
//提供方配置
<dubbo:service retries="2" />
//消费方配置
<dubbo:reference retries="2" />
```

2.3.2 超时时间

由于网络或服务端不可靠，会导致调用出现一种不确定的中间状态（超时）。为了避免超时导致客户端资源（线程）挂起耗尽，必须设置超时时间。我们可以消费者 配置也可以在提供者配置

- Dubbo消费者

```
全局超时配置
<dubbo:consumer timeout="5000" />

指定接口以及特定方法超时配置
<dubbo:reference interface="com.foo.BarService" timeout="2000">
  <dubbo:method name="sayHello" timeout="3000" />
</dubbo:reference>
```

- Dubbo提供者

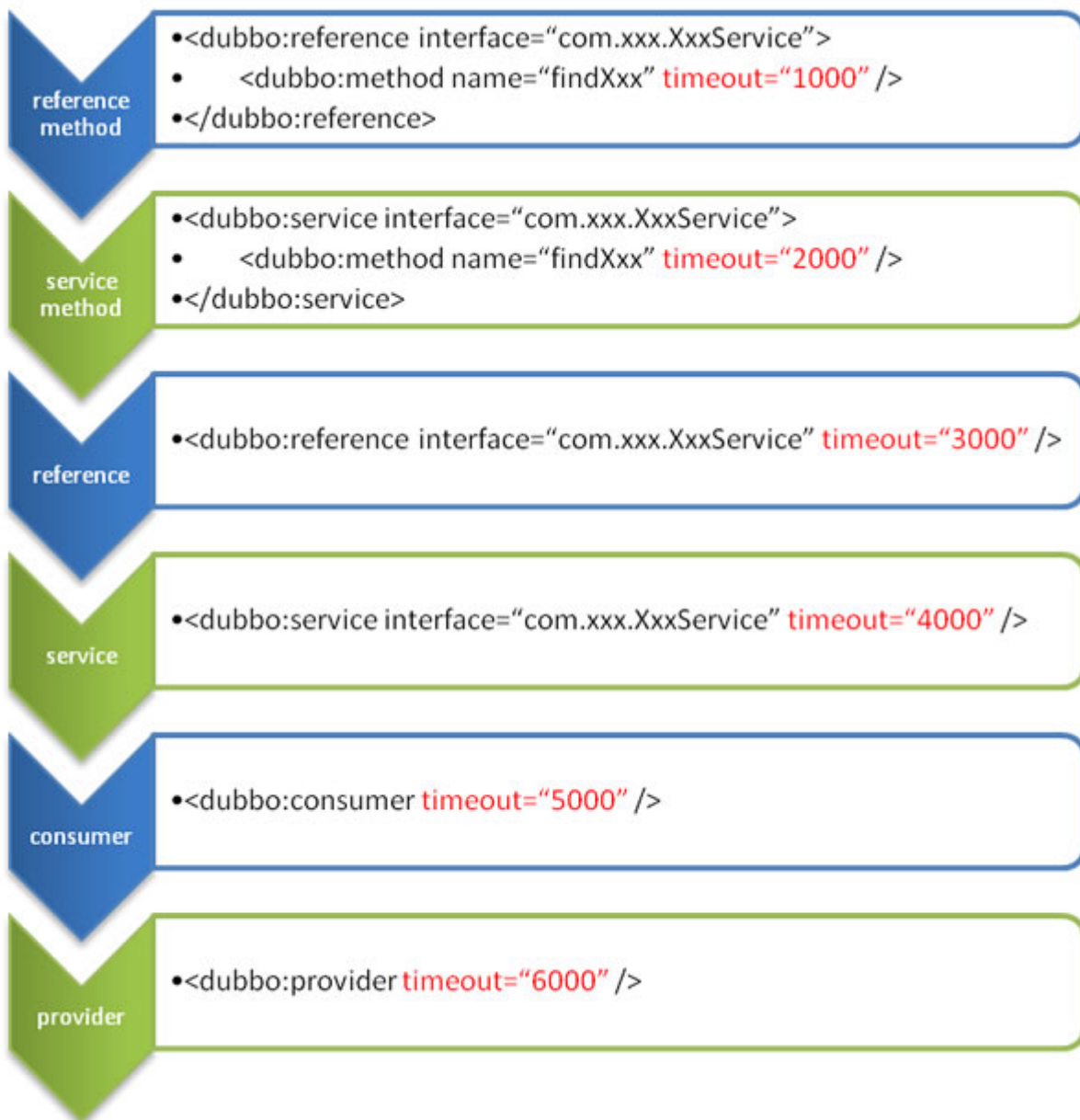
```
全局超时配置
<dubbo:provider timeout="5000" />

指定接口以及特定方法超时配置
<dubbo:provider interface="com.foo.BarService" timeout="2000">
  <dubbo:method name="sayHello" timeout="3000" />
</dubbo:provider>
```

2.3.3 配置的优先级

以 `timeout` 为例，显示了配置的查找顺序，其它 `retries`, `loadbalance`, `actives` 等类似：

- 方法级优先，接口级次之，全局配置再次之。
- 如果级别一样，则消费方优先，提供方次之。



3.SpringBoot整合Dubbo的使用

3.1版本对应关系

versions	Java	Spring Boot	Dubbo
0.2.0	1.8+	2.0.x	2.6.2 +
0.1.1	1.7+	1.5.x	2.6.2 +

3.2工程的创建

3.2.1boot-dubbo-user(war) 提供者

- pom.xml

```

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.1.RELEASE</version>
  <relativePath />
</parent>

<dependencies>

  <dependency>
    <groupId>com.itheima</groupId>
    <artifactId>dubbo-interface</artifactId>
    <version>1.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>com.alibaba.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>0.2.0</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

```

- 创建配置文件application.yml

```

server:
  port: 9090
dubbo:
  application:
    name: boot-dubbo-user #服务的名字(不要重复)
  registry:
    address: 127.0.0.1:2181 #注册中心的地址
    protocol: zookeeper #注册中心的协议为zookeeper
  protocol:
    name: dubbo #协议

```

- 创建启动类(添加 @EnableDubbo)

```

@EnableDubbo
@SpringBootApplication
public class UserApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserApplication.class,args);
    }
}

```

- 创建UserServiceImpl(**@Service**需要用dubbo的)

```

@Service
public class UserServiceImpl implements UserService {
    /**
     * 按照用户id返回所有的收货地址
     * @param userId
     * @return
     */
    public List<Address> findUserAddressList(String userId) {
        //模拟dao查询数据库
        List<Address> list = new ArrayList<Address>();
        list.add(new Address(1,"北京","1","张三","12306"));
        list.add(new Address(2,"武汉","2","李四","18170"));
        return list;
    }
}

```

3.2.2boot-dubbo-order(war) 消费者

- pom.xml

```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.1.RELEASE</version>
  <relativePath />
</parent>

<dependencies>

  <dependency>
    <groupId>com.itheima</groupId>
    <artifactId>dubbo-interface</artifactId>
    <version>1.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>com.alibaba.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>0.2.0</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

```

- 创建配置文件application.yml

```

server:
  port: 8080
dubbo:
  application:
    name: boot-dubbo-order #服务的名字(不要重复)
  registry:
    address: 127.0.0.1:2181 #注册中心的地址
    protocol: zookeeper #注册中心的协议为zookeeper
  protocol:
    name: dubbo #协议

```

- 创建启动类(添加 @EnableDubbo)


```

@EnableDubbo
@SpringBootApplication
public class OrderApplication {

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

}

```

- 创建Controller

```

@RestController
@RequestMapping("/order")
public class OrderController {
    @Reference
    private UserService userService;
    @RequestMapping("/save")
    public List<Address> save(){
        List<Address> list = userService.findUserAddressList("1");
        System.out.println(list);
        //调用业务...
        return list;
    }
}

```

三,Dubbo负载均衡,容错,高可用

1.负载均衡

1.1负载均衡概述

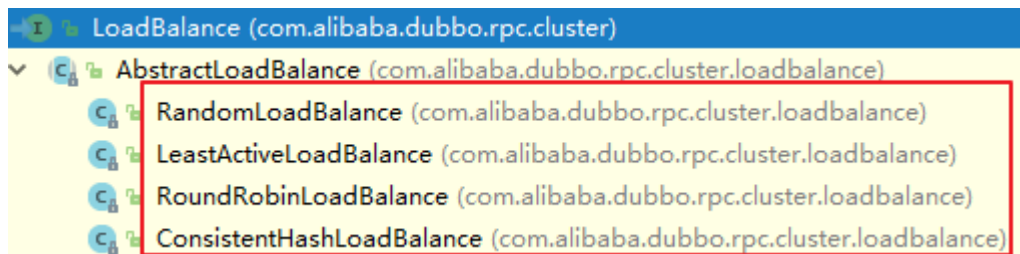
LoadBalance 中文意思为负载均衡，它的职责是将网络请求，或者其他形式的负载“均摊”到不同的机器上。避免集群中部分服务器压力过大，而另一些服务器比较空闲的情况。通过负载均衡，可以让每台服务器获取到适合自己处理能力的负载。在为高负载服务器分流的同时，还可以避免资源浪费，一举两得。

负载均衡可分为软件负载均衡和硬件负载均衡。在我们日常开发中，一般很难接触到硬件负载均衡。但软件负载均衡还是可以接触到的，比如 Nginx。在 Dubbo 中，也有负载均衡的概念和相应的实现。Dubbo 需要对服务消费者的调用请求进行分配，避免少数服务提供者负载过大。服务提供者负载过大，会导致部分请求超时。因此将负载均衡到每个服务提供者上，是非常必要的。

文档: <http://dubbo.apache.org/zh-cn/docs/user/demos/loadbalance.html>

http://dubbo.apache.org/zh-cn/docs/source_code_guide/loadbalance.html

1.2Dubbo中负载均衡策略



- Random LoadBalance(默认)

随机，按权重设置随机概率。

在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

- RoundRobin LoadBalance

轮询，按公约后的权重设置轮询比率。

存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

- LeastActive LoadBalance

最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。

使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

- ConsistentHash LoadBalance

一致性 Hash，相同参数的请求总是发到同一提供者。

当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

缺省只对第一个参数 Hash，如果要修改，请配置

缺省用 160 份虚拟节点，如果要修改，请配置

1.3配置

1.3.1xml方式

我们可以在提供方配置也可以在消费方配置. 有如下几种,任选一种

- 服务端服务级别

```
<dubbo:service interface="..." loadbalance="roundrobin" />
```

- 服务端方法级别

```
<dubbo:service interface="...">
  <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:service>
```

- 客户端服务级别

```
<dubbo:reference interface="..." loadbalance="roundrobin" />
```

- 客户端方法级别

```
<dubbo:reference interface="...">
  <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:reference>
```

1.3.2 注解方式

- 提供者配置

```
@Service(loadbalance = "")
public class UserServiceImpl implements UserService {}
```

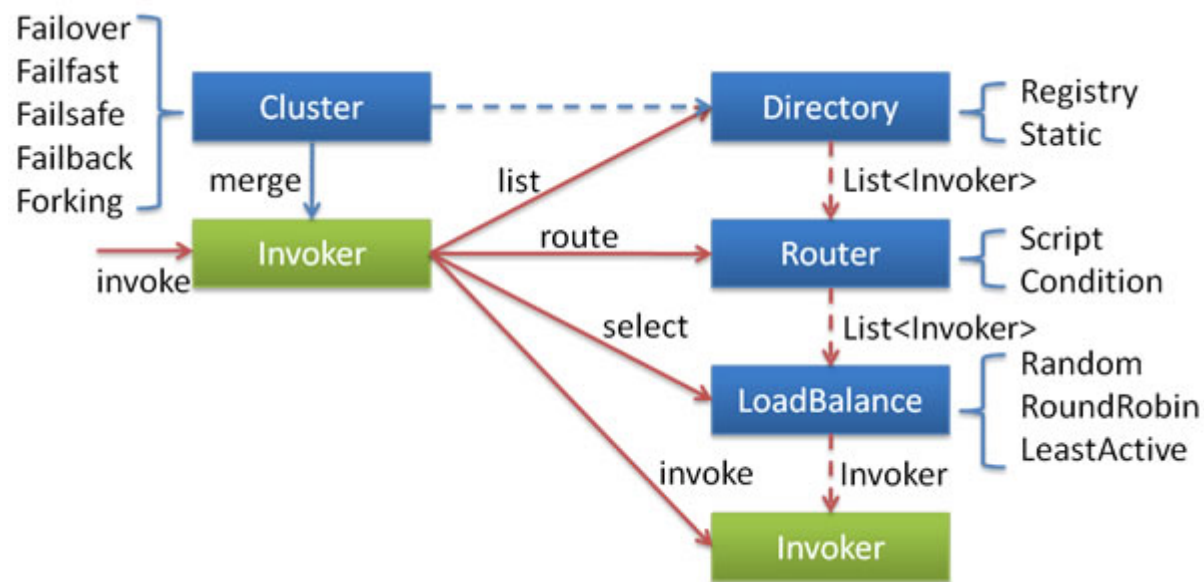
- 消费者配置,通过loadbalance属性

```
@Reference(loadbalance = "roundrobin ")
private UserService userService;
```

2. 集群容错

2.1 容错概述

在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。



各节点关系：

- 这里的 **Invoker** 是 **Provider** 的一个可调用 **Service** 的抽象，**Invoker** 封装了 **Provider** 地址及 **Service** 接口信息
- Directory** 代表多个 **Invoker**，可以把它看成 **List**，但与 **List** 不同的是，它的值可能是动态变化的，比如注册中心推送变更
- 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，比如读写分离，应用隔离等
- LoadBalance** 负责从多个 **Invoker** 中选出具体的一个用于本次调用，选的过程包含了负载均衡算法，调用失败后，需要重选

2.2Dubbo中容错策略

- Failover Cluster

失败自动切换，当出现失败，重试其它服务器. 通常用于读操作，但重试会带来更长延迟。可通过 `retries="2"` 来设置重试次数(不含第一次)。可以在提供方配置也可以在消费方配置

```
//提供方配置
<dubbo:service retries="2" />
//消费方配置
<dubbo:reference retries="2" />
```

- Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

- Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

- Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

- Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 `forks="2"` 来设置最大并行数。

- Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错 [2]。通常用于通知所有提供者更新缓存或日志等本地资源信息。

2.3配置

2.3.1xml方式

- 服务提供方

```
<dubbo:service cluster="failsafe" />
```

- 服务消费方

```
<dubbo:reference cluster="failsafe" />
```

2.3.2注解方式

- 服务提供方

```
@Service(cluster = "failsafe")
```

- 服务消费方

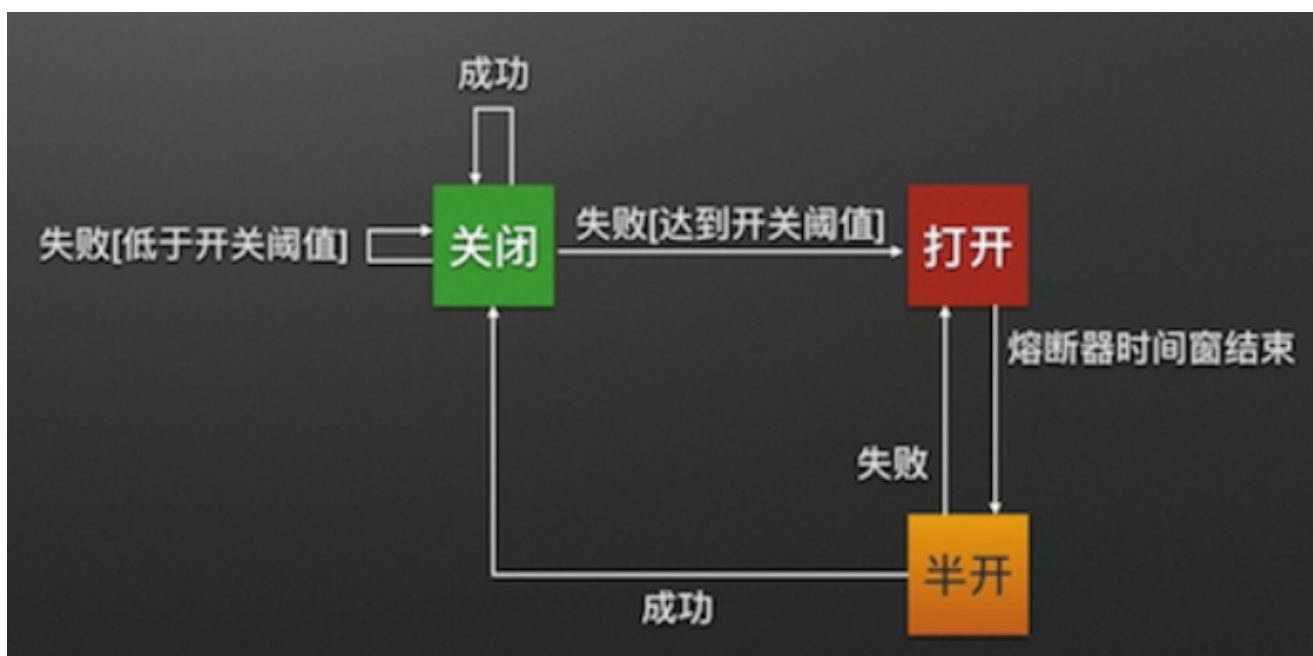
```
@Reference(cluster = "failsafe")
```

3.整合熔断器Hystrix

3.1Hystrix概述

Hystrix是一个用于处理分布式系统的延迟和容错的开源库，Hystrix 能使你的系统在出现依赖服务失效的时候，通过隔离系统所依赖的服务，防止服务级联失败，同时提供失败回退机制，更优雅地应对失效，并使你的系统能更快地从异常中恢复。

“断路器”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要地占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。



3.2整合Hystrix进行容错

3.2.1提供方

- 在pom.xml添加Hystrix起步依赖

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Finchley.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

```

- 在启动类上面开启Hystrix(@EnableHystrix)

```

@EnableHystrix
@EnableDubbo
@SpringBootApplication
public class UserApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserApplication.class,args);
    }
}

```

- 在提供的方法上面添加注解@HystrixCommand

```

@Service
public class UserServiceImpl implements UserService {
    /**
     * 按照用户id返回所有的收货地址
     * @param userId
     * @return
     */
    @HystrixCommand
    public List<Address> findUserAddressList(String userId) {
        //模拟dao查询数据库
        System.out.println("UserServiceImpl...");
        List<Address> list = new ArrayList<Address>();
        list.add(new Address(1,"北京","1","张三","12306"));
        list.add(new Address(2,"武汉","2","李四","18170"));
        return list;
    }
}

```

3.2.2消费者

- 在pom.xml添加Hystrix起步依赖

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Finchley.SR1</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

```

- 在启动类上面开启Hystrix(@EnableHystrix)

```

@EnableHystrix
@EnableDubbo
@SpringBootApplication
public class OrderApplication {

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

}

```

- 在调用的方法上面添加注解@HystrixCommand

```

@RestController
@RequestMapping("/order")
public class OrderController {
    @Reference(cluster = "")
    private UserService userService;

    @HystrixCommand(fallbackMethod = "nativeMethod")
    @RequestMapping("/save")
    public List<Address> save(){
        List<Address> list = userService.findUserAddressList("1");
        System.out.println(list);
        //调用业务...
        return list;
    }

    //当远程方法调用失败,会触发当前方法
    public List<Address> nativeMethod(){
        List<Address> list = new ArrayList<Address>();
        list.add(new Address(1,"默认地址","1","默认收货人","默认电话"));
        return list;
    }
}

```

4.Zookeeper集群

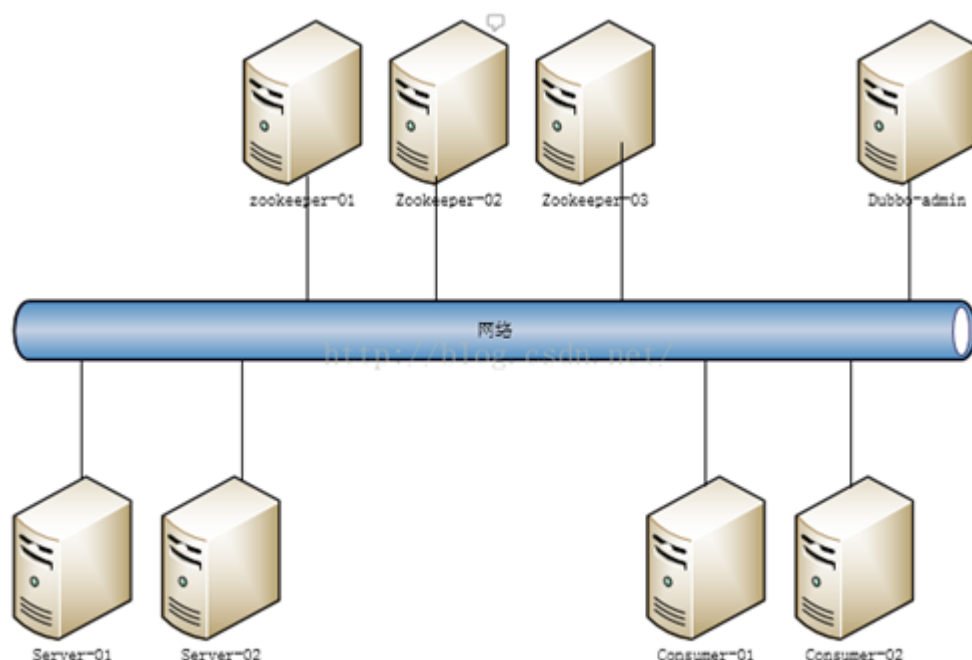
4.1.Zookeeper集群简介

4.1.1为什么搭建Zookeeper集群

大部分分布式应用需要一个主控、协调器或者控制器来管理物理分布的子进程。目前，大多数都要开发私有的协调程序，缺乏一个通用机制，协调程序的反复编写浪费，且难以形成通用、伸缩性好的协调器，zookeeper提供通用的分布式锁服务，用以协调分布式应用。所以说zookeeper是分布式应用的协作服务。

zookeeper作为注册中心，服务器和客户端都要访问，如果有大量的并发，肯定会有等待。所以可以通过zookeeper集群解决。

下面是zookeeper集群部署结构图：



4.1.2 Leader选举

Zookeeper的启动过程中leader选举是非常重要而且最复杂的一个环节。那么什么是leader选举呢？zookeeper为什么需要leader选举呢？zookeeper的leader选举的过程又是什么样子的？

首先我们来看看什么是leader选举。其实这个很好理解，leader选举就像总统选举一样，每人一票，获得多数票的人就当选为总统了。在zookeeper集群中也是一样，每个节点都会投票，如果某个节点获得超过半数以上的节点的投票，则该节点就是leader节点了。

以一个简单的例子来说明整个选举的过程。

假设有五台服务器组成的zookeeper集群,它们的id从1-5,同时它们都是最新启动的,也就是没有历史数据,在存放数据量这一点上,都是一样的.假设这些服务器依序启动,来看看会发生什么。

- 1) 服务器1启动,此时只有它一台服务器启动了,它发出去的报没有任何响应,所以它的选举状态一直是LOOKING状态
- 2) 服务器2启动,它与最开始启动的服务器1进行通信,互相交换自己的选举结果,由于两者都没有历史数据,所以id值较大的服务器2胜出,但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3),所以服务器1,2还是继续保持LOOKING状态。
- 3) 服务器3启动,根据前面的理论分析,服务器3成为服务器1,2,3中的老大,而与上面不同的是,此时有三台服务器选举了它,所以它成为了这次选举的leader。
- 4) 服务器4启动,根据前面的分析,理论上服务器4应该是服务器1,2,3,4中最大的,但是由于前面已经有半数以上的服务器选举了服务器3,所以它只能接收当小弟的命了。
- 5) 服务器5启动,同4一样,当小弟

4.2 搭建Zookeeper集群

4.2.1 搭建要求

真实的集群是需要部署在不同的服务器上的，但是在我们测试时同时启动十几个虚拟机内存会吃不消，所以我们通常会搭建伪集群，也就是把所有的服务都搭建在一台虚拟机上，用端口进行区分。

我们这里要求搭建一个三个节点的Zookeeper集群（伪集群）。

4.2.2 准备工作

(1) 安装JDK 【此步骤省略】。

(2) Zookeeper压缩包上传到服务器

(3) 将Zookeeper解压，创建data目录，将 conf下zoo_sample.cfg 文件改名为 zoo.cfg

(4) 建立/usr/local/zookeeper-cluster目录，将解压后的Zookeeper复制到以下三个目录

/usr/local/zookeeper-cluster/zookeeper-1

/usr/local/zookeeper-cluster/zookeeper-2

/usr/local/zookeeper-cluster/zookeeper-3

```
[root@localhost ~]# mkdir /usr/local/zookeeper-cluster
[root@localhost ~]# cp -r zookeeper-3.4.6 /usr/local/zookeeper-cluster/zookeeper-1
[root@localhost ~]# cp -r zookeeper-3.4.6 /usr/local/zookeeper-cluster/zookeeper-2
[root@localhost ~]# cp -r zookeeper-3.4.6 /usr/local/zookeeper-cluster/zookeeper-3
```

(5) 配置每一个Zookeeper 的dataDir (zoo.cfg) clientPort 分别为2181 2182 2183

修改/usr/local/zookeeper-cluster/zookeeper-1/conf/zoo.cfg

```
clientPort=2181
dataDir=/usr/local/zookeeper-cluster/zookeeper-1/data
```

修改/usr/local/zookeeper-cluster/zookeeper-2/conf/zoo.cfg

```
clientPort=2182
dataDir=/usr/local/zookeeper-cluster/zookeeper-2/data
```

修改/usr/local/zookeeper-cluster/zookeeper-3/conf/zoo.cfg

```
clientPort=2183
dataDir=/usr/local/zookeeper-cluster/zookeeper-3/data
```

4.2.3配置集群

(1) 在每个zookeeper的 data 目录下创建一个 myid 文件，内容分别是1、2、3。这个文件就是记录每个服务器的ID

-----知识点小贴士-----

如果你要创建的文本文件内容比较简单，我们可以通过echo 命令快速创建文件格式为：

echo 内容 >文件名

例如我们为第一个zookeeper指定ID为1，则输入命令

```
[root@localhost data]# echo 1 >myid
[root@localhost data]# cat myid
1
```

(2) 在每一个zookeeper 的 zoo.cfg配置客户端访问端口 (clientPort) 和集群服务器IP列表。

集群服务器IP列表如下

```
server.1=192.168.25.140:2881:3881
server.2=192.168.25.140:2882:3882
server.3=192.168.25.140:2883:3883
```

解释: server.服务器ID=服务器IP地址: 服务器之间通信端口: 服务器之间投票选举端口

4.2.4 启动集群

(1) 启动集群就是分别启动每个实例

```
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-1/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-1/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-2/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-2/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-3/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-3/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
```

(2) 启动后我们查询一下每个实例的运行状态

先查询第一个服务, Mode为follower表示是跟随者(从)

```
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-1/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-1/bin/../conf/zoo.cfg
Mode: follower
```

再查询第二个服务Mod为leader表示是领导者(主)

```
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-2/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-2/bin/../conf/zoo.cfg
Mode: leader
```

查询第三个为跟随者(从)

```
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-3/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-3/bin/../conf/zoo.cfg
Mode: follower
```

4.2.5 模拟集群异常

(1) 首先我们先测试如果是从服务器挂掉, 会怎么样

把3号服务器停掉, 观察1号和2号, 发现状态并没有变化

```
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-3/bin/zkServer.sh stop
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-3/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-2/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-2/bin/../conf/zoo.cfg
Mode: leader
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-1/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-1/bin/../conf/zoo.cfg
Mode: follower
```

由此得出结论, 3个节点的集群, 从服务器挂掉, 集群正常

(2) 我们再把1号服务器(从服务器)也停掉, 查看2号(主服务器)的状态, 发现已经停止运行了。

```
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-1/bin/zkServer.sh stop
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-1/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-2/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-2/bin/../conf/zoo.cfg
Error contacting service. It is probably not running.
```

由此得出结论，3个节点的集群，2个从服务器都挂掉，主服务器也无法运行。因为可运行的机器没有超过集群总数量的半数。

(3) 我们再次把1号服务器启动起来，发现2号服务器又开始正常工作了。而且依然是领导者。

```
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-1/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-1/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-2/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-2/bin/../conf/zoo.cfg
Mode: leader
```

(4) 我们把3号服务器也启动起来，把2号服务器停掉（汗~~干嘛？领导挂了？）停掉后观察1号和3号的状态。

```
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-2/bin/zkServer.sh stop
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-2/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-1/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-1/bin/../conf/zoo.cfg
Mode: follower
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-3/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-3/bin/../conf/zoo.cfg
Mode: leader
```

发现新的leader产生了~

由此我们得出结论，当集群中的主服务器挂了，集群中的其他服务器会自动进行选举状态，然后产生新得leader

(5) 我们再次测试，当我们把2号服务器重新启动起来（汗~~这是丧尸啊!）启动后，会发生什么？2号服务器会再次成为新的领导吗？我们看结果

```
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-2/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-2/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-2/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-2/bin/../conf/zoo.cfg
Mode: follower

[root@localhost ~]# /usr/local/zookeeper-cluster/zookeeper-3/bin/zkServer.sh status
JMX enabled by default
Using config: /usr/local/zookeeper-cluster/zookeeper-3/bin/../conf/zoo.cfg
Mode: leader
```

我们会发现，2号服务器启动后依然是跟随者（从服务器），3号服务器依然是领导者（主服务器），没有撼动3号服务器的领导地位。

由此我们得出结论，当领导者产生后，再次有新服务器加入集群，不会影响到现任领导者。

4.3.Dubbo连接zookeeper集群

- 修改服务提供者和服务调用者的spring 配置文件

```
<!-- 指定注册中心地址 -->
<dubbo:registry
    protocol="zookeeper" address="192.168.25.140:2181,192.168.25.140:2182,192.168.25.140:2183">
</dubbo:registry>
```

四,Dubbo原理

1.Netty

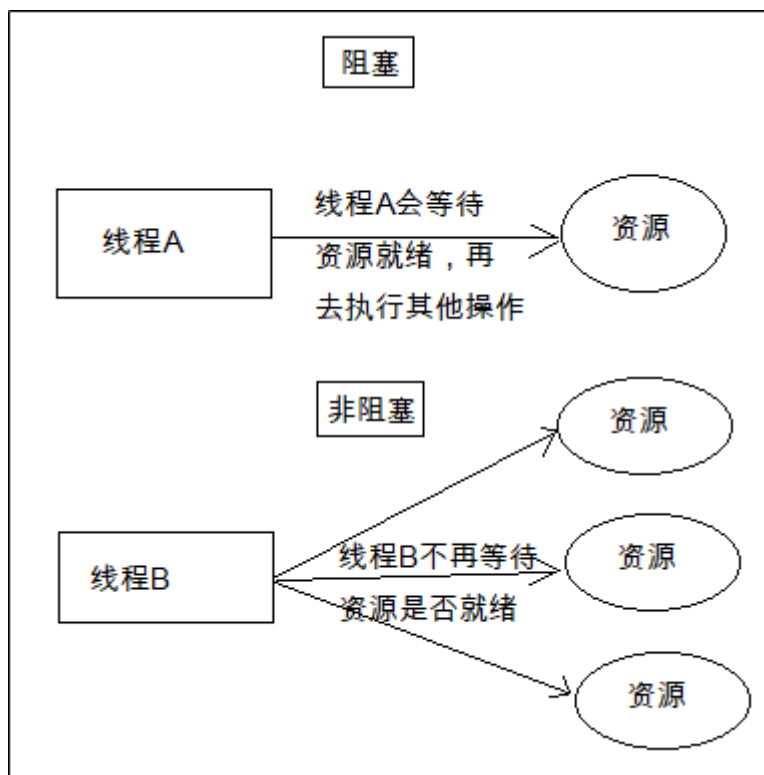
1.1Netty概述

Netty 是由 JBOSS 提供的一个 Java 开源框架。Netty 提供异步的、基于事件驱动的网络应用程序框架，用以快速开发高性能、高可靠性的网络 IO 程序。Netty 是一个基于 NIO 的网络编程框架，使用 Netty 可以帮助你快速、简单的开发出一个网络应用，相当于简化和流程化了 NIO 的开发过程。作为当前最流行的 NIO 框架，Netty 在互联网领域、大数据分布式计算领域、游戏行业、通信行业等获得了广泛的应用，知名的 Elasticsearch、Dubbo 框架内部都采用了 Netty。

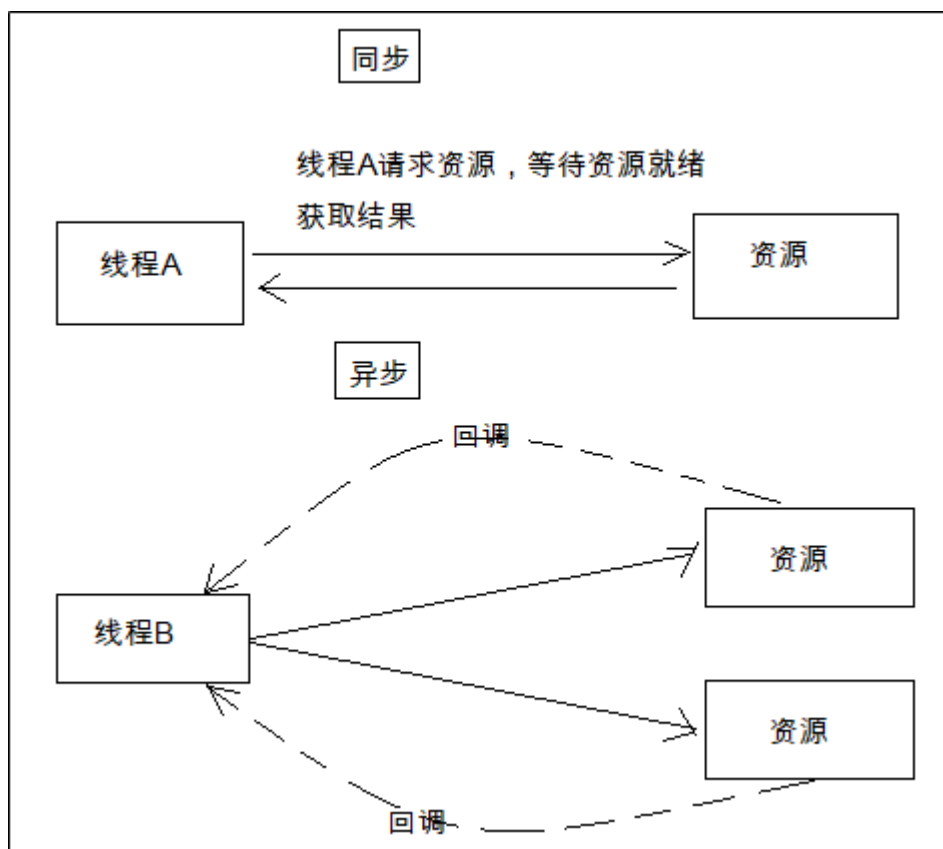
1.2BIO,NIO,AIO介绍与区别

1.2.1阻塞和非阻塞,同步和异步

- 阻塞和非阻塞(主要指的是访问IO的线程是否会阻塞（或者说是等待）线程访问资源，该资源是否准备就绪的一种处理方式。)



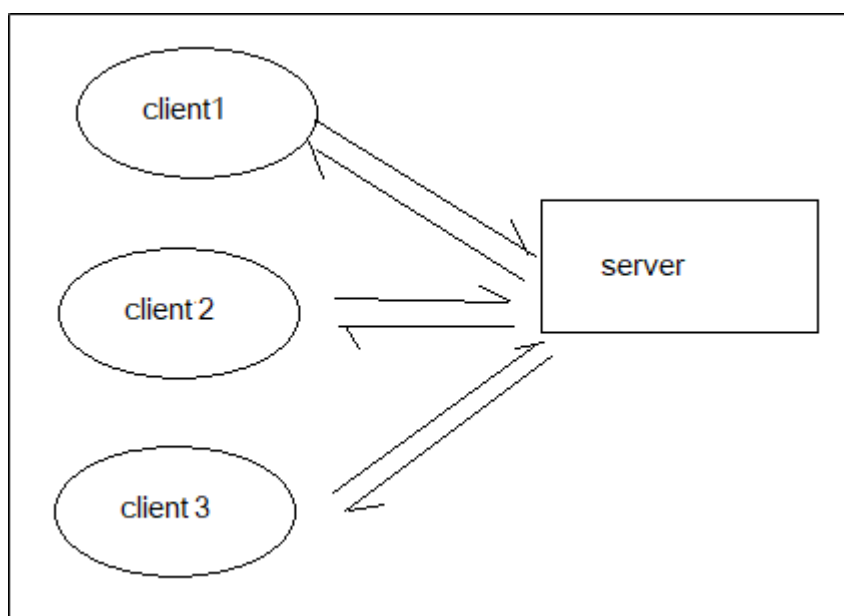
- 同步和异步(主要是指的数据的请求方式,同步和异步是指访问数据的一种机制)



1.2.2 BIO

同步阻塞IO，Block IO，IO操作时会阻塞线程，并发处理能力低。

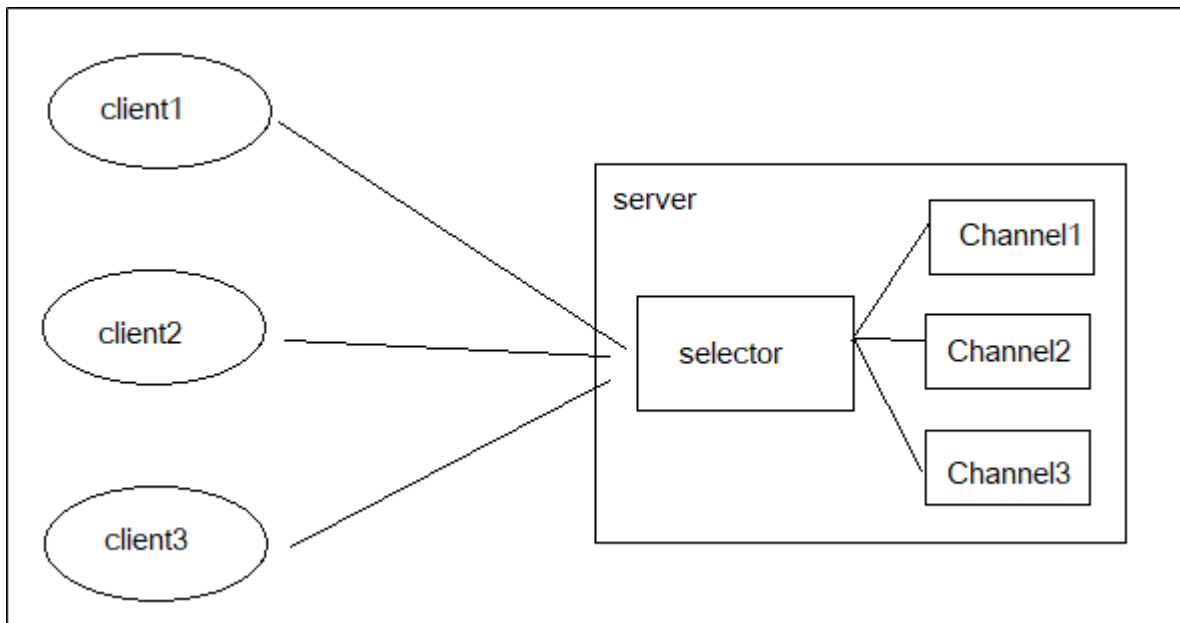
我们熟知的Socket编程就是BIO，一个socket连接一个处理线程（这个线程负责这个Socket连接的一系列数据传输操作）。阻塞的原因在于：操作系统允许的线程数量是有限的，多个socket申请与服务端建立连接时，服务端不能提供相应数量的处理线程，没有分配到处处理线程的连接就会阻塞等待或被拒绝



1.2.3 NIO

同步非阻塞IO，None-Block IO

NIO是对BIO的改进，基于Reactor模型。我们知道，一个socket连接只有在特点时候才会发生数据传输IO操作，大部分时间这个“数据通道”是空闲的，但还是占用着线程。NIO作出的改进就是“一个请求一个线程”，在连接到服务端的众多socket中，只有需要进行IO操作的才能获取服务端的处理线程进行IO。这样就不会因为线程不够用而限制了socket的接入。



1.2.4AIO

异步非阻塞IO

这种IO模型是由操作系统先完成了客户端请求处理再通知服务器去启动线程进行处理。AIO也称NIO2.0，在JDK7**开始支持。

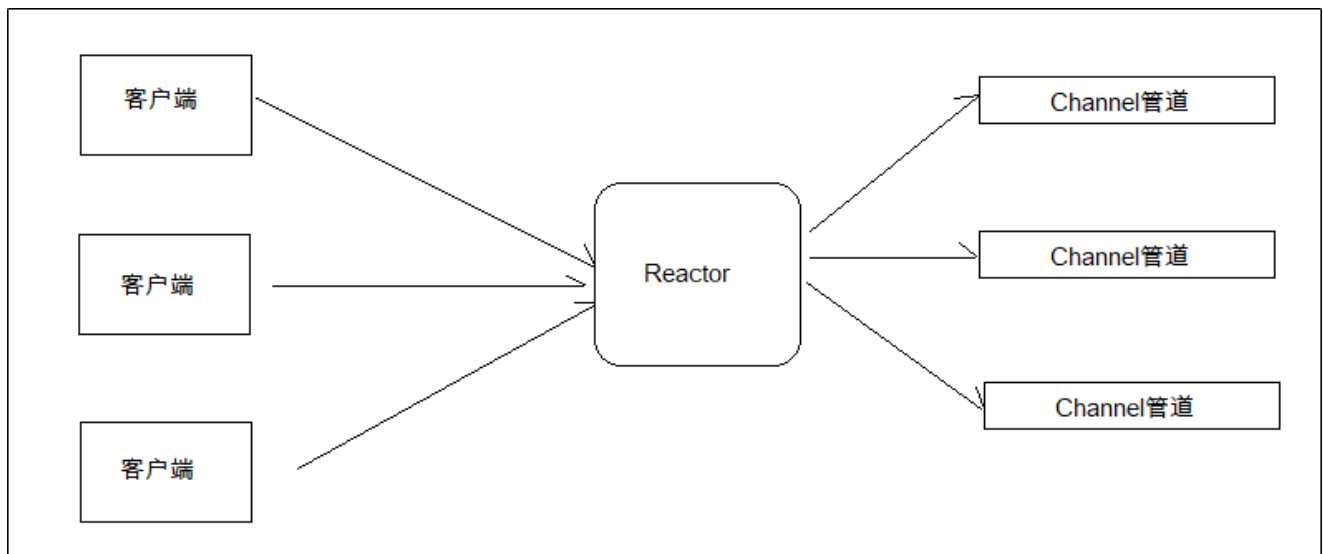
1.3NettyReactor模型

1.3.1单线程模型

用户发起IO请求到Reactor线程,Ractor线程将用户的IO请求放入到通道，然后再进行后续处理处理完成后，Reactor线程重新获得控制权，继续其他客户端的处理

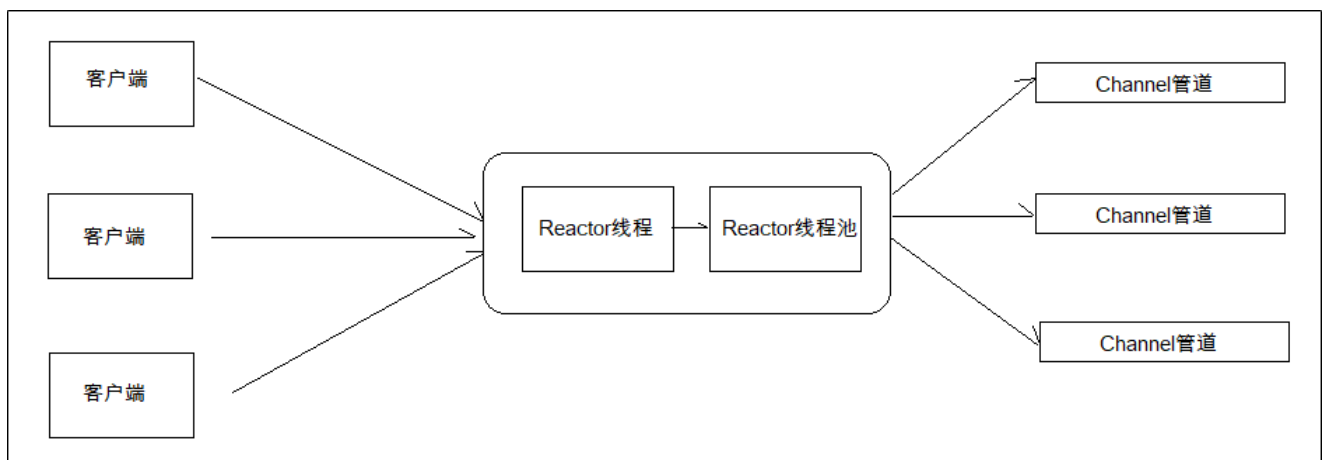
这种模型一个时间点只有一个任务在执行，这个任务执行完了，再去执行下一个任务。

1. 但单线程的Reactor模型每一个用户事件都在一个线程中执行：
2. 性能有极限，不能处理成百上千的事件
3. 当负荷达到一定程度时，性能将会下降
4. 某一个事件处理器发生故障，不能继续处理其他事件



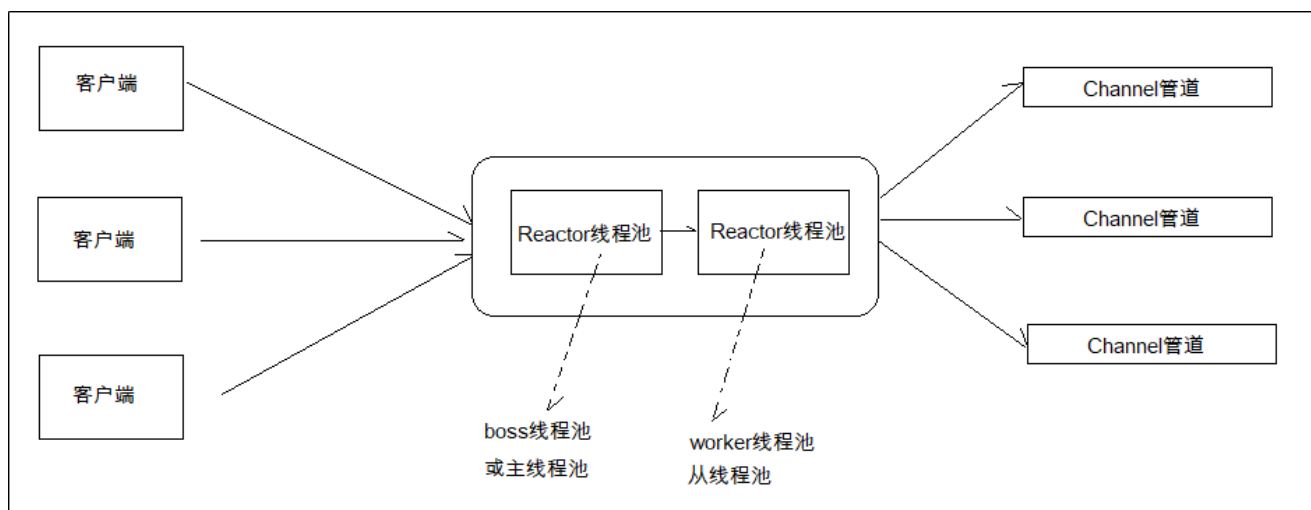
1.3.2 Reactor多线程模型

Reactor多线程模型是由一组NIO线程来处理IO操作（之前是单个线程），所以在请求处理上会比上一中模型效率更高，可以处理更多的客户端请求。这种模式使用多个线程执行多个任务，任务可以同时执行，但是如果并发仍然很大，Reactor仍然无法处理大量的客户端请求



1.3.3 Reactor主从多线程模型

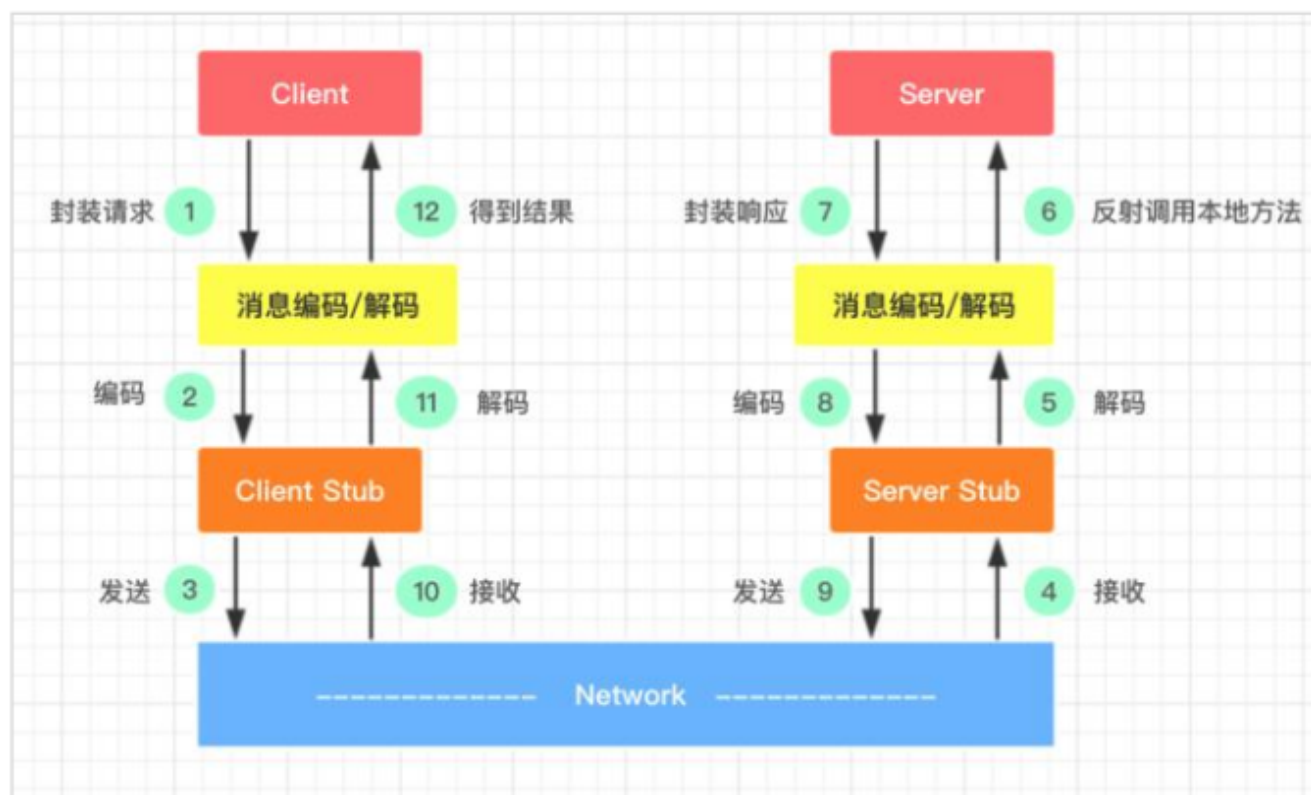
这种线程模型是Netty推荐的线程模型，这种模型适用于高并发场景，一组线程池接收请求，一组线程池处理IO。



2.RPC

2.1概述

RPC (Remote Procedure Call)，即远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络实现的技术。常见的RPC框架有: 源自阿里的Dubbo，Google出品的gRPC等等。



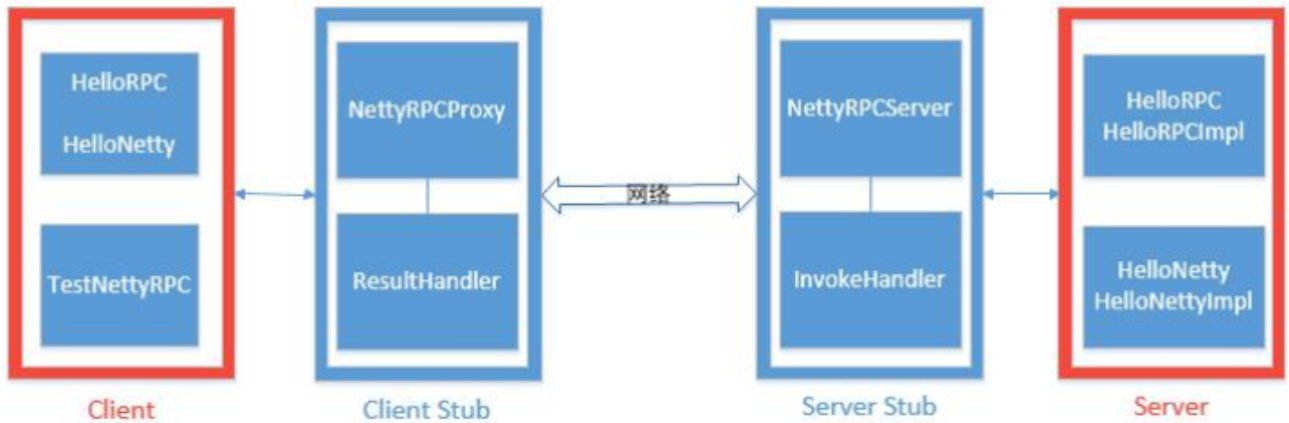
1. 服务消费方(client)以本地调用方式调用服务
2. client stub 接收到调用后负责将方法、参数等封装成能够进行网络传输的消息体
3. client stub 将消息进行编码并发送到服务端
4. server stub 收到消息后进行解码
5. server stub 根据解码结果调用本地的服务
6. 本地服务执行并将结果返回给 server stub
7. server stub 将返回导入结果进行编码并发送至消费方

- 8. client stub 接收到消息并进行解码
- 9. 服务消费方(client)得到结果

RPC 的目标就是将 2-8 这些步骤都封装起来， 用户无需关心这些细节， 可以像调用本地方法一样即可完成远程服务调用。

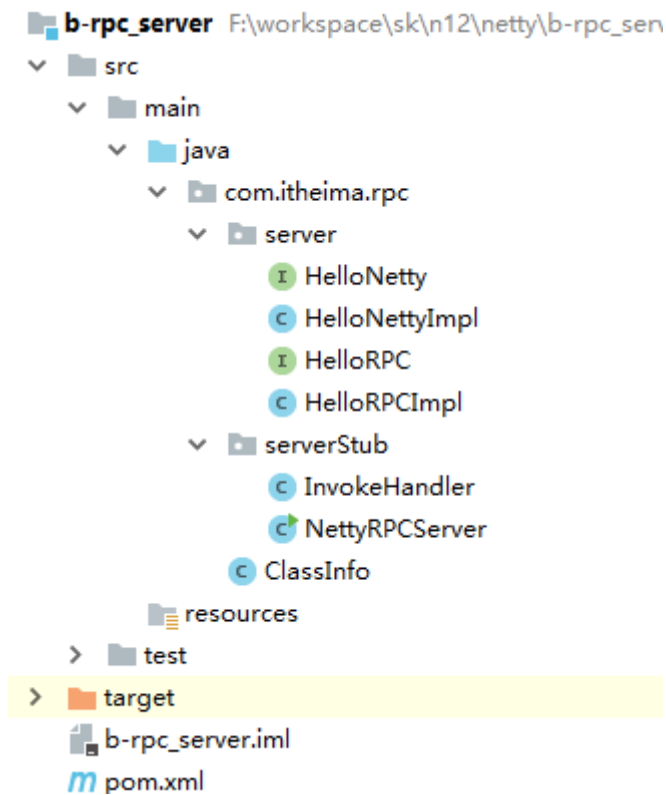
2.2自定义RPC

2.2.1结构设计

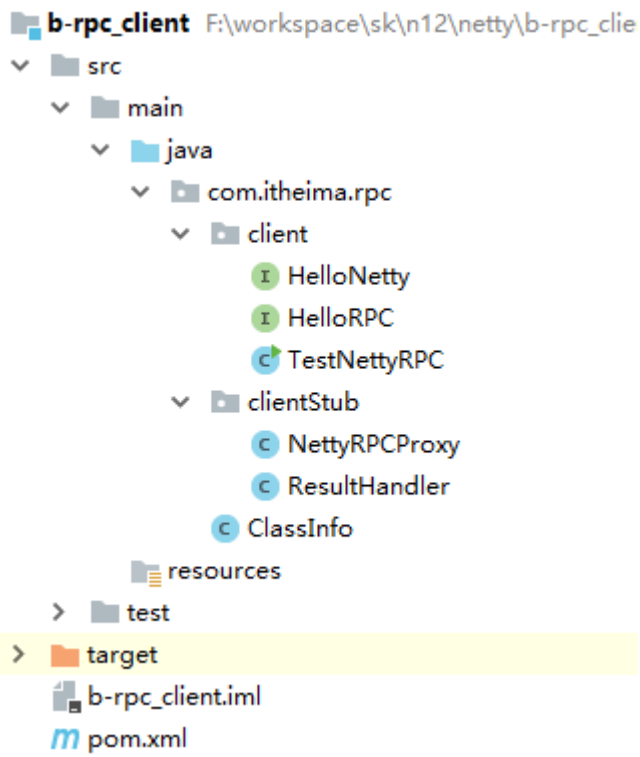


2.2.2代码实现

2.2.2.1 rpc_server



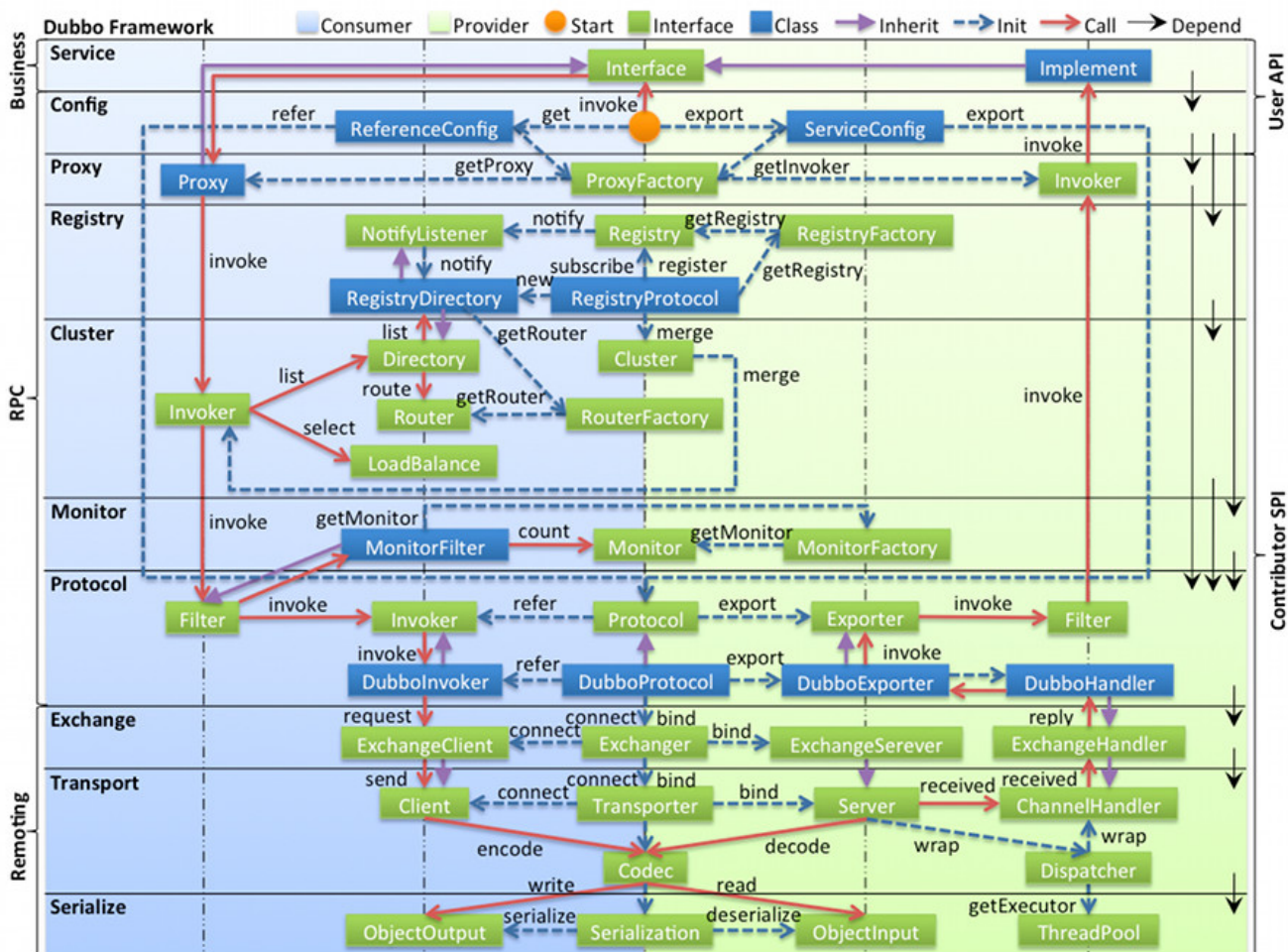
2.2.2.2 rpc_client



3.Dubbo原理

java中的SPI机制

3.1框架设计



- config 配置层：对外配置接口，以 ServiceConfig, ReferenceConfig 为中心，可以直接初始化配置类，也可以通过 spring 解析配置生成配置类
- proxy 服务代理层：服务接口透明代理，生成服务的客户端 Stub 和服务端 Skeleton, 以 ServiceProxy 为中心，扩展接口为 ProxyFactory
- registry 注册中心层：封装服务地址的注册与发现，以服务 URL 为中心，扩展接口为 RegistryFactory, Registry, RegistryService
- cluster 路由层：封装多个提供者的路由及负载均衡，并桥接注册中心，以 Invoker 为中心，扩展接口为 Cluster, Directory, Router, LoadBalance
- monitor 监控层：RPC 调用次数和调用时间监控，以 Statistics 为中心，扩展接口为 MonitorFactory, Monitor, MonitorService
- protocol 远程调用层：封装 RPC 调用，以 Invocation, Result 为中心，扩展接口为 Protocol, Invoker, Exporter
- exchange 信息交换层：封装请求响应模式，同步转异步，以 Request, Response 为中心，扩展接口为 Exchanger, ExchangeChannel, ExchangeClient, ExchangeServer
- transport 网络传输层：抽象 mina 和 netty 为统一接口，以 Message 为中心，扩展接口为 Channel, Transporter, Client, Server, Codec
- serialize 数据序列化层：可复用的一些工具，扩展接口为 Serialization, ObjectInput, ObjectOutput, ThreadPool

3.2 容器启动-配置文件的解析

在Spring中, 通过BeanDefinitionParser来解析标签. BeanDefinitionParser是一个接口, 当前解析我们的配置文件的是DubboBeanDefinitionParser.

在DubboBeanDefinitionParser解析之前, 会在DubboNamespaceHandler进行标签和类的对应绑定.

application ---> ApplicationConfig.class

registry ---> RegistryConfig.class

provider ---> ProviderConfig.class

consumer ---> ConsumerConfig.class

service ---> ServiceBean.class

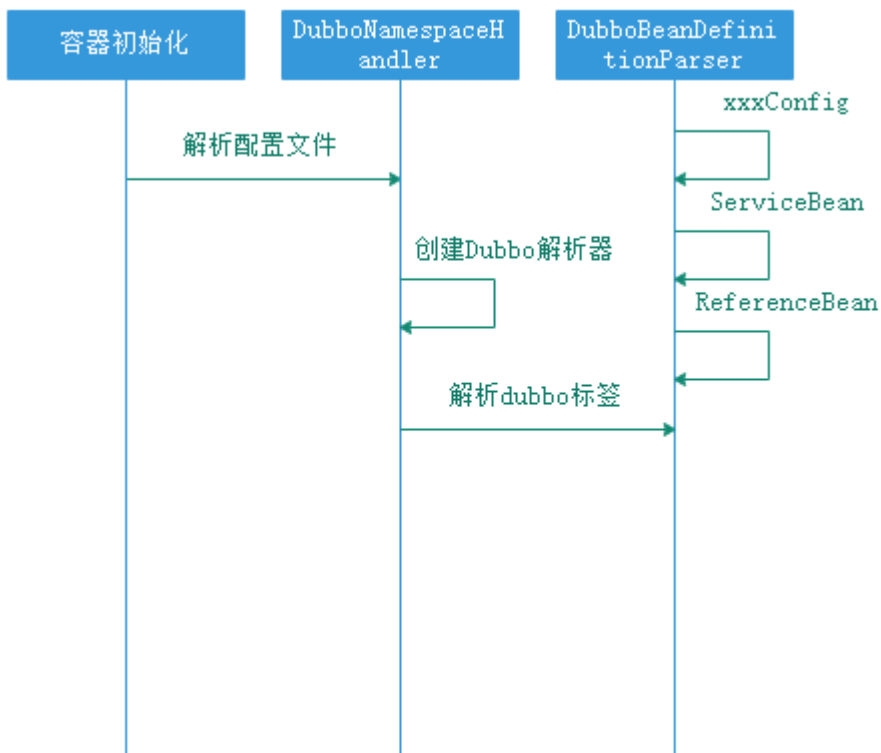
reference ---> ReferenceBean.class

- 配置文件

```
<!--应用名-->
<dubbo:application name="dubbo-user"></dubbo:application>
<!--注册中心-->
<dubbo:registry address="zookeeper://127.0.0.1:2181" />
<!--使用dubbo协议, 将服务暴露在20880端口 -->
<dubbo:protocol name="dubbo" port="20880" />
<!--【xml版本】-->
<!-- 指定需要暴露的服务 -->
<dubbo:service interface="com.itheima.service.UserService" ref="userService" />
```

- DubboNamespaceHandler的源码

```
@Override
public void init() {
    registerBeanDefinitionParser("application", new
DubboBeanDefinitionParser(ApplicationConfig.class, true));
    registerBeanDefinitionParser("module", new DubboBeanDefinitionParser(ModuleConfig.class,
true));
    registerBeanDefinitionParser("registry", new
DubboBeanDefinitionParser(RegistryConfig.class, true));
    registerBeanDefinitionParser("monitor", new
DubboBeanDefinitionParser(MonitorConfig.class, true));
    registerBeanDefinitionParser("provider", new
DubboBeanDefinitionParser(ProviderConfig.class, true));
    registerBeanDefinitionParser("consumer", new
DubboBeanDefinitionParser(ConsumerConfig.class, true));
    registerBeanDefinitionParser("protocol", new
DubboBeanDefinitionParser(ProtocolConfig.class, true));
    registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBean.class,
true));
    registerBeanDefinitionParser("reference", new
DubboBeanDefinitionParser(ReferenceBean.class, false));
    registerBeanDefinitionParser("annotation", new AnnotationBeanDefinitionParser());
}
```

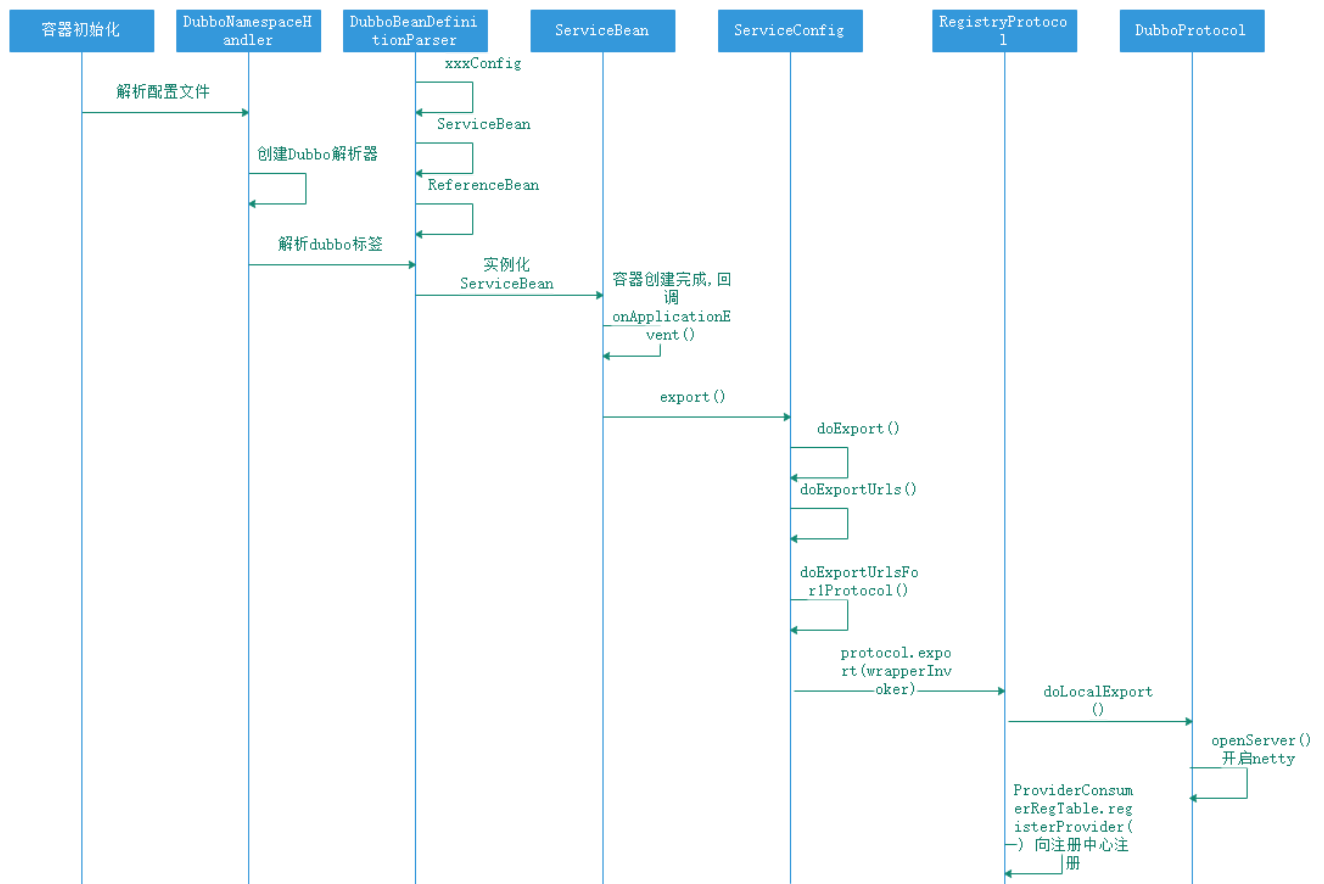


3.3服务暴露

在3.2中,我们了解到配置的 `<dubbo:service interface="com.itheima.service.UserService" ref="userService" />` 被解析成了对应的ServiceBean.class. ServiceBean实现了InitializingBean和ApplicationListener,所以当ServiceBean被实例化后会调用afterPropertiesSet(), 当容器创建完成之后会调用onApplicationEvent()方法

```

public class ServiceBean<T> extends ServiceConfig<T> implements InitializingBean,
ApplicationListener<ContextRefreshedEvent>.... {
    ...
}
  
```



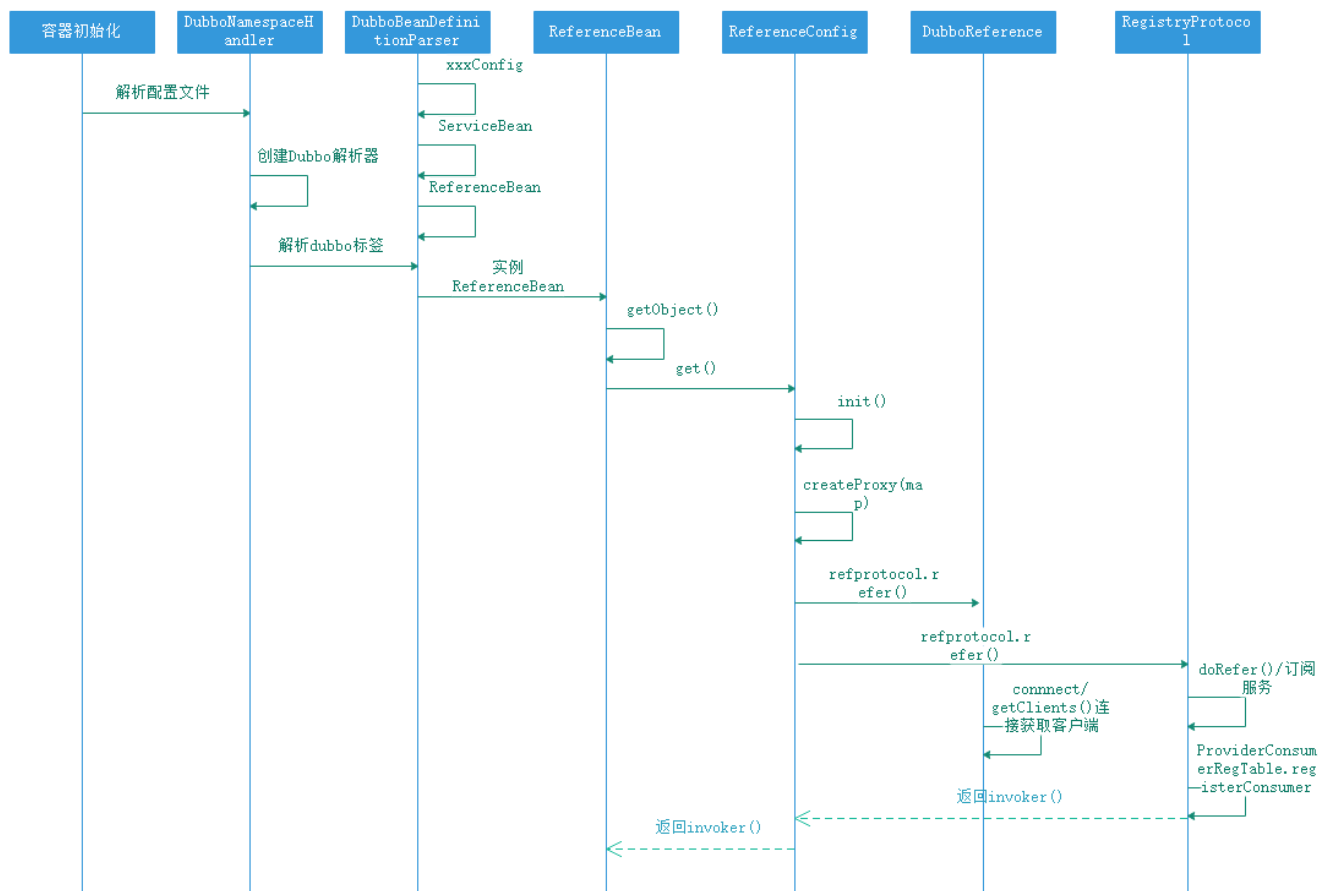
3.4服务引用

在3.2中,我们了解到配置的 `<dubbo:reference id="userService" interface="com.itheima.service.UserService"></dubbo:reference>` 被解析成了对应的 `ReferenceBean.class`. `ReferenceBean` 实现了 `FactoryBean`, 也就是说我们注入的 `userService` 要自动从容器获得, 并且是调用 `getObject()` 方法来获得的

```

public class ReferenceBean<T> extends ReferenceConfig<T> implements FactoryBean...{
    ...
    @Override
    public Object getObject() throws Exception {
        return get();
    }
}

```



3.5 服务调用

