官网：<http://dubbo.apache.org/zh-cn/>

Dubbo高级面试题：<http://dubbo.apache.org/zh-cn/docs/source_code_guide/dubbo-spi.html>

代码案例：

xml配置方式： <https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_074_rpc_dubbo_xml> 注解配置方式： <https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_075_rpc_dubbo_annotation>

# Dubbo 概念介绍

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 是什么

Dubbo 是一个分布式服务框架，致力于提供高性能和透明化的 RPC 远程服务调用方案，以及 SOA 服务治理方案。简单的说，dubbo 就是个服务框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有 dubbo 这样的分布式服务框架的需求，并且本质上是个服务调用的东东，说白了就是个远程服务调用的分布式框架

其核心部分包含:

1》远程通讯: 提供对多种基于长连接的 NIO 框架抽象封装，包括多种线程模型，序列化，以及“请求-响应”模式的信息交换方式。

2》集群容错: 提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。

3》自动发现: 基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

## Dubbo 能做什么

1.透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何 API 侵入。

2.软负载均衡及容错机制，可在内网替代 F5 等硬件负载均衡器，降低成本，减少单点。

3. 服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的 IP 地址，并且能够平滑添加或删除服务提供者。

## Dubbo的架构

dubbo 架构图如下所示（官方提供）：

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: 统计服务的调用次调和调用时间的监控中心。

Container: 服务运行容器。

对于这些角色来说，其他都还好，Monitor 可能猿友们前期使用会把它忽略，但是后期会发现它的作用十分明显哦，如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？为了解决这个问题，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，直到响应时间到达阀值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

调用关系说明：

0 服务容器负责启动，加载，运行服务提供者。

1 服务提供者在启动时，向注册中心注册自己提供的服务。

2 服务消费者在启动时，向注册中心订阅自己所需的服务。

3 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。

4 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。

5 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使用方法

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，而希望通过 API 的方式进行调用（不推荐）

# 搭建实例

## Zookeeper 的搭建

预先启动。

## 搭建项目

测试项目使用maven搭建，分模块。

### dubbo-demo模块项目

配置如下。作为项目的根节点，是不能有源代码的。

demo-api是定义了项目的所有的接口

demo-provider实现了demo-api定义的接口，并注册dubbo服务

demo-consumer使用dubbo服务，调用接口。

<modelVersion>4.0.0</modelVersion>

<groupId>org.demo</groupId>

<artifactId>dubbo-demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<name>dubbo-demo</name>

<url>http://maven.apache.org</url>

<modules>

<module>demo-api</module>

<module>demo-consumer</module>

<module>demo-provider</module>

<module>demo-web</module>

</modules>

<!-- 属性 -->

<properties>

<!-- project encoding -->

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>1.8</java.version><!-- JDK版本 build 时使用 -->

<!-- sources resultCode version -->

<sources.version>1.7</sources.version>

<!-- common util method jar -->

<common-lang3.version>3.5</common-lang3.version>

<!-- test jar -->

<junit.version>4.12</junit.version>

<!-- database connection -->

<mysql-connector>5.1.38</mysql-connector>

<!-- logger -->

<slf4j.version>1.7.21</slf4j.version>

<!-- logback -->

<logback.version>1.1.7</logback.version>

<!-- io util -->

<commons-io.version>2.5</commons-io.version>

<!-- upload util -->

<commons-fileupload.version>1.3.2</commons-fileupload.version>

<!-- json support -->

<fastjson.version>1.2.20</fastjson.version>

<!-- html -->

<jsoup.version>1.10.1</jsoup.version>

<!-- spring version -->

<spring.version>4.2.8.RELEASE</spring.version>

<!-- compiler-plugin version -->

<maven-compiler-plugin.version>3.5.1</maven-compiler-plugin.version>

<!-- surefile-plugin test util -->

<surefire-plugin.version>2.19.1</surefire-plugin.version>

<!-- mail -->

<javax-mail.version>1.4.7</javax-mail.version>

<!-- lomlok -->

<lombak.version>1.16.10</lombak.version>

<!-- api文档 -->

<springfox-swagger-ui.version>2.2.2</springfox-swagger-ui.version>

<!-- mybatis -->

<mybatis.version>1.1.1</mybatis.version>

<!-- 操作html -->

<jsoup.version>1.9.2</jsoup.version>

<!-- excel -->

<jxl.version>2.6.12</jxl.version>

<!-- microsoft office -->

<poi.version>3.10-beta2</poi.version>

<!-- dubbo -->

<dubbo.version>2.5.3</dubbo.version>

<jackson.version>2.4.0</jackson.version>

</properties>

<dependencies>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.7</version>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>dubbo</artifactId>

<version>${dubbo.version}</version>

<exclusions>

<exclusion>

<groupId>org.springframework</groupId>

<artifactId>spring</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>${jackson.version}</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.module</groupId>

<artifactId>jackson-module-jaxb-annotations</artifactId>

<version>${jackson.version}</version>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.28</version>

</dependency>

<!-- zk依赖 -->

<dependency>

<groupId>com.github.sgroschupf</groupId>

<artifactId>zkclient</artifactId>

<version>0.1</version>

</dependency>

<!-- spring 依赖 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jms</artifactId>

<version>${spring.version}</version>

</dependency>

</dependencies>

<build>

<finalName>dubbo\_demo</finalName>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<source>${java.version}</source>

<target>${java.version}</target>

<encoding>${project.build.sourceEncoding}</encoding>

</configuration>

</plugin>

</plugins>

</build>

### demo-api模块

本模块不属于dubbo的一部分。

#### 配置文件如下

<parent>

<groupId>org.demo</groupId>

<artifactId>dubbo-demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>demo-api</artifactId>

<name>demo-api</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

#### 定义的接口

User类就是一个普通的java类

public interface UserFacade {

UserDto getById(Integer id);

List<UserDto> getAll();

}

### demo-provider模块

本模块是dubbo的服务端。

#### pom配置

<parent>

<groupId>org.demo</groupId>

<artifactId>dubbo-demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>demo-provider</artifactId>

<name>demo-service</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.demo</groupId>

<artifactId>demo-api</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

</dependencies>

#### 实现接口

public class UserFacadeImpl implements UserFacade {

private UserService userService;

public UserDto getById(Integer id) {

return this.userService.getById(id);

}

public List<UserDto> getAll() {

return this.userService.getAll();

}

public void setUserService(UserService userService) {

this.userService = userService;

}

}

#### 配置dubbo

<!-- 用户服务类 -->

<bean id="userService" class="org.demo.chapter\_01\_base.service.impl.UserServiceImpl"></bean>

<!-- 对外接口实现类 -->

<bean id="userFacade" class="org.demo.chapter\_01\_base.facade.impl.UserFacadeImpl">

<property name="userService" ref="userService" />

</bean>

<!-- 1.配置提供方应用信息，用于计算依赖关系

name 应用名称

owner 应用拥有者

organization 应用所属组织/团队/项目

-->

<dubbo:application name="dubbo-demo-provider" owner="zhangxue" organization="myProject"/>

<!-- 2. 配置注册中心

address 暴露服务的地址

protocol 注册中心服务的协议，比如zookeeper,multicast,redis,simple

-->

<dubbo:registry address="zookeeper://127.0.0.1:2181" protocol="zookeeper"/>

<!-- 3. 配置dubbo接口协议以及暴露服务的端口

name 接口协议，比如dubbo,rmi,hessian,http,webservice,thrift,memcache,redis等

port 暴露服务的端口

-->

<dubbo:protocol name="dubbo" port="29014"/>

<!-- 4. 声明需要暴露的服务接口

interface 接口

ref 接口的实现类

group 1. 当一个接口有多种实现时，可以group区分

2. 按组合并返回结果 ，比如菜单服务，接口一样，但有多种实现，用group区分，现在消费方需从每种group中调用一次返回结果，合并结果返回，这样就可以实现聚合菜单项。

3. 多人合作时可以使用

version 版本控制，在接口升级后，保持旧的接口依旧可以使用

timeout 连接超时时间

-->

<dubbo:service interface="org.demo.api.chapter\_01\_base.facade.UserFacade"

ref="userFacade" timeout="5000" group="dubbo-demo-group" version="1.0.0"/>

<!-- 配置dubbo-monitor服务监控中心 -->

<dubbo:monitor protocol="registry" address="127.0.0.1:12080"></dubbo:monitor>

#### 启动dubbo服务

package org.demo.chapter\_01\_base;

import java.io.IOException;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ProviderMain {

public static void main(String[] args) {

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("chapter\_01\_base/dubbo-provider.xml");

context.start();

System.out.println("服务已启动");

try {

System.in.read();//让此程序一直跑，表示一直提供服务

} catch (IOException e) {

e.printStackTrace();

}

}

}

### demo-consumer模块

#### pom配置

<parent>

<groupId>org.demo</groupId>

<artifactId>dubbo-demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>demo-consumer</artifactId>

<name>demo-consumer</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.demo</groupId>

<artifactId>demo-api</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

</dependencies>

#### dubbo配置

<!-- 1.配置提供方应用信息，用于计算依赖关系，不是匹配条件，不要与提供方一样

name 应用名称

owner 应用拥有者

organization 应用所属组织/团队/项目

-->

<dubbo:application name="dubbo-demo-consumer" owner="zhangxue" organization="myProject"/>

<!-- 2. 配置注册中心

address 暴露服务的地址

protocol 注册中心服务的协议，比如zookeeper,multicast,redis,simple

id 为一个注册中心定义一个名字，用来区分多个服务中心

-->

<dubbo:registry address="zookeeper://127.0.0.1:2181" protocol="zookeeper" id="dubbo-demo-registry"/>

<!-- 4. 声明需要暴露的服务接口

interface 接口

ref 接口的实现类

group 1. 当一个接口有多种实现时，可以group区分

2. 按组合并返回结果 ，比如菜单服务，接口一样，但有多种实现，用group区分，现在消费方需从每种group中调用一次返回结果，合并结果返回，这样就可以实现聚合菜单项。

3. 多人合作时可以使用

version 版本控制，在接口升级后，保持旧的接口依旧可以使用

timeout 连接超时时间

retries 连接失败后重试次数 int

check 是否检验服务是否可用 Boolean

registry 指定接口注册在哪个注册中心，在一个消费者中，可能同时调用来自多个注册中心的服务，registry是用来标识服务来源的

-->

<!-- 生成远程服务代理，可以和本地bean一样使用userFacade -->

<dubbo:reference id="userFacade" interface="org.demo.api.chapter\_01\_base.facade.UserFacade" retries="0"

check="false" timeout="5000" group="dubbo-demo-group" version="1.0.0" registry="dubbo-demo-registry" >

</dubbo:reference>

#### 调用服务

package org.demo.client.chapter\_01\_base;

import org.demo.api.chapter\_01\_base.facade.UserFacade;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ConsumerMain {

public static void main(String[] args) {

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(

new String[] { "chapter\_01\_base/dubbo-consumer.xml" });

context.start();

UserFacade userSerivce = (UserFacade) context.getBean("userFacade");

System.out.println(userSerivce.getById(2));

}

}

### 运行

依次执行ServiceMain.java、ClientMain.java。会在ClientMain的控制窗口出现结果输出。

# Dubbo-admin

## 编译安装

因为zookeeper只是一个黑框，我们无法看到是否存在了什么提供者或消费者，这时就要借助Dubbo-Admin管理平台来实时的查看，也可以通过这个平台来管理提者和消费者。

dubbo-admin.war可在网上百度去下载，但是我下载了好几个war包，发布上去服务启动都报错，这个时候大概是我们系统的JDK和编译dubbo-admin.war的JDK版本不同导致的了。所以我之后直接下载了dubbo-master的源代码，然后自己编译了一个war包，这样就不会存在启动报错的问题了。（这里强烈建议自己编译一个，网上找的基本都不行，试了很多个，最后还是自己来搞）其中自己电脑编译的过程如下，一定在安装maven和jdk!

dubbo的所有源码可在https://github.com/alibaba/dubbo上下载。下好之后解压。解压后的文件内容，这里你其它的都不用去管，只管dubbo-admin。

因为这里只需要编译Dubbo-Admin，所以打开Cmd,然后进入解压路径 E:\下载\dubbo-master\dubbo-master\dubbo-admin,有人把整个工程都编译了，其实没有必要，只需要编译dubbo-admin即可，有什么依赖的，maven会自动帮你解决。

输入命令进行编译：mvn package -Dmaven.skip.test=true

这里要自己把maven的环境变量配置好，这里的-Dmaven.skip.test表示不打包测试包。

编译没有问题的话，就会出现下面的界面。生成war就知道怎么弄了吧。

## 配置

（1）配置文件位置

（2）配置文件的内容

dubbo.registry.address=zookeeper://127.0.0.1:2181

dubbo.admin.root.password=root

dubbo.admin.guest.password=guest

## 启动

放在tomcat中，启动项目。

访问：http://localhost:8080/dubbo-admin-2.5.5/

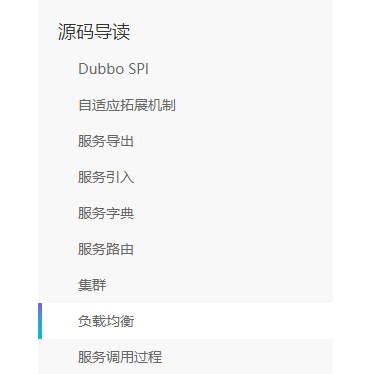
输入：root/root即可。

# Dubbo的特性

查看官网，有详细案例

# dubbo原理

官网有非常详细的案例与源码分析。



## 集群容错模式

<https://www.cnblogs.com/yxh1008/p/9251693.html>

服务保护的原则上是避免发生类似雪崩效应，尽量将异常控制在服务周围，不要扩散开。

dubbo自身的重试机制，默认3次，当失败时会进行重试，这样在某个时间点出现性能问题，然后调用方再连续重复调用，很容易引起雪崩，建议的话还是很据业务情况规划好如何进行异常处理，何时进行重试。

在生产环境默认是不推荐使用失败重试的。

服务保护的话，目前我们主要从以下几个方面来实施：

1.考虑服务的dubbo线程池类型（fix线程池的话考虑线程池大小）、数据库连接池、dubbo连接数限制是否都合适。

2.考虑服务超时时间和重试的关系，设置合适的值。

3.一定时间内服务异常数较大，则可考虑使用failfast让客户端请求直接返回或者让客户端不再请求。

### Failover Cluster 失败自动切换

调用实例失败后，继续调用其他实例。假如有 3 个实例：A, B, C，当调用 A 失败后，再调用 B，如果还是失败，则调用 C。

这里的异常不是服务端程序出现的异常，而是dubbo的RpcException才会触发重试（[异常类型](#_Dubbo的异常)）

### Failfast Cluster 快速失败，抛出异常

调用实例失败后，如果有报错，则直接抛出异常。

这里的异常不是服务端程序出现的异常，而是dubbo的RpcException才会触发重试（[异常类型](#_Dubbo的异常)）

### Failsafe Cluster 快速失败，不抛出异常

调用实例失败后，如果有报错，则忽略掉异常，返回一个正常的空结果。

这里的异常不是服务端程序出现的异常，而是dubbo的RpcException才会触发重试（[异常类型](#_Dubbo的异常)）

### Failback Cluster 失败后定时重试

调用实例发生异常后，一段时间后重新再调用，直到调用成功。

这里的异常不是服务端程序出现的异常，而是dubbo的RpcException才会触发重试（[异常类型](#_Dubbo的异常)）

### Forking Cluster 并行调用多个实例，只要一个成功就返回

并行调用多个实例，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。通过 timeout=”1000” 来设置调用超时时间。

如何使用(forks默认就是2，在使用forking的时候生效，forks该版本不能设置)

### Broadcast Cluster 广播调用所有实例，有一个报错则抛出异常

广播调用所有提供者，逐个调用，任意一台报错则报错。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。

### Available Cluster 可用的实例

调用目前可用的实例（只调用一个），如果当前没有可用的实例，则抛出异常。

### Mergeable Cluster 合并结果

该集群容错模式下，可以合并结果集，一般和 group 一起使用。

假如有 4 个provider，

A(group=dubbo\_provider\_1) B(group=dubbo\_provider\_1)

C(group=dubbo\_provider\_2) D(group=dubbo\_provider\_2)

消费者 E(group=\*)

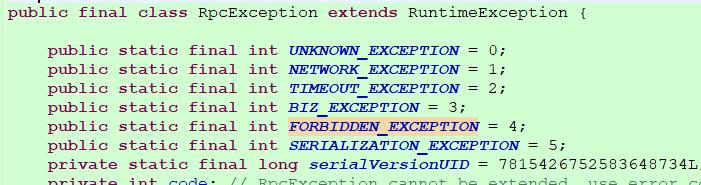
消费端配置： <dubbo:reference merger="true" />

现在有两个分组，dubbo\_provider\_1 和 dubbo\_provider\_2 。消费者会从AB中选择一个调用， CD中选择一个调用。其实合并结果就是分组聚合。如果只有AB两个服务的话就只会调用其中一个。

## Dubbo的异常

Dubbo的异常只有一个类型：RpcException

Dubbo在设计的时候，使用RpcException代表的6中类型



## Zk的作用

zk在dubbo中是服务注册与发现的注册中心,dubbo的调用过程是consumer和provider在启动的时候就和注册中心建立一个socket长连接。provider将自己的服务注册到注册中心上,注册中心将可用的提供者列表notify给consumer,consumer会将列表存储到本地缓存,consumer选举出一个要调用的提供者,去远程调用。

1. 如果这时候使用的是单点的zk,当zk宕机了,会发生什么呢？

zk宕机后不会影响现有consumer和provider之间的调用,但是新的provider想要注册到注册中心上是不行的,因为zk已经宕机了。因此单点zk一旦宕机就会影响新的提供者的注册,和新的消费者去订阅可用列表。

1. 搭建集群时搭建几台呢？

建议是搭建奇数台,最少搭建三台,搭建偶数可以吗?是可以的,但是在选举过程中没有太大用处,因为zk的leader也是遵从半数规则,使用公式计算半数(zk服务器的总数/2+1),3的半数是2,4的半数是3,因此3台和4台允许宕机的机器数都是1台。

1. 数据的一致性如何保证呢?

zk的数据结构,是类似于树形结构的，我们可以看到在com.foo.BarService这个接口下的providers下可以有多个节点,也即多个提供者,假设BarService这个接口对应的提供者有两个,这时有两个客户端同时请求这个接口,一个客户端请求提供者1,另一个客户端请求提供者2,同时执行的业务逻辑是将a的值加1,他们请求到的a的值都是0,这时,最终的结果是什么呢?在分布式情况下保证数据一致性是一个必须思考的问题,解决办法有很多,例如,分布式锁,版本控制等。

zk有自己的机制,在集群的情况下,zk会选举出一个leader负责写操作,剩下的都可以负载读操作,这样就可以将读写分离,保证的单一性,因此就不会出现上述数据不一致的情况了

## 【高德】Dubbo调用过程

学习笔记：<https://www.cnblogs.com/aspirant/p/9002663.html>

服务调用过程源码分析：<https://my.oschina.net/LucasZhu/blog/1928494>

RPC调用过程： <https://blog.csdn.net/sinat_41721615/article/details/97919847>

服务之间是怎么进行消费的: <https://mp.weixin.qq.com/s/en-LDZiI9Rd3jB5aFjFnHQ>

官网讲的最好。

Consumer端的调用方法栈如下（集群模式）



Provider调用过程



## 【美团】dubbo 怎么判断一个响应是哪个请求的？

我们一般使用dubbo作为同步调用的接口，但是dubbo默认使用的是netty作为底层网络通信，netty是异步的

有总结性的教学博客：

1.原理浅析： <https://www.cnblogs.com/aspirant/p/9002631.html>

2.集群容错模式-负载均衡模式：<https://www.cnblogs.com/aspirant/p/9001498.html>

4.dubbo的特点，8种通信协议之对比：<https://www.cnblogs.com/aspirant/p/9006670.html>

5.Dubbo 从下载到编译成功：<https://www.cnblogs.com/aspirant/p/9007288.html>

6.dubbo 架构图 以及调用过程：<https://www.cnblogs.com/aspirant/p/9002663.html>

源码讲解dubbo：<https://www.jianshu.com/nb/32484773>

简单说：在接口时同步调用的情况下

1.在消费者端，一次dubbo接口调用，会在被封装成一个request对象，其中包括一个id值，id是一个自增的id值，每次请求id都会自增，所有请求都有一个独一无二的id（在一个消费端），这个id保存在map中，key=id, value=调用方信息封装成对象。发送请求后，dubbo就会生产一个FutureTask的对象，通过condition的await方法阻塞式的等待消费返回。消息返回后，根据id找到找到调用方对象，获得FutureTask进行single，线程运行，获得调用结果。这样就是一个同步阻塞式的接口调用

2.生产者端的话，接口请求的时候，包括这个id，通过RPCContext获得消费者的ip:port，后面的调用也一样，通过一个新的request的id，去执行service、dao等，最终将结果根据id找到消费者的请求id，后面是一样的了

# 常见的Dubbo调优和经验

## dubbo一个提供方和一个消费方，默认使用单一长连接

如果消费方调用提供方其中一个服务比较慢，则会造成其它服务缓慢，解决办法是设置多个连接。

但连接数过多也会造成服务端连接暴满的问题，需要根据实际情况设置。

全局设置：

<dubbo:protocol name="dubbo" connections="2" />

单个服务设置：

<dubbo:service connections=”2”>或<dubbo:reference connections=”2”>表示该服务使用独立两条长连接。

## 事件分发模式

Dispatcher调度策略

* all 所有消息都派发到线程池，包括请求，响应，连接事件，断开事件，心跳等。
* direct 所有消息都不派发到线程池，全部在IO线程上直接执行。
* message 只有请求响应消息派发到线程池，其它连接断开事件，心跳等消息，直接在IO线程上执行。
* execution 只请求消息派发到线程池，不含响应，响应和其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
* connection 在 IO 线程上，将连接断开事件放入队列，有序逐个执行，其它消息派发到线程池。

<dubbo:protocol name="dubbo" port="20880" dispatcher="all" />

通过代码可以知道默认的是all

@SPI(AllDispatcher.NAME)

public interface Dispatcher {

/\*\*

\* dispatch the message to threadpool.

\* @param handler

\* @param url

\* @return channel handler

\*/

@Adaptive({Constants.DISPATCHER\_KEY, "dispather", "channel.handler"})

// The last two parameters are reserved for compatibility with the old configuration

ChannelHandler dispatch(ChannelHandler handler, URL url);

}

## 线程池配置

* fixed 固定大小线程池，启动时建立线程，不关闭，一直持有。(缺省)
* cached 缓存线程池，空闲一分钟自动删除，需要时重建。
* limited 可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。
* eager 优先创建Worker线程池。在任务数量大于corePoolSize但是小于maximumPoolSize时，优先创建Worker来处理任务。当任务数量大于maximumPoolSize时，将任务放入阻塞队列中。阻塞队列充满时抛出RejectedExecutionException。(相比于cached:cached在任务数量超过maximumPoolSize时直接抛出异常而不是将任务放入阻塞队列)

<dubbo:protocol name="dubbo" port="${dubbo.protocol.port}" server="netty" client="netty" serialization="dubbo" charset="UTF-8" threadpool="fixed" threads="500" queues="0" buffer="8192" accepts="0" payload="8388608"

iothreads=“9” />

默认线程池核心线程数为：200,最大线程数为200,queue为SyncronouseQueue。

考虑下，如果出现请求200个处理线程都不够，再来一个请求会发生什么情况？

底层会抛一个RejectedExecutionException，使用的是dubbo自己的拒绝策略：AbortPolicyWithReport。

这里最好不要设置queues。

如果设置了，因为在请求比较多时，如果服务提供方处理不过来，会将请求存储在queue，但因为是先进先出，所以之前早点的请求会被先处理，处理完后由于有dubbo超时时间这批请求实际是无效的。

接着导致之后新的请求就算服务已经恢复正常速度，由于还要先处理之前旧的请求导致这批请求都无效。

初始化代码：

public Executor getExecutor(URL url) {

String name = url.getParameter(Constants.THREAD\_NAME\_KEY, Constants.DEFAULT\_THREAD\_NAME);

int threads = url.getParameter(Constants.THREADS\_KEY, Constants.DEFAULT\_THREADS);

int queues = url.getParameter(Constants.QUEUES\_KEY, Constants.DEFAULT\_QUEUES);

return new ThreadPoolExecutor(threads, threads, 0, TimeUnit.MILLISECONDS,

queues == 0 ? new SynchronousQueue<Runnable>() :

(queues < 0 ? new LinkedBlockingQueue<Runnable>()

: new LinkedBlockingQueue<Runnable>(queues)),

new NamedThreadFactory(name, true), new AbortPolicyWithReport(name, url));

}

## 超时时间设置

根据不同业务设置超时时间，有些后台任务，需要设置长点。默认超时时间为6秒。

面向用户的服务，超时时间不能过长，如果这个服务出现问题，会导致雪崩。

项目超时一般的场景

1.客户端耗时大，也就是超时异常时的client elapsed xxx，这个是从创建Future对象开始到使用channel发出请求的这段时间，中间没有复杂操作，只要CPU没问题基本不会出现大耗时，顶多1ms属于正常

2.IOThread繁忙，默认情况下，dubbo协议一个客户端与一个服务提供者会建立一个共享长连接，如果某个客户端处于特别繁忙而且一直往一个服务提供者塞请求，可能造成IOThread阻塞，一般非常特殊的情况才会出现

3.服务端工作线程池中线程全部繁忙，接收消息后塞入队列等待，如果等待时间比预想长会引起超时

4.网络抖动，如果上述情况都排除了，还出现在请求发出后，服务接收请求前超过预想时间，只能归类到网络抖动了，需要SA一起查看问题

5.服务自身耗时大，这个需要应用自身做好耗时统计，当出现这种情况的时候需要用数据来说明问题及规划优化方案，建议采用缓存埋点的方式统计服务中各个执行阶段的耗时情况，最终如果超过预想时间则把缓存统计的耗时情况打日志，减少日志量，且能够得到更明确的信息

## dubbo调用链监控

使用开源的cat，pinpoint或自研。

## dubbo限流

针对某个接口做限流

服务消费方：

<dubbo:reference>connections

防止服务提供方接收过多连接：

<dubbo:protocol name="dubbo"accepts="1000"/>

## 请求响应数据大小限制

<dubbo:protocal> payload

默认8M

不要传送大包

## 业务的熔断降级

dubbo默认不提供，可以使用hystrix去做

## 服务提供方及消费者限制调用某个服务的并发数

//限制服务端，这个服务可以同时并发的线程数

<dubbo:service interface="com.foo.BarService" executes="10" />

//限制服务端，这个服务可以同时存在的连接数

<dubbo:service interface="com.foo.BarService" actives="10" />

<dubbo:reference interface="com.foo.BarService" actives="10" />

## dubbo本身上层有心跳，底层还设置了tcp的keepAlive

这样做的原因可能是担心dubbo线程池无可用线程用于心跳检测，导致服务端连接不释放。

## 服务不能拆的太细

原因：

1.在组装某个业务结果时，涉及远程调用太多

2.服务方过多，消费方建立的socket也会比较多。

## Dubbo请求拥堵场景分析&优化

<https://blog.51cto.com/nxlhero/2515849>

<https://mp.weixin.qq.com/s/dPbLLrltH5K7ul82lkxHUg>

<https://mp.weixin.qq.com/s/yUe30i4qVopD0LEpaQbbPw>

# 思考

## 自己如何写一个装饰器模式 如何实现一个rpc框架

<https://blog.csdn.net/u014513171/article/details/93161597>

## 【好未来】dubbo的扩展机制介绍下？如何自己扩展一个dubbo的扩展协议

<https://blog.csdn.net/qq_33589510/article/details/95177225>

## 服务暴露过程

<https://mp.weixin.qq.com/s/VGEONhIw8JZLGHS72iGZMg>

Dubbo服务在暴露的时候是会进行远程暴露与本地暴露。本地暴露是为了本地方法调用，不用去远程走一遍。

## 平滑迁移 Dubbo 服务

双注册中心方案。比如zk迁移到nacos。

<https://mp.weixin.qq.com/s/UpcS_gmh39nMS07N6ih-xg>