# dubbo概述

Dubbo是阿里开源的远程服务调用的分布式框架，提供了SOA 服务治理方案。

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

Dubbo是基于TCP协议的独立的RPC框架，而SpringCloud是基于HTTP协议的家族式的框架。

注：使用Dubbo框架时注意要将传输的JavaBean类实现序列化接口，因为Dubbo底层有一个hession协议，要求将传输的JavaBean实现序列化并转换为二进制数据流进行传输。

Dubbo出生于阿里系，是阿里巴巴服务化治理的核心框架，并被广泛应用于中国各互联网公司；只需要通过spring配置的方式即可完成服务化，对于应用无入侵。但框架本身的成熟度以及文档的完善程度，完全能满足各大互联网公司的业务需求。如果我们需要使用配置中心、分布式跟踪这些内容都需要自己去集成，这样无形中增加了使用 Dubbo 的难度。

Spring Cloud 是大名鼎鼎的 Spring 家族的产品， 专注于企业级开源框架的研发。 Spring Cloud 自从发展到现在，仍然在不断的高速发展，几乎考虑了服务治理的方方面面，开发起来非常的便利和简单。

Spring Cloud更新的非常快，企业需要根据自身的研发水平和所处阶段选择合适的架构来解决业务问题，不管是Dubbo还是Spring Cloud都是实现微服务有效的工具。

## dubbo架构

dubbo的架构主要有五个角色/核心组件，分为是Container（容器）、Provider（服务的提供方）、Registry（注册中心）、Consumer（服务的消费方）和Monitor（监控中心）。



### 容器

主要负责启动、加载、运行服务提供者。这里的容器当dubbo和spring整合之后，可以理解为spring的容器。

### 服务提供者

暴露服务的服务提供方，服务提供者在启动时，向注册中心注册自己提供的服务。

任意一台服务提供者宕机，由于服务提供者没有状态，所以不影响使用；如果服务提供者全部宕机，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。

服务提供者出现变更，比如增加新的机器部署，注册中心基于长连接将推送服务提供者信息给消费者；

### 服务消费者

调用远程服务的服务消费方，服务消费者在启动时，向注册中心订阅自己所需的服务，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。

#### dubbo的负载均衡算法

dubbo提供的负载均衡策略有四种：

1、随机均衡算法：按权重设置随机概率，dubbo中默认使用这种负载均衡算法；

2、权重轮询均衡算法：按照公约后的权重设置轮询比率，能考虑到每台服务器的性能，所以，在实际应用中比较常见；

3、最少活跃调用数均衡算法；

4、一致性hash均衡算法：相同参数的请求总是发到同一台提供者。

### 注册中心

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。注册中心只负责地址的注册和查找，相当于目录服务。只有在容器启动时，服务提供者和消费者才与注册中心进行交互，整个过程中，注册中心不参与数据传输，不转发请求，压力较小。

注册中心会通过部署集群来避免某一台注册中心宕机而产生的问题。集群中的任意一台宕机后，将自动切换到另外一台，不会影响已运行的提供者和消费者；如果集群中所有注册中心全部宕机之后，服务提供者和服务消费者仍能通过本地缓存通讯；数据库宕机后，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。

#### 长连接

一个连接上可以连续的发送多个数据包，在连接保持期间，如果没有数据包发送，需要双方发检测包以维持此连接；比如：数据库的连接用长连接，聊天室，实时游戏。

#### 短连接

指通信的双方有数据交互时，建立一个连接，数据发送完成之后，则会断开连接。适用于网页浏览等数据刷新频度较低的场景。

### 监控中心

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。监控中心宕机不影响使用，只是丢失部分的采样数据。dubbo-admin可以通过监控中心的可视化界面，进行禁止服务和截止消费者（大量恶意访问的ip）。

**调用关系说明**

服务容器负责启动，加载，运行服务提供者。

服务提供者在启动时，向注册中心注册自己提供的服务。

服务消费者在启动时，向注册中心订阅自己所需的服务。

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。

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。

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支持的协议

Dubbo支持Dubbo协议、RMI协议、hessian协议、Http协议等

### Dubbo协议

缺省（默认）协议 、采用了单一长连接和NIO （非阻塞式IO）异步通讯、使用线程池并发处理请求，能减少握手和加大并发效率、采用的是Hession二进制序列化 、性能较好，推荐使用。

主要应用于传入传出参数数据包较小（建议小于100K），消费者比提供者个数多，由于是单一连接，因为尽量不要传输大文件。

### RMI协议

采用JDK标准的RMI（**Remote Method Invocation**）协议（基于TCP协议）、堵塞式短连接、JDK标准序列化方式、同步通讯。适用于消费者和提供者个数差不多的，可传文件。测试发现偶尔会连接失败，需要重建Stub 。

Stub是指存根，为了屏蔽客户调用远程主机上的对象，必须提供某种方式来模拟本地对象，这个本地对象称为存根。

### Hessian协议

采用http通讯，采用Servlet暴露服务，多连接短连接的同步传输方式，采用hession的二进制序列化，适合提供者比消费者多。

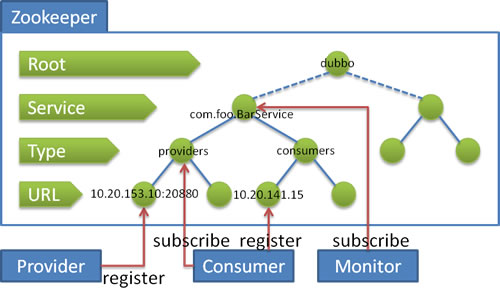
## RPC协议

RPC（Remote Procedure Call Protocol）协议通俗的说，就是两台服务器A、B，应用A部署在A服务器上，应用B部署在B服务器上，A应用想要调用B服务器上B应用提供的函数（方法），由于不在一个内存空间，不能直接调用，需要通过网络通讯、协议、寻址的方式来表达调用的语义和传达调用的数据。RPC的亮点就是将远程调用的细节隐藏起来，使得调用远程服务像调用本地服务一样简单。

## SOA架构

SOA（Service-Oriented Architecture）:面向服务的架构，也可以理解为资源调度和治理中心。所谓服务，拿项目三举个最简单的例子，把商品、订单、搜索等核心业务抽取出来，作为独立的服务。和面向对象来比较，面向对象的程序是由一个一个的类组成的，而面向服务的架构是由一个一个服务组成的。如果有多个系统、多个终端（PC、APP等）都想调用我这个服务，那么就可以通过同一套协议和接口规范来调用。而服务越来越多，容量和资源利用率不相同的时候，可以增加一个调度中心基于访问的压力实时的管理集群容量，从而提高集群利用率，因此Dubbo就可以在中间起到的资源（服务）调度和治理中心。

**Dubbo在zookeeper中注册信息的结构**



# 在Linux中配置dubbo

dubbo本身并不是一个服务软件。它其实就是一个jar包能够帮你的java程序连接到zookeeper，并利用zookeeper消费、提供服务。所以你不用在Linux上启动什么dubbo服务。但是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。

安装dubbo监控程序需要在Linux虚拟机上传tomcat和dubbo的压缩包。

## 配置dubbo

1、将dubbo的rar压缩包上传到Linux虚拟机并解压

|  |
| --- |
| unzip dubbo-admin-2.6.0.rar |

2、修改tomcat的配置文件，在server.xml配置文件尾部添加

|  |
| --- |
| <Context path="dubbo" docBase="/opt/dubbo-admin-2.6.0/" debug="0" privileged="true" /> |

debug表示该项目是否支持以debug形式启动；privileged表示该项目的优先级；baseDoc的值是dubbo的解压目录。

如下配置以后，就不需要再将dubbo框架存放到tomcat的webApp目录下了。

|  |
| --- |
| <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"  prefix="localhost\_access\_log." suffix=".txt"  pattern="%h %l %u %t &quot;%r&quot; %s %b" />  <Context path="dubbo" docBase="/opt/dubbo-admin-2.6.0/" />  </Host>  </Engine>  </Service>  </Server> |

3、在Linux虚拟机上启动tomcat服务器

|  |
| --- |
| sh startup.sh |

4、测试是否全部配置成功，在浏览器中输入<http://192.168.235.23:8080/dubbo/>，然后在弹框中填写账户密码，账户密码均为root，如果能够进入dubbo界面则表示配置成功。



## 配置tomcat开机自启

1、进入目录/etc/init.d，通过vim编辑器编写开机自启脚本，将如下内容复制进脚本文件中

|  |
| --- |
| [root@xmm init.d]# vim dubbo-admin  #!/bin/bash  #chkconfig:2345 20 90  i#description:dubbo-admin  #processname:dubbo-admin  CATALANA\_HOME=/opt/apache-tomcat-7.0.75  export JAVA\_HOME=/opt/jdk1.8.0\_121  case $1 in  start)  echo "Starting Tomcat..."  $CATALANA\_HOME/bin/startup.sh  ;;  stop)  echo "Stopping Tomcat..."  $CATALANA\_HOME/bin/shutdown.sh  ;;  restart)  echo "Stopping Tomcat..."  $CATALANA\_HOME/bin/shutdown.sh  sleep 2  echo  echo "Starting Tomcat..."  $CATALANA\_HOME/bin/startup.sh  ;;  \*)  echo "Usage: tomcat {start|stop|restart}"  ;; esac |

2、将该脚本注册到服务中并修改脚本文件的权限

|  |
| --- |
| [root@xmm init.d]# chkconfig --add dubbo-admin  [root@xmm init.d]# chmod 777 dubbo-admin |

3、启动tomcat服务器

|  |
| --- |
| [root@xmm init.d]# service dubbo-admin start |

# zookeeper

Zookeeper是Apache Hadoop的一个子项目，作为分布式协调作用（类似于我们的大脑），树形的目录结构，支持变更操作，同时也是Dubbo官方推荐的注册中心。

## 注册中心的作用

### 公示服务信息

注册中心就像是一个大型公告板，这里发布的信息，整个项目中各个模块都能够看到。

### 管理服务信息

服务调用框架访问注册中心读取服务信息，从而对开发人员屏蔽相关细节。在开发时简化代码，实现远程方法的声明式调用。

声明式方法远程调用是指像调用本地方法一样，调用远程方法。

## 典型的注册中心

**Dubbo(远程方法调用框架)的搭档Zookeeper**

Zookeeper之所以能用来作为dubbo的注册中心来使用，主要是应用到dubbo的命名功能 。在SOA架构、集群和分布式环境下，子项目之间的调用关系会变得越来越复杂，因为需要一个服务器专门给我们管理服务的信息和调节、管理这些服务，让我们的侧重点放在项目中的业务上，因为zookeeper的命名功能就可以充当这样一个服务器。

zookeeper命名功能：利用Zookeeper的分层结构，可以把系统中的各种服务的名称、地址、以及目录信息存放在Zookeeper中，需要的时候去Zookeeper中读取。

Zookeeper的数据模型是由一系列的Znode数据节点组成，和文件系统类似。但是与传统的磁盘文件系统不同的是，首先，zookeeper的数据全部存储在内存中，性能高；其次，zookeeper也支持集群，实现了高可用；同时基于zookeeper的特性，也支持事件监听 （服务的暴露方发生变化，可以进行推送），因此zookeeper适合作为dubbo的注册中心区使用。

**SpringCloud家族中的Eureka注册中心**

SpringCloud有五个基本的组件：

 Eureka：注册中心

 Ribbon：负载均衡

 Feign：声明式调用

 Hystrix：服务熔断、降级、监控

 Zuul：网关

**redis注册中心**

redis作为注册中心时用key-value（Hash）来存储数据

主key:服务器名和类型

Map中的key：url地址

Map中的value：过期时间，判断脏数据，脏数据由监控中心删除（要求服务器时间必须相同）

利用redis中的Publish/Subscribe事件通知数据变更。总之，redis作为注册中心来使用的话，支持集群，性能高，但是要求所有服务器的时间必须同步，要求较高。

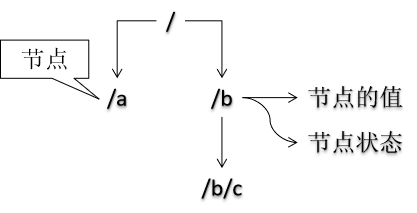
**simple注册中心**

本身就是一个普通的dubbo服务，能减少第三方依赖，不支持集群，不适合生产环境。

## zookeeper实现原理

Zookeeper会维护一个类似于标准的文件系统的具有层次关系的数据结构。这个文件系统中每个子目录项都被称为znode节点，这个znode节点也可以有子节点，每个节点都可以存储数据，客户端也可以对这些node节点进行getChildren，getData,exists方法，同时也可以在znode tree路径上设置watch（类似于监听），当watch路径上发生节点create、delete、update的时候，会通知到client。client可以得到通知后，再获取数据，执行业务逻辑操作。Zookeeper 的作用主要是用来维护和监控存储的node节点上这些数据的状态变化，通过监控这些数据状态的变化，从而达到基于数据的集群管理。

## zookeeper相关概念



ZooKeeper使用树形结构管理数据。而且以“/”作为树形结构的根节点。树形结构中的每一个节点都称为“znode”。文件系统中的目录可以存放其他目录和文件，znode中可以存放其他znode，也可以对应一个具体的值。znode和它对应的值之间是键值对的关系。

每一个znode上同时还有一套状态信息，称为：stat。

ZooKeeper 的设计目标是将那些复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集，并以一系列简单易用的接口提供给用户使用。

原语： 操作系统或计算机网络用语范畴。是由若干条指令组成的，用于完成一定功能的一个过程。具有不可分割性·即原语的执行必须是连续的，在执行过程中不允许被中断。

ZooKeeper 是一个典型的分布式数据一致性解决方案，分布式应用程序可以基于 ZooKeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

Zookeeper 一个最常用的使用场景就是用于担任服务生产者和服务消费者的注册中心。服务生产者将自己提供的服务注册到Zookeeper中心，服务的消费者在进行服务调用的时候先到Zookeeper中查找服务，获取到服务生产者的详细信息之后，再去调用服务生产者的内容与数据。

### 概念

#### 会话（Session）

Session 指的是 ZooKeeper 服务器与客户端会话。在 ZooKeeper 中，一个客户端连接是指客户端和服务器之间的一个 TCP 长连接。客户端启动的时候，首先会与服务器建立一个 TCP 连接，从第一次连接建立开始，客户端会话的生命周期也开始了。通过这个连接，客户端能够通过心跳检测与服务器保持有效的会话，也能够向Zookeeper服务器发送请求并接受响应，同时还能够通过该连接接收来自服务器的Watch事件通知。 Session的sessionTimeout值用来设置一个客户端会话的超时时间。当由于服务器压力太大、网络故障或是客户端主动断开连接等各种原因导致客户端连接断开时，只要在sessionTimeout规定的时间内能够重新连接上集群中任意一台服务器，那么之前创建的会话仍然有效。

在为客户端创建会话之前，服务端首先会为每个客户端都分配一个sessionID。由于 sessionID 是 Zookeeper 会话的一个重要标识，许多与会话相关的运行机制都是基于这个 sessionID 的，因此，无论是哪台服务器为客户端分配的 sessionID，都务必保证全局唯一。

#### Znode

在谈到分布式的时候，我们通常说的“节点"是指组成集群的每一台机器。然而，在Zookeeper中，“节点"分为两类，第一类同样是指构成集群的机器，我们称之为机器节点；第二类则是指数据模型中的数据单元，我们称之为数据节点一一ZNode。

Zookeeper将所有数据存储在内存中，数据模型是一棵树（Znode Tree)，由斜杠（/）的进行分割的路径，就是一个Znode，例如/foo/path1。每个上都会保存自己的数据内容，同时还会保存一系列属性信息。

在Zookeeper中，node可以分为持久节点和临时节点两类。所谓持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在Zookeeper上。而临时节点就不一样了，它的生命周期和客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。另外，ZooKeeper还允许用户为每个节点添加一个特殊的属性：SEQUENTIAL.一旦节点被标记上这个属性，那么在这个节点被创建的时候，Zookeeper会自动在其节点名后面追加上一个整型数字，这个整型数字是一个由父节点维护的自增数字。

#### 版本

在前面我们已经提到，Zookeeper 的每个 ZNode 上都会存储数据，对应于每个ZNode，Zookeeper 都会为其维护一个叫作 Stat 的数据结构，Stat中记录了这个 ZNode 的三个数据版本，分别是version（当前ZNode的版本）、cversion（当前ZNode子节点的版本）和 cversion（当前ZNode的ACL版本）。

#### Watcher

Watcher（事件监听器），是Zookeeper中的一个很重要的特性。Zookeeper允许用户在指定节点上注册一些Watcher，并且在一些特定事件触发的时候，ZooKeeper服务端会将事件通知到感兴趣的客户端上去，该机制是Zookeeper实现分布式协调服务的重要特性。

#### ACL

Zookeeper采用ACL（AccessControlLists）策略来进行权限控制，类似于 UNIX 文件系统的权限控制。Zookeeper 定义了如下5种权限。

CREATE 创建子节点的权限

READ 获取节点数据和子节点列表的权限

WRITE 更新节点数据的权限

DELETE 删除子节点的权限

ADMIN 设置节点ACL的权限

其中尤其需要注意的是，CREATE和DELETE这两种权限都是针对子节点的权限控制。

### 特点

**顺序一致性**： 从同一客户端发起的事务请求，最终将会严格地按照顺序被应用到 ZooKeeper 中去。

**原子性**： 所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群中所有的机器都成功应用了某一个事务，要么都没有应用。

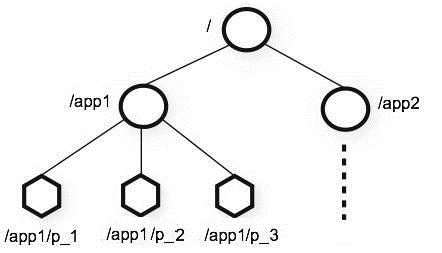
**单一系统映像** ： 无论客户端连到哪一个 ZooKeeper 服务器上，其看到的服务端数据模型都是一致的。

**可靠性**： 一旦一次更改请求被应用，更改的结果就会被持久化，直到被下一次更改覆盖。

### 设计目标

#### 简单的数据模型

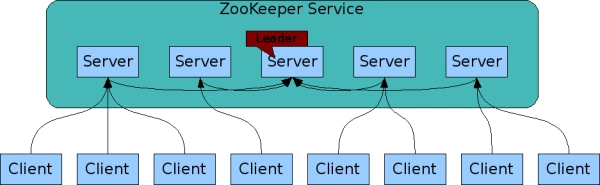
ZooKeeper 允许分布式进程通过共享的层次结构命名空间进行相互协调，这与标准文件系统类似。 名称空间由 ZooKeeper 中的数据寄存器组成 - 称为znode，这些类似于文件和目录。 与为存储设计的典型文件系统不同，ZooKeeper数据保存在内存中，这意味着ZooKeeper可以实现高吞吐量和低延迟。



#### 可构建集群

为了保证高可用，最好是以集群形态来部署 ZooKeeper，这样只要集群中大部分机器是可用的（能够容忍一定的机器故障），那么zookeeper本身仍然是可用的。 客户端在使用 ZooKeeper 时，需要知道集群机器列表，通过与集群中的某一台机器建立 TCP 连接来使用服务，客户端使用这个TCP链接来发送请求、获取结果、获取监听事件以及发送心跳包。如果这个连接异常断开了，客户端可以连接到另外的机器上。

ZooKeeper 官方提供的架构图：



上图中每一个Server代表一个安装Zookeeper服务的服务器。组成 ZooKeeper 服务的服务器都会在内存中维护当前的服务器状态，并且每台服务器之间都互相保持着通信。集群间通过 Zab 协议（Zookeeper Atomic Broadcast）来保持数据的一致性。

#### 顺序访问

对于来自客户端的每个更新请求，ZooKeeper 都会分配一个全局唯一的递增编号，这个编号反应了所有事务操作的先后顺序，应用程序可以使用 ZooKeeper 这个特性来实现更高层次的同步原语。 这个编号也叫做时间戳——zxid（Zookeeper Transaction Id）

#### 高性能

ZooKeeper 是高性能的。 在“读”多于“写”的应用程序中尤其地高性能，因为“写”会导致所有的服务器间同步状态。（“读”多于“写”是协调服务的典型场景。）

### 配置文件

**tickTime**

通信心跳数,ZooKeeper服务器心跳时间，单位毫秒

ZooKeeper使用的基本时间，服务器之间或客户端与服务器之间维持心跳的时间间隔，也就是每个tickTime时间就会发送一个心跳，时间单位为毫秒。

用于心跳机制，并且设置最小的session超时时间为两倍心跳时间(session的最小超时时间是2\*tickTime)。

**initLimit**

LF初始通信时限

集群中的Follower跟随者服务器(F)与Leader领导者服务器(L)之间初始连接时能容忍的最多心跳数（tickTime的数量）。

投票选举新Leader的初始化时间，Follower在启动过程中，会从Leader同步所有最新数据，然后确定自己能够对外服务的起始状态。

Leader允许Follower在initLimit时间内完成这个工作。

**syncLimit**

LF同步通信时限

集群中Leader与Follower之间的最大响应时间单位，假如响应超过syncLimit \* tickTime，Leader认为Follwer死掉，从服务器列表中删除Follwer。

在运行过程中，Leader负责与ZooKeeper集群中所有机器进行通信，例如通过一些心跳检测机制，来检测机器的存活状态。

如果L发出心跳包在syncLimit之后，还没有从F那收到响应，那么就认为这个F已经不在线了。

**dataDir**

数据文件目录+数据持久化路径

保存内存数据库快照信息的位置，如果没有其他说明，更新的事务日志也保存到数据库。

**clientPort**

客户端连接端口

# zookeeper命令

## 一般命令

**ZooKeeper服务器与客户端**

启动服务器：./zkServer.sh start

停止服务器：./zkServer.sh stop

启动客户端：./zkCli.sh

退出客户端：[zk: localhost:2181(CONNECTED) 6] quit

**ls znode**

查看当前znode中所包含的内容

**ls2 znode**

查看当前节点数据并能看到更新次数等数据

**stat znode**

查看节点状态

**create [-s] [-e] path data acl**

普通创建：不带有-s、-e参数

-s：含有序列

-e：临时（重启或者超时消失）

**set znode**

设置节点的具体值

set 节点 value值

**get znode**

获得节点的值

get节点

**delete znode**

可以删除指定znode，当该znode拥有子znode时，必须先删除其所有子znode，否则操作将失败。

**rmr znode**

rmr命令可用于代替delete命令，rmr是一个递归删除命令，如果发生指定节点拥有子节点时，rmr命令会首先删除子节点。

## 四字命令

**介绍**

ZooKeeper支持某些特定的四字命令，他们大多是用来查询ZooKeeper服务的当前状态及相关信息的，使用时通过telnet或nc向ZooKeeper提交相应命令。

[root@right bin]# echo ruok | nc localhost 2181

imok[root@right bin]#

**nc命令**

nc命令需要安装对应的程序才可以使用。

yum install -y nc

**常用四字命令**

ruok：测试服务是否处于正确状态。如果确实如此，那么服务返回“imok ”，否则不做任何响应

stat：输出关于性能和连接的客户端的列表

conf：输出相关服务配置的详细信息

cons：列出所有连接到服务器的客户端的完全的连接 /会话的详细信息。包括“接受 / 发送”的包数量、会话id 、操作延迟、最后的操作执行等等信息

dump：列出未经处理的会话和临时节点

envi：输出关于服务环境的详细信息（区别于conf命令）

reqs：列出未经处理的请求

wchs：列出服务器watch的详细信息

wchc：通过session列出服务器watch的详细信息，它的输出是一个与watch相关的会话的列表

wchp：通过路径列出服务器 watch的详细信息。它输出一个与 session相关的路径

# zookeeper节点

## 节点类型

**PERSISTENT-持久化目录节点**

客户端与zookeeper断开连接后，该节点依旧存在

**PERSISTENT\_SEQUENTIAL-持久化顺序编号目录节点**

客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

**EPHEMERAL-临时目录节点**

客户端与zookeeper断开连接后，该节点被删除

**EPHEMERAL\_SEQUENTIAL-临时顺序编号目录节点**

客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号

## 节点状态

### 介绍

znode维护了一个stat结构，这个stat包含数据变化的版本号、访问控制列表变化、还有时间戳。版本号和时间戳一起，可让ZooKeeper验证缓存和协调更新。每次znode的数据发生了变化，版本号就增加。

例如：无论何时客户端检索数据，它也一起检索数据的版本号。并且当客户端执行更新或删除时，客户端必须提供他正在改变的znode的版本号。如果它提供的版本号和真实的数据版本号不一致，更新将会失败。

### 属性

czxid：引起这个znode创建的zxid，创建节点的事务的zxid（ZooKeeper Transaction Id）

ctime：znode被创建的毫秒数(从1970年开始)

mzxid：znode最后更新的zxid

mtime：znode最后修改的毫秒数(从1970年开始)

pZxid：znode最后更新的子节点zxid

cversion：znode子节点变化号，znode子节点修改次数

dataversion：znode数据变化号

aclVersion：znode访问控制列表的变化号

ephemeralOwner：如果是临时节点，这个是znode拥有者的session id。如果不是临时节点则是0。

dataLength：znode的数据长度

numChildren：znode子节点数量

# zookeeper的Java客户端

## 依赖信息

|  |
| --- |
| <dependency>  <groupId>com.github.sgroschupf</groupId>  <artifactId>zkclient</artifactId>  <version>0.1</version> </dependency> <dependency>  <groupId>org.apache.zookeeper</groupId>  <artifactId>zookeeper</artifactId>  <version>3.4.9</version> </dependency> |

## ZooKeeper类方法

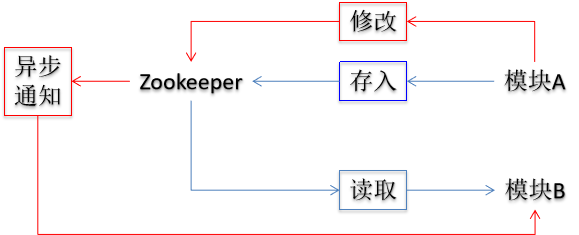
|  |  |
| --- | --- |
| Zookeeper表格 | |
| void | **close**()断开ZooKeeper服务器连接 |
| String | **create**(overload) 创建节点 |
| void | **exists**(overload) 判断节点是否存在 |
| byte[] | **getData**(overload) 获取节点上的数据 |
| Stat | **setData**(overload) 设置节点上的数据 |

## 示例

|  |
| --- |
| private ZooKeeper zooKeeper; {  try {  String connectString = "192.168.235.23:2181";  int sessionTimeout = 5000;  Watcher watcher = new Watcher() {  @Override  public void process(WatchedEvent watchedEvent) {   }  };  zooKeeper = new ZooKeeper(connectString, sessionTimeout, watcher);  } catch (IOException e) {  e.printStackTrace();  } } @Test public void testSetData() throws KeeperException, InterruptedException {  String path = "/fruit/apple";  boolean watch = false;  Stat stat = new Stat();  byte[] nodeData = zooKeeper.getData(path, watch, stat);  String result = new String(nodeData);  System.*err*.println(result);  // Zookeeper要求基于最新版修改数据,版本值使用-1则不进行版本检查  int dataVersion = -1;  Stat resultStat = zooKeeper.setData(path, "green".getBytes(), dataVersion);  System.*err*.println(resultStat); } |

# zookeeper异步通知机制

## 工作机制



在分布式项目中随着业务功能越来越多，具体的功能模块也会越来越多，一个大型的电商项目能达到几十个模块甚至更多。这么多业务模块的工程有可能需要共享一些信息，这些信息一旦发生变化，在各个相关模块工程中手动逐一修改会非常麻烦，甚至可能发生遗漏，严重的可能导致系统崩溃，造成经济损失。

使用ZooKeeper的通知机制后，各个模块工程在特定znode上设置Watcher（观察者）来监控当前节点上值的变化。一旦Watcher检测到了数据变化就会立即通知模块工程，从而自动实现“一处修改，处处生效”的效果。

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。

ZooKeeper支持Watch（观察）机制，客户端可以在每个znode结点上设置一个Watcher（观察者）。如果被观察服务端的znode结点有变更，那么Watcher就会被触发，这个Watcher所属的客户端将接收到一个通知包被告知结点已经发生变化，这就是把相应的事件通知给设置过Watcher的Client端。

ZooKeeper里的所有读取操作：getData(),getChildren()和exists()都有设置Watch的选项。

总结成一句话：ZooKeeper的观察机制是一种异步回调的触发机制。

当数据有了变化时zkServer向客户端发送一个Watch通知，这是个一次性动作，触发一次就失效了。

如果想继续Watch的话，需要客户端重新设置Watcher。因此如果你得到了一个Watch事件，并且在将来继续得到节点变化通知，那么就必须另外设置一个新的Watcher继续观察。

节点有不同的改动方式。可以认为ZooKeeper维护两个观察列表：数据观察和子节点观察。getData()和exists()设置数据观察。getChildren()设置子节点观察。此外，还可以认为不同的返回数据有不同的观察。getData()和exists()返回节点的数据，而getChildren()返回子节点列表。所以，setData()将为znode触发数据观察。成功的create()将为新创建的节点触发数据观察，为其父节点触发子节点观察。成功的delete()将会为被删除的节点触发数据观察以及子节点观察（因为节点不能再有子节点了），为其父节点触发子节点观察。如果一个节点设置存在观察时尚未创建，并且在断开连接后执行节点创建以及删除操作，那么这个节点上设置的观察事件客户端接收不到，事件会丢失。

## 一次性通知

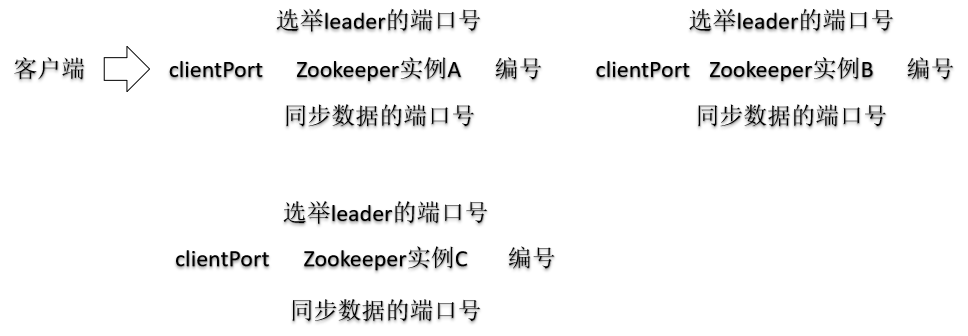
|  |
| --- |
| public void testNoticeOnce() throws KeeperException, InterruptedException {  String path = "/fruit/apple";  Stat stat = new Stat();  // 创建Watcher对象，用于监控节点值  Watcher watcher = new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*err*.println("☆☆☆哇塞！我接收到通知了耶！☆☆☆");  }  };  byte[] result = zooKeeper.getData(path, watcher, stat);  System.*err*.println("当前节点值="+new String(result));  while(true) {  Thread.*sleep*(5000);  System.*err*.println("药不能停！~~~~~~~~~~");  } } |

## 持续通知

|  |
| --- |
| public void testNoticeForEver() throws InterruptedException {  String path = "/fruit/apple";  getDataMine(zooKeeper, path);  while(true) {  Thread.*sleep*(5000);  System.*err*.println("药不能停！~~~~~~~~~~"+Thread.*currentThread*().getName());  } } private void getDataMine(ZooKeeper zooKeeper, String path) {  System.*err*.println("getDataMine()方法开始");  try {  byte[] result = zooKeeper.getData(path, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*err*.println("process()方法开始"+Thread.*currentThread*().getName());  // 在调用process方法时，Watcher对象失效，那么为了持续监控，  // 再调用getDataMine()方法设置新Watcher  getDataMine(zooKeeper, path);  System.*err*.println("process()方法结束"+Thread.*currentThread*().getName());  }  }, new Stat());  System.*err*.println("☆☆☆☆☆☆☆☆☆☆当前节点值="+new String(result));  } catch (Exception e) {  e.printStackTrace();  }  System.*err*.println("getDataMine()方法结束"); } |

# zookeeper集群

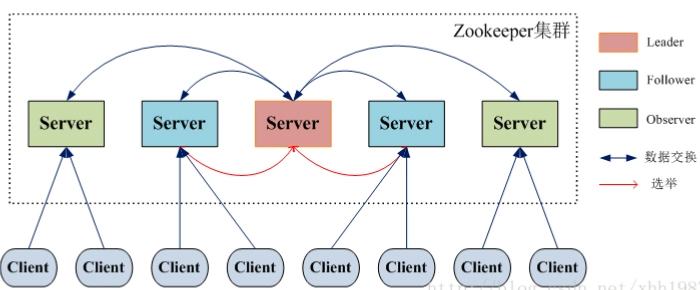
## 数据通信机制



## 集群角色

最典型集群模式： Master/Slave 模式（主备模式）。在这种模式中，通常 Master服务器作为主服务器提供写服务，其他的 Slave 服务器从服务器通过异步复制的方式获取 Master 服务器最新的数据提供读服务。

但是，在 ZooKeeper 中没有选择传统的 Master/Slave 概念，而是引入了Leader、Follower 和 Observer 三种角色。如下图所示



ZooKeeper 集群中的所有机器通过一个 Leader 选举过程来选定一台称为 “Leader” 的机器，Leader 既可以为客户端提供写服务又能提供读服务。除了 Leader 外，Follower 和 Observer 都只能提供读服务。Follower 和 Observer 唯一的区别在于 Observer 机器不参与 Leader 的选举过程，也不参与写操作的“过半写成功”策略，因此 Observer 机器可以在不影响写性能的情况下提升集群的读性能。



## 服务器名称和地址

集群信息（服务器编号，服务器地址，LF通信端口，选举端口），这个配置项的书写格式比较特殊，规则如下：server.N=YYY:A:B

N 服务器编号

YYY 服务器的IP地址

A LF通信端口，用来和leader交换信息

B leader挂掉后选举新leader的端口

## 搭建步骤

第1步：创建集群所在目录

mkdir /opt/cluster-zk

第2步：重新解压tar包到集群目录

tar -zxvf /opt/zookeeper-3.4.9.tar.gz -C /opt/cluster-zk/

第3步：复制新的解压目录

cp -r /opt/cluster-zk/zookeeper-3.4.9/ /opt/cluster-zk/zkone

第4步：配置zkone

i：创建zoo.cfg配置文件

cp /opt/cluster-zk/zkone/conf/zoo\_sample.cfg /opt/cluster-zk/zkone/conf/zoo.cfg

ii：创建数据目录

mkdir /opt/cluster-zk/zkone/data

iii：在数据目录中创建编号文件

vim /opt/cluster-zk/zkone/data/myid

iv：编辑编号文件，内容就是当前服务器实例的编号

1

v：配置zoo.cfg

dataDir=/opt/cluster-zk/zkone/data

clientPort=1000

server.1=127.0.0.1:1001:1002

server.2=127.0.0.1:2001:2002

server.3=127.0.0.1:3001:3002

第5步：配置zktwo

i：把zktwo复制出来

cp -r /opt/cluster-zk/zkone /opt/cluster-zk/zktwo

ii：修改myid文件中编号值

vim /opt/cluster-zk/zktwo/data/myid

将1改成2

iii：修改zoo.cfg

dataDir=/opt/cluster-zk/zktwo/data

clientPort=2000

第6步：配置zkthree

i：把zkthree复制出来

cp -r /opt/cluster-zk/zkone /opt/cluster-zk/zkthree

ii：修改myid文件中编号值

vim /opt/cluster-zk/zkthree/data/myid

将1改成3

iii：修改zoo.cfg

dataDir=/opt/cluster-zk/zkthree/data

clientPort=3000

## 创建操作服务器的可执行脚本

**创建文件**

|  |
| --- |
| /opt/cluster-zk/start.sh  /opt/cluster-zk/stop.sh  /opt/cluster-zk/status.sh |

**编辑启动服务器命令**

|  |
| --- |
| /opt/cluster-zk/zkone/bin/zkServer.sh start  /opt/cluster-zk/zktwo/bin/zkServer.sh start  /opt/cluster-zk/zkthree/bin/zkServer.sh start |

**编辑停止服务器命令**

|  |
| --- |
| /opt/cluster-zk/zkone/bin/zkServer.sh stop  /opt/cluster-zk/zktwo/bin/zkServer.sh stop  /opt/cluster-zk/zkthree/bin/zkServer.sh stop |

**编辑查看服务器状态命令**

|  |
| --- |
| /opt/cluster-zk/zkone/bin/zkServer.sh status  /opt/cluster-zk/zktwo/bin/zkServer.sh status  /opt/cluster-zk/zkthree/bin/zkServer.sh status |

**给脚本文件设置可执行权限**

|  |
| --- |
| chmod 755 /opt/cluster-zk/start.sh  chmod 755 /opt/cluster-zk/stop.sh  chmod 755 /opt/cluster-zk/status.sh |

**执行脚本**

|  |
| --- |
| [root@xmm cluster-zk]# ./start.sh  ZooKeeper JMX enabled by default  Using config: /opt/cluster-zk/zkone/bin/../conf/zoo.cfg  Starting zookeeper ... STARTED  ZooKeeper JMX enabled by default  Using config: /opt/cluster-zk/zktwo/bin/../conf/zoo.cfg  Starting zookeeper ... STARTED  ZooKeeper JMX enabled by default  Using config: /opt/cluster-zk/zkthree/bin/../conf/zoo.cfg  Starting zookeeper ... STARTED  [root@xmm cluster-zk]# ./stop.sh  ZooKeeper JMX enabled by default  Using config: /opt/cluster-zk/zkone/bin/../conf/zoo.cfg  Stopping zookeeper ... STOPPED  ZooKeeper JMX enabled by default  Using config: /opt/cluster-zk/zktwo/bin/../conf/zoo.cfg  Stopping zookeeper ... STOPPED  ZooKeeper JMX enabled by default  Using config: /opt/cluster-zk/zkthree/bin/../conf/zoo.cfg  Stopping zookeeper ... STOPPED  [root@xmm cluster-zk]# ./status.sh  ZooKeeper JMX enabled by default  Using config: /opt/cluster-zk/zkone/bin/../conf/zoo.cfg  Mode: follower  ZooKeeper JMX enabled by default  Using config: /opt/cluster-zk/zktwo/bin/../conf/zoo.cfg  Mode: follower  ZooKeeper JMX enabled by default  Using config: /opt/cluster-zk/zkthree/bin/../conf/zoo.cfg  Mode: leader |

**客户端登录**

|  |
| --- |
| /opt/zookeeper-3.4.9/bin/zkCli.sh -server 127.0.0.1:1000  /opt/zookeeper-3.4.9/bin/zkCli.sh -server 127.0.0.1:2000  /opt/zookeeper-3.4.9/bin/zkCli.sh -server 127.0.0.1:3000 |

### 测试

 leader、follower分别写入数据，到其他服务器查看

 follower宕机、重启查看数据是否可以同步

 leader宕机、重启查看服务器状态

### 集群中服务器数量

结论：一般来说，集群中服务器数量最好设置为单数。

集群中有超过一半的服务器正常工作，则整个集群判断为正常工作，对外提供服务时大致满足预期。

**推导：**

共2实例：宕机1实例，剩下1=2/2，没有超过一半。死亡容忍度为0。

共3实例：宕机1实例，剩下2>3/2，超过一半。死亡容忍度为1。

共4实例：宕机1实例，剩下3>4/2，超过一半。死亡容忍度为1。

共5实例：宕机2实例，剩下3>5/2，超过一半。死亡容忍度为2。

共6实例：宕机2实例，剩下4>6/2，超过一半。死亡容忍度为2。

## Linux上安装

zookeeper需要在jvm虚拟机上运行，所以要有jdk支持。

|  |
| --- |
| [root@xmm opt]# java -version  java version "1.8.0\_121"  Java(TM) SE Runtime Environment (build 1.8.0\_121-b13)  Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode) |

上传并解压zookeeper的tar包，然后设置其配置文件，首先将zoo\_sample.cfg文件复制一份并命名为zoo.cfg(zookeeper的配置文件名必须是zoo.cfg)。然后修改其中的配置。

|  |
| --- |
| [root@xmm conf]# cp zoo\_sample.cfg zoo.cfg  [root@xmm conf]# vim zoo.cfg  dataDir=/opt/zookeeper-3.4.9/data |

创建数据目录

|  |
| --- |
| [root@xmm zookeeper-3.4.9]# mkdir data |

## 总结

ZooKeeper 本身就是一个分布式程序（只要半数以上节点存活，ZooKeeper 就能正常服务）。

为了保证高可用，最好是以集群形态来部署 ZooKeeper，这样只要集群中大部分机器是可用的（能够容忍一定的机器故障），那么 ZooKeeper 本身仍然是可用的。

ZooKeeper 将数据保存在内存中，这也就保证了 高吞吐量和低延迟（但是内存限制了能够存储的容量不太大，此限制也是保持znode中存储的数据量较小的进一步原因）。

ZooKeeper 是高性能的。 在“读”多于“写”的应用程序中尤其地高性能，因为“写”会导致所有的服务器间同步状态。（“读”多于“写”是协调服务的典型场景。）

ZooKeeper有临时节点的概念。 当创建临时节点的客户端会话一直保持活动，瞬时节点就一直存在。而当会话终结时，瞬时节点被删除。持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在Zookeeper上。

ZooKeeper 底层其实只提供了两个功能：①管理（存储、读取）用户程序提交的数据；②为用户程序提交数据节点监听服务。

# zookeeper &ZAB 协议&Paxos算法

## ZAB 协议&Paxos算法

Paxos 算法应该可以说是 ZooKeeper 的灵魂了。但是，ZooKeeper 并没有完全采用 Paxos算法 ，而是使用 ZAB 协议作为其保证数据一致性的核心算法。另外，在ZooKeeper的官方文档中也指出，ZAB协议并不像 Paxos 算法那样，是一种通用的分布式一致性算法，它是一种特别为Zookeeper设计的崩溃可恢复的原子消息广播算法。

## ZAB 协议介绍

ZAB（ZooKeeper Atomic Broadcast 原子广播） 协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。 在 ZooKeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性，基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间的数据一致性。

### 崩溃恢复和消息广播

ZAB协议包括两种基本的模式，分别是 崩溃恢复和消息广播。当整个服务框架在启动过程中，或是当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB 协议就会进人恢复模式并选举产生新的Leader服务器。当选举产生了新的 Leader 服务器，同时集群中已经有过半的机器与该Leader服务器完成了状态同步之后，ZAB协议就会退出恢复模式。其中，所谓的状态同步是指数据同步，用来保证集群中存在过半的机器能够和Leader服务器的数据状态保持一致。

当集群中已经有过半的Follower服务器完成了和Leader服务器的状态同步，那么整个服务框架就可以进人消息广播模式了。 当一台同样遵守ZAB协议的服务器启动后加人到集群中时，如果此时集群中已经存在一个Leader服务器在负责进行消息广播，那么新加人的服务器就会自觉地进人数据恢复模式：找到Leader所在的服务器，并与其进行数据同步，然后一起参与到消息广播流程中去。正如上文介绍中所说的，ZooKeeper设计成只允许唯一的一个Leader服务器来进行事务请求的处理。Leader服务器在接收到客户端的事务请求后，会生成对应的事务提案并发起一轮广播协议；而如果集群中的其他机器接收到客户端的事务请求，那么这些非Leader服务器会首先将这个事务请求转发给Leader服务器。

# dubbo和zookeeper实例

## 创建Common工程

com.xmm.dubbo.api.StudentRemoteService

|  |
| --- |
| public interface StudentRemoteService {  ResultEntity<String> saveStudentRemote(Student student);  ResultEntity<List<Student>> getStudentList(); } |

com.xmm.dubbo.entity.Student.java

该类必须要实现序列化接口，因为该类的对象会在网络上进行传输，实现序列化接口可以使其以二进制的形式进行传输。

|  |
| --- |
| public class Student implements Serializable {  private static final long *serialVersionUID* = 1L;  private String stuNumber;  private String stuName;  private String stuAge;  private Date birthday;  private List<String> subjects;  } |

com.xmm.dubbo.special.ResultEntity

|  |
| --- |
| /\* 为了让项目开发过程中代码更加规范，团队协作更加顺滑，声明ResultEntity类，  \* 作用是：所有Ajax请求返回的结果都是ResultEntity类型  \* 附加作用：Consumer调用Provider时统一返回类型\*/ public class ResultEntity<T> implements Serializable {  private static final long *serialVersionUID* = 1L;  public static final String *NO\_MESSAGE* = "NO\_MESSAGE";  public static final String *SUCCESS* = "SUCCESS";  public static final String *FAILED* = "FAILED";  // 响应结果中包含的数据  private T data;  // 请求处理的结果是成功还是失败  private String result;  // 请求处理失败时的提示消息  private String message;  public ResultEntity() {  }  public ResultEntity(T data, String result, String message) {  super();  this.data = data;  this.result = result;  this.message = message;  }  public static <T> ResultEntity<T> successWithoutData() {  return new ResultEntity<>(null, *SUCCESS*, *NO\_MESSAGE*);  }  public static <T> ResultEntity<T> successWithData(T data) {  return new ResultEntity<>(data, *SUCCESS*, *NO\_MESSAGE*);  }  public static <T> ResultEntity<T> failed(String message) {  return new ResultEntity<>(null, *FAILED*, message);  }  public T getData() {  return data;  }  public void setData(T data) {  this.data = data;  }  public String getResult() {  return result;  }  public void setResult(String result) {  this.result = result;  }  public String getMessage() {  return message;  }  public void setMessage(String message) {  this.message = message;  }  @Override  public String toString() {  return "ResultEntity [data=" + data + ", result=" + result + ", message=" + message + "]";  } } |

URLEncodeAndDecode

该类是为了将zookeeper中的节点的值进行UTF-8的转码，使其能够阅读。

|  |
| --- |
| public class URLEncodeAndDecode {  public static void main(String[] args) throws UnsupportedEncodingException {  // Zookeeper中存储的信息是UTF-8编码以后的结果  String providerRegistryInfo = "dubbo%3A%2F%2F192.168.2.146%3A20880%2Fcom.atguigu.dubbo.api.StudentRemoteService%3Fanyhost%3Dtrue%26application%3Datguigu-dubbo-provider%26dubbo%3D2.5.5%26generic%3Dfalse%26interface%3Dcom.atguigu.dubbo.api.StudentRemoteService%26methods%3DgetStudentList%2CsaveStudentRemote%26pid%3D22136%26revision%3DCommon%26side%3Dprovider%26timestamp%3D1562900043268";  // 使用UTF-8解码查看原始数据  String decode = URLDecoder.*decode*(providerRegistryInfo, "UTF-8");  // 打印原始数据  System.*out*.println(decode);  String consumerRegistryInfo = "consumer%3A%2F%2F192.168.2.146%2Fcom.atguigu.dubbo.api.StudentRemoteService%3Fapplication%3Datguigu-dubbo-consumer%26category%3Dconsumers%26check%3Dfalse%26dubbo%3D2.5.5%26interface%3Dcom.atguigu.dubbo.api.StudentRemoteService%26methods%3DgetStudentList%2CsaveStudentRemote%26pid%3D21280%26revision%3D0.0.1-SNAPSHOT%26side%3Dconsumer%26timestamp%3D1562901949185";  decode = URLDecoder.*decode*(consumerRegistryInfo, "UTF-8");  System.*out*.println(decode);  } } |

## 创建Provider工程

com.xmm.dubbo.api.impl.StudentRemoteServiceImpl

|  |
| --- |
| public class StudentRemoteServiceImpl implements StudentRemoteService {  @Override  public ResultEntity<String> saveStudentRemote(Student student) {  System.*err*.println(student);  return ResultEntity.*successWithoutData*();  }  @Override  public ResultEntity<List<Student>> getStudentList() {  List<Student> students = new ArrayList<>();  students.add(new Student("553792", "justin", "25",   new Date(), Arrays.*asList*("Java","PHP","MySQL")));  students.add(new Student("690812", "harry", "20",   new Date(), Arrays.*asList*("Oracel","Python","Ruby")));  students.add(new Student("479018", "jetty", "35",   new Date(), Arrays.*asList*("C","go","Js")));  return ResultEntity.*successWithData*(students);  } } |

src/main/resources/spring-dubbo.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://code.alibabatech.com/schema/dubbo"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">  <!-- 配置当前服务名称，便于在注册中心注册 -->  <dubbo:application name="xmm-dubbo-provider"/>  <!-- 指定当前使用的注册中心 -->  <dubbo:registry protocol="zookeeper" address="192.168.235.23" port="2181"/>  <!-- 配置当前Provider服务对外暴露的端口号 -->  <dubbo:protocol name="dubbo" port="20880"/>  <!-- 配置对外暴露的接口实现类的bean -->  <bean id="studentRemoteService"  class="com.xmm.dubbo.api.impl.StudentRemoteServiceImpl"/>  <!-- 将接口和实现类关联起来 -->  <dubbo:service interface="com.xmm.dubbo.api.StudentRemoteService"  ref="studentRemoteService"/> </beans> |

src/main/resources/log4j.properties

|  |
| --- |
| log4j.rootLogger=INFO,myConsole log4j.appender.myConsole=org.apache.log4j.ConsoleAppender log4j.appender.myConsole.Target=System.out log4j.appender.myConsole.layout=org.apache.log4j.PatternLayout log4j.appender.myConsole.layout.ConversionPattern=[%-5p] %d(%r) --> [%t] %l: %m %x %n |

web/WEB-INF/web.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"  version="4.0">  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring-dubbo.xml</param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener> </web-app> |

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.xmm.learning</groupId>  <artifactId>DubboProvider</artifactId>  <version>1.0-SNAPSHOT</version>  <dependencies>  <groupId>com.xmm.learning</groupId>  <artifactId>DubboCommon</artifactId>  <version>1.0-SNAPSHOT</version>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>dubbo</artifactId>  <version>2.5.5</version>  </dependency>  <dependency>  <groupId>cglib</groupId>  <artifactId>cglib</artifactId>  <version>2.2</version>  </dependency>  <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.6.8</version>  </dependency>  <dependency>  <groupId>com.github.sgroschupf</groupId>  <artifactId>zkclient</artifactId>  <version>0.1</version>  </dependency>  <dependency>  </dependencies> </project> |

创建tomcat服务器，修改tomcat服务器的端口号避免冲突，然后将Provider工程部署到tomcat服务器上并开启该tomcat服务器。

## 创建Consumer工程

com.xmm.dubbo.handler.StudentHandler

|  |
| --- |
| @Controller public class StudentHandler {  // 装配封装远程方法功能的接口  @Autowired  private StudentRemoteService studentRemoteService;  @RequestMapping("/save/student")  public String saveStudent(  @RequestParam("stuNumber") String stuNumber,   @RequestParam("stuName") String stuName,   Model model) {  System.*err*.println(studentRemoteService);  Student student = new Student();  student.setStuNumber(stuNumber);  student.setStuName(stuName);  // 调用远程方法（像调用本地方法一样）  ResultEntity<String> resultEntity = studentRemoteService.saveStudentRemote(student);  // 将调用结果存入模型  model.addAttribute("resultEntity", resultEntity);  return "target";  }  @RequestMapping("/get/student/list")  public String getStudentList(Model model) {  System.*err*.println(studentRemoteService);  ResultEntity<List<Student>> studentList = studentRemoteService.getStudentList();  model.addAttribute("studentList", studentList);  return "target";  } } |

src/main/resources/spring-mvc.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:context="http://www.springframework.org/schema/context"  xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"  xmlns:mvc="http://www.springframework.org/schema/mvc"  xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd  http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">  <!-- SpringMVC相关配置 -->  <context:component-scan  base-package="com.xmm.dubbo.handler" />  <bean id="viewResolver"  class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/WEB-INF/" />  <property name="suffix" value=".jsp" />  </bean>  <mvc:annotation-driven />  <mvc:default-servlet-handler />  <!-- Dubbo配置：当前应用名称 -->  <dubbo:application name="xmm-dubbo-consumer" />  <!-- Dubbo配置：指定注册中心位置 -->  <dubbo:registry protocol="zookeeper"  address="192.168.235.23" port="2181" />  <!-- Dubbo配置：引用Provider提供的可以远程调用的服务 -->  <dubbo:reference interface="com.xmm.dubbo.api.StudentRemoteService" id="studentRemoteService"/> </beans> |

web.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"  version="4.0">  <servlet>  <servlet-name>springDispatcherServlet</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring-mvc.xml</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>springDispatcherServlet</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping> </web-app> |

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.xmm.learning</groupId>  <artifactId>DubboConsumer</artifactId>  <version>1.0-SNAPSHOT</version>  <dependencies>  <dependency>  <groupId>com.xmm.learning</groupId>  <artifactId>DubboProvider</artifactId>  <version>1.0-SNAPSHOT</version>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>dubbo</artifactId>  <version>2.5.5</version>  </dependency>  <dependency>  <groupId>cglib</groupId>  <artifactId>cglib</artifactId>  <version>2.2</version>  </dependency>  <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.6.8</version>  </dependency>  <dependency>  <groupId>com.github.sgroschupf</groupId>  <artifactId>zkclient</artifactId>  <version>0.1</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>4.3.10.RELEASE</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>4.0.0</version>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet.jsp</groupId>  <artifactId>jsp-api</artifactId>  <version>2.1</version>  <scope>provided</scope>  </dependency>  </dependencies> </project> |

创建tomcat服务器，修改tomcat服务器的端口号避免冲突，然后将Consumer工程部署到tomcat服务器上并开启该tomcat服务器。