

Zookeeper讲义

教学目标

- 了解zookeeper
- 了解zookeeper的应用场景
- 了解zookeeper的基本概念和数据模型
- 能够搭建和配置zookeeper
- 熟练操作zookeeper服务端和客户端命令
- 能够使用java api 操作zookeeper
- 理解zookeeper watch机制
- 能搭建zookeeper集群

1、介绍zookeeper

【目标】

1：了解Zookeeper的概念

2：了解分布式的概念

【路径】

1：Zookeeper概述

2：Zookeeper的发展历程

3：什么是分布式

4：Zookeeper的应用场景

【讲解】

1.1、zookeeper概述

ZooKeeper从字面意思理解，【Zoo - 动物园，Keeper - 管理员】动物园中有很多种动物，这里的动物就可以比作分布式环境下多种多样的服务，而ZooKeeper做的就是管理这些服务。

Apache ZooKeeper的系统为分布式协调是构建分布式应用的高性能服务。

ZooKeeper 本质上是一个分布式的小文件存储系统。提供基于类似于文件系统的目录树方式的数据存储，并且可以对树中的节点进行有效管理。从而用来维护和监控你存储的数据的状态变化。通过监控这些数据状态的变化，从而可以达到基于数据的集群管理。

ZooKeeper 适用于存储和协同相关的关键数据，不适合用于大数据量存储。

1.2、zookeeper的发展历程

ZooKeeper 最早起源于雅虎研究院的一个研究小组。当时研究人员发现，在雅虎内部很多大型系统基本都需要依赖一个系统来进行分布式协同，但是这些系统往往都存在分布式单点问题。

所以，雅虎的开发人员就开发了一个通用的无单点问题的分布式协调框架，这就是ZooKeeper。

ZooKeeper之后在开源界被大量使用，很多著名开源项目都在使用zookeeper，例如：

- Hadoop：使用ZooKeeper 做Namenode 的高可用。
- HBase：保证集群中只有一个master，保存hbase:meta表的位置，保存集群中的RegionServer列表。

- Kafka：集群成员管理，controller 节点选举。

1.3、什么是分布式

1.3.1、集中式系统

集中式系统，集中式系统中整个项目就是一个独立的应用，整个应用也就是整个项目，所有的东西都在一个应用里面。部署到一个服务器上。

1.3.2、分布式系统

随着公司的发展，应用的客户变多，功能也日益完善，加了很多的功能，整个项目在一个tomcat上跑，tomcat说它也很累，能不能少跑点代码，这时候就产生了。我们可以把大项目按功能划分为很多的模块，比如说单独一个系统处理订单，一个处理用户登录，一个处理后台等等，然后每一个模块都单独在一个tomcat中跑，合起来就是一个完整的大项目，这样每一个tomcat都非常轻松。



分布式系统的描述总结是：

- 多台计算机构成
- 计算机之间通过网络进行通信
- 彼此进行交互
- 共同目标

1.4、zookeeper的应用场景

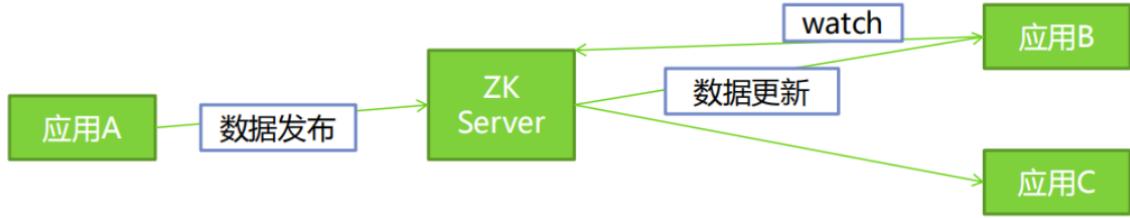
1.4.1、注册中心

分布式应用中，通常需要有一套完整的命名规则，既能够产生唯一的名称又便于人识别和记住，通常情况下用树形的名称结构是一个理想的选择，树形的名称结构是一个有层次的目录结构。通过调用Zookeeper提供的创建节点的API，能够很容易创建一个全局唯一的path，这个path就可以作为一个名称。

阿里巴巴集团开源的分布式服务框架Dubbo中使用ZooKeeper来作为其命名服务，维护全局的服务地址列表。

1.4.2、配置中心

数据发布/订阅即所谓的配置中心：发布者将数据发布到ZooKeeper一系列节点上面，订阅者进行数据订阅，当数据有变化时，可以及时得到数据的变化通知，达到动态获取数据的目的。



ZooKeeper 采用的是推拉结合的方式。

- 1、推: 服务端会推给注册了监控节点的客户端 Watcher 事件通知
- 2、拉: 客户端获得通知后, 然后主动到服务端拉取最新的数据

1.4.3、分布式锁（了解）

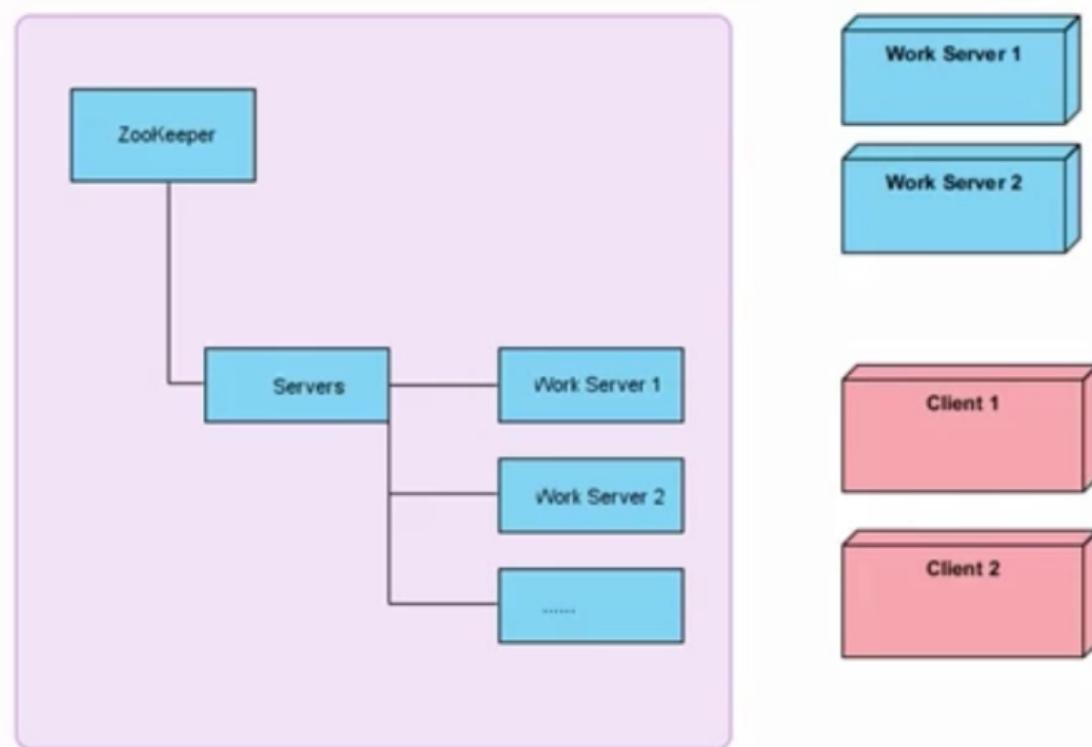
分布式锁是控制分布式系统之间同步访问共享资源的一种方式。在分布式系统中, 常常需要协调他们的动作。如果不同的系统或是同一个系统的不同主机之间共享了一个或一组资源, 那么访问这些资源的时候, 往往需要互斥来防止彼此干扰来保证一致性, 在这种情况下, 便需要使用到分布式锁。

1.4.4、分布式队列（了解）

在传统的单进程编程中, 我们使用队列来存储一些数据结构, 用来在多线程之间共享或传递数据。分布式环境下, 我们同样需要一个类似单进程队列的组件, 用来实现跨进程、跨主机、跨网络的数据共享和数据传递, 这就是我们的分布式队列。

1.4.5、负载均衡

负载均衡是通过负载均衡算法, 用来把对某种资源的访问分摊给不同的设备, 从而减轻单点的压力。



上图中左侧为ZooKeeper集群, 右侧上方为工作服务器, 下面为客户端。每台工作服务器在启动时都会去ZooKeeper的servers节点下注册临时节点, 每台客户端在启动时都会去servers节点下取得所有可用的工作服务器列表, 并通过一定的负载均衡算法计算得出一台工作服务器, 并与之建立网络连接

【小结】

2: Zookeeper的发展历程

3: 什么是分布式

4: Zookeeper的应用场景

2、zookeeper环境搭建

【目标】

1: 在window上安装Zookeeper

【路径】

1: 安装Zookeeper的前提

2: 在window上安装Zookeeper

3: zookeeper基本操作

【讲解】

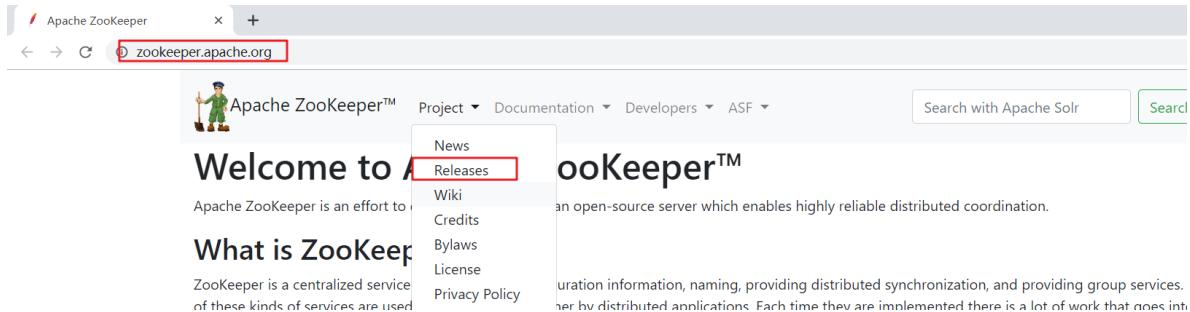
2.1、前提

必须安装jdk 1.8，配置jdk环境变量，步骤略

2.2、安装zookeeper

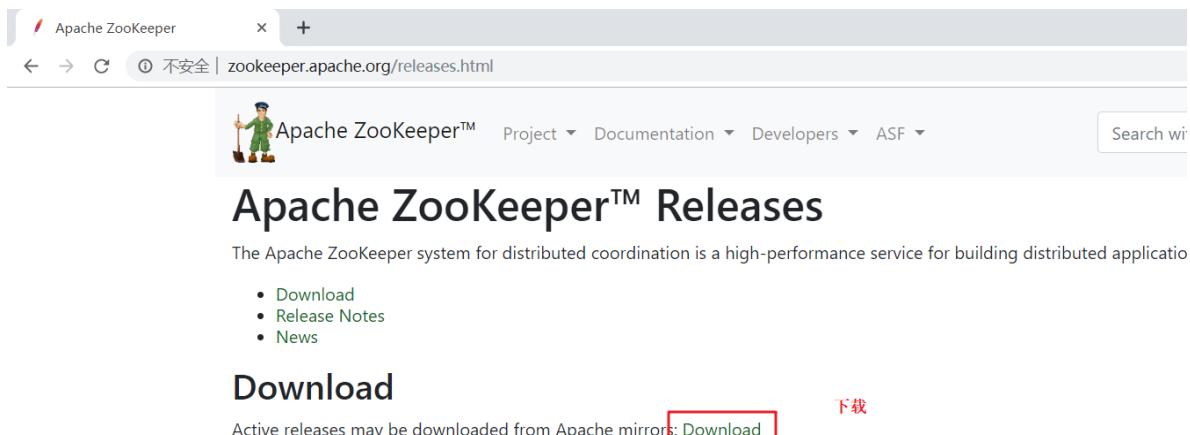
2.2.1、下载

下载地址: <http://zookeeper.apache.org>



The screenshot shows the Apache ZooKeeper homepage. The URL in the address bar is 'zookeeper.apache.org'. The page features a cartoon character at the top left. A navigation bar at the top includes 'Project', 'Documentation', 'Developers', and 'ASF'. A dropdown menu is open over the 'Project' link, with 'Releases' highlighted by a red box. Other options in the menu include 'News', 'Wiki', 'Credits', 'Bylaws', 'License', and 'Privacy Policy'. To the right of the menu, the text 'ooKeeper™' is displayed, followed by a brief description of the project as an open-source server for distributed coordination. A search bar is located at the top right.

查看zookeeper的更新历史



The screenshot shows the 'Apache ZooKeeper Releases' page. The URL in the address bar is 'zookeeper.apache.org/releases.html'. The page title is 'Apache ZooKeeper™ Releases'. It contains a brief introduction about the system being a high-performance service for building distributed applications. Below this, there is a list of download links: 'Download', 'Release Notes', and 'News'. At the bottom of the page, there is a large 'Download' button with a red border and the Chinese character '下载' (Download) in red. A note below the button states 'Active releases may be downloaded from Apache mirrors: [Download](#)'.

zookeeper下载页的地址

We suggest the following mirror site for your download:
<http://mirrors.tuna.tsinghua.edu.cn/apache/zookeeper/>

Other mirror sites are suggested below.

It is essential that you verify the integrity of the downloaded file using the PGP signature ([.asc](#) file) or a hash ([.md5](#) or [.sha](#) file).

Please only use the backup mirrors to download KEYS, PGP signatures and hashes (SHA* etc) -- or if no other mirrors are working.

HTTP
<http://mirrors.tuna.tsinghua.edu.cn/apache/zookeeper/>

zookeeper选择下载的版本

Please make sure you're downloading from [a nearby mirror site](#), not from www.apache.org.

We suggest downloading the current [stable](#) release.

Older releases are available from the [archives](#). 点击archives 可以查看历史版本

Name	Last modified	Size	Description
Parent Directory	-	-	
current/	2019-10-16 08:35	-	
stable/	2019-10-16 08:35	-	
zookeeper-3.4.14/	2019-04-01 22:45	-	
zookeeper-3.5.5/	2019-05-20 18:41	-	
zookeeper-3.5.6/	2019-10-16 08:35	-	

这些是较新的版本

2.2.2、解压

解压到没有中文路径的目录下（不要出现中文和空格）

名称	修改日期	类型	大小
tmp	2019/11/3 10:51	文件夹	
tomcat-dubbo-admin	2019/4/14 19:11	文件夹	
xp oracle	2019/2/13 7:52	文件夹	
zookeeper-3.4.13	2019/6/5 14:00	文件夹	

2.2.3、修改配置文件

在zookeeper路径下创建一个data目录

名称	修改日期	类型
bin	2019/8/7 13:36	文件夹
conf	2019/8/8 10:10	文件夹
contrib	2019/6/3 22:53	文件夹
data	2019/8/7 13:35	文件夹
dist-maven	2019/6/3 22:53	文件夹

修改配置文件

conf路径中复制一份zoo_sample.cfg,改名为 zoo.cfg

名称	修改日期	类型	大小
configuration.xsl	2018/6/30 1:04	XSL 样式表	1 KB
log4j.properties	2018/6/30 1:04	PROPERTIES 文件	3 KB
zoo.cfg	2019/8/7 13:25	CFG 文件	1 KB
zoo.cfg.bak	2019/8/7 13:25	BAK 文件	1 KB
zoo_sample.cfg	2019/8/7 13:25	CFG 文件	1 KB

指定保存数据的目录: data目录

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=../data
dataLogDir=../log
# the port at which the clients will connect
clientPort=2181
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
#
# Be sure to read the maintenance section of the
# administrator guide before turning on autopurge.
```

如果需要日志，可以创建log文件夹，指定dataLogDir属性

2.2.4、启动zookeeper

打开bin路径，双击zkServer.cmd,启动zookeeper服务

名称	修改日期	类型	大小
README.txt	2018/6/30 1:04	文本文档	1 KB
zkCleanup.sh	2018/6/30 1:04	Shell Script	2 KB
zkCli.cmd	2018/6/30 1:04	Windows 命令脚本	2 KB
zkCli.sh	2018/6/30 1:04	Shell Script	2 KB
zkEnv.cmd	2018/6/30 1:04	Windows 命令脚本	2 KB
zkEnv.sh	2018/6/30 1:04	Shell Script	3 KB
zkServer.cmd	2018/6/30 1:04	Windows 命令脚本	2 KB
zkServer.sh	2018/6/30 1:04	Shell Script	7 KB
zkTxnLogToolkit.cmd	2018/6/30 1:04	Windows 命令脚本	1 KB
zkTxnLogToolkit.sh	2018/6/30 1:04	Shell Script	2 KB

2.2.5、启动客户端测试

启动客户端，看到 'Welcome to Zookeeper!' 说明成功

```
C:\WINDOWS\system32\cmd.exe
0_202\bin;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files (x86)\MySQL\MySQL Server 5.5\bin;F:\apache-maven-3.3.9\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\WINDOWS\System32\OpenSSH;E:\tools\qq\Bin;C:\Program Files\Git\cmd;C:\Program Files\TortoiseGit\bin;C:\Users\sun\AppData\Local\Microsoft\WindowsApps;C:\Program Files (x86)\SSH Communications Security\SSH Secure Shell;
2019-11-04 09:24:23,032 [myid:] - INFO  [main:Environment@100] - Client environment:java.io.tmpdir=C:\Users\sun\AppData\Local\Temp\Welcome to ZooKeeper!
2019-11-04 09:24:23,032 [myid:] - INFO  [main:Environment@100] - Client environment:java.compiler=<NA>
2019-11-04 09:24:23,032 [myid:] - INFO  [main:Environment@100] - Client environment:os.name=Windows 10
2019-11-04 09:24:23,032 [myid:] - INFO  [main:Environment@100] - Client environment:os.arch=amd64
2019-11-04 09:24:23,032 [myid:] - INFO  [main:Environment@100] - Client environment:os.version=10.0
2019-11-04 09:24:23,032 [myid:] - INFO  [main:Environment@100] - Client environment:user.name=sun
2019-11-04 09:24:23,032 [myid:] - INFO  [main:Environment@100] - Client environment:user.home=C:\Users\sun
2019-11-04 09:24:23,032 [myid:] - INFO  [main:Environment@100] - Client environment:user.dir=F:\zookeeper-3.4.13\bin
2019-11-04 09:24:23,032 [myid:] - INFO  [main:ZooKeeper@442] - Initiating client connection, connectString=localhost:2181
sessionTimeout=30000 watcher=org.apache.zookeeper.ZooKeeperMain$MyWatcher@69d0a921
WATCHER:::
WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0]
```

【小结】

- 1: 安装Zookeeper的前提
- 2: 在window上安装Zookeeper

3、zookeeper基本操作

【目标】

- 1: Zookeeper的客户端命令
- 2: Zookeeper的java的api操作

【路径】

- 1: Zookeeper的数据结构
- 2: 节点的分类
 - 持久性
 - 临时性
- 3: 客户端命令 (创建、查询、修改、删除)
- 4: Zookeeper的java的api介绍 (创建、查询、修改、删除)
- 5: Zookeeper的watch机制
 - NodeCache
 - PathChildrenCache
 - TreeCache

【讲解】

3.1、zookeeper数据结构

ZooKeeper 的数据模型是层次模型。层次模型常见于文件系统。例如：我的电脑可以分为多个盘符（例如C、D、E等），每个盘符下可以创建多个目录，每个目录下面可以创建文件，也可以创建子目录，最终构成了一个树型结构。通过这种树型结构的目录，我们可以将文件分门别类的进行存放，方便我们后期查找。而且磁盘上的每个文件都有一个唯一的访问路径，例如：

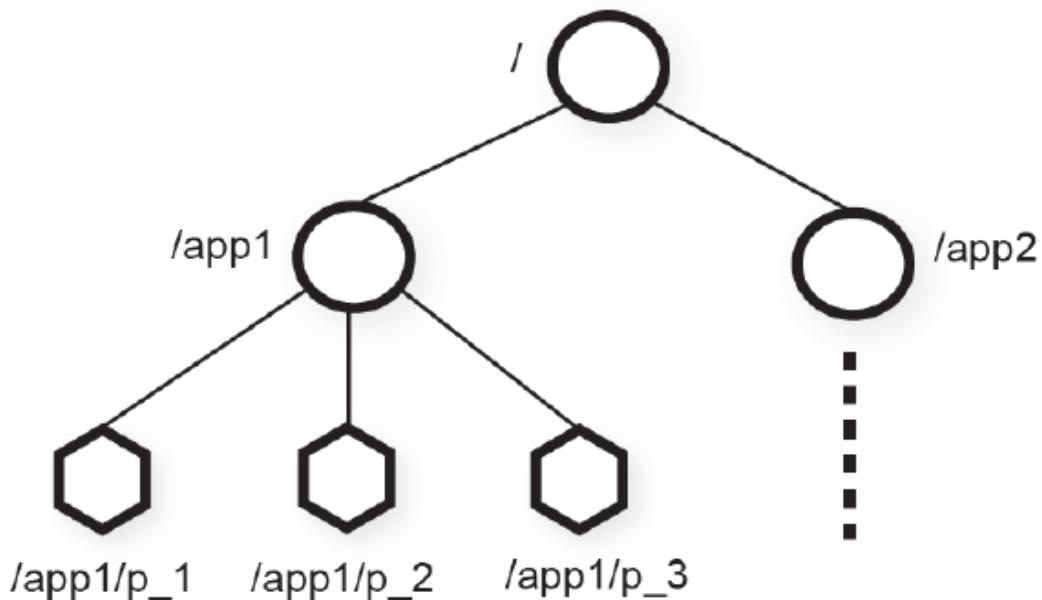
C:\Windows\itcast\hello.txt。

- ✓ 我的电脑
- ✓ Windows (C:)
 - > Program Files
 - > Program Files (x86)
 - > ProgramData
 - Recovery
 - > Windows
 - > 用户
- > LENOVO (D:)

层次模型和key-value 模型是两种主流的数据模型。ZooKeeper 使用文件系统模型主要基于以下两点考虑：

1. 文件系统的树形结构便于表达数据之间的层次关系。
2. 文件系统的树形结构便于为不同的应用分配独立的命名空间（namespace）。

ZooKeeper 的层次模型称作data tree。Datatree 的每个节点叫作znode (Zookeeper node)。不同于文件系统，每个节点都可以保存数据。每个节点都有一个版本(version)。版本从0 开始计数。



如图所示，data tree中有两个子树，用于应用1(/app1)和应用2 (/app2)。

每个客户端进程pi 创建一个znode节点 p_i 在 /app1下， /app1/p_1就代表一个客户端在运行。

3.2、节点的分类

一：一个znode可以是持久性的，也可以是临时性的

1. 持久性znode[PERSISTENT]，这个znode一旦创建不会丢失，无论是zookeeper宕机，还是client宕机。
2. 临时性的znode[EPHEMERAL]，如果zookeeper宕机了，或者client在指定的timeout时间内没有连接server，都会被认为丢失。

二：znode也可以是顺序性的，每一个顺序性的znode关联一个唯一的单调递增整数。这个单调递增整数是znode名字的后缀。

3. 持久顺序性的znode(PERSISTENT_SEQUENTIAL):znode 处理具备持久性的znode的特点之外，znode的名称具备顺序性。
4. 临时顺序性的znode(EPHEMERAL_SEQUENTIAL):znode处理具备临时性的znode特点， znode的名称具备顺序性。

3.2、客户端命令

3.2.1、查询所有命令

```
help
```

```
[zk: localhost:2181(CONNECTED) 0] help [帮助] 查询zookeeper所有命令
ZooKeeper -server host:port cmd args
    stat path [watch]
    set path data [version]
    ls path [watch]
    delquota [-n|-b] path
    ls2 path [watch]
    setAcl path acl
    setquota -n|-b val path
    history
    redo cmdno
    printwatches on|off
    delete path [version]
    sync path
    listquota path
    rmr path
    get path [watch]
    create [-s] [-e] path data acl
    addauth scheme auth
    quit
    getAcl path
    close
    connect host:port
```

3.2.2、查询跟路径下的节点

```
ls /zookeeper
```

查看zookeeper节点

```
[zk: localhost:2181(CONNECTED) 1] ls / [帮助] 查询根路径下的节点
[dubbo, zookeeper]
[zk: localhost:2181(CONNECTED) 2] ls /zookeeper [帮助] 查询zookeeper节点下的节点
[quota]
```

3.2.3、创建普通永久节点

```
create /app1 "helloworld"
```

创建app1节点，值为helloworld

```
[zk: localhost:2181(CONNECTED) 3] create /app1 "helloworld" [帮助] 创建 /app1节点，对应的value 'helloworld'
Created /app1
```

3.2.4、创建带序号永久节点

```
create -s /hello "helloworld"
```

```
[zk: localhost:2181(CONNECTED) 0] create -s /hello 'hello' 创建带有需要的永久性节点 hello, 会自动添加序号
Created /hello0000000006
[zk: localhost:2181(CONNECTED) 1] create -s /hello 'hello'
Created /hello0000000007
[zk: localhost:2181(CONNECTED) 2] ls /
[dubbo, zookeeper, app1, hello0000000006, hello0000000007] 执行两次命令, 每次创建的节点序号不同
[zk: localhost:2181(CONNECTED) 4] create /hello 'hello'
Created /hello
[zk: localhost:2181(CONNECTED) 5] create -s /hello/a 'a' 以下是在hello节点中创建两个带有序号的a节点
Created /hello/a000000000
[zk: localhost:2181(CONNECTED) 6] create -s /hello/a 'aa'
Created /hello/a000000001
[zk: localhost:2181(CONNECTED) 7] ls /hello
[a000000000, a000000001]
[zk: localhost:2181(CONNECTED) 8]
```

3.2.5、创建普通临时节点

```
create -e /app3 'app3'
```

-e:表示普通临时节点

```
[zk: localhost:2181(CONNECTED) 8] create -e /app3 'app3' 创建临时的节点
Created /app3
```

关闭客户端, 再次打开查看 app3节点消失

3.2.6、创建带序号临时节点

```
create -e -s /app4 'app4'
```

-e:表示普通临时节点

-s:表示带序号节点

```
[zk: localhost:2181(CONNECTED) 1] create -e -s /app4 'app4'
Created /app40000000011
```

关闭客户端, 再次打开查看 app4节点消失

3.2.7、查询节点数据

```
get /app1
```

```
[zk: localhost:2181(CONNECTED) 12] get /app1
helloworld
cZxid = 0xcf
ctime = Mon Nov 04 10:07:16 CST 2019
mZxid = 0xcf
mtime = Mon Nov 04 10:07:16 CST 2019
pZxid = 0xcf
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 10
numChildren = 0
[zk: localhost:2181(CONNECTED) 13]
```

```
# .....节点的状态信息, 也称为stat结构体.....
# 创建该znode的事务的zxid(ZooKeeper Transaction ID)
# 事务ID是ZooKeeper为每次更新操作/事务操作分配一个全局唯一的id, 表示zxid, 值越小, 表示越先执行
cZxid = 0x4454 # 0x0表示十六进制数
```

```
ctime = Thu Jan 01 08:00:00 CST 1970 # 创建时间
mZxid = 0x4454 # 最后一次更新的zxid
mtime = Thu Jan 01 08:00:00 CST 1970 # 最后一次更新的时间
pZxid = 0x4454 # 最后更新的子节点的zxid
cversion = 5 # 子节点的变化号, 表示子节点被修改的次数
dataVersion = 0 # 表示当前节点的数据变化号, 0表示当前节点从未被修改过
aclVersion = 0 # 访问控制列表的变化号 access control list
# 如果临时节点, 表示当前节点的拥有者的sessionid
ephemeralOwner = 0x0 # 如果不是临时节点, 则值为0
dataLength = 13 # 数据长度
numChildren = 1 # 子节点的数量
```

3.2.8、修改节点数据

```
set /app1 'hello'
```

```
[zk: localhost:2181(CONNECTED) 13] set /app1 'hello'
cZxid = 0xcf
ctime = Mon Nov 04 10:07:16 CST 2019
mZxid = 0xf3
mtime = Mon Nov 04 11:22:56 CST 2019
pZxid = 0xcf
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

3.2.9、删除节点

```
delete /hello000000000006
```

```
[zk: localhost:2181(CONNECTED) 14] ls /
[zookeeper, dubbo, hello, app1, hello0000000006, hello0000000007]
[zk: localhost:2181(CONNECTED) 15] delete /hello0000000006
[zk: localhost:2181(CONNECTED) 16] ls /
[zookeeper, dubbo, hello, app1, hello0000000007]
```

3.2.10、递归删除节点

```
delete /hello
```

```
rmr /hello
```

```
[zk: localhost:2181(CONNECTED) 16] ls /
[zookeeper, dubbo, hello, app1, hello0000000007]      查询所有节点
[zk: localhost:2181(CONNECTED) 17] ls /hello          查询/hello下的节点
[a0000000000, a0000000001]
[zk: localhost:2181(CONNECTED) 18] delete /hello      使用delete命令删除/hello节点, 提示节点非空
Node not empty: /hello
[zk: localhost:2181(CONNECTED) 19] rmr /hello        使用rmr命令删除 /hello节点, 再次查询, 消失
[zk: localhost:2181(CONNECTED) 20] ls /
[zookeeper, dubbo, app1, hello0000000007]
```

3.2.11、查看节点状态

```
stat /zookeeper
```

```
[zk: localhost:2181(CONNECTED) 22] stat /zookeeper
cZxid = 0x0
ctime = Thu Jan 01 08:00:00 CST 1970
nZxid = 0x0
ntime = Thu Jan 01 08:00:00 CST 1970
pZxid = 0x0
version = -1
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 0
numChildren = 1
```

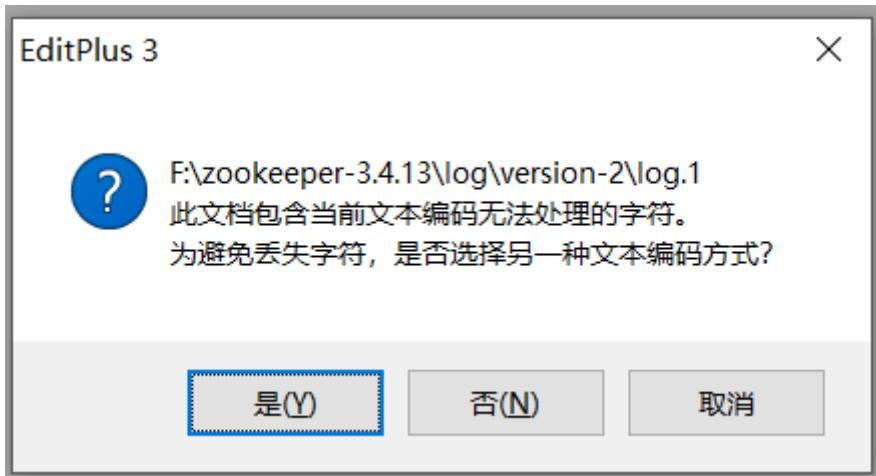
3.4.12、日志的可视化

- 这是日志的存储路径

此电脑 > teach (F:) > zookeeper-3.4.13 > log > version-2

名称	修改日期	类型	大小
log.1	2019/11/3 11:03	1 文件	65,537 KB
log.1e	2019/11/3 13:10	1E 文件	65,537 KB
log.4d7	2019/11/8 10:59	4D7 文件	65,537 KB
log.13d	2019/11/6 18:23	13D 文件	65,537 KB
log.20e	2019/11/7 15:53	20E 文件	65,537 KB
log.30	2019/11/3 18:27	30 文件	65,537 KB
log.ca	2019/11/4 12:04	CA 文件	65,537 KB
log.ff	2019/11/4 17:29	FF 文件	65,537 KB

- 日志都是以二进制文件存储的，使用记事本打开，无意义。



- 为了能正常查看日志，把查看日志需要的jar包放到统一路径下

此电脑 > teach (F:) > zookeeper-3.4.13 > log > version-2

名称	修改日期	类型	大小
log.1	2019/11/3 11:03	1 文件	65,537 KB
log.1e	2019/11/3 13:10	1E 文件	65,537 KB
log.4d7	2019/11/8 10:59	4D7 文件	65,537 KB
log.13d	2019/11/6 18:23	13D 文件	65,537 KB
log.20e	2019/11/7 15:53	20E 文件	65,537 KB
log.30	2019/11/3 18:27	30 文件	65,537 KB
log.ca	2019/11/4 12:04	CA 文件	65,537 KB
log.ff	2019/11/4 17:29	FF 文件	65,537 KB
slf4j-api-1.7.25.jar	2018/6/30 1:04	Executable Jar File	41 KB
zookeeper-3.4.13.jar	2018/6/30 1:05	Executable Jar File	1,474 KB

- 使用命令可以直接查看正常日志

```
java -classpath ".;*" org.apache.zookeeper.server.LogFormatter log.1
```

```
F:\zookeeper-3.4.13\log\version-2>java -classpath ". ;*" org.apache.zookeeper.server.LogFormatter log.1
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
ZooKeeper Transactional Log File with dbid 0 txnlog format version 2
19-11-3 上午11时03分02秒 session 0x100059a7c8f0000 cxid 0x0 zxid 0x1 createSession 30000
19-11-3 上午11时03分38秒 session 0x100059a7c8f0000 cxid 0x0 zxid 0x2 closeSession null
19-11-3 上午11时03分38秒 session 0x100059a7c8f0001 cxid 0x0 zxid 0x3 createSession 40000
19-11-3 上午11时03分38秒 session 0x100059a7c8f0001 cxid 0x4 zxid 0x4 create '/dubbo,#3137322e31362e31372e323434,v{s{31,s{'world,'anyone'}}},F,1
19-11-3 上午11时03分38秒 session 0x100059a7c8f0001 cxid 0x5 zxid 0x5 create '/dubbo/com.itheima.service.CheckGroupService,#3137322e31362e31372e323434,v{s{31,s{'world,'anyone'}}},F,1
19-11-3 上午11时03分38秒 session 0x100059a7c8f0001 cxid 0x6 zxid 0x6 create '/dubbo/com.itheima.service.CheckGroupService/consumers,#3137322e31362e31372e323434,v{s{31,s{'world,'anyone'}}},F,1
```

3.3、zookeeper 的java Api介绍

3.3.1、ZooKeepe常用Java API

- **原生Java API (不推荐使用)**

ZooKeeper 原生Java API位于org.apache.ZooKeeper包中

ZooKeeper-3.x.x. Jar (这里有多个版本) 为官方提供的 java API

- **Apache Curator (推荐使用)**

Apache Curator是 Apache ZooKeeper的Java客户端库。

Curator.项目的目标是简化ZooKeeper客户端的使用。

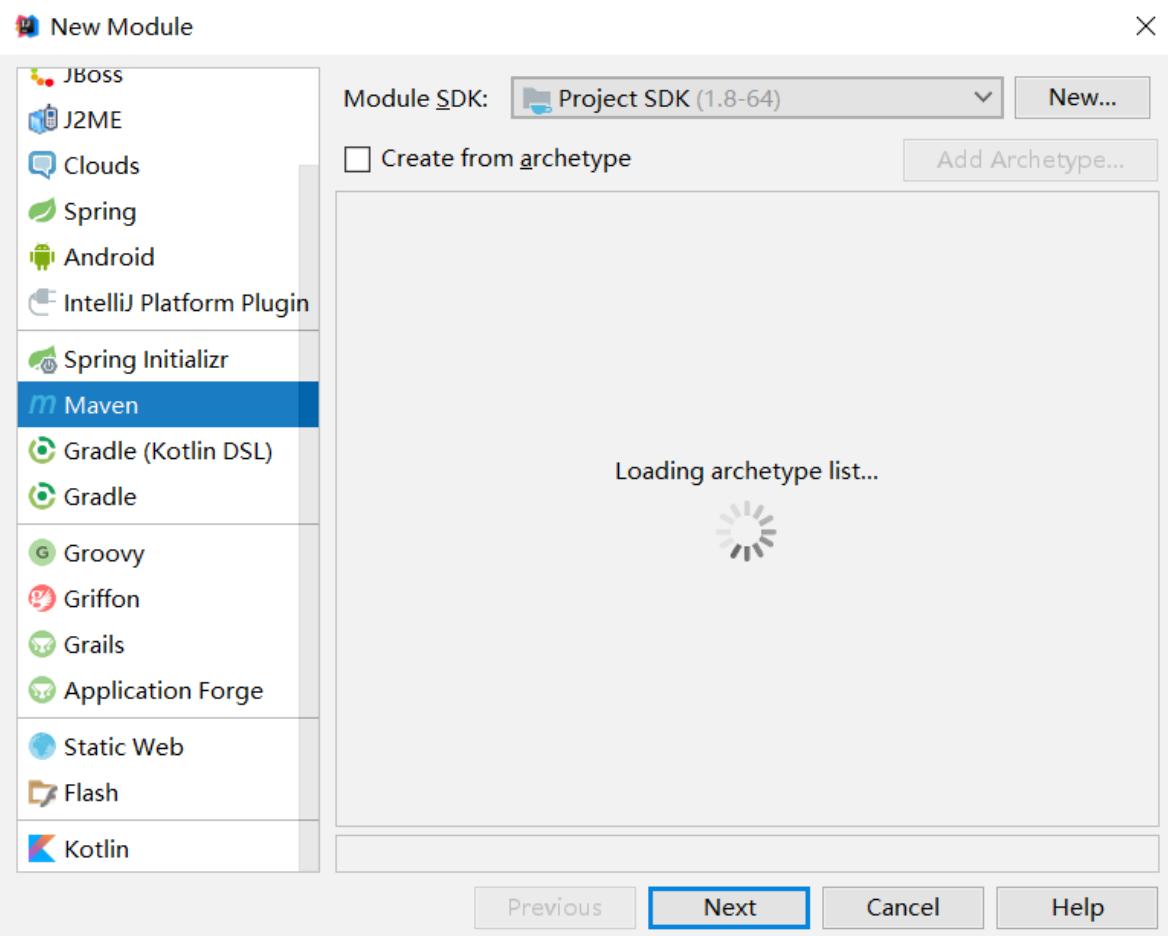
另外 Curator为常见的分布式协同服务提供了高质量的实现。

Apache Curator最初是Netflix研发的,后来捐献了 Apache基金会,目前是 Apache的顶级项目

- **ZkClient (不推荐使用)**

Github上一个开源的ZooKeeper客户端, 由datameer的工程师Stefan Groschupf和Peter Voss一起开发。 zkclient-x.x.Jar也是在源生 api 基础之上进行扩展的开源 JAVA 客户端。

3.3.2、创建java 工程，导入依赖

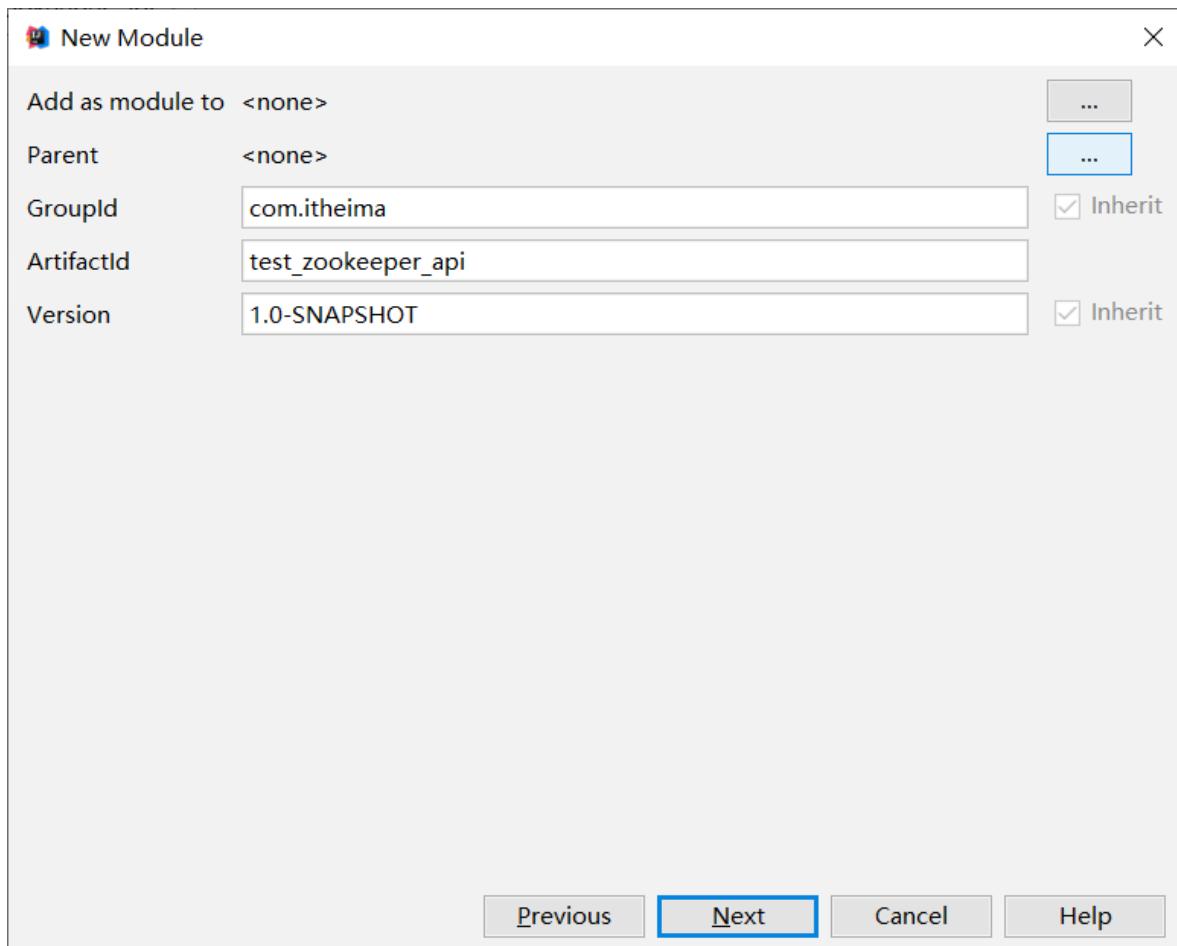


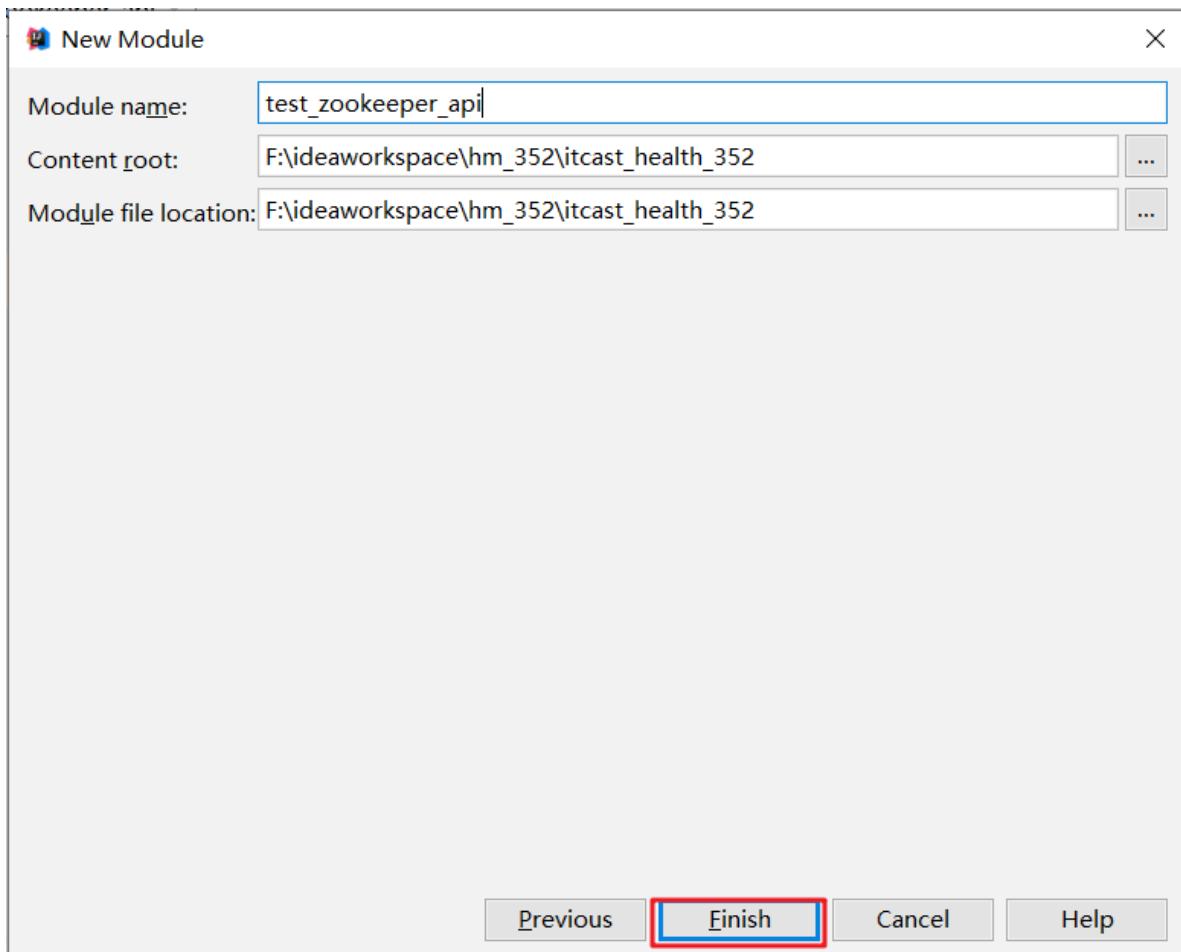
New Module

Add as module to <none>

Parent	<none>	...
GroupId	com.itheima	<input checked="" type="checkbox"/> Inherit
ArtifactId	test_zookeeper_api	...
Version	1.0-SNAPSHOT	<input checked="" type="checkbox"/> Inherit

Previous Next Cancel Help





导入依赖

```
<!--zookeeper的依赖-->
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.7</version>
</dependency>
<!--      zookeeper CuratorFramework 是Netflix公司开发一款连接zookeeper服务的框架，通过
封装的一套高级API 简化了ZooKeeper的操作，提供了比较全面的功能，除了基础的节点的操作，节点的监
听，还有集群的连接以及重试。-->
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-framework</artifactId>
    <version>4.0.1</version>
</dependency>
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>4.0.1</version>
</dependency>
<!--测试-->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

3.3.3、创建节点

```
//1. 创建一个空节点(a) (只能创建一层节点)
//2. 创建一个有内容的b节点 (只能创建一层节点)
//3. 创建持久节点, 同时创建多层节点
//4. 创建带有的序号的持久节点
//5. 创建临时节点 (客户端关闭, 节点消失), 设置延时5秒关闭 (Thread.sleep(5000))
//6. 创建临时带序号节点 (客户端关闭, 节点消失), 设置延时5秒关闭 (Thread.sleep(5000))
```

```
/**
 * RetryPolicy: 失败的重试策略的公共接口
 * ExponentialBackoffRetry是 公共接口的其中一个实现类
 * 参数1: 初始化sleep的时间, 用于计算之后的每次重试的sleep时间
 * 参数2: 最大重试次数
 * 参数3 (可以省略): 最大sleep时间, 如果上述的当前sleep计算出来比这个大, 那么sleep用这个时间
 */
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,3,10);
//创建客户端
/**
 * 参数1: 连接的ip地址和端口号
 * 参数2: 会话超时时间, 单位毫秒
 * 参数3: 连接超时时间, 单位毫秒
 * 参数4: 失败重试策略
 */
CuratorFramework client =
    CuratorFrameworkFactory.newClient("127.0.0.1:2181",3000,1000,retryPolicy);
//开启客户端(会阻塞到会话连接成功为止)
client.start();
/**
 * 创建节点
 */
//1. 创建一个空节点(a) (只能创建一层节点)
// client.create().forPath("/a");
//2. 创建一个有内容的b节点 (只能创建一层节点)
// client.create().forPath("/b", "这是b节点的内容".getBytes());
//3. 创建多层节点
// (creatingParentsIfNeeded) 是否需要递归创建节点
// withMode(CreateMode.PERSISTENT) 创建持久性 b节点
//
client.create().creatingParentsIfNeeded().withMode(CreateMode.PERSISTENT).forPath("/g");
//4. 创建带有的序号的节点
//
client.create().creatingParentsIfNeeded().withMode(CreateMode.PERSISTENT_SEQUENTIAL).forPath("/e");
//5. 创建临时节点 (客户端关闭, 节点消失), 设置延时5秒关闭
//
client.create().creatingParentsIfNeeded().withMode(CreateMode.EPHEMERAL).forPath("/f");
//6. 创建临时带序号节点 (客户端关闭, 节点消失), 设置延时5秒关闭
client.create().creatingParentsIfNeeded().withMode(CreateMode.EPHEMERAL_SEQUENTIAL).forPath("/f");
Thread.sleep(5000);
//关闭客户端
client.close();
```

3.3.4、修改节点数据

```
//创建失败策略对象
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,1);
//创建客户端
CuratorFramework client =
CuratorFrameworkFactory.newClient("127.0.0.1:2181",1000,1000,retryPolicy);
client.start();
//修改节点
client.setData().forPath("/a/b", "abc".getBytes());
client.close();
```

3.3.5、节点数据查询

```
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 1);
CuratorFramework client =
    CuratorFrameworkFactory.newClient("127.0.0.1",1000,1000, retryPolicy);
client.start();
// 查询节点数据
byte[] bytes = client.getData().forPath("/a/b");
System.out.println(new String(bytes));
```

3.3.6、删除节点

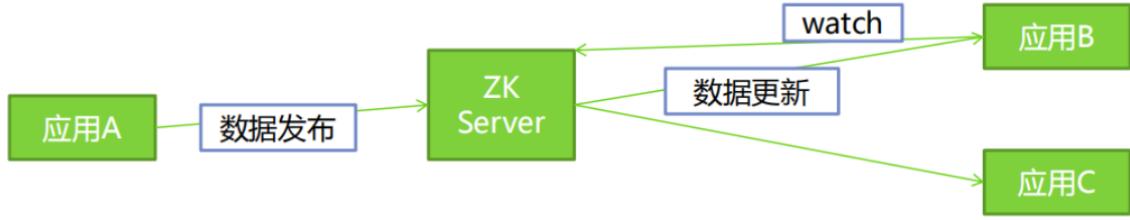
```
//重试策略
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,1);
//创建客户端
CuratorFramework client =
    CuratorFrameworkFactory.newClient("127.0.0.1",1000,1000, retryPolicy);
//启动客户端
client.start();
//删除一个子节点
client.delete().forPath("/a");
// 删除节点并递归删除其子节点
client.delete().deletingChildrenIfNeeded().forPath("/a");
//强制保证删除一个节点
//只要客户端会话有效，那么Curator会在后台持续进行删除操作，直到节点删除成功。
// 比如遇到一些网络异常的情况，此guaranteed的强制删除就会很有效果。
client.delete().guaranteed().deletingChildrenIfNeeded().forPath("/a");
//关闭客户端
client.close();
```

【小结】

```
//重试策略
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,1);
//创建客户端
CuratorFramework client =
    CuratorFrameworkFactory.newClient("127.0.0.1",1000,1000, retryPolicy);
// 节点创建、删除...
// 节点数据的修改、查询...
```

3.4、watch机制

回顾：Zookeeper的应用场景中配置中心，其中看到watch机制



3.4.1、什么是watch机制

zookeeper作为一款成熟的分布式协调框架，订阅-发布功能是很重要的一个。所谓订阅发布功能，其实说白了就是观察者模式。观察者会订阅一些感兴趣的主题，然后这些主题一旦变化了，就会自动通知到这些观察者。

zookeeper的订阅发布也就是watch机制，是一个轻量级的设计。因为它采用了一种推拉结合的模式。一旦服务端感知主题变了，那么只会发送一个事件类型和节点信息给关注的客户端，而不会包括具体的变更内容，所以事件本身是轻量级的，这就是所谓的“推”部分。然后，收到变更通知的客户端需要自己去拉变更的数据，这就是“拉”部分。watch机制分为添加数据和监听节点。

Curator在这方面做了优化，Curator引入了Cache的概念来实现对ZooKeeper服务器端进行事件监听。Cache是Curator对事件监听的包装，其对事件的监听可以近似看做是一个本地缓存视图和远程ZooKeeper视图的对比过程。而且Curator会自动的再次监听，我们就不需要自己手动的重复监听了。

Curator中的cache共有三种

- NodeCache (监听和缓存根节点变化)
- PathChildrenCache (监听和缓存子节点变化)
- TreeCache (监听和缓存根节点变化和子节点变化)

下面我们分别对三种cache详解

3.4.2、NodeCache

• 介绍

NodeCache是用来监听节点的数据变化的，当监听的节点的数据发生变化的时候就会回调对应的函数。

• 增加监听

```

//创建重试策略
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 1);
//创建客户端
CuratorFramework client = CuratorFrameworkFactory.newClient("127.0.0.1:2181",
1000, 1000, retryPolicy);
//开启客户端
client.start();
System.out.println("连接成功");
//创建节点数据监听对象
final NodeCache nodeCache = new NodeCache(client, "/hello");
//开始缓存
/**
 * 参数为true: 可以直接获取监听的节点,
System.out.println(nodeCache.getCurrentData()); 为ChildData{path='/aa',
stat=607,765,1580205779732,1580973376268,2,1,0,0,5,1,608
, data=[97, 98, 99, 100, 101]}
 * 参数为false: 不可以获取监听的节点, System.out.println(nodeCache.getCurrentData());
为null

```

```

*/
nodeCache.start(true);
System.out.println(nodeCache.getCurrentData());
//添加监听对象
nodeCache.getListenable().addListener(new NodeCacheListener() {
    //如果节点数据有变化，会回调该方法
    public void nodeChanged() throws Exception {
        String data = new String(nodeCache.getCurrentData().getData());
        System.out.println("数据watcher: 路径=" +
nodeCache.getCurrentData().getPath()
                + ":data=" + data);
    }
});
System.in.read();

```

- 测试

修改节点的数据

```
[zk: localhost:2181(CONNECTED) 64] set /hello/a 'bcd'
cZxid = 0x532
ctime = Sat Nov 09 10:14:45 CST 2019
mZxid = 0x542
mtime = Sat Nov 09 10:24:30 CST 2019
pZxid = 0x532
cversion = 0
dataVersion = 6
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 3
numChildren = 0
```

控制台显示

数据Watcher: 路径=/hello/a:data=bcd

3.4.3、PathChildrenCache

- 介绍

PathChildrenCache是用来监听指定节点的子节点变化情况

- 增加监听

```

RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,1);
CuratorFramework client = CuratorFrameworkFactory.newClient("127.0.0.1:2181",
1000, 1000, retryPolicy);
client.start();
//监听指定节点的子节点变化情况包括新增子节点 子节点数据变更 和子节点删除
//true表示用于配置是否把节点内容缓存起来，如果配置为true，客户端在接收到节点列表变更的同时，也能够获取到节点的数据内容（即：event.getData().getData()），如果为false 则无法取到数据内容（即：event.getData().getData()）
PathChildrenCache childrenCache = new PathChildrenCache(client,"/hello",true);
/**
 * NORMAL: 普通启动方式，在启动时缓存子节点数据
 * POST_INITIALIZED_EVENT: 在启动时缓存子节点数据，提示初始化
 * BUILD_INITIAL_CACHE: 在启动时什么都不会输出

```

```

* 在官方解释中说是因为这种模式会在start执行执行之前先执行rebuild的方法，而rebuild的方法不
会发出任何事件通知。
*/
childrenCache.start(PathChildrenCache.StartMode.POST_INITIALIZED_EVENT);
System.out.println(childrenCache.getCurrentData());
//添加监听
childrenCache.getListenable().addListener(new PathChildrenCacheListener() {
    @Override
    public void childEvent(CuratorFramework client, PathChildrenCacheEvent
event) throws Exception {
        if(event.getType() == PathChildrenCacheEvent.Type.CHILD_UPDATED){
            System.out.println("子节点更新");
            System.out.println("节点:"+event.getData().getPath());
            System.out.println("数据" + new String(event.getData().getData()));
        }else if(event.getType() == PathChildrenCacheEvent.Type.INITIALIZED ){
            System.out.println("初始化操作");
        }else if(event.getType() == PathChildrenCacheEvent.Type.CHILD_REMOVED ){
            System.out.println("删除子节点");
            System.out.println("节点:"+event.getData().getPath());
            System.out.println("数据" + new String(event.getData().getData()));
        }else if(event.getType() == PathChildrenCacheEvent.Type.CHILD_ADDED ){
            System.out.println("添加子节点");
            System.out.println("节点:"+event.getData().getPath());
            System.out.println("数据" + new String(event.getData().getData()));
        }else if(event.getType() ==
PathChildrenCacheEvent.Type.CONNECTION_SUSPENDED ){
            System.out.println("连接失效");
        }else if(event.getType() ==
PathChildrenCacheEvent.Type.CONNECTION_RECONNECTED ){
            System.out.println("重新连接");
        }else if(event.getType() == PathChildrenCacheEvent.Type.CONNECTION_LOST
){
            System.out.println("连接失效后稍等一会儿执行");
        }
    }
});
System.in.read(); // 使线程阻塞

```

3.4.4、TreeCache

- 介绍

TreeCache有点像上面两种Cache的结合体，NodeCache能够监听自身节点的数据变化（或者是创建该节点），PathChildrenCache能够监听自身节点下的子节点的变化，而TreeCache既能够监听自身节点的变化、也能够监听子节点的变化。

- 添加监听

```

RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,1);
CuratorFramework client = CuratorFrameworkFactory.newClient("127.0.0.1:2181",
1000, 1000, retryPolicy);
client.start();
TreeCache treeCache = new TreeCache(client, "/hello");
treeCache.start();
System.out.println(treeCache.getCurrentData("/hello"));
treeCache.getListenable().addListener(new TreeCacheListener() {
    @Override

```

```

public void childEvent(CuratorFramework client, TreeCacheEvent event) throws
Exception {
    if(event.getType() == TreeCacheEvent.Type.NODE_ADDED){
        System.out.println(event.getData().getPath() + "节点添加");
    }else if (event.getType() == TreeCacheEvent.Type.NODE_REMOVED){
        System.out.println(event.getData().getPath() + "节点移除");
    }else if(event.getType() == TreeCacheEvent.Type.NODE_UPDATED){
        System.out.println(event.getData().getPath() + "节点修改");
    }else if(event.getType() == TreeCacheEvent.Type.INITIALIZED){
        System.out.println("初始化完成");
    }else if(event.getType() ==TreeCacheEvent.Type.CONNECTION_SUSPENDED)
{
    System.out.println("连接过时");
} else if(event.getType()
==TreeCacheEvent.Type.CONNECTION_RECONNECTED){
    System.out.println("重新连接");
} else if(event.getType() ==TreeCacheEvent.Type.CONNECTION_LOST){
    System.out.println("连接过时一段时间");
}
}
);
System.in.read();

```

【小结】

- 1: Zookeeper的数据结构 (树型结构)
- 2: 节点的分类 (4个)
 - 持久性 (带序号、不带序号)
 - 临时性 (带序号、不带序号)
- 3: 客户端命令 (创建、查询、修改、删除)
- 4: Zookeeper的java的api介绍 (创建、查询、修改、删除)
 - Curator的客户端

```

RetryPolicy retryPolicy = new ExponentialBackoffRetry(3000,3);
CuratorFramework client =
CuratorFrameworkFactory.newClient("127.0.0.1:2181", 3000, 3000,
retryPolicy);

```

- 5: Zookeeper的watch机制
 - NodeCache
 - PathChildrenCache
 - TreeCache (监听和缓存根几点变化和子节点变化) (重点)

4、zookeeper集群搭建

【目标】

- 1: 搭建Zookeeper集群

【路径】

- 1: 了解集群

- 集群介绍
- 集群模式

2: 集群原理及架构图

3: 搭建集群 (使用linux)

4: 集群中Leader选举机制

5: 测试集群

【讲解】

4.1、了解集群

4.1.1、集群介绍

Zookeeper 集群搭建指的是 ZooKeeper 分布式模式安装。通常由 $2n+1$ 台 servers 组成。这是因为为了保证 Leader 选举（基于 Paxos 算法的实现）能或得到多数的支持，所以 ZooKeeper 集群的数量一般为奇数。

4.1.2、集群模式

ZooKeeper集群搭建有两种方式：

- **伪分布式集群搭建：**

所谓伪分布式集群搭建ZooKeeper集群其实就是在一台机器上启动多个ZooKeeper，在启动每个ZooKeeper时分别使用不同的配置文件zoo.cfg来启动,每个配置文件使用不同的配置参数(clientPort端口号、dataDir数据目录).

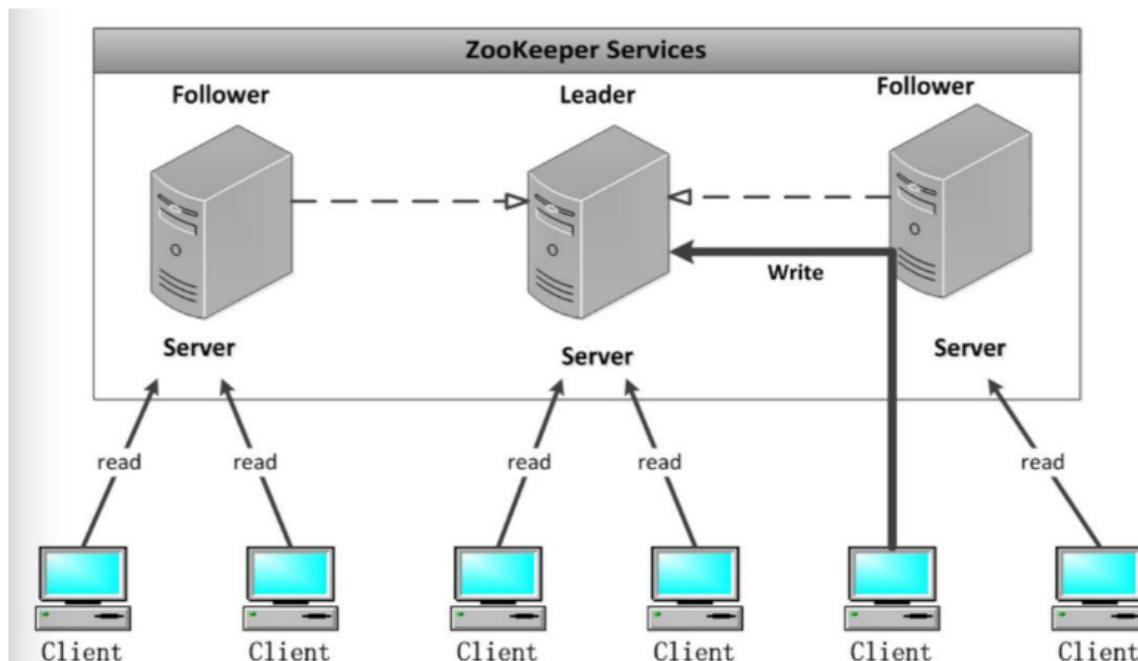
在zoo.cfg中配置多个server.id，其中ip都是当前机器，而端口各不相同，启动时就是伪集群模式了。

这种模式和单机模式产生的问题是一样的。也是用在测试环境中使用。

- **完全分布式集群搭建：**

多台机器各自配置zoo.cfg文件，将各自互相加入服务器列表，每个节点占用一台机器

4.2、架构图



Leader:Zookeeper(领导者):集群工作的核心

事务请求（写操作）的唯一调度和处理者，保证集群事务处理的顺序性；

集群内部各个服务器的调度者。

对于 create, setData, delete 等有写操作的请求，则需要统一转发给 leader 处理，leader 需要决定编号、执行操作，这个过程称为一个事务。

Follower (跟随者)

处理客户端非事务（读操作）请求，转发事务请求给 Leader；参与集群 Leader 选举投票 $2n-1$ 台可以做集群投票。

Observer:观察者角色

针对访问量比较大的 zookeeper 集群，还可新增观察者角色。观察 Zookeeper 集群的最新状态变化并将这些状态同步过来，其对于非事务请求可以进行独立处理，对于事务请求，则会转发给 Leader 服务器进行处理。

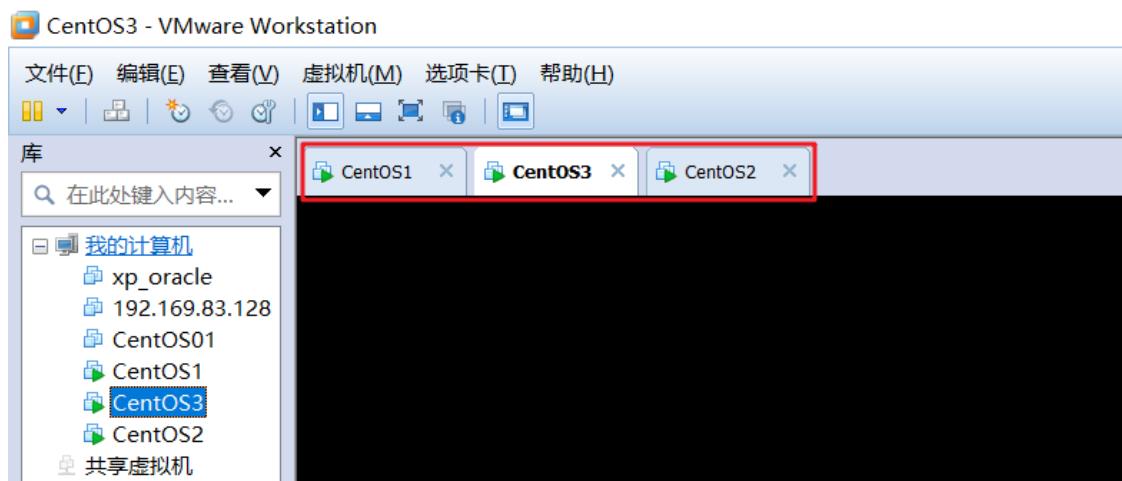
不会参与任何形式的投票只提供非事务服务，通常用于在不影响集群事务处理能力的前提下提升集群的非事务处理能力。

4.3、搭建过程

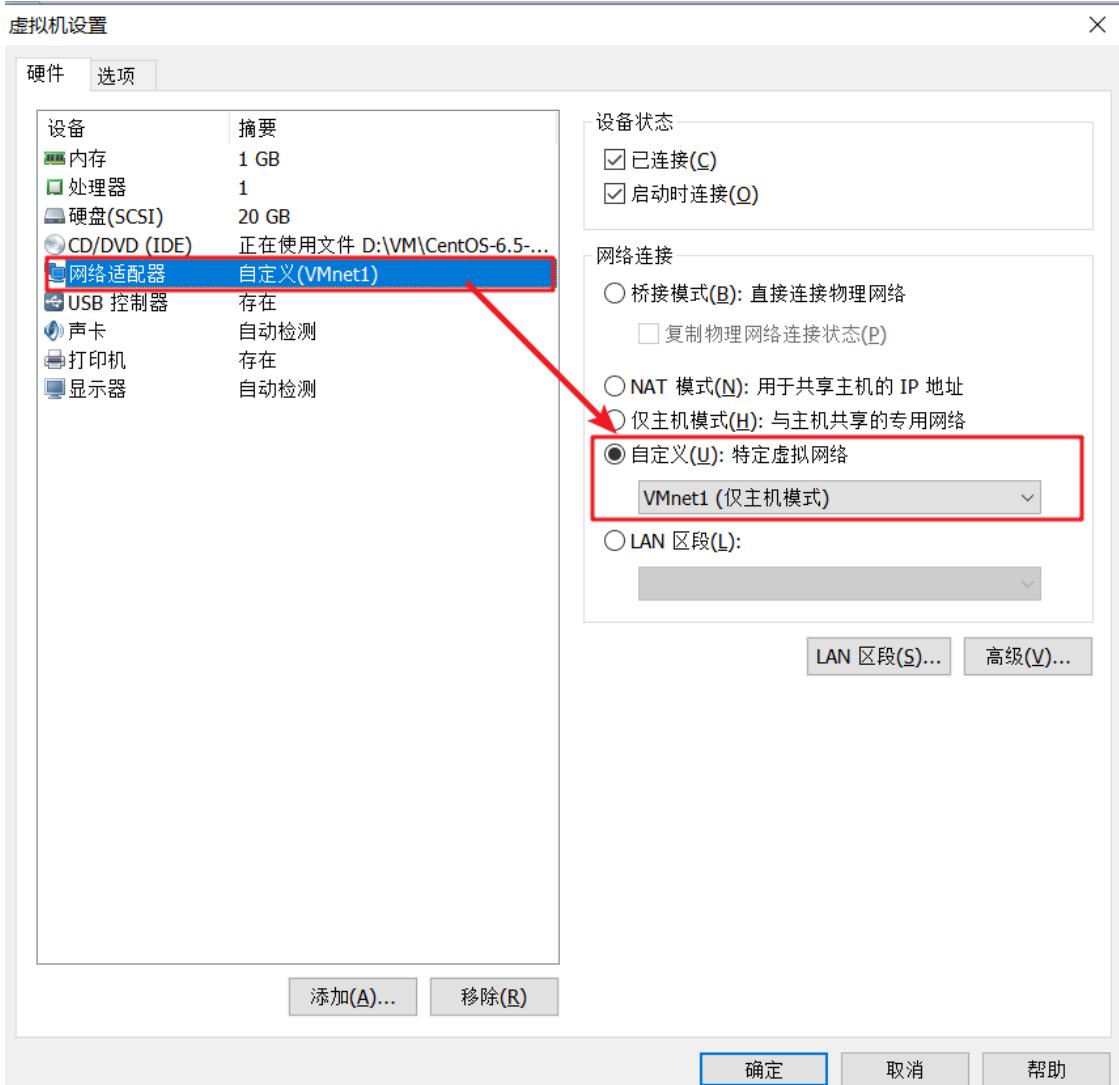
4.3.1、准备三台电脑（虚拟机）

账号：root 密码：root

- 安装三个linux虚拟机



- 配置虚拟网卡（三台电脑配置一个虚拟网卡）（虚拟网卡ip为：192.168.174.0）



- 为每台虚拟机配置静态ip地址

第一步 `zookeeper1` **第二步** `vm-zookeeper2`

```
[root@tensquare ~]# vi /etc/sysconfig/network-scripts/ifcfg-ens33
```

第三步 `zookeeper1` **第二步** `vm-zookeeper2`

```
TYPE="Ethernet"
PROXY_METHOD="none"
BROWSER_ONLY="no"
BOOTPROTO="static" 静态ip
DEFROUTE="yes"
IPV4_FAILURE_FATAL="no"
IPV6INIT="yes"
IPV6_AUTOCONF="yes"
IPV6_DEFROUTE="yes"
IPV6_FAILURE_FATAL="no"
IPV6_ADDR_GEN_MODE="stable-privacy"
NAME="ens33"
IPADDR="192.168.174.128" ip地址
UUID="06405b69-11bb-409e-b8d4-af1822e5d27"
DEVICE="ens33"
ONBOOT="yes"
```

三台虚拟机分别是：192.168.174.128、192.168.174.129、192.168.174.130

关闭防火墙: `service iptables stop`

4.3.2、安装jdk（略）

- 查看是否已经安装jdk

命令: rpm -qa|grep java

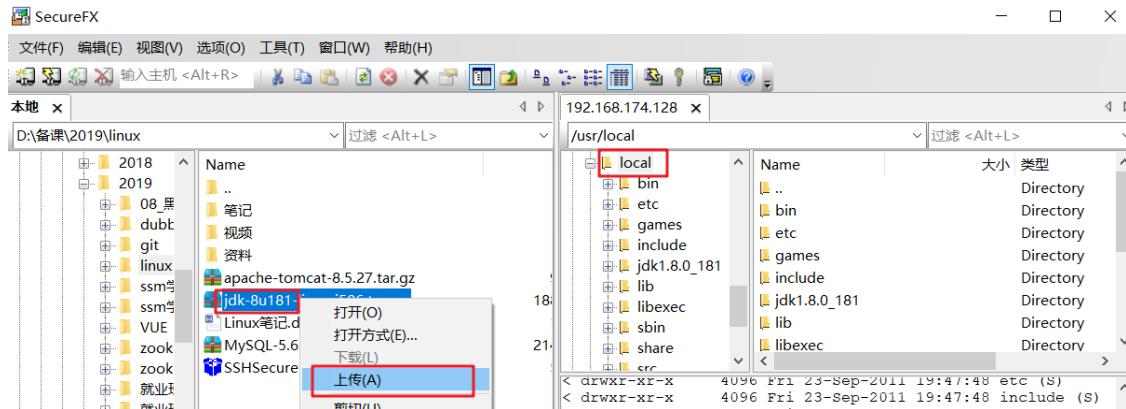
```
java-1.6.0-openjdk-1.6.0.35-1.13.7.1.el6_6.i686
tzdata-java-2015e-1.el6.noarch
java-1.7.0-openjdk-1.7.0.79-2.5.5.4.el6.i686
```

- 卸载已安装的jdk

命令: rpm -e --nodeps

```
[root@itheima23 ~]# rpm -e --nodeps java-1.7.0-openjdk-1.7.0.79-2.5.5.4.el6.i686
[root@itheima23 ~]# rpm -e --nodeps java-1.6.0-openjdk-1.6.0.35-1.13.7.1.el6_6.i686
```

- 上传jdk到linux



- 解压jdk

命令: tar -vxf jdk-8u181-linux-i586.tar.gz

- 配置环境变量

命令: cd / 退回根目录

命令: cd etc

命令: vim profile (编辑etc/profile文件,将如下配置粘贴到文件中)

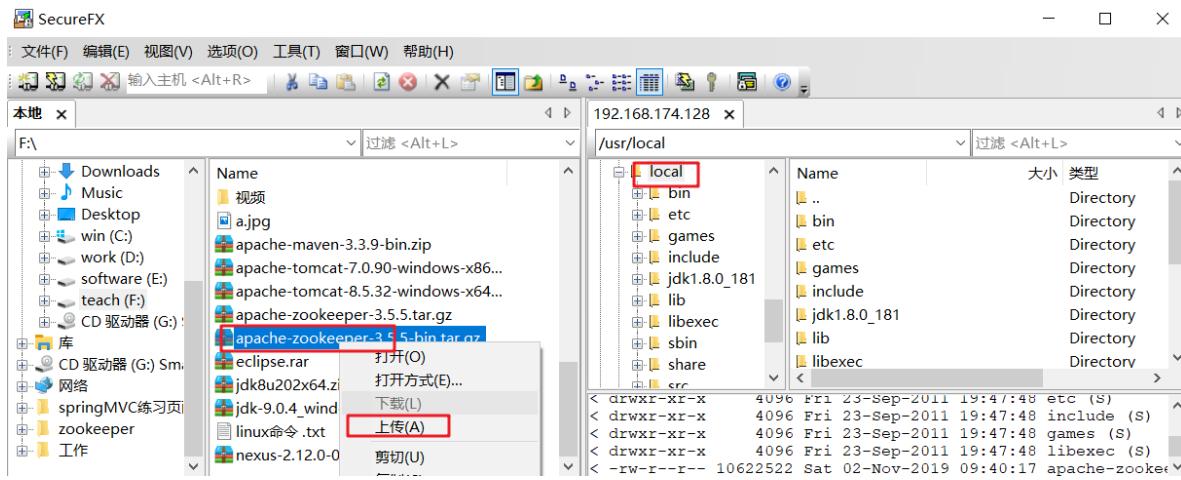
```
#set java environment
JAVA_HOME=/usr/local/jdk1.8.0_181
CLASSPATH=.:$JAVA_HOME/lib/tools.jar
PATH=$JAVA_HOME/bin:$PATH
export JAVA_HOME CLASSPATH PATH
```

- 重新加载profile文件, 即激活profile文件

命令: source profile

4.3.3、上传zookeeper到虚拟机并解压

- 上传到虚拟机



- 解压

命令: tar -zvxf apache-zookeeper-3.5.5-bin.tar.gz (解压)

命令: mv apache-zookeeper-3.5.5-bin zookeeper (重命名文件夹)

命令: mv zookeeper /usr/local/ (将zookeeper移动到/usr/local目录中)

命令: cd /usr/local (进入local路径)

4.3.4、配置zookeeper

- 创建data目录

命令: cd zookeeper (进入zookeeper目录)

命令: mkdir data

- 修改conf/zoo.cfg

命令: cd conf (进入conf目录)

命令: cp zoo_sample.cfg zoo.cfg (复制zoo_sample.cfg, 文件名为zoo.cfg)

```
[root@localhost conf]# ll    查看目录
总用量 20
-rw-r--r--. 1 2002 2002 535 2月 15 2019 configuration.xls
-rw-r--r--. 1 2002 2002 2712 4月 2 2019 log4j.properties
-rw-r--r--. 1 root root 1038 11月 12 03:04 zoo.cfg
-rw-r--r--. 1 root root 158 11月 12 03:04 zoo.cfg.dynamic.next
-rw-r--r--. 1 2002 2002 922 2月 15 2019 zoo_sample.cfg
```

命令: vim zoo.cfg (修改zoo.cfg)

内容修改:

dataDir = /usr/local/zookeeper/data

添加:

server.1=192.168.174.128:2182:3182

server.2=192.168.174.129:2182:3182

server.3=192.168.174.130:2182:3182

```

# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sake
dataDir=/usr/local/zookeeper/data
# the port at which the clients will connect
clientPort=2181
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
#
# Be sure to read the maintenance section of the
# administrator guide before turning on autopurge.
#
# http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc_maintenance
#
# The number of snapshots to retain in datadir
#autpurge.snapRetainCount=3
# Purge task interval in hours
# Set to "0" to disable auto purge feature
#autpurge.purgeInterval=1
server.1=192.168.174.128:2182:3182
server.2=192.168.174.129:2182:3182
server.3=192.168.174.130:2182:3182

```

zookeeper数据保存目录

zookeeper默认端口号

集群机器配置

参数详解(了解)

1) **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。

它用于心跳机制, 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 $2 * \text{tickTime}$)

2) **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), 用它来限定集群中的Zookeeper服务器连接到Leader的时限。

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F在**initLimit**时间内完成这个工作。

3) **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**Follower**死掉, 从服务器列表中删除**Follower**。

在运行过程中, **Leader**负责与ZK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。

4) **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。

5) **clientPort:** 客户端连接端口

监听客户端连接的端口

6) 集群中服务的列表

server.1=192.168.174.128:2182:3182

server.2=192.168.174.129:2182:3182

server.3=192.168.174.130:2182:3182

server.* 后面的数据对应myid中的数字

ip地址对应每台虚拟机的ip地址

2182: 表示的是这个服务器与集群中的 **Leader** 服务器交换信息的端口

3182: 表示的是万一集群中的 **Leader** 服务器挂了, 需要一个端口来重新进行选举, 选出一个新的 **Leader**, 而这个端口就是用来执行选举时服务器相互通信的端口

- 在data目录创建创建myid

命令: cd .. (退出conf目录)

命令：cd data (进入data目录)

命令：vim myid (修改myid文件,分别设置为1,2,3)

注意：不要着急启动，关闭虚拟机。复制出两个虚拟机分别修改myid文件为2,3

4.3.5、启动zookeeper

命令：cd .. (退出data目录)

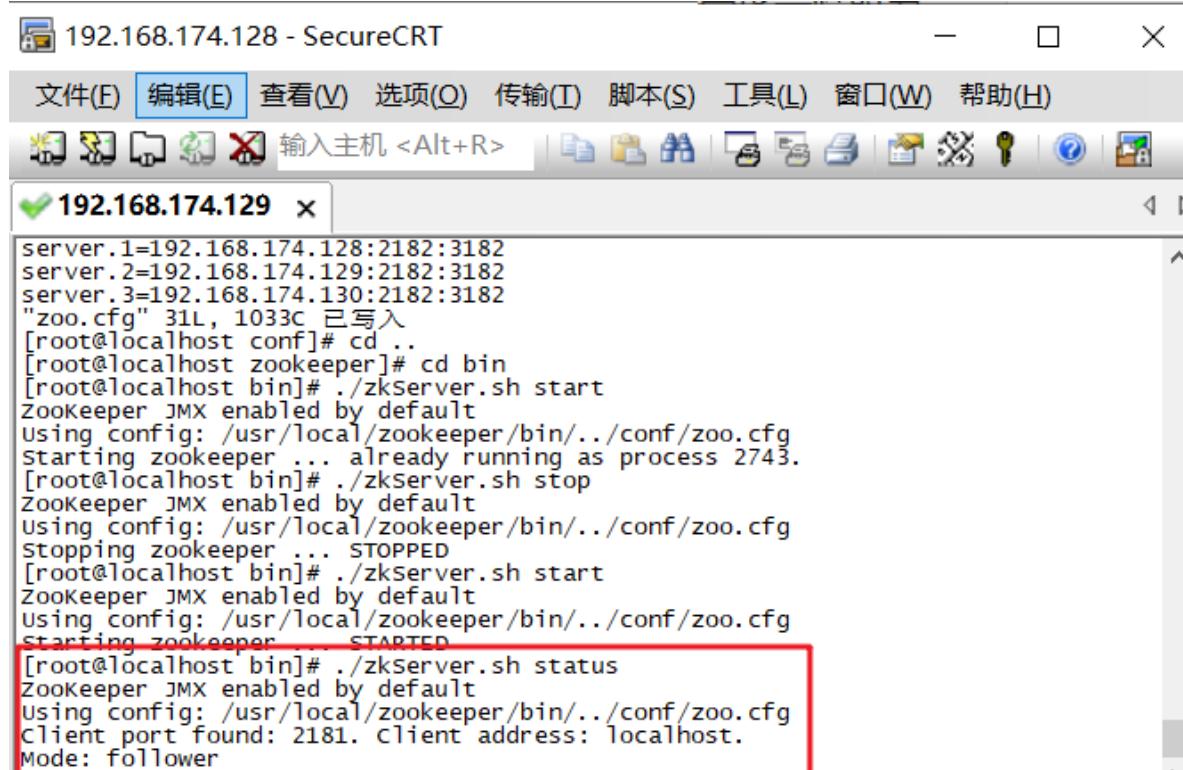
命令：cd bin(进入bin路径)

命令：./zkServer.sh start

```
[root@localhost bin]# ./zkServer.sh start
ZooKeeper JMX enabled by default
using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
```

4.3.6、查看zookeeper状态

命令：./zkServer.sh status



The screenshot shows a SecureCRT window titled '192.168.174.129'. The terminal session displays the following command-line interaction:

```
server.1=192.168.174.128:2182:3182
server.2=192.168.174.129:2182:3182
server.3=192.168.174.130:2182:3182
"zoo.cfg" 31L, 1033c 已写入
[root@localhost conf]# cd ..
[root@localhost zookeeper]# cd bin
[root@localhost bin]# ./zkServer.sh start
ZooKeeper JMX enabled by default
using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... already running as process 2743.
[root@localhost bin]# ./zkServer.sh stop
ZooKeeper JMX enabled by default
using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
[root@localhost bin]# ./zkServer.sh start
ZooKeeper JMX enabled by default
using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@localhost bin]# ./zkServer.sh status
ZooKeeper JMX enabled by default
using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Client port found: 2181. Client address: localhost.
Mode: follower
```

The screenshot shows two terminal windows from the SecureCRT application. Both windows have the title bar "192.168.174.129 - SecureCRT" and "192.168.174.130 - SecureCRT". The menu bars include "文件(F)" (File), "编辑(E)" (Edit), "查看(V)" (View), "选项(O)" (Options), "传输(I)" (Transfer), "脚本(S)" (Script), "工具(L)" (Tools), "窗口(W)" (Windows), and "帮助(H)" (Help). The toolbars contain various icons for file operations like Open, Save, Print, and Help.

The terminal session for server 192.168.174.129 shows the following command history:

```

server.1=192.168.174.128:2182:3182
server.2=192.168.174.129:2182:3182
server.3=192.168.174.130:2182:3182
"zoo.cfg" 31L, 1033C 已写入
[root@localhost conf]# cd ..
[root@localhost zookeeper]# cd bin
[root@localhost bin]# ./zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... already running as process 2743.
[root@localhost bin]# ./zkServer.sh stop
ZooKeeper JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
[root@localhost bin]# ./zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@localhost bin]# ./zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
client port found: 2181. client address: localhost.
Mode: follower
[root@localhost bin]#

```

The status output for server 192.168.174.129 shows it is in "follower" mode. A red box highlights the "STARTED" status message and the "client address: localhost." line.

The terminal session for server 192.168.174.130 shows the following command history:

```

server.1=192.168.174.128:2182:3182
server.2=192.168.174.129:2182:3182
server.3=192.168.174.130:2182:3182
"zoo.cfg" 31L, 1038C 已写入
[root@localhost conf]# cd ..
[root@localhost zookeeper]# cd bin
[root@localhost bin]# ./zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... already running as process 2558.
[root@localhost bin]# ./zkServer.sh stop
ZooKeeper JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
[root@localhost bin]# ./zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@localhost bin]# ./zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
client port found: 2181. client address: localhost.
Mode: leader
[root@localhost bin]#

```

The status output for server 192.168.174.130 shows it is in "leader" mode. A red box highlights the "STARTED" status message and the "client address: localhost." line.

4.4、leader选举

在领导者选举的过程中，如果某台ZooKeeper获得了超过半数的选票，则此ZooKeeper就可以成为Leader了。

- 服务器1启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，服务器1的状态一直属于Looking(选举状态)。
- 服务器2启动，给自己投票，同时与之前启动的服务器1交换结果，

每个Server发出一个投票由于是初始情况，1和2都会将自己作为Leader服务器来进行投票，每次投票会包含所推举的服务器的myid和ZXID，使用(myid, ZXID)来表示，此时1的投票为(1, 0)，2的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。

接受来自各个服务器的投票集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票、是否来自LOOKING状态的服务器

处理投票。针对每一个投票，服务器都需要将别人的投票和自己的投票进行PK，PK规则如下

- 优先检查ZXID。ZXID比较大的服务器优先作为Leader。
- 如果ZXID相同，那么就比较myid。myid较大的服务器作为Leader服务器

对于服务器1而言，由于服务器2的编号大，更新服务器1的投票为(2, 0)，然后重新投票，对于2而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可，此时集群节点状态为LOOKING。

统计投票。每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息

- 服务器3启动，进行统计后，判断是否已经有过半机器接受到相同的投票信息，对于1、2、3而言，已统计出集群中已经有3台机器接受了(3, 0)的投票信息，此时投票数正好大于半数，便认为已经选出了Leader，

改变服务器状态。一旦确定了Leader，每个服务器就会更新自己的状态，如果是Follower，那么就变更为FOLLOWING，如果是Leader，就变更为LEADING

ZXID：即zookeeper事务id号。ZooKeeper状态的每一次改变，都对应着一个递增的Transaction id，该id称为zxid

4.5、测试集群

第一步：3台集群，随便挑选一个启动，执行命令./zkCli.sh

```
[root@itheima bin]# ./zkCli.sh
Connecting to localhost:2181
2020-01-05 04:14:27,548 [myid:] - INFO  [main:Environment@100] - 
ent:zookeeper.version=3.4.13-2d71af4dbe22557fda74f9a9b4309b15a74
06/29/2018 04:05 GMT
2020-01-05 04:14:27,552 [myid:] - INFO  [main:Environment@100] - 
ent:host.name=<NA>
2020-01-05 04:14:27,552 [myid:] - INFO  [main:Environment@100] - 
ent:java.version=1.8.0_181
2020-01-05 04:14:27,558 [myid:] - INFO  [main:Environment@100] -
```

第二步：添加节点数据

```
[zk: localhost:2181(CONNECTED) 1] ls /
[zookeeper]
[zk: localhost:2181(CONNECTED) 2] create /hello 'aaa'
Created /hello
[zk: localhost:2181(CONNECTED) 3]
```

第三步：找另一台机器，测试查询，可以获取hello节点的数据

```
[zk: localhost:2181(CONNECTED) 0] ls /
[hello, zookeeper]
[zk: localhost:2181(CONNECTED) 1] get /hello
aaa
cZxid = 0x100000002
ctime = Sun Jan 05 04:17:36 CST 2020
mZxid = 0x100000002
mtime = Sun Jan 05 04:17:36 CST 2020
pZxid = 0x100000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 3
numChildren = 0
```

也可以使用代码测试

```
@Test
public void createNode() throws Exception {
    RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,3);
    CuratorFramework client =
CuratorFrameworkFactory.newClient("192.168.174.128:2181", 3000, 3000,
retryPolicy);
    client.start();

    client.create().creatingParentsIfNeeded().withMode(CreateMode.PERSISTENT).forPat
h("/bbb/b1", "haha".getBytes());
    client.close();
}

@Test
public void updateNode() throws Exception {
    //创建失败策略对象
    RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,1);
    CuratorFramework client =
CuratorFrameworkFactory.newClient("192.168.174.129:2181",1000,1000,retryPolicy);
    client.start();
    //修改节点
    client.setData().forPath("/bbb/b1", "fff".getBytes());
    client.close();
}

@Test
public void getData() throws Exception {
    RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 1);
    CuratorFramework client =
CuratorFrameworkFactory.newClient("192.168.174.130:2181",1000,1000,
retryPolicy);
    client.start();
    // 查询节点数据
    byte[] bytes = client.getData().forPath("/bbb/b1");
    System.out.println(new String(bytes));
    client.close();
}

@Test
public void deleteNode() throws Exception {
    RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 1);
    CuratorFramework client =
```

```
CuratorFrameworkFactory.newClient("192.168.174.130:2181", 1000, 1000,  
retryPolicy);  
client.start();  
// 递归删除节点  
client.delete().deletingChildrenIfNeeded().forPath("/bbb");  
client.close();  
}
```

测试使用任何一个IP都可以获取

测试如果有个机器宕机，（./zkServer.sh stop），会重新选取领导者。

【小结】

1：了解集群

- 采用多台虚拟机的方式搭建集群

2：集群架构图

3：搭建集群（使用linux）

4：集群中Leader选举机制（了解）

5：测试集群

Apache Dubbo

学习目标

- 了解应用架构演进过程
- 了解RPC远程调用方式
- 掌握Dubbo框架的架构
- 掌握Zookeeper注册中心的基本使用
- 掌握Dubbo生产者和消费者的开发
- 了解Dubbo的管理控制台的使用
- 了解Dubbo的相关配置
- 了解Dubbo的负载均衡
- 了解Dubbo的配置中心

1. 应用架构的演进过程

【目标】

了解软件架构的演进过程

【路径】

1：主流的互联网技术特点

2：架构演变的过程

(1) : 单体架构

(2) : 垂直架构

(3) : 分布式服务架构

(4) : SOA架构

(5) : 微服务架构

【讲解】

1.1. 主流的互联网技术特点

分布式、高并发、集群、负载均衡、高可用。

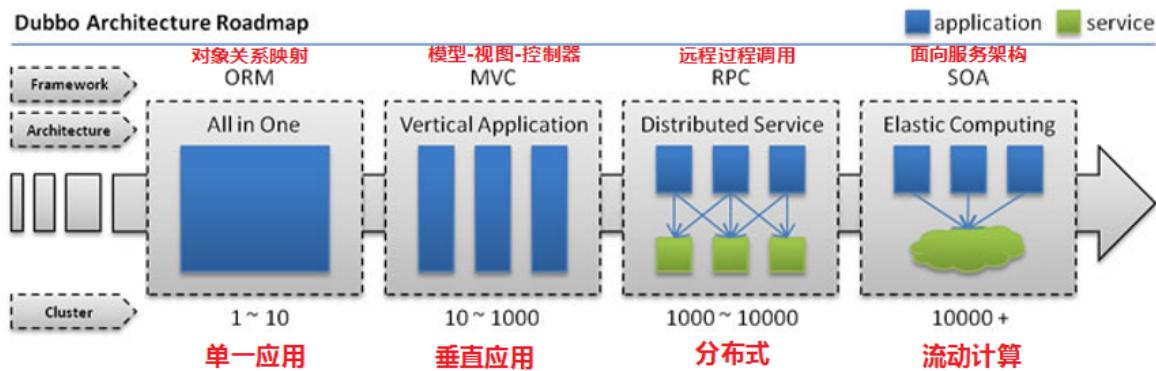
分布式：一件事情拆开来做。

集群：一件事情大家一起做。

负载均衡：将请求平均分配到不同的服务器中，达到均衡的目的。

高并发：同一时刻，处理同一件事情的处理能力（解决方案：分布式、集群、负载均衡）

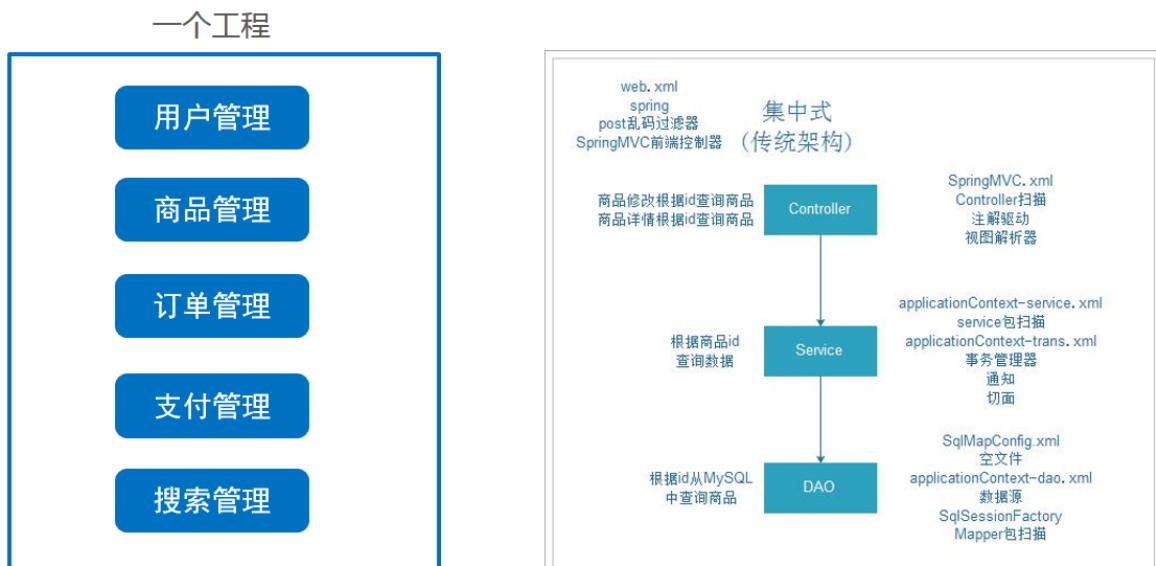
高可用：系统都是可用的。



1.2. 架构演变的过程

1.2.1. 单一应用架构 (all in one)

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架(ORM)是关键。



- 架构优点：

架构简单，前期开发成本低、开发周期短，适合小型项目（OA、CRM、ERP）。

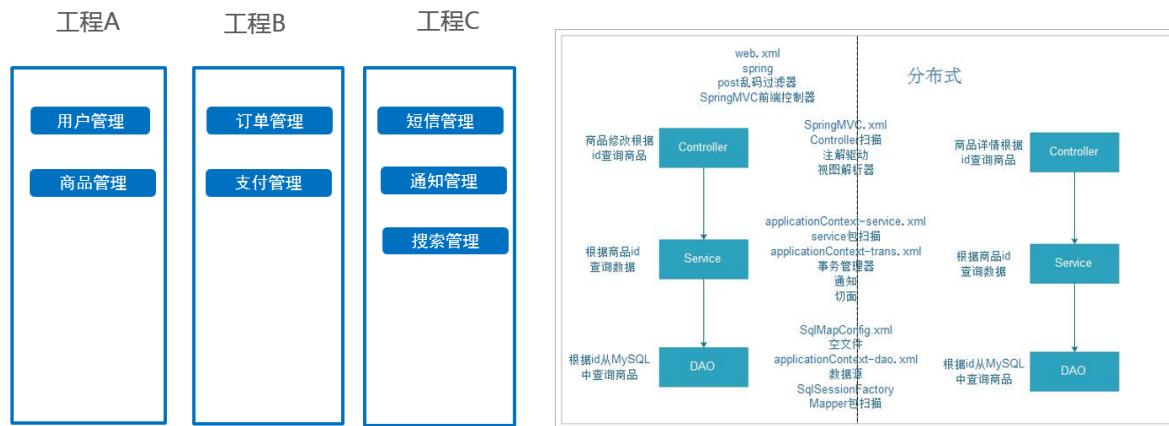
- 架构缺点：

全部功能集成在一个工程中

- (1) 业务代码耦合度高，不易维护。
- (2) 维护成本高，不易拓展
- (3) 并发量大，不易解决
- (4) 技术栈受限，只能使用一种语言开发。

1.2.2. 垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的Web框架(MVC)是关键。



- 架构优点：

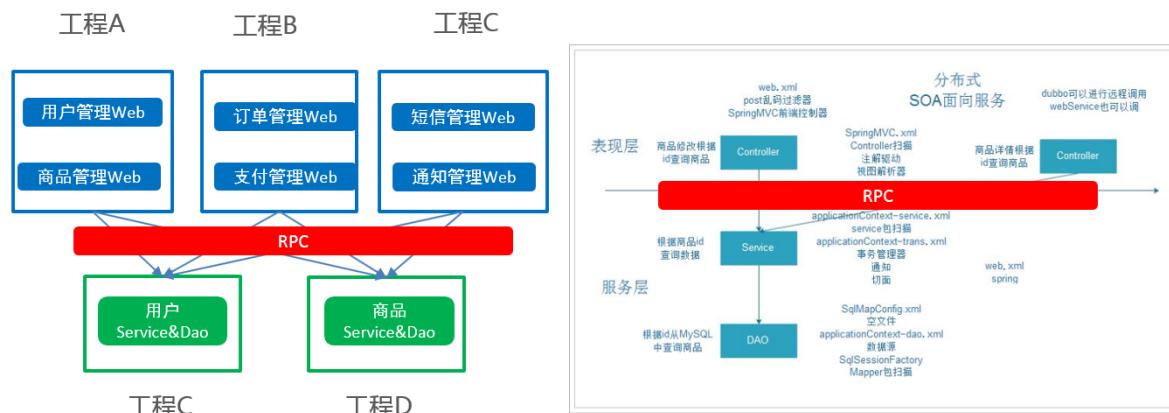
- (1) 业务代码相对解耦
- (2) 维护成本相对易于拓展（修改一个功能，可以直接修改一个项目，单独部署）
- (3) 并发量大相对易于解决（搭建集群）
- (4) 技术栈可扩展（不同的系统可以用不同的编程语言编写）。

- 架构缺点：

代码之间存在数据、方法的冗余

1.2.3. 分布式服务架构

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的分布式服务框架(RPC)是关键。



- 架构优点：

- (1) 业务代码完全解耦，并可实现通用

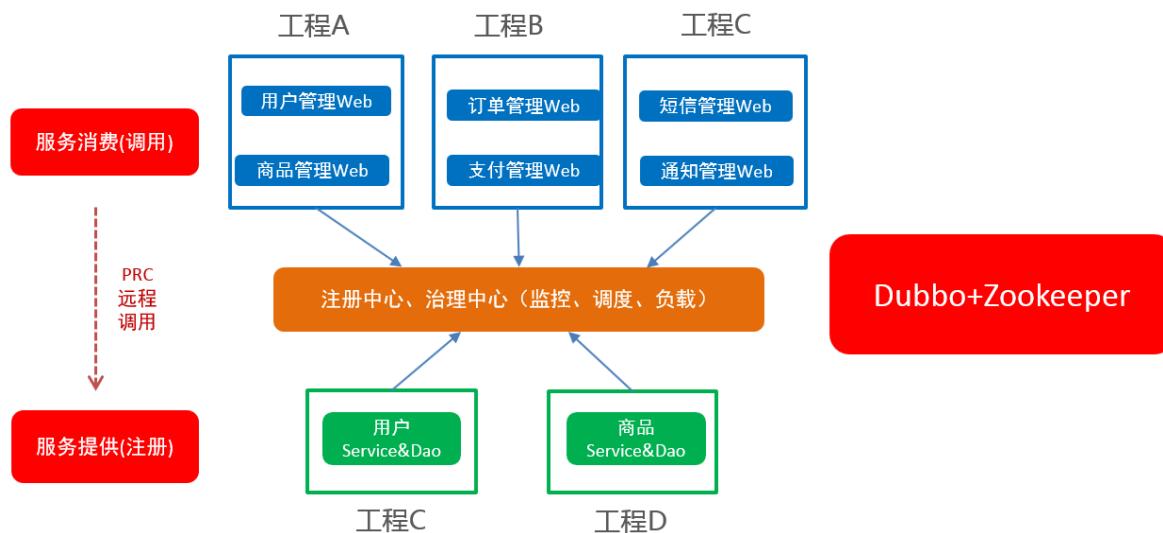
- (2) 维护成本易于拓展 (修改一个功能，可以直接修改一个项目，单独部署)
- (3) 并发量大易于解决 (搭建集群)
- (4) 技术栈完全扩展 (不同的系统可以用不同的编程语言编写)。

- 架构缺点：

缺少统一管理资源调度的框架

1.2.4. 流动计算架构 (SOA)

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键。



资源调度和治理中心的框架：dubbo+zookeeper

- 架构优点：

- (1) 业务代码完全解耦，并可实现通用
- (2) 维护成本易于拓展 (修改一个功能，可以直接修改一个项目，单独部署)
- (3) 并发量大易于解决 (搭建集群)
- (4) 技术栈完全扩展 (不同的系统可以用不同的编程语言编写)。

【小结】

1: 单体架构

全部功能集中在一个项目内 (All in one)。

2: 垂直架构

按照业务进行切割，形成小的单体项目。

3: SOA架构 (项目一)

面向服务的架构 (SOA) 是一个组件模型,全称为：Service-Oriented Architecture，它将应用程序的不同功能单元 (称为服务) 进行拆分，并通过这些服务之间定义良好的接口和契约联系起来。接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。这使得构建在各种各样的系统中的服务可以以一种统一和通用的方式进行交互。

可以使用dubbo作为调度的工具 (RPC协议)

4: 微服务架构 (项目二)

将系统服务层完全独立出来，抽取为一个一个的微服务。

特点一：抽取的粒度更细，遵循单一原则，数据可以在服务之间完成数据传输（一般使用restful请求调用资源）。

特点二：采用轻量级框架协议传输。（可以使用springcloudy）（http协议）

特点三：每个服务都使用不同的数据库，完全独立和解耦。

2. RPC（远程过程调用）

【目标】

了解什么是RPC

【路径】

1: RPC介绍

2: RPC组件

3: RPC调用

【讲解】

2.1. RPC介绍

Remote Procedure Call 远程过程调用，是分布式架构的核心，按响应方式分如下两种：

同步调用：客户端调用服务方方法，等待直到服务方返回结果或者超时，再继续自己的操作。

异步调用：客户端把消息发送给中间件，不再等待服务端返回，直接继续自己的操作。

- 是一种进程间的通信方式
- 它允许应用程序调用网络上的另一个应用程序中的方法
- 对于服务的消费者而言，无需了解远程调用的底层细节，是透明的

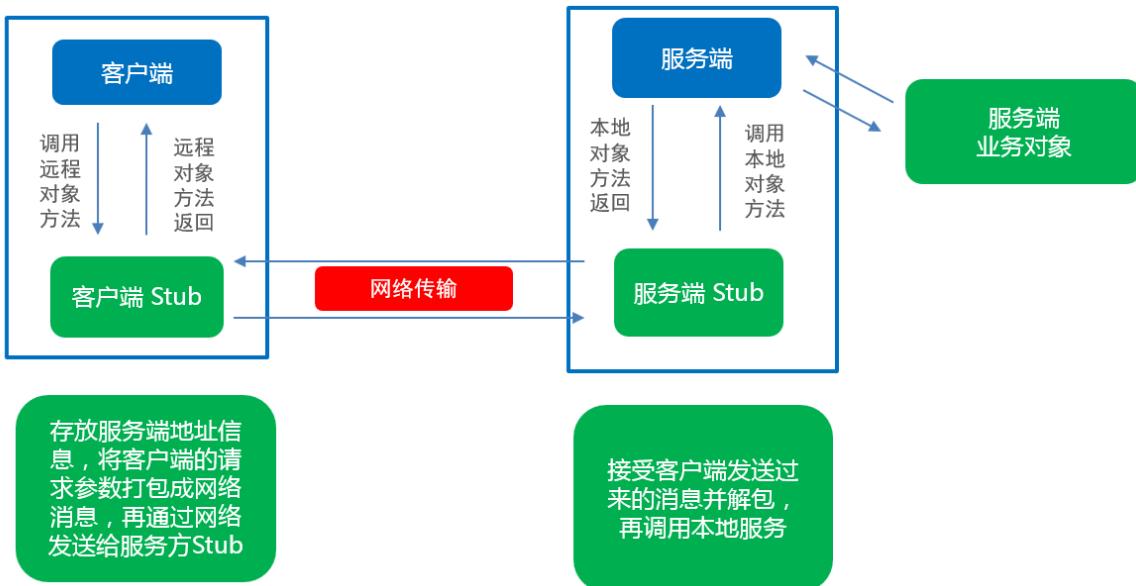
需要注意的是RPC并不是一个具体的技术，而是指整个网络远程调用过程。

RPC是一个泛化的概念，严格来说一切远程过程调用手段都属于RPC范畴。各种开发语言都有自己的RPC框架。Java中的RPC框架比较多，广泛使用的有RMI、Hessian、Dubbo、spring Cloud等。

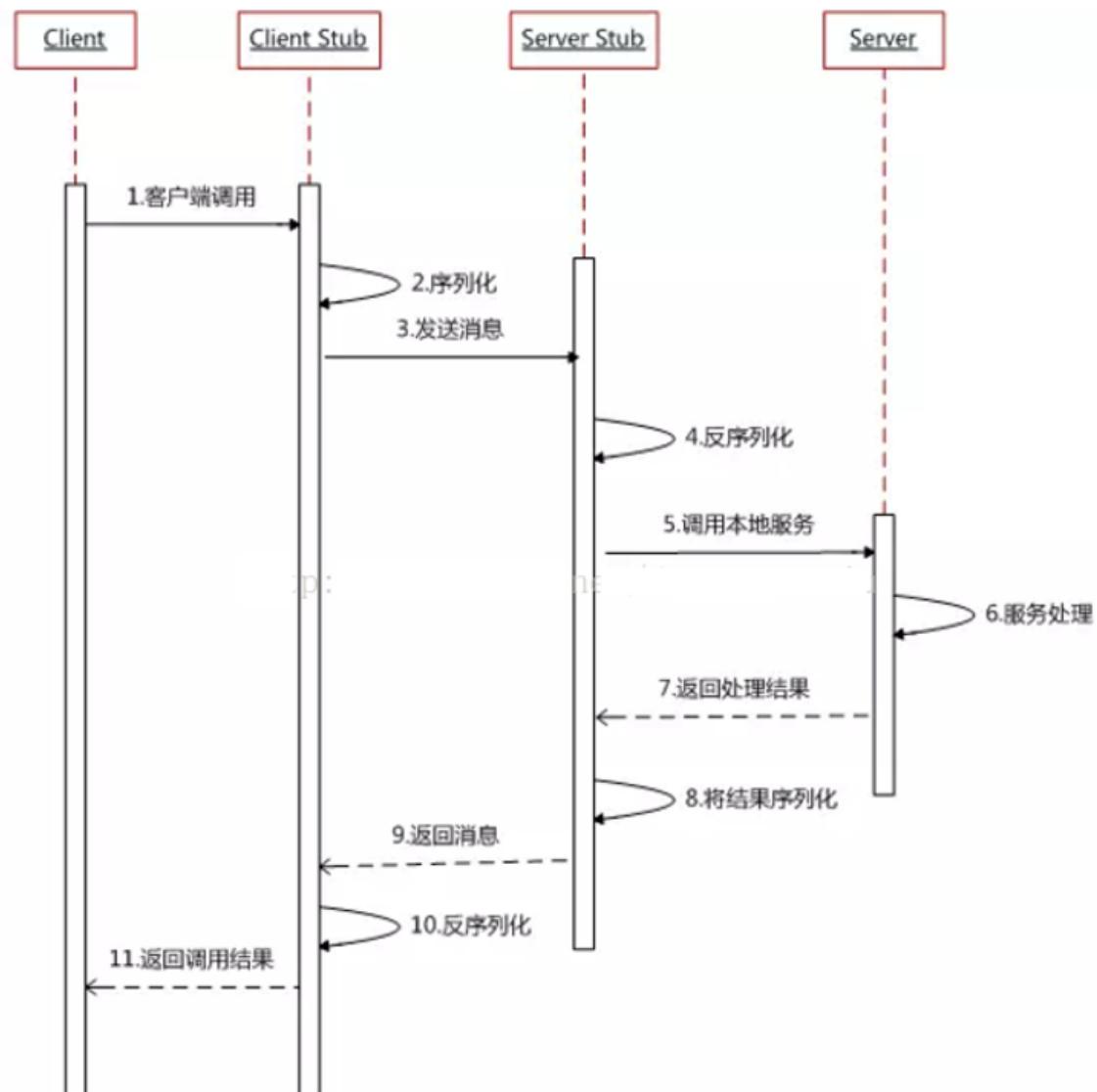
2.2. RPC组件

简单来说一个RPC架构里包含如下4个组件：

- 1、客户端(Client)：服务调用者
- 2、客户端存根(Client Stub)：存放服务端地址信息，将客户端的请求参数打包成网络消息，再通过网络发给服务方
- 3、服务端存根(Server Stub)：接受客户端发送过来的消息并解包，再调用本地服务
- 4、服务端(Server)：服务提供者。



2.3. RPC调用



- 1、服务调用方 (client) 调用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、参数等组装成能够进行网络传输的消息体

在Java里就是序列化的过程

- 3、 client stub找到服务地址，并将消息通过网络发送到服务端；
- 4、 server stub收到消息后进行解码,在Java里就是反序列化的过程；
- 5、 server stub根据解码结果调用本地的服务；
- 6、 本地服务执行处理逻辑；
- 7、 本地服务将结果返回给server stub；
- 8、 server stub将返回结果打包成消息，Java里的序列化；
- 9、 server stub将打包后的消息通过网络并发送至消费方；
- 10、 client stub接收到消息，并进行解码, Java里的反序列化；
- 11、 服务调用方（client）得到最终结果。

【小结】

- 1：RPC介绍
- 2：RPC组件
- 3：RPC调用

3. Apache Dubbo概述

【目标】

- 什么是dubbo？
- dubbo的架构是什么（图）？

【路径】

- 1：dubbo简介
- 2：dubbo架构

【讲解】

3.1. Dubbo简介

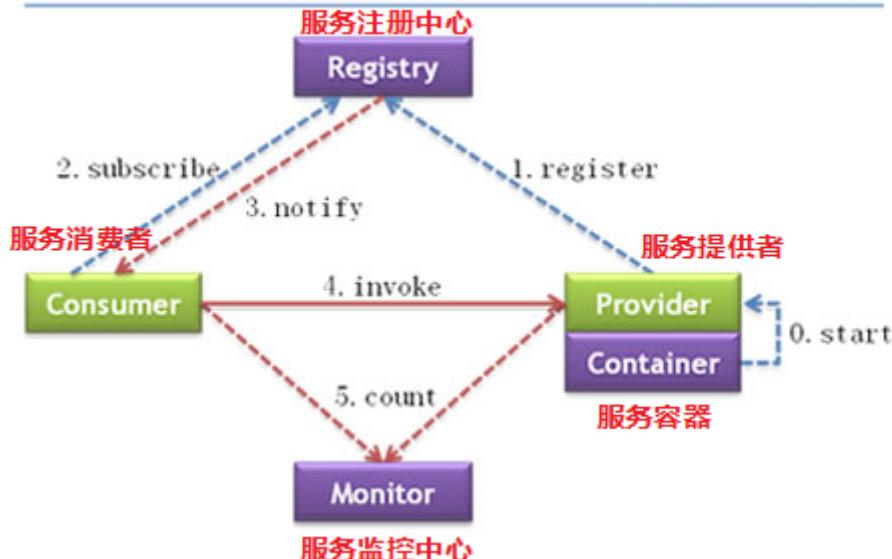
Apache Dubbo是一款高性能的Java RPC框架。其前身是阿里巴巴公司开源的一个高性能、轻量级的开源Java RPC框架，可以和Spring框架无缝集成。

Dubbo官网地址：<http://dubbo.apache.org>

Dubbo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。

3.2. Dubbo架构

Dubbo架构图（Dubbo官方提供）如下：



节点角色说明：

节点	角色名称
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

虚线都是异步访问，实线都是同步访问

蓝色虚线：在启动时完成的功能

红色虚线(实线)都是程序运行过程中执行的功能

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。

什么是长连接？

短连接是指通讯双方有数据交互时，就建立一个连接，数据发送完成后，则断开此连接，即每次连接只完成一项业务的发送。

长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是短连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，下次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接，如果用短连接频繁的通信会造成socket错误，而且频繁的socket创建也是对资源的浪费。

而像WEB网站的http服务一般都用**短连接**，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用**短连接**会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

总之，长连接和短连接的选择要视情况而定。

【小结】

1: dubbo简介

2: dubbo架构

4. Dubbo快速开发

【目标】

使用dubbo，完成服务消费者，调用，服务提供者方法

【路径】

1: 环境准备

2: 创建父工程 (dubbo_parent)

3: 创建公共子模块(dubbo_common)

4: 创建接口子模块(dubbo_interface)

5: 创建服务提供者模块(dubbo_provider)

6: 创建服务消费者模块(dubbo_consumer)

7: Zookeeper中存放Dubbo服务结构(注册中心)

【讲解】

Dubbo作为一个RPC框架，其最核心的功能就是要实现跨网络的远程调用，服务提供者、服务消费者会使用共同的接口，故本小节先创建一个父工程，父工程下有4个子模块，一个是公共子模块，一个是接口模块，一个是服务提供者模块，一个是服务消费者模块。通过Dubbo来实现服务消费方远程调用服务提供方的方法。

实现的业务为：根据id查询用户对象

业务描述：页面发送请求：user/findById.do?id=1 根据id从数据库获取用户对象

实现步骤：

- 环境准备：创建数据库，创建t_user表
- 创建父工程，基于maven，打包方式为pom，工程名：dubbo_parent
- 创建公共子模块，创建user实体类，打包方式为jar，工程名：dubbo_common
- 创建接口子模块，在父工程的基础上，打包方式为jar，模块名：dubbo_interface
- 创建服务提供者子模块，在父工程的基础上，打包方式为war，模块名：dubbo_provider
- 创建服务消费者子模块，在父工程的基础上，打包方式为war，模块名：dubbo_consumer

4.1. 环境准备

创建数据库表

```
create database itcast_dubbo;

CREATE TABLE `t_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(20) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

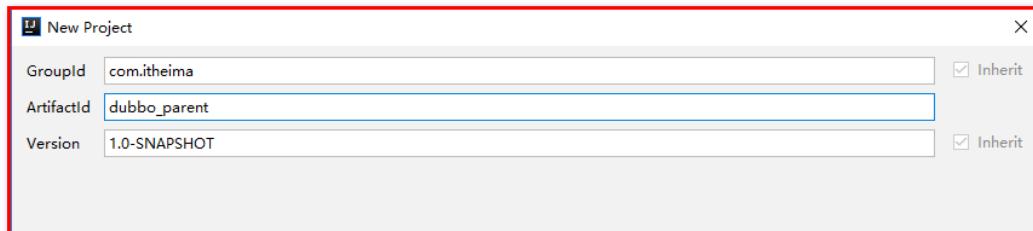
INSERT INTO t_user(username,age) VALUES("张三",22);
INSERT INTO t_user(username,age) VALUES("李四",20);
INSERT INTO t_user(username,age) VALUES("王五",25);
```

4.2. 创建父工程

父工程，不实现任何代码，主要是添加工程需要的库的依赖管理（DependencyManagement），依赖管理就是解决项目中多个模块间公共依赖的版本号、scope的控制范围。本项目需要使用spring-webmvc，使用dubbo（务必2.6.2以上版本）、zookeeper及其客户端（curator-framework）、Spring、Mybatis依赖库。

- GroupId:com.itheima
- ArtifactId:dubbo_parent

1. 创建完工程后，如图所示：



2. 修改pom.xml，抽取版本、增加依赖库管理，全部内容如下：

```
<?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_study</artifactId>
    <packaging>pom</packaging>
```

```
<version>1.0-SNAPSHOT</version>
<modules>
    <module>dubbo_common</module>
    <module>dubbo_interface</module>
    <module>dubbo_provider</module>
</modules>

<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <spring.webmvc.version>5.0.5.RELEASE</spring.webmvc.version>
    <dubbo.version>2.6.2</dubbo.version>
    <zookeeper.version>3.4.7</zookeeper.version>
    <curator.verion>4.0.1</curator.verion>
    <mybatis.version>3.4.5</mybatis.version>
    <mysql.version>5.1.47</mysql.version>
    <mybatis-spring.version>1.3.2</mybatis-spring.version>
    <druid.version>1.1.9</druid.version>
    <log.version>1.2.17</log.version>
    <slf4j.version>1.7.25</slf4j.version>
</properties>
<dependencyManagement>
    <dependencies>
        <!--springmvc的环境-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.webmvc.version}</version>
        </dependency>
        <!--dubbo的环境-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>dubbo</artifactId>
            <version>${dubbo.version}</version>
        </dependency>
        <!--zookeeper的环境-->
        <dependency>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
            <version>${zookeeper.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-framework</artifactId>
            <version>${curator.verion}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-recipes</artifactId>
            <version>${curator.verion}</version>
        </dependency>
        <!--mybatis的环境-->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>${mybatis.version}</version>
        </dependency>
    </dependencies>

```

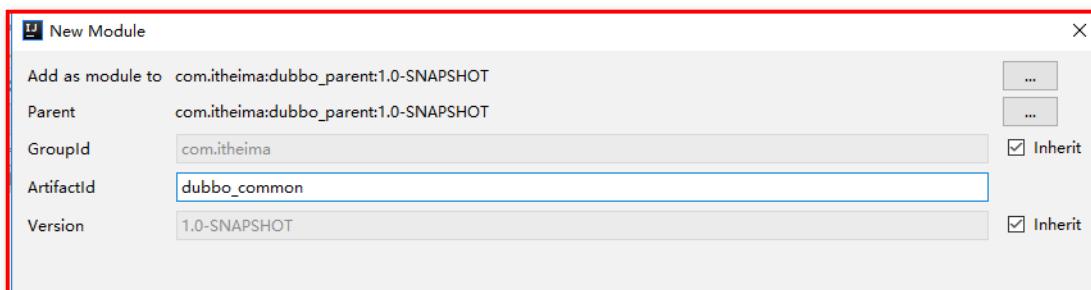
```

<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>${mybatis-spring.version}</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>${druid.version}</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>${mysql.version}</version>
</dependency>
<!--spring的环境-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${spring.webmvc.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.webmvc.version}</version>
</dependency>
<!--日志的环境-->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>${log.version}</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${slf4j.version}</version>
</dependency>
</dependencies>
</dependencyManagement>
</project>

```

4.3. 创建公共子模块

- 在当前父工程的基础上创建子模块
- GroupId:com.itheima
- ArtifactId:dubbo_common



1: pom.xml

```
<packaging>jar</packaging>
```

2: 在com.itheima.pojo包中创建User实体类

```
package com.itheima.pojo;

import java.io.Serializable;

public class User implements Serializable {
    private Integer id;
    private String username;
    private Integer age;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", username='" + username + '\'' +
            ", age=" + age +
            '}';
    }
}
```

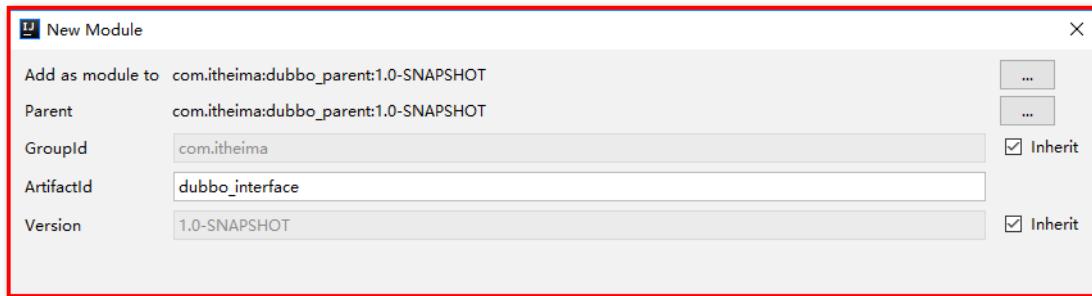
4.4. 创建接口子模块

此模块，主要放业务接口的定义，它是服务消费者模块和服务提供者模块的公共依赖模块。

- 在当前父工程的基础上创建子模块

- GroupId:com.itheima
- ArtifactId:dubbo_interface

1. 创建子模块:



2. pom.xml, 添加依赖

```
<packaging>jar</packaging>
<dependencies>
    <dependency>
        <groupId>com.itheima</groupId>
        <artifactId>dubbo_common</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
</dependencies>
```

3. 在包com.itheima.service创建业务接口

```
package com.itheima.service;

import com.itheima.pojo.User;

public interface UserService {
    public User findById(Integer id);
}
```

如图所示:



4.5. 服务提供者模块

此模块是服务提供者模块，需要在容器启动时，把服务注册到zookeeper,故需要引入spring-webmvc,zookeeper及客户端依赖。

【路径】

实现步骤：

1. 创建子模块dubbo_provider，使用war，导入依赖坐标mybatis, mybatis-spring, druid-data-source, mysql驱动包, spring-webmvc、dubbo依赖和编译插件tomcat7,还需要依赖dubbo_interface

2. java源代码目录

增加com.itheima.service.impl包及创建UserServiceImpl实现类，业务实现类UserServiceImpl，需要实现UserService接口

增加com.itheima.dao包以及创建UserDao接口，在resources资源目录中增加com/itheima/dao，创建UserDao.xml映射文件

3. resources资源目录

◦ 创建spring-dubbo.xml 配置文件：

- 配置dubbo的应用名称
- 配置dubbo注册中心Zookeeper地址
- 配置需要暴露的业务接口及实例

◦ 创建spring-dao.xml

- 配置数据源
- 配置sqlSessionFactory对象
- 扫描dao包，创建dao接口的动态代理对象

◦ 创建spring-service.xml

- 开启注解扫描，扫描service包
- 配置事务

4. 将资料的log4j.properties配置文件拷贝到resources目录下

5. 在web.xml文件中，配置spring监听器，指定spring配置文件加载位置（需要加载多个spring的配置文件）

6. 启动服务提供者

实现过程：

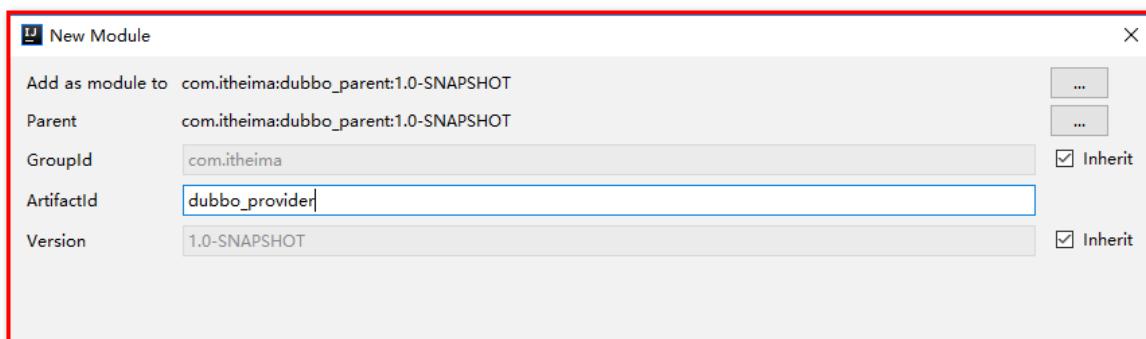
4.5.1. 创建子模块dubbo_provider（或者叫做dubbo_service）

- 在当前父工程的基础上创建子模块
- 打包方式为war
- 项目坐标如下

GroupId:com.itheima

ArtifactId:dubbo_provider

工程创建完成后，如图所示：



修改pom.xml文件， 打包方式为war， 增加依赖库， 新增编译插件tomcat7， 端口81

```
<packaging>war</packaging>

<dependencies>
    <dependency>
        <groupId>com.itheima</groupId>
        <artifactId>dubbo_interface</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>dubbo</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.zookeeper</groupId>
        <artifactId>zookeeper</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.curator</groupId>
        <artifactId>curator-framework</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.curator</groupId>
        <artifactId>curator-recipes</artifactId>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
    </dependency>
</dependencies>

<build>
```

```
<finalName>dubbo_provider</finalName>
<plugins>
    <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <configuration>
            <path>/</path>
            <port>81</port>
        </configuration>
    </plugin>
</plugins>
</build>
```

4.5.2. 初始化java资源目录

1: 在main下,创建子目录java (刷新maven工程) ,增加com.itheima.service.impl包及创建 UserServiceImpl实现类

```
package com.itheima.service.impl;

import com.itheima.pojo.User;
import com.itheima.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;

public class UserServiceImpl implements UserService {

    @Autowired
    UserDao userDao;

    @Override
    public User findById(Integer id) {
        return userDao.findById(id);
    }
}
```

2: 增加com.itheima.dao包, 创建UserDao接口

```
package com.itheima.dao;

import com.itheima.pojo.User;

public interface UserDao {
    User findById(Integer id);
}
```

3: 在resource下创建com/itheima/dao目录, 创建UserDao接口的映射文件, 内容如下

```

<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.itheima.dao.UserDao">
    <select id="findById" resultType="com.itheima.pojo.User"
parameterType="int">
        select * from t_user where id = #{id}
    </select>
</mapper>

```

4.5.3. 初始化resources目录

在main下创建子目录resources目录，在resources目录下

创建spring-dubbo.xml，

spring-dao.xml，

spring-service.xml 配置文件

创建log4j的配置文件。

1: 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://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">

    <!-- dubbo应用的名称-->
    <dubbo:application name="dubbo-provider"/>
    <!-- 服务注册中新的地址-->
    <dubbo:registry address="zookeeper://127.0.0.1:2181"/>

    <!--指定暴露的服务接口及实例-->
    <dubbo:service interface="com.itheima.service.UserService"
                  ref="userServicve"/>
    <!--配置业务类实例-->
    <bean id="userServicve"
          class="com.itheima.service.impl.UserServiceImpl"/>
</beans>

```

2: jdbc.properties

```

jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/itcast_dubbo
jdbc.user=root
jdbc.password=admin

```

3: spring-dao.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:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
           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.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx.xsd
       ">

    <!--加载属性文件-->
    <context:property-placeholder location="classpath:jdbc.properties">
    </context:property-placeholder>
    <!--配置数据源-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.user}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
    <!--配置sqlSessionFactory-->
    <bean id="sqlSessionFactory"
          class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <!--配置别名映射-->
        <property name="typeAliasesPackage" value="com.itheima.pojo"/>
        <!-- 分页插件pagehelper: 后续配置 -->
    </bean>
    <!--配置mapper文件扫描-->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="com.itheima.dao"/>
    </bean>
</beans>

```

4: spring-service.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:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
           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.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx.xsd
       ">

    <!--配置事务管理器-->
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

```

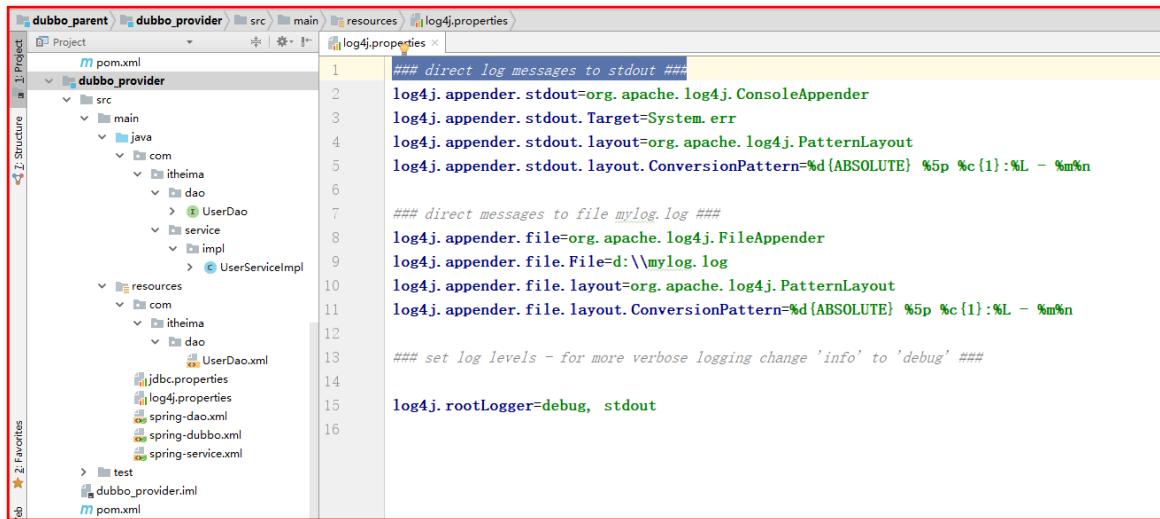
```

<property name="dataSource" ref="dataSource"/>
</bean>
<!--事务注解驱动，在Service类上添加@Transactional-->
<tx:annotation-driven/>
</beans>

```

将资料的log4j.properties配置文件拷贝到resources目录下

效果如下



4.5.4. 配置web.xml文件

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://java.sun.com/xml/ns/javaee"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
          id="WebApp_ID" version="3.0">

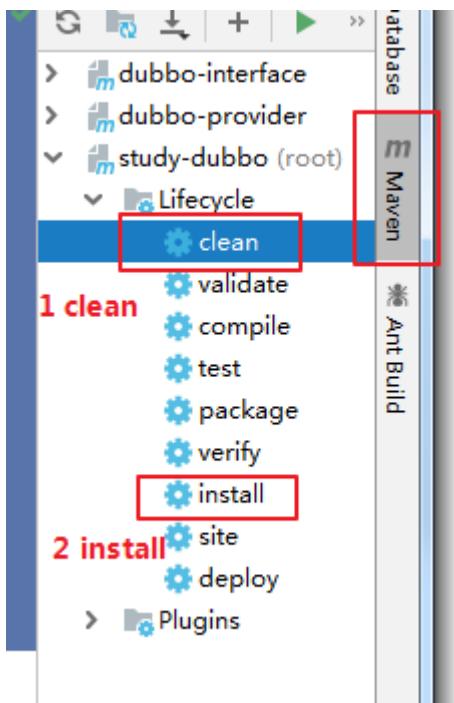
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring*.xml</param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
</web-app>

```

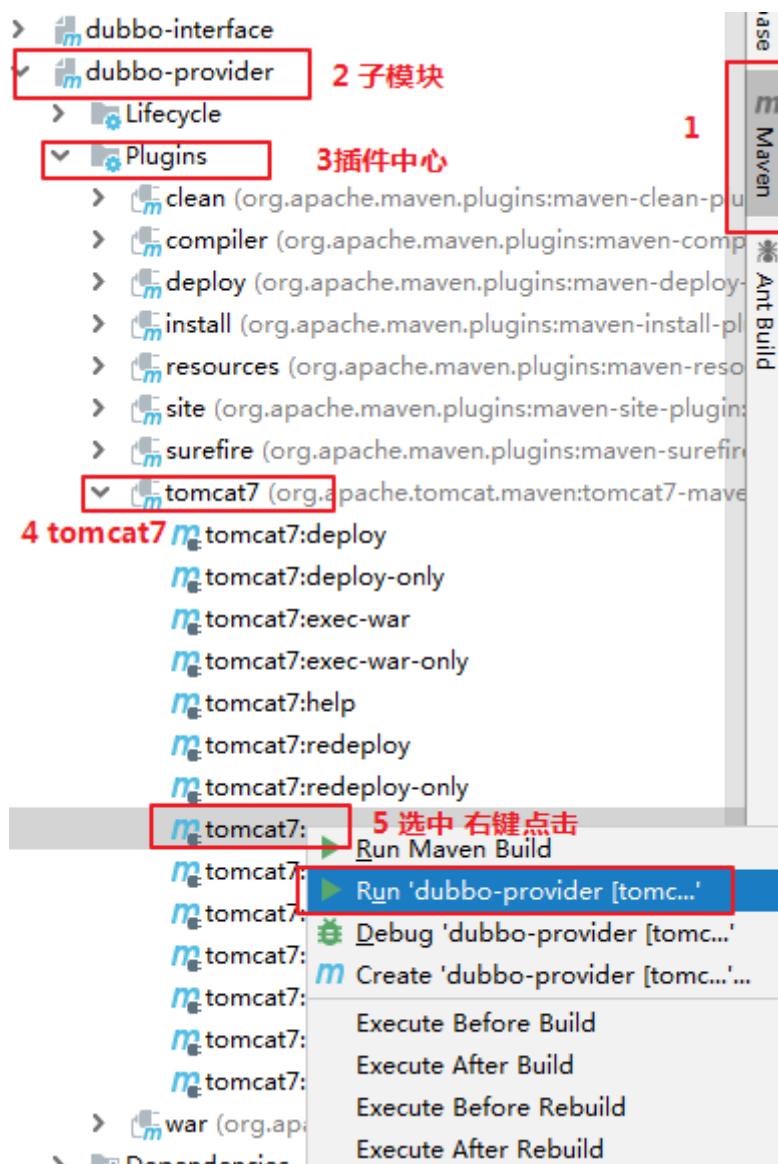
4.5.5. 启动服务提供者

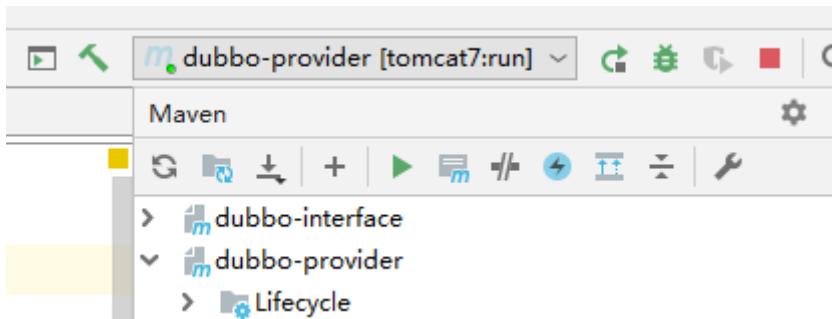
需要在父工程整体clean、install后，在子模块使用tomcat7插件运行

1、父工程clean、install



2、启动当前应用，启动后默认注册服务到zookeeper注册中心





3、检查是否注册到zookeeper

- (1) 启动zookeeper，作为dubbo的注册中心
- (2) 登录zookeeper客户端，直接查看ls /dubbo/com.itheima.service.UserService节点

```
numChildren = 1
[zk: localhost:2181(CONNECTED) 11] ls /dubbo/com.itheima.service.UserService/providers
[]
```

- 如果 /dubbo下面没有这个节点，说明没有注册上，
- 如果有，内容是空，说明已经掉线
- 正常注册并连接在线，如图所示：

```
[zk: localhost:2181(CONNECTED) 4] ls /dubbo/com.itheima.service.UserService/providers
[dubbo%3A%2F%2F192.168.175.1%3A20880%2Fcom.itheima.service.UserService%3Fanyhost%3Dtrue%26application%3Ddubbo-provider%26dubbo%3D2.6.2%26generic%3Dfalse%26interface%3Dcom.itheima.service.UserService%26methods%3DfindById%26pid%3D15204%26revision%3D1.0-SNAPSHOT%26side%3Dprovider%26timestamp%3D1579188434365]
```

注意：

- 消费者与提供者应用名称不能相同
- 如果有多个服务提供者，名称不能相同，通信端口也不能相同
- 只有服务提供者才会配置服务发布的协议，默认是dubbo协议，端口号是20880

可以在spring-dubbo.xml中配置协议的端口

```
<!--发布dubbo协议，默认端口20880-->
<dubbo:protocol name="dubbo" port="20881"></dubbo:protocol>
```

4.6. 服务消费者模块

此模块是服务消费者模块，此模块基于Web应用，需要引入spring-webmvc，需要在容器启动时，去zookeeper注册中心订阅服务，需要引入dubbo、zookeeper及客户端依赖。

实现步骤：

1. 创建子模块dubbo_consumer，打包方式为war包，导入依赖坐标spring-webmvc、dubbo、log4j，编译插件tomcat7以及dubbo_interface（用来调用接口方法）
2. java源代码目录
增加com.itheima.controller包及创建UserController类
控制器类UserController，提供web方法findById(Integer id)
3. resources资源目录
创建spring-dubbo.xml 配置文件：
配置dubbo的应用名称

配置dubbo注册中心Zookeeper地址

配置需要订阅的业务接口及引用

创建spring-mvc.xml

开启注解扫描

开启mvc注解驱动

4. 将资料的log4j.properties配置文件拷贝到resources目录下

5. 在web.xml文件中，配置SpringMVC

6. 启动服务消费者，并测试访问

实现过程：

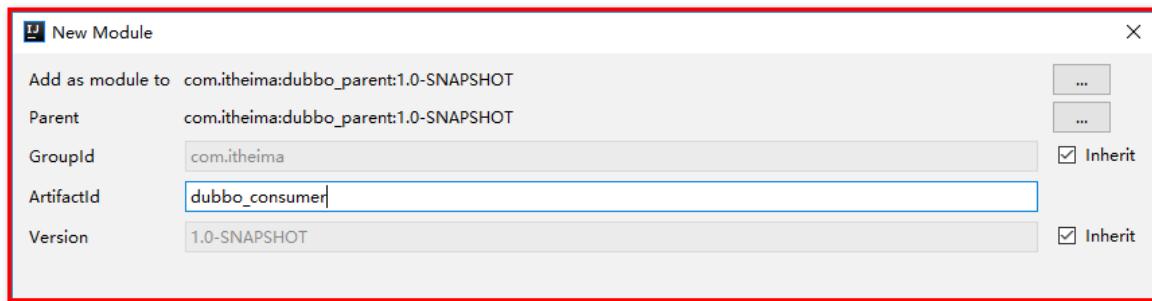
4.6.1. 创建子模块dubbo_consumer（或者叫做dubbo_controller）

- 在当前父工程的基础上创建子模块
- 坐标如下

GroupId:com.itheima

ArtifactId:dubbo_consumer

1：创建工程，如图所示：



2：修改pom.xml文件，增加依赖库，新增编译插件tomcat7,端口80

```
<packaging>war</packaging>

<dependencies>
    <dependency>
        <groupId>com.itheima</groupId>
        <artifactId>dubbo_interface</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.9.0</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.9.0</version>
    </dependency>
    <dependency>
```

```

        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>2.9.0</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>dubbo</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.zookeeper</groupId>
        <artifactId>zookeeper</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.curator</groupId>
        <artifactId>curator-framework</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.curator</groupId>
        <artifactId>curator-recipes</artifactId>
    </dependency>
</dependencies>

<build>
    <finalName>dubbo_consumer</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.tomcat.maven</groupId>
            <artifactId>tomcat7-maven-plugin</artifactId>
            <configuration>
                <path>/</path>
                <port>80</port>
            </configuration>
        </plugin>
    </plugins>
</build>

```

4.6.2. 初始化java资源目录

在main下,创建子目录java (刷新maven工程) ,增加com.itheima.controller包及创建UserController控制类 (通过配置文件初始化) , 该类调用远程业务UserService, 实现调用findById方法。

```

package com.itheima.controller;

import com.itheima.pojo.User;
import com.itheima.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

```

```

@Controller
@ResponseBody
@RequestMapping("/user")
public class UserController {

    @Autowired
    UserService userService;

    @RequestMapping("/findById")
    public User findById(Integer id){
        User user = userService.findById(id);
        return user;
    }
}

```

4.6.3. 初始化resources资源目录

在main下,创建子目录resources目录, 在resources目录下

1: 创建spring-dubbo.xml ,

2: spring-mvc.xml配置文件

3: 创建log4j的配置文件。

1: spring-dubbo.xml文件, 如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       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">
    <!-- dubbo基本配置-->
    <dubbo:application name="dubbo-consumer"/>
    <dubbo:registry address="zookeeper://127.0.0.1:2181"/>

    <!--订阅远程服务对象, id的名称和Controller类中的UserService接口名称要一致-->
    <dubbo:reference id="userService"
interface="com.itheima.service.UserService"/>

</beans>

```

2: spring-mvc.xml配置, 如下

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       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://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://code.alibabatech.com/schema/dubbo
    http://code.alibabatech.com/schema/dubbo/dubbo.xsd">
    <!--自动扫包-->
    <context:component-scan base-package="com.itheima.controller"/>
    <!--mvc注解驱动-->
    <mvc:annotation-driven></mvc:annotation-driven>

    <!--导入dubbo的配置-->
    <import resource="classpath:spring-dubbo.xml"></import>
</beans>

```

3: 导入log4j.properties

4.6.4. 配置web.xml

配置springmvc的核心控制器，用来加载spring-mvc.xml

```

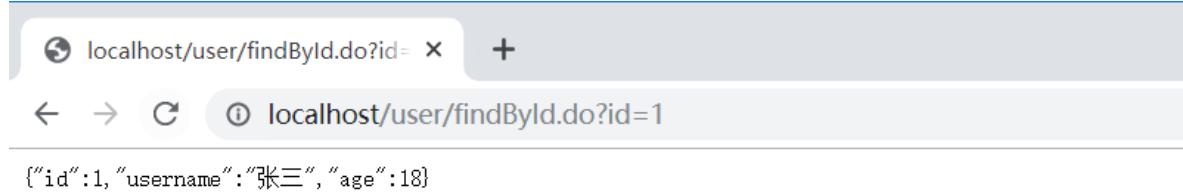
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://java.sun.com/xml/ns/javaee"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
          id="WebApp_ID" version="3.0">
    <!--springmvc的核心控制器-->
    <servlet>
        <servlet-name>springMVC</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>springMVC</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <!--解决post请求的乱码过滤器-->
    <filter>
        <filter-name>CharacterEncodingFilter</filter-name>
        <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>

```

```
</filter-mapping>  
</web-app>
```

4.6.5. 启动服务消费者并测试

在浏览器输入<http://localhost:80/user/findById.do?id=1>, 查看浏览器输出结果



注意：因为是RPC的框架，要求传递的参数和实体类要实现序列化

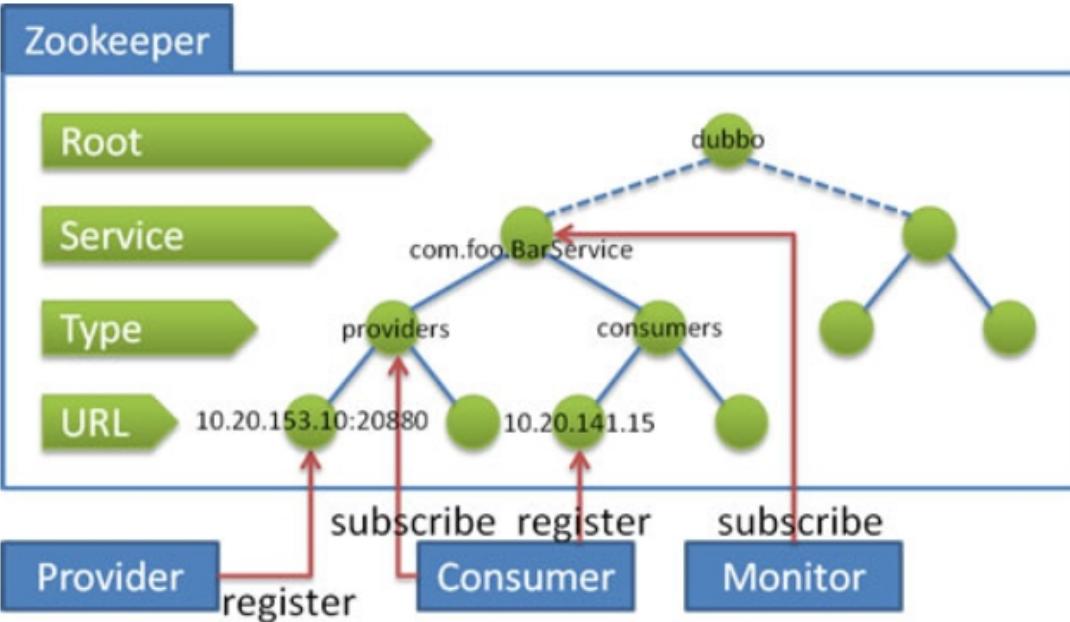
参数：Integer类型（实现序列化接口java.io.Serializable）

返回值：User（实现序列化接口java.io.Serializable），如果不进行序列化，抛出异常

```
HTTP Status 500 - Request processing failed; nested exception is com.alibaba.dubbo.rpc.RpcException: Failed to invoke  
findById in the service com.itheima.service.UserService. Tried 3 times of the providers [172.18.81.1:20880] (1/1) from  
127.0.0.1:2181 on the consumer 172.18.81.1 using the dubbo version 2.6.2. Last error is: Failed to invoke remote method  
provider: dubbo://172.18.81.1:20880/com.itheima.service.UserService?  
anyhost=true&application=dubbo_consumer&check=false&dubbo=2.6.2&generic=false&interface=com.itheima.servic  
SNAPSHOT&side=consumer&timestamp=1581403157871, cause: Failed to send response: Response [id=2, version=2.0,  
status=20, event=false, error=null, result=RpcResult [result=User{id=3, username='张三', age=22}, exception=null]],  
java.lang.IllegalStateException: Serialized class com.itheima.pojo.User must implement java.io.Serializable  
  
type Exception report  
  
message Request processing failed; nested exception is com.alibaba.dubbo.rpc.RpcException: Failed to invoke the method findById in the service com.itheima.service.UserService. Tried 3 times of the providers [172.18.81.1:20880] (1/1) from  
127.0.0.1:2181 on the consumer 172.18.81.1 using the dubbo version 2.6.2. Last error is: Failed to invoke remote method: findById, provider: dubbo://172.18.81.1:20880/com.itheima.service.UserService?  
anyhost=true&application=dubbo_consumer&check=false&dubbo=2.6.2&generic=false&interface=com.itheima.service.UserService&methods=findById&pid=6032&register.ip=172.18.81.1&remote.timestamp=158140315561  
SNAPSHOT&side=consumer&timestamp=1581403157871, cause: Failed to send response: Response [id=2, version=2.0, status=20, event=false, error=null, result=RpcResult [result=User{id=3, username='张三', age=22}, exception=null]],  
java.lang.IllegalStateException: Serialized class com.itheima.pojo.User must implement java.io.Serializable  
  
description The server encountered an internal error that prevented it from fulfilling this request.  
  
exception  
org.springframework.web.util.NestedServletException: Request processing failed; nested exception is com.alibaba.dubbo.rpc.RpcException: Failed to invoke the method findById in the service com.itheima.service.UserService. Tried 3 times of the providers [172.18.81.1:20880] (1/1) from  
127.0.0.1:2181 on the consumer 172.18.81.1 using the dubbo version 2.6.2. Last error is: Failed to invoke remote method: findById, provider: dubbo://172.18.81.1:20880/com.itheima.service.UserService?  
anyhost=true&application=dubbo_consumer&check=false&dubbo=2.6.2&generic=false&interface=com.itheima.service.UserService&methods=findById&pid=6032&register.ip=172.18.81.1&remote.timestamp=158140315561  
SNAPSHOT&side=consumer&timestamp=1581403157871, cause: Failed to send response: Response [id=2, version=2.0, status=20, event=false, error=null, result=RpcResult [result=User{id=3, username='张三', age=22}, exception=null]],  
java.lang.IllegalStateException: Serialized class com.itheima.pojo.User must implement java.io.Serializable  
  
at com.alibaba.com.caucho.hessian.io.SerializerFactory.getDefaultSerializer(SerializerFactory.java:395)  
at com.alibaba.com.caucho.hessian.io.SerializerFactory.getSerializer(SerializerFactory.java:369)  
at com.alibaba.com.caucho.hessian.io.Hessian2Output.writeObject(Hessian2Output.java:389)
```

4.7. Zookeeper中存放Dubbo服务结构（作为Dubbo运行的注册中心）

Zookeeper树型目录服务：



流程说明：

- 1：服务提供者(Provider)启动时: 向 /dubbo/com.foo.BarService/providers 目录下写入自己的 URL 地址
- 2：服务消费者(Consumer)启动时: 订阅 /dubbo/com.foo.BarService/providers 目录下的提供者 URL 地址。并向 /dubbo/com.foo.BarService/consumers 目录下写入自己的 URL 地址
- 3：监控中心(Monitor)启动时: 订阅 /dubbo/com.foo.BarService 目录下的所有提供者和消费者 URL 地址

【小结】

- 1: 环境准备
- 2: 创建父工程 (dubbo_parent)
- 3: 创建公共子模块(dubbo_common)
- 4: 创建接口子模块(dubbo_interface)
- 5: 创建服务提供者模块(dubbo_provider)
- 6: 创建服务消费者模块(dubbo_consumer)
- 7: Zookeeper中存放Dubbo服务结构(注册中心)

5. Dubbo管理控制台

我们在开发时，需要知道Zookeeper注册中心都注册了哪些服务，有哪些消费者来消费这些服务。我们可以通过部署一个管理中心来实现。其实管理中心就是一个web应用，部署到tomcat即可。

【目标】

Dubbo管理控制台的使用（即Dubbo监控中心）

【路径】

- 1: 安装 (dubbo-admin.war)
- 2: 使用 (dubbo-admin.war)

【讲解】

5.1. 安装

安装步骤：

- (1) 将资料中的dubbo-admin.war文件复制到tomcat的webapps目录下
- (2) 启动tomcat，此war文件会自动解压
- (3) 修改WEB-INF下的dubbo.properties文件

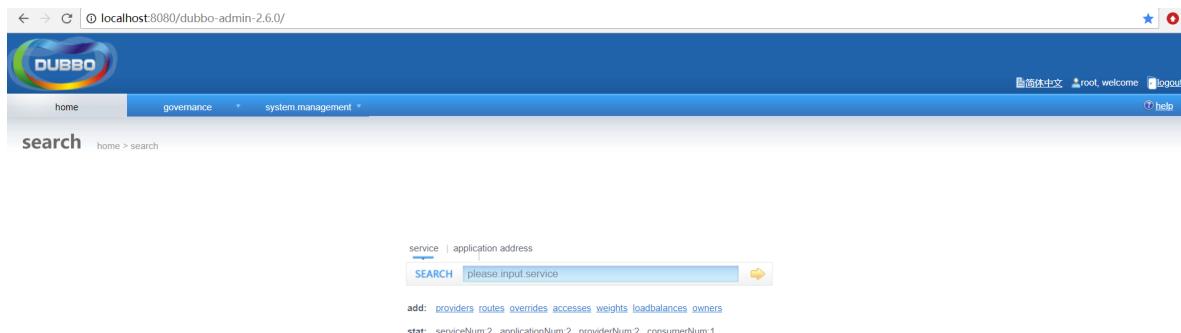
```
# 注意dubbo.registry.address对应的值需要对应当前使用的Zookeeper的ip地址和端口号
dubbo.registry.address=zookeeper://localhost:2181
dubbo.admin.root.password=root
dubbo.admin.guest.password=guest
```

- (4) 重启tomcat

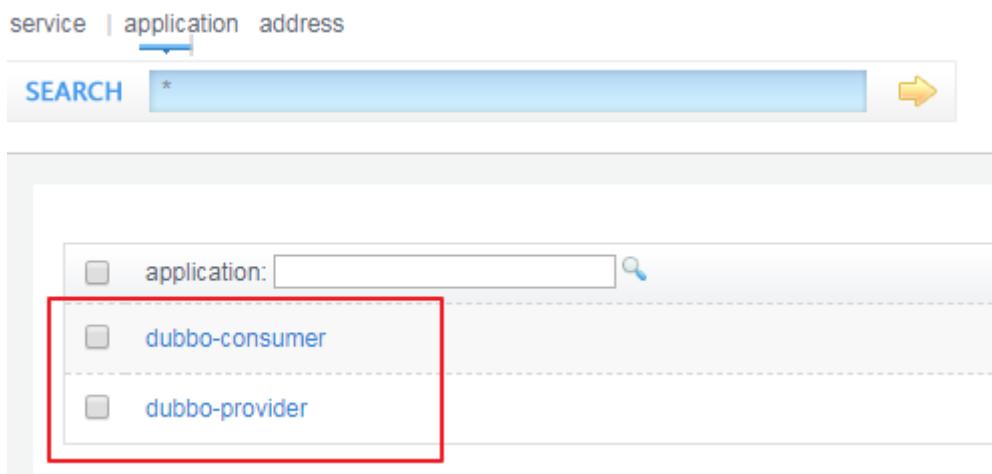
5.2. 使用

操作步骤：

- (1) 访问<http://localhost:8080/dubbo-admin/>，输入用户名(root)和密码(root)



- (2) 启动服务提供者工程和服务消费者工程，可以在查看到对应的信息



The screenshot shows two main sections of the Dubbo Admin interface:

- Providers Section:** Shows a search bar with "com.itheima.study.dubbo.service.UserService" and a list of providers. One provider entry is highlighted with a red box: "192.168.31.76:8888" (address), "100" (weight), "dynamic" (type), "enabled" (status), and "ok" (check).
- Consumers Section:** Shows a search bar with "192.168.31.76" and a list of consumers. One consumer entry is highlighted with a red box: "192.168.31.76" (address), "dubbo-consumer" (application), "Allowed" (access), "no.mocked" (mock), "unrouted" (Route), and "notified(1)" (notify).

【小结】

1: 安装 (dubbo-admin.war) , 放置到tomcat, 修改WEB-INF下的dubbo.properties文件

2: 使用 (dubbo-admin.war)

访问<http://localhost:8080/dubbo-admin/>, 输入用户名(root)和密码(root)

注意: 这里需要使用jdk1.8

6. Dubbo相关配置说明

【目标】

Dubbo相关配置说明

【路径】

1: 包扫描 (dubbo注解配置)

2: 服务接口访问协议

- dubbo协议
- rmi协议

3: 启动时检查

4: 超时调用

【讲解】

6.1. 包扫描

```
<dubbo:annotation package="com.itheima.service" />
```

服务提供者和服务消费者前面章节实现都是基于配置文件进行服务注册与订阅，如果使用包扫描，可以使用注解方式实现，推荐使用这种方式。

61.1. 服务提供者，使用注解实现

第一步：在spring-dubbo.xml中配置

```
<dubbo:annotation package="com.itheima.service" />
```

服务提供者和服务消费者都需要配置，表示包扫描，作用是扫描指定包(包括子包)下的类。

同时去掉以下配置：

```
<!--指定暴露的服务接口及实例-->
<dubbo:service interface="com.itheima.service.userService"
    ref="userService"/>
<!--配置业务类实例-->
<bean id="userService"
    class="com.itheima.service.impl.UserServiceImpl"/>
```

第二步：在HelloServiceImpl的类上使用注解：

```
@Service
public class UserServiceImpl implements UserService { }
```

注意：其中@Service是dubbo包下（com.alibaba.dubbo.config.annotation.Service）的注解。表示提供服务

6.1.2. 服务消费者，使用注解实现

第一步：在spring-consumer.xml中配置

```
<dubbo:annotation package="com.itheima.controller" />
```

这里的dubbo注解扫描

同时去掉：

```
<!--订阅远程服务对象，id的名称和Controller类中的UserService接口名称要一致-->
<dubbo:reference id="userService" interface="com.itheima.service.UserService"/>
```

第二步：在Controller类中使用

@Reference注解

```
@Controller
@ResponseBody
@RequestMapping("/user")
public class UserController {

    // @Autowired
    @Reference
    UserService userService;
}
```

注意：其中@Reference是dubbo包下（com.alibaba.dubbo.config.annotation.Reference）的注解。
表示订阅服务

6.1.3. 重启服务测试使用

- 重启服务提供者模块 dubbo-provider
- 重启服务消费者模块 dubbo-consumer
- 在浏览器输入测试URL：

<http://localhost/user/findById.do?id=1>, 查看浏览器输出结果。

6.2. 服务接口访问协议

```
<dubbo:protocol name="dubbo" port="20881"></dubbo:protocol>
```

一般在服务提供者一方配置，可以指定使用的协议名称和端口号。

其中Dubbo支持的协议有：dubbo、rmi、hessian、http、webservice、rest、redis等。

推荐使用的是dubbo协议，默认端口号：20880。

dubbo 协议采用单一长连接和 NIO 异步通讯，适合于小数据量、大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。不适合传送大数据量的服务，比如传文件，传视频等，除非请求数量很低。

也可以在同一个工程中配置多个协议，不同服务可以使用不同的协议，例如：

```
<!-- 多协议配置 -->
<dubbo:protocol name="dubbo" port="20881" />
<dubbo:protocol name="rmi" port="1099" />
<!-- 使用dubbo协议暴露服务 -->
<dubbo:service interface="com.itheima.service.HelloService" ref="helloService"
protocol="dubbo" />
<!-- 使用rmi协议暴露服务 -->
<dubbo:service interface="com.itheima.service.DemoService" ref="demoService"
protocol="rmi" />
<!-- 使用dubbo rmi协议暴露服务 -->
<dubbo:service interface="com.itheima.service.UserService" ref="userService"/>

<bean id="helloService" class="com.itheima.service.impl.HelloServiceImpl" />
<bean id="demoService" class="com.itheima.service.impl.DemoServiceImpl" />
<bean id="userService" class="com.itheima.service.impl.UserServiceImpl" />
```

同时去掉：

```
<dubbo:annotation package="com.itheima.service" />
```

也可以使用注解配置多个协议

```
@Service(protocol = "dubbo")
public class HelloServiceImpl implements HelloService { }
```

```
@Service(protocol = "rmi")
public class DemoServiceImpl implements DemoService {
```

dubbo协议：

- 连接个数：单连接
- 连接方式：长连接
- 传输协议：TCP
- 传输方式：NIO异步传输
- 序列化：Hessian二进制序列化
- 适用范围：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，单一消费者无法压满提供者，尽量不要用dubbo协议传输大文件或超大字符串。
- 适用场景：常规远程服务方法调用

rmi协议：

- 连接个数：多连接
- 连接方式：短连接
- 传输协议：TCP
- 传输方式：同步传输
- 序列化：Java标准二进制序列化
- 适用范围：传入传出参数数据包大小混合，消费者与提供者个数差不多，可传文件。
- 适用场景：常规远程服务方法调用，与原生RMI服务互操作

详情使用可通过博客文章：<https://www.cnblogs.com/duanxz/p/3555876.html>了解

注意：测试完成后，将rmi相关代码和配置注释掉

启用dubbo扫描配置：`<dubbo:annotation package="com.itheima.service" />`

6.3. 启动时检查

```
<dubbo:consumer check="false"/>
```

上面这个配置需要配置在服务消费者一方，如果不配置默认check值为true。Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，以便上线时，能及早发现问题。可以通过将check值改为false来关闭检查。

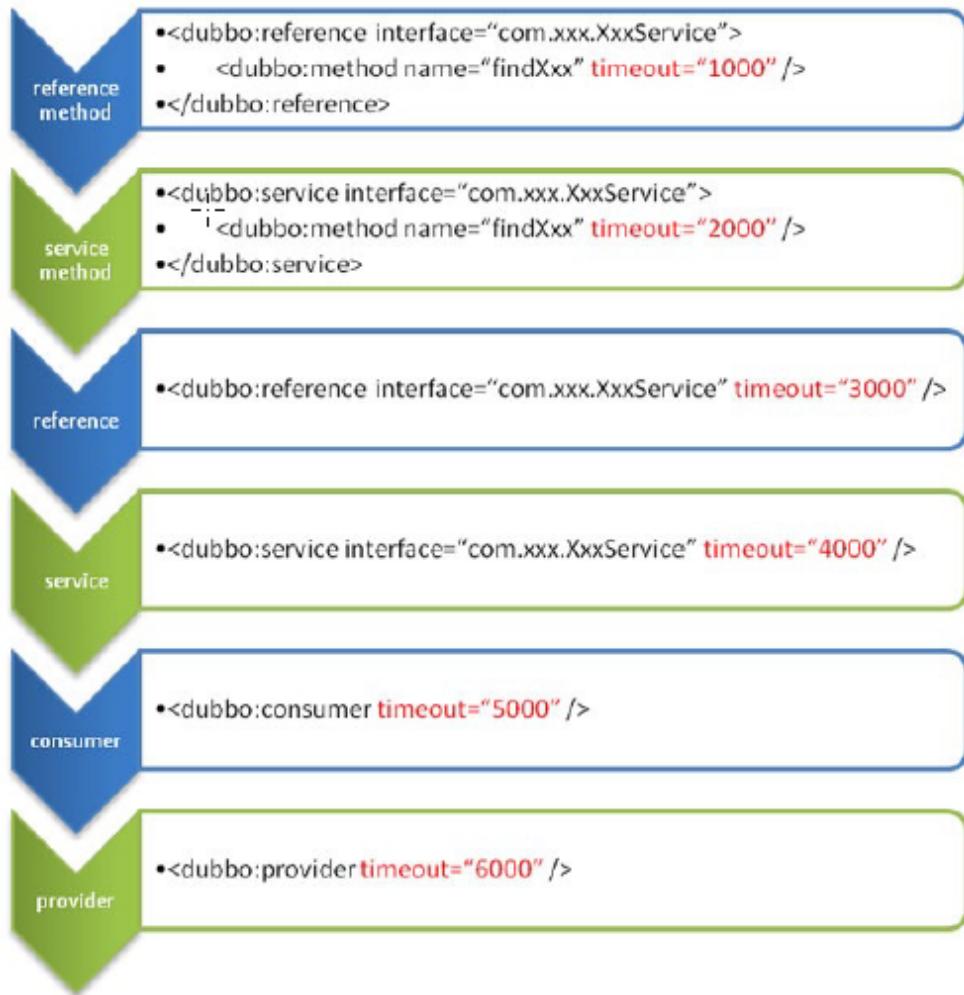
建议在开发阶段将check值设置为false，在生产环境下改为true。

如果设置为true，启动服务消费者，会抛出异常，表示没有服务提供者

```
at java.lang.Thread.run(Thread.java:748)
Caused by: java.lang.IllegalStateException: Failed to check the status of the service com.itheima.service.UserService. No provider available
at com.alibaba.dubbo.config.ReferenceConfig.createProxy(ReferenceConfig.java:422)
at com.alibaba.dubbo.config.ReferenceConfig.init(ReferenceConfig.java:333)
```

6.4. 超时调用

默认的情况下，dubbo调用的时间为一秒钟，如果超过一秒钟就会报错，所以我们可以设置超时时间长些，保证调用不出问题，这个时间需要根据业务来进行确定。



建议由服务提供方设置超时，因为一个方法需要执行多长时间，服务提供方更清楚，如果一

(1) 修改消费者配置文件,增加如下配置:

```

<!--超时时间为3秒钟-->
<dubbo:consumer timeout="3000"></dubbo:consumer>

```

(2) 修改提供者配置文件，增加如下配置

```

<!--超时时间设置为3秒钟-->
<dubbo:provider timeout="3000"></dubbo:provider>

```

(3) 修改UserServiceImpl代码测试

```
@Override  
public User findById(Integer id) {  
    System.out.println("开始睡眠。 . . . . ");  
    try {  
        Thread.sleep(3000);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    return userDao.findById(id);  
}
```

【小结】

1: 包扫描 (dubbo注解配置)

```
<dubbo:annotation package="com.itheima"></dubbo:annotation>
```

2: 服务接口访问协议

(服务提供者)

- dubbo协议
- rmi协议

```
<!--配置Dubbo的协议 (dubbo协议, 默认端口20880-->  
<dubbo:protocol name="dubbo" port="20881"></dubbo:protocol>  
<!--配置rmi的协议-->  
<dubbo:protocol name="rmi" port="1099"></dubbo:protocol>
```

3: 启动时检查

(服务消费者)

```
<dubbo:consumer check="false"></dubbo:consumer>
```

4: 超时调用

(服务消费者)

```
<dubbo:consumer check="false" timeout="10000"></dubbo:consumer>
```

(服务提供者)

```
<dubbo:provider timeout="10000"></dubbo:provider>
```

7. 负载均衡

【目标】

Dubbo配置负载均衡

【路径】

1: 负载均衡介绍

2: 测试负载均衡效果

【讲解】

7.1. 介绍

负载均衡 (Load Balance) : 其实就是将请求分摊到多个操作单元上进行执行，从而共同完成工作任务。

在集群负载均衡时，Dubbo 提供了多种均衡策略（包括随机random、轮询roundrobin、最少活跃调用数leastactive），缺省【默认】为random随机调用。

配置负载均衡策略，既可以在服务提供者一方配置（@Service(loadbalance = "roundrobin")），也可以在服务消费者一方配置（@Reference(loadbalance = "roundrobin")），两者取一

- 如下在服务消费者指定负载均衡策略，可在@Reference添加@Reference(loadbalance = "roundrobin")

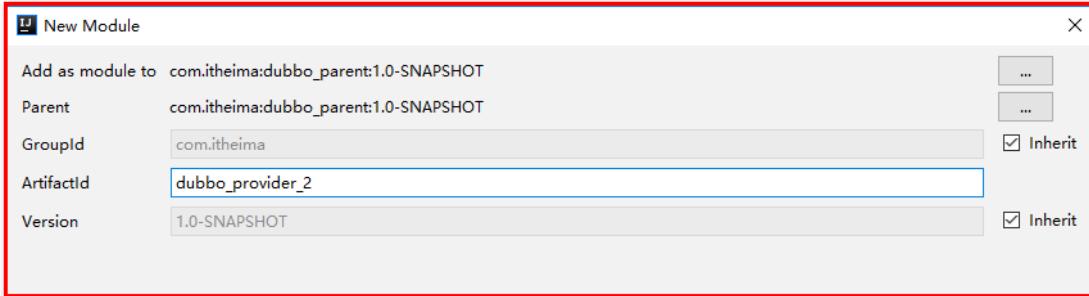
```
@RestController  
@RequestMapping("/user")  
public class UserController {  
  
    @Reference(loadbalance = "roundrobin")  
    private UserService userService;
```

7.2. 测试负载均衡效果

增加一个提供者，提供相同的服务；

正式生产环境中，最终会把服务端部署到多台机器上，故不需要修改任何代码，只需要部署到不同机器即可测试。如果是单机测试，必须通过修改该提供者的dubbo协议端口和web服务端口来进行区分部署。

1. 创建子模块dubbo_provider_2



2. 参考dubbo_provider 项目

复制pom.xml

复制main

复制test

3. 修改端口

(1) dubbo协议: dubbo_provider -- 20880

dubbo_provider_2-- 20881

```

11
12      <!--dubbo_provider 的dubbo协议-->
13      <dubbo:protocol name="dubbo" port="20880"></dubbo:protocol>
14
beans > dubbo:protocol
dubbo provider 2\src\main\resources\spring-dubbo.xml
Application context not configured for this file
11
12      <!--dubbo_provider_2 的dubbo协议-->
13      <dubbo:protocol name="dubbo" port="20881"></dubbo:protocol>
14

```

(2) web服务端口号: dubbo_provider -- 81

dubbo_provider_2-- 82

```

dubbo_provider
<plugins>
  <plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <configuration>
      <path>/</path>
      <port>81</port>
    </configuration>
  </plugin>
</plugins>
project > build > plugins > plugin > configuration > port
dubbo_provider_2
<plugins>
  <plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <configuration>
      <path>/</path>
      <port>82</port>
    </configuration>
  </plugin>
</plugins>
</build>

```

(3) 在service方法中添加输出语句（20880、20881），用来区分不同的应用

```

dubbo_provider\src\main\java\com\dubbo\provider\UserServiceImpl.java
16     UserDao userDao;
17
18     @Override
19     public User findById(Integer id) {
20         System.out.println("dubbo协议端口号: 20880");
21         return userDao.findById(id);
22     }
UserServiceimpl > findById()
dubbo_provider_2\src\main\java\com\dubbo\provider\UserServiceImpl.java
17
18     @Override
19     public User findById(Integer id) {
20         System.out.println("dubbo协议端口号: 20881");
21         return userDao.findById(id);
22     }
23

```

4. 消费者配置负载均衡

```
RestController  
@RequestMapping("/user")  
public class UserController {  
  
    @Reference(loadbalance = "random")  
    UserService userService; // 配置负载均衡  
  
    @RequestMapping("/findById")  
    @ResponseBody  
    public User findById(Integer id) {  
        User user = userService.findById(id);  
        return user;  
    }  
}
```

其中：

```
@Reference(loadbalance = "roundrobin") // 表示轮询  
@Reference(loadbalance = "random") // 表示随机（默认）
```

5. 访问测试

安装父工程到本地仓库

启动两个提供者 (dubbo_provider、dubbo_provider2) : 端口号: 81 , 82

启动消费者，访问: <http://localhost:80/user/findById.do?id=1>

【小结】

1: 负载均衡介绍

2: 测试负载均衡效果

8. 配置中心

【目标】

Dubbo配置中心

【路径】

1: 配置中心环境介绍

2: 实现配置中心

(1) 在Zookeeper中添加数据源所需配置

(2) 在dubbo-common中导入jar包

(3) 修改数据源，读取Zookeeper中数据

3: watch机制

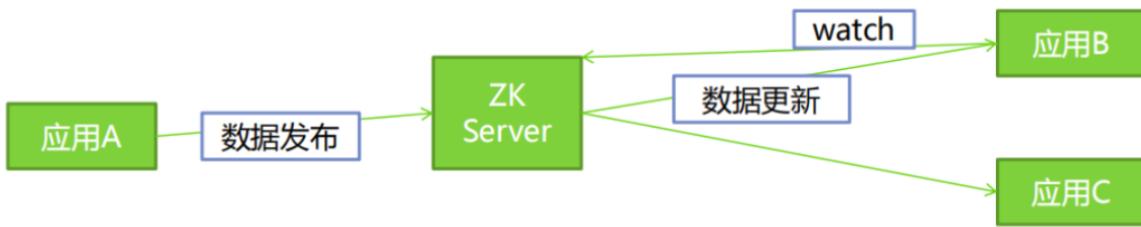
(1) 添加监听

(2) 获取容器对象

【讲解】

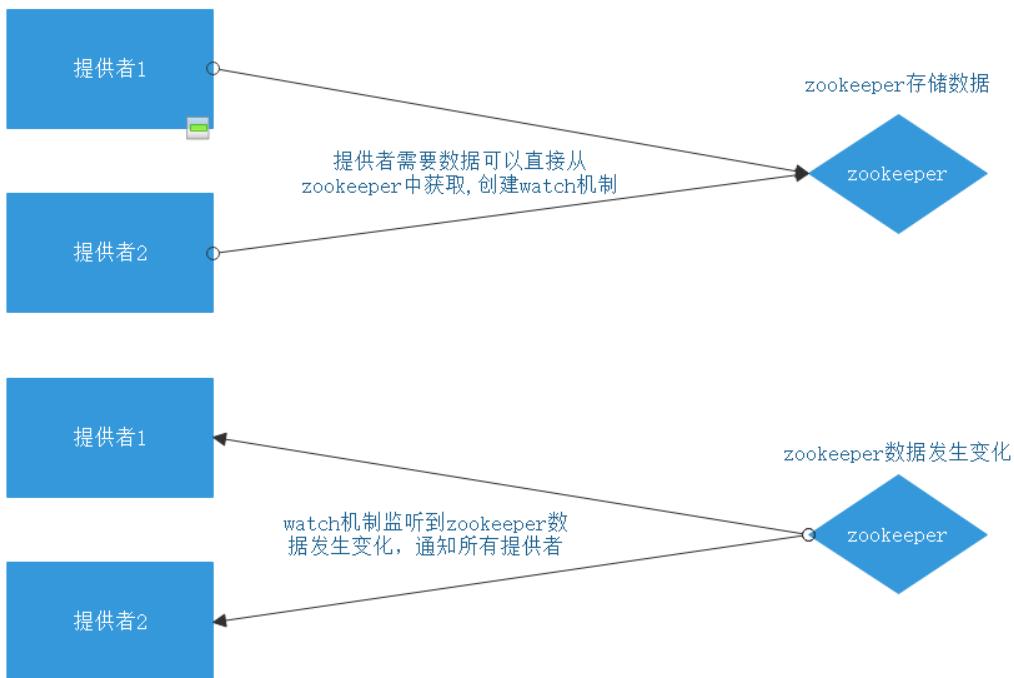
8.1. 环境介绍

数据发布/订阅即所谓的配置中心：发布者将数据发布到ZooKeeper一系列节点上面，订阅者进行数据订阅，当数据有变化时，可以及时得到数据的变化通知，达到**动态及时获取数据**的目的。



现在项目中有两个提供者，配置了相同的数据源，如果此时要修改数据源，必须同时修改两个才可以。

我们可以将数据源中需要的配置信息存储在zookeeper中，如果修改数据源配置，使用zookeeper的watch机制，同时对提供者的数据源信息更新。如下图所示：



8.2. 实现配置中心

【路径】

- 1: 在zookeeper中添加数据源所需配置
- 2: 在dubbo-common中导入jar包
- 3: 修改数据源，读取zookeeper中数据源所需配置数据
 - (1) 在dubbo-common中创建工具类：SettingCenterUtil,继承PropertyPlaceholderConfigurer
 - (2) 编写载入zookeeper中配置文件，传递到Properties属性中
 - (3) 重写processProperties方法
 - (4) 修改spring-dao.xml
- 4: watch机制
 - (1) 添加监听

(2) 获取容器对象，刷新spring容器：SettingCenterUtil实现ApplicationContextAware

8.2.1. 在zookeeper中添加数据源所需配置

```
[zk: localhost:2181(CONNECTED) 14] create /config 'config'  
Created /config  
[zk: localhost:2181(CONNECTED) 15] create /config/jdbc.url 'jdbc:mysql://localhost:3306/test_01'  
Created /config/jdbc.url  
[zk: localhost:2181(CONNECTED) 16] create /config/jdbc.user 'root'  
Created /config/jdbc.user  
[zk: localhost:2181(CONNECTED) 17] create /config/jdbc.password 'root'  
Created /config/jdbc.password  
[zk: localhost:2181(CONNECTED) 18] create /config/jdbc.driver 'com.mysql.jdbc.Driver'  
Created /config/jdbc.driver
```

让我们的程序读取Zookeeper中的配置，而不再从程序的中的jdbc.properties文件中读取。

8.2.2. 在dubbo-common中导入jar包

```
<dependencies>  
    <dependency>  
        <groupId>org.springframework</groupId>  
        <artifactId>spring-webmvc</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>com.alibaba</groupId>  
        <artifactId>dubbo</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>org.apache.zookeeper</groupId>  
        <artifactId>zookeeper</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>org.apache.curator</groupId>  
        <artifactId>curator-framework</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>org.apache.curator</groupId>  
        <artifactId>curator-recipes</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>org.mybatis</groupId>  
        <artifactId>mybatis</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>org.mybatis</groupId>  
        <artifactId>mybatis-spring</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>com.alibaba</groupId>  
        <artifactId>druid</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>mysql</groupId>  
        <artifactId>mysql-connector-java</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>org.springframework</groupId>  
        <artifactId>spring-tx</artifactId>  
    </dependency>  
    <dependency>
```

```

<groupId>org.springframework</groupId>
<artifactId>spring-jdbc</artifactId>
</dependency>
</dependencies>

```

8.2.3. 修改数据源，读取zookeeper中数据

8.2.3.1. 在dubbo-common中创建工具类：SettingCenterUtil,继承PropertyPlaceholderConfigurer

The screenshot shows the Eclipse IDE interface. On the left is the package explorer view with the project structure:

```

dubbo_common
  src
    main
      java
        com
          itheima
            pojo
              User
            utils
              SettingCenterUtil
            resources
  test
  target
  dubbo_common.iml
  pom.xml
  dubbo_consumer
  dubbo_interface
  dubbo_provider
  src
    main

```

The right side shows the code editor with the file `SettingCenterUtil.java`. The code imports `BeansException`, `ConfigurableListableBeanFactory`, `PropertyPlaceholderConfigurer`, and `Properties`. It includes a Javadoc block with annotations `@author`, `@Company`, and `@Version`. The class `SettingCenterUtil` extends `PropertyPlaceholderConfigurer`.

```

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.config.PropertyPlaceholderConfigurer;
import java.util.Properties;

/**
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class SettingCenterUtil extends PropertyPlaceholderConfigurer {

```

8.2.3.2. 编写载入zookeeper中配置文件，传递到Properties属性中

```

/**
 * 载入zookeeper数据
 * @param props
 */
private void loadzk(Properties props){
    //创建重试策略
    RetryPolicy retryPolicy = new ExponentialBackoffRetry(3000,1);
    //创建客户端
    CuratorFramework client =
    CuratorFrameworkFactory.newClient("127.0.0.1:2181", 3000, 3000, retryPolicy);
    client.start();
    try {
        byte[] urlBytes = client.getData().forPath("/config/jdbc.url");
        props.setProperty("jdbc.url",new String(urlBytes));
        byte[] userBytes = client.getData().forPath("/config/jdbc.user");
        props.setProperty("jdbc.user",new String(userBytes));
        byte[] pwdBytes = client.getData().forPath("/config/jdbc.password");
        props.setProperty("jdbc.password",new String(pwdBytes));
        byte[] driverBytes = client.getData().forPath("/config/jdbc.driver");
        props.setProperty("jdbc.driver",new String(driverBytes));
    } catch (Exception e) {
        e.printStackTrace();
    }
    client.close();
}

```

8.2.3.3. 重写processProperties方法

```

/**
 * 处理properties内容,相当于此标签
 * <context:property-placeholder location="classpath:jdbc.properties">
</context:property-placeholder>
 * @param beanFactoryToProcess

```

```

    * @param props
    * @throws BeansException
    */
@Override
protected void processProperties(ConfigurableListableBeanFactory beanFactoryToProcess, Properties props) throws BeansException {
    // 载入zookeeper配置信息，即从Zookeeper中获取数据源的连接信息
    loadZk(props);
    super.processProperties(beanFactoryToProcess, props);
}

```

8.2.3.4. 修改spring-dao.xml

注释掉：

```

<context:property-placeholder location="classpath:jdbc.properties">
</context:property-placeholder>

```

添加：

```

<bean id="settingCenterUtil" class="com.itheima.utils.SettingCenterUtil"></bean>

```

spring-dao.xml中配置

```

<!--加载属性文件-->
<!--<context:property-placeholder location="classpath:jdbc.properties"></context:property-placeholder>

<!--创建配置中心对象-->
<bean class="com.itheima.utils.SettingCenterUtil"></bean>

```

8.2.4. watch机制

8.2.4.1. 添加监听

```

/**
 * 添加watch机制
 * @param props
 */
private void addwatch(Properties props) {
    RetryPolicy retryPolicy = new ExponentialBackoffRetry(3000,1);
    CuratorFramework client =
CuratorFrameworkFactory.newClient("127.0.0.1:2181", 3000, 3000, retryPolicy);
    client.start();
    //创建缓存机制，监听 /config 路径
    TreeCache treeCache = new TreeCache(client,"/config");
    //启动缓存机制
    try {
        treeCache.start();
    } catch (Exception e) {
        e.printStackTrace();
    }
    //添加监听
    treeCache.getListenable().addListener(new TreeCacheListener() {
        /**
         *
         * @param client zookeeper客户端对象
        */
    });
}

```

```

        * @param event zookeeper改变触发的事件
        * @throws Exception
        */
    @Override
    public void childEvent(CuratorFramework client, TreeCacheEvent event)
throws Exception {
    //如果是修改事件则，更新数据
    if(event.getType() == TreeCacheEvent.Type.NODE_UPDATED){
        //如果修改的是jdbc.url路径，修改jdbc.url的配置
        if(event.getData().getPath().equals("/config/jdbc.url")){
            props.setProperty("jdbc.url",new
String(event.getData().getData()));
        }else
        if(event.getData().getPath().equals("/config/jdbc.driver")){
            props.setProperty("jdbc.driver",new
String(event.getData().getData()));
        }else if(event.getData().getPath().equals("/config/jdbc.user")){
            props.setProperty("jdbc.user",new
String(event.getData().getData()));
        }else
        if(event.getData().getPath().equals("/config/jdbc.password")){
            props.setProperty("jdbc.password",new
String(event.getData().getData()));
        }
    }
}
);
}
}

```

注意：

- 1: 不要关闭client，否则无法进行监控
- 2: 修改完成后必须刷新spring容器的对象

8.2.4.2. 获取容器对象，刷新spring容器

1: 实现ApplicationContextAware接口，重写setApplicationContext方法，获取applicationContext对象。

```

public class SettingCenterUtil extends PropertyPlaceholderConfigurer implements ApplicationContextAware {

    XmlWebApplicationContext applicationContext;
    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = (XmlWebApplicationContext) applicationContext;
    }
}

```

2: XmlWebApplicationContext 容器对象，提供了refresh方法，可以刷新容器中的对象，故强制转换。

```

//修改完成后必须刷新spring容器的对象
applicationContext.refresh();

```

3: 在processProperties的方法中，添加addWatch(props);

```
protected void processProperties(ConfigurableListableBeanFactory beanFactoryToProcess, Properties props) throws BeansException {
    loadZk(props);
    addwatch(props);
    super.processProperties(beanFactoryToProcess, props);
}
```

4: 修改Zookeeper的配置

```
[zk: localhost:2181(CONNECTED) 10] ls /config
[jdbc.password, jdbc.user, jdbc.url, jdbc.driver]
[zk: localhost:2181(CONNECTED) 11] set /config/jdbc.url "jdbc:mysql:///test"
cZxid = 0x13d
ctime = Fri Jan 17 10:40:59 CST 2020
mZxid = 0x164
mtime = Fri Jan 17 11:44:18 CST 2020
pZxid = 0x13d
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 18
numChildren = 0
[zk: localhost:2181(CONNECTED) 12] get /config/jdbc.url
[jdbc:mysql:///test]
cZxid = 0x13d
ctime = Fri Jan 17 10:40:59 CST 2020
mZxid = 0x164
mtime = Fri Jan 17 11:44:18 CST 2020
pZxid = 0x13d
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 18
numChildren = 0
[zk: localhost:2181(CONNECTED) 13]
```

【小结】

1: 配置中心环境介绍

2: 实现配置中心

(1) 在Zookeeper中添加数据源所需配置

(2) 在dubbo-common中导入jar包

(3) 修改数据源，读取Zookeeper中数据

3: watch机制

(1) 添加监听

(2) 获取容器对象