# 大数据基础

## 大数据课程介绍

什么是大数据？

海量数据的处理

大数据用在哪？

一句话：用在需要对海量数据进行处理的任何场合

举个例子：

1、搜索引擎——需要对海量文档建索引库

2、各种电商——需要对用户的点击日志（海量）进行各种分析

3、电信运营商——需要对用户的上网行为（海量）进行分析

4、广告系统——需要对用户进行画像（海量）以便于做精准投放

……

大数据学什么？

1、学分布式系统的思想

2、学框架

（hadoop<基础，离线处理> + storm <实时处理的王者> + spark<加分项，技术趋势，擅长内存计算，迭代运算（机器学习）>）

以什么心态学习大数据？

1、平和

2、要有长远目光

以什么方法学习大数据？

1、要有主次之分(重点内容，深入掌握！)

2、一定要多动手实践（说三遍！！！）

3、学习时主要关注点：

a、最基本——知道这个框架是用来干什么的

b、应用——掌握框架的用法（编程规范、api、框架之间的衔接整合）

c、核心框架要掌握一定深度的原理

d、数据分析系统整体性的把握

### 四个阶段

**1、大数据基础部分**

a、linux的操作和shell编程

b、zookeeper 分布式系统的协调服务

c、redis内存数据库

d、java的并发包编程；反射；socket编程；NIO；netty；RPC；JVM的内存管理

**2、离线（批处理）计算系统**

a、hdfs分布式文件系统

b、mapreduce分布式运算框架

c、hive基于hadoop的数据仓库工具

d、hbase基于hadoop的nosql数据库

e、离线计算系统的辅助工具—— flume、sqoop、azkaban

f、离线计算的项目案例

**3、实时流式计算系统**

a、storm

b、kafka、flume、redis

c、实时计算的项目案例—— 京东实时交易风控、实时推荐系统

**4、内存计算体系**

a、scala编程语言——基于JVM的语言

b、spark

c、spark-core spar-sql spark-streaming（实时流式处理） spark-mllib（机器学习算法库）

d、ELK

e、项目—— 魔兽世界游戏日志分析

## Linux操作基础

### Linux系统安装

#### Linux简介

1、Linux是一套**免费使用**和**自由传播**的类Unix**操作系统（主要用在服务器上）**，是一个基于POSIX和UNIX的多用户、多任务、支持多线程和多CPU的操作系统。它能运行主要的UNIX工具软件、应用程序和网络协议。它支持32位和64位硬件。Linux继承了Unix以网络为核心的设计思想，是一个性能稳定的多用户网络操作系统。

2、Linux操作系统诞生于1991 年10 月5 日（这是第一次正式向外公布时间）。Linux存在着**许多不同的Linux发行版本，但它们都使用了Linux内核**。Linux可安装在各种计算机硬件设备中，比如手机、平板电脑、路由器、视频游戏控制台、台式计算机、大型机和超级计算机。

补充：linux内核必须加上一个“界面”软件，才能让用户去使用

“界面”分两类：

a、命令行界面（有很多种，最流行的一种是 bash shell）

b、图形界面（也有很多种，目前最流行的有两种：gnome kde）

3、常见发行版

红帽企业版：RedHat Enterprise Linux（RHEL），红帽桌面版Fedora

Centos社区发行版：源自RedHat企业版，相比少了一些收费的应用软件

Ubuntu：桌面版、服务器版、移动端

麒麟kylin：国防科技大学发行

红旗RedFlag：中科院发行

#### VMWare虚拟机

Vmware公司出品的一套虚拟化软件，可以在物理机上用软件的方式模拟出虚拟的计算机系统，而虚拟出的计算机系统可以像一台真实的机器一样，安装操作系统，运行软件等

类似的产品还有Oracle出品的免费版虚拟机软件VirtualBox

#### Linux系统安装/图形化界面使用

1.linux文件目录

bin:二进制文件

sbin:系统的二进制文件

dev:系统的设备

etc:配置文件

home:普通用户的主目录

usr:linux共享资源,用户装的软件一般装在这里(unix system resource)

tmp:临时目录

2.图形界面到命令行界面: # init 3

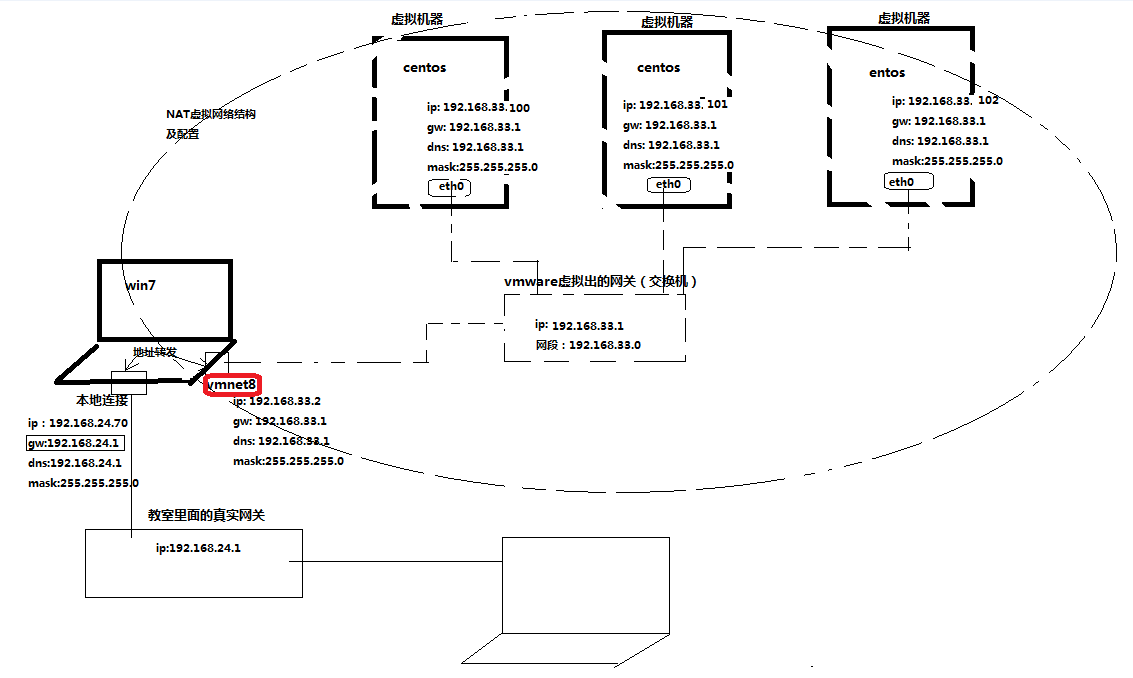
#### 网络基础及局域网配置

1.简单局域网结构

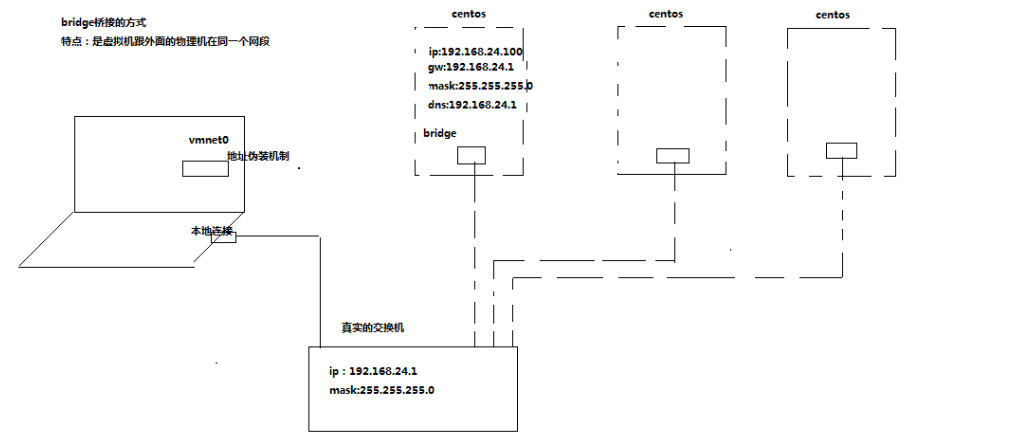
setup命令可以有提示界面配置网络连接

#### VMware的三种网络模式

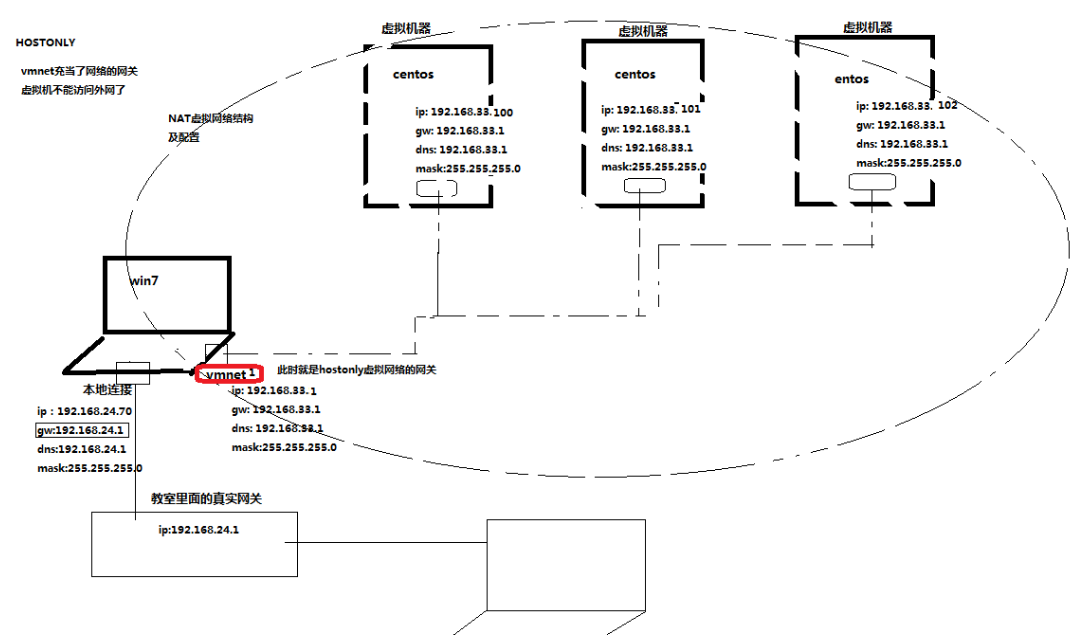
NAT模式:



桥接模式:



HOSTONLY模式



3.ssh远程连接Linux服务器

### Linux常用命令

#### 日常操作命令

1、查看当前所在的工作目录的全路径 pwd

[test@hdp-node-01 ~]$ pwd

*/home/test*

2、查看当前系统的时间 date

[test@hdp-node-01 ~]$ date +%Y-%m-%d

*2016-05-18*

[test@hdp-node-01 ~]$ date +%Y-%m-%d --date="-1 day" #加减也可以 month | year 求昨天的日期

*2016-05-17*

*设置时间*

*date -s "2016-05-23 01:01"* ## 修改时间

*修改时间后，需要写入硬件bios才能在重启之后依然生效*

*hwclock -w*

3、查看有谁在线（哪些人登陆到了服务器）

who 查看当前在线

[test@hdp-node-01 ~]$ who

root tty1 2016-05-18 11:46

hadoop tty3 2016-05-18 11:46

hadoop pts/0 2016-05-18 10:57 (192.168.33.1)

last 查看最近的登陆历史记录

[test@hdp-node-01 ~]$ last -3

hadoop tty3 Wed May 18 11:46 still logged in

root tty1 Wed May 18 11:46 still logged in

hadoop pts/0 192.168.33.1 Wed May 18 10:57 still logged in

4、关机/重启

## 关机（必须用root用户）

shutdown -h now ## 立刻关机

shutdown -h +10 ## 10分钟以后关机

shutdown -h 12:00:00 ## 12点整的时候关机

halt # 等于立刻关机

## 重启

shutdown -r now

reboot # 等于立刻重启

5、清屏

clear ## 或者用快捷键 ctrl + l

6、退出当前进程

ctrl+c 有些程序也可以用q键退出

7、**挂起当前进程**

ctrl+z ## 进程会挂起到后台

bg jobid ## 让进程在后台继续执行

fg jobid ## 让进程回到前台

8、echo

相当于java中System.out.println(userName)

|  |
| --- |
| [root@localhost bbb]# a="婷婷是我的梦中情人"  [root@localhost bbb]#  [root@localhost bbb]# echo a  a  [root@localhost bbb]# echo $a  婷婷是我的梦中情人 |

#### 目录操作

1 查看目录信息

ls / ## 查看根目录下的子节点（文件夹和文件）信息

ls -al ## -a是显示隐藏文件 -l是以更详细的列表形式显示

ls -l 有一个别名： ll 可以直接使用ll <是两个L>

2 切换工作目录

cd /home/hadoop ## 切换到用户主目录

cd ~ ## 切换到用户主目录

cd 什么路径都不带，则回到用户的主目录

cd - ## 回退到上次所在的目录

3 创建文件夹

mkdir aaa ## 这是相对路径的写法

mkdir /data ## 这是绝对路径的写法

mkdir -p aaa/bbb/ccc ## 级联创建目录

4 删除文件夹

rmdir aaa ## 可以删除空目录

rm -r aaa ## 可以把aaa整个文件夹及其中的所有子节点全部删除

rm -rf aaa ## 强制删除aaa

5 修改文件夹名称

mv aaa angelababy

mv本质上是移动

mv install.log aaa/ 将当前目录下的install.log 移动到aaa文件夹中去

rename 可以用来批量更改文件名

|  |
| --- |
| [root@localhost bbb]# ll  total 0  -rw-r--r--. 1 root root 0 May 22 15:58 1.txt  -rw-r--r--. 1 root root 0 May 22 15:58 2.txt  -rw-r--r--. 1 root root 0 May 22 15:58 3.txt  [root@localhost bbb]# rename .txt .txt.bak \*  [root@localhost bbb]# ll  total 0  -rw-r--r--. 1 root root 0 May 22 15:58 1.txt.bak  -rw-r--r--. 1 root root 0 May 22 15:58 2.txt.bak  -rw-r--r--. 1 root root 0 May 22 15:58 3.txt.bak |

#### 增删改查文件,vi操作,解压缩

**1 创建文件**

touch somefile.1

## 创建一个空文件

echo "i miss you,my baby" > somefile.2

## 利用**重定向**“>”的功能，将一条指令的输出结果写入到一个文件中，会覆盖原文件内容，如果指定的文件不存在，则会创建出来

echo "huangxiaoming ,gun dan" >> somefile.2

## 将一条指令的输出结果追加到一个文件中，不会覆盖原文件内容

**2 vi文本编辑器**

1、最基本用法

vi somefile.4

有三种模式:一般模式;编辑模式;命令模式

1/ 首先会进入“**一般模式**”，此模式只接受各种命令快捷键，不能编辑文件内容

2/ 按i键，就会从一般模式进入**编辑模式**，此模式下，敲入的都是文件内容

3/ 编辑完成之后，按**Esc键退出编辑模式**，回到一般模式；

4/ 再按**：**，进入“**底行命令模式**”，输入**wq**命令，回车即可保存退出

2、常用快捷键

一些有用的快捷键（在一般模式下使用）：

a 在光标后一位开始插入

A 在该行的最后插入

I 在该行的最前面插入

gg 直接跳到文件的首行

G 直接跳到文件的末行

dd 删除一行

3dd 删除3行

yy 复制一行

3yy 复制3行

p 粘贴

u undo

~~ctrl + r redo~~

v 进入字符选择模式，选择完成后，按y复制，按p粘贴

ctrl+v 进入块选择模式，选择完成后，按y复制，按p粘贴

shift+v 进入行选择模式，选择完成后，按y复制，按p粘贴

3、查找并替换

（在底行命令模式中输入）

1显示行号

:set nu

2 隐藏行号

:set nonu

3 查找关键字

:/you ## 效果：查找文件中出现的you，并定位到第一个找到的地方，按n可以定位到下一个匹配位置（按N定位到上一个）

4 替换操作

:s/sad/bbb 查找光标所在行的第一个sad，替换为bbb

:%s/sad/bbb 查找文件中所有sad，替换为bbb

~~:1,$s/hadoop/root/g 将第一行到最后一行的hadoop替换为root~~

~~:1,$s/hadoop/root/c 将第一行到最后一行的hadoop替换为root(有提示)~~

3 拷贝/删除/移动

cp somefile.1 /home/hadoop/

rm /home/hadoop/somefile.1

rm -rf /home/hadoop/somefile.1

mv /home/hadoop/somefile.1 ../

4 查看文件内容

cat somefile 一次性将文件内容全部输出（控制台）

分页查看文件的命令：

more somefile 可以翻页查看, 下翻一页(空格) 上翻一页（b） 退出（q）

less somefile 可以翻页查看,下翻一页(空格) 上翻一页（b），上翻一行(↑) 下翻一行（↓）

可以搜索关键字（/keyword）

跳到文件末尾： G

跳到文件首行： gg

退出less ： q

tail -10 install.log 查看文件尾部的10行

tail +10 install.log 查看文件 10-->末行

**tail -f** install.log 小f跟踪文件的唯一inode号，就算文件改名后，还是跟踪原来这个inode表示的文件

tail -F install.log 大F按照文件名来跟踪

head -10 install.log 查看文件头部的10行

5 打包压缩

1、gzip压缩

gzip a.txt

2、解压

~~gunzip a.txt.gz~~

**gzip -d a.txt.gz**

3、bzip2压缩

bzip2 a

4、解压

bunzip2 a.bz2

bzip2 -d a.bz2

5、**打包**：将指定文件或文件夹

tar -cvf bak.tar ./aaa

将/etc/password追加文件到bak.tar中(r)

tar -rvf bak.tar /etc/password

6、解压

tar -xvf bak.tar

7、**打包并压缩**

**tar -czvf a.tar.gz aaa/**

**8、解包并解压缩(重要的事情说三遍!!!)**

**tar -xzvf a.tar.gz**

解压到/usr/下

tar -xzvf a.tar.gz **-C /usr**

9、查看**压缩**包内容

tar -tzvf a.tar.gz

zip/unzip

10、打包并压缩成bz2

tar -jcvf a.tar.bz2

11、解压bz2

tar -jxvf a.tar.bz2

#### 查找命令

**1 常用查找命令的使用**

1、查找可执行的命令所在的路径：

which ls

2、查找可执行的命令和帮助的位置：

whereis ls

3、从某个文件夹开始查找文件

**find / -name "hadooop\*"**

find / -name "hadooop\*" -ls

5、查找并删除

find / -name "hadooop\*" -ok rm {} \;

find / -name "hadooop\*" -exec rm {} \;

6、查找用户为hadoop的文件

find /usr -user hadoop -ls

7、查找用户为hadoop的文件夹

find /home -user hadoop -type d -ls

8、查找权限为777的文件

find / -perm -777 -type d -ls

\*\*、显示历史命令

history

**2 grep命令**

1/ 基本使用

查询包含hadoop的行

grep hadoop /etc/password

grep aaa ./\*.txt

2/ cut截取以:分割保留第七段

grep hadoop /etc/passwd | cut -d: -f7

3/ 查询不包含hadoop的行

grep -v hadoop /etc/passwd

4/ 正则表达包含hadoop

grep 'hadoop' /etc/passwd

5/ 正则表达(点代表任意一个字符)

grep 'h.\*p' /etc/passwd

6/ 正则表达以hadoop开头

grep '^hadoop' /etc/passwd

7/ 正则表达以hadoop结尾

grep 'hadoop$' /etc/passwd

~~正则表达式的简单规则：~~

. : 任意一个字符

a\* : 任意多个a(零个或多个a)

a? : 零个或一个a

a+ : 一个或多个a

.\* : 任意多个任意字符

\. : 转义.

o\{2\} : o重复两次

查找不是以#开头的行

grep -v '^#' a.txt | grep -v '^$'

以h或r开头的

grep '^[hr]' /etc/passwd

不是以h和r开头的

grep '^[^hr]' /etc/passwd

不是以h到r开头的

grep '^[^h-r]' /etc/passwd

#### 文件权限操作

**1 linux文件权限的描述格式解读**

drwxr-xr-x （也可以用二进制表示 111 101 101 --> 755）

d：标识节点类型（d：文件夹 -：文件 l:链接）

r:可读 w:可写 x:可执行

第一组rwx： ## 表示这个文件的**拥有者**对它的权限：可读可写可执行

第二组r-x： ## 表示这个文件的**所属组用户**对它的权限：可读，不可写，可执行

第三组r-x： ## 表示这个文件的**其他用户**（相对于上面两类用户）对它的权限：可读，不可写，可执行

**2 修改文件权限(普通用户也可以执行)**

chmod **u**+x haha.dat ## 表示将haha.dat对**所属用户**的权限增加x

chmod **g-**rw haha.dat ## 表示将haha.dat对**所属组**的rw权限取消

chmod **o-**rw haha.dat ## 表示将haha.dat对**其他人**的rw权限取消

chmod **a-x** haha.dat ## 表示将haha.dat对**所用户**取消x权限

也可以用数字的方式来修改权限

chmod 664 haha.dat

就会修改成 rw-rw-r--

如果要将一个文件夹的所有内容权限统一修改，则可以-R参数

chmod -R 770 aaa/

**3 修改文件所有权(只有root才能执行)**

<只有root权限能执行>

chown angela aaa ## 改变所属用户

chown :angela aaa ## 改变所属组

chown angela:angela aaa/ ## 同时修改所属用户和所属组

*补充：*

*r: 对文件来说，是可读取内容； 对文件夹来说，是可以ls*

*w: 对文件来说，是可修改文件的内容；对文件夹来说，是可以在其中创建或者删除子节点*

*x： 对文件来说，是能否运行这个文件；对文件夹来说，是能否cd进入这个目录*

#### 基本用户管理

需要我们掌握的：

添加一个用户：

1、 useradd spark

2、 passwd spark 根据提示设置密码；

即可

删除一个用户：

userdel -r spark 加一个-r就表示把用户及用户的主目录都删除

**1 添加用户**

添加一个tom用户，设置它属于users组，并添加注释信息

分步完成：**useradd tom**

usermod -g users tom

usermod -c "hr tom" tom

一步完成：useradd -g users -c "hr tom" tom

设置tom用户的密码

passwd tom

**2 修改用户**

修改tom用户的登陆名为tomcat

usermod -l tomcat tom

将tomcat添加到sys和root组中

usermod -G sys,root tomcat

查看tomcat的组信息

groups tomcat

**3 用户组操作**

添加一个叫america的组

groupadd america

将jerry添加到america组中

usermod -g america jerry

将tomcat用户从root组和sys组删除

gpasswd -d tomcat root

gpasswd -d tomcat sys

将america组名修改为am

groupmod -n am america

**4 为用户配置sudo权限**

用root编辑 **vi /etc/sudoers**

在文件的如下位置，为hadoop添加一行即可

root ALL=(ALL) ALL

hadoop ALL=(ALL) ALL

然后，hadoop用户就可以用sudo来执行系统级别的指令

[hadoop@shizhan ~]$ sudo useradd huangxiaoming

#### 系统管理(挂载关盘,服务启动关闭)

**1 挂载外部存储设备**

可以挂载光盘、硬盘、磁带、光盘镜像文件等

1/ 挂载光驱

mkdir /mnt/cdrom 创建一个目录，用来挂载

mount -t iso9660 -o ro **/dev/cdrom** /mnt/cdrom/ 将设备/dev/cdrom挂载到 挂载点 ： /mnt/cdrom中

2/ 挂载光盘镜像文件（.iso文件）

mount -t iso9660 -o loop **/home/hadoop/Centos-6.7.DVD.iso** /mnt/centos

*注：挂载的资源在重启后即失效，需要重新挂载。要想自动挂载，可以将挂载信息设置到/etc/fstab配置文件中，如下：*

|  |
| --- |
| /dev/cdrom /mnt/cdrom iso9660 defaults 0 0  /root/CentOS-6.7-x86\_64-bin-DVD1.iso /mnt/centos iso9660 defaults,ro,loop 0 0 |

3/ 卸载 umount

umount /mnt/cdrom

\*\* 存储空间查看

df -h

**2 统计文件或文件夹的大小**

du -sh /mnt/cdrom/packages ## 统计指定路径下的所有子目录和文件的大小(disk usage summary)

df -h 查看磁盘的剩余空间(disk free)

**3 系统服务管理**

service --status-all # 查看系统所有的后台服务进程

(sudo service --status-all | grep ssh #筛选出想要的内容)

service sshd status # 查看指定的后台服务进程的状态

service sshd stop

service sshd start

service sshd restart

配置后台服务进程的开机自启

chkconfig httpd on ## 让httpd服务开机自启

chkconfig httpd off ## 让httpd服务开机不要自启

|  |
| --- |
| *[root@localhost ~]# chkconfig httpd off*  *[root@localhost ~]# chkconfig --list | grep httpd = chkconfig httpd --list*  *httpd 0:off 1:off 2:off 3:off 4:off 5:off 6:off*  *[root@localhost ~]# chkconfig --level 35 httpd on #指定级别的启动*  *[root@localhost ~]# chkconfig --list | grep httpd*  *httpd 0:off 1:off 2:off 3:on 4:off 5:on 6:off* |

**4 系统启动级别管理**

vi /etc/inittab

# Default runlevel. The runlevels used are:

# 0 - halt (Do NOT set initdefault to this)

# 1 - Single user mode

# 2 - Multiuser, without NFS (The same as 3, if you do not have networking)

# 3 - Full multiuser mode ## 没有图形界面的全功能的多用户的启动级别

# 4 - unused

# 5 - X11 ## 有图形界面的启动级别

# 6 - reboot (Do NOT set initdefault to this)

#

id:3:initdefault: ## 配置默认启动级别

## 通常将默认启动级别设置为：3

**5 进程管理**

top

free

ps -ef | grep ssh

kill -9 2358 ## 将指定进程号的进程杀死

注意：grep搜索关键词的时候会把自己也搜索出来，对比以下两种写法

|  |
| --- |
| *[root@localhost ~]# ps -ef | grep sixunhuan*  *root 2857 2465 30 02:41 pts/0 00:00:07 sh sixunhuan.sh*  *root 2874 2858 0 02:42 pts/1 00:00:00 grep sixunhuan*  *[root@localhost ~]# ps -ef | grep sixunhuan | grep -v grep*  *root 2857 2465 34 02:41 pts/0 00:00:25 sh sixunhuan.sh*  *[root@localhost ~]# kill -9 2857* |

#### SSH免密钥登录

**1 SSH工作机制**

**1、相关概念**

SSH 为 Secure Shell（安全外壳协议） 的缩写。

很多ftp、pop和telnet在本质上都是不安全的，因为它们在网络上用明文传送口令和数据，别有用心的人非常容易就可以截获这些口令和数据。而SSH就是专为远程登录会话和其他网络服务提供安全性的协议。

SSH的具体实现是由**客户端和服务端**的软件组成的

服务端是一个守护进程(sshd)，他在后台运行并响应来自客户端的连接请求。

客户端包含ssh程序以及像scp（远程拷贝）、slogin（远程登陆）、sftp（安全文件传输）等其他的应用程序。

**2、认证机制**

从客户端来看，SSH提供两种级别的安全验证。

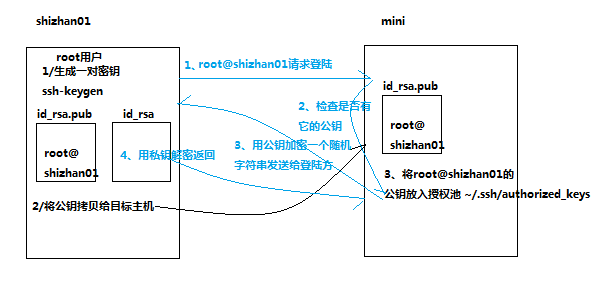
* 第一种方式（基于口令的安全验证）

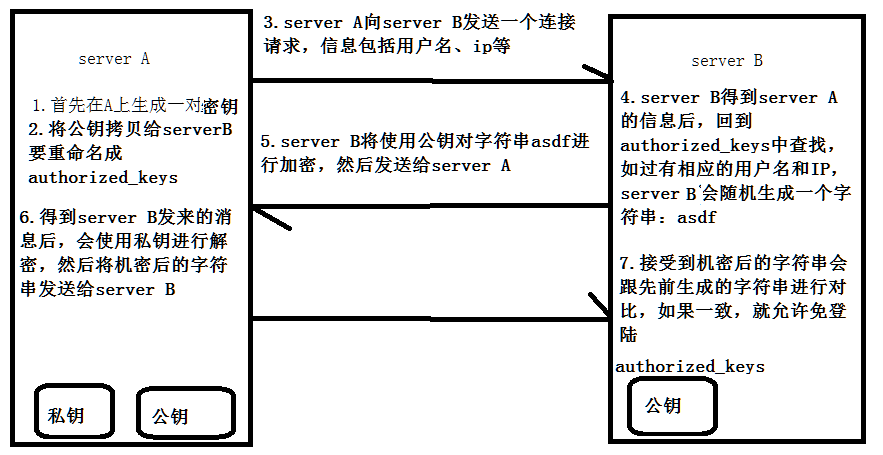
只要你知道自己帐号和口令，就可以登录到远程主机。

* 第二种方式（基于密钥的安全验证）

需要依靠密匙，也就是你必须为自己创建一对密匙，并把公用密匙放在需要访问的服务器上。如果你要连接到SSH服务器上，客户端软件就会向服务器发出请求，请求用你的密匙进行安全验证。服务器收到请求之后，先在该服务器上你的主目录下寻找你的公用密匙，然后把它和你发送过来的公用密匙进行比较。如果两个密匙一致，服务器就用公用密匙加密“质询”（challenge）并把它发送给客户端软件。客户端软件收到“质询”之后就可以用你的私人密匙解密再把它发送给服务器。

**2 密钥登陆方式配置**





假如 A 要登陆 B

在A上操作：

1/ 首先生成密钥对

ssh-keygen (提示时，直接回车即可)

ssh-keygen -t rsa （四个回车） # -t rsa是非对称加密

2/ 再将A自己的公钥拷贝并追加到B的授权列表文件authorized\_keys中

ssh-copy-id B

#### 网络管理

**1 主机名配置**

1/ 查看主机名

hostname

2/ 修改主机名(重启后无效)

hostname hadoop

3/ 修改主机名(重启后永久生效)

**vi /ect/sysconfig/network**

**2 IP地址配置**

修改IP地址

1/ 方式一：setup

用root输入setup命令，进入交互式修改界面

2/ 方式二：修改配置文件

(重启后永久生效)

vi /etc/sysconfig/network-scripts/ifcfg-eth0

3/ 方式三：ifconfig命令

(重启后无效)

ifconfig eth0 192.168.12.22

**3 域名映射**

/etc/hosts文件 用于在通过主机名进行访问时做ip地址解析之用

所以，你想访问一个什么样的主机名，就需要把这个主机名和它对应的ip地址配置在/etc/hosts文件中

|  |
| --- |
| [root@angelababy01 ~]# vi /etc/hosts  127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4  ::1 localhost localhost.localdomain localhost6 localhost6.localdomain6  192.168.33.11 angelababy01  192.168.33.12 angelababy02  192.168.33.13 angelababy03 |

**4 网络服务管理**

1/ 后台服务管理

service network status 查看指定服务的状态

service network stop 停止指定服务

service network start 启动指定服务

service network restart 重启指定服务

service --status-all 查看系统中所有的后台服务

2/ 设置后台服务的自启配置

chkconfig 查看所有服务器自启配置

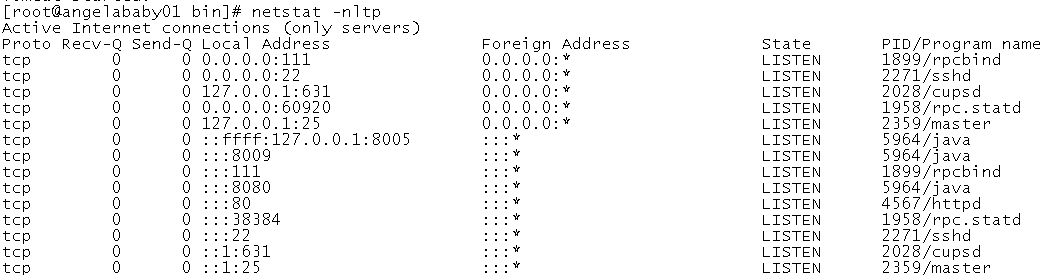
chkconfig iptables off 关掉指定服务的自动启动

chkconfig iptables on 开启指定服务的自动启动

**5 系统中网络进程的端口监听情况**

netstat -nltp

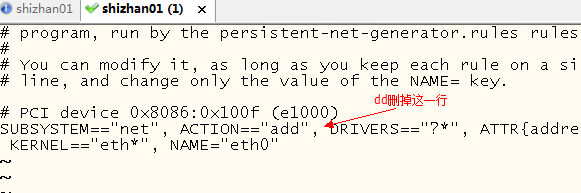
ps -ef|grep 3306



\*\* 补充：Linux的网卡坏了怎么办

1、先删掉旧网卡

vi /etc/udev/rules.d/70-persistent-net.rules



2、halt linux系统

3、通过vmware卸载原来的网卡，再新增一块网卡

4、70-persistent-net.rules 里面会自动生成那块新网卡的硬件信息

5、编辑网卡的配置文件/etc/sysconfig/network-script/ifcfg-eth0

删掉其中的UUID HWADDR

并且修改IP配置等，如下：

|  |
| --- |
| DEVICE="eth0"  BOOTPROTO=none  ONBOOT="yes"  TYPE="Ethernet"  IPADDR=192.168.33.20  PREFIX=24  GATEWAY=192.168.33.1  DNS1=192.168.33.1  NAME="System eth0" |

6、service network restart 重启网络服务即可

### linux常用软件安装

#### Linux系统软件安装方式

Linux上的软件安装有以下几种常见方式：

1. 二进制发布包

软件已经针对具体平台编译打包发布，只要解压，修改配置即可

1. RPM发布包(redhat package managment )

软件已经按照redhat的包管理工具规范RPM进行打包发布，需要获取到相应的软件RPM发布包，然后用RPM命令进行安装

1. Yum在线安装

软件已经以RPM规范打包，但发布在了网络上的一些服务器上，可用yum在线安装服务器上存在的rpm软件，并且会自动解决软件安装过程中的库依赖问题

（注：类似于maven）

1. 源码编译安装

软件以源码工程的形式发布，需要获取到源码工程后用相应开发工具进行编译打包部署

#### JAVA软件安装——JDK安装

1、上传jdk压缩包

通过sftp工具上传即可

2、解压jdk压缩包

tar -zxvf jdk-7u67-linux-x64.gz -C /usr/local/

3、修改环境变量PATH

vi /etc/profile

在文件最后加两行：

export JAVA\_HOME=/usr/local/jdk1.7.0\_67

export PATH=$PATH:$JAVA\_HOME/bin

4、让环境变量生效

source /etc/profile

即可

#### JAVA软件安装——Tomcat安装

tar -zxvf /soft/apache-tomcat-7.0.47.tar.gz -C /usr/local/

cd /usr/local/apache-tomcat-7.0.47/bin/

./startup.sh

#### Desktop版MySql数据库安装

**desktop版的centos中安装mysql**

**没有那么麻烦，基本上都不缺依赖，过程如下：针对MySQL5.6版本的**

1、先装server

rpm -ivh MySQL-server-5.6.26-1.linux\_glibc2.5.x86\_64.rpm

发现了包冲突，执行删除包的命令解决：

rpm -e mysql-libs-5.1.73-5.el6\_6.x86\_64 --nodeps

然后继续重新安装server

rpm -ivh MySQL-server-5.6.26-1.linux\_glibc2.5.x86\_64.rpm

一定要确保server安装成功！！！

2、再装client

rpm -ivh MySQL-client-5.6.26-1.linux\_glibc2.5.x86\_64.rpm

3、启动server

service mysql start

检查启动是否成功：

service mysql status 或者 查看端口：netstat -nltp | grep 3306 看是否有进程在监听3306

4、用客户端连接到mysql-server上进行root密码的修改

注：在server安装时已经生成了一个随机的root密码 ，查看： cat /root/.mysql\_secret

然后，运行一个脚本来交互式地修改root密码即可：/usr/bin/mysql\_secure\_installation

#### mini版安装mysql服务

可以用yum方式来安装，此处演示在本地使用rpm包安装

*[root@mini ~]# rpm -ivh MySQL-server-5.1.73-1.glibc23.i386.rpm*

1、报错——缺少依赖包perl

error: Failed dependencies:

perl is needed by MySQL-server-5.1.73-1.glibc23.i386

*解决：yum install -y perl-devel*

2、报错——缺少依赖包libc.so.6

*[root@mini ~]# rpm -ivh MySQL-server-5.1.73-1.glibc23.i386.rpm*

error: Failed dependencies:

libc.so.6 is needed by MySQL-server-5.1.73-1.glibc23.i386

*解决：yum install -y libc.so.6*

3、报错——缺少依赖包 libgcc\_s.so.1

*[root@mini ~]# rpm -ivh MySQL-server-5.1.73-1.glibc23.i386.rpm*

error: Failed dependencies:

libgcc\_s.so.1 is needed by MySQL-server-5.1.73-1.glibc23.i386

*解决：yum install -y libc.so.1*

4、报错——依赖包冲突

*[root@mini ~]# rpm -ivh MySQL-server-5.1.73-1.glibc23.i386.rpm*

conflicts with file from package mysql-libs-5.1.73-5.el6\_6.x86\_64

*解决：*

*卸载依赖的冲突包： rpm -e mysql-libs-5.1.73-5.el6\_6 --nodeps*

5、安装完成——注意提示信息

再次执行安装

[root@mini ~]# rpm -ivh MySQL-server-5.1.73-1.glibc23.i386.rpm

*Preparing... ########################################### [100%]*

*1:MySQL-server ########################################### [100%]*

*PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !*

*To do so, start the server, then issue the following commands:*

*/usr/bin/mysqladmin -u root password 'new-password'*

*/usr/bin/mysqladmin -u root -h mini password 'new-password'*

*Alternatively you can run:*

*/usr/bin/mysql\_secure\_installation*

*\*\* 补充：*

如果发生rpm包冲突conflict，怎么处理

首先，找到有哪些包冲突

rpm -qa | grep mysql

然后，卸载冲突的包

rpm -e mysql-libs-5.1.73-5.el6\_6.x86\_64 --nodeps

#### 安装mysql客户端程序

1、用rpm命令安装客户端

*[root@mini ~]# rpm -ivh MySQL-client-5.1.73-1.glibc23.i386.rpm*

*error: Failed dependencies:*

*libncursesw.so.5 is needed by MySQL-client-5.1.73-1.glibc23.i386*

报错缺依赖，解决之：

yum install -y libncursesw.so.5 ## yum本地源需要更换光盘为dvd2

再次执行安装

[root@mini ~]# rpm -ivh MySQL-client-5.1.73-1.glibc23.i386.rpm

*Preparing... ########################################### [100%]*

*1:MySQL-client ########################################### [100%]*

#### 修改mysql用户root密码

安装完成后，先确定mysql服务进程是否启动

netstat -nltp

如果有mysql，会监听3306端口

如果没有，则手动启动

service mysql start

1、按照前面的提示，修改密码

[root@mini ~]# /usr/bin/mysql\_secure\_installation

进入交互式提示设置，按提示操作即可

补充：在资料中又给了一个**5.6的mysql**版本，这个安装完成后，mysql的root密码是自动生成的，注意看提示，密码保存在哪个位置

/root/.mysql\_secret

安装完成后，直接用这个生成的随机密码即可登录mysql

进入mysql后当然可以使用sql来修改root的密码

mysql>set password=password("root");

mysql>flush privileges;

4.5.4 开启mysql的**远程登录权限**

**mysql>GRANT ALL PRIVILEGES ON \*.\* TO 'myuser'@'%' IDENTIFIED BY 'mypassword' WITH GRANT OPTION;**

**mysql>FLUSH PRIVILEGES;**

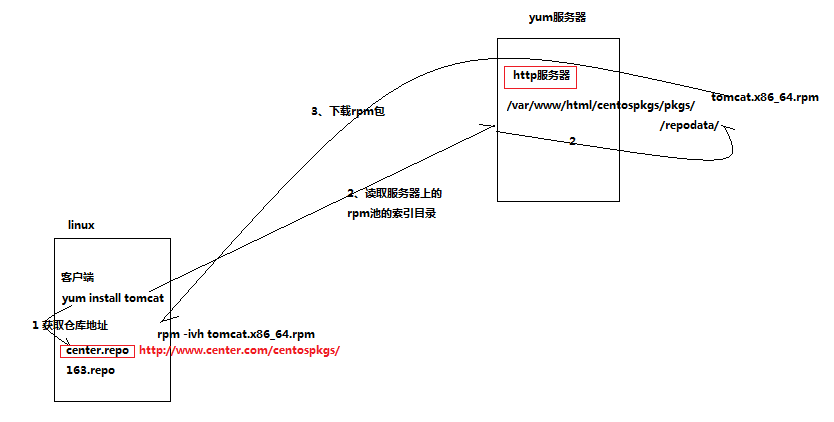
### YUM方式软件安装

#### 从网络源使用yum安装软件

yum install -y mysql

#### 使用本地Yum源安装软件

yum仓库服务器本质上就是一台http服务器，服务器的目录中放置了rpm包，及rpm包的索引信息文件，即可为yum客户端提供rpm文件下载



既然如此，则完全可以在“本地”制作私有的yum库来为内网中的linux提供yum安装源

#### 制作私有yum源的两种方式

##### 制作基于本地磁盘路径的yum源

本地yum源，即yum源就在系统的本地目录中，所以连web服务器都不用，只要准备好rpm库存放路径及相关索引配置信息即可

示例：将Centos-DVD.iso盘制作成一个本地yum源

##### 制作基于内网web服务器的yum源

总结：

1、在一台机器上部署一个httpd 的 web服务器，确保httpd运行成功(/var/www/html)

2、将光盘镜像文件插入虚拟机的光驱

3、将光驱挂载到httpd的html/centos目录中(/var/www/html)

4、修改yum的repo库配置文件/etc/yum.repos.d/centos-local-web.repo，如下：

|  |
| --- |
| [c6-local-web]  name=CentOS-local-web  baseurl=http://angelababy01/centos6  gpgcheck=0  enabled=1  gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-6 |

5、接下来，就可以用yum安装库中拥有的任何软件了

yum install -y tomcat6.x86\_64

详解版：

1、先安装一个http服务器： 首选httpd

先找一下我们要的包在仓库中有没有

yum list | grep httpd

yum install -y httpd

2、安装完成后，检查httpd的运行状态

service httpd status

3、在httpd服务的web目录中放入rpm库

比如，将centos安装光盘中的rpm库放入：

mkdir /var/www/html/centos

cp -r /mnt/cdrom/\* /var/www/html/centos/

*注：上面这种方式比较浪费空间，可以不用拷贝，其实只要创建一个软连接即可*

ln -s /mnt/cdrom /var/www/html/centos

mount -t iso9660 -o ro /dev/cdrom /var/www/centos/ #另外的方法

4、用浏览器访问一下httpd服务，看是否能看到rpm库

http://mini1/centos

5、配置yum客户端repo地址文件

将本地http服务器加入repo地址

首先，将内置的源全部disable掉

cd /etc/yum.repos.d/

rename .repo .repo.bak \*

然后，新建一个repo源，

vi innet.repo

|  |
| --- |
| [innet]  name=innet  baseurl=http://192.168.33.20/centos  gpgcheck=0  enabled=1  gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-6  ~ |

接下来，检查我们自定义的源是否生效

|  |
| --- |
| [root@shizhan01 yum.repos.d]# yum clean all  [root@shizhan01 yum.repos.d]# yum repolist all  Loaded plugins: fastestmirror, refresh-packagekit, security  Loading mirror speeds from cached hostfile  innet | 4.0 kB 00:00  innet/primary\_db | 4.6 MB 00:00  repo id repo name status  innet innet 6,575  repolist: 6,575 |

再接下来，就可以使用yum像从公网安装软件一样在内网服务器上下载软件进行安装了

yum install -y 你要的软件

##### ~~自定义rmp包repo~~

1、在http服务器的web目录/var/www/html下建一个文件夹用来存放rpm包

mkdir /var/www/html/myrpms

cp ~/MySQL-\* /var/www/html/myrpms

2、然后执行命令生成索引信息：

cd /var/www/html/myrpms

createrepo ./

3、如果提示命令找不到

[root@mini ~]# createrepo

-bash: createrepo: command not found

则安装该命令

[root@mini ~]# yum install -y createrepo

4、命令安装成功后，再次执行

[root@mini html]# createrepo ./

*Spawning worker 0 with 2 pkgs*

*Workers Finished*

*Gathering worker results*

*Saving Primary metadata*

*Saving file lists metadata*

*Saving other metadata*

*Generating sqlite DBs*

*Sqlite DBs complete*

5、可以看到生成的索引信息



6、通过yum客户端查看

yum list | grep MySQL

### C语言软件源码编译安装——redis服务器安装

用源码工程来编译安装

1/ 到官网下载最新stable版

2/ 解压源码并进入目录 tar -zxvf redis-2.8.19.tar.gz -C ./redis-src/

3/ make

如果报错提示缺少gcc，则安装gcc ： yum install -y gcc

如果报错提示：Newer version of jemalloc required

则在make时加参数：make MALLOC=libc

4/ 安装redis，指定安装目录,如 /usr/local/redis

make PREFIX=/usr/local/redis install

6/ 拷贝一份配置文件到安装目录下

切换到源码目录，里面有一份配置文件 redis.conf，然后将其拷贝到安装路径下

cp redis.conf /usr/local/redis/

7/ 启动redis

cd /usr/local/redis

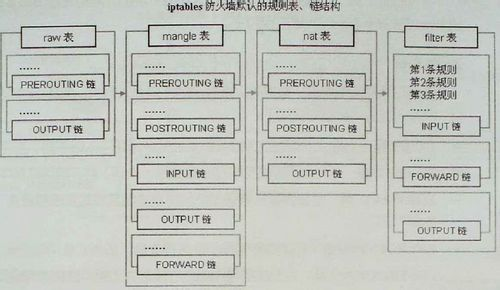
bin/redis-server redis.conf

### Linux防火墙配置

#### 防火墙配置

防火墙根据配置文件/etc/sysconfig/iptables来控制本机的“出、入”网络访问行为

其对行为的配置策略有四个策略表



#### 必备技能

查看防火墙状态

service iptables status

关闭防火墙

service iptables stop

启动防火墙

service iptables start

禁止防火墙自启

chkconfig iptables off

#### ~~扩展了解~~

1、列出iptables规则

iptables -L -n

列出iptables规则并显示规则编号

iptables -L -n --line-numbers

2、列出iptables nat表规则（默认是filter表）

iptables -L -n -t nat

3、清除默认规则（注意默认是filter表，如果对nat表操作要加-t nat）

#清除所有规则

iptables -F

#重启iptables发现规则依然存在，因为没有保存

service iptables restart

#保存配置

service iptables save

4、禁止ssh登陆（若果服务器在机房，一定要小心）

iptables -A INPUT -p tcp --dport 22 -j DROP

#删除规则

iptables -D INPUT -p tcp --dport 22 -j DROP

加入一条INPUT规则开放80端口

iptables -I INPUT -p tcp --dport 80 -j ACCEPT

~~加入一条NAT规则~~

~~iptables -t nat -A PREROUTING -d 119.254.228.20 -p tcp -m tcp --dport 8022 -j DNAT --to-destination 172.16.0.111:22~~

~~删除一条NAT规则~~

~~iptables -t nat -D PREROUTING 6~~

### 高级文本处理命令

#### cut命令

cut命令可以从一个文本文件或者文本流中提取文本列。

cut语法

[root@www ~]# cut -d'分隔字符' -f fields ## 用于有特定分隔字符

[root@www ~]# cut -c 字符区间 ## 用于排列整齐的信息

选项与参数：

-d：后面接分隔字符。与 -f 一起使用；

-f：依据 -d 的分隔字符将一段信息分割成为数段，用 -f 取出第几段的意思；

-c：以字符 (characters) 的单位取出固定字符区间；

PATH 变量如下

[root@www ~]# echo $PATH

*/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/bin:/usr/X11R6/bin:/usr/games*

*# 1 | 2 | 3 | 4 | 5 | 6 | 7*

将 PATH 变量取出，找出第五个路径。

# echo $PATH | cut -d':' -f 5

*/usr/local/bin*

将 PATH 变量取出，找出第三和第五个路径。

#echo $PATH | cut -d ':' -f 3,5

*/sbin:/usr/local/bin*

将 PATH 变量取出，找出第三到最后一个路径。

echo $PATH | cut -d ':' -f 3-

*/sbin:/usr/sbin:/usr/local/bin:/usr/X11R6/bin:/usr/games*

将 PATH 变量取出，找出第一到第三个路径。

#echo $PATH | cut -d ':' -f 1-3

*/bin:/usr/bin:/sbin:*

将 PATH 变量取出，找出第一到第三，还有第五个路径。

#echo $PATH | cut -d ':' -f 1-3,5

*/bin:/usr/bin:/sbin:/usr/local/bin*

实用例子:只显示/etc/passwd的用户和shell

#cat /etc/passwd | cut -d ':' -f 1,7

*root:/bin/bash*

*daemon:/bin/sh*

*bin:/bin/sh*

#### sed命令

**1/ 删除：d命令**

$ sed '2d' example -----删除example文件的第二行。

$ sed '2,$d' example -----删除example文件的第二行到末尾所有行。

$ sed '$d' example -----删除example文件的最后一行。

$ sed '/test/'d example -----删除example文件所有包含test的行。

**2/ 替换：s命令**

$ sed 's/test/mytest/g' example

*## 在整行范围内把test替换为mytest。如果没有g标记，则只有每行第一个匹配的test被替换成mytest。*

$ sed -n 's/^test/mytest/p' example

*## (-n)选项和p标志一起使用表示只打印那些发生替换的行。也就是说，如果某一行开头的test被替换成mytest，就打印它。*

$ sed 's/^192.168.0.1/&localhost/' example

*## &符号表示追加一个串到找到的串后。所有以192.168.0.1开头的行都会被替换成它自已加 localhost，变成192.168.0.1localhost。*

$ sed -n 's/\(love\)able/\1rs/p' example

*## love被标记为1，所有loveable会被替换成lovers，而且替换的行会被打印出来。*

$ sed 's#10#100#g' example

*## 不论什么字符，紧跟着s命令的都被认为是新的分隔符，所以，“#”在这里是分隔符，代替了默认的“/”分隔符。表示把所有10替换成100。*

*选定行的范围：逗号*

$ sed -n '/test/,/check/p' example

*## 所有在模板test和check所确定的范围内的行都被打印。*

$ sed -n '5,/^test/p' example

*## 打印从第五行开始到第一个包含以test开始的行之间的所有行。*

$ sed '/test/,/check/s/$/sed test/' example

*## 对于模板test和west之间的行，每行的末尾用字符串sed test替换。*

*多点编辑：e命令*

$ sed -e '1,5d' -e 's/test/check/' example

*## (-e)选项允许在同一行里执行多条命令。如例子所示，第一条命令删除1至5行，第二条命令用check替换test。命令的执行顺序对结果有影响。如果两个命令都是替换命令，那么第一个替换命令将影响第二个替换命令的结果。*

$ sed --expression='s/test/check/' --expression='/love/d' example

*## 一个比-e更好的命令是--expression。它能给sed表达式赋值。*

**3/ 从文件读入：r命令**

$ sed '/test/r file' example

*-----file里的内容被读进来，显示在与test匹配的行下面，如果匹配多行，则file的内容将显示在所有匹配行的下面。*

**4/ 写入文件：w命令**

$ sed -n '/test/w file' example

-----在example中所有包含test的行都被写入file里。

**5/ 追加命令：a命令**

$ sed '/^test/a\\--->this is a example' example

*## '--->this is a example'被追加到以test开头的行后面，sed要求命令a后面有一个反斜杠。*

**6/ 插入：i命令**

$ sed '/test/i\\some thing new -------------------------' example

如果test被匹配，则把反斜杠后面的文本插入到匹配行的前面。

**7/ 下一个：n命令**

$ sed '/test/{ n; s/aa/bb/; }' example

-----如果test被匹配，则移动到匹配行的下一行，替换这一行的aa，变为bb，并打印该行，然后继续。

**8/ 退出：q命令**

$ sed '10q' example

-----打印完第10行后，退出sed。

#### awk命令

awk是一个强大的文本分析工具，相对于grep的查找，sed的编辑，awk在其对数据分析并生成报告时，显得尤为强大。简单来说awk就是把文件逐行的读入，以空格为默认分隔符将每行切片，切开的部分再进行各种分析处理。

假设last -n 5的输出如下

[root@www ~]# last -n 5 ## 仅取出前五行

*root pts/1 192.168.1.100 Tue Feb 10 11:21 still logged in*

*root pts/1 192.168.1.100 Tue Feb 10 00:46 - 02:28 (01:41)*

*root pts/1 192.168.1.100 Mon Feb 9 11:41 - 18:30 (06:48)*

*dmtsai pts/1 192.168.1.100 Mon Feb 9 11:41 - 11:41 (00:00)*

*root tty1 Fri Sep 5 14:09 - 14:10 (00:01)*

如果只是显示最近登录的5个帐号

# last -n 5 | awk '{print $1}'

*root*

*root*

*root*

*dmtsai*

*root*

awk工作流程是这样的：读入有'\n'换行符分割的一条记录，然后将记录按指定的域分隔符划分域，填充域，$0则表示所有域,$1表示第一个域,$n表示第n个域。默认域分隔符是"空白键" 或 "[tab]键",所以$1表示登录用户，$3表示登录用户ip,以此类推。

如果只是显示/etc/passwd的账户

# cat /etc/passwd |awk -F ':' '{print $1}'

*root*

*daemon*

*bin*

*sys*

这种是awk+action的示例，每行都会执行action{print $1}。

-F指定域分隔符为':'

如果只是显示/etc/passwd的账户和账户对应的shell,而账户与shell之间以tab键分割

#cat /etc/passwd |awk -F ':' '{print $1"\t"$7}'

root /bin/bash

daemon /bin/sh

bin /bin/sh

sys /bin/sh

如果只是显示/etc/passwd的账户和账户对应的shell,而账户与shell之间以逗号分割,而且在所有行添加列名name,shell,在最后一行添加"blue,/bin/nosh"。

cat /etc/passwd |awk -F ':' 'BEGIN {print "**name,shell**"} {print $1","$7} END {print "**blue,/bin/nosh**"}'

***name,shell***

*root,/bin/bash*

*daemon,/bin/sh*

*bin,/bin/sh*

*sys,/bin/sh*

*....*

***blue,/bin/nosh***

awk工作流程是这样的：先执行BEGING，然后读取文件，读入有/n换行符分割的一条记录，然后将记录按指定的域分隔符划分域，填充域，$0则表示所有域,$1表示第一个域,$n表示第n个域,随后开始执行模式所对应的动作action。接着开始读入第二条记录······直到所有的记录都读完，最后执行END操作。

搜索/etc/passwd有root关键字的所有行

#awk -F: '/root/' /etc/passwd

root:x:0:0:root:/root:/bin/bash

这种是pattern的使用示例，匹配了pattern(这里是root)的行才会执行action(没有指定action，默认输出每行的内容)。

搜索支持正则，例如找root开头的: awk -F: '/^root/' /etc/passwd

搜索/etc/passwd有root关键字的所有行，并显示对应的shell

# awk -F':' '/root/{print $7}' /etc/passwd

/bin/bash

这里指定了action{print $7}

统计/etc/passwd:文件名，每行的行号，每行的列数，对应的完整行内容:

#awk -F ':' '{print "filename:" FILENAME ",linenumber:" NR ",columns:" NF ",linecontent:"$0}' */etc/passwd*

*filename:/etc/passwd,linenumber:1,columns:7,linecontent:root:x:0:0:root:/root:/bin/bash*

*filename:/etc/passwd,linenumber:2,columns:7,linecontent:daemon:x:1:1:daemon:/usr/sbin:/bin/sh*

*filename:/etc/passwd,linenumber:3,columns:7,linecontent:bin:x:2:2:bin:/bin:/bin/sh*

*filename:/etc/passwd,linenumber:4,columns:7,linecontent:sys:x:3:3:sys:/dev:/bin/sh*

使用printf替代print,可以让代码更加简洁，易读

awk -F ':' '{printf("filename:%s,linenumber:%s,columns:%s,linecontent:%s\n",FILENAME,NR,NF,$0)}' /etc/passwd

### Linux定时任务crontab

#### 命令功能

通过crontab 命令，我们可以在固定的间隔时间执行指定的系统指令或 shell script脚本。时间间隔的单位可以是分钟、小时、日、月、周及以上的任意组合。这个命令非常适合周期性的日志分析或数据备份等工作。

#### 安装crontab

yum install crontabs

服务操作说明：

service crond start  ## 启动服务

service crond stop   ## 关闭服务

service crond restart ## 重启服务

service crond reload  ## 重新载入配置

## 查看crontab服务状态：

service crond status

## 手动启动crontab服务：

service crond start

## 查看crontab服务是否已设置为开机启动，执行命令：

chkconfig --list

## 加入开机自动启动：

chkconfig  --level 35 crond on

#### 命令格式

crontab [-u user] file

crontab [-u user] [ -e | -l | -r ]

参数说明：

-u user：用来设定某个用户的crontab服务，例如，“-u ixdba”表示设定ixdba用户的crontab服务，此参数一般有root用户来运行。

file：file是命令文件的名字,表示将file做为crontab的任务列表文件并载入crontab。

-e：编辑某个用户的crontab文件内容。如果不指定用户，则表示编辑当前用户的crontab文件。

-l：显示某个用户的crontab文件内容，如果不指定用户，则表示显示当前用户的crontab文件内容。

-r：删除定时任务配置，从/var/spool/cron目录中删除某个用户的crontab文件，如果不指定用户，则默认删除当前用户的crontab文件。

-i：在删除用户的crontab文件时给确认提示。

命令示例：

crontab file [-u user] ## 用指定的文件替代目前的crontab。

***必掌握：***

crontab -l [-u user] ## 列出用户目前的crontab.

crontab -e [-u user] ## 编辑用户目前的crontab.

#### 调度配置

##### 配置说明

基本格式 :

\*　　\*　　\*　　\*　　\*　　command

分　 时　 日　 月　 周　 命令

第1列表示分钟1～59 每分钟用\*或者 \*/1表示

第2列表示小时0～23（0表示0点） 7-9表示：8点到10点之间

第3列表示日期1～31

第4列表示月份1～12

第5列标识号星期0～6（0表示星期天）

第6列要运行的命令

##### 配置示例

\*/1 \* \* \* \* date >> /root/date.txt

上面的例子表示每分钟执行一次date命令

30 21 \* \* \* /usr/local/etc/rc.d/httpd restart

上面的例子表示每晚的21:30重启apache。

45 4 1,10,22 \* \* /usr/local/etc/rc.d/httpd restart

上面的例子表示每月1、10、22日的4 : 45重启apache。

10 1 \* \* 6,0 /usr/local/etc/rc.d/httpd restart

上面的例子表示每周六、周日的1 : 10重启apache。

0,30 18-23 \* \* \* /usr/local/etc/rc.d/httpd restart

上面的例子表示在每天18 : 00至23 : 00之间每隔30分钟重启apache。

0 23 \* \* 6 /usr/local/etc/rc.d/httpd restart

上面的例子表示每星期六的11 : 00 pm重启apache。

\* \*/1 \* \* \* /usr/local/etc/rc.d/httpd restart

上面的例子每一小时重启apache

\* 23-7/1 \* \* \* /usr/local/etc/rc.d/httpd restart

上面的例子晚上11点到早上7点之间，每隔一小时重启apache

0 11 4 \* mon-wed /usr/local/etc/rc.d/httpd restart

上面的例子每月的4号与每周一到周三的11点重启apache

0 4 1 jan \* /usr/local/etc/rc.d/httpd restart

上面的例子一月一号的4点重启apache

## shell编程

Shell是用户与内核进行交互操作的一种接口，目前最流行的Shell称为bash Shell

Shell也是一门编程语言**<解释型的编程语言>**，即shell脚本<就是在用linux的shell命令编程>

一个系统可以存在多个shell，可以通过cat /etc/shells命令查看系统中安装的shell，不同的shell可能支持的命令语法是不相同的

### 基本格式

代码写在普通文本文件中，通常以 .sh为后缀名

vi hello.sh

|  |
| --- |
| #!/bin/bash ## 表示用哪一种shell解析器来解析执行我们的这个脚本程序  echo "hello world" ## 注释也可以写在这里  ## 这是一行注释 |

执行脚本

sh hello.sh

或给脚本添加x权限，直接执行

chmod 755 hello.sh

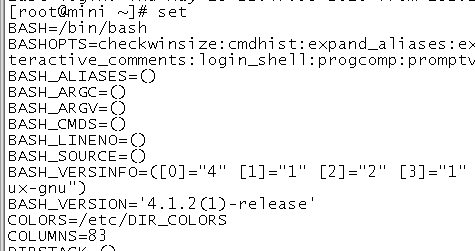
./hello.sh

### 基本语法

#### 系统变量

Linux Shell中的变量分为“系统变量”和“用户自定义变量”

可以通过set命令查看系统变量



系统变量：$HOME、$PWD、$SHELL、$USER等等

#### 自定义变量

**1、语法**

变量＝值 （例如STR=abc）

等号两侧不能有空格

变量名称一般习惯为大写

使用变量： $arg

双引号和单引号有区别，

**双引号仅将空格脱意，会解析变量意思**

**单引号会将变量引用比如$param脱意 不会解析变量**

**2、示例**

STR="hello world"

A=9

echo $A

echo $STR

如果想打印 hello worlds is greater 怎么办？

*echo $STRs is greate 行吗？*

不行，正确写法是：

echo **${*STR}****s is greate*

unset A 撤销变量 A

readonly B=2 声明静态的变量 B=2，不能 unset

export A #可把变量提升为当前shell进程中的全局环境变量，可供其他子shell程序使用

注意理解export：

|  |
| --- |
| [root@shizhan01 scripts]# vi a.sh  #!/bin/bash  a="a in a.sh"  echo $a  /root/scripts/b.sh |

|  |
| --- |
| [root@shizhan01 scripts]# vi b.sh  #!/bin/bash  b="b in b.sh"  echo $b  echo $a |

然后执行 ./a.sh ，会发现 b脚本中并没有把a脚本中定义的a变量打印出来

如果要在b中打印出a脚本的变量a，需要在a脚本中把变量a做export定义

此时，a变量就成了a.sh脚本所在bash进程的全局变量，该进程的所有子进程都能访问到变量a

另一种方式：

如果在a.sh脚本中用如下方式调用b.sh

|  |
| --- |
| . ./b.sh ## 注意：重点关注最前面那个 “.”号 |

或者

|  |
| --- |
| source ./b.sh ## |

则，b.sh就在a.sh所在的bash进程空间中运行

**总结：**

**1、a.sh中直接调用b.sh，会让b.sh在a所在的bash进程的“子进程”空间中执行**

**2、而子进程空间只能访问父进程中用export定义的变量**

**3、一个shell进程无法将自己定义的变量提升到父进程空间中去**

**4、“.”号执行脚本时，会让脚本在调用者所在的shell进程空间中执行**

**3、反引号赋值**

A=`ls -la` ## 反引号，运行里面的命令，并把结果返回给变量A

A=$(ls -la) ## 等价于反引号

**4、特殊变量**

**$? 表示上一个命令退出的状态码**

$$ 表示当前进程编号

**$0 表示当前脚本名称** /etc/init.d/crond

**$n 表示n位置的输入参数（n代表数字，n>=1）**

**$# 表示参数的个数，常用于循环**

$\*和$@ 都表示参数列表

/etc/init.d/rpcbind

***注：$\*与$@区别***

$\* 和 $@ 都表示传递给函数或脚本的所有参数

* 不被双引号" "包含时——

$\* 和 $@ 都以$1 $2 … $n 的形式组成参数列表

* 当它们被双引号" "包含时——

"$\*" 会将所有的参数作为一个整体，以**"**$1 $2 … $n**"**的形式组成一个整串；

"$@" 会将各个参数分开，以"$1" "$2" … "$n" 的形式组成一个参数列表

### 运算符

#### 算数运算

* **用expr**

格式 expr m + n 或$((m+n)) 注意expr运算符间要有空格

例如计算（2＋3 ）×4 的值

1 .分步计算  
 S=`expr 2 + 3` 注意加号左右有空格  
 expr $S \\* 4 ## \*号需要转义

2.一步完成计算

expr `expr 2 + 3 ` \\* 4

echo `expr \`expr 2 + 3\` \\* 4`

* **用(())**

((1+2))

(((2+3)\*4))

count=1

((count++))

echo $count

但是要想取到运算结果，需要用$引用

a=$((1+2))

* **用$[]执行运算**

a=$[1+2]

echo $a

### 流程控制

#### if语法([]条件判断)

* **语法格式**

if condition   
then   
    statements   
[elif condition   
    then statements. ..]   
[else   
    statements ]   
fi

* **示例**

|  |
| --- |
| #!/bin/bash  **read -p "please input your name:" NAME** ## read命令用于从控制台读取输入数据  ## printf '%s\n' $NAME  if [ $NAME = root ] #注意空格  then  echo "hello ${NAME}, welcome !"  elif [ $NAME = itcast ]  then  echo "hello ${NAME}, welcome !"  else  echo "SB, get out here !"  fi |

* **判断条件**

1/ 条件判断基本语法

[ condition ]  *(注意condition前后要有空格)*

#非空返回true，可使用$?验证（0为true，>1为false）

[ itcast ]

#空返回false

[ ]

注意[ ]内部的=周边的空格，有区别：

|  |
| --- |
| [root@shizhan01 scripts]# if [ a = b ];then echo ok;else echo notok;fi  notok  [root@shizhan01 scripts]# if [ a=b ];then echo ok;else echo notok;fi  ok |

短路（理解为三元运算符）

[ condition ] && echo OK || echo notok

条件满足，执行&&后面的语句；条件不满足，执行|| 后面的语句

2/ 条件判断组合

注：[] 与[[ ]] 的区别：[[ ]] 中逻辑组合可以使用 && || 符号

而[] 里面逻辑组合可以用 -a -o

|  |
| --- |
| [root@mini ~]# if [ a = b && b = c ]; then echo ok;else echo notok;fi  *-bash: [: missing `]'*  *notok*  [root@mini ~]# if [ a = b -a b = b ]; then echo ok;else echo notok;fi  *notok*  [root@mini ~]# if [ a = b -o b = b ]; then echo ok;else echo notok;fi  *ok*  [root@mini ~]# if [[ a = b && b = b ]]; then echo ok;else echo notok;fi  *notok*  [root@mini ~]# if [[ a = b || b = b ]]; then echo ok;else echo notok;fi  *ok* |

3/ 常用判断运算符

**字符串比较**：= !=

-z 字符串长度是为0返回true

-n 字符串长度是不为0返回true

if [ 'aa' = 'bb' ]; then echo ok; else echo notok;fi

if [ -n "aa" ]; then echo ok; else echo notok;fi

if [ -z "" ]; then echo ok; else echo notok;fi

**整数比较：**

-lt 小于

-le 小于等于

-eq 等于

-gt 大于

-ge 大于等于

-ne 不等于

**文件判断：**

-d 是否为目录

if [ -d /bin ]; then echo ok; else echo notok;fi

-f 是否为文件

*if [ -f /bin/ls ]; then echo ok; else echo notok;fi*

-e 是否存在

*if [ -e /bin/ls ]; then echo ok; else echo notok;fi*

#### while语法

* **方式一**

while expression

do

command

…

done

* **方式二**

i=1

while ((i<=3))

do

echo $i

let i++

done

#### case语法

case $1 in

start)

echo "starting"

;;

stop)

echo "stoping"

;;

\*)

echo "Usage: {start|stop}"

esac

#### for语法

* **方式一**

for N in 1 2 3

do

echo $N

done

或

for N in 1 2 3; do echo $N; done

或

for N in {1..3}; do echo $N; done

* **方式二**

for ((i = 0; i <= 5; i++))

do

echo "welcome $i times"

done

或

for ((i = 0; i <= 5; i++)); do echo "welcome $i times"; done

### 函数使用

* **1.函数定义**

|  |
| --- |
| #!/bin/sh  # func1.sh  hello() ## 函数定义  {  echo "Hello there today's date is `date +%Y-%m-%d`"  # return 2 ###返回值其实是状态码，只能在[0-255]范围内  }  hello  # echo $? 获取函数的return值  echo "now going to the function hello"  echo "back from the function" |

函数调用：

function hello()

或 function hello

或 hello

*注意：*

*1.必须在调用函数地方之前，先声明函数，shell脚本是逐行运行。不会像其它语言一样先预编译*

*2.函数返回值，只能通过$? 系统变量获得，可以显示加：return 返回，如果不加，将以最后一条命令运行结果，作为返回值。 return后跟数值n(0-255)*

*~~脚本调试：~~*

*sh -vx helloWorld.sh*

*或者在脚本中增加set -x*

* **2.函数参数**

|  |
| --- |
| #!/bin/bash  # fun1.sh  funWithParam(){  echo "第一个参数为 $1 !"  echo "第二个参数为 $2 !"  echo "第十个参数为 $10 !"  echo "第十个参数为 ${10} !"  echo "第十一个参数为 ${11} !"  echo "参数总数有 $# 个!"  echo "作为一个字符串输出所有参数 $\* !"  }  funWithParam 1 2 3 4 5 6 7 8 9 34 73 |

*注意，$10 不能获取第十个参数，获取第十个参数需要${10}。当n>=10时，需要使用${n}来获取参数。*

* **3.函数返回值**

|  |
| --- |
| #!/bin/bash  # fun2.sh  funWithReturn(){  echo "这个函数会对输入的两个数字进行相加运算..."  echo "输入第一个数字: "  read aNum  echo "输入第二个数字: "  read anotherNum  echo "两个数字分别为 $aNum 和 $anotherNum !"  return $(($aNum+$anotherNum))  }  funWithReturn  echo "输入的两个数字之和为 $? !" |

* 4.跨脚本调用函数

假如上述的脚本文件fun2.sh保存在此路径： /root/fun2.sh

则可在脚本fun\_other.sh中调用脚本fun2.sh中的函数

|  |
| --- |
| #!/bin/bash  # fun\_other.sh  **.**  /root/fun2.sh ## 注： **.** 和 **/** 之间有空格  # 或者 source /root/fun2.sh  funWithParam 11 22 33 44 55 66 77 88 99 100 101 |

### shell编程综合练习

自动化软件部署脚本

#### 需求

* **1.需求描述**

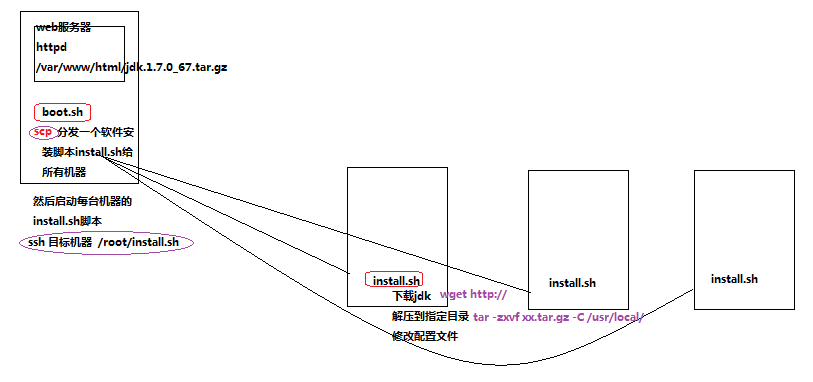
公司内有一个N个节点的集群，需要统一安装一些软件（jdk）

需要开发一个脚本，实现对集群中的N台节点批量自动下载、安装jdk

* **2.思路**

1/ 编写一个**启动脚本**，用来发送一个软件**安装脚本**到每一台机器

2/ 然后启动每台机器上的软件安装脚本来执行软件下载和安装



* **3.expect的使用**

痛点：使用scp命令远程拷贝文件时，会有人机交互的过程，如何让脚本完成人机交互？

妙药： expect

用法示例：

先观察 ssh localhost 的过程

再看expect的功能

|  |
| --- |
| #!/bin/bash/expect  ## exp\_test.sh  set timeout -1;  spawn ssh localhost;  expect {  "(yes/no)" {send "yes\r";exp\_continue;}  "password:" {send "hadoop\r";exp\_continue;}  eof {exit 0;}  } |

*执行： expect -f exp\_test.sh*

#### 准备内网软件下载服务器

选择一台服务器（比如mini）作为软件源服务器

* *安装httpd（如果已有，可跳过）*

yum install -y httpd

service httpd start

chkconfig --level 35 httpd on

制作局域网yum源

1/ 挂载centos安装光盘到/mnt/cdrom *（如果已有，可跳过）*

mkdir /mnt/cdrom

mount -t iso9660 -o loop /dev/cdrom /mnt/cdrom

2/ 将本地yum库放入httpd服务器

ln -s /mnt/cdrom /var/www/html/centos

*检查点：用浏览器访问 http://mini/centos 看能否看到光盘内容*

* 编写repo配置

vi /etc/yum.repos.d/centos.repo

|  |
| --- |
| [c6-httpd]  name=CentOS-httpd  baseurl=http://192.168.33.3/centos  gpgcheck=0  enabled=1 |

* 分发repo配置到局域网,这个可以不做

从母鸡shizhan01上把centos.repo拷贝给所有需要自动安装软件的服务器（仔鸡mini1/mini2）

cd /etc/yum.repos.d/

scp /etc/yum.repos.d/innet.repo mini1:$PWD

scp /etc/yum.repos.d/innet.repo mini2:$PWD

* 准备一个jdk安装包放在内网web服务器上

#### 脚本开发

* 1.启动脚本

vi **boot.sh**

|  |
| --- |
| #!/bin/bash  SERVERS="mini1 mini2"  PASSWORD=hadoop  BASE\_SERVER=192.168.33.11  ## 实现免密登陆配置的函数  auto\_ssh\_copy\_id() {  expect -c "set timeout -1;  spawn ssh-copy-id $1;  expect {  \*(yes/no)\* {send -- yes\r;exp\_continue;}  \*assword:\* {send -- $2\r;exp\_continue;}  eof {exit 0;}  }";  }  ssh\_copy\_id\_to\_all() {  for SERVER in $SERVERS  do  auto\_ssh\_copy\_id $SERVER $PASSWORD  done  }  ## 调用免密登陆配置函数，实现母鸡到各仔鸡的免密登陆配置  ssh\_copy\_id\_to\_all  ## 完成分发install.sh到各仔鸡的操作  ## 并让仔鸡启动install.sh  for SERVER in $SERVERS  do  scp install.sh root@$SERVER:/root  ssh root@$SERVER /root/install.sh  done |

* 2.安装执行脚本

vi install.sh

|  |
| --- |
| #!/bin/bash  BASE\_SERVER=192.168.33.11  ## 为本机安装wget命令  yum install -y wget  ## 使用wget从母鸡的web服务器上下载jdk压缩包  wget $BASE\_SERVER/soft/jdk-7u67-linux-x64.gz  ## 将下载的压缩包解压  tar -zxvf jdk-7u67-linux-x64.gz -C /usr/local  ## 修改profile配置文件  cat >> /etc/profile << EOF  export JAVA\_HOME=/usr/local/jdk1.7.0\_67  export PATH=\$PATH:\$JAVA\_HOME/bin  EOF |

* 3.启动脚本

只要在baseServer即mini上启动boot.sh即可

***作业：***

批量部署脚本，存在一些不太完美的地方，比如，目标机器名需要写死在脚本中，请改造这一点：

可以将所有需要安装软件的机器名写在一个文件：比如slaves中

让脚本自动读取slaves文件中的机器名来批量安装

cat slaves | while read host

do

echo $host

expect -c "set timeout -f

spawn ssh-copy-id $host"

done

## 分布式协调服务zookeeper

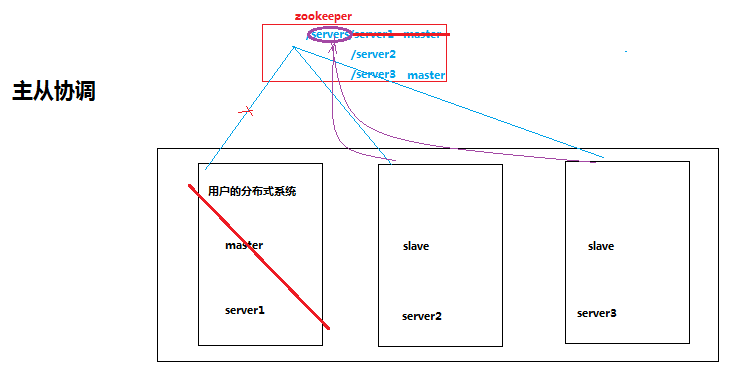
在线感知

1/所有服务节点，上线时，先去zookeeper上注册一个子节点

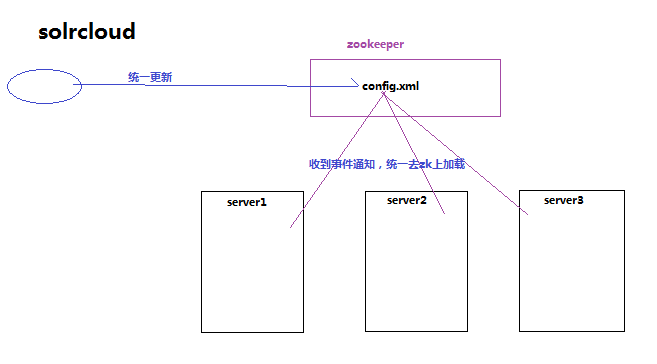
2/“服务请求者”，监听父节点

3/当有新的服务节点加入，或者有旧的服务节点掉线， “服务请求者”就能收到通知，就可以去zookeeper上查看最新的在线服务器列表

**主从协调**



**配置管理**



**名称服务**

**分布式锁**

### 简介及应用场景

* 概念介绍

Zookeeper是一个分布式协调服务；就是为用户的分布式应用程序提供协调服务

1、zookeeper是为别的分布式程序服务的

2、Zookeeper本身就是一个分布式程序（只要有半数以上节点存活，zk就能正常服务）

3、Zookeeper所提供的服务涵盖：主从协调、服务器节点动态上下线、统一配置管理、分布式共享锁、统一名称服务……

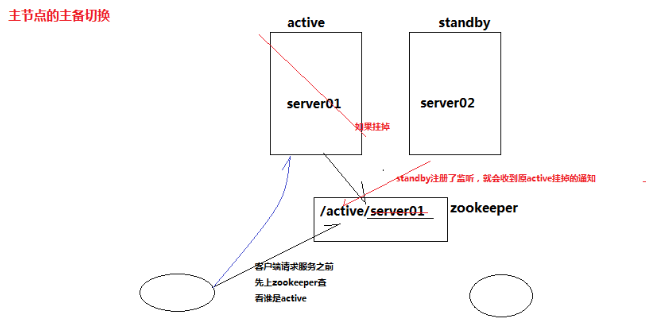
4、虽然说可以提供各种服务，但是zookeeper在底层其实只提供了**两个功能**：

a、管理(存储，读取)用户程序提交的数据；

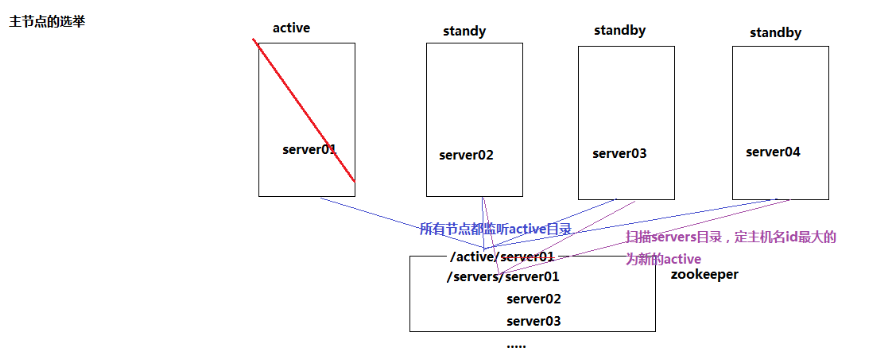
b、并为用户程序提供数据节点监听服务；

* 3.2 常用应用场景

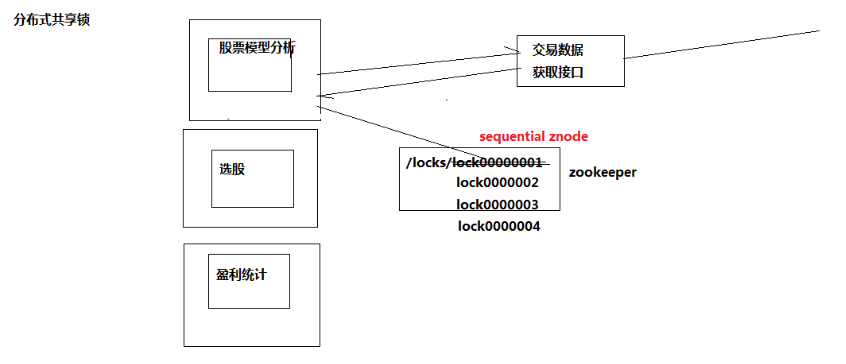
1.主节点的主备切换



2.主节点的选举



3.分布式共享锁



### 集群部署

#### Zookeeper工作机制

Zookeeper集群角色

Zookeeper集群的角色：Leader 和 follower （Observer）

zk集群最好配成奇数个节点

只要集群中有半数以上节点存活，集群就能提供服务

#### Zookeeper部署

* **step1:机器准备**

1/ 安装到3台虚拟机上

2/ 安装好JDK

3/ 上传安装包。上传用工具。

4/ 解压

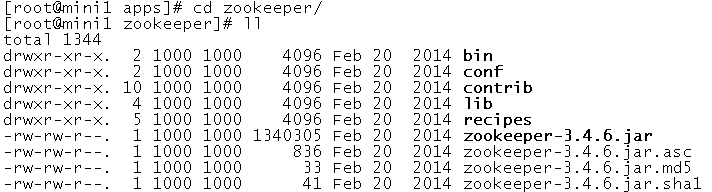
~~su - hadoop（切换到hadoop用户）~~

tar -zxvf zookeeper-3.4.5.tar.gz（解压）

5/ 重命名

mv zookeeper-3.4.5 zookeeper（重命名文件夹zookeeper-3.4.5为zookeeper）

可以删除里面一些源码工程相关的文件，剩下的是这些：



* **step2:修改环境变量**

*（注意：3台zookeeper都需要修改）*

~~1/ su – root(切换用户到root)~~

2/ vi /etc/profile(修改文件)

3/ 添加内容：

export ZOOKEEPER\_HOME=/home/hadoop/zookeeper

export PATH=$PATH:$ZOOKEEPER\_HOME/bin

4/ 加载环境配置：

source /etc/profile

5/ 修改完成后切换回hadoop用户：

~~su - hadoop~~

* **step3:修改Zookeeper配置文件**

1、用root用户操作

cd zookeeper/conf

cp zoo\_sample.cfg zoo.cfg

2、vi zoo.cfg

3、添加内容：

dataDir=/root/apps/zookeeper/zkdata

~~dataLogDir=/home/hadoop/zookeeper/log~~

server.1=mini1:2888:3888 ## (心跳端口、选举端口)

server.2=mini2:2888:3888

server.3=mini3:2888:3888

4、创建文件夹：

cd /home/hadoop/zookeeper/

mkdir zkdata

~~mkdir -m 755 log~~

5、在data文件夹下新建myid文件，myid的文件内容为：

cd zkdata

echo 1 > myid

* **step4:分发安装包到其他机器**

scp -r /root/apps root@mini2:/root/

scp -r /root/apps root@mini3:/root/

这样写也行:scp -r /root/apps/zk root@mini2:$pwd

* **step5:修改其他机器的配置文件**

1、修改myid文件

到mini2上：修改myid为：2

到mini3上：修改myid为：3

* **step6:启动（每台机器）**

注：

1、事先将三台服务器的防火墙都关掉

2、全网统一hosts映射

先配好一台上的hosts

然后：

scp /etc/hosts mini2:/etc

scp /etc/hosts mini3:/etc

3、然后一台一台地启动

bin/zkServer.sh start

或者编写一个脚本来批量启动所有机器：

for host in "mini1 mini2 mini3"

do

ssh $host "source/etc/profile;/root/apps/zookeeper/bin/zkServer.sh start"

done

* **step7:查看集群状态**

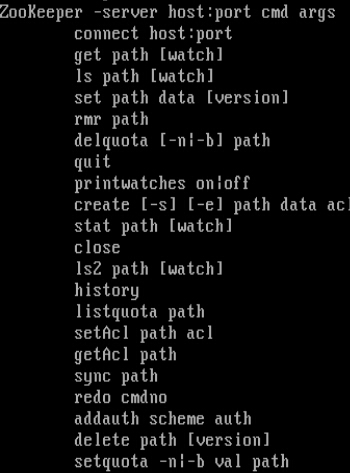
1、jps（查看进程）

2、zkServer.sh status（查看集群状态，主从信息）

### 命令行操作

* 客户端连接

运行 zkCli.sh –server <ip>进入命令行工具.help可以查看命令



* 查看znode路径

ls /mygirls

* 获取znode数据

get /mygirls

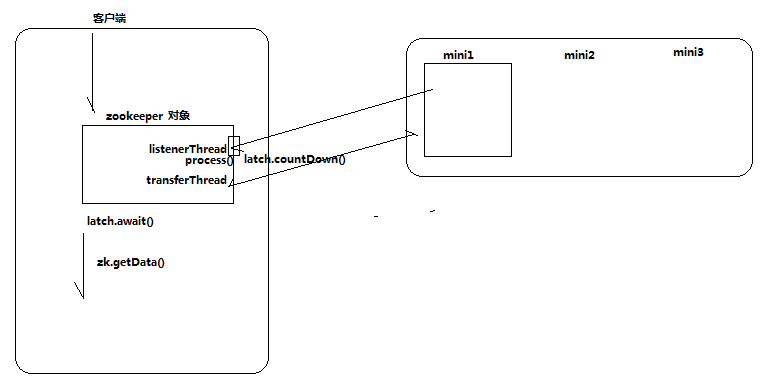
* 监听znode事件

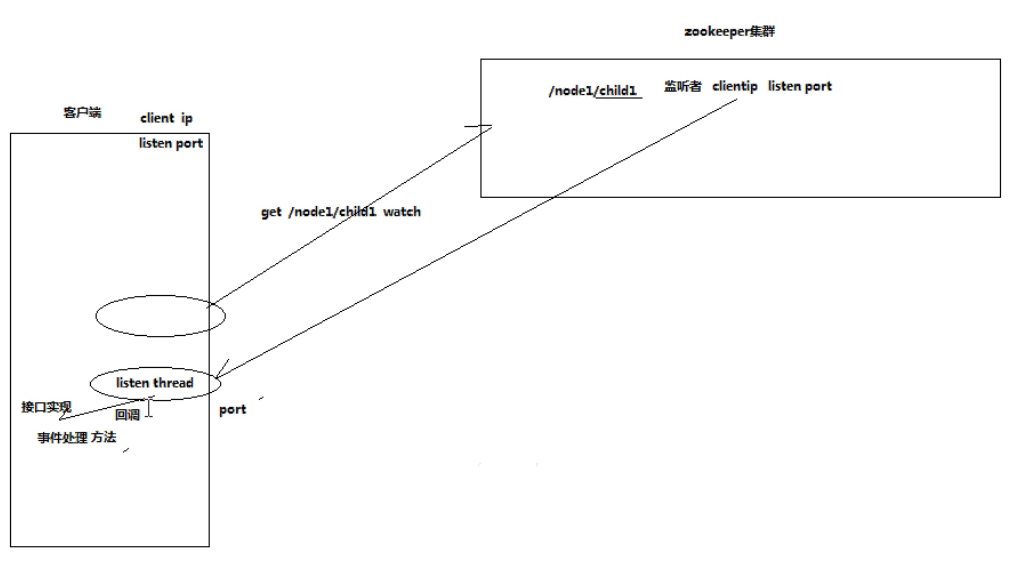
ls /mygirls watch ## 就对一个节点的子节点变化事件注册了监听

get /mygirls watch ## 就对一个节点的数据内容变化事件注册了监听

注意： 监听器只生效一次

监听器的工作机制，其实是在客户端会专门创建一个监听线程，在本机的一个端口上等待zk集群发送过来事件





### zookeeper客户端API

* 基本使用

org.apache.zookeeper.Zookeeper是客户端入口主类，负责建立与server的会话

它提供以下几类主要方法 ：

|  |  |
| --- | --- |
| 功能 | 描述 |
| create | 在本地目录树中创建一个节点 |
| delete | 删除一个节点 |
| exists | 测试本地是否存在目标节点 |
| get/set data | 从目标节点上读取 / 写数据 |
| get/set ACL | 获取 / 设置目标节点访问控制列表信息 |
| get children | 检索一个子节点上的列表 |
| sync | 等待要被传送的数据 |

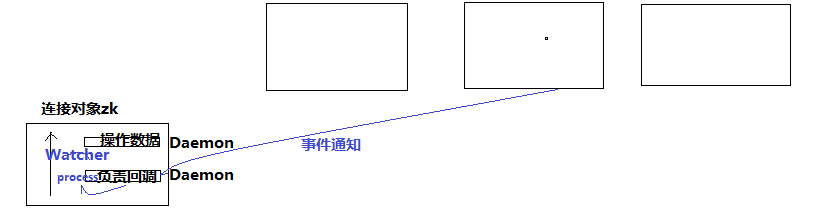
表 1 ： ZooKeeper API 描述

* 增删改查znode数据

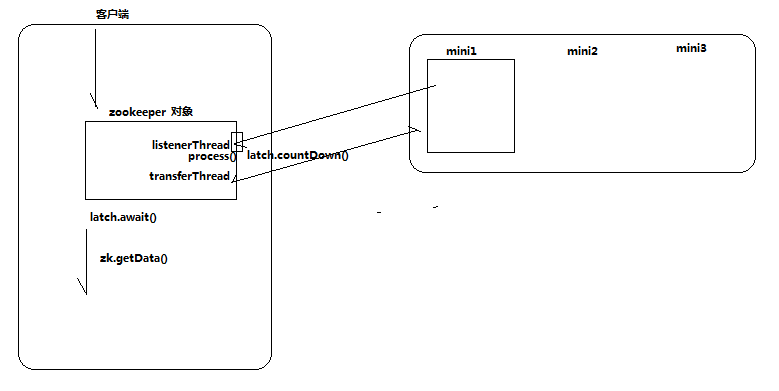
|  |
| --- |
| public class SimpleDemo {  // 会话超时时间，设置为与系统默认时间一致  private static final int SESSION\_TIMEOUT = 30000;  // 创建 ZooKeeper 实例  ZooKeeper zk;  // 创建 Watcher 实例  Watcher wh = new Watcher() {  public void process(org.apache.zookeeper.WatchedEvent event)  {  System.out.println(event.toString());  }  };  // 初始化 ZooKeeper 实例  private void createZKInstance() throws IOException  {  zk = new ZooKeeper("weekend01:2181", SimpleDemo.SESSION\_TIMEOUT, this.wh);  }  private void ZKOperations() throws IOException, InterruptedException, KeeperException  {  System.out.println("/n1. 创建 ZooKeeper 节点 (znode ： zoo2, 数据： myData2 ，权限： OPEN\_ACL\_UNSAFE ，节点类型： Persistent");  zk.create("/zoo2", "myData2".getBytes(), Ids.OPEN\_ACL\_UNSAFE, CreateMode.PERSISTENT);  System.out.println("/n2. 查看是否创建成功： ");  System.out.println(new String(zk.getData("/zoo2", false, null)));  System.out.println("/n3. 修改节点数据 ");  zk.setData("/zoo2", "shenlan211314".getBytes(), -1);  System.out.println("/n4. 查看是否修改成功： ");  System.out.println(new String(zk.getData("/zoo2", false, null)));  System.out.println("/n5. 删除节点 ");  zk.delete("/zoo2", -1);  System.out.println("/n6. 查看节点是否被删除： ");  System.out.println(" 节点状态： [" + zk.exists("/zoo2", false) + "]");  }  private void ZKClose() throws InterruptedException  {  zk.close();  }  public static void main(String[] args) throws IOException, InterruptedException, KeeperException {  SimpleDemo dm = new SimpleDemo();  dm.createZKInstance();  dm.ZKOperations();  dm.ZKClose();  }  } |

* 监听znode

Zookeeper的监听器工作机制



监听器是一个接口，我们的代码中可以实现Wather这个接口，实现其中的process方法，方法中即我们自己的业务逻辑



监听器的注册是在获取数据的操作中实现：

getData(path,watch?)监听的事件是：节点数据变化事件

getChildren(path,watch?)监听的事件是：节点下的子节点增减变化事件

### zookeeper核心工作机制

#### zookeeper特性

1、Zookeeper：一个leader，多个follower组成的集群

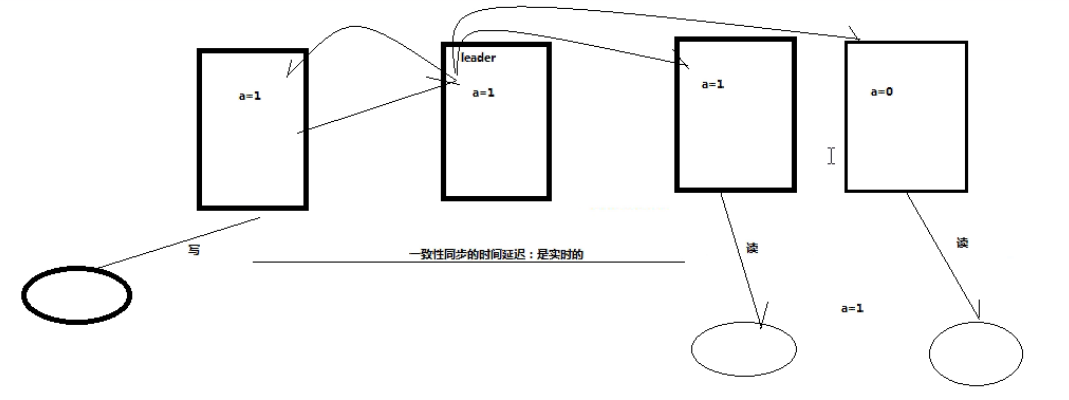
2、全局数据一致：每个server保存一份相同的数据副本，client无论连接到哪个server，数据都是一致的

3、分布式读写，更新请求转发，由leader实施

4、更新请求顺序进行，来自同一个client的更新请求按其发送顺序依次执行

5、数据更新原子性，一次数据更新要么成功（半数以上节点成功），要么失败

6、实时性，在一定时间范围内，client能读到最新数据



paxos一致性算法:读取数据的时候会去取版本来比对

#### zookeeper数据结构

* 概况

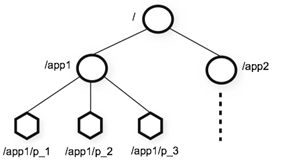
1、层次化的目录结构，命名符合常规文件系统规范(见下图)

2、每个节点在zookeeper中叫做znode,并且其有一个唯一的路径标识

3、节点Znode可以包含数据(只能存储很小量的数据，<1M;最好是1k字节以内)和子节点（但是EPHEMERAL类型的节点不能有子节点，下一页详细讲解）

4、客户端应用可以在节点上设置监视器（后续详细讲解）

* 数据结构图



* 节点类型

1、Znode有两种类型：

短暂（ephemeral）（断开连接自己删除）

持久（persistent）（断开连接不删除）

2、Znode有四种形式的目录节点（默认是persistent ）

PERSISTENT

PERSISTENT\_SEQUENTIAL（持久序列/test0000000019 ）

EPHEMERAL

EPHEMERAL\_SEQUENTIAL

3、创建znode时设置顺序标识，znode名称后会附加一个值，顺序号是一个单调递增的计数器，由父节点维护

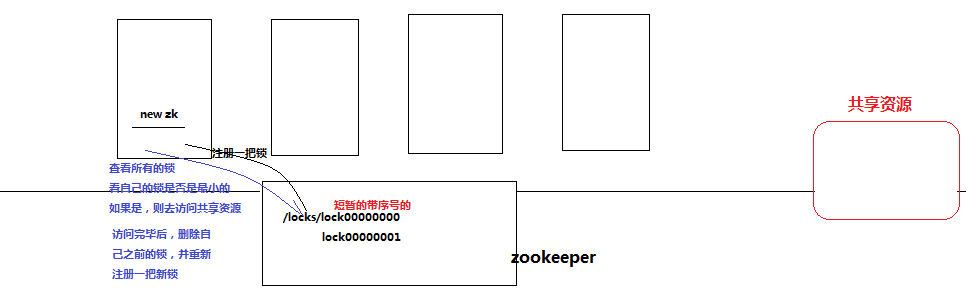
4、在分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序

### 应用案例-分布式共享锁

* 1、需求描述

在我们自己的分布式业务系统中，可能会存在某种资源，需要被整个系统的各台服务器共享访问，但是只允许一台服务器同时访问

* 2、设计思路



* 3、代码开发

|  |
| --- |
| public class DistributedClientMy {  // 超时时间  private static final int SESSION\_TIMEOUT = 5000;  // zookeeper server列表  private String hosts = "spark01:2181,spark02:2181,spark03:2181";  private String groupNode = "locks";  private String subNode = "sub";  private boolean haveLock = false;  private ZooKeeper zk;  // 当前client创建的子节点  private volatile String thisPath;  /\*\*  \* 连接zookeeper  \*/  public void connectZookeeper() throws Exception {  zk = new ZooKeeper("spark01:2181", SESSION\_TIMEOUT, new Watcher() {  public void process(WatchedEvent event) {  try {  // 子节点发生变化  if (event.getType() == EventType.NodeChildrenChanged && event.getPath().equals("/" + groupNode)) {  // thisPath是否是列表中的最小节点  List<String> childrenNodes = zk.getChildren("/" + groupNode, true);  String thisNode = thisPath.substring(("/" + groupNode + "/").length());  // 排序  Collections.sort(childrenNodes);  if (childrenNodes.indexOf(thisNode) == 0) {  doSomething();  thisPath = zk.create("/" + groupNode + "/" + subNode, null, Ids.OPEN\_ACL\_UNSAFE,  CreateMode.EPHEMERAL\_SEQUENTIAL);  }  }  } catch (Exception e) {  e.printStackTrace();  }  }  });  // 创建子节点  thisPath = zk.create("/" + groupNode + "/" + subNode, null, Ids.OPEN\_ACL\_UNSAFE,  CreateMode.EPHEMERAL\_SEQUENTIAL);  // wait一小会, 让结果更清晰一些  Thread.sleep(new Random().nextInt(1000));  // 监听子节点的变化  List<String> childrenNodes = zk.getChildren("/" + groupNode, true);  // 列表中只有一个子节点, 那肯定就是thisPath, 说明client获得锁  if (childrenNodes.size() == 1) {  doSomething();  thisPath = zk.create("/" + groupNode + "/" + subNode, null, Ids.OPEN\_ACL\_UNSAFE,  CreateMode.EPHEMERAL\_SEQUENTIAL);  }  }  /\*\*  \* 共享资源的访问逻辑写在这个方法中  \*/  private void doSomething() throws Exception {  try {  System.out.println("gain lock: " + thisPath);  Thread.sleep(2000);  // do something  } finally {  System.out.println("finished: " + thisPath);  // 将thisPath删除, 监听thisPath的client将获得通知  // 相当于释放锁  zk.delete(this.thisPath, -1);  }  }  public static void main(String[] args) throws Exception {  DistributedClientMy dl = new DistributedClientMy();  dl.connectZookeeper();  Thread.sleep(Long.MAX\_VALUE);  }    } |

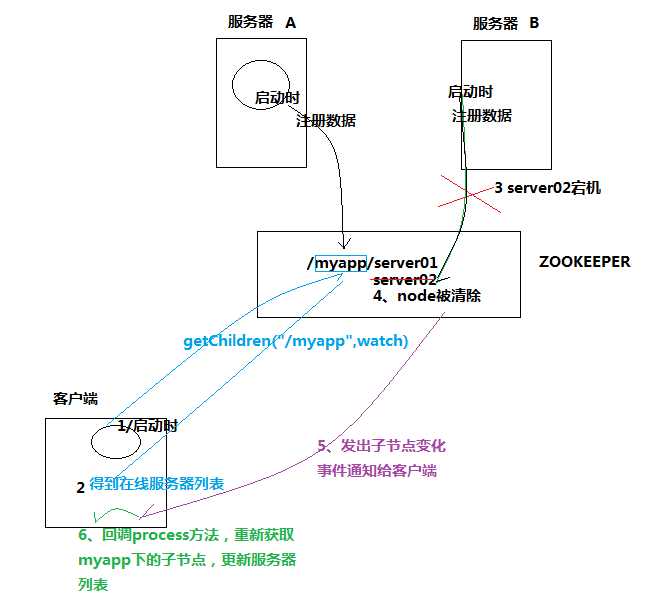
### 应用案例-服务器上下线感知

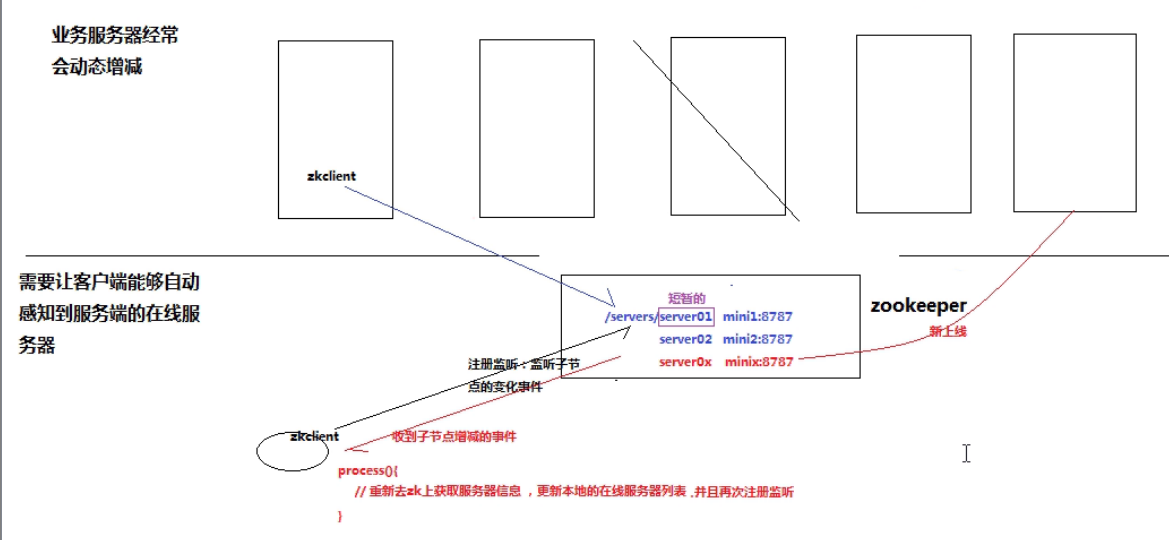
* 需求描述

某分布式系统中，主节点可以有多台，可以动态上下线

任意一台客户端都能实时感知到主节点服务器的上下线

* 设计思路





* 代码开发
* 客户端实现

|  |
| --- |
| public class AppClient {  private String groupNode = "sgroup";  private ZooKeeper zk;  private Stat stat = new Stat();  private volatile List<String> serverList;  /\*\*  \* 连接zookeeper  \*/  public void connectZookeeper() throws Exception {  zk  = new ZooKeeper("localhost:4180,localhost:4181,localhost:4182", 5000, new Watcher() {  public void process(WatchedEvent event) {  // 如果发生了"/sgroup"节点下的子节点变化事件, 更新server列表, 并重新注册监听  if (event.getType() == EventType.NodeChildrenChanged  && ("/" + groupNode).equals(event.getPath())) {  try {  updateServerList();  } catch (Exception e) {  e.printStackTrace();  }  }  }  });  updateServerList();  }  /\*\*  \* 更新server列表  \*/  private void updateServerList() throws Exception {  List<String> newServerList = new ArrayList<String>();  // 获取并监听groupNode的子节点变化  // watch参数为true, 表示监听子节点变化事件.  // 每次都需要重新注册监听, 因为一次注册, 只能监听一次事件, 如果还想继续保持监听, 必须重新注册  List<String> subList = zk.getChildren("/" + groupNode, true);  for (String subNode : subList) {  // 获取每个子节点下关联的server地址  byte[] data = zk.getData("/" + groupNode + "/" + subNode, false, stat);  newServerList.add(new String(data, "utf-8"));  }  // 替换server列表  serverList = newServerList;  System.out.println("server list updated: " + serverList);  }  /\*\*  \* client的工作逻辑写在这个方法中  \* 此处不做任何处理, 只让client sleep  \*/  public void handle() throws InterruptedException {  Thread.sleep(Long.MAX\_VALUE);  }  public static void main(String[] args) throws Exception {  AppClient ac = new AppClient();  ac.connectZookeeper();  ac.handle();  }  } |

* 2、服务器端实现

|  |
| --- |
| public class AppServer {  private String groupNode = "sgroup";  private String subNode = "sub";  /\*\*  \* 连接zookeeper  \* @param address server的地址  \*/  public void connectZookeeper(String address) throws Exception {  ZooKeeper zk = new ZooKeeper(  "localhost:4180,localhost:4181,localhost:4182",  5000, new Watcher() {  public void process(WatchedEvent event) {  // 不做处理  }  });  // 在"/sgroup"下创建子节点  // 子节点的类型设置为EPHEMERAL\_SEQUENTIAL, 表明这是一个临时节点, 且在子节点的名称后面加上一串数字后缀  // 将server的地址数据关联到新创建的子节点上  String createdPath = zk.create("/" + groupNode + "/" + subNode, address.getBytes("utf-8"),  Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL\_SEQUENTIAL);  System.out.println("create: " + createdPath);  }    /\*\*  \* server的工作逻辑写在这个方法中  \* 此处不做任何处理, 只让server sleep  \*/  public void handle() throws InterruptedException {  Thread.sleep(Long.MAX\_VALUE);  }    public static void main(String[] args) throws Exception {  // 在参数中指定server的地址  if (args.length == 0) {  System.err.println("The first argument must be server address");  System.exit(1);  }    AppServer as = new AppServer();  as.connectZookeeper(args[0]);  as.handle();  }  } |

### 原理补充

Zookeeper虽然在配置文件中并没有指定master和slave

但是，zookeeper工作时，是有一个节点为leader，其他则为follower

Leader是通过内部的选举机制临时产生的

* **zookeeper的选举机制（zk的数据一致性核心算法paxos）**

以一个简单的例子来说明整个选举的过程.

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

1) 服务器1启动,此时只有它一台服务器启动了,它发出去的报没有任何响应,所以它的选举状态一直是LOOKING状态

2) 服务器2启动,它与最开始启动的服务器1进行通信,互相交换自己的选举结果,由于两者都没有历史数据,所以id值较大的服务器2胜出,但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3),所以服务器1,2还是继续保持LOOKING状态.

3) 服务器3启动,根据前面的理论分析,服务器3成为服务器1,2,3中的老大,而与上面不同的是,此时有三台服务器选举了它,所以它成为了这次选举的leader.

4) 服务器4启动,根据前面的分析,理论上服务器4应该是服务器1,2,3,4中最大的,但是由于前面已经有半数以上的服务器选举了服务器3,所以它只能接收当小弟的命了.

5) 服务器5启动,同4一样,当小弟.

* **非全新集群的选举机制(数据恢复)**

那么，初始化的时候，是按照上述的说明进行选举的，但是当zookeeper运行了一段时间之后，有机器down掉，重新选举时，选举过程就相对复杂了。

需要加入数据version、leader id和逻辑时钟。

数据version：数据新的version就大，数据每次更新都会更新version。

Leader id：就是我们配置的myid中的值，每个机器一个。

逻辑时钟：这个值从0开始递增,每次选举对应一个值,也就是说:  如果在同一次选举中,那么这个值应该是一致的 ;  逻辑时钟值越大,说明这一次选举leader的进程更新.

选举的标准就变成：

1、逻辑时钟小的选举结果被忽略，重新投票

2、统一逻辑时钟后，数据id大的胜出

3、数据id相同的情况下，leader id大的胜出

根据这个规则选出leader。

## 内存数据库redis

### 介绍

* 01/ nosql介绍

NoSQL：一类新出现的数据库(not only sql)，它的特点：

1. 不支持SQL语法
2. 存储结构跟传统关系型数据库中的那种关系表完全不同，nosql中存储的数据都是KV形式
3. NoSQL的世界中没有一种通用的语言，每种nosql数据库都有自己的api和语法，以及擅长的业务场景
4. NoSQL中的产品种类相当多：
   1. Mongodb 文档型nosql数据库，擅长做CMS系统（内容管理系统）
   2. Redis 内存数据库，**数据结构**服务器，号称瑞士军刀（精巧），只要你有足够的想象力，它可以还给你无限惊喜
   3. Hbase hadoop生态系统中原生的一种nosql数据库，重量级的分布式nosql数据库，用于海量数据的场景
   4. Cassandra hadoop生态系统中原生的一种分布式nosql数据库，后起之秀

NoSQL和SQL数据库的比较：

1、适用场景不同：sql数据库适合用于关系特别复杂的数据查询场景，nosql反之

2、“事务”特性的支持：sql对事务的支持非常完善，而nosql基本不支持事务

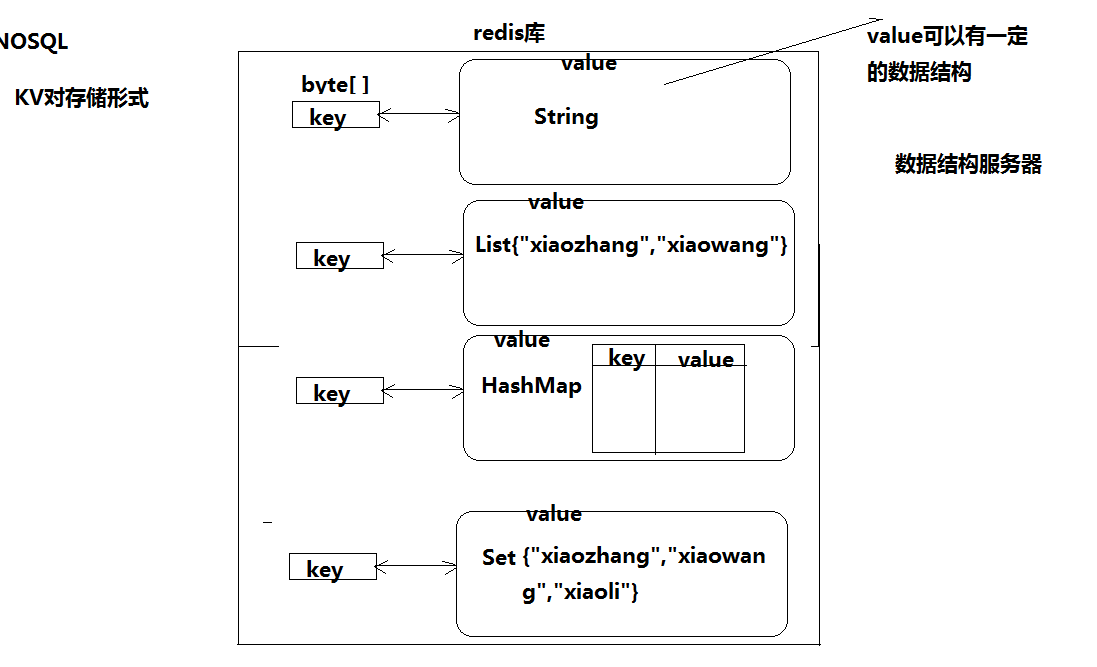
3、两者在不断地取长补短，呈现融合趋势

02/ redis介绍

* 2.1 简述

Redis是一个高性能的**kv对**缓存和内存数据库(存的不像mysql那样的表)

Redis的存储结构就是key-value，形式如下：



注： redis中的value内部可以支持各种数据结构类型，比如可以存入一个普通的string，还可以存list，set，hashmap，sortedSet（有序的set）

2.2 redis应用场景

A、用来做缓存(ehcache/memcached)——redis的所有数据是放在内存中的（内存数据库）

B、可以在某些特定应用场景下替代传统数据库——比如社交类的应用

C、在一些大型系统中，巧妙地实现一些特定的功能：session共享、购物车

只要你有丰富的想象力，redis可以用在可以给你无限的惊喜…….

2.3 redis的特性

1、redis数据访问速度快（数据在内存中）

2、redis有数据持久化机制（持久化机制有两种：1、定期将内存数据dump到磁盘；2、aof(append only file)持久化机制——用记日志的方式记录每一条数据更新操作，一旦出现灾难事件，可以通过日志重放来恢复整个数据库）

3、redis支持集群模式（容量可以线性扩展）

4、redis相比其他缓存工具（ehcach/memcached），有一个鲜明的优势：支持丰富的数据结构

### 安装redis

* 获取源码包

1、先去官网（<http://redis.io/download> ）下载一个源码工程（redis官网版本只支持linux/微软开源事业部维护了一个windows版本）

2、把安装包上传到服务器，解压缩

* 编译源码

1、切换到解压出来的源码工程目录中

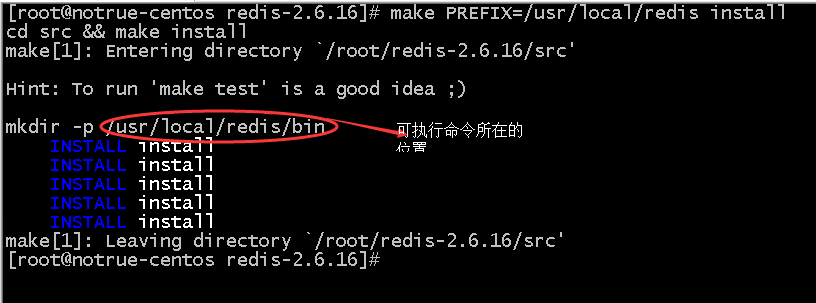
cd redis-2.6.16

2、用make命令来对redis的c语言源码工程进行编译

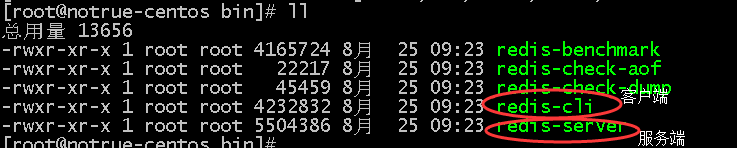
3、编译完成之后，用make install命令进行安装

|  |
| --- |
| [root@notrue-centos redis-2.6.16]# make PREFIX=/usr/local/redis install |

安装成功的显示：



* 启动redis服务
* 进入redis的bin目录



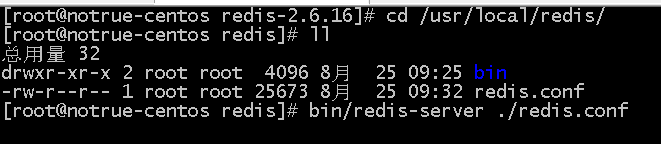
* 2、准备配置文件

Redis服务在启动的时候可以指定配置文件，那，我们可以从redis的源码目录中拷贝一份配置文件模板到redis的安装目录，修改后使用

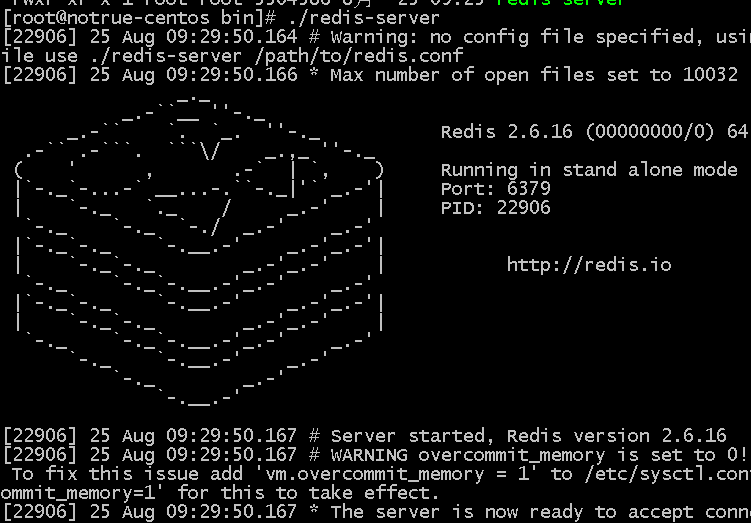
|  |
| --- |
| [root@notrue-centos redis-2.6.16]# cp /root/redis-2.6.16/redis.conf /usr/local/redis/ |

* 3、启动redis服务

并指定使用的配置文件



* 4、启动成功的显示



* 5、启动为后台服务

上述启动方法，会让redis服务进程运行在console前台，最好应该放到后台运行，可将启动命令改为如下方式：

**1/ 方式一**

|  |
| --- |
| [root@notrue-centos redis]# nohup bin/redis-server ./redis.conf 1>/dev/null 2>&1 & |

Nohup：控制台关闭或闲置超时，也不退出

1>/dev/null ： 把程序的“1”——标准输出，重定向到文件/dev/null

2>&1 ： 把程序的“2”——错误输出，重定向到“1”所去的文件

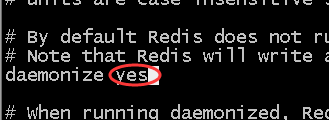
& : 把程序放到后台运行

**2/ 方式二**

修改配置文件，

vi redis.conf

修改其中一个配置



保存文件后再用普通命令启动，也可以启动为后台模式

[root@notrue-centos redis]# bin/redis-server ./redis.conf

### 客户端连接

1. 用redis自带的命令行客户端

|  |
| --- |
| [root@notrue-centos redis]# bin/redis-cli -h notrue-centos -p 6379  redis notrue-centos:6379> ping  PONG  redis notrue-centos:6379> |

1. 或者用redis的api客户端连接

新建一个maven工程，导入jedis的maven依赖坐标

|  |
| --- |
| <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.7.2</version>  <type>jar</type>  <scope>compile</scope>  </dependency> |

然后写一个类用来测试服务器跟客户端的连通性：

|  |
| --- |
| public class RedisClientConnectionTest {  public static void main(String[] args) {  // 构造一个redis的客户端对象  Jedis jedis = new Jedis("pinshutang.zicp.net", 6379);  String ping = jedis.ping();  System.out.println(ping);  }  } |

### 数据类型

#### String数据类型

(常作为缓存使用)

1/插入和读取一条string类型的数据

|  |
| --- |
| redis notrue-centos:6379> set sessionid-0001 "zhangsan"  OK  redis notrue-centos:6379> get sessionid-0001  "zhangsan" |

2/对string类型数据进行增减（前提是这条数据的value可以看成数字）

|  |
| --- |
| DECR key  INCR key  DECRBY key decrement  INCRBY key increment |

3/一次性插入或者获取多条数据

|  |
| --- |
| MGET key1 key2  MSET key1 value1 key2 value2 ….. |

4/在插入一条string类型数据的同时为它指定一个存活期限

|  |
| --- |
| setex bancao 10 weige  ### bancao这条数据就只会存活10秒钟，过期会被redis自动清除 |

应用：将一个自定义的对象比如product存入redis

实现方式一：将对象序列化成byte数组

|  |
| --- |
| /\*\*  \* 将对象缓存到redis的string结构数据中  \* @throws Exception  \*  \*/  @Test  public void testObjectCache() throws Exception{    ProductInfo p = new ProductInfo();    p.setName("苏菲");  p.setDescription("angelababy专用");  p.setCatelog("unknow");  p.setPrice(10.8);  //将对象序列化成字节数组    ByteArrayOutputStream ba = new ByteArrayOutputStream();  ObjectOutputStream oos = new ObjectOutputStream(ba);    //用对象序列化流来将p对象序列化，然后把序列化之后的二进制数据写到ba流中  oos.writeObject(p);    //将ba流转成byte数组  byte[] pBytes = ba.toByteArray();        //将对象序列化之后的byte数组存到redis的string结构数据中  jedis.set("product:01".getBytes(), pBytes);      //根据key从redis中取出对象的byte数据  byte[] pBytesResp = jedis.get("product:01".getBytes());    //将byte数据反序列出对象  ByteArrayInputStream bi = new ByteArrayInputStream(pBytesResp);    ObjectInputStream oi = new ObjectInputStream(bi);    //从对象读取流中读取出p对象  ProductInfo pResp = (ProductInfo) oi.readObject();    System.out.println(pResp);      } |

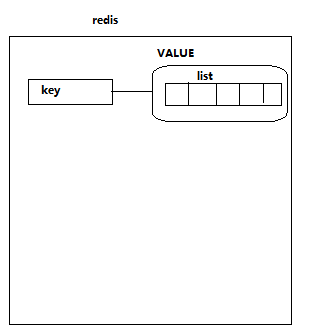
实现方式二：

将对象转成json字符串来存取

|  |
| --- |
| /\*\*  \* 将对象转成json字符串缓存到redis的string结构数据中  \*/  @Test  public void testObjectToJsonCache(){      ProductInfo p = new ProductInfo();    p.setName("ABC");  p.setDescription("刘亦菲专用");  p.setCatelog("夜用型");  p.setPrice(10.8);  //利用gson将对象转成json串  Gson gson = new Gson();  String pJson = gson.toJson(p);    //将json串存入redis  jedis.set("prodcut:02", pJson);      //从redis中取出对象的json串  String pJsonResp = jedis.get("prodcut:02");    //将返回的json解析成对象  ProductInfo pResponse = gson.fromJson(pJsonResp, ProductInfo.class);    //显示对象的属性  System.out.println(pResponse);      } |

#### List数据结构

* List图示



* List功能演示

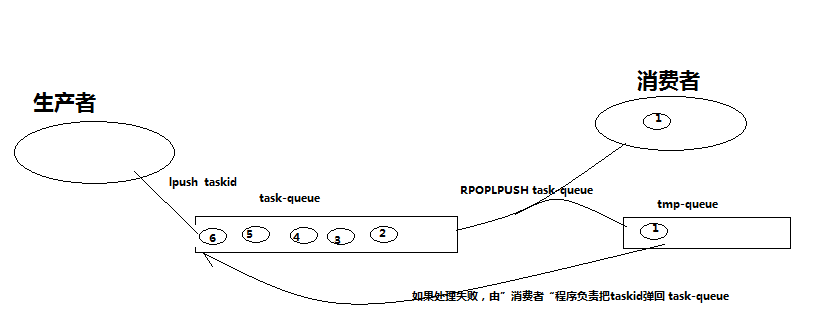
|  |
| --- |
| #从头部（左边）插入数据  redis>LPUSH key value1 value2 value3  #从尾部（右边）插入数据  redis>RPUSH key value1 value2 value3  #读取list中指定范围的values  redis>LRANGE key start end  redis> lrange task-queue 0 -1 读取整个list  #从头部弹出一个元素  LPOP key  #从尾部弹出一个元素  RPOP key  #从一个list的尾部弹出一个元素插入到另一个list  RPOPLPUSH key1 key2 ## 这是一个原子性操作 |

* List的应用案例demo
* 1 需求描述

任务调度系统：

生产者不断产生任务，放入task-queue排队

消费者不断拿出任务来处理，同时放入一个tmp-queue暂存，如果任务处理成功，则清除tmp-queue，否则，将任务弹回task-queue



* 2 代码实现

**1/ 生产者**

——模拟产生任务

|  |
| --- |
| public class TaskProducer {  // 获取一个redis的客户端连接对象  public static Jedis getRedisConnection(String host, int port) {  Jedis jedis = new Jedis(host, port);  return jedis;  }  public static void main(String[] args) {  Jedis jedis = getRedisConnection("angelababy", 6379);  Random random = new Random();  // 生成任务  while (true) {  try {  // 生成任务的速度有一定的随机性，在1-2秒之间  Thread.sleep(random.nextInt(1000) + 1000);  // 生成一个任务  String taskid = UUID.randomUUID().toString();  // 往任务队列"task-queue"中插入，第一次插入时，"task-queue"还不存在  //但是lpush方法会在redis库中创建一条新的list数据  jedis.lpush("task-queue", taskid);  System.out.println("向任务队列中插入了一个新的任务: " + taskid);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  } |

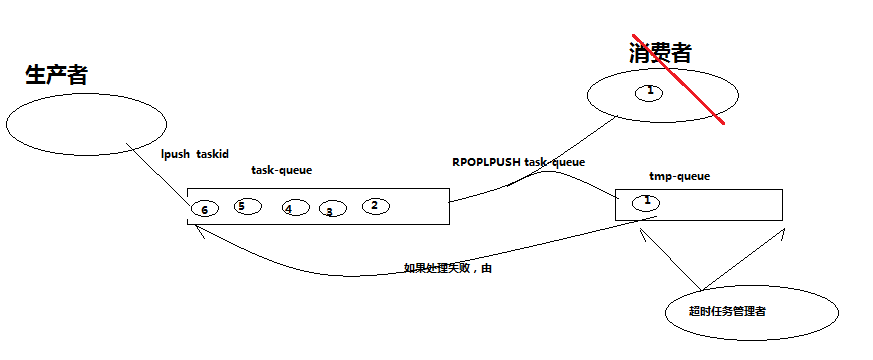
**2/ 消费者**

——模拟处理任务，并且管理暂存队列

|  |
| --- |
| public class TaskConsumer {  public static void main(String[] args) {  Jedis jedis = new Jedis("angelababy", 6379);  Random random = new Random();  while (true) {  try {  // 从task-queue中取一个任务，同时放入"tmp-queue"  String taskid = jedis.rpoplpush("task-queue", "tmp-queue");  // 模拟处理任务  Thread.sleep(1000);  // 模拟有成功又有失败的情况  int nextInt = random.nextInt(13);  if (nextInt % 7 == 0) { // 模拟失败的情况  // 失败的情况下，需要将任务从"tmp-queue"弹回"task-queue"  jedis.rpoplpush("tmp-queue", "task-queue");  System.out.println("-------任务处理失败： " + taskid);  } else { // 模拟成功的情况  // 成功的情况下，将任务从"tmp-queue"清除  jedis.rpop("tmp-queue");  System.out.println("任务处理成功： " + taskid);  }  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  } |

上述机制是一个简化版，真实版的任务调度系统会更加复杂，如下所示：

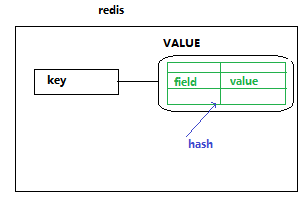
（增加了一个专门用来管理暂存队列的角色，以便就算消费者程序失败退出，那些处理失败的任务依然可以被弹回task-queue）



#### Hash数据结构

* Hash图示

Redis中的Hashes类型可以看成具有String Key和String Value的map容器



* Hash功能演示

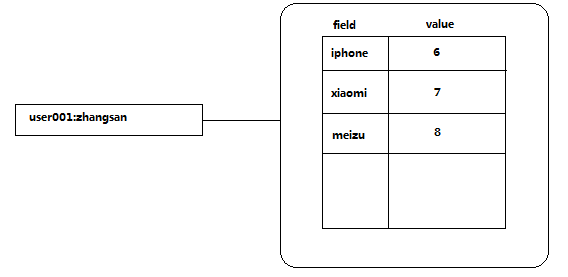
1、往redis库中插入一条hash类型的数据

|  |
| --- |
| redis> HSET key field value |

举例：

|  |
| --- |
| redis 127.0.0.1:6379> hset user001:zhangsan iphone 6  (integer) 1  redis 127.0.0.1:6379> hset user001:zhangsan xiaomi 7  (integer) 1  redis 127.0.0.1:6379> hset user001:zhangsan meizu 8  (integer) 1 |

在redis库中就形成了这样一条数据：



2、从redis库中获取一条hash类型数据的value

* 取出一条hash类型数据中所有field-value对

|  |
| --- |
| redis 127.0.0.1:6379> hgetall user001:zhangsan  1) "iphone"  2) "6"  3) "xiaomi"  4) "7"  5) "meizu"  6) "8" |

* 取出hash数据中所有fields

|  |
| --- |
| redis 127.0.0.1:6379> HKEYS user001:zhangsan  1) "iphone"  2) "xiaomi"  3) "meizu" |

* 取出hash数据中所有的value

|  |
| --- |
| redis 127.0.0.1:6379> hvals user001:zhangsan  1) "6"  2) "7"  3) "8" |

* 取出hash数据中一个指定field的值

|  |
| --- |
| redis 127.0.0.1:6379> hget user001:zhangsan xiaomi  "8" |

* 为hash数据中指定的一个field的值进行增减

|  |
| --- |
| redis 127.0.0.1:6379> HINCRBY user001:zhangsan xiaomi 1  (integer) 8 |

* 从hash数据中删除一个字段field及其值

|  |
| --- |
| redis 127.0.0.1:6379> hgetall user001:zhangsan  1) "iphone"  2) "6"  3) "xiaomi"  4) "7"  5) "meizu"  6) "8"  redis 127.0.0.1:6379> HDEL user001:zhangsan iphone  (integer) 1  redis 127.0.0.1:6379> hgetall user001:zhangsan  1) "xiaomi"  2) "7"  3) "meizu"  4) "8" |

#### Set数据结构

集合的特点：无序、无重复元素

1. 插入一条set数据

|  |
| --- |
| redis 127.0.0.1:6379> sadd frieds:zhangsan bingbing baby fengjie furong ruhua tingting  (integer) 6  redis 127.0.0.1:6379> scard frieds:zhangsan  (integer) 6  redis 127.0.0.1:6379> |

2、获取一条set数据的所有members

|  |
| --- |
| redis 127.0.0.1:6379> smembers frieds:zhangsan  1) "fengjie"  2) "baby"  3) "furong"  4) "bingbing"  5) "tingting"  6) "ruhua" |

3、判断一个成员是否属于某条指定的set数据

|  |
| --- |
| redis 127.0.0.1:6379> sismember frieds:zhangsan liuyifei #如果不是，则返回0  (integer) 0  redis 127.0.0.1:6379> sismember frieds:zhangsan baby #如果是，则返回1  (integer) 1 |

4、求两个set数据的差集

|  |
| --- |
| #求差集  redis 127.0.0.1:6379> sdiff frieds:zhangsan friends:xiaotao  1) "furong"  2) "fengjie"  3) "ruhua"  4) "feifei"  #求差集，并将结果存入到另一个set  redis 127.0.0.1:6379> sdiffstore zhangsan-xiaotao frieds:zhangsan friends:xiaotao  (integer) 4  #查看差集结果  redis 127.0.0.1:6379> smembers zhangsan-xiaotao  1) "furong"  2) "fengjie"  3) "ruhua"  4) "feifei" |

1. 求交集，求并集

|  |
| --- |
| #求交集  redis 127.0.0.1:6379> sinterstore zhangsan:xiaotao frieds:zhangsan friends:xiaotao  (integer) 2  redis 127.0.0.1:6379> smembers zhangsan:xiaotao  1) "bingbing"  2) "baby"  #求并集  redis 127.0.0.1:6379> sunion frieds:zhangsan friends:xiaotao  1) "fengjie"  2) "tangwei"  3) "liuyifei"  4) "bingbing"  5) "ruhua"  6) "feifei"  7) "baby"  8) "songhuiqiao"  9) "furong"  10) "yangmi" |

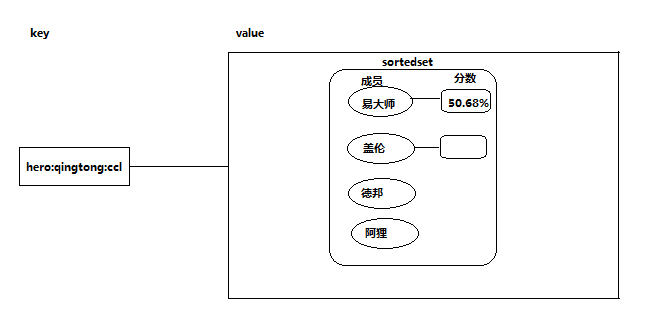
#### SortedSet数据结构

* sortedSet图示

sortedset中存储的成员都有一个附带的分数值

而redis就可以根据分数来对成员进行各种排序（正序、倒序）

1. sortedSet存储内容示意图：



* SortedSet功能演示

1、往redis库中插入一条sortedset数据

|  |
| --- |
| redis 127.0.0.1:6379> zadd nansheng:yanzhi:bang 70 liudehua 90 huangbo 100 weixiaobao 250 yangwei 59 xiaotao  (integer) 5 |

1. 从sortedset中查询有序结果

|  |
| --- |
| #正序结果  redis 127.0.0.1:6379> zrange nanshen:yanzhi:bang 0 4  1) "xiaotao"  2) "liudehua"  3) "huangbo"  4) "weixiaobao"  5) "yangwei"  #倒序结果  redis 127.0.0.1:6379> zrevrange nanshen:yanzhi:bang 0 4  1) "yangwei"  2) "weixiaobao"  3) "huangbo"  4) "liudehua"  5) "xiaotao" |

1. 查询某个成员的名次

|  |
| --- |
| #在正序榜中的名次  redis 127.0.0.1:6379> zrank nanshen:yanzhi:bang xiaotao  (integer) 0  #在倒序榜中的名次  redis 127.0.0.1:6379> zrevrank nanshen:yanzhi:bang xiaotao  (integer) 4 |

4、修改成员的分数

|  |
| --- |
| redis 127.0.0.1:6379> zincrby nanshen:yanzhi:bang 300 xiaotao  "359"  redis 127.0.0.1:6379> zrevrank nanshen:yanzhi:bang xiaotao  (integer) 0 |

### 应用案例一

案例：

Lol盒子英雄数据排行榜：

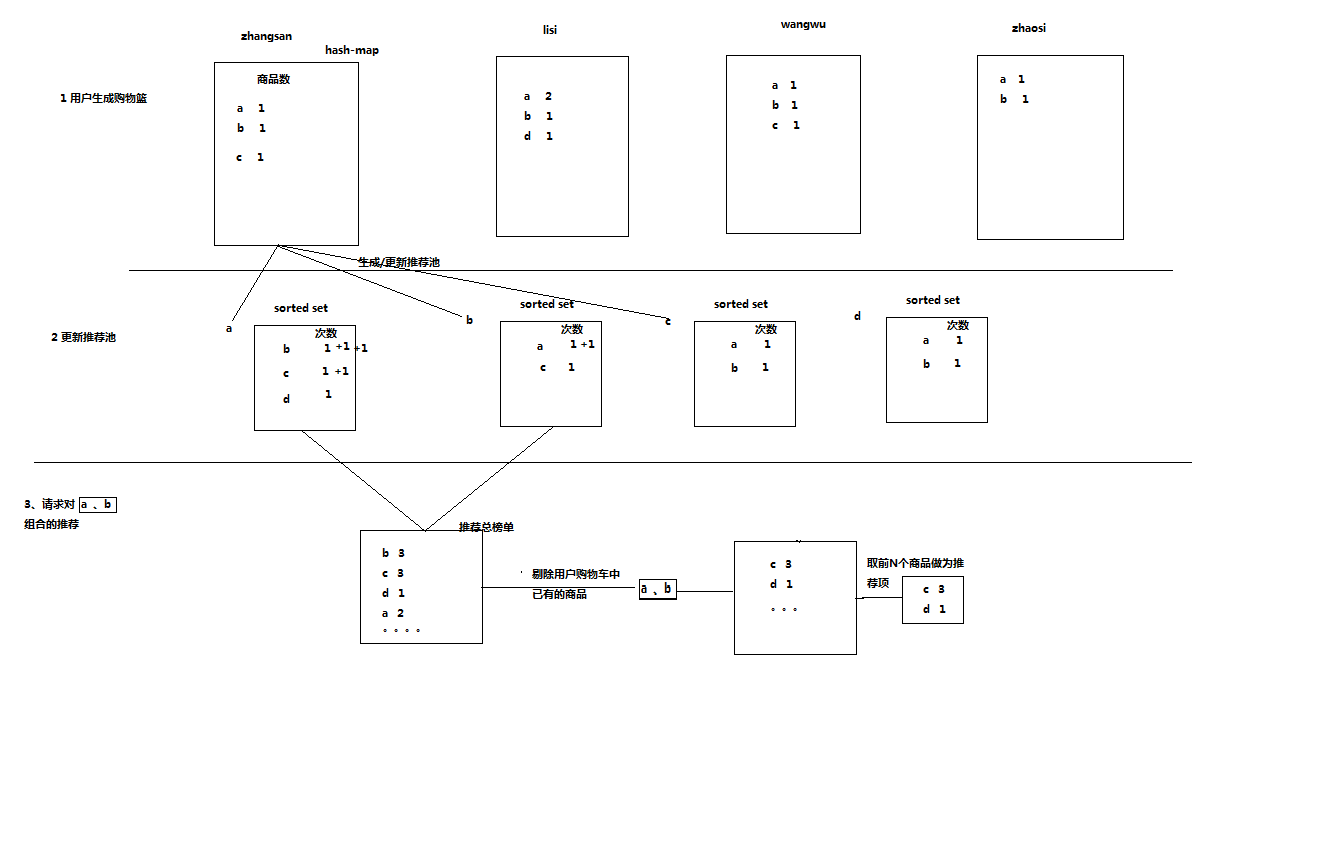
1. 在redis中需要一个榜单所对应的sortedset数据
2. 玩家每选择一个英雄打一场游戏，就对sortedset数据的相应的英雄分数+1
3. Lol盒子上查看榜单时，就调用zrange来看榜单中的排序结果

### 应用案例二

实现步骤：

1. 每来一个用户创建购物车，就对购物车中的每一件商品在redis中插入一条以商品名称为key的sortedset，成员为同时出现在购物车中的其他商品，分数为1
2. 每新产生一个购物车，就对车中的商品更新redis中的sortedset，将同时出现的商品的分数增1
3. 推荐时，用户放入一件商品到购物车，则从这件商品对应的sortedset中查询分数最高的同现商品，推荐给该用户

设计思想如下，详见《代码》



## Java高级特性增强

### 多线程基础知识

* 进程介绍

不管是我们开发的应用程序，还是我们运行的其他的应用程序，都需要先把程序安装在本地的硬盘上。然后找到这个程序的启动文件，启动程序的时候，其实是电脑把当前的这个程序加载到内存中，在内存中需要给当前的程序分配一段独立的运行空间。这片空间就专门负责当前这个程序的运行。

不同的应用程序运行的过程中都需要在内存中分配自己独立的运行空间，彼此之间不会相互的影响。我们把每个独立应用程序在内存的独立空间称为当前应用程序运行的一个进程。

**进程：它是内存中的一段独立的空间，可以负责当前应用程序的运行。当前这个进程负责调度当前程序中的所有运行细节。**

* 线程介绍

启动的QQ聊天软件，需要和多个人进行聊天。这时多个人之间是不能相互影响，但是它们都位于当前QQ这个软件运行时所分配的内容的独立空间中。

在一个进程中，每个独立的功能都需要独立的去运行，这时又需要把当前这个进程划分成多个运行区域，每个独立的小区域（小单元）称为一个线程。

**线程：它是位于进程中，负责当前进程中的某个具备独立运行资格的空间。**

进程是负责整个程序的运行，而线程是程序中具体的某个独立功能的运行。**一个进程中至少应该有一个线程。**

* 多线程介绍

现在的操作系统基本都是多用户，多任务的操作系统。每个任务就是一个进程。而在这个进程中就会有线程。

真正可以完成程序运行和功能的实现靠的是进程中的线程。

多线程：在一个进程中，我们同时开启多个线程，让多个线程同时去完成某些任务（功能）。

多线程的目的：提高程序的运行效率。

* 多线程运行的原理

cpu在线程中做时间片的切换。

其实真正电脑中的程序的运行不是同时在运行的。CPU负责程序的运行，而CPU在运行程序的过程中某个时刻点上，它其实只能运行一个程序。而不是多个程序。而CPU它可以在多个程序之间进行高速的切换。而切换频率和速度太快，导致人的肉看看不到。

每个程序就是进程， 而每个进程中会有多个线程，而CPU是在这些线程之间进行切换。

了解了CPU对一个任务的执行过程，我们就必须知道，多线程可以提高程序的运行效率，但不能无限制的开线程。

* 实现线程的两种方式

1、继承Thread的原理

见代码MyThreadWithExtends

2、声明实现 Runnable 接口的类

见代码MyThreadWithImpliment

3、还可以实现Callable接口

### 同步关键字介绍

* **synchronized**

synchronized是用来实现**线程同步**的！！！

加同步格式：

synchronized( 需要一个任意的**对象**（锁） ){

代码块中放操作共享数据的代码。

}

见代码MySynchronized

* synchronized的缺陷

synchronized是java中的一个关键字，也就是说是Java语言内置的特性。

如果一个代码块被synchronized修饰了，当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，等待获取锁的线程释放锁，而这里获取锁的线程释放锁只会有两种情况：

　　1）获取锁的线程执行完了该代码块，然后线程释放对锁的占有；

2）线程执行发生异常，此时JVM会让线程自动释放锁。

例子1：

　　如果这个获取锁的线程由于要等待IO或者其他原因（比如调用sleep方法）被阻塞了，但是又没有释放锁，其他线程便只能干巴巴地等待，试想一下，这多么影响程序执行效率。

　　因此就需要有一种机制可以不让等待的线程一直无期限地等待下去（比如只等待一定的时间或者能够响应中断），通过Lock就可以办到。

例子2：

当有多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作会发生冲突现象，但是读操作和读操作不会发生冲突现象。

　　但是采用synchronized关键字来实现同步的话，就会导致一个问题：

如果多个线程都只是进行读操作，当一个线程在进行读操作时，其他线程只能等待无法进行读操作。

　　因此就需要一种机制来使得多个线程都只是进行读操作时，线程之间不会发生冲突，通过Lock就可以办到。

　　另外，通过Lock可以知道线程有没有成功获取到锁。这个是synchronized无法办到的。

　　总的来说，也就是说Lock提供了比synchronized更多的功能。

* **lock**
* lock和synchronized的区别

　　1）Lock不是Java语言内置的，synchronized是Java语言的关键字，因此是内置特性。Lock是一个类，通过这个类可以实现同步访问；

　　2）Lock和synchronized有一点非常大的不同，采用**synchronized不需要用户去手动释放锁**，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而**Lock则必须要用户去手动释放锁**，如果没有主动释放锁，就有可能导致出现死锁现象。

* java.util.concurrent.locks包下常用的类
  + - * + Lock

　　首先要说明的就是Lock，通过查看Lock的源码可知，Lock是一个接口：

|  |
| --- |
| public interface Lock {  void lock();  void lockInterruptibly() throws InterruptedException;  boolean tryLock();  boolean tryLock(long time, TimeUnit unit) throws InterruptedException;  void unlock();  } |

Lock接口中每个方法的使用：

lock()、tryLock()、tryLock(long time, TimeUnit unit)、lockInterruptibly()是用来获取锁的。 unLock()方法是用来释放锁的。

**四个获取锁方法的区别：**

　　lock()方法是平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。

由于在前面讲到如果采用Lock，必须主动去释放锁，并且在发生异常时，不会自动释放锁。因此一般来说，使用Lock必须在try{}catch{}块中进行，并且将释放锁的操作放在finally块中进行，以保证锁一定被被释放，防止死锁的发生。

tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法**无论如何都会立即返回。**在拿不到锁时不会一直在那等待。

　　tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

　　lockInterruptibly()方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。

　　注意，当一个线程获取了锁之后，是不会被interrupt()方法中断的。

　　因此当通过lockInterruptibly()方法获取某个锁时，如果不能获取到，只有进行等待的情况下，是可以响应中断的。

　　而用synchronized修饰的话，当一个线程处于等待某个锁的状态，是无法被中断的，只有一直等待下去。

* + ReentrantLock

直接使用lock接口的话，我们需要实现很多方法，不太方便，ReentrantLock是唯一实现了Lock接口的类，并且ReentrantLock提供了更多的方法，ReentrantLock，意思是“可重入锁”。

以下是ReentrantLock的使用案例：

　　例子1，lock()的正确使用方法

见代码MyLockTest

例子2，tryLock()的使用方法

见代码MyTryLock

例子3，lockInterruptibly()响应中断的使用方法：

见代码MyInterruptibly

* ReadWriteLock (可以区别对待读、写的操作)

　　ReadWriteLock也是一个接口，在它里面只定义了两个方法：

|  |
| --- |
| public interface ReadWriteLock {  /\*\*  \* Returns the lock used for reading.  \*  \* @return the lock used for reading.  \*/  Lock readLock();    /\*\*  \* Returns the lock used for writing.  \*  \* @return the lock used for writing.  \*/  Lock writeLock();  } |

　　一个用来获取读锁，一个用来获取写锁。**也就是说将文件的读写操作分开，分成2个锁来分配给线程，从而使得多个线程可以同时进行读操作。**下面的ReentrantReadWriteLock实现了ReadWriteLock接口。

* ReentrantReadWriteLock

　　ReentrantReadWriteLock里面提供了很多丰富的方法，不过最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁。

下面通过几个例子来看一下ReentrantReadWriteLock具体用法。

例子1：　　假如有多个线程要同时进行读操作的话，先看一下synchronized达到的效果

见代码MySynchronizedReadWrite

例子2：改成用读写锁的话：

见代码MyReentrantReadWriteLock

注意：

　　不过要注意的是，如果有一个线程已经占用了读锁，则此时其他线程如果要申请写锁，则申请写锁的线程会一直等待释放读锁。

如果有一个线程已经占用了写锁，则此时其他线程如果申请写锁或者读锁，则申请的线程会一直等待释放写锁。

* Lock和synchronized的选择

　　1）Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；

　　2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

　　3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

　　4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

　　5）Lock可以提高多个线程进行读操作的效率。

　　在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。所以说，在具体使用时要根据适当情况选择。

### Java并发包

JDK5.0 以后的版本都引入了高级并发特性，大多数的特性在java.util.concurrent 包中，是专门用于多线程发编程的，充分利用了现代多处理器和多核心系统的功能以编写大规模并发应用程序。主要包含原子量、并发集合、同步器、可重入锁，并对线程池的构造提供

了强力的支持。

**线程池**

* 线程池的5中创建方式：
* Single Thread Executor : 只有一个线程的线程池，因此所有提交的任务是顺序执行，

代码： Executors.newSingleThreadExecutor()

* Cached Thread Pool : 线程池里有很多线程需要同时执行，老的可用线程将被新的任务触发重新执行，如果线程超过60秒内没执行，那么将被终止并从池中删除，

代码：Executors.newCachedThreadPool()

* Fixed Thread Pool : 拥有固定线程数的线程池，如果没有任务执行，那么线程会一直等待，

代码： Executors.newFixedThreadPool(4)

在构造函数中的参数4是线程池的大小，你可以随意设置，也可以和cpu的核数量(线程数量)保持一致，获取cpu的数量int cpuNums = Runtime.getRuntime().availableProcessors();

* Scheduled Thread Pool : 用来调度即将执行的任务的线程池，

代码：Executors.newScheduledThreadPool()

* Single Thread Scheduled Pool : 只有一个线程，用来调度执行将来的任务，代码：Executors.newSingleThreadScheduledExecutor()
* 线程池的使用

所谓给线程池提交任务，就是：

1、你将任务（业务处理逻辑）写到一个runnable或者callable的执行方法<run() | call()>

2、将这个runnable对象提交给线程池即可

提交 Runnable ，任务完成后 Future 对象返回 null

见代码：ThreadPoolWithRunable

提交 Callable，该方法返回一个 Future 实例表示任务的状态

见代码：ThreadPoolWithcallable

### 并发包在软件中的应用

消息队列常用于有生产者和消费者两类角色的多线程同步场景

BlockingQueue也是java.util.concurrent下的主要用来控制线程同步的工具。

主要的方法是：put、take一对阻塞存取；add、poll一对非阻塞存取。

插入:

1)add(anObject):把anObject加到BlockingQueue里,即如果BlockingQueue可以容纳,则返回true,否则抛出异常

2)offer(anObject):表示如果可能的话,将anObject加到BlockingQueue里,即如果BlockingQueue可以容纳,则返回true,否则返回false.

3)put(anObject):把anObject加到BlockingQueue里,如果BlockQueue没有空间,则调用此方法的线程被阻塞直到BlockingQueue里面有空间再继续.

读取：

4)poll(time):取走BlockingQueue里排在首位的对象,若不能立即取出,则可以等time参数规定的时间,取不到时返回null

5)take():取走BlockingQueue里排在首位的对象,若BlockingQueue为空,阻断进入等待状态直到Blocking有新的对象被加入为止

其他

int remainingCapacity();返回队列剩余的容量，在队列插入和获取的时候，不要瞎搞，数 据可能不准

boolean remove(Object o); 从队列移除元素，如果存在，即移除一个或者更多，队列改 变了返回true

public boolean contains(Object o); 查看队列是否存在这个元素，存在返回true

int drainTo(Collection<? super E> c); 传入的集合中的元素，如果在队列中存在，那么将 队列中的元素移动到集合中

int drainTo(Collection<? super E> c, int maxElements); 和上面方法的区别在于，制定了移 动的数量

BlockingQueue有四个具体的实现类,常用的两种实现类为：

1、ArrayBlockingQueue：一个由**数组支持的有界阻塞队列**，规定大小的BlockingQueue,其构造函数必须带一个int参数来指明其大小.其所含的对象是以FIFO(先入先出)顺序排序的。

2、LinkedBlockingQueue：大小不定的BlockingQueue,若其构造函数带一个规定大小的参数,生成的BlockingQueue有大小限制,若不带大小参数,所生成的BlockingQueue的大小由Integer.MAX\_VALUE来决定.其所含的对象是以FIFO(先入先出)顺序排序的。

LinkedBlockingQueue **可以指定容量，也可以不指定**，不指定的话，默认最大是Integer.MAX\_VALUE,其中主要用到put和take方法，put方法在队列满的时候会阻塞直到有队列成员被消费，take方法在队列空的时候会阻塞，直到有队列成员被放进来。

LinkedBlockingQueue和ArrayBlockingQueue区别：

LinkedBlockingQueue和ArrayBlockingQueue比较起来,它们背后所用的数据结构不一样,导致LinkedBlockingQueue的数据吞吐量要大于ArrayBlockingQueue,但在线程数量很大时其性能的可预见性低于ArrayBlockingQueue.

生产者消费者的示例代码：

见代码 TestBlockingQueue TestBlockingQueueConsumer TestBlockingQueueProducer

### JMS技术

* **什么是JMS**

JMS即Java消息服务（Java Message Service）应用程序**接口**是一个Java平台中关于**面向消息中间件（MOM）的API**，**用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信**。Java消息服务是一个与具体平台无关的API，绝大多数MOM提供商都对JMS提供支持。

**JMS是一种与厂商无关的 API**，用来访问消息收发系统消息。它类似于JDBC(Java Database Connectivity)：这里，JDBC 是可以用来访问许多不同关系数据库的 API，而 JMS 则提供同样与厂商无关的访问方法，以访问消息收发服务。许多厂商都支持 JMS，包括 IBM 的 MQSeries、BEA的 Weblogic JMS service和 Progress 的 SonicMQ，这只是几个例子。 JMS 使您能够通过消息收发服务（有时称为消息中介程序或路由器）从一个 JMS 客户机向另一个 JMS客户机发送消息。**消息是 JMS 中的一种类型对象，由两部分组成：报头和消息主体**。报头由路由信息以及有关该消息的元数据组成。消息主体则携带着应用程序的数据或有效负载。根据有效负载的类型来划分，可以将消息分为几种类型，它们分别携带：简单文本(TextMessage)、可序列化的对象 (ObjectMessage)、属性集合 (MapMessage)、字节流 (BytesMessage)、原始值流 (StreamMessage)，还有无有效负载的消息 (Message)。

* **2.JMS规范**
* 专业技术规范

JMS（Java Messaging Service）**是Java平台上有关面向消息中间件(MOM)的技术规范**，它便于消息系统中的Java应用程序进行消息交换,并且通过提供标准的产生、发送、接收消息的接口简化企业应用的开发，翻译为Java消息服务。

* 体系架构

JMS由以下元素组成。

**JMS提供者：**连接面向消息中间件的，**JMS接口的一个实现**。提供者可以是Java平台的JMS实现，也可以是非Java平台的面向消息中间件的适配器。

**JMS客户**：生产或消费基于消息的Java的应用程序或对象。

**JMS生产者**：创建并发送消息的JMS客户。

**JMS消费者**：接收消息的JMS客户。

**JMS消息**：包括可以在JMS客户之间传递的数据的对象

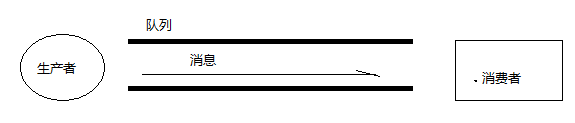
**JMS队列**：一个容纳那些被发送的等待阅读的消息的区域。一旦一个消息被阅读，该消息将被从队列中移走。

**JMS主题**：一种支持发送消息给多个订阅者的机制。

* Java消息服务应用程序结构支持两种模型

1. 点对点或**队列模型**

在点对点或队列模型下，一个生产者向一个特定的队列发布消息，一个消费者从该队列中读取消息。这里，生产者知道消费者的队列，并直接将消息发送到消费者的队列。



这种模式被概括为：

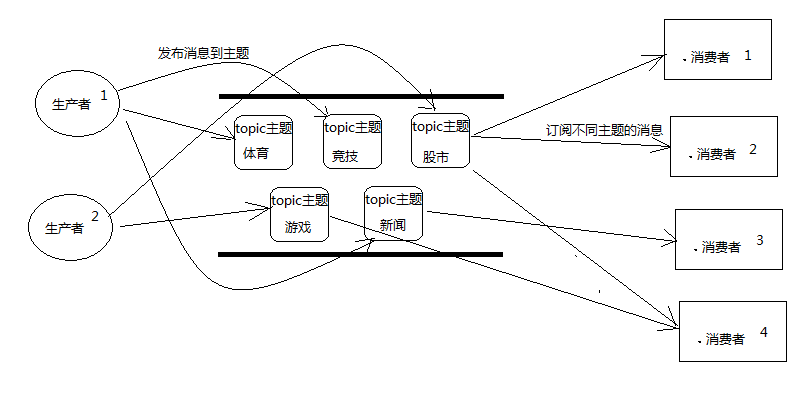
只有一个消费者将获得消息

生产者不需要在接收者消费该消息期间处于运行状态，接收者也同样不需要在消息发送时处于运行状态。

每一个成功处理的消息都由接收者签收

2、**发布者/订阅者模型**

发布者/订阅者模型支持向一个特定的消息主题发布消息。0或多个订阅者可能对接收来自特定消息主题的消息感兴趣。在这种模型下，发布者和订阅者彼此不知道对方。这种模式好比是匿名公告板。

这种模式被概括为：

多个消费者可以获得消息

在发布者和订阅者之间存在时间依赖性。发布者需要建立一个订阅（subscription），以便客户能够订阅。订阅者必须保持持续的活动状态以接收消息，除非订阅者建立了持久的订阅。在那种情况下，在订阅者未连接时发布的消息将在订阅者重新连接时重新发布。

* 常用的JMS实现

要使用Java消息服务，你必须要有一个JMS提供者，管理会话和队列。既有开源的提供者也有专有的提供者。

* 开源的提供者包括：

**Apache ActiveMQ**

JBoss 社区所研发的 HornetQ

Joram

Coridan的MantaRay

The OpenJMS Group的OpenJMS

* 专有的提供者包括：

BEA的BEA WebLogic Server JMS

TIBCO Software的EMS

GigaSpaces Technologies的GigaSpaces

Softwired 2006的iBus

IONA Technologies的IONA JMS

SeeBeyond的IQManager（2005年8月被Sun Microsystems并购）

webMethods的JMS+ -

my-channels的Nirvana

Sonic Software的SonicMQ

SwiftMQ的SwiftMQ

IBM的WebSphere MQ

* 代码演示

*注：新版直接解压即可运行*

1.下载ActiveMQ

去官方网站下载：http://activemq.apache.org/

2.运行ActiveMQ

解压缩apache-activemq-5.5.1-bin.zip，

~~修改配置文件activeMQ.xml，将0.0.0.0修改为localhost~~

|  |
| --- |
| <transportConnectors>  <transportConnector name="openwire" uri="tcp://localhost:61616"/>  <transportConnector name="ssl" uri="ssl://localhost:61617"/>  <transportConnector name="stomp" uri="stomp://localhost:61613"/>  <transportConnector uri="http://localhost:8081"/>  <transportConnector uri="udp://localhost:61618"/> |

然后双击apache-activemq-5.5.1\bin\win64\activemq.bat运行ActiveMQ程序。

启动ActiveMQ以后，登陆：http://localhost:8161/admin/，创建一个Queue，命名为FirstQueue。

3.运行代码

package cn.itcast\_03\_mq.queue

package cn.itcast\_03\_mq.topic

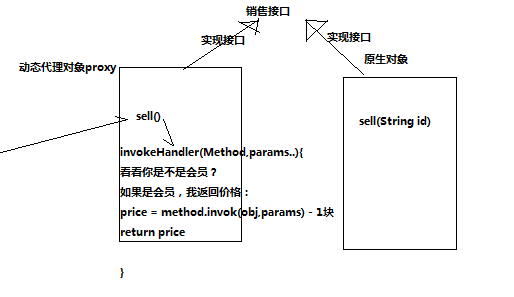
### Java反射.动态代理.socket通信

#### 反射

通过反射的方式可以获取class对象中的属性、方法、构造函数等，一下是实例：

|  |
| --- |
| package cn.java.reflect;  import java.lang.reflect.Constructor;  import java.lang.reflect.Field;  import java.lang.reflect.Method;  import java.util.ArrayList;  import java.util.List;  import org.junit.Before;  import org.junit.Test;  public class MyReflect {  public String className = null;  @SuppressWarnings("rawtypes")  public Class personClass = null;  /\*\*  \* 反射Person类  \* @throws Exception  \*/  @Before  public void init() throws Exception {  className = "cn.java.reflect.Person";  personClass = Class.forName(className);  }  /\*\*  \*获取某个class文件对象  \*/  @Test  public void getClassName() throws Exception {  System.out.println(personClass);  }  /\*\*  \*获取某个class文件对象的另一种方式  \*/  @Test  public void getClassName2() throws Exception {  System.out.println(Person.class);  }  /\*\*  \*创建一个class文件表示的真实对象，底层会调用空参数的构造方法  \*/  @Test  public void getNewInstance() throws Exception {  System.out.println(personClass.newInstance());  }  /\*\*  \*获取非私有的构造函数  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getPublicConstructor() throws Exception {  Constructor constructor = personClass.getConstructor(Long.class,String.class);  Person person = (Person)constructor.newInstance(100L,"zhangsan");  System.out.println(person.getId());  System.out.println(person.getName());  }  /\*\*  \*获得私有的构造函数  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getPrivateConstructor() throws Exception {  Constructor con = personClass.getDeclaredConstructor(String.class);  con.setAccessible(true);//强制取消Java的权限检测  Person person2 = (Person)con.newInstance("zhangsan");  System.out.println(person2.getName());  }  /\*\*  \*获取非私有的成员变量  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getNotPrivateField() throws Exception {  Constructor constructor = personClass.getConstructor(Long.class,String.class);  Object obj = constructor.newInstance(100L,"zhangsan");    Field field = personClass.getField("name");  field.set(obj, "lisi");  System.out.println(field.get(obj));  }  /\*\*  \*获取私有的成员变量  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getPrivateField() throws Exception {  Constructor constructor = personClass.getConstructor(Long.class);  Object obj = constructor.newInstance(100L);    Field field2 = personClass.getDeclaredField("id");  field2.setAccessible(true);//强制取消Java的权限检测  field2.set(obj,10000L);  System.out.println(field2.get(obj));  }  /\*\*  \*获取非私有的成员函数  \*/  @SuppressWarnings({ "unchecked" })  @Test  public void getNotPrivateMethod() throws Exception {  System.out.println(personClass.getMethod("toString"));    Object obj = personClass.newInstance();//获取空参的构造函数  Object object = personClass.getMethod("toString").invoke(obj);  System.out.println(object);  }  /\*\*  \*获取私有的成员函数  \*/  @SuppressWarnings("unchecked")  @Test  public void getPrivateMethod() throws Exception {  Object obj = personClass.newInstance();//获取空参的构造函数  Method method = personClass.getDeclaredMethod("getSomeThing");  method.setAccessible(true);  Object value = method.invoke(obj);  System.out.println(value);  }  /\*\*  \*  \*/  @Test  public void otherMethod() throws Exception {  //当前加载这个class文件的那个类加载器对象  System.out.println(personClass.getClassLoader());  //获取某个类实现的所有接口  Class[] interfaces = personClass.getInterfaces();  for (Class class1 : interfaces) {  System.out.println(class1);  }  //反射当前这个类的直接父类  System.out.println(personClass.getGenericSuperclass());  /\*\*  \* getResourceAsStream这个方法可以获取到一个输入流，这个输入流会关联到name所表示的那个文件上。  \*/  //path 不以’/'开头时默认是从此类所在的包下取资源，以’/'开头则是从ClassPath根下获取。其只是通过path构造一个绝对路径，最终还是由ClassLoader获取资源。  System.out.println(personClass.getResourceAsStream("/log4j.properties"));  //默认则是从ClassPath根下获取，path不能以’/'开头，最终是由ClassLoader获取资源。  System.out.println(personClass.getResourceAsStream("/log4j.properties"));    //判断当前的Class对象表示是否是数组  System.out.println(personClass.isArray());  System.out.println(new String[3].getClass().isArray());    //判断当前的Class对象表示是否是枚举类  System.out.println(personClass.isEnum());  System.out.println(Class.forName("cn.java.reflect.City").isEnum());    //判断当前的Class对象表示是否是接口  System.out.println(personClass.isInterface());  System.out.println(Class.forName("cn.java.reflect.TestInterface").isInterface());      }  } |

#### 动态代理



在之前的代码调用阶段，我们用action调用service的方法实现业务即可。

由于之前在service中实现的业务可能不能够满足当先客户的要求，需要我们重新修改service中的方法，但是service的方法不只在我们这个模块使用，在其他模块也在调用，其他模块调用的时候，现有的service方法已经能够满足业务需求，所以我们不能只为了我们的业务而修改service，导致其他模块授影响。

那怎么办呢？

可以通过动态代理的方式，扩展我们的service中的方法实现，使得在原油的方法中增加更多的业务，而不是实际修改service中的方法，这种实现技术就叫做动态代理。

动态代理：*在不修改原业务的基础上，基于原业务方法，进行重新的扩展，实现新的业务。*

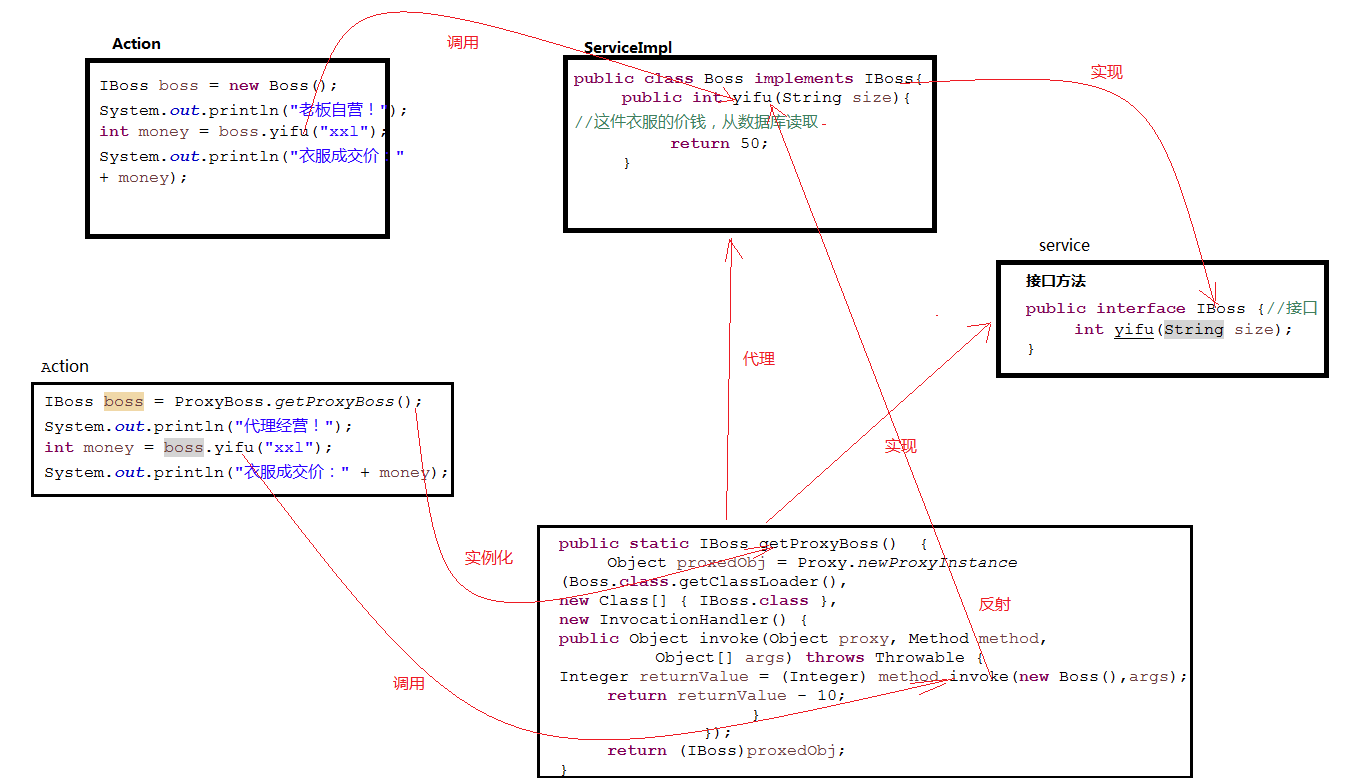
例如下面的例子：

1. 旧业务

买家调用action，购买衣服，衣服在数据库的标价为50元，购买流程就是简单的调用。

1. 新业务

在原先的价格上可以使用优惠券，但是这个功能在以前没有实现过，我们通过代理类，代理了原先的接口方法，在这个方法的基础上，修改了返回值。



代理实现流程：

1. 书写代理类和代理方法，在代理方法中实现代理Proxy.newProxyInstance
2. 代理中需要的参数分别为：被代理的类的类加载器soneObjectclass.getClassLoader()，被代理类的所有实现接口new Class[] { Interface.class }，句柄方法new InvocationHandler()
3. 在句柄方法中复写invoke方法，invoke方法的输入有3个参数Object proxy（代理类对象）, Method method（被代理类的方法）,Object[] args（被代理类方法的传入参数），在这个方法中，我们可以定制化的开发新的业务。
4. 获取代理类，强转成被代理的接口
5. 最后，我们可以像没被代理一样，调用接口的认可方法，方法被调用后，方法名和参数列表将被传入代理类的invoke方法中，进行新业务的逻辑流程。

原业务接口IBoss

|  |
| --- |
| public interface IBoss {//接口  int yifu(String size);  } |

原业务实现类

|  |
| --- |
| public class Boss implements IBoss{  public int yifu(String size){  System.err.println("天猫小强旗舰店，老板给客户发快递----衣服型号："+size);  //这件衣服的价钱，从数据库读取  return 50;  }  public void kuzi(){  System.err.println("天猫小强旗舰店，老板给客户发快递----裤子");  }  } |

原业务调用

|  |
| --- |
| public class SaleAction {  @Test  public void saleByBossSelf() throws Exception {  IBoss boss = new Boss();  System.out.println("老板自营！");  int money = boss.yifu("xxl");  System.out.println("衣服成交价：" + money);  }  } |

代理类

|  |
| --- |
| public static IBoss getProxyBoss(final int discountCoupon) throws Exception {  Object proxedObj = Proxy.newProxyInstance(Boss.class.getClassLoader(),  new Class[] { IBoss.class }, new InvocationHandler() {  public Object invoke(Object proxy, Method method,  Object[] args) throws Throwable {  Integer returnValue = (Integer) method.invoke(new Boss(),  args);// 调用原始对象以后返回的值  return returnValue - discountCoupon;  }  });  return (IBoss)proxedObj;  }  } |

新业务调用

|  |
| --- |
| public class ProxySaleAction {  @Test  public void saleByProxy() throws Exception {  IBoss boss = ProxyBoss.getProxyBoss(20);// 将代理的方法实例化成接口  System.out.println("代理经营！");  int money = boss.yifu("xxl");// 调用接口的方法，实际上调用方式没有变  System.out.println("衣服成交价：" + money);  }  } |

#### socket编程

注意：此处讲的socket编程模型是jdk1.4之前的机制BIO (同步阻塞IO)

1.4之后引入了NIO （同步非阻塞IO）

1.7之后引入了AIO （NIO 2） （异步非阻塞IO）

基本编程模型：

服务端

核心API ServerSocket

流程： 先创建一个服务，然后绑定在服务器的IP地址和端口

等待客户端的连接请求

收到连接请求后，接受请求，建立了一个TCP连接

从建立的连接中获取到socket输入、输出流（两个流都是同步阻塞的）

通过两个流进行数据的交互

客户端

核心API Socket

流程： 先向服务端请求连接

一旦被服务器接受，连接就创建好了

从tcp连接中获取socket输入、输出流

通过两个流进行数据的交互

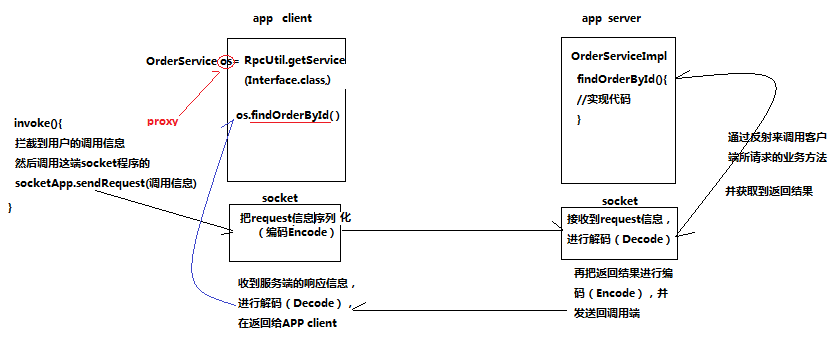
## 轻量级RPC框架开发

### RPC原理

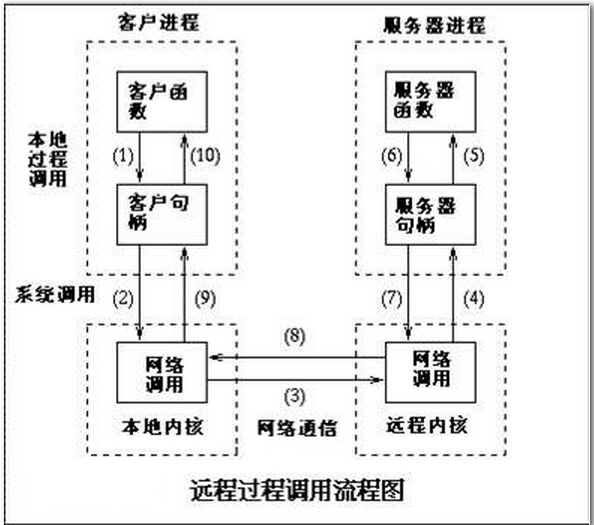
* 什么是RPC

RPC（Remote Procedure Call Protocol）——远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。

RPC采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行。



* RPC原理



运行时,一次客户机对服务器的RPC调用,其内部操作大致有如下十步：

1.调用客户端句柄；执行传送参数

2.调用本地系统内核发送网络消息

3.消息传送到远程主机

4.服务器句柄得到消息并取得参数

5.执行远程过程

6.执行的过程将结果返回服务器句柄

7.服务器句柄返回结果，调用远程系统内核

8.消息传回本地主机

9.客户句柄由内核接收消息

10.客户接收句柄返回的数据

* hadoopRPC演示

见代码

### NIO原理

* 简介

nio 是New IO 的简称，在jdk1.4 里提供的新api 。

Sun 官方标榜的特性如下：

为所有的原始类型提供(Buffer)缓存支持。

字符集编码解码解决方案。

Channel ：一个新的原始I/O 抽象。

支持锁和内存映射文件的文件访问接口。

提供多路(non-bloking) **非阻塞式**的高伸缩性网络I/O 。

* 传统socket和socket nio代码

基本编程模型：

服务端

核心API ServerSocket

流程： 先创建一个服务，然后绑定在服务器的IP地址和端口

等待客户端的连接请求

收到连接请求后，接受请求，建立了一个TCP连接

从建立的连接中获取到socket输入、输出流（两个流都是同步阻塞的）

通过两个流进行数据的交互

客户端

核心API Socket

流程： 先向服务端请求连接

一旦被服务器接受，连接就创建好了

从tcp连接中获取socket输入、输出流

通过两个流进行数据的交互

见代码

* socket /nio原理

**1）阻塞和非阻塞：**

阻塞和非阻塞是进程在**访问数据**的时候，数据是否准备就绪的一种处理方式。

当数据没有准备好的时候，

阻塞：往往需要等待缓冲区中的数据准备好之后才处理，否则一直等待。

非阻塞：当我们的进程访问我们的数据缓冲区的时候，数据没有准备好的时候，直接返回，不需要等待。有数据的时候，也直接返回

**2）同步和异步**

同步和异步都是基于应用程序和操作系统**处理IO事件**所采用的方式:

同步：应用程序要直接参与IO事件的操作；

异步：所有的IO读写事件交给操作系统去处理；

同步的方式在处理IO事件的时候，必须阻塞在某个方法上面等待我们的IO事件完成（阻塞在IO事件或者通过轮询IO事件的方式）；对于异步来说，所有的IO读写都交给了操作系统，这个时候，我们可以去做其他的事情，并不需要去完成真正的IO操作。当操作系统完成IO之后，给我们的应用程序一个通知就可以了。

同步有两种实现模式：

1）阻塞到IO事件 阻塞到read 或者 write 方法上，这个时候我们就完全不能做自己的事情。（在这种情况下，我们只能把读写方法放置到线程中，然后阻塞线程的方式来实现并发服务，对线程的性能开销比较大）

2）IO事件的轮询 --在linux c语言编程中叫做多路复用技术（select模式）

读写事件交给一个专门的线程来处理，这个线程完成IO事件的注册功能，还有就是不断地去轮询我们的读写缓冲区（操作系统），看是否有数据准备好，然后通知我们的相应的业务处理线程。这样的话，我们的业务处理线程就可以做其他的事情。在这种模式下，阻塞的不是所有的IO线程，而是**阻塞的只是select线程**

比喻说明：

Client Selector 管家 BOSS

当客人来的时候，就给管家说，我来了（注册），管家得到这个注册信息后，就给BOSS说，我这里有一个或者多个客人。BOSS就说你去给某人A这件东西（IO数据），给另外一个人B另一件东西。这个时候，客人是可以去做自己的事情（比如看看花园等等），当管家知道BOSS给他任务后，他就会去找对应的某人（根据客人的注册信息），告诉他BOSS给他了某样东西。

JAVA IO模型

基于以上4中IO模型，JAVA对应的实现有：

BIO--同步阻塞： JDK1.4以前我们使用的都是BIO

阻塞到我们的读写方法，阻塞到线程来提高并发性能，但是效果不是很好

NIO--同步非阻塞：JDK1.4 linux多路复用技术（select模式） 实现IO事件的轮询方式：同步非阻塞的模式，**这种方式目前是主流的网络通信模式**

mina netty ——网络通信框架，比自己写NIO要容易些，并且代码可读性更好

AIO：JDK1.7（NIO2）真正的异步非阻塞IO(基于linux的epoll模式）

AIO目前使用的还比较少

小结：1）BIO阻塞的IO

2）NIO select多路复用+非阻塞 同步非阻塞

3）AIO异步非阻塞IO

3、NIO原理

通过selector（选择器），管理所有的IO事件：

客户端的connection事件

服务端的accept事件

客户端和服务端的读写事件

selector如何进行事件管理？

当IO事件注册给我们的选择器的时候，选择器会给他们分配一个key（可以简单的理解成一个事件的标签）

当IO事件就绪后，可以通过key值来找到相应的管道channel，然后通过管道发送数据和接收数据等操作

数据缓冲区：

通过bytebuffer来实现，提供了很多读写的方法 put() get()

* 2.1.4 NIO示例代码

*<见代码工程>*

### Netty常用API学习

* netty简介

Netty是基于Java NIO的网络通信框架.

Netty是一个NIO client-server(客户端服务器)框架，使用Netty可以快速开发网络应用，例如服务器和客户端协议。Netty提供了一种新的方式来使开发网络应用程序，这种新的方式使得它很容易使用和有很强的扩展性。Netty的内部实现时很复杂的，但是Netty提供了简单易用的api从网络处理代码中解耦业务逻辑。Netty是完全基于NIO实现的，所以整个Netty都是异步的。

网络应用程序通常需要有较高的可扩展性，无论是Netty还是其他的基于Java NIO的框架，都会提供可扩展性的解决方案。Netty中一个关键组成部分是它的异步特性.

* netty的helloworld
* 下载netty包

• 下载netty包，下载地址<http://netty.io/>

* 服务端启动类

|  |
| --- |
| package com.netty.demo.server;  import io.netty.bootstrap.ServerBootstrap;  import io.netty.channel.Channel;  import io.netty.channel.ChannelFuture;  import io.netty.channel.ChannelInitializer;  import io.netty.channel.EventLoopGroup;  import io.netty.channel.nio.NioEventLoopGroup;  import io.netty.channel.socket.nio.NioServerSocketChannel;  /\*\*  \* • 配置服务器功能，如线程、端口 • 实现服务器处理程序，它包含业务逻辑，决定当有一个请求连接或接收数据时该做什么  \*  \* @author wilson  \*  \*/  public class EchoServer {  private final int port;  public EchoServer(int port) {  this.port = port;  }  public void start() throws Exception {  EventLoopGroup eventLoopGroup = null;  try {  //创建ServerBootstrap实例来引导绑定和启动服务器  ServerBootstrap serverBootstrap = new ServerBootstrap();  //创建NioEventLoopGroup对象来处理事件，如接受新连接、接收数据、写数据等等  eventLoopGroup = new NioEventLoopGroup();  //指定通道类型为NioServerSocketChannel，设置InetSocketAddress让服务器监听某个端口已等待客户端连接。  serverBootstrap.group(eventLoopGroup).channel(NioServerSocketChannel.class).localAddress("localhost",port).childHandler(new ChannelInitializer<Channel>() {  //设置childHandler执行所有的连接请求  @Override  protected void initChannel(Channel ch) throws Exception {  ch.pipeline().addLast(new EchoServerHandler());  }  });  // 最后绑定服务器等待直到绑定完成，调用sync()方法会阻塞直到服务器完成绑定,然后服务器等待通道关闭，因为使用sync()，所以关闭操作也会被阻塞。  ChannelFuture channelFuture = serverBootstrap.bind().sync();  System.out.println("开始监听，端口为：" + channelFuture.channel().localAddress());  channelFuture.channel().closeFuture().sync();  } finally {  eventLoopGroup.shutdownGracefully().sync();  }  }  public static void main(String[] args) throws Exception {  new EchoServer(20000).start();  }  } |

* 服务端回调方法

|  |
| --- |
| package com.netty.demo.server;  import io.netty.buffer.ByteBuf;  import io.netty.buffer.Unpooled;  import io.netty.channel.ChannelFutureListener;  import io.netty.channel.ChannelHandlerContext;  import io.netty.channel.ChannelInboundHandlerAdapter;  import java.util.Date;  public class EchoServerHandler extends ChannelInboundHandlerAdapter {  @Override  public void channelRead(ChannelHandlerContext ctx, Object msg)  throws Exception {  System.out.println("server 读取数据……");  //读取数据  ByteBuf buf = (ByteBuf) msg;  byte[] req = new byte[buf.readableBytes()];  buf.readBytes(req);  String body = new String(req, "UTF-8");  System.out.println("接收客户端数据:" + body);  //向客户端写数据  System.out.println("server向client发送数据");  String currentTime = new Date(System.currentTimeMillis()).toString();  ByteBuf resp = Unpooled.copiedBuffer(currentTime.getBytes());  ctx.write(resp);  }  @Override  public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {  System.out.println("server 读取数据完毕..");  ctx.flush();//刷新后才将数据发出到SocketChannel  }  @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)  throws Exception {  cause.printStackTrace();  ctx.close();  }  } |

* 客户端启动类

|  |
| --- |
| package com.netty.demo.client;  import io.netty.bootstrap.Bootstrap;  import io.netty.channel.ChannelFuture;  import io.netty.channel.ChannelInitializer;  import io.netty.channel.EventLoopGroup;  import io.netty.channel.nio.NioEventLoopGroup;  import io.netty.channel.socket.SocketChannel;  import io.netty.channel.socket.nio.NioSocketChannel;  import java.net.InetSocketAddress;  /\*\*  \* • 连接服务器 • 写数据到服务器 • 等待接受服务器返回相同的数据 • 关闭连接  \*  \* @author wilson  \*  \*/  public class EchoClient {  private final String host;  private final int port;  public EchoClient(String host, int port) {  this.host = host;  this.port = port;  }  public void start() throws Exception {  EventLoopGroup nioEventLoopGroup = null;  try {  //创建Bootstrap对象用来引导启动客户端  Bootstrap bootstrap = new Bootstrap();  //创建EventLoopGroup对象并设置到Bootstrap中，EventLoopGroup可以理解为是一个线程池，这个线程池用来处理连接、接受数据、发送数据  nioEventLoopGroup = new NioEventLoopGroup();  //创建InetSocketAddress并设置到Bootstrap中，InetSocketAddress是指定连接的服务器地址  bootstrap.group(nioEventLoopGroup).channel(NioSocketChannel.class).remoteAddress(new InetSocketAddress(host, port))  .handler(new ChannelInitializer<SocketChannel>() {  //添加一个ChannelHandler，客户端成功连接服务器后就会被执行  @Override  protected void initChannel(SocketChannel ch)  throws Exception {  ch.pipeline().addLast(new EchoClientHandler());  }  });  // • 调用Bootstrap.connect()来连接服务器  ChannelFuture f = bootstrap.connect().sync();  // • 最后关闭EventLoopGroup来释放资源  f.channel().closeFuture().sync();  } finally {  nioEventLoopGroup.shutdownGracefully().sync();  }  }  public static void main(String[] args) throws Exception {  new EchoClient("localhost", 20000).start();  }  } |

* 客户端回调方法

|  |
| --- |
| package com.netty.demo.client;  import io.netty.buffer.ByteBuf;  import io.netty.buffer.ByteBufUtil;  import io.netty.buffer.Unpooled;  import io.netty.channel.ChannelHandlerContext;  import io.netty.channel.SimpleChannelInboundHandler;    public class EchoClientHandler extends SimpleChannelInboundHandler<ByteBuf> {  //客户端连接服务器后被调用  @Override  public void channelActive(ChannelHandlerContext ctx) throws Exception {  System.out.println("客户端连接服务器，开始发送数据……");  byte[] req = "QUERY TIME ORDER".getBytes();  ByteBuf firstMessage = Unpooled.buffer(req.length);  firstMessage.writeBytes(req);  ctx.writeAndFlush(firstMessage);  }  //• 从服务器接收到数据后调用  @Override  protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {  System.out.println("client 读取server数据..");  //服务端返回消息后  ByteBuf buf = (ByteBuf) msg;  byte[] req = new byte[buf.readableBytes()];  buf.readBytes(req);  String body = new String(req, "UTF-8");  System.out.println("服务端数据为 :" + body);  }  //• 发生异常时被调用  @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  System.out.println("client exceptionCaught..");  // 释放资源  ctx.close();  }  } |

* netty中handler的执行顺序

Handler在netty中，占据着非常重要的地位。Handler与Servlet中的filter很像，通过Handler可以完成通讯报文的解码编码、拦截指定的报文、统一对错误进行处理、统一对请求进行计数、控制Handler执行与否。一句话，没有它做不到的只有你想不到的。

Netty中的所有handler都实现自ChannelHandler接口。

按照输入输出来分，分为两大类：

**ChannelInboundHandler**对接收到的报文进行处理，一般用来执行解码、读取数据、进行业务处理等；

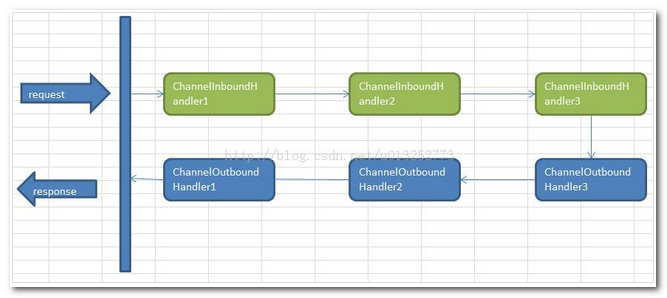
**ChannelOutboundHandler**对发出去的报文进行处理，一般用来进行编码、发送报文到对端。

Netty中，可以注册多个handler。

ChannelInboundHandler按照注册的先后**顺序**执行；

ChannelOutboundHandler按照注册的先后**逆序**执行，

如下图所示：



* 代码

见代码

* 总结

在使用Handler的过程中，需要注意：

1、ChannelInboundHandler之间的传递，通过调用 ctx.fireChannelRead(msg) 实现；调用ctx.write(msg) 将传递到ChannelOutboundHandler。

2、ctx.write()方法执行后，需要调用flush()方法才能令它立即执行。

3、流水线pipeline中outhandler不能放在最后，否则不生效

如果使用addlast方法来组装handler，则为以下执行顺序：

// 注册两个InboundHandler，执行顺序为注册顺序，所以应该是InboundHandler1 InboundHandler2

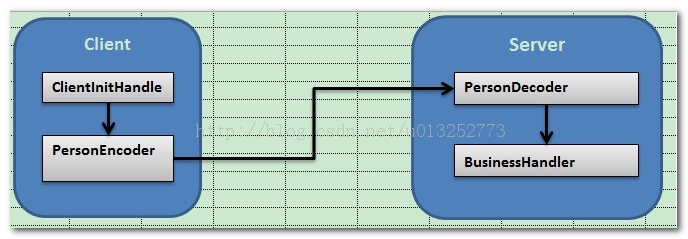
// 注册两个OutboundHandler，执行顺序为注册顺序的逆序，所以应该是OutboundHandler2 OutboundHandler1

### netty发送对象

* 3.2.1 简介

Netty中，通讯的双方建立连接后，会把数据按照ByteBuf的方式进行传输，例如http协议中，就是通过HttpRequestDecoder对ByteBuf数据流进行处理，转换成http的对象。基于这个思路，可自定义一种通讯协议：Server和客户端直接传输java对象。

实现的原理是通过Encoder把java对象转换成ByteBuf流进行传输，通过Decoder把ByteBuf转换成java对象进行处理，处理逻辑如下图所示：



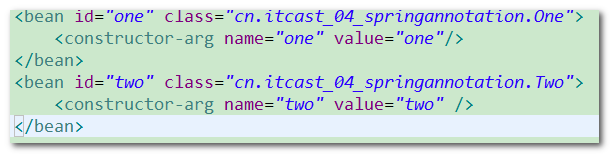
* 3.2.2 代码

见代码

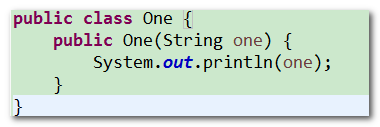
### SpringAOC/IOP学习

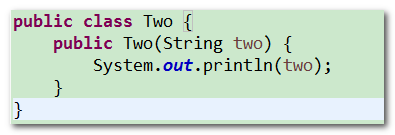
##### spring的初始化顺序

在spring的配置文件中配置bean，如下

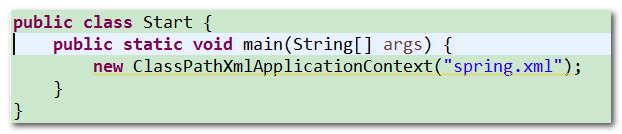


在One类和Two类中，分别实现一个参数的构造如下

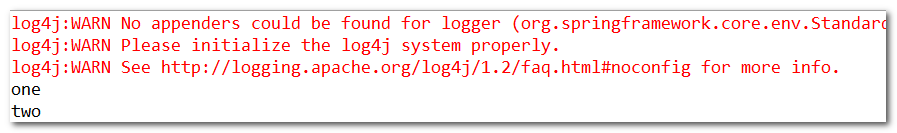




加载spring配置文件，初始化bean如下



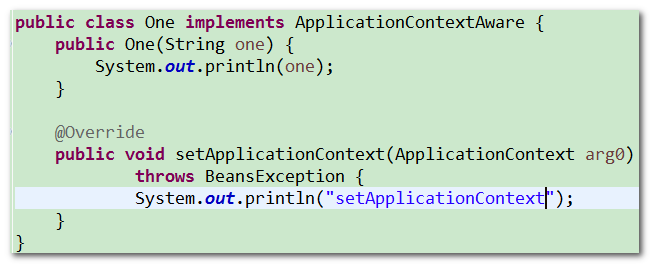
那么。结果如何呢？



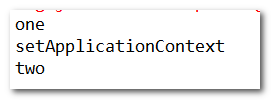
**结论：spring会按照bean的顺序依次初始化xml中配置的所有bean**

* 通过ApplicationContextAware加载Spring上下文环境

在One中实现ApplicationContextAware接口会出现如何的变换呢？

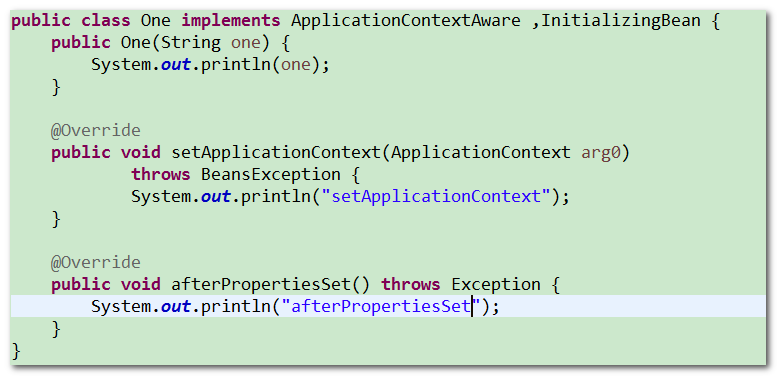


结果

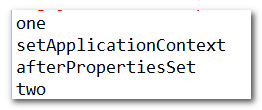


* InitializingBean的作用

在One中实现InitializingBean接口呢?



结果：



* 如果使用注解@Component

使用@Component注入类，那么它的顺序是如何呢？

* 结论

1. spring先检查注解注入的bean，并将它们实例化
2. 然后spring初始化bean的顺序是按照xml中配置的顺序依次执行构造
3. 如果某个类实现了ApplicationContextAware接口，会在类初始化完成后调用setApplicationContext（）方法进行操作
4. 如果某个类实现了InitializingBean接口，会在类初始化完成后，并在setApplicationContext（）方法执行完毕后，调用afterPropertiesSet（）方法进行操作

##### 注解使用回顾

1、在spring中，用注解来向Spring容器注册Bean。需要在applicationContext.xml中注册<context:component-scan base-package=”pagkage1[,pagkage2,…,pagkageN]”/>。

2、如果某个类的头上带有特定的注解@Component/@Repository/@Service/@Controller，就会将这个对象作为Bean注册进Spring容器

3、在使用spring管理的bean时，无需在对调用的对象进行new的过程，只需使用@Autowired将需要的bean注入本类即可

##### 自定义注解

* 解释

1、自定义注解的作用：在反射中获取注解，以取得注解修饰的“类、方法、属性”的相关解释。

2、java内置注解

@Target 表示该注解用于什么地方，可能的 ElemenetType 参数包括：

ElemenetType.CONSTRUCTOR 构造器声明

ElemenetType.FIELD 域声明（包括 enum 实例）

ElemenetType.LOCAL\_VARIABLE 局部变量声明

ElemenetType.METHOD 方法声明

ElemenetType.PACKAGE 包声明

ElemenetType.PARAMETER 参数声明

ElemenetType.TYPE 类，接口（包括注解类型）或enum声明

@Retention 表示在什么级别保存该注解信息。可选的 RetentionPolicy 参数包括：

RetentionPolicy.SOURCE 注解将被编译器丢弃

RetentionPolicy.CLASS 注解在class文件中可用，但会被JVM丢弃

RetentionPolicy.RUNTIME JVM将在运行期也保留注释，因此可以通过反射机制读取注解的信息。

* 实现

定义自定义注解

|  |
| --- |
| @Target({ ElementType.TYPE })//注解用在接口上  @Retention(RetentionPolicy.RUNTIME)//VM将在运行期也保留注释，因此可以通过反射机制读取注解的信息  @Component  public @interface RpcService {  String value();  } |

2、将直接类加到需要使用的类上，我们可以通过获取注解，来得到这个类

|  |
| --- |
| @RpcService("HelloService")  public class HelloServiceImpl implements HelloService {  public String hello(String name) {  return "Hello! " + name;  }  } |

3、类实现的接口

|  |
| --- |
| public interface HelloService {  String hello(String name);  } |

4、通过ApplicationContext获取所有标记这个注解的类

|  |
| --- |
| @Component  public class MyServer implements ApplicationContextAware {  @SuppressWarnings("resource")  public static void main(String[] args) {  new ClassPathXmlApplicationContext("spring2.xml");  }  public void setApplicationContext(ApplicationContext ctx)  throws BeansException {  Map<String, Object> serviceBeanMap = ctx  .getBeansWithAnnotation(RpcService.class);  for (Object serviceBean : serviceBeanMap.values()) {  try {  Method method = serviceBean.getClass().getMethod("hello", new Class[]{String.class});  Object invoke = method.invoke(serviceBean, "bbb");  System.out.println(invoke);  } catch (Exception e) {  e.printStackTrace();  }  }  }  } |

1. 结合spring实现junit测试

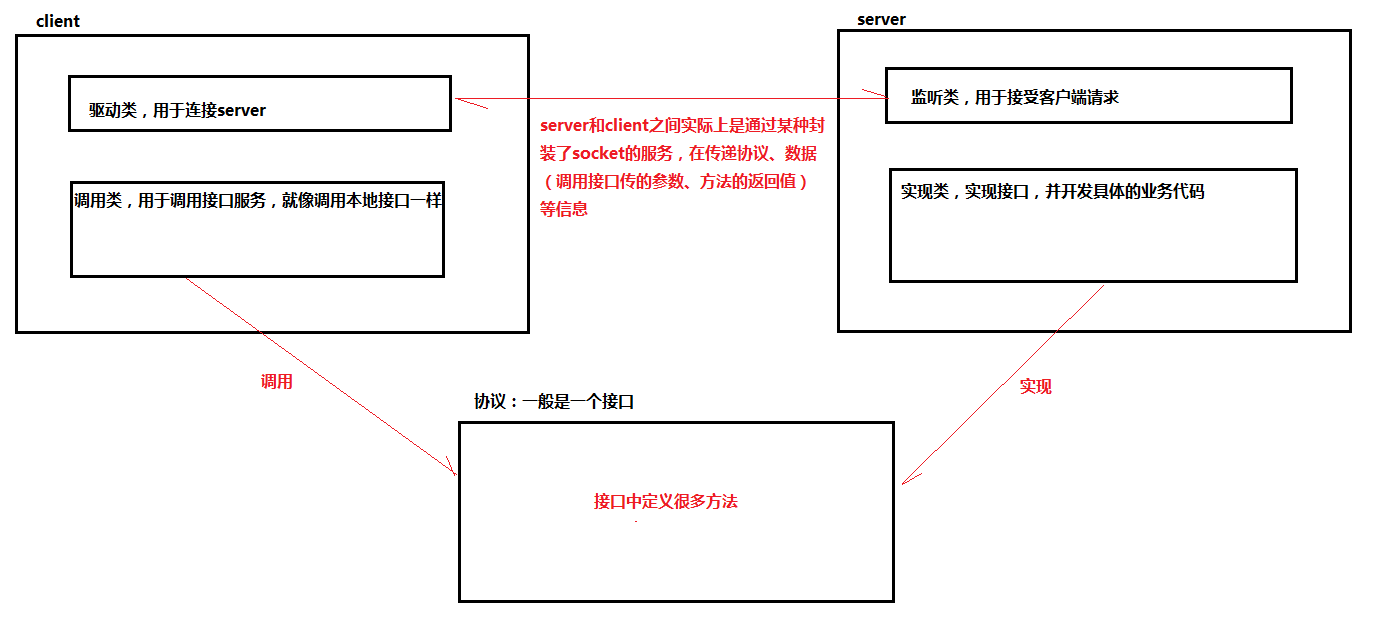
|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration(locations = "classpath:spring2.xml")  public class MyServer implements ApplicationContextAware {  @Test  public void helloTest1() {  }  public void setApplicationContext(ApplicationContext ctx)  throws BeansException {  Map<String, Object> serviceBeanMap = ctx  .getBeansWithAnnotation(RpcService.class);  for (Object serviceBean : serviceBeanMap.values()) {  try {  Method method = serviceBean.getClass().getMethod("hello",  new Class[] { String.class });  Object invoke = method.invoke(serviceBean, "bbb");  System.out.println(invoke);  } catch (Exception e) {  e.printStackTrace();  }  }  }  } |

### 轻量级RPC框架开发

#### 轻量级RPC框架需求分析及原理分析

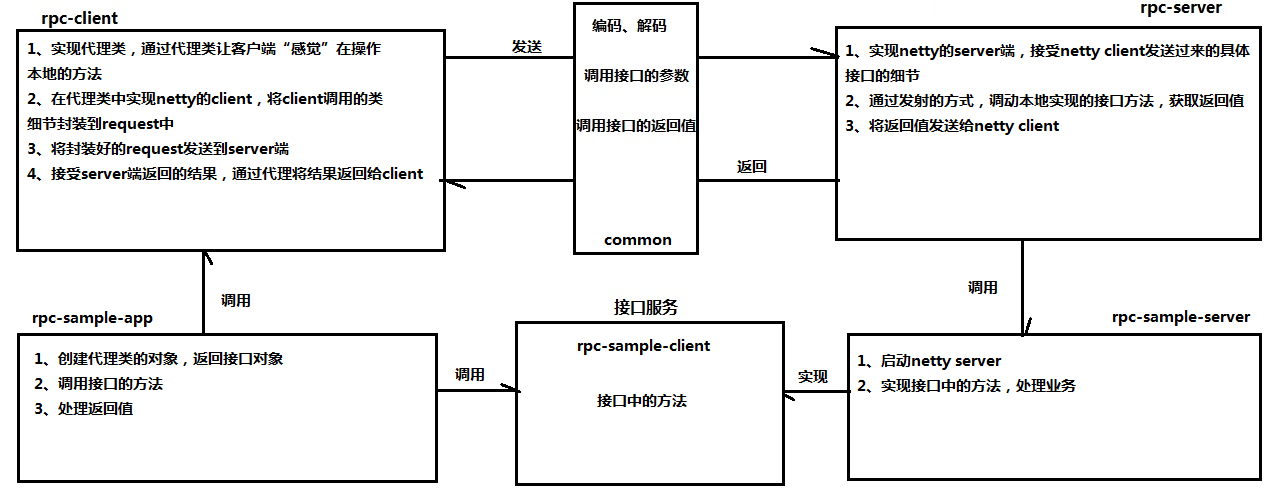
* netty实现的RPC的缺点

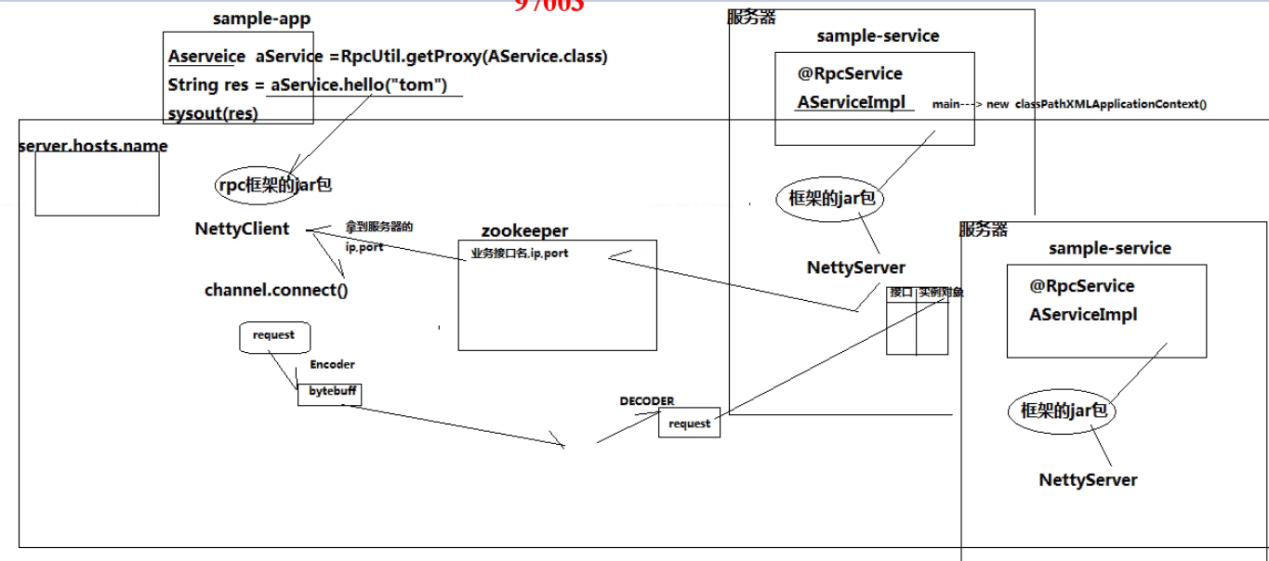
在我们平常使用的RPC中，例如webservice，使用的习惯类似于下图



但是netty的实现过于底层，我们不能够像以前一样只关心方法的调用，而是要关心数据的传输，对于不熟悉netty的开发者，需要了解很多netty的概念和逻辑，才能实现RPC的调用。

应上面的需求，我们需要基于netty实现一个我们熟悉的RPC框架。逻辑如下：



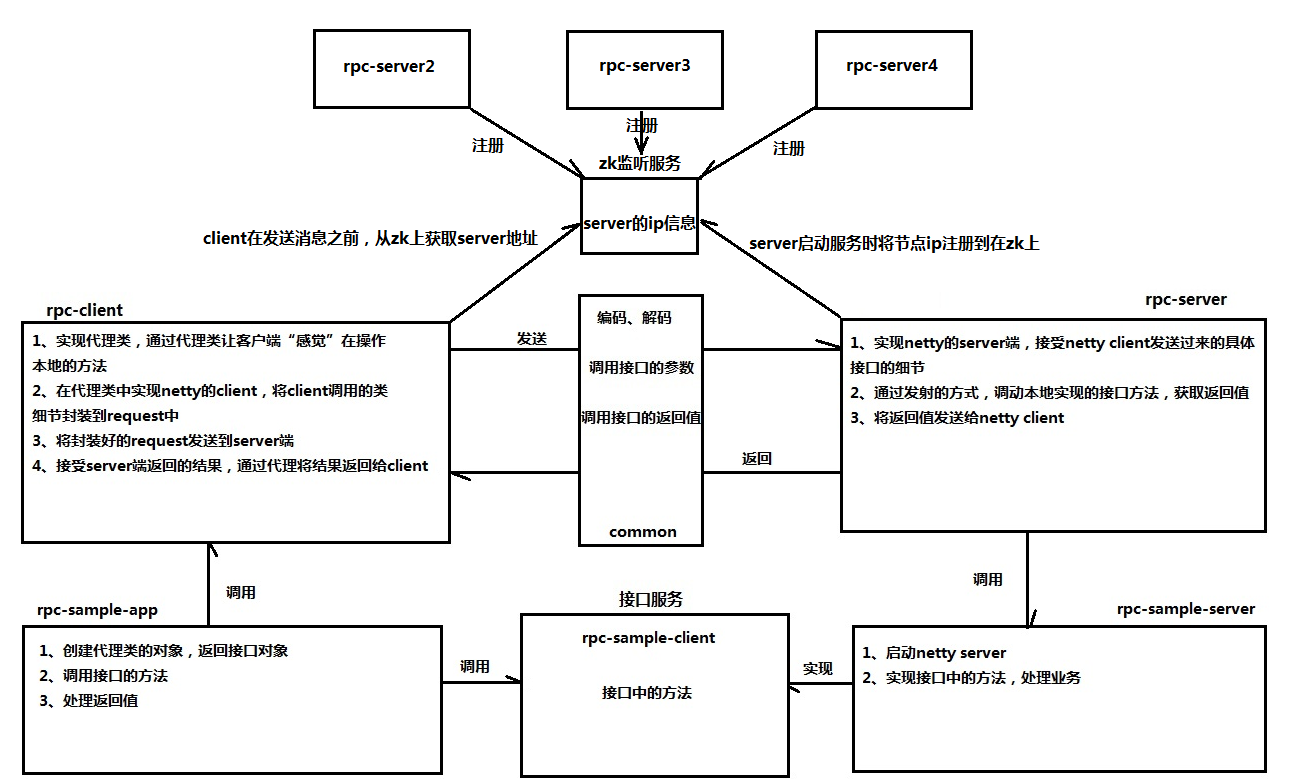


#### 轻量级RPC框架开发（仿写dubbo）

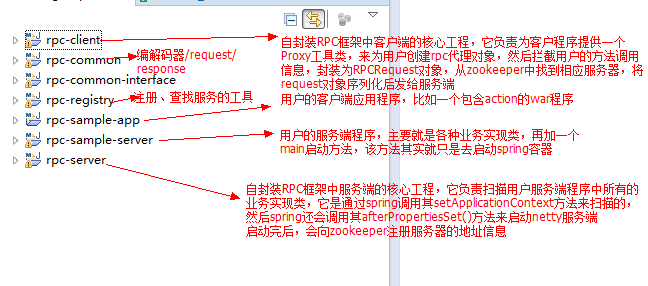
* zk在框架中的应用

在上面的框架中，server端存在着一个问题，就是单点问题，也就是说，当服务端“挂了”之后，框架的使用就造成了单点屏障。

我们可以通过zookeeper来实现服务端的负载均衡



**项目结构：**



**工程之间的依赖关系**

