# Java

## java多线程

## 集合

## 数据结构

### 3.1线性表

线性表是常见且最简单的一种数据结构，它是由n个数据元素的有序序列。实现线性表的方式有两种

1. 使用数组存储线性表的元素（即用一组连续的存储单元依次存储线性表的数据元素）

数组是一种大小固定的数据结构，对线性表的所有操作都可以通过数组来实现，虽然数组一旦创建之后，它的大小将无法改变了，但是数组不能再存储线性表中的新元素的时候，我们可以创建一个大的数组来替换当前数组，这样就可以使用数组实现动态的数据结构。

1. 使用链表存储数据元素（即用一组任意的存储单元存储线性表的数据元素，存储单元可以是连续的也可以是不连续的）

链表是一种物理存储单元上非连续，非顺序的数据结构，数据元素的逻辑是通过链表中的指针链接次序来实现的，链表有一系列节点组成，这些节点不必在内存中相连。每个节点有数据部分data和链部分next，next指向下一个节点，这样当添加或删除时，只需要改变相关节点的next的指向，效率很高。

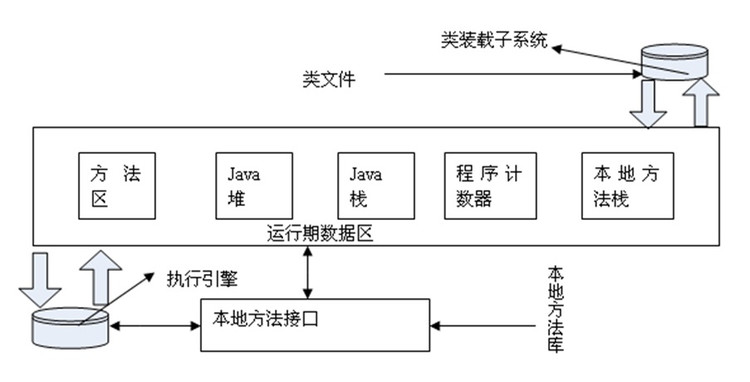
Data Next -> Data Next -> Data Next .......

### 3.2栈与队列

栈和队列也是比较常见的数据结构，它们是比较特殊的线性表，因为对于栈来说，访问、插入和删除元素只能在栈顶进行，对于队列来说元素只能从队列尾插入，从队列头访问和删除

1. 栈是限制插入和删除只能在一个位置上进行的表，该位置叫表的末端，叫做栈顶，对栈的基本操作有push(进栈)和pop(出栈)，前者相当于插入后者相当于删除最后一个元素，栈有时又叫做LIFO(Last in First out)后进先出
2. 队列是一种特殊的线性表，特殊之处在于它只允许表的前端进行删除操作，而后端进行插入操，和栈一样，队列是一种操作受限制的的线性表。进行插入操作的端称为队尾，进行删除操作的端称队头

## JVM



### 4.1 堆栈

1.堆

实例化的对象

2.栈

用来保存临时变量，

3.方法区

用来保存系统的类信息，比如类的字段，方法、常量池。方法区也可以成为永久区，如果方法区保存的东西太多了会导致内存溢出的错误。

1.复制算法

|  |  |  |  |
| --- | --- | --- | --- |
| Eden区 | S0(from) | S1(to) | tenured区 |

Java生成的对象首先会在eden区（新生代）当对象经过垃圾垃圾回收之后如果没有被回收的对象会进入S0区，如果垃圾被回收了那么S0直接将会被清除，如果没有被回收则会被复制到S1区（S0,S1是两个大小相等的区域） 在对象不断被调用跟生成的过程中系统会不断的从S0中将对象复制到S1区，没有被复制的对象将会直接被清除。在每次经历过垃圾回收之后，对象的年龄+1（即程序计数器+1）当程序年龄达到一定的阶段之后会将该对象存入老年区（tenured区）。

2.

### 4.3 堆分配参数

|  |
| --- |
| -XX:+PrintGC 使用这个参数，虚拟机启动后，只要遇到GC就会打印日志。  -XX:+UseSerialGC 配置串行回收器  -XX:+PrintGCDetails 可以查看详细信息，包括各个区的情况  -Xms:设置java程序启动时初始堆大小  -Xmx:设置java程序能获得的最大堆大小  -XX:+PrintCommandLineFlags 可以将隐式或者显示传给虚拟机的参数输出  例：-XX:+PrintGC -Xms5m -Xmx20m -XX:+UseSerialGC -XX:+PrintGCDetails  -Xmn:可以设置新生代的大小，设置一个比较大的新生代会减少老年代的大小，这个参数对系统性能以及GC行为有很大的影响，新生代大小一般会设置整个堆空间的1/3到1/4左右  -XX:SurvivorRatio 用来设置新生代中eden空间和from/to空间的比例。含义：-XX:SurvivorRatio=eden/from=eden/to  -XX:NewRatio=老年的/新生代  -XX:HeapDumpPath=d:/test.dump 当堆栈溢出之后会生成dump文件  -Xss: 设置栈的最大深度，  -XX:PermSize=64M -XX:MaxPermSize=64M 设置方法区（永久区）如果系统运行时产生大量的类，就需要设置一个相对合适的方法区 |

# 二、Linux CentOS 7.0

## 1、CentOS 7.0默认使用的是firewall作为防火墙

### 1.1、查看防火墙状态

service iptables status

### 1.2、关闭firewall：

|  |
| --- |
| systemctl stop firewalld.service #停止firewall  systemctl disable firewalld.service #禁止firewall开机启动  firewall-cmd --state #查看默认防火墙状态（关闭后显示notrunning，开启后显示running） |

### 1.3、iptables防火墙（这里iptables已经安装，下面进行配置）

|  |
| --- |
| vi/etc/sysconfig/iptables #编辑防火墙配置文件  # sampleconfiguration for iptables service  # you can edit thismanually or use system-config-firewall  # please do not askus to add additional ports/services to this default configuration  \*filter  :INPUT ACCEPT [0:0]  :FORWARD ACCEPT[0:0]  :OUTPUT ACCEPT[0:0]  -A INPUT -m state--state RELATED,ESTABLISHED -j ACCEPT  -A INPUT -p icmp -jACCEPT  -A INPUT -i lo -jACCEPT  -A INPUT -p tcp -mstate --state NEW -m tcp --dport 22 -j ACCEPT  -A INPUT -p tcp -m state --state NEW -m tcp --dport 80 -jACCEPT  -A INPUT -p tcp -m state --state NEW -m tcp --dport 8080-j ACCEPT  -A INPUT -j REJECT--reject-with icmp-host-prohibited  -A FORWARD -jREJECT --reject-with icmp-host-prohibited  COMMIT  :wq! #保存退出  备注：这里使用80和8080端口为例。\*\*\*部分一般添加到“-A INPUT -p tcp -m state --state NEW -m tcp--dport 22 -j ACCEPT”行的上面或者下面，切记不要添加到最后一行，否则防火墙重启后不生效。  systemctlrestart iptables.service #最后重启防火墙使配置生效  systemctlenable iptables.service #设置防火墙开机启动 |

## 2、解压文件

tar -cvxf XXXX

## 3、设置权限

chmod 777 aaa

## 设置不要图形界面的linux

|  |
| --- |
| 1.修改 /etc/inittab 文件，将“id:5:initdefault:”这一行的"id:"后的数字（默认为5）改为 3即可。  2.开机以命令模式启动，执行： systemctl set-default multi-user.target  开机以图形界面启动，执行： systemctl set-default graphical.target |

## 5、重启网络

service network restart

## 6、命令

|  |
| --- |
| 1.touch 创建文件  2.cat 查看文件（少量）  3.more 查看文件（大量）支持翻页  4.head 查看文件的前几行  5.tail 查看文件的后几行 tail -f aaa 动态查看文件，可用于日志监控  6.ln -s(link) 创建软连接文件/不加s则是创建硬连接  7.chmod u+x 给当前用户添加x权限 chomd u-x 取消当前用户的x权限  8.find 文件搜索 -name 文件名称 -size 文件大小  9.chown -R fliay:fliay xxxx 给用户赋值权限  10.locate 查找文件根据linux数据库内部的索引updatedb命令，可以手工更新updatedb数据库  11.gzip 压缩的时候不保留源文件，并且只能压缩文件不能压缩目录  gunzip 解压文件，不保留源文件  例：gzip aaa.txt 解压 gunzip aaa.txt.gzip  12 tar [zcvf] [zxvf] 打包文件名.tar.gz 保留源文件  -c 产生tar打包文件（必选）  -x 产生解压缩文件（必选）  -v 显示详细信息  -f 指定压缩后的文件名  -z 打包同时压缩  例：打包tar -zcvf aaa.tar.gz aaa/  解压tar -zxvf aa.tar.gz   1. zip 选项[-r] 【压缩后文件名称】 【源文件】   unzip 解压缩的文件 -d表示文件解压后存放的路径   1. 将内容输出到文件中   例： cat aaa.txt > a.log  cat aaa.txt>>a.log  两个>为追加一个>为覆盖   1. wall<a.log 广播 2. :%s/old/new/g 全文替换指定字符串 3. :n1,n2s/old/new/g 在一定范围内替换指定字符串 4. r 取代光标所在处字符 5. R 从光标所在处开始替换字符，按ESC结束 6. u 取消上一步操作 |

## 7、

# 三、Nginx

## 1、CentOS7.0安装Nginx 1.7.4 安装准备

首先由于nginx的一些模块依赖一些lib库，所以在安装nginx之前，必须先安装这些lib库，这些依赖库主要有g++、gcc、openssl-devel、pcre-devel和zlib-devel 所以执行如下命令安装

1. $   yum install gcc-c++
2. $   yum install pcre pcre-devel
3. $   yum install zlib zlib-devel
4. $   yum install openssl openssl-devel

## 2、安装Nginx

### 2.1、安装之前，最好检查一下是否已经安装有nginx

1. $   find -name nginx

### 2.2、如果系统已经安装了nginx，那么就先卸载

1. $   yum remove nginx

### 2.3、首先进入/usr/local目录

1. $   cd /usr/local

### 2.4、从官网下载最新版的nginx

1. $   wget http://nginx.org/download/nginx-1.7.4.tar.gz

### 2.5、解压nginx压缩包

1. $   tar -zxvf nginx-1.7.4.tar.gz

### 2.6、会产生一个nginx-1.7.4 目录，这时进入nginx-1.7.4目录

1. $   cd  nginx-1.7.4

### 2.7、接下来安装，使用--prefix参数指定nginx安装的目录,make、make install安装y

1. $   ./configure  $默认安装在/usr/local/nginx
2. $   make
3. $   make install

### 2.8、如果没有报错，顺利完成后，最好看一下nginx的安装目录 默认的安装目录/usr/local

1. $   whereis nginx
2. $ cd/usr/local/nginx
3. conf html logs sbin
4. cd sbin
5. Nginx
6. #查看所有的命令
7. [root@localhost conf]# cd ../conf
8. [root@localhost conf]# ls
9. fastcgi.conf koi-win scgi\_params
10. fastcgi.conf.default mime.types scgi\_params.default
11. fastcgi\_params mime.types.default uwsgi\_params
12. fastcgi\_params.default nginx.conf uwsgi\_params.default
13. koi-utf nginx.conf.default win-utf

## 3、测试nginx 的启动

1. $ [root@localhost nginx]# cd sbin/
2. [root@localhost sbin]# ls
3. nginx
4. [root@localhost sbin]# ./nginx
5. [root@localhost sbin]# ./nginx -t
6. nginx: the configuration file /usr/local/nginx/conf/nginx.conf syntax is ok
7. nginx: configuration file /usr/local/nginx/conf/nginx.conf test is successful

## 启动 nginx

### 4.1、启动命令

1.[root@localhost sbin]# ./nginx

nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)

nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)

nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)

nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)

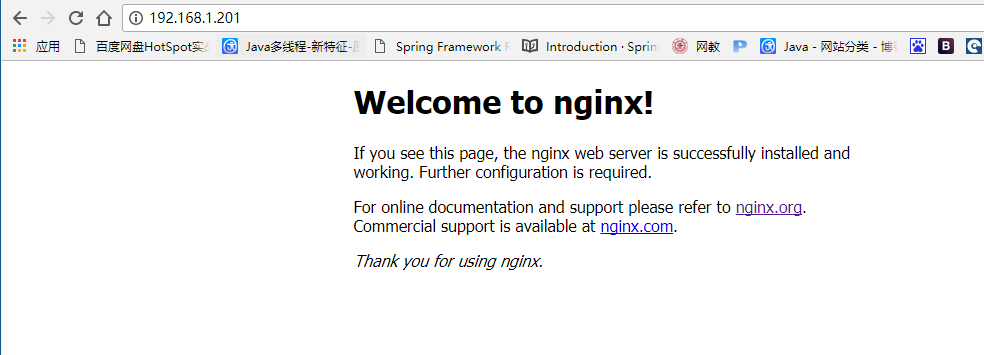
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)

nginx: [emerg] still could not bind()

### 4.2、通过远程访问ip地址

防火墙开启80端口 firewall-cmd --zone=public --add-port=80/tcp --permanent

重启防火墙 firewall-cmd --reload



### 4.3、停止nginx

1.[root@localhost sbin]# ./nginx -s stop 或 ./nginx -s quit

### 4.4、重启nginx

1. [root@localhost sbin]# ./nginx -s stop 或 ./nginx -s reload

## 5、查看进程

1.[root@localhost sbin]# ps -ef | grep nginx(java|tomcat)

## 6、Nginx Http模块

### 6.1、Nginx.conf文件配置结构

http{ //协议级别

.....

server{//服务器级别

......

location /{ //请求级别

}

}

}

### 6.2、 HTTP Core 核心模块

### 6.3、 Location 配置

location匹配命令

~      #波浪线表示执行一个正则匹配，区分大小写  
~\*    #表示执行一个正则匹配，不区分大小写  
^~    #^~表示普通字符匹配，如果该选项匹配，只匹配该选项，不匹配别的选项，一般用来匹配目录  
=      #进行普通字符精确匹配  
@     #"@" 定义一个命名的 location，使用在内部定向时，例如 error\_page, try\_files

location 匹配的优先级(与location在配置文件中的顺序无关)  
= 精确匹配会第一个被处理。如果发现精确匹配，nginx停止搜索其他匹配。  
普通字符匹配，正则表达式规则和长的块规则将被优先和查询匹配，也就是说如果该项匹配还需去看有没有正则表达式匹配和更长的匹配。  
^~ 则只匹配该规则，nginx停止搜索其他匹配，否则nginx会继续处理其他location指令。  
最后匹配理带有"~"和"~\*"的指令，如果找到相应的匹配，则nginx停止搜索其他匹配；当没有正则表达式或者没有正则表达式被匹配的情况下，那么匹配程度最高的逐字匹配指令会被使用。

location 优先级官方文档

Directives with the = prefix that match the query exactly. If found, searching stops.

All remaining directives with conventional strings, longest match first. If this match used the ^~ prefix, searching stops.

Regular expressions, in order of definition in the configuration file.

If #3 yielded a match, that result is used. Else the match from #2 is used.

=前缀的指令严格匹配这个查询。如果找到，停止搜索。

所有剩下的常规字符串，最长的匹配。如果这个匹配使用^〜前缀，搜索停止。

正则表达式，在配置文件中定义的顺序。

如果第3条规则产生匹配的话，结果被使用。否则，使用第2条规则的结果。

例如

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | location  = / {    # 只匹配"/".    [ configuration A ]  }  location  / {    # 匹配任何请求，因为所有请求都是以"/"开始    # 但是更长字符匹配或者正则表达式匹配会优先匹配    [ configuration B ]  }  location ^~ /images/ {    # 匹配任何以 /images/ 开始的请求，并停止匹配 其它location    [ configuration C ]  }  location ~\* .(gif|jpg|jpeg)$ {    # 匹配以 gif, jpg, or jpeg结尾的请求.    # 但是所有 /images/ 目录的请求将由 [Configuration C]处理.    [ configuration D ]  } |

请求URI例子:

/ -> 符合configuration A

/documents/document.html -> 符合configuration B

/images/1.gif -> 符合configuration C

/documents/1.jpg ->符合 configuration D

@location 例子  
error\_page 404 = @fetch;

location @fetch(  
proxy\_pass http://fetch;  
)

# LinuxCentOS 7.0安装jdk

## 4.1 安装jdk

### 1、上传jdk文件

将jdk-8u144-linux-x64.tar.gz上传至/usr/java 目录下

### 2、执行解压命令

输入 tar -zxvf jdk-8u144-linux-x64.tar.gz

### 执行vim ~/.bashrc

### 4、输入环境变量配置信息

export JAVA\_HOME=/usr/java/jdk1.8.0\_144

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

export CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar

### 保存并退出 WQ

### 运行bashrc文件

输入source ~/.bashrc文件

### 验证是否配置成功 输入javac java

## 4.2 linux 远程拷贝scp

scp -r aaa / 192.168.1.210:/user/local

# Maven

## 1. maven 插件

### 1.1 eclipse 用maven添加Tomcat插件

<build>

<!-- 配置插件 -->

<plugins>

<plugin>

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

<artifactId>tomcat7-maven-plugin</artifactId>

<configuration>

<port>8080</port>

<path>/</path>

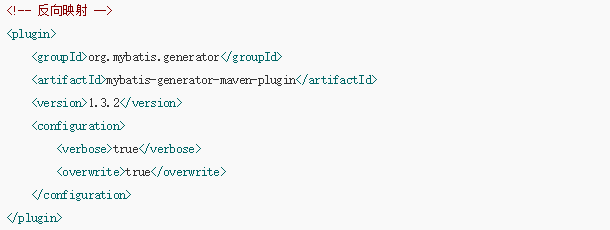
</configuration>

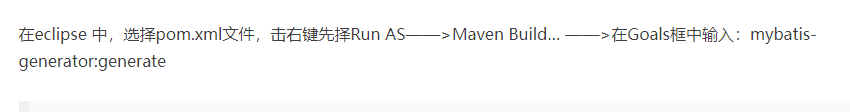
</plugin>

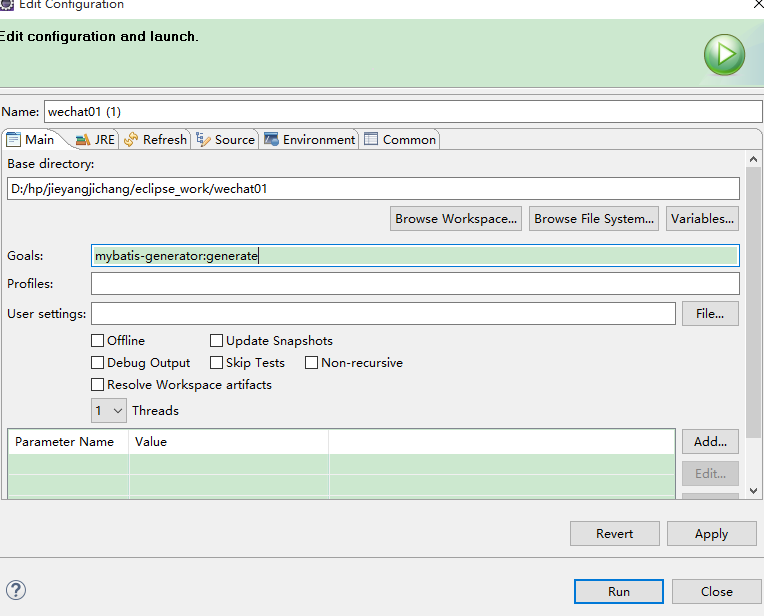
</plugins>

</build>

### 1.2 使用maven插件反向映射generatorConfig.xml生成代码







## maven 导入jar包

mvn install:install-file -Dfile=I:\rome-1.0.jar -DgroupId=com.sun.syndication -DartifactId=rome -Dversion=1.0 -Dpackaging=jar

# 连接池配置

## 1.配置\_DruidDataSource参考配置

|  |
| --- |
| <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close">  <!-- 数据库驱动 -->  <property name="driverClassName" value="${jdbc.driverClassName}" />  <!-- 基本属性 url、user、password -->  <property name="url" value="${jdbc\_url}" />  <property name="username" value="${jdbc\_user}" />  <property name="password" value="${jdbc\_password}" />  <!-- 配置初始化大小、最小、最大 -->  <property name="initialSize" value="1" />  <property name="minIdle" value="1" />  <property name="maxActive" value="20" />  <!-- 配置获取连接等待超时的时间 -->  <property name="maxWait" value="60000" />  <!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->  <property name="timeBetweenEvictionRunsMillis" value="60000" />  <!-- 配置一个连接在池中最小生存的时间，单位是毫秒 -->  <property name="minEvictableIdleTimeMillis" value="300000" />  <property name="validationQuery" value="SELECT 'x'" />  <property name="testWhileIdle" value="true" />  <property name="testOnBorrow" value="false" />  <property name="testOnReturn" value="false" />  <!-- 打开PSCache，并且指定每个连接上PSCache的大小 -->  <property name="poolPreparedStatements" value="true" />  <property name="maxPoolPreparedStatementPerConnectionSize" value="20" />  <!-- 配置监控统计拦截的filters -->  <property name="filters" value="stat" />  </bean> |

通常来说，只需要修改initialSize、minIdle、maxActive。

如果用Oracle，则把poolPreparedStatements配置为true，mysql可以配置为false。分库分表较多的数据库，建议配置为false。

druid常见问题汇总：

<https://github.com/alibaba/druid/wiki/%E5%B8%B8%E8%A7%81%E9%97%AE%E9%A2%98>

# 配置ftp服务

## 1.安装vsftpd组件

安装完后，有/etc/vsftpd/vsftpd.conf 文件，是vsftp的配置文件。

[root@bogon ~]# yum -y install vsftpd

## 2.添加一个ftp用户

此用户就是用来登录ftp服务器用的。

[root@bogon ~]# useradd ftpuser

这样一个用户建完，可以用这个登录，记得用普通登录不要用匿名了。登录后默认的路径为 /home/ftpuser.

## 3.给ftp用户添加密码。

[root@bogon ~]# passwd ftpuser

输入两次密码后修改密码。

## 4.修改配置文件

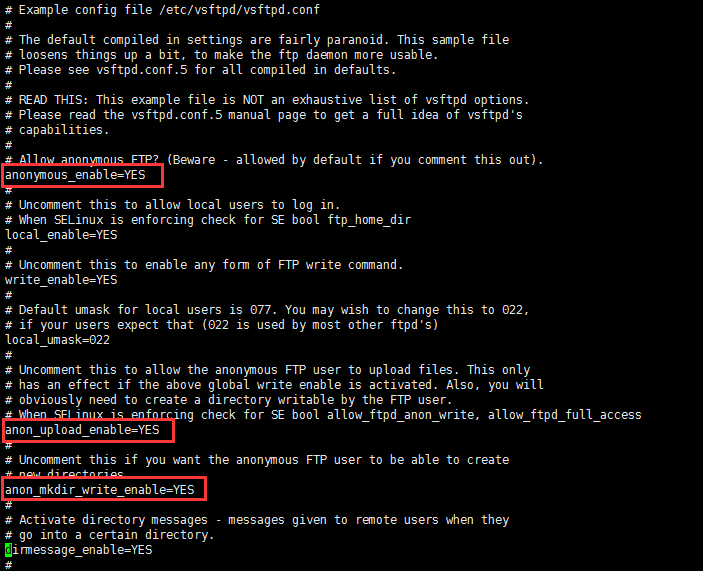
vi /etc/vsftpd/vsftpd.conf

    保证下面3项为YES

    anonymous\_enable=YES

    anon\_upload\_enable=YES

    anon\_mkdir\_write\_enable=YES

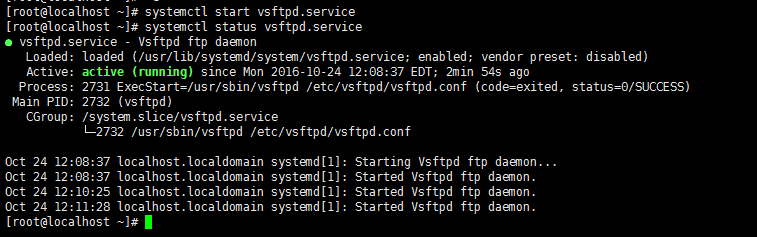


## 设置vsftpd开机启动

systemctl enable vsftpd.service

1. 启动并查看vsftpd服务状态，systemctl启动服务成功不会有任何提示，绿色的active表示服务正在运行

systemctl start vsftpd.service  
systemctl status vsftpd.service



## 6.文件读写

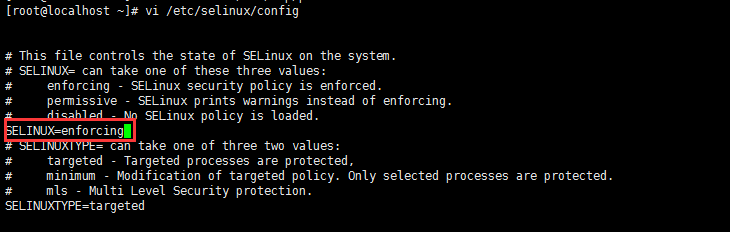
设置文件夹权限，将pub文件夹的权限设置为777

chmod 777 -R /var/ftp/pub

## 6.关闭selinux服务

vi /etc/selinux/config

  将SELINUX=enforcing改为：SELINUX=disabled



## 6.重启ftp服务

service vsftpd restart

## 7.设置开机启动vsftpd ftp服务

[root@bogon ~]# chkconfig vsftpd on

# redis缓存服务器

## 6.安装redis环境

1. 需要安装gcc

gcc：yum install gcc-c++

1. 从官网下载redis安装包

[http://redis.io/download](http://redis.io/download" \t "http://www.runoob.com/redis/_blank)

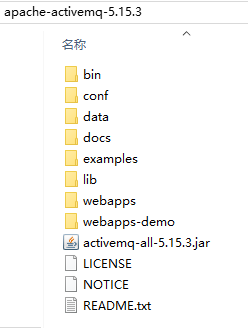
1. 在linux中输入 wget http://download.redis.io/releases/redis-4.0.6.tar.gz

# ActiveMQ

## 下载

登录activeMQ官网下载 <http://activemq.apache.org/download.html>

## 解压



## bin目录

bin目录中有win32与win64 两个文件夹 根据当前的操作系统来启动不同的启动项

## 双击activemq.bat进行启动

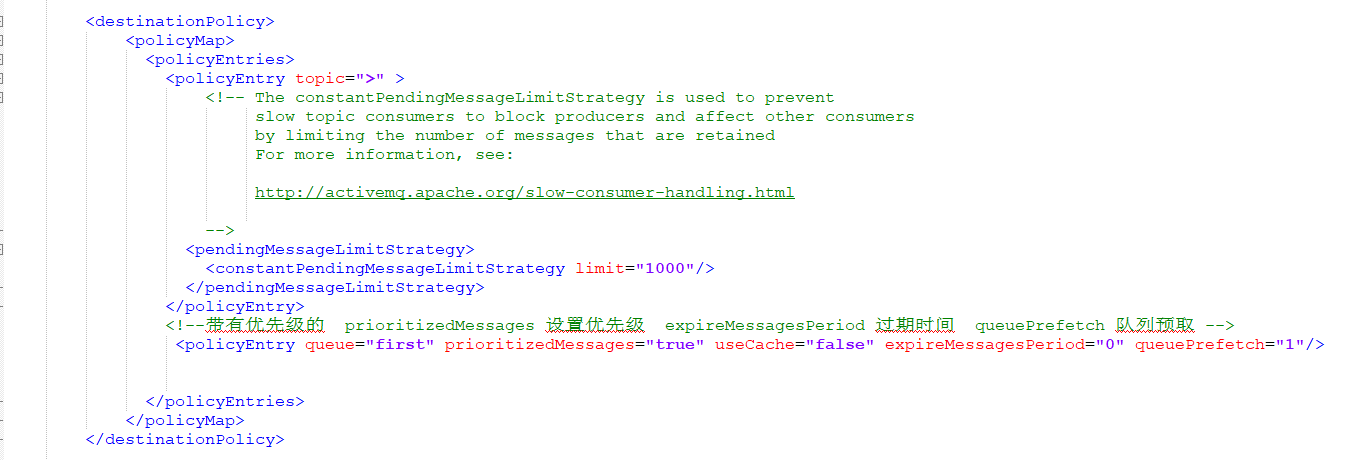
## 访问activemq后台

activemq默认的网页后台访问地址为localhost:8161 点击进行登录身份的认证，默认是admin，admin 可以在conf目录的 activemq.xml 来进行配置

## 配置优先级队列

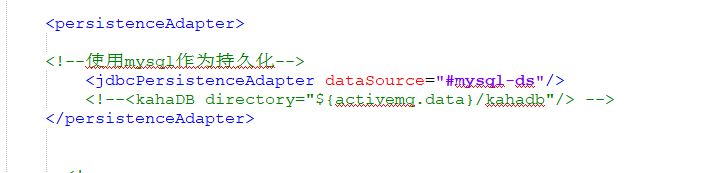
在如下的标签中进行配置队列的优先级， <policyEntry queue="first" prioritizedMessages="true" useCache="false" expireMessagesPeriod="0" queuePrefetch="1"/> prioritizedMessages

设置优先级 expireMessagesPeriod 过期时间 queuePrefetch 队列预取



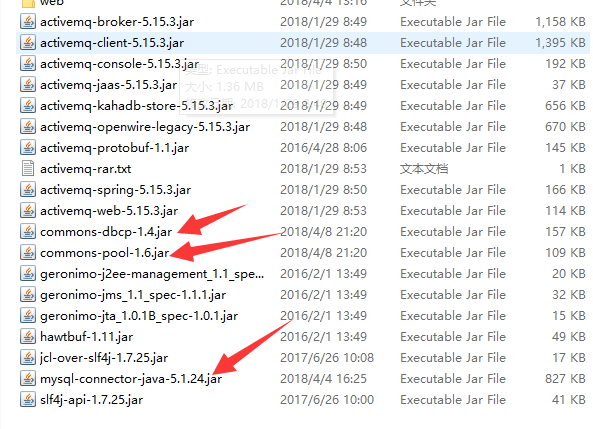
1. 添加mysql持久化

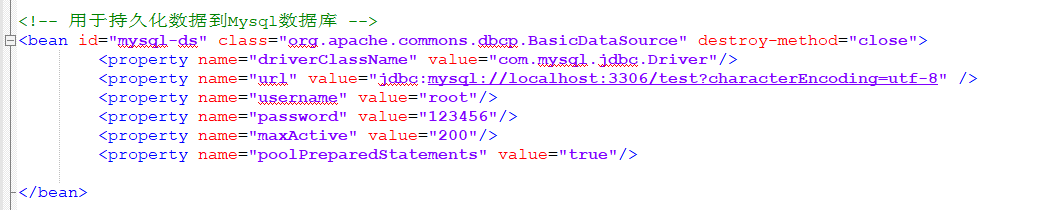
将默认的kahadb注释，添加自定义的mysql-ds



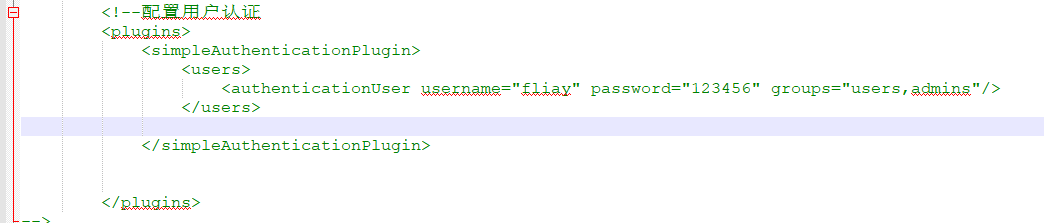
自定义mysql-ds bean配置 需要在activemq目录下的lib目录中添加如下包，否则会报错

mysql-connector-java-5.1.24，commons-pool-1.6，commons-dbcp-1.4



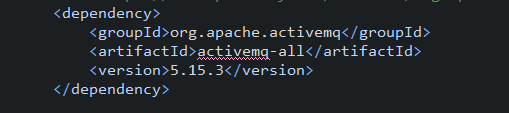


1. 配置用户认证



用户认证是在java登录activemq的时候使用的，默认是admin，admin

1. 引入activemq的相关的maven库jar



1. 编写测试案例 sender

public static void main(String[] args) throws JMSException, InterruptedException {

// 第一步，建立ConnectionFactory工厂对象，需要填入用户名，密码，以及要连接的地址，均使用默认的即可，默认端口：“tcp://localhost:616161”

ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("admin",

"admin", "tcp://localhost:61616");

// 第二步，简历ConnectionFactory工厂对象我们创建一个connection链接，并且调用Connection的start方法开启链接，Connection默认是关闭的

Connection connection = connectionFactory.createConnection();

connection.start();

//第三步，通过connection对象创建session会话（上下文环境对象），用于接受消息，参数配置1为是否启用事务，参数配置2为签收模式，一般我们设置自动签收

Session session = connection.createSession(Boolean.TRUE, Session.AUTO\_ACKNOWLEDGE);//自动签收模式 Boolean.TRUE 为支持事务提交

//Session session = connection.createSession(Boolean.TRUE, Session.CLIENT\_ACKNOWLEDGE);//客户端必须签收

//第四步，通过Session 创建destination对象，指的是客户端用来指定生产消息目标和消费消息来源的对象，在PTP模式中，Destination被称作

Destination destination = session.createQueue("first");

//第五步，我们需要通过Session对象创建消息的发送和接收对象（生产者和消费者）MessageProducer/MessageConsumer

MessageProducer producer = session.createProducer(null);

//第六步，我们可以使用MessageProducer的setDeliveryMode方法为其设置持久化特性和非持久化特性（DeliveryMode）

//producer.setDeliveryMode(DeliveryMode.NON\_PERSISTENT);//非持久化

producer.setDeliveryMode(DeliveryMode.PERSISTENT);

//第七步，最后我们使用JMS规范的TextMessage形式创建数据（通过Session对象），并用MessageProduce r的send方法发送数据，同理客户端用receive方法获取数据

for(int i =1;i<10;i++){

TextMessage msg = session.createTextMessage("我是消息内容"+i);

//producer.send(destination,msg);

producer.send(destination, msg, DeliveryMode.PERSISTENT, i, 1000\*20);//i是优先级，20000存活时间

}

TimeUnit.SECONDS.sleep(3);

//提交事务，整体提交数据

session.commit();

if(connection!=null){

connection.close();

}

}

1. 编写案例receiver

public static void main(String[] args) throws JMSException {

// 第一步，建立ConnectionFactory工厂对象，需要填入用户名，密码，以及链接地址，均使用默认即可，默认端口为：tcp://localhost:61616

ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("admin",

"admin", "tcp://localhost:61616");

// 第二步，通过ConnectionFactory工程对象我们创建一个Connection连接，并且调用Connection的start方法开启链接，Connection默认是关闭的

Connection connection = connectionFactory.createConnection();

connection.start();// 开启

// 第三步，通过connection对象创建session会话（上下文环境对象），用于接收消息，参数配置1为是否启用事务，参数配置2为签收模式，一般设置自动签收

Session session = connection.createSession(Boolean.FALSE, Session.AUTO\_ACKNOWLEDGE);//自动签收模式

//Session session = connection.createSession(Boolean.FALSE, Session.CLIENT\_ACKNOWLEDGE);//客户端必须签收

//第四步，通过Session创建Destination对象，指的是一个客户端用来指定生产消息目标和消费消息来源的对象，在PTP模式中，Destination被称作Queue即队列

Destination destination = session.createQueue("first");

//第五步，通过session创建MessageConsumer

MessageConsumer consumer = session.createConsumer(destination);

while(true){

TextMessage msg = (TextMessage) consumer.receive();

//msg.acknowledge();//手动签收 在Session.CLIENT\_ACKNOWLEDGE模式下必须手动签收一下

System.out.println("消费数据："+msg.getText());

}

}

# windows 批处理

## 删除文件的命令

del /f/s/q \*.prefs .DS\_Store

# Netty



## Netty实现通信的步骤

1. 创建两个NIO线程组，一个专门用于网络事件处理（接受客户端的链接），另一个则进行网络通信读写。
2. 创建一个ServerBootstrap对象，配置Netty的一系列参数，例如接受传出数据的缓存大小等
3. 创建一个实际处理数据的类ChannelInitializer，进行初始化的准备工作，比如设置接受传出数据的字符集、格式、已经实际处理数据的接口。
4. 绑定端口，执行同步阻塞方法等待服务器端启动即可。

## Netty写一个Helloword

1. 创建server服务器端

|  |
| --- |
| public class Server {  /\*\*  \* @param args  \* 对于channelOption.SO\_BACKLOG的解释：  \* 服务器端TCP内核模块维护有两个队列，我们称为A,B  \* 客户端想服务器端connect的时候，会发送带有SYN标志的包（第一次握手）  \* 服务器收到客户端发来的SYN时，想客户端发送SYN ACK确认（第二次握手）  \* 此时TCP内核模块把客户端连接从A队列移到B队列，连接完成，应用程序的accept会返回，  \* 也就是说accept从B队列中去除完成三次握手的连接  \* A队列和B队列的长度之和是backlog，当A，B队列的长度之和大于backlog时，新连接将会被TCP内核拒绝，  \* 所以，如果backlog过小，可能会出现accept速度跟不上， A,B队列满了，导致新的客户端无法连接。  \* 要注意的是：backlog对程序支持的连接数，backlog影响的只是还没有accept取出的连接。  \* @throws InterruptedException  \*/  public static void main(String[] args) throws InterruptedException {  int port = 3000;  // 创建用于服务器端接收客户端连接的  EventLoopGroup sGroup = new NioEventLoopGroup();  // 创建用于进行网络通信的通道  EventLoopGroup cGroup = new NioEventLoopGroup();  // 创建辅助工具类，用于服务器通道的一系列配置  ServerBootstrap b = new ServerBootstrap();  // 首先创建要与线程组进行绑定  b.group(sGroup, cGroup)  // 绑定成功之后需要指定NIO的模式  .channel(NioServerSocketChannel.class)  // 设置tcp缓冲区大小  .option(ChannelOption.SO\_BACKLOG, 1024)  // 设置发送缓冲区大小（buffer缓冲区大小）  .option(ChannelOption.SO\_SNDBUF, 32 \* 1024)  // 设置接收缓冲区大小  .option(ChannelOption.SO\_RCVBUF, 32 \* 1024)  //保持连接  .option(ChannelOption.SO\_KEEPALIVE,true)  // 获取客户端连接通道  .childHandler(new ChannelInitializer<SocketChannel>() {  @Override  protected void initChannel(SocketChannel ch) throws Exception {  // 用于过滤数据，编码，解码，在这里配置具体的数据接收方法和处理方法  ch.pipeline().addLast(new ServerHandler());  }  });  // 进行绑定异步通道  ChannelFuture cf = b.bind(port).sync();  cf.channel().closeFuture().sync();  cGroup.shutdownGracefully();  sGroup.shutdownGracefully();  }  } |

1. ServerHandler处理

|  |
| --- |
| public class ServerHandler extends ChannelHandlerAdapter {  @Override  public void handlerAdded(ChannelHandlerContext ctx) throws Exception {  SimpleDateFormat simple = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");  String response = simple.format(new Date())+":success receive a message";  ctx.writeAndFlush(Unpooled.copiedBuffer(response.getBytes()));  }  @Override  public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  SimpleDateFormat simple = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");  ByteBuf buf = (ByteBuf) msg;  //创建大小为buf可读空间大小的字节数组  byte[] req = new byte[buf.readableBytes()];  buf.readBytes(req);  String body = new String(req,"UTF-8");  System.out.println("client:"+body);  //服务器端的响应  String response = simple.format(new Date())+":success receive a message";  ctx.writeAndFlush(Unpooled.copiedBuffer(response.getBytes()));  }  } |

1. 创建客户端

|  |
| --- |
| public class Client {  public static void main(String[] args) throws InterruptedException {  int port =3000;  String host="127.0.0.1";  EventLoopGroup cGroup = new NioEventLoopGroup();  Bootstrap b = new Bootstrap();  b.group(cGroup)  //.option(ChannelOption.SO\_TIMEOUT, 20000)超时  .channel(NioSocketChannel.class)  .handler(new ChannelInitializer<SocketChannel>() {  @Override  protected void initChannel(SocketChannel ch) throws Exception {  ch.pipeline().addLast(new ClientHandler());  }  });  ChannelFuture cf = b.connect(host,port);  cf.channel().writeAndFlush(Unpooled.copiedBuffer("An message from client".getBytes()));  Thread.sleep(1000);  cf.channel().closeFuture().sync();  cGroup.shutdownGracefully();  }  } |

1. 客户端处理

|  |
| --- |
| public class ClientHandler extends ChannelHandlerAdapter {    @Override  public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  try {  ByteBuf buf = (ByteBuf) msg;  byte[] req = new byte[buf.readableBytes()];  buf.readBytes(req);  String body = new String(req, "UTF-8");  System.out.println("client:" + body);  } finally {  ReferenceCountUtil.release(msg);  }  }  } |

# HTTP

## http常见协议

HTTP协议的响应消息：相应消息也是由三部分组成的：状态行、请求头、响应正文响应状态种类：

1xx:提示信息，表示请求已经接收继续处理。

2xx:成功。表示请求已经接收成功

3xx:重定向。要完成的请求必须进行更进一步的操作

4xx:客户端错误。可能是请求语法错误或者请求无法实现。

5xx:服务器端错误。服务器韦恩那个处理请求（可能内部出现异常）

# Mongodb常用命令

mongorestore.exe -d AreaPictureSystem D:\AreaPictureSystem

db.resource.remove({});

db.resource.find().pretty();

db.dropDatabase() 删除数据库

# JDK1.8

# Redis

## 15.1非关系型数据库特点

1. 数据模型比较简单
2. 需要灵活性更强的IT系统
3. 对数据库性能要求高
4. 不需要高度的数据一致性
5. 对于给定的key，比较容易映射复杂的环境

## 15.2 Redis简介

redis是以key-value形式存储，和传统的关系型数据库不一样，不一定遵循传统数据库的一些基本要求（非关系型的，分布式的，开源的，水平可扩展的）

优点：对数据高并发读写

对海量数据的搞笑率存储和访问

对数据的可扩展性和高可用性

缺点：redis（ACID处理非常简单）

无法做到太复杂的关系型数据库模型

Redis是以key-value store存储，data structure service 数据结构服务器。键可以包含(string)字符串，哈希，(list)链表，(set)集合，(zset)有序集合。这些数据结合都支持push/pop、add/remove 及交集和并集以及更丰富的操作，redis支持各种不同的方式排序，为了保证效率，数据都是缓存在内存中，它也可以周期性的把更新的数据写入磁盘或者吧修改操作写入追加到文件中。

## 15.3 LinuxRedis安装

参考博客https://www.cnblogs.com/zuidongfeng/p/8032505.html

redis 默认占用6379端口

### 15.3.1 下载redis 包

在命令窗口中输入 wget http://download.redis.io/releases/redis-4.0.6.tar.gz

### 15.3.2 解压redis压缩包

tar -zxvf redis-4.0.6.tar.gz

### 15.3.3 yum安装gcc依赖

yum install gcc

### 15.3.4 编译安装

cd redis-4.0.6

编译安装

make MALLOC=libc

将redis-4.0.6/src目录下的文件加到/user/local/bin目录

cd src &&make install

### 15.3.5 进入src目录测试启动redis

输入指令 ./redis-server

### 15.3.6 设置后台进程访问

1.修改redis.conf文件

将daemonize no => 修改为 daemonize yes

2.指定redis.conf文件启动

./redis-server /sortware/redis-4.0.6/redis.conf

3.查看redis进程

ps -aux |grep redis

4.杀死进程

kill -9 进程id

### 15.3.7 开机自启动

1.进入 etc目录创建redis文件夹

mkdir redis

2.进入安装的redis目录将redis.conf文件考分到etc下的redis目录并且需要将名称更改为6379.conf

cp redis.conf /etc/redis/6379.conf

3.将redis的启动脚本复制到/etc/init.d目录下

进入安装包下的utils目录

cp redis\_init\_script /etc/init.d/redisd

4.设置开机自启动先进入init.d目录下

输入cp redis\_init\_script /etc/init.d/redisd

出现service redisd does not support chkconfig

redisd不支持chkconfig

解决：

vim编辑redisd文件在第一行加入两行注释保存退出

在第一行添加#chkconfig:2345 90 10

在第二行添加#description: Redis is a persistent key -value database

注释的意思是redis服务必须运行级2，3，4，5下被启动或者关闭，启动的优先级是90，关闭的优先级是10

#!/bin/sh

#chkconfig:2345 90 10

#description: Redis is a persistent key-val database

5.再次执行开启自启动命令

chkconfig redisd on

### 15.3.8 redis 开启关闭命令

service redisd start

service redisd stop

### 15.3.9 进入redis服务器

|  |
| --- |
| [root@localhost ~]# redis-cli  127.0.0.1:6379> |

## 15.4 redis基本数据类型

redis一共分为5中数据类型：String，Hash，LIst，Set，ZSet

String类型是包含很多种类型的特殊类型，并且是二进制安全的。比如序列化的对象进行存储，比如一张图片进行二进制存储，比如一个简单的字符串，数值等等。

set和get方法：

设置值：set name bhz 取值get name （说明设置name多次覆盖）

删除值：del name

## 15.5 String类型

### 15.4.1 keys

|  |
| --- |
| 127.0.0.1:6379> keys \* 输入keys \* 获取所有redis的键值  (empty list or set) |

### 15.4.2 设置键值

|  |
| --- |
| 127.0.0.1:6379> set user admin 设置user 的值为admin  OK |

### 15.4.3获取值

|  |
| --- |
| 127.0.0.1:6379> get user 使用get来获取key的值  "admin" |

### 15.4.4 删除key

|  |
| --- |
| 127.0.0.1:6379> del user 删除key 返回1  (integer) 1 |

### 15.4.5 setnx 不存在设置值，存在返回0

|  |
| --- |
| 127.0.0.1:6379> setnx user 123  (integer) 1  127.0.0.1:6379> setnx user 1234  (integer) 0 |

### 15.4.6 setex 设置有效期，有效期过后返回nil（即null）

|  |
| --- |
| 127.0.0.1:6379> setex fliay 10 22222 设置fliay的值为22222有效期为10秒钟  OK  127.0.0.1:6379> get fliay 10秒过后获取值返回结果为nil  (nil) |

### 15.4.6 setrange 替换字符串

|  |
| --- |
| 127.0.0.1:6379> set fliay 100  OK  127.0.0.1:6379> setrange fliay 2 a 2表示需要替换的位置此处是将2替换成2  (integer) 3  127.0.0.1:6379> get fliay  "10a" 获取值返回结果10a |

### 15.4.7 mset mget 一次性设置多个值和一次性获取多个值

|  |
| --- |
| 127.0.0.1:6379> mset a 1 b 2 c 3 d 4  OK  127.0.0.1:6379> mget a b c d e  1) "1"  2) "2"  3) "3"  4) "4"  5) (nil) |

### 15.4.8 getset 先获取旧值再获取新值

|  |
| --- |
| 127.0.0.1:6379> getset fliay 520  "10a"  127.0.0.1:6379> get fliay  "520" |

15.4.9 incr decr 递增递减

|  |
| --- |
| 127.0.0.1:6379> get fliay  "520"  127.0.0.1:6379> incr fliay  (integer) 521  127.0.0.1:6379> decr fliay  (integer) 520 |

### 15.4.10 incrby decrby 指定值递增 递减

|  |
| --- |
| 127.0.0.1:6379> get fliay  "520"  127.0.0.1:6379> incrby fliay 3  (integer) 523  127.0.0.1:6379> decrby fliay 10  (integer) 513 |

### 15.4.11 append 字符串追加

|  |
| --- |
| 127.0.0.1:6379> get fliay  "513"  127.0.0.1:6379> append fliay hello  (integer) 8  127.0.0.1:6379> get fliay  "513hello" |

### 15.4.12 strlen 获取字符串长度

|  |
| --- |
| 127.0.0.1:6379> get fliay  "513hello"  127.0.0.1:6379> strlen fliay  (integer) 8 |

## 15.6 Hash 类型

Hash类型是String的field和value的映射，或者说是一个String集合，它特别适合存储对象，相比较而言，将一个对象类型存储在Hash类型里要比存储在String类型里占用更少的内存空间，并且方便存取整个对象。

### 15.6.1 hset hget hash键值存储

hset myhset field hello) hset集合，myhash是集合名字 field是字段名 hello是值

|  |
| --- |
| 127.0.0.1:6379> hset user name fliay  (integer) 1  127.0.0.1:6379> hset user age 28  (integer) 1  127.0.0.1:6379> hset user id 1  (integer) 1  127.0.0.1:6379> hget user name  "fliay" |

### 15.6.2 hmset hmget hash键值批量存取

|  |
| --- |
| 127.0.0.1:6379> hmset student name fliay  OK  127.0.0.1:6379> hmset student age 8  OK  127.0.0.1:6379> hmset student id 1  OK  127.0.0.1:6379> hmget student id age name  1) "1"  2) "8"  3) "fliay" |

### 15.6.3 hsetnx 与setnx 大同小异（不存在则添加，存在则返回0）

|  |
| --- |
| 127.0.0.1:6379> hsetnx user id 1  (integer) 1  127.0.0.1:6379> hsetnx user id 2  (integer) 0 |

### 15.6.4 hincrby 和hdecrby 集合递增和递减

|  |
| --- |
| 127.0.0.1:6379> hset teacher age 33  (integer) 1  127.0.0.1:6379> hget teacher age  "33"  127.0.0.1:6379> hincrby teacher age 1  (integer) 34  127.0.0.1:6379> hget teacher age  "34" |

### 15.6.5 hexists 判断是否存在如果存在返回，不存在返回0

|  |
| --- |
| 127.0.0.1:6379> hexists user name  (integer) 0  127.0.0.1:6379> hexists teacher age  (integer) 1 |

### 15.6.6 hlen返回hash集合里的所有键值数

|  |
| --- |
| 127.0.0.1:6379> hlen student  (integer) 3 |

### 15.6.7 hdel删除指定hash的field

|  |
| --- |
| 127.0.0.1:6379> hdel student age  (integer) 1  127.0.0.1:6379> hget student age  (nil) |

### 15.6.8 hkeys 获取hash里面所有的hash

|  |
| --- |
| 127.0.0.1:6379> hkeys student  1) "name"  2) "id" |

### 15.6.9 hvals 获取hash里面所有的value

|  |
| --- |
| 127.0.0.1:6379> hvals student  1) "fliay"  2) "2" |

### 15.6.10 hgetall 获取hash里面所有的key和value

|  |
| --- |
| 127.0.0.1:6379> hgetall student  1) "name"  2) "fliay"  3) "id"  4) "2" |

## 15.7 List 类型

List类型是一个链表结构的集合，主要功能有push,pop,获取元素等，更详细的说，List类型是一个双端链表的结构，我们可以通过相关操作进行集合的头部或者尾部添加删除元素，List的设计非常简单精巧，既可以做栈又可以做队列，满足绝大数需求。

### 15.7.1 lpush 从头部加速元素（栈）先进后出

|  |
| --- |
| 127.0.0.1:6379> lpush list1 1 2 3 4 5 6 7  (integer) 7 |

### 15.7.2 lrange 从结合中取数据

|  |
| --- |
| 127.0.0.1:6379> lrange list1 0 5  1) "7"  2) "6"  3) "5"  4) "4"  5) "3"  6) "2" |

### 15.7.3 rpush 从尾部加入元素（队列）先进先出（取数据同上）

|  |
| --- |
| 127.0.0.1:6379> lpush list1 1 2 3 4 5 6 7  (integer) 7 |

### 15.7.4 linsert 插入元素

|  |
| --- |
| 127.0.0.1:6379> linsert list before 5 999  (integer) 0  127.0.0.1:6379> linsert list1 before 5 999  (integer) 9  127.0.0.1:6379> lrange list1 0 5  1) "7"  2) "6"  3) "999"  4) "5" |

### 15.7.5 lset 将指定下标的元素替换掉

|  |
| --- |
| 127.0.0.1:6379> lset list1 3 2  OK  127.0.0.1:6379> LRANGE list1 0 5  1) "7"  2) "6"  3) "999"  4) "2"  5) "4"  6) "3" |

### 15.7.6 lrem 删除元素，返回删除个数

|  |
| --- |
| 127.0.0.1:6379> LREM list1 1 7  (integer) 1  127.0.0.1:6379> LRANGE list1 0 5  1) "6"  2) "999"  3) "2"  4) "4"  5) "3"  6) "2" |

### 15.7.7 ltrim 保留指定key的值范围内的数据

|  |
| --- |
| 127.0.0.1:6379> ltrim list1 0 3  OK  127.0.0.1:6379> LRANGE list1 0 8  1) "6"  2) "999"  3) "2"  4) "4"  127.0.0.1:6379> |

### 15.7.8 lpop 从list的头部删除元素，并返回删除元素

|  |
| --- |
| 127.0.0.1:6379> LPOP list1  "6" |

### 15.7.9 rpop 从list的尾部删除元素，并返回删除元素

|  |
| --- |
| 127.0.0.1:6379> rpop list1  "4" |

### 15.7.10 rpoplpush 先从尾部删除元素，然后从头部加入元素

|  |
| --- |
| 127.0.0.1:6379> RPOPLPUSH list1 1231  "999"  127.0.0.1:6379> LRANGE list1 0 -1  1) "7"  2) "6"  3) "5"  4) "4"  5) "3"  6) "2"  7) "1" |

### 15.7.11 lindex 返回名称为key的list中index位置的元素

|  |
| --- |
| 127.0.0.1:6379> lindex list1 1  "6" |

### 15.7.12 llen 返回元素个数

|  |
| --- |
| 127.0.0.1:6379> llen list1  (integer) 7 |

## 15.8 set类型和zset类型

set集合是string类型的无序集合，set是通过hashtable实现的，对集合我们可以取交集、并集、差集

set是无序的集合，zset是有序的集合

### 15.8.1 sadd 向名称为key的set集合中添加元素

|  |
| --- |
| 127.0.0.1:6379> sadd user1 aaa  (integer) 1  127.0.0.1:6379> sadd user1 bbb  (integer) 1  127.0.0.1:6379> sadd user1 ccc  (integer) 1  127.0.0.1:6379> SMEMBERS user1 使用SMEMBERS 查询  1) "aaa"  2) "bbb"  3) "ccc" |

### 15.8.2 srem 删除set集合中的元素

### 15.8.3 spop 随机返回删除的key

### 15.8.4 sdiff 返回两个集合的不同元素（哪个集合在前面就以哪个集合为标准）

### 15.8.5 sdiffstore 将返回的不同元素存储到另外一个集合里

### 15.8.6 smove 从一个set集合移动到另一个set集合

### 15.8.7 scard 查看集合里元素个数

### 15.8.8 sismember 判断某元素是否为集合中的元素（1代表集合中的元素 0代表不是）

### 15.8.9 srandmember 随机返回一个元素

### 15.8.10 zadd 有序的集合

|  |
| --- |
| 127.0.0.1:6379> zadd u 1 1  (integer) 1  127.0.0.1:6379> zadd u 2 2  (integer) 1  127.0.0.1:6379> zadd u 5 5  (integer) 1  127.0.0.1:6379> zadd u 4 4  (integer) 1  127.0.0.1:6379> zrange u 0 -1 withscores  1) "1"  2) "1"  3) "2"  4) "2"  5) "4"  6) "4"  7) "5"  8) "5" |

## 15.9 Redis高级命令

### 15.9.1 expire 设置过期时间 使用ttl查看还有多久过期

|  |
| --- |
| 127.0.0.1:6379> set name fliay  OK  127.0.0.1:6379> EXPIRE name 20 设置过期时间为20秒  (integer) 1  127.0.0.1:6379> ttl name  (integer) 5 返回还有5秒过期 |

### 15.9.2 psersist 取消过期时间

### 15.9.3 select 选择数据库

redis一共有16个数据库

|  |
| --- |
| 127.0.0.1:6379> select 1 选择1号数据库  OK |

### 15.9.4 move 将数据移动到其它数据库

|  |
| --- |
| 127.0.0.1:6379> select 1  OK  127.0.0.1:6379[1]> keys \*  (empty list or set)  127.0.0.1:6379[1]> set age 11  OK  127.0.0.1:6379[1]> keys \*   1. "age"   127.0.0.1:6379> move user 1 将数据库0中的user移动到数据库1中  (integer) 1  127.0.0.1:6379> select 1 选择数据库1  OK  127.0.0.1:6379[1]> keys \*  1) "user"  2) "age" |

### 15.9.5 randomkey 随机返回数据里面的一个key

|  |
| --- |
| 127.0.0.1:6379[1]> randomkey  "user" |

15.9.6 rename 重命名key

|  |
| --- |
| 127.0.0.1:6379[1]> rename user users  OK  127.0.0.1:6379[1]> keys \*  1) "users"  2) "age" |

## 15.10 主从复制

### 15.10.1 主从复制

1.一个master可以拥有多个slave

2.多个slave可以连接同一个master之外还可以连接到其他的slave

3.主从复制不会阻塞在同步数据时master可以继续处理client请求

4.提供系统的伸缩性

## 15.11 主从配置

### 15.11.1 配置slaveof从服务器

|  |
| --- |
| vim redis.conf 找到如下行配置  主服务器需要添加密码  requirepass 123456 （主从都要设置）  masterauth 123456 （主从都要设置）  # slaveof <masterip> <masterport>  slaveof 192.168.1.149 6379  配置完成 |

## 15.12 启动添加密码以及关闭修改shell

|  |
| --- |
| service redis start -a 123456  修改redis服务脚本，加入如下所示的红色授权信息即可：  vi /etc/init.d/redis  $CLIEXEC -a "password" -p $REDISPORT shutdown  redis-cli -a 123456 |

## 15.13 哨兵

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 有了主从复制的实现以后，我们如果想对主从服务器进行监控，那么在redis2.5以后提供了一个“哨兵”的机制，顾名思义，哨兵的意思就是监控Redis系统的运行状况，主要功能有两点：   1. 监控著数据库和从数据库是否正常运行 2. 主数据库出现故障的时候，可以自动将从数据库转换为主数据库，实现自动切换。(在其中一台从服务器进行配置) 3. copy文件sentinel.conf 到 /etc/redis中 4. 修改sentinel.conf文件   sentinel monltor mymaster 192.168.1.149 6379 #名称、IP、端口、投票选举次数  sentinel down-after-millisenconds mymaster 5000 #默认1s监测一次，这里配置超时5000毫秒为宕机。  sentinel failover-timeout mymaster 900000  sentinel parallel-syncs mymaster 2  sentinel can-failover mymaster yes   1. 启动sentinel哨兵   redis-server /etc/redis/sentinel.conf --sentinel&   1. 查看哨兵相关信息命令   redis-cli -h 192.168.1.155 -p 26379 info Sentinel   1. 关闭朱富强查看集群信息   redis-cli -h 192.168.1.149 -p 6379 shutdown   1. 实现步骤 2. 进入redis安装的目录将sentinel.conf 拷贝到/etc/redis  |  | | --- | | cp sentinel.conf /etc/redis/ |  1. 查看sentinel.conf 哨兵的端口号是port 26379   将所有日志，数据输出到/etc/redis/目录下   |  | | --- | | dir /etc/redis |  1. 配置哨兵  |  | | --- | | sentinel monitor mymaster 192.168.1.149 6379 2 #名称、IP、端口、投票选举次数 |  1. 配置宕机时间  |  | | --- | | sentinel down-after-milliseconds mymaster 30000 #默认1秒检测一次，这里配置30秒检测一次 |  1. 配置超时时间  |  | | --- | | sentinel failover-timeout mymaster 180000 #配置故障转移时间为18秒 |  1. 配置从节点数量  |  | | --- | | sentinel parallel-syncs mymaster 2 #配置从节点数量 |  1. 启动  |  | | --- | | redis-server sentinel.conf --sentinel & #添加shutdown为关闭 |  1. 查看哨兵命令  |  | | --- | | redis-cli -h 192.168.1.155 -p 26379 info Sentinel | |

## 15.14 事务

redis的事务是非常简单的，使用方法如下：

首先是hi用multi方法打开事务，然后进行设置，这时设置的数据都会放入列表里进行保存，最后使用exec执行，吧数据一次存储到redis中，使用discard方法取消事务。

|  |
| --- |
| 127.0.0.1:6379> multi  OK  127.0.0.1:6379> set a 1  QUEUED  127.0.0.1:6379> set b 2  QUEUED  127.0.0.1:6379> set c 3  QUEUED  127.0.0.1:6379> exec 进行事务提交  1) OK  2) OK  3) OK  127.0.0.1:6379> multi 执行取消事务  OK  127.0.0.1:6379> set aa 1  QUEUED  127.0.0.1:6379> set bb2  (error) ERR wrong number of arguments for 'set' command  127.0.0.1:6379> set bb 2  QUEUED  127.0.0.1:6379> set cc 3  QUEUED  127.0.0.1:6379> discard  OK |

## 15.15 Redis 持久化机制

redis是一个支持持久化的内存数据库，也就是说redis需要经常讲内存中的数据同步到硬盘中保证持久化。

### 15.15.1 snapshotting （快照）默认方式

将内存中以快照的方式写入到二进制文件中，默认为dump.rdb.可以通过配置自动做快照持久化的方式。我们可以配置redis在n秒内如果超过m个key则修改就自动做快照。

snapshotting配置

save 900 1 #900秒内如果超过1个key被修改，则发起快照保存

save 300 10 #300 秒内如果超过10个key被修改，则发起快照保存

save 60 10000 。。。。。。。。。。

### 15.15.2 append-only file（缩写aof）的方式

（有点类似于oracle日志）由于快照方式是在一定时间间隔做一次，所以可能发生redis意外down的情况就会丢失最后一次快照的所有修改的数据、aof对比快照方式有更好的持久化性，是由于在使用aof时，redis会将每一个收到的写明了都通过write函数追加到命令中。当redis重新启动的时候会重置执行文件中保存的写命令来在内存中重建这个数据库的内容。这个文件在bin目录下：appendonly.aof。aof不是立即写到硬盘上，可以通过配置文件修改强制写到硬盘中。

aof设置：

appendonly yes //启动aof持久化方式有三种修改方式

#appendfsync always //收到写命令就立即写入到磁盘

#appendfsync everysec//每秒钟写入磁盘一次，在性能和持久化方面做了很好的折中

#appendfsync no //完全依赖os性能最好持久化没保证

|  |
| --- |
| redis配置文件配置：  appendonly yes 需要将appendonly设置成yes  # The name of the append only file (default: "appendonly.aof")  appendfilename "appendonly.aof" 将日志输出到appendonly.aof中 |

## 15.16 Redis发布订阅

|  |
| --- |
| 多个客户端可以通过subscribe [频道] 进行订阅  服务端可以通过publish [频道] 发布的内容进行发布消息  127.0.0.1:6379> publish cmcc hello  (integer) 0  127.0.0.1:6379> subscribe cmcc  Reading messages... (press Ctrl-C to quit)  1) "subscribe"  2) "cmcc"  3) (integer) 1  1) "message"  2) "cmcc"  3) "hello" |

## 15.17 虚拟内存的使用

redis暂时会把不经常访问的数据从内存交换到磁盘中，腾出宝贵的空间，用于其他需要范围的数据，这需要对vm相关进行配置。（3.0版本是不带vm特性的配置无效）

|  |
| --- |
| 修改配置文件：redis.conf  vm-enabled yes #开启vm功能  vm-swap-file /temp/redis.swap #交换处理的value保存的文件路径  vm-max-mernory 1000000 #redis使用的最大内存上限  vm-page-size 32 #每个页面的大小为32个字节  vm-max-threads 4 #用于执行value对象还如缓存的工作线程数量 |

## 15.18 集群搭建

redis3.0以前，提供了Sentine工具来监控Master的状态，如果Master异常，则会做主从切换，将salve作为master，将master作为salve。其配置也是稍微的负责，并且各方面表现一般。现在redis3.0已经支持集群的容错功能，并且非常简单。集群搭建至少需要三个master

|  |  |
| --- | --- |
| 第一步：创建一个文件夹 redis-cluster 然后在其下面分别创建6个文件夹如下：  1、midkr -p mkdir -p /etc/redis/redis-cluster  2、cd /redis-cluster  3、mkdir 7001、mkdir 7002、 mkdir 7003、 mkdir 7004、mkdir 7005、mkdir 7006  第二步：把之前的redis.conf配置文件分别copy到700\*下，进行修改各个文件内容，也就是对700\*的每一个copy的redis.conf文件进行修改！如下：  1、daemonize yes (后台启动)  2、port 700\* （分别对每个机器的端口号进行设置）  3、bind 192.168.1.149 （必须绑定当前机器的ip，不然入坑）  4、dir /etc/redis/redis-cluster/700\*(指定数据文件存放位置，必须要指定不同的目录位置，不然会丢失数据)  5、cluster-enabled yes 启动集群模式  6、cluster-config-file nodes 700\*.conf(这里700\*最好和port对应上)  7、cluster-node-timeout 5000  8、appendonly yes  第三步：把修改后的配置文件，分别拷贝到各个文件夹下，注意每个文件夹修改端口好，并且nodes文件也要不相同  第四步：由于redis集群需要使用ruby命令，所以我们需要安装ruby  1、yum install ruby  2、yum install rubygems  3、yum install redis（安装redis和ruby的接口）  第五步：分别启动6个redis实例，然后槛车是否启动成功  1、redis-server /etc/redis/700\*/redis.conf  2、ps -el|grep redis 查看是否启动成功  3、netstat -tunpl |grep redis 监控redis  第六步：首先到redis3.0的安装目录下，然后执行redis-trib.rb命令  1、cd /sortware/redis-4.0.6/src  2、./redis-trib.rb create --replicas 1 192.168.1.149:7001 192.168.1.149:7002 192.168.1.149:7003 192.168.1.149:7004 192.168.1.149:7005 192.168.1.149:7006  注意：如果在配置文件中设置了密码需要注释掉否则入坑  如果启动出现错误如果是由于ruby版本过老的话需要重新安装ruby，安装说明详情查看  <https://www.cnblogs.com/carryping/p/7447823.html>  第七步：到此为止集群搭建成功进行验证：  1、连接任意一个客户端即可：./redis-cli -c -h -p(-c 表示集群模式，指定ip地址和端口号)如：  /redis-cli -c -h 127.0.0.1 -p 700\*   |  | | --- | | [root@localhost redis-cluster]# redis-cli -c -h 127.0.0.1 -p 7001  127.0.0.1:7001> keys \*  (empty list or set)  127.0.0.1:7001> set a 123  -> Redirected to slot [15495] located at 127.0.0.1:7003  OK |   2、进行验证：cluster info（查看集群信息）、cluster nodes（查看节点列表）  3、进行数据操作验证  4、关闭集群则需要逐个关闭使用命令：  /redis-cli -c -h 127.0.0.1 -p 700\* shutdown |

## 15.19 集群操作

### 15.19.1 添加节点（一个master 一个slave）

|  |
| --- |
| 1.创建7007与7008两个文件夹  2.将7006中的redis.conf拷贝到7007与7008目录下并且修改对应的Ip  3.修改内容如下：  port:7007/7008  dir /ect/redis/redis-cluster/7007/(7008/)  cluster-config-file nodes 7007(7008).conf |

### 15.19.2 ./redis-trib.rb 中的操作指令

|  |
| --- |
| 1. create ：创建一个集群环境 host1:port1 .... hostN:portN  2.cail ：可以执行redis命令  3.add-node ：将一个节点添加到集群中，第一个参数为新节点的IP，第二个参数为急群众任意一个已经存在的节点port （ip:port）  4.del-node ：删除一个节点  5.reshard ：重新分片  6. check ：检查集群 |

### 15.19.3 新增一个主节点 7007

|  |
| --- |
| 1.使用add-node命令：绿色问新增的节点，红色为已经存在的节点  [root@localhost redis-cluster]# ./redis-trib.rb add-node 192.168.1.149:7007 192.168.1.149:7001  >>> Adding node 192.168.1.149:7007 to cluster 192.168.1.149:7001  >>> Performing Cluster Check (using node 192.168.1.149:7001)  >>> Send CLUSTER MEET to node 192.168.1.149:7007 to make it join the cluster.  [OK] New node added correctly.  查看7007是否已经添加成功  redis-cli -c -h 192.168.1.149 -p 7007  输入 cluster nodes 查看节点信息。  2.为7007分配slot槽 使用redis-trib命令，找到集群中的任意一个主节点（红色位置表现急群众的任意一个主节点对其进行重新分片工作）  ./redis-trib.rb reshard 192.168.1.149:7001  How many slots do you want to move (from 1 to 16384)? 500 分配的槽数量  What is the receiving node ID? f8b5e53085f40ba2e39d96c77bc2f77a2e0ef939 分配槽的ID  Please enter all the source node IDs.  Type 'all' to use all the nodes as source nodes for the hash slots. 所有节点合力分配  Type 'done' once you entered all the source nodes IDs. 只通过当前选择的节点来分配  3.需要执行replicate命令来指定当前节点（从节点）首先需要登录新加的几点7008节点的客户端，然后使用集群命令进行操作，把当前的7008（slave）节点指定到下一个主节点下（这里使用之前创建的7007主节点，红色表示节点Id）  首先需要将7008加入集群 ./redis-trib.rb add-node 192.168.1.149:7008 192.168.1.149:7001  [root@localhost redis-cluster]# redis-cli -c -h 192.168.1.149 -p 7008  192.168.1.149:7008> cluster replicate f8b5e53085f40ba2e39d96c77bc2f77a2e0ef939 （id是7007的id）  OK |

### 15.19.4 移除一个节点

|  |
| --- |
| 删除一个节点7008 slave,输入del-node命令，指定删除节点ip和端口，以及节点id（红色为7008节点id）  ./redis-trib.rb del-node 192.168.1.149:7008 8da2abf5bd7f9f6ac182bbe8c0a7989e6f906e50 |

### 15.19.5 主节点移除

|  |
| --- |
| 1. 删除7007 mastsr 主节点之前，我们需要先把其全部的数据（slot槽）移动到其它节点上去。否则会出现数据丢失。目前删除节点的槽无法做到平均分配，只能分配到一个节点上去。   [root@localhost redis-cluster]#./redis-trib.rb reshard 192.168.1.149:7007  M: f8b5e53085f40ba2e39d96c77bc2f77a2e0ef939 192.168.1.149:7007  slots:0-165,5461-5627,10923-11088 (499 slots) master  How many slots do you want to move (from 1 to 16384)? 499将所有的槽还回去  What is the receiving node ID? 306409c0695403d4f45ffb7c74d55d2b2913db49  Please enter all the source node IDs.  Type 'all' to use all the nodes as source nodes for the hash slots.  Type 'done' once you entered all the source nodes IDs.  输入返还的节点（7007）id 306409c0695403d4f45ffb7c74d55d2b2913db49  输入done否则会入坑！  下一步就可以通过按照 15.19.4的操作将节点进行删除 |

## 15.20 Redis 问题

### 15.20.1.在redis中遇到存储较慢的情况：

原因是因为redis开启了AOF记录日志，为了体现redis的高可靠性。多线程并发的情况下去访问redis 写的性能 也相对来说降低了

### 15.20.2 redis4.0.6版本需要使用Jedis2.8.1

在使用的过程中redis服务器使用了4.0.6的版本，如果jedis使用的版本老了会出现无法连接的情况，版本过于新的话也会出现各种错误，建议使用dedis2.8.1

# Java&redis

Jedis就是redis支持java的第三方类库，我们可以使用Jedis类库操作redis数据库。大体上在3.0之前我们使用Jedis数据库的api比较全面，但是目前java第三方可用库更新比较

## 16.1 需要用到的jar包

|  |
| --- |
| <!-- java使用redis 需要的jar-->  <!-- https://mvnrepository.com/artifact/redis.clients/jedis -->  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.7.2</version>  </dependency> |

## 16.2 连接案例（单节点）

|  |
| --- |
| // 创建单个节点  Jedis jedis = new Jedis("192.168.1.149", 6379);  jedis.auth("123456");// 设置密码  String a = jedis.get("aaa");  System.out.println(a); |

## 16.3 创建多个分片

|  |
| --- |
| //主从，哨兵 使用  private static ShardedJedis shard;  //连接池  private static ShardedJedisPool pool;  @BeforeClass  public static void setUpBeforeClass() {  // 分片  List<JedisShardInfo> shards = Arrays.asList(new JedisShardInfo("192.168.1.159", 6379));  shard = new ShardedJedis(shards);  GenericObjectPoolConfig goConfig = new GenericObjectPoolConfig();  goConfig.setMaxTotal(100);  goConfig.setMaxIdle(20);  goConfig.setMaxWaitMillis(-1);  goConfig.setTestOnBorrow(true);  pool = new ShardedJedisPool(goConfig, shards);  } |

## 16.4 Spring-data-redis 整合

Spring-data-redis集群配置

|  |
| --- |
| <!-- 加载Properties文件 -->  <context:property-placeholder location="classpath:\*.properties"  ignore-unresolvable="true" />  <!-- 配置扫描包 -->  <context:component-scan base-package="com.java.\*,org.redis.\*" />  <!-- 配置集群 -->  <bean id="redisClusterConfiguration"  class="org.springframework.data.redis.connection.RedisClusterConfiguration">  <property name="maxRedirects" value="3"></property>  <!-- 节点 配置 -->  <property name="clusterNodes">  <set>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7001" />  </bean>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7002" />  </bean>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7003" />  </bean>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7004" />  </bean>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7005" />  </bean>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7006" />  </bean>  </set>  </property>  </bean>  <bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">  <property name="maxIdle" value="20" />  <property name="maxTotal" value="100" />  <property name="maxWaitMillis" value="6000" />  <property name="testOnBorrow" value="true" />  </bean>  <!--<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">  <property name="maxIdle" value="${redis.maxIdle}" /> <property name="maxTotal"  value="${redis.maxActive}" /> <property name="maxWaitMillis" value="${redis.maxWait}"  /> <property name="testOnBorrow" value="${redis.testOnBorrow}" /> </bean> -->    <!-- 配置redis连接 -->  <bean id="jeidsConnectionFactory"  class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">  <constructor-arg ref="redisClusterConfiguration" />  <constructor-arg ref="jedisPoolConfig" />  </bean>  <!-- redis Template -->  <bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">  <property name="connectionFactory" ref="jeidsConnectionFactory"></property>  <property name="keySerializer">  <bean  class="org.springframework.data.redis.serializer.StringRedisSerializer" />  </property>  <property name="valueSerializer">  <bean  class="org.springframework.data.redis.serializer.StringRedisSerializer" />  </property>  <property name="hashKeySerializer">  <bean  class="org.springframework.data.redis.serializer.StringRedisSerializer" />  </property>  <property name="hashValueSerializer">  <bean  class="org.springframework.data.redis.serializer.StringRedisSerializer" />  </property>  </bean> |

Test

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration(locations="classpath:applicationContext-context.xml")  public class SpringDataRedisTest {  @Autowired  private RedisTemplate redisTemplate;  @Autowired  private RedisServiceImpl redisService ;  @Test  public void test1(){  //redisTemplate.opsForValue().set("studentname", "jay");  String name = (String) redisTemplate.opsForValue().get("age");  System.out.println(name);  }  } |

# Spring

## 17.1 spring配置

### 17.1.1 引入properties配置文件

|  |
| --- |
| <!--如果你有多个数据源需要通过<context:property-placeholder管理，且不愿意放在一个配置文件里，那么一定要加上ignore-unresolvable=“true"-->  <context:property-placeholder location="classpath:\*.properties" ignore-unresolvable="true" /> |

# 十八、Spring Boot

## 18.1 Spring Boot简介

简化Spring应用开发的一个框架

整个Spring技术栈的一个大整合

J2EE开发的一站式解决方案

## 18.2微服务

微服务：架构风格

一个应用应该是一组小型服务；可以通过HTTP的方式进行胡同。

每一个功能元素最终都是一个可独立的替换和独立升级的软件单元。

## 18.3 SpringBoot 配置

### 18.3.1 parent 模块 pom.xml 配置

|  |
| --- |
| <!-- spring boot -->  <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot</artifactId>  <version>1.5.9.RELEASE</version>  </dependency>  <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-parent -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.9.RELEASE</version>  <type>pom</type>  </dependency>  <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-dependencies -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>1.5.9.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <version>1.5.9.RELEASE</version>  </dependency> |

### 18.3.2 SpringBoot子模块配置

|  |
| --- |
| <dependencies>  <!-- spring boot -->  <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot</artifactId>  </dependency>  <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId> org.springframework.boot </groupId>  <artifactId> spring-boot-maven-plugin </artifactId>  </plugin>  </plugins>  </build> |

### 18.3.3 SpringBoot程序入口

|  |
| --- |
| /\*\*  \* @author fliay  \* @EnableAutoConfiguration 来标注一个主程序 说明是一个springboot应用  \*/  @RestController  @EnableAutoConfiguration  public class Example {  @RequestMapping("/")  String home() {  return "Hello World!";  }  public static void main(String[] args) throws Exception {  //spring应用启动  SpringApplication.run(Example.class, args);  } |

## 18.4 java -jar 启动部署SpringBoot

### 18.4.1 需要添加打包插件

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId> org.springframework.boot </groupId>  <artifactId> spring-boot-maven-plugin </artifactId>  <configuration>  <!-- 指定SpringBoot程序的main函数入口类 -->  <mainClass>com.java.run.Example</mainClass>  </configuration>  <executions>  <execution>  <goals>  <goal>repackage</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> |

18.4.2 启动jar

首先需要将项目打包生成jar文件，然后打开命令窗口，通过java -jar xxx.jar来启动springboot项目

## 18.5 启动器

spring-boot-start:spring-boot 场景启动器，它帮助我们导入了web模块正常运行所依赖的组建。

Spring Boot所有的功能都抽取出来，做成一个个的starters（启动器），只需要在项目里面引入这些starter相关场景的所有依赖都会导入进来。要用什么功能就导入什么场景的启动器。

## 18.6 spring注解

@SpringBootApplication:Spring Boot 应用标注在某个类上说明这个类是SpringBoot的主配置类，SpringBoot就应该运行这个类的main方法来启动SpringBoot应用

@SpringBootConfiguration:Spring Boot的配置类：

标注在某个类上，表示这是一个SpringBoot的配置类;

@Configuration:配置类上来标注这个注解；

配置类----配置文件；配置类也是容器中的一个组件；@Component

@EnableAutoConfiguration:开启自动配置功能；

@AutoConfigurationPackage：自动配置包

@Import(AutoConfigurationPackages.Registrar.class)：spring的底层注解@Import，给容器中导入一个组件；导入的组件由AutoConfigurationPackages.Registrar.class将主配置类（@SpringBootApplication标注的类）的所在包及下面所有子包里面的所有组件扫描到Spring容器；

## 18.7 Spring Boot 配置

SpringBoot使用一个全局的配置文件,配置文件名称是固定的

1. application.properties
2. application.yml（YAML是一个标记语言）

以数据为中心，比json、xml更适合做配置文件

yml语法：

-使用缩进表示层级刮泥

-缩进时不允许使用tab键，只允许使用空格。

-缩进空格数目不重要，只要相同层级的元素左侧对齐即可

-大小写敏感

yml支持的三种数据结构：

-对象：键值对集合

-数组：一组按次序排列的值

-字面量：单个的、不可再分的值

详细参考 [yml语法](https://www.cnblogs.com/lexiaofei/p/6806753.html)

## 18.8 @PropertySource&ImportResource

@ImportResource(locations={"classpath:beans.xml"})//导入Spring的配置文件

@PropertySource("classpath:person.properties")//告诉springBoot读取classpath目录下的指定配置文件

## 18.9 SpringBoot IOC注入

|  |
| --- |
| /\*\*  \* @Configuration:指明当前类是一个配置类；就是用来替代之前的Spring配置文件  \* @author fliay 在配置文件中使用<bean></bean>标签添加组件  \*/  @Configuration  public class MyAppConfig {  /\*\*  \* 将方法的返回值添加到容器中；容器中这个组建默认的id就是方法名  \*  \* @return  \*/  @Bean  public HelloService hellService() {  System.out.println("添加helloService主键");  return new HelloService();  }  } |

## 18.10 配置文件占位符

|  |
| --- |
| ${random.int}  ${username:fliay} 如果username==null那么使用fliay |

具体参考[占位符](https://blog.csdn.net/wu2374633583/article/details/80073271)

## 18.11 Profile

18.11.1 多profile

我们在主配置文件中编写的时候，文件名可以说application-{profile}.properties/yml

默认使用的是application.properties的方式

18.11.2 激活profile配置

在application.properties中添加spring.profiles.active=dev 红色部分为定义文件夹名称的时候{profile}的名称

## 18.12 配置文件加载位置

springBoot 启动会扫描一下位置的application.properties或者application.yml文件作为SpringBoot的默认配置文件。

-file:./config/

-file:../

-classPATH:/config/

-classpath:/

优先级是有高到底，高优先级的配置会覆盖低优先级的配置。

SpringBoot会从这四个位置全部加载著配置文件；互补配置。

## 18.13 配置项目访问路径

server.context-path=/application

## 18.14 debug模式

在properties配置文件中设置debug=true

## 18.15 设置日志的输出级别

在项目中添加slf4j相关的日志jar包

通过在properties配置文件中输入loggin.level.com.java(com.java为自己项目中的类路径)=trace

## 18.16 默认配置

### 18.16.1 全局常规设置

### 18.16.2 指定日志配置文件位置

不指定路径在当前项目下生成springboot.log

可以指定完整的路径

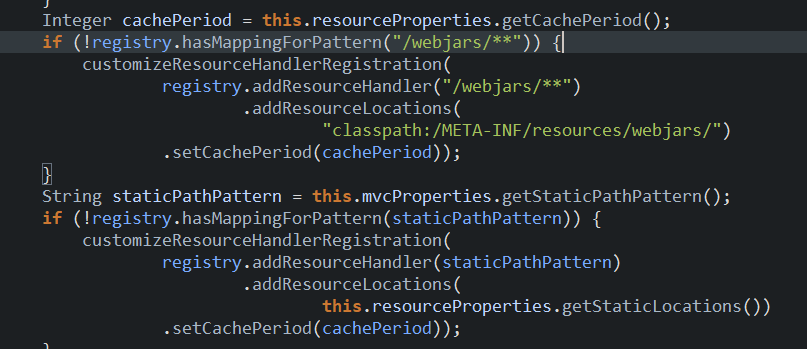
logging.file=springboot.log/logging.file=C:/springboot.log

在当前磁盘的路径下创建spring文件夹和里面的log文件夹，使用spring.log作为默认文件

logging.path=/spring/log

## 18.17 SpringBoot Web

### 18.17.1 SpringBoot对静态资源的映射



|  |
| --- |
| 默认会在webjars目录下查找内容 webjars：是以jar包的方式引入静态文件。 <https://www.webjars.org/> 打开网页，选择需要的maven配置，保存到项目中，启动项目通过访问工程下的webjars下的jquery进行访问  <dependency>  <groupId>org.webjars</groupId>  <artifactId>jquery</artifactId>  <version>3.3.1-1</version>  </dependency>  静态资源的文件夹↓  “/\*\*”访问当前项目的任何资源当前项目的根路径  “classpath:/META-INF/resources”,  “classpath:/resources/”，  “classpath:/static/”,  “classpath:/public/”  favicon.ico放在resources目录下  #设置静态资源路径  spring.resources.static-locations=classpath:/resources |

18.17.2 模版引擎

jsp velocity freemarker thymeleaf

SpringBoot推荐thymeleaf语法更简单功能更强大

1.引入thymeleaf配置

2.SpringBoot配置SpringMVC扩展

|  |
| --- |
| //使用WebMvcConfigurerAdapter可以用来扩展SpringMVC的功能  @Configuration  public class MyMvcConfig extends WebMvcConfigurerAdapter {  @Override  public void addViewControllers(ViewControllerRegistry registry) {  //浏览器发送/admin 请求会来到success  registry.addViewController("/admin").setViewName("hello");  }  } |