# 一、Lombok框架

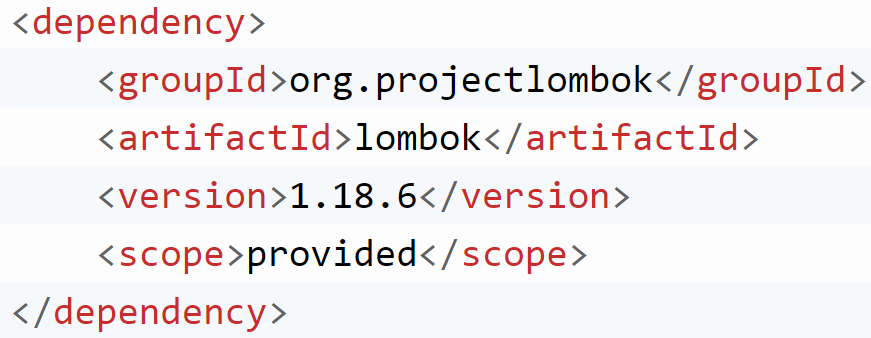
## 1. Lombok的作用

如果用一句话来介绍lombok的话，那么lombok的作用就是通过简单的注解来精简代码以达到消除冗长代码的目的。

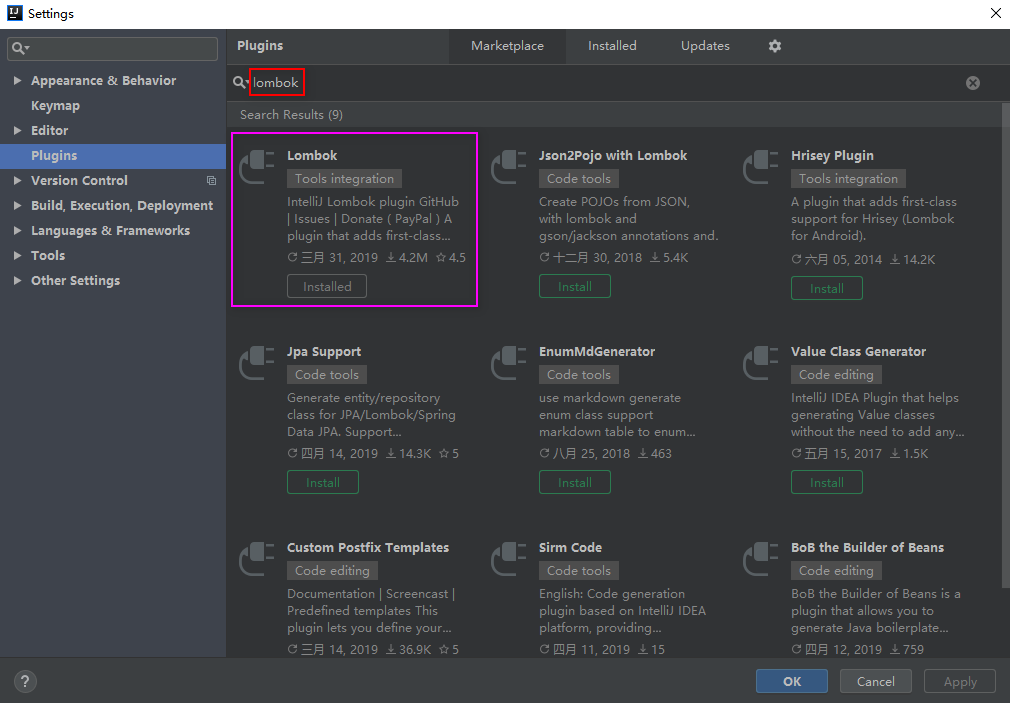
## 2. Lombok的优点

* 提高编码效率；
* 使代码更简洁；
* 消除冗长代码；
* 避免修改字段名字时忘记修改方法名。

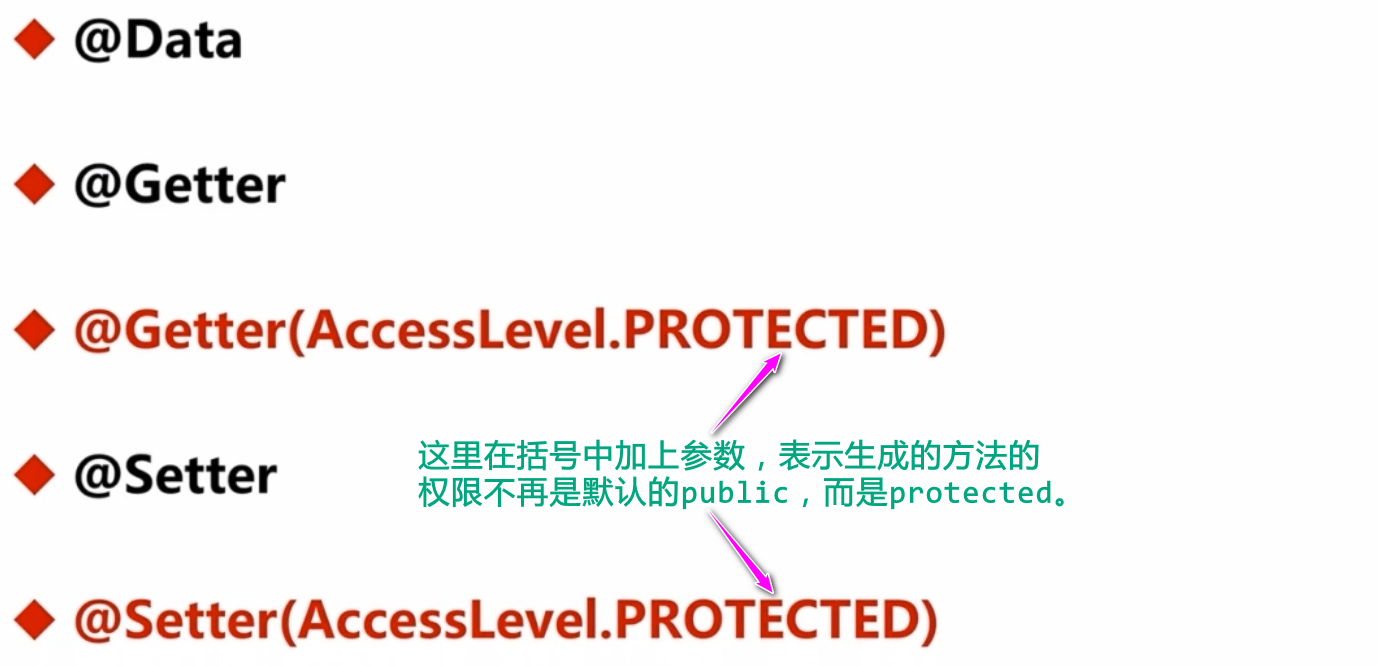
## 3. pom.xml中关于Lombok的依赖

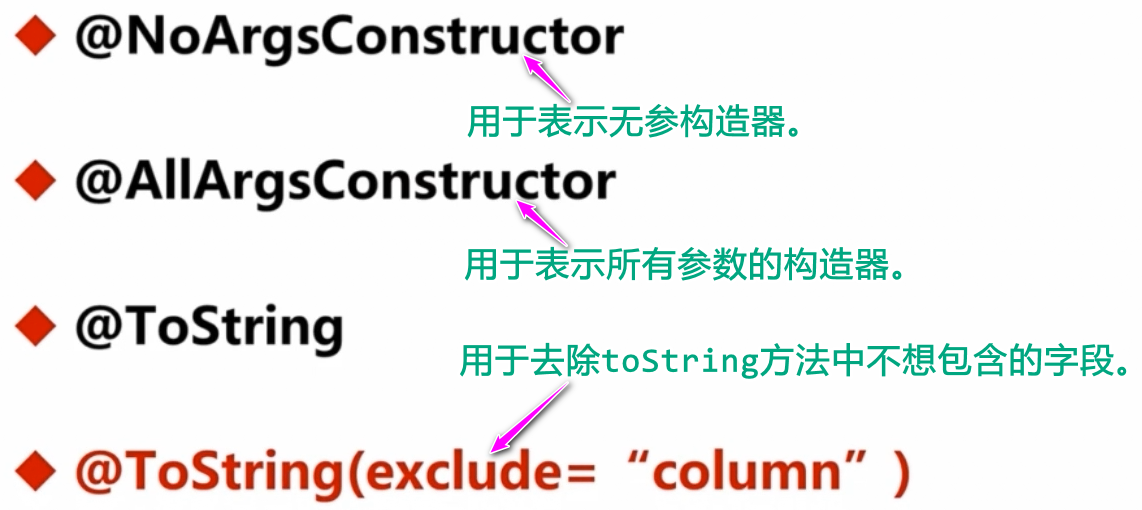


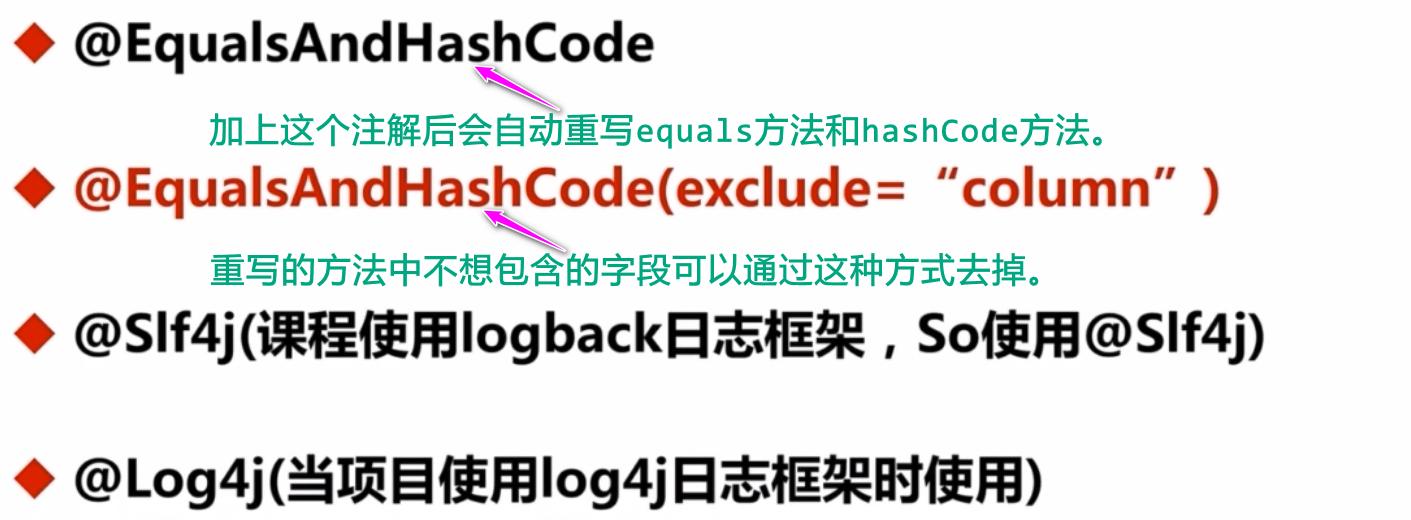
上面的scope标签有没有都可以，都不影响。另外，如果是在IDEA中使用lombok框架的话，对于maven项目，除了要在pom.xml中引入依赖外，还要在IEDA中安装相应的插件，如下图所示。

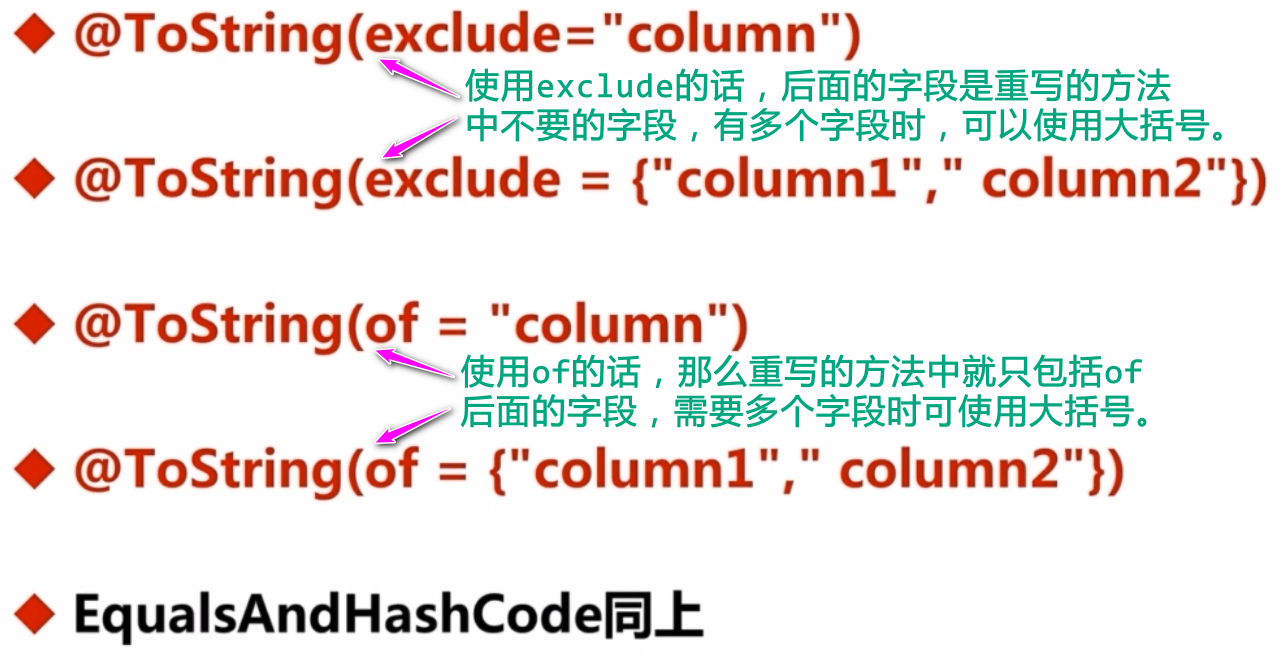


## 4. Lombok的常用注解





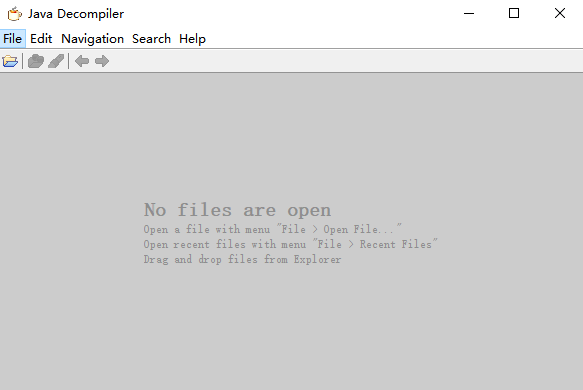






# 二、反编译工具

这里要说的反编译的工具就是java Decompiler，我们可以使用这个工具把class文件进行反编译，从而查看到对应的源码文件，该工具的界面如下。



# 三、maven的环境隔离

## 1. 使用maven环境隔离的原因

自己在本地开发时几乎是不需要使用到环境隔离的，但是在实际的项目开发中进行环境隔离是很有必要的。因为实际的项目开发中是会有多个环境的，不同的环境是会存在一定差异的，为了让这些环境之间互不影响，这时就需要用到Maven的环境隔离了。

实际的项目环境一般包括本地开发环境（Local），开发环境（Dev），测试环境（Beta）以及线上环境（Prod）。由于我是在本地开发，本地开发环境和开发环境是一样的，所以下面在配置的时候，本地开发环境就不配置了。

|  |
| --- |
| 隔离的各个环境之间的各种配置可能存在的一些常见差异：   * FTP服务器相关的配置信息不一样； * 数据库配置不一样； * 支付接口不一样，例如支付宝分沙箱环境和真实环境，我们在开发的时候可以用沙箱环境，但是上线后就要用真实环境了； * 框架的配置文件不一样，例如日志的输出路径等等。 |

以上提到的只是一部分的差异，我们在本地开发时，明明项目是可以正常运行的，但是一到线上就是各种报错，这就是线下的开发环境和线上环境不同导致的。由于环境不同，所以在本地开发的项目，即便在本地能正常运行，但是在发布到线上后，很多配置的内容其实还是需要进行修改的，否则就会报错。

|  |
| --- |
| maven环境隔离解决的实际问题：   * 避免人工修改的弊端，因为人为修改容易犯错； * 轻松分环境编译、打包、部署等等； |

我们把由于环境不同而可能内容不同的配置文件拷贝成多份，比如开发环境一份、测试环境一份、线上环境一份。然后在开发环境下需要进行打包等操作时，就把开发环境中的那份打包进来；如果是线上环境需要进行打包等操作时，就把线上环境的那份打包进来，这样一来就完成了分环境进行打包等操作。如果不是拷贝多份的话，就需要根据环境的不同来修改配置文件的内容，这样就会很容易出错。

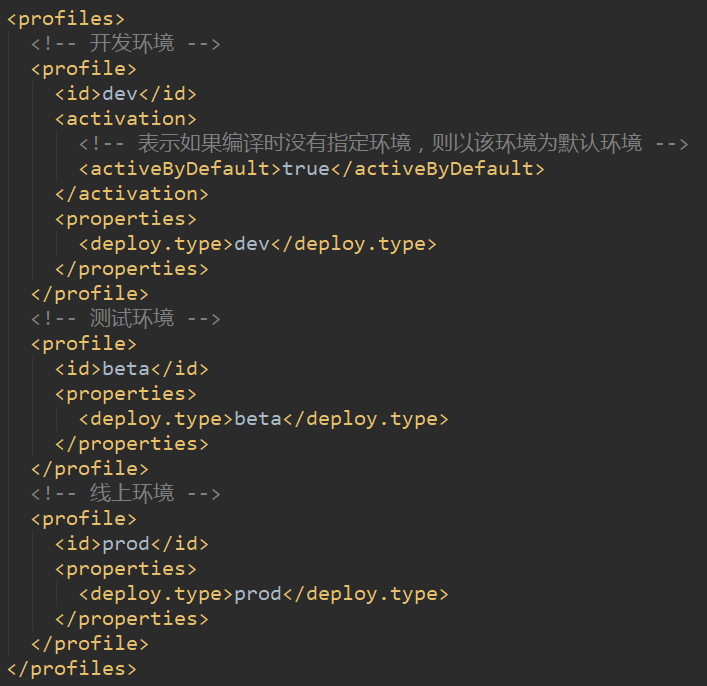
## 2. 配置maven环境隔离的步骤

### 2.1 pom.xml中的配置

①首先在build节点中增加如下配置：

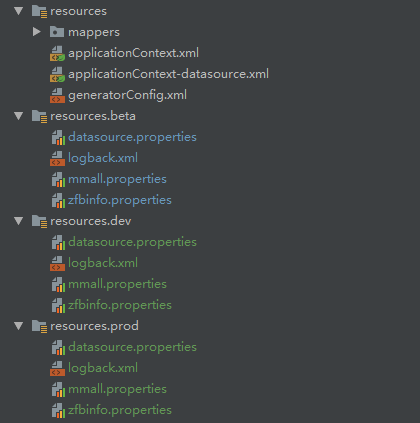


②然后在pom.xml中增加profiles节点，并作如下配置：

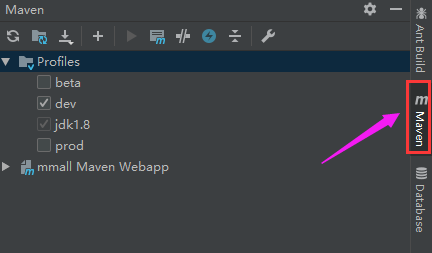


### 2.2 新建针对不同环境的文件夹

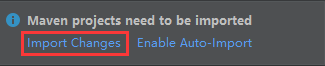
这里把可能会因为环境不同而内容不同的配置文件分别拷贝到不同环境的文件夹下，如下图所示。拷贝好之后，再根据环境来修改对应该环境的目录下相应的配置文件的内容。



### 2.3 在IDEA中设置默认环境



目前使用的环境是开发环境（dev），需要注意的是，这里的对勾分实心的对勾，还有半实心的对勾，像下面jdk1.8前面的对勾就是半实心的，而我们在设置默认环境的时候一定要打实心的对勾。如果我们想要更改默认环境的话，就在这里更改就可以了。每次更改的时候，上图的下面会出现下图中所示的内容。这时要点击我标记的地点，以便能够保存更改。



### 2.4 执行打包等命令

如果要对项目执行打包操作的话，就在命令提示符中输入以下命令：

|  |
| --- |
| mvn clean package -Dmaven.test.skip=true -Pdev |

这个命令是针对开发环境的，如果是测试环境的话，以上命令其他部分不变，最后面的部分变成-Pbeta，同理，线上环境的话，就变成-Pprod。如果是使用以下命令的话，就代表使用默认环境进行打包，而之前在pom.xml中配置的默认环境是开发环境（Dev）。

|  |
| --- |
| mvn clean package -Dmaven.test.skip=true |

# 四、Tomcat集群与nginx负载均衡

## 1. 基本概念

我之前在作总结的时候，有专门讲过集群、分布式以及负载均衡的区别。我对nginx负载均衡的理解就是，当浏览器中的请求到达nginx的时候，由于与nginx相关联的tomcat服务器有很多，所以nginx会把用户的请求随机分发给其中的某个服务器来处理，由于请求是随机分发给任意一个服务器来处理的，所以不会出现过多请求都交给某一个服务器来处理的情况，这一行为就是nginx的负载均衡。这样做的好处是不会使某一个服务器的负载过重，从而就达到了负载均衡的目的。不过由于每个服务器的性能可能不一样，所以我们也可以通过权重的方式，把更多的请求分发给性能较好的服务器来处理。

与nginx相关联的多个tomcat服务器就叫做tomcat集群，其实集群可以理解成是很多人在干同一件事。就是由于请求太多了，一个服务器不够用，所以才会用多个服务器一起来处理请求，这其实就是集群。下面就主要讲一些关于集群相关的知识。

## 2. Tomcat集群的作用和实现原理

### 2.1 Tomcat集群的作用

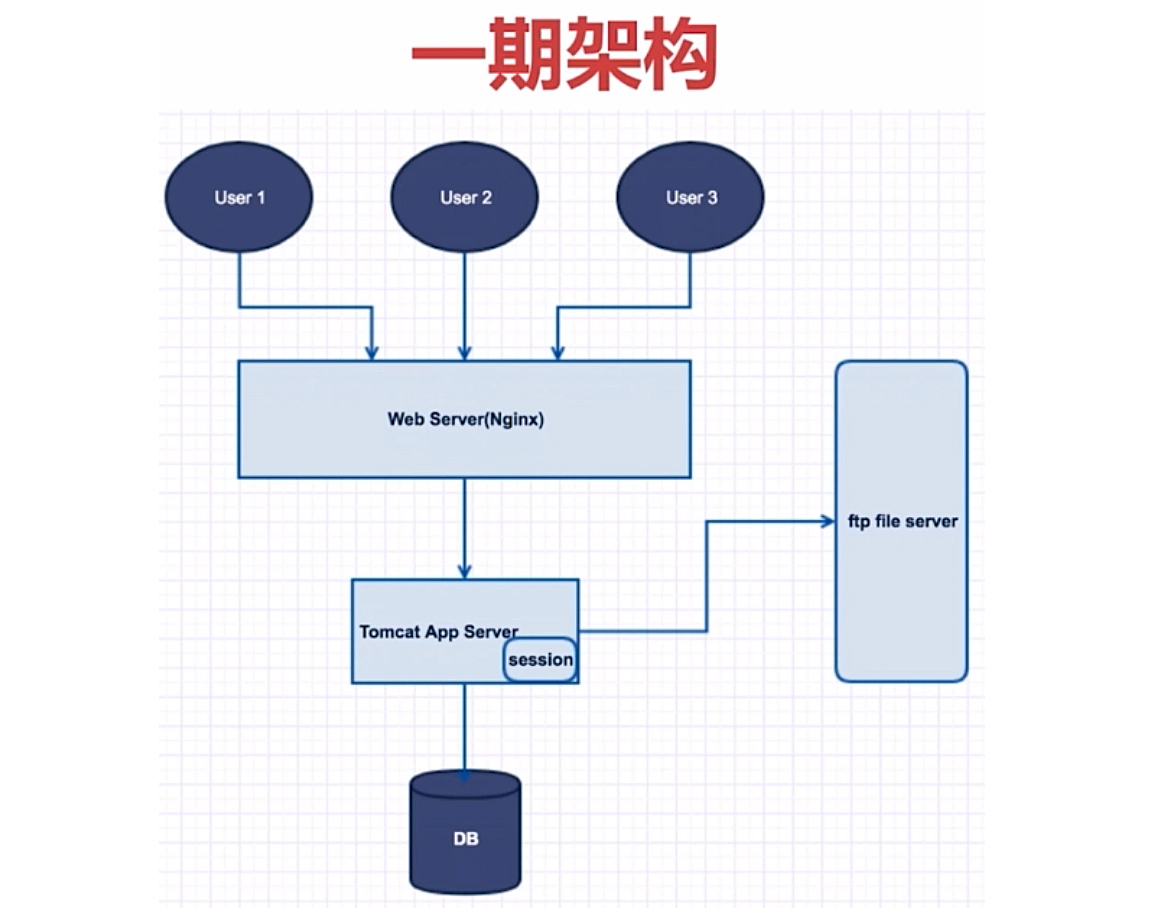
* 提高服务的性能，并发能力以及高可用性；
* 提供项目架构的横向扩展能力。

### 2.2 Tomcat集群的实现原理

* 通过nginx负载均衡进行请求转发。

注：tomcat集群不一样就只有以上提到的两个作用，也不一定就上面提到的一个实现原理，我这里记录的都只是老师在课上提到的。

## 3. 一期架构和二期架构对比





## 4. nginx负载均衡的常见策略







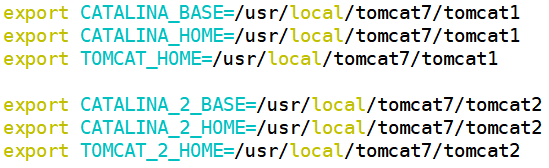
以上关于权重这一策略，我这里特别讲一下，该项目中使用的就是这种策略。上图中在使用weight的时候，分别使其等于15和10。这里需要说明的是，权重表示的只是概率问题，weight=15表示请求分到该服务器上的概率是weight=10时的1.5倍。并不是说有25个请求就一定有15个被分到weight=15所在服务器上。

## 5. 集群和负载均衡的具体步骤

在叙述详细步骤前先说一下，我下面步骤中演示的tomcat和nginx都是在linux环境下，不过在访问的时候，是在windows下的浏览器中进行访问的。

### 5.1 配置环境变量

首先在linux的/usr/local/tomcat7目录下增加两个tomcat，分别命名为tomcat1和tomcat2，然后执行vim /etc/profile打开profile文件，打开后在该文件中增加下图中这些内容。



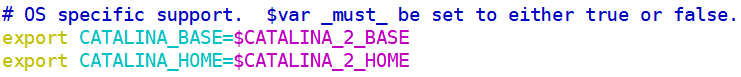
以上配置的意思是在集群中设置两个tomcat服务器，如果还想配置更多的话，就按照图中的方式依次加上就可以了。保存退出后执行source /etc/profile使配置生效。如果不执行这个命令，而是重启虚拟机的话，也可以生效。

注：如果之前在profile文件中配置过tomcat，记得给注释掉，以免对这里配置tomcat集群产生影响。

### 5.2 修改tomcat2中的catalina.sh文件

这里tomcat1的catalina.sh文件不进行修改，只修改tomcat2的catalina.sh文件，具体位置为：/usr/local/tomcat7/tomcat2/bin/catalina.sh。

打开tomcat2的bin目录下的catalina.sh文件后，在下图的注释下面新增两行内容，新增的具体内容如下图所示。新增成功后，保存退出。

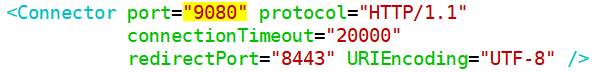


### 5.3 修改tomcat2中的server.xml文件

修改完catalina.sh后，来到tomcat2的conf目录下，修改该目录下的server.xml文件，主要是修改三个端口。首先是把下图中的8005端口改成9005。



然后是把下图中的8080端口改成9080端口。



最后是把下图中的8009端口改成9009端口。



修改完成之后，保存退出即可。需要提一下的是，以上三个端口不一定非要改成图中的那三个端口，改成其他的也行，但是必须要保证端口没有被占用。

### 5.4 分别进行访问测试

启动tomcat1的bin目录下的startup.sh以及tomcat2的bin目录下的startup.sh。在启动之前不要忘记在linux的防火墙中开放这里修改的9080端口。另外，为了看出两个tomcat的区别，可以把tomcat2的webapps/ROOT目录下的tomcat.png图片换成其他的图片，但是名字还叫这个，以便观察区别。修改好之后，就可以在windows下的浏览器中分别进行访问了。

由于我linux系统的ip是192.168.31.18，所以访问地址为192.168.31.18:8080和192.168.31.18:9080。

### 5.5 修改windows以及linux中的hosts

由于tomcat以及nginx都安装在了linux中，而我是在windows的浏览器中进行访问的，所以必须要修改windows的hosts文件，在windows的hosts文件中添加下图中所示的内容即可，下图中的192.168.31.18是linux系统的ip地址。

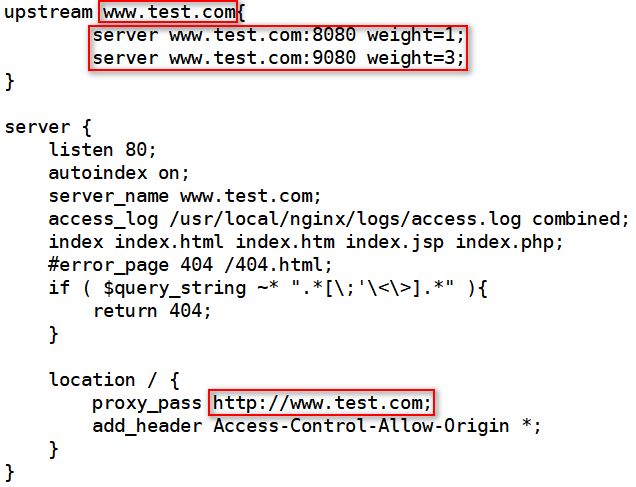


至于为什么要修改linux中的hosts文件，这个我在下面会讲到，其实不修改linux中的hosts文件也是可以的。如果要修改的话，步骤如下。首先通过vim /etc/hosts命令打开hosts文件，然后在hosts文件中加上下图中内容，保存退出即可。



### 5.6 配置nginx的conf文件

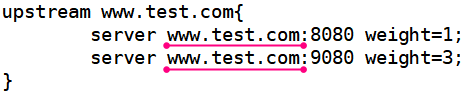
由于在一期的项目中，已经在/usr/local/nginx/conf目录下创建了vhost目录，且已经在/usr/local/nginx/conf目录下的nginx.conf文件中使用include包含了vhost目录中的所有文件，所以这里直接操作vhost目录中的文件即可。首先在vhost目录下新建一个www.test.com.conf文件，然后给该文件添加内容，添加的内容如下。



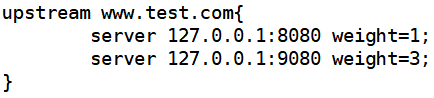
添加完成之后，保存退出。如果之前没有启动nginx的话，启动nginx。如果已经启动了的话，就在/usr/local/nginx/sbin目录下通过./nginx -s reload命令重启nginx。然后还可以通过./nginx -t命令测试是否重启成功。

Nginx启动成功后，由于之前已经启动了tomcat1和tomcat2，所以可以在windows的浏览器中访问www.test.com，然后不停刷新页面，如果之前对tomcat2的tomcat.png文件进行过修改的话，会发现出现的页面是不固定的。因为通过nginx会把请求随机分发给我们配置的对应8080端口的tomcat1以及对应9080端口的tomcat2。而tomcat2显示的页面中的图片又进行了修改，所以一直刷新的话，页面其实是在两个不同的页面之间进行切换。哪个页面出现的概率大，取决于上面conf文件中配置的权重。

这里说一下为什么要修改linux中的hosts文件。www.test.com.conf文件内容的部分截图如下。

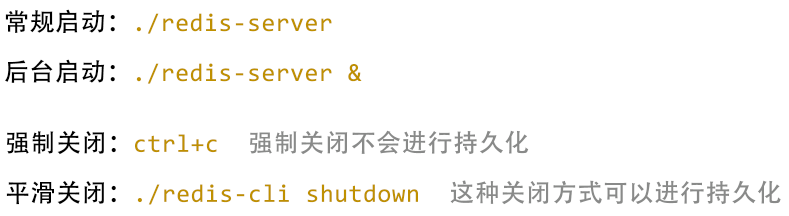


该图中的server后面之所以可以使用www.test.com:8080或www.test.com:9080，就是因为在linux中修改了hosts文件。如果没有修改hosts文件的话，这里这样写是访问不到内容的。如果不想修改linux中的hosts文件也行，不过就不能再像上图中那么写了，而是要写成下图中这样。



# 五、Redis快速入门

## 1. redis的启动和关闭



使用带&的方式启动redis服务后，即便使用ctrl+c进行强制关闭，后台还是处于启动状态，通过查看进程可以看出来。



以上启动redis服务和客户端都是使用默认的6379端口启动的，如果不想使用默认端口的话，可以使用如下启动方式。



除了以上提到的启动和关闭的方式外，还可以通过修改配置文件的方式启动。

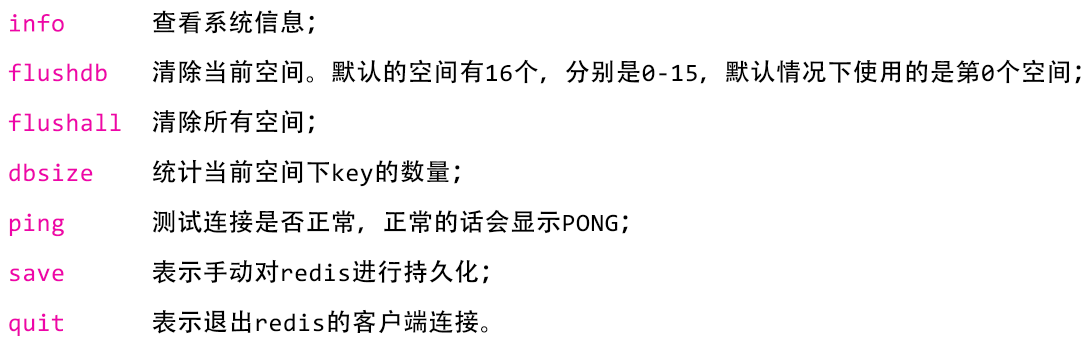


很多时候，我们启动客户端其实是要输入密码了，下面这种方式就是带密码启动。

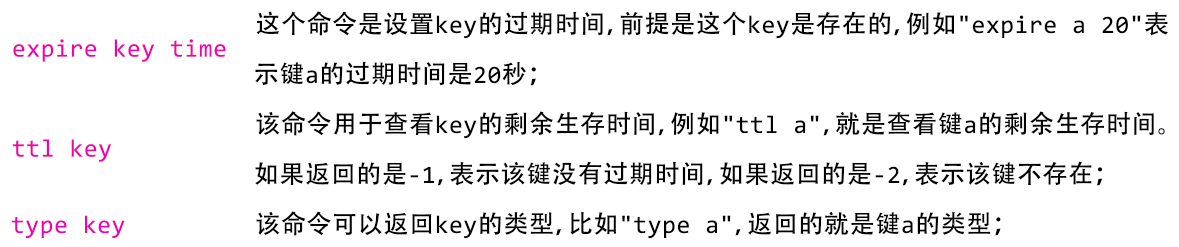


## 2. redis常用的命令

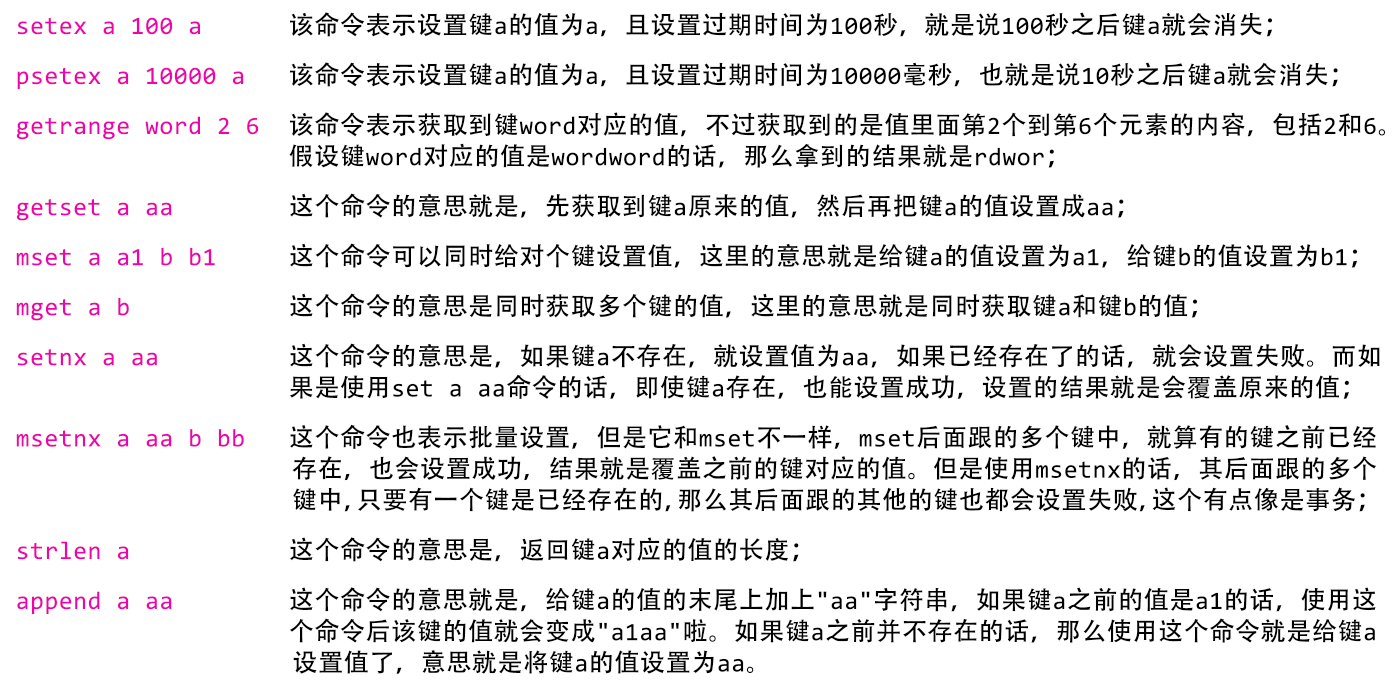
### 2.1 常用的系统命令



### 2.2 redis常用的键命令



### 2.3 redis数据结构之String字符串



关于redis的其他类型的数据结构这里就不再讲啦，因为在这个项目之前学redis的时候我作过总结，上面都有讲过，具体可以参考那时候作的总结。

# 六、单点登录及其相关知识点

## 1. 使用jedis连接redis

### 1.1 连接池的构建

该项目中创建了一个RedisPool类，通过这个类完成对连接池的构建，在构建jedis连接池的时候，主要涉及JedisPool和Jedis这两个类。在RedisPool类中还设置了连接池的一些参数，如最大连接数、最大空闲数、最小空闲数、连接redis的ip地址、端口号以及超时时间等。另外，RedisPool类中还设置了一个公开的获取Jedis的方法。

### 1.2 jedis常用API的封装

连接池构建好之后，我们就可以通过jedis连接池连接redis了。在RedisPool类中通过JedisPool类是可以获取到Jedis类的，而如果想要对redis进行操作，就需要用到Jedis类的一些API，这里把常用的一些方法都封装在了RedisPoolUtil类中了。比如使用expire方法设置redis中key的有效期、使用set方法添加一个键值对、使用setex方法添加一个带有效期的键值对、使用get方法获取key的value值、使用del方法删除一个键值对等。

由于RedisPool类已经被应用在了RedisPoolUtil类的所有方法中，所以直接调用RedisPoolUtil类中的相关方法就可以连接上redis并对其进行相应的操作。

## 2. JsonUtil类的封装

这里要说的JsonUtil类是一个相当强大的工具类，也是可以直接应用到工作中的一个类。这个类中主要封装的是java对象和json字符串之间相互转换的方法，比如java对象转成json字符串的方法、json字符串转成java对象的方法等。由于老师把各种可能性都考虑到了，并且都封装进了这个类中，所以这个方法的复用性很强。

在这个工具类中有一个相当重要的类，它就是ObjectMapper类。ObjectMapper是Jackson提供的一个类，作用是实现java对象与json格式之间的相互转化。它可以将对象转换成json字符串，反之亦然。JsonUtil这个工具类就是靠着ObjectMapper类才完成java对象与json字符串之间的相互转换的。

我大致说一下这个工具类在该项目中主要的应用场景。在集群状态下，如果要实现单点登录的话，用户登陆的session信息就不能放在tomcat服务器中，而是要放在redis中。但是在redis中，是没法储存一个对象的，而用户的登录信息又是一个user对象，这时候如果想把这个登录信息储存到redis中，最好的办法就是把这个对象转成json字符串，然后再进行存储。这时候这个JsonUtil工具类就派上用场啦，可以在储存用户登陆信息的时候，先使用工具类把这个user对象转成json字符串储存在redis中，然后当想获取这个用户的登录信息的时候，再使用这个工具类把json字符串转成对象返回给前端即可。

## 3. CookieUtil类的封装

这个工具类中主要封装了三个方法，它们的作用分别是新增一个cookie信息并响应给浏览器、获取浏览器的请求中的cookie信息、删除浏览器中指定的cookie信息。

如果想实现用户登陆信息的共享，离不开这个工具类。由于在tomcat集群环境下，会有多个tomcat服务器，而我们又不知道用户的登录请求会被nginx分发给哪个服务器，而不同的服务器中的信息又不能共享。如果想要解决这个问题，就需要把用户的登录信息储存到redis中。

当用户登录的时候，我们可以获取到用户的登录信息以及这次会话的session的id值，这时候我们可以把session的id值作为键值对的key，把用户的登录信息转json字符串后的内容作为键值对的value储存到redis中。与此同时，还要将session的id值传入到CookieUtil类中要响应给浏览器的方法中。在这个方法中，首先要创建一个cookie对象，这个cookie的name值可以自己定义一个，而value值就是传入的session的id值，然后给这个cookie对象的一些常用属性设置好值之后就返回给浏览器。

当用户再次访问的时候，不管用户的请求被分发到哪个服务器中都没关系。因为这个工具类的另一个方法可以获取到请求中所有的cookie，然后这个方法会对这些cookie进行判断，看有没有哪个cookie的name值是我们之前自己定义的那个name值，如果找到了这样的cookie，那么就返回这个cookie的value值，而这个value值就是我们之前存的session的id值。然后再把这个value值作为键值对中的key值到redis中获取这个键对应的value值，再把这个字符串形式的value值转成对象，如果这个对象不为空的话，就说明该用户已经登录啦，这样一来，就在集群环境下获取到了用户的登录信息啦，同时也解决了用户登陆信息共享的问题。

如果用户执行了退出登录的操作，那么我们就调用CookieUtil工具类中另一个方法删除浏览器中的cookie信息，同时也删除redis中该用户的登录信息即可。

## 4. 重置session有效期

这里是使用一个过滤器类来完成对session有效期的重置的，这里说一下为什么要重置session的有效期。由于session的有效期是30分钟，所以这就导致一个结果，那就是即使实现了session共享，但是当用户登陆30分钟后，登陆信息就会失效，就又要重新登录了，这样用户体验就很不好了。这时候可以引入一个可以拦截所有以"\*.do"结尾的请求的过滤器类，而使用过滤器类的作用就是，在用户登陆后，只要用户和网站进行交互，就会拦截到交互的请求，然后把session的过期时间重置成30分钟。这里说的交互分很多种，比如用户查看了商品详情，比如用户把商品加入购物车等等。

至于为什么非要进行下重置，我这里再举个例子。假设用户现在登录了，这时候session还有30分钟就会过期。假设用户登陆后过了5分钟才与网站进行交互，比如查看了商品详情什么的。如果不重置session的过期时间，那么再过25分钟，这个用户的登录信息就会失效，就要重新登录了。但是如果重置了，那么session的有效期就又会变成30分钟，也就是说，重置后的session就又变成是30分钟后失效，而不是25分钟。如果用户再与网站进行交互，就又会重置session的过期时间。这样一来，只要用户在30分钟内至少和网站交互一次，那么该用户的登录信息就永不会失效。

## 5. 总结

关于单点登录这一块，主要讲到了jedis的一些相关知识，还有就是一些工具类的封装。其实我觉得只要解决了在集群环境下session共享的问题，那么就能够实现单点登录了。由于是集群环境，所以肯定不止一个tomcat服务器，当用户的登录请求到达nginx的时候，我们并不知道nginx把请求转发到哪个tomcat服务器中啦。假设用户的登录请求被转发到tomcat1中啦，而请求中是携带有用户的session信息的，而用户的登录信息都在这里面。如果不能实现session共享，那么万一用户又查看了下个人信息，而查看个人信息的这个请求正好是由tomcat2来处理的，而tomcat2中是没有用户的登录信息的，这时候就会认为该用户没有登录，然后提示用户进行登录。可是用户刚登录过，然后查看一下个人信息居然还要再登录一下，这样用户体验就很不好啦。

而实现session共享的方法就是，在用户登录的时候，不把用户的session信息保存在任何一台tomcat服务器中，而是把它保存在redis中。假设用户的登录请求是交由tomcat1来处理的，这时候把请求中携带的用户的session信息保存到redis。假设用户查看个人信息的请求是由tomcat2来处理的，这时首先会在redis中寻找看有没有该用户对应的session信息，如果能找到，就说明该用户已经登录，就不用再登录啦；如果没找到，就说明该用户还没有登录，就提示用户登录。

# 七、Redis分布式

## 1. redis分布式算法原理

### 1.1 传统分布式算法

传统的分布式算法主要是通过取模的方式来存储数据的。假设现在有四个redis，分别是redis0、redis1、redis2、redis3。这里使用redis0而不是使用redis1作为第一个redis是为了在下面举例的时候方便理解。下面会讲到对4取模，毕竟对4取模的话，得到的值只能是0、1、2、3中的某一个，然后这4个数正好可以和redis0、redis1、redis2、redis3中的数字进行对应，这样的话也可以更方便理解一点。然后再假设有20个数据，每个数据在取模前都会先进行hash散列，为了方便理解，假设这个20个数据在经过hash散列之后就变成了从1到20这20个数字。由于传统分布式算法是通过取模存储数据的，而这里有四个redis，所以在散列之后每个数字都要和4进行取模，这里提到的取模其实就是拿散列后的这些数字分别去除以4，然后得到的余数就是取模的值。由于是和4进行取模，所以取模后的值只可能是0、1、2、3中的某一个。如果取模的值是0，那么就把这个数据存到redis0中；如果取模的值是1，就存到redis1中；如果取模的值是2，就存到redis2中；如果取模的值是3，就存到redis3中。然后得到的结果如下图所示。



从图中可以看出，这20个数字都已经被分别存到了相应的redis中啦。这20个数字都是经过hash后的结果，所以也就是说，哪个数字存到哪个redis中，就代表这个数字在经过hash前的哪个数据存到哪个redis中。举个例子，比如5这个数字，从图中可以看出，它是被存到redis1中了的，那么就代表5这个数字在进行hash前对应的那个数据被存到了redis1中。

然后可能由于业务需要，4个redis已经不够用了，所以就需要再新增一个redis，比如新增了一个redis4。然后这时候就有5个redis了，所以这时候就需要把这20个数字重新对5取模然后再进行储存，如图所示。



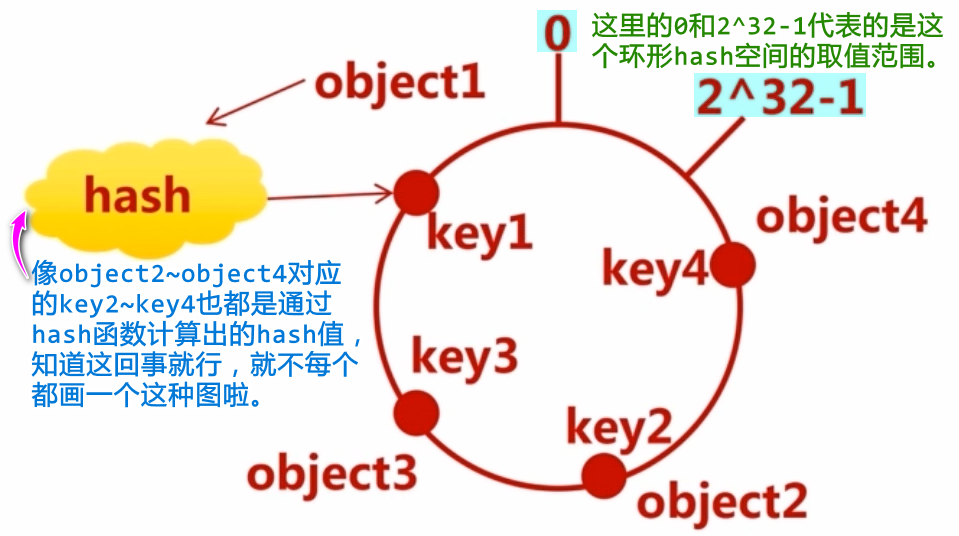
从图中我们可以看出，在新增了一个redis后，这20个数字中只有图中标记为蓝色部分的那4个数字的储存位置是在新增一个redis前后保持不变的，它们之前储存在哪个redis中，在新增了一个redis后还是储存在哪个redis中。所以这个命中率就是20%，即4/20=20%。

这个传统的分布式算法存在一个很大的问题，从上面两张图就可以看出，在新增了一个redis后，很多数据的储存位置都变了，这也就意味着改变redis的数量对所有数据的影响都比较大，有一种牵一发而动全身的感觉，我们肯定不希望是这种情况，我们希望的情况是，改变redis的数量只会影响到很少的数据，下面的Consistent hashing一致性算法就可以解决这个问题。

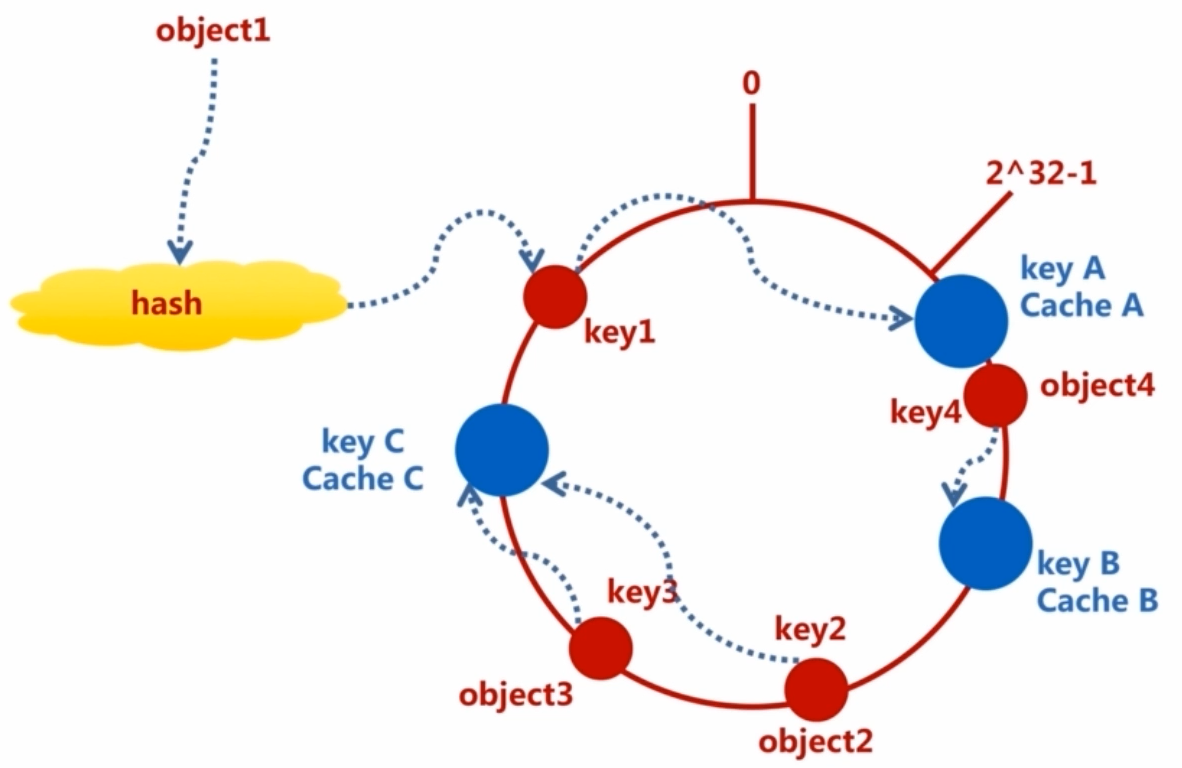
### 1.2 Consistent hashing一致性算法

这里的Consistent hashing一致性算法其实就是一致性hash算法。在讲解这个知识点之前，需要先讲解一下环形hash空间。通常hash算法都是将数据作为value值映射在一个32位的key值当中，如果把这个key值形成的数轴卷起来，就会变成一个首尾相接的圆环，即环形hash空间，而它的取值范围是0~232-1。

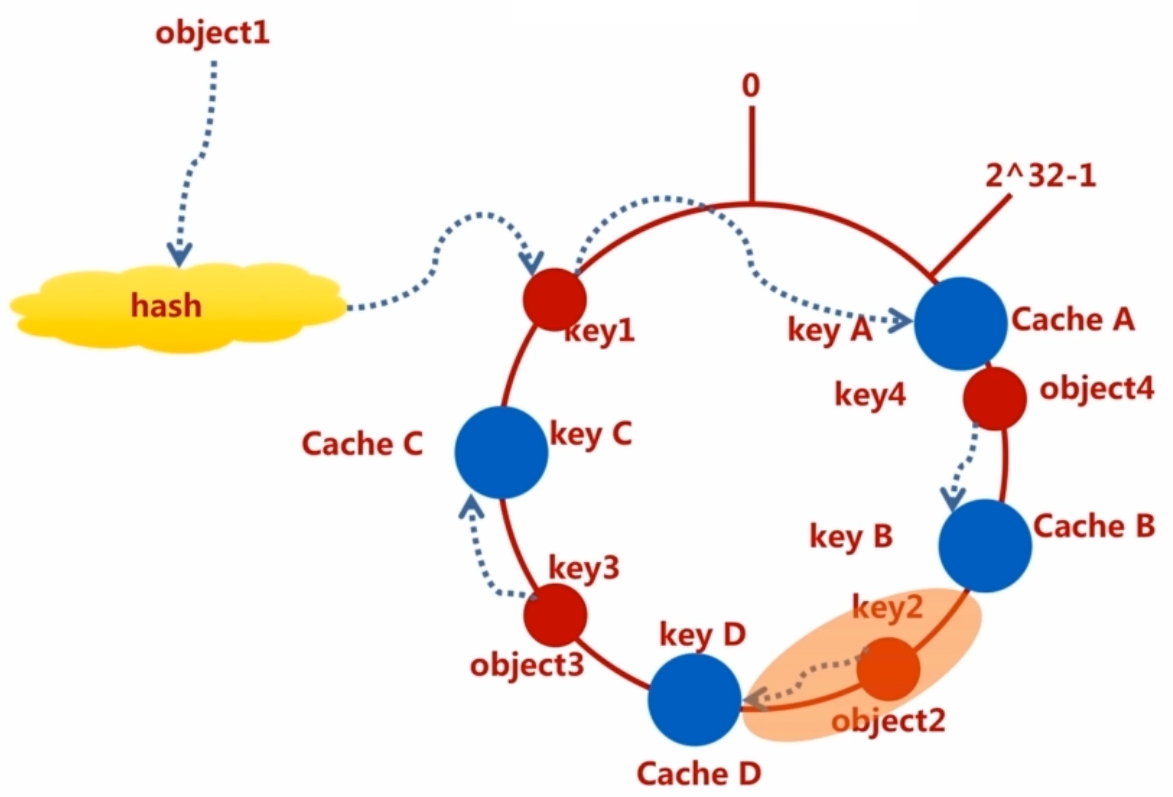
假设现在有四个对象，即object1~object4，通过hash函数可以计算出他们的hash值，而计算出的4个hash值肯定会落在环形hash空间上，假设object1~object4对应的hash值是key1~key4，那么结果就如下图所示。



然后将cache也映射到这个环形hash空间上，并且使用相同的hash算法。假设有3个cache且它们对应的hash值分别是keyA、keyB、keyC的话，那么如下图所示。我个人的理解是，这里的object就是需要被储存在redis中的数据，而cache就是redis。

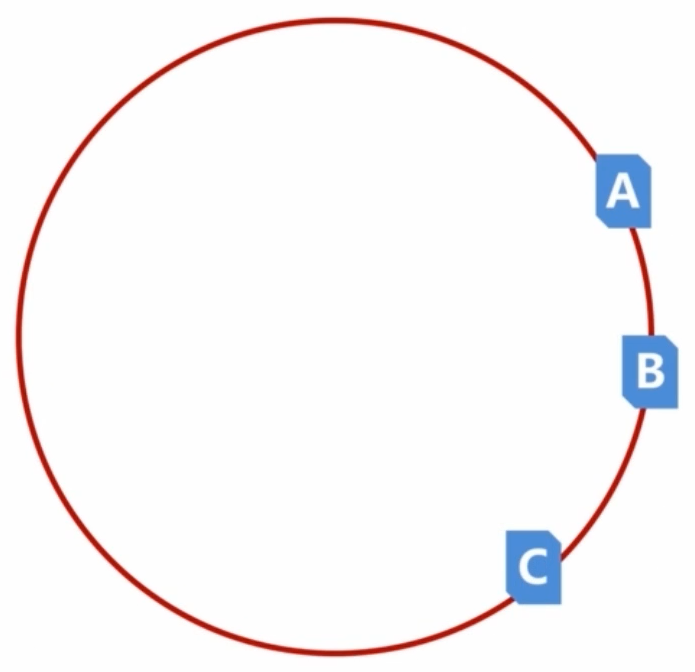


关于上面这个图，是这个意思。Key1沿着圆环顺时针出发，首先会遇到keyA，然后key1就会被存到这个蓝色的keyA中。同理，key4会被存到keyB中，而key2和key3都会被存到keyC中。如果这时候因为业务需要，新增了一个Cache的话，假设这个Cache对应的hash值是keyD的话，那么结果如图所示。

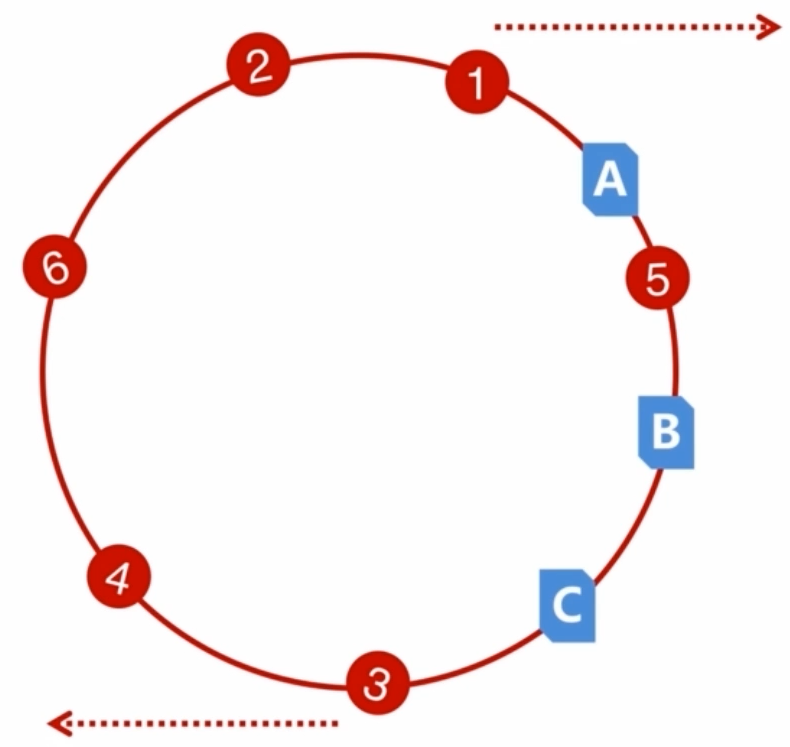


拿这个图和上个图进行对比就可以发现，当新增一个Cache的时候，只有key2的储存位置从原来的keyC变成了现在的keyD，其它数据的储存位置都是不变的。就算还有key5、key6等更多数据落在圆环上也不要紧，因为新增一个CacheD后对整个圆环来说，影响的范围其实就只有图中标记的在key2附近的那一段而已。由此可见，和传统的分布式算法相比，这种一致性hash算法的命中率更高，这也正是我们想要的结果。

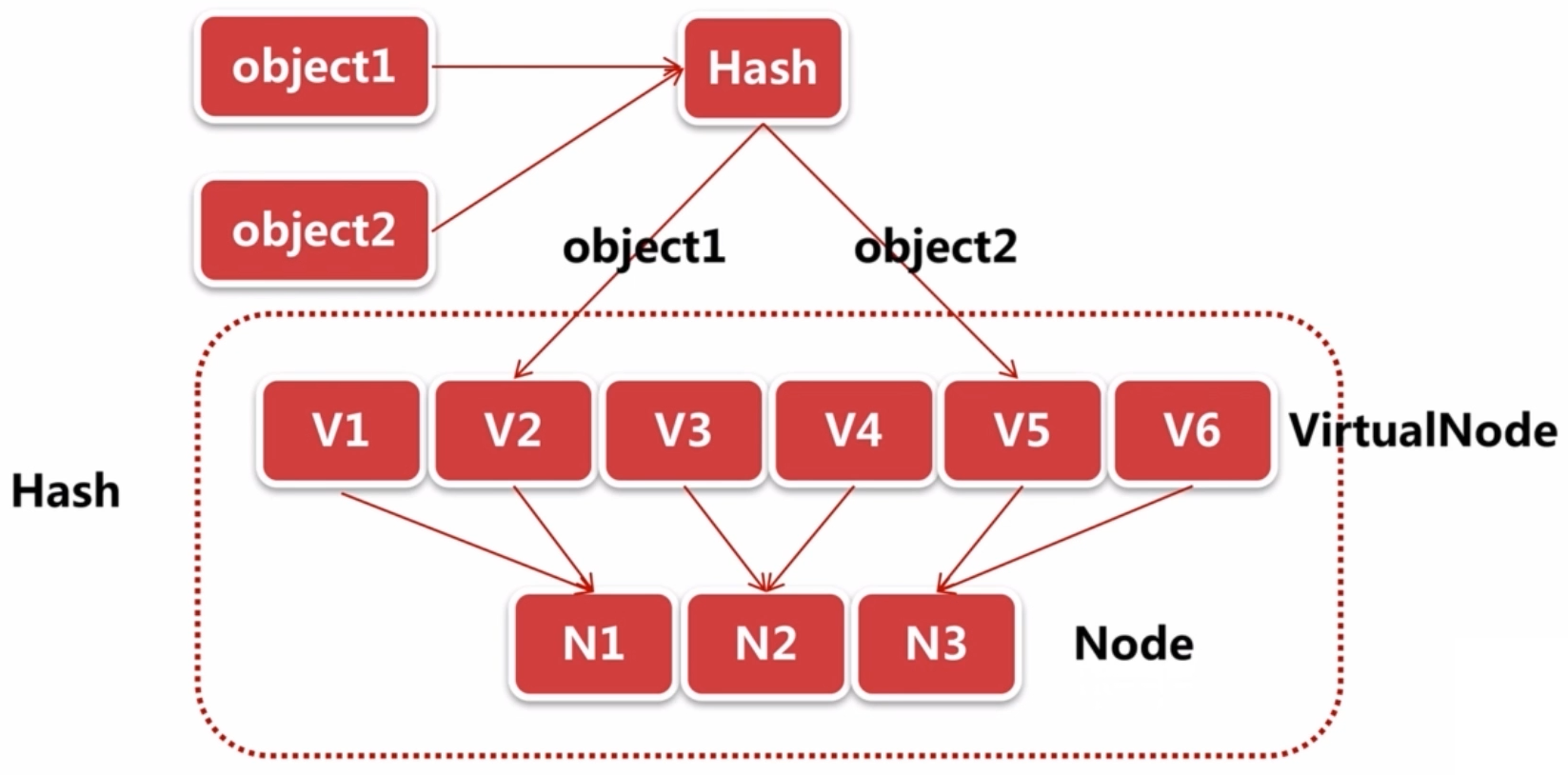
但是有时候现实情况可能要比我们想象中要麻烦很多，如果真像上面说的那么圆满自然是最好不过，但是现实情况并不是那样。在现实情况中，落在圆环上的多个Cache的位置可能并不会像上图中那么分散，反而往往会很集中，如下图所示。



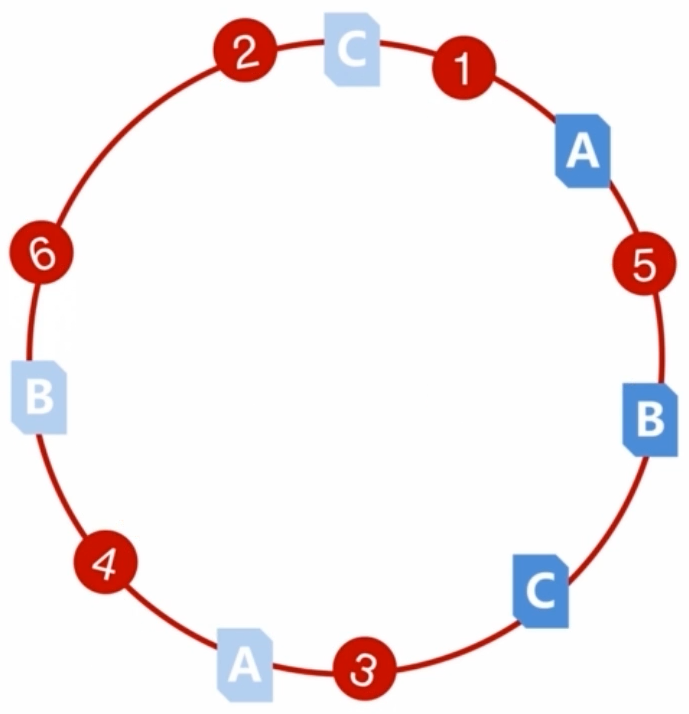
上图中的A、B、C指的都是Cache，出现上图中的情况是由于hash的倾斜性。假设这时候有6个object落在了圆环中，且这6个object经过hash函数计算出的hash值就是下图中的1~6的话，这时候出现的问题就是，1、2、3、4、6都被存在了A中，5被存在了B中，而C中却是什么都没有，这就导致了A、B、C这3个Cache负载不均衡。



为了解决上面说的Cache负载不均衡的问题，便引入了虚拟节点，虚拟节点的模型图如下图所示。



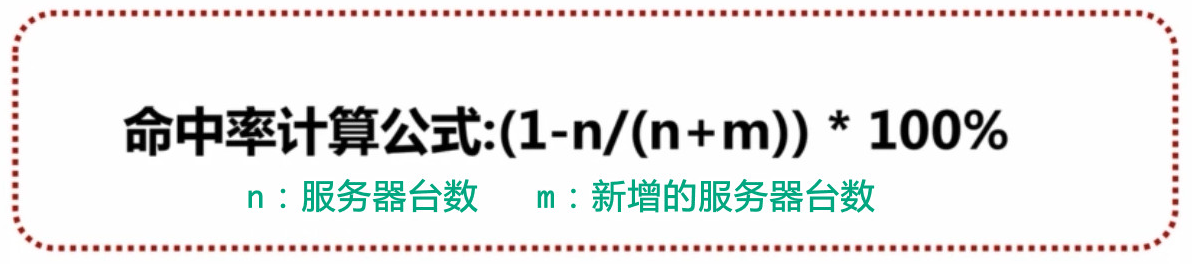
上图中的N1、N2、N3其实都是一个个的Cache，而在上图中，每一个Cache都有两个虚拟节点，假设这时候有object1和object2两个对象，它们经过hash函数计算出的hash值肯定都会落在圆环上，假设这时候这两个hash值一个储存在了V2这个虚拟节点上，另一个储存在了V5这个虚拟节点上。由于V2和V5这两个节点分别是N1和N3的虚拟节点，所以其实object1的hash值是储存在了N1这个Cache上，而object2的hash值是储存在了N3这个Cache上。下面再用图来举个例子，如下图。



上图中，有A、B、C三个Cache，同时也有它们对应的3个虚拟节点。由于储存在虚拟节点中就等于是储存在该虚拟节点对应的Cache中，所以上图中的1和3会储存在A中，4和5会储存在B中，2和6会储存在C中。引入虚拟节点后，就可以解决hash倾斜性的问题了。

或许我们会想，万一虚拟节点和它们对应的真实节点都受hash倾斜性影响怎么办？这样的话，不是依然会出现Cache负载不均衡的情况吗？其实并不会，因为只要虚拟节点足够多，那么圆环上就会有密密麻麻的虚拟节点，这时候每一个Cache都对应了很多的虚拟节点。这样的话，不管数据落在圆环的哪个地方，由于虚拟节点很多而且分布不均匀，所以不会出现有些Cache上存很多数据，有些Cache上却只存很少数据的情况。而且在增加或减少Cache数量的时候，对所有数据的影响也是很小的。

下图是命中率计算公式，从公式中可以看出，使用一致性hash算法的话，随着分布式中服务器数量的不断扩大，命中率就会越来越高，这个算法的优点就会越来越明显。



## 2. redis分布式缓冲实战

首先肯定就是启动两个redis，一个可以使用默认方式进行启动，另一个可以在修改过端口号之后通过redis.conf这个文件进行启动。然后回到代码中，首先就是在项目中创建一个RedisShardedPool类，这个类和之前创建的RedisPool类很类似，不过由于要引入分布式的概念了，所以RedisShardedPool类还是和RedisPool类有些区别的。比如连接池不能再使用JedisPool类创建了，而是换成了ShardedJedisPool类了。而且由于是分布式的，所以还新增了另一个redis的ip和端口号的信息，而且还增加了JedisShardInfo这个类等等。另外，还创建了一个RedisShardedPoolUtil工具类，这个工具类和之前创建的RedisPoolUtil类也很相像，不过也还是有区别的。既然用到了分布式，那么之前在那些Controller中用到的RedisPoolUtil肯定都要用RedisShardedPoolUtil进行替换。我这里只是大致说一下编码的思路，如果想了解具体细节可以直接看源码。

## 3. 集群和分布式的区别

关于它们的区别，老师说了一句话这里可以提一下，老师说集群是一种物理形态，分布式是一种工作方式。另外，我对集群和分布式的区别的理解是，集群其实就是单纯地增加服务器的数量，集群中的服务器之间是没有什么协作关系的，但是分布式不一样，分布式的服务器之间是有协作关系的。集群相当于是很多服务器在同时处理一个大任务，而把一个大任务拆分成很多子任务，然后交给不同的服务器来处理则属于分布式。由于分布式下的服务器是在处理一个大任务中的各个子任务，所以这些服务器处理的内容是不一样的，而且它们之间是可能存在协作关系的。

# 八、SpringMVC全局异常处理

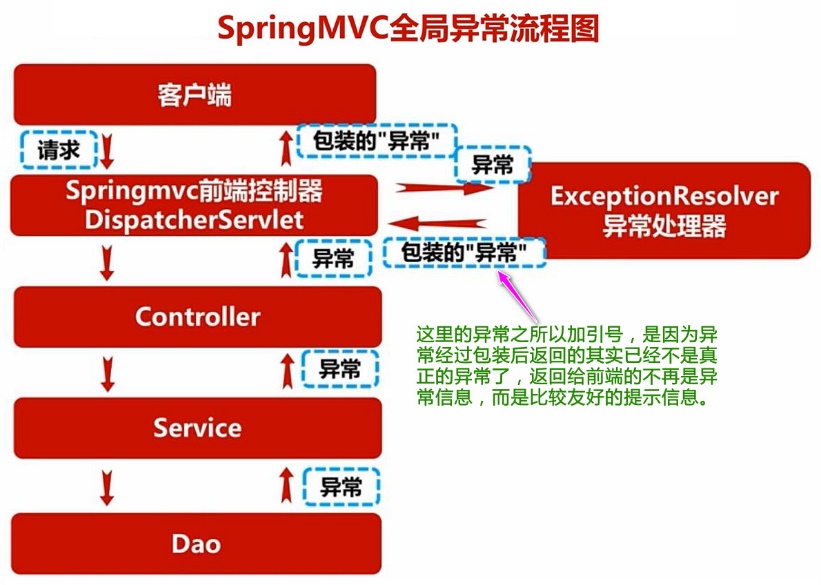
在这之前老师还讲了Spring Session框架，这个框架可以零侵入实现单点登录。虽然这个框架可以解决集群环境下Session共享的问题，但是使用这个框架之后就没法进行redis分布式了，所以这里我就不再对这个框架进行总结了，了解即可。

这里要讲的全局异常处理的逻辑其实很简单，我这里就大致说一下。首先说一下为什么要进行全局异常处理。如果不进行全局异常处理的话，在用户通过浏览器进行访问的时候一旦出现异常，浏览器中就会显示异常信息，而且一般这个信息还比较详细。这样一方面对用户来说也不友好，另一方面由于异常信息中暴露了很多代码中的信息，所以也不安全。

由于用到了注解，所以首先要修改一下Spring和SpringMVC对应的配置文件中关于扫描包的位置。由于是使用SpringMVC 处理全局异常，所以为了避免重复扫描，要在Spring对应的配置文件中修改一下扫描包的位置，让这个配置文件不扫描Controller包下的所有类。也要在SpringMVC对应的配置文件中修改扫描的位置，修改成只扫描Controller包下的类。

配置文件设置好之后就是写代码啦。首先创建一个ExceptionResolver类，然后让这个类实现HandlerExceptionResolver接口，并实现这个接口下的resolveException方法，之后直接在这个方法中写代码就可以了。有一点需要特别强调的是，如果想要让这个类能够实现全局异常处理，一定要使用@Component注解将这个类注入到Spring容器中，不然这个类是没法进行全局异常处理的。到此，关于SpringMVC全局异常处理的逻辑就讲完啦。另外，下面是有无SpringMVC全局异常处理的两个对照图，我粘贴过来了，以供参考。



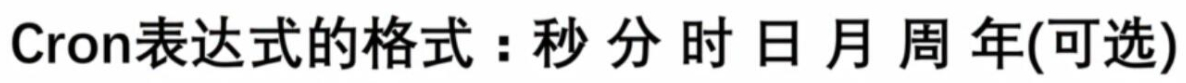


# 九、Spring Schedule实现定时关单

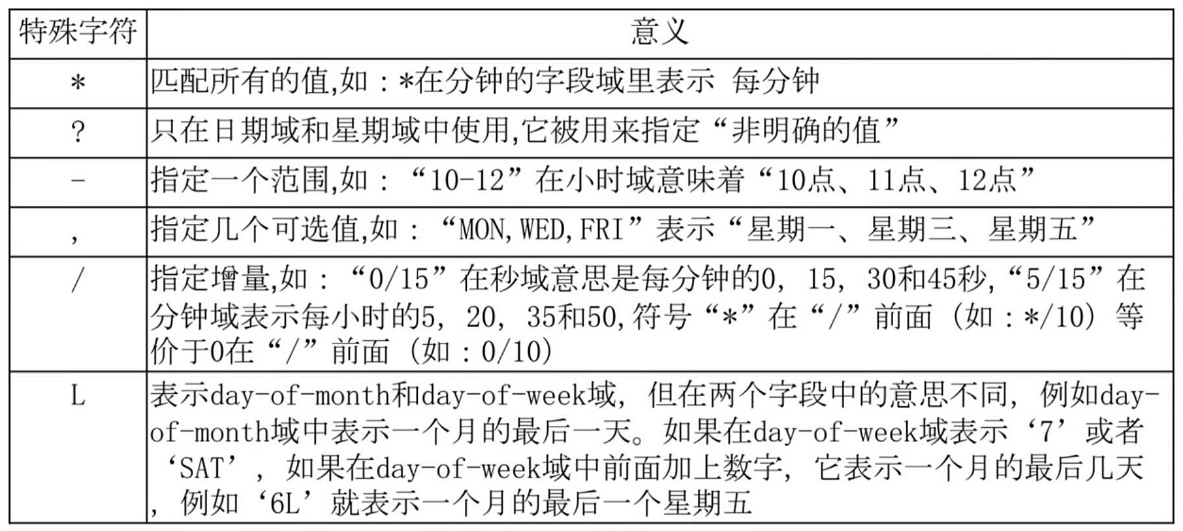
## 1. Spring Schedule入门

用一句话介绍Spring Schedule就是，它可以用于作业调度，如定时任务。在这个项目中，Spring Schedule的作用就是设置定时任务，定时关闭未付款的订单。

如果要使用Spring Schedule设置定时任务，就需要了解Spring Schedule Cron表达式，Cron表达式的格式如下。



以上Cron表达式的7个字段中，每个字段都有一些可以允许的特殊字符，这些特殊字符有很多，我这里只介绍常用的，如下图所示。



举个例子，比如"0 \*/1 \* \* \* ?"这样一个表达式就表示每一分钟，用在定时关单的业务逻辑中时，这么使用的意思就是，每分钟执行一下关闭订单的代码。如果我们设置超时时间为两个小时的话，那么就表示每过一分钟就把两小时前未付款的订单关闭。

## 2. mysql的行锁和表锁

在关闭订单的时候会涉及到mysql中行锁和表锁的概念。我的理解是，行锁是给表中某一行加上锁，加上锁之后，假设在并发访问的时候有两个用户同时访问这一行数据，那么在其中一个用户还没访问结束前，另一个用户不能对这行数据进行修改等操作。表锁跟行锁类似，只不过表锁锁住的是整个表，行锁锁住的只是一行。

在mysql中的锁又分为乐观锁和悲观锁，乐观锁一般是在表中加上一个version的字段，然后在进行一些操作时，通过对比操作前后version字段是否一致来判断是否存在并发问题；而悲观锁一般是在sql语句后加上"for update"，在这个项目中用到的就是悲观锁，使用悲观锁的前提是mysql使用的是InnoDB引擎。悲观锁相对乐观锁要简单一些，不过也是有一些注意事项的。由于条件和结果不同，悲观锁锁住的内容也不同，有可能是锁住一张表(表锁)，也有可能只是锁住一行(行锁)。

以查询为例，假设一个表有主键，如果查询的时候用主键进行查询，且能查询出对应的结果，那么使用悲观锁锁住的就是一行(行锁)；如果使用主键查询，但是没有查询出对应的结果，那么就不会上锁；如果使用主键查询，但是主键不明确(比如主键id不等于2，这就属于不明确，因为不为2的id值可能有很多)的话，那么锁住的就是整个表(表锁)；如果并没有使用主键，而是使用其他字段进行查询的话，那么锁住的也是整个表(表锁)。

## 3. 定时关单实战

首先要在Spring的配置文件(如applicationContext.xml)中引入如下标签。



然后需要在该配置文件中引入task标签对应的约束，引入约束后Spring Schedule的配置就算弄好啦，之后就是开始写代码啦。

首先在订单相关的service实现中创建一个关闭订单的方法，该方法传入的参数是超时时间，这里的超时时间会在调用该关闭订单的方法时设置一个两小时的默认值，也就是说，让未付款的订单在两个小时之后自动关闭。在关闭订单的方法中，首先用现在的时间减去超时时间，得到的就是订单的创建时间，然后在数据库中查出所有未付款且创建时间小于上面那个相减后得到的创建时间的订单。一个订单下可能有多个商品，所以如果要关闭订单的话，必须要让该订单下所有的商品再回到库存中，即让该订单下所有商品对应的库存数变成原来的库存数与该未付款订单中对应商品的数量之和。库存数修改完成之后，再把该订单的状态改成已取消状态即可。

以上只是取消一个未付款订单的代码，由于未付款的订单可能会有很多，所以我们不可能人为地一个个去进行取消，所以这时候就需要设置一个定时任务，让系统每隔一段时间自动检测是否有超时的订单，如果有就进行取消。这里就是使用Spring Schedule来设置定时任务的。

新建一个专门用于设置定时任务的类，然后使用@Component注解把这个类注入到Spring容器中，接着就是在该类中创建一个自动取消订单的方法。在这个方法上，我们要设置间隔时间，这里设置的是一分钟，即每隔一分钟检测一下是否有未付款时间超过两个小时的订单，如果有就进行取消。在方法上使用@Scheduled(cron="0 \*/1 \* \* \* ?")就代表使用Spring Schedule框架启动了定时任务，且每隔一分钟就会检测一次。这个方法中的代码逻辑就很简单了，就是调用service实现中关闭订单的方法，不过在调用这个方法之前会先设置一个超时时间的默认值，然后传入到这个方法中。

# 九、Redis的分布式锁

在上面使用Spring Schedule实现定时关单的时候，代码中是有缺陷的。虽然可以正常执行定时任务，但是如果是在tomcat集群环境下的话，就会每台tomcat服务器都执行这个定时任务，这样不仅会影响服务器的性能，还有可能造成数据错乱。我们希望的是，只要有一台服务器在执行定时任务，那么其他服务器这时候就不要再执行啦。也就是说，这个定时任务应该是随机让某一台服务器来执行，而不是所有服务器同时执行。

上面在使用Spring Schedule的时候，设置的是每分钟执行一次定时任务。假设这时候有tomcatA和tomcatB两台服务器的话，我们希望的结果是，当tomcatA执行定时任务的时候，tomcatB就不会再执行。另外，由于每分钟都执行一次定时任务，所以这个定时任务这一分钟是在tomcatA上执行的，下一分钟就有可能是在tomcatB上执行的啦。如果想要达到上面的要求，那么就需要在执行定时任务的代码中加入一个分布式锁啦。

## 1. 分布式锁的代码逻辑

加上分布式锁后的代码逻辑是这样的。首先使用redis的setnx方法设置一个锁。这个方法的参数中的key值和value值都是由我们自定义的，这里设置key值为lockname。另外，如果这个方法设置成功，方法就会返回一个1，如果设置失败，就会返回0。假设这个方法的返回值是变量setnxResult，如果变量setnxResult不为空且等于1的话，就代表设置成功，只要满足这个条件，就说明可以获取到锁啦。我们把执行定时关单的代码放到锁中，这样一来，就只有获取到锁后才能执行定时关单啦。

假设现在有一个线程获取到了锁，然后它正在执行定时关单的代码。如果这时候另一个线程也想获取到锁，那么它首先肯定是要执行setnx方法，但是由于锁还没有被释放，即lockname这个键还没有消失，所以执行setnx方法的返回值会是0，这样的话，该线程将无法获取到锁，那么自然也就没法执行定时关单的代码啦，这样也就不会出现多个线程同时执行定时关单的代码的情况啦。不过为了防止死锁现象，在一个线程获取到锁并执行完定时关单的代码后，就要立刻释放锁。释放锁的方法也很简单，只要将lockname这个键删除即可。当然，也可以给这个键设置一个几秒的有效期，那么几秒之后这个键就会自动消失，那么这个锁自然也就被释放啦，这样其他线程自然就可以获取到锁啦。

## 2. 分布式锁之双重防死锁

在正常情况下，使用上面提到的分布式锁就足以解决多个服务器同时执行定时关单的问题啦，而且也不会出现死锁的情况。但是假设现在tomcatA在执行定时任务的代码时，刚通过执行setnx方法设置好锁，tomcatA就故障了。那么出现的后果就是，下次当tomcatB来执行定时任务的代码时，由于tomcatA设置了锁，但是还没来得及释放锁就故障了，所以tomcatB调用setnx方法时返回的结果肯定是0，因为锁已经存在了，并且还没有被释放。这样一来，就永远无法进入锁中执行里面关闭订单的代码啦，这样就形成了死锁。如果想解决这种问题，就需要进行双重防死锁了。

上面在讲分布式锁的时候，提到了setnx的方法中的value值是可以自定义的，但是如果要想做到双重防死锁的话，这个value值就不能自定义了，这个value值要等于当前时间与锁的有效期之和。由于双重防死锁的业务逻辑过于复杂，我没法再用文字表述，所以就直接把代码粘贴过来啦，不过我也加了很详细的注释，具体内容如下图所示。

