# 数据库

## MySql基础（李玉婷）

所有数据库的表信息都在myemployees，girls这两个数据库中

### 进阶二：条件查询

select 查询列表 from 表名 where 筛选条件

分类：

1. **按照条件表达式筛选**

**条件表达式有： > < = <>(不等)**

Demo: 查询工资大于12000的员工信息

SELECT \* FROM employees WHERE salary >12000

Demo: 查询部门编号不等于90号的员工名和部门编号

1. **按照逻辑表达式筛选：**

**and or not**

Demo: 查询工资在10000到20000之间的员工名、工资以及奖金

SELECT last\_name,salary,commission\_pct

FROM employees

WHERE salary>=10000 AND salary<=20000

Demo: 查询部门编号不是在90到110之间，或者工资高于15000的员工信息

SELECT \* FROM employees

WHERE NOT(department\_id>=90 AND department\_id<=110) OR salary>15000

1. **模糊查询**

**like between and in is null**

**like一般和通配符搭配使用，%任意多个字符,包含0个字符，\_任意单个字符**

Demo: 查询员工名中包含字符a的员工信息

SELECT \* FROM employees

WHERE last\_name LIKE '%a%'

Demo: 查询员工名中第三个字符为e，第五个字符为a的员工名和工资

SELECT last\_name,salary FROM employees

WHERE last\_name LIKE '\_\_e\_a%'

Demo: 查询员工名中第二个字符为\_的员工名

SELECT last\_name FROM employees

WHERE last\_name LIKE '\_\\_%' 此处我们应该使用 \ 进行转义，让\_不再表示通配符

也可以这样

SELECT last\_name FROM employees

WHERE last\_name LIKE '\_$\_%' ESCAPE '$' 使用ESCAPE关键字来

**使用between and 可以提高语句的简洁度**

**包含临界值**

**两个临界值不要调换顺序**

Demo: 查询员工编号在100到120之间的员工信息

SELECT \* FROM employees

WHERE employee\_id BETWEEN 100 AND 120

**In: 含义：判断某字段的值是否属于in列表中的某一项**

**特点：**

**①使用in提高语句简洁度**

**②in列表的值类型必须一致或兼容**

**③in列表中不支持通配符**

Demo: 查询员工的工种编号是 IT\_PROG、AD\_VP、AD\_PRES中的一个员工名和工种编号

SELECT last\_name,job\_id FROM employees

WHERE job\_id IN('IT\_PROG','AD\_VP','AD\_PRES')

**!=或<>不能用于判断null值**

**is null或is not null 可以判断null值**

Demo: 查询没有奖金的员工名和奖金率

SELECT last\_name,commission\_pct FROM employees

WHERE commission\_pct IS null

**安全等于 <=>**

**IS NULL:仅仅可以判断NULL值，可读性较高，建议使用**

**<=> :既可以判断NULL值，又可以判断普通的数值，可读性较低**

查询没有奖金的员工名和奖金率

SELECT last\_name,commission\_pct FROM employees

WHERE commission\_pct <=> NULL

查询工资为12000的员工信息

SELECT \* FROM employees

WHERE salary <=> 12000

### 进阶三：排序查询

语法：

select 查询列表

from 表名

【where 筛选条件】

order by 排序的字段或表达式;

**特点：**

**1、asc代表的是升序，可以省略**

**desc代表的是降序**

**2、order by子句可以支持 单个字段、别名、表达式、函数、多个字段**

**3、order by子句在查询语句的最后面，除了limit子句**

添加筛选条件再排序

Demo：查询部门编号>=90的员工信息，并按员工编号降序

SELECT \* FROM employees

WHERE department\_id >=90

ORDER BY employee\_id desc

按表达式排序

Demo: 查询员工信息 按年薪降序

SELECT \* FROM employees

ORDER BY salary\*12\*(1+IFNULL(commission\_pct,0)) desc

这里需要注意的是年薪=月薪\*12\*(1+奖金)，奖金先要判断一下在表中是否为null

按别名排序

Demo: 查询员工信息 按年薪升序

SELECT \* ,salary\*12\*(1+IFNULL(commission\_pct,0)) AS '年薪'FROM employees

ORDER BY '年薪'

按函数排序

Demo：查询员工名，并且按名字的长度降序

SELECT last\_name FROM employees

ORDER BY LENGTH(last\_name) desc

按多个字段排序

Demo: 查询员工信息，要求先按工资降序，再按employee\_id升序

先按salary排序，再按employee\_id排序

SELECT \* FROM employees

ORDER BY salary DESC,employee\_id

### 进阶四：常见函数

#### 单行函数

##### 字符函数

1.Length()：获取参数值的字节个数

2.Concat():拼接字符串

Demo: SELECT CONCAT(last\_name,'\_',first\_name) ‘姓名’ FROM employees

3.Upper lower:转化成大写，小写

Demo:将姓变大写，名变小写，然后拼接

SELECT CONCAT(UPPER(last\_name),'\_',LOWER(first\_name)) FROM employees

Substr():截取字符串 SELECT SUBSTR('shoushoushuaibi',9) output ----🡪shuaibi

Trim():去前后空格

##### 数学函数

round() :四舍五入 ceil()向上取整，返回大于等于该参数的最小整数

floor:向下取整 truncate(1.69,1) 保留1位小数//1.6

mod取余 select mod(5,2) 🡺 select 5%2 = 1

##### 日期函数

now()：返回当前系统的日期和时间

curdate()：返回当前系统日期，不包含时间

select str\_to\_date(‘1998-3-2’,’%Y-%c-%d’) As output

date\_format 将日期转化成字符

SELECT DATE\_FORMAT(NOW(),'%y年%m月%d日')=》19年06月24日 AS output

Demo：查询有奖金的员工名和入职日期

SELECT last\_name,DATE\_FORMAT(hiredate,'%y年%m月%d日')

FROM employees

WHERE commission\_pct IS NOT null

##### 流程控制函数

SELECT last\_name,commission\_pct,IF(commission\_pct IS NULL,'没奖金','有奖金')AS 备注

FROM employees

**case函数的使用一： switch case 的效果**

**mysql中**

**case 要判断的字段或表达式**

**when 常量1 then 要显示的值1或语句1;**

**when 常量2 then 要显示的值2或语句2;**

**...**

**else 要显示的值n或语句n;**

**end**

demo：查询员工的工资，要求：

部门号=30，显示的工资为1.1倍

部门号=40，显示的工资为1.2倍

部门号=50，显示的工资为1.3倍

其他部门，显示的工资为原工资

SELECT salary 原始工资,department\_id,

CASE department\_id

WHEN 30 THEN salary\*1.1

WHEN 40 THEN salary\*1.2

WHEN 50 THEN salary\*1.3

ELSE salary

END AS 新工资

FROM employees;

**case 函数的使用二：类似于 多重if**

**mysql中：**

**case**

**when 条件1 then 要显示的值1或语句1**

**when 条件2 then 要显示的值2或语句2**

**。。。**

**else 要显示的值n或语句n**

**end**

Demo：查询员工的工资的情况

如果工资>20000,显示A级别

如果工资>15000,显示B级别

如果工资>10000，显示C级别

否则，显示D级别

SELECT salary,

CASE

WHEN salary>20000 THEN 'A'

WHEN salary>15000 THEN 'B'

WHEN salary>10000 THEN 'C'

ELSE 'D'

END AS 工资级别

FROM employees;

#### 分组函数

**Sum求和**

**avg平均值**

**max最大值**

**min最小值**

**count计算个数**

**特点：**

**1、sum、avg一般用于处理数值型**

**max、min、count可以处理任何类型**

**2、以上分组函数都忽略null值**

**3、可以和distinct搭配实现去重的运算**

SELECT MAX(salary) FROM employees 去重之后计算工资的最大值

**4、count函数的单独介绍**

**一般使用count(\*)用作统计行数**

**5、和分组函数一同查询的字段要求是group by后的字段**

### 进阶五：分组查询

**语法：**

**select 查询列表**

**from 表**

**【where 筛选条件】**

**group by 分组的字段**

**【order by 排序的字段】;**

**特点：**

**1、和分组函数一同查询的字段必须是group by后出现的字段**

**2、筛选分为两类：分组前筛选和分组后筛选**

**针对的表 位置 连接的关键字**

**分组前筛选 原始表 group by前 where**

**分组后筛选 group by后的结果集 group by后 having**

**问题1：分组函数做筛选能不能放在where后面**

**答：不能**

**问题2：where——group by——having**

**一般来讲，能用分组前筛选的，尽量使用分组前筛选，提高效率**

**3、分组可以按单个字段也可以按多个字段**

**4、可以搭配着排序使用**

分组前筛选

Demo1: 查询每个工种的最高工资

SELECT MAX(salary),job\_id FROM employees

GROUP BY job\_id

Demo2：查询每个位置的部门个数

SELECT COUNT(department\_id),department\_id

FROM departments

GROUP BY location\_id

Demo3：查询邮箱中包含a字符的 每个部门的最高工资

SELECT MAX(salary),email

FROM employees

WHERE email LIKE '%a%'

GROUP BY department\_id

注意：这里是先筛选出邮箱中包含a字符的部门，再按照每个部门进行分组

Demo：查询有奖金的每个领导手下员工的最高工资

SELECT MAX(salary),COUNT(\*)

FROM employees

WHERE commission\_pct IS NOT NULL

GROUP BY manager\_id

分组后筛选

Demo：查询哪个部门的员工个数>5

思路：分步查询

1）.先查询每个部门的员工个数

2.根据1）的结果查询哪个部门的员工个数大于5 （根据**分组之后的结果及进行筛选**用having）

SELECT COUNT(\*),department\_id

FROM employees

GROUP BY department\_id

HAVING(COUNT(\*)>5)

Demo：每个工种有奖金的员工的最高工资>12000的工种编号和最高工资

1. 查询每个工种有奖金的员工的最高工资

SELECT job\_id,MAX(salary)

FROM employees

WHERE commission\_pct IS NOT NULL

GROUP BY job\_id

1. 根据1）的结果继续筛选

SELECT job\_id,MAX(salary)

FROM employees

WHERE commission\_pct IS NOT NULL

GROUP BY job\_id

HAVING(MAX(salary>12000))

方法：如果你不能第一时间判断筛选条件放在group by前面还是后面可以先把select XXX from XXX group by先写好；再分析，我们发现有奖金的员工可以在原来的employees这张表中显示出来，因此写在group by之前用where来筛选。

Demo：领导编号>102的每个领导手下的最低工资大于5000的领导编号和最低工资

SELECT manager\_id,MIN(salary)

FROM employees

WHERE manager\_id >102

GROUP BY manager\_id

HAVING(MIN(salary)>5000)

添加排序

Demo：每个工种有奖金的员工的最高工资>6000的工种编号和最高工资,按最高工资升序

SELECT job\_id,MAX(salary)

FROM employees

WHERE commission\_pct IS NOT NULL

GROUP BY job\_id

HAVING(MAX(salary)>6000)

ORDER BY MAX(salary) ASC

按表达式或者函数分组

Demo：按员工姓名的长度分组，查询每一组的员工个数，筛选员工个数>5的有哪些

SELECT COUNT(\*),LENGTH(last\_name)

FROM employees

GROUP BY LENGTH(last\_name)

HAVING (COUNT(\*)>5)

按多个字段分组

Demo：查询每个工种每个部门的最低工资,并按最低工资降序

SELECT MIN(salary)

FROM employees

GROUP BY job\_id,department\_id

ORDER BY MIN(salary)

### 进阶六：连接查询（多表查询）

**含义：又称多表查询，当查询的字段来自于多个表时，就会用到连接查询**

**笛卡尔乘积现象：表1 有m行，表2有n行，结果=m\*n行**

**发生原因：没有有效的连接条件**

**如何避免：添加有效的连接条件**

**分类：**

**按年代分类：**

##### sql92标准:仅仅支持内连接

**内连接：**

###### 等值连接

Demo：查询女神名和对应的男神名

SELECT g.NAME,b.boyName

FROM beauty g,boys b

WHERE g.boyfriend\_id = b.id ------注意给表起别名

**为表起别名**

**高语句的简洁度**

**区分多个重名的字段**

Demo：查询员工名和对应的部门名

SELECT last\_name,department\_name

FROM employees e,departments d

WHERE e.department\_id = d.department\_id

**可以加筛选**

Demo：查询有奖金的员工名、部门名

SELECT e.last\_name,d.department\_name

FROM employees e,departments d

WHERE commission\_pct IS NOT NULL AND e.department\_id = d.department\_id

Demo：查询城市名中第二个字符为o的部门名和城市名

SELECT department\_name,city

FROM departments d ,locations l

WHERE city LIKE '\_o%' AND d.location\_id = l.location\_id

**可以加分组**

Demo：查询每个城市的部门个数

SELECT COUNT(department\_id) 每个城市的部门个数

FROM departments d,locations l

WHERE d.location\_id = l.location\_id

GROUP BY city

Demo：查询有奖金的每个部门的部门名和部门的领导编号和该部门的最低工资

SELECT d.department\_name,d.manager\_id,MIN(salary)

FROM departments d,employees e

WHERE commission\_pct IS NOT NULL

AND d.department\_id = e.department\_id

GROUP BY d.department\_id

Demo：查询每个工种的工种名和员工的个数，并且按员工个数降序

SELECT job\_title,COUNT(\*)

FROM employees e,jobs j

WHERE e.job\_id = j.job\_id

GROUP BY e.job\_id

ORDER BY COUNT(\*) desc

**可以实现三表连接**

Demo：查询员工名、部门名和所在的城市

SELECT last\_name,department\_name,city

FROM employees e,departments d,locations l

WHERE e.department\_id = d.department\_id AND d.location\_id = l.location\_id

###### 非等值连接

Demo：查询员工的工资和工资级别

SELECT salary,grade\_level

FROM employees e,job\_grades j

WHERE salary BETWEEN lowest\_sal AND highest\_sal

###### 自连接

**其实自连接和等值连接类似，只不过自连接涉及到的表就是它自己**

Demo: 查询 员工名和上级的名称

思路就是把一张表当成两张表甚至更多张表去使用，等值条件就是employee\_id= manager\_id

SELECT e.last\_name,e.manager\_id

FROM employees e,employees m

WHERE e.manager\_id = m.employee\_id

##### sql99标准【推荐】：支持内连接+外连接（左外和右外）+交叉连接

**语法：**

**select 查询列表**

**from 表1 别名 【连接类型】**

**join 表2 别名**

**on 连接条件**

**【where 筛选条件】**

**【group by 分组】**

**【having 筛选条件】**

**【order by 排序列表】**

###### 内连接：

**语法：**

**select 查询列表**

**from 表1 别名**

**inner join 表2 别名**

**on 连接条件;**

**特点：**

**①添加排序、分组、筛选**

**②inner可以省略**

**③ 筛选条件放在where后面，连接条件放在on后面，提高分离性，便于阅读**

**④inner join连接和sql92语法中的等值连接效果是一样的，都是查询多表的交集**

**等值**

Demo：查询员工名、部门名

SELECT last\_name,department\_name

FROM employees e JOIN departments d

ON e.department\_id = d.department\_id

Demo：查询名字中包含e的员工名和工种名

SELECT last\_name,job\_title

FROM employees e

INNER JOIN jobs j

ON e.job\_id = j.job\_id

WHERE e.last\_name LIKE '%e%'

Demo：查询部门个数>3的城市名和部门个数

SELECT l.city,COUNT(\*)

FROM departments d

INNER JOIN locations l

ON d.location\_id = l.location\_id

GROUP BY city

HAVING(COUNT(\*)>3)

Demo：查询哪个部门的员工个数>3的部门名和员工个数，并按个数降序

SELECT d.department\_name,COUNT(\*)

FROM employees e

INNER JOIN departments d

ON e.department\_id = d.department\_id

GROUP BY d.department\_name

HAVING(COUNT(\*)>3)

ORDER BY COUNT(\*) Desc

Demo：查询员工名、部门名、工种名，并按部门名降序

SELECT e.last\_name,d.department\_name,j.job\_title

FROM employees e

INNER JOIN departments d

ON e.department\_id = d.department\_id

INNER JOIN jobs j

ON e.job\_id = j.job\_id

ORDER BY d.department\_name

**非等值**

Demo：查询员工的工资级别

SELECT employee\_id,grade\_level

FROM employees e

INNER JOIN job\_grades j

WHERE e.salary BETWEEN j.lowest\_sal AND highest\_sal

Demo：查询工资级别的个数>20的个数，并且按工资级别降序

SELECT j.grade\_level,COUNT(\*)

FROM employees e

INNER JOIN job\_grades j

ON e.salary BETWEEN j.lowest\_sal AND j.highest\_sal

GROUP BY j.grade\_level

HAVING(COUNT(\*)>20)

ORDER BY j.grade\_level desc

**自连接**

查询员工的名字、上级的名字

SELECT e.last\_name 员工的名字,m.last\_name 上级的名字

FROM employees e

INNER JOIN employees m

ON e.manager\_id = m.employee\_id

查询姓名中包含字符k的员工的名字、上级的名字

SELECT e.last\_name 员工的名字,m.last\_name 上级的名字

FROM employees e

INNER JOIN employees m

ON e.manager\_id = m.employee\_id

WHERE e.last\_name LIKE '%k%'

###### 外连接：

**应用场景：用于查询一个表中有，另一个表没有的记录**

**特点：**

**1、外连接的查询结果为主表中的所有记录**

**如果从表中有和它匹配的，则显示匹配的值**

**如果从表中没有和它匹配的，则显示null**

**外连接查询结果=内连接结果+主表中有而从表没有的记录**

**2、左外连接，left join左边的是主表**

Demo：查询男朋友 不在男神表的的女神名

SELECT g.name,b.boyName

FROM beauty g

LEFT OUTER JOIN boys b

ON g.boyfriend\_id = b.id

WHERE b.boyName IS NULL

查询哪个部门没有员工

SELECT department\_name,employee\_id

FROM departments d

LEFT OUTER JOIN employees e

ON d.department\_id = e.department\_id

WHERE employee\_id IS null

**右外连接，right join右边的是主表**

**3、左外和右外交换两个表的顺序，可以实现同样的效果**

**4、全外连接=内连接的结果+表1中有但表2没有的+表2中有但表1没有的**

###### 交叉连接

交叉连接不带WHERE子句，它返回被连接的两个表所有数据行的笛卡尔积，返回结果集合中的数据行数等于第一个表中符合查询条件的数据行数乘以第二个表中符合查询条件的数据行数。

### 进阶七：子查询

#### #一、where或having后面

**1、标量子查询（单行子查询）**

**2、列子查询（多行子查询）**

**3、行子查询（多列多行）**

**特点：**

**①子查询放在小括号内**

**②子查询一般放在条件的右侧**

**③标量子查询，一般搭配着单行操作符使用**

**> < >= <= = <>**

**列子查询，一般搭配着多行操作符使用**

**in、any/some、all**

**④子查询的执行优先于主查询执行，主查询的条件用到了子查询的结果**

标量子查询（单行子查询）

Demo：谁的工资比 Abel 高?

1. 先查Abel的工资

SELECT salary

FROM employees

WHERE last\_name = "Abel"

单行单列

1. 查询员工的信息，满足salary>1

SELECT \*

FROM employees

WHERE salary > (SELECT salary

FROM employees

WHERE last\_name = "Abel")

Demo：返回job\_id与141号员工相同，salary比143号员工多的员工 姓名，job\_id 和工资

SELECT last\_name,job\_id,salary

FROM employees

WHERE job\_id = (

SELECT job\_id

FROM employees

WHERE employee\_id = '141'

)AND

salary>(

SELECT salary

FROM employees

WHERE employee\_id = '143'

)

Demo：返回公司工资最少的员工的last\_name,job\_id和salary

SELECT last\_name,job\_id,salary

FROM employees

WHERE salary = (

SELECT MIN(salary)

FROM employees

)

Demo：查询最低工资大于50号部门最低工资的部门id和其最低工资

SELECT department\_id,MIN(salary)

FROM employees

GROUP BY department\_id

HAVING MIN(salary)>(

SELECT MIN(salary)

FROM employees

WHERE department\_id = 50

)

当标量子查询的结果不是一行一列的情况，就是非法使用标量子查询

列子查询（多行子查询）

Demo：返回location\_id是1400或1700的部门中的所有员工姓名

1. 查询location\_id是1400或者1700的部门id
2. 查询员工姓名，要求员工所在的部门编号是1中的1个

SELECT last\_name

FROM employees

WHERE department\_id IN(

SELECT DISTINCT department\_id

FROM departments

WHERE location\_id IN(1400,1700)

);

行子查询（结果集一行多列或多行多列）

Demo：查询员工编号最小并且工资最高的员工信息

SELECT \*

FROM employees

WHERE employee\_id=(

SELECT MIN(employee\_id)

FROM employees

)AND salary=(

SELECT MAX(salary)

FROM employees

);

也可以写成

SELECT \*

FROM employees

WHERE (employee\_id,salary)=(

SELECT MIN(employee\_id),MAX(salary)

FROM employees

);

#### #二、select后面

**仅仅支持标量子查询**

Demo：查询每个部门的员工个数

SELECT d.\*,(SELECT COUNT(\*)

FROM employees e

WHERE e.department\_id = d.`department\_id`)个数

FROM departments d

###### \*\*\*\*\*面试题：

1. **有一个表match\_record，数据如下：**

|  |  |
| --- | --- |
| match\_date | Result |
| 2005-05-09 | 胜 |
| 2005-05-09 | 胜 |
| 2005-05-09 | 负 |
| 2005-05-09 | 负 |
| 2005-05-09 | 负 |
| 2005-05-10 | 胜 |
| 2005-05-10 | 负 |
| 2005-05-10 | 负 |

如果要生成下列结果, 使用oracle 该如何写sql语句（只写一种即可）?

|  |
| --- |
| 比赛日期 |胜 |负  ----------------------  2005-05-09 |2 |3  2005-05-10 |1 |2 |

分析：这道题是子查询，条件查询，分组查询加使用别名的综合型sql查询题

select match\_date as 比赛日期,(select count(\*) from m where Result = ‘胜’) as 胜,(select count(\*) from m where Result = ‘负’) as 负

from match\_record m

group by match\_date

#### #三、from后面

/\*

将子查询结果充当一张表，要求必须起别名

\*/

#案例：查询每个部门的平均工资的工资等级（非等值查询加子查询）

#①查询每个部门的平均工资

SELECT AVG(salary),department\_id

FROM employees

GROUP BY department\_id

SELECT \* FROM job\_grades;

#②连接①的结果集和job\_grades表，筛选条件平均工资 between lowest\_sal and highest\_sal

SELECT ag\_dep.\*,g.`grade\_level`

FROM (

SELECT AVG(salary) ag,department\_id

FROM employees

GROUP BY department\_id

) ag\_dep

INNER JOIN job\_grades g

ON ag\_dep.ag BETWEEN lowest\_sal AND highest\_sal;

#### #四、exists后面（相关子查询）

/\*

语法：

exists(完整的查询语句)

结果：

1或0

\*/

SELECT EXISTS(SELECT employee\_id FROM employees WHERE salary=300000);

#案例1：查询有员工的部门名

#in

SELECT department\_name

FROM departments d

WHERE d.`department\_id` IN(

SELECT department\_id

FROM employees

)

#exists

SELECT department\_name

FROM departments d

WHERE EXISTS(

SELECT \*

FROM employees e

WHERE d.`department\_id`=e.`department\_id`

);

#案例2：查询没有女朋友的男神信息

#in

SELECT bo.\*

FROM boys bo

WHERE bo.id NOT IN(

SELECT boyfriend\_id

FROM beauty

)

#exists

SELECT bo.\*

FROM boys bo

WHERE NOT EXISTS(

SELECT boyfriend\_id

FROM beauty b

WHERE bo.`id`=b.`boyfriend\_id`

);

### 进阶八：分页查询

**#进阶8：分页查询 ★**

**/\***

**应用场景：当要显示的数据，一页显示不全，需要分页提交sql请求**

**语法：**

**select 查询列表**

**from 表**

**【join type join 表2**

**on 连接条件**

**where 筛选条件**

**group by 分组字段**

**having 分组后的筛选**

**order by 排序的字段】**

**limit 【offset,】size;**

**offset要显示条目的起始索引（起始索引从0开始）**

**size 要显示的条目个数**

**特点：**

**①limit语句放在查询语句的最后**

**②公式**

**要显示的页数 page，每页的条目数size**

**select 查询列表**

**from 表**

**limit (page-1)\*size,size;**

**size=10**

**page**

**1 0**

**2 10**

**3 20**

Demo：查询前五条员工信息

SELECT \*

FROM employees

LIMIT 0,5

Demo：有奖金的员工信息，并且工资较高的前10名显示出来

SELECT \*

FROM employees

WHERE commission\_pct IS NOT NULL

ORDER BY salary DESC

LIMIT 0,10

面试题：

返回查询结果的第一行：

这里可以使用分页查询

### 进阶九：联合查询

**/\***

**union 联合 合并：将多条查询语句的结果合并成一个结果**

**语法：**

**查询语句1**

**union**

**查询语句2**

**union**

**...**

**应用场景：**

**要查询的结果来自于多个表，且多个表没有直接的连接关系，但查询的信息一致时**

**特点：★**

**1、要求多条查询语句的查询列数是一致的！**

**2、要求多条查询语句的查询的每一列的类型和顺序最好一致**

**3、union关键字默认去重，如果使用union all 可以包含重复项**

**\*/**

Demo：查询部门编号>90或邮箱包含a的员工信息

SELECT \* FROM employees WHERE department\_id>90

UNION

SELECT \* FROM employees WHERE email LIKE '%a%'

### 进阶十：数组操作语言

**#一、插入语句**

**#方式一：经典的插入**

**/\***

**语法：**

**insert into 表名(列名,...) values(值1,...);**

**\*/**

**#二、修改语句**

**/\***

**1.修改单表的记录★**

**语法：**

**update 表名**

**set 列=新值,列=新值,...**

**where 筛选条件;**

**2.修改多表的记录【补充】**

**语法：**

**sql92语法：**

**update 表1 别名,表2 别名**

**set 列=值,...**

**where 连接条件**

**and 筛选条件;**

**----------------------------------------------**

**sql99语法：**

**update 表1 别名**

**inner|left|right join 表2 别名**

**on 连接条件**

**set 列=值,...**

**where 筛选条件;**

**\*/**

**#三、删除语句**

**/\***

**方式一：delete**

**语法：**

**1、单表的删除【★】**

**delete from 表名 where 筛选条件**

**2、多表的删除【补充】**

**sql92语法：**

**delete 表1的别名,表2的别名**

**from 表1 别名,表2 别名**

**where 连接条件**

**and 筛选条件;**

**sql99语法：**

**delete 表1的别名,表2的别名**

**from 表1 别名**

**inner|left|right join 表2 别名 on 连接条件**

**where 筛选条件;**

**方式二：truncate**

**语法：truncate table 表名;**

**#delete pk truncate【面试题★】**

**1.delete 可以加where 条件，truncate不能加**

**2.truncate删除，效率高一丢丢**

**3.假如要删除的表中有自增长列，**

**如果用delete删除后，再插入数据，自增长列的值从断点开始，**

**而truncate删除后，再插入数据，自增长列的值从1开始。**

**4.truncate删除没有返回值，delete删除有返回值**

**5.truncate删除不能回滚，delete删除可以回滚.**

### Mysql的事务

一个或一组sql语句组成一个执行单元，这个执行单元要么全部执行，要么全部不执行

#### 事务的特性：

ACID

原子性：一个事务不可再分割，要么都执行要么都不执行

一致性：一个事务执行会使数据从一个一致状态切换到另外一个一致状态

隔离性：一个事务的执行不受其他事务的干扰

持久性：一个事务一旦提交，则会永久的改变数据库的数据.

### 并发事务带来的一些问题：

#### 更新丢失

当两个或多个事务选择同一行，然后基于最初选定的值更新该行时，由于每个事务都不知道其他事务的存在，就会发生丢失更新问题－－最后的更新覆盖了由其他事务所做的更新。

例如，两个程序员修改同一java文件。每程序员独立地更改其副本，然后保存更改后的副本，这样就覆盖了原始文档。最后保存其更改副本的编辑人员覆盖前一个程序员所做的更改。

如果在一个程序员完成并提交事务之前，另一个程序员不能访问同一文件，则可避免此问题。

九江银行面试题：如果有一个事务要提交，他后面的事务需要回滚，最终会怎么样？后面的事务会覆盖前面的事务（更新丢失问题）

#### 脏读

一个事务正在对一条记录做修改，在这个事务完成并提交前，这条记录的数据就处于不一致状态；这时，另一个事务也来读取同一条记录，如果不加控制，第二个事务读取了这些“脏”数据，并据此做进一步的处理，就会产生未提交的数据依赖关系。这种现象被形象地叫做”脏读”。

一句话：事务A读取到了事务B已修改但尚未提交的的数据，还在这个数据基础上做了操作。此时，如果B事务回滚，A读取的数据无效，不符合一致性要求。

#### 不可重复读

在一个事务内，多次读同一个数据。在这个事务还没有结束时，另一个事务也访问该同一数据。那么，在第一个事务的两次读数据之间。由于第二个事务的修改，那么第一个事务读到的数据可能不一样，这样就发生了在一个事务内两次读到的数据是不一样的，因此称为不可重复读，即原始读取不可重复。

一句话：一个事务范围内两个相同的查询却返回了不同数据。

#### 幻读

一个事务按相同的查询条件重新读取以前检索过的数据，却发现其他事务插入了满足其查询条件的新数据，这种现象就称为“幻读”。

一句话：事务A 读取到了事务B提交的新增数据，不符合隔离性。

### 事务的隔离级别：

脏读 不可重复读 幻读

read uncommitted：√ √ √

read committed： × √ √

repeatable read： × × √

serializable × ×

应该满足：

1. 在同一个事务中，多次查询的结果应该是一样的（产生的问题：不可重复读）
2. 在同一个事务中，如果你跟新了字段内容但是没有提交事务，查询得到的字段内容应该是未跟新过的；只有提交了事务，查询到的字段内容才是跟新过的。（产生的问题：读已提交，）

**mysql中默认 第三个隔离级别 repeatable read可重复度，确保事务可以多次从一个字段中读取到相同的值。在这个事务期间，禁止其他事务对这个字段进行修改（更新），可避免脏读和不可重复读，但是幻读问题依然存在。**

### sql的触发器，函数，存储过程

触发器

存储过程

见思维脑图

## Oracle数据库、sql和PLSQL（宋红康）

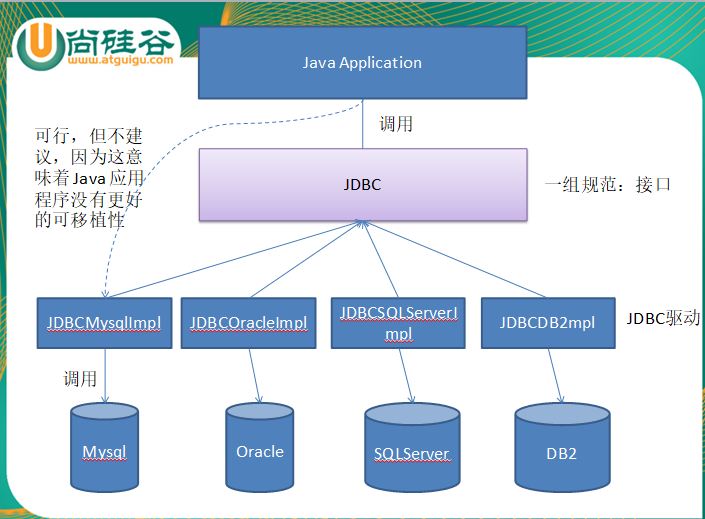
## JDBC（尚硅谷佟刚）

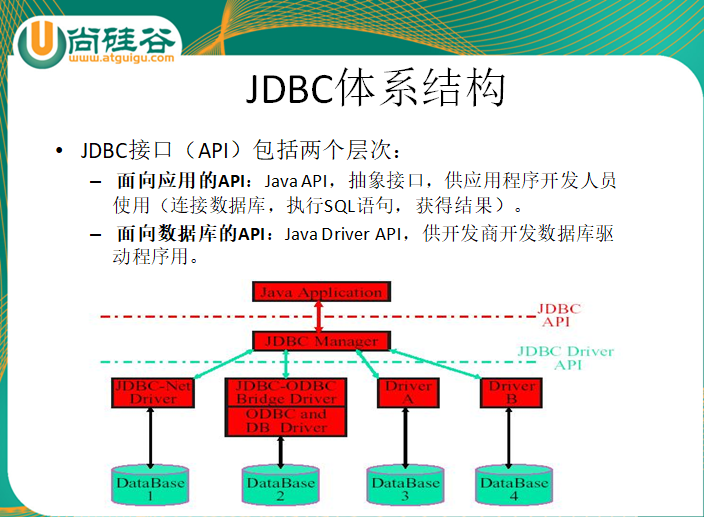
在没有JDBC之前，Java程序员若想连接不同的数据库，都要花费大量的成本（人力物力）去学习如何连接不同的数据库；并且，如果我们的项目需求发生了变化，需要更换数据库时，则需要改变大量的源代码，代码的重用性或者说可扩展性太差，这是十分不方便的，怎么办呢？在数据库和java应用程序之间加了一个JDBC,JDBC给开发人员提供了一组规范，我们只需面向JDBC编程就行，无需关心底层是如何实现连接数据库的，而把连接数据库的一些细节和原理交给各个数据库的厂商，由他们来完成数据库驱动程序的实现

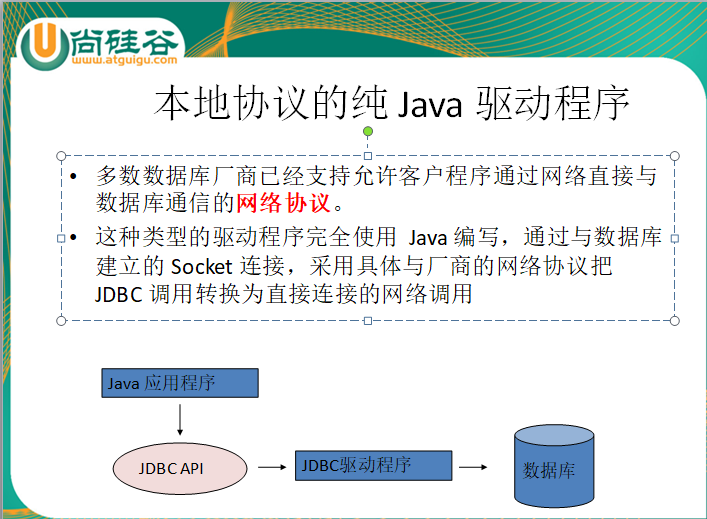
我们的jdk中的java.sql包中仅仅提供了Connection接口，但是没有提供接口的实现类；那由谁提供Connection的实现类呢？由各个数据库的厂商提供，但是我们只需要面向这种接口编程就可以了，具体厂商提供其对应的实现类。这里回顾一下我们面向对象阶段学习的多态，父类引用指向子类对象，我们在编译时只能调用接口（或者抽象类）中的方法，而运行时则会去调用子类的重写方法；同样的道理，java.sql包中提供的是接口，数据库厂商提供的数据库驱动程序中则提供了如何连接数据库的具体实现细节。

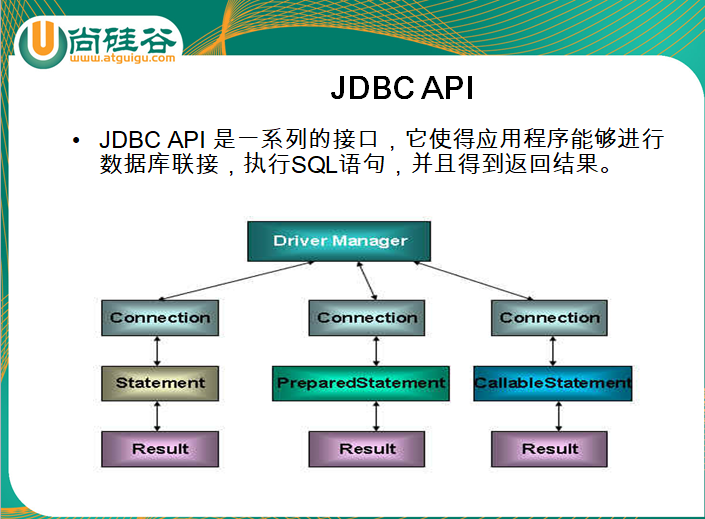
各个数据库厂商根据JDBC的规范制作的JDBC实现类的类库，称作JDBC驱动；











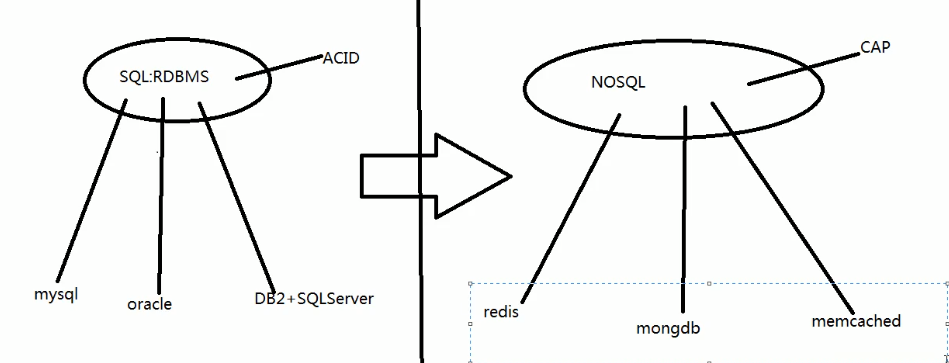


# 分布式

## Netty

## Redis

天上飞的理念必然有落地的实现



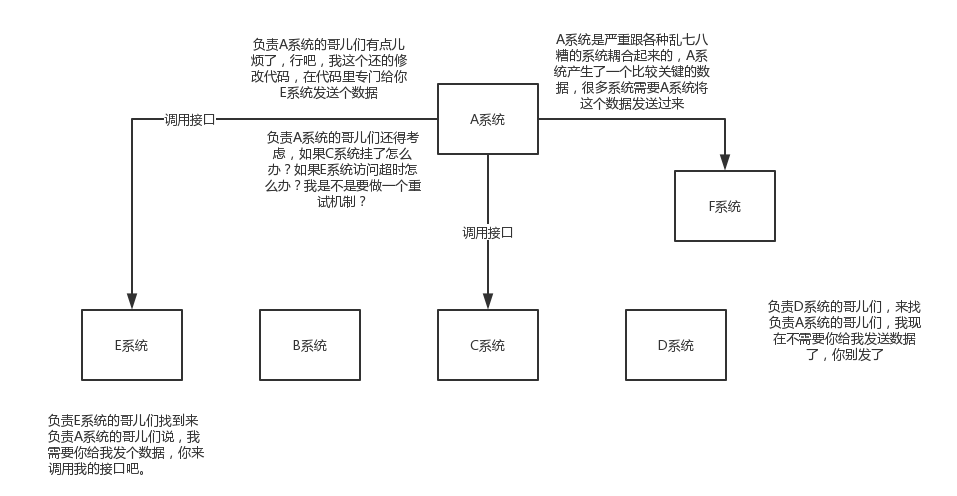
# 消息中间件

## ActiveMq（尚硅谷周阳）

### 为什么要引入MQ？

#### 消息队列的常用使用场景之一：解耦

**不使用MQ的系统耦合场景：**

****

比如说我在后台商品管理系统中（item\_service），需要把商品信息添加到数据库中；然而，其他系统也需要我们新添加进数据库的商品数据；我们在添加完之后需要另外写不同的接口给索引和缓存系统（search-service）,把添加进数据库中的商品信息同样加入到索引库中和缓存中。然后此时，又冒出来一个X系统，也需要我把商品数据传给它，我还需要再写一个接口（或者对已有的代码进行修改），专门为了给X系统发送商品数据；

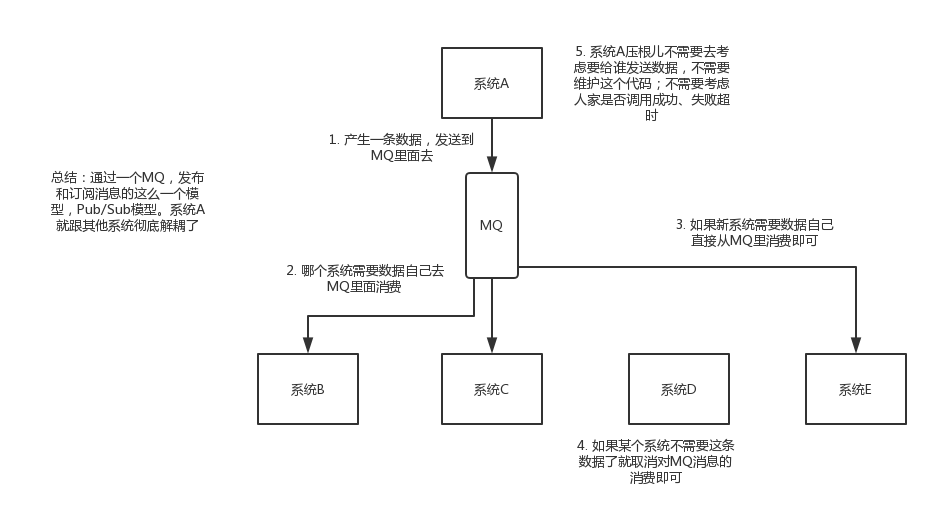
**使用MQ之后的解耦场景：**

系统A产生了一条数据，发送到了MQ中，发送完了就不管了，谁需要发送的这条数据，就自己去MQ中自己消费就行了；如果新增的这个系统也需要这条数据，可以自己添加接收消息的代码到MQ中自己去消费即可。

系统A完全不用考虑给谁发送数据，也完全不需要考虑是否接口调用成功，以及万一有一天别的系统不需要调用该接口时，我们还要删除该接口（或者修改代码）

每增加一个下游的功能，都要对上游的相关接口进行改造

总结：通过MQ的这么一个Pub/Sub（发布和订阅）模型，实现了A系统与其他系统之间的解耦；当新的模块添加进来的时候，可以做到代码改动最小。

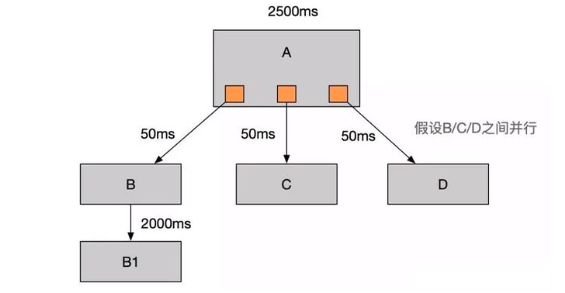
****

#### 消息队列的常用使用场景之二：异步

不用MQ的同步高延时请求场景：

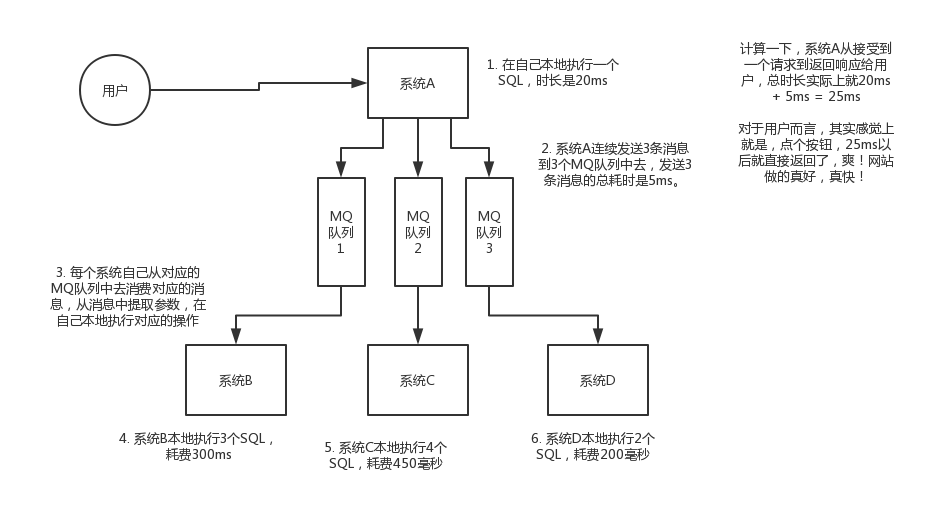
系统A的功能可能需要多处调用其他系统的接口，多接口成串行请求访问，增加了请求时间。

RPC接口基本上都是同步调用，整体的服务性能遵守木桶原理，即整体系统的耗时取决于链路中最慢的那个接口。



强弱依赖梳理能将非关键调用链路的操作异步化从而提升系统整体的吞吐能力。

若使用MQ，可以使用异步调用，可以节省高延时接口的调用性能，我只需花费往消息队列中发送消息的时间成本。

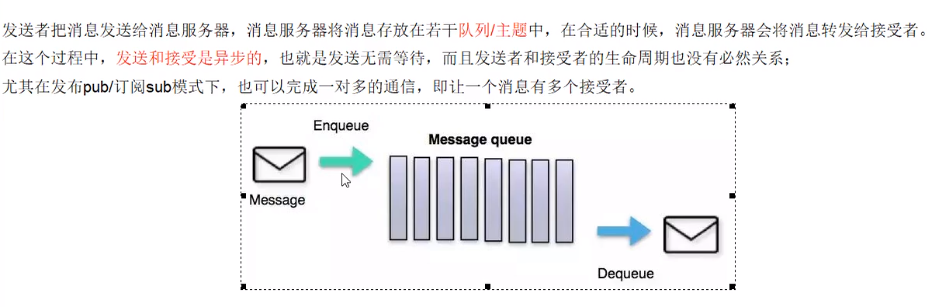
****

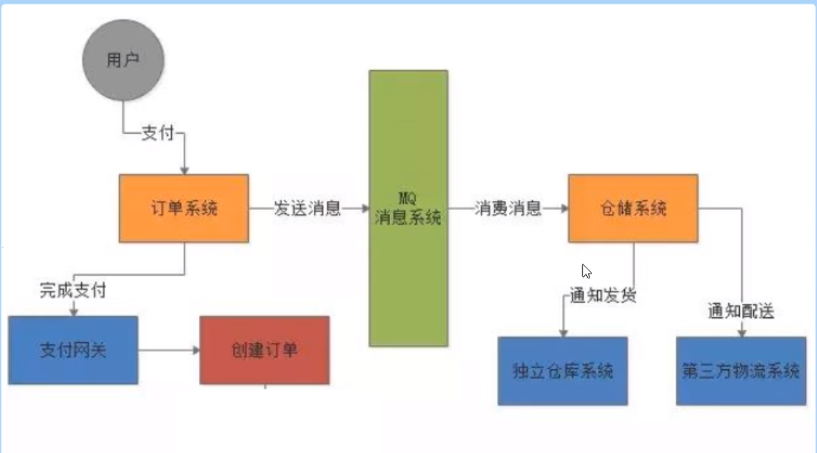
#### 消息队列的常用使用场景之三：削峰

大量的请求在短时间内涌入系统中高峰期可能达到每秒钟5000多个请求，而系统如果是直接去访问mysql的，可能会导致数据库抗不住高并发的请求压力而宕机；

因此我们需要在接入系统之前**设置流量缓冲池**，可以让后端系统按照自身吞吐能力进行消费，不被冲跨

### MQ的定义





### 3.在linux下安装ActiveMq

1. 从官网下载ActiveMq

<http://activemq.apache.org/components/classic/download/>

1. 存放在/opt目录下
2. 解压缩

tar -xzvf apache-activemq-5.15.9-bin.tar.gz

1. 在根目录下创建myactiveMQ，拷贝压缩包到myactiveMQ目录下
2. 启动ActiveMq

./activemq start

1. ActiveMq默认进程的端口是61616

ps -ef | grep activemq | grep -v grep

1. netstat –anp | grep 61616

[root@localhost bin]# netstat -anp | grep 61616

tcp6 0 0 :::61616 :::\* LISTEN 13461/java

1. 
2. 关闭ActiveMq

./activeMq stop

### 4.ActiveMq控制台

Linux Centos7关闭防火墙及关闭开机启动防火墙

1、直接关闭防火墙

systemctl stop firewalld.service

2、禁止firewall开机启动

systemctl disable firewalld.service

1. http://<IP>：8161/admin 默认用户名：admin 密码：admin

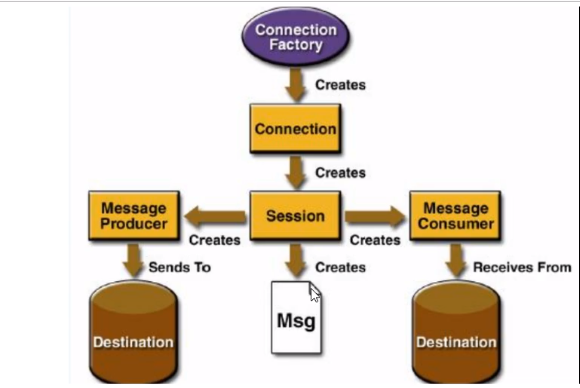
### 5.java编码实现ActiveMq通讯

java编码MQ标准API

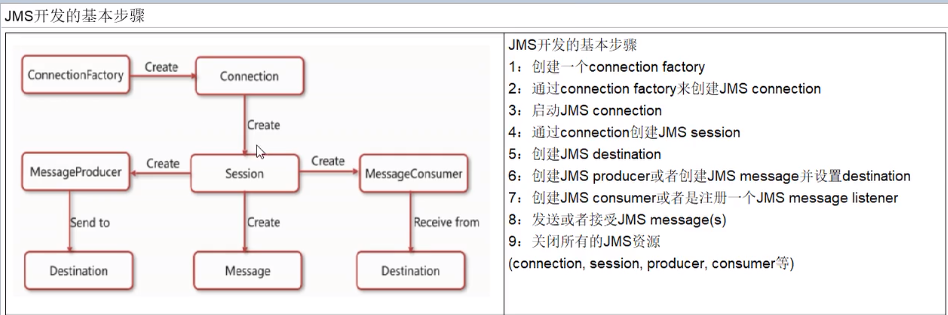
1.创建pom工程

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.atguigu.activemq</groupId>  
 <artifactId>activemq\_demo</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <maven.compiler.target>1.8</maven.compiler.target>  
 </properties>  
  
 <dependencies>  
 <!--activeMq所需要的jar包配置-->  
 <dependency>  
 <groupId>org.apache.activemq</groupId>  
 <artifactId>activemq-all</artifactId>  
 <version>5.15.9</version>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.xbean</groupId>  
 <artifactId>xbean-spring</artifactId>  
 <version>3.16</version>  
 </dependency>  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
 <version>1.7.25</version>  
 </dependency>  
 <dependency>  
 <groupId>ch.qos.logback</groupId>  
 <artifactId>logback-classic</artifactId>  
 <version>1.2.3</version>  
 </dependency>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.16.18</version>  
 <scope>provided</scope>  
 </dependency>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 </dependency>  
 </dependencies>  
</project>

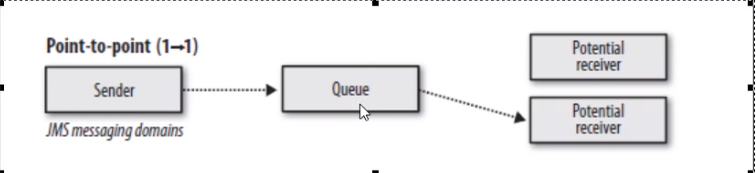
2.JMS:java消息服务的一种编程规范---JavaEE体系中的一种，下图是JMS编码总体架构



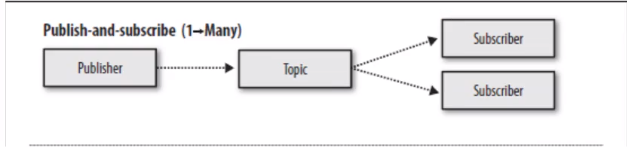
我们编写java消息队列的代码必须遵循JMS开发的基本步骤：



1. 目的地：分为队列(Queue)和主题(Topic)，也代表着ActiveMq发送消息的两大模式特性
2. point-to-point

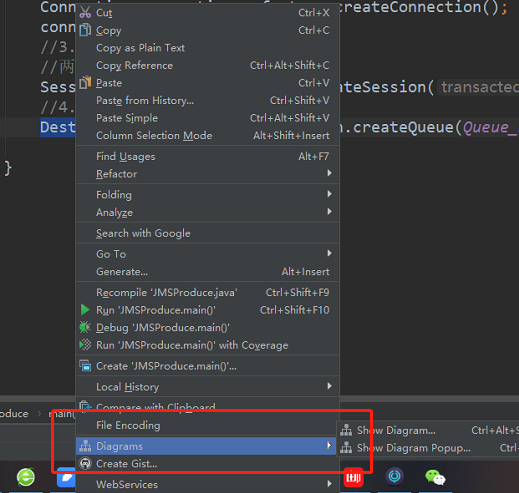


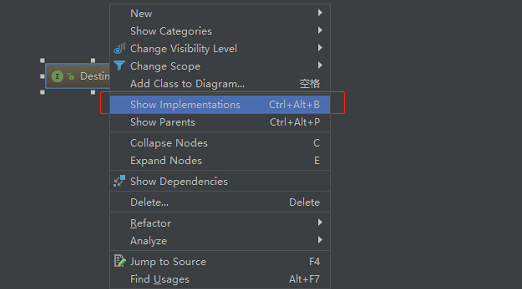
1. publish-and-subscribe

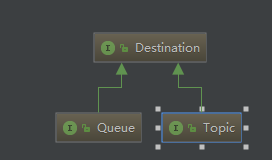


1. idea中查看类和接口的体系结构：

右键Diagrams---->show Diagram show implementations



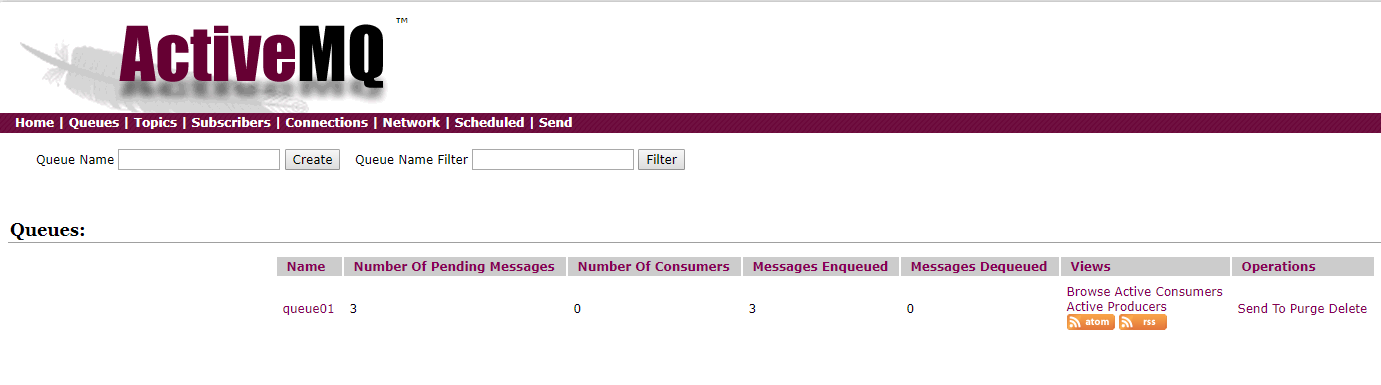


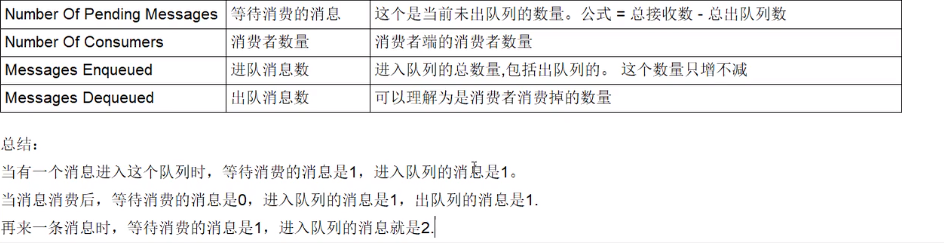


#### 消息生产者编码

public class JMSProduce {  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *Queue\_NAME* = "queue01";  
  
 public static void main(String[] args) throws JMSException {  
 //1.创建连接工厂，按照给定的url地址，采用默认的用户名密码  
 ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory("shisiqi","66261815",*DEFAULT\_BROKER\_BIND\_URL*);  
 //2.通过连接工厂，获得连接Connection并启动访问  
 Connection connection = factory.createConnection();  
 connection.start();  
 //3.创建会话Session  
 //两个参数，一个叫事务，一个叫签收  
 Session session = connection.createSession(false, Session.*AUTO\_ACKNOWLEDGE*);  
 //4.创建目的地  
 Queue queue = session.createQueue(*Queue\_NAME*);  
 //5.创建消息的生产者  
 MessageProducer producer = session.createProducer(queue);  
 //6.使用消息生产者MessageProducer发送消息到MQ队列里面  
 for (int i = 1; i <=3 ; i++) {  
 //7.创建消息  
 TextMessage textMessage = session.createTextMessage("msg---->" + i);  
 //8.通过MessageProducer对象来发送消息  
 producer.send(textMessage);  
 }  
 //9.关闭资源  
 producer.close();  
 session.close();  
 connection.close();  
 System.*out*.println("消息发布到MQ完成");  
 }  
}

控制台显示：





#### 消息消费者编码

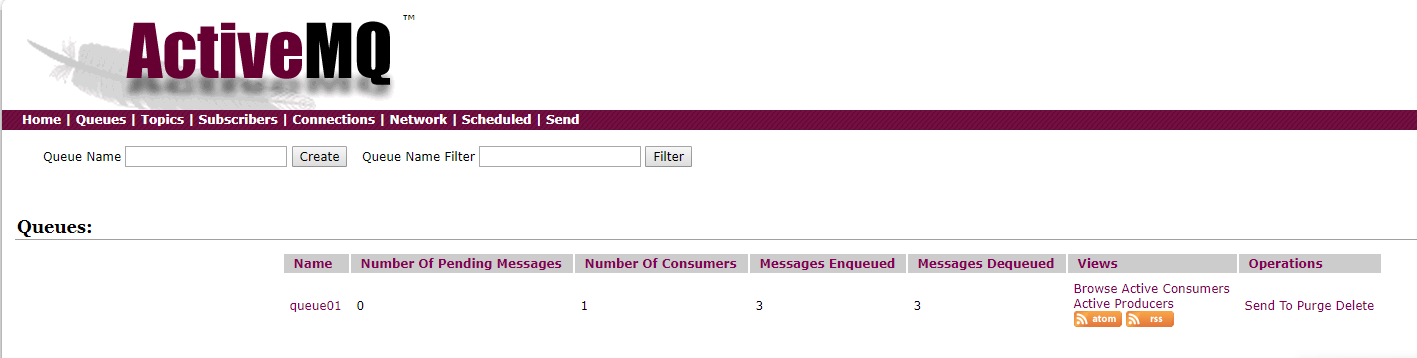
##### 一、同步阻塞方式receive()

public class JMSConsumer {  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *Queue\_NAME* = "queue01";  
 public static void main(String[] args) throws JMSException {

System.*out*.println("我是1号消费者");

// System.*out*.println("我是2号消费者");  
 ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(*DEFAULT\_BROKER\_BIND\_URL*);  
 Connection connection = factory.createConnection();  
 connection.start();  
 Session session = connection.createSession(false, Session.*AUTO\_ACKNOWLEDGE*);  
 //创建目的地  
 Queue queue = session.createQueue(*Queue\_NAME*);  
 //创建消费者对象  
 MessageConsumer consumer = session.createConsumer(queue);  
 while (true){  
 TextMessage textMessage = (TextMessage) consumer.receive();  
 if (textMessage != null){  
 System.*out*.println("消费者接收消息" + textMessage.getText());  
 }else {  
 break;  
 }  
 }  
 consumer.close();  
 session.close();  
 connection.close();  
 }  
}

控制台显示：

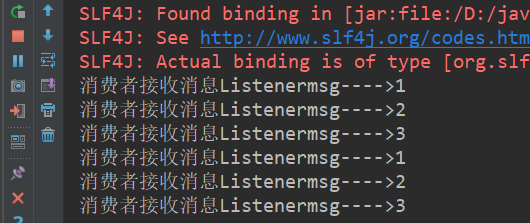


消费者也分两种：

1. 第一种是一直处于接收消息的状态，也就是Number of Consumer=1 (一直等待)
2. 第二种就是设置了定时时间的消费者，若在指定时间内没有接收到消息，它就自动关闭连接了，而此时Number of Consumer =0（过时不候）

##### 二、异步非阻塞（通过监听的方式来消费消息）

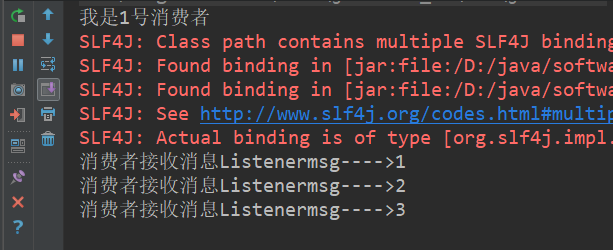
//通过监听的方式来消费消息  
consumer.setMessageListener(new MessageListener() {  
 @Override  
 public void onMessage(Message message) {  
 if (null != message && message instanceof TextMessage){  
 TextMessage textMessage = (TextMessage) message;  
 try {  
 System.*out*.println("消费者接收消息" + textMessage.getText());  
 } catch (JMSException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
});  
System.*in*.read();  
consumer.close();  
session.close();  
connection.close();

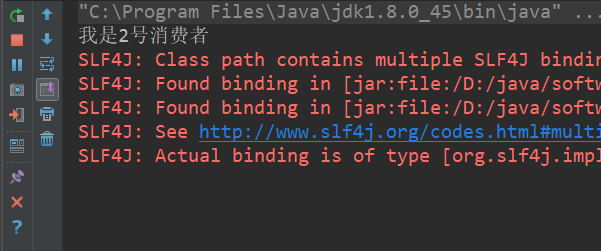


##### 问题：

1. 先生产一条消息，发送到消息队列中，只启动1号消费者，再启动2号消费者 Y
2. 先生产一条消息，发送到消息队列中，先启动1号消费者，再启动2号消费者

2号消费者还能消费吗？

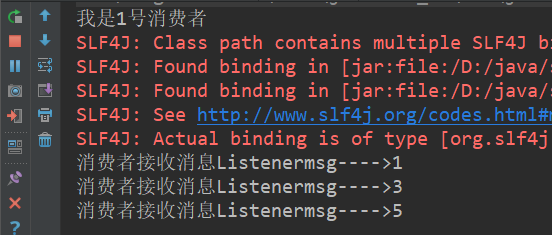


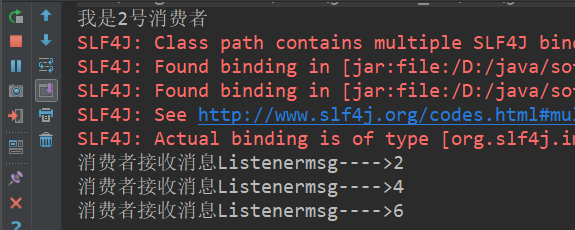




很明显，1号消费者连接了之后直接把队列中的消息给“吃掉了”，这样2号消费者就消费不了消息

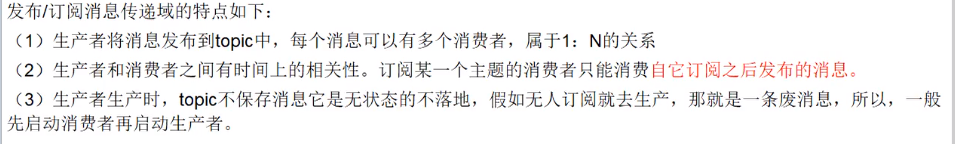
1. 先启动两个消费者，再生产6条消息，请问，消费情况如何？





结果很明显是一人一半，平均分配（有点负载均衡和轮询的机制）

#### Destination是Topic类型（pub/sub）





##### 发布主题生产者

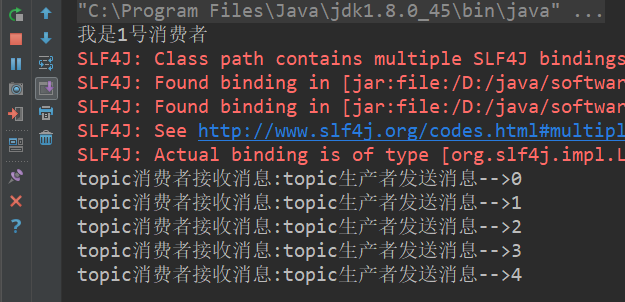
public class JmsProduce\_Topic {  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *Topic\_NAME* = "topic01";  
  
 public static void main(String[] args) throws JMSException {  
 //创建ConnectionFactory对象  
 ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(*DEFAULT\_BROKER\_BIND\_URL*);  
 //获取连接，并开启连接  
 Connection connection = factory.createConnection();  
 connection.start();  
 //获取Session对象  
 Session session = connection.createSession(false,Session.*AUTO\_ACKNOWLEDGE*);  
 //创建目的地  
 Topic topic = session.createTopic(*Topic\_NAME*);  
 //创建生产者对象  
 MessageProducer messageProducer = session.createProducer(topic);  
 for (int i = 0; i < 5; i++) {  
 //创建消息  
 TextMessage textMessage = session.createTextMessage("topic生产者发送消息-->" + i);  
 //使用生产者对象把消息发送到消息队列中  
 messageProducer.send(textMessage);  
 }  
 //关闭资源  
 messageProducer.close();  
 session.close();  
 connection.close();  
 System.*out*.println("Topic消息发布到MQ");  
 }  
}

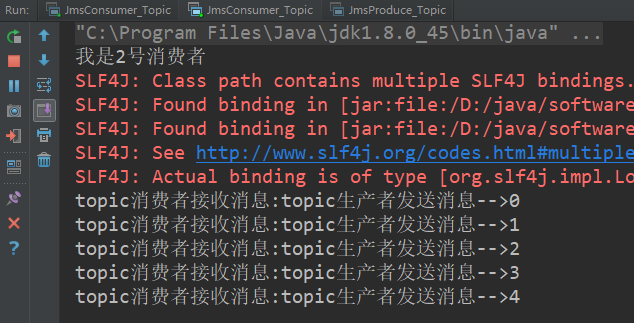
##### 订阅主题消费者

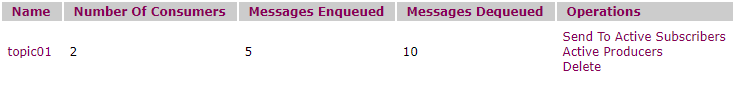
public class JmsConsumer\_Topic {  
  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *Topic\_NAME* = "topic01";  
 public static void main(String[] args) throws JMSException, IOException {  
 System.*out*.println("我是2号消费者");  
 ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(*DEFAULT\_BROKER\_BIND\_URL*);  
 Connection connection = factory.createConnection();  
 connection.start();  
 Session session = connection.createSession(false, Session.*AUTO\_ACKNOWLEDGE*);  
 //创建目的地  
 Topic topic = session.createTopic(*Topic\_NAME*);  
 //创建消费者对象  
 MessageConsumer messageConsumer = session.createConsumer(topic);  
 //使用监听的方式来监听消息  
 messageConsumer.setMessageListener(new MessageListener() {  
 @Override  
 public void onMessage(Message message) {  
 if(null != message && message instanceof TextMessage){  
 TextMessage textMessage = (TextMessage) message;  
 try {  
 System.*out*.println("topic消费者接收消息" +":"+textMessage.getText());  
 } catch (JMSException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 });  
 System.*in*.read();  
 messageConsumer.close();  
 session.close();  
 connection.close();  
 }  
}

##### 正常情况下：

先启动了订阅，再启动生产，不然发送的都是废消息







##### 错误情况下：

先启动生产，在启动订阅，订阅方不能接收到任何消息

#### 队列和主题有什么区别呢？

Queue消息若暂时没有被消费，也不会丢失，会存储在队列中；若先生产消息，则就算有再多的消费者，生产的多条消息都将会被第一个消费者消费，其他消费者拿不到任何消息；若是先启动多个消费者，再启动生产者，那么消息将负载均衡给不同的消费者进行消费（平均分配）

而Topic采用的是发布-订阅模式，如果发布消息的同时，并没有订阅方去消费消息，则消息会丢失；如果有多个订阅者，那么这些订阅者都将会收到消息

工作模式：

传递完整性：

处理效率：

### 6.JMS组成的四大元素



四大元素：

JMS provider：实现JMS接口和规范的消息中间件，也就是我们的MQ服务器

JMS producer：消息生产者，创建和发送JMS消息的客户端应用

JMS consumer：消息消费者，接收和处理JMS消息的客户端应用

JMS message：JMS消息

#### Message消息头

1. JMSDestination:消息发送的目的地，主要指queue和topic
2. JMSDeliveryMode:

持久和非持久模式。什么叫持久和非持久模式？

一条持久性的消息：应该被传送“一次仅仅一次”，这就意味着如果JMS提供者出现故障，该消息并不会丢失

一条非持久的消息：最多会传送一次，这就意味着服务器若出现故障，该消息将永远丢失。

1. JMSExpiration:

可以设置消息在一定时间后过期，默认是永不过期的；

表示一个消息的有效期。只有在这个有效期内，消息消费者才可以消费这个消息。

如果消息发送后，在消息过期时间之后消息还没有被发送到目的地，则该消息被清除

默认值为0，表示消息永不过期。可以通过下面的方式设置：  
       producer.setTimeToLive(3600000); //有效期1小时 （1000毫秒 \* 60秒 \* 60分）

1. JMSPriority：

消息的优先级。0-4为普通消息，5-9为加急消息。可以通过下面方式设置：

producer.setPriority(9);

1. JMSMessageID：（解决消息中间件幂等性问题）

一个字符串用来唯一标示一个消息。

#### 消息体

封装具体的消息数据

有5种消息体格式：

TextMessage:普通字符串，包含一个String

MapMessage：一个Map类型的消息，key为String类型，值为java基本类型

BytesMessage：二进制数组消息，用标准流操作来顺序的填充和读取

StreamMessage：java数据流消息，用标准流操作来顺序地填充和读取

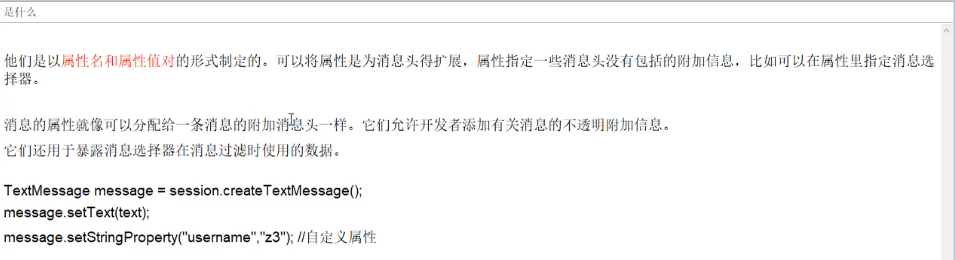
ObjectMessage：对象消息，包含一个可序列化的对象

发送和接收的消息类型必须一致

#### 消息属性

如果需要除了消息头字段以外的值，那么可以使用消息属性

识别/去重/重点标注等操作非常有用的方法



### 7.JMS的可靠性(重要！！！)

MQ挂了，那么消息的持久化和丢失情况分别如何？

#### 持久性

##### 参数设置说明：

非持久：当服务器宕机，消息不存在

messageProducer.setDeliveryMode(DeliveryMode.*NON\_PERSISTENT*);

持久：当服务器宕机，消息依然存在（不设置默认是持久的）

messageProducer.setDeliveryMode(DeliveryMode.*PERSISTENT*);

##### 持久的Queue：

持久化消息是队列默认传送模式，此模式保证这些消息只能被传送一次和成功使用一次。对于这些消息，可靠性是优先考虑的因素。可靠性的另一个重要方面是确保持久性消息传送至目标后，消息服务在向消费者传递他们之前不会丢失这些消息

##### 持久的Topic：

###### 代码

发布者：

public class JmsProduce\_Topic\_Persist {  
  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *Topic\_NAME* = "topic-persist";  
  
 public static void main(String[] args) throws JMSException {  
 //创建连接工厂  
 ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(*DEFAULT\_BROKER\_BIND\_URL*);  
 //创建连接对象  
 Connection connection = factory.createConnection();  
 //创建Session对象  
 Session session = connection.createSession(false,Session.*AUTO\_ACKNOWLEDGE*);  
 //创建目的地  
 Topic topic = session.createTopic(*Topic\_NAME*);  
 //创建生产者对象  
 MessageProducer producer = session.createProducer(topic);  
 //声明我们生产的是一个持久化的topic  
 producer.setDeliveryMode(DeliveryMode.*PERSISTENT*);  
 //开启连接  
 connection.start();  
  
 for (int i = 1; i <= 5; i++) {  
 TextMessage textMessage = session.createTextMessage("msg-persist" + i);  
 producer.send(textMessage);  
 }  
 producer.close();  
 session.close();  
 connection.close();  
 }  
}

订阅者：

public class JmsConsumer\_Topic\_Persist {  
  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *Topic\_NAME* = "topic-persist";  
 public static void main(String[] args) throws JMSException {  
 System.*out*.println("我是消费者zhangsan");  
 //创建ConnectionFactory对象  
 ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(*DEFAULT\_BROKER\_BIND\_URL*);  
 //获取连接，并开启连接  
 Connection connection = factory.createConnection();  
 //表明有一个叫zhangsan的订阅  
 connection.setClientID("zhangsan");  
 //获取Session对象  
 Session session = connection.createSession(false,Session.*AUTO\_ACKNOWLEDGE*);  
 //创建目的地  
 Topic topic = session.createTopic(*Topic\_NAME*);  
 //创建订阅者去订阅这个topic  
 TopicSubscriber topicSubscriber = session.createDurableSubscriber(topic, "remark");  
 connection.start();  
 Message message = topicSubscriber.receive();  
 while (null != message) {  
 TextMessage textMessage = (TextMessage) message;  
 System.*out*.println("收到的持久化topic" + textMessage.getText());  
 message = topicSubscriber.receive(1000L);  
 }  
 //关闭资源  
 session.close();  
 connection.close();  
 System.*out*.println("Topic消息发布到MQ");  
 }  
}

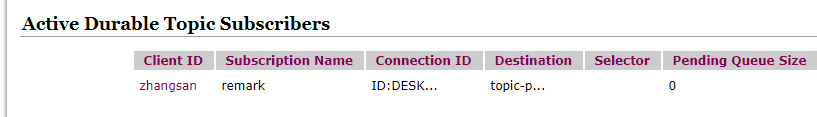
控制台：

1）先启动订阅者：

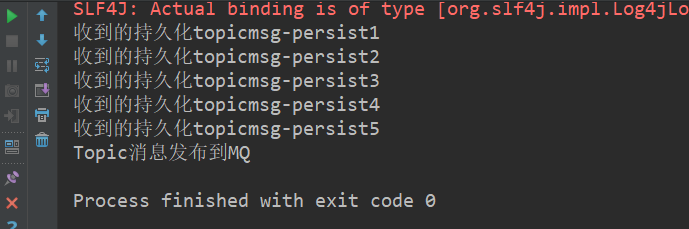
启动订阅者的同时设置了阻塞时间（监听时间为1s），若在1s之内没有接收到其他消息将关闭连接即订阅者将会处于离线状态



处于激活状态的，持久化的，主题的订阅者

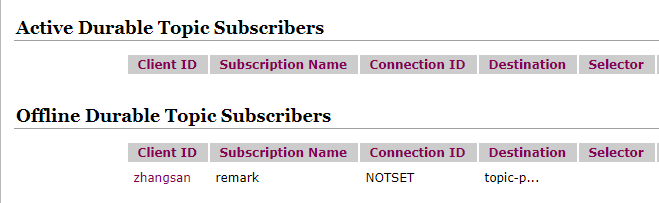


2）再启动发布者



消费者数量为1，进队消息数和出队消息数（被消费）都是5条





1. 有一种情况是先启动订阅者，然后马上关闭（不让它持续receive）；然后启动发布者，再次启动订阅者；订阅者将能够收到发布者发布的消息。

结论：

1. 一定要先运行一次订阅者，等于是向MQ注册，类似我订阅了这个公众号。
2. 然后再运行生产者发送消息
3. 此时，订阅者无论是否在线都将会接收到消息；若是现在不在线的话，将会在下次上线的时候把没有接收到的消息全部接收下来。

类似微信公众号订阅发布

#### 事务

##### Producer提交时的事务：

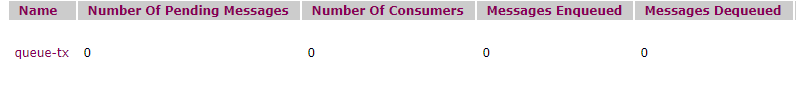
False:只要执行send，进入到队列中

关闭事务，那第二个签收参数的设置需要有效

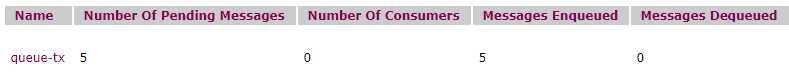
True:先执行send再执行commit，消息才会被真正地提交到队列中

消息需要批量发送，需要缓冲处理

当事务设置为true并且没有提交事务地情况下：控制台显示信息：



当事务设置为true并且在session.close()之前关闭了事务，控制台显示信息：



说明此时消息已经真正地被提交到了队列中

我们可以通过以下代码来保证当程序出现异常时，能够实现回滚，避免往队列中发送错误数据：

try {  
 session.commit();  
} catch (Exception e) {  
 e.printStackTrace();  
 session.rollback();  
} finally {  
 if(null != session){  
 session.close();  
 }  
}

**事务偏生产者，签收偏消费者**

**当事务选择了true,则签收的设置无关紧要了；**

**当事务选择了false,则签收的设置就变得关键**

Consumer消费时地事务：

当Consumer处设置事务为true时并且没有提交情况下，明明消费者消费了消息，还是会有与生产者发出的个数相同的消息出现在队列中。（后面会详细分析）

#### 签收

##### 非事务

###### 自动签收

Session.*AUTO\_ACKNOWLEDGE*

###### 手动签收

Session.*CLIENT\_ACKNOWLEDGE*

签收主要是看消费者，因此我们在生产者这边的签收选择了AUTO\_ACKNOWLEDGE;并且事务选择了false,避免了事务因素的干扰；

**在消费者处，签收选择了手动签收CLIENT\_ACKNOWLEDGE；当然事务还是选择了false**

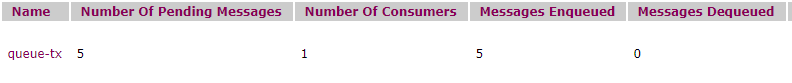
**在这样的情况下，会发生签收过的消息可以被重复签收的问题**

JmsProduce\_tx:

public class JmsProduce\_tx {  
  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *queue\_NAME* = "queue-tx";  
 public static void main(String[] args) throws JMSException {  
 ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(*DEFAULT\_BROKER\_BIND\_URL*);  
 Connection connection = factory.createConnection();  
 connection.start();  
 Session session = connection.createSession(false, Session.*AUTO\_ACKNOWLEDGE*);  
 //创建Destnation  
 Queue queue = session.createQueue(*queue\_NAME*);  
 //创建生产者对象  
 MessageProducer messageProducer = session.createProducer(queue);  
 for (int i = 1; i <=5 ; i++) {  
 //创建消息  
 TextMessage textMessage = session.createTextMessage("tx msg--->" + i);  
 messageProducer.send(textMessage);  
 }  
 //关闭资源  
 messageProducer.close();  
 session.close();  
 connection.close();  
 System.*out*.println("生产消息发送完毕");  
 }  
}

JmsConsumer\_tx:

public class JmsConsumer\_tx {  
  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *queue\_NAME* = "queue-tx";  
 public static void main(String[] args) throws JMSException, IOException {  
 ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(*DEFAULT\_BROKER\_BIND\_URL*);  
 Connection connection = factory.createConnection();  
 connection.start();  
 Session session = connection.createSession(false, Session.*CLIENT\_ACKNOWLEDGE*);  
 Queue queue = session.createQueue(*queue\_NAME*);  
 MessageConsumer messageConsumer = session.createConsumer(queue);  
 while (true){  
 TextMessage textMessage = (TextMessage) messageConsumer.receive(5000L);  
 if (textMessage != null){  
 System.*out*.println("消费者接收消息" + textMessage.getText());  
 }else {  
 break;  
 }  
 }  
 messageConsumer.close();  
 session.close();  
 connection.close();  
 System.*out*.println("完成接收消息");  
 }  
}



可以发现进队消息数一直是5，出队消息数一直是0，消息留在了队列中可以被无限制消费

而把消费者的连接

过了5秒中之后消费方不再监听queue(灯灭了)，队列中依然还有消息滞留

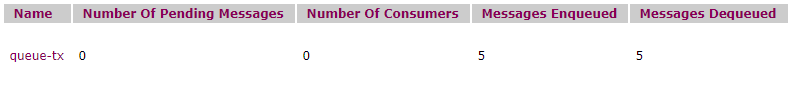


这个问题怎么解决？

需要客户端调用acknowledge方法手动签收

while (true){  
 TextMessage textMessage = (TextMessage) messageConsumer.receive(5000L);  
 if (textMessage != null){  
 System.*out*.println("消费者接收消息" + textMessage.getText());  
 //客户端调用acknowledge进行手动签收  
 textMessage.acknowledge();  
 }else {  
 break;  
 }  
}

此时控制台：



队列中的消息不存在滞留消息，并且已经全部消费完毕，不存在重复消费问题

###### 允许重复消息：

允许重复签收（用的比较少）

##### 事务模式下

如果按照事务来提交，就默认为自动提交了（就算写成了手动提交CLIENT\_ACKNOWLEDGE），也不起作用。因此消费方的事务设置为true的时候，只需要在session.close之前提交事务session.commit()即可，无论签收是否设置为手动签收都不起作用，仍是自动提交，不会出现消息重复消费的问题，也不需要调用客户端的acknowledge方法

#### 签收和事务的关系：

在事务会话中，当一个事务被成功提交则消息被自动签收

如果事务回滚，则消息会被再次传送

非事务会话中，消息何时被确认取决于创建会话时的应答模式（ackknowledge mode）

### 8.ActiveMq的Broker

**相当于一个ActiveMq服务器实例**，**说白了Broker其实就是实现了用代码的形式启动ActiveMq，将MQ嵌入到java代码中，以便随时用，随时启动，在用的时候再去启动这样能节省了资源，也保证了可靠性**；以前我们都需要在linux服务器上去起Activemq的进程才能使用Activemq，而现在我们可以使用Activemq的Broker，他作为ActiveMq服务器实例提供给我们使用；

不同的conf配置文件模拟不同的实例；我们可以在conf目录下复制多个不同的activemq.xml配置文件,然后在bin目录下使用 ./activemq start xbean:file:/ myactiveMQ/apache-activemq-5.15.9/conf/activemq02.xml命令来起不同的activemq消息服务器实例

那如何用代码的形式启动ActiveMq，将MQ嵌入到java代码中？现在来学习一下嵌入式Broker。用ActiveMQ Broker作为独立的消息服务器来构建Java应用。ActiveMQ支持在vm中通信基于嵌入式的broker,能够无缝地集成其他java应用

1. pom.xml

再次引入pom的依赖

<dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-databind</artifactId>  
 <version>2.9.5</version>  
</dependency>

1. EmbedBroker

public static void main(String[] args) throws Exception {  
 //ActiveMQ支持在vm中通信基于嵌入式的broker  
 //将迷你版的activemq潜入到java程序里  
 BrokerService brokerService = new BrokerService();  
 brokerService.setUseJmx(true);  
 brokerService.addConnector("tcp://localhost:61616");  
 brokerService.start();  
}

1. 队列验证

再次启动之前我们测试使用过的JmsProducer和JmsConsumer，验证可以通过嵌入式的MQ来发送和接收消息

### 9.Spring整合ActiveMq

#### 1. Maven修改，需要添加Spring支持JMS的包

在原先的依赖上继续添加和Spring整合相关的dependency

<!--activemq对JMS的支持，整合Spring和ActiveMq-->  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-jms</artifactId>  
 <version>4.3.23.RELEASE</version>  
</dependency>  
<!--activemq所需要的pool包配置-->  
<dependency>  
 <groupId>org.apache.activemq</groupId>  
 <artifactId>activemq-pool</artifactId>  
 <version>5.15.9</version>  
</dependency>  
<!--Spring-Aop等相关jar-->  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-core</artifactId>  
 <version>4.3.23.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>4.3.23.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-aop</artifactId>  
 <version>4.3.23.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-orm</artifactId>  
 <version>4.3.23.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-tx</artifactId>  
 <version>4.3.23.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.aspectj</groupId>  
 <artifactId>aspectjrt</artifactId>  
 <version>1.6.1</version>  
</dependency>  
<dependency>  
 <groupId>aspectj</groupId>  
 <artifactId>aspectjweaver</artifactId>  
 <version>1.5.3</version>  
</dependency>  
<dependency>  
 <groupId>cglib</groupId>  
 <artifactId>cglib</artifactId>  
 <version>2.1\_2</version>  
</dependency>

#### 2.Spring的配置文件:applicationContext.xml

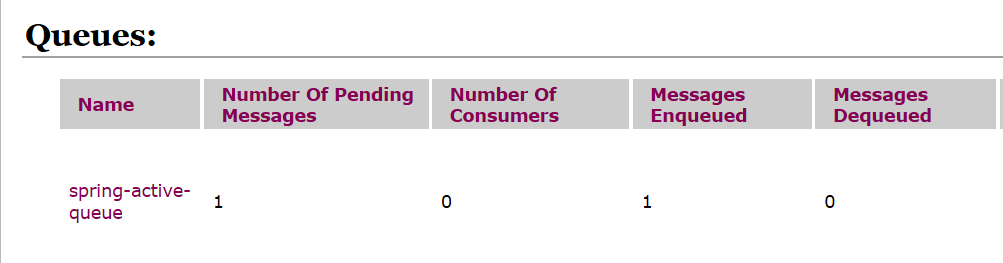
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/c"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  
 <!--开启包的自动扫描-->  
 <context: component-scan base-package="com.atguigu.activemq" />  
  
 <!--配置生产者-->  
 <bean id="jmsFactory" class="org.apache.activemq.pool.PooledConnectionFactory" destroy-method="stop">  
 <property name="connectionFactory">  
 <!--真正可以产生Connection的ConnectionFactory,由对应的JMS服务厂商提供-->  
 <bean class="org.apache.activemq.ActiveMQConnectionFactory" >  
 <property name="brokerURL" value="tcp://192.168.18.105:61616"/>  
 </bean>  
 </property>  
 </bean>  
  
 <!--这个是队列目的地，点对点的-->  
 <bean id="destinationQueue" class="org.apache.activemq.command.ActiveMQQueue">  
 <constructor-arg index="0" value="spring-active-queue"/>  
 </bean>  
  
  
 <!--Spring提供的JMS工具类，它可以进行消息发送，接收等-->  
 <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">  
 <property name="connectionFactory" ref="jmsFactory"/>  
 <property name="defaultDestination" ref="destinationQueue"/>  
 <property name="messageConverter">  
 <bean class="org.springframework.jms.support.converter.SimpleMessageConverter"/>  
 </property>  
 </bean>  
</beans>

#### 3队列

生产者：SpringMQ\_Producer

*/\*\*  
 \*  
 \* 其实就是用Spring给我们提供的JmsTemplate对象来发送消息，  
 \* 取代了之前创建ConnnectionFactory,Connection和Session对象  
 \* 减少了许多繁杂冗余的代码  
 \*/*@Service  
public class SpringMQ\_Producer {  
  
 @Autowired  
 private JmsTemplate jmsTemplate;  
  
 public static void main(String[] args) {  
 ApplicationContext ioc = new ClassPathXmlApplicationContext("applicationContext.xml");  
  
 */\*\*  
 \* 这里把当前生产者类当作是我们3层架构中的Service层  
 \* 通过@Service注解的方式把当前类作为对象注入到容器中；  
 \* 因此，这里就能通过ioc.getBean("xxx")的方式获取到当前类的对象  
 \*/* SpringMQ\_Producer springMQ\_producer = (SpringMQ\_Producer) ioc.getBean("springMQ\_Producer");  
 */\*\*  
 \* 这里解释一下为什么不用send方法的另外一个构造函数：  
 \* 另外一个构造函数是：send(String destination,MessageCreator messageCreator)  
 \* 因为destination这个参数我们已经在applicationContext.xml中做了配置，  
 \* 让他自动注入到JmsTemplate对象中（能用配置，尽量用配置，减少硬编码）  
 \*/* springMQ\_producer.jmsTemplate.send(new MessageCreator() {  
 @Override  
 public Message createMessage(Session session) throws JMSException {  
 TextMessage textMessage = session.createTextMessage("Spring和ActiveMq的整合Case111...");  
 return textMessage;  
 }  
 });  
  
 System.*out*.println("send task over !!!!");  
 }  
}

控制台显示：消息已经成功发出



消费者：SpringMQ\_Consumer

（不是很理解周洋老师为什么要用jmsTemplate.receiveAndConvert）

@Service  
public class SpringMQ\_Consumer {  
 @Autowired  
 private JmsTemplate jmsTemplate;  
 public static void main(String[] args) throws JMSException {  
 //1.获取IOC容器  
 ApplicationContext ioc = new ClassPathXmlApplicationContext("applicationContext.xml");  
 //2.获取当前类的对象  
 SpringMQ\_Consumer springMQ\_consumer = (SpringMQ\_Consumer) ioc.getBean("springMQ\_Consumer");  
 //3.接收消息,返回textMessage对象  
 //receiveAndConvert方法的返回值类型是Object，可以把消息转换成任何一种你想转成的对象  
 //String retValue = (String) springMQ\_consumer.jmsTemplate.receiveAndConvert();  
 //System.out.println(retValue);  
 TextMessage message = (TextMessage) springMQ\_consumer.jmsTemplate.receive();  
 System.*out*.println(message.getText());  
 }  
}

#### 4.主题

**主题目的地的代码，其实队列一模一样，这就是Spring Aop面向切面编程思想的体现；我们把目的地的类型配在了xml配置中，当我们需要使用queue的Destination时，就使用代表队列的bean；当我们需要使用topic类型的Destination时，就使用代表主题的bean；并且在JmsTemplate中引用你想使用的目的地类型。**

**补充：我们在配置xml的同时可以复习一下Spring框架的相关知识：比如Spring依赖注入的3中方式：**

**1.set方法注入；2.构造方法注入3.注解注入；**

**把value属性注入到destination中使用的是构造方法注入；**

**而把destination和ConnectionFactory注入到JmsTemplate中使用的是set方法注入，我们可以按住ctrl键选中<proerty>标签中的name点进到JmsTemplate类中查看到setDefaultDestination方法；**

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd">  
 <!--开启包的自动扫描-->  
 <!--idea中约束为灰色时表示没有被使用到-->  
 <context:component-scan base-package="com.atguigu.activemq"/>  
  
 <!--配置生产者-->  
 <bean id="jmsFactory" class="org.apache.activemq.pool.PooledConnectionFactory" destroy-method="stop">  
 <property name="connectionFactory">  
 <!--真正可以产生Connection的ConnectionFactory,由对应的JMS服务厂商提供-->  
 <bean class="org.apache.activemq.ActiveMQConnectionFactory" >  
 <property name="brokerURL" value="tcp://192.168.18.105:61616"/>  
 </bean>  
 </property>  
 </bean>  
  
 <!--这个是队列目的地，点对点的-->  
 <bean id="destinationQueue" class="org.apache.activemq.command.ActiveMQQueue">  
 <constructor-arg index="0" value="spring-active-queue"/>  
 </bean>  
  
 <!--这个是主题目的地，发布/订阅模式-->  
 <bean id="activeMQTopic" class="org.apache.activemq.command.ActiveMQTopic" >  
 <constructor-arg index="0" value="spring-active-topic"/>  
 </bean>  
 <!--Spring提供的JMS工具类，它可以进行消息发送，接收等-->  
 <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">  
 <property name="connectionFactory" ref="jmsFactory"/>  
 <!--<property name="defaultDestination" ref="destinationQueue"/>-->  
 <property name="defaultDestination" ref="activeMQTopic"/>  
 <property name="messageConverter">  
 <bean class="org.springframework.jms.support.converter.SimpleMessageConverter"/>  
 </property>  
 </bean>  
</beans>

#### 5. 在Spring里面实现消费者不启动直接通过配置监听完成

消费者使用监听的方式来消费消息，我们之前是new MessageListener()接口使用匿名内部类的方式来重写onMessage方法，且在onMessage中完成了对于消息的接收和消费。

现在需要3步：

1. 配置applicationContext.xml文件

可以使用bean标签注入MyMessageListener

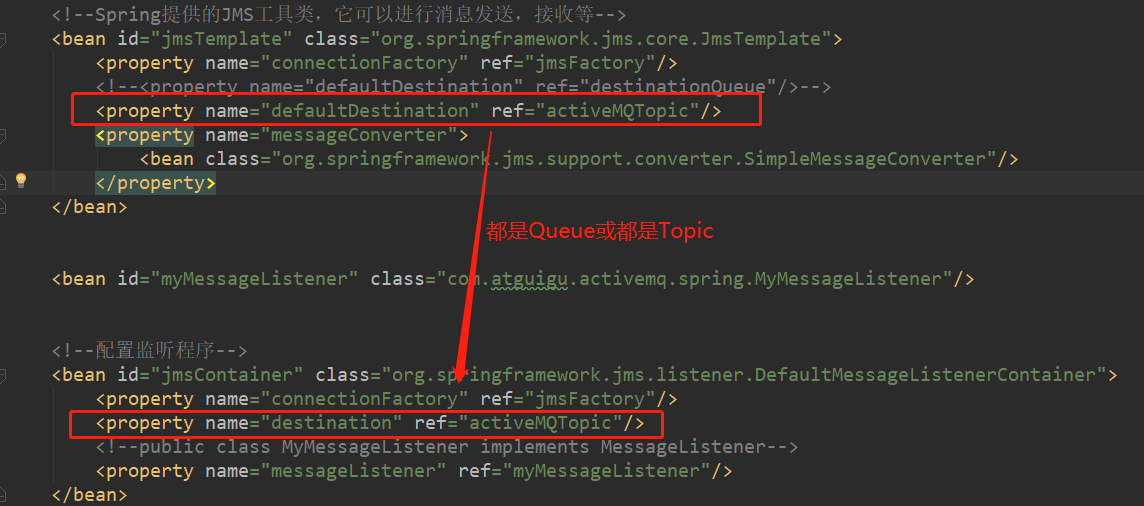
<bean id="myMessageListener" class="com.atguigu.activemq.spring.MyMessageListener"/>  
<!--配置监听程序-->  
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">  
 <property name="connectionFactory" ref="jmsFactory"/>  
 <property name="destination" ref="activeMQTopic"/>  
 <!--public class MyMessageListener implements MessageListener-->  
 <property name="messageListener" ref="myMessageListener"/>  
</bean>

也可以使用@Component注解把MyMessageListener直接加入到容器中

@Component  
public class MyMessageListener implements MessageListener {

需要注意配置文件中:

切记切记！！！！！



1. 定义一个类MyMessageListener去实现MessageListener接口（实现监听功能），并使用@Component注解把这个类注入到Spring的ioc容器中，这样DefaultMessageListenerContainer才能获取到MyMessageListener的ref(引用)，这样才能注入成功！！

@Component  
public class MyMessageListener implements MessageListener {  
 @Override  
 public void onMessage(Message message) {  
  
 if(null != message && message instanceof TextMessage){  
 TextMessage textMessage = (TextMessage) message;  
 try {  
 System.*out*.println(textMessage.getText());  
 } catch (JMSException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

1. 只需要启动生产者，不需要启动消费者，自动会监听记录

### 10.Springboot整合ActiveMq

#### 队列

##### 生产者

1. 新建Maven工程并设置报名和类名，工程名：boot\_mq\_produce,包名：com.atguigu.boot.activemq
2. pom.xml添加依赖

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>1.5.9.RELEASE</version>  
 </parent>  
  
 <groupId>com.atguigu</groupId>  
 <artifactId>boot\_mq\_produce</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <maven.compiler.target>1.8</maven.compiler.target>  
 </properties>

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-activemq</artifactId>  
 <version>2.1.5.RELEASE</version>  
 </dependency>  
 </dependencies>  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
</project>

1. 配置application.yml

**server:  
 port:** 7777  
  
**spring:  
 activemq:  
 broker-url:** tcp://192.168.18.105:61616 *#MQ服务器地址* **user:** admin  
 **password:** admin  
 **jms:  
 pub-sub-domain: false** *#false = Queue(这个配置不写默认就是队列) True = Topic  
  
#自己定义队列名称***myqueue:** boot-activemq-queue

1. 配置bean

@Component

//注解@EnableJms设置在@Configuration类上，用来声明对 JMS 注解的支持

@EnableJms  
public class ConfigBean {  
 */\*\*  
 \* 使用@Value注解从yml配置文件中读取值  
 \*/* @Value("${myqueue}")  
 private String myqueue;  
 */\*\*  
 \** ***@Bean注解相当于xml配置中的<bean>标签*** *\* 把对象注入到容器中  
 \*/* @Bean  
 public Queue queue(){  
 return new ActiveMQQueue(myqueue);  
 }  
}

1. Queue\_Produce

@Component  
public class Queue\_Producer {  
 @Autowired  
 private JmsMessagingTemplate jmsMessagingTemplate;  
 @Autowired  
 private Queue queue;  
 public void produceMsg(){  
 jmsMessagingTemplate.convertAndSend(queue,"\*\*\*\*\*: " + UUID.*randomUUID*().toString().substring(0,6));  
 }  
}

1. 主启动类

@SpringBootApplication  
public class MainApp\_Produce {  
 public static void main(String[] args) {  
 SpringApplication.*run*(MainApp\_Produce.class,args);  
 }  
}

1. 测试单元：使用触发投递需要多谢一个测试类，点一下，发送一个消息

@SpringBootTest(classes = MainApp\_Produce.class)  
@RunWith(SpringJUnit4ClassRunner.class)  
@WebAppConfiguration  
public class TestActiveMQ {  
 @Autowired  
 private Queue\_Producer queue\_producer;  
 @Test  
 public void send() throws Exception{  
 queue\_producer.produceMsg();  
 }  
}

过程总结：

1. 在applica配置文件中配置队列名

Boot-active-queue

1. 在配置类中读取到队列名，并根据队列名创建出Queue类型的ActiveMQ目的地对象
2. 使用jmsMessagingTemplate.send(queue)来发送消息
3. 启动主配置类，就能使消息发送出去
4. 以上的投送方式属于触发投递；另外还有一种业务场景：间隔定投；

新需求：要求每隔3秒钟往MQ中推送消息以下定时发送case，案列修改

1. 修改Queue\_Produce新增定时投递方法

//间隔3秒钟定投  
@Scheduled(fixedDelay = 3000)  
public void produceMsgScheduled(){  
 jmsMessagingTemplate.convertAndSend(queue,"\*\*\*\*\*Scheduled: " + UUID.*randomUUID*().toString().substring(0,6));  
}

1. 修改主启动类MainApp\_Produce

@SpringBootApplication  
//使@Scheduled注解生效  
@EnableScheduling  
public class MainApp\_Produce {  
 public static void main(String[] args) {  
 SpringApplication.*run*(MainApp\_Produce.class,args);  
 }  
}

在Spring boot中，主配置类就好像一个大管家，只有它同意你开启某个注解，你才能使用该注解去实现某个功能

1. 直接开始主启动类MainApp\_Produce，间隔发消息ok

##### 消费者

1. 新建Maven工程
2. 配置pom.xml（和boot\_mq\_produce工程的一致）
3. 配置文件application.yml

**server:  
 port:** 8888  
  
**spring:  
 jms:  
 pub-sub-domain: false  
 activemq:  
 broker-url:** tcp://192.168.18.105:61616  
 **user:** admin  
 **password:** admin  
  
*#自己定义队列名称***myqueue:** boot-activemq-queue

1. 编写Queue\_Consumer来接收消息

@Component  
public class Queue\_Consumer {  
 //使用注解的方式实现对消息的监听  
 @JmsListener(destination = "${myqueue}" )  
 public void receive(TextMessage textMessage) throws JMSException {  
 System.*out*.println("消费者收到消息"+ textMessage.getText());  
 }  
}

1. 开始主启动类MainApp\_Consumer

@SpringBootApplication  
public class MainApp\_Consumer {  
 public static void main(String[] args) {  
 SpringApplication.*run*(MainApp\_Consumer.class,args);  
 }  
}

能接收到生产者发过来的消息：



#### 发布订阅

##### Topic生产者：

1. 新建Maven工程并设置包名类名（工程名：boot\_mq\_topic\_produce、包名：com.atguigu.boot.activemq.topic.produce）
2. pom.xml（和之前一致）
3. application.yml

**server:  
 port:** 8081  
**spring:  
 activemq:  
 broker-url:** tcp://192.168.18.105:61616 *#MQ服务器地址* **user:** admin  
 **password:** admin  
 **jms:  
 pub-sub-domain: true***#自己定义队列名称***mytopic:** boot-activemq-topic

1. 配置Bean --->类似Spring框架中的applicationContext.xml文件

补充一点：注入到容器中的这个bean对象其实就是我们在学ActiveMQ的HelloWorld的时候，创建的那个Destination对象（Queue queue = session.createQueue(*Queue\_NAME*);）

@Component  
public class ConfigBean {  
  
 @Value("${mytopic}")  
 private String topicName;

//把Topic对象注入到ioc容器中  
 @Bean  
 public Topic topic(){  
 return new ActiveMQTopic(topicName);  
 }  
}

1. Topic\_Produce

@Component  
public class Topic\_Produce {  
 @Autowired  
 private JmsMessagingTemplate jmsMessagingTemplate;  
 @Autowired  
 private Topic topic;  
 */\*\*  
 \* 偷个懒直接使用定时调度来发送消息  
 \* 免得再写一个测试类，点一下发送一次消息  
 \*/* @Scheduled(fixedDelay = 3000)  
 public void produceTopic(){  
 jmsMessagingTemplate.convertAndSend(topic ,"主题消息：" +UUID.*randomUUID*().toString().substring(0,4));  
 }  
}

1. 主启动类MainApp\_TopicProduce

@SpringBootApplication  
@EnableScheduling  
public class MainApp\_TopicProduce {  
 public static void main(String[] args) {  
 SpringApplication.*run*(MainApp\_TopicProduce.class,args);  
 }  
}

##### Topic消费者：

1. 新建Maven工程并设置包名和类名（工程名：boot\_mq\_topic\_consumer 包名：com.atguigu.boot.activemq.topic.comsumer）
2. pom.xml（和之前一致）
3. application.yml

这里需要模拟两个消费者在同时监听生产者发布的消息（在开始主启动类之后，再去改写application.yml中的端口号，再开始主启动类）

**server:  
 port:** 8082  
**spring:  
 activemq:  
 broker-url:** tcp://192.168.18.105:61616  
 **user:** admin  
 **password:** admin  
 **jms:  
 pub-sub-domain: true  
mytopic:** boot-activemq-topic

1. Topic\_Consumer

@Component  
public class Topic\_Consumer {  
 @JmsListener(destination = "${mytopic}")  
 public void receive(TextMessage textMessage) throws JMSException {  
 System.*out*.println("topic消费者收到订阅的主题" + textMessage.getText());  
 }  
}

1. 模拟两个订阅者主启动类，先启动订阅者，再启动发布者（MainApp8082/MainApp8083）



1. 在开始生产者的主启动类之后（每隔3秒发送一次消息），消费者就能收到消息

Scheduled\_topic send ok!

Scheduled\_topic send ok!

Scheduled\_topic send ok!

Scheduled\_topic send ok!

Scheduled\_topic send ok!

Scheduled\_topic send ok!

Scheduled\_topic send ok!

Scheduled\_topic send ok!

**8082端口号代表的消费者收到的消息：**

topic消费者收到订阅的主题主题消息：8160

topic消费者收到订阅的主题主题消息：4b70

topic消费者收到订阅的主题主题消息：b52d

topic消费者收到订阅的主题主题消息：cb30

topic消费者收到订阅的主题主题消息：9010

topic消费者收到订阅的主题主题消息：cf1e

topic消费者收到订阅的主题主题消息：26ac

topic消费者收到订阅的主题主题消息：84c0

**8083端口号代表的消费者收到的消息：**

topic消费者收到订阅的主题主题消息：33b8

topic消费者收到订阅的主题主题消息：c4ba

topic消费者收到订阅的主题主题消息：450d

topic消费者收到订阅的主题主题消息：6eca

topic消费者收到订阅的主题主题消息：6a07

topic消费者收到订阅的主题主题消息：77b7

topic消费者收到订阅的主题主题消息：2db1

topic消费者收到订阅的主题主题消息：1894

### 11.ActiveMq的传输协议

#### 面试题

1. 默认的61616端口号如何修改

在activemq.xml中的<transportConnector>标签中进行替换

1. 你生产上的链接协议如何配置？使用tcp吗？

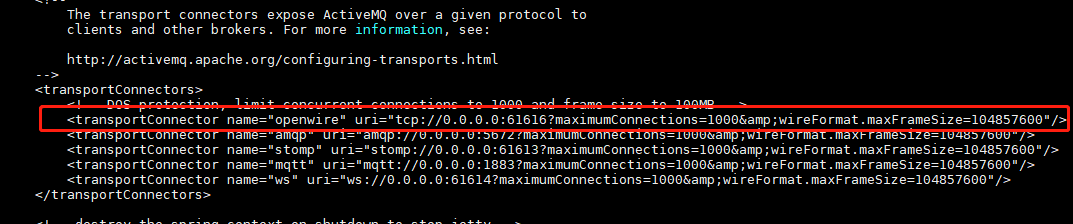
更多使用的是NIO

#### 是什么？

ActiveMQ支持的client-broker通讯协议有：TCP、NIO、UDP、SSL、Https、VM

其中配置Transport Connector的文件在activeMQ安装目录的conf/activemq.xml中的<transportConnectors>标签之内





在上文给出的头部信息中，URI描述信息的头部都是采用协议名称：

比如：描述amqp协议的监听端口时，采用的URI描述格式为“amqp://……”

描述Stomp协议的监听端口时，采用的URI描述格式为“Stomp://……”

唯独在进行openwire协议描述时，URI头采用的是：”tcp://…”

这是因为ActiveMQ默认的消息协议就是openwire

#### 有哪些？

1、TCP协议：这是默认的broker配置，TCP的client监听端口是：61616

* 在网络传输前必须要序列化数据，消息是通过一个叫Wire Protocol的来序列化成字节流；默认情况下ActiveMQ把Wire Protocol叫做OpenWire,它的目的是促使网络上的效率和数据快速交互
* TCP的URI形式为 tcp://hostname:port?key=value&key=value&key=value

Transport Options详情：http://activemq.apache.org/configuration-version-5-transports

* TCP传输的优点：

1. TCP协议传输可靠性高，稳定性强。
2. 高效性：字节流方式传递，效率很高
3. 有效性，可用性：应用广发，支持任何平台

2、NIO协议

* NIO协议与TCP协议类似，但NIO更侧重于底层的访问操作。它允许开发人员对同一资源有更多的client调用和服务端有更多的负载
* 适合NIO协议的场景：

1. 可能有大量的client去连接到broker上，一般情况下，大量的client去连接broker是被操作系统的线程锁限制的。因此，NIO的实现比TCP需要更少的线程去运行，是所以建议使用NIO协议
2. 可能对于broker有一个很迟钝的网络传输，NIO比TCP提供更好的性能

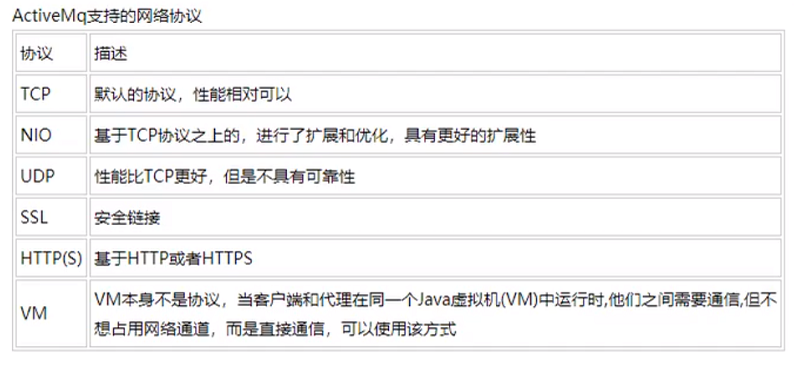
* NIO链接的URI形式：nio://hostname:port?key=value?key=value
* Xml配置：

<transportConnectors>

<transportConnector name="nio" uri="nio://localhost:61618?trace=true"/>

</transportConnectors>

小结：



#### NIO案列演示

1. 修改配置文件activemq.xml

* 如果不特别指定ActiveMq的网络监听端口，那么这些端口都将使用BIO网络IO模型（OpenWire,Stomp,AMQP）;所以为了首先提高单节点的网络吞吐性能，我们需要指定ActiveMQ的网络IO模型
* 如下URI格式头使用”nio”开头，表示这个端口使用的是以TCP协议为基础的NIO网络IO模型

<transportConnectors>

<transportConnector name="nio" uri="nio://0.0.0.0:61618?trace=true"/>

</transportConnectors>



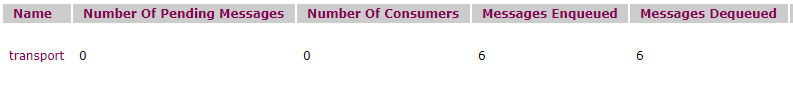
1. 生产和消费两端协议代码修改

**因为我把xml配置中默认基于BIO的tcp协议的那一行配置注释掉了**，因此原先代码中的URI不再适用。

我们在生产者和消费者处都把URI替换为基于NIO的tcp协议，并且把端口号改成与配置文件中一致的，否则会报异常

public static final String *DEFAULT\_BROKER\_BIND\_URL* = "nio://192.168.18.105:61618";

1. 运行验证



消费者接收消息Listenermsg---->1

消费者接收消息Listenermsg---->2

消费者接收消息Listenermsg---->3

消费者接收消息Listenermsg---->4

消费者接收消息Listenermsg---->5

消费者接收消息Listenermsg---->6

这样我们就可以根据业务需求的不同在bio和nio协议之间进行切换

#### NIO案列演示增强

* 上述NIO性能已经不错了，如何进一步优化？
* 问题：URI格式头以nio开头，表示这个端口使用以TCP协议为基础的NIO网络IO模型。但是这样的设置方式，只能使这个端口支持OpenWire协议

怎么让这个端口，既支持NIO网络IO模型，又让他支持多个协议呢？

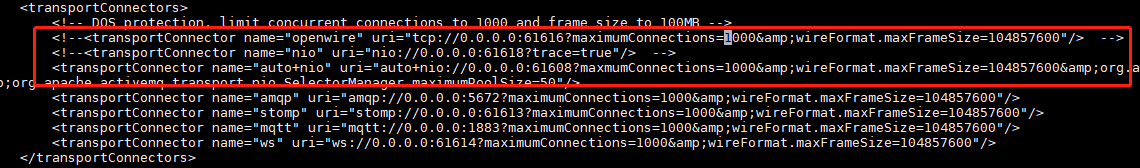
* 解决：

1. 使用Auto关键字：
2. 使用“+“符号来为端口设置多种特性
3. 如果我们既需要一个端口支持NIO网络IO模型，又需要它支持多个协议

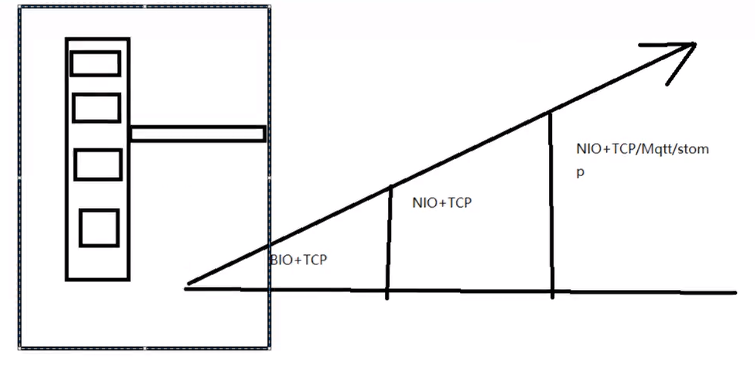
在xml中进行配置auto + nio

<transportConnector name="auto+nio"uri="auto+nio://0.0.0.0:61608? maxConnections=1000&amp;wireFormat.maxFrameSize=104857600&amp;org.apache.activemq.transport.nio.SelectorManager.corePoolSize=20&amp;org.apache.activemq.transport.nio.SelectorManager.maximumPoolSize=50"/>





在xml配置文件把OpenWire和nio注释掉的情况下**还是能发通消息(并且尝试了在不同的协议的情况下分别做了尝试，都能成功发送消息)**，说明以上配置能自动检测（侦测）到我发出的是什么协议的URI（它可以自动侦测到你发的请求是基于Tcp、Stomp、mqtt还是amqp协议），并根据（基于）该协议去发送消息（如左图所示，相当于一个适配器模式！！）



### 12.ActiveMq消息存储和持久化

在面试层面上，面试官若要问你怎样保持ActiveMQ消息的可靠性，需要从4个纬度去回答。持久（模式），事务，签收和可持久化；前三者都属于MQ自带，而可持久化需要借助于外力；

为了保证高可用性，我们可以把收到的消息持久化到磁盘上（可以是mysql或者是oracle数据库），这样的操作就被称作持久化；

另外为了进一步提升高可用性，我们可以把MQ的消息同步到另一台机器的硬盘中，这样一旦MQ所在的机器宕机，不会造成数据的丢失。

#### 官网：

<http://activemq.apache.org/persistence>

在V5.3之后推荐你使用[KahaDB](http://activemq.apache.org/kahadb),在V5.9之后推荐使用 [Replicated LevelDB Store](http://activemq.apache.org/replicated-leveldb-store)

#### 是什么

**面试题**

**请你谈一下AMQ持久化机制**

回忆阳哥讲过的redis持久化机制有几种？

说明：MQ服务器宕机了，消息不会丢失的机制

**为了避免意外宕机以后丢失信息，需要做到重启以后可以恢复消息队列，消息系统一般都会采用持久化机制。ActiveMQ的消息持久化机制有：JDBC，AMQ，KaHaDB和LevelDB,无论采取哪种持久化方式，消息的存储逻辑都是一致的。**

**就是在发送者在消息发送出去后，消息中心首先将消息存储到本地数据文件，内存数据库或者远程数据库等再将消息发送给接收者，成功则将消息从存储中删除，失败则继续尝试发送。**

**消息中心启动之后首先要检查指定的存储位置，如果有未发送成功的消息，则需要把消息发送出去。**

#### 有哪些持久化机制？

##### AMQ Message Store（了解）

基于文件的存储方式，是以前的（V5.3之前）默认消息存储方式，现在不用了

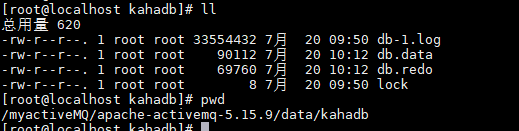
##### KaHaDB消息存储（默认）

基于日志文件，从ActiveMq5.4开始默认的持久化插件。

从ActiveMQ5.4起，官网推荐使用KaHaDB来作为ActiveMQ的持久化机制。



为ActiveMQ服务器实例的消息配置持久化。默认的持久化机制是KaHaDB存储



说明：

KaHaDB是目前默认的存储方式，可用于任何场景，提高了性能和恢复能力

消息存储使用一个事务日志和仅仅用一个索引文件来存储它所有的地址

KaHaDB是一个专门针对消息持久化的解决方案，它对典型的消息使用模式进行了优化

数据被追加到data logs中。当不再需要log文件中的数据的时候，log会被丢弃

KaHaDB的存储原理：

KaHaDB在消息保存目录中只有4类文件和一个lock，跟ActiveMQ的其他几种文件存储引擎相比这就非常简洁了（应该还有一个db.free文件）

1. db-<Number>.log： KaHaDB存储消息到预定义大小（默认日志文件的大小为32M）的数据记录文件中，文件命名为db-<Number>.log。当数据文件已满时，一个新的文件会随之创建，number数值也会随之递增，它随着消息数量的增多，如每32M一个文件，文件名按照数字进行编号，如db-1.log、db-2.log、db-3.log…..当不再有引用到数据文件中的任何消息时，文件会被删除或归档
2. db.data该文件包含了持久化的BTree索引，索引了消息数据记录中的消息，他是消息的索引文件，本质上是B-Tree(B树)，使用B-Tree作为索引指向db-<Number>.log中的消息
3. db.free 当前db.data文件里哪些页面是空闲的，文件具体内容是指所有空闲页的ID
4. db.redo用来进行消息恢复，如果KaHaDB消息存储在强制退出后启动，用于恢复BTree索引
5. lock文件锁：表示当前获得KaHaDB读写权限的broker（有点像mysql的悲观锁，进行读操作的时候共享，写操作的时候独占）

##### JDBC消息存储（工作中常用）

消息会真实的进入我们的mysql或者Oracle数据库，后面会真刀实枪地上干货！

##### Level DB消息存储（了解）

这种文件系统是从ActiveMQ5.8以后引进地，和KaHaDB非常相似，也是基于文件系统地本地数据库储存形式，但是提供比KaHaDB更快地持久性

他不使用BTree来实现索引预写日志，而是使用基于LevelDB的索引

默认配置如下：

<persistenceAdapter>

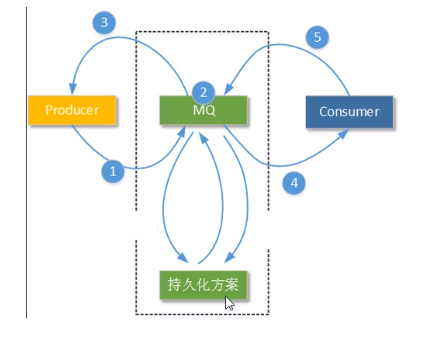
<levelDB directory="activemq-data"/>

</persistenceAdapter>

##### JDBC Message Store with ActiveMQ Journal（工作中常用）

#### JDBC消息存储（重要！！！）

##### MQ+Mysql



##### 添加mysql数据库的驱动包到lib文件夹

把mysql-connector-java-5.1.38.jar拷贝到/myactiveMQ/apache-activemq-5.15.9/lib

1. Jdbc PersistenceAdapter配置

<persistenceAdapter>

<kahaDB directory="${activemq.data}/kahadb"/>

</persistenceAdapter>

替换成：

<persistenceAdapter>

<jdbcPersistenceAdapter dataSource="#mysql-ds"/>

</persistenceAdapter>

dataSource指定将要引用的持久化数据库的bean名称，createTableOnstartup：是否在启动的时候创建数据库表，不写默认值是true,这样每次启动都会去创建数据库表，一般是第一次启动的时候设置为true之后改成false

##### 数据库连接池配置

<bean id="mysql-ds" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://192.168.18.106:3306/activemq?relaxAutoCommit=true"/>

<property name="username" value="root"/>

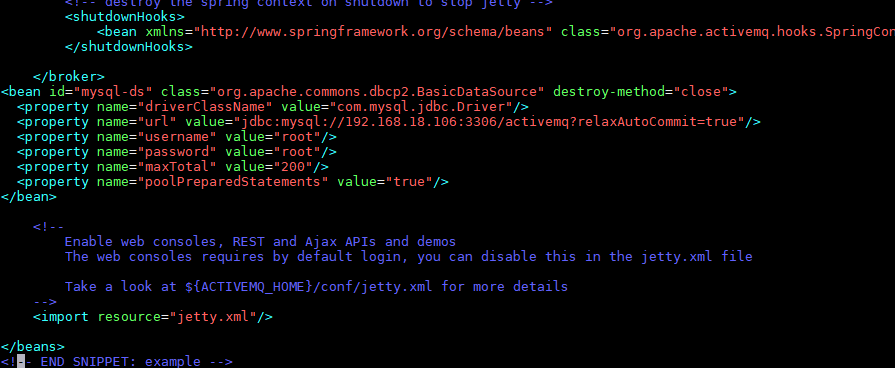
<property name="password" value="root"/>

<property name="maxTotal" value="200"/>

<property name="poolPreparedStatements" value="true"/>

</bean>

注意，数据库链接的配置应该在<broker>标签后，在<import>标签之前



##### 建仓sql和建表说明

1. 在SQLyog中使用创建数据库语句：create databse activemq
2. 三张表的说明

ACTIVENQ\_MSGS**消息表**，queue和topic都存在里面，结构如下

ID：自增的数据库主键

CONTAINER：消息的destination

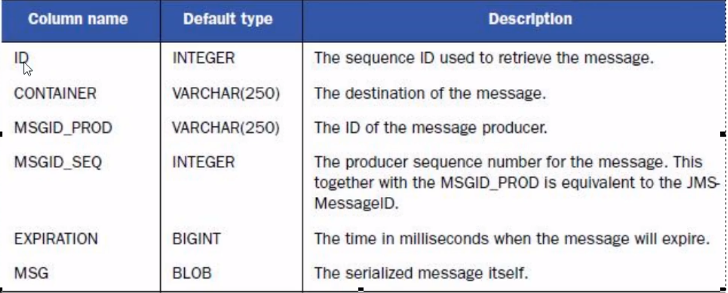
MSGID\_PROD：消息发送者的主键

MSGID\_SEQ：是发送消息的顺序，MSGID\_PROD+ MSGID\_SEQ可以组成JMS的MessageID

EXPIRATION：消息的过期时间，存储的是从1970-01-01到现在的毫秒数

MSG：消息本体的java序列化对象的二进制数据

PRIORITY：优先级，从0-9，数值越大优先级越高



ACTIVE\_ACKS：**用于储存订阅关系**，如果是持久化Topic，订阅者和服务器的订阅关系在这个表保存，数据库字段如下：

CONTAINER：消息的destination

SUB\_DEST：如果使用static集群，这个字段会有集群其他系统的信息

CLIENT\_ID：每个订阅者都必须有一个唯一的客户端ID用以区分

SUB\_NAME：订阅者名称

SELECTOR：选择器，可以选择只消费满足条件的消息。条件可以用自定义属性实现，可支持多属性AND和OR操作

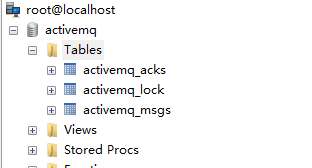
LAST\_ASKED\_ID：记录消费过的消息的ID



ACTIVE\_LOCK：在集群环境中才有用，只有一个Broker可以获得消息，称为Master Broker,其他只能作为备份等待Master Broker不可用，才能称为下一个Master Broker，这个表用来记录哪个Broker是当前的Master Broker



新建数据库ok+xml中的bean配置ok+代码运行ok就能自动生成3张表



**如果无法启动activemq，请检查西面两个问题（巨坑！！！）：**

1. **需要关闭linux和windows的防火墙，需要保证linux服务器（我配置的Devops虚拟机）和Window能互相ping通**
2. **给Windows上的mysql数据库添加Linux用户的权限**

**GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'%' IDENTIFIED BY 'root' WITH GRANT OPTION;**

**FLUSH PRIVILEGES;**

**3.** **建议用bin/activemq console方式启动，可以及时查看错误信息。**

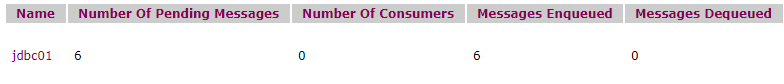
##### 代码运行验证

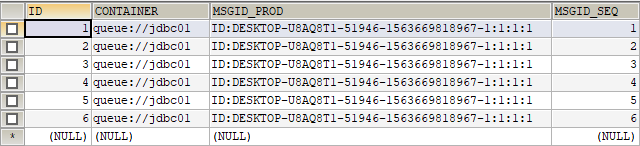
一定要开启持久化：messageProducer.setDeliveryMode(DeliveryMode.PERSISTENT);

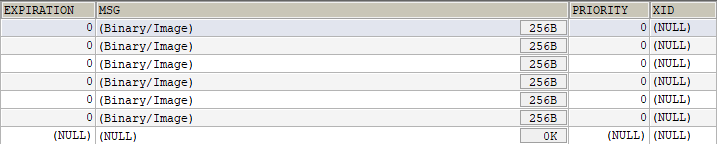
###### 队列：

生产:

public class JMSProduce {  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *Queue\_NAME* = "jdbc01";  
  
 public static void main(String[] args) throws JMSException {  
 //1.创建连接工厂，按照给定的url地址，采用默认的用户名密码  
 ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(*DEFAULT\_BROKER\_BIND\_URL*);  
 //2.通过连接工厂，获得连接Connection并启动访问  
 Connection connection = factory.createConnection();  
 connection.start();  
 //3.创建会话Session  
 //两个参数，一个叫事务，一个叫签收  
 Session session = connection.createSession(false, Session.*AUTO\_ACKNOWLEDGE*);  
 //4.创建目的地  
 Queue queue = session.createQueue(*Queue\_NAME*);  
 //5.创建消息的生产者  
 MessageProducer producer = session.createProducer(queue);  
 //持久化  
 producer.setDeliveryMode(DeliveryMode.*PERSISTENT*);  
 //6.使用消息生产者MessageProducer发送消息到MQ队列里面  
 for (int i = 1; i <=6 ; i++) {  
 //7.创建消息  
 TextMessage textMessage = session.createTextMessage("jdbc msg---->" + i);  
 //8.通过MessageProducer对象来发送消息  
 producer.send(textMessage);  
 }  
 //9.关闭资源  
 producer.close();  
 session.close();  
 connection.close();  
 System.*out*.println("消息发布到MQ完成");  
 }  
}







消费:

消费者接收消息jdbc msg---->1

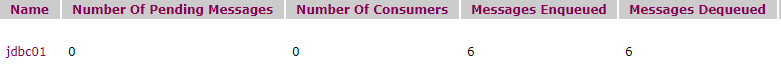
消费者接收消息jdbc msg---->2

消费者接收消息jdbc msg---->3

消费者接收消息jdbc msg---->4

消费者接收消息jdbc msg---->5

消费者接收消息jdbc msg---->6



数据库的消息全部消失



在点对点的类型中，当DeliveryMode设置为Non\_ PERSISTENT时，消息将被保存在内存中；当DeliveryMode设置为PERSISTENT时，消息将会被保存在broker相应的文件或者中

并且在点对点的类型中，消息一旦被消费，就从broker中删除了

###### Topic：

生产：

public class TopicProduce\_Persistent {  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *Topic\_NAME* = "topic-jdbc-persistent";  
 public static void main(String[] args) throws JMSException {  
 //创建ConnectionFactory对象  
 ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(*DEFAULT\_BROKER\_BIND\_URL*);  
 //获取连接  
 Connection connection = factory.createConnection();  
 //获取Session对象  
 Session session = connection.createSession(false,Session.*AUTO\_ACKNOWLEDGE*);  
 //创建目的地  
 Topic topic = session.createTopic(*Topic\_NAME*);  
 //创建生产者对象  
 MessageProducer messageProducer = session.createProducer(topic);  
 //设置消息的持久化  
 messageProducer.setDeliveryMode(DeliveryMode.*PERSISTENT*);  
 //开启连接  
 connection.start();  
 for (int i = 0; i < 5; i++) {  
 //创建消息  
 TextMessage textMessage = session.createTextMessage("topic生产者发送消息-->" + i);  
 //使用生产者对象把消息发送到消息队列中  
 messageProducer.send(textMessage);  
 }  
 //关闭资源  
 messageProducer.close();  
 session.close();  
 connection.close();  
 System.*out*.println("Topic消息发布到MQ");  
 }  
}

消费：

public class TopicConsumer\_Persistent {  
 public static final String *DEFAULT\_BROKER\_BIND\_URL* = "tcp://192.168.18.105:61616";  
 public static final String *Topic\_NAME* = "topic-jdbc-persistent";  
 public static void main(String[] args) throws JMSException {  
 //1.创建连接工厂，使用默认的用户名密码  
 ActiveMQConnectionFactory activeMQConnectionFactory = new ActiveMQConnectionFactory(*DEFAULT\_BROKER\_BIND\_URL*);  
 //2.获得连接并启动  
 Connection connection = activeMQConnectionFactory.createConnection();  
 //3.需要在连接上设置消费者id,表明消费方有一个叫atguigu的订阅者  
 connection.setClientID("atguigu01");  
 //4.创建会话,两个参数：1.事务是否提交 2.默认的签收方式  
 Session session = connection.createSession(false,Session.*AUTO\_ACKNOWLEDGE*);  
 //5.创建主题  
 Topic topic = session.createTopic(*Topic\_NAME*);  
 //6.要创建TopicSubscriber来订阅  
 TopicSubscriber topicSubscriber = session.createDurableSubscriber(topic, "mq-jdbc");  
 //7.一定要先运行一次，等于向消息服务中间件注册这个消费者，然后再运行客户端发送消息，  
 //这个时候，无论消费者是否在线，都会接受到；不在线的话，下次连接上的时候，会把没有接收到的消息接受下来  
 connection.start();  
 //8.接收消息  
 Message message = topicSubscriber.receive();  
 while (null != message && message instanceof TextMessage){  
 TextMessage textMessage = (TextMessage) message;  
 System.*out*.println("消费者收到带持久化的Topic信息：" + textMessage.getText());  
 message = topicSubscriber.receive(1000L);  
 }  
 //关闭资源  
 session.close();  
 connection.close();  
 }  
}



SUB\_NAME:订阅者名称；Client\_ID：每个订阅者都拥有的唯一的客户端ID





先启动消费者的订阅，因此消费者一直处于在线监听状态；

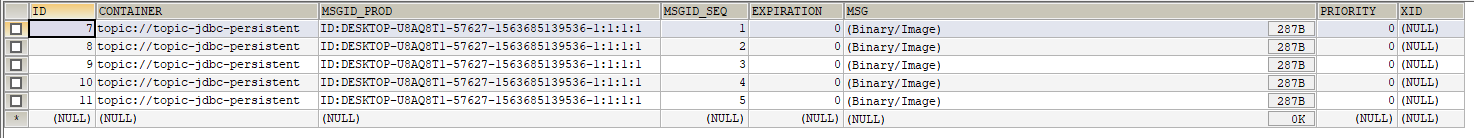
再启动生产者：



消费者接受完消息也就离线了：



而MQ中消息将会持久化在数据库中：



这也是保证了数据不会丢失的一大重要手段，形成了MQ的高可用性

##### 数据库情况

一旦运行了生产code：

点到点：active\_msg表的变化情况（见上一节截图）

发布订阅：

Mysql：

Queue：在运行消费看看表active\_msg表的变化情况（见上一节截图）

Topic： 先启动消费者订阅再运行生产，看看activemq\_acks表

##### 小总结

如果是Queue：

在没有消费者消费的情况下会将消息保存到activemq\_msgs表中，只要有任意一个消费者已经消费过了，消费之后这些消息将会立即被删除。

如果是Topic：

一般是先启动消费订阅然后在生产的情况下会将消息保存到activemq\_msgs中，而activemq\_acks中记录了订阅消息等信息

而当你把离线的订阅者详情删除之后，数据库这边activemq\_acks表中的订阅信息也将会被同步删除

##### 开发有坑

* 数据库驱动jar包和对应的数据库连接池jar包放在lib目录下
* creatTablesOnStartup属性，在第一次启动activemq时，设置为true(默认不写也是true)，ActiveMq服务节点会自动创建所需要的数据表。启动后可以去掉这个属性，或者跟改为false;

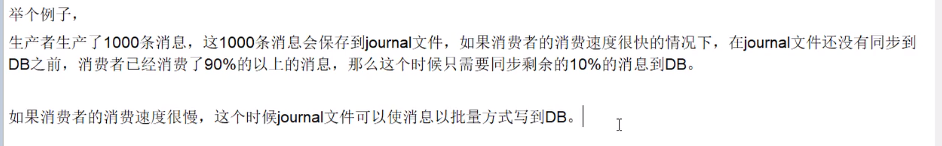
#### JDBC Message Store with AactiveMq Journal

##### 是什么？

这种方式克服了jdbc store的不足，jdbc每次消息过来，都需要去写库和读库。

AactiveMq Journal，使用高速缓存写入技术，大大提高了性能。

当消费者消费的速度能够及时跟上生产者消息的生产速度时，journal文件能够大大减少需要写入到DB中的消息



##### 配置：

在activemq.xml中，把之前配置的persistenceAdapter改成persistenceFactory



##### 重新配置后激活

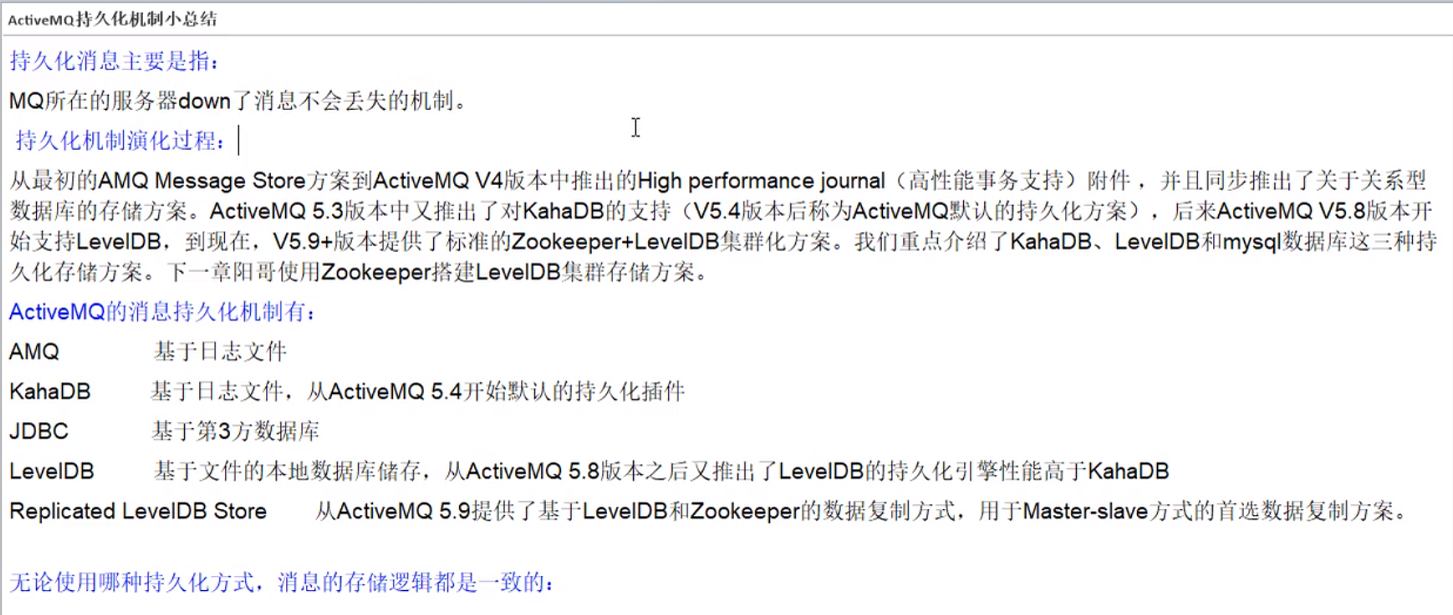
进入到bin目录后先，./activemq stop；再 ./activemq start

##### 运行

Queue ：

当生产完消息之后，去activemq\_msgs中去查询数据的时候，没有任何信息，因为消息都被存放（优先堆积）在journal文件中（journal文件有点像数据库缓冲层，有消息先写入journal文件），这样减少频繁地往数据库中写入数据，而消费者可以从journal文件中正常地读取消息；而过了7，8分钟消息才会缓缓地把数据同步到mysql的表中；在消费者消费速度很快的情况下，如果在消息被同步到数据库之前，消费者就已经消费了journal文件中90%的消息，这样只会有剩余10%的消息同步到数据库中。（journal文件被称之为高速缓存日志文件）

#### ActiveMQ持久化机制小总结





### 13.ActiveMQ多节点集群

#### 面试题：

引入消息队列之后该如何保证其高可用性

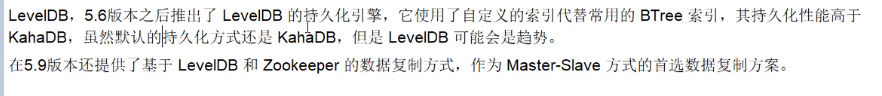
#### 是什么？

基于zookeeper和leveldb搭建ActiveMQ集群。集群仅提供主备方式的高可用集群功能，避免单点故障。

#### Zookeeper+replicated-leveldb-store的主从集群

##### 三种集群方式对比：

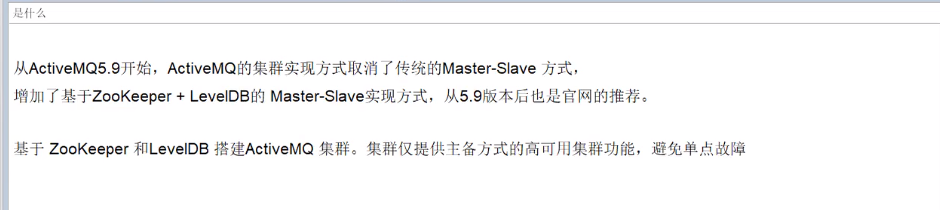
1. 基于sharedFileSystem共享文件系统
2. 基于jdbc
3. **基于可复制的leveldb(very fast,但需要zookeeper服务器)**



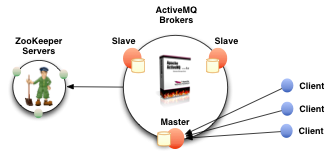
##### 本次案列使用Zookeeper+replicated-leveldb-store：

###### 是什么？

<http://activemq.apache.org/replicated-leveldb-store.html>



###### 官网集群原理图：





###### 部署规划和步骤

###### 集群可用性测试

### 14.高级特性和大厂常考重点

# 微服务架构

# 并发编程

## JUC(java.util.current)尚硅谷李贺飞

### Volatile关键字与内存可见性问题

在程序执行过程中，jvm会为每个执行任务的线程都开辟出一个缓存空间用来提高效率。

供多线程访问的成员变量通常都是在主存中（可以理解为堆内存）；

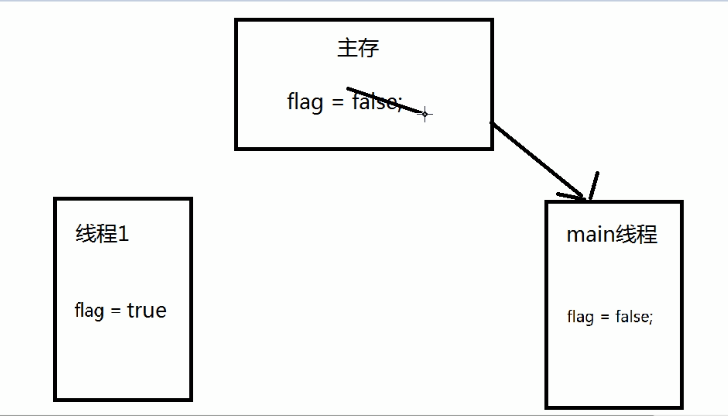
现在有这么一个场景：

有两个线程：一个是新创建的线程1，另一个是main线程；

线程1要去访问主存中的共享数据，并修改成员变量的值；在这个过程中，线程1会先把主存中的共享数据读到线程单独开辟的缓存中来，然后再修改共享数据的值，然后再把改完的值写到主存中去；

此时，在线程1把共享数据写入到主存中去之前，main线程也会先去把主存中的共享数据读取到线程的缓存中（注意：此时的共享数据还是线程1没修改之前的）。在main线程中使用循环语句while(true)来不停的去判断共享数据有没有变化，但是while(true)调用的其实是系统比较底层的代码，它的执行效率非常的高，以至于main线程都没有机会从主存中再次读取一次数据。于是会对之前读取到缓存内部的共享数据进行判断，如果没有变化就一直循环。。。于是就一直处于执行状态

我们也可以把volatile关键字理解成线程1和main线程的读操作和写操作全都在主存中进行。



**代码：**

*/\*\*  
 \* 1.Volatile关键字  
 \*/*public class TestVolatile {  
 public static void main(String[] args) {  
 ThreadDemo td = new ThreadDemo();  
 new Thread(td).start();  
 while (true){  
 if(td.isFlag()){  
 System.*out*.println("-----------------------------");  
 break;  
 }  
 }  
 }  
}  
class ThreadDemo implements Runnable{  
 private boolean flag =false;  
 public boolean isFlag() {  
 return flag;  
 }  
 public void setFlag(boolean flag) {  
 this.flag = flag;  
 }  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(200);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 flag = true;  
 System.*out*.println("flag=" + isFlag());  
 }  
}

内存可见性错误：当读操作和写操作在不同的线程中执行时，我们无法确保执行读操作的线程能适时的看到其他线程写入的值

我们的确可以使用同步锁来保证线程会定时地刷新缓存，去主存中读取数据；

while (true){  
 synchronized (td) {  
 if(td.isFlag()){  
 System.*out*.println("-----------------------------");  
 break;  
 }  
 }  
}

但是这种加锁地方式效率极低，因此我们可以使用volatile关键字，当多个线程对共享数进行操作时，可以保证内存中的共享数据是可见的（我们可以理解为，这个线程每次都会去主存中读取最新的共享数据到该线程的缓存中），效率比使用锁效率高很多

private volatile boolean flag =false;

volatile相较于synchronized是一种较轻量级的同步策略

1. volatile不具备”互斥性”（什么叫互斥性？比如当一个线程抢到了这把锁之后其他的线程进不了synchronized的代码块中，synchronized也被称为互斥锁）
2. 不能保证变量的原子性

### 原子变量与CAS算法

关于volatile关键字的不能保证变量的原子性问题,我们通过下面这个案列来体会和理解：

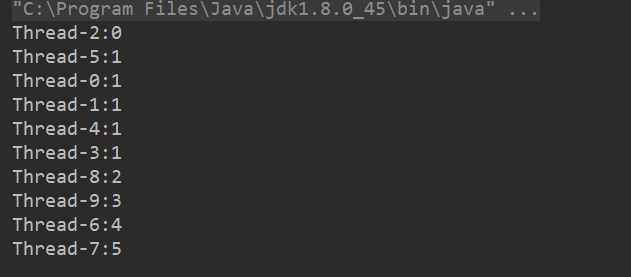
在java基础阶段我们都知道 i = i++ 经历3个阶段 读，改，写。

*\* 过程分析：1.先把i赋值给temp临时变量 int temp = i;  
\* 2.i自增1 i= i+1；  
\* 3.再把temp中的值赋值给i int i = temp*

我们可以把这3个阶段看成一个原子，应该是不可分割的；但是若在线程1赋值之后准备改但是还没改共享数据的时候，线程2插入进来，读取了共享数据的值（两个线程读取到了相同值得共享数据），这样就会导致多线程并发情况下，i++会出现重复值的情况

public class TestAtomicDemo {  
 public static void main(String[] args) {  
 AtomicDemo ad = new AtomicDemo();  
 for (int i = 0; i < 10; i++) {  
 new Thread(ad).start();  
 }  
 }  
}  
class AtomicDemo implements Runnable{  
 private int serialNum = 0;  
 public int getSerialNum() {  
 return serialNum++;  
 }  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(Thread.*currentThread*().getName()+":"+getSerialNum());  
 }  
}

在多线程访问情况下，不同的线程会得到相同的共享数据



当我们使用volatile关键字的时候，我们或许能够解决内存可见性的问题，但是依然无法保证变量的原子性

如何解决？

在jdk1.5之后，java.util.current.atmoci包下提供了常用的原子变量，原子变量是如何解决内存可见性问题，以及如何保证变量的原子性的呢？

1. java.util.current.atmoic中的变量都是用volatile关键字进行修饰的

private volatile int value;

1. CAS（Compare-And-Swap）算法保证数据的原子性

CAS算法是硬件对于并发操作共享数据的支持

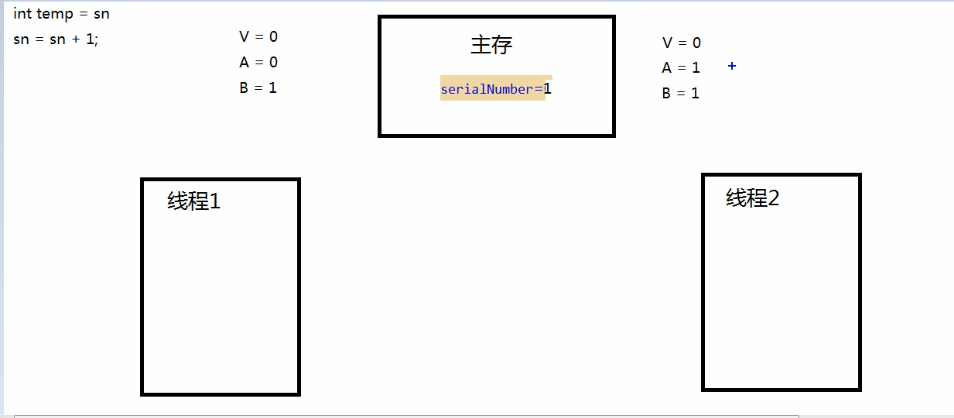
CAS包含了3个操作数：

1．内存值V 2.预估值A 3.跟新值B

当且仅当V==A,V=B,否则将不做任何操作

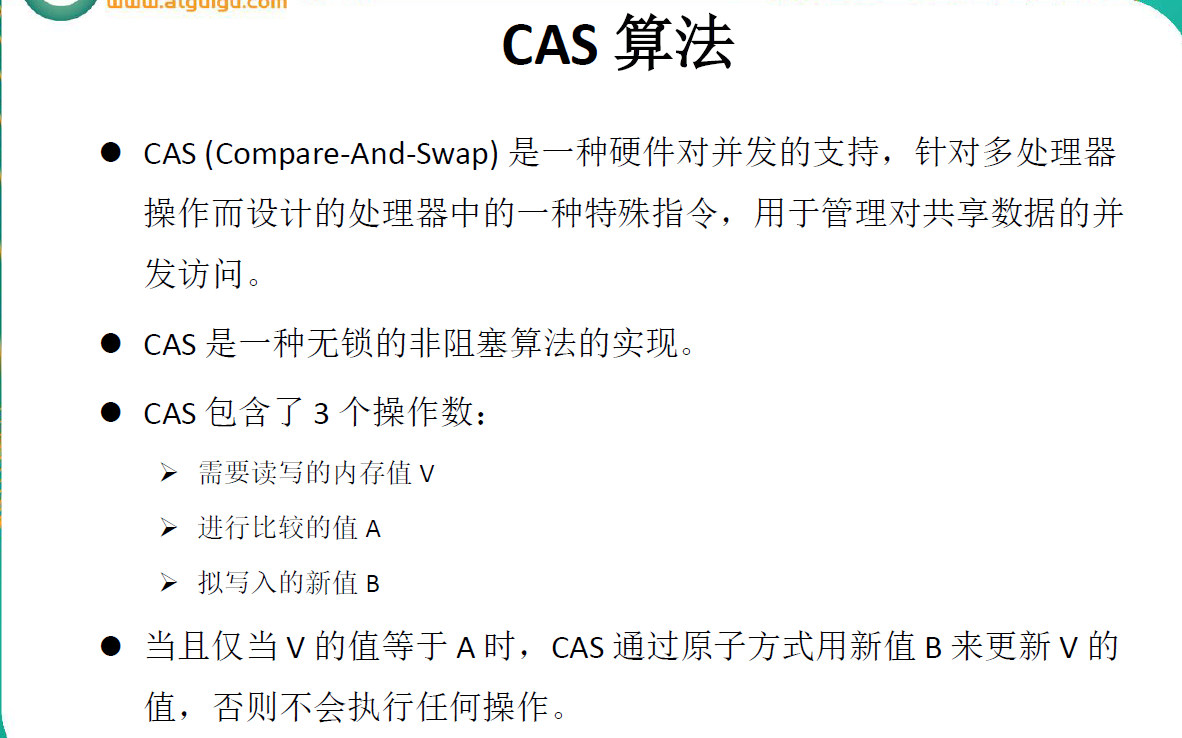
CAS如何保证数据的原子性？

1）int temp = serialNumber; 2）serialNumber = serialNumber+1;



**线程1先从主存中读取内存值V=0；它在要替换之前会再次去读取一下主存中的旧值，A也是0；B经过运算后是1。B想要跟新主存中的共享数据值，必须先判断A和V是否相等，若相等，则可以跟新主存中的共享数据；若不等则不会做任何的操作。**

**若main线程在线程1还未跟新数据的时候访问主存中的共享数据（读取内存中的值V=0），但是当它再次去读取主存中的共享数据时却发现，主存中的共享数据已经被线程1改为serialNumber=1；于是main线程中的内存值V和预估值A不相等，此时便不会发生任何操作**



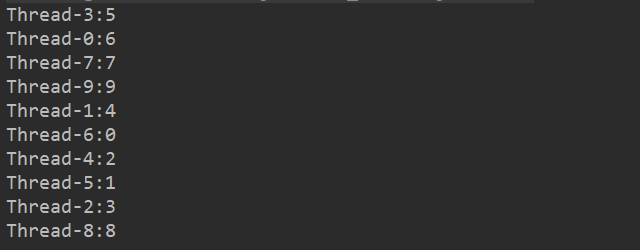
如果多个线程并发地对主存中地数据进行修改的时候，有且只有一个线程会成功，其他线程都将会失败；

CAS算法比我们原先解决多线程并发问题的同步锁方法效率要高，因为我们知道，使用了同步锁之后，当一个线程进入了同步代码块之后其他线程就会处于阻塞状态，必须等该线程释放了锁之后，才能进入到同步代码块中执行线程体，而一旦执行完，又进入了阻塞状态；而CAS算法一旦这次V和A的值不相等，导致跟新主存中的共享数据失败了之后，它不会阻塞，不会放弃CPU给它的执行权，它可以再次从读取V的值，A的值，判断A和V是否相等，跟新B的值到主存中的共享数据中。因此CAS算法比同步锁的效率要高很多

使用了java.util.atmoic包下的对象作为共享数据（实现CAS算法）

public class TestAtomicDemo {  
 public static void main(String[] args) {  
 AtomicDemo ad = new AtomicDemo();  
 for (int i = 0; i < 10; i++) {  
 new Thread(ad).start();  
 }  
 }  
}  
class AtomicDemo implements Runnable{  
 private AtomicInteger serialNum = new AtomicInteger();  
 public int getSerialNum() {  
 return serialNum.getAndIncrement();  
 }  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(Thread.*currentThread*().getName()+":"+getSerialNum());  
 }  
}

不同线程拿到的共享数据不会重复，避免了线程安全问题，并且比同步锁的实现效率要高



### ConcurrentHashMap锁分段机制

Java.util.concurrent包下提供了多种并发容器类来改进同步容器的性能

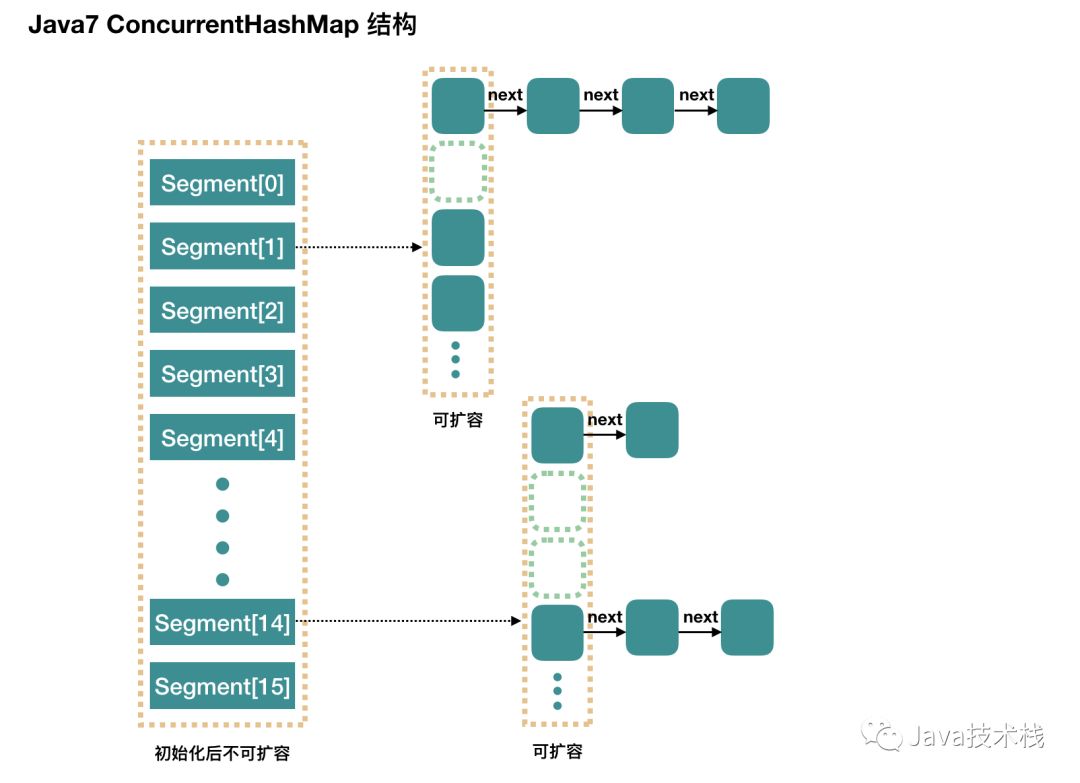
**1.HashMap和Hashtable的区别：**HashMap是线程不安全的，HashTable是线程安全的；

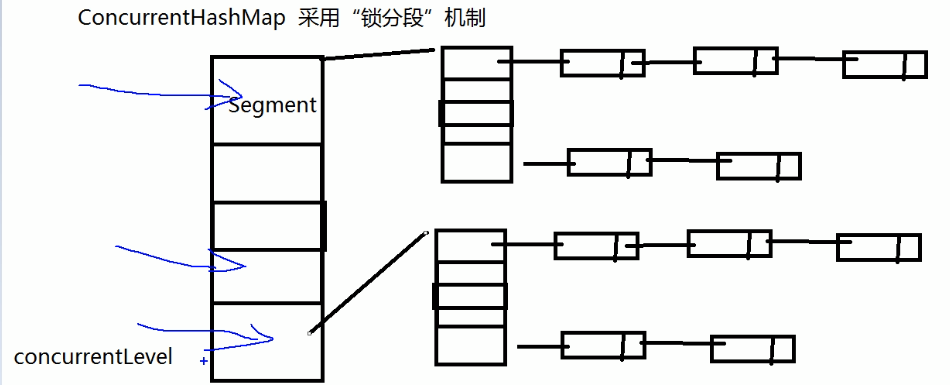
**2.为什么Hashtable效率低？**因为Hashtable使用的独占锁，锁的是hashtable整张表；当多线程并发时，一个线程执行的时候，其他线程就被所在了hash表外面，无法对集合进行操作；从而导致，同一时间只有一个线程能操作hashtable，把并行改成了串行，执行效率低！

**3.ConcurrentHashMap采用了锁分段机制**

通过将整个Map分成16个Segment，ConcurrentHashMap就是一个Segment数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment。那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术

而在jdk1.8中全面实现了CAS算法





### CountDownLatch

CountDownLatch一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待

在完成某些运算之时，只有当其他的线程的运算全部完成时，当前运算才会执行。

使用场景：在main线程中计算其他分线程执行完成所需要的时间；

1）在创建CountDownLatchde 对象的时候传一个线程数进去，该线程数记录了出主线程外其他分线程的数量；

2）在每个线程执行完run方法中的线程体后需要latch.countDown(),让主线程需要等待的分线程数量-1，知道latch=0，此时便能执行主线程中的计时操作。

*/\*\*  
 \* 闭锁：在完成某些运算时，只有当其他线程的运算全部完成，  
 \* 当前运算才会继续执行  
 \*  
 \*/*public class TestCountDownLatch {  
 public static void main(String[] args) {  
 final CountDownLatch latch = new CountDownLatch(5);  
 LatchDemo ld = new LatchDemo(latch);  
 long start = System.*currentTimeMillis*();  
 for (int i = 0; i < 5; i++) {  
 new Thread(ld).start();  
 }  
 try {  
 latch.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 //需要等待10个分线程全部执行完了，主线程才去执行，这里需要使用闭锁  
 long end = System.*currentTimeMillis*();  
 System.*out*.println("耗费时间为："+ (end-start));  
 }  
}  
  
class LatchDemo implements Runnable{  
  
 private CountDownLatch latch;  
  
 public LatchDemo(CountDownLatch latch) {  
 this.latch = latch;  
 }  
  
 @Override  
 public void run() {  
 synchronized (this){  
 try {  
 for (int i = 0; i < 50000; i++) {  
 if (i%2 == 0){  
 System.*out*.println(i);  
 }  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 latch.countDown();  
 }  
 }  
 }  
}

### 创建线程的第三种方式：实现Callable接口

相较于实现Runnable接口，实现Callable接口的这种创建线程的方式的优点：

1. 方法可以有返回值
2. 并且可以抛出异常。
3. 帮我们创建另外一个线程执行不同的任务，而主线程这边也可以做一些事，并且在

分线程结束之后可以接收到分线程的返回结果，与主线程进行汇总，提高了效率

所以我们在创建线程的方式上也会发生改变：

* 还是先创建一个ThreadDemo类去实现Callable接口，注意Callable接口是有泛型的，泛型类型就是重写run方法的返回值类型
* 重写run方法，里面就是线程的执行体
* 在main线程中创建ThreadDemo类的实例对象
* 把Thread Demo的实例对象作为参数传到FutureTask的构造器中,创建出FutureTask得实例对象result
* new Thread(result).start开启线程
* 需要在main线程中接收运算后的结果

*/\*\*  
 \* 创建执行线程的方式三：实现Callable接口  
 \* 执行Callable方式，需要FutureTask实现类的支持，用于接收运算结果  
 \* FutureTask是Future接口的实现类  
 \*/*public class TestCallable {  
 public static void main(String[] args) {  
 ThreadDemo1 td1 = new ThreadDemo1();  
 FutureTask<Integer> result = new FutureTask<>(td1);  
 new Thread(result).start();  
 //接收线程运算后的结果  
 try {  
 Integer sum = result.get();  
 System.*out*.println("总和为："+ sum);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
}  
  
class ThreadDemo1 implements Callable<Integer>{  
 @Override  
 public Integer call() throws Exception {  
 int sum = 0;  
 for (int i = 0; i <= 100; i++) {  
 sum+=i;  
 }  
 return sum;  
 }  
}

注意：接收结果一定是在创建的线程结束之后执行的，FutureTask也可用于闭锁

### 同步锁lock

在jdk1.5之前只能使用同步代码块和同步方法来解决线程安全问题，而在jdk1.5之后JUC包提供了另外一种上锁方式：lock

比起synchronized这种块式的上锁方式，同步锁可以帮助我们更灵活的进行lock和unlock，是一种比同步代码块更加灵活的处理多线程安全问题的一种解决方式

而这种锁需要我们手动自己去释放锁，因此unlock必须放在finnally块中，否则会造成多线程的堵塞问题。

使用lock解决线程安全问题

*/\*\*  
 \* 用于解决多线程安全问题的方式：lock  
 \* synchronized 隐式锁  
 \* lock 显示锁  
 \*/*public class TestLock {  
 public static void main(String[] args) {  
 Ticket t = new Ticket();  
 new Thread(t,"一号窗口").start();  
 new Thread(t,"二号窗口").start();  
 new Thread(t,"三号窗口").start();  
 }  
}  
  
class Ticket implements Runnable{  
 private int ticket = 100;  
 private Lock lock = new ReentrantLock();  
 @Override  
 public void run() {  
 while (true){  
 try {  
 lock.lock();  
 if (ticket>0){  
 try {  
 Thread.*sleep*(300);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(Thread.*currentThread*().getName()+ "完成售票,余票为" + --ticket);  
 }else {  
 break;  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();//释放锁  
 }  
 }  
 }  
}

### 同步锁lock的等待唤醒机制（生产者消费者案列，虚假唤醒）

1.回顾之前使用同步方法实现等待唤醒机制（使用Object中的wait和notify方法）

public class TestProductorAndConsumer {  
 public static void main(String[] args) {  
 Clerk clerk = new Clerk();  
 Productor productor = new Productor(clerk);  
 Consumer consumer = new Consumer(clerk);  
 new Thread(productor).start();  
 new Thread(consumer).start();  
 }  
  
}  
  
//店员  
class Clerk{  
 private int product = 0;  
 //进货  
 public synchronized void get(){  
 while(product>=10){  
 System.*out*.println("产品已满");  
 try {  
 this.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 } System.*out*.println(Thread.*currentThread*().getName()+":"+ ++product);  
 this.notifyAll();  
   
 }  
  
 //卖货  
 public synchronized void sale(){  
 while(product<=0){  
 System.*out*.println("缺货");  
 try {  
 this.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 } System.*out*.println(Thread.*currentThread*().getName()+":"+ --product);  
 this.notifyAll();   
 }  
}  
  
//生产者  
class Productor implements Runnable{  
 private Clerk clerk;  
  
 public Productor(Clerk clerk) {  
 this.clerk = clerk;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 20; i++) {  
 clerk.get();  
 }  
 }  
}  
  
//消费者  
class Consumer implements Runnable{  
 private Clerk clerk;  
  
 public Consumer(Clerk clerk) {  
 this.clerk = clerk;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 20; i++) {  
 clerk.sale();  
 }  
 }  
}

**为了避免虚假唤醒，必须把wait和notify放在循环中使用**

**什么是虚假唤醒？**

**举个例子，我们现在有一个生产者-消费者队列和三个线程。**

**1） 1号线程从队列中获取了一个元素，此时队列变为空。**

**2） 2号线程也想从队列中获取一个元素，但此时队列为空，2号线程便只能进入阻塞(cond.wait())，等待队列非空。**

**3） 这时，3号线程将一个元素入队，并调用cond.notify()唤醒条件变量。**

**4） 处于等待状态的2号线程接收到3号线程的唤醒信号，便准备解除阻塞状态，执行接下来的任务(获取队列中的元素)。**

**5） 然而可能出现这样的情况：当2号线程准备获得队列的锁，去获取队列中的元素时，此时1号线程刚好执行完之前的元素操作，返回再去请求队列中的元素，1号线程便获得队列的锁，检查到队列非空，就获取到了3号线程刚刚入队的元素，然后释放队列锁。**

**6） 等到2号线程获得队列锁，判断发现队列仍为空，1号线程“偷走了”这个元素，所以对于2号线程而言，这次唤醒就是“虚假”的，它需要再次等待队列非空。**

当使用同步锁实现等待唤醒机制：

我们使用Conditions方法的await，signal和signalAll来替换Object类中的wait,notify和notifyAll方法

public class TestProductorAndConsumerForLock {  
  
 public static void main(String[] args) {  
 Clerk1 clerk1 = new Clerk1();  
 Productor1 productor1 = new Productor1(clerk1);  
 Consumer1 consumer1 = new Consumer1(clerk1);  
 new Thread(productor1,"生产者A").start();  
 new Thread(consumer1,"消费者B").start();  
 }  
}  
class Clerk1{  
  
 private Lock lock = new ReentrantLock();  
 private Condition condition = lock.newCondition();  
 private int product = 0;  
 //进货  
 public void get(){  
 lock.lock();  
 try {  
 while (product>=5){  
 System.*out*.println("产品已满");  
 try {  
 condition.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println(Thread.*currentThread*().getName()+":"+ ++product);  
 condition.signalAll();  
 } finally {  
 lock.unlock();  
 }  
  
 }  
  
 //卖货  
 public void sale(){  
 lock.lock();  
 try {  
 while(product<=0){  
 System.*out*.println("缺货");  
 try {  
 condition.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println(Thread.*currentThread*().getName()+":"+ --product);  
 condition.signalAll();  
 } finally {  
 lock.unlock();  
 }  
  
 }  
}  
//消费者  
class Consumer1 implements Runnable{  
 private Clerk1 clerk;  
 public Consumer1(Clerk1 clerk) {  
 this.clerk = clerk;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 20; i++) {  
 clerk.sale();  
 }  
 }  
}  
  
//生产者  
class Productor1 implements Runnable{  
 private Clerk1 clerk;  
  
 public Productor1(Clerk1 clerk) {  
 this.clerk = clerk;  
 }  
 @Override  
 public void run() {  
 for (int i = 0; i < 20; i++) {  
 clerk.get();  
 }  
 }  
}

### 线程按序交替

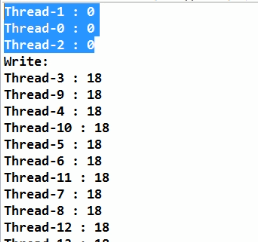
面试题引入：

*/\*\*  
 \* 面试题引入：  
 编写一个程序，开启3个线程，这3个线程的ID分别为A，B，C,每个线程将自己的ＩＤ在屏幕上打印10遍，  
 要求输出的结果必须按顺序显示，如ABCABCABC…一次递归  
 \*  
 \*/*public class TestABCAlternate {  
 public static void main(String[] args) {  
 AlternateDemo ad = new AlternateDemo();  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for (int i = 1; i <=10; i++) {  
 ad.loopA();  
 }  
 }  
 },"A").start();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for (int i = 1; i <=10; i++) {  
 ad.loopB();  
 }  
 }  
 },"B").start();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for (int i = 1; i <= 10; i++) {  
 ad.loopC(i);  
 }  
 }  
 },"C").start();  
 }  
  
}  
  
class AlternateDemo{  
 //由于线程ABC是要求按序显示的，因此需要给每个线程定一个标记位  
 private int number = 1;//当前线程的标记  
 private Lock lock = new ReentrantLock();  
 private Condition condition1 = lock.newCondition();  
 private Condition condition2 = lock.newCondition();  
 private Condition condition3 = lock.newCondition();  
 public void loopA(){  
 lock.lock();  
 try {  
 //1.判断当前标记是否是1  
 if(number != 1){  
 condition1.await();  
 }  
 //2.打印  
 for (int i = 0; i <1; i++) {  
 System.*out*.print(Thread.*currentThread*().getName());  
 }  
 //3.唤醒  
 number = 2;  
 condition2.signal();  
  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
  
 public void loopB(){  
 lock.lock();  
 try {  
 //1.判断当前标记是否是1  
 if(number != 2){  
 condition2.await();  
 }  
 //2.打印  
 for (int i = 0; i < 1; i++) {  
 System.*out*.print(Thread.*currentThread*().getName());  
 }  
 //3.唤醒  
 number = 3;  
 condition3.signal();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
  
 public void loopC(int totalPage){  
 lock.lock();  
 try {  
 //1.判断当前标记是否是1  
 if(number != 3){  
 condition3.await();  
 }  
 //2.打印  
 for (int i = 0; i <1; i++) {  
 System.*out*.print(Thread.*currentThread*().getName()+ totalPage);  
 }  
 //3.唤醒  
 number = 1;  
 condition1.signal();  
  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
}

### ReadWriteLock读写锁

读锁可以让多个多线程并发地持有，而写锁是独占的

*/\*\*  
 \* 写写和读写需要"互斥"  
 \* 读读不需要互斥  
 \*  
 \*/*public class TestReadAndWriteLock {  
 public static void main(String[] args) {  
 ReadWriteLockDemo rwd = new ReadWriteLockDemo();  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 rwd.set((int) (Math.*random*()\*101));  
 }  
 },"写线程").start();  
  
 for (int i = 0; i < 100; i++) {  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 rwd.get();  
 }  
 }).start();  
 }  
 }  
}  
  
class ReadWriteLockDemo{  
 private int number = 0;  
 private ReadWriteLock lock = new ReentrantReadWriteLock();  
 //读  
 public void get(){  
 lock.readLock().lock();  
 try {  
 System.*out*.println(Thread.*currentThread*().getName() +":"+ number);  
 } finally {  
 lock.readLock().unlock();  
 }  
 }  
 //写  
 public void set(int number){  
 lock.writeLock().lock();  
 try {  
 System.*out*.println(Thread.*currentThread*().getName());  
 this.number = number;  
 } finally {  
 lock.writeLock().unlock();  
 }  
 }  
}



### 线程八锁

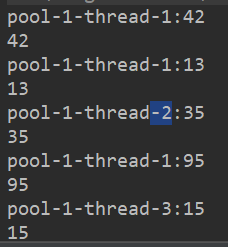
### 线程池

创建线程的第四种方式：使用线程池

*\*\*  
 \* 一、线程池：提供了一个线程队列，队列中保存着所有等待状态的线程  
 \* 避免了频繁的创建和销毁线程所带来的额外开销，提高了响应速度  
 \* 二、线程池的体系结构  
 \* java.util.concurrent.Executor:负责线程的使用和调度的根接口  
 \* |----ExecutorService 子接口，线程池的主要接口  
 \* |----ThreadPoolExecutor 线程池的实现类  
 \* |----ScheduledExecutorService:子接口：负责线程的调度  
 \* |---ScheduledThreadPoolExecutor 继承了ThreadPoolExecutor，实现了ScheduledExecutorService  
 \* 三、工具类：Executors  
 \* newFixedThreadPool:帮我们创建固定大小的线程池  
 \* newCacheThreadPool:可以可以根据需求自动更改线程池的数量  
 \* newSingleTheadExecutor:创建只有单个线程的线程池  
 \* newScheduledThreadPool:帮我们创建固定大小的线程池，并且可以延迟或定时执行任务  
 \*/*

public class TestSchduledThreadPool {  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
 //1.创建线程池  
 ScheduledExecutorService pool = Executors.*newScheduledThreadPool*(5);  
  
 for (int i = 0; i < 5; i++) {  
 //通过线程池帮我们创建固定数量的线程，并调度每个线程延迟3秒执行  
 ScheduledFuture<Integer> result = pool.schedule(new Callable<Integer>() {  
  
 @Override  
 public Integer call() throws Exception {  
 int num = new Random().nextInt(100);  
 System.*out*.println(Thread.*currentThread*().getName() + ":" + num);  
 return num;  
 }  
 }, 3, TimeUnit.*SECONDS*);//设置线程执行的延迟时间  
  
 //接收线程返回的结果值  
 System.*out*.println(result.get());  
 }  
 //关闭线程  
 pool.shutdown();  
  
 }  
}

该线程池中调用不同的线程来执行任务



### ForkjoinPool 分支合并框架\_工作窃取

# 框架

## Springboot

### SpringBoot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

### SpringBoot的优点：

1. **使用嵌入式的Servlet容器，应用无需打成war包**
2. **Starters的自动依赖和版本控制**

若我们需要使用jdbc功能，导入jdbc的启动器；若我们需要redis相关的功能只需要导入redis相关的启动器，并且帮我们自动管理依赖的版本号

1. **大量的自动配置，简化开发，还可以对默认的配置进行修改**（覆盖application.yml中的配置）
2. **无需配置xml，无代码生成，开箱即用**
3. **快速创建独立运行的Spring项目以及与主流框架的集成**

### 一、Spring Boot入门

1. 导入Spring boot相关依赖

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.9.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

1. 编写主程序，用来启动SpringBoot应用

/\*\*

\* @SpringBootApplication 来标注一个主程序类，说明这是一个Spring Boot应用

\*/

@SpringBootApplication

public class HelloWorldMainApplication {

public static void main(String[] args) {

// Spring应用启动起来

SpringApplication.run(HelloWorldMainApplication.class,args);

}

}

1. 编写相关的Controller，Service

@Controller

public class HelloController {

@ResponseBody

@RequestMapping("/hello")

public String hello(){

return "Hello World!";

}

}

1. 简化部署

<!-- 这个插件，可以将应用打包成一个可执行的jar包；-->

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

将这个应用打成jar包，直接使用java -jar的命令进行执行；

#### Hello World深究

##### pom文件

###### 父项目

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.9.RELEASE</version>

</parent>

他的父项目是

<parent>

<groupId>org.springframework.boot</groupId>

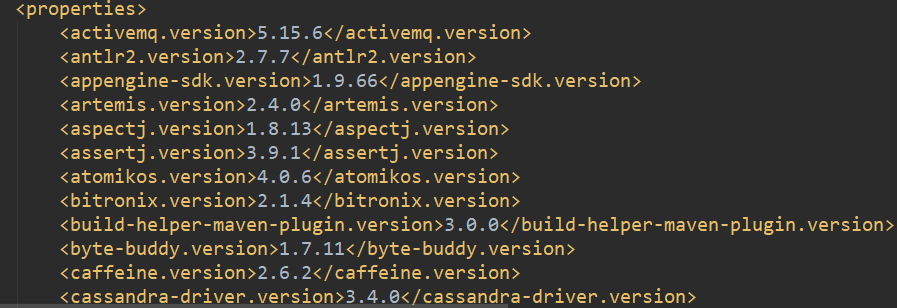
<artifactId>spring-boot-dependencies</artifactId>

<version>1.5.9.RELEASE</version>

<relativePath>../../spring-boot-dependencies</relativePath>

</parent>

他来真正管理Spring Boot应用里面的所有依赖版本；（Spring boot版本仲裁中心）



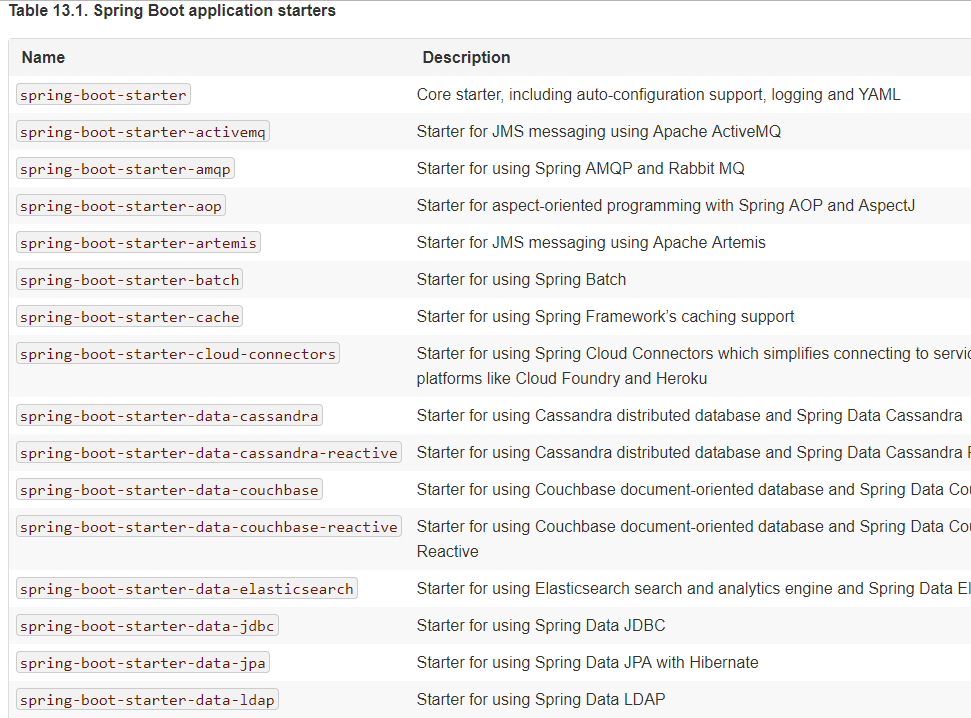
也就是说如果在父项目已经声明版本号，在pom.xml中导入依赖时可以不用写版本号；但是在父项目中没有定义的，在pom.xml中导入依赖时需要指定版本号

###### 启动器

Spring boot场景启动器，spring-boot-starter-web，帮我们导入了web模块正常运行所依赖的组件；

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 <version>2.0.6.RELEASE</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-json</artifactId>  
 <version>2.0.6.RELEASE</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-tomcat</artifactId>  
 <version>2.0.6.RELEASE</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.hibernate.validator</groupId>  
 <artifactId>hibernate-validator</artifactId>  
 <version>6.0.13.Final</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-web</artifactId>  
 <version>5.0.10.RELEASE</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>5.0.10.RELEASE</version>  
 <scope>compile</scope>  
 </dependency>  
</dependencies>

Spring Boot将所有的功能场景都抽取出来，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



##### 主程序类，主入口类

**@SpringBootApplication:在某个类上添加该注解，代表这个类时Spring boot的主配置类，Spring boot就应该运行这个类的main方法来启动Spring boot应用**。

其实是个组合注解

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan(excludeFilters = {

@Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),

@Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })

public @interface SpringBootApplication {

**@SpringBootConfiguration:Spring Boot的配置类；**

**标注在某个类上，表示这是一个Spring Boot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；**

以前我们需要配置的东西，Spring Boot帮我们自动配置；@EnableAutoConfiguration告诉SpringBoot开启自动配置功能；这样自动配置才能生效；

@AutoConfigurationPackage

@Import(EnableAutoConfigurationImportSelector.class)

public @interface EnableAutoConfiguration {

@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；==**

@Import(EnableAutoConfigurationImportSelector.class)；

给容器中导入组件？

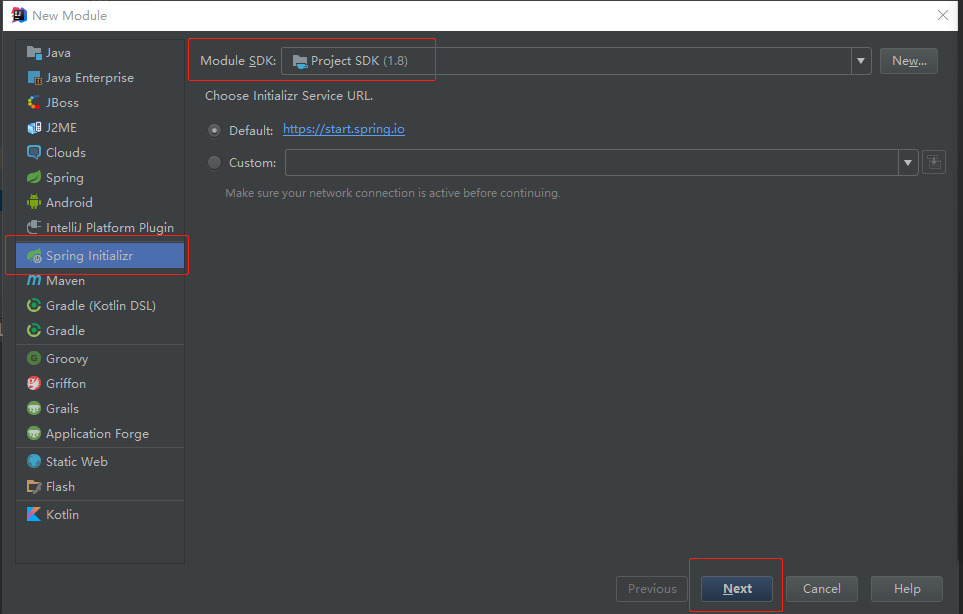
EnableAutoConfigurationImportSelector：导入哪些组件的选择器；

将所有需要导入的组件以全类名的方式返回；这些组件就会被添加到容器中；

会给容器中导入非常多的自动配置类（xxxAutoConfiguration）；就是给容器中导入这个场景需要的所有组件，并配置好这些组件；

##### 快速搭建SpringBoot项目

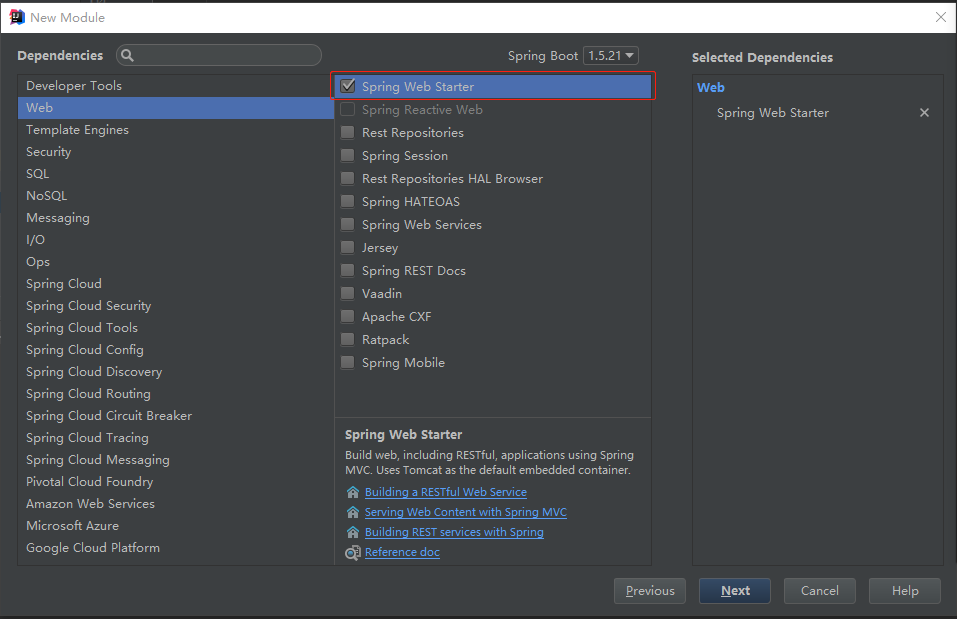
1. 点击Spring Initializr（spring的脚手架）



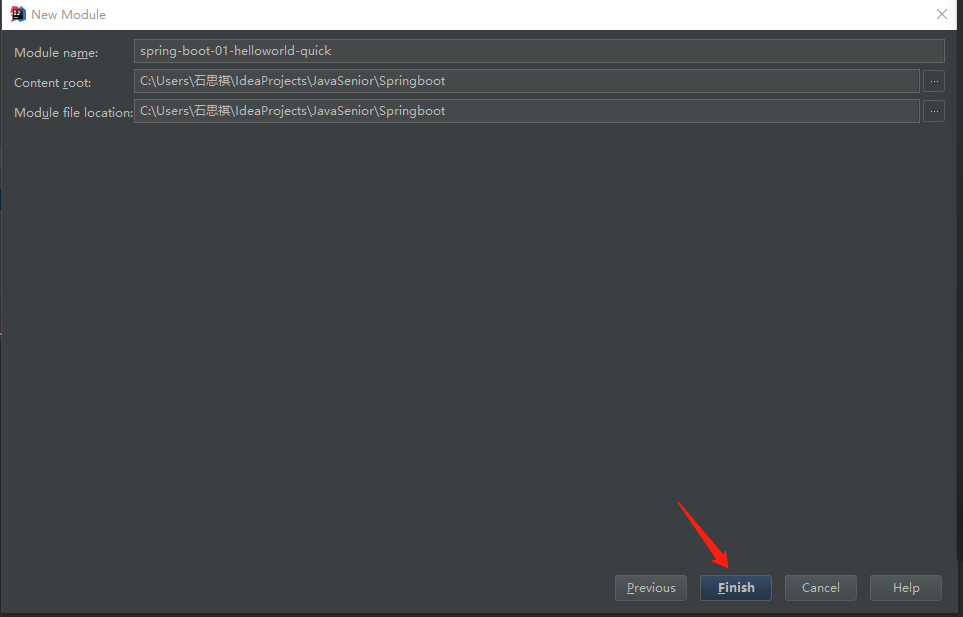
1. 填写包名和项目名



1. 选择你需要导入的启动器名称



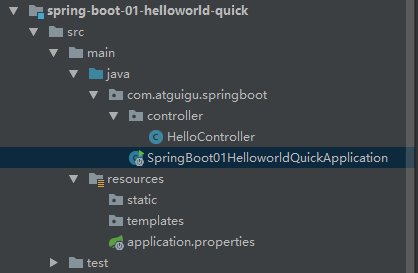
1. 点击finish，向导会联网帮你创建SpringBoot项目



1. Springboot会自动帮我们创建引导类,resources文件夹下的static,templates目录和application.properties文件（也是可以是application.yml）

resources文件夹中目录结构

* static：保存所有的静态资源； js css images；
* templates：保存所有的模板页面；（Spring Boot默认jar包使用嵌入式的Tomcat，默认不支持JSP页面）；可以使用模板引擎（freemarker、thymeleaf）；
* application.properties：Spring Boot应用的配置文件；可以修改一些默认设置；



1. 我们只需要在与引导类相同的目录下，创建Controller包，并在其中创建HelloController

//这个类的所有方法返回的数据直接写给浏览器  
//如果方法的返回值是对象，还能以json数据响应给浏览器  
@RestController  
public class HelloController {  
 @RequestMapping("hello")  
 public String Hello(){  
 return "hello world springboot-quick";  
 }  
}

1. @responseBody注解的作用是将controller的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到response对象的body区，通常用来返回JSON数据或者是XML数据，需要注意的呢，在使用此注解之后不会再走试图处理器，而是直接将数据写入到输入流中，他的效果等同于通过response对象输出指定格式的数据。

### 二、 SpringBoot配置（配置文件、加载顺序、配置原理）

**\*\*\*\*\*在这一章中，所提到的向IOC容器中添加组件，意思就是把对象注入IOC容器中**

#### 1.配置文件

Springboot会使用一个全局的配置文件，名字是固定的，可以是这两种：

* application.properties
* application.yml

配置文件的作用就是要用来修改SpringBoot自动配置的默认值（我们需要去重新配置来覆盖默认的配置）；SpringBoot在底层已经帮我们都配置好；

yaml文件：以数据为中心，比json，xml更适合做配置文件

**server:  
 port:** 8082

#### 2.配置文件值注入：

##### 从yml文件中获取值的两种方式

* 先写java bean类

public class Person {  
 private String lastname;  
 private Integer age;  
 private Boolean boss;  
 private Date birth;  
 private Map<String,Object> maps;  
 private List<Object> lists;  
 private Dog dog;  
 //配置get，set方法和toString方法  
}

public class Dog {  
 private String name;  
 private Integer age;

//同样需要配置get,set和toString  
}

* 配置yaml文件

**person:  
 lastname:** zhangsan  
 **age:** 18  
 **boss: false  
 birth:** 2019/7/14  
 **maps:** {**k1:** v1,**k2:** v2}  
 **lists:** - lisi  
 - zhaoliu  
 **dog:  
 name:** 小狗  
 **age:** 2

* 将配置文件中每一个属性的值，映射到Person javabean这个组件中

*/\*\*  
 \** ***@ConfigurationProperties:*** *告诉Springboot将本类中的属性和配置文件中相关的配置进行绑定  
 prefix = "person" ：告诉Springboot配置文件哪个下面的所有属性进行一一映射  
 只有这个组件是容器中的组件，才能使用容器提供的功能  
 \*  
 \*/*@Component  
@ConfigurationProperties(prefix = "person")//有点批量注入属性的感觉  
public class Person {

* 我们可以在pom.xml文件中导入配置文件处理器，以后编写配置就有提示了

<!--导入配置文件处理器，配置文件进行绑定就会有提示-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-configuration-processor</artifactId>  
 <optional>true</optional>  
</dependency>

* 除了使用@ConfigurationProperties注解,我们还可以使用Spring框架提供的@Value注解来获取配置文件中某个属性的值

@Value("${person.lastname}")

* 在xml配置年代：

<proerty>标签中的name是指定要赋值的属性名，而value则是为了给对象的属性进行赋值；

<bean>标签的作用相当于是把Person类加入IOC容器的管理，作为容器的组件；

<bean id=”person”class=”com.atguigu.bean.Person”>

<property name=”lastName” value=”zhangsan”>

</bean>

这里的value相当于注解@Value,而bean相当于注解@Component

##### @value和@ConfigurationProperties的区别

|  |  |  |
| --- | --- | --- |
|  | @ConfigurationProperties | @Value |
| 功能 | 批量注入配置文件中的属性 | 一个一个指定 |
| 松散绑定（松散语法） | 支持 | 不支持 |
| SpEL（要查资料） | 不支持 | 支持 |
| JSR303数据校验 | 支持 | 不支持 |
| 复杂类型封装 | 支持 | 不支持 |

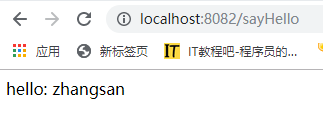
**如果说，我们只是在某个业务逻辑中需要获取一下配置文件中的某项值，使用@Value；**

**如果说，我们专门编写了一个javaBean来和配置文件进行映射，我们就直接使用@ConfigurationProperties；**

* 测试是否配置文件中的值能成功赋值到属性中：

@Value("${person.lastname}")  
private String name;

@RequestMapping("sayHello")  
public String sayHello(){  
 return "hello: " + name;  
}



##### @PropertySource注解

当你想把对象的相关信息，区别于其他配置信息。可以配置在自己定义的properties文件中（比如案列中定义的person.properties文件）；

然后使用@PropertySource注解来加载指定的配置文件：

@PropertySource(value = {"classpath:person.properties"})  
@Component  
@ConfigurationProperties(prefix = "person")  
public class Person {  
 private String lastname;  
 private Integer age;  
 private Boolean boss;  
 private Date birth;  
 private Map<String,Object> maps;  
 private List<Object> lists;  
 private Dog dog;

Person.properties

person.lastname=lisi  
person.age=12  
person.boss=false  
person.birth=2019/7/14  
person.maps.k1=v1  
person.maps.k2=14  
person.lists=a,b,c  
person.dog.name=wangwang  
person.dog.age=13

并且必须保证application.properties或者application.yml中的person属性的配置全部删除！！切记！！因为如果不删除原先的配置，@ConfigurationProperties会读取application.properties配置文件中的person属性。所以要么不加@ConfigurationProperties注解，要么把application.properties配置文件中的person属性删掉

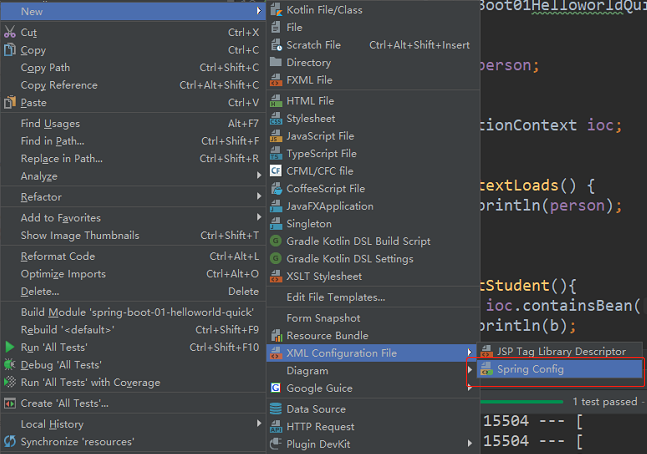
##### @ImportResource

导入Spring的配置文件，让配置文件中的内容生效

SpringBoot中是没有Spring的配置文件的，就算我们自己在resources目录下创建了Spring的配置文件，比如：applicationContext.xml来配置bean的信息，Springboot也不能自动识别。想让Spring的配置文件生效，加载进来，需要使用@importResource注解，该注解需要放在某个配置类上，才能生效

案列：

1. 创建Spring配置文件application-student.xml



编写javabean实体类

public class Student {}

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  
 <bean id="student" class="com.atguigu.springboot.bean.Student">  
 </bean>  
</beans>

1. 测试此时ioc容器中是否有Student这个bean

@Autowired  
ApplicationContext ioc;

@Test  
public void testStudent(){  
 boolean b = ioc.containsBean("student");  
 System.*out*.println(b);  
}

false

1. 此时，只需要在主引导类上加上@importResource注解

@ImportResource(locations = {"classpath:applicationContext-student.xml"})  
@SpringBootApplication  
public class SpringBoot01HelloworldQuickApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringBoot01HelloworldQuickApplication.class, args);  
 }  
}

再次测试ioc容器中是否有Student，就变成true了

1. 以上方法并不是Springboot推荐的做法，因为书写xml配置效率及其低下；

Springboot不推荐使用xml配置，而是推荐全注解的方式来给容器中添加组件（或者说把对象的创建交给ioc容器来管理，让容器来帮我们创建对象）；

把配置文件改成配置类：

@Bean标注在一个方法上，方法的返回值就是Bean的类型，方法名就是Bean的id

*/\*\*  
 \* 指明该类是一个配置类  
 \*/*@Configuration  
public class MyAppConfig {  
  
 //将方法的返回值添加到容器中；容器中这个组件的默认id就是方法名  
 @Bean  
 public Student student(){  
 System.*out*.println("配置类给容器中添加组件了");  
 return new Student();  
 }  
}

再去跑一边前面那个测试：

控制台成功输出了“配置类给容器中添加组件了”，并且是true，说明使用@Bean和@Configuration注解可以成功往容器中注册组件

#### 配置文件占位符

##### 随机数

${random.value}、${random.int}、${random.long}

${random.int(10)}、${random.int[1024,65536]}

##### 占位符获取之前配置的值，如果没有可用“：”指定默认值

person.last-name=张三${random.uuid}

person.age=${random.int}

person.birth=2017/12/15

person.boss=false

person.maps.k1=v1

person.maps.k2=14

person.lists=a,b,c

person.dog.name=${person.hello:hello}\_dog

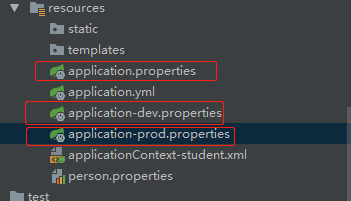
person.dog.age=15

#### 4.Profile

##### 1.多Profile文件

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

默认使用application.properties的配置；



##### yml支持多文档块方式

使用这样的方式更加方便，不需要写多个properties文件了

**server:  
 port:** 8081  
**spring:  
 profiles:  
 active:** prod  
  
---  
**server:  
 port:** 8083  
**spring:  
 profiles:** dev  
---  
  
**server:  
 port:** 8084  
**spring:  
 profiles:** prod *#指定属于哪个环境*

##### 3、激活指定profile

1、在配置文件中指定 spring.profiles.active=dev

在application.properties文件中进行配置：

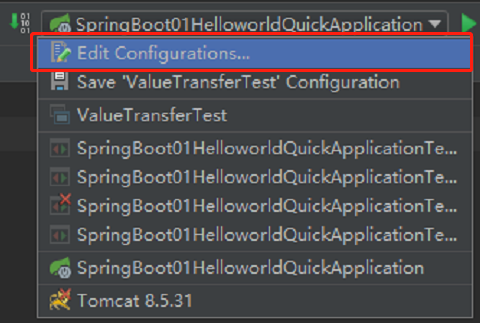
server.port=8081  
spring.profiles.active=dev

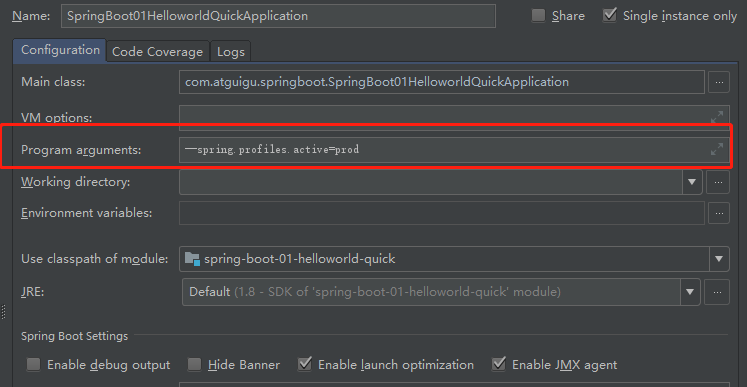
这样启动Springboot的端口号就改成application-dev.properties中配置的了

2、命令行：

java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=prod；

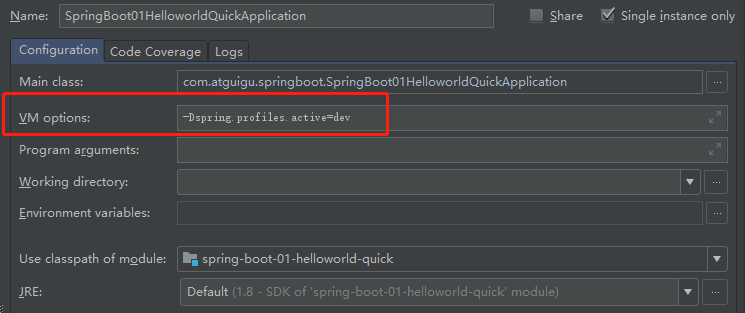
可以直接在测试的时候，配置传入命令行参数





3、虚拟机参数；

-Dspring.profiles.active=dev



#### 配置文件加载位置

springboot 启动会扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件

–file:./config/

–file:./ -------->当前项目的根路径下

–classpath:/config/

–classpath:/ ------->这个指的就是resources/application.properties

优先级由高到底，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主配置文件；互补配置；

==我们还可以通过spring.config.location来改变默认的配置文件位置==

**项目打包好以后，我们可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置；指定配置文件和默认加载的这些配置文件共同起作用形成互补配置；**

java –jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --spring.config.location=G:/application.properties

#### 外部配置的加载顺序

==SpringBoot也可以从以下位置加载配置； 优先级从高到低；高优先级的配置覆盖低优先级的配置，所有的配置会形成互补配置==

1.命令行参数

所有的配置都可以在命令行上进行指定

java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --server.port=8087 --server.context-path=/abc

多个配置用空格分开； --配置项=值

2.来自java:comp/env的JNDI属

3.Java系统属性（System.getProperties()）

4.操作系统环境变量

5.RandomValuePropertySource配置的random.\*属性值

==由jar包外向jar包内进行寻找；==

==优先加载带profile==

6.jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件

7.jar包内部的application-{profile}.properties或application.yml(带spring.profile)配置文件

==再来加载不带profile==

8.jar包外部的application.properties或application.yml(不带spring.profile)配置文件

9.jar包内部的application.properties或application.yml(不带spring.profile)配置文件

10.@Configuration注解类上的@PropertySource

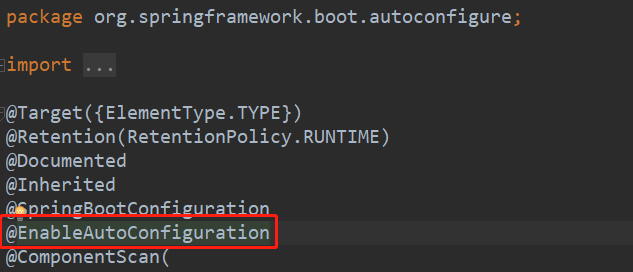
11.通过SpringApplication.setDefaultProperties指定的默认属性

#### 自动配置原理

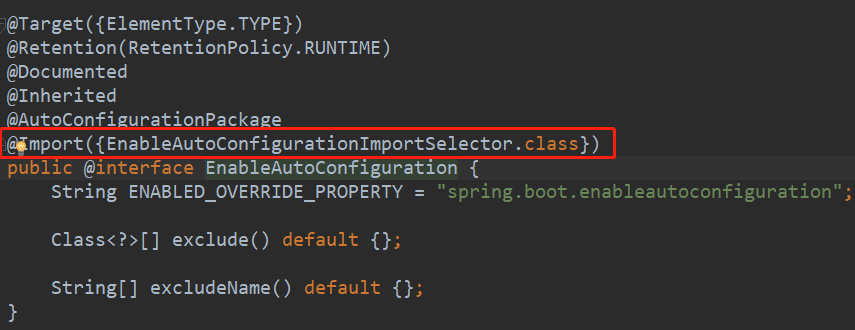
配置文件到底能写什么？怎么写？

##### 如何分析Spring boot的自动配置原理，分析步骤如下：

1. 点进主配置类中的@SpringbootApplication注解



1. 再点进@EnableAutoConfiguration注解



1. 再点进{EnableAutoConfigurationImportSelector选择器中，去查看它的父类AutoConfigurationImportSelector，它的父类提供了这么一个方法：selectImports()

该方法的返回值是一个由configurations集合对象转化得到的数组。

于是我们去看一下那个得到List集合对象configurations的方法getCandidateConfigurations

public String[] selectImports(AnnotationMetadata annotationMetadata) {  
 if (!this.isEnabled(annotationMetadata)) {  
 return NO\_IMPORTS;  
 } else {  
 try {  
 AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);  
 AnnotationAttributes attributes = this.getAttributes(annotationMetadata);  
 List<String> configurations = this.getCandidateConfigurations(annotationMetadata, attributes);  
 configurations = this.removeDuplicates(configurations);  
 configurations = this.sort(configurations, autoConfigurationMetadata);  
 Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);  
 this.checkExcludedClasses(configurations, exclusions);  
 configurations.removeAll(exclusions);  
 configurations = this.filter(configurations, autoConfigurationMetadata);  
 this.fireAutoConfigurationImportEvents(configurations, exclusions);  
 return (String[])configurations.toArray(new String[configurations.size()]);  
 } catch (IOException var6) {  
 throw new IllegalStateException(var6);  
 }  
 }  
}

1. 点进getCandidateConfigurations：

protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {  
List<String> configurations = SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClass(), this.getBeanClassLoader());  
 Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.factories. If you are using a custom packaging, make sure that file is correct.");  
 return configurations;  
}

点进getSpringFactoriesLoaderFactoryClass()，查看传到loadFactoryNames方法的参数：

protected Class<?> getSpringFactoriesLoaderFactoryClass() {  
 return EnableAutoConfiguration.class;  
}

可见，就是把EnableAutoConfiguration.class这个类传进去了

1. 点进loadFactoryNames方法：
2. 获取spring.factories文件的绝对路径url，把一大堆的url放在枚举类中
3. 创建一个用于收集添加到容器的组件集合result
4. 遍历枚举类对象urls中的每一条url
5. 使用Spring框架提供的工具类PropertiesLoaderUtils去读取（加载）每一条url所对应的配置文件，返回得到Properties对象（属性列表key-value键值对）

这里的key指的就是

org.springframework.boot.autoconfigure.EnableAutoConfiguration=

value指的就是

org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration

1. getProperty()方法将去搜索此属性列表中指定键的属性值，由于传进来的参数是

EnableAutoConfiguration.class，因此propertyValue指的就是那一大串的xxxAutoConfiguration，也就是要添加到ioc容器中的组件

public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {  
 String factoryClassName = factoryClass.getName();  
  
 try {  
 Enumeration<URL> urls = classLoader != null ? classLoader.getResources("META-INF/spring.factories") : ClassLoader.getSystemResources("META-INF/spring.factories");  
 ArrayList result = new ArrayList();  
  
 while(urls.hasMoreElements()) {  
 URL url = (URL)urls.nextElement();  
 Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource(url));  
 String propertyValue = properties.getProperty(factoryClassName);  
 String[] var8 = StringUtils.commaDelimitedListToStringArray(propertyValue);  
 int var9 = var8.length;  
  
 for(int var10 = 0; var10 < var9; ++var10) {  
 String factoryName = var8[var10];  
 result.add(factoryName.trim());  
 }  
 }  
  
 return result;  
 } catch (IOException var12) {  
 throw new IllegalArgumentException("Unable to load factories from location [META-INF/spring.factories]", var12);  
 }  
}

##### 自动配置原理：

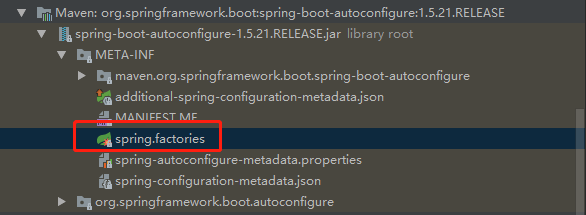
1. Springboot启动的时候会去加载主配置类，开启了自动配置功能@EnableAutoConfiguration
2. @EnableAutoConfiguration作用：

* 利用EnableAutoConfigurationImportSelector给容器中导入一些组件
* 可以查看selectImports()方法的内容；

其中有这么一个方法： 获取候选的配置（configurations获取到的就是那一大堆的xxxAutoConfiguration）

List<String> configurations = this.getCandidateConfigurations(annotationMetadata, attributes);

* 扫描所有jar包类路径下的META-INF/spring.factories文件
* 把META-INF/spring.factories文件的内容包装成一个properties对象
* 从properties对象中获取factoryClassName的值，并且使用add方法添加到新创建的ArrayList集合对象result中，直接返回，返回的结果就是要交给容器中的
* 从properties中获取到EnableAutoConfiguration.class类的类名对应的值，然后把它们添加在容器中



# Auto Configure  
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\  
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\  
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\  
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\  
。。。。后面省略

* 将类路径下META-INF/spring.factories里面配置的所有EnableAutoConfiguration的值加入到了容器中
* 每一个这样的xxxAutoConfiguration类都是容器中的一个组件，都加入到容器中；用他们来做自动配置；

1. 每一个自动配置类进行自动配置功能
2. 以HttpEncodingAutoConfiguration（Http编码自动配置）为例解释自动配置原理

//声明此类是一个配置类，和之前编写的配置文件一致，也可以给容器中添加组件

@Configuration

//启动指定类的ConfigurationProperties功能

//将配置文件中对应的值和HttpEncodingProperties绑定起来

//并把HttpEncodingProperties加入到ioc容器中  
@EnableConfigurationProperties({HttpEncodingProperties.class})

//Spring底层@Conditional注解（Spring注解版）

//根据不同的条件，如果满足指定的条件，整个配置类里面的配置就会生效；

//判断当前应用是否是web应用，如果是，当前配置类生效  
@ConditionalOnWebApplication

////判断当前项目有没有这个类CharacterEncodingFilter；

//SpringMVC中进行乱码解决的过滤器；  
@ConditionalOnClass({CharacterEncodingFilter.class})

//判断配置文件中是否存在某个配置 spring.http.encoding.enabled；

//如果不存在，判断也是成立的

//即使我们配置文件中不配置pring.http.encoding.enabled=true，

也是默认生效的；  
@ConditionalOnProperty(  
 prefix = "spring.http.encoding",  
 value = {"enabled"},  
 matchIfMissing = true  
)  
public class HttpEncodingAutoConfiguration {

//它已经和SpringBoot的配置文件映射了

private final HttpEncodingProperties properties;

//只有一个有参构造器的情况下，参数的值就会从容器中拿  
public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {  
 this.properties = properties;  
}

@Bean //给容器中添加一个组件，这个组件的某些值需要从properties中获取  
@ConditionalOnMissingBean({CharacterEncodingFilter.class})  
public CharacterEncodingFilter characterEncodingFilter() {  
 CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();  
 filter.setEncoding(this.properties.getCharset().name());  
 filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));  
 filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));  
 return filter;  
}

**根据当前不同的条件判断，决定这个配置类是否生效？**

**一但这个配置类生效；这个配置类就会给容器中添加各种组件；这些组件的属性是从对应的properties类中获取的，这些类里面的每一个属性又是和配置文件绑定的；**

**我们可以在配置文件中对配置类中的属性值进行修改，或者说是覆盖原有的默认配置**

spring.http.encoding.charset=utf-8

spring.http.encoding.force=true

1. 点进HttpEncodingProperties.class这个类中

所有在配置文件中能配置的有关“spring.http.encoding”的属性，都是在HttpEncodingProperties这个类中封装着。

配置文件中能配置什么属性，我们可以参照这个（包含某种功能）配置文件对应的配置类；比如当前的这个HttpEncodingProperties所包含的功能就是http编码自动配置，帮我们解决乱码问题的

//这个注解，前面刚学过，从配置文件中获取指定的值和bean的属性进行绑定

@ConfigurationProperties(  
 prefix = "spring.http.encoding"  
)  
public class HttpEncodingProperties {  
 public static final Charset DEFAULT\_CHARSET = Charset.forName("UTF-8");

##### 精髓：

**1）、SpringBoot启动会加载大量的自动配置类**

**2）、我们看我们需要的功能有没有SpringBoot默认写好的自动配置类；**

**3）、我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件有，我们就不需要再来配置了）**

**4）、给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们就可以在配置文件中指定这些属性的值；**

xxxxAutoConfigurartion：自动配置类；

给容器中添加组件

xxxxProperties:封装配置文件中相关属性；

**自动配置类必须在一定的条件下才能生效；**

我们怎么知道哪些自动配置类生效；

**==我们可以通过启用 debug=true属性；来让控制台打印自动配置报告==**，这样我们就可以很方便的知道哪些自动配置类生效；

Positive matches: （自动配置类启用的）

-----------------

DispatcherServletAutoConfiguration matched:

- @ConditionalOnClass found required class 'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

- @ConditionalOnWebApplication (required) found 'session' scope (OnWebApplicationCondition)

DispatcherServletAutoConfiguration.DispatcherServletConfiguration matched:

- @ConditionalOnClass found required class 'javax.servlet.ServletRegistration'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

- Default DispatcherServlet did not find dispatcher servlet beans (DispatcherServletAutoConfiguration.DefaultDispatcherServletCondition)

DispatcherServletAutoConfiguration.DispatcherServletRegistrationConfiguration matched:

- @ConditionalOnClass found required class 'javax.servlet.ServletRegistration'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

- DispatcherServlet Registration did not find servlet registration bean (DispatcherServletAutoConfiguration.DispatcherServletRegistrationCondition)

Negative matches: （没有启动，没有匹配成功的自动配置类）

-----------------

ActiveMQAutoConfiguration:

Did not match:

- @ConditionalOnClass did not find required classes 'javax.jms.ConnectionFactory', 'org.apache.activemq.ActiveMQConnectionFactory' (OnClassCondition)

AopAutoConfiguration:

Did not match:

- @ConditionalOnClass did not find required classes 'org.aspectj.lang.annotation.Aspect', 'org.aspectj.lang.reflect.Advice' (OnClassCondition)

**附加 Spring底层的Conditional注解原理要去看Spring注解版视频**

### 三、Springboot与日志

### 四、Springboot与Web开发

#### 1.简介

使用SpringBoot；

**1）、创建SpringBoot应用，选中我们需要的模块；**

**2）、SpringBoot已经默认将这些场景配置好了，只需要在配置文件中指定少量配置就可以运行起来**

**3）、自己编写业务代码；**

**自动配置原理？**

这个场景SpringBoot帮我们配置了什么？能不能修改？能修改哪些配置？能不能扩展？xxx

xxxxAutoConfiguration：帮我们给容器中自动配置组件；

xxxxProperties:配置类来封装配置文件的内容；

#### 2.Springboot对静态资源的映射规则

ResourceProperties

@ConfigurationProperties(  
 prefix = "spring.resources",  
 ignoreUnknownFields = false  
)  
public class ResourceProperties implements ResourceLoaderAware, InitializingBean {

//可以设置和静态资源有关的参数，如缓存时间等

WebMvcAutoConfiguration

public void addResourceHandlers(ResourceHandlerRegistry registry) {  
 if (!this.resourceProperties.isAddMappings()) {  
 logger.debug("Default resource handling disabled");  
 } else {  
 Integer cachePeriod = this.resourceProperties.getCachePeriod();  
 if (!registry.hasMappingForPattern("/webjars/\*\*")) {  
 this.customizeResourceHandlerRegistration(registry.addResourceHandler(new String[]{"/webjars/\*\*"}).addResourceLocations(new String[]{"classpath:/META-INF/resources/webjars/"}).setCachePeriod(cachePeriod));  
 }

。。。。。//省略  
 }  
}

1）、所有 /webjars/\*\* ，都去 classpath:/META-INF/resources/webjars/ 找资源；webjars：以jar包的方式引入静态资源；

<!--引入jquery-webjar-->在访问的时候只需要写webjars下面资源的名称即可

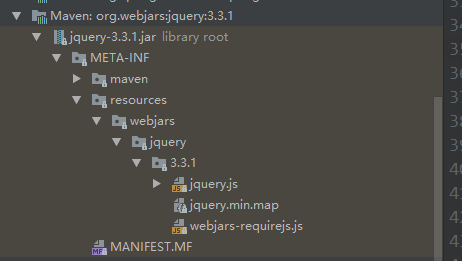
<dependency>

<groupId>org.webjars</groupId>

<artifactId>jquery</artifactId>

<version>3.3.1</version>

</dependency>



localhost:8080/webjars/jquery/3.3.1/jquery.js

2）、

也是WebMvcAutoConfiguration中的代码：

String staticPathPattern = this.mvcProperties.getStaticPathPattern();  
if (!registry.hasMappingForPattern(staticPathPattern)) {  
 this.customizeResourceHandlerRegistration(registry.addResourceHandler(new String[]{staticPathPattern}).addResourceLocations(this.resourceProperties.getStaticLocations()).setCachePeriod(cachePeriod));  
}

点进getStaticLocations方法里，进入ResourceProperties

public String[] getStaticLocations() {  
 return this.staticLocations;  
}

在ResourceProperties的构造器中：

public ResourceProperties() {  
 this.staticLocations = RESOURCE\_LOCATIONS;  
 this.addMappings = true;  
 this.chain = new ResourceProperties.Chain();  
}

private static final String[] RESOURCE\_LOCATIONS;

private static final String[] CLASSPATH\_RESOURCE\_LOCATIONS = new String[]{"classpath:/META-INF/resources/", "classpath:/resources/", "classpath:/static/", "classpath:/public/"};

private static final String[] SERVLET\_RESOURCE\_LOCATIONS = new String[]{"/"};

static {  
 RESOURCE\_LOCATIONS = new String[CLASSPATH\_RESOURCE\_LOCATIONS.length + SERVLET\_RESOURCE\_LOCATIONS.length];  
 System.arraycopy(SERVLET\_RESOURCE\_LOCATIONS, 0, RESOURCE\_LOCATIONS, 0, SERVLET\_RESOURCE\_LOCATIONS.length);  
 System.arraycopy(CLASSPATH\_RESOURCE\_LOCATIONS, 0, RESOURCE\_LOCATIONS, SERVLET\_RESOURCE\_LOCATIONS.length, CLASSPATH\_RESOURCE\_LOCATIONS.length);  
}

把静态资源RESOURCE\_LOCATIONS赋值给了staticLocations；而RESOURCE\_LOCATIONS是静态代码块中被创建出来的字符串数组，并且还给字符串数组定义了长度,长度= CLASSPATH\_RESOURCE\_LOCATIONS的长度+ SERVLET\_RESOURCE\_LOCATIONS的长度；并且把那两个静态数组中的内容都拷贝到RESOURCE\_LOCATIONS数组中；

访问当前项目的任何资源，都去（静态资源的文件夹）找映射

"classpath:/META-INF/resources/",

"classpath:/resources/",

"classpath:/static/",

"classpath:/public/"

"/"：当前项目的根路径

localhost:8080/abc === 去静态资源文件夹里面找abc

3）、欢迎页； 静态资源文件夹下的所有index.html页面；被"/\*\*"映射

@Bean  
public WebMvcAutoConfiguration.WelcomePageHandlerMapping welcomePageHandlerMapping(ResourceProperties resourceProperties) {  
 return new WebMvcAutoConfiguration.WelcomePageHandlerMapping(resourceProperties.getWelcomePage(), this.mvcProperties.getStaticPathPattern());  
}

点进getWelcomePage()方法中查看，即又进入了ResourceProperties中

public Resource getWelcomePage() {  
 String[] var1 = this.getStaticWelcomePageLocations();  
 int var2 = var1.length;

再点进getStaticWelcomePageLocations中：

private String[] getStaticWelcomePageLocations() {  
 String[] result = new String[this.staticLocations.length];  
  
 for(int i = 0; i < result.length; ++i) {  
 String location = this.staticLocations[i];  
 if (!location.endsWith("/")) {  
 location = location + "/";  
 }

//表示所有静态资源文件夹下的index.html都被认作是欢迎页

//上面列举的4条url代表静态资源文件  
 result[i] = location + "index.html";   
 }  
 return result;  
}

再点进上面代码的getStaticPathPattern()中

public String getStaticPathPattern() {  
 return this.staticPathPattern;  
}

public WebMvcProperties() {

。。。  
 this.staticPathPattern = "/\*\*";//表示index.html欢迎页被”/\*\*”锁映射  
 。。。  
}

需要放在classpath/static或者classpath/public或者classpath/resources或者classpath:/META-INF/resources/这几个静态资源文件夹下都能被”/\*\*映射”

localhost:8080/ 找index页面

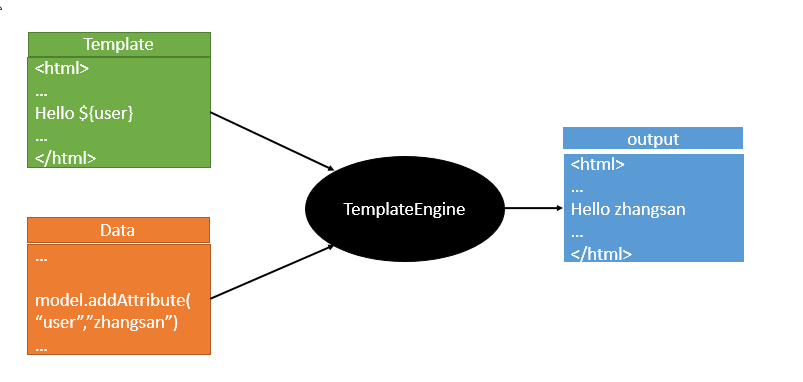
4）、所有的 \*\*/favicon.ico 都是在静态资源文件下找；

@Bean  
public SimpleUrlHandlerMapping faviconHandlerMapping() {  
 SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();  
 mapping.setOrder(-2147483647);  
 mapping.setUrlMap(Collections.singletonMap("\*\*/favicon.ico", this.faviconRequestHandler()));  
 return mapping;  
}

我们也可以对静态资源文件的默认路径进行修改：

只需要在application.properties文件中对spring. resources.staticLocations的值进行修改

#### 3.模板引擎



SpringBoot推荐的Thymeleaf：语法更简单，功能更强大

引入Thymeleaf的启动器

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

切换thymeleaf版本

<properties>

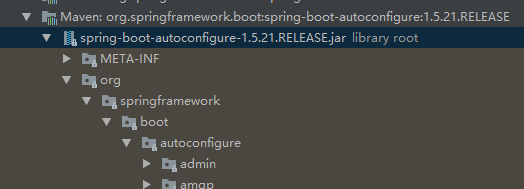
<thymeleaf.version>3.0.9.RELEASE</thymeleaf.version>

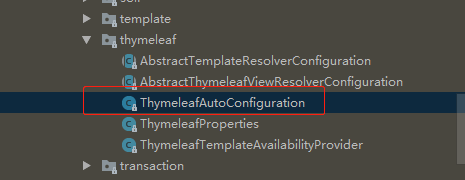
<!-- 布局功能的支持程序 thymeleaf3主程序 layout2以上版本 -->

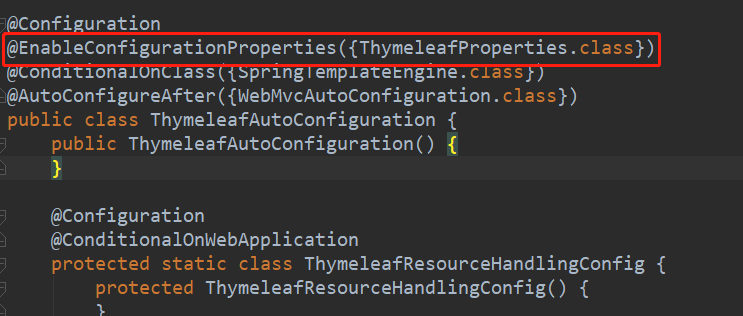
<!-- thymeleaf2 layout1-->

<thymeleaf-layout-dialect.version>2.2.2</thymeleaf-layout-dialect.version>

</properties>







点进ThymeleafProperties.class中查看：

@ConfigurationProperties(  
 prefix = "spring.thymeleaf"  
)  
public class ThymeleafProperties {  
 private static final Charset DEFAULT\_ENCODING = Charset.forName("UTF-8");  
 private static final MimeType DEFAULT\_CONTENT\_TYPE = MimeType.valueOf("text/html");  
 public static final String DEFAULT\_PREFIX = "classpath:/templates/";  
 public static final String DEFAULT\_SUFFIX = ".html";  
 private boolean checkTemplate = true;  
 private boolean checkTemplateLocation = true;  
 private String prefix = "classpath:/templates/";  
 private String suffix = ".html";  
 private String mode = "HTML5";  
 private Charset encoding;  
 private MimeType contentType;  
 private boolean cache;  
 private Integer templateResolverOrder;  
 private String[] viewNames;  
 private String[] excludedViewNames;  
 private boolean enabled;

可以看出：只要我们把HTML页面放在classpath:/templates/，thymeleaf就能自动渲染

学习使用Thymeleaf:

##### 1.导入thymeleaf的名称空间(不导入的话，就没有提示)

<html lang="en" xmlns:th="http://www.thymeleaf.org">

##### 2.使用thymeleaf

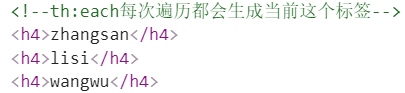
@RequestMapping("/success")  
public String success(Map<String,Object> maps){  
 maps.put("key","你好");  
 return "success";  
}

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
 <h1>成功！</h1>  
 <div th:text="${key}">这里显示map的value值，会替换原先的html的值</div>  
</body>  
</html>

##### 3.语法规则：（详细查阅文档）

@RequestMapping("/success")  
public String success(Map<String,Object> maps){  
 maps.put("key","<h1>你好！！！</h1>");  
 maps.put("users", Arrays.*asList*("zhangsan","lisi","wangwu"));  
 return "success";  
}

<body>  
 <h1>成功！</h1>  
 <hr/>  
 <!--th:text不转义，显示<h1>-->  
 <div th:text="${key}">111</div>  
 <!--th:utext转义，显示大标题内容-->  
 <div th:utext="${key}">222</div>  
 <hr/>  
 <!--th:each每次遍历都会生成当前这个标签-->  
 <h4 th:text="${users}" th:each="users:${users}"></h4>  
 <h4>  
 <span th:text="${users}" th:each="users:${users}"></span>  
 </h4>  
</body>





#### 4.SpringMVC自动配置