

# Java(多看源码)

## 一.JAVA基础

### 1.1-JAVA本身

1. JAVA是解释性语言（编译后的代码不能直接执行，**需要解释器**，换个角度，JAVA编写的代码保存后是为.java形式的，如果想运行，还需要额外再编译为.class文件才能在解释器中运行，**不要忘记编译了**）

（C/C++是编译性语言，可以直接被机器执行）

//JAVA会自动内存回收

//JAVA有接口,程序包,可以多线程

2. JAVA是跨平台的语言，运行逻辑：（一次编译，到处运行）

（.java是源文件，通过编译器的编译变成.class文件，如果出错就会编译出错，形成不了.class文件

.class是字节码文件）

//JDK：(Java Development Kit Java 开发工具包) JDK = JRE + java 的开发工具 [java, javac, javadoc, javap 等]

（安装路径不能有中文或特殊符号）

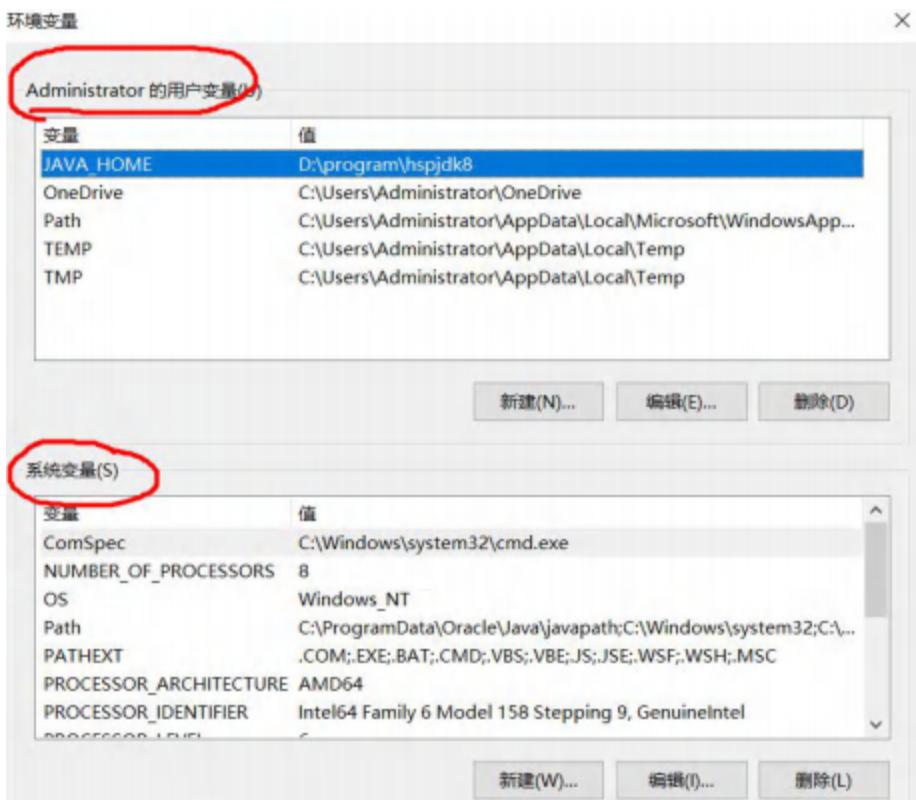
//JRE(Java Runtime Environment Java 运行环境) JRE = JVM + Java 的核心类库[类]

（有JRE就可以运行.class文件）

//JVM: Java 虚拟机 [java virtual machine]

3. 配置path（指定目录寻找执行程序,）：

1. 我的电脑--属性--高级系统设置--环境变量
2. 增加 JAVA\_HOME 环境变量, 指向 jdk的安装目录 d:\program\hspjdk8
3. 编辑 path 环境变量, 增加 %JAVA\_HOME%\bin
4. 打开DOS命令行, 任意目录下敲入javac/java。如果出现javac 的参数信息, 配置成功。



#### 4. 运行原理:



## 1.2-JAVA入门

### 1. 简单举例：

```

public class Hello { //编写一个 main 方法
public static void main(String[] args) {
    System.out.println("Darling,I love you!"); } //表示输出"hello,world~"到屏幕
}
  
```

//println输出自带换行,print不带

//一个源文件中最多只能有一个public类，且文件名要与该public类名相同（没有public类就对文件名不限定），其它类个数不限，编译后每一个类都会生成一个.class文件

//执行出口是main（类比C++），每个类都能有main，因为.class文件每个类都有，自然运行哪个类的.class，就是运行哪个类的main函数

//若要输出字符串加数字,格式可为:

\*\*System.out.println("a="+a);\*\*此处a为数字

## 2. 转义字符：

控制台输入 tab 键，可以实现命令补全,就是自动补齐一个语段

编写时，tab右移，shift+tab左移

\t：一个制表位，实现对齐的功能

```
String s1="hxq";
String s2="19";
String s3="100";
String s4="g";
String s5="xjh";
System.out.println("姓名\t"+ "年龄\t"+ "成绩\t"+ "性别\t"+ "爱好\n"+
s1+"\t"+s2+"\t"+s3+"\t"+s4+"\t"+s5);
```

```
I love 胡欣琦 forever~
姓名    年龄    成绩    性别    爱好
hxq      19      100      g        xjh
```

\n：换行符

\：一个\

"：一个"

'：一个'

\r :一个回车 System.out.println("Darling and I,\rlover hxq"); //会输出lover hxq and I,

## 3. 注释(先选中,再快捷键ctrl+/一次可注释多行)：

单行注释：//

多行注释：/\* \*/ (内部不允许嵌套多行注释)

文档注释（例）：

```
/*
*@author xjh&hxq
*@version 1.0
*/
```

## 4. 代码与命名规范：

- 类、方法的注释，要以javadoc的方式来写。
- 非Java Doc的注释，往往是给代码的维护者看的，着重告述读者为什么这样写，如何修改，注意什么问题等
- 使用tab操作，实现缩进，默认整体向右边移动，时候用shift+tab整体向左移
- 运算符和 = 两边习惯性各加一个空格。比如： $2 + 4 * 5 + 345 - 89$
- 源文件使用utf-8编码
- 行宽度不要超过80字符
- 代码编写次行风格和行尾风格(!!)

```
public ArrayList(int initialCapacity) {  
    if (initialCapacity > 0) {  
        this.elementData = new Object[initialCapacity];  
    } else if (initialCapacity == 0) {  
        this.elementData = EMPTY_ELEMENTDATA;  
    } else {  
        throw new IllegalArgumentException("Illegal Capacity: "+  
                                         initialCapacity);  
    }  
}
```

行尾风格,推荐

```
public ArrayList(int initialCapacity)  
{  
    if (initialCapacity > 0)  
    {  
        this.elementData = new Object[initialCapacity];  
    } else if (initialCapacity == 0)  
    {  
        this.elementData = EMPTY_ELEMENTDATA;  
    } else  
    {  
        throw new IllegalArgumentException("Illegal Capacity: "+  
                                         initialCapacity);  
    }  
}
```

次行风格

开发过程尽量不使用诸如a,b的变量名,这也是很容易理解的吧

- Java 对各种变量、方法和类等命名时使用的字符序列称为标识符
- 凡是自己可以起名字的地方都叫标识符 int num1 = 90;

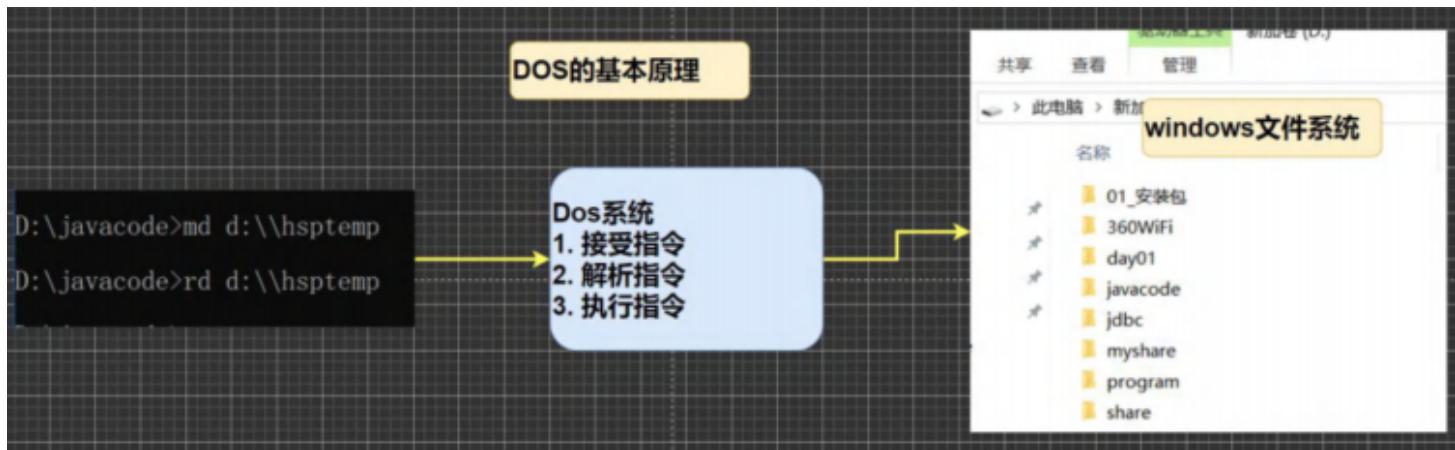
### ● 标识符的命名规则(必须遵守)

- 由26个英文字母大小写，0-9，\_或\$组成
- 数字不可以开头。int 3ab = 1;//错误
- 不可以使用关键字和保留字，但能包含关键字和保留字。
- Java中严格区分大小写，长度无限制。int totalNum = 10; int n = 90;
- 标识符不能包含空格。int a b = 90;

- 包名：多单词组成时所有字母都小写：aaa.bbb.ccc //比如 com.hsp.crm
- 类名、接口名：多单词组成时，所有单词的首字母大写：XxxYyyZzz [大驼峰]  
比如： TankShotGame
- 变量名、方法名：多单词组成时，第一个单词首字母小写，第二个单词开始每个单词首字母大写：xxxYyyZzz [小驼峰，简称 驼峰法]  
比如： tankShotGame
- 常量名：所有字母都大写。多单词时每个单词用下划线连接：XXX\_YYY\_ZZZ  
比如： 定义一个所得税率 TAX\_RATE

Java 保留字：现有 Java 版本尚未使用，但以后版本可能会作为关键字使用。自己命名标识符时要避免使用这些保留字 byValue、cast、future、generic、inner、operator、outer、rest、var、goto、const

## 5. DOS命令（就是windows控制台命令，了解）：



1) 查看当前目录是什么内容 dir

dir dir d:\abc2\test200

2) 切换到其他盘下：盘符号 cd : change directory

案例演示：切换到 c 盘 cd /D c:

3) 切换到当前盘的其他目录下 (使用相对路径和绝对路径演示), ..\ 表示上一级目录

案例演示： cd d:\abc2\test200 cd ..\..abc2\test200

4) 切换到上一级：

案例演示： cd ..

5) 切换到根目录： cd \

案例演示： cd \

6) 查看指定的目录下所有的子级目录 tree

7) 清屏 cls [苍老师]

8) 退出 DOS exit

9) 说明：因为小伙伴后面使用 DOS 非常少，所以对下面的几个指令，老韩给大家演示下，大家了解即可

## 6. 相对路径和绝对路径：

相对路径：从当前目录开始定位(..\ 表示上一目录)

绝对路径：从顶级目录开始定位 (c盘, d盘等)

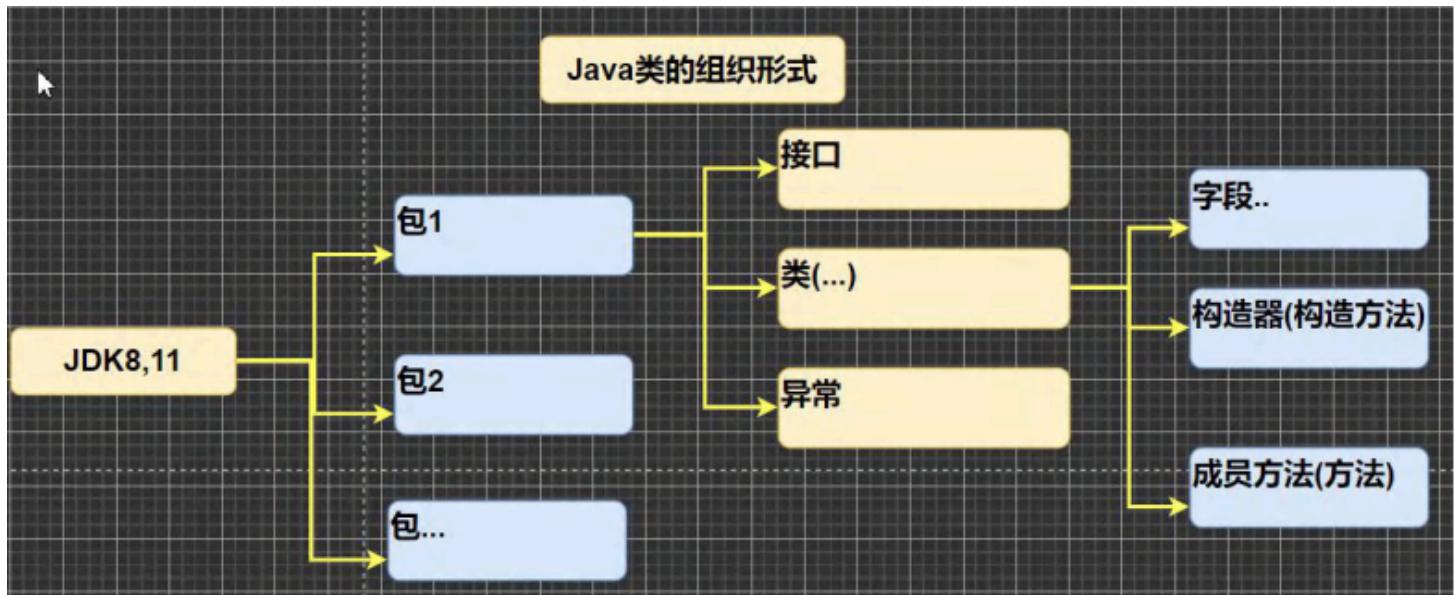
## 7. JAVA的API文档:

1. API (Application Programming Interface, 应用程序编程接口) 是 Java 提供的基本编程接口 (java提供的类还有相关的方法)。中文在线文档: <https://www.matools.com>



2. Java语言提供了大量的基础类，因此 Oracle公司也为这些基础类提供了相应的API文档，用于告诉开发者如何使用这些类，以及这些类里包含的方法。

3. Java类的组织形式 [!图]
4. 举例说明如何使用 ArrayList类有哪些方法.  
答: 包->类->方法  
直接索引。Math



## 8. 字符编码表

### ● 介绍一下字符编码表 [sublime测试]

ASCII (ASCII 编码表 一个字节表示 , 一个128个字符, 实际上一个字节可以表示256个字符,只用128个)  
Unicode (Unicode 编码表 固定大小的编码 使用两个字节来表示字符, 字母和汉字统一都是占用两个字节这样浪费空间 )

utf-8 (编码表, 大小可变的编码 字母使用1个字节, 汉字使用3个字节)

gbk (可以表示汉字, 而且范围广, 字母使用1个字节, 汉字2个字节)

gb2312 (可以表示汉字, gb2312 < gbk)

big5 码(繁体中文, 台湾, 香港)

# 1.3 变量

## 1. 三个要素：

1. 类型(int是4个字节,double是8个字节)
2. 名称(在同一个作用域不能重名,即例如已经声明一个变量int a=50,不能在后续再添加一个int a=70,  
应当直接写作a=70)
3. 值

## 2. 声明及赋值规则同C++

## 3. "+"运算

1. 两方均为数值时加法运算(char类型算作数值,哪怕为中文字符,结果也为其数值运算,结果是一个数)
2. 其一为字符串时作拼接运算
3. 顺序为从左到右

## 4. 数据类型(JAVA版):

### 1. 基本数据类型:

1. int(整数类型,以下皆为全拼)
  1. byte:1字节(1字节,1byte = 8bit)
  2. short:2字节
  3. int:4字节(2的31次方31-1 ~ -2的31次方)
  4. long:8字节(声明且只有long常量后面要加'L'或者'L',别的类型后面加会报错,形式:long a = 100L;)
2. 浮点数 = 符号位+指数位+尾数位

(浮点类型,即小数类型,尾数位可能会丢失,让精度损失,也就是说float和double类型运算可能会出现小数点后很多位数值有偏差的情况,运算时需要格外注意这点,因此小数都是近似值)

类 型	占 用 存 储 空 间	范 围
单精度float	4字节	-3.403E38 ~ 3.403E38
双精度double	8字节	-1.798E308 ~ 1.798E308

- 与整数类型类似, Java 浮点类型也有固定的范围和字段长度, 不受具体OS的影响。[float 4 个字节 double 是 8个字节]
  - Java 的浮点型常量(具体值)默认为double型, 声明float型常量, 须后加 'f' 或 'F'
  - 浮点型常量有两种表示形式**
    - 十进制数形式: 如: 5.12 512.0f .512 (必须有小数点)
    - 科学计数法形式:如: 5.12e2 [5.12\*10的2次方] 5.12E-2 [5.12/10的2次方]
  - 通常情况下, 应该使用double型, 因为它比float型更精确。[举例说明]
 

```
double num9 = 2.1234567851;
float num10 = 2.1234567851F;
```
  - 浮点数使用陷阱: 2.7 和 8.1 / 3 比较
- ```
double num7 = 2.7;
double num8 = 8.1 / 3;
if( Math.abs(num7 - num8) < 0.00001) {
    System.out.println("相等~~");
```

// 此处3.的0,512允许省略0,而浮点数512会变成512.0来突出有小数点的特征,对于小数直接的相等运算是计算差值绝对值在某个精度区间判断,而非以完全相等判断,当然,直接赋值是精确可相等的

- float:4字节(必须在常量后加f或F,因为浮点数的值默认为double类型,所以要特别说明)
- double:8字节(可以存放float类型的值,也就是说double b = 3.0F可以,不过空间会有浪费)
- char(单个字符型,存放单个字符'a',常用单引号,可以将字符型赋值为特殊字符型常量,例如char c3 = '\n',多个字符用字符串string):2字节  
(char的本质是整数,对应的unicode码,所以可以加减运算,同C++)
- boolean(布尔型,存放true和false):1字节  
//若要return布尔型的数据,return true;return false;

## 2. 引用数据类型:

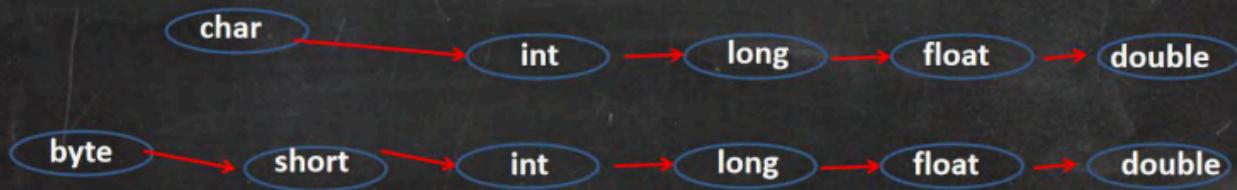
- class (类)
- interface (接口)
- [] (数组)

## 1. 基本数据类型转换:

### ✓ 介绍

当java程序在进行赋值或者运算时，精度小的类型自动转换为精度大的数据类型，这个就是**自动类型转换**。

### ✓ 数据类型按精度(容量)大小排序为(背，规则)



### ✓ 看一个基本案例 AutoConvert.java

```
int a = 'c';//对  
double d = 80;//对
```

## 3.15.2 自动类型转换注意和细节

1. 有多种类型的数据混合运算时，系统首先自动将所有数据转换成容量最大的那种数据类型，然后再进行计算。

```
int num2 = 10;  
float f = num2 + 1.2;  
double d2 = num2 + 1.2;
```

2. 当我们把精度(容量)大的数据类型赋值给精度(容量)小的数据类型时，就会报错，反之就会进行自动类型转换。

3. (byte, short) 和 char 之间不会相互自动转换。

```
byte b = 10;  
char c = b;  
double d3 = 100;  
int num3 = 1.1;/
```

4. byte, short, char 他们三者可以计算，在计算时首先转换为 int 类型。

5. boolean 不参与转换

6. 自动提升原则：表达式结果的类型自动提升为操作数中最大的类型

```
byte b = 10;  
char c = 90;  
short s = b+c;  
short s2 = 10 + 90;
```

### 看老师演示 AutoConvertDetail.java

```
//boolean b = 1;  
boolean b = true;  
int num1 = (int)b;:/
```

简而言之就是精度大的可以用精度小的赋值，而 char 本质上是一个整数，通过编码规则变成的字符，所以可以用 char 的字符赋值给 int 等精度更大的数据类型，精度大与精度小的运算结果以精度最大的数据类型为准。

## 2. 强制类型转换(从精度大变为精度小):

形式:int x = (int)10\*3.5

1. 当进行数据的大小从大——>小，就需要使用到强制转换
2. 强转符号只针对于最近的操作数有效，往往会使用小括号提升优先级

```
//int x = (int)10*3.5+6*1.5;  
int y = (int)(10*3.5+6*1.5);  
System.out.println(y);
```

3. char类型可以保存 int的常量值，但不能保存int的变量值，需要强转

```
char c1 = 100; //ok  
int m = 100; //ok  
char c2 = m; //错误  
char c3 = (char)m; //ok  
System.out.println(c3); //100对应的字符
```

4. byte和short, char 类型在进行运算时，当做int类型处理。

//char只能存int的常量,不能用int的变量赋值

//强转符号只针对于最近的操作数有效，往往会使用小括号提升优先级

//int x = (int)10\*3.5+6\*1.5;//编译错误： double -> int

int x = (int)(10\*3.5+6\*1.5); // (int)44.0 -> 44

### 判断是否能够通过编译

1. short s = 12; //ok  
s = s-9; //错误 int ->short
2. byte b = 10; //ok  
b = b + 11; //错误 int->byte  
b = (byte)(b+11); //正确， 使用强转
3. char c = 'a'; //ok  
int i = 16; //ok  
float d = .314F; //ok  
double result = c + i + d; //ok float->double
4. byte b = 16; //ok  
short s = 14; //ok  
short t = s + b; //错误 int ->short

//short,byte,char运算结束自动变成int

//需要跟数字运算时,数字一般默认为整数int,小数double (另一个角度上,为了凸显小数,即便是整数也要变成xx.0),注意可能需要强制转换

### 3. 与String互相转换注意String的S一定是大写的,否则错误:

#### 1. 基本类型转String:

```
int n1 = 100;
float n2 = 1.1f;
double n3 = 3.4;
boolean b1 = true;
String str1 = n1 + "";
String str2 = n2 + "";
String str3 = n3 + "";
String str4 = b1 + "";
System.out.println(str1 + " " + str2 + " " + str3 + " " + str4);
```

//在后面加上 +"" 即可

#### 2. String转基本类型:

```
Integer.parseInt("123");
Double.parseDouble("123.1");
Float.parseFloat("123.45");
Short.parseShort("12");
Long.parseLong("12345");
Boolean.parseBoolean("true");
Byte.parseByte("12");
```

//int的不能用例如double和float的String类型来转变,此处也遵循精度大的兼容精度小的规则,boolean不被看作0和1,所以也不能转换,否则会出现异常,程序终止

//String变成char时指String第一个字符变为char

## 5.Double和double和Decimal(精准计算小数)

Double是一个包装类,可以包含double类型的值,默认值null

double是一个原始数据类型,默认值0.0

BigDecimal:需要以字符串作为参数,可以精准计算小数,避免浮点的误差

而 Decimal 是精确计算，所以一般牵扯到金钱的计算，都使用 Decimal。

```
import java.math.BigDecimal;

public class BigDecimalExample {
    public static void main(String[] args) {
        BigDecimal num1 = new BigDecimal("0.1");
        BigDecimal num2 = new BigDecimal("0.2");

        BigDecimal sum = num1.add(num2);
        BigDecimal product = num1.multiply(num2);

        System.out.println("Sum: " + sum);
        System.out.println("Product: " + product);
    }
}

//输出
Sum: 0.3
Product: 0.02
```

# 1.4 运算符:(可以当作在C++的基础上加上python的&,|,%,/)

## 1. 算术运算符:

| 运算符 | 运算             | 范例            | 结果        |
|-----|----------------|---------------|-----------|
| +   | 正号             | +7            | 7         |
| -   | 负号             | b=11; -b      | -11       |
| +   | 加              | 9+9           | 18        |
| -   | 减              | 10-8          | 2         |
| *   | 乘              | 7*8           | 56        |
| /   | 除              | 9/9           | 1         |
| %   | 取模(取余)         | 11%9          | 2         |
| ++  | 自增(前) : 先运算后取值 | a=2;b=++a;    | a=3;b=3   |
| ++  | 自增(后) : 先取值后运算 | a=2;b=a++;    | a=3;b=2   |
| --  | 自减(前) : 先运算后取值 | a=2;b=- -a    | a=1;b=1   |
| --  | 自减(后) : 先取值后运算 | a=2;b=a- -    | a=1;b=2   |
| +   | 字符串相加          | "hsp" + "edu" | "hsp edu" |

// % 取余,本质为a % b = a - a / b \* b(与python语法相同)

// 除号,10/4 = 2,而10.0/4 = 2.5,此处注意数据类型(与python语法相同)

// ++和--与C++同理,例:i=i++,i不变

## 2. 关系运算符(结果均为boolean型):

| 运算符        | 运算        | 范例                      | 结果    |
|------------|-----------|-------------------------|-------|
| ==         | 相等于       | 8==7                    | false |
| !=         | 不等于       | 8!=7                    | true  |
| <          | 小于        | 8<7                     | false |
| >          | 大于        | 8>7                     | true  |
| <=         | 小于等于      | 8<=7                    | false |
| >=         | 大于等于      | 8>=7                    | true  |
| instanceof | 检查是否是类的对象 | "hsp" instanceof String | true  |

//注意instanceof

//(不论是否为中文)字符串判断是否相等时,使用equals()函数,如下:

```
public static void main(String[] args) {  
    Scanner inp = new Scanner(System.in, charsetName: "GBK");  
    String ans = "我爱你";  
    do{  
        System.out.println("胡欣琦说:你爱我吗");  
        ans=inp.next();  
        if(ans.equals(anObject:"我爱你")){  
            System.out.println("胡欣琦亲了徐璟昊一下");  
        }  
    }while(!ans.equals(anObject:"我爱你"));  
}
```

### 3. 逻辑运算符:

1. & 逻辑与:全1出1,否则为0(两条件均判断)
2. && 短路与:全1出1,否则为0(若前者为0,则直接跳过第二个条件,出0)
3. | 逻辑或:有1出1,全0出0(同理,两条件均判断)
4. || 短路或:有1出1,全0出0(同理,前者为1则跳过后者,直接出1)
5. ! 非/取反:与C++同理
6. ^ 逻辑异或:不同出1,相同出0

### 4. 赋值运算符(=,+=,-=等,与C++同理):

1. 运算顺序从右往左
2. 赋值运算符的左边只能是变量,右边可以是变量、表达式、常量值
3. 复合赋值运算符会进行类型转换。

```
//复合赋值运算符会进行类型转换  
byte b = 3;  
b += 2; // 等价 b = (byte)(b + 2);  
b++; // b = (byte)(b+1);
```

### 5. 三元运算符

条件表达式 ? 表达式 1: 表达式 2;

运算规则:

1. 如果条件表达式为 true, 运算后的结果是表达式 1;
2. 如果条件表达式为 false, 运算后的结果是表达式 2;

//与C++同理

//可参与类型转换

```
int a = 3;
```

```
int b = 8;
```

```
int c = a > b ? (int)1.1 : (int)3.4;//可以的
```

```
double d = a > b ? a : b + 3;//可以的，满足 int -> double
```

## 6. 运算符优先级:

一览表，不要背，使用多了，就熟悉了。

|      |                      |
|------|----------------------|
|      | . 0 0 ; ,            |
| R->L | ++ -- ~ !(data type) |
| L->R | * / %                |
| L->R | +                    |
| L->R | << >> >>> 位移         |
| L->R | < > <= >= instanceof |
| L->R | == !=                |
| L->R | &                    |
| L->R | ^                    |
| L->R |                      |
| L->R | &&                   |
| L->R |                      |
| L->R | ? :                  |
| R->L | = *= /= %=           |
|      | + = -= <<= >>=       |
|      | >>>= &= ^=  =        |

## 7. 进制:

1. 二进制：0,1 .以**0b 或 0B** 开头

```
int n1=0b1010;为10
```

2. 十进制：0-9

```
int n2=1010;为1010
```

3. 八进制：0-7， 以**数字 0**开头表示

```
int n3=01010;为520
```

4. 十六进制：0-9 及 A(10)-F(15) ,以**0x 或0X**开头,此处A~F不区分大小写

```
int n4=0X1010;为4112
```

## 8. 位运算:

1. 运算符:

|        |                           |
|--------|---------------------------|
| 按位与&   | : 两位全为 1 , 结果为1, 否则为0     |
| 按位或    | : 两位有一个为1, 结果为1, 否则为0     |
| 按位异或 ^ | : 两位一个为0,一个为1, 结果为1, 否则为0 |
| 按位取反~  | : 0->1 ,1->0              |

算术右移 >>: 低位溢出,符号位不变,并用符号位补溢出的高位

算术左移 <<: 符号位不变,低位补 0

>>> 逻辑右移也叫无符号右移,运算规则是: 低位溢出, 高位补 0

特别说明: 没有 <<< 符号

//注意<<与>>不改变正负,准确来描述是改变绝对值

2. 原码补码反码:

对于有符号的而言:

1. 二进制的最高位是符号位: 0表示正数,1表示负数 (老韩口诀: 0->0 1-> -)
2. 正数的原码, 反码, 补码都一样 (三码合一)
3. 负数的反码=它的原码符号位不变, 其它位取反(0->1,1->0)
4. 负数的补码=它的反码+1, 负数的反码 = 负数的补码 - 1
5. 0的反码, 补码都是0
6. java没有无符号数, 换言之, java中的数都是有符号的
7. 在计算机运算的时候, 都是以**补码**的方式来运算的.
8. 当我们看运算结果的时候, 要看他的**原码**(重点)

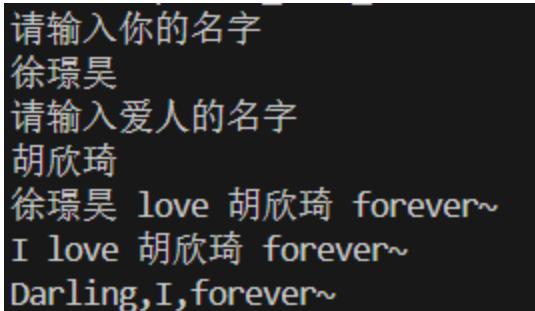
# 1.5 输入

## 1. 键盘输入语句形式：

```
import java.util.Scanner;//导入类Scanner所在的包,类比C++的函数库
public class Helloworld{
    public static void main(String[] args) {
        Scanner myScanner=new Scanner(System.in,"GBK");
        //创建一个Scanner类的对象
        //中文需声明编码格式,否则中文变乱码

        System.out.println("请输入你的名字");
        String lover1=myScanner.next();
        System.out.println("请输入爱人的名字");
        String lover2=myScanner.next();
        //next()接收字符串,nextInt()接收数字,nextDouble()接收小数...
        //要是懒得改函数类型那就要手动强制改变数据类型
        //next().charAt(0)或者写成这样,0代表把字符串索引位置为0的字符提出变为char
        //next会等待用户输入,再将用户输入赋值给lover1,lover2

        System.out.println( lover1+" love "+lover2+" forever~");
        String mylover = "胡欣琦";
        System.out.println("I love "+mylover+" forever~");
        System.out.println("Darling,I,forever~");
        //这一段是夹杂的私货XD
    }
}
```



A terminal window displaying the output of a Java program. The program prompts the user for their name and the name of their lover, then prints them out with the string "love" and "forever~". The output is as follows:

```
请输入你的名字
徐璟昊
请输入爱人的名字
胡欣琦
徐璟昊 love 胡欣琦 forever~
I love 胡欣琦 forever~
Darling,I,forever~
```

## 2. 流程(逻辑结构,递归在1.7里面):

1. 顺序(字面意思)
2. 分支(if,else if,else,与C++语法相同)
3. 嵌套(字面意思,分支套分支,建议不超过三层,为了阅读)
4. switch分支(与C++语法相同):

```

//记得导入 import java.util.Scanner;
Scanner inp=new Scanner(System.in);
String str1=inp.next();
switch(str1){
    case "a":System.out.println("星期一");
    case "b":System.out.println("星期二");
    case "c":System.out.println("星期三");break;
    case "d":System.out.println("星期四");break;
    case "e":System.out.println("星期五");break;
    case "f":System.out.println("星期六");break;
    case "g":System.out.println("星期日");
    default:System.out.println("输错了你个小猪");break;
    //如果条件匹配语句不写break,则会无视后续条件继续往下实现所有的语句
    //直到遇到break语句退出或全部语句(包括default)实现完毕
}

```

```

a
星期一
星期二
星期三
I love 胡欣琦 forever~
b
星期二
星期三
I love 胡欣琦 forever~
d
星期四
I love 胡欣琦 forever~
g
星期日
输错了你个小猪
I love 胡欣琦 forever~

```

## 5. 循环(建议最多套三层循环,为了可读):

### i. for循环(语法同C++):

多语句条件:

```

int count = 3; ①
for (int i = 0, j = 0; ② i < count; i++, j += 2) { ③
    System.out.println("i=" + i + " j=" + j);
}

```

//增强for:

for(int i : nums){//i的数据类型要与nums中的一致(接口,类,枚举什么的)}

```
System.out.println(i)//依次从数组nums取出数据赋给i,直至取完退出
}
```

ii. while循环(语法同C++)

iii. do...while循环(语法同C++,先执行一次循环再同while):

循环变量初始化;

do{

    循环体(语句);

    循环变量迭代;

}while(循环条件);

iv. break(C++基础上,不同处:可以指定确定退出哪一层循环,未指定则默认退出最近的):

1. break语句出现在多层嵌套的语句块中时,可以通过标签指明要终止的是哪一层语句块 **BreakDetail.java**

2. 标签的基本使用

```
label1: { .....
label2:   { .....
label3:     {
      .....  
      break ;
      .....
    }
  }
}
```

```
label1:
for(int j = 0; j < 4; j++){
label2:
  for(int i = 0; i < 10; i++){
    if(i == 2){
      break label1;
    }
    System.out.println("i = " + i);
  }
}
```

老韩解读

- (1)break语句可以指定退出哪层
- (2)label1是标签,名字由程序员指定。
- (3)break后指定到哪个label就退出到哪里
- (4)在实际的开发中,老韩尽量不要使用标签。
- (5)如果没有指定break,默认退出最近的循环体

输出什么?并分析原因

//标签说明终止该条标签对应的循环

//不建议用标签是因为可读性

v. continue(C++基础上,不同处同break)

vi. return(同C++,直接跳出方法(即C++中的函数))

# 1.6数据结构

## 1. 数组:

- 1) 数组是多个相同类型数据的组合，实现对这些数据的统一管理
- 2) 数组中的元素可以是任何数据类型，包括基本类型和引用类型，但是不能混用。
- 3) 数组创建后，如果没有赋值，有默认值  
`int 0, short 0, byte 0, long 0, float 0.0, double 0.0, char \u0000, boolean false, String null`
- 4) 使用数组的步骤 1. 声明数组并开辟空间 2 给数组各个元素赋值 3 使用数组
- 5) 数组的下标是从 0 开始的。
- 6) 数组下标必须在指定范围内使用，否则报：下标越界异常，比如
- 7) 数组属引用类型，数组型数据是对象(object)

//若数据类型比数组类型精度小，则会强制类型转换，例如int变double

1. 静态初始化:例如:`double[] hens={x,y,z,...};double hens[]={x,y,z,...};`
2. 动态初始化:例如:
  - i. `double hens[]= new double[5];`
  - ii. `double hens[];`  
`hens= new double[5]`  
//先声明再分配空间  
//若要赋值:`double hens[]= new double[]{1.0,2.0,3.0,4.0,5.0};`  
//这个算静态数组,Java不能同时用 [...] 和 {...}，也就是说不能同时指定数组大小和初始化值  
//声明方式:`int[] x` 或者 `int x[]`;推荐前者,方便阅读
3. 查询:
  - i. 单个数据(同C++语法下标查询)
  - ii. 长度:`hens.length`即可
4. 赋值(数组赋值默认为引用传递,赋值地址,即指向同一地址的值):  
//也就是说基本数据类型赋值是独立赋值,互不影响  
//也就是说可以通过逐个赋值基本类型数据来让数组间数据相同又各自独立  
//同时关于后续的类,赋值也为引用传递,指向同一地址

```

int[] arr1 = {1,2,3};
int[] arr2 = arr1;
arr[0] = 10;
//此处即为引用赋值
//(改变arr[0],同时会改变arr1[0]的值为10)

```

## 5. 数组扩容/删减:(创建新数组,拷贝添加/删减,再指向新数组)

```

arr = arrNew;
//指向新数组,也就是说两个数组此时指向地址相同,会一起改变数值
//可使用do...while来控制退出

```

## 6. 排序:

- i. 内部排序(将数据存到内部处理器中排序:**交换,选择,插入**)
- ii. 外部排序法(数据量过大需借助外部储存排序:**合并,直接合并**)
- iii. 冒泡排序法(字面意思)

## 7. 查找:

- i. 顺序查找(**若为字符串,则难以用二分法,用findNum.equals(num[p])**)
- ii. 二分法:(前提是**有固定规律可比较先后**)

```

while(first<=last){
    int p=(first+last)/2;
    if(num[p]==findNum){
        System.out.println("存在,下标为"+p);
        findFlag=1;
        break;
    }
    if(num[p]<findNum)
        first=p+1;//划重点
    if(num[p]>findNum)
        last=p-1;//划重点
}
if(findFlag==0)
System.out.println("不存在");

//私货:
String mylover = "胡欣琦";
System.out.println("I love "+mylover+" forever~");
System.out.println("Darling,I,forever~");

```

## 2. 二维数组:

(同C++,**arr[i][j]****arr[i].length**,即将arr[i]本身看作一个数组)

## 1. 动态初始化:

i. 类型[] 数组名=new 类型[大小][大小]

(例:int a[][]=new int[2][3])

//直接声明+分配

ii. int arr[][],

arr = new int[2][3];

//先声明再分配

iii. int[][] arr = new int[3][];

//通过for循环分别给每一列开辟空间

//适用于不同行列数不同的情况

## 2. 静态初始化:

例:int[][] arr = {{1,1,1}, {8,8,9},{100}};

//动态初始化也可以直接这样赋值过来

//声明方式:int[] y 或者 int[] y[] 或者 int y[][],

# 1.6 JMM(Java的多线程内存交互规则)

## 1. 关键概念：Java Memory Model(JMM)

1. 主内存(内存条/硬件):**所有线程共享的内存区域，存储共享变量**(全局变量、堆内存中的对象)
2. 工作内存(栈空间):JVM为每个线程私有的内存空间，**存储主内存变量的副本**(线程栈中的局部变量副本)
3. Happens-Before:定义**操作之间的顺序关系**，确保可见性(线程 A 的写操作对线程 B 可见)
4. 内存屏障(Memory Barrier):禁止特定类型的指令重排序(通过 volatile 或 synchronized 隐式插入)
5. 可见性问题:线程修改共享变量后,其它线程无法立即看到修改(**未同步到主内存**,可能陷入死循环)
6. 原子性问题:**复合操作被线程切换打断(如i++)**(原子性酷似SQL的"事务"概念)
7. 有序性问题:**编译器/处理器优化导致代码执行顺序与编写顺序不一致**

## 2. 具体内容:

1. 线程解锁前，必须把共享变量的值刷新回主内存
2. 线程加锁前，必须读取主内存的最新值到自己的工作内存
3. 加锁与解锁要同一把锁

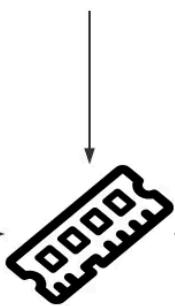
### 3. 数据储存,硬盘,内存和CPU

当数据量比较少的时候  
存储到数据库中  
mysql 数据库装在硬盘上



硬盘

当数据请求量大的时候，数据库反应不过来  
就需要更快的执行速度  
有时候我们将数据存储到内存中(比如说：redis)



内存

假设CPU计算完了，但内存还没  
反应过来，CPU不能干耗的等  
待，所以这时就有一个CPU缓存



CPU缓存

cpu 的计算速度更快



CPU

执行速度:硬盘<内存<CPU缓存

### 4. 一般进程:

线程从主内存拷贝变量,在私有的工作区域进行操作,再将变量写回主内存

### 5. volatile关键字的作用:(Happens-Before规则中的)

#### 5.1. 优点:保证可见性

1. 保证可见性:通过volatile对某个变量进行修饰,使得其在修改的第一时间就会**强制读写操作主内存,实现可见性**
2. 示例1:使用volatile防止指令重排序(单例模式的双重检查锁)

```

class Singleton {
    private static volatile Singleton instance;
    //singleton:单例模式,确保一个类只有一个实例(构造函数私有化),并用一个静态方法返回

    public static Singleton getInstance() {
        if (instance == null) { // 第一次检查
            synchronized (Singleton.class) {
                if (instance == null) { // 第二次检查
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}

```

## 2. 示例2:一次性发布(one-time safe publication)确保对象初始化完成后才对其他线程可见

```

class Singleton {
    private static volatile Singleton instance;
    //singleton:单例模式,确保一个类只有一个实例(构造函数私有化),并用一个静态方法返回

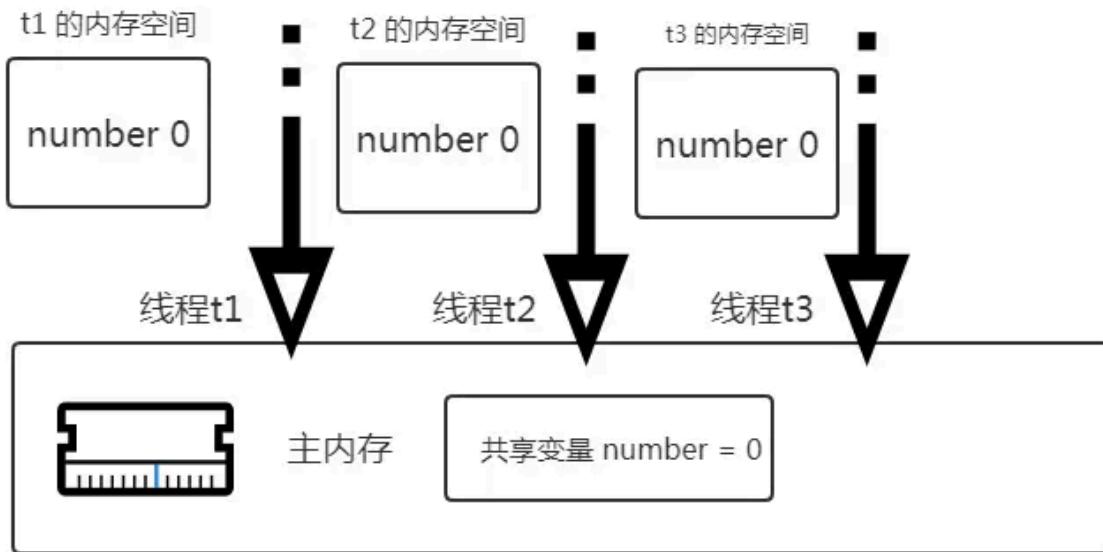
    public static Singleton getInstance() {
        if (instance == null) { // 第一次检查
            synchronized (Singleton.class) {
                if (instance == null) { // 第二次检查
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}

```

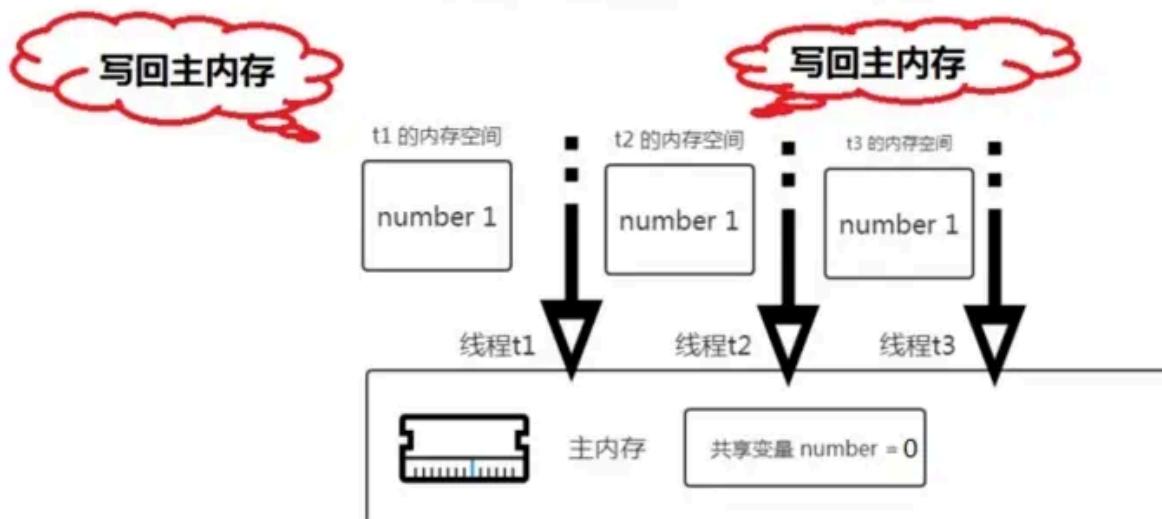
## 5.2 缺点及解决方法:不保证原子性

### 1. 原因:

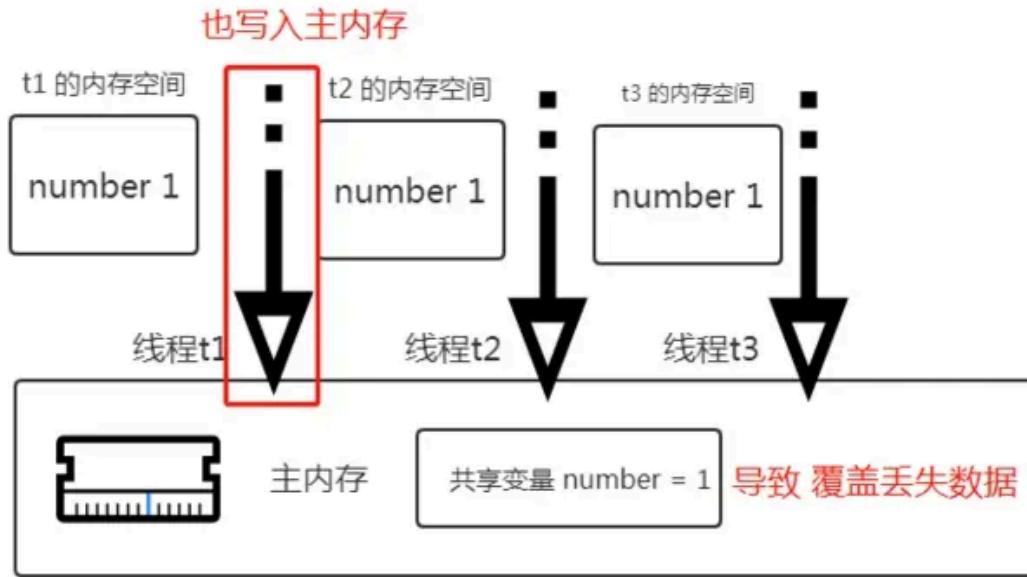
我们的t1、t2、t3首先要将变量从主内存拷贝的自己的工作内存空间，然后对变量进行操作



将变量从主内存拷贝的自己的工作内存空间，然后对变量进行操作操作完成后将变量写回主内存



但是抱歉：太快了，其他线程还没执行完后，在没有反应过来的前提下，之前挂起的t1 马上也写入主内存中



2. 原因例子:i++可分解为：

- 读取当前值 (read i)
  - 计算新值 (i + 1)
  - 写入新值 (write i)
- iv. 多线程情况下,可能被打断(线程 A 读取 i=5 → 线程 B 读取 i=5 → 线程 A 写入 i=6 → 线程 B 写入 i=6,结果应为7,实际得到6),volatile仅保证单次读/写原子性和可见性,而无法保证多步骤复合操作的原子性

3. 解决方法:

- 使用AtomicInteger:(CAS(compare-and-swap)机制将复合操作合并为一条原子指令)
- CAS原理:

```
public final int incrementAndGet() {  
    int current; // 当前值  
    int next; // 新值  
    do {  
        current = get(); // 读取当前值  
        next = current + 1; // 计算新值  
    } while (!compareAndSet(current, next)); // CAS 尝试写入  
    // 如果当前值等于 expectedValue，则原子性地更新为 newValue，返回 true；否则不更新，返回 false  
    return next;  
}
```

## 5.3 缺点:不替代锁机制

无法解决多个变量间的复合条件竞争问题

## 5.4 volatile,AtomicInteger,LongAdder选择

- 简单状态标志 → volatile(内存屏障)
- 计数器、累加器 → AtomicInteger(CAS依赖CPU指令+自旋(循环重试,无锁),依赖硬件,适合低竞争)
- 高并发统计 → LongAdder(分段累加,适合高竞争)

## 1.7 Java8新特性:

### 1.总览:

| 特性名称         | 描述                                                     | 示例或说明                                                                                              |
|--------------|--------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| Lambda 表达式   | 简化匿名内部类, 支持函数式编程                                       | (a, b) -> a + b 代替匿名类实现接口                                                                          |
| 函数式接口        | 仅含一个抽象方法的接口, 可用 <code>@FunctionalInterface</code> 注解标记 | Runnable, Comparator, 或自定义接口<br><code>@FunctionalInterface interface MyFunc { void run(); }</code> |
| Stream API   | 提供链式操作处理集合数据, 支持并行处理                                   | <code>list.stream().filter(x -&gt; x &gt; 0).collect(Collectors.toList())</code>                   |
| Optional 类   | 封装可能为 <code>null</code> 的对象, 减少空指针异常                   | <code>Optional.ofNullable(value).orElse("default")</code>                                          |
| 方法引用         | 简化 Lambda 表达式, 直接引用现有方法                                | <code>System.out::println</code> 等价于 <code>x -&gt; System.out.println(x)</code>                    |
| 接口的默认方法与静态方法 | 接口可定义默认实现和静态方法, 增强扩展性                                  | <code>interface A { default void print() { System.out.println("默认方法"); } }</code>                  |

|                   |                      |                                                                                               |
|-------------------|----------------------|-----------------------------------------------------------------------------------------------|
| 并行数组排序            | 使用多线程加速数组排序          | <code>Arrays.parallelSort(array)</code>                                                       |
| 重复注解              | 允许同一位置多次使用相同注解       | <code>@Repeatable</code> 注解配合容器注解使用                                                           |
| 类型注解              | 注解可应用于更多位置（如泛型、异常等）  | <code>List&lt;@NonNull String&gt; list</code>                                                 |
| CompletableFuture | 增强异步编程能力，支持链式调用和组合操作 | <code>CompletableFuture.supplyAsync(() -&gt; "result").thenAccept(System.out::println)</code> |

## 2.Lambda表达式：

Lambda 表达式它是一种简洁的语法，用于创建匿名函数，主要用于简化函数式接口（只有一个抽象方法的接口）的使用。其基本语法有以下两种形式：

- `(parameters) -> expression`：当 Lambda 体只有一个表达式时使用，表达式的结果会作为返回值。
- `(parameters) -> { statements; }`：当 Lambda 体包含多条语句时，需要使用大括号将语句括起来，若有返回值则需要使用 `return` 语句。

传统的匿名内部类实现方式代码较为冗长，而 Lambda 表达式可以用更简洁的语法实现相同的功能。比如，使用匿名内部类实现 `Runnable` 接口

```
public class AnonymousClassExample {  
    public static void main(String[] args) {  
        Thread t1 = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("Running using anonymous class");  
            }  
        });  
        t1.start();  
    }  
}
```

java

使用 Lambda 表达式实现相同功能：

```
public class LambdaExample {  
    public static void main(String[] args) {  
        Thread t1 = new Thread(() -> System.out.println("Running using lambda expression"));  
        t1.start();  
    }  
}
```

java

还有，Lambda 表达式能够更清晰地表达代码的意图，尤其是在处理集合操作时，如过滤、映射等。比如，过滤出列表中所有偶数

```
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
  
public class ReadabilityExample {  
    public static void main(String[] args) {  
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);  
        // 使用 Lambda 表达式结合 Stream API 过滤偶数  
        List<Integer> evenNumbers = numbers.stream()  
            .filter(n -> n % 2 == 0)  
            .collect(Collectors.toList());  
        System.out.println(evenNumbers);  
    }  
}
```

java

还有，Lambda 表达式使得 Java 支持函数式编程范式，允许将函数作为参数传递，从而可以编写更灵活、可复用的代码。比如定义一个通用的计算函数。

```
java
interface Calculator {
    int calculate(int a, int b);
}

public class FunctionalProgrammingExample {
    public static int operate(int a, int b, Calculator calculator) {
        return calculator.calculate(a, b);
    }

    public static void main(String[] args) {
        // 使用 Lambda 表达式传递加法函数
        int sum = operate(3, 5, (x, y) -> x + y);
        System.out.println("Sum: " + sum);

        // 使用 Lambda 表达式传递乘法函数
        int product = operate(3, 5, (x, y) -> x * y);
        System.out.println("Product: " + product);
    }
}
```

虽然 Lambda 表达式优点蛮多的，不过也有一些缺点，比如会增加调试困难，因为 Lambda 表达式是匿名的，在调试时很难定位具体是哪个 Lambda 表达式出现了问题。尤其是当 Lambda 表达式嵌套使用或者比较复杂时，调试难度会进一步增加。

## 3.stream的API

### 1. 介绍:

Java 8引入了Stream API，它提供了一种高效且易于使用的数据处理方式，特别适合集合对象的操作，如过滤、映射、排序等。Stream API不仅可以提高代码的可读性和简洁性，还能利用多核处理器的优势进行并行处理。让我们通过两个具体的例子来感受下Java Stream API带来的便利，对比在Stream API引入之前的传统做法。

**问题场景：**从一个列表中筛选出所有长度大于3的字符串，并收集到一个新的列表中。

**没有Stream API的做法：**

```
java
List<String> originalList = Arrays.asList("apple", "fig", "banana", "kiwi");
List<String> filteredList = new ArrayList<>();

for (String item : originalList) {
    if (item.length() > 3) {
        filteredList.add(item);
    }
}
```

这段代码需要显式地创建一个新的ArrayList，并通过循环遍历原列表，手动检查每个元素是否满足条件，然后添加到新列表中。

**使用Stream API的做法：**

```
java
List<String> originalList = Arrays.asList("apple", "fig", "banana", "kiwi");
List<String> filteredList = originalList.stream()
    .filter(s -> s.length() > 3)
    .collect(Collectors.toList());
```

这里，我们直接在原始列表上调用 `.stream()` 方法创建了一个流，使用 `.filter()` 中间操作筛选出长度大于3的字符串，最后使用 `.collect(Collectors.toList())` 终端操作将结果收集到一个新的列表中。代码更加简洁明了，逻辑一目了然。

计算数字总和：

## 没有Stream API的做法：

```
java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = 0;
for (Integer number : numbers) {
    sum += number;
}
```

这个传统的for-each循环遍历列表中的每一个元素，累加它们的值来计算总和。

## 使用Stream API的做法：

```
java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
    .mapToInt(Integer::intValue)
    .sum();
```

通过Stream API，我们可以先使用 `.mapToInt()` 将Integer流转换为IntStream（这是为了高效处理基本类型），然后直接调用 `.sum()` 方法来计算总和，极大地简化了代码。

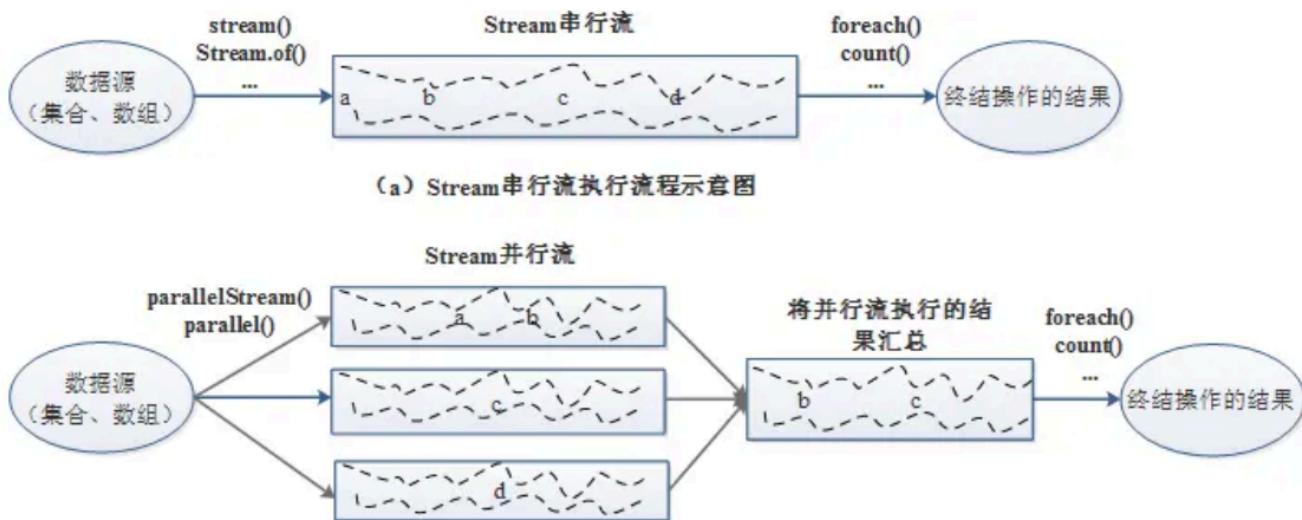
## 2.并行流:

## Stream流的并行API是什么？

是 ParallelStream。

并行流（ParallelStream）就是将源数据分为多个子流对象进行多线程操作，然后将处理的结果再汇总为一个流对象，底层是使用通用的 fork/join 池来实现，即将一个任务拆分成多个“小任务”并行计算，再把多个“小任务”的结果合并成总的计算结果

Stream串行流与并行流的主要区别：



对CPU密集型的任务来说，并行流使用ForkJoinPool线程池，为每个CPU分配一个任务，这是非常有效的，但是如果任务不是CPU密集的，而是I/O密集的，并且任务数相对线程数比较大，那么直接用 ParallelStream并不是很好的选择。

## 4.completableFuture

CompletableFuture是由Java 8引入的，在Java8之前我们一般通过Future实现异步。

- Future用于表示异步计算的结果，只能通过阻塞或者轮询的方式获取结果，而且不支持设置回调方法，Java 8之前若要设置回调一般会使用guava的ListenableFuture，回调的引入又会导致臭名昭著的回调地狱（下面的例子会通过ListenableFuture的使用来具体进行展示）。
- CompletableFuture对Future进行了扩展，可以通过设置回调的方式处理计算结果，同时也支持组合操作，支持进一步的编排，同时一定程度解决了回调地狱的问题。

下面将举例来说明，我们通过ListenableFuture、CompletableFuture来实现异步的差异。假设有三个操作 step1、step2、step3存在依赖关系，其中step3的执行依赖step1和step2的结果。

Future(ListenableFuture)的实现（回调地狱）如下：

java

```
ExecutorService executor = Executors.newFixedThreadPool(5);
ListeningExecutorService guavaExecutor = MoreExecutors.listeningDecorator(executor);
ListenableFuture<String> future1 = guavaExecutor.submit(() -> {
    //step 1
    System.out.println("执行step 1");
    return "step1 result";
});
ListenableFuture<String> future2 = guavaExecutor.submit(() -> {
    //step 2
    System.out.println("执行step 2");
    return "step2 result";
});
ListenableFuture<List<String>> future1And2 = Futures.allAsList(future1, future2);
Futures.addCallback(future1And2, new FutureCallback<List<String>>() {
    @Override
    public void onSuccess(List<String> result) {
        System.out.println(result);
        ListenableFuture<String> future3 = guavaExecutor.submit(() -> {
            System.out.println("执行step 3");
            return "step3 result";
        });
        Futures.addCallback(future3, new FutureCallback<String>() {
            @Override
```

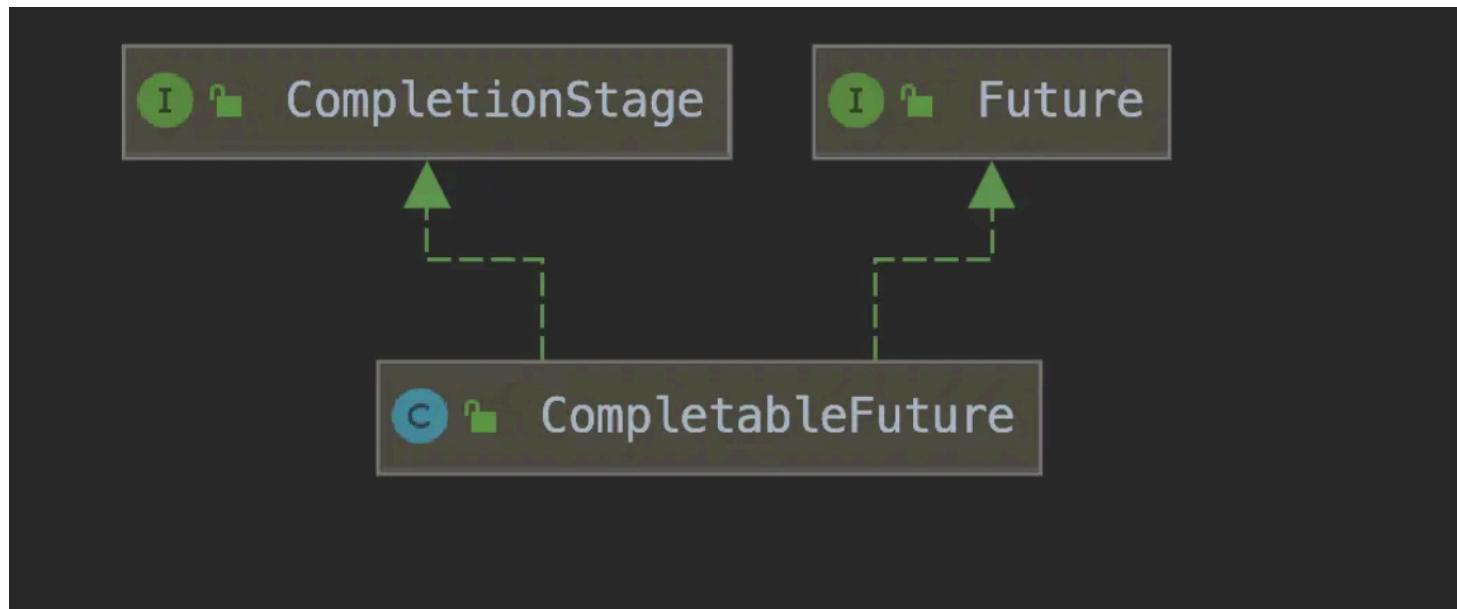
```
                public void onSuccess(String result) {
                    System.out.println(result);
                }
                @Override
                public void onFailure(Throwable t) {
                }
            }, guavaExecutor);
        }

        @Override
        public void onFailure(Throwable t) {
    }), guavaExecutor);
```

CompletableFuture的实现如下：

```
ExecutorService executor = Executors.newFixedThreadPool(5);
CompletableFuture<String> cf1 = CompletableFuture.supplyAsync(() -> {
    System.out.println("执行step 1");
    return "step1 result";
}, executor);
CompletableFuture<String> cf2 = CompletableFuture.supplyAsync(() -> {
    System.out.println("执行step 2");
    return "step2 result";
});
cf1.thenCombine(cf2, (result1, result2) -> {
    System.out.println(result1 + " , " + result2);
    System.out.println("执行step 3");
    return "step3 result";
}).thenAccept(result3 -> System.out.println(result3));
```

显然，CompletableFuture的实现更为简洁，可读性更好。



CompletableFuture实现了两个接口（如上图所示）：Future、CompletionStage。

- Future表示异步计算的结果，CompletionStage用于表示异步执行过程中的一*个*步骤（Stage），这个步骤可能是由另外一个CompletionStage触发的，随着当前步骤的完成，也可能会触发其他一系列CompletionStage的执行。
- 从而我们可以根据实际业务对这些步骤进行多样化的编排组合，CompletionStage接口正是定义了这样的能力，我们可以通过其提供的thenApply、thenCompose等函数式编程方法来组合编排这些步骤。

```
// 链式调用示例：异步获取价格并计算税费
CompletableFuture.supplyAsync(() -> fetchPrice()) // 异步获取价格
    .thenApply(price -> price * 1.2)           // 同步计算税费
    .thenAccept(total -> sendNotification(total)) // 异步发送通知
    .exceptionally(ex -> {                      // 统一异常处理
        log.error("任务失败", ex);
        return null;
    });
});
```

## 核心功能：

1. **任务创建**: `supplyAsync` (有返回值) 、 `runAsync` (无返回值) 。
2. **结果转换**: `thenApply` (同步处理) 、 `thenApplyAsync` (异步处理) 。
3. **结果消费**: `thenAccept` (消费结果) 、 `thenRun` (无结果触发动作) 。
4. **任务组合**: `thenCombine` (合并两任务结果) 、 `allOf / anyOf` (全完成/任一完成) 。
5. **异常处理**: `exceptionally` (捕获异常并兜底) 、 `handle` (结果与异常统一处理) 。

## 1.8 Java 21 新特性:

### 新语言特性:

- **Switch 语句的模式匹配:** 该功能在 Java 21 中也得到了增强。它允许在 `switch` 的 `case` 标签中使用模式匹配，使操作更加灵活和类型安全，减少了样板代码和潜在错误。例如，对于不同类型的账户类，可以在 `switch` 语句中直接根据账户类型的模式来获取相应的余额，如 `case savingsAccount sa ->`  
`result = sa.getSavings();`
- **数组模式:** 将模式匹配扩展到数组中，使开发者能够在条件语句中更高效地解构和检查数组内容。例如，`if (arr instanceof int[] {1, 2, 3})`，可以直接判断数组 `arr` 是否匹配指定的模式。
- **字符串模板（预览版）:** 提供了一种更可读、更易维护的方式来构建复杂字符串，支持在字符串字面量中直接嵌入表达式。例如，以前可能需要使用 `"hello " + name + ", welcome to the geeksforgeeks!"` 这样的方式来拼接字符串，在 Java 21 中可以使用 `hello {name}, welcome to the geeksforgeeks!` 这种更简洁的写法

### 并发特性方面:

- **虚拟线程:** 这是 Java 21 引入的一种轻量级并发的新选择。它通过共享堆栈的方式，大大降低了内存消耗，同时提高了应用程序的吞吐量和响应速度。可以使用静态构建方法、构建器或 `ExecutorService` 来创建和使用虚拟线程。
- **Scoped Values（范围值）:** 提供了一种在线程间共享不可变数据的新方式，避免使用传统的线程局部存储，促进了更好的封装性和线程安全，可用于在不通过方法参数传递的情况下，传递上下文信息，如用户会话或配置设置。

## 1.9 序列化和反序列化:

### 怎么把一个对象从一个jvm转移到另一个jvm?

- **使用序列化和反序列化:** 将对象序列化为字节流，并将其发送到另一个 JVM，然后在另一个 JVM 中反序列化字节流恢复对象。这可以通过 Java 的 `ObjectOutputStream` 和 `ObjectInputStream` 来实现。
- **使用消息传递机制:** 利用消息传递机制，比如使用消息队列（如 RabbitMQ、Kafka）或者通过网络套接字进行通信，将对象从一个 JVM 发送到另一个。这需要自定义协议来序列化对象并在另一个 JVM 中反序列化。
- **使用远程方法调用（RPC）:** 可以使用远程方法调用框架，如 gRPC，来实现对象在不同 JVM 之间的传输。远程方法调用可以让你在分布式系统中调用远程 JVM 上的对象的方法。
- **使用共享数据库或缓存:** 将对象存储在共享数据库（如 MySQL、PostgreSQL）或共享缓存（如 Redis）中，让不同的 JVM 可以访问这些共享数据。这种方法适用于需要共享数据但不需要直接传输对象的场景。

Java 默认的序列化虽然实现方便，但却存在安全漏洞、不跨语言以及性能差等缺陷。

- 无法跨语言： Java 序列化目前只适用基于 Java 语言实现的框架，其它语言大部分都没有使用 Java 的序列化框架，也没有实现 Java 序列化这套协议。因此，如果是两个基于不同语言编写的应用程序相互通信，则无法实现两个应用服务之间传输对象的序列化与反序列化。
- 容易被攻击： Java 序列化是不安全的，我们知道对象是通过在 ObjectInputStream 上调用 readObject() 方法进行反序列化的，这个方法其实是一个神奇的构造器，它可以将类路径上几乎所有实现了 Serializable 接口的对象都实例化。这也就意味着，在反序列化字节流的过程中，该方法可以执行任意类型的代码，这是非常危险的。
- 序列化后的流太大：序列化后的二进制流大小能体现序列化的性能。序列化后的二进制数组越大，占用的存储空间就越多，存储硬件的成本就越高。如果我们是进行网络传输，则占用的带宽就更多，这时就会影响到系统的吞吐量。

我会考虑用主流序列化框架，比如FastJson、Protobuf来替代Java 序列化。

如果追求性能的话，Protobuf 序列化框架会比较合适，Protobuf 的这种数据存储格式，不仅压缩存储数据的效果好，在编码和解码的性能方面也很高效。Protobuf 的编码和解码过程结合.proto 文件格式，加上 Protocol Buffer 独特的编码格式，只需要简单的数据运算以及位移等操作就可以完成编码与解码。可以说 Protobuf 的整体性能非常优秀。

如何实现序列化和反序列化：

## 将对象转为二进制字节流具体怎么实现？

其实，像序列化和反序列化，无论这些可逆操作是什么机制，都会有对应的[处理和解析协议](#)，例如加密和解密，TCP的粘包和拆包，序列化机制是通过序列化协议来进行处理的，和 class 文件类似，它其实是定义了序列化后的字节流格式，然后对此格式进行操作，生成符合格式的字节流或者将字节流解析成对象。

在Java中通过序列化对象流来完成序列化和反序列化：

- ObjectOutputStream：通过writeObject() 方法做序列化操作。
- ObjectInputStrean：通过readObject()方法做反序列化操作。

只有实现了Serializable或Externalizable接口的类的对象才能被序列化，否则抛出异常！

实现对象序列化：

- 让类实现Serializable接口：

```
import java.io.Serializable;  
  
public class MyClass implements Serializable {  
    // class code  
}
```

- 创建输出流并写入对象：

```
import java.io.FileOutputStream;  
import java.io.ObjectOutputStream;  
  
MyClass obj = new MyClass();  
try {  
    FileOutputStream fileOut = new FileOutputStream("object.ser");  
    ObjectOutputStream out = new ObjectOutputStream(fileOut);  
    out.writeObject(obj);  
    out.close();  
    fileOut.close();  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

实现对象反序列化：

- 创建输入流并读取对象：

```
java
import java.io.FileInputStream;
import java.io.ObjectInputStream;

MyClass newObj = null;
try {
    FileInputStream fileIn = new FileInputStream("object.ser");
    ObjectInputStream in = new ObjectInputStream(fileIn);
    newObj = (MyClass) in.readObject();
    in.close();
    fileIn.close();
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
```

通过以上步骤，对象obj会被序列化并写入到文件"object.ser"中，然后通过反序列化操作，从文件中读取字节流并恢复为对象newObj。这种方式可以方便地将对象转换为字节流用于持久化存储、网络传输等操作。需要注意的是，要确保类实现了Serializable接口，并且所有成员变量都是Serializable的才能被正确序列化。

# 1.10 跳表(数据结构)

## 1. 跳表是什么?

跳表是一种 **多层有序链表**，通过建立多级索引实现快速查找。其核心特点：

- **多层结构**: 每个节点随机拥有多个层级指针 (类似电梯的快速通道)
- **空间换时间**: 高层指针跨越更多节点，加速查找 (平均时间复杂度  $O(\log n)$ )
- **动态更新**: 插入/删除时自动调整索引层级，无需复杂平衡操作

**类比理解**: 想象一本字典，除了常规字母排序外，还有按首字母的快速索引页 (如A、M、Z)，跳表的多层索引类似这种多级导航机制。

## 2. 跳表的核心作用

- **快速访问有序数据**: 适用于高频查询、动态更新的场景 (如排行榜、实时日志)
- **替代平衡树**: 相比红黑树等结构，实现更简单且并发性能更好
- **典型应用**:
  - Redis**有序集合** (Sorted Set) 底层实现
  - LevelDB/RocksDB 的 MemTable 数据结构
  - Apache Lucene 的倒排索引合并

### 3.1 节点结构

Java

```
class SkipListNode {  
    int key;  
    int value;  
    SkipListNode[] forward; // 各层指针数组  
    // 如forward[0]为底层指针, forward[2]为第3层指针  
}
```

### 3.2 随机层数生成

Java

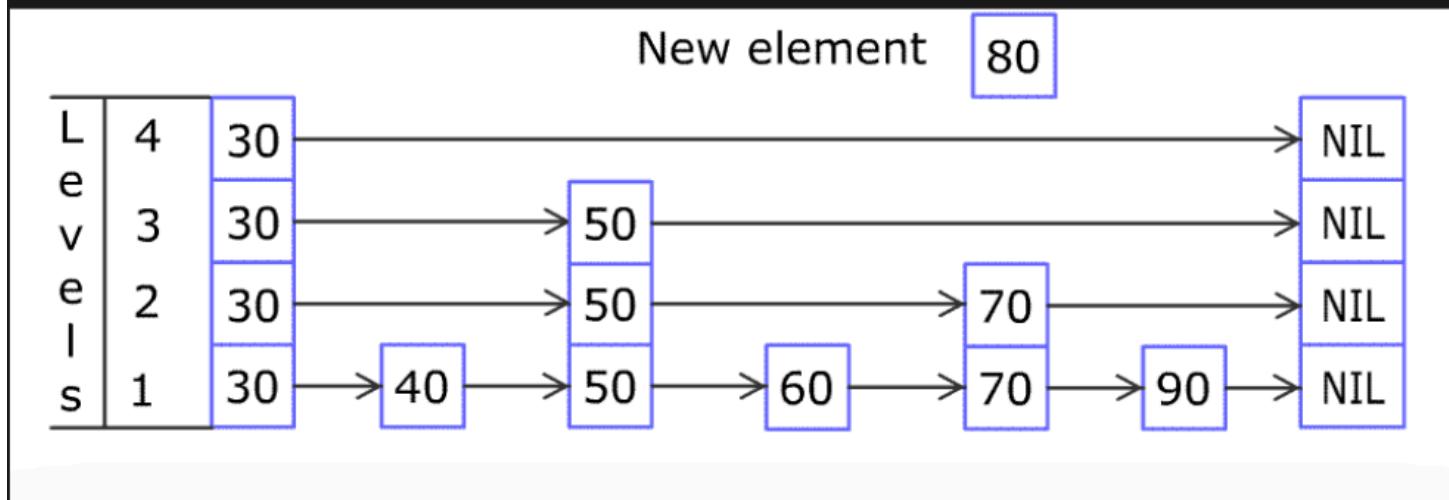
```
private int randomLevel() {  
    int level = 1;  
    while (Math.random() < P && level < MAX_LEVEL) {  
        level++;  
    }  
    return level;  
}
```

- **概率参数P**: 通常取0.25~0.5, 控制索引密度

- **MAX\_LEVEL**: 限制最大层数 (如Redis中设置为32层)

### 3.3 查找过程

1. 从最高层开始向右遍历, 找到最后一个小于目标值的节点
2. 降层继续查找, 直到最底层
3. 若找到相等节点则返回, 否则不存在



### 3.4 插入过程

1. 查找插入位置，记录各层的前驱节点
2. 随机生成新节点层数
3. 从底层到高层，将新节点插入到各层链表中

### 4. 设计思路剖析

| 设计原则   | 实现策略                              |
|--------|-----------------------------------|
| 多层索引   | 高层指针跨越更多节点，形成“快速通道”               |
| 随机化层数  | 避免极端情况（如所有节点都只有1层），保证平均性能         |
| 动态调整   | 插入/删除时自动更新索引，无需全局重构               |
| 空间效率权衡 | 通过概率控制索引密度（约每1/P个节点有高层指针），平衡时间与空间 |

### 5. 对比其他数据结构

| 特性      | 跳表                    | 红黑树               | 普通链表        |
|---------|-----------------------|-------------------|-------------|
| 查找时间复杂度 | $O(\log n)$ (平均)      | $O(\log n)$ (严格)  | $O(n)$      |
| 插入复杂度   | $O(\log n)$ (平均)      | $O(\log n)$ (需旋转) | $O(1)$ (无序) |
| 实现难度    | ★★★ (代码约100行)         | ★★★★★ (需处理多种旋转情况) | ★           |
| 并发性能    | ★★★ (CAS无锁优化)         | ★ (需全局锁)          | ★★ (局部锁)    |
| 内存占用    | 约多出30%-50%空间 (存储多层指针) | 每个节点2个指针          | 每个节点1个指针    |

优点：

- 代码实现简单 (比红黑树减少70%代码量)
- 天然支持范围查询 (如查找 $10 \leq \text{score} \leq 20$ 的数据)
- 并发优化友好 (分段锁或CAS无锁实现)

### 缺点:

- 内存占用较高 (Redis中跳表比字典多消耗约40%内存)
- 最坏情况下性能波动 (但概率极低)

## 7. 生产环境使用建议

- **选择场景:** 高频范围查询、需要动态更新的有序集合

- **调优参数:**

- **MAX\_LEVEL:** 根据数据规模调整 (默认16可支持 $2^{16}$ 个节点)
  - **上升概率P:** 降低P值减少内存，提高查询速度 (Redis使用P=0.25)

- **并发控制:**

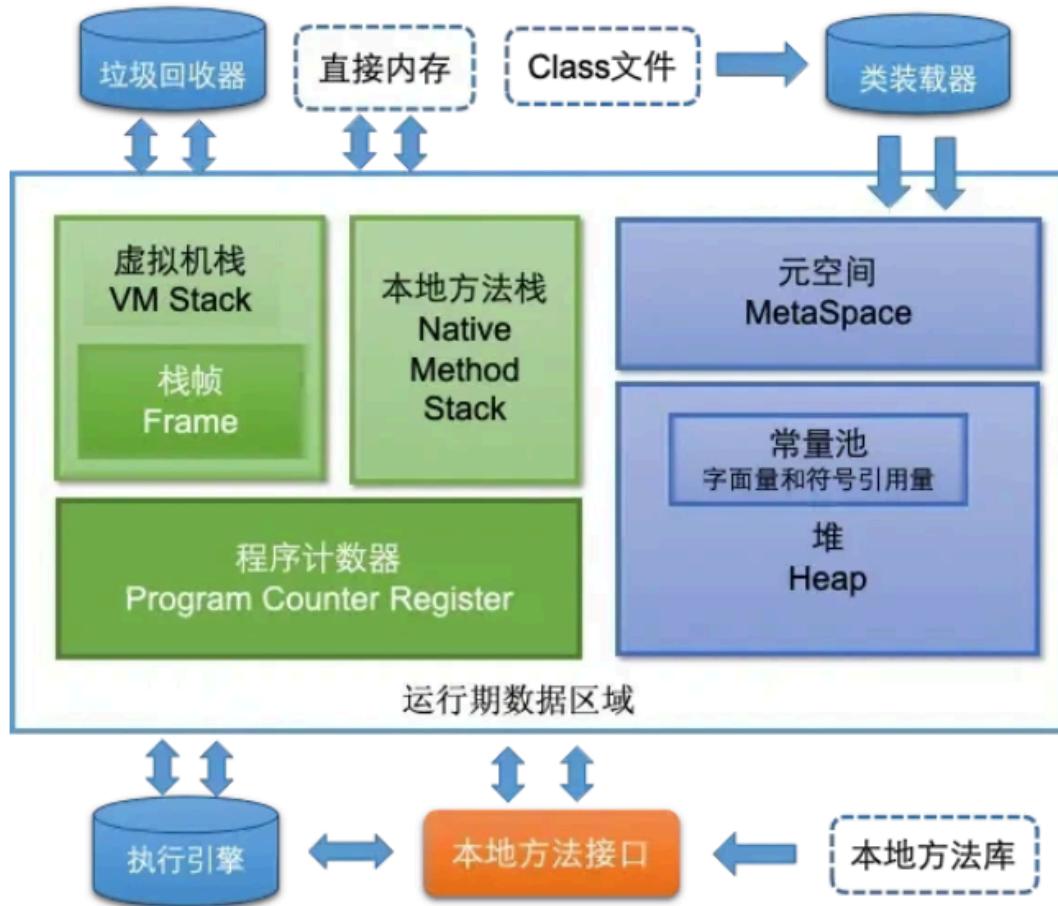
- 读多写少: 读写锁 (ReadWriteLock)
  - 高并发写入: 无锁CAS (如Java的 ConcurrentSkipListMap )

## 1.11 JVM:

### 1. 八股:

1. 内存模型:

根据 JDK 8 规范，JVM 运行时内存共分为虚拟机栈、堆、元空间、程序计数器、本地方法栈五个部分。还有一部分内存叫直接内存，属于操作系统的本地内存，也是可以直接操作的。



JVM的内存结构主要分为以下几个部分：

- **程序计数器**：可以看作是当前线程所执行的字节码的行号指示器，用于存储当前线程正在执行的 Java 方法的 JVM 指令地址。如果线程执行的是 Native 方法，计数器值为 null。是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域，生命周期与线程相同。
- **Java 虚拟机栈**：每个线程都有自己独立的 Java 虚拟机栈，生命周期与线程相同。每个方法在执行时都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。可能会抛出 StackOverflowError 和 OutOfMemoryError 异常。
- **本地方法栈**：与 Java 虚拟机栈类似，主要为虚拟机使用到的 Native 方法服务，在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。本地方法执行时也会创建栈帧，同样可能出现 StackOverflowError 和 OutOfMemoryError 两种错误。
- **Java 堆**：是 JVM 中最大的一块内存区域，被所有线程共享，在虚拟机启动时创建，用于存放对象实例。从内存回收角度，堆被划分为新生代和老年代，新生代又分为 Eden 区和两个 Survivor 区（From Survivor 和 To Survivor）。如果在堆中没有内存完成实例分配，并且堆也无法扩展时会抛出 OutOfMemoryError 异常。
- **方法区（元空间）**：在 JDK 1.8 及以后的版本中，方法区被元空间取代，使用本地内存。用于存储已被虚拟机加载的类信息、常量、静态变量等数据。虽然方法区被描述为堆的逻辑部分，但有“非堆”的别名。方法区可以选择不实现垃圾收集，内存不足时会抛出 OutOfMemoryError 异常。
- **运行时常量池**：是方法区的一部分，用于存放编译期生成的各种字面量和符号引用，具有动态性，运行时也可将新的常量放入池中。当无法申请到足够内存时，会抛出 OutOfMemoryError 异常。
- **直接内存**：不属于 JVM 运行时数据区的一部分，通过 NIO 类引入，是一种堆外内存，可以显著提高 I/O 性能。直接内存的使用受到本机总内存的限制，若分配不当，可能导致 OutOfMemoryError 异常。

## JVM内存模型里的堆和栈有什么区别？

- **用途：**栈主要用于存储局部变量、方法调用的参数、方法返回地址以及一些临时数据。每当一个方法被调用，一个栈帧（stack frame）就会在栈中创建，用于存储该方法的信息，当方法执行完毕，栈帧也会被移除。堆用于存储对象的实例（包括类的实例和数组）。当你使用 `new` 关键字创建一个对象时，对象的实例就会在堆上分配空间。
- **生命周期：**栈中的数据具有确定的生命周期，当一个方法调用结束时，其对应的栈帧就会被销毁，栈中存储的局部变量也会随之消失。堆中的对象生命周期不确定，对象会在垃圾回收机制（Garbage Collection, GC）检测到对象不再被引用时才被回收。
- **存取速度：**栈的存取速度通常比堆快，因为栈遵循先进后出（LIFO, Last In First Out）的原则，操作简单快速。堆的存取速度相对较慢，因为对象在堆上的分配和回收需要更多的时间，而且垃圾回收机制的运行也会影响性能。
- **存储空间：**栈的空间相对较小，且固定，由操作系统管理。当栈溢出时，通常是因为递归过深或局部变量过大。堆的空间较大，动态扩展，由JVM管理。堆溢出通常是由创建了太多的大对象或未能及时回收不再使用的对象。
- **可见性：**栈中的数据对线程是私有的，每个线程有自己的栈空间。堆中的数据对线程是共享的，所有线程都可以访问堆上的对象。

## 栈中存的到底是指针还是对象？

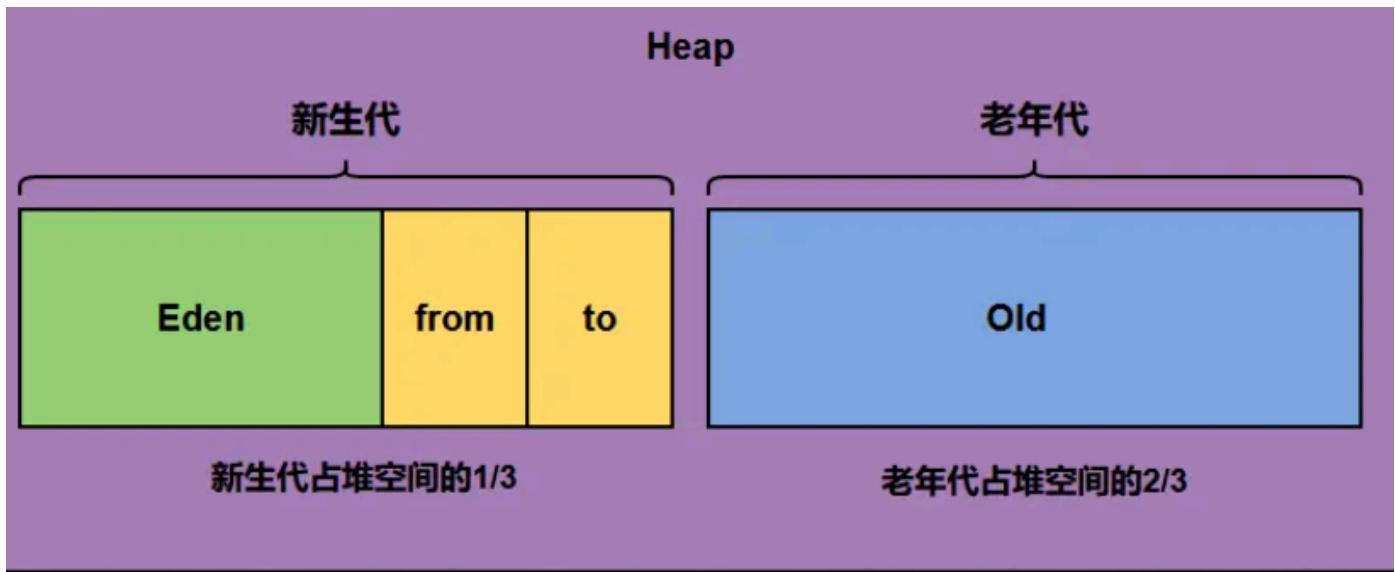
在JVM内存模型中，栈（Stack）主要用于管理线程的局部变量和方法调用的上下文，而堆（Heap）则是用于存储所有类的实例和数组。

当我们在栈中讨论“存储”时，实际上指的是存储基本类型的数据（如int, double等）和对象的引用，而不是对象本身。

这里的关键点是，栈中存储的**不是**对象，而是**对象的引用**。也就是说，当你在方法中声明一个对象，比如 `MyObject obj = new MyObject();`，这里的 `obj` 实际上是一个存储在栈上的引用，指向堆中实际的对象实例。这个引用是一个固定大小的数据（例如在64位系统上是8字节），它指向堆中分配给对象的内存区域。

## 堆分为哪几部分呢？

Java堆（Heap）是Java虚拟机（JVM）中内存管理的一个重要区域，主要用于存放对象实例和数组。随着JVM的发展和不同垃圾收集器的实现，堆的具体划分可能会有所不同，但通常可以分为以下几个部分：



**新生代 (Young Generation)** :新生代分为Eden Space和Survivor Space。在Eden Space中，大多数新创建的对象首先存放在这里。Eden区相对较小，当Eden区满时，会触发一次Minor GC（新生代垃圾回收）。在Survivor Spaces中，通常分为两个相等大小的区域，称为S0（Survivor 0）和S1（Survivor 1）。在每次Minor GC后，存活下来的对象会被移动到其中一个Survivor空间，以继续它们的生命周期。这两个区域轮流充当对象的中转站，帮助区分短暂停活的对象和长期存活的对象。

**老年代 (Old Generation/Tenured Generation)** :存放过一次或多次Minor GC仍存活的对象会被移动到老年代。老年代中的对象生命周期较长，因此Major GC（也称为Full GC，涉及老年代的垃圾回收）发生的频率相对较低，但其执行时间通常比Minor GC长。老年代的空间通常比新生代大，以存储更多的长期存活对象。

**元空间 (Metaspace)** :从Java 8开始，永久代（Permanent Generation）被元空间取代，用于存储类的元数据信息，如类的结构信息（如字段、方法信息等）。元空间并不在Java堆中，而是使用本地内存，这解决了永久代容易出现的内存溢出问题。

**大对象区 (Large Object Space / Humongous Objects)** :在某些JVM实现中（如G1垃圾收集器），为大对象分配了专门的区域，称为大对象区或Humongous Objects区域。大对象是指需要大量连续内存空间的对象，如大数据。这类对象直接分配在老年代，以避免因频繁的年轻代晋升而导致的内存碎片化问题。

## 如果有个大对象一般是在哪个区域？

大对象通常会直接分配到老年代。

新生代主要用于存放生命周期较短的对象，并且其内存空间相对较小。如果将大对象分配到新生代，可能会很快导致新生代空间不足，从而频繁触发 Minor GC。而每次 Minor GC 都需要进行对象的复制和移动操作，这会带来一定的性能开销。将大对象直接分配到老年代，可以减少新生代的内存压力，降低 Minor GC 的频率。

大对象通常需要连续的内存空间，如果在新生代中频繁分配和回收大对象，容易产生内存碎片，导致后续分配大对象时可能因为内存不连续而失败。老年代的空间相对较大，更适合存储大对象，有助于减少内存碎片的产生。

## 程序计数器的作用，为什么是私有的？

Java程序是支持多线程一起运行的，多个线程一起运行的时候cpu会有一个调动器组件给它们分配时间片，比如说会给线程1分给一个时间片，它在时间片内如果它的代码没有执行完，它就会把线程1的状态执行一个暂停，切换到线程2去，执行线程2的代码，等线程2的代码执行到了一定程度，线程2的时间片用完了，再切换回来，再继续执行线程1剩余部分的代码。

我们考虑一下，如果在线程切换的过程中，下一条指令执行到哪里了，是不是还是会用到我们的程序计数器啊。每个线程都有自己的程序计数器，因为它们各自执行的代码的指令地址是不一样的呀，所以每个线程都应该有自己的程序计数器。

## 方法区中的方法的执行过程？

当程序中通过对象或类直接调用某个方法时，主要包括以下几个步骤：

- **解析方法调用**：JVM会根据方法的符号引用找到实际的方法地址（如果之前没有解析过的话）。
- **栈帧创建**：在调用一个方法前，JVM会在当前线程的Java虚拟机栈中为该方法分配一个新的栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。
- **执行方法**：执行方法内的字节码指令，涉及的操作可能包括局部变量的读写、操作数栈的操作、跳转控制、对象创建、方法调用等。
- **返回处理**：方法执行完毕后，可能会返回一个结果给调用者，并清理当前栈帧，恢复调用者的执行环境。

## 方法区中还有哪些东西？

《深入理解Java虚拟机》书中对方法区（Method Area）存储内容描述如下：它用于存储已被虚拟机加载的**类型信息、常量、静态变量、即时编译器编译后的代码缓存等**。

- 类信息：包括类的结构信息、类的访问修饰符、父类与接口等信息。
- 常量池：存储类和接口中的常量，包括字面值常量、符号引用，以及运行时常量池。
- 静态变量：存储类的静态变量，这些变量在类初始化的时候被赋值。
- 方法字节码：存储类的方法字节码，即编译后的代码。
- 符号引用：存储类和方法的符号引用，是一种直接引用不同于直接引用的引用类型。
- 运行时常量池：存储着在类文件中的常量池数据，在类加载后在方法区生成该运行时常量池。
- 常量池缓存：用于提升类加载的效率，将常用的常量缓存起来方便使用。

## String保存在哪里呢？

String 保存在字符串常量池中，不同于其他对象，它的值是不可变的，且可以被多个引用共享。

## String s = new String ("abc") 执行过程中分别对应哪些内存区域？

首先，我们看到这个代码中有一个new关键字，我们知道new指令是创建一个类的实例对象并完成加载初始化的，因此这个字符串对象是在[运行期](#)才能确定的，创建的字符串对象是在[堆内存上](#)。

其次，在String的构造方法中传递了一个字符串abc，由于这里的abc是被final修饰的属性，所以它是一个字符串常量。在首次构建这个对象时，JVM拿字面量"abc"去字符串常量池试图获取其对应String对象的引用。于是在堆中创建了一个"abc"的String对象，并将其引用保存到字符串常量池中，然后返回；

所以，[如果abc这个字符串常量不存在，则创建两个对象，分别是abc这个字符串常量，以及new String这个实例对象。如果abc这字符串常量存在，则只会创建一个对象。](#)

## 引用类型有哪些？有什么区别？

引用类型主要分为强软弱虚四种：

- 强引用指的就是代码中普遍存在的赋值方式，比如A a = new A()这种。强引用关联的对象，永远不会被GC回收。
- 软引用可以用SoftReference来描述，指的是那些有用但是不是必须要的对象。系统在发生内存溢出前会对这类引用的对象进行回收。
- 弱引用可以用WeakReference来描述，他的强度比软引用更低一点，弱引用的对象下一次GC的时候一定会被回收，而不管内存是否足够。
- 虚引用也被称作幻影引用，是最弱的引用关系，可以用PhantomReference来描述，他必须和ReferenceQueue一起使用，同样的当发生GC的时候，虚引用也会被回收。可以用虚引用来管理堆外内存。

## 弱引用了解吗？举例说明在哪里可以用？

Java中的弱引用是一种引用类型，它不会阻止一个对象被垃圾回收。

在Java中，弱引用是通过 `Java.lang.ref.WeakReference` 类实现的。弱引用的一个主要用途是创建非强制性的对象引用，这些引用可以在内存压力大时被垃圾回收器清理，从而避免内存泄露。

弱引用的使用场景：

- **缓存系统**：弱引用常用于实现缓存，特别是当希望缓存项能够在内存压力下自动释放时。如果缓存的大小不受控制，可能会导致内存溢出。使用弱引用来维护缓存，可以让JVM在需要更多内存时自动清理这些缓存对象。
- **对象池**：在对象池中，弱引用可以用来管理那些暂时不使用的对象。当对象不再被强引用时，它们可以被垃圾回收，释放内存。
- **避免内存泄露**：当一个对象不应该被长期引用时，使用弱引用可以防止该对象被意外地保留，从而避免潜在的内存泄露。

示例代码：

假设我们有一个缓存系统，我们使用弱引用来维护缓存中的对象：

```
import Java.lang.ref.WeakReference;
import Java.util.HashMap;
import Java.util.Map;

public class CacheExample {

    private Map<String, WeakReference<MyHeavyObject>> cache = new HashMap<>();

    public MyHeavyObject get(String key) {
        WeakReference<MyHeavyObject> ref = cache.get(key);
        if (ref != null) {
            return ref.get();
        } else {
            MyHeavyObject obj = new MyHeavyObject();
            cache.put(key, new WeakReference<>(obj));
            return obj;
        }
    }

    // 假设MyHeavyObject是一个占用大量内存的对象
    private static class MyHeavyObject {
        private byte[] largeData = new byte[1024 * 1024 * 10]; // 10MB data
    }
}
```

在这个例子中，使用 `WeakReference` 来存储 `MyHeavyObject` 实例，当内存压力增大时，垃圾回收器可以自由地回收这些对象，而不会影响缓存的正常运行。

如果一个对象被垃圾回收，下次尝试从缓存中获取时，`get()` 方法会返回 `null`，这时我们可以重新创建对象并将其放入缓存中。因此，使用弱引用时要注意，一旦对象被垃圾回收，通过弱引用获取的对象可能会变为 `null`，因此在使用前通常需要检查这一点。

## 内存泄漏和内存溢出的理解？

**内存泄露：**内存泄露是指程序在运行过程中不再使用的对象仍然被引用，而无法被垃圾收集器回收，从而导致可用内存逐渐减少。虽然在Java中，垃圾回收机制会自动回收不再使用的对象，但如果有对象仍被不再使用的引用持有，垃圾收集器无法回收这些内存，最终可能导致程序的内存使用不断增加。

内存泄露常见原因：

- **静态集合：**使用静态数据结构（如 `HashMap` 或 `ArrayList`）存储对象，且未清理。
- **事件监听：**未取消对事件源的监听，导致对象持续被引用。
- **线程：**未停止的线程可能持有对象引用，无法被回收。

**内存溢出：**内存溢出是指Java虚拟机（JVM）在申请内存时，无法找到足够的内存，最终引发 `OutOfMemoryError`。这通常发生在堆内存不足以存放新创建的对象时。

内存溢出常见原因：

- **大量对象创建：**程序中不断创建大量对象，超出JVM堆的限制。
- **持久引用：**大型数据结构（如缓存、集合等）长时间持有对象引用，导致内存累积。
- **递归调用：**深度递归导致栈溢出。

## jvm 内存结构有哪几种内存溢出的情况？

- **堆内存溢出：**当出现`Java.lang.OutOfMemoryError:Java heap space`异常时，就是堆内存溢出了。原因是代码中可能存在大对象分配，或者发生了内存泄露，导致在多次GC之后，还是无法找到一块足够大的内存容纳当前对象。
- **栈溢出：**如果我们写一段程序不断的进行递归调用，而且没有退出条件，就会导致不断地进行压栈。类似这种情况，JVM 实际会抛出 `StackOverflowError`；当然，如果 JVM 试图去扩展栈空间的时候失败，则会抛出 `OutOfMemoryError`。
- **元空间溢出：**元空间的溢出，系统会抛出`Java.lang.OutOfMemoryError: Metaspace`。出现这个异常的问题的原因是系统的代码非常多或引用的第三方包非常多或者通过动态代码生成类加载等方法，导致元空间的内存占用很大。
- **直接内存内存溢出：**在使用`ByteBuffer`中的`allocateDirect()`的时候会用到，很多JavaNIO(像netty)的框架中被封装为其他的方法，出现该问题时会抛出`Java.lang.OutOfMemoryError: Direct buffer memory`异常。

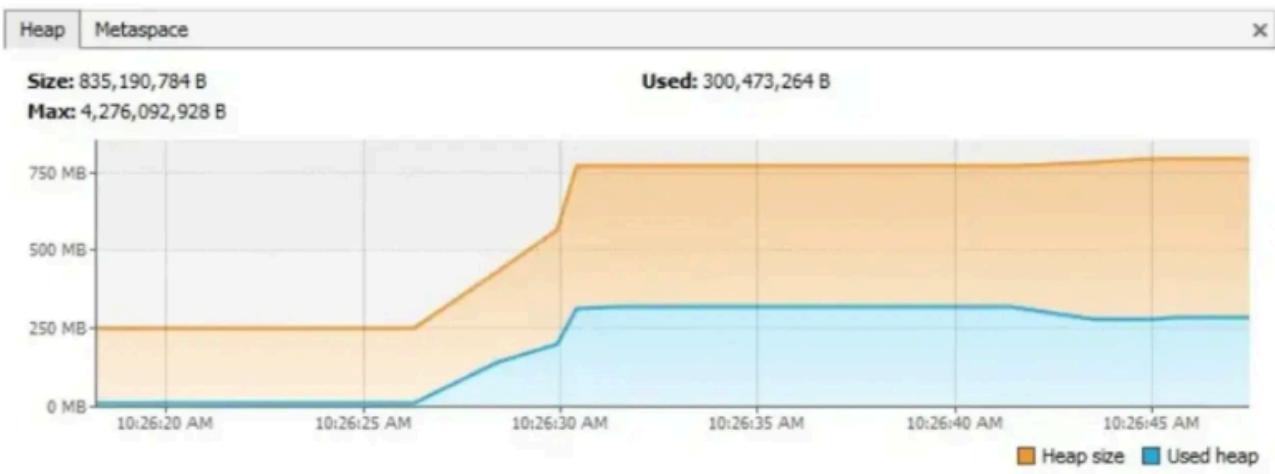
# 有具体的内存泄漏和内存溢出的例子么请举例及解决方案?

## 1、静态属性导致内存泄露

会导致内存泄露的一种情况就是大量使用static静态变量。在Java中，静态属性的生命周期通常伴随着应用整个生命周期（除非ClassLoader符合垃圾回收的条件）。下面来看一个具体的会导致内存泄露的实例：

```
public class StaticTest {  
    public static List<Double> list = new ArrayList<>();  
    public void populateList() {  
        for (int i = 0; i < 10000000; i++) {  
            list.add(Math.random());  
        }  
        Log.info("Debug Point 2");  
    }  
    public static void main(String[] args) {  
        Log.info("Debug Point 1");  
        new StaticTest().populateList();  
        Log.info("Debug Point 3");  
    }  
}
```

如果监控内存堆内存的变化，会发现在打印Point1和Point2之间，堆内存会有一个明显的增长趋势图。但当执行完populateList方法之后，对堆内存并没有被垃圾回收器进行回收。



但针对上述程序，如果将定义list的变量前的static关键字去掉，再次执行程序，会发现内存发生了具体的变化。VisualVM监控信息如下图：



对比两个图可以看出，程序执行的前半部分内存使用情况都一样，但当执行完 populateList 方法之后，后者不再有引用指向对应的数据，垃圾回收器便进行了回收操作。因此，我们要十分留意 static 的变量，如果集合或大量的对象定义为 static 的，它们会停留在整个应用程序的生命周期当中。而它们所占用的内存空间，本可以用于其他地方。

那么如何优化呢？第一，进来减少静态变量；第二，如果使用单例，尽量采用懒加载。

## 2、未关闭的资源

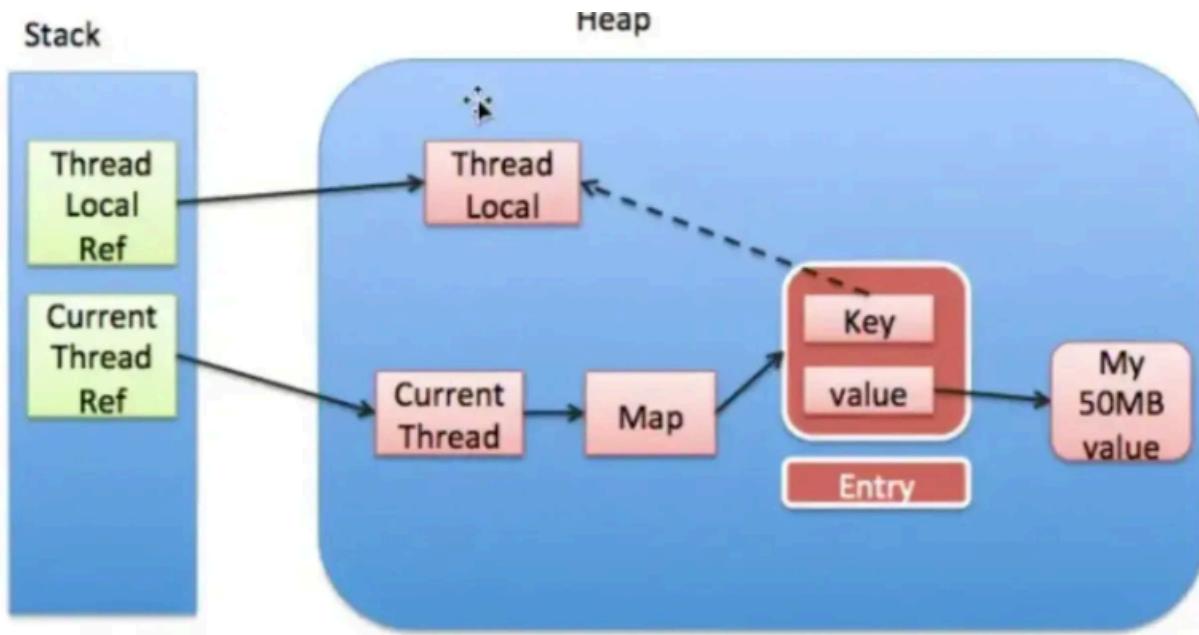
无论什么时候当我们创建一个连接或打开一个流，JVM 都会分配内存给这些资源。比如，数据库链接、输入流和 session 对象。

忘记关闭这些资源，会阻塞内存，从而导致 GC 无法进行清理。特别是当程序发生异常时，没有在 finally 中进行资源关闭的情况。这些未正常关闭的连接，如果不进行处理，轻则影响程序性能，重则导致 OutOfMemoryError 异常发生。

如果进行处理呢？第一，始终记得在 finally 中进行资源的关闭；第二，关闭连接的自身代码不能发生异常；第三，Java 7 以上版本可使用 try-with-resources 代码方式进行资源关闭。

## 3、使用 ThreadLocal

ThreadLocal 提供了线程本地变量，它可以保证访问到的变量属于当前线程，每个线程都保存有一个变量副本，每个线程的变量都不同。ThreadLocal 相当于提供了一种线程隔离，将变量与线程相绑定，从而实现线程安全的特性。



ThreadLocal的实现中，每个Thread维护一个ThreadLocalMap映射表，key是ThreadLocal实例本身，value是真正需要存储的Object。

ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用来引用它，那么系统GC时，这个ThreadLocal势必会被回收，这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value。

如果当前线程迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链： Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value永远无法回收，造成内存泄漏。

如何解决此问题？

- 第一，使用ThreadLocal提供的remove方法，可对当前线程中的value值进行移除；
- 第二，不要使用ThreadLocal.set(null) 的方式清除value，它实际上并没有清除值，而是查找与当前线程关联的Map并将键值对分别设置为当前线程和null。
- 第三，最好将ThreadLocal视为需要在finally块中关闭的资源，以确保即使在发生异常的情况下也始终关闭该资源。

```

try {
    threadLocal.set(System.nanoTime());
    //... further processing
} finally {
    threadLocal.remove();
}

```

java

2. 类初始化与加载：

## 创建对象的过程？

Java对象创建过程



在Java中创建对象的过程包括以下几个步骤：

1. **类加载检查**：虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的**符号引用**，并且检查这个符号引用代表的类是否已被**加载过、解析和初始化**过。如果没有，那必须先执行相应的**类加载过程**。
2. **分配内存**：在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的**内存大小**在**类加载**完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。
3. **初始化零值**：内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。
4. **进行必要设置，比如对象头**：初始化零值完成之后，虚拟机要对对象进行**必要的设置**，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。这些信息存放在**对象头**中。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。
5. **执行 init 方法**：在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚刚开始——构造函数，即class文件中的方法还没有执行，所有的字段都还为零，**对象需要的其他资源和状态信息**还没有按照预定的意图构造好。所以一般来说，执行 new 指令之后会接着执行方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全被构造出来。

## 对象的生命周期

对象的生命周期包括创建、使用和销毁三个阶段：

- **创建**：对象通过关键字new在堆内存中被实例化，构造函数被调用，对象的内存空间被分配。
- **使用**：对象被引用并执行相应的操作，可以通过引用访问对象的属性和方法，在程序运行过程中被不断使用。
- **销毁**：当对象不再被引用时，通过垃圾回收机制自动回收对象所占用的内存空间。垃圾回收器会在适当的时候检测并回收不再被引用的对象，释放对象占用的内存空间，完成对象的销毁过程。

## 类加载器有哪些？

启动类加载器  
Bootstrap ClassLoader



扩展类加载器  
Extension ClassLoader



应用程序类加载器  
Application ClassLoader



用户自定义类加载器  
User ClassLoader

用户自定义类加载器  
User ClassLoader

- **启动类加载器 (Bootstrap Class Loader)**: 这是最顶层的类加载器，负责加载Java的核心库（如位于jre/lib/rt.jar中的类），它是用C++编写的，是JVM的一部分。启动类加载器无法被Java程序直接引用。
- **扩展类加载器 (Extension Class Loader)**: 它是Java语言实现的，继承自ClassLoader类，负责加载Java扩展目录（jre/lib/ext或由系统变量Java.ext.dirs指定的目录）下的jar包和类库。扩展类加载器由启动类加载器加载，并且父加载器就是启动类加载器。
- **系统类加载器 (System Class Loader) / 应用程序类加载器 (Application Class Loader)**: 这也是Java语言实现的，负责加载用户类路径（ClassPath）上的指定类库，是我们平时编写Java程序时默认使用的类加载器。系统类加载器的父加载器是扩展类加载器。它可以通过ClassLoader.getSystemClassLoader()方法获取到。
- **自定义类加载器 (Custom Class Loader)**: 开发者可以根据需求定制类的加载方式，比如从网络加载class文件、数据库、甚至是加密的文件中加载类等。自定义类加载器可以用来扩展Java应用程序的灵活性和安全性，是Java动态性的一个重要体现。

这些类加载器之间的关系形成了双亲委派模型，其核心思想是当一个类加载器收到类加载的请求时，首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中。

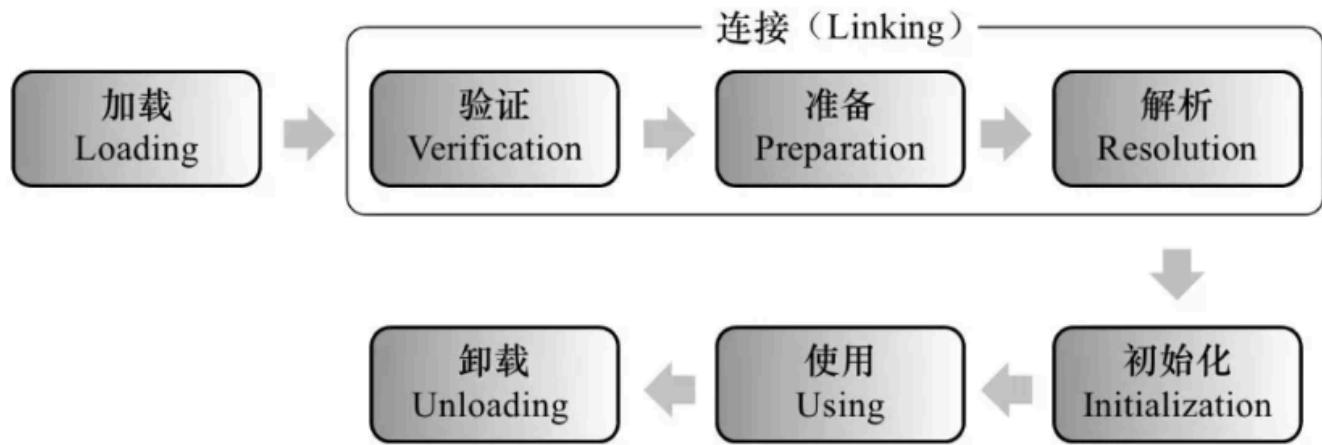
只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

## 双亲委派模型的作用

- **保证类的唯一性**: 通过委托机制，确保了所有加载请求都会传递到启动类加载器，避免了不同类加载器重复加载相同类的情况，保证了Java核心类库的统一性，也防止了用户自定义类覆盖核心类库的可能。
- **保证安全性**: 由于Java核心库被启动类加载器加载，而启动类加载器只加载信任的类路径中的类，这样可以防止不可信的类假冒核心类，增强了系统的安全性。例如，恶意代码无法自定义一个Java.lang.System类并加载到JVM中，因为这个请求会被委托给启动类加载器，而启动类加载器只会加载标准的Java库中的类。
- **支持隔离和层次划分**: 双亲委派模型支持不同层次的类加载器服务于不同的类加载需求，如应用程序类加载器加载用户代码，扩展类加载器加载扩展框架，启动类加载器加载核心库。这种层次化的划分有助于实现沙箱安全机制，保证了各个层级类加载器的职责清晰，也便于维护和扩展。
- **简化了加载流程**: 通过委派，大部分类能够被正确的类加载器加载，减少了每个加载器需要处理的类的数量，简化了类的加载过程，提高了加载效率。

## 讲一下类加载过程？

类从被加载到虚拟机内存开始，到卸载出内存为止，它的整个生命周期包括以下 7 个阶段：



- **加载**: 通过类的全限定名（包名 + 类名），获取到该类的.class文件的二进制字节流，将二进制字节流所代表的静态存储结构，转化为方法区运行时的数据结构，在内存中生成一个代表该类的Java.lang.Class对象，作为方法区这个类的各种数据的访问入口
- **连接**: 验证、准备、解析 3 个阶段统称为连接。
  - **验证**: 确保class文件中的字节流包含的信息，符合当前虚拟机的要求，保证这个被加载的class类的正确性，不会危害到虚拟机的安全。验证阶段大致会完成以下四个阶段的检验动作：文件格式校验、元数据验证、字节码验证、符号引用验证
  - **准备**: 为类中的静态字段分配内存，并设置默认的初始值，比如int类型初始值是0。被final修饰的static字段不会设置，因为final在编译的时候就分配了
  - **解析**: 解析阶段是虚拟机将常量池的「符号引用」直接替换为「直接引用」的过程。符号引用是以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用的时候可以无歧义地定位到目标即可。直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄，直接引用是和虚拟机实现的内存布局相关的。如果有了直接引用，那引用的目标必定已经存在在内存中了。
- **初始化**: 初始化是整个类加载过程的最后一个阶段，初始化阶段简单来说就是执行类的构造器方法（`new`），要注意的是这里的构造器方法并不是开发者写的，而是编译器自动生成的。
- **使用**: 使用类或者创建对象
- **卸载**: 如果有下面的情况，类就会被卸载：
  1. 该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。
  2. 加载该类的ClassLoader已经被回收。
  3. 类对应的Java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法。

## 讲一下类的加载和双亲委派原则

我们把 Java 的类加载过程分为三个主要步骤：加载、链接、初始化。

首先是加载阶段（Loading），它是 Java 将字节码数据从不同的数据源读取到 JVM 中，并映射为 JVM 认可的数据结构（Class 对象），这里的数据源可能是各种各样的形态，如 jar 文件、class 文件，甚至是网络数据源等；如果输入数据不是 ClassFile 的结构，则会抛出 ClassFormatError。

加载阶段是用户参与的阶段，我们可以自定义类加载器，去实现自己的类加载过程。

第二阶段是链接（Linking），这是核心的步骤，简单说是把原始的类定义信息平滑地转化入 JVM 运行的过程中。这里可进一步细分为三个步骤：

- 验证（Verification），这是虚拟机安全的重要保障，JVM 需要核验字节信息是符合 Java 虚拟机规范的，否则就被认为是 VerifyError，这样就防止了恶意信息或者不合规的信息危害 JVM 的运行，验证阶段有可能触发更多 class 的加载。
- 准备（Preparation），创建类或接口中的静态变量，并初始化静态变量的初始值。但这里的“初始化”和下面的显式初始化阶段是有区别的，侧重点在于分配所需要的内存空间，不会去执行更进一步的 JVM 指令。
- 解析（Resolution），在这一步会将常量池中的符号引用（symbolic reference）替换为直接引用。

最后是初始化阶段（initialization），这一步真正去执行类初始化的代码逻辑，包括静态字段赋值的动作，以及执行类定义中的静态初始化块内的逻辑，编译器在编译阶段就会把这部分逻辑整理好，父类型的初始化逻辑优先于当前类型的逻辑。

再来谈谈双亲委派模型，简单说就是当类加载器（Class-Loader）试图加载某个类型的时候，除非父加载器找不到相应类型，否则尽量将这个任务代理给当前加载器的父加载器去做。使用委派模型的目的是避免重复加载 Java 类型。

### 3. 垃圾回收：

## 什么是Java里的垃圾回收？如何触发垃圾回收？

垃圾回收（Garbage Collection, GC）是自动管理内存的一种机制，它负责自动释放不再被程序引用的对象所占用的内存，这种机制减少了内存泄漏和内存管理错误的可能性。垃圾回收可以通过多种方式触发，具体如下：

- **内存不足时**：当JVM检测到堆内存不足，无法为新的对象分配内存时，会自动触发垃圾回收。
- **手动请求**：虽然垃圾回收是自动的，开发者可以通过调用 `System.gc()` 或 `Runtime.getRuntime().gc()` 建议 JVM 进行垃圾回收。不过这只是一个建议，并不能保证立即执行。
- **JVM参数**：启动 Java 应用时可以通过 JVM 参数来调整垃圾回收的行为，比如：`-Xmx`（最大堆大小）、`-Xms`（初始堆大小）等。
- **对象数量或内存使用达到阈值**：垃圾收集器内部实现了一些策略，以监控对象的创建和内存使用，达到某个阈值时触发垃圾回收。

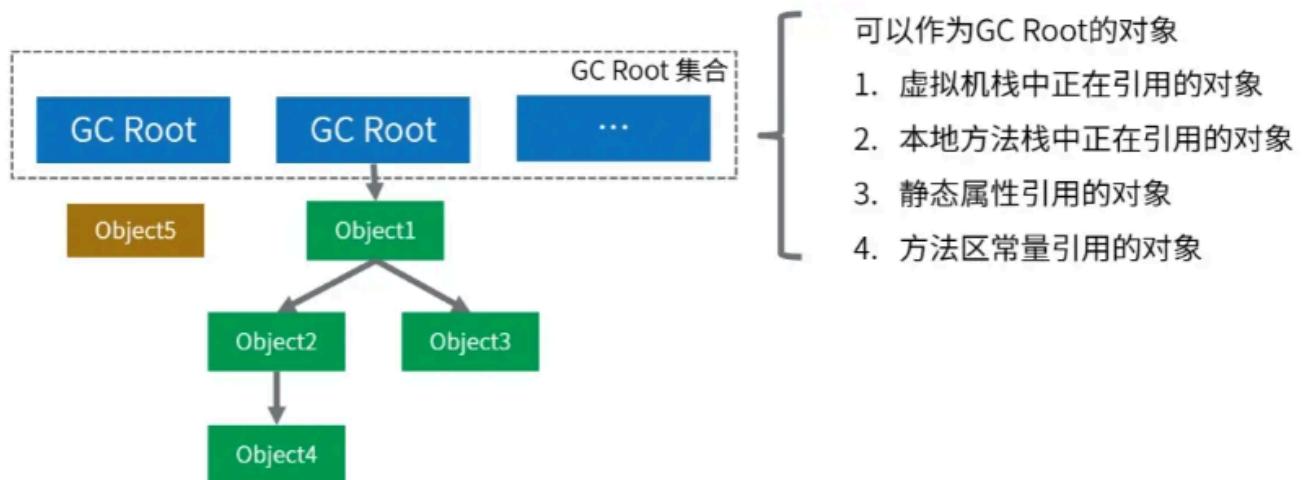
## 判断垃圾的方法有哪些？

在Java中，判断对象是否为垃圾（即不再被使用，可以被垃圾回收器回收）主要依据两种主流的垃圾回收算法来实现：**引用计数法和可达性分析算法**。

### 引用计数法（Reference Counting）

- **原理**：为每个对象分配一个引用计数器，每当有一个地方引用它时，计数器加1；当引用失效时，计数器减1。当计数器为0时，表示对象不再被任何变量引用，可以被回收。
- **缺点**：不能解决循环引用的问题，即两个对象相互引用，但不再被其他任何对象引用，这时引用计数器不会为0，导致对象无法被回收。

## 可达性分析算法 (Reachability Analysis)



Java虚拟机主要采用此算法来判断对象是否为垃圾。

- **原理：**从一组称为GC Roots（垃圾收集根）的对象出发，向下追溯它们引用的对象，以及这些对象引用的其他对象，以此类推。如果一个对象到GC Roots没有任何引用链相连（即从GC Roots到这个对象不可达），那么这个对象就被认为是不可达的，可以被回收。GC Roots对象包括：虚拟机栈（栈帧中的本地变量表）中引用的对象、方法区中类静态属性引用的对象、本地方法栈中JNI（Java Native Interface）引用的对象、活跃线程的引用等。

## 垃圾回收算法是什么，是为了解决了什么问题？

JVM有垃圾回收机制的原因是为了解决内存管理的问题。在传统的编程语言中，开发人员需要手动分配和释放内存，这可能导致内存泄漏、内存溢出等问题。而Java作为一种高级语言，旨在提供更简单、更安全的编程环境，因此引入了垃圾回收机制来自动管理内存。

垃圾回收机制的主要目标是**自动检测和回收**不再使用的对象，从而释放它们所占用的内存空间。这样可以避免内存泄漏（一些对象被分配了内存却无法被释放，导致内存资源的浪费）。同时，垃圾回收机制还可以防止内存溢出（即程序需要的内存超过了可用内存的情况）。

通过垃圾回收机制，JVM可以在程序运行时自动识别和清理不再使用的对象，使得开发人员无需手动管理内存。这样可以提高开发效率、减少错误，并且使程序更加可靠和稳定。

## 垃圾回收算法有哪些？

- **标记-清除算法**：标记-清除算法分为“标记”和“清除”两个阶段，首先通过可达性分析，标记出所有需要回收的对象，然后统一回收所有被标记的对象。标记-清除算法有两个缺陷，一个是效率问题，标记和清除的过程效率都不高，另外一个是，清除结束后会造成大量的碎片空间。有可能会造成在申请大块内存的时候因为没有足够的连续空间导致再次 GC。

- 复制算法**: 为了解决碎片空间的问题, 出现了“**复制算法**”。**复制算法**的原理是, 将内存分成两块, 每次申请内存时都使用其中的一块, 当内存不够时, 将这一块内存中所有存活的复制到另一块上。然后将然后再把已使用的内存整个清理掉。**复制算法**解决了空间碎片的问题。但是也带来了新的问题。因为每次在申请内存时, 都只能使用一半的内存空间。内存利用率严重不足。
- 标记-整理算法**: **复制算法**在 GC 之后存活对象较少的情况下效率比较高, 但如果存活对象比较多时, 会执行较多的复制操作, 效率就会下降。而老年代的对象在 GC 之后的存活率就比较高, 所以就有人提出了“**标记-整理算法**”。**标记-整理算法**的“**标记**”过程与“**标记-清除算法**”的标记过程一致, 但标记之后不会直接清理。而是将所有存活对象都移动到内存的一端。移动结束后直接清理掉剩余部分。
- 分代回收算法**: 分代收集是将内存划分成了新生代和老年代。分配的依据是对象的生存周期, 或者说经历过的 GC 次数。对象创建时, 一般在新生代申请内存, 当经历一次 GC 之后如果还存活, 那么对象的年龄 +1。当年龄超过一定值(默认是 15, 可以通过参数 -XX:MaxTenuringThreshold 来设定)后, 如果对象还存活, 那么该对象会进入老年代。

## 垃圾回收器有哪些?

| 垃圾收集器                      | 类型                         | 作用域             | 使用算法          | 特点     | 适用场景                         |
|----------------------------|----------------------------|-----------------|---------------|--------|------------------------------|
| Serial                     | 串行回收                       | 新生代             | 复制算法          | 响应速度优先 | 适用于单核 CPU环境下的 Client 模式      |
| Serial Old                 | 串行回收                       | 老年代             | 标记-压缩算法       | 响应速度优先 | 适用于单核 CPU环境下的 Client 模式      |
| ParNew                     | 并行回收                       | 新生代             | 复制算法          | 响应速度优先 | 多核 CPU环境中 Server模式下与 CMS配合使用 |
| Parallel Scavenge          | 并行回收                       | 新生代             | 复制算法          | 吞吐量优先  | 适用于后台运算, 而交互少的场景             |
| Parallel Old               | 并行回收                       | 老年代             | 标记-压缩算法       | 吞吐量优先  | 适用于后台运算, 而交互少的场景             |
| CMS(Concurrent Mark-Sweep) | 并发回收                       | 老年代             | 标记-清除算法       | 响应速度优先 | 适用于B/S业务, 也就是交互多的场景          |
| G1(Garbage-First)          | 并发,并行回收(此收集器后期优化后并行方式同时存在) | 新生代& 老年代(整堆收集器) | 复制算法& 标记-压缩算法 | 响应速度优先 | 面向服务端的应用                     |

- Serial收集器 (复制算法): 新生代单线程收集器，标记和清理都是单线程，优点是简单高效；
- ParNew收集器 (复制算法): 新生代收并行集器，实际上是Serial收集器的多线程版本，在多核CPU环境下有着比Serial更好的表现；
- Parallel Scavenge收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；
- Serial Old收集器 (标记-整理算法): 老年代单线程收集器，Serial收集器的老年代版本；
- Parallel Old收集器 (标记-整理算法): 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器的老年代版本；
- CMS(Concurrent Mark Sweep)收集器 (标记-清除算法): 老年代并行收集器，以获取最短回收停顿时间为为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。
- G1(Garbage First)收集器 (标记-整理算法): Java堆并行收集器，G1收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代

## 标记清除算法的缺点是什么？

主要缺点有两个：

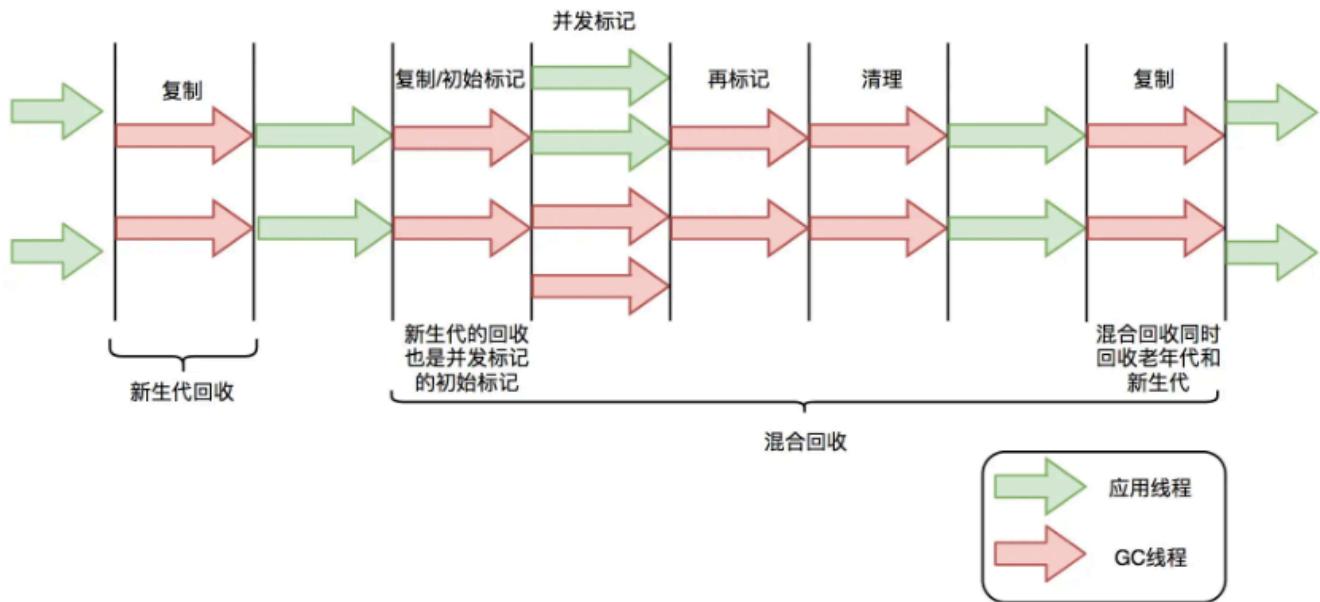
- 一个是效率问题，标记和清除过程的效率都不高；
- 另外一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

## 垃圾回收算法哪些阶段会stop the world?

标记-复制算法应用在CMS新生代（ParNew是CMS默认的新生代垃圾回收器）和G1垃圾回收器中。标记-复制算法可以分为三个阶段：

- 标记阶段，即从GC Roots集合开始，标记活跃对象；
- 转移阶段，即把活跃对象复制到新的内存地址上；
- 重定位阶段，因为转移导致对象的地址发生了变化，在重定位阶段，所有指向对象旧地址的指针都要调整到对象新的地址上。

下面以G1为例，通过G1中标记-复制算法过程（G1的Young GC和Mixed GC均采用该算法），分析G1停顿耗时的主要瓶颈。G1垃圾回收周期如下图所示：



G1的混合回收过程可以分为标记阶段、清理阶段和复制阶段。

## 标记阶段停顿分析

- 初始标记阶段：初始标记阶段是指从GC Roots出发标记全部直接子节点的过程，该阶段是STW的。由于GC Roots数量不多，通常该阶段耗时非常短。
- 并发标记阶段：并发标记阶段是指从GC Roots开始对堆中对象进行可达性分析，找出存活对象。该阶段是并发的，即应用线程和GC线程可以同时活动。并发标记耗时相对长很多，但因为不是STW，所以我们不太关心该阶段耗时的长短。
- 再标记阶段：重新标记那些在并发标记阶段发生变化的对象。该阶段是STW的。

## 清理阶段停顿分析

- 清理阶段清点出有存活对象的分区和没有存活对象的分区，该阶段不会清理垃圾对象，也不会执行存活对象的复制。该阶段是STW的。

## 复制阶段停顿分析

- 复制算法中的转移阶段需要分配新内存和复制对象的成员变量。转移阶段是STW的，其中内存分配通常耗时非常短，但对象成员变量的复制耗时有可能较长，这是因为复制耗时与存活对象数量与对象复杂度成正比。对象越复杂，复制耗时越长。

四个STW过程中，初始标记因为只标记GC Roots，耗时较短。再标记因为对象数少，耗时也较短。清理阶段因为内存分区数量少，耗时也较短。转移阶段要处理所有存活的对象，耗时会较长。

因此，G1停顿时间的瓶颈主要是标记-复制中的转移阶段STW。

## minorGC、majorGC、fullGC的区别，什么场景触发full GC

在Java中，垃圾回收机制是自动管理内存的重要组成部分。根据其作用范围和触发条件的不同，可以将GC分为三种类型：Minor GC（也称为Young GC）、Major GC（有时也称为Old GC）以及Full GC。以下是这三种GC的区别和触发场景：

### Minor GC (Young GC)

- **作用范围：**只针对年轻代进行回收，包括Eden区和两个Survivor区（S0和S1）。
- **触发条件：**当Eden区空间不足时，JVM会触发一次Minor GC，将Eden区和一个Survivor区中的存活对象移动到另一个Survivor区或老年代（Old Generation）。
- **特点：**通常发生得非常频繁，因为年轻代中对象的生命周期较短，回收效率高，暂停时间相对较短。

### Major GC

- **作用范围：**主要针对老年代进行回收，但不一定只回收老年代。
- **触发条件：**当老年代空间不足时，或者系统检测到年轻代对象晋升到老年代的速度过快，可能会触发Major GC。
- **特点：**相比Minor GC，Major GC发生的频率较低，但每次回收可能需要更长的时间，因为老年代中的对象存活率较高。

### Full GC

- **作用范围：**对整个堆内存（包括年轻代、老年代以及永久代/元空间）进行回收。
- **触发条件：**
  - 直接调用 `System.gc()` 或 `Runtime.getRuntime().gc()` 方法时，虽然不能保证立即执行，但JVM会尝试执行Full GC。
  - Minor GC（新生代垃圾回收）时，如果存活的对象无法全部放入老年代，或者老年代空间不足以容纳存活的对象，则会触发Full GC，对整个堆内存进行回收。
  - 当永久代（Java 8之前的版本）或元空间（Java 8及以后的版本）空间不足时。
- **特点：**Full GC是最昂贵的操作，因为它需要停止所有的工作线程（Stop The World），遍历整个堆内存来查找和回收不再使用的对象，因此应尽量减少Full GC的触发。

# 垃圾回收器 CMS 和 G1的区别？

## 区别一：使用的范围不一样：

- CMS收集器是老年代的收集器，可以配合新生代的Serial和ParNew收集器一起使用
- G1收集器收集范围是老年代和新生代。不需要结合其他收集器使用

## 区别二：STW的时间：

- CMS收集器以最小的停顿时间为为目标的收集器。
- G1收集器可预测[垃圾回收](#)的停顿时间（建立可预测的停顿时间模型）

## 区别三：垃圾碎片

- CMS收集器是使用“标记-清除”算法进行的垃圾回收，容易产生内存碎片
- G1收集器使用的是“标记-整理”算法，进行了空间整合，没有内存空间碎片。

## 区别四：垃圾回收的过程不一样

#### 区别四：垃圾回收的过程不一样

## CMS收集器

1. 初始标记
2. 并发标记
3. 重新标记
4. 并发清楚

注意这两个收集器第四阶段得不同

## G1收集器

1. 初始标记
2. 并发标记
3. 最终标记
4. 筛选回收

#### 区别五: CMS会产生浮动垃圾

- CMS产生浮动垃圾过多时会退化为serial old，效率低，因为在上图的第四阶段，CMS清除垃圾时是并发清除的，这个时候，垃圾回收线程和用户线程同时工作会产生浮动垃圾，也就意味着CMS垃圾回收器必须预留一部分内存空间用于存放浮动垃圾
- 而G1没有浮动垃圾，G1的筛选回收是多个垃圾回收线程并行gc的，没有浮动垃圾的回收，在执行‘并发清理’步骤时，用户线程也会同时产生一部分可回收对象，但是这部分可回收对象只能在下次执行清理时才会被回收。如果在清理过程中预留给用户线程的内存不足就会出现‘Concurrent Mode Failure’，一旦出现此错误时便会切换到SerialOld收集方式。

## 什么情况下使用CMS，什么情况使用G1?

CMS适用场景：

- **低延迟需求**：适用于对停顿时间要求敏感的应用程序。
- **老生代收集**：主要针对老年代的垃圾回收。
- **碎片化管理**：容易出现内存碎片，可能需要定期进行Full GC来压缩内存空间。

G1适用场景：

- **大堆内存**：适用于需要管理大内存堆的场景，能够有效处理数GB以上的堆内存。
- **对内存碎片敏感**：G1通过紧凑整理来减少内存碎片，降低了碎片化对性能的影响。
- **比较平衡的性能**：G1在提供较低停顿时间的同时，也保持了相对较高的吞吐量。

## G1回收器的特色是什么？

**G1 的特点：**

- G1最大的特点是引入分区的思路，弱化了分代的概念。
- 合理利用垃圾收集各个周期的资源，解决了其他收集器、甚至 CMS 的众多缺陷

**G1 相比较 CMS 的改进：**

- **算法**：G1 基于标记--整理算法，不会产生空间碎片，在分配大对象时，不会因无法得到连续的空间，而提前触发一次 FULL GC。
- **停顿时间可控**：G1可以通过设置预期停顿时间（Pause Time）来控制垃圾收集时间避免应用雪崩现象。
- **并行与并发**：G1 能更充分的利用 CPU 多核环境下的硬件优势，来缩短 stop the world 的停顿时间。

## GC只会对堆进行GC吗？

JVM 的垃圾回收器不仅仅会对堆进行垃圾回收，它还会对方法区进行垃圾回收。

1. **堆 (Heap)**：堆是用于存储对象实例的内存区域。大部分的垃圾回收工作都发生在堆上，因为大多数对象都会被分配在堆上，而垃圾回收的重点通常也是回收堆中不再被引用的对象，以释放内存空间。
2. **方法区 (Method Area)**：方法区是用于存储类信息、常量、静态变量等数据的区域。虽然方法区中的垃圾回收与堆有所不同，但是同样存在对不再需要的常量、无用的类信息等进行清理的过程。

# 二.面向对象编程(类与对象:(面向对象编程:OOP))

## (一).面向对象编程(基础部分)

### 1.1 创建示例:(类比C++)

```
class Cat{  
    String name;  
    int age;  
    String color;  
    double weight;  
} //自定义类  
new Cat(); //没啥用,但是可以作为一次性的匿名对象来使用其成员函数(方法)  
Cat cat1 = new Cat(); //声明cat1,并将自建类对象赋值给cat1  
//两者合一即为此处的直接创建  
cat1.name = "小白"; //对自建类对象具体赋值属性  
cat1.age = 3;  
cat1.color = "白色";  
cat1.weight = 10;
```

//正确创建类的位置:

```
public class HelloWorld{
    Run | Debug
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        B b = new B();
        Person peo = new Person();
        peo.age=520;
        peo.name="胡欣琦";
        System.out.println("I love "+peo.name+" forever~");
        System.out.println("Darling,I,forever~");
    }
}

class Person{
    int age;
    String name;
}

class B{
    public void personIn(Person p){
        p.age=1314;
        p.name="我们";
        return;
    }
}
```

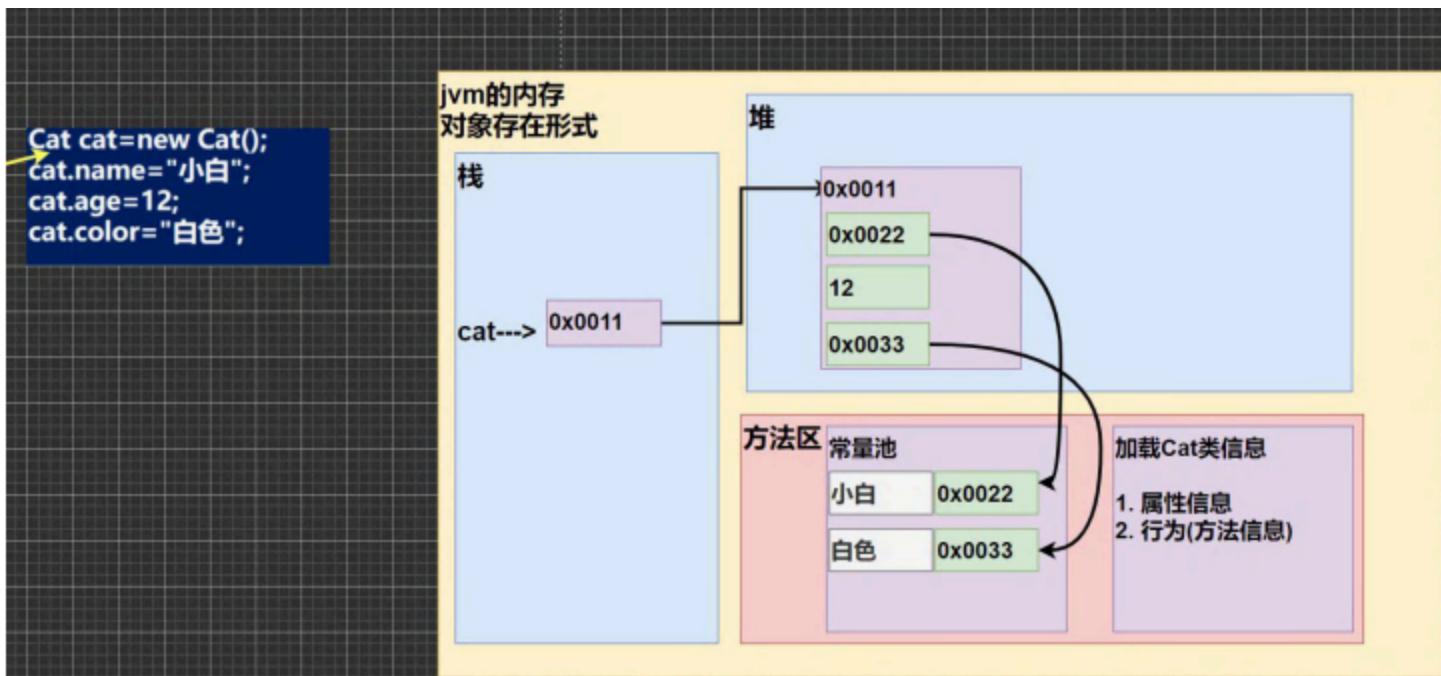
//应当在主类外面另外创建

```
We520
我们1314
I love 胡欣琦 forever~
Darling,I,forever~
```

## 1.2 访问示例:(与C++相似,类似于表达数组的长度 str.length)

```
System.out.println("第 1 只猫信息" + cat1.name + " " + cat1.age + " " + cat1.col
```

## 1.3 对象在内存中存在形式:



## 1.4 类的属性(同C++, 属性=成员变量=field(字段))

## 1.5 类的成员方法(同C++, 相当于兹定于类中的自定义函数)

```
class Person{  
    int age;  
    String name;  
    public void speak(){  
  
        //public表示为公开函数  
        //Java中将公开私密单独对每个函数赋予  
        //其余与C++都一样  
  
        System.out.println("我爱你");  
    }  
}  
Person p1= new Person();//记得加括号  
p1.age=19;  
p1.name="胡欣琦";  
Person p2=p1;  
System.out.println(p2.name);  
p1.speak();  
p1.name="徐璟昊";  
System.out.println(p2.name);  
p2.speak();
```

### 1. 关于参数与返回:

- i. 与C++一样,当然也有形参p3.hanshu(int a,int b)什么的可导入,但是与C++不同的是:不能给形参赋默认值  
(也就是说,不能写成(int a=xxx)的形式)
- ii. 当然,和C++一样,在成员函数里面改变基本数据类型的形参的值不改变实参的值  
(不过若是引用类型(自定义类(若要使用另一自定义类的方法,见3.跨类调用)或者数组),则可改变)  
(但是也仅限于其属性可改变,若试图改变其整个如p=null;则不会改变其值)
- iii. 输入参数需要是设定参数类型的相同或者兼容类型,个数与顺序需一致
- iv. 当然也可以通过return x;来返回值,void可以单独写return;

### 2. 参数与方法体注意事项与细节:

- ✓ 返回数据类型
  - 1) 一个方法最多有一个返回值 [思考, 如何返回多个结果 返回数组 ]
  - 2) 返回类型可以为任意类型, 包含基本类型或引用类型(数组, 对象)
  - 3) 如果方法要求有返回数据类型, 则方法体中最后的执行语句必须为 `return` 值; 而且要求返回值类型必须和 `return` 值类型一致或兼容
  - 4) 如果方法是 `void`, 则方法体中可以没有 `return` 语句, 或者 只写 `return;`
- ✓ 方法名

遵循驼峰命名法, 最好见名知义, 表达出该功能的意思即可, 比如 得到两个数的和 `getSum`, 开发中按照规范

### ✓ 形参列表

1. 一个方法可以有0个参数, 也可以有多个参数, 中间用逗号隔开, 比如 `getSum(int n1,int n2)`
2. 参数类型可以为任意类型, 包含基本类型或引用类型, 比如 `printArr(int[][] map)`
3. 调用带参数的方法时, 一定对应着参数列表传入相同类型或兼容类型的参数! 【`getSum`】
4. 方法定义时的参数称为形式参数, 简称形参; 方法调用时的传入参数称为实际参数, 简称实参, 实参和形参的类型要一致或兼容、个数、顺序必须一致! [演示]

### ✓ 方法体

里面写完功能的具体的语句, 可以为输入、输出、变量、运算、分支、循环、方法调用, 但里面不能再定义方法! 即: 方法不能嵌套定义。[演示]

### ✓ 方法调用细节说明(!!!)

1. 同一个类中的方法调用: 直接调用即可。比如 `print(参数);`  
案例演示: A类 `sayOk` 调用 `print()`
2. 跨类中的方法A类调用B类方法: 需要通过对象名调用。比如 `对象名.方法名(参数);` 案例演示: B类 `sayHello` 调用 `print()`
3. 特别说明一下: 跨类的方法调用和方法的访问修饰符相关, 先暂时这么提一下, 后面我们讲到访问修饰符时, 还要再细说。

//这里就是public,protected,private和默认的区别了,能否直接使用

3. 跨类调用:

//跨类中的方法 A 类调用 B 类方法：需要通过对象名调用

```
public void m1() {  
    //创建 B 对象，然后在调用方法即可  
    System.out.println("m1() 方法被调用");  
    B b = new B();  
    b.hi();  
  
    System.out.println("m1() 继续执行:");  
}
```

```
class B {  
  
    public void hi() {  
        System.out.println("B 类中的 hi() 被执行");  
    }  
}
```

//其实就是在A类方法(成员函数)里面创建B类的对象b,然后通过该对象b来使用B类方法(函数)

## 1.6 类与对象的内存分配机制(直接 = ,代表p2指向p1的对象):

```
class Person{  
    int age;  
    String name;  
}  
Person p1= new Person();  
p1.age=19;  
p1.name="胡欣琦";  
Person p2=p1;  
p1.name="徐璟昊";  
System.out.println(p2.name);
```

1. 内存结构:

**栈:一般存放基本数据类型(局部变量)**

**堆:存放对象(Cat cat , 数组等,还要字符串池)//字符串池案例可见错误及解决方案的6**

**方法区：常量池(常量，比如字符串)，类加载信息**

2. 创建对象流程:

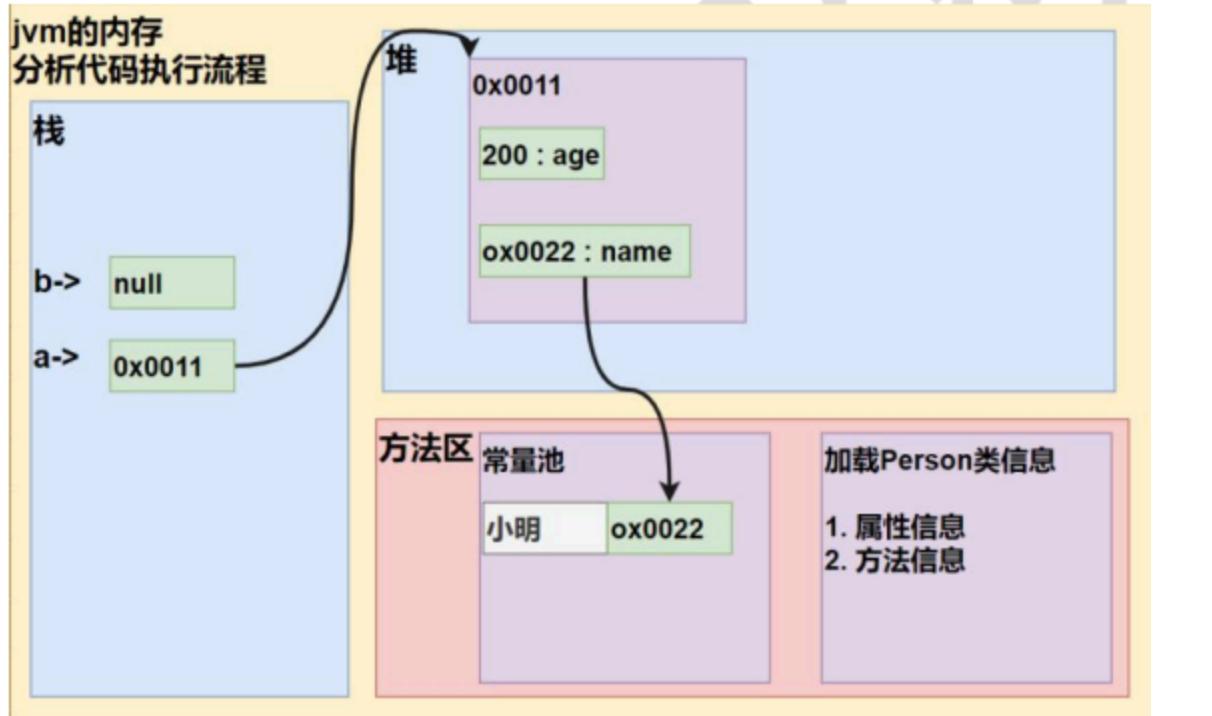
```
Person p = new Person();
p.name = "jack";
p.age = 10
```

- 1) 先加载 Person 类信息(属性和方法信息，只会加载一次)
- 2) 在堆中分配空间，进行默认初始化(看规则)
- 3) 把地址赋给 p , p 就指向对象
- 4) 进行指定初始化， 比如 p.name ="jack" p.age = 10

3. 示例:

//我们看看下面一段代码，会输出什么信息：

```
Person a=new Person();
a.age=10;
a.name="小明";
Person b;
b=a;
System.out.println(b.name); //小明
b.age=200;
b = null;
System.out.println(a.age); //200
System.out.println(b.age); //异常
```



## 1.7 方法递归(含示例):

1. 执行一个方法时，就创建一个新的受保护的独立空间(栈空间)
2. 方法的局部变量是独立的，不会相互影响，比如n变量
3. 如果方法中使用的是引用类型变量(比如数组，对象)，就会共享该引用类型的数据。
4. 递归必须向退出递归的条件逼近，否则就是无限递归，出现 `StackOverflowError`，死龟了：)
5. 当一个方法执行完毕，或者遇到`return`，就会返回，遵守谁调用，就将结果返回给谁，同时当方法执行完毕或者返回时，该方法也就执行完毕。

示例：

## 1. (迷宫找路):

```
public boolean findWay(int[][] map,int i,int j){  
    if(map[6][5]==2){//终点  
        return true;  
    }  
    else{  
        if(map[i][j]==0){//以下右上左顺序找路  
            map[i][j]=2;  
            if(findWay(map,i+1,j))  
                return true;  
            else if(findWay(map,i,j+1))  
                return true;  
            else if(findWay(map,i-1,j))  
                return true;  
            else if(findWay(map,i,j-1))  
                return true;  
            else{  
                map[i][j]=3;  
                return false;  
            }  
        }  
        else{  
            return false;  
        }  
    }  
}
```

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 0 | 0 | 1 |
| 1 | 3 | 1 | 2 | 0 | 0 | 1 |
| 1 | 1 | 1 | 2 | 0 | 0 | 1 |
| 1 | 0 | 0 | 2 | 0 | 0 | 1 |
| 1 | 0 | 0 | 2 | 0 | 0 | 1 |
| 1 | 0 | 0 | 2 | 2 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

## 2. 八皇后问题(国际象棋放八个皇后,不能互相攻击):

```
public class Helloworld{
    public static void main(String[] args) {
        //Scanner inp = new Scanner(System.in);
        int[][] map = new int[8][8];//创建棋盘
        Move alte = new Move();//创建自定义类
        for(int i=0;i<8;i++){
            for(int j=0;j<8;j++){
                System.out.print(map[i][j]+" ");
            }
            System.out.println();
        }
        alte.findWay(map,0,0);
        System.out.println(cnt);
    }

    static int cnt=0;//创建静态统计数

    static class Move{
        public boolean judge(int[][] map,int length,int n){
            //通过看上,左上,右上是否有皇后来判断皇后是否安全
            for(int i=0;i<=length;i++){
                if(map[i][n]==7)
                    return false;
                if(n+i<8)
                    if(map[length-i][n+i] == 7)
                        return false;
                if(n-i>=0)
                    if(map[length-i][n-i] == 7)
                        return false;
            }
            return true;
        }
        public void findWay(int[][] map,int row,int col){
            //递归尝试放置皇后
            if(row==8){//成功放置八个皇后, row行, col列
                cnt += 1;
                for (int i = 0; i < 8; i++) {//打印
                    for (int j = 0; j < 8; j++)
                        System.out.print(map[i][j] + " ");
                    System.out.println();
                }
                System.out.println();
            }
        }
    }
}
```

```

        return;
    }
    for(int i = col;i < 8; i++){
        if(judge(map, row, i)){
            map[row][i] = 7;
            findWay(map, row+1, 0); //递归放置下一个皇后
            map[row][i] = 0; //回溯,移除皇后
            //这有点让人茅塞顿开
            //通过递归来阻止回溯
            //通过不递归来回溯
        }
    }
}
}

```

## 1.8 方法重载(同C++)

### 1. 定义:

**允许同一个类中，多个同名方法的存在，仅要求形参列表不一致(只包括类型或者个数或者顺序,但是参数名字和返回类型不算)**

(也就是说参数名字不同或者仅仅是返回类型不同都不算,因为无法区分参数信息嘛,就不能重载)

### 2. 例:

案例：类： MyCalculator 方法： calculate

- 1) calculate(int n1, int n2) //两个整数的和
- 2) calculate(int n1, double n2) //一个整数，一个 double 的和
- 3) calculate(double n2, int n1)//一个 double ,一个 Int 和
- 4) calculate(int n1, int n2,int n3)//三个 int 的和

## 1.9 public和static类(与10.合并查看)

### 1. public:

- i. 一个Java文件中只能有一个公共类(如果有的话)，并且文件名必须与类名相同
- ii. 这个类是公共的，可以被任何包中的任何类访问
- iii. 非public类就只能在一个包中(的其他类)使用,可以在一个JAVA文件中存在多个

### 2. static(不依赖外部类的实例):

- i. 是其外部类的一个静态成员,可通过外部类的类名直接访问
- ii. 不依赖外部类的实例,所以不能访问外部类的非静态成员
- iii. 同样的,可在无外部类实例的情况下创建

iv. 非static类是外部类的一个成员，依赖于外部类的实例,所以需要一个外部类的实例来创建一个内部类的实例,可以直接访问外部类的所有成员，包括私有成员,与外部类的实例绑定

## 1.10 外部类和内部类(与9.合并查看)

1. 外部类:

不被包含在某一类之内的类,只有public class(最多存在一个)和class之分,也就是说,外部类除了public class外的类不能有修饰符

2. 内部类:

在某一类之内的类,可以有public,protected,private,static,public/protected/private static等修饰符或者默认的class

## 1.11 可变参数

(JAVA允许将同一个类中多个同名同功能但参数个数不同的方法，封装成一个方法。

通过可变参数实现)

1. 本质:数组(通过数组大小变化改变参数数量,接收实参可以是任意个,可以为数组,但是必须是同一类型,可变参数再怎么可变,也还是有专一的地方的嘛)

2. 示例:

```
public int sum(int... nums) {  
    //System.out.println("接收的参数个数=" + nums.length);  
    int res = 0;  
    for(int i = 0; i < nums.length; i++) {  
        res += nums[i];  
    }  
    return res;  
}
```

3. 注意事项

- i. 可以与普通参数一起在形参列表,但是可变参数必须在最后(因为可以接收任意数量实参嘛)
- ii. 同理,一个形参列表最多只能有一个可变参数

## 1.12 变量作用域(同C++)

```
class Darling{  
String name;  
String lover;  
public void love(){String live=...;}  
}
```

- 全局变量(属性):有默认值,可以不赋值,作用域为整个类,比如上面的name和lover,**可以加修饰符  
(public,protected,private,static)**  
(随着对象创建而创建,随对象销毁而销毁)
- 局部变量:没有默认值,必须赋值才能使用,除了全局变量(属性)以外的变量,作用域为其代码块中,比如上面的live,**不能加修饰符**  
(随着代码块执行而创建,随代码块结束而销毁)

## 1.13 自动构造器(与C++基本相同)

- 基础使用示例(与C++语法相同):

//当我们 new 一个对象时, 直接通过构造器指定名字和年龄

```
Person p1 = new Person("smith", 80);
```

```
class Person {
String name;
int age;
public Person(String pName, int pAge) { //这一段就是构造器
System.out.println("构造器被调用~~ 完成对象的属性初始化");
name = pName;
age = pAge;
}
```

//1. 构造器没有返回值, 也不能写 void

//2. 构造器的名称和类 Person 一样

//3. 构造器形参列表规则和成员方法一样

**如果程序员没有定义构造器, 系统会自动给类生成一个默认无参构造器(也叫默认构造器), 比如 Dog (){}, 使用javap指令 反编译看看**

**一旦定义了自己的构造器,默认的构造器就覆盖了, 就不能再使用默认的无参构造器, 除非显式的定义一下,即: Dog(){} 写 (这点很重要)**

\*\*注意!\*\*也就是说,若是使用了自建的构造器,就无法再进行Person p = new Person()()默认构造了,除非再写上Person(){};强调添加该构造方式

- 构造器重载(与C++和成员方法重载相同)
- 案例流程分析:

```
class Person{//类Person
    int age=90;
    String name;
    Person(String n,int a){//构造器
        name=n;//给属性赋值
        age=a;...
    }
    Person p=new Person("小倩",20);
}
● 流程分析(面试题)
1. 加载Person类信息(Person.class) , 只会加载一次
2. 在堆中分配空间(地址)
3. 完成对象初始化 [3.1 默认初始化 age=0 name=null 3.2 显式初始化
age=90,name=null, 3.3 构造器的初始化 age =20, name=小倩]
4. 在对象在堆中的地址,返回给 p(p 是对象名,也可以理解成是对象的引用)
```

## 1.14 this关键字(同C++,也较为重要)

(哪个对象调用,this就代表哪个对象)

1. 使用示例:

```
public Dog(String name, int age){//构造器
    //this.name 就是当前对象的属性 name
    this.name = name;
    //this.age 就是当前对象的属性 age
    this.age = age;
    System.out.println("this.hashCode=" + this.hashCode());
}
```

```
public void info(){//成员方法,输出属性 x 信息
    System.out.println("this.hashCode=" + this.hashCode());
    System.out.println(name + "\t" + age + "\t");
}
```

2. 注意事项:

- i. this可以来访问本类的属性、方法、**构造器**,可用于区分当前类的属性和局部变量
- ii. 访问成员方法的语法: **this.方法名(参数列表)**
- iii. 访问构造器的语法: **this(参数列表)**  
(即只能用于在构造器中访问另外一个构造器, 必须放在第一条语句)

示例:

```
Person(int age,int x){...}
Person(int age,int x,String name){
    this(age,x); //只能放在第一句,也就是说只能写一句并且要在第一行
    this.name=name;
}
```

#### iv. 只能在类定义的方法中使用

## 1.15 关于类的对象赋值(为引用传递.即赋值地址)

示例:

```
Person p1 = new Person();
Person p2 = p1;
```

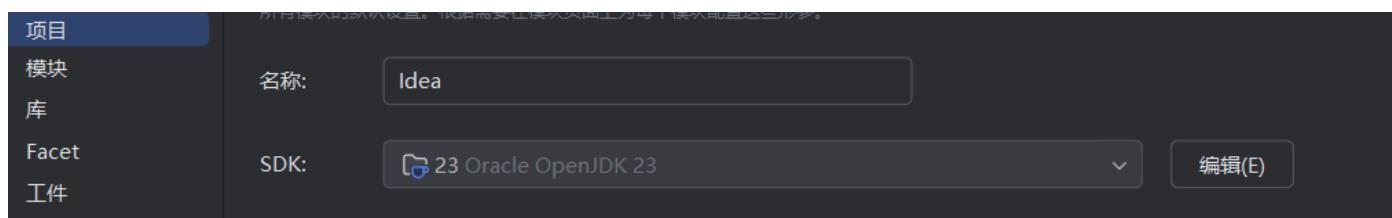
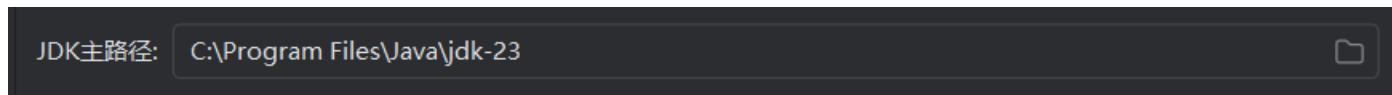
//此时p2与p1指向同一个对象,改变其一就会都改变

## (二).面向对象编程(中级部分)

### 2.1 Idea

1. 特点(以项目为概念管理源码)

2. 环境配置:



直接设置路径即可

3. 快捷键:

- 1) 删除当前行, 默认是 `ctrl + Y` 自己配置 `ctrl + d`
- 2) 复制当前行, 自己配置 `ctrl + alt +` 向下光标
- 3) 补全代码 `alt + /`
- 4) 添加注释和取消注释 `ctrl + /` 【第一次是添加注释, 第二次是取消注释】
- 5) 导入该行需要的类 先配置 `auto import`, 然后使用 `alt+enter` 即可
- 6) 快速格式化代码 `ctrl + alt + L`
- 7) 快速运行程序 自己定义 `alt + R`
- 8) 生成构造器等 `alt + insert` [提高开发效率]
- 9) 查看一个类的层级关系 `ctrl + H` [学习继承后, 非常有用]
- 10) 将光标放在一个方法上, 输入 `ctrl + B`, 可以定位到方法 [学继承后, 非常有用]
- 11) 自动的分配变量名, 通过 在后面假 `.var` [老师最喜欢的]

**file -> settings -> editor-> Live templates ->  
查看有哪些模板快捷键/可以自己增加模板**

**模板可以高效的完成开发, 提高速度**

## 2.2 包(本质接收不同的文件夹/目录保存类文件)

### 1. 作用:

1. 区分相同名字的类
2. 类很多时,更好的管理类
3. 控制访问范围

### 2. 基本语法:

1. `package`:关键字,表示打包  
(声明当前类所在的包,需要放在最上面,当然,一个类最多只有一句`package`)
2. `com.hxqlovezjh`:表示包名
3. 引用包:
  - i. `import java.util.*;` //引入整个包,不过一般不建议
  - ii. `import java.util.Scanner;` //只引入一个类Scanner

### 3. 命名规范:

一般是小写字母+小圆点一般是  
com.公司名.项目名.业务模块名  
比如: com.hspedu.oa.model; com.hspedu.oa.controller;  
举例:  
com.sina.crm.user //用户模块  
com.sina.crm.order // 订单模块  
com.sina.crm.utils //工具类

## 2.3 常用包:

java.lang.\* //lang 包是基本包, 默认引入, 不需要再引入.  
java.util.\* //util 包, 系统提供的工具包, 工具类, 使用 Scanner  
java.net.\* //网络包, 网络开发  
java.awt.\* //是做 java 的界面开发, GUI

## 2.4 常用类:

1. java.util:
  - i. Arrays:

```
int[] arr = {-1,20,2,13,3};  
Arrays.sort(arr); //从小到大排序
```
  - ii. Object:(所有类的最高父类,自动导入)  

```
x.equals(y);
```

## 2.5 访问修饰符

1. 四种修饰符:
  - i. public:公开级别,能在不同包中使用
  - ii. protected:受保护级别,能在子类和同一个包中使用

- 1. 同一个包中的其他类:** 同一个包 (Package) 内的其他类可以访问 `protected` 成员, 无需任何特殊权限。
- 2. 不同包中的子类:** 在不同包中的子类 (即继承了包含 `protected` 成员的类) 也可以访问这些 `protected` 成员。这是因为继承关系超越了包的界限。
- 3. 同一个包中的子类:** 如果子类与被继承的类在同一个包中, 那么子类可以访问父类的 `protected` 成员。
- 4. 不同包中的非子类:** 不能访问 `protected` 成员。即使两个类不在同一个包中, 如果它们没有继承关系, 那么它们也不能相互访问对方的 `protected` 成员。
- 5. 同一个类:** 同一个类内部可以访问它自己的 `protected` 成员。
- 6. 无法通过类名直接访问:** 不能通过类名直接访问 `protected` 成员, 必须通过实例化对象或通过子类的对象来访问。

iii. 默认: 只能在同一个包中使用

iv. `private`: 私有级别, 只能在同一个类中使用

| 1 | 访问级别 | 访问控制修饰符                | 同类 | 同包 | 子类 | 不同包 |
|---|------|------------------------|----|----|----|-----|
| 2 | 公开   | <code>public</code>    | ✓  | ✓  | ✓  | ✓   |
| 3 | 受保护  | <code>protected</code> | ✓  | ✓  | ✓  | ✗   |
| 4 | 默认   | 没有修饰符                  | ✓  | ✓  | ✗  | ✗   |
| 5 | 私有   | <code>private</code>   | ✓  | ✗  | ✗  | ✗   |

## 2. 注意事项:

- i. 只有 `默认` 和 `public` 能修饰类和接口
- ii. 四种都可以修饰类中的属性和成员方法

## 2.6 面向对象编程三大特征--封装, 继承, 多态

### 1. 封装(同C++)

#### 1. 理解:

将抽象的数据(属性)与对数据的操作(方法)封装在一起, 数据被保护在内部, 只有通过被授权的操作(方法), 才能对数据进行操作  
(也就是说让数据不会被乱七八糟的东西改变, 只能通过指定方法改变, 将数据分存起来)

#### 2. 目的:

保护数据安全, 还可以隐藏操作的实现细节(因为是直接调用)

### 3. 实现:

1. 私有化属性
2. 加入公共化方法用于赋值属性和获取属性的值
3. 示例:

```
class Person{  
    private String name;  
    private double balance;  
    private String password;  
    public Person(){}
    public Person(String name,double balance,String password  
        setName(name);  
        setBalance(balance);  
        setPassword(password);  
    }  
    public void setName(String name){  
        if(name.length() >= 2 && name.length() <=3 ) {  
            this.name = name;  
        }  
        else{  
            System.out.println("名字的长度不对，需要(2-3)个字");  
            this.name = "无名人";  
        }  
    }  
}
```

## 2. 继承(同C++) (子类与父类)

### 1. 理解:

建立查找的关系,通过继承像链表一样连结

示例:

```

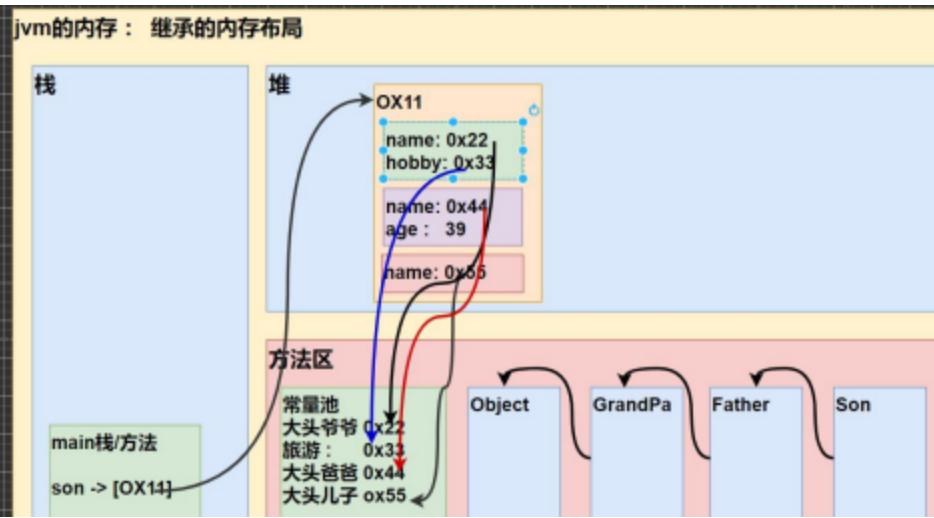
public class ExtendsTheory {
    public static void main(String[] args) {
        Son son = new Son(); //内存的布局
    }
}

class GrandPa { //父类
    String name = "大头爷爷";
    String hobby = "旅游";
}

class Father extends GrandPa { //父类
    String name = "大头爸爸";
    int age = 39;
}

class Son extends Father { //子类
    String name = "大头儿子";
}

```



## 2. 目的:

增加代码复用性,增加代码拓展性和维护性

## 3. 实现:

```
class 子类 extends 父类{
```

```
...
```

- 1) 子类就会自动拥有父类定义的属性和方法
- 2) 父类又叫 超类, 基类。
- 3) 子类又叫派生类。

## 4. 注意事项:

1. 非私有的属性和方法(包括public,protected和默认)可以在子类直接访问, 但私有属性和方法(private)不能在子类直接访问, 需通过父类提供公共的方法去访问
2. 子类必须调用父类的构造器先对父类初始化(显而易见),既然如此,若父类有无参构造器,则会默认调用该无参构造器,而若父类只有有参构造器(未额外定义无参构造器),则必须用super去对父类初始化,示例:

(super必须在子类构造器的第一行(要先初始化父类嘛))

(又因为super()和this()都要在第一行,所以不能同时在同一构造器使用)

```
// 子类的构造器
public Child(int childField) {
    super(10); // 显式调用父类的有参构造器
    this.childField = childField;
}
```

//super();和什么都不写都指调用无参构造器(默认)

3. java 所有类都是 Object 类的子类, Object 是所有类的基类(父类).而父类构造器的调用不限于直接父类,一直往上追溯直到Object类都行,但是必须一个一个按照父类的父类的父类的顺序追溯(坏了,我的附庸的附庸真是我的附庸了)
4. 一个父类可以有许多个子类,但是一个子类最多继承一个父类(子类只能直接调用 直接父类的 非私有部分 ,但是可以通过链条一样的方式可以调用父类的父类...)
5. 子类和父类是is-a关系

### "is-a" 关系的定义:

- **子类是父类的一种**: 子类继承了父类的所有属性和方法, 并且可以有自己的特定属性和方法。
- **子类拥有父类的所有特性**: 子类的对象可以被视为父类的对象, 具有父类的所有行为和状态。
- **多态性**: 子类可以替换父类, 出现在任何需要父类对象的地方。

## 5. super(关键字)详解

1. 含义:代表父类的引用,用于访问父类的**非private**属性,方法,构造器

2. 语法:

- i. 访问属性:super.属性名;
- ii. 访问方法:super.方法名(参数列表);
- iii. 访问构造器:super(参数列表);**//只能放在第一句,且只能出现一句**

3. 目的:

- i. 父类子类各自初始化各自的属性
- ii. 区分子类父类重名的成员(属性/方法)

**super的访问不限于直接父类, 如果爷爷类和本类中有同名的成员, 也可以使用super去访问爷爷类的成员; 如果多个基类(上级类)中都有同名的成员, 使用super访问遵循就近原则。A->B->C, 当然也需要遵守访问权限的相关规则**

//就像链表一样

4. 与this的区别:

| No. | 区别点   | this                         | super                 |
|-----|-------|------------------------------|-----------------------|
| 1   | 访问属性  | 访问本类中的属性, 如果本类没有此属性则从父类中继续查找 | 从父类开始查找属性             |
| 2   | 调用方法  | 访问本类中的方法, 如果本类没有此方法则从父类继续查找. | 从父类开始查找方法             |
| 3   | 调用构造器 | 调用本类构造器, 必须放在构造器的首行          | 调用父类构造器, 必须放在子类构造器的首行 |
| 4   | 特殊    | 表示当前对象                       | 子类中访问父类对象             |

## 6.方法覆盖/重写(属性不行,不行,不行)

### 1. 理解:

子类方法与父类方法名称,返回类型(或者是父类方法的子类也行,比如子类String,父类Object),参数数量类型顺序一样,就运行子类的类,把父类的类覆盖了(显而易见)

//但是不影响父类的修饰符的范围

## 3.多态(封装和继承基础之上)

### 1. 理解:多态有多种表现形式:

1. 方法:方法重写/重载
2. 对象:一个对象的编译类型(定义时确定)和运行类型(看 = 的右边)可以不一致

### 2. 示例:

//Dog和Cat类继承Animal类

```
Animal animal = new Dog();
animal.cry();
animal = new Cat();
animal.cry();
```

不可名状的犬型黑影正在嘶吼和轻语  
不可名状的猫型黑影正在梦呓和低语

### 3. 使用注意事项:

1. 多态前提是两个对象(类)存在继承关系
2. 向上转型:

**1) 本质：父类的引用指向了子类的对象**  
**2) 语法：父类类型 引用名 = new 子类类型();**  
**3) 特点：编译类型看左边，运行类型看右边。**  
    **可以调用父类中的所有成员(需遵守访问权限)，**  
    **不能调用子类中特有成员；**  
    **最终运行效果看子类的具体实现！**

(子类用父类的new,需要遵守访问权限用)

### 3. 向下转型：

**1) 语法：子类类型 引用名 = (子类类型) 父类引用;**  
**2) 只能强转父类的引用，不能强转父类的对象**  
**3) 要求父类的引用必须指向的是当前目标类型的对象**  
**4) 当向下转型后，可以调用子类类型中所有的成员**

(父类用子类的new,全部都能用)

## 4.JAVA动态绑定机制(very重要)

理解：

1. 调用对象方法时,该方法与该对象的内存地址/运行类型绑定
2. 调用对象属性时,没有动态绑定机制,哪里声明,哪里使用

```
class A {//父类
    public int i = 10;
    public int sum() {
        return getI() + 10;
    }
    public int sum1() {
        return i + 10;
    }
    public int getI() {
        return i;
    }
}
```

```
class B extends A {//子类
    public int i = 20;
    public int sum() {
        return i + 20;
    }
    public int getI() {
        return i;
    }
    public int sum1() {
        return i + 10;
    }
}
```

```
//main方法中
A a = new B();//向上转型
System.out.println(a.sum());//?40
System.out.println(a.sum1());//?30
2min
```

### java的动态绑定机制

1. 当调用对象方法的时候, 该方法会和该对象的内存地址/运行类型绑定
2. 当调用对象属性时, 没有动态绑定机制, 哪里声明, 那里使用

//可以这么理解,此处new了B类,就相当于B为替身,用替身去操作,相当于此时a就占据了B类,所以此处用B的属性去进行操作,并且依据B作为优先查询的类,若无实现方法,则向上看看父类有没有

## 5. 多态应用

1. 多态数组:

示例:(Student和Teacher是Person的子类)

```

Person[] persons = new Person[5];
persons[0] = new Person("jack", 20);
persons[1] = new Student("mary", 18, 100);
persons[2] = new Student("smith", 19, 30.1);
persons[3] = new Teacher("scott", 30, 20000);
persons[4] = new Teacher("king", 50, 25000);
//循环遍历多态数组，调用 say
for (int i = 0; i < persons.length; i++) {
    //person[i] 编译类型是 Person ,运行类型是根据实际情况有 JVM 来判断
    System.out.println(persons[i].say());//动态绑定机制
    //下处使用 类型判断 + 向下转型.
    if (persons[i] instanceof Student) {
        //判断 person[i] 的运行类型是不是 Student
        //instanceof是判断对象是否是特点类的二元操作符
        Student student = (Student) persons[i];//向下转型
        student.study();
        //也可以使用一条语句 ((Student)persons[i]).study();
    } else if (persons[i] instanceof Teacher) {
        Teacher teacher = (Teacher) persons[i];
        teacher.teach();
    } else if (persons[i] instanceof Person) {
        System.out.println("你的类型有误，请自己检查...");
```

```

    } else {
        System.out.println("你的类型有误，请自己检查...");
    }
    //下面是私货，XD
}
```

```

    String MyLover = "胡欣琦";
    System.out.println("I love " + MyLover + " forever~");
    System.out.println("Darling,I,forever~");
```

2. 多态参数(方法定义形参类型是父类,实参类型允许是子类--父类兼容子类):

## 2.7 Object类详解

### 1. equals(与下一点的hashCode结合看)

1. equals 和 ==

==可以判断基本类型(比较的是值)和引用类型(类,接口,数组,看地址是否相等)

equals只能判断引用类型(默认判断地址是否相等,但是可以重写去比较对象的属性/状态,重写后也只能是引用类型)

2. 重写equals(就跟普通的重写一样,在自定义类里面设置就行)

(通常会一同重写hashCode方法)

示例:

```
public boolean equals(Object obj){  
    if(this == obj){  
        return true;  
    }  
    if(obj instanceof Manager){  
        Manager m = (Manager)obj;  
        return this.bonus.equals(m.bonus)&&this.getName().equals(m.getName());  
    }  
    return false;  
}
```

## 2.hashCode方法(作为哈希函数,提供哈希值,为了提高哈希表的性能)

### 1. 哈希表(一种数据结构):提供快速的数据插入和查找功能

用一种哈希函数来计算数据的存储位置,这个位置通常称为哈希码或哈希值,通过这种方式,哈希表能够以接近常数时间的性能进行数据的访问,这使得它非常适合于需要快速查找、插入和删除的场景

### 2. hashCode方法与哈希表>equals的关系:

1. 哈希表:调用hashCode()方法计算哈希值,同一个对象返回同一个值

2. equals:若两个对象通过equals方法比较是相同的,那么hashCode方法返回的哈希值也应相同,通常重写equals,hashCode也一起重写,确保相等对象有相同的哈希码

### 3. hashCode自身要求:应以快速的计算尽量均匀分布哈希值,减少哈希冲突

### 4. hashCode示例

```
AA aa = new AA();  
AA aa2 = new AA();  
AA aa3 = aa;  
System.out.println("aa.hashCode()=" + aa.hashCode());  
System.out.println("aa2.hashCode()=" + aa2.hashCode());  
System.out.println("aa3.hashCode()=" + aa3.hashCode());
```

```
aa.hashCode()=189568618  
aa2.hashCode()=664223387  
aa3.hashCode()=189568618  
I love 胡欣琦 forever~  
Darling,I,forever~
```

//此处aa与aa3是同一对象,返回同一哈希值

### 3. **toString方法(子类可重写来返回对象的属性信息)**

#### 1. 基本介绍:

默认返回: 全类名+@+哈希值的十六进制

#### 2. 重写示例:

```
Employee emp = new Employee("Tiga","光之巨人",123456789);  
System.out.println(emp.toString()+"hashCode="+emp.hashCode());  
System.out.println("==当直接输出一个对象时, toString 方法会被默认的调用==");  
System.out.println(emp);  
//输出emp等价于输出emp.toString()  
...  
...  
public String toString(){  
    return "Employee{"+"name="+name+","+"job="+job+","+"sal="+sal+"}";  
}
```

```
Employee{name=Tiga, job=光之巨人, sal=1.23456789E8}hashCode=511754216  
==当直接输出一个对象时, toString 方法会被默认的调用==  
Employee{name=Tiga, job=光之巨人, sal=1.23456789E8}
```

### 4. **finalize方法(释放非内存资源,实际开发几乎用不到,面试可能用)**

1. **资源清理:** finalize() 方法可以用来释放非内存资源, 如文件句柄、数据库连接或网络资源。

2. **对象复用:** 在某些情况下, finalize() 方法可以用来恢复对象的状态, 从而避免垃圾回收器回收该对象。

#### 1. 默认使用:

对象被确认为垃圾时, 调用对象的finalize方法回收内存, 子类可重写来释放资源

#### 2. 主动用法:

```

Car bmw = new Car("宝马");
//这时 car 对象就是一个垃圾,垃圾回收器就会回收(销毁)对象,在销毁对象前,会调用该对象的 finalize 方法
//,程序员就可以在 finalize 中,写自己的业务逻辑代码(比如释放资源: 数据库连接,或者打开文件..)
//,如果程序员不重写 finalize,那么就会调用 Object 类的 finalize, 即默认处理

//,如果程序员重写了 finalize, 就可以实现自己的逻辑

bmw = null;

System.gc(); //主动调用垃圾回收器

System.out.println("程序退出了....");

```

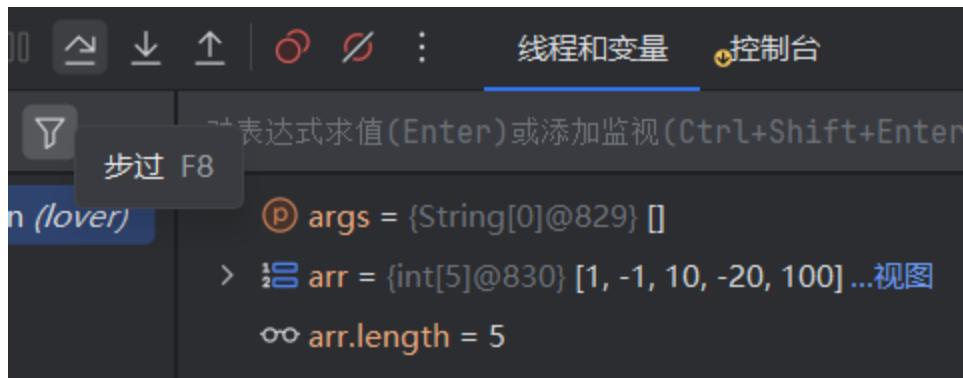
### 3. 不推荐使用原因:

- 1. 不保证会被调用:** 由于垃圾回收器的实现和行为可能不同,  
所以不能保证 finalize() 方法一定会被调用。
- 2. 不可预测性:** finalize() 方法的调用时机是不确定的, 可能  
在对象成为垃圾后的任何时间调用。
- 3. 性能开销:** 依赖 finalize() 方法来释放资源可能导致性能  
问题, 因为垃圾回收器的运行是不可预测的。
- 4. 不推荐使用:** 由于上述原因, Java 社区普遍不推荐使用  
finalize() 方法。相反, 应该使用 try-with-resources 语句  
或 try-finally 块来确保资源被正确释放。

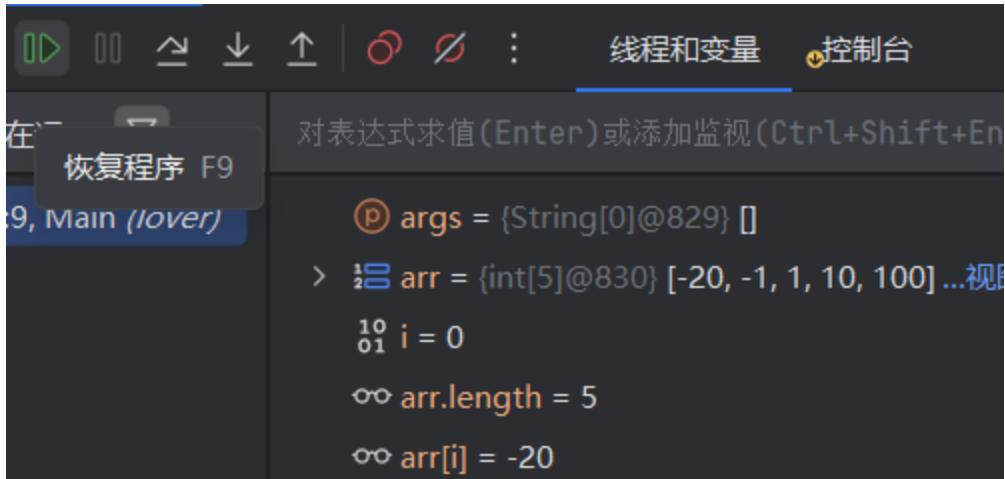
## 2.8 断点调试(debug,一步一步查看源码执行过程找bug)

### 1. 基本信息与使用:

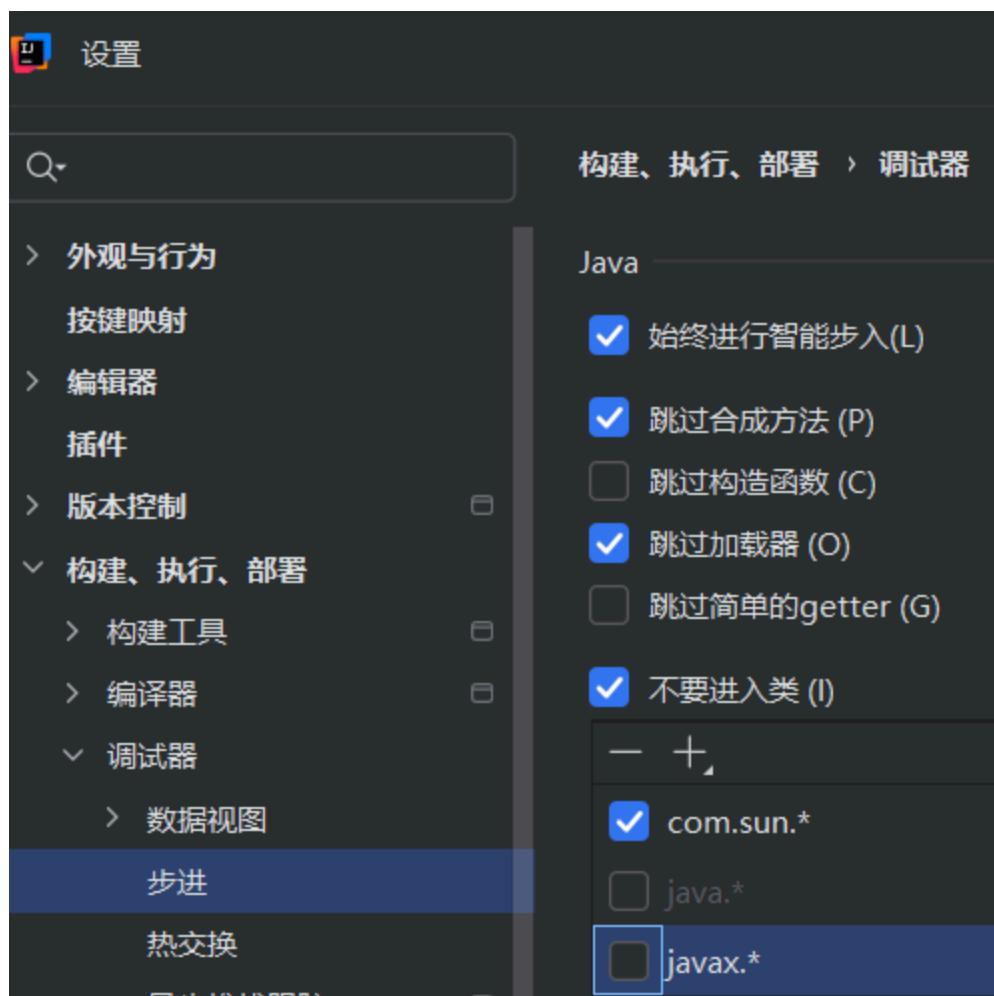
1. 断点调试过程是**运行状态**,过程中可以临时增删断点(反正是执行状态,唉嘿),**不能到达的断点就跳过**
2. 是**指在程序某一行设置一个断点,调试到该行会停住**
3. **一步一步执行:步过(F8)**



#### 4. 直接执行到下一断点:恢复程序(F9)

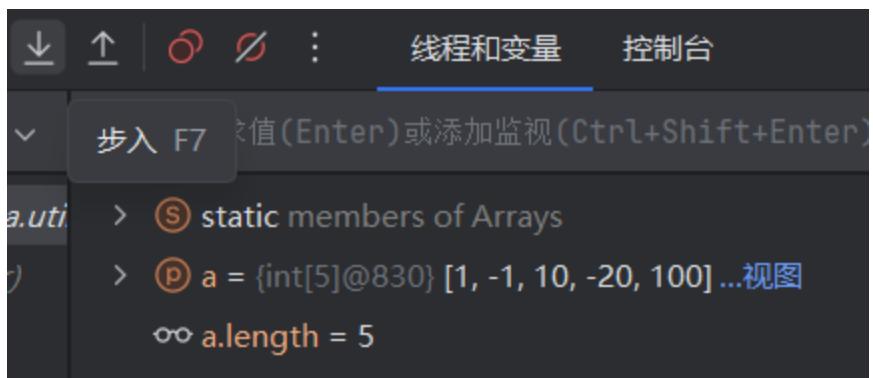


#### 2. 进入源码:



//java 和 javax 取消勾选\*\*

随后在调试时,选择步入↓(F7)即可



//可以不断步入直到根源

## 2.9 项目:房屋出租系统:

### 1.工具类(别人写好的包当作工具用)Utility

1. 代码:

```
package ...;
import java.util.*;

public class Utility {
    //静态属性。。
    private static Scanner scanner = new Scanner(System.in);

    /**
     * 功能：读取键盘输入的一个菜单选项，值：1—5的范围
     * @return 1—5
     */
    public static char readMenuSelection() {
        char c;
        for (; ; ) {
            String str = readKeyBoard(1, false); //包含一个字符的字符串
            c = str.charAt(0); //将字符串转换成字符char类型
            if (c != '1' && c != '2' &&
                c != '3' && c != '4' && c != '5') {
                System.out.print("选择错误，请重新输入：");
            } else break;
        }
        return c;
    }

    /**
     * 功能：读取键盘输入的一个字符
     * @return 一个字符
     */
    public static char readChar() {
        String str = readKeyBoard(1, false); //就是一个字符
        return str.charAt(0);
    }

    /**
     * 功能：读取键盘输入的一个字符，如果直接按回车，则返回指定的默认值；否则返回输入的那个字符
     * @param defaultValue 指定的默认值
     * @return 默认值或输入的字符
     */
    public static char readChar(char defaultValue) {
        String str = readKeyBoard(1, true); //要么是空字符串，要么是一个字符
        return (str.length() == 0) ? defaultValue : str.charAt(0);
    }
}
```

```
/**  
 * 功能：读取键盘输入的整型，长度小于2位  
 * @return 整数  
 */  
public static int readInt() {  
    int n;  
    for (; ; ) {  
        String str = readKeyBoard(10, false); //一个整数，长度<=10位  
        try {  
            n = Integer.parseInt(str); //将字符串转换成整数  
            break;  
        } catch (NumberFormatException e) {  
            System.out.print("数字输入错误，请重新输入： ");  
        }  
    }  
    return n;  
}  
/**  
 * 功能：读取键盘输入的 整数或默认值，如果直接回车，则返回默认值，否则返回输入的整数  
 * @param defaultValue 指定的默认值  
 * @return 整数或默认值  
 */  
public static int readInt(int defaultValue) {  
    int n;  
    for (; ; ) {  
        String str = readKeyBoard(10, true);  
        if (str.equals("")) {  
            return defaultValue;  
        }  
        //异常处理...  
        try {  
            n = Integer.parseInt(str);  
            break;  
        } catch (NumberFormatException e) {  
            System.out.print("数字输入错误，请重新输入： ");  
        }  
    }  
    return n;  
}  
/**
```

```

* 功能：读取键盘输入的指定长度的字符串
* @param limit 限制的长度
* @return 指定长度的字符串
*/

```

```

public static String readString(int limit) {
    return readKeyBoard(limit, false);
}

/**
 * 功能：读取键盘输入的指定长度的字符串或默认值，如果直接回车，返回默认值，否则返回字符串
* @param limit 限制的长度
* @param defaultValue 指定的默认值
* @return 指定长度的字符串
*/

```

```

public static String readString(int limit, String defaultValue) {
    String str = readKeyBoard(limit, true);
    return str.equals("")? defaultValue : str;
}

```

```

/**
 * 功能：读取键盘输入的确认选项，Y或N
 * 将小的功能，封装到一个方法中.
 * @return Y或N
*/

```

```

public static char readConfirmSelection() {
    System.out.println("请输入你的选择(Y/N): 请小心选择");
    char c;
    for ( ; ) { //无限循环
        //在这里，将接受到字符，转成了大写字母
        //y => Y n=>N
        String str = readKeyBoard(1, false).toUpperCase();
        c = str.charAt(0);
        if (c == 'Y' || c == 'N') {
            break;
        } else {
            System.out.print("选择错误，请重新输入: ");
        }
    }
    return c;
}

```

```
/**  
 * 功能： 读取一个字符串  
 * @param limit 读取的长度  
 * @param blankReturn 如果为true ,表示 可以读空字符串。  
 * 如果为false表示 不能读空字符串。  
 *  
 *      如果输入为空， 或者输入大于limit的长度， 就会提示重新输入。  
 * @return  
 */  
private static String readKeyBoard(int limit, boolean blankReturn) {  
  
    //定义了字符串  
    String line = "";  
  
    //scanner.hasNextLine() 判断有没有下一行  
    while (scanner.hasNextLine()) {  
        line = scanner.nextLine(); //读取这一行  
  
        //如果line.length=0, 即用户没有输入任何内容， 直接回车  
        if (line.length() == 0) {  
            if (blankReturn) return line; //如果blankReturn=true,可以返回空串  
            else continue; //如果blankReturn=false,不接受空串， 必须输入内容  
        }  
  
        //如果用户输入的内容大于了 limit, 就提示重写输入  
        //如果用户输入的内容 >0 <= limit ,我就接受  
        if (line.length() < 1 || line.length() > limit) {  
            System.out.print("输入长度（不能大于" + limit + "）错误，请重新输入：" );  
            continue;  
        }  
        break;  
    }  
  
    return line;  
}  
}
```

## (三).面向对象编程(高级部分)

### 3.1 main方法语法

#### 1. 理解语法:

解释main方法的形式：**public static void main(String[] args){}**

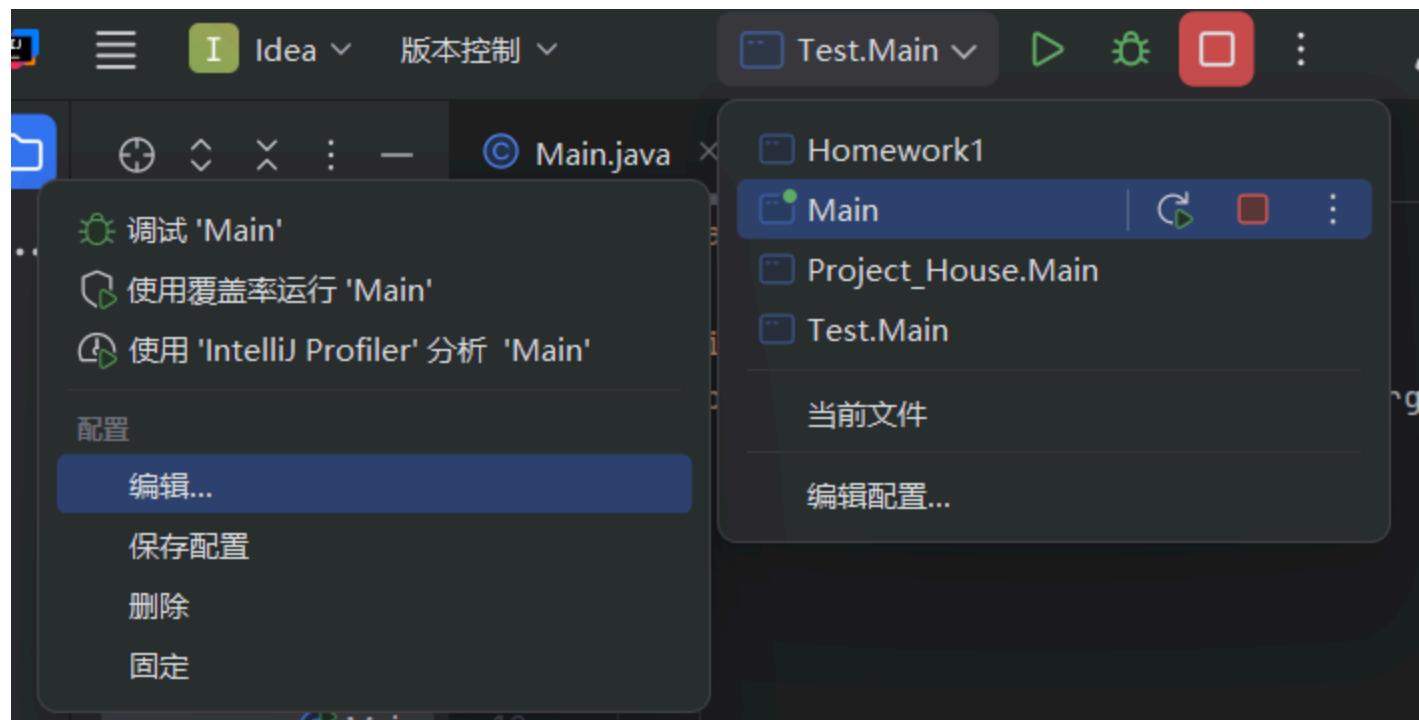
1. main方法时虚拟机调用
2. java虚拟机需要调用类的main()方法，所以该方法的访问权限必须是public
3. java虚拟机在执行main()方法时不必创建对象，所以该方法必须是static
4. 该方法接收String类型的数组参数，该数组中保存执行java命令时传递给所运行的类的参数.案例演示，接收参数.
5. java 执行的程序 参数1 参数2 参数3 [举例说明:]

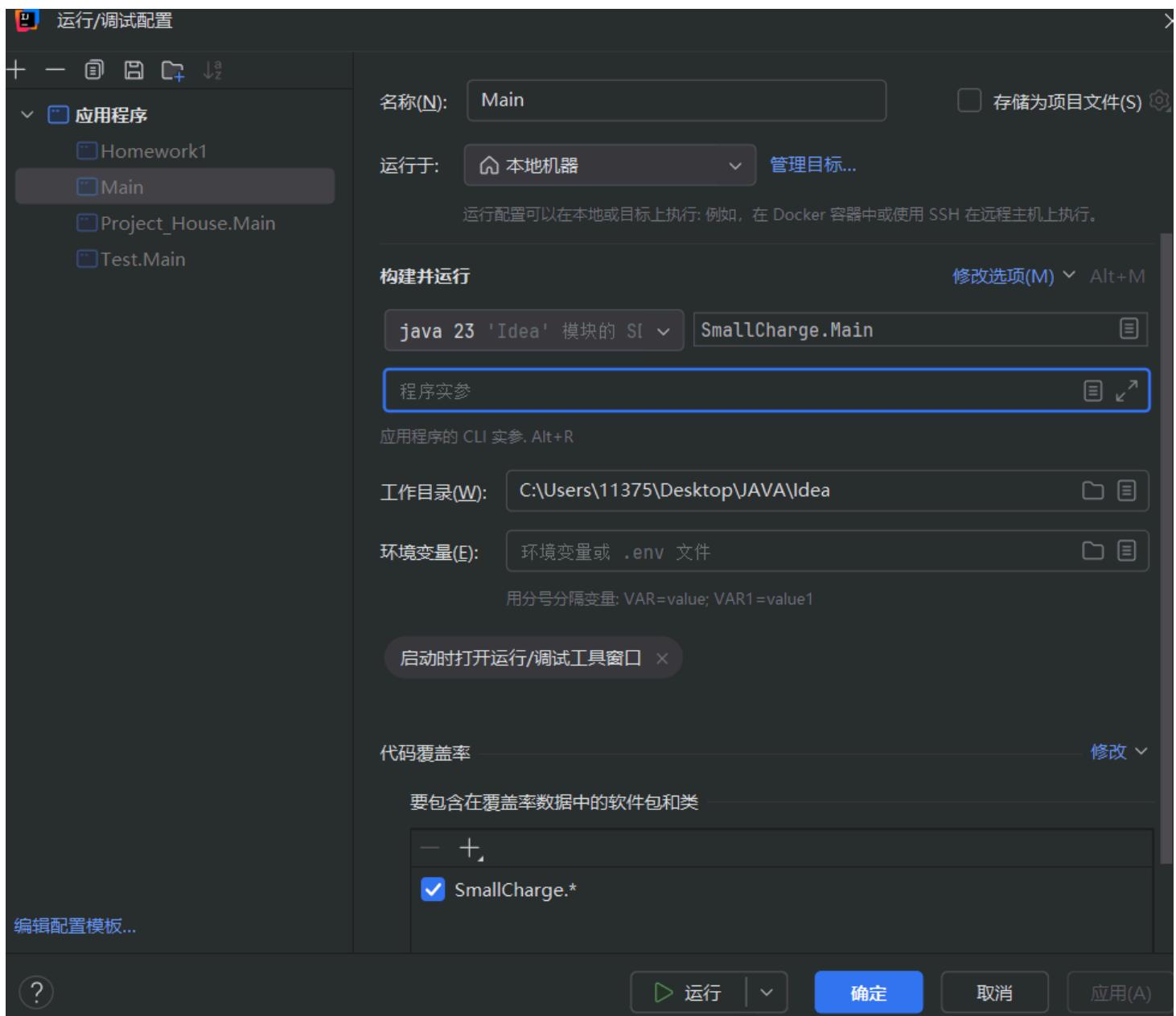


#### 2. 注意事项:

1. 可以直接调用 main 方法所在类的静态方法或静态属性。
2. **不能直接访问该类中的非静态成员**，必须创建该类的一个实例对象后，才能通过这个对象去访问类中的非静态成员(毕竟main也是静态方法)

### 3. main动态传值(在IDEA中传递参数给main)





## 3.2 类变量与类方法

### 1. 类变量(静态变量/静态属性, 通过static的共享特性实现):

1. 示例:

```
class Child { //类
    private String name;
    //定义一个变量 count ,是一个类变量(静态变量) static 静态
    //该变量最大的特点就是会被 Child 类的所有的对象实例共享
    public static int count = 0;
    //写成static private的顺序也不是不行
```

```
//static变量是同一个类所有对象共享  
//在类加载时初始化生成,随类消亡而销毁  
2. 访问(必须满足访问修饰符的权限与范围):  
    类名.类变量名  
    对象名.类变量名
```

## 2.类方法(静态方法,类比类变量):

1. 特点:
  - i. 类方法(静态方法)才能直接访问静态属性/变量(**都属于类本身,而非某一个实例**)
  - ii. 常用于工具类,不涉及任何和对象相关的成员(**不需要创建对象就可以直接作为工具使用**)
  - iii. 与普通方法一样都随类加载而加载,结构信息储存在方法区
  - iv. 类方法不能使用与对象有关(类方法**不绑定对象而是绑定类**)的关键字,例如this和super等会访问实例对象属性的关键字
  - v. **普通成员方法可以访问非静态成员和静态成员**
  - vi. 可以通过**类名/对象名.类方法名**访问  
**//静态绑定类,对象是类的一个实例,所以对象可以改变使用静态,但是静态不能访问非静态的对象**(在满足访问权限的基础上)
    - (1) 静态方法, 只能访问静态成员
    - (2) 非静态方法, 可以访问所有的成员
    - (3) 在编写代码时, 仍然要遵守访问权限规则

## 3.3 代码块

### 1. 理解:

代码块(初始化块):

是类中的成员(类的一部分),类似于方法(但是没有方法名,没有返回,没有参数,不用通过对象或类调用,而是在类加载/创建对象时自动调用,仅仅是一个方法体,一个字面意思的代码块),将逻辑语句封装在方法体中,用{}包围起来

### 2. 语法:

```
修饰符 {  
    代码  
};  
//修饰符只有static(静态代码块和非静态代码块), ;可以省略
```

### 3. 益处:

1. 相当于另一种形式的构造器(补充构造),可以做初始化操作
2. 提高代码重用性

### 4. 注意事项(有关new的顺序):

1. static代码块既然是静态的,就也随类加载而初始化(**static绑定类**),且只会执行一次(普通代码块就随对象创建而执行),且只能直接调用静态属性/方法(**普通代码块可以调用任意成员,同静态方法与普通方法**)
2. 类的加载时候:
  - i. 创建实例时(new)
  - ii. 创建子类对象实例时(父类也会加载)
  - iii. 使用类的静态成员时(绑定绑定)
3. 普通代码块在创建对象实例时调用一次(**绑定对象实例**),仅仅**使用类的静态成员时不会执行**(这是加载类,又不是加载实例对象,当然不执行嘞)
4. 创建对象时(new),类的调用顺序:
  - i. 调用静态代码块和静态属性初始化(按顺序调用)
  - ii. 调用普通代码块和普通属性的初始化(按顺序使用)
  - iii. 调用构造方法
5. 构造器最前面隐含 \*\*super() (加载父类)\*\* 和 **调用普通代码块**,静态代码块与静态属性在类加载时就已经执行完毕,在构造器之前
6. 继承:

- ① 父类的静态代码块和静态属性(优先级一样, 按定义顺序执行)
- ② 子类的静态代码块和静态属性(优先级一样, 按定义顺序执行)
- ③ 父类的普通代码块和普通属性初始化(优先级一样, 按定义顺序执行)
- ④ 父类的构造方法
- ⑤ 子类的普通代码块和普通属性初始化(优先级一样, 按定义顺序执行)
- ⑥ 子类的构造方法 // 面试题

AAAAA extends BBBBB 类 演示 [ 10Min ] 55 **CodeBlockDetail04.java**

## 特殊章节(个人总结之static,联系3.2与3.3)

1. 类中的静态是绑定类的,与类共生死
2. 非静态是绑定对象实例的,与实例共生死
3. 对象实例是按照类的基础创建的,所以**绑定对象实例的非静态可以访问绑定类的静态**
4. 注意,就算是父类和子类,也是**先把父子的静态执行完了再进行父子的非静态和构造器执行**  
//吐槽一下,就这些东西绕来绕去的

## 3.4 类的单例设计模式(饿汉式与懒汉式)

### 1. 理解:

采取一定的方法在整个系统中,对某个类只能存在一个对象实例,且该类只提供一个取得对象实例的方法,整个系统可以通过一个全局访问点,单例类的实例化过程线程安全

(所以不用new所想保留的对象,所以要用static,因为只有单个实例,而且在外部使用该类时,所创建的对象实际上是一种指向,借用该对象使用方法来返回实际保留的单例对象)

- 1) 构造器私有化 =》 防止直接 new
- 2) 类的内部创建对象
- 3) 向外暴露一个静态的公共方法。 getInstance

### 2. 饿汉式与懒汉式

1. 饿汉式:类加载时就创建实例(适用于实例化耗时不多的情况)

```
class Cat{  
    private String name;  
    public static int n1 = 100;  
    private static Cat cat = new Cat("小白");  
    //类加载时是单线程的,且类加载只发生一次,所以不会递归  
    private Cat(String name) {  
        System.out.println("构造器被调用");  
        this.name = name;  
    }  
    public static Cat getInstance() {  
        return cat;  
    }  
}
```

在多线程环境中,如果多个线程同时访问 getInstance() 方法,可能会导致 cat 被多次创建。为了解决这个问题,可以使用synchronized 关键字或者静态内部类来实现线程安全的单例

```
public static synchronized Cat getInstance() {  
    if (cat == null) {  
        cat = new Cat("小白");  
    }  
    return cat;  
}
```

2. 懒汉式:只有在需要时才创建实例(线程不安全)

```

class Cat{
    private String name;
    public static int n1 = 100;
    private static Cat cat;
    private Cat(String name) {
        System.out.println("构造器被调用");
        this.name = name;
    }
    public static Cat getInstance() {
        if(cat == null) {
            cat = new Cat("小可爱");
        }
        return cat;
    }
}

```

## 3.5 final关键字

### 1.理解:

字面意思,被修饰的类/属性/方法/局部变量 不会被继承/重写/修改(final,指代最终所确定的值,所以不会被改变)

例如:

final class A{...}//不被继承(一般类是final就不用把方法也变成final)

public final void hi(...){...}//不被重写,但是不代表不能被继承(类不是final就行)

public final double weLove = 1314.520;//不被修改

### 2.注意事项:

1. 被final修饰的基本数据属性不会变,所以可以叫常量(用xx\_xx\_xx命名),但是引用类型数据可变,如:

引用数据类型, `final` 修饰意味着这个引用变量不能再指向其他对象, 但对象本身的内容是可以改变的。例如, `final StringBuilder sb = new StringBuilder("Hello");`, 不能让 `sb` 再指向其他 `StringBuilder` 对象, 但可以通过 `sb.append(" World");` 来修改字符串的内容。

2. 因为不能修改,所以必须要有初值(定义赋值,代码块赋值,构造器赋值,final的静态属性不能在构造器赋值/静态属性绑定类而不绑定对象的构造嘛,而且因为静态的new在构造器之前,而且若在构造器里面可能会赋不同的值,就违反了其规则)

3. final不能修饰构造器

4. final往往与static搭配使用,不会导致类加载

5. 包装类(Integer,Double,Boolean等都是final),String是final类

6. final类虽然不能继承,但是可以实例化对象(比如上面的String str = ... 😊)

## 3.6 抽象类(模板设计模式)

### 1. 理解:

有些方法需要声明,但是不确定如何实现时(比如需要继承作不同的用法,但是父类本身的方法没啥用),可以用抽象方法,整个类就是抽象类  
(也就是说所谓抽象方法就是没有实现的方法,作为跳板由子类继承实现)

### 2. 使用语法:

```
abstract class Animal{  
    private String name;  
    public Animal(String name) {  
        this.name = name;  
    }  
    public abstract void eat(); //就是这句  
}
```

### 3. 注意事项/细节:

1. **抽象类不能被实例化(因为不能执行所有的操作,实例对象是不完整的)**
2. 抽象类可以没有抽象方法(虽然这也太抽象了,感觉没啥用,不过这样子还可以用来封装抽象类里面的方法,还能让程序员被迫写继承来保存模板的一致性,因为没法实例化嘛),**有抽象方法的一定是抽象类(而且必须用abstract声明)**
3. 一个类继承了抽象类,就必须实现其所有抽象方法,不然因为继承的特性,会使得该子类依旧不完整(除非该子类也是抽象类)
4. **abstract只能修饰类和方法**
5. 抽象方法不能写大括号(主体),因为它不能实现
6. 抽象方法不能用private/final/static修饰(因为有了这几个就不能重写了呀,就一直不完整了)
7. 对于继承的子类中的继承抽象方法:

```
@Override 2个用法  
public void work() {  
    |  
}
```

在Java中，`@Override`注解用于明确指出某个方法声明打算重写基类中的另一个方法声明。这是一个标记，告诉编译器：“这个方法应该是一个重写的方法”。编译器会检查这个注解，并确保以下两点：

- 1. 存在性：**在父类或实现了接口中确实存在一个匹配的方法声明。
- 2. 一致性：**方法的签名（方法名、参数列表）和访问权限与被重写的方法一致。

如果这些条件不满足，编译器将报错，这有助于避免因拼写错误或参数类型不匹配等问题导致的重写错误。

```
//@Override便于阅读和查错
```

#### 4. 获取任务完成时间(效率)

```
public abstract void job(); //抽象方法
public void calculateTime() { //实现方法，调用 job 方法
    long start = System.currentTimeMillis(); //得到开始的时间
    job(); //动态绑定机制
    long end = System.currentTimeMillis(); //得的结束的时间
    System.out.println("任务执行时间 " + (end - start));
}
```

```
//用long让时间更精确
```

## 3.7 接口

### 1. 理解：

接口就是ultra抽象的抽象类,所有方法都是抽象方法,即没有方法体

(特别说明:以上仅限于jdk8.0之前,jdk8.0及之后接口类可以有静态方法,默认方法,也就是说随时更新,其实也不是不能有方法的具体实现)

## 2. 接口创建/连接/使用过程示例:

### 1. 创建接口(创建usb):

```
public interface Usb{ //接口
    public void start();
    public void stop();
}
```

### 2. 连接接口(将数据传入usb中,类似于继承,但是语法不同):

```
public class Camera implements Usb{//实现接口,把接口方法实现
//其实需要的话,可以这么写class D extends C implements B,A {...}
    @Override
    public void start() {
        System.out.println("相机开始工作...");
    }
    @Override
    public void stop() {
        System.out.println("相机停止工作....");
    }
}
```

### 3. 创建接收接口(接收usb数据)

```
public class Computer {
    public void work(Usb usb){
        usb.start();
        usb.stop();
    }
}
```

### 4. 将作为接口的数据与接收端连接

```
//在主函数中
Camera camera = new Camera();
Computer computer = new Computer();
computer.work(camera);
```

## 3. 使用注意事项和细节

1. 接口不能实例化(喵的连抽象类都不行,接口不是更不行吗,理由相同,类创建的对象不完整)

2. 接口中所有方法都是**public**,其抽象方法可以不用**abstract**修饰(因为全是,所以省略了,不代表不存在)
3. 同抽象类,与接口相接,就必须将其所有方法实现(否则不完整,除非是同样不完整的抽象类与其相接)
4. 一个类可以和多个接口相接(但是**抽象类**一次只能继承一个,还是有区别的),这样写就行:

```
class Pig implements IB,IC {  
    ...}
```

5. 但是接口不能继承其他类,不过可以继承多个别的接口(套娃是吧)
6. 接口中的属性只能是常数,也就是**final**,而且还得是**public static final**这么一串修饰符(当然也就必须得初始化),既然只能是这个,那也是可以省略的嘞  
(不能实例化,所以要静态**static**)
7. 接口自身的修饰符只能是**public** 和 默认,
8. 访问接口属性语法:**接口名.属性名**
- 9.

## 4. 与继承比较:

1. 继承价值:解决代码复用性和可维护性  
    接口价值:事先设计好规范(方法),用其它类实现
2. 接口比继承更灵活,继承是is-a(继承,一个是另一个的特例)关系,接口是like-a(接口,一个实现了另一个的规范,实现规范≠继承)关系
3. 一定程度实现实现代码解耦(减少各组件间的依赖关系,更容易理解测试重用)

## 5. 多态特性:

1. 多态参数(与接口相接的数据都可以**以接口类型作为参数被接收**,反过来说,**接口类型的变量可以指向实现接口的类的对象**,类比父类和子类)
2. 多态数组(类比父类和子类)//依旧是动态绑定
3. 多态传递(接口继承另一个接口的体现)

## 3. 内部类

### 1. 理解:

字面意思,类中类,真套娃

### 2. 语法(没有语法,直接写里面就行)

### 3. 分类(1) 局部内部类和匿名内部类(定义在局部位置上,方法内/方法参数/代码块中)

#### 1. 局部内部类(有类名)

特点:

1. 可以看成是以类来当成局部变量
2. 可以直接访问外部类的所有成员,包括私有!!
3. 不能添加访问修饰符(因为作为局部变量,所以不能用),不过可以用final修饰
4. \*\*外部类与内部类重名时,遵循就近原则,\*\*若访问外部类成员:**外部类名.this.成员**(访问语法为重点!)

**4. 局部内部类---访问---->外部类的成员 [访问方式: 直接访问]**

**5. 外部类---访问---->局部内部类的成员**

**访问方式: 创建对象, 再访问(注意: 必须在作用域内)**

**记住:(1)局部内部类定义在方法中/代码块**

**(2) 作用域在方法体或者代码块中**

**(3) 本质仍然是一个类**

**6. 外部其他类---不能访问---->局部内部类 ( 因为 局部内部类地位是一个局部变量)**

## 2. 匿名内部类(重点,字面意思,无类名)

### 1. 概述:

**//(1) 本质是类(2) 内部类(3) 该类没有名字  
(4)同时还是一个对象**

**说明: 匿名内部类是定义在外部类  
的局部位置, 比如方法中, 并且没有类名**

### 1. 匿名内部类的基本语法

```
new 类或接口(参数列表){  
    类体  
};
```

### **【案例演示 AnonymousInnerClass.java】**

```
//其实系统会自动分配名字  
//匿名内部类用来解决那些只想调用一次的类就不再调用的问题(不用繁琐的再创建一个新的类去调用)  
//记得临时创建完对象的后面加分号,可看示例
```

### 2. 示例(在方法里面):

i. 基于接口的匿名内部类:

```
class Outer04{
    ...
    public void method(){
        ...
        IA tiger = new IA() {
            @Override
            public void cry() {
                System.out.println("老虎叫唤...");
            }
        }; //记得加分号
        System.out.println("tiger 的运行类型=" + tiger.getClass()); //Outer04$1
        tiger.cry();
        ...
    }
    ...
}
```

//此处IA为接口名,cry()为接口内所有方法

//此处相当于通过匿名内部类的语法临时创建了该对象并调用方法

//tiger的**编译类型为IA,运行类型为匿名内部类,Outer04\$1(系统分配名)**

//底层:

```
class Outer04$1 implements IA {
    @Override
    public void cry() {
        System.out.println("老虎叫唤...");
    }
}
```

使用示例:

```
public interface Calculator {//接口
    public double work(double d1, double d2);
}
...
public class Cellphone{//类
    ...
    public void test(Calculator cal,double d1,double d2){
        double result = cal.work(d1,d2);
        System.out.println("结果为"+result);
    }
    ...
}
...
public static void main(String[] args){//在主函数实现
    Cellphone cellphone = new Cellphone();
    double d1 = 1.0;
    double d2 = 520.0;
    cellphone.test(new Calculator() {
        @Override
        public double work(double d1, double d2) {
            return d1+d2;
        }
    },d1,d2); //记得加分号
}
```

## 2. 基于类的匿名内部类:

```
class Outer04{
    ...
    public void method(){
        ...
        Father father = new Father("jack"){
            @Override
            public void test() {
                System.out.println("匿名内部类重写了 test 方法");
            }
        }; //记得加分号
        System.out.println("father 对象的运行类型=" + father.getClass()); //Outer04$2
        father.test();
        ...
    }
    ...
}
```

//底层:

```
class Outer04$2 extends Father{
    @Override
    public void test() {
        System.out.println("匿名内部类重写了 test 方法");
    }
}
```

### 3. 基于抽象类的匿名内部类:

```
abstract class Animal { //抽象类
    abstract void eat();
}

...
class Outer04{//类

    ...
    public void test(){
        ...
        Animal animal = new Animal(){//匿名内部类基于抽象类实现
            @Override
            void eat() {
                System.out.println("小狗吃骨头...");
            }
        }; //记得加分号
        animal.eat();
        ...
    }
    ...
}
```

### 3. 特点:

- i. 匿名内部类**既是类的定义,本身又是一个对象(临时)**,所以语法比较特殊
- ii. 同样不能添加访问修饰符,因为匿名内部类地位上作为一个**以类/抽象类/接口创建的临时对象的局部变量**  
**(所以可以直接访问外部类所有成员,包括私有的)**  
**(所以外部类不能访问匿名内部类)**
- iii. \*\*外部类与内部类重名时,遵循就近原则,\*\*若访问外部类成员:**外部类名.this.成员**

### 4. 最佳实践(**当作实参直接传递**)

```

interface IL { //接口
    void show();
}

...
public static void main(String[] args){ //在主函数中使用时实现该匿名内部类
    f1(new IL() {
        @Override
        public void show() {
            System.out.println("I love you ~");
        }
    }); //里面的实参就是匿名内部类,其实就是 f1(匿名内部类);
    public static void f1(IL il) { //形参是接口类型
        il.show(); //也可以通过创建别的类的对象实现该函数
    }
}

```

### 3. 分类(2) 成员内部类和静态内部类(定义在成员位置上,在方法外)

#### 1. 成员内部类(没有static修饰)

1. 特点:

- i. 同样可以访问外部类所有成员,包括私有的
- ii. 可添加任意访问修饰符(public,protected,private,默认)  
(因为其就可以当作是类的一个成员)
- iii. 作用域:与外部类其他成员一样,在整个类体中
- iv. 访问权限:

**4. 成员内部类---访问---->外部类成员(比如:  
属性) [访问方式: 直接访问] (说明)**

**5. 外部类---访问----->成员内部类 (说明)  
访问方式: 创建对象, 再访问**

**6. 外部其他类---访问---->成员内部类**

- v. \*\*外部类与内部类重名时,遵循就近原则,\*\*若访问外部类成员:**外部类名.this.成员**(这玩意怎么已经出现这么多次了,好像都是这样的)

2. 使用的三种方式:

- i. 直接当成外部类成员访问:

```

public static void main(String[] args) {
    ...
    Outer outer = new Outer();
    Outer.Inner inner = outer.new Inner();
    ...
}

```

ii. 在外部类编写方法返回内部类对象:

```

public static void main(String[] args) {
    ...
    Outer outer = new Outer();
    Outer.Inner inner = outer.getInner();
    ...
}

class Outer{
    ...
    private int n1 = 10;//与成员内部类重名属性
    private String name = "张三";//不与内部类重名属性,内部类可以直接访问
    public class Inner{//成员内部类
        ...
        private int n1 = 66;//与外部类重名属性
        public void say() {
            System.out.println("n1 = " + n1 + " name = " + name + " 外部类的 n1=" + Outer.this.
                hi();
        }
    }
    public Inner getInner(){
        return new Inner();
    }
    ...
}

```

iii. 在外部类创建方法使用内部类的方法:

在外部类的方法中new内部类的对象,通过该对象使用内部类的方法

## 2. 静态内部类(有static修饰,名字都写着呢)

1. 特点:

- i. 可以直接访问外部类所有静态成员,包括私有的,但是它是静态的,所以当然不能访问非静态的嘞
- ii. 可添加任意访问修饰符(public,protected,private,默认)  
(因为其就可以当作是类的一个成员)
- iii. 作用域:与外部类其他成员一样,在整个类体中

iv. 访问权限:

- 4. 静态内部类---访问---->外部类(比如: 静态属性) [访问方式: 直接访问所有静态成员]
- 5. 外部类---访问----->静态内部类 访问方式: 创建对象, 再访问

## 6. 外部其他类---访问---->静态内部类

v. \*\*外部类与内部类重名时,遵循就近原则,\*\*若访问外部类成员:**外部类名.this.成员**(结果这么四个内部类全是这样的,无语了)

2. 使用:

- i. 满足访问权限的情况下,直接通过类名访问(同成员内部类)
- ii. 在外部类编写方法返回内部类对象实例(同成员内部类)
- iii. 在外部类创建方法使用内部类的方法(同成员内部类)

# 三.JAVA使用基础

## 3.1 枚举和注解

### 1. 理解枚举(enum,enumeration):

- 1. 枚举是一组**常量**的集合
- 2. 可以把枚举当作一种特殊的类, **只包含一组 特定的 , 有限的 对象**

### 2. 枚举的两种实现和

#### 1. 自定义类实现枚举(往类里面全塞常量)

- 1. 特点:
  - i. 不需要提供改变属性的方法,因为**枚举对象通常只读**
  - ii. 枚举对象/属性用**final + static**共同修饰
  - iii. **枚举对象名通常大写**(常量的命名规范)
  - iv. 枚举对象**可以有多个属性**
  - v. 1) 构造器私有化
  - 2) 本类内部创建一组对象[四个 春夏秋冬]
  - 3) 对外暴露对象 (通过为对象添加 **public final static** 修饰符)
  - 4) 可以提供 **get** 方法, 但是不要提供 **set**
- 2. 示例:

```
public static void main(String[] args) {
    ...
    System.out.println(Season.AUTUMN); //直接调用输出
    System.out.println(Season.SPRING);
    ...
}

...
class Season { //类
    private String name;
    private String desc; //描述
    public static final Season SPRING = new Season("春天", "温暖");
    public static final Season WINTER = new Season("冬天", "寒冷");
    public static final Season AUTUMN = new Season("秋天", "凉爽");
    public static final Season SUMMER = new Season("夏天", "炎热");
    private Season(String name, String desc) { //将构造器私有化,防止被new
        this.name = name;
        this.desc = desc;
    }
    public String toString() {
        return "name=" + name + ", desc=" + desc; //改变输出格式,否则输出地址
    }
}
```

//1. 将构造器私有化,目的防止 直接 new

//2. 去掉 setXxx 方法, 防止属性被修改

//3. 在 Season 内部, 直接创建固定的对象

//4. 优化, 可以加入 final 修饰符

## 2. 使用enum关键字实现枚举

1. 示例:

```

enum Season2 {
    SPRING("春", "暖"), SUMMER("夏", "热"), AUTUMN("秋", "凉"), WINTER("冬", "冷"), SEASON, SEASON();
    //可以直接用,等价于public static final SPRING = new Season("春", "暖")
    //用逗号相隔
    //enum的枚举必须写在第一行
    //可以用以下定义的无参构造器构造常数,但是若改变了toString,会按其无参输出null
    private String name;
    private String desc;
    private Season2(){}
    private Season2(String name, String desc){
        this.name=name;
        this.desc=desc;
    }
    public String getName() {
        return name;
    }
    public String getDesc() {
        return desc;
    }
    public String toString() {//否则默认输出常量名
        return name + " " + desc;
    }
}

```

## 2. 注意事项(还有部分在示例中):

- i. 用enum关键字开发枚举类时(该枚举类还是final类),会默认继承Enum类(所以不能继承其他类)  
(但是可以实现接口,如下面所提)

## 3. Enum类常用方法(语法:枚举对象.方法名()):

- i. **toString**:Enum类已重写过,返回当前对象名,子类可以重写该方法,返回对象的属性信息等
- ii. **name**: 返回当前对象名 (常量名), 子类中不能重写
- iii. **ordinal**: 返回当前对象的位置号, 默认从 0 开始
- iv. **values**: 返回当前枚举类中所有的常量(返回**枚举类数组**)
- v. **valueOf**: 将字符串转换成枚举对象, 字符串必须为已有的常量名, 否则报异常(更像是查找一样的感觉,看看有没有叫这个名字的value,但是没找到就会报错)
- vi. **compareTo**: 比较两个枚举常量(**比较编号是否相同**)

## 3.用enum实现接口

### 1. 形式: enum 类名 implements 接口 1, 接口 2{}

### 2. 示例:

```
interface IPlaying {
    public void playing();
}

enum Music implements IPlaying {
    CLASSICMUSIC;
    @Override
    public void playing() {
        System.out.println("播放好听的音乐...");
    }
}
```

### 3. 注解(元数据):

#### 1.理解:

比如前面的@Override,是一个类似于提示符一样的东西,就像是括号里面的补充说明一样(在个人使用情况下作用不是很明显,但是在大型项目中就很凸显作用)

#### 2.三种常见的注解:

1. **@Override**:注解该方法为重写方法,会额外检查一下
2. **@Deprecated**:表示某元素(类,方法等)已过时(就是不推荐用,但也不是不行,可以做版本升级兼容过渡使用),可以修饰方法/类/字段/包/参数...

**2. 可以修饰方法, 类, 字段, 包, 参数 等等**

**3. @Target(value={CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})**

3. **@SuppressWarnings**:抑制编译器警告,抑制范围与放置位置(具体语句,方法,类,包括main类)相关
  - i. 使用语法:

```
@SuppressWarnings({"rawtypes", "unchecked", "unused"})
public class SWarnings{...}
//抑制范围就是整个SWarnings类
//在{}中写入想抑制的警告信息
```

- ii. 警告信息:

all, 抑制所有警告

boxing, 抑制与封装/拆装作业相关的警告

//cast, 抑制与强制转型作业相关的警告

//dep-ann, 抑制与淘汰注释相关的警告

//deprecation, 抑制与淘汰的相关警告

//fallthrough, 抑制与 switch 陈述式中遗漏 break 相关的警告

//finally, 抑制与未传回 finally 区块相关的警告

//hiding, 抑制与隐藏变数的区域变数相关的警告

//incomplete-switch, 抑制与 switch 陈述式(enum case)中遗漏项目相关的警告

//javadoc, 抑制与 javadoc 相关的警告

//nls, 抑制与非 nls 字串文字相关的警告

//null, 抑制与空值分析相关的警告

//rawtypes, 抑制与使用 raw 类型相关的警告

//resource, 抑制与使用 Closeable 类型的资源相关的警告

//restriction, 抑制与使用不建议或禁止参照相关的警告

//serial, 抑制与可序列化的类别遗漏 serialVersionUID 栏位相关的警告

//static-access, 抑制与静态存取不正确的警告

//static-method, 抑制与可能宣告为 static 的方法相关的警告

//super, 抑制与置换方法相关但不含 super 呼叫的警告

//synthetic-access, 抑制与内部类别的存取未最佳化相关的警告

//sync-override, 抑制因为置换同步方法而遗漏同步化的警告

//unchecked, 抑制与未检查的作业相关的警告

//unqualified-field-access, 抑制与栏位存取不合格相关的警告

//unused, 抑制与未用的程式码及停用的程式码相关的警告

iii. 常用:

- 1) **unchecked** 是忽略没有检查的警告
- 2) **rawtypes** 是忽略没有指定泛型的警告(传参时没有指定泛型的警告错误)
- 3) **unused** 是忽略没有使用某个变量的警告错误
- 4) **@SuppressWarnings** 可以修饰的程序元素为, 查看@Target
- 5) 生成@SuperssWarnings 时, 不用背, 直接点击左侧的黄色提示, 就可以选择(注意可以指定生成的位置)

### 3. 四种元注解(修饰注解的注解)与定义注解的关键字(@interface):

#### 1. @Target:(元注解)指定注解的使用范围(在哪)

➤ **@Target的源码说明**

```
@Documented
@Retention(RetentionPolicy.RUNTIME) // 它的作用范围是 RUNTIME
@Target(ElementType.ANNOTATION_TYPE) // 这里的ANNOTATION_TYPE 说明@Target 只能修饰注解
public @interface Target { //说明它是注解
    /**
     * Returns an array of the kinds of elements an annotation type
     * can be applied to.
     * @return an array of the kinds of elements an annotation type
     * can be applied to
     */
    ElementType[] value(); //可以简单看一下ElementType 的取值 // 通过Enum 比如 : TYPE等
}
```

**@Target(value={CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})**

#### 2. @Retention:(元注解)指定注解的作用范围(能影响哪)

- i. 使用语法:@Retention(RetentionPolicy.SOURCE)
- ii. 三种范围:SOURCE(编译器使用后就丢弃),CLASS(JVM不保留注解,编译器把注解记录在class文件中),RUNTIME(JVM会保留注解,编译器把注解记录在class文件中,本身意为运行时)

#### 3. @Documented:(元注解)指定注解是在javadoc体现(就是生成文档时能看到这个注解),定义为 Documented的注解必须设置Retention值为RUNTIME

➤ 看一个案例

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE,
PARAMETER, TYPE})
public @interface Deprecated { //一个Deprecated 注解@Documented, 则javadoc会看到
Deprecated}
```

#### 4. @Inherited:(元注解)子类会继承父类注解,就是将注解给继承到子类去

#### 5. @interface:定义注解的关键字

例如@Override源码为:

```
public @interface Override {
}
```

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

#### 4. 常用开发注解:

1. 自带:

@Override: 检查重写  
@NotNull: 确保值不为null  
@NotEmpty: 确保字符串、集合不为空  
@Size: 限制字符串长度或集合大小  
@Min/@Max: 限制数值范围  
@Email: 验证邮箱格式  
@Pattern: 使用正则表达式验证  
@Validated: 数据校验  
@RequestBody: 接收入参  
@Cacheable: 用于将方法的返回结果缓存起来  
@CacheEvict: 用于从缓存中移除一个或多个缓存项

```
1 @Override
2 @Cacheable(cacheNames = "models",key = "#roleId",unless = "#r
3     public Models queryById(Serializable roleId) {
4         return getById(roleId);
5     }
6     @CacheEvict(cacheNames = "models", key = "#models.roleId")
7     @Override
8     public boolean updateModels(Models models) {
9         return updateById(models);
10    }
```

当 `updateModels` 方法被调用并成功更新数据库中的数据时，Spring 会从名为 "models" 的缓存中移除一个键为 `models.roleId` 的缓存项。

这两个注解一起工作，确保了在查询时可以快速从缓存中获取数据，而在更新数据时能够及时清除缓存，从而避免缓存数据与实际数据不一致的问题。

## 2. Lombok库:

- @Getter/@Setter: 自动生成getter和setter方法
- @ToString: 自动生成toString方法
- @EqualsAndHashCode: 自动生成equals和hashCode方法
- @Data: 包含@Getter, @Setter, @ToString, @EqualsAndHashCode
- @Builder: 生成建造者模式代码

```
//数据更新操作  
Models build = Models.builder()  
    .roleId(roleId)  
    .isMasterWork(status)  
    .build();  
return modelsService.updateModels(build);
```

@NoArgsConstructor/@AllArgsConstructor: 生成构造函数

## 3.2 异常

### 1. 理解(这还要介绍?)

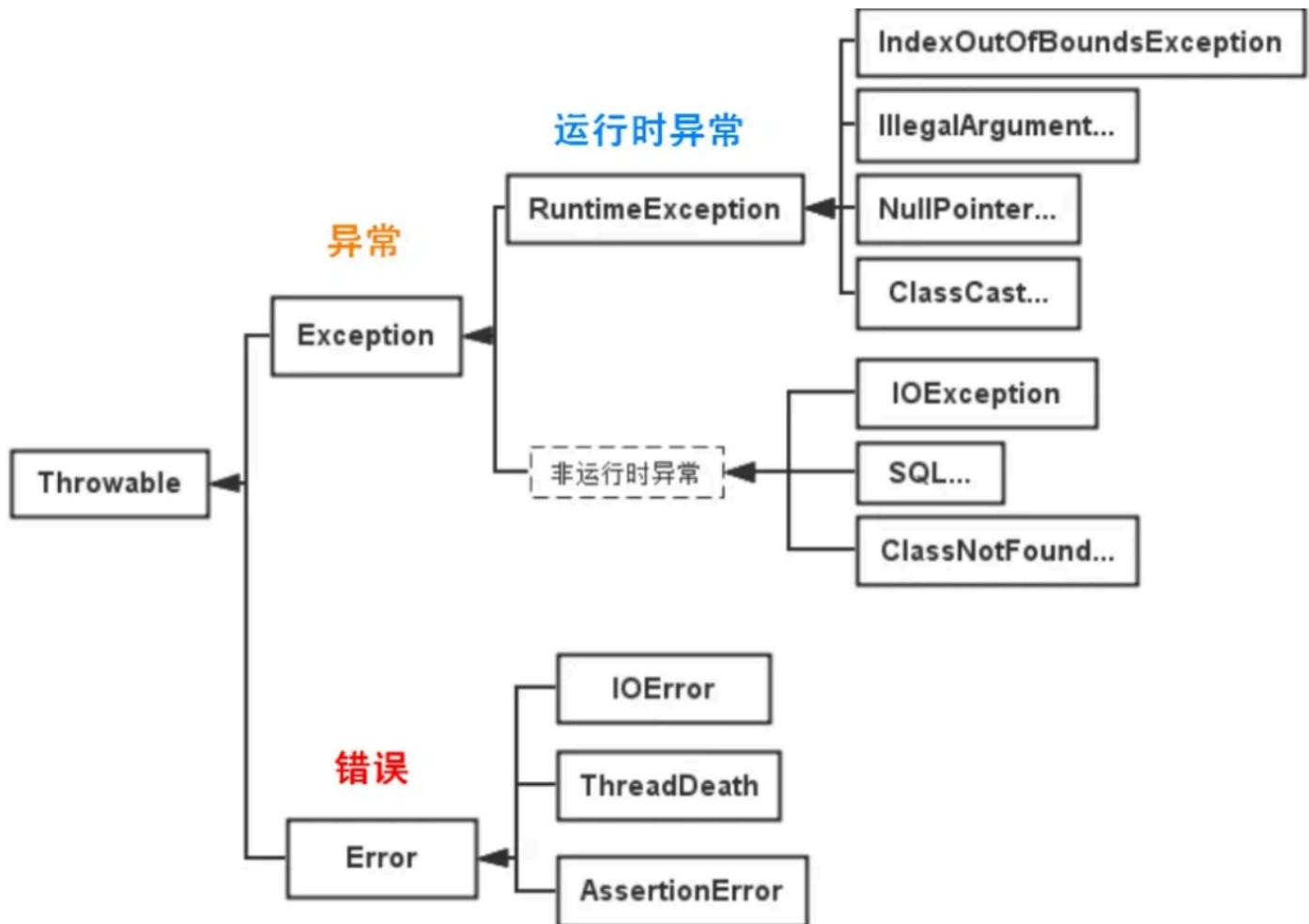
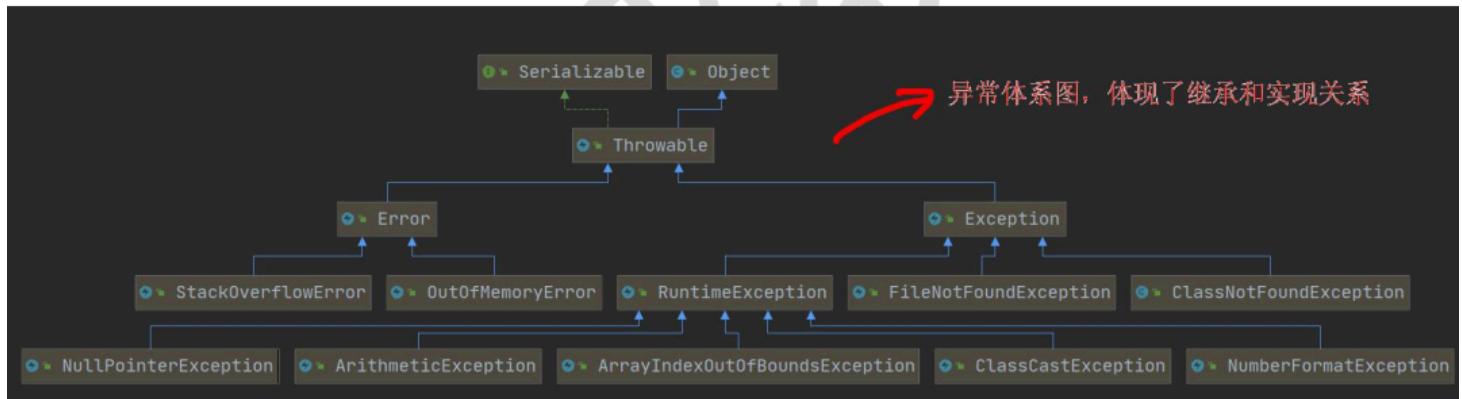
### 2. 异常总体系分类:

执行过程中所发生的异常事件可分为两大类

**Error(错误):** Java虚拟机无法解决的严重问题。如：JVM系统内部错误、资源耗尽等严重情况。比如：StackOverflowError[栈溢出]和OOM(out of memory)，Error是严重错误，程序会崩溃。

**Exception:** 其它因编程错误或偶然的外在因素导致的一般性问题，可以使用针对性的代码进行处理。例如空指针访问，试图读取不存在的文件，网络连接中断等等，Exception分为两大类：**运行时异常**[程序运行时，发生的异常]和**编译时异常**[编程时，编译器检查出的异常]。

### 12.4.1 异常体系图



### 3. 常见运行异常(编译没错,运行出错,就是字没写错,但是有语病):

1. `NullPointerException` 空指针异常

例:

```
String name = null;  
System.out.println(name.length());
```

```
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "String.length()" because "name" is null
        at Test.Main.main(Main.java:6)
```

## 2. ArithmeticException 数学运算异常

例:

```
int num = 10/0;
```

```
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
Exception in thread "main" java.lang.ArithmaticException Create breakpoint : / by zero
        at Test.Main.main(Main.java:6)
```

## 3. ArrayIndexOutOfBoundsException 数组下标越界异常

例:

```
int[] num ={1,3,1,4};
for(int i = 0; i <= num.length; i++){
    System.out.println(num[i]);
}
```

```
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
1
3
1
4
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException Create breakpoint :
    Index 4 out of bounds for length 4
        at Test.Main.main(Main.java:8)
```

## 4. ClassCastException 类型转换异常

例:

```
A a = new B();
B b = (B)a;
C c = (C)b;
```

```
class A{} 3个用法 2个
class B extends A{}
class C extends A{}
```

```
Exception in thread "main" java.lang.ClassCastException Create breakpoint  
at com.hspedu.exception_.ClassCastException_.main(ClassCastExcepti
```

## 5. NumberFormatException 数字格式不正确异常

例：

```
String name = "love";  
int num = Integer.parseInt(name);
```

```
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8  
Exception in thread "main" java.lang.NumberFormatException Create breakpoint : For  
input string: "love"  
at java.base/java.lang.NumberFormatException.forInputString  
(NumberFormatException.java:67)  
at java.base/java.lang.Integer.parseInt(Integer.java:588)  
at java.base/java.lang.Integer.parseInt(Integer.java:685)  
at Test.Main.main(Main.java:6)
```

## 4. 常见编译异常(直接就是字本身都写错了)

- ✓ **SQLException** //操作数据库时，查询表可能发生异常
- ✓ **IOException** //操作文件时，发生的异常
- ✓ **FileNotFoundException** //当操作一个不存在的文件时，发生异常
- ✓ **ClassNotFoundException** //加载类，而该类不存在时，异常
- ✓ **EOFException** // 操作文件，到文件末尾，发生异常
- ✓ **IllegalArgumentException** //参数异常

//大多为SQL和文件编程

## 5. 异常处理机制语法:

### 1. try-catch-finally(捕获异常,需要人工处理)

```
try{
    ...
    //可能错误的语句,异常对象传递给catch
    //出现异常语句就直接跳过剩下的语句到catch或者finally
    ...
} catch (ArithmaticException e) { //若有异常,则捕获错误信息
    System.out.println("算术异常=" + e.getMessage()); //可以有多个catch捕获不同异常,但是如果发生异常,只执行一个
} catch (Exception e){ //这一句包含所有的异常
    e.printStackTrace();
}
finally{
    ...//不论try是否有异常都会执行(所以通常为释放资源的代码)
    //finally可以没有,但是try-catch必须有
    //如果finally里面会return xxx,那么会因为强制执行finally语句而return该语句里面的东西
    //就算上面已经return了,finally里面没return,也要强制执行finally的语句
}
```

示例:

```
try {
    int res = 1314/0;
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

/ by zero

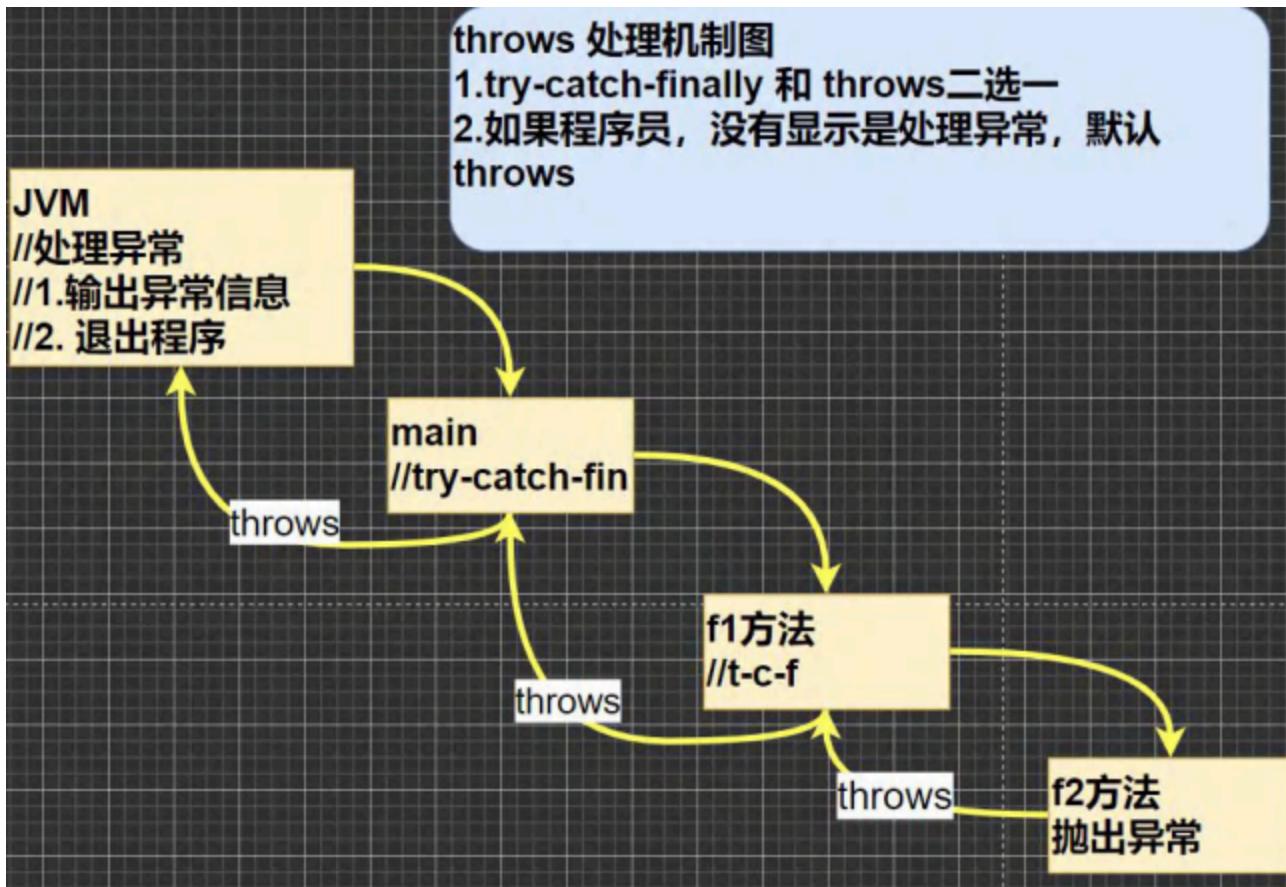
可以进行 try-finally 配合使用, 这种用法相当于没有捕获异常,

因此程序会直接崩掉/退出。应用场景, 就是执行一段代码, 不管是否发生异常,都必须执行某个业务逻辑

### 2. throws(将异常抛出,交给调用者(方法)处理,最顶级的就是JVM)

//异常可能显示其父类

流程图:



示例:

```
public static void readFile(String file) throws FileNotFoundException {
// .....
// 读文件的操作可能产生FileNotFoundException类型的异常
FileInputStream fis = new FileInputStream("d://aa.txt");
// .....
}
```

## 抛出异常为什么不用throws?

如果异常是未检查异常或者在方法内部被捕获和处理了，那么就不需要使用throws。

- **Unchecked Exceptions:** 未检查异常 (unchecked exceptions) 是继承自RuntimeException类或Error类的异常，编译器不强制要求进行异常处理。因此，对于这些异常，不需要在方法签名中使用throws来声明。示例包括NullPointerException、ArrayIndexOutOfBoundsException等。
- **捕获和处理异常:** 另一种常见情况是，在方法内部捕获了可能抛出的异常，并在方法内部处理它们，而不是通过throws子句将它们传递到调用者。这种情况下，方法可以处理异常而无需在方法签名中使用throws。

### 3. 注意事项和使用细节:

1. 编译异常,程序必须处理(try-catch或throws)
2. 运行异常,若没有处理,默认为throws方式处理
3. 子类重写父类异常时:子类的异常要么与父类一致,要么为父类异常的子类
4. throws过程若有try-catch(处理异常),就可以不throws
5. finally和try冲突时,finally的优先级更高:

**try{return "a"} finally{return "b"}这条语句返回啥**

finally块中的return语句会覆盖try块中的return返回,因此,该语句将返回"b"。

### 6. 自定义异常:

#### 1. 理解:

当程序错误并未在Throwable子类描述处理,可自己设计异常类来描述

#### 2. 步骤:

- 1) 定义类: 自定义异常类名(程序员自己写) 继承Exception或RuntimeException**
- 2) 如果继承Exception, 属于编译异常**
- 3) 如果继承RuntimeException, 属于运行异常(一般来说, 继承RuntimeException)**

#### 3. 示例:

```
public class Main{
    public static void main(String[] args){
        int age = 520;
        if(!(age>=18&&age<=120)){
            throw new AgeException("你个小猪");
        }
        System.out.println(age);
        ...
    }
    ...
}

class AgeException extends RuntimeException{
    public AgeException(String message){
        super(message);
    }
}
```

```
Exception in thread "main" Test.AgeException Create breakpoint : 你个小猪
at Test.Main.main(Main.java:9)
```

## 7. throw和throws区别:

|        | 意义           | 位置    | 后面跟的东西 |
|--------|--------------|-------|--------|
| throws | 异常处理的一种方式    | 方法声明处 | 异常类型   |
| throw  | 手动生成异常对象的关键字 | 方法体中  | 异常对象   |

示例:

```
class ReturnExceptionDemo {  
    static void methodA() {  
        try {  
            System.out.println("进入方法A");  
            throw new RuntimeException("制造异常");  
        } finally {  
            System.out.println("用A方法的finally");  
        }  
    }  
    static void methodB() {  
        try {  
            System.out.println("进入方法B");  
            return;  
        } finally {  
            System.out.println("调用B方法的finally");  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    try {  
        ReturnExceptionDemo.methodA();  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
    ReturnExceptionDemo.methodB();  
}  
  
输出内容  
1.进入方法A  
2.用A方法的finally  
3.制造异常  
4.进入方法B  
5.调用B方法的finally
```

## 8. 使用案例:

```
public class Main{
    public static void main(String[] args){
        String s = "1314520";
        try{
            s = reverse(s,6,5); //此处发生异常,在方法体里面有主动throw
        }catch(Exception e){
            System.out.println(e);
            //java.lang.RuntimeException: 参数不正确
            System.out.println(e.getMessage());
            //参数不正确
            return; //退出函数,使得不再进行
        }
        System.out.println(s);
    }

    public static String reverse(String str,int start,int end){
        if(end >= str.length() || start >= end || str == null){
            throw new RuntimeException("参数不正确");
            //如发生以上情况,则主动throw一个RuntimeException异常并给予信息"参数不正确"
        }
        char[] chars = str.toCharArray();
        for(int i = start,j = end;i<j;i++,j--){
            char temp = chars[i];
            chars[i] = chars[j];
            chars[j] = temp;
        }
        return new String(chars);
    }
}
```

## 3.3 真-常用类

### 1. 包装类(八种基本数据类型):

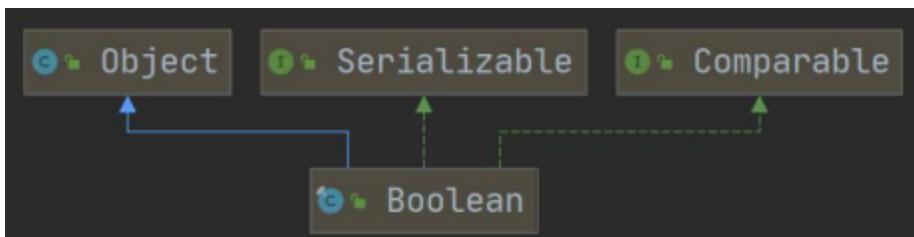
#### 1. 分类:

1. 大致:

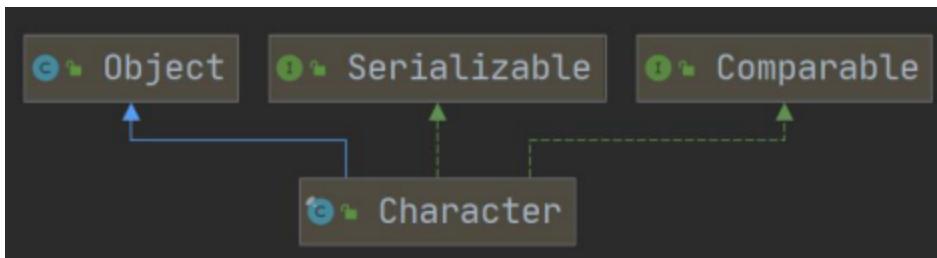
| 基本数据类型  | 包装类       |
|---------|-----------|
| boolean | Boolean   |
| char    | Character |
| byte    | Byte      |
| short   | Short     |
| int     | Integer   |
| long    | Long      |
| float   | Float     |
| double  | Double    |

## 2. 具体父子类:

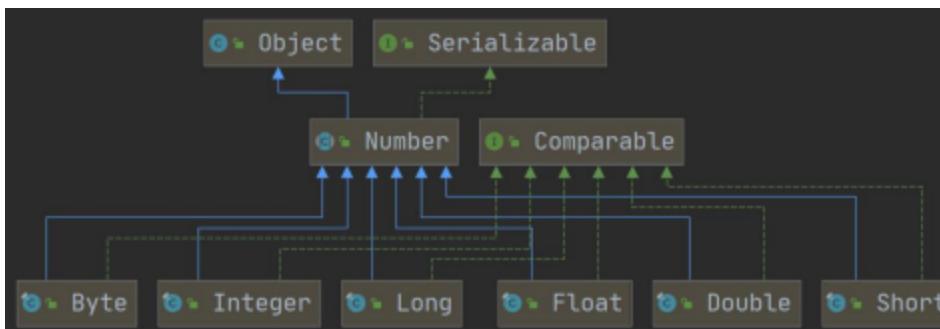
### i. Boolean:



#### ii. Character:



### iii. 其余六种:



## 2. 包装类与基本数据类型转换:

## 1. 运行逻辑:

- 1) jdk5 前的手动装箱和拆箱方式， 装箱: 基本类型->包装类型, 反之, 拆箱
  - 2) jdk5 以后(含jdk5) 的自动装箱和拆箱方式
  - 3) 自动装箱底层调用的是valueOf方法, 比如Integer.valueOf()

## 2. 以int和Integer示例:

i. 自动装箱/拆箱(jdk5后,直接大胆等号写就行):

```
int n2 = 200;
Integer integer2 = n2;
//自动装箱 int->Integer
//底层使用的是 Integer.valueOf(n2)
int n3 = integer2;
//自动拆箱 Integer->int
//底层仍然使用的是 intValue()方法
```

ii. 手动装箱/拆箱(jdk5前,了解):

```
int n1 = 100;
Integer integer = new Integer(n1);
Integer integer1 = Integer.valueOf(n1);
```

3. 强制数据转换(依旧存在):

```
Object obj1 = true? new Integer(1) : new Double(2.0);
大师
System.out.println(obj1);// 什么? 1.0
```

//为一个整体,Double精度大于Integer,所以(运行类型)结果转换为Double

### 3. 包装类和String类型相互转换

以Integer和String示例:

1. Integer到String:

```
Integer i = 100;
String str1 = i + "";//方式 1
String str2 = i.toString();//方式 2
String str3 = String.valueOf(i);//方式 3
```

2. String到Integer:

```
String str4 = "12345";
Integer i2 = Integer.parseInt(str4);//方法1 使用到自动装箱
Integer i3 = new Integer(str4);//方法2 构造器
```

### 4. Integer类常用方法:

```
System.out.println(Integer.MIN_VALUE);//返回最小值
System.out.println(Integer.MAX_VALUE);//返回最大值
```

## 5. Character类常用方法:

```
System.out.println(Character.isDigit('a'));//判断是不是数字  
System.out.println(Character.isLetter('a'));//判断是不是字母  
System.out.println(Character.isUpperCase('a'));//判断是不是大写  
System.out.println(Character.isLowerCase('a'));//判断是不是小写  
System.out.println(Character.isWhitespace('a'));//判断是不是空格  
System.out.println(Character.toUpperCase('a'));//转成大写  
System.out.println(Character.toLowerCase('A'));//转成小写
```

## 6. 特别提醒包装类与基本数据类型区别:

### 1. 本质区别:

1)

```
例int n1 = 1314;  
int n2 = 1314;  
System.out.println(n1 == n2);//true  
//n1和n2在线上,比较值是否相等
```

1)

```
Integer n3 = 1314;  
Integer n4 = 1314;  
System.out.println(n3 == n4);//false  
//n3和n4是对象的引用,在堆上,比较引用地址, Integer已经缓存好了-128到127的整数值,所以如果数值在其中,本质上会是获取一个已经创建好的实例,所以此时会相等  
//但是若数值不在范围中,则会创建新的不同的对象,也就是会出现下面这种情况
```

1)

```
Integer n5 = 14;  
Integer n6 = 14;  
System.out.println(n5 == n6);//true  
//int本质上只是一种数据类型,而Integer本质上是一种类,所以会创建对象(在超出已缓存范围时),n3和n4依旧是两个不同的对象,所以实际上比较的是内存中的地址,为false
```

1)

```
Integer n7 = new Integer(14);  
Integer n8 = new Integer(14);  
Integer n9 = 14;  
System.out.println(n7 == n8);//false  
System.out.println(n8 == n9);//false  
//这里就是直接创建俩Integer的对象了,自然是不相等的  
//当然,拿n9的值跟对象的地址比也不相等
```

## 2. 上述示例总结:

- i. Integer有两种方式,一种为直接Integer n1 = 1314;另一种为Integer n1 = new Integer(1314);
- ii. 第一种方式会考察赋值范围是否在-128到127之间,在此期间的对象都已在Integer缓存完毕,所以会直接赋予同一对象的地址值,而超出范围的才会额外单独创建对象(独立分配空间),跟第二种方式类似
- iii. 第二种方式会直接创建新对象(独立分配空间)
- iv. 两种方式无论如何依旧是类的对象(在堆中),所以在 == 中比较的是地址值,而int不一样,int数据类型的对象储存在栈中,在 == 中比较的是指向值,所以Integer怎么赋值都不能跟int相等

## 2. String类(保存字符串,有双引号,用Unicode字符编码,一个字符(部分字母和汉字)占俩字节)

### 1. 构造与说明:

String类较常用构造器(其它看手册):

```
String s1 = new String(); //  
String s2 = new String(String original);  
String s3 = new String(char[] a);  
String s4 = new String(char[] a,int startIndex,int count)
```

```
// 常用的有 String s1 = new String(); //  
//String s2 = new String(String original);  
//String s3 = new String(char[] a);  
//String s4 = new String(char[] a,int startIndex,int count)  
//String s5 = new String(byte[] b)
```



//串行化:指的是一个类可以被转换成一系列字节, 这些字节可以在网络上传输或者保存到文件中, 之后

又可以被重新组合和转换成原来的对象。这个过程称为对象的序列化（Serialization）和反序列化（Deserialization）

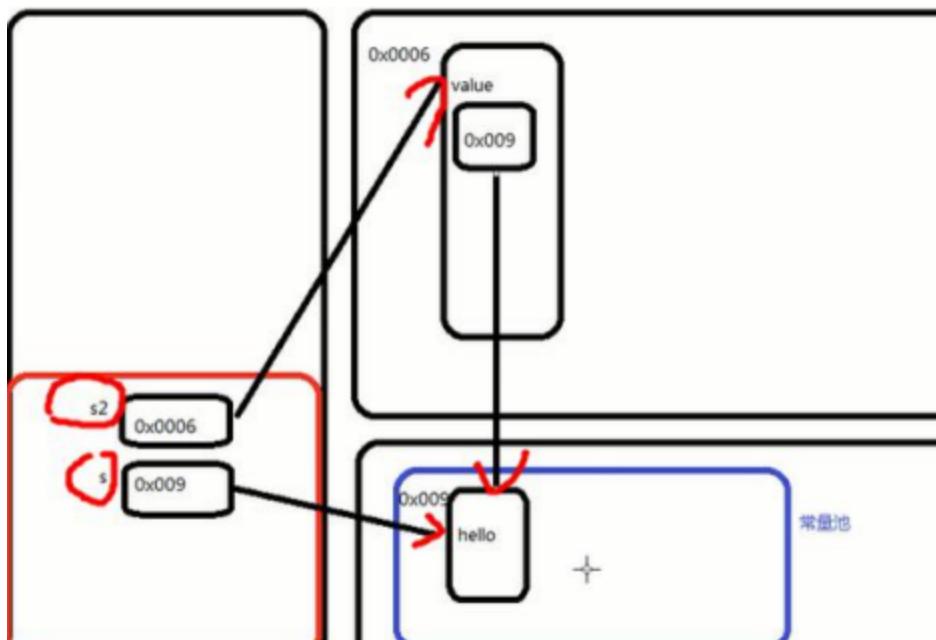
## 2. 创建对象的两种方法不同：

**方式一：直接赋值 String s = "hsp";**

**方式二：调用构造器 String s2 = new String("hsp");**

1. 方式一：先从常量池查看是否有“hsp”数据空间，如果有，直接指向；如果没有则重新创建，然后指向。**s最终指向的是常量池的空间地址**
2. 方式二：先在堆中创建空间，里面维护了value属性，指向常量池的hsp空间。**如果常量池没有“hsp”，重新创建，如果有，直接通过value指向。最终指向的是堆中的空间地址。**

(内存图不同：)



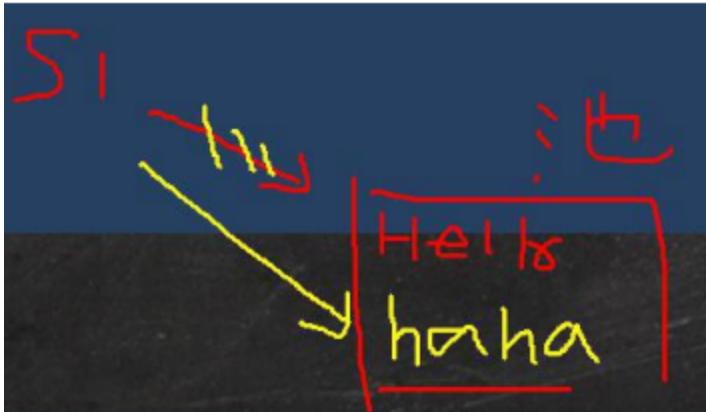
## 3. String的特别(有关new分配指向空间和equals,在3.):

1. 可以串行化:实现**网络传输**
2. 实现了接口:**对象可以比较大小**
3. 是**final类**,不能被继承也代表**不可变的字符序列**(对象一旦被分配,内容不可变)

//但是对于String str = new String("love");使用 new String() 创建了一个新实例，后续对变量赋新的字符串字面量的值也不会修改原有对象(**原有对象等待回收或者被其他引用**)，而是让变量引用一个新的对象,也就是说**str可以改变指向的值**

例:

```
String s1 = "hello";
s1="haha"; //1min
//创建了2个对象.
```



```
String a = "hello"; //创建 a对象
String b = "abc"; //创建 b对象
String c=a+b; 创建了几个对象? 画出内存图?
//关键就是要分析 String c = a + b; 到底是如何执行的
//一共有3对象, 如图。
```



图片7-2

**老韩小结:** 底层是 `StringBuilder sb = new StringBuilder(); sb.append(a); sb.append(b);` `sb` 是在堆中, 并且 `append` 是在原来字符串的基础上追加的。  
**重要规则**, `String c1 = "ab" + "cd";` 常量相加, 看的是池。 `String c1 = a + b;` 变量相加, 是在堆中

(但是`String we = "love"+"forever";`只算一句,所以这里只创建一个)

```
String s1 = "hspedu"; //s1 指向池中的 "hspedu"
String s2 = "java"; // s2 指向池中的 "java"
String s5 = "hspedujava"; //s5 指向池中的 "hspedujava"
String s6 = (s1 + s2).intern(); //s6 指向池中的 "hspedujava"
System.out.println(s5 == s6); //T
System.out.println(s5.equals(s6)); //T
```

//只要没有new就不会额外分配指向内容的空间,所以在类里面的非 new 的String反而也会指向常量池  
//多看源码  
4. String存在属性private final char value[];(用于存放字符串内容)  
//此处的value为final类型,不能被修改地址,不过单个字符内容可修改

```
String name = "jack";
name = "tom";
final char[] value = {'a','b','c'};
char[] v2 = {'t','o','m'};
value[0] = 'H';
//value = v2; 不可以修改 value 地址
```

5. String的equals重写(导致equals会比较String的值而非地址):

```
public final class String
{
    public boolean equals(Object anObject) {
        if (this == anObject) {
            return true;
        }
        return (anObject instanceof String aString)
            && (!COMPACT_STRINGS || this.coder == aString.coder)
            && StringLatin1.equals(value, aString.value);
    }
}
```

```
String a = new String("abc");
String b =new String("abc");
System.out.println(a.equals(b));//T
System.out.println(a==b); //F
```

#### 4. 特别关于intern方法:

```
String a = "hsp"; //a 指向 常量池的 "hsp"  
String b = new String("hsp"); //b 指向堆中对象  
System.out.println(a.equals(b)); //T  
System.out.println(a==b); //F  
System.out.println(a==b.intern()); //T //intern方法自己先查看API  
System.out.println(b==b.intern()); //F
```

知识点：

当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（用 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添加到池中，并返回此 String 对象的引用

老韩解读：(1) b.intern() 方法最终返回的是常量池的地址（对象）。

//一回生二回熟，用equals判断常量池中有没有该字符串，没有就加进去，并返回地址（第一次），否则就会返回这个字符串本身

#### 5. String常见方法:

1. equals:比较内容是否相同(区分大小写)

例: System.out.println(str1.equals(str2));

2. equalsIgnoreCase:比较内容是否相同(忽略大小写)

例: System.out.println(str1.equalsIgnoreCase(str2));

3. 字符串.length():获取字符个数/字符串长度

4. indexOf:获取字符(char)在字符串对象中第一次出现的索引(找不到返回-1)

例:

```
String s1 = "wer@terwe@g@";  
int index = s1.indexOf('@');//3
```

5. lastIndexOf:获取字符(char)在字符串对象中最后一次出现的索引(找不到返回-1)

例:

```
String s1 = "wer@terwe@g@";  
int index = s1.lastIndexOf('@');//11
```

6. substring:截取指定范围的子串

例:

```
String name = "hello,张三";
```

```
System.out.println(name.substring(6));//从索引 6 开始截取后面所有的内容 张三  
System.out.println(name.substring(2,5));//从索引2取到索引5(前闭后开)lo
```

## 7. toUpperCase:全部转换成大写

例:

```
String s1 = s2.toUpperCase();
```

## 8. toLowerCase:全部转换成小写

例:

```
String s1 = s2.toLowerCase();
```

## 9. concat:拼接字符串

例:

```
String s1 = "xjh";  
s1 = s1.concat(" love").concat(" hxq").concat(" forever~");//xjh love hxq forever~
```

## 10. replace:替换字符串中的字符(产生新串,不改变原串)

例:

```
s1 = "I love hxq forever~";  
String s11 = s1.replace("I", "xjh");  
System.out.println(s1);//I love hxq forever~  
System.out.println(s11);//xjh love hxq forever~
```

## 11. split:分割字符串(分为数组元素,如有特殊字符,需要加入转义符)

例:

```
String poem = "锄禾日当午,汗滴禾下土,谁知盘中餐,粒粒皆辛苦";  
String[] split = poem.split(",");  
poem = "E:\aaa\bbb";  
split = poem.split("\\\\");//意为按"\\\"分割,{\"E:\","aaa","bbb"};
```

## 12. toCharArray:转为字符(char)数组

例:

```
char[] char = str.toCharArray();
```

## 13. compareTo:比较两字符串大小(前大正数,后大负数,相等出0)

例:

```
String a = "jcck";// len = 3  
String b = "jack";// len = 4
```

```
System.out.println(a.compareTo(b)); //返回'c'-'a'的值  
//if (c1 != c2) {  
// return c1 - c2;  
// }返回第一个不同字符的大小之差  
//若长度不同,但是前面都相同,返回str1.length-str2.length(长度之差)
```

## 14. format:格式占位符

### 1. 占位符:

- i. %s :字符串
- ii. %c :字符
- iii. %d :整型
- iv. %.2f :浮点型

//后面有个 . 再加上 **2f**

//用小数替换,四舍五入保留两位

### 2. 示例:

```
String formatStr = "我的姓名是%s 年龄是%d, 成绩是%.2f 性别是%c.希望大家喜欢我! ";  
String info = String.format(formatStr, name, age, score, gender);
```

## 3. StringBuffer类(可以理解为可变长度的String,可以对字符串增删)

### 1. StringBuffer类自身(源码):

```
/*  
 * public final class StringBuffer  
 * extends AbstractStringBuilder  
 * implements java.io.Serializable, CharSequence  
 */  
1. StringBuffer 是final 类  
2. 实现了Serializable 接口, 可以  
   保存到文件, 或网络传输  
3. 继承了抽象类  
   AbstractStringBuilder  
4. AbstractStringBuilder 属性  
   char[] value , 存放的字符序列
```

1. 直接父类是 AbstractStringBuilder
2. 实现了 Serializable, 即StringBuffer的对象可以串行化
3. 在父类中 AbstractStringBuilder 有属性 **char[] value**存放字符串内容,不是 final,所以才能**改变字符串**(而String是final,所以String不可变)  
//value 数组存放字符串内容, 引出存放在堆中  
//改变字符串不用每次都更换地址(即**不是每次创建新对象**), 所以**效率高于 String**
4. StringBuffer 是一个 final 类, **不能被继承**

### 2. 构造对象(同String)

```
StringBuffer sBuffer = new StringBuffer("love");  
//当然也不能new StringBuffer(null);  
//因为源码中会创建大小,调用length,但是null没有length,会变成空指针异常NullPointerException
```

### 3. 与String比较:

- 1) String保存的是字符串常量，里面的值不能更改，每次String类的更新实际上就是更改地址，效率较低 //private final char value[];
- 2) StringBuffer保存的是字符串变量，里面的值可以更改，每次StringBuffer的更新实际上可以更新内容，不用每次更新地址，效率较高 //char[] value; // 这个放在堆.

### 4. String与StringBuffer相互转换:

1. String到StringBuffer:

i. 方法一(构造器):

```
String str = "love";  
StringBuffer sBuffer = new StringBuffer(str);
```

ii. 方法二 append方法:

```
StringBuffer sBuffer1 = new StringBuffer();  
sBuffer1.append(str);
```

2. StringBuffer到String:

i. 方法一(构造器):

```
StringBuffer sBuffer = new StringBuffer("love");  
String s1 = new String(sBuffer);
```

ii. 方法二(toString)

```
String s2 = sBuffer.toString();
```

### 5. StringBuffer常见方法:

1. append:在末尾增加字符串(包括null,特殊)

例:

```
StringBuffer s = new StringBuffer("xjh");  
s.append(" love hxq");//xjh love hxq  
s.append(" forever").append('~').append(true).append(1314.1314);//xjh love hxq  
forever~true1314.1314  
//此时1314.1314部分长度为8,变成单独的字符了  
//如果append(null),会真的在末尾加上"null"的字符串
```

## 2. delete:删除指定索引范围的字符串(前闭后开)

例:

```
s.delete(22,26);//xjh love hxq forever~1314.1314
```

## 3. replace:修改指定索引范围的字符串(前闭后开)

例:

```
s.replace(22,26,"520");//xjh love hxq forever~520.1314
```

//替换字符串长度可以与原串不同

## 4. insert:将字符串插入字符串

例:

```
s.insert(22,"!!");//xjh love hxq forever~!!520.1314
```

//将包括索引为22的字符以后全部后移

## 5. indexOf(查索引,同String)

## 6. length(长度,同String)

# 4. StringBuilder类(被设计为StringBuffer的简易上位,大多数情况下比它快)

## 1. 介绍:

```
public final class StringBuilder
    extends AbstractStringBuilder
    implements Appendable, java.io.Serializable, Comparable<StringBuilder>, CharSequence
{
```

1. 提供一个与StringBuffer兼容的API,但不保证同步(StringBuffer不是线程安全)
2. 直接父类为AbstractStringBuilder,有属性char[] value存放字符串(不是final所以才能修改)(所以字符串序列是堆中)
3. 实现了Serializable接口,即StringBuilder的对象可以串行化(可网络传输/保存在文件)
4. 为final类,不能被继承
5. 用在字符串缓冲区被单个线程使用的时候(因为没有作互斥处理/没有synchronized关键字)
6. 主要操作为append和insert方法(可重载接收任意类型的数据)

## 2. 方法(与StringBuilder相同)

# 5. String,StringBuffer,StringBuilder区别

String:不可变字符串(是final),效率低,复用率高

```
string s = "a"; //创建了一个字符串  
s += "b"; //实际上原来的"a"字符串对象已经丢弃了，现在又产生了一个字符串s + "b" (也就是"ab")。如果多次执行这些改变串内容的操作，会导致大量副本字符串对象存留在内存中，降低效率。如果这样的操作放到循环中，会极大影响程序的性能 => 结论：如果我们对String 做大量修改，不要使用String
```

StringBuffer:可变字符串(不是final),效率较高,线程安全,使用了synchronized实现同步

StringBuilder:可变字符串(不是final),效率最高,线程不安全

1. 如果字符串存在大量的修改操作，一般使用 StringBuffer 或StringBuilder
2. 如果字符串存在大量的修改操作，并在单线程的情况，使用 StringBuilder
3. 如果字符串存在大量的修改操作，并在多线程的情况，使用 StringBuffer
4. 如果我们字符串很少修改，被多个对象引用，使用String, 比如配置信息等

## 5. Math类(基本作为工具类使用,含有大量静态方法)

### 1.abs:绝对值

```
int abs = Math.abs(-9);  
//int,double,float,long都行
```

### 2.pow:求幂

```
double pow = Math.pow(2, 4);  
//2的4次方
```

### 3.ceil:向上取整(转成double)

```
double ceil = Math.ceil(3.9); //4.0
```

### 4.floor:向下取整(转成double)

```
double ceil = Math.floor(4.001); //4.0
```

### 5.round:四舍五入(float返回int,double返回long)

```
long round = Math.round(5.51); //6
```

### 6.sqrt:开方

```
double sqrt = Math.sqrt(9.0); //3.0
```

## 7.random:随机数(返回[0,1)的随机小数)

```
//Math.random()(b-a) 返回的就是 0<= 数 <=b-a  
//(int)(a) <= x <= (int)(a + Math.random()(b-a+1))
```

## 8.max/min:返回最大值/最小值

```
int min = Math.min(520,1314);//只能比较两个数的大小并返回  
System.out.println(min);//520
```

# 6. Array类(基本作为工具类使用,含有大量静态方法)

## 1. toString:返回数组的字符串形式

```
Integer[] integers = {1, 20, 90};  
System.out.println(Arrays.toString(integers));  
//输出内容:[1, 20, 90]
```

## 2. sort:排序(自然和定制-匿名内部类+动态绑定)

//数组是引用类型，所以通过sort排序后，会直接影响到实参arr

1. sort默认的排序(由小到大):

```
Integer arr[] = {1, -1, 7, 0, 89};  
Arrays.sort(arr);  
//arr = {-1, 0, 1, 7, 89}
```

2. 通过接口实现的自定义排序规则:

```
Arrays.sort(arr, new Comparator() {  
    @Override  
    public int compare(Object o1, Object o2)  
    {  
        Integer i1 = (Integer) o1;  
        Integer i2 = (Integer) o2;  
        return i2 - i1;  
    }  
});
```

//此处示例变为由大到小排序

(1) Arrays.sort(arr, new Comparator()

(2) 最终到 TimSort 类的 private static <T> void binarySort(T[] a, int lo, int hi, int start,  
Comparator<? super T> c)()

(3) 执行到 binarySort 方法的代码, 会根据动态绑定机制 c.compare() 执行我们传入的  
匿名内部类的 compare()

```
while (left < right) {  
    int mid = (left + right) >>> 1;  
    if (c.compare(pivot, a[mid]) < 0)  
        right = mid;  
    else  
        left = mid + 1;  
}
```

(4) new Comparator() {

```
    @Override  
    public int compare(Object o1, Object o2) {  
        Integer i1 = (Integer) o1;  
        Integer i2 = (Integer) o2;  
        return i2 - i1;  
    }  
}
```

(5) public int compare(Object o1, Object o2) 返回的值>0 还是 <0

会影响整个排序结果. 这就充分体现了 接口编程+动态绑定+匿名内部类的综合使用

### 3. binarySearch:二分查找(前提是排好序)

int index = Arrays.binarySearch(arr, 3); //返回3的索引

如果不存在该元素, 就返回 -(low + 1); // key not found

## 4. copyOf:复制数组元素

```
Integer[] arr = {...};  
Integer[] newArr = Arrays.copyOf(arr,arr.length);
```

## 5. fill:数组元素的填充

```
Integer[] num = new Integer[]{13,14,520};  
Array.fill(num,1314);  
//num = {1314,1314,1314}
```

## 6. equals:比较两个数组元素内容是否完全一致(顺序和内容)

```
boolean equals = Arrays.equals(arr1,arr2);
```

## 7. asList:将一组值转换为list

```
List asList = Arrays.asList(2,3,4,5,6,1);  
System.out.println("asList=" + asList);  
System.out.println("asList 的运行类型" +asList.getClass());  
asList=[2, 3, 4, 5, 6, 1]  
asList 的运行类型class java.util.Arrays$ArrayList
```

//运行类型(为静态内部类)java.util.Arrays#ArrayList是Arrays类中的

## 7. System类(的常见方法)

### 1. exit:退出当前程序

```
System.exit(0); //0是一种状态,exit(0)表示程序退出
```

### 2. arraycopy:复制数组元素(适合底层调用,一般还是用Arrays.copyOf)

```
System.arraycopy(src, 0, dest, 0, src.length);  
//src:源数组  
//第一个0:从源数组的那个索引位置copy  
//dest:目标数组,要接收拷贝数据的数组  
//第二个0:接收数组的那个索引  
//src.length:从源数组拷贝多少数据到目标数组
```

3. currentTimeMillens:返回当前时间距离1970-1-1的毫秒数(数据类型为long)

```
long startTime = System.currentTimeMillis();
```

## 4. gc:运行垃圾回收机制

```
System.gc();
```

## 8. BigInteger类和BigDecimal类(分别保存较大的整型和精度更高的浮点型)

## 1. 使用场景:

## 1. BigInteger:

```
//需导入import java.math.BigDecimal;
```

```
BigInteger bl = new BigInteger("131452013145201314520");
```

//BigInteger用于**超级大的整数**(long都不够用的那种)

## 2. BigDecimal:

```
//需导入import java.math.BigInteger;
```

```
BigDecimal bD = new BigDecimal("1999.111111111199999999999977788");
```

```
System.out.println(bD)
```

//BigDecimal用于**超级大精度的浮点数**(double都不够用的那种)

## 2. 加减乘除方法:

## 1. add:加

```
System.out.println(bigInteger.add(bigInteger2));
```

```
System.out.println(bigDecimal.add(bigDecimal2));
```

## 2. subtract:

```
System.out.println(bigInteger.subtract(bigInteger2));
```

```
System.out.println(decimal.subtract(decimal2));
```

### 3. multiply:

```
System.out.println(bigInteger.multiply(bigInteger2));
```

```
System.out.println(decimal.multiply(decimal2));
```

#### 4. divide:

```
System.out.println(bigInteger.divide(bigInteger2));
System.out.println(decimal.divide(decimal2));
//可能抛出异常 ArithmeticException(除数为0)
```

## 9. 日期类

### 1. Date类(第一代日期类,精确到毫秒,表示特定的瞬间,java.util.Date)

#### 1. 特别之处:

默认输出格式为国外的方式,需要对格式进行转换

#### 2. 使用:

##### 1. 获取系统时间:

```
Date d1 = new Date(); //获取当前系统时间
System.out.println("当前日期=" + d1);
//Thu Oct 17 23:14:54 CST 2024
//美国中部标准时间2024.10.17.星期四.23:14:54
```

##### 2. 通过指定毫秒数得到时间:

```
Date d1 = new Date(1314520); //对比1970.1.1过去了多少毫秒
System.out.println(d1);
//Thu Jan 01 08:21:54 CST 1970
```

### 2. SimpleDateFormat(第一代日期类,格式和解析日期的类,java.text.SimpleDateFormat)

示例:

```

Date d1 = new Date(); //通过Date类获取系统时间
SimpleDateFormat sdf = new SimpleDateFormat("yyyy 年 MM 月 dd 日 hh:mm:ss E"); //自定义格式
String format = sdf.format(d1);
System.out.println("当前日期=" + format);
//format:将日期转换成指定格式的字符串String
String s = "1996 年 01 月 01 日 10:20:30 星期一";
Date parse = null;
try {
    parse = sdf.parse(s);
} catch (ParseException e) {
    System.out.println("出错了");
    throw new RuntimeException(e);
}
System.out.println("parse=" + sdf.format(parse));
//可以将字符串String转换成Date格式,但是sdf必须与输入所想转换的格式一样,否则出错

```

### 3. Calendar(第二代日期类,日历抽象类,java.util.Calendar)

#### 1. 特别之处:

1. Calendar是一个**抽象类,构造器为private**.提供大量的方法与字段,没有提供格式化的类,**可自行组合**
2. 改为24小时制:Calendar.HOUR ==改成=> Calendar.HOUR\_OF\_DAY

#### 2. 使用:

```

Calendar calendar = Calendar.getInstance(); //构造器为private,必须通过getInstance方法创建实例
System.out.println(calendar);
System.out.println(calendar.get(Calendar.YEAR)); //年
System.out.println(calendar.get(Calendar.MONTH)+1); //月,但是从0 编号,需+1
System.out.println(calendar.get(Calendar.DAY_OF_MONTH)); //日
System.out.println(calendar.get(Calendar.HOUR_OF_DAY)); //小时
System.out.println(calendar.get(Calendar.MINUTE)); //分钟
System.out.println(calendar.get(Calendar.SECOND)); //秒
System.out.println(calendar.get(Calendar.MILLISECOND)); //毫秒
System.out.println(calendar.get(Calendar.DAY_OF_WEEK)); //星期几,但是以星期天为一星期的起点
System.out.println(calendar.get(Calendar.DAY_OF_WEEK_IN_MONTH)); //该月份第几个星期

```

## 4. 第三代日期类

### 1. LocalDate(日期-年月日), LocalTime(时间-时分秒), LocalDateTime(年月日时分秒)

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.DateTimeFormatter
...
LocalDateTime ldt = LocalDateTime.now();
//返回当前时间,三种类同语法
System.out.println(ldt);
System.out.println("年=" + ldt.getYear());//年
System.out.println("月=" + ldt.getMonth());//英文月
System.out.println("月=" + ldt.getMonthValue());//数字月
System.out.println("日=" + ldt.getDayOfMonth());//日
System.out.println("时=" + ldt.getHour());//时
System.out.println("分=" + ldt.getMinute());//分
System.out.println("秒=" + ldt.getSecond());//秒
LocalDateTime localDateTime = ldt.plusDays(1314);
System.out.println("1314 天后=" + dateTimeFormatter.format(localDateTime));//时间增加
LocalDateTime localDateTime2 = ldt.minusMinutes(520);
System.out.println("520 分钟前 日期=" + dateTimeFormatter.format(localDateTime2));//时间减少
...
```

```
2024-10-17T23:50:55.395864600
年=2024
月=OCTOBER
月=10
日=17
时=23
分=50
秒=55
```

```
1314 天后=2028-05-23 23:53:57
520 分钟前 日期=2024-10-17 15:13:57
```

## 2. DateTimeFormatter(格式日期类,与第一代日期类的SimpleDateFormat类似)

```
//接上文
DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
//自定义输出的日期格式
String format = dateTimeFormatter.format(ldt);
//依旧使用format转换
System.out.println("格式化的日期=" + format);
```

格式化的日期=2024-10-17 23:50:55

## 5. Instant(时间戳,可与Date转换)

### 1. 使用语法:

```
Instant now = Instant.now(); //通过静态方法 now() 获取表示当前时间戳的对象
Date date = Date.from(now); //Instant到Date
Instant instant = date.toInstant(); //Date到Instant
System.out.println(now);
System.out.println(date);
System.out.println(instant);
```

```
2024-10-17T16:02:58.788575400Z
Fri Oct 18 00:02:58 CST 2024
2024-10-17T16:02:58.788Z
```

## 10. 关于重写compareTo:

需要将对应的类实现接口Comparable,示例:

```
public class MyDate implements Comparable<MyDate>{
    ...
    public int compareTo(MyDate o) {
        ...
    }
}
```

## 3.4 集合

### 特殊章节:八股

1. 总览:

1. **ArrayList:** 动态数组，实现了List接口，支持动态增长。
2. **LinkedList:** 双向链表，也实现了List接口，支持快速的插入和删除操作。
3. **HashMap:** 基于哈希表的Map实现，存储键值对，通过键快速查找值。
4. **HashSet:** 基于HashMap实现的Set集合，用于存储唯一元素。
5. **TreeMap:** 基于红黑树实现的有序Map集合，可以按照键的顺序进行排序。
6. **LinkedHashMap:** 基于哈希表和双向链表实现的Map集合，保持插入顺序或访问顺序。
7. **PriorityQueue:** 优先队列，可以按照比较器或元素的自然顺序进行排序。

List是有序的Collection，使用此接口能够精确的控制每个元素的插入位置，用户能根据索引访问List中元素。常用的实现List的类有LinkedList，ArrayList，Vector，Stack。

- ArrayList是容量可变的非线程安全列表，其底层使用数组实现。当几何扩容时，会创建更大的数组，并把原数组复制到新数组。ArrayList支持对元素的快速随机访问，但插入与删除速度很慢。
- LinkedList本质是一个双向链表，与ArrayList相比，其插入和删除速度更快，但随机访问速度更慢。

Set不允许存在重复的元素，与List不同，set中的元素是无序的。常用的实现有HashSet，LinkedHashSet和TreeSet。

- HashSet通过HashMap实现，HashMap的Key即HashSet存储的元素，所有Key都是用相同的Value，一个名为PRESENT的Object类型常量。使用Key保证元素唯一性，但不保证有序性。由于HashSet是HashMap实现的，因此线程不安全。
- LinkedHashSet继承自HashSet，通过LinkedHashMap实现，使用双向链表维护元素插入顺序。
- TreeSet通过TreeMap实现的，添加元素到集合时按照比较规则将其插入合适的位置，保证插入后的集合仍然有序。

Map 是一个键值对集合，存储键、值和之间的映射。Key 无序，唯一；value 不要求有序，允许重复。Map 没有继承于 Collection 接口，从 Map 集合中检索元素时，只要给出键对象，就会返回对应的值对象。主要实现有 TreeMap、HashMap、HashTable、LinkedHashMap、ConcurrentHashMap

- **HashMap**: JDK1.8 之前 HashMap 由数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突），JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间
- **LinkedHashMap**: LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。
- **HashTable**: 数组+链表组成的，数组是 HashTable 的主体，链表则是主要为了解决哈希冲突而存在的
- **TreeMap**: 红黑树（自平衡的排序二叉树）
- **ConcurrentHashMap**: Node数组+链表+红黑树实现，线程安全的（jdk1.8以前Segment锁，1.8以后 volatile + CAS 或者 synchronized）

## 现代线程安全集合选择矩阵

| 场景特征    | 推荐集合                     | 技术优势                   |
|---------|--------------------------|------------------------|
| 高频更新键值对 | ConcurrentHashMap        | 分段锁/CAS混合模式，锁粒度最细      |
| 读多写少列表  | CopyOnWriteArrayList     | 无锁读+写时复制，极致读性能         |
| 无界任务队列  | ConcurrentLinkedQueue    | CAS无锁算法，百万级吞吐量         |
| 定时任务调度  | DelayQueue               | 优先级堆+ReentrantLock条件队列 |
| 分布式锁同步  | ConcurrentHashMap.KeySet | computeIfAbsent原子操作    |

在 java.util 包中的线程安全的类主要 2 个，其他都是非线程安全的。

- **Vector**: 线程安全的动态数组，其内部方法基本都经过synchronized修饰，如果不需要线程安全，并不建议选择，毕竟同步是有额外开销的。Vector 内部是使用对象数组来保存数据，可以根据需要自动的增加容量，当数组已满时，会创建新的数组，并拷贝原有数组数据。
- **Hashtable**: 线程安全的哈希表，HashTable 的加锁方法是给每个方法加上 synchronized 关键字，这样锁住的是整个 Table 对象，不支持 null 键和值，由于同步导致的性能开销，所以已经很少被推荐使用，如果要保证线程安全的哈希表，可以用ConcurrentHashMap。

---

java.util.concurrent 包提供的都是线程安全的集合：

并发Map：

- **ConcurrentHashMap**：它与 HashTable 的主要区别是二者加锁粒度的不同，在JDK1.7， ConcurrentHashMap加的是分段锁，也就是Segment锁，每个Segment 含有整个 table 的一部分，这样不同分段之间的并发操作就互不影响。在JDK 1.8，它取消了Segment字段，直接在table元素上加锁，实现对每一行进行加锁，进一步减小了并发冲突的概率。对于put操作，如果Key对应的数组元素为 null，则通过CAS操作（Compare and Swap）将其设置为当前值。如果Key对应的数组元素（也即链表表头或者树的根元素）不为null，则对该元素使用 synchronized 关键字申请锁，然后进行操作。如果该 put 操作使得当前链表长度超过一定阈值，则将该链表转换为红黑树，从而提高寻址效率。
- **ConcurrentSkipListMap**：实现了一个基于SkipList（跳表）算法的可排序的并发集合，SkipList是一种可以在对数预期时间内完成搜索、插入、删除等操作的数据结构，通过维护多个指向其他元素的“跳跃”链接来实现高效查找。

并发Set：

- **ConcurrentSkipListSet**：是线程安全的有序的集合。底层是使用ConcurrentSkipListMap实现。
- **CopyOnWriteArrayList**：是线程安全的Set实现，它是线程安全的无序的集合，可以将它理解成线程安全的HashSet。有意思的是，CopyOnWriteArrayList和HashSet虽然都继承于共同的父类AbstractSet；但是，HashSet是通过“散列表”实现的，而CopyOnWriteArrayList则是通过“动态数组（CopyOnWriteArrayList）”实现的，并不是散列表。

并发List：

- **CopyOnWriteArrayList**：它是 ArrayList 的线程安全的变体，其中所有写操作（add, set等）都通过对底层数组进行全新复制来实现，允许存储 null 元素。即当对象进行写操作时，使用了Lock锁做同步处理，内部拷贝了原数组，并在新数组上进行添加操作，最后将新数组替换掉旧数组；若进行的读操作，则直接返回结果，操作过程中不需要进行同步。

并发 Queue:

- **ConcurrentLinkedQueue**: 是一个适用于高并发场景下的队列，它通过无锁的方式(CAS)，实现了高并发状态下的高性能。通常，ConcurrentLinkedQueue 的性能要好于 BlockingQueue。
- **BlockingQueue**: 与 ConcurrentLinkedQueue 的使用场景不同，BlockingQueue 的主要功能并不是在于提升高并发时的队列性能，而在于简化多线程间的数据共享。BlockingQueue 提供一种读写阻塞等待的机制，即如果消费者速度较快，则 BlockingQueue 则可能被清空，此时消费线程再试图从 BlockingQueue 读取数据时就会被阻塞。反之，如果生产线程较快，则 BlockingQueue 可能会被装满，此时，生产线程再试图向 BlockingQueue 队列装入数据时，便会被阻塞等待。

并发 Deque:

- **LinkedBlockingDeque**: 是一个线程安全的双端队列实现。它的内部使用链表结构，每一个节点都维护了一个前驱节点和一个后驱节点。LinkedBlockingDeque 没有进行读写锁的分离，因此同一时间只能有一个线程对其进行操作
- **ConcurrentLinkedDeque**: ConcurrentLinkedDeque是一种基于链接节点的无限并发链表。可以安全地并发执行插入、删除和访问操作。当许多线程同时访问一个公共集合时，ConcurrentLinkedDeque是一个合适的选择。

## Collections和Collection的区别

- Collection是Java集合框架中的一个接口，它是所有集合类的基础接口。它定义了一组通用的操作和方法，如添加、删除、遍历等，用于操作和管理一组对象。Collection接口有许多实现类，如List、Set和Queue等。
- Collections（注意有一个s）是Java提供的一个工具类，位于java.util包中。它提供了一系列静态方法，用于对集合进行操作和算法。Collections类中的方法包括排序、查找、替换、反转、随机化等等。这些方法可以对实现了Collection接口的集合进行操作，如List和Set。

## 2. 集合遍历:

- **使用 forEach 方法:** Java 8引入了 forEach 方法，可以对集合进行快速遍历。

```
java
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");

list.forEach(element -> System.out.println(element));
```

- **Stream API:** Java 8的Stream API提供了丰富的功能，可以对集合进行函数式操作，如过滤、映射等。

```
java
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");

list.stream().forEach(element -> System.out.println(element));
```

list:

`CopyOnWriteArrayList` 的 `remove` 方法同样可以删除指定下标的元素。由于 `CopyOnWriteArrayList` 在写操作时会创建一个新的数组，所以删除操作的时间复杂度取决于数组的复制速度，通常为  $O(n)$ ， $n$  为数组的长度。但在并发环境下，它的删除操作不会影响读操作，具有较好的并发性能。示例代码如下：

```
java
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListRemoveExample {
    public static void main(String[] args) {
        CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);

        // 删除下标为1的元素
        list.remove(1);

        System.out.println(list);
    }
}
```

ArrayList和LinkedList都是Java中常见的集合类，它们都实现了List接口。

- **底层数据结构不同**: ArrayList使用数组实现，通过索引进行快速访问元素。LinkedList使用链表实现，通过节点之间的指针进行元素的访问和操作。
- **插入和删除操作的效率不同**: ArrayList在尾部的插入和删除操作效率较高，但在中间或开头的插入和删除操作效率较低，需要移动元素。LinkedList在任意位置的插入和删除操作效率都比较高，因为只需要调整节点之间的指针，但是LinkedList是不支持随机访问的，所以除了头结点外插入和删除的时间复杂度都是 $O(n)$ ，效率也不是很高所以LinkedList基本没人用。
- **随机访问的效率不同**: ArrayList支持通过索引进行快速随机访问，时间复杂度为 $O(1)$ 。LinkedList需要从头或尾开始遍历链表，时间复杂度为 $O(n)$ 。
- **空间占用**: ArrayList在创建时需要分配一段连续的内存空间，因此会占用较大的空间。LinkedList每个节点只需要存储元素和指针，因此相对较小。
- **使用场景**: ArrayList适用于频繁随机访问和尾部的插入删除操作，而LinkedList适用于频繁的中间插入删除操作和不需要随机访问的场景。
- **线程安全**: 这两个集合都不是线程安全的，Vector是线程安全的

## ArrayList线程安全吗？把ArrayList变成线程安全有哪些方法？

不是线程安全的，ArrayList变成线程安全的方式有：

- 使用Collections类的synchronizedList方法将ArrayList包装成线程安全的List:

```
List<String> synchronizedList = Collections.synchronizedList(arrayList);
```

java

- 使用CopyOnWriteArrayList类代替ArrayList，它是一个线程安全的List实现：

```
CopyOnWriteArrayList<String> copyOnWriteArrayList = new CopyOnWriteArrayList<>(arrayList);
```

java

- 使用Vector类代替ArrayList，Vector是线程安全的List实现：

```
Vector<String> vector = new Vector<>(arrayList);
```

java

# 为什么ArrayList不是线程安全的，具体来说是哪里不安全？

在高并发添加数据下，ArrayList会暴露三个问题；

- 部分值为null（我们并没有add null进去）
- 索引越界异常
- size与我们add的数量不符

为了知道这三种情况是怎么发生的，ArrayList，add 增加元素的代码如下：

```
public boolean add(E e) {  
    ensureCapacityInternal(size + 1); // Increments modCount!!  
    elementData[size++] = e;  
    return true;  
}
```

java

ensureCapacityInternal()这个方法的详细代码我们可以暂时不看，它的作用就是判断如果将当前的新元素加到列表后面，列表的elementData数组的大小是否满足，如果size + 1的这个需求长度大于了elementData这个数组的长度，那么就要对这个数组进行扩容。

大体可以分为三步：

- 判断数组需不需要扩容，如果需要的话，调用grow方法进行扩容；
- 将数组的size位置设置值（因为数组的下标是从0开始的）；
- 将当前集合的大小加1

下面我们来分析三种情况都是如何产生的：

- 部分值为null：当线程1走到了扩容那里发现当前size是9，而数组容量是10，所以不用扩容，这时候cpu让出执行权，线程2也进来了，发现size是9，而数组容量是10，所以不用扩容，这时候线程1继续执行，将数组下标索引为9的位置set值了，还没有来得及执行size++，这时候线程2也来执行了，又把数组下标索引为9的位置set了一遍，这时候两个先后进行size++，导致下标索引10的地方就为null了。
- 索引越界异常：线程1走到扩容那里发现当前size是9，数组容量是10不用扩容，cpu让出执行权，线程2也发现不用扩容，这时候数组的容量就是10，而线程1 set完之后size++，这时候线程2再进来size就是10，数组的大小只有10，而你要设置下标索引为10的就会越界（数组的下标索引从0开始）；
- size与我们add的数量不符：这个基本上每次都会发生，这个理解起来也很简单，因为size++本身就不原子操作，可以分为三步：获取size的值，将size的值加1，将新的size值覆盖掉原来的，线程1和线程2拿到一样的size值加完了同时覆盖，就会导致一次没有加上，所以肯定不会与我们add的数量保持一致的；

## ArrayList 和 LinkedList 的应用场景？

- ArrayList适用于需要频繁访问集合元素的场景。它基于数组实现，可以通过索引快速访问元素，因此在按索引查找、遍历和随机访问元素的操作上具有较高的性能。当需要频繁访问和遍历集合元素，并且集合大小不经常改变时，推荐使用ArrayList
- LinkedList适用于频繁进行插入和删除操作的场景。它基于链表实现，插入和删除元素的操作只需要调整节点的指针，因此在插入和删除操作上具有较高的性能。当需要频繁进行插入和删除操作，或者集合大小经常改变时，可以考虑使用LinkedList。

## ArrayList的扩容机制说一下

ArrayList在添加元素时，如果当前元素个数已经达到了内部数组的容量上限，就会触发扩容操作。

ArrayList的扩容操作主要包括以下几个步骤：

- 计算新的容量：一般情况下，新的容量会扩大为原容量的1.5倍（在JDK 10之后，扩容策略做了调整），然后检查是否超过了最大容量限制。
- 创建新的数组：根据计算得到的新容量，创建一个新的更大的数组。
- 将元素复制：将原来数组中的元素逐个复制到新数组中。
- 更新引用：将ArrayList内部指向原数组的引用指向新数组。
- 完成扩容：扩容完成后，可以继续添加新元素。

ArrayList的扩容操作涉及到数组的复制和内存的重新分配，所以在频繁添加大量元素时，扩容操作可能会影响性能。为了减少扩容带来的性能损耗，可以在初始化ArrayList时预分配足够大的容量，避免频繁触发扩容操作。

之所以扩容是 1.5 倍，是因为 1.5 可以充分利用移位操作，减少浮点数或者运算时间和运算次数。

```
// 新容量计算  
int newCapacity = oldCapacity + (oldCapacity >> 1);
```

java

## 线程安全的 List， CopyonWriteArrayList是如何实现线程安全的

CopyOnWriteArrayList底层也是通过一个数组保存数据，使用volatile关键字修饰数组，保证当前线程对数组对象重新赋值后，其他线程可以及时感知到。

```
private transient volatile Object[] array;
```

java

在写入操作时，加了一把互斥锁ReentrantLock以保证线程安全。

java

```
public boolean add(E e) {  
    //获取锁  
    final ReentrantLock lock = this.lock;  
    //加锁  
    lock.lock();  
    try {  
        //获取到当前List集合保存数据的数组  
        Object[] elements = getArray();  
        //获取该数组的长度（这是一个伏笔，同时len也是新数组的最后一个元素的索引值）  
        int len = elements.length;  
        //将当前数组拷贝一份的同时，让其长度加1  
        Object[] newElements = Arrays.copyOf(elements, len + 1);  
        //将加入的元素放在新数组最后一位，len不是旧数组长度吗，为什么现在用它当成新数组的最后一个元素的索引值  
        newElements[len] = e;  
        //替换引用，将数组的引用指向给新数组的地址  
        setArray(newElements);  
        return true;  
    } finally {  
        //释放锁  
        lock.unlock();  
    }  
}
```

看到源码可以知道写入新元素时，首先会先将原来的数组拷贝一份并且让原来数组的长度+1后就得到了一个新数组，新数组里的元素和旧数组的元素一样并且长度比旧数组多一个长度，然后将新加入的元素放置都在新数组最后一个位置后，用新数组的地址替换掉老数组的地址就能得到最新的数据了。

在我们执行替换地址操作之前，读取的是老数组的数据，数据是有效数据；执行替换地址操作之后，读取的是新数组的数据，同样也是有效数据，而且使用该方式能比读写都加锁要更加的效率。

现在我们来看读操作，读是没有加锁的，所以读是一直都能读

java

```
public E get(int index) {  
    return get(getArray(), index);  
}
```

3. map:

## 常见的Map集合（非线程安全）：

- `HashMap` 是基于哈希表实现的 `Map`，它根据键的哈希值来存储和获取键值对，JDK 1.8中是用数组+链表+红黑树来实现的。`HashMap` 是非线程安全的，在多线程环境下，当多个线程同时对 `HashMap` 进行操作时，可能会导致数据不一致或出现死循环等问题。比如在扩容时，多个线程可能会同时修改哈希表的结构，从而破坏数据的完整性。
- `LinkedHashMap` 继承自 `HashMap`，它在 `HashMap` 的基础上，使用双向链表维护了键值对的插入顺序或访问顺序，使得迭代顺序与插入顺序或访问顺序一致。由于它继承自 `HashMap`，在多线程并发访问时，同样会出现与 `HashMap` 类似的线程安全问题。
- `TreeMap` 是基于红黑树实现的 `Map`，它可以对键进行排序，默认按照自然顺序排序，也可以通过指定的比较器进行排序。`TreeMap` 是非线程安全的，在多线程环境下，如果多个线程同时对 `TreeMap` 进行插入、删除等操作，可能会破坏红黑树的结构，导致数据不一致或程序出现异常。

## 常见的Map集合（线程安全）：

- `Hashtable` 是早期 Java 提供的线程安全的 `Map` 实现，它的实现方式与 `HashMap` 类似，但在方法上使用了 `synchronized` 关键字来保证线程安全。通过在每个可能修改 `Hashtable` 状态的方法上加上 `synchronized` 关键字，使得在同一时刻，只能有一个线程能够访问 `Hashtable` 的这些方法，从而保证了线程安全。
- `ConcurrentHashMap` 在 JDK 1.8 以前采用了分段锁等技术来提高并发性能。在 `ConcurrentHashMap` 中，将数据分成多个段（Segment），每个段都有自己的锁。在进行插入、删除等操作时，只需要获取相应段的锁，而不是整个 `Map` 的锁，这样可以允许多个线程同时访问不同的段，提高了并发访问的效率。在 JDK 1.8 以后是通过 `volatile + CAS` 或者 `synchronized` 来保证线程安全的。

- 使用 Lambda 表达式和forEach()方法：在 Java 8 及以上版本中，可以使用 Lambda 表达式和 `forEach()` 方法来遍历 `Map`，这种方式更加简洁和函数式。

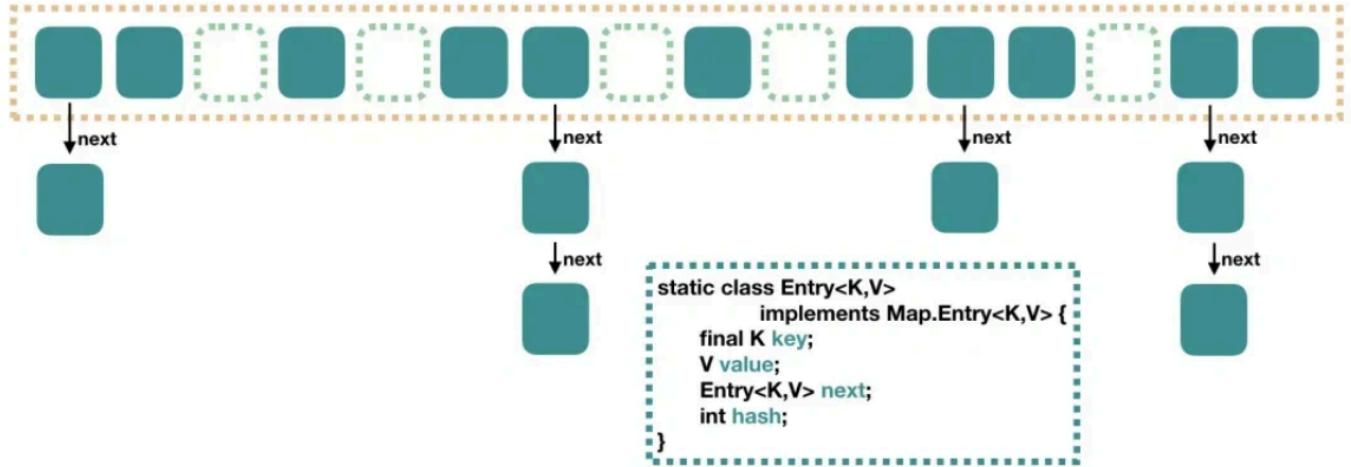
```
java
import java.util.HashMap;
import java.util.Map;

public class MapTraversalExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("key1", 1);
        map.put("key2", 2);
        map.put("key3", 3);

        // 使用Lambda表达式和forEach()方法遍历Map
        map.forEach((key, value) -> System.out.println("Key: " + key + ", Value: " + value));
    }
}
```

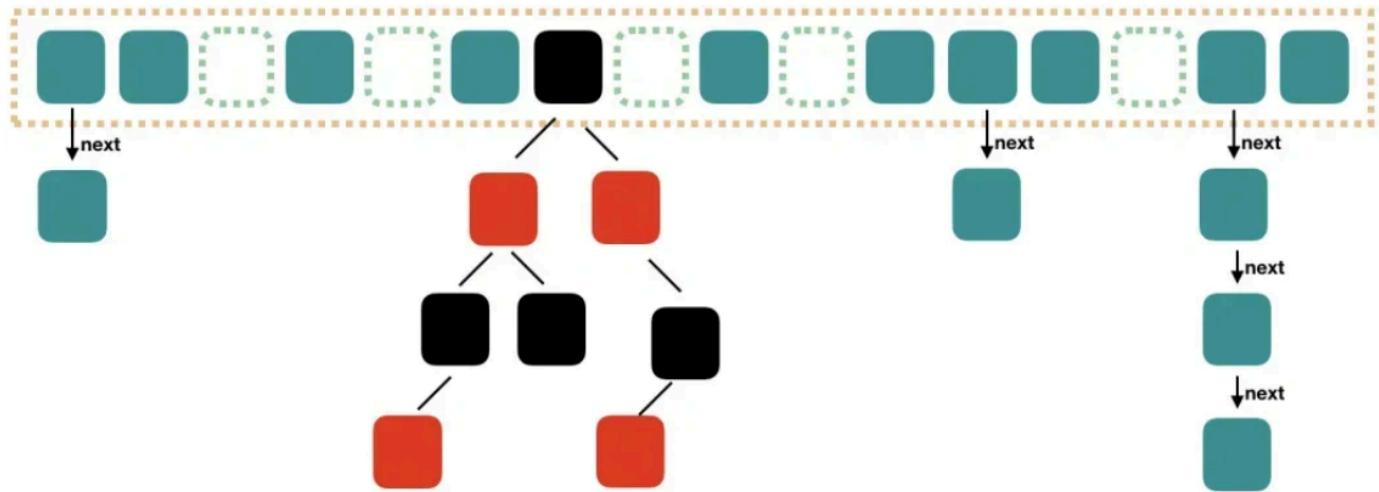
在 JDK 1.7 版本之前，`HashMap` 数据结构是数组和链表，`HashMap` 通过哈希算法将元素的键（Key）映射到数组中的槽位（Bucket）。如果多个键映射到同一个槽位，它们会以链表的形式存储在同一个槽位上，因为链表的查询时间是  $O(n)$ ，所以冲突很严重，一个索引上的链表非常长，效率就很低了。

## Java7 HashMap 结构



所以在 **JDK 1.8** 版本的时候做了优化，当一个链表的长度超过8的时候就转换数据结构，不再使用链表存储，而是使用**红黑树**，查找时使用红黑树，时间复杂度  $O(\log n)$ ，可以提高查询性能，但是在数量较少时，即数量小于6时，会将红黑树转换回链表。

## Java8 HashMap 结构



## 了解的哈希冲突解决方法有哪些？

- 链接法：使用链表或其他数据结构来存储冲突的键值对，将它们链接在同一个哈希桶中。
- 开放寻址法：在哈希表中找到另一个可用的位置来存储冲突的键值对，而不是存储在链表中。常见的开放寻址方法包括线性探测、二次探测和双重散列。
- 再哈希法（Rehashing）：当发生冲突时，使用另一个哈希函数再次计算键的哈希值，直到找到一个空槽来存储键值对。
- 哈希桶扩容：当哈希冲突过多时，可以动态地扩大哈希桶的数量，重新分配键值对，以减少冲突的概率。

| 方法         | 时间复杂度                 | 空间效率 | 适用场景                    |
|------------|-----------------------|------|-------------------------|
| 线性探测       | 查询最坏 $O(n)$           | 高    | 内存敏感的小数据集               |
| 链地址法 (链表)  | 平均 $O(1)$ , 最坏 $O(n)$ | 中    | 通用型键值存储 (如Java HashMap) |
| 链地址法 (红黑树) | 最坏 $O(\log n)$        | 低    | 高冲突场景优化                 |
| 双重哈希       | 平均 $O(1)$             | 高    | 高性能缓存系统                 |
| 再哈希法       | 扩容时 $O(n)$            | 动态调整 | 数据量波动大的实时系统             |
| 公共溢出区      | 溢出区 $O(m)$            | 高    | 内存受限的专用设备               |

### 实际应用策略:

- **数据库索引**: 多采用链地址法 (B+树优化)
- **分布式缓存**: 组合使用一致性哈希与再哈希 (如Redis Cluster)
- **实时风控系统**: 开放定址法 + SIMD指令加速探测

## HashMap是线程安全的吗?

hashmap不是线程安全的， hashmap在多线程会存在下面的问题：

- JDK 1.7 HashMap 采用数组 + 链表的数据结构，多线程背景下，在数组扩容的时候，存在 Entry 链死循环和数据丢失问题。
- JDK 1.8 HashMap 采用数组 + 链表 + 红黑二叉树的数据结构，优化了 1.7 中数组扩容的方案，解决了 Entry 链死循环和数据丢失问题。但是多线程背景下，put 方法存在数据覆盖的问题。

如果要保证线程安全，可以通过这些方法来保证：

- 多线程环境可以使用Collections.synchronizedMap同步加锁的方式，还可以使用HashTable，但是同步的方式显然性能不达标，而ConcurrentHashMap更适合高并发场景使用。
- ConcurrentHashMap在JDK1.7和1.8的版本改动比较大，1.7使用Segment+HashEntry分段锁的方式实现，1.8则抛弃了Segment，改为使用CAS+synchronized+Node实现，同样也加入了红黑树，避免链表过长导致性能的问题。

HashMap HashMap的put()方法用于向HashMap中添加键值对，当调用HashMap的put()方法时，会按照以下详细流程执行（JDK8 1.8版本）：

第一步：根据要添加的键的哈希码计算在数组中的位置（索引）。

第二步：检查该位置是否为空（即没有键值对存在）

- 如果为空，则直接在该位置创建一个新的Entry对象来存储键值对。将要添加的键值对作为该Entry的键和值，并保存在数组的对应位置。将HashMap的修改次数（modCount）加1，以便在进行迭代时发现并发修改。

第三步：如果该位置已经存在其他键值对，检查该位置的第一个键值对的哈希码和键是否与要添加的键值对相同？

- 如果相同，则表示找到了相同的键，直接将新的值替换旧的值，完成更新操作。

第四步：如果第一个键值对的哈希码和键不相同，则需要遍历链表或红黑树来查找是否有相同的键：

如果键值对集合是链表结构，从链表的头部开始逐个比较键的哈希码和equals()方法，直到找到相同的键或达到链表末尾。

- 如果找到了相同的键，则使用新的值取代旧的值，即更新键对应的值。
- 如果没有找到相同的键，则将新的键值对添加到链表的头部。

如果键值对集合是红黑树结构，在红黑树中使用哈希码和equals()方法进行查找。根据键的哈希码，定位到红黑树中的某个节点，然后逐个比较键，直到找到相同的键或达到红黑树末尾。

- 如果找到了相同的键，则使用新的值取代旧的值，即更新键对应的值。
- 如果没有找到相同的键，则将新的键值对添加到红黑树中。

第五步：检查链表长度是否达到阈值（默认为8）：

- 如果链表长度超过阈值，且HashMap的数组长度大于等于64，则会将链表转换为红黑树，以提高查询效率。

## 第六步：检查负载因子是否超过阈值（默认为0.75）：

- 如果键值对的数量 (size) 与数组的长度的比值大于阈值，则需要进行扩容操作。

## 第七步：扩容操作：

- 创建一个新的两倍大小的数组。
- 将旧数组中的键值对重新计算哈希码并分配到新数组中的位置。
- 更新HashMap的数组引用和阈值参数。

## 第八步：完成添加操作。

此外，HashMap是非线程安全的，如果在多线程环境下使用，需要采取额外的同步措施或使用线程安全的 ConcurrentHashMap。

## HashMap的put(key,val)和get(key)过程

- 存储对象时，我们将K/V传给put方法时，它调用hashCode计算hash从而得到bucket位置，进一步存储，HashMap会根据当前bucket的占用情况自动调整容量(超过Load Factor则resize为原来的2倍)。
- 获取对象时，我们将K传给get，它调用hashCode计算hash从而得到bucket位置，并进一步调用equals()方法确定键值对。如果发生碰撞的时候，HashMap通过链表将产生碰撞冲突的元素组织起来，在Java 8中，如果一个bucket中碰撞冲突的元素超过某个限制(默认是8)，则使用红黑树来替换链表，从而提高速度。

## hashmap 调用get方法一定安全吗？

不是，调用 get 方法有几点需要注意的地方：

- **空指针异常 (NullPointerException)**：如果你尝试用 `null` 作为键调用 `get` 方法，而 `HashMap` 没有被初始化（即为 `null`），那么会抛出空指针异常。不过，如果 `HashMap` 已经初始化，使用 `null` 作为键是允许的，因为 `HashMap` 支持 `null` 键。
- **线程安全**：`HashMap` 本身不是线程安全的。如果在多线程环境中，没有适当的同步措施，同时对 `HashMap` 进行读写操作可能会导致不可预测的行为。例如，在一个线程中调用 `get` 方法读取数据，而另一个线程同时修改了结构（如增加或删除元素），可能会导致读取操作得到错误的结果或抛出 `ConcurrentModificationException`。如果需要在多线程环境中使用类似 `HashMap` 的数据结构，可以考虑使用 `ConcurrentHashMap`。

## HashMap一般用什么做Key？为啥String适合做Key呢？

用 string 做 key，因为 String对象是不可变的，一旦创建就不能被修改，这确保了Key的稳定性。如果Key是可变的，可能会导致hashCode和equals方法的不一致，进而影响HashMap的正确性。

## 为什么HashMap要用红黑树而不是平衡二叉树？

- 平衡二叉树追求的是一种“**完全平衡**”状态：任何结点的左右子树的高度差不会超过 1，优势是树的结点是很平均分配的。这个要求实在是太严了，导致每次进行插入/删除节点的时候，几乎都会破坏平衡树的第二个规则，进而我们都需要通过**左旋**和**右旋**来进行调整，使之再次成为一颗符合要求的平衡树。
- 红黑树不追求这种完全平衡状态，而是追求一种“**弱平衡**”状态：整个树最长路径不会超过最短路径的 2 倍。优势是虽然牺牲了一部分查找的性能效率，但是能够换取一部分维持树平衡状态的成本。与平衡树不同的是，红黑树在插入、删除等操作，**不会像平衡树那样，频繁着破坏红黑树的规则，所以不需要频繁着调整**，这也是我们为什么大多数情况下使用红黑树的原因。

## hashmap key可以为null吗？

可以为 null。

- hashMap中使用hash()方法来计算key的哈希值，当key为空时，直接另key的哈希值为0，不走key.hashCode()方法；

```
static final int hash(Object key) {  
    int h;  
    //当key等于null的时候，不走hashCode()方法  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}
```

- hashMap虽然支持key和value为null，但是null作为key只能有一个，null作为value可以有多个；
- 因为hashMap中，如果key值一样，那么会覆盖相同key值的value为最新，所以key为null只能有一个。

## 重写HashMap的equal和hashcode方法需要注意什么？

HashMap使用Key对象的hashCode()和equals方法去决定key-value对的索引。当我们试着从HashMap中获取值的时候，这些方法也会被用到。如果这些方法没有被正确地实现，在这种情况下，两个不同Key也许会产生相同的hashCode()和equals()输出，HashMap将会认为它们是相同的，然后覆盖它们，而非把它们存储到不同的地方。

同样的，所有不允许存储重复数据的集合类都使用hashCode()和equals()去查找重复，所以正确实现它们非常重要。equals()和hashCode()的实现应该遵循以下规则：

- 如果o1.equals(o2)，那么o1.hashCode() == o2.hashCode()总是为true的。
- 如果o1.hashCode() == o2.hashCode()，并不意味着o1.equals(o2)会为true。

## 重写HashMap的equal方法不当会出现什么问题?

HashMap在比较元素时，会先通过hashCode进行比较，相同的情况下再通过equals进行比较。

所以 equals相等的两个对象， hashCode一定相等。 hashCode相等的两个对象， equals不一定相等（比如散列冲突的情况）

重写了equals方法，不重写hashCode方法时，可能会出现equals方法返回为true，而hashCode方法却返回false，这样的一个后果会导致在hashmap等类中存储多个一模一样的对象，导致出现覆盖存储的数据的问题，这与 hashmap只能有唯一的key的规范不符合。

## 列举HashMap在多线程下可能会出现的问题?

- JDK1.7中的 HashMap 使用头插法插入元素，在多线程的环境下，扩容的时候有可能导致环形链表的出现，形成死循环。因此，JDK1.8使用尾插法插入元素，在扩容时会保持链表元素原本的顺序，不会出现环形链表的问题。
- 多线程同时执行 put 操作，如果计算出来的索引位置是相同的，那会造成前一个 key 被后一个 key 覆盖，从而导致元素的丢失。此问题在JDK 1.7和 JDK 1.8 中都存在。

## HashMap的扩容机制介绍一下

hashMap默认的负载因子是0.75，即如果hashmap中的元素个数超过了总容量75%，则会触发扩容，扩容分为两个步骤：

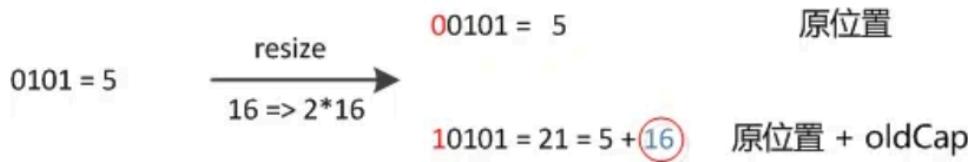
- 第1步是对哈希表长度的扩展（2倍）
- 第2步是将旧哈希表中的数据放到新的哈希表中。

因为我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。

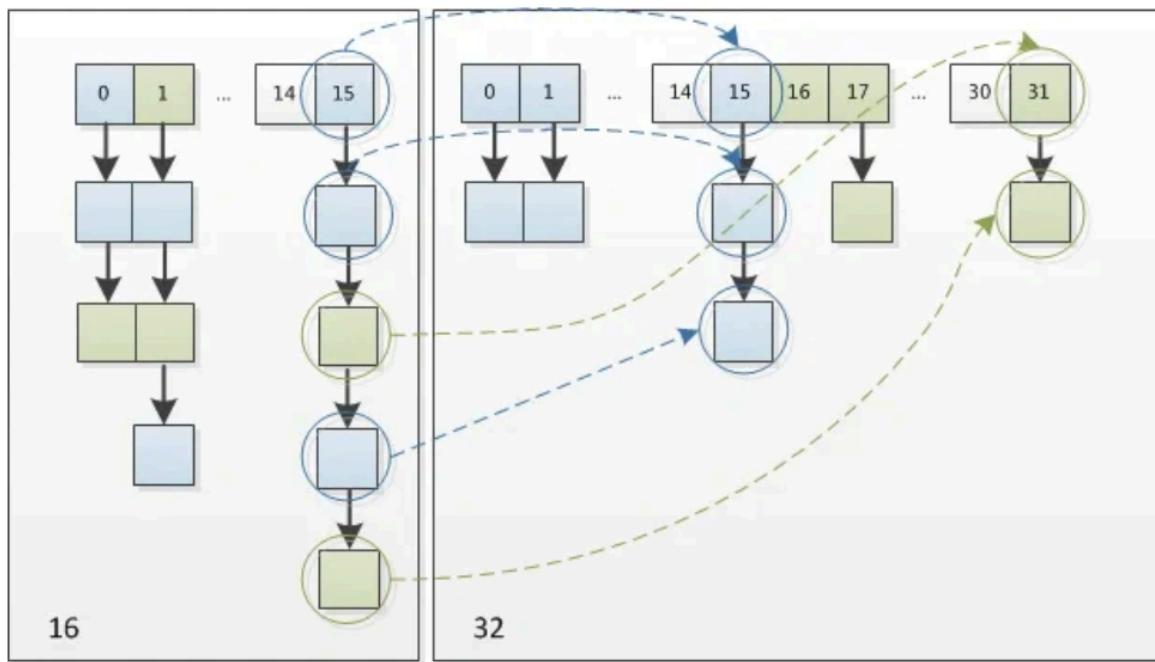
如我们从16扩展为32时，具体的变化如下所示：

|       |                                                |   |                                                 |
|-------|------------------------------------------------|---|-------------------------------------------------|
| n - 1 | 0000 0000 0000 0000 0000 0000 <b>1111</b>      | → | 1111 1111 1111 1111 0000 1111 000 <b>1 1111</b> |
| hash1 | 1111 1111 1111 1111 0000 1111 0000 <b>0101</b> | → | 1111 1111 1111 1111 0000 1111 000 <b>0 0101</b> |
| hash2 | 1111 1111 1111 1111 0000 1111 0001 <b>0101</b> |   | 1111 1111 1111 1111 0000 1111 000 <b>1 0101</b> |

因此元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



因此，我们在扩充HashMap的时候，不需要重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”。可以看看下图为16扩充为32的resize示意图：



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。

HashMap初始大小为16

## HashMap的大小为什么是2的n次方大小呢?

在 JDK1.7 中，HashMap 整个扩容过程就是分别取出数组元素，一般该元素是最后一个放入链表中的元素，然后遍历以该元素为头的单向链表元素，依据每个被遍历元素的 hash 值计算其在新数组中的下标，然后进行交换。这样的扩容方式会将原来哈希冲突的单向链表尾部变成扩容后单向链表的头部。

而在 JDK 1.8 中，HashMap 对扩容操作做了优化。由于扩容数组的长度是 2 倍关系，所以对于假设初始 `tableSize = 4` 要扩容到 8 来说就是 0100 到 1000 的变化（左移一位就是 2 倍），在扩容中只用判断原来的 hash 值和左移动的一位 (`newTable` 的值) 按位与操作是 0 或 1 就行，0 的话索引不变，1 的话索引变成原索引加上扩容前数组。

之所以能通过这种“与运算”来重新分配索引，是因为 hash 值本来就是随机的，而 hash 按位与上 `newTable` 得到的 0 (扩容前的索引位置) 和 1 (扩容前索引位置加上扩容前数组长度的数值索引处) 就是随机的，所以扩容的过程就能把之前哈希冲突的元素再随机分布到不同的索引中去。

## 说说hashmap的负载因子

HashMap 负载因子 `loadFactor` 的默认值是 0.75，当 HashMap 中的元素个数超过了容量的 75% 时，就会进行扩容。

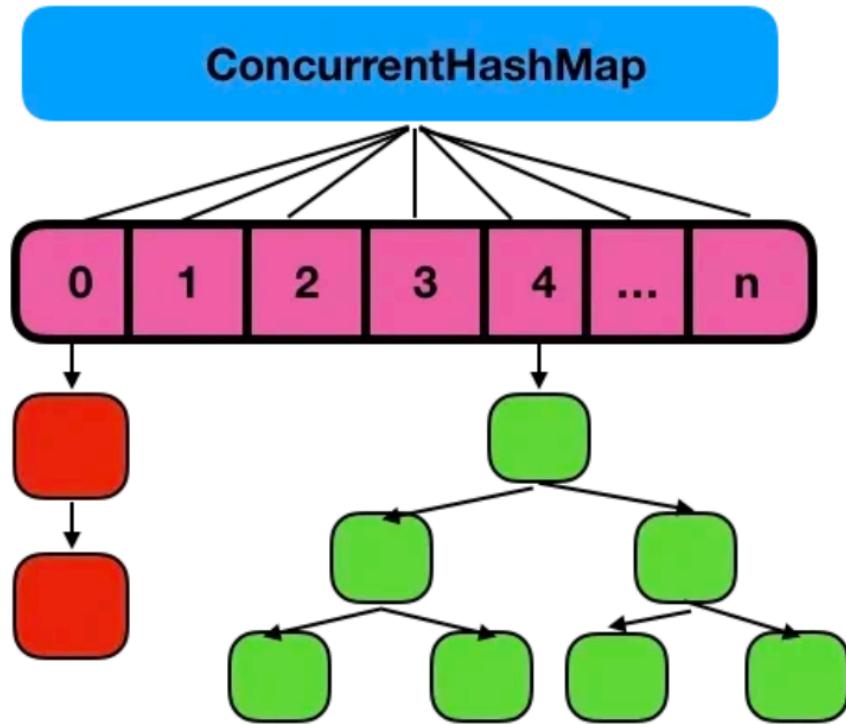
默认负载因子为 0.75，是因为它提供了空间和时间复杂度之间的良好平衡。

负载因子太低会导致大量的空桶浪费空间，负载因子太高会导致大量的碰撞，降低性能。0.75 的负载因子在这两个因素之间取得了良好的平衡。

## Hashmap和Hashtable有什么不一样的? Hashmap一般怎么用?

- **HashMap线程不安全**，效率高一点，可以存储null的key和value，null的key只能有一个，null的value可以有多个。默认初始容量为16，每次扩充变为原来2倍。创建时如果给定了初始容量，则扩充为2的幂次方大小。底层数据结构为数组+链表，插入元素后如果链表长度大于阈值（默认为8），先判断数组长度是否小于64，如果小于，则扩充数组，反之将链表转化为红黑树，以减少搜索时间。
- **HashTable线程安全**，效率低一点，其内部方法基本都经过synchronized修饰，不可以有null的key和value。默认初始容量为11，每次扩容变为原来的 $2n+1$ 。创建时给定了初始容量，会直接用给定的大小。底层数据结构为数组+链表。它基本被淘汰了，要保证线程安全可以用ConcurrentHashMap。
- **怎么用**：HashMap主要用来存储键值对，可以调用put方法向其中加入元素，调用get方法获取某个键对应的值，也可以通过containsKey方法查看某个键是否存在等

以在数据比较多的情况下访问是很慢的，因为要遍历整个链表，而 JDK 1.8 则使用了数组 + 链表/红黑树的方式优化了 ConcurrentHashMap 的实现，具体实现结构如下：



JDK 1.8 ConcurrentHashMap 主要通过 volatile + CAS 或者 synchronized 来实现的线程安全的。添加元素时首先会判断容器是否为空：

- 如果为空则使用 volatile 加 CAS 来初始化
- 如果容器不为空，则根据存储的元素计算该位置是否为空。
  - 如果根据存储的元素计算结果为空，则利用 CAS 设置该节点；
  - 如果根据存储的元素计算结果不为空，则使用 synchronized，然后，遍历桶中的数据，并替换或新增节点到桶中，最后再判断是否需要转为红黑树，这样就能保证并发访问时的线程安全了。

如果把上面的执行用一句话归纳的话，就相当于是 ConcurrentHashMap 通过对头结点加锁来保证线程安全的，锁的粒度相比 Segment 来说更小了，发生冲突和加锁的频率降低了，并发操作的性能就提高了。

而且 JDK 1.8 使用的是红黑树优化了之前的固定链表，那么当数据量比较大的时候，查询性能也得到了很大的提升，从之前的  $O(n)$  优化到了  $O(\log n)$  的时间复杂度。

## 分段锁怎么加锁的？

在 ConcurrentHashMap 中，将整个数据结构分为多个 Segment，每个 Segment 都类似于一个小的 HashMap，每个 Segment 都有自己的锁，不同 Segment 之间的操作互不影响，从而提高并发性能。

在 ConcurrentHashMap 中，对于插入、更新、删除等操作，需要先定位到具体的 Segment，然后再在该 Segment 上加锁，而不是像传统的 HashMap 一样对整个数据结构加锁。这样可以使得不同 Segment 之间的操作并行进行，提高了并发性能。

## 分段锁是可重入的吗？

JDK 1.7 ConcurrentHashMap 中的分段锁是用了 ReentrantLock，是一个可重入的锁。

## 已经用了 synchronized，为什么还要用 CAS 呢？

ConcurrentHashMap 使用这两种手段来保证线程安全主要是一种权衡的考虑，在某些操作中使用 synchronized，还是使用 CAS，主要是根据锁竞争程度来判断的。

比如：在 putVal 中，如果计算出来的 hash 槽没有存放元素，那么就可以直接使用 CAS 来进行设置值，这是因为在设置元素的时候，因为 hash 值经过了各种扰动后，造成 hash 碰撞的几率较低，那么我们可以预测使用较少的自旋来完成具体的 hash 落槽操作。

当发生了 hash 碰撞的时候说明容量不够用了或者已经有大量线程访问了，因此这时候使用 synchronized 来处理 hash 碰撞比 CAS 效率要高，因为发生了 hash 碰撞大概率来说是线程竞争比较强烈。

## Java 8+ (CAS + 细粒度 synchronized)

• **数据结构升级**: 摒弃 Segment, 哈希桶改为 Node 数组, 支持链表与红黑树 (链表长度  $\geq 8$  时树化)。

• **加锁机制**:

◦ **读操作**:

◦ 完全无锁, 通过 volatile 关键字和 Unsafe 类的原子操作 (如 getObjectVolatile) 保证可见性。

◦ **写操作**:

◦ **空桶插入**: 通过 CAS 原子操作插入新节点 (无需锁)。

Java

```
if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {  
    if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value)))  
        break; // CAS 成功, 无需锁  
}
```

◦ **哈希冲突处理**: 锁定链表头节点或红黑树根节点 (synchronized 块)。

Java

```
synchronized (f) { // 锁住链表头节点  
    if (tabAt(tab, i) == f) {  
        // 处理链表或红黑树的插入/更新  
    }  
}
```

◦ **扩容**:

◦ 多线程协助扩容, 迁移数据时通过 ForwardingNode 标记, 避免全表锁。

• **特点**:

◦ **锁粒度**: **单个哈希桶的头节点级别** (冲突时锁定链表的头节点)。

◦ **并发性能**: CAS 无锁化提升插入速度, 细粒度锁减少竞争。

| 版本      | 锁机制                       | 粒度          | 适用场景          |
|---------|---------------------------|-------------|---------------|
| Java 7  | ReentrantLock (Segment级锁) | 分段锁 (默认16段) | 低并发写、简单查询场景   |
| Java 8+ | CAS + synchronized (桶级锁)  | 单个桶的头节点     | 高并发读写、大规模数据存储 |

#### 4. 性能优化核心

- CAS 优势：
  - 无锁插入空桶，避免不必要的线程阻塞。
  - 适合低冲突场景（如新键插入），减少锁开销。
- synchronized 优化：
  - JDK 6+ 对 synchronized 进行了锁升级优化（偏向锁 → 轻量级锁 → 重量级锁），覆盖多数短时锁竞争场景。
  - 桶级锁将锁竞争范围限制在单个哈希桶内，并发度显著提升。

#### 5. 实际应用建议

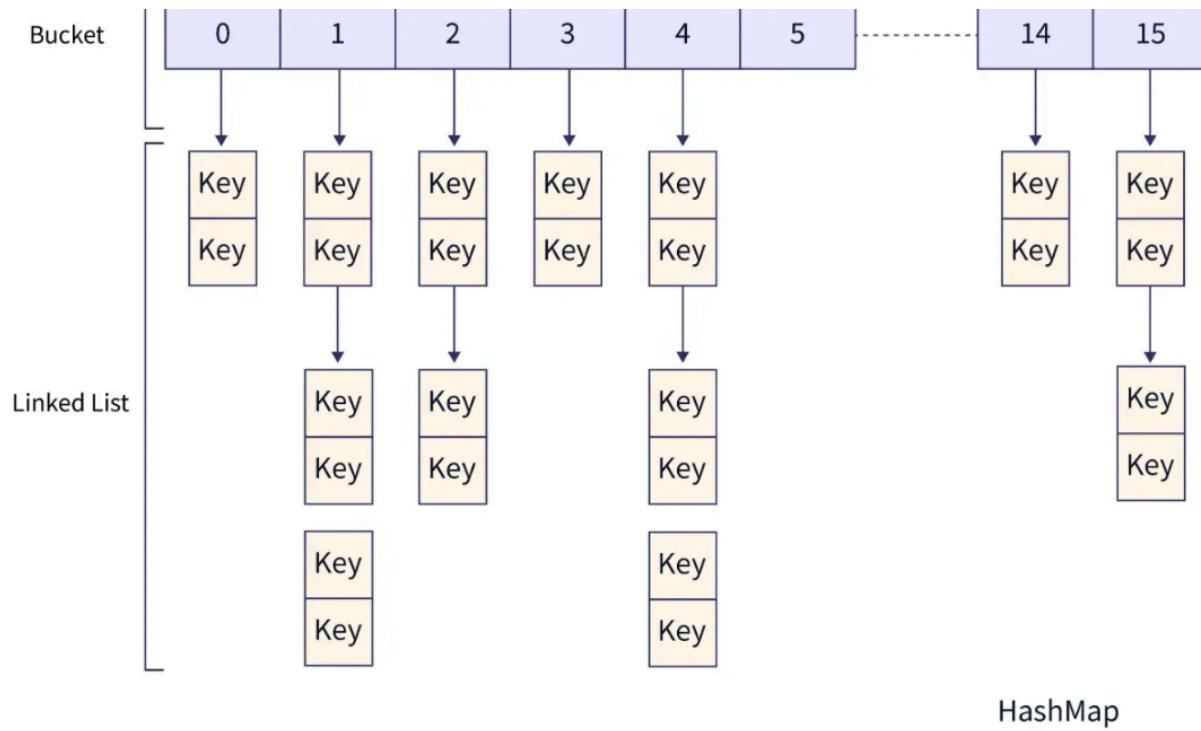
1. 优先使用 Java 8+ 版本：默认性能更优，无需手动分段。

### ConcurrentHashMap用了悲观锁还是乐观锁？

悲观锁和乐观锁都有用到。

添加元素时首先会判断容器是否为空：

- 如果为空则使用 volatile 加 **CAS（乐观锁）** 来初始化。
- 如果容器不为空，则根据存储的元素计算该位置是否为空。
- 如果根据存储的元素计算结果为空，则利用 **CAS（乐观锁）** 设置该节点；
- 如果根据存储的元素计算结果不为空，则使用 **synchronized（悲观锁）**，然后，遍历桶中的数据，并替换或新增节点到桶中，最后再判断是否需要转为红黑树，这样就能保证并发访问时的线程安全了。



- Hashtable的底层数据结构主要是[数组加上链表](#)，数组是主体，链表是解决hash冲突存在的。
- HashTable是线程安全的，实现方式是[Hashtable的所有公共方法均采用synchronized关键字](#)，当一个线程访问同步方法，另一个线程也访问的时候，就会陷入阻塞或者轮询的状态。

可以看到，[Hashtable是通过使用了 synchronized 关键字来保证其线程安全。](#)

在Java中，可以使用synchronized关键字来标记一个方法或者代码块，当某个线程调用该对象的synchronized方法或者访问synchronized代码块时，这个线程便获得了该对象的锁，其他线程暂时无法访问这个方法，只有等待这个方法执行完毕或者代码块执行完毕，这个线程才会释放该对象的锁，其他线程才能执行这个方法或者代码块。

## Hashtable 和 ConcurrentHashMap有什么区别

### 底层数据结构：

- jdk7之前的ConcurrentHashMap底层采用的是分段的数组+链表实现，jdk8之后采用的是数组+链表/红黑树；
- HashTable采用的是数组+链表，数组是主体，链表是解决hash冲突存在的。

### 实现线程安全的方式：

- jdk8以前，ConcurrentHashMap采用分段锁，对整个数组进行了分段分割，每一把锁只锁容器里的一部分数据，多线程访问不同数据段里的数据，就不会存在锁竞争，提高了并发访问；jdk8以后，直接采用数组+链表/红黑树，并发控制使用CAS和synchronized操作，更加提高了速度。
- HashTable：所有的方法都加了锁来保证线程安全，但是效率非常的低下，当一个线程访问同步方法，另一个线程也访问的时候，就会陷入阻塞或者轮询的状态。

## 说一下HashMap和Hashtable、ConcurrentMap的区别

- HashMap线程不安全，效率高一点，可以存储null的key和value，null的key只能有一个，null的value可以有多个。默认初始容量为16，每次扩充变为原来2倍。创建时如果给定了初始容量，则扩充为2的幂次方大小。底层数据结构为数组+链表，插入元素后如果链表长度大于阈值（默认为8），先判断数组长度是否小于64，如果小于，则扩充数组，反之将链表转化为红黑树，以减少搜索时间。
- HashTable线程安全，效率低一点，其内部方法基本都经过synchronized修饰，不可以有null的key和value。默认初始容量为11，每次扩容变为原来的 $2n+1$ 。创建时给定了初始容量，会直接用给定的大小。底层数据结构为数组+链表。它基本被淘汰了，要保证线程安全可以用ConcurrentHashMap。
- ConcurrentHashMap是Java中的一个线程安全的哈希表实现，它可以在多线程环境下并发地进行读写操作，而不需要像传统的HashTable那样在读写时加锁。ConcurrentHashMap的实现原理主要基于分段锁和CAS操作。它将整个哈希表分成了多Segment（段），每个Segment都类似于一个小的HashMap，它拥有自己的数组和一个独立的锁。在ConcurrentHashMap中，读操作不需要锁，可以直接对Segment进行读取，而写操作则只需要锁定对应的Segment，而不是整个哈希表，这样可以大大提高并发性能。

| 层级   | 实现机制                | 保障目标         |
|------|---------------------|--------------|
| 硬件层  | CPU 原子指令 (如CMPXCHG) | 物理内存操作的不可分割性 |
| JVM层 | Unsafe 类 + 内存屏障     | 跨平台适配与内存一致性  |
| 语义层  | 复合操作的原子性封装          | 开发者无需关心底层细节  |

通过这三层协同，CAS既能实现高效的无锁并发（吞吐量比锁高5-10倍），又能保证线程安全，成为现代高并发框架的核心基础。

4. Set:

## Set集合有什么特点？如何实现key无重复的？

- **set集合特点：**Set集合中的元素是唯一的，不会出现重复的元素。
- **set实现原理：**Set集合通过内部的数据结构（如哈希表、红黑树等）来实现key的无重复。当向Set集合中插入元素时，会先根据元素的hashCode值来确定元素的存储位置，然后再通过equals方法来判断是否已经存在相同的元素，如果存在则不会再次插入，保证了元素的唯一性。

## 有序的Set是什么？记录插入顺序的集合是什么？

- **有序的 Set 是TreeSet和LinkedHashSet。** TreeSet是基于红黑树实现，保证元素的自然顺序。LinkedHashSet是基于双重链表和哈希表的结合来实现元素的有序存储，保证元素添加的自然顺序
- **记录插入顺序的集合通常指的是LinkedHashSet，** 它不仅保证元素的唯一性，还可以保持元素的插入顺序。当需要在Set集合中记录元素的插入顺序时，可以选择使用LinkedHashSet来实现。

# 1. 理解集合

## 1. 集合与数组对比：

数组：

- 1) 长度开始时必须指定，而且一旦指定，不能更改
- 2) 保存的必须为同一类型的元素
- 3) 使用数组进行增加/删除元素的示意代码 – 比较麻烦

**写出Person数组扩容示意代码。**

```
Person[] pers = new Person[1]; //大小是1  
per[0]=new Person();
```

**//增加新的Person对象？**

```
Person[] pers2 = new Person[pers.length+1]; //新创建数组  
for(){} //拷贝pers数组的元素到pers2  
pers2[pers2.length-1]=new Person();//添加新的对象
```

集合：

- 1) 可以动态保存任意多个对象，使用比较方便！
- 2) 提供了一系列方便的操作对象的方法：add、remove、set、get等
- 3) 使用集合添加/删除新元素的示意代码 - 简洁了

## 2. 集合的整体总览框架体系(重点!)：

集合主要有**单列集合**和**双列集合**

## 1. Iterable

i. Collection接口(都是单列集合)(单单储存值)

```
ArrayList arrayList = new ArrayList();
```

```
arrayList.add("jack");
```

```
arrayList.add("tom");
```

a. List接口

a. Vector

b. ArrayList

c. LinkedList

b. Set接口

a. TreeSet

b. HashSet

## 2. Map接口(都是双列集合)(K-V存储,键key和值value一一对应)

```
HashMap hashMap = new HashMap();
```

```
hashMap.put("NO1", "北京");
```

```
hashMap.put("NO2", "上海");
```



i. HashMap

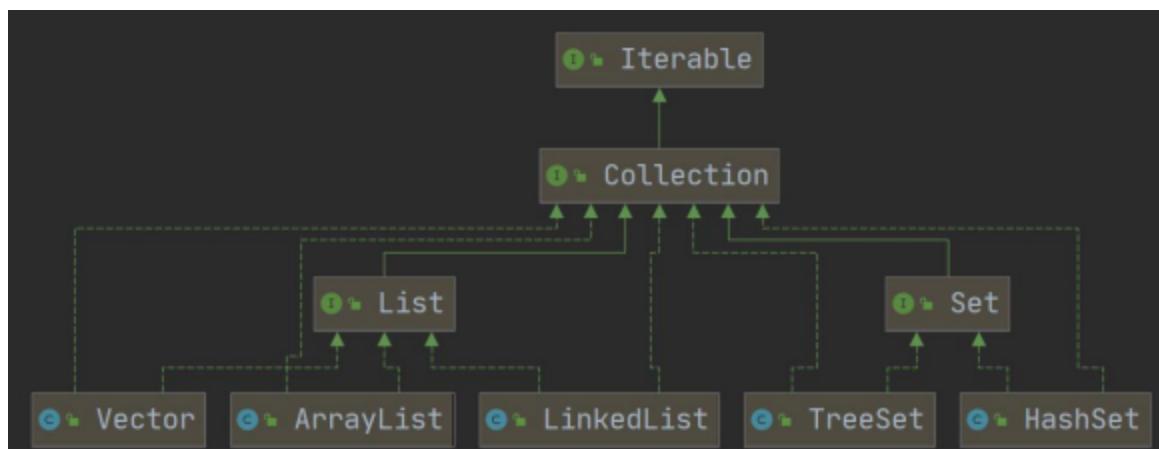
a. LinkedHashMap

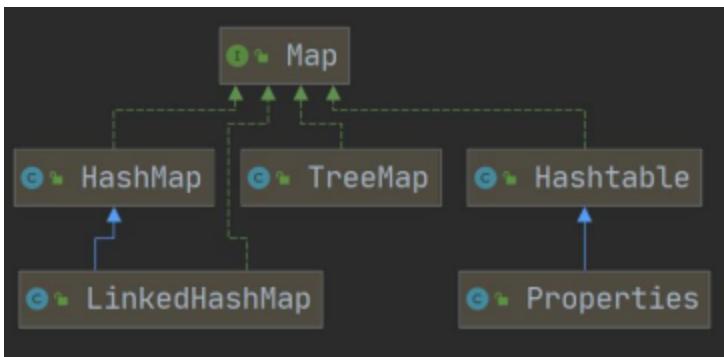
ii. TreeMap

iii. Hashtable

a. Properties

## 3. 总览图





## 2. 具体接口及常用方法

### 1. Collection接口

#### 1. 特点:

1. Collection实现子类可以存放多个元素,每个元素可以是Object
2. 有些Collention实现类可以存放重复元素,有些不行
3. 有些Collention实现类有序(List),有些不有序(Set)
4. Collection接口没有直接的实现子类,而是通过子接口Set和List实现

#### 2. Collection常用方法(以List为例,其实都有这些方法):

##### 1. add:添加单个元素

```

//import java.util.ArrayList;
//import java.util.List;
List list = new ArrayList();
list.add("lover");
list.add(520);
list.add(true);
list.add(1314.0)
list.add(2);
list.add(4); //集合可以动态保存任意类型数据
System.out.println(list);
//["lover", 520, true, 1314.0, 2, 4]
  
```

##### 2. remove:删除指定元素

```

list.remove(2);
//["lover", 520, 1314.0, 2, 4]
//可在括号内输入想删除的元素/索引
//如有冲突,优先以索引为标准
  
```

### **3. contains:查找元素是否存在(返回boolean类型)**

```
System.out.println(list.contains("lover"));//true
```

### **4. size:获取元素个数**

```
System.out.println(list.size());//5
```

### **5. isEmpty:判断是否为空**

```
System.out.println(list.isEmpty());//false
```

### **6. clear:清空list**

```
list.clear();//list变成了[]  
//假装这句语句不算数
```

### **7. addAll:添加多个元素**

```
ArrayList list2 = new ArrayList();  
list2.add("xjh");  
list2.add("hxq");  
list.addAll(6,list2);  
//前面的6表示在哪个索引位置插入(将该索引位置及之后的元素后移)  
//但是添加索引最大值为list.size(),即最后一个元素的下一个位置  
//索引可以没有,默认加到末尾  
//[lover, 520, 1314.0, 2, 4, xjh, hxq]
```

### **8. containsAll:查找每个元素是否都存在**

```
ArrayList list2 = new ArrayList();  
list2.add("lover");  
list2.add(1314.0);  
System.out.println(list.containsAll(list2));//true  
//不考虑元素顺序排列,仅看是否都存在
```

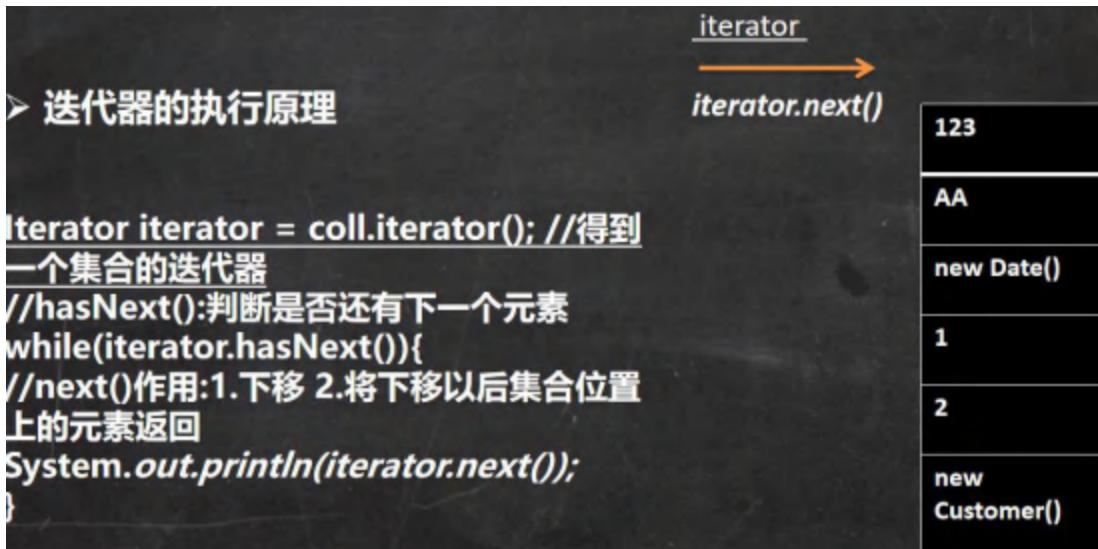
### **9. removeAll:删除多个元素**

```
ArrayList list3 = new ArrayList();  
list3.add(2);  
list3.add(4);  
list.removeAll(list3);  
System.out.println(list);  
//[lover, 520, 1314.0, xjh, hxq]
```

### 3.1 遍历元素的方式1:Iterator(迭代器)

#### 1. 介绍Iterator(迭代器)

1. Iterator对象称为迭代器,主要用于遍历Collection集合的元素,本身不存放对象(负责遍历,不负责储存)
2. 所有实现Collection接口的集合类都有一个iterator()方法,用于返回一个实现Iterator接口的对象(返回迭代器)
3. 原理如图:



#### 2. Iterator接口方法

##### 1. 方法:

- i. hasNext():判断是否还有下一个元素
- ii. next():下移并将下移后集合位置的元素返回,但是必须先调用hasNext()  
//先判断有next才能到next
- iii. remove():

##### 2. 使用示例:

```
//import.java.util.Iterator;
Collection col = new ArrayList();
col.add(new Book("三国演义", "罗贯中", 10.1));
col.add(new Book("小李飞刀", "古龙", 5.1));
col.add(new Book("红楼梦", "曹雪芹", 34.6));
Iterator iterator = col.iterator(); //先得到对应的迭代器
//每次遍历都需要重置迭代器
while (iterator.hasNext()) {
Object obj = iterator.next();
System.out.println("obj=" + obj);
```

```
}
```

```
//退出循环时iterator迭代器指向最后的元素
```

### 3.2 遍历元素的方式2:for循环增强(就是简化版的迭代器)

#### 1. 基本语法:

```
for(元素类型 元素名:集合名/数组名){  
    访问元素  
}
```

#### 2. 使用示例:

```
ArrayList list = new ArrayList();  
Dog dog1 = new Dog("大黄",3984);  
Dog dog2 = new Dog("无上尊狗",9999999);  
list.add(dog1);  
list.add(dog2);  
for(Object i:list){//Object类型多好用嘛  
    System.out.println(i);  
}
```

```
while(iterator.hasNext()){//迭代器使用示例  
    Dog d = (Dog)iterator.next();  
    System.out.println(d);  
}  
Dog [name=大黄, age=3984]  
Dog [name=无上尊狗, age=9999999]  
Dog [name=大黄, age=3984]  
Dog [name=无上尊狗, age=9999999]
```

#### 4. List(是Collection接口的子接口)

对于线程安全的 `List`，如 `CopyOnWriteArrayList`，由于其采用了写时复制的机制，在遍历的同时可以进行修改操作，不会抛出 `ConcurrentModificationException` 异常，但可能会读取到旧的数据，因为修改操作是在新的副本上进行的。

#### 1. 介绍:

1. List接口是Collection接口的子接口
2. List中的**元素有序(添加与取出顺序一致),且可重复**
3. 因为有序,所以**支持索引**的存在

#### 4. java.util 接口 List<E>

所有超级接口：

Collection<E>, Iterable<E>

所有已知实现类：

AbstractList, AbstractSequentialList, ArrayList, AttributeList,  
CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

**常用的有： ArrayList、 LinkedList 和 Vector。**

### 2. List接口的常用方法

1. list.add(index, Object obj);  
//index作用同addAll
2. list.addAll(index, list2);  
//在index的位置插入多个元素,原index及后面的元素集体后移,若无index,默认加在末尾  
//index的值最大为list.size(),即最后一个元素的下一个位置
3. String str = list.get(int index); //获得指定index位置的元素
4. int index = list.indexOf(Object obj); //返回obj首次出现的位置
5. int lastindex = list.lastIndexOf(Object obj); //返回obj最后出现的位置
6. String back = list.remove(index); //移除index索引处的元素并返回
7. list.set(int index, Object obj); //将obj替换index位置的元素
8. List listSub = list.subList(start, end); //返回list索引start到end的子集合(前闭后开)

### 3. List的三种遍历方式:(ArrayList和LinkedList用法相同)

1. iterator(迭代器,同Collection演示)

```
Iterator it = list.iterator();
while(it.hasNext()){
    Object o = it.next();
    System.out.println(o);
    //直接System.out.println(it.next());也行
}
```

2. 增强for:

```
for(Object i:list){
    System.out.println(i);
}
```

3. 普通for

```
for(int i = 0 ;i<list.size() ;i++){
    Object o = list.get(i);
}
```

```
System.out.println(o);
}

1) 方式一：使用iterator
Iterator iter = col.iterator();
while(iter.hasNext()){
    Object o = iter.next();
}

2) 方式二：使用增强for
for(Object o:col){

}

3) 方式三：使用普通for
for(int i=0;i<list.size();i++){
    Object object = list.get(i);
    System.out.println(object);
}
```

#### 4.1 ArrayList(基本等同于Vector,线程不安全/执行效率高,建议单线程使用)

##### 1. 用一个Object类型的数组

//transient Object[] elementData; transient表示短暂的,该属性不会被序列化

##### 2. 创建ArrayList对象时,无参构造器,数组初始容量为0,第一次添加扩容为10,再次扩容为1.5倍(只要有容量,均扩大为1.5倍),扩容用的是Arrays.copyOf()

##### 额外介绍:线程同步安全:

线程同步安全:在多线程环境中,当多个线程访问共享资源时,保证各个线程对共享资源的访问互不干扰,且最终的结果与预期一致

//当多个线程并发执行时,每个线程都能正确地获得资源的访问权,能正确地执行所需的操作,而不会因为资源冲突或其他线程的干扰而产生错误或不一致的结果



## 1. 互斥性 (Mutual Exclusion) :

- 确保在任何时刻，只有一个线程能够访问共享资源。这通常通过使用锁（如互斥锁，Mutex）来实现。

## 2. 可见性 (Visibility) :

- 确保一个线程对共享资源的修改能够被其他线程看到。在某些情况下，由于处理器缓存和内存屏障的原因，一个线程所做的更改可能不会立即对其他线程可见。

## 3. 原子性 (Atomicity) :

- 确保操作是不可分割的，即操作要么完全执行，要么完全不执行。这通常通过原子操作或事务来实现。

## 4. 顺序性 (Ordering) :

- 确保操作的顺序符合程序的逻辑。在多线程环境中，由于线程调度的不确定性，操作的顺序可能与代码中的顺序不同。

为了实现线程同步安全，可以采用以下机制：

- 锁 (Locks)**：使用锁来控制对共享资源的访问，确保在任何时候只有一个线程可以访问资源。
- 同步块 (Synchronized Blocks)**：在Java中，可以使用synchronized关键字来创建同步块，确保代码块在同一时间只能被一个线程执行。
- 同步方法 (Synchronized Methods)**：在Java中，可以将方法声明为synchronized，这样每次只有一个线程可以执行该方法。
- volatile关键字**：在Java中，volatile关键字用于确保变量的修改对所有线程立即可见，且在读取时不会从缓存中读取，而是直接从主内存中读取。
- 原子类 (Atomic Classes)**：如AtomicInteger等，提供了一组不需要使用锁的原子操作。
- 线程局部存储 (Thread Local Storage)**：使用ThreadLocal类为每个线程提供独立的变量副本，避免共享状态。

## 4.2 Vector(基本等同于ArrayList,需要线程同步安全时使用,但效率不高,安全跟效率不可兼得)

### 1. 基本介绍:

1. 用一个Object类型的数组
2. 线程同步(线程安全),其操作方法带有synchronized(确保任意时刻只有一个线程可以执行某个特定代码块)
3. 创建Vector对象时,无参构造器,数组初始容量为10,有参构造器可指定初始容量,再次扩容为2倍(只要有容量,均扩大为2倍),扩容用的是Arrays.copyOf()

### 2. 与ArrayList对比:

|           | 底层结构          | 版本     | 线程安全 (同步) 效率 | 扩容倍数                                                   |
|-----------|---------------|--------|--------------|--------------------------------------------------------|
| ArrayList | 可变数组          | jdk1.2 | 不安全, 效率高     | 如果有参构造1.5倍<br>如果是无参<br>1.第一次10<br>2.从第二次开始按1.5扩        |
| Vector    | 可变数组 Object[] | jdk1.0 | 安全, 效率不高     | 如果是无参, 默认10<br>, 满后, 就按2倍扩容<br><br>如果指定大小, 则每次直接按2倍扩容. |

## 4.3 LinkedList(线程不安全,没有实现线程同步安全,效率较高,双向链表和双向队列)

### 1. 底层操作:

1. 底层使用一个双向链表,有first首节点和last尾节点,可添加任意元素(包括null)
2. 每个节点(Node对象)有prev,next,item三属性(双向链表嘛)

### 2. 使用示例:

1. LinkedList linkedList = new LinkedList(); //此时first=last=null
2. remove(); //默认删除第一个节点,也可输入索引/要删除的对象(若重名,则优先删除索引对应的对象)
3. add,addAll,set,get(同List)
4. Iterator(迭代器遍历,同List)/for增强/for普通
  - i. 使用示例:

```
Iterator iterator = linkedList.iterator();
while (iterator.hasNext()) {
    Object next = iterator.next();
    System.out.println("next=" + next);
}
System.out.println("=LinkedList 遍历增强 for==");
```

```

for (Object o1 : linkedList) {
    System.out.println("o1=" + o1);
}
System.out.println("=LinkeList 遍历普通 for==");
for (int i = 0; i < linkedList.size(); i++) {
    System.out.println(linkedList.get(i));
}

```

5. addFirst,addLast,removeFirst,removeLast//字面意思

### 3. 与ArrayList对比(数组与链表各自的优势):

ArrayList:改查多时用

LinkedList:增删多时用

|            | 底层结构 | 增删的效率      | 改查的效率 |
|------------|------|------------|-------|
| ArrayList  | 可变数组 | 较低<br>数组扩容 | 较高    |
| LinkedList | 双向链表 | 较高，通过链表追加. | 较低    |

## 5 Set接口

### 1. 特点:

- 无序(添加与取出顺序不一致,不过取出有自己的固定顺序),无索引
- 因为无序,所以不能重复元素(更准确的说,同一个元素最多储存一个),最多只能有一个null(类比数学意义上的集合)

### 2. Set常用方法与遍历方法(是Collection子接口,当然也有Collection的方法)

- 常用方法:add等与Collection一致
- 遍历方法(没有索引,自然不能用索引遍历):

#### i. 迭代器:

```

Iterator iterator = setexample.iterator();
while (iterator.hasNext()) {
    Object obj = iterator.next();
    System.out.println("obj=" + obj);
}

```

#### ii. 增强for(普通for不行,因为没有索引):

```

for (Object o : setexample) {
    System.out.println("o=" + o);
}

```

## 5.1 HashSet

### 1. 基本介绍:

- 实现了set接口,而且HashSet实际上是HashMap

源码:

```
transient HashMap<E, Object> map;  
public HashSet(){  
    map = new HashMap<>();  
}
```

- 无序,不能重复元素(更准确的说,同一个元素最多储存一个),可以存放有且只有一个null(毕竟是Set的子接口嘛)

### 2. 基本方法实现(特别之处):

remove(Object o),add(Object o); //会返回boolean,以示是否添加/删除成功,当然,new的对象每一个都是不同的

### 3. (add函数)底层(HashMap--数组+链表+红黑树)

- 添加元素时,得到其hash值,转为索引值
- 在储存数据表table中,看该索引值是否已有存放元素,若没有则直接加入,若有,则调用**equals函数比较**,相同则放弃添加,不相同则添加到最后(以**链表方式添加**)

//String的**equals重写**(导致**equals**会**比较String的值而非地址**):

```
public final class String  
  
    public boolean equals(Object anObject) {  
        if (this == anObject) {  
            return true;  
        }  
        return (anObject instanceof String aString)  
            && (!COMPACT_STRINGS || this.coder == aString.coder)  
            && StringLatin1.equals(value, aString.value);  
    }
```

```
String a = new String("abc");  
String b = new String("abc");  
System.out.println(a.equals(b)); //T  
System.out.println(a==b); //F
```

于是导致(new的String只要值一样就判定为同一个):

```
System.out.println(list.add(new String( original: "lover")));
System.out.println(list.add(new String( original: "lover")));
```

```
true
false
```

3. Java8中,若一条链表元素数达到TREEIFY\_THRESHOLD(默认8),且table大小

$\geq \text{MIN\_TREEIFY\_CAPACITY}$ (默认64),则会树化(红黑树)

(但是Java9之后就没有这种"树化"机制了)

4. 重点的add底层:

2. 执行 add()

```
public boolean add(E e) { //e = "java"
    return map.put(e, PRESENT) == null; //static PRESENT = new Object();
}
```

3. 执行 put(), 该方法会执行 hash(key) 得到 key 对应的 hash 值 算法  $h = \text{key.hashCode()} \wedge (\text{h} \gg 16)$

```
public V put(K key, V value) { //key = "java" value = PRESENT 共享
    return putVal(hash(key), key, value, false, true);
}
```

#### 4. 执行putVal:

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {  
    Node<K,V>[] tab; Node<K,V> p; int n, i;  
    //定义了辅助变量  
    //table 就是 HashMap 的一个数组，类型是 Node[]  
    if ((tab = table) == null || (n = tab.length) == 0)  
        n = (tab = resize()).length;  
    //若table为null或大小为0，则进行第一次扩容，扩到16个空间  
    if ((p = tab[i = (n - 1) & hash]) == null)  
        //根据 key得到 hash,得到该key应存放到 table 表的哪个位置，并把这个位置的对象，赋给 p  
        //如果 p 为 null，表示还没有存放元素，创建一个 Node (key="520",value=PRESENT)  
        tab[i] = newNode(hash, key, value, null);  
    else {  
        Node<K,V> e; K k;  
        //辅助变量  
        if (p.hash == hash &&((k = p.key) == key || (key != null && key.equals(k))))  
            e = p;  
        //若当前索引位置对应的链表的第一个元素和准备添加的 key 的 hash 值一样  
        //并且满足 下面两个条件之一：  
        // (1) 准备加入的 key 和 p 指向的 Node 结点的 key 是同一个对象  
        // (2) p 指向的 Node 结点的 key 的 equals() 和准备加入的 key 比较后相同  
        //就不能加入  
        //其实就是节点p是否有相同的哈希值和键  
        else if (p instanceof TreeNode)  
            //判断p是不是一颗红黑树  
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  
            //如果是红黑树的话，调用红黑树的putTreeVal方法来添加/更新值  
        else {  
            //p不是红黑树  
            for (int binCount = 0; ; ++binCount) {  
                //遍历查找哈希值和键相同的节点  
                if ((e = p.next) == null) {  
                    //到达末尾，新建新节点并加到末尾，并检查是否达到了树化的阈值TREEIFY_THRESHOLD  
                    p.next = newNode(hash, key, value, null);  
                    if (binCount >= TREEIFY_THRESHOLD - 1)  
                        treeifyBin(tab, hash);  
                    //达到树化阈值，进行树化  
                    break;  
                }  
                if (e.hash == hash &&((k = e.key) == key || (key != null && key.equals(k))))  
                    //找到具有相同哈希值和键的节点，退出循环  
                    break;  
                p = e;//更新为下一节点，继续循环  
            }  
        }  
    }  
}
```

```

        }
    }

    if (e != null) {//找到了节点e(存在键),进行更新
        V oldValue = e.value;
        if (!onlyIfAbsent || oldValue == null)
            e.value = value;
        //若onlyIfAbsent为false或原值为null, 则更新该节点的值
        afterNodeAccess(e);
        return oldValue;
        //返回旧值,表示操作失败
    }
}

++modCount;//记录HashMap修改次数的计数器
if (++size > threshold)
    resize();//扩容
afterNodeInsertion(evict);//此处为空方法,留给子类实现操作的
return null;//表示插入操作完成,没有旧值需要返回
}

```

#### 4. 扩容机制:

1. 第一次添加时table数组扩容到16,临界值(threshold)是16\*加载因子(loadFactor),即临界值为12
2. table到临界值就会2倍扩容到32,并产生新的临界值 $32 \times 0.75 = 24$ ,以此类推
3. Java8中,若一条链表元素数达到TREEIFY\_THRESHOLD(默认8),且table大小  
 $\geq \text{MIN\_TREEIFY\_CAPACITY}$ (默认64),则会树化(红黑树),否则依旧为**数组扩容机制**  
(但是Java9之后就没有这种"树化"机制了)

#### 5.1 特别案例(add(类的对象)所需修改)

1. equals(一般与hashcode一起重写,确保同一对象有相同的哈希码):

```
//前缀必须为public boolean equals(Object o),因为原方法就是这样的,**重写需要方法名与参数列表都相
public boolean equals(Object o) {
    if(this == o) {
        //若对象地址一模一样,返回true
        return true;
    }
    if(o == null || getClass()!=o.getClass())
        //若o为null或者类型不一样,返回false
        return false;
    Employee employee = (Employee) o;
    //将其同一类型化
    return this.name.equals(employee.getName()) && this.age == employee.getAge();
    //重写判断条件,因为add判断也用equals函数,所以需要修改equals
}
```

## 2. hashCode(一般与equals一起重写,确保同一对象有相同的哈希码):

```
public int hashCode() {
    return Objects.hash(name,age);
}
```

## 3. toString:

```
public String toString() {
    return "Employee [name=" + name + ", age=" + age + "]";
}
```

重写前println(hashset):

```
[Test.Employee@165f3d4, Test.Employee@165f3e6, Test.Employee@168edc0]
```

//会输出地址

重写后println(hashset):

```
[Employee [name=小明, age=18], Employee [name=小明, age=36], Employee [name=小
红, age=18]]
```

//会输出自定义模板

## 5.2 特别案例之remove(哈希位置不变但是储存变了)

若重写了hashCode和equals方法,则会导致哈希值计算出错,在HashSet中remove的时候remove不掉原  
先的对象

```
class Lover{
    public String name;
    public Lover(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
    public String toString(){
        return name;
    }

    @Override
    public boolean equals(Object o){
        if(o instanceof Lover){
            return name.equals(((Lover)o).name);
        }
        return false;
    }
    public int hashCode(){
        return name.hashCode();
    }
}

...
Lover l1 = new Lover("hxq");
Lover l2 = new Lover("xjh");
HashSet set = new HashSet();
set.add(l1);
set.add(l2);
System.out.println(set);
l1.name="胡欣琦";
System.out.println(set);
set.remove(l1);
System.out.println(set);
```

```
[hxq, xjh]
[胡欣琦, xjh]
[胡欣琦, xjh]
```

## 5.2 LinkedHashSet(HashSet的子类,数组+双向链表)

1. 是HashSet的子类,底层是LinkedHashMap(数组+双向链表)
2. 根据hashCode值决定元素储存位置,又以链表维护元素次序(使得元素看起来是以插入顺序保存的)  
//也就是先求hash值确定元素在table的位置,再将元素加入到双向链表中(添加原则与HashSet一致),所以可以保证插入顺序与遍历顺序一致
3. 当然,作为HashSet的子类,Set的子类的子类,LinkedHashSet也不允许重复元素

### 说明

- 1) 在LinkedHastSet 中维护了一个hash表和双向链表  
( LinkedHastSet 有 head 和 tail )
- 2) 每一个节点有 before 和 after 属性, 这样可以形成双向链表
- 3) 在添加一个元素时, 先求hash值, 在求索引, 确定该元素在 table 的位置, 然后将添加的元素加入到双向链表(如果已经存在, 不添加[原则和hashset一样])  

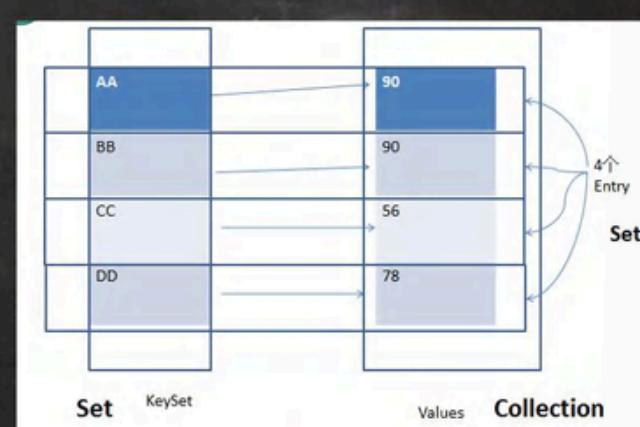
```
tail.next = newElement // 示意代码  
newElement.pre = tail  
tail = newEelment;
```
- 4) 这样的话, 我们遍历LinkedHastSet 也能确保插入顺序和遍历顺序一致

## 2. Map接口(以下均为JDK8的Map接口特点):

### 1. Map特点(以JDK8的Map接口特点):

1. Map和Collection并列存在,保存有映射关系的数据(Key-Value一一对应)
2. key和value可以是任何引用类型的数据,会封装到HashMap\$Node对象中,常用String类作为key(key不能重复,value可以)
3. key和value可以是null(但是key有唯一性,所以就算是null的key也只能有一个)
4. key-value示意图:

8) Map存放数据的key-value示意图, 一对 k-v 是放在一个HashMap\$Node中的, 有因为Node 实现了 Entry 接口, 有些书上也说 一对k-v就是一个Entry(如图) [代码演示]



## 2. 常用方法:

### 1. put:增加元素/修改元素值

```
Map map = new HashMap();
map.put("myLover", "hxq");
map.put("myLover", "胡欣琦");//改变myLover对应的value
map.put("111", 1);
map.put("222", 2);
map.put("333", 3);
map.put("444", 4);
//{myLover=胡欣琦, 111=1, 222=2, 333=3, 444=4}
```

### 2. remove:删除映射关系

```
map.remove("444");//输入key删除
//{myLover=胡欣琦, 111=1, 222=2, 333=3}
```

### 3. get:根据键获取值

```
Object lover = map.get("myLover");
System.out.println("lover=" + lover);
//lover=胡欣琦
```

### 4. size:获取元素个数

```
System.out.println(map.size());//4
```

### 5. isEmpty:判断个数是否为0

```
System.out.println(map.isEmpty());//false
```

### 6. clear:清除key和value

```
//map.clear();
//System.out.println("map=" + map);//map={}
```

### 7. containsKey:查找key是否存在

```
System.out.println(map.containsKey("myLover"));//true
```

### 3. 遍历方法(与List和Set基本原理一样):

- 先用Set取出所有的key,再通过value取出对应value:

```

//增强for:
Set keyset = map.keySet();
for (Object key : keyset) {
    System.out.println(key + "-" + map.get(key));
}
//迭代器:
Iterator iterator = keyset.iterator();
while (iterator.hasNext()) {
    Object key = iterator.next();
    System.out.println(key + "-" + map.get(key));
}

```

## 2. 直接取出所有value:

```

Collection values = map.values();
//增强for:
for (Object value : values) {
    System.out.println(value);
}
//迭代器:
Iterator iterator = values.iterator();
while (iterator.hasNext()) {
    Object value = iterator.next();
    System.out.println(value);
}

```

## 3. 通过EntrySet获取key和value:

```

Set entrySet = map.entrySet(); // EntrySet<Map.Entry<K,V>>
//增强for:
for (Object entry : entrySet) {
    //将 entry 转成 Map.Entry
    Map.Entry m = (Map.Entry) entry;
    System.out.println(m.getKey() + "-" + m.getValue());
}
//迭代器:
Iterator iterator = entrySet.iterator();
while (iterator.hasNext()) {
    Object entry = iterator.next();
    Map.Entry m = (Map.Entry) entry;
    System.out.println(m.getKey() + "-" + m.getValue());
}

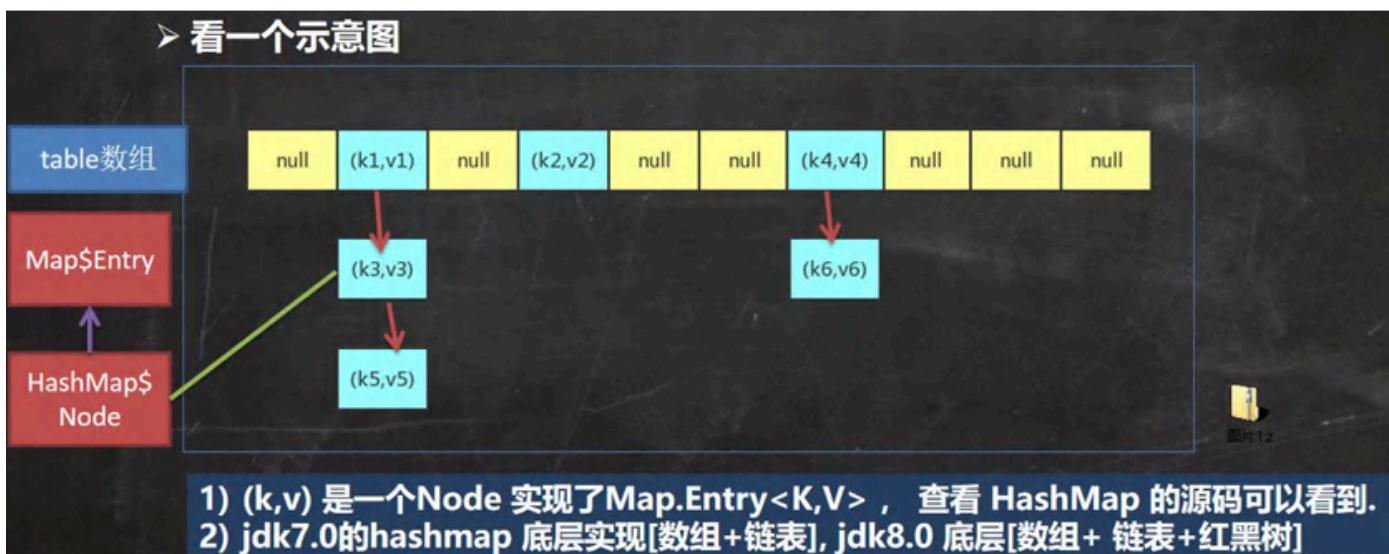
```

## 4.1 HashMap(线程不安全,效率高):

### 1. 小结:

- 1) Map接口的常用实现类: **HashMap**、**Hashtable**和**Properties**。
- 2) **HashMap**是 Map 接口使用频率最高的实现类。
- 3) **HashMap** 是以 key-val 对的方式来存储数据(**HashMap\$Node**类型) [案例 Entry ]
- 4) key 不能重复, 但是值可以重复, 允许使用null键和null值。
- 5) 如果添加相同的key , 则会覆盖原来的key-val , 等同于修改.(key不会替换, val会替)
- 6) 与HashSet一样, 不保证映射的顺序, 因为底层是以hash表的方式来存储的. (jdk8 hashMap 底层 数组+链表+红黑树)
- 7) **HashMap**没有实现同步, 因此是线程不安全的, 方法没有做同步互斥的操作, 没有 synchronized

### 2. 机制及源码:



### 4.1.3.3 HashMap 底层机制及源码剖析

**HashMapSource.java 先说结论-》 debug源码.**

➤ 扩容机制 [和HashSet相同]

- 1) **HashMap**底层维护了**Node**类型的数组**table**, 默认为null
- 2) 当创建对象时, 将加载因子(loadfactor)初始化为0.75.
- 3) 当添加key-val时, 通过**key**的哈希值得到在**table**的索引。然后判断该索引处是否有元素, 如果没有元素直接添加。如果该索引处有元素, 继续判断该元素的**key**和准备加入的**key**相是否等, 如果相等, 则直接替换**val**; 如果不相等需要判断是树结构还是链表结构, 做出相应处理。如果添加时发现容量不够, 则需要扩容。
- 4) 第1次添加, 则需要扩容**table**容量为16, 临界值(threshold)为12 ( $16 * 0.75$ )
- 5) 以后再扩容, 则需要扩容**table**容量为原来的2倍(32), 临界值为原来的2倍, 即24, 依次类推.
- 6) 在Java8中, 如果一条链表的元素个数超过 **TREEIFY\_THRESHOLD**(默认是 8 ), 并且 **table**的大小  $\geq \text{MIN\_TREEIFY\_CAPACITY}(默认64), 就会进行树化(红黑树)$

## 4.2 Hashtable(线程安全.效率较低):

- 除了不让null键null值(不能有null)以外,与HashMap语法一样
- 对比:

|           | 版本  | 线程安全 (同步) | 效率 | 允许null键null值 |
|-----------|-----|-----------|----|--------------|
| HashMap   | 1.2 | 不安全       | 高  | 可以           |
| Hashtable | 1.0 | 安全        | 较低 | 不可以          |

## 4.3 Properties(继承Hashtable类):

- 基本介绍:
  - 继承Hashtable类,键对值,与Hashtable使用类似(无null)
  - 可以从xx.properties文件加载数据到该类对象来读取和修改
  - xx.properties文件通常作为配置文件

## 3.1 如何选择实现类:

1) 先判断存储的类型 (一组对象[单列]或一组键值对[双列])

2) 一组对象[单列]: Collection接口

    允许重复: List

        增删多: LinkedList [底层维护了一个双向链表]

        改查多: ArrayList [底层维护 Object类型的可变数组]

    不允许重复: Set

        无序: HashSet [底层是HashMap, 维护了一个哈希表 即(数组+链表+红黑树)]

        排序: TreeSet [老韩举例说明]

        插入和取出顺序一致: LinkedHashSet , 维护数组+双向链表

3) 一组键值对[双列]: Map

    键无序: HashMap [底层是: 哈希表 jdk7: 数组+链表, jdk8: 数组+链表+红黑树]

    键排序: TreeMap [老韩举例说明]

    键插入和取出顺序一致: LinkedHashMap

    读取文件 Properties

关于TreeSet排序:

```
TreeSet treeSet = new TreeSet();
TreeSet treeSet = new TreeSet(new Comparator() {
    @Override
    public int compare(Object o1, Object o2) {
        //若要求加入的元素, 按照长度大小排序
        return ((String) o1).length() - ((String) o2).length();
        //也可调用String的 compareTo方法进行字符串大小比较
    }
})
```

关于TreeMap排序(键排序):

```
TreeMap treeMap = new TreeMap();
TreeMap treeMap = new TreeMap(new Comparator() {
    @Override
    public int compare(Object o1, Object o2) {
        //按照传入的 key(String) 的大小进行排序
        //return ((String) o2).compareTo((String) o1);
        //按照key(String) 的长度大小排序
        return ((String) o2).length() - ((String) o1).length();
    }
});
```

## 3.2 关于HashSet与TreeSet的去重区别

### 1. HashSet:

hashCode() + equals(), 先看索引, 再看equals遍历比较

### 2. TreeSet(底层基于红黑树实现)

看Comparator()可自定义来比较, 具体比较规范为Comparable接口的compareTo(默认会将键Key类型转为Comparable的实现类, 从而使用Key类型的compareTo)(所以要求元素必须实现Comparable接口!) 所以未实现Comparable接口的对象就add不了(也就是说类什么的,int什么的不行, String可以) 自定义写法:

```
class Lover implements Comparable{
    ...
    @Override
    public int compareTo(...){
        ...
    }
}
```

//String的compareTo: 以字节数组的形式, 先看编码器, 再比较字节数组(能比较不同编码)

## 3. Collections工具类(操作集合的工具类)

### 1. 介绍:

1. 是一个操作Set, List, Map等集合的工具类
2. 提供了一系列静态方法对集合元素进行排序, 查询, 修改

## 2. 排序,查找,替换(static):

### 1. reverse(List):反转List中元素顺序

```
//import java.util.Collections;导入工具类  
List list = new ArrayList();  
list.add(0,"hxq");  
list.add(1,"大荒");  
list.add(2,"大江");  
list.add(3,"小灯");  
//[hxq, 大荒, 大江, 小灯]  
Collections.reverse(list);  
//必须要有整个Collections.静态方法  
//[小灯, 大江, 大荒, hxq]
```

### 2. shuffle(List):对List集合元素随机排序

```
//Collections.shuffle(list);
```

### 3. sort(List):根据元素自然顺序对其升序排列

```
Collections.sort(list);  
//[hxq, 大江, 大荒, 小灯]
```

### 4. sort(List,Comparator):根据指定Comparator顺序对其排序(max,min方法同理)

```
//import java.util.Comparator;  
Collections.sort(list, new Comparator() {  
    @Override  
    public int compare(Object o1, Object o2) {  
        //可以加入校验代码.  
        return ((String) o2).length()- ((String) o1).length();  
    }  
    //按字符串长度排序  
});  
//[hxq, 小灯, 大江, 大荒]
```

### 5. swap(List, int,int):将List集合索引i与j元素互换

```
Collections.swap(list,2,3);  
System.out.println("list=" + list);  
//[hxq, 小灯, 大荒, 大江]
```

## 6. frequency(Collection, Object):返回集合中指定元素出现次数

```
//int i = Collections.frequency(list, "hxq");
```

## 7. copy(list2, list):把list复制到list2

```
ArrayList list2 = new ArrayList();
```

## 8. replaceAll(List, oldValue, newValue):用newValue替换List所有旧值

```
Collections.replaceAll(list, "小灯", "老灯");
```

```
//[hxq, 老灯, 大荒, 大江]
```

# 3.5 泛型(参数化类型)

## 1. 特点:

1. 编译时, 检查添加元素的类型, 并且不提示编译警告(ClassCastException), 提高安全性
2. 减少类型转换的次数(否则, 以下面为例, 要转为Object储存作媒介), 提高效率
3. 可在类声明时, 通过一个标识表示类中属性/返回值/参数的类型
4. 泛型类的特别: 在类定义时有几个<泛型参数>, 创建对象就必须提供相同数量的<泛型参数>(要么不用泛型, 要么用同样数量的泛型)
5. 示例:

```

Lover<String> l1 = new Lover<String>("hxq");
Lover<String> l2 = new Lover<String>("xjh");
ArrayList<Lover> Lovers = new ArrayList<Lover>();
Lovers.add(l1);
Lovers.add(l2);
System.out.println(Lovers);
for(Lover lover :Lovers){
    System.out.println(lover.getName());
}
...
class Lover<S>{
    public String name;
    public Lover(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
    public String toString(){
        return name;
    }
}

```

## 2. 泛型的使用(接口,类等)

### 1. 声明(任何字母都行,常用T,即type):

1. 接口:interface 接口名<T...>{...}
2. 类:class 类名<K,V...>{...}

### 2. 实例化(对象,遍历器也得加泛型):

1. 对象>List lovers = new ArrayList(...);
2. 遍历器:Iterator iterator = Lover.iterator();

### 3. 注意事项:

1. 泛型只能是引用类型,不能是基本数据类型(但是可以用Integer等类)
2. 给泛型指定具体类型后,可传入该类型及子类型:

```
Pig<A> aPig = new Pig<A>(new A());
```

```
aPig.f();
```

```
Pig<A> aPig2 = new Pig<A>(new B());
```

```
aPig2.f();
```

### 3. 关于泛型对象的使用示例:

- i. List Lovers = new ArrayList<>();//默认为Lover
- ii. Lover l1 = new Lover("hxq");//默认为Object
- iii. Lover l1 = new Lover<>("hxq");//默认为Object
- iv. Lover l1 = new Lover<>("hxq");//默认为Lover
- v. Lover l1 = new Lover("hxq");

vi. 关于泛型对象的使用小结:没有指定类型时,默认Object,后面的<>内的泛型参数可以省略,默认与前面一致(必须一致)(相当于对对碰//?)

## 4. 自定义泛型类:

- a. 声明: class 类名<T,V...>{...}
- b. 在类里面,属性/方法/返回值都能用泛型
- c. 泛型类的类型在创建对象时确定(创建对象时需要指定类型)
- d. 静态方法不能用泛型(都不用导入还怎么用嘛)
- e. 使用泛型的数组不能初始化

## 5. 自定义泛型接口:

- a. 声明: interface 接口名<T,V...>{}
- b. 静态成员同样不能用泛型(都不用导入还怎么用嘛)
- c. 接口的类型在继承接口或实现接口时确定,默认Object

## 6. 自定义泛型方法:

- a. 声明:(访问+static+final)修饰符 <T,V...> 返回类型 方法名(参数列表){...}  
//<T,V...>跟泛型类一样,是定义泛型参数(**在类的基础上额外添加新的泛型**)  
//若类已有所需泛型,则可省略<...>
- b. 可以用在**普通类/泛型类**中
- c. 泛型方法**类型在调用时确定**(而非创建对象时)

## 7. 关于继承和通配符(就是这个 ?, 可以方便使用泛型, 而不用一个一个列出来):

- a. 泛型不具备继承性
- b. 支持任意泛型类型
- c. 支持A及其所有子类
- d. 支持A及所有父类

## 3.6 JUnit(单元测试类, 需在Maven的pom.xml依赖项里面配置)

@Test:标记测试方法

@BeforeClass:在所有测试方法执行前执行一次, 通常用于静态资源的初始化

@AfterClass:在所有测试方法执行后执行一次, 通常用于清理静态资源

@Before:在每个测试方法执行前执行, 用于设置测试环境

@After:在每个测试方法执行后执行, 用于清理测试环境

@Ignore:标记一个测试方法或测试类, 使其在当前测试执行中被忽略

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class Calculator {

    @Test
    public void testAddition() {
        // 测试加法方法
        //当然,一般会把一个类的所有测试方法放在另一个类方便管理
        assertEquals("2 + 3 应该等于 5", 5, Calculator.add(2, 3));
    }

    public static int add(int a, int b) {
        return a + b;
    }
}
```

## 3.7 Assert(断言)

测试方法通常包含断言, 用来检查代码是否按预期工作

- a. assertEquals(expected, actual): 用于检查两个值是否相等
- b. assertNotEquals(expected, actual): 检查两个值是否不相等
- c. assertTrue(condition): 检查条件是否为true
- d. assertFalse(condition): 检查条件是否为false
- e. assertNull(object): 检查对象是否为null
- f. assertNotNull(object): 检查对象是否不为null
- g. assertThrows(type, executable): 检查可执行代码块是否抛出了指定类型的异常

## 3.8 画图(暂时没看,JFrame画板/事件处理)

## 3.9. 多线程

### 特殊章节：八股：

- a. 多线程：

#### 保证数据的一致性有哪些方案呢？

- **事务管理**: 使用数据库事务来确保一组数据库操作要么全部成功提交，要么全部失败回滚。通过ACID（原子性、一致性、隔离性、持久性）属性，数据库事务可以保证数据的一致性。
- **锁机制**: 使用锁来实现对共享资源的互斥访问。在 Java 中，可以使用 synchronized 关键字、ReentrantLock 或其他锁机制来控制并发访问，从而避免并发操作导致数据不一致。
- **版本控制**: 通过乐观锁的方式，在更新数据时记录数据的版本信息，从而避免同时对同一数据进行修改，进而保证数据的一致性。

## 线程的创建方式有哪些?

### 1.继承Thread类

这是最直接的一种方式，用户自定义类继承java.lang.Thread类，重写其run()方法，run()方法中定义了线程执行的具体任务。创建该类的实例后，通过调用start()方法启动线程。

```
class MyThread extends Thread {  
    @Override  
    public void run() {  
        // 线程执行的代码  
    }  
  
    public static void main(String[] args) {  
        MyThread t = new MyThread();  
        t.start();  
    }  
}
```

java

### 采用继承Thread类方式

- 优点：编写简单，如果需要访问当前线程，无需使用Thread.currentThread()方法，直接使用this，即可获得当前线程
- 缺点：因为线程类已经继承了Thread类，所以不能再继承其他的父类

### 2.实现Runnable接口

如果一个类已经继承了其他类，就不能再继承Thread类，此时可以实现java.lang.Runnable接口。实现Runnable接口需要重写run()方法，然后将此Runnable对象作为参数传递给Thread类的构造器，创建Thread对象后调用其start()方法启动线程。

```
class MyRunnable implements Runnable {  
    @Override  
    public void run() {  
        // 线程执行的代码  
    }  
  
    public static void main(String[] args) {  
        Thread t = new Thread(new MyRunnable());  
        t.start();  
    }  
}
```

java

采用实现Runnable接口方式：

- 优点：线程类只是实现了Runnable接口，还可以继承其他的类。在这种方式下，可以多个线程共享同一个目标对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。
- 缺点：编程稍微复杂，如果需要访问当前线程，必须使用Thread.currentThread()方法。

### 3. 实现Callable接口与FutureTask

java.util.concurrent.Callable接口类似于Runnable，但Callable的call()方法可以有返回值并且可以抛出异常。要执行Callable任务，需将它包装进一个FutureTask，因为Thread类的构造器只接受Runnable参数，而FutureTask实现了Runnable接口。

```
java
class MyCallable implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        // 线程执行的代码，这里返回一个整型结果
        return 1;
    }
}

public static void main(String[] args) {
    MyCallable task = new MyCallable();
    FutureTask<Integer> futureTask = new FutureTask<>(task);
    Thread t = new Thread(futureTask);
    t.start();
}
```

```
try {
    Integer result = futureTask.get(); // 获取线程执行结果
    System.out.println("Result: " + result);
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
}
```

采用实现Callable接口方式：

- 缺点：编程稍微复杂，如果需要访问当前线程，必须调用Thread.currentThread()方法。
- 优点：线程只是实现Runnable或实现Callable接口，还可以继承其他类。这种方式下，多个线程可以共享一个target对象，非常适合多线程处理同一份资源的情形。

## 4. 使用线程池 (Executor框架)

从Java 5开始引入的java.util.concurrent.ExecutorService和相关类提供了线程池的支持，这是一种更高效的线程管理方式，避免了频繁创建和销毁线程的开销。可以通过Executors类的静态方法创建不同类型的线程池。

```
class Task implements Runnable {  
    @Override  
    public void run() {  
        // 线程执行的代码  
    }  
}  
  
public static void main(String[] args) {  
    ExecutorService executor = Executors.newFixedThreadPool(10); // 创建固定大小的线程池  
    for (int i = 0; i < 100; i++) {  
        executor.submit(new Task()); // 提交任务到线程池执行  
    }  
    executor.shutdown(); // 关闭线程池  
}
```

采用线程池方式：

- 缺点：线程池增加了程序的复杂度，特别是当涉及线程池参数调整和故障排查时。错误的配置可能导致死锁、资源耗尽等问题，这些问题的诊断和修复可能较为复杂。
- 优点：线程池可以重用预先创建的线程，避免了线程创建和销毁的开销，显著提高了程序的性能。对于需要快速响应的并发请求，线程池可以迅速提供线程来处理任务，减少等待时间。并且，线程池能够有效控制运行的线程数量，防止因创建过多线程导致的系统资源耗尽（如内存溢出）。通过合理配置线程池大小，可以最大化CPU利用率和系统吞吐量。

## 怎么启动线程？

启动线程的通过Thread类的[start\(\)](#)。

```
// 创建两个线程，用start启动线程  
MyThread myThread1 = new MyThread();  
MyThread myThread2 = new MyThread();  
myThread1.start();  
myThread2.start();
```

## 如何停止一个线程的运行?

主要有这些方法：

- **异常法停止**: 线程调用interrupt()方法后，在线程的run方法中判断当前对象的interrupted()状态，如果是中断状态则抛出异常，达到中断线程的效果。
- **在沉睡中停止**: 先将线程sleep，然后调用interrupt标记中断状态，interrupt会将阻塞状态的线程中断。会抛出中断异常，达到停止线程的效果
- **stop()暴力停止**: 线程调用stop()方法会被暴力停止，方法已弃用，该方法会有不好的后果：强制让线程停止有可能使一些清理性的工作得不到完成。
- **使用return停止线程**: 调用interrupt标记为中断状态后，在run方法中判断当前线程状态，如果为中断状态则return，能达到停止线程的效果。

## 调用 interrupt 是如何让线程抛出异常的?

每个线程都有一个与之关联的布尔属性来表示其中断状态，中断状态的初始值为false，当一个线程被其它线程调用 `Thread.interrupt()` 方法中断时，会根据实际情况做出响应。

- 如果该线程正在执行低级别的可中断方法（如 `Thread.sleep()`、`Thread.join()` 或 `Object.wait()`），则会解除阻塞并**抛出 `InterruptedException` 异常**。
- 否则 `Thread.interrupt()` 仅设置线程的中断状态，在该被中断的线程中稍后可通过轮询中断状态来决定是否要停止当前正在执行的任务。

## Java线程的状态有哪些？

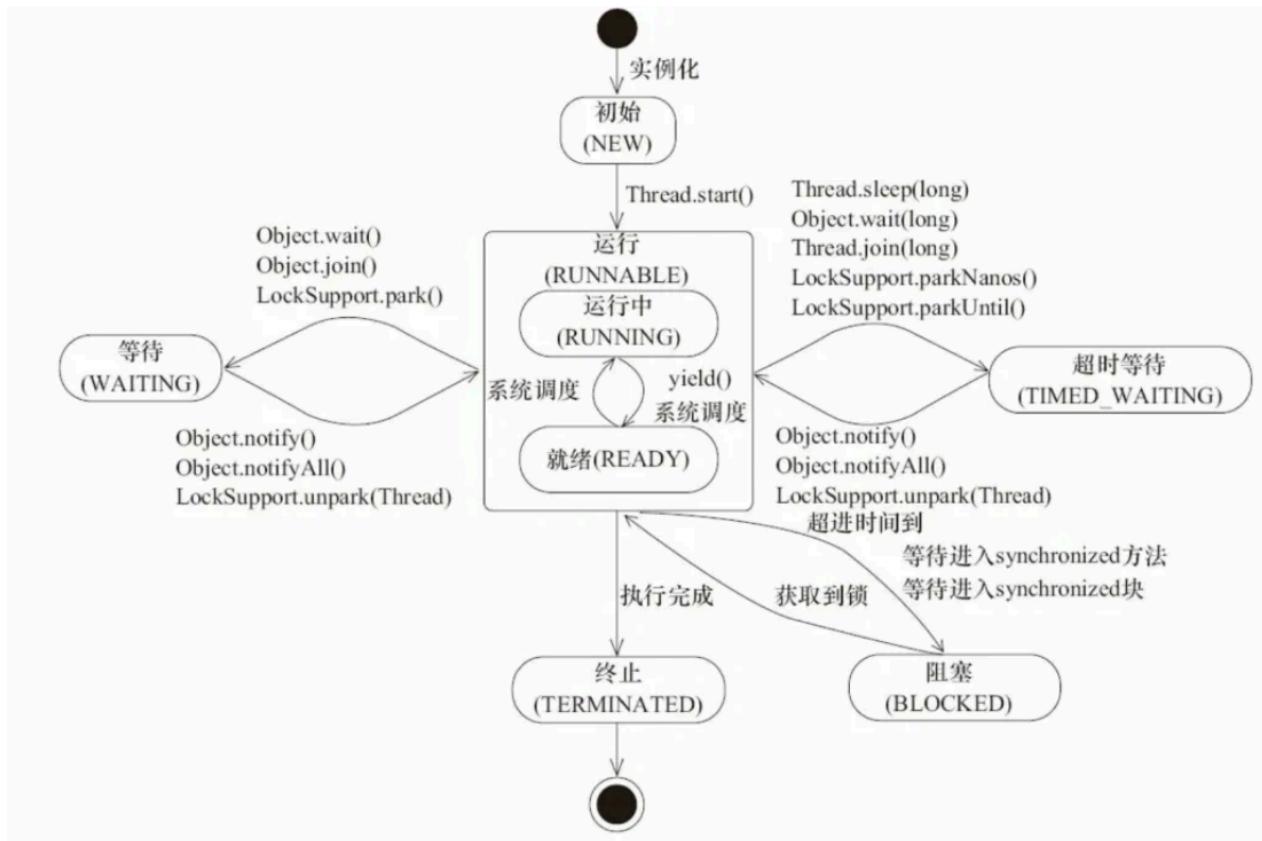


图4-1 Java线程状态变迁

源自《Java并发编程艺术》`java.lang.Thread.State`枚举类中定义了六种线程的状态，可以调用线程`Thread.getState()`方法[获取当前线程的状态](#)。

| 线程状态          | 解释                                                        |
|---------------|-----------------------------------------------------------|
| NEW           | 尚未启动的线程状态，即线程创建， <a href="#">还未调用start方法</a>              |
| RUNNABLE      | <a href="#">就绪状态</a> （调用start，等待调度）+ <a href="#">正在运行</a> |
| BLOCKED       | <a href="#">等待监视器锁</a> 时，陷入阻塞状态                           |
| WAITING       | 等待状态的线程正在 <a href="#">等待</a> 另一线程执行特定的操作（如notify）         |
| TIMED_WAITING | 具有 <a href="#">指定等待时间</a> 的等待状态                           |
| TERMINATED    | 线程完成执行， <a href="#">终止状态</a>                              |

# sleep 和 wait 的区别是什么？

对比例表：

| 特性   | sleep()        | wait()                       |
|------|----------------|------------------------------|
| 所属类  | Thread 类（静态方法） | Object 类（实例方法）               |
| 锁释放  | ✗              | ✓                            |
| 使用前提 | 任意位置调用         | 必须在同步块内（持有锁）                 |
| 唤醒机制 | 超时自动恢复         | 需 notify() / notifyAll() 或超时 |
| 设计用途 | 暂停线程执行，不涉及锁协作  | 线程间协调，释放锁让其他线程工作             |

- 所属分类的不同：** sleep 是 Thread 类的静态方法，可以在任何地方直接通过 Thread.sleep() 调用，无需依赖对象实例。wait 是 object 类的实例方法，这意味着必须通过对象实例来调用。
- 锁释放的情况：** Thread.sleep() 在调用时，线程会暂停执行指定的时间，但不会释放持有的对象锁。也就是说，在 sleep 期间，其他线程无法获得该线程持有的锁。Object.wait()：调用该方法时，线程会释放持有的对象锁，进入等待状态，直到其他线程调用相同对象的 notify() 或 notifyAll() 方法唤醒它
- 使用条件：** sleep 可在任意位置调用，无需事先获取锁。wait 必须在同步块或同步方法内调用（即线程需持有该对象的锁），否则抛出 IllegalMonitorStateException。
- 唤醒机制：** sleep 休眠时间结束后，线程自动恢复到就绪状态，等待CPU调度。wait 需要其他线程调用相同对象的 notify() 或 notifyAll() 方法才能被唤醒。notify() 会随机唤醒一个在该对象上等待的线程，而 notifyAll() 会唤醒所有在该对象上等待的线程。

一个是线程，一个是对象

## sleep会释放cpu吗？

是的，调用 `Thread.sleep()` 时，线程会释放 CPU，但不会释放持有的锁。

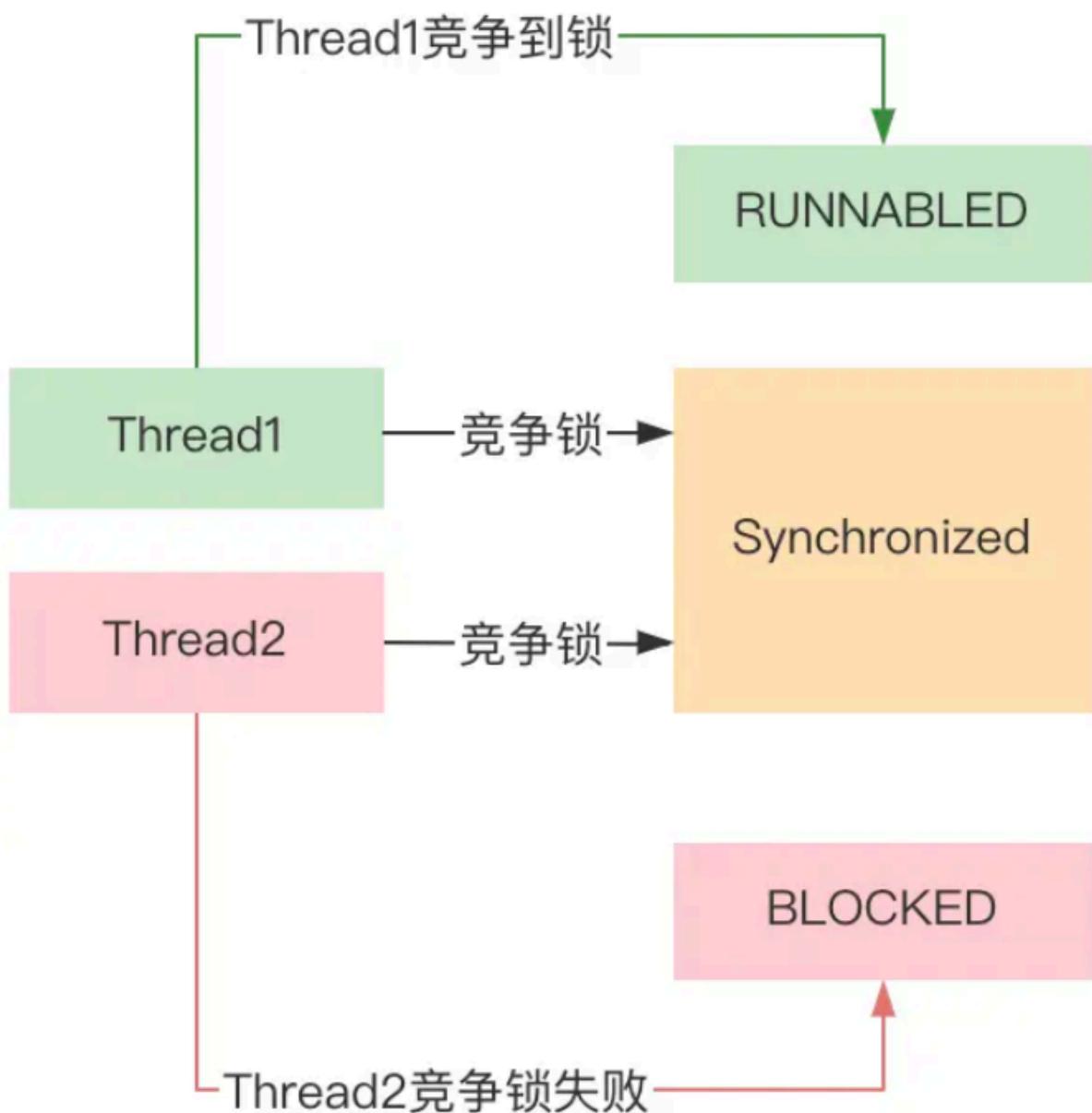
当线程调用 `sleep()` 后，会主动让出 CPU 时间片，进入 `TIMED_WAITING` 状态。此时操作系统会触发调度，将 CPU 分配给其他处于就绪状态的线程。这样其他线程（无论是需要同一锁的线程还是不相关线程）便有机会执行。

`sleep()` 不会释放线程已持有的任何锁（如 `synchronized` 同步代码块或方法中获取的锁）。因此，如果有其他线程试图获取同一把锁，它们仍会被阻塞，直到原线程退出同步代码块。

## blocked和waiting有啥区别

区别如下：

- **触发条件**:线程进入BLOCKED状态通常是因为试图获取一个对象的锁（monitor lock），但该锁已经被另一个线程持有。这通常发生在尝试进入synchronized块或方法时，如果锁已被占用，则线程将被阻塞直到锁可用。线程进入WAITING状态是因为它正在等待另一个线程执行某些操作，例如调用Object.wait()方法、Thread.join()方法或LockSupport.park()方法。在这种状态下，线程将不会消耗CPU资源，并且不会参与锁的竞争。



- **唤醒机制:**当一个线程被阻塞等待锁时，一旦锁被释放，线程将有机会重新尝试获取锁。如果锁此时未被其他线程获取，那么线程可以从BLOCKED状态变为RUNNABLE状态。线程在WAITING状态中需要被显式唤醒。例如，如果线程调用了Object.wait()，那么它必须等待另一个线程调用同一对象上的Object.notify()或Object.notifyAll()方法才能被唤醒。

所以，BLOCKED和WAITING两个状态最大的区别有两个：

- BLOCKED是锁竞争失败后被被动触发的状态，WAITING是人为的主动触发的状态
- BLOCKED的唤醒时自动触发的，而WAITING状态是必须要通过特定的方法来主动唤醒

## wait 状态下的线程如何进行恢复到 running 状态？

线程从 等待 (WAIT) 状态恢复到 运行 (RUNNING) 状态的核心机制是 **通过外部事件触发或资源可用性变化，比如等待的线程被其他线程对象唤醒，`notify()` 和 `notifyAll()`。**

```
java
synchronized (lock) {
    // 线程进入等待状态，释放锁
    lock.wait();
}

// 其他线程调用以下代码唤醒等待线程
synchronized (lock) {
    lock.notify();      // 唤醒单个线程
    // lock.notifyAll(); // 唤醒所有等待线程
}
```

## notify 和 notifyAll 的区别?

同样是唤醒等待的线程，同样最多只有一个线程能获得锁，同样不能控制哪个线程获得锁。

区别在于：

- notify：唤醒一个线程，其他线程依然处于wait的等待唤醒状态，如果被唤醒的线程结束时没调用 notify，其他线程就永远没人去唤醒，只能等待超时，或者被中断
- notifyAll：所有线程退出wait的状态，开始竞争锁，但只有一个线程能抢到，这个线程执行完后，其他线程又会有一个幸运儿脱颖而出得到锁

## notify 选择哪个线程?

notify在源码的注释中说到notify选择唤醒的线程是任意的，但是依赖于具体实现的jvm。

```
/**  
 * Wakes up a single thread that is waiting on this object's  
 * monitor. If any threads are waiting on this object, one of them  
 * is chosen to be awakened. The choice is arbitrary and occurs at  
 * the discretion of the implementation. A thread waits on an object's  
 * monitor by calling one of the {@code wait} methods.  
 * <p>
```

JVM有很多实现，比较流行的就是hotspot，hotspot对notofy()的实现并不是我们认为的随机唤醒，而是“先进先出”的顺序唤醒。

## 不同的线程之间如何通信？

共享变量是最基本的线程间通信方式。多个线程可以访问和修改同一个共享变量，从而实现信息的传递。为了保证线程安全，通常需要使用 `synchronized` 关键字或 `volatile` 关键字。

```
java
class SharedVariableExample {
    // 使用 volatile 关键字保证变量的可见性
    private static volatile boolean flag = false;

    public static void main(String[] args) {
        // 生产者线程
        Thread producer = new Thread(() -> {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 修改共享变量
            flag = true;
            System.out.println("Producer: Flag is set to true.");
        });

        // 消费者线程
        Thread consumer = new Thread(() -> {
            while (!flag) {
                // 等待共享变量被修改
            }
            System.out.println("Consumer: Flag is now true.");
        });

        producer.start();
        consumer.start();
    }
}
```

## 代码解释

- `volatile` 关键字确保了 `flag` 变量在多个线程之间的可见性，即一个线程修改了 `flag` 的值，其他线程能立即看到。
- 生产者线程在睡眠 2 秒后将 `flag` 设置为 `true`，消费者线程在 `flag` 为 `false` 时一直等待，直到 `flag` 变为 `true` 才继续执行。

`Object` 类中的 `wait()`、`notify()` 和 `notifyAll()` 方法可以用于线程间的协作。`wait()` 方法使当前线程进入等待状态，`notify()` 方法唤醒在此对象监视器上等待的单个线程，`notifyAll()` 方法唤醒在此对象监视器上等待的所有线程。

```
java
class WaitNotifyExample {
    private static final Object lock = new Object();

    public static void main(String[] args) {
        // 生产者线程
        Thread producer = new Thread(() -> {
            synchronized (lock) {
                try {
                    System.out.println("Producer: Producing...");
                    Thread.sleep(2000);
                    System.out.println("Producer: Production finished. Notifying consumer.");
                    // 唤醒等待的线程
                    lock.notify();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

```
// 消费者线程
Thread consumer = new Thread(() -> {
    synchronized (lock) {
        try {
            System.out.println("Consumer: Waiting for production to finish.");
            // 进入等待状态
            lock.wait();
            System.out.println("Consumer: Production finished. Consuming...");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});
consumer.start();
producer.start();
}
}
```

代码解释：

- `lock` 是一个用于同步的对象，生产者和消费者线程都需要获取该对象的锁才能执行相应的操作。
- 消费者线程调用 `lock.wait()` 方法进入等待状态，释放锁；生产者线程执行完生产任务后调用 `lock.notify()` 方法唤醒等待的消费者线程。

`java.util.concurrent.locks` 包中的 `Lock` 和 `Condition` 接口提供了比 `synchronized` 更灵活的线程间通信方式。`Condition` 接口的 `await()` 方法类似于 `wait()` 方法，`signal()` 方法类似于 `notify()` 方法，`signalAll()` 方法类似于 `notifyAll()` 方法。

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class LockConditionExample {
    private static final Lock lock = new ReentrantLock();
    private static final Condition condition = lock.newCondition();

    public static void main(String[] args) {
        // 生产者线程
        Thread producer = new Thread(() -> {
            lock.lock();
            try {
                System.out.println("Producer: Producing...");
                Thread.sleep(2000);
                System.out.println("Producer: Production finished. Notifying consumer.");
                // 唤醒等待的线程
                condition.signal();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        });
        // 消费者线程
        Thread consumer = new Thread(() -> {
            lock.lock();
            try {
                System.out.println("Consumer: Waiting for production to finish.");
                // 进入等待状态
                condition.await();
                System.out.println("Consumer: Production finished. Consuming...");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        });
        consumer.start();
        producer.start();
    }
}
```

代码解释：

- `ReentrantLock` 是 `Lock` 接口的一个实现类，`condition` 是通过 `lock.newCondition()` 方法创建的。
- 消费者线程调用 `condition.await()` 方法进入等待状态，生产者线程执行完生产任务后调用 `condition.signal()` 方法唤醒等待的消费者线程。

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

class BlockingQueueExample {
    private static final BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(1);

    public static void main(String[] args) {
        // 生产者线程
        Thread producer = new Thread(() -> {
            try {
                System.out.println("Producer: Producing...");
                queue.put(1);
                System.out.println("Producer: Production finished.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        // 消费者线程
        Thread consumer = new Thread(() -> {
            try {
                System.out.println("Consumer: Waiting for production to finish.");
                int item = queue.take();
                System.out.println("Consumer: Consumed item: " + item);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        consumer.start();
        producer.start();
    }
}
```

代码解释：

- `LinkedBlockingQueue` 是 `BlockingQueue` 接口的一个实现类，容量为 1。
- 生产者线程调用 `queue.put(1)` 方法将元素插入队列，如果队列已满，线程会被阻塞；消费者线程调用 `queue.take()` 方法从队列中取出元素，如果队列为空，线程会被阻塞。

## 线程间通信方式有哪些？

1、Object类的wait()、notify()和notifyAll()方法。这是Java中最基础的线程间通信方式，基于对象的监视器（锁）机制。

- `wait()`：使当前线程进入等待状态，直到其他线程调用该对象的`notify()`或`notifyAll()`方法。
- `notify()`：唤醒在此对象监视器上等待的单个线程。
- `notifyAll()`：唤醒在此对象监视器上等待的所有线程。

### 1. `wait()/notify()`：基于监视器锁的协作

- **原理：**通过对象的监视器锁（`synchronized`）实现等待/唤醒机制。
  - `wait()`：释放锁并进入等待状态。
  - `notify()` / `notifyAll()`：唤醒等待线程（需重新竞争锁）。
- **用途：**生产者-消费者模型、条件等待（如资源就绪）。
- **示例：**

```
Java

synchronized (lock) {
    while (condition) lock.wait(); // 等待条件
    // 处理逻辑
    lock.notifyAll();           // 唤醒其他线程
}
```

- **特点：**

- 必须配合`synchronized`使用。
- 需要手动处理虚假唤醒（通过循环检查条件）。
- 不可重复使用（需重新设置条件）。

2、`Lock` 和 `Condition` 接口。`Lock` 接口提供了比`synchronized`更灵活的锁机制，`Condition`接口则配合`Lock`实现线程间的等待/通知机制。

- `await()`：使当前线程进入等待状态，直到被其他线程唤醒。
- `signal()`：唤醒一个等待在该`Condition`上的线程。
- `signalAll()`：唤醒所有等待在该`Condition`上的线程。

3、`volatile`关键字。`volatile`关键字用于保证变量的可见性，即当一个变量被声明为`volatile`时，它会保证对该变量的写操作会立即刷新到主内存中，而读操作会从主内存中读取最新的值。

4、CountDownLatch。`CountDownLatch` 是一个同步辅助类，它允许一个或多个线程等待其他线程完成操作。

- `CountDownLatch(int count)`：构造函数，指定需要等待的线程数量。
- `countDown()`：减少计数器的值。
- `await()`：使当前线程等待，直到计数器的值为 0。

#### 2. `CountDownLatch`：一次性倒计时器

- **原理：** 初始化计数器值，线程调用 `countDown()` 递减，调用 `await()` 阻塞直到计数器归零。
- **用途：** 主线程等待多个子任务完成（如并行计算后汇总）。
- **示例：**

Java

```
CountDownLatch latch = new CountDownLatch(3);
// 子任务中调用 latch.countDown();
latch.await(); // 主线程等待子任务完成
```

- **特点：**

- 一次性的，计数器归零后无法重置。
- 适用于任务分阶段完成的通知场景。

5、CyclicBarrier。`CyclicBarrier` 是一个同步辅助类，它允许一组线程相互等待，直到所有线程都到达某个公共屏障点。

- `CyclicBarrier(int parties, Runnable barrierAction)`：构造函数，指定参与的线程数量和所有线程到达屏障点后要执行的操作。
- `await()`：使当前线程等待，直到所有线程都到达屏障点。

### 3. CyclicBarrier：可重用的栅栏同步

- **原理：**设置固定线程数作为屏障点，所有线程到达屏障时被阻塞，直到最后一个线程到达后一起释放。
- **用途：**多线程分阶段并行计算（如分布式计算中间结果汇总）。
- **示例：**

Java

```
CyclicBarrier barrier = new CyclicBarrier(3, () -> {
    System.out.println("All threads reached barrier");
});
// 线程中调用 barrier.await();
```

- **特点：**

- 可重复使用（自动重置计数器）。
- 支持所有线程到达后的回调任务。

6、Semaphore。Semaphore是一个计数信号量，它可以控制同时访问特定资源的线程数量。

- `Semaphore(int permits)`：构造函数，指定信号量的初始许可数量。
- `acquire()`：获取一个许可，如果没有可用许可则阻塞。
- `release()`：释放一个许可。

### 4. Semaphore：信号量资源控制

- **原理：**维护一个许可数，线程通过 `acquire()` 获取许可，`release()` 释放许可。
- **用途：**限制资源并发访问数（如数据库连接池控制）。
- **示例：**

Java

```
Semaphore semaphore = new Semaphore(5); // 最多5个线程同时访问
semaphore.acquire(); // 获取许可
// 访问资源后释放
semaphore.release();
```

- **特点：**

- 支持公平/非公平模式。
- 可动态调整许可数量。

| 方式              | 核心目的       | 是否可重用    | 底层机制      | 适用场景        |
|-----------------|------------|----------|-----------|-------------|
| wait()/notify() | 条件等待与唤醒    | 否 (手动重置) | 对象监视器锁    | 生产者-消费者模型   |
| CountDownLatch  | 一次性任务等待    | 否        | 共享计数器     | 主线程等待多子任务完成 |
| CyclicBarrier   | 多线程同步到达屏障点 | 是        | 循环计数器与条件锁 | 分阶段并行计算     |
| Semaphore       | 控制资源访问并发量  | 是        | 许可计数器     | 连接池、限流      |

### 选择建议

- **条件协作:** 优先使用 `wait()/notify()` 或更高层的 `Condition` 接口。
- **任务等待:** 一次性任务用 `CountDownLatch`，可重用场景用 `CyclicBarrier`。
- **资源控制:** `Semaphore` 适合限制并发资源访问。

## 如何停止一个线程？

在 Java 中，停止线程的正确方式是 [通过协作式的逻辑控制线程终止](#)，而非强制暴力终止（如已废弃的 `Thread.stop()`）。以下是实现安全停止线程的多种方法：

**第一种方式：通过共享标志位主动终止。** 定义一个可见的状态变量，由主线程控制其值，工作线程循环检测该变量以决定是否退出。

```
java
public class SafeStopWithFlag implements Runnable {
    // 使用 volatile 保证可见性
    private volatile boolean running = true;

    @Override
    public void run() {
        while (running) {
            try {
                // 处理任务逻辑
                System.out.println("Thread is running...");
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // 捕获中断异常后设置 running=false
                running = false;
                Thread.currentThread().interrupt(); // 重新设置中断标志
            }
        }
        System.out.println("Thread terminated safely.");
    }

    // 停止线程的方法（由外部调用）
    public void stop() {
        running = false;
    }
}
```

调用方式：

```
SafeStopWithFlag task = new SafeStopWithFlag();
Thread thread = new Thread(task);
thread.start();
// 某个时刻调用停止
Thread.sleep(3000);
task.stop();
```

java

**第二种方式使用线程中断机制。**通过 `Thread.interrupt()` 触发线程中断状态，结合中断检测逻辑实现安全停止。

```
public class InterruptExample implements Runnable {
    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            try {
                System.out.println("Working...");
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // 当阻塞时被中断，抛出异常并清除中断状态
                System.out.println("Interrupted during sleep!");
                Thread.currentThread().interrupt(); // 重新设置中断标志
            }
        }
        System.out.println("Thread terminated by interrupt.");
    }
}
```

java

调用方式：

```
Thread thread = new Thread(new InterruptExample());
thread.start();
// 中断线程
Thread.sleep(3000);
thread.interrupt();
```

- `interrupt()` 不会立刻终止线程，只是设置中断标志位。
- 线程需手动检查中断状态（`isInterrupted()`）或触发可中断操作（如 `sleep()`，`wait()`，`join()`）响应中断。
- 阻塞操作中收到中断请求时，会抛出 `InterruptedException` 并清除中断状态。

第三种方式通过 `Future` 取消任务。使用线程池提交任务，并通过 `Future.cancel()` 停止线程，依赖中断机制。

```
public class FutureCancelDemo {  
    public static void main(String[] args) {  
        ExecutorService executor = Executors.newSingleThreadExecutor();  
        Future<?> future = executor.submit(() -> {  
            while (!Thread.currentThread().isInterrupted()) {  
                System.out.println("Task running...");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    System.out.println("Task interrupted.");  
                    Thread.currentThread().interrupt();  
                }  
            }  
        });  
  
        try {  
            Thread.sleep(3000);  
            future.cancel(true); // true表示尝试中断任务线程  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        } finally {  
            executor.shutdown();  
        }  
    }  
}
```

java

\*\*第四种方式\*\*\*\*处理不可中断的阻塞操作。\*\*某些 I/O 或同步操作（如 `Socket.accept()`、`Lock.lock()`）无法通过中断直接响应。此时需结合资源关闭操作。比如，关闭 `Socket` 释放阻塞。

```
java
public class SocketHandler implements Runnable {
    private ServerSocket serverSocket;

    public SocketHandler(ServerSocket serverSocket) {
        this.serverSocket = serverSocket;
    }

    @Override
    public void run() {
        try {
            // serverSocket.accept()阻塞时无法响应中断
            while (!Thread.currentThread().isInterrupted()) {
                Socket socket = serverSocket.accept();
                // 处理连接...
            }
        } catch (IOException e) {
            if (Thread.currentThread().isInterrupted()) {
                System.out.println("Thread stopped by interrupt.");
            }
        }
    }

    // 特殊关闭方法（销毁资源）
    public void stop() {
        try {
            serverSocket.close(); // 关闭资源使accept()抛出异常
        } catch (IOException e) {
            System.out.println("Error closing socket: " + e);
        }
    }
}
```

调用方式：调用 `stop()` 方法关闭资源以解除阻塞。

线程停止的正确实践，如下表格：

| 方法                           | 适用场景                 | 注意事项                                           |
|------------------------------|----------------------|------------------------------------------------|
| 循环检测标志位                      | 简单无阻塞的逻辑             | 确保标志位使用 <code>volatile</code> 或通过锁保证可见性        |
| 中断机制                         | 可中断的阻塞操作             | 正确处理 <code>InterruptedException</code> 并恢复中断标志 |
| <code>Future.cancel()</code> | 线程池管理任务              | 需要线程池任务支持中断处理机制                                |
| 资源关闭                         | 不可中断的阻塞操作（如 Sockets） | 显式关闭资源触发异常，结合中断状态判断回滚                          |

避免使用以下已废弃方法：

- `Thread.stop()`：暴力终止，可能导致状态不一致。
- `Thread.suspend() / resume()`：易导致死锁。

b. 并发安全：

## juc包下你常用的类？

线程池相关：

- `ThreadPoolExecutor`：最核心的线程池类，用于创建和管理线程池。通过它可以灵活地配置线程池的参数，如核心线程数、最大线程数、任务队列等，以满足不同的并发处理需求。
- `Executors`：线程池工厂类，提供了一系列静态方法来创建不同类型的线程池，如 `newFixedThreadPool`（创建固定线程数的线程池）、`newCachedThreadPool`（创建可缓存线程池）、`newSingleThreadExecutor`（创建单线程线程池）等，方便开发者快速创建线程池。

并发集合类：

- `ConcurrentHashMap`：线程安全的哈希映射表，用于在多线程环境下高效地存储和访问键值对。它采用了分段锁等技术，允许多个线程同时访问不同的段，提高了并发性能，在高并发场景下比传统的 `Hashtable` 性能更好。
- `CopyOnWriteArrayList`：线程安全的列表，在对列表进行修改操作时，会创建一个新的底层数组，将修改操作应用到新数组上，而读操作仍然可以在旧数组上进行，从而实现了读写分离，提高了并发读的性能，适用于读多写少的场景。

## 同步工具类：

- `CountDownLatch`：允许一个或多个线程等待其他一组线程完成操作后再继续执行。它通过一个计数器来实现，计数器初始化为线程的数量，每个线程完成任务后调用 `countDown` 方法将计数器减一，当计数器为零时，等待的线程可以继续执行。常用于多个线程完成各自任务后，再进行汇总或下一步操作的场景。
- `CyclicBarrier`：让一组线程互相等待，直到所有线程都到达某个屏障点后，再一起继续执行。与 `CountDownLatch` 不同的是，`CyclicBarrier` 可以重复使用，当所有线程都通过屏障后，计数器会重置，可以再次用于下一轮的等待。适用于多个线程需要协同工作，在某个阶段完成后一起进入下一个阶段的场景。
- `Semaphore`：信号量，用于控制同时访问某个资源的线程数量。它维护了一个许可计数器，线程在访问资源前需要获取许可，如果有可用许可，则获取成功并将许可计数器减一，否则线程需要等待，直到有其他线程释放许可。常用于控制对有限资源的访问，如数据库连接池、线程池中的线程数量等。

## 原子类：

- `AtomicInteger`：原子整数类，提供了对整数类型的原子操作，如自增、自减、比较并交换等。通过硬件级别的原子指令来保证操作的原子性和线程安全性，避免了使用锁带来的性能开销，在多线程环境下对整数进行计数、状态标记等操作非常方便。
- `AtomicReference`：原子引用类，用于对对象引用进行原子操作。可以保证在多线程环境下，对对象的更新操作是原子性的，即要么全部成功，要么全部失败，不会出现数据不一致的情况。常用于实现无锁数据结构或需要对对象进行原子更新的场景。

## 怎么保证多线程安全？

- **synchronized关键字**: 可以使用 `synchronized` 关键字来同步代码块或方法，确保同一时刻只有一个线程可以访问这些代码。对象锁是通过 `synchronized` 关键字锁定对象的监视器（monitor）来实现的。

```
public synchronized void someMethod() { /* ... */ }

public void anotherMethod() {
    synchronized (someObject) {
        /* ... */
    }
}
```

java

- **volatile关键字**: `volatile` 关键字用于变量，确保所有线程看到的是该变量的最新值，而不是可能存储在本地寄存器中的副本。

```
public volatile int sharedVariable;
```

java

- **Lock接口和ReentrantLock类:** `java.util.concurrent.locks.Lock` 接口提供了比 `synchronized` 更强大的锁定机制，`ReentrantLock` 是一个实现该接口的例子，提供了更灵活的锁管理和更高的性能。

```
private final ReentrantLock lock = new ReentrantLock();

public void someMethod() {
    lock.lock();
    try {
        /* ... */
    } finally {
        lock.unlock();
    }
}
```

java

- **原子类:** Java并发库（`java.util.concurrent.atomic`）提供了原子类，如 `AtomicInteger`、`AtomicLong` 等，这些类提供了原子操作，可以用于更新基本类型的变量而无需额外的同步。

示例：

```
AtomicInteger counter = new AtomicInteger(0);

int newValue = counter.incrementAndGet();
```

java

- **线程局部变量:** `ThreadLocal` 类可以为每个线程提供独立的变量副本，这样每个线程都拥有自己的变量，消除了竞争条件。

```
ThreadLocal<Integer> threadLocalVar = new ThreadLocal<>();

threadLocalVar.set(10);
int value = threadLocalVar.get();
```

java

- **并发集合:** 使用 `java.util.concurrent` 包中的线程安全集合，如 `ConcurrentHashMap`、`ConcurrentLinkedQueue` 等，这些集合内部已经实现了线程安全的逻辑。
- **JUC工具类:** 使用 `java.util.concurrent` 包中的一些工具类可以用于控制线程间的同步和协作。例如：`Semaphore` 和 `CyclicBarrier` 等。

## Java中有哪些常用的锁，在什么场景下使用？

Java中的锁是用于管理多线程并发访问共享资源的关键机制。锁可以确保在任意给定时间内只有一个线程可以访问特定的资源，从而避免数据竞争和不一致性。Java提供了多种锁机制，可以分为以下几类：

- **内置锁（synchronized）**：Java中的 `synchronized` 关键字是内置锁机制的基础，可以用于方法或代码块。当一个线程进入 `synchronized` 代码块或方法时，它会获取关联对象的锁；当线程离开该代码块或方法时，锁会被释放。如果其他线程尝试获取同一个对象的锁，它们将被阻塞，直到锁被释放。其中，`synchronized` 加锁时有无锁、偏向锁、轻量级锁和重量级锁几个级别。偏向锁用于当一个线程进入同步块时，如果没有任何其他线程竞争，就会使用偏向锁，以减少锁的开销。轻量级锁使用线程栈上的数据结构，避免了操作系统级别的锁。重量级锁则涉及操作系统级的互斥锁。
- **ReentrantLock**：`java.util.concurrent.locks.ReentrantLock` 是一个显式的锁类，提供了比 `synchronized` 更高级的功能，如可中断的锁等待、定时锁等待、公平锁选项等。`ReentrantLock` 使用 `lock()` 和 `unlock()` 方法来获取和释放锁。其中，公平锁按照线程请求锁的顺序来分配锁，保证了锁分配的公平性，但可能增加锁的等待时间。非公平锁不保证锁分配的顺序，可以减少锁的竞争，提高性能，但可能造成某些线程的饥饿。
- **读写锁（ReadWriteLock）**：`java.util.concurrent.locks.ReadWriteLock` 接口定义了一种锁，允许多个读取者同时访问共享资源，但只允许一个写入者。读写锁通常用于读取远多于写入的情况，以提高并发性。
- **乐观锁和悲观锁**：悲观锁（Pessimistic Locking）通常指在访问数据前就锁定资源，假设最坏的情况，即数据很可能被其他线程修改。`synchronized` 和 `ReentrantLock` 都是悲观锁的例子。乐观锁（Optimistic Locking）通常不锁定资源，而是在更新数据时检查数据是否已被其他线程修改。乐观锁常使用版本号或时间戳来实现。
- **自旋锁**：自旋锁是一种锁机制，线程在等待锁时会持续循环检查锁是否可用，而不是放弃CPU并阻塞。通常可以使用CAS来实现。这在锁等待时间很短的情况下可以提高性能，但过度自旋会浪费CPU资源。

|      |         |          |           |
|------|---------|----------|-----------|
| 无锁   | 无竞争     | 无        | 1ns级      |
| 偏向锁  | 单线程重复访问 | 线程ID验证   | 2ns级      |
| 轻量级锁 | 多线程交替访问 | CAS + 自旋 | 10-100ns级 |
| 重量级锁 | 多线程高竞争  | 内核互斥量    | 1μs-1ms级  |

## 设计思路与优化建议

### 1. 自适应自旋优化 (JDK6+) :

- 根据历史自旋成功次数动态调整自旋次数，避免无意义空转。

### 2. 批量重偏向与撤销 (Bulk Rebiasing) :

- 当一类对象的偏向锁被多个线程争用时，JVM 会批量撤销偏向锁，减少性能波动。

### 3. 锁消除 (Lock Elision) :

- JIT 编译器分析逃逸对象，若无竞争可能，直接消除锁操作（如局部字符串拼接）。

### 4. 实战建议:

- **低竞争场景**: 依赖锁升级机制，无需手动优化。
- **高并发场景**: 优先使用 `ReentrantLock` 或 `StampedLock` (更细粒度控制)。

通过锁升级机制，`synchronized` 在保证线程安全的前提下，大幅降低了无竞争或低竞争场景的开销，成为现代高并发应用的可靠选择。

## 示例：使用ReadWriteLock

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Cache {
    private ReadWriteLock lock = new ReentrantReadWriteLock();
    private Lock readLock = lock.readLock();
    private Lock writeLock = lock.writeLock();
    private Object data;

    public Object readData() {
        readLock.lock();
        try {
            return data;
        } finally {
            readLock.unlock();
        }
    }

    public void writeData(Object newData) {
        writeLock.lock();
        try {
            data = newData;
        } finally {
            writeLock.unlock();
        }
    }
}
```

- c. 并发工具： ConcurrentHashMap, Semaphore, CyclicBarrier, CountDownLatch, Future 和 Callable

- **Future 和 Callable**: Callable 是一个类似于 Runnable 的接口，但它可以返回结果，并且可以抛出异常。Future 用于表示一个异步计算的结果，可以通过它来获取 Callable 任务的执行结果或取消任务。代码如下：

```
java

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FutureCallableExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newSingleThreadExecutor();

        Callable<Integer> callable = () -> {
            System.out.println(Thread.currentThread().getName() + " 开始执行 Callable 任务");
            Thread.sleep(2000); // 模拟耗时操作
            return 42; // 返回结果
        };

        Future<Integer> future = executorService.submit(callable);
        System.out.println("主线程继续执行其他任务");

        try {
            Integer result = future.get(); // 等待 Callable 任务完成并获取结果
            System.out.println("Callable 任务的结果: " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }

        executorService.shutdown();
    }
}
```

d. synchronized:

## synchronized和reentrantlock及其应用场景？

### synchronized 工作原理

synchronized是Java提供的原子性内置锁，这种内置的并且使用者看不到的锁也被称为[监视器锁](#)，

使用synchronized之后，会在编译之后在同步的代码块前后加上monitorenter和monitorexit字节码指令，他依赖操作系统底层互斥锁实现。他的作用主要就是实现原子性操作和解决共享变量的内存可见性问题。

执行monitorenter指令时会尝试获取对象锁，如果对象没有被锁定或者已经获得了锁，锁的计数器+1。此时其他竞争锁的线程则会进入等待队列中。执行monitorexit指令时则会把计数器-1，当计数器值为0时，则锁释放，处于等待队列中的线程再继续竞争锁。

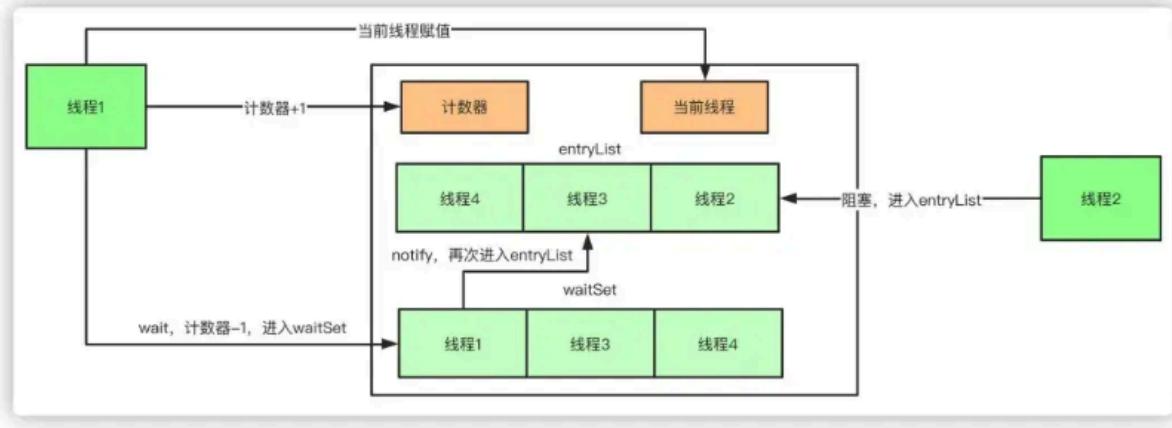
synchronized是排它锁，当一个线程获得锁之后，其他线程必须等待该线程释放锁后才能获得锁，而且由于Java中的线程和操作系统原生线程是一一对应的，线程被阻塞或者唤醒时时会从用户态切换到内核态，这种转换非常消耗性能。

从内存语义来说，加锁的过程会清除工作内存中的共享变量，再从主内存读取，而释放锁的过程则是将工作内存中的共享变量写回主内存。

实际上大部分时候我认为说到monitorenter就行了，但是为了更清楚的描述，还是再具体一点。

如果再深入到源码来说，synchronized实际上有两个队列waitSet和entryList。

1. 当多个线程进入同步代码块时，首先进入entryList
2. 有一个线程获取到monitor锁后，就赋值给当前线程，并且计数器+1
3. 如果线程调用wait方法，将释放锁，当前线程置为null，计数器-1，同时进入waitSet等待被唤醒，调用notify或者notifyAll之后又会进入entryList竞争锁
4. 如果线程执行完毕，同样释放锁，计数器-1，当前线程置为null



### reentrantlock工作原理

ReentrantLock 的底层实现主要依赖于 AbstractQueuedSynchronizer (AQS) 这个抽象类。AQS 是一个提供了基本同步机制的框架，其中包括了队列、状态值等。

ReentrantLock 在 AQS 的基础上通过内部类 Sync 来实现具体的锁操作。不同的 Sync 子类实现了公平锁和非公平锁的不同逻辑：

- 可中断性：** ReentrantLock 实现了可中断性，这意味着线程在等待锁的过程中，可以被其他线程中断而提前结束等待。在底层，ReentrantLock 使用了与 LockSupport.park() 和 LockSupport.unpark() 相关的机制来实现可中断性。
- 设置超时时间：** ReentrantLock 支持在尝试获取锁时设置超时时间，即等待一定时间后如果还未获得锁，则放弃锁的获取。这是通过内部的 tryAcquireNanos 方法来实现的。
- 公平锁和非公平锁：** 在直接创建 ReentrantLock 对象时，默认情况下是非公平锁。公平锁是按照线程等待的顺序来获取锁，而非公平锁则允许多个线程在同一时刻竞争锁，不考虑它们申请锁的顺序。公平锁可以通过在创建 ReentrantLock 时传入 true 来设置，例如：

```
ReentrantLock fairLock = new ReentrantLock(true);
```

java

- **多个条件变量：** ReentrantLock 支持多个条件变量，每个条件变量可以与一个 ReentrantLock 关联。这使得线程可以更灵活地进行等待和唤醒操作，而不仅仅是基于对象监视器的 wait() 和 notify()。多个条件变量的实现依赖于 Condition 接口，例如：

```
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();
// 使用下面方法进行等待和唤醒
condition.await();
condition.signal();
```

java

- **可重入性：** ReentrantLock 支持可重入性，即同一个线程可以多次获得同一把锁，而不会造成死锁。这是通过内部的 holdCount 计数来实现的。当一个线程多次获取锁时，holdCount 递增，释放锁时递减，只有当 holdCount 为零时，其他线程才有机会获取锁。

### 应用场景的区别

`synchronized`:

- **简单同步需求：** 当你需要对代码块或方法进行简单的同步控制时，`synchronized` 是一个很好的选择。它使用起来简单，不需要额外的资源管理，因为锁会在方法退出或代码块执行完毕后自动释放。
- **代码块同步：** 如果你想对特定代码段进行同步，而不是整个方法，可以使用 `synchronized` 代码块。这可以让你更精细地控制同步的范围，从而减少锁的持有时间，提高并发性能。
- **内置锁的使用：** `synchronized` 关键字使用对象的内置锁（也称为监视器锁），这在需要使用对象作为锁对象的情况下很有用，尤其是在对象状态与锁保护的代码紧密相关时。

`ReentrantLock`:

- **高级锁功能需求：** `ReentrantLock` 提供了 `synchronized` 所不具备的高级功能，如公平锁、响应中断、定时锁尝试、以及多个条件变量。当你需要这些功能时，`ReentrantLock` 是更好的选择。
- **性能优化：** 在高度竞争的环境中，`ReentrantLock` 可以提供比 `synchronized` 更好的性能，因为它提供了更细粒度的控制，如尝试锁定和定时锁定，可以减少线程阻塞的可能性。
- **复杂同步结构：** 当你需要更复杂的同步结构，如需要多个条件变量来协调线程之间的通信时，`ReentrantLock` 及其配套的 `Condition` 对象可以提供更灵活的解决方案。

综上，`synchronized` 适用于简单同步需求和不需要额外锁功能的场景，而 `ReentrantLock` 适用于需要更高级锁功能、性能优化或复杂同步逻辑的情况。选择哪种同步机制取决于具体的应用需求和性能考虑。

## 除了用synchronized，还有什么方法可以实现线程同步？

- **使用 ReentrantLock 类：** ReentrantLock 是一个可重入的互斥锁，相比 synchronized 提供了更灵活的锁定和解锁操作。它还支持公平锁和非公平锁，以及可以响应中断的锁获取操作。
- **使用 volatile 关键字：** 虽然 volatile 不是一种锁机制，但它可以确保变量的可见性。当一个变量被声明为 volatile 后，线程将直接从主内存中读取该变量的值，这样就能保证线程间变量的可见性。但它不具备原子性。
- **使用 Atomic 类：** Java 提供了一系列的原子类，例如 AtomicInteger、AtomicLong、AtomicReference 等，用于实现对单个变量的原子操作，这些类在实现细节上利用了 CAS（Compare-And-Swap）算法，可以用来实现无锁的线程安全。

## synchronized锁静态方法和普通方法区别？

锁的对象不同：

- **普通方法：** 锁的是当前对象实例（`this`）。同一对象实例的 synchronized 普通方法，同一时间只能被一个线程访问；不同对象实例间互不影响，可被不同线程同时访问各自的同步普通方法。
- **静态方法：** 锁的是当前类的 `Class` 对象。由于类的 `Class` 对象全局唯一，无论多少个对象实例，该静态同步方法同一时间只能被一个线程访问。

作用范围不同：

- **普通方法：** 仅对同一对象实例的同步方法调用互斥，不同对象实例的同步普通方法可并行执行。
- **静态方法：** 对整个类的所有实例的该静态方法调用都互斥，一个线程进入静态同步方法，其他线程无法进入同一类任何实例的该方法。

多实例场景影响不同：

- **普通方法：** 多线程访问不同对象实例的同步普通方法时，可同时执行。
- **静态方法：** 不管有多少对象实例，同一时间仅一个线程能执行该静态同步方法。

## synchronized和reentrantlock区别？

synchronized 和 ReentrantLock 都是 Java 中提供的可重入锁：

- **用法不同：** synchronized 可用来修饰普通方法、静态方法和代码块，而 ReentrantLock 只能用在代码块上。
- **获取锁和释放锁方式不同：** synchronized 会自动加锁和释放锁，当进入 synchronized 修饰的代码块之后会自动加锁，当离开 synchronized 的代码段之后会自动释放锁。而 ReentrantLock 需要手动加锁和释放锁
- **锁类型不同：** synchronized 属于非公平锁，而 ReentrantLock 既可以是公平锁也可以是非公平锁。
- **响应中断不同：** ReentrantLock 可以响应中断，解决死锁的问题，而 synchronized 不能响应中断。
- **底层实现不同：** synchronized 是 JVM 层面通过监视器实现的，而 ReentrantLock 是基于 AQS 实现的。

## 怎么理解可重入锁？

可重入锁是指同一个线程在获取了锁之后，可以再次重复获取该锁而不会造成死锁或其他问题。当一个线程持有锁时，如果再次尝试获取该锁，就会成功获取而不会被阻塞。

ReentrantLock实现可重入锁的机制是基于线程持有锁的计数器。

- 当一个线程第一次获取锁时，计数器会加1，表示该线程持有了锁。在此之后，如果同一个线程再次获取锁，计数器会再次加1。每次线程成功获取锁时，都会将计数器加1。
- 当线程释放锁时，计数器会相应地减1。只有当计数器减到0时，锁才会完全释放，其他线程才有机会获取锁。

这种计数器的设计使得同一个线程可以多次获取同一个锁，而不会造成死锁或其他问题。每次获取锁时，计数器加1；每次释放锁时，计数器减1。只有当计数器减到0时，锁才会完全释放。

ReentrantLock通过这种计数器的方式，实现了可重入锁的机制。它允许同一个线程多次获取同一个锁，并且能够正确地处理锁的获取和释放，避免了死锁和其他并发问题。

## synchronized 支持重入吗？如何实现的？

synchronized是基于原子性的内部锁机制，是可重入的，因此在一个线程调用synchronized方法的同时在其方法体内部调用该对象另一个synchronized方法，也就是说一个线程得到一个对象锁后再次请求该对象锁，是允许的，这就是synchronized的可重入性。

synchronized底层是利用计算机系统mutex Lock实现的。每一个可重入锁都会关联一个线程ID和一个锁状态status。

当一个线程请求方法时，会去检查锁状态。

1. 如果锁状态是0，代表该锁没有被占用，使用CAS操作获取锁，将线程ID替换成自己的线程ID。
2. 如果锁状态不是0，代表有线程在访问该方法。此时，如果线程ID是自己的线程ID，如果是可重入锁，会将status自增1，然后获取到该锁，进而执行相应的方法；如果是非重入锁，就会进入阻塞队列等待。

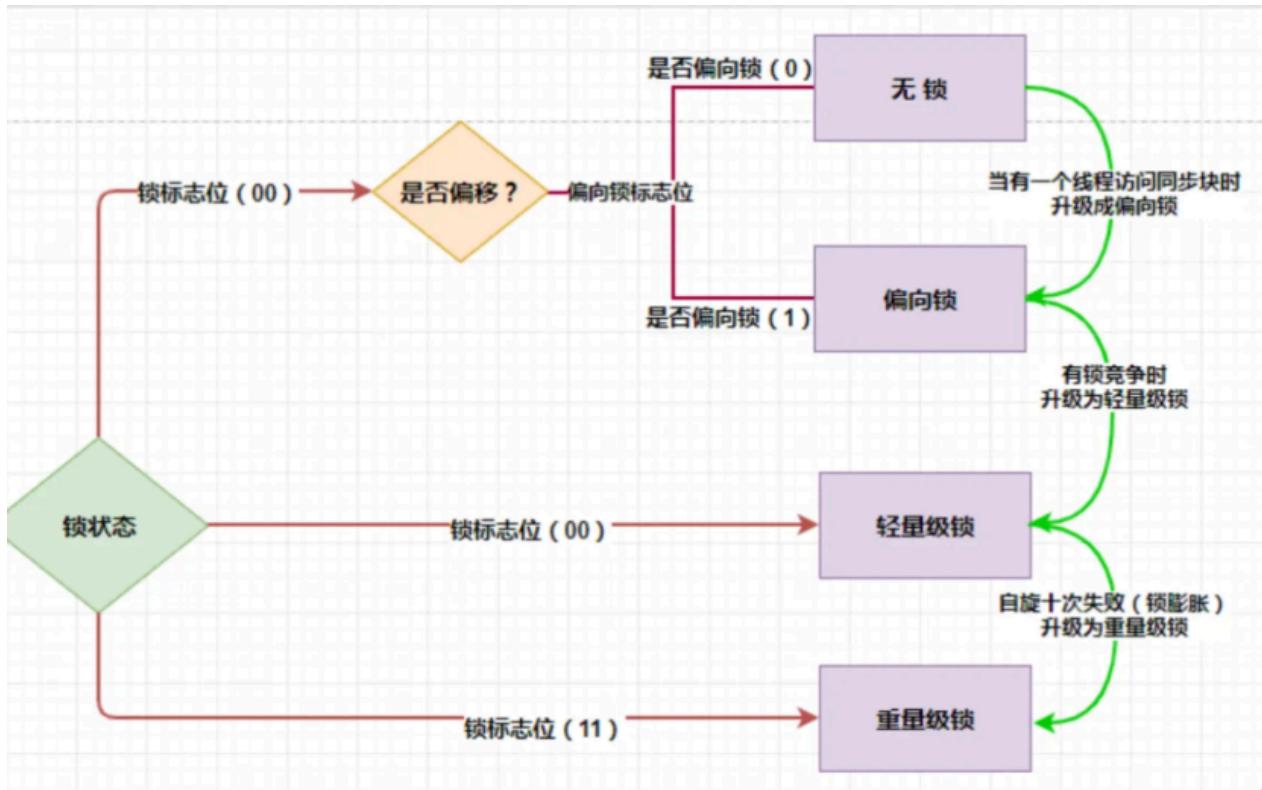
在释放锁时，

1. 如果是可重入锁的，每一次退出方法，就会将status减1，直至status的值为0，最后释放该锁。
2. 如果非可重入锁的，线程退出方法，直接就会释放该锁。

## syncronized锁升级的过程讲一下

具体的锁升级的过程是：**无锁->偏向锁->轻量级锁->重量级锁。**

- **无锁**: 这是没有开启偏向锁的时候的状态，在JDK1.6之后偏向锁的默认开启的，但是有一个偏向延迟，需要在JVM启动之后的多少秒之后才能开启，这个可以通过JVM参数进行设置，同时是否开启偏向锁也可以通过JVM参数设置。
- **偏向锁**: 这个是在偏向锁开启之后的锁的状态，如果还没有一个线程拿到这个锁的话，这个状态叫做匿名偏向，当一个线程拿到偏向锁的时候，下次想要竞争锁只需要拿线程ID跟MarkWord当中存储的线程ID进行比较，如果线程ID相同则直接获取锁（相当于锁偏向于这个线程），不需要进行CAS操作和将线程挂起的操作。
- **轻量级锁**: 在这个状态下线程主要是通过CAS操作实现的。将对象的MarkWord存储到线程的虚拟机栈上，然后通过CAS将对象的MarkWord的内容设置为指向Displaced Mark Word的指针，如果设置成功则获取锁。在线程出临界区的时候，也需要使用CAS，如果使用CAS替换成功则同步成功，如果失败表示有其他线程在获取锁，那么就需要在释放锁之后将被挂起的线程唤醒。
- **重量级锁**: 当有两个以上的线程获取锁的时候轻量级锁就会升级为重量级锁，因为CAS如果没有成功的话始终都在自旋，进行while循环操作，这是非常消耗CPU的，但是在升级为重量级锁之后，线程会被操作系统调度然后挂起，这可以节约CPU资源。



线程A进入 synchronized 开始抢锁，JVM 会判断当前是否是偏向锁的状态，如果是就会根据 Mark Word 中存储的线程 ID 来判断，当前线程A是否就是持有偏向锁的线程。如果是，则忽略 check，线程A直接执行临界区内的代码。

但如果 Mark Word 里的线程不是线程 A，就会通过自旋尝试获取锁，如果获取到了，就将 Mark Word 中的线程 ID 改为自己的；如果竞争失败，就会立马撤销偏向锁，膨胀为轻量级锁。

后续的竞争线程都会通过自旋来尝试获取锁，如果自旋成功那么锁的状态仍然是轻量级锁。然而如果竞争失败，锁会膨胀为重量级锁，后续等待的竞争的线程都会被阻塞。

## JVM对Synchronized的优化？

synchronized 核心优化方案主要包含以下 4 个：

- 锁膨胀**：synchronized 从无锁升级到偏向锁，再到轻量级锁，最后到重量级锁的过程，它叫做锁膨胀也叫做锁升级。JDK 1.6 之前，synchronized 是重量级锁，也就是说 synchronized 在释放和获取锁时都会从用户态转换成内核态，而转换的效率是比较低的。但有了锁膨胀机制之后，synchronized 的状态就多了无锁、偏向锁以及轻量级锁了，这时候在进行并发操作时，大部分的场景都不需要用户态到内核态的转换了，这样就大幅的提升了 synchronized 的性能。
- 锁消除**：指的是在某些情况下，JVM 虚拟机如果检测不到某段代码被共享和竞争的可能性，就会将这段代码所属的同步锁消除掉，从而到底提高程序性能的目的。
- 锁粗化**：将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。
- 自适应自旋锁**：指通过自身循环，尝试获取锁的一种方式，优点在于它避免一些线程的挂起和恢复操作，因为挂起线程和恢复线程都需要从用户态转入内核态，这个过程是比较慢的，所以通过自旋的方式可以一定程度上避免线程挂起和恢复所造成的性能开销。

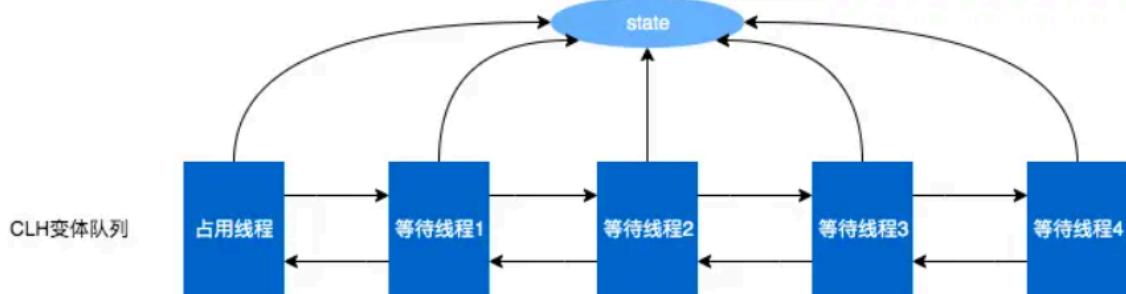
### e. CAS和AQS:

AQS全称为AbstractQueuedSynchronizer，是Java中的一个抽象类。AQS是一个用于构建锁、同步器、协作工具类的工具类（框架）。

AQS核心思想是，如果被请求的共享资源空闲，那么就将当前请求资源的线程设置为有效的工作线程，将共享资源设置为锁定状态；如果共享资源被占用，就需要一定的阻塞等待唤醒机制来保证锁分配。这个机制主要用的是CLH队列的变体实现的，将暂时获取不到锁的线程加入到队列中。

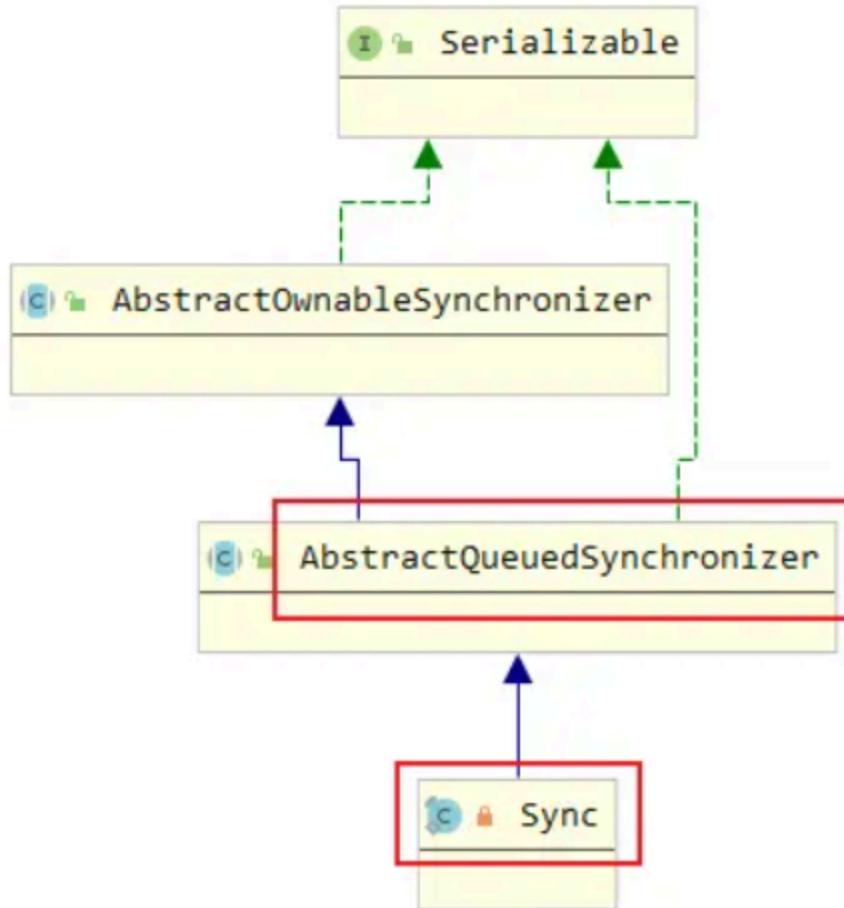
CLH：Craig、Landin and Hagersten队列，是单向链表，AQS中的队列是CLH变体的虚拟双向队列（FIFO），AQS是通过将每条请求共享资源的线程封装成一个节点来实现锁的分配。

主要原理图如下：



AQS使用一个Volatile的int类型的成员变量来表示同步状态，通过内置的FIFO队列来完成资源获取的排队工作，通过CAS完成对State值的修改。

其中 Sync 是这些类中都有的内部类，其结构如下：



可以看到： Sync 是 AQS 的实现。 AQS 主要完成的任务：

- 同步状态（比如说计数器）的原子性管理；
- 线程的阻塞和解除阻塞；
- 队列的管理。

AQS最核心的就是三大部分：

- 状态：state；
- 控制线程抢锁和配合的FIFO队列（双向链表）；
- 期望协作工具类去实现的获取/释放等重要方法（重写）。

### 状态state

- 这里state的具体含义，会根据具体实现类的不同而不同：比如在Semaphore里，他表示剩余许可证的数量；在CountDownLatch里，它表示还需要倒数的数量；在ReentrantLock中，state用来表示“锁”的占有情况，包括可重入计数，当state的值为0的时候，标识该Lock不被任何线程所占有。
- state是volatile修饰的，并被并发修改，所以修改state的方法都需要保证线程安全，比如getState、setState以及compareAndSetState操作来读取和更新这个状态。这些方法都依赖于unsafe类。

### FIFO队列

- 这个队列用来存放“等待的线程”，AQS就是“排队管理器”，当多个线程争用同一把锁时，必须有排队机制将那些没能拿到锁的线程串在一起。当锁释放时，锁管理器就会挑选一个合适的线程来占有这个刚刚释放的锁。
- AQS会维护一个等待的线程队列，把线程都放到这个队列里，这个队列是双向链表形式。

### 实现获取/释放等方法

- 这里的获取和释放方法，是利用AQS的协作工具类里最重要的方法，是由协作类自己去实现的，并且含义各不相同；
- 获取方法：获取操作会以来state变量，经常会阻塞（比如获取不到锁的时候）。在Semaphore中，获取就是acquire方法，作用是获取一个许可证；而在CountDownLatch里面，获取就是await方法，作用是等待，直到倒数结束；
- 释放方法：在Semaphore中，释放就是release方法，作用是释放一个许可证；在CountDownLatch里面，获取就是countDown方法，作用是将倒数的数减一；
- 需要每个实现类重写tryAcquire和tryRelease等方法。

CAS 和 AQS 两者的区别：

- CAS 是一种乐观锁机制，它包含三个操作数：内存位置 (V)、预期值 (A) 和新值 (B)。CAS 操作的逻辑是，如果内存位置 V 的值等于预期值 A，则将其更新为新值 B，否则不做任何操作。整个过程是原子性的，通常由硬件指令支持，如在现代处理器上，`cmpxchg` 指令可以实现 CAS 操作。
- AQS 是一个用于构建锁和同步器的框架，许多同步器如 `ReentrantLock`、`Semaphore`、`CountDownLatch` 等都是基于 AQS 构建的。AQS 使用一个 `volatile` 的整数变量 `state` 来表示同步状态，通过内置的 `FIFO` 队列来管理等待线程。它提供了一些基本的操作，如 `acquire` (获取资源) 和 `release` (释放资源)，这些操作会修改 `state` 的值，并根据 `state` 的值来判断线程是否可以获取或释放资源。AQS 的 `acquire` 操作通常会先尝试获取资源，如果失败，线程将被添加到等待队列中，并阻塞等待。`release` 操作会释放资源，并唤醒等待队列中的线程。

CAS 和 AQS 两者的联系：

- **CAS 为 AQS 提供原子操作支持**: AQS 内部使用 CAS 操作来更新 `state` 变量，以实现线程安全的状态修改。在 `acquire` 操作中，当线程尝试获取资源时，会使用 CAS 操作尝试将 `state` 从一个值更新为另一个值，如果更新失败，说明资源已被占用，线程会进入等待队列。在 `release` 操作中，当线程释放资源时，也会使用 CAS 操作将 `state` 恢复到相应的值，以保证状态更新的原子性。

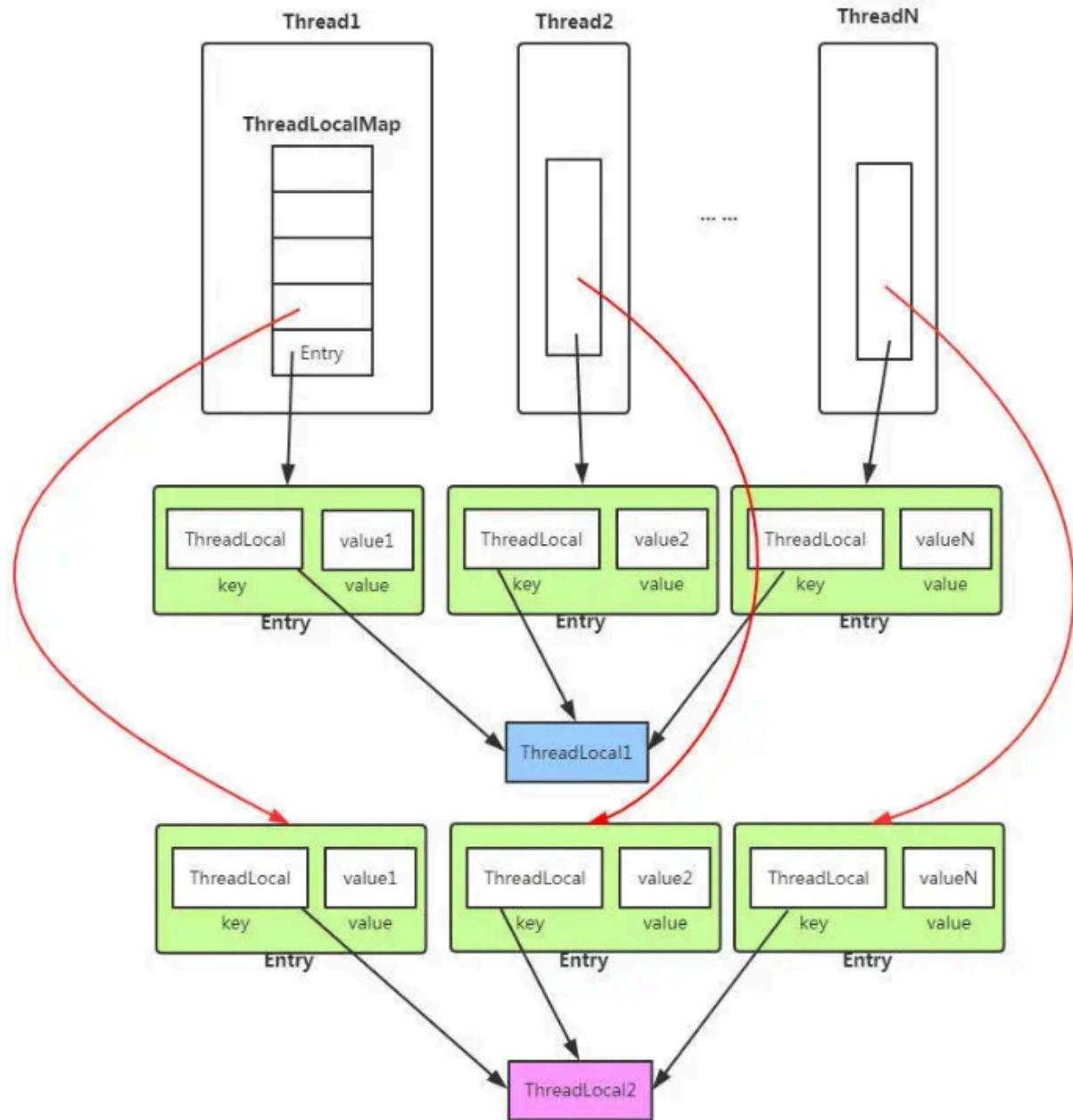
## 如何用 AQS 实现一个可重入的公平锁？

AQS 实现一个可重入的公平锁的详细步骤：

1. **继承 AbstractQueuedSynchronizer**: 创建一个内部类继承自 `AbstractQueuedSynchronizer`，重写 `tryAcquire`、`tryRelease`、`isHeldExclusively` 等方法，这些方法将用于实现锁的获取、释放和判断锁是否被当前线程持有。
2. **实现可重入逻辑**: 在 `tryAcquire` 方法中，检查当前线程是否已经持有锁，如果是，则增加锁的持有次数 (通过 `state` 变量)；如果不是，尝试使用 CAS 操作来获取锁。
3. **实现公平性**: 在 `tryAcquire` 方法中，按照队列顺序来获取锁，即先检查等待队列中是否有线程在等待，如果有，当前线程必须进入队列等待，而不是直接竞争锁。
4. **创建锁的外部类**: 创建一个外部类，内部持有 `AbstractQueuedSynchronizer` 的子类对象，并提供 `lock` 和 `unlock` 方法，这些方法将调用 `AbstractQueuedSynchronizer` 子类中的方法。

## Threadlocal作用，原理，具体里面存的key value是啥，会有什么问题，如何解决？

ThreadLocal 是Java中用于解决线程安全问题的一种机制，它允许创建线程局部变量，即每个线程都有自己独立的变量副本，从而避免了线程间的资源共享和同步问题。



从内存结构图，我们可以看到：

- Thread类中，有个ThreadLocal.ThreadLocalMap 的成员变量。
- ThreadLocalMap内部维护了Entry数组，每个Entry代表一个完整的对象，key是ThreadLocal本身，value是ThreadLocal的泛型对象值。

### ThreadLocal的作用

- **线程隔离**: ThreadLocal 为每个线程提供了独立的变量副本，这意味着线程之间不会相互影响，可以安全地在多线程环境中使用这些变量而不必担心数据竞争或同步问题。
- **降低耦合度**: 在同一个线程内的多个函数或组件之间，使用 ThreadLocal 可以减少参数的传递，降低代码之间的耦合度，使代码更加清晰和模块化。
- **性能优势**: 由于 ThreadLocal 避免了线程间的同步开销，所以在大量线程并发执行时，相比传统的锁机制，它可以提供更好的性能。

### ThreadLocal的原理

ThreadLocal 的实现依赖于 Thread 类中的一个 ThreadLocalMap 字段，这是一个存储 ThreadLocal 变量本身和对应值的映射。每个线程都有自己的 ThreadLocalMap 实例，用于存储该线程所持有的所有 ThreadLocal 变量的值。

当你创建一个 ThreadLocal 变量时，它实际上就是一个 ThreadLocal 对象的实例。每个 ThreadLocal 对象都可以存储任意类型的值，这个值对每个线程来说是独立的。

- 当调用 ThreadLocal 的 get() 方法时，ThreadLocal 会检查当前线程的 ThreadLocalMap 中是否有与之关联的值。
  - 如果有，返回该值；
  - 如果没有，会调用 initialValue() 方法（如果重写了的话）来初始化该值，然后将其放入 ThreadLocalMap 中并返回。
- 当调用 set() 方法时，ThreadLocal 会将给定的值与当前线程关联起来，即在当前线程的 ThreadLocalMap 中存储一个键值对，键是 ThreadLocal 对象自身，值是传入的值。
- 当调用 remove() 方法时，会从当前线程的 ThreadLocalMap 中移除与该 ThreadLocal 对象关联的条目。

## 可能存在的问题

当一个线程结束时，其 `ThreadLocalMap` 也会随之销毁，但是 `ThreadLocal` 对象本身不会立即被垃圾回收，直到没有其他引用指向它为止。

因此，在使用 `ThreadLocal` 时需要注意，**如果不显式调用 `remove()` 方法，或者线程结束时未正确清理 `ThreadLocal` 变量，可能会导致内存泄漏，因为 `ThreadLocalMap` 会持续持有 `ThreadLocal` 变量的引用，即使这些变量不再被其他地方引用。**

因此，实际应用中需要在使用完 `ThreadLocal` 变量后调用 `remove()` 方法释放资源。

## 悲观锁和乐观锁的区别？

- 乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。
- 悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像 `synchronized`，不管三七二十一，直接上了锁就操作资源了。

## Java中想实现一个乐观锁，都有哪些方式？

1. **CAS (Compare and Swap) 操作：** CAS 是乐观锁的基础。Java 提供了 `java.util.concurrent.atomic` 包，包含各种原子变量类（如 `AtomicInteger`、`AtomicLong`），这些类使用 CAS 操作实现了线程安全的原子操作，可以用来实现乐观锁。
2. **版本号控制：** 增加一个版本号字段记录数据更新时候的版本，每次更新时递增版本号。在更新数据时，同时比较版本号，若当前版本号和更新前获取的版本号一致，则更新成功，否则失败。
3. **时间戳：** 使用时间戳记录数据的更新时间，在更新数据时，在比较时间戳。如果当前时间戳大于数据的时间戳，则说明数据已经被其他线程更新，更新失败。

## CAS有什么缺点？

CAS的缺点主要有3点：

- **ABA问题**: ABA的问题指的是在CAS更新的过程中，当读取到的值是A，然后准备赋值的时候仍然是A，但是实际上有可能A的值被改成了B，然后又被改回了A，这个CAS更新的漏洞就叫做ABA。只是ABA的问题大部分场景下都不影响并发的最终效果。Java中有AtomicStampedReference来解决这个问题，他加入了预期标志和更新后标志两个字段，更新时不光检查值，还要检查当前的标志是否等于预期标志，全部相等的话才会更新。
- **循环时间长开销大**: 自旋CAS的方式如果长时间不成功，会给CPU带来很大的开销。
- **只能保证一个共享变量的原子操作**: 只对一个共享变量操作可以保证原子性，但是多个可以通过AtomicReference来处理或者使用锁synchronized实现。

## 为什么不能所有的锁都用CAS？

CAS操作是基于循环重试的机制，如果CAS操作一直未能成功，线程会一直自旋重试，占用CPU资源。在高并发情况下，大量线程自旋会导致CPU资源浪费。

## CAS有什么问题，Java是怎么解决的？

会有ABA的问题，变量值在操作过程中先被其他线程从 **A** 修改为 **B**，又被改回 **A**，CAS无法感知中途变化，导致操作误判为“未变更”。比如：

- 线程1读取变量为 **A**，准备改为 **C**。
- 此时线程2将变量 **A → B → A**。
- 线程1的CAS执行时发现仍是 **A**，但状态已丢失中间变化。

Java提供的工具类会在CAS操作中增加**版本号（Stamp）或标记**，每次修改都更新版本号，使得即使值相同也能识别变更历史。比如，可以用AtomicStampedReference来解决ABA问题，通过比对值和**版本号**识别ABA问题。

```
java
AtomicStampedReference<Integer> ref = new AtomicStampedReference<>(100, 0);

// 尝试修改值并更新版本号
boolean success = ref.compareAndSet(100, 200, 0, 1);
// 前提：当前值=100 且 版本号=0，才会更新为 (200,1)
```

## volatile关键字有什么作用？

volatile作用有 2 个：

- **保证变量对所有线程的可见性。**当一个变量被声明为volatile时，它会保证对这个变量的写操作会立即刷新到主存中，而对这个变量的读操作会直接从主存中读取，从而确保了多线程环境下对该变量访问的可见性。这意味着一个线程修改了volatile变量的值，其他线程能够立刻看到这个修改，不会受到各自线程工作内存的影响。
- **禁止指令重排序优化。**volatile关键字在Java中主要通过内存屏障来禁止特定类型的指令重排序。
  - 1) **写-写（Write-Write）屏障：**在对volatile变量执行写操作之前，会插入一个写屏障。这确保了在该变量写操作之前的所有普通写操作都已完成，防止了这些写操作被移到volatile写操作之后。
  - 2) **读-写（Read-Write）屏障：**在对volatile变量执行读操作之后，会插入一个读屏障。它确保了对volatile变量的读操作之后的所有普通读操作都不会被提前到volatile读之前执行，保证了读取到的数据是最新的。
  - 3) **写-读（Write-Read）屏障：**这是最重要的一个屏障，它发生在volatile写之后和volatile读之前。这个屏障确保了volatile写操作之前的所有内存操作（包括写操作）都不会被重排序到volatile读之后，同时也确保了volatile读操作之后的所有内存操作（包括读操作）都不会被重排序到volatile写之前。

## 指令重排序的原理是什么？

在执行程序时，为了提高性能，处理器和编译器常常会对指令进行重排序，但是重排序要满足下面 2 个条件才能进行：

- 在单线程环境下不能改变程序运行的结果
- 存在数据依赖关系的不允许重排序。

所以重排序不会对单线程有影响，只会破坏多线程的执行语义。

我们看这个例子，A和C之间存在数据依赖关系，同时B和C之间也存在数据依赖关系。因此在最终执行的指令序列中，C不能被重排序到A和B的前面，如果C排到A和B的前面，那么程序的结果将会被改变。但A和B之间没有数据依赖关系，编译器和处理器可以重排序A和B之间的执行顺序。

```
1 double width = 15.67;           //A
2 double height = 14.32;          //B
3 double area = width * height; //C
```

## volatile可以保证线程安全吗？

volatile关键字可以保证可见性，但不能保证原子性，因此不能完全保证线程安全。volatile关键字用于修饰变量，当一个线程修改了volatile修饰的变量的值，其他线程能够立即看到最新的值，从而避免了线程之间的数据不一致。

但是，volatile并不能解决多线程并发下的复合操作问题，比如`i++`这种操作不是原子操作，如果多个线程同时对`i`进行自增操作，volatile不能保证线程安全。对于复合操作，需要使用synchronized关键字或者Lock来保证原子性和线程安全。

## volatile和synchronized比较？

Synchronized解决了多线程访问共享资源时可能出现的竞态条件和数据不一致的问题，保证了线程安全性。Volatile解决了变量在多线程环境下的可见性和有序性问题，确保了变量的修改对其他线程是可见的。

- Synchronized: Synchronized是一种排他性的同步机制，保证了多个线程访问共享资源时的互斥性，即同一时刻只允许一个线程访问共享资源。通过对代码块或方法添加Synchronized关键字来实现同步。
- Volatile: Volatile是一种轻量级的同步机制，用来保证变量的可见性和禁止指令重排序。当一个变量被声明为Volatile时，线程在读取该变量时会直接从内存中读取，而不会使用缓存，同时对该变量的写操作会立即刷回主内存，而不是缓存在本地内存中。

## 什么是公平锁和非公平锁？

- **公平锁：**指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁。公平锁的优点在于各个线程公平平等，每个线程等待一段时间后，都有执行的机会，而它的缺点就在于整体执行速度更慢，吞吐量更小。
- **非公平锁：**多个线程加锁时直接尝试获取锁，能抢到锁就直接占有锁，抢不到才会到等待队列的队尾等待。非公平锁的优势就在于整体执行速度更快，吞吐量更大，但同时也可能产生线程饥饿问题，也就是说如果一直有线程插队，那么在等待队列中的线程可能长时间得不到运行。

## 非公平锁吞吐量为什么比公平锁大？

- **公平锁执行流程：**获取锁时，先将线程自己添加到等待队列的队尾并休眠，当某线程用完锁之后，会去唤醒等待队列中队首的线程尝试去获取锁，锁的使用顺序也就是队列中的先后顺序，在整个过程中，线程会从运行状态切换到休眠状态，再从休眠状态恢复成运行状态，但线程每次休眠和恢复都需要从用户态转换成内核态，而这个状态的转换是比较慢的，所以公平锁的执行速度会比较慢。
- **非公平锁执行流程：**当线程获取锁时，会先通过 CAS 尝试获取锁，如果获取成功就直接拥有锁，如果获取锁失败才会进入等待队列，等待下次尝试获取锁。这样做的好处是，获取锁不用遵循先到先得的规则，从而避免了线程休眠和恢复的操作，这样就加速了程序的执行效率。

## Synchronized是公平锁吗？

Synchronized不属于公平锁，ReentrantLock是公平锁。

## ReentrantLock是怎么实现公平锁的？

我们来看一下公平锁与非公平锁的加锁方法的源码。公平锁的锁获取源码如下：

```
protected final boolean tryAcquire(int acquires) {  
  
    final Thread current = Thread.currentThread();  
    int c = getState();  
  
    if (c == 0) {  
  
        if (!hasQueuedPredecessors() && //这里判断了 hasQueuedPredecessors()  
            compareAndSetState(0, acquires)) {  
  
            setExclusiveOwnerThread(current);  
  
            return true;  
        }  
  
    } else if (current == getExclusiveOwnerThread()) {  
  
        int nextc = c + acquires;  
  
        if (nextc < 0) {  
            throw new Error("Maximum lock count exceeded");  
        }  
        setState(nextc);  
        return true;  
  
    }  
    return false;  
}
```

不公平锁的锁获取源码如下：

java

```
final boolean nonfairTryAcquire(int acquires) {  
  
    final Thread current = Thread.currentThread();  
    int c = getState();  
  
    if (c == 0) {  
  
        if (compareAndSetState(0, acquires)) { //这里没有判断 hasQueuedPredecessors()  
  
            setExclusiveOwnerThread(current);  
  
            return true;  
  
        }  
  
    }  
  
    else if (current == getExclusiveOwnerThread()) {  
  
        int nextc = c + acquires;  
  
        if (nextc < 0) // overflow  
  
            throw new Error("Maximum lock count exceeded");  
  
        setState(nextc);  
  
        return true;  
  
    }  
  
    return false;  
}
```

通过对比，我们可以明显的看出公平锁与非公平锁的 lock() 方法唯一的区别就在于公平锁在获取锁时多了一个限制条件：hasQueuedPredecessors() 为 false，这个方法就是判断在等待队列中是否已经有线程在排队了。

这也就是公平锁和非公平锁的核心区别，如果是公平锁，那么一旦已经有线程在排队了，当前线程就不再尝试获取锁；对于非公平锁而言，无论是否已经有线程在排队，都会尝试获取一下锁，获取不到的话，再去排队。这里有一个特例需要我们注意，针对 tryLock() 方法，它不遵守设定的公平原则。

例如，当有线程执行 tryLock() 方法的时候，一旦有线程释放了锁，那么这个正在 tryLock 的线程就能获取到锁，即使设置的是公平锁模式，即使在它之前已经有其他正在等待队列中等待的线程，简单地说就是 tryLock 可以插队。

看它的源码就会发现：

```
public boolean tryLock() {  
    return sync.nonfairTryAcquire(1);  
}
```

这里调用的就是 nonfairTryAcquire()，表明了是不公平的，和锁本身是否是公平锁无关。综上所述，公平锁就是会按照多个线程申请锁的顺序来获取锁，从而实现公平的特性。

非公平锁加锁时不考虑排队等待情况，直接尝试获取锁，所以存在后申请却先获得锁的情况，但由此也提高了整体的效率。

## 什么情况会产生死锁问题？如何解决？

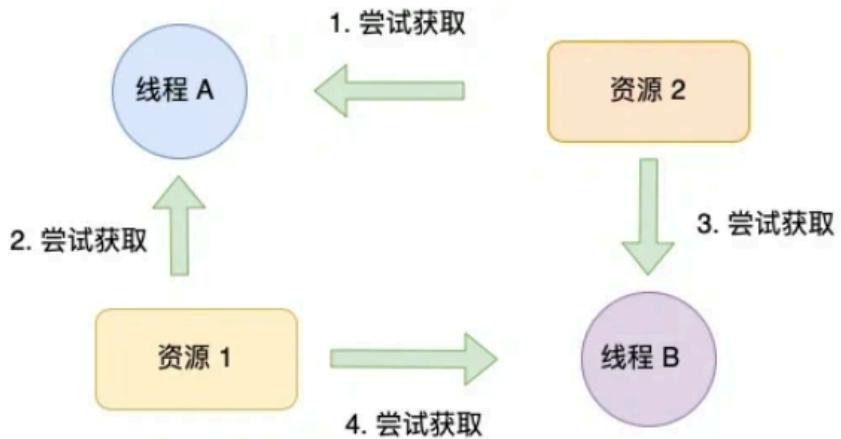
死锁只有**同时满足**以下四个条件才会发生：

- 互斥条件：互斥条件是指**多个线程不能同时使用同一个资源**。
- 持有并等待条件：持有并等待条件是指，当线程 A 已经持有了资源 1，又想申请资源 2，而资源 2 已经被线程 C 持有了，所以线程 A 就会处于等待状态，但是**线程 A 在等待资源 2 的同时并不会释放自己已经持有的资源 1**。
- 不可剥夺条件：不可剥夺条件是指，当线程已经持有了资源，在**自己使用完之前不能被其他线程获取**，线程 B 如果也想使用此资源，则只能在线程 A 使用完并释放后才能获取。
- 环路等待条件：环路等待条件指的是，在死锁发生的时候，**两个线程获取资源的顺序构成了环形链**。

例如，线程 A 持有资源 R1 并试图获取资源 R2，而线程 B 持有资源 R2 并试图获取资源 R1，此时两个线程相互等待对方释放资源，从而导致死锁。

避免死锁问题就只需要破坏其中一个条件就可以，最常见的并且可行的就是**使用资源有序分配法，来破坏环路等待条件。**

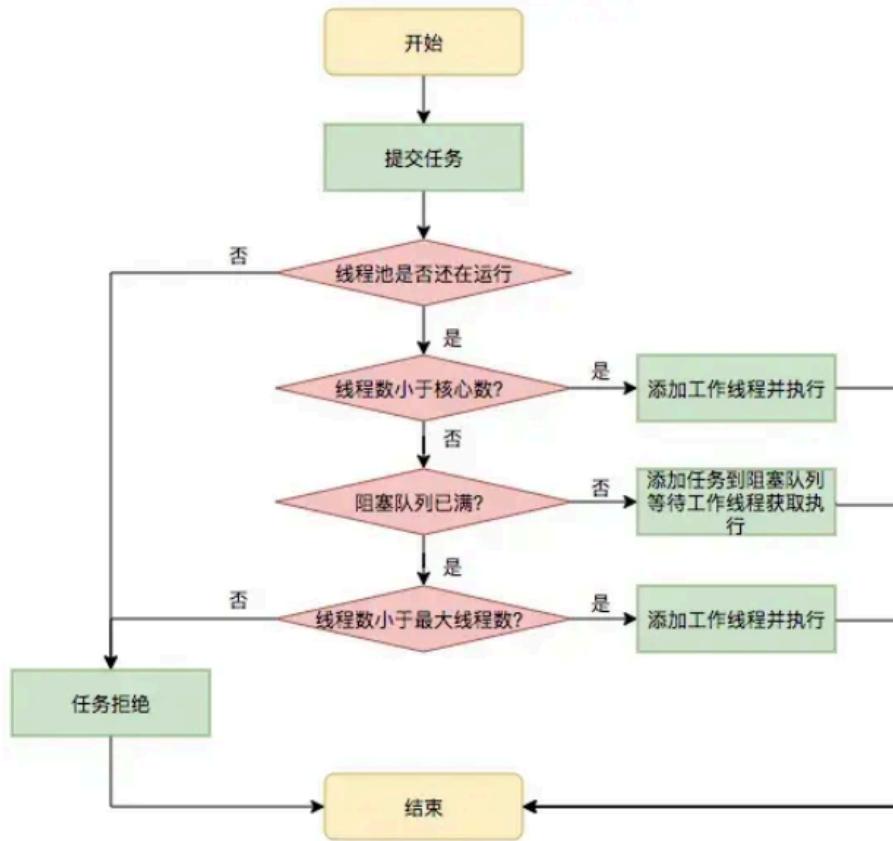
那什么是资源有序分配法呢？线程 A 和 线程 B 获取资源的顺序要一样，当线程 A 是先尝试获取资源 A，然后尝试获取资源 B 的时候，线程 B 同样也是先尝试获取资源 A，然后尝试获取资源 B。也就是说，线程 A 和 线程 B 总是以相同的顺序申请自己想要的资源。



f. 线程池:

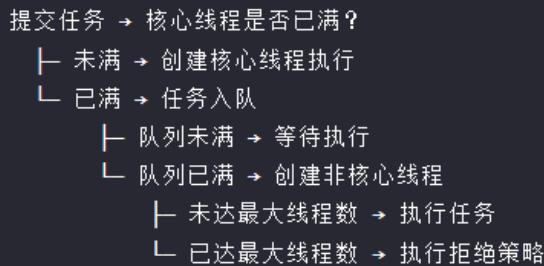
## 介绍一下线程池的工作原理

线程池是为了减少频繁的创建线程和销毁线程带来的性能损耗，线程池的工作原理如下图：



线程池分为核心线程池，线程池的最大容量，还有等待任务的队列，提交一个任务，如果核心线程没有满，就创建一个线程，如果满了，就是会加入等待队列，如果等待队列满了，就会增加线程，如果达到最大线程数量，如果都达到最大线程数量，就会按照一些丢弃的策略进行处理。

任务执行流程如下：



## 线程池的参数有哪些?

线程池的构造函数有7个参数：

```

public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {

```

- **corePoolSize**: 线程池核心线程数量。默认情况下，线程池中线程的数量如果  $\leq \text{corePoolSize}$ ，那么即使这些线程处于空闲状态，那也不会被销毁。
- **maximumPoolSize**: 线程池中最多可容纳的线程数量。当一个新任务交给线程池，如果此时线程池中有空闲的线程，就会直接执行，如果没有空闲的线程且当前线程池的线程数量小于maximumPoolSize，就会创建新的线程来执行任务，否则就会将该任务加入到阻塞队列中，如果阻塞队列满了，就会创建一个新线程，从阻塞队列头部取出一个任务来执行，并将新任务加入到阻塞队列末尾。如果当前线程池中线程的数量等于maximumPoolSize，就不会创建新线程，就会去执行拒绝策略。
- **keepAliveTime**: 当线程池中线程的数量大于corePoolSize，并且某个线程的空闲时间超过了keepAliveTime，那么这个线程就会被销毁。
- **unit**: 就是keepAliveTime时间的单位。
- **workQueue**: 工作队列。当没有空闲的线程执行新任务时，该任务就会被放入工作队列中，等待执行。
- **threadFactory**: 线程工厂。可以用来给线程取名字等等
- **handler**: 拒绝策略。当一个新任务交给线程池，如果此时线程池中有空闲的线程，就会直接执行，如果没有空闲的线程，就会将该任务加入到阻塞队列中，如果阻塞队列满了，就会创建一个新线程，从阻塞队列头部取出一个任务来执行，并将新任务加入到阻塞队列末尾。如果当前线程池中线程的数量等于maximumPoolSize，就不会创建新线程，就会去执行拒绝策略

## 线程池工作队列满了有哪些拒接策略？

当线程池的任务队列满了之后，线程池会执行指定的拒绝策略来应对，常用的四种拒绝策略包括：CallerRunsPolicy、AbortPolicy、DiscardPolicy、DiscardOldestPolicy，此外，还可以通过实现RejectedExecutionHandler接口来自定义拒绝策略。

四种预置的拒绝策略：

- CallerRunsPolicy，使用线程池的调用者所在的线程去执行被拒绝的任务，除非线程池被停止或者线程池的任务队列已有空缺。
- AbortPolicy，直接抛出一个任务被线程池拒绝的异常。
- DiscardPolicy，不做任何处理，静默拒绝提交的任务。
- DiscardOldestPolicy，抛弃最老的任务，然后执行该任务。
- 自定义拒绝策略，通过实现接口可以自定义任务拒绝策略。

## 有线程池参数设置的经验吗？

核心线程数（corePoolSize）设置的经验：

- CPU密集型：corePoolSize = CPU核数 + 1（避免过多线程竞争CPU）
- IO密集型：corePoolSize = CPU核数 × 2（或更高，具体看IO等待时间）

场景一：电商场景，特点瞬时高并发、任务处理时间短，线程池的配置可设置如下：

```
new ThreadPoolExecutor(  
    16, // corePoolSize = 16 (假设8核CPU × 2)  
    32, // maximumPoolSize = 32 (突发流量扩容)  
    10, TimeUnit.SECONDS, // 非核心线程空闲10秒回收  
    new SynchronousQueue<>(), // 不缓存任务，直接扩容线程  
    new AbortPolicy() // 直接拒绝，避免系统过载  
);
```

java

说明：

- 使用 SynchronousQueue 确保任务直达线程，避免队列延迟。
- 拒绝策略快速失败，前端返回“活动火爆”提示，结合降级策略（如缓存预热）。

场景二：后台数据处理服务，特点稳定流量、任务处理时间长（秒级）、允许一定延迟，线程池的配置可设置如下：

```
new ThreadPoolExecutor(  
    8, // corePoolSize = 8 (8核CPU)  
    8, // maximumPoolSize = 8 (禁止扩容，避免资源耗尽)  
    0, TimeUnit.SECONDS, // 不回收线程  
    new ArrayBlockingQueue<>(1000), // 有界队列，容量1000  
    new CallerRunsPolicy() // 队列满后由调用线程执行  
);
```

说明：

- 固定线程数避免资源波动，队列缓冲任务，拒绝策略兜底。
- 配合监控告警（如队列使用率>80%触发扩容）。

场景三：微服务HTTP请求处理，特点IO密集型、依赖下游服务响应时间，线程池的配置可设置如下：

```
new ThreadPoolExecutor(  
    16, // corePoolSize = 16 (8核 * 2)  
    64, // maximumPoolSize = 64 (应对慢下游)  
    60, TimeUnit.SECONDS, // 非核心线程空闲60秒回收  
    new LinkedBlockingQueue<>(200), // 有界队列容量200  
    new CustomRetryPolicy() // 自定义拒绝策略 (重试或降级)  
);
```

说明：

- 根据下游RT（响应时间）调整线程数，队列防止瞬时峰值。
- 自定义拒绝策略将任务暂存Redis，异步重试。

## 核心线程数设置为0可不可以？

可以，当核心线程数为0的时候，会创建一个非核心线程进行执行。

从下面的源码也可以看到，当核心线程数为 0 时，来了一个任务之后，会先将任务添加到任务队列，同时也会判断当前工作的线程数是否为 0，如果为 0，则会创建线程来执行线程池的任务。

```
    */
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, core: true))
            return;
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker( firstTask: null, core: false);
    }
    else if (!addWorker(command, core: false))
        reject(command);
```

## 线程池种类有哪些？

- ScheduledThreadPool：可以设置定期的执行任务，它支持定时或周期性执行任务，比如每隔 10 秒钟执行一次任务，我通过这个实现类设置定期执行任务的策略。
- FixedThreadPool：它的核心线程数和最大线程数是一样的，所以可以把它看作是固定线程数的线程池，它的特点是线程池中的线程数除了初始阶段需要从 0 开始增加外，之后的线程数量就是固定的，就算任务数超过线程数，线程池也不会再创建更多的线程来处理任务，而是会把超出线程处理能力的任务放到任务队列中进行等待。而且就算任务队列满了，到了本该继续增加线程数的时候，由于它的最大线程数和核心线程数是一样的，所以也无法再增加新的线程了。
- CachedThreadPool：可以称作可缓存线程池，它的特点在线程数是几乎可以无限增加的（实际最大可以达到 Integer.MAX\_VALUE，为  $2^{31}-1$ ，这个数非常大，所以基本不可能达到），而当线程闲置时还可以对线程进行回收。也就是说该线程池的线程数量不是固定不变的，当然它也有一个用于存储提交任务的队列，但这个队列是 SynchronousQueue，队列的容量为 0，实际不存储任何任务，它只负责对任务进行中转和传递，所以效率比较高。
- SingleThreadExecutor：它会使用唯一的线程去执行任务，原理和 FixedThreadPool 是一样的，只不过这里线程只有一个，如果线程在执行任务的过程中发生异常，线程池也会重新创建一个线程来执行后续的任务。这种线程池由于只有一个线程，所以非常适合用于所有任务都需要按被提交的顺序依次执行的场景，而前几种线程池不一定能够保障任务的执行顺序等于被提交的顺序，因为它们是多线程并行执行的。
- SingleThreadScheduledExecutor：它实际和 ScheduledThreadPool 线程池非常相似，它只是 ScheduledThreadPool 的一个特例，内部只有一个线程。

## 线程池一般是怎么用的？

Java 中的 Executors 类定义了一些快捷的工具方法，来帮助我们快速创建线程池。《阿里巴巴 Java 开发手册》中提到，禁止使用这些方法来创建线程池，而应该手动 new ThreadPoolExecutor 来创建线程池。这一条规则的背后，是大量血淋淋的生产事故，最典型的就是 newFixedThreadPool 和 newCachedThreadPool，可能因为资源耗尽导致 OOM 问题。

所以，不建议使用 Executors 提供的两种快捷的线程池，原因如下：

- 我们需要根据自己的场景、并发情况来评估线程池的几个核心参数，包括核心线程数、最大线程数、线程回收策略、工作队列的类型，以及拒绝策略，确保线程池的工作行为符合需求，一般都需要设置有界的工作队列和可控的线程数。
- 任何时候，都应该为自定义线程池指定有意义的名称，以方便排查问题。当出现线程数量暴增、线程死锁、线程占用大量 CPU、线程执行出现异常等问题时，我们往往会抓取线程栈。此时，有意义的线程名称，就可以方便我们定位问题。

除了建议手动声明线程池以外，我还建议用一些监控手段来观察线程池的状态。线程池这个组件往往会展现得任劳任怨、默默无闻，除非是出现了拒绝策略，否则压力再大都不会抛出一个异常。如果我们能提前观察到线程池队列的积压，或者线程数量的快速膨胀，往往可以提早发现并解决问题。

## 线程池中shutdown(), shutdownNow()这两个方法有什么作用？

从下面的源码【高亮】注释可以很清晰的看出两者的区别：

- shutdown使用了以后会置状态为SHUTDOWN，正在执行的任务会继续执行下去，没有被执行的则中断。此时，则不能再往线程池中添加任何任务，否则将会抛出 RejectedExecutionException 异常
- 而 shutdownNow 为STOP，并试图停止所有正在执行的线程，不再处理还在池队列中等待的任务，当然，它会返回那些未执行的任务。它试图终止线程的方法是通过调用 Thread.interrupt() 方法来实现的，但是这种方法的作用有限，如果线程中没有sleep、wait、Condition、定时锁等应用，interrupt()方法是无法中断当前的线程的。所以，ShutdownNow()并不代表线程池就一定立即就能退出，它可能必须要等待所有正在执行的任务都执行完成了才能退出。

[shutdown 源码：](#)

```
public void shutdown() {  
    final ReentrantLock mainLock = this.mainLock;  
    mainLock.lock();  
    try {  
        checkShutdownAccess();  
        // 高亮  
        advanceRunState(SHUTDOWN);  
        interruptIdleWorkers();  
        onShutdown();  
    } finally {  
        mainLock.unlock();  
    }  
    tryTerminate();  
}
```

java

shutdownNow 源码：

```
public List<Runnable> shutdownNow() {  
    List<Runnable> tasks;  
    final ReentrantLock mainLock = this.mainLock;  
    mainLock.lock();  
    try {  
        checkShutdownAccess();  
        // 高亮  
        advanceRunState(STOP);  
        interruptWorkers();  
        // 高亮  
        tasks = drainQueue();  
    } finally {  
        mainLock.unlock();  
    }  
    tryTerminate();  
    // 高亮  
    return tasks;  
}
```

java

## 提交给线程池中的任务可以被撤回吗？

可以，当向线程池提交任务时，会得到一个 `Future` 对象。这个 `Future` 对象提供了几种方法来管理任务的执行，包括取消任务。

取消任务的主要方法是 `Future` 接口中的 `cancel(boolean mayInterruptIfRunning)` 方法。这个方法尝试取消执行的任务。参数 `mayInterruptIfRunning` 指示是否允许中断正在执行的任务。如果设置为 `true`，则表示如果任务已经开始执行，那么允许中断任务；如果设置为 `false`，任务已经开始执行则不会被中断。

```
public interface Future<V> {  
    // 是否取消线程的执行  
    boolean cancel(boolean mayInterruptIfRunning);  
    // 线程是否被取消  
    boolean isCancelled();  
    // 线程是否执行完毕  
    boolean isDone();  
    // 立即获得线程返回的结果  
    V get() throws InterruptedException, ExecutionException;  
    // 延时时间后再获得线程返回的结果  
    V get(long timeout, TimeUnit unit)  
        throws InterruptedException, ExecutionException, TimeoutException;  
}
```

java

取消线程池中任务的方式，代码如下，通过 `future` 对象的 `cancel(boolean)` 函数来定向取消特定的任务。

```
public static void main(String[] args) {
    ExecutorService service = Executors.newSingleThreadExecutor();
    Future future = service.submit(new TheradDemo());

    try {
        // 可能抛出异常
        future.get();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    } finally {
        //终止任务的执行
        future.cancel(true);
    }
}
```

java

g. 场景：

### 单例模型既然已经用了synchronized，为什么还要在加volatile？

使用 `synchronized` 和 `volatile` 一起，可以创建一个既线程安全又能正确初始化的单例模式，避免了多线程环境下的各种潜在问题。这是一种比较完善的线程安全的单例模式实现方式，尤其适用于高并发环境。

```
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

java

`synchronized` 关键字的作用用于确保在多线程环境下，只有一个线程能够进入同步块（这里是 `synchronized (Singleton.class)`）。在创建单例对象时，通过 `synchronized` 保证了创建过程的线程安全性，避免多个线程同时创建多个单例对象。

`volatile` 确保了对象引用的可见性和创建过程的有序性，避免了由于指令重排序而导致的错误。

`instance = new Singleton();` 这行代码并不是一个原子操作，它实际上可以分解为以下几个步骤：

- 分配内存空间。
- 实例化对象。
- 将对象引用赋值给 `instance`。

由于 Java 内存模型允许编译器和处理器对指令进行重排序，在没有 `volatile` 的情况下，可能会出现重排序，例如先将对象引用赋值给 `instance`，但对象的实例化操作尚未完成。

这样，其他线程在检查 `instance == null` 时，会认为单例已经创建，从而得到一个未完全初始化的对象，导致错误。

`volatile` 可以保证变量的可见性和禁止指令重排序。它确保对 `instance` 的修改对所有线程都是可见的，并且保证了上述三个步骤按顺序执行，避免了在单例创建过程中因指令重排序而导致的问题。

## 1. 几个概念：

### 1. 线程与进程：

进程：程序的一次执行过程，产生、存在、消亡，是动态过程

线程：由进程创造，是其一个实体（一个进程可以有多个线程）

### 2. 单线程与多线程：

单线程：顾名思义，同一时刻只允许执行一个线程

多线程：同理，同一时间可执行多线程（例如同时下载文件）

### 3. 并发与并行：

并发：同一时刻多个任务交替执行

并行：同一时刻多个任务同时执行

## 2. 创建线程（继承 Thread 类或实现 Runnable 接口）：

### a. Thread：

```
public static void main(String[] args) {
    Darling lover = new Darling();
    lover.start(); //创建新线程,此时有一个lover线程和主线程
    System.out.println("主线程继续执行" + Thread.currentThread().getName()); //名字 main
    for(int i = 0; i < 60; i++) {
        System.out.println("主线程 i=" + i);
        try {
            //这里都需要try检查一下
            Thread.sleep(1000);
        }catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Darling extends Thread {
    int times = 0;
    public void run(){
        while(true){
            try{
                Thread.sleep(1000);
            }catch(InterruptedException e){
                e.printStackTrace();
            }
            System.out.println("I love hxq ~"+(++times));
            if(times == 65){
                break;
            }
        }
    }
}
```

a. Runnable(多线程共享资源):

```

public static void main(String[] args) {
    T t = new T();
    Thread t1 = new Thread(t);
    Thread t2 = new Thread(t);
    t1.start();
    t2.start();
}

class T implements Runnable {
    private static double rest = 10000;
    @Override
    public void run(){
        while(true) {
            //一次只能有一个线程持有该锁
            synchronized (this) {
                if(rest<1000){
                    System.out.println("余额不足");
                    break;
                }
                rest -= 1000;
                System.out.println(rest+" -1000");
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

### 3. 继承Thread类/实现Runnable接口区别

- a. 本质上没有区别,Thread类实现了Runnable接口
- b. 实现Runnable接口适合多个线程共享同一资源(且避免的单线程的限制)

### 特殊章节: 为什么是start():

start()是启动一个新的线程运行run()

run()只是一个普通的方法

## 4. 线程方法:

1. **setName**:设置线程名称

2. **getName**:返回线程名称

3. **start**:执行线程(Java虚拟机底层调用线程的start0方法)

4. **run**:调用线程对象run方法

5. **setPriority**:更改线程优先级

6. **getPriority**:获取线程优先级

7. **sleep**:让正在执行的线程暂停执行一段时间

8. **interrupt**:中断线程

(没有结束进程,所以一般用于中断正在休眠的线程)

9. **yield**:线程礼让cpu,让其他线程执行(时间不定,所以不一定成功)

不会释放锁或同步资源

10. **join**:线程的插队,优先完成插入线程任务

会释放当前进程的锁

## 5. 用户线程和守护线程:

用户线程(工作线程):由用户程序创建的线程,任务执行完或通知方式结束

守护线程:一般为工作线程服务,所有用户线程结束,守护线程自动结束(比如垃圾回收机制也是一种守护线程)

## 使用守护线程:

```
Thread daemonThread = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        ...//守护线程执行的代码  
    }  
});  
daemonThread.setDaemon(true); // 将线程设置为守护线程  
daemonThread.start();  
try {  
    Thread.sleep(5000); // 主线程暂停5秒,即空转运行5秒主程序  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}  
// 主线程结束,守护线程结束  
System.out.println("主线程结束, 守护线程结束");
```

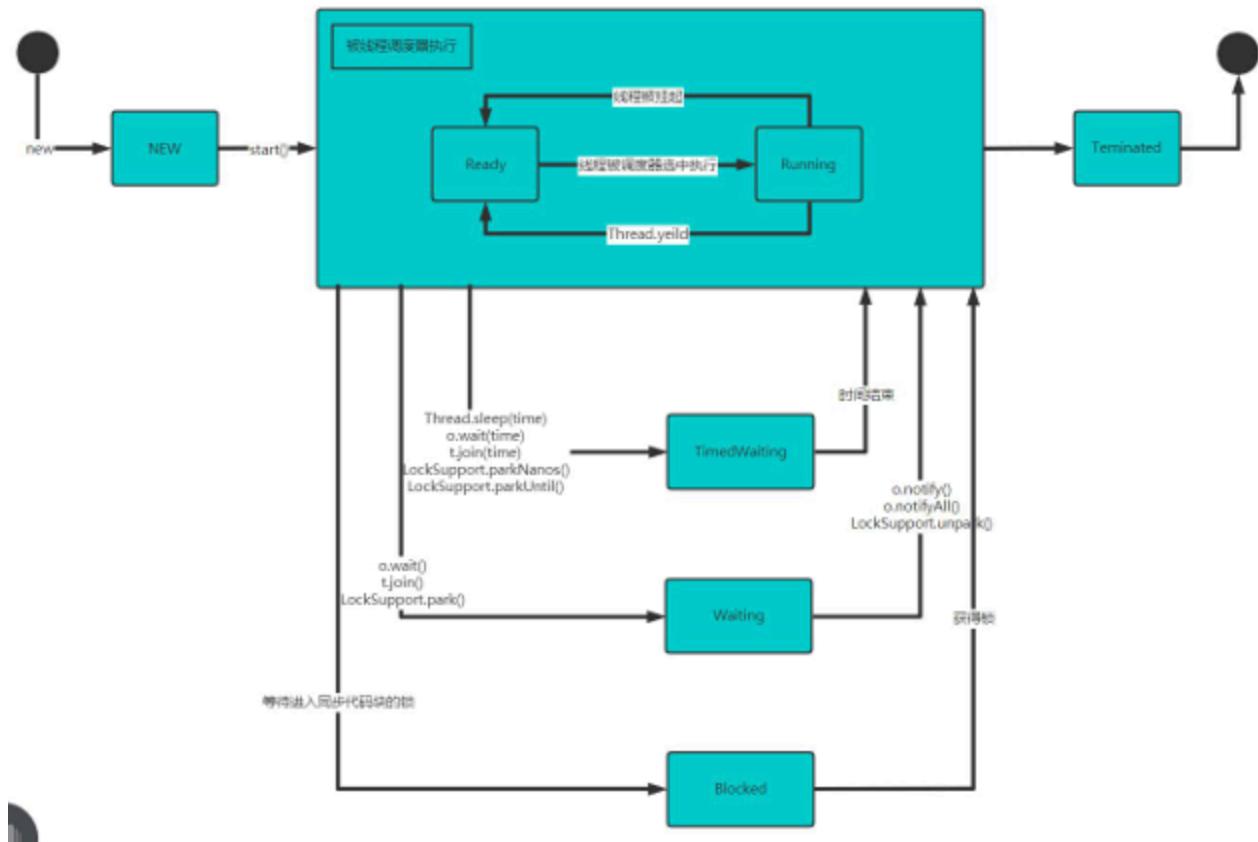
## 6. 线程的生命周期:

### 1. 线程状态总览图:

线程状态。线程可以处于以下状态之一：

- NEW  
尚未启动的线程处于此状态。
- RUNNABLE  
在Java虚拟机中执行的线程处于此状态。
- BLOCKED  
被阻塞等待监视器锁定的线程处于此状态。
- WAITING  
正在等待另一个线程执行特定动作的线程处于此状态。
- TIMED\_WAITING  
正在等待另一个线程执行动作达到指定等待时间的线程处于此状态。
- TERMINATED  
已退出的线程处于此状态。

## 2. 线程状态转换图:



## 3. 查看线程状态:

```
System.out.println("状态:"+t.getState());
```

## 7. 线程的同步(多线程与锁):

### 1. 线程的同步机制:

保证数据在同一时刻最多只有一个线程访问,保护数据完整性

### 2. 具体同步方法(Synchronized):

由Synchronized修饰的对象,任意时刻只能由一个线程访问

#### a. 同步代码块:

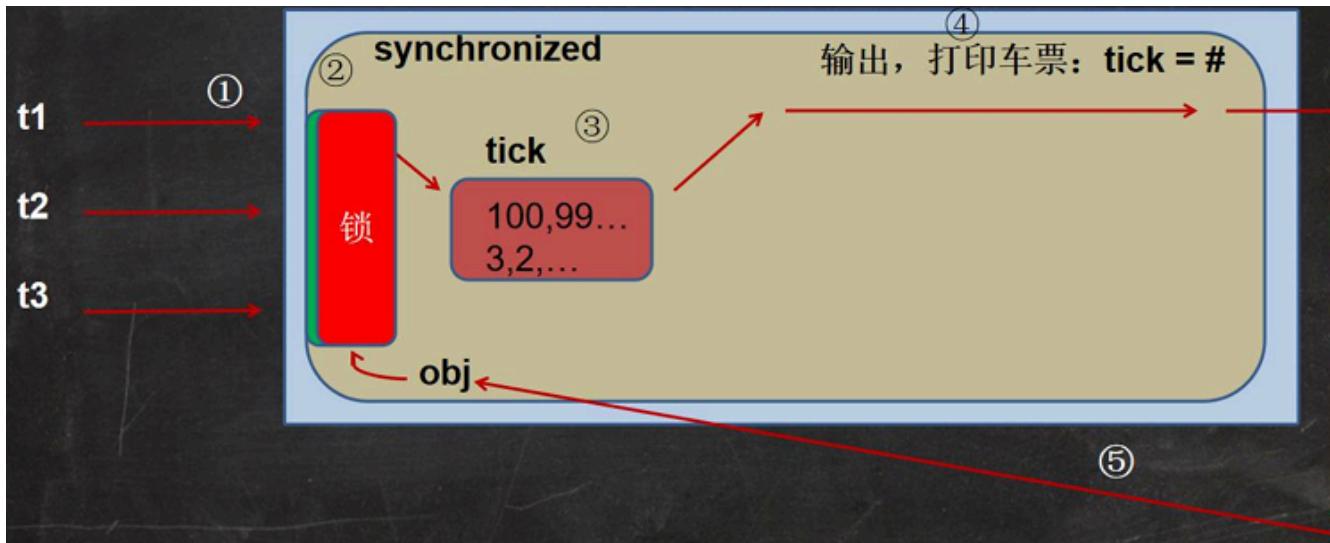
```
synchronized(对象){  
    //获得对象的锁才能操作同步代码  
    //需要被同步的代码  
}
```

#### b. 同步方法:

```
public synchronized void m (String name){
```

```
//需要被同步的代码  
}
```

### 3. 同步原理:



### 4. 互斥锁:

- Synchronized与互斥锁联系,由Synchronized修饰的对象,任意时刻只能由一个线程访问
- 同步的局限性:程序执行效率降低
- 同步方法(非静态)的锁默认是this(当前对象,用于同一对象的不同线程,需要同一个锁),也可以是其他对象(同一对象)(也跟对象与非静态方法对应相联系,每个使用这个方法的实例对象都有自己的锁)
- 同步方法(静态)的锁对象默认是当前类本身(还是跟类与静态方法的联系相关,类对应静态方法,所以所有使用该静态方法的对象都会使用/竞争同一个锁)
- 使用示例:

```
class T implements Runnable{
    private int ticketNum = 100;//让多个线程共享 ticketNum
    private boolean loop = true;//控制 run 方法变量
    Object object = new Object();
    //静态方法在方法上修饰
    public synchronized static void r1() {
        System.out.println("m1");
    }
    //静态方法在代码块上修饰
    public static void r2() {
        synchronized (T.class) {
            System.out.println("m2");
        }
    }
    //非静态方法在方法上修饰
    public synchronized void r3() {
        if (ticketNum <= 0) {
            System.out.println("结束...");
            loop = false;
            return;
        }
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    //非静态方法在代码块上修饰
    public void r4() {
        //此处object为类内定义的Object属性
        synchronized (object) {
            if (ticketNum <= 0) {
                System.out.println("结束...");
                loop = false;
                return;
            }
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
@Override  
public void run() {  
    while (loop) {  
        r4();  
    }  
}
```

## 5. 死锁(需避免,多线程互相占对方的锁资源且不相让导致的死结):

- a. 线程可以**按任意顺序获得多个锁**
- b. 获得下一个锁需要**先释放之前持有的所有锁**
  - a. 尝试获取锁
  - b. 若被阻塞,会进入等待队列
  - c. 持有该锁的线程释放锁
  - d. 等待队列中的一个线程唤醒锁
  - e. 该线程重新尝试获取锁
  - f. 执行同步代码
  - g. 执行完毕,释放锁
- c. 一个锁在任意时刻**只能被一个线程持有**
- d. 死锁的发生:A有锁1,B有锁2,A要获取锁2,B要获取锁1,由上述过程可得,形成死循环/死锁(比如说A尝试获取锁2,被阻塞,无法释放锁1,于是B无法获取锁1,进入阻塞,无法释放锁2,所有A无法获取锁2,被阻塞...)

## 6. 释放锁:

- a. 可以释放锁:
  - 1. **当前线程的同步方法、同步代码块执行结束**  
**案例: 上厕所, 完事出来**
  - 2. **当前线程在同步代码块、同步方法中遇到break、return。**  
**案例: 没有正常的完事, 经理叫他修改bug, 不得已出来**
  - 3. **当前线程在同步代码块、同步方法中出现了未处理的Error或Exception, 导致异常**  
**案例: 没有正常的完事, 发现忘带纸, 不得已出来**
  - 4. **当前线程在同步代码块、同步方法中执行了线程对象的wait()方法, 当前线程暂停, 放锁。**  
**案例: 没有正常完事, 觉得需要酝酿下, 所以出来等会再进去**
- b. 无法释放锁:

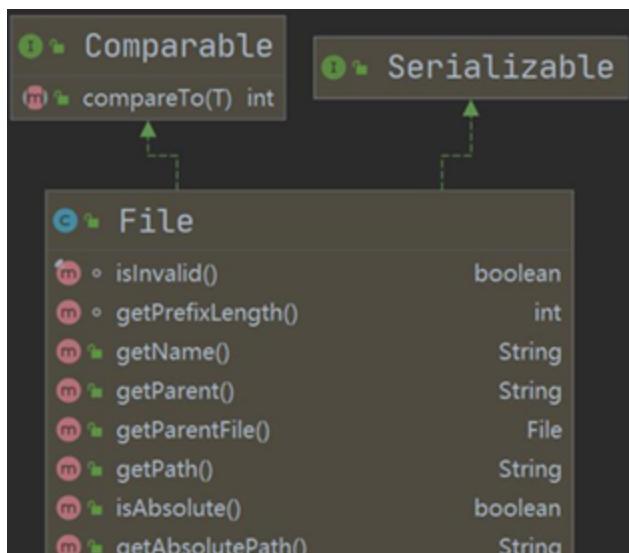
- 线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行，不会释放锁  
案例：上厕所，太困了，在坑位上眯了一会
- 线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁。  
提示：应尽量避免使用suspend()和resume()来控制线程，方法不再推荐使用

## 3.10 IO流(Input/Output):

### 1. 文件(File)与流：

#### 1. 概念(四个抽象类)

a. 文件：



- b. 流：数据在数据源(文件)和程序(内存)之间的路径  
c. 输入流：数据源(文件)到程序(内存)的路径  
d. 输出流：程序(内存)到数据源(文件)的路径  
e. 抽象类：InputStream(字节输入流), OutputStream(字节输出流), Reader(字符输入流), Writer(字符输出流)

按操作数据单位不同分为：字节流(8 bit) 二进制文件，字符流(按字符) 文本文件

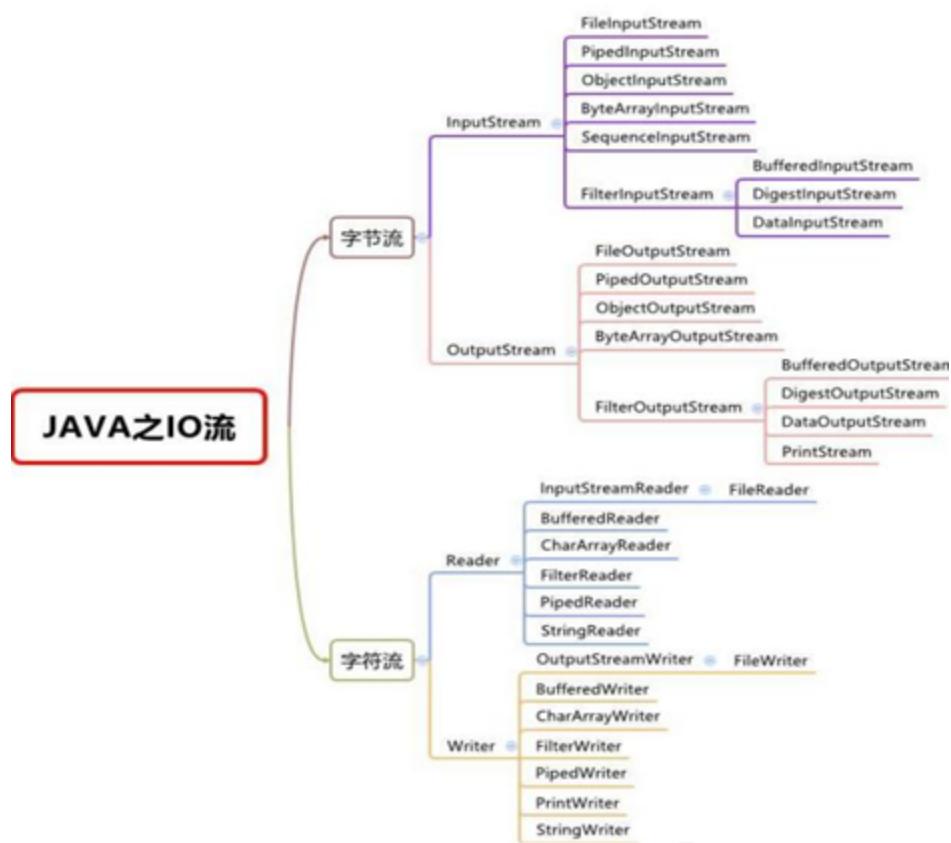
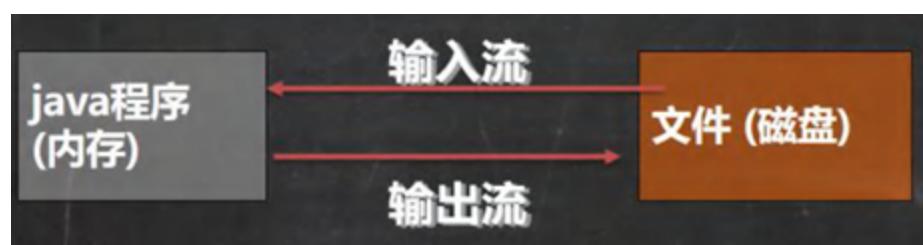
按数据流的流向不同分为：输入流，输出流

按流的角色的不同分为：节点流，处理流/包装流

| (抽象基类) | 字节流          | 字符流    |
|--------|--------------|--------|
| 输入流    | InputStream  | Reader |
| 输出流    | OutputStream | Writer |

- 1) Java的IO流共涉及40多个类，实际上非常规则，都是从如上4个抽象基类派生的。
- 2) 由这四个类派生出来的子类名称都是以其父类名作为子类名后缀。

## 2. IO流体系图：



## 2. 常见文件操作(java中目录也是一种文件):

file可以是文件(路径),也可以是目录()路径

```
a. File file = new File(String pathname)
    //根据路径创建一个File对象
b. File file = new File(File parent, String child)
    //根据父目录文件+子路径构建File对象
c. File file = new File(String parent, String child)
    //根据父目录+子路径构建File对象
d. 文件名.createNewFile();
    //真的创建一个新文件,否则仅仅是File对象
try {
    file.createNewFile();
    System.out.println("文件创建成功")
} catch (IOException e) {
    e.printStackTrace();
}
e. System.out.println("文件名字=" + file.getName());
    //getName、getAbsolutePath、getParent、length、exists、isFile、isDirectory
    System.out.println("文件绝对路径=" + file.getAbsolutePath());
    System.out.println("文件父级目录=" + file.getParent());
    System.out.println("文件大小(字节)=" + file.length());
    System.out.println("文件是否存在=" + file.exists());//T
    System.out.println("是不是一个文件=" + file.isFile());//T
    System.out.println("是不是一个目录=" + file.isDirectory());//F
f. file.mkdir();//Boolean
    //创建一级目录
g. file.mkdirs();//Boolean
    //创建多级目录
h. file.delete();//Boolean
    //删除空目录或文件
```

### 3. FileInputStream(字节输入流,InputStream的实现类)

#### 1. 介绍:

- a. 是InputStream的实现类,一种字节输入流
- b. 是处理二进制数据的基础,以字节为单位进行读写

#### 2. 使用案例:

```
String filePath = "e:\\hello.txt";
//字节数组,一次读取8个字节
//字节数组可以减少调用次数,增加读取/写入效率
byte[] buf = new byte[8];
int readLen = 0;

//try中创建的算局部变量,所以在try之外创建
FileInputStream fileInputStream = null;

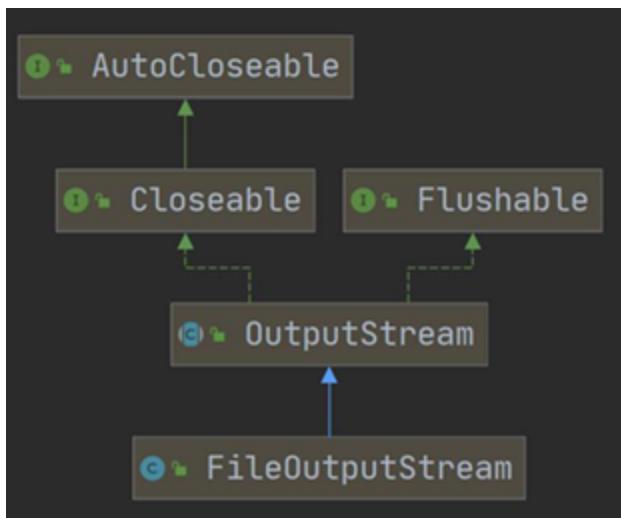
try {
    //创建FileInputStream对象, 用于读取文件
    fileInputStream = new FileInputStream(filePath);

    //输入流读取最多buf.length字节的数据到字节数组
    //若无可用输入,此方法将阻塞, 直到某些输入可用
    //若返回-1, 表示读取完毕
    //若读取正常, 返回实际读取的字节数
    while ((readLen = fileInputStream.read(buf)) != -1) {
        System.out.print(new String(buf, 0, readLen));
    }

} catch (IOException e) {
    e.printStackTrace();
} finally {
    //关闭文件流,释放资源
    try {
        fileInputStream.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## 4. FileOutputStream(字节输出流, OutputStream的实现类)

### 1. 介绍:



## 2. 使用案例:

```
String filePath = "e:\\a.txt";
//创建FileOutputStream对象
FileOutputStream fileOutputStream = null;
try {
    //new FileOutputStream(filePath) 创建方式，写入内容会清空并覆盖原来的内容/创建文件
    //new FileOutputStream(filePath, true) 创建方式，写入内容追加到文件后面/创建文件
    fileOutputStream = new FileOutputStream(filePath, true);

    //写入一个字节
    //fileOutputStream.write('H');//


    //写入字符串
    String str = "love hxq forever~";
    //str.getBytes() 可以把 字符串-> 字节数组
    //write(byte[] b, int off, int len) 将 len 字节从位于偏移量off(起始位置)写入此文件输出;
    //创建字符数组b,使用该方法可以减少输出流调用次数,增加写入效率
    //因为每次调用write都会执行一次I/O操作
    fileOutputStream.write(str.getBytes(), 0, 3);

} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        fileOutputStream.close();
    }catch(IOException){
        e.printStackTrace();
    }
}
```

# 同时使用FileInputStream, FileOutputStream

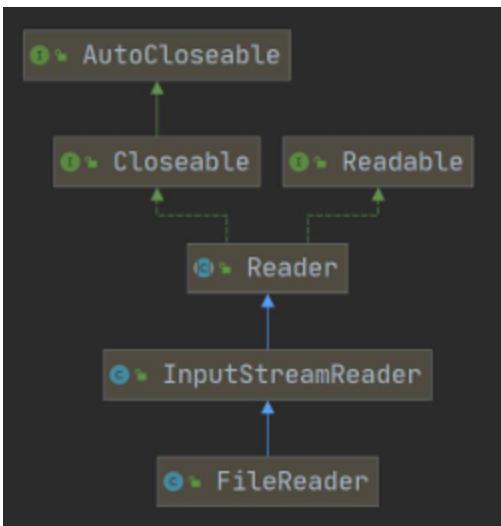
```
try {
    FileInputStream = new FileInputStream(srcFilePath);
    FileOutputStream = new FileOutputStream(destFilePath);
    //定义一个字节数组,提高读取效果
    byte[] buf = new byte[1024];
    int readLen = 0;
    while ((readLen = fileInputStream.read(buf)) != -1) {
        //读取到一部分后,就写入到文件,边读边写
        FileOutputStream.write(buf, 0, readLen); //一定要使用这个方法
    }

} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        //关闭输入流和输出流, 释放资源
        if (fileInputStream != null) {
            fileInputStream.close();
        }
        if (fileOutputStream != null) {
            fileOutputStream.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## 5. FileReader和 FileWriter

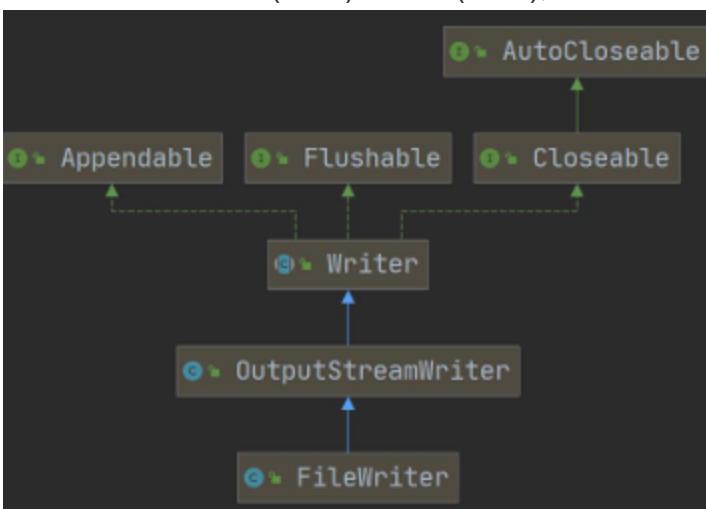
### 1. 介绍:

- a. FileReader:



### b. `FileWriter`:

使用后必须要`close(关闭)`或`flush(刷新)`,否则写入不到指定文件



## 2. 相关方法:

### 1. `FileReader`:

- 1) `new FileReader(File/String)`
- 2) `read`:每次读取单个字符, 返回该字符, 如果到文件末尾返回-1
- 3) `read(char[])`: 批量读取多个字符到数组, 返回读取到的字符数, 如果到文件末尾返回相关API:
  - 1) `new String(char[])`: 将`char[]`转换成`String`
  - 2) `new String(char[], off, len)`: 将`char[]`的指定部分转换成`String`

## 2. FileWriter:

- 1) new FileWriter(File/String): 覆盖模式, 相当于流的指针在首端
- 2) new FileWriter(File/String,true): 追加模式, 相当于流的指针在尾端
- 3) write(int):写入单个字符
- 4) write(char[]):写入指定数组
- 5) write(char[],off,len):写入指定数组的指定部分
- 6) write (string) : 写入整个字符串
- 7) write(string,off,len):写入字符串的指定部分

相关API: String类: toCharArray:将String转换成char[]

➤ 注意:

FileWriter使用后, 必须要关闭(close)或刷新(flush), 否则写入不到指定的文件!

## 3. 使用案例:

### 1. 单个字符读取文件:

```
String filePath = "e:\\story.txt";
//创建FileReader 对象
FileReader fileReader = null;

//data储存ASCII值
int data = 0;

try {
    fileReader = new FileReader(filePath);

    //使用read循环读取单个字符
    while ((data = fileReader.read()) != -1) {
        System.out.print((char) data);
    }

} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (fileReader != null) {
            fileReader.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## 2. 字符数组读取文件:

```
System.out.println("~~~readFile02 ~~~");
String filePath = "e:\\story.txt";
//创建FileReader 对象
FileReader fileReader = null;

//储存每次读取的字符数
int readLen = 0;

//字符数组作缓冲区
char[] buf = new char[8];
try {
    fileReader = new FileReader(filePath);

    //循环读取 使用read(buf)读取到缓冲区， 返回实际读取到的字符数
    //如果返回-1， 说明到文件结束
    while ((readLen = fileReader.read(buf)) != -1) {
        System.out.print(new String(buf, 0, readLen));
    }

} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (fileReader != null) {
            fileReader.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

### 3. FileWriter写入文件:

```
String filePath = "e:\\note.txt";

//创建FileWriter 对象
FileWriter fileWriter = null;

//字符数组包含要写入的字符
char[] chars = {'a', 'b', 'c'};
try {
    //默认覆盖写入
    fileWriter = new FileWriter(filePath);

    //write(int):写入单个字符
    fileWriter.write('H');

    //write(char[]):写入字符数组
    fileWriter.write(chars);

    //write(char[],off,len):写入字符数组的指定部分
    fileWriter.write("韩顺平教育".toCharArray(), 0, 3);

    //write(string):写入整个字符串
    fileWriter.write("I love hxq forever~");

    //write(string,off,len):写入字符串的指定部分
    fileWriter.write("上海天津", 0, 2);
    //数据量大时,可使用循环操作

} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        //close:,刷新(flush)缓冲区并关闭流
        //数据会先写入缓冲区,减少磁盘访问次数,提高效率
        //flush(刷新)会将缓冲区所有数据强制写入文件,不必等待缓冲区填满
        //所以不进行flush(close包含flush)的话可能不会写入到文件中
        fileWriter.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## 3.11 节点流和处理流:

### 1. 理解:

#### 1. 概念:

节点流:从一个特定的数据源读写数据

节点流可以从一个特定的数据源读写数据,如FileReader、FileWriter [源码]



处理流(包装流):"连接"已存在的流之上,提供更强大的读写功能

处理流(也叫包装流)是“连接”在已存在的流(节点流或处理流)之上,为程序提供更为强大的读写功能,也更加灵活,如BufferedReader、BufferedWriter [源码]



### 2. 总览图:

| 分 类   | 字节输入流                              | 字节输出流                               | 字符输入流                           | 字符输出流                            |
|-------|------------------------------------|-------------------------------------|---------------------------------|----------------------------------|
| 抽象基类  | <i>InputStream</i>                 | <i>OutputStream</i>                 | <i>Reader</i>                   | <i>Writer</i>                    |
| 访问文件  | <b><i>FileInputStream</i></b>      | <b><i>FileOutputStream</i></b>      | <b><i>FileReader</i></b>        | <b><i>FileWriter</i></b>         |
| 访问数组  | <b><i>ByteArrayInputStream</i></b> | <b><i>ByteArrayOutputStream</i></b> | <b><i>CharArrayReader</i></b>   | <b><i>CharArrayWriter</i></b>    |
| 访问管道  | <b><i>PipedInputStream</i></b>     | <b><i>PipedOutputStream</i></b>     | <b><i>PipedReader</i></b>       | <b><i>PipedWriter</i></b>        |
| 访问字符串 |                                    |                                     | <b><i>StringReader</i></b>      | <b><i>StringWriter</i></b>       |
| 缓冲流   | <b><i>BufferedInputStream</i></b>  | <b><i>BufferedOutputStream</i></b>  | <b><i>BufferedReader</i></b>    | <b><i>BufferedWriter</i></b>     |
| 转换流   |                                    |                                     | <b><i>InputStreamReader</i></b> | <b><i>OutputStreamWriter</i></b> |
| 对象流   | <b><i>ObjectInputStream</i></b>    | <b><i>ObjectOutputStream</i></b>    |                                 |                                  |
| 抽象基类  | <i>FilterInputStream</i>           | <i>FilterOutputStream</i>           | <b><i>FilterReader</i></b>      | <b><i>FilterWriter</i></b>       |
| 打印流   |                                    | <b><i>PrintStream</i></b>           |                                 | <b><i>PrintWriter</i></b>        |
| 推回输入流 | <b><i>PushbackInputStream</i></b>  |                                     | <b><i>PushbackReader</i></b>    |                                  |
| 特殊流   | <b><i>DataInputStream</i></b>      | <b><i>DataOutputStream</i></b>      |                                 |                                  |

节点流  
处理流

### 2. 区别和联系:

- 节点流是底层流/低级流,直接与数据源相接
- 处理流(包装流)包装节点流,既消除不同节点流的实现差异,又提供了更方便的方法完成输入输出
- 处理流(包装流)对节点流进行包装,为修饰器设计模式,不会直接与数据源相连

### 3. 处理流的功能:

- a. 性能提高: **增加缓冲** 提高输入输出效率
- b. 操作便捷: 提供一系列方法**一次输入输出大批数据**,使其更灵活方便

### 4. 处理流的类:

#### 1. 字符流**BufferedReader**和**BufferedWriter**

##### 1. BufferedReader读取文本文件案例

```
String filePath="c:\\lover.java";
//创建bufferedReader
BufferedReader bufferedReader = new BufferedReader(new FileReader(filePath));

String line; //按行读取，效率高
//bufferedReader.readLine() 是按行读取文件
//返回null时,表示文件读取完毕
while ((line = bufferedReader.readLine()) != null) {
    System.out.println(line);
}

//关闭处理流:只需要关闭BufferedReader,底层会自动关闭节点流
bufferedReader.close();
```

##### 2. BufferedWriter写入文件案例

```
String filePath="c:\\lover.txt";
//创建BufferedWriter
//newFileWriter(filePath,true)表示以追加的方式写入
//newFileWriter(filePath),表示以覆盖的方式写入
BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(filePath));

//newLine():插入一个和系统相关的换行
bufferedWriter.write("Darling,I love you forever~");
bufferedWriter.newLine();

//关闭外层处理流即可,传入的new FileWriter(filePath),会在底层关闭
bufferedWriter.close();
```

### 3. 文本文件拷贝案例:

```
//BufferedReader 和 BufferedWriter 是安装字符操作
//字符操作二进制文件[声音,视频,doc, pdf], 可能造成文件损坏
//所以操作二进制文件应使用字节操作:BufferedInputStream,BufferedOutputStream
String srcFilePath = "c:\\a.java";
String destFilePath = "c:\\a2.java";
String srcFilePath = "c:\\Our video.avi";
String destFilePath = "c:\\Our video2.avi";

//用于读取和写入
BufferedReader br = null;
BufferedWriter bw = null;

//存储读取的每一行文本
String line;
try {
    br = new BufferedReader(new FileReader(srcFilePath));
    bw =new BufferedWriter(new FileWriter(destFilePath));

    //readLine仅仅读取一行内容,不包括换行
    while ((line = br.readLine()) != null) {
        //边读边写,并在每一行后插入换行
        bw.write(line);
        bw.newLine();
    }
    System.out.println("拷贝完毕...");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    //关闭流
    try {
        if(br != null) {
            br.close();
        }
        if(bw != null) {
            bw.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

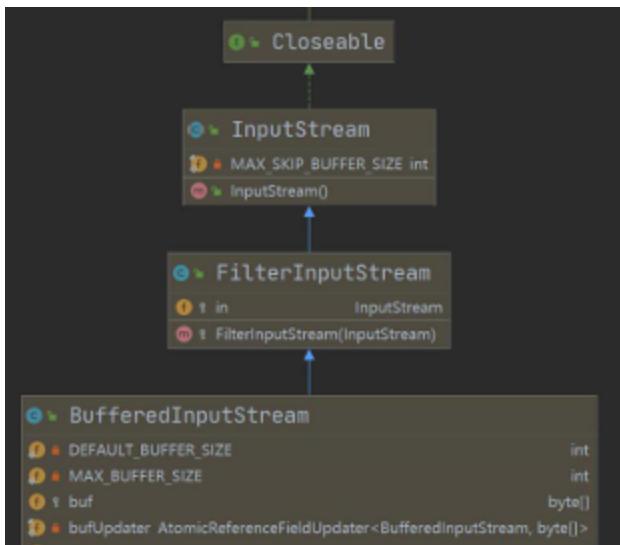
## 2. 字节流BufferedInputStream和BufferedOutputStream

### 1. 介绍:

a. BufferedInputStream(字节流):



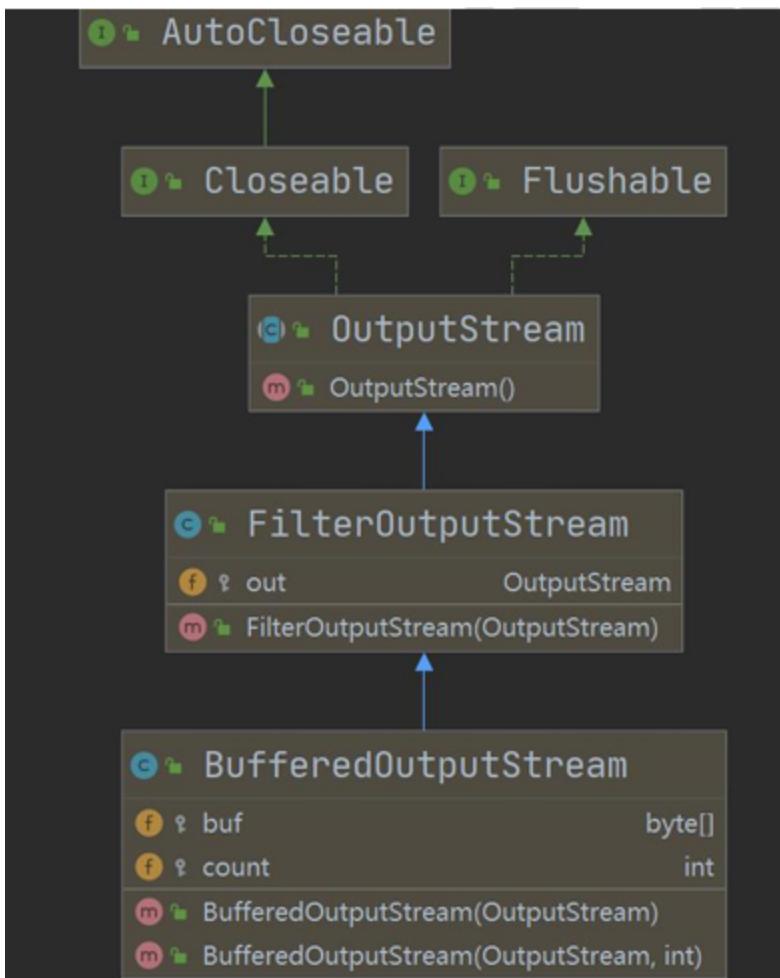
**BufferedInputStream是字节流**  
**， 在创建 BufferedInputStream 时，会创建一个内部缓冲区数组.**



b. BufferedOutputStream(字节流):



**BufferedOutputStream是字节流**  
**， 实现缓冲的输出流，可以将**  
**多个字节写入底层输出流中，而**  
**不必对每次字节写入调用底层**  
**系统**



## 2. 图片/音乐文件拷贝案例:

```
String srcFilePath = "c:\\we.jpg";
String destFilePath = "c:\\our love.jpg";
String srcFilePath = "c:\\lover.avi";
String destFilePath = "c:\\lover2.avi";
String srcFilePath = "c:\\a.java";
String destFilePath = "c:\\a2.java";

//创建读取和写入二进制文件对象
BufferedInputStream bis = null;
BufferedOutputStream bos = null;
try {
    bis = new BufferedInputStream(new FileInputStream(srcFilePath));
    bos = new BufferedOutputStream(new FileOutputStream(destFilePath));

    //储存读取的字节
    byte[] buff = new byte[1024];
    int readLen = 0;

    //循环读取文件
    //当返回-1 时，就表示文件读取完毕
    while ((readLen = bis.read(buff)) != -1) {
        bos.write(buff, 0, readLen);
    }
    System.out.println("文件拷贝完毕~~~");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    //关闭流，关闭外层的处理流即可，底层会去关闭节点流
    try {
        if(bis != null) {
            bis.close();
        }
        if(bos != null) {
            bos.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## 特殊章节:序列化和反序列化:

序列化:保存数据时,保存其值和数据类型

反序列化:恢复数据时,恢复其值和数据类型

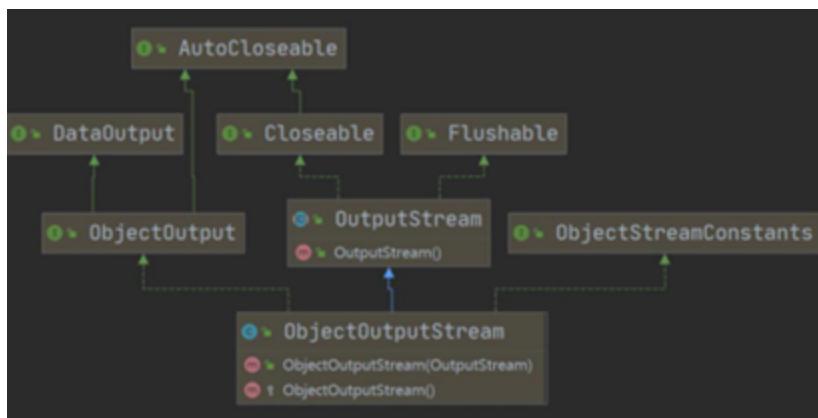
类实现接口Serializable或Externalizable->类可序列化->对象支持序列化机制



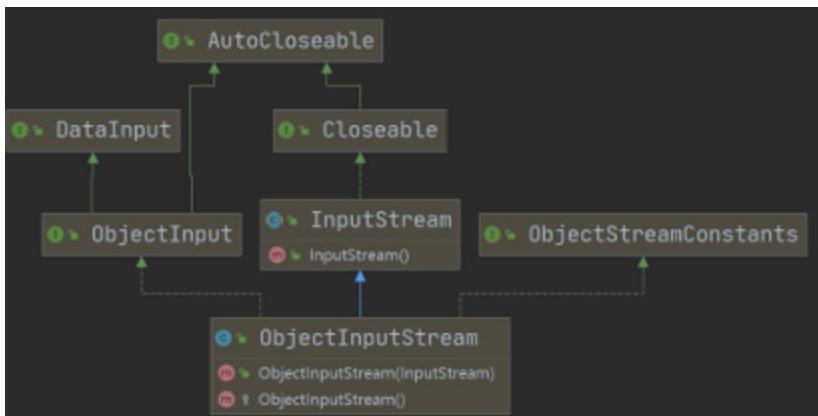
### 3. 对象流(处理流)ObjectInputStream和ObjectOutputStream

#### 1. 介绍:

a. ObjectOutputStream(序列化):



b. ObjectInputStream(反序列化):



## 2. 序列化数据案例:

```

//序列化后，保存的文件格式，按他的格式保存
//指定保存序列化数据路径
String filePath = "c:\\\\data.dat";

try{
    //创建序列化对象输出流
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filePath));

    // 写入基本数据类型，Java 自动将这些数据类型转换为对应的包装类，并序列化
    oos.writeInt(100);    //int->Integer (实现了Serializable)
    oos.writeBoolean(true); //boolean->Boolean (实现了Serializable)
    oos.writeChar('a');   //char->Character (实现了Serializable)
    oos.writeDouble(9.5); //double->Double (实现了Serializable)
    oos.writeUTF("lover");//String

    //class Dog implements Serializable{...}(也需要实现Serializable)
    oos.writeObject(new Dog("大黄", 10, "中华", "白色"));

    //关闭对象输出流：触发序列化过程并关闭输出流底层的文件输出流
    oos.close();
    System.out.println("数据保存完毕(序列化形式)");
} catch (IOException e) {
    e.printStackTrace();
}

```

### 3. 反序列化数据案例:

```
//创建输入流对象
ObjectInputStream ois=null
try{
    ois = new ObjectInputStream(new FileInputStream("src\\data.dat"));
    //读取数据
    //读取顺序必须与写入顺序一致
    System.out.println(ois.readInt());
    System.out.println(ois.readBoolean());
    System.out.println(ois.readChar());
    System.out.println(ois.readDouble());
    System.out.println(ois.readUTF()); //读取字符串
    System.out.println(ois.readObject()); //读取自定义对象,每个自定义对象都可用这个继续读取
} catch (IOException e) {
    e.printStackTrace();
} finally {
    //关闭对象输入流:同时会关闭底层的文件输入流
    try {
        if (ois != null) {
            ois.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

### 4. 注意事项:

- a. 读写顺序必须一致
- b. 序列化或反序列化对象,必须实现Serializable接口(另外一个也行,但是最好用这个,这个没有别的方法)
- c. 序列化的类建议添加**SerialVersionUID**,提高版本兼容性
- d. 序列化对象:默认将所有属性序列化,除了static或transient修饰的成员
- e. 所以序列化对象时,也要求属性类型也需要实现序列化接口(Serializable)
- f. 序列化具备可继承性,父类实现序列化,子类也默认实现序列化

### 4. 标准输入输出流(字节流):System.in/out

System.in:InputStream类型,从键盘接收数据

System.out:PrintStream类型,输出到显示器

## 5. 转换流(字符流)InputStreamReader和OutputStreamWriter

### 1. 介绍:

- a. InputStreamReader为Reader的子类,将InputStream(字节流)转换为Reader(字符流)
- b. OutputStreamWriter为Writer子类,将OutputStream(字节流)转换为Writer(字符流)
- c. 处理纯文本数据,字符流效率高并可解决中文乱码问题
- d. 可在使用时指定编码形式(utf-8,gbk等...)

### 2.1 转换和读取案例:

```
String filePath="c:\\a.txt";

//将文件输入流(字节流)转换为字符流并指定为GBK,并用BufferedReader包装
BufferedReader br = new BufferedReader(
    new InputStreamReader(new FileInputStream(filePath), "gbk"))

String s = br.readLine();
System.out.println("读取内容=" + s);

//关闭外层流,释放资源
br.close()
```

### 2.2 转换和写入案例:

```
//将文件输出流(字节流)转换为字符流并指定为GBK
OutputStreamWriter osw =new OutputStreamWriter(
    new FileOutputStream("c:\\a.txt"), "gbk");

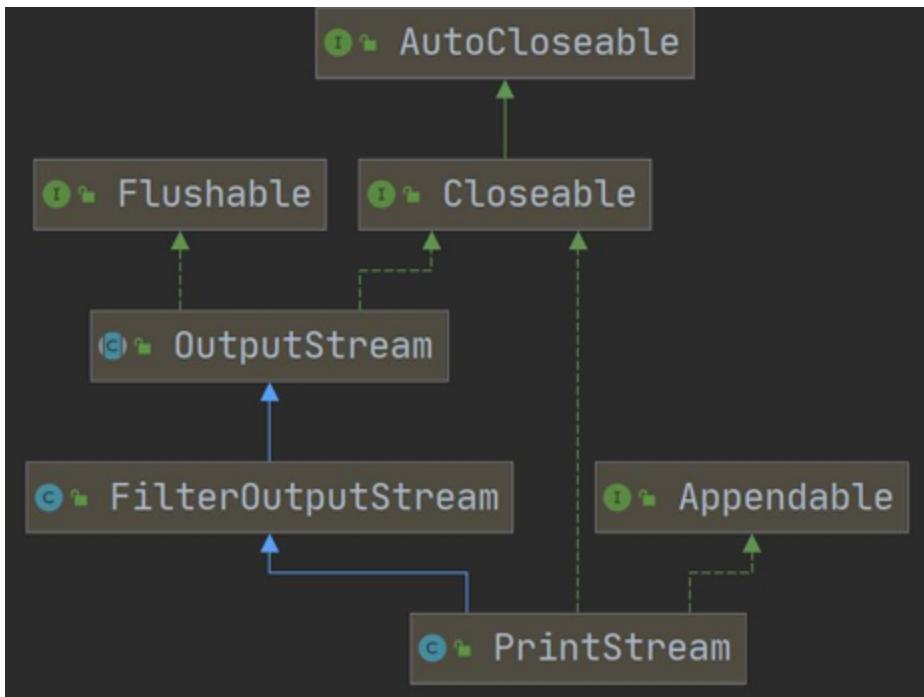
//写入到文件中
osw.write("Darling I love you forever~");

//关闭对象,释放资源并确保数据写入
osw.close();
```

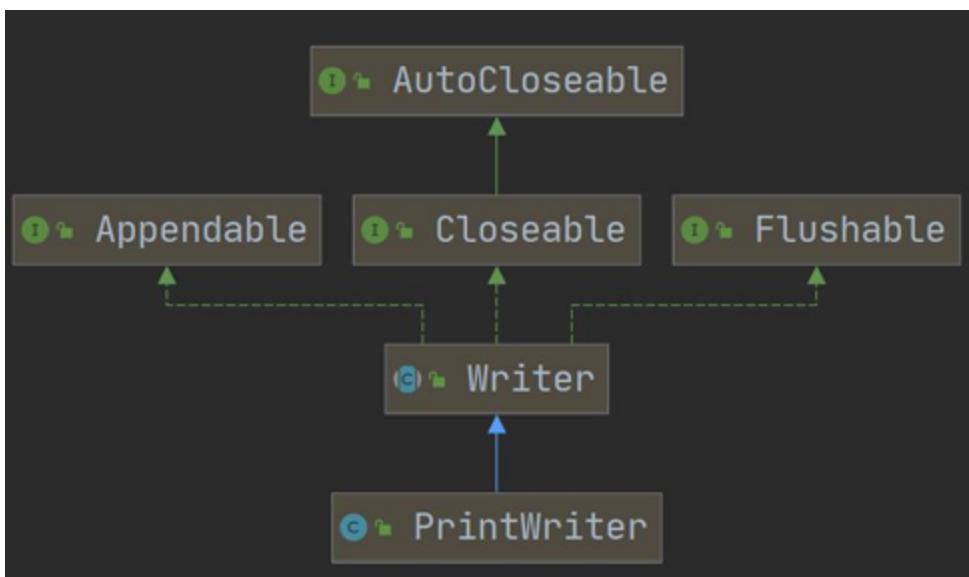
## 3.12 打印流(只有输出流)PrintStream和PrintWriter

### 1. 介绍:

- a. PrintStream:



b. PrintWriter:



## 2. 使用案例:

```
//指定输出到 标准输出
PrintWriter printWriter = new PrintWriter(System.out);
//指定字符编码与输出到lover.txt
PrintWriter printWriter = new PrintWriter(new FileWriter("c:\\lover.txt"));
printWriter.print("Darling I love you forever~");

//flush强制输出并关闭流，确保数据写入到文件
printWriter.close();

//PrintStream默认输出至显示器(标准输出)
PrintStream out = System.out;
out.print("Darling I love you forever~");

//print 底层使用的是write，可直接调用write进行打印/输出
//需要字符串转为字节再输出文本
out.write("Darling I love you forever~".getBytes());
out.close();

//修改打印流输出位置/设备
System.setOut(new PrintStream("c:\\f1.txt"));
System.out.println("Darling I love you forever~");
```

## 3.13 Properties类(读写配置文件):

### 1. 介绍:

- a. 专门用于读写配置文件的集合类

配置文件格式:键=值

- b. 键值对不需要空格,值不需要用引号,默认为String

## 2. 常见方法:

### 3) Properties的常见方法

- **load**: 加载配置文件的键值对到Properties对象
- **list**: 将数据显示到指定设备
- **getProperty(key)**: 根据键获取值
- **setProperty(key,value)**: 设置键值对到Properties对象
- **store**: 将Properties中的键值对存储到配置文件, 在idea中, 保存信息到配置文件, 如果含有中文, 会存储为unicode码

<http://tool.chinaz.com/tools/unicode.aspx> unicode码查询工具

### 3.1 读取案例:

```
Properties properties = new Properties();
//加载配置文件
properties.load(new FileReader("src\\mysql.properties"));

//用list将所有属性输出到标准输出
properties.list(System.out);

//获取properties对象的属性并输出
String user = properties.getProperty("user");
String pwd = properties.getProperty("pwd");
System.out.println("用户名=" + user);
System.out.println("密码是=" + pwd);
```

### 3.2 添加修改案例:

```
Properties properties = new Properties();
//设置属性,键值对
//没key就创建,有就修改,即为覆盖制
properties.setProperty("charset", "utf8");
properties.setProperty("pwd", "888888");
//中文会被保存为其unicode码值
properties.setProperty("user", "汤姆");

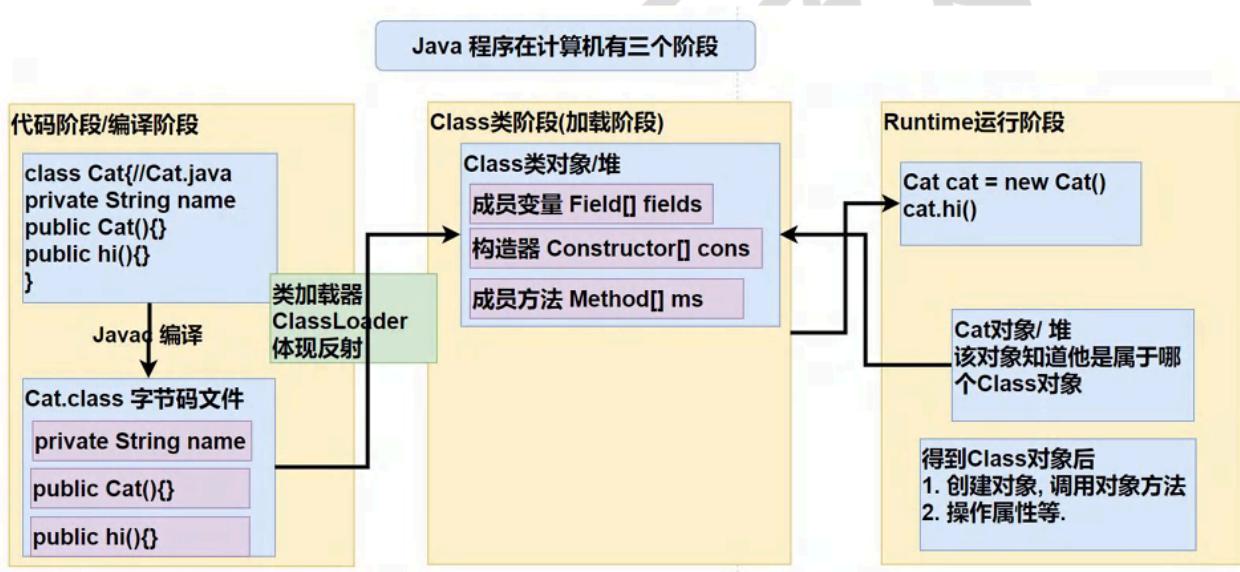
//创建输出流并储存,第一个参数为输出流(储存路径),第二个为注释
properties.store(new FileOutputStream("src\\mysql2.properties"), null);
```

## 3.14 反射(运行时动态检查.修改类):

### 1. 基本概念与优缺点:

- a. 能够允许程序在运行时无视访问限制符获取所有类的信息并操作其属性和方法
- b. (加载完类后)通过一个Class类型的类(一个类只有一个Class对象)实现
- c. Class对象拥有类的完整信息(反射会破坏类的封装性)
- d. 反射中可将方法,成员变量,构造器等等都能当成对象
- e. 但是反射是解释执行(可动态修改/更新代码),运行速度较慢

//java.lang.Class



1. Method和Field、Constructor对象都有setAccessible()方法
2. setAccessible作用是启动和禁用访问安全检查的开关
3. 参数值为true表示 反射的对象在使用时取消访问检查, 提高反射的效率。参数值为false则表示反射的对象执行访问检查

java.lang.reflect  
类 AccessibleObject

java.lang.Object  
└ java.lang.reflect.AccessibleObject

所有已实现的接口:  
[AnnotatedElement](#)

直接已知子类:  
[Constructor](#), [Field](#), [Method](#)

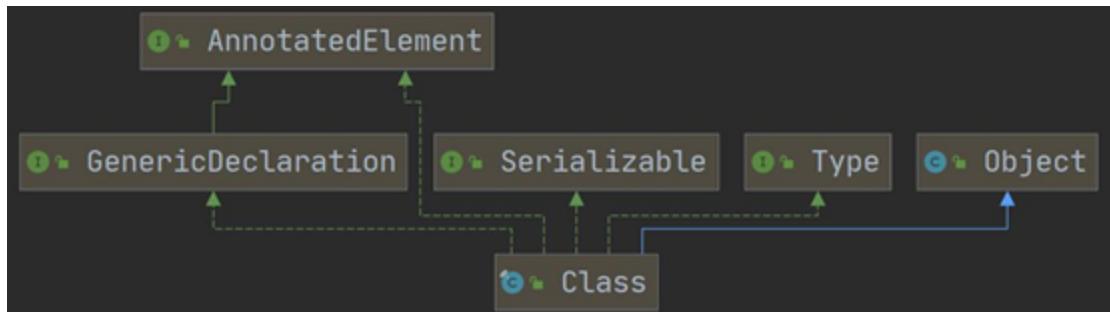
## 2. 使用案例:

```
//加载类,返回Class类型的对象cls, classfullpath为类的地址  
Class cls = Class.forName(classfullpath);  
//通过newInstance创建实例  
Object o = cls.getDeclaredConstructor().newInstance();  
//Method 方法对象,反射中可把方法视为对象(万物皆对象)  
Method method1 = cls.getMethod(methodName);  
//通过方法对象method1使用方法  
method1.invoke(o);  
//Field 对象表示某个类的成员变量,getField 不能得到私有的属性  
Field nameField = cls.getField("age");  
//Constructor 对象表示构造器,()中可指定构造器参数类型, 返回无参构造器  
Constructor constructor = cls.getConstructor();
```

## 3. Class类

### 1. 介绍:

- a. Class也是类,因此也继承Object类



- b. Class类对象不是new出来的,而是系统创建的(见3)
- c. 对于某个类的Class类对象,在内存中只有一份,因为类只加载一次
- d. 每个类的实例都会记得自己是由哪个Class实例所生成
- e. 通过Class对象可完整得到一个类的完整结构(通过一系列API)
- f. Class对象存放在堆
- g. 类的字节码二进制数据,是放在方法区的,有的地方称为类的元数据(包括方法代码,变量名,方法名,访问权限等等)
- h. 有Class类对象:接口,枚举enum,注解annotation,基本数据类型和各自类

### 2. 方法:

- a. `forName(String name)`: 获取到Car类 对应的 Class对象

```
Class cls = Class.forName(classAllPath); //表示不确定的Java类型
```

- b. getPackage().getName():得到包名  
System.out.println(cls.getPackage().getName());//包名
- c. getName():得到全类名  
System.out.println(cls.getName());
- d. newInstance():返回该Class对象cla的一个实例  
Car car = (Car) cls.newInstance();
- e. getField(属性名):获取属性  
Field brand = cls.getField("brand");  
System.out.println(brand.get(car));//获取该car对象的brand属性
- f. 属性名.set(实例,赋值):给属性赋值  
brand.set(car, "奔驰");
- g. getFields():返回所有的属性(字段)  
Field[] fields = cls.getFields();

### 3. 获得Class类对象的三种方式:

- a. 对广泛的类(此处cls1,2,3,4为同一对象)
  - a. 通过类地址(配置文件):Class<?> cls1 = Class.forName(classAllPath)
  - b. 通过类名:Class cla2 = Cat.calss
  - c. 通过对象:Class cla3 = cat.getClass()//无法获得类的动态加载
  - d. 通过类加载器:ClassLoader classLoader = car.getClass().getClassLoader();  
Class cls4 = classLoader.loadClass(classAllPath);  
//可获得类的动态加载
- b. 对基本数据类型(int,boolean...):  
Class integerClass = int.class;  
//此处为int的Class对象
- c. 对基本数据类型的包装类(Integer...):  
Class integerClass = Integer.TYPE;  
//此处也为int的Class对象

## 4. 类的加载实际情况:



加载:将**字节码**从不同数据源转化为**二进制字节流**加载到**内存中**,并生成对应**Class**对象

验证:(字面意思)

准备:**在方法区中分配内存并初始化**

解析:**将常量池内的符号引用替换为直接引用**

初始化:自动收集类所有静态的语句并合并(所以使用类静态属性也会让类加载)

## 5. 各类方法:

### 23.7.1 第一组: java.lang.Class 类

**com.hspedu.reflection ReflectionUtils.java**

1. `getName`: 获取全类名
2. `getSimpleName`: 获取简单类名
3. `getFields`: 获取所有public修饰的属性, 包含本类以及父类的
4. `getDeclaredFields`: 获取本类中所有属性
5. `getMethods`: 获取所有public修饰的方法, 包含本类以及父类的
6. `getDeclaredMethods`: 获取本类中所有方法
7. `getConstructors`: 获取本类所有public修饰的构造器
8. `getDeclaredConstructors`: 获取本类中所有构造器
9. `getPackage`: 以Package形式返回包信息
10. `getSuperClass`: 以Class形式返回父类信息
11. `getInterfaces`: 以Class[]形式返回接口信息
12. `getAnnotations`: 以Annotation[]形式返回注解信息

### 23.7.2 第二组: java.lang.reflect.Field 类

1. `getModifiers`: 以int形式返回修饰符  
[说明: 默认修饰符是0, public是1, private是2, protected是4, static是8, final是16],  $public(1) + static(8) = 9$
2. `getType`: 以Class形式返回类型
3. `getName`: 返回属性名

### 23.7.3 第三组: java.lang.reflect.Method 类

1. `getModifiers`: 以int形式返回修饰符  
[说明: 默认修饰符是0, public是1, private是2, protected是4, static是8, final是16]
2. `getReturnType`: 以Class形式获取返回类型
3. `getName`: 返回方法名
4. `getParameterTypes`: 以Class[]形式返回参数类型数组

### 23.7.4 第四组: java.lang.reflect.Constructor 类

1. `getModifiers`: 以int形式返回修饰符
2. `getName`: 返回构造器名(全类名)
3. `getParameterTypes`: 以Class[]形式返回参数类型数组

## 6. 用反射Class对象创建对象:

- a. 使用public无参构造器newInstance:

```
Class<?> userClass = Class.forName(类地址);
Object o = userClass.newInstance();
```

- b. 使用类中指定的private构造器:

```
Constructor<?> constructor = userClass.getConstructor(String.class);
```

```
Object lover = constructor.newInstance("lover");
//此处需满足类中有 private Lover(String s){...}
c. 直接用反射开放访问private构造器
Constructor<?> constructor1 = userClass.getDeclaredConstructor(int.class,
String.class);
//setAccessible(true)表示现在可以访问私有成员(平时尊重而不访问)
constructor1.setAccessible(true);
Object lover = constructor.newInstance("lover");
```

## 3. 15 正则表达式:

### 1. 介绍:

- a. 对字符串执行模式匹配的技术
- b. 匹配中文可能会有问题

### 2. 底层实现(重要):

目标:匹配数字

```

String content = "文本...";
// \d 表示一个任意的数字
String reg = "(\\d\\d)(\\d\\d)";
// 创建模式对象-即正则表达式对象
Pattern pattern = Pattern.compile(reg);
// 创建匹配器matcher,以正则表达式的规则匹配content字符串
Matcher matcher = pattern.matcher(content);
while (matcher.find()) {
    // 小结
    // 1. 若正则表达式有() 即分组,如此处的reg
    // 2. 找到后,将子字符串的开始的索引记录到 matcher对象的属性 int[]groups;
        // 2.1 groups[0] = 0 , 把该子字符串的结束的索引+1的值记录到 groups[1]=4
        // 2.2 记录1组()匹配到的字符串 groups[2]=0 groups[3]=2
        // 2.3 记录2组()匹配到的字符串 groups[4]=2 groups[5]=4
        // 2.4 记录oldLast值为 子字符串的结束索引+1的值,下次执行find时从该值开始匹配
    // 3. find()方法的:group(0) 表示匹配到的子字符串
        // 3.1 group(1) 表示匹配到的子字符串的第1组字串
        // 3.2 group(2) 表示匹配到的子字符串的第2组字串
        // 3.3 但分组的数不能越界.
    System.out.println("找到: " + matcher.group(0));
    System.out.println("第 1 组()匹配到的值=" + matcher.group(1));
    System.out.println("第 2 组()匹配到的值=" + matcher.group(2));
}

```

### 3. 语法-元字符:

#### 1. 元字符按功能分类:

- a. 限定符
- b. 选择匹配符
- c. 分组组合和反向引用符
- d. 特殊字符
- e. 字符匹配符
- f. 定位符

#### 2. 转移符"(元字符):

检索特殊字符的,和之前提到过的功能一样

需要用到转移符的字符: \* + () \$ / \ ? [] ^ { }

### 3. 字符匹配符(元字符):

| 符号  | 符号       | 示例     | 解释                        |
|-----|----------|--------|---------------------------|
| [ ] | 可接收的字符列表 | [efgh] | e、f、g、h中的任意1个字符           |
| [^] | 不接收的字符列表 | [^abc] | 除a、b、c之外的任意1个字符，包括数字和特殊符号 |
| -   | 连字符      | A-Z    | 任意单个大写字母                  |

|    |                                 |            |                              |                     |
|----|---------------------------------|------------|------------------------------|---------------------|
| .  | 匹配除 \n 以外的任何字符                  | a..b       | 以a开头，b结尾，中间包括2个任意字符的长度为4的字符串 | aaab、aefb、a35b、a#*b |
| \d | 匹配单个数字字符，相当于[0-9]               | \d{3}(\d)? | 包含3个或4个数字的字符串                | 123、9876            |
| \D | 匹配单个非数字字符，相当于[^0-9]             | \D(\d)*    | 以单个非数字字符开头，后接任意个数字字符串        | a、A342              |
| \w | 匹配单个数字、大小写字母字符，相当于[0-9a-zA-Z]   | \d{3}\w{4} | 以3个数字字符开头的长度为7的数字字母字符串       | 234abcd、12345Pe     |
| \W | 匹配单个非数字、大小写字母字符，相当于[^0-9a-zA-Z] | \W+\d{2}   | 以至少1个非数字字母字符开头，2个数字字符结尾的字符串  | #29、#@10            |

### 4. 选择匹配符'|'(元字符):

与 "||"、"或" 概念一致: 匹配之前或之后的表达式

### 5. 限定符(元字符):

用于指定前面字符和组合项连续出现多少次

| 符号  | 含义                     | 示例        | 说明                    | 匹配输入                         |
|-----|------------------------|-----------|-----------------------|------------------------------|
| *   | 指定字符重复0次或n次 (无要求) 零到多  | (abc)*    | 仅包含任意个abc的字符串，等效于\w*  | abc、<br>abcabcabc            |
| +   | 指定字符重复1次或n次 (至少一次) 1到多 | m+(abc)*  | 以至少1个m开头，后接任意个abc的字符串 | m、 mabc、<br>mabcabc          |
| ?   | 指定字符重复0次或1次 (最多一次) 0到1 | m+abc?    | 以至少1个m开头，后接ab或abc的字符串 | mab、 mabc、<br>mmab、<br>mmabc |
| {n} | 只能输入n个字符               | [abcd]{3} | 由abcd中字母组成的任意长度为3的字符串 | abc、 dbc、<br>adc             |

|       |                    |             |                              |                            |
|-------|--------------------|-------------|------------------------------|----------------------------|
| {n,}  | 指定至少 n 个匹配         | [abcd]{3,}  | 由abcd中字母组成的任意长度不小于3的字符串      | aab、 dbc、<br>aaabdc        |
| {n,m} | 指定至少 n 个但不多于 m 个匹配 | [abcd]{3,5} | 由abcd中字母组成的任意长度不小于3，不大于5的字符串 | abc、 abcd、<br>aaaaa、 bcdab |

## 6. 定位符(元字符):

|    |             |                                |                                   |                                                                            |
|----|-------------|--------------------------------|-----------------------------------|----------------------------------------------------------------------------|
| ^  | 指定起始字符      | <code>^[0-9]+[a-z]*</code>     | 以至少1个数字开头，后接任意个小写字母的字符串           | 123、6aa、555edf                                                             |
| \$ | 指定结束字符      | <code>^[0-9]\-[a-z]+\\$</code> | 以1个数字开头后接连字符“-”，并以至少1个小写字母结尾的字符串  | 1-a                                                                        |
| \b | 匹配目标字符串的边界  | <code>han\b</code>             | 这里说的字符串的边界指的是子串间有空格，或者是目标字符串的结束位置 | hanshunping<br>s <span style="background-color: yellow;">han nnhan</span>  |
| \B | 匹配目标字符串的非边界 | <code>han\B</code>             | 和\b的含义刚刚相反                        | hanshunping<br>s <span style="background-color: yellow;">phan nnhan</span> |

## 7. 分组:

捕获匹配:匹配并保存

非捕获匹配:仅匹配不保存

|                  |                                                                                                                                             |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| (pattern)        | 非命名捕获。捕获匹配的子字符串。编号为零的第一个捕获是由整个正则表达式模式匹配的文本，其它捕获结果则根据左括号的顺序从1开始自动编号。                                                                         |
| (?<name>pattern) | 命名捕获。将匹配的子字符串捕获到一个组名称或编号名称中。用于name的字符串不能包含任何标点符号，并且不能以数字开头。可以使用单引号替代尖括号，例如 (?'name')                                                        |
| (?:pattern)      | 匹配 pattern 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用"or"字符 ( ) 组合模式部件的情况很有用。例如，'industr(?:y ies)' 是比 'industry industries' 更经济的表达式。          |
| (?=pattern)      | 它是一个非捕获匹配。例如，'Windows (?=95 98 NT 2000)' 匹配 "Windows 2000" 中的 "Windows"，但不匹配 "Windows 3.1" 中的 "Windows"。                                    |
| (?!pattern)      | 该表达式匹配不处于匹配 pattern 的字符串的起始点的搜索字符串。它是一个非捕获匹配。例如，'Windows (?!95 98 NT 2000)' 匹配 "Windows 3.1" 中的 "Windows"，但不匹配 "Windows 2000" 中的 "Windows"。 |

## 8. 反向引用:

引用之前匹配到的捕获组来进一步匹配

反向引用通常是通过 `\n` (其中n是一个数字) 来实现的, n代表你想要引用的捕获组的序号。例如, `\1` 引用第一个捕获组的内容, `\2` 引用第二个捕获组的内容, 以此类推。

## 示例

假设我们有一个字符串 `"123abc123"`, 我们想要匹配连续出现的相同数字:

regex □ 复制代码

```
正则表达式: (\d) \d* \1
匹配字符串: "123abc123"
```

在这个例子中:

- `(\d)` 创建了一个捕获组, 用来匹配任意一个数字。
- `\d*` 匹配零个或多个数字。
- `\1` 是一个反向引用, 它引用了第一个捕获组中匹配到的数字。

所以, 这个正则表达式会匹配 `"123"`, 因为 `\1` 引用了第一个捕获组中匹配到的数字 `1`, 并且确保了后面跟着的数字

## 4. 常用类:

- a. Pattern类:正则表达式对象,接收一个String的正则表达式创建
- b. Matcher类:对输入字符串进行解释和匹配,通过pattern获得
- c. PatternSyntaxException类:表示正则表达式的语法错误

## 5. String中的使用

```
String content = "文本...";
```

- a. 替换:

```
//将 JDK1.3 和 JDK1.4 替换成JDK
content = content.replaceAll("JDK1\.3|JDK1\.4", "JDK");
```

b. 验证:是否是138或139开头的11位数字

```
if (content.matches("1(38|39)\\d{8}")) {  
    System.out.println("验证成功");  
} else {  
    System.out.println("验证失败");  
}
```

c. 分割:

```
//要求按照 # 或者- 或者 ~ 或者 数字 来分割  
content = "hello#abc-jack12smith~北京";  
String[] split = content.split("#|-|~|\\d+");  
for (String s : split) {  
    System.out.println(s);  
}
```

## 四.网络编程(java.net包)

### 1. 一些概念

#### 1.1 ip地址:

- a. 用于唯一标识网络中的每台计算机/主机
- b. 查看ip地址:ipconfig
- c. 表示形式(0~255):xx.xx.xx.xx
- d. ip地址组成=网络地址+主机地址  
(127.0.0.1表示本机位置)
- e. ipv4地址分类:

|    |           |                  |              |
|----|-----------|------------------|--------------|
| A类 | 0         | 7位<br>网 络 号      | 24位<br>主 机 号 |
| B类 | 1 0       | 14位<br>网 络 号     | 16位<br>主 机 号 |
| C类 | 1 1 0     | 21位<br>网 络 号     | 8位<br>主 机 号  |
| D类 | 1 1 1 0   | 28位<br>多 播 组 号   |              |
| E类 | 1 1 1 1 0 | 27位<br>(留 待 后 用) |              |

| 类型 | 范 围                         |
|----|-----------------------------|
| A  | 0.0.0.0 到 127.255.255.255   |
| B  | 128.0.0.0 到 191.255.255.255 |
| C  | 192.0.0.0 到 223.255.255.255 |
| D  | 224.0.0.0 到 239.255.255.255 |
| E  | 240.0.0.0 到 247.255.255.255 |

## 1.2 域名:

将ip地址映射成域名(HTTP协议)

## 1.3 端口号:

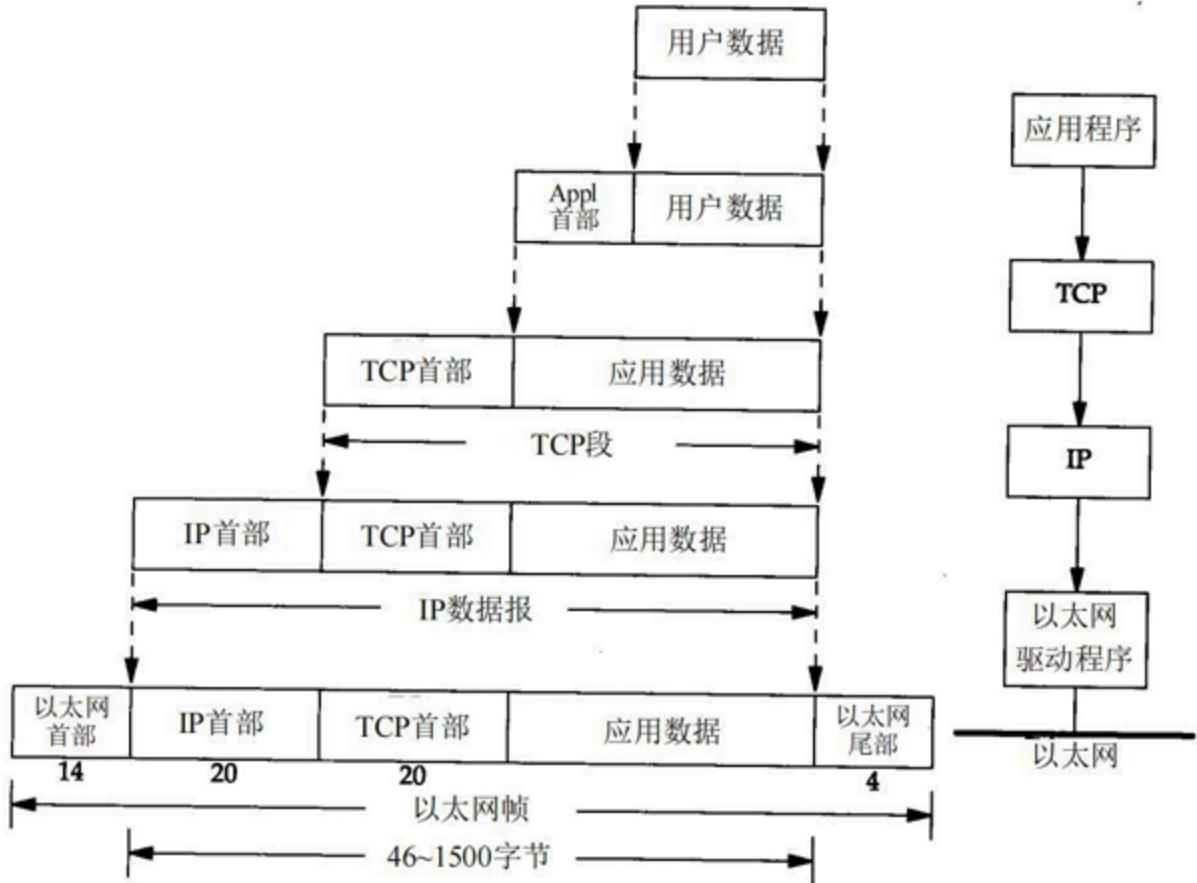
- a. 端口号:标识计算机上某个特定的网络程序
- b. 两个字节,以整数形式:0~65535
- c. 常见端口号:
  - a. tomcat:8080
  - b. mysql:3306
  - c. oracle:1521
  - d. sqlserver:1433

### 1.4.1 网络通信协议:

- a. 网络层的IP协议+传输层的TCP协议

| OSI模型 | TCP/IP模型  | TCP/IP模型各层对应协议         |
|-------|-----------|------------------------|
| 应用层   |           | HTTP、ftp、telnet、DNS... |
| 表示层   | 应用层       |                        |
| 会话层   |           |                        |
| 传输层   | 传输层 (TCP) | TCP、UDP、...            |
| 网络层   | 网络层 (IP)  | IP、ICMP、ARP...         |
| 数据链路层 |           |                        |
| 物理层   | 物理+数据链路层  | Link                   |

c.



## 1.4.2 传输层协议TCP, UDP:

a. TCP协议(传输控制协议):

- a. 使用前需要建立TCP连接,形成传输数据通道
- b. 传输前使用三次握手 是可靠的

## 1. 第一次握手 (SYN) :

- 客户端发送一个SYN (同步序列编号) 报文到服务器，并进入SYN\_SENT状态，等待服务器确认。
- 在这个报文中，客户端会选择一个初始序列号 (Sequence Number) 。

## 2. 第二次握手 (SYN-ACK) :

- 服务器收到客户端的SYN报文后，需要确认客户端的SYN，同时自己也发送一个SYN报文作为回应，并进入SYN\_RECEIVED状态。
- 服务器发送的SYN-ACK报文中包含对客户端SYN报文的确认(ACK)以及自己的初始序列号。

## 3. 第三次握手 (ACK) :

- 客户端收到服务器的SYN-ACK报文后，会发送一个ACK报文来确认服务器的SYN。
- 客户端发送ACK报文后，连接就进入了ESTABLISHED状态，此时客户端和服务器都准备好进行数据传输了。
  - c. TCP通信两个应用进程:客户端,服务端
  - d. 可进行大数据量传输
  - e. 传输完毕需释放已建立连接(效率低)
- b. UDP协议(用户数据协议):
  - a. 数据,源,目的封装为数据报,无需连接,不可靠
  - b. 数据报大小限制64kb,不适合传输大数据量
  - c. 无需释放资源,速度快(例如短信)

## 1.5 Socket套接字

- a. Socket开发网络应用程序被广泛采用
- b. 通信两端都需要Socket,为两及其通信的端点
- c. Socket运行程序把网络连接当作流,数据在其通过IO传输
- d. 发起通信为客户端,等待通信请求为服务端

## 2. InetAddress类

### 1. 相关方法及使用:

a. getLocalHost:获取本机InetAddress对象

```
//System.out.println(InetAddress.getLocalHost());
```

b. getByName:根据指定主机/域名获取ip地址对象

```
//System.out.println(InetAddress.getByName("ThinkPad-PC"));
```

```
//System.out.println(InetAddress.getByName("www.lover.com"));
```

c. getHostName:获取InetAddress对象主机名

```
//System.out.println(InetAddress
```

```
.getByName("www.lover.com")
```

```
.getHostName())
```

d. getHostAddress:获取InetAddress对象地址

```
//System.out.println(InetAddress
```

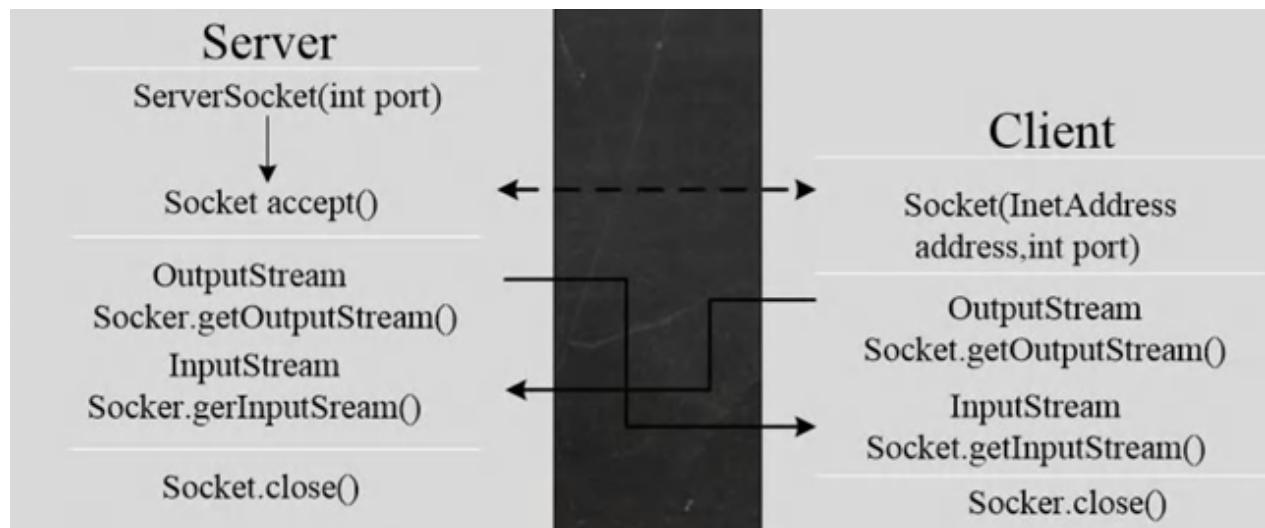
```
.getByName("www.lover.com")
```

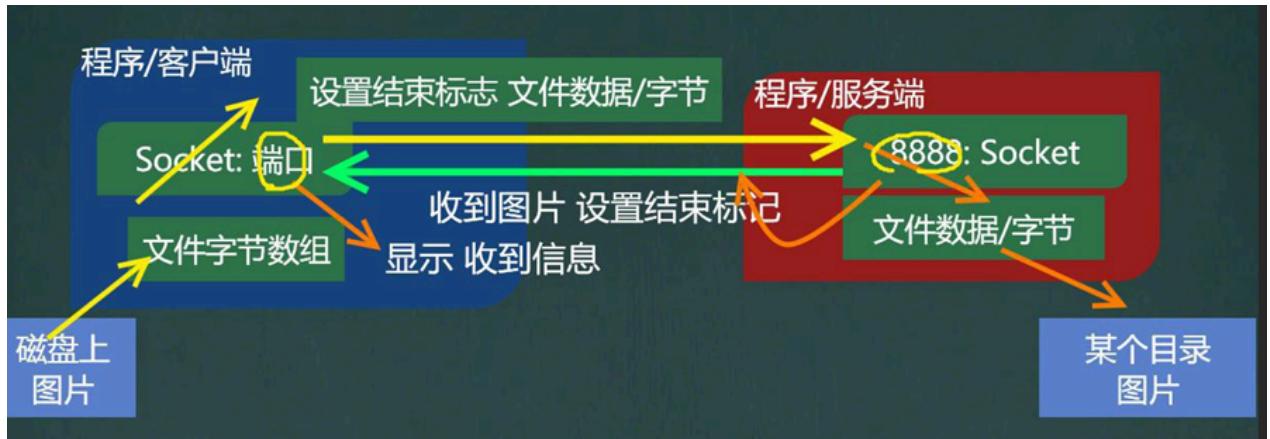
```
.getHostAddress())
```

## 3. TCP网络通信编程:

### 1. 基本介绍:

- a. 基于客户端-服务端的网络通信
- b. 底层使用TCP/IP协议
- c. 基于Socket的TCP编程





## 2. 应用案例(暂且不管)

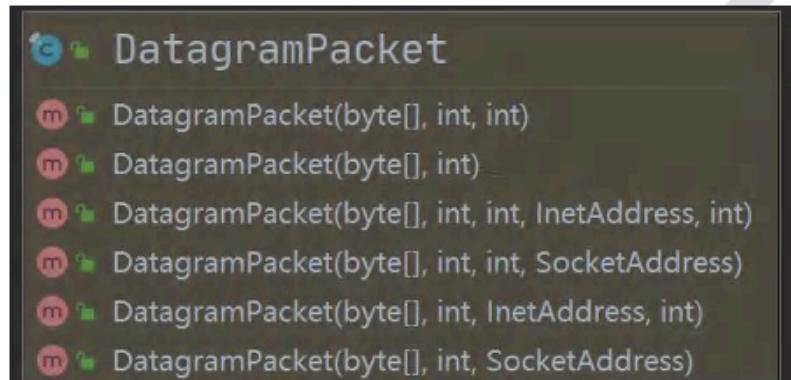
## 4. UDP网络通信编程(无连接,低延迟,不保证数据可靠):

### 1. 基本介绍:

1. 类 **DatagramSocket** 和 **DatagramPacket**[数据包/数据报] 实现了基于 UDP 协议网络程序。
2. UDP数据报通过数据报套接字 **DatagramSocket** 发送和接收，系统不保证UDP 数据报一定能够安全送到目的地，也不能确定什么时候可以抵达。
3. **DatagramPacket** 对象封装了UDP数据报，在数据报中包含了发送端的IP地址和端口号以及接收端的IP地址和端口号。
4. UDP协议中每个数据报都给出了完整的地址信息，因此无须建立发送方和接收方的连接

## 2. 基本流程:

1. 核心的两个类/对象 DatagramSocket与DatagramPacket
2. 建立发送端, 接收端(没有服务端和客户端概念)
3. 发送数据前, 建立数据包/报 DatagramPacket对象
4. 调用DatagramSocket的发送、接收方法
5. 关闭DatagramSocket



# 五.数据库

## 1. MySql和JDBC

### 1. MySql和JDBC

#### 1. MySQL自身:

端口(port)3306:默认使用端口(可更改)

实质上是一个**数据库管理系统(DBMS)**,可以管理多个数据库

而一个数据库可以创建多个表来保存信息

#### 2. MySQL使用:

##### 1. 分类:

**DDL : 数据定义语句 [create 表 , 库...]**

**DML : 数据操作语句 [增加 insert , 修改 update, 删除 delete]**

**DQL : 数据查询语句 [select ]**

**DCL : 数据控制语句 [管理数据库: 比如用户权限 grant revoke ]**

## 2. 关于库的具体语句:

a. #使用指令创建数据库

```
CREATE DATABASE lover;
```

b. #删除数据库指令

```
DROP DATABASE lover;
```

c. #创建一个使用utf8字符集的lover2数据库

```
CREATE DATABASE lover2 CHARACTER SET utf8;
```

d. #创建一个使用utf8字符集,并带校对规则的lover3数据库

```
CREATE DATABASE lover3 CHARACTER SET utf8 COLLATE utf8_bin;
```

e. #WHERE从哪个字段NAME='tom'查询名字是tom

```
SELECT* FROM t1 WHERE NAME='tom'
```

f. #查看当前数据库服务器中的所有数据库

```
SHOW DATABASES;
```

g. #查看前面创建的lover数据库的定义信息

```
SHOW CREATE DATABASE lover ;//用反引号规避关键字
```

h. #删除前面创建的lover数据库

```
DROP DATABASE lover;
```

i. #这个备份的文件就是对应的sql语句

```
mysqldump -u root -p -B lover2 lover3 > C:\文件名.sql
```

j. #恢复数据库(注意:进入Mysql命令行再执行)

```
source C:\文件名.sql
```

## 3. 关于表的具体语句(dml语句):

a. 创建表(create):

```
CREATE TABLE user (
    id(指定列名) INT(指定列类型),
    name VARCHAR(255),
    password VARCHAR(255),
    birthday DATE)
CHARACTER SET utf8(字符集) COLLATE utf8_bin(校对规则) ENGINE INNODB(储存引擎);
```

b. 修改表(alter):

## 使用 ALTER TABLE 语句追加, 修改, 或删除列的语法.

|     |                                                                                                      |
|-----|------------------------------------------------------------------------------------------------------|
| 添加列 | <b>ALTER TABLE tablename<br/>ADD (column datatype [DEFAULT expr]<br/>[, column datatype]...);</b>    |
| 修改列 | <b>ALTER TABLE tablename<br/>MODIFY (column datatype [DEFAULT expr]<br/>[, column datatype]...);</b> |
| 删除列 | <b>ALTER TABLE tablename<br/>DROP (column);</b><br><b>查看表的结构 : desc 表名; -- 可以查看表的列</b>               |

**修改表名: Rename table 表名 to 新表名**

**修改表字符集: alter table 表名 character set 字符集;**

- c. 插入数据(insert):
  - d. 更新数据(update):
    - a. UPDATE 表名 SET 列名 = xxx;(修改所有记录)
    - b. UPDATE 表名 SET 列名 = xxx WHERE user\_name = xxx;(单独修改user为xxx的记录)
  - e. 删除数据(delete):
    - a. 删除表:DROP TABLE 表名;
    - b. 删除表所有记录:DELETE FROM 表名;
    - c. 删除表中特定用户的记录:DELETE FROM 表名 WHERE user\_name = xxx;
    - d. 不能删除某一列的数据(但可以用update设为null或")
  - f. 查询数据(select):
    - a. 查询加强:
      - %: 表示 0 到多个任意字符 \_: 表示单个任意字符
      - ?如何显示首字符为 S 的员工姓名和工资
- SELECT ename, sal FROM emp
- WHERE ename LIKE 'S%'
- ?如何显示第三个字符为大写 O 的所有员工的姓名和工资
- SELECT ename, sal FROM emp
- WHERE ename LIKE '\_\_O%'
- b. 分页查询:

```
SELECT * FROM emp
```

```
    ORDER BY empno
```

```
        LIMIT 每页显示记录数 * (第几页-1), 每页显示记录数
```

c. 多表查询: select ename, sal, grade from emp , salgrade where sal between losal and hisal;

d. 子查询/嵌套查询(嵌入在其他sql语句的select语句):

```
SELECT *
```

```
    FROM emp
```

```
    WHERE deptno = (
```

```
        SELECT deptno
```

```
        FROM emp
```

```
        WHERE ename = 'SMITH'
```

```
)
```

//可以当临时表使用

g. 排序查询结果(order by):

```
SELECT column1, column2, column3...
    FROM table;
    order by column asc|desc, ...
```

1. Order by 指定排序的列，排序的列既可以是表中的列名，也可以是select语句后指定的列名。
2. Asc 升序[默认]、Desc 降序
3. ORDER BY 子句应位于SELECT语句的结尾。

h. 统计数据(count):SELECT COUNT(\*) FROM student WHERE math>90;

i. 求和(sum):SELECT SUM(math+english+chinese) FROM student;

//avg,min和max同理

j. 数据分组(group by):SELECT name,id,section group by section;

//按照section列来分组name和id

4. 外连接:

5. mysql约束(用于确保数据库的数据满足特定商业规则):

1. primary key(主键):

字段名 字段类型 **primary key**

用于唯一的标示表行的数据,当定义主键约束后, 该列不能重复

a.

1. **primary key**不能重复而且不能为null。
2. 一张表最多只能有一个主键, 但可以是复合主键
3. **主键的指定方式 有两种**
  - 直接在字段名后指定: 字段名 **primakry key**
  - 在表定义最后写 **primary key(列名)**;
4. 使用**desc 表名**, 可以看到**primary key**的情况.
5. **老师提醒: 在实际开发中, 每个表往往都会设计一个主键.**

b. 主键的指定方式:

- a. 直接在字段名后指定: 字段名 **primakry key**
- b. 在表定义最后写 **primary key(列名)**

2. not null(非空):

如果在列上定义了not null,那么当插入数据时, 必须为列提供数据。

字段名 字段类型 **not null**

3. unique(唯一):

当定义了唯一约束后, 该列值是不能重复的..

字段名 字段类型 **unique**

• **unique 细节(注意): unique.sql**

1. 如果没有指定 not null , 则 unique 字段可以有多个null
2. 一张表可以有多个unique字段

4. foreign key(外键):

- a. 用于定义主表与从表的关系(外键约束定义在从表上,主表必须有主键约束或unique约束)

- b. 要求外键列数据必须在主表的主键列存在或为null
- c. 表类型为innodb(支持外键)
- d. 外键字段的类型要和主键字段的类型一致(长度可以不同)  
外键字段的值，必须在主键字段中出现过，或者为null [前提是外键字段允许为null]  
一旦建立主外键的关系，数据不能随意删除了.

e.

外键示意图

班级表 主表

| id  | nam    | add |
|-----|--------|-----|
| 100 | java01 | 北京  |
| 200 | web2   | 上海  |

学生表 外键所在表

| id | name | class_id  |
|----|------|-----------|
| 1  | tom  | 100       |
| 2  | jack | 200       |
| 3  | hsp  | 300[插入失败] |

如果我们要求，每个学生所在的班级号 class\_id 是存在的班级编号  
就可以把 class\_id 做成外键约束

## 5. check(检查条件满足与否)

用于强制行数据必须满足的条件,假定在sal列上定义了check约束,并要求sal列值在1000 ~ 2000之间如果不在1000 ~ 2000之间就会提示出错。

☞ 老师提示： oracle 和 sql server 均支持check ,但是mysql5.7 目前还不支持check ,只做语法校验，但不会生效。 **check.sql**

基本语法： 列名 类型 **check** (check条件)

user 表

id, name, sex(man,woman), sal(大于100 小于 900)



在mysql中实现check的功能，一般是在程序中控制，或者通过触发器完成。

## 6. 自增长:

字段名 整型 primary key **auto\_increment**

添加 自增长的字段方式

```
insert into xxx (字段1, 字段2.....) values(null, '值'....);  
insert into xxx (字段2.....) values('值1', '值2'....);  
insert into xxx values(null, '值1',.....)
```

1. 一般来说自增长是和primary key 配合使用的
2. 自增长也可以单独使用[但是需要配合一个unique]
3. 自增长修饰的字段为整数型的(虽然小数也可以但是非常非常少这样使用)
4. 自增长默认从 1开始, 你也可以通过如下命令修改alter table 表名 auto\_increment = 新的开始值;
5. 如果你添加数据时, 给自增长字段(列) 指定的有值, 则以指定的值为准, **如果指定了自增长, 一般来说, 就按照自增长的规则来添加数据.**

### 3. mysql索引(B+树):

#### 1. 索引类型:

1. 主键索引, 主键自动的为主索引 (类型Primary key)
2. 唯一索引 (UNIQUE)
3. 普通索引 (INDEX)
4. 全文索引 (FULLTEXT) [适用于MyISAM]  
一般开发, 不使用mysql自带的全文索引, 而是使用: 全文搜索 Solr 和 ElasticSearch (ES)

```
create table t1 (
id int primary key, -- 主键, 同时也是索引, 称为主键索引.
name varchar(32));
create table t2(
id int unique, -- id是唯一的, 同时也是索引, 称为unique索引.
```

#### 2. 索引使用:

1. 添加索引 (建小表测试 id , name ) index\_use.sql  
`create [UNIQUE] index index_name on tbl_name (col_name [(length)]  
[ASC | DESC], ...);`
2. alter table table\_name ADD INDEX [index\_name] (index\_col\_name,...)
3. 添加主键(索引) ALTER TABLE 表名 ADD PRIMARY KEY(列名...);
4. 删除索引  
`DROP INDEX index_name ON tbl_name;`
5. alter table table\_name drop index index\_name;
6. 删除主键索引 比较特别: `alter table t_b drop primary key;`
7. 查询索引(三种方式)  
`show index(es) from table_name;`  
`show keys from table_name;`  
`desc table Name;`

### 4. mysql事务

#### 1. 介绍:

事务用于保持数据一致性, 由一组相关的dml语句组成(要么全成功要么全失败)

## 2. 事务与锁:

执行事务操作时,MySQL会在表上加锁,防止其他用户改表数据

## 3. 事务的操作:

1. start transaction -- **开始一个事务**
2. savepoint 保存点名 -- 设置保存点
3. rollback to 保存点名 -- 回退事务
4. rollback -- 回退全部事务
5. commit -- 提交事务, 所有的操作生效, 不能回退

### 细节:

1. 没有设置保存点
2. 多个保存点
3. 存储引擎
4. 开始事务方式

在介绍回退事务前, 先介绍一下保存点(savepoint). 保存点是事务中的点. 用于取消部分事务, 当结束事务时 (commit), 会自动的删除该事务所定义的所有保存点. 当执行回退事务时, 通过指定保存点可以回退到指定的点, 这里我们作图说明

使用commit语句可以提交事务. 当执行了commit语句后, 会确认事务的变化、结束事务、删除保存点、释放锁, 数据生效。当使用commit语句结束事务子后, 其它会话[其他连接]将可以看到事务变化后的新数据[所有数据就正式生效.]

1. 创建一张测试表

```
CREATE TABLE t27  
(id INT,  
`name` VARCHAR(32));
```

2. 开始事务

```
START TRANSACTION
```

3. 设置保存点

```
SAVEPOINT a
```

4. 执行dml操作

```
INSERT INTO t27 VALUES(100, 'tom');
```

```
SELECT* FROM t27;
```

```
SAVE POINT b
```

5. 执行dml操作

```
INSERT INTO t27 VALUES(200, 'jack');
```

6. 回退到 b

```
ROLLBACK TO b
```

7. 继续回退 a

```
ROLLBACK TO a
```

```
ROLLBACK //如果这样, 表示直接回退到事务开始的状态.
```

```
COMMIT;
```

1. 如果不开始事务， 默认情况下， dml操作是自动提交的，不能回滚
2. 如果开始一个事务， 你没有创建保存点. 你可以执行 rollback， 默认就是回退到你事务开始的状态.
3. 你也可以在这个事务中(还没有提交时), 创建多个保存点.比如: savepoint aaa; 执行 dml , savepoint bbb;
4. 你可以在事务没有提交前，选择回退到哪个保存点.
5. mysql的事务机制需要innodb的存储引擎才可以使用,myisam不好使.
6. 开始一个事务 start transaction, set autocommit=off;

#### 4. 事务的隔离级别:

##### 1. 类别:

**脏读 (dirty read):** 当一个事务读取另一个事务尚未提交的改变 (update,insert,delete)时，产生脏读

**不可重复读 (nonrepeatable read):** 同一查询在同一事务中多次进行，由于其他提交事务所做的修改或删除，每次返回不同的结果集，此时发生不可重复读。

**幻读 (phantom read):** 同一查询在同一事务中多次进行，由于其他提交事务所做的插入操作，每次返回不同的结果集，此时发生幻读。

概念:Mysql隔离级别定义了事务与事务之间的隔离程度。

| Mysql隔离级别 (4种)                | 脏读 | 不可重复读 | 幻读 | 加锁读 |
|-------------------------------|----|-------|----|-----|
| 读未提交 (Read uncommitted)       | V  | V     | V  | 不加锁 |
| 读已提交 (Read committed)         | X  | V     | V  | 不加锁 |
| 可重复读 (Repeatable read)        | X  | X     | X  | 不加锁 |
| 可串行化 (Serializable) [演示重开客户端] | X  | X     | X  | 加锁  |

**说明:** V 可能出现 X 不会出现

## 2. 语法:

1. 查看当前会话隔离级别 **set transaction.sql**  
`select @@tx_isolation;`
2. 查看系统当前隔离级别  
`select @@global.tx_isolation;`
3. 设置当前会话隔离级别  
`set session transaction isolation level repeatable read;`
4. 设置系统当前隔离级别  
`set global transaction isolation level repeatable read;`
5. mysql 默认的事务隔离级别是 repeatable read ,一般情况下, 没有特殊要求, 没有必要修改 (因为该级别可以满足绝大部分项目需求)

## 4. 事务的特性(acid):

### 1. 原子性 (Atomicity)

原子性是指事务是一个不可分割的工作单位, 事务中的操作要么都发生, 要么都不发生。

### 2. 一致性 (Consistency)

事务必须使数据库从一个一致性状态变换到另外一个一致性状态

### 3. 隔离性 (Isolation)

事务的隔离性是多个用户并发访问数据库时, 数据库为每一个用户开启的事务, 不能被其他事务的操作数据所干扰, 多个并发事务之间要相互隔离。

### 4. 持久性 (Durability)

持久性是指一个事务一旦被提交, 它对数据库中数据的改变就是永久性的, 接下来即使数据库发生故障也不应该对其有任何影响

## 5. Mysql表类型和储存引擎

### 1. 基本介绍:

- a. 表类型由**储存引擎决定**, 包括MyISAM,innoDB,Memory等
- b. 数据表支持六种类型:CSV,Memory,ARCHIVE,MRG\_MYISAM,InnoDB
- c. MYISAM和MEMORY属于"非事务安全"型, 其他为"事务安全型"

| 特点      | Myisam | InnoDB |  | Memory | Archive |
|---------|--------|--------|--|--------|---------|
| 批量插入的速度 | 高      | 低      |  | 高      | 非常高     |
| 事务安全    |        | 支持     |  |        |         |
| 全文索引    | 支持     |        |  |        |         |
| 锁机制     | 表锁     | 行锁     |  | 表锁     | 行锁      |
| 存储限制    | 没有     | 64TB   |  | 有      | 没有      |
| B树索引    | 支持     | 支持     |  | 支持     |         |
| 哈希索引    |        | 支持     |  | 支持     |         |
| 集群索引    |        | 支持     |  |        |         |
| 数据缓存    |        | 支持     |  | 支持     |         |
| 索引缓存    | 支持     | 支持     |  | 支持     |         |
| 数据可压缩   | 支持     |        |  |        | 支持      |
| 空间使用    | 低      | 高      |  | N/A    | 非常低     |
| 内存使用    | 低      | 高      |  | 中等     | 低       |
| 支持外键    |        | 支持     |  |        |         |

## 2. 细节说明(MyISAM、InnoDB、MEMORY)

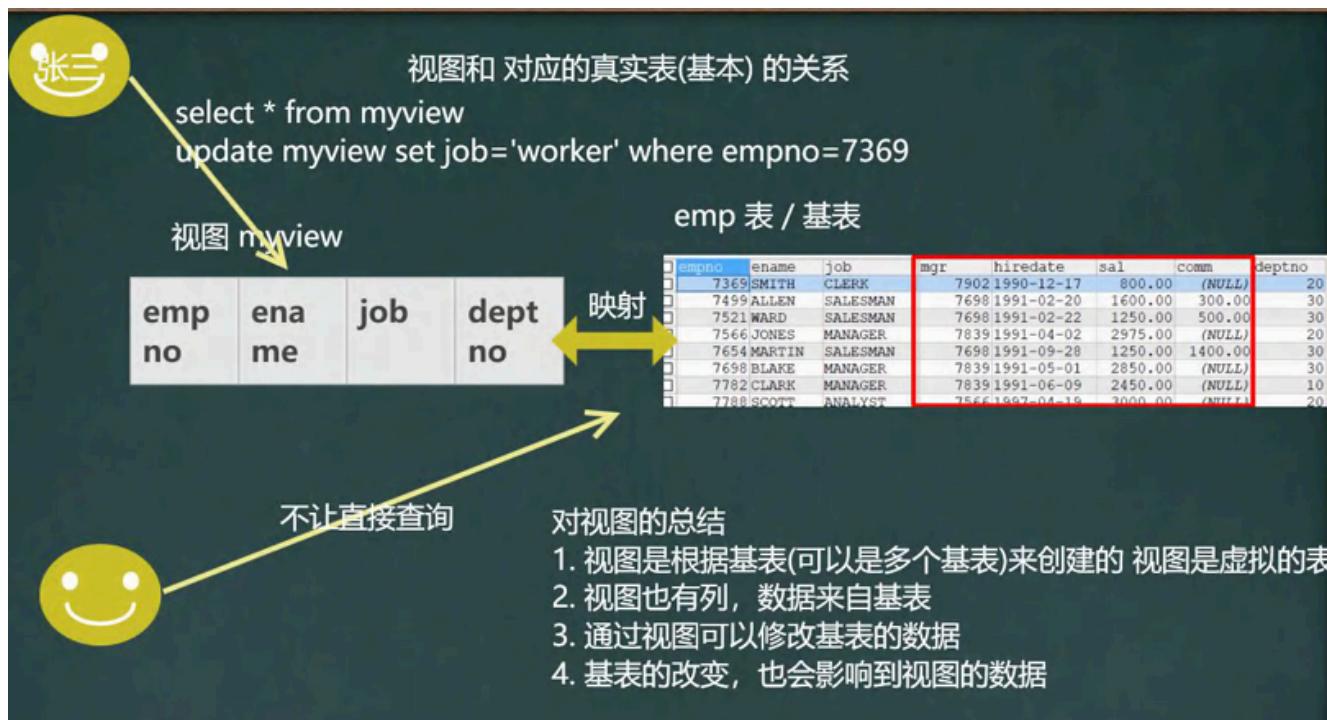
1. MyISAM不支持事务、也不支持外键，但其访问速度快，对事务完整性没有要求
  2. InnoDB存储引擎提供了具有提交、回滚和崩溃恢复能力的事务安全。但是比起MyISAM存储引擎，InnoDB写的处理效率差一些并且会占用更多的磁盘空间以保留数据和索引。
  3. MEMORY存储引擎使用存在内存中的内容来创建表。每个MEMORY表只实际对应一个磁盘文件。MEMORY类型的表访问非常得快，因为它的数据是放在内存中的，并且默认使用HASH索引。但是一旦MySQL服务关闭，表中的数据就会丢失掉，表的结构还在。
- 
1. 如果你的应用不需要事务，处理的只是基本的CRUD操作，那么MyISAM是不二选择，速度快
  2. 如果需要支持事务，选择InnoDB。
  3. Memory 存储引擎就是将数据存储在内存中，由于没有磁盘I/O的等待，速度极快。但由于是内存存储引擎，所做的任何修改在服务器重启后都将消失。（经典用法 **用户的在线状态()**）

**ALTER TABLE `表名` ENGINE = 储存引擎;**

## 6. 视图(view)

### 1. 介绍:

视图是一个虚拟表,内容由查询定义



**安全。**一些数据表有着重要的信息。有些字段是保密的, 不能让用户直接看到。这时就可以创建一个视图, 在这张视图中只保留一部分字段。这样, 用户就可以查询自己需要的字段, 不能查看保密的字段。

**性能。**关系数据库的数据常常会分表存储, 使用外键建立这些表的之间关系。这时, 数据库查询通常会用到连接 ([JOIN](#))。这样做不但麻烦, 效率相对也比较低。如果建立一个视图, 将相关的表和字段组合在一起, 就可以避免使用[JOIN](#)查询数据。

**灵活。**如果系统中有一张旧的表, 这张表由于设计的问题, 即将被废弃。然而, 很多应用都是基于这张表, 不易修改。这时就可以建立一张视图, 视图中的数据直接映射到[新建](#)的表。这样, 就可以少做很多改动, 也达到了升级数据表的目的。

### 2. 使用语法:

1. **create view** 视图名 **as select语句**
2. **alter view** 视图名 **as select语句** -- **更新成新的视图**
3. **SHOW CREATE VIEW** 视图名
4. **drop view** 视图名1.视图名2

### 3. 细节说明:

2. 视图的数据变化会影响到基表，基表的数据变化也会影响到视图[insert update delete ]

---->针对前面的雇员管理系统-----

```
mysql> create view myview as select empno ,ename ,job, comm from emp;
```

```
mysql> select * from myview;
```

```
mysql> update myview set comm=200 where empno=7369; //修改视图,对基表都有变化
```

```
mysql> update emp set comm=100 where empno=7369; //修改基表, 对视图也有变化
```

3. 视图中可以再使用视图，数据仍然来自基表..【案例演示】

## 7. Mysql管理:

### 1. 用户(users):

mysql中的用户，都存储在系统数据库mysql中 user 表中

| host      | user          | authentication string               |
|-----------|---------------|-------------------------------------|
| localhost | root          | *81220D972A52D4C51BB1C37518A2613... |
| localhost | mysql.session | *THISISNOTAVALIDPASSWORDTHATCANB... |
| localhost | mysql.sys     | *THISISNOTAVALIDPASSWORDTHATCANB... |

其中user表的重要字段说明：

1. host: 允许登录的“位置”，localhost表示该用户只允许本机登录，也可以指定ip地址，比如:192.168.1.100
2. user: 用户名；
3. authentication\_string: 密码，是通过mysql的password()函数加密之后的密码。

```
create user '用户名' @' 允许登录位置' identified by '密码'
```

说明：创建用户，同时指定密码

```
drop user '用户名' @' 允许登录位置' ;
```

修改自己的密码：

```
set password = password('密码');
```

修改他人的密码（需要有修改用户密码权限）：

```
set password for '用户名'@'登录位置' = password('密码');
```

### **基本语法:**

```
grant 权限列表 on 库.对象名 to '用户名' @' 登录位置' 【identified by '密码'】
```

#### **说明:**

1, 权限列表, 多个权限用逗号分开

```
grant select on .....
```

```
grant select, delete, create on .....
```

```
grant all [privileges] on ..... //表示赋予该用户在该对象上的所有权限
```

#### **2.特别说明**

\*.\* : 代表本系统中的所有数据库的所有对象(表, 视图, 存储过程)

库.\* : 表示某个数据库中的所有数据对象(表, 视图, 存储过程等)

3, identified by可以省略, 也可以写出.

(1)如果用户存在, 就是修改该用户的密码。

(2)如果该用户不存在, 就是创建该用户!

### **基本语法:**

```
revoke 权限列表 on 库.对象名 from '用户名'@"登录位置";
```

如果权限没有生效, 可以执行下面命令.

### **基本语法:**

```
FLUSH PRIVILEGES;
```

1. 在创建用户的时候, 如果不指定Host, 则为%, %表示所有IP都有连接权限  
`create user xxx;`

2. 你也可以这样指定

`create user 'xxx'@'192.168.1.%'` 表示 xxx用户在 192.168.1.\*的ip可以登录mysql

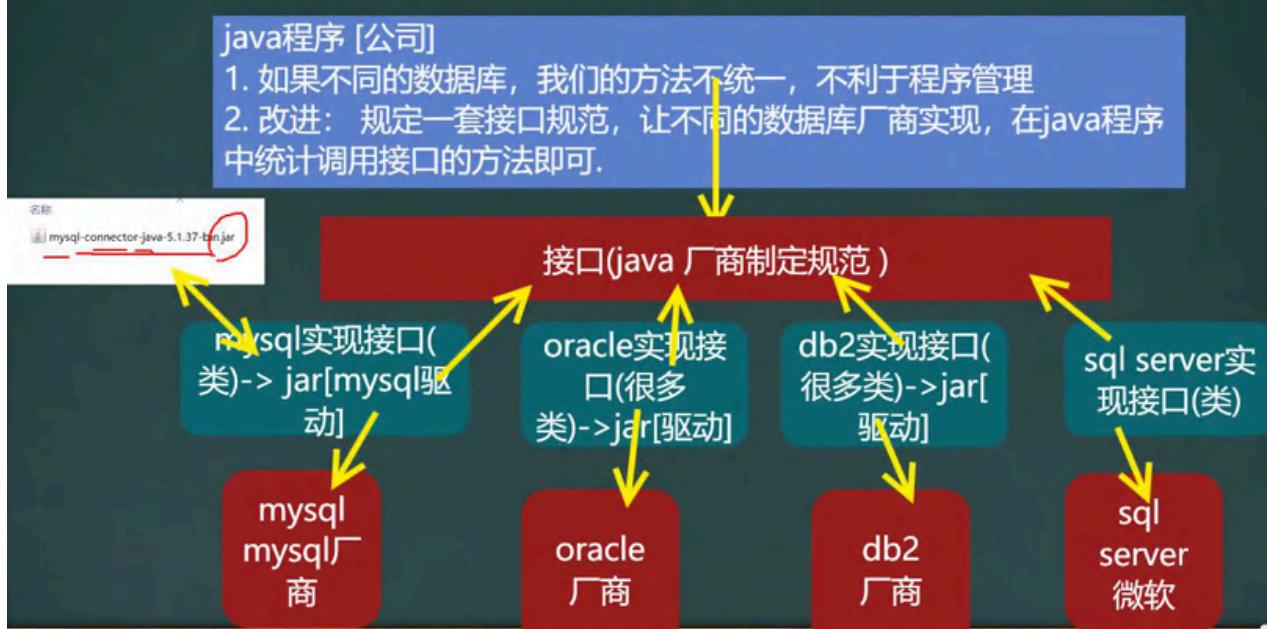
3. 在删除用户的时候, 如果 host 不是 %, 需要明确指定 '用户名'@'host值'

## **2. JDBC**

### **1. JDBC自身介绍:**

- a. 作用:为访问不同数据库提供了统一的接口,可连接任何提供了JDBC驱动的数据库系统
- b. 益处:只需要面向JDBC这一个用于数据库操作的接口API即可,规范了应用程序与数据库的连接
- c. 原理:

## jdbc的原理示意图



## 2. 模拟JDBC:

```
public interface JdbcInterface{  
    //连接  
    public Object getConnection();  
    //crud  
    public void crud();  
    //关闭连接  
    public void close();  
}  
  
//模拟mysql数据库实现jdbc:  
public class MysqlJdbc implements JdbcInterface{  
    @Override  
    public Object getConnection(){  
        System.out.println("得到mysql的连接");  
        return null;  
    }  
    @Override  
    public void crud(){  
        System.out.println("完成mysql增删改查");  
    }  
    @Override  
    public void close(){  
        System.out.println("关闭mysql的连接");  
    }  
}
```

### 3. 获取数据库连接的方式:

a. 用Driver实现类对象(静态加载,灵活差,依赖强,基本不用):

```
Driver driver = new com.mysql.jdbc.Driver();
String url = "jdbc:mysql://localhost:3306/jdbc_db";
Properties info = new Properties();
info.setProperty("user", "root");
info.setProperty("password", "lover");
Connection conn = driver.connect(url, info);
```

b. 通过反射实现(灵活,但速度慢,有安全限制,已不被推荐)

```
Class clazz = Class.forName("com.mysql.jdbc.Driver");
Driver driver = (Driver) clazz.newInstance();
String url = "jdbc:mysql://localhost:3306/jdbc_db";
Properties info = new Properties();
info.setProperty("user", "root");
info.setProperty("password", "abc123");
Connection conn = driver.connect(url, info);
```

c. 用DriverManager替换Driver+反射(动态加载按需,兼容,性能较差)

```
Class clazz = Class.forName("com.mysql.jdbc.Driver");
Driver driver = (Driver) clazz.newInstance();
String url = "jdbc:mysql://localhost:3306/jdbc_db";
String user = "root";
String password = "hsp";
DriverManager.registerDriver(driver);
Connection conn = DriverManager.getConnection(url, user, password);
```

d. 推荐方法,为JDBC4.0及更高版本推荐方法:

```
Class.forName("com.mysql.jdbc.Driver");
String url = "jdbc:mysql://localhost:3306/jdbc_db";
String user = "root";
String password = "hsp";
Connection conn = DriverManager.getConnection(url, user, password);
```

e. 通过配置文件(便于集中管理,环境适应,不过需要根据环境调整):

```
Properties properties = new Properties();
properties.load(new FileInputStream("src\\mysql.properties"));
String user = properties.getProperty("user");
String password = properties.getProperty("password");
String driver = properties.getProperty("driver");
String url = properties.getProperty("url");
Class.forName(driver);
Connection connection = DriverManager.getConnection(url, user, password)
//jdbc.properties:
user=root
password=root
url=jdbc:mysql://localhost:3306/girls
driver=com.mysql.jdbc.Driver
```

## 4. ResultSet(表示结果集的数据表)

### 4.1 介绍:

- a. ResultSet封装了**数据库查询返回的数据表**,并提供了访问查询结果的方法,允许遍历结果集中的每一行
- b. ResultSet可看作**一滚动的游标**(向前,向后,跳转特定行等操作)
- c. 因为ResultSet**存了整个结果集,所以消耗内存大**,频繁让该"游标"前后滚动会使性能下降
- d. **而且依赖于数据库连接,连接关闭,结果集关闭**
- e. 不过也因此,查询灵活,数据安全,可直接对数据库进行更新操作

## 4.2 使用案例:

```
//注册驱动  
Class.forName(driver);  
//得到连接  
Connection connection = DriverManager.getConnection(url, user, password);  
//得到Statement  
Statement statement = connection.createStatement();  
//组织SQL语句,返回单个ResultSet对象  
String sql = "select id, name , sex, borndate from actor"  
//执行SQL语句,返回ResultSet对象  
ResultSet resultSet = statement.executeQuery(sql);  
//光标后移,到底返回false  
while (resultSet.next()) {  
    int id = resultSet.getInt(1);//获取该行的第1列  
    String name = resultSet.getString(2);//获取该行的第2列  
    String sex = resultSet.getString(3);  
    Date date = resultSet.getDate(4);  
}  
//关闭连接,前提是它们不为null  
resultSet.close();  
statement.close();  
connection.close();
```

## 5. Statement(执行静态SQL语句并返回结果的对象ResultSet)

### 1. 介绍:

执行任何类型的SQL语句(DDL和DML),简单易用灵活

但是不提供参数绑定,可能会受到SQL注入攻击,类型不安全,且每次执行SQL语句都要解析编译

建议用PreparedStatement代替

## 6. PreparedStatement(执行静态SQL语句并返回结果的对象ResultSet)

### 1. 介绍:

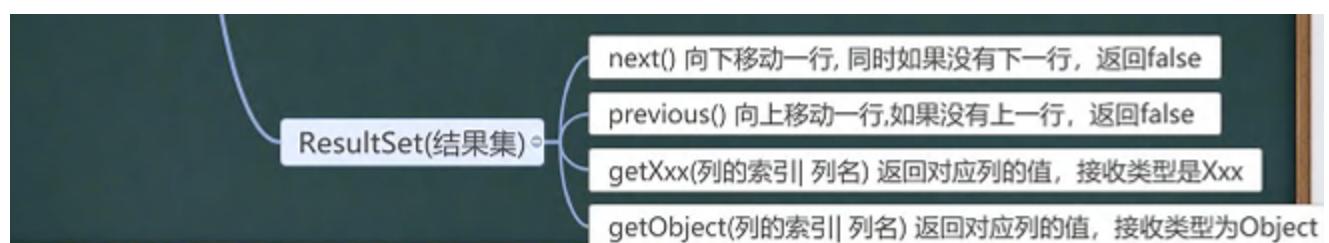
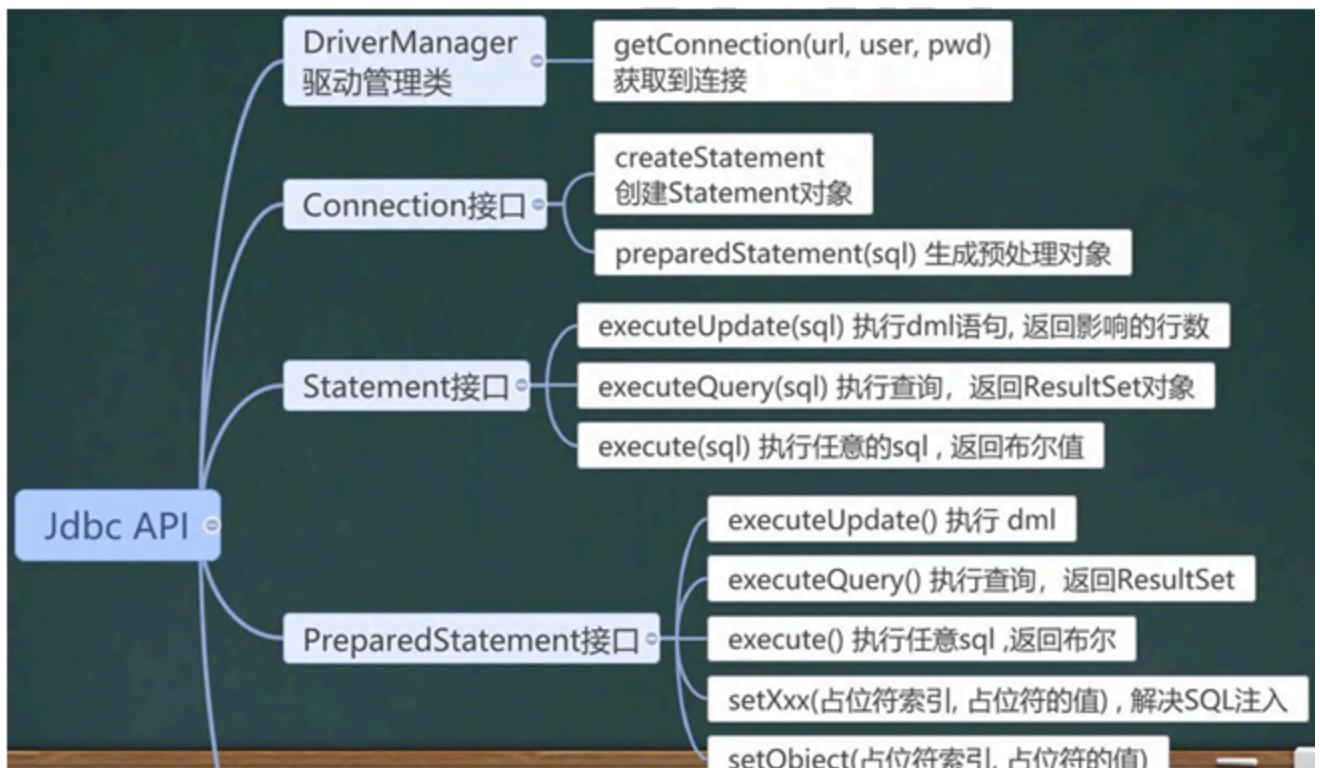
预编译SQL语句,执行同一语句效率高

绑定了参数,可以防止SQL注入,支持批处理,相对复杂一点

## 2. 使用(参考ResultSet中的Statement的前面步骤)

```
String sql = "select name , pwd from admin where name =? and pwd = ?";  
//给占位符?赋值  
PreparedStatement preparedStatement = connection.prepareStatement(sql);  
preparedStatement.setString(1, admin_name);  
preparedStatement.setString(2, admin_pwd);  
//执行select语句用executeQuery  
ResultSet resultSet=preparedStatement.executeQuery(sql);
```

## 7. JDBC相关API小结:



## 8. JDBCUtils的封装(自定义工具类)

```
//在 static 代码块去初始化
static {
    try {
        Properties properties = new Properties();
        properties.load(new FileInputStream("src\\mysql.properties"));
        //读取相关的属性值
        user = properties.getProperty("user");
        password = properties.getProperty("password");
        url = properties.getProperty("url");
        driver = properties.getProperty("driver");
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

## 9. JDBC使用事务示例:

```
Connection connection = null;
String sql = "update account set balance = balance- 100 where id = 1";
String sql2 = "update account set balance = balance + 100 where id = 2";
//创建PreparedStatement对象
PreparedStatement preparedStatement = null;
try {
    //connection默认自动提交
    connection = JDBCUtils.getConnection();

    //将connection设置为不自动提交,开启事务
    connection.setAutoCommit(false);

    preparedStatement = connection.prepareStatement(sql);
    preparedStatement.executeUpdate(); // 执行第 1 条 sql
    int i = 1 / 0; //抛出异常
    preparedStatement = connection.prepareStatement(sql2);
    preparedStatement.executeUpdate();
    connection.commit();
} catch (SQLException e) {
    //此处可进行回滚,即撤销执行的SQL
    //默认回滚到事务开始的状态。
    System.out.println("执行发生了异常,撤销执行的sql");
    try {
        //执行回滚
        connection.rollback();
    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
    e.printStackTrace();
} finally {
    //关闭资源
    JDBCUtils.close(null, preparedStatement, connection);
}
```

## 10. 批处理-JDBC:

```
String sql = "insert into admin2 values(null, ?, ?)";
PreparedStatement preparedStatement = connection.prepareStatement(sql);
//执行5000条
for (int i = 0; i < 5000; i++) {
    preparedStatement.setString(1, "lover" + i);
    preparedStatement.setString(2, "666");
    //加入到批处理包中
    preparedStatement.addBatch();
    //当有1000 条记录时，在批量执行
    if((i + 1) % 1000 == 0) {
        //每满1000条sql执行以此清空
        preparedStatement.executeBatch();
        preparedStatement.clearBatch();
    }
}
//关闭连接
JDBCUtills.close(null, preparedStatement, connection);
```

## 2. 数据库连接池

### 1. 种类:

传统JDBC数据库连接:使用DriverManager,每次建立连接都需要将Connection加载到内存,验证IP地址用户名密码,使用完后断开,**不能控制连接数量.连接过多可能导致内存泄漏**  
**close关闭连接是将Connection对象放回连接池,即重复使用已有的连接**

- a. JDBC的数据库连接池:使用DataSource接口表现,由第三方实现
- b. C3P0数据库连接池:速度慢,稳定性好
- c. DBCP数据库连接池:速度比C3P0较快,但不稳定
- d. Proxool数据库连接池:可以监控连接池状态,稳定性比C3P0较差
- e. BoneCP数据库连接池:速度
- f. Druid(阿里的):集DBCP,C3P0,Proxool优点于一身

### 2. C3P0使用示例:

- a. 指定相关参数

```
import com.mchange.v2.c3p0.ComboPooledDataSource;
...
//创建一个数据源对象
ComboPooledDataSource comboPooledDataSource = new ComboPooledDataSource();
//通过配置文件properties获取相关连接的信息
Properties properties = new Properties();
properties.load(new FileInputStream("src\\mysql.properties"));
//读取相关的属性值
String user = properties.getProperty("user");
...
//给数据源设置相关的参数
//注意：连接管理是由 comboPooledDataSource 来管理
comboPooledDataSource.setDriverClass(driver);
comboPooledDataSource.setJdbcUrl(url);
...
//设置初始化连接数
comboPooledDataSource.setInitialPoolSize(10);
//最大连接数
comboPooledDataSource.setMaxPoolSize(50);
for (int i = 0; i < 5000; i++) {
    //这个方法就是从 DataSource 接口
    Connection connection = comboPooledDataSource.getConnection();
    connection.close();
}
```

b. 通过配置文件：

```
ComboPooledDataSource comboPooledDataSource = new ComboPooledDataSource("lover")
for (int i = 0; i < 500000; i++) {
    //这个方法就是从 DataSource 接口
    Connection connection = comboPooledDataSource.getConnection();
    connection.close();
}
```

### 3. Druid使用示例

使用配置文件

```
import com.alibaba.druid.pool.DruidDataSourceFactory;
Properties properties = new Properties();
properties.load(new FileInputStream("src\\druid.properties"));
DataSource dataSource =
DruidDataSourceFactory.createDataSource(properties);
for (int i = 0; i < 500000; i++) {
    Connection connection = dataSource.getConnection();
    System.out.println(connection.getClass());
    //System.out.println("连接成功!");
    connection.close();
}
```

## 4. DBUtils(简化JDBC操作)

### 特殊章节:JavaBean:

a. 介绍:

JavaBean的主要特点包括：

**公共类（Public Class）**：JavaBean 必须是一个公共类，这意味着它可以在其他类中被访问和实例化。

**无参构造函数**：JavaBean 必须有一个公共的、无参的构造函数，以便在创建JavaBean对象时不需要任何参数。

**私有属性**：JavaBean 的属性通常是私有的，它们只能通过公共的getter和setter方法被外部访问和修改。

**getter和setter方法**：属性通常由一对getter和setter方法来访问和修改。getter方法用于获取属性的值，setter方法用于设置属性的值。

**命名规则**：getter和setter方法的命名遵循特定的规则。getter方法通常以“get”开头，后面跟着属性名（首字母大写），而setter方法通常以“set”开头，后面跟着属性名（首字母大写）。

JavaBean 的作用主要体现在以下几个方面：

**封装**：JavaBean 通过私有属性和公共的getter和setter方法来实现数据的封装，使得外部类只能通过这些方法来访问和修改属性，从而保护了数据的安全性和一致性。

**可重用**：JavaBean 是可重用的组件，可以在不同的应用程序和框架中重复使用，提高了代码的复用性和维护性。

**MVC模式**：在 MVC（模型-视图-控制器）开发模式中，JavaBean 通常作为模型（Model）层的一部分，用于封装数据和业务逻辑。

---

b. 表与JavaBean的关系：

| 表 |                                     |
|---|-------------------------------------|
| 田 | t_department                        |
| 田 | t_employee                          |
| 田 | 栏位                                  |
|   | eid, int(11)                        |
|   | ename, varchar(20)                  |
|   | tel, char(11)                       |
|   | gender, char(1), Nullable           |
|   | salary, double, Nullable            |
|   | commission_pct, double, Nullable    |
|   | birthday, date, Nullable            |
|   | hiredate, date, Nullable            |
|   | job_id, int(11), Nullable           |
|   | email, varchar(32), Nullable        |
|   | mid, int(11), Nullable              |
|   | address, varchar(150), Nullable     |
|   | native_place, varchar(10), Nullable |
|   | did, int(11), Nullable              |

int,double 等在 Java 中都用包装类, 因为 mysql 中的所有类型都可能是 NULL, 而 Java 只有引用数据类型才有 NULL 值

```
public class Employee {
    private Integer eid;
    private String ename;
    private String tel;
    private String gender;//mysql 中用 char, 在 Java 中也要用 String
    private Double salary;
    private Double commissionPct;
    private Date birthday;//此处用 String 或 Date
    private Date hiredate;
    private Integer jobId;
    private String email;
    private Integer mid;
    private String address;
    private String nativePlace;
    private Integer did;
    ...
}
```

## 1. 介绍:

是Apache Commons项目的一部分

a. 优点:

- a. 简化了JDBC操作
- b. 支持自动关闭资源
- c. 支持结果集处理, 将其转换为不同JAVA对象(列表, 映射, 自定义JavaBean)
- d. 支持事务管理, 简化数据库事务操作

b. 缺点:

- a. DbUtils封装可能引入一些性能开销(尤其是处理大数据)

- C.
1. commons-dbutils 是 Apache 组织提供的一个开源 JDBC工具类库, 它是对 JDBC的封装, 使用dbutils能极大简化jdbc编码的工作量[真的]。
    - DbUtils类
    - 1. QueryRunner类: 该类封装了SQL的执行, 是线程安全的。可以实现增、删、改、查、批处理
    - 2. 使用QueryRunner类实现查询
    - 3. ResultSetHandler接口: 该接口用于处理 java.sql.ResultSet, 将数据按要求转换为另一种形式,

ArrayHandler: 把结果集中的第一行数据转成对象数组。

ArrayListHandler: 把结果集中的每一行数据都转成一个数组, 再存放到List中。

BeanHandler: 将结果集中的第一行数据封装到一个对应的JavaBean实例中。

BeanListHandler: 将结果集中的每一行数据都封装到一个对应的JavaBean实例中, 存放到List里。

ColumnListHandler: 将结果集中某一列的数据存放到List中。

KeyedHandler(name): 将结果集中的每行数据都封装到Map里, 再把这些map再存到一个map里, 其key为指定的key。

MapHandler: 将结果集中的第一行数据封装到一个Map里, key是列名, value就是对应的值。

MapListHandler: 将结果集中的每一行数据都封装到一个Map里, 然后再存放到List

## 2 crud使用案例:

```
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.BeanHandler;
import org.apache.commons.dbutils.handlers.BeanListHandler;
import org.apache.commons.dbutils.handlers.ScalarHandler;

//得到 连接 (druid)
Connection connection = JDBCUtilsByDruid.getConnection();
//创建 QueryRunner
//使用 DBUtils 类和接口 , 先引入DBUtils 相关的jar, 加入到本Project
QueryRunner queryRunner = new QueryRunner();

String sql1 = "select id, name from actor where id >= ?";
//query执行相关的方法并返回ArrayList结果集, 使用BeanListHandler
List<Actor> list = queryRunner.query(connection, sql1, new BeanListHandler<>(Actor.class));

String sql2 = "select * from actor where id = ?";
//返回单行记录<-->单个对象 , 使用的Hander 是 BeanHandler
Actor actor = queryRunner.query(connection, sql2, new BeanHandler<>(Actor.class), 10);
System.out.println(actor);

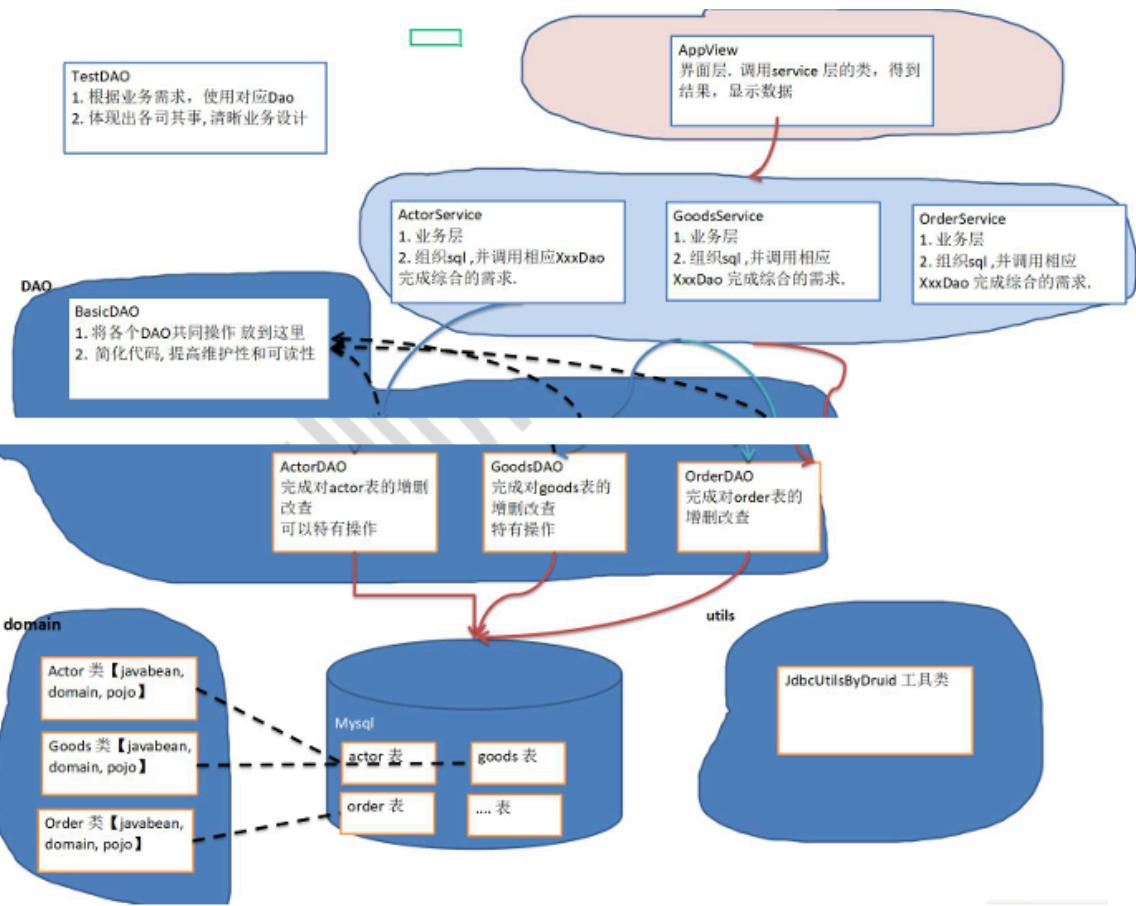
//update返回受影响的行数
String sql = "update actor set name = ? where id = ?";
int affectedRow = queryRunner.update(connection, sql, "林青霞", "女", "1966-10-10", "116");

// 释放资源
JDBCUtilsByDruid.close(null, null, connection);
```

## 5. DAO

### 1. 介绍:

- a. DAO:data access object数据访问对象
- b. crud通用方法BasicDAO:



- c. BasicDAO: 专门与数据库交互(crud)的通用类
- d. 实现诸如:User表-User.java(JavaBean)-UserDao.java

## 2. DAP和POJO和Domain:

**JavaBean** 是一种遵循特定写法的Java类，它主要用于封装数据，并提供getter和setter方法以便于属性的存取。JavaBean的主要特征包括：

- 类是公共的 (public) 。
- 有一个无参的公共构造函数。
- 属性私有 (private) 。
- 属性拥有公共的getter和setter方法。
- 可以有其他方法，如业务逻辑方法。

JavaBean通常用于图形用户界面 (GUI) 开发中，用于绑定数据。

**POJO** (Plain Old Java Object) 是一种简单的Java类，它没有继承任何特殊类或实现任何特殊接口（如Serializable）。POJO的主要目的是尽可能地简单，通常只包含类的getter和setter方法以及属性。

POJO主要用于数据传输对象 (DTO)，在Spring框架中经常使用。

**Domain对象** (领域对象) 是表示应用程序领域概念的对象。它们通常用于表示业务实体，如用户、订单、账户等。Domain对象通常包含业务逻辑和数据。

Domain对象与POJO和JavaBean的主要区别在于，Domain对象专注于业务逻辑和数据模型的表示，而不仅仅是数据的封装。

### 3. BasicDAO使用案例:

dao,utils,javabean(Actor类,此处不展开)

```
//Utils,此处基于druid数据库连接池的工具类
import com.alibaba.druid.pool.DruidDataSourceFactory;
import javax.sql.DataSource;
...
public class JDBCUtilsByDruid {
    private static DataSource ds;
    //在静态代码块完成 ds初始化
    static {
        Properties properties = new Properties();
        try {
            properties.load(new FileInputStream("src\\druid.properties"));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    //编写getConnection 方法
    public static Connection getConnection() throws SQLException {
        return ds.getConnection();
    }
    //close:将Connection对象放回连接池,而非真的断连
    public static void close(ResultSet resultSet, Statement statement, Connection connection) {
        try {
            if (resultSet != null) {
                resultSet.close();
            }
            if (statement != null) {
                statement.close();
            }
            if (connection != null) {
                connection.close();
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
//BasicDAO,为其它DAO父类,使用apache-dbutils
public class BasicDAO<T> { //泛型指定具体类型
    private QueryRunner qr = new QueryRunner();
    //开发通用的dml方法, 针对任意的表
    public int update(String sql, Object... parameters) {
        Connection connection = null;
        try {
            connection = JDBCUtilsByDruid.getConnection();
            int update = qr.update(connection, sql, parameters);
            return update;
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            JDBCUtilsByDruid.close(null, null, connection);
        }
    }

    //查询单行结果 的通用方法
    public T querySingle(String sql, Class<T> clazz, Object... parameters) {
        Connection connection = null;
        try {
            connection = JDBCUtilsByDruid.getConnection();
            return qr.query(connection, sql, new BeanHandler<T>(clazz), parameters);
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            JDBCUtilsByDruid.close(null, null, connection);
        }
    }

    ...
}

//余下同理
}

public class ActorDAO extends BasicDAO<Actor>{
    //1.就有BasicDAO的方法
    //2.根据业务需求, 可以编写特有的方法.
}
```

### 3. mybatis和mybatis plus(拓展,简化开发用)

## 六.AOP(面向切面编程-先行版,正式版在Spring里面)

### 6.1 概念:

AOP(Aspect-Oriented Programming,面向切面编程):

是一种编程范式,将横切关注点(cross-cutting concerns)从主要业务逻辑中分离,提高代码的模块化程度

(日志记录就是一个典型的横切关注点)

### 6.2 组成部分:

- a. 切面(Aspect):LogAspect 类就是一个切面, 它封装了横切关注点 (日志记录) 的逻辑
- b. 连接点(Join Point):程序执行过程中的某个特定位置, 例如方法调用、方法执行等  
//例:每个控制器方法的执行都是一个连接点。
- c. 切入点(Pointcut):定义了哪些连接点会被拦截。在我们的例子中, @Pointcut("execution(\* net.chatmindai.springboot3learn.controller...(..))") 定义了一个切入点, 它匹配所有控制器类中的所有方法。
- d. 通知(Advice):在切入点处要执行的代码。我们使用了 @Around 通知, 它可以在方法执行前后都进行处理。
- e. 引入(Introduction):允许我们向现有的类添加新方法或属性。我们的例子中没有使用这个特性。
- f. 目标对象(Target Object):被一个或者多个切面所通知的对象, 也就是我们的控制器类。
- g. AOP代理(AOP Proxy):AOP框架创建的对象, 用来实现切面契约(包括通知方法执行等功能)

### 6.3 益处:

- a. 集中管理日志记录逻辑, 避免在每个控制器方法中重复编写日志代码。
- b. 轻松地修改或扩展日志记录行为, 而无需修改控制器代码。
- c. 保持控制器代码的整洁, 专注于业务逻辑
- d. 实现了关注点分离, 提高了代码的模块化程度和可维护性

## 6.4 常见应用：

a. 事务管理：

- 自动在方法开始时开启事务，在方法结束时提交或回滚事务。
- 这样可以将事务逻辑与业务逻辑分离，使代码更加清晰。

b. 安全控制：

- 实现方法级别的访问控制，检查用户权限。
- 自动进行身份验证和授权。

c. 性能监控：

- 记录方法的执行时间，识别性能瓶颈。
- 收集方法调用统计信息，用于性能分析。

d. 缓存处理：

- 自动缓存方法返回值，提高系统性能。
- 在数据更新时自动清除相关缓存。

e. 异常处理：

- 统一处理异常，减少重复的 try-catch 代码。
- 自动记录异常信息，便于问题诊断。

f. 重试机制：

- 对于可能失败的操作（如网络请求），自动进行重试。

g. 数据验证：

- 在方法执行前自动验证输入参数。

h. 审计跟踪：

- 记录谁在什么时间执行了什么操作，用于审计目的。

i. 动态代理：

- 实现诸如延迟加载等功能。

j. 资源管理：

- 自动打开和关闭资源，如数据库连接。

k. 多语言支持：

- 根据用户的语言偏好自动切换语言。

l. 分布式追踪：

- 在微服务架构中，跟踪请求在不同服务间的传播。

m. 动态行为修改：

- 在运行时动态修改类的行为，而无需修改源代码。

n. 配置管理：

- 动态注入配置信息，实现配置的集中管理。

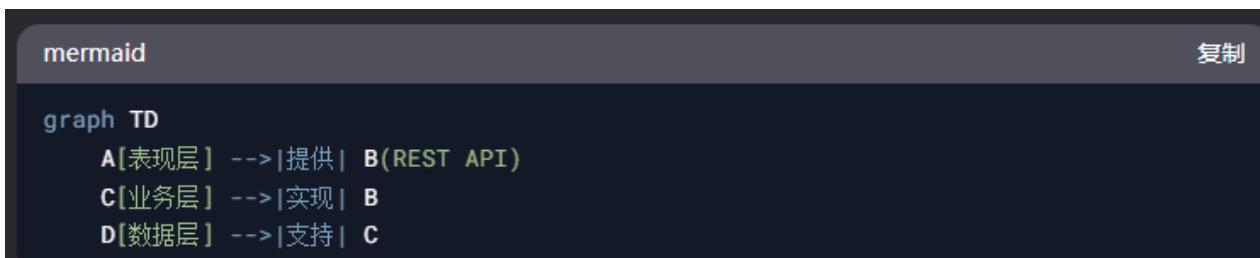
o. 数据脱敏：

- 在日志记录或数据传输时自动对敏感信息进行脱敏处理。

# 特殊章节:API:

受不了了,老是忘记这玩意是什么意思,气气,记记:

- API:**Application Programming Interface**,应用程序编程接口,是一组明确定义的规则和协议
  - 可用的操作 (方法/函数)
  - 输入参数 (请求格式)
  - 输出结果 (响应格式)
  - 通信协议 (如HTTP、gRPC等)
- 四大要素:
  - 端点:访问路径,如/api/user
  - 方法:HTTP操作类型,如GET/POST
  - 参数:Parameters,请求携带的数据,如查询参数/请求体/路径变量
  - 响应:即返回的数据格式,如JSON/XML/二进制流
- 在Spring Framework中:



## 2. 各层职责

| 层级  | 相关Spring组件                      | API相关职责              |
|-----|---------------------------------|----------------------|
| 表现层 | @RestController @RequestMapping | 定义API端点, 处理HTTP请求/响应 |
| 业务层 | @Service @Transactional         | 实现API业务逻辑, 处理事务      |
| 数据层 | @Repository JpaRepository       | 提供数据访问API, 支持CRUD操作  |

- 在不同层级的具体案例:
  - 表现层API:Spring MVC/WebFlux(使用@RestController)
  - 业务层API:Spring Core(使用@Service)
  - 数据层API:MyBatis/JDBC/Spring Data(使用@Repository标记组件)

# 七.Spring(Java登神时刻):

## 特殊章节:八股:

### 7.1 Spring Framework系统框架

#### 1. 核心概念:

- Core Container: 核心容器 (Beans、Core、context、SpEL)
- AOP: 面向切面编程
- Aspects: **AOP思想实现组件, @Aspect 标记, 实现日志、事务、权限等横切关注点**
- Data Access: 数据访问
- Data Integration: 数据集成
- Web: Web开发
- Test: 单元测试和集成测试
- 工厂: **是一种设计模式, 用于封装对象的创建逻辑,数据库连接池复用,多环境支持 (工厂: JDBC,Redis,JMS消息队列)**
- 组件扫描: **Spring扫描组件会自动处理对应的组件标识(如为@Component实例化Bean)**

在如MainApplication.java这个配置类中:

```
//默认扫描
@SpringBootApplication // 隐含 @ComponentScan

//指定路径
@Configuration
@ComponentScan(basePackages = "com.example.service") // 仅扫描指定包
public class AppConfig {
    public static void main(String[] args){
        ...
    }
}
```

- ORM: **Object Relational Mapping,对象映射关系**
- ORM框架: 如MyBaits,MyBatis Plus,Hibernate,通过映射简化数据库操作
- Entity: 字面意思,实体
- Controller: **处理 HTTP 请求的入口, @RestController 标记,调用Service,返回响应**
- Service: **业务逻辑实现层, @Service 标记,字面意思,业务逻辑**

- Configuration: 配置类 `@Configuration` 标记, 定义 Bean 的创建方式 (如第三方库集成)

```
//示例：配置Redis连接
@Configuration
public class RedisConfig {
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory factory) {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(factory);
        template.setKeySerializer(new StringRedisSerializer());
        return template;
    }
}
```

- POJO: Plain Old Java Object, 普通 Java 对象, 仅作为数据载体或工具类 (DTO 是 POJO 的一种)
- DAO: Data Access Object, 数据访问对象, 是设计模式的一种, 负责封装与数据源 (大多情况下是数据库) 的交互逻辑, 将CRUD(增删改查)从业务逻辑剥离, 为不同数据源提供统一访问格式(实现: 传统JDBC DAO, ORM框架如MyBatis, Spring Data JPA), 通常用 `@Repository` 标记 (例如 `UserMapper.java`, 为接口类型, 针对名 user 的数据库的方法储存)

```
// 1. 定义DAO接口
public interface UserDao extends JpaRepository<User, Long> {
    // 自定义查询方法
    List<User> findByAgeGreaterThanOrEqual(int age);
}

// 2. 业务层调用DAO
@Service
public class UserService {
    @Autowired
    private UserDao userDao;
    public List<User> getUsersOverAge(int age) {
        return userDao.findByAgeGreaterThanOrEqual(age);
    }
}
```

- DTO: Data Transfer Object, 数据传输对象(类似于实体, entity), 用于跨层或跨网络传输数据的容器对象, 通常不包含业务逻辑, 仅作为数据的载体, 能减少通信次数并优化传输效率 (实现: 如 MapStruct, 可导入该依赖简化开发)
- 耦合度: 指代码间的依赖程度, 耦合度越高越紧密, 越难以拓展难以修改难以复用

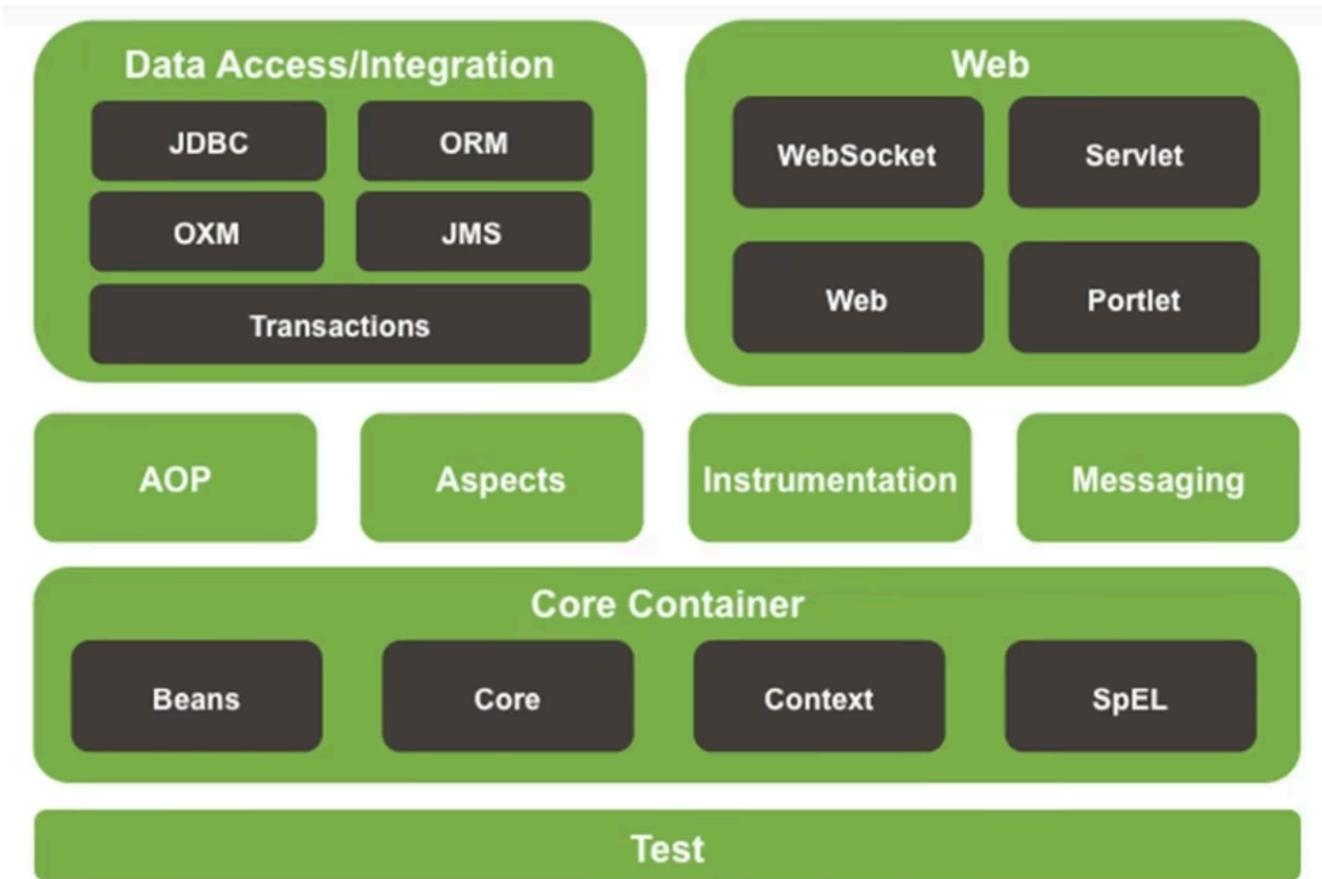
- IoC: **Inversion of Control**, 控制反转, 是一种设计原则, 将对象的创建和管理权限由代码内部转移到外部容器, 实现解耦, 支持动态配置(xml, 注解, 或代码定义依赖关系)(依赖注入是实现其的一种方式)
- DI: **Dependency Injection**, 依赖注入, 可以解耦, 使得耦合度变低, 代码更易复用及单元测试, 而依赖注入举例: 依赖接口, 而非具体实现, 构造函数注入(Spring 注解@Autowired)/Setter 方法注入/字段注入(不建议), (而不是类自己创建对象), 便于测试和配置管理
- Bean: 是IoC容器管理的对象实例(可以是一个方法的返回值)(实例化-属性赋值-初始化 @PostConstruct-使用-销毁@PreDestroy), 对业务逻辑封装, 比如UserService

```

@Service
public class UserService {
    @Autowired
    private UserRepository userRepository; // 依赖注入其他 Bean

    @Transactional
    // 通过 @Transactional 实现数据库事务控制
    public User createUser(UserCreateRequest request) {
        // 业务逻辑: 校验、处理、保存
        User user = new User(request.getUsername(), request.getEmail());
        return userRepository.save(user);
    }
}

```



## 2. IoC(控制反转)

### 2.1 IoC案例:

- 添加Spring的依赖jar包 spring-context
- resources下添加spring配置文件applicationContext.xml//**虽然现在一般都用yml了**
- 在配置文件中注册bean
- 使用Spring提供的接口完成IOC容器的创建：`ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationContext.xml");`
- 使用getBean获取bean进行方法调用：`BookService bookService=(BookService)ctx.getBean("bookservice");bookService.methodName();`

### 2.2 DI案例:

- 删除业务层中使用new方式创建的dao对象
- 在ServiceImpl类中，为dao提供对应的setter方法
- 在配置文件中添加依赖注入的配置：
  - 注意两个bookDao含义不同：1、name中的bookDao作用是让Spring的IOC容器在获取到名称后将首字母大写，前面加set寻找对应的setBookDao方法进行对象注入。2、ref中的作用是让Spring在IOC容器中找到id为bookDao的Bean对象给bookService进行注入。

## 3. bean:

### 3.1 介绍:

- bean的别名
- bean配置：scope属性，singleton为单例，prototype为多例
- bean**默认为单例**,避免对象频繁创建与销毁
- 若对象是有状态对象，即该对象有成员变量可以用来存储数据的，因为所有请求线程共用一个bean，所以存在线程安全问题。封装实例的域对象不适合交给容器进行管理
- 若对象是无状态对象，方法中的局部变量在方法调用完成后会被销毁，所以不会存在线程安全问题。表现层、业务层、数据层、工具对象适合交给容器进行管理。

### 3.2 实例化:

#### 1. 静态工厂实例化:

- 工厂中配置静态方法：

```
public class AppForInstanceOrder {  
    public static void main(String[] args) {  
        //通过静态工厂创建对象  
        OrderDao orderDao = OrderDaoFactory.getOrderDao();  
        orderDao.save();  
    }  
}
```

- 在配置文件中添加

## 2. 实例工厂实例化:

- 在配置文件中添加以下内容，先注册Factory再注册bean

```
<bean id="userFactory" class="com.itheima.factory.UserDaoFactory"/>
```

```
<bean id="userDao" factory-method="getUserDao" factory-bean="userFactory"/>
```

- 先创建实例工厂对象再通过实例工厂对象创建对象

```
//创建实例工厂对象  
UserDaoFactory userDaoFactory = new UserDaoFactory();  
//通过实例工厂对象创建对象  
UserDao userDao = userDaoFactory.getUserDao();  
userDao.save();
```

## 3.3 FactoryBean(简化配置):

- 创建一个UserDaoFactoryBean的类，实现FactoryBean接口，重写接口的方法

```

public class UserDaoFactoryBean implements FactoryBean<UserDao> {
    //代替原始实例工厂中创建对象的方法
    public UserDao getObject() throws Exception {
        return new UserDaoImpl();
    }
    //返回所创建类的Class对象
    public Class<?> getObjectType() {
        return UserDao.class;
    }
}

//配置文件如下：
<bean id="userDao" class="com.itheima.factory.UserDaoFactoryBean"/>
//默认单例，可以在FactoryBean中重写isSingleton() 方法修改返回为false改为多例模式

```

## 4. DI(依赖注入):

- 注解驱动/注解开发（无需手动配置）：

使用 @Autowired、@Component 等注解时，Spring 会自动完成依赖注入，无需手动修改 XML 或配置类(前提是**在配置类中开启组件扫描**)

- XML/Java 配置（需手动配置）：

若使用 XML 或 @Bean 方法显式定义依赖，需手动编写配置

### 4.1 setter注入:

**谨慎使用 Setter 注入：仅用于可选依赖或动态配置**

- a. 引用类型：

- 在bean中定义引用类型属性并提供可访问的set方法
- 配置中使用property标签ref属性注入引用类型对象

- b. 简单类型

- 同上
- 配置中使用property标签value属性注入简单类型数据

```
public class OrderService {  
    private PaymentGateway paymentGateway; // 非 final 字段  
  
    // Setter 方法注入,简单的直接拉到参数列表里面  
    public void setPaymentGateway(PaymentGateway paymentGateway) {  
        this.paymentGateway = paymentGateway;  
    }  
  
    public void processOrder(Order order) {  
        paymentGateway.charge(order.getAmount());  
    }  
}  
  
// 使用示例  
OrderService orderService = new OrderService();  
orderService.setPaymentGateway(new PayPalGateway()); // 动态注入依赖
```

## 4.2 构造器注入:

可以确保依赖不可变且明确

```
public class UserService {  
    private final UserRepository userRepository; // 不可变依赖  
  
    // 构造函数注入,就是构造函数的时候用  
    public UserService(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
  
    public void saveUser(User user) {  
        userRepository.save(user);  
    }  
}  
  
// 使用示例  
UserRepository repository = new JdbcUserRepository();  
UserService userService = new UserService(repository); // 注入依赖
```

## 4.3 字段注入:

避免字段注入：除非在简单原型或框架强制要求下

```

@Component
//该注解会自动注册Bean,将UserService标记为组件
public class ProductService {
    //字段注入,通过@Autowired标识直接表示注入
    @Autowired
    private ProductRepository productRepository;

    public Product findProductById(Long id) {
        return productRepository.findById(id);
    }
}

// Spring 容器自动注入依赖
ProductService productService = context.getBean(ProductService.class);

```

## 4.4 方法注入(特殊场景):

通过普通方法或工厂方法动态获取依赖

```

// 抽象类定义查找方法
public abstract class ReportGenerator {
    // 每次生成新的依赖实例
    @Lookup
    protected abstract DataFetcher createDataFetcher();

    public void generateReport() {
        DataFetcher fetcher = createDataFetcher();
        fetcher.fetchData();
    }
}

// 容器动态实现方法
DataFetcher dataFetcher = new CsvDataFetcher();
ReportGenerator generator = new ReportGenerator() {
    @Override
    protected DataFetcher createDataFetcher() {
        return dataFetcher; // 返回具体实现
    }
};

```

## 5. 纯注解开发模式:

### 5.1 配置与启动类:

注解开发定义bean只需要在配置文件中加一行,所以后续的Spring3.0直接不需要配置文件xml,而将配置作为一个java类放到项目中

```
@Configuration  
@ComponentScan("扫描的路径")  
public class SpringConfig{  
    public static void main(String[] args){  
        ApplicationContext ctx=new AnnotationConfigApplicationContext(SpringConfig.class)  
        //其余一样  
    }  
}
```

### 5.2 bean管理:

- 类前加@Scope("prototype")设置多例
- @PostConstruct设置构造后方法
- @PreDestroy设置销毁前方法

### 5.3 依赖注入:

- @Autowired 按类型装配(Spring原生注解,仅限Spring),需要@Qualifier 指定 Bean 名称(全部注入都支持)

```
@Autowired // 默认按类型注入  
private PaymentService paymentService;  
  
@Autowired  
@Qualifier("wechatPay") // 按名称指定 Bean  
private PaymentService wechatPay;
```

- @Resource 按名称分配(Java标准注解)(仅支持字段和方法注入)

```
@Resource(name = "alipay") // 直接按名称注入  
private PaymentService paymentService;
```

- 不需要setter方法

但是若根据类型在容器中找到多个bean,注入参数的属性名又和容器中bean的名称不一致,就需要使用@Qualifier来指定注入哪个名称的bean对象(注意必须与@Autowired一起使用)

- 简单类型注入:

```
@Value("com.mysql.jdbc.Driver")
private String driver;
@Value("jdbc:mysql://localhost:3306/spring_db")
private String url;
@Value("root")
private String userName;
@Value("root")
private String password;
}

@Bean
public DataSource dataSource(){
    DruidDataSource ds = new DruidDataSource();
    ds.setDriverClassName(driver);
    ds.setUrl(url);
    ds.setUsername(userName);
    ds.setPassword(password);
    return ds;
}
```

## 6. Spring整合Mybatis:

## 6.1 手动配置(复杂,灵活性高)

- MyBatis程序核心对象分析

```
// 1. 创建SqlSessionFactoryBuilder对象  
SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  
// 2. 加载SqlMapConfig.xml 配置文件  
InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");  
// 3. 创建SqlSessionFactory 对象  
SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  
// 4. 获取SqlSession  
SqlSession sqlSession = sqlSessionFactory.openSession();  
// 5. 执行SqlSession 对象执行查询, 获取结果User  
AccountDao accountDao = sqlSession.getMapper(AccountDao.class);  
Account ac = accountDao.findById(2);  
System.out.println(ac);  
// 6. 释放资源  
sqlSession.close();
```

初始化SqlSessionFactory

获取连接, 获取实现

获取数据层接口

关闭连接

- 整合MyBatis

```

<configuration>
    <properties resource="jdbc.properties"></properties>
    <typeAliases>
        <package name="com.itheima.domain"/>
    </typeAliases>
    <environments default="mysql">
        <environment id="mysql">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"></property>
                <property name="url" value="${jdbc.url}"></property>
                <property name="username" value="${jdbc.username}"></property>
                <property name="password" value="${jdbc.password}"></property>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <package name="com.itheima.dao"></package>
    </mappers>
</configuration>

```

初始化属性数据

初始化类型别名

初始化dataSource

初始化映射配置

- 1.项目中导入整合需要的jar包：
  - spring-jdbc、mabatis-spring
- 2.创建Spring的主配置类
- 3.创建数据源的配置类

```

public class JdbcConfig {
    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String userName;
    @Value("${jdbc.password}")
    private String password;

    @Bean
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName(driver);
        ds.setUrl(url);
        ds.setUsername(userName);
        ds.setPassword(password);
        return ds;
    }
}

```

- 4. 主配置类中读 properties 并引入数据源配置类

```

@Configuration
@ComponentScan("com.itheima")
@PropertySource("classpath:jdbc.properties")
@Import(JdbcConfig.class)

```

- 5. 创建 MyBatis 配置类并配置 SqlSessionFactory

```

public class MybatisConfig {
    // 定义 bean, SqlSessionFactoryBean, 用于产生 SqlSessionFactory 对象
    // 使用 SqlSessionFactoryBean 封装 SqlSessionFactory 需要的环境信息
    @Bean
    public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource){
        SqlSessionFactoryBean ssfb = new SqlSessionFactoryBean();
        // 设置模型类的别名扫描
        ssfb.setTypeAliasesPackage("com.itheima.domain");
        // 设置数据源
        ssfb.setDataSource(dataSource);
        return ssfb;
    }
    // 定义 bean, 返回 MapperScannerConfigurer 对象
    @Bean
    public MapperScannerConfigurer mapperScannerConfigurer(){
        MapperScannerConfigurer msc = new MapperScannerConfigurer();
        msc.setBasePackage("com.itheima.dao");
        return msc;
    }
}

```

- 6. 主配置类中引入 MyBatis 配置类

## 6.2 spring boot 自动配置(简单, 灵活性一般)

适用于 Spring Boot 项目, 难以深度定制 MyBatis 底层配置 (需通过 ConfigurationCustomizer 扩展)

- 添加依赖: 直接引入 mybatis-spring-boot-starter。
- 配置参数: 在 application.yml 中配置数据源和 MyBatis 属性。
- 注解驱动: 通过 @Mapper 或 @MapperScan 扫描接口

## 7. Spring整合JUnit:

```
//设置类运行器  
@RunWith(SpringJUnit4ClassRunner.class)  
//设置Spring环境对应的配置类  
@ContextConfiguration(classes = {SpringConfiguration.class}) //加载配置类  
//@ContextConfiguration(locations={"classpath:applicationContext.xml"})//加载配置文件  
public class AccountServiceTest {  
    //支持自动装配注入bean  
    @Autowired  
    private AccountService accountService;  
    @Test  
    public void testFindById(){  
        System.out.println(accountService.findById(1));  
    }  
    @Test  
    public void testfindAll(){  
        System.out.println(accountService.findAll());  
    }  
}
```

## 8. AOP(面向切面编程-正式版)

### 8.1 核心概念:

- 连接点:程序执行过程中的任意位置,粒度为执行方法,抛出异常,设置变量等
  - 在**SpringAOP中**,理解为方法的执行
- 切入点:匹配连接点的式子
  - 在**SpringAOP中**,一个切入点可以只描述一个具体方法,也可以匹配多个方法
- 通知:在切入点执行的操作,也就是共性功能
  - 在**SpringAOP中**,功能最终以方法的形式呈现
- 通知类:定义通知的类
- 切面(Aspect):描述通知与切入点的对应关系

### 8.2 作用:

- 一种编程范式, 指导开发者如何组织程序结构
- 作用: 在不惊动原始设计的基础上为其进行功能增强
- Spring理念:无侵入(减少对应用代码的强制约束和污染,无需适应框架--因为**Spring只需要添加注解即可,不需要继承什么接口**)

### 8.3 实现:

- a. 添加依赖:

```
<!-- Spring AOP 依赖 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

- b. 定义切面类:

```
// 使用 @Aspect 注解标记类，并配置切点和通知
@Aspect
@Component
public class LoggingAspect {

    // 配置切点：匹配所有Service层的public方法
    @Pointcut("execution(public * com.example.service.*.*(..))")
    public void serviceLayer() {}

    // 前置通知
    @Before("serviceLayer()")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("调用方法: " + joinPoint.getSignature().getName());
    }

    // 环绕通知（可控制方法执行）
    @Around("serviceLayer()")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        System.out.println("方法开始: " + joinPoint.getSignature());
        Object result = joinPoint.proceed(); // 执行目标方法
        System.out.println("方法结束: " + joinPoint.getSignature());
        return result;
    }
}
```

- c. 启用注解格式AOP功能:在配置类添加 @EnableAspectJAutoProxy

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {...}
```

- d. 总共工作流程:
  - 1.Spring容器启动
  - 2.读取所有切面配置中的切入点
  - 3.初始化bean，判定bean对应的类中的方法是否匹配到任意切入点
    - 匹配成功，创建原始对象的代理对象
    - 匹配失败，创建对象
  - 4.获取bean执行方法
    - 获取bean，调用方法并执行，完成操作

## 8.4 具体语法及管理:

### 1. 切入点表达式:

- a. 标准格式:动作关键字(访问修饰符 返回值 包名.类/接口名.方法名(参数) 异常名)
- b. \*:单个独立的任意符号，可以独立出现，也可以作为前缀或者后缀的匹配符出现
  - execution(public \* com.itheima..UserService.find(\*))
  - 匹配com.itheima包下的任意包中的UserService类或接口中所有find开头的带有一个参数的方法
- c. ...: 多个连续的任意符号，可以独立出现，常用于简化包名与参数的书写
  - execution(public User com..UserService.findById(..))
  - 匹配com包下的任意包中的UserService类或接口中所有名称为findById的方法
- d. +: 专用于匹配子类类型
  - execution(\* \*..Service+(..))
  - 使用率较低，描述子类的，JavaEE开发继承机会就一次，使用都很慎重，所以很少用它。\*Service+，表示所有以Service结尾的接口的子类
- e. 书写技巧:

对于切入点表达式的编写其实是很灵活的，那么在编写的时候，有没有什么好的技巧让我们用用：

- 所有代码按照标准规范开发，否则以下技巧全部失效
- 描述切入点通常描述接口，而不描述实现类，如果描述到实现类，就出现紧耦合了
- 访问控制修饰符针对接口开发均采用public描述（可省略访问控制修饰符描述）
- 返回值类型对于增删改类使用精准类型加速匹配，对于查询类使用\*通配快速描述
- 包名书写尽量不使用..匹配，效率过低，常用\*做单个包描述匹配，或精准匹配
- 接口名/类名书写名称与模块相关的采用\*匹配，例如UserService书写成\*Service，绑定业务层接口名
- 方法名书写以动词进行精准匹配，名词采用匹配，例如getById书写成getBy,selectAll书写成selectAll
- 参数规则较为复杂，根据业务方法灵活调整
- 通常不使用异常作为匹配规则

## 2. AOP通知类型:

```
public class ClassName {  
  
    public int methodName() {  
        //代码1  
        try{  
            //代码2  
            //原始的业务操作  
            //代码3  
        } catch (Exception e) {  
            //代码4  
        }  
        //代码5  
    }  
}
```

图中对代码进行了标注：  
1. 前置通知添加的位置：指向代码1  
2. 理解为我们要增强的方法：指向代码2  
3. 返回后通知添加的位置：指向代码3  
4. 抛出异常后通知添加的位置：指向代码4  
5. 后置通知添加的位置：指向代码5

- 前置通知@Before()
- 后置通知@After()
- 环绕通知@Around()

- 需要在方法里表示对原始方法的调用:

```
@Around("pt()")
public void around(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("around before advice ...");
    //表示对原始操作的调用
    pjp.proceed();
    System.out.println("around after advice ...");
}
```

- 原始方法有返回值时，需要根据原始方法的返回值来设置环绕通知的返回值

```
@Around("pt2()")
public Object aroundSelect(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("around before advice ...");

    //表示对原始操作的调用
    Object ret = pjp.proceed();

    System.out.println("around after advice ...");
    return ret;
}
```

- 返回后通知@AfterReturning()
- 抛出异常后通知@AfterThrowing
- 案例:通过AOP测试业务执行万次所需时间

```

@Component
@Aspect
public class ProjectAdvice {
    //配置业务层的所有方法
    @Pointcut("execution(* com.itheima.service.*Service.*(..))")
    private void servicePt(){}
    //对ProjectAdvice.servicePt()可以简写为下面的方式
    @Around("servicePt()")
    public void runSpeed(ProceedingJoinPoint pjp){
        //获取执行签名信息
        Signature signature = pjp.getSignature();
        //通过签名获取执行操作名称(接口名)
        String className = signature.getDeclaringTypeName();
        //通过签名获取执行操作名称(方法名)
        String methodName = signature.getName();

        long start = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            pjp.proceed();
        }
        long end = System.currentTimeMillis();
        System.out.println("万次执行: " + className + "." + methodName + "---->" +(end-start) + ' ')
    }
}

```

### 3.AOP通知获取数据:

- 非环绕方式:在方法上添加JoinPoint,通过JoinPoint来获取参数

```

@Component
@Aspect
public class MyAdvice {
    @Pointcut("execution(* com.itheima.dao.BookDao.findName(..))")
    private void pt(){}


    @Before("pt()")
    public void before(JoinPoint jp)
        Object[] args = jp.getArgs();
        System.out.println(Arrays.toString(args));
        System.out.println("before advice ...");
    }
    //...其他的略
}

```

- 环绕方式:使用ProceedingJoinPoint获取参数
  - pjp.proceed()方法有两个构造方法，一个无参一个有参
    - 调用无参构造方法，当原始方法有参数，会在调用过程中自动传入参数
    - 当需要修改原始方法的参数时，只能采用带有参数的方法
- 获取返回值:只有返回后和环绕两个通知类型的返回值可以获取
  - @AfterReturning(value = "pt()",returning = "ret")
- 获取异常:@AfterThrowing(value = "pt()",throwing = "t")

## 9. Spring事务:

### 9.1 实现步骤:

- 需要事务管理的方法上添加注解@Transactional
  - **@Transactional可以写在接口类上、接口方法上、实现类上和实现类方法上**
- **2.在jdbcconfig中配置事务管理器:**

```

//配置事务管理器，mybatis使用的是jdbc事务
@Bean
public PlatformTransactionManager transactionManager(DataSource dataSource){
    DataSourceTransactionManager transactionManager = new DataSourceTransactionManager();
    transactionManager.setDataSource(dataSource);
    return transactionManager;
...
}

```

- **3.在SpringConfig开启事务注解@EnableTransactionManagement**

## 9.2 事务配置:

属性	作用	示例
readOnly	设置是否为只读事务	readOnly=true 只读事务
timeout	设置事务超时时间	timeout = -1 (永不超时)
rollbackFor	设置事务回滚异常 (class)	rollbackFor = {NullPointerException.class}
rollbackForClassName	设置事务回滚异常 (String)	同上格式为字符串
noRollbackFor	设置事务不回滚异常 (class)	noRollbackFor = {NullPointerException.class}
noRollbackForClassName	设置事务不回滚异常 (String)	同上格式为字符串
isolation	设置事务隔离级别	isolation = Isolation.DEFAULT
propagation	设置事务传播行为	.....

上面这些属性都可以在@Transactional注解的参数上进行设置

- readOnly: true只读事务, false读写事务, 增删改要设为false, 查询设为true。
- timeout: 设置**超时时间单位秒**, 在多长时间之内事务没有提交成功就自动回滚, **-1表示不设置超时时间**。
- rollbackFor: 当出现指定异常进行事务回滚
- noRollbackFor: 当出现指定异常不进行事务回滚
  - 思考: 出现异常事务会自动回滚, 这个是之前就已经知道的
  - noRollbackFor是设定对于指定的异常不回滚
  - 而rollbackFor是指定回滚异常, 对于异常事务不应该都回滚么, 为什么还要指定?
    - 这块需要更正一个知识点, **并不是所有的异常都会回滚事务, Spring的事务只会对Error异常和RuntimeException异常及其子类进行事务回滚, 其他的异常类型是不会回滚的, 对应IOException不符合上述条件所以不回滚**

```

public interface AccountService {
    /**
     * 转账操作
     * @param out 传出方
     * @param in 转入方
     * @param money 金额
     */
    //配置当前接口方法具有事务
    public void transfer(String out, String in, Double money) throws IOException
}

@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;
    @Transactional
    //改为@Transactional(rollbackFor = {IOException.class})即可处理该异常
    public void transfer(String out, String in, Double money) throws IOException
        accountDao.outMoney(out, money);
        //int i = 1/0; //这个异常事务会回滚
        if(true){
            throw new IOException(); //这个异常事务不会回滚
        }
        accountDao.inMoney(in, money);
    }
}

```

- rollbackForClassName等同于rollbackFor,只不过属性为异常的类全名字符串
- noRollbackForClassName等同于noRollbackFor, 只不过属性为异常的类全名字符串
- isolation设置事务的隔离级别
  - DEFAULT :默认隔离级别, 会采用数据库的隔离级别
  - READ\_UNCOMMITTED : 读未提交
  - READ\_COMMITTED : 读已提交
  - REPEATABLE\_READ : 重复读取
  - SERIALIZABLE: 串行化(最高级别的事务隔离,强制事务串行执行保证数据一致性)

### 9.3 事务传播:

修改logService改变事务的传播行为:

```

@Service
public class LogServiceImpl implements LogService {
    @Autowired
    private LogDao logDao;

    //propagation设置事务属性：传播行为设置为当前操作需要新事务
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void log(String out, String in, Double money) {
        logDao.log("转账操作由" + out + "到" + in + "，金额：" + money);
    }
}

```

此处代码：不管转账是否成功，都会记录日志

## 7.2 SpringMVC(web层开发技术,同Servlet):

### 1. 案例：

- 1、导入SpringMVC和Servlet坐标
- 2、创建SpringMVC控制类（Controller类）

```

@Controller
public class UserController {

    @RequestMapping("/save")
    public void save(){
        System.out.println("user save ...");
    }
}

```

- 3、创建配置类

```

@Configuration
@ComponentScan("com.itheima.controller")
public class SpringMvcConfig {...}

```

- 4、初始化Servlet容器，加载SpringMVC环境，并设置SpringMVC技术处理的请求(传统手动配置,无需依赖Spring Boot自动配置,但是繁琐, Spring Boot自动配置是现代化开发趋势)
  - 手动配置

```

public class ServletContainersInitConfig extends AbstractDispatcherServletInitializer {
    //手动注册、加载springmvc配置类
    @Override
    protected WebApplicationContext createServletApplicationContext() {
        //初始化WebApplicationContext对象
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();
        //加载指定配置类
        ctx.register(SpringMvcConfig.class);
        return ctx;
    }

    //设置由springmvc控制器处理的请求映射路径
    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

    //加载spring配置类
    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }
}

```

- 自动配置:

```

@SpringBootApplication
// 包含 @Configuration、@EnableAutoConfiguration

public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args); // 自动启动内嵌 Tomcat
    }
}

```

- 5、设置返回数据为json

```
@Controller  
public class UserController {  
    @RequestMapping("/save")  
    @ResponseBody  
    public String save(){  
        System.out.println("user save ...");  
        return "{\"info':'springmvc'}";  
    }  
}
```

## 2. 注解:

- a. @Controller
  - 类注解
  - 设定SpringMVC的核心控制器bean
- b. @RequestMapping
  - 类注解或方法注解
  - 设置当前控制器方法请求访问路径
- c. @ResponseBody
  - 类注解或方法注解
  - 设置当前控制器方法响应内容为当前返回值,无需解析

## 3. 工作流程:

- 启动服务器初始化过程:

- 1. 服务器启动, 执行 ServletContainerInitConfig 类, 初始化 web 容器
  - 2. 执行 createServletApplicationContext 方法, 创建了 WebApplicationContext 对象
  - 3. 加载 SpringMvcConfig
  - 4. 执行 @ComponentScan 加载对应的 bean
  - 5. 加载 UserController, 每个 @RequestMapping 的名称对应一个具体的方法
  - 6. 执行 getServletMappings 方法, 定义所有的请求都通过 SpringMVC
- 单次请求过程:
    - 1. 发送请求 localhost/save
    - 2. web 容器发现所有请求都经过 SpringMVC, 将请求交给 SpringMVC 处理
    - 3. 解析请求路径 /save
    - 4. 由 /save 匹配执行对应的方法 save()
    - 5. 执行 save()
    - 6. 检测到有 @ResponseBody, 直接将 save 方法的返回值作为响应体返回给请求方.

## 4. bean加载控制

### Controller加载控制与业务bean加载控制

- 名称: @ComponentScan
- 类型: **类注解**
- 范例:

```
@Configuration  
@ComponentScan(value = "com.itheima",  
    excludeFilters = @ComponentScan.Filter(  
        type = FilterType.ANNOTATION,  
        classes = Controller.class  
    )  
)  
public class SpringConfig {  
}
```

- 属性

- excludeFilters: 排除扫描路径中加载的bean, 需要指定类别 (type) 与具体项 (classes)
- includeFilters: 加载指定的bean, 需要指定类别 (type) 与具体项 (classes)

- bean的加载格式

```
public class ServletContainersInitConfig extends AbstractDispatcherServletInitializer {  
    protected WebApplicationContext createServletApplicationContext() {  
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();  
        ctx.register(SpringMvcConfig.class);  
        return ctx;  
    }  
    protected WebApplicationContext createRootApplicationContext() {  
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();  
        ctx.register(SpringConfig.class);  
        return ctx;  
    }  
    protected String[] getServletMappings() {  
        return new String[]{"/"};  
    }  
}
```

- 简化开发:

- 简化开发

```
public class ServletContainersInitConfig extends AbstractAnnotationConfigDispatcherServletInitializer{  
    protected Class<?>[] getServletConfigClasses() {  
        return new Class[]{SpringMvcConfig.class};  
    }  
    protected String[] getServletMappings() {  
        return new String[]{"/"};  
    }  
    protected Class<?>[] getRootConfigClasses() {  
        return new Class[]{SpringConfig.class};  
    }  
}
```

## 5. 请求与响应:

### 5.1 乱码处理:

#### 参数传递

#### 乱码处理

```
@Override  
protected Filter[] getServletFilters() {  
    CharacterEncodingFilter filter = new CharacterEncodingFilter();  
    filter.setEncoding("UTF-8");  
    return new Filter[]{filter};  
}
```

### 5.2 传入POJO对象

- 绑定传入参数和形参:@RequestParam("name") String userName
- 传入POJO对象(不强制遵守任何规范,仅使用Java原生语法实现的对象)
- 请求参数名与形参对象属性名相同, 定义POJO类型形参即可接收参数;嵌套POJO参数:  
请求参数名与形参对象属性名相同, 按照对象层次结构关系即可接收嵌套POJO属性参数:

```

    @PostMapping("/saveUserWithAddress")
    public String saveUserWithAddress(User user) {
        System.out.println(user.getAddress().getCity()); // 自动注入嵌套属性
        return "success";
    }

```

### 3. 请求参数格式:

POST /saveUserWithAddress  
 name=李四&address.province=浙江&address.city=杭州

## 参数绑定规则总结

场景	请求参数格式	参数绑定结果
普通POJO属性	name=value	user.setName(value)
嵌套POJO属性	对象.属性=value	user.getAddress().setCity(value)

### 5.3 传入json数据:

- 在SpringMVC配置类中开启@EnableWebMvc注解
- 参数前加@RequestBody

### 5.4 @RequestBody与@RequestParam区别:

- 区别
  - @RequestParam用于接收url地址传参，表单传参【application/x-www-form-urlencoded】
  - @RequestBody用于接收json数据【application/json】
- 应用
  - 后期开发中，发送json格式数据为主，@RequestBody应用较广
  - 如果发送非json格式数据，选用@RequestParam接收请求参数

## 5.5 传入日期型参数:

```
@RequestMapping("/dataParam")
@ResponseBody
public String dataParam(Date date,
                        @DateTimeFormat(pattern="yyyy-MM-dd") Date date1,
                        @DateTimeFormat(pattern="yyyy/MM/dd HH:mm:ss") Date date2)
{
    System.out.println("参数传递 date ==> " + date);
    System.out.println("参数传递 date1(yyyy-MM-dd) ==> " + date1);
    System.out.println("参数传递 date2(yyyy/MM/dd HH:mm:ss) ==> " + date2);
    return "{ 'module': 'data param' }";
}
```

## 5.6 响应:

- 在方法上加上@ResponseBody，会自动将返回值转换成json数据返回给请求者。
- 注解作用：设置当前控制器返回值作为响应体(即return什么东西就返回什么东西作为响应)

# 6. REST(一种软件架构风格):

## 6.1 简介:

- REST (**Representational State Transfer**)，表现形式状态转换,它是一种软件架构**风格**(是一种约定而非规范,可打破)
- 当我们想表示一个网络资源的时候，可以使用两种方式：
  - 传统风格资源描述形式
    - <http://localhost/user/getById?id=1> 查询id为1的用户信息
    - <http://localhost/user/saveUser> 保存用户信息
  - REST风格描述形式
    - <http://localhost/user/1>
    - <http://localhost/user>
- 传统方式一般是一个请求url对应一种操作，麻烦，不安全，可通过读取了请求url地址，就大概知道该url实现的是一个什么样的操作
- 查看REST风格的描述，请求地址变的简单了，并且光看请求URL并不是很能猜出来该URL的具体功能
- 所以REST的优点有：
  - 隐藏资源的访问行为，无法通过地址得知对资源是何种操作
  - 书写简化
- 请求的方式比较多，比较常用的分别是GET,POST,PUT,DELETE
  - 发送GET请求是用来做查询

- 发送POST请求是用来做新增
- 发送PUT请求是用来做修改
- 发送DELETE请求是用来做删除
- RESTful:根据REST风格对资源进行的访问

## 6.2 案例:

```
@RequestMapping(value = "/users/{id}", method = RequestMethod.DELETE)
@ResponseBody
public String delete(@PathVariable Integer id){
    System.out.println("user delete..." + id);
    return "{\"module\":\"user delete\"}";
}
```

@PathVariable 表示绑定路径参数与处理器方法形参间的关系，要求路径参数名与形参名一一对应

```
@RequestBody @RequestParam @PathVariable
```

- 区别
  - @RequestParam用于接收url地址传参或表单传参
  - @RequestBody用于接收json数据
  - @PathVariable用于接收路径参数，使用{参数名称}描述路径参数
- 应用
  - 后期开发中，发送请求参数超过1个时，以json格式为主，@RequestBody应用较广
  - 如果发送非json格式数据，选用@RequestParam接收请求参数
  - 采用RESTful进行开发，当参数数量较少时，例如1个，可以采用@PathVariable接收请求路径变量，通常用于传递id值

## 6.3 具体的知识点:

- @RestController:类注解,设置当前控制器类为RESTful风格，等同于@Controller与@ResponseBody两个注解组合功能
- @GetMapping @PostMapping @PutMapping @DeleteMapping:
  - 方法注解,设置当前控制器方法请求访问路径与请求动作，每种对应一个请求动作，例如@GetMapping对应GET请求
  - 相关属性:value,默认表示请求访问路径

## 7.3 SSM整合(Spring, Spring MVC, MyBatis整合):

### 1. 概况:

- Spring: 统一管理所有组件 (Controller、Service、DAO)
  - 核心功能: 依赖注入 (DI)、面向切面编程 (AOP)、事务管理。
  - 作用: 作为容器整合其他框架, 管理 Bean 的生命周期。
- Spring MVC: 处理 Web 层逻辑
  - 核心功能: 基于 MVC 模式的 Web 框架, 处理 HTTP 请求和响应。
  - 作用: 控制层 (Controller) 实现路由分发、参数绑定、视图渲染。
- MyBatis: 负责数据访问
  - 核心功能: ORM (对象关系映射) 框架, 简化数据库操作。
  - 作用: 持久层 (DAO) 通过 XML/注解将 SQL 映射到 Java 对象。

层级	框架	职责
Web 层	Spring MVC	接收请求、参数校验、返回响应 (JSON/HTML)
Service 层	Spring	业务逻辑处理、事务控制
DAO 层	MyBatis	数据库操作 (增删改查)

### 七、SSM 与 Spring Boot 对比

特性	SSM	Spring Boot
配置方式	手动配置 XML/Java Config	自动配置 (约定大于配置)
启动速度	较慢 (依赖外部服务器)	快 (内嵌 Tomcat)
适用场景	需要深度定制的项目	快速开发、微服务架构

### 八、总结

SSM 整合是 Java 传统企业级开发的核心方案, 适合对架构控制要求高、需灵活定制 SQL 的场景。而 Spring Boot 更适合追求开发效率的现代应用。掌握 SSM 整合, 是理解 Java Web 开发底层逻辑的重要基础!

配置项	SSM 配置方式	Spring Boot 配置方式
项目初始化	手动创建 web.xml、Spring XML 配置文件	通过 Spring Initializr 生成，自动配置依赖
依赖管理	手动添加 Maven/Gradle 依赖	使用 starter 依赖（如 spring-boot-starter-web）
数据源配置	在 XML 中定义 DataSource 和 SqlSessionFactory	application.properties 中设置数据源参数
事务管理	手动配置 PlatformTransactionManager	自动配置 DataSourceTransactionManager
视图解析器	手动配置 InternalResourceViewResolver	默认集成 Thymeleaf/FreeMarker，自动配置视图

内嵌服务器	需外部部署 Tomcat	内嵌 Tomcat/Jetty，通过 spring-boot-starter-web 自动启用
日志配置	手动集成 Log4j/SLF4J	默认使用 Logback，通过 application.properties 配置

- 数据源和MyBatis:

```
<!-- spring-dao.xml -->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="url" value="jdbc:mysql://localhost:3306/test"/>
    <property name="username" value="root"/>
    <property name="password" value="123456"/>
</bean>

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mapperLocations" value="classpath:mapper/*.xml"/>
</bean>
```

## Spring Boot (自动配置 + 属性文件) :

### Properties

```
# application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
mybatis.mapper-locations=classpath:mapper/*.xml
```

- 事务管理:

- **SSM** (手动声明事务管理器) :

XML

```
<!-- spring-service.xml -->
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
<tx:annotation-driven transaction-manager="transactionManager"/>
```

- **Spring Boot** (自动启用事务) :

无需配置，直接通过 `@Transactional` 注解使用：

Java

```
@Service
public class UserService {
    @Transactional
    public void updateUser(User user) { ... }
}
```

- Spring MVC:

- SSM (手动配置视图解析器) :

XML

```
<!-- spring-mvc.xml -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>
```

- Spring Boot (默认约定或通过属性配置) :

Properties

```
# application.properties
spring.mvc.view.prefix=/templates/
spring.mvc.view.suffix=.html
```

- 日志配置:

- **SSM** (手动集成 Log4j) :

#### XML

```
<!-- log4j.xml -->
<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d{yyyy-MM-dd HH:mm:ss} %-5p
%c{1}:%L - %m%n"/>
    </layout>
</appender>
<root>
    <priority value="INFO"/>
    <appender-ref ref="CONSOLE"/>
</root>
```

- **Spring Boot** (默认使用 Logback, 通过属性文件配置) :

#### Properties

```
# application.properties
logging.level.root=INFO
logging.file.name=app.log
```

- 总架构对比:

## 传统SSM架构

复制

表现层(Spring MVC) → 业务层(Spring) → 持久层(MyBatis)



## Spring Boot整合架构

复制

表现层(Spring Boot Web) → 业务层(Spring) → 持久层(MyBatis/MyBatis-Plus)



## 总结对比表

组件	传统SSM实现方式	Spring Boot整合方式	优势改进
表现层	@Controller + XML配置	@RestController + 自动配置	简化配置，支持 RESTful
业务层	@Service + 大量 XML配置	@Service + 自动注入	依赖注入更简洁
持久层	MyBatis + 大量 XML配置	MyBatis + 注解/XML + 自动配置	灵活选择注解或 XML
配置	多个XML文件	application.yml/application.properties	配置集中化，更易维护
事务	XML声明式事务	@Transactional注解	更直观，代码耦合度低
启动	外部Tomcat + web.xml	内嵌服务器 + 主类main方法	部署简单，独立应用
通过Spring Boot整合SSM框架，可以大幅减少XML配置，利用自动配置和约定优于配置的原则，使项目结构更清晰，开发效率更高。			

## 2. SSM整合环境步骤：

- 创建SpringConfig配置类：

```
@Configuration  
@ComponentScan({"com.itheima.service"})  
@PropertySource("classpath:jdbc.properties")  
@Import({JdbcConfig.class, MyBatisConfig.class})  
@EnableTransactionManagement  
public class SpringConfig {...}
```

- 创建JdbcConfig配置类：

```
public class JdbcConfig {  
    @Value("${jdbc.driver}")  
    private String driver;  
    @Value("${jdbc.url}")  
    private String url;  
    @Value("${jdbc.username}")  
    private String username;  
    @Value("${jdbc.password}")  
    private String password;  
  
    @Bean  
    public DataSource dataSource(){  
        DruidDataSource dataSource = new DruidDataSource();  
        dataSource.setDriverClassName(driver);  
        dataSource.setUrl(url);  
        dataSource.setUsername(username);  
        dataSource.setPassword(password);  
        return dataSource;  
    }  
  
    @Bean  
    public PlatformTransactionManager transactionManager(DataSource dataSource){  
        DataSourceTransactionManager ds = new DataSourceTransactionManager();  
        ds.setDataSource(dataSource);  
        return ds;  
    }  
}
```

- 创建MybatisConfig配置类:

```

public class MyBatisConfig {
    @Bean
    public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource){
        SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();
        factoryBean.setDataSource(dataSource);
        factoryBean.setTypeAliasesPackage("com.itheima.domain");
        return factoryBean;
    }

    @Bean
    public MapperScannerConfigurer mapperScannerConfigurer(){
        MapperScannerConfigurer msc = new MapperScannerConfigurer();
        msc.setBasePackage("com.itheima.dao");
        return msc;
    }
}

```

- 创建jdbc.properties:在resources下提供jdbc.properties,设置数据库连接四要素

```

jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/ssm_db
jdbc.username=root
jdbc.password=root

```

- 创建SpringMVC配置类:

```

@Configuration
@ComponentScan("com.itheima.controller")
@EnableWebMvc
public class SpringMvcConfig {...}

```

- 创建Web项目入口配置类:

```
public class ServletConfig extends AbstractAnnotationConfigDispatcherServletInitializer {
    //加载Spring配置类
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{SpringConfig.class};
    }
    //加载SpringMVC配置类
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{SpringMvcConfig.class};
    }
    //设置SpringMVC请求地址拦截规则
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
    //设置post请求中文乱码过滤器
    @Override
    protected Filter[] getServletFilters() {
        CharacterEncodingFilter filter = new CharacterEncodingFilter();
        filter.setEncoding("utf-8");
        return new Filter[]{filter};
    }
}
```

### 3. 测试接口：

JUnit和Postman分别测试业务层和表现层

## 4. 表现层数据封装:

- 前端接收数据格式—封装特殊消息到message(msg)属性中

- 增删改

```
{  
    "code": 20031,  
    "data": true  
}
```

- 查单条

```
{  
    "code": 20040,  
    "data": null,  
    "msg": "数据查询失败，请重试！"  
}
```

- 查全部

```
{  
    "code": 20041,  
    "data": [  
        {  
            "id": 1,  
            "type": "计算机理论",  
            "name": "Spring实战 第5版",  
            "description": "Spring入门经典教程"  
        },  
        {  
            "id": 2,  
            "type": "计算机理论",  
            "name": "Spring 5核心原理与30个类手写实战",  
            "description": "十年沉淀之作"  
        }  
    ]  
}
```

用户得到什么信息？

- 设置统一数据返回结果类

- 设置统一数据返回结果类

```
public class Result {  
    private Object data;  
    private Integer code;  
    private String msg;  
}
```

- 设置统一数据返回结果类

```

public class Result {
    //描述统一格式中的数据
    private Object data;
    //描述统一格式中的编码，用于区分操作，可以简化配置0或1表示成功失败
    private Integer code;
    //描述统一格式中的消息，可选属性
    private String msg;

    //构造方法是方便对象的创建
    public Result() {
    }
    public Result(Integer code, Object data) {
        this.data = data;
        this.code = code;
    }
    public Result(Integer code, Object data, String msg) {
        this.data = data;
        this.code = code;
        this.msg = msg;
    }
    //setter...getter...省略
}

```

- 设置返回码code类:

```

//状态码
public class Code {
    public static final Integer SAVE_OK = 20011;
    public static final Integer DELETE_OK = 20021;
    public static final Integer UPDATE_OK = 20031;
    public static final Integer GET_OK = 20041;

    public static final Integer SAVE_ERR = 20010;
    public static final Integer DELETE_ERR = 20020;
    public static final Integer UPDATE_ERR = 20030;
    public static final Integer GET_ERR = 20040;
}

```

- 修改同步Controller的返回值:

```
//统一每一个控制器方法返回值Result(data,code,msg)
@RestController
@RequestMapping("/books")
public class BookController {

    @Autowired
    private BookService bookService;

    @PostMapping
    public Result save(@RequestBody Book book) {
        boolean flag = bookService.save(book);
        return new Result(flag ? Code.SAVE_OK:Code.SAVE_ERR,flag);
    }

    @PutMapping
    public Result update(@RequestBody Book book) {
        boolean flag = bookService.update(book);
        return new Result(flag ? Code.UPDATE_OK:Code.UPDATE_ERR,flag);
    }

    @DeleteMapping("/{id}")
    public Result delete(@PathVariable Integer id) {
        boolean flag = bookService.delete(id);
        return new Result(flag ? Code.DELETE_OK:Code.DELETE_ERR,flag);
    }

    @GetMapping("/{id}")
    public Result getById(@PathVariable Integer id) {
        Book book = bookService.getById(id);
        Integer code = book != null ? Code.GET_OK : Code.GET_ERR;
        String msg = book != null ? "" : "数据查询失败, 请重试!";
        return new Result(code,book,msg);
    }

    @GetMapping
    public Result getAll() {
        List<Book> bookList = bookService.getAll();
        Integer code = bookList != null ? Code.GET_OK : Code.GET_ERR;
        String msg = bookList != null ? "" : "数据查询失败, 请重试!";
        return new Result(code,bookList,msg);
    }
}
```

## 5. 异常处理器:

### 5.1 常见异常:

- 框架内部抛出的异常：因使用不合规导致
- 数据层抛出的异常：因外部服务器故障导致（例如：服务器访问超时）
- 业务层抛出的异常：因业务逻辑书写错误导致（例如：遍历业务书写操作，导致索引异常等）
- 表现层抛出的异常：因数据收集、校验等规则导致（例如：不匹配的数据类型间导致异常）
- 工具类抛出的异常：因工具类书写不严谨不够健壮导致（例如：必要释放的连接长期未释放等）

### 5.2 异常处理器:

- 异常处理器
  - 集中的、统一的处理项目中出现的异常

```
@RestControllerAdvice  
public class ProjectExceptionAdvice {  
    @ExceptionHandler(Exception.class)  
    public Result doException(Exception ex){  
        return new Result(666,null);  
    }  
}
```

徐易昊

### 5.3 项目异常分类及处理方案:

- 业务异常(传递对应消息,提醒规范)
  - 规范的用户行为产生的异常
  - 不规范的用户行为操作产生的异常
- 系统异常(发送消息安抚,提醒维护,记录日志)
  - 项目运行过程中可预计且无法避免的异常
- 其他异常
  - 编程人员未预期到的异常

### 5.4 步骤:

- 1. 自定义异常类:

```
//自定义异常处理器，封装系统异常信息
public class SystemException extends RuntimeException{
    private Integer code;

    public Integer getCode() {
        return code;
    }

    public void setCode(Integer code) {
        this.code = code;
    }

    public SystemException(Integer code, String message) {
        super(message);
        this.code = code;
    }

    public SystemException(Integer code, String message, Throwable cause) {
        super(message, cause);
        this.code = code;
    }

}

//自定义异常处理器，封装业务异常信息
public class BusinessException extends RuntimeException{
    private Integer code;

    public Integer getCode() {
        return code;
    }

    public void setCode(Integer code) {
        this.code = code;
    }

    public BusinessException(Integer code, String message) {
        super(message);
        this.code = code;
    }

    public BusinessException(Integer code, String message, Throwable cause) {
        super(message, cause);
        this.code = code;
    }
}
```

```
    }  
}
```

- 2. 将其他异常包装成自定义异常

```
public Book getById(Integer id) {  
    //模拟业务异常， 包装成自定义异常  
    if(id == 1){  
        throw new BusinessException(Code.BUSINESS_ERR,"请不要使用你的技术挑战我的耐性!");  
    }  
    //模拟系统异常， 将可能出现的异常进行包装， 转换成自定义异常  
    try{  
        int i = 1/0;  
    }catch (Exception e){  
        throw new SystemException(Code.SYSTEM_TIMEOUT_ERR,"服务器访问超时，请重试!",e)  
    }  
    return bookDao.getById(id);  
}
```

- 3. 处理器类中处理自定义异常:

```
//@RestControllerAdvice标识当前类为REST风格的异常处理器
@RestControllerAdvice
public class ProjectExceptionAdvice {

    //ExceptionHandler(..):设置当前处理器类对应的异常类型
    @ExceptionHandler(SystemException.class)
    public Result doSystemException(SystemException ex){
        //记录日志
        //发送消息给运维
        //发送邮件给开发人员,ex对象发送给开发人员
        return new Result(ex.getCode(),null,ex.getMessage());
    }

    @ExceptionHandler(BusinessException.class)
    public Result doBusinessException(BusinessException ex){
        return new Result(ex.getCode(),null,ex.getMessage());
    }

    //除了自定义的异常处理器,保留对Exception类型的异常处理,用于处理非预期的异常
    @ExceptionHandler(Exception.class)
    public Result doOtherException(Exception ex){
        //记录日志
        //发送消息给运维
        //发送邮件给开发人员,ex对象发送给开发人员
        return new Result(Code.SYSTEM_UNKNOW_ERR,null,"系统繁忙,请稍后再试!");
    }
}
```

## 6. SpringMVC放行静态资源

- 新建SpringMvcSupport
-

```

@Configuration
public class SpringMvcSupport extends WebMvcConfigurationSupport {
    @Override
    protected void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/pages/**").addResourceLocations("/pages/");
        registry.addResourceHandler("/css/**").addResourceLocations("/css/");
        registry.addResourceHandler("/js/**").addResourceLocations("/js/");
        registry.addResourceHandler("/plugins/**").addResourceLocations("/plugins/");
    }
}

```

- 在SpringMvcConfig中扫描SpringMvcSupport

## 7. 拦截器:

### 7.1 简介:

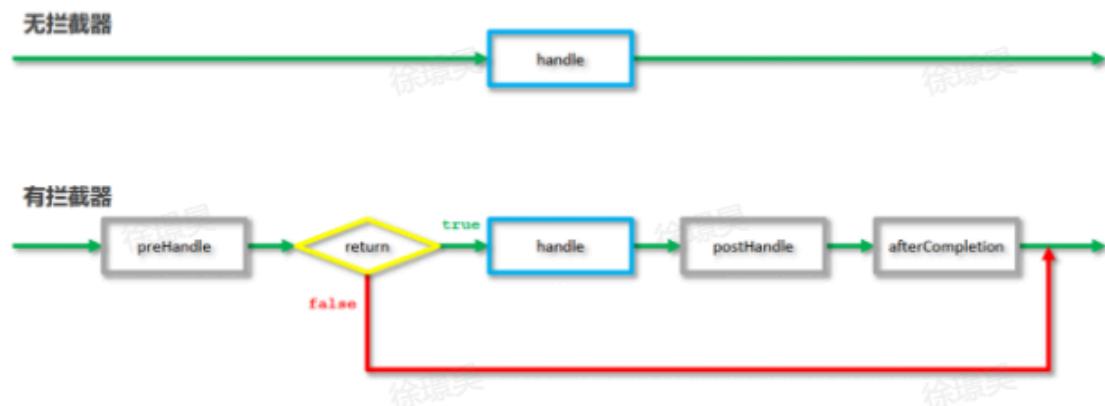
- 拦截器是一种动态拦截方法调用的机制,在SpringMVC中动态拦截控制器方法的执行.
- 作用:在指定方法调用前后执行预先设定后的代码\阻止原始方法的执行
- 参数:
  - request:请求对象,获取请求数据中的内容, 如获取请求头的Content-Type
  - response:响应对象
  - handler:被调用的处理器对象, 本质上是一个方法对象, 对反射中的Method对象进行了再包装,获取方法的相关信息

场景	拦截器实现逻辑
用户认证	在 preHandle() 中校验 Token, 未登录则返回 401 错误。
权限控制	校验用户角色是否匹配接口权限, 无权限则返回 403 错误。
日志记录	在 preHandle() 记录请求开始时间, 在 afterCompletion() 计算耗时并写入日志。
防重复提交	在 preHandle() 中生成唯一 Token, 提交后校验 Token 是否有效。

### 6. 拦截器与过滤器的区别

对比项	拦截器 (Interceptor)	过滤器 (Filter)
归属	Spring MVC 框架的一部分	Servlet 规范的一部分
作用范围	仅拦截 Controller 请求	拦截所有请求 (包括静态资源)
依赖	依赖 Spring 容器	不依赖 Spring, 基于 Servlet API
执行顺序	在过滤器之后、Controller 之前执行	在拦截器之前执行

## 拦截器的执行流程:



当有拦截器后，请求会先进入preHandle方法，

如果方法返回true，则放行继续执行后面的handle[controller的方法]和后面的方法

如果返回false，则直接跳过后面方法的执行。

## 7.2 拦截器拦截规则:

- 拦截器类:

```
//定义拦截器类，实现HandlerInterceptor接口
//注意当前类必须受Spring容器控制
@Component
public class ProjectInterceptor implements HandlerInterceptor {
    @Override
    //原始方法调用前执行的内容
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response) {
        System.out.println("preHandle...");
        return true;
    }

    @Override
    //原始方法调用后执行的内容
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
                           System.out.println("postHandle..."));
    }

    @Override
    //原始方法调用完成后执行的内容
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response) {
        System.out.println("afterCompletion...");
    }
}
```

- 拦截器配置：

```

@Configuration
public class SpringMvcSupport extends WebMvcConfigurerSupport {
    @Autowired
    private ProjectInterceptor projectInterceptor;

    @Override
    protected void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/pages/**").addResourceLocations("/pages/");
    }

    //InterceptorRegistry:拦截器注册中心
    @Override
    protected void addInterceptors(InterceptorRegistry registry) {
        //将自定义拦截器注册到Spring MVC中
        //addPathPatterns("/books","/books/*");设置拦截路径,/**拦截所有请求
        // /api/**: 拦截所有api开头的请求
        registry.addInterceptor(projectInterceptor).addPathPatterns("/books","/books/");
    }
}

```

- 简化SpringMvcSupport编写:

```

@Configuration
@ComponentScan({"com.itheima.controller"})
@EnableWebMvc
//实现WebMvcConfigurer接口可以简化开发，但具有一定的侵入性
public class SpringMvcConfig implements WebMvcConfigurer {
    @Autowired
    private ProjectInterceptor projectInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //配置多拦截器
        registry.addInterceptor(projectInterceptor).addPathPatterns("/books","/books/");
    }
}

```

## 7.3 拦截器参数:

- 前置处理方法:(字面意思,原始方法运行之前运行)

```
public boolean preHandle(HttpServletRequest request,
                         HttpServletResponse response,
                         Object handler) throws Exception {
    System.out.println("preHandle");
    HandlerMethod hm = (HandlerMethod)handler;
    //获取方法名称
    String methodName = hm.getMethod().getName();
    System.out.println("preHandle..."+methodName);
    return true;
}
```

- 后置处理方法:(字面意思,原始方法运行后运行, 如果原始方法被拦截, 则不执行)

```
//modelAndView:如果处理器执行完成具有返回结果, 可以读取到对应数据与页面信息, 并进行调整
//不过现在返回的都是json, 该参数使用率不高
public void postHandle(HttpServletRequest request,
                        HttpServletResponse response,
                        Object handler,
                        ModelAndView modelAndView) throws Exception {
    System.out.println("postHandle");
}
```

- 完成处理方法:(拦截器最后执行的方法, 无论原始方法是否执行,类似于finally)

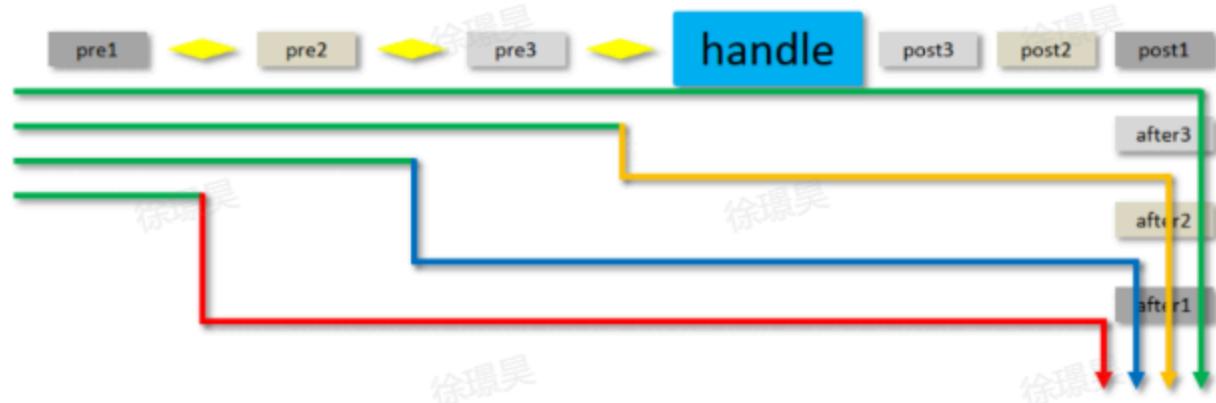
```
public void afterCompletion(HttpServletRequest request,
                            HttpServletResponse response,
                            Object handler,
                            Exception ex) throws Exception {
    System.out.println("afterCompletion");
}
```

## 多个拦截器

拦截器执行的顺序是和配置顺序有关。就和前面所提到的运维人员进入机房的案例，先进后出。

- 当配置多个拦截器时，形成拦截器链
- 拦截器链的运行顺序参照拦截器添加顺序为准
- 当拦截器中出现对原始处理器的拦截，后面的拦截器均终止运行
- 当拦截器运行中断，仅运行配置在前面的拦截器的 afterCompletion 操作

■ 按照1、2、3的顺序配置



preHandle: 与配置顺序相同，必定运行

postHandle: 与配置顺序相反，可能不运行

afterCompletion: 与配置顺序相反，可能不运行。

- **路径排除:** 避免拦截静态资源或登录接口。

Java

```
.addPathPatterns("/**")
.excludePathPatterns("/login", "/css/**", "/js/**");
```

- **多拦截器顺序:** 通过 `order()` 方法设置优先级。

Java

```
registry.addInterceptor(A).order(1);
registry.addInterceptor(B).order(2); // B 在 A 之后执行
```

## 7.4 Spring Boot(现代化开发,简化Spring)

### 1. 推出作用:

简化Spring应用的初始搭建以及开发过程

SpringBoot配置文件优先级:properties>yml>yaml

### 2. yaml格式:

容易阅读,以数据为核心,重数据轻格式

扩展名: .yml(主流), .yaml

#### 2.1 语法规则:

- 大小写敏感
- 属性层级使用多行描述,每行结尾使用冒号结束
- 使用缩进表示层级关系,同层级左侧对齐,只允许使用空格不能使用tab
- 属性值之前添加空格
- #表示注释
- 数组格式加减号-
- 使用 `@Value("表达式")` 注解可以从配合文件中读取数据, 注解中用于读取属性名引用方式是: \${一级属性名.二级属性名.....},例如:

```

@RestController
@RequestMapping("/books")
public class BookController {

    @Value("${lesson}")
    private String lesson;
    @Value("${server.port}")
    private Integer port;
    @Value("${enterprise.subject[0]}")
    private String subject_00;

    @GetMapping("/{id}")
    public String getById(@PathVariable Integer id){
        System.out.println(lesson);
        System.out.println(port);
        System.out.println(subject_00);
        return "hello , spring boot!";
    }
}

```

## 2.2: Environment对象:

- 上面方式读取到的数据特别零散，SpringBoot 还可以使用 @Autowired 注解注入 Environment 对象的方式读取数据。这种方式 SpringBoot 会将配置文件中所有的数据封装到 Environment 对象中，如果需要使用哪个数据只需要通过调用 Environment 对象的 getProperty(String name) 方法获取

```

@RestController
@RequestMapping("/books")
public class BookController {

    @Autowired
    private Environment env;

    @GetMapping("/{id}")
    public String getById(@PathVariable Integer id){
        System.out.println(env.getProperty("lesson"));
        System.out.println(env.getProperty("enterprise.name"));
        System.out.println(env.getProperty("enterprise.subject[0]"));
        return "hello , spring boot!";
    }
}

```

- SpringBoot 还提供了将配置文件中的数据封装到我们自定义的实体类对象中的方式。具体操作如下：**(自定义实体类存储数据)**

- 将实体类 bean 的创建交给 Spring 管理。
- 在类上添加 @Component 注解
- 使用 @ConfigurationProperties 注解表示加载配置文件
- 在该注解中也可以使用 prefix 属性指定只加载指定前缀的数据
- 在 BookController 中进行注入

```

//实体类
@Component
@ConfigurationProperties(prefix = "enterprise")
public class Enterprise {
    private String name;
    private int age;
    private String tel;
    private String[] subject;

    @Override
    public String toString() {
        return "Enterprise{" +
            "name='" + name + '\'' +
            ", age=" + age +
            ", tel='" + tel + '\'' +
            ", subject=" + Arrays.toString(subject) +
            '}';
    }
    //setter/getter其他代码
}

//Controller类
@RestController
@RequestMapping("/books")
public class BookController {

    @Autowired
    private Enterprise enterprise;

    @GetMapping("/{id}")
    public String getById(@PathVariable Integer id){
        System.out.println(enterprise.getName());
        System.out.println(enterprise.getAge());
        System.out.println(enterprise.getSubject());
        System.out.println(enterprise.getTel());
        System.out.println(enterprise.getSubject()[0]);
        return "hello , spring boot!";
    }
}

```

自定义方法警告提示及解决方法:

==注意：==

使用第三种方式，在实体类上有如下警告提示



```
4 import org.springframework.stereotype.Component;
5
6 import java.util.Arrays;
7
8 @Component
9 @ConfigurationProperties(prefix = "enterprise")
10 public class Enterprise {
11     private String name;
12     private int age;
```

这个警告提示解决是在 `pom.xml` 中添加如下依赖即可

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-configuration-processor
4     <optional>true</optional>
5 </dependency>
```

### 3. 多环境：

#### 1. yaml文件：

```
spring:
  config:
    activate:
      on-profile: dev
```

#### 2. properties文件：

properties 类型的配置文件配置多环境需要定义不同的配置文件

- `application-dev.properties` 是开发环境的配置文件。我们在该文件中配置端口号为 80, 配置: `server.port=80`
- `application-test.properties` 是测试环境的配置文件。我们在该文件中配置端口号为 81, 配置: `server.port=81`

- application-pro.properties 是生产环境的配置文件。我们在该文件中配置端口号为 82, 配置: server.port=82
- SpringBoot 只会默认加载名为 application.properties 的配置文件, 所以需要在 application.properties 配置文件中设置启用哪个配置文件, 配置如下:  
spring.profiles.active=pro

## 4. SpringBoot整合JUnit:

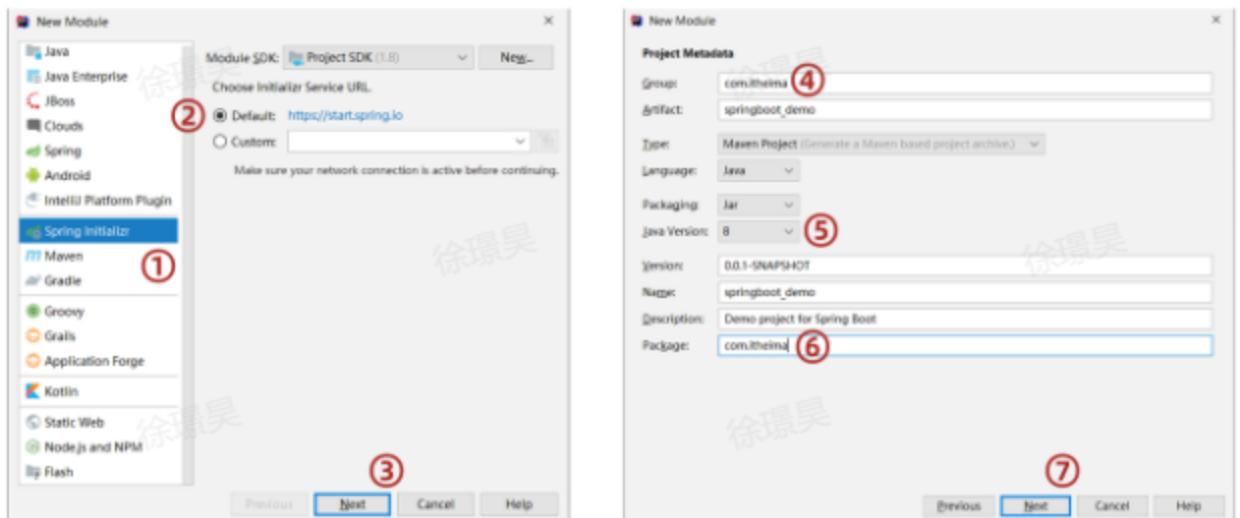
SpringBoot 整合 junit 特别简单, 分为以下三步完成

- 在测试类上添加 `SpringBootTest` 注解
- 使用 `@Autowired` 注入要测试的资源
- 定义测试方法进行测试

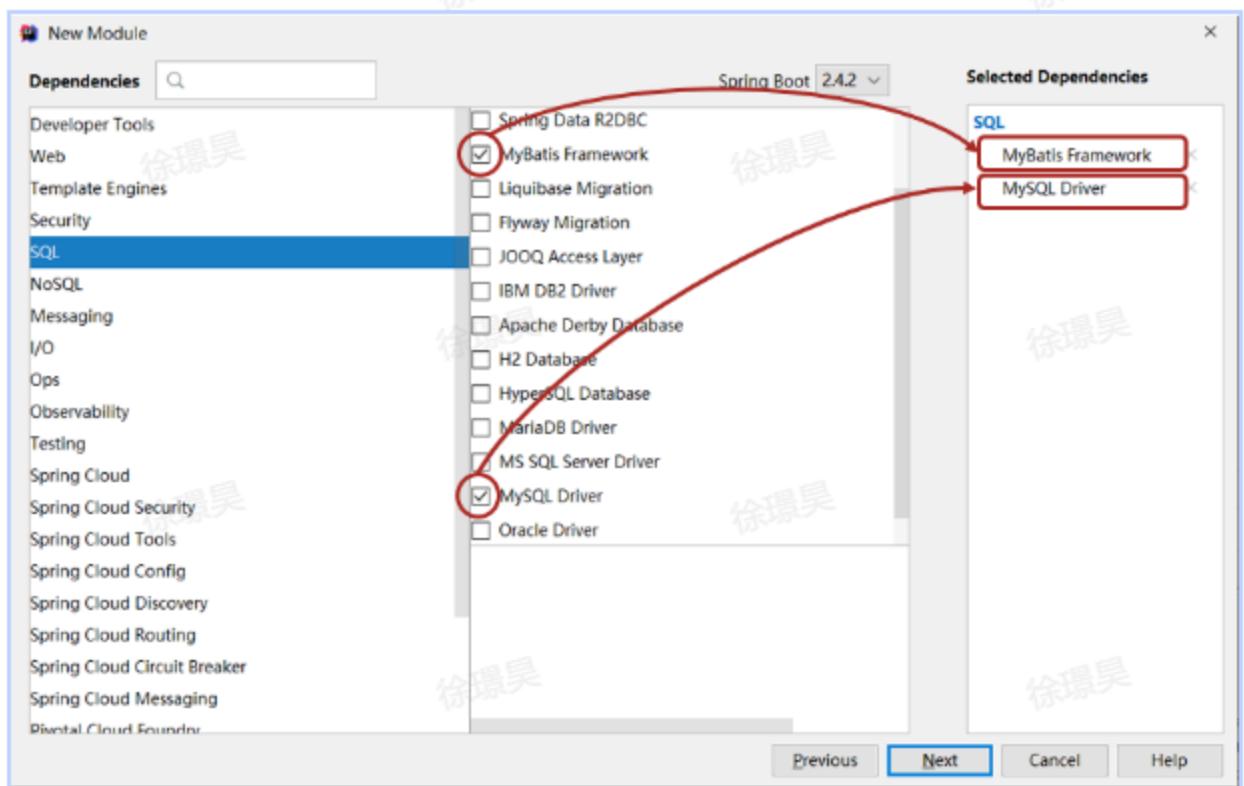
## 5. SpringBoot整合MyBatis:

### 5.1 创建模块:

- 创建新模块, 选择 Spring Initializr, 并配置模块相关基础信息



- 选择当前模块需要使用的技术集 (MyBatis、MySQL)



## 5.2 定义实体类(例):

```
public class Book {  
    private Integer id;  
    private String name;  
    private String type;  
    private String description;  
  
    //setter and getter  
    //toString  
}
```

## 5.3 定义dao接口(例):

```
//需要修正,查看5.6测试  
public interface BookDao {  
    @Select("select * from tbl_book where id = #{id}")  
    public Book getById(Integer id);  
}
```

## 5.4 定义测试类(例):

```
@SpringBootTest
class Springboot08MybatisApplicationTests {
    @Autowired
    private BookDao bookDao;

    @Test
    void test GetById() {
        Book book = bookDao.getById(1);
        System.out.println(book);
    }
}
```

## 5.5 数据库编写配置:

在 application.yml 配置文件中配置如下内容:

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db
    username: root
    password: root
```

## 5.6 运行测试及修正:

运行测试方法，我们会看到如下错误信息

```
① Tests failed: 1 of 1 test - 2 ms
  org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'bookController': Injection of autowired dependencies failed; nested exception is java.lang.IllegalArgumentException: No qualifying bean of type 'com.itheima.dao.BookDao' available: expected at least 1 bean which qualifies as autowire candidate. Dependency annotations: {@org.springframework.beans.factory.annotation.Autowired(required=true)}
    at org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor.postProcessBeforeInitialization(AutowiredAnnotationBeanPostProcessor.java:339)
    at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.postProcessBeforeInitialization(AbstractAutowireCapableBeanFactory.java:395)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.preInstantiateSingletons(DefaultListableBeanFactory.java:552)
    at org.springframework.context.support.AbstractApplicationContext.finishBeanFactoryInitialization(AbstractApplicationContext.java:829)
    at org.springframework.context.support.AbstractApplicationContext.refresh(AbstractApplicationContext.java:541)
    at org.springframework.boot.SpringApplication.refresh(SpringApplication.java:744)
    at org.springframework.boot.SpringApplication.refreshContext(SpringApplication.java:397)
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:315)
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:805)
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:814)
Caused by: java.lang.IllegalArgumentException: No qualifying bean of type 'com.itheima.dao.BookDao' available: expected at least 1 bean which qualifies as autowire candidate. Dependency annotations: {@org.springframework.beans.factory.annotation.Autowired(required=true)}
    at org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor$AutowiredFieldElement.resolveFieldValue(AutowiredAnnotationBeanPostProcessor.java:615)
    at org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor$AutowiredFieldElement.inject(AutowiredAnnotationBeanPostProcessor.java:573)
    at org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor.postProcessBeforeInitialization(AutowiredAnnotationBeanPostProcessor.java:336)
    ... 10 more
Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException: No matching bean found for provided arguments: [com.itheima.dao.BookDao]
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.raiseNoMatchingBeanFound(DefaultListableBeanFactory.java:1790)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.doResolveDependency(DefaultListableBeanFactory.java:1346)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveDependency(DefaultListableBeanFactory.java:1300)
    at org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor$AutowiredFieldElement.resolveFieldValue(AutowiredAnnotationBeanPostProcessor.java:615)
    ... 10 more
```

错误信息显示在 Spring 容器中没有 BookDao 类型的 bean,为什么会出现这种情况呢?

原因: Mybatis 会扫描接口并创建接口的代码对象交给 Spring 管理,但是现在并没有告诉 Mybatis 哪个是 dao 接口。而我们要解决这个问题需要在BookDao 接口上使用 @Mapper注解添加其信息:

改进后的dao接口:  
(个人看法：干脆叫BookMapping得了)

```
@Mapper
public interface BookDao {
    @Select("select * from tbl_book where id = #{id}")
    public Book getById(Integer id);
}
```

## 5.7 validation：数据验证

- a. spring-boot-starter-validation:是Spring Boot的验证启动器,主要用于数据验证
- b. 作用:
  - 提供了方便的注解来验证输入数据的有效性
  - 可以在实体类、DTO(可以当作用于数据传输的实体)等上使用注解来定义验证规则
  - 即可以验证用户输入,确保API接收的数据符合预期
  - 减少了手动编写验证代码,提高其可读性与维护性
- c. 常用注解:
  - @NotNull: 确保值不为null
  - @NotEmpty: 确保字符串、集合不为空
  - @Size: 限制字符串长度或集合大小
  - @Min/@Max: 限制数值范围
  - @Email: 验证邮箱格式
  - @Pattern: 使用正则表达式验证

## 5.8 实现JWT鉴权:

- JWT:(Json Web Token):**是一种开放标准 (RFC 7519)，用于在各方之间安全地传输信息作为JSON对象,包含:**
  - Header: 包含令牌类型和使用的哈希算法
  - Payload: 包含声明 (用户信息和其他数据)
  - Signature: 用于验证消息在传输过程中没有被更改
- 作用:
  - 身份验证: 用户登录后，服务器生成JWT返回给客户端，后续请求携带此令牌
  - 信息交换: 可以在Payload中安全地传输信息
  - 无状态: 服务端不需要存储会话信息，适合分布式系统
  - 跨域认证: 适合前后端分离项目和微服务架构
- 在Spring Boot中实现:
  - 添加依赖:

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.11.5</version>
</dependency>
```

- 在yml/properties配置JWT的密钥及有效时间(**或者在工具类设置常量**):

```
jwt.secret=mySecretKey12345678901234567890123456789012
jwt.expiration=3600 # 1小时
```

- 设置JWT工具类(封装JWT):

```
import io.jsonwebtoken.*;
import io.jsonwebtoken.security.Keys;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import javax.crypto.SecretKey;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

@Component
public class JwtTokenUtil {

    @Value("${jwt.secret}")
    private String secret;

    @Value("${jwt.expiration}")
    private Long expiration;

    // 生成密钥,包含用户信息
    private SecretKey generateKey() {
        return Keys.hmacShaKeyFor(secret.getBytes());
    }

    // 生成token
    public String generateToken(String username) {
        Map<String, Object> claims = new HashMap<>();
        return Jwts.builder()
            .setClaims(claims)
            .setSubject(username)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + expiration))
            .signWith(generateKey(), SignatureAlgorithm.HS256)
            .compact();
    }

    // 从token中获取用户名,提取用户信息
    public String getUsernameFromToken(String token) {
        return Jwts.parserBuilder()
            .setSigningKey(generateKey())
            .build()
            .parseClaimsJws(token)
            .getBody()
    }
}
```

```
        .getSubject();
    }

    // 验证token的有效性
    public boolean validateToken(String token) {
        try {
            Jwts.parserBuilder()
                .setSigningKey(generateKey())
                .build()
                .parseClaimsJws(token);
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}
```

- 实现登录接口(Controller):处理用户认证并颁发令牌
- 实现JWT过滤器:拦截请求并处理JWT认证

```
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Autowired
    private JwtTokenUtil jwtTokenUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
        throws ServletException, IOException {

        // 1. 从请求头获取token
        String token = request.getHeader("Authorization");

        // 2. 如果没有token，直接放行(可能有其他认证方式)
        if (StringUtils.isBlank(token) || !token.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        // 3. 提取真正的token
        token = token.substring(7);

        try {
            // 4. 验证token
            if (jwtTokenUtil.validateToken(token)) {
                String username = jwtTokenUtil.getUsernameFromToken(token);

                // 5. 设置认证信息
                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(username, null, new AuthenticationDetailsSource());
                authentication.setDetails(new WebAuthenticationDetailsSource());
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        } catch (Exception e) {
            logger.error("JWT令牌验证失败", e);
        }

        filterChain.doFilter(request, response);
    }
}
```

- 配置Spring Security:集成JWT到安全框架
- 安全框架:**身份验证,授权控制,数据保护,会话管理,审计日志(各层都有用)**

```

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtAuthenticationFilter jwtAuthenticationFilter;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.SINGLE_USE)
            .and()
            .authorizeRequests()
            .antMatchers("/auth/**").permitAll() // 登录接口放行
            .anyRequest().authenticated(); // 其他接口需要认证

        // 添加JWT过滤器
        http.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);
    }

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}

```

## 7.5 MyBatisPlus:

### 1. 配置:(gradle/maven依赖坐标)

g:com.baomidou  
a:mybatis-plus-boot-starter

### 2. 简单案例:

- 根据数据库创建实体类Dao接口

```

@Mapper
public interface UserDao extends BaseMapper<User>{
}

```

功能	自定义接口	MP接口
新增	boolean save(T t)	int insert(T t)
删除	boolean delete(int id)	int deleteById(Serializable id)
修改	boolean update(T t)	int updateById(T t)
根据id查询	T getById(int id)	T selectById(Serializable id)
查询全部	List<T> getAll()	List<T> selectList()
分页查询	PageInfo<T> getAll(int page, int size)	IPage<T> selectPage(IPage<T> page)
按条件查询	List<T> getAll(Condition condition)	IPage<T> selectPage(Wrapper<T> queryWrapper)

- delete相关:

- Q:为什么删除方法参数是个序列化类:

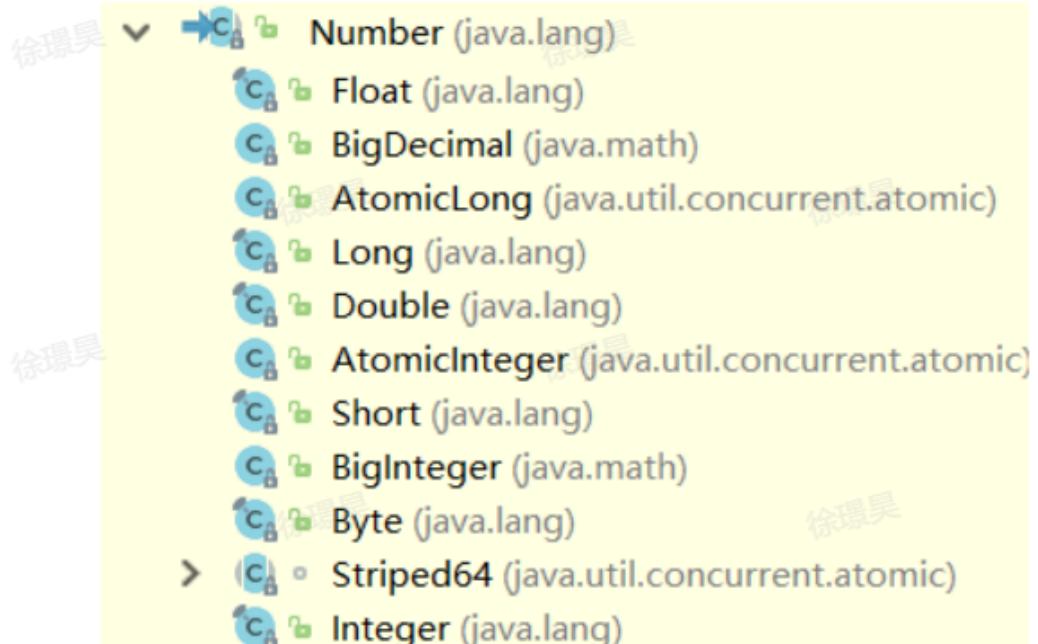
- int deleteById (Serializable id)

- **public final class String**

**implements java.io.Serializable**

**public abstract class Number**

**implements java.io.Serializable**



- A:由图可得:

- `String`和`Number`是`Serializable`的子类,

- Number又是Float,Double,Integer等类的父类,
- 能作为主键的数据类型都已经是Serializable的子类,
- MP使用Serializable作为参数类型,就好比我们可以用Object接收任何数据类型一样。
- int:返回值类型,数据删除成功返回1,未删除数据返回0
- select相关:
  - 查询所有>List selectList(Wrapper queryWrapper)
    - Wrapper: 用来构建条件查询的条件,目前我们没有可直接传为Null
    - List:因为查询的是所有,所以返回的数据是一个集合
  - 在测试类的操作:

```

@SpringBootTest
class MybatisplusQuickstartApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll() {
        List<User> userList = userDao.selectList(null);
        System.out.println(userList);
    }
}

```

### 3. Lombok:

- Lombok是一个Java库,用于减少样板代码。
- 通过注解自动生成常用的Java代码,如getter、setter、构造函数等。
  - @Slf4j:简化日志(Simple Logging Facade for Java):

Java

```
@Slf4j
public class OrderService {
    public void createOrder() {
        log.info("订单创建成功"); // 直接使用log 对象
        log.error("订单支付失败");
    }
}
```

等效于以下手动代码：

Java

```
public class OrderService {
    // Lombok 自动生成这行代码
    private static final Logger log = LoggerFactory.getLogger(OrderService.class);

    public void createOrder() {
        log.info("订单创建成功");
        log.error("订单支付失败");
    }
}
```

- @Setter:为模型类的属性提供setter方法
- @Getter:为模型类的属性提供getter方法
- @ToString:为模型类的属性提供toString方法
- @EqualsAndHashCode:为模型类的属性提供equals和hashcode方法
- @Data:是个组合注解，包含上面的注解的功能
- @NoArgsConstructor:提供一个无参构造函数
- @AllArgsConstructor:提供一个包含所有参数的构造函数

```
◦ @NotBlank(message = "名称不能为空")
@Length(min = 2, max = 50, message = "名称长度必须在2至50之间")
private String name;

@Min(value = 0, message = "年龄必须大于或等于0")
@Max(value = 150, message = "年龄必须小于或等于150")
private int age;

@NotNull(message = "邮箱不能为空")
@email(message = "邮箱格式不正确")
private String email;

@Pattern(regexp = "^\d{3}-\d{8}$", message = "手机号码格式错误")
private String phoneNumber;

@Past(message = "生日必须是过去的日期")
private LocalDate birthDate;

@Future(message = "计划日期必须是将来的日期")
private LocalDate planDate;

@Positive(message = "分数必须为正数")
private double score;

@Size(min = 1, max = 5, message = "兴趣爱好列表必须包含1至5项")
private List<String> hobbies;

@AssertTrue(message = "必须同意服务条款")
private boolean agreeTerms;
```

## 4. 分页功能:

### 4.1 简介:

- IPage selectPage(IPage page, Wrapper queryWrapper)
  - IPage:用来构建分页查询条件

- Wrapper：用来构建条件查询的条件，目前我们没有可直接传为Null
- IPage:返回值，你会发现构建分页条件和方法的返回值都是IPage  
IPage是一个接口，我们需要找到它的实现类来构建它，具体的实现类，可以进入到IPage类中按ctrl+h,会找到其有一个实现类为Page

## 4.2 实现步骤：

- 调用方法传入参数获取返回值:

```

@SpringBootTest
class MybatisplusQuickstartApplicationTests {

    @Autowired
    private UserDao userDao;

    //分页查询
    @Test
    void testSelectPage(){
        //创建IPage分页对象,设置分页参数,1为当前页码, 3为每页显示的记录数
        //User实体类要用@TableName注解映射数据库表名
        // @TableName(value = "user", autoResultMap = true)
        //autoResultMap = true 时, MyBatis-Plus 会为该实体类自动生成一个默认的 ResultMa
        IPage<User> page=new Page<>(1,3);

        //执行分页查询
        userDao.selectPage(page,null);
        //获取分页结果
        System.out.println("当前页码值: "+page.getCurrent());
        System.out.println("每页显示数: "+page.getSize());
        System.out.println("一共多少页: "+page.getPages());
        System.out.println("一共多少条数据: "+page.getTotal());
        System.out.println("数据: "+page.getRecords());
    }
}

```

- 设置分页拦截器: 这个拦截器MP已经为我们提供好了，我们只需要将其配置成Spring管理的bean对象即可

```
@Configuration
public class MybatisPlusConfig {

    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){
        //1 创建MybatisPlusInterceptor拦截器对象
        MybatisPlusInterceptor mpInterceptor=new MybatisPlusInterceptor();
        //2 添加分页拦截器
        mpInterceptor.addInnerInterceptor(new PaginationInnerInterceptor());
        return mpInterceptor;
    }
}
```

## 5. 条件查询(使用Wrapper类):

### 5.1 QueryWrapper:

```
QueryWrapper qw = new QueryWrapper();
qw.lt("age",18);

List<User> userList = userDao.selectList(qw);
```

- lt: 小于(less than)

### 5.2 QueryWrapper的基础上使用lambda:

```
QueryWrapper<User> qw = new QueryWrapper<User>();

qw.lambda().lt(User::getAge, 10);//添加条件
List<User> userList = userDao.selectList(qw);
```

### 5.3 LambdaQuerywrapper:

```
LambdaQueryWrapper<User> lqw = new LambdaQueryWrapper<User>();

lqw.lt(User::getAge, 10);
List<User> userList = userDao.selectList(lqw);
```

## 5.4 多条件构建:

```
LambdaQueryWrapper<User> lqw = new LambdaQueryWrapper<User>();  
  
lqw.lt(User::getAge, 30);  
lqw.gt(User::getAge, 10);  
List<User> userList = userDao.selectList(lqw);
```

- gt:大于(Greater than)
- 多条件支持链式编程
- or()相当于or,不加默认为and

## 5.5 null判定:

个人认为:还是@Sql直接限制传入数据条件好使

- lt():
  - lt()方法

```
(m) * lt(SFunction<User, ?> column, Object val)  
(m) lt(boolean condition, SFunction<User, ?> column, Object val)
```

- condition为boolean类型, 返回true, 则添加条件, 返回false则不添加条件

```

@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll(){
        //模拟页面传递过来的查询数据
        UserQuery uq = new UserQuery();
        uq.setAge(10);
        uq.setAge2(30);
        LambdaQueryWrapper<User> lqw = new LambdaQueryWrapper<User>();
        lqw.lt(null!=uq.getAge2(),User::getAge, uq.getAge2());
        lqw.gt(null!=uq.getAge(),User::getAge, uq.getAge());
        List<User> userList = userDao.selectList(lqw);
        System.out.println(userList);
    }
}

```

## 6. 查询投影(与SQL语言高度相似):

### 6.1 select:

```

LambdaQueryWrapper<User> lqw = new LambdaQueryWrapper<User>();

lqw.select(User::getId,User::getName,User::getAge);
List<User> userList = userDao.selectList(lqw);

QueryWrapper<User> lqw = new QueryWrapper<User>();

lqw.select("id","name","age","tel");
List<User> userList = userDao.selectList(lqw);

```

### 6.2 聚合查询:

聚合函数查询，完成count、max、min、avg、sum的使用

```

@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll(){
        QueryWrapper<User> lqw = new QueryWrapper<User>();

        //lqw.select("count(*) as count");
        //SELECT count(*) as count FROM user

        //lqw.select("max(age) as maxAge");
        //SELECT max(age) as maxAge FROM user

        //lqw.select("min(age) as minAge");
        //SELECT min(age) as minAge FROM user

        //lqw.select("sum(age) as sumAge");
        //SELECT sum(age) as sumAge FROM user

        lqw.select("avg(age) as avgAge");
        //SELECT avg(age) as avgAge FROM user

        List<Map<String, Object>> userList = userDao.selectMaps(lqw);
        System.out.println(userList);
    }
}

```

## 6.3 分组查询:

```

QueryWrapper<User> lqw = new QueryWrapper<User>();

lqw.select("count(*) as count,tel");
lqw.groupBy("tel");
List<Map<String, Object>> list = userDao.selectMaps(lqw);

```

## 7. 查询条件(与SQL语言高度相似):

### 7.1 等值查询:

- eq(): 相当于equal, 对应的sql语句为
  - SELECT id,name,password,age,tel FROM user WHERE (name = ? AND password = ?)
  - selectList: 查询结果为多个或者单个
  - selectOne: 查询结果为单个
- 范围查询

```
@Test
void testGetAll(){
LambdaQueryWrapper<User> lqw = new LambdaQueryWrapper<User>();

lqw.eq(User::getName, "Jerry").eq(User::getPassword, "jerry");
User loginUser = userDao.selectOne(lqw);

System.out.println(loginUser);
}
```

### 7.2 范围查询:

```
lqw.between(User::getAge, 10, 30);
```

- ge: 大于等于
- lt: 小于
- lte: 小于等于

### 7.3 模糊查询:

```
lqw.likeLeft(User::getName, "J");
```

- like(): 前后加百分号, 如 %J%
- likeLeft(): 前面加百分号, 如 %J
- likeRight(): 后面加百分号, 如 J%

### 7.4 排序查询:

```
lwq.orderBy(true, false, User::getId);
```

- orderBy排序
  - condition: 条件, true则添加排序, false则不添加排序
  - isAsc: 是否为升序, true升序, false降序

- columns:排序字段，可以有多个
- orderByAsc/Desc(单个column):按照指定字段进行升序/降序
- orderByAsc/Desc(多个column):按照多个字段进行升序/降序
- orderByAsc/Desc
  - condition:条件，true添加排序，false不添加排序
  - 多个columns：按照多个字段进行排序

## 特殊章节 @TableName/@TableField:

- @TableField
  - 类型:属性注解
  - 位置:属性定义上方
  - 作用:设置当前属性对应的数据库表中的字段关系
  - 相关字段:
    - value(默认): 设置数据库表字段名称
    - exist:设置属性在数据库表字段中是否存在，默认为true，此属性不能与value合并使用
    - select:设置属性是否参与查询，此属性与select()映射配置不冲突
- @TableName
  - 类型:类注解
  - 位置:类定义上方
  - 作用:设置当前类对应于数据库表关系
  - 相关字段:value(默认): 设置数据库表名称

## 8. DML编程控制: (Data Manipulation Language,数据操作语言)

### 8.1 @TableId:

- 类型:属性注解
  - 位置:表示**主键的属性定义上方**
  - 作用:设置当前类中主键属性的生成策略
  - 相关属性:
    - value(默认): 设置数据库表主键名称
    - type:**设置主键属性的生成策略**，值查照IdType的枚举值
  - type相关生成策略(主键生成策略)
  - NONE: 不设置id生成策略,约等于INPUT
  - AUTO:数据库ID自增,这种策略**适合在数据库服务器只有1台的情况下使用,不可作为分布式ID使用**

- INPUT: 用户手工输入id
- ASSIGN\_ID: 雪花算法生成id(可兼容数值型与字符串型)
- ASSIGN\_UUID: 以UUID生成算法作为id生成策略
- 其他的几个策略均已过时，都将被ASSIGN\_ID和ASSIGN\_UUID代替掉

## 特殊章节:分布式ID及实现

- 分布式ID: 是在分布式系统中生成的全局唯一标识符，用于在多个节点、服务或数据库中唯一标识数据，避免因单点生成导致的ID冲突。其核心在于解决传统单机数据库自增ID的局限性，适应高并发、高可用的分布式环境
- 常见方案:
  - 雪花算法 (Snowflake, Twitter开源): 时间戳 + 机器ID + 序列号
    - 趋势递增生成快, 依赖系统时钟, 警惕时钟回拨问题(美团Leaf、百度UidGenerator)
  - 数据库号段模式: 预分配ID段, 减少数据库访问压力
    - 服务从数据库获取一个ID段 (如1~1000), 内存中分配ID, 用完后重新申请(美团Leaf-Segment方案, 通过双Buffer预加载避免性能抖动)
  - UUID: 基于随机数或MAC地址的128位字符串
  - Redis自增: 利用Redis的原子操作生成递增ID
    - INCR 或 INCRBY 生成递增ID
    - 如生成订单号ORDER:20231101:0001
    - 依赖Redis可用性, 需要持久化配置
  - MongoDB ObjectId: 时间戳 + 机器ID + 进程ID + 计数器
    - 如5f4d87e6d6b4a31e8c7b3f8a, 内置生成, 无需额外依赖

方案	适用场景	优点	缺点
雪花算法	高并发、有序ID需求 (如订单)	高性能、趋势递增	依赖时钟, 需解决回拨问题
数据库号段	中小规模系统, 分库分表	减少DB压力, 可控性强	需维护号段表, 存在短暂不连续
UUID	简单场景, 无递增要求	无需中心化节点, 生成简单	无序、存储空间大、可读性差
Redis自增	临时性ID生成 (如活动编号)	简单高效	依赖Redis可用性

## 8.2 多记录操作:

```
//删除指定多条数据
List<Long> list = new ArrayList<>();
list.add(1402551342481838081L);
list.add(1402553134049501186L);
list.add(1402553619611430913L);
userDao.deleteBatchIds(list);

//查询指定多条数据
List<Long> list = new ArrayList<>();
list.add(1L);
list.add(3L);
list.add(4L);
userDao.selectBatchIds(list);
```

## 8.3 逻辑删除:

- Q:如果每个表都要有逻辑删除，那么就需要在每个模型类的属性上添加@TableLogic注解
- A:在配置文件中添加全局配置:本质是修改操作,在查询数据时会自动带上该字段

```
mybatis-plus:
  global-config:
    db-config:
      # 逻辑删除字段名
      logic-delete-field: deleted
      # 逻辑删除字面值: 未删除为0
      logic-not-delete-value: 0
      # 逻辑删除字面值: 删除为1
      logic-delete-value: 1
```

## 8.4 @TableLogic:

- 类型:属性注解
  - 位置:设置类中**表示删除字段的属性定义上方**
  - 作用:标识该字段为进行逻辑删除的字段
  - 相关属性:
    - value: 逻辑未删除值
    - delval:逻辑删除值

## 9. 乐观锁:

### 9.1 为什么要乐观锁:

业务并发现象带来的问题: 秒杀

- 假如有100个商品或者票在出售，为了能保证每个商品或者票只能被一个人购买，如何保证不会出现超买或者重复卖
- 对于这一类问题，其实有很多的解决方案可以使用
- 第一个最先想到的就是锁，锁在一台服务器中是可以解决的，但是如果在多台服务器下锁就没有办法控制，比如12306有两台服务器在进行卖票，在两台服务器上都添加锁的话，那也有可能会导致在同一时刻有两个线程在进行卖票，还是会出现并发问题
- 我们接下来介绍的这种方式是针对于小型企业的解决方案，因为数据库本身的性能就是个瓶颈，如果对其并发量超过2000以上的就需要考虑其他的解决方案了。
- 即乐观锁主要解决的问题是当要更新一条记录的时候，希望这条记录没有被别人更新。

### 9.2 实现思路:

(个人理解:每次修改添加一个状态类,根据状态类判断是不是已修改,若已修改则后来的线程修改会失败)

- 数据库表中添加version列，比如默认值给1
- 第一个线程要修改数据之前，取出记录时，获取当前数据库中的version=1
- 第二个线程要修改数据之前，取出记录时，获取当前数据库中的version=1
- 第一个线程执行更新时，  
  set version = newVersion where version = oldVersion
  - newVersion = version+1 [2]
  - oldVersion = version [1]
- 第二个线程执行更新时，  
  set version = newVersion where version = oldVersion
  - newVersion = version+1 [2]
  - oldVersion = version [1]
- 假如这两个线程都来更新数据，第一个和第二个线程都可能先执行
  - 假如第一个线程先执行更新，会把version改为2，
  - 第二个线程再更新的时候，  
    set version = 2 where version = 1, 此时数据库表的数据  
    version已经为2，所以第二个线程会修改失败
  - 假如第二个线程先执行更新，会把version改为2，
  - 第一个线程再更新的时候，  
    set version = 2 where version = 1, 此时数据库表的数据  
    version已经为2，所以第一个线程会修改失败
  - 不管谁先执行都会确保只能有一个线程更新数据，这就是MP提供的乐观锁的实现原理分析。

## 9.3 实现步骤:

- 步骤1:数据库表添加列(列名可以任意,比如使用version,给列设置默认值为1):

The screenshot shows the MySQL Workbench interface for managing the 'tbl\_user' table. The table has columns: id (bigint), name (varchar), pwd (varchar), age (int), tel (varchar), deleted (int), and version (int). The 'version' column is highlighted with a red border. Below the table, there is a configuration panel for the 'version' column, specifically for the 'Default' value, which is set to '1'. There are also checkboxes for 'Auto Increment' and 'No Nulls'.

名	类型	长度	小数点	不是 null	虚拟	键	注释
id	bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		1
name	varchar	32	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
pwd	varchar	32	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
age	int	3	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
tel	varchar	32	0	<input type="checkbox"/>	<input type="checkbox"/>		
deleted	int	1	0	<input type="checkbox"/>	<input type="checkbox"/>		
version	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		

- 步骤2:在模型类中添加对应的属性,并根据添加的字段列名,在模型类中添加对应的属性值

```
@Data  
//@TableName("tbl_user") 可以不写是因为配置了全局配置  
public class User {  
    @TableId(type = IdType.ASSIGN_UUID)  
    private String id;  
    private String name;  
    @TableField(value="pwd",select=false)  
    private String password;  
    private Integer age;  
    private String tel;  
    @TableField(exist=false)  
    private Integer online;  
    private Integer deleted;  
  
    //添加对应的属性值  
    @Version  
    private Integer version;  
}
```

- 步骤3:添加乐观锁的拦截器:

```

@Configuration
public class MpConfig {
    @Bean
    public MybatisPlusInterceptor mpInterceptor() {
        //1.定义Mp拦截器
        MybatisPlusInterceptor mpInterceptor = new MybatisPlusInterceptor();

        //2.添加/启用 乐观锁拦截器(具体逻辑MyBatis-Plus已经封装好了)
        mpInterceptor.addInnerInterceptor(new OptimisticLockerInnerInterceptor());

        return mpInterceptor;
    }
}

```

- 步骤4:执行更新操作(此步骤为具体检验乐观锁是否成功添加)

```

@SpringBootTest
class Mybatisplus03DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testUpdate(){
        User user = new User();
        user.setId(3L);

        user.setName("Jock666");
        userDao.updateById(user);

        //需在修改时一同更新version字段
        user.setVersion(1);
    }
}

```

## 10. 快速开发(代码生成器,真正的简化开发原因):

### 10.1 原理:

简而言之,就是保留框架的基础上,通过设置其变量来直接生成一些重复化公式化的代码(**提供数据库表和字段信息**)

## **10.2 实现步骤:**

- a. 创建一个Maven项目
- b. 在pom.xml包导入mybatis-plus的jar包

```
<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <!--此处指定父项目-->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.1</version>
    </parent>
    <groupId>com.itheima</groupId>
    <artifactId>mybatisplus_04_generator</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <!--spring webmvc,添加web依赖-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!--mybatisplus,添加mybatis-plus-->
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-boot-starter</artifactId>
            <version>3.4.1</version>
        </dependency>

        <!--druid,添加druid数据库连接池(阿里巴巴)-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.16</version>
        </dependency>

        <!--mysql,添加MySQL数据库的JDBC驱动-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <!--表示编译不需要,运行需要-->
            <scope>runtime</scope>
        </dependency>
    </dependencies>

```

```
</dependency>

<!--test,添加spring boot的测试支持-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <!--表示只在测试时使用-->
    <scope>test</scope>
</dependency>

<!--lombok,通过注解简化代码-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
</dependency>

<!--代码生成器,添加本章主角-->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-generator</artifactId>
    <version>3.4.1</version>
</dependency>

<!--velocity模板引擎,用于代码生成器生成代码时的模板处理-->
<dependency>
    <groupId>org.apache.velocity</groupId>
    <artifactId>velocity-engine-core</artifactId>
    <version>2.3</version>
</dependency>

</dependencies>

<build>
    <plugins>
        <!--SpringBoot的Maven构建插件,提供打包可执行jar的功能-->
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>
```

c. 编写引导类:

```
@SpringBootApplication
public class Mybatisplus04GeneratorApplication {
    public static void main(String[] args) {
        SpringApplication.run(Mybatisplus04GeneratorApplication.class, args);
    }
}
```

d. 创建代码生成类:

```
public class CodeGenerator {
    public static void main(String[] args) {
        //1.获取代码生成器的对象
        AutoGenerator autoGenerator = new AutoGenerator();

        //设置数据库相关配置
        DataSourceConfig dataSource = new DataSourceConfig();
        dataSource.setDriverName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/mybatisplus_db?serverTimezone=UTC");
        dataSource.setUsername("root");
        dataSource.setPassword("root");
        autoGenerator.setDataSource(dataSource);

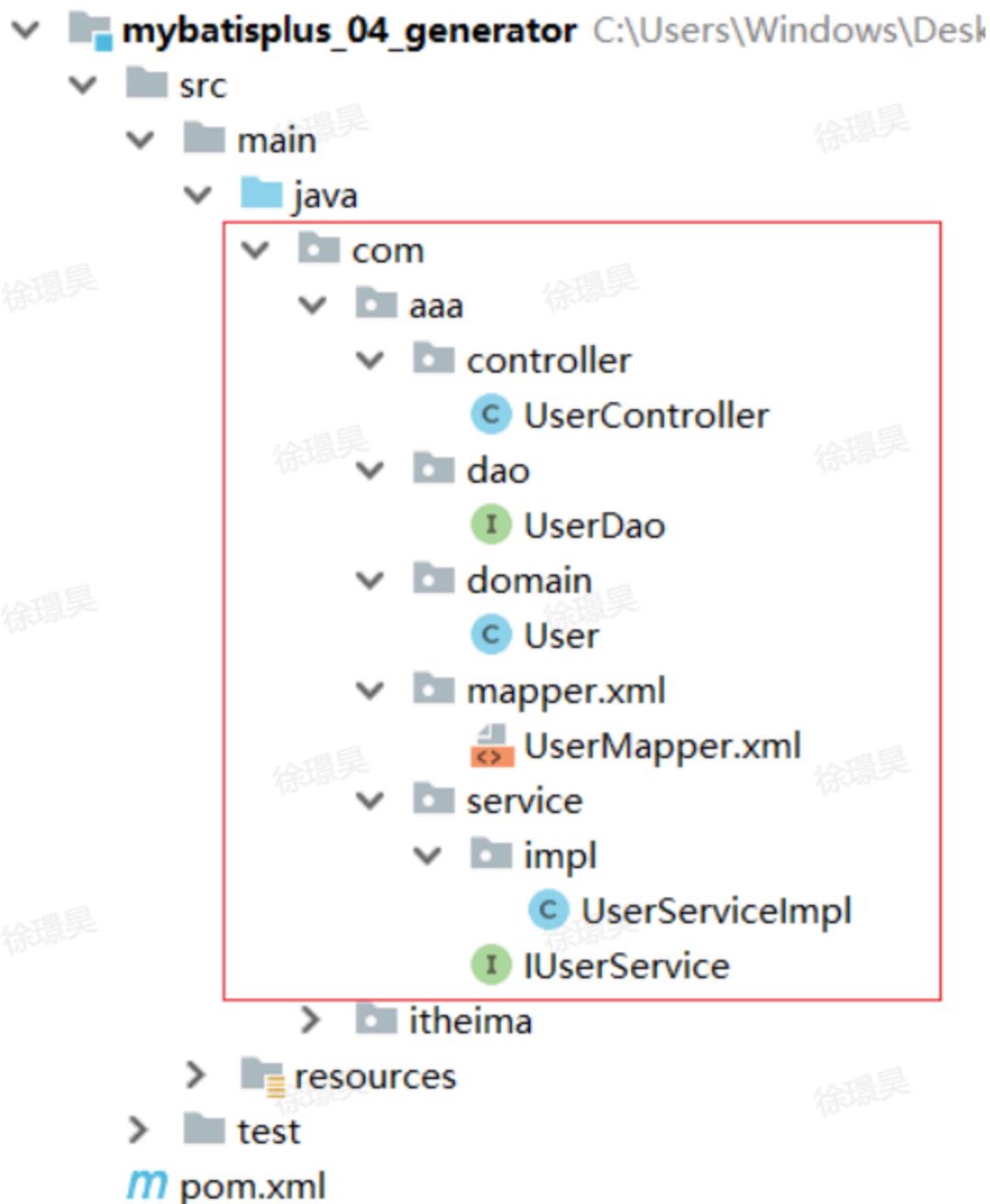
        //设置全局配置
        GlobalConfig globalConfig = new GlobalConfig();
        globalConfig.setOutputDir(System.getProperty("user.dir")+"/mybatisplus_04_generator");
        globalConfig.setOpen(false);      //设置生成完毕后是否打开生成代码所在的目录
        globalConfig.setAuthor("黑马程序员");      //设置作者
        globalConfig.setFileOverride(true);      //设置是否覆盖原始生成的文件
        globalConfig.setMapperName("%sDao");      //设置数据层接口名, %s为占位符, 指代模块名
        globalConfig.setIdType(IdType.ASSIGN_ID);  //设置Id生成策略
        autoGenerator.setGlobalConfig(globalConfig);

        //设置包名相关配置
        PackageConfig packageInfo = new PackageConfig();
        packageInfo.setParent("com.aaa");      //设置生成的包名, 与代码所在位置不冲突, 二者不能重名
        packageInfo.setEntity("domain");      //设置实体类包名
        packageInfo.setMapper("dao");      //设置数据层包名
        autoGenerator.setPackageInfo(packageInfo);

        //策略设置
        StrategyConfig strategyConfig = new StrategyConfig();
        strategyConfig.setInclude("tbl_user");  //设置当前参与生成的表名, 参数为可变参数
        strategyConfig.setTablePrefix("tbl_");  //设置数据库表的前缀名称, 模块名 = 数据库名
        strategyConfig.setRestControllerStyle(true);  //设置是否启用Rest风格
        strategyConfig.setVersionFieldName("version");  //设置乐观锁字段名
        strategyConfig.setLogicDeleteFieldName("deleted");  //设置逻辑删除字段名
        strategyConfig.setEntityLombokModel(true);  //设置是否启用lombok
        autoGenerator.setStrategy(strategyConfig);
        //2.执行生成操作
        autoGenerator.execute();
```

```
    }  
}
```

e. 运行程序:自动生成代码(controller,service,mapper,entity):



## 10.3 service中的CRUD生成(继承接口):

```
public interface UserService extends IService<User>{
    ...
}

@Service
public class UserServiceImpl extends ServiceImpl<UserDao, User> implements UserService
{
    ...
}
```

# 7.6 SpringCloud:

## 1. 分布式基础:

### 1.1 单体到集群:

- 节点:相当于是一个个服务器
- 数据库:一般跟业务层是不同的数据库
- IP和域名:IP是访问基底, 域名是为了方便记忆(都需要购买)
- 副本:由于单个服务器性能有限, 于是用多个服务器作副本(复制品)来让整个系统容纳更大的流量
- 集群:多个服务器共同工作(如多个副本一块, 仅仅只是多台机器就行, 是一种物理形态, 而分布式更多是一种**架构方式**)就叫集群
- 负载均衡:由一个网关(如Nginx)来作为流量的出入口, 由特定的算法管控和分配流量, 使得不会让某台服务器的负载过大而其它服务器没什么负载
- 路由:上述网关完成的工作, 由某种规则将流量分配至可处理业务的服务器的过程, 就是路由
- 扩容/缩容:由流量大小来增添/释放服务器副本资源
- 单体的问题:无法应对高并发
- 集群的问题:更新时需要统一下线所有项目重新打包部署(其实还在单体架构里面), 并且java支持打包但是其它语言不一定, 不能够很好的支持多语言团队协作

## 单体架构 (ALL IN ONE)

所有功能模块都在一个项目

1、项目打包

优点：开发部署

缺点：无法应对高并发



域名    IP    节点    应用    数据库



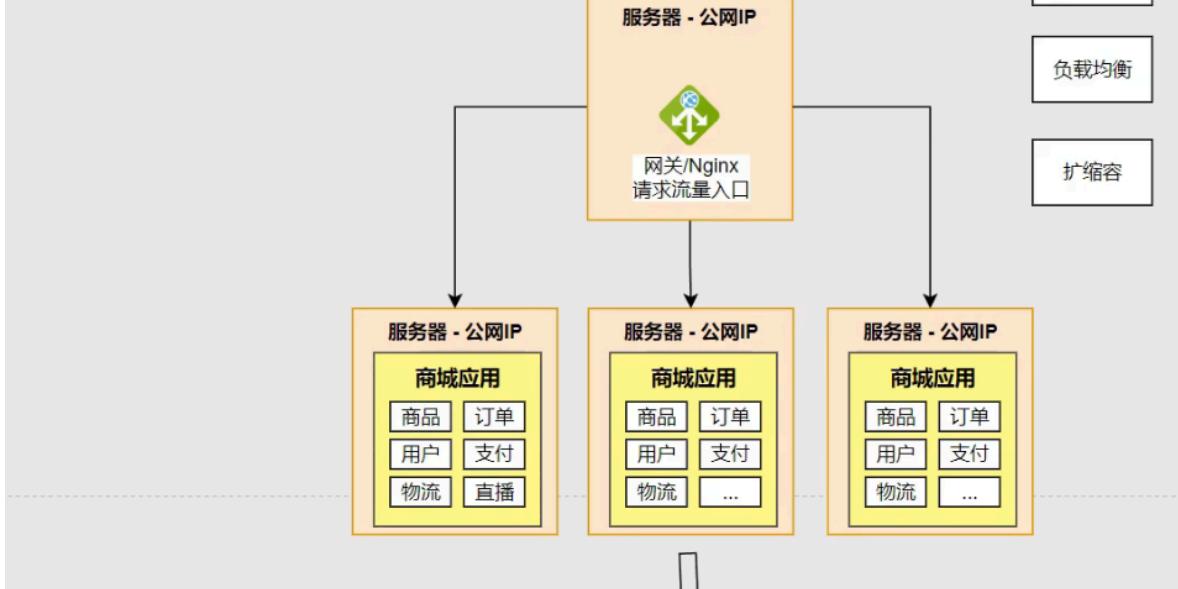
## 集群架构

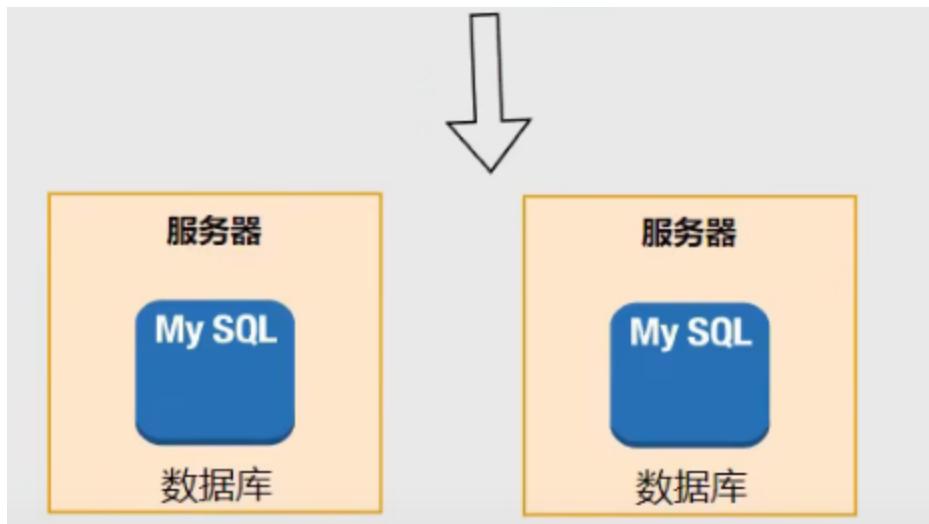
解决大并发

问题1：模块化升级

订单功能经常升级：v1.0, v2.0

问题2：RTT直连模块





## 1.2 集群到分布式：

- 为了解决上述问题,我们将单体架构变成分布式,也就是把整个大的应用根据功能拆分成各个小内容并分别部署在不同机器上(同样的,数据库也分别按功能拆分部署),也就是**微服务**  
 ==> 使得数据隔离,并且和语言无关  
 ==> 微服务即使用**Spring Boot**实现
- 当然为了避免单点故障,我们不能将某一个服务的所有副本部署在同一个节点(服务器)上,所以一个服务的不同副本一般会在多个节点(服务器)存在  
 ==> 不同微服务之间的链接,即**远程调用**,可以使用**Spring Cloud OpenFeign**  
 ==> 这会带来一个问题,若是某个节点宕机,**远程调用**服务时必须得去发现该服务副本所在的其它节点ip并访问,因此,我们需要一个组件来保存每个服务的副本所在的不同节点  
 ==> 即**注册中心**,保存各微服务副本所在的节点ip,实现服务发现和服务注册功能,可以使用**Spring Cloud Alibaba Nacos**  
 ==> 当然这里因为有多个副本,同样可以使用**网关**技术,实现同一服务在不同节点的负载均衡,可以使用**Spring Cloud Gateway**
- 同时,即便拆分了功能,需要更新时依旧需要重新打包部署来模块化升级,还是麻烦  
 ==> 所以,注册中心一般会和配置中心搭配,统一保存所有配置,需要修改时只需要在配置中心统一修改即可  
 ==> 改变配置后还会推送配置变更给指定微服务  
 ==> 实现不下线服务而实时改变
- 不过远程调用还存在其它问题,若是某个微服务去调用另一个微服务时卡顿,会影响后续微服务和后续的后续微服务的调用,也就是牵一发而动全身,若是发展到最后会演变成**服务雪崩**  
 ==> 因此,我们引入一个**服务熔断**机制: 当某个服务在规定实际内的调用失败次数/比例大于一定值时,会强制该服务后续指定时间内都无法调用其它内容而直接返回失败  
 ==> 简而言之,也就是触发该机制时,会短时间内实现**快速失败**(直接返回失败而不访问),而

不影响其它微服务的调用和使用

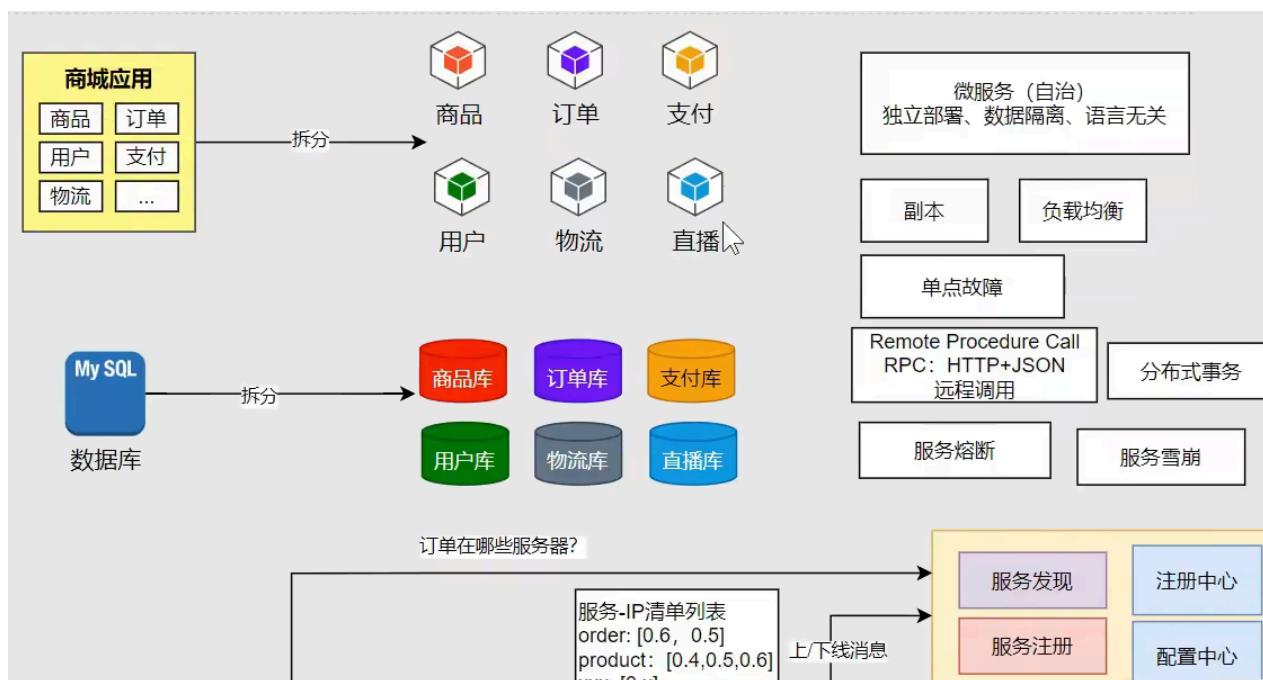
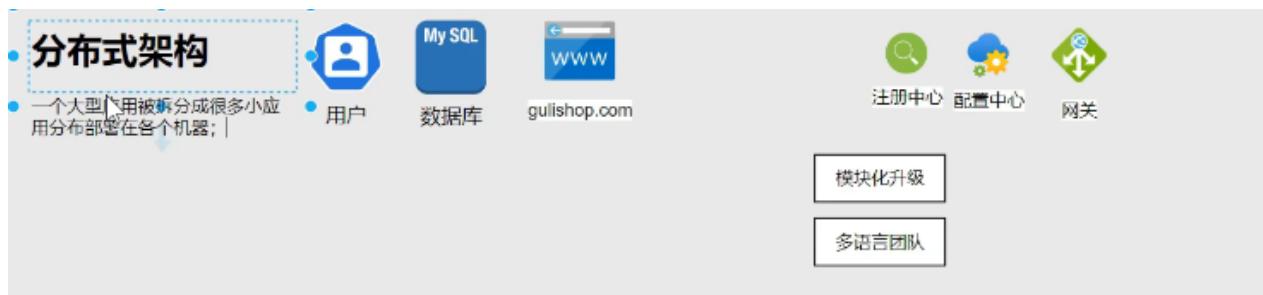
==> 可以使用**Spring Cloud Alibaba Sentinel(哨兵)**

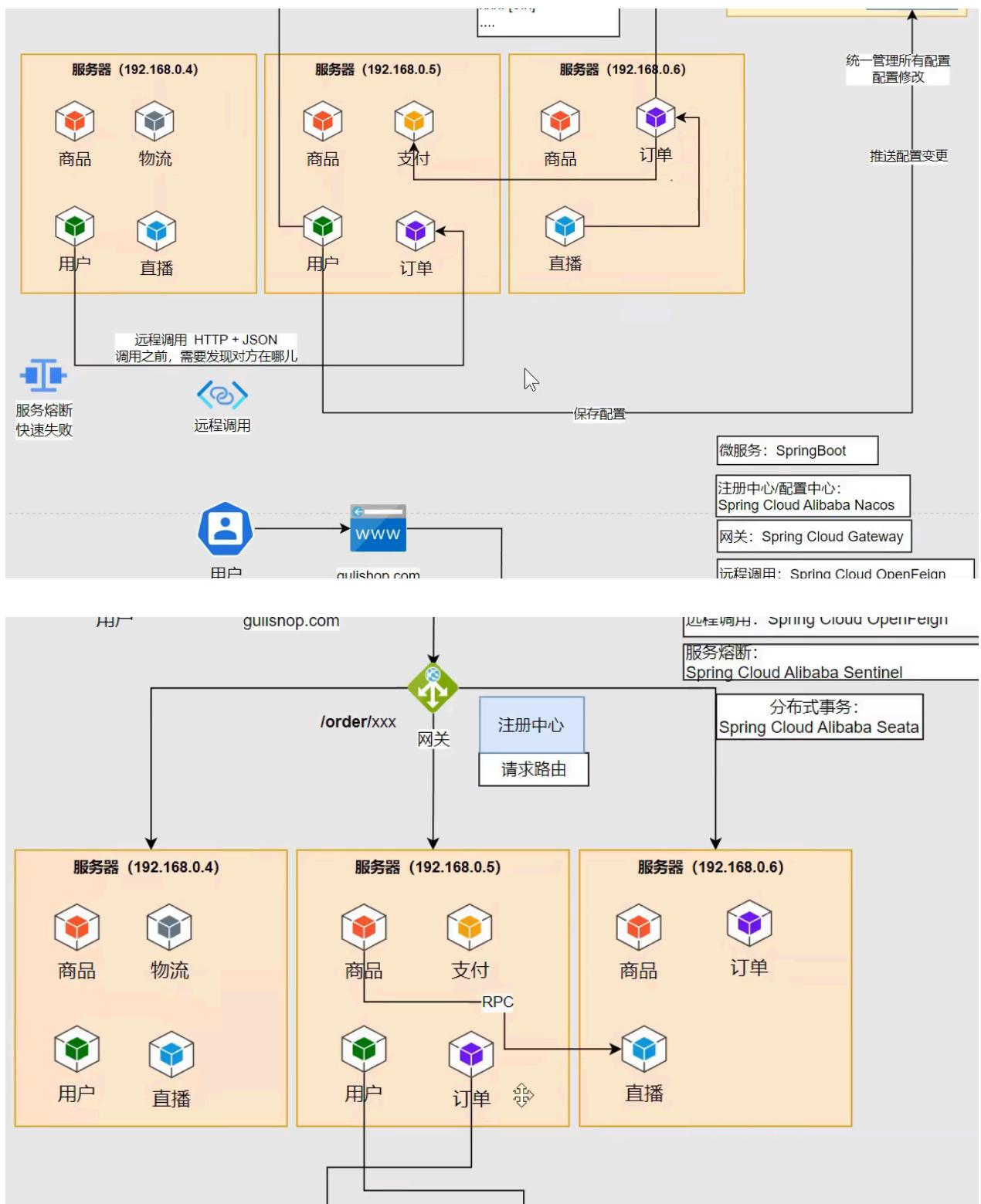
- 解决了业务层的问题,还有数据库的,数据库因为更好的分布式设计实现数据隔离,但是这会引发一个问题:

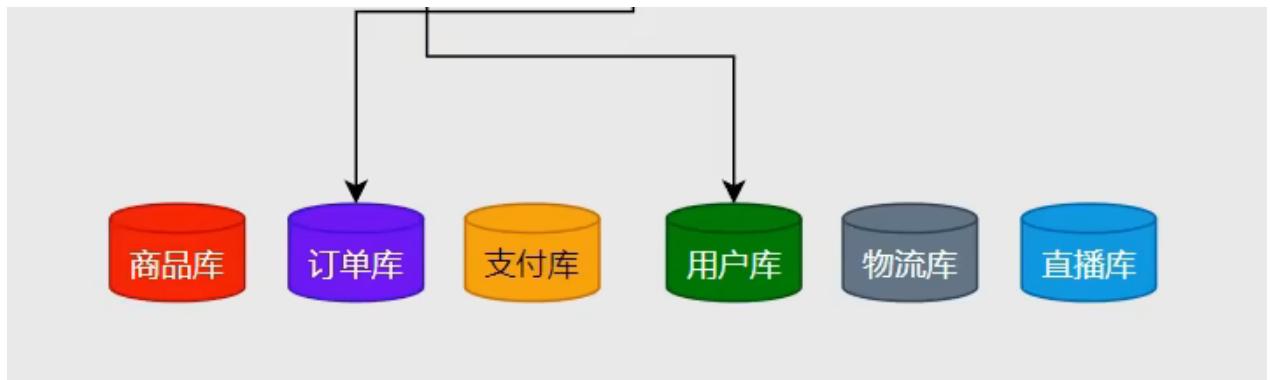
==> 如果某两个/多个不同的数据库需要进行的操作得同时成功/失败(如下单成功时会同时影响订单库和用户库), 该如何实现这种跨数据库的**事务**

==> 即我们需要实现**分布式事务**

==> 可以使用**Spring Cloud Alibaba Seata**







## 2. 环境配置:

- 使用父子工程(Maven多模块)的项目结构更好的管理微服务项目

在现代微服务架构中，**Spring Cloud** 提供了一整套工具和技术栈来简化分布式系统的开发。为了更好地组织和管理复杂的微服务项目，使用 **Maven 多模块 (父子工程<sup>+</sup>)** 是一种高效的方法。

**父子工程** 是 Maven 中的一种项目结构，通过一个父项目 (**Parent Project**) 管理和多个子项目 (**Module**)。父项目定义了所有子项目的通用配置和依赖，而子项目则继承这些配置并实现具体的功能模块。

### 主要优点

- 统一管理依赖:** 所有子项目共享相同的依赖版本。
- 集中配置:** 集中管理插件、属性和其他配置。
- 简化构建过程:** 使用一个命令即可构建所有子项目。
- 提高可维护性:** 修改配置或依赖只需在一个地方进行。

在分布式系统开发中，“副项目”通常指：

### 1. 子模块/子项目 (Submodule)

- 大型分布式系统被拆分成多个独立服务（微服务），每个服务作为一个独立项目
- 这些子项目共同组成完整的系统，称为“副项目”

### 2. 项目结构示例：

Bash

```
parent-project/      # 主项目 (聚合项目)
|—— user-service/   # 副项目①：用户服务
|—— order-service/  # 副项目②：订单服务
|—— payment-service/ # 副项目③：支付服务
└—— pom.xml         # 父级POM
```

### 3. 副项目特点：

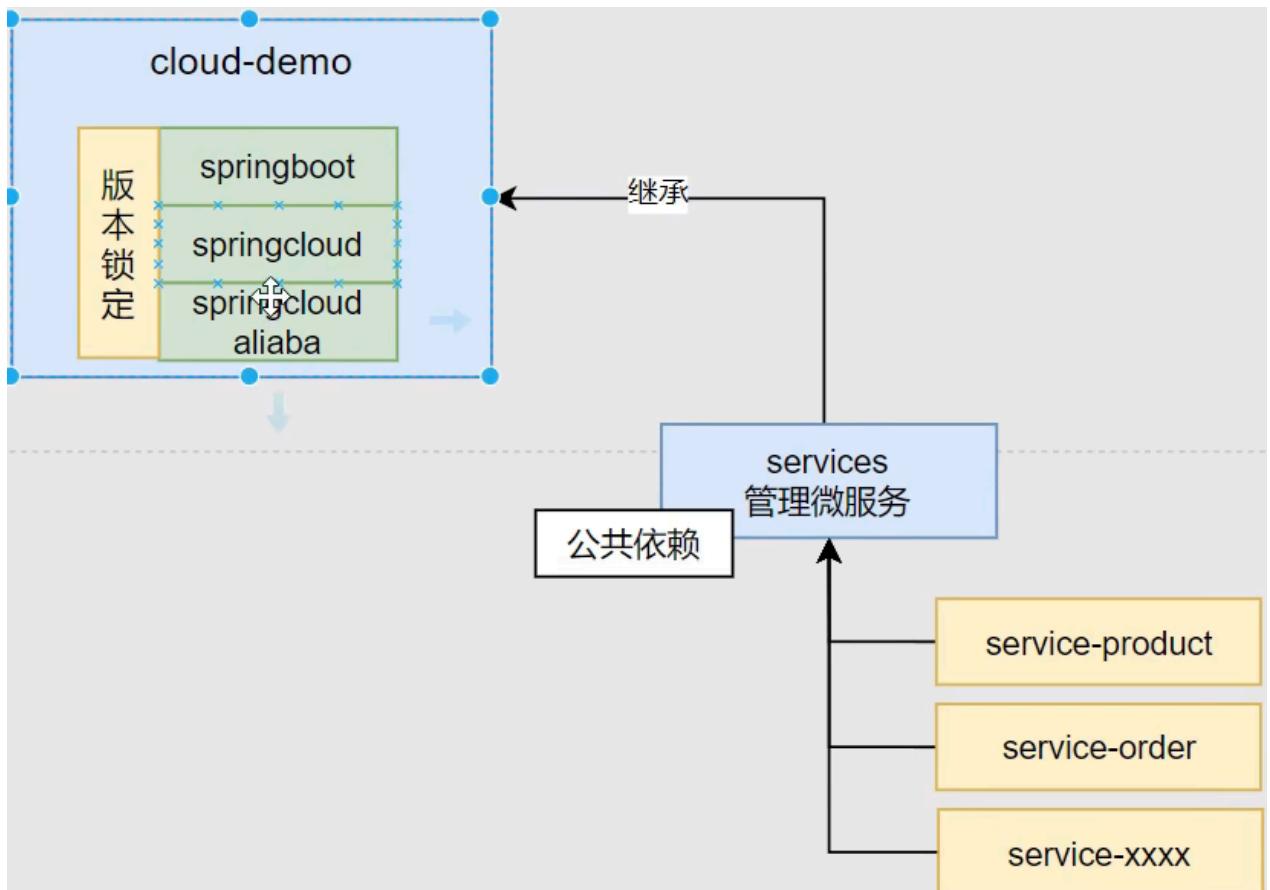
- 独立开发、构建和部署
- 有自己的代码库和生命周期
- 通过父POM继承公共配置

- 创建微服务架构项目
- 引入 SpringCloud、Spring Cloud Alibaba 相关依赖
- 注意版本适配

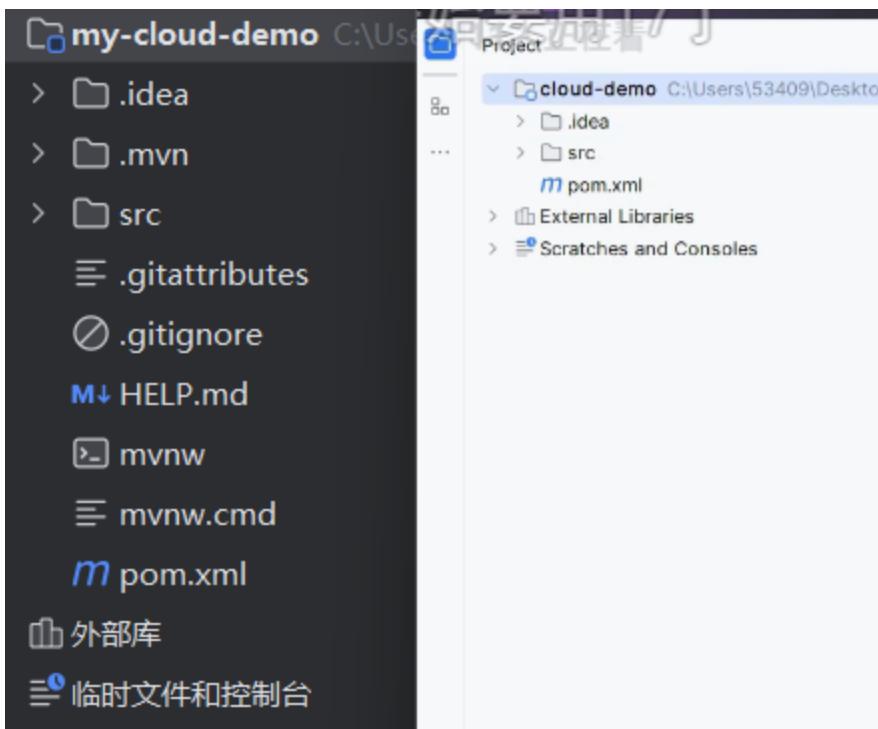
SpringBoot版本	SpringCloud版本	SpringCloud Alibaba版本
3.4.x +	2024.0.x	未适配
3.2.x - 3.3.x	2023.0.x	2023.0.*
3.0.2 - 3.2.x	2022.0.x	2022.0.*
2.6.x - 2.7.x	2021.0.x	2021.0.*
2.4.x - 2.5.x	2020.0.x	2020.0.*
2.3.x -	Hoxton/Greenwich -	2.2.* -

<https://github.com/alibaba/spring-cloud-alibaba/wiki/版本说明>

框架版本	SpringBoot 3.3.4	组件版本	Nacos 2.4.3
	SpringCloud 2023.0.3		Sentinel 1.8.8
	SpringCloud Alibaba 2023.0.3.2		Seata 2.2.0



- 删掉一些没用的文件



```
m pom.xml
4 <modelVersion>4.0.0</modelVersion>
5 <parent>
6   <groupId>org.springframework.boot</groupId>
7   <artifactId>spring-boot-starter-parent</artifactId>
8   <version>3.4.7</version>
9   <relativePath/> <!-- lookup parent from central -->
10 </parent>
11 <packaging>pom</packaging>
12 <groupId>com.example</groupId>
13 <artifactId>my-cloud-demo</artifactId>
14 <version>0.0.1-SNAPSHOT</version>
15 <name>my-cloud-demo</name>
16 <description>my-cloud-demo</description>
17 <url/>
18 <licenses>
```

- 添加了后,表示该项目为父项目,不生成实际构件(JAR/WAR)而只用于管理子模块和共享配置,父项目无需写代码,可以把src也删了,父项目还需要做依赖管理和插件统一配置,可以再

删如下部分

```
11 <packaging>pom</packaging>
12 <groupId>com.atguigu</groupId>
13 <artifactId>cloud-demo</artifactId>
14 <version>0.0.1-SNAPSHOT</version>
15 <name>cloud-demo</name>
16 <description>cloud-demo</description>
17 <url/>
18 <licenses>
19 | <license/>
20 </licenses>
21 <developers>
22 | <developer/>
23 </developers>
24 <scm>
25 | <connection/>
26 | <developerConnection/>
27 | <tag/>
28 | <url/>
29 </scm>
30 <properties>
31 | <java.version>17</java.version>
```

```
<properties>
    <java.version>17</java.version>
</properties>
<dependencies> ⚡ Edit Starters...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        Dependency Analyzer

```

这些依赖都先删掉

- 整合spring cloud和alibaba, 定义版本配置和依赖:

```
<properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <spring-cloud.version>2023.0.3</spring-cloud.version>
    <spring-cloud-alibaba.version>2023.0.3.2</spring-cloud-alibaba.version>
</properties>
```

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>${spring-cloud-alibaba.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

类型	文件扩展名	用途	内容结构	运行方式	典型场景
PO M	.pom	项目管理	XML 元数据文件	不可直接运行	多模块项目管理
JAR	.jar	Java 库/可执行应用	类文件 + 资源 + 元数据	java -jar 或类路径引用	工具库、微服务、桌面应用
WA R	.war	Web 应用程序	JAR + Web 资源 + 描述文件	部署到 Servlet 容器	传统 Java Web 应用

```
<scope>import</scope>
```

- **含义**: 依赖作用域
- **关键解析**:
  - `import` 是特殊作用域, **仅在 `<dependencyManagement>` 中有效**
  - 它会将 BOM 文件中定义的所有依赖版本导入当前项目的依赖管理
- **作用**: 继承 Spring Cloud 官方定义的所有组件版本, 无需手动指定每个组件的版本

### XML

```
<!-- 传统方式: 需手动指定每个依赖版本 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
    <version>4.1.1</version> <!-- 需单独维护 -->
</dependency>

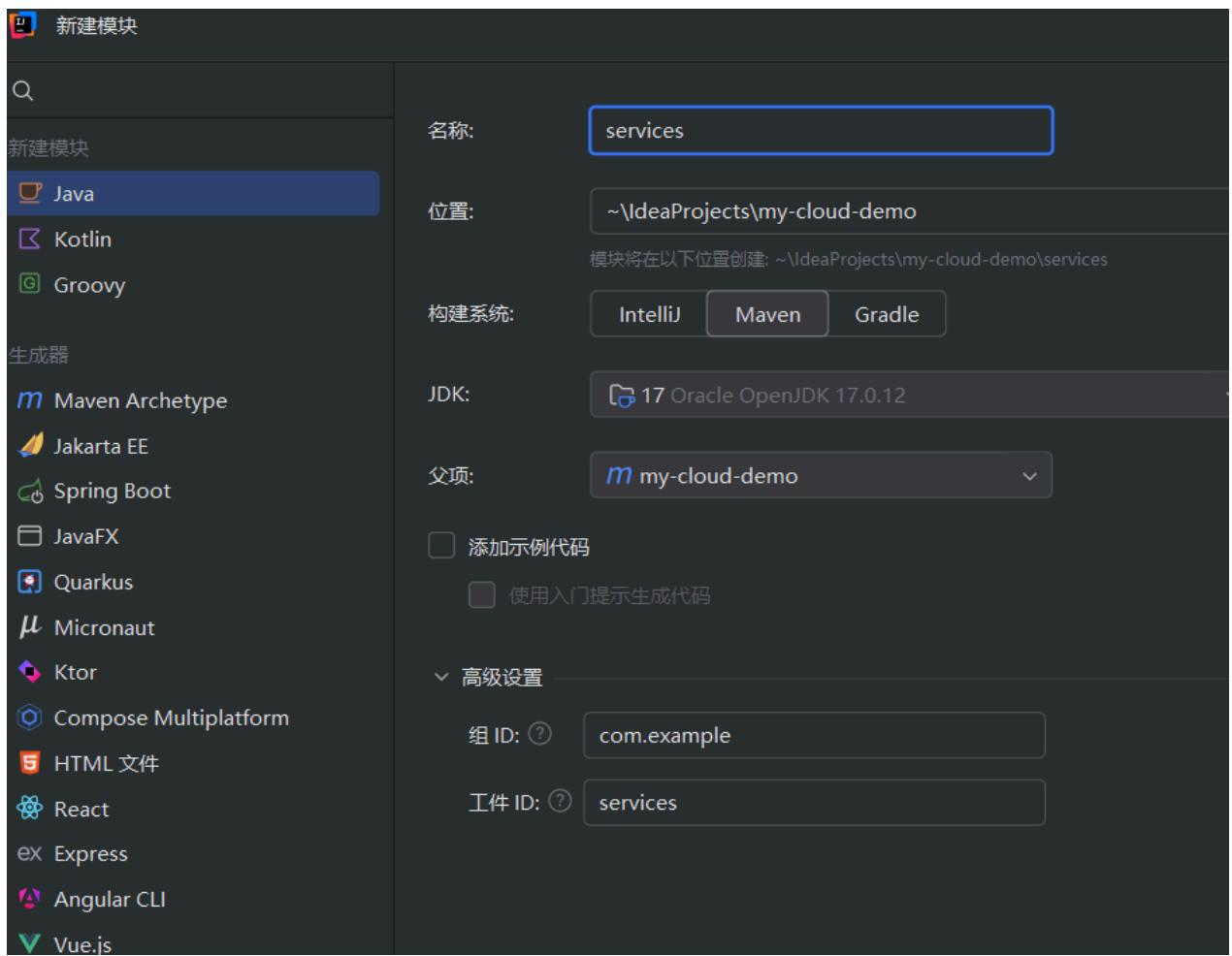
<!-- BOM方式: 只需在dependencyManagement中导入一次 -->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>2023.0.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

```
<maven.compiler.source>17</maven.compiler.source>
<maven.compiler.target>17</maven.compiler.target>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<spring-cloud.version>2023.0.3</spring-cloud.version>
<spring-cloud-alibaba.version>2023.0.3.2</spring-cloud-alibaba.version>
```

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>${spring-cloud-alibaba.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- 创建services继承父项目：

右键my-cloud-demo选择普通java而非springboot, services同样作为父项目管理子项目(所有services), 将打包改为pom并删除src

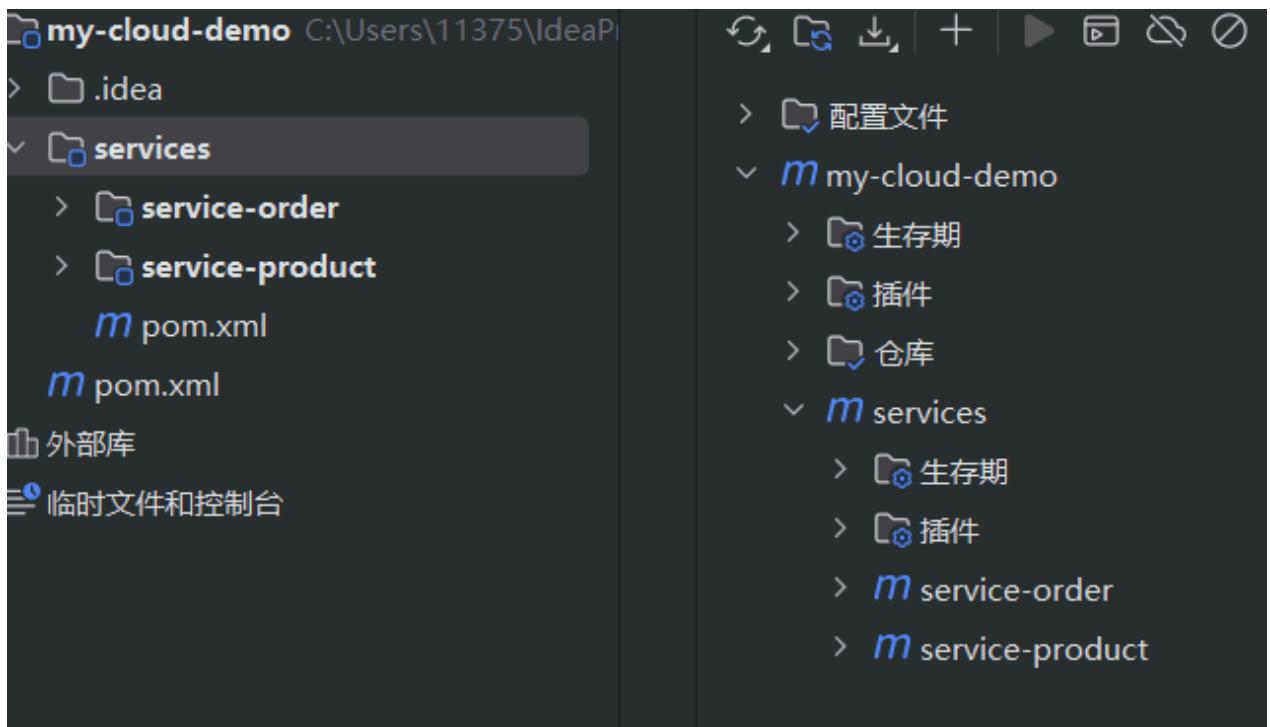


```
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
5      <modelVersion>4.0.0</modelVersion>
6      <parent>
7          <groupId>com.example</groupId>
8          <artifactId>my-cloud-demo</artifactId>
9          <version>0.0.1-SNAPSHOT</version>
10     </parent>
11
12     <packaging>pom</packaging>
13
14     <artifactId>services</artifactId>
15
16     <properties>
17         <maven.compiler.>
18             <maven.compiler.>
```

删除  
删除目录 "src"？  
"src"中的所有文件和子目录将被删除。  
您可能无法完全撤消此操作!

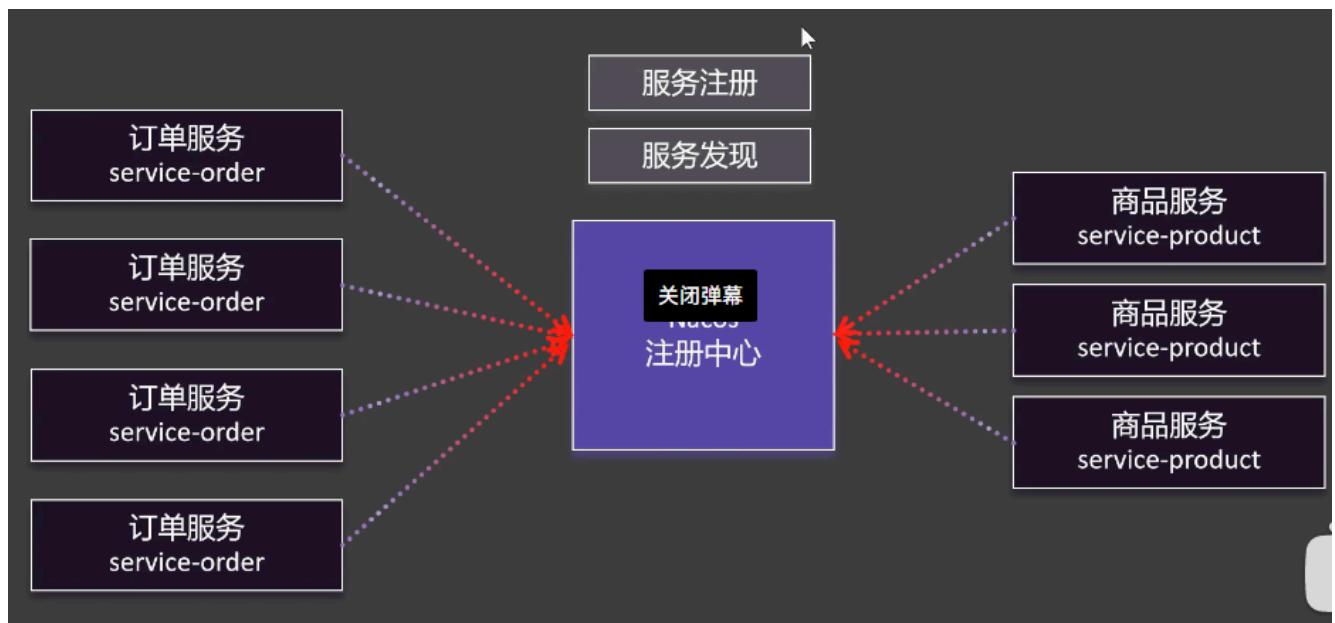
删除 取消

- 在services下创建两个子项目,项目结构图可在Maven中分组进行查看



### 3. Nacos(注册中心/配置中心):

#### 1. 环境配置:

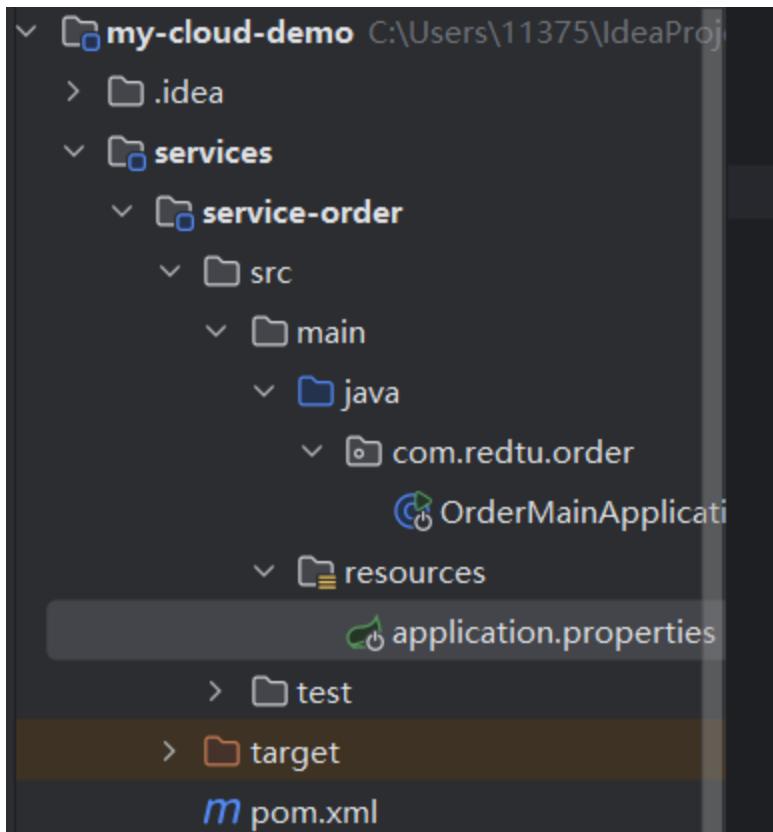


去官网下对应版本,然后解压进入bin页面,启动startup.cmd,即进入cmd界面,输入startup.cmd -m standalone启动单机模式(集群模式暂不介绍)

可进入localhost:8848/nacos进入管理页面

The screenshot shows the Nacos 2.4.3 Settings Center. The left sidebar has a dropdown menu with '模式 standalone' selected. The main content area displays a message about enabling authentication and includes sections for '样式主题' (with '明亮' selected), '命名空间样式' (with '标签' selected), and '系统语言' (with '中文' selected). A large blue '应用' button is at the bottom.

## 2. 注册-服务注册:



流程	内容	核心
步骤1	启动微服务	SpringBoot 微服务web项目启动
步骤2	引入服务发现依赖	spring-cloud-starter-alibaba-nacos-discovery
步骤3	配置Nacos地址	spring.cloud.nacos.server-addr=127.0.0.1:8848
步骤4	查看注册中心效果	访问 <a href="http://localhost:8848/nacos">http://localhost:8848/nacos</a>
步骤5	集群模式启动测试	单机情况下通过改变端口模拟微服务集群

- 导入web开发的依赖

OrderMainApplication.java      m pom.xml (service-order) ×

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    <properties>
        <maven.compiler.target>21</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies> ⚡ 添加启动器...
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
```

- 编写主程序

OrderMainApplication.java ×    m pom.xml (service-order)

```
1 package com.redtu.order;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class OrderMainApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(OrderMainApplication.class, args);
11     }
12 }
```

- 编写配置:占用端口,nacos地址,该微服务注册名称

OrderMainApplication.java      application.properties ×    m pom.xml

```
spring.application.name=service-order
server.port=8000

spring.cloud.nacos.server-addr=127.0.0.1:8848
```

- 启动服务,打开8848/nacos网页端可以看到注册成功(此处不同使用nacos的微服务需要在 application.properties 中配置server-port 使用不同的端口),当然得启动了然后会进行注册,如果关闭了那么也会把注册内容退出

The screenshot shows the Nacos service list interface. On the left, there's a sidebar with a tree view under the 'standalone' tab, showing '列表' (List) selected. The main area is titled '服务列表' (Service List) and has a 'public' tab selected. There are search and filter fields for '服务名称' (Service Name) and '分组名称' (Group Name). A table lists the services:

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
service-order	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
service-product	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

At the bottom, there are pagination controls: '每页显示: 10' (Items per page: 10), '总数: 2' (Total: 2), and navigation buttons: '< 上一页' (Previous page), '1' (Current page), and '下一页 >' (Next page).

- 模拟集群时,右键复制配置,然后点击修改选项 ==> 程序实参 ,在配置文件里面能写的这里都能写,将端口修改一下不占用同一端口就好:

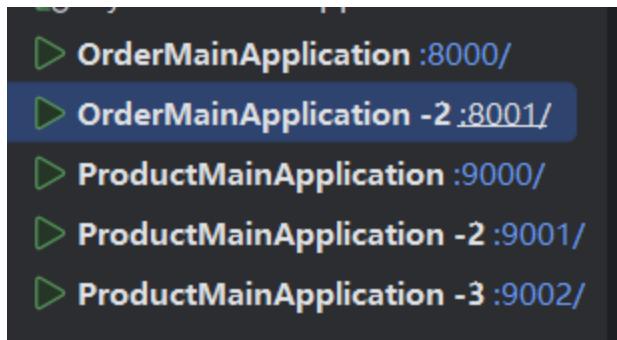
The screenshot shows the 'Edit Configuration' dialog in IntelliJ IDEA for the 'OrderMainApplication - 2' configuration. The 'Build and Run' section contains the command line:

```
java 17 'service-order' : -cp service-order com.redtu.order.OrderMain
```

The argument `--server.port=8001` is highlighted with a red box.

The right side of the dialog shows Java configuration options:

- Spring** section: 有效配置文件 (Valid configuration files) is checked.
- Java** section: 运行前不构建 (Build before run) is checked.
- Program arguments** (程序实参) is checked and highlighted with a red box.



服务列表

public

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
service-order	DEFAULT_GROUP	1	2	2	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
service-product	DEFAULT_GROUP	1	3	3	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

每页显示: 10 总数: 2 < 上一页 1 下一页

集群 DEFAULT 集群配置

元数据过滤 key value 添加过滤

IP	端口	临时实例	权重	健康状态	元数据	操作
192.168.31.140	8001	true	1	true	preserved.register.source=SPRING_CLOUD	<a href="#">编辑</a> <a href="#">下线</a>
192.168.31.140	8000	true	1	true	preserved.register.source=SPRING_CLOUD	<a href="#">编辑</a> <a href="#">下线</a>

### 3. 注册-服务发现:

流程	内容	核心
步骤1	开启服务发现功能	@EnableDiscoveryClient
步骤2	测试服务发现API	DiscoveryClient
步骤3	测试服务发现API	NacosServiceDiscovery

- 开启服务发现功能:添加注解@EnableDiscoveryClient(可自动化进行服务发现)

```
OrderMainApplication.java ×

1 package com.redtu.order;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
6
7 @EnableDiscoveryClient
8 @SpringBootApplication
9 public class OrderMainApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(OrderMainApplication.class, args);
13     }
14 }
```

- 测试准备:先导入测试依赖,再创建测试类(包名需与被测试类包名一致)

## 1. Spring Boot 的默认扫描规则

Java

```
@SpringBootApplication // 等价于  
@SpringBootConfiguration  
@EnableAutoConfiguration  
@ComponentScan(basePackages = "主启动类所在包")
```

- **测试类继承该规则**: 测试类默认扫描其**所在包及其子包**
- **风险点**: 测试类在不同包时, 被测试类可能不在扫描范围

## 2. 包名一致时的天然保障

```
src/main/java  
└── com.example.service  
    └── UserService.java  
  
src/test/java  
└── com.example.service // 相同包!  
    └── UserServiceTest.java // 自动扫描覆盖
```

**零配置注入**: 无需任何额外配置即可注入被测试类

```
<dependencies> ⚡ 添加启动器...  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-test</artifactId>  
        <scope>test</scope>  
    </dependency>
```

- 测试DiscoveryClient(spring+的标准规范)

```
12     public class DiscoveryTest {  
13  
14         @Autowired  
15         DiscoveryClient discoveryClient;  
16  
17         @Test  
18         void discoveryClientTest(){  
19             for(String service : discoveryClient.getServices()){  
20                 System.out.println("service" + service);  
21                 //获取ip和端口  
22                 List<ServiceInstance> instances = discoveryClient.getInstances(service);  
23                 for(ServiceInstance instance : instances){  
24                     System.out.println("ip" + instance.getHost() + ":" + instance.getPort());  
25                 }  
26             }  
27         }  
28     }  
29  
30     :  
31  
32     ✓ 测试已通过: 1共 1 个测试 - 1秒 693毫秒  
33  
34 2025-07-14T23:56:33.254+08:00 INFO 12132 --- [service-product] [main] c.a.n.c.a.AbstractNacosConfig...  
35 2025-07-14T23:56:33.255+08:00 INFO 12132 --- [service-product] [main] c.a.n.c.a.d.NacosConfig...  
36 serviceservice-order  
37 ip192.168.31.140:8001  
38 ip192.168.31.140:8000  
39 serviceservice-product  
40 ip192.168.31.140:9001  
41 ip192.168.31.140:9002  
42 ip192.168.31.140:9000
```

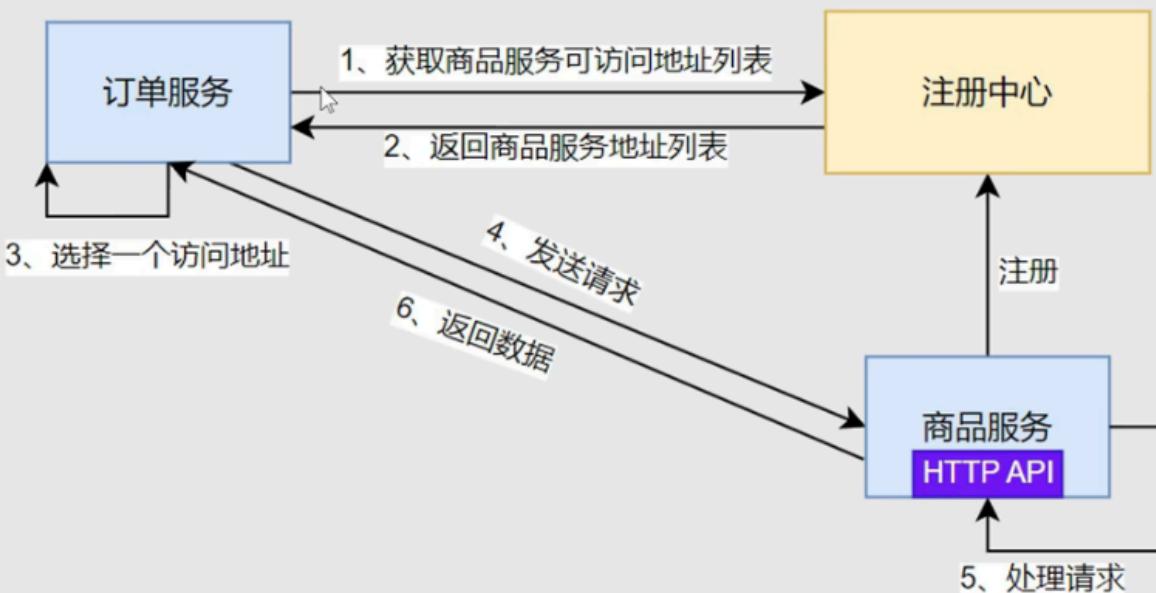
- 测试NacosServiceDiscovery(只有nacos引入时能用)

```
13     public class DiscoveryTest {
14
15         @Autowired
16         NacosServiceDiscovery nacosServiceDiscovery;
17
18         @Test
19         void nacosServiceDiscoveryTest() throws Exception{
20             for(String service : nacosServiceDiscovery.getServices()){
21                 System.out.println("service" + service);
22                 List<ServiceInstance> instances = nacosServiceDiscovery.getInstances(service);
23                 for(ServiceInstance instance : instances){
24                     System.out.println("ip" + instance.getHost() + ":" + instance.getPort());
25                 }
26             }
27         }
28     }
29
30
31
```

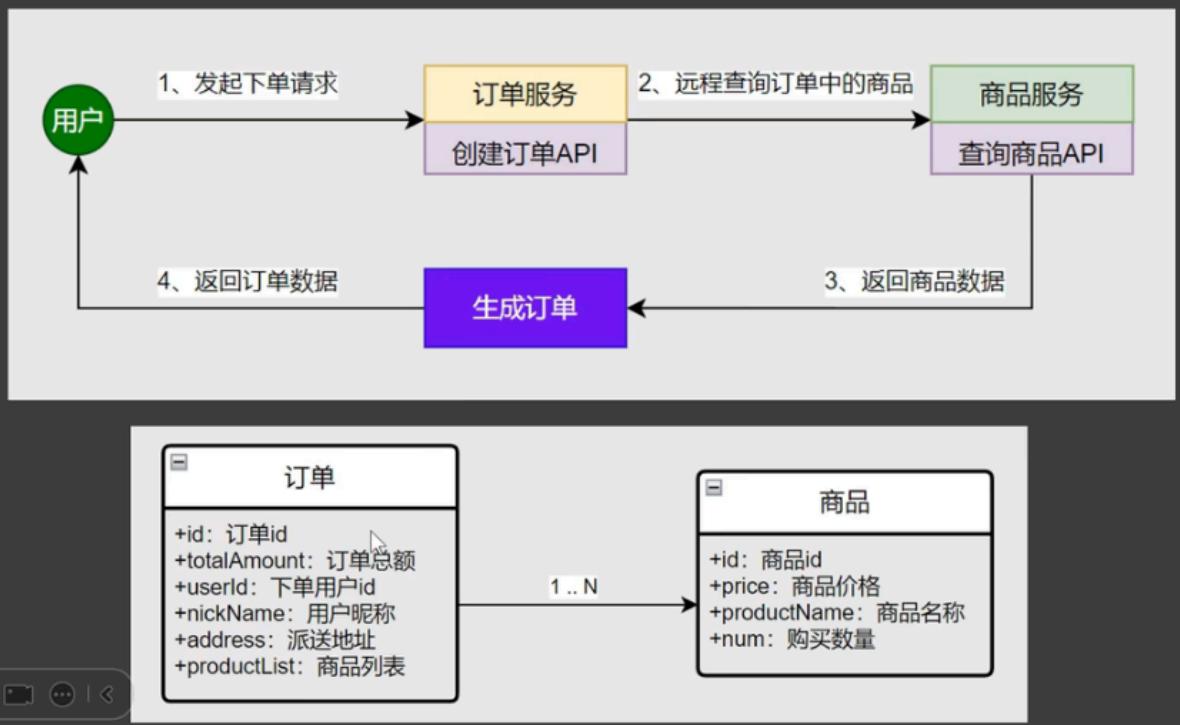
测试已通过: 1共 1 个测试 – 1秒 855毫秒

```
2025-07-15T00:01:54.098+08:00 INFO 27760 --- [service-product] [main] c.a.n.c.a.Abstract
2025-07-15T00:01:54.098+08:00 INFO 27760 --- [service-product] [main] c.a.n.c.a.d.NacosA
serviceservice-order
ip192.168.31.140:8001
ip192.168.31.140:8000
serviceservice-product
ip192.168.31.140:9001
ip192.168.31.140:9002
ip192.168.31.140:9000
```

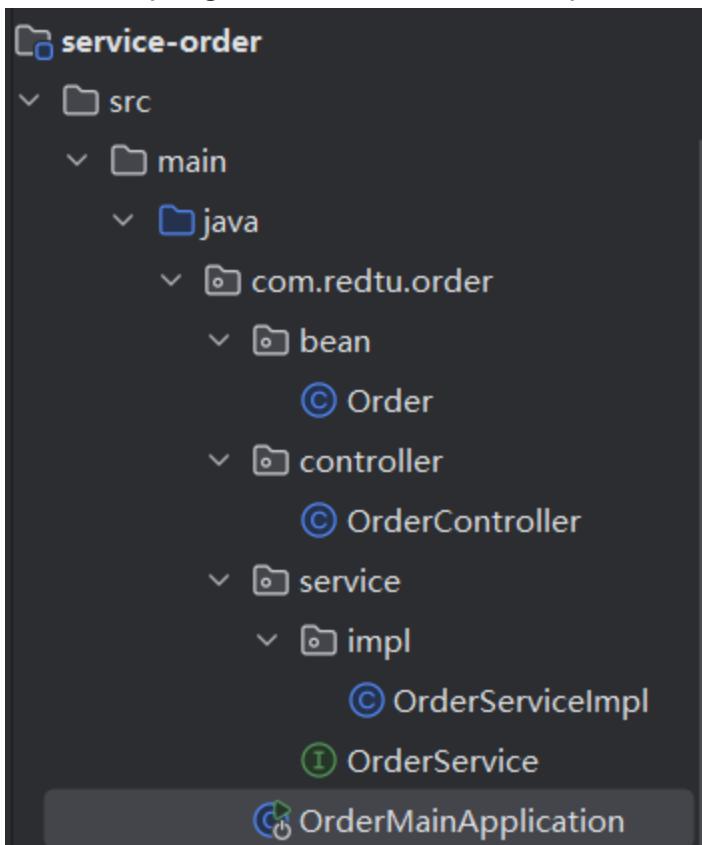
### 3. 注册-远程调用(后续使用openFeign):



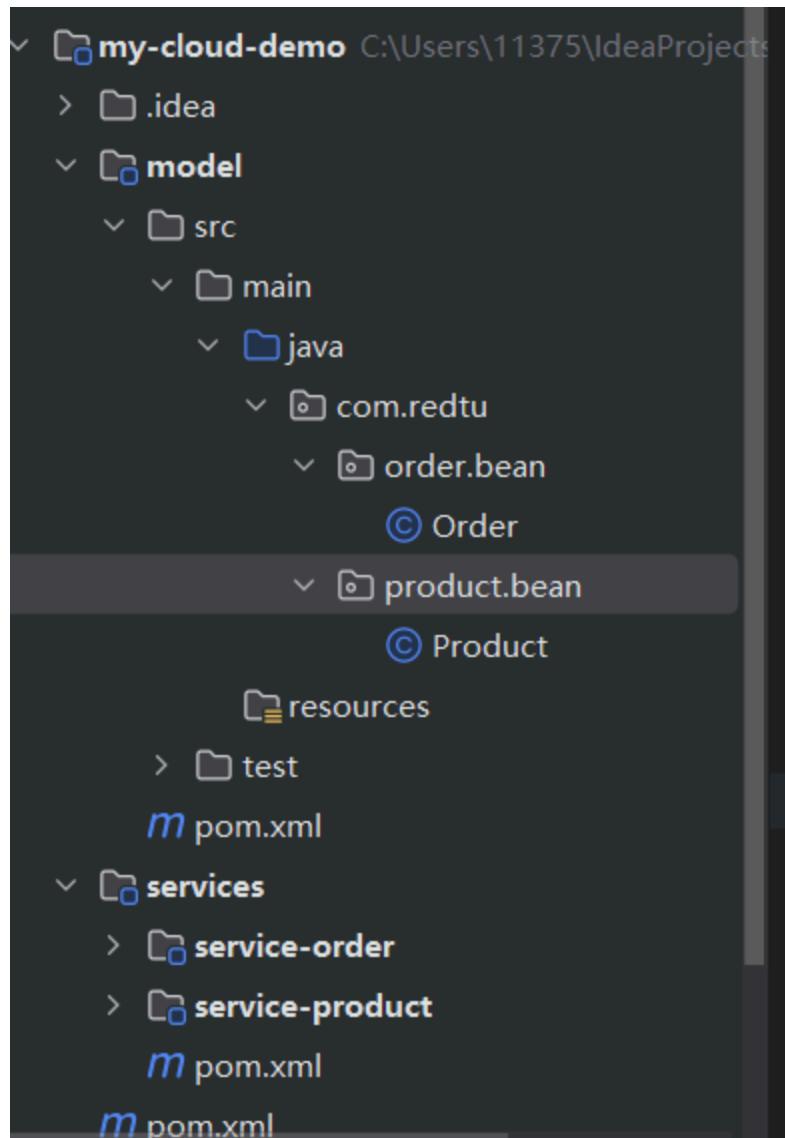
# 远程调用 - 下单场景



- 写上经典springboot三件套,在services的pom.xml导入lombok依赖



- 将bean统一打包到模型层model,并将model作为依赖导入到services的pom.xml中(让所有微服务都能调用模型)(此处作为依赖导入时,groupId和artifactId可修改但是需保持一致)



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5           http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <parent>
7     <groupId>com.example</groupId>
8     <artifactId>my-cloud-demo</artifactId>
9     <version>0.0.1-SNAPSHOT</version>
10    </parent>
11
12    <artifactId>model</artifactId>
13    <groupId>com.redtu</groupId>
14
15
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies> ⚡ 添加启动器...
    <dependency>
      <groupId>com.redtu</groupId>
      <artifactId>model</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>
  
```

- 原理:调url的数据获取(相当于自己对自己发起了一次http请求),此处为了更简单的演示,直接手动填了url

```
OrderServiceImpl.java  OrderController.java  OrderServiceConfig.java  OrderMainApplication.java
19  public class OrderServiceImpl implements OrderService {
27      public Order createOrder(Long productId, Long userId) {
33          order.setNickName("hxq");
34          order.setAddress("MoGanMountain");
35          order.setProductList(Arrays.asList(product));
36
37          return order;
38      }
39
40      private Product getProductFromRemote(Long productId){ 1个用法
41          List<ServiceInstance> instances = discoveryClient.getInstances(serviceId: "service-product");
42          ServiceInstance serviceInstance = instances.get(0);
43          String url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/product/" + productId;
44          log.info("url: " + url);
45          return restTemplate.getForObject(url, Product.class);
46      }
47  }
```

#### 4. 注册-负载均衡(后续的openFeign会自带负载均衡):

流程	内容	核心
步骤1	引入负载均衡依赖	spring-cloud-starter-loadbalancer
步骤2	测试负载均衡API	LoadBalancerClient
步骤3	测试远程调用	RestTemplate
步骤4	测试负载均衡调用	@LoadBalanced

思考：注册中心宕机，远程调用还能成功吗？

- 订单Order负载均衡调用别人，在order中添加loadbalancer依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

- 测试loadbalancer的choose是负载均衡的选一个，而DiscoveryClient是将所有都选出，需要自己写算法去实现负载均衡

```

15
16 @Test
17 public void test() {
18     ServiceInstance serviceInstance = loadBalancerClient.choose(serviceld: "service-order");
19     String url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/product/or
Config
ler
ation
20     System.out.println("url: " + url);
21     serviceInstance = loadBalancerClient.choose(serviceld: "service-order");
22     url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/product";
23     System.out.println("url: " + url);
24     serviceInstance = loadBalancerClient.choose(serviceld: "service-order");
25     url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/product";
26     System.out.println("url: " + url);
27     serviceInstance = loadBalancerClient.choose(serviceld: "service-order");
28     url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/product";
System.out.println("url: " + url);

```

截图工具

```

✓ 测试已通过: 1共 1 个测试 - 1秒 640毫秒
2025-07-16T01:27:02.977+08:00  INFO 7636 --- [service-order] [main] c.a.n.c.a.d.NacosAbilityManagerHolder
2025-07-16T01:27:02.977+08:00  INFO 7636 --- [service-order] [main] c.a.n.c.a.d.NacosAbilityManagerHolder
url: http://192.168.31.140:8000/product/order
url: http://192.168.31.140:8001/product
url: http://192.168.31.140:8000/product
url: http://192.168.31.140:8001/product
url: http://192.168.31.140:8000/product

```

- 应用:

```

OrderServiceImpl.java × pom.xml (service-order) OrderController.java OrderServiceConfig.java OrderMainApplication
20     public class OrderServiceImpl implements OrderService {
30         public Order createOrder(Long productId, Long userId) {
37             order.setAddress("MoGanMountain");
38             order.setProductList(Arrays.asList(product));
39
40             return order;
41         }
42
43         private Product getProductFromRemoteWithLoadBalance(Long productId){ 1个用法
44             ServiceInstance choose = loadBalancerClient.choose(serviceld: "service-product");
45             String url = "http://" + choose.getHost() + ":" + choose.getPort() + "/product/" + productId;
46             log.info("url: " + url);
47             return restTemplate.getForObject(url, Product.class);
48         }

```

```

▶ ProductMainApplication :9000/
▶ OrderMainApplication -2 :8001/
▶ ProductMainApplication -3 :9002/
▶ ProductMainApplication -2 :9001/
: Started OrderMainApplication in 2.789 seconds (prod)
: Initializing Spring DispatcherServlet 'dispatcherServlet'
: Initializing Servlet 'dispatcherServlet'
: Completed initialization in 0 ms
Impl : url: http://192.168.31.140:9002/product/1314
Impl : url: http://192.168.31.140:9000/product/1314
Impl : url: http://192.168.31.140:9001/product/1314
Impl : url: http://192.168.31.140:9002/product/1314
Impl : url: http://192.168.31.140:9000/product/1314
Impl : url: http://192.168.31.140:9001/product/1314
Impl : url: http://192.168.31.140:9002/product/1314

```

- 更简单的方式(使用spring注释),在远程调用(RPC)上添加@LoadBalance注解,将url的localhost:port,即ip-port部分直接用需要调用的微服务其在application.properties中定义的微服务名称替换即可

```
@Configuration  
public class OrderServiceConfig {  
  
    @LoadBalanced  
    @Bean  
    public RestTemplate restTemplate() {  
        return new RestTemplate();  
    }  
}
```

```
private Product getProductFromRemoteWithLoadBalanceAnnotation(Long productId){ 1个用法  
    String url = "http://service-product/product/" + productId;  
    log.info("url: " + url);  
    return restTemplate.getForObject(url, Product.class);  
}
```

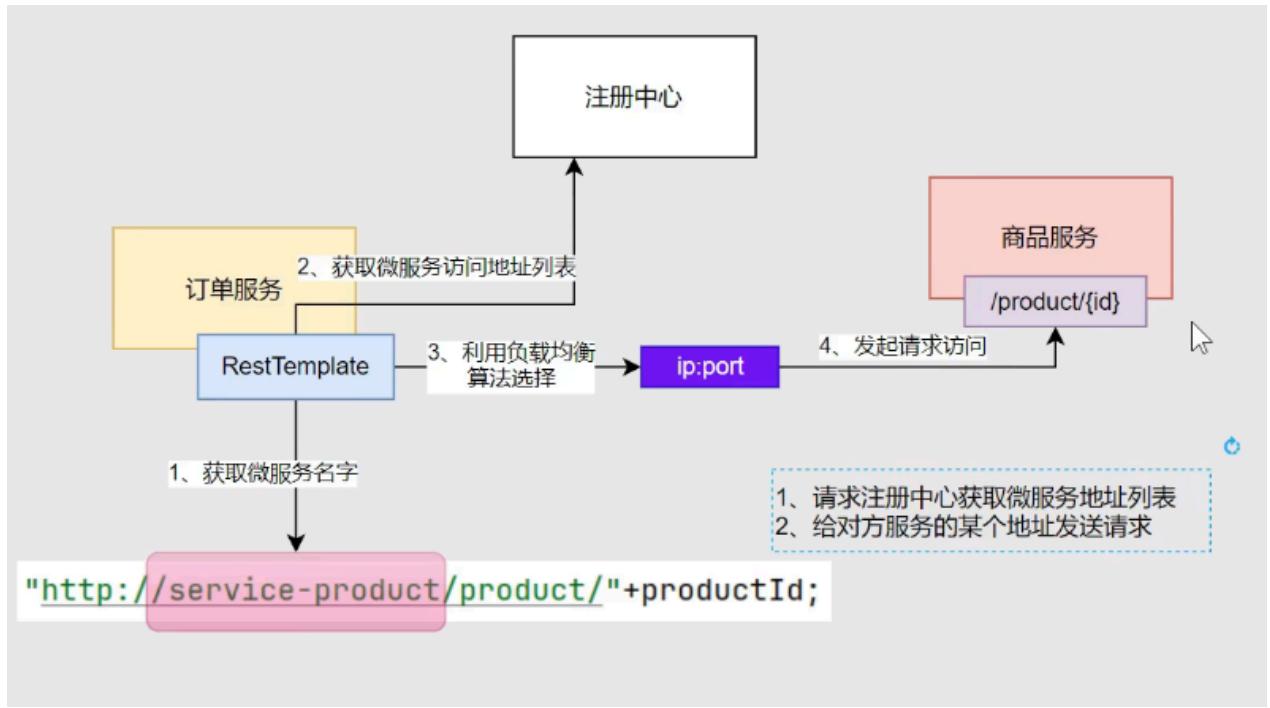
## 5. 注册-面试题:

- 问题:

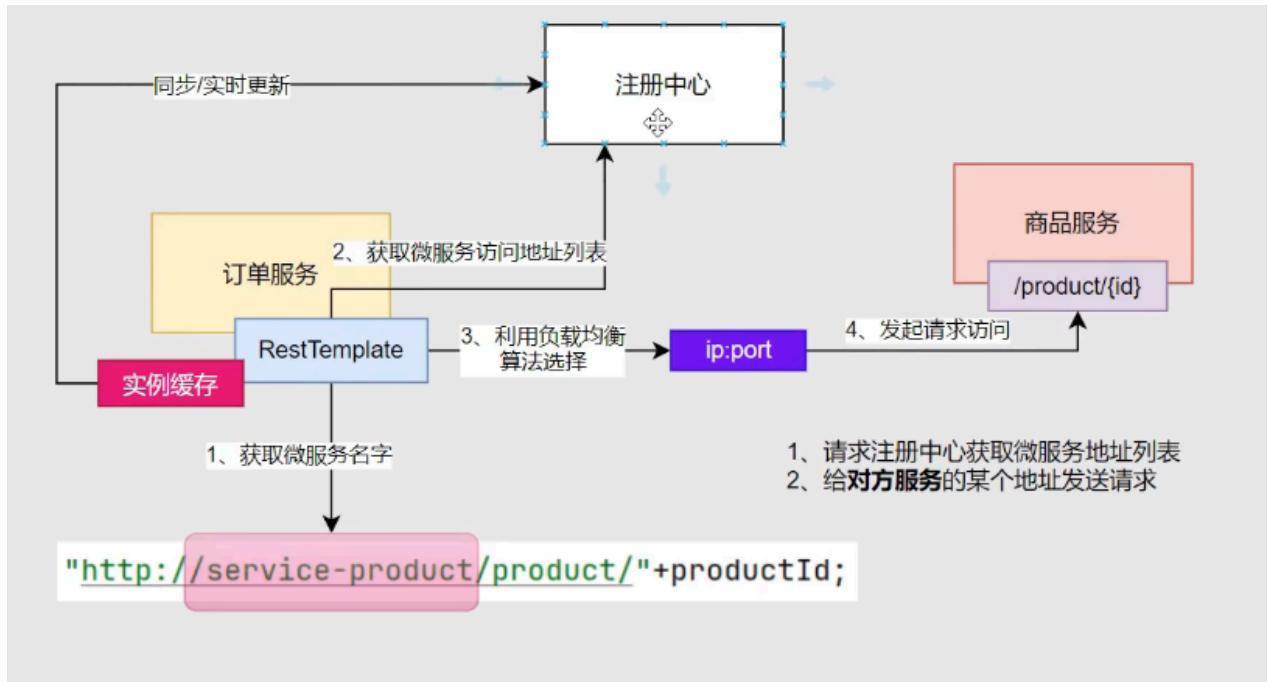
流程	内容	核心
步骤1	引入负载均衡依赖	spring-cloud-starter-loadbalancer
步骤2	测试负载均衡API	LoadBalancerClient
步骤3	测试远程调用	RestTemplate
步骤4	测试负载均衡调用	@LoadBalanced

思考：注册中心宕机，远程调用还能成功吗？

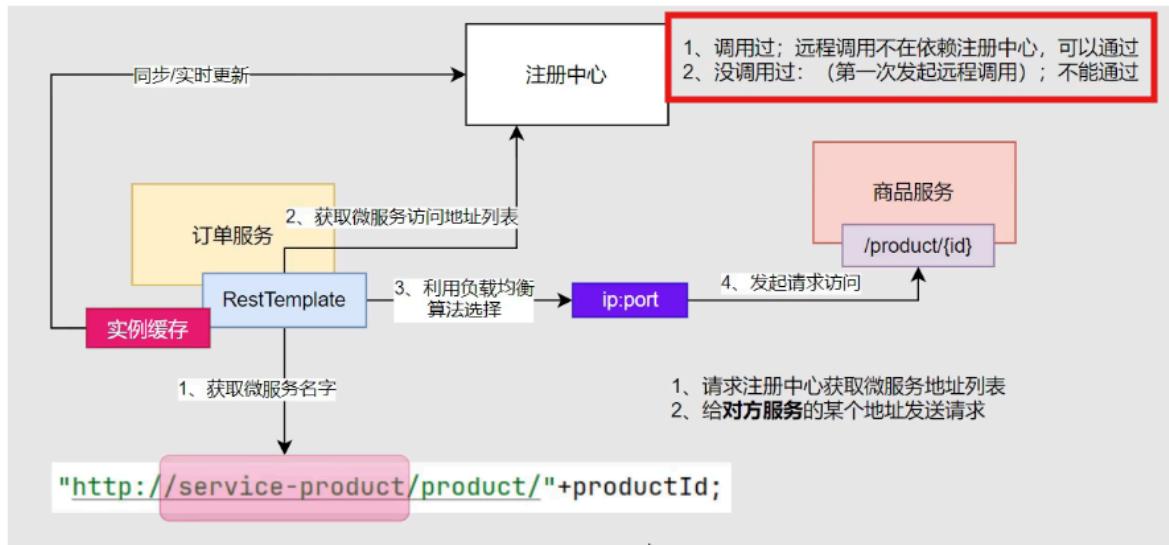
- 原有流程:



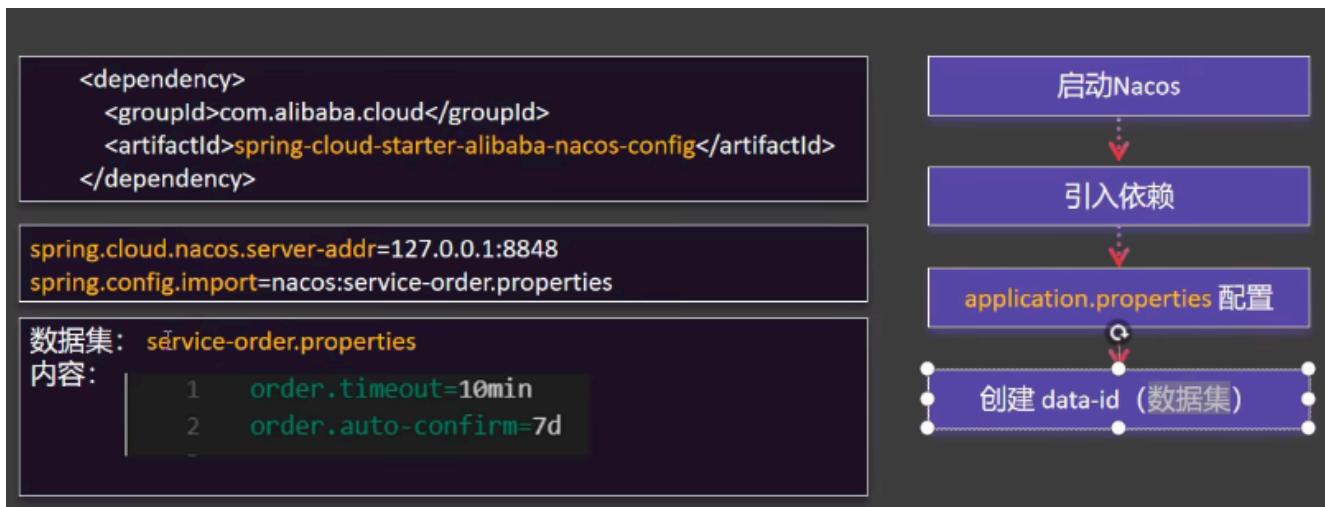
- 实际情况不会轻易宕机,没必要每次都实时访问注册中心,引入缓存与注册中心同步



- 对问题:



## 6. 配置-基本使用:



- 导入依赖:

```

m pom.xml (services) × ⚡ OrderServiceImpl.java

2 <project xmlns="http://maven.apache.org/POM/4.0.0"
20 <properties>
21     <maven.compiler.source>17</maven.compiler.source>
22     <maven.compiler.target>17</maven.compiler.target>
23     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
24 </properties>
25
26 <dependencies> ⚡ 添加启动器...
27     <!-- 配置中心-->
28     <dependency>
29         <groupId>com.alibaba.cloud</groupId>
30         <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
31     </dependency>

```

- 编辑配置nacos的config:即需要在nacos网页中编辑并发布配置(相当于统一定义了一些常量),同时需要在application.properties中导入对应的nacos配置名称(import配置),在获取配置的值添加注解@Value("\${xxx}")

## 编辑配置

\* 命名空间

\* Data ID **service-order.properties**

\* Group **DEFAULT\_GROUP**

[更多高级选项](#)

描述

Beta发布  默认不要勾选。

配置格式  TEXT  JSON  XML  YAML  HTML  Properties

配置内容②:

```

1 order.timeout=30min
2 order.auto-confirm=1314d

```

```

spring.application.name=service-order
server.port=8000
spring.cloud.nacos.server-addr=127.0.0.1:8848
spring.config.import=nacos:service-order.properties

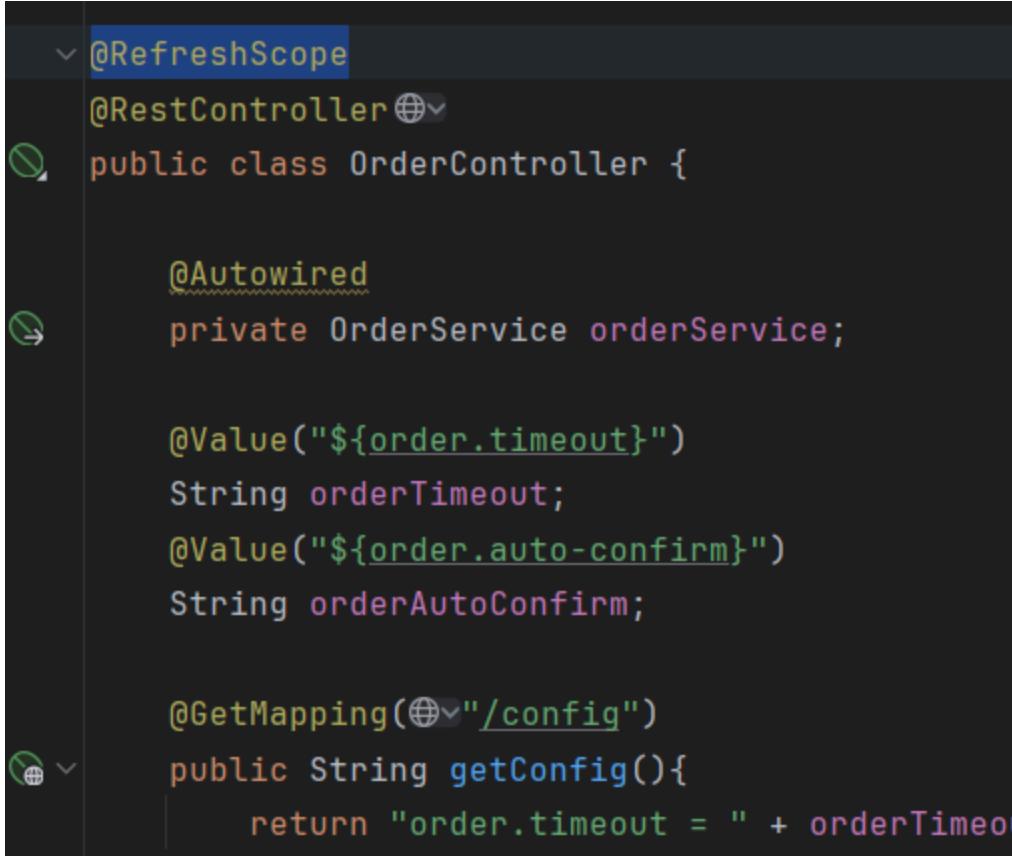
```

```

    @Value("${order.timeout}")
    String orderTimeout;
    @Value("${order.auto-confirm}")
    String orderAutoConfirm;

```

- 实时更新:在需要调用配置的类中添加@RefreshScope注解,在nacos中更改配置时,会将修改的配置参数实时更新



```

    @RefreshScope
    @RestController
    public class OrderController {

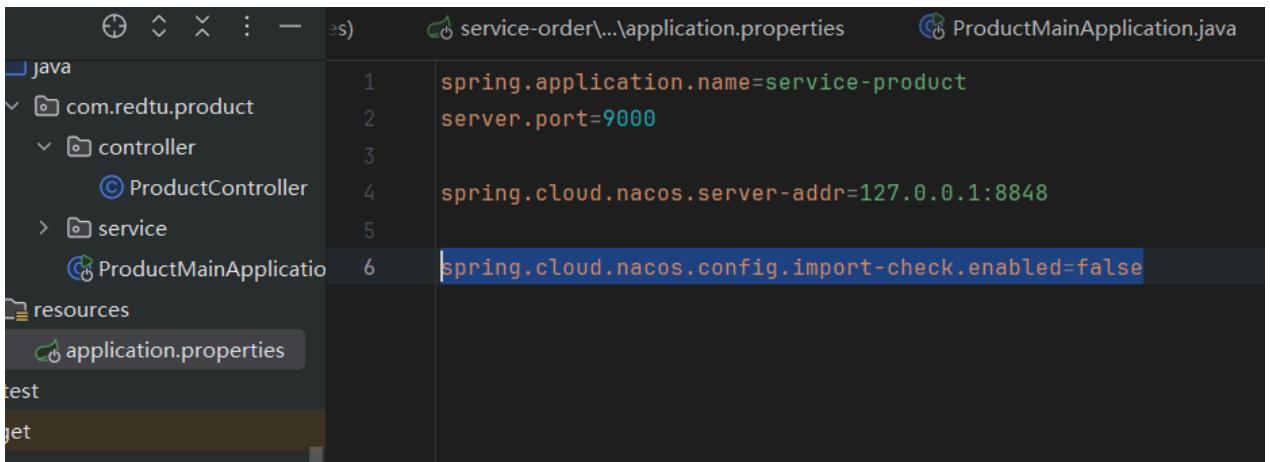
        @Autowired
        private OrderService orderService;

        @Value("${order.timeout}")
        String orderTimeout;
        @Value("${order.auto-confirm}")
        String orderAutoConfirm;

        @GetMapping("/config")
        public String getConfig(){
            return "order.timeout = " + orderTimeout;
        }
    }

```

- 报错解决:忽略配置检查,在一些暂时用不到nacos配置的微服务中,可以添加**spring.cloud.nacos.config.import-check.enabled=false**,即禁用导入检查,使得忽略nacos的config的import检查(需要使用配置时也可以直接像刚才那样导入就行)



File	Content
application.properties	<pre> spring.application.name=service-product server.port=9000 spring.cloud.nacos.server-addr=127.0.0.1:8848 spring.cloud.nacos.config.import-check.enabled=false </pre>
ProductMainApplication.java	

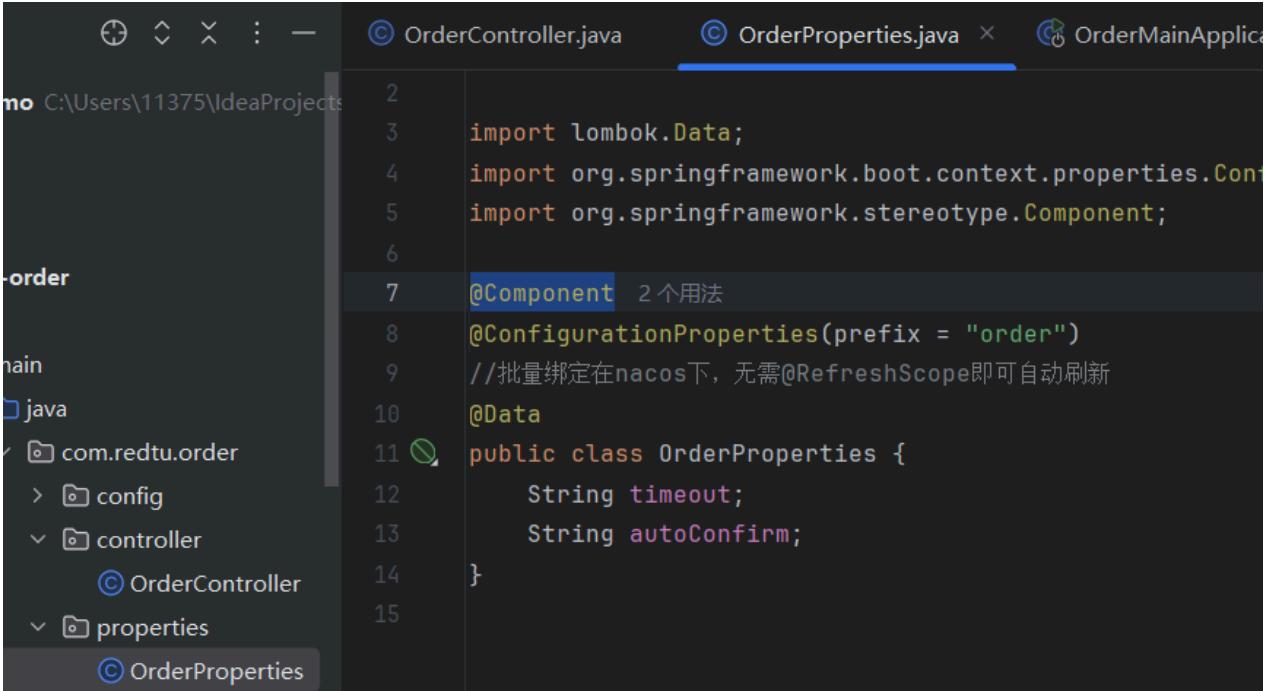
## 7. 配置-动态刷新与配置监听

### 配置中心 - 动态刷新

#### • 使用步骤

- `@Value("${xx}")` 获取配置 + `@RefreshScope` 实现自动刷新
- `@ConfigurationProperties` 无感自动刷新
- `NacosConfigManager` 监听配置变化

- 集成配置项并注入:(需要添加注解@Component才能被spring扫描到)



The screenshot shows an IDE interface with several tabs at the top: OrderController.java, OrderProperties.java (which is currently selected), and OrderMainApplication.java. The left sidebar shows a project structure with packages like com.redtu.order, config, controller, and properties. The OrderProperties.java file contains the following code:

```
2 import lombok.Data;
3 import org.springframework.boot.context.properties.ConfigurationProperties;
4 import org.springframework.stereotype.Component;
5
6
7 @Component 2个用法
8 @ConfigurationProperties(prefix = "order")
9 //批量绑定在nacos下，无需@RefreshScope即可自动刷新
10 @Data
11 public class OrderProperties {
12     String timeout;
13     String autoConfirm;
14 }
15
```

```
© OrderController.java × © OrderProperties.java © OrderMainApplication.java ©  
16  public class OrderController {  
22      //     String orderTimeout;  
23      //     @Value("${order.auto-confirmed}")  
24      //     String orderAutoConfirm;  
25  
26      @Autowired  
27  ⚡ private OrderProperties orderProperties;  
28  
29      @GetMapping(PathVariable("config"))  
30  ⚡ public String getConfig(){  
31          return "order.timeout = " + orderProperties.getTimeout() +  
32      }  
33  
34      @GetMapping(PathVariable("create"))
```

```
© OrderController.java × © OrderProperties.java © OrderMainApplication.java ©  
16  public class OrderController {  
22      //     String orderTimeout;  
23      //     @Value("${order.auto-confirmed}")  
24      //     String orderAutoConfirm;  
25  
26      @Autowired  
27  ⚡ private OrderProperties orderProperties;  
28  
29      @GetMapping(PathVariable("config"))  
30  ⚡ public String getConfig(){  
31          return "order.timeout = " + orderProperties.getTimeout() +  
32      }  
33  
34      @GetMapping(PathVariable("create"))
```

- 在OrderMainApplication设置监听器:

```
17 public class OrderMainApplication {  
18     //1、项目启动就监听配置文件变化  
19     //2、发生变化后拿到变化值  
20     //3、发送邮件  
21     @Bean  
22     ApplicationRunner applicationRunner(NacosConfigManager nacosConfigManager) {  
23         return ApplicationArguments args -> {  
24             ConfigService configService = nacosConfigManager.getConfigService();  
25             configService.addListener("service-order.properties", "DEFAULT_GROUP", new Listener() {  
26                 @Override  
27                 public Executor getExecutor() {  
28                     return Executors.newFixedThreadPool(4);  
29                 }  
30                 @Override 0个用法  
31                 public void receiveConfigInfo(String configInfo) {  
32                     System.out.println("变化的配置信息"+configInfo);  
33                     System.out.println("假装发送了邮件~");  
34                 }  
35             });  
36         };  
37     }  
38 };
```

控制台

```
变化的配置信息order.timeout=1314min  
order.auto-confirm=520d  
假装发送了邮件~
```

## 8. 配置-面试题：

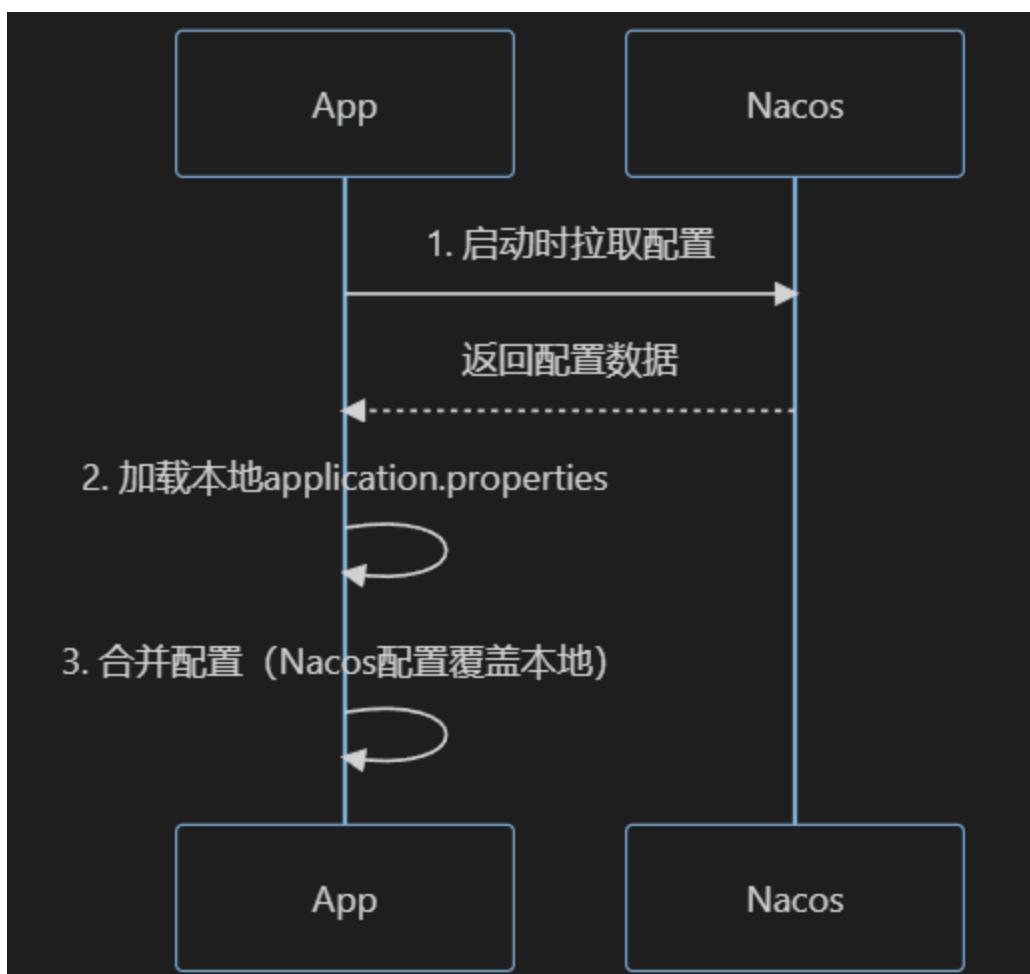
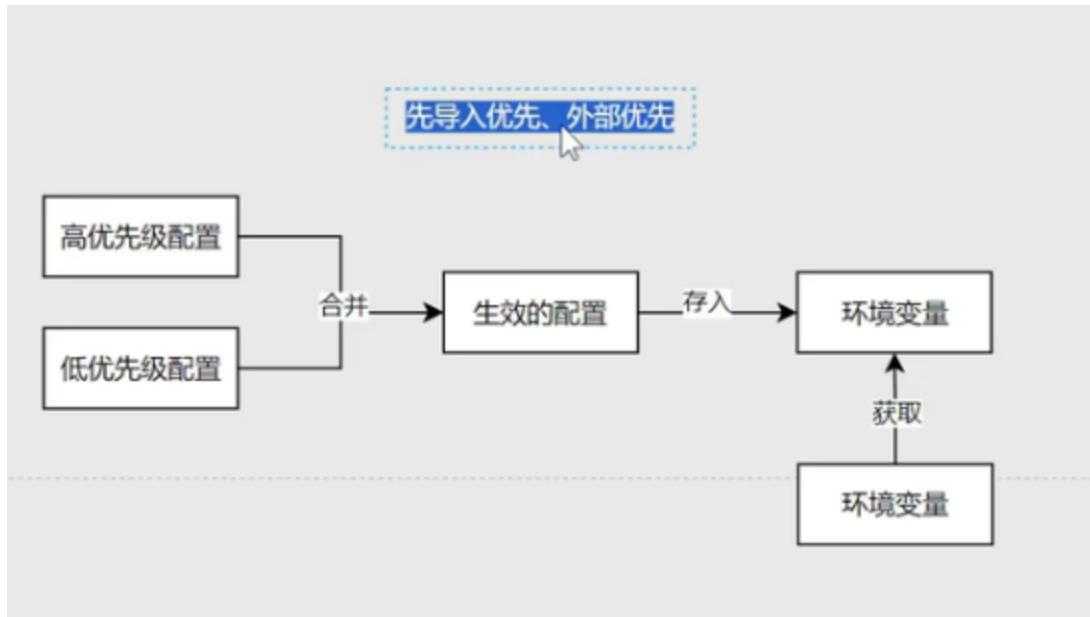
### 配置中心 - 动态刷新

#### • 使用步骤

- `@Value("${xx}")` 获取配置 + `@RefreshScope` 实现自动刷新
- `@ConfigurationProperties` 无感自动刷新
- `NacosConfigManager` 监听配置变化

思考： Nacos中的数据集 和 `application.properties` 有相同的配置项，哪个生效？

- nacos优先级>本地,本地import导入时遵循：后加载者覆盖先加载者



### 1. Nacos > 本地规则:

远程配置中心的配置优先级永远高于本地文件

### 2. 本地 import 覆盖规则:

- 在单个 `import` 声明中，右侧文件配置覆盖左侧文件配置
- 主 `application.properties` 覆盖所有 `import` 导入配置

## 8. 配置-数据隔离与环境切换:

### 配置中心 - 数据隔离

#### • 需求描述

- 项目有多套环境: `dev`, `test`, `prod`
- 每个微服务，同一种配置，在每套环境的值都不一样。
  - 如: `database.properties`
  - 如: `common.properties`
- 项目可以通过切换环境，加载本环境的配置

#### • 难点

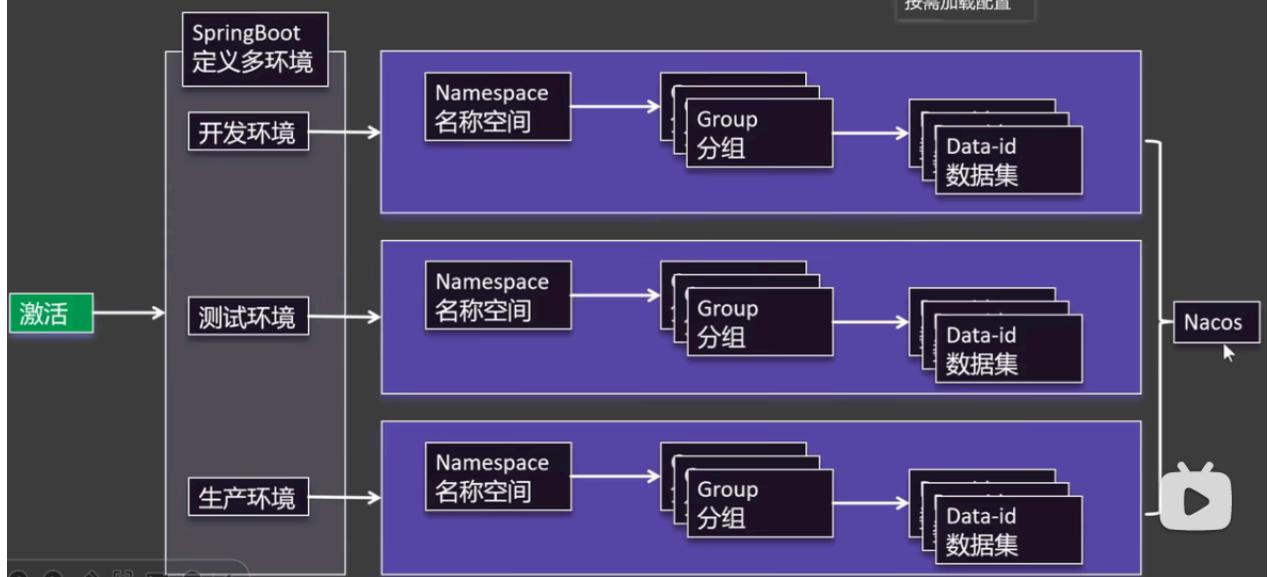
- 区分多套环境
- 区分多种微服务
- 区分多种配置
- 按需加载配置

- 原理: 名称空间(namespace)(如public)区分多套环境, 分组(group)区分多种微服务, 由数据集(data-id)区分多种配置, 最后springboot去激活对应的环境

# 配置中心 - 数据隔离

区分多种微服务  
区分多种配置  
按需加载配置

尚硅谷 bilibili



- 在命名空间创建多套环境,在配置中心配置不同环境的配置

## 命名空间

新建命名空间	刷新			
命名空间名称	命名空间ID	描述	配置数	操作
public(保留空间)			1	<a href="#">详情</a> <a href="#">删除</a> <a href="#">编辑</a>
dev	dev	开发	3	<a href="#">详情</a> <a href="#">删除</a> <a href="#">编辑</a>
prod	prod	生产	3	<a href="#">详情</a> <a href="#">删除</a> <a href="#">编辑</a>
test	test	测试	3	<a href="#">详情</a> <a href="#">删除</a> <a href="#">编辑</a>

## 配置管理

命名空间

开发

public | dev | prod | test

创建配置

Data ID

已开启默认模糊查询

Group

已开启默认模糊查询

默认模糊匹配

查询到 3 条满足要求的配置。

<input type="checkbox"/>	Data Id	Group	格式	归属应用
<input type="checkbox"/>	common.properties	order	Properties	
<input type="checkbox"/>	database.properties	order	Properties	
<input type="checkbox"/>	common.properties	product	Properties	

删除

克隆

导出

- 动态切换(按需加载):把application.properties整改为application.yml

The screenshot shows a code editor with several tabs at the top: application.yml, OrderProperties.java, OrderController.java, and OrderMainApplication.java. The application.yml tab is active, displaying the following configuration:

```
1 server:
2   port: 8000
3 spring:
4   profiles:
5     active: dev
6     #目前由(手动?)动态调配使用哪套配置环境
7   application:
8     name: service-order
9   cloud:
10    nacos:
11      server-addr: 127.0.0.1:8848
12      #原导入nacos服务ip-port
13      config:
14        namespace: ${spring.config.activate:dev}
15        #动态取值与默认值: ${}
16        import-check:
17          enabled: false
18          #可以设置不检查有没有 没有import数据集 的情况(一般情况下回报错)
```

```
---  
#多文件写法配置不同环境的配置-实际开发过程中还是多文件比较好吧（  
  
spring:  
    config:  
        import:  
            #导入 对应环境 下的 对应分组的对应数据集  
            - nacos:common.properties?group=order  
            - nacos:database.properties?group=order  
        activate:  
            #设置对应环境  
            on-profile: dev  
  
---  
  
spring:  
    config:  
        import:  
            - nacos:common.properties?group=order  
            - nacos:database.properties?group=order  
            - nacos:love.properties?group=order  
        activate:  
            on-profile: test  
  
---  
  
spring:  
    config:  
        import:  
            - nacos:common.properties?group=order  
            - nacos:database.properties?group=order  
            - nacos:love.properties?group=order  
        activate:  
            on-profile: prod
```

## 9. 总结:



## 4. OpenFeign(远程调用):

### 1. 基础使用(声明式API):

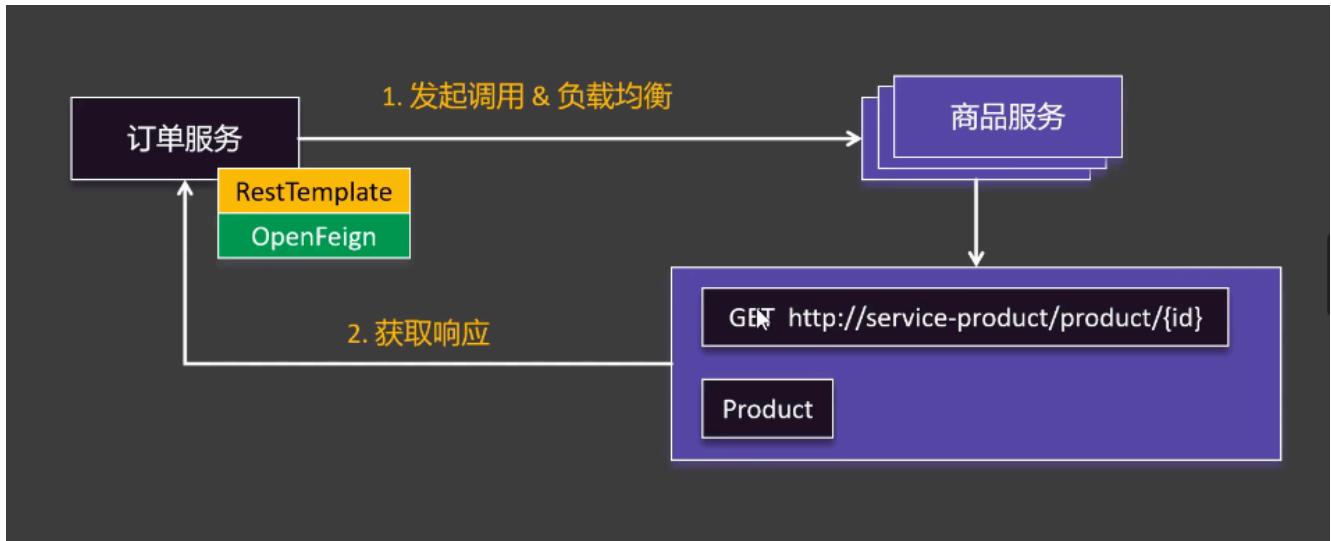
声明式API:三步走,添加依赖->添加注解->将功能分离出来形成一个独立的类,使用时  
@Autowried注入该类即可

#### Declarative REST Client

尚硅谷

- **声明式 REST 客户端** vs **编程式 REST 客户端** (`RestTemplate`)
- **注解驱动**
  - 指定远程地址: `@FeignClient`
  - 指定请求方式: `@GetMapping`、`@PostMapping`、`@DeleteMapping` ...
  - 指定携带数据: `@RequestHeader`、`@RequestParam`、`@RequestBody` ...
  - 指定结果返回: 响应模型

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```



- 引入依赖并在controller上添加注解@EnableFeignClients启用远程调用功能:

```

application.yml      pom.xml (services)      OrderProperties.java      OrderController.java
 2   <project xmlns="http://maven.apache.org/POM/4.0.0"
 26     <dependencies> ⏪ 添加启动器...
 33     <dependency>
 34       <groupId>com.redtu</groupId>
 35       <artifactId>model</artifactId>
 36       <version>0.0.1-SNAPSHOT</version>
 37     </dependency>
 38     <!--服务发现-->
 39     <dependency>
 40       <groupId>com.alibaba.cloud</groupId>
 41       <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
 42     </dependency>
 43
 44     <!--远程调用-->
 45     <dependency>
 46       <groupId>org.springframework.cloud</groupId>
 47       <artifactId>spring-cloud-starter-openfeign</artifactId>
 48     </dependency>
 49

```

- 创建远程调用类(添加@FeignClient注解,并添加被远程调用的微服务名称-在application.yml中定义的,该注解同时会自动启用负载均衡功能):

```
application.yml      pom.xml (services)      OrderController.java      PruductFeignClient.java      Order
C:\Users\11375\IdeaProje
er
va
] com.redtu.order
  config
  controller
  feign
    PruductFeignClient
  properties
  service
1 package com.redtu.order.feign;
2
3 import org.springframework.cloud.openfeign.FeignClient;
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.PathVariable;
6 import org.springframework.web.bind.annotation.RequestHeader;
7
8 @FeignClient(value = "service-product")//远程调用的微服务名称 0个用法
9 public interface PruductFeignClient {
10
11     //mvc注解的两套使用逻辑(由该类的注解决定)
12     //1、标注在Controller上,是接受请求
13     //2、标注在FeignClient上,是发送请求
14     @GetMapping(@RequestMapping("/product/{id}"))
15     void getProductById(@PathVariable("id") Long id , @RequestHeader String token);
16 }
17
```

- 因为返回的是json,可以设置返回类型,自动转换,如product

```
//mvc注解的两套使用逻辑(由该类的注解决定)
//1、标注在Controller上,是接受请求
//2、标注在FeignClient上,是发送请求
@GetMapping(@RequestMapping("/product/{id}"))
Product| getProductById(@PathVariable("id") Long id );
```

## 2.调用第三方API:

- 与调用自家的api相似,不同之处在于,此时在@FeignClient注解后面的名称可以自定义(因为第三方api没在自家定义微服务名称),但是后面需要添加url指定地址,其余同调用自家api

```
@FeignClient(value = "weather-client", url = "http://aliv18.data.moji.com")
public interface WeatherFeignClient {

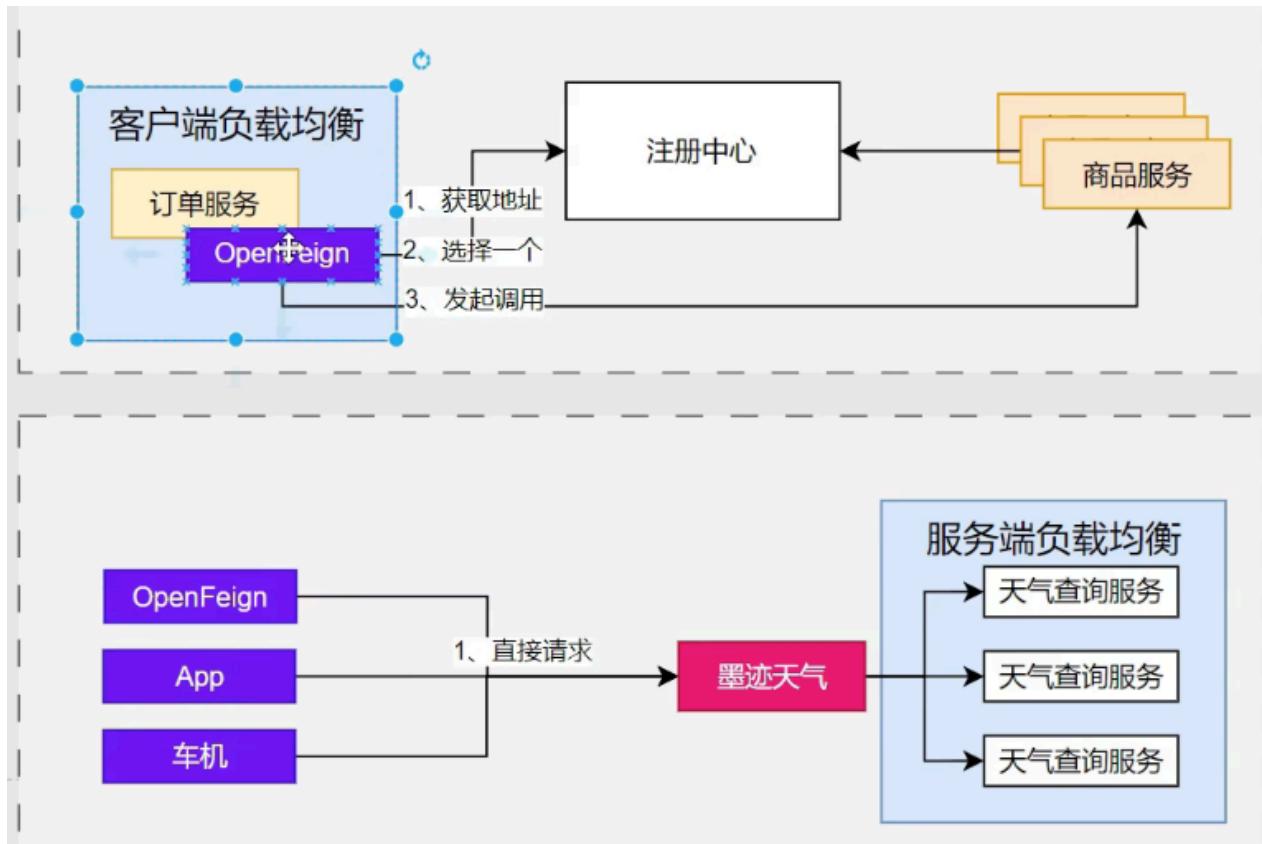
    @PostMapping(@RequestMapping("/whapi/json/alicityweather/condition"))
    String getWeather(@RequestHeader("Authorization") String auth,
                      @RequestParam("token") String token,
                      @RequestParam("cityId") String cityId);
}
```

### 3.面试题:

- 小技巧：如何编写好OpenFeign声明式的远程调用接口
  - 业务API：直接复制对方Controller签名即可
  - 第三方API：根据接口文档确定请求如何发

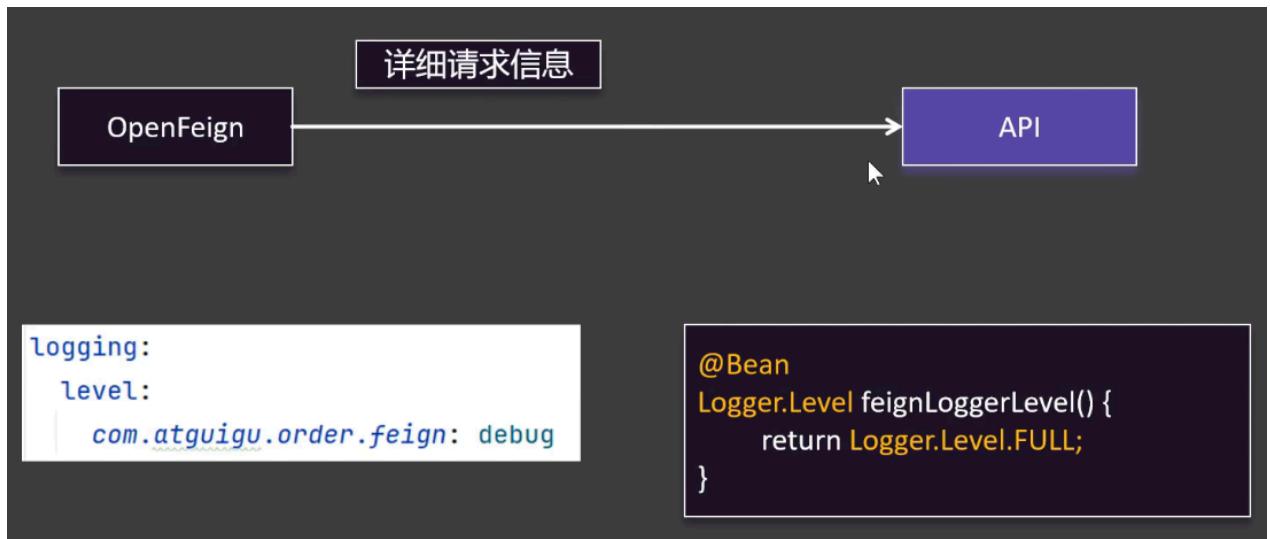
### 面试题：客户端负载均衡与服务端负载均衡区别

- 解答：客户端是发起调用方选择一个地址调用微服务，服务端是接收调用方选择一个微服务节点调用返回（客户端负载均衡需要根据注册中心知道微服务注册的所有节点再去调用，而服务端负载均衡是不需要让客户端知道注册节点的）



### 4.请求日志:

- 开启配置,添加组件:



application.yml

```

3   spring:
9     cloud:
10    nacos:
18      #可以设置不检查有没有 import 数据
19
20    logging:
21      level:
22        com.redtu.order.feign: debug
23
  
```

application.yml

```

6   import org.springframework.context.annotation.
7   import org.springframework.web.client.RestTe
8
9   @Configuration
10  public class OrderServiceConfig {
11
12  @Bean
13    Logger.Level feignLoggerLevel() {
14      return Logger.Level.FULL;
15    }
16
  
```

```

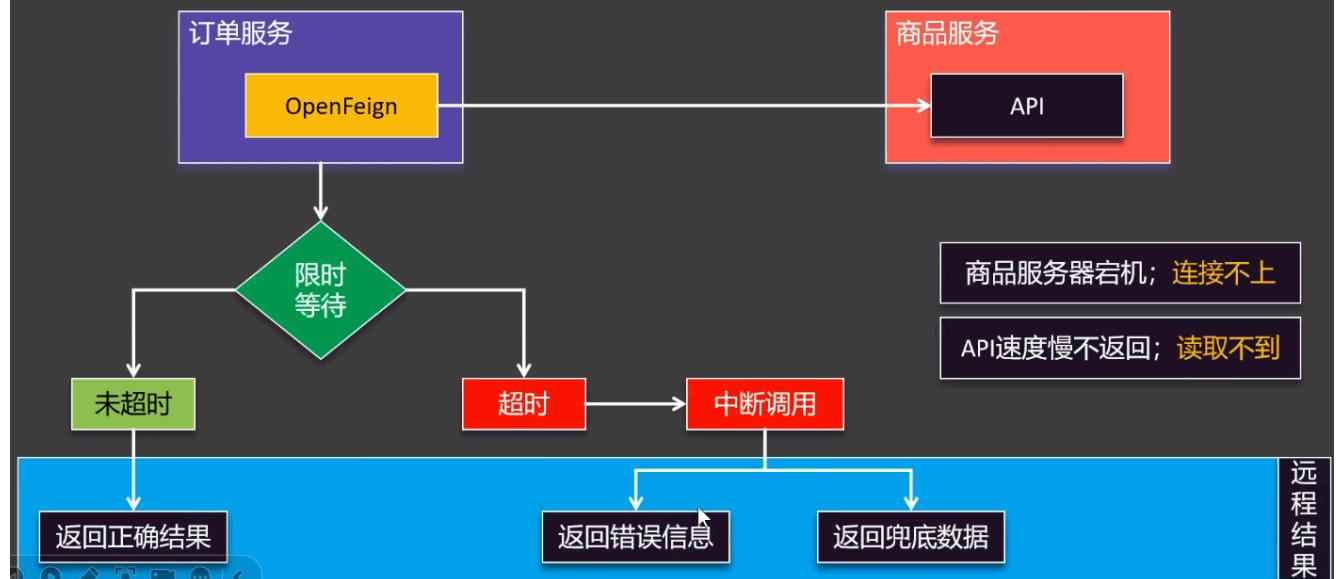
    /ductFeignClient
    : [ProductFeignClient#getProductById]
    : [ProductFeignClient#getId] <---- END HTTP (63-byte body)
    : [ProductFeignClient#getId] ---> GET http://service-product/product/1314 HTTP/1.1
    : [ProductFeignClient#getId] ---> END HTTP (0-byte body)
    : [ProductFeignClient#getId] <---- HTTP/1.1 200 (2ms)
    : [ProductFeignClient#getId] connection: keep-alive
    : [ProductFeignClient#getId] content-type: application/json
    : [ProductFeignClient#getId] date: Fri, 18 Jul 2025 18:58:15 GMT
    : [ProductFeignClient#getId] keep-alive: timeout=60
    : [ProductFeignClient#getId] transfer-encoding: chunked
    : [ProductFeignClient#getId]
    : [ProductFeignClient#getId] {"id":1314,"price":99,"productName":"巧克力-1314","num":520}
    : [ProductFeignClient#getId] <---- END HTTP (63-byte body)

```

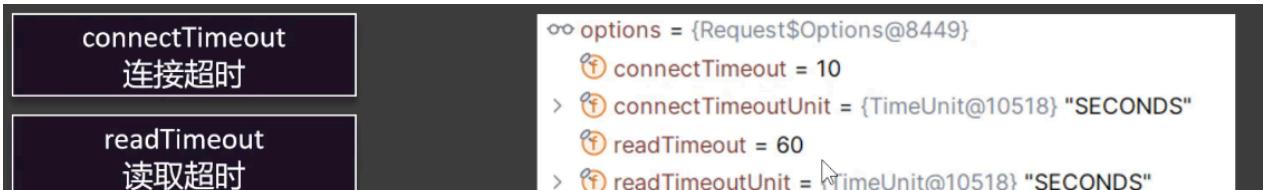
## 5. 超时控制与重试机制:

### 进阶用法 - 超时控制

尚硅谷 



- 默认配置: 连接超时10秒, 读取60秒



- 手动在application.yml中配置(可配置默认配置和对特定微服务的配置):

```
spring:
  cloud:
    openfeign:
      client:
        config:
          feignName:
            url: http://remote-service.com
            connectTimeout: 5000
             readTimeout: 5000
            loggerLevel: full
            errorDecoder: com.example.SimpleErrorDecoder
            retryer: com.example.SimpleRetryer
            defaultQueryParameters:
              query: queryValue
            defaultRequestHeaders:
              header: headerValue
            requestInterceptors:
              - com.example.FooRequestInterceptor
              - com.example.BarRequestInterceptor
```

orderServiceImpl.java application.yml

```
server:
  port: 8000
spring:
  profiles:
    active: prod
    include: feign
```

File tree:

- src
  - main
    - java
      - com.atguigu.order
        - config
        - controller
          - OrderController
        - feign
          - ProductFeignClient
          - WeatherFeignClient
        - properties
        - service
      - OrderMainApplication
    - resources
      - application.properties
      - application.yml
      - application-feign.yml**

Content of application-feign.yml:

```

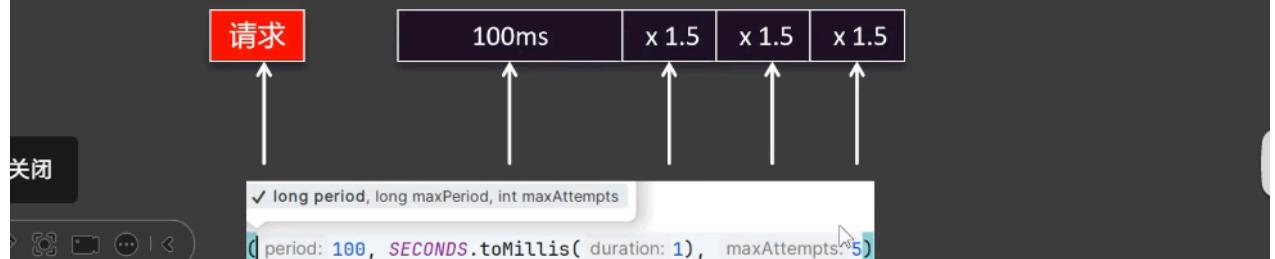
1 spring:
2   cloud:
3     openfeign:
4       client:
5         config:
6           default:
7             logger-level: full
8             connect-timeout: 1000
9             read-timeout: 2000
10      service-product:
11        logger-level: full
12        connect-timeout: 3000
13        read-timeout: 5000

```

- 重试器retryer:

- 远程调用超时失败后，还可以进行多次尝试，如果某次成功返回ok，如果多次依然失败则结束调用，返回错误

A bean of `Retryer.NEVER_RETRY` with the type `Retryer` is created by default, which will disable retrying. Notice this retrying behavior is different from the Feign default one, where it will automatically retry `IOExceptions`, treating them as transient network related exceptions, and any `RetryableException` thrown from an `ErrorDecoder`.



- 在配置类添加重试器容器即可(openFeign支持但是自己不带,所有在配置类添加重试器容器就可以了)

```
application.yml          OrderServiceConfig.java      OrderCont
7 import org.springframework.context.annotation.L
8 import org.springframework.web.client.RestTempla
9
10 @Configuration
11 public class OrderServiceConfig {
12
13     @Bean
14     Retryer retryer(){
15         return new Retryer.Default();
16     }
17 }
```

## 6.拦截器(此处演示请求拦截器):

- 需求:添加一个一次性令牌x-token



- openFeign依旧是支持的,所以编写一个拦截器添加注释将其纳入spring组件自动扫描即可

```

application.yml
7 import java.util.UUID;
8
9
10 @Component
11 public class XTokenRequestInterceptor implements RequestInterceptor {
12
13     /**
14      * 请求拦截器
15      * @param template 请求模板
16      */
17     @Override
18     public void apply(RequestTemplate requestTemplate) {
19         System.out.println("=====XTokenRequestInterceptor=====");
20         requestTemplate.header("name: " + "X-Token", UUID.randomUUID().toString());
21     }
22 }

```

DemoApplication

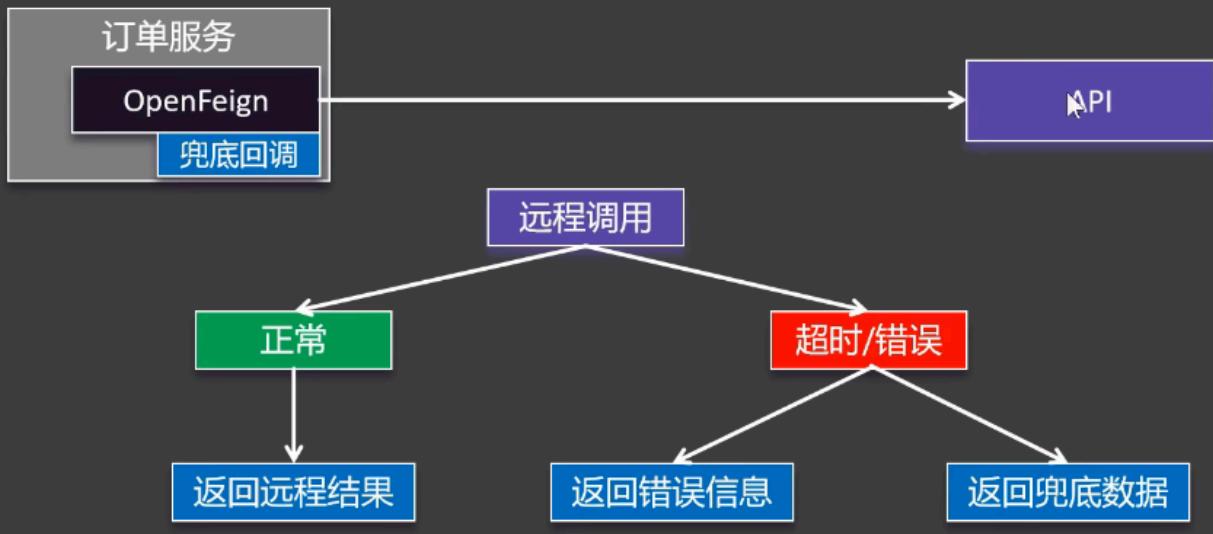
控制台

2025-07-19T04:40:00.451+08:00 [INFO] 22068 --- [service-order] [nio-8001-exec-1] o.s.web.servlet.DispatcherServlet: =====XTokenRequestInterceptor=====

## 7.兜底返回(Fallback):

### Fallback: 兜底返回

- 注意: 此功能需要整合 Sentinel 才能实现



- 编写兜底返回类(注意添加进spring组件)

```

    @Component
    public class ProductFeignClientFallback implements ProductFeignClient {
        @Override 1个用法
        public Product getProductById(Long id) {
            System.out.println("兜底兜底, 回调回调");
            Product product = new Product();
            product.setId(id);
            product.setPrice(new BigDecimal("0"));
            product.setProductName("未知商品");
            product.setNum(0);

            return product;
        }
    }

```

控制台输出：

```

2025-07-19T04:59:27.458+08:00 DEBUG 13020 --- [service-order] [nio-800]
2025-07-19T04:59:27.458+08:00 DEBUG 13020 --- [service-order] [nio-800]
2025-07-19T04:59:27.458+08:00 DEBUG 13020 --- [service-order] [nio-800]
2025-07-19T04:59:27.512+08:00 WARN 13020 --- [service-order] [nio-800]
2025-07-19T04:59:27.514+08:00 WARN 13020 --- [service-order] [nio-800]
2025-07-19T04:59:27.515+08:00 DEBUG 13020 --- [service-order] [nio-800]
兜底兜底, 回调回调

```

- 配置到@FeignClient中：

```

@FeignClient(value = "service-product", fallback = ProductFeignClientFallback.class)//远程调用的微服务名称
public interface ProductFeignClient {

    //mvc注解的两套使用逻辑(由该类的注解决定)
    //1、标注在Controller上,是接受请求
    //2、标注在FeignClient上,是发送请求
    @GetMapping("/{product}/{id}") 1个实现
    Product getProductById(@PathVariable("id") Long id);
}

```

- 兜底回调还需要sentinel熔断的配合,添加配置与依赖即可:

application.yml

```
spring:
  #可以设置不检查有没有 import
  logging:
    level:
      com.redtu.order.feign: debug
  feign:
    sentinel:
      enabled: true
---
```

pom.xml (service-order)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
<properties>
  <maven.compiler.source>17</maven.compiler.source>
  <maven.compiler.target>17</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies> ⚡ 添加启动器...
  <dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
  </dependency>
```

- 至此,完成了兜底回调方法:

localhost:8001/create?userId=520&productId=1314

优质打印 □

```
{"id":1,"totalAmount":0,"userId":0,"nickName":"hxq","address":"McGanMountain","productList":[{"id":1314,"price":0,"productName":"未知商品","num":0}]} 
```

## 8. 总结:

# OpenFeign - 总结

- 1. 熟练编写 OpenFeign 远程调用客户端
- 2. 熟练配置 OpenFeign 客户端属性
  - 连接超时
  - 读取超时
  - .....
- 3. 掌握 拦截器 用法
- 4. 掌握 Fallback 兜底返回机制 及 用法

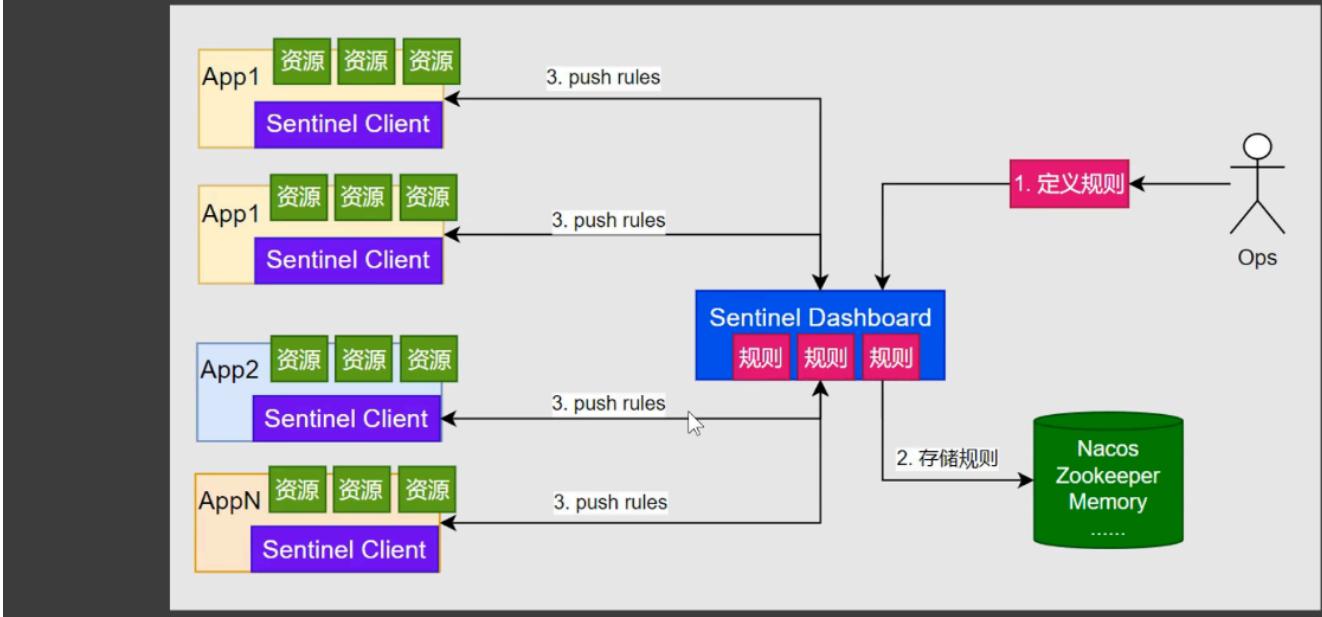
## 5. Sentinel(服务保护--限流,熔断降级):

### 1. 工作原理:

- 随着微服务的流行，服务和服务之间的稳定性变得越来越重要。Spring Cloud Alibaba Sentinel 以流量为切入点，从流量控制、流量路由、熔断降级、系统自适应过载保护、热点流量防护等多个维度保护服务的稳定性。



# 架构原理



## 资源&规则

尚硅谷

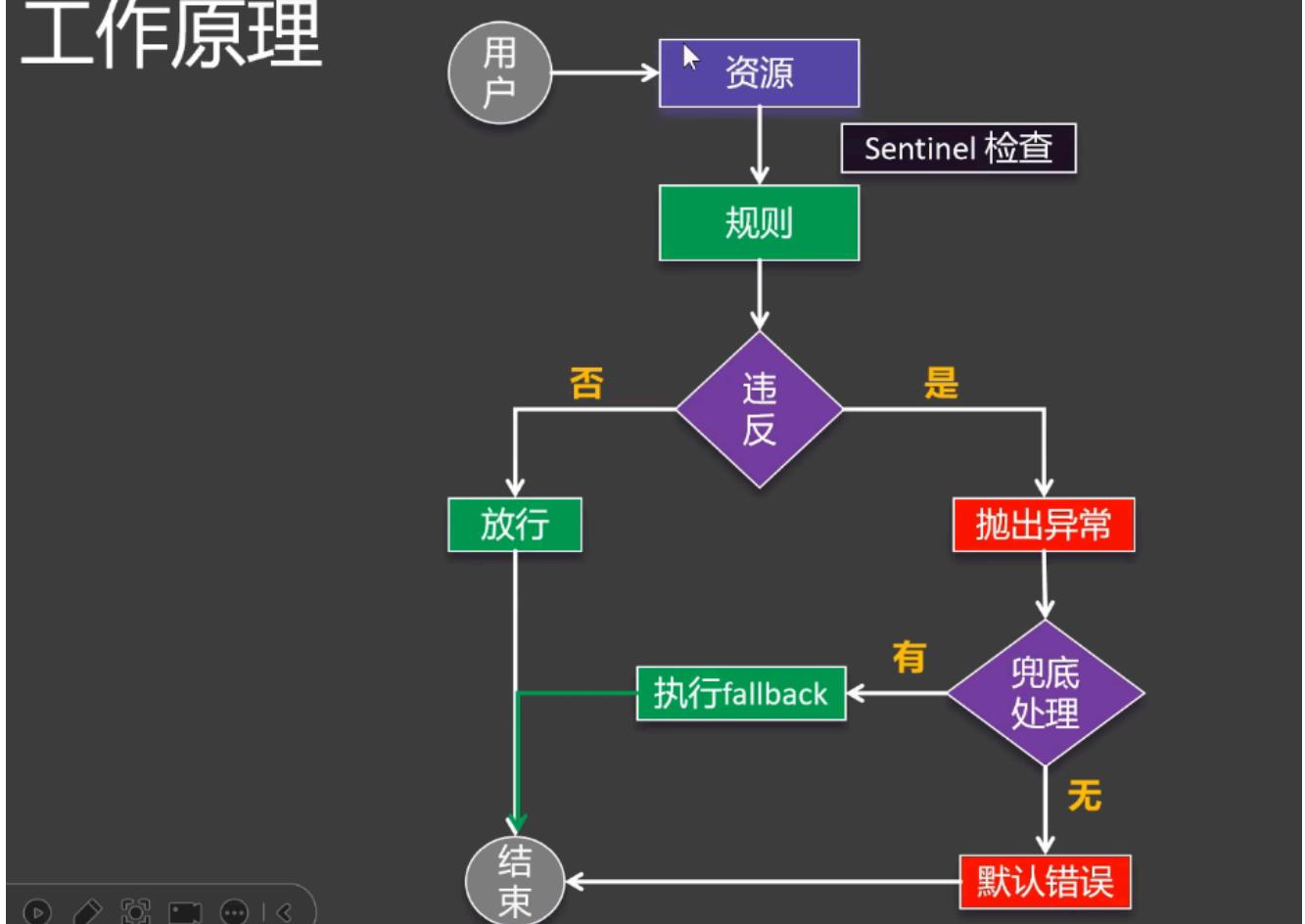
- 定义资源：

- 主流框架自动适配（Web Servlet、Dubbo、Spring Cloud、gRPC、Spring WebFlux、Reactor）；所有Web接口均为资源
- 编程式：SphU API
- 声明式：@SentinelResource

- 定义规则：

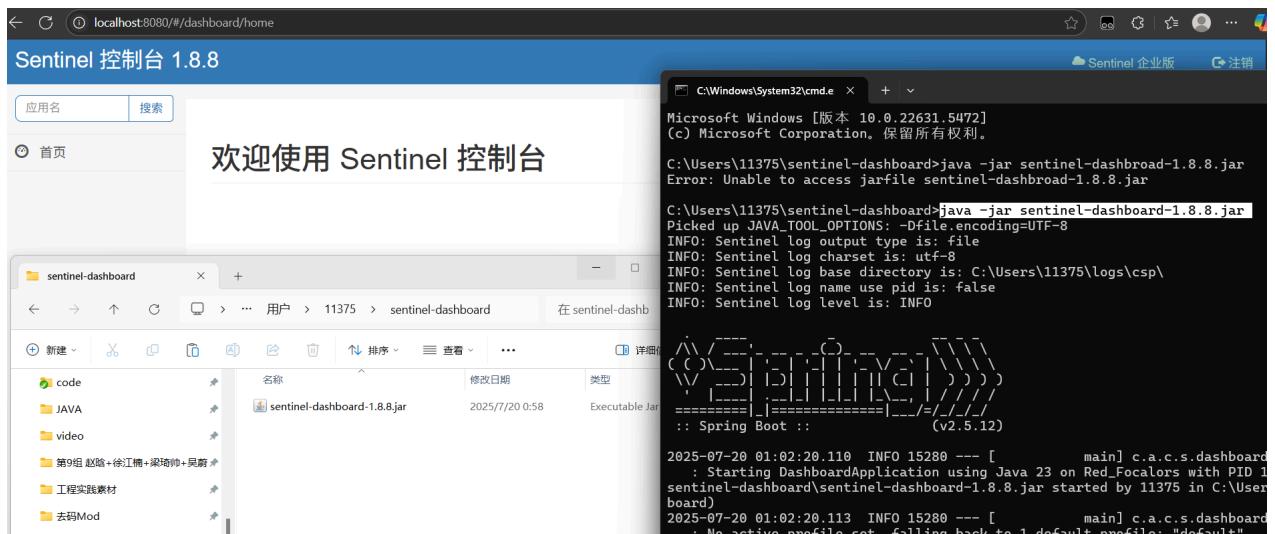
- 流量控制（FlowRule）
- 熔断降级（DegradeRule）
- 系统保护（SystemRule）
- 来源访问控制（AuthorityRule）
- 热点参数（ParamFlowRule）

# 工作原理



## 2. 整合:

- 在cmd中启动,进入网页,默认账户密码均为sentinel:



- 日常添加依赖,添加配置:

```
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
20   <properties>
21     <maven.compiler.source>17</maven.compiler.source>
22     <maven.compiler.target>17</maven.compiler.target>
23     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
24   </properties>
25
26   <dependencies> ⚡ 添加启动器...
27     <!-- 启用sentinel-->
28     <dependency>
29       <groupId>com.alibaba.cloud</groupId>
30       <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
31     </dependency>
32     <!-- 配置中心-->
```

```
3 spring:
8   name: service-order
9
10  cloud:
11    nacos:
12      server-addr: 127.0.0.1:8848
13      #原导入nacos服务ip-port
14      config:
15        namespace: ${spring.config.activate:dev}
16        #动态取值与默认值: ${}
17        import-check:
18          enabled: false
19          #可以设置不检查有没有 没有import数据集 的情况(一般情况下回报错)
20          sentinel:
21            transport:
22              dashboard: localhost:8080
23              eager: true
```

- 日常添加注解(标注其为"资源",如此处标注其名为"createOrder")

ServiceConfig.java      © OrderController.java      © OrderServiceImpl.java ×

```

public class OrderServiceImpl implements OrderService {

    @Autowired
    ProductFeignClient productFeignClient;

    @SentinelResource(value = "createOrder") 1个用法
    @Override
    public Order createOrder(Long productId, Long userId) {
        //Product product = getProductFromRemoteWithLoadBalanceAnnotation();
        Product product = productFeignClient.getProductById(productId);
    }
}

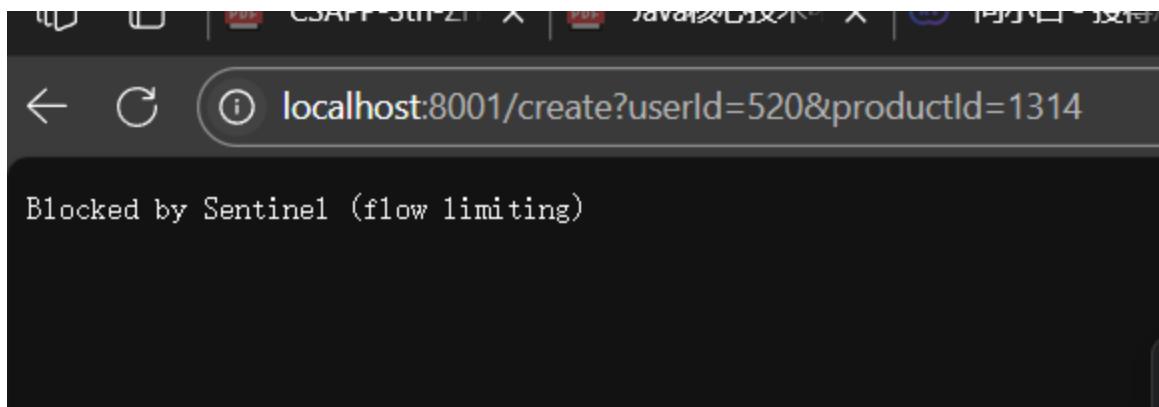
```

资源名	通过QPS	拒绝QPS	并发数	平均RT	分钟通过	分钟拒绝	操作
sentinel_default_context	0	0	0	0	0	0	<button>+ 流控</button> <button>+ 热断</button> <button>+ 热点</button> <button>+ 授权</button>
sentinel_spring_web_context	0	0	0	0	5	0	<button>+ 流控</button> <button>+ 热断</button> <button>+ 热点</button> <button>+ 授权</button>
/create	0	0	0	0	5	0	<button>+ 流控</button> <button>+ 热断</button> <button>+ 热点</button> <button>+ 授权</button>
createOrder	0	0	0	0	5	0	<button>+ 流控</button> <button>+ 热断</button> <button>+ 热点</button> <button>+ 授权</button>
GEThttp://service-product/product{id}	0	0	0	0	5	0	<button>+ 流控</button> <button>+ 热断</button> <button>+ 热点</button> <button>+ 授权</button>

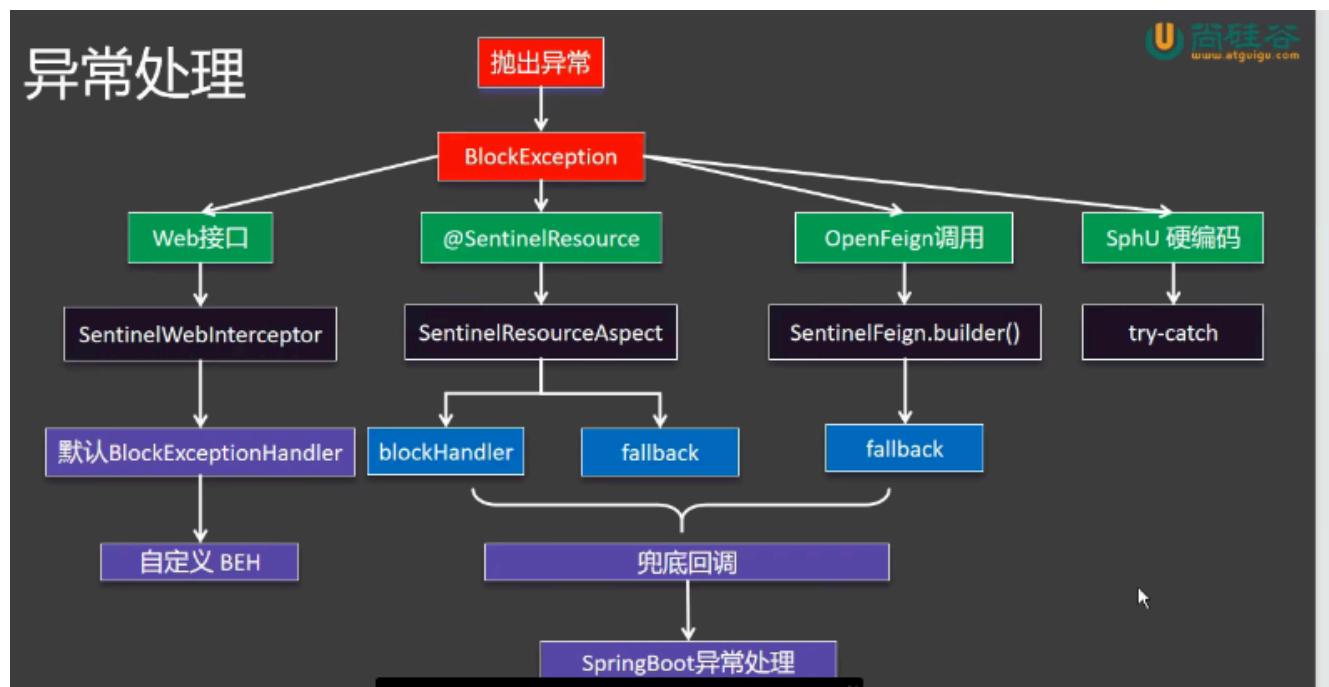
- 使用流控规则:(QPS:每秒请求数量)

编辑流控规则

资源名	/create
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数
单机阈值	1
是否集群	<input type="checkbox"/>
高级选项	
<button>保存</button> <button>取消</button>	



### 3. 异常处理:



- 对于保护的机制(如web):用调用HandleIntercepotor接口,获取资源名(请求路径),在方法执行之前进入资源保护流程,看是否违背规则(如违背,会自动返回默认的错误值,来自DefaultBlockExceptionHandler,也就是说,如果需要自定义返回内容,需自定义编写**BlockExceptionHandler**,添加进容器即可生效)
- 对于Web接口(遇到问题时返回什么东西--一般后端返回json给前端)的自定义返回:
  - 编写一个总的返回信息放在models里面:

```
MyBlockExceptionHandler.java    R.java
```

```
import lombok.Data;

@Data 12个用法
public class R {

    private Integer code;
    private String message;
    private Object data;

    @ public static R ok(String message, Object data) {
        R r = new R();
        r.setCode(200);
        r.setMessage(message);
        r.setData(data);
        return r;
    }
}
```

- 添加自定义类:1.添加@Component注解使其放进容器中;2.添加json工具,将R类型转为json的String;3.设置一下返回的类型和编码

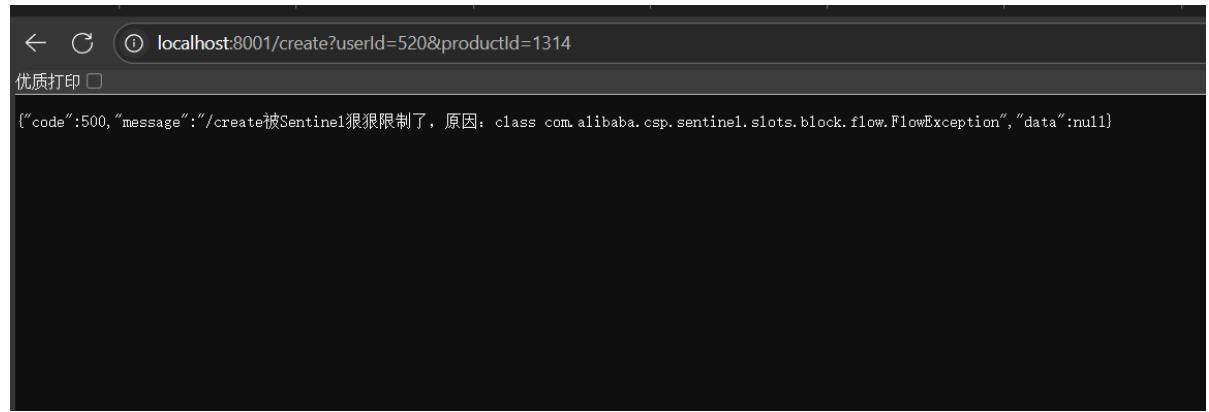
```
MyBlockExceptionHandler.java
```

```
import com.alibaba.csp.sentinel.adapter.spring.webmvc_v6x.callback.BlockExceptionHandler;
import com.alibaba.csp.sentinel.slots.block.BlockException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.redtu.common.R;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Component;
import java.io.PrintWriter;

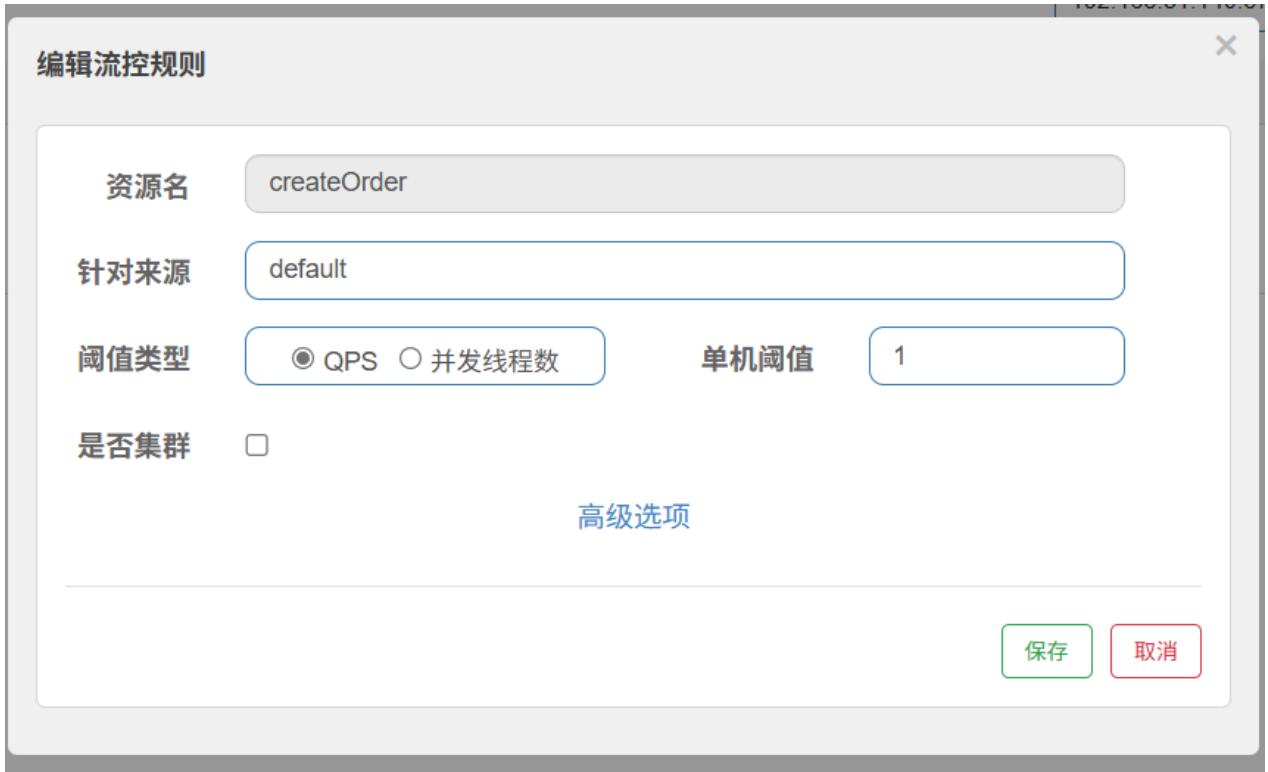
@Component
public class MyBlockExceptionHandler implements BlockExceptionHandler {
    private ObjectMapper objectMapper = new ObjectMapper(); 1个用法
    //springMVC整合JACKSON的JSON工具

    @Override
    public void handle(HttpServletRequest httpServletRequest, HttpServletResponse response,
                       String s, BlockException e) throws Exception {
        response.setContentType("application/json;charset=utf-8");
        PrintWriter writer = response.getWriter();

        R error = R.error(code: 500, message: s + "被Sentinel狠狠限制了, 原因: " + e.getClass());
        String json = objectMapper.writeValueAsString(error);
        writer.write(json);
        writer.flush();
        writer.close();
    }
}
```



- 对于@SentinelResource注解标注的资源:



- 原有: 使用了一个切面来管理所有被注解的资源, 并且可以启用两种在注解上标注的异常处理机制, blockHandler和fallback, 如果都没有, 就回到springboot自带的异常处理机制执行

```
@SentinelResource(value = "createOrder") 1个用法
@Override
public Order createOrder(Long productId, Long use
    //Product product = getProductFromRemoteWithL
    Product product = productFeignClient.getProdu
    Order order = new Order();
    order.setId(1L);
    order.setTotalAmount(product.getPrice().mult
```

# Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Jul 20 05:47:42 CST 2025

There was an unexpected error (type=Internal Server Error, status=500).

- 自定义:

```
@SentinelResource(value = "createOrder", blockHandler = "createOrderFallback") 1个用法
@Override
public Order createOrder(Long productId, Long userId) {
    //Product product = getProductFromRemoteWithLoadBalanceAnnotation(productId);
    Product product = productFeignClient.getProductById(productId);
    Order order = new Order();
    order.setId(1L);
    order.setTotalAmount(product.getPrice().multiply(new BigDecimal(product.getNum())));
    order.setUserId(0L);
    order.setNickName("hxq");
    order.setAddress("MoGanMountain");
    order.setProductList(Arrays.asList(product));

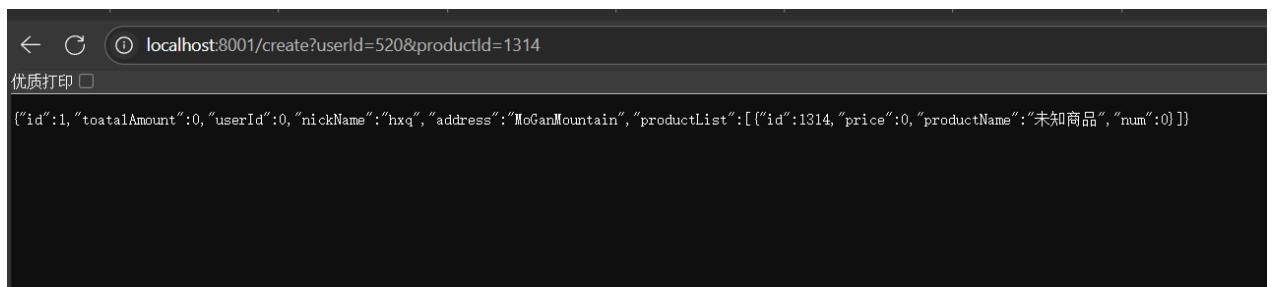
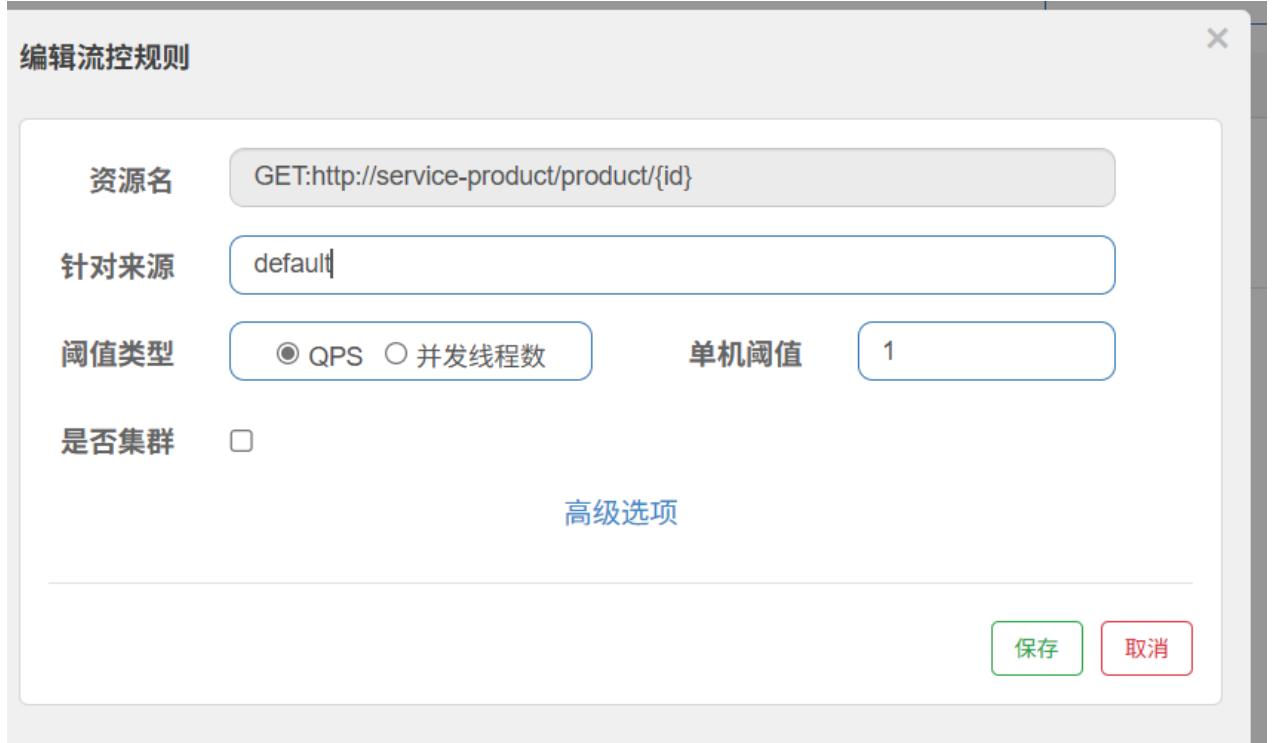
    return order;
}

//自定义兜底回调
private Order createOrderFallback(Long productId, Long userId, BlockException e) { 0个用法
    Order order = new Order();
    order.setId(0L);
    order.setTotalAmount(BigDecimal.ZERO);
    order.setUserId(0L);
    order.setNickName("什么都没有");
    order.setAddress("异常信息"+e.getClass().getName());

    return order;
}
```

```
{"id":0,"totalAmount":0,"userId":0,"nickName":"什么都没有","address":"异常信息com.alibaba.csp.sentinel.slots.block.flow.FlowException","productList":null}
```

- 对于openFeign远程调用:优先使用openFeign自己的兜底回调fallback,否则会回到springboot的错误处理(一般会设计一个全局异常处理器)



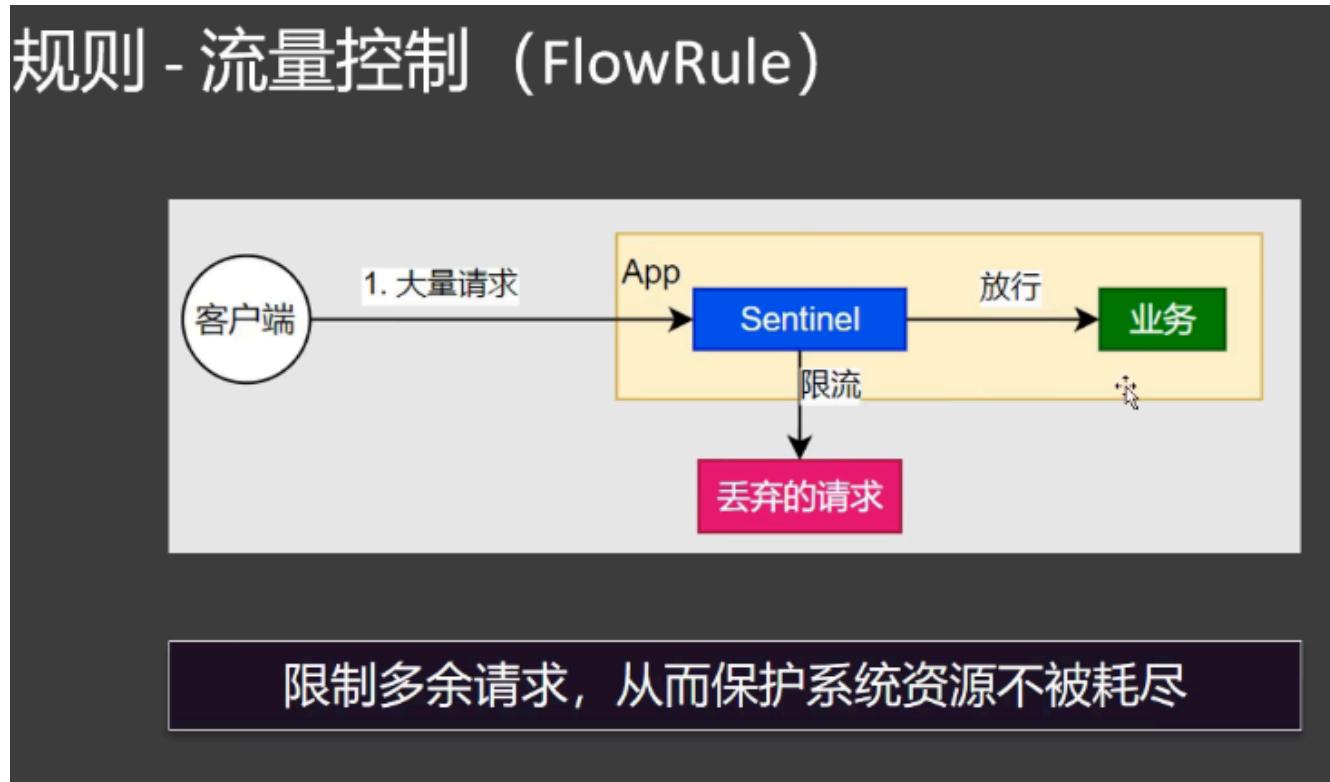
- 示例全局异常处理器:

```

1 package com.redtu.order.exception;
2
3 import org.springframework.web.bind.annotation.ExceptionHandler;
4 import org.springframework.web.bind.annotation.RestControllerAdvice;
5
6 //全局异常处理器
7 //ResponseBody 返回json数据
8 @RestControllerAdvice //为以上俩注解的合体
9 public class GlobalExceptionHandler {
10
11     //    @ExceptionHandler(Throwable.class)
12     //    public String error(Throwable e) {
13     //        return e.getMessage();
14     //    }
15 }

```

#### 4. 流控规则:



- 阈值类型: QPS: 每秒处理请求数，并行需要考虑线程池
- 集群模式下: 单机均摊表示每个节点最多放行均摊阈值数量，而总体阈值表示所有节点之和最多为xxx

新增流控规则

资源名	sentinel_default_context		
针对来源	default		
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数	均摊阈值	1
是否集群	<input checked="" type="checkbox"/>	集群阈值模式	<input checked="" type="radio"/> 单机均摊 <input type="radio"/> 总体阈值
失败退化	<input type="checkbox"/> <small>i 如果 Token Server 不可用是否退化到单机限流</small>		
<small>新增并继续添加</small> <span style="background-color: green; border: 1px solid black; padding: 2px 10px;">新增</span> <span style="background-color: red; border: 1px solid black; padding: 2px 10px;">取消</span>			

- 流控模式-(流控模式只有流控效果为快速失败才支持)(直接): 直接控制某资源的访问，超出阈值直接丢弃

## 新增流控规则

资源名	sentinel_default_context
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数
	单机阈值
是否集群	<input type="checkbox"/>
流控模式	<input checked="" type="radio"/> 直接 <input type="radio"/> 关联 <input type="radio"/> 链路
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待

[关闭高级选项](#)

[新增并继续添加](#) 新增 取消

## 规则 - 流量控制 (FlowRule) - 流控模式

直接  关联  链路

调用关系包括调用方、被调用方；一个方法又可能会调用其它方法，形成一个调用链路的层次关系；有了调用链路的统计信息，我们可以衍生出多种流量控制手段。

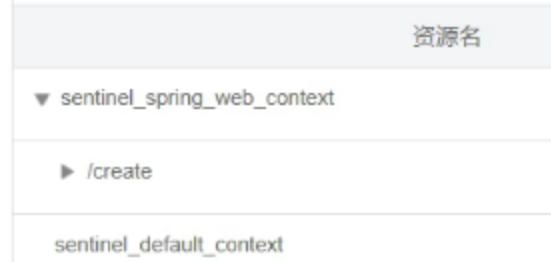
**直接策略**

**链路策略**

**关联策略**

- 流控模式(链路):针对限制某一个调用链对该资源的访问(可以让其它资源对其的调用不受限制),需要关闭web上下文一致,让链路区分
  - 关闭前:

## 簇点链路



- 关闭:

```
application.yml
spring:
  cloud:
    nacos:
      config:
        namespace: ${spring.config.namespace}
        #动态取值与默认值: ${}
        import-check:
          enabled: false
          #可以设置不检查有没有 没有import的配置
        sentinel:
          transport:
            dashboard: localhost:8080
            eager: true
        web-context-unify: false
      logging:
        level:
          com.redtu.order.feign: debug
      feign:
        sentinel:
          enabled: true
```

```
application.yml      © OrderController.java ×  © OrderServiceImpl.java      © GlobalExceptionHandler.java

public class OrderController {

    @GetMapping("/create")
    public Order createOrder(@RequestParam("productId") Long productId,
                            @RequestParam("userId") Long userId) {
        Order order = orderService.createOrder(productId,userId);
        return order;
    }

    @GetMapping("/seckill")
    public Order seckill(@RequestParam("productId") Long productId,
                        @RequestParam("userId") Long userId) {
        Order order = orderService.createOrder(productId,userId);
        order.setId(Long.MAX_VALUE);
        return order;
    }
}
```

资源名	通过QPS	拒绝QPS	并发数	平均RT	分钟通过
▼ /seckill	0	0	0	0	2
▼ /seckill	0	0	0	0	2
▼ createOrder	0	0	0	0	2
GET:http://service-product/product/{id}	0	0	0	0	2
▼ /error	0	0	0	0	0
/error	0	0	0	0	0
▼ /**	0	0	0	0	0
/**	0	0	0	0	0
sentinel_default_context	0	0	0	0	0
▼ /create	0	0	0	0	2
▼ /create	0	0	0	0	2
▼ createOrder	0	0	0	0	2
GET:http://service-product/product/{id}	0	0	0	0	2

- 设置链路流控

## 新增流控规则

资源名	createOrder
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数 <span style="float: right;">单机阈值</span> <span style="float: right;">单机阈值</span>
是否集群	<input type="checkbox"/>
流控模式	<input type="radio"/> 直接 <input type="radio"/> 关联 <input checked="" type="radio"/> 链路
入口资源	/seckill
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待
<a href="#">关闭高级选项</a>	
<a href="#">新增并继续添加</a> <span style="border: 1px solid green; padding: 2px 5px; background-color: #e0f2e0;">新增</span> <span style="border: 1px solid red; border-radius: 50%; padding: 2px 5px; color: red; background-color: white;">取消</span>	

- 流控模式(关联):比如说在一个读写数据库的场景下,将写关联读,仅在写的流量大的时候会对读进行限流,即对写关联的调用限流
  - 对readDb进行限制,关联与wirteDb(即由writeDb掌控readDb的限制)

## 新增流控规则

资源名 /readDb

针对来源 default

阈值类型  QPS  并发线程数 单机阈值 1

是否集群

流控模式  直接  关联  链路

关联资源 /writeDb

流控效果  快速失败  Warm Up  排队等待

[关闭高级选项](#)

[新增并继续添加](#) [新增](#) [取消](#)



- 流控效果(快速失败):顾名思义

## 新增流控规则

资源名	/readDb		
针对来源	default		
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数	单机阈值	1
是否集群	<input type="checkbox"/>		
流控模式	<input checked="" type="radio"/> 直接 <input type="radio"/> 关联 <input type="radio"/> 链路		
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待		

[关闭高级选项](#)

The screenshot shows a Java code editor with the file `OrderController.java` open. The code defines a controller for handling orders. It includes methods for a seckill operation and for writing and reading data from a database.

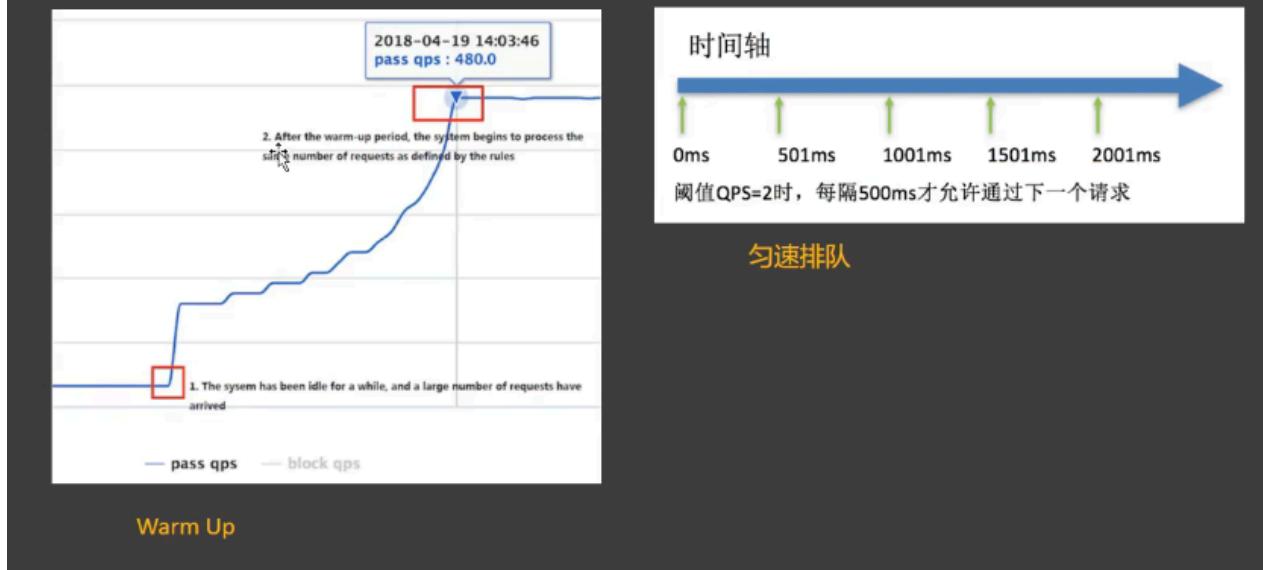
```
20 public class OrderController {  
47     public Order seckill(@RequestParam("product")  
50         Long id) {  
51         Order order = new Order();  
52         order.setId(Long.MAX_VALUE);  
53         return order;  
54     }  
55     @GetMapping("/writeDb")  
56     public String writeDb(){  
57         return "writeDb success.....";  
58     }  
59     @GetMapping("/readDb")  
60     public String readDb(){  
61         log.info("readDb success.....");  
62         return "readDb success.....";  
63     }  
64 }  
65 }
```

```
application.yml x © OrderController.java © MyBlockExceptionHandler.java x © OrderService.java  
© MyBlockExceptionHandler.java  
  
@Component  
public class MyBlockExceptionHandler implements BlockExceptionHandler {  
    private ObjectMapper objectMapper = new ObjectMapper(); 1个用法  
    //springMVC整合JACKSON的JSON工具  
  
    @Override  
    public void handle(HttpServletRequest httpServletRequest, HttpServletResponse response, String s, BlockException e) throws Exception {  
        response.setStatus(429); //too many requests 请求过量  
        response.setContentType("application/json; charset=utf-8");  
        PrintWriter writer = response.getWriter();  
  
        R error = R.error(code: 500, message: s + "被Sentinel狠狠限制了，原因：" + e.getMessage());  
        String json = objectMapper.writeValueAsString(error);  
        writer.write(json);  
        writer.flush();  
    }  
}
```



- 流控效果(warm up):让系统慢慢适应,设置冷启动周期(比如说设为三秒,第一秒放三分之一个请求,后面逐步递增直到顶峰-即设置的QPS阈值,每秒多余请求丢弃),添加了预热

## 规则 - 流量控制 (FlowRule) - 流控效果

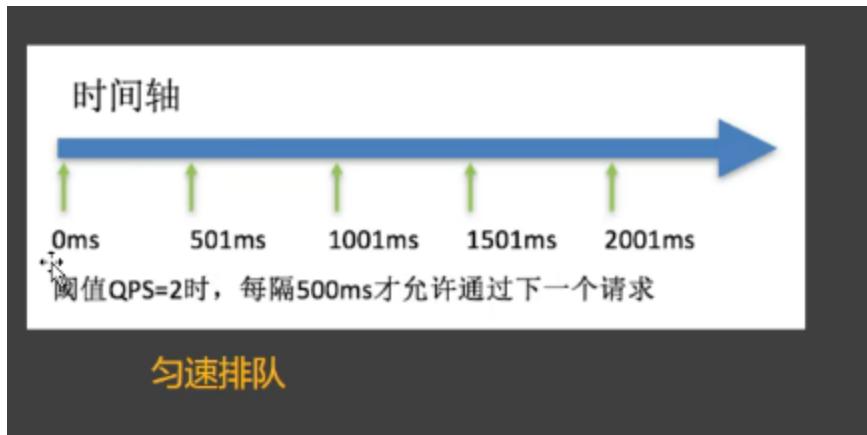


### 新增流控规则

资源名	/readDb		
针对来源	default		
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数	单机阈值	10
是否集群	<input type="checkbox"/>		
流控模式	<input checked="" type="radio"/> 直接 <input type="radio"/> 关联 <input type="radio"/> 链路		
流控效果	<input type="radio"/> 快速失败 <input checked="" type="radio"/> Warm Up <input type="radio"/> 排队等待		
预热时长	3		
<a href="#">关闭高级选项</a>			
		<a href="#">新增并继续添加</a>	<a href="#">新增</a> <a href="#">取消</a>



- 流控效果(排队等待):让多余请求进行排队,提供一个最大等待时间,超出最大等待时间即丢弃(底层是漏桶算法)



## 新增流控规则

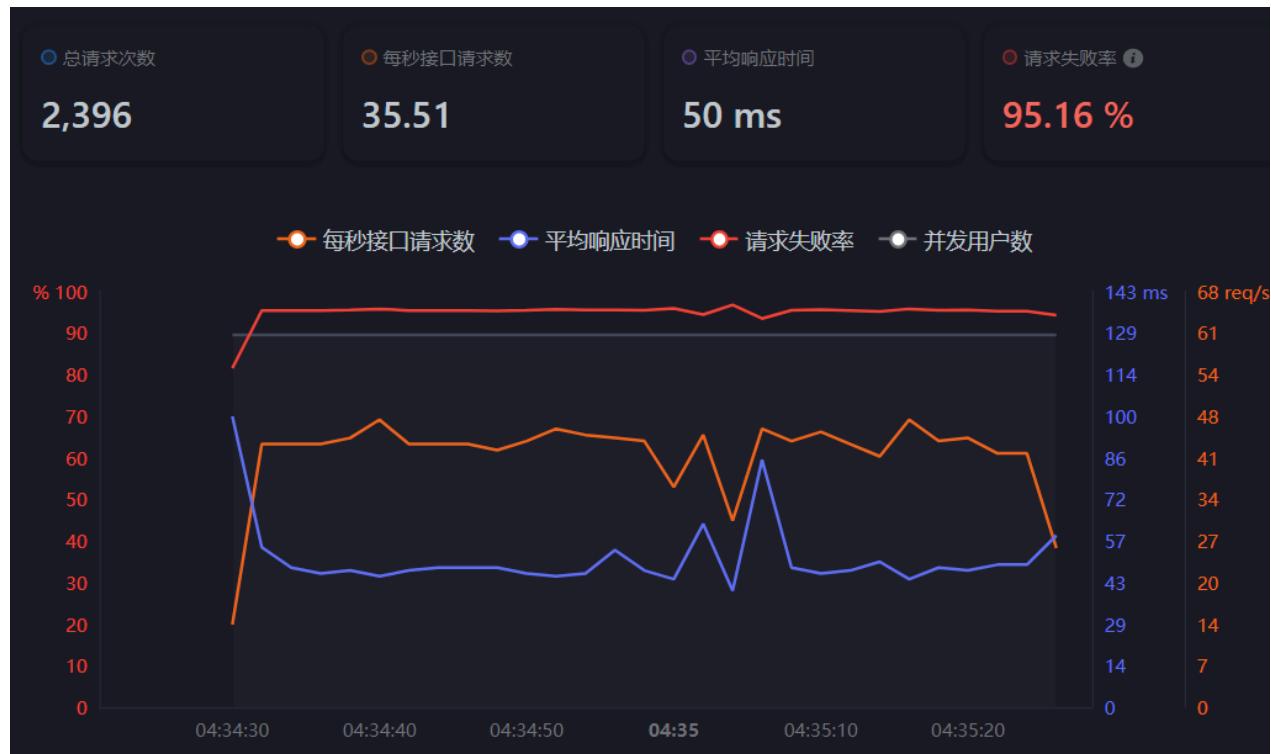
资源名	/readDb		
针对来源	default		
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数	单机阈值	2
是否集群	<input type="checkbox"/>		
流控模式	<input checked="" type="radio"/> 直接 <input type="radio"/> 关联 <input type="radio"/> 链路		
流控效果	<input type="radio"/> 快速失败 <input type="radio"/> Warm Up <input checked="" type="radio"/> 排队等待		
超时时间	1000		

[关闭高级选项](#)

[新增并继续添加](#)

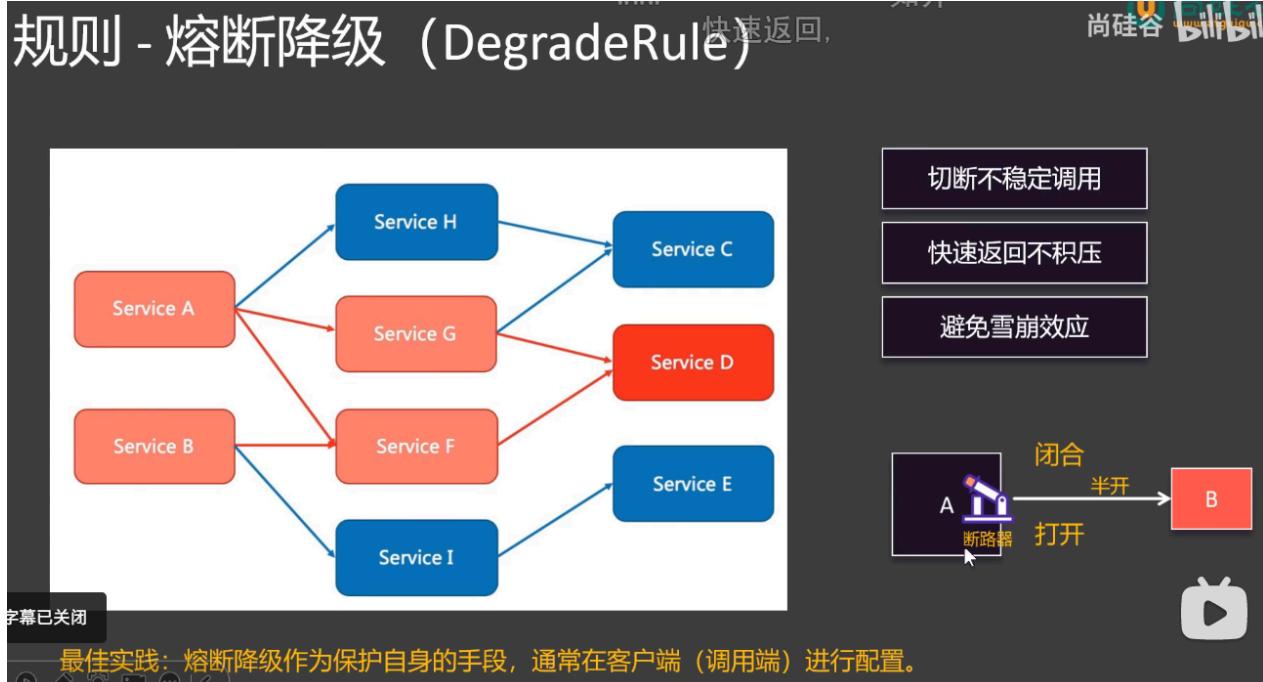
[新增](#)

[取消](#)

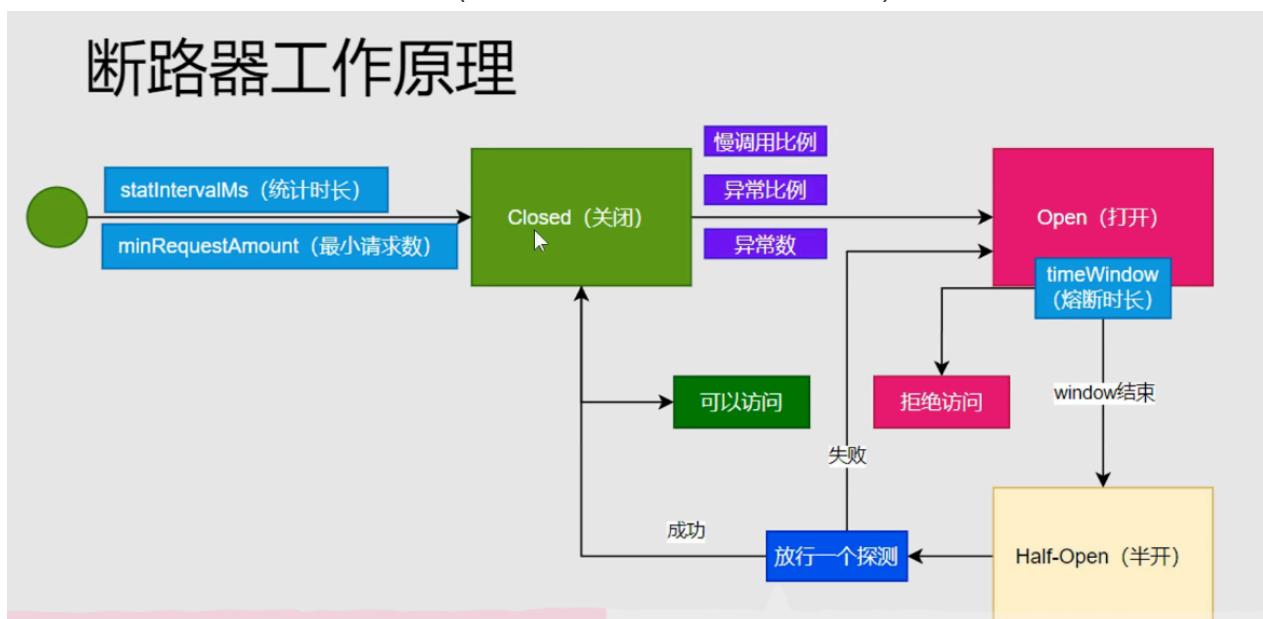


## 5. 熔断规则:

- 熔断降级:及时切断,然后快速返回,从而避免服务雪崩(即一个服务慢了引起整个服务系统都崩掉)



- 断路器工作原理:统计多少时间内调用的情况,如果达到预设的熔断策略阈值,即打开断路器,一定时间内(熔断窗口timeWindow)直接快速返回失败,后变为半开状态,测试调用情况是否成功来决定断路器是否重新关闭(试探试探这个请求好不好使了)



- 熔断策略(慢调用比例):熔断原因是让调用端一定时间内能不能调用被调用端(**远程调用的兜底返回**,所以**远程调用要记得写兜底fallbcak**),所以一般配在**远程访问的调用端**,为了方便测试给被调用端添加一个休眠时间;RT:response time响应时间

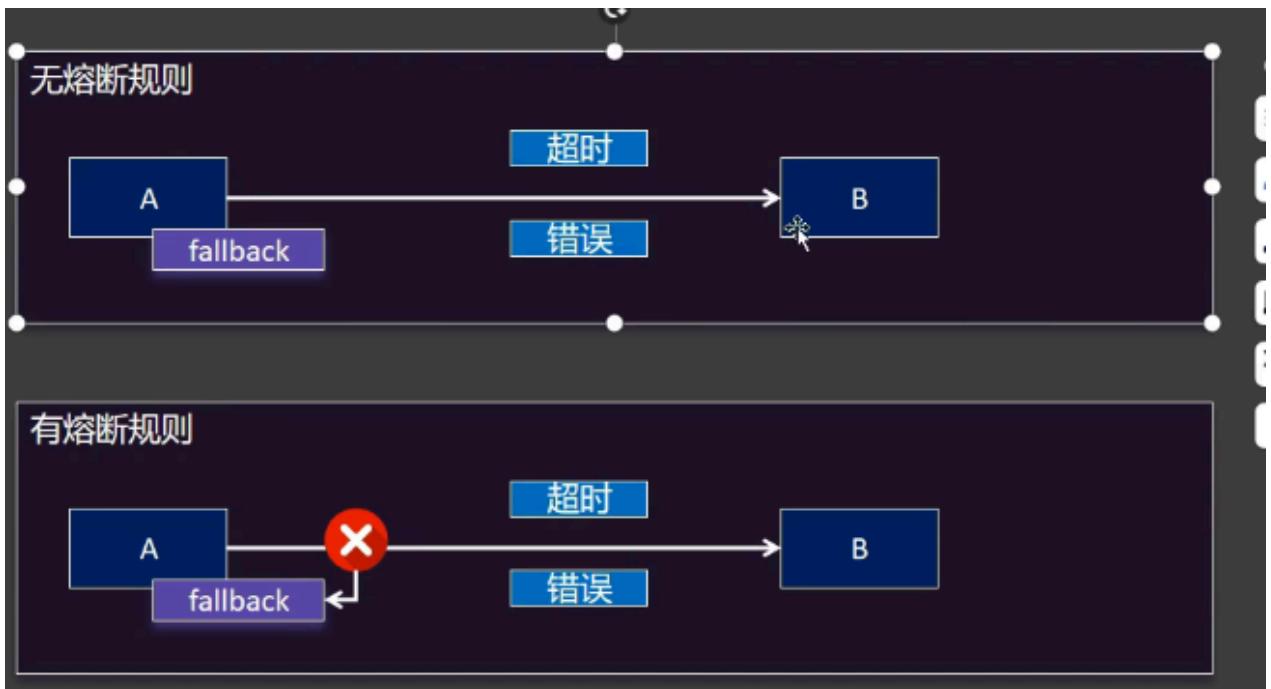
## 新增熔断规则

资源名	GET:http://service-product/product/{id}			
熔断策略	<input checked="" type="radio"/> 慢调用比例 <input type="radio"/> 异常比例 <input type="radio"/> 异常数			
最大 RT	1000	比例阈值	0.8	
熔断时长	30	s	最小请求数	5
统计时长	5000	ms		
		新增并继续添加	新增	
		取消		

```
2
3     @RestController
4     public class ProductController {
5
6         @Autowired
7         private ProductService productService;
8
9         @GetMapping("/product/{id}")
10        public Product getProduct(@PathVariable("id") Long productId) {
11            Product product = productService.getProductById(productId);
12            try {
13                TimeUnit.SECONDS.sleep( timeout: 2);
14            } catch (InterruptedException e) {
15                throw new RuntimeException(e);
16            }
17            return product;
18        }
19    }
```

```
{"id":1,"totalAmount":0,"userId":0,"nickName":"hxq","address":"MoGanMountain","productList":[{"id":1314,"price":0,"productName":"未知商品","num":0}]} 
```

- 熔断策略(异常比例):无熔断-会远程调用,发现异常,然后再返回错误,进行fallback,有熔断-熔断后直接fallback而不进行远程调用,响应更快



- 手动添加异常与配置:

```
ation.yml x © OrderController.java © ProductController.java x © MyBlockException.java
```

```
public class ProductController {
```

```
    private ProductService productService;
```

```
    @GetMapping("http://localhost:8001/product/{id}")
    public Product getProduct(@PathVariable("id") Long productId) {
        Product product = productService.getProductById(productId);
        int i = 10/0;
        // try {
        //     TimeUnit.SECONDS.sleep(2);
        // } catch (InterruptedException e) {
        //     throw new RuntimeException(e);
        // }
        return product;
    }
}
```

新增熔断规则

资源名	GET:http://service-product/product/{id}		
熔断策略	<input type="radio"/> 慢调用比例 <input checked="" type="radio"/> 异常比例 <input type="radio"/> 异常数		
比例阈值	0.8		
熔断时长	30	s	最小请求数
统计时长	5000	ms	

- 熔断策略(异常数):不管比例,仅看数量

新增熔断规则

资源名	GET:http://service-product/product/{id}		
熔断策略	<input type="radio"/> 慢调用比例 <input type="radio"/> 异常比例 <input checked="" type="radio"/> 异常数		
异常数	10	▼	
熔断时长	30	s	最小请求数
统计时长	5000	ms	

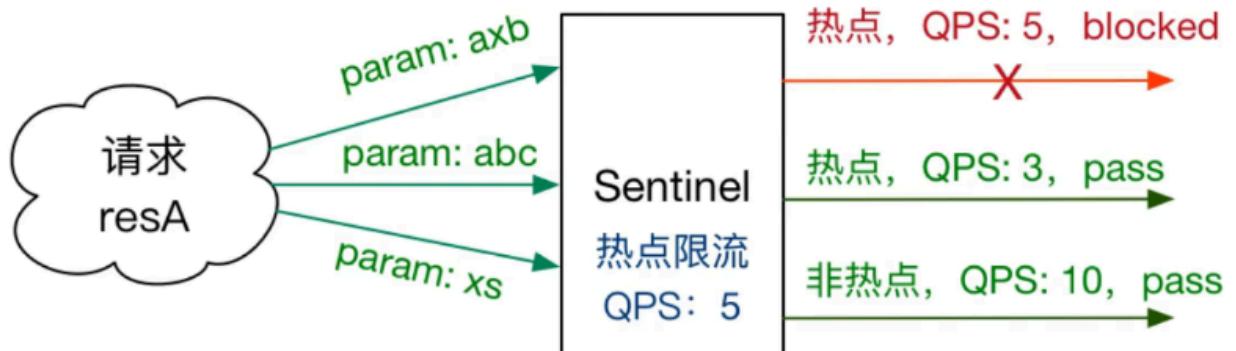
## 6.热点规则(热点参数限流):

- 流控规则:对资源访问量限制
- 热点规则:对访问时的参数进行限制,如满足条件则对流量限制

何为热点？热点即经常访问的数据。很多时候我们希望统计某个热点数据中访问频次最高的 Top K 数据，并对其进行限制。比如：

- 商品 ID 为参数，统计一段时间内最常购买的商品 ID 并进行限制
- 用户 ID 为参数，针对一段时间内频繁访问的用户 ID 进行限制

热点参数限流会统计传入参数中的热点参数，并根据配置的限流阈值与模式，对包含热点参数的资源调用进行限流。热点参数限流可以看做是一种特殊的流量控制，仅对包含热点参数的资源调用生效。



- 需求1：每个用户秒杀 QPS 不得超过 1 (秒杀下单 userId 级别)
- 需求2：6号用户是vvip，不限制QPS (例外情况)
- 需求3：666号是下架商品，不允许访问

## 通过热点参数如何完成这三个需求

对于 `@SentinelResource` 注解方式定义的资源，若注解作用的方法上有参数，Sentinel 会将它们作为参数传入 `SphU.entry(res, args)`。比如以下的方法里面 `uid` 和 `type` 会分别作为第一个和第二个参数传入 Sentinel API，从而可以用于热点规则判断：

```
@SentinelResource("myMethod")
public Result doSomething(String uid, int type) {
    // some logic here...
}
```

**注意：**目前 Sentinel 自带的 adapter 仅 Dubbo 方法埋点带了热点参数，其它适配模块（如 Web）默认不支持热点规则，可通过自定义埋点方式指定新的资源名并传入希望的参数。注意自定义埋点的资源名不要和适配模块生成的资源名重复，否则会导致重复统计。

- 用热点参数满足上述三个需求，先搭建环境：自定义一个资源及其fallback，此处框起来的名称需不同

```
application.yml × OrderController.java × ProductController.java MyBlockExceptionHandler.java OrderSe
public class OrderController {
    @GetMapping("/seckill")
    @SentinelResource(value = "seckill-order", fallback = "seckillFallback")
    public Order seckill(@RequestParam("productId") Long productId,
                         @RequestParam("userId") Long userId) {
        Order order = orderService.createOrder(productId, userId);
        order.setId(Long.MAX_VALUE);
        return order;
    }
    private Order seckillFallback(Long productId, Long userId, BlockException e){ 0个用法
        System.out.println("seckillFallback.....");
        Order order = new Order();
        order.setId(productId);
        order.setUserId(userId);
        order.setAddress("异常信息: " + e.getClass());
        return order;
    }
}
```

- 需求一:

- 此处参数索引与api中定义的顺序一致(0表示第一个,此处1表示第二个-userId),当然如果实际请求userId为null,根本就没携带该参数的话,就不参与热点参数流控



```
@GetMapping("/seckill")
@SentinelResource(value = "seckill-order", fallback = "seckillFallback")
public Order seckill(@RequestParam("productId") Long productId,
                     @RequestParam("userId") Long userId) {
    Order order = orderService.createOrder(productId, userId);
    order.setId(Long.MAX_VALUE);
    return order;
}
```

- 需求二:进入原有热点规则进行编辑,进入高级选项,添加例外项:

资源名 seckill-order

限流模式 QPS 模式

参数索引 1

单机阈值 1 秒

是否集群

参数例外项

参数类型	long			
参数值	6	限流阈值	1000000	+ 添加
参数值	参数类型	限流阈值	操作	

关闭高级选项

保存 取消

localhost:8080 显示

修改规则成功

确定

资源名

限流模式

参数索引 1

单机阈值 1

统计窗口时长 1 秒

是否集群

---

参数例外项

参数类型 long

参数值 例外项参数值 限流阈值 限流阈值 + 添加

参数值	参数类型	限流阈值	操作
6	long	1000000	<span>删除</span>

关闭高级选项

保存 取消

- 需求三:添加新规则,检查商品id参数,并进入高级选项将666号商品进行例外(不允许访问)

## 编辑热点规则

资源名	seckill-order		
限流模式	QPS 模式		
参数索引	0		
单机阈值	100000000000	统计窗口时长	1 秒
是否集群	<input type="checkbox"/>		
<b>参数例外项</b>			
参数类型	<input type="button" value="▼"/>		
参数值	例外项参数值	限流阈值	限流阈值
+ 添加			
参数值	参数类型	限流阈值	操作
666	long	0	<input type="button" value="删除"/>
<a href="#">关闭高级选项</a>			
<a href="#">保存</a> <a href="#">取消</a>			

← ⏪ ⓘ localhost:8001/seckill?userId=6&productId=666

## Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Jul 23 05:26:23 CST 2025

There was an unexpected error (type=Internal Server Error, status=500).

## 7.有关fallback和blockHandler的区别

```
@GetMapping("/seckill") @Ify
@SentinelResource(value = "seckill-order", blockHandler = "seckillFallback")
public Order seckill(@RequestParam(value = "userId", required = false) Long userId,
                      @RequestParam(value = "productId", defaultValue = "1000") Long productId)
{
    Order order = orderService.createOrder(productId, userId);
    order.setId(Long.MAX_VALUE);
    return order;
}
```

```
public Order seckillFallback(Long userId, Long productId, BlockException exception){ no usage
    System.out.println("seckillFallback....");
    Order order = new Order();
    order.setId(productId);
    order.setUserId(userId);
```

- a. 有blockHandler优先处理blockHandler
- b. blockHandler专门处理留空的异常(如null)
- c. fallback还可以处理业务异常,不过在兜底回调方法中的异常参数类型改为Throwable就可以处理了

```
@GetMapping("/seckill") @Ify
@SentinelResource(value = "seckill-order", fallback = "seckillFallback")
public Order seckill(@RequestParam(value = "userId", required = false) Long userId,
                      @RequestParam(value = "productId", defaultValue = "1000") Long productId)
{
    Order order = orderService.createOrder(productId, userId);
    order.setId(Long.MAX_VALUE);
    return order;
}
```

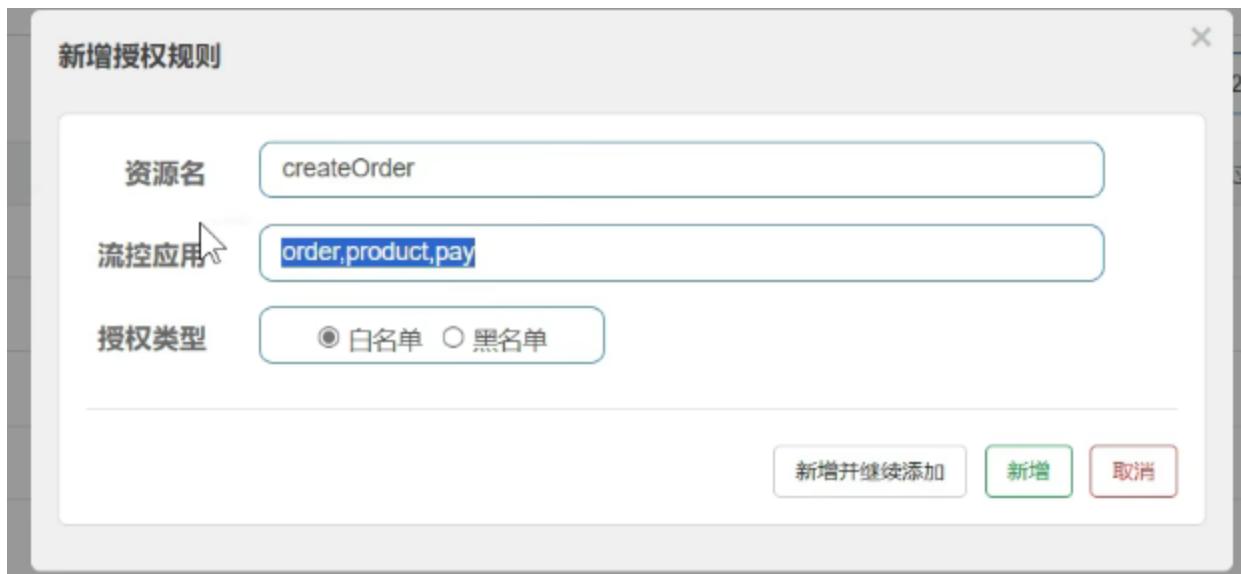
```
public Order seckillFallback(Long userId, Long productId, Throwable exception){ no usages
    System.out.println("seckillFallback....");
    Order order = new Order();
    order.setId(productId);
```

C ⓘ localhost:8000/seckill?productId=666&userId=6

```
{"id": 666,
"totalAmount": null,
"userId": 6,
"nickName": null,
"address": "异常信息: class com.alibaba.csp.sentinel.slots.block.flow.param.ParamFlowException",
"productList": null}
```

## 8. 总结:

- 授权规则(网关什么的完全可以取代):就是黑白名单



- 系统规则(精读/效果上也没啥用,尤其是有关容器应用会自带调控):对整个系统的控制

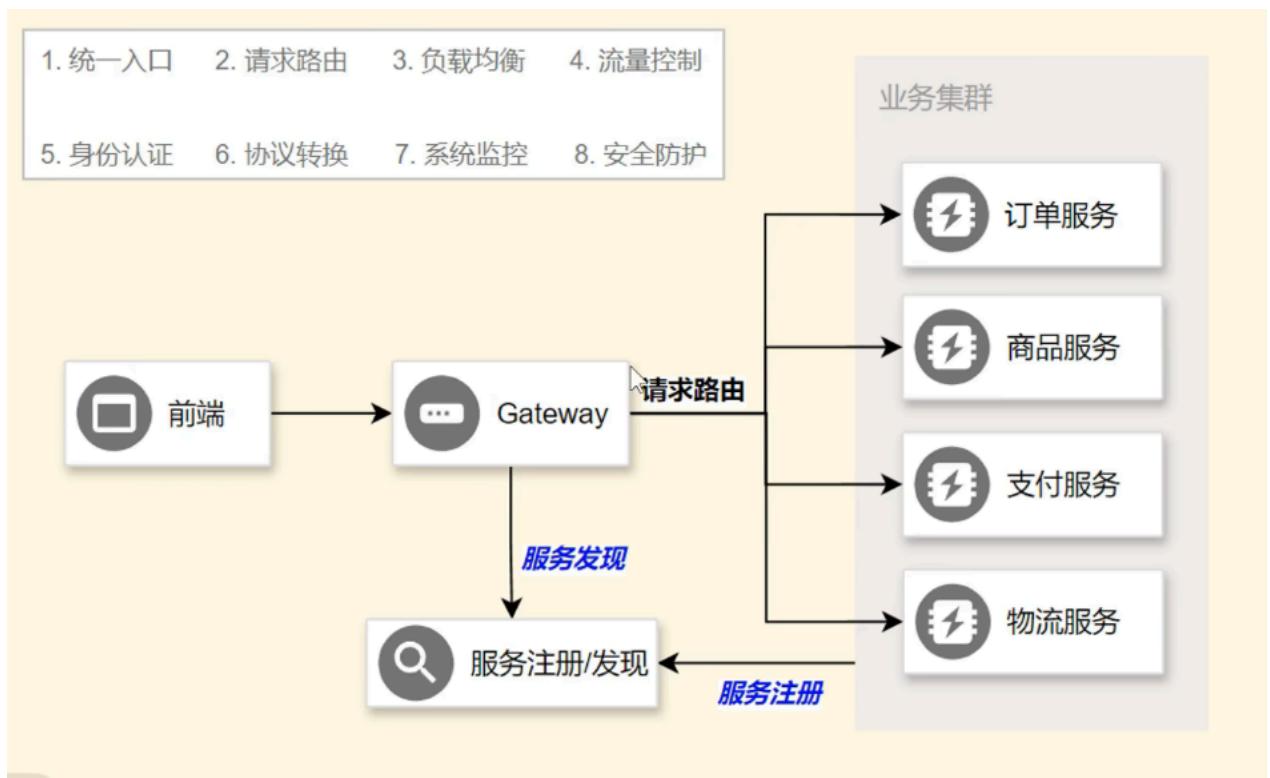
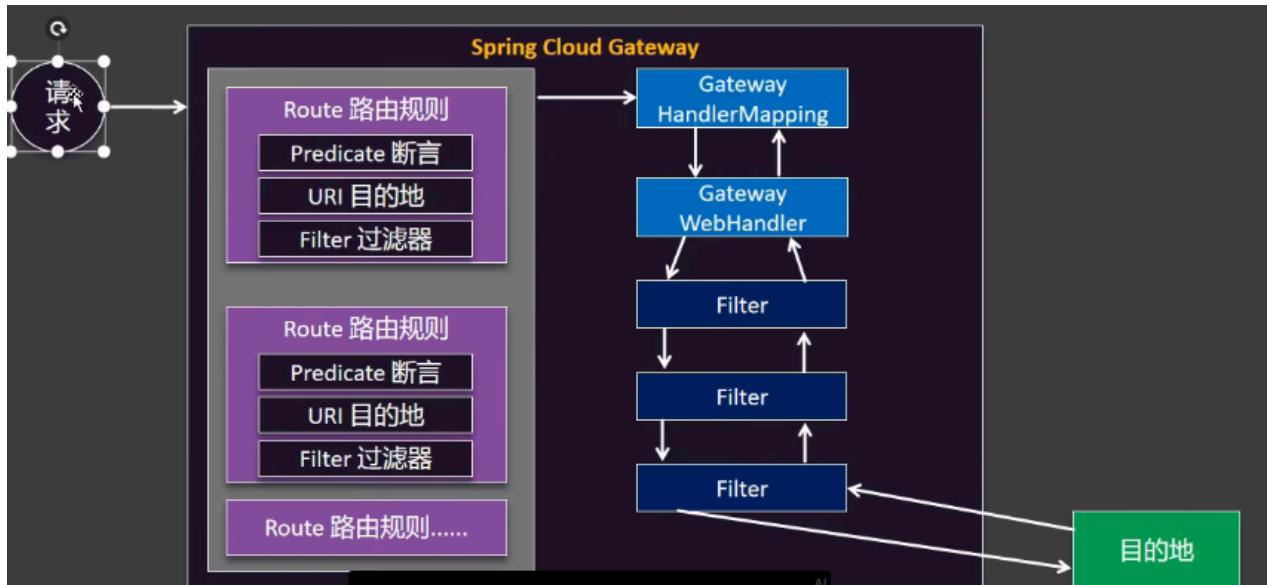


- 规则持久化:结合nacos存入nacos,连上数据库让规则持久化(应用重启不丢失)

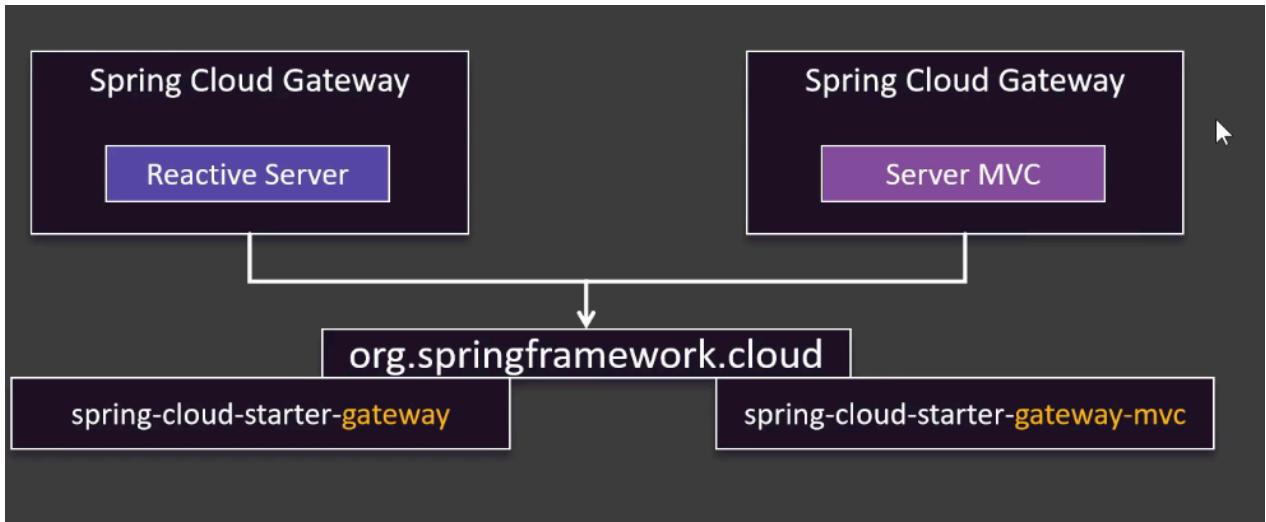
## 6. Gateway(网关):

### 1. 功能与创建:

- 工作原理:

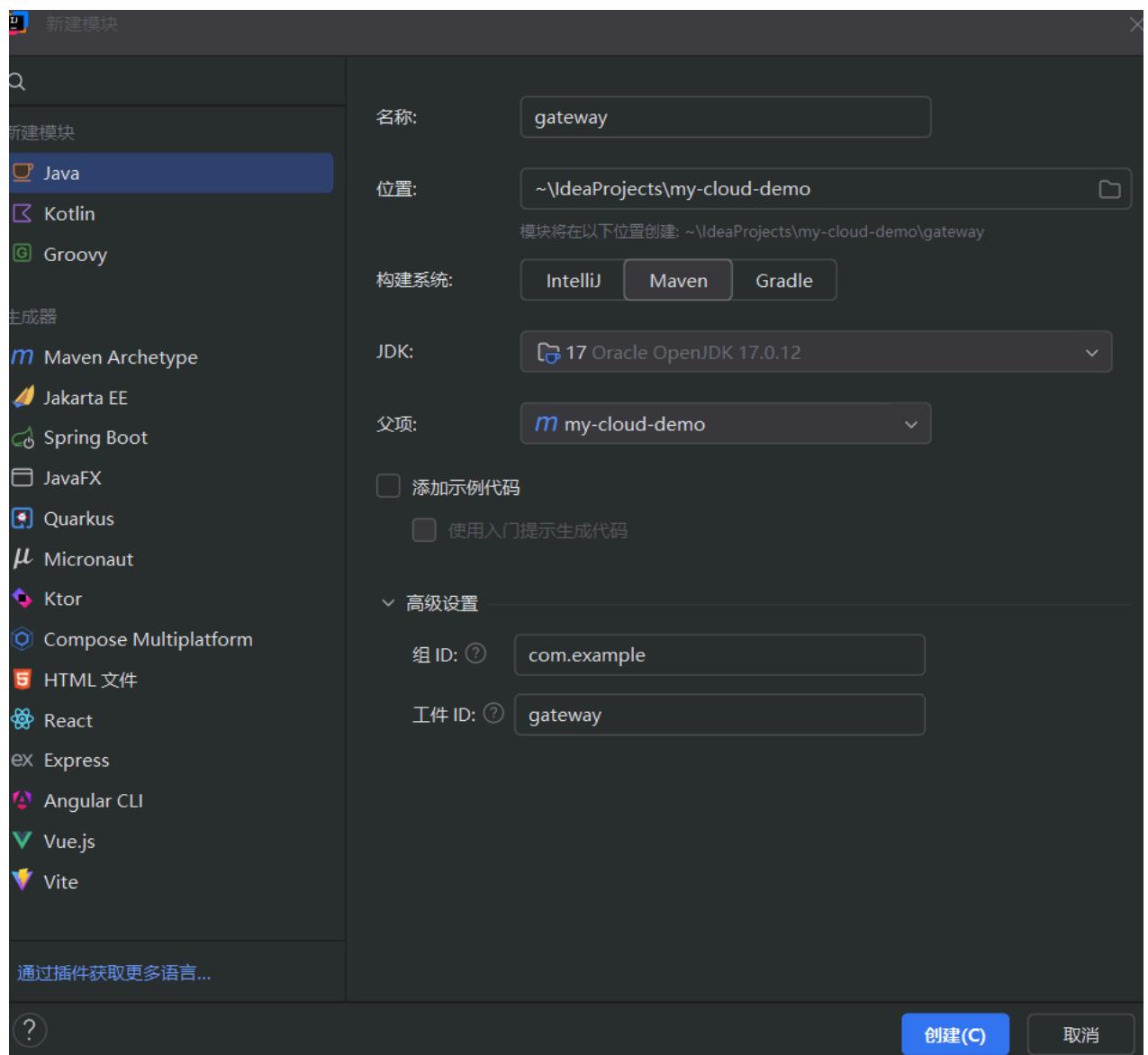


- 左边为响应式编程的网关,右边是传统网关(推荐用左边的)
- 响应式编程:类似于声明式编程,开发者通过声明数据流的依赖关系(类似于触发器trigger),**声明式编程更关注静态,响应式更关注动态**
- 导入starter即可



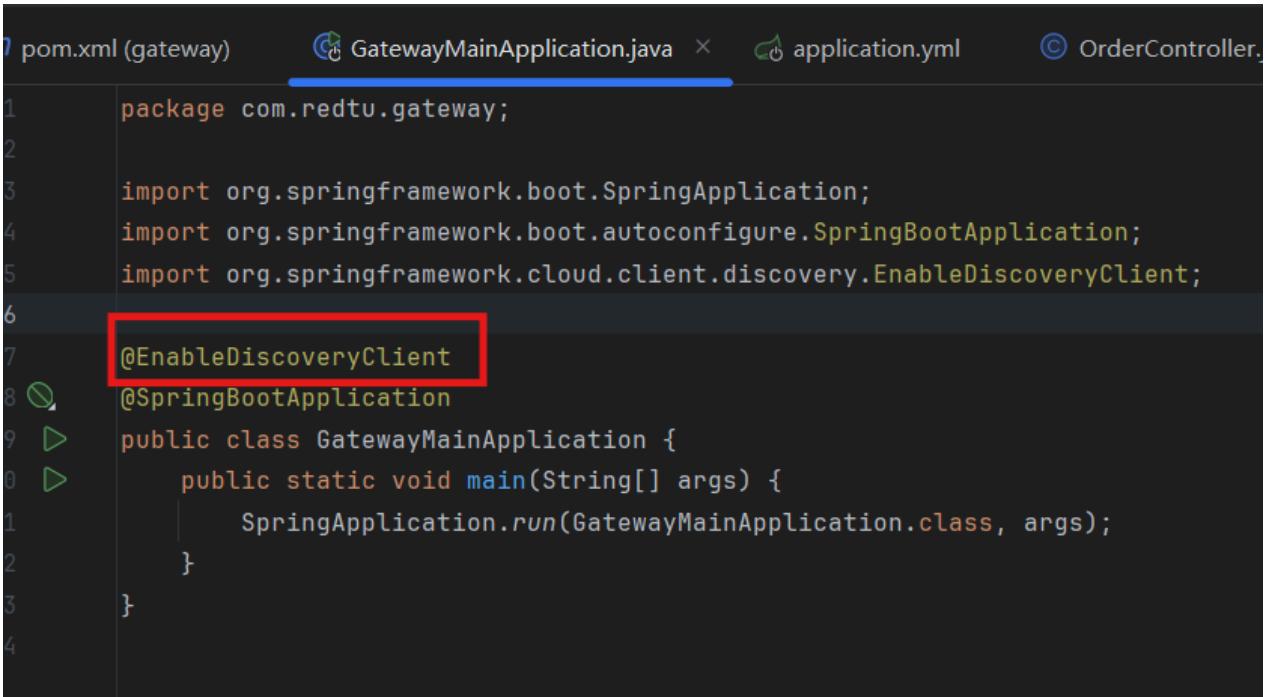
## 需求

- 1. 客户端发送 `/api/order/**` 转到 `service-order`
  - 2. 客户端发送 `/api/product/**` 转到 `service-product`
  - 3. 以上转发有负载均衡效果
- 创建网关: 网关不属于业务, 所以创建在架构下面, 网关对服务进行调度(**自然需要知道服务在哪, 所以需要引入注册中心, 为了实现网关的负载均衡调用, 还需要引入负载均衡依赖**)
  - 然后添加依赖->创建主函数(当然要使用服务发现功能需要添加`@EnableDiscoveryClient`)->编写配置

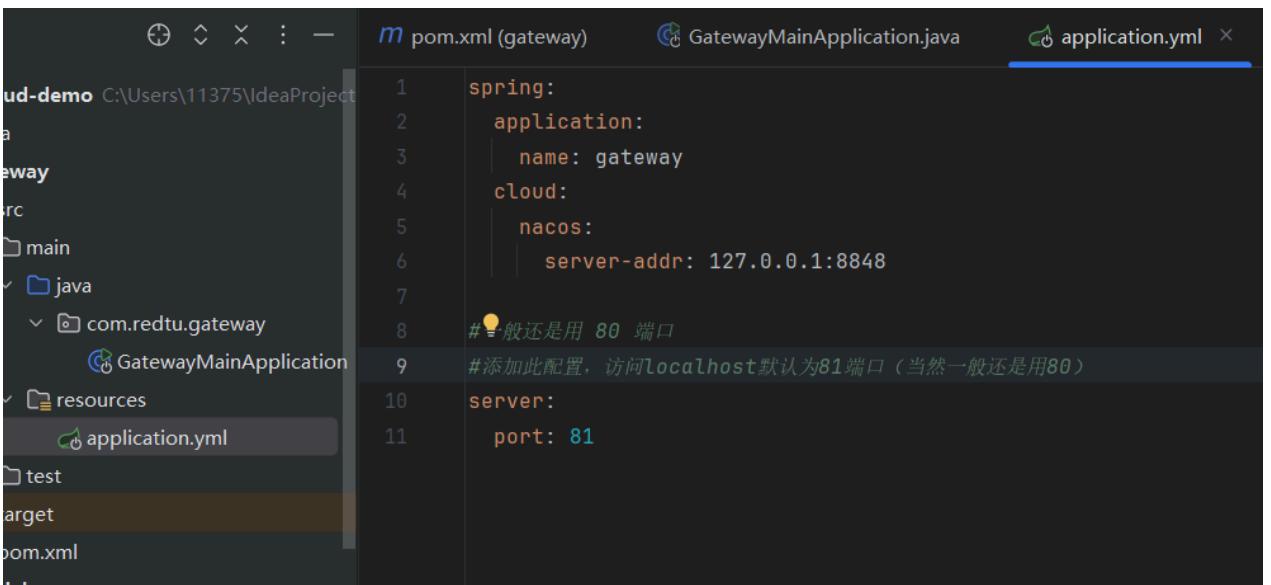


```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>gateway</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>Gateway</name>
    <description>A Spring Cloud Gateway application</description>
    <parent>
        <groupId>com.redtu.gateway</groupId>
        <artifactId>GatewayMainApplication</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>../GatewayMainApplication/pom.xml</relativePath>
    </parent>
    <dependencies>
        <!-- 注册中心 -->
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        </dependency>
        <!-- 负载均衡 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-loadbalancer</artifactId>
        </dependency>
        <!-- 网关 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId>
        </dependency>
    </dependencies>

```



```
1 package com.redtu.gateway;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
6
7 @EnableDiscoveryClient
8 @SpringBootApplication
9 public class GatewayMainApplication {
10     public static void main(String[] args) {
11         SpringApplication.run(GatewayMainApplication.class, args);
12     }
13 }
```

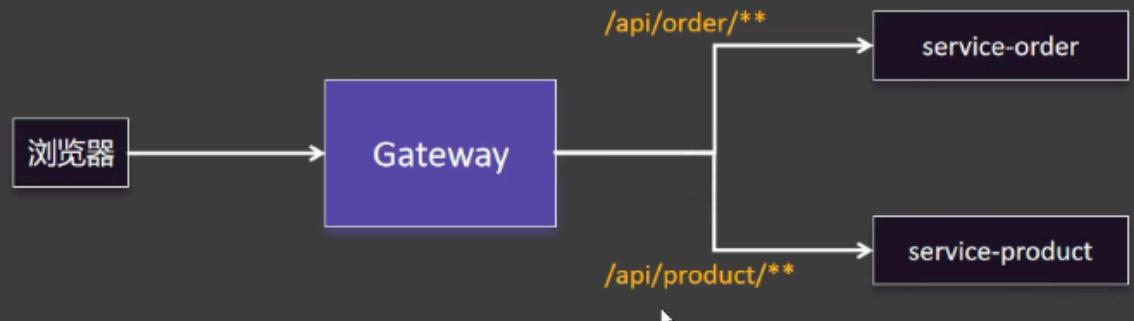


```
spring:
  application:
    name: gateway
  cloud:
    nacos:
      server-addr: 127.0.0.1:8848
#一般还是用 80 端口
#添加此配置，访问localhost默认为81端口（当然一般还是用80）
server:
  port: 81
```

## 2. 路由:

### 需求

- 1. 客户端发送 `/api/order/**` 转到 `service-order`
- 2. 客户端发送 `/api/product/**` 转到 `service-product`
- 3. 以上转发有负载均衡效果



- 路由规则:配置文件配置路由规则,或者编码编写路由规则
- 用配置文件配置路由规则:(可添加order确定各个规则的优先级,否则从上到下依次判断)

```
spring:
  profiles:
    include: route
    // ...
    name: gateway
  cloud:
    nacos:
      server-addr: 127.0.0.1:8848
#一般还是用 80 端口
#添加此配置, 访问localhost默认为81端口(当然一般还是用80)
server:
  port: 81
```

m pom.xml (gateway) application.yml application-route.yml ©

```
1 spring:
2   cloud:
3     gateway:
4       routes:
5         - id: order-route
6           uri: lb://service-order #lb:load-balance, 负载均衡
7           predicates: #断言, 遵守规则即转发到上述uri
8             - Path=/api/order/** #设置请求来源的路径
9
10        - id: product-route
11          uri: lb://service-product
12          predicates:
13            - Path=/api/product/**
```

© RouteDefinition.java × application.yml application-route.yml © OrderContr

```
36
37
38
39
40 作者: Spencer Gibb
41 @Validated
42 public class RouteDefinition {
43
44     private String id;
45
46     @NotEmpty
47     @Valid
48     private List<PredicateDefinition> predicates = new ArrayList<>();
49
50     @Valid
51     private List<FilterDefinition> filters = new ArrayList<>();
52
53     @NotNull
54     private URI uri;
55
56     private Map<String, Object> metadata = new HashMap<>();
57
58     private int order = 0;
```

- 补充各访问的前缀路径(除了controller,远程调用也需要单独添加一份)

```

1 package com.redtu.product.controller;
2
3
4 > import ...
5
6
7
8
9
10
11
12
13
14 @RequestMapping("/api/product")
15 @RestController

```

```

pom.xml (gateway) © ProductController.java © OrderController.java × ⓘ
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping("/api/order")
@Slf4j
//OpenScope

```

```

m pom.xml (gateway) © ProductController.java © OrderController.java ⓘ ProductFeignClient.java ×
4 import com.redtu.product.Product;
5 import org.springframework.cloud.openfeign.FeignClient;
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.PathVariable;
8 import org.springframework.web.bind.annotation.RequestHeader;
9 import org.springframework.web.bind.annotation.RequestMapping;
10
11 @RequestMapping("/api/product") 4个用法 1个实现
12 @FeignClient(value = "service-product", fallback = ProductFeignClientFallback.class)//远程
13 ⓘ public interface ProductFeignClient {
14
15     //mvc注解的两套使用逻辑(由该类的注解决定)
16     //1、标注在Controller上,是接受请求
17     //2、标注在FeignClient上,是发送请求
18     @GetMapping("/product/{id}") 1个实现
19     Product getProductById(@PathVariable("id") Long id );
20 }

```

### 3. 断言:

- 写法:(原写法是短写法,下面是全写法,了解即可)

The screenshot shows a code editor with two tabs open: 'PredicateDefinition.java' and 'application-route.yml'. The 'PredicateDefinition.java' tab is active, displaying Java code for a class named 'PredicateDefinition' with annotations like @Validated and @NotNull, and a private field 'name'. The 'application-route.yml' tab is also visible.

```
ductController.java      application-route.yml      © PredicateDefinition.java × ©
作者: Spencer Gibb
@Validated
public class PredicateDefinition {
    @NotNull
    private String name;
    private Map<String, String> args = new LinkedHashMap<>();
}
```

The screenshot shows a code editor with three tabs open: 'productController.java', 'application-route.yml', and 'RoutePredicateFactory'. The 'application-route.yml' tab is active, displaying YAML configuration for a gateway route named 'order-route'. It specifies an 'uri' of 'lb://service-order #lb:load-balance' and a 'predicates' section with a 'name' of 'Path'. The 'args' section includes a 'patterns' of '/api/order/\*\*' and a 'matchTrailingSlash' setting of 'true'.

```
productController.java      application-route.yml ×      RoutePredicateFactory
spring:
  cloud:
    gateway:
      routes:
        - id: order-route
          uri: lb://service-order #lb:load-balance, 负载均衡
          predicates: #断言, 遵守规则即转发到上述uri
            - name: Path    #设置请求来源的路径
          args:
            patterns: /api/order/**
            matchTrailingSlash: true
```

- 断言全种类(在断言工厂查看,除去RoutePredicateFactory剩下的部分即为其断言种类,ctrl+H查看):

The screenshot shows a Java code editor with the file `RoutePredicateFactory.java` open. The code defines a class `RoutePredicateFactory<C>` that extends `ShortcutConfigurableRoutePredicateFactory<C>`. It includes methods like `apply(Consumer<ServerWebExchange> consumer)` and `newConfig()`. To the right of the code editor is a tree view titled "层次结构" (Hierarchical Structure) showing the inheritance path of `RoutePredicateFactory`, starting from `AbstractRoutePredicateFactory` and listing various concrete implementations.

```

RoutePredicateFactory<C> extends ShortcutConfigurableRoutePredicateFactory<C>

    KEY = "pattern";

    @Value("path")
    protected String pattern;

    @Value("method")
    protected HttpMethod method;

    @Value("headers")
    protected Map<String, String> headers;

    @Value("cookies")
    protected Map<String, String> cookies;

    @Value("body")
    protected Predicate<String> body;

    @Value("remote-addr")
    protected RemoteAddrPredicateFactory<C> remoteAddr;

    @Value("weight")
    protected WeightRoutePredicateFactory<C> weight;

    @Value("query")
    protected QueryRoutePredicateFactory<C> query;

    @Value("between")
    protected BetweenRoutePredicateFactory<C> between;

    @Value("host")
    protected HostRoutePredicateFactory<C> host;
}


```

## Predicate - 断言



- 如果一个路由有多个断言,需全部匹配才可路由过去

```
    gateway:
      routes:
        - id: bing-route
          uri: https://cn.bing.com/
          predicates:
            - name: Path
              args:
                patterns: /search
            - name: Query
              args:
                param: q
                regexp: haha
          order: 10
          metadata:
            hello: world
        - id: order-route
          uri: lb://service-order
          predicates:
            - name: Path
              args:
                patterns: /api/order/**
                matchTrailingSlash: true
          order: 1
        - id: product-route
          uri: lb://service-product
```

这多个必须完全匹配才可以路由到指定位置

- 自定义断言工厂示例:

```
spring:
  cloud:
    gateway:
      routes:
        - id: bing-route
          uri: https://cn.bing.com/
          predicates:
            - name: Path
              args:
                patterns: /search
            - name: Query
              args:
                param: q
                regexp: haha
#           - Vip=user,leifengyang
```

```
ProductController.java      application-route.yml      VipRoutePredicateFactory.java      QueryRoutePredicateFactory.java      OrderCo
16
17     @Component //应用到容器中使其生效
18     public class VipRoutePredicateFactory extends AbstractRoutePredicateFactory<VipRoutePredicateFactory.Config> {
19
20         public VipRoutePredicateFactory() { super(Config.class); }
21
22         /*
23
24             *断言应用条件
25             */
26
27             @Override
28             public Predicate<ServerWebExchange> apply(Config config) {...}
29
30             /*
31             *指定短格式的顺序
32             */
33             @Override
34             public List<String> shortcutFieldOrder() { return Arrays.asList("param", "value"); }
35
36             /*
37             *配置断言的参数/输入内容/判断值
38             */
39             @Validated 3个用法
40             public static class Config {
41
42                 //传user参数的
43                 @NotEmpty 3个用法
44                 private String param;
45
46                 //指定参数等于什么值的
47                 @NotEmpty 3个用法
48                 private String value;
49
50
51
52
53
54
55
56
57
58
59
60
61
```

```
/*
*断言应用条件
*/
@Override
public Predicate<ServerWebExchange> apply(Config config) {
    return new GatewayPredicate() {
        @Override
        public boolean test(ServerWebExchange exchange) {
            ServerHttpRequest request = exchange.getRequest();

            String first = request.getQueryParams().getFirst(config.param);
            return StringUtils.hasText(first) && first.equals(config.value);
        }
    };
}

/*
*指定短格式的顺序
*/
@Override
public List<String> shortcutFieldOrder() {
    return Arrays.asList("param", "value");
}
```

```
*配置断言的参数/输入内容/判断值
*/
@Validated 3个用法
public static class Config {

    //传user参数的
    @NotEmpty 3个用法
    private String param;

    //指定参数等于什么值的
    @NotEmpty 3个用法
    private String value;

    public @NotEmpty String getParam() { 0个用法
        return param;
    }

    public @NotEmpty String getValue() { 0个用法
        return value;
    }

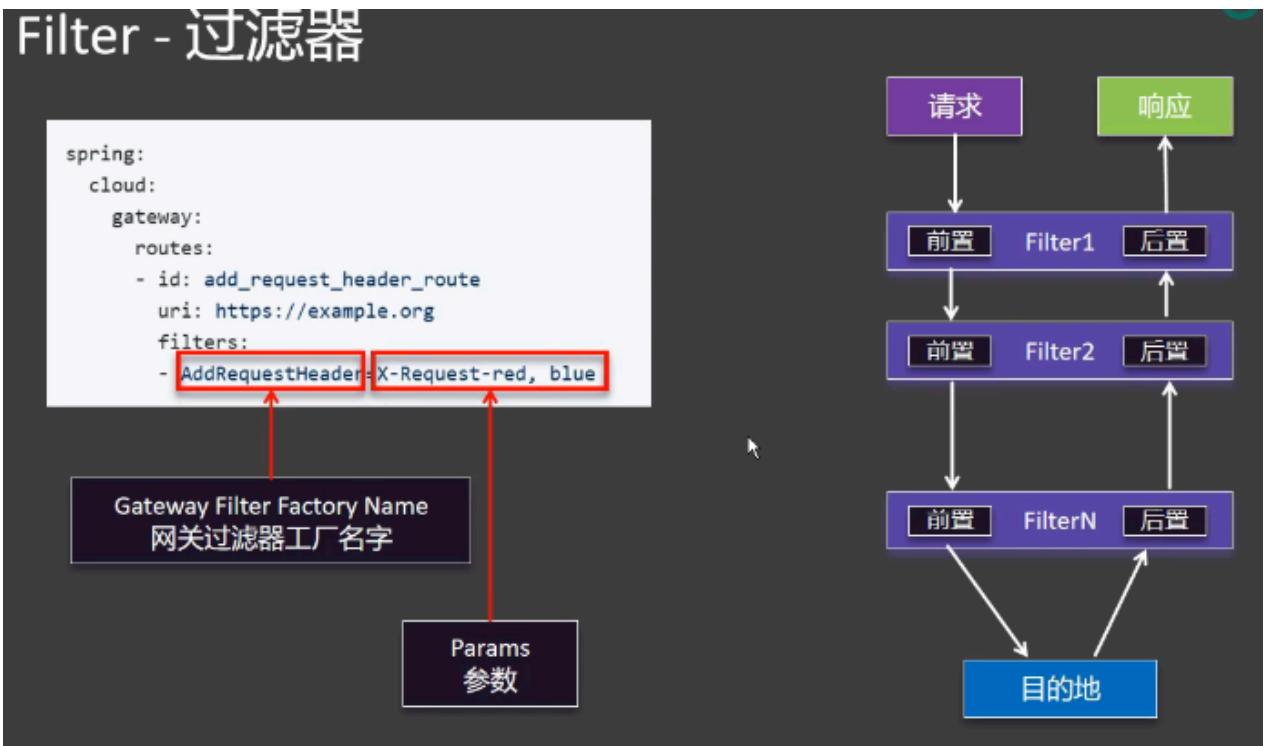
    public Config(@NotEmpty String param, @NotEmpty String value) { 0个用法
        this.param = param;
    }

    public Config(@NotEmpty String value) { 0个用法
        this.value = value;
    }
}
```

## 4.过滤器:

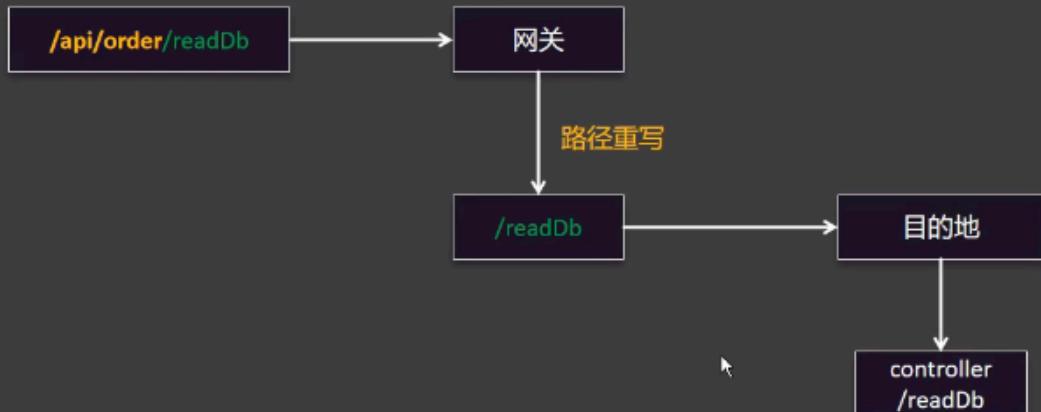
- 基本使用:

# Filter - 过滤器



- 路径重写(以该功能为例): 可以用于在api使用时, 自动添加基准路径之类的

## 路径重写 - rewritePath



我们就无需给每一个微服务都加基准路径了

```

spring:
  cloud:
    gateway:
      routes:
        - id: order-route
          uri: lb://service-order #lb:load-balance, 负载均衡
          predicates: #断言, 遵守规则即转发到上述uri
            - name: Path #设置请求来源的路径
              args:
                patterns: /api/order/**
                matchTrailingSlash: true
          filters:
            #下述写法: 捕获/api/order后面的内容并封装到segment中发给目标微服务
            - RewritePath=/api/order/?(<segment>.*),/$\{segment}
              #添加Header|
              - AddResponseHeader=X-Response-Hxq,Darling
        - id: product-route
          uri: lb://service-product
          predicates:

```

名称	标头	预览	响应	启动器	时间
readDb	<b>常规</b> 请求网址: http://localhost:81/api/c 请求方法: GET 状态代码: 200 OK 远程地址: [:1]:81 引荐来源网址政策: strict-origin-when-cross				
chunk-BqmnBEHA.js	<b>响应标头</b> <input type="checkbox"/> 原始				
content-38bac38e.js					
chunk-BrS3hlmy.js					
chunk-DWVY3kAK.js					
translate-api-87f8aacfjs					
background-df100036.js					
lodash-cfb98fc7.js					
type-84b5ea10.js					
data:image/png;base...					

- 默认filter:在路由配置文件yml中配置,给路由配置中的路由都添加该过滤器,按路由定义顺序执行生效,可以在路由级别被覆盖或者调整,适用于如路径重写,请求头修改等逻辑

```
spring:
  cloud:
    gateway:
      routes:
        - id: product-route
          uri: lb://service-product
          predicates:
            - Path=/api/product/**
          filters:
            - AddResponseHeader=X-Response-Hxq,Darling
          default-filters:
            - #添加Header
```

The screenshot shows a code editor with several tabs open. The active tab is 'application-route.yml'. The code defines a route named 'product-route' that maps to 'lb://service-product'. It includes a 'predicates' section with a path predicate and a 'filters' section with a response header filter. A 'default-filters' section is also present. The code is annotated with comments like '#添加Header'.

- 全局filter:通过代码或配置全局注册配置,对所有请求都生效(适用于通用逻辑,如日志或者认证什么的),无法被单个路由覆盖

```
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Component
public class RtGlobalFilter implements GlobalFilter, Ordered {
  @Override
  public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
    ServerHttpRequest request = exchange.getRequest();
    ServerHttpResponse response = exchange.getResponse();

    String uri = request.getURI().toString();
    long start = System.currentTimeMillis();
    log.info("请求【{}】开始:时间: {}",uri,start);
    //以上为前置逻辑

    Mono<Void> filter = chain.filter(exchange)
      .doFinally(( SignalType result)->{
        //这里内部为后置逻辑, 保证在上面放行之后才会做的事情, 因为它为异步放行
        long end = System.currentTimeMillis();
        log.info("请求【{}】结束:时间: {}, 耗时:{} ",uri,end,start);
      });//异步放行
    return filter;
  }

  @Override
  public int getOrder() {
    return 0;
  }
}
```

The screenshot shows a code editor with several tabs open. The active tab is 'RtGlobalFilter.java'. The code implements the 'GlobalFilter' interface and the 'Ordered' interface. It overrides the 'filter' method, which performs some logging before calling the next filter in the chain. It also overrides the 'getOrder' method to return 0. The code is annotated with comments explaining its purpose.

- 自定义过滤器工厂:类似于断言工厂的自定义,此处以添加一次性令牌为例(注意中间的响应式API写法)

```

15 import java.util.UUID;
16
17 @Component
18 public class OnceTokenGatewayFilterFactory extends AbstractNameValueGatewayFilterFactory {
19     ...
20     @Override
21     public GatewayFilter apply(NameValueConfig config) {
22         return new GatewayFilter() {
23             ...
24             @Override
25             public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
26                 ...
27                 //每次响应前添加一次性令牌, 支持UUID, JWT等格式
28                 return chain.filter(exchange)
29                     .then(Mono.fromRunnable(() -> {
30                         ServerHttpResponse response = exchange.getResponse();
31                         HttpHeaders headers = response.getHeaders();
32                         String value = config.getValue();
33                         if("uuid".equalsIgnoreCase(value)) {
34                             value = UUID.randomUUID().toString();
35                         }
36                         if("jwt".equalsIgnoreCase(value)) {
37                             value = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ
38                             headers.add(config.getName(), value);
39                         }
40                     }));
41             };
42         };
43     }
}

```

### 1. 请求处理阶段:

- 执行过滤器链中的各个过滤器
- 此时请求可能被处理、修改或中断

### 2. 响应准备阶段:

- 当过滤器链执行完毕后, `chain.filter(exchange)`返回完成信号
- 触发`.then()`中的回调函数

### 3. 头部修改阶段:

- 执行`Mono.fromRunnable()`中的代码块
- 直接修改响应对象的headers
- 这些修改会被包含在最终发送给客户端的HTTP响应中

# 八.拓展:

## 1. Docker快速入门:

### 1.1 作用:

- 使得开发环境一样(类似虚拟机,但是不会模拟其硬件,更加轻量化)
- 容器(container):其实就是环境
- 镜像(Image):近似于一个虚拟机的快照,包含部署的应用及关联的库,通过镜像可创建一个个容器
- Dockerfile:用于创建镜像(自动化脚本)

### 1.2 Dockerfile的编写:

FROM 环境官方名称: 标签版本(标签可在docker hub的镜像页面查看)

WORKDIR /路径(指定后续所有Docker命令的工作路径-working directory,若不存在会自动创建)

COPY<本地路径><目标路径>( . 表示根目录所有文件)

RUN 任意shell命令(如常见的pwd,rm,echo都是合法的,创建镜像时使用,也就是搭建镜像的快照环境)

...

CMD[] (用CMD指定Docker容器运行后需要执行的命令)

列举所有的容器: `docker ps`

停止容器: `docker stop <容器 ID>`

重启容器: `docker restart <容器 ID>`

删除容器: `docker rm <容器 ID>`

启动一个远程 Shell: `docker exec -it <容器 ID> /bin/bash`

### 1.3 Volume数据卷:

- 问题:删除容器时,其修改与数据会同关闭虚拟机一样全部丢失
- 解决:使用Volume来创建一个类似于 **共享与本地主机和不同容器的文件夹(就像是数据库和不同主机连接它一样)**
- 使用:
  - 创建:`docker volume create 名字`

- 容器启动后: docker run -dp 80:5000 -v 路径 名字 容器名  
(在该路径写入的所有数据都会永久保存在该数据卷)

## 1.4 docker compose--多容器协作:

停止并删除全容器 compose down(数据卷需要手动删,除非加上--volumes参数)

## 1.5 Docker和Kubernetes(k8s, 只是因为中间省略了八个字母)联系:

Docker:适用于单主机多容器

Kubernetes:将各容器分发到【集群(cluster)】上管理(主要改变的是容器的管理)

## 2. Redis

### 2.1 介绍:

Redis(Remote Dictionary Server): 即远程字典服务，是一个开源的使用C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API

#### 知识拓展

##### ◦ 为什么说Redis支持事务?

Redis支持事务是因为它提供了MULTI、EXEC、DISCARD和WATCH等命令，使得用户可以将多个命令打包成一个事务，保证事务的原子性。使用Redis事务，用户可以将多个命令一次性发送给Redis服务器，并在EXEC命令执行时，Redis会按照事务中的顺序执行所有的命令。如果事务中的任何一个命令执行失败，Redis会撤销该事务中已经执行的所有命令，从而保证了事务的原子性。另外，Redis还提供了WATCH命令，可以监控一个或多个键，当任何一个被监控的键被其他客户端修改时，当前事务就会被中断。这个特性可以防止多个客户端之间的数据竞争。因此，Redis提供了一种简单而有效的事务机制，可以用来保证数据的完整性和一致性。

##### ◦ Redis中的事务和MySQL中的事务有什么不同?

- 数据存储方式不同。Redis是基于内存的键值对数据库，而MySQL是关系型数据库，数据存储方式不同。
- 事务支持的粒度不同。Redis的事务支持的粒度是单个命令，即一组命令的执行是原子性的；而MySQL的事务支持的粒度是整个事务，即一组SQL语句的执行是原子性的。
- 并发控制机制不同。Redis的事务采用乐观锁(WATCH命令)，如果监控的键发生变化，事务就会被回滚，而MySQL的事务采用悲观锁或MVCC机制，可以避免读写冲突。
- 支持的隔离级别不同。MySQL支持多个隔离级别，如读未提交、读已提交、可重复读和串行化等，而Redis仅支持串行化隔离级别。
- 实现方式不同。Redis的事务是在客户端实现的，而MySQL的事务是在服务器端实现的。
- 回滚机制不同。Redis的事务在发生错误时不会回滚已经执行的命令，而是继续执行后面的命令；而MySQL的事务会回滚所有已经执行的SQL语句。

总的来说，Redis和MySQL的事务机制有很大的差异，适用的场景也不同。Redis的事务适合于多个命令的批量操作，而MySQL的事务适合于复杂的业务逻辑，需要对多个SQL语句进行控制的场景。

## 认识NoSQL

	SQL	NoSQL	
数据结构	结构化(Structured)	非结构化	#1 键值类型 (Redis) #2 文档类型 (MongoDB) #3 列类型 (HBase) #4 Graph类型 (Neo4j)
数据关联	关联的(Relational)	无关联的	
查询方式	SQL查询	非SQL	
事务特性	ACID	BASE	
存储方式	磁盘	内存	
扩展性	垂直	水平	
使用场景	1) 数据结构固定 2) 相关业务对数据安全性、一致性要求较高	1) 数据结构不固定 2) 对一致性、安全性要求不高 3) 对性能要求	

## 认识Redis

Redis诞生于2009年全称是**Remote Dictionary Server**, 远程词典服务器

**特征:**

- 键值 (key-value) 型, value支持多种不同数据结构, 功能丰富
- 单线程, 每个命令具备原子性
- 低延迟, 速度快 (基于内存、IO多路复用、良好的编码)。
- 支持数据持久化
- 支持主从集群、分片集群
- 支持多语言客户端

Redis安装完成后就自带了命令行客户端：redis-cli，使用方式如下：

```
redis> redis-cli [options] [commands]
```

其中常见的options有：

- `-h 127.0.0.1`：指定要连接的redis节点的IP地址，默认是127.0.0.1
- `-p 6379`：指定要连接的redis节点的端口，默认是6379
- `-a 123321`：指定redis的访问密码

其中的commands就是Redis的操作命令，例如：

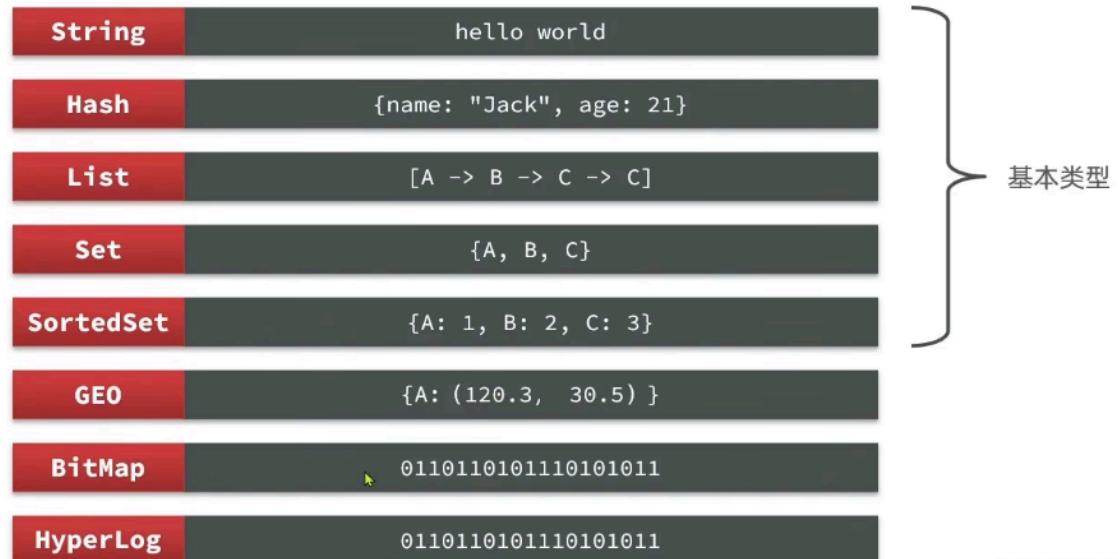
- `ping`：与redis服务端做心跳测试，服务端正常会返回 `pong`

不指定command时，会进入 `redis-cli` 的交互控制台：

GEO是地理坐标

## Redis数据结构介绍

Redis是一个key-value的数据库，key一般是String类型，不过value的类型多种多样：



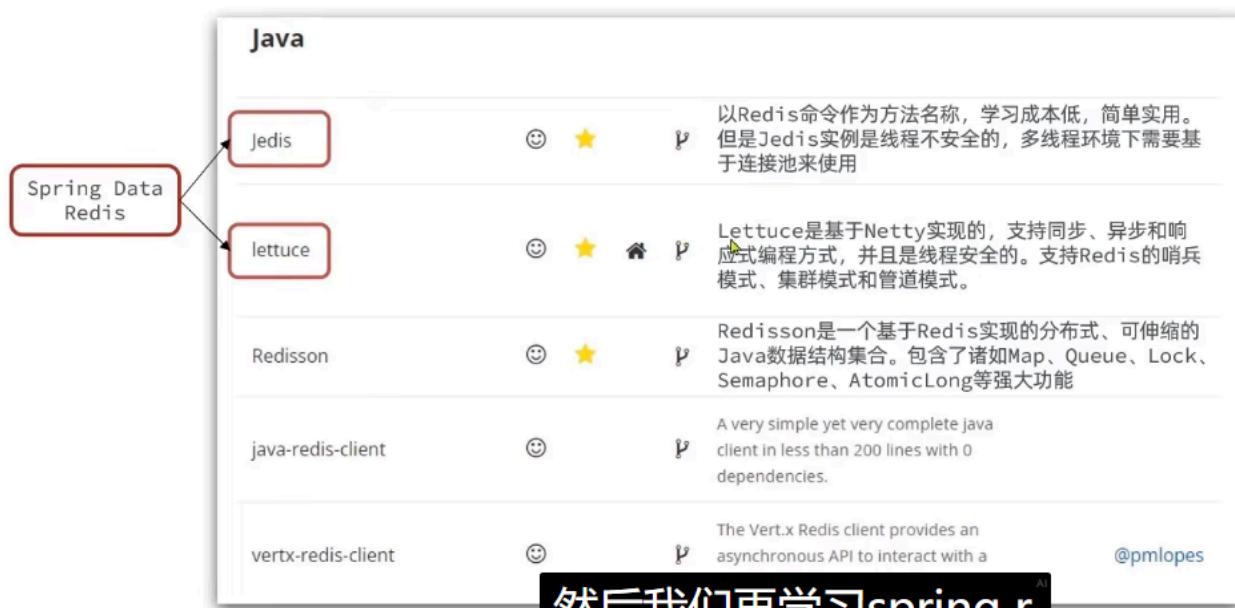
## SortedSet类型

Redis的SortedSet是一个可排序的set集合，与Java中的TreeSet有些类似，但底层数据结构却差别很大。SortedSet中的每一个元素都带有一个score属性，可以基于score属性对元素排序，底层的实现是一个跳表（SkipList）加hash表。SortedSet具备下列特性：

- 可排序
- 元素不重复
- 查询速度快

因为SortedSet的可排序特性，经常被用来实现排行榜这样的功能。

## Redis的Java客户端



## 2.2 特性：

- 高性能：Redis是一个基于内存的数据库，它的性能非常高。它的数据可以存储在内存中，这使得它的读写速度非常快。
- 数据结构多样：Redis支持多种数据结构，包括字符串、列表、哈希表、集合和有序集合等。这些数据结构的支持使得Redis在不同的应用场景中都能够得到很好的应用。
- 持久化存储：Redis支持将数据持久化到磁盘上，保证了数据的安全性。它提供了两种持久化方式：RDB（Redis Database）和AOF（Append-Only File）。
- 支持事务：Redis支持事务，可以将多个命令打包成一个事务，保证事务的原子性，从而保证了数据的完整性。
- PS：Redis支持事务是因为它提供了MULTI、EXEC、DISCARD和WATCH等命令，使得用户可以将多个命令打包成一个事务，保证事务的原子性。（注意：单个Redis结点支持事务，但是Redis集群不支持事务，即核心命令依旧是单线程）

- 分布式: Redis支持分布式部署, 可以将数据分散到多个节点上, 从而提高了系统的可扩展性和容错性。
- 发布/订阅模式: Redis支持发布/订阅模式, 可以在多个客户端之间实现实时消息推送。
- 简单易用: Redis的命令非常简单易用, 学习和使用都比较容易。同时, Redis还提供了丰富的客户端库, 可以方便地集成到各种编程语言中。

## 2.3 应用:

- 缓存: Redis最常见的应用场景之一, 将常用的数据存储在Redis中, 可以有效地减轻数据库的压力, 提高系统的响应速度。
- 计数器: 利用Redis的原子性和高速读写的特点, 可以实现实时计数器、访问次数统计等功能。
- 排行榜和统计分析: 通过将计数器和有序集合结合起来, 可以实现排行榜和统计分析功能, 如热门商品排行、网站访问排行等
- 实时消息系统: 利用Redis的发布/订阅机制, 可以构建实时消息系统, 如聊天室、实时通知等。
- 地理位置应用: 通过Redis的地理位置功能, 可以实现周边搜索、定位服务等功能。
- 分布式锁: 利用Redis的SETNX命令和过期时间特性, 可以实现分布式锁, 保证多个进程/线程之间的数据一致性。
- 分布式缓存: 通过Redis的集群和哨兵机制, 可以构建分布式缓存系统, 提高系统的可靠性和可扩展性。
- 会话管理: 将用户的会话信息存储在Redis中, 可以实现分布式会话管理, 提高系统的可靠性和可伸缩性

## 2.4 jedis(Java Redis)

线程不安全

## Jedis使用的基本步骤：

1. 引入依赖

2. 创建Jedis对象，建立连接

3. 使用Jedis，方法名与Redis命令一致

4. 释放资源

### Jedis连接池

Jedis本身是线程不安全的，并且频繁的创建和销毁连接会有性能损耗，因此我们推荐大家使用Jedis连接池代替Jedis的直连方式。

```
public class JedisConnectionFactory {
    private static final JedisPool jedisPool;

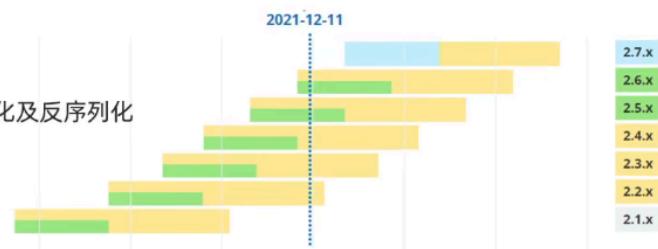
    static {
        JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();
        // 最大连接
        jedisPoolConfig.setMaxTotal(8);
        // 最大空闲连接
        jedisPoolConfig.setMaxIdle(8);
        // 最小空闲连接
        jedisPoolConfig.setMinIdle(0);
        // 设置最长等待时间, ms
        jedisPoolConfig.setMaxWaitMillis(200);
        jedisPool = new JedisPool(jedisPoolConfig, "192.168.150.101", 6379,
            1000, "123321");
    }
    // 获取Jedis对象
    public static Jedis getJedis(){
        return jedisPool.getResource();
    }
}
```

## 2.5 SpringDataRedis

### SpringDataRedis

SpringData是Spring中数据操作的模块，包含对各种数据库的集成，其中对Redis的集成模块就叫做SpringDataRedis，官网地址：<https://spring.io/projects/spring-data-redis>

- 提供了对不同Redis客户端的整合（Lettuce和Jedis）
- 提供了RedisTemplate统一API来操作Redis
- 支持Redis的发布订阅模型
- 支持Redis哨兵和Redis集群
- 支持基于Lettuce的响应式编程
- 支持基于JDK、JSON、字符串、Spring对象的数据序列化及反序列化
- 支持基于Redis的JDKCollection实现



### SpringDataRedis快速入门

SpringDataRedis中提供了RedisTemplate工具类，其中封装了各种对Redis的操作。并且将不同数据类型的操作API封装到了不同的类型中：

API	返回值类型	说明
<code>redisTemplate.opsForValue()</code>	<code>ValueOperations</code>	操作 <code>String</code> 类型数据
<code>redisTemplate.opsForHash()</code>	<code>HashOperations</code>	操作 <code>Hash</code> 类型数据
<code>redisTemplate.opsForList()</code>	<code>ListOperations</code>	操作 <code>List</code> 类型数据
<code>redisTemplate.opsForSet()</code>	<code>SetOperations</code>	操作 <code>Set</code> 类型数据
<code>redisTemplate.opsForZSet()</code>	<code>ZSetOperations</code>	操作 <code>SortedSet</code> 类型数据
<code>redisTemplate</code>		通用的命令

## SpringDataRedis快速入门

### 3. 注入RedisTemplate

```
@Autowired  
private RedisTemplate redisTemplate;
```

### 4. 编写测试

```
@SpringBootTest  
public class RedisTest {  
  
    @Autowired  
    private RedisTemplate redisTemplate;  
  
    @Test  
    void testString() {  
        // 插入一条string类型数据  
        redisTemplate.opsForValue().set("name", "李四");  
        // 读取一条string类型数据  
        Object name = redisTemplate.opsForValue().get("name");  
        System.out.println("name = " + name);  
    }  
}
```

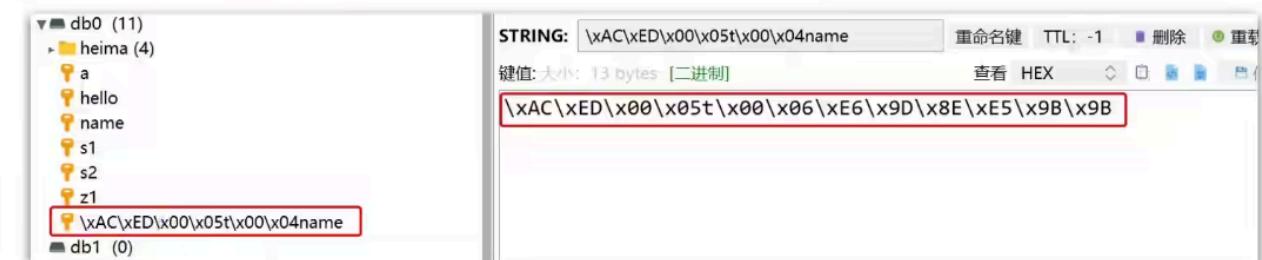
ok

SpringDataRedis的使用步骤：

1. 引入spring-boot-starter-data-redis依赖
2. 在application.yml配置Redis信息
3. 注入RedisTemplate

## SpringDataRedis的序列化方式

RedisTemplate可以接收任意Object作为值写入Redis，只不过写入前会把Object序列化为字节形式，默认是采用JDK序列化，得到的结果是这样的：



The screenshot shows the Redis Data Grid interface. On the left, there's a tree view of databases db0 and db1. In db0, there's a 'heimat' folder containing keys 'a', 'hello', 'name', 's1', 's2', and 'z1'. A key 'name' is selected and its value is shown on the right. The value is displayed in two formats: STRING: '\xAC\xED\x00\x05t\x00\x04name' and HEX: '\xAC\xED\x00\x05t\x00\x06\xE6\x9D\x8E\xE5\x9B\x9B'. The HEX value is highlighted with a red border.

缺点：

- 可读性差
- 内存占用较大

## SpringDataRedis的序列化方式

我们可以自定义RedisTemplate的序列化方式，代码如下：

```
@Bean
public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory redisConnectionFactory)
    throws UnknownHostException {
    // 创建Template
    RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();
    // 设置连接工厂
    redisTemplate.setConnectionFactory(redisConnectionFactory);
    // 设置序列化工具
    GenericJackson2JsonRedisSerializer jsonRedisSerializer =
        new GenericJackson2JsonRedisSerializer();
    // key和hashKey采用 string序列化
    redisTemplate.setKeySerializer(RedisSerializer.string());
    redisTemplate.setHashKeySerializer(RedisSerializer.string());
    // value和hashValue采用 JSON序列化
    redisTemplate.setValueSerializer(jsonRedisSerializer);
    redisTemplate.setHashValueSerializer(jsonRedisSerializer);
    return redisTemplate;
}
```

## StringRedisTemplate

尽管JSON的序列化方式可以满足我们的需求，但依然存在一些问题，如图：



The screenshot shows the Redis Data Grid interface. A key 'user:100' is selected, and its value is a JSON object: { "@class": "com.heimat.redis.pojo.User", "name": "虎哥", "age": 21 }. The '@class' field is highlighted with a red border.

为了在反序列化时知道对象的类型，JSON序列化器会将类的class类型写入json结果中，存入Redis，会带来额外的内存开销。

## StringRedisTemplate

为了节省内存空间，我们并不会使用JSON序列化器来处理value，而是统一使用String序列化器，要求只能存储String类型的key和value。当需要存储Java对象时，手动完成对象的序列化和反序列化。



Spring默认提供了一个StringRedisTemplate类，它的key和value的序列化方式默认就是String方式。省去了我们自定义RedisTemplate的过程：

```
@Autowired
private StringRedisTemplate stringRedisTemplate;
// JSON工具
private static final ObjectMapper mapper = new ObjectMapper();
@Test
void testStringTemplate() throws JsonProcessingException {
    // 准备对象
    User user = new User("虎哥", 18);
    // 手动序列化
    String json = mapper.writeValueAsString(user);
    // 写入一条数据到redis
    stringRedisTemplate.opsForValue().set("user:200", json);

    // 读取数据
    String val = stringRedisTemplate.opsForValue().get("user:200");
    // 反序列化
    User user1 = mapper.readValue(val, User.class);
    System.out.println("user1 = " + user1);
}
```

RedisTemplate的两种序列化实践方案：

方案一：

1. 自定义RedisTemplate
2. 修改RedisTemplate的序列化器为GenericJackson2JsonRedisSerializer

方案二：

1. 使用StringRedisTemplate
2. 写入Redis时，手动把对象序列化为JSON
3. 读取Redis时，手动把读取到的JSON反序列化为对象

```
/**  
 * Redis使用FastJson序列化  
 */  
public class FastJsonRedisSerializer<T> implements RedisSerializer<T>  
{  
  
    public static final Charset DEFAULT_CHARSET = Charset.forName("UTF-8");  
  
    private Class<T> clazz;  
  
    static  
{  
        ParserConfig.getGlobalInstance().setAutoTypeSupport(true);  
    }  
  
    public FastJsonRedisSerializer(Class<T> clazz)  
    {  
        super();  
        this.clazz = clazz;  
    }  
  
    @Override  
    public byte[] serialize(T t) throws SerializationException  
{  
        if (t == null)  
        {  
            return new byte[0];  
            // 如果对象为null， 返回一个空的字节数组  
        }  
        return JSON.toJSONString(t, SerializerFeature.WriteClassName).getBytes(DEFAULT_  
            // 如果对象不为null:  
            // 1. 使用FastJSON将对象t序列化为JSON字符串  
            // 2. SerializerFeature.WriteClassName 参数会在JSON字符串中写入对象的类名信息  
            // 3. 将生成的JSON字符串转换为UTF-8编码的字节数组并返回  
    }  
  
    @Override  
    public T deserialize(byte[] bytes) throws SerializationException  
{  
        if (bytes == null || bytes.length <= 0)  
        {  
            return null;  
            // 如果字节数组为空， 返回null  
        }  
    }  
}
```

```

    }

    String str = new String(bytes, DEFAULT_CHARSET);
    // 将字节数组转换为UTF-8编码的字符串

    return JSON.parseObject(str, clazz);
    // 使用FastJSON将字符串解析为clazz类型的Java对象并返回
}

protected JavaType getJavaType(Class<?> clazz)
{
    return TypeFactory.defaultInstance().constructType(clazz);
}

}

```

## 2.6 验证码及注册登录

- session：基于cookie实现，储存在服务端，即时失效，扩展性差，性能相对不够好，适用于高安全场景与短期会话，无需跨域的场景

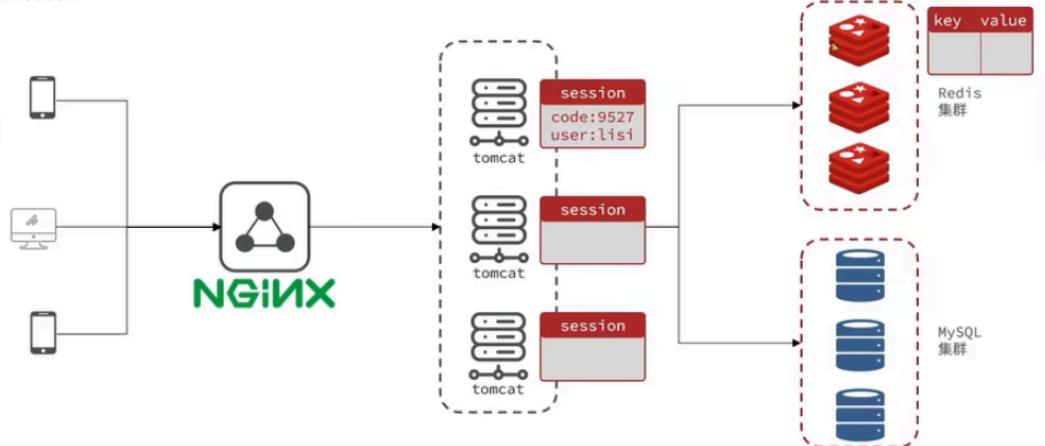
**集群的session共享问题**

**用token啊，你这样会造成服务器中心化，一蹦全崩**

**session共享问题：**多台Tomcat并不共享session存储空间，当请求切换到不同tomcat服务时导致数据丢失的问题。

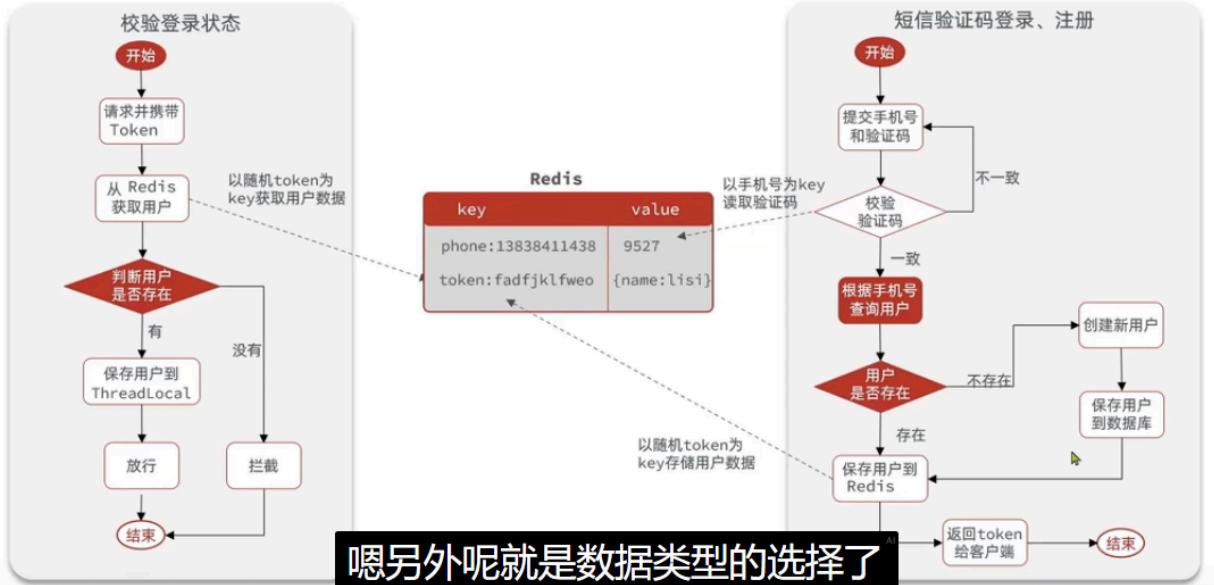
session的替代方案应该满足：

- 数据共享
- 内存存储
- key、value结构



- JWT：基于token实现，token会携带用户数据，无需服务端储存，储存在客户端，天然支持分布式架构，可携带数据，减少数据库查询，体积较大，适用于分布式系统，无状态API(RESTful接口)，跨域认证
- Redis：基于token实现，天然支持分布式，token仅作为登录凭证，需要自行二次开发实现token逻辑，需维护Redis可用

## 基于Redis实现共享session登录



## 2.7 缓存：

### 1. 更新策略：

缓存更新策略

	内存淘汰	超时剔除	主动更新
说明	不用自己维护，利用Redis的内存淘汰机制，当内存不足时自动淘汰部分数据。下次查询时更新缓存。	给缓存数据添加TTL时间，到期后自动删除缓存。下次查询时更新缓存。	编写业务逻辑，在修改数据库的同时，更新缓存。 
一致性	差	一般	好
维护成本	无	低	高

业务场景：

- 低一致性需求：使用内存淘汰机制。例如店铺类型的查询缓存
- 高一致性需求：主动更新，并以超时剔除作为兜底方案。例如店铺详情查询的缓存

01

### Cache Aside Pattern

由缓存的调用者，在更新数据库的同时更新缓存

02

### Read/Write Through Pattern

缓存与数据库整合为一个服务，由服务来维护一致性。调用者调用该服务，无需关心缓存一致性问题。

03

### Write Behind Caching Pattern

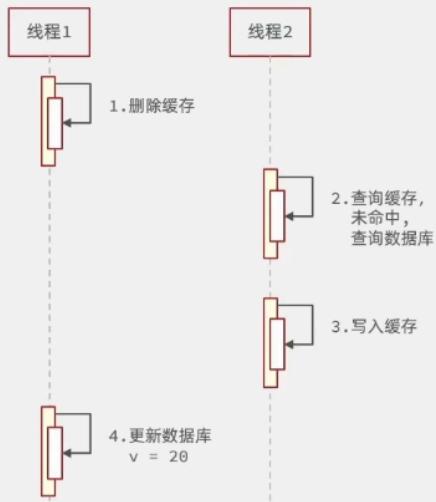
调用者只操作缓存，由其它线程异步的将缓存数据持久化到数据库，保证最终一致。



## Cache Aside Pattern 就是没更新的值

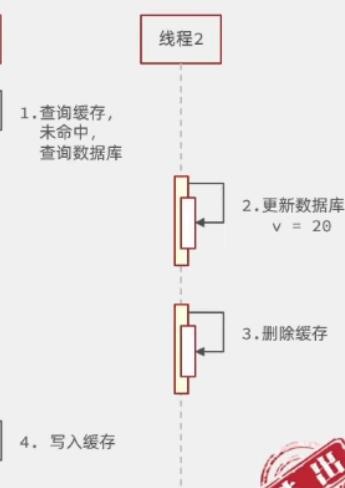
缓存 10

### 先删除缓存，再操作数据库



数据库 20

### 先操作数据库，再删除缓存



## 缓存更新策略的最佳实践方案：

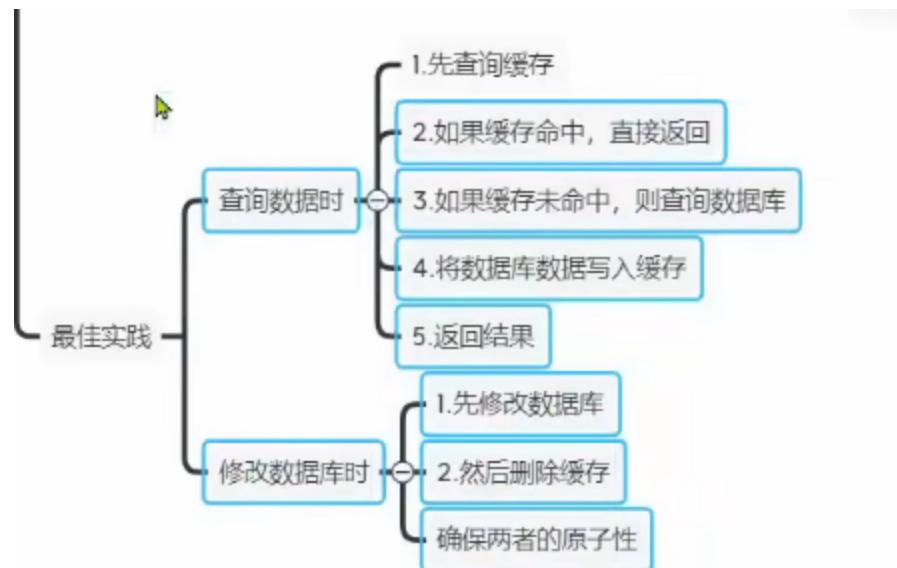
1. 低一致性需求：使用Redis自带的内存淘汰机制
2. 高一致性需求：主动更新，并以超时剔除作为兜底方案

### ◆ 读操作：

- 缓存命中则直接返回
- 缓存未命中则查询数据库，并写入缓存，设定超时时间

### ◆ 写操作：

- 先写数据库，然后再删除缓存
- 要确保数据库与缓存操作的原子性



## 2. 缓存穿透：

缓存穿透是指客户端请求的数据在缓存中和数据库中都不存在，这样缓存永远不会生效，这些请求都会打到数据库。

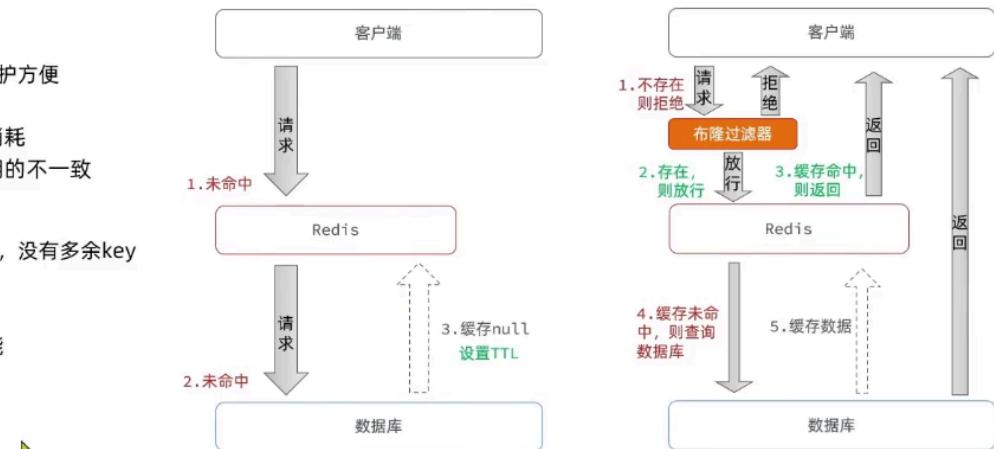
常见的解决方案有两种：

- 缓存空对象

- ◆ 优点：实现简单，维护方便
- ◆ 缺点：
  - 额外的内存消耗
  - 可能造成短期的不一致

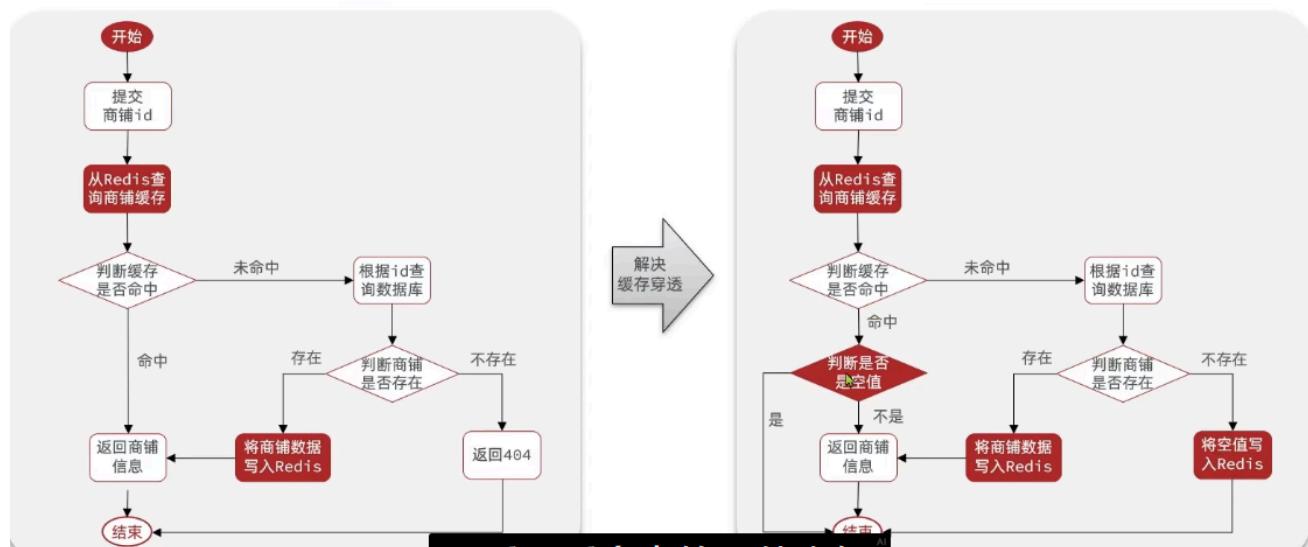
- 布隆过滤

- ◆ 优点：内存占用较少，没有多余key
- ◆ 缺点：
  - 实现复杂
  - 存在误判可能



### 缓存穿透

这边



## 缓存穿透产生的原因是什么？

- 用户请求的数据在缓存中和数据库中都不存在，不断发起这样的请求，给数据库带来巨大压力

## 缓存穿透的解决方案有哪些？

- 缓存null值
- 布隆过滤
- 增强id的复杂度，避免被猜测id规律
- 做好数据的基础格式校验
- 加强用户权限校验
- 做好热点参数的限流

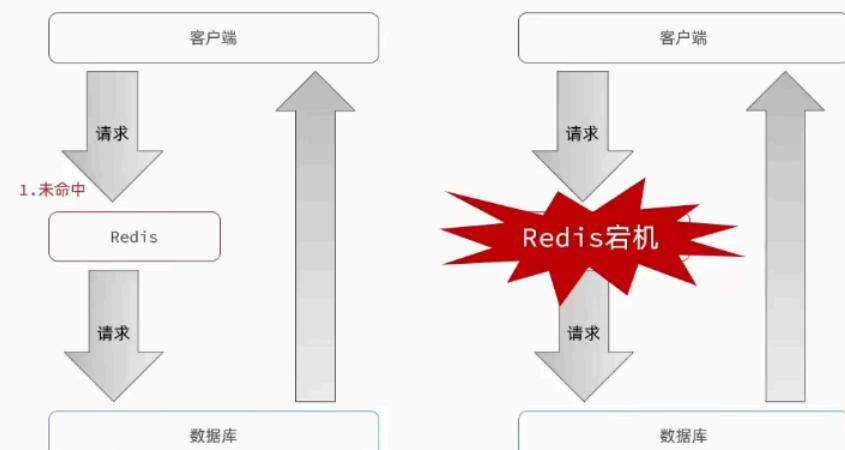
### 3. 缓存雪崩：

#### 缓存雪崩

缓存雪崩是指在同一时段大量的缓存key同时失效或者Redis服务宕机，导致大量请求到达数据库，带来巨大压力。

##### 解决方案：

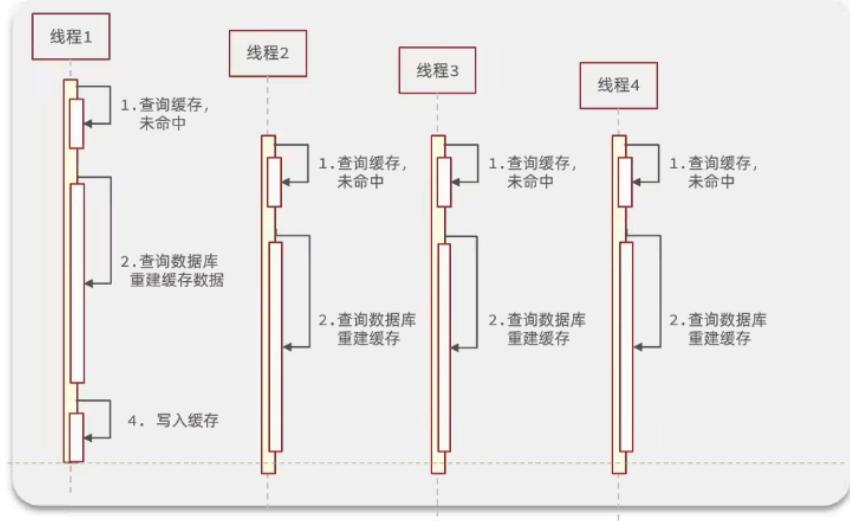
- ◆ 给不同的Key的TTL添加随机值
- ◆ 利用Redis集群提高服务的可用性
- ◆ 给缓存业务添加降级限流策略
- ◆ 给业务添加多级缓存



## 4. 缓存击穿：

### 缓存击穿

缓存击穿问题也叫热点Key问题，就是一个被高并发访问并且缓存重建业务较复杂的key突然失效了，无数的请求访问会在瞬间给数据库带来巨大的冲击。

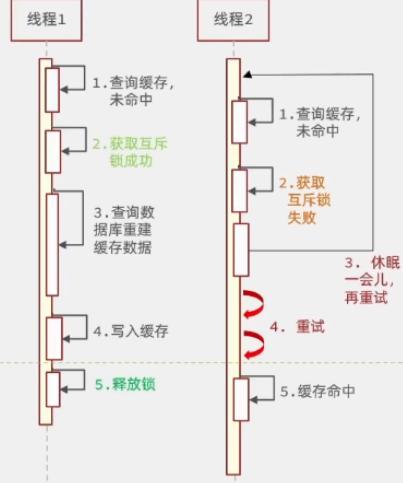


www.itheima.com

夕一夕汉，夕一夕不休，夕取夕时的夕，夕云取大夕的夕个：

### 缓存击穿

#### 互斥锁



#### 逻辑过期



那么这两种方案啊

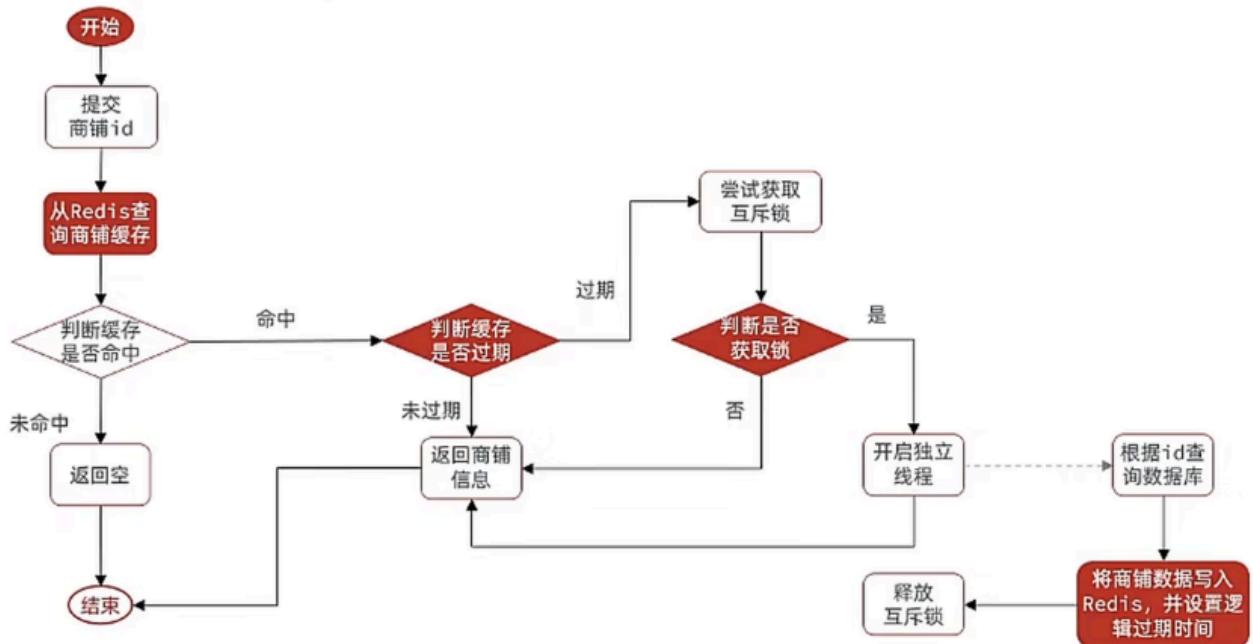
高级软件人才培训专家

显示颜色显示 显示颜色显示 显示颜色显示 显示颜色显示 显示颜色显示 显示颜色显示

解决方案	优点	缺点
<b>互斥锁</b>	<ul style="list-style-type: none"> <li>没有额外的内存消耗</li> <li>保证一致性</li> <li>实现简单</li> </ul>	<ul style="list-style-type: none"> <li>线程需要等待，性能受影响</li> <li>可能有死锁风险</li> </ul>
<b>逻辑过期</b>	<ul style="list-style-type: none"> <li>线程无需等待，性能较好</li> </ul>	<ul style="list-style-type: none"> <li>不保证一致性</li> <li>有额外内存消耗</li> <li>实现复杂</li> </ul>

## 基于逻辑过期方式解决缓存击穿问题

需求：修改根据id查询商铺的业务，基于逻辑过期方式来解决缓存击穿问题



## 2.8 秒杀券(锁):

### 1.全局ID生成器:

#### 全局ID生成器

每个店铺都可以发布优惠券：

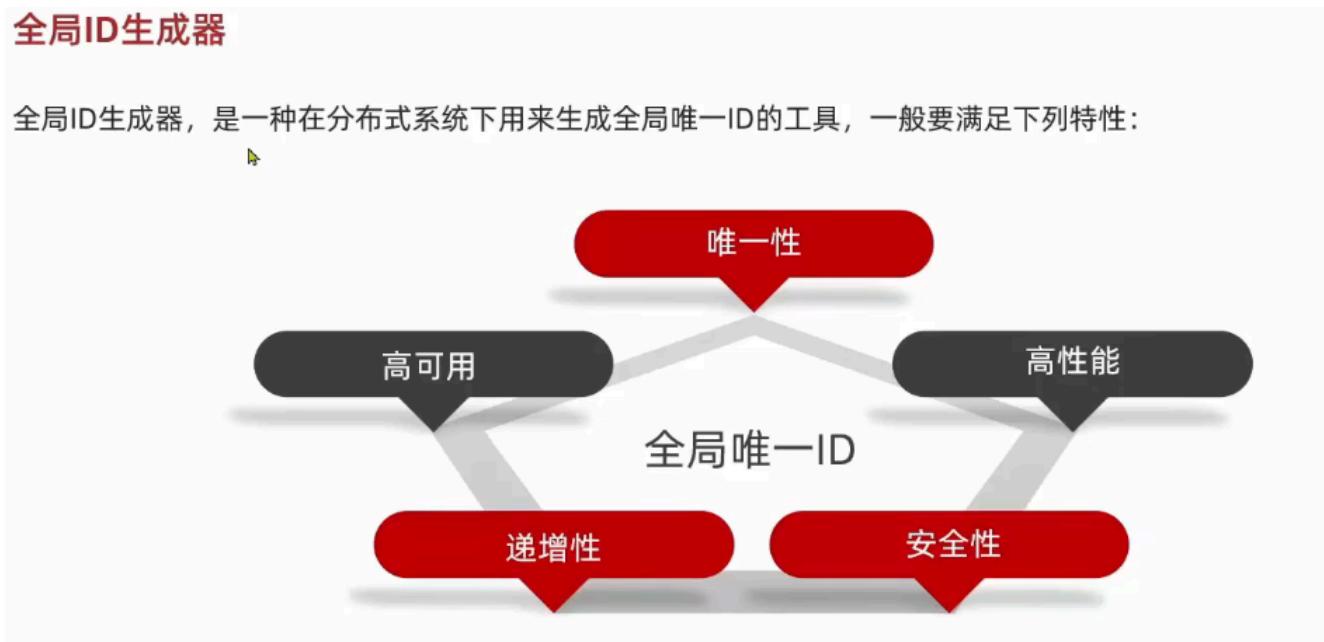


当用户抢购时，就会生成订单并保存到tb\_voucher\_order这张表中，而订单表如果使用数据库自增ID就存在一些问题：

- id的规律性太明显
- 受单表数据量的限制

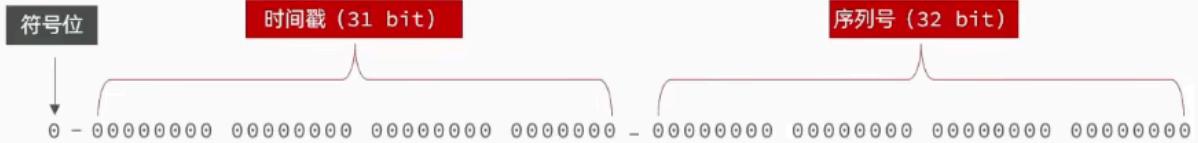
#### 全局ID生成器

全局ID生成器，是一种在分布式系统下用来生成全局唯一ID的工具，一般要满足下列特性：



## 全局ID生成器

为了增加ID的安全性，我们可以不直接使用Redis自增的数值，而是拼接一些其它信息：



ID的组成部分：

- ◆ 符号位：1bit，永远为0
- ◆ 时间戳：31bit，以秒为单位，可以使用69年
- ◆ 序列号：32bit，秒内的计数器，支持每秒产生 $2^{32}$ 个不同ID

全局唯一ID生成策略：

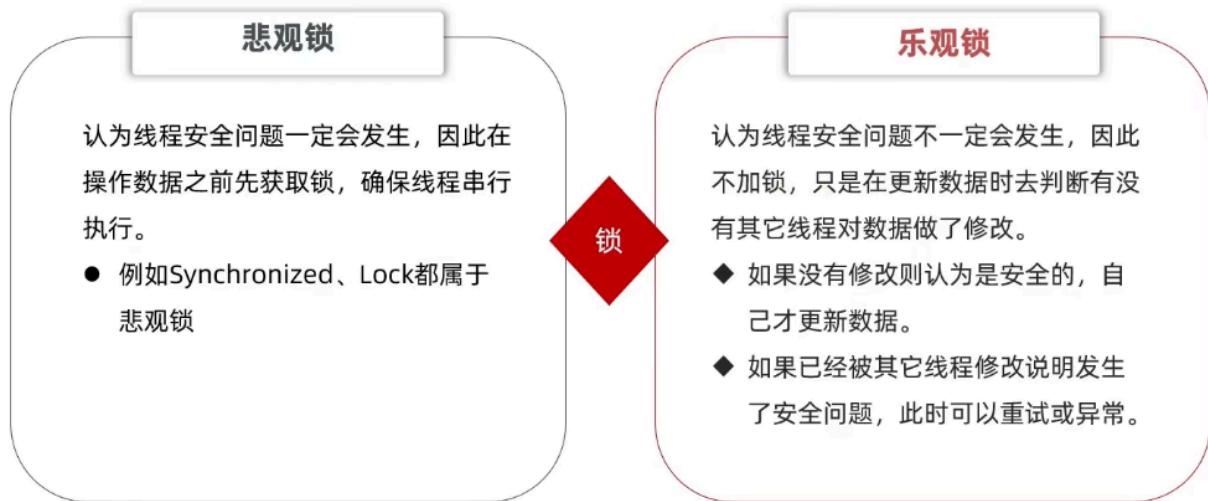
- UUID
- Redis自增
- snowflake算法
- 数据库自增

Redis自增ID策略：

- 每天一个key，方便统计订单量
- ID构造是 时间戳 + 计数器

## 2.超卖与锁(单机/集群):

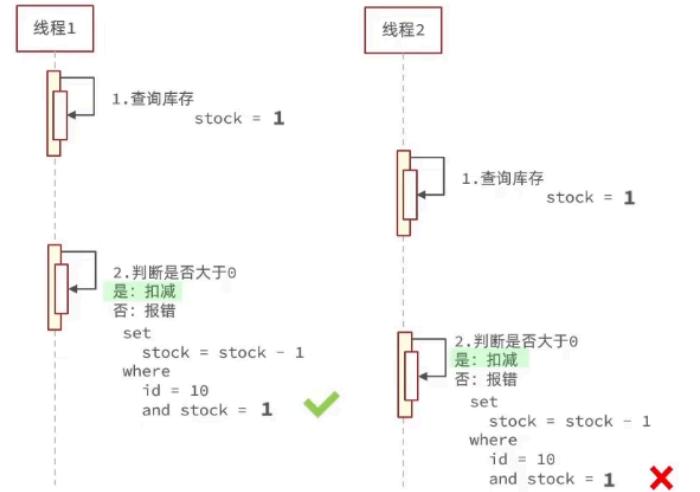
超卖问题是典型的多线程安全问题，针对这一问题的常见解决方案就是加锁：



乐观锁的关键是判断之前查询得到的数据是否有被修改过，常见的方法有两种：

### ◆ CAS法

id	stock
10	0



- 仅单机(可解决多线程,但无法解决多进程,需要分布式锁):
  - 事务：使用代理对象来实现Spring的事务管理(其不拦截类的内部方法调用)
  - 如何使用代理对象：调用AopContext.currentProxy(),并添加  
@EnableAspectJAutoProxy(exposeProxy=true)暴露代理对象,添加aspectj依赖,或者  
注入方法类自己直接调用函数与添加依赖
- 集群模式下:
  - Redis的互斥锁

分布式锁的核心是实现多进程之间互斥，而满足这一点的方式有很多，常见的有三种：

	MySQL	Redis	Zookeeper
互斥	利用mysql本身的互斥锁机制	利用setnx这样的互斥命令	利用节点的唯一性和有序性实现互斥
高可用	好	好	好
高性能	一般	好	一般
安全性	断开连接，自动释放锁	利用锁超时时间，到期释放	临时节点，断开连接自动释放

## 基于Redis的分布式锁

实现分布式锁时需要实现的两个基本方法：

- 获取锁：

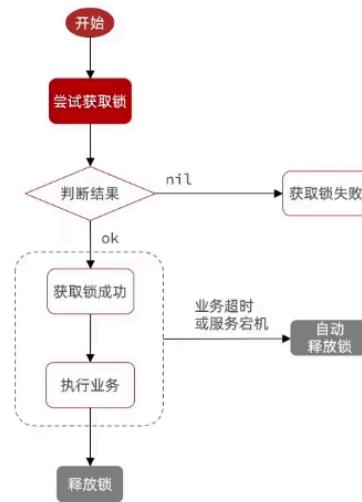
- 互斥：确保只能有一个线程获取锁
- 非阻塞：尝试一次，成功返回true，失败返回false

```
# 添加锁, NX是互斥、EX是设置超时时间  
SET lock thread1 NX EX 10
```

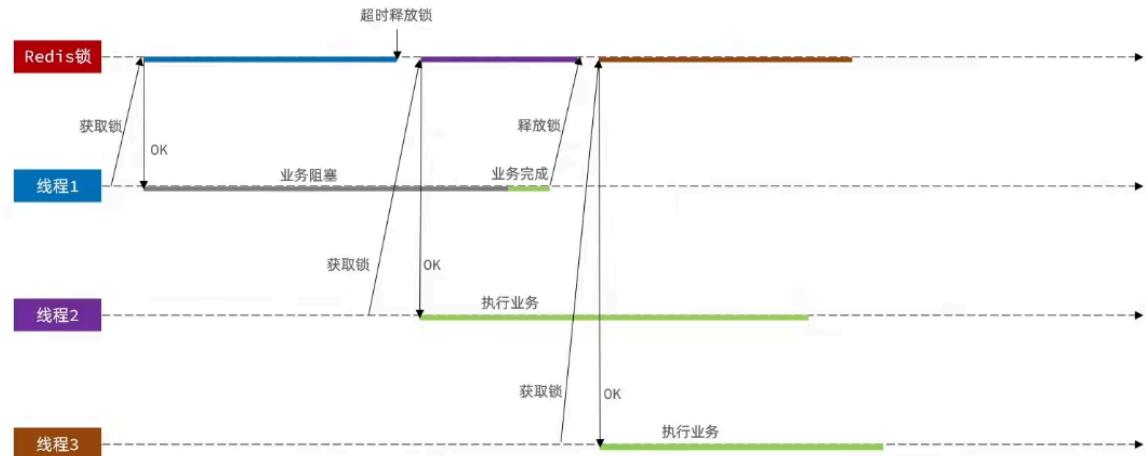
- 释放锁：

- 手动释放
- 超时释放：获取锁时添加一个超时时间

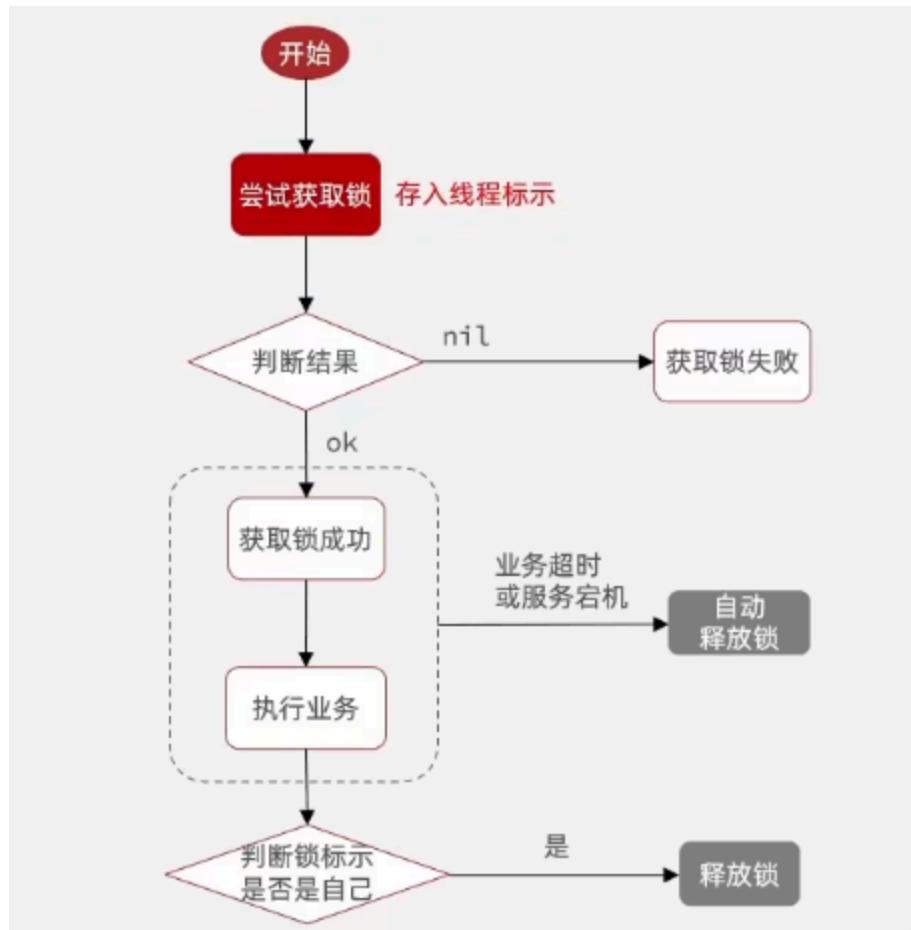
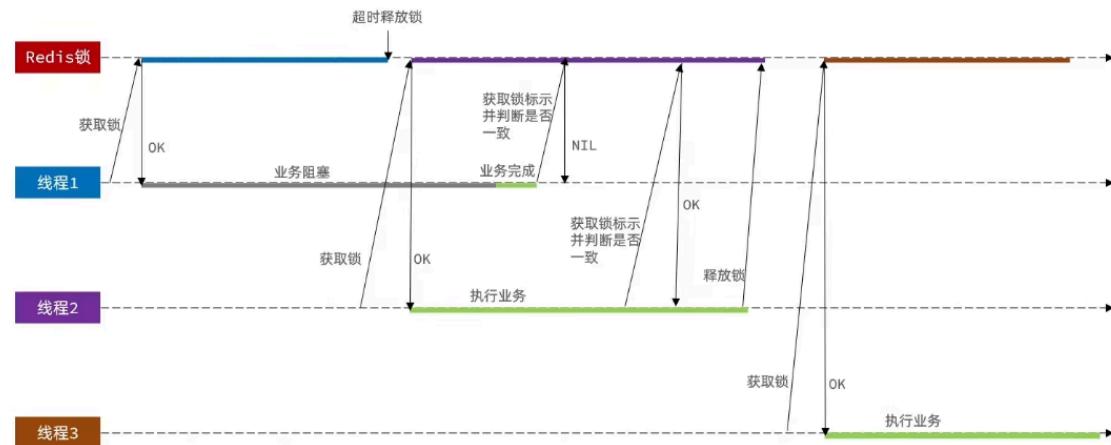
```
# 释放锁, 删掉即可  
DEL key
```



## 基于Redis的分布式锁



优化误删-解决一人多单问题:(添加锁标识)



Lua脚本(使判断锁标识与释放锁为原子性-没啥用其实)

## Redis的Lua脚本

Redis提供了Lua脚本功能，在一个脚本中编写多条Redis命令，确保多条命令执行时的原子性。Lua是一种编程语言，它的基本语法大家可以参考网站：<https://www.runoob.com/lua/lua-tutorial.html>

这里重点介绍Redis提供的调用函数，语法如下：

```
# 执行redis命令  
redis.call('命令名称', 'key', '其它参数', ...)
```

例如，我们要执行set name jack，则脚本是这样：

```
# 执行 set name jack  
redis.call('set', 'name', 'jack')
```

例如，我们要先执行set name Rose，再执行get name，则脚本如下：

```
# 先执行 set name jack  
redis.call('set', 'name', 'jack')  
# 再执行 get name  
local name = redis.call('get', 'name')  
# 返回  
return name
```

高级操作人

## Redis的Lua脚本

写好脚本以后，需要用Redis命令来调用脚本，调用脚本的常见命令如下：

```
127.0.0.1:6379> help @scripting  
  
EVAL script numkeys key [key ...] arg [arg ...]  
summary: Execute a Lua script server side  
since: 2.6.0
```

例如，我们要执行 `redis.call('set', 'name', 'jack')` 这个脚本，语法如下：

```
# 调用脚本  
EVAL "return redis.call('set', 'name', 'jack')" 0  
  
↓  
脚本内容          脚本需要的key类型的参数个数
```

如果脚本中的key、value不想写死，可以作为参数传递。key类型参数会放入KEYS数组，其它参数会放入ARGV数组，在脚本中可以从KEYS和ARGV数组获取这些参数：

```
# 调用脚本  
EVAL "return redis.call('set', KEYS[1], ARGV[1])" 1  
  
↓  
脚本内容          脚本需要的key类型的参数个数
```

高级操作人

## 基于Redis的分布式锁优化

基于setnx实现的分布式锁存在下面的问题：



- Redisson(用锁的时候单独配, springboot自带的会替代原先spring对redis的配置实现)

### Redisson

官方：[https://redisson.org](#)

Redisson是一个在Redis的基础上实现的Java驻内存数据网格（In-Memory Data Grid）。它不仅提供了一系列的分布式的Java常用对象，还提供了许多分布式服务，其中就包含了各种分布式锁的实现。

- 8. 分布式锁 (Lock) 和同步器 (Synchronizer)
  - 8.1. 可重入锁 (Reentrant Lock)
  - 8.2. 公平锁 (Fair Lock)
  - 8.3. 联锁 (MultiLock)
  - 8.4. 红锁 (RedLock)
  - 8.5. 读写锁 (ReadWriteLock)
  - 8.6. 信号量 (Semaphore)
  - 8.7. 可过期性信号量 (PermitExpirableSemaphore)
  - 8.8. 闭锁 (CountDownLatch)

官网地址：<https://redisson.org>

GitHub地址：<https://github.com/redisson/redisson>

## Redisson入门

1. 引入依赖：

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.13.6</version>
</dependency>
```

2. 配置Redisson客户端：

```
@Configuration
public class RedisConfig {
    @Bean
    public RedissonClient redissonClient() {
        // 配置类
        Config config = new Config();
        // 添加redis地址, 这里添加了单点的地址, 也可以使用config.useClusterServers() 添加集群地址
        config.useSingleServer().setAddress("redis://192.168.150.101:6379").setPassword("123321");
        // 创建客户端
        return Redisson.create(config);
    }
}
```

### 3. 使用Redisson的分布式锁

```

@Resource
private RedissonClient redissonClient;

@Test
void testRedisson() throws InterruptedException {
    // 获取锁(可重入), 指定锁的名称
    RLock lock = redissonClient.getLock("anyLock");
    // 尝试获取锁, 参数分别是: 获取锁的最大等待时间(期间会重试), 锁自动释放时间, 时间单位
    boolean isLock = lock.tryLock(1, 10, TimeUnit.SECONDS);
    // 判断释放获取成功
    if(isLock){
        try {
            System.out.println("执行业务");
        }finally {
            // 释放锁
            lock.unlock();
        }
    }
}

```

其实在创建所对象啊

### Redisson可重入锁原理

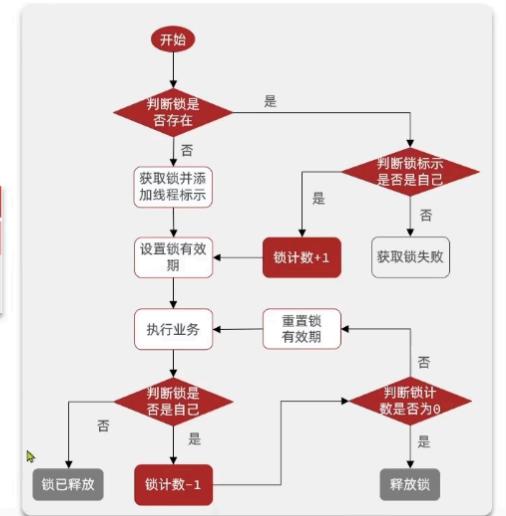
```

// 创建锁对象
RLock lock = redissonClient.getLock("lock");

@Test
void method1() {
    boolean isLock = lock.tryLock();
    if(!isLock){
        log.error("获取锁失败, 1");
        return;
    }
    try {
        log.info("获取锁成功, 1");
        method2();
    } finally {
        log.info("释放锁, 1");
        lock.unlock();
    }
}
void method2(){
    boolean isLock = lock.tryLock();
    if(!isLock){
        log.error("获取锁失败, 2");
        return;
    }
    try {
        log.info("获取锁成功, 2");
    } finally {
        log.info("释放锁, 2");
        lock.unlock();
    }
}

```

KEY	VALUE	
	field	value



方案	优点	缺点	适用场景
Redisson 可重入锁	1. 避免同一线程死锁 2. 自动续期 3. 高并发性能好	1. 依赖 Redis 可用性 2. 网络延迟敏感	分布式系统、高频短事务 (如电商库存扣减)
不可重入锁 (如 Redis SETNX)	1. 实现简单 2. 无计数器 开销	1. 同一线程嵌套调用 会死锁 2. 需手动续期	简单场景、无嵌套调用需求
数据库行锁 (如 SELECT FOR UPDATE)	1. 强一致性 2. 与事务集成方便	1. 性能低 (高并发瓶颈) 2. 不可重入	低频强一致性场景 (如财务对账)
ZooKeeper 分布式锁	1. 强一致性 (CP) 2. 原生支持可重入	1. 性能较低 2. 实现复杂	强一致性需求 (如配置管理、选主)

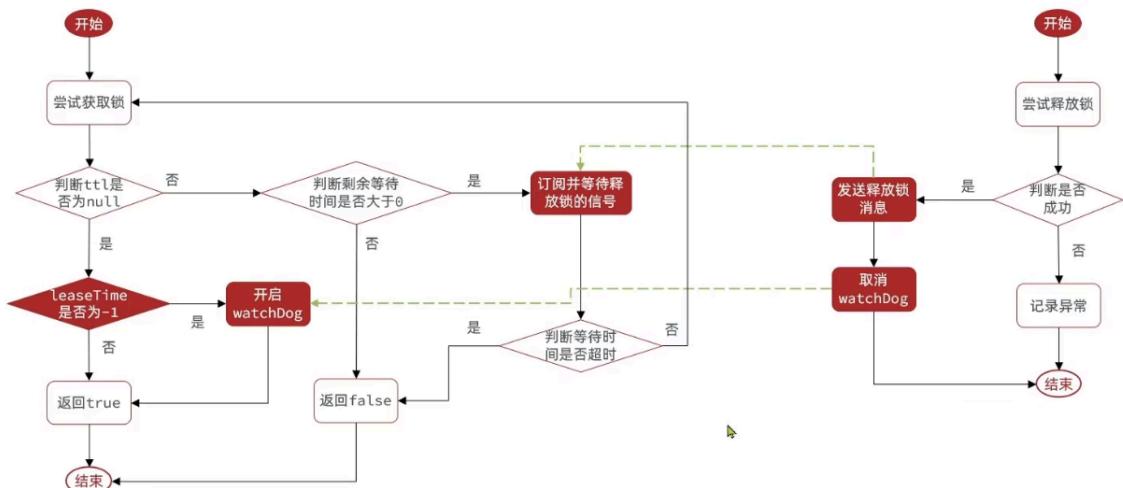
## (可重入锁的必要性:本质是解决重复调用的死锁问题)

```
public class LockTest {  
    static ReentrantLock lock = new ReentrantLock();  
  
    public void m1(){  
        lock.lock();  
        try {  
            m2();  
        }finally {  
            lock.unlock();  
        }  
    }  
  
    private void m2() {  
        //既然m1已经加锁了，为什么m2还需要加锁？不加锁一样能够保证m1调用m2的安全性  
        lock.lock();  
        try {  
            System.out.println("同步代码块");  
        }finally {  
            lock.unlock();  
        }  
    }  
}
```

在这段代码中，m1调用了m2，既然m1已经加锁了，为什么还要给m2加锁？

原因是 既然m2独立出来成了一个方法，你就没法保证你在调用m1的方法的同时，别的人不通过m1调用m2，而是直接调用m2。如果m2不加锁，这里就必然会出现线程安全+问题。

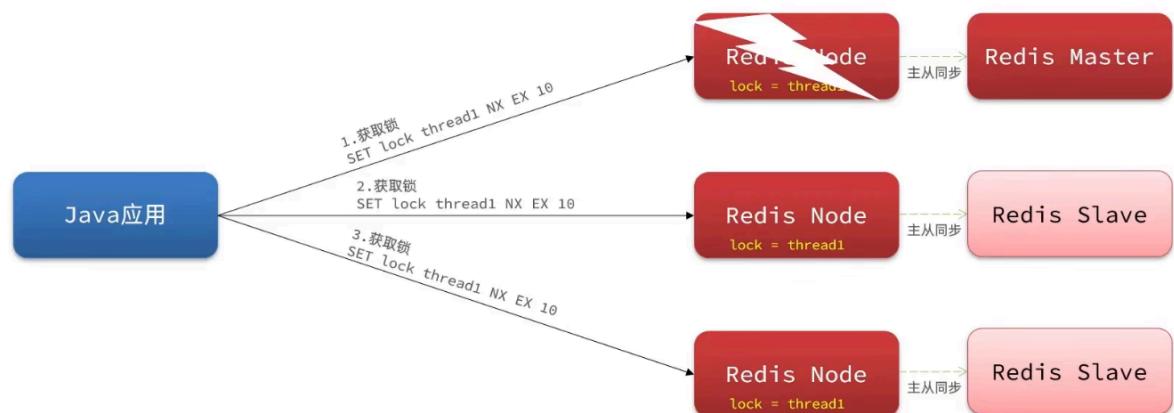
## Redisson分布式锁原理



## Redisson分布式锁原理：

- **可重入**: 利用hash结构记录线程id和重入次数
- **可重试**: 利用信号量和PubSub功能实现等待、唤醒，获取锁失败的重试机制
- **超时续约**: 利用watchDog，每隔一段时间（releaseTime / 3），重置超时时间

## Redisson分布式锁主从一致性问题



- 总结:

## 1) 不可重入Redis分布式锁:

◆ 原理：利用setnx的互斥性；利用ex避免死锁；释放锁时判断线程标示

◆ 缺陷：不可重入、无法重试、锁超时失效

## 2) 可重入的Redis分布式锁:

◆ 原理：利用hash结构，记录线程标示和重入次数；利用watchDog延续锁时间；利用信号量控制锁重试等待

◆ 缺陷：redis宕机引起锁失效问题

## 3) Redisson的multiLock:

◆ 原理：多个独立的Redis节点，必须在所有节点都获取重入锁，才算获取锁成功

◆ 缺陷：运维成本高、实现复杂

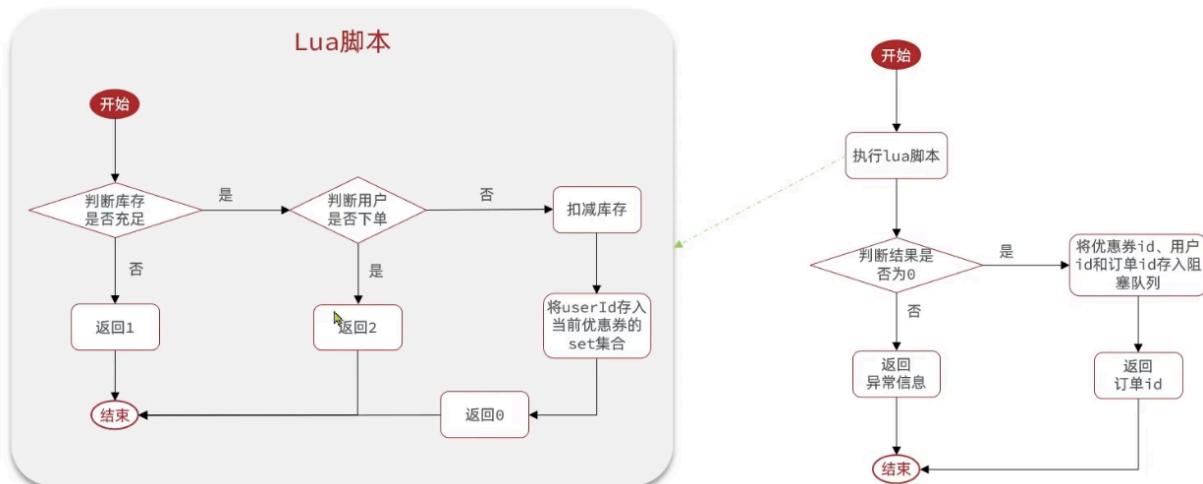
- 优化：

### Redis优化秒杀

KEY	VALUE
stock:vid:7	100

KEY	VALUE
order:vid:7	1 , 2 , 3 , 5 , 7 , 8



## 秒杀业务的优化思路是什么？

- ① 先利用Redis完成库存余量、一人一单判断，完成抢单业务
- ② 再将下单业务放入阻塞队列，利用独立线程异步下单

基于阻塞队列的异步秒杀存在哪些问题？

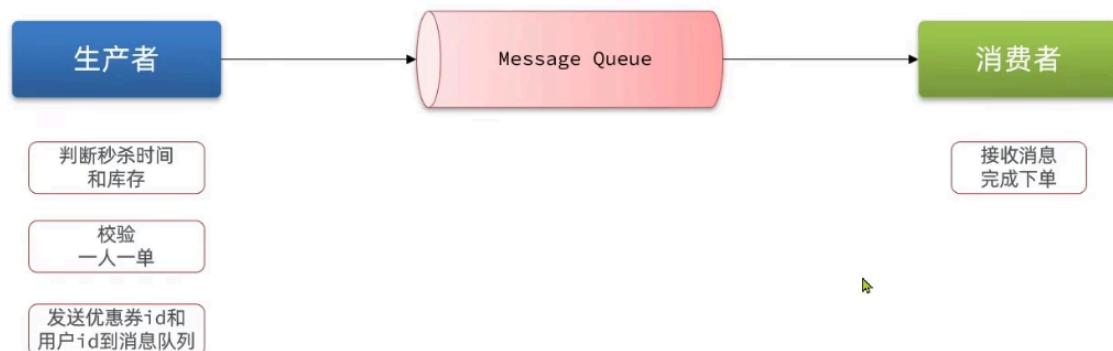
- 内存限制问题
- 数据安全问题

(引入消息队列,此处可以后联系到别的消息队列,如kafka,rocketMQ等)

### Redis消息队列实现异步秒杀

消息队列（Message Queue），字面意思就是存放消息的队列。最简单的消息队列模型包括3个角色：

- 消息队列：存储和管理消息，也被称为消息代理（Message Broker）
- 生产者：发送消息到消息队列
- 消费者：从消息队列获取消息并处理消息



## Redis消息队列实现异步秒杀

消息队列（Message Queue），字面意思就是存放消息的队列。最简单的消息队列模型包括3个角色：

- 消息队列：存储和管理消息，也被称为消息代理（Message Broker）
- 生产者：发送消息到消息队列
- 消费者：从消息队列获取消息并处理消息

Redis提供了三种不同的方式来实现消息队列：

- ◆ list结构：基于List结构模拟消息队列
- ◆ PubSub：基本的点对点消息模型
- ◆ Stream：比较完善的消息队列模型

## 2.9 Redis的三种消息队列(了解即可)：

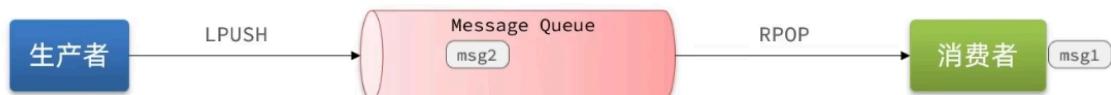
- 基于List:

### 基于List结构模拟消息队列

消息队列（Message Queue），字面意思就是存放消息的队列。而Redis的list数据结构是一个双向链表，很容易模拟出队列效果。

队列是入口和出口不在一边，因此我们可以利用：LPUSH 结合 RPOP、或者 RPUSH 结合 LPOP来实现。

不过要注意的是，当队列中没有消息时RPOP或LPOP操作会返回null，并不像JVM的阻塞队列那样会阻塞并等待消息。因此这里应该使用**BRPOP**或者**BLPOP**来实现阻塞效果。



# 基于List的消息队列有哪些优缺点？

## 优点：

- 利用Redis存储，不受限于JVM内存上限
- 基于Redis的持久化机制，数据安全性有保证
- 可以满足消息有序性

## 缺点：

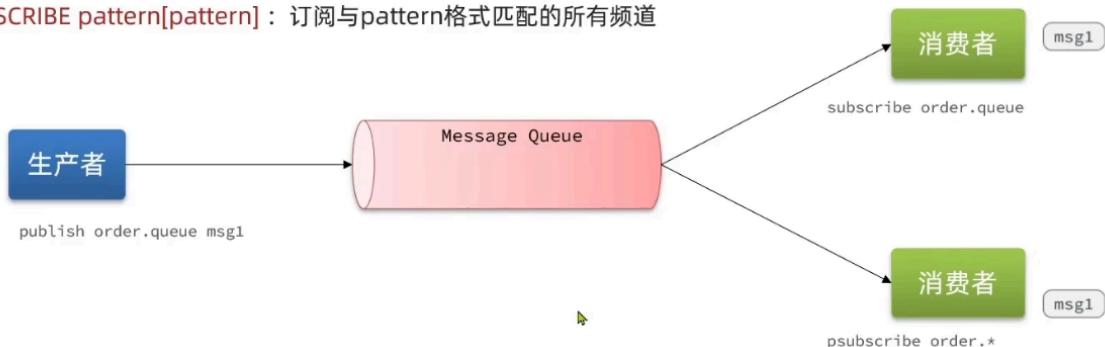
- 无法避免消息丢失
- 只支持单消费者

### • 基于PubSub:

#### 基于PubSub的消息队列

**PubSub（发布订阅）**是Redis2.0版本引入的消息传递模型。顾名思义，消费者可以订阅一个或多个channel，生产者向对应channel发送消息后，所有订阅者都能收到相关消息。

- **SUBSCRIBE channel [channel]**：订阅一个或多个频道
- **PUBLISH channel msg**：向一个频道发送消息
- **PSUBSCRIBE pattern[pattern]**：订阅与pattern格式匹配的所有频道



# 基于PubSub的消息队列有哪些优缺点？

## 优点：

- 采用发布订阅模型，支持多生产、多消费

## 缺点：

- 不支持数据持久化
- 无法避免消息丢失
- 消息堆积有上限，超出时数据丢失

- 基于Stream单消费模式：

### 基于Stream的消息队列

Stream 是 Redis 5.0 引入的一种新数据类型，可以实现一个功能非常完善的消息队列。

发送消息的命令：



例如：

```
# 创建名为 users 的队列，并向其中发送一个消息，内容是：{name=jack,age=21}，并且使用Redis自动生成ID
127.0.0.1:6379> XADD users * name jack age 21
"1644805700523-0"
```

### 基于Stream的消息队列-XREAD

读取消息的方式之一：XREAD



XREAD阻塞方式，读取最新的消息：

```
127.0.0.1:6379> XREAD COUNT 1 BLOCK 1000 STREAMS users $  
(nil)  
(1.07s)
```

在业务开发中，我们可以循环的调用XREAD阻塞方式来查询最新消息，从而实现持续监听队列的效果，伪代码如下：

```
1 while(true){  
2     // 尝试读取队列中的消息，最多阻塞2秒  
3     Object msg = redis.execute("XREAD COUNT 1 BLOCK 2000 STREAMS users $");  
4     if(msg == null){  
5         continue;  
6     }  
7     // 处理消息  
8     handleMessage(msg);  
9 }
```

注意

当我们指定起始ID为\$时，代表读取最新的消息，如果我们处理一条消息的过程中，又有超过1条以上的消息到达队列，则下次获取时也只能获取到最新的一条，会出现漏读消息的问题。

## STREAM类型消息队列的XREAD命令特点：

- 消息可回溯
- 一个消息可以被多个消费者读取
- 可以阻塞读取
- 有消息漏读的风险
- 基于Stream消费者组模式

## 基于Stream的消息队列-消费者组

**消费者组（Consumer Group）**：将多个消费者划分到一个组中，监听同一个队列。具备下列特点：

01

### 消息分流

队列中的消息会分流给组内的不同消费者，而不是重复消费，从而加快消息处理的速度

02

### 消息标示

消费者组会维护一个标示，记录最后一个被处理的消息，哪怕消费者宕机重启，还会从标示之后读取消息。确保每一个消息都会被消费

03

### 消息确认

消费者获取消息后，消息处于 pending 状态，并存入一个 pending-list。当处理完成后需要通过 XACK 来确认消息，标记消息为已处理，才会从 pending-list 移除。

## 基于Stream的消息队列-消费者组

创建消费者组：

```
XGROUP CREATE key groupName ID [MKSTREAM]
```

- key：队列名称
- groupName：消费者组名称
- ID：起始ID标示，\$代表队列中最后一个消息，0则代表队列中第一个消息
- MKSTREAM：队列不存在时自动创建队列

其它常见命令：

```
# 删除指定的消费者组
XGROUP DESTORY key groupName

# 给指定的消费者组添加消费者
XGROUP CREATECONSUMER key groupname consumername

# 删除消费者组中的指定消费者
XGROUP DELCONSUMER key groupname consumername
```

## 基于Stream的消息队列-消费者组

从消费者组读取消息：

```
XREADGROUP GROUP group consumer [COUNT count] [BLOCK milliseconds] [NOACK] STREAMS  
key [key ...] ID [ID ...]
```

- **group**: 消费组名称
- **consumer**: 消费者名称, 如果消费者不存在, 会自动创建一个消费者
- **count**: 本次查询的最大数量
- **BLOCK milliseconds**: 当没有消息时最长等待时间
- **NOACK**: 无需手动ACK, 获取到消息后自动确认
- **STREAMS key**: 指定队列名称
- **ID**: 获取消息的起始ID:
  - ">": 从下一个未消费的消息开始
  - 其它: 根据指定id从pending-list中获取已消费但未确认的消息, 例如0, 是从pending-list中的第一个消息开始

## STREAM类型消息队列的XREADGROUP命令特点：

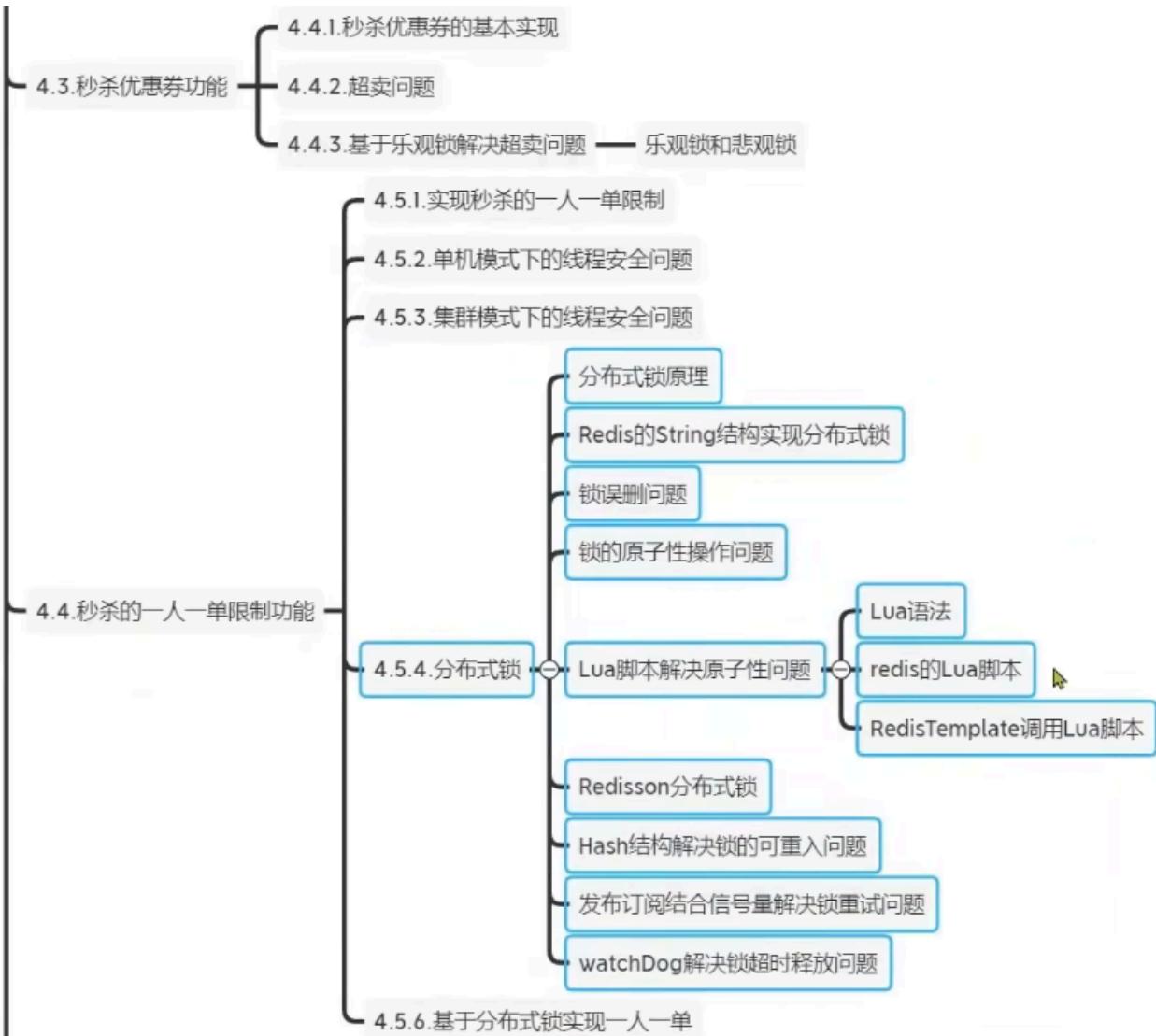
- 消息可回溯
- 可以多消费者争抢消息, 加快消费速度
- 可以阻塞读取
- 没有消息漏读的风险
- 有消息确认机制, 保证消息至少被消费一次
- 总结:

	List	PubSub	Stream
<b>消息持久化</b>	支持	不支持	支持
<b>阻塞读取</b>	支持	支持	支持
<b>消息堆积处理</b>	受限于内存空间，可以利用多消费者加快处理	受限于消费者缓冲区	受限于队列长度，可以利用消费者组提高消费速度，减少堆积
<b>消息确认机制</b>	不支持	不支持	支持
<b>消息回溯</b>	不支持	不支持	支持

## 基于Redis的Stream结构作为消息队列，实现异步秒杀下单

需求：

- ① 创建一个Stream类型的消息队列，名为stream.orders
- ② 修改之前的秒杀下单Lua脚本，在认定有抢购资格后，直接向stream.orders中添加消息，内容包含voucherId、userId、orderId
- ③ 项目启动时，开启一个线程任务，尝试获取stream.orders中的消息，完成下单



## 2.10 点赞排行榜:

### 实现查询点赞排行榜的接口

需求：按照点赞时间先后排序，返回Top5的用户

	List	Set	SortedSet
排序方式	按添加顺序排序	无法排序	根据score值排序
唯一性	不唯一	唯一	唯一
查找方式	按索引查找或首尾查找	根据元素查找	根据元素查找

Set使用opsForSet

SortedSet使用opsForZSet

## 2.11 关注推送:

### Feed流的模式

Feed流产品有两种常见模式：

- **Timeline**: 不做内容筛选，简单的按照内容发布时间排序，常用于好友或关注。例如朋友圈
  - 优点：信息全面，不会有缺失。并且实现也相对简单
  - 缺点：信息噪音较多，用户不一定感兴趣，内容获取效率低
- **智能排序**: 利用智能算法屏蔽掉违规的、用户不感兴趣的内容。推送用户感兴趣信息来吸引用户
  - 优点：投喂用户感兴趣信息，用户粘度很高，容易沉迷
  - 缺点：如果算法不精准，可能起到反作用

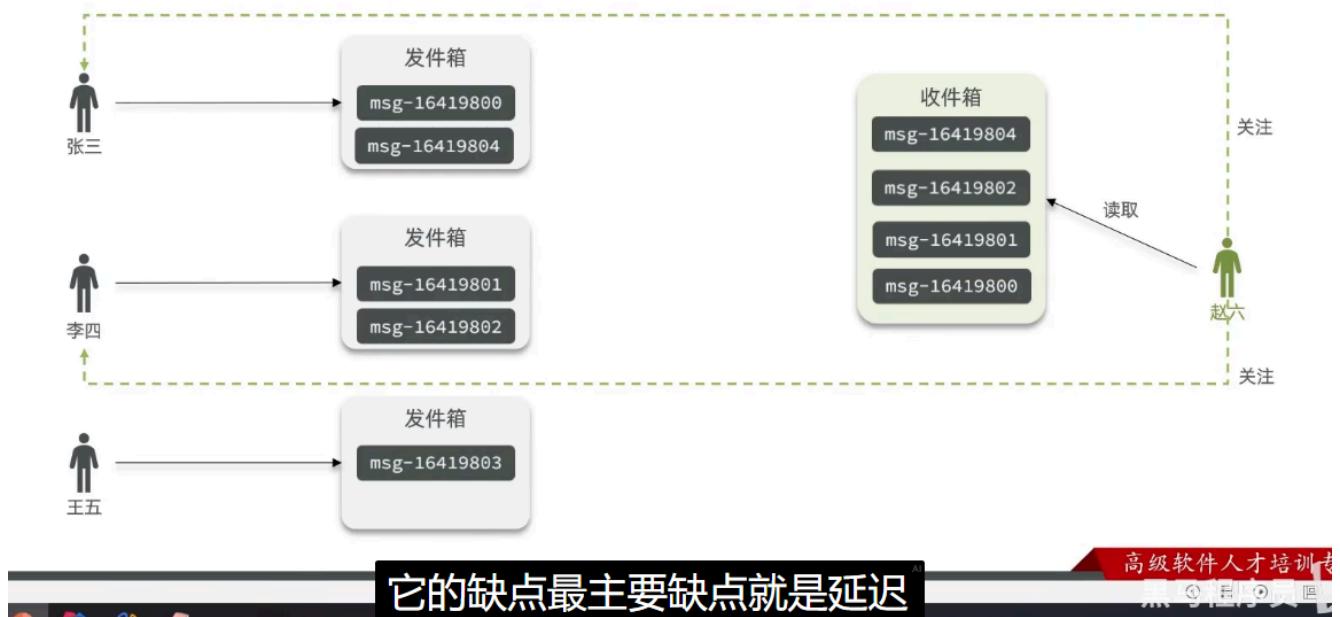
本例中的个人页面，是基于关注的好友来做Feed流，因此采用Timeline的模式。该模式的实现方案有三种：

- ① 拉模式
- ② 推模式
- ③ 推拉结合

**推拉结合**

## Feed流的实现方案1

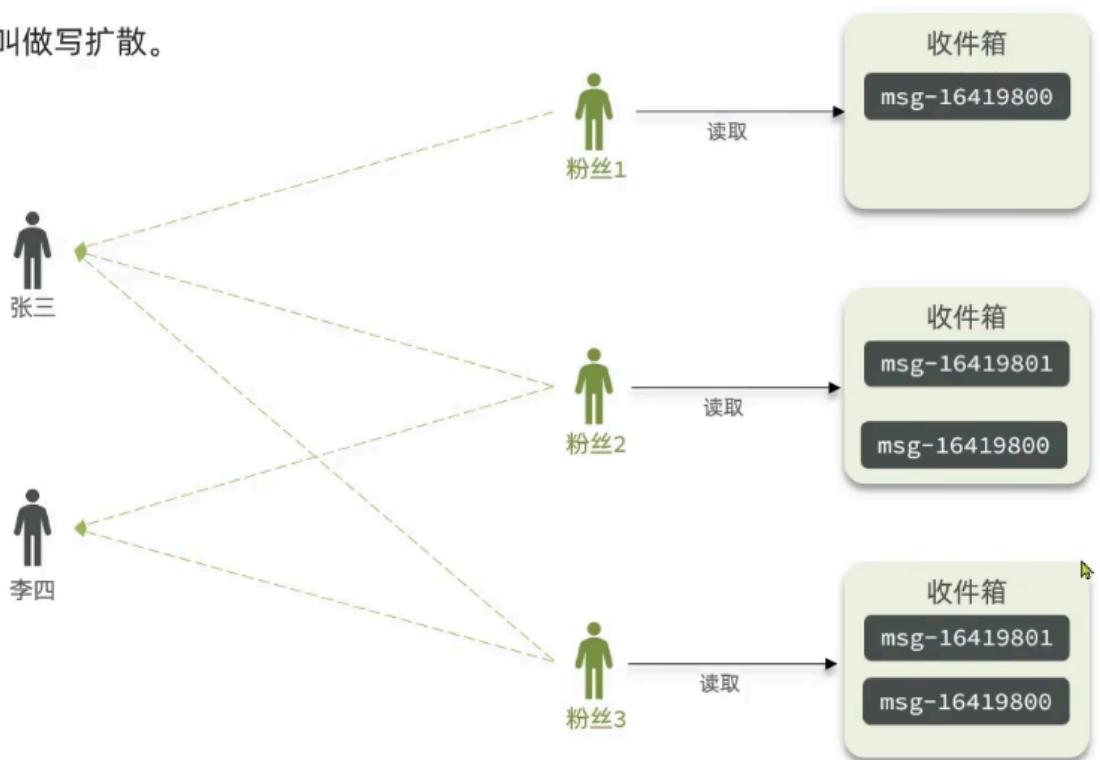
拉模式：也叫做读扩散。



## Feed流的实现方案2

高内存

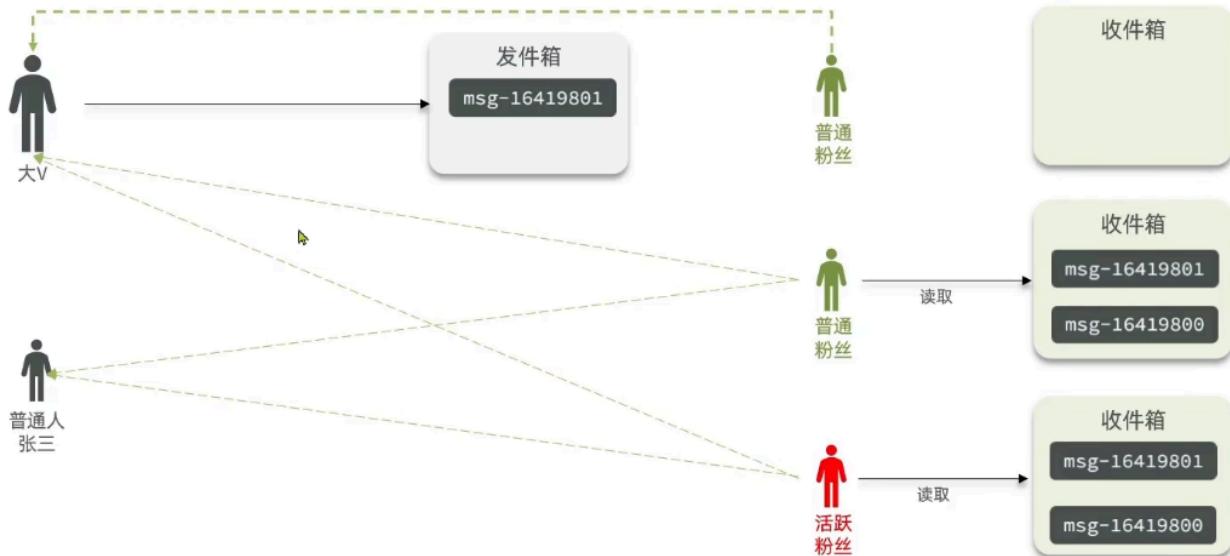
推模式：也叫做写扩散。



它的内存占用会比较高

## Feed流的实现方案3

**推拉结合模式：**也叫做读写混合，兼具推和拉两种模式的优点。



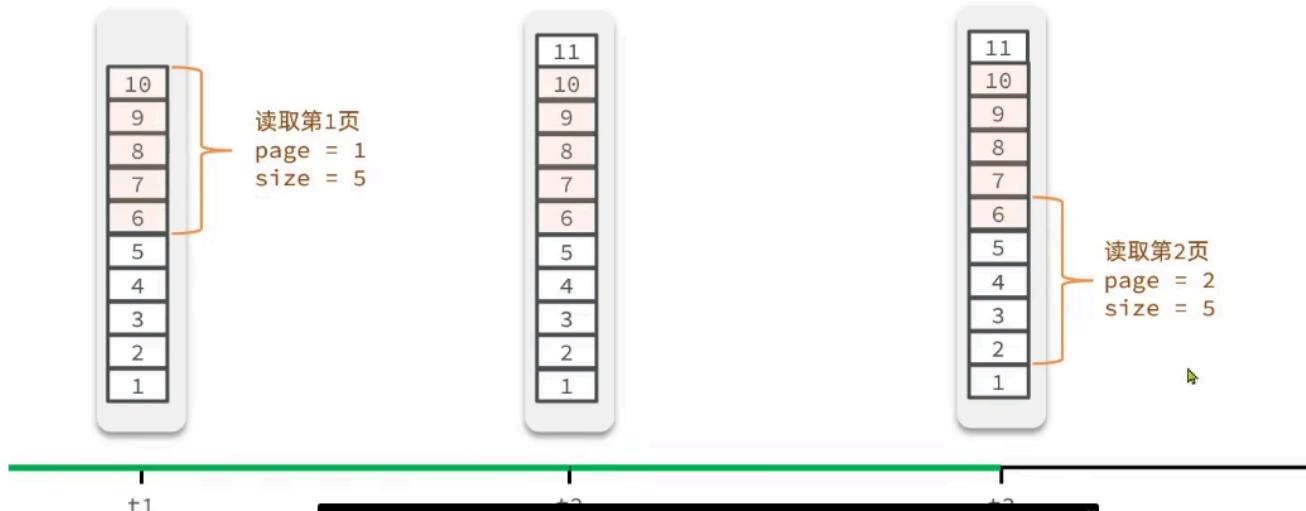
## Feed流的实现方案

241.0W

	拉模式	推模式	推拉结合
写比例	低	高	中
读比例	高	低	中
用户读取延迟	高	低	低
实现难度	复杂	简单	很复杂
使用场景	很少使用	用户量少、没有大V	过千万的用户量，有大V

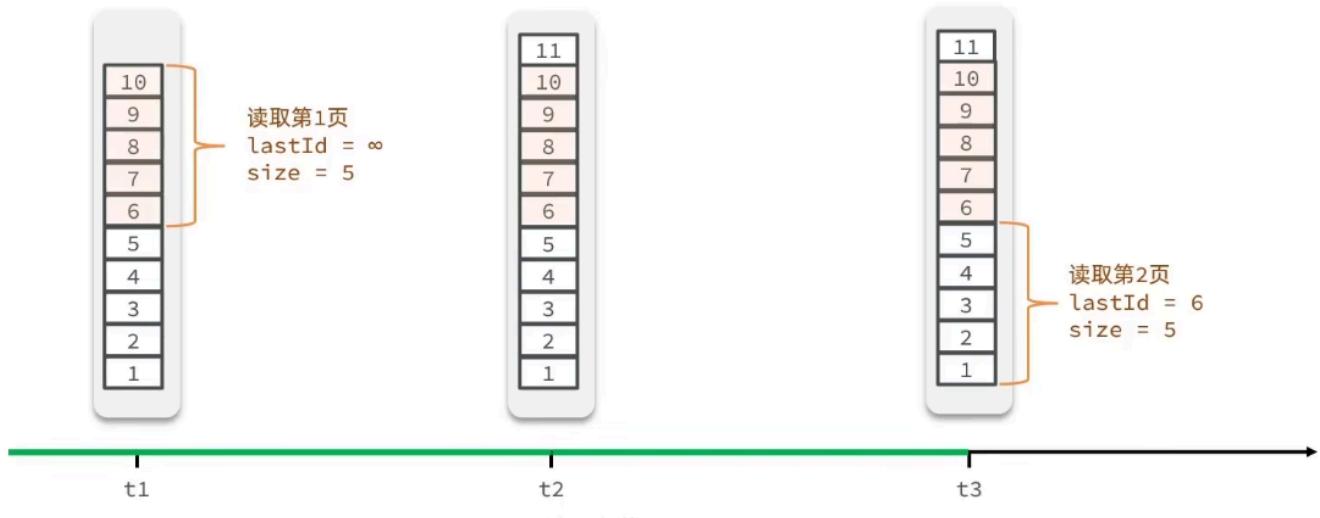
## Feed流的分页问题

Feed流中的数据会不断更新，所以数据的角标也在变化，因此不能采用传统的分页模式。



## Feed流的滚动分页

Feed流中的数据会不断更新，所以数据的角标也在变化，因此不能采用传统的分页模式。



(数据有变化的情况下避免用list,因为list只支持角标查询,会无法支持滚动分页,因此使用SortedSet,通过score当时间戳来实现滚动分页)

## 2.12 附近商家:

### GEO数据结构

GEO就是Geolocation的简写形式，代表地理坐标。Redis在3.2版本中加入了对GEO的支持，允许存储地理坐标信息，帮助我们根据经纬度来检索数据。常见的命令有：

GEOADD: 添加一个地理空间信息，包含：经度（longitude）、纬度（latitude）、值（member）

GEODIST: 计算指定的两个点之间的距离并返回

GEOHASH: 将指定member的坐标转为hash字符串形式并返回

GEOPOS: 返回指定member的坐标

GEORADIUS: 指定圆心、半径，找到该圆内包含的所有member，并按照与圆心之间的距离排序后返回。6.2以后已废弃

GEOSEARCH: 在指定范围内搜索member，并按照与指定点之间的距离排序后返回。范围可以是圆形或矩形。6.2.新功能

GEOSEARCHSTORE: 与GEOSEARCH功能一致，不过可以把结果存储到一个指定的key。6.2.新功能

### 附近商户搜索

在首页中点击某个频道，即可看到频道下的商户：

The screenshot illustrates the search process for nearby merchants. On the left, the main homepage shows various service categories like Beauty, KTV, etc. In the center, a detailed view of the 'Food' category lists several restaurants with their names, ratings, and addresses. On the right, a screenshot of a browser's developer tools shows the API request details for this search, including the URL, method, status code, and response body.

	说明
请求方式	GET
请求路径	/shop/of/type
请求参数	typeId: 商户类型 current: 页码, 滚动查询 x: 经度 y: 纬度
返回值	List<Shop>: 符合要求的商户信息

## 附近商户搜索

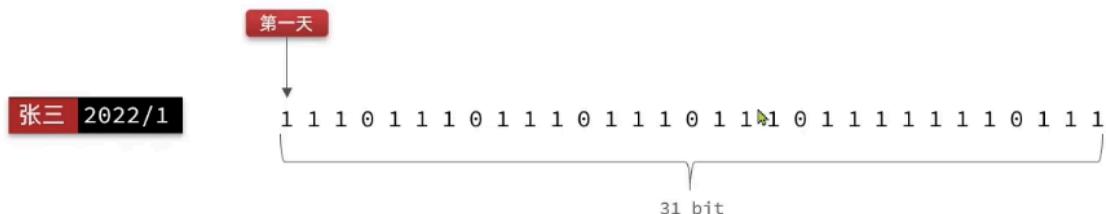
按照商户类型做分组，类型相同的商户作为同一组，以typeld为key存入同一个GEO集合中即可

Key	Value	Score
shop:geo:美食	 海底捞火锅	4069152240174578
	 新白鹿	4069879450313142
shop:geo:KTV	 星聚会 KTV	4069885469876391
	 KALEDI KTV 开乐迪	4069885424176331

## 2.13 签到:

### BitMap用法

我们按月来统计用户签到信息，签到记录为1，未签到则记录为0.



把每一个bit位对应当月的每一天，形成了映射关系。用0和1标示业务状态，这种思路就称为**位图 (BitMap)**。

Redis中是利用string类型数据结构实现**BitMap**，因此最大上限是512M，转换为bit则是  $2^{32}$ 个bit位。

## BitMap用法

Redis中是利用string类型数据结构实现BitMap，因此最大上限是512M，转换为bit则是  $2^{32}$ 个bit位。

BitMap的操作命令有：

[SETBIT](#)：向指定位置（offset）存入一个0或1

[GETBIT](#)：获取指定位置（offset）的bit值

[BITCOUNT](#)：统计BitMap中值为1的bit位的数量

[BITFIELD](#)：操作（查询、修改、自增）BitMap中bit数组中的指定位置（offset）的值

[BITFIELD\\_RO](#)：获取BitMap中bit数组，并以十进制形式返回

[BITOP](#)：将多个BitMap的结果做位运算（与、或、异或）

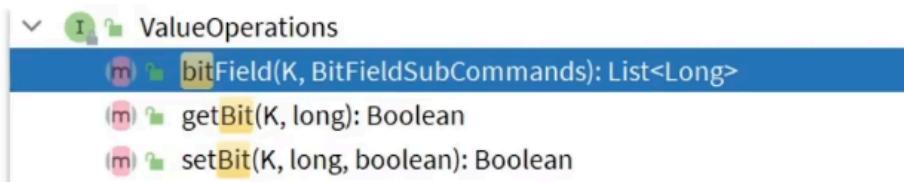
[BITPOS](#)：查找bit数组中指定范围内第一个0或1出现的位置

## 签到功能

需求：实现签到接口，将当前用户当天签到信息保存到Redis中

说明	
请求方式	Post
请求路径	/user/sign
请求参数	无
返回值	无

提示：因为BitMap底层是基于String数据结构，因此其操作也都封装在字符串相关操作中了。



**问题1：**什么叫做连续签到天数？

从最后一次签到开始**向前统计**，直到遇到**第一次未签到为止**，计算总的签到次数，就是连续签到天数。

```
1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1
```

**问题2：**如何得到本月到今天为止的所有签到数据？

BITFIELD key GET u[dayOfMonth] 0

```
1 0 1 1 1
1
```

**问题3：**如何从后向前遍历每个bit位？

与 1 做与运算，就能得到最后一个bit位。

随后右移1位，下一个bit位就成为了最后一个bit位。

## 2.14 UV统计：

### HyperLogLog用法

首先我们搞懂两个概念：

- **UV：**全称**Unique Visitor**，也叫独立访客量，是指通过互联网访问、浏览这个网页的自然人。1天内同一个用户多次访问该网站，只记录1次。
- **PV：**全称**Page View**，也叫页面访问量或点击量，用户每访问网站的一个页面，记录1次PV，用户多次打开页面，则记录多次PV。往往用来衡量网站的流量。

UV统计在服务端做会比较麻烦，因为要判断该用户是否已经统计过了，需要将统计过的用户信息保存。但是如果每个访问的用户都保存到Redis中，数据量会非常恐怖。

## HyperLogLog用法

Hyperloglog(HLL)是从Loglog算法派生的概率算法，用于确定非常大的集合的基数，而不需要存储其所有值。相关算法原理大家可以参考：<https://juejin.cn/post/6844903785744056333#heading-0>

Redis中的HLL是基于string结构实现的，单个HLL的内存永远小于16kb，内存占用低的令人发指！作为代价，其测量结果是概率性的，有小于0.81%的误差。不过对于UV统计来说，这完全可以忽略。

```
PFADD key element [element ...]
summary: Adds the specified elements to the specified HyperLogLog.
since: 2.8.9

PFCOUNT key [key ...]
summary: Return the approximated cardinality of the set(s) observed by
since: 2.8.9

PFMERGE destkey sourcekey [sourcekey ...]
summary: Merge N different HyperLogLogs into a single one.
since: 2.8.9
```

# 九. 算法

## 1. 贪心算法：

思路：通过局部最优达到整体最优

问题：极端用例（顺序数组、几乎有序的数组）使得排序耗时增加

原因：递归树高度增加，时间复杂度成为  $O(N^2)$

解决方法：随机选择切分元素 pivot

又出现了新问题：随机选择 pivot 对数组中有大量重复元素的用例失效

解决方法 1：把等于 pivot 的元素「平均地」分到数组的两侧

解决方法 2：把等于 pivot 的元素挤到中间，递归区间可以大大减少

解决方法一就是二路快排

解决方法二就是三路快排

## 2. 快速排序：

### 2.1 一般方法：

```
class Solution {  
    public int[] sortedSquares(int[] nums) {  
        for(int i = 0;i < nums.length;i++){  
            nums[i] = nums[i]*nums[i];  
        }  
        quickSort(nums,0,nums.length-1);  
        return nums;  
    }  
    //快速排序递归  
    public void quickSort(int[] nums ,int left ,int right){  
        if(left>=right){  
            return;  
        }  
        int key = patition(nums,left,right);  
        quickSort(nums,left,key-1);  
        quickSort(nums,key+1,right);  
    }  
  
    //快速排序实现  
    public int patition(int[] nums,int left,int right){  
        int med = midNum(left,(left+right)/2,right);  
        swap(nums,med,left);  
        int i = left;  
        int j = right;  
        while(i<j){  
            while(i<j && nums[j] >= nums[left])  
                j--;  
            while(i<j && nums[i] <= nums[left])  
                i++;  
            swap(nums,i,j);  
        }  
        swap(nums,i,left);  
        return i;  
    }  
  
    //交换数组两索引的值  
    public void swap(int[] nums,int i,int j){  
        int temp = nums[i];
```

```
    nums[i] = nums[j];
    nums[j] = temp;
}

//随机选择基准值(中位数方法)
public int midNum(int left, int mid, int right){
    if((left <= mid && mid <= right) || (right <= mid && left <= mid)){
        return mid;
    }
    if((left <= right && mid >= right) || (right <= mid && left <= right)){
        return right;
    }
    return mid;
}
}
```

## 2.2 二路快排:

使得与基准数相同的数平均分到两边

```
class Solution {
    public int[] sortedSquares(int[] nums) {
        for(int i = 0;i < nums.length;i++){
            nums[i] = nums[i]*nums[i];
        }
        quickSort(nums,0,nums.length-1);
        return nums;
    }
    //快速排序递归
    public void quickSort(int[] nums ,int left ,int right){
        if(left>=right){
            return;
        }
        int key = patition(nums,left,right);
        quickSort(nums,left,key-1);
        quickSort(nums,key+1,right);
    }

    //快速排序实现
    public int patition(int[] nums,int left,int right){
        int med = midNum(left,(left+right)/2,right);
        swap(nums,med,left);

        //修改点1
        int i = left+1;
        int j = right;

        //修改点2,变为j<i时才退出循环,且寻找严格大于/小于基准的数
        while(i <= j){
            while(i<=j && nums[j] > nums[left])
                j--;
            while(i<=j && nums[i] < nums[left])
                i++;
            //修改点3,允许相等时交换
            if(i<=j){
                swap(nums,i,j);
                i++;
                j--;
            }
        }
        //修改点4,改为与j交换
        swap(nums,j,left);
        return j;
    }
}
```

```
}

//交换数组两索引的值
public void swap(int[] nums,int i,int j){
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}

//随机选择基准值(中位数方法)
public int midNum(int left, int mid, int right){
    if((left <= mid && mid <= right) || (right <= mid && left <= mid)){
        return mid;
    }
    if((left <= right && mid >= right) || (right <= mid && left <= right)){
        return right;
    }
    return mid;
}
}
```

## 2.3 三路快排:

分为大于/等于/小于基准三个区块

```

public int[] sortedSquares(int[] nums) {
    ...
    quickSort(nums, 0, nums.length - 1);
    return nums;
}

public void quickSort(int[] nums, int left, int right) {
    if (left >= right) return;
    // 调用三路快排
    int[] bounds = partition3Way(nums, left, right);
    quickSort(nums, left, bounds[0]);
    quickSort(nums, bounds[1], right);
}

// 三路分区方法
private int[] partition3Way(int[] nums, int left, int right) {
    int med = midNum(left, (left + right) / 2, right);
    swap(nums, med, left);
    int pivot = nums[left];

    int lt = left;      // 小于区的右边界,初始无小于,在最左边元素索引+1的位子
    int gt = right;    // 大于区的左边界,初始无大于,在最右边元素索引-1的位子
    int i = left + 1; // 当前指针

    while (i <= gt) {
        if (nums[i] < pivot) {
            swap(nums, i++, lt++);
        } else if (nums[i] > pivot) {
            swap(nums, i, gt--);
        } else {
            i++;
        }
    }
    return new int[]{lt - 1, gt + 1}; // 返回两个分界点
}

```

### 3. 动态规划:

核心思路:分解问题,通过子问题来求解原问题

- 子问题相互依赖,分解过程会出现重叠子问题

- 最优子结构:原问题最优解由子问题最优解所得
- **无后效性(适合动态规划的判断条件):**当前问题的状态后续发展仅与当前状态相关,与以前状态无关(可能需要拓展当前状态的包含内容)

## 3.1 普通爬楼梯

理解：在优化空间的情况下有两种存储方式：

- 通过轮转，以此像走路一样交替向前，交替保留当前值 //不推荐，JVM会隐性调用操作数栈储存中间结果
- 通过活塞，一个空间只保留当前值，一个只保留上一个值(消耗空间为固定值，内存反而更小)

## 3.2 0-1背包问题：

每个物品只能选取一次，求最大价值

dp初始化思路:

- 放物品i：背包空出物品i的容量后，背包容量为 $j - weight[i]$ ， $dp[i - 1][j - weight[i]]$  为背包容量为 $j - weight[i]$ 且不放物品i的最大价值，那么 $dp[i - 1][j - weight[i]] + value[i]$  (物品i的价值)，就是背包放物品i得到的最大价值

递归公式：  $dp[i][j] = \max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);$

3. dp数组如何初始化

关于初始化，一定要和dp数组的定义吻合，否则到递推公式的时候就会越来越乱。

首先从 $dp[i][j]$ 的定义出发，如果背包容量j为0的话，即 $dp[i][0]$ ，无论是选取哪些物品，背包价值总和一定为0。如图：

dp[i][j]		背包重量j:				
		0	1	2	3	4
物品0:	0					
	0					
	0					
物品1:						
物品2:						



### 3.3 完全背包问题(近似于二维爬楼梯)

- 物品可重复选取

在 01背包理论基础（二维数组） 中，背包先空留出物品1的容量，此时容量为1，只考虑放物品0的最大价值是  $dp[0][1]$ ，因为01背包每个物品只有一个，既然空出物品1，那背包中也不会再有物品1！

而在完全背包中，物品是可以放无限个，所以 即使空出物品1空间重量，那背包中也可能还有物品1，所以此时我们依然考虑放 物品0 和 物品1 的最大价值即：  $dp[1][1]$ ， 而不是  $dp[0][1]$

所以 放物品1 的情况 =  $dp[1][1] + \text{物品1 的价值}$ ，推导方向如图：

		背包重量:	0	1	2	3	4
物品0:		0	15	30	45	60	
物品1:		0	15	30	45	60	
物品2:							

(注意上图和 01背包理论基础（二维数组） 中的区别，对于理解完全背包很重要)

两种情况，分别是放物品1 和 不放物品1，我们要取最大值（毕竟求的是最大价值）

## 代码随想录

物品0:	0	15	30	45	60
物品1:	0	15	30	45	60
物品2:					

(注意上图和 01背包理论基础（二维数组） 中的区别，对于理解完全背包很重要)

两种情况，分别是放物品1 和 不放物品1，我们要取最大值（毕竟求的是最大价值）

```
dp[1][4] = max(dp[0][4], dp[1][1] + 物品1 的价值)
```

以上过程，抽象化如下：

- **不放物品i**: 背包容量为j，里面不放物品i的最大价值是 $dp[i - 1][j]$ 。
- **放物品i**: 背包空出物品i的容量后，背包容量为 $j - weight[i]$ ， $dp[i][j - weight[i]]$  为背包容量为 $j - weight[i]$ 且不放物品i的最大价值，那么 $dp[i][j - weight[i]] + value[i]$  （物品i的价值），就是背包放物品i得到的最大价值

递推公式：  $dp[i][j] = \max(dp[i - 1][j], dp[i][j - weight[i]] + value[i]);$

(注意，完全背包二维dp数组 和 01背包二维dp数组 递推公式区别，01背包中是  $dp[i - 1][j - weight[i]] + value[i]$  )

- 个人理解:对该例子而言：
  - 在没有空出物品1的容量之前是不会产生变化或者可能性的
  - 所以必须要 索引减去一个物品1的容量的位置，以此来看其上一次产生变化所选择的最大值
  - 将该最大值与当前情况做对比,从而获取最优解
- 遍历顺序：
  - 如果求组合数就是外层for循环遍历物品，内层for遍历背包(**物品顺序固定**)
  - 如果求排列数就是外层for遍历背包，内层for循环遍历物品(**物品顺序可变**)