

# 1.java介绍

2021年7月26日 11:37

\*常用的DOS命令

--dir:列出当前目录下的文件以及文件夹

--md:创建目录

--rd:删除目录

--cd:进入指定目录

--cd..退回到上一级目录

--cd/:退回到根目录

--del:删除文件

--exit:退出dos命令行

使用cmd搜索命令行指令框

\*java语言的特点

--面向对象性

两个要素：类、对象

三个特征：封装、继承、多态

--健壮性

去除指针、垃圾自动回收

--跨平台性

不同系统提供不同JVM，使其可跨平台

--JDK(java Development Kit java开发工具包)包含 (JRE)

--JRE=JVM+JavaSE核心类库

--下载JDK即可

D:\app\javatools\jdk1.8.0\_301

目录下

bin开发常用指令集

include头文件

jre运行环境

lib jar包

src.zip 一些类库

--配置环境变量

为了在任意文件下都可以运行java开发工具

右键计算机点击下方属性，打开系统高级属性

path:windows系统执行命令时要搜寻的路径

可以把文件路径直接放到path中

也可以

D:\app\javatools\jdk1.8.0\_301\bin

将D:\app\javatools\jdk1.8.0\_301赋值给JAVA\_HOME变量

%JAVA\_HOME%\bin加入path

(尽量这样配，之后要使用)

## 2.java语言概述

2021年7月26日 17:06

.java文件---编译 (javac.exe) --->.class文件---运行(java.exe)-->结果  
源文件 字节码文件

```
public class HelloChina{  
    public static void main(String[] args){  
        System.out.println("HelloWorld!");  
    }  
}
```

--编译格式：javac 文件名.java

--运行格式：java 类名

-- (在一个java源文件中可以声明多个class，但是只能一个类声明为public且要求声明为public的类名与源文件名相同)

--程序的入口是main()方法，格式是固定的

--常用输出语句

```
System.out.println("HelloWorld!"); //先输出后换行
```

```
System.out.print("HelloWorld!"); //只输出不换行
```

--编译后，会生成多个字节码文件，字节码文件名与java源文件中的类名相同

三种注释

//单行注释

/\*多行注释\*/

--不能嵌套使用

/\*\*文档注释\*/

特点：注释内容可以被JDK提供的工具javadoc所解析，生成一套以文件形式体现的该程序的说明文档。

Java API文档

--API (Application Programming Interface,应用程序编程接口)

是Java提供的基本编程接口

--Java语言提供了大量的基础的类和相应的API文档，

告诉我们如何使用类和其中的方法

# 3.Java基本语法-命名

2021年7月28日 11:42

## 1.\*关键字与保留字

定义：被Java语言赋予特殊含义的字符串

特点：关键字字母都是小写

\*保留字：现在版本尚未使用，以后可能使用

## 2.标识符

字母、数字、\_、\$

--不能有空格，不能以数字开头

--严格区分大小写

## 3.命名规范

--包名：多单词组成时所有都小写：xxxxyy

--类名、接口名：大驼峰法

--变量名、方法名：小驼峰法

--常量名：所有字母都大写，多单词\_连接，AAA\_BBB

注意：尽可能见名知意

# 4. Java基本语法-变量类型

2021年7月28日 12:22

\*数据类型

--基本数据类型

整数类型byte,short,int,long

浮点类型float,double

字符型char

布尔型boolean

--引用数据类型

类class

接口interface

数组array

整数类型

--java各整数类型有固定的表数范围和字段长度，不受OS影响

--java整形常量默认为int型，声明long型需要后面加L (l)

--通常声明为int，除非数太大

浮点型

--java的浮点型常量默认为double型

声明float型常量，后面需要加F (f)

--单精度float (4字节)

--双精度double (8字节) \*常用

字符型：char(1字符=2字节)

char c1='a';--只能写一个

char c2='\n';--转义字符

char c3='\\u0123';--unicode编码

乱码问题

使用命令行读文件时，命令行默认使用GBK编码方式 (ANSI)

如果文件使用utf-8会中文乱码

布尔值boolean

boolean b1=true;

true

false

# 5.java基本语法-运算规则

2021年7月28日 17:25

自动类型提升

范围小的与大的运算，提升为范围大的

byte、char、short、int-->long-->float-->double

特别：当byte,char,short进行运算时都会转换为int型

包括本身（可以避免溢出）

**大类型赋值给小类型出错**

强制类型转换

需要使用强制转换符（），可能导致精度损失。

int i=(int)12.9;//截断操作

## 6.java基本语法-String

2021年7月28日 18:03

声明String类型变量时，使用""

```
String s1="Hello World";
```

String可以和8种基本数据类型变量做运算，且只能连接运算，仍为String

# 7.java基本语法-运算符

2021年7月28日 19:42

## 算术运算符

+、-、\*、/、%、++、--

自增，自减不会改变本身变量的数据类型。

```
short a=10;  
a=a+10-->编译不通过  
a++-->short
```

## 赋值运算符

=  
+=, -=, \*=, /=, %= (不会改变变量数据类型)

## 比较运算符结果为boolean

==, !=, <,>,<=,>=, instanceof

## 逻辑运算符

& (逻辑与) , | (逻辑或) , ! (逻辑非)  
&& (短路与) , || (短路或) , ^ (逻辑异或)

&和&&

--运算结果相同

--&无短路, &&有短路

--|和||类似

--优先使用&&和||

## 位运算符

<<	左移每移一位乘2 (补0)
>>	右移每移一位/2 (补1)
>>>	无符号右移

&	按位与
	按位或
^	按位异或
~	按位取反

--操作的都是整型数据

## 三元运算符 (条件运算符)

(条件表达式) ? 表达式1: 表达式2;

可以使用if-else时优先用三元运算符，执行效率高

编译时表达式1、2，类型会统一

## 优先级



运算符的优先级（从高到低）

优先级	描述	运算符
1	括号	() []
2	正负号	+ -
3	自增自减，非	++ -- !
4	乘除，取余	* / %
5	加减	+ -
6	移位运算	<< >> >>>
7	大小关系	> >= < <=
8	相等关系	== !=
9	按位与	&
10	按位异或	^
11	按位或	
12	逻辑与	&&
13	逻辑或	
14	条件运算	? :
15	赋值运算	= += -= *= /= %=
16	位赋值运算	&=  = <<= >>= >>>=

如果在程序中，要改变运算顺序，可以使用()  
。

常用字符与ASCII代码对照表

# 8.java基本语法-流程控制语句

2021年7月29日 12:00

顺序结构

分支结构

循环结构

从键盘获取变量：需要使用Scanner类

步骤：

1. 导包： import java.util.Scanner
2. Scanner的实例化： Scanner scan=new Scanner(System.in);
3. 调用Scanner类的相关方法来获取指定类型的变量  
a=scan.nextInt(); //scan.nextXxx()/next()  
输入类型与要求不匹配时，会报异常，终止程序

if-else结构

```
If(表达式){  
    语句1  
}else{  
    语句2  
}
```

```
switch(表达式){  
    case 常量1: 语句1; break;  
    case 常量2: 语句2; break;  
    case 常量3: 语句3; break;  
    case 常量4: 语句4; break;  
    default: 默认语句;  
}
```

--表达式中数据类型

byte、 short、 char、 int、 枚举类型、 String类型

for循环

while循环

do-while循环

```
while (true) {  
    不满足时使用break终止  
}
```

特殊关键词的使用

break

continue

默认结束最后循环

带标签的break与continue

```
Label:for(...)  
break label;
```

结束带指定标签的循环

质数例子

```
class LogicText{  
    public static void main(String[]args){  
        int count=0;  
        label:for(int i=2;i<=100000;i++){  
            for(int j=2;j<=Math.sqrt(i);++j){  
                if(i%j==0)  
                    continue label;  
                count++;  
            }  
        }  
    }  
}
```

# 9.java基本语法-Eclipse使用

2021年7月30日 8:44

下载后直接找到文件直接解压，点开EXE文件即可

--选择workspace,不要默认

--Navigator,Package Explorer,Outline,Console为常用窗口

--透视图选择JavaEE,方便以后使用写Web

--Window下进行字体和编码utf-8设置

--Window下进行视图配置与刷新，调整New时显示的文件

新建工程---src下新建包--包下新建class文件

导入已有工程--File--Import--General--Existing Projects into workspace

--选择工程--选择复制到我们的workspace (通常可选)

导入已有源代码：直接复制，然后在包下粘贴

(如有乱码问题，将文件改为utf-8编码形式)

最后在文件上方加来源，如在哪个包内

删除工程，右键Delete，勾选的话会从硬盘上删掉，不勾只是从当前工作区移除

新开workspace需要重新设置

空间下 .metadata 文件夹保存的是Eclipse配置信息

查看源代码：Ctrl键，鼠标移动到类上，在Outline上看索引

ctrl O 快速查找

在编写代码中显示程序员的相关信息

window-preferences-java-code style-code templates

-comments

/\*\*回车，可以显示类信息和方法信息

# 10.java基本语法-数组

2021年7月30日 10:00

数组 (Array) :

数组名

元素

索引

数组长度

特点：引用类型，顺序排列，连续的内存空间，  
长度确定后不能修改，元素类型任意

分类：一维数组、二维数组、基本数据类型数组、  
引用数据类型数组

一维数组

1.int[] ids;//声明

//静态初始化：数组的初始化和数组元素的赋值操作同时进行

ids=new int[] {1001,1002,1003,1004};

//动态初始化：数组的初始化和数组元素的赋值操作分开进行

String[] names=new String[5];

数组一旦初始化完成长度就确定了，通过索引进行调用

2.数组长度

使用属性: .length

3.数组元素的默认初始化值

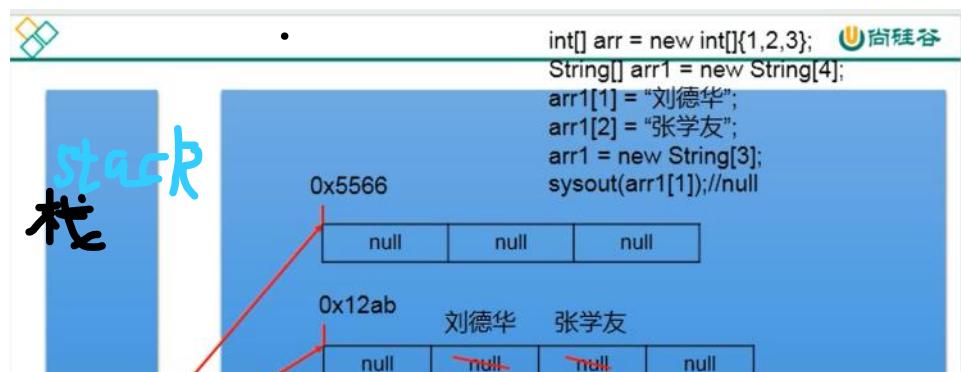
整型：0

浮点型：0.0

char型：0或'\u0000'

boolean: false

引用数据类型：null





## 二维数组

### 1. //静态初始化

```
int[][] arr1=new int[][] {{1,2,3},{4,5,6}}
```

### //动态初始化

```
int[][] arr2=new int[3][2];
arr1[0][1]--2
```

### 2. 长度获取使用length属性，返回几排

### 3. 遍历

```
for(int i=0;i<arr1.length;++i) {
    for(int j=0;j<arr1[i].length;++j) {
        System.out.print(arr1[i][j]);
    }
    System.out.println();
}
```

### 4. 二维数组元素的默认初始化值

```
Int [][] arr=new int[3][3];
```

arr[0]//地址值

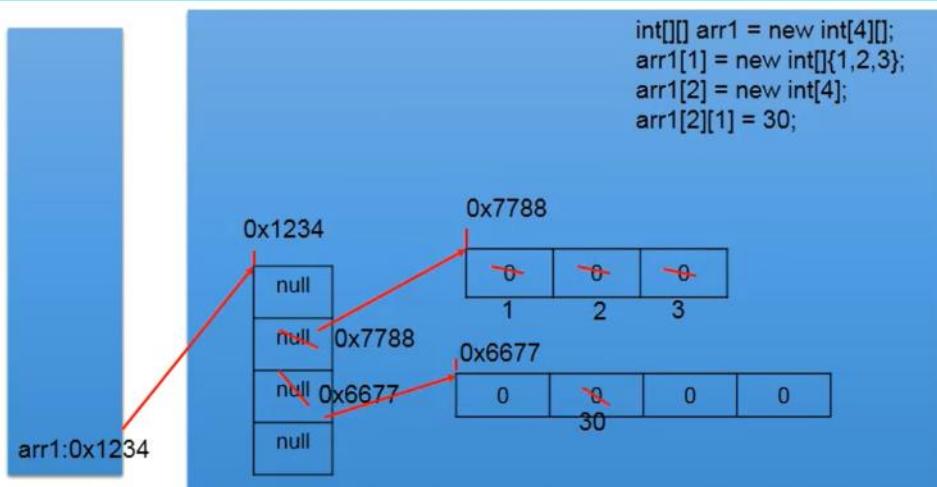
arr[0][0]//0

外层都是地址值，内层都是与一维一样的值

```
Int[][] arr=new int[3][]
```

arr[0]--null

arr[0][1]--报错



# 11. 数组中常见算法

2021年7月30日 12:41

1. 数组元素的赋值 (杨辉三角、回形数等)
2. 数值型数组最大、最小、平均、和;
3. 数组的复制、反转、查找 (线性查找、二分查找)
4. 数组元素的排序算法

```
Int[] arr1=new int[3];
Int[] arr2;
arr2=arr1;
```

数组变量相互赋值，传的是地址，仍指向同一个数组

复制

```
arr2=new int[3];
```

然后再用for循环赋值才是真的复制数组。

str1.equals(str2)--比较两个字符串内容是否相等

## 十大内部排序算法

- 选择排序
  - 直接选择排序、堆排序
- 交换排序
  - 冒泡排序、快速排序
- 插入排序
  - 直接插入排序、折半插入排序、Shell排序
- 归并排序
- 桶式排序
- 基数排序

# 12.Arrays工具类的使用

2021年7月30日 21:23

Import java.util.Arrays;--导入工具类

例：

boolean equals(int[] a,int[] b);//判断两个数组是否相等

(使用类方法调用，如Arrays.equals())

String toString(int[]a)//输出数组信息

void fill(int[]a,int val)//指定值填入数组

void sort(int[] a)//对数组进行排序

int binarySearch(int[]a,int key)//对排序后的数组进行二分查找

更多直接查找API

# 13.数组使用中的常见异常

2021年7月30日 21:55

空指针异常：NullPointerException

-索引越界

-二维数组中

# 14.面向对象（上）-创建对象

2021年7月31日 9:36

面向过程 (pop, Procedure Oriented Programming)

强调功能，以函数为最小单位

面向对象 (oop, Object Oriented Programming)

强调具备功能的对象，以类/对象为最小单位

封装

继承

多态

面向对象的两个要素：

类：对一类事物的描述，是抽象的、概念上的定义

对象：是实际存在的该类事物的个体，因此也称为实例 (instance)

oop重点是类的设计

类的设计就是类的成员的设计

Field=属性=成员变量

Method= (成员) 方法=函数

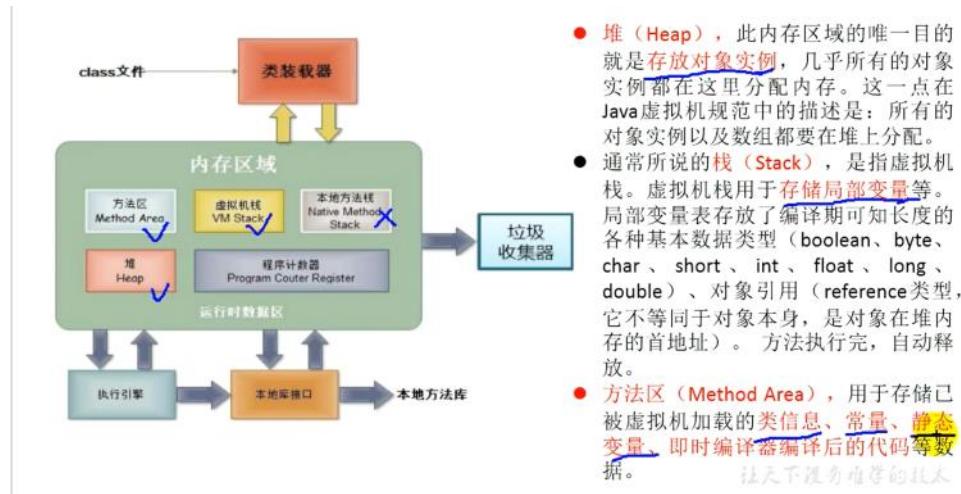
```
public class Oop {  
    public static void main(String[] args) {  
        Person p1=new Person();//创建对象  
        p1.eat();  
        p1.name="Tom";  
        p1.age=20;  
        p1.isMale=true;  
    }  
}  
  
class Person{  
    String name;//属性  
    int age=1;  
    boolean isMale;  
    public void eat() {//方法  
        System.out.println(666);  
    }  
}
```

1. 创建类，设计类成员
2. 创建类的对象
3. 使用 . 调用属性/方法

创建多个对象时每个对象都有类的属性（非static），修改一个对象不影响其他对象。

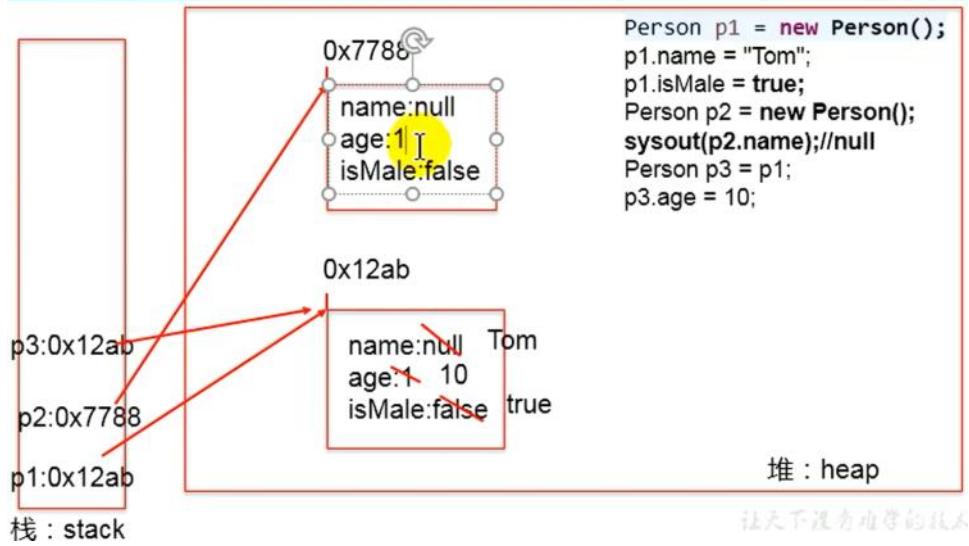
# 15.面向对象-内存解析

2021年7月31日 11:15



## 对象的内存解析

尚硅谷



# 16.面向对象-属性与局部变量

2021年7月31日 11:24

属性（成员变量）与局部变量的区别

--不同点

属性：直接定义在类的{}内，可以在声明属性时，指明其权限

使用权限修饰符

常用的权限修饰符：private、public、缺省(默认)、protected -->封装性

局部变量：声明在方法内、方法形参、代码块内

、构造器形参、构造器内部的变量

不可以使用权限修饰符

--相同点

定义变量格式

先声明，后使用

变量都有对应的作用域

默认初始化值情况

--属性

整型：0

浮点型：0.0

字符型：0或'\u0000'

布尔型：false

--局部变量

没有默认初始化值，必须进行显示赋值

特别的：形参在调用方法时才赋值

内存中加载位置

--属性：

加载到堆空间（非static）

--局部变量

加载到栈空间

# 17.面向对象-方法声明与使用

2021年7月31日 18:11

方法的声明：

```
权限修饰符 返回值类型 方法名 (形参列表) {  
    方法体;  
}
```

java四种权限修饰符

--private、public、缺省(默认)、protected

返回值类型：有返回值vs无返回值

--如果方法有返回值则必须在方法声明时，指定返回值的类型

同时需要return关键字返回该类型变量

--无返回值使用void，通常不使用return，但是可以使用 return; 结束方法

方法名：

-属于标识符，需要见名知意

形参列表：

格式：数据类型1 形参1, 数据类型2 形参2,.....

方法体：

具体操作

return关键字作用：

①结束方法；

②返回相应数据类型

方法的使用：

可以调用当前类的属性或方法，方法也可以自己调用自己

递归方法。

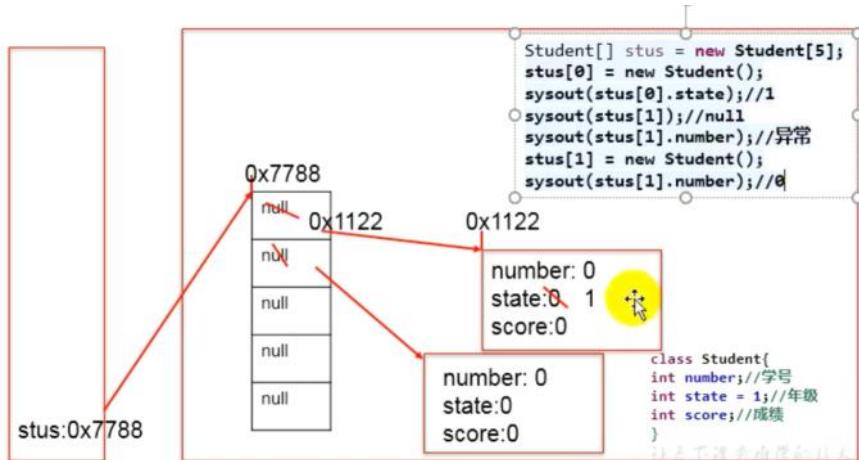
方法不能嵌套定义

# 18. 对象数组

2021年7月31日 20:08

声明方法与String型一样

```
Student[] sts=new Student[20];//对象数组
for(int i=0;i<sts.length;++i) {
    sts[i]=new Student();//给数组元素赋值
    sts[i].number=i+1;//给元素的属性赋值
    sts[i].state=(int)(Math.random()*(6-1+1)+1);
    sts[i].score=(int)(Math.random()*(100+1));
}
```



# 19.面向对象理解

2021年8月1日 10:26

1.在Java语言范畴中，我们都将功能、结构等封装到类中，通过类的实例化来调用具体的功能结构

>Scanner, String

>文件： File

>网络资源： URL

2.涉及到Java语言与前端、后端的数据库交互时，前后端的结构在Java层面交互时，都体现为类、对象。

## 20.匿名对象

2021年8月1日 10:42

理解：创建的对象，没有显式的赋值给变量名

特征：只能调用一次

使用：可以做实际参数传给形参，从而可以在方法中调用

new Person();

# 21.方法的重载、方法的递归

2021年8月1日 11:09

定义：在同一个类中允许存在一个以上的同名方法，只要参数个数或者类型不同即可，称为函数重载。

(同类同方法，参数列表不相同)

与权限修饰符，返回值类型，参数名称无关。

通过对象调用方法时，确定指定方法

方法名--参数列表

方法自己调用自己称为递归

## 22. 可变个数形参的方法

2021年8月1日 11:45

可变个数形参的格式：

数据类型 ... 变量名---public void show (String ... sts)

--当调用可变个数形参的方法时，传的参数可数0, 1, 2....

--传的参数必须都是对应数据类型

--与同类型数组作为参数的方法不能共存

--将参数当成数组遍历

--可变个数形参在方法的形参中必须声明在末尾

## 23.值传递机制

2021年8月1日 12:08

赋值机制：

变量是基本数据类型，此时赋值的是变量所保存的数据值

变量是引用类型，赋值地址值

实参，形参传递机制：

基本数据类型进行值传递

引用数据类型进行地址传递

# 24.面向对象-封装与隐藏

2021年8月1日 18:11

隐藏对象内部的复杂性，只对外公开简单的接口。

便于外界调用，从而提高系统的可扩展性、可维护性

通俗的说，把该隐藏的隐藏起来，该暴露的暴露出来。

这就是封装性的设计思想

封装性的一个体现：

将类的属性xx私有化（private），同时，

提供公共的（public）方法来获取（getxxx）和设置（setxxx）私有属性。

拓展：封装性的体现：

- ①属性私有化（如上）
  - ②不对外暴露的私有的方法
  - ③单例模式
- .
- .

1.权限修饰符----封装性的体现

Java规定的4中权限修饰符：

private、缺省(默认)、protected、public

## 四种访问权限修饰符

Java权限修饰符public、protected、private置于类的成员定义前，用来限定对象对该类成员的访问权限。

修饰符	类内部	同一个包	不同包的子类	同一个工程
private	Yes			
(缺省)	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

对于class的权限修饰只可以用public和default(缺省)。  
➤ public类可以在任意地方被访问。  
➤ default类只可以被同一个包内部的类访问。

2.4种权限修饰符可以用来修饰类及类的内部结构：

属性、方法、构造器、内部类

具体的：修饰类的内部结构：都可以用

修饰类，只能用：public、缺省

总结：Java提供了4中权限修饰符来修饰类及类的内部结构，  
体现了类与类内部结构在被调用时可见性的大小。

# 25.面向对象-构造器的使用

2021年8月2日 10:09

构造器 (constructor) 是类的结构之一

构造器：

--作用：①创建对象 new + 构造器

②初始化对象的信息（传参数）

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

--如果没有显示的定义类的构造器，则系统默认提供一个空参构造器。

--定义构造器的格式：权限修饰符 类同名（形参列表）{}

--一个类中定义的多个构造器，彼此构成重载

--一旦显示定义了构造器，系统不在提供默认空参构造器

--一个类中至少会有一个构造器

属性赋值总结：

①默认初始化

②显式初始化

③构造器中赋值

④通过 对象.方法 或者 对象.属性 的方式赋值

以上操作先后顺序：①-②-③-④

# 26.面向对象-JavaBean的使用

2021年8月2日 10:54

JavaBean是一种Java语言写成的可重用组件。

JavaBean是指符合如下标准的Java类

- >类是公共的
- >有一个无参的公共构造器
- >有属性，且有对应的get、set方法

# 27.面向对象-this的使用

2021年8月2日 11:00

this关键字的使用

--this可以修饰：属性、方法、构造器

--this修饰属性和方法，表当前对象或正在创建的对象

this.field

this.method

--通常会省略this，特殊的如果方法形参和类的属性同名，则不能省略

构造器中使用时，表示当前正在创建的对象的属性或者方法，也可省略

this调用构造器

--在类的构造器中可以显式的使用"this(形参列表)"来调用本类

中指定的其他构造器（由参数确定）

--构造器中不能通过这种方式调用自己

--如果一个类中有n个构造器，则最多n-1个使用this，避免死循环

--构造器的调用"this(形参列表)"必须声明在构造器的首行，所以只能声明一个

# 28.面向对象-package

2021年8月2日 15:17

package关键字的使用

- 为了更好的实现项目中类的管理，提供包的概念
- 使用package声明类或接口所属的包，声明在首行
- 包，属于标识符，遵循标识符的命名规则xxxxyyzzz,见名知意
- 每“.”一次，就代表一层文件目录
- 同一个包内不能命名同名的接口、类，不同包下可以。

## JDK中重要的包介绍

1. **java.lang**----包含一些Java语言的核心类，如String、Math、Integer、System和Thread，提供常用功能
2. **java.net**----包含执行与网络相关的操作的类和接口。
3. **java.io** ----包含能提供多种输入/输出功能的类。
4. **java.util**----包含一些实用工具类，如定义系统特性、接口的集合框架类、使用与日期日历相关的函数。
5. **java.text**----包含了一些java格式化相关的类
6. **java.sql**----包含了java进行JDBC数据库编程的相关类/接口
7. **java.awt**----包含了构成抽象窗口工具集（abstract window toolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面(GUI)。 B/S C/S

# 29.MVC设计模式

2021年8月2日 15:29

## MVC设计模式

MVC是常用的设计模式之一，将整个程序分为三个层次：视图模型层，控制器层，与数据模型层。这种将程序输入输出、数据处理，以及数据的展示分离开来的设计模式使程序结构变的灵活而且清晰，同时也描述了程序各个对象间的通信方式，降低了程序的耦合性。

### 模型层 **model** 主要处理数据

- >数据对象封装 model.bean/domain
- >数据库操作类 model.dao
- >数据库 model.db

### 控制层 **controller** 处理业务逻辑

- >应用界面相关 controller.activity
- >存放fragment controller.fragment
- >显示列表的适配器 controller.adapter
- >服务相关的 controller.service
- >抽取的基类 controller.base

### 视图层 **view** 显示数据

- >相关工具类 view.utils
- >自定义view view.ui

https://www.bilibili.com

## 30.面向对象-import

2021年8月2日 15:37

import: 导入

--在源文件中显式的使用import结构导入  
指定包下的类、接口

--声明在包的声明和类的声明之间，需要导入多个则并列写上。

--可以使用"xxx.\*",方式导入xxx包下所有的类、接口，如果该包下有子包，仍需再导入

--如果使用的类或接口时java.lang包下定义的，  
则可以省略import结构，但有子包仍需导入

--如果类或接口在本包下，省略

--如果源文件中，使用了不同包下同名的类，则必须至少  
有一个类需要以全类名的方式显示。 (java.util.Date d1=new java.util.Date();)

--子包仍需导入

--import static: 导入指定类或接口中的静态结构：属性或方法

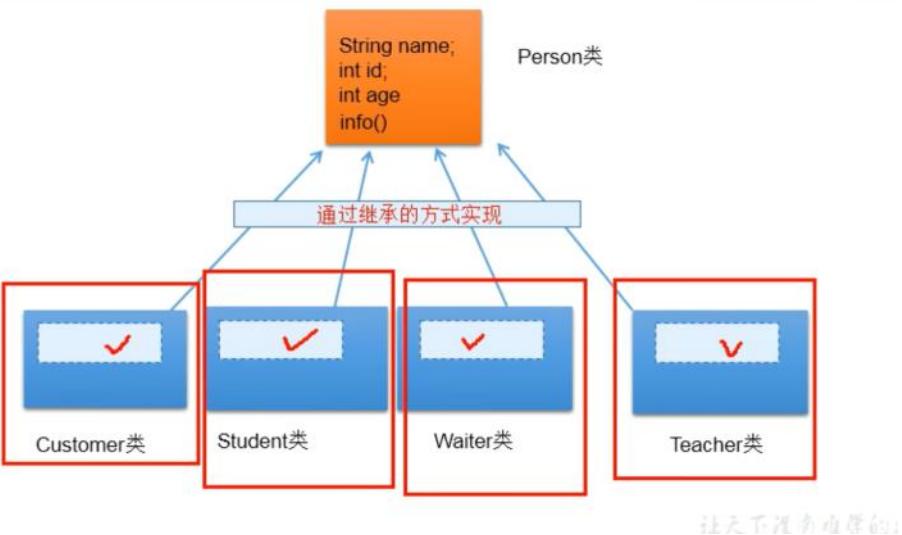
# 31.Eclipse常用快捷键

2021年8月2日 18:20

- 1.补全代码的声明: alt+ /
- 2.快速修复: Ctrl+1
- 3.批量导包: Ctrl+shift+o
- 4.使用单行注释: Ctrl+ /
- 5.使用多行注释: Ctrl+shift+ /
- 6.取消多行注释: Ctrl+shift+\
- 7.复制指定行的代码: Ctrl+alt+down 或 Ctrl+alt+up
- 8.删除指定行的代码: Ctrl+d
- 9.上下移动代码: alt+up 或 alt+down
- 10.切换到下一行代码空位: shift+enter
- 11.切换到上一行代码空位: Ctrl+shift+enter
- 12.如何查看源码: Ctrl+选中指定的结构 或 Ctrl+shift+t
- 13.退回到前一个编辑的页面: alt+left
- 14.退回到下一个编辑页面: alt+right
- 15.光标选中指定的类, 查看继承树结构: ctrl+t
- 16.复制代码: Ctrl+c
- 17.撤销: Ctrl+z
- 18.反撤销: Ctrl+y
- 19.剪切: Ctrl+x
- 20.粘贴: Ctrl+v
- 21.保存: ctrl+s
- 22.全选: Ctrl+a
- 23.格式化代码: ctrl+shift+f
- 24.选中数行, 整体后移: tab
- 25.选中数行, 整体前移: shift+tab
- 26.在当前类中, 显示类结构, 并支持搜索指定的方法、属性等: Ctrl+o
- 27.批量修改指定的变量名、方法名、类名等: alt+shift+r
- 28.选中的结构变成大写: ctrl+shift+x
- 29.选中的结构变成小写: ctrl+shift+y
- 30.调出生成getter/setter/构造器等结构: alt+shift+s
- 31.显示当前选择资源(工程/文件)的属性: alt+enter
- 32.快速查找: 参考选中的Word快速定位到下一个: Ctrl+K
- 33.查看指定结构使用过的位置: ctrl+alt+g

## 32.面向对象-继承性与object

2021年8月3日 16:38



### 继承性 (inheritance)

#### 1.好处

- 减少代码冗余，提高代码的复用性
- 便于功能的扩展
- 为之后多态性的使用，提供了前提

#### 2.继承性的格式

```
class A extends B{}
```

A:子类、派生类、subclass

B:父类、超类、基类、superclass

--体现：一旦子类A继承父类B以后，子类A就  
获取到了父类B声明的结构：属性和方法

特别的：私有的属性和方法也可以获取到，  
封装性不会打破，也需要在子类中使用公有方法调用。

--子类继承父类后，也可以声明自己特有的属性和方法，拓展  
功能，不同于集合关系。

#### 3.Java中关于继承性的规定

- ①一个类可以有多个子类
- ②类的单继承性：一个子类只能有一个父类
- ③子父类是相对的概念。
- ④子类直接继承的父类称为直接父类，间接继承的称为间接父类
- ⑤子类继承父类以后，就获取了直接父类与间接父类的属性与方法。

#### 4.object

- ①如果我们没有显式的声明一个类的父类的话，则此类继承于java.lang.Object类
- ②所有的java类（除java.lang.Object类之外）都直接或间接的继承于java.lang.Object类
- ③意味着，所有的java类具有java.lang.Object类声明的功能。

# 33.Eclipse-Debug使用

2021年8月3日 18:24

如何调试程序：

1.System.out.println();

2.Eclipse Debug 使用

--双击设置断点

--右键-debug as-打开debug透视图

--step over 下一行

--step into 进入方法

--step return 离开方法

--Terminate 结束程序

--Resume 下一个断点

如果step into无法进入方法，是JRE配置问题

需要进入debug as里修改jre为JDK中的jre

操作	作用
step into 跳入 (f5)	进入当前行所调用的方法中
step over 跳过 (f6)	执行完当前行的语句，进入下一行
step return 跳回 (f7)	执行完当前行所在的方法，进入下一行
drop to frame	回到当前行所在方法的第一行
resume 恢复	执行完当前行所在断点的所有代码，进入下一个断点，如果没有就结束
Terminate 终止	停止 JVM，后面的程序不会再执行

# 34.面向对象-方法的重写

2021年8月3日 19:13

方法的重写 (override/overwrite)

1.重写：子类继承父类以后，可以对父类中同名同参数的方法，进行覆盖操作。

2.应用：重写后，当创建子类对象后，通过子类对象调用子父类中的同名同参数的方法是，实际调用的是子类重写父类的方法。

3.重写的规定

方法的声明：权限修饰符 返回值类型 方法名 (形参列表) throws 异常的类型{  
    //方法体  
}

①子类重写的方法的方法名和形参列表与父元素被重写的方法的方法名和形参列表相同

②子类重写的方法权限修饰符 $\geq$ 父类中被重写的方法

>特殊情况，子类不能重写父类中声明为private权限的方法，无法覆盖

③返回值类型：

>父类被重写的方法的返回值类型是void，则子类重写的方法的返回值只能是void

>父类被重写的方法的返回值类型是A类型，则重写的方法的返回值可以是A类或A类的子类。

>父类返回值基本数据类型，子类返回值必须是相同的基本数据类型

④子类重写的方法抛出的异常类型 $\leq$ 父类被重写方法抛出的异常的类型  
(后面异常处理)

！！！子类和父类中同名同参数的方法

都声明为非static的（考虑重写），或者都声明为static的（此时不是重写，两个独立存在）

# 35.面向对象-super关键字使用

2021年8月4日 11:26

super关键字的使用：

--super理解为父类的

--super可以用来调用：属性、方法、构造器

--super使用

①可以在子类的方法和构造器中，

通过“super.属性/方法”，显式的调用父类方法与属性

通常省略。

②特殊的：当子类和父类中定义了同名的属性，调用父类中属性时

需要使用“super.属性”

③父类方法在子类中被重写，还需要调用父类方法时，使用“super.方法”

--super调用构造器

①在子类构造器中显式的使用“super（形参列表）”的方式调用父类中指定构造器。

②“super（形参列表）”必须放在子类构造器首行！

③this(...)与super (...) 只能使用一个。

④当构造器的首行没有显式声明this(...)或super (...), 则默认调用

空参的super () 调用父类空参的构造器；

⑤在类的多个构造器中至少有一个类的构造器使用“super（形参列表）”

# 36.面向对象--子类对象实例化过程

2021年8月4日 12:07

子类对象实例化过程

①从结果上获取了父类属性与方法

创建子类的对象，在堆空间中，就会加载所有父类中声明的属性。

②从过程上看：

当通过子类构造器创建子类对象时，会直接或者间接调用父类的构造器

直到调用了java.lang.Object类中空参的构造器为止，加载所有父类的结构

子类对象才能调用

③虽然创建子类对象时，调用了父类的构造器，

但是只创建了一个对象，即new的子类对象

！！！对于继承看写的源码com.zqf.account

# 37.面向对象-多态性

2021年8月4日 13:14

## 多态性

### 1. 对象多态性：

父类的引用指向子类的对象（或者叫子类的对象赋值给父类的引用）

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

### 2. 多态的使用：虚拟方法调用

--编译期只能调用父类中声明的方法，但在运行期

实际执行的是子类重写父类的方法。

总结：编译看左边，运行看右边

--多态性是运行时行为

### 3. 多态性使用的前提：

①类的继承性

②方法的重写

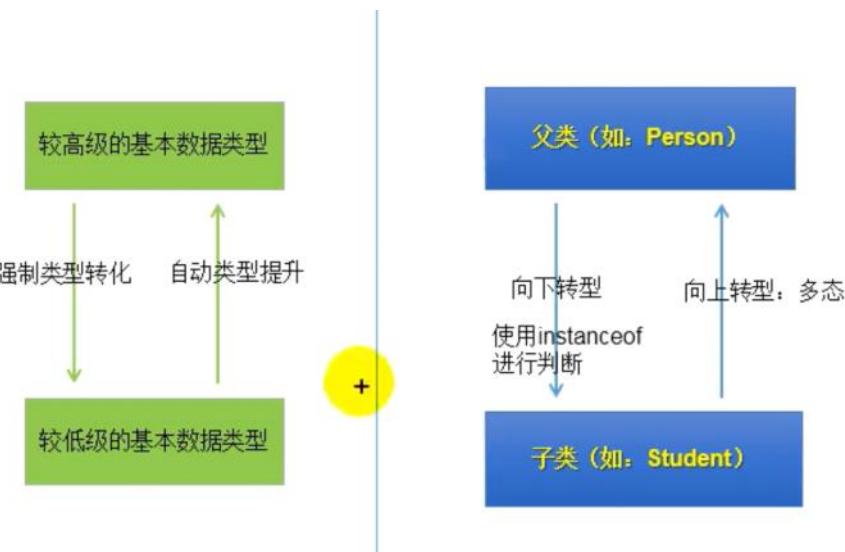
### 4. 对象的多态性只适用于方法，不适用于属性

### 5. 使用场景之一，比如可以把父类作为方法的参数，

然后传递子类对象，在方法体调用重写方法，就不用再为

每一个子类声明类似的方法了。

补充：



1. 创建对象多态性后，内存中实际上加载了子类特有的属性和方法  
由于声明类型是父类，所有只能调用父类中声明的属性和方法。

2. 使用强制类型转换符

```
Person p1=new Student();
Student s1=(Student)p1;
```

3. 使用强转时，可能出现ClassCastException的异常

instanceof关键字的使用

a instanceof A：判断对象a是否是类A的实例，是返回true，不是false

为了避免向下转型出现异常，转前先使用instanceof判断，true就向下转型

例：

```
if (p1 instanceof Student) {
    Student s1=(Student)p1;
}
```

!!!只要是Student (new的对象) 的父类都返回true

# 38.Object类的使用

2021年8月4日 19:52

Java.lang.Object类

- 1.Object类是所有类的根父类
- 2.Object类中的方法和属性具有通用性。
- 3.Object类只声明了一个空参的构造器

属性：无

方法： equals()/toString()/getClass()/hashCode()/  
clone()/finalize()/wait()/notify()/notifyAll()

==运算符（总结）：

- 1.可以使用在基本数据类型变量和引用数据类型变量中
- 2.基本数据类型比较值的大小
- 3.引用数据类型比较两个变量地址值是否相同，即两个引用是否指向同一个对象

equals () 方法的使用

- 1.是一个方法
- 2.只适用于引用数据类型
- 3.Object类中equals () 的定义：

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

说明：Object类中的此方法与==作用相同
- 4.像String/Date/File/包装类等重写了equals () 方法。  
重写后只比较实体内容是否相同
- 5.我们自定义的类使用equals () ,希望可以比较实体内容  
则需要重写。

toString()方法的使用

- 1.当我们输出一个对象的引用时，与使用 对象.toString() 一样输出地址
- 2.Object 源码 

```
public String toString() {  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```
- 3.像String/Date/File/包装类等重写了toString()方法  
使得在调用对象的toString()方法时直接输出实体对象
- 4.重写toString () 方法，使其返回实体内容

# 39.面向对象-Junit单元测试

2021年8月6日 10:05

Java中的Junit单元测试

步骤：

- 1.选中当前工程-右键选择：build path- add libraries - Junit4 - 下一步
- 2.创建Java类，进行单元测试。

Java类要求：

- ①此类是public的
- ②此类提供公共的无参的构造器
- 3.此类中声明单元测试方法

此时的单元测试方法：方法的权限是public，没有返回值，没有形参。

- 4.此单元测试方法上需要声明注解@Test,并在单元测试中

导入：import org.junit.Test

- 5.声明好单元测试方法后，就可以在方法体内测试相关的代码

- 6.写完代码以后，左键双击单元测试方法名：右键：run as-Junit Test

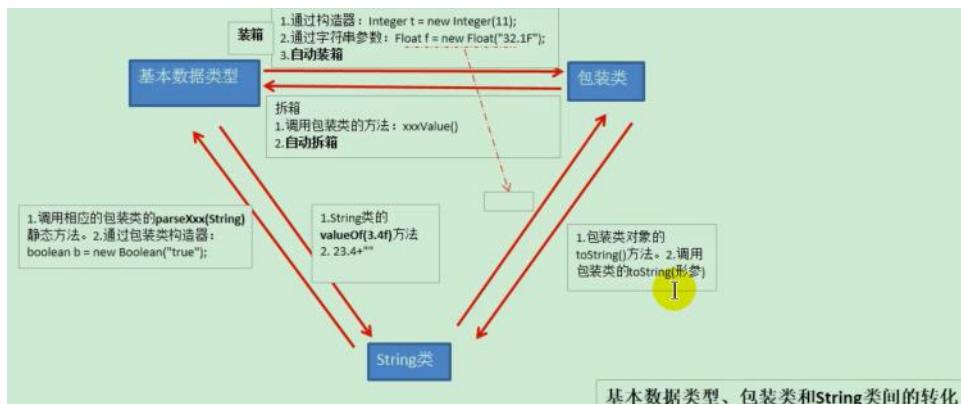
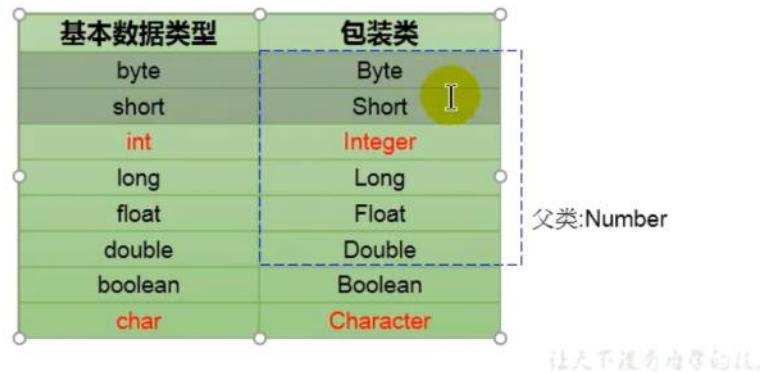
说明：

- 1.如果执行无异常，绿条
- 2.如果执行结果出现异常：红条

# 40.面向对象-包装类 (Wrapper)

2021年8月6日 10:28

- 针对八种基本数据类型定义相应的引用类型—包装类（封装类）
- 有了类的特点，就可以调用类中的方法，Java才是真正的面向对象



1. Java提供了8种基本数据类型对应的包装类，使得基本数据类型具有类的特征

2. 基本数据类型、包装类、String三者之间的转换

基本-包装： Integer in1=new Integer(45);---调用包装类的构造器

包装-基本： int in2=in1.intValue();-----调用包装类的xxxValue()

JDK5.0新特性：自动装箱与自动拆箱（常用）

int num2=10;

Integer num1=num2;//自动装箱:基本-包装

Int num3=num1;//自动拆箱： 包装-基本

基本数据类型、包装类-->String

①连接运算： String str1=num+"";

②调用String 的 valueOf(xxx);

String str2=String.valueOf(num);

String-->基本数据类型、包装类

调用包装类的parseXxx (String) 方法

int num1=Integer.parseInt(str1);//不能转的会报异常

## 一个例题

```
@Test
public void test3() {
    Integer i = new Integer(1);
    Integer j = new Integer(1);
    System.out.println(i == j); //false

    //Integer内部定义了IntegerCache结构，IntegerCache中定义了Integer[],
    //保存了从-128~127范围的整数。如果我们使用自动装箱的方式，给Integer赋值的范围在
    //[-1]28~127范围内时，可以直接使用数组中的元素，不用再去new了。目的：提高效率

    Integer m = 1;
    Integer n = 1;
    System.out.println(m == n); //true

    Integer x = 128;
    Integer y = 128;
    System.out.println(x == y); //false
}
```

# 41.面向对象-static关键字

2021年8月6日 15:48

static关键字的使用

1.static静态的

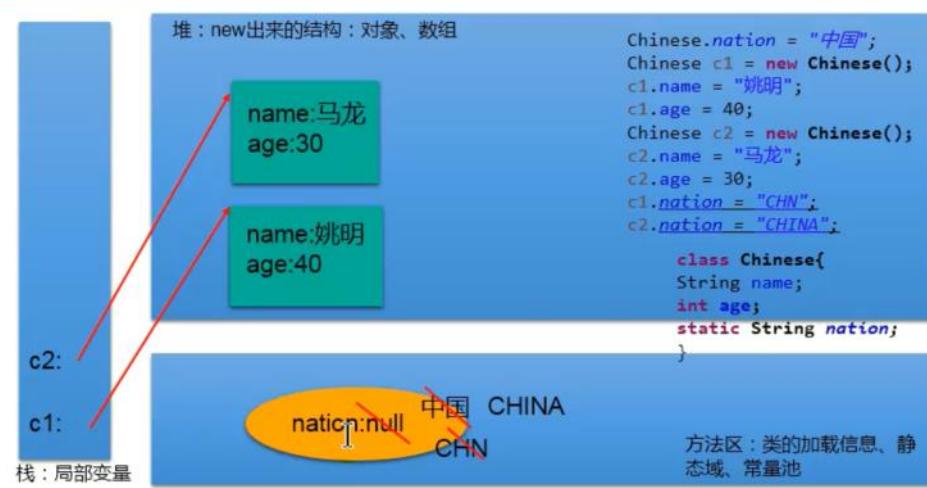
2.static可以用来修饰：属性、方法、代码块、内部类

3.static修饰的属性：静态属性（静态变量、类变量），未使用的称为非静态属性（实例变量）

- ①实例变量：每个类的实例（对象）都独立创建，修改一个对象属性不影响其他对象
- ②静态变量：多个对象共享一个静态变量，通过任何对象修改都会导致其他对象相同属性改变。
- ③静态变量随着类的加载而加载，可以通过“类.静态变量”的方式调用，见下表。  
类只加载一次，所以静态变量在内存也只会存在一份，存在方法区的静态域中。

	静态变量	实例变量
类	ok	no
对象	ok	Ok

④静态变量的加载早于对象的创建



4.static修饰方法：静态方法

- ①随着类的加载加载，可以通过“类.静态方法”调用
- ②调用

	静态方法	非静态方法

类	yes	no
对象	yes	no

③静态方法中只能调用静态方法或属性

非静态方法中，静态与非静态的方法、属性都可以调用

④static注意点

在静态方法内，不能使用this和super关键字。

可以从声明周期进行理解

5.开发中如何确定是否声明为static

>属性值不会随对象改变

>类中的常量也通常用static修饰

>操作静态属性的方法通常设置为静态的

>工具类中的方法，习惯声明为static的，直接用类调用

如Math,Arrays...

# 42.面向对象-设计模式

2021年8月6日 18:09

**单例 (Singleton) 设计模式**

- 设计模式是在大量的实践中总结和理论化之后优选的代码结构、编程风格、以及解决问题的思考方式。设计模免去我们自己再思考和摸索。式就像是经典的棋谱，不同的棋局，我们用不同的棋谱，“套路”
- 所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。如果我们要让类在一个虚拟机中只能产生一个对象，我们首先必须将类的构造器的访问权限设置为private，这样，就不能用new操作符在类的外部产生类的对象了，但在类内部仍可以产生该类的对象。因为在类的外部开始还无法得到类的对象，只能调用该类的某个静态方法以返回类内部创建的对象，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的该类对象的变量也必须定义成静态的。

创建型模式，共5种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。  
结构型模式，共7种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。  
行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

## 单例 (Singleton) 设计模式

- 设计模式是在大量的实践中总结和理论化之后优选的代码结构、编程风格、以及解决问题的思考方式。设计模免去我们自己再思考和摸索。式就像是经典的棋谱，不同的棋局，我们用不同的棋谱，“套路”
- 所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。如果我们要让类在一个虚拟机中只能产生一个对象，我们首先必须将类的构造器的访问权限设置为private，这样，就不能用new操作符在类的外部产生类的对象了，但在类内部仍可以产生该类的对象。因为在类的外部开始还无法得到类的对象，只能调用该类的某个静态方法以返回类内部创建的对象，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的该类对象的变量也必须定义成静态的。

### 单例(Singleton)设计模式-应用场景



- 网站的计数器，一般也是单例模式实现，否则难以同步。
- 应用程序的日志应用，一般都使用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。
- 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。
- 项目中，读取配置文件的类，一般也只有一个对象。没有必要每次使用配置文件数据，都生成一个对象去读取。
- Application 也是单例的典型应用
- Windows的Task Manager (任务管理器)就是很典型的单例模式
- Windows的Recycle Bin (回收站)也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。

### 单例设计模式的实现总结

### 单例模式的饿汉式实现

- 1.私有化类的构造器
- 2.内部创建类的对象 (static类型)  
private static Person p1=new Person();
- 3.创建获取类的方法 (static类型)

单例模式的懒汉式实现

- 1.私有化类的构造器
- 2.声明当前类对象，没有初始化  
private static Person p1;
- 3.声明public、static的返回当前类对象的方法  
new之前判断类是否已经创建对象，避免调一次创建一次

懒汉式与饿汉式的区别

饿汉式：

- >缺点：对象生命周期过长，不用时占用内存
- >优点：饿汉式是线程安全的

懒汉式：

- >优点：延迟对象的创建
- >目前缺点：线程不安全 -->到多线程内容时优化

```
/**  
 * @author oscarzqf  
 * @description 单例懒汉式实现完善  
 * @create 2021-08-11-17:10  
 */  
public class Bank {  
    private Bank(){}
    private static Bank instance=null;
    public static Bank getInstance(){  
        if(instance==null) {  
            synchronized (Bank.class) {  
                if(instance==null){  
                    instance = new Bank();  
                }  
            }  
        }  
        return instance;  
    }
}
```

## 43.面向对象-main的说明

2021年8月6日 19:13

main () 方法的使用说明：

- 1.main () 方法作为程序的入口
- 2.main () 方法也是一个普通的静态方法
- 3.main () 方法可以作为与控制台交互的方式（之前：使用Scanner）

# 44.类的成员-代码块（或初始化块）

2021年8月6日 19:29

## 代码块

1.代码块的作用：用来初始化类或者对象

2.代码块修饰只能使用static

3.分类：

>静态代码块 static{}

-内部可以有输出语句

-随着类的加载而执行，而且只执行一次。

-初始化类的信息

-如果一个类中定义多个静态代码块，按声明顺序执行

-静态代码块先于非静态代码块先执行。

-只能调用静态结构

>非静态代码块 {}

-内部可以有输出语句

-随着对象的创建而执行，每创建一个对象执行一次。

-可以在创建对象时对对象的属性等进行初始化

-如果一个类中定义多个非静态代码块，按声明顺序执行

-静态与非静态属性与方法都可以调用

总结：执行：由父及子，静态先行

## 4.对属性可以赋值顺序

①默认初始化

②显式初始化

③构造器初始化

④对象.属性 对象.方法

⑤代码块中赋值

顺序：①-②/⑤-③-④

# 45.面向对象-finally关键字

2021年8月7日 10:22

final：最终的

1.final可以用来修饰：类、方法、变量

2.final 修饰的类：

>此类不能被其他类所继承，比如：String类、StringBuffer类

3.final 修饰方法

>表明此方法不能被重写

比如：Object类中的getClass(),不能被重写

4.final修饰变量

>此时的“变量”就称为是一个常量

>final修饰属性：可以考虑的位置有：显式初始化、代码块中赋值、  
构造器中初始化

>final修饰局部变量，尤其修饰形参时，表明形参时常量，  
只能使用，不能修改。

5.static final 用来修饰属性：全局常量

# 46.面向对象-抽象类-模板方法

2021年8月7日 14:32

abstract关键字的使用

1.abstract：抽象的

2.abstract：

可以修饰：类、方法

3.abstract修饰类：抽象类

>此类不能实例化

>抽象类中一定有构造器，便于子类实例化时调用。

>开发中都会提供抽象类的子类，通过实例化子类完成相应操作

4.abstract修饰方法：抽象方法

>抽象方法只有方法的声明，没有方法体

public abstract void eat();

>包含抽象方法的类，一定是一个抽象类。

>子类重写了父类(直接加间接)中的所有抽象方法后，才可以实例化。

>若没有重写则意味着这个子类也是一个抽象类，需要abstract修饰

5.abstract使用上的注意点：

>不能修饰：属性、构造器等结构

>abstract不能用来修饰私有方法、静态方法、final的方法、final的类

6.抽象类的匿名子类

```
//创建了一匿名子类的对象: p
Person p = new Person(){
    @Override
    public void eat() {
        System.out.println("吃东西");
    }

    @Override
    public void breath() {
        System.out.println("好好呼吸");
    }
};
```

创建对象时对抽象类Person的抽象方法进行重写，可以创建匿名子类，用一次

## 7.模板方法设计模式 (TemplateMethod)

开发中实现一个具体的算法时，整体步骤已经确定，父类中已经写好了但是某些部分易变，可以抽象出来，供不同的类实现，这就是一种模板模式。

模板方法设计模式是编程中经常用得到的模式。各个框架、类库中都有他的影子，比如常见的有：

- 数据库访问的封装
- Junit单元测试
- JavaWeb的Servlet中关于doGet/doPost方法调用
- Hibernate中模板程序
- Spring中JDBCTemplate、HibernateTemplate等

8.IO流中设计到的抽象类：InputStream/OutputStream/Reader/Writer  
在其内部定义的read () 、 write () 方法。

# 47.面向对象-接口 (interface)

2021年8月7日 15:37

## 接口的使用

### 1.接口使用interface来定义

权限修饰符 interface 接口名 {}

### 2.Java中，接口和类时并列的两个结构

### 3.如何定义接口：定义接口中的成员

>JDK7及之前：只能定义全局常量和抽象方法

-全局常量：public static final 的，但是写时可以省略

-抽象方法：public abstract的，但是也可以省略

>JDK8:除了定义全局常量和抽象方法之外，还可以定义静态方法、默认方法（略）

### 4.接口中不能定义构造器，意味着接口不能实例化

### 5.Java开发中，接口通过类实现 (implements) 的方式使用

>如果实现类重写了接口中所有的抽象方法，则实现类可以实例化

>如果未覆盖所有抽象方法，则实现类为抽象类

例：public class 类名 implements 接口名{}

### 6.Java类可以实现多个接口-->弥补了Java单继承性的局限性

格式：

class AA extends BB implements CC,DD,EE...{}

先写继承后写实现

### 7.接口和接口之间可以继承，而且可以多继承

### 8.接口的具体使用

-可以体现多态性，实现类的对象赋值给接口的引用

-接口实际上可以看做是一种规范

-开发中，体会面向接口编程！（JDBC）

### 9.接口的匿名实现类的对象（直接重写其中抽象的方法）

```
USB phone = new USB(){

    @Override
    public void start() {
        System.out.println("手机开始工作");
    }

    @Override
    public void stop() {
        System.out.println("手机结束工作");
    }

};
```

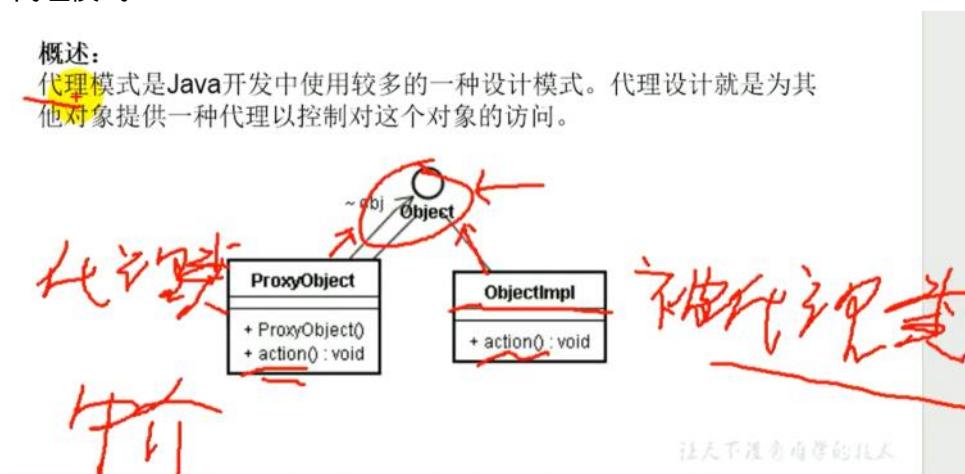
# 48. 接口应用-代理模式 (Proxy) - 工厂模式

2021年8月7日 17:44

## 代理模式

### 概述：

代理模式是Java开发中使用较多的一种设计模式。代理设计就是为其他对象提供一种代理以控制对这个对象的访问。



### 应用场景：

- 安全代理：屏蔽对真实角色的直接访问。
- 远程代理：通过代理类处理远程方法调用（RMI）。
- 延迟加载：先加载轻量级的代理对象，真正需要再加载真实对象。

比如你要开发一个大文档查看软件，大文档中有大的图片，有可能一个图片有100MB，在打开文件时，不可能将所有的图片都显示出来，这样就可以使用代理模式，当需要查看图片时，用proxy来进行大图片的打开。

### 分类

- 静态代理（静态定义代理类）
- 动态代理（动态生成代理类）
  - ✓ JDK自带的动态代理，需要反射等知识

## 工厂模式

### · 接口的应用：工厂模式

工厂模式：实现了创建者与调用者的分离，即将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的。

其实设计模式和面向对象设计原则都是为了使得开发项目更加容易扩展和维护，解决方式就是一个“分工”。

社会的发展也是这样，分工越来越细。

原始社会的人：人什么都要会，自己种，自己打猎，自己织衣服，自己治病。

现在的人：可以只会一样，其他都不会，只会 Java 也能活，不会做饭，不会开车，不会……

# 49.面向对象-Java8接口新特性

2021年8月8日 10:21

静态方法：

```
public static void f1()
```

默认方法：

```
public default int f2(){}--public可以省略
```

1.接口中的静态方法只能通过接口调用

```
接口.method ()
```

2.实现类的对象可以调用接口中定义的默认方法

如果实现类重写了接口中的默认方法，调用时调用

重写的方法

```
public int f2()
```

3.如果子类（或实现类）继承的父类和实现接口中声明了同名同参数

的方法，在子类没重写此方法时，默认调用父类中方法-->类优先原则

4.如果实现类的多个接口中定义了同名同参数的默认方法，没重写会报错

必须在实现类中重写该方法

5.调用接口中的默认方法

```
接口.super.method ()
```

# 50.面向对象-内部类的使用

2021年8月8日 11:11

- 当一个事物的内部，还有一个部分需要一个完整的结构进行描述，而这个内部的完整的结构又只为外部事物提供服务，那么整个内部的完整结构最好使用内部类。
- 在Java中，允许一个类的定义位于另一个类的内部，前者称为**内部类**，后者称为**外部类**。
- Inner class**一般用在定义它的类或语句块之内，在外部引用它时必须给出完整的名称。  
➤**Inner class**的名字不能与包含它的外部类类名相同；
- 分类：**成员内部类**（**static**成员内部类和非**static**成员内部类）  
**局部内部类**（不带修饰符）、**匿名内部类**

类的内部成员：内部类

1. Java中允许将一个类A声明在类B中，则类A就是内部类，类B称为外部类

2. 内部类的分类：

成员内部类：静态+非静态

局部内部类：方法内、代码块内、构造器内

3. 成员内部类

>作为类的成员

>调用外部类的结构

>可以被**static**修饰

>可以被四种权限成员修饰

>作为类，类内可以定义属性、方法、构造器等

>可以被**final**修饰，表示不能被继承

>可以被**abstract**修饰，不能实例化

4. 关注如下问题即可

>如何实例化成员内部类的对象

```
//创建Dog实例(静态的成员内部类):
Person.Dog dog = new Person.Dog();
dog.show();

//创建Bird实例(非静态的成员内部类):
Person.Bird bird = new Person.Bird(); //错误的
Person p = new Person();
Person.Bird bird = p.new Bird();
bird.sing();
```

>如何在成员内部类中区分调用外部类的结构

内部类中用this.xxx，表示内部类中结构

内部类中使用 外部类.this.xxx 表示调用外部类中结构  
(当结构名冲突时使用)

>开发中局部内部类的使用

用的不多，源码可能有

5.规定：在局部内部类的方法中，如果要调用局部内部类所在的方法中的局部变量  
要求此局部变量声明为final

外部类{

方法{

变量 (final)

内部类{

方法

}

}

}

6.总结：

成员内部类和局部内部类，编译后都会生成字节码文件

格式：成员内部类：外部类\$内部类名.class

局部内部类：外部类\$数字 内部类名.class

# 51. 异常处理-异常体系结构

2021年8月8日 15:48

- 异常：在Java语言中，将程序执行中发生的不正常情况称为“异常”。  
(开发过程中的语法错误和逻辑错误不是异常)

- Java程序在执行过程中所发生的异常事件可分为两类：

➤ **Error**: 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**和**OOM**，一般不编写针对性的代码进行处理。

➤ **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
- ✓ 数组角标越界

让天下没有难学的技术

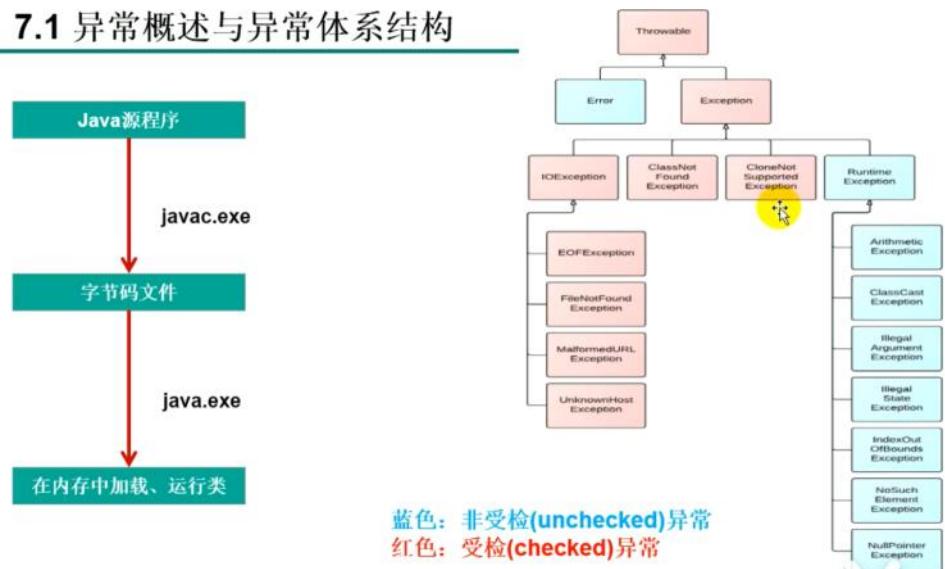
- 对于这些错误，一般有两种**解决方法**：一是遇到错误就终止程序的运行。另一种方法是由程序员在编写程序时，就考虑到错误的检测、错误消息的提示，以及错误的处理。



- 捕获错误最理想的是在**编译期间**，但有的错误只有在**运行时**才会发生。  
比如：**除数为0**，**数组下标越界**等

➤ 分类：**编译时异常**和**运行时异常**

## 7.1 异常概述与异常体系结构



### 1. 异常的体系结构

Java.lang.Throwable

>java.lang.Error: 严重错误，不编写代码处理

>java.lang.Exception: 可以进行异常的处理

----编译时异常 (checked)

\*\*\*\*FileNotFoundException

\*\*\*\*IOException

\*\*\*\*ClassNotFoundException

-----运行时异常 (unchecked, RuntimeException)

\*\*\*\*NullPointerException

\*\*\*\*ArrayIndexOutOfBoundsException

\*\*\*\*ClassCastException

\*\*\*\*NumberFormatException (数值转换异常)

\*\*\*\*InputMismatchException

\*\*\*\*ArithmeticeExceptioon(算术异常)

## 52.异常处理-两种方式

2021年8月8日 16:26

### 1.异常的处理：抓抛模型

抛：程序在正常执行过程中一旦出现异常，就会在异常代码处生成一个对应异常类型的对象，并将此对象抛出。一旦抛出对象后，其余代码就不在执行

抓：可以理解为处理异常的该方式

方式一：try-catch-finally

方式二：throws+异常类型

### 2.try-catch-finally的使用

```
try{
    //可能出现异常的代码
}catch (异常类型1 变量名1) {
    //处理异常的方式1
}catch (异常类型2 变量名2) {
    //异常处理方式2
}
.....
finally{
    //一定会执行的代码
}
```

说明：

①finally是可选的

②使用try将可能发生异常的代码包装起来，一旦出现异常就会生成对应异常类的对象根据此对象的类型，进入catch进行匹配

③一旦异常匹配到某个catch时，就进入执行代码处理异常  
处理完成跳出try-catch结构（无finally），继续执行后面的代码。

④catch中的异常类型如果是子父类关系，则子类写父类前面，否则报错  
其他位置随意。

⑤常用的异常对象处理方法：

-String getMessage()//返回字符串  
-printStackTrace()//输出错误具体内容

⑥try结构中声明的变量，外面不能用。

⑦try-catch-finally可以嵌套使用

体会1：使用try-catch-finally处理编译时异常，将一个编译时可能  
出现的异常，延迟到运行时出现。

体会2：开发中，由于运行时异常比较常见，所以我们通常不针对运行时异常  
编写try-catch-finally，报错需要修改代码。    针对编译时异常，一定要考虑异常的处理  
编译更像未雨绸缪，比如文件的有无。

### 3.finally的使用

①是可选的

②finally中声明的是一定被执行的代码，即使catch中又出现了异常，try中有return语句，  
catch中有return语句等情况

③像数据库连接、输入输出流、网络编程Socket等资源，JVM是不能自动  
回收的，我们需要自己手动释放资源，此时的资源释放，就需要  
声明在finally中

### 4.方式二：throws+异常类型

```
public class ExceptionTest2 {  
  
    public void method1() throws FileNotFoundException, IOException{  
        File file = new File("hello.txt");  
        FileInputStream fis = new FileInputStream(file);  
  
        int data = fis.read();  
        while(data != -1){  
            System.out.print((char) data);  
            data = fis.read();  
        }  
  
        fis.close();  
    }  
}
```

① “throws+异常类型” 写在方法的声明处，指明此方法执行时，可能会抛出的异常类型  
一旦执行方法体时出现异常，仍然会在异常代码处生成一个异常的对象，此对象满足throws后  
异常的类型时，就会被抛出，异常代码后续代码不再执行

②子类重写的方法抛出的异常类型 <= 父类被重写方法抛出的异常的类型

原因：多态性影响，如果子类类型大于父类，无法处理。

③try-catch-finally：真正去处理了异常

throws只是将异常抛给了方法的调用者，并没有真正处理掉。

④开发中如何选择使用哪种方法

>如果父类中被重写的方法没有throws异常，则子类重写的方法也不能使用throws，只能使用try-catch-finally方式处理

>方法a中的几个方法递进执行，建议将这几个方法使用throws方法处理  
然后在a中使用try-catch-finally处理

# 53. 异常处理-手动抛出异常与自定义异常类

2021年8月8日 19:03

1. 关于异常对象的产生：

- ① 系统自动生成的异常对象
- ② 手动生成一个异常对象，并抛出（throw），使用java提供的异常类

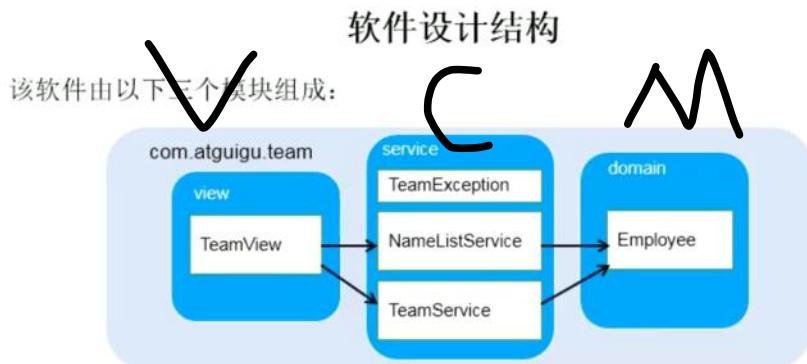
2. 自定义异常类

- > 继承现有的异常结构： RuntimeException / Exception
- > 提供全局常量： serialVersionUID(序列号唯一标识类)
- > 提供重载的构造器（空 + 参）

```
class Student{  
    private int id;  
  
    public void regist(int id) throws Exception {  
        if(id > 0){  
            this.id = id;  
        }else{  
            // 手动抛出异常对象  
            throw new RuntimeException("您输入的数据非法！");  
            // throw new Exception("您输入的数据非法！");  
            throw new MyException("不能输入负数");  
        }  
    }  
}
```

# 54. 面向对象-项目三TeamSchedule

2021年8月9日 9:37

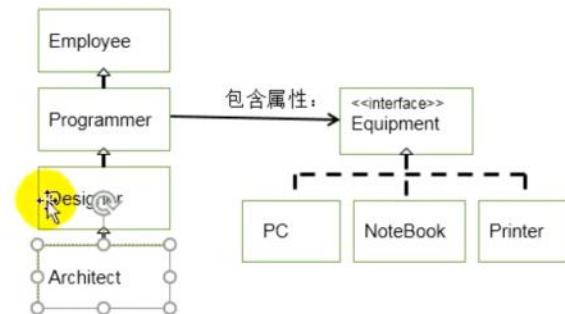


- com.atguigu.team.view模块为主控模块，负责菜单的显示和处理用户操作
- com.atguigu.team.service模块为实体对象（Employee及其子类如程序员等）的管理模块，NameListService和TeamService类分别用各自的数组来管理公司员工和开发团队成员对象
- domain模块为Employee及其子类等JavaBean类所在的包

往期不推荐看的课

## 软件设计结构

- com.atguigu.team.domain模块中包含了所有实体类：

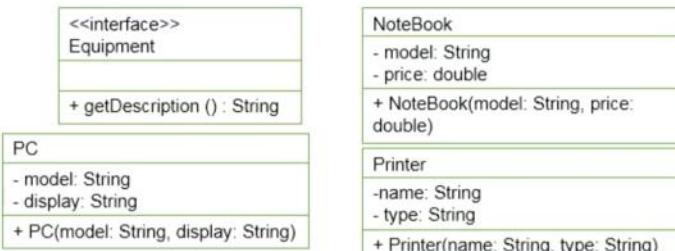


- 其中程序员(Programmer)及其子类，均会领用某种电子设备(Equipment)。

## 第1步 — 创建项目基本组件

- 1. 完成以下工作：
  1. 创建TeamSchedule项目
  2. 按照设计要求，创建所有包
  3. 将项目提供的几个类复制到相应的包中  
(view包中： TSUtility.java; service包中： Data.java)
- 2. 按照设计要求，在com.atguigu.team.domain包中，创建Equipment接口及其各实现子类代码
- 3. 按照设计要求，在com.atguigu.team.domain包中，创建Employee类及其各子类代码
- 4. 检验代码的正确性

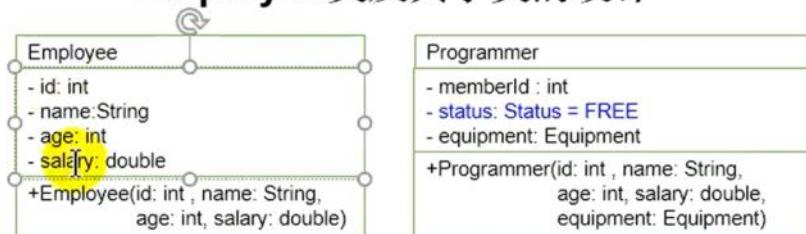
## Equipment接口及其实现子类的设计



- 说明：
  - model 表示机器的型号
  - display 表示显示器名称
  - type 表示机器的类型
- 根据需要提供各属性的get/set方法以及重载构造器
- 实现类实现接口的方法，返回各自属性的信息

让天下没有难学的

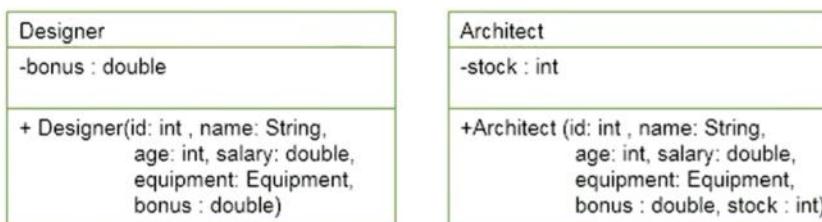
## Employee类及其子类的设计



- 说明：
  - memberId 用来记录成员加入开发团队后在团队中的ID
  - Status是项目service包下自定义的类，声明三个对象属性，分别表示成员的状态。
    - FREE-空闲
    - BUSY-已加入开发团队
    - VOCATION-正在休假
  - equipment 表示该成员领用的设备
- 可根据需要为类提供各属性的get/set方法以及重载构造器

让天下没有难学的

## Employee类及其子类的设计



- 说明：
  - bonus 表示奖金
  - stock 表示公司奖励的股票数量
- 可根据需要为类提供各属性的get/set方法以及重载构造器

## 第2步 — 实现service包中的类

1. 按照设计要求编写NameListService类
2. 在NameListService类中临时添加一个main方法中，作为单元测试方法。
3. 在方法中创建NameListService对象，然后分别用模拟数据调用该对象的各个方法，以测试是否正确。  
注：测试应细化到包含了所有非正常的情况，以确保方法完全正确。
4. 重复1-3步，完成TeamService类的开发



## TeamService类的设计

TeamService	
- counter: int = 1	
- MAX_MEMBER: final int = 5	
- team: Programmer[] = new Programmer[MAX_MEMBER];	
- total: int = 0;	
+ getTeam(): Programmer[]	
+ addMember(e: Employee) throws TeamException: void	
+ removeMember(memberId: int) throws TeamException: void	

功能：关于开发团队成员的管理：添加、删除等。

说明：

- counter为静态变量，用来为开发团队新增成员自动生成团队中的唯一ID，即 memberId。（提示：应使用增1的方式）
- MAX\_MEMBER：表示开发团队最大成员数
- team数组：用来保存当前团队中的各成员对象
- total：记录团队成员的实际人数

需求说明

如果添加操作因某种原因失败，将显示类似以下信息（失败原因视具体原因而不同）：  
1-团队列表 2-添加团队成员 3-删除团队成员 4-退出 请选择(1-4) : 2

team

添加成员

请输入要添加的员工ID : 2  
添加失败，原因：该员工已是某团队成员  
按回车键继续...

失败信息包含以下几种：

- 成员已满，无法添加
- 该成员不是开发人员，无法添加
- 该员工已在本开发团队中
- 该员工已是某团队成员
- 该员正在休假，无法添加
- 团队中至多只能有一名架构师
- 团队中至多只能有两名设计师
- 团队中至多只能有三名程序员

## TeamView类的设计

TeamView	
- listSvc: NameListService = new NameListService()	
- teamSvc: TeamService = new TeamService()	
+ enterMainMenu(): void	
- listAllEmployees(): void	
- getTeam(): void	
- addMember(): void	
- deleteMember(): void	
+ main(args: String[]): void	

- 说明：
  - listSvc和teamSvc属性：供类中的方法使用
  - enterMainMenu ()方法：主界面显示及控制方法。
  - 以下方法仅供enterMainMenu()方法调用：
    - ✓ listAllEmployees ()方法：以表格形式列出公司所有成员
    - ✓ getTeam()方法：显示团队成员列表操作
    - ✓ addMember ()方法：实现添加成员操作
    - ✓ deleteMember ()方法：实现删除成员操作

总结：

1.写完一个模块最好进行单元测试

- 2.遇到的问题，用对象数组长度遍历数组并调用方法，而不是实际长度，导致空指针异常
- 3.注意instanceof得正确使用
- 4.使用MVC设计模式
- 5.接口、类多态的使用，以及与数组的结合。
- 6.父类private，需要方法获取
- 7.注意向下转型
- 8.自定义异常的使用

# 55.java高级-IDEA使用

2021年8月10日 10:14



IntelliJ IDEA 的安装、配置与使用

## IntelliJ IDEA 的安装、配置与使用-简化版

尚硅谷 Java 研究院-宋红康

[www.atguigu.com](http://www.atguigu.com)

### 一、 IntelliJ IDEA 介绍

#### 1.JetBrains 公司介绍

IDEA(<https://www.jetbrains.com/idea/>)是 JetBrains 公司的产品，公司旗下还有其它产品，比如：

- WebStorm：用于开发 JavaScript、HTML5、CSS3 等前端技术；
- PyCharm：用于开发 python
- PhpStorm：用于开发 PHP
- RubyMine：用于开发 Ruby/Rails
- AppCode：用于开发 Objective - C/Swift
- CLion：用于开发 C/C++
- DataGrip：用于开发数据库和 SQL
- Rider：用于开发.NET
- GoLand：用于开发 Go
  
- Android Studio：用于开发 android(google 基于 IDEA 社区版进行迭代)



Check out our IDEs

**IntelliJ IDEA**  
The most Intelligent Java IDE

**PyCharm**  
Python IDE for professional developers

**WebStorm**  
The smartest JavaScript IDE

**PhpStorm**  
Lightning-smart PHP IDE

**CLion**  
A smart cross-platform IDE for C and C++

**Rider**  
Cross-platform .NET IDE

**DataGrip**  
Many databases, one tool

**RubyMine**  
The most Intelligent Ruby IDE

**AppCode**  
Smart IDE for iOS/macOS development

**Gogland**  
Up and coming Go IDE

Cross-platform .NET IDE

Many databases, one tool

The most intelligent Ruby IDE

Smart IDE for iOS/macOS development

Up and coming Go IDE

## 2. IntelliJ IDEA 介绍

IDEA，全称 IntelliJ IDEA，是 Java 语言的集成开发环境，IDEA 在业界被公认为是最好的 Java 开发工具之一，尤其在智能代码助手、代码自动提示、重构、J2EE 支持、Ant、JUnit、CVS 整合、代码审查、创新的 GUI 设计等方面的功能可以说是超常的。

IntelliJ IDEA 在 2015 年的官网上这样介绍自己：

**Excel at enterprise, mobile and web development with Java, Scala and Groovy, with all the latest modern technologies and frameworks available out of the box.**

简明翻译：IntelliJ IDEA 主要用于支持 Java、Scala、Groovy 等语言的开发工具，同时具备支持目前主流的技术和框架，擅长于企业应用、移动应用和 Web 应用的开发。

## 3. IDEA 的主要功能介绍

语言支持上：

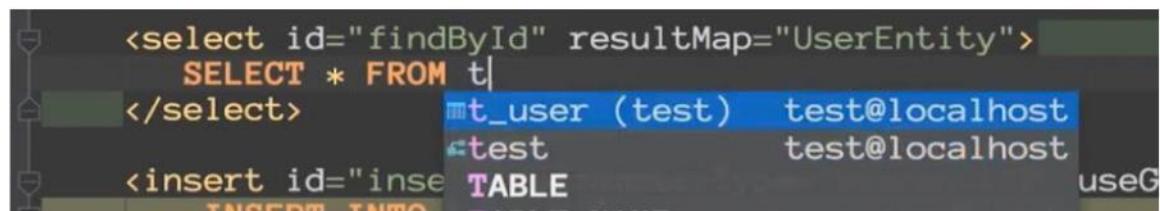
安装插件后支持	SQL类	基本JVM
PHP	PostgreSQL	Java
Python	MySQL	Groovy
Ruby	Oracle	
Scala	SQL Server	
Kotlin		
Clojure		

其他支持：

支持的框架	额外支持的语言代码提示	支持的容器
Spring MVC	HTML5	Tomcat
GWT	CSS3	TomEE
Vaadin	SASS	WebLogin
Play	LESS	JBoss
Grails	JavaScript	Jetty
Web Services	CoffeeScript	WebSphere
JSF	Node.js	
Struts	ActionScript	
Hibernate		
Flex		

## 4. IDEA 的主要优势：(相较于 Eclipse 而言)

- ① 强大的整合能力。比如：Git、Maven、Spring 等
- ② 提示功能的快速、便捷
- ③ 提示功能的范围广



```
<select id="findById" resultMap="UserEntity">
    SELECT * FROM t|
</select>      t_user (test) test@localhost
                test          test@localhost
<insert id="inse TABLE
    INSERT INTO
```

- ④ 好用的快捷键和代码模板 private static final psf
- ⑤ 精准搜索

## 5. IDEA 的下载地址：(官网)

<https://www.jetbrains.com/idea/download/#section=windows>

IDEA 分为两个版本：**旗舰版(Ultimate)**和**社区版(Community)**。

旗舰版收费(限 30 天免费试用)，社区版免费，这和 Eclipse 有很大区别。



The screenshot shows the official IntelliJ IDEA download page. It features a large 'IJ' logo on the left. Below it, there's version information: Version: 2018.1.5, Build: 181.5261.24, Released: June 13, 2018, and Release notes. To the right, there's a section titled 'Download IntelliJ IDEA' with three download links for Windows, macOS, and Linux. Below this, two boxes compare the 'Ultimate' and 'Community' editions. The 'Ultimate' edition is for 'Web, mobile and enterprise development' and includes a 'Free trial' download link. The 'Community' edition is for 'Java, Groovy, Scala and Android development' and is described as 'Free, open-source'. A detailed comparison table follows, showing features like Java, Kotlin, Groovy, Scala support; Android support; Maven, Gradle, SBT support; Git, SVN, Mercurial, CVS support; Detecting Duplicates; Performance, TFS support; JavaScript, TypeScript support; Java EE, Spring, GWT, Vaadin, Play, Grails, Other Frameworks support; and Database Tools, SQL support. Both editions have checkmarks in all categories.

License	Ultimate	Community
Commercial		Open-source, Apache 2.0
Java, Kotlin, Groovy, Scala	✓	✓
Android	✓	✓
Maven, Gradle, SBT	✓	✓
Git, SVN, Mercurial, CVS	✓	✓
Detecting Duplicates	✓	
Performance, TFS	✓	
JavaScript, TypeScript	✓	
Java EE, Spring, GWT, Vaadin, Play, Grails, Other Frameworks	✓	
Database Tools, SQL	✓	

这里提供了不同操作系统下的两个不同版本的安装文件。

两个不同版本的详细对比，可以参照官网：

[https://www.jetbrains.com/idea/features/editions\\_comparison\\_matrix.html](https://www.jetbrains.com/idea/features/editions_comparison_matrix.html)

## 6. 官网提供的详细使用文档：

<https://www.jetbrains.com/help/idea/meet-intellij-idea.html>

## 二、windows 下安装过程

### 1. 安装前的准备

#### 1.1 硬件要求(Hardware requirements)

内存: 2 GB RAM minimum, 4 GB RAM recommended

硬盘: 1.5 GB hard disk space + at least 1 GB for caches

屏幕: 1024x768 minimum screen resolution

个人建议配置: 内存 **8G** 或以上, CPU 最好 i5 以上, 最好安装块固态硬盘(**SSD**), 将 IDEA 安装在固态硬盘上, 这样流畅度会加快很多。

#### 1.2 软件要求(Software requirements)

操作系统: Microsoft Windows 10/8/7/Vista/2003/XP (32 or 64 bit)

软件环境:

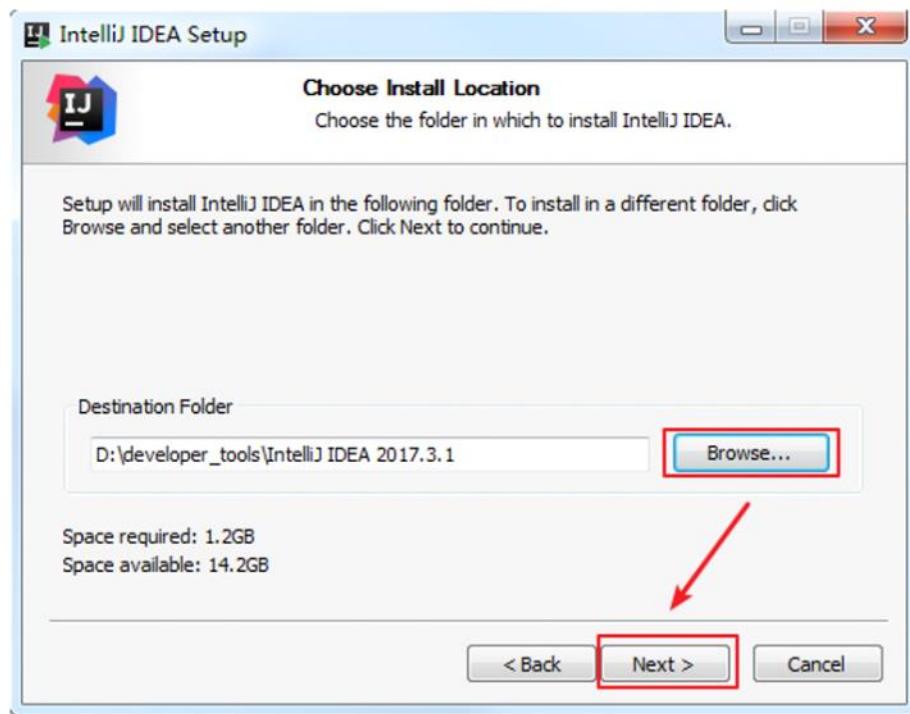
- JRE 1.8 is bundled with the IntelliJ IDEA distribution. You do not need to install Java on your computer to run IntelliJ IDEA.
- A standalone JDK is required for Java development.

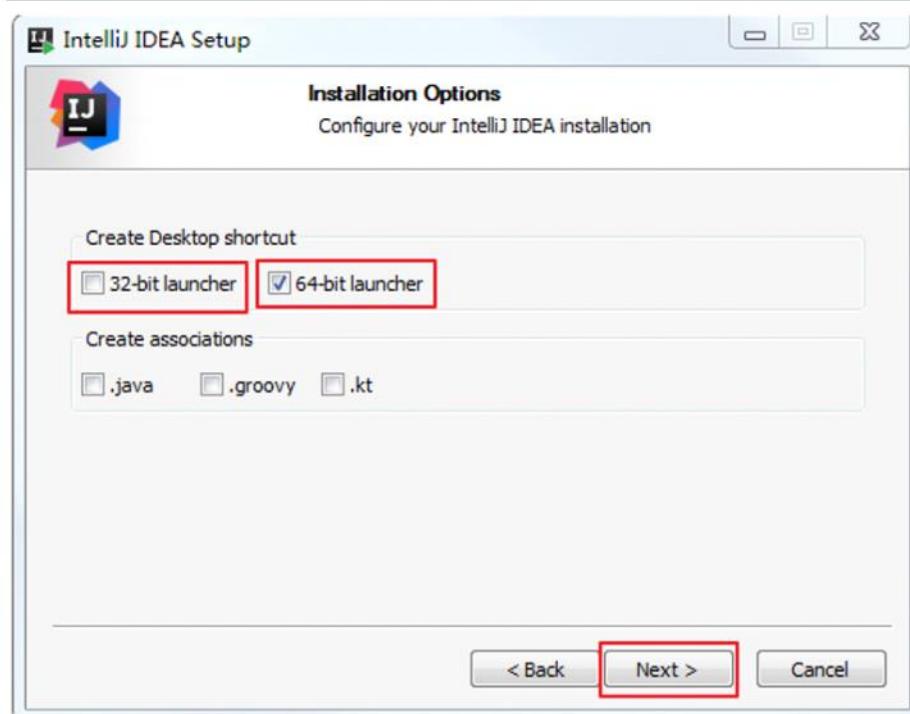
注意: 这里如果没有安装 JDK 的话, 请参考提供的文档《尚硅谷\_宋红康\_JDK8 的下载\_安装\_配置.pdf》进行安装配置。

### 2. 具体安装过程

双击:

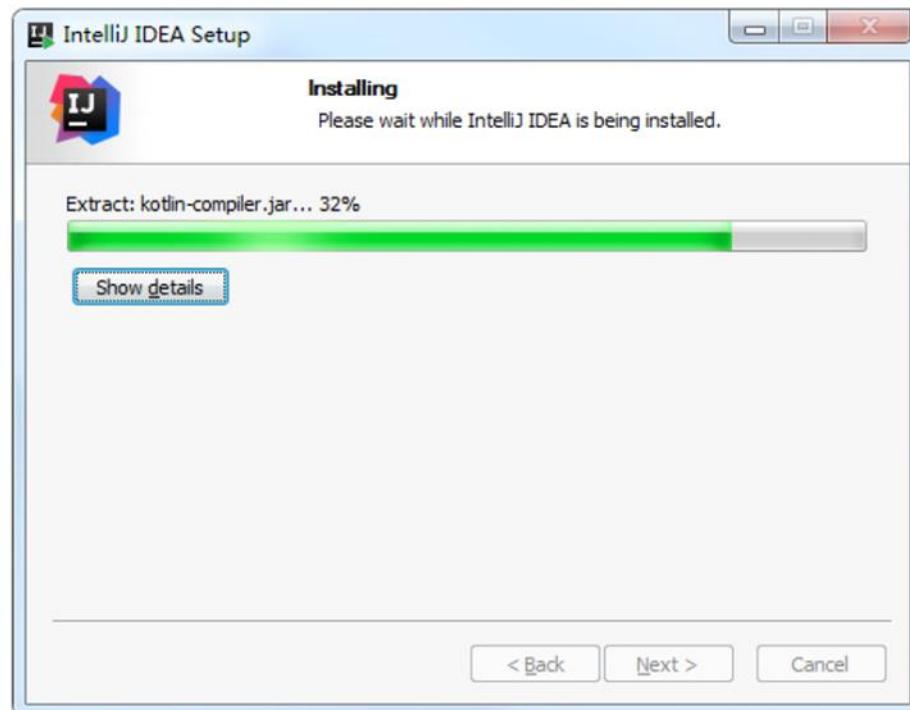






- 确认 32 位版还是 64 位版
- 确认是否与.java、.groovy、.kt 格式文件进行关联，这里也可以选择不关联。





### 3. 安装总结

从安装上来看，IntelliJ IDEA 对硬件的要求似乎不是很高。可是实际在开发中其实并不是这样的，因为 IntelliJ IDEA 执行时会有大量的缓存、索引文件，所以如果你正在使用 Eclipse / MyEclipse，想通过 IntelliJ IDEA 来解决计算机的卡、慢等问题，这基本上是不可能的，本质上你应该对自己的硬件设备进行升级。

### 4. 查看安装目录结构

developer (D:) > developer_tools > IntelliJ IDEA 2017.1.4 >			
共享 ▾ 新建文件夹			
名称	修改日期	类型	大小
bin	2017/8/22 星期...	文件夹	
help	2017/8/22 星期...	文件夹	
jre64	2017/8/22 星期...	文件夹	
lib	2017/8/22 星期...	文件夹	
license	2017/8/22 星期...	文件夹	
plugins	2017/8/22 星期...	文件夹	
redist	2017/8/22 星期...	文件夹	
build.txt	2017/6/6 星期二 ...	TXT 文件	1 KB
Install-Windows-zip.txt	2017/6/6 星期二 ...	TXT 文件	3 KB
ipr.reg	2017/6/6 星期二 ...	注册表项	1 KB

**bin:** 容器，执行文件和启动参数等

**help:** 快捷键文档和其他帮助文档

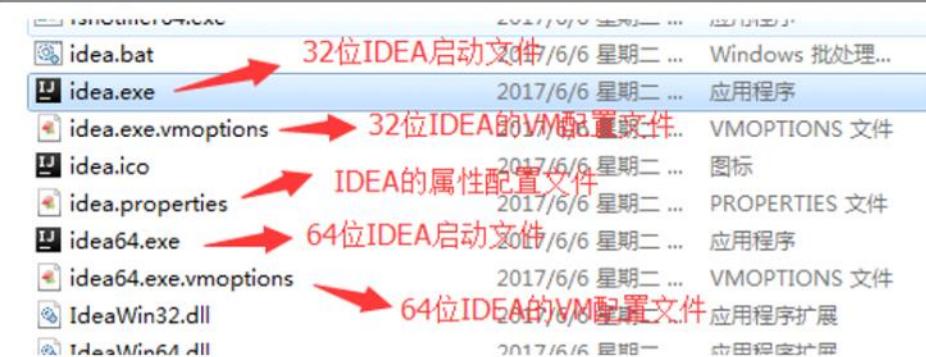
**jre64:** 64 位java 运行环境

**lib:** idea 依赖的类库

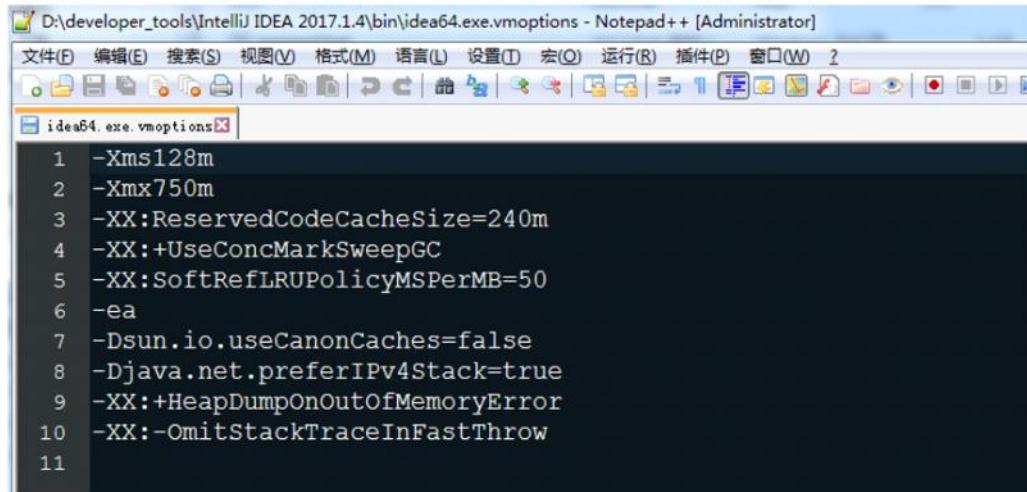
**license:** 各个插件许可

**plugin:** 插件

其中： bin 目录下：



这里以我的电脑系统(64位 windows7, 16G 内存)为例, 说明一下如何调整 VM 配置文件:



```
D:\developer_tools\IntelliJ IDEA 2017.1.4\bin\idea64.exe.vmoptions - Notepad++ [Administrator]
1 -Xms128m
2 -Xmx750m
3 -XX:ReservedCodeCacheSize=240m
4 -XX:+UseConcMarkSweepGC
5 -XX:SoftRefLRUPolicyMSPerMB=50
6 -ea
7 -Dsun.io.useCanonCaches=false
8 -Djava.net.preferIPv4Stack=true
9 -XX:+HeapDumpOnOutOfMemoryError
10 -XX:-OmitStackTraceInFastThrow
11
```

1. 大家根据电脑系统的位数, 选择 32 位的 VM 配置文件或者 64 位的 VM 配置文件
2. 32 位操作系统内存不会超过 4G, 所以没有多大空间可以调整, 建议不用调整了
3. 64 位操作系统中 8G 内存以下的机子或是静态页面开发者是无需修改的。
4. 64 位操作系统且内存大于 8G 的, 如果你是开发大型项目、Java 项目或是 Android 项目, 建议进行修改, 常修改的就是下面 3 个参数:

**-Xms128m**, 16 G 内存的机器可尝试设置为 **-Xms512m**

(设置初始的内存数, 增加该值可以提高 Java 程序的启动速度。)

**-Xmx750m**, 16 G 内存的机器可尝试设置为 **-Xmx1500m**

(设置最大内存数, 提高该值, 可以减少内存 Garage 收集的频率, 提高程序性能)

**-XX:ReservedCodeCacheSize=240m**, 16G 内存的机器可尝试设置为

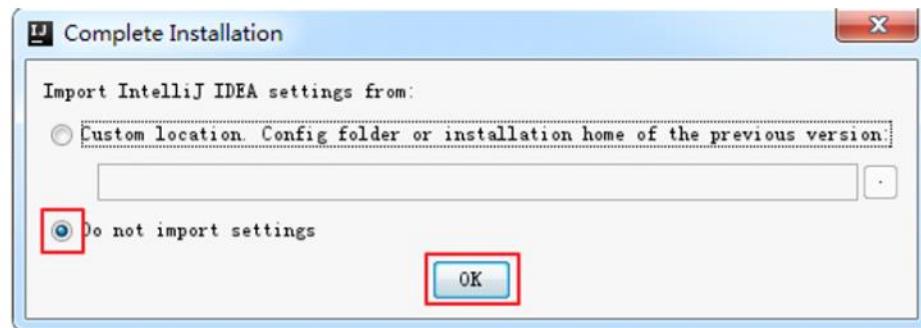
**-XX:ReservedCodeCacheSize=500m**

(保留代码占用的内存容量)

### 三、启动应用后简单配置

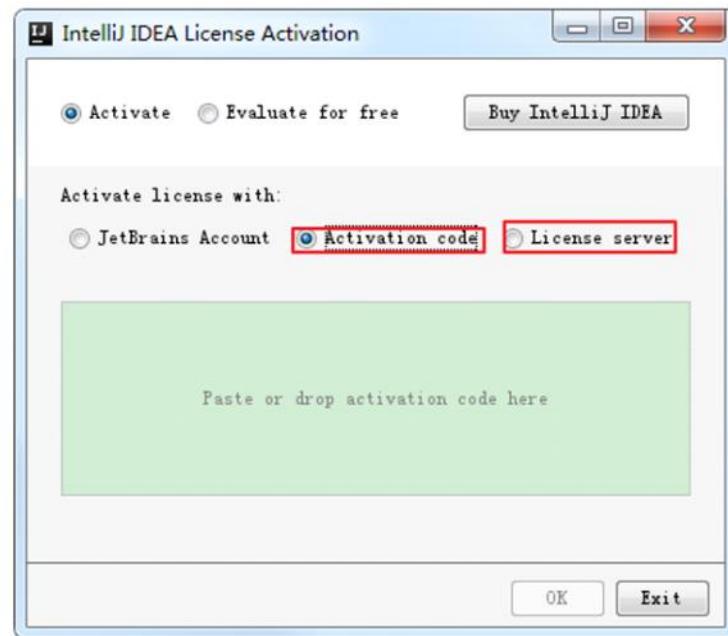
#### 1.是否导入已有设置

首次启动，会弹出如下的对话框。选择不导入已有的设置。



#### 2.激活

然后根据提供的激活文档《IDEA2017-2018\_激活方法》或百度：idea 破解码，填入：lisence server 的具体值即可。（需要联网）或者 选择 Activation code，根据文档提供的激活码，同样可以激活。（不需要联网）

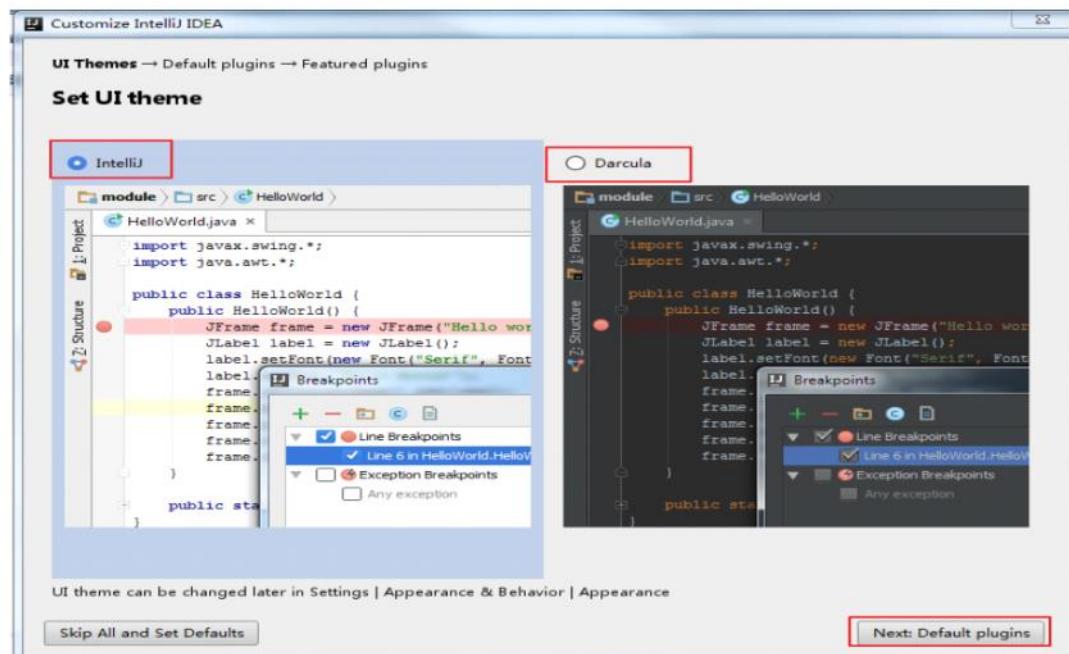


补充：

对于 IDEA 2019.2 月版本，需要按照如下方式激活：

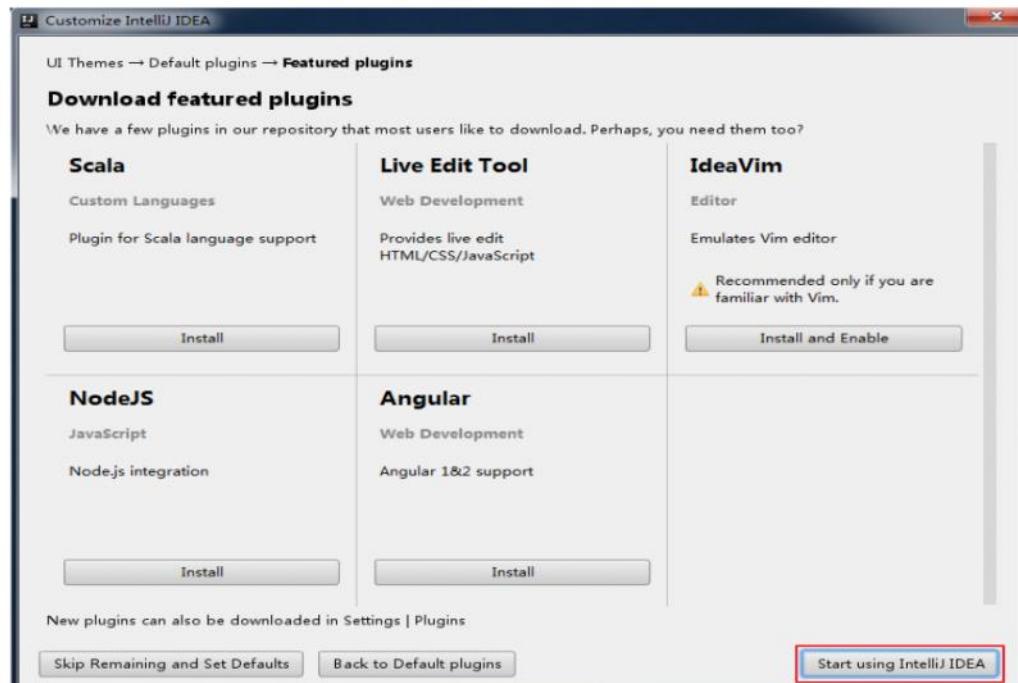
《参见 2019.2 注册文档》

### 3. 设置主题



这里根据个人喜好，进行选择，也可以选择跳过(skip all and set defaults)。后面在 settings 里也可以再设置主题等。这里选择： Next:Default plugins

## 4. 设置插件



设置 IDEA 中的各种插件，可以选择自定义设置、删除，或者安装本身不存在的插件（比如：支持 Scala 的插件）。这里不设置，后面也可以通过界面菜单栏的

settings 进行设置。

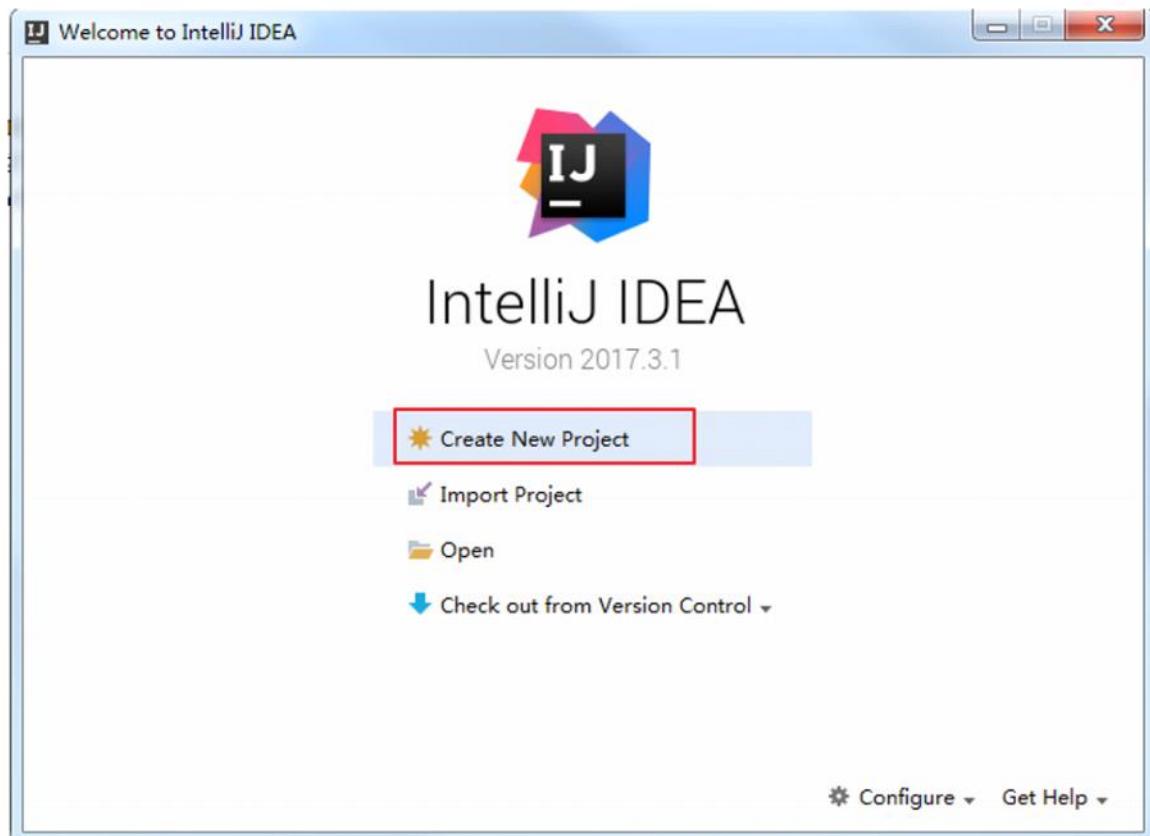
IDEA 插件官方下载地址: <https://plugins.jetbrains.com/idea>

## 5.启动页面



## 四、创建 Java 工程，运行 HelloWorld

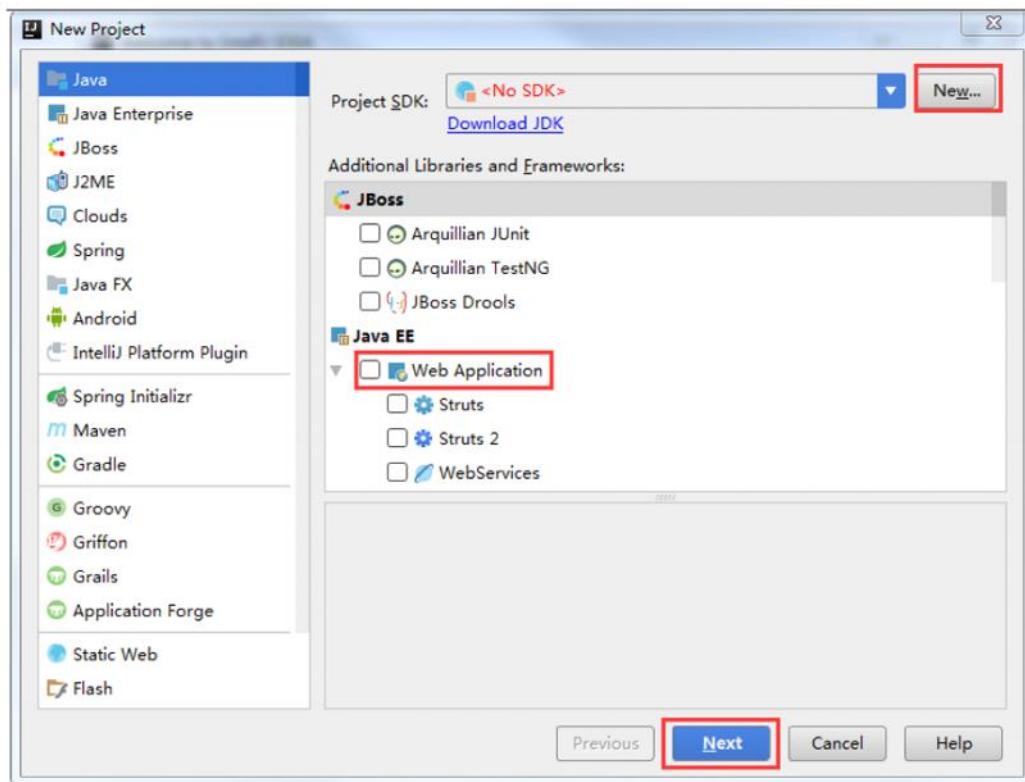
### 1. 创建 Java 工程



- Create New Project: 创建一个新的工程
- Import Project: 导入一个现有的工程
- Open: 打开一个已有工程。比如：可以打开 Eclipse 项目。
- Check out from Version Control: 可以通过服务器上的项目地址 check out Github 上面项目或其他 Git 托管服务器上的项目

这里选择 Create New Project，需要明确一下概念：

IntelliJ IDEA 没有类似 Eclipse 的工作空间的概念（Workspaces），最大单元就是 Project。这里可以把 Project 理解为 Eclipse 中的 Workspace。

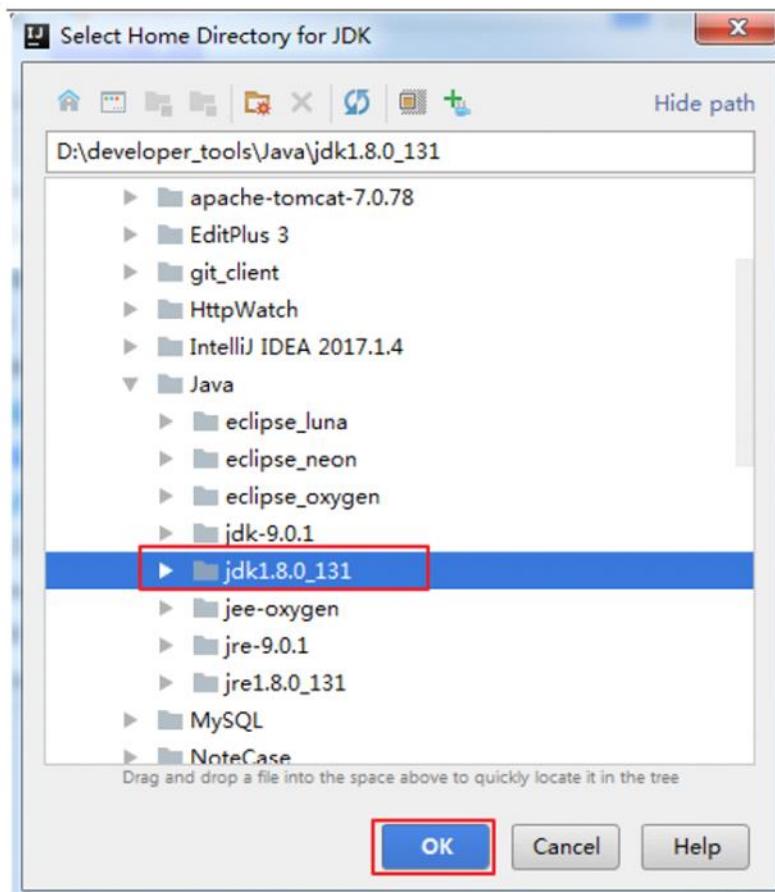


选择指定目录下的 JDK 作为 Project SDK。

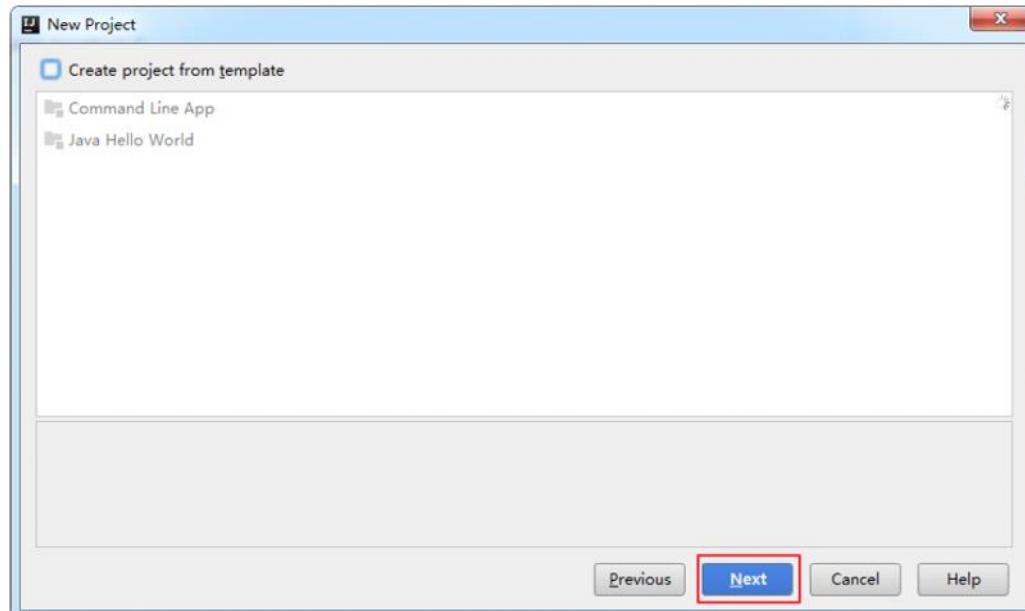
如果要创建 Web 工程，则需要勾选上面的 Web Application。如果不需创建 Web 工程的话，则不需要勾选。这里先不勾选，只是创建简单的 Java 工程。

其中，选择 New:

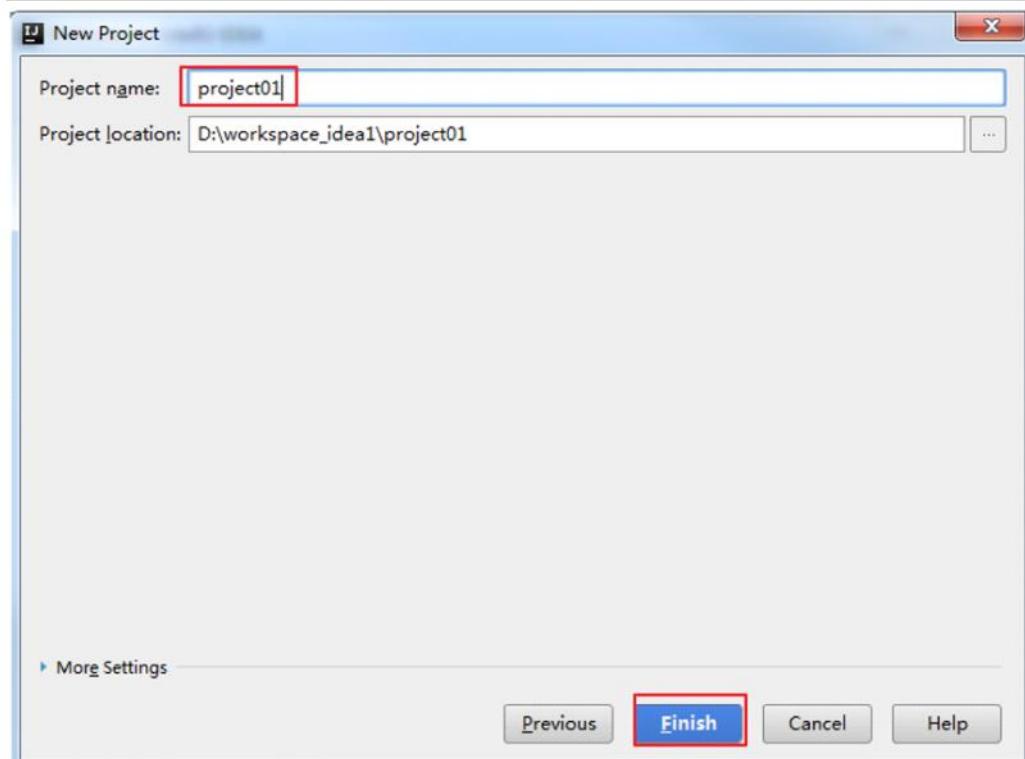
选择 jdk 的安装路径所在位置：



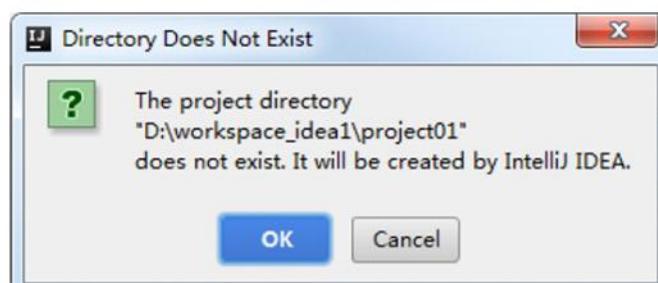
点击 OK 以后，选择 Next:



这里不用勾选。选择 Next，进入下一个页面：

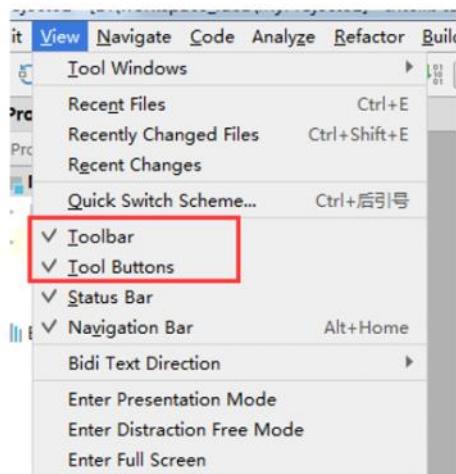


给创建的工程起一个名字，点击 finish。



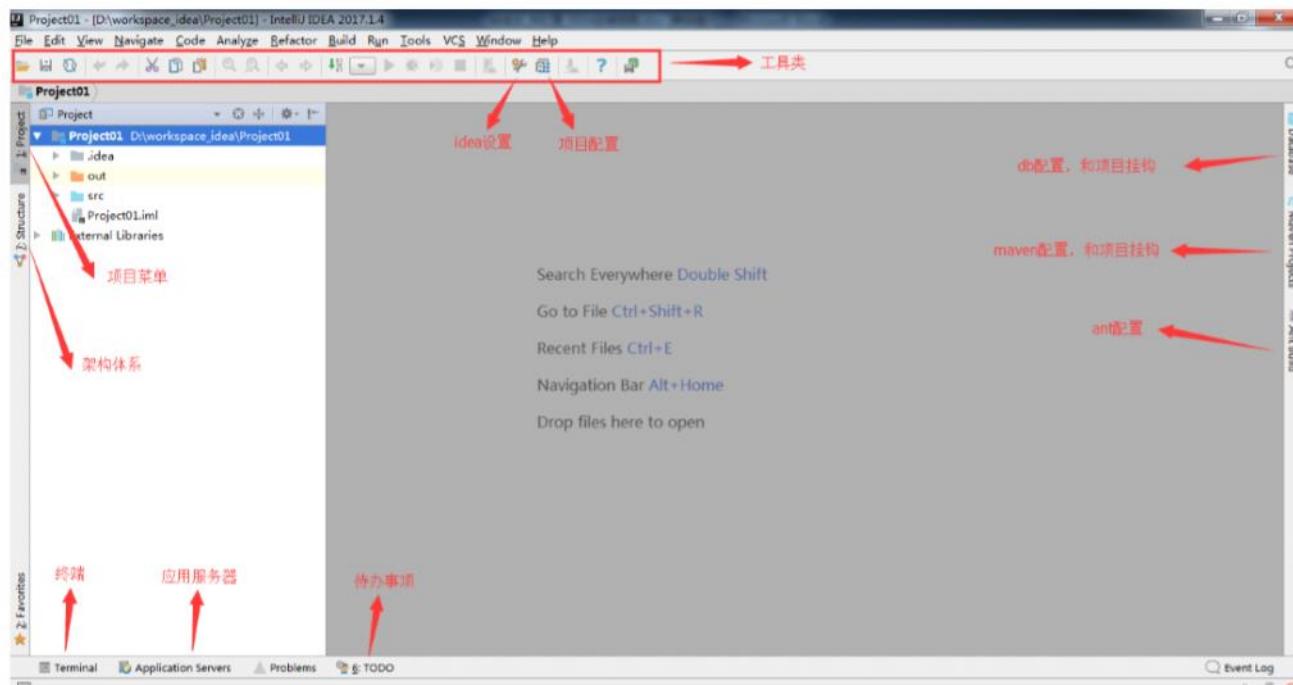
点击 OK 即可。

## 2. 设置显示常见的视图



调出工具条和按钮组

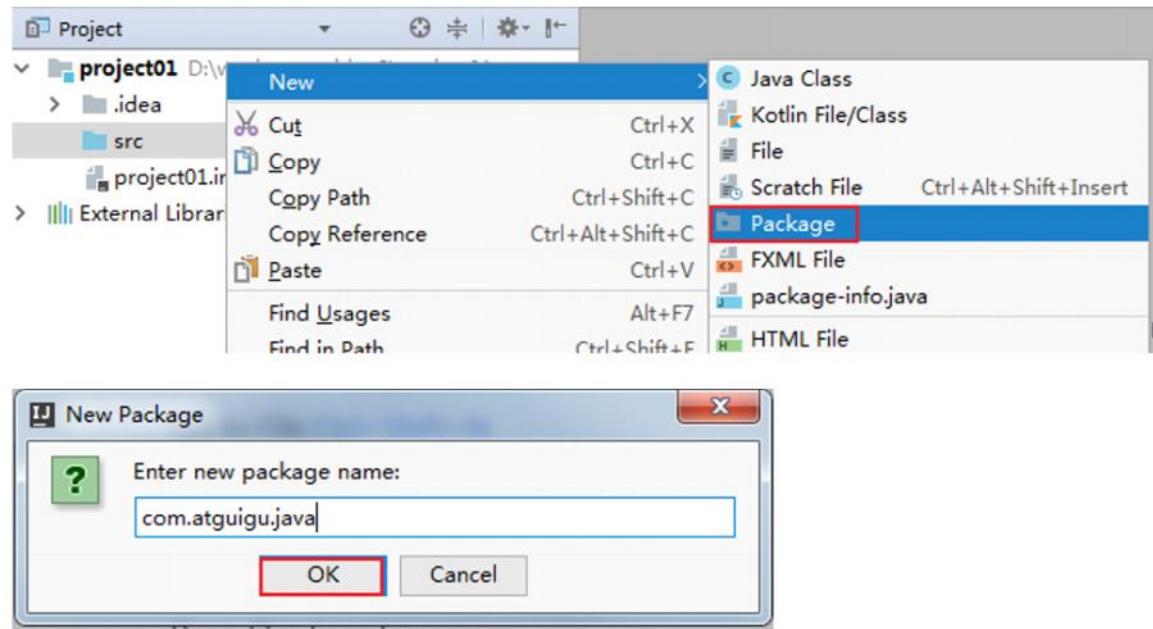
## 3. 工程界面展示



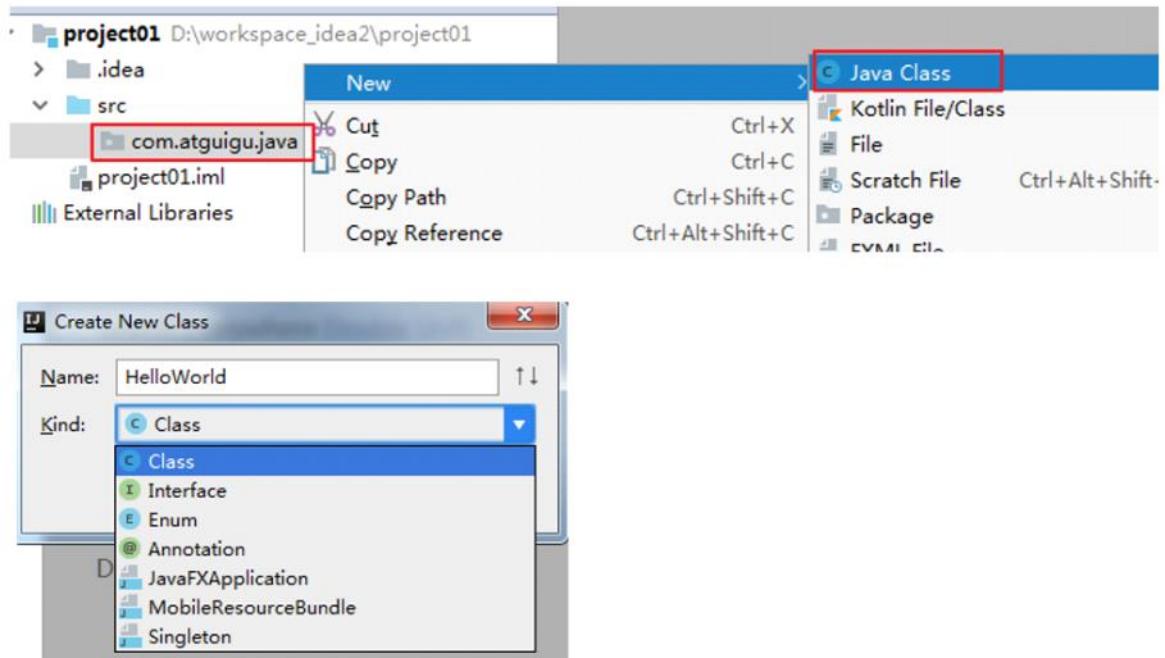
- 工程下的 src 类似于 Eclipse 下的 src 目录，用于存放代码。
- 工程下的.idea 和 project01.iml 文件都是 IDEA 工程特有的。类似于 Eclipse 工程下的.settings、.classpath、.project 等。

## 4. 创建 package 和 class

接着在 src 目录下创建一个 package:



在包下 new-class:



不管是创建 class, 还是 interface, 还是 annotation, 都是选择 new – java class,

然后在下拉框中选择创建的结构的类型。

接着在类 `HelloWorld` 里声明主方法，输出 `helloworld`，完成测试。



说明：在 IDEA 里要说的是，写完代码，不用点击保存。IDEA 会自动保存代码。

## 5. 创建模块(Module)

1. 在 Eclipse 中我们有 `Workspace` (工作空间) 和 `Project` (工程) 的概念，在 IDEA 中只有 `Project` (工程) 和 `Module` (模块) 的概念。这里的对应关系为：

**IDEA 官网说明：**

An Eclipse workspace is similar to a project in IntelliJ IDEA

An Eclipse project maps to a module in IntelliJ IDEA

**翻译：**

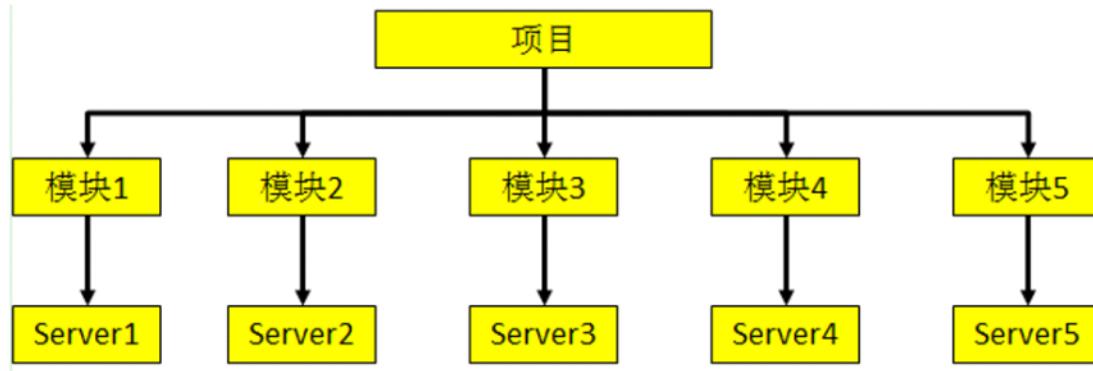
Eclipse 中 `workspace` 相当于 IDEA 中的 `Project`

Eclipse 中 `Project` 相当于 IDEA 中的 `Module`

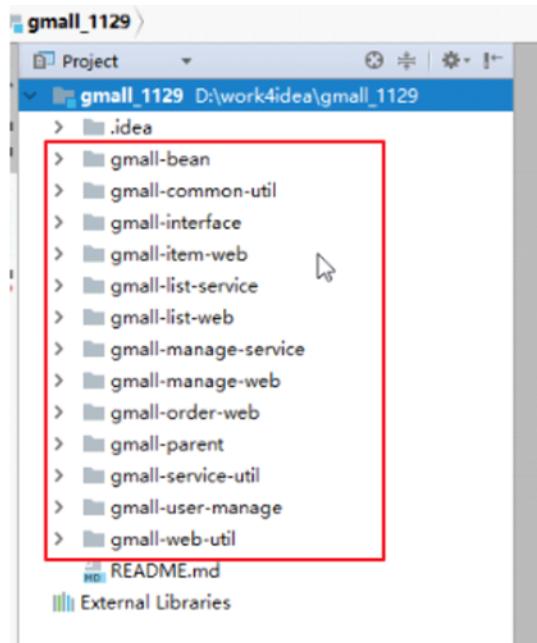
这个地方刚开始用的时候会很容易理不清它们之间的关系。

2. 从 Eclipse 转过来的人总是下意识地要在同一个窗口管理  $n$  个项目，这在 IntelliJ IDEA 是无法做到的。IntelliJ IDEA 提供的解决方案是打开多个项目实例，即打开多个项目窗口。即：一个 `Project` 打开一个 `Window` 窗口。
3. 在 IntelliJ IDEA 中 `Project` 是最顶级的级别，次级别是 `Module`。一个 `Project`

可以有多个 **Module**。目前主流的大型项目都是分布式部署的，结构都是类似这种多 **Module** 结构。



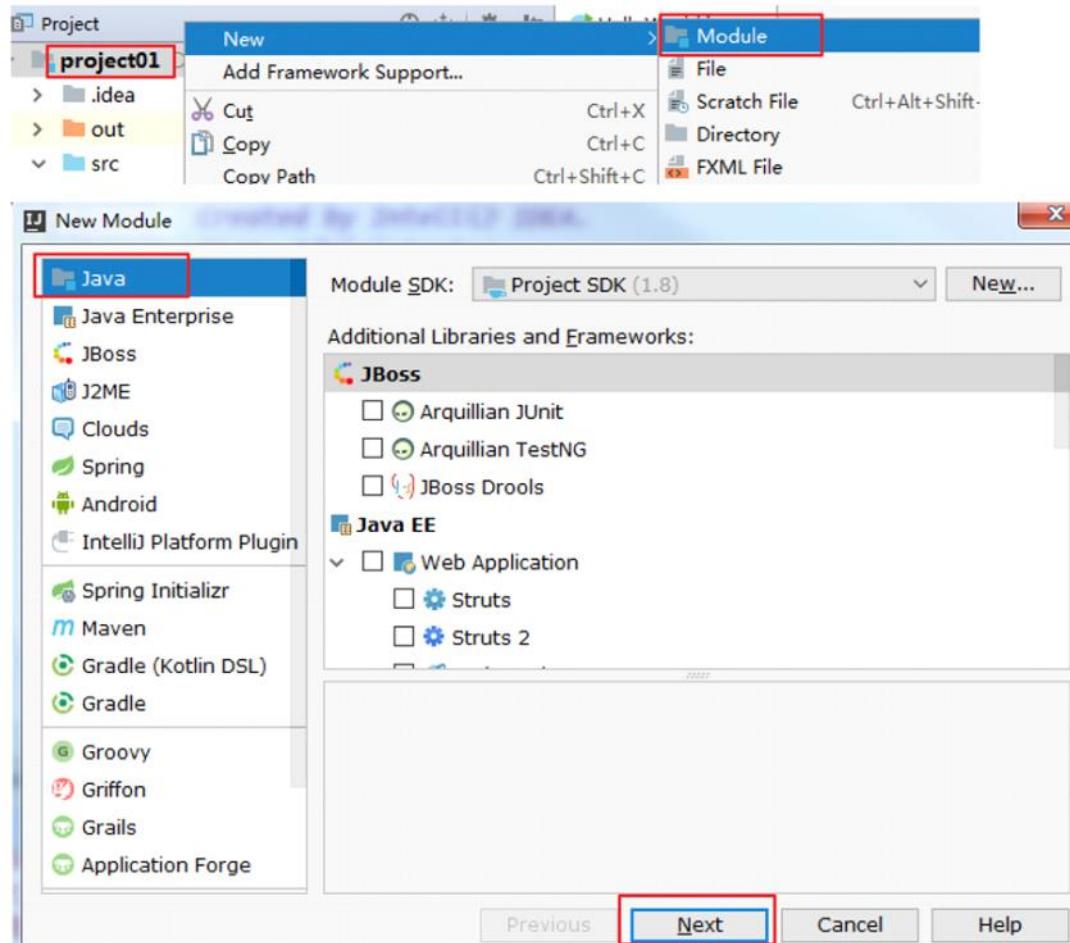
这类项目一般是这样划分的，比如：`core Module`、`web Module`、`plugin Module`、`solr Module` 等等，模块之间彼此可以相互依赖。通过这些 **Module** 的命名也可以看出，他们之间都是处于同一个项目业务下的模块，彼此之间是有不可分割的业务关系的。举例：



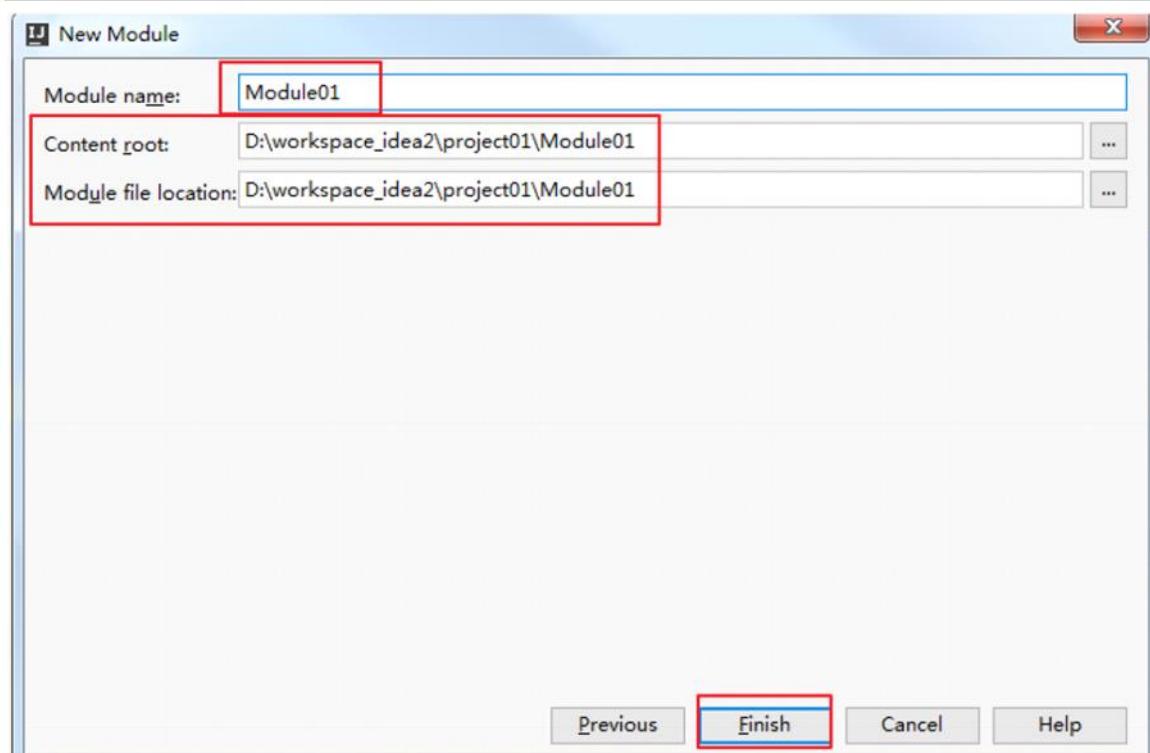
4. 相比较于多 **Module** 项目，小项目就无需搞得这么复杂。只有一个 **Module** 的结构 IntelliJ IDEA 也是支持的，并且 IntelliJ IDEA 创建项目的时候，默认就是单

Module 的结构的。

下面，我们演示如何创建 Module:

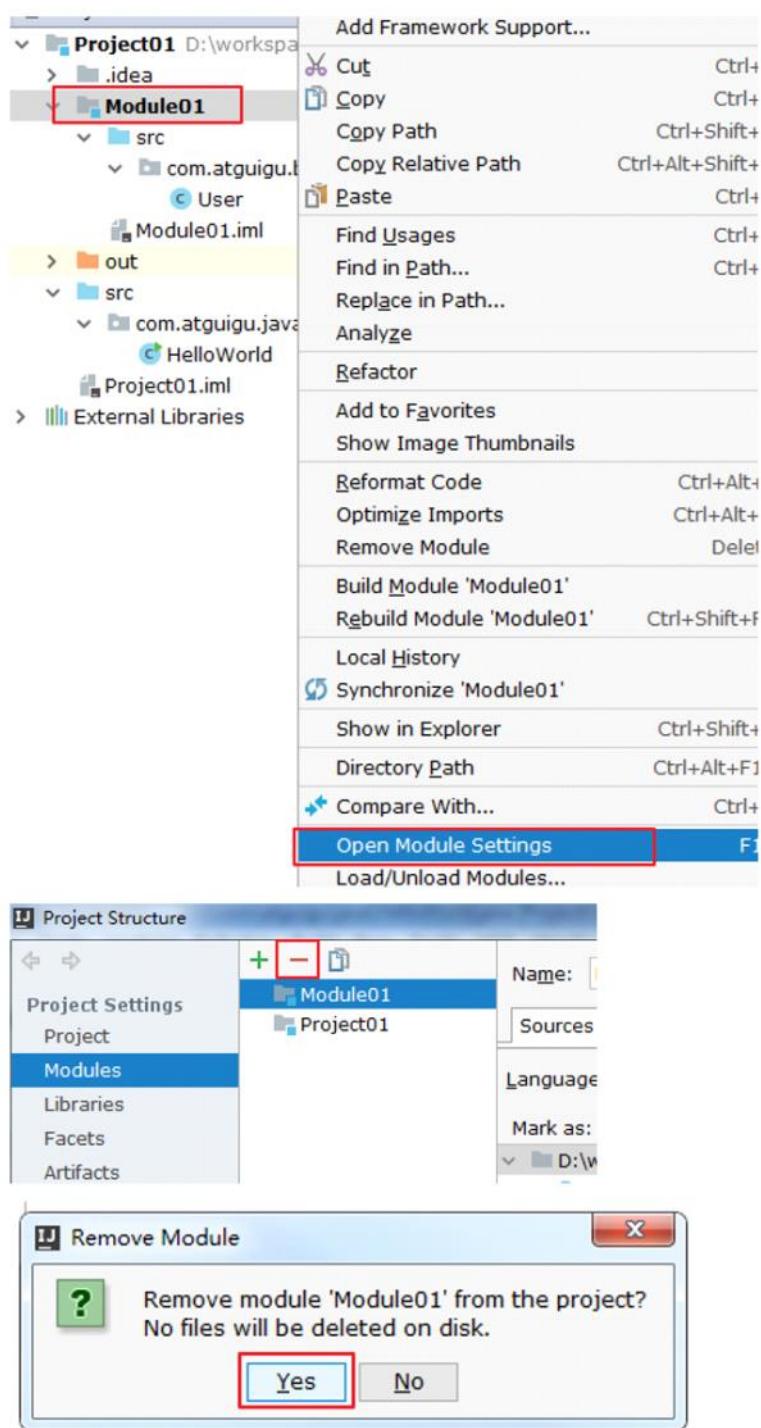


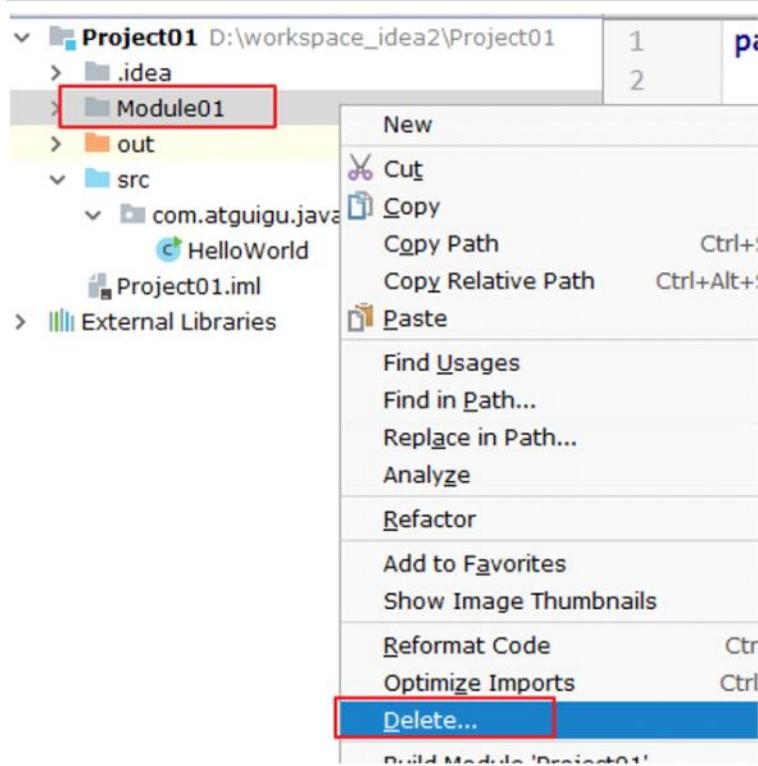
接着选择 Next:



之后，我们可以在 Module 的 src 里写代码，此时 Project 工程下的 src 就没什么用了。可以删掉。

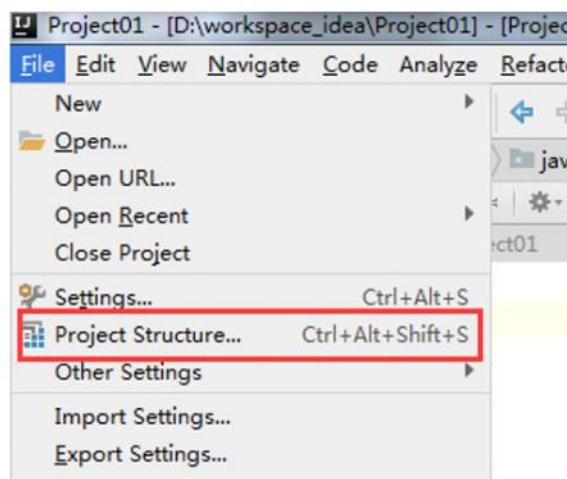
## 6. 如何删除模块



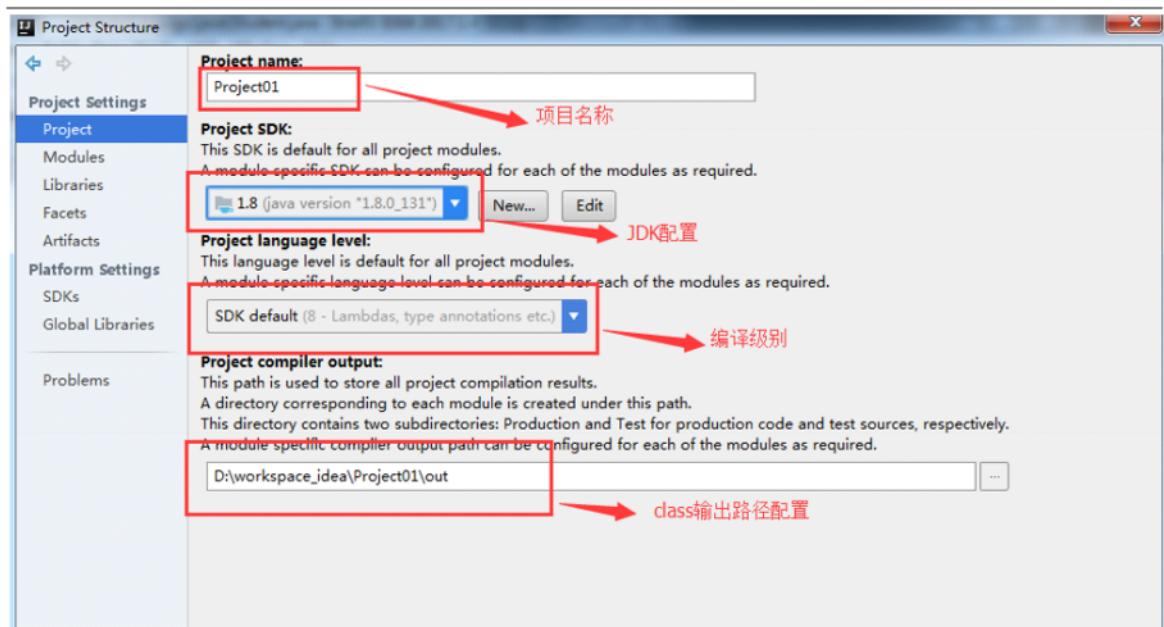


此时的删除，会从硬盘上将此 module 删除掉。

## 7.查看项目配置



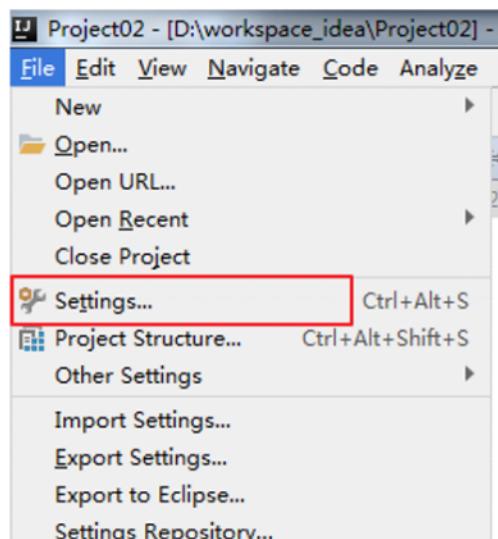
进入项目结构：



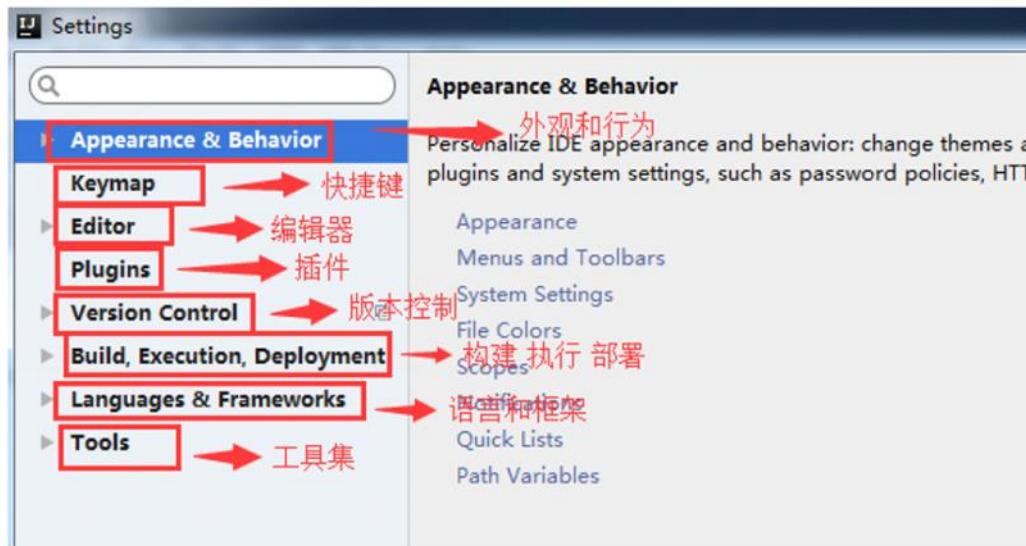
## 五、常用配置

IntelliJ IDEA 有很多人性化的设置我们必须单独拿出来讲解，也因为这些人性化的设置让那些 IntelliJ IDEA 死忠粉更加死心塌地使用它和分享它。

进入设置界面：

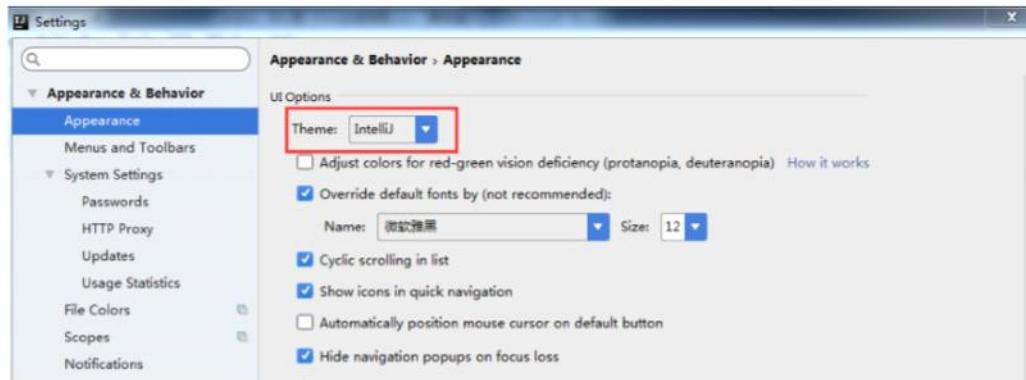


目录结构如下：



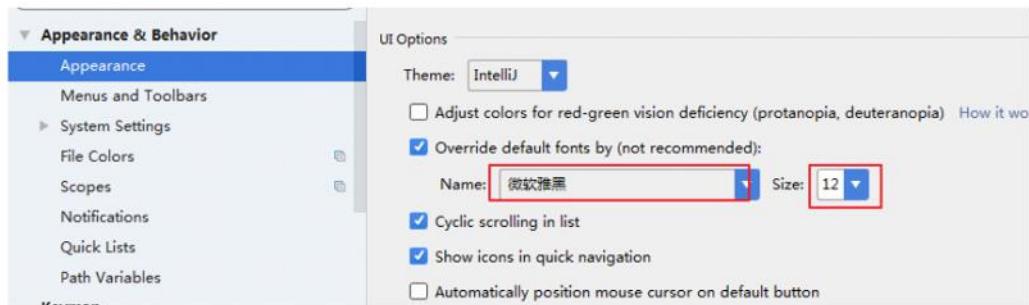
## 1.Appearance & Behavior

### 1.1 设置主题



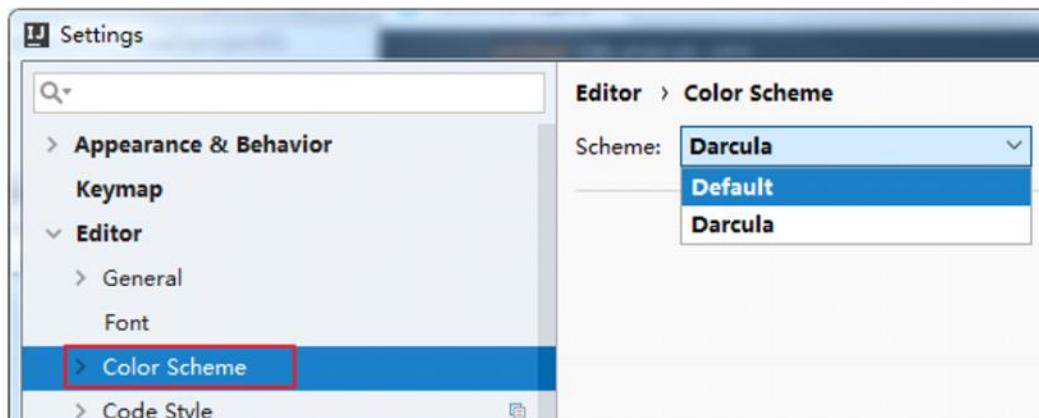
这里默认提供了三套主题：IntelliJ, Darcula, Windows。这里可以根据自己的喜好进行选择。

## 1.2 设置窗体及菜单的字体及字体大小 (可忽略)

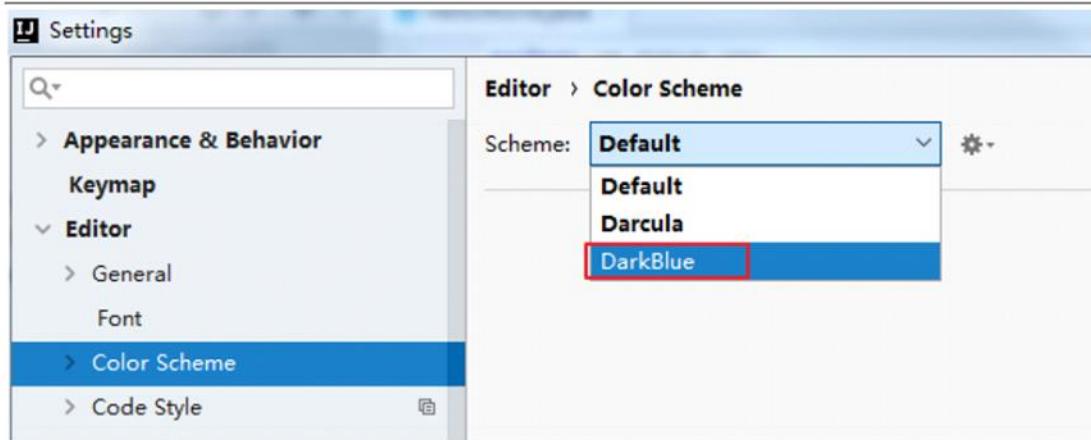


## 1.3 补充:设置编辑区主题 (可忽略)

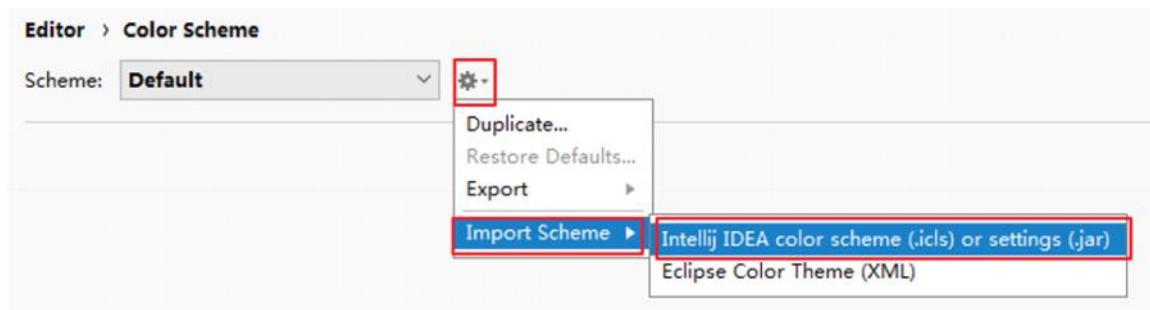
IDEA 默认提供了两个编辑区主题，可以通过如下的方式进行选择。



- 如果想要更多的主题效果的话，可以到如下的网站下载：  
<http://www.riaway.com/>
- 下载以后，导入主题：（方式一）  
file -> import settings -> 选中下载的主题 jar 文件 -> 一路确认 -> 重启。  
重启以后，新主题会自动启用。如果没有启用，可以如下方式选择：

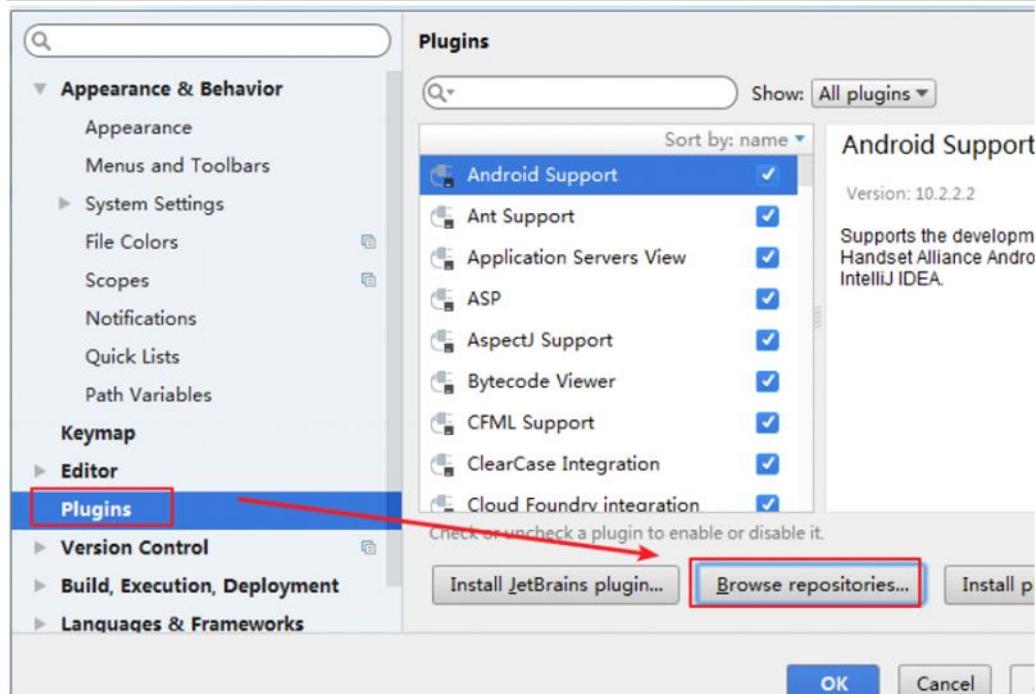


➤ 下载以后，导入主题：（方式二）

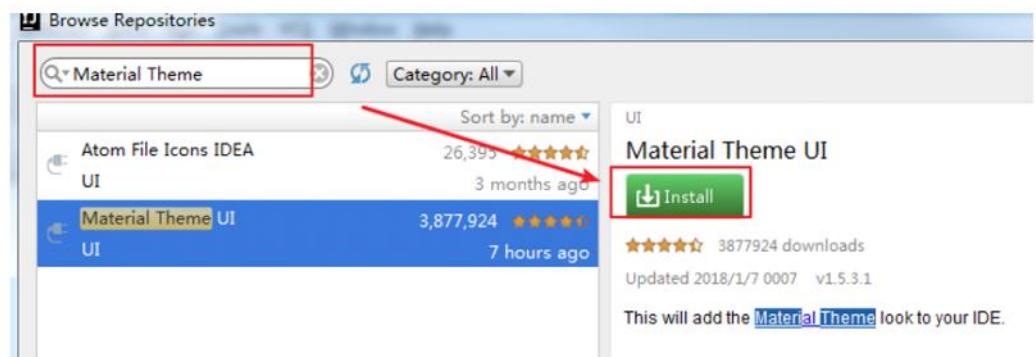


## 1.4 补充:通过插件(plug-ins)更换主题

喜欢黑色主题的话，还可以下载插件：Material Theme UI



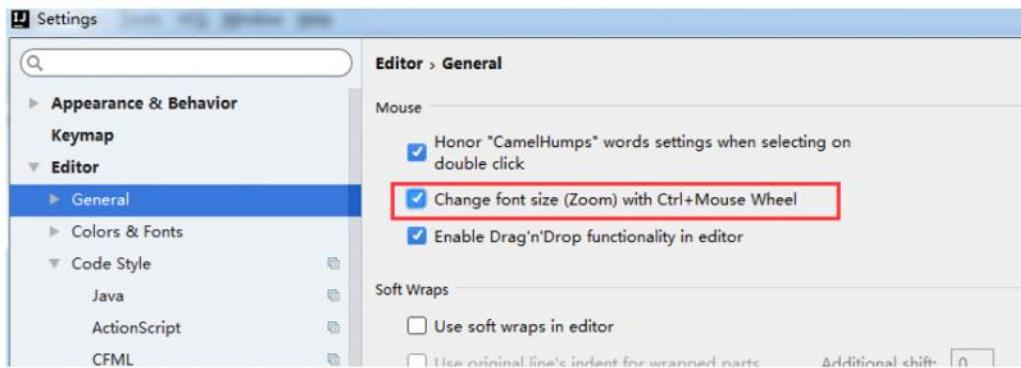
点击按钮以后，在联网环境下搜索如下的插件-安装-重启 IDEA 即可：



如果对安装的主题插件不满意，还可以找到此插件，进行卸载 - 重启 IDEA 即可。

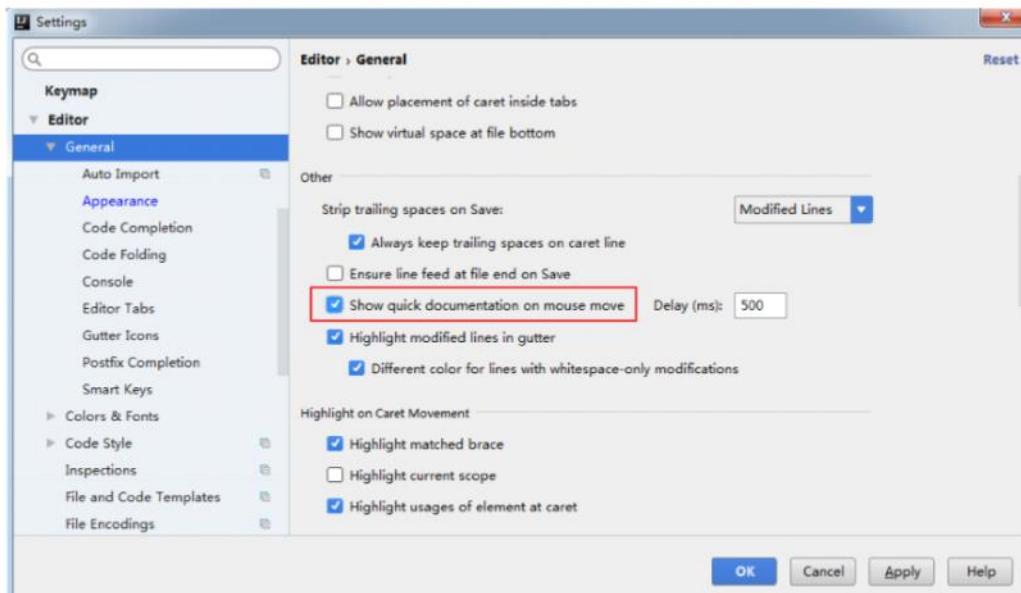
## 2. Editor - General

### 2.1 设置鼠标滚轮修改字体大小(可忽略)

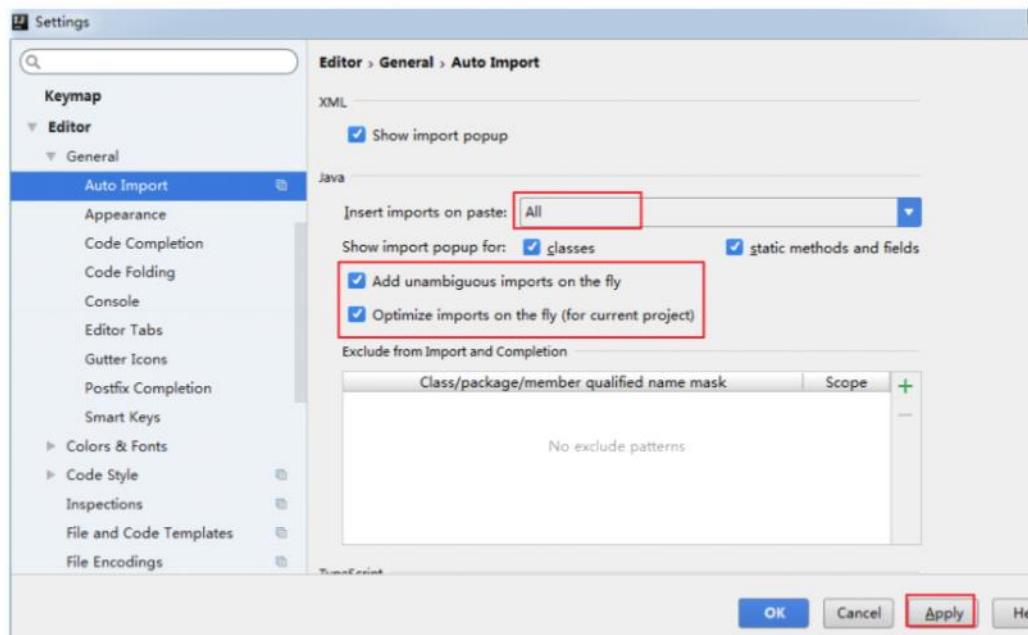


我们可以勾选此设置后，增加 `Ctrl + 鼠标滚轮` 快捷键来控制代码字体大小显示。

### 2.2 设置鼠标悬浮提示

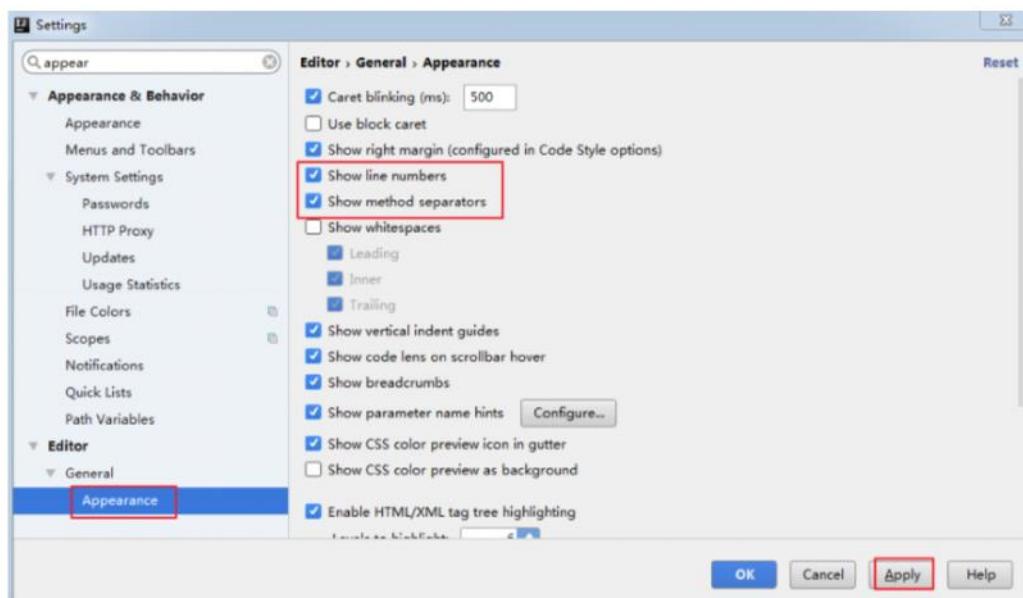


## 2.3 设置自动导包功能



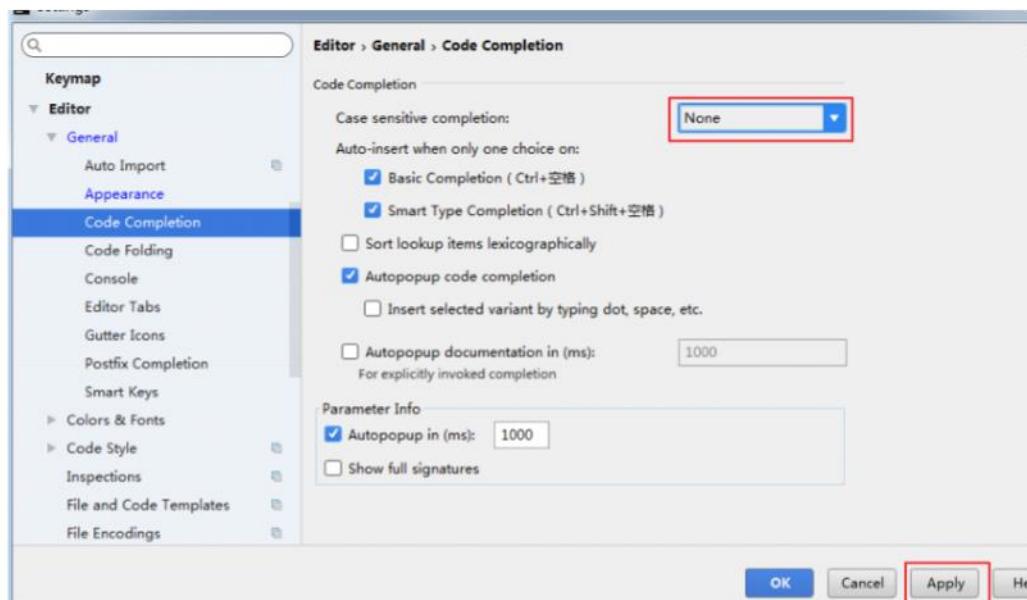
- Add unambiguous imports on the fly: 自动导入不明确的结构
- Optimize imports on the fly: 自动帮我们优化导入的包

## 2.4 设置显示行号和方法间的分隔符



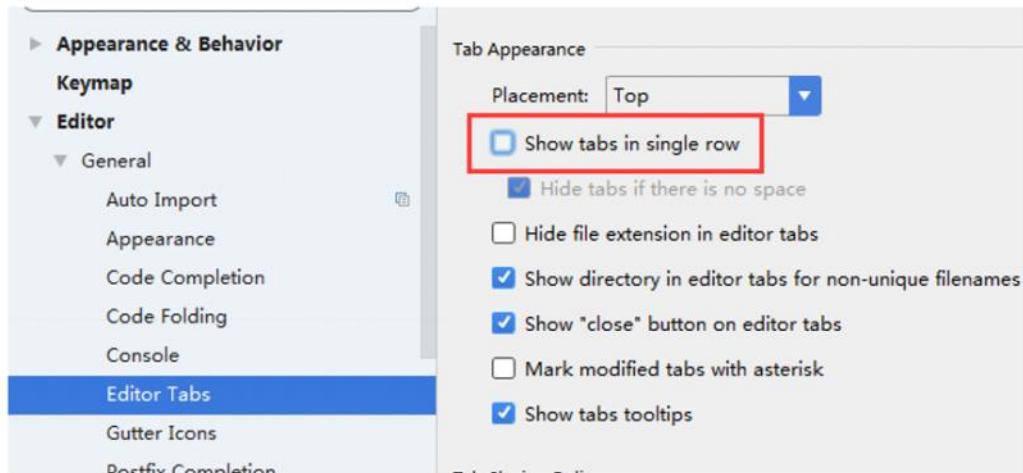
- 如上图红圈所示，可以勾选 **Show line numbers**: 显示行数。我建议一般这个要勾选上。
- 如上图红圈所示，可以勾选 **Show method separators**: 显示方法分隔线。这种线有助于我们区分开方法，所以建议勾选上。

## 2.5 忽略大小写提示



- IntelliJ IDEA 的代码提示和补充功能有一个特性：区分大小写。如上图标注所示，默认就是 **First letter** 区分大小写的。
- 区分大小写的情况是这样的：比如我们在 Java 代码文件中输入 `stringBuffer`，IntelliJ IDEA 默认是不会帮我们提示或是代码补充的，但是如果我们将输入改为 `StringBuffer` 就可以进行代码提示和补充。
- 如果想不区分大小写的话，改为 **None** 选项即可。

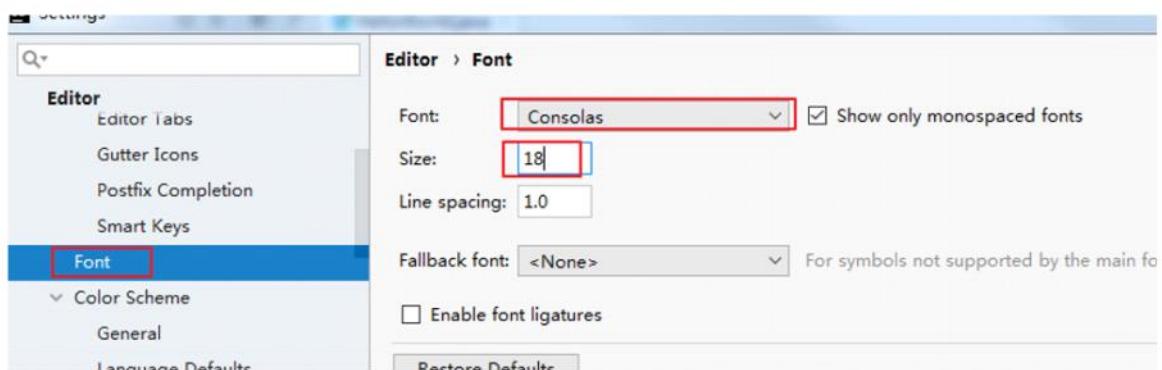
## 2.6 设置取消单行显示 tabs 的操作



如上图标注所示，在打开很多文件的时候，IntelliJ IDEA 默认是把所有打开的文件名 Tab 单行显示的。但是我个人现在的习惯是使用多行，多行效率比单行高，因为单行会隐藏超过界面部分 Tab，这样找文件不方便。

## 3. Editor - Font

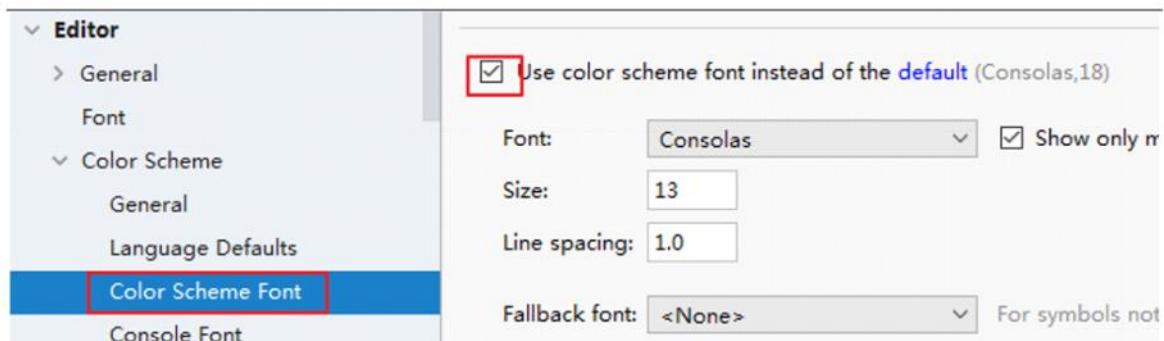
### 3.1 设置默认的字体、字体大小、字体行间距



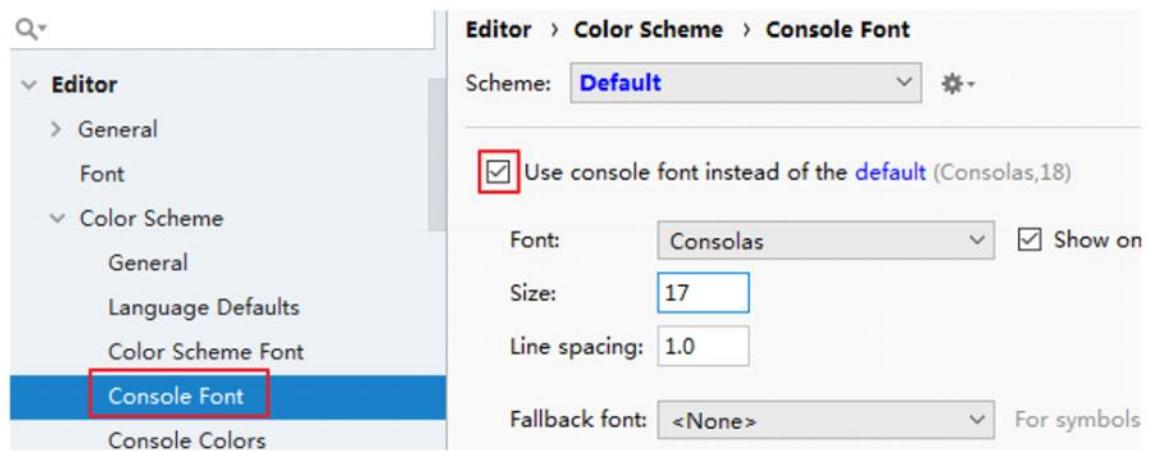
## 4. Editor - Color Scheme

### 4.1 修改当前主题的字体、字体大小、字体行间距(可忽略)

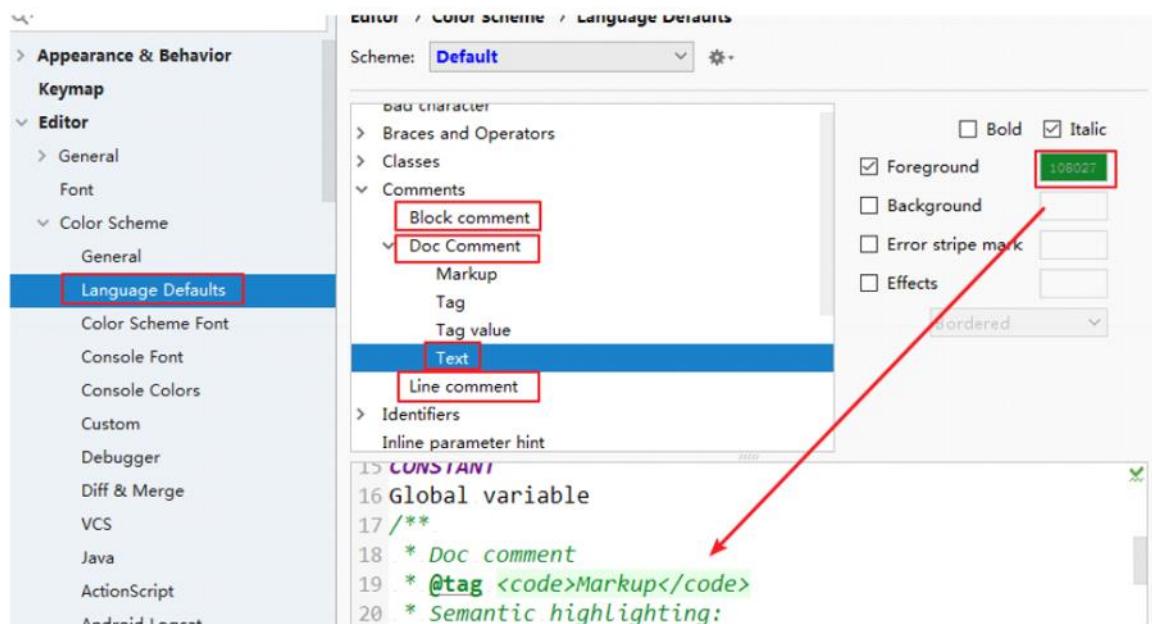
如果当前主题不希望使用默认字体、字体大小、字体行间距，还可以单独设置：



## 4.2 修改当前主题的控制台输出的字体及字体大小(可忽略)



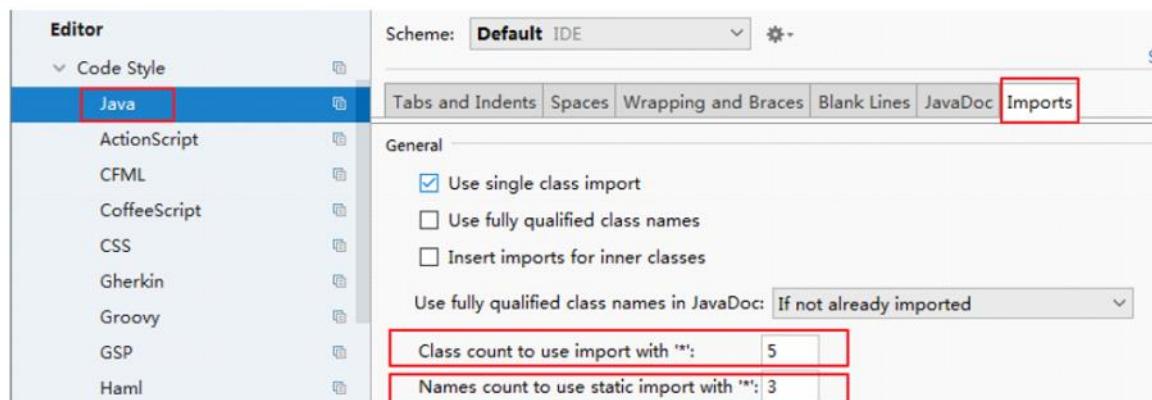
## 4.3 修改代码中注释的字体颜色



- Doc Comment – Text: 修改文档注释的字体颜色
- Block comment: 修改多行注释的字体颜色
- Line comment: 修改单行注释的字体颜色

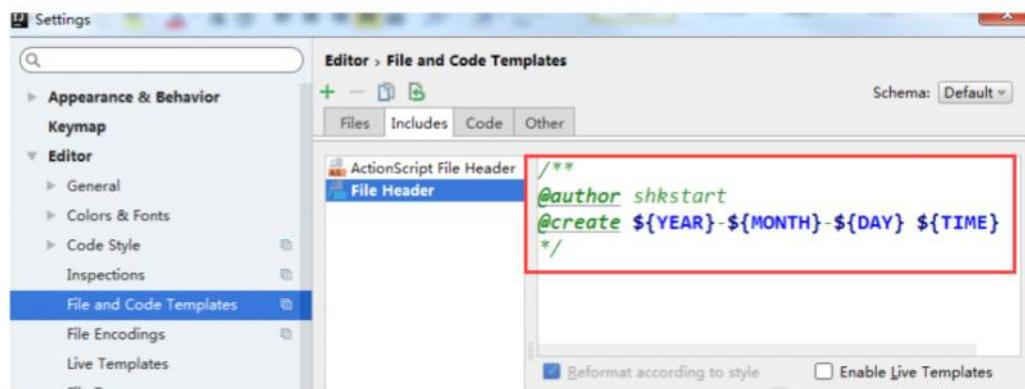
## 5. Editor – Code Style

### 5.1 设置超过指定 import 个数，改为\* (可忽略)



## 6. Editor – File and Code Templates

### 6.1 修改类头的文档注释信息



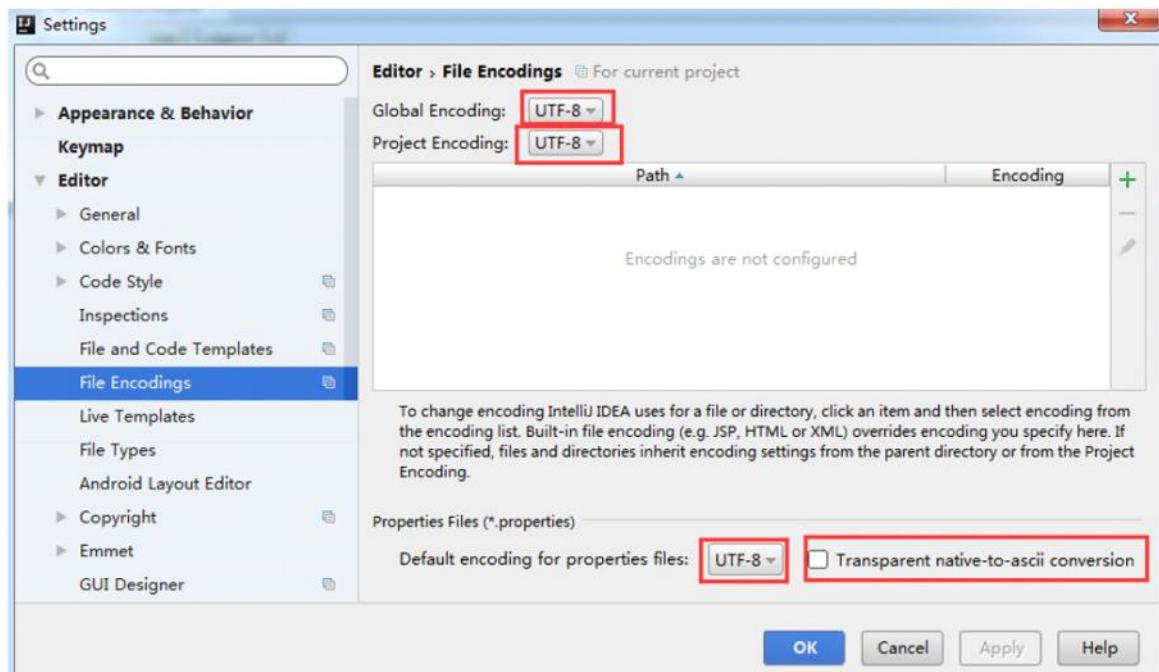
```
/**  
 * @author shkstart  
 * @create ${YEAR}-${MONTH}-${DAY} ${TIME}  
 */
```

常用的预设的变量，这里直接贴出官网给的：

```
 ${PACKAGE_NAME} - the name of the target package where the new class or interface will be created.  
 ${PROJECT_NAME} - the name of the current project.  
 ${FILE_NAME} - the name of the PHP file that will be created.  
 ${NAME} - the name of the new file which you specify in the New File dialog box during the file creation.  
 ${USER} - the login name of the current user.  
 ${DATE} - the current system date.  
 ${TIME} - the current system time.  
 ${YEAR} - the current year.  
 ${MONTH} - the current month.  
 ${DAY} - the current day of the month.  
 ${HOUR} - the current hour.  
 ${MINUTE} - the current minute.  
 ${PRODUCT_NAME} - the name of the IDE in which the file will be created.  
 ${MONTH_NAME_SHORT} - the first 3 letters of the month name. Example: Jan, Feb, etc.  
 ${MONTH_NAME_FULL} - full name of a month. Example: January, February, etc.
```

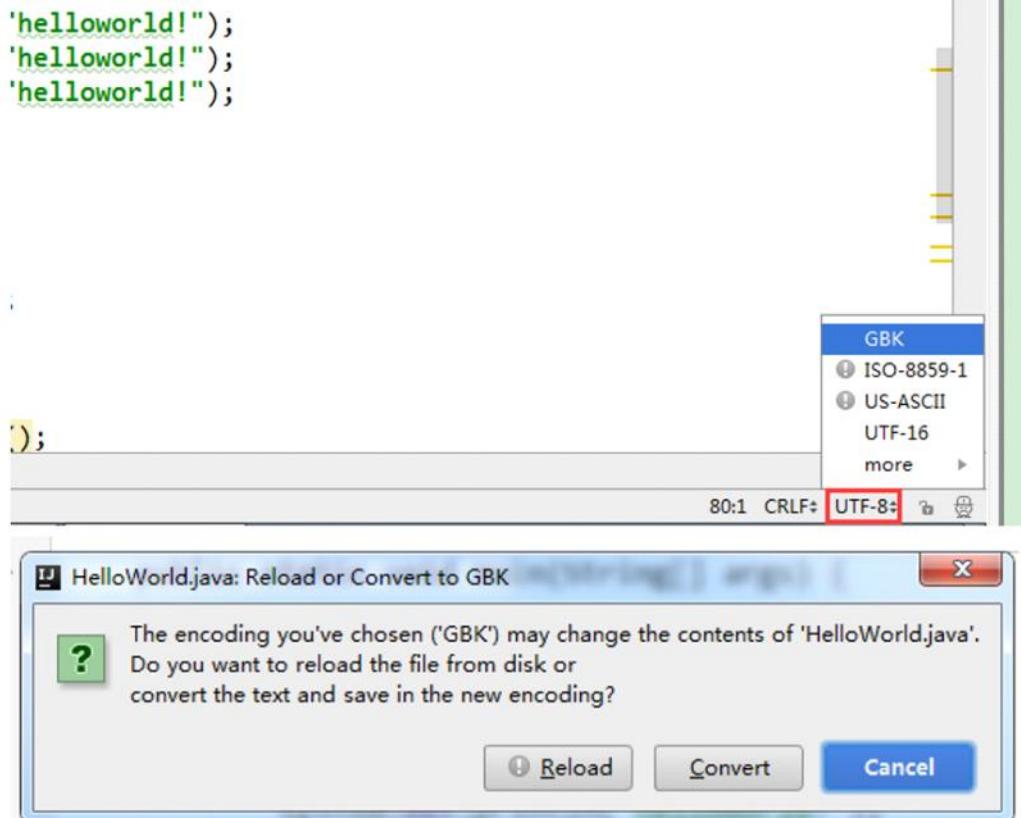
## 7. Editor – File Encodings

### 7.1 设置项目文件编码



说明：Transparent native-to-ascii conversion 主要用于转换 ascii，一般都要勾选，不然 Properties 文件中的注释显示的都不会是中文。

## 7.2 设置当前源文件的编码(可忽略)

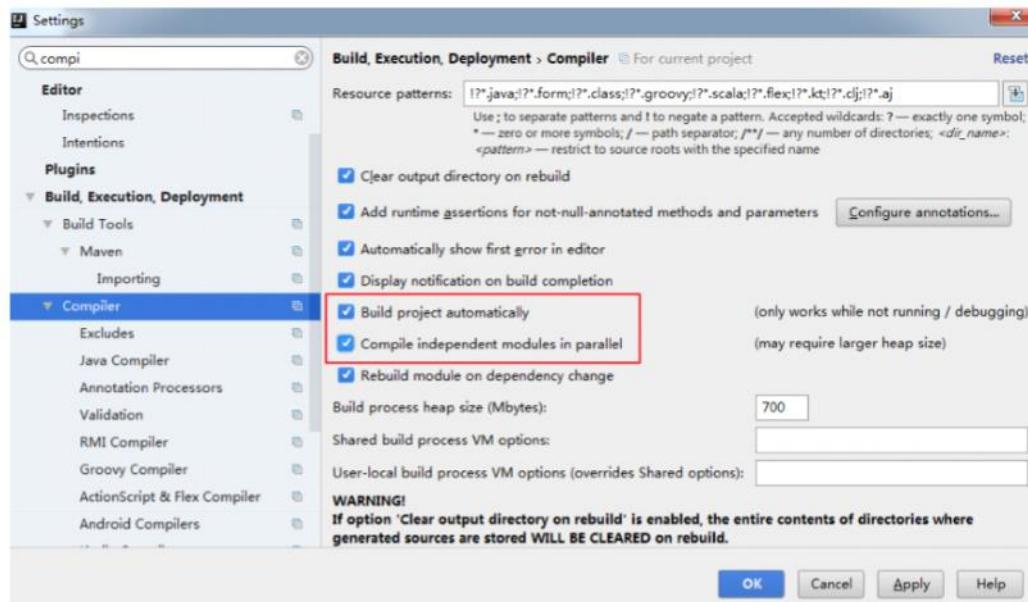


对单独文件的编码修改还可以点击右下角的编码设置区。如果代码内容中包含中文，则会弹出如上的操作选择。其中：

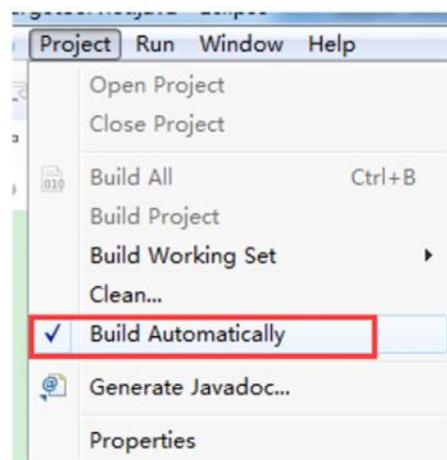
- ① **Reload** 表示使用新编码重新加载，新编码不会保存到文件中，重新打开此文件，旧编码是什么依旧还是什么。
- ② **Convert** 表示使用新编码进行转换，新编码会保存到文件中，重新打开此文件，新编码是什么则是什么。
- ③ 含有中文的代码文件，**Convert** 之后可能会使中文变成乱码，所以在转换成请做好备份，不然可能出现转换过程变成乱码，无法还原。

## 8. Build, Execution, Deployment

### 8.1 设置自动编译

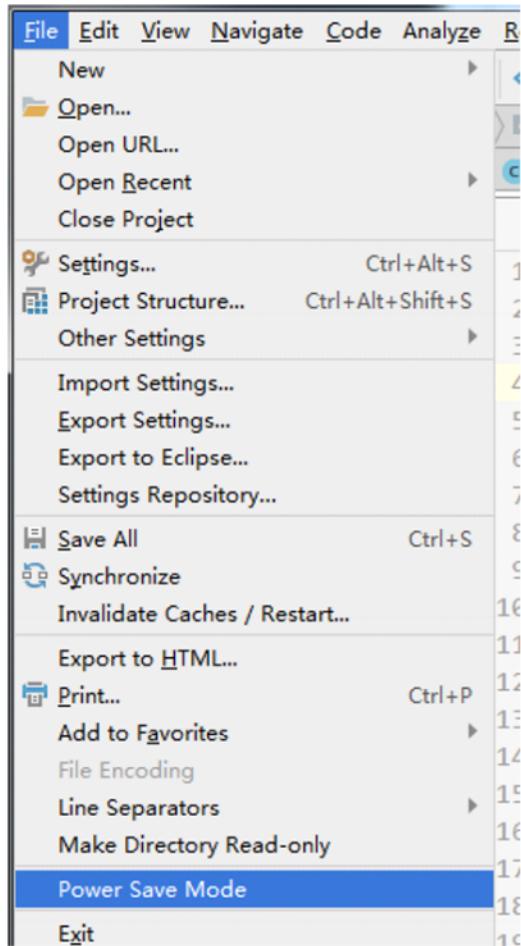


- 构建就是以我们编写的 java 代码、框架配置文件、国际化等其他资源文件、JSP 页面和图片等资源作为“原材料”，去“生产”出一个可以运行的项目的过程。
- IntelliJ Idea 默认状态为不自动编译状态，Eclipse 默认为自动编译：



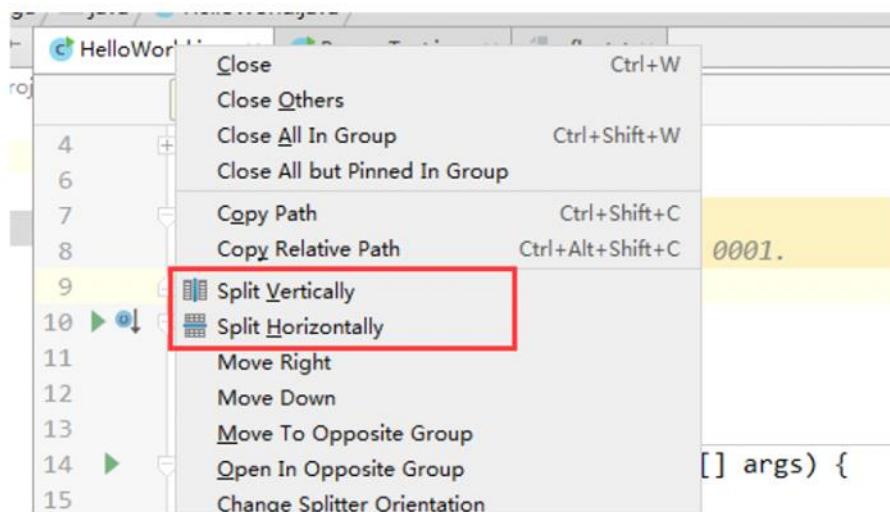
很多朋友都是从 Eclipse 转到 IntelliJ 的，这常常导致我们在需要操作 class 文件时忘记对修改后的 java 类文件进行重新编译，从而对旧文件进行了操作。

## 9. 设置为省电模式 (可忽略)



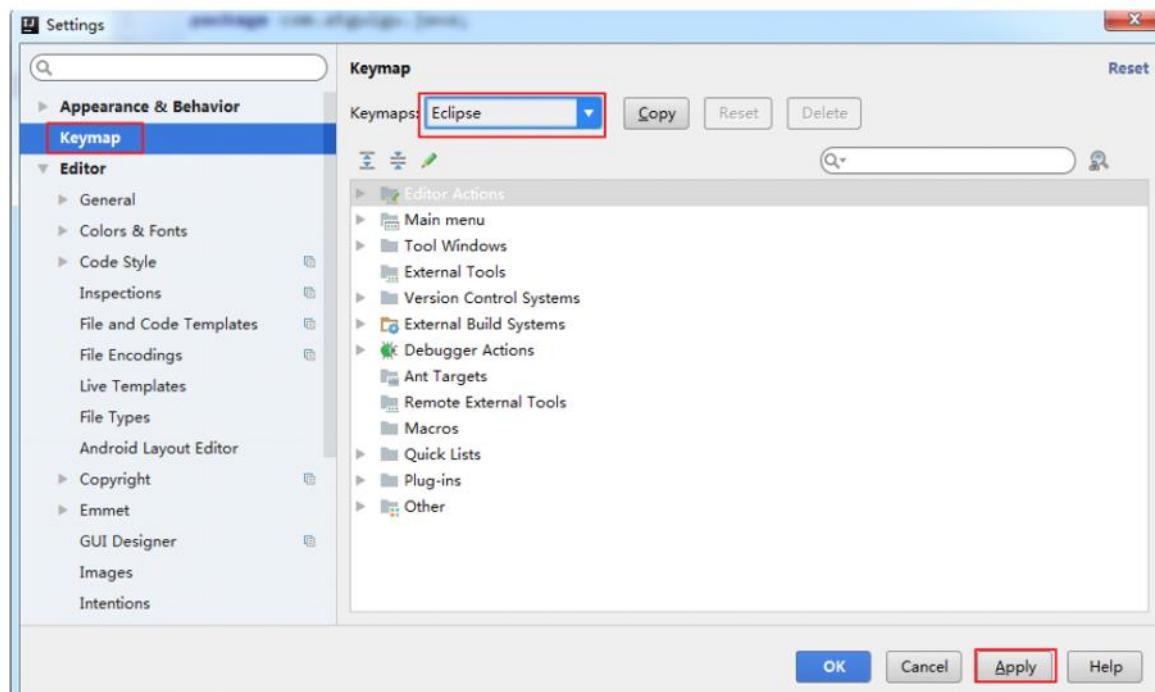
如上图所示，IntelliJ IDEA 有一种叫做 **省电模式** 的状态，开启这种模式之后 IntelliJ IDEA 会关掉代码检查和代码提示等功能。所以一般也可认为这是一种 **阅读模式**，如果你在开发过程中遇到突然代码文件不能进行检查和提示，可以来看看这里是否有开启该功能。

## 10. 设置代码水平或垂直显示

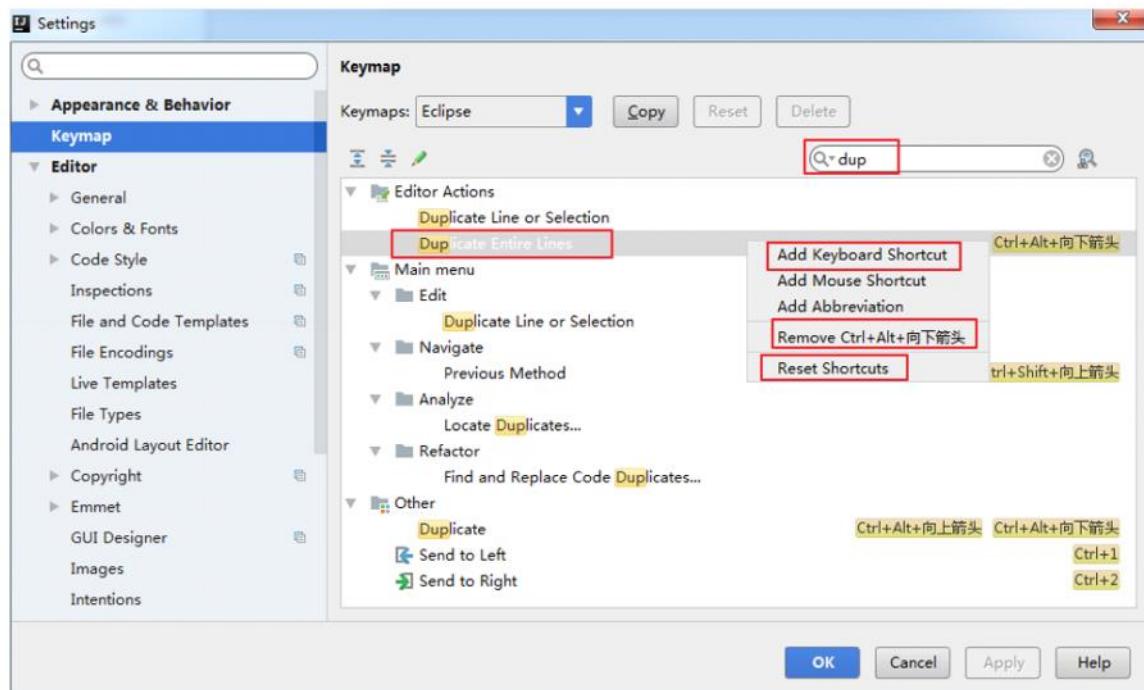


## 六、设置快捷键(Keymap)

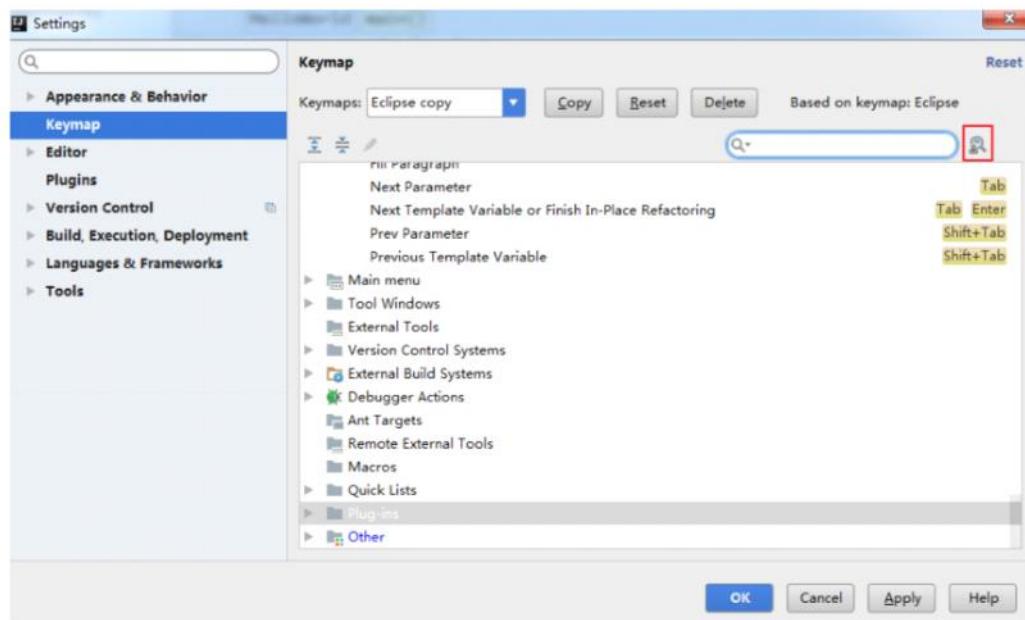
### 1. 设置快捷键为 Eclipse 的快捷键



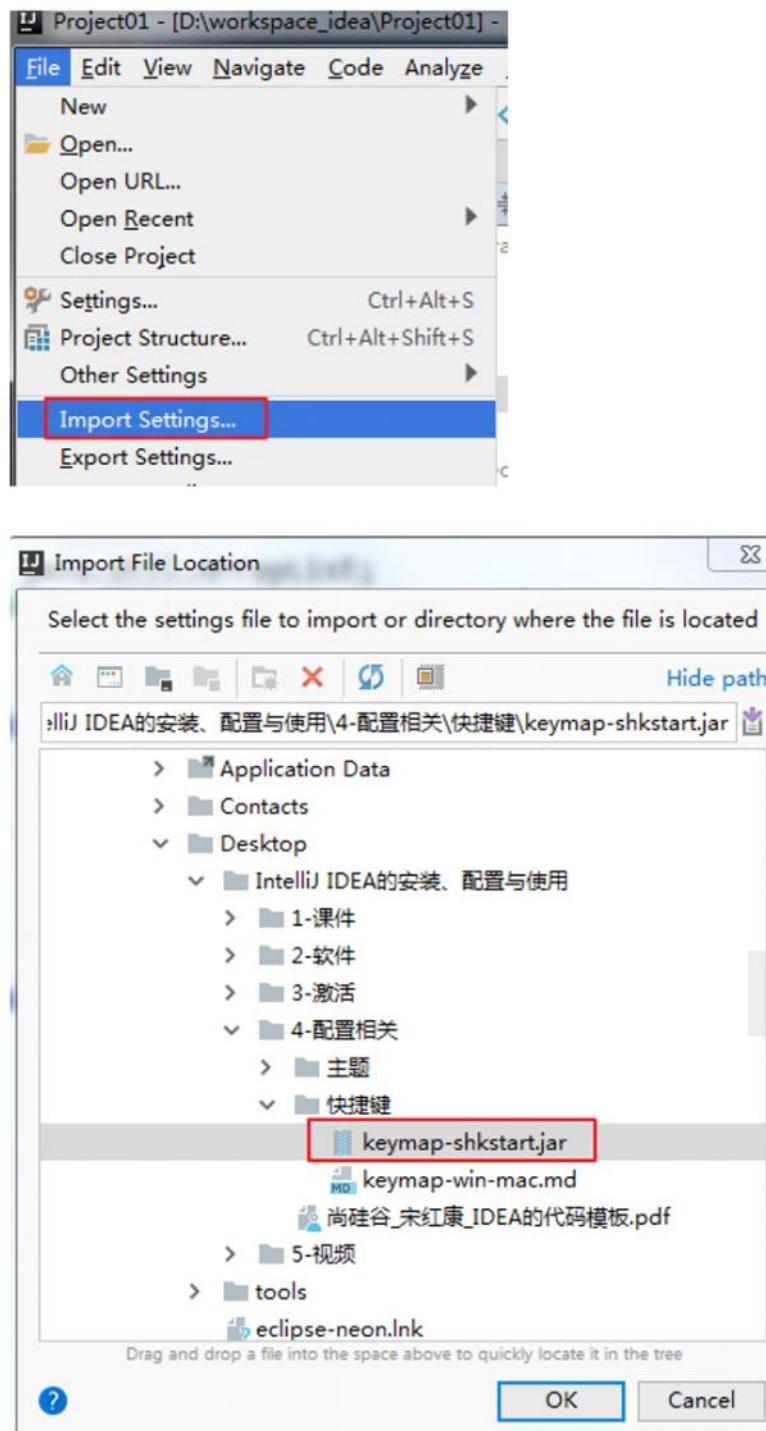
## 2.通过快捷键功能修改快捷键设置



## 3.通过指定快捷键，查看或修改其功能



## 4. 导入已有的设置



点击 OK 之后，重启 IDEA 即可。

## 5. 常用快捷键

尚硅谷 · 宋红康 设置版	
1	执行(run)
2	提示补全 (Class Name Completion)
3	单行注释
4	多行注释
5	向下复制一行 (Duplicate Lines)
6	删除一行或选中行 (delete line)
7	向下移动行(move statement down)
8	向上移动行(move statement up)
9	向下开始新的一行(start new line)
10	向上开始新的一行 (Start New Line before current)
11	如何查看源码 (class)
12	万能解错/生成返回值变量
13	退回到前一个编辑的页面 (back)
14	进入到下一个编辑的页面(针对于上条) (forward)
15	查看继承关系(type hierarchy)
16	格式化代码(reformat code)
17	提示方法参数类型(Parameter Info)
18	复制代码
19	撤销
20	反撤销
21	剪切
22	粘贴
23	保存
24	全选
25	选中数行, 整体往后移动
26	选中数行, 整体往前移动
27	查看类的结构: 类似于 eclipse 的 outline
28	重构: 修改变量名与方法名(rename)
29	大写转小写/小写转大写(toggle case)

30	生成构造器/get/set/toString	alt +shift + s
31	查看文档说明(quick documentation)	F2
32	收起所有方法(collapse all)	alt + shift + c
33	打开所有方法(expand all)	alt+shift+x
34	打开代码所在硬盘文件夹(show in explorer)	ctrl+shift+x
35	生成 try-catch 等(surround with)	alt+shift+z
36	局部变量抽取为成员变量(introduce field)	alt+shift+f
37	查找/替换(当前)	ctrl+f
38	查找(全局)	ctrl+h
39	查找文件	double Shift
40	查看类的继承结构图>Show UML Diagram)	ctrl + shift + u
41	查看方法的多层次重写结构(method hierarchy)	ctrl+alt+h
42	添加到收藏(add to favorites)	ctrl+alt+f
43	抽取方法(Extract Method)	alt+shift+m
44	打开最近修改的文件(Recently Files)	ctrl+E
45	关闭当前打开的代码栏(close)	ctrl + w
46	关闭打开的所有代码栏(close all)	ctrl + shift + w
47	快速搜索类中的错误(next highlighted error)	ctrl + shift + q
48	选择要粘贴的内容>Show in Explorer)	ctrl+shift+v
49	查找方法在哪里被调用(Call Hierarchy)	ctrl+shift+h

## 七、关于模板(Templates)

(Editor – Live Templates 和 Editor – General – Postfix Completion)

### 1. Live Templates(实时代码模板)功能介绍

它的原理就是配置一些常用代码字母缩写，在输入简写时可以出现你预定义的固

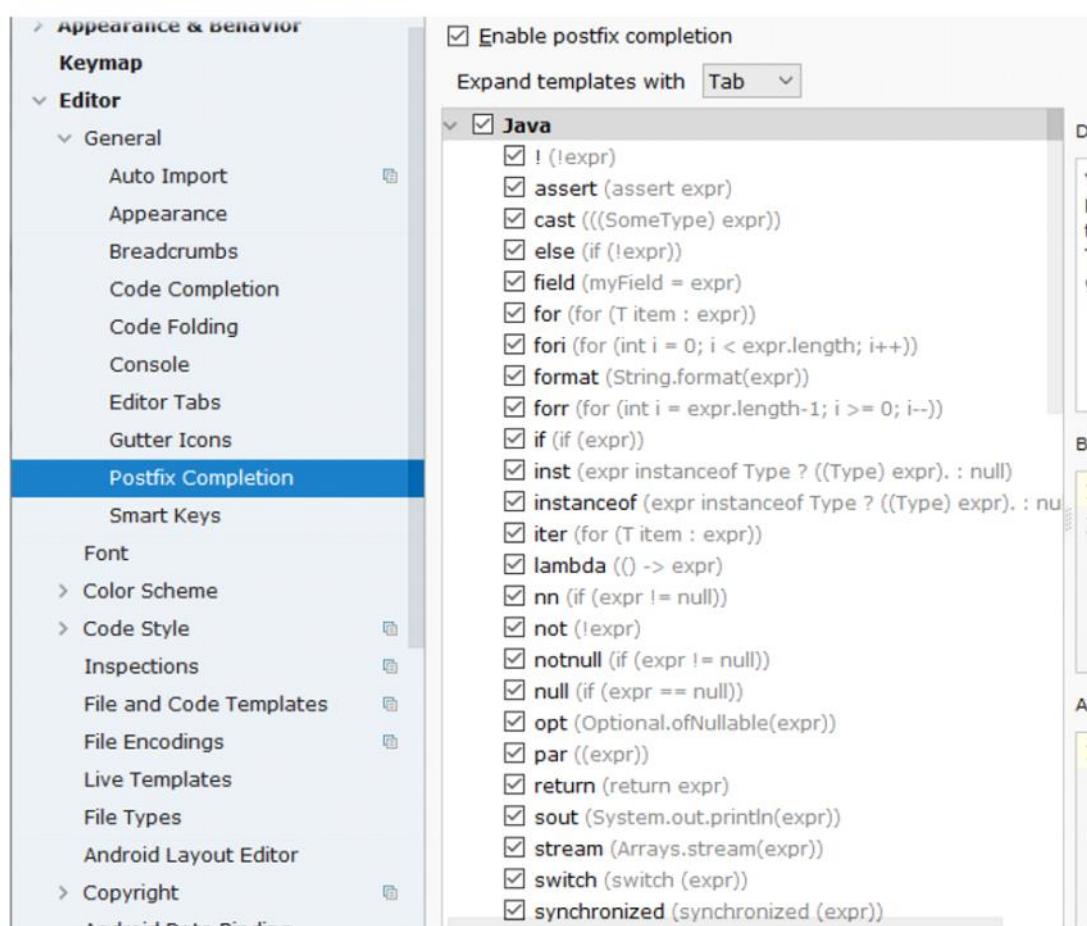
定模式的代码，使得开发效率大大提高，同时也可以增加个性化。最简单的例子就是在 Java 中输入 sout 会出现 System.out.println();

官方介绍 Live Templates:

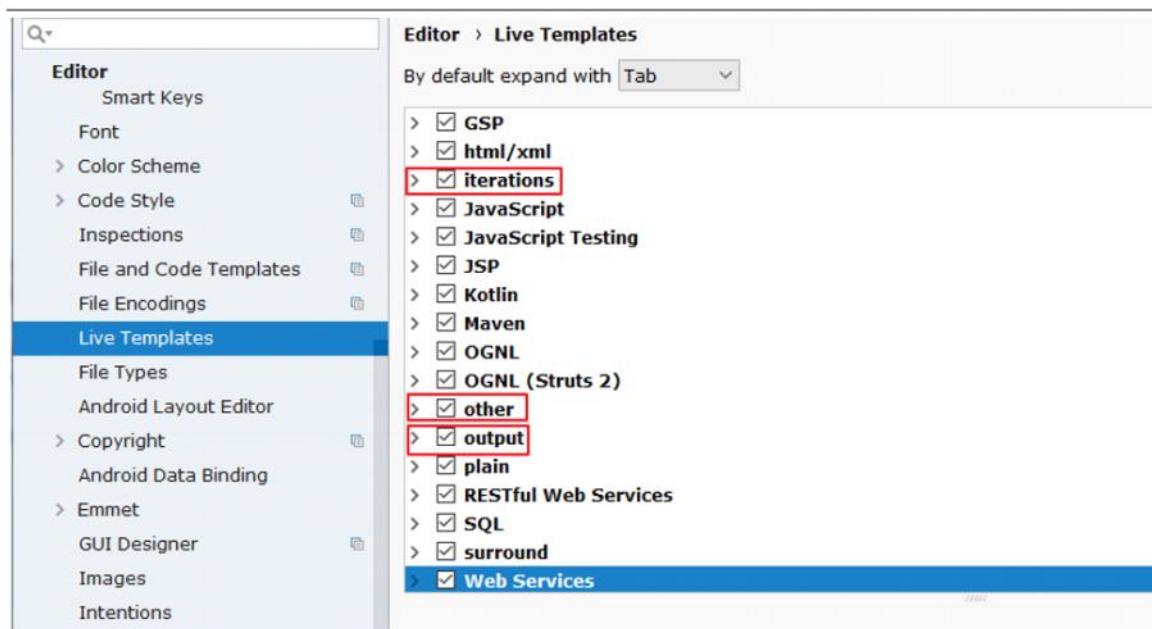
<https://www.jetbrains.com/help/idea/using-live-templates.html>

## 2. 已有的常用模板

**Postfix Completion** 默认如下:



**Live Templates** 默认如下:



二者的区别：Live Templates 可以自定义，而 Postfix Completion 不可以。同时，有些操作二者都提供了模板，Postfix Templates 较 Live Templates 能快 0.01 秒

举例：

### 2.1 psvm：可生成 main 方法

### 2.2 sout : System.out.println() 快捷输出

类似的：

```
soutp=System.out.println("方法形参名 = " + 形参名);
soutv=System.out.println("变量名 = " + 变量);
soutm=System.out.println("当前类名.当前方法");
"abc".sout => System.out.println("abc");
```

### 2.3 fori：可生成 for 循环

类似的：

iter: 可生成增强 for 循环

itar: 可生成普通 for 循环

## 2.4 list.for : 可生成集合 list 的 for 循环

```
List<String> list = new ArrayList<String>();
```

输入: list.for 即可输出

```
for(String s:list){
```

```
}
```

又如: list.fori 或 list.forr

## 2.5 ifn: 可生成 if(xxx = null)

类似的:

inn: 可生成 if(xxx != null) 或 xxx.nn 或 xxx.null

## 2.6 prsf: 可生成 private static final

类似的:

psf: 可生成 public static final

psfi: 可生成 public static final int

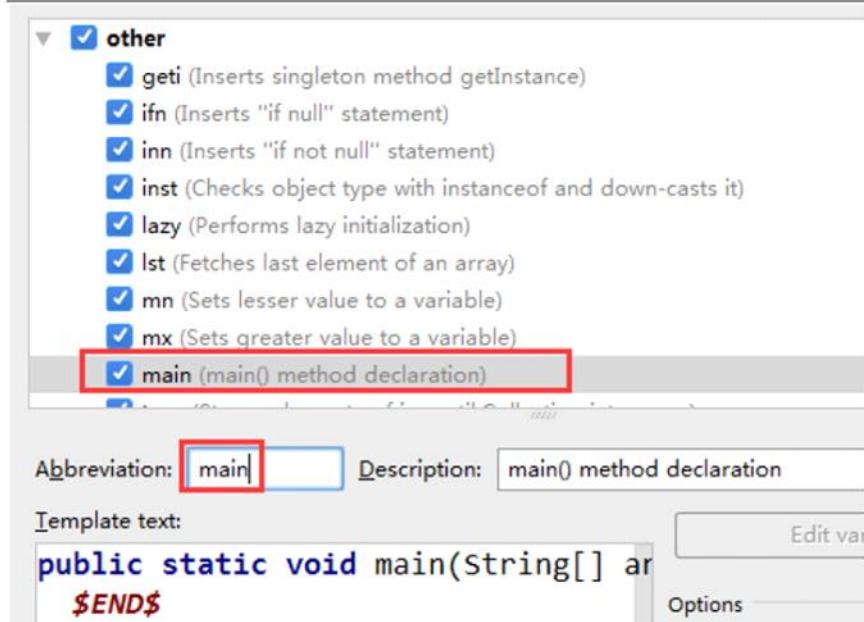
psfs: 可生成 public static final String

## 3.修改现有模板:Live Templates

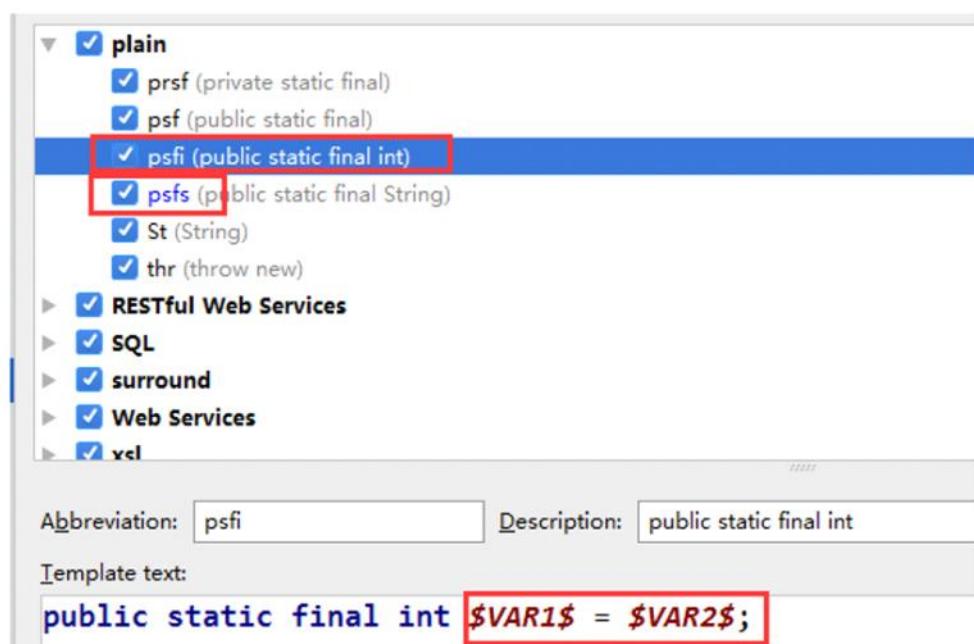
如果对于现有的模板, 感觉不习惯、不适应的, 可以修改:

### 修改 1:

通过调用 psvm 调用 main 方法不习惯, 可以改为跟 Eclipse 一样, 使用 main 调取。



## 修改 2:

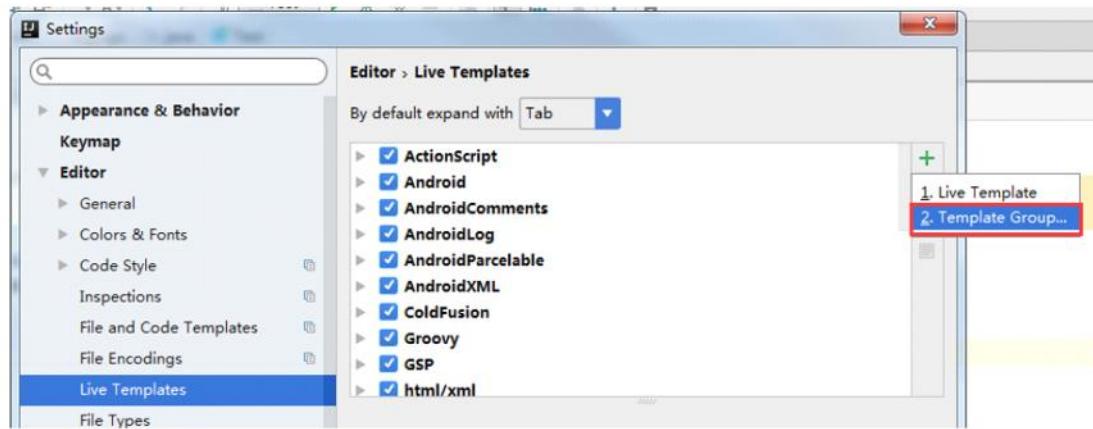


类似的还可以修改 psfs。

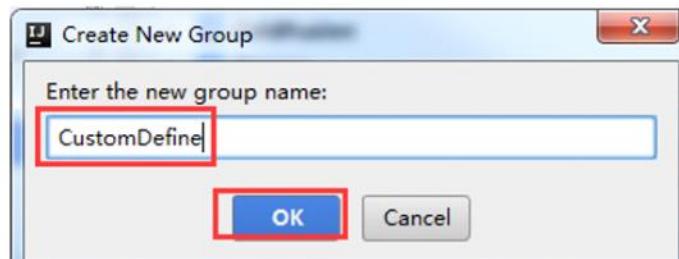
## 4. 自定义模板

IDEA 提供了很多现成的 Templates。但你也可以根据自己的需要创建新的

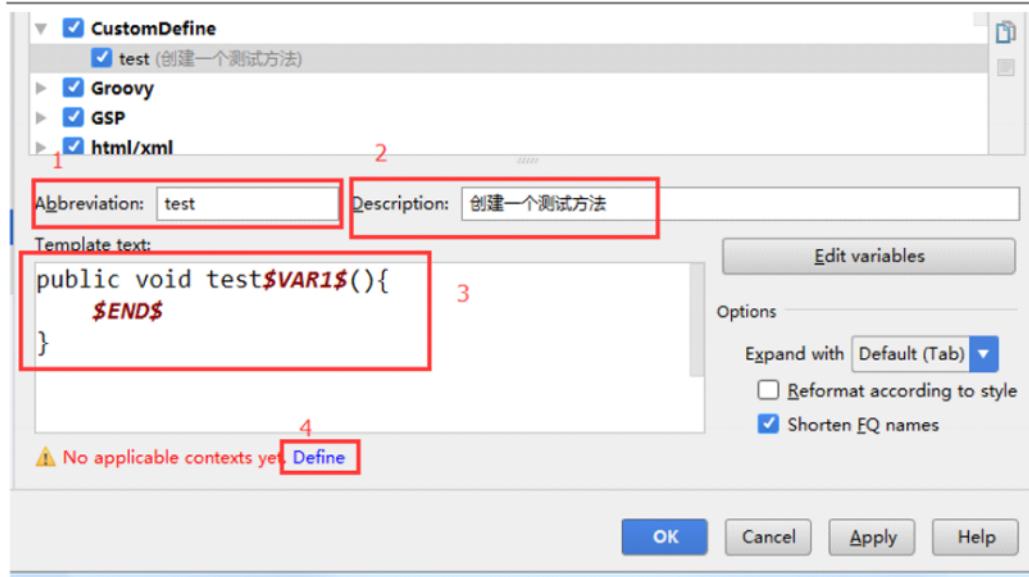
Template。



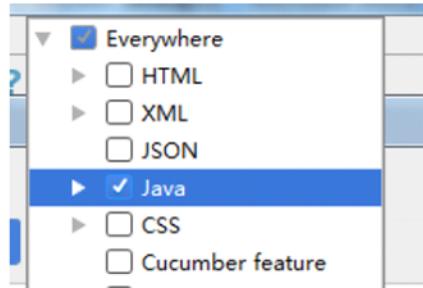
先定义一个模板的组：



选中自定义的模板组，点击“+”来定义模板。



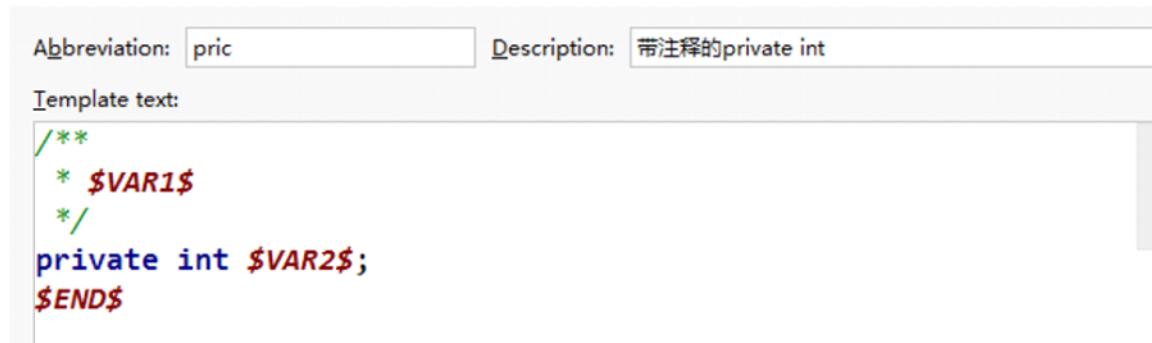
1. Abbreviation: 模板的缩略名称
2. Description: 模板的描述
3. Template text: 模板的代码片段
4. 应用范围。比如点击 Define。选择如下:



可以如上的方式定义个测试方法，然后在 java 类文件中测试即可。

类似的可以再配置如下的几个 Template:

1.



2.

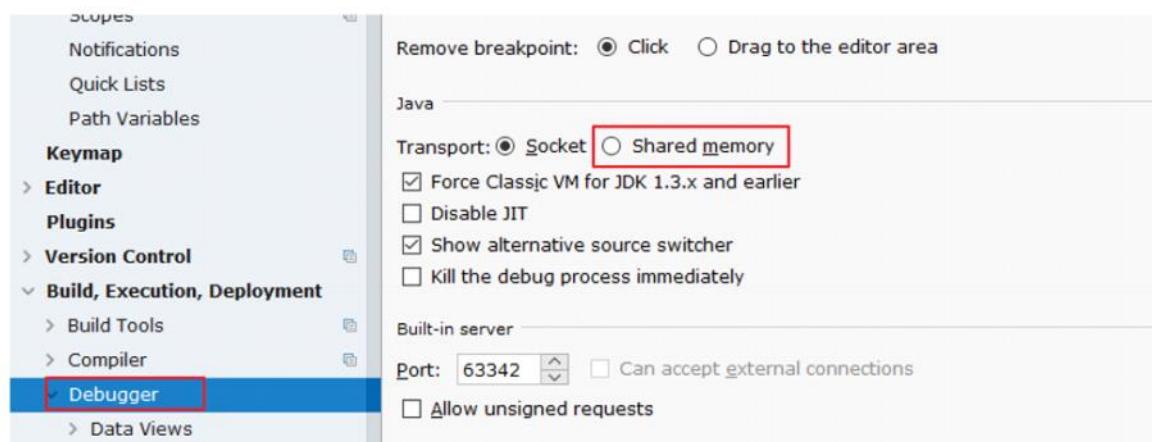
Abbreviation: prsc Description: 带注释的private String

Template text:

```
/**  
 * $VAR1$  
 */  
private String $VAR2$;  
$END$
```

## 八、断点调试

### 1. Debug 的设置



设置 Debug 连接方式，默认是 Socket。Shared memory 是 Windows 特有的一个属性，一般在 Windows 系统下建议使用此设置，内存占用相对较少。

### 2. 常用断点调试快捷键



step over 进入下一步，如果当前行断点是一个方法，则不进入当前方法体内



step into 进入下一步，如果当前行断点是一个方法，则进入当前方法体内

- force step into 进入下一步，如果当前行断点是一个方法，则进入当前方法体内
- step out 跳出
- resume program 恢复程序运行，但如果该断点下面代码还有断点则停在下一个断点上
- stop 停止
- mute breakpoints 点中，使得所有的断点失效
- view breakpoints 查看所有断点

对于常用的 Debug 的快捷键，需要大家熟练掌握。

### 3. 条件断点

说明：

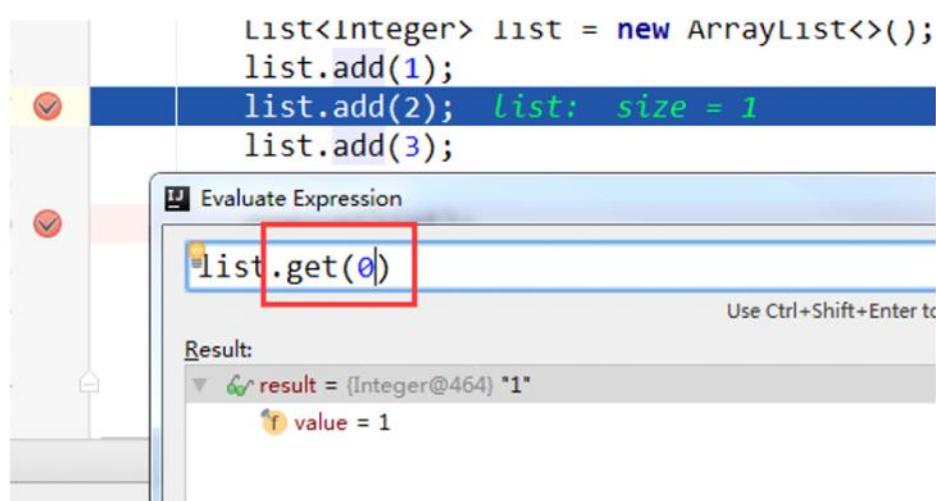
调试的时候，在循环里增加条件判断，可以极大的提高效率，心情也能愉悦。

具体操作：

在断点处右击调出条件断点。可以在满足某个条件下，实施断点。

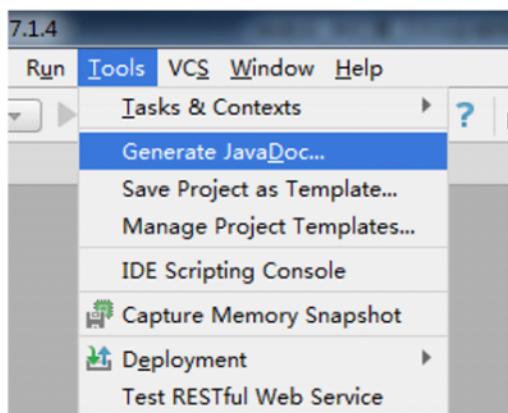
查看表达式的值(Ctrl + u)：

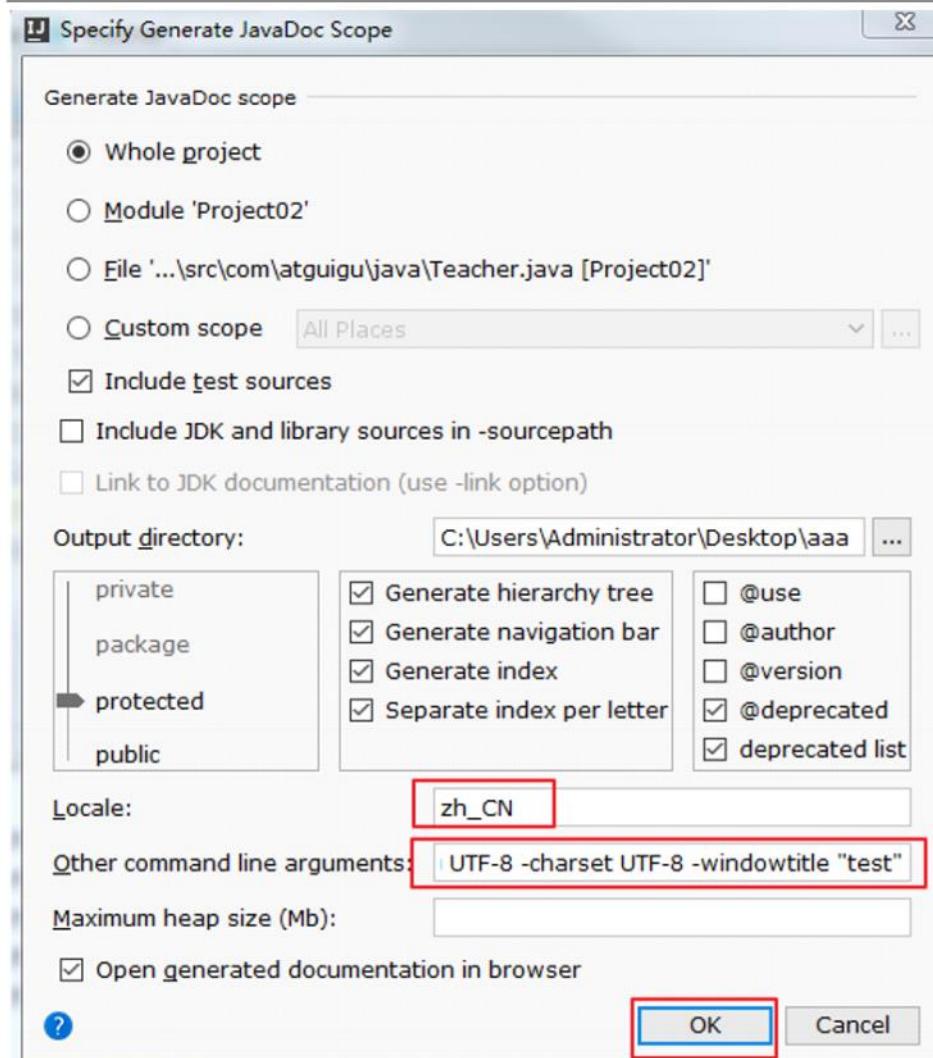
选择行，ctrl + u。还可以在查看框中输入编写代码时的其他方法：



## 九、其它设置

### 1.生成 javadoc





输入：

Locale: 输入语言类型: zh\_CN

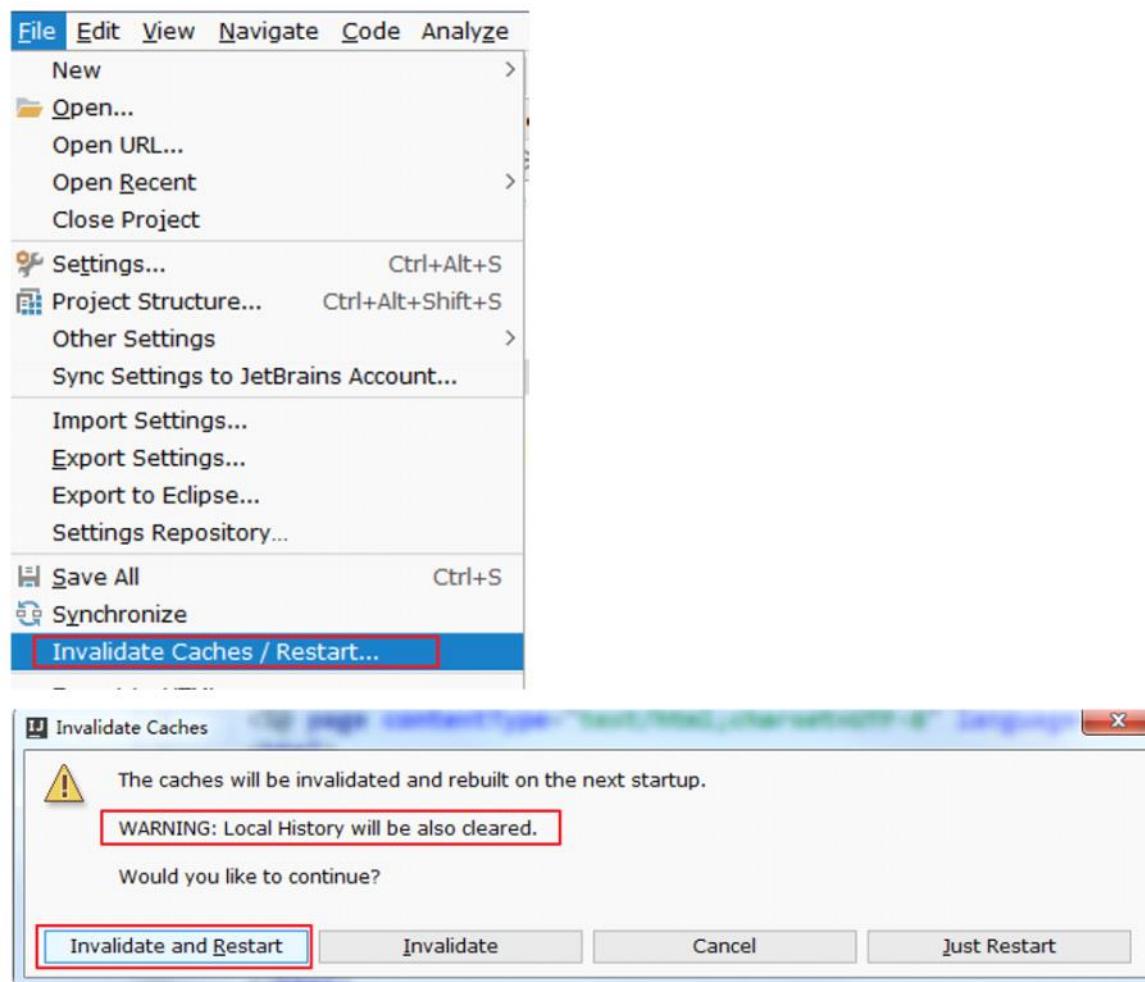
Other command line arguments: -encoding UTF-8 -charset UTF-8

## 2. 缓存和索引的清理

IntelliJ IDEA 首次加载项目的时候，都会创建索引，而创建索引的时间跟项目的文件多少成正比。在 IntelliJ IDEA 创建索引过程中即使你编辑了代码也是编译不了、运行不起来的，所以还是安安静静等 IntelliJ IDEA 创建索引完成。

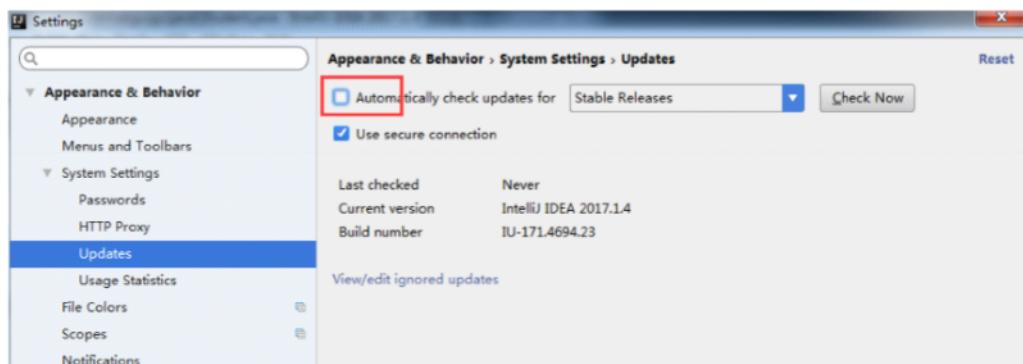
IntelliJ IDEA 的缓存和索引主要是用来加快文件查询，从而加快各种查找、代码提示等操作的速度，所以 IntelliJ IDEA 的索引的重要性再强调一次也不为过。

但是，IntelliJ IDEA 的索引和缓存并不是一直会良好地支持 IntelliJ IDEA 的，某些特殊条件下，IntelliJ IDEA 的缓存和索引文件也是会损坏的，比如：断电、蓝屏引起的强制关机，当你重新打开 IntelliJ IDEA，很可能 IntelliJ IDEA 会报各种莫名其妙错误，甚至项目打不开，IntelliJ IDEA 主题还原成默认状态。即使没有断电、蓝屏，也会有莫名奇怪的问题的时候，也很有可能是 IntelliJ IDEA 缓存和索引出现了问题，这种情况还不少。遇到此类问题也不用过多担心。我们可以清理缓存和索引。如下：



- 一般建议点击 **Invalidate and Restart**, 这样会比较干净。
- 上图警告: 清除索引和缓存会使得 IntelliJ IDEA 的 **Local History** 丢失。所以如果你项目没有加入到版本控制, 而你又需要你项目文件的历史更改记录, 那你最好备份下你的 **LocalHistory** 目录。目录地址在: C:\Users\当前登录的系统用户名\.IntelliJdea14\system\LocalHistory 建议使用硬盘的全文搜索, 这样效率更高。
- 通过上面方式清除缓存、索引本质也就是去删除 C 盘下的 **system** 目录下的对应的文件而已, 所以如果你不用上述方法也可以删除整个 **system**。当 IntelliJ IDEA 再次启动项目的时候会重新创建新的 **system** 目录以及对应项目缓存和索引。

### 3.取消更新

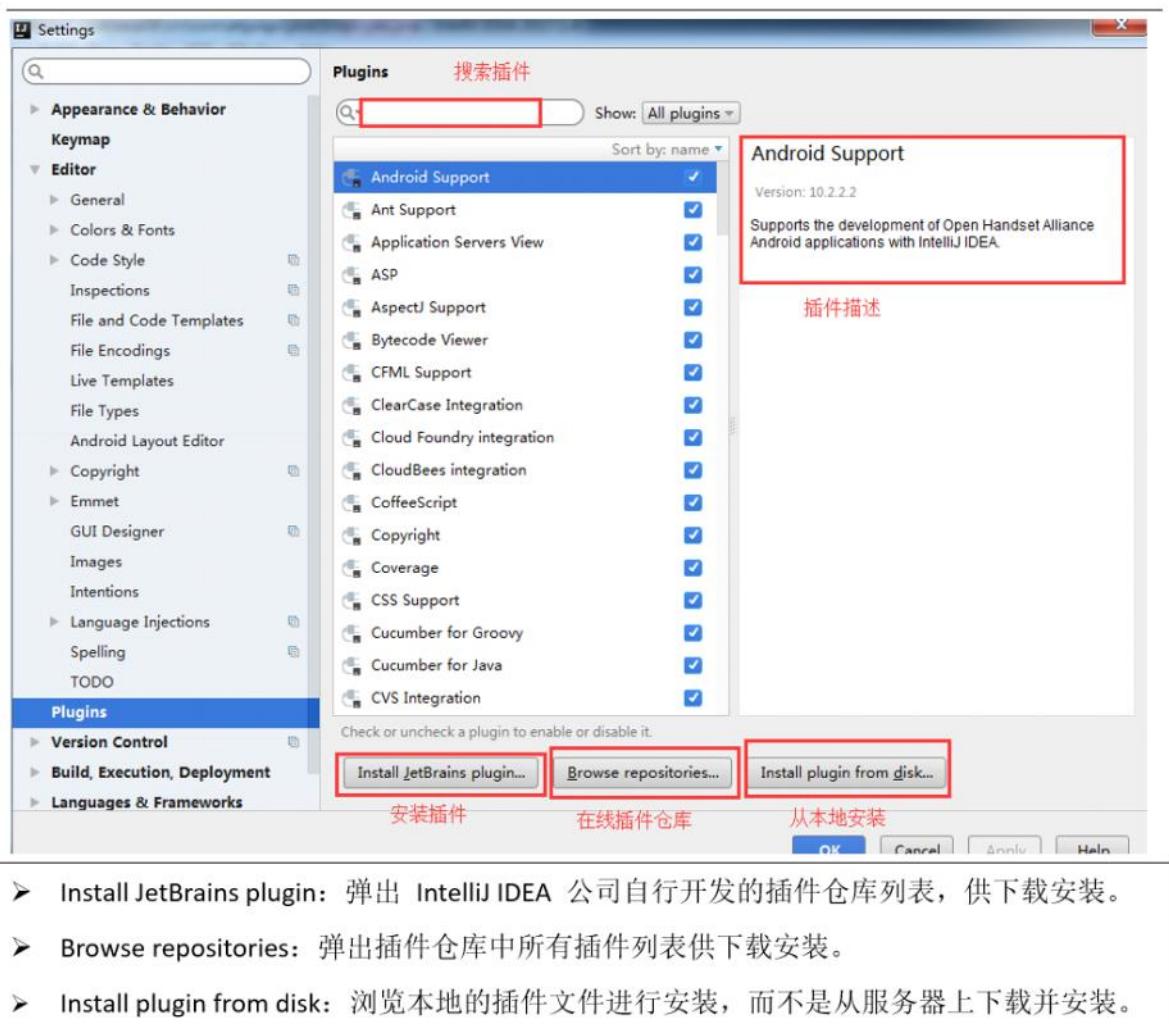


取消勾选: 即可取消更新

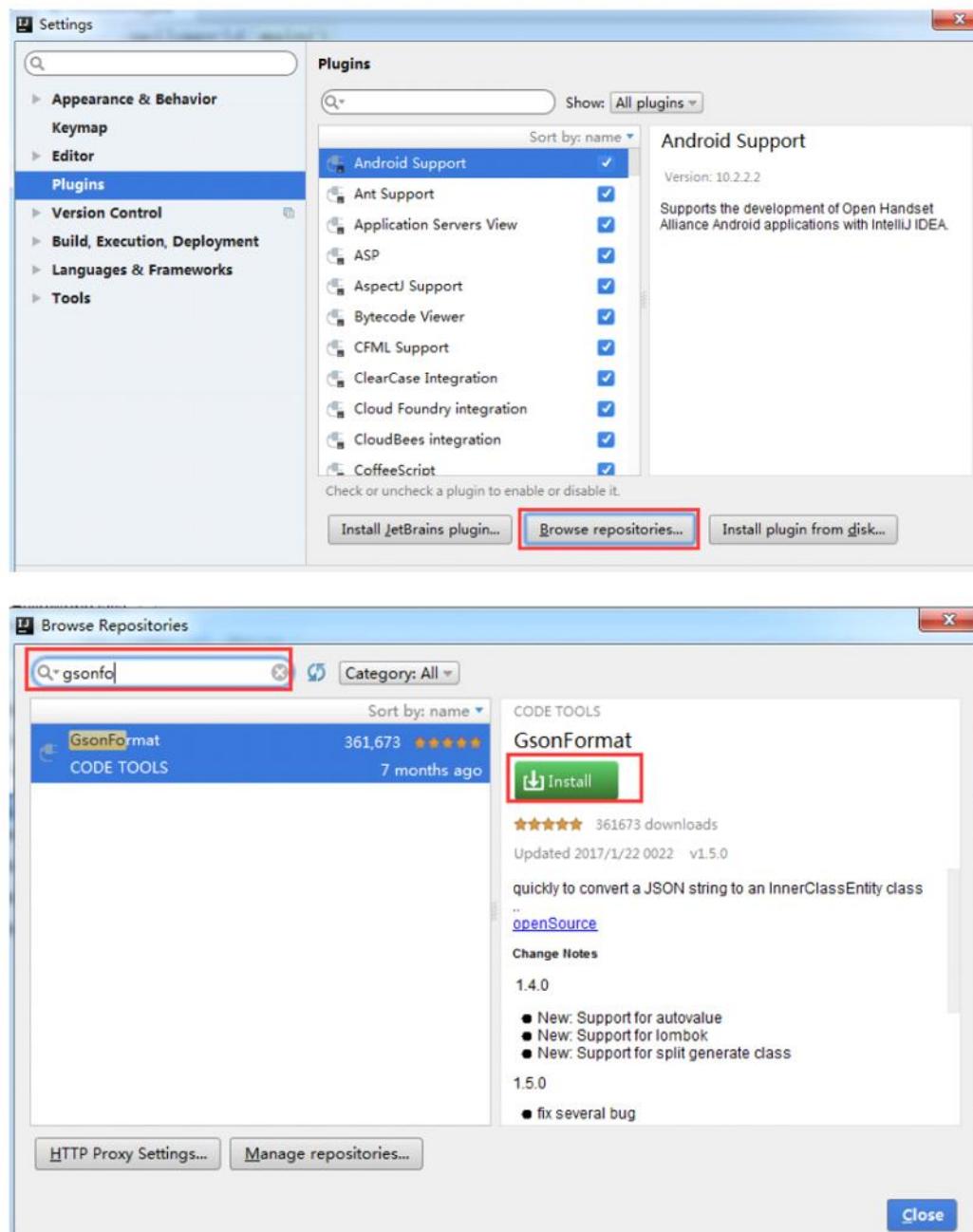
### 4.插件的使用

在 IntelliJ IDEA 的安装讲解中我们其实已经知道, IntelliJ IDEA 本身很多功能也都是通过插件的方式来实现的。

官网插件库: <https://plugins.jetbrains.com/>



需要特别注意的是：在国内的网络下，经常出现显示不了插件列表，或是显示了插件列表，无法下载完成安装。这时候请自行打开 VPN，一般都可以得到解决。



如上图演示，在线安装 IntelliJ IDEA 插件库中的插件。安装完以后会提示重启，才可以使用插件。

#### 常用插件推荐：

插件名称	插件介绍	官网地址
------	------	------



插件名称	插件介绍	官网地址
Key promoter	快捷键提示	<a href="https://plugins.jetbrains.com/plugin/4455?pr=idea">https://plugins.jetbrains.com/plugin/4455?pr=idea</a>
CamelCase	驼峰式命名和下划线命名交替变化	<a href="https://plugins.jetbrains.com/plugin/7160?pr=idea">https://plugins.jetbrains.com/plugin/7160?pr=idea</a>
CheckStyle-IDEA	代码样式检查	<a href="https://plugins.jetbrains.com/plugin/1065?pr=idea">https://plugins.jetbrains.com/plugin/1065?pr=idea</a>
FindBugs-IDEA	代码 Bug 检查	<a href="https://plugins.jetbrains.com/plugin/3847?pr=idea">https://plugins.jetbrains.com/plugin/3847?pr=idea</a>
Statistic	代码统计	<a href="https://plugins.jetbrains.com/plugin/4509?pr=idea">https://plugins.jetbrains.com/plugin/4509?pr=idea</a>
JRebel Plugin	热部署	<a href="https://plugins.jetbrains.com/plugin/?id=4441">https://plugins.jetbrains.com/plugin/?id=4441</a>
CodeGlance	在编辑代码最右侧，显示一块代码小地图	<a href="https://plugins.jetbrains.com/plugin/7275?pr=idea">https://plugins.jetbrains.com/plugin/7275?pr=idea</a>
Eclipse Code Formatter	使用 Eclipse 的代码格式化风格，在一个团队中如果公司有规定格式化风格，这个可以使用。	<a href="https://plugins.jetbrains.com/plugin/6546?pr=idea">https://plugins.jetbrains.com/plugin/6546?pr=idea</a>
GsonFormat	把 JSON 字符串直接实例化成类	<a href="https://plugins.jetbrains.com/plugin/7654?pr=idea">https://plugins.jetbrains.com/plugin/7654?pr=idea</a>

# 56.Java高级-多线程

2021年8月10日 14:40

● **程序(program)**是为完成特定任务、用某种语言编写的一组指令的集合。即指一段静态的代码，静态对象。

● **进程(process)**是程序的一次执行过程，或是正在运行的一个程序。是一个动态的过程：有它自身的产生、存在和消亡的过程。——生命周期

➢ 如：运行中的QQ，运行中的MP3播放器

➢ 程序是静态的，进程是动态的

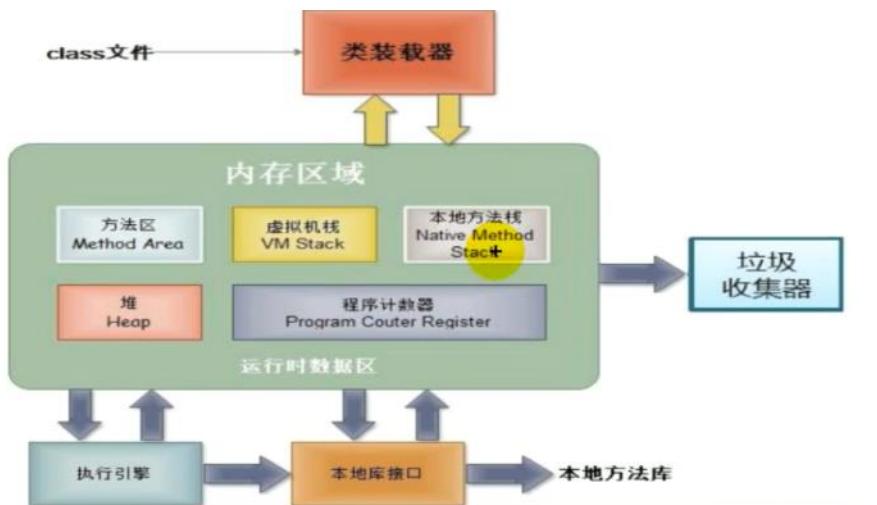
➢ 进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域

● **线程(thread)**，进程可进一步细化为线程，是1个程序内部的一条执行路径。

➢ 若一个进程同一时间并行执行多个线程，就是支持多线程的

➢ 线程作为调度和执行的单位，每个线程拥有独立的运行栈和程序计数器(pc)，线程切换的开销小

➢ 一个进程中的多个线程共享相同的内存单元/内存地址空间 ➤ 它们从同一堆中分配对象，可以访问相同的变量和对象。这就使得线程间通信更简便、高效。但多个线程操作共享的系统资源可能就会带来安全的隐患。



1 Class Loader类加载器

2 Execution Engine执行引擎负责解释命令，提交操作系统执行。

3 Native Interface 本地接口

4 Runtime Data Area 运行数据区

让天下没有难学的IT

## ● 单核CPU和多核CPU的理解

➢ 单核CPU，其实是一种假的多线程，因为在一段时间单元内，也只能执行一个线程的任务。例如：虽然有多车道，但是收费站只有一个工作人员在收费，只有收了费才能通过，那么CPU就好比收费人员。如果有某个人不想交钱，那么收费人员可以把他“挂起”（晾着他，等他想通了，准备好了钱，再去收费）。但是因为CPU时间单元特别短，因此感觉不出来。

➢ 如果是多核的话，才能更好的发挥多线程的效率。（现在的服务器都是多核的）

➢ 一个Java应用程序java.exe，其实至少有三个线程：**main()**主线程，**gc()**垃圾回收线程，异常处理线程。当然如果发生异常，会影响主线程。

## ● 并行与并发

➢ 并行：多个CPU同时执行多个任务。比如：多人同时做不同的事。

➢ 并发：一个CPU(采用时间片)同时执行多个任务。比如：秒杀、多人做同一件事。

让天下没有难学的IT

## 使用多线程的优点

背景：以单核CPU为例，只使用单个线程先后完成多个任务（调用多个方法），肯定比用多个线程来完成用的时间更短，为何仍需多线程呢？

多线程程序的优点：

1. 提高应用程序的响应。对图形化界面更有意义，可增强用户体验。
2. 提高计算机系统CPU的利用率
3. 改善程序结构。将既长又复杂的进程分为多个线程，独立运行，利于理解和修改

## 何时需要多线程

- 程序需要同时执行两个或多个任务。
- 程序需要实现一些需要等待的任务时，如用户输入、文件读写操作、网络操作、搜索等。
- 需要一些后台运行的程序时。



## 补充：线程的分类

Java中的线程分为两类：一种是**守护线程**，一种是**用户线程**。

- 它们在几乎各个方面都是相同的，唯一的区别是判断JVM何时离开。
- 守护线程是用来服务用户线程的，通过在**start()**方法前调用**thread.setDaemon(true)**可以把一个用户线程变成一个守护线程。
- Java垃圾回收就是一个典型的守护线程。
- 若JVM中都是守护线程，当前JVM将退出。
- 形象理解：**兔死狗烹，鸟尽弓藏**

# 57.Java高级-多线程创建

2021年8月10日 15:15

## 线程的创建和启动

- Java语言的JVM允许程序运行多个线程，它通过**java.lang.Thread**类来体现。

- Thread类的特性

- 每个线程都是通过某个特定Thread对象的run()方法来完成操作的，经常把run()方法的主体称为线程体
- 通过该Thread对象的start()方法来启动这个线程，而非直接调用run()

多线程的创建：

方式一：继承于Thread类

\*\*\*\*\*具体步骤：

1. 创建一个继承于Thread类的子类
2. 重写Thread类的run()--->将此线程执行的操作声明在run中
3. 创建Thread类的子类的对象
4. 通过对对象调用start()
  - ①启动当前线程
  - ②调用当前线程的run()

```
package com.zqf.senior;
/**
 * @author oscarzqf
 * @description
 * @create 2021-08-10-15:33
 */
public class MyThread extends Thread{
    public void run(){
        for (int i = 0; i <=100; i++) {
            System.out.println(i);
        }
    }
}
```

```
package com.zqf.senior;

public class test01 {
    public static void main(String[] args) {
        MyThread t1=new MyThread();
        //线程执行
        t1.start();
        //main线程中执行
        for (int i=0;i<1000;i++) System.out.print("*");
    }
}
```

问题总结：

- 1.不能通过调用run()启动线程，这本质是调用方法
- 2.已经start () 的线程不能再次start(),会报异常，需要重新创建一个线程对象再调用

\*\*\*\*\*简化步骤：创建Thread类的匿名子类的对象

```
public class ThreadDemo {
    public static void main(String[] args) {
        (new Thread(){
            public void run(){
                for (int i = 0; i < 101; i++) {
                    System.out.println(i);
                }
            }
        }).start();
    }
}
```

方式二：实现Runnable接口

- 1.创建一个实现了Runnable接口的类
- 2.实现类重写Runnable中的抽象方法：run()
- 3.创建实现类的对象
- 4.将此对象作为参数传递到Thread类的构造器中  
创建Thread类的对象
- 5.通过Thread类的对象调用start()

```
public class Window extends Thread{  
    private static int ticket=100;//设置为共用静态  
    @Override//创建三个窗口共卖100张票，但是存在线程安全问题没有解决  
    public void run() {  
        while (true){  
            if(ticket>0){  
                System.out.println(getName()+"：卖票，票号为：" +ticket);  
                ticket--;  
            } else{  
                break;  
            }  
        }  
    }  
}
```

```
public class MyyThread implements Runnable{  
    private int ticket=100;  
    public void run(){  
        while (true){  
            if(ticket>0){  
                System.out.println(Thread.currentThread().getName()  
                    +":卖票，票号为：" +ticket);  
                ticket--;  
            } else{  
                break;  
            }  
        }  
    }  
}  
class MyyThreadTest{  
    public static void main(String[] args) {  
        MyyThread m1=new MyyThread(); //一个对象  
        Thread t1=new Thread(m1);  
        t1.setName("ticket000 1");  
        t1.start();  
        //再启动一个线程  
        Thread t2=new Thread(m1);  
        t2.setName("ticket000 2");  
        t2.start();  
    }  
}
```

两种方式的对比：

开发中优先选择实现Runnable接口的方式

原因：

1. 实现的方式没有类的单继承性的局限性
2. 实现的方式更适合来处理多个线程有共享数据的情况

联系：

1. public class Thread implements Runnable
2. 两种方式都需要重写run()
3. 两种方式启动线程需要调用start ()

# 58.Java高级-Thread常用方法

2021年8月10日 16:21

Thread类中常用的方法：

- 1.start():启动当前线程；调用当前线程的run();
- 2.run():通常需要重写，将创建线程执行的操作声明在方法内
- 3.currentThread():静态方法，返回执行当前代码的线程
- 4.getName():获取当前线程的名字
- 5.setName():设置当前线程的名字，start之前；也可以通过构造器设置
- 6.yield():释放当前CPU的执行权，(CPU可能又给分配到当前线程)
- 7.join():在线程A中调用线程B的join()方法，此时线程A进入阻塞状态  
直到线程B执行完成，线程A才结束阻塞状态,等待资源分配，会抛异常
- 8.stop():已经过时。当执行此方法时，强制结束当前线程
- 9.sleep(long millitime):让当前线程阻塞指定的毫秒数，之后等待分配,会抛异常，  
需要try-catch处理
- 10.isAlive():判断当前线程是否存活

！！！ 查看API或者源码，解决异常问题

# 59.Java高级-线程调度

2021年8月11日 12:16

## 线程的调度

### ● 调度策略

➤ 时间片



➤ 抢占式：高优先级的线程抢占CPU

### ● Java的调度方法

➤ 同优先级线程组成先进先出队列（先到先服务），使用时间片策略

➤ 对高优先级，使用优先调度的抢占式策略

### ● 线程的优先级等级

➤ MAX\_PRIORITY: 10

➤ MIN\_PRIORITY: 1

➤ NORM\_PRIORITY: 5

### ● 涉及的方法

➤ getPriority()：返回线程优先值

➤ setPriority(int newPriority)：改变线程的优先级

I

### ● 说明

➤ 线程创建时继承父线程的优先级

➤ 低优先级只是获得调度的概率低，并非一定是在高优先级线程之后才被调用

+线程的优先级：

1.MAX\_PRIORITY:10-->

MIN\_PRIORITY:1----->

NORM\_PRIORITY:5--->默认优先级

2.当前线程优先级的获取和设置

-getPriority():获取线程的优先级

-setPriority(int p):设置线程的优先级

说明：高优先级的线程要抢占低优先级线程CPU的执行

权。但是只是概率上讲，高优先级的线程执行概率大。

并不是等高优先级执行完才执行低优先级

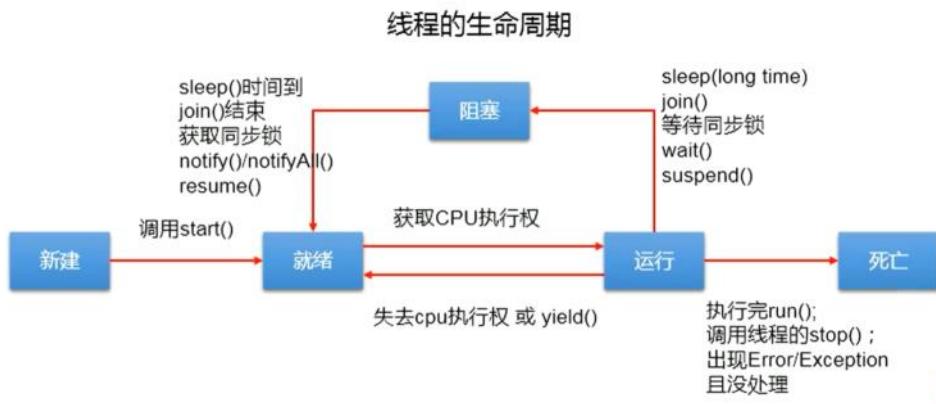
# 60.Java高级-线程生命周期

2021年8月11日 15:04

## ● JDK中用**Thread.State**类定义了线程的几种状态

要想实现多线程，必须在主线程中创建新的线程对象。Java语言使用**Thread**类及其子类的对象来表示线程，在它的一个完整的生命周期中通常要经历如下的五种状态：

- 新建**：当一个**Thread**类或其子类的对象被声明并创建时，新生的线程对象处于新建状态
- 就绪**：处于新建状态的线程被**start()**后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件，只是没分配到CPU资源
- 运行**：当就绪的线程被调度并获得CPU资源时，便进入运行状态，**run()**方法定义了线程的操作和功能
- 阻塞**：在某种特殊情况下，被人为挂起或执行输入输出操作时，让出CPU并临时中止自己的执行，进入阻塞状态
- 死亡**：线程完成了它的全部工作或线程被提前强制性地中止或出现异常导致结束

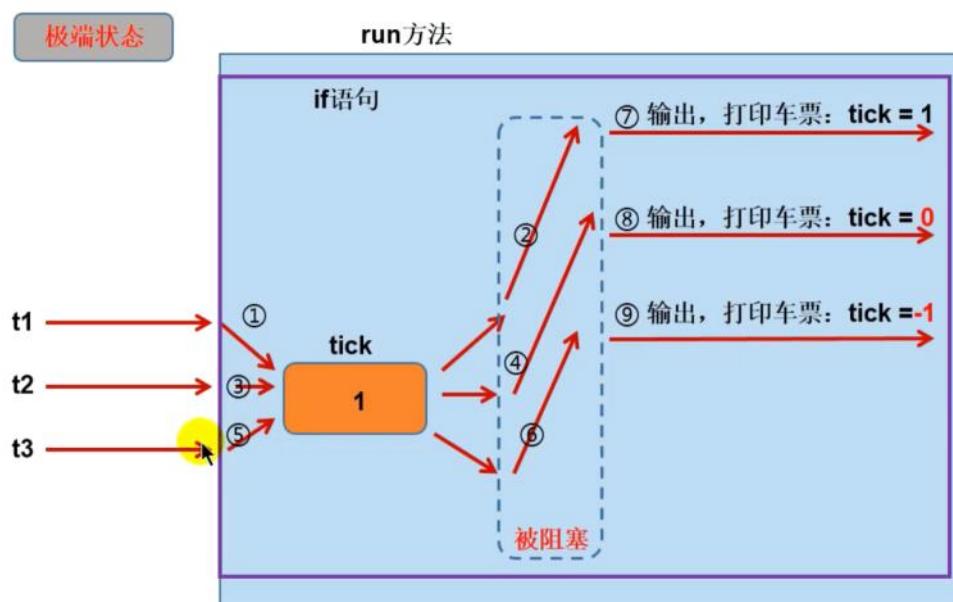
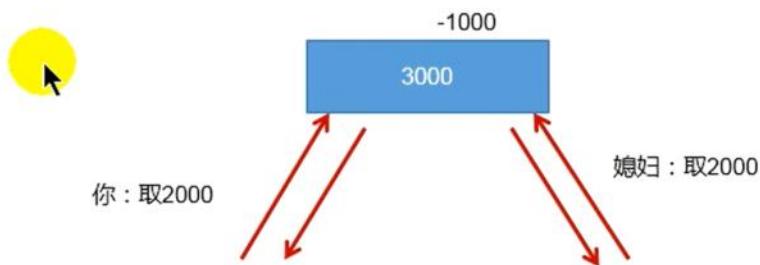


# 61.Java高级-线程的同步

2021年8月11日 15:24

## ● 问题的提出

- 多个线程执行的不确定性引起执行结果的不稳定
- 多个线程对账本的共享，会造成操作的不完整性，会破坏数据。



### 1. 问题：

卖票过程中，出现重票，错票-->出现了线程安全问题

### 2. 原因：

某个线程还未完成操作时，其他线程就参与进来，操作数据，导致条件等错误

### 3. 如何解决：

当一个线程在操作ticket (数据) 的时候，其他线程才能参与进来  
直到线程a操作完成ticket, 其他线程才能参与进来，即使a被阻塞

### 4. 在Java中，我们通过同步机制，来解决线程安全问题

## 5. 同步的好处，解决了线程的安全问题

局限性：操作同步代码时，只有一个线程参与，相当于一个单线程问题

### 方式一：同步代码块

```
synchronized(同步监视器){  
    //1.需要被同步的代码：操作共享数据的代码-->不能包多或包少  
    //2.共享数据：多个线程共同操作的变量，如ticket  
    //3.同步监视器：俗称：锁。任何一个类的对象都可以充当锁  
    //4.要求多个线程必须共用同一把锁【必须共用一个对象（任意）】  
    //，可以新建或者用this(接口实现),或者用 [类.class]只加载一次是唯一的。  
}
```

```
public class MyThread implements Runnable {  
    private int ticket = 100;  
    Object obj = new Object();  
    public void run() {  
        while (true) {  
            synchronized (obj) {  
                if (ticket > 0) {  
                    System.out.println(Thread.currentThread().getName()  
                        + ":卖票，票号为：" + ticket);  
                    ticket--;  
                } else {  
                    break;  
                }  
            }  
        }  
    }  
}
```

### 方式二：同步方法

使用同步方法解决实现Runnable接口的线程安全问题

```
public class MyThread implements Runnable {
    private int ticket = 100;
    public void run() {
        while (ticket!=0) {
            show();
        }
    }
    private synchronized void show(){//同步监视器: this
        if (ticket > 0) {
            System.out.println(Thread.currentThread().getName()
                + ":卖票, 票号为: " + ticket);
            ticket--;
        }
    }
}
```

使用同步方法来解决继承Thread类方式中的线程安全问题

```
public class Window extends Thread{
    private static int ticket=100;//设置为共用静态
    @Override//创建三个窗口共卖100张票
    public void run() {
        while (ticket!=0){
            show();
        }
    }
    private static synchronized void show(){//同步监视器: Window.class
        //private synchronized void show(){//同步监视器t1,t2,t3
        if(ticket>0) {
            System.out.println(Thread.currentThread().getName() + ":卖票, 票号为: "
                + ticket);
            ticket--;
        }
    }
}
```

关于同步方法的总结：

1. 同步方法仍然涉及到同步监视器，只是不用我们显式声明

2. 非静态的同步方法，同步监视器是：this

静态的同步方法，同步监视器是：当前类本身 类.class

# 62.Java高级-线程死锁问题

2021年8月11日 17:35

## 线程的死锁问题

### ●死锁

- 不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁
- 出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续

### ●解决方法

- 专门的算法、原则
- 尽量减少同步资源的定义
- 尽量避免嵌套同步

# 63. Java高级-线程的同步\*

2021年8月11日 17:56

## 3. Lock(锁)

- 从JDK 5.0开始，Java提供了更强大的线程同步机制——通过显式定义同步锁对象来实现同步。同步锁使用Lock对象充当。
- java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象。
- ReentrantLock类实现了Lock，它拥有与synchronized相同的并发性和内存语义，在实现线程安全的控制中，比较常用的是ReentrantLock，可以显式加锁、释放锁。

## synchronized与Lock的对比

- Lock是显式锁（手动开启和关闭锁，别忘记关闭锁），synchronized是隐式锁，出了作用域自动释放
- Lock只有代码块锁，synchronized有代码块锁和方法锁
- 使用Lock锁，JVM将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性（提供更多的子类）

### 优先使用顺序：

Lock → 同步代码块（已经进入了方法体，分配了相应资源）→ 同步方法  
(在方法体之外)

### 方式三：Lock锁--->JDK5.0新增

```
public class MyThread implements Runnable {  
    private int ticket = 100;  
    private ReentrantLock lock=new ReentrantLock();  
    public void run() {  
        while(true){  
            try {  
                lock.lock(); //锁上方法  
                if (ticket > 0) {  
                    System.out.println(Thread.currentThread().getName()  
                        + ":卖票，票号为：" + ticket);  
                    ticket--;  
                } else {  
                    break;  
                }  
            }finally {  
                lock.unlock(); //解锁方法  
            }  
        }  
    }  
}
```

synchronized与lock异同：

相同：都可以解决线程安全问题

不同：synchronized机制在执行完相应的同步代码后自动

释放同步监视器

Lock需要手动开启同步lock(), 手动结束同步unlock()

保证Lock对几个线程是唯一的

# 64.Java高级-线程的通信

2021年8月12日 16:18

线程通信例子：两个线程交替打印1-100

```
public class Communication implements Runnable{
    private int number=1;
    @Override
    public void run() {
        while (true){
            synchronized (this) {
                notifyAll(); //唤醒所有等待
                if (number <= 100) {
                    System.out.println(Thread.currentThread().getName() + number);
                    number++;
                    try {
                        //使得调用此方法的线程进入阻塞状态
                        wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                } else {
                    break;
                }
            }
        }
    }
}
```

涉及的三个方法：

wait():一旦执行此方法，当前线程就进入阻塞状态，并释放同步监视器

notify(): 一旦执行此方法，就会唤醒被wait的一个线程，如果有多个，唤醒优先级高的。

notifyAll():一旦执行此方法，就会唤醒所有被wait的线程

说明：

1.wait(),notify(),notifyAll()三个方法必须使用在同步代码块或者同步方法中

2.三个方法的调用者必须是同步代码块或同步方法中的同步监视器

否则，会出现IllegalMonitorStateException异常

3.三个方法定义在java.lang.Object类中

sleep () 和wait () 的异同

相同点：

都可以使线程进入阻塞状态

不同点：

1.声明位置不同

2.sleep()可以在任何需要的场景下使用， wait()必须在同步代码块/同步方法中

3.如果两者都在同步方法/同步代码块中使用， sleep不会释放同步监视器

而wait会

```
class Clerk{
    private int productCount=0;
    //两个同步方法使用同一把锁this，所有一个进去，另一个需要等待
    public synchronized void produceProduct(){
        if(productCount<20){
            System.out.println(Thread.currentThread().getName()
                +"开始生产第"+(productCount+1)+"个产品");
            productCount++;
            notify(); //生产后唤醒消费者
        }else{
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public synchronized void consumeProduct(){
        if(productCount>0){
            System.out.println(Thread.currentThread().getName()
                +"开始消费第"+productCount+"个产品");
            productCount--;
            notify(); //消费后唤醒生产者
        }else {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

## 生产者消费者问题总结

使用同一把锁的两个同步方法，允许一个线程调用

将同一个对象传入不同线程，操作一个对象中的数据

# 65.Java高级-JDK5.0新增线程创建

2021年8月12日 16:57

创建线程的方式三：实现Callable接口

1. 创建一个实现Callable的实现类
2. 实现call()方法，将此线程所需操作放入
3. 创建Callable()接口实现类的对象
4. 将此Callable()接口实现类的对象作为参数传递到FutureTask构造器中，创建FutureTask对象
5. 将FutureTask的对象作为参数传递到Thread类的构造器中创建Thread对象，并调用start()
6. 可以获取call方法的返回值

```
//1. 创建一个实现Callable的实现类
public class ThreadNew implements Callable {
    @Override
    //2. 实现call()方法，将此线程所需操作放入
    public Object call() throws Exception {
        int sum=0;
        for (int i = 0; i <=100; i++) {
            if(i%2==0){
                System.out.println(i);
                sum+=i;
            }
        }
        return sum;
    }
}

class ThrewNewTest{
    public static void main(String[] args) {
        //3. 创建Callable()接口实现类的对象
        ThreadNew threadnew=new ThreadNew();
        //4. 将此Callable()接口实现类的对象作为参数传递到FutureTask
        //构造器中，创建FutureTask对象
        FutureTask futureTask=new FutureTask(threadnew);
        //5. 将FutureTask的对象作为参数传递到Thread类的构造器中创建
        //Thread对象，并调用start()
        new Thread(futureTask).start();
        try {
            //6. 可以获取call方法的返回值
            //get()返回值即是重写call的返回值
            Object sum=futureTask.get();
            System.out.println(sum);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

如何理解实现Callable接口比实现Runnable创建多线程更强大

- 1.call()可以有返回值
- 2.call()可以抛出异常，被外面操作捕获，获取异常信息
- 3.Callable是支持泛型的

## 创建线程的方式四：使用线程池

- **背景：**经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。
- **思路：**提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。
- **好处：**
  - 提高响应速度（减少了创建新线程的时间）
  - 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
  - 便于线程管理
    - ✓ corePoolSize: 核心池的大小
    - ✓ maximumPoolSize: 最大线程数
    - ✓ keepAliveTime: 线程没有任务时最多保持多长时间后会终止
    - ✓ ...

让天下没有难学的技术

## 线程池相关API

- JDK 5.0起提供了线程池相关API：**ExecutorService** 和 **Executors**
- **ExecutorService：**真正的线程池接口。**常见子类ThreadPoolExecutor**
  - void execute(Runnable command)：执行任务/命令，没有返回值，一般用来执行Runnable
  - <T> Future<T> submit(Callable<T> task)：执行任务，有返回值，一般又来执行Callable
  - void shutdown()：关闭连接池
- **Executors：**工具类、线程池的工厂类，用于创建并返回不同类型的线程池
  - Executors.newCachedThreadPool()：创建一个可根据需要创建新线程的线程池
  - Executors.newFixedThreadPool(n)：创建一个可重用固定线程数的线程池
  - Executors.newSingleThreadExecutor()：创建一个只有一个线程的线程池
  - Executors.newScheduledThreadPool(n)：创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

让天下没有难学的技术

```
public class ExecutorTest {
    public static void main(String[] args) {
        //1.提供指定线程数量的线程池,这里有多态性
        ExecutorService service=Executors.newFixedThreadPool( nThreads: 10);
        //如果想设置属性需要向下转型为接口的具体实现类

        //2.执行指定线程的操作
        //service.execute(new IntPut());//适用于Runnable
        //service.submit(new IntPut());//适用于Callable
        //3.关闭线程池
        service.shutdown();
    }
}

class IntPut implements Callable { //Callable/Runnable
    @Override
    public Object call() throws Exception {
        for(int i=0;i<101;++i){
            System.out.println(i);
        }
    }
}
```

```
public Object call() throws Exception {  
    for(int i=0;i<101;++i){  
        System.out.println(i);  
    }  
    return null;  
}
```

Callable接口中是一个有返回值的call方法。

这种提交的方式会返回一个Future对象，这个Future对象代表这线程的执行结果，当主线程调用Future的get方法的时候会获取到从线程中返回的结果数据。

Future是接口，FutureTest实现了RunnableFuture接口，而后者又实现了Future 接口

# 66.常用类-String

2021年8月12日 19:48

String:字符串，使用一对 "" 表示

1.String声明为final的，不可被继承

2.String实现了Serializable接口：表示字符串是支持序列化的

实现了Comparable接口：表示String可以比较大小

3.String内部定义了final char[] value 用于存储字符串数据

4.String：代表不可变的字符序列。简称：不可变性

体现：

-对字符串重新赋值时，需要重新指定内存区域赋值，不能使用原有的value数组

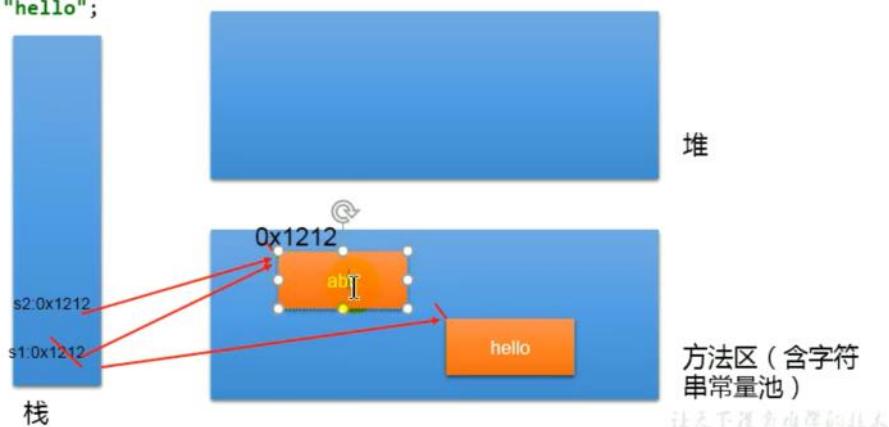
-对现有的字符串进行连接操作时，也需要重新指定内存区域

-当调用String的replace()方法修改字符串时，也需要重新指定

5.通过字面量的方式给字符串赋值，此时的字符串值声明在字符串常量池中

6.字符串常量池不会存相同内容的字符串

```
String s1 = "abc"; //字面量的定义方式
String s2 = "abc";
s1 = "hello";
```



7.String的实例化方式

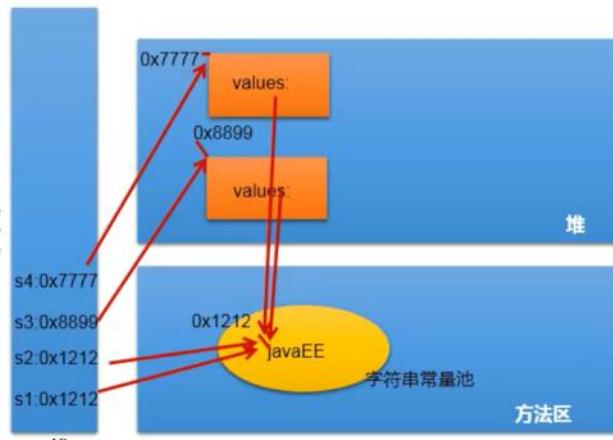
方式一：通过字面量定义的方式

方式二：通过new+构造器的方式

## 练习1

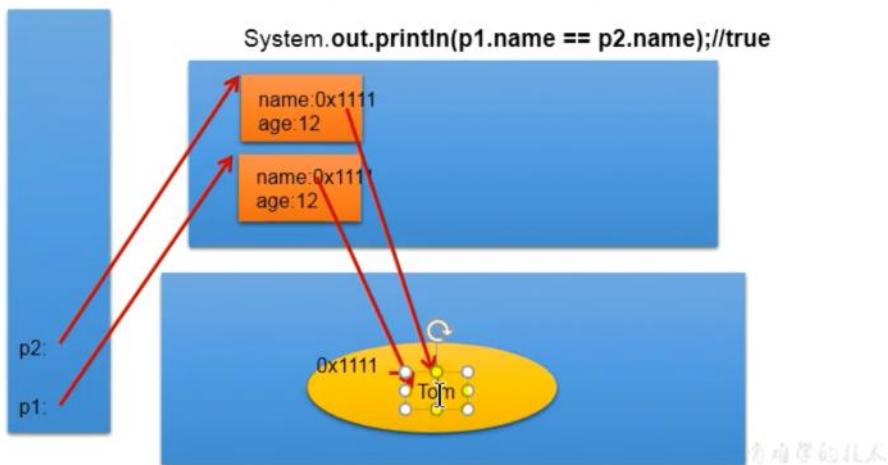
```
String s1 = "javaEE";
String s2 = "javaEE";
String s3 = new String("javaEE");
String s4 = new String("javaEE");

System.out.println(s1 == s2); //true
System.out.println(s1 == s3); //false
System.out.println(s1 == s4); //false
System.out.println(s3 == s4); //false
```



```
Person p1 = new Person("Tom",12);
Person p2 = new Person("Tom",12);

System.out.println(p1.name == p2.name); //true
```



## 8.不同拼接方式对比

```
String s1 = "javaEE";
String s2 = "hadoop";

String s3 = "javaEEhadoop";
String s4 = "javaEE" + "hadoop";
String s5 = s1 + "hadoop";
String s6 = "javaEE" + s2;
String s7 = s1 + s2;
```

```
System.out.println(s3 == s4); //true
System.out.println(s3 == s5); //false
System.out.println(s3 == s6); //false
System.out.println(s3 == s7); //false
System.out.println(s5 == s6); //false
System.out.println(s5 == s7); //false
System.out.println(s6 == s7); //false
```

结论：

- 常量与常量的拼接结果在常量池，且不会存在相同内容的常量
- 只要有一个是变量，结果就在堆中
- 如果拼接的结果调用intern()方法，返回值就在常量池中

# 67. 常用类-String常用方法

2021年8月13日 10:31

1.int length():返回字符串的长度

2.char charAt(int index):返回某索引处的字符

3.boolean isEmpty():判断字符串是否为空字符串

4.String toLowerCase():将所有字符转换为小写

5.String toUpperCase():

6.String trim():返回字符串的副本，忽略前导和尾部空白

7.boolean equals(Object obj):比较字符串的内容是否相等

8.boolean equalsIgnoreCase():与equals()相似，只是忽略大小写

9.String concat(String str):连接字符串，等价于+

10.int compareTo(String anotherString):比较两个字符串的大小，大正小负

11.String substring( int beginIndex): 截取字符串的子串

12.String substring(int begin,int end):截取字符串的子串，前闭后开

13.boolean endsWith(String str):测试此字符串是否以指定字符串结尾

14.boolean startsWith():

15.boolean startsWith(String prefix,int toffset):测试此字符串从指定索引开始的子字符串是否以指定前缀开始

16.boolean contains(CharSequence s):当此字符串包含指定的char值序列时，返回true

17.int indexOf(String str):返回指定子串的第一次出现索引，没有返回-1

18.int indexOf(String str,int fromIndex)

19.int lastIndexOf(String str)

20.int lastIndexOf(String str,int fromIndex)

## 替换

21.String replace(char oldChar,char newChar):返回一个新的字符串，替换指定字符

22.String replace(CharSequence target,CharSequence replacement):使用指定字面值替换子串。

23.String replaceAll(String regex,String replacement):使用指定字符串替换所有满足正则表达式的子串

24.String replaceFirst(String regex,String replacement)

## 匹配

25.Boolean matches(String regex):告知此字符串是否匹配给定的正则表达式

## 切片

26.String[] split(String regex):根据正则表达式的匹配拆分此字符串

27.String[] split(String regex,int limit):不超过limit个

String--->char[]之间的转换:

toCharArray():调用String的此方法

char[]->String: 调用String的构造器

编码: String--->byte[]:调用String的getBytes(),使用默认的字符集进行编码

解码: byte[]--->String:调用String的构造器, (选择重写的对应构造器)

编码: 字符串--->字节 (看得懂-->看不懂的二进制数据)

解码: 字节--->字符串 (看不懂的二进制数据--->看得懂)

说明: 解码时, 必须要求解码与编码所用字符集一致,

否则会出现乱码

# 68. 常用类-StringBuffer和StringBuilder

2021年8月13日 16:27

String: 不可变的字符序列

StringBuffer: 可变的字符序列; 线程安全, 效率低。

StringBuilder: 可变的字符序列; JDK5.0新增, 是线程不安全的, 效率高

三者底层都使用char[]存储

效率排列: StringBuilder>StringBuffer>String

源码分析:

```
String str=new String(); //char[] value=new char[0];
String str1=new String( original: "abc"); //char[] value=char[]{'a','b','c'};

StringBuffer sb1=new StringBuffer(); //char [] value=new char[16]; 底层创建了长度为16的char[]数组
sb1.append('a'); //value[0]='a';
StringBuffer sb2=new StringBuffer("abc"); //char[] value=new char["abc".length+16];
```

扩容问题:

如果要添加的数据底层数组放不下了, 就需要扩容底层数组

默认情况下扩容为原来的两倍+2, 同时将原来的数组复制过去

指导建议:

开发中建议使用: StringBuffer(int capacity)或StringBuilder(int capacity)

指定初始容量

StringBuffer的常用方法

增

StringBuffer append(xxx): 拼接字符串

删

StringBuffer delete(int start, int end): 删除指定位置内容

StringBuffer replace(int start, int end, String str): 把[start, end]位置替换为str

插

StringBuffer insert(int offset, xxx): 在指定位置插入xxx

StringBuffer reverse(): 把当前字符串序列反转

public int indexOf(String str)

public String substring(int start, int end): 返回一个子串

长度

public int length()

查

```
public char charAt(int n)
```

改

```
public void setCharAt(int n,char ch)
```

使用toString(),输出字符串

# 69.JDK8之前日期和时间API

2021年8月13日 17:30

```
//JDK8之前日期和时间API
//1.system类中的currentTimeMillis()
long t1=System.currentTimeMillis(); //获取时间戳
System.out.println(t1);

//2.java.util.Date类
//----两个构造器的使用-----
//构造器一Date()创建当前时间的Date对象
Date date=new Date();
//构造器二：创建指定时间戳的Date对象
Date date1=new Date(1000000000L);
//----两个方法的使用-----
System.out.println(date.toString()); //显示Fri Aug 13 17:40:06 CST 2021
System.out.println(date.getTime()); //获取当前对象的时间戳

//3.java.sql.Date (util下Date的子类) 对应着数据库中的日期类型的变量
//实例化
java.sql.Date date2=new java.sql.Date(111111111);
System.out.println(date2.toString()); //1970-01-02,只显示年月日
//将java.util.Date对象转换为java.sql.Date
Date date3=new Date();
java.sql.Date date4=new java.sql.Date(date3.getTime());
```

4.java.text.SimpleDateFormat

```
//SimpleDateFormat的实例化
SimpleDateFormat sdf =new SimpleDateFormat(); //默认构造器
//格式化：日期--->字符串
Date date=new Date();
System.out.println(date); //Sat Aug 14 11:18:39 CST 2021
String format=sdf.format(date);
System.out.println(format); //21-8-14 上午11:18
//解析：格式化的逆过程，字符串--->日期
String str="21-8-14 上午11:18";
Date date1= null;
try {
    date1 = sdf.parse(str);
} catch (ParseException e) {
    e.printStackTrace();
}
System.out.println(date1); //Sat Aug 14 11:18:00 CST 2021
//按照指定方式格式化和解析：调用带参的构造器
SimpleDateFormat sdf1=new SimpleDateFormat( pattern: "yyyy-MM-dd hh:mm:ss");
//格式化
System.out.println(sdf1.format(date)); //2021-08-14 11:31:09
//解析，类型不符合构造器格式的会报异常
Date date2= null;
try {
    date2 = sdf1.parse( source: "2021-08-14 11:30:55");
} catch (ParseException e) {
    e.printStackTrace();
}
System.out.println(date2); //Sat Aug 14 11:30:55 CST 2021
```

## 5.Calendar

```
//Calendar (抽象类)
//实例化方式一：创建子类（GregorianCalendar）的对象
//实例化方式二：调用其静态方法getInstance()
Calendar calendar=Calendar.getInstance();
//常用方法
//get()
int days=calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(days); //第14天
//set()
calendar.set(Calendar.DAY_OF_MONTH, 22);
days=calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(days); //22
//add()
calendar.add(Calendar.DAY_OF_MONTH, amount: -3);
days=calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(days); //19
//getTime():日历类-->Date
Date date=calendar.getTime();
//setTime():Date-->日历类
Date date1=new Date();
calendar.setTime(date1);
```

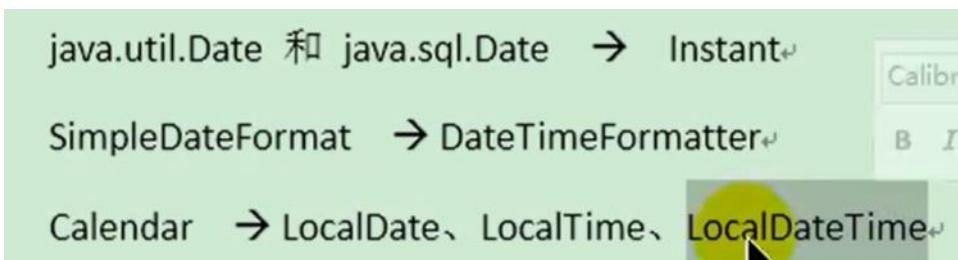
# 70.JDK8中新日期时间API

2021年8月14日 12:08

```
//LocalDate、LocalTime、LocalDateTime的使用
//now():获取当前的日期、时间、日期加时间
LocalDate localDate=LocalDate.now();
LocalTime localTime=LocalTime.now();
LocalDateTime localDateTime=LocalDateTime.now();
//of():设置指定的年、月、日、时、分、秒，没有偏移量
LocalDateTime localDateTime1=LocalDateTime.of( year: 2021, month: 8, dayOfMonth: 24, hour: 12, minute: 42);
//getXxx():获取相关属性
localDateTime.getDayOfMonth();
localDateTime.getDayOfWeek();
localDateTime.getDayOfYear();
//体现不可变性
//withXxx:设置相关属性
LocalDate localDate1=localDate.withDayOfMonth(22);
//plusXxx:不可变性，属性加
LocalDate localDate2=localDate.plusDays(3);
//minusXxx:不可变性，属性减
LocalDate localDate3=localDate.minusDays(3);
```

```
//Instant的使用，类似于java.util.Date类
//获取本初子午线对应的时间
Instant instant=Instant.now();
//东八区时间，添加时间的偏移量
OffsetDateTime offsetDateTime=instant.atOffset(ZoneOffset.ofHours(8));
//获取UTC开始的对应毫秒数
long milli=instant.toEpochMilli();
//ofEpochMilli():通过给定的毫秒数，获取Instant实例
Instant instant1=Instant.ofEpochMilli(222222222L);
```

```
//DateTimeFormatter:格式化或解析日期、时间
//方式一：预定义的标准格式
DateTimeFormatter formatter=DateTimeFormatter.ISO_LOCAL_DATE_TIME;
//格式化：日期-->字符串
LocalDateTime localDateTime2=LocalDateTime.now();
String str1=formatter.format(localDateTime2);
//解析：字符串-->日期
TemporalAccessor parse=formatter.parse( text: "2020-08-14T13:21:33.222" );
//方式二：本地化相关的格式，有几种格式类型选择
DateTimeFormatter dateTimeFormatter=DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);
dateTimeFormatter.format(LocalDate.now());
//方式三：自定义的格式，如：ofPattern("yyyy-MM-dd" hh:mm:ss")
DateTimeFormatter formatter1=DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm:ss");
//格式化
String str=formatter1.format(LocalDateTime.now());
//解析需要与自定义格式一致
```



# 71.Java比较器

2021年8月14日 17:09

Java中的对象，正常情况下，只能进行比较==或!=，不能使用><

在开发中需要对对象排序，使用两个接口中任何一个

Comparable

Comparator

Comparable接口的使用：自然排序

1.像String、包装类等实现了Comparable接口，重写了compareTo()

方法，给出了比较两个对象的方式

2.像String、包装类重写compareTo()方法后，进行了从小到大的排列

3.重写compareTo(obj)的规则

如果当前对象this大于形参对象obj，返回正整数

如果小于返回负整数

如果等于返回零

4.如果自定义类需要排序，我们需要实现Comparable接口，

重写compareTo()方法，在方法中声明如何排序

5.如果是对象数组，直接调用Arrays.sort(arr)排序

```
//指明商品比较大小的方式:照价格从低到高排序,再照产品名称从高到低排序
@Override
public int compareTo(Object o) {
    System.out.println("*****");
    if(o instanceof Goods){
        Goods goods = (Goods)o;
        //方式一:
        if(this.price > goods.price){
            return 1;
        }else if(this.price < goods.price){
            return -1;
        }else{
            return 0;
        }
        //方式二:
        return Double.compare(this.price,goods.price);
    }
    return 0;
    throw new RuntimeException("传入的数据类型不一致！");
}
```

Comparator接口的使用：定制排序

1.背景

当元素的类型没有实现java.lang.Comparable接口而又不方便修改代码

或者实现了java.lang.Comparable接口的排序规则不适合当前的操作，

可以考虑使用Comparator的对象来排序

## 2.重写compare(Object o1, Object o2)

```
Comparator com = new Comparator() {
    //指明商品比较大小的方式：照产品名称从低到高排序，再照价格从高到低排序
    @Override
    public int compare(Object o1, Object o2) {
        if(o1 instanceof Goods && o2 instanceof Goods){
            Goods g1 = (Goods)o1;
            Goods g2 = (Goods)o2;
            if(g1.getName().equals(g2.getName())){
                return -Double.compare(g1.getPrice(),g2.getPrice());
            }else{
                return g1.getName().compareTo(g2.getName());
            }
        }
        throw new RuntimeException("输入的数据类型不一致");
    }
}

使用：
Arrays.sort(goods,com);
```

两者对比：

Comparable接口：一旦指定，保证实现类对象任何位置都可以比较大小

Comparator接口：属于临时性的比较

## 72.System/Math/BigInteger/BigDecimal

2021年8月14日 18:26

System类：

void exit(int status):status=0正常退出，status=1异常退出

void gc():请求系统进行垃圾回收

### ➤ String getProperty(String key):

该方法的作用是获得系统中属性名为key的属性对应的值。系统中常见的属性名以及属性的作用如下表所示：

属性名	属性说明
java.version	Java 运行时环境版本
java.home	Java 安装目录
os.name	操作系统的名称
os.version	操作系统的版本
user.name	用户的账户名称
user.home	用户的主目录
user.dir	用户的当前工作目录

Math类

**java.lang.Math**提供了一系列静态方法用于科学计算。其方法的参数和返回值类型一般为**double**型。

**abs** 绝对值

**acos,asin,atan,cos,sin,tan** 三角函数

**sqrt** 平方根

**pow(double a,doble b)** a的b次幂

**log** 自然对数

**exp** e为底指数

**max(double a,double b)**

**min(double a,double b)**

**random()** 返回0.0到1.0的随机数

**long round(double a)** double型数据a转换为long型（四舍五入）

**toDegrees(double angrad)** 弧度—>角度

**toRadians(double angdeg)** 角度—>弧度

BigInteger类

- **Integer**类作为int的包装类，能存储的最大整型值为 $2^{31}-1$ ，**Long**类也是有限的，最大为 $2^{63}-1$ 。如果要表示再大的整数，不管是基本数据类型还是他们的包装类都无能为力，更不用说进行运算了。| **I**
- **java.math**包的**BigInteger**可以表示不可变的任意精度的整数。**BigInteger** 提供所有 Java 的基本整数操作符的对应物，并提供 **java.lang.Math** 的所有相关方法。另外，**BigInteger** 还提供以下运算：模算术、GCD 计算、质数测试、素数生成、位操作以及一些其他操作。
- 构造器
  - **BigInteger(String val)**: 根据字符串构建**BigInteger**对象

## BigDecimal类

- 一般的**Float**类和**Double**类可以用来做科学计算或工程计算，但在**商业计算中，要求数字精度比较高，故用到**java.math.BigDecimal**类。**
- **BigDecimal**类支持不可变的、任意精度的有符号十进制定点数。
- 构造器
  - **public BigDecimal(double val)**
  - **public BigDecimal(String val)**
- 常用方法
  - **public BigDecimal add(BigDecimal augend)**
  - **public BigDecimal subtract(BigDecimal subtrahend)**
  - **public BigDecimal multiply(BigDecimal multiplicand)**
  - **public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode)**

# 73. 枚举类

2021年8月14日 19:26

- 类的对象只有有限个，确定的。举例如下：

- 星期：Monday(星期一)、.....、Sunday(星期天)
- 性别：Man(男)、Woman(女)
- 季节：Spring(春节).....Winter(冬天)
- 支付方式：Cash(现金)、WeChatPay(微信)、Alipay(支付宝)、BankCard(银行卡)、CreditCard(信用卡)
- 就职状态：Busy、Free、Vocation、Dimission
- 订单状态：Nonpayment(未付款)、Paid(已付款)、Fulfilled(已配货)、Delivered(已发货)、Return(退货)、Checked(已确认)
- 线程状态：创建、就绪、运行、阻塞、死亡

- 当需要定义一组常量时，强烈建议使用枚举类

## 一、枚举类的使用

1. 枚举类的理解：类的对象只有有限个，确定的。
2. 当需要定义一组常量时，建议使用枚举类
3. 如果枚举类中只有一个对象，则可以作为单例模式的实现方式。

## 二、定义枚举类

方式一：JDK5.0之前，自定义枚举类

```
//自定义枚举类
class Season{
    //1. 声明Season对象的属性：private final修饰
    private final String seasonName;
    private final String seasonDesc;
    //2. 私有化类的构造器
    private Season(String seasonName, String seasonDesc){
        this.seasonName=seasonName;
        this.seasonDesc=seasonDesc;
    }
    //3. 提供当前枚举类的对象
    public static final Season SPRING=new Season("春天", "春暖花开");
    public static final Season SUMMER=new Season("夏天", "烈日炎炎");
    //4. 其他诉求1：获取枚举类对象的属性
    public String getSeasonName() {
        return seasonName;
    }
    public String getSeasonDesc() {
        return seasonDesc;
    }
    //其他诉求2：重写toString()方法
    @Override
    public String toString() {
        return "Season{" +
            "seasonName='" + seasonName + '\'' +
            ", seasonDesc='" + seasonDesc + '\'' +
            '}';
    }
}
```

## 方式二：JDK5.0,可以使用enum关键字自定义枚举类

```
//使用enum关键字定义枚举类，父类为java.lang.Enum
enum Season{
    //1.提供当前枚举类的对象，多个对象之间用","隔开，末尾用";"结束
    SPRING(seasonName: "春天", seasonDesc: "春暖花开"),
    SUMMER(seasonName: "夏天", seasonDesc: "烈日炎炎");

    //2.声明Season对象的属性: private final修饰
    private final String seasonName;
    private final String seasonDesc;

    //3.私有化类的构造器，并给对象赋值
    private Season(String seasonName, String seasonDesc){
        this.seasonName=seasonName;
        this.seasonDesc=seasonDesc;
    }

    //4.其他诉求1：获取枚举类对象的属性
    public String getSeasonName() {
        return seasonName;
    }

    public String getSeasonDesc() {
        return seasonDesc;
    }

    //其他诉求2：选择重写toString()方法，不重写默认为输出类的名字
    @Override
    public String toString() {
        return "Season{" +
            "seasonName='" + seasonName + '\'' +
            ", seasonDesc='" + seasonDesc + '\'' +
            '}';
    }
}
```

### 三、Enum类主要方法

values():返回枚举类型的对象数组。该方法可方便遍历所有的枚举值

toString():返回当前枚举类对象常量的名称

valueOf(String str):根据对象的名称，返回对应的对象,没有则抛异常

### 四、使用enum关键字定义的枚举类实现接口情况

情况一：实现接口，在enum类中实现抽象方法

情况二：每个枚举类对象后面都使用{}，在其中实现抽象方法

实现个性化方法实现，调用专属方法

# 74.注解 (Annotation)

2021年8月15日 10:45

- 从 JDK 5.0 开始, Java 增加了对元数据(MetaData)的支持, 也就是 Annotation(注解)
- Annotation 其实就是代码里的**特殊标记**, 这些标记可以在编译, 类加载, 运行时被读取, 并执行相应的处理。通过使用 Annotation, 程序员可以在不改变原有逻辑的情况下, 在源文件中嵌入一些补充信息。代码分析工具、开发工具和部署工具可以通过这些补充信息进行验证或者进行部署。
- Annotation 可以像修饰符一样被使用, 可用于**修饰包,类, 构造器, 方法, 成员变量, 参数, 局部变量的声明**, 这些信息被保存在 Annotation 的“name=value”对中。
- 在JavaSE中, 注解的使用目的比较简单, 例如标记过时的功能, 忽略警告等。在JavaEE/Android中注解占据了更重要的角色, 例如用来配置应用程序的任何切面, 代替JavaEE旧版中所遗留的繁冗代码和XML配置等。
- 未来的开发模式都是基于注解的, JPA是基于注解的, Spring2.5以上都是基于注解的, Hibernate3.x以后也是基于注解的, 现在的Struts2有一部分也是基于注解的了, 注解是一种趋势, 一定程度上可以说: 框架 = 注解 + 反射 + 设计模式。

## Annotation示例

- 使用 Annotation 时要在其前面增加 @ 符号, 并**把该 Annotation 当成一个修饰符使用**。用于修饰它支持的程序元素

### ●示例一：生成文档相关的注解

@author 标明开发该类模块的作者, 多个作者之间使用, 分割  
@version 标明该类模块的版本  
@see 参考转向, 也就是相关主题  
@since 从哪个版本开始增加的  
@param 对方法中某参数的说明, 如果没有参数就不能写  
@return 对方法返回值的说明, 如果方法的返回值类型是void就不能写  
@exception 对方法可能抛出的异常进行说明, 如果方法没有用throws显式抛出的异常就不能写其中  
    @param @return 和 @exception 这三个标记都是只用于方法的。  
    @param的格式要求: @param 形参名 形参类型 形参说明  
    @return 的格式要求: @return 返回值类型 返回值说明  
    @exception的格式要求: @exception 异常类型 异常说明  
    @param和@exception可以并列多个

让天下没有难学的技术

### ●示例二：在编译时进行格式检查(JDK内置的三个基本注解)

➢@Override: 限定重写父类方法, 该注解只能用于方法 I

➢@Deprecated: 用于表示所修饰的元素(类, 方法等)已过时。通常是因为所修饰的结构危险或存在更好的选择

➢@SuppressWarnings: 抑制编译器警告

### ● 示例三：跟踪代码依赖性，实现替代配置文件功能

➤ Servlet3.0提供了注解(annotation),使得不再需要在web.xml文件中进行Servlet的部署。

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {}
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        doGet(request, response);
    }
}
```

```
<servlet>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>com.servlet.LoginServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/login</url-pattern>
</servlet-mapping>
```

不推荐这样写

### ➤ spring框架中关于“事务”的管理

```
@Transactional(propagation=Propagation.REQUIRES_NEW,
    isolation=Isolation.READ_COMMITTED,readOnly=false,timeout=3)
public void buyBook(String username, String isbn) {
    //1.查询书的单价
    int price = bookShopDao.findBookPriceByisbn(isbn);
    //2.更新库存
    bookShopDao.updateBookStock(isbn);
    //3.更新用户的余额
    bookShopDao.updateUserAccount(username, price);
}
```

```
<!-- 配置事务属性 -->
<tx:advice transaction-manager="dataSourceTransactionManager" id="txAdvice">
    <tx:attributes>
        <!-- 配置每个方法使用的事务属性 -->
        <tx:method name="buyBook" propagation="REQUIRES_NEW"
            isolation="READ_COMMITTED" read-only="false" timeout="3" />
    </tx:attributes>
</tx:advice>
```

注解的使用：

1.理解Annotation

见上一

2.Annotation的使用示例

见三个示例

3.如何自定义注解（用的少）

参照SuppressWarnings

①注解声明为：@interface

②内部定成员，通常使用value表示

③可以指定默认值，使用default定义

④如果自定义注解没有成员，表明是一个表示作用

⑤如果注解有成员，使用时需要指定值

自定义注解必须配上注解的信息处理流程（使用反射）才有意义

```
public @interface MyAnnotation {  
    String value() default "hello";  
}
```

- 定义新的 Annotation 类型使用 `@interface` 关键字
- 自定义注解自动继承了 `java.lang.annotation.Annotation` 接口
- Annotation 的成员变量在 Annotation 定义中以无参数方法的形式来声明。其方法名和返回值定义了该成员的名字和类型。我们称为配置参数。类型只能是八种基本数据类型、String 类型、Class 类型、enum 类型、Annotation 类型、以上所有类型的数组。
- 可以在定义 Annotation 的成员变量时为其指定初始值，指定成员变量的初始值可使用 `default` 关键字
- 如果只有一个参数成员，建议使用 `参数名为value`
- 如果定义的注解含有配置参数，那么使用时必须指定参数值，除非它有默认值。格式是“参数名 = 参数值”，如果只有一个参数成员，且名称为 `value`，可以省略“`value=`”
- 没有成员定义的 Annotation 称为 **标记**；包含成员变量的 Annotation 称为元数据 **Annotation**

注意：自定义注解必须配上注解的信息处理流程才有意义。

#### 4.JDK中的元注解

①元注解：对现有的注解进行解释说明的注解

\*\*\*\*\*自定义注解常用

Retention:指定所修饰的Annotation的声明周期：

SOURCE\CLASS(默认)\RUNTIME放在枚举类中

只有声明为RUNTIME生命周期的注解，才能通过反射获取

`@Retention(RetentionPolicy.RUNTIME)`

Target:指定被修饰的注解能修饰哪些程序元素

\*\*\*\*\*使用较少

Documented:表示所修饰的注解在被javadoc解析时保留下来

Inherited:被它修饰的Annotation将具有继承性，父类使用注解，子类也会继承

#### 5.通过反射获取注解信息--见后面反射

#### 6.JDK8中注解的新特性：可重复注解、类型注解

①可重复注解（例）

--在MyAnnotation上声明@Repeatable,成员值为MyAnnotations.class 【成员变量为MyAnnotation的数组的注解】

--MyAnnotation的Target/Retention/Inherited和MyAnnotations相同。

## ②类型注解

ElementType.TYPE\_PARAMETER 表示该注解能写在类型变量的声明语句中（如：泛型声明）

ElementType.TYPE\_USE 表示该注解能写在使用类型的任何语句中

（加入到Target中就会支持）

# 75.Java集合框架

2021年8月15日 13:43

## 一、集合框架的概述

1. 集合、数组都是对多个数据进行存储操作的结构，简称Java容器

说明：此时的存储，主要指的是内存层面的存储，不涉及到持久化的存储  
(.txt,.jpg,.avi,数据库中)

## 2. 数组

--存储多个数据方面的特点：

>一旦初始化以后，长度就确定了

>一旦定义好，其元素的类型也就确定了，只能操作指定类型的数据

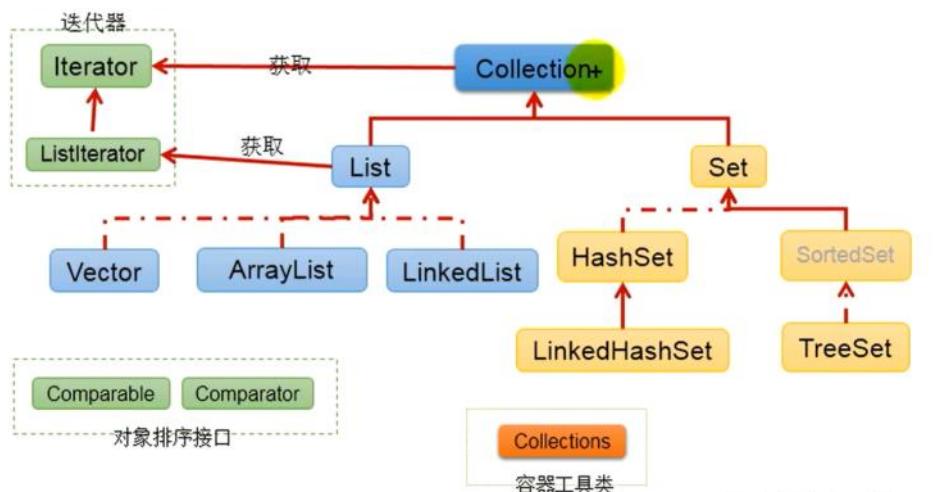
--数组的缺点：

>一旦长度确定，不可修改长度

>数组中提供的方法有限，插入删除数据十分不方便，效率低

>获取数组中实际元素的个数需求，没有现成的方法

>数组存储的特点：有序、可重复。对于无序、不可重复的需求不能满足



## 二、集合框架

|----Collection接口：单列集合，用来存储一个一个的对象

    |---List接口：存储有序的、可重复的数据。 --->(动态数组)

        |---ArrayList、LinkedList、Vector

    |---Set接口：存储无序的、不可重复的数据。 --->(高中讲的"集合")

        |---HashSet、LinkedHashSet、TreeSet

|----Map接口：双列集合，用来存储一对一对(key--value)的数据---->函数：y=f(x)

    |---HashMap、LinkedHashMap、TreeMap、Hashtable、Properties

## 三、Collection接口中的方法的使用

```

//collection接口中方法的使用
Collection coll=new ArrayList();
//1.add(Object e):将元素e添加到集合coll中
coll.add("aa");
coll.add(22);

//2.int size():获取添加的元素个数
System.out.println(coll.size());

//3.addAll(Collection coll):将一个集合中的元素添加到当前集合
Collection coll1=new ArrayList();
coll1.add(11);
coll1.add("abcc");
coll.addAll(coll1);

//4.clear():清空集合元素
coll.clear();

//5.boolean isEmpty():判断当前集合是否为空(size=0)
coll.isEmpty();

//6.boolean contains(Object obj):判断当前集合是否包含obj,本质调用obj对象的equals()方法
//比较内容还是地址需按equals()判断,向Collection接口实现类的对象中添加数据obj时,
//要求obj所在的类要重写equals()
coll.contains(22);
```



```

//7.boolean containsAll(Collection coll):判断形参集合中所有元素是否都在当前集合中
coll.containsAll(coll1);

//8.boolean remove(Object obj):true删除成功, false失败, 从当前集合删除obj元素
coll.remove(22);

//9.boolean removeAll(Collection coll1):差集:从当前集合移除参数集合相同的所有元素
coll.removeAll(coll1);

//10.retainAll(Collection coll1):交集:获取当前集合与参数集合的交集, 保存给当前集合
coll.retainAll(coll1);

//11.boolean equals(Object obj):判断当前集合和形参集合元素是否相同
coll.equals(coll1);

//12.hashCode():返回当前对象的哈希值
System.out.println(coll.hashCode());

//13.集合--->数组: toArray()
Object[] arr=coll.toArray();
//拓展:数组--->集合:调用Arrays类的静态方法asList()
//如果是基本类型数组会识别为一个数组元素, 需要传入包装类数组
List list= Arrays.asList(new String[]{"11","abc"});

//14.iterator():返回Iterator接口的实例, 用于遍历集合元素。

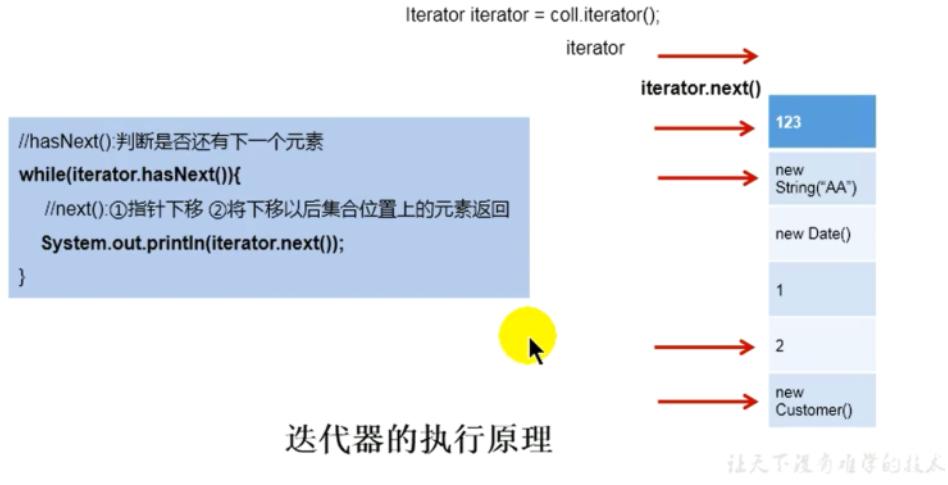
```

#### 四、Collection集合元素的遍历，使用迭代器Iterator接口

##### 1.内部的方法：hasNext()与next()

```

Collection coll2=new ArrayList();
coll2.add("aa");
coll2.add(22);
Iterator iterator=coll2.iterator();
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
```



## 2 迭代常见问题

```

//错误方式一：缺元素
Iterator iterator=coll2.iterator();
while (iterator.next()!=null){
    System.out.println(iterator.next());
}

//错误方式二：集合对象每次调用iterator()方法都会得到一个全新的迭代器对象，指向第一个元素之前
while(coll2.iterator().hasNext()){
    System.out.println(coll2.iterator().next());
}

//异常
//NoSuchElementException - 如果迭代没有更多的元素 |

```

4. 内部定义了remove(), 可以在遍历的时候, 删除集合中的元素。

此方法不同于集合直接调用remove()

```

Collection coll2=new ArrayList();
coll2.add(66);
coll2.add(33);
coll2.add(22);
Iterator iterator= coll2.iterator();
while(iterator.hasNext()){
    int temp=(int)iterator.next();
    if(33==temp){
        iterator.remove();
    }
}

//IllegalStateException - 如果 next方法尚未被调用, 或者 remove方法在上次调用
// next方法之后已经被调用, 会报异常

```

5.jdk5.0新增了foreach循环, 用于遍历集合、数组, (增强for循环)

```
Collection coll2=new ArrayList();
coll2.add(66);
coll2.add(33);
coll2.add(22);
//for(集合元素的类型 局部变量 : 集合对象)
//内部仍然调用了迭代器
for (Object obj:coll2){
    System.out.println(obj);
}
```

# 76.Collection子接口-List接口

2021年8月16日 11:26

## 一、List框架

|---Collection接口:单列集合，用来存储一个一个的对象

|---List接口:存储有序的、可重复的数据。 --->(动态数组)

三个实现类 |---ArrayList: 作为List接口的主要实现类，线程不安全，效率高。

底层使用Object[] elementData存储

|--LinkedList:对于频繁的插入和删除操作，使用LinkedList效率比ArrayList高

底层使用双向链表存储

|--Vector : 作为List接口的古老实现类，线程安全，效率低。

底层使用Object[] elementData存储

三个类的异同：

同：都实现了List接口，存储数据特点相同

不同：见上

## 二、ArrayList源码分析

1.Jdk 7情况下：

```
ArrayList list =new ArrayList(); //底层创建长度10的Object[] elementData  
List.add(123); //elementData[0]=new Integer(123);  
//如果某次添加导致底层数组容量不够，则扩容  
//默认情况下，扩容为原来的1.5倍，同时需要将原有数组数据复制到新数组中  
结论：开发中使用带参的构造器：ArrayList list =new ArrayList(int capacity);
```

2.Jdk 8情况下：

```
ArrayList list =new ArrayList(); //底层Object[] elementData初始化为{}，没有创建长度10  
List.add(123); //第一次调用add()时，底层才创建长度为10的数组，并添加数据  
后续与jdk7一样
```

3.jdk7中ArrayList的创建类似于单例的饿汉式， jdk8类似于懒汉式，

延迟了数组创建，节省内存

## 三、LinkedList源码分析

```
LinkedList list=new LinkedList(); //内部声明了Node类型的first和last属性，默认值为null  
List.add(123); //将123封装到Node中，创建了Node对象  
其中，Node双向链表的一个节点
```

#### 四、List接口常用方法

```
ArrayList list =new ArrayList();
list.add(12);
list.add(13);
list.add(55);
list.add(1);
//1.void add(int index, Object ele):在index位置插入ele
list.add( index: 3, element: 50);

//2.boolean addAll(int index, Collection eles):
//从index开始将集合eles中所有元素添加到当前集合
List list1=Arrays.asList(1,2,3);
list.addAll( index: 1,list1);
System.out.println(list);

//3.Object get(index):获取索引为index位置的元素
System.out.println(list.get(0));

//4.int indexOf(Object obj):返回obj在集合中首次出现的位置, 没有返回-1

//5.int lastIndexOf(Object obj)

//6.Object remove(int index):
//移除指定index位置的元素, 并返回此元素
list.remove( index: 2);

//7.Object set(int index, Object ele):设置指定位置的元素为ele
list.set(0,222);
System.out.println(list);

//8.List subList(int fromIndex,int toIndex)
//返回[fromIndex,toIndex)子集合
List subList=list.subList(1,4);
System.out.println(subList);
```

总结：常用方法

增： add(Object obj)

删： remove(int index)/remove(Object obj)

改： set()

查: get(index)

插: add(int index, Object obj)

长度: size()

遍历: ①Iterator迭代器方式

②增强for循环

③普通的循环

List也需要重写equals()方法

# 77.Collection子接口-Set接口

2021年8月16日 17:21

## 一、Set框架

|---Collection接口:单列集合，用来存储一个一个的对象

|---Set接口：存储无序的、不可重复的数据。 --->(高中讲的"集合")

|---HashSet:作为Set接口的主要实现类；线程不安全；可以存储null值

底层：数组+链表(JDK7)

|---LinkedHashSet：作为HashSet的子类，使得遍历其内部数据时可以按照添加的顺序遍历,遍历效率高于HashSet

|---TreeSet:可以按照添加对象的指定属性，进行排序

## 二、Set数据特点理解

1.Set接口中没有额外定义新的方法，使用的都是Collection中声明过的方法

以HashSet为例

2.无序性：

不等于随机性。存储的数据在底层数组中并非按照数组索引的顺序添加  
而是根据数据的哈希值

3.不可重复性：

保证添加的元素按照equals()判断时，不能返回true，  
即：相同的元素只能添加一个

4.添加元素的过程：以HashSet为例：

向HashSet中添加元素a，首先调用元素a所在类的hashCode()方法，  
计算元素a的哈希值。此哈希值接着通过某种算法计算出在HashSet底层  
数组中的哈希地址（即为：索引位置），判断数组此位置上是否有元素：

如果此位置没有其他元素，则a添加成功。----->成功①

如果此位置上有其他元素b（或以链表形式存在多个元素）则比较元素ab的哈希值：

如果hash值不同，则a添加成功----->成功②

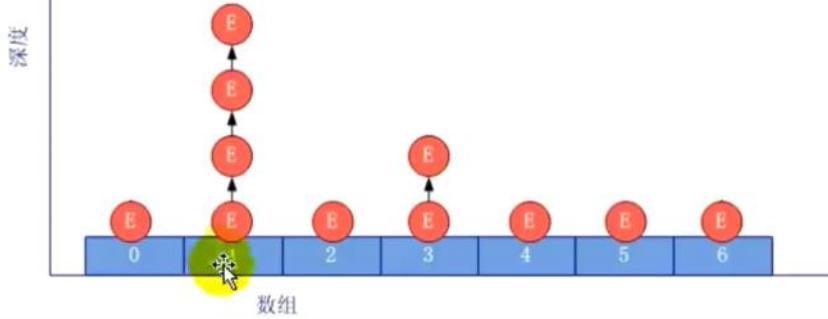
如果hash值相同，接着调用元素a所在的equals()方法：

equals()返回true,元素a添加失败

equals()返回false,元素a添加成功----->成功③

对于②③解决冲突方法：链地址法（7上8下），以链表形式存储，  
jdk7新的放上面，jdk8放下面

## Set实现类之一： HashSet



底层也是数组，初始容量为16，当如果使用率超过0.75， $(16 \times 0.75 = 12)$ 就会扩容为原来的2倍。（16扩容为32，依次为64,128....等）

### 三、 hashCode()与equals()的重写

#### Eclipse/IDEA工具里hashCode()的重写

以Eclipse/IDEA为例，在自定义类中可以调用工具自动重写equals和hashCode。  
问题：**为什么用Eclipse/IDEA复写hashCode方法，有31这个数字？**

- 选择系数的时候要选择尽量大的系数。因为如果计算出来的hash地址越大，所谓的“冲突”就越少，查找起来效率也会提高。（减少冲突）
- 并且31只占用5bits,相乘造成数据溢出的概率较小。
- 31可以由 $i * 31 == (i << 5) - 1$ 来表示,现在很多虚拟机里面都有做相关优化。（提高算法效率）
- 31是一个素数，素数作用就是如果我用一个数字来乘以这个素数，那么最终出来的结果只能被素数本身和被乘数还有1来整除！（减少冲突）

要求：

向Set中添加的数据，其所在类一定要重写hashCode()和equals()

重写的两种方法尽可能保持一致性:相等的对象必须具有相等的散列码

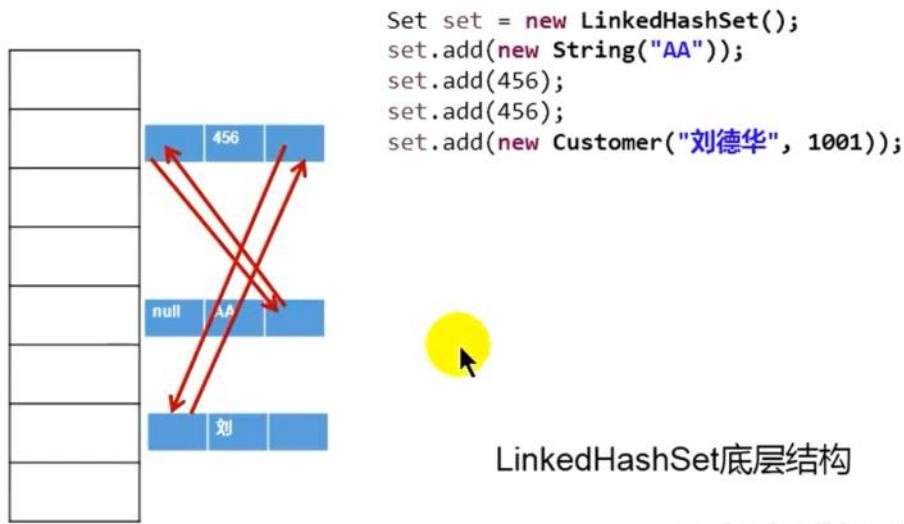
重写技巧：equals()中比较的，用来散列；直接IDE生成。

### 四、 LinkedHashSet的使用

LinkedHashSet作为HashSet的子类，在添加数据集的同时，每个元素

还维护了两个引用，记录此数据前、后元素

优点：对于频繁的遍历操作，LinkHashSet效率高于HashSet



## 五、TreeSet的使用

1. 向TreeSet中添加的数据，要求是相同类的对象。
2. 两种排序方式：自然排序(实现Comparable接口)使用空参TreeSet构造器和定制排序(Comparator) 使用带参的TreeSet构造器,参数为定制排序对象
3. 自然排序中，比较两个对象是否相等的标准为：compareTo()返回0，不再是equals()
4. 定制排序中，比较两个对象是否相同的标准是：compare()返回0，不再是equals()



再具体就不说了，可以参看<http://www.cnblogs.com/yangecnu/p/Introduce-Red-Black-Tree.html>,对红黑树的讲解写得不错。

## 关于HashSet的一个易错例题

```
public void test3(){
    HashSet set = new HashSet();
    Person p1 = new Person( id: 1001, name: "AA");
    Person p2 = new Person( id: 1002, name: "BB");

    set.add(p1);
    set.add(p2);
    System.out.println(set);

    p1.name = "CC";
    set.remove(p1);
    System.out.println(set);
    set.add(new Person( id: 1001, name: "CC"));
    System.out.println(set);
    set.add(new Person( id: 1001, name: "AA"));
    System.out.println(set);

}
```

# 78.Map接口

2021年8月17日 14:58

## 一、Map接口框架

|----Map接口：双列集合，用来存储一对一对(key--value)的数据---->函数：y=f(x)

|---HashMap：作为Map的主要实现类；线程不安全，效率高；

可以存储null, key,value；

底层：数组+链表（jdk7之前）

数组+链表+红黑树（jdk 8）

|---LinkedHashMap：保证在遍历map元素时，可以按照添加顺序

实现遍历。原因是底层结构基础上添加了前后指针

对于频繁的遍历操作，效率高于HashMap

|---TreeMap:保证按照添加的key-value对进行排序，实现排序遍历；

使用考虑key的自然排序/定制排序；

底层使用红黑树；

|---Hashtable：作为古老的实现类；线程安全，效率低；

不能存储null, key,value；

|---Properties:常用来处理配置文件。key和value都是String类型

## 二、对Map结构的理解

1.Map中的key：无序的、不可重复的，使用Set存储所有的key。--->key所在的类要重写equals()和hashCode()(以HashMap为例)

2.Map中的value:无序的、可重复的，使用Collection存储所有的value --->value所在类要重写equals()

3.一个键值对：key-value构成了一个Entry对象

4.Map中的entry：无序的、不可重复的，使用Set存储所有的entry

## 三、HashMap的底层实现原理

Jdk7为例：

1.HashMap map=new HashMap():在实例化后底层创建了长16的一位数组Entry[] table。  
...可能已经执行了多次put...

2.map.put(key1,value1):首先调用key1所在类的hashCode()计算key1的哈希值，此哈希值经过某种算法得到在Entry数组中的存放位置。  
如果此位置上的数据为空，此时的key1-value1添加成功  
如果此位置上的数据不为空,(意味着此位置上已经存在一个或者多个数据)，比较key1和已经存在的数据的哈希值：

如果key1的哈希值与已经存在数据的哈希值都不同，此时key1-value1添加成功

如果key1的哈希值与已经存在数据的哈希值相同，继续比较：调用key1所在类的equals()方法，比较：

如果equals()返回false:此时key1-value1添加成功

如果equals()返回true:使用value1替换原来的value

补充：解决冲突方法:链表

3.在不断的添加过程中，当超出临界值，且要存放的位置非空

默认扩容方式：扩容为原来的2倍，并将原有的数据复制过来

Jdk8与jdk7的一些不同：

1.new HashMap():底层没有创建一个长度为16的数组

2.jdk8底层的数组是：Node[], 不是Entry[]

3.首次调用put()方法时，底层创建长度为16的数组

4.jdk7底层只有：数组+链表

Jdk8底层：数组+链表+红黑树

当数组的某一个索引位置上的元素以链表形式存放的数据个数>8且

当前数组长度 $\geq$ 64时，此时此索引位置上的所有数据改为使用红黑树存储（查找效率高），数组长度<64，则只扩容来解决。

源码重要量

DEFAULT\_INITIAL\_CAPACITY=1<<4;// aka16: HashMap的默认值

DEFAULT\_LOAD\_FACTOR=0.75f;HashMap的默认装载因子

threshold:扩容的临界值=容量\*填充因子：16\*0.75=12

TREEIFY\_THRESHOLD=8;Bucket中链表长度大于该默认值，转化为红黑树

MIN\_TREEIFY\_CAPACITY：桶中的Node被树化时的最小的hash表容量：64

四、 LinkedHashMap的底层实现原理

源码中给Node加入两个指针，分别指向前一个和后一个

五、 Map中常用的方法：

```

Map map= new HashMap();
//添加、删除、修改操作
// put():添加元素
map.put(44,33);
map.put(22,99);
System.out.println(map);
//putAll(Map map):将参数map全部添加到当前map
Map map1= new HashMap();
map1.put(1,2);
map1.put(5,52);
map.putAll(map1);
//remove(Object key):根据key移除元素，并返回value，没有返回null
map.remove( key: 44);
//clear():清空数据，与map=null不同
map.clear();

//元素查询操作
//Object get(Object key):获取指定key对应的value
//boolean containsKey(Object key):是否包含指定的key
//boolean containsValue(Object value):是否包含指定的value
//int size():返回当前map中key-value对的个数
//boolean isEmpty():判断当前map是否为空
//boolean equals(Object obj):判断当前map和参数对象obj是否相等

//元视图操作的方法
//Set keySet():返回所有key构成的Set集合，来遍历key
Set set=map.keySet();
Iterator iterator=set.iterator();
//Collection values:返回所有values构成的Collection集合
Collection values=map.values();
//Set entrySet():返回所有key-value对构成的Set集合
Set entrySet=map.entrySet();
Iterator iterator1=entrySet.iterator();
while(iterator1.hasNext()){
    Object ob1=iterator1.next();
    //entrySet集合中的元素都是entry
    Map.Entry entry=(Map.Entry)ob1;
    System.out.println(entry.getKey()+"---->"+entry.getValue())
}

```

## 总结：常用方法

增： put()

删： remove()

改： put()

查： get()

长度： size()

遍历： keySet()/values()/entrySet()

## 六、 TreeMap

1. 向TreeMap中添加key-value，要求key必须是由同一个类创建的对象
2. 要按照key进行排序：自然排序、定制排序

## 七、Properties

1. 常用来处理配置文件。key和value都是String类型

```
FileInputStream fis=null;
try {
    Properties pros=new Properties();
    fis =new FileInputStream( name: "jdbc.properties");//默认使用工程下文件
    pros.load(fis);//加载流对应的文件
    String name = pros.getProperty("name");
    String password = pros.getProperty("password");
    System.out.println(name+"="+password);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        fis.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

# 79.Collections工具类

2021年8月17日 18:39

Collections:操作Collection、Map的工具类

- 1.reverse(List):反转List中元素的顺序
- 2.shuffle(List):对List集合进行随机排序
- 3.sort(List):根据元素的自然顺序对指定的List集合元素按升序排序
- 4.sort(List,Comparator):根据指定的Compartor产生的顺序对List集合进行排序
- 5.swap(List,int i,int j):将指定List集合的i处和j处元素交换位置

6.Object max(Collection):根据元素的自然顺序，返回指定集合的最大元素

7.Object max(Collection,Comparator)

8.Object min(Collection)

9.Object min(Collection,comparator)

10.int frequency(Collection, Object):返回指定元素在集合中出现次数

11.void copy(List dest,List src):将src中的数内容复制到dest中，List dest=Arrays.asList(new Object[List.size()])  
这样创建可避免长度异常

12.boolean replaceAll(List list, Object oldVal, Object newVal)

Collections类提供了多个synchronizedXxx()方法，该方法可将指定的集合包装成线程同步的集合从而解决多线程并发访问集合时线程安全问题

List list1=synchronizedList(list);

# 80.泛型 (Generic)

2021年8月18日 10:22

泛型的使用

1.jdk 5.0新增的特性

2.//在集合中使用泛型之前的情况：

问题一：类型不安全,加任何类型都可

问题二：强转时可能出现ClassCastException

//在集合中使用泛型的情况

ArrayList例

不能用基本数据类型，在编译时就会进行类型检查，保证数据安全

```
ArrayList<Integer> list=new ArrayList<Integer>();
list.add(12);
```

避免了强转操作

```
int data=list.get(0);
```

Map例

```
Map<String,Integer> map=new HashMap<String,Integer>();
map.put("tom",22)
```

泛型嵌套

```
Set<Map.Entry<String,Integer>> entry=map.entrySet(); //Map.Entry, 因为Entry是Map的内部接口, 省略map
需要导入java.util.Map.*
```

3.总结：

- ①集合或集合类在jdk5.0时都修改为带泛型的结构
- ②在实例化集合类时，可以指明具体的泛型类型
- ③指明完之后，在集合类或接口中，内部结构使用到类的泛型的位置，都指定为实例化时的泛型类型  
比如：add(E e)---->实例化后：add(Integer e)
- ④泛型类型必须是类，基本数据类型需要用包装类
- ⑤如果实例化时，没有指明泛型的类型，则默认使用Object

4.自定义泛型结构：泛型类、泛型接口；泛型方法

-----关于自定义泛型类，泛型接口：

- ①如果定义了泛型类，实例化没有指明类的泛型，则认为此泛型类型为Object类型
- ②要求：如果定义了泛型类，建议在实例化时指明类的泛型
- ③子类继承带泛型的父类时，指明了泛型类型，则子类实例化时不用再带泛型
- ④子类继承带泛型父类，如果未指明泛型类型，则子类仍然是泛型类
- ⑤泛型不同的引用不能相互赋值
- ⑥静态方法不能使用类的泛型
- ⑦异常类不能声明为泛型类

⑧造泛型数组T[] arr=(T[])new Object[10];

```
class Father<T1, T2> {  
}  
// 子类不保留父类的泛型  
// 1)没有类型 擦除  
class Son<A, B> extends Father{//等价于class Son extends Father<Object,Object>{  
}  
  
// 2)具体类型  
class Son2<A, B> extends Father<Integer, String> {  
}  
// 子类保留父类的泛型  
// 1)全部保留  
class Son3<T1, T2, A, B> extends Father<T1, T2> {  
}  
// 2)部分保留  
class Son4<T2, A, B> extends Father<Integer, T2> {  
}
```

-----泛型方法：在方法中出现了泛型的结构，泛型参数与类的泛型参数无关，泛型方法所在类是不是泛型类都可以

①泛型方法在调用时指明类型（与实参一样），所以可以是static 的

```
public <E> List<E> copyFromArrayList(E[] arr){  
  
    ArrayList<E> list = new ArrayList<>();  
  
    for(E e : arr){  
        list.add(e);  
    }  
    return list;
```

## 5. 泛型在继承方面的体现

类A是类B的父类，G<A>和G<B>二者不具有父子类关系，二者是并列关系，不能相互赋值。

补充：类A是类B的父类，A<G>是B<G>的父类--List<G>list=new ArrayList<G>();

## 6. 通配符的使用

通配符：？

类A是类B的父类，G<A>和G<B>二者不具有父子类关系，二者共同的父类是G<?>

添加：对于List<?>就不能向其内部添加数据，null除外

读取：允许读取数据，读取数据的类型为Object

```
@Test  
public void test3(){  
    List<Object> list1 = null;  
    List<String> list2 = null;  
  
    List<?> list = null;  
  
    list = list1;  
    list = list2;  
  
    print(list1);  
    print(list2);  
}  
  
public void print(List<?> list){  
    Iterator<?> iterator = list.iterator();  
    while(iterator.hasNext()){  
        Object obj = iterator.next();  
        System.out.println(obj);  
    }  
}
```

## 7.有限制条件的通配符

? extends A:可以看做是G<A>和G<B>的父类，其中B是A的子类 (<=A)

读取数据：类型为A或者Object

写入数据：不可

? super A:可以看做是G<A>和G<B>的父类，其中B是A的父类 (>=)

读取数据：类型为Object

写入数据：<=A都可以

# 81.File类

2021年8月18日 17:13

## 一、File类的使用

1.File类的一个对象，代表一个文件或一个文件目录（文件夹）

2.File类声明在java.io包下

3.File实例的创建

File(String filePath)

File(String parentPath, String childPath)

File(File parentFile, String childPath)

//1.相对路径：相较于某个路径下，指明的路径

//2.绝对路径：包含盘符在内的文件或文件目录的路径

//3.路径分隔符windows:\ unix:/

//构造器1

```
File file=new File( pathname: "hello.txt");//相对于当前
```

```
File file1=new File( pathname: "D:\\java\\ideacode\\javaSenior\\day07\\hello.txt");
```

//构造器2

```
File file2=new File( parent: "D:\\java\\ideacode", child: "javaSenior");
```

//构造器3

```
File file3=new File(file2, child: "helllo.txt");
```

4.File类中涉及到关于文件或者文件目录的创建、删除、重命名、修改时间、文件大小等方法，并未涉及写入或读取文件内容的操作。如果需要，必须使用IO流来完成

5.后续File类的对象常会作为参数传递到流的构造器中，指明读取或写入的终点

## 二、File类方法

### 读取

1.public String getAbsolutePath():获取绝对路径

2.public String getPath(): 获取路径

3.public String getName():获取名称

4.public String getParent():获取上层文件目录路径。没有返回null

5.public long length():获取文件长度（即：字节数）。不能获取目录长度

6.public long lastModified():获取最后一次的修改时间，毫秒值

7.public String[] list():获取指定目录下的所有文件或者文件目录的名称数组

8.public File[] listFiles():获取指定目录下的所有文件或者文件目录的File数组

9.public boolean renameTo(File dest):把文件重命名为指定的文件路径

例：file1.renameTo(file2)要想保证返回true,需要保证file1在硬盘中是存在的，且file2在硬盘中不存在。

### 判断

10.public boolean isDirectory():判断是否是文件目录\*

11. public boolean isFile(): 判断是否是文件\*
12. public boolean exists(): 判断是否存在\*
13. public boolean canRead(): 判断是否可读
14. public boolean canWrite(): 判断是否可写
15. public boolean isHidden(): 判断是否隐藏

创建硬盘中对应的文件或文件目录

public boolean createNewFile(): 创建文件。若文件存在，则不创建，返回false

public boolean mkdir(): 创建文件目录，如果此文件目录存在，则不创建，如果此文件的上层目录  
不存在则也不创建

public boolean mkdirs(): 创建文件目录。如果上层文件目录不存在，一并创建

删除硬盘中对应的文件或文件目录

public boolean delete(): 删除文件或者文件夹

要想删除成功，该目录下不能有文件或者文件目录

删除注意事项：java中的删除不走回收站。

# 82.IO流1

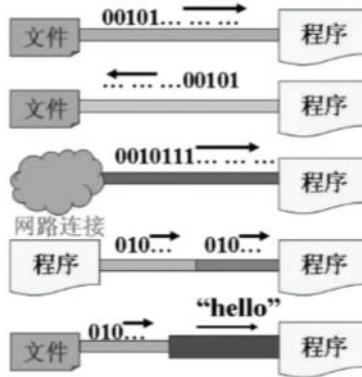
2021年8月19日 10:03

## 一、IO流原理及流的分类

### Java IO原理

●输入：读取外部数据（磁盘、光盘等存储设备的数据）到程序（内存）中。

●输出：将程序（内存）数据输出到磁盘、光盘等存储设备中。



### 流的分类

- 按操作**数据单位**不同分为：字节流(8 bit), 字符流(16 bit)
- 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

{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，都是从如下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。

# IO 流体系

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	InputStream	OutputStream	Reader	Writer
访问文件	FileInputStream	FileOutputStream	FileReader	FileWriter
访问数组	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
访问管道	PipedInputStream	PipedOutputStream	PipedReader	Pipewriter
访问字符串			StringReader	StringWriter
缓冲流	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
转换流			InputStreamReader	OutputStreamWriter
对象流	ObjectInputStream	ObjectOutputStream		
	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
打印流		PrintStream		PrintWriter
推回输入流	PushbackInputStream		PushbackReader	
特殊流	DataInputStream	DataOutputStream		

## 1. 流的分类

## 2. 流的体系结构

抽象基类	节点流 (或文件流)	缓冲流 (处理流的一种)
InputStream	FileInputStream(read(byte[] buffer))	BufferedInputStream
OutputStream	FileOutputStream(write(byte[] buffer,0,len))	BufferedOutputStream
Reader	FileReader(read(char[] cbuf))	BufferedReader(readLine())
Writer	FileWriter(write(char[] cbuf,0,len))	BufferedWriter(flush())

//对于文本文件(.txt,.java,.c,.cpp)使用字符流处理

//对于非文本文件(图片, 视频, .doc, .ppt...)使用字节流处理

## 二、使用字符流

//不能用来处理字节数据, 只能文本数据

### 1. 读取FileReader

```
//*****将文件内容读入内存并输出
//1.read()的理解:返回读入的一个字符。如果到达文件末尾, 返回-1
//2.异常的处理: 为了保证流资源一定可以执行关闭操作。需要使用try-catch-finally处理
//3.读入的文件一定要存在, 否则报FileNotFoundException。
//①实例化File类的对象, 指明操作的文件
File file=new File( pathname: "hello.txt");
//②提供具体的流
FileReader fr=null;
try {
    fr = new FileReader(file);
    //③数据的读入
    //read():返回读入的一个字符。如果到达文件末尾, 返回-1
    int data;
    while((data=fr.read())!=-1){
        System.out.print((char)data);
    }
} catch (IOException e) {
    e.printStackTrace();
}finally {
    //④流的关闭操作
    try {
```

```

} finally {
    //④流的关闭操作
    try {
        if(fr!=null)
            fr.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```

//对read()操作升级：使用read的重载方法
//read(char[] cbuf):返回每次读入cbuf数组中的字符的个数。如果到文件末尾，返回-1
char[] cbuf=new char[5];
int len;
while((len=fr.read(cbuf))!=-1){
    //方式一
    for(int i=0;i<len;++i){
        System.out.println(cbuf[i]);
    }
    //方式二
    String str=new String(cbuf, offset: 0, len);
    System.out.print(str);
}

```

## 2.写入FileWriter

```

//文件写入操作
//1.输出操作对应的File可以不存在，不会报异常
//2.文件在硬盘中不存在，自动创建
//    对应文件存在：流构造器使用：FileWriter(file)/FileWriter(file, false)会覆盖原有文件
//    流构造器使用：FileWriter(file, true)不会覆盖原有文件，而是添加
//①通过File对象，指明要操作的文件
File file=new File( pathname: "hello.txt");
//②提供FileWriter对象，用于数据的写出
FileWriter fw=null;
try {
    fw=new FileWriter(file, append: true);
    //③写出的操作
    fw.write( str: "abhcs");
    fw.write( str: "\nanjbhjcdvca");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    //④流资源的关闭
    try {
        if(fw!=null)
            fw.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

## 3.复制

```
//文件复制操作
//①创建File对象，指明读入和写出的文件
File file=new File( pathname: "hello.txt");
File file1=new File( pathname: "hello1.txt");
//②提供输入输出流对象
FileWriter fw1=null;
FileReader fw=null;
try {
    //③数据输入与写出操作
    fw=new FileReader(file);
    fw1=new FileWriter(file1, append: true);
    char[] sbuf=new char[5];
    int len;
    while ((len=fw.read(sbuf))!=-1){
        fw1.write(sbuf, off: 0, len);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    //④流资源的关闭
    try {
        if(fw!=null)
            fw.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(fw1!=null)
            fw1.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

### 三、使用字节流

1. 读取FileInputStream
2. 写入FileOutputStream
3. 复制

```

//测试字节流, FileOutputStream , FileInputStream使用
//①创建File对象, 指明读入和写出的文件
File file=new File( pathname: "head.jpeg");
File file1=new File( pathname: "head1.jpeg");
//②提供输入输出流对象
FileInputStream fw=null;
FileOutputStream fw1=null;
try {
    //③读取写入进行复制
    fw=new FileInputStream(file);
    fw1=new FileOutputStream(file1, append: true);
    byte[] sbuf=new byte[100];
    int len;
    while ((len=fw.read(sbuf))!=-1){
        fw1.write(sbuf, off: 0, len);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    //④流资源的关闭
    try {
        if(fw!=null)
            fw.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(fw1!=null)
            fw1.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

## 四、缓冲流

BufferedInputStream

BufferedOutputStream

BufferedReader

BufferedWriter

```

//1.文件对象
File file1=new File( pathname: "head.jpeg");
File file2=new File( pathname: "head2.jpeg");
BufferedInputStream bis1=null;
BufferedOutputStream bis2=null;
try {

    //2.造流
    //2.1造节点流
    FileInputStream fis1=new FileInputStream(file1);
    FileOutputStream fis2=new FileOutputStream(file2);
    //2.2造缓冲流
    bis1=new BufferedInputStream(fis1);
    bis2=new BufferedOutputStream(fis2);
}

```

```
//1.文件对象
File file1=new File( pathname: "head.jpeg");
File file2=new File( pathname: "head2.jpeg");
BufferedInputStream bis1=null;
BufferedOutputStream bis2=null;
try {

    //2.造流
    //2.1造节点流
    FileInputStream fis1=new FileInputStream(file1);
    FileOutputStream fis2=new FileOutputStream(file2);
    //2.2造缓冲流
    bis1=new BufferedInputStream(fis1);
    bis2=new BufferedOutputStream(fis2);
    //3.复制的细节：读取、写入
    byte[] buffer=new byte[1024];
    int len;
    while((len= bis1.read(buffer))!=-1){
        bis2.write(buffer, off: 0, len);
        // bis2.flush(); //刷新缓冲区，已经自动完成了
    }

} catch (IOException e) {
    e.printStackTrace();
} finally {
    //4.资源的关闭
    //要求：先关闭外层的流，再关闭内层的流。
    //说明：在关闭外层流的同时内层也会自动关闭，所以可以使用
    try {
        if(bis1!=null)
            bis1.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(bis2!=null)
            bis2.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- 1.作用：提高流的读取、写入效率
- 2.原因：内部提供了一个缓冲区
- 3.处理流就是包在已有流之上
- 4.使用匿名对象简化1.2步，并添加字符流特有方法readLine()

```
BufferedReader bis1=null;
BufferedWriter bis2=null;
try {
    bis1=new BufferedReader(new FileReader(new File( pathname: "hello1.txt")));
    bis2=new BufferedWriter(new FileWriter(new File( pathname: "hello2.txt")));
    //复制的细节: 读取、写入
    //方式一
    char[] buffer=new char[5];
    int len;
    while((len=bis1.read())!=-1){
        bis2.write(buffer, off: 0,len);
    }
    //方式二: 使用String
    String data;
    while((data=bis1.readLine())!=null){
        //方法1
        bis2.write( str: data+"\n"); //data中不包含换行符
        //方法2
        bis2.write(data);
        bis2.newLine(); //提供换行操作
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    //资源关闭
    try {
        if(bis1!=null)
            bis1.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(bis2!=null)
            bis2.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## 五、转换流

### 1. 属于字符流

InputStreamReader: 将一个字节的输入流转换为字符的输入流

OutputStreamWriter: 将一个字符的输出流转换为字节的输出流

2. 作用: 提供字节流和字符流之间的转换

3. 解码: 字节、字节数组-->字符、字符数组

编码: 字符、字符数组-->字节、字节数组

解码:

```

InputStreamReader ir=null;
try {
    FileInputStream fin=new FileInputStream( name: "hello.txt");
    //InputStreamReader ir=new InputStreamReader(fin); //使用系统默认字符集
    //参数二指明了字符集, 具体使用哪个字符集, 取决于文件保存时使用的字符集
    ir=new InputStreamReader(fin, charsetName: "UTF-8");
    char[] buffer=new char[5];
    int len;
    while ((len=ir.read(buffer))!=-1){
        String str=new String(buffer, offset: 0, len);
        System.out.print(str);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if(ir!=null){
            ir.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

## 解码与编码结合

```

//解码、编码 (gbk)
InputStreamReader ir=null;
OutputStreamWriter irs=null;
try {
    FileInputStream fin=new FileInputStream( name: "hello.txt");
    FileOutputStream fins=new FileOutputStream( name: "hello-gbk.txt");
    //InputStreamReader ir=new InputStreamReader(fin); //使用系统默认字符集
    //参数二指明了字符集, 具体使用哪个字符集, 取决于文件保存时使用的字符集
    ir=new InputStreamReader(fin, charsetName: "UTF-8");
    irs=new OutputStreamWriter(fins, charsetName: "gbk");
    char[] buffer=new char[5];
    int len;
    while ((len=ir.read(buffer))!=-1){
        irs.write(buffer, off: 0, len);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try{
        if(ir!=null){
            ir.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(irs!=null)
            irs.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

## 4.字符集

### 补充：字符编码

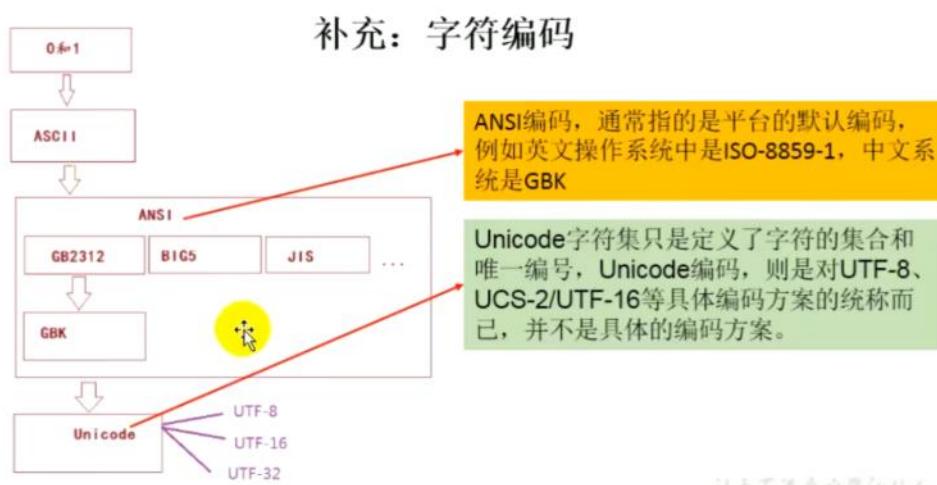
#### ● 编码表的由来

计算机只能识别二进制数据，早期由来是电信号。为了方便应用计算机，让它可以识别各个国家的文字。就将各个国家的文字用数字来表示，并一一对应，形成一张表。这就是编码表。

#### ● 常见的编码表

- **ASCII**: 美国标准信息交换码。
  - ✓ 用一个字节的7位可以表示。
- **ISO8859-1**: 拉丁码表。欧洲码表
  - ✓ 用一个字节的8位表示。
- **GB2312**: 中国的中文编码表。最多两个字节编码所有字符
- **GBK**: 中国的中文编码表升级，融合了更多的中文文字符号。最多两个字节编码
- **Unicode**: 国际标准码，融合了目前人类使用的所有字符。为每个字符分配唯一的字符码。所有的文字都用两个字节来表示。
- **UTF-8**: 变长的编码方式，可用1-4个字节来表示一个字符。

### 补充：字符编码



# 83.IO流2

2021年8月19日 19:26

## 六、其他流

### 1.标准的输入、输出流

①

System.in:标准的输入流， 默认从键盘输入

System.out: 标准的输出流， 默认从控制台输出

②

System类的setIn(InputStream is),setOut(PrintStream ps)方式重新指定输入和输出的流

```
//读取用户的输入
//方式一：使用Scanner实现，调用next()返回一个字符串
//方式二：使用System.in实现。System.in--->转换流--->BufferedReader的readLine()
BufferedReader br=null;
try {
    InputStreamReader isr=new InputStreamReader(System.in);
    br=new BufferedReader(isr);
    System.out.println("请输入字符串：");
    br.readLine();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if(br!=null)
            br.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

### 2.打印流

PrintStream

PrintWriter

①提供了一系列重载的print()和println()方法

```
PrintStream ps = null;
try {
    FileOutputStream fos = new FileOutputStream(new File( pathname: "D:\\IO\\text.txt"));
    // 创建打印输出流，设置为自动刷新模式(写入换行符或字节 '\n' 时都会刷新输出缓冲区)
    ps = new PrintStream(fos, autoFlush: true);
    if (ps != null) { // 把标准输出流(控制台输出)改成文件
        System.setOut(ps);
    }

    for (int i = 0; i <= 255; i++) { // 输出ASCII字符
        System.out.print((char) i);
        if (i % 50 == 0) { // 每50个数据一行
            System.out.println(); // 换行
        }
    }
}
```

### 3.数据流

DataInputStream和DataOutputStream

①作用：用于读取或写出基本数据类型的变量或字符串

②读取不同类型的数据的顺序要与当初写入文件时保存的数据相同

- 为了方便地操作Java语言的基本数据类型和String的数据，可以使用数据流。
- 数据流有两个类：(用于读取和写出基本数据类型、String类的数据)
  - **DataInputStream** 和 **DataOutputStream**
  - 分别“套接”在 **InputStream** 和 **OutputStream** 子类的流上

- **DataInputStream**中的方法

boolean readBoolean()	byte readByte()
char readChar()	float readFloat()
double readDouble()	short readShort()
long readLong()	int readInt()
String readUTF()	void readFully(byte[] b)

- **DataOutputStream**中的方法

➢ 将上述的方法的**read**改为相应的**write**即可。

让天下没有难学的IT

# 84.IO流3

2021年8月20日 10:22

## 对象的序列化

- 对象序列化机制允许把内存中的Java对象转换成平台无关的二进制流，从而允许把这种二进制流持久地保存在磁盘上，或通过网络将这种二进制流传输到另一个网络节点。当其它程序获取了这种二进制流，就可以恢复成原来的Java对象
- 序列化的好处在于可将任何实现了Serializable接口的对象转化为字节数据，使其在保存和传输时可被还原
- 序列化是RMI（Remote Method Invoke – 远程方法调用）过程的参数和返回值都必须实现的机制，而RMI是JavaEE的基础。因此序列化机制是JavaEE平台的基础
- 如果需要让某个对象支持序列化机制，则必须让对象所属的类及其属性是可序列化的，为了让某个类是可序列化的，该类必须实现如下两个接口之一。否则，会抛出NotSerializableException异常

> Serializable  
> Externalizable

让天下没有难学的技术

### 一、对象流的使用

1.ObjectInputStream和ObjectOutputStream

2.作用：用于存储和读取基本数据类型数据或对象的处理流。

可以把Java中的对象写入到数据源中，也能把对象从数据源中还原回来

#### 3.序列化过程

```
//序列化过程：将内存中的Java对象保存到磁盘中或通过网络传输出去
//使用ObjectOutputStream实现
ObjectOutputStream oos= null;
try {
    oos = new ObjectOutputStream(new FileOutputStream( name: "object.dat"));
    oos.writeObject(new String( original: "北京欢迎你！")); //写入
    oos.flush(); //刷新操作
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if(oos!=null)
            oos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

#### 4.反序列化过程

```
//反序列化：将磁盘文件中的对象还原为内存中的一个java对象
//使用ObjectInputStream来实现
ObjectInputStream ois= null;
try {
    ois = new ObjectInputStream(new FileInputStream( name: "object.dat"));
    Object obj=ois.readObject();
    String str=(String)obj;
    System.out.println(str);
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

```

//反序列化：将磁盘文件中的对象还原为内存中的一个java对象
//使用ObjectInputStream来实现
ObjectInputStream ois= null;
try {
    ois = new ObjectInputStream(new FileInputStream( name: "object.dat"));
    Object obj=ois.readObject();
    String str=(String)obj;
    System.out.println(str);
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} finally {
    try {
        if(ois!=null)
        ois.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

5.要想序列化一个自定义对象需要满足如下的要求，才能序列化

- ①需要实现接口：Serializable
- ②当前类提供一个任意常量private static final long serialVersionUID=111111111L

## 对象的序列化

● 凡是实现Serializable接口的类都有一个表示序列化版本标识符的静态变量：

- **private static final long serialVersionUID;**
- serialVersionUID用来表明类的不同版本间的兼容性。简言之，其目的是以序列化对象进行版本控制，有关各版本反序列化时是否兼容。
- 如果类没有显示定义这个静态变量，它的值是Java运行时环境根据类的内部细节自动生成的。若类的实例变量做了修改，serialVersionUID 可能发生变化。故建议，显式声明。

● 简单来说，Java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地相应实体类的serialVersionUID进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常。**(InvalidCastException)**

③还必须保证其内部的属性是可序列化的

如内部类也需要实现接口

④补充：static和transient修饰的成员变量不能被序列化，某属性不想被序列化可以加

## 二、随机存取文件流（RandomAccessFile类）

RandomAccessFile的使用

1.RandomAccessFile直接继承于java.lang.Object类，实现了DataInput和DataOutput接口

## 2.RandomAccessFile既可以作为一个输入流，又可以作为一个输出流

```
RandomAccessFile raf1 = new RandomAccessFile(new File( pathname: "爱情与友情.jpg"), mode: "r");
RandomAccessFile raf2 = new RandomAccessFile(new File( pathname: "爱情与友情1.jpg"), mode: "rw");

byte[] buffer = new byte[1024];
int len;
while((len = raf1.read(buffer)) != -1){
    raf2.write(buffer, off: 0, len);
}

raf1.close();
raf2.close();
```

3.如果RandomAccessFile作为输出流时，写出文件不存在，则执行时自动创建  
如果存在则对原有文件内容进行覆盖。（默认从头开始覆盖）

4.可以通过相关的操作，实现RandomAccessFile的插入操作（seek()）

## RandomAccessFile 类

- RandomAccessFile 声明在java.io包下，但直接继承于java.lang.Object类。并且它实现了DataInput、DataOutput这两个接口，也就意味着这个类既可以读也可以写。
- RandomAccessFile 类支持“随机访问”的方式，程序可以直接跳到文件的任意地方来读、写文件
  - 支持只访问文件的部分内容
  - 可以向已存在的文件后追加内容
- RandomAccessFile 对象包含一个记录指针，用以标示当前读写处的位置。  
RandomAccessFile 类对象可以自由移动记录指针：
  - long getFilePointer(): 获取文件记录指针的当前位置
  - void seek(**long pos**): 将文件记录指针定位到 pos 位置

## RandomAccessFile 类

- 构造器
  - public RandomAccessFile(File file, String mode)
  - public RandomAccessFile(String name, String mode)
- 创建 RandomAccessFile 类实例需要指定一个 mode 参数，该参数指定 RandomAccessFile 的访问模式：
  - r: 以只读方式打开
  - rw: 打开以便读取和写入
  - rwd: 打开以便读取和写入；同步文件内容的更新
  - rws: 打开以便读取和写入；同步文件内容和元数据的更新
- 如果模式为只读r，则不会创建文件，而是会去读取一个已经存在的文件，如果读取的文件不存在则会出现异常。如果模式为rw读写。如果文件不存在则会去创建文件，如果存在则不会创建。

### 三、NIO概述

## Java NIO 概述

- Java NIO (New IO, Non-Blocking IO)是从Java 1.4版本开始引入的一套新的IO API，可以替代标准的Java IO API。NIO与原来的IO有同样的作用和目的，但是使用的方式完全不同，NIO支持面向缓冲区的( IO是面向流的)、基于通道的IO操作。NIO将以更加高效的方式进行文件的读写操作。
- Java API中提供了两套NIO，一套是针对标准输入输出NIO，另一套就是网络编程NIO。
  - |----java.nio.channels.Channel
    - ✓ |----FileChannel: 处理本地文件
    - ✓ |----SocketChannel: TCP网络编程的客户端的Channel
    - ✓ |----ServerSocketChannel: TCP网络编程的服务器端的Channel
    - ✓ |----DatagramChannel: UDP网络编程中发送端和接收端的Channel

## Path、Paths和Files核心API

- 早期的Java只提供了一个File类来访问文件系统，但File类的功能比较有限，所提供的方法性能也不高。而且，大多数方法在出错时仅返回失败，并不会提供异常信息。
- NIO.2为了弥补这种不足，引入了Path接口，代表一个平台无关的平台路径，描述了目录结构中文件的位置。Path可以看成是File类的升级版本，实际引用的资源也可以不存在。
- 在以前IO操作都是这样写的：

```
import java.io.File;
File file = new File("index.html");
```
- 但在Java7中，我们可以这样写：

```
import java.nio.file.Path;
import java.nio.file.Paths;
Path path = Paths.get("index.html");
```

### 四、第三方jar包实现数据读写

新建directory--命名libs---粘贴jar包---Add as library

# 85. 网络编程

2021年8月20日 18:46

## 一、网络编程概述

### 1. 网络通信中的两个主要问题

--如何准确的定位网络上的一台或多台主机；定位主机上特定的应用

--找到主机后如何可靠高效的进行数据传输

### 2. 网络编程的两个要素：

--IP和端口号

--提供网络通信协议：TCP/IP参考模型（应用层、传输层、网络层、物理+数据链路层）

## 二、IP和端口号

### 1. IP地址（InetAddress）

①唯一的标识互联网上的计算机（通信实体）

②在JAVA中使用InetAddress类代表IP

③IP分类：IPv4和IPv6；万维网和局域网；

④域名：www.baidu.com

⑤本地回路地址：127.0.0.1 对应着：localhost

⑥实例化InetAddress：

两个方法：getByName(String host)、getLocalHost()

两个常用方法：getHostName()/getHostAddress()

```
InetAddress inte1=InetAddress.getByName("192.168.10.14");
System.out.println(inte1);
InetAddress inte2=InetAddress.getByName("www.sina.com");
System.out.println(inte2);
//获取本地IP
InetAddress inte3=InetAddress.getByName("127.0.0.1");
InetAddress inte4=InetAddress.getLocalHost();
```

### 2. 端口号

①正在计算机上运行的进程

②范围：被规定为一个16位整数0-65535

### 3. 端口号与IP地址的组合的出一个网络套接字：Socket

## 三、网络通信协议

### 1. TCP网络编程

例：客户端发内容给服务端，服务端输出控制台

```
@Test
public void client(){
    //1. 创建Socket对象，指明服务器端的IP和端口号
```

```
public void client(){
    //1.创建Socket对象，指明服务器端的IP和端口号
    Socket socket= null;
    OutputStream os= null;
    try {
        InetAddress inet=InetAddress.getByName("127.0.0.1");
        socket = new Socket(inet, port: 8899);
        //2.获取一个输出流，用于输出数据
        os = socket.getOutputStream();
        //3.写出数据的操作
        os.write("你好，我是客户端".getBytes());
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        //4.资源的关闭
        try {
            if(os!=null)
                os.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            if(socket!=null)
                socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
@Test
public void server(){

    ServerSocket ss= null;
    Socket socket= null;
    InputStream is= null;
    ByteArrayOutputStream baos= null;
    try {
        //1.创建服务器端的ServerSocket，指明自己的端口号
        ss = new ServerSocket( port: 8899);
        //2.调用accept()接受来自于客户端的socket
        socket = ss.accept();
        //3.获取输入流
        is = socket.getInputStream();
        //4.读取输入流中的数据
        //使用下面的类装全部数据，防止中文乱码
        baos = new ByteArrayOutputStream();
        byte[] buffer=new byte[5];
    }
}
```

```
//使用下面的类实现部分数据，防止中文乱码
baos = new ByteArrayOutputStream();
byte[] buffer=new byte[5];
int len;
while((len=is.read(buffer))!=-1){
    baos.write(buffer, 0, len);
}
System.out.println(baos.toString());
System.out.println(socket.getInetAddress().getHostAddress());
} catch (IOException e) {
    e.printStackTrace();
} finally {
    //5.关闭资源
    try {
        if(baos!=null)
            baos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(is!=null)
            is.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(socket!=null)
            socket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(ss!=null)
            ss.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

例：发送图片加服务器反馈

```
@Test
public void client1() {
    Socket socket = null;
    OutputStream os = null;
    FileInputStream file=null;
    InputStream ins1=null;
    try {
        //创建socket()并设置接受方ip与端口号
        socket = new Socket(InetAddress.getByName("127.0.0.1"), port 8888);
        os = socket.getOutputStream();
        file = new FileInputStream("D:\\test\\file1.txt");
        ins1 = new FileInputStream("D:\\test\\file2.txt");
        byte[] b = new byte[1024];
        int len;
        while ((len = ins1.read(b)) != -1) {
            os.write(b, 0, len);
        }
        os.flush();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (socket != null) {
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (os != null) {
            try {
                os.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (file != null) {
            try {
                file.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (ins1 != null) {
            try {
                ins1.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
InputStream ins1=null;
try {
    //创建socket()并设置接受方ip与端口号
    socket = new Socket(InetAddress.getByName("127.0.0.1"), port: 9090);
    //获取输出流
    os = socket.getOutputStream();
    //要发送的图片
    file = new FileInputStream( name: "head.jpeg");
    //读取并发送
    byte[] buffer = new byte[1024];
    int len;
    while ((len = file.read(buffer)) != -1) {
        os.write(buffer, off: 0, len);
    }
    //传递发送完毕信号，避免服务器端一直读取并等待，read()阻塞
    socket.shutdownOutput();
    //接受客户端反馈
    //ByteArrayOutputStream内置可变数组，可避免乱码
    ByteArrayOutputStream baos=new ByteArrayOutputStream();
    ins1=socket.getInputStream();
    byte[] buffers=new byte[10];
    int len1;
    while((len1=ins1.read(buffers))!=-1){
        baos.write(buffers, off: 0, len1);
    }
    System.out.println(baos.toString());
}

} catch (IOException e) {
    e.printStackTrace();
} finally {
    //关闭资源
    try {
        if(file!=null)
            file.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(os!=null)
            os.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(socket!=null)
            socket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(ins1!=null)
            ins1.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

@Test
public void servers(){

    ServerSocket ss= null;
    Socket socket= null;
    InputStream is= null;
    FileOutputStream fileo=null;
    OutputStream os1=null;
    try {
        ss = new ServerSocket( port: 9090);
        socket = ss.accept();
        is = socket.getInputStream();
        fileo=new FileOutputStream( name: "headll.jpeg", append: true);
    }
}
```

```
socket = ss.accept();
is = socket.getInputStream();
fileo=new FileOutputStream( name: "headll.jpeg", append: true);

byte[] buffer=new byte[1024];
int len;
while((len=is.read(buffer))!=-1){
    fileo.write(buffer, off: 0, len);
}
//反馈给客户端
os1=socket.getOutputStream();
os1.write("服务器已经收到".getBytes(StandardCharsets.UTF_8));
} catch (IOException e) {
    e.printStackTrace();
} finally {

    try {
        if(fileo!=null)
            fileo.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(is!=null)
            is.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(socket!=null)
            socket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(ss!=null)
            ss.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(os1!=null)
            os1.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
```

客户端：自定义/浏览器

服务端：自定义/Tomcat服务器

## 2.UDP网络编程

```

//发送端
@Test
public void sender(){
    DatagramSocket socket= null;
    try {
        socket = new DatagramSocket();
        String str="我是UDP方式发送的";
        byte[] data=str.getBytes(StandardCharsets.UTF_8);
        InetAddress inet=InetAddress.getLocalHost();
        DatagramPacket packet=new DatagramPacket(data, offset: 0,data.length,inet, port: 8888);
        socket.send(packet);
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        socket.close();
    }
}

//接收端
@Test
public void receive(){
    DatagramSocket socket= null;
    try {
        socket = new DatagramSocket( port: 8888);
        byte[] buffer=new byte[100];
        DatagramPacket packet=new DatagramPacket(buffer, offset: 0,buffer.length);
        socket.receive(packet);
        System.out.println(new String(packet.getData(), offset: 0, packet.getLength()));
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        socket.close();
    }
}

```

### 3.URL网络编程

①URL:统一资源定位符，对应互联网上某个资源

URL的基本结构由5部分组成：

②<传输协议>://<主机名>:<端口号>/<文件名>#片段名?参数列表

```

URL url = new URL( spec: "http://localhost:8080/examples/beauty.jl"
//          public String getProtocol( )      获取该URL的协议名
//          System.out.println(url.getProtocol());
//          public String getHost( )         获取该URL的主机名
//          System.out.println(url.getHost());
//          public String getPort( )         获取该URL的端口号
//          System.out.println(url.getPort());
//          public String getPath( )        获取该URL的文件路径
//          System.out.println(url.getPath());
//          public String getFile( )        获取该URL的文件名
//          System.out.println(url.getFile());
//          public String getQuery( )       获取该URL的查询名
//          System.out.println(url.getQuery());

```

```
InputStream ins=null;
FileOutputStream fos=null;
HttpURLConnection urlConnection=null;
try {
    URL url=new URL( spec: "https://www.jd.com/?source=enterprise&cu=true&utm_source=browser.lenovo.com.cn&utm_
//连接
urlConnection=(HttpURLConnection)url.openConnection();
urlConnection.connect();
ins = urlConnection.getInputStream();
fos=new FileOutputStream( name:"URLTest.html");
byte[] buffer=new byte[1024];
int len;
while((len=ins.read(buffer))!=-1){
    fos.write(buffer, off: 0 ,len);
}
} catch (IOException e) {
    e.printStackTrace();
}finally{
    try {
        if(ins!=null)
            ins.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if(fos!=null)
            fos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
//断开连接
if(urlConnection!=null)
urlConnection.disconnect();
}
```

TCP：可靠的数据传输（三次握手、四次挥手）进行大数据量的传输；效率低

UDP：不可靠；以数据报形式发送，数据报限定64k；效率高；

# 86. 反射

2021年8月23日 15:13

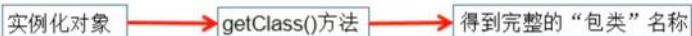
## 一、java反射机制的概述

### Java Reflection

● **Reflection**（反射）是被视为**动态语言**的关键，反射机制允许程序在执行期借助于**Reflection API**取得任何类的内部信息，并能直接操作任意对象的内部属性及方法。

● 加载完类之后，在堆内存的方法区中就产生了一个**Class**类型的对象（一个类只有一个**Class**对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象看到类的结构。**这个对象就像一面镜子，透过这个镜子看到类的结构，所以，我们形象的称之为：反射。**

正常方式： 引入需要的“包类”名称 → 通过new实例化 → 取得实例化对象

反射方式： 实例化对象 → getClass()方法 → 得到完整的“包类”名称

## Java反射机制研究及应用

### ● Java反射机制提供的功能

- 在运行时判断任意一个对象所属的类
- 在运行时构造任意一个类的对象
- 在运行时判断任意一个类所具有的成员变量和方法
- 在运行时获取泛型信息
- 在运行时调用任意一个对象的成员变量和方法
- 在运行时处理注解
- 生成动态代理

## 反射相关的主要API

- **java.lang.Class**: 代表一个类

- **java.lang.reflect.Method**: 代表类的方法
- **java.lang.reflect.Field**: 代表类的成员变量
- **java.lang.reflect.Constructor**: 代表类的构造器
- ... ...

```

//反射之后，对于Person的操作
@Test
public void test2() throws Exception{
    Class clazz = Person.class;
    //1.通过反射，创建Person类的对象
    Constructor cons = clazz.getConstructor(String.class,int.class);
    Object obj = cons.newInstance( ...initargs: "Tom", 12);
    Person p = (Person) obj;
    System.out.println(p.toString());
    //2.通过反射，调用对象指定的属性、方法
    //调用属性
    Field age = clazz.getDeclaredField( name: "age");
    age.set(p,10);
    System.out.println(p.toString());

    //调用方法
    Method show = clazz.getDeclaredMethod( name: "show");
    show.invoke(p);
}

//通过反射，可以调用Person类的私有结构的。比如：私有的构造器、方法、属性
//调用私有的构造器
Constructor cons1 = clazz.getDeclaredConstructor(String.class);
cons1.setAccessible(true);
Person p1 = (Person) cons1.newInstance( ...initargs: "Jerry");
System.out.println(p1);

//调用私有的属性
Field name = clazz.getDeclaredField( name: "name");
name.setAccessible(true);
name.set(p1,"HanMeimei");
System.out.println(p1);

//调用私有的方法
Method showNation = clazz.getDeclaredMethod( name: "showNation", String.
showNation.setAccessible(true);
showNation.invoke(p1, ...args: "中国");//相当于p1.showNation("")
```

疑问：new的方式和反射都可以调用公共的结构，开发中使用哪种方式

建议：new

使用情景：反射的方式；反射的特征：动态性

疑问：反射机制与面向对象的封装性是不是矛盾？如何看待两个技术？

不矛盾。

## 二、java.lang.Class类的理解

### 1.类的加载过程：

程序经过javac.exe命令，生成字节码文件（.class），接着使用java.exe对字节码文件进行解释运行。相当于将某个字节码文件加载到内存中，此过程称为类的加载。  
加载到内存中的类称为运行时类。此运行时类就作为Class的一个实例。

### 2.Class的实例就对应着一个运行时类

### 3.加载到内存中的运行时类，会缓存一段时间。在此期间内，可以通过不同方式获取

这个运行时类。

#### 4. Class类实例方式

```
//方式一：调用运行时类的属性：.class  
Class<Person> clazz1=Person.class;  
  
//方式二：通过运行时类的对象,调用getClass()方法  
Person p1=new Person();  
Class clazz2=p1.getClass();  
  
//方式三：调用Class的静态方法：forName(String classPath)***  
Class clazz3=Class.forName("com.zqf.java.Person");//类的全路径  
  
//方式四（了解）：使用类的加载器：ClassLoader  
ClassLoader classLoader =ReflectionTest.class.getClassLoader();  
Class clazz4=classLoader.loadClass("com.zqf.java.Person");
```

5. Class实例可以时是哪些结构

#### 哪些类型可以有Class对象？

(1) class:

外部类，成员(成员内部类，静态内部类)，局部内部类，匿名内部类

(2) interface: 接口

(3) []: 数组

(4) enum: 枚举

(5) annotation: 注解@interface

(6) primitive type: 基本数据类型

(7) void

```
Class c1 = Object.class;  
Class c2 = Comparable.class;  
Class c3 = String[].class;  
Class c4 = int[][] .class;  
Class c5 = ElementType.class;  
Class c6 = Override.class;  
Class c7 = int.class;  
Class c8 = void.class;  
Class c9 = Class.class; I  
  
int[] a = new int[10];  
int[] b = new int[100];  
Class c10 = a.getClass();  
Class c11 = b.getClass();  
// 只要元素类型与维度一样，就是同一个Class  
System.out.println(c10 == c11);
```

#### 了解：类的加载过程

当程序主动使用某个类时，如果该类还未被加载到内存中，则系统会通过

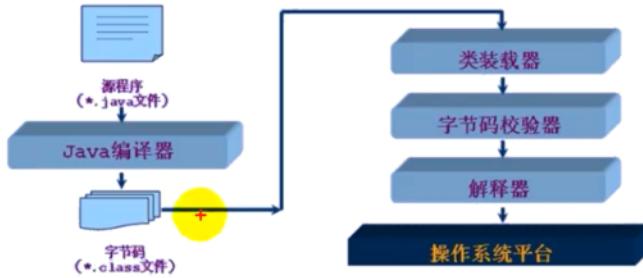
# 了解：类的加载过程

当程序主动使用某个类时，如果该类还未被加载到内存中，则系统会通过如下三个步骤来对该类进行初始化。



- 加载：将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区的运行时数据结构，然后生成一个代表这个类的java.lang.Class对象，作为方法区中类数据的访问入口（即引用地址）。所有需要访问和使用类数据只能通过这个Class对象。这个加载的过程需要类加载器参与。
- 链接：将Java类的二进制代码合并到JVM的运行状态之中的过程。
  - 验证：确保加载的类信息符合JVM规范，例如：以cafe开头，没有安全方面的问题
  - 准备：正式为类变量（static）分配内存并设置类变量默认初始值的阶段，这些内存都将在方法区中进行分配。
  - 解析：虚拟机常量池内的符号引用（常量名）替换为直接引用（地址）的过程。
- 初始化：
  - 执行类构造器<clinit>()方法的过程。类构造器<clinit>()方法是由编译期自动收集类中所有类变量的赋值动作和静态代码块中的语句合并产生的。（类构造器是构造类信息的，不是构造该类对象的构造器）。
  - 当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。|
  - 虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步。

```
public class ClassLoadingTest {  
    public static void main(String[] args) {  
        System.out.println(A.m);  
    }  
}  
  
class A {  
    static {  
        m = 300;  
    }  
    static int m = 100;  
}  
//第二步：链接结束后m=0  
//第三步：初始化后，m的值由<clinit>()方法执行决定  
//          这个A的类构造器<clinit>()方法由类变量的赋值和静态代码块中的语句按照顺序合并产生，类似于  
//          <clinit>(){  
//              m = 300;  
//              m = 100;  
//          }
```

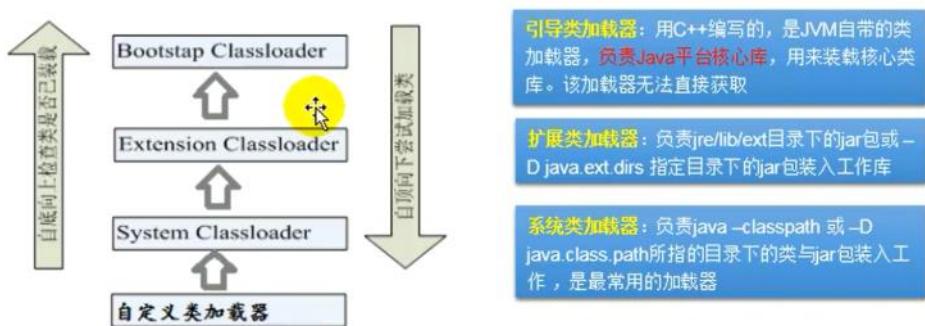


类加载器的作用：

- 类加载的作用：将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区的运行时数据结构，然后在堆中生成一个代表这个类的java.lang.Class对象，作为方法区中类数据的访问入口。
- 类缓存：标准的JavaSE类加载器可以按要求查找类，但一旦某个类被加载到类加载器中，它将维持加载（缓存）一段时间。不过JVM垃圾回收机制可以回收这些Class对象。

## 了解：ClassLoader

类加载器作用是用来把类(class)装载进内存的。JVM 规范定义了如下类型的类的加载器。



自定义类调用系统类加载器，引导类加载器无法获取，返回null；

### 三、properties加载配置文件

读取方式一：

```
FileInputStream fis=null;
try {
    Properties pros=new Properties();
    fis =new FileInputStream( name: "jdbc.properties");//默认使用工程下文件
    pros.load(fis); //加载流对应的文件
    String name = pros.getProperty("name");
    String password = pros.getProperty("password");
    System.out.println(name+"="+password);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        fis.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

读取方式二：

```
public class exer1 {
    public void test(){
        Properties pros=new Properties();
        InputStream is=null;
        try {
            ClassLoader classLoader=exer1.class.getClassLoader();
            //配置文件默认为当moudle下的src中
            is=classLoader.getResourceAsStream( name: "jdbc1.properties");
            pros.load(is);
            String user=pros.getProperty("user");
            String password=pros.getProperty("password");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if(is!=null)
                    is.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

#### 四、使用

##### 1.通过反射创建对应运行时类的实例

```
try {
    //获取运行时类
    Class<Person> clazz=Person.class;
    //newInstance():调用此方法，创建对应的运行时类对象,
    // 内部调用运行时类的空参构造器（权限为非private）
    Person obj=clazz.newInstance();
    System.out.println(obj);
} catch (InstantiationException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
}
```

##### 2.获取运行时类的完整结构

###### ①获取当前运行时类的属性结构（了解）

```
//获取属性结构
//getFields():获取当前运行时类及其父类中声明为public的属性
Field[] fields=clazz.getFields();
for(Field f1:fields){
    System.out.println(f1);
    //获取权限修饰符
    int modifiers = f1.getModifiers();
    System.out.println(Modifier.toString(modifiers));
    //获取数据类型
    Class type = f1.getType();
    System.out.println(type.getName());
    //获取变量名
    String fname=f1.getName();
}
//getDeclaredFields():获取当前运行时类中声明的所有属性（不包含父类中声明属性）
Field[] declaredFields=clazz.getDeclaredFields();
```

## ②获取方法（了解）

```
//获取方法
//getMethods():获取当前运行时类及其所有父类中声明为public的方法
Method[] methods = clazz.getMethods();
//.getDeclaredMethods():获取当前运行时类中所有的方法，不包含父类
Method[] declaredMethods = clazz.getDeclaredMethods();
for(Method m:declaredMethods){
    //获取方法声明的注解
    Annotation[] annotations = m.getAnnotations();
    //获取权限修饰符
    int modifiers = m.getModifiers();
    System.out.println(Modifier.toString(modifiers));
    //返回值类型
    Class returnType = m.getReturnType();
    System.out.println(returnType.getName());
    //方法名
    System.out.println(m.getName());
    //获取形参列表
    Class[] parameterTypes = m.getParameterTypes();
    //抛出的异常，无异常exceptionTypes.length==0
    Class[] exceptionTypes = m.getExceptionTypes();
}
```

## ③其他

```
//获取构造器结构
//getConstructors(): 获取当前运行时类中中声明为public的构造器
Constructor[] constructors = clazz.getConstructors();
//getDeclaredConstructors():获取当前运行时类中声明的所有的构造器
Constructor[] declaredConstructors = clazz.getDeclaredConstructors();

//获取运行时类的父类
Class superclass = clazz.getSuperclass();
//获取运行时类带泛型的父类
Type genericSuperclass = clazz.getGenericSuperclass();
ParameterizedType paramType=(ParameterizedType) genericSuperclass;
//获取运行时类带泛型的父类的泛型
Type[] actualTypeArguments = paramType.getActualTypeArguments();
System.out.println(actualTypeArguments[0].getTypeName());

//获取运行时类实现的接口
Class[] interfaces = clazz.getInterfaces();
//获取运行时类父类实现的接口
Class[] interfaces1 = clazz.getSuperclass().getInterfaces();

//获取运行时类所在的包
Package aPackage = clazz.getPackage();

//获取运行时类声明的注解
Annotation[] annotations1 = clazz.getAnnotations();
```

### 3.调用运行时类的指定属性/方法

#### 属性

```

Class<Person> clazz=Person.class;
//创建运行时类对象
Person person = clazz.newInstance();
//获取指定的属性:要求运行时类属性为public的,通常不使用
Field id=clazz.getField( name: "id");
//设置当前属性的值
//set():参数1:指明设置哪个对象的属性 参数2:将此属性值设置为多少
id.set(person,1001);
//获取当前属性的值
//get():参数1:获取哪个对象的当前属性值
int pId=(int)id.get(person);
System.out.println(pId);

//掌握这种
//1.getDeclaredField(String fieldName):获取运行时类中指定变量名的属性
Field name = clazz.getDeclaredField( name: "name");
//2.保证当前属性是可访问的
name.setAccessible(true);
//3.获取、设置指定对象的此属性值
name.set(person,"zhangqianfeng");

```

## 方法

```

//操作运行时类中的指定的方法--需要掌握
//1.获取指定的某个方法
//getDeclaredMethod(),参数1:指明获取的方法的名称 参数2: 指明获取方法的形参列表
Method show = clazz.getDeclaredMethod( name: "show", String.class);
//2.保证当前方法是可访问的
show.setAccessible(true);
//3.invoke():参数1: 方法的调用对象 参数2:给方法形参赋值的实参
// invoke()方法的返回值即为对应类中调用的方法的返回值
String zhang =(String)show.invoke(person, ...args: "zhang");

//如何调用静态方法, 对象参数设置为null或者类.class
Method show1 = clazz.getDeclaredMethod( name: "show");
show1.setAccessible(true);
//如果没有返回值此时invoke()方法返回null
show1.invoke(Person.class);

//如何调用运行时类中指定的构造器 (用的少)
//1.获取指定的构造器
//getDeclaredConstructor():参数:指明构造器的参数列表
Constructor<Person> declaredConstructor = clazz.getDeclaredConstructor(String.class);
//2.保证构造器是可访问的
declaredConstructor.setAccessible(true);
//3.调用此构造器创建运行时类对象
Person person1 = declaredConstructor.newInstance( ...initargs: "Tom");

```

# 87. 反射的应用-动态代理

2021年8月30日 12:45

## ●代理设计模式的原理：

使用一个代理将对象包装起来，然后用该代理对象取代原始对象。任何对原始对象的调用都要通过代理。代理对象决定是否以及何时将方法调用转到原始对象上。

●之前为大家讲解过代理机制的操作，属于静态代理，特征是代理类和目标对象的类都是在编译期间确定下来，不利于程序的扩展。同时，每一个代理类只能为一个接口服务，这样在程序开发中必然产生过多的代理。**最好可以通过一个代理类完成全部的代理功能。**

●动态代理是指客户通过代理类来调用其它对象的方法，并且是在程序运行时根据需要动态创建目标类的代理对象。

●动态代理使用场合：

- 调试
- 远程方法调用

●动态代理相比于静态代理的优点：

抽象角色中（接口）声明的所有方法都被转移到调用处理器一个集中的方法中处理，这样，我们可以更加灵活和统一的处理众多的方法。

## 一、动态代理举例 (day10)

```
package com.zqf.exer;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * 动态代理的举例
 * @author oscarzqf
 * @description
 * @create 2021-08-30-15:23
 */
interface Human{
    String getBelief();
    void eat(String food);
}

//被代理类
class SuperMan implements Human{
    @Override
    public String getBelief() {
        return "I believe I can fly!";
    }

    @Override
    public void eat(String food) {
        System.out.println("我喜欢吃"+food);
    }
}

//实现动态代理，需要解决的问题？
//一、如何根据加载到内存中的被代理类，动态的创建一个类及其对象
//二、通过代理类的对象调用方法时，如何动态调用被代理类中同名方法
class ProxyFactory{
    //解决问题1
```

```

//二、通过代理类的对象调用方法时，如何动态调用被代理类中同名方法
class ProxyFactory{
    //解决问题1
    public static Object getProxyInstance(Object obj){//obj:被代理类的对象
        MyInvocationHandler handler = new MyInvocationHandler();
        handler.bind(obj);
        return Proxy.newProxyInstance(obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),handler);
    }
}

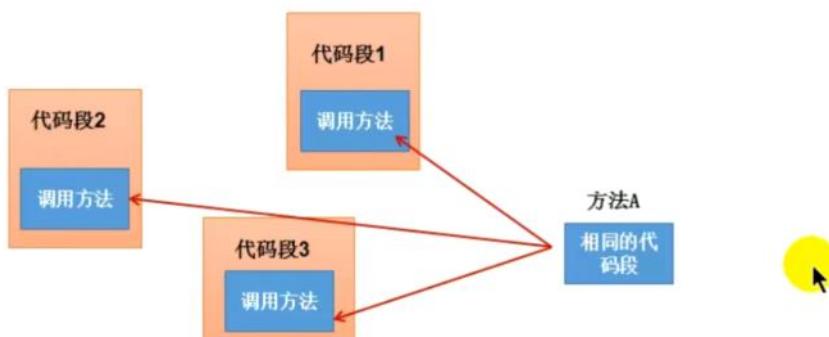
class MyInvocationHandler implements InvocationHandler{
    private Object obj;//赋值时需要使用被代理类对象赋值
    public void bind(Object obj){
        this.obj=obj;
    }
    //当我们通过代理类的对象调用方法a时，就会自动调用如下的方法：invoke()
    //将被代理类要执行的方法a的功能声明在invoke()中
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //method即为代理类对象调用的方法，此方法也就作为被代理类对象要调用的方法
        //obj: 被代理的对象
        Object returnValue = method.invoke(obj, args);
        //上述方法的返回值就作为invoke()的返回值
        return returnValue;
    }
}

public class ProxyTest {
    public static void main(String[] args) {
        SuperMan superMan=new SuperMan();
        //proxyInstance:代理类对象
        Human proxyInstance = (Human)ProxyFactory.getProxyInstance(superMan);
        //当通过代理类对象调用方法时，会自动调用被代理类中同名的方法
        String belief = proxyInstance.getBelief();
        System.out.println(belief);
        proxyInstance.eat(food:"hotpot");
    }
}

```

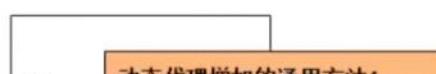
## 二、

### 动态代理与AOP (Aspect Orient Programming)

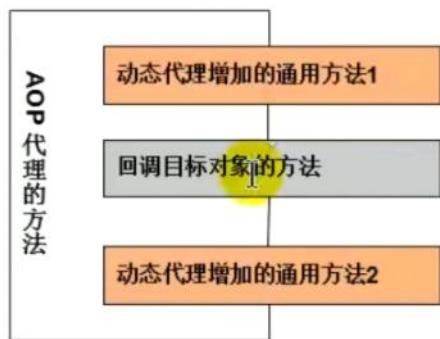


改进后的说明：代码段1、代码段2、代码段3和深色代码段分离开了，但代码段1、2、3又和一个特定的方法A耦合了！最理想的效果是：代码块1、2、3既可以执行方法A，又无须在程序中以硬编码的方式直接调用深色代码的方法

### 动态代理与AOP (Aspect Orient Programming)



## 动态代理与AOP (Aspect Orient Programming)



# 88.Java8的其他新特性

2021年8月30日 17:16

## Java 8新特性简介

Java 8 (又称为 jdk 1.8) 是 Java 语言开发的一个主要版本。

Java 8 是oracle公司于2014年3月发布，可以看成是自Java 5以来最具革命性的版本。Java 8为Java语言、编译器、类库、开发工具与JVM带来了大量新特性。

- 速度更快
- 代码更少(增加了新的语法：**Lambda 表达式**)
- 强大的 **Stream API**
- 便于并行 | I
- 最大化减少空指针异常： **Optional**
- **Nashorn**引擎，允许在JVM上运行JS应用

### 一、Lambda表达式的使用

#### 为什么使用 **Lambda 表达式**

Lambda 是一个**匿名函数**，我们可以把 Lambda 表达式理解为是一段可以**传递的代码**（将代码像数据一样进行传递）。使用它可以写出更简洁、更灵活的代码。作为一种更紧凑的代码风格，使Java的语言表达能力得到了提升。

##### 1.例子

```

//普通写法
Comparator<Integer> com1=new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return Integer.compare(o1,o2);
    }
};

//Lambda表达式写法
Comparator<Integer> com2=(o1,o2)->Integer.compare(o1,o2);

//方法引用
Comparator<Integer> com3=Integer::compare;

```

## 2.格式

- >:Lambda操作符 或 箭头操作符
- >左边： Lambda形参列表（就是接口中抽象方法的形参列表）
- >右边： Lambda体（就是重写的抽象方法的方法体）

## 3.Lambda表达式的使用：（分6种情况）

### ①无参无返回值

```
Runnable r1=()->{sout("111");
};
```

### ②需要一个参数，但是没有返回值

```
Consumer<String> con=(String s)->{sout(s);
};
```

### ③数据类型可以省略，因为可以由编译器推断得出，称为“类型推断”

```
Consumer<String> con=(s)->{sout(s);
};
```

### ④若只需要一个参数，参数的（）可以省略

```
Consumer<String> con= s->{sout(s);
};
```

### ⑤需要两个或两个以上的参数，多条执行语句，并且可以有返回值

```
Comparator<Integer> com=(o1,o2)->{
    sout(1);
    sout(2);
    sout(3);
    return o1.compareTo(o2);
};
```

### ⑥只有一条语句时，return与大括号若有，都可以省略

```
Comparator<Integer> com= (o1,o2) ->o1.compareTo(o2);
```

总结：

左边：省略参数类型，省略（）一个

右边：Lambda体使用{}包裹；一条语句省略{}，return；

## 4.Lambda表达式的本质：作为函数式接口的实例

5.如果一个接口中只声明了一个抽象方法，则此接口就称为函数式接口

\*可以使用@FunctionalInterface注解，检验接口是否为函数接口

\*@FunctionalInterface javadoc解析会保留

\*使用Lambda实例化

\*java.util.function包下定义了Java8丰富的函数式接口

### Java 内置四大核心函数式接口

函数式接口	参数类型	返回类型	用途
Consumer<T> 消费型接口	T	void	对类型为T的对象应用操作，包含方法： <code>void accept(T t)</code>
Supplier<T> 供给型接口	无	T	返回类型为T的对象，包含方法： <code>T get()</code>
Function<T, R> 函数型接口	T	R	对类型为T的对象应用操作，并返回结果。结果是R类型的对象。包含方法： <code>R apply(T t)</code>
Predicate<T> 断定型接口	T	boolean	确定类型为T的对象是否满足某约束，并返回 boolean 值。包含方法： <code>boolean test(T t)</code>

### 其他接口

函数式接口	参数类型	返回类型	用途
BiFunction<T, U, R>	T, U	R	对类型为 T, U 参数应用操作，返回 R 类型的结果。包含方法为： <code>R apply(T t, U u);</code>
UnaryOperator<T> (Function子接口)	T	T	对类型为T的对象进行一元运算，并返回T类型的结果。包含方法为： <code>T apply(T t);</code>
BinaryOperator<T> (BiFunction 子接口)	T, T	T	对类型为T的对象进行二元运算，并返回T类型的结果。包含方法为： <code>T apply(T t1, T t2);</code>
BiConsumer<T, U>	T, U	void	对类型为T, U 参数应用操作。 包含方法为： <code>void accept(T t, U u)</code>
BiPredicate<T, U>	T, U	boolean	包含方法为： <code>boolean test(T t, U u)</code>
ToIntFunction<T> ToLongFunction<T> ToDoubleFunction<T>	T	int long double	分别计算int、long、double值的函数
IntFunction<R> LongFunction<R> DoubleFunction<R>	int long double	R	参数分别为int、long、double类型的函数

## 二、方法引用

1.使用情景：

当要传递给Lambda体的操作，已经有了实现方法了，可以使用方法引用

2.方法引用，本质是Lambda表达式作为函数接口的实例，

所以方法引用也是函数接口的实例

3.使用格式：

类（或对象）：：方法名

4.具体分三种情况

对象：：非静态方法

类：：静态方法

类：：非静态方法

```
//对象：：实例方法
//Consumer中的void accept(T t)
//PrintStream中的void println(T t)
PrintStream ps=System.out;
Consumer<String> cust=ps::println;
cust.accept( t: "测试方法引用的例子");

//类：：静态方法
//Comparator中的int compare(T t1,T t2)
//Integer中的int compare(T t1,T t2)
Comparator<Integer> com1=Integer::compare;
System.out.println(com1.compare(12,3));

//类：：非静态方法（难）
//Comparator中的 int compare(T t1,T t2)
//String 中的int t1.compareTo(t2)
Comparator<String> com2=String::compareTo;
System.out.println(com2.compare("abc","bcd"));
```

5.方法引用使用的要求：要求接口中的抽象方法的形参列表和返回值类型与方法引用的方法的形参列表和返回值类型相同！

### 三、构造器引用

```
//构造器引用
//Supplier中的T get()
//Employee类的空参构造器：Employee()
Supplier<Employee> sup=Employee::new;
//Function中的R apply(T t)
Function<String,Employee> fn=Employee::new;
//BiFunction中的R apply(T t,U u)
```

和方法的引用类似，函数式接口的抽象方法的形参列表和构造器的形参列表一致  
抽象方法的返回值类型即为构造器所属的类的类型

### 四、数组引用

```
//可以把数组看做一个特殊的类，则写法与构造器一致
//Function中的R apply(T t)
Function<Integer, String[]> fun = String[]::new;
String[] apply = fun.apply(t: 10);
for(String val: apply){
    System.out.println(val);
}
```

## 五、强大的Stream API

### Stream API说明

- Java8中有两大最为重要的改变。第一个是 **Lambda 表达式**；另外一个则是 **Stream API**。
- **Stream API ( java.util.stream)** 把真正的函数式编程风格引入到Java中。这是目前为止对Java类库最好的补充，因为**Stream API**可以极大提供Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。
- **Stream** 是 Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。**使用 Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。**也可以使用 **Stream API** 来并行执行操作。简言之，**Stream API** 提供了一种高效且易于使用的处理数据的方式。
- 实际开发中，项目中多数数据源都来自于**Mysql, Oracle**等。但现在数据源可以更多了，有**MongDB, Redis**等，而这些**NoSQL**的数据就需要**Java**层面去处理。
- Stream 和 Collection 集合的区别：**Collection** 是一种静态的内存数据结构，而 **Stream** 是有关计算的。前者是主要面向内存，存储在内存中，后者主要是面向 CPU，通过 CPU 实现计算。

### 什么是 Stream

#### Stream到底是什么呢？

是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。

“**集合讲的是数据，Stream讲的是计算！**”

#### 注意：

- ① Stream 自己不会存储元素。
- ② Stream 不会改变源对象。相反，他们会返回一个持有结果的新Stream。
- ③ Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

# Stream 的操作三个步骤

## ● 1- 创建 Stream

一个数据源（如：集合、数组），获取一个流

## ● 2- 中间操作

一个中间操作链，对数据源的数据进行处理

## ● 3- 终止操作(终端操作)

一旦执行终止操作，就执行中间操作链，并产生结果。之后，不会再被使用



## 1. 创建 Stream 对象

```
//创建方式一：通过集合
List<Employee> employees=new ArrayList<>();
//default Stream<E> stream():返回一个顺序流
Stream<Employee> stream=employees.stream();
//default Stream<E> parallelStream():返回一个并行流
Stream<Employee> parallelStream=employees.parallelStream();

//创建方式二：通过数组
//调用Arrays类的static <T> Stream<T> stream(T[] array):返回一个流
int[] arr={1,2,3,4};
IntStream stream1= Arrays.stream(arr);
Employee[] arr1=new Employee[3];
Stream<Employee> stream2=Arrays.stream(arr1);

//创建方式三：通过Stream的of()
Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5);

//创建方式四：创建无限流(了解)
//迭代
//public static<T> Stream<T> iterate(final T seed,final UnaryOperator<T> f)
//遍历前10个偶数
Stream.iterate( seed: 0,t->t+2).limit(10).forEach(System.out::println);
//生成
//public static<T> Stream<T> generate(Supplier<T> s)
Stream.generate(Math::random).limit(10).forEach(System.out::println);
```

## 2. Stream 的中间操作

### ① 筛选与切片

```
List<Employee> employees=new ArrayList<>();
//筛选与切片
//filter(Predicate p)--接收Lambda，从流中排除某些元素
//查询工资大于7000的员工
employees.stream().filter(e->e.getSalary()>7000).forEach(System.out::println);
//limit(n)--截断流，使其元素不超过给定的数量
employees.stream().limit(3).forEach(System.out::println);
//skip(n)--跳过元素，返回一个扔掉了前n个元素的流。若流中元素不足n，则返回一个空流。
employees.stream().skip(3).forEach(System.out::println);
//distinct()--筛选，通过流所产生的元素的hashCode()和equals()去除重复元素
employees.stream().distinct().forEach(System.out::println);
```

## ②映射

```
//map(Function f)--接收一个函数作为参数，将元素转换成其他形式或提取信息，  
//该元素会被应用到每个元素上，并将其映射成一个新的元素  
List<String> list =Arrays.asList("aa","cc","bb","dd");  
list.stream().map(str->str.toUpperCase()).forEach(System.out::println);  
//flatMap(Functon f)--接收一个函数作为参数，将流中的每个值都换成另一个流，  
//然后把所有流连接成一个流。  
//函数f返回一个Stream
```

## ③排序

```
//排序sorted()--自然排序  
List<Integer> list1=Arrays.asList(12,22,33,44,11,2,3,7,66);  
list1.stream().sorted().forEach(System.out::println);  
//sorted(Comparator com)--定制排序  
list1.stream().sorted((e1,e2)->{  
    return -e1.compareTo(e2);  
}).forEach(System.out::print);
```

## 3.终止操作

### ①匹配与查找

```
//allMatch(Predicate p)--检查是否匹配所有元素  
boolean b = list1.stream().allMatch(e -> e > 0);  
//anyMatch(Predicate p)--检查是否至少匹配一个元素  
//noneMatch(Predicate p)--检查是否没有匹配的元素  
//findFirst()--返回第一个元素  
//findAny()--返回当前流中的任意一个元素  
//count()--返回流中元素的总个数  
//max(Comparator c)--返回流中的最大值  
//min(Comparator c)--返回流中  
//forEach(Consumer c)--内部迭代
```

### ②归约

```
List<Integer> list1=Arrays.asList(12,22,33,44,11,2,3,7,66);  
//reduce(T identity,BinaryOperator)--可以将流中的元素反复结合起来  
//得到一个值。返回T  
Integer reduce = list1.stream().reduce( identity: 0, Integer::sum);  
System.out.println(reduce);  
//reduce(BinaryOperator b)--可以将流中元素反复结合起来，  
//得到一个值，返回Optional<T>, 无初始值  
Optional<Integer> reduce1 = list1.stream().reduce(Integer::sum);  
//Optional<Integer> reduce1 = list1.stream().reduce((a,b)->a+b);  
System.out.println(reduce1);
```

### ③收集

# Stream 的终止操作

## 3-收集

方法	描述
collect(Collector c)	将流转换为其他形式。接收一个 Collector 接口的实现，用于给 Stream 中元素做汇总的方法

Collector 接口中方法的实现决定了如何对流执行收集的操作(如收集到 List、Set、Map)。

另外，Collectors 实用类提供了很多静态方法，可以方便地创建常见收集器实例，具体方法与实例如下表：

## 16.4 强大的Stream API: Collectors

方法	返回类型	作用
toList	List<T>	把流中元素收集到List
List<Employee> emps= list.stream().collect(Collectors.toList());		
toSet	Set<T>	把流中元素收集到Set
Set<Employee> emps= list.stream().collect(Collectors.toSet());		
toCollection	Collection<T>	把流中元素收集到创建的集合
Collection<Employee> emps =list.stream().collect(Collectors.toCollection(ArrayList::new));		
counting	Long	计算流中元素的个数
long count = list.stream().collect(Collectors.counting());		
summingInt	Integer	对流中元素的整数属性求和
int total=list.stream().collect(Collectors.summingInt(Employee::getSalary));		
averagingInt	Double	计算流中元素Integer属性的平均值
double avg = list.stream().collect(Collectors.averagingInt(Employee::getSalary));		
summarizingInt	IntSummaryStatistics	收集流中Integer属性的统计值。如：平均值
int Summary Statisticsiss= list.stream().collect(Collectors.summarizingInt(Employee::getSalary));		

joining	String	连接流中每个字符串
String str= list.stream().map(Employee::getName).collect(Collectors.joining());		
maxBy	Optional<T>	根据比较器选择最大值
Optional<Emp>max= list.stream().collect(Collectors.maxBy(comparingInt(Employee::getSalary)));		
minBy	Optional<T>	根据比较器选择最小值
Optional<Emp> min = list.stream().collect(Collectors.minBy(comparingInt(Employee::getSalary)));		
reducing	归约产生的类型	从一个作为累加器的初始值开始，利用BinaryOperator与流中元素逐个结合，从而归约成单个值
int total=list.stream().collect(Collectors.reducing(0, Employee::getSalar, Integer::sum));		
collectingAndThen	转换函数返回的类型	包裹另一个收集器，对其结果转换函数
int how= list.stream().collect(Collectors.collectingAndThen(Collectors.toList(), List::size));		
groupingBy	Map<K, List<T>>	根据某属性值对流分组，属性为K，结果为V
Map<Emp.Status, List<Emp>> map= list.stream() .collect(Collectors.groupingBy(Employee::getStatus));		
partitioningBy	Map<Boolean, List<T>>	根据true或false进行分区
Map<Boolean,List<Emp>> vd = list.stream().collect(Collectors.partitioningBy(Employee::getManage));		

```
List<Integer> list1=Arrays.asList(12,22,33,44,11,2,3,7,66);  
//collect(Collector c)--将流转换为其他形式。接收一个Collector  
//接口的实现，用于给 Stream 中元素做汇总的方法  
List<Integer> collect = list1.stream().filter(a -> a > 30)  
    .collect(Collectors.toList());  
collect.forEach(System.out::println);
```

## 六、Optional类

### 为了避免空指针创建的类

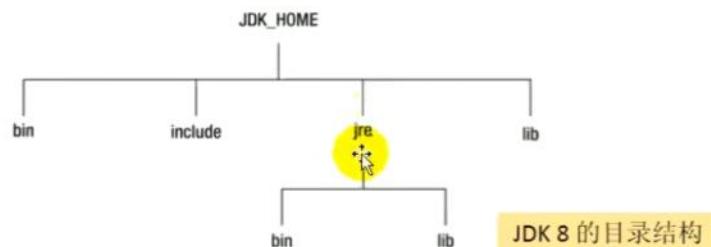
- 到目前为止，臭名昭著的空指针异常是导致Java应用程序失败的最常见原因。以前，为了解决空指针异常，Google公司著名的Guava项目引入了Optional类，Guava通过使用检查空值的方式来防止代码污染，它鼓励程序员写更干净的代码。受到Google Guava的启发，Optional类已经成为Java 8类库的一部分。
- Optional<T> 类(java.util.Optional) 是一个容器类，它可以保存类型T的值，代表这个值存在。或者仅仅保存null，表示这个值不存在。原来用 null 表示一个值不存在，现在 Optional 可以更好的表达这个概念。并且可以避免空指针异常。
- Optional类的Javadoc描述如下：这是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。
- Optional提供很多有用的方法，这样我们就不用显式进行空值检测。
- 创建Optional类对象的方法：
  - Optional.of(T t)：创建一个 Optional 实例，t必须非空；
  - Optional.empty()：创建一个空的 Optional 实例
  - Optional.ofNullable(T t)：t可以为null
- 判断Optional容器中是否包含对象：
  - boolean isPresent()：判断是否包含对象
  - void ifPresent(Consumer<? super T> consumer)：如果有值，就执行Consumer接口的实现代码，并且该值会作为参数传给它。
- 获取Optional容器的对象：
  - T get()：如果调用对象包含值，返回该值，否则抛异常
  - T orElse(T other)：如果有值则将其返回，否则返回指定的other对象。
  - T orElseGet(Supplier<? extends T> other)：如果有值则将其返回，否则返回由Supplier接口实现提供的对象。
  - T orElseThrow(Supplier<? extends X> exceptionSupplier)：如果有值则将其返回，否则抛出由Supplier接口实现提供的异常。

```
public class OptionalTest {  
    public static void main(String[] args) {  
        Girl girl=new Girl(name: "001");  
        //of(T t):创建一个Optional实例，t必须非空  
        Optional<Girl> optionalGirl=Optional.of(girl);  
        //ofNullable(T t):t可以为null  
        Optional<Girl> optionalGirl1=Optional.ofNullable(girl);  
        //orElse(T other):如果有值则返回，否则返回other。  
        // 相当于设置一个默认值避免空指针  
        Girl zahngjxj = optionalGirl1.orElse(new Girl(name: "zahngjxj")); //一定非空  
        System.out.println(zahngjxj.name);  
    }  
}  
class Girl{  
    String name;  
    public Girl() {  
    }  
    public Girl(String name) {  
        this.name = name;  
    }  
}
```

## JDK 9 的发布

- 经过4次跳票，历经曲折的Java 9 终于在2017年9月21日发布。
- 从Java 9 这个版本开始，Java 的计划发布周期是 **6个月**，下一个 Java 的主版本将于 2018 年 3 月发布，命名为 **Java 18.3**，紧接着再过六个月将发布 **Java 18.9**。
- 这意味着Java的更新从传统的以**特性驱动**的发布周期，转变为以**时间驱动**的（6 个月为周期）发布模式，并逐步的将 Oracle JDK 原商业特性进行开源。
- 针对企业客户的需求，Oracle 将以**三年为周期**发布长期支持版本（**long term support**）。
- Java 9 提供了**超过150项新功能**特性，包括备受期待的模块化系统、可交互的 **REPL 工具**: jshell, JDK 编译工具, Java 公共 API 和私有代码，以及安全增强、扩展提升、性能管理改善等。可以说**Java 9是一个庞大的系统工程，完全做了一个整体改变。**

oracle.com 下载新版JDK，文档

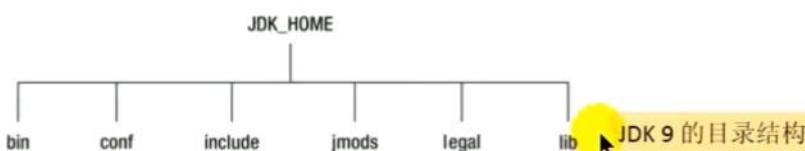


<b>bin 目录</b>	包含命令行开发和调试工具，如javac, jar和javadoc。
<b>include 目录</b>	包含在编译本地代码时使用的C/C++头文件
<b>lib 目录</b>	包含JDK工具的几个JAR和其他类型的文件。它有一个tools.jar文件，其中包含javac编译器的Java类
<b>jre/bin 目录</b>	包含基本命令，如java命令。在Windows平台上，它包含系统的运行时动态链接库（DLL）。
<b>jre/lib 目录</b>	包含用户可编辑的配置文件，如.properties和.policy文件。包含几个JAR。rt.jar文件包含运行时的Java类和资源文件。

见后

## 一、JDK 和 JRE 目录结构的改变

尚硅谷



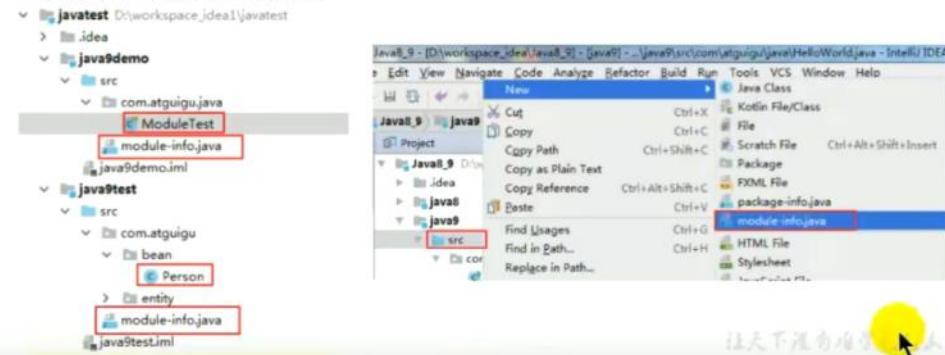
没有名为jre的子目录	
<b>bin 目录</b>	包含所有命令。在Windows平台上，它继续包含系统的运行时动态链接库。
<b>conf 目录</b>	包含用户可编辑的配置文件，例如以前位于jre\lib目录中的.properties和.policy文件
<b>include 目录</b>	包含要在以前编译本地代码时使用的C/C++头文件。它只存在于JDK中
<b>jmods 目录</b>	包含JMOD格式的平台模块。创建自定义运行时映像时需要它。它只存在于JDK中
<b>legal 目录</b>	包含法律声明
<b>lib 目录</b>	包含非Windows平台上的动态链接本地库。其子目录和文件不应由开发人员直接编辑或使用

### 1.模块化

- 谈到 Java 9 大家往往第一个想到的就是 Jigsaw 项目。众所周知，Java 已经发展超过 20 年（95 年最初发布），Java 和相关生态在不断丰富的同时也越来越暴露出一些问题：

- Java 运行环境的膨胀和臃肿。每次JVM启动的时候，至少会有30~60MB的内存加载，主要原因是JVM需要加载rt.jar，不管其中的类是否被classloader加载，第一步整个jar都会被JVM加载到内存当中去（而模块化可以根据模块的需要加载程序运行需要的class）
- 当代码库越来越大，创建复杂，盘根错节的“意大利面条式代码”的几率呈指数级的增长。不同版本的类库交叉依赖导致让人头疼的问题，这些都阻碍了 Java 开发和运行效率的提升。
- 很难真正地对代码进行封装，而系统并没有对不同部分（也就是 JAR 文件）之间的依赖关系有个明确的概念。每一个公共类都可以被类路径之下任何其它的公共类所访问到，这样就会导致无意中使用了并不想被公开访问的 API。

模块将由通常的类和新的模块声明文件（**module-info.java**）组成。该文件是位于 **java** 代码结构的顶层，该模块描述符明确地定义了我们的**模块需要什么依赖关系，以及哪些模块被外部使用**。在**exports**子句中未提及的所有包默认情况下将封装在模块中，不能在外部使用。



### 需要被使用的模块

```
moudle 模块名{
    exports 包名;
}
```

### 使用模块

```
moudle 模块名{
    requires 模块名
}
```

## 2. Java 的REPL工具：jShell命令

## ● 产生背景

像Python 和 Scala 之类语言早就有交互式编程环境 REPL (read - evaluate - print - loop)了，以交互式的方式对语句和表达式进行求值。开发者只需要输入一些代码，就可以在编译前获得对程序的反馈。而之前的Java版本要想执行代码，必须创建文件、声明类、提供测试方法方可实现。

## ● 设计理念

即写即得、快速运行

## ● 实现目标

- Java 9 中终于拥有了 REPL 工具：jShell。让 Java 可以像脚本语言一样运行，从控制台启动 jShell，利用 jShell 在没有创建类的情况下直接声明变量，计算表达式，执行语句。即开发时可以在命令行里直接运行 Java 的代码，而无需创建 Java 文件，无需跟人解释 “`public static void main(String[] args)`” 这句废话。
- jShell 也可以从文件中加载语句或者将语句保存到文件中。
- jShell 也可以是 tab 键进行自动补全和自动添加分号。

## 3. 语法改进：接口的私有方法

```
interface MyInterface{  
    //方法的权限修饰符都是public  
    static void staticMethod(){  
        System.out.println("我是静态方法");  
    }  
    default void defaultMethod(){  
        System.out.println("我是默认方法");  
    }  
    private void privateMethod(){  
        System.out.println("我是私有方法");  
    }  
    void add(); //抽象方法  
}
```

## 4. 钻石操作符的升级使用

```
//钻石操作符与匿名实现类在java 8中不能共存，在java 9中可以使用  
public static void main(String[] args) {  
    Comparator<Object> com=new Comparator<Object>() {  
        @Override  
        public int compare(Object o1, Object o2) {  
            return 0;  
        }  
    };
```

## 5. try操作的升级

```

//java8 中可以实现资源的自动关闭，但资源的初始化必须创建在()中
try(InputStreamReader reader=new InputStreamReader(System.in)){
    char[] cbuf=new char[20];
    int len;
    if((len=reader.read(cbuf))!=-1){
        String str=new String(cbuf, offset: 0, len);
        System.out.println(str);
    }
} catch (IOException e) {
    e.printStackTrace();
}

//java 9中资源自动关闭操作，资源的实例化可以放在外面，多个，隔开
//此时的资源属性是常量，声明为final的，不可修改：
InputStreamReader reader=new InputStreamReader(System.in);
try(reader){
    char[] cbuf=new char[20];
    int len;
    if((len=reader.read(cbuf))!=-1){
        String str=new String(cbuf, offset: 0, len);
        System.out.println(str);
    }
} catch (IOException e) {
    e.printStackTrace();
}

```

6.String存储结构变更，数组变为byte[],StringBuffer也变了，与他们有关的都变了

### Motivation

The current implementation of the String class stores characters in a char array, using two bytes (sixteen bits) for each character. Data gathered from many different applications indicates that strings are a major component of heap usage and, moreover, that most String objects contain only Latin-1 characters. Such characters require only one byte of storage, hence half of the space in the internal char arrays of such String objects is going unused.

### Description

We propose to change the internal representation of the String class from a UTF-16 char array to a byte array plus an encoding-flag field. The new String class will store characters encoded either as ISO-8859-1/Latin-1 (one byte per character), or as UTF-16 (two bytes per character), based upon the contents of the string. The encoding flag will indicate which encoding is used.

7.集合工厂方法：快速创建只读集合

```

//java8
List<Integer> list=new ArrayList<>();
list.add(1);
list.add(66);
//返回只读集合
list= Collections.unmodifiableList(list);
System.out.println(list.get(0));
//此时的到的集合也是只读集合
List<Integer> list1= Arrays.asList(1,2,3,4);
//java9 of()方法
List<Integer> integers = List.of(1, 2, 3);
Set<Integer> integers1 = Set.of(1, 2, 3);

```

## 8.InputStream的新方法: transferTo()

把输入流中的所有数据直接自动的复制到输出流中

## 9.增强的Stream API

- Java 的 Stream API 是 java 标准库最好的改进之一，让开发者能够快速运算，从而能够有效的利用数据并行计算。Java 8 提供的 Stream 能够利用多核架构实现声明式的数据处理。
- 在 Java 9 中，Stream API 变得更好，Stream 接口中添加了 4 个新的方法：takeWhile, dropWhile, ofNullable，还有个 iterate 方法的新重载方法，可以让你提供一个 Predicate (判断条件) 来指定什么时候结束迭代。
- 除了对 Stream 本身的扩展，Optional 和 Stream 之间的结合也得到了改进。现在可以通过 Optional 的新方法 stream() 将一个 Optional 对象转换为一个 (可能是空的) Stream 对象。

```

List<Integer> list=Arrays.asList(1,2,1,8,99,66,55,77);
//takeWhile(): 返回从开头开始的按照指定规则尽可能多的元素，遇见不满足元素就终止
list.stream().takeWhile(p->p<3).forEach(System.out::println);
//dropWhile():与takeWhile相反，返回剩余的元素
list.stream().dropWhile(p->p<3).forEach(System.out::println);

//of()参数中的多个元素，可以包含null值，不能只包含一个null，of(null)，报异常
Stream<Integer> integerStream = Stream.of(1, 2, 1, null);
//ofNullable():参数可以为单个null;

//java8
Stream.iterate( seed: 0, x->x+1).limit(10).forEach(System.out::println);
//java9重载方法，自定义终止条件
Stream.iterate( seed: 0, x->x<100, x->x+1).forEach(System.out::println);

```

## 10.Optional获取Stream的方法

```
//Optional提供了新的方法stream()
List<Integer> list=Arrays.asList(1,2,1,8,99,66,55,77);
Optional<List<Integer>> list1 = Optional.ofNullable(list);
Stream<List<Integer>> stream = list1.stream();
stream.flatMap(x->x.stream()).forEach(System.out::println);
```

## Java10新特性(了解)

### 1.var

```
//局部变量的类型推断
//声明变量时，根据所赋的值推断变量的类型
var list :List<Integer> =Arrays.asList(1,2,1,8,99,66,55,77);
var num=10;
var list1=new ArrayList<String >();
for(var i:list){
    System.out.println(i);
}
//不能使用的情况
//1.不赋值不能推断 var a;
//2.Lambda表达式左边函数式接口不能声明为var
//3.方法引用中，左边函数式接口不能var
//4.数组静态初始化中 var arr={1,2,3}不可
//5.方法的返回类型
//6.方法的参数类型
//7.构造器的参数类型
//8.属性
//9.catch块
```

### 工作原理

在处理 var 时，编译器先是查看表达式右边部分，并根据右边变量值的类型进行推断，作为左边变量的类型，然后将该类型写入字节码当中。

### 注意

#### ● var不是一个关键字

你不需要担心变量名或方法名会与 var发生冲突，因为 var实际上并不是一个关键字，而是一个类型名，只有在编译器需要知道类型的地方才需要用到它。除此之外，它就是一个普通合法的标识符。也就是说，除了不能用它作为类名，其他的都可以，但极少人会用它作为类名。

#### ● 这不是JavaScript

首先我要说明的是，var并不会改变Java是一门静态类型语言的事实。编译器负责推断出类型，并把结果写入字节码文件，就好像是开发人员自己敲入类型一样。

下面是使用 IntelliJ（实际上是 Fernflower 的反编译器）反编译器反编译出的代码：

## 2. 集合中新增copyOf(),用于创建一个只读的集合

```
//示例1:  
var list1 = List.of("Java", "Python", "C");  
var copy1 = List.copyOf(list1);  
System.out.println(list1 == copy1); // true  
  
//示例2:  
var list2 = new ArrayList<String>();  
list2.add("aaa");  
var copy2 = List.copyOf(list2);  
System.out.println(list2 == copy2); // false  
  
//示例1和2代码基本一致，为什么一个为true，一个为false?  
//结论：copyOf(Xxx coll)：如果参数coll本身就是一个只读集合，则copyOf()返回值即  
//如果参数coll不是一个只读集合，则copyOf()返回一个新的集合，这个集合是只读的。
```

## Java11 (LTS, 长期支持版本) 新特性

### 1.String中新增的方法

isBlank():字符串是否为空白

strip():去除首尾空白

stripTrailing():去除尾部空格

stripLeading():去除首部空格

repeat(int count):复制几次字符串

lines().count():行数统计

### 2.Optional

Optional 也增加了几个非常酷的方法，现在可以很方便的将一个 Optional 转换为一个 Stream，或者当一个空 Optional 时给它一个替代的。

新增方法	描述	新增的版本
boolean isEmpty()	判断value是否为空	JDK 11
ifPresentOrElse(Consumer<? extends T> action, Runnable emptyAction)	value非空，执行参数1功能；如果value为空，执行参数2功能	JDK 9
Optional<T> or(Supplier<? extends Optional<T>> supplier)	value非空，返回对应的Optional；value为空，返回形参封装的Optional	JDK 9
Stream<T> stream()	value非空，返回仅包含此value的Stream；否则，返回一个空的Stream	JDK 9
T orElseThrow()	value非空，返回value；否则抛异常 NoSuchElementException	JDK 10

### 3. 局部变量类型推断的升级

```
//错误的形式：必须要有类型，可以加上var  
Consumer<String> con1 = (@Deprecated t) -> System.out.println(t.toUpperCase());  
//正确的形式：  
// 使用var的好处是在使用Lambda表达式时给参数加上注解。  
Consumer<String> con2 = (@Deprecated var t) -> System.out.println(t.toUpperCase());
```

### 4. HTTP客户端API

- HTTP，用于传输网页的协议，早在1997年就被采用在目前的1.1版本中。直到2015年，HTTP2才成为标准。



- HTTP/1.1和HTTP/2的主要区别是如何在客户端和服务器之间构建和传输数据。HTTP/1.1依赖于请求/响应周期。HTTP/2允许服务器“push”数据：它可以发送比客户端请求更多的数据。这使得它可以优先处理并发送对于首先加载网页至关重要的数据。
- 这是Java 9开始引入的一个处理HTTP请求的HTTP Client API，该API支持同步和异步，而在Java 11中已经为正式可用状态，你可以在java.net包中找到这个API。
- 它将替代仅适用于blocking模式的HttpURLConnection（HttpURLConnection是在HTTP 1.0的时代创建的，并使用了协议无关的方法），并提供对WebSocket和HTTP/2的支持。

## 同步与异步代码

```

HttpClient client = HttpClient.newHttpClient();
HttpRequest request =
    HttpRequest.newBuilder(URI.create("http://127.0.0.1:8080/test/")).build();
BodyHandler<String> responseBodyHandler = BodyHandlers.ofString();
HttpResponse<String> response = client.send(request, responseBodyHandler);
String body = response.body();
System.out.println(body);

HttpClient client = HttpClient.newHttpClient();
HttpRequest request =
    HttpRequest.newBuilder(URI.create("http://127.0.0.1:8080/test/")).build();
BodyHandler<String> responseBodyHandler = BodyHandlers.ofString();
CompletableFuture<HttpResponse<String>> sendAsync =
    client.sendAsync(request, responseBodyHandler);
sendAsync.thenApply(t -> t.body()).thenAccept(System.out::println);
//HttpResponse<String> response = sendAsync.get();
//String body = response.body();
//System.out.println(body);

```

## 5.简化的编译运行程序

看下面的代码。

```

// 编译
javac Javastack.java
// 运行
java Javastack

```

在我们的认知里面，要运行一个Java源代码必须先编译，再运行，两步执行动作。而在未来的Java 11版本中，通过一个java命令就直接搞定了，如以下所示：

**java Javastack.java**

一个命令编译运行源代码的注意点：

- 执行源文件中的第一个类，第一个类必须包含主方法。
- 并且不可以使用其它源文件中的自定义类，本文件中的自定义类是可以使用的。

## 6.

废除Nashorn javascript引擎，在后续版本准备移除掉，有需要的可以考虑使用GraalVM。

## 7.ZGC

- GC是java主要优势之一。然而，当GC停顿太长，就会开始影响应用的响应时间。消除或者减少GC停顿时长，java将对更广泛的应用场景是一个更有吸引力的平台。此外，现代系统中可用内存不断增长，用户和程序员希望JVM能够以高效的方式充分利用这些内存，并且无需长时间的GC暂停时间。

- ZGC, A Scalable Low-Latency Garbage Collector(Experimental)

ZGC，这应该是JDK11最为瞩目的特性，没有之一。但是后面带了Experimental，说明这还不建议用到生产环境。

- ZGC是一个并发，基于region，压缩型的垃圾收集器，只有root扫描阶段会STW(stop the world)，因此GC停顿时间不会随着堆的增长和存活对象的增长而变长。

- 优势：

- GC暂停时间不会超过10ms
- 既能处理几百兆的小堆，也能处理几个T的大堆(OMG)
- 和G1相比，应用吞吐能力不会下降超过15%
- 为未来的GC功能和利用colord指针以及Load barriers优化奠定基础
- 初始只支持64位系统

- ZGC的设计目标是：支持TB级内存容量，暂停时间低(<10ms)，对整个程序吞吐量的影响小于15%。将来还可以扩展实现机制，以支持不少令人兴奋的功能，例如多层堆（即热对象置于DRAM和冷对象置于NVMe闪存），或压缩堆。

## 8.其他

- Unicode 10
- Deprecate the Pack200 Tools and API
- 新的Epsilon垃圾收集器
- 完全支持Linux容器（包括Docker）
- 支持G1上的并行完全垃圾收集
- 最新的HTTPS安全协议TLS 1.3
- Java Flight Recorder

# 在当前JDK中看不到什么？

## 一个标准化和轻量级的JSON API

一个标准化和轻量级的JSON API被许多Java开发人员所青睐。但是由于资金问题无法在Java当前版本中见到，但并不会削减掉。Java平台首席架构师Mark Reinhold在JDK 9邮件列中说：“这个JEP将是平台上的一个有用的补充，但是在计划中，它并不像Oracle资助的其他功能那么重要，可能会重新考虑JDK 10或更高版本中实现。”

- 随着云计算和AI等技术浪潮，当前的计算模式和场景正在发生翻天覆地的变化，不仅对Java的发展速度提出了更高要求，也深刻影响着Java技术的发展方向。传统的大型企业或互联网应用，正在被云端、容器化应用、模块化的微服务甚至是函数(FaaS, Function-as-a-Service)所替代。
- Java虽然标榜面向对象编程，却毫不顾忌的加入面向接口编程思想，又扯出匿名对象之概念，每增加一个新的东西，对Java的根本所在的面向对象思想的一次冲击。反观Python，抓住面向对象的本质，又能在函数编程思想方面游刃有余。Java对标C/C++，以抛掉内存管理为卖点，却又陷入了JVM优化的噩梦。选择比努力更重要，选择Java的人更需要对它有更清晰的认识。
- Java需要在新的计算场景下，改进开发效率。这话说的有点笼统，我谈一些自己的体会，Java代码虽然进行了一些类型推断等改进，更易用的集合API等，但仍然给开发者留下了过于刻板、形式主义的印象，这是一个长期的改进方向。

# 思维导图xmind文件

2021年9月3日 22:50



Java语言基础  
编程