1.java语言概述

1.1高级语言运行机制

- 计算高级语言按执行分为：解释型语言和编译型语言。python属于解释型语言，java属于编译型语言
- 解释型就是逐行解释成特定平台的机器码并立即执行；编译型就是使用专门编译器，根据特定平台将源码一次性翻译成机器能够读懂的机器码，并包装成该平台可执行的格式并执行

1.2 java语言概述

- sun公司：美国斯坦福大学在1995年推出的高级编程语言
- java之父：詹姆斯 高斯林（james Gosling）
- 被Oracle收购，JDK：OracleJDK，OpenJDK

1.3 Java语言特点及JDK和JRE关系

- 开源、跨平台、面向对象、安全性、多线程
- JDK（Java Development kit） 包含JRE（java runtime environment）和开发工具（编译工具javac 和执行工具java）
- JRE包含JVM（java虚拟机）和核心类库（异常处理，数据类型string Long Boolean，常用工具字符串处理类）

2.java核心基础

2.1注释

- java中注释的分类
  1. 单行注释 //
  2. 多行注释 /*注释的内容*/
  3. 文档注释 /** 文档注释的内容*/

2.2关键字

- 关键字：在java中，具有特定含义的单词，小写

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

image-20220224145612548

2.3标识符

- 是创建类、接口、方法、变量、常量等的规则
- 命名规则
  1. 由52个字符（26小写，26大写），0~9数字，下划线_，$美元符号
  2. 不能以数字开头
  3. 标识符不能以java关键字重名
  4. 尽量见名知意
- 命名规范（约定），类、接口、方法、变量、常量
  1. 类和接口是呀大驼峰命名法，如果遇到多个单词，每个单词首字母大写
  2. 变量，使用小驼峰命名法，第一个单词首字母小写，其他单词首字母大写
  3. 常量，每个单词都大写，单词之间使用_分割
  4. 包，全部小写，包的层级之间使用. ,包就是带有层级的文件夹

2.4变量和数据类型

- 变量的定义方式，两种方式：
  1. 先声明后使用
  2. 声明变量并赋值
- 在程序执行的过程中，其值会在数据的范围之内发生变化。Java是强类型语言，值需要指定数据范围
- 数据类型
  1. 基础数据类型
     a. byte 字节
     b. short 二个字节
     c. int 四个字节
     d. long 八个字节
     e. float 四个字节，单精度浮点类型
     f. double 八个字节，双精度浮点类型
     g. char 字符类型
     h. boolean 布尔类型

2. 基本数据类型的默认值
    a. 基本数据类型 char short int long double float 默认值为0
    b. boolean类型默认是false
3. 引用数据类型
    a. 数组
    b. 类
    c. 接口
4. 变量的进阶
    a. 动态代码块和静态代码块和值的作用域

2.5数据类型转换

- 不同数据类型之间会进行计算，这些数据类型的取值范围不一样，直接进行计算会造成数据的损失，这种时候就需要进行一种类型到另一种数据类型的转换
- 常用数据类型转换分类
    1. 隐身（自动）转换
        a. 将小类型的数据自动转换成大类型的数字
        b. 自动转换
    2. 显示（强制）转换
        a. 将大类型的数字强制转换成小类型的数值
        b. 小类型 变量=（小类型强转）（待强转的大类型）
- 显示（强制）类型转换-- 小类型 变量=（小类型）（待转的数值）

```
<br class="Apple-interchange-newline"><div></div>

public class DataTypeDemo03 {
    public static void main(String[] args) {
        short s = 11;
        //将大的数据类型转换成小的数据类型，需要强转 (小类型)
        s = (short) (s + 3);
        System.out.println(s);
    }
}
```

2.6运算符

- 用于连接常量或者变量的符号
- 比如
    1. 算数运算符：+ - / * ++（自增） --（自减）

2. 逻辑运算符：|| && ！ 【短路运算】
3. 比较运算符：> < >= <=
4. 赋值运算符：= += -= *= /=
5. 三元运算符（表达式）？（表达式1）：（表达式2）【if...else... 简写】

2.7键盘录入-输入

- Scanner类
- 开发步骤

```
import java.util.Scanner
public class 文件名{
    public static void main(String[] args){
        Scanner 变量名=new Scanner(System.in)
    }
}
```

2.8流程控制

- 程序执行的顺序：分为三大类：顺序结构、选择结构、循环结构
- 顺序结构
  - 程序从上到下，从左到右依次执行
- 选择结构
  - if 语句
    - 单分支

```
if(判断条件){
    语句块1;
}
```

    - 双分支

```
if(判断条件){
    语句块1;
}else{
    语句块2;
}
```

    - 多分支

```
if(判断条件){
    语句块1;
```

```
}else if(判断条件2){
    语句块2;
}else{
    语句块3;
}
```

- 选择结构：switch语句
  - 多分支判断语句，switch语句

```
switch(变量){
    case 值1: 表达式1;
              break;
    case 值2: 表达式2;
              break;
    case 值3: 表达式3;
              break;
    default:
        表达式n;
}
```

  - case穿透
    - 就是在表达式中添加break;
- 循环结构
  - 循环结构，某个代码块，进行多次循环执行，由初始化条件、判断条件、控制条件、循环体四部分组成
  - 循环结构分类
    1. for循环语句

```
# 格式
 for(初始化条件;判断条件;控制条件){
     循环体;
 }
```

    2. whlie语句

```
# 格式
初始化条件
while(判断条件){
        循环体;
        控制条件
}
```

    3. do...while语句

```
# 格式
 初始化条件
 do{
    循环体;
        控制条件;
 }while(判断条件);
```

4. 区别
    a. for循环，在循环体外不能使用到初始化变量；while和do...while可以使用到初始化条件
    b. for循环使用固定次数的循环；while和do...while一般情况下使用不固定的循环

- 死循环
  - 程序代码块持续执行，这种循环必定循环（死循环）
  - 死循环分为三类
    - for（；；）
    - while（true）{}（*）
    - do{}while（true）；
- 循环跳转之break或continue
  - 循环跳转，在满足一定条件时，直接跳出循环或跳过档次循环
  - 循环跳转分类
    - break：跳出循环
    - continue：跳过档次循环
- 循环嵌套
  - 嵌套循环，就是在一个循环语句中包含另一个循环语句。外层循环执行一次，内层循环执行一圈

2.9数组

- 将相同的数据类型存储到容器中使用数组
- 格式

1. 动态初始化

```
# 数组动态初始化格式
  数据类型[] 数组名 = new 数据类型[10];  (*)
  数据类型 数组名[] = new 数据类型[10];
```

2. 静态初始化

```
# 数组静态初始化格式
  数据类型[] 数组名 = {1,2,3,4,5,6}; (*)
  数据类型[] 数组名 = new 数据类型[]{1,2,3,4,5,6};
```

3. 描述：

a. 数据类型：当前数组的值的数据类型

b. [] :代表当前是一个数组

c. new数据类型[10]： 实例化一个数组对象

d. 10：代表数组的长度为10，数组的长度的固定的

e. 数组名：当前数组的标识符，遵循小驼峰命名法，多个单词，第一个单词，从第二个单词首字母大写

- 数组的特点
  1. 数组赋值引用类型，得到地址
  2. 数组类型都有下标（索引），索引从0开始
  3. 数组实例化，对数组的值都有默认值
     a. byte、short、int、long默认值为0
     b. float和double 默认值是0.0
     c. boolean默认值是false
- 数组的基本用法
  - 获取数组指定索引的元素
    - 数组名称[下标]：当前下标所在的位置对用的值
  - 修改数组定制的元素值
    - 数组名称[下标]= 值 ： 将值赋值给当前数组对应的下标位置
  - 获取数组的长度
    - 数组名称.length :获取当前数组的长度
- 数组的内存图（引用类型）

数组的内存原理图

```
public class ArrayDemo02 {
    public static void main(String[] args) {
        //1. 定义一个长度为5的int类型的数组.
        int[] arr = new int[5];
        //2. 打印数组中的第3个元素.
        System.out.println(arr[2]);
        //3. 设置数组中的第一个元素值为11.
        arr[0] = 11;
        //4. 获取数组中的第一个元素值, 并将其赋值给变量a, 然后打印
        int a = arr[0];
        System.out.println(a);
        //5. 打印数组的长度.
        System.out.println(arr.length);
    }
}
1.通过 javac 将ArrayDemo02.java编译成为 ArrayDemo02.class
文件保存到内存的方法区;
2.调用 main 方法进栈，从上往下，从左往右依次执行代码
3.局部变量 int[] arr 等待接收堆中分配的内存的首地址
4.堆中根据数组的大小分配数组空间长度为 5 ，将数组
初始化为 0
5.将arr 指向数组空间的首地址
6.打印输出index为2的位置数据
7.数组的第一个元素进行赋值为11
8.将数组的值赋值给局部变量 a
9.打印输出 a 的值
10.获取数组的长度
```

内存图解： 内存原理图了解即可。

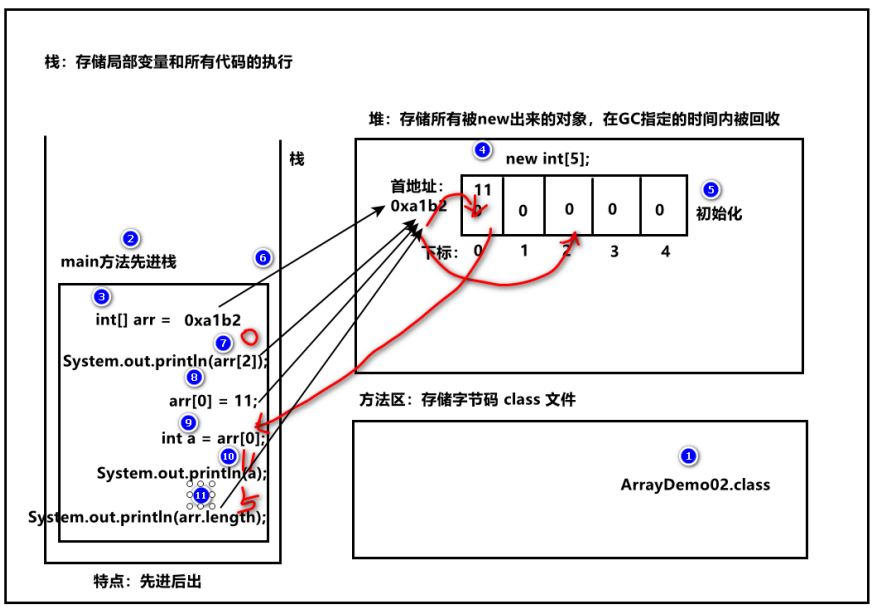


image-20220225154147211

- 内存分为5个部分
  1. 栈： 先进后出
  2. 堆：new对象
  3. 方法区：二进制字节码文件

4. 寄存器
5. 本地方法区

- 数组常见的问题
  - 数组越界问题
    - int[] arr =new int[3];
    - arr[3] 报数组越界，数组索引从0开始
  - 数组空指针问题null，
    - 实际开发中，对数组进行赋值，值为空，遍历读取数组的值就会出现空指针异常
    - arr[2]=speed #因为赋值的speed为空，造成遍历数组的时候出现空指针异常

2.10方法的介绍

- 方法的概念
  - 具有独立功能的代码块，类型与python中的函数
- 方法的定义格式

```
修饰符  返回类型  方法名称(参数类型 参数名1,参数类型 参数名2...){

        代码块;
    return 语句;
}
```

  - 修饰符：
    - public - 公共，任意包和类都能访问
    - protected - 保护，继承父类使用
    - private - 私有 ， 只能在当前类中访问
  - 返回类型：当前方法的返回的数据类型，void - 当前返回的是空类型
  - 方法名称：当前方法的标识符 - 小驼峰命名法
  - 参数类型 参数列表：方法的输入参数的列表，0个或者多个
  - 代码块：独立实现的功能
- 方法的注意事项
  1. 方法与方法之间是平级关系，不能嵌套定义
  2. 方法必须先创建才可以使用，该过程称为：方法定义
  3. 方法自身不会直接运行，而是需要我们手动调用方法后，它才会执行，该过程称为方法调用
  4. 方法的功能越单一越好
  5. 定义方法的时候写在参数列表中的参数，都是：形参
     a. 形参：形容调用方法的时候，需要传入什么类型的参数
  6. 调用方法的时候，传入的具体的值（变量或者常量都可以），叫实参
     a. 实参：调用方法时，实际参与运算的数据

- 方法的重载
  - 方法重载就是在类中，不同方法，方法名称相同，参数列表不同，这样方法成为重载方法
  - 参数列表不同①参数的个数不同，②参数的类型不同
  - 应用场景
    - 不同数据类型的数值计算
    - 不同集合之间数据计算（Set int[] List Map Stack）

2.11面向对象

- 面向对象的三大特征
  - 封装性
    - 私有privaate字段，方法，外部不能直接访问
  - 继承性
    - extends，子类继承父类的共有的方法
  - 多态性
    - 在不同的场景下，形态是不同的
- 面向对象
  - 类和对象详解
    - 类：是属性和行为的集合，是一个抽象的概念
    - 对象：对类的具体的实现
    - 属性：也叫字段，==成员方法==，就是事务的描述信息，在类内方法外成员变量，在方法内，局部变量
    - 行为: ==成员方法==，指的就是事物能够做什么，不能是static，静态方法的调用 类名.方法名()
    - 成员（成员变量，成员方法）
    - 接口和抽象类，抽象方法，继承，实现

```
package sz.base.java.oom;
public class Phone {
    //使用private 修饰成员变量，将这些成员变量隐藏起来
    //这就是封装性，只能在内部访问
    private String brand;
    public double price;
    public String color;
    //构造方法的目的，就是实现成员变量的赋值
    //无参构造方法
    //构造器的定义格式
    //1.构造方法名称必须和类名完全一致
    //2.构造方法没有返回值类型
    //3.构造方法没有返回值，可以写return,一般情况下不写
```

```
    //alt +insert快捷键
    public Phone() { }
    //this: 读取当前对象的成员变量 this.变量 来获取
    public Phone(String brand, double price, String color) {
        this.brand = brand;
        this.price = price;
        this.color = color; }
    //get set 方法就是提供队成员变量的获取和修改的方法
    public String getBrand() {
        return brand;
    }
    public void setBrand(String brand) {
        this.brand = brand;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public void call() {
        System.out.println("打电话");
    }
    public void sendMessage() {
        System.out.println("发短信");
    }
}
```

- **封装**
  - 概念：将对象的成员变量和成员方法隐藏起来，不允许外部程序直接访问对象内部信息，而是通过该类所提供的方法来实现对内部信息的操作和访问
  - 封装的好处
    - 隐藏类的实现细节
    - 让程序使用者只能通过读写方法（比如getter或者setter方法）来访问数据和写入数据，限制对成员变量

的不合理访问

- 便于修改，提高代码的可维护性

- 封装的特性
  - private关键字
    - private关键字来限定当前的字段是否能被访问
    - 如果需要读取指定字段的值，可以使用getxxx setxxx方法
  - this关键字--相当于python的self
    - java中变量遵循就近原则
    - 读取当前对象的成员变量 this.变量 来获取
    - 不加this：如果方法内没有变量，则获取的就是外部的变量属性

```java
package dz.base.java.oop;

public class Student {
    //成员变量
    private String name = "zhangsan";

    public void show() {
        //方法内的变量叫做 局部变量
        String name = "wangwu";
        //就近原则，读取局部变量的值
        System.out.println(name);
    }

    public static void main(String[] args) {
        //s1 是 student类一个对象
        Student s1=new Student();
        //s1 对象调用show（）方法
        s1.show();
    }
}

```

  - 构造方法
    - 构造器概念：
      - 类似于python的__init__()，在这个构造方法中成员变量赋值
      - 比如使用private修饰成员变量，将这些成员变量隐藏起来
    - 构造方法名和类名完全一致（包括大小写）
    - 构造方法没有返回值，连void都不能写

    - 注意事项
        - 默认每个类都有一个无参构建
        - 如果用户自定义了一个构造方法，需要用户提供一个无参构造

- **继承**
    - 将多个类存在相同属性和行为单独抽取到一个类中，这些类想调用这些属性和行为只需要调用那个类即可，这种关系就叫继承
    - 特点
        - 继承关系只能单继承，每个类最多只能有一个直接父类，可以多层继承
        - java.lang.Object类是所有类的父类，所以所有对象都能调用java.lang.Object的示例方法
        - 格式

```
public class SuperClass{ 成员变量; 成员方法 }
public class SubClass extends SuperClass{ 成员变量; 成员方法 }
# SubClass 继承于 SuperClass
# SubClass 是 SuperClass 子类
# SuperClass 是父类， SubClass 是子类
```

- 向上转换格式

```
父类类型 对象  = new 子类类型();
对象.父类的方法() ，指向是子类的实现
```

- 向下转换格式
    - 父类拿不到子类自身的增强的方法，通过将父类强转成子类对象，调用子类的特有的方法

```
父类类型 对象  = new 子类类型();
子类类型 对象2 = (子类类型)对象;
对象2.子类的特有的方法();
```

- 好处
    1. 提高了代码的复用性
    2. 提高了代码的可维护性
    3. 让类与类之间产生关系，是多态的前提，多态需要类之间的关系，类之间的关系extends和implements关系
- 缺点
    1. 增强了类与类之间的耦合性
    2. 开发原则，高内聚、低耦合
    3. 耦合：类与类之间的交叉关系；内聚：类独立完成工作的能力
- 继承中的成员特点
    - java中使用变量遵循就近原则，局部位置有就使用，没有就去本类的成员位置找，有就使用，没有就去

父类的成员位置找，有就使用，没有就报错。

1. 成员变量：就近原则
2. 成员方法：就近原则
3. 构造方法：子空参访问父空参，子全参访问父全参

- 方法重写
  - 子类总是以父类为基础，额外增加新的成员变量和方法，某些时候，子类需要重写父类的方法
  - 子类继承父类，如果子类中的方法（方法名、方法参数、返回类型）完全一样，方法的重写
  - 子类具有自己特征和功能，父类的方法进行重写
  - 注意事项
    - 子类重写父类的方法需要@override复写
    - 父类中的私有方法和变量不能被重写
    - 子类继承父类，对父类方法重写，调用的方法时子类的实现

- **多态**
  - 概念：同一个事物（或者对象）在不同时刻表现出来的不同状态，实际上多态就是父类的引用指向子类的对象。编译时类型由生命该变量的数据类型决定，运行时有实际赋给该变量的对象决定。编译和运行时类型不一样，就可能出现多态。
  - 使用多态的前提
    - 要有继承关系
    - 要有方法的重写
    - 有要父类的引用子对象

```java
package dz.base.java.doutai;

public class WaterTest {
    public static void main(String[] args) {
        //液态水
        Water water = new Water();
        water.status();

        //冰
        Water ice = new Ice();
        ice.status();

        //气态
        Water steam = new Steam();
        steam.status();
    }
}
```

- **两个关键字**
  - final
    - final修饰类，说明当前不能被继承
    - final修饰方法，说明当前方法不能被重写
    - final修饰变量，说明当前变量只能赋值一次
  - static
    - static可以修饰类的成员变量和成员方法，成员变量和成员方法，谁new是谁的，static修饰类.变量（方法）
      1. 随着类的加载而加载--不会new的时候才加载，创建的时候就会被保存
      2. 优先于对象存在
      3. 被static修饰的内容，能被该类下所有的对象共享：这也是我们判断是否使用静态关键字的条件
      4. 可以通过类名.的形式调用
    - 访问特点
      - 静态方法只能访问静态的成员变量和静态的成员方法（静态只能访问静态）
    - 注意事项
      - 在静态方法中，是没有this，super关键字的
      - 因为静态的内容是随着类的加载而加载，而this和super是随着对象的创建而存在（先进内存的，不能访问后进内存的）
    - 总结
      - 1.成员方法既可以访问静态变量static，也可以访问成员变量（不带static）
      - 2.静态方法中只能调用静态变量，不能调用成员变量；
      - 3.成员方法中既可以调用成员方法，也可以调用静态方法
      - 4.静态方法中只能调用静态方法，不能调用成员方法
- **抽象类**
  - 概念：一个没有方法体的方法被定义为抽象方法，而类中如果抽象方法，该类就是抽象类
  - 抽象类必须使用abstract关键字进行定义
  - 抽象类中有抽象方法，抽象方法没有定义

```
package dz.base.java.oop.abstracts;
/**
 * 应用场景：
 * 1.约定要做的事情，并没有具体怎么做，需要使用抽象方法。
 * 2.抽象类中，可能有一部分方法是没有实现的，需要继承之后再去重写实现
 * //抽象类
 * public abstract class 类名{
```

```
 *      //抽象方法
 *      public abstract 返回类型 方法名称（参数列表）;
 * }
 * 注意事项：
 * 1.抽象类不能被实例化，不能被new
 * 2.抽象类中不一定有抽象方法，如果有抽象方法一定是抽象类
 * 3.抽象类的子类
 *  3.1既可以是抽象类
 *  3.2也可以是普通类
 *
 * 可以被继承重写抽象方法，不能实例化
 * 当成员没有方法体的时候，这个方法叫做抽象方法，需要被abstract修饰
 * 当一个类中有抽象方法时，这个类就叫做抽象类，需要被abstract修饰
 */
public abstract class Animal {
    public abstract void eat();
    public void run(){
        System.out.println("动物会跑");
    }
}
```

- **接口**
  - 比抽象类更加抽象，接口中只有有抽象方法和常量（固定值），接口中没有实现的方法。
  - 接口应用场景--扩展类的功能（成员方法）

```
public interface Creature {
    public void eat();

    public void breath();
}
```

  - 使用implements来实现接口，接口中方法必须都被实现，认为接口中方法时实现类的约定。

```
public class SunFlow implements Creature {
    @Override
    public void eat() {
        System.out.println("光合作用");
    }

    @Override
    public void breath() {
```

```
        System.out.println("吸收氧气");
    }

    public void start(){

    }
}
```

- 注意事项
  - 抽象类和接口区别：
    1. 抽象类需要使用extends实现，接口使用implements
    2. 抽象类继承只能继承（只能继承一个父类），接口可以多实现（可以实现多个接口）
    3. 接口实现必须全部方法实现，抽象类可以部分实现
- 类和接口之间的关系
  - 类与类之间：继承关系，只能单继承，不能多继承，但是可以多层继承
  - 类与接口之间：实现关系，可以单实现，因为可以多实现，还可以在继承一个类的同时实现多个接口
  - 接口与接口之间：继承关系，可以单继承，也可以多继承

内部类

- 内部类的概念
  - 成员内部类-定义在成员位置的内部类，一般使用来对类的功能延伸，多用与底层源码
  - 局部内部类-定义在局部位置的内部类，常用于匿名内部类
  - 对于某个类独有的功能，可以使用成员内部类

```
/**
 * Author itcast
 * Date 2021/10/10 11:13
 * Desc TODO
 */
public class AnimalClass {

    //成员内部类
    public class Feature{  //成员内部类
        //当前AnimalClass增强功能
        // 飞
        private String fly;
        // 观赏功能
        private String view;
    }
```

```
16
17      public void method(){
18          class Value{  //局部内部类
19              private String price;
20          }
21      }
22  }
23
```

- **匿名内部类**
  - 指的是没有名字的局部内部类
  - 格式

```
1  new 类名或接口名{
2      @Override
3      //重写类中所有的抽象方法
4  }
```

  - 本质==匿名内部类就是一个子类对象
  - 使用场景
    - 1.当对成员方法之调用一次，如果多次就创建子类
    - 可以作为方法的实参进行传递
    - 案例-实现Animal抽象类，实现抽象类，打印输出实现类的方法

```
1  public class AnimalTest01 {
2      public static void main(String[] args) {
3          //可以通过多态来获取猫类中的 eat 方法
4          //Animal cat = new Cat();
5          //通过匿名内部类实现
6          //本质—匿名内部类就是一个 子类对象
7          new Animal() {
8              @Override
9              public void eat() {
10                 System.out.println("猫吃鱼");
11             }
12         };
13         //
14         System.out.println("-------------------------");
15        // print(cat);
16         System.out.println("-------------------------");
```

```
17          print(new Animal() {
18              @Override
19              public void eat() {
20                  System.out.println("猫吃鱼");
21              }
22          });
23
24          //如何使用
25          print(new Animal() {
26              @Override
27              public void eat() {
28                  //实现需要抽象方法
29              }
30          });
31      }
32      private static void print(Animal animal){
33          animal.eat();
34      }
35  }
```

- **API**
  - 应用程序接口，查看API文档，类似python的pip安装的扩展
  - 常见API
    - Object类
      - Object类是所有类的超类，父类

Modifier and Type	Method and Description
protected Object	clone() 创建并返回此对象的副本。
boolean	equals(Object obj) 指示一些其他对象是否等于此。 **判断两个对象是否相等**
protected void	finalize() 当垃圾收集确定不再有对该对象的引用时，垃圾收集器在对象上调用该对象。
类<?>	getClass() 返回此 Object的运行时类。
int	hashCode() 返回对象的哈希码值。 **获取 hash值**
void	notify() 唤醒正在等待对象监视器的单个线程。
void	notifyAll() 唤醒正在等待对象监视器的所有线程。
String	toString() 返回对象的字符串表示形式。 **将对象转换成字符串**
void	wait() 导致当前线程等待，直到另一个线程调用该对象的 notify()方法或 notifyAll()方法。
void	wait(long timeout) 导致当前线程等待，直到另一个线程调用 notify()方法或该对象的 notifyAll()方法，或者指定的时间已过。
void	wait(long timeout, int nanos) 导致当前线程等待，直到另一个线程调用该对象的 notify()方法或 notifyAll()方法，或者某些其他线程中断当前线程，或一定量的实时时间。

image-20211009161515374

- 案例

```java
public class ObjectTest {
    public static void main(String[] args) {
        /*String zs1 = new String("张三");
        //定义了两个对象
        String zs2 = new String("张三");

        //zs1 地址 zs2第二个对象的地址，两个地址是不同的
        System.out.println(zs1==zs2);  // false
        //当判断两个字符串是否相同，equals 判断两个字符串是否相等
        //将当前的字符串转换成 字符数组，然后一一比较字符是否相等，如果都相等，字符串相等。
        System.out.println(zs1.equals(zs2));*/  // true

        //直接指向常量池
        String zs1 = "张三";
        String zs2 = "张三";
        System.out.println(zs1 == zs2);  // true
    }
}
```

- String类
  - 字符串类型
  - equal方法-将字符串转换成字符数组，一个一个字符比较，如果长度相等，每个字符都相等，认为当前两个字符串相等（只会判断字符是否相等并非引用对象）
  - 翻转案例

```java
package sz.base.java.api;

import java.util.Scanner;

//将用户输入的字符串反转
public class StingReverse {
    public static void main(String[] args) {
        //1.从控制台输入一个字符串
        System.out.println("请输入一个字符串: ");
        Scanner sc = new Scanner(System.in);
        //2.编写一个字符串反转的方法
        String next = sc.next();
        //3.调用反转字符串方法，并打印输出
```

```
        String reverse = reverse(next);
        System.out.println(reverse);

        reverse(next, new StringBuilder());
        System.out.println(reverse);
    }

    /**
     * 输入abc返回cba
     *
     * @param str
     * @return
     */
    public static String reverse(String str) {
        //将字符串转换成字符数组
        char[] chars = str.toCharArray();
        //定义一个字符串用于拼接字符
        String reverseStr = "";
        //倒序显示每个字符，将其拼接成字符串
        for (int i = chars.length - 1; i >= 0; i--) {
            reverseStr += chars[i];

        }
        //返回字符串
        return reverseStr;
    }

    //互为重载方法 stringBuilder里面是一个数组
    public static String reverse(String str, StringBuilder builder) {
        StringBuilder stringBuilder = builder.append(str);
        //StringBuilder 有一个反转工具
        return stringBuilder.reverse().toString();
    }
}

```

- StringBuilder类
  - 用于拼接、增加、删除、插入字符、数字、字符串等操作工具类
  - 案例

```java
package sz.base.java.api;

/**
 * 创建一个StringBuilder 内容abc
 * 在abc追加cde
 * 在b后面在插入一个|
 * 打印输出结果
 */
public class StringBuilderDemo {
    public static void main(String[] args) {
        //1.创建StringBuilder
        StringBuilder builder = new StringBuilder();
        //2.将abc插入到StringBuilder 中
        builder.append("abc");
        //3.将cde插入到abc 中
        builder.append("cde");
        //4.将b后插入| insert
        builder.insert(2, '|');
        //5.转换成字符串并打印输出
        String s = builder.toString();
        System.out.println(s);
    }
}

```

- 数组元素排序
  - 数组工具类-Arrays
  - 案例

```java
package sz.base.java.api;

import java.util.Arrays;
//对数组进行排序
public class ArraysDemo {
    public static void main(String[] args) {
        int[] arr = {26, 45, 23, 64, 123};
        Arrays.sort(arr);
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
```

```
11
12          }
13      }
14  }
15
```

- 包装类
  - 为了对基本类型进行更多更方便的操作，Java就针对每一种基本类型提供了一个对应的引用类型，这就是包装类
  - 八个基本数据类型
    - byte、short、int、long、float、double、char、boolean
  - 对基本数据类型进行封装，封装成类，引用数据类型，这个类就叫做包装类
    - Byte、Short、Integer、Long、Float、Double、Character、Boolean
  - 案例

```java
package sz.base.java.api;

public class PackageClazz {
    public static void main(String[] args) {
        //将int类型转换成包装类
        int a = 10;
        Integer integer = new Integer(a);
        System.out.println(integer);
        //将数字字符串转换成数字类型
        Integer integer1 = new Integer("10");
        //获取int类型的最大值和最小值
        int maxValue = Integer.MAX_VALUE;
        int minValue = Integer.MIN_VALUE;
        System.out.println("maxValue: " + maxValue + " minValue: " + minValue);

        //将整数类型转换字符串类型
        String s1 = 10 + "";
        //将Integer包装类转换成int类型
        int ii = integer1.intValue();
        System.out.println(ii);

        //自动装箱将数字转换成Integer
        Integer i2 = 10;
        i2 += 20;
```

```
        System.out.println(i2);
        //自动拆箱，把Integer转换成int
        int i3=i2;
    }
}

```

- **Date日期类**
  - Date中包含当前时间和时间戳
  - Date日期类实例化，实例化获取当前系统时间的时间戳
  - DateFormat日期格式工具类
  - 案例

```
package sz.base.java.api;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.logging.SimpleFormatter;

/**
 * 需求-
 * 1.获取当前系统的时间，并将其转换成yyyy-MM-dd HH:mm:ss格式
 * 2.指定一个时间戳，打印输出指定的时间
 */
public class DateDemo {
    public static void main(String[] args) {
        //创建格式化对象
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");

        //获取当前的系统时间，通过无参构造
        Date date = new Date();
        //打印输出当前的时间戳
        System.out.println(date.getTime());
        String format = simpleDateFormat.format(date);
        System.out.println(format);
        //设置一个任意的时间戳
        date.setTime(1646030419893L);
        //打印输出这个时间的字符串
        String format1 = simpleDateFormat.format(date);
```

```
        System.out.println(format1);
    }
}

```

- Calendar日历类
  - 计算当前天在一年中哪一天，第几天，在一周中第几天，一个月中第几天，日期进行操作的类
  - 案例

```
/**
 * Author itcast
 * Date 2021/10/10 9:32
 * Desc TODO
 */
public class CalendarDemo3 {
    public static void main(String[] args) {
        //获取 Calendar 的实例
        Calendar instance = Calendar.getInstance();
        //获取当前的年、月、日、在一周某一天
        Date time = instance.getTime();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String dateTimeStr = sdf.format(time);
        System.out.println(dateTimeStr);

        int hour = instance.get(Calendar.HOUR);
        int minute = instance.get(Calendar.MINUTE);
        int day = instance.get(Calendar.DAY_OF_MONTH);
        //一年12个月，从0个月开始，
        int month = instance.get(Calendar.MONTH) + 1;
        System.out.println(hour+":"+minute+" Month:"+ month +" day:"+day);
        //设置当前的时间
        instance.set(2021,10,10,9,42,15);
        //获取设置的值
        Date setTime = instance.getTime();
        System.out.println(sdf.format(setTime));

         System.out.println("请输入年份");
         Scanner date = new Scanner(System.in);
         int date1 = date.nextInt();
```

```
        //创建日历对象
        Calendar c = Calendar.getInstance();
        //设置日期为 年份/03/01号 Calendar#set(int year, int month, int day);  月份: 0-11
        c.set(date1, 02, 01);
        //给日期-1天 把当前时间往前推一天.
        //如果是1则代表的是对年份操作，2是对月份操作，3是对星期操作，5是对日期操作，11是对小时操作，12是对分钟操作，13是对秒操作，14是对毫秒操作
        c.add(5,-1);
        //获取这个月的天数
        int ss = c.get(Calendar.DAY_OF_MONTH);
        System.out.println(ss);
    }
}
```

- 案例2

```
/**
 * 4. 定义工具类DateUtils，该类有两个方法：date2String()，string2Date()，分别用来格式化，解析日期.
 * 在测试类中，测试上述的两个方法.
 */
public class day03_4 {
    //定义一个时间格式
    private static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    /**
     * 实现日期类型转换成日期字符串
     *
     * @param date
     * @return
     */
    public static String date2String(Date date) {
        String format = sdf.format(date);
        return format;
    }
    public static Date string2Date(String dateStr)  throws ParseException{
        Date parse = sdf.parse(dateStr);
        return parse;
    }
    public static void main(String[] args) {
        String s=date2String(new Date());
        System.out.println(s);
```

```
    }
}
```

```
/**
 * 6. 提示用户录入他的出生年月日，计算这个用户一共活了多少天，并将结果打印到控制台上.
 */

public class day03_6 {
    public static void main(String[] args) throws ParseException {
        Scanner sc = new Scanner(System.in);
        //1.获取用户输入的年、月、日
        int year = sc.nextInt();
        Scanner sc1 = new Scanner(System.in);
        int month = sc1.nextInt();
        Scanner sc2 = new Scanner(System.in);
        int day = sc2.nextInt();
        //2.拼接成时间字符串并转换成日期 Date
        String dateStr = year + "-" + month + "-" + day;
        Date date1 = day03_4.string2Date(dateStr);
        //3.获取当前时间的日期 -2 步得到的日期
        //getTime（）获取当前时间戳
        long dates = (new Date().getTime() - date1.getTime()) / 1000 / 3600 / 24;
        //4.将日期的差值/1000/3600/24 得到天
        //5.打印输出天数
        System.out.println(dates);
    }

}
```

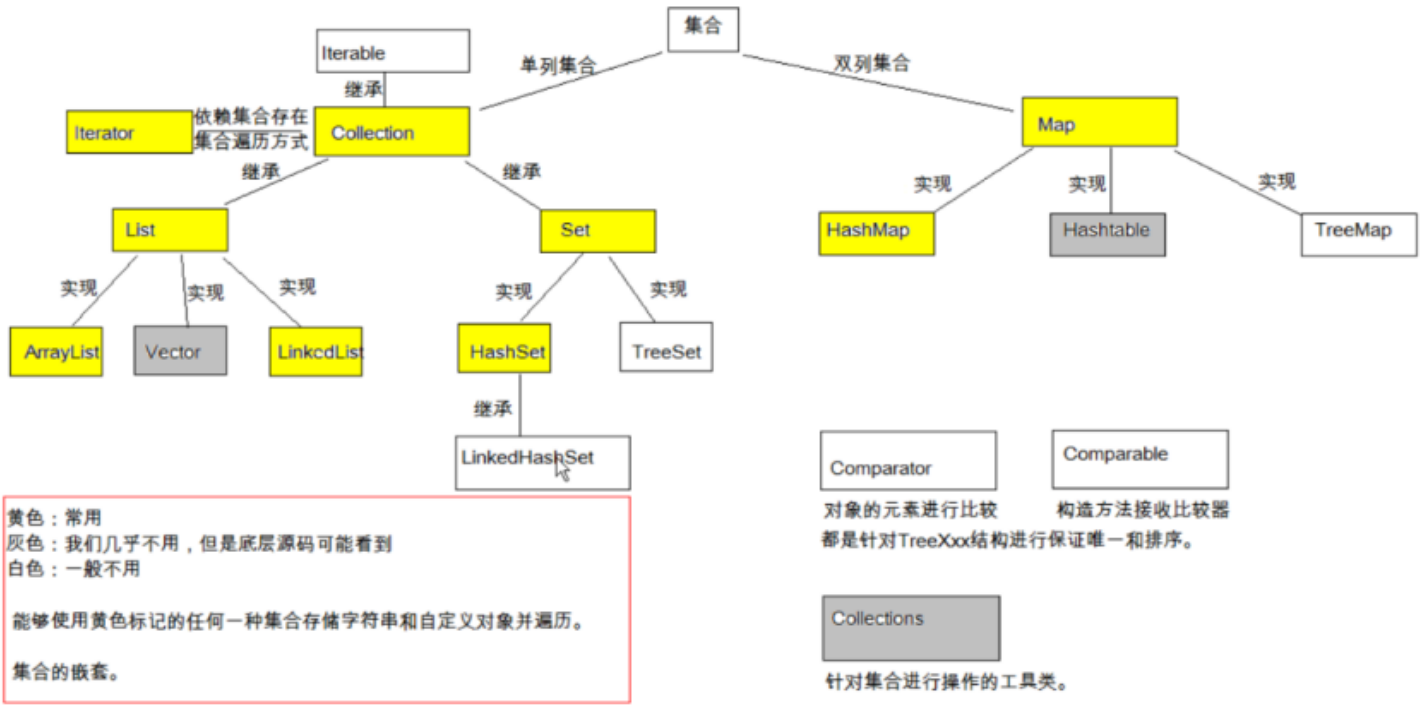
2.12集合

- 集合分为两类：单列集合和双列集合。程序=数据结构+算法
- 从常用集合分类三类：
  - List：单列值，有序，可以重复
  - Set：单列值，无序，不能重复
  - Map：key->value 键值对

- **单列集合概述**
  - 集合中，存储的类型可以不同，存储数据的集合长度是可变的

- 单列集合 顶级接口Collection，需要实现类，常用的类
- ArrayList（可变数组）LinkedList（链表）
- HashSet（哈希表）（无序、不能重复、没有索引）TreeSet（有序集合）

- **Collection集合**
  - 主要用于存储单值的集合，包括List和Set



Vector
Collections
Arrays
AbstractCollection
.2

E：Element 元素类型
类型：引用类型
E 要使用引用类型

```
rface Collection<E> extends Iterable<E> {
y Operations


rns the number of elements in this collection.  If this collec
ains more than <tt>Integer.MAX_VALUE</tt> elements, returns
Integer.MAX_VALUE</tt>.

urn the number of elements in this collection
```

image-20211010145626240

- 案例

```java
package sz.base.java.simple;

import java.util.ArrayList;
import java.util.Collection;

//创建Collection集合，往集合中添加3个字符串，打印输出
public class CollectionDemo {
    public static void main(String[] args) {
        //接口的引用指向子类的实现
        Collection<String> words = new ArrayList();
        //添加3个字符串
        words.add("hello");
        words.add("world");
        words.add("hadoop");
        //打印集合对象
        System.out.println(words);
        //删除元素
        words.remove("hello");
        System.out.println(words);
        //清空集合
/*        words.clear();
        System.out.println(words);*/
        // 是否包含字符，返回true
        boolean hadoop = words.contains("hadoop");
        System.out.println(hadoop);
        //当前是否为空
        boolean empty = words.isEmpty();
        System.out.println(empty);
        //里面有几个元素
        int size = words.size();
        System.out.println(size);
    }
}

```

- 集合的遍历操作

```java
package sz.base.java.simple;
```

```java

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

//遍历迭代出集合中的所有元素，打印出来集合中的每个值
public class IteratorDemo {
    public static void main(String[] args) {
        //接口的引用指向子类的实现
        Collection<String> words = new ArrayList<String>();
        //添加3个字符串
        words.add("hello");
        words.add("world");
        words.add("hadoop");
        //获取集合的迭代器，遍历输出
        Iterator<String> iterator = words.iterator();
        while(iterator.hasNext()){
            String next = iterator.next();
            System.out.println(next);
        }
    }
}
```

- **List集合**
  - List接口是Collection接口的子接口，List接口对Collection进行功能的增强
  - List集合是有序集合，实现List接口都是有序ArrayList LInkedList实现类
  - List集合的元素特点是：有序，可重复，元素有索引
  - 案例--循环遍历List集合中的元素和for增强for循环看下面例子
- **增强for循环**
  - for循环的基础，简化遍历数组和Collection的方式

```java
for(元素的数据类型 元素 : 集合名称){
    System.out.println(元素)
}
```

  - 案例

```java
package sz.base.java.simple;

import java.util.ArrayList;
```

```java
import java.util.Iterator;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        List<Student> student = new ArrayList<>();
        student.add(new Student("zhangsan", 24));
        student.add(new Student("zhangsan", 24));
        student.add(new Student("zhangsan", 24));
        //遍历打印输出List集合中的值
        //遍历1-iterator 迭代输出
        Iterator<Student> iterator = student.iterator();
        while (iterator.hasNext()) {
            Student next = iterator.next();
            System.out.println(next.toString());
        }
        //遍历2-普通的for循环
        //size方法：代表List集合的大小
        for (int i = 0; i < student.size(); i++) {
            Student student1 = student.get(i);
            System.out.println(student1);
        }
        //遍历3-增强for循环遍历打印输出,（数据类型 元素：集合）
        for (Student s : student) {
            System.out.println(s);
        }
    }


    public static class Student {
        //成员变量
        private String name;
        private int age;

        //成员方法
        //构造器
        public Student(String name, int age) {
            this.name = name;
            this.age = age;
        }
```

```
44
45          public String getName() {
46              return name;
47          }
48
49          public void setName(String name) {
50              this.name = name;
51          }
52
53          public int getAge() {
54              return age;
55          }
56
57          public void setAge(int age) {
58              this.age = age;
59          }
60
61          @Override
62          public String toString() {
63              return "Student{" +
64                      "name='" + name + '\'' +
65                      ", age=" + age +
66                      '}';
67          }
68      }
69  }
```

常见的数据结构

- 数据结构是数据处理方式，处理的结构
- 数据结构分类线性表和非线性表
  - 线性表分为数组、链表、集合set等
  - 非线性结构分为树、图、森林
- Java常见数据结构-线性结构
  - 1.栈 stack： 先进后出
  - 2.队列 queue->LinkedList： 先进先出
  - 3.数组 ArrayList： 可变不可变
  - 4.链表 List-> LinkedList

List集合的子类

- List集合常见的子类ArrayList（可变数组）LinkedList（链表）
- Arrays.asList(String对象) 可以把String数组转换成列表
- 两者区别
  - 1.ArrayList可变数组，有索引，能重复，有序；查找快，随机读（get（index）），插入和删除相对慢
  - 2.LinkedList链表，没有索引，能重复，有序；查找相对慢，插入和删除相对快

```
package sz.base.java.simple;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        List<Student> student = new ArrayList<>();
        student.add(new Student("zhangsan", 24));
        student.add(new Student("zhangsan", 24));
        student.add(new Student("zhangsan", 24));
        //遍历打印输出List集合中的值
        //遍历1-iterator 迭代输出
        Iterator<Student> iterator = student.iterator();
        while (iterator.hasNext()) {
            Student next = iterator.next();
            System.out.println(next.toString());
        }
        //遍历2-普通的for循环
        //size方法：代表List集合的大小
        for (int i = 0; i < student.size(); i++) {
            Student student1 = student.get(i);
            System.out.println(student1);
        }
        //遍历3-增强for循环遍历打印输出,（数据类型 元素：集合）
        for (Student s : student) {
            System.out.println(s);
        }
    }

```

```
32
33      public static class Student {
34          //成员变量
35          private String name;
36          private int age;
37
38          //成员方法
39          //构造器
40          public Student(String name, int age) {
41              this.name = name;
42              this.age = age;
43          }
44
45          public String getName() {
46              return name;
47          }
48
49          public void setName(String name) {
50              this.name = name;
51          }
52
53          public int getAge() {
54              return age;
55          }
56
57          public void setAge(int age) {
58              this.age = age;
59          }
60
61          @Override
62          public String toString() {
63              return "Student{" +
64                      "name='" + name + '\'' +
65                      ", age=" + age +
66                      '}';
67          }
68      }
69  }
70
```

Set集合

- Set集合，存储的数据唯一，没有索引，无序
- Set集合的常见实现类，HashSet和TreeSet

```java
package sz.base.java.simple;

import java.util.HashSet;
import java.util.Objects;
import java.util.Set;

public class SetDemo2 {
    public static void main(String[] args) {
        Set<Student> student = new HashSet<>();
        Student changssan = new Student("zhangsaj", 35);
        student.add(changssan);
        Student zhangsan = new Student("zhangsaj", 35);
        student.add(zhangsan);
        for (Student i : student) {
            System.out.println(i);
        }
    }

    public static class Student {
        //成员变量
        private String name;
        private int age;

        //成员方法
        //构造器
        public Student(String name, int age) {
            this.name = name;
            this.age = age;
        }

        public String getName() {
            return name;
        }

```

```
35          public void setName(String name) {
36              this.name = name;
37          }
38
39          public int getAge() {
40              return age;
41          }
42
43          public void setAge(int age) {
44              this.age = age;
45          }
46
47          @Override
48          public boolean equals(Object o) {
49              if (this == o) return true;
50              if (!(o instanceof Student)) return false;
51              Student student = (Student) o;
52              return age == student.age && Objects.equals(name, student.name);
53          }
54
55          @Override
56          public int hashCode() {
57              return Objects.hash(name, age);
58          }
59
60          @Override
61          public String toString() {
62              return "Student{" +
63                      "name='" + name + '\'' +
64                      ", age=" + age +
65                      '}';
66          }
67      }
68
69      private static class set<T> {
70      }
71  }
72
```

2.13 双列集合

- **双列集合的概述（常用方法）**
  - 就是两列键值对key ->value ， 数据结构约束对key，key必须保证唯一，值 可以重复；如果key相同情况下，value被覆盖
  - 双列集合Map集合
  - Map集合分类
    - 1.HashMap基于key值的hash，存储
    - 2.TreeMap基于key值有序的
  - 案例

```
package sz.base.java.map;


import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class HashMapDemo {
    public static void main(String[] args) {
        //1.定义Map集合，键是学号，值是学生的名字，（键值都是字符串类型）
        Map<String, String> hashMap = new HashMap<>();
        //2.往Map集合中添加3对元素
        hashMap.put("100001", "张三");
        hashMap.put("100002", "李四");
        hashMap.put("100003", "王五");
        //3.打印Map集合对象
        System.out.println(hashMap);
        //4 使用entrySet实体打印并输出每个entry
        Iterator<Map.Entry<String, Sring>> interator = hashMap.entrySet().iterator();
        while (interator.hasNext()) {
            Map.Entry<String, String> next = interator.next();
            String key = next.getKey();
            String value = next.getValue();
            System.out.println(key + value);
        }
        //remove 根据指定的key删除entry
```

```
        String remove = hashMap.remove("100001");
        System.out.println(remove);//得到张三而不是学号,张三是value

        //判断集合是否包含指定的键
        boolean b = hashMap.containsKey("100001");
        System.out.println(b);

        //判断集合是否包含指定的值
        boolean b1 = hashMap.containsValue("张三");
        System.out.println(b1);

        //判断集合是否为空
        boolean empty = hashMap.isEmpty();
        System.out.println(empty);

        //集合的长度
        int size = hashMap.size();
        System.out.println(size);

/*          //清空集合
        hashMap.clear();
        System.out.println(hashMap);*/
        //get方法获取,参数是key值
        Set<String> strings = hashMap.keySet();
        for(String key :strings){
            System.out.println(key+hashMap.get(key));
        }
    }

}

```

- **Map集合的获取功能**
  - 遍历打印自定义类的HashMap案例

```
Map<String, ListDemo02.Student> maps = new HashMap<>();
        maps.put("100001",new ListDemo02.Student("zhangsan",20));
        for (String key : maps.keySet()) {
            System.out.println(maps.get(key));
        }
```

- 键值是Student，值是地址String

```java
/**
 * Author itcast
 * Date 2021/10/10 17:58
 * 1.创建HashMap集合，键是学生对象(Student)，值是居住地(String).
 * 2.往HashMap集合中添加3组数据.
 * 3.通过两种方式，遍历HashMap集合.
 * 注意：HashMap集合想保证键的唯一性，依赖hashCode()和equals()这两个方法.
 */
public class MapDemo02 {
    public static void main(String[] args) {
        Map<Student,String> ss = new HashMap<Student,String>();
        ss.put(
                new Student("zhangsan",20),
                "广东省深圳市宝安区"
        );
        ss.put(
                new Student("zhangsan",20),
                "广东省深圳市罗湖区"
        );
        for (Student key : ss.keySet()) {
            System.out.println(ss.get(key));
        }
    }
    public static class Student{
        private String name;
        private int age;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public int getAge() {
            return age;
        }
```

```
39
40          public void setAge(int age) {
41              this.age = age;
42          }
43
44          public Student(String name, int age) {
45              this.name = name;
46              this.age = age;
47          }
48
49          @Override
50          public String toString() {
51              return "Student{" +
52                      "name='" + name + '\'' +
53                      ", age=" + age +
54                      '}';
55          }
56
57          /**
58           * 重写两个方法 equals 和 hashCode两个方法
59           * equals 判断两个对象是否相等
60           *  这两个对象中的成员变量的值是否相等，如果成员变量的值都相等，说明两个对象是相等的。
61           */
62          @Override
63          public boolean equals(Object o) {
64              if (this == o) return true; //this 地址值相同,
65              if (o == null || getClass() != o.getClass()) return false;
66
67              Student student = (Student) o;
68
69              if (age != student.age) return false;
70              return name != null ? name.equals(student.name) : student.name == null;
71          }
72
73          @Override
74          public int hashCode() {
75              int result = name != null ? name.hashCode() : 0;
76              result = 31 * result + age;
77              return result;
78          }
```

```
    }
}
```

- **判断字符串是否存在某个值**
  - 案例

```
第一种
1.可以使用Arrays类中binarySearch(Object[] a, Object key) 方法来查找是否存在某个值，如果某个值存在则返回值>= 0，反之返回值则 < 0
优点：使用二分查找法，效率快捷。
缺点：查询的数组必须是有序的，如果不是有序的话，使用此方法是没有用的。
String[] array = {"1","2","3","4"};
int index = Arrays.binarySearch(array,"2");
System.out.println("index:" + index); //--- index:1
index = Arrays.binarySearch(array,"0");
System.out.println("index:" + index); //--- index:-1
index = Arrays.binarySearch(array,"5");
System.out.println("index:" + index); //--- index:-5

第二种
2.使用Arrays类中asList()方法将数组转化为List()列表，在使用contains()方法判断数组中是否存在某个值
优点：数组可以是乱序的，没有顺序的要求。
缺点：查询效率上可能稍慢，但应该不会影响大局。
String[] array = {"1","2","3","4"};
boolean flag = Arrays.asList(array).contains("2");
System.out.println("flag:" + flag);//--- flag:true


```

- **可变参数**
  - 概念：可变参数又称参数个数可变，它用作方法的形参出现，那么方法参数个数就是可变的了
  - 格式
    - 修饰符 返回值类型 方法名（数据类型... 变量名）{}
    - 案例

```
/**
 * Author itcast
 * Date 2022/2/28 17:25
 * Desc get方法，传入两个参数符号，可变参数，最终得到结果
 */
```

```java
public class VariableParams {
    public static void main(String[] args) {
        int sum = get("+", 100, 1, 2, 3, 4, 5, 6, 7, 8, 9);
        System.out.println(sum);
    }

    public static int get(String symbol,int init, int... arr){
        if (symbol.equalsIgnoreCase("-")){
            for (int i = 0; i < arr.length; i++) {
                init -= arr[i];
            }
        }else if(symbol.equalsIgnoreCase("+")){
            for (int i = 0; i < arr.length; i++) {
                init += arr[i];
            }
        }else{
            System.out.println("没有这个符号的计算");
        }
        return init;
    }
}
```

- 注意事项
  - 这里的变量其实是一个数组
  - 可变参数的底层就是一个数组
  - 如果一个方法有多个参数，其中包含可变参数，可变参数要放在最后
  - 方法的形参列表有且只能有一个可变参数，并且可变参数要放到形参列表的最后

2.14 Lambda表达式

- 概念：Lambda表达式本身就是匿名方法（代码块），就是将方法（代码块）赋值给一个变量，Lambda的类型都是一个接口
- 演化：将方法（代码块）赋值给了一个变量，就是一个Lambda表达式
- 案例：无参Lambda

```java
package lambda;

/**
 * 1.已知接口Animal中有一个抽象方法
 * 2.在测试类AnimalTest中定义show(Animal an)方法，实现调用Animal#eat（）方法
 * 3.并在main方法中，调用AnimalTest#show（）方法
```

```
 */
public  class AnimalTest {
    public static void main(String[] args) {
        AnimalTest animalTest = new AnimalTest();
        //第一种方式是，匿名内部类
        //匿名内部类就是实现接口或者抽象方法，但是没有类名
        AnimalTest.show(new Animal() {
            @Override
            public void eat() {
                System.out.println("猫吃鱼");
            }
        });
        //第二种方式：使用lambda表达式来实现
        //void.eat()
        AnimalTest.show(() -> System.out.println("猫吃老鼠"));
//
    }

    public static void show(Animal an) {
        an.eat();
    }
}

/**
*提前定义一个接口方法为eat（）抽象方法
*/
/**
 * 1.已知接口Animal中有一个抽象方法
 * 2.在测试类AnimalTest中定义show(Animal an)方法，实现调用Animal#eat（）方法
 * 3.并在main方法中，调用AnimalTest#show（）方法
 */
public interface Animal {
    void eat();
}
```

- 详解Lambda表达式
  - 格式：（形式参数）-> {代码块}
  - 解释
    - 形式参数：如果有多个参数，参数之间用逗号隔开，如果没有参数，留空即可
    - ->：由英文中划线和大于符号组成，固定写法。代表指向动作

        - 代码块：是我们具体要做的事情，也就是我们以前写的方法体内容
        - 组成Lambda表达式的三要素：形式参数、箭头、代码块
    - 使用：有一个接口，且接口中有仅有一个抽象方法
    - 省略规则
        - 参数类型可以省略，但是有多个参数的情况下，不能只省略一个
        - 如果参数有且仅有一个，那么小括号可以省略
        - 如果代码块的语句只有一条，可以省略大括号和分号，和return关键字
        - 案例：有参数的Lambda

```
package lambda2;

/**
 * 一个方法时：useFlyable（Flyable f）
 * 一个方法是主方法，在主方法中调用useFlyable方法
 */
public class FlyableDemo {

    public static void main(String[] args) {
        //使用匿名内部类的方法创建
        useFlyable(new Flyable() {
            @Override
            public void fly(String s) {
                System.out.println(s + "行");
            }
        });
        useFlyable((String s) -> {
            System.out.println(s + "行1");
        });

        //使用Lambda表达式来创建
    }

    public static void useFlyable(Flyable f) {
        f.fly("坐直升机飞");
    }
}


//接口
```

```java
/**
 * 定义一个接口（Flyable），里面定义一个抽象方法，void fly（string s）
 * 定义一个测试类
 */
public interface Flyable {
    void fly(String s);
}
```

- 案例，多个参数

```java
package lambda3;

public class AddableDemo {
    public static void main(String[] args) {
        //使用匿名内部类实现累加求和
        useAddable(new Addable() {
            @Override
            public int add(int x, int y) {
                return 2 * x + y;
            }
        }, 10, 20);
        //使用Lambda表达式来求和
        useAddable((int x, int y) -> {
            return 5 * x + y - 15;
        }, 10, 20);
    }

    public static void useAddable(Addable a, int x, int y) {
        int add = a.add(x, y);
        System.out.println(add);

    }
}

//接口

public interface Addable {
    //声明一个方法，方法里面有两个参数
    //返回这两个整数之和
    int add(int x, int y);
```

```
}

```

- 方法引用与构造器

```
package lambda4;

import java.util.Comparator;
import java.util.function.*;

public class LambdaMethod {
    public static void main(String[] args) {
        /*
        public void print(int x){System.out.println(x)}
         */
        //lambda返回的变量,Consumer是一个接口，默认就有的，来一个参数处理一条
        //Consumer代表的是仅有一个参数输入，没有返回类型的操作
        Consumer<String> tConsumer = (x) -> System.out.println(x);
        //等价于，::所有里面只有一个输入值就会调用
        Consumer<String> tConsumer2 = System.out::println;

        //将一个字符串类型转换成int类型
        Function<String, Integer> stringIntegerFunction = (String x) ->
Integer.parseInt(x);
        System.out.println(stringIntegerFunction.apply("234"));
        //等价于
        Function<String, Integer> stringIntegerIntegerBiFunction = Integer::parseInt;
        System.out.println(stringIntegerIntegerBiFunction.apply("135"));


        //输入两个参数
        IntBinaryOperator intBinaryOperator = (int x, int y) -> Integer.compare(x + y,
y);
        System.out.println(intBinaryOperator.applyAsInt(2, 2));
        //等价于
        Comparator<Integer> compare = Integer::compare;
        System.out.println(compare.compare(2, 3));

        //自定义Student类，打印输出调用name
        Student wangwu = new Student("王五", 28);
        Supplier<String> runnable = () -> wangwu.getName();
```

```
        System.out.println("wangwu name" + runnable.get());
        ///等价于
        Supplier<String> getname = wangwu::getName;
        System.out.println("wangwu" + getname.get());
    }

}

```

- Lambda表达式和匿名内部类的区别
  - 所需类型不同
    - 匿名内部类：可以是接口，也可以是抽象类，还可以是具体类
    - Lambda表达式：只能是接口
  - 使用限制
    - 如果接口中有且仅有一个抽象方法，可以使用Lambda表达式，也可以使用匿名内部类
    - 如果接口中多于一个抽象方法，只能使用匿名内部类，而不能使用Lambda表达式
  - 实现原理不同
    - 匿名内部类：编译之后，产生一个单独的.class字节码文件
    - Lambda表达式：编译之后，没有一个单独的.class字节码文件，对应的字节码会在运行的时候动态生成
- 案例链式

```
import java.util.ArrayList;
import java.util.List;

public class StramDemo {
    public static void main(String[] args) {
        List<String> lists = new ArrayList<>();
        lists.add("hello");
        lists.add("hello");
        lists.add("hello");
        lists.add("hello");
        lists.add("hello");
        //stream:流，每个数组都有  forEach:遍历 后面是Lambda表达式简写
        lists.stream().forEach(System.out::println);
        lists.stream().map(x -> x.length()).forEach(System.out::println);
    }
}

```

2.15 输入流和输出流（IO流）简介

- 概述
  - IO流就是用来处理设备间的数据传输问题的，将文件上传、下载、复制问题
    - i指的是Input（输入），o指的是Output（输出）
    - 应用场景
      - 文件复制
      - 文件上传
      - 文件下载
    - 分类
      - 按照流向分
        - 输入流：读取数据
        - 输出流：写入数据
      - 按照操作分
        - 字节流：以字节为单位来操作数据
        - 字符流：以字符为单位来操作数据
      - 字节输出流FileOutputStream

```
package iosrtream;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import java.nio.charset.StandardCharsets;

/**
 * 1.创建FileOutputStream对象，关联指定的目的地文件
 * 2.往文件中写入a ，b ，c
 * 注意：
 * 1.如果目的地文件不存在，程序会自动创建
 * 2.如果目的文件的路径不存在，程序会报错
 */
public class FileOutputStreamDemo {
    public static void main(String[] args) {
        //FileOutputStream有构造器
        try {
            FileOutputStream fileOutputStreamDemo = new
FileOutputStream("D:\\logs\\file.txt", true);
```

```
            byte[] bytes={'a','b','c'};
            fileOutputStreamDemo.write(bytes);

            byte[] bytes1="hello world".getBytes();
            fileOutputStreamDemo.write(bytes1);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }catch (IOException e){
            e.printStackTrace();
        }
    }
}

```

- 字节输入流FileInputStream

```
package iosrtream;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;

/**
 * 1.创建FileInputStream对象，关联指定的数据源文件
 * 2.通过一次读取一个字节形式，读取该文件中的数据
 */
public class FileInputStreamDemo {
    public static void main(String[] args) {
        //有构造器，可以直接new
        try {
            FileInputStream fis = new FileInputStream("D:\\logs\\file.txt");
            /*//a=97,返回的是ASCII码
            int byte1 = fis.read();
            int byte2 = fis.read();
            int byte3 = fis.read();
            System.out.println(byte1 + " " + byte2 + " " + byte3);*/
            /**
             * 一次读取一批数据，一次读取出来10个字符
```

```
             */
            int len = 0;
            //定义一个字符组，为10个字节
            byte[] bytes = new byte[10];
            System.out.println(Arrays.toString(bytes));
            //len = 读取的字节具体的值
            while ((len = fis.read(bytes)) != -1) {
                //字节数组，从哪开始读，读取几个
                System.out.println(new String(bytes, 0, len));
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

- **工具类FileUtils**

```

import java.io.File;
import java.io.IOException;

public class FileUtilOutputDemo {
    public static void main(String[] args) {
        //使用FileUtils.write()，底层实现的就是FileOutputStream
        //为了简化起见，直接使用FileUtils工具类实现文件内容的读写
        try {
            //文件，内容
            FileUtils.write(
                    new File("D:\\logs\\file1.txt"),
                    "abc",
                    true
            );
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
}

```

- 复制文件案例

```
package iosrtream;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

//读取文件数据并将其拷贝到目标文件中
public class FileCopyDemo {
    public static void main(String[] args) {

        //1.读取指定的文件
        FileOutputStream fis = null;
        FileOutputStream fos = null;
        try {
            fis = new FileOutputStream("D:\\logs\\file.txt");
            //2.拷贝到指定的文件
            fos = new FileOutputStream("D:\\logs\\file_copy.txt");

            int length = 0;
            byte[] bytes = new byte[1024];
            while ((true)) {
                fos.write(bytes, 0, length);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e ){
            e.printStackTrace();
        }

        //3.释放资源
        finally {
            try {
                fis.close();
                fos.close();
            } catch (IOException e) {
```

```
                e.printStackTrace();
            }
        }
    }
}

```

```
package iosrtream;

import org.apache.commons.io.FileUtils;

import java.io.File;
import java.io.IOException;

/**
 * 需求，从文件 file.txt 中使用FileUtil工具读取文件并将其拷贝到另外一个文件中
 */
public class FileUtilCopy {
    public static void main(String[] args) {
        try {
            FileUtils.copyFile(
                    new File("D:\\logs\\file.txt"),
                    new File("D:\\logs\\file_copy2.txt")
            );
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}

```

- **序列化流**

- 概念：对象序列化和反序列化：就是将对象保存到磁盘或者网络传输中，将对象和二进制之间进行转换
- 对象序列化流：ObjectOutputStream
- 对象反序列化流：ObjectInputStream
- 如果进行序列化和反序列化
  - 序列化或者反序列化只需要在类对象上实现接口implements Serializable
  - ObjectInputStream.readObject（Object obj）：将二进制转换成对象

- ObjectOutputStream.writeObject（Object obj）：将对象转换成二进制

- 序列化

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class ObjectOutputStreamDemo {
    public static void main(String[] args) {
        //将对象Student要以二进制的形式写入磁盘
        try {
            ObjectOutputStream oos = new ObjectOutputStream(
                    new FileOutputStream("D:\\logs\\student.txt")
            );
            //定义一个Student对象
            Student s=new Student();
            s.setName("张三丰");
            s.setAge(20);
            oos.writeObject(s);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- 反序列化

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

/**
 * 实现读取序列化之后的文件，将其转换成对象并打印输出
 * 需要使用到 ObjectInputStream 对象
 */

public class ObjectInputStreamDemo {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        ObjectInputStream objectInputStream = new ObjectInputStream(
                new FileInputStream("D:\\logs\\student.txt")
```

```
        );
        Object o = objectInputStream.readObject();
        Student s1 = (Student) o;
        //打印输出这个对象
        System.out.println(s1.toString());
    }
}

```

2.16 JDBC

- 数据库表
  - java程序数据写入到数据库中
  - java 数据库
  - 类 表（字段和记录）
  - 成员变量 字段（字段名和字段类型）
  - 对象 记录（一行一行的数据）
- **JDBC概述和原理**
  - java客户端通过java database connectivity JDBC 操作数据库的Java API
  - 如何使用
    - 1.导入jar包 mysql-connector-java-8.0.17-bin.jar
    - 2.操作的常见接口和方法
    - Class.forName("com.mysql.java.Driver")
    - Connection:连接数据库，获取数据库连接
    - Statement：创建执行SQL的statement，通过它执行SQL可以是DDL、DML、DQL、DCL
    - ResultSet：生成记录集，仅executeQuery select 查询时候才会有ResultSet，返回值，如果有记录就是记录条数，否则null
    - PreparedStatement：预编译statement
    - 案例

```
package sz.base.java.jdbc;

import java.sql.*;

/**
 * 实现读取MySQL数据库中t_student 表的值
 * 将其封装到Student类对象中
 * 并将其打印输出
```

```
 * 开发步骤：
 * 1.创建对象Student 要 t_student 表一一对应上
 * 表字段=成员变量  表字段类型=成员变量的类型
 * 每个对象 = > 表的每一条记录
 * 2.加载MySQL的驱动 ： DriverManager 用于注册驱动
 * 3.获取连接MySQL的Connection  :表示与数据库创建的连接
 * 4.创建statement对象  ： 操作数据库SQL语句的对象
 * 5.生成结果集ResultSet  ： 结果集或一张虚拟表
 * 6.遍历ResultSet结果集
 * 7.如果存在记录，读取每个字段并将其封装到Student对象中
 * 8.打印出每行记录
 * 9.关闭结果集、statement、数据库连接
 */
public class ReadStudent {
    public static void main(String[] args) {
        //1.创建对象Student 要 t_student 表一一对应上
        // * 表字段=成员变量  表字段类型=成员变量的类型
        // 每个对象 = > 表的每一条记录
        //2.加载MySQL的驱动 ： DriverManager 用于注册驱动
        try {
/*
            Class.forName("com.mysql.jdbc.Driver");
            // * 3.获取连接MySQL的Connection  :表示与数据库创建的连接
            Connection connection = DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/day06?useSSL=false&characterEncoding=utf8",
                    "root",
                    "root"
            );
*/
            // 4.创建statement对象  ： 操作数据库SQL语句的对象
            Connection connection = JDBCUtils.getConnection();
            Statement stmt = connection.createStatement();
            //executeQuery(sql)：执行查询语句，返回resultest结果集
            //executeUpdate(sql)：执行更新、删除、插入语句返回影响的行数
            //5.生成结果集ResultSet  ： 结果集或一张虚拟表
            ResultSet resultSet = stmt.executeQuery("select sid,sname from t_student");
            // 6.遍历ResultSet结果集
            //resultSet.next()：是否有下一行的记录的值，如果有为true否则false
            while (resultSet.next()) {
```

```
                // 7.如果存在记录，读取每个字段并将其封装到Student对象中
                int sid = resultSet.getInt("sid");
                String sname = resultSet.getString("sname");
                Student s = new Student(sid, sname);
                //  8.打印出每行记录
                System.out.println(s);
            }
            // 9.关闭结果集、statement、数据库连接
            JDBCUtils.close(connection, stmt, resultSet);
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }
}

```

- **JDBC工具类**

```
package sz.base.java.jdbc;

import java.sql.*;

/**
 * 此类是一个工具类，主要用于
 * 1.获取连接数据库的对象connection
 * 2.获取关闭数据库的对象close
 */
public class JDBCUtils {
    //定义常量URL 用户名 和密码
    private static final String URL = "jdbc:mysql://localhost:3306/day06?
useSSL=false&characterEncoding=utf8";
    private static final String user = "root";
    private static final String PASSWORD = "root";


    //开发步骤

    /**
     * 1.初始化MySQL或Oracle的驱动
```

```
21      * 2.创建获取connection的方法
22      * 3.创建关闭connection的方法
23      */
24     //静态代码块主要用于初始化静态变量，在类被初始化，被装载到内存中
25     //初始化工作,1.初始化MySQL或Oracle的驱动
26     static {
27         try {
28             Class.forName("com.mysql.cj.jdbc.Driver");
29         } catch (ClassNotFoundException e) {
30             e.printStackTrace();
31         }
32     }
33
34     public static Connection getConnection() {
35         //定义静态变量
36         Connection conn = null;
37         try {
38             conn = DriverManager.getConnection(
39                     URL,
40                     user,
41                     PASSWORD
42             );
43         } catch (SQLException e) {
44             e.printStackTrace();
45         }
46         return conn;
47     }
48     //创建关闭connection的方法
49     public static void close(Connection connection, Statement stmt, ResultSet resultSet)
   throws SQLException {
50         if (resultSet != null && !resultSet.isClosed()) resultSet.close();
51         if (!stmt.isClosed()) stmt.close();
52         if (!connection.isClosed()) connection.close();
53     }
54 }
55
```

- SQL的增删改查

- **预编译执行平台**

- 为了防止出现SQL注入问题，使用PreparedStatement来解决对应的问题

- PreparedStatement是statement的子类，具备如下特点：
  - 1.性能高
  - 2.会把SQL语句先编译
  - 3.能过滤掉用户输入的关键词
- 处理每条SQL语句中所有的实际参数都必须使用占位符？替换

```java
package sz.base.java.jdbc;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;

public class AddStuPreParedStmt {
    public static void main(String[] args) throws SQLException {
        //1.获取数据库的连接
        Connection connection = JDBCUtils.getConnection();

        //3.执行插入更新操作
        String sql = "insert into t_student(sid,sname) values (?,?)";
        //2.创建Statement PreparedStatement 预处理对象代码：
        PreparedStatement statement = connection.prepareStatement(sql);
        // statement.execute(sql) 如果能够返回第一条数据就返回 true; 否则update delete 都返回 false
        // statement.executeQuery(sql) 执行查询操作,返回ResultSet结果集
        //??从1开始：
        statement.setInt(1,5);
        statement.setString(2,"Jack Ma");
        int lines = statement.executeUpdate();
        if (lines > 0) {
            System.out.println("插入数据表成功，影响的行数为：" + lines);
        }
        //4.关闭连接
        JDBCUtils.close(connection, statement, null);
    }
}

```

- **建立连接池重写工具类**

- 创建连接池的目的
  - 解决建立数据库连接消费 资源和时间很多的问题，提高性能
- 常见连接池工具类
  - DBCP2 Apache 免费开源，支持返回数据库连接资源
  - C3P0 Apache 免费开源，支持数据连接资源自动回收
  - DRUID 阿里的数据库连接池
- 开发步骤
  - 1.下载添加 c3p0 jar包
  - 2.修改配置文件并加载当前class类路径下stc下c3p0-config.xml连接数据库地址，端口，数据库，连接池相关参数

```
<c3p0-config>
    <!-- 使用默认的配置读取连接池对象 -->
    <default-config>
        <!--  连接参数 -->
        <property name="driverClass">com.mysql.cj.jdbc.Driver</property>
        <property name="jdbcUrl">jdbc:mysql://localhost:3306/day06</property>
        <property name="user">root</property>
        <property name="password">root</property>

        <!-- 连接池参数 -->
        <property name="initialPoolSize">5</property>
        <property name="maxPoolSize">10</property>
        <property name="checkoutTimeout">2000</property>
        <property name="maxIdleTime">1000</property>
    </default-config>
</c3p0-config>
```

  - 3.修改java JDBC API 数据库连接 ComboPooledDataSource 连接，非Connection连接

```
ComboPooledDataSource cpd = new ComboPooledDataSource();
conn = cpd.getConnection();
```

```

import com.mchange.v2.c3p0.ComboPooledDataSource;

import javax.sql.DataSource;
import java.sql.Connection;
```

```java
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JdbcUtils {
        //创建一个C3P0的连接池对象（使用c3p0-config.xml中default-config标签中对应的参数）
        public static DataSource ds = new ComboPooledDataSource();

        //从池中获得一个连接
        public static Connection getConnection() throws SQLException {
            return ds.getConnection();
        }

        //释放资源
        public static void closeAll(ResultSet rs, Statement stmt, Connection conn){
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
                rs = null;
            }
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
                stmt = null;
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
                conn = null;
            }
        }
```

```
46
47 }
48
```

- JDBC和cp30创建连接区别
  - 1.JDBC Class.forName("com.mysql.jdbc.Driver")获取
  - DriverManager.getConnection(url,root,password)
  - 2.ComboppooledDataSource 获取示例，对象.getConnection()获取连接
  - 3.c3p0配置连接池的属性，初始化的连接池的数据，最大的连接的数量

2.17实务操作

- 事务的概念
  - 逻辑上一组操作，要么都成功，要么都失败，就是事务
- 事务的特征ACID
  - 1.原子性 Atomicity：组织事务的各个逻辑单元已经是最小单元，不可再分，要么同时成功、要么同时失败
  - 2.一致性 Consistency ： 事务执行前后，数据保持一致
  - 3.隔离性 Isolation ： 一个事务执行的时候，不应该受到其他事务的干扰
  - 4.持久性 Durability ： 无论事务执行成功与否，结果都会永久的存储到数据表中
- 常见mysql事务操作
- 默认MySQL自动提交：来一条处理一条commit数据表
- 默认MySQL中事务自动提交autocommit，手动维护事务提交
  - 1.关闭自动提交conn.autoCommit(false)
  - 2.startTransaction()
  - 3.如果成功，提交事务commit

```java
package sz.base.java.jdbc;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * 模拟张三给李四转账这个事情
 * 要求转账要么都成功要么都失败
 */
public class TransactionDemo {
    public static void main(String[] args) {
        //1.获取连接connection
        Connection connection = JDBCUtils.getConnection();
```

```
        //2.关闭自动提交事务
        try {
            //默认情况下MySQL是自动提交事务的 autoCommit=true
            connection.setAutoCommit(false);
            //获取statement
            Statement statement = connection.createStatement();
            //3.执行张三的账号-1000的操作
            int rows1 = statement.executeUpdate("update account set money=money-1000 where name ='jack' ");
            //4.抛出一个异常，报个错
            new Exception("出错了");
            //5.执行李四的账号+1000的操作
            int rows2 = statement.executeUpdate("update  account set money=money+1000 where name='rose'");
            //6.处理张三和李四两个人的执行情况都要影响，提交成功
            if (rows1 > 0 && rows2 > 0) {
                System.out.println("转账1000成功！");
                //7.手动提交事务
                connection.commit();
            } else {
                System.out.println("转账失败，请检查！");
                connection.rollback();
            }

        } catch (SQLException e) {
            e.printStackTrace();
            System.out.println("转账失败，请检查！");
            try {
                //8.如果出现异常，给回滚机制
                connection.rollback();
            } catch (SQLException ex) {
                ex.printStackTrace();
            } finally {
            }
        }
    }
}

```

2.18多线程

- 进程和线程
  - 进程是资源分配的最小单位，线程是CPU调度的最小单位
  - 进行指的是可执行文件、程序，静态的概念
  - 线程指的是进行执行路径，执行单元
- 进程和线程的关系就相当于 进行=火车，线程=车厢
- 实现线程类
  - 核心类 Thread
  - 接口 Runable
  - 工具类 Executors.newThreadCache() 通过获取线程池（了解）
- 成员方法
  - 1.run():实现具体线程做的事情
  - 2.start():实现当前线程开启/执行
  - 3.sleep(): 当前线程睡眠，阻塞，不往后执行
  - 4.join()：插入线程执行，先执行join进来的线程，执行完毕执行再接着执行，join(10)：当前线程等待10s钟
- 多线程实现的方式
  - 方式1：继承Thread类
  - 方式2：实现Runnable接口的优势
    - 1.适合多个相同的程序代码的线程去共享同一个资源
    - 2.可以避免java中的单继承的局限性
    - 3.增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和线程独立
    - 4.线程池只能放入实现Runable或Callable类线程，不能直接放入继承Thread的类
- **Thread方式**

```
1
2  /**
3   * 线程执行的，实现多线程，继承Thread 类
4   * 打印输出0~300 执行线程的逻辑，并将哪个线程执行的 名称 也打印出来
5   */
6  public class ThreadDemo {
7      public static void main(String[] args) {
8          //执行了线程
9          CustomThread th = new CustomThread();
10         th.start();
11         //在主线程中也打印输出300 次
12         for (int i = 0; i < 300; i++) {
13             System.out.println("main主线打印输出！" +i);
```

```
        }
    }
}

```

```
public class CustomThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 300; i++) {
            System.out.println(this.getName() + ": execute! " + i);
        }
    }
}

```

```
import sz.base.java.runnable.CustomRunnble;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 线程池的方式
 */
public class ExecutorsDemo {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        executorService.submit(new CustomRunnble());
        for (int i = 0; i < 300; i++) {
            System.out.println("main主方法: "+i);

        }
    }
}

```

- **Runnable方式**

```java
package sz.base.java.runnable;

public class CustomRunnble implements Runnable{
    @Override
    public void run() {
        for (int i = 0; i < 300; i++) {
            System.out.println(Thread.currentThread().getName()+":execute!"+i);
        }
    }
}

```

```java
public class RunnableDemo {
    public static void main(String[] args) {
        CustomRunnble run=new CustomRunnble();
        //自定义Thread
        Thread th =new Thread(run);
        th.start();
        //主方法打印 0 ~ 300
        for (int i = 0; i < 300; i++) {
            System.out.println("main主方法: "+i);
        }
    }
}

```

- **线程池方式**

```java
import sz.base.java.runnable.CustomRunnble;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
/**
 * 线程池的方式
 */
public class ExecutorsDemo {
    public static void main(String[] args) {
        //创建线程池，参数为设置多少个线程
        ExecutorService executorService = Executors.newFixedThreadPool(10);
```

```
        //提交给线程池，CustomRunnble是继承Thread的类重写run方法
        executorService.submit(new CustomRunnble());
        for (int i = 0; i < 300; i++) {
            System.out.println("main主方法: "+i);

        }
    }
}

```

- **线程安全**
- 程序每次执行结果和单线程运行结果是一样的，而且其他变量的值也是一样的，就是线程安全的
- 线程同步
  - 多个线程访问同一个资源，多线程同时写操作造成资源的安全性问题，要解决重复和不存在票问题，使用synchronized同步机制来解决，三种方法
    - 1.同步代码块synchronize(对象)
    - 2.同步方法
    - 3.lock锁机制-ReentrantLock(公平锁和非公平锁)

```
package sz.base.java.thread;

/**
 * 实现卖票的功能，在此类中，主要实现一个卖票功能的线程功能
 * 开发步骤:
 * 1.实现一个Runnable接口，重写run方法
 * 2.循环售卖100张电影票，直到卖完为止
 */
public class SaleTicket implements Runnable {
    //初始化电影票
    private static int tickets = 100;
    //加锁，任意类型,防止多个线程同时访问成员变量
    final Object saleTicket = new Object();

    /**
     * 实现卖票的功能，卖掉100张电影票
     */
    @Override
    public void run() {
        while (true) {
```

```
            if (tickets > 0) {
                //线程休眠时间设置为200ms
                try {
                    Thread.sleep(200);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                //获取一下当前线程的名字
                String currentName = Thread.currentThread().getName();
                //调用同步方法
                ticketDesc(currentName);
                //同步代码块
               /* synchronized (saleTicket) {
                    System.out.println(currentName + " 还剩票数: " + tickets--);
                }*/
            } else {
                break;
            }
        }
    }
    //同步方法 ,synchronized加锁
    public static synchronized void ticketDesc(String curr) {
        System.out.println(curr + " 还剩票数: " + tickets--);
    }
}
```

- **死锁**
- 死锁指的是多个线程同时抢多把锁，因为CPU执行线程的随机性，从而导致线程卡死的情况

```
package sz.base.java.thread;

public class DeadLockDemo {
    //定义两把锁
    private static Object LockA= new Object();
    private static Object LockB= new Object();
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true){
```

```
                    synchronized (LockA) {
                        System.out.println(Thread.currentThread().getName()+"获取到锁A，等待锁B");
                        synchronized (LockB){
                            System.out.println("锁A 和 锁B 都获取的到了");
                        }
                    }
                }
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true){
                    synchronized (LockB){
                        System.out.println(Thread.currentThread().getName() + " 获取到锁B，等待锁A");
                        synchronized (LockA){
                            System.out.println("锁A 和 锁B 都获取到了");
                        }
                    }
                }
            }
        }).start();
    }
}

```

- **线程的生命周期**
- 1.新建
- 2.就绪
- 3.运行（可能发生阻塞或等待）
  - 阻塞：多指IO流阻塞，认为不可控
  - 等待：多指wait（），sleep（），人为可控
- 4.死亡
- **线程安全的类**
- 1.StringBuffer 是线程安全，类似于StringBUilder（线程不安全）
- Vector => ArrayList ： 类似于数组用法

- HashTable =》 类似于HashMap

- **守护进程**

- 概念：在一个JVM不存在任意一个正在运行的非守护线程时候，JVM进程就会退出
- 应用场景
  - 监控主线程，比如说读取kafka或者执行或者执行spark任务，监控当前主任务是否执行失败
  - GC垃圾回收机制
  - 提供服务支持

```java
package sz.base.java.thread;

/**
 * 在main方法中创建一个线程，main执行完毕会关闭主main线程
 * 开启的其他 线程会一直执行下去，直到结束
 */
public class ThreadDemo {
    public static void main(String[] args) throws InterruptedException {
        //当前JVM虚拟机
        Runtime.getRuntime().addShutdownHook(new Thread(() -> System.out.println("当前JVM 虚拟机main方法退出"))
        );
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("当前程序正在运行。。。");
                }
            }
        });
        //创建守护进程，后台进程，如果main方法中没有非守护进程执行了，守护进程跟着退出
        thread.setDaemon(true);
        thread.start();

        Thread.sleep(1000);
        System.out.println("当前程序执行完毕");
```

```
    }
}

```

2.19异常

- 概念：Java中的异常指的是程序出现不正常的情况
- 异常错误的分类，都继承Throwable父类
  - Error错误
  - Excption异常
- JVM的默认处理方法
  - try...catch...finally
  - 声明抛出异常格式
    - throws 异常的类型
  - 访问异常信息
  - 所有的异常对象包含如下几个常用方法

```
getMessage(): 返回该异常的详细描述字符串。
printStackTrace(): 将该异常的跟踪栈信息输出到标准错误输出。
printStateTrace(PrintStream s): 将该异常的跟踪栈信息输出到指定输出流。
getStackTrace(): 返回该异常的跟踪栈信息。
```

- 案例 空指针异常

```
import java.text.SimpleDateFormat;
import java.util.Date;

//空指针异常错误
public class ExceptionDemo2 {
    public static void main(String[] args) {
        SimpleDateFormat sdf = null;
        try {
            String format = sdf.format(new Date());
            System.out.println(format);
        } catch (NullPointerException e1) {
            String message = e1.getMessage();
            System.out.println("空指针异常错误" + message);
            e1.printStackTrace();
        }
    }
```

```
}
```

2.20Junit测试

- 介绍
  - Junit是一个Java语言的单元测试框架，属于白盒测试，用于取代java的main，Junit属于第三方工具，需要导入jar包后使用
  - 导入junit包
  - 将junit类设置到当前项目中
  - 通过注解来判断是否进行测试
- 常见注解
  - @test： 测试当前的方法
  - @Before： 在测试工作之前需要测试的方法
  - @After： 在最后的方法注解

```java
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class JunitDemo {
    @Before
    public void before() {
        System.out.println("创建数据库的连接");
    }

    @Test
    public void insert() {
        System.out.println("将记录插入到数据表中");
    }

    @After
    public void close() {
        System.out.println("关闭数据库的连接");
    }
}
```

2.21Maven的基本使用

- Maven的概述和作用

- 构建项目构建项目管理的软件，可以管理项目构建（clear、compile、package、install、deploy）、依赖管理（坐标groupID，artifact，version），帮助开发导入jar包，直接导入jar包的坐标就可以，导入管理第三方jar
- 项目管理生命周期
  - compile 编译
  - clear 清除
  - package 打包
  - install 安装
  - test 测试
  - deploy 部署
- 工程分模块进行构建
  - 模块与模块
  - 子模块与子模块之间，子模块与父模块之间

- Maven的仓库分类
  - 本地仓库-本机上存储的所有jar包
  - 远程仓库-局域网内的服务器上的所有jar包
  - 中央仓库-阿里云仓库，163仓库，maven2
- Maven的坐标
  - groupId：包Id
  - artifactId： 模块iD

Maven的安装和环境变量配置

- Maven官方下载地址

```
https://dlcdn.apache.org/maven/maven-3/3.8.4/binaries/apache-maven-3.8.4-bin.zip
```

- 安装
  - 只需要解压缩到指定目录 D:\maven 下，在目录下创建一个repo文件用于保存jar包
- 配置环境变量
  - 设置 MAVEN_HOME： D:\maven
  - 设置 PATH： %MAVEN_HOME%\bin
- Maven配置本地仓库

```
# 只需要在 maven 配置目录下 settings.conf

<localRepository>E:\maven\repo - 指定成自己本地目录</localRepository>

#配置阿里云仓库
 <mirror>
```

```
7
8           <mirror>
9                   <id>alimaven</id>
10                  <name>aliyun maven</name>
11                  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
12                  <mirrorOf>central</mirrorOf>
13           </mirror>
14
15          <mirror>
16                  <id>aliyunmaven</id>
17                  <mirrorOf>*</mirrorOf>
18                  <name>阿里云spring插件仓库</name>
19                  <url>https://maven.aliyun.com/repository/spring-plugin</url>
20          </mirror>
21
22
23  </mirror>
```

- 在控制台输入,出现版本号即可配置成功

```
1  mvn -version
```

- IDEA整合本地Maven工程

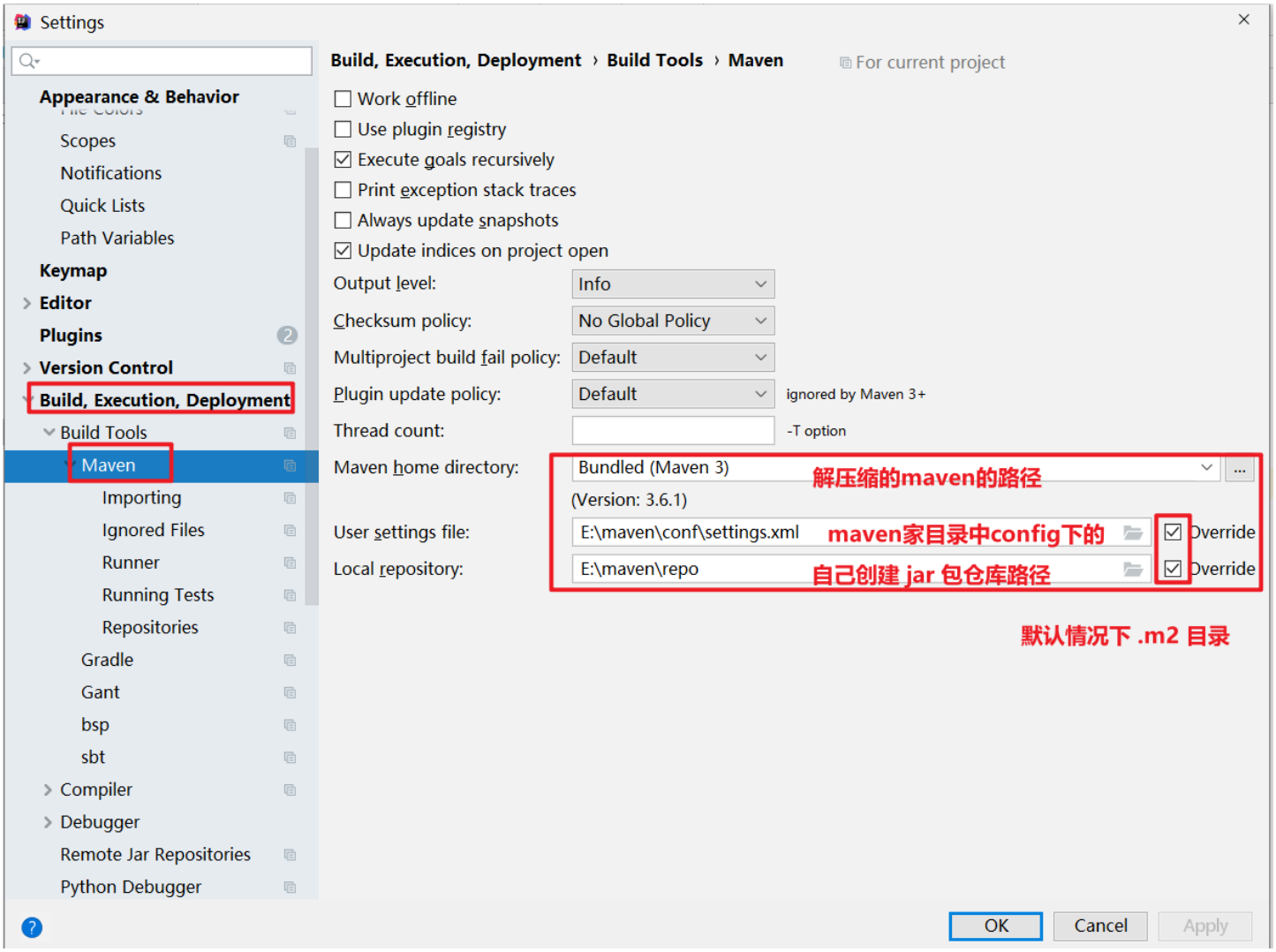


image-20211015101658781

- 创建Maven项目

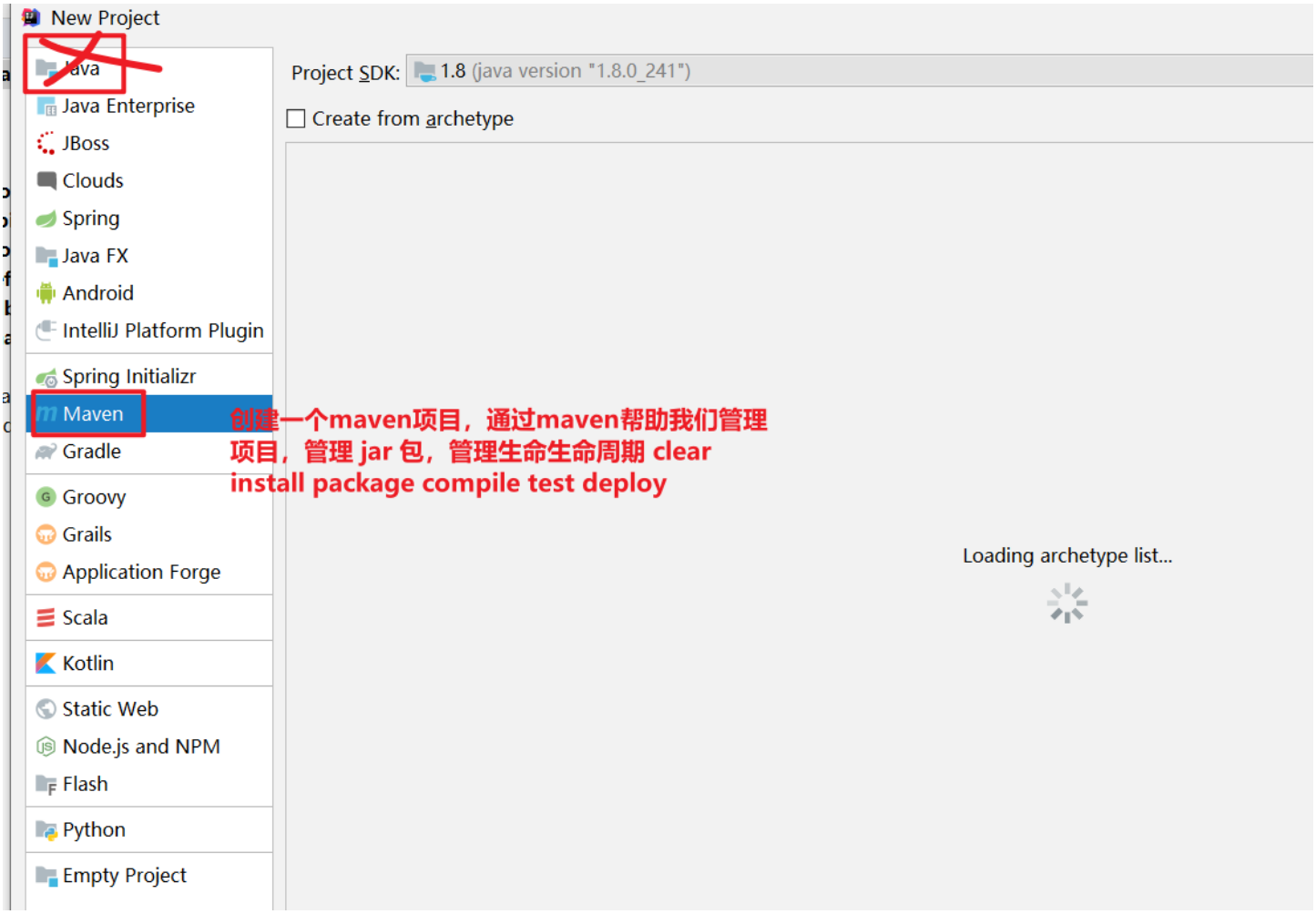
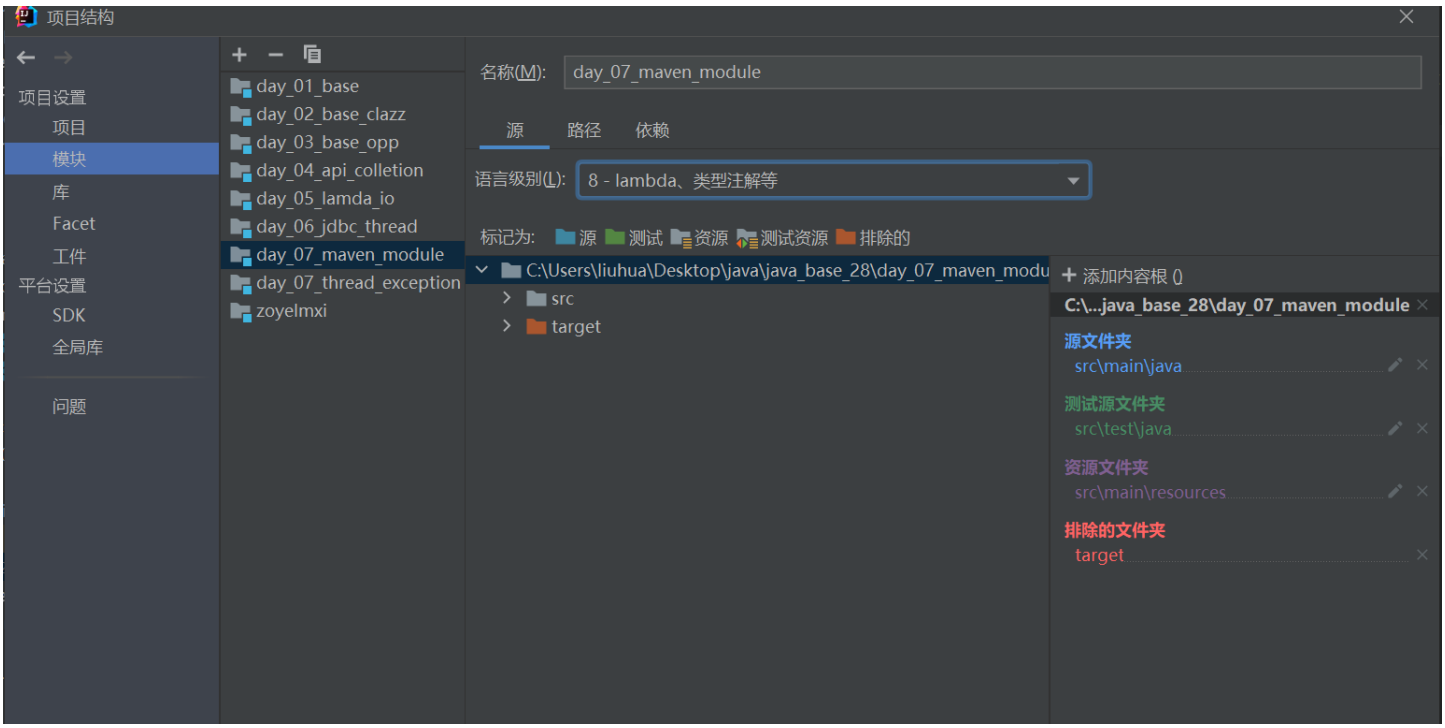


image-20211015101854265

- 项目结构选择 8



- Maven的常见指令
  - clear 清理所有target目录下所有jar包文件也包括配置等
  - compile 编译程序，只编译不运行
  - test测试 /src/test/java下面所有测试类，junit @Test 自动测试

- package 将程序打成jar包，jar包存放在target目录下
- install 安装，将打包保存到本地仓库

依赖管理

- 重新导入jar包-创建Maven项目会有pom.xml文件配置
- 在pom.xml文件--maven插件，配置坐标

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>cn.itcast</groupId>
    <artifactId>maven_javaweb02</artifactId>
    <version>1.0-SNAPSHOT</version>
<dependencies>
坐标文件
</dependencies>
    <build>
        <plugins>
            <!-- java编译插件 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

- 项目创建会有一个resource--配置文件都放进这里--比如c3p0-config.xml文件

配置tomcat插件

```xml
<!-- 配置Tomcat插件 -->
<plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
        <configuration>
            <path>/aaa</path>
            <port>8888</port>
        </configuration>
</plugin>
```

- 注意问题
  - Maven的中央仓库只有Tomcat7.x版本
  - 如果想使用Tomact8.x版本

```
1. 需要从第三方仓库查找
2. 或者使用idea集成外部的tomcat插件
```

- **扩展插件：IDEA的lombok插件**
- 作用：不用重写tostring getname setname等方法
- 下载IDEA插件Lombok

- 在pom.xml到如lombok的jar包

Project Lombok » 1.18.12

Spice up your java: Automatic Resource Management, automatic generation of getters, setters, equals, hashCode and toString, and more!

License	MIT
Categories	Code Generators
HomePage	https://projectlombok.org
Date	(Feb 07, 2020)
Files	jar (1.7 MB) View All
Repositories	Central
Used By	14,762 artifacts

Note: There is a new version for this artifact

New Version	1.18.22

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
    <scope>provided</scope>
</dependency>
```

```
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
    <scope>provided</scope>
</dependency>

```

- 注解
  - @Data //set方法 get方法 toString方法 没有全参构造
  - @AllArgsConstructor
  - @NoArgsConstructor
- 测试

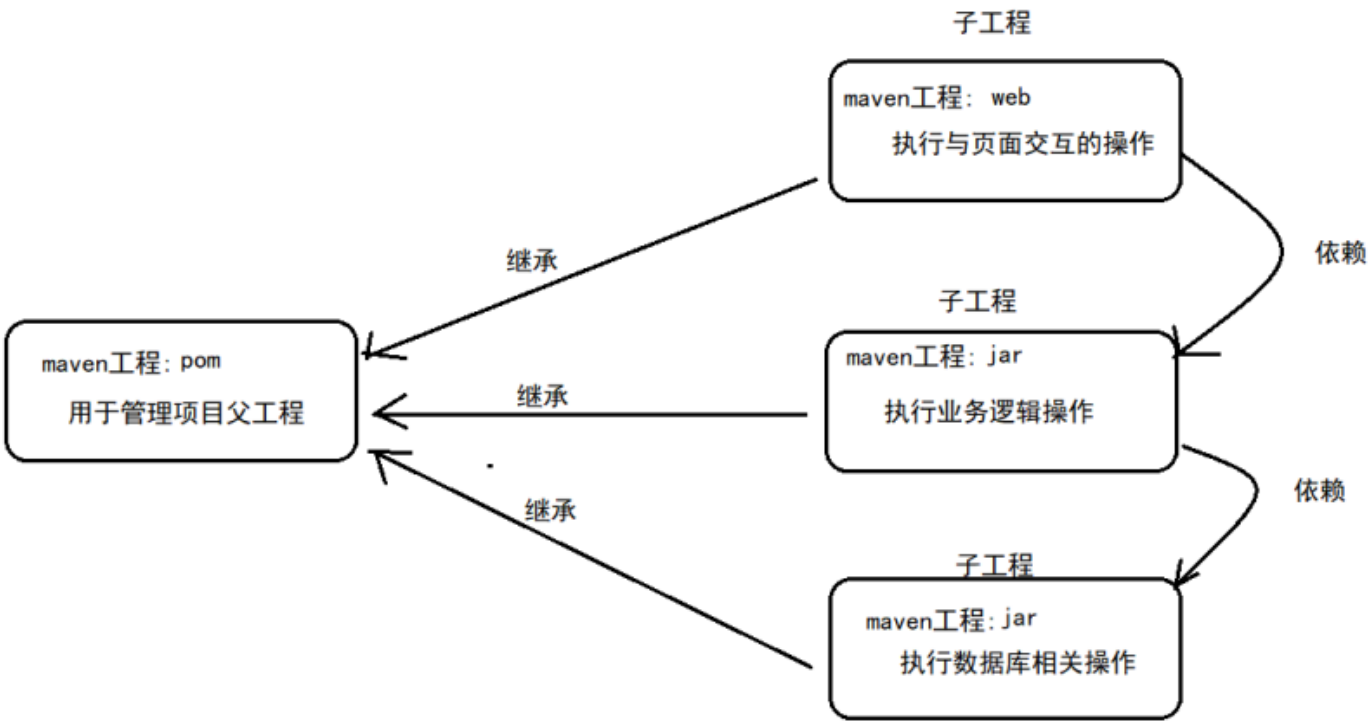
```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Student {
    private String name;
    private int age;
```

```
}

```

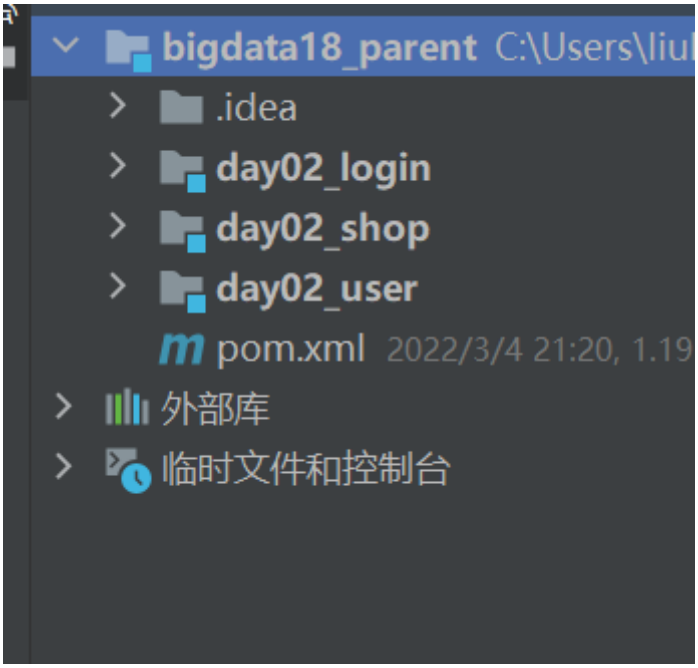
```
public class StudentDemo {
    public static void main(String[] args) {
        Student student = new Student("张三",20);
        System.out.println(student);
    }
}

```

- **聚合工程**

- 子工程继承父工程，子工程之间相互依赖
- 子工程之间需要package install才能能被另一个子工程寻找到



- 在父工程，主要是可以对多个子工程进行管理：比如 锁定jar包的版本,定义一些公共的插件(jdk的编译插件)等
- 在子工程中 可以根据自己的业务需要，使用对应的jar包，如果在父工程中定义了对应jar包的版本后，子工程此时不需要指定jar包版本，同时如果父工程已经定义了相关的插件，子工程也无需定义
- 下面是父工程中创建3个子工程的配置--父工程删除src文件
- 在父工程添加插件，子工程都会继承
- 在父工程里面添加jar包，子工程也会进行继承的
- <dependencyManagement>锁定版本，此时在父工程中并不会真正的将这个jar包导入



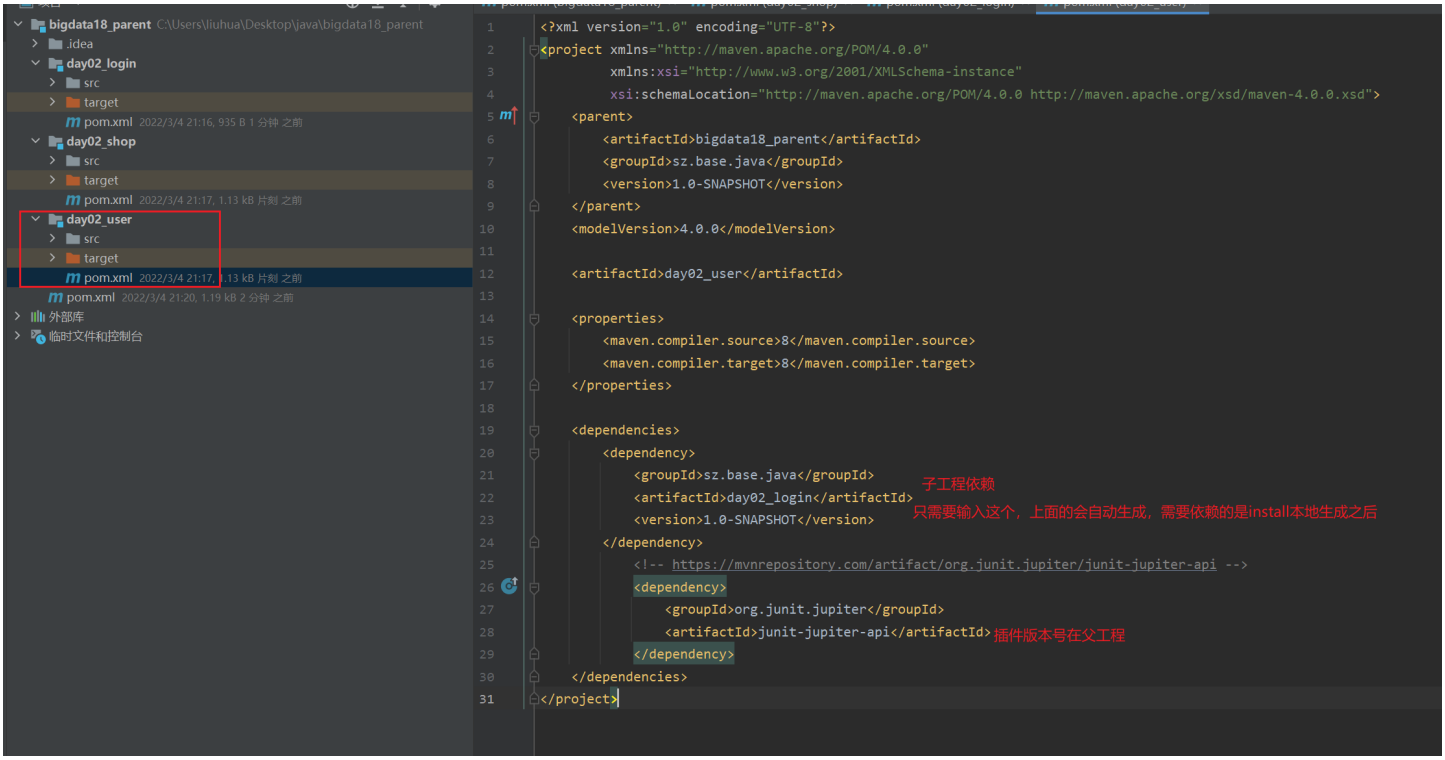
- 父工程pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>sz.base.java</groupId>
    <artifactId>bigdata18_parent</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <modules>
        <module>day02_login</module>  子工程的继承
        <module>day02_shop</module>
        <module>day02_user</module>
    </modules>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>  版本号
    </properties>
    <dependencyManagement>
        <dependencies>  插件坐标
            <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
            <dependency>
                <groupId>org.junit.jupiter</groupId>
                <artifactId>junit-jupiter-api</artifactId>
                <version>5.8.2</version>
                <scope>test</scope>  插件版本号
            </dependency>

        </dependencies>
    </dependencyManagement>

</project>
```

```xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
```

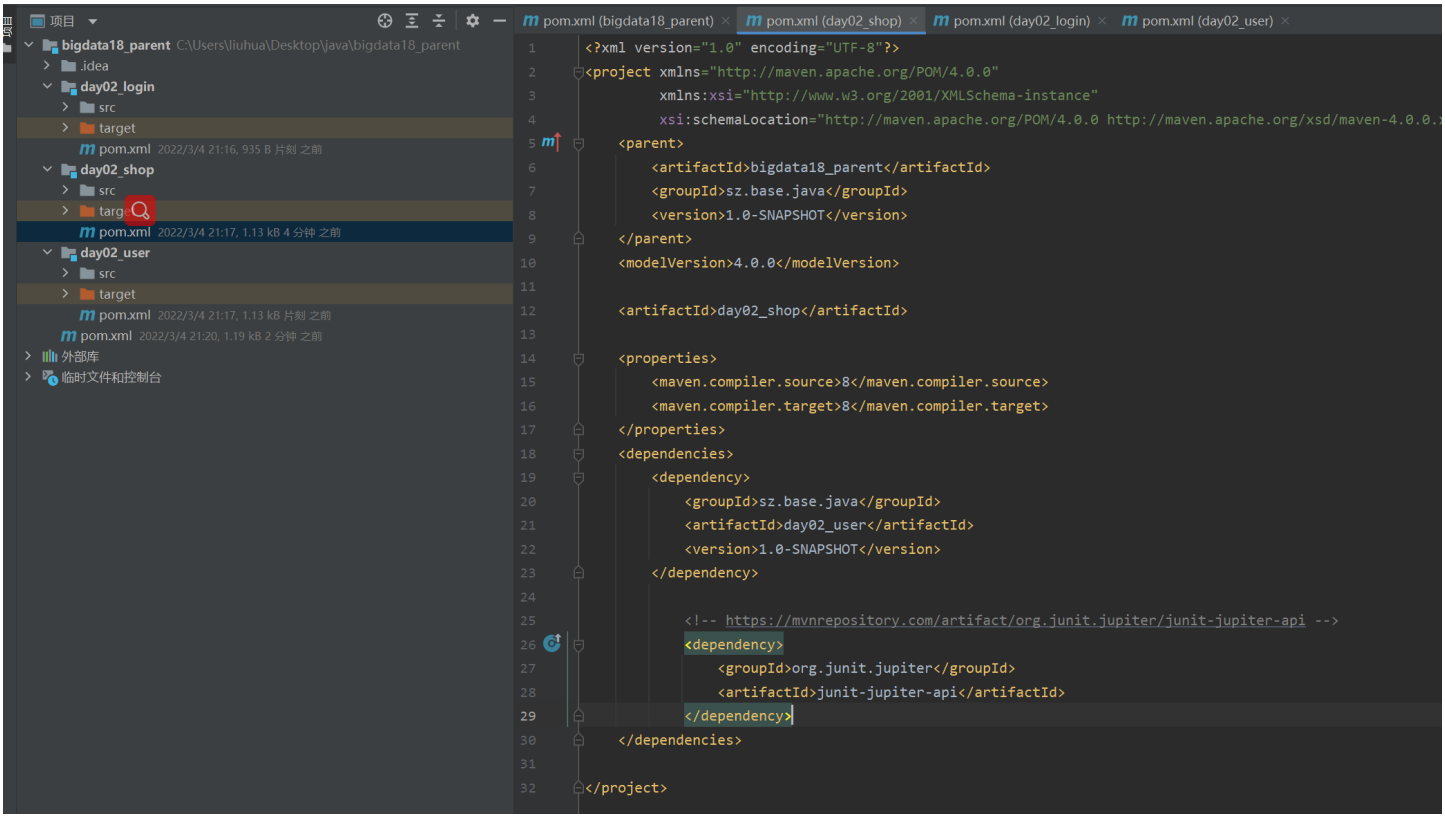
```
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5       <modelVersion>4.0.0</modelVersion>
6
7       <groupId>sz.base.java</groupId>
8       <artifactId>bigdata18_parent</artifactId>
9       <packaging>pom</packaging>
10      <version>1.0-SNAPSHOT</version>
11      <modules>
12          <module>day02_login</module>
13          <module>day02_shop</module>
14          <module>day02_user</module>
15      </modules>
16
17      <properties>
18          <maven.compiler.source>8</maven.compiler.source>
19          <maven.compiler.target>8</maven.compiler.target>
20      </properties>
21      <dependencyManagement>
22          <dependencies>
23              <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api
-->
24              <dependency>
25                  <groupId>org.junit.jupiter</groupId>
26                  <artifactId>junit-jupiter-api</artifactId>
27                  <version>5.8.2</version>
28                  <scope>test</scope>
29              </dependency>
30
31          </dependencies>
32      </dependencyManagement>
33
34  </project>
```

- 子工程1



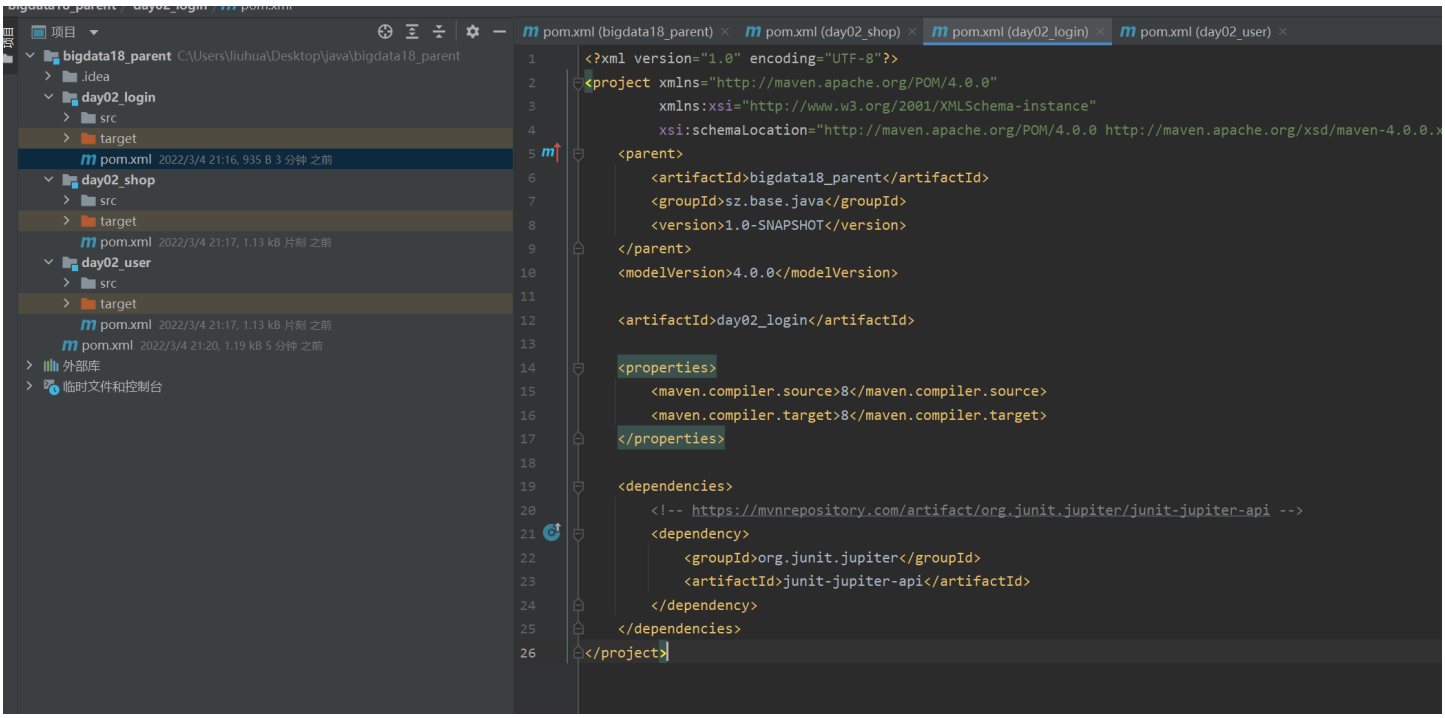

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>bigdata18_parent</artifactId>
        <groupId>sz.base.java</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>day02_user</artifactId>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>sz.base.java</groupId>
            <artifactId>day02_login</artifactId>
            <version>1.0-SNAPSHOT</version>
```

```
        </dependency>
            <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api
-->
            <dependency>
                <groupId>org.junit.jupiter</groupId>
                <artifactId>junit-jupiter-api</artifactId>
            </dependency>
    </dependencies>
</project>
```




```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>bigdata18_parent</artifactId>
        <groupId>sz.base.java</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>day02_shop</artifactId>

```

```
    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>sz.base.java</groupId>
            <artifactId>day02_user</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>

            <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api
-->
            <dependency>
                <groupId>org.junit.jupiter</groupId>
                <artifactId>junit-jupiter-api</artifactId>
            </dependency>
    </dependencies>

</project>
```




```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
    <parent>
        <artifactId>bigdata18_parent</artifactId>
        <groupId>sz.base.java</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>day02_login</artifactId>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
        </dependency>
    </dependencies>
</project>
```