

2020-5-5会议记录

微信群：重要通知

QQ群：讲义、资料、文档、技术提问与答疑

CCtalk：课程录播、作业（19：30左右发）

上午预习：重视预习

下午预习：及时回顾上午讲的东西

晚上答疑：20：00 - 21：30

打卡：（重视）

提前5分钟进腾讯会议 挂着

迟到一次10块钱 每天最多3次

晚上10点统一发

教材：课件与讲义

作业不会：问老师

推荐用纸质的笔记本 上课就好好的上课 别敲代码

IDEA

ctrl + alt + m: 把main方法中的代码抽出来成为一个独立的方法

new ClassName() + alt + enter 快速创建一个类的对象

shift + F6 : 变量名后按，可以修改全部的变量名

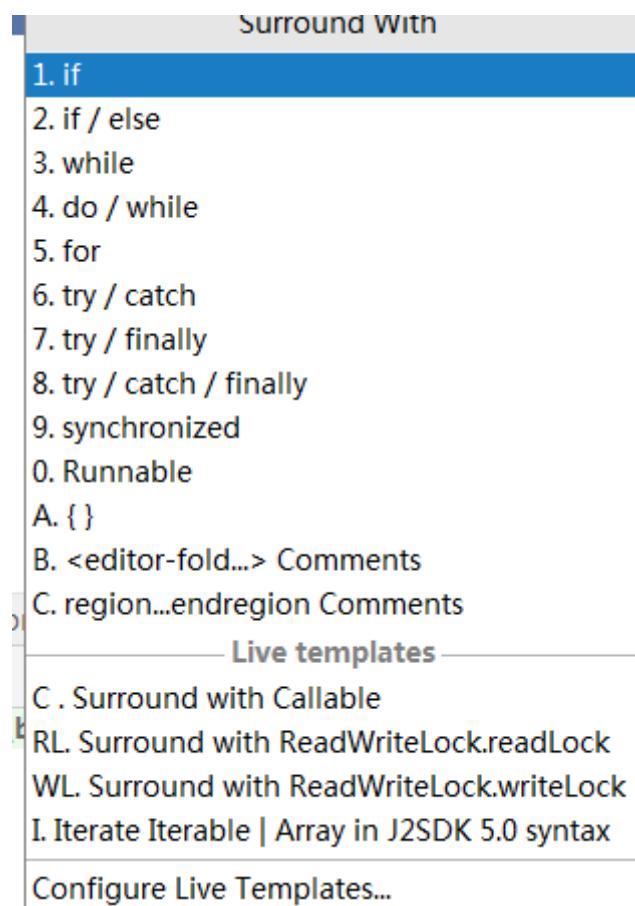
alt+insert: 定义各种方法快捷键

alt + enter: 补全很多东西

如果要看源码： ctrl + 鼠标左键点击

如果要找目标方法： 目标类中 ctrl + f12 然后键盘搜索要找的方法名

ctrl + alt + t

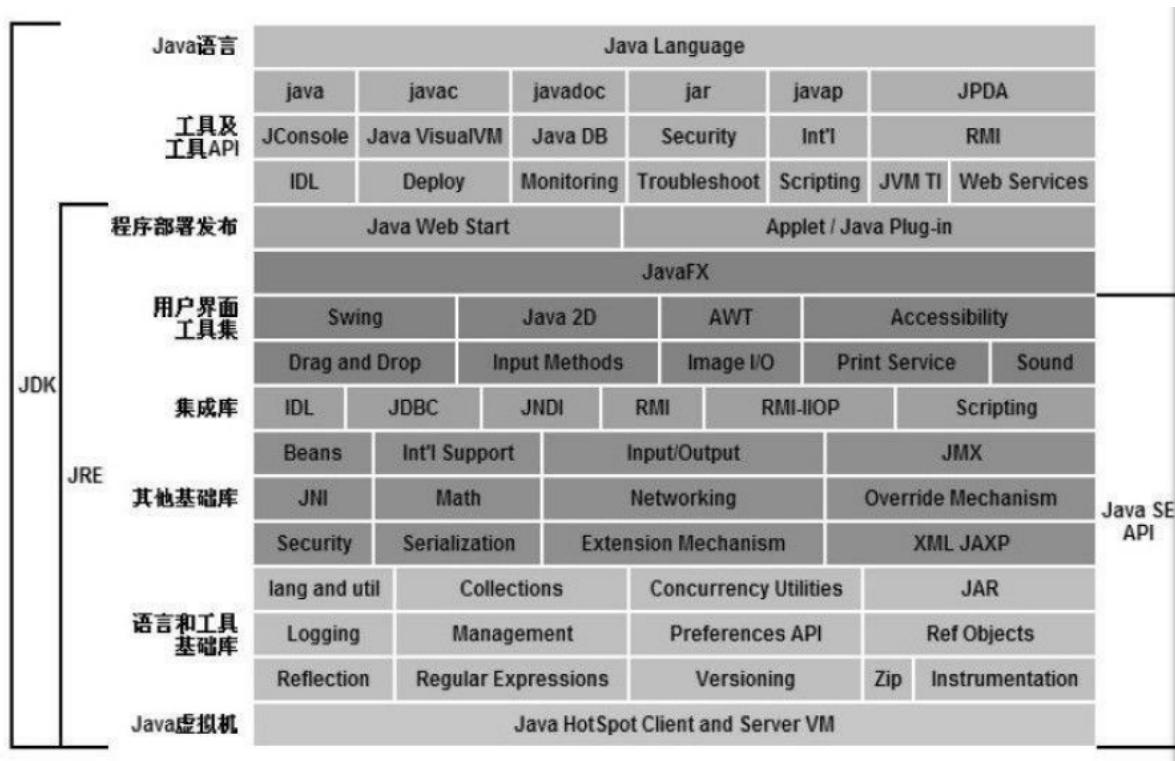


Week 1

day 1

Java语言版本

JDK



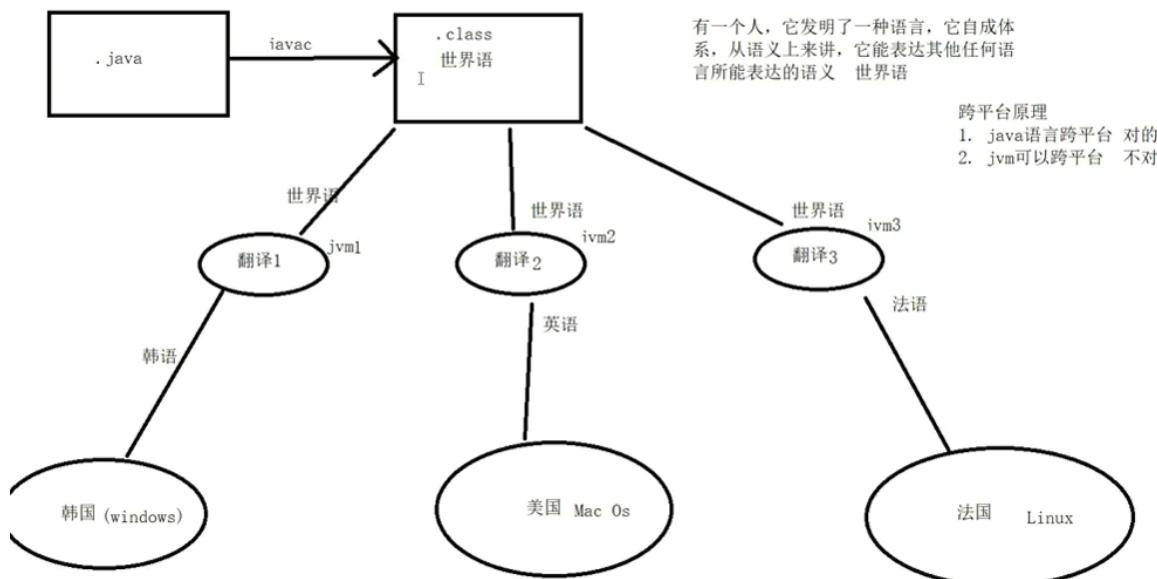
jdk(oracle) vs openJdk

jdk8

long term support: jdk8 jdk11 jdk17

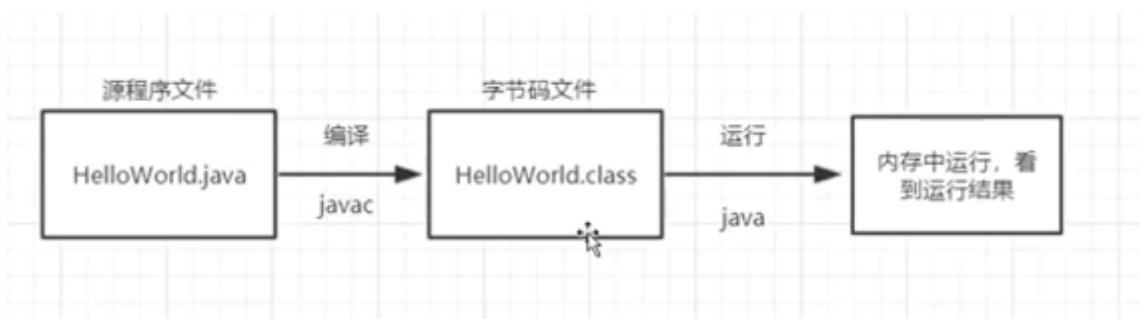
单机场景Javase与企业场景javaee不同

跨平台



javac javap 互逆运算

7. JAVA程序运行原理



配置Path环境变量：告诉OS Javac到哪里去找

需在当前目录下 javac

8. Path和Classpath环境变量

环境变量的配置

- path环境变量

- 告诉操作系统到哪里去找javac可执行程序
- 配置

- classpath环境变量

- 给jvm用，告诉jvm到哪里加载字节码文件
- 配置

day 2

```
// 在java语言中，一个java文件中只能定义一个被  
// public 修饰的类，且被public 修饰的类的类名，必须和Java文件的文件名相同
```

在idea中，如果要写代码：

1. 在idea中，首先新建一个project（idea的一个窗口中，一次只能显示一个project的内容）
2. 一个project就相当于一个工作空间(workspace)，一个project中可以有多个moudle，
多个moudle之间相互独立
3. 可以在一个moudle下写java代码

关键字 标识符

命名

常见命名规则：驼峰命名

包命名：就类似于在操作系统中，以文件夹的形式组织文件，在java语言中，以包来组织java中的类
关于包来说，为了防止类的命名冲突，一个包下不可以定义同名的类，但是不同包可以定义同名的类

如果，不同的coder，定义了相同的类名此时，只要保证同名的类在不同的包下就可以
也就是说，解决类的同名问题依靠的是包名的不同

为了保证包名的唯一，以域名(唯一的)反转的形式来命名包

baidu.com

com.cskaoyan.name

com.zs.name

单级 test 一个全部小写的单词

多级包 com.cskaoyan.name 以域名反转的方式来命名，单词全部小写，单词之间以.分隔

类和接口命名

单个： 首字母大写，其余字母全部小写 Student

多个单词： 每个单词首字母大写，其余字母全部小写 JavaBasic MaxAge

变量和方法的命名：

单个： 所有字母小写 value

多个单词： 第一个单词首字母小写，从第二个单词开始，每个单词首字母大写 intValue

常量的命名：所有字母全部大写

单个： 单词的所有字母全部大写即可 MAX IP NONE

多个单词： 每个单词全部大写，单词之间以_来分隔： MAX_AGE MAX_VALUE IP_ADDRESS

注释 先写注释再写代码

```
* @version 1.0
*
* 注释：用于解释说明的文字
*
* java注释的分类：
* 1. 单行注释
*/
// 2. 多行注释（不能嵌套使用）
// 3. 文档注释(仅从注释的角度来说他们没有任何区别，但是文档注释有一个特殊的效果) /**注释内容*/
/*千万不要忽视注释的作用！！
1.写代码
2.项目维护，阅读代码*/
```

//我通过class关键字定义了一个类，它的名字叫Demo1Note
public class Demo1Note {

```
    public static void main(String[] args) {
```

//千万不要忽视注释的作用，尤其是对于初学者。|

常量：

常量：在运行过程中，其值不会发生改变的量

常量的分类：

1. 字面值常量

2. 自定义常量（面向对象部分讲）

字面值常量：

字符串常量：双引号引起的内容 "wangdao" 代表固定的字符序列 wangdao

整形常量：所有整数 1, 2

小数常量：所有小数 0.1 3.2

字符常量：用单引号引起的内容 'a' '我'

布尔常量：只有 true or false

空常量： null (面向对象)

进制

```
*  
* Java语言中表示不同进制的语法  
* 二进制：由0,1组成，以0b开头，比如0b1100  
* 八进制：由0~7组成，以0开头，比如014  
* 十进制：由0~9组成，默认10进制，比如12  
* 十六进制：由0~9, A~F(或a~f)表示0~15，以0x开头  
*/  
ublic class Demo2Int {  
  
public static void main(String[] args) {  
  
    //12对应的2进制表示 以0b开头  
    System.out.println(0b1100);  
  
    //八进制的表示 以0开头  
    System.out.println(011);  
  
    //10进制的输出  
    System.out.println(11);  
  
    //16进制 以0x开头  
    System.out.println(0x11);  
}
```

整数：正负数在计算机中都是补码表示 运算起来会比较快

数据类型转化

```
//  
byte b1 = 1, b2 = 2, b;  
//  
b = b1 + b2; //两个byte类型的值参与运算，他们都会首先被转化成int  
//这是因为，编译器简单的判断了一下，不会超出byte类型变量的表示范围  
b = 1 + 2;  
//b = 202 + 1;  
}  
  
}
```

原码补码

回忆一下，刚才我们针对进制讲的例子，都是针对正整数的例子？

1234 -1

负数在计算机中如何表示呢？

正和负两种不同的状态，可以用一位二进制数，约定0表示正号，1表示负号，最高位

假设字长为8位，8位二进制来表示一个数

0,0001100

当给二进制表示，引入符号位之后计算机中就可以同时表示正负数

1. 把一个固定字长的二进制数的最高位，当做是该二进制数符号位 → 原码表示

原码表示

0,0001100 - 0,1110000

我们如果计算机中，如果直接用原码来计算数值位，和符号位计算分开来计算，
计算太慢了，不方便

之所以引入补码的原因，是因为一旦二进数用补码表示，
符号位和数值位可以一起参与运算。

首先，二进制的补码表示和原码表示有关系

1. 对于一个正数的原码表示，它的原码表示就是它的补码表示

2. 对于一个负数的原码表示，它的源码表示和补码表示有如下转化规则：

a. 原码表示的符号位不变，其余各位，依次取反(0变1,1变0)

b. 末尾 +1

原码 → 补码
-12 → 1,0001100 → 1,1110011
+1
1,1110100

补码对应的原码表示？ 因为原码和补码 它们互为补码
对补码求补 → 补码的补码 → 补码对应的源码

1,1110100 → 1,0001011
+1
1,0001100

0 原码表示 正0: 0,0000000
负0: 1,0000000

补码表示
正0: 0,0000000
负0: 1,0000000 → 1,1111111
+1
0,0000000

在补码中0，只有一种表示方式 0,0000000 -128 = 127
1,0000000 规定指定字长的补码表示中 多表示一个负数，-2^k (字长-k)

最高位为1，区域各位都是0

让改补码表示多表示一个负数 -128

day 3

+

```
* 加法
    正数
    字符串拼接:
    "hello" + "world" ="helloworld"

*/
public class AddOperator {

    public static void main(String[] args) {
        //1加法
        int a = 1;
        int b = 2;
        System.out.println(a + b);

        //2. 表示正数的正号
        a = +1;

        //3. 字符串拼接
        // 操作数1 + 操作数2 两个操作数中, 只要至少有一个是字符串, +操作执行的就是字符串拼接
        System.out.println("hello" + 'a' + 1); // helloa1
        System.out.println('a' + 1 + "hello"); // 98hello
        System.out.println("5+5=" + 5 + 5); //5+5=55
        System.out.println(5 + 5 + " = 5 + 5"); //10=5+5
    }
}
```

& vs &&(短路)

<< 左移结果需在数据范围内

>>有符号 填充符号位

>>>无符号 填充0

交换两个数的四种方法

```

//2. 请自己实现两个整数变量的交换(才是开发中常用的)
int a = 10;
int b = 200;
//两个变量的交换
int tmp = a;
a = b;
b = tmp;
System.out.println("a = " + a + "----- b = " + b);

//变量交换的实现方式2: //ctrl + alt + l
a = 10;
b = 200;
a = a + b;
b = a - b; // a + b - b = a
a = a - b; // a + b - a = b
System.out.println("a = " + a + "----- b = " + b);

//实现方式3:
a = 10;
b = 200;
a = a ^ b;
b = a ^ b; // a ^ (b ^ b) = a
a = a ^ b; // a ^ b ^ a -> [a ^ a] ^ b
System.out.println("a = " + a + "----- b = " + b);

//实现方式4
a = 10;
b = 200;
a = (a+b) - (b=a);
System.out.println("a = " + a + "----- b = " + b);

```

sc.nextLine() 以回车为分隔符x

day 4

if语句后不管单条还是多条语句 均用括号

三目运算符结果一定要是一个值

```
//三目运算符的结果一定是一个值
a > b ? System.out.println("大者是" + a) : System.out.println("大者是" + b);
```

Not a statement

在绝大部分场景下，三目运算符和if双分支选择结构，都可以相互替代。

但是，如果选择结构执行的仅仅只是一个操作，没有返回值，此时if双分支选择结构不可以和三目运算相互替代。

switch case 穿越

if的使用场景

针对结果是boolean类型的判断 if(true)

分支对应多个表达式的多个取值

switch的使用场景 | I

针对结果是固定类型的判断 switch(byte, short, int, char, String)

表达式的取值范围只是固定的离散值集合，分支对应表达式的某一种取值

```
 */
public class compare {
```

```
    public static void main(String[] args) {
```

//分支对应多个表达式的多个取值

```
        int x = 0;
```

```
        int y = 100;
```

```
        if (x > 0) {
```

// x > 0

```
    } else if (y < 0) {
```

// 0=< y <0

```
}
```

```
    switch (x) {
```

case 0: //一个值对应一个分支

```
        break;
```

```
}
```

```
compare
```

Week 2

day 1

循环结构：初始化语句 条件判断语句 循环体语句 循环控制语句

for i 代码自动补全 for loop

for each element in collection

```
1 for (int num : nums)
2     do something with num;
```

带标签的break

```
outer:
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if( j == 2) {
            break outer;
        }
        System.out.println("i = " + i + " -- " + "j = " + j);
    }
}
```

outer: 给外层循环命名

continue outer vs break

continue outer是进入下一次标记的外层的迭代

break是跳出当前层的循环

```
outer:  
for (int i = 0; i < 3; i++) {  
  
    for (int j = 0; j < 3; j++) {  
        if( j == 1) {  
            continue outer;  
            //break;  
        }  
        System.out.println("i = " + i + " -- " + "j = " + j);  
    }  
}
```

方法

代码复用

调用过程

根据名字方法

传递参数 值传递

计算

返回结果

调用方法

接收调用 输出调用 直接调用

方法定义内不能再定义方法

方法重载

名字同 参数列表不同 返回类型可以不同 (因为可以直接调用 不会显示返回值 编译器没法判断)

(编译器区分方法) 方法签名 : 方法名 + 参数列表

传参时 转化最近的

```
byte -> short -> int -> long
```

平时都是直接到 int

★★自动（隐式、默认）类型转换与强制（显式）类型转换★★

1) boolean类型不参与转换

2) 默认转换

A:从小到大

B:byte,short,char --> int --> long --> float --> double

C:byte,short,char之间不相互转换，直接转成int类型参与运算。

3) 强制转换

A:从大到小

B:可能会有精度的损失，一般不建议这样使用。

C:格式：

目标数据类型 变量名 = (目标数据类型) (被转换的数据);

day 2

数组

随机存取原理：

(数组中只放同一种数据类型原因)

1. 相同数据类型 访问变量的时候，存储空间有地址，找到变量地址，访问

addr = baseAddr + (数组单元编号)*数据宽度

数组初始化：

(创建对象) 先在堆中申请分配存储空间 JVM再赋初值 0, false, null

再把数组的首地址返回给引用类型的变量

```
1 | int[] array = new int[3];
```

动态(=new int[size]) 可确定长度

静态(={elements}) 可确定数组元素

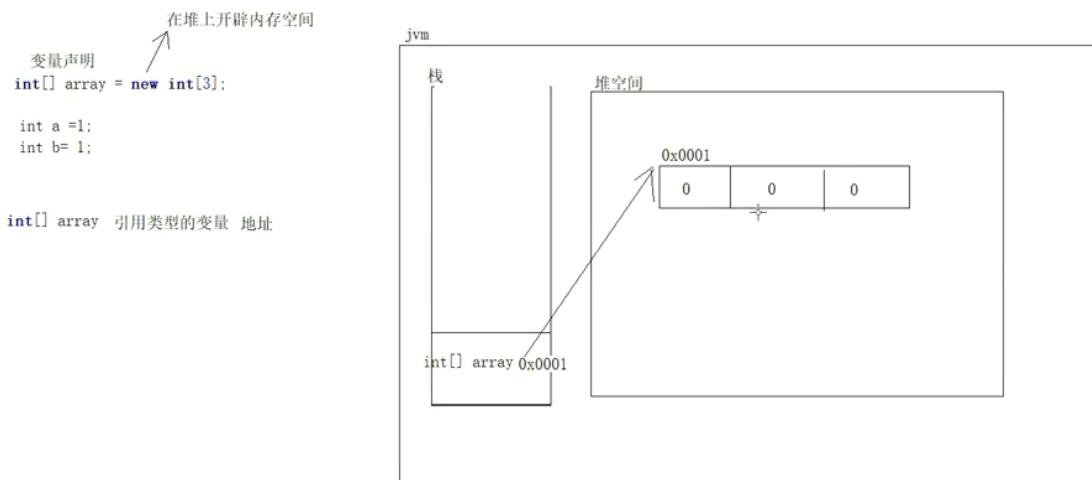
JVM内存分区

栈 存储局部变量 迄今为止我们代码中几乎所有的变量都在栈
堆 存储new出来的东西 数组，对应的存储空间在堆中
方法区 (后面讲)
本地方法栈 (系统相关) java语言 可以调用c/c++
程序计数器 指明代码执行的位置

堆与栈

栈上变量		堆上的变量
变量类型	局部变量	new出来的东西
初值	局部变量的初始值，coder自己给变量赋初值	天然有初始值，jvm赋予的 整数型(byte int等): 0 字符型 char: '\u0000' 码值0对应字符 小数型 float, double: 0.0 布尔类型: false 引用类型: null
内存管理方式	管理和作用域有关	内存管理就和栈内存的管理方式 垃圾回收期 ↗

引用变量是局部变量，局部变量就是存储在栈上的



输出数组名

```
[I@4554617c  [表示一维数组 I 整数类型 4554617c 数组实际存储的首地址]
```

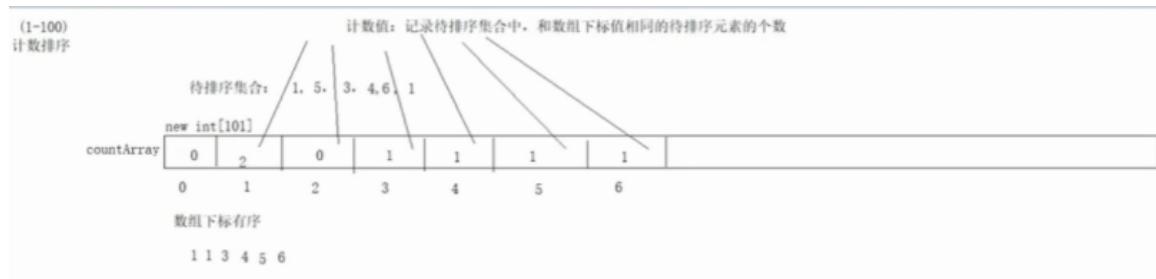
null:一个不存在的内存地址

数组的属性：length(因为数组也是一个对象)

Arrays.toString(): 把一个数组转成一个字符串

计数排序：空间换时间 时间复杂度O(n) 空间复杂度O(n)

但是要预先知道待排序列的数值范围



```
1 // 传入的数组中的数值范围必须在0-a.length之间
```

```

2 public static void countingSort(int[] a) {
3     int[] count = new int[a.length + 1];
4     // 遍历原数组
5     for (int i = 0; i < a.length; i++) {
6         count[a[i]]++; // 记录a[i]的出现次数
7     }
8     int index = 0;
9     // 遍历计数数组
10    for (int i = 0; i < count.length; i++) {
11        while (count[i] != 0) { // 如果数值i出现过
12            a[index++] = i; // 重新建表
13            count[i]--; // i出现次数减1
14        }
15    }
16 }

```

two pointers:记得改变下标值！

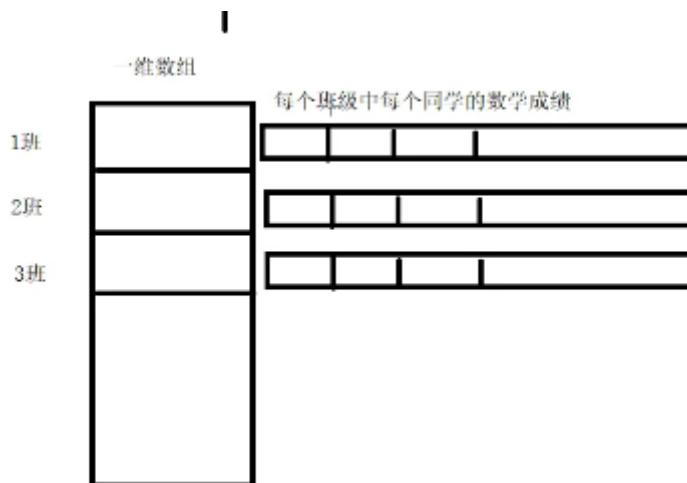
day 3

二维数组

引入：

处理两个维度

本质上还是一个一维数组，只是数组中的每个元素是指向一个一维数组的引用(首地址)

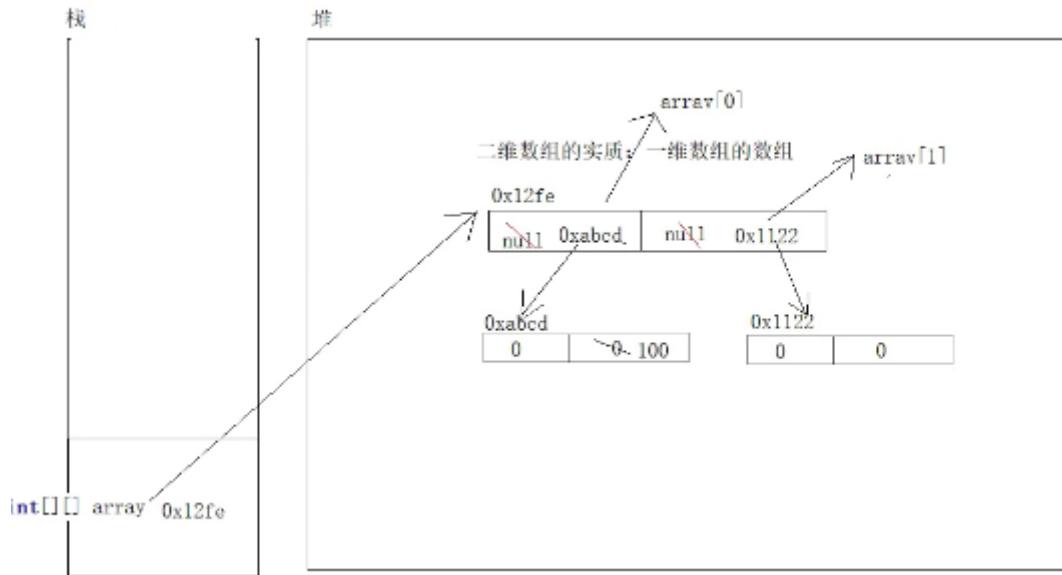


二维数组的内存映像

```

1 | int[][] array = new int[2][2];

```



遍历：对每个元素访问一次且仅一次

day 4

参数传递：

都是值传递 形参是实参的一个副本 形参的改变不会影响到实参 但当传递的是引用时 可以通过这个引用去访问或者修改堆中的对象的内容（如果有权限的话）

栈的内存管理

一个执行中的方法在栈中会对应一个栈帧

栈帧在方法被调用执行时分配，在执行完毕后被销毁

同一个方法多次执行时，每次执行都会分配栈帧（没有递归出口时 栈溢出的原因）

局部变量：定义在方法体中的变量（形式参数也是属于局部变量）

stackoverflow(也是问答网站)

栈空间的内存大小有限



汉诺塔算法

```
*  
* @param n      待搬运的圆盘数量  
* @param start  待搬运的圆盘所在的杆的名字(起始杆)  
* @param end    待搬运的圆盘, 所要搬运到的目标杆的名字  
* @param middle 在本次搬运过程中所使用的辅助杆的名字  
*/  
public static void hanoi(int n, char start, char end, char middle) {  
  
    // 递归出口  
    if (n == 1) {  
        // 直接解决规模为1的子问题  
        System.out.println(start + " -> " + end);  
        return;  
    }  
  
    // 此时问题还可以分解为子问题  
  
    // 将最大的圆盘的之上的n-1个盘, 搬运到辅助杆上  
    hanoi(n - 1, start, middle, end);  
  
    // 解决, 最大的哪一个盘的搬运问题  
    System.out.println(start + " -> " + end);  
  
    // 将辅助杆上的n-1个盘, 搬运到目标杆上去(以原来的start杆为辅助)  
    hanoi(n - 1, middle, end, start);  
}
```

递归

分而治之 大规模问题转换为小规模的问题

自己调用自己

递归出口（先写）+ 递归公式

递：依次次向前询问

归：从第一排的同学（递归出口），一次向后传递确切排数

面向对象基本概念

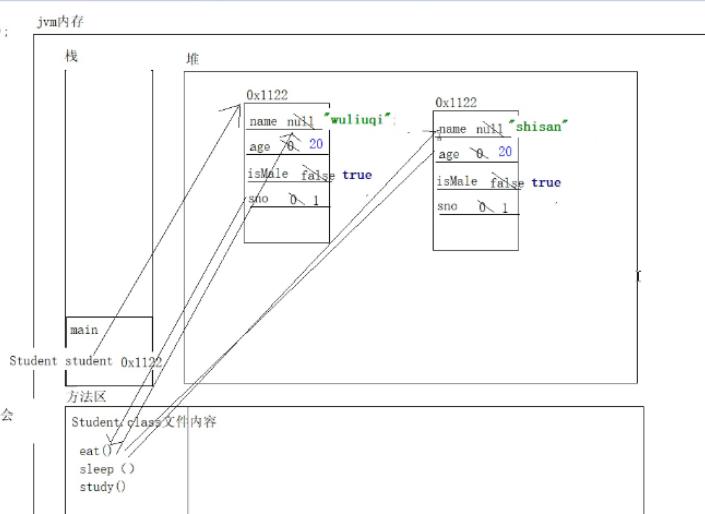
对象：属性 + 行为

类：向JVM描述同种类型的对象的属性与行为



```
//定义学生类，描述所有学生对象
class Student {
    //姓名属性
    String name;
    //年龄属性
    int age;
    //性别
    boolean isMale;
    //学号
    int sno;

    //吃饭行为
    public void eat() {
        System.out.println("eating");
    }
    //睡觉行为
    public void sleep() {
        System.out.println("sleeping");
    }
    //学习行为
    public void study() {
        System.out.println("studying");
    }
}
```



day 5

面向对象的基本语法

同一个包 (类似于C++中的namespace) 下的类不能同名

首次创建一个类的对象(或者访问一个类的静态成员)的时候 会触发该类的类加载 加载类的字节码文件到方法区中

(因为类是我们自定义的数据类型 , JVM需要先认识它)

堆中的变量天然有初值 (JVM给赋值的) : null false 0..

创建类对象时的内存映像

3个引用变量 2个对象

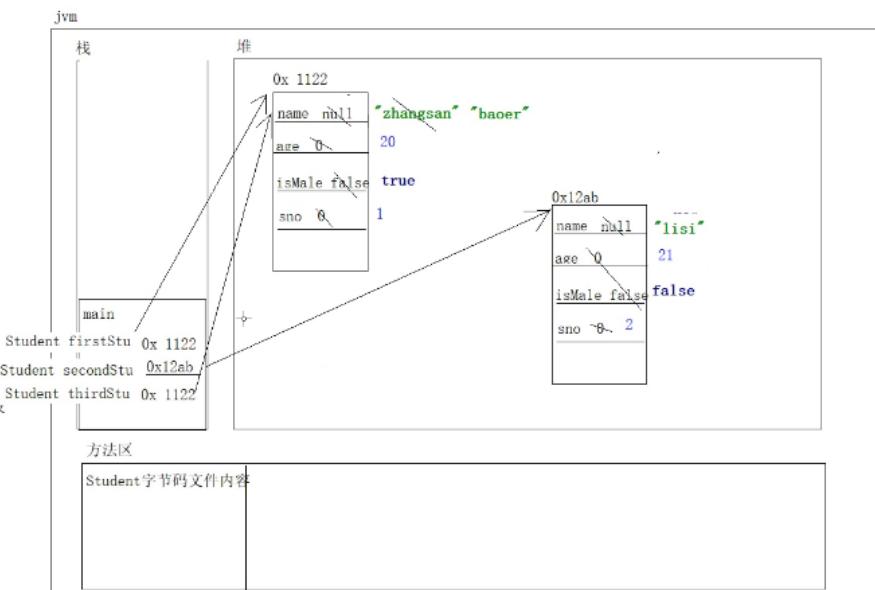
thirdStu and firstStu指向的是堆中的同一个对象

```
// 创建第一个Student对象
Student firstStu = new Student();
firstStu.name = "zhangsan";
firstStu.age = 20;
firstStu.isMale = true;
firstStu.sno = 1;

// 创建第二个Student对象
Student secondStu = new Student();
secondStu.name = "lisi";
secondStu.age = 21;
secondStu.isMale = false;
secondStu.sno = 2;

//所谓的第三个对象
Student thirdStu = firstStu;
thirdStu.name = "baoer";
```

在首次创建, 一个类的对象的时候, 会触发该类的类加载



面向对象特殊语法

局部变量与成员变量比较

- 所处的位置不同：局部变量定义在方法体内，成员变量定义在类中方法体外
- 在内存中的存储的位置不同：局部变量存储在方法对应的栈帧中，成员变量存储在堆中的对应的对象中（局部变量：定义在方法中的形参或者变量）
- 初始化值不同：局部变量没有初始值，成员变量天然有初值
- 生命周期不同：局部变量依附于栈帧而存在，成员变量依附于对象而存在

JVM创建对象的固定流程

只是 new ClassName(); 这里 赋值地址不属于创建对象 创建对象完后才传递地址值

- 在堆中申请对象的内存空间

2. 给对象的成员变量赋默认初值 (0, false, null)
3. 调用构造方法完成成员变量初始化的工作

构造方法

与类同名 且没有返回类型 可以重载的一个方法 (通过实参列表与形参列表的匹配确定具体使用那个构造方法)

作用 : 在创建对象时完成对象的成员变量初始化的工作 (创建对象的最后一步)

语法 : public + ClassName

实际上是第二次赋值 第一次由JVM自动进行

当在类中没有定义任何一个构造方法时 JVM会自动添加一个默认的无参构造方法

签名 : 方法名 + 参数列表

无参构造方法使用场景 : 希望自己去定义对象成员的默认取值 而不是用系统分配的

建议 : 不管需不需要都添加一个无参的构造方法

this (实际开发中使用)

代表指向当前对象的引用

当前对象的含义 :

构造方法中 : 在创建对象时 , 正在创建的那个对象 (JVM已经赋过一遍默认初值了)

成员方法中 : 调用本方法的对象

```
1 | public Student(int age) {  
2 |     this.age = age;  
3 | }
```

成员变量的隐藏问题 : 局部变量与成员变量与同名 成员变量会被局部变量隐藏起来

this的作用

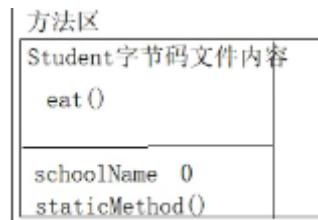
1. 解决成员变量被局部变量隐藏的问题
2. 访问当前对象的成员 (变量或者方法)
3. 访问当前对象的构造方法

```
1 | public Student() {  
2 |     this("", -1, true, -1);  
3 |     // 在无参构造方法中调用四参的构造方法来设定对象成员变量的默认值  
4 |     // 必须在构造方法的第一条语句  
5 |     // 代码重用  
6 | }
```

static

1. 被static修饰的成员 , **被类的所有对象共享**(所以也可以通过对象调用静态方法)

2. 可直接通过类名访问到静态成员（非静态成员要先创建对象）
3. 随着类的加载而加载（类加载的触发：首次创建一个类的对象或者是首次访问一个类的静态成员时）
4. 优先于对象而存在（跟类一起被加载到方法区中）



注意事项：

1. 在静态上下文中无法访问**当前对象**的非静态成员（不能使用this），但是可以自己new出一个对象，访问这个创建的对象的非静态成员（因为静态方法执行的时候当前对象可能都不存在）
2. 不管是静态方法还是非静态方法，方法体中都不能用static修饰变量（因为方法体存储在栈帧，而静态变量存储在方法区：类加载先进行）
3. 静态方法通常是作为工具方法来使用（访问的都是参数的数据：Arrays.toString()）

day 6

静态成员变量与普通成员变量

1. 所属不同：类/对象
2. 内存中的位置不同：方法区的静态区/堆内存
3. 在内存出现的时间不同：存在依附于类（类加载时），存在依附于对象（对象创建时）
4. 调用不同：静态成员变量：类名或对象 | 普通成员变量：对象

代码块

局部代码块

1. 声明的方式与出现的位置：声明方式{}，出现在方法体中
2. 执行时机：随着其所在方法的执行而执行

可忽略不计的优点：限定变量的生命周期，及早释放，提高内存利用率（要求代码的可维护性）

在嵌套的代码块中不能定义同名变量（Java语言的要求）

给成员变量赋初值的三种方法：（JVM赋默认初值后）

1. 成员变量初始化语句（先一般而言成员变量都定义在构造代码块之前）
2. 构造代码块（中）
3. 构造方法（后）

构造代码块

1. 声明方式和出现位置：声明方式{}，出现在类中方法体外

2. 执行时机：每次在创建对象的时候执行

使用场景：

1. 把多个构造方法中相同的代码放到构造代码块中执行，并且在构造方法前执行（避免代码冗余）
2. 用来初始化对象成员变量的值

静态代码块

1. 声明方式和出现位置：声明方式static{}，出现在类中方法体外

2. 执行时机：随着类加载而执行，同一个类在一个JVM中最多只会被加载一次，所以静态代码块也只会被执行一次

使用场景：最多只需执行一次的代码（如：连接数据库）

同步代码块（多线程讲）

package

包：类似于操作系统中的文件目录，用来组织类

第一行使用package声明可以使文件中定义的类成为指定包的成员

语法 package + 包名

通常以组织机构的域名反转形式作为其所有包的通用前缀：com.somecompany.apps

注意事项：src下的文件没有包名，属于默认包

2

import

类的完全限定名：包名.类名

在类体中使用了与当前类不同包的类名时：

1. 使用不同包类的完全限定名
2. 使用import声明，为编译器找到该类的定义信息

默认使用当前包中的类 导入后使用导入的包中的类

import注意事项

1. import声明在package声明之后，在类声明之前，import + 类的完全限定名
2. java.lang包中的类被隐式导入，可直接使用其中的类
3. 智能导入方式：import 包名.*;

根据需要导入，避免使用多条import声明

（智能导包不能嵌套导包，且是按需导包：如果当前包下有这个类，就不会再导入同名的类）

Week 3

day 1

面向过程思想与面向对象思想(了解)

面向过程思想（动词的集合）：程序功能由一系列有序的动作来完成

面向对象思想（名词的集合）：程序由一系列对象和对象间的消息组成

```
/*
    以面向过程的思想，描述 把大象装进冰箱
    1. 打开冰箱
    2. 把大象塞进冰箱
    3. 关上冰箱
*/
public void procedure() {

    //1. 打开冰箱
    //2. 把大象塞进冰箱
    //3. 关上冰箱
}

/*
    以面向对象的思想，描述 把大象装进冰箱
    给对象发送消息：在对象上调用其某一个方法
*/
public void object() {

    // 冰箱对象, in(大象对象)
```

消息：

方法调用就是给对象发消息

在一个对象上调用了某个方法，对方法调用，就被看成是发送给这个对象的消息，对象收到这个消息后，就会执行相对应的动作(执行方法)

面向过程程序设计：函数（方法）是程序的基本单位

面向对象程序设计：类是程序的基本单位

访问权限修饰符

对类中成员的访问控制（成员变量与成员方法一样）

1. public : 任何类都可以访问，实际上就是没有访问控制
2. protected : 定义成员的类中和同包中的其他类或者不同包中的子类可访问
3. default: 定义成员的类中和同包中的其他类可访问
4. private: 只有在定义成员的类中可访问

对类的访问控制(目前接触到的就两个)

public: 任何类都可以访问

default:同包中的类可以访问

private修饰符的作用 : (例如 : private方法)

使用户不要触碰不该触碰的代码

类库设计者可以安全的修改实现细节 (不会影响到其他的类 , 因为不会被其他的类调用)

看源码 不要过多关注private的方法 (只有在定义的类中能调用)

封装

了解即可

信息隐藏 , 将数据和基于数据的操作封装在一起

外部只有通过被授权的操作才能访问到内部的数据

对于private成员变量 , 定义public的set方法和get方法

好处:

1. 满足在某些场景下对私有成员变量进行访问的需求
2. 实现了对私有成员变量读写的分离
3. 在set方法中可以通过代码控制别人的修改 (加控制逻辑)

给成员变量赋值 :

1. 无参构造方法 + setXXX()
2. 带参构造方法

继承

第一层含义 : 类定义代码的代码复用

第二层含义 : 用继承描述类之间的“is-a”关系

继承概述

1. 不劳而获
2. 描述了不同数据类型之间“is-a”的关系 (student is a person)
3. 子类不写额外的代码就可以拥有父类中的成员

语法 : class 子类名 extends 父类名 { }

继承的优点

1. 代码复用
2. 提高了代码的可维护性 (减少了代码冗余 , 但是也是双刃剑)
3. 弱化了Java中的类型约束 (可以让父类引用指向子类对象 , 多态的一个前提条件)

继承的缺点

父类中的修改会不加选择地会出现在所有的子类中

继承的局限性

只能单重继承：一个类最多只能有一个父类

但是可以间接继承父类的父类中定义的成员

继承的注意事项

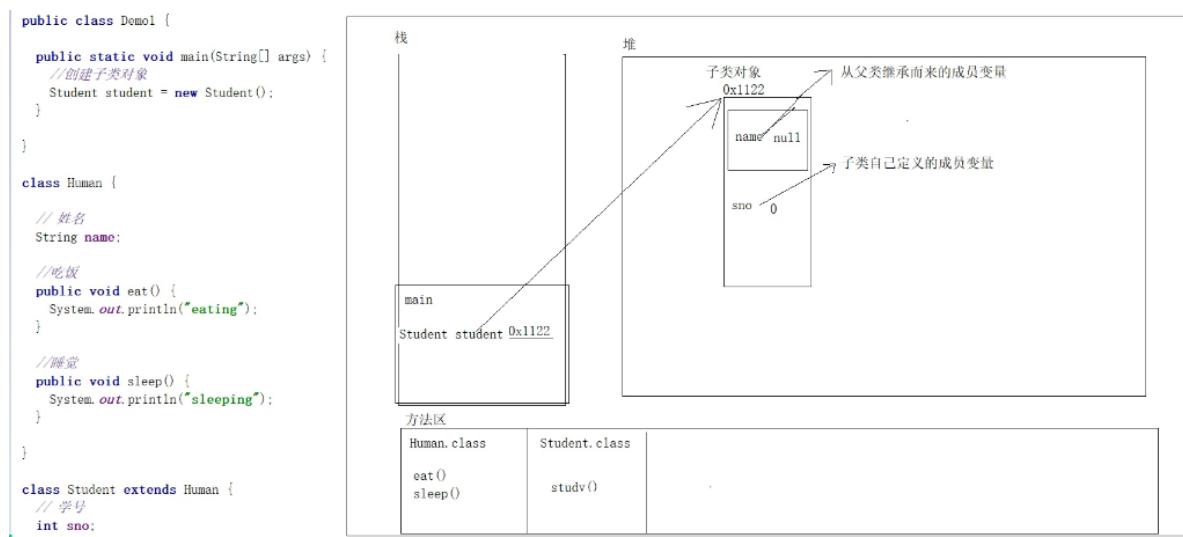
1. 子类不能访问父类中的私有成员
2. **子类不会继承父类中的构造方法** (所以需要自己重新定义)

子类对象的初始化

过程：

(如果是首次的话) 先加载父类，再加载子类，再初始化父类对象，最后再初始化子类对象

有继承时 创建对象的内存映像



问题：如何保证父类的构造方法先于子类的构造方法执行(先父后子)

方式：

隐式初始化：(先调用父类构造方法，JVM隐式完成)

1. 父类提供了默认的无参构造方法
2. 且子类的构造方法中没有显式调用父类的其他构造方法
3. 那么在JVM会自动先执行父类的构造方法，在执行子类的构造方法

显式初始化：

coder通过代码来保证父类的构造方法先于子类的构造方法执行

super关键字：代表对父类对象的引用

this vs super

this	super
表示对当前对象的引用	表示对父类对象的引用
用于访问当前对象的成员(变量或者方法)	用于访问父类对象的成员
用于在构造方法用调用当前类中定义的其他构造方法this(实参列表) (在第一句)	用于在子类的构造方法中调用父类的构造方法super(实参列表) (在第一句)

注意事项：

1. 保证子类中的每个构造方法都要满足子类对象的初始化流程（先父后子）
2. 子类构造方法中通过super调用父类的构造方法时，super语句必须在子类构造方法的第一条语句
3. this(),super()都可以出现在构造方法的方法体中，都需要出现在第一条语句的位置，且他们不能同时存在

day 2

protected跨包访问的细节（子类访问父类，跨包子类可见，是指跨包子类的对象可见）

在跨包的子类的非静态成员方法（属于对象）中可以**直接访问**或者是通过**创建子类对象**访问父类的**protected成员**

静态上下文和通过创建父类对象均访问不到

```
1 package day7.onePackage;
2
3 public class Father {
4     protected int protectedValue = 10;
5
6     protected void protectedMethod() {
7         System.out.println("this method is protected");
8     }
9 }
10
```

```
1 package day7.anotherPackage;
2
3 import day7.onePackage.Father;
4
5 public class Son extends Father {
6     public static void main(String[] args) {
7         // 在子类体内
8         // 可以通过创建子类对象访问父类的protected成员
9         Son son = new Son();
10        System.out.println(son.protectedValue); // 10
11 }
```

```
11     son.protectedMethod(); // this method is protected
12
13     // 不能通过创建父类对象访问父类的protected成员
14     Father father = new Father();
15     System.out.println(father.protectedValue); // wrong
16     father.protectedMethod(); // wrong
17
18     // 不能在静态上下文直接访问父类的protected成员
19     System.out.println(protectedValue); // wrong
20     protectedMethod(); // wrong
21 }
22
23 public void sonNonStaticMethod() {
24     System.out.println(protectedValue); // 10
25     protectedMethod(); // this method is protected
26 }
27 }
```

父类域的隐藏

在子类中定义了与父类中的成员变量同名的成员变量

结论：

1. 在子类对象上，调用子类中的方法访问，访问到的就是子类中定义的同名成员变量
2. 在子类对象上，调用父类中的方法访问，访问到的就是父类中定义的同名成员变量

tip:在子类方法体中，可以通过super关键字访问到父类中定义的同名成员变量

访问规则

通过子类对象调用子类方法，访问成员变量值的规则（那就是非静态方法）

首先在子类对象中，查找子类自己定义的成员变量中有没有目标成员变量

如果有就直接访问

如果没有就到父类对象上查找目标成员变量(因为有些成员变量是父类中定义的而子类中没有定义)

tip:子类的静态方法中不能使用super，因为可能对象都不存在

方法的覆盖

在子类中定义了与父类方法声明一模一样的方法

不管是在子类类体中还是在父类类体中，访问到的都是子类中定义那个成员方法

tip:可以在子类对象中，通过super关键字访问到父类中定义的成员方法

覆盖：在子类对象上，在父类方法体中，访问同名成员方法，调用的还是子类的同名成员方法

覆盖原因：



在**子类对象**上调用方法 查找目标方法过程：

是先在子类中查找，如果没找到才到父类中查找，所以如果子类中定义了该成员方法，父类的同名成员方法就不可能被调用

使用场景：

用于在子类中修改父类方法的方法实现

(以后可以用 -> 设计模式：经验的复用 设计类的经验)

覆盖条件 (方法声明一模一样)

方法声明 = 访问权限修饰符 + 返回类型 + 方法签名 (方法名 + 参数列表)

1. 访问权限：子类中定义的方法的访问权限要不小于父类方法的访问权限 (public > protected)
(因为有可能通过父类引用调用该方法)
2. 返回类型：基本数据类型，子类与父类要相同
引用数据类型，子类的返回类型可以是父类返回类型的子类
(因为可以把子类看做是一个父类)
3. 方法签名必须一样

注意事项 (不能被覆盖的父类)

1. 私有成员方法
2. 被修饰为final的成员方法
3. 静态成员方法
4. 构造方法 (不能被子类继承，更别说覆盖了)

final关键字

1. 修饰类：该类不能被继承
2. 修饰方法：该方法不能被覆盖
3. 修饰变量：变量就成了自定义常量，只能被赋值一次
 1. 修饰局部变量：在使用之前被赋值，且仅能被赋值一次
 2. 修饰成员变量：在对象创建完之前被赋值，且仅能被赋值一次 (coder显式赋值，不包括VM的默认赋值) (一般是:private + final)

tip: 被final修饰的变量也称之为自定义常量, final int MAX = 10;

add:

```
public static void main(String[] args) {  
    final int a = 10;  
  
    // final 仅仅修饰引用变量  
    final A obj = new A();  
  
    // 引用变量被赋值之后其值不能变:  
    //obj = new A();  
  
    // 但是final引用变量所指向的对象  
    obj.i = 10;  
    obj.i = 100;  
}  
}  
}  
  
class A {  
    int i;  
}
```

多态

同一个对象的**行为**，在不同的条件下，会表现出不同的效果

(翻译：同一个父类引用，在指向不同的子类对象时，会调用不同的方法实现)

前提条件：继承 + 方法覆盖 + 父类引用指向子类对象

tip:多态针对的是行为 而不是属性

成员访问特点

成员变量：编译看左边，运行看左边

成员方法：编译看左边，运行看右边

1. 编译看左边：用父类引用指向子类对象时，通过引用变量可以访问到的子类成员的范围是由引用类型来决定的（即不能通过父类引用，来访问子类中自身特有的成员）（遥控器与电视机）
2. 对于成员变量，运行看左边：成员变量作为对象的属性，用来描述对象的外貌，子类引用赋值给父类引用，相当于给子类对象披上了父类类型的马甲，它的外貌特征表现出来的就是父类的外貌特征（成员变量/属性）
3. 对于成员方法，运行看右边：（多态）通过引用变量调用的方法，由其所指向的对象类型决定

运行：实际运行的效果

```
// 演示多态的基本效果  
//basic();  
  
//再次演示多态的效果  
//testBasicPolyporphism();  
  
//测试多态成员的访问特征  
Animal animal = new Dog();  
//System.out.println(animal.name); //xxxAnimal  
  
animal = new Cat();  
//System.out.println(animal.name); //xxxAnimal  
  
//测试多态中成员方法的访问特征  
Animal an;  
  
an = new Dog();  
an.eat(); //狗啃骨头  
  
an = new Cat();  
an.eat(); //猫吃鱼
```

多态的优点

1. 提高了程序的可维护性 (由继承来保证)
2. 提高了程序的可扩展性 (由多态来保证)

多态的缺点：不能通过父类引用来访问子类特有的功能 (因为编译看左边)

引用变量的类型转化 (父类和子类)

子类类型的引用变量 -> 父类类型的引用变量 向上转型 (编译器天然允许)

父类类型的引用变量 -> 子类类型的引用变量 向下转型 (需要用到强制类型转换)

instanceof运算符：判断目标对象是否是指定类型的对象

对象的引用变量 instanceof 目标类型的类名

tip: null instanceof 任意一个类 结果始终为false

```
1 | if (animal instanceof Duck) {  
2 |     Duck duck = (Duck)animal;  
3 |     duck.swim();  
4 | }
```

day 3

抽象类

包含有抽象方法的类 或者 被abstract修饰的类 (包含抽象方法的类必须是抽象类)

抽象类和抽象方法必须用abstract修饰

abstract class 类名 {} (比如说动物就应该是一个抽象类)

public abstract void eat(); (无法具体地确定动物的吃的行为)

特征

1. 抽象类无法实例化：无法 new 抽象类实例，但是可以通过抽象类类型的引用指向一个具体子类对象
2. 抽象类的子类：可以是具体类（必须覆盖实现抽象类中所有的抽象方法，否则无法调用方法）或者是抽象类（没有覆盖实现，只要没有覆盖完）

注意事项

1. 有抽象方法的类一定是抽象类
2. 抽象类不一定有抽象方法（非抽象方法，用于代码重用）

抽象类的成员特点

1. 构造方法：同普通类
2. 成员变量：同普通类
3. 成员方法：
 1. 抽象方法：仅仅是为了声明有这样的行为，而无法确定具体的实现
 2. 非抽象方法：代码复用

抽象类不能实例化，为什么要有构造方法

因为抽象类中也可以定义成员变量，所以在创建子类对象时也需要给这些抽象父类中的成员变量进行初始化，

而初始化的工作就交给抽象父类的构造方法完成

abstract不能和哪些关键字共存

1. private
2. final
3. static

冲突原因：调用抽象方法时，都是通过多态，实际调用的是具体子类中的覆盖实现，而被private, final, static修饰的方法都不允许在子类中被覆盖，那就意味着这些方法只有声明而没有实现，无法运行

tip:方法 + abstract：强制规定子类必须去实现这些抽象方法

day 4

接口

接口的语法

1. 格式 : interface 接口名 {}
2. interface也可用来表示一种数据类型 : 类定义的是一个数据集以及基于这个数据集的一组操作 , 且这组操作之间是有间接关系的 , 因为访问的是同一个数据集 ; 接口描述一组具有特殊功能的行为 , 这些行为之间可以完全没有任何关系
3. 类实现一个接口: class 类名 implements 接口名 {}, 实现关系实质上是一种继承关系
4. 接口不能直接实例化 , 但是可以间接实例化: 接口引用指向一个接口实现子类对象
5. 接口的子类可以是抽象类也可以是具体类 (要求实现接口中的所有抽象方法)

接口的特点

1. 没有构造方法
2. 成员变量只能是自定义常量 , 默认修饰符为 public static final
3. 成员方法只能是抽象方法 , 默认修饰符 public abstract

Java语言的多重继承

1. 接口与接口之间可以多重继承
2. 一个类可以实现多个接口

完整的类定义语法 :

```
class 类名 extends 另一个类的类名 implements 接口1, 接口2, 接口3 {}
```

抽象类与接口的的比较

	抽象类	接口
成员区别	有构造方法、成员变量：变量、成员方法：有抽象方法和非抽象方法	没有构造方法、成员变量：自定义常量(public static final)、成员方法：只有抽象方法 (JDK7 及以前)
关系区别	类之间：单继承	接口之间：可以多继承 类与接口：多实现
设计理念区别	is-a关系：一个类只能有一个父类，且一个子类对象可以被看作是一个父类对象	like-a关系：实现一个接口的类，它就具有了这个接口所描述的一些功能

tips:

1. 抽象类也可以实现接口(因为抽象类可以有非抽象方法)
2. 接口的访问权限与类相同：public/ default

JDK8及以后接口中可以有方法实现

1. 默认方法(default +)：折中策略 修改接口，为接口添加一些方法实现，同时不会影响到已经实现接口的类（如果声明的是默认的抽象方法的话，所有实现这个接口的类都要去覆盖实现这个抽象方法）
2. 静态方法(static +)：作为工具方法使用，只有在定义静态方法的接口中可以直接调用，其他地方需要通过接口名.静态方法的方式来使用

tips:普通类中子类可以直接调用父类静态方法（因为：被static修饰的成员，**被类的所有对象共享**(所以也可以通过对对象调用静态方法)）

```

/*
public interface JDK8Interface {

    //默认方法(非静态的方法)
    default void defaultMethod() {
        System.out.println("default method");
    }

    //静态方法
    static void staticMethod() {
        System.out.println("staticMethod");

        //在静态方法中调用默认方法 不可以(静态上下文中无法访问非静态的成员)
        //defaultMethod();
    }
}

class TestJdk8Class implements JDK8Interface {

    //在子类类体中调用默认方法
    public void testDefaultMethod() {
        defaultMethod();
    }
}

```

内部类

定义在其他类内部的类

分类：

1. 成员位置内部类（在类中方法体外）
2. 局部内部类（在方法体内）

访问特点



1. 内部类可以直接访问外部的成员，包括私有成员
2. 外部类要访问内部类的成员，必须通过创建内部类的对象

成员位置内部类

（非静态的成员位置内部类依赖于外部类对象而存在，就跟普通的非静态成员一样）

定义在外部类中的成员位置内部类，与类的其他成员一样，非静态的成员位置内部类依赖于外部类的对
象而存在，所以在静态上下文中无法直接访问到外部类的非静态成员位置类，需要先创建一个外部类对
象，通过这个外部类对象来访问

在外部类的外部创建内部类的对象

外部类名.内部类名 对象名 = 外部类对象.内部类对象 (outer.Inner inner = new Outer().new
Inner())

创建好之后就可以通过对对象名，访问内部类的成员

外部类的内部：

```
Inner inner = new Outer().new Inner();
```

成员位置内部类修饰符

default/private/static

静态上下文：静态方法、静态代码块、静态成员位置内部类：均不能直接访问非静态成员（因为非静态成员依赖于对象而存在，静态上下文中可能只有类加载了，而对象还没有创建）

局部内部类

在方法体中定义的类，只有在方法体中通过创建局部内部类对象才能访问（先定义后使用，访问特征跟其他内部类一样）

局部内部类特征

在局部内部类中，只能方法中定义的自定义常量

由于↓

生命周期冲突

1. 局部变量的生命周期与栈帧相关
2. 而局部内部类对象存储在堆上，对象的回收与栈帧的销毁无关（方法运行完，局部变量不在了，但是对象还在）
3. 所以当方法运行完之后，如果还可以通过其他引用找到这个局部类对象，就可以通过这个对象访问局部变量，但此时局部变量随着栈帧一起销毁了

（外部类名）Outer.this表示外部类的当前对象

（就把非静态成员位置内部类当做一个非静态成员来看待就行，依附于对象而存在）

只有在嵌套定义的内部类时使用

Outer.this：外部类的当前对象

Inner.this：内部类的当前对象

```
//补全程序
// 在控制台分别输出10, 20, 30
class Outer {
    public int num = 10;
} class Inner {
    public int num = 20;
    public void show() {
        int num = 30;
        // 类名.this 这种this的写法，它和静态没有任何关系，这样的写法仅仅只是表示this的嵌套层级
        System.out.println(Outer.this.num);
        // 利用this访问内部类中的同名成员变量
        System.out.println(Inner.this.num); //20
        // 直接通过变量名，访问内部类中同名的局部变量num
        System.out.println(num); //30
    }
}
```

心得：

1. 用接口引用指向一个实现它的子类对象
2. 对每个private成员变量 设置get/set方法
3. 非静态成员属于对象/静态成员属于类
4. 在用于输出的方法中用上get方法获取对象相关信息

day 5

匿名内部类对象

本质：

实际上是一个继承了一个类（一般是抽象类），或者是实现了一个接口的匿名子类对象（在其类定义(代码块)中可以重写父类/父接口中的抽象方法）

优点

一般的创建子类对象的方法：

1. 定义子类
2. 创建子类对象

匿名内部类对象（两步化作一步）：

语法：new 父类名或者父接口名() {重写方法}; (new.. 这里出来的就是一个引用，别想太多)

（因为没有定义这个具体的子类（子类没有自己的类名），所以只能用父类/父接口的引用指向它）

想要使用的是子类覆盖了父类中的那个方法

定义子类和创建匿名子类对象同步进行（每次都是定义一个新的子类并创建这个子类对象）

用于每次只需调用单个方法，想要再调用方法的话要重新创建一个匿名内部类对象

前提：存在一个类或者接口（这里的类可以是具体类也可以是抽象类）

匿名对象

new MyClass().first();(创建一个对象，然后直接调用它的成员，并不保存它的引用)

使用场景：在只需要使用一次且一次某个抽象类或者某个接口的子类对象的时候使用（创建完对象后，没有让某个引用变量存储它的引用）

1. 当方法的形式参数是接口类型的时候，可以传入一个匿名内部类对象（因为接口无法直接实例化）
2. 当方法返回类型是接口类型时：return + new 父类名或者父接口名() {重写方法}; (因为return语句在方法的一次执行中最多只会执行一次)

练习

```
*  
 * 要求在控制台输出"HelloWorld"  
 */  
public class Exercise {  
  
    public static void main(String[] args) {  
        Outer.method().show();  
    }  
}  
  
interface Inter { void show(); }  
//补齐代码  
class Outer {  
  
    static Inter method() {  
  
        return new Inter() {  
            @Override  
            public void show() {  
                System.out.println("HelloWorld");  
            }  
        };  
    }  
}
```

Object类

JDK文档用JDK1.8英文

API : 类中定义的方法

地位 : Java中所有类 , 或直接或间接的父类

构造方法 : 只有无参构造方法 public Object();

(用于实现子类对象的隐式初始化 , 保证父类的构造方法先于子类的构造方法执行) (先初始化父类对象再初始化子类对象)

隐式初始化 : (先调用父类构造方法 , JVM隐式完成)

1. 父类提供了默认的无参构造方法
2. 且子类的构造方法中没有显式调用父类的其他构造方法
3. 那么在JVM会自动先执行父类的构造方法 , 再执行子类的构造方法

成员变量 : 无

成员方法

1. getClass()
2. toString()
3. equals()
4. hashCode()

5. clone()

6. finalize()(了解即可)

getClass()

```
public final Class getClass()
```

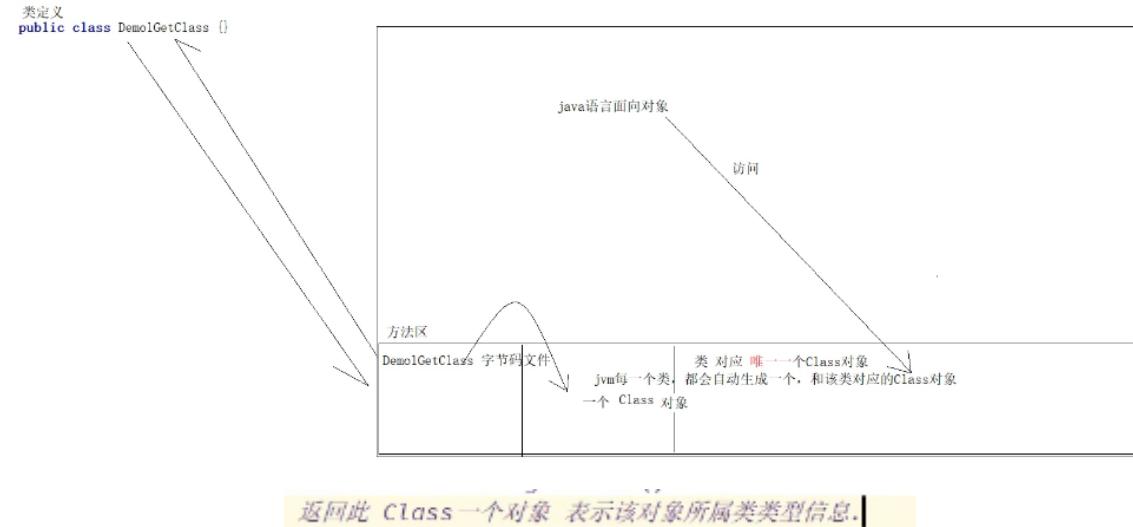
作用：返回的是一个Class对象（该对象的运行时类）

```
Class obj = 对象.getClass()
```

Class 对象里面封装了一个类的类定义信息（全类名、构造方法、成员变量、成员方法）

一个类只有一个Class对象（所以obj1.getClass() == obj2.getClass() 实际上就是在比较地址值）

当类加载的时候，JVM会为每个类自动生成一个Class对象（存储在方法区）



toString()

```
public String toString()
```

作用：返回该对象的字符串表示（建议所有子类都重写该方法）

字符串：必须要能代表一个对象

默认实现：getClass().getName() + "@" + Integer.toHexString(hashCode())

覆盖实现：类名 + 成员变量及其取值

toString()方法自动调用的情况：

1. println()中（如果覆盖了该方法的话）
2. 一个对象和一个字符串拼接的时候

equals()

```
public boolean equals(Obejct obj)
```

作用：指示某个对象与当前对象是否相等

相等：同种类型且属性的取值相等

所要满足的特性（了解）

1. 自反性： $x.equals(x) == true$
2. 对称性 $x.equals(y) == y.equals(x) == true$
3. 传递性
4. 一致性
5. 对于任何非空引用 x , $x.equals(null)$ 都应该返回false

`equals` 方法在非空对象引用上实现相等关系：

1. 自反性：对于任何非空引用值 x , $x.equals(x)$ 都应返回 true。自己和自己相等
2. 对称性：对于任何非空引用值 x 和 y , 当且仅当 $y.equals(x)$ 返回 true 时, $x.equals(y)$ 才应返回 true。
 y 等于 x , 且 x 等于 y
3. 传递性：对于任何非空引用值 x 、 y 和 z , 如果 $x.equals(y)$ 返回 true, 并且 $y.equals(z)$ 返回 true, 那么 $x.equals(z)$ 应返回 true。
如果 x 等于 y , 且 y 等于 z , 则 x 等于 z
4. 一致性：对于任何非空引用值 x 和 y , 多次调用 $x.equals(y)$ 始终返回 true 或始终返回 false, 前提是对象上 `equals` 比较中所用的信息没有被修改。
用来比较的对象的成员变量值没有被修改
5. 对于任何非空引用值 x , $x.equals(null)$ 都应返回 false。
人和对象和null比较，结果确定false

默认实现：比较的是对象的地址值（等价于`==`）

tips:对于浮点数的比较：

```
💡 // 对于小数类型的，变量的比较通常使用Double.compare() 来比较两个double小数值  
    return Double.compare(this.doubleValue, anotherObj.doubleValue) == 0;  
}
```

hashCode()

`public int hashCode()`

作用：把一个对象转换为一个整数，使得这个整数尽可能唯一地表示这个对象（int整数是固定大小的）

hash function

一个将任意大小的数据集映射成一个到一个固定大小的数据集中

eg：所有整数 % 11 -> [0,10]，字符串：字符累加

理想下：希望一一对应

hash冲突：被映射的不同的对象具有相同的hash码值

默认实现：输出的是对象的内存地址的10进制表示（语言层面不需要这个）

hashCode的常规协定

1. 对象的hash码值由比较对象相等时所用的信息计算得到（属性取值）
2. 相等的两个对象，他们的hash码值应该相等
3. 不相等的两个对象，hash码值也可以相同（冲突）

clone()

protected Object clone()

作用：创建并返回对象的一个副本（与原对象不同（地址不同）但是相等的对象）

注意事项：

1. protected权限问题：类A和类B都是Object类的直接子类，在**类A类体**中可以直接访问类A继承的Object的clone()，也可以通过创建类A对象来访问类A继承的Object的clone()，但是不能通过创建类B对象来访问类B继承自Object的clone()（是啊，protected的方法是跨包子类对象可见，必须在自己的类体中才能访问到自己继承的protected成员）
2. 被复制的类必须实现Clonable接口（标记接口，空接口）（如果对象的类不能实现Clonable接口，则会抛出CloneNotSupportedException）
3. Object类的clone()的默认实现是浅拷贝（单纯的成员变量的值复制）

标记接口的作用：

```
// 一个类实现了空接口，那么它的数据类型就发生了变化
MyClass myClass = new MyClass();
NullInterface nullInterface = myClass;

// 一个类实现了NullInterface空接口，我们就说该类被打上了一个标记
// 一旦一个类被打上了空接口的标记，识别标记

if (myClass instanceof NullInterface) {
    // 针对有这种标记类的对象，做特殊处理
}
```

浅拷贝和深拷贝

Object的Clone方法：浅拷贝（复制的不够完全）

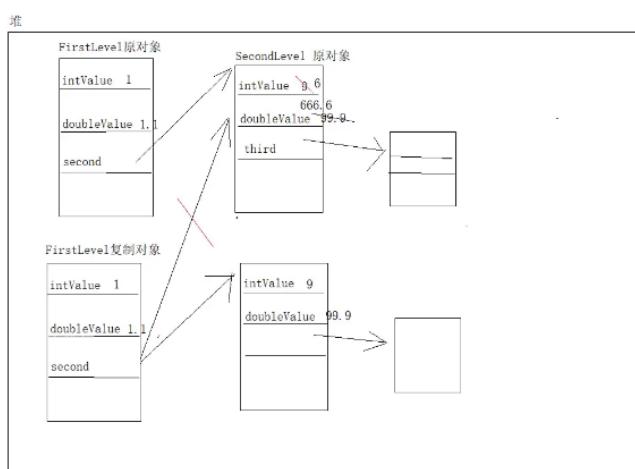
1. 在堆上开辟复制对象的内存
2. 将原对象成员变量的值，依次复制到复制对象中。（对于引用类型的成员变量，仅仅只复制引用变量的值）

和浅拷贝相对，还有深拷贝：
所谓深拷贝，就是指对原对象的完全复制，即复制对象和原对象完全独立。

深拷贝和浅拷贝唯一的区别，仅在于对原对象引用类型成员变量的复制：
a. 浅拷贝只复制引用变量的值
b. 而深拷贝，复制的原对象引用变量所指向的对象

如果要完成对一个对象的深拷贝：
1. 调用Object的clone，首先对原对象做前拷贝。

2. 紧接着修改浅拷贝之后的复制对象的，所有引用变量值，让他们指向相应的复制对象



深拷贝的实现：（类似递归）

1. 首先调用Object类的clone()，完成浅拷贝
2. 再让引用类型成员变量指向他们深拷贝后的对象（去补充）

finalize()

```
protected void finalize()
```

作用：当垃圾回收器确定没有引用指向该对象的时候，垃圾回收器会自动调用这个方法，完成对象的回收工作

实际开发中不会使用（规避不确定性，因为垃圾回收是GC自动完成的，所以对象的finalize执行时机也不确定）

回收原因：因为对象会占用操作系统的资源

回收时机：当对象变成垃圾时（没有任何引用指向它）

不确定性：因为垃圾回收器执行时机不确定所以垃圾对象的finalize方法执行时机也不确定

Week 4

day 1

String

概念：是由多个字符组成的一串数据（字符序列）

构造方法

1. public String() 创建一个空字符串
2. String(byte[] bytes) 接收一个字节数组 codepoint {97, 98 ,99} -> "abc"
3. public String(byte[] bytes,int offset,int length) 部分，从offset开始的length个字节
4. public String(char[] value) 接收一个字符数组
5. public String(char[] value,int offset,int count)
6. public String(String original) 没什么

特征

1. 字符串是常量，其值在创建之后不能更改（Java语言特性）
2. 不可变指的是不能改变一个已经存在的字符串对象的内容，对字符串进行修改操作（拼接/截取）的话就会在堆上创建一个新的字符串对象，并返回新的对象的引用

字符串常量池（对字符串做计算就会在堆上创建对象 两个常量相加除外）“ ”引起的字符串，字符串字面值常量

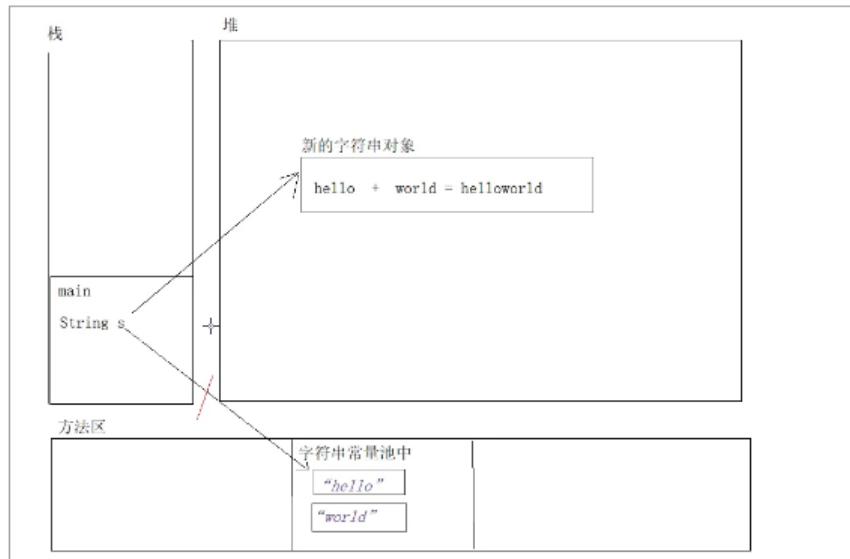
```

String s = "hello";
s += "world"

一个字符串字面值常量，所代表的也是一个String对象

常量池： 字面值常量（java语言层面）
字符串常量池：存储字符串字面值常量
"hello", "world"

```



```

1 | String s = new String("hello") // 创建了两个字符串对象，一个在字符串常量池，一个在堆上
2 | vs
3 | String s = "hello" // 只是在字符串常量池上创建了一个对象

```

对于两个常量 编译器直接拼接（就像1 + 2 自动转化为3）

```

System.out.println(s3==("hello" + "world")); // ??? true
// System.out.println(s3 == "helloworld");

```

运算规则

- 当参与字符串拼接的两个字符串中，至少有一个是字符串的引用变量，此时拼接运算，必然会在堆上创建新的字符串对象（`s1 + s2` `s1 + "hello"` 都会在堆上创建新的对象）
- 只有参与字符串拼接运算两个字符串，都是字符串常量的时候，此时不会在堆上创建新的字符串对象

常用API

判断功能

- `boolean equals(Object obj)` 比较内容
- `boolean equalsIgnoreCase(String str)` 忽略大小写比较内容
- `boolean contains(String str)` 是否包含
- `boolean startsWith(String str)` 是否以此为开头
- `boolean endsWith(String str)` 是否以此为结尾（用于判断文件类型 ".txt"）
- `boolean isEmpty()` 是否是空字符串

获取功能（原字符串不变）

- `int length()` 字符个数
- `char charAt(int index)` 获取指定下标的字符
- `int indexOf(int ch)` 就返回字符首次出现的位置，如果没找到返回-1
- `int indexOf(String str)` 查找当前字符串中，目标字符串首次出现的位置（如果包含），找不到，返回-1，这里的位置是指目标字符串的第一个字符在当前字符串对象中的位置

5. int indexOf(int ch,int fromIndex) 从当前字符串对象的**指定位置**开始，查找**首次**出现的指定字符的位置(如果没找到返回-1)
6. int indexOf(String str,int fromIndex) 指定从当前字符串对象的**指定位置**开始，查找**首次**出现的指定字符串的位置(如果没找到返回-1)
7. String substring(int start) [start]
8. String substring(int start,int end) [start, end)

转换功能

字符串 ->

1. byte[] getBytes() 获取一个用来表示字符串对象字符序列的字节数组
2. char[] toCharArray() 获取一个用来表示字符串对象字符序列的字符数组

其他数据 -> 字符串

1. static String valueOf(char[] chs) (String.valueOf(charArray))
2. static String valueOf(int i) (String.valueOf(10)) -> "10"

字符大小写的转化

1. String toLowerCase()
2. String toUpperCase()

String concat(String str) 字符串拼接 作用等价于+ 实现的字符串拼接(了解)

其他功能

1. 替换功能

String replace(char old,char new) 新替换旧
String replace(String old,String new)

2. 去掉字符串中的空格字符(开头和结尾)

String trim()

3. 比较功能

int compareTo(String str) 根据字典序，比较两个字符串大小
int compareIgnoreCase(String str)

字符串的比较，如何比较

按照字典序，比较字符串的大小。字典序原本的含义：英文单词在字典中出现的先后顺序，在字典中，先出现的字符串小，后出现的字符串大

具体实现：是根据两个字符串字符串**从左往右数，第一个对应位置的不同字符**，来决定两字符串的大小

Comparable接口

表示了一种抽象的比较大小的规则(协议)

int compareTo(String o) 规定了一种比较的规则
<0 当前对象小于待比较对象

>0 大于

=0 等于

接口表示了一种规则（协议）

```
/** The value is used for character storage. */
private final char[] value;

String
public int compareTo(String anotherString) {
    int len1 = value.length; // 获取到当前字符串，所代表的字符序列中包含的字符个数
    int len2 = anotherString.value.length; // 获取带比较字符串，所代表的字符序列中包含的字符个数
    int lim = Math.min(len1, len2); // 取两个字符串长度小者
    char v1[] = value;
    char v2[] = anotherString.value;
    int k = 0; // k 指示下一个带比较的位置
    while (k < lim) {
        char c1 = v1[k];
        char c2 = v2[k];
        if (c1 != c2) {
            return c1 - c2; // 如果说c1 < c2 < 0 小于关系
        } // 如果c1 > c2 > 0 大于关系
        k++;
    }
    return len1 - len2; // 如果两字符串对应位置的字符都相同
}
```

第一种情况：
当前字符串： hello
带比较字符串： hella

第二种情况：
当前字符串： hello
待比较字符串：elloworld

lim 4

lim 5

补

```
public String[] split(String regex)
```

StringBuffer

概念：

线程安全的可变字符序列（线程安全：多线程环境下访问数据的正确性）

类似于String的字符串缓冲区，但内容可变

构造方法

```
public StringBuffer()
public StringBuffer(int capacity)
public StringBuffer(String str)
```

capacity vs length(StringBuffer会自动扩容)

capacity:字符缓冲区本身的大小（16个字符）

length：字符缓冲区中真正包含的字符个数

成员方法

添加功能：不管任何数据类型都可以被添加到字符缓冲区中，如果该类型不是字符串类型：转化成字符串类型

```
public StringBuffer append(String str) 尾部添加
```

```
public StringBuffer insert(int offset, String str) offset位置添加
```

删除功能

```
public StringBuffer deleteCharAt(int index)  
public StringBuffer delete(int start,int end) [start, end)
```

链式调用 (函数的返回类型是对象的引用类型 + return this 就是返回调用这个方法的对象)

```
//链式调用 向stringbuffer插入 'a' 1 true  
stringBuffer.insert( offset: 1,  c: 'a')  
    .append(1)  
    .append(false);  
System.out.println(stringBuffer.toString());
```

```
1  @Override  
2  public synchronized StringBuffer append(String str) {  
3      toStringCache = null;  
4      super.append(str);  
5      return this;  
6  }
```

替换功能

```
public StringBuffer replace(int start,int end,String str)
```

反转功能

```
public StringBuffer reverse()
```

截取功能

```
public String substring(int start)  
public String substring(int start,int end) [start, end)
```

面试题

- String, StringBuffer和StringBuilder有啥区别
- StringBuffer和StringBuilder从效率上来说哪个更快?

String不可变 StringBuffer和StringBuilder可变

StringBuffer针对多线程环境 StringBuilder针对单线程环境 (效率上说StringBuilder更快)

Date

表示特定的瞬间，精确到毫秒

构造方法

Date()

Date(long date)

方法

public long getTime()

DateFormat

1. 是日期/时间 格式化 子类的 抽象类
2. 它以与语言无关的方式格式化并解析日期或时间
3. 因为是抽象类，所以实际使用的是SimpleDateFormat这个实现子类

相应用对关系

y 年

M 表示年中的月份

d 表示月份中的天数

H 表示一天中的小时数

m 小时中的分钟

s 分钟中的秒数

常用： yyyy/MM/dd HH:mm:ss

public Date parse(String source)

把一个用**字符串表示的时间**转化成一个**Date对象**，该对象表示的时间点，就是你用字符串表示的那个时间点

public final String format(Date date)

把一个**Date对象**表示成一个**指定格式的表示时间的字符串**

Math

成员方法

public static int abs(int a) //求绝对值

public static double ceil(double a) // 取整 向大的方向取整

public static double floor(double a)// 取整 向小的方向取

public static int max(int a,int b) 还有min

public static double pow(double a,double b) a^b

public static double random() // 返回带正号的 double 值，该值大于等于 0.0 且小于 1.0 [0.0 1.0)
随机数。

public static int round(float a) 取整 4舍5入的取整

public static double sqrt(double a) 返回正确舍入的 double 值的正平方根

day 2

异常

基本概念

异常：用来表示Java程序运行过程中的错误（信息）

异常机制：Java语言程序获取运行过程中的错误并处理错误的机制（异常的发现 + 异常的处理）

异常机制的由来：

1. C语言时代的错误处理，紧靠口头的约定，没有任何语法的强制约束
2. Java的基本理念：将错误尽可能的摒弃在jvm之外，最好是在编译阶段就发现错误，但很多错误是程序运行时才发生的，所以java语言提供了一种统一的机制，**让程序在自己出错时能够知道怎么去处理这些错误**

异常机制的本质：

1. 提供了一种**一致性的错误报告模型**（一旦发现错误，自己可以处理就处理，如果自己不能处理就将该错误信息报告给上层（调用者），上层也是一样的处理方法）
2. 使得类的构建者与使用者之间可以进行可靠的沟通

异常的分类（根据错误的严重程度）

1. Error：程序层面无法处理的错误，所以不关心
2. Exception：在程序中或许可以处理的错误
 1. 编译时异常 CheckableException：可预见的，要求在代码编写时处理（出门时检查有没有带钥匙）
包括：Exception和它的直接子类（除RuntimeException之外的）
 2. 运行时异常 RuntimeException：不可预见的，不要求在代码编写时处理（出门后发生交通事故）
包括：RuntimeException和它的直接子类

JVM默认异常处理流程（JVM调用main method）

实际上就是调用了异常对象的printStackTrace()方法

1. 当代码执行到发生错误的地方，JVM会首先终止程序的执行，转而执行JVM自己的错误处理流程
2. 在发生错误的地方，收集错误信息，产生一个描述错误的对象（将错误信息封装成一个异常对象）
3. 将错误信息输出到控制台窗口中

Java语言提供的自己的异常处理

try-catch代码块

异常的捕获

try代码块：当try代码块中的代码出现异常时，该异常信息会被JVM收集起来，并交给相应的异常处理器（catch代码块）

catch分支：异常处理器

单分支的异常处理

1. try代码块中的代码发生错误后，JVM会在发生错误的代码处，收集错误信息，并将错误信息封装到一个异常对象中
2. 而在错误代码后面的代码就不会得到执行，JVM会直接跳转到相应的错误处理器中执行开发者自己写的错误处理代码（catch中的代码只有在try块中的代码出错时才会执行）（`num = num / 0` 只会执行右边的代码，而不会执行赋值）
3. 错误处理器中的代码执行完后，程序继续正常执行try-catch代码块之后的代码

多分支的异常处理

原因：希望对try代码块中出现的不同类型的异常，能够进行针对性的处理

匹配过程：

1. 根据实际的异常对象的类型与异常分支（异常处理器）声明的异常类型从上到下进行类型匹配
2. 一旦匹配成功就将这个异常对象传递给该异常分支
3. 类似于if-else语句，一次匹配中最多只会执行多个catch分支中的一个

注意事项

1. 如果不同的异常分支中的异常类型间存在父子关系，那就应该讲子类放在父类前面（否则子类中的代码不会得到执行，异常对象都进入了父类的catch代码块）
2. 不是try代码块中产生的所有异常都能被catch块中代码处理，只有在类型匹配成功时才会处理

tip: Exception是所有异常的父类， RuntimeException是所有运行时异常的父类

tip: 一个异常分支可以处理多种类型的异常，以避免代码冗余

catch(异常类型1 | 类型2 | 类型3 ... e) {...} 声明处理多种类型的异常

获取异常信息

Throwable类中的方法

`String getMessage()`：获取异常产生的原因（\ by zero）

`String toString()`：获取异常类名和异常信息

`void printStackTrace()`：获取异常类名和异常信息，以及异常在程序中的出现位置，并将其打印到控制台

运行时异常与编译时异常

运行时异常：所有的RuntimeException类及其子类都是运行时异常(空指针、数组索引越界、除零异常)

编译时异常：其他的异常 (Exception类及其不包括RuntimeException的子类)

语法上的区别：

运行时异常：无需显式处理，但也可以和编译时异常一样处理

编译时异常：必须显式处理，否则无法通过编译 (throws声明或者try-catch)

异常的抛出

throws (声明异常列表，针对的是针对编译时异常，运行时异常可以不用)

作用：

1. 在方法定义时使用
2. 声明该方法可能会抛出的异常类型
3. 对于编译时异常，可以在语法层面强制调用者处理该异常（自己处理或者继续向上报告）

语法：

方法声明：修饰符 + 返回类型 + 方法名 + 参数列表 + throws + 异常列表

注意事项

1. 对于运行时异常，**JVM会自动将异常信息向上抛出**，而对于编译时异常则必须通过**throws关键字**声明才能向上抛出
2. 异常列表中的异常类型间有逗号隔开，且类型之间最好不要有父子关系（如果是在调用者处自动生成的代码，只会为父类类型的异常来生成生成对应的代码块）
3. 方法覆盖时子类的异常列表必须与父类的兼容（只针对编译时异常）
 1. 子类声明的异常类型只能是父类声明的异常类型的子类或一样
 2. 父类没有异常列表时子类也不能有
 3. 父类有异常列表，子类可以没有

throw (自己抛出异常，不是JVM做)

作用：

1. 在方法体中使用
2. 用于主动在程序中抛出异常
3. 每次都只能抛出某个具体的异常对象

执行特征：一旦执行了**throw语句**，代码的执行直接转到上一层（方法调用处）

语法：throw + 异常对象

注意事项：当要抛出编译时异常时必须配合**throws**使用（抛出编译时异常 不管是自己还是JVM 方法那里都要+ throws ）

throws vs throw

throws

1. 用在方法声明后，后面跟的是异常类名
2. 可以跟多个异常类名，用逗号隔开
3. 表示抛出异常给该方法的调用者处理
4. throws表示出现异常的一种可能性，并不一定会发生这些异常

throw

1. 用在方法体内，后面跟的是异常对象
 2. 只能抛出一个异常对象
 3. 表示抛出异常（也可以在方法体内处理，但没有人会这么做）
 4. 执行了throw语句则一定会抛出某个异常对象
-

异常处理的总结

1. 异常处理策略：①捕获并处理 ②向上抛出
 2. 异常处理原则：如果在方法体内部可以处理就自己处理，自己处理不了就向上抛出
 3. 感知：异常一旦被捕获后没有再向上抛出，那么上层是感知不到这个异常的
-

finally

执行特征：

除特殊情况外，finally代码块中的代码一定会执行（不管try代码块中是否发生异常，或者是在finally代码块之前有return语句执行）

特殊情况：在执行到finally代码块之前JVM退出了（如System.exit(0)）

作用：（因为它一定会执行）用于释放资源，在IO流操作和数据库操作中会见到

自定义异常

原因：需要自己定义异常来描述特定场景下的异常情况

（我们自己认为是错误，JVM不这么认为 如：input < 0）

作用：可以专门处理特定类型的异常

实现：

1. 编译时异常：继承自Exception
 2. 运行时异常：继承自RuntimeException
-

面试题：final finally finalize的区别

1. final:修饰类之后，该类不能被继承；修饰变量之后，该变量变成自定义常量；修饰方法之后，该方法不能被覆盖
2. finally修饰代码块
 1. 对于try-catch-finally代码块而言，不管是否发生异常，finally代码块中的代码都会执行
 2. 即使在finally代码块之前有return语句，finally代码块依旧会执行

3. 特殊情况：只有在执行到finally之前JVM退出了（System.exit(0)），finally代码块才不会执行
3. finalize() Object类中的一个方法：在对象变成垃圾，且被垃圾回收器回收之前，JVM会在该对象上调用finalize方法一次且仅一次

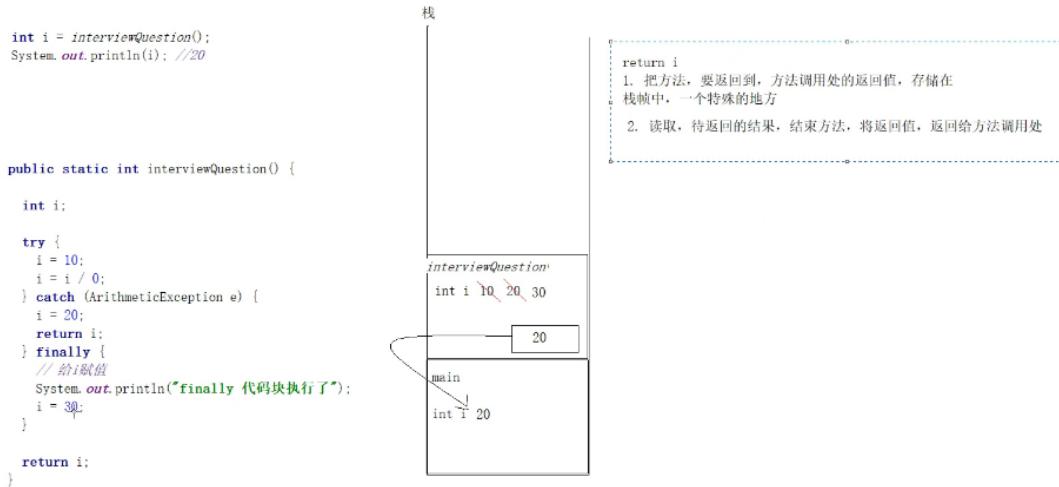
面试题

2. 如果catch里面有return语句，请问finally的代码还会执行吗？会执行

如果会，请问是在return前还是return后。 finally既不在return之前，也不在return语句之后，而是在return语句执行的中间

return i; (先存后返)

1. 先存储返回值到方法对应的栈帧中的一个特殊位置（局部变量表的特殊位置）
2. 再读取暂存的返回值，结束方法的执行，将返回值返回给到方法调用处



day 3

File

为什么要学File类

操作系统中需要永久保存的数据都是以文件的形式存在，想要操作这个永久保存的数据，就首先必须了解Java语言如何描述、表示文件

概述

File类：文件和目录的路径名的抽象表示形式（逻辑上存在 物理上不一定存在）

File类对象：和路径名字符串在表示操作系统文件或者目录的层面是等价的

文件路径

绝对路径：完整的路径名，不需要任何其他信息就可以定位它所表示的文件（E:\demo\first\a.txt）

相对路径：不完整的路径名，必须使用取自其他路径名的信息进行解释（（E:\demo\）second\a.txt）

相对路径：java.io包中的类总是根据**当前用户目录**来解析相对路径名，由系统属性user.dir指定，通常为JVM的调用目录

路径表示

类Unix：绝对路径名的前缀永远书“/”，相对路径名没有前缀，表示根目录的绝对路径名的前缀为“/”且名称序列为空

绝对路径：/home/zs/6379.conf

相对路径：zs/a.txt

根目录：/

Windows：包含盘符的路径名前缀由驱动器号和一个“：“组成，如果路径名是绝对路径名，还可能后跟“\\”

绝对路径：E:\zs\z.txt

相对路径：没有盘符前缀 z\z.txt

转义字符

```
// 特殊字符的表示  
System.out.println('n');  
  
// 给n这个字符，前面加了一个\，输出的不是n这个字符，而是一个换行符  
// \表示转义，给n加了\，\n就不在表示n这个字符  
System.out.println('\\n');  
  
💡 // \本身表示的含义是转义，但是本身有一个\字符  
System.out.println('\\\\');
```

API

构造方法

File (String pathname) 创建一个File对象，该对象表示的文件或目录就是pathname这个**路径名字符串**表示的操作系统中的文件或目录(逻辑上存在，物理上不一定存在)

File (String parent, String child)重要，当需要在父目录下创建文件时

File (File parent, String child)

创建功能(因为File对象逻辑上存在，物理上不一定存在)

public boolean createNewFile() 物理创建文件

public boolean mkdir() 物理创建目录

public boolean mkdirs()

mkdir vs mkdirs

1. mkdir 只能在已存在的目录下创建新的目录
2. mkdirs 当目标目录的父目录不存在时，会把目标目录和父目录一起创建好

删除功能 (删除有内容的目录 : 后序遍历 , 先子后父)

public boolean delete(): 删除文件或目录 , 如果File对象表示一个目录 , 则该目录必须为空才能删除

重命名功能

public boolean renameTo(File dest)

1. 源文件和修改后的目标文件在同一目录的时候 : 重命名
2. 源文件和修改后的目标文件不在同一目录的时候 : 移动文件 + 重命名

判断功能 (都是访问物理文件 / 目录的属性得到的)

public boolean isFile() 判断当前File对象是否表示文件

public boolean isDirectory() 判断当前File对象是否表示目录

public boolean exists() 判断File对象表示的文件或目录是否物理上也存在

后面三个都是通过文件属性可知

public boolean canRead() 判断文件是否可读

public boolean canWrite() 判断文件是否可写

public boolean isHidden() 判断文件是否隐藏文件

基本获取功能

public File getAbsoluteFile() 获取文件的绝对路径字符串

public String getPath() 获取文件的路径字符串

public String getName() 获取文件的文件名 (文件名包含文件后缀)

public long length(), 返回由此抽象路径名表示的文件的长度。如果此路径名表示一个**目录** , 则返回值是不确定的。大小单位为**字节**

public long lastModified() 表示文件最后一次被修改的时间的 long 值 , 用与时间点 (1970 年 1 月 1 日 , 00:00:00 GMT) 之间的毫秒数表示

高级获取功能 (针对目录)

public String[] list()

返回一个**字符串数组** , 这些字符串指定**此抽象路径名**表示的目录中的文件和目录
如果此抽象路径名不表示一个目录 , 那么此方法将返回 null

(每个字符串是一个**文件名**而非完整路径)

public File[] listFiles() (常用)

返回一个**File对象数组** , 每个数组中的元素对应目录中的文件或目录
如果此抽象路径名不表示一个目录 , 那么此方法将返回 null (如果对文件使用listFiles()返回结果就是 null)

自定义获取功能 (重载了listFiles)

```
File[] listFiles(FileFilter filter)

public interface FileFilter {  
    boolean accept(File pathname);  
}
```

true表明File对象满足，false表明File对象不满足条件（要被过滤掉）

场景：只想获取目标目录中.txt文件（pathname.getName().endsWith(".txt")）

用匿名内部类对象

day 4

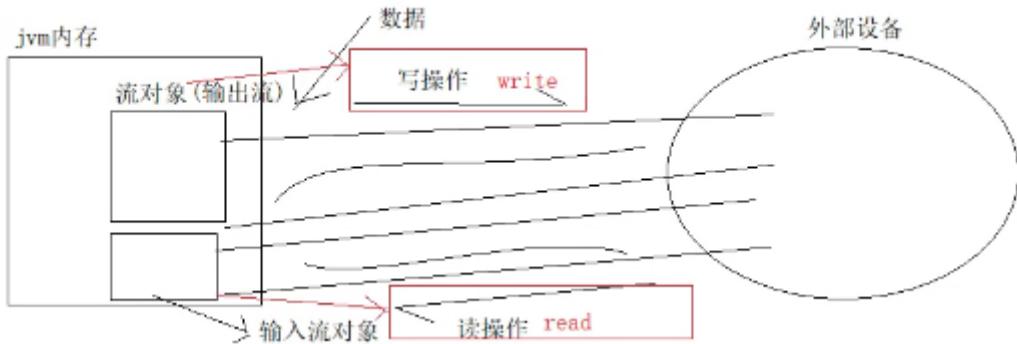
IO产生的原因

1. 内存中的数据一断电就会消失，所以所有需要永久保存的数据都是以文件的形式存储在外部设备中
2. 但程序运行时，又需要读取相关的数据到内存中才能运行
3. 同时内存的容量远小于外部设备（硬盘），所以在内存与外部设备之间需要进行频繁的数据传输

Java语言引入了流，基于流模型来完成IO的功能

流模型：

1. 基于流的数据传输，在数据传输前会在内存与外部设备之间建立数据传输通道（创建流对象）
2. 在数据传输通道中，数据的传输类似于水的流动(写和读才能让数据流起来)（内存->外设 外设->内存）



IO流：用来处理JVM与外部设备之间的数据传输，通过流(stream)的方式

IO流的分类：

1. 按照数据流动的方向（输入输出都是以内存为基准）
 1. 输入流：内存 <- 外设
 2. 输出流：内存 -> 外设
2. 按照传输数据的单位
 1. 字节流：传输的是以字节为单位的二进制数据（01）
 2. 字符流：传输的是以字符为单位的字符数据

使用场景：

1. 文本数据用字符流(文本数据就是txt打开能看的数据)
2. 除文本数据以外的数据用字节流

IO流的基类：(都是抽象类，只能间接实例化)

1. 字节流
 1. 输入流：InputStream
 2. 输出流：OutputStream
2. 字符流
 1. 输入流：Reader
 2. 输出流：Writer

注：

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的后缀

如：InputStream的子类FileInputStream。

如：Reader的子类FileReader

字节流

OutputStream(字节输出流)

实例化：

抽象类，只能间接实例化

构造方法：

创建一个文件输出流，向指定的目标文件写入字节数据

FileOutputStream(File file)

FileOutputStream(String name)

写方法 (内存->外设)

public void write(int b) 向输出流写入一个字节，只写入b的低8位，高24位会被忽略 (int为32位)

public void write(byte[] b) 向输出流写入一个字节数组中的所有字节

public void write(byte[] b,int off,int len) 向输出流写入一个字节数组中的指定位置开始的len个字节

(常用：write(buffer, 0, len) , 因为buffer数组有可能就没装满)

字节流写数据常见问题

1. 创建字节输出流到底做了哪些事 (先两端，再中间)

1. JVM首先到外部设备中，查找目标文件，当发现目标文件不存在的时候，JVM会首先创建目标文件 (内容为空) (前提条件：目标文件的父目录要存在)，当发现目标文件存在的时候JVM默认会首先清空目标文件内容，然后从文件头开始写入数据
2. 在内存中创建FileOutputStream对象
3. 在FileOutputStream对象与目标文件之间建立数据传输通道
2. 数据写成功后为什么要close

关闭此输出流并释放与此流有关的系统资源

3. 如何实现数据的换行

out.write(System.lineSeparator().getBytes()) windows: '\r' + '\n' 类unix: '\n'

4. 如何实现数据的追加写入 (创建两个不同的输出流对象)

选择重载的构造方法 FileOutputStream(File file, boolean append), 第二个参数设置为true , 则字节将写入文件末尾处而不是文件开始处

5. 给IO流操作加上异常处理

FileInputStream(创建字节输入流)

1.

InputStream (字节输入流)

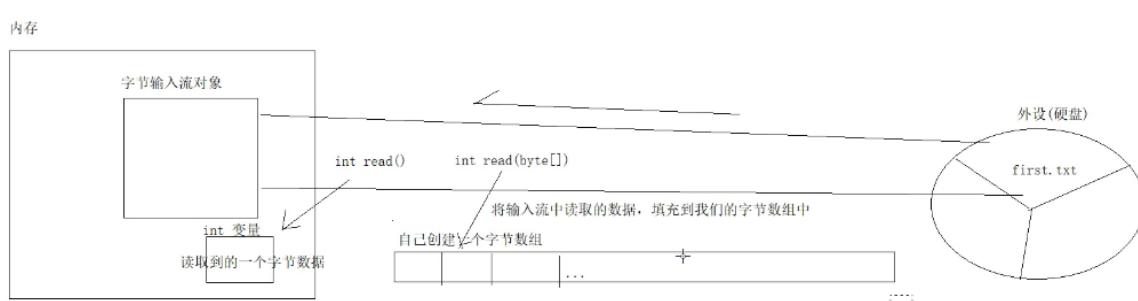
实例化 : 间接实例化

构造方法:

创建一个字节输入流 , 从指定的目标文件中读取字节数据

FileInputStream(File file)

FileInputStream(String name)



read方法 (外设->内存) (创建好流对象后仅仅创建了数据传输通道)

public int read() 从该输入流中读取一个字节数据 , 如果返回值为-1 , 则表明已到达文件末尾

public int read(byte[] b) 从该输入流中读取最多b.length个字节到字节数组中 , 如果返回值为-1 , 则表明已到达文件末尾 (普通情况下返回值为读入到字节数组中的字节数 , 用于赋值给len)

字节流两种复制方式比较

一次读写一个字节效率高还是一次读写一个字节数组效率高 : 字节数组

原因 :

JVM每次IO都需要与操作系统交互 , 都要付出一次通信代价 (IO中断)

一次读写一个字节 : 意味着传输一个字节就要付出一次通信代价

一次读写一个字节数组 : 意味着传输一个字节数组才付出一次通信代价

字节流 复制 数据两种方式比较：

一次复制一个字节

一次复制一个字节数组

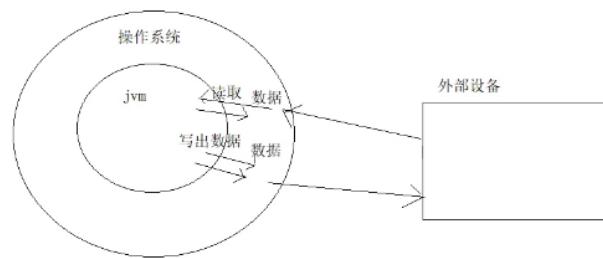
从效率角度讲，哪种方式比较好呢？为什么？

第二种方式比较好

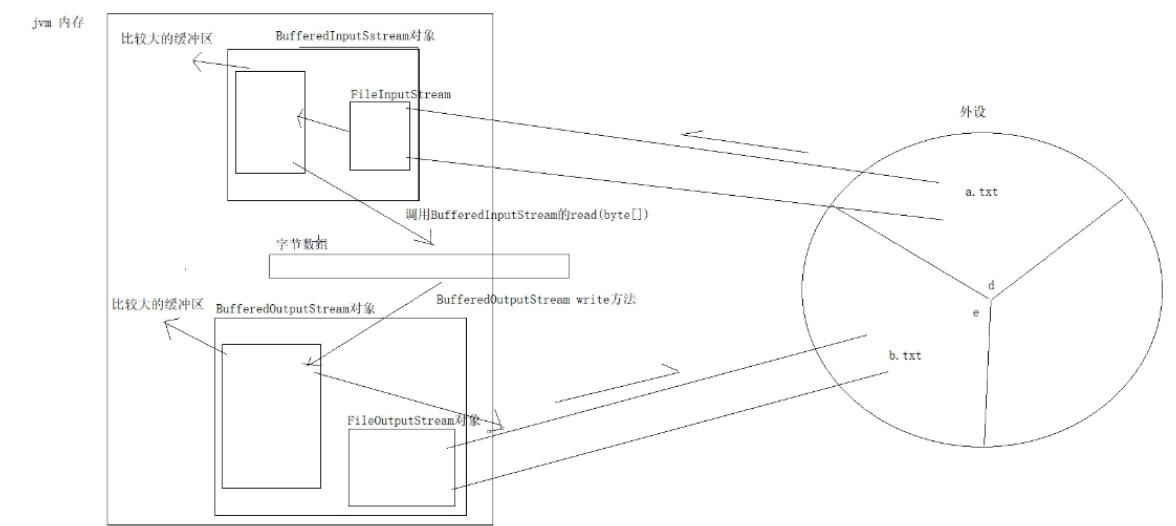
每次jvm实际执行IO，都需要和操作系统交互，这意味着每一次内存和外设的数据传输都需要付出一次通信代价

a. 如果一次传输一个字节数据，这意味着，传输一个字节，就要付出一次通信代价

b. 如果一次传输一个字节数组，这意味着，传输一个字节数组的多个字节数据，才付出一次通信代价



字节缓冲流（包装流）（空间换时间，减少JVM与操作系统的交互）需要先放满缓冲区再从缓冲区中提取数据



产生原因：字节流一次读写一个数组的速度明显比一次读写一个字节的速度快很多，就是因为加入的数组这样的缓冲区效果，Java设计时也考虑到了，所以提供了字节缓冲流

`read(byte[])` 先从输入流中把数据写入`BufferedInputStream`对象的缓冲区，缓冲区填满后，再将数据转入字节数组

`write(byte[], 0, len)` 把数据从字节数组中转移到`BufferedOutputStream`对象中的缓冲区，缓冲区填满后，再把数据从缓冲区写入到输出流中

字节缓冲流分类

字节缓冲输出流`BufferedOutputStream`

`BufferedOutputStream(OutputStream out)`：创建一个新的缓冲输出流，以将数据写入指定的底层输出流。

字节缓冲输入流`BufferedInputStream`

`BufferedInputStream(InputStream in)`

创建一个`BufferedInputStream`并保存其参数，即输入流`in`

提前清空缓冲区方法（在缓冲区还没装满的时候）

`flush`方法：刷新此输出流并强制输出所有缓冲的输出字节

close方法：先调用flush方法，再关闭流

注：关闭流的时候，我们只需要关闭包装流即可，因为包装流会负责关闭它所包装的底层流

心得：

1. 尽量用变量而非常量（易更改）
2. 尽量减少IO次数，把要处理的数据先集合起来
3. 流对象引用变量在try-catch-finally块之外声明，初值赋为null，在try块中指向具体的流对象，方便在finally块中用closeQuietly方法关闭
4. 尽可能把单独的功能抽成一个方法如copyFile
5. 可以用FileFilter的匿名内部类对象来表示筛选文件的条件
6. dir.mkdirs()确保操作的目录是存在的
7. 传参的时候可以直接用匿名对象

day 5

字符流

字符流产生原因

用字节流操作中文等文本数据不方便，因为有些字符需要用多个字节表示，而字节流的传输单位是字节，所以在字节流中可能会出现不完整的字符表示

核心原因：数据逻辑单位不同

字符的表示

字符：在计算机中都是对应一个整数值

字符集（charset编码表）：一个规定了字符到其对应整数值的映射的集合('a' - 97)

常见字符集

ASCII：美国标准信息交换码。用一个字节的7位可以表示。（00000000-01111111)(0-127)

ISO8859-1：拉丁码表。欧洲码表用一个字节的8位表示。（latin-1）

GB2312：中国的中文编码表。

GBK：中国的中文编码表升级，融合了更多的中文文字符号（简体中文操作系统使用）

GB18030：GBK的取代版本

BIG-5码：通行于台湾、香港地区的一个繁体字编码方案，俗称“大五码”。

Unicode：国际标准码，融合了多种文字（只存在于逻辑中）（以下为Unicode的两个变体）

1.UTF-16**定长编码**两个字节表示一个字符（JVM内部使用，所以JVM中字符所占存储空间都是2个字节）

2.UTF-8：**可变长度**来表示一个字符，它定义了一种“区间规则”，

这种规则可以和ASCII编码保持最大程度的兼容：

它将Unicode编码为00000000-0000007F的字符，用单个字节来表示

它将Unicode编码为00000080-000007FF的字符用两个字节表示

它将Unicode编码为00000800-0000FFFF的字符用3字节表示

1字节 0xxxxxxxx

2字节 110xxxxx 10xxxxxx

3字节 1110xxxx 10xxxxxx 10xxxxxx

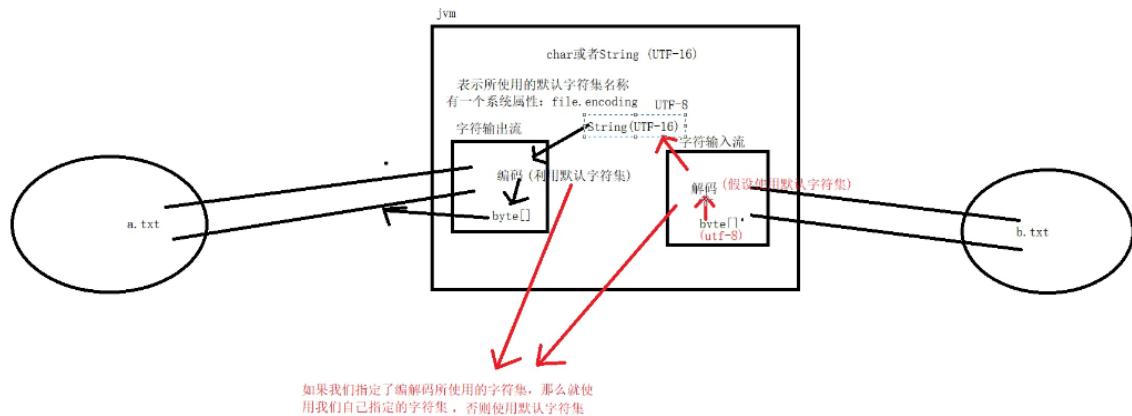
注：JVM中的编解码用的是UTF-16，固定用两个字节表示一个字符

编解码（基于指定的字符集）（编解码都是指我们自己使用的字符集，和JVM存储的无关）

编码：字符 -> 整数值（编码值）

解码：整数值 -> 字符

编码与解码必须使用同一个字符集，否则会出现乱码问题



起点是字符 终点也是字符（字符在JVM中的编码方式是UTF-16）

Java语言层面 编码与解码

编码：字符串对象.getBytes(String charsetName)（IDEA默认使用UTF-8来编码和解码）

解码：String(byte[], String charsetName)

String(byte[], int offset, int len, String charsetName)

默认字符集：当没有指定所用编码表或字符集的时候，在IDEA中默认使用UTF-8，如果没有使用IDEA原有的默认字符集是GBK

常识：gbk字符集中，用两个字节表示一个中文字符

utf-8字符集中，中文字符通常用3个字节表示一个中文字符

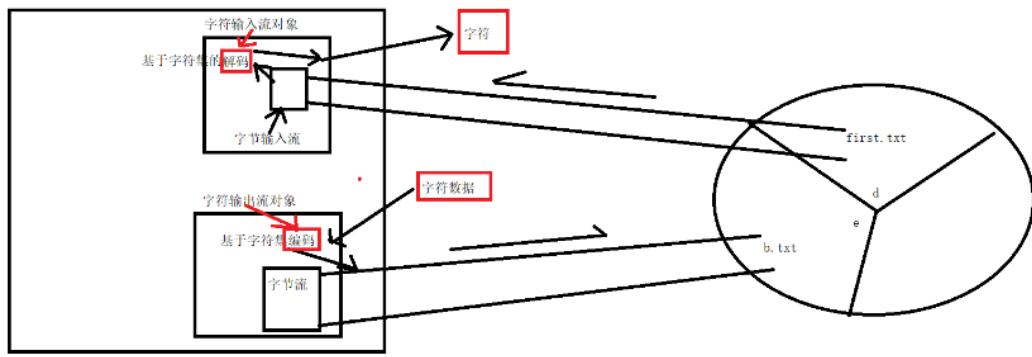
字符流的实质（包装流）

字符流 = 字节流 + 字符集（根据指定的字符集，进行编码解码）

字符输出流：编码 + 字节流

字符输入流：解码 + 字节流

字符流 = 字节流 + 编码表 (根据指定编码表, 编解码的过程) 都是包装流



字符流体系

字符流的基类 : Writer + Reader

Writer具体子类: OutputStreamWriter (两步), FileWriter (一步), BufferedWriter

字符输出流

OutputStreamWriter

使用转化流包括两步：

1. 创建底层的字节流
2. 包装字节流，指定字符集并完成转换流的创建

构造方法：(包装一个字节流，第二个参数可以指定字符集)

```
public OutputStreamWriter(OutputStream out)
```

创建使用默认字符编码(对于IDEA而言UTF-8, 但是原生情况下默认字符集GBk)的 OutputStreamWriter。

```
public OutputStreamWriter(OutputStream out, String charsetName) (非常体现本质)
```

创建使用给定字符集的 OutputStreamWriter。

字符输出流注意事项：

字符流自带一个小缓冲区 (为了实现编解码功能)

所以要用flush或者close功能才能保证缓冲区的数据传到目标文件中

FileWriter:用来写入字符文件的便捷类 (转化流的简化)

构造方法

```
public FileWriter(String fileName),
```

创建使用默认字符集的，字符输出流，专门向文件中写入字符数据

```
public FileWriter(File file)
```

```
public FileWriter(String fileName, boolean append)
```

该构造方法用来实现简化流的文件的追加写入



转化流 vs (FileReader和FileWriter)

1. 创建流对象的角度 : FileReader和FileWriter更方便
2. 指定编解码字符集的角度 : FileReader和FileWriter无法指定编解码使用的字符集 (就是用默认的)

BufferedWriter (包装流的包装流)

构造方法 :

`BufferedWriter(Writer out)`

创建一个使用默认大小输出缓冲区的缓冲字符输出流。

`BufferedWriter(Writer out, int sz)`

创建一个使用给定大小输出缓冲区的新缓冲字符输出流

特有的方法 : `void newLine()` 向流中写入换行符

write方法

`void write(char[] cbuf)` 写入字符数组。 (写入整个字符数组)

void write(char[] cbuf, int off, int len) 写入字符数组的某一部分 (常用 , 因为字符数组可能没装满)

`void write(int c)` 写入单个字符。 要写入的字符包含在给定整数值的 16 个低位中 , 16 高位被忽略

void write(String str) 写入字符串。

void write(String str, int off, int len)写入字符串的某一部分。

注 : 字符流的write方法可以直接写入字符串 , 字节流的write方法则要先将字符串转化为字节数组

Reader: (包装流) (抽象类)

字符输入流

具体子类 : InputStreamReader, FileReader, BufferedReader

InputStreamReader

构造方法

`public InputStreamReader(InputStream in)`

创建一个使用默认字符集的 InputStreamReader。

`public InputStreamReader(InputStream in, String charsetName)`

创建使用指定字符集的 InputStreamReader。

FileReader用来读取字符文件的便捷类 (包装流的包装流)

构造方法

`public FileReader(String fileName)`

创建使用默认字符集的字符输入流 , 从指定文件中读取字符数据

`public FileReader(File file)`

BufferedReader (同样是出于减少IO次数的考虑)

构造方法

BufferedReader(Reader in)

创建一个使用默认大小输入缓冲区的缓冲字符输入流。

BufferedReader(Reader in, int sz)

创建一个使用指定大小输入缓冲区的缓冲字符输入流。

特有方法: String readLine() 读取一行文本数据(不包括该行的换行符)

read方法

int read()

读取单个字符。如果已到达流的末尾，则返回 -1

int read(char[] cbuf)

将字符读入数组。读取的字符数，如果已到达流的末尾，则返回 -1

int read(char[] cbuf, int off, int len) (没什么用，浪费了0-off-1这部分的内存空间)

将字符读入数组的某一部分。

为什么不能用字符流传输图片和文件

1. 因为图片和视频的文件数据都有自己的编码方式，它们的编码方法跟字符的编码方式没有任何关系
 2. 所以在图片和视频的编码数据中会存在一些字符集中不存在的编码值，我们在用字符流读图片和视频的文件数据的时候，就会遇到在字符集中找不到的编码值
 3. 在解码的过程中遇到不认识的编码值，要么丢弃要么用特定的符号???带表示未知编码值对应的字符
 4. 这就意味着视频和图片的数据在数据传输的过程中被改变了（跟乱码一个原理：编解码所使用的字符集不同）
-

标准输入输出流

标准输入流

System类中的in代表了标准输入，默认的输入设备为键盘

System.in 类型是 InputStream(字节输入流)

读方法：阻塞方法，一直等待输入

标准输出流

System类中的out代表了系统的标准输出，默认的输出设备是显示器（控制台窗口）

System.out的类型是PrintStream (OutputStream字节输出流的子类)

PrintStream：把任意类型的数据转化为**字符串**来进行输出

练习

练习：利用System.in 完成Scanner的nextLine()的功能(读取键盘输入的一行字符串)。

```
1     string s;
2     InputStream in = System.in;
3     // 利用字符缓冲流的readLine方法
4     // 因为System.in是字节输入流 所以要经过两层包装
5     BufferedReader br = new BufferedReader(new InputStreamReader(in));
6     while (!(s = br.readLine()).equals("886")) {
7         System.out.println(s);
8     }
```

其他流

Datastream

DataOutputStream:

数据输出流允许应用程序以适当方式将基本 Java 数据类型写入输出流中

DataOutputStream(OutputStream out)

创建一个新的数据输出流，将数据写入指定基础输出流（包装流）

```
1     FileOutputStream fos = new FileOutputStream("c.txt");
2     DataOutputStream dos = new DataOutputStream(fos);
3
4     //写入整数
5     int a = 1000;
6     dos.writeInt(a);
7
8     FileInputStream fis = new FileInputStream("c.txt");
9     DataInputStream dis = new DataInputStream(fis);
10    int i = dis.readInt();
11    System.out.println(++i); // 1001
12
```

PrintStream

打印流

字节流打印流 PrintStream

字符打印流 PrintWriter

打印流特点

1. 只能指定数据的目的地，不能指明数据的来源
2. 可以操作任意类型的数据。
3. 如果启动了自动刷新，能够自动刷新。

要想有自动刷新功能，对于PrintWriter而言

1. public PrintWriter(OutputStream out, boolean autoFlush) autoFlush打开自动刷新功能
2. 调用 println、printf 或 format才会有自动刷新功能
4. 可以操作文件的流

```

1  PrintWriter printwriter = new PrintWriter("d.txt");
2
3  //按行复制文本文件
4  BufferedReader br = new BufferedReader(new
5  FileReader("Exercise01.java"));
6
7  String lingStr;
8
9  while ((lingStr = br.readLine()) != null) {
10     printwriter.println(lingStr); //会自动追加换行符
11 }
12
13 br.close();
printwriter.close();

```

序列化流

ObjectOutputStream:序列化

ObjectOutputStream 将 Java 对象的基本数据类型和对象中持有的对象写入 OutputStream。

可以使用 ObjectInputStream 读取（重构）对象。通过在流中使用文件可以实现对象的持久存储

ObjectOutputStream(OutputStream out)创建写入指定 OutputStream 的 ObjectOutputStream。

注意：只有实现了Serializable接口的对象才可以被ObjectOutputStream写入到文件中，进行持久化存储

ObjectInputStream:反序列化

ObjectInputStream 对以前使用 ObjectOutputStream 写入的基本数据和对象进行反序列化。

ObjectInputStream(InputStream in)

有两种类型的成员变量的值不会被序列化：

1. transient修饰的成员变量（一般用于修饰密码这种不愿意被解析出来的成员变量）
2. static 修饰的成员变量的值

```

1  Demo1ObjectStream os = new Demo1ObjectStream();
2  MyObject toStore = new MyObject(30, 5.5, os);
3  MyObject toStore = new MyObject();
4
5  FileOutputStream fos = new FileOutputStream("t.tmp");
6  ObjectOutputStream oos = new ObjectOutputStream(fos);
// 将对象序列化后存储进文件中保存
7  oos.writeObject(toStore);
8
9  oos.close();
10
11
12  FileInputStream fis = new FileInputStream("t.tmp");
13  ObjectInputStream ois = new ObjectInputStream(fis);
// 从文件中反序列化读取出对象
14

```

```
15     Myobject o = (MyObject) ois.readObject();
16     System.out.println(o);
17 // MyObject{i=30, j=5.5, os=Demo1ObjectStream{i=100}, isBoolean=false}
```

Week 5

day 1

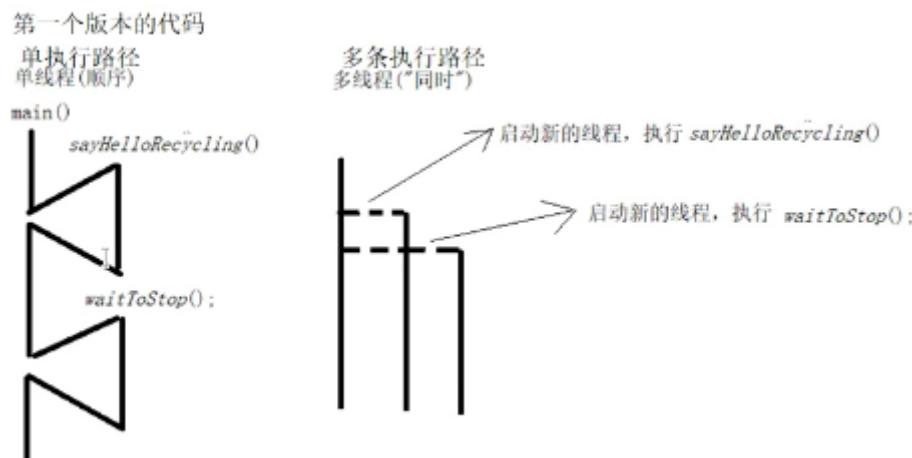
多线程

直观效果：多线程可以使得我们程序中不同的功能“同时”运行起来

多线程的理解

一个线程就代表这条代码的执行路径，所有代码都是在执行路径中运行的（包括main method）

1. 同一条执行路径中的代码，只能按照代码的先后顺序一次执行
2. 不同执行路径中的代码，可以“**同时**”执行



线程之间相互独立，互不干扰

前提：计算机中的运算工作（广义）是由CPU完成的，CPU（的时间片）是我们计算机中最宝贵的资源

为什么会有**多进程**（减少CPU的闲置时间，提高CPU利用率）

1. 单道批处理：

单道：在整个操作系统中，同一时间，内存中只有一个程序在运行，程序的运行，只能是上一个程序运行完，下一个程序开始运行

批处理：在程序运行过程中，不会有任何响应，直到程序运行结束。（没有交互性）

单道批处理操作系统，它并不能很好的利用CPU的计算时间

比如：假设，在单道批处理系统中，运行了一个程序，假设在它的程序中，它需要执行IO(和打印机传输数据)，IO的数据传输过程中有很大一部分时间，是不会用到CPU的计算功能，此时CPU闲置。

2. 多道批处理操作系统(出现了多进程，程序是运行在进程中)就是为了提高CPU的利用率

多道：在操作系统中，同时内存中，可以有多个应用程序，在运行，这样一来，一旦某个应用程序，不需要使用cpu的计算功能，操作系统就会把cpu分配给内存中的其他应用程序来计算。这样一来，cpu就大大提高了。

应用程序的执行：当应用程序，占用cpu执行时间，这个应用才算真正的执行

在多道批处理系统中，多个应用程序，交替执行，看起来好像多个应用程序在“同时”(并发)运行

核心原因：进程的交替执行，交替过程，是需要付出额外代价的——进程的上下文切换(不小)

3. 现代操作系统，引入了另外一个东西，线程(程序运行在线程中) 线程切换的代价，要小于进程切换的代价

线程，又被称为轻量级进程，一个进程中可以有多个线程

同一个进程中的多个线程，线程上下文切换的时候，付出额外代价，小的多

同时的理解：

并发：并发执行，一段时间内，多个程序同时运行（但是同一时刻，只有一个程序执行）(如果只有一个CPU的话)

并行(多CPU)：并行执行，同一个时间点，多个程序同时运行

通常我们生活中所说的同时，指的是并行

为什么会有线程

- 操作系统的角度：线程切换的代价比进程切换的代价小，进一步提高CPU利用率(让CPU把主要精力放在处理任务而非保存上下文上)
- 程序的角度：让程序中的不同功能，“同时”运行起来

进程切换时的额外工作：保存与恢复上下文（都需要使用到CPU）

Java程序的运行

Java命令运行Java程序的过程

- Java命令首先启动了一个JVM进程
- JVM进程在执行的时候首先会创建一个main线程
- 然后再main线程中运行main方法中的代码

JVM是多线程还是单线程 (多线程：main线程 + 至少还有一个垃圾回收线程：运行着垃圾回收器)

```
1 while (true) {  
2     int[] ints = new int[8192];  
3     // 可以一直运行，说明产生的垃圾有垃圾回收线程进行回收  
4 }
```

多线程的第一种实现方式

实现步骤：

1. 定义一个类继承Thread类
2. 在子类中重写父类的Run方法
3. 创建子类的对象
4. 在子类对象上调用start方法，启动线程

注意事项：

1. **Thread类（或者子类）的对象代表一个线程**
2. 重写run方法的原因：因为只有run方法中的代码才会在线程中执行，所以重写是为了保证线程执行的是我们想要执行的代码（也可在run方法中调用其他方法）
3. 启动线程：通过start方法，这样run方法中的代码才是运行在一个子线程当中，直接使用run方法属于方法调用，仍然是单线程
4. 一个线程只能被启动一次，如果我们要启动多个线程只能创建多个对象，并分别启动它们

线程API

线程信息API

线程名

```
public final String getName()
public final void setName(String name)

static Thread currentThread()
```

返回对当前正在执行的线程对象的引用

如何获取main线程的线程名：在main方法中调用Thread.currentThread.getName()

优先级:运行时间越长，优先级越低（动态：操作系统层面的实现）

```
public final int getPriority()
public final void setPriority(int priority)
```

Java中线程调度为抢占式（线程调度由系统的决定，切换不由线程本身来决定）

线程调度：给线程分配cpu的使用权的过程
假设在单CPU的情况下
线程的两种主要调度模型
协同式线程调度(Cooperative Thread-Scheduling)
协同式线程调度：线程的执行时间由线程自己来控制，线程把自己的工作做完之后，要主动通知系统切换到另外一个线程上去
协同式线程调度，最大的好处是实现简单，同时由于线程要把自己的事情干完之后，才会通知系统进行线程切换，所以线程切换对线程自己来说是可知的。

抢占式调度(Preemptive Thread-Scheduling)
抢占式线程调度：每个线程将有系统来分配运行时间，线程的切换不由线程本身来决定。
这种实现线程调度的方式下，线程执行的时间是系统可控的

在抢占式线程调度中，一般是根据线程优先级来决定，该那个线程使用cpu。线程的优先级，是会随着线程的执行，而降低。



注意事项

1. 优先级的取值范围1-10，默认的线程优先级是5

2. Java语言中的Thread的priority，它其实没有什么太大用处(统计意义，总的执行概率)，只是对操作系统的建议
3. 因为线程调度，归根结底主要是操作系统内核，去完成的，jvm起不了决定性作用
操作系统中，有自己的一套优先级规则(静态优先级 + 动态优先级(类似于高响应比优先))，基本上我们设置的静态优先级意义不是很大

线程控制API

sleep()

```
public static native void sleep(long millis)
```

在指定的毫秒数内让**当前正在执行的线程**暂停执行，
当前线程指的是sleep的调用线程 (like:Thread.sleep(2000))

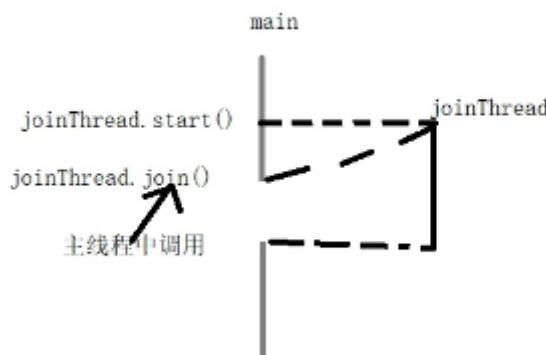
native关键字：表明该方法是由其他语言实现的，但是可以在Java语言中调用它

join()

```
public final void join()
```

等待该线程终止
谁等待谁：当前线程等待(调用join方法的线程)
等待谁呢：等待该线程(调用join的那个线程对象所代表的线程)

```
1 // in main Thread
2 joinThread.join();
3 // means main wait for joinThread to finish
4 // or joinThread will join main
5 // 效果上很像方法调用
```



yield()(礼让方法)(只能暂停一瞬间之后CPU调用谁不一定，实际开发中没人使用)

```
public static native void yield()
```

让当前线程放弃cpu的执行权

setDaemon()

```
public final void setDaemon(boolean on)
将该线程标记为守护线程( true ) 或用户线程(false)
当正在运行的线程都是守护线程时 , JVM终止执行
该方法必须在启动线程前调用
```

守护线程使用场景 : 垃圾回收器线程就是一个守护线程 (因为垃圾都是在用户线程中产生的 , 如果用户线程不存在 , 那就意味着没有垃圾产生 , 垃圾回收器线程也就没有存在的必要了)

interrupt() (只针对处于阻塞状态的线程)

```
public void interrupt()
如果线程在调用 Object 类的 wait()、wait(long) 或 wait(long, int) 方法 ,
或者该类的 join()、join(long)、join(long, int)、sleep(long) 或 sleep(long, int) 方法过程中受阻 ,
它还将收到一个 InterruptedException。
```

打断线程的阻塞状态

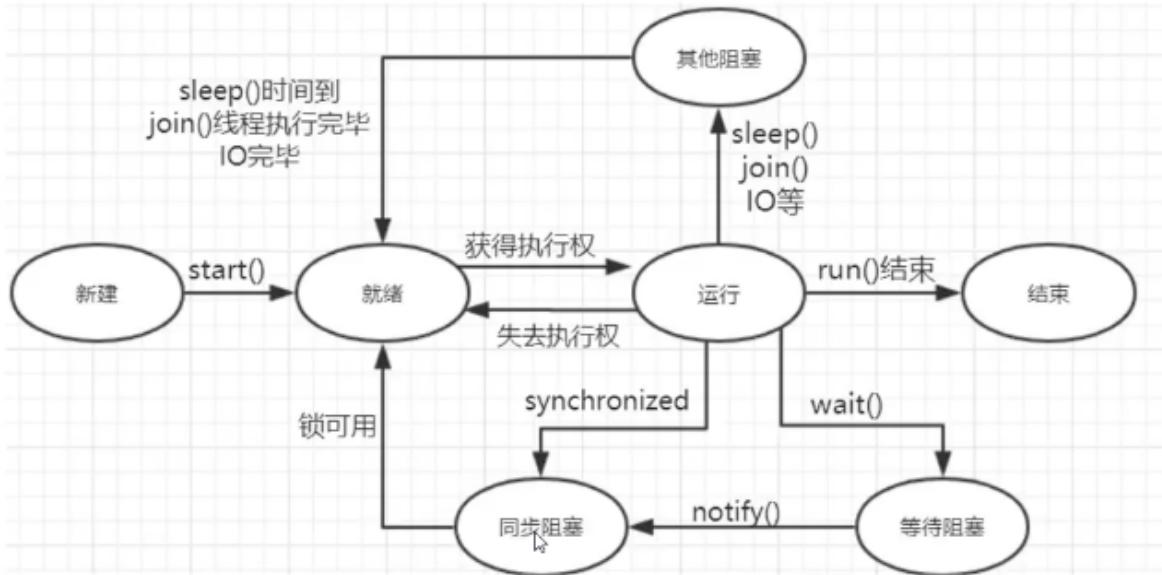
线程的生命周期

```
新建
线程处于刚刚创建的状态
就绪
有执行资格 , 等待cpu调度获得执行权
运行
取得执行权 , 正在cpu上执行
阻塞
无执行资格 , 无执行权
死亡
线程正常或异常终止(run()方法执行完毕) , 线程对象成为垃圾 , 等待·垃圾回收器回收
```

线程结束的标准 : run方法中的代码执行完毕

线程的状态转化图

```
同步阻塞 : 线程加锁失败
等待阻塞 : 调用wait方法 -> 被唤醒后先争夺锁
```



多线程的第二种实现方式

实现步骤

1. 定义实现Runnable接口的子类
2. 实现Runnable接口中的run方法（抽象方法）
3. 创建该Runnable接口子类的对象
4. 在创建Thread对象的时候，将Runnable接口的子类对象作为初始化参数传递给Thread对象
5. 启动Thread对象（thread.start()）

Thread对象：线程

Runnable子类对象：线程中所要执行的任务

注意事项

1. Runnable接口子类中的run方法代码会执行在子线程中，所以在定义子类的时候要实现run方法
2. 但接口的子类对象并不代表线程，而是代表线程中所要执行的任务（逻辑上：执行路径与执行路径中所要执行的任务应当是没有关系的）

传参与执行时的关键代码（Thread类）

```

//3. 创建接口子类对象
MyRunnable myRunnableTask = new MyRunnable();

//4. 在创建Thread对象的时候，将创建好的Runnable子类对象
//作为初始化参数，传递给Thread对象
Thread thread = new Thread(myRunnableTask); // 表示一个线程

//5. 启动线程
thread.start();

```

Thread类中的关键源代码

```

/* What will be run */
private Runnable target;

public Thread(Runnable target) {
    this.target = target;
}

public void start() {
    // 调用start0真正启动一个线程
    start0();
}

private native void start0();

Thread类的run方法，一定会被在子线程中调用
public void run() {
    if (target != null) {
        target.run(); → 在Thread类的run方法
    }
}

```

在Thread类的run方法
中被调用。
target.run()也运行在
子线程中

```

1 // Thread类（父类）的run方法
2 public void run() {
3     // 如果在子类中没有重写run方法，调用的就是父类中的这个run方法
4     // 如果有Runnable接口子类对象传入的话会调用接口子类对象中实现的run方法
5     // 如果没有传入就什么都不执行
6     if (target != null) {
7         target.run();
8     }
9 }

```

两种的实现方式的比较

1. 方法一实现步骤比较少
2. 方法一存在类的单重继承的局限性
3. 方法二将线程和线程中执行的任务解耦
4. 方法二便于多线程数据的共享(Runnable对象可以只有一个，多个Thread对象（多个线程）可以共享接口子类对象中的普通成员变量）

题目：

```

1     new Thread(new Runnable() {
2         @Override
3         public void run() {
4             System.out.println("Runnable匿名子类的run方法");
5         }
6     ) {
7         @Override
8         public void run() {
9             System.out.println("Thread匿名子类的run方法");
10        }
11    }.start(); // 输出：Thread匿名子类的run方法

```

原理：

1. 建立了一个Thread类的匿名内部类对象
2. 然后向这个对象的构造方法中传入了一个Runnable接口的匿名内部类对象
3. 在Thread的匿名内部类（匿名子类）中，覆盖了Thread类的run方法
4. 当在Thread的匿名内部类对象上调用start方法时，该Thread对象所代表的线程启动
5. start方法会调用Thread类中的run方法，由于是在匿名内部类对象上调用，所以实际上会执行子类中覆盖的方法（多态）（Thread（父类）中的run方法默认是执行传入的Runnable接口子类对象的run方法或什么都不执行）

day 2

多线程数据安全问题

问题：在多线程环境下，多个线程同时访问共享数据时可能会出现不正确的结果（如卖票的多卖与超卖：线程切换导致多个线程同时进入临界区：卖票涉及到：判断当前剩余票数、读取当前票数、票数减1、存回）

问题原因：（前提条件）

1. 多线程环境
2. 多线程共享数据
3. 对共享数据的非原子操作

原子操作：要么都执行 要么都不执行（执行过程不能被打断）

解决：打破前提条件之一即可，但只能打破第3个条件（需求决定）：把对共享变量的一组操作变为原子操作

方法：把一组操作变成原子操作

逻辑上

1. 在一个线程对共享变量的一组操作中禁止发生线程切换（不可行，Java是抢占式调度，线程切换由系统决定）
2. 给共享变量加锁，从而保证
 1. 只有加锁的线程可以访问到共享变量
 2. 加锁线程在没有完成对共享变量的一组操作之前，不会释放锁
 3. 只要不释放锁，其他线程就算得到调度执行也无法访问该共享变量

语言层面

同步代码块（对线程做了线程同步，某个时刻只能有一个代码块执行代码块中的代码）

语法

```
synchronized(锁对象) {  
    需要同步的代码块 ( 对象共享变量的一组操作 )  
}
```

锁对象可以是任何Java中的对象

synchronized + 锁对象

一次只能有一个线程执行同步代码块中的代码

同步代码块的细节 (加锁解锁过程) (加锁和释放锁都是由JVM隐式完成 : 通过设置对象中的标志位)

1. java语言中所有的对象都可以作为锁对象 , 因为所有的对象中 , 都有一个标志位 , 这个标志位就是用来表示 , 加锁和解锁两种锁的状态
2. 执行synchronized代码块前 , JVM会尝试在当前线程中设置锁对象的标志位 , 从而对锁对象加锁
 1. 如果此时锁对象处于未加锁状态 , JVM就会设置锁对象的标志位 (加锁) , 并在锁对象中记录是哪个线程加的锁 , 然后让加锁成功的当前线程执行同步代码块中的代码
 2. **如果此时锁对象已经被加上锁 , 且当前线程不是加锁线程 , 系统会让当前线程处于阻塞状态 , 直到加锁线程执行完了对共享变量的一组操作 , 锁被释放 (就会卡在这里 , 不会继续向下执行)**
3. 成功获取锁的线程在执行完synchronized代码块后 , JVM会自动重置锁标志位 , 将锁对象变成未上锁状态

注意事项 :

锁对象可以是任何对象 , 但是如果要控制多个同步代码块对同一个共享变量的访问的话必须**使用同一个锁对象**

同步代码块相关概念

同步和异步

同步 : 你走我不走 , 你不走我走 (线程不能同时进入临界区 : 互斥)

异步 : 你走你的 , 我走我的 (多线程天生如此)

同步代码块 : 通过线程同步 , 构造原子操作 , 解决了线程安全问题

同步的好处与坏处 :

1. 解决了线程安全问题
2. 相比较于异步 , 会出现等待锁释放而造成的线程的阻塞 , 从而降低了程序运行效率

同步方法 (被synchronized修饰的方法 : 一种特殊形式的同步代码块)

同步方法的锁对象 : 当前对象this (在哪个对象上调用同步方法 , 哪个对象就是同步方法的锁对象)

静态方法也可以是同步方法:锁对象是静态方法所属的类的Class对象

Lock锁对象

Lock只是个接口，使用的是其实现类ReentrantLock

实现同步代码块：Lock锁（除synchronized外）

锁对象的比较

Lock锁对象：加锁和释放锁都是由Lock锁对象调用自己的方法完成 lock() unlock()

synchronized代码块中的锁对象（仅有表示锁状态的标志位）：加锁和释放锁都是由JVM隐式完成
都可以完成构造同步代码块的工作（实现线程同步）

Lock锁对象.lock()

需要同步的代码块

Lock锁对象.unlock()

两种加锁方式的比较（Lock锁对象要自己用代码确保锁的释放）

```
1 // jdk文档中说明的lock对象的使用方式
2 Lock l = ...
3 l.lock();
4 try {
5     // access the resource protected by this lock
6 } finally {
7     l.unlock();
8 }
```

推荐使用synchronized代码块（jdk1.8开始效率就相差无几了）

java.util.concurrent（多线程进阶必看的包）

死锁

产生场景：同步的一个弊端：嵌套锁可能会导致死锁（一个是外LockA，内LockB，另一个是外LockB，内LockA）

概念：两个或两个以上的线程在执行过程中，因为竞争资源而产生的一种相互等待的现象

如何解决：

1. 让线程获得锁的顺序都相同
2. 将获取多把锁的过程变成一个原子操作（在最外面加锁：要么获取多把锁，要么一把锁都不获取）

day 3

生产者消费者模型

问题描述: 需要解决的问题：**线程同步**（保证对缓冲区的互斥访问）和**线程间的协作**（生产后有产品了消费，消费后有空位了生产）

1. 有多个生产者和多个消费者，和一个容量为n的缓冲区

生产者负责生产产品放入缓冲区，消费者负责从缓冲区中取产品出来消费

2. 多个生产者和多个消费者各自都是以异步的方式运行

3. 约束条件

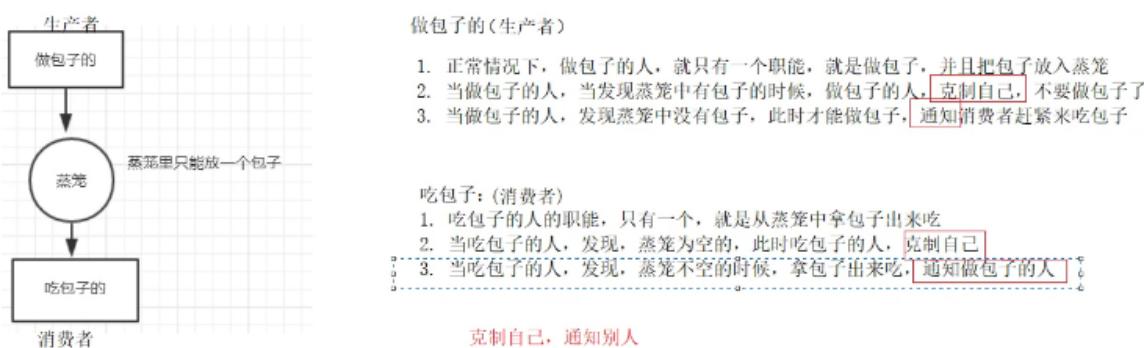
当缓冲区空的时候，不允许消费者到缓冲区中取数据(消费者线程阻塞，同时释放锁)

当缓冲区满的时候，不允许生产者向缓冲区中放入数据（生产线程阻塞，同时释放锁）

同时缓冲区中的一个单元，只能放入一个产品

简化版：缓冲区容量为1的生产者消费者问题，即做包子和卖包子问题

逻辑



关键动作：克制自己，通知别人

API

Object类中wait()和notify()都是在**当前线程持有的锁对象**（任意Java对象）上调用的

wait() (克制自己)

public final void wait()

作用：在哪个线程上调用了它自己持有的锁对象.wait()方法，就会导致哪个线程阻塞，同时放弃持有锁对象

注意事项：

1. 调用wait的条件：当前线程必须持有锁对象，才能调用锁对象的wait()方法
2. 调用时发生什么：wait()方法执行后，线程放弃对锁对象的持有并进入阻塞状态
3. 唤醒条件：直到在其他线程中调用了同一个锁对象的notify/notifyAll方法，该线程才会被唤醒（唤醒后需要得到调度执行，然后重新争夺锁对象：尝试给锁对象加锁）（notify是有可能，notifyAll

是一定会被唤醒)

notify() (通知别人)

唤醒在该锁对象上等待的任意一个线程 (其他线程不会立即得到执行 , 其他线程要等到当前线程释放锁之后才能竞争得到该锁对象)

notifyAll() (在有多个生产者和消费者的场景下使用)

唤醒在该锁对象上等待的所有线程

代码实现 (需要完成两个任务 : 线程同步和生产者与消费者的协作)

第一个版本 (把主要功能放在生产任务和消费任务这两个类中实现)

第二个版本 (主要功能放在蒸笼这个类中实现)

当有多个生产者与消费者的时候

需要用notifyAll() 来保证生产后必有消费 , 消费后必有生产 , 避免出现所有线程都阻塞的情况

当有多个生产者和消费者的时候 , 此时注意 , 我们一定要使用notifyAll通知别人 , 为什么 ?

假设现在 , 其他3个线程都 因为wait方法处于阻塞状态 , 假设现在只有 做包子的1线程 在执行 ,
假设现在蒸笼里没有包子

1. 做包子的1线程 , 看了下蒸笼为空 , 做包子 , 通知别人 , 假设用notify选择的是做包子的2线程来执行
2. 做包子的线程2 , 被唤醒 , 执行 , 看了下有包子 , 做包子的线程2阻塞自己(wait)
3. 现在只有做包子的线程1 , 开始执行 , 一看 , 有包子 , 阻止自己

虚假唤醒 : notifyAll()可能唤醒了在逻辑上不应该唤醒的线程 , 这些线程即便是竞争得到锁对象后 , 又会因为不满足执行条件而重新进入阻塞状态 (生产者唤醒了另一个生产者)

面试题

sleep() vs wait()

1. 所属不同

sleep定义在Thread类 , 静态方法

wait定义在 Object类中 , 非静态方法

2. 唤醒条件不同

sleep方法是休眠时间到

在其他线程中的同一个锁对象上 , 调用了notify或notifyAll方法

3. 使用条件不同 :

sleep 没有任何前提条件

wait()要求当前线程必须持有锁对象 , 才能在锁对象上调用wait()

4. 导致线程进入阻塞状态的时候 , 线程对锁对象的持有情况不同 (最主要最核心)

线程因为sleep()方法而进入阻塞状态的时候 , 不会放弃对锁对象的持有

但是线程因为wait()方法而进入阻塞状态的时候 , 会放弃对锁对象的持有

线程池

day 4

实质

实现两台主机之间的进程与进程的通信

即Java程序，利用计算机网络中的传输层提供的功能，实现底层数据传输，
并基于该底层的数据传输功能，实现Java应用程序的功能

三要素：IP地址 + 端口号 + 传输层协议（套接字 = IP地址 + 端口号）

1. IP

唯一标识网络中的一台计算机

一个InetAddress类对象，表示一个ip地址

通过InetAddress.getByName方法，得到InetAddress对象

2. 端口号

唯一标识某个主机中的进程

ip + 端口号才能唯一确定要通信的目标进程

取值0-65535，但是我们可以用的端口号只能是1024-65535

3. 传输层协议

Java类库中，定义了各种**Socket类(类名都带Socket)**，来抽象传输层的功能，供我们的应用程序使用。

对于TCP和UDP分别定义类不同类型的Socket类，来实现分别基于TCP和UDP协议的传输。

TCP

发送端(客户端)

Socket

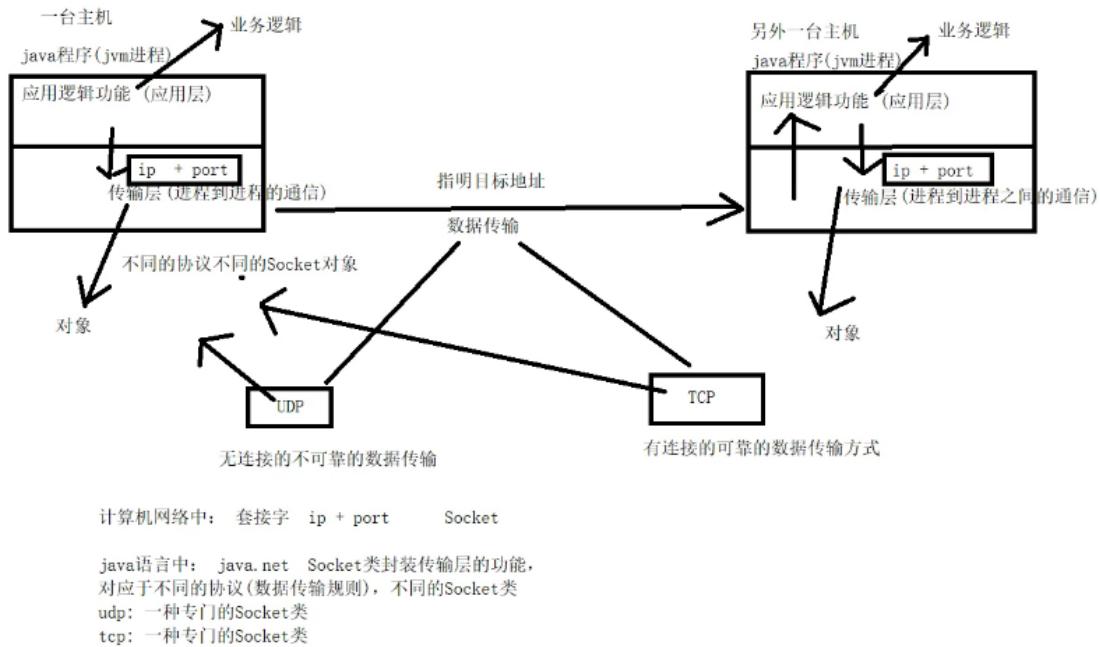
接收端(服务器端)

ServerSocket (监听某个端口) 和 Socket

UDP

发送端/接收端

DatagramSocket



UDP

类似于发短信

发送端 sender

1. 创建UDP的socket对象
2. 将要发送的数据封装成数据包 (packet)
3. 通过UDP的socket对象将数据包发出
4. 释放资源 (释放socket) close()

套接字对象

DatagramSocket (实现基于UDP协议 , 实现进程与进程间的通信)

用来表示发送和接收数据报包的套接字

1. 一个DatagramSocket对象 , 既可以发送数据 , 又可以接收数据
2. DatagramSocket发送和接收的都是数据报包

构造方法：

Datagram(int port)

创建数据报套接字 , 并将其绑定到本地主机的指定端口 (本机IP + 端口号)

数据报包(用于发送的)

用于发送的构造方法 : 需传入要发送的字节数组和目的主机地址和目的端口号 (IP + port)

DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)

表示数据报包 , 用来实现无连接包投递服务 (类似于信封 , 里面封装了要发送的数据 + 收件人地址)

1. buf: 数据报包中封装的实际传输的数据

2. offset : 从buf字节数组从哪个位置开始读取数据
3. length : 要发送的字节数
4. address : 目的主机地址
5. port : 目的端口号

send方法

```
public void send(DatagramPacket p)
```

利用一个socket对象，发送数据报包

注意事项

在接收端还没有启动的时候，基于UDP协议实现的发送端就可以发送数据（不会报错，只是接受端收不到）

证明：UDP是一个无连接的不可靠的传输协议

接收端 receiver

1. 建立UDP的socket对象
2. 创建用于接受数据的数据报包，通过socket对象的receive方法来接收数据
3. 通过数据报包对象的功能来解析接收到的数据
4. 释放资源（释放socket）

套接字对象

```
DatagramSocket(int port)
```

创建数据报套接字，并将其绑定到本机上的指定端口

数据报包（用于接收的）

创建用于接收数据的数据报包

```
DatagramPacket(byte[] buf, int offset, int length)
```

1. buf : 用于保存传入数据的缓冲区
2. offset: 缓冲区的偏移量
3. length : 读取的字节数

receive方法

```
public void receive(DatagramPacket p)
```

从此套接字对象接收数据报包

当方法返回时，DatagramPacket中已经接收到了传输过来的数据与源主机的IP地址和源端口号

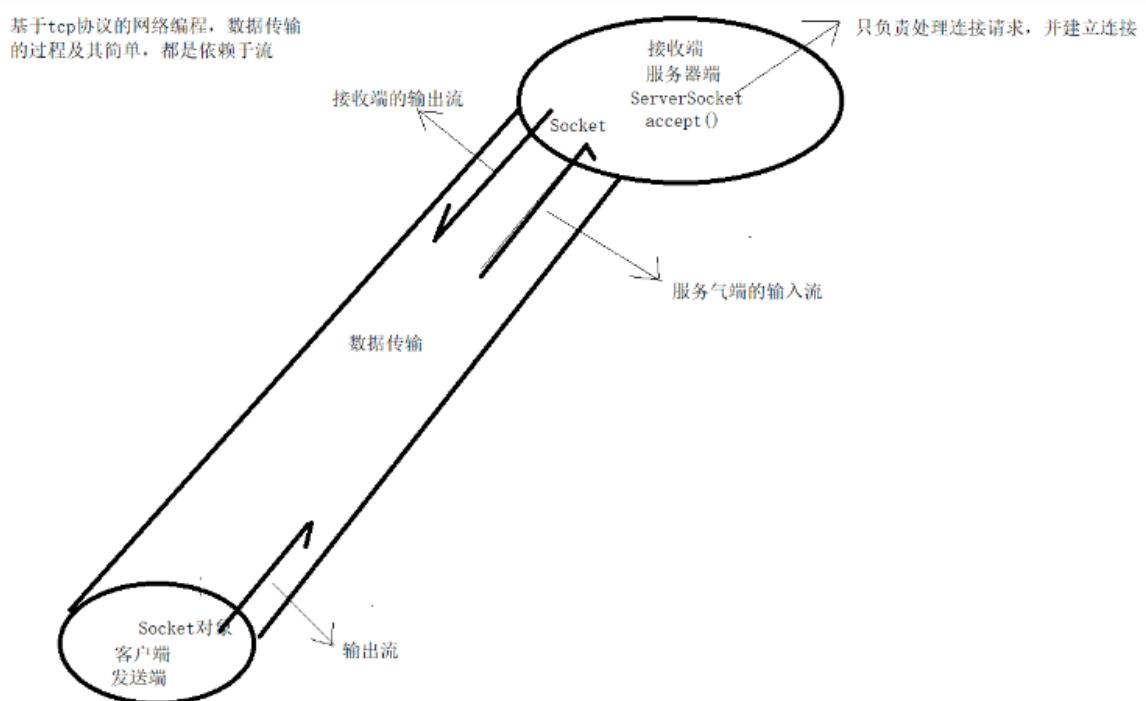
正确顺序：先运行接收端（等待接收数据报包），再运行发送端

注意事项

1. `receive`是一个阻塞方法，当没有接收到数据的时候，该方法会使当前线程阻塞
2. 当两次运行接收端程序的时候：`BindException: Address already in use: Cannot bind`
因为在一台主机上，一个端口号只能给一个进程使用

TCP

基于TCP的socket的数据传输是基于流来实现的



发送端（客户端）

1. 建立客户端的socket对象（确定要连接服务器）
2. 如果对象建立成功，就表明已经建立了数据传输的通道，就可以根据需要通过socket对象进行基于流的输入和输出
3. 向流中读取或写入数据
4. 释放资源(`close()`)

套接字对象

实现客户端套接字

`Socket(String host, int port)`：创建一个流套接字，并将其连接到指定主机上的指定端口号

1. host：要连接的服务器端的IP地址
2. port：要连接的服务器端的端口号

自己的IP地址和端口号：IP地址默认绑定本机的，端口号由系统随机指定（1024-65535）

从socket对象中获取输入或输出流

OutputStream getOutputStream()

在Socket对象上获取输出流，用于发送数据

InputStream getInputStream()

在Socket对象上获取输入流，用于接收数据

注意事项

在TCP的接收端还未运行的时候，运行发送端：Connection refused

连接失败，无法发送数据

接收端（服务器端）

1. 建立ServerSocket对象，在指定端口监听客户端连接请求
2. 收到客户端连接请求后，建立socket连接
3. 如果连接建立成功，就表明已经建立了数据传输的通道，就可以在该通道通过IO进行数据的读取和写入
从socket中根据需要获取输入流和输出流
4. 根据需要向流中写入或读取数据
5. 释放资源

套接字对象

ServerSocket(int port)：创建绑定到特定端口的服务器套接字（默认绑定本机IP）

accept方法

public Socket accept()

侦听接受到此套接字的连接，在接受到客户端的连接请求之前一直阻塞

accept方法在、接收到客户端连接请求后，会创建一个；Socket对象，
该Socket对象和客户端Socket对象建立连接

注意事项：

1. Socket对象中的InputStream的read方法是一个阻塞方法
2. 解决客户端等待服务器端响应。而产生的相互等待的问题

（客户端已经发送完数据并等待服务器端的反馈，而服务器端的字节输入流不知道已经发送完了，
还在等待客户端继续发送数据，这样两端都处于阻塞状态中，所以需要客户端在输出完数据后主动
断开输出流，关闭从c->s这条数据连接通道）

1. 可以通过自定义结束标志解决（不靠谱，不能保证文件内容中不会出现同样的标志符）

- 利用Socket对象的socket.shutdownOutput()方法解决（告知接受方自己已经结束传输，不要再等了）

心得：

- read是阻塞方法，除非是读文件（读到文件末尾会返回-1），否则都需要考虑一下是否存在阻塞问题（没有阻塞可能是因为发送端的socket的关闭了，所以数据传输通道断开了）
- 字节缓冲流可以用，在字节缓冲输出流处刷新即可：bos.flush()
- 与主要逻辑无关的代码要抽成一个独立的方法**

day 5

反射

引入

- 比如说我们现在持有了一个对象：A a = new A();但是在不看A类定义代码的前提下，我们并不知道如何去使用它，因为我们不知道a对象中有哪些成员
- JVM认识我们定义的类的方法：通过加载并解析类对应的字节码文件.class文件（存储在外设中）
- 在JVM类加载过程中，对每个类都会产生一个Class对象（存储在方法区），一个Class对象就包含了一个类定义的完整信息，所以我们可以在程序运行的过程中，通过访问某个类对应的Class对象来获取相应的类定义的信息

反射：在程序运行中，通过访问某个类唯一对应的Class对象，来获取相应的类定义的信息

类定义信息

- 构造方法 -> 创建对象
- 成员变量 -> 在该类型的**任意对象**上访问该成员变量(private也可以)
- 成员方法 -> 在该类型的**任意对象**上调用该成员方法(private也可以)

类加载 -> 获取Class对象 -> 获取表示相应成员的对象

类加载

类加载过程

- 加载

通过一个类的全限定名来获取这个类的二进制字节流(InputStream)，并且在内存（方法区）中生产一个代表这个类的Class对象（作为这个类的各种数据的访问入口）

- 连接

验证：确保被加载类的正确性（针对语义）（因为有人可能不通过编译器，直接用字节码写文件）

准备：为类的静态成员（变量和方法）分配内存，并设默认初值

解析：将类中的符号引用替换为直接饮用（将引用转化为地址）

3. 初始化

给静态成员变量赋初值（代码中显式的），执行静态代码块的内容（类名.class的方式获得Class对象不会触发这个）

类加载时机（什么时候会触发类的加载）

1. 首次创建这个类的对象
2. 首次访问这个类的静态成员
3. 使用反射方式来强制创建某个类或接口对应的Class对象(Class.forName())
4. 首次创建这个类的子类对象，会先加载父类
5. 直接使用 java.exe来运行某个主类：java Hello

类加载器（由谁来做类加载的工作）

把字节码文件以流的形式读取到内存

1. Bootstrap ClassLoader 根类加载器（负责Java核心类的加载，JDK中JRE的lib目录下rt.jar，JVM启动的时候比如lang）
2. Extension ClassLoader 扩展类加载器（负责JRE的扩展目录中jar包（理解为压缩包就好）的加载，在JDK中JRE的lib目录下ext目录）
3. System ClassLoader 系统类加载器（负责加载自己定义的Java类）

获取Class对象

1. 通过对象：对象.getClass()
2. 通过类的字面值常量 类名.class
3. 通过Class类的静态方法

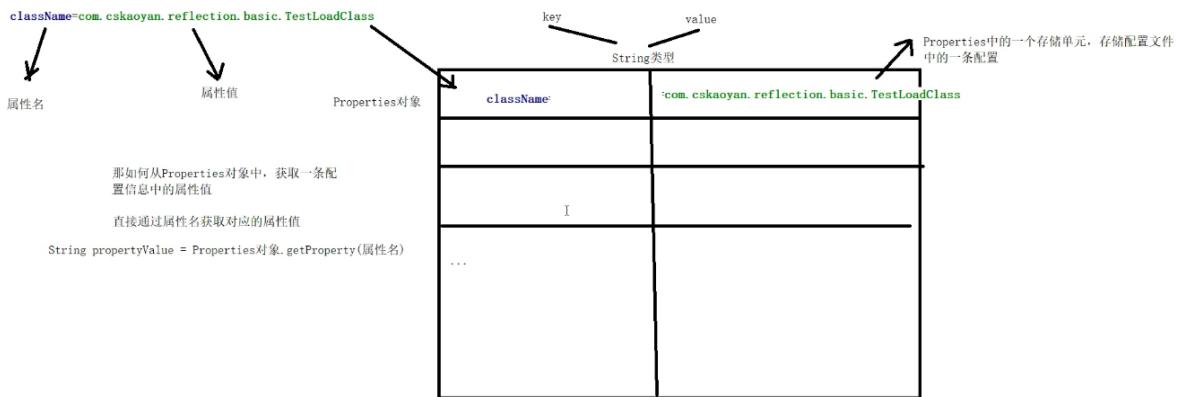
```
public static Class forName(String className)
```

注意事项

1. forName中使用的类名必须是全类名：否则：ClassNotFoundException
2. 类名.class的方法所触发的类加载过程不完整：静态代码块中的内容就没执行
3. 通常使用forName的方法来获取类的Class对象（或完成类加载的过程）：灵活性

Properties对象

```
1 Properties properties = new Properties(); // key 和 value都是String类型
2 properties.load(InputStream in); // 从输入流中读取属性列表（键值对）。
3 String value = properties.getProperty(key);
```



获取类信息

获取构造方法信息

前提：Constructor类描述构造方法，所以一个Constructor对象就对应一个构造方法

获取类中定义的多个构造方法(即Constructor对象)

`Constructor[] getConstructors()`

获取Class对象所代表的类中所有的public的构造方法

`Constructor[] getDeclaredConstructors()`

获取Class对象所代表的类中所有的构造方法

获取指定单个的构造方法

前提：

1. 通过参数列表来指定所求的构造方法
2. 数据类型... 可变参数，该类型对应的参数个数时可变的0,1,2.. 但是必须放在参数列表的最后一个位置
3. Class... parameterTypes 表明所要获取的目标构造方法的形参列表（数据类型.class 基本数据类型也有）

`Constructor getConstructor(Class.. parameterTypes)`

获取Class对象所代表的类中的public的一个构造方法

`Constructor getDeclaredConstructor(Class.. parameterTypes)`

获取Class对象所代表的类中一个任意访问权限的构造方法

通过Constructor对象 创建类的对象

`newInstance(Object... initargs)`

传入初始化参数初始化一个类的对象

获取成员变量信息

Field类：描述成员变量，一个Field对象代表一个成员变量

tips: void setAccessible(boolean flag) 暴力破解权限问题

获取类中定义的多个成员变量

Field[] getFields()

获取当前Class对象所表示的类与其父类中定义的public成员变量

Field[] getDeclaredFields()

获取当前Class对象所表示的类中定义的所有成员变量（不包括父类中的）

获取类中指定的单个成员变量

Field getField(String name)

通过指定的变量名，获取子类和父类中指定名称的public成员变量(Field对象表示)

注意事项

该方法在子类中查找指定名称的public的成员变量时，顺序是先子类后父类，

如果子类和父类中都没有指定名称的成员变量，则抛出NoSuchFieldException

Field getDeclaredField(String name)

获取类中任意权限的指定名称的成员变量（不包括父类中定义的）

使用获取到的成员变量

Object get(Object obj)

Field对象.get(指定对象)

通过Field对象获取该指定对象的成员变量的值

注：虽然返回一个Object对象，但是对于基本数据类型也可以这样

void set(Object obj, Object value)

Field对象.set(指定对象, 新值)

注：基本数据类型也可以传入第二个参数（应该是用到了自动装箱）

获取类中定义的成员方法信息

Method类：描述类中定义的方法

一个Method对象表示类中定义的一个方法

获取类中定义的多个方法

Method[] getMethods()

获取当前Class对象表示的类及其父类中定义的所有public的成员方法

Method[] getDeclaredMethods()

获取当前Class对象表示的类定义的所有权限的成员方法（不包括父类中的）

获取类中的定义的单个方法

如何确定要获取那个方法：通过方法签名：方法名 + 参数列表

Method getMethod(String name, Class... parameterTypes)

获取当前Class对象表示的类及其父类中定义的public的成员方法

Method getDeclaredMethod(String name, Class... parameterTypes)

获取当前Class对象表示的类定义任意访问权限的成员方法（不包括父类）

使用获取到的Method对象

Object invoke(Object obj, Object... args)

在某对象上，调用该方法（并传递实际参数值）

obj 要调用方法的那个对象

args，传递的实际参数

Method对象.invoke(指定对象, 实际参数)

类中定义的静态成员的获取与使用

静态成员不依赖于对象，传入参数那里，对象设置为null即可

静态变量

```
1 public class Demo1 {  
2  
3     public static void main(String[] args) throws NoSuchFieldException,  
4         IllegalAccessException {  
5         Class fieldClass = TestStaticField.class;  
6  
7         // 类中定义的静态变量的获取和普通成员变量没有区别  
8         Field anInt = fieldClass.getDeclaredField("anInt");  
9         System.out.println(anInt);  
10  
11         // 使用
```

```
12 //获取静态变量的值，因为静态变量不依赖于对象
13 anInt.setAccessible(true);
14 int o = (int) anInt.get(null);
15 System.out.println(o);
16
17 }
18 }
19 }
20
21 class TestStaticField {
22     private static int anInt = 100;
23 }
```

静态方法

```
1 public class Demo2 {
2
3     public static void main(String[] args)
4         throws NoSuchMethodException, InvocationTargetException,
5         IllegalAccessException {
5         Class methodClass = TestStaticMethod.class;
6
7         //获取类中定义的静态信息
8         Method staticMehod = methodClass.getDeclaredMethod("staticMehod",
9         boolean.class);
10
11         //使用静态方法，不依赖于对象，对象那里设置为null
12         staticMehod.setAccessible(true);
13         int invoke = (int) staticMehod.invoke(null, true);
14         System.out.println(invoke);
15
16     }
17 }
18 }
19
20 class TestStaticMethod {
21
22
23     private static int staticMehod(boolean b) {
24
25         return 10;
26     }
27
28 }
29 }
30 }
```

Week 6

day 1

注解

为我们在代码中添加信息提供了一种标准化的方法，使我们可以在稍后的某个时刻非常方便地使用这些信息

(注解本身只是一种额外信息，注解的处理和注解本身没有关系)

在代码中添加额外信息

1. 注释

人为的约定，javac在编译时对其视而不见，只有固定语法，没有标准形式，只有人能看懂

2. 注解

为我们在代码中添加信息提供了一种标准化的方法，(额外信息的表示都是通过Java相应的数据类型值)，使我们可以在稍后的某个时刻非常方便地使用这些信息(通过代码来获取(如反射技术)，并使用这些定义的额外信息)

常见注解

1. @Override：检查子类中的方法是否覆盖了父类中的方法
2. @Deprecated：该方法已经过时，官方不推荐使用该方法(存在一定的缺陷)，在后续的JDK中可能删除该方法

注解注意事项

1. 注解本身只是用来传达代码之外的额外信息
2. 注解伴随着的一些效果表现与注解本身无关，是由注解处理器造成的

主要内容

1. 注解的定义(标准形式)
2. 注解的使用(按照定义好的标准形式，给代码添加额外信息：[创建注解实例](#))
3. 注解的处理(定义注解处理器接受并处理这些额外信息)
4. 元注解(注解的注解)

注解的定义(自定义注解)

Java语言中本身只定义了极少数的注解，但我们可以自定义注解

定义注解格式

```
1 public @interface 注解名 {  
2     定义体  
3 }
```

注意事项

1. 注解的定义与类、接口的定义非常像，因为它也表示一种数据类型，它其中定义了：

1. 具体包含多少条信息
2. 每条信息是怎样的（都有标准形式）
2. @必不可少，少了@就成了接口定义了
3. 注解之间不能继承

注解体应该如何定义

```
1 | @interface 注解名 {  
2 |     // 第一条信息的标准形式  
3 |     返回值类型 方法名1();  
4 |     // 返回值类型：该条信息数据类型；方法名：一条信息的名字  
5 | }
```

自定义注解体的说明

1. 注解体的格式类似于接口中的方法定义，但含义完全不同
2. 方法名：一条信息的名字；返回值类型：该条信息数据类型
3. 每一个条注解信息，数据类型只能是以下几种
 1. 所有基本数据类型
 2. String类型
 3. Class类型
 4. 注解类型
 5. 以及以上类型的数组

注解的使用

注意：定义某种类型的注解，实际上也就是定义了某种类型的额外信息的标准形式

注解类型 —— 注解实例（类似于：类——对象）

注解使用的语法

@注解的类型名（属性名1 = 属性值1...）//一个注解实例

这里的属性名：注解中定义的一条信息的名称

注解使用的实质：创建某个类型的注解实例，并给该实例中的每个属性赋值，从而一个注解实例就表示了一个条具体的额外信息

注意事项：必须保证定义中的每个属性都有确定的值

1. 可以在创建注解实例的时候给属性赋值
2. 也可以在定义注解的时候给属性赋默认值

默认值：如果在创建注解实例的时候没有给某个属性显示赋值，如果该属性有默认值的话，属性的值就会自动取默认值

3. 引用类型的数据：默认值不能为null
4. 特殊的简化情况：当属性名称为value，且在使用注解实例时只需要给value属性赋值的时候，可以
直接往括号里面填入属性取值，而不用写出属性名

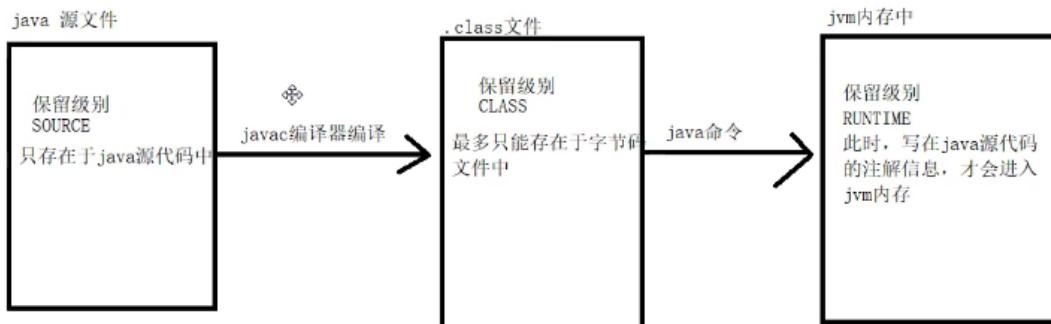
注解的处理

1. 必须使用注解处理器来获取与处理注解信息
2. 注解处理器本身并无特殊语法：通过一些其他的方式（如反射技术），获得所需注解信息，然后根据需求实现特殊功能

元注解

所有写在java源文件中的代码，都会被加载到jvm中（只不过形式不同）
和java源代码对比，注解就不一样了，并非写在java源代码中的注解就一定会进入jvm内存。
如何知道，写在java代码中的注解是否会出现在内存？这就和注解的保留级别有关系了
注解保留级别：用来指定，注解存在的状态

和Java程序的存在状态一致，注解有3种保留级别：
SOURCE：注解将被编译器丢弃（class文件中没有）
CLASS：注解在class文件中可用，但会被JVM丢弃（内存没有）
RUNTIME：JVM在运行时，也会保留注解信息



注解的默认保留级别：CLASS（运行时已经没了）

概念

元注解：注解的注解

常用的元注解：

1. **@Retention**：用来声明注释的保留级别
 1. `RetentionPolicy.SOURCE` 只存在于源代码中
 2. `RetentionPolicy.CLASS` 最多只能存在于字节码文件中
 3. `RetentionPolicy.RUNTIME` JVM在运行时，也会保留注解信息
2. **@Target**：声明和限定注解使用的地方
 1. 整个类或者接口 `ElementType.TYPE`
 2. 成员变量 `ElementType.FIELD`
 3. 构造方法 `ElementType.CONSTRUCTOR`
 4. 成员方法 `ElementType.METHOD`
 5. 可以用类似于数组静态初始化的方式，同时声明注释可以使用在多个地方

注：不管注解实例时添加在哪里的，都有办法去获取它

isAnnotationPresent(Class type) 判断当前对象（如Field对象）上，是否添加了该注解

getAnnotation(Class type) 获取在当前对象上添加的**注解实例**

1. 作用在类上 Class对象
2. 构造方法 Constructor对象
3. 成员变量 Field对象
4. 成员方法 Method对象

注：因为 StudentFactory运用到了反射技术

所以需要在运行时使用注解，所以注解的保留级别设置为RUNTIME

注解和配置文件

配置文件：

优点：可配置，不用改源码

缺点：不直观，开发效率低

注解：

优点：直观，开发效率高

缺点：硬编码，修改后需要重新编译运行

结论：

1. 和代码直接相关的配置：用注解
 2. 和代码运行环境相关的配置：用配置文件
-

GC

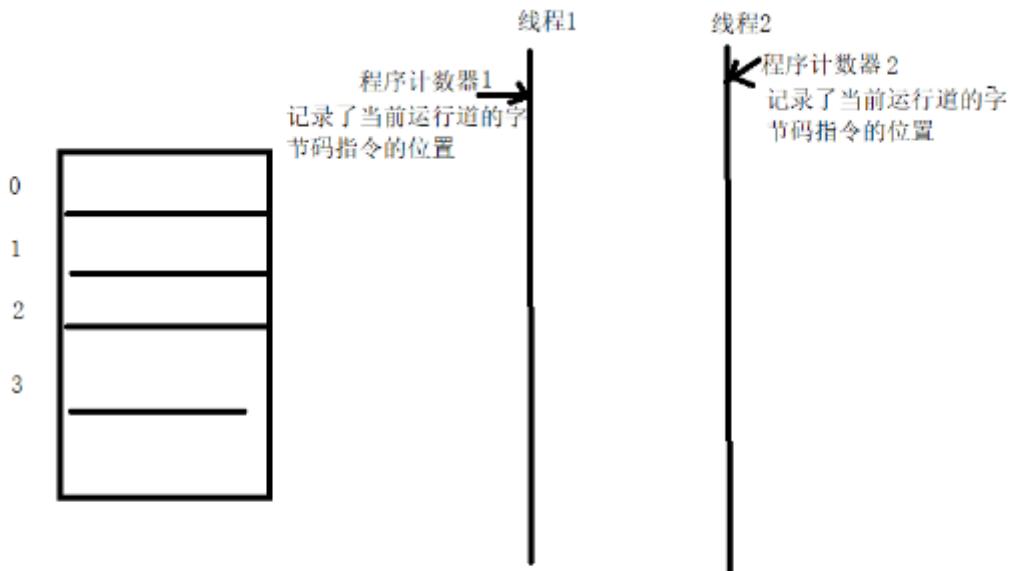
JVM运行时内存区域（从线程角度）

线程私有：程序计数器、Java虚拟机栈、本地方法栈（线程私有的地方不会产生操作共享变量带来的问题）

线程共享：堆、方法区

程序计数器：线程私有（线程隔离）

当前线程所执行的字节码的行号指示器

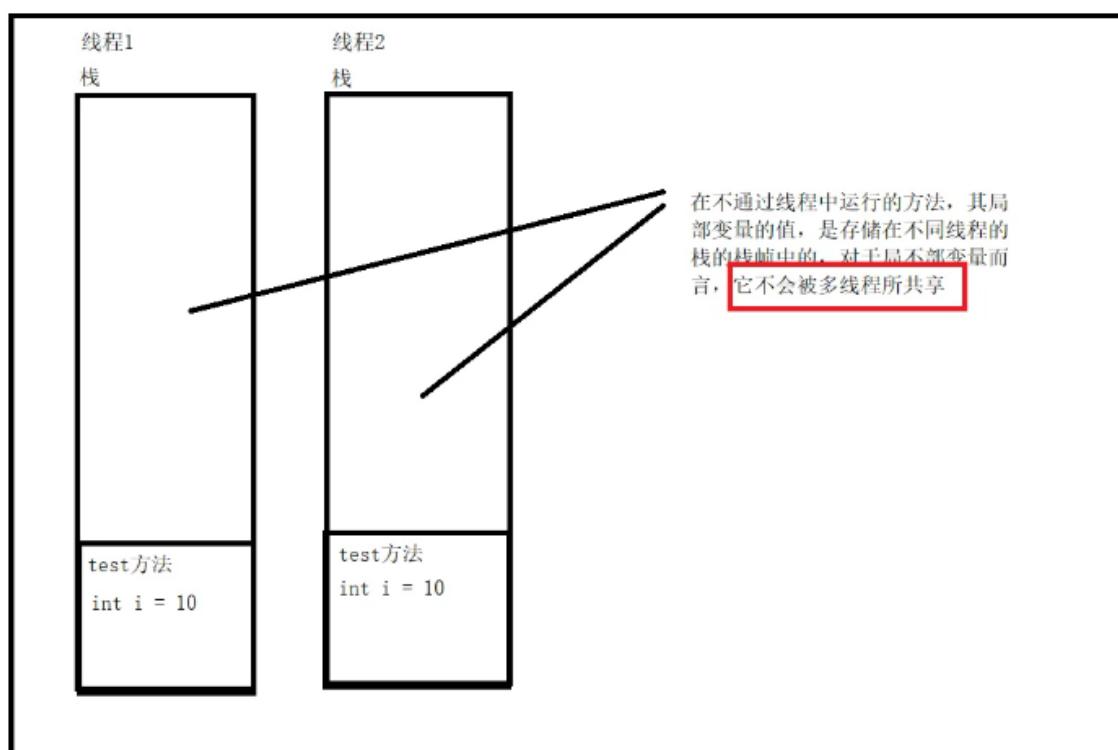


这意味着，每个线程都有自己的程序计数器，即程序计数器这块内存，是线程私有的内存空间

虚拟机栈：线程私有

它描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。线程私有(线程隔离)

栈内存，是线程私有的内存空间
方法，在哪个线程中被调用，本次方法运行所需的栈帧就在哪个线程对应的栈上分配



本地方法栈：线程私有

堆：多线程共享

此内存区域的唯一目的就是存放对象，一个JVM实例只存在一个堆，堆内存的大小是可以调节的。
堆内存是线程共享的。

方法区：多线程共享

方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量等数据。

JVM内存管理的方式

显示内存管理 (C/C++)

内存的申请和释放是程序开发者的职责

常见问题：

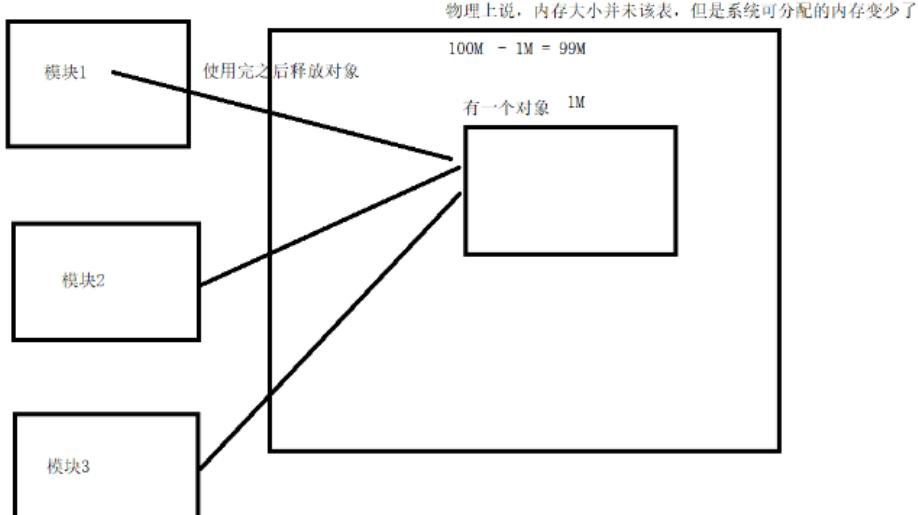
1. 内存泄漏：申请了内存空间，使用完毕后未主动释放（导致可用的内存空间变少）
2. 野指针：使用了一个指针，但该指针指向的内存空间已经被释放掉了（就是可能指向的地方现在装的是其他的东西，而不是原来的那个对象了）

显式的内存管理 (C/C++)

内存管理(内存的申请和释放)是 程序开发者 的职责
语言层面，提供api，供开发者管理内存的功能：
1. malloc 向系统申请指定大小的内存
2. free 释放一片内存空间

显示内存管理的常见问题：

野指针：使用了一个指针，但是该指针指向的内存空间已经被free
内存泄漏问题：内存空间已经申请，使用完毕后未主动释放



隐式内存管理(java/C#)

内存的回收由垃圾回收期自动管理

优点：增加了程序的可靠性（没有了野指针），减少了memory leak

缺点：无法控制GC的时间，GC会影响系统性能

垃圾回收的三个问题

1. 如何判断垃圾
2. 如何回收垃圾
3. 回收垃圾的时机

如何判断垃圾

1. 引用计数法

给对象添加一个引用计数器（Int型变量），每当有一个引用变量指向该对象的时候，计数器+1，引用失效时，计数器-1，当计数器的数值为0时，也就是没有引用变量指向该对象时，该对象就无法被我们访问到，自然就成为了垃圾

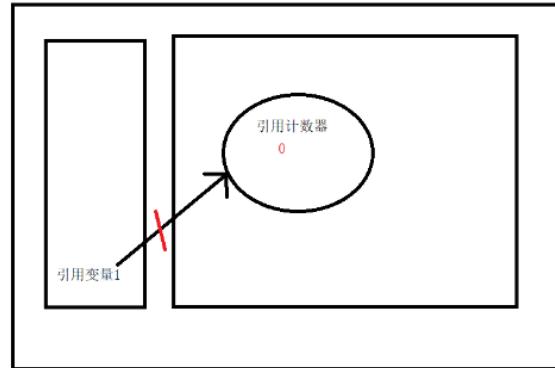
如何判定一个对象成为垃圾，其核心依据是 → 看一个对象是否还可以访问到，如果一对象再也访问不到了，就认为这个对象变成了垃圾



什么情况下，一个对象就再也无法被访问到了呢？引用变量就是我们手中的遥控器，当没有任何一个有效的引用变量指向一个对象的时候，该对象我们就再也访问不到，从而变成垃圾

确定哪些对象已经变成了垃圾，最简单的算法就是引用计数法

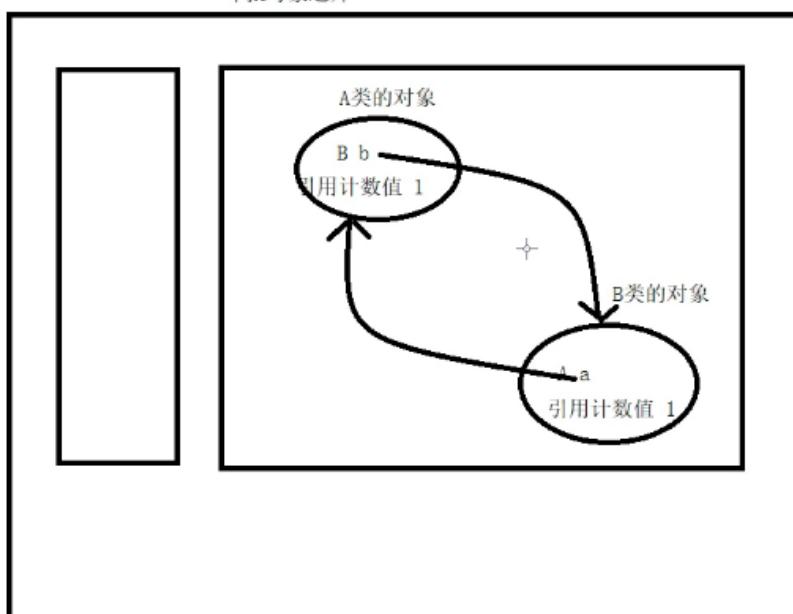
1. 给对象添加一个引用计数器
2. 每当一个地方引用它时，计数器加1
3. 每当引用失效时，计数器减少1
4. 当计数器的数值为0时，也就是对象无法被引用时，表明对象不可在使用



引用计数法弊端：循环引用

除了A类型对象中的引用b执行B对象，且B类型对象的引用a指向A对象之外

当对象之间存在循环引用的时候，引用计数法是无法正确判定垃圾对象的



2. 根搜索算法 (实际使用的)

以GC Roots这个集合出发，遍历搜索所有可达的对象，不可达的对象被标记为垃圾

为了克服，引用计数法无法正确应对的循环引用问题所以出现了另外一种判断垃圾对象的算法 ——> 根搜索算法

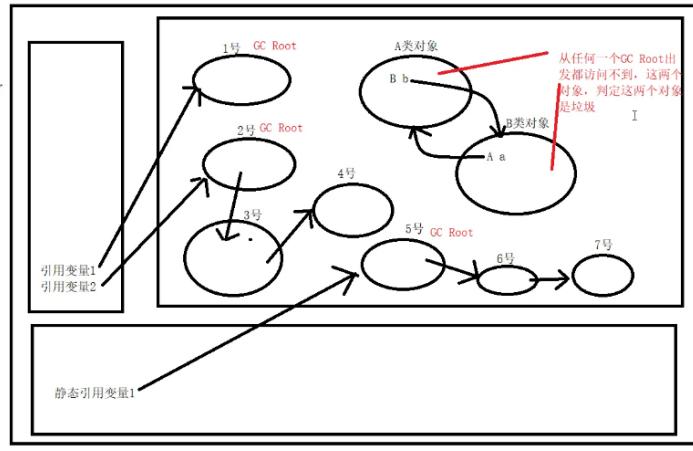
根搜索算法

1. 这个算法的基本思想是将一系列称为“GC Roots”的对象作为起始点
2. 从这些节点开始向下搜索
3. 搜索所走的路径称为引用链
4. 当一个对象到所有的GC root之间没有任何引用链相连，此时，就认为该对象变成了垃圾

GC Roots包含对象呢？

1. 虚拟机栈中引用的对象
2. 方法区中的静态属性引用的对象
3. 静态引用类型的成员变量，类名的方式一定能够访问到

1. 这两种类型的引用，其实都是我们可以直接访问到的引用。
2. 而这两种引用指向的对象，是我们一定可以访问到的对象



如何回收垃圾

1. 标记清除算法

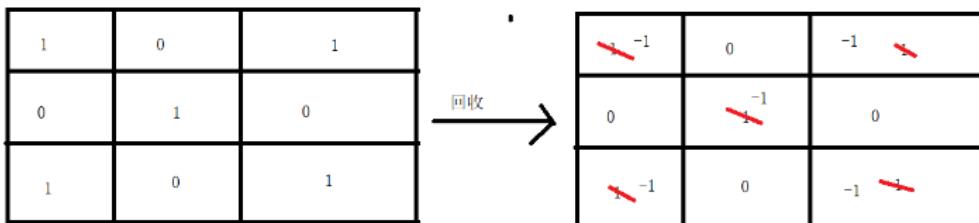
以0表示存活对象

以1表示垃圾对象

以-1表示可以使用的空闲空间

回收之后，总共有5片空闲内存的

但是假设，如果我们要给一个对象分配一个包含3片连续内存的内存空间

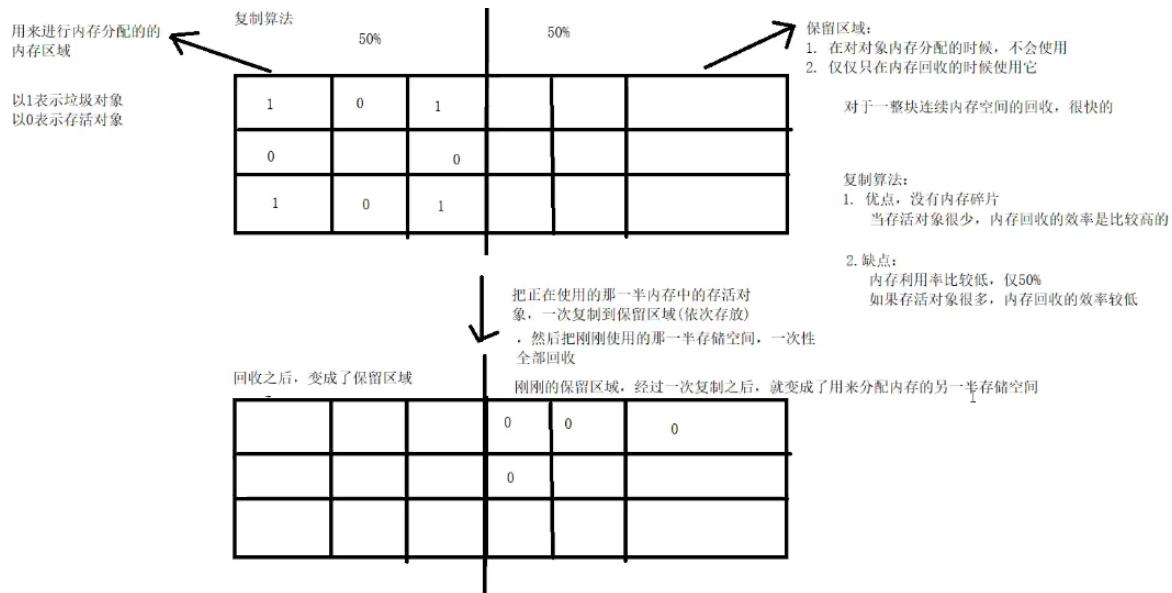


标记清除算法：

优点：回收很简单

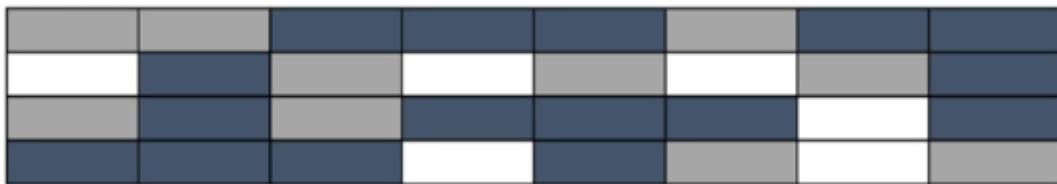
缺点：容易导致内存碎片

2. 标记复制算法

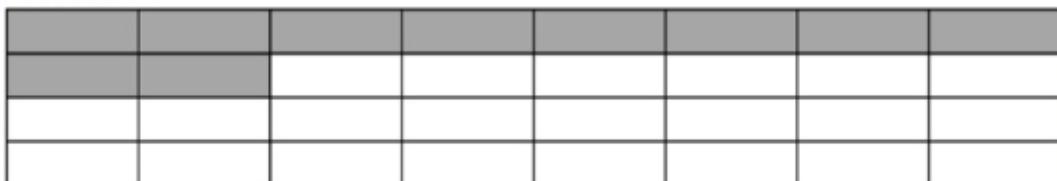


3. 标记整理算法

回收前状态：



回收后状态：

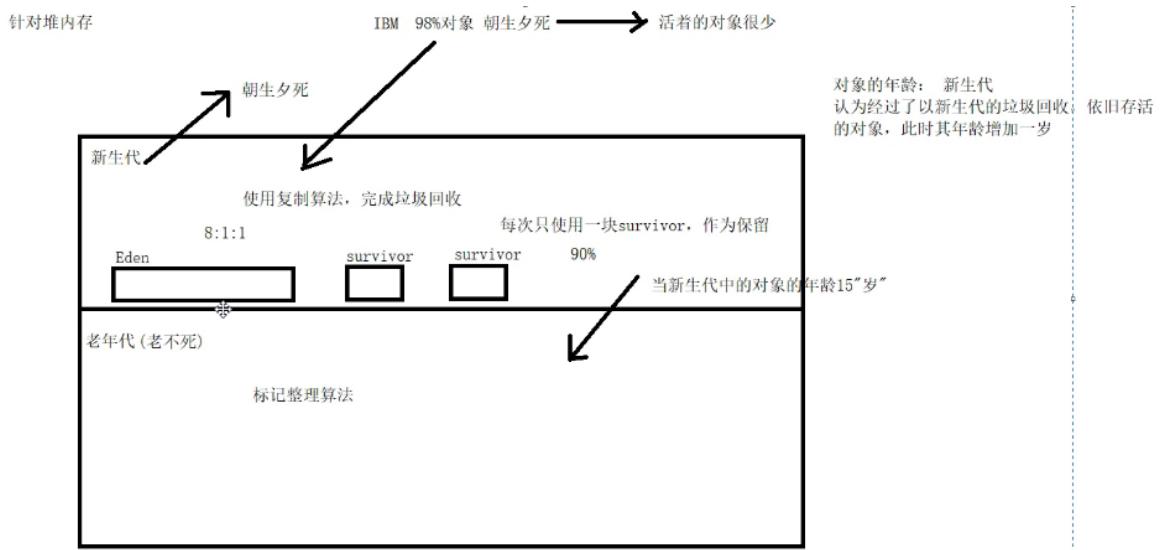


存活对象

可回收

未使用

4. 分代收集算法（实际使用的）



何时触发垃圾回收

1. 申请堆空间失败（内存溢出）后会触发GC回收
2. 系统进入空闲一段时间后
3. 主动调用GC（不推荐）

GC相关概念

1. Shallow size：对象本身占用的内存大小（对象头（里面有用于锁对象的标志位等）+ 成员变量）
2. Retained size：该对象自己的shallow size + 只能由该对象直接或间接访问到的对象的shallow size之和
 表明该对象被GC后所能回收的内存的总和（因为该对象被回收后，只能通过它来访问的对象也就成了垃圾了，也要回收）

内存相关问题

Out of Memory 内存溢出
Heap OOM 堆溢出
Stack Overflow 栈溢出

内存泄露（memory leak）：已分配的堆内存未及时释放，造成可用内存的减少（原来可用100M，申请了20M一直未释放，所以现在可用的堆内存就只有80M）

内存溢出（申请堆空间失败，没有可分配的内存）：内存泄露有可能导致内存溢出，但不是一定会导致内存溢出

装箱和拆箱

一个包装类对象：维护了一个对应基本数据类型的数据的值

- byte ——> Byte
- short ——> Short
- int ——> Integer
- long ——> Long
- char ——> Character
- float ——> Float
- double ——> Double
- boolean ——> Boolean

```
Integer integer = new Integer(1000);
```

自动装箱

```
1 int a = 100;
2 // 装箱 把一个基本数据类型的值 -> 封装到了其包装类的对象
3 // 1. 对基本类型的数据，创建一个包装类对象，并且将其对应的基本数据类型的值，封装到该
对对象中
4 // 2. 把自动创建的包装这个int值的Integer对象，并返回其引用
5
6 //自动装箱
7 Integer aInteger = a; // Integer aInteger = new Integer(a)
8 System.out.println(aInteger.intValue());
```

自动拆箱

```
1 Integer integerValue = new Integer(1000);
2
3 // 自动拆箱
4 int c = integerValue; // int c = integerValue.intValue(); 自动调用了该方法
```

有些场景下基本数据类型的变量和它们对应的包装类对象是不等价的

```
1. 1 int[] arr = new int[10];
2 // 这里就不成立，一个是整型数组，一个是对象数组，数据类型不同，不能赋值
3 //Integer[] arr1 = arr;
```

```
2. 1 class Father {
2
3     int test() {
4         return 0;
5     }
6
7 }
8
9 class Son extends Father {
10    // 不是同种数据类型，所以不能覆盖
11    // 在这种场景下，包装类和其对应的基本数据类型，不等价
12    //Integer test() {
13    //     return 0;
14    //}
15
16 }
```